# *Pokročilé dátové štruktúry*

Kubo Kováč

# Obsah

# Úvod

# Časť I

# Úvod

# Kapitola 1

# Dve chuťovky

Na úvod sa zoznámime s dvoma „chuťovkami“, ktoré nám poslúžia ako vstupná brána do sveta dátových štruktúr – transpozíciou matice a štruktúrou Union-Find. Prvý problém, transpozícia matice, sa na prvý pohľad môže javiť ako triviálny. Veď čo už by sa na ňom dalo pokaziť alebo vylepšiť? Ukážeme si, že tento zdanlivo jednoduchý problém skrýva zaujímavé prekvapenia, ktoré vyjdú na povrch najmä pri praktickej implementácii.

S druhým problémom preskočíme do teoretickej roviny. Predstavíme si elegantnú a veľmi jednoduchú dátovú štruktúru s názvom Union-Find (alebo aj disjunktné množiny). Hoci jej implementácia zaberá zopár riadkov zdrojového kódu, analýza časovej zložitosti patrí medzi zložitejšie kapitoly z dátových štruktúr. Ukážeme si úplne nový prístup, ako možno nazerať na analýzu časovej zložitosti.

## 1.1 Transpozícia

Majme štvorcovú maticu rozmeru $N \times N$ a našou úlohou je transponovať ju, teda „preklopiť“ podľa hlavnej diagonály. Výsledkom transpozície je nová matica, v ktorej sa prvky na pozíciách $(i, j)$ a $(j, i)$ navzájom vymenia. Inak povedané, riadky sa stanú stĺpcami a stĺpce riadkami.

Otázka znie: ako na to?

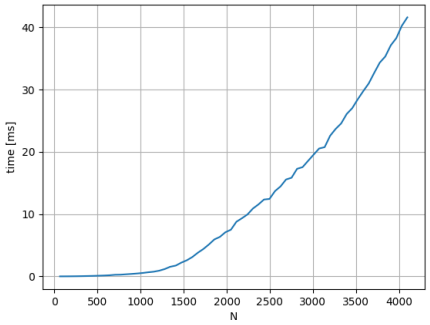
Tu je priamočiare riešenie:

Časová zložitosť je zjavne $\Theta(N^2)$, takže graf bude vyzerať nejak takto, však? (Však?)

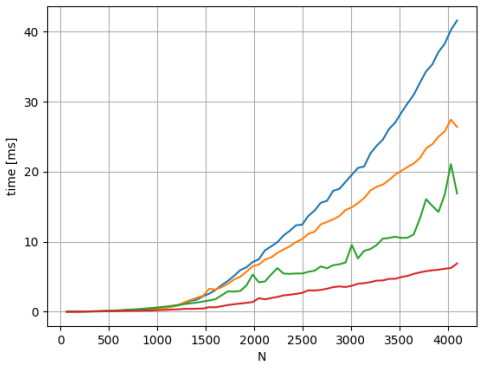
Taktiež je zjavné, že musíme premiesniť $\Theta(N^2)$ prvkov. Akýkoľvek algoritmus musí spraviť $\Theta(N^2)$ čítaní a zápisov do pamäte, takže lepšie to nejde.

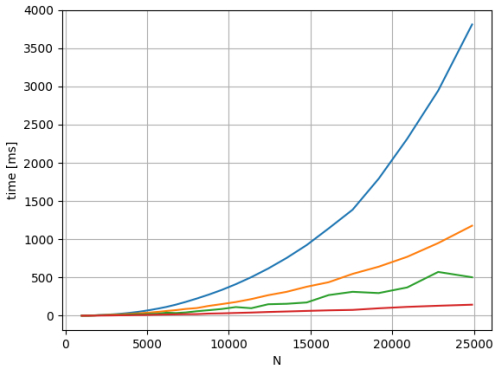
```
const int N = 1000;
int A[N][N];

void transpose() {
  for (int i=0; i<N; ++i)
    for (int j=i+1; j<N; ++j)
      swap(A[i][j], A[j][i]);
}
```




**Šok #1: existujú lepšie algoritmy:**

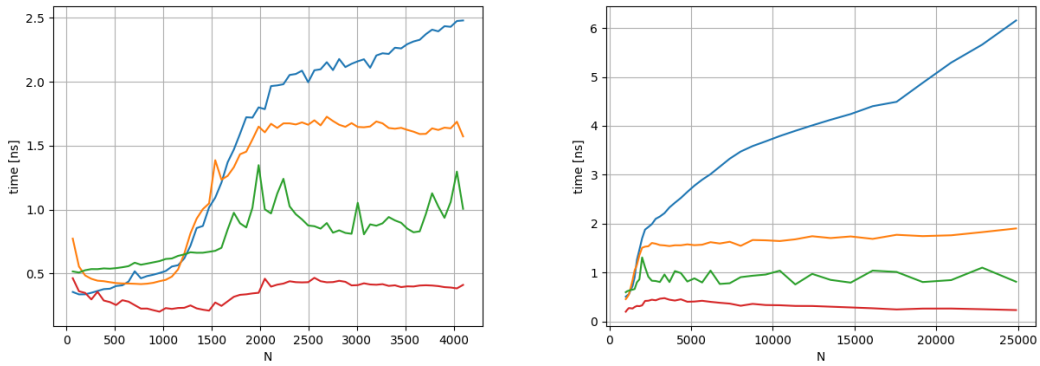



Vľavo je graf pre stredne veľké $N \leq 4\,000$, vpravo je ten istý graf pre veľké $N$ až po 25 000. Čas je v milisekundách. Náš kód je modrá čiara – teda to najpomalšie riešenie. Ako sa to dá lepšie?

Mimochodom, tvrdili sme, že časová zložitosť nášho riešenia rastie ako druhá mocnina $N$. Graf naozaj pripomína parabolu, ale najlepší spôsob ako sa o tom presvedčiť je, že do grafu nanesieme „čas deleno $N^2$“, a vyjde nám konštanta. (Však?)

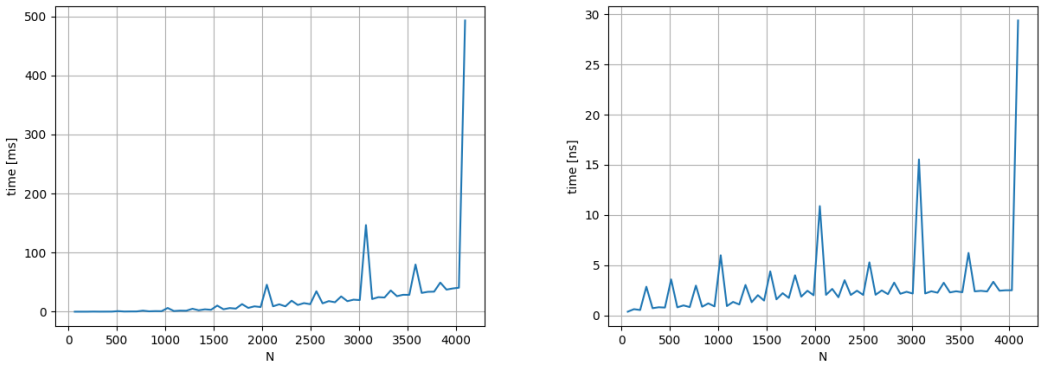
**Šok #2: Čas deleno $N^2$ nie je konštanta?**




Tu vidíme čas deleno $N^2$, čo môžeme interpretovať ako čas v prepočte na jeden prvok matice, v nanosekundách.

Prečo to stúpa? Prečo to nie je konštanta??

**Šok #3: Klamal som. V skutočnosti graf pre náš kód vyzerá takto:**




Vľavo celkový čas v milisekundách, vpravo delené $N^2$ v nanosekundách.

Čo majú, do čerta, znamenať tie zuby??? Čo sú tie špicaté výkyvy v čase behu?

(Nie, nie je to chyba merania.)

Prvým zlepšením základného Union-Find algoritmu je technika nazývaná Union by Rank. Jej cieľom je udržiavať stromy čo najplytšie, aby operácia find(x) bola rýchla.

Zavedieme pojem rank pre každý vrchol, ktorý slúži ako odhad výšky stromu. Na začiatku má každý vrchol rank rovný nule — keďže každý je v samostatnej jednoprvkovej množine (strom výšky 0).

Operácia union(x, y) potom funguje nasledovne:

Najprv nájdeme korene stromov, do ktorých patria x a y.

Potom pripojíme strom s menším rankom pod strom s väčším rankom.

Ak majú oba stromy rovnaký rank, jeden z nich (ľubovoľný) sa stane koreňom a jeho rank sa zvýši o 1.

Dúfam, že vás tieto výsledky aspoň trochu prekvapili – a možno aj zmiatli. To bol cieľ. Zatiaľ vám odpovede neprezradím a nechám vás trochu sa potrápiť s vlastnými hypotézami. Prejdime teraz k druhej „chuťovke“ a na záver sa k transpozícii vrátime a uzavrieme celú kapitolu s lepším pochopením toho, čo sa vlastne deje „pod kapotou“.

## 1.2 Union-find

Chceme si udržiavať *neorientovaný graf*, ktorý sa dynamicky vyvíja – postupne doň *pridávame hrany* a zároveň chceme *rýchlo zisťovať súvislosť medzi vrcholmi*. Chceli by sme teda dátovú štruktúru, ktorá podporuje nasledujúce operácie:

- *join*$(x, y)$: prepojí vrcholy $x$ a $y$ hranou,
- *connected*$(x, y)$: zistí, či medzi vrcholmi $x$ a $y$ existuje cesta.

Takáto funkcionalita je napríklad presne to, čo potrebujeme v Kruskalovom algoritme na hľadanie najlacnejšej kostry v grafe (problému minimálnej kostry sa ešte budeme viac venovať v časti o haldách) alebo pri unifikácii.

Každú *množinu reprezentujeme ako strom*, pričom *koreň stromu slúži ako reprezentant celej množiny*. Všetky prvky v množine „ukazujú smerom nahor“ – teda smerom ku koreňu.

Operácie potom vyzerajú nasledovne:

- *find*$(x)$: Prechádzame od vrcholu $x$ nahor, až kým nenájde koreň, teda reprezentanta množiny obsahujúcej prvok $x$.
- *union*$(x, y)$: Najprv nájdeme korene stromov, v ktorých sa nachádzajú $x$ a $y$. Ak ide o ten istý strom (teda už patria do tej istej množiny), neurobíme nič. Inak napojíme jeden strom pod druhý, konkrétne jeden koreň nastavíme ako syna toho druhého.

Toto riešenie je veľmi jednoduché, má však jednu zásadnú nevýhodu: Operácia find môže trvať čas úmerný výške stromu – a keďže bez ďalších optimalizácií sa môžu stromy „natiahnuť“ do jednej dlhej reťaze, v najhoršom prípade môže byť časová zložitosť až lineárna.

**Vylepšenie #1: Union by Rank.** Prvým vylepšením základného algoritmu je technika nazývaná *Union by Rank*. Cieľom je udržiavať stromy čo najplytšie, aby bola operácia find(x) rýchla.

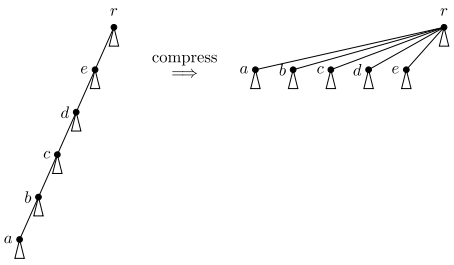
Zavedieme pojem rank pre každý vrchol, ktorý slúži ako odhad výšky stromu. Na začiatku má každý vrchol rank rovný nule – keďže každý je v samostatnej

jednoprvkovej množine (strom výšky 0). Pri operácia union(x, y) potom vždy pripojíme strom s menším rankom pod strom s väčším rankom. Ak majú oba stromy rovnaký rank, jeden z nich (ľubovoľný) sa stane koreňom a jeho rank sa zvýši o 1.

Indukciou ľahko dokážeme, že:

- strom s rankom $r$ má výšku najviac $r$
- strom s rankom $r$ má aspoň $2^r$ vrcholov
- teda rank $r$ môže mať najviac $n/2^r$ vrcholov
- z toho vyplýva, že maximálny rank je $\lg n$, čo je zároveň maximálna hĺbka vrcholov a teda časová zložitosť klesne na $O(\lg n)$

**Vylepšenie #2: Path Compression.** Druhé vylepšenie, ktoré dramaticky znižuje časovú zložitosť operácií, sa nazýva path compression, teda „stlačenie cesty“. Princíp je jednoduchý: *Po tom ako počas operácie find(x) nájdeme koreň, sa vrátime naspäť a všetky vrcholy, ktoré sme navštívili po ceste, napojíme priamo pod tento koreň.* Týmto spôsobom sa výrazne zníži hĺbka stromu pre všetky budúce volania *find* nielen pre $x$, ale aj pre skoro všetky vrcholy na ceste ku koreňu a vrcholy v ich podstromoch.




Na prvé počutie to môže znieť ako komplikovaný algoritmus, avšak jeho implementácia je veľmi jednoduchá. Ako prvé si stačí uvedomiť, že na rozdiel od iných štruktúr (napr. vyhľadávacie stromy alebo písmenkové stromy), kde si potrebujeme pre každý vrchol pamätať všetkých jeho synov, v union-finde prechádzame stromy výlučne smerom nahor, takže si stačí pamätať pre každý vrchol smerník na jeho otca. Dokonca môžeme vrcholy očíslovať $1, 2, 3, \ldots, n$ a v obyčajnom poli si zapamätať pre každý vrchol $x$ jeho otca $p[x]$ (pre korene si môžeme poznačiť napr. $p[x] = 0$).

Okrem poľa otcov si ešte potrebujeme pamätať ranky. Mohli by sme si ich pamätať v samostatnom poli, avšak keď si uvedomíme, že ranky potrebujeme poznať iba pre korene (a korene nemajú otca), môžeme informáciu o rankoch napchať do toho istého poľa. Použijeme nasledovný trik: pre vrcholy, ktoré nie sú koreň, bude $p[x] > 0$ číslo ich rodiča; pre korene si v $p[x]$ uložíme hodnotu $-\operatorname{rank}(x) \leq 0$.

Tu je jednoduchá implementácia na ∼15 riadkov:

```
int p[MAX]; // ak p[x] > 0, p[x] je otec x
            // ak p[x] <= 0, tak -p[x] je rank(x)

int find(int x) {
  if (p[x] <= 0) {
    return x; // koren
  } else {
    int root = find(p[x]); // rekurzivne
    p[x] = root;
  }
}

void union(int x, int y) { link(find(x), find(y)); }

void link(int rx, int ry) {  // predpokladame, ze rx, ry su korene
  if (rx == ry) return;                        // a)
  if (p[rx] == p[ry]) { p[rx]=ry; p[ry]--; } // b)
  else if (-p[rx] < -p[ry]) p[rx]=ry;         // c)
  else p[ry] = rx;                             // d)
}
```

a) rovnaké stromy – nerobíme nič

b) $r_x$ aj $r_y$ majú rovnaký rank; napojíme $r_x$ pod $r_y$ a $r_y$ zvýšime rank (pripomeňme, že v poli $p$ máme uloženú hodnotu $-rank$, takže túto hodnotu znížime o 1)

c) $r_x$ má menší rank, takže napojíme $r_x$ pod $r_y$

d) $r_y$ má menší rank, takže napojíme $r_y$ pod $r_x$

A tu je triková randomizovaná implementácia na dva riadky (s tým rozdielom, že pre otca koreňa si poznačíme 0 a pri rozhodovaní, ktorý strom podpojíme pod ktorý pri unione, si hodíme mincou):

```
int f(int x) { return p[x] ? p[x] = f(p[x]) : x;  }
void u(int x, int y) { rand() % 2 ? p[f(x)] = f(y) : p[f(y)] = f(x); }
```

## 1.3 Analýza union-findu

Všimnime si, že každá operácia union pozostáva z dvoch volaní find (na nájdenie koreňov príslušných množín), plus prelinkovania, čo je však len konštantne veľa práce. Preto sa pri analýze časovej zložitosti zameriame iba na operáciu find.

Ďalej si uvedomme, že časová zložitosť operácie $find(x)$ je úmerná dĺžke cesty od vrcholu $x$ ku koreňu, teda počtu „skokov“, ktoré musíme vykonať, aby sme sa ku koreňu dostali.

Dokážeme:

**Veta 1.1.** *Ak máme najviac $n < 2^{65\,535}$ prvkov, potom ľubovoľná postupnosť $m$ volaní* find *vykoná najviac $6m + 2n$ skokov.*

*Vo všeobecnosti platí, že ľubovoľná postupnosť $m$ operácií union a find má celkovú časovú zložitosť $O((m+n)\log^* n)$, kde $\log^*$ je tzv.* iterovaný logaritmus.

Len pre porovnanie: milión je $10^6 < 2^{20}$, miliarda je $10^9 < 2^{30}$. Počet atómov v pozorovateľnom vesmíre sa odhaduje na menej ako $2^{300}$ (niekde medzi $10^{78}$ a $10^{82}$). Inými slovami, *v praxi* nikdy nebudeme mať viac ako $2^{65\,535}$ prvkov. Ani náhodou. A teda pre všetky praktické účely je zložitosť union-findu konštantná (v priemere na jednu operáciu).

A v teórii? V teórii sa, samozrejme, zaujímame, ako sa algoritmus chová pre $n$ rastúce do nekonečna a po $2^{65\,535}$ ešte nasleduje veľa čísiel. V skutočnosti zložitosť union-findu (v priemere na jednu operáciu) *nie je* konštantná, ale rastie veľmi veľmi pomaly, pomalšie ako iterovaný logaritmus.

Čo je to iterovaný logaritmus? Zoberte číslo $n$ a zadajte ho do kalkulačky. Potom opakovane stláčajte tlačidlo „`log`“, kým výsledok neklesne na hodnotu menšiu alebo rovnú 1. Počet stlačení logaritmu, ktoré ste museli vykonať, je práve $\log^* n$.

Napríklad: $\log^* 65\,536 = 4$, pretože:

$$65\,536 \to 16 \to 4 \to 2 \to 1;$$

potrebovali sme stlačiť „`log`“ štyrikrát. Kolko je $\log^* 2^{65\,536}$? 5, pretože z $2^{65\,536}$ sa dostaneme na 65 536 už po jednom logaritme, a ďalej pokračujeme ako v predchádzajúcom príklade. Takže celkovo potrebujeme iba 5 stlačení.

Iterovaný logaritmus pre $n \to \infty$ rastie do nekonečna:

$$\log^* 2^{2^{65\,536}} = \log^* 2^{2^{2^{2^2}}} = 6, \qquad \log^* 2^{2^{2^{65,536}}} = \log^* 2^{2^{2^{2^{2^2}}}} = 7, \text{ atď.}$$

Každým pridaním ďalšieho poschodia zväčšíme iterovaný logaritmus o 1, takže ako nám bude vežička exponentov rásť do nekonečna, bude iterovaný logaritmus rásť donekonečna. Hoci nepredstaviteľne pomaly.

■ **Dôkaz.** Začnime niekoľkými jednoduchými pozorovaniami:

- Vrcholy s vysokým rankom sú zriedkavé. Pripomeňme si, že strom s rankom $r$ obsahuje aspoň $2^r$ vrcholov. To znamená, že najviac $n/2^r$ vrcholov môže dosiahnuť rank (a teda aj výšku) $r$. Napríklad iba 16-tina vrcholov môže mať rank aspoň 4 a iba 65 536-tina vrcholov môže dosiahnuť rank 16.

- Rank rodiča je vždy aspoň o 1 vyšší ako rank jeho detí. To znamená, že ak ideme po ceste od ľubovoľného vrcholu ku koreňu, hodnoty rankov striktne narastajú.

- Každý vrchol začína s rankom 0. Rank sa môže zvyšovvať len pokým je daný vrchol koreňom. Akonáhle vrchol pripojíme pod iný, jeho rank sa už nikdy nezmení. Jeho rodičovi však môže rank ďalej rásť.

- Pre všetky vrcholy okrem koreňov definujeme hodnotu $gap(x)$ ako rozdiel $gap(x) = rank(parent(x)) - rank(x)$. Pri každom použití path compression (t.j. stlačení cesty) sa všetky vrcholy okrem koreňa a jeho bezprostredného dieťaťa presunú pod „vyššieho“ rodiča, teda pod vrchol s vyšším rankom. Výsledkom je, že $gap(x)$ sa pre všetky tieto vrcholy zvýši.

Rozdeľme ranky do troch úrovní:

- Úroveň 0: ranky od 0 do 3,
- Úroveň 1: ranky od 4 do $2^4 - 1 = 15$,
- Úroveň 2: ranky od 16 do $2^{16} - 1 = 65\,536$,
- ... a mohli by sme takto pokračovať, ale pre $n < 2^{65\,536}$ viac úrovní nebude.

Poďme teraz spočítať celkový počet „skokov“ vykonaných počas všetkých $m$ operácií find dokopy. Všetky skoky rozdelíme do štyroch kategorií:

1. **Úroveň 0 (ranky 0–3):** Na tejto úrovni strávime najviac 3 skoky pri každej operácií, teda spolu najviac $3m$ skokov.
2. **Finálne skoky ku koreňu:** Pre každý find(x) si osobitne započítame posledný skok do koreňa. To je najviac $m$ skokov.
3. **Skoky medzi úrovňami:** Keď počas find preskočíme z jednej úrovne do druhej, každý takýto „medziúrovňový“ skok započítame zvlášť. Keďže máme len 3 úrovne, týchto skokov je najviac $2m$.
4. Zostáva spočítať **skoky vo vnútri úrovní 1 a 2, ktoré nie sú finálne.** A tu prichádzame k pointe dôkazu (prečo skoky delíme na kategórie, ktoré rátame zvlášť):

   (a) **Všetky skoky v rámci úrovne 1:** sú také skoky, z vrcholu $x$ do jeho rodiča $y = parent(x)$, že oba vrcholy $x$ aj $y$ majú ranky v rozmedzí $4 \leq rank(x) < rank(y) < 16$. To znamená, že $gap(x) = rank(parent(x)) - rank(x)$ je najviac 11. Avšak keďže počítame iba nefinálne skoky, $y$ nie je koreň a má otca $z$ s ešte väčším rankom. A tu využijeme, že používame kompresiu cesty: Pri každom finde sa pri kompresii cesty $x$ napojí na koreň, t.j. na $z$ alebo ešte vyššie. Tým pádom hodnota $gap(x)$ vzrastie aspoň o 1. To znamená, že po 11-tich findoch, ktoré prechádzajú cez $x$ a skok z $x$ na otca nie je finálny, narastie $gap(x)$ aspoň na 12 a to znamená, že $x$ už bude napojený na vrchol na vyššej úrovni. To znamená, že všetky ďalšie findy prechádzajúce cez $x$ už budú medziúrovňové a tie tu nepočítame (už sme ich započítali v bode 3.).

   Zhrnutie: Cez každý vrchol spravíme najviac 11 nefinálnych skokov v rámci úrovne 1. Zároveň však platí, že na úroveň 1 sa dostane iba 16-tina vrcholov. Z toho vyplýva, že celkový počet skokov v rámci úrovne 1 je najviac $11 \times (n/16) < n$.

(b) **Všetky skoky v rámci úrovne 2:** spočítame podobne: Ranky sú tu od 16 do $2^{16}-1$, takže $gap(x)$ musí byť menej ako $2^{16}$. Po $2^{16}$ findoch cez $x$ narastie $gap(x)$ tak, že otec $x$ už nutne patrí na úroveň 3 alebo vyššie.

Takže spravíme menej ako $2^{16}$ operácií na každý vrchol na 2. úrovni. Avšak 2. úroveň znamená rank aspoň 16 a iba $2^{16}$-tina všetkých vrcholov tento rank niekedy dosiahne. Spolu to je teda menej ako $2^{16} \times (n/2^{16}) = n$ skokov.

Takto dostávame, že všetkých skokov je najviac $6m + 2n$ (ak $n < 2^{65\,535}$). Skúste si rozmyslieť, ako treba dôkaz upraviť pre všeobecné $n$.[1] □

**Aká je skutočná zložitosť tejto dátovej štruktúry?** Ukázali sme, že union-find so spájaním podľa ranku a kompresiou cesty trvá najviac $O((m+n)\log^* n)$, teda $\log^* n$ na jednu operáciu (ak $m \geq n$). Ale aká je *skutočná* zložitosť tejto dátovej štruktúry? Možno si poviete: *„Nie je v skutočnosti tá zložitosť konštantá? Nie je toto všetko len slabá analýza? Keby sme sa viac snažili, nevedeli by sme dokázať lepšiu hornú hranicu?“*

Odpoveď znie: Áno a Nie. R. E. Tarjan 1975; R. Tarjan a Van Leeuwen 1984 dokázal lepší odhad: že $m$ union-find operácií trvá $O(m\alpha(m,n))$, kde $\alpha$ je funkcia, ktorá súvisí s inverznou Ackermannovou funkciou, ktorá síce rastie do nekonečna, ale ešte oveľa pomalšie ako iterovaný logaritmus. Pre všetky praktické odhady môžeme predpokladať, že $\alpha(m,n) \leq 3$. Fascinujúce je, že v zápätí ukázal aj dolný odhad (R. E. Tarjan 1979), tzn. našiel takú postupnosť operácií union a find, pri ktorej spravíme aspoň $\Omega(m\alpha(m,n))$ skokov! To znamená, že *skutočná* zložitosť *nie je* konštantná a naozaj v najhoršom prípade rastie do nekonečna, hoci veľmi veľmi pomaly.

## 1.4 Späť k transpozícii matíc

Poďme si konečne prezradiť, čo sa to vlastne porobilo pri transpozícii matíc. Odpovede na všetky záhady súvisia s pamäťovým systémom moderných počítačov a vyrovnávacou pamäťou (cache).

Moderné procesory sú extrémne rýchle – vykonávajú miliardy operácií za sekundu. Naproti tomu hlavná pamäť (RAM) je relatívne pomalá – čas na čítanie/písanie do RAM je rádovo sto-krát pomalšie ako je rýchlosť samotného procesora. To znamená, že ak by procesor musel na každý údaj čakať, kým dorazí z RAM, väčšinu času by sa nudil. Aby sa tento problém vyriešil, moderné procesory majú viacúrovňový systém cache pamätí – malých, ale extrémne rýchlych

[1] Pre všeobecné $n$ by sme namiesto 3 úrovní mali $\log^* n$ úrovní: Každá ďalšia úroveň by mala ranky v rozsahu $[k, 2^k)$, takže na $i$-tej úrovni by boli ranky od $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{i+1}$ po $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{i+2} - 1$.

V kategóriach 1. a 2. je rovnako veľa skokov, ale v kategórií 3. bude $O((\log^* n) \times m)$ medziúrovňových skokov a 4. v rámci každej úrovne spravíme najviac $n$ nefinálnych skokov, ale máme $\log^* n$ úrovní, takže spolu $O((\log^* n) \times n)$.

pamäťových blokov, ktoré uchovávajú najčastejšie používané údaje. Súčasné počítače mávajú tri úrovne cache: tzv. L1, L2, a L3, každá ďalšia úroveň je väčšia, zato pomalšia. Napríklad počítač, na ktorom píšem túto kapitolu má 256KiB L1d dátovej cache, 256KiB inštrukčnej L1i cache (sem idú inštrukcie, ktoré procesor vykonáva), 8MiB L2 cache, 16MiB L3 cache a 58GiB RAM. Na tejto architektúre má každý procesor svoju vlastnú L1 a L2 cache, ale L3 a RAM sú spoločné.

Keď chce procesor načítať nejaký údaj z pamäti, najskôr sa pozrie do L1 cache, ak tam údaj nie je (tzv. L1 cache miss), pozrie sa do L2 cache, potom do L3, až nakoniec, ak sa údaj nenachádza ani v L3, načíta ho z RAM. Pri načítaní sa zároveň údaje prekopírujú do nižších úrovní cache (napr. z RAM do L3, z L3 do L2, z L2 do L1), tak, aby opakovaný prístup k nim už bol rýchly. A tu sa dostávame k podstatnej časti: nebolo by efektívne, keby sme pri načítaní skopírovali len 1 bajt, alebo 1 int (4 bajty). Pri každom prístupe do pamäte sa prenáša celý *blok pamäte*, tzv. cache line, typicky veľkosti 64 bajtov[2]. To má veľmi citeľné praktické dôsledky!

Vyskúšajte si nasledujúci jednoduchý experiment: Zoberte si maticu $N \times N$ (napríklad typu int) a zmerajte, koľko trvá, ak budete jej prvky čítať riadok po riadku; potom to porovnajte s prípadom, keď maticu čítate stĺpec po stĺpci; nakoniec to porovnajte s prípadom, keď prečítate všetky prvky matice, ale v *náhodnom* poradí. V štandardnom teoretickom modeli, kde predpokladáme, že každá základná operácia procesora trvá konštantný čas, má každý algoritmus zložitosť $N^2$, avšak v praxi bude medzi týmito troma prechodmi obrovský rozdiel.

Pri čítaní po riadkoch využívame cache najefektívnejšie. Do jednej cache line sa zmestí 16 intov (32-bitových) a vždy po tom, ako prečítame jeden, prečítame aj ďalších 15 nasledujúcich, čo je takmer zadarmo. Naopak, ak čítame v náhodnom poradí, pri dostatočne veľkej matici, ktorá je mnohonásobne väčšia ako L3, bude takmer každý prístup do pamäte L3 cache miss a program pobeží rádovo sto-krát dlhšie.

Pri čítaní po stĺpcoch zakaždým prečítame 4 bajty zo 64 a prejdeme na ďalší riadok... Tu bude čas závisieť od uloženia v pamäti: máme maticu v jednom veľkom bloku pamäte, alebo ju máme uloženú ako pole riadkov, kde každý jeden riadok je samostatne naalokovaný blok pamäte? Napríklad v C++, máme statické pole `int A[N][N]`, alebo `vector<vector<int>> A`? V tom prvom prípade bude aj čítanie po stĺpcoch rýchle vďaka tomu, že procesor je v tomto prípade schopný predpovedať adresu nasledujúceho prístupu (políčko v rovnakom stĺpci ale o 1 nižšom riadku je o $4 \times N$ bajtov ďalej). Ďalší trik moderných procesorov je tzv. *prefetching* – ak vedia dopredu predpovedať, ktorú adresu v pamäti bude treba, môžu už dopredu začať načítavať. Naopak, v druhom prípade, môže byť každý riadok úplne inde a ťažké predpovedať ďalší prístup do pamäte.

Dodajme ešte, že keď už je cache plná, pri načítaní nových dát musíme vždy niečo vyhodiť, aby sme uvoľnili miesto pre nové dáta. Ktoré údaje sa vyhodia a vplyv na chovanie nášho programu sa ťažko predpovedá – treba merať.

[2] závisí od architektúry, najnovšie počítače už majú aj 128 bajtové cache line

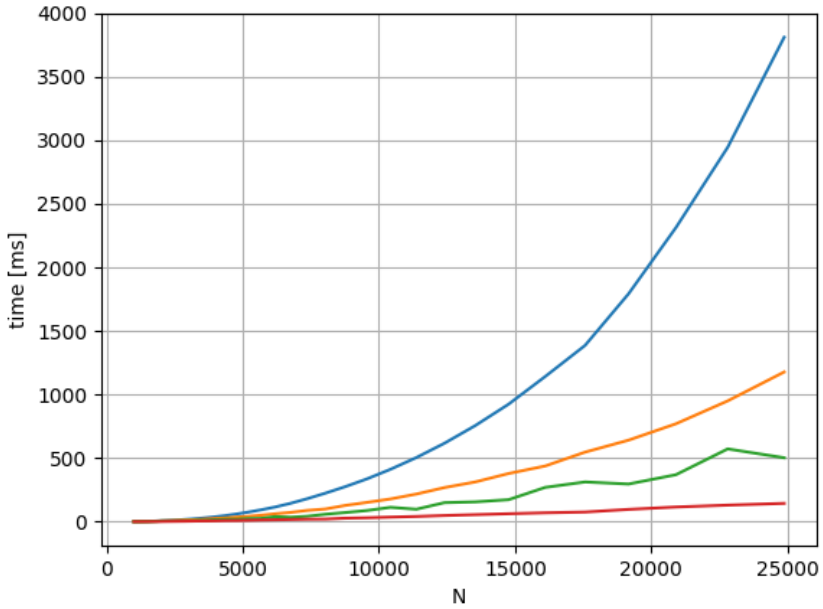
Ale vráťme sa k transpozícií matíc. Šok #2 by sme už po tomto úvoďme mali vedieť ľahko vysvetliť: Čas deleno $N^2$ *je* konštanta, avšak zatiaľčo pre veľmi malé $N < 256$ sa celá matica zmestí do L1 cache, pre $N > 1500$ sa už nezmestí ani do L2 a pre $N > 2048$ ani do L3 cache. S rastúcim $N$ pribúdajú cache missy a to, že čas na prvok matice stúpne „iba“ na $\approx$ 15-násobok (z 0.4ns na 6ns) je dôkaz, že cachovanie ešte stále trochu funguje. Ak by sme predsalen chceli použiť idealizovaný teoretický model, kde každá operácia má jednotkovú cenu, museli by sme do tej ceny započítať, že v najhoršom prípade pristupujeme až do RAM, čo sú desiatky nanosekúnd. (O prípade, že sa dáta nezmestia ani do RAM a pagingu niekedy inokedy.)

Skúsme teraz porozmýšľať, ako algoritmus na transpozíciu matice zlepšiť. Pamätajte, že je dané, ktoré dvojice políčok treba vymeniť, ale nie je dané, v akom poradíV skutočnosti je poradie prístupov do pamäte to jediné, čo budeme ďalej meniť. Pripomeňme, že v pôvodnom algoritme postupne prechádzame všetky dvojice $(i, j)$, pričom vymieňame prvky $A[i][j] \leftrightarrow A[j][i]$. Políčka $(i, j)$ síce prechádzame po riadkoch, čo je dobre, ale tým pádom pozície $(j, i)$ prechádzame po stĺpcoch, čo je zle, lebo pri tom využijeme iba 4/64 cache line.

**Riešenie 2:** Rozdeľme maticu na bloky veľkosti $B \times B$. Postupne prechádzame blok po bloku po riadkoch a každý blok $A[i..i+B-1][j..j+B-1]$ vymeníme s (transponovaným) blokom $A[j..j+B-1][i..i+B-1]$.

```
void transpose_block(vector<vector<int>> &A) {
  int N = A.size();
  int B = 64;
  for (int k = 0; k < N; k += B) {
    // transponuj blok [k..k+B][k..k+B] na uhlopriecke
    for (int i = k; i < k + B && i < N; ++i)
      for (int j = i + 1; j < k + B && j < N; ++j)
        swap(A[i][j], A[j][i]);
    for (int l = k + B; l < N; l += B)
      // transponuj blok [k..k+B][l..l+B] <-> [l..l+B][k..k+B]
      for (int i = k; i < k + B && i < N; ++i)
        for (int j = l; j < l + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
  }
}
```

Prvé tri vnorené cykly riešia bloky na uhlopriečke (treba ošetriť zvlášť), nasledujúce cykly vymieňajú bloky so ľavými hornými rohmi $(k, l) \leftrightarrow (l, k)$. Najlepšiu hodnotu $B$ som zvolil skusmo.




A teraz pozor... chvíľka napätia... tento algoritmus sa umiestnil až na treťom mieste! Oranžová čiara na grafe vyššie.

To znamená, že sa to dá ešte lepšie

Čo sa dá ešte zlepšiť? Nuž, zastavme sa pri otázke, akú veľkosť bloku $B$ zvoliť. Je možné, že to záleží aj od toho, na ktorú úroveň cache sa pýtame – chceme minimalizovať počet L1 cache missov? alebo L2? alebo L3?

**Riešenie 3:** Skúsme použiť tú istú myšlienku ešte raz Maticu najskôr rozdelíme na veľké bloky veľkosti $B \times B$. Maticu budeme prechádzať postupne po veľkých blokoch. Keď však budeme vymieňať $A[x..x+B-1][y..y+B-1] \leftrightarrow A[y..y+B-1][x..x+B-1]$, spravíme to tak, že ich najskôr prerozdelíme na menšie bloky $b \times b$, a tie budeme postupne vymieňať.

Treba trochu zaťať zuby a voilà:

```
void transpose_block2(vector<vector<int>> &A) {
  int N = A.size();
  int B = 1040;
  int b = 4;
  for (int x = 0; x < N; x += B) {
    for (int k = x; k < x + B && k < N; k += b) {
      for (int i = k; i < k + b && i < N; ++i)
        // transponuj blok [k..k+B][k..k+B] na uhlopriecke
```

```
        for (int j = i + 1; j < k + b && j < N; ++j)
          swap(A[i][j], A[j][i]);
      for (int l = k + b; l < x + B && l<N; l += b)
        // velky blok na uhlopriecke, maly blok mimo
        for (int i = k; i < k + b && i < N; ++i)
          for (int j = l; j < l + b && j < N; ++j)
            swap(A[i][j], A[j][i]);
    }
    for (int y = x + B; y < N; y += B)
      for (int k = x; k < x + B && k<N; k += b)
        for (int l = y; l < y + B && l < N; l += b)
          // transponuj blok [k..k+B][l..l+B] <-> [l..l+B][k..k+B]
          for (int i = k; i < k + b && i < N; ++i)
            for (int j = l; j < l + b && j < N; ++j)
              swap(A[i][j], A[j][i]);
  }
}
```

Krása. 6 vnorených for-cyklov.

A čo sme dosiahli? Druhé miesto. Zelená čiara na grafe vyššie.

Až sa začínam báť, čo bude nasledovať. Ako povedal klasik: *„Do kedy my takto chceme? Dokéédýýý?“*

**Riešenie 4:** Nebojte sa, nejdem písať troj-úrovňové riešenie (veľké, stredné, malé bloky) s ôsmimi vnorenými for-cyklami. Kdeže. Keď už, tak ideme all in. Myšlienku delenia na bloky použijeme rekurzívne!

Ako transponujeme $N \times N$ maticu $A$? Rozdelíme ju na štyri podmatice $\frac{N}{2} \times \frac{N}{2}$:

$$A = \begin{pmatrix} B & C \\ D & E \end{pmatrix} \qquad \text{a transponujeme:} \qquad A^T = \begin{pmatrix} B^T & D^T \\ C^T & E^T \end{pmatrix}.$$

Všimnite si, že podmatice $B$ a $E$ na diagonále sa iba transponujú, $C$ a $D$ sa transponujú a navzájom vymenia. Ako? Rekurzívne: opäť sa rozdelia na štyri podmatice a tie sa vymieňajú a transponujú. Takto pokračujeme v rekurzívnom delení, kým sa nedostaneme na dostatočne malé matičky, ktoré vymeníme a transponujeme klasicky dvoma for-cyklami.

```
void transpose_rec(const int N, vector<vector<int>> &A,
                   int B, int i0, int j0) {
  if (B <= 4) {
    if (i0 == j0) {
      for (int i = i0; i < i0 + B && i < N; ++i)
        for (int j = i + 1; j < i0 + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
    } else {
```

```
      for (int i = i0; i < i0 + B && i < N; ++i)
        for (int j = j0; j < j0 + B && j < N; ++j)
          swap(A[i][j], A[j][i]);
    }
  } else {
    int h = B / 2;
    transpose_rec(N, A, h, i0, j0);
    transpose_rec(N, A, B - h, i0 + h, j0);
    if (i0 != j0) transpose_rec(N, A, B - h, i0, j0 + h);
    transpose_rec(N, A, B - h, i0 + h, j0 + h);
  }
}

transpose_rec(N, A, N, 0, 0, 0, 0);
```

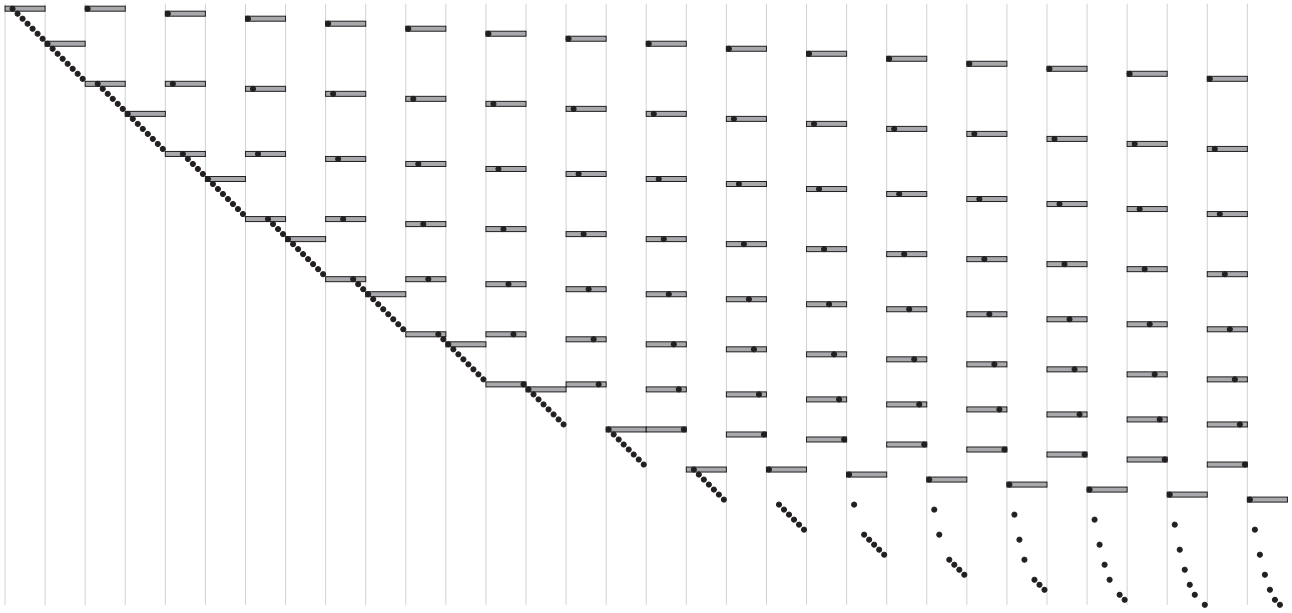
Dámy a páni, toto je naše víťazné riešenie, červená čiara na grafe vyššie.

## Referencie

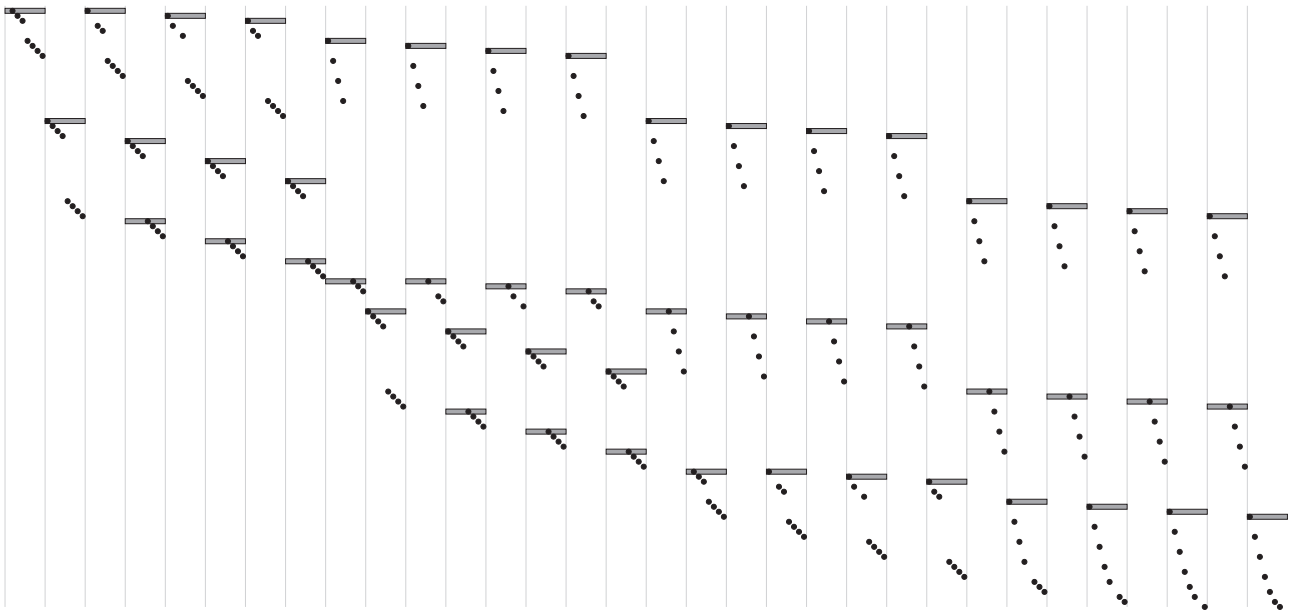
Tarjan, R.E. a J. Van Leeuwen (1984). „Worst-case analysis of set union algorithms“. In: *Journal of the ACM (JACM)* 31.2, s. 245–281.

Tarjan, Robert Endre (1975). „Efficiency of a Good But Not Linear Set Union Algorithm“. In: *J. ACM* 22.2, s. 215–225.

— (1979). „A Class of Algorithms which Require Nonlinear Time to Maintain Disjoint Sets“. In: *J. Comput. Syst. Sci.* 18.2, s. 110–127.

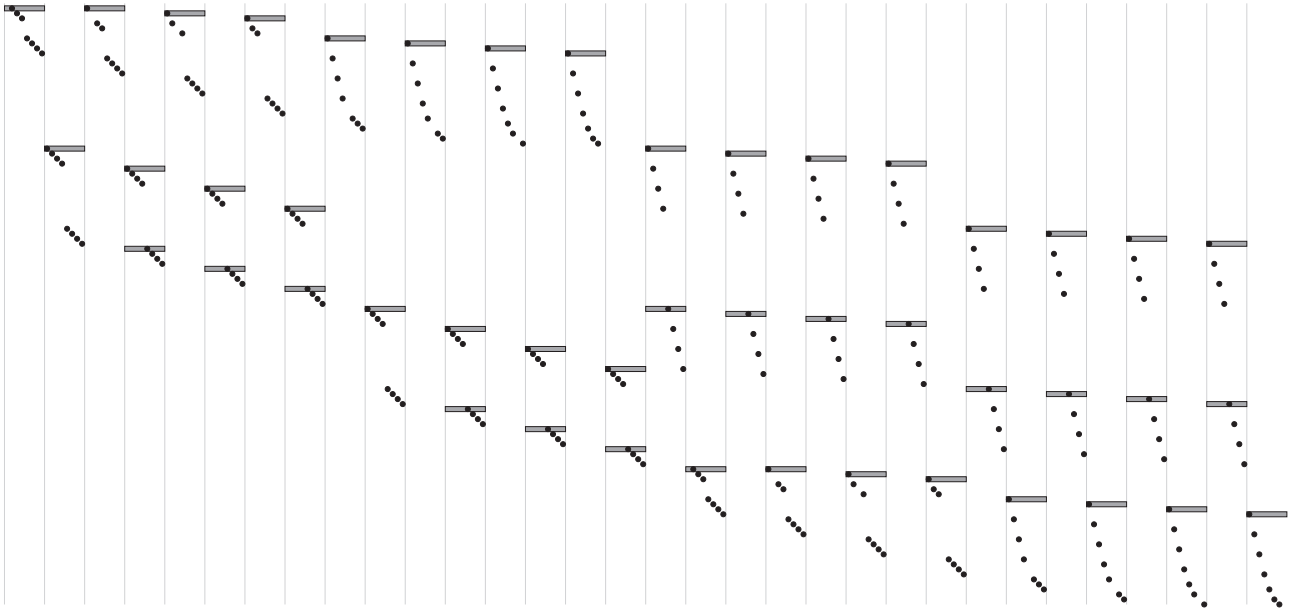


(a) Prechod po riadkoch.

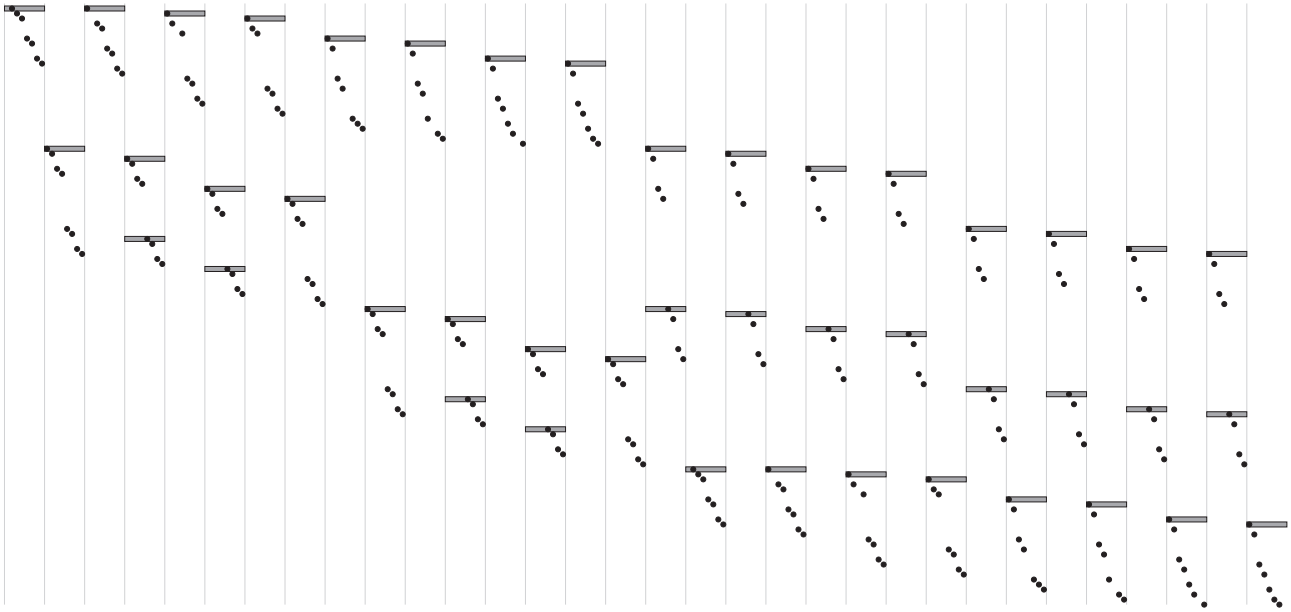


(b) Prechod po blokoch

Obr. 1.2: Transponovanie matice $16 \times 16$: a) po riadkoch a b) po blokoch $4 \times 4$. Os $x$ zľava doprava predstavuje pamäť, os $y$ zhora nadol predstavuje čas. Na obrázku je zobrazený priebeh algoritmu: každá čierna bodka znamená prístup do pamäte, šedý obdĺžnik znázorňuje cache miss a načítanie cache line. V tomto hračkárskom príklade máme cache line dĺžky 8 prvkov a cache má kapacitu 8 cache line. Jednotlivé riešenia spôsobia postupne 115 a 50 cache missov.



(a) Dve úrovne blokov.



(b) Rekurzívne riešenie.

Obr. 1.3: [pokračovanie] Transponovanie matice $16 \times 16$: c) dve úrovne blokov $4 \times 4$ a $8 \times 8$, d) rekurzívne. Tiero riešenia spôsobia postupne 46, respektíve 44 cache missov.

# Kapitola 2

# Vektory

Statické polia (klasické array) majú jednu veľkú nevýhodu: potrebujeme poznať veľkosť vopred.

## 2.1 Dynamické zväčšovanie a zmenšovanie

Základný nápad je veľmi jednoduchý: Keď je kapacita plná, vektor si rezervuje nový väčší blok pamäte, skopíruje všetky existujúce prvky na nové miesto, a pokračuje ďalej s väčším priestorom.

### Amortizovaná analýza

Ukážeme si, že amortizovaný čas jednej operácie `push_back` do vektora je $O(1)$, ak vektor pri každom zaplnení zdvojnásobí svoju kapacitu.

Použijeme na to účtovnícku metódu.

## 2.2 Ako sa nestreliť do nohy

## 2.3 Optimalizácie

### Bitvektory

Ako ušetriť pamäť pri uchovávaní veľkého množstva binárnych dát.

### Zväčšovanie

Knižnica Folly a jej optimalizácie

### Spolupráca s alokátorom

### Rozloženie v pamäti

Vec vs. SmallVec a TinyVec

### Kopírovanie

### Vektor štruktúr alebo štruktúra vektorov?

# Kapitola 3

# Amortizovaná analýza

**Definícia 3.1.** *Označme reálnu cenu $i$-tej operácie $c_i$. Amortizovaná zložitosť je $a_i$, ak pre každú postupnosť operácií*

$$\sum_i c_i \leq sum_i a_i$$

## 3.1 Potenciálová metóda

Nech $D_i$ je dátová štruktúra po $i$-tej operácii. Potenciál je funkcia

$$\Phi : \mathcal{D} \to \mathbb{R}.$$

Často dokonca pri analýze volíme potenciál, ktorý je nezáporný.

Amortizovanú zložitosť potom definujeme ako

$$a_i = c_i + \Delta\Phi = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

Celková amortizovaná zložitosť postupnosti operácií je potom

$$\begin{aligned} \sum_{i=1}^{m} a_i &= \sum_{i=1}^{m} c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= \left(\sum_{i=1}^{m} c_i\right) + \Phi(D_n) - \Phi(D_0) \end{aligned}$$

Ak $\Phi(D_n) - \Phi(D_0) \geq 0$, tak z toho vyplýva, že $\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} c_i$.

Väčšinou volíme potenciál, kde $D_0$ je prázdna štruktúra s nulovým potenciálom $\Phi(D_0) = 0$ a potenciál je vždy nezáporný. V tom prípade $\Phi(D_n) - \Phi(D_0) = \Phi(D_n) \geq 0$ a nerovnosť platí.

# Časť II

# Haldy

# Kapitola 4

# Najkratšie cesty

# Kapitola 5

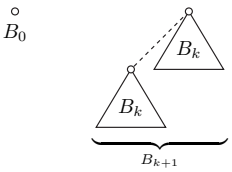
# Binomiálna halda

Binomiálnu haldu vynašiel Jean Vuillemin v roku 1976. V praxi síce často prehráva s obyčajnou binárnou haldou (najmä kvôli väčším konštantám skrytým v $O$-notácii), no má jednu unikátnu schopnosť: dokáže efektívne zlúčiť dve haldy v logaritmickom čase, na rozdiel od klasickej binárnej haldy, ktorá na to potrebuje lineárny čas.

Pre nás sú binomiálne haldy zaujímavé najmä ako odrazový mostík na ceste k sofistikovanejším štruktúram, konkrétne k Fibonacciho halde, ku ktorej sa postupne prepracujeme v ďalšej kapitole.

## 5.1 Binomiálne stromy

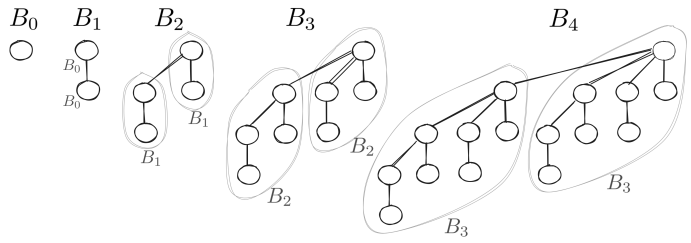
Základným stavebným prvkom binomiálnej haldy je *binomiálny strom*. Binomiálny strom rádu $k$ (označujeme $B_k$) definujeme rekurzívne:




- $B_0$ je strom s jediným vrcholom,

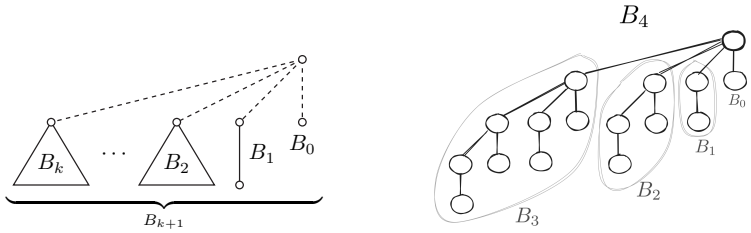
- $B_{k+1}$ vznikne tak, že zoberieme dva stromy $B_k$ a jeden z nich pripojíme ako najľavejšie dieťa koreňa druhého.

Takto vyzerajú binomálne stromy rádu 0 až 4:




Z definície vieme hneď odvodiť niekoľko dôležitých vlastností: Binomiálny strom rádu $k$ má

- presne $2^k$ vrcholov,
- výšku $k$,
- a deti koreňa tvoria (zľava doprava) stromy rádu $B_{k-1}, B_{k-2}, \ldots, B_0$.




V každom vrchole binomiálneho stromu je uložený práve jeden prvok haldy. Okrem toho budeme vyžadovať, aby bol každý binomiálny strom *haldovito usporiadaný*: kľúč vo vrchole je vždy menší alebo rovný kľúčom všetkých jeho detí (a teda aj všetkých potomkov). Z tohto *min-heap invariantu* vyplýva, že najmenší prvok celého stromu sa nachádza vždy v jeho koreni.

## 5.2 Štruktúra binomiálnej haldy

Binomiálna halda s $n$ prvkami sa skladá z viacerých binomiálnych stromov. Ako presne? Každý binomiálny strom má veľkosť rovnú mocnine 2, potrebujeme teda z mocnín 2 „poskladať“ číslo $n$...

Vezmime napríklad $n = 41$. V binárnom zápise je to `101001`, pretože číslo 41 vieme jednoznačne rozložiť na súčet mocnín dvojky:

$$41 = 32 + 8 + 1 = 2^5 + 2^3 + 2^0.$$

Binomiálnu haldu so 41 vrcholmi preto môžeme poskladať zo stromov $B_5$, $B_3$ a $B_0$ (veľkostí $2^5$, $2^3$ a $2^0$, spolu 41 vrcholov).

Podobne napríklad $n = 75$ má binárny zápis `1001011`, lebo

$$75 = 64 + 8 + 2 + 1 = 2^6 + 2^3 + 2^1 + 2^0.$$

Halda s $n = 75$ vrcholmi preto obsahuje stromy $B_6$, $B_3$, $B_1$ a $B_0$.

Vo všeobecnosti môžeme *binomiálnu haldu* definovať ako zoznam niekoľkých binomiálnych stromov $B_i$, pričom každý z nich má *iný rád* $i$. Vďaka tejto vlastnosti má binomiálna halda s $n$ vrcholmi vždy pevne určený tvar, ktorý priamo zodpovedá binárnemu zápisu čísla $n$: ak je $k$-ty bit zápisu rovný `1`, halda obsahuje strom $B_k$; ak je rovný `0`, strom $B_k$ sa v halde nenachádza.

Z tejto súvislosti vidíme, že binomiálna halda s $n$ prvkami obsahuje najviac $\log_2 n$ stromov a ich rády sa pohybujú od 0 po $\lfloor \log_2 n \rfloor$. Keďže výška každého stromu sa rovná jeho rádu, aj výška stromov je nanajvyš logaritmická. To je dôležité, pretože všetky základné operácie s binomiálnou haldou majú zložitosť úmernú buď počtu stromov, alebo ich výške, teda v najhoršom prípade $O(\log n)$.

## 5.3 Spájanie háld

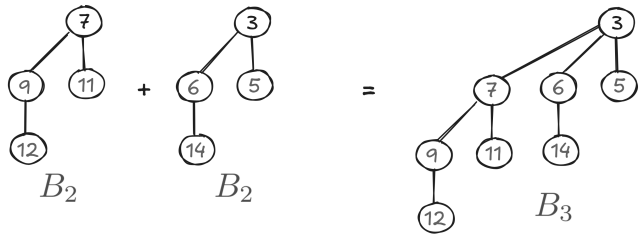
Spájanie dvoch binomiálnych háld prebieha veľmi podobne ako sčítanie čísel v dvojkovej sústave. Ukážme si to na príklade $41 + 75$:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 0 | 1 | 0 | 0 | 1 |
| + | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

Začíname sprava: $1 + 1$ je 2, takže do výsledku zapíšeme 0 a prenesieme 1 do vyššieho rádu. Na ďalšej pozícii máme $0 + 1 +$ prenesená $1 = 2$, takže opäť zapíšeme 0 a prenesieme 1 ďalej. Takto pokračujeme, kým nesčítame všetky bity oboch čísel.

Pri spájaní binomiálnych háld postupujeme analogicky, len namiesto sčítania bitov porovnávame a prípadne spájame binomiálne stromy. Ak sa na určitej „pozícii“ stretnú dva stromy rovnakého rádu, zlúčime ich do jedného stromu o jeden rád väčšieho. Pri zlúčení porovnáme kľúče v koreňoch a väčší koreň zavesíme pod menší, čím zachovávame min-heap invariant. Novovzniknutý strom sa posunie na ďalšiu pozíciu presne tak, ako sa pri binárnom sčítaní prenáša jednotka do vyššieho rádu.

Tu je príklad zlúčenia dvoch $B_2$ stromov do jedného $B_3$:



Keďže $3 < 7$, menší kľúč ostáva v koreni a vrchol 7 pripojíme pod neho. Táto operácia trvá len konštantný čas – vyžaduje len jedno porovnanie a úpravu niekoľkých smerníkov.

Vezmime ako príklad dve haldy veľkosti $n = 41$ a $n = 75$. Už vieme, že sa budú skladať z binomálnych stromov ($B_5$, $B_3$, $B_0$) a ($B_6$, $B_3$, $B_1$, $B_0$). Spájanie prebehne nasledovne (pozri tiež obr. 5.1):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `merge` $H_1$ | – | $B_5$ | – | $B_3$ | – | – | $B_0$ |
| a $H_2$ | $B_6$ | – | – | $B_3$ | – | $B_1$ | $B_0$ |
| výsledok | $B_6$ | $B_5$ | $B_4$ | – | $B_2$ | – | – |

Rád 0: Obe haldy obsahujú stromy $B_0$ → zlúčime ich na $B_1$ a prenesieme do vyššieho rádu,

Rád 1: $B_1$ z druhej haldy + prenesený $B_1$ → zlúčime na $B_2$ a prenesieme do vyššieho rádu,

Rád 2: $B_2$ v pôvodných haldách nie je → vo výsledku ostane prenesený $B_2$,

Rád 3: obe haldy majú $B_3$ → zlúčime na $B_4$ a prenesieme

Rád 4: $B_4$ sa inde nenachádza → vo výsledku ostane prenesený $B_4$,

Rád 5: $B_5$ je len v prvej halde → ostáva vo výsledku,

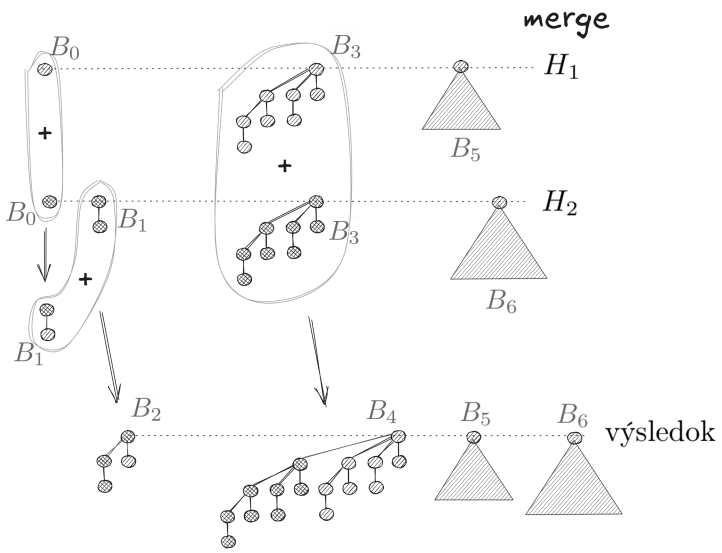
Rád 6: $B_6$ je len v druhej halde → ostáva vo výsledku.

**Zložitosť.** Pri zlučovaní prechádzame iba zoznam koreňov oboch háld a tých je len logaritmicky veľa. Každý krok spájania (porovnanie dvoch koreňov a prípadné zlúčenie stromov rovnakého rádu) trvá pri vhodnej reprezentácii v pamäti konštantný čas (implementačné detaily prenecháme ako cvičenie cteným čitateľom). Celková časová zložitosť spájania je preto $O(\log n_1 + \log n_2)$, kde $n_1$ a $n_2$ sú veľkosti spájaných háld.

## 5.4 Ostatné operácie

Všetky ostatné operácie binomiálnej haldy už sú jednoduché a vieme ich realizovať pomocou spájania.

**Vloženie prvku (insert).** Na vloženie nového prvku $x$ vytvoríme binomiálnu haldu obsahujúcu jediný vrchol ($B_0$ s prvkom $x$ v koreni), a túto jednoprvkovú haldu zlúčime s pôvodnou. Časová zložitosť vloženia je $O(\log n)$.




Obr. 5.1: „Sčítanie“ binomiálnych háld ($B_5,\ B_3,\ B_0$) a ($B_6,\ B_3,\ B_1,\ B_0$). Výsledkom je ($B_6,\ B_5,\ B_4,\ B_2$).

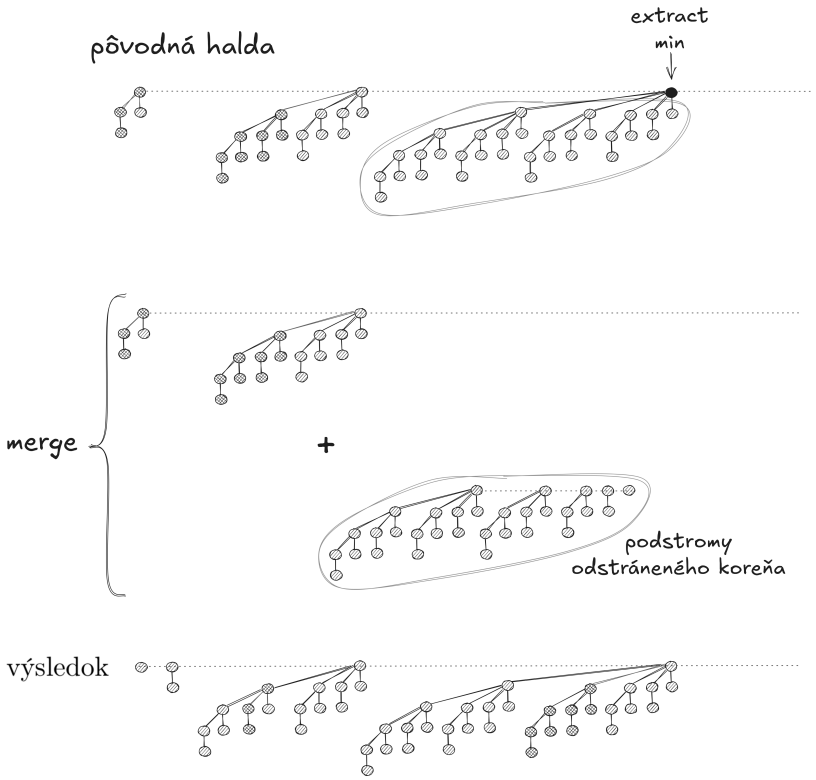
**Nájdenie minima (get-min).** Najmenší prvok sa vždy nachádza v jednom z koreňov binomiálnych stromov. Stačí teda prejsť všetky korene a vybrať ten s najmenším kľúčom. Keďže koreňov je najviac $\log_2 n$, hľadanie trvá $O(\log n)$.

Efektívnejšia možnosť je udržiavať spolu s haldou aj smerník na koreň s minimom a pri operáciách, ktoré ho môžu zmeniť, smerník aktualizovať. Takto vieme nájsť minimum v konštantnom čase, keďže výsledok bude vždy predpočítaný.

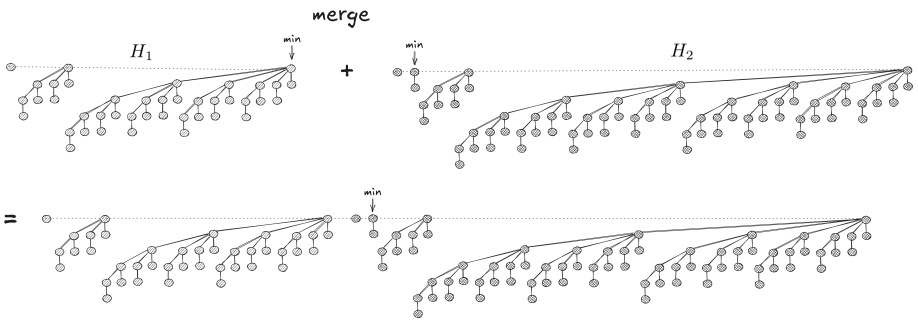
**Odstránenie minima (extract-min).** Vezmeme koreň s minimálnym kľúčom a odstránime ho. Jeho deti sú rôzne binomiálne stromy nižších rádov – spolu teda tvoria haldu, ktorú následne jednoducho zlúčime s pôvodnou haldou (pozri obr. 5.2). Výsledná časová zložitosť je $O(\log n)$.

**Zníženie kľúča (decrease-key).** Ak chceme zmenšiť kľúč v niektorom vrchole, nahradíme ho menšou hodnotou a „prebubleme“ ho nahor (podobne ako v klasickej binárnej halde). To znamená, že opakovane vymieňame vrchol so svojím rodičom a stúpame nahor, až kým nenarazíme na rodiča s menším kľúčom alebo sa nedostaneme až do koreňa. Týmto obnovíme min-heap invariant.

Výška stromu rádu $k$ je $k$, pričom najväčší možný rád v halde s $n$ prvkami je $\lfloor \log_2 n \rfloor$. Preto je časová zložitosť operácie $O(\log n)$.




Obr. 5.2: Odstránenie minima: koreň s najmenším kľúčom sa odstráni. Jeho podstromy tvoria samostatnú haldu, ktorú zlúčime s pôvodnou. Skúška správnosti: v dvojkovej sústave $\texttt{11100} - \texttt{1} = \texttt{11011}$, takže ak z $(B_4,\ B_3,\ B_2)$ odstránime jeden vrchol a upraceme, ostane nám $(B_4,\ B_3,\ B_1,\ B_0)$




Obr. 5.3: Spojenie dvoch lenivých háld: jednoducho spojíme dva zoznamy. Minimum vyberieme porovnaním predpočítaných minimím $H_1$ a $H_2$.

### Zhrnutie

| **Operácia** | **Časová zložitosť** |
|---|---|
| insert | $O(\log n)$ |
| merge | $O(\log n_1 + \log n_2)$ |
| get-min | $O(1)$ |
| extract-min | $O(\log n)$ |
| decrease-key | $O(\log n)$ |

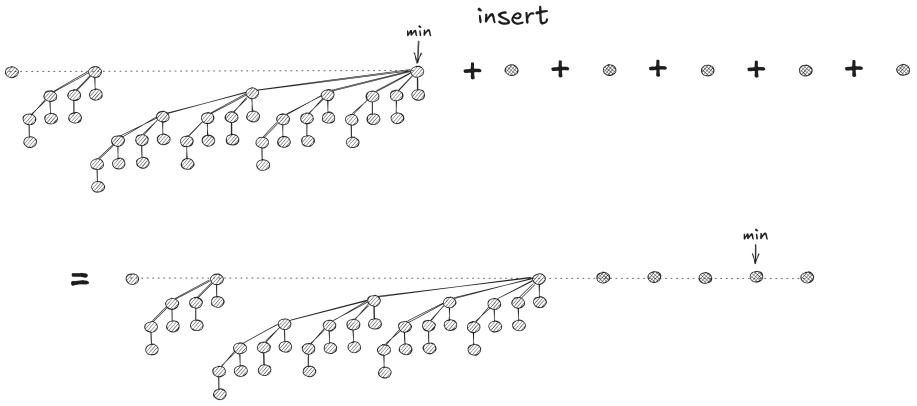
## 5.5 Lenivá binomiálna halda

Spájanie v logaritmickom čase je už samo osebe veľký pokrok. Natíska sa však otázka: nedalo by sa to ešte rýchlejšie? Na prvý pohľad to znie nereálne – zoznam koreňov má predsa logaritmickú dĺžku a pri spájaní ho musíme celý prejsť. Ukáže sa však, že ak si dovolíme trochu „porušiť poriadok“ a časť práce odložíme na neskôr, vieme `merge` (a teda aj `insert`) vykonať dokonca v *konštantnom amortizovanom čase*.

Základná myšlienka je jednoduchá: pri klasickej binomiálnej halde sú korene stromov pekne usporiadané a každý strom má *iný rád*. Pri spájaní vždy okamžite „upraceme“ všetky kolízie – ak sa stretnú dva stromy rovnakého rádu, zlúčime ich do jedného stromu s vyšším rádom.

V *lenivej* verzii však na toto upratovanie nemáme čas. Pri spájaní jednoducho pripojíme zoznam koreňov druhej haldy k zoznamu prvej a upratovanie odložíme na neskôr. Ak haldy reprezentujeme ako *spájané zoznamy* binomiálnych stromov, celé spájanie sa obmedzí na nastavenie zopár smerníkov, čo zvládneme v konštantnom čase.

Táto stratégia má však svoju cenu: V halde sa môžu nachádzať viaceré stromy rovnakého rádu (pozri obr. 5.3). Špeciálne ak do haldy vložíme $n$ nových prvkov, vznikne nám reťaz $n$ koreňov (pozri obr. 5.4). Ak začíname z prázdnej




Obr. 5.4: Lenivé vkladanie prvkov. Keďže upratovanie odkladáme na neskôr, vzniká neutriedený spájaný zoznam koreňov.

haldy, dostaneme veľmi degenerovanú verziu „binomiálnej haldy“: všetky stromy majú nulovú hĺbku, nulový rád a počet koreňov je lineárny. Takáto štruktúra je v podstate obyčajný spájaný zoznam.

„No moment. Ale ako potom efektívne vyberieme minimum?“ môže napadnúť pozorného čitateľa. Dobrá otázka! Pri odstránení minima totiž musíme nájsť nový najmenší prvok, čo v prípade spájaného zoznamu koreňov znamená lineárny prechod. Znamená to, že sme prišli o logaritmickú zložitosť pre túto kľúčovú operáciu prioritnej fronty?

Odpoveď znie: *nie tak celkom*. Prišli sme síce o logaritmický čas v najhoršom prípade, no ukážeme si, ako dosiahnuť, že operácia `extract-min` zostane rýchla *z amortizovaného hľadiska*.

Keďže pri `extract-min` aj tak musíme prejsť všetky korene, aby sme určili nové minimum, využijeme tento nevyhnutný lineárny prechod rovno aj na *upratovanie*. Počas neho postupne pozlučujeme stromy rovnakého rádu do väčších, čím obnovíme tvar klasickej binomiálnej haldy. Výsledkom je, že po náročnom prvom upratovaní je halda opäť „v poriadku“ a *nasledujúce* operácie `extract-min`, spojené s upratovaním a hľadaním minima, už prebiehajú omnoho rýchlejšie.

Napríklad ak vložíme $n$ prvkov a následne $n$-krát vyberieme minimum, všetky vkladania zaberú konštantný čas. Prvý výber spolu s upratаním trvá lineárny čas, no každý ďalší už len logaritmický čas. Všimnite si, že takýto prístup je pravdepodobne aj prakticky rýchlejší: namiesto $n$-krát [vloženie s upratаním] spravíme len [$n$-krát vloženie] a jedno veľké upratovanie. Počas prvých $n$ rýchlych operácií si stihneme „nasporiť“ dostatok kreditov, ktorými potom zaplatíme jednu drahšiu operáciu. Samozrejme, jeden príklad ešte nie je dôkaz. V nasledujúcej sekcii si ukážeme *poriadny* dôkaz, že `merge` má amortizovanú zložitosť $O(1)$ a `extract-min` $O(\log n)$, a to pre ľubovoľnú postupnosť

operácií.

Ešte predtým než sa do toho pustíme, skúste si rozmyslieť, ako toto upratovanie implementovať v čase lineárnom od počtu koreňov. Uvedomte si, že viacero koreňov môže mať rovnaký rád a môžu byť ľubovoľne rozhádzané po celom zozname.

## 5.6 Amortizovaná analýza

**Veta 5.1.** *V lenivej binomiálnej halde majú operácie nasledujúcu amortizovanú zložitosť:*

- *`insert` a `merge`* – $O(1)$,
- *`extract-min`* – $O(\log n)$.

Najskôr si ukážeme dôkaz pomocou účtovníckej metódy a následne ho prerozprávame rečou potenciálovej analýzy.

Predstavme si, že za každú vykonanú jednotkovú prácu platíme 1$ (t.j. 1$ pokryje $O(1)$ času). Za každú operáciu dostaneme určitý počet dolárov a našou úlohou bude ukázať, že tieto peniaze vždy postačia na zaplatenie všetkej práce a nikdy sa nedostaneme do mínusu. Pripomeňme, že pri amortizovanej analýze nemusíme minúť všetky peniaze hneď – časť si môžeme odložiť na neskôr, aby sme ňou pokryli náklady pomalších operácií.

■ **Dôkaz #1.** Ukážeme, že ak za každú operáciu dostaneme nasledovné množstvo peňazí:

- za `insert` 2$,
- za `merge` 1$ a
- za `extract-min` dostaneme $2\lfloor \lg n \rfloor + 1$ dolárov,

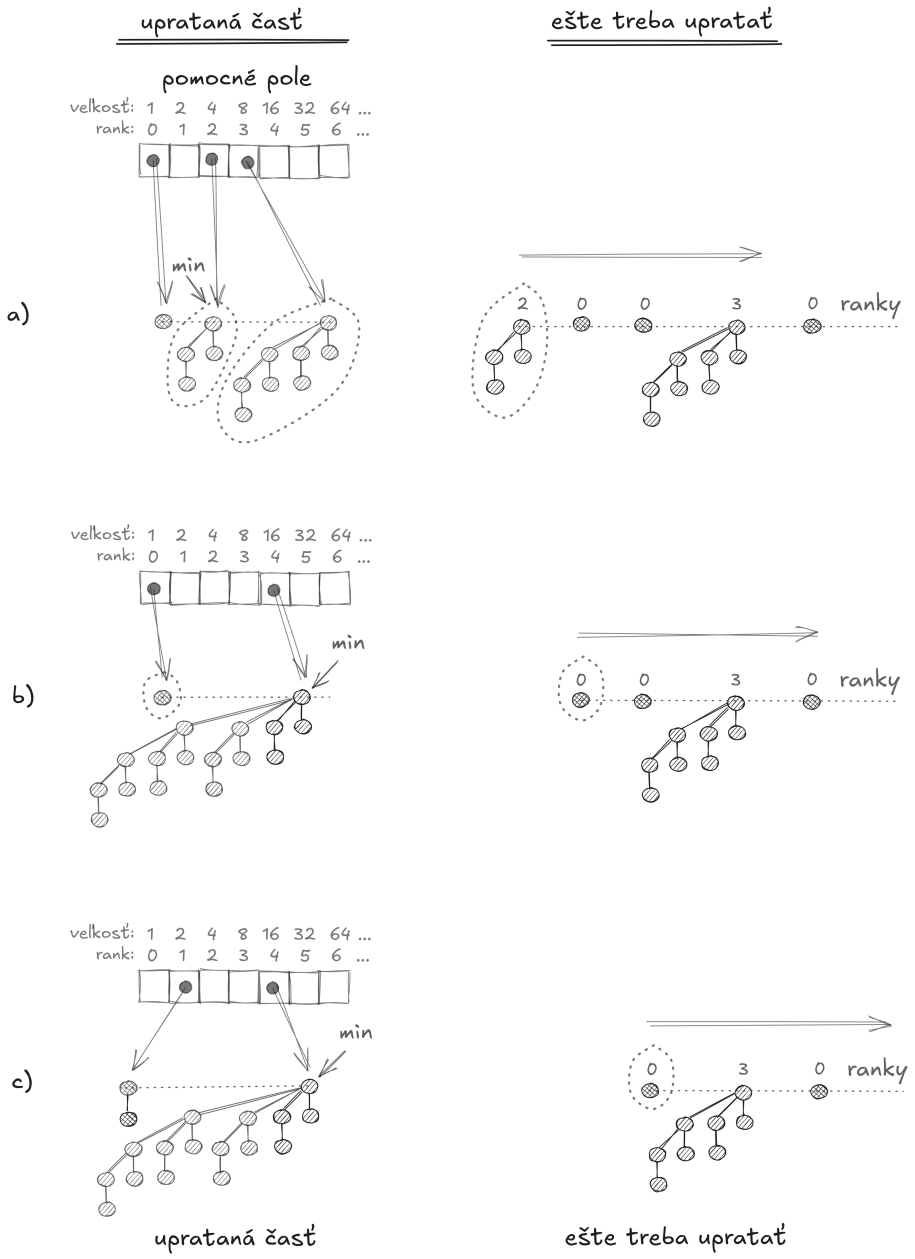
dokážeme nimi pokryť všetku potrebnú prácu.

Pri účtovaní sa budeme opierať o nasledujúci invariant:

**Invariant.** Každý koreň má vždy odložený 1$ na budúce upratovanie.

**Operácia `merge`.** Spojenie dvoch zoznamov a výber nového minima zaberie len konštantný čas. Túto prácu zaplatíme z prideleného 1$. Invariant ostáva automaticky zachovaný, pretože každý koreň mal už pred operáciou a stále má svoj 1$ odložený.

**Operácia `insert`.** Vytvorenie nového vrcholu a jeho pripojenie ako koreňa trvá $O(1)$. Jeden pridelený dolár zaplatí za túto prácu, druhý uložíme na účet nového koreňa, aby sme zachovali invariant.




Obr. 5.5: Upratovanie

**Operácia `extract-min`.** Rozdelíme ju na tri fázy:

1. *Odstránenie minima.* Odstránime koreň s najmenším kľúčom a jeho deti vložíme medzi ostatné korene. Tento krok zaplatí odstránený koreň zo svojho odloženého dolára.

2. *Príspevok novým koreňom.* Každé dieťa sa stane novým koreňom, preto mu vložíme na účet 1\$. Spolu takto prerozdelíme najviac $\lfloor \lg n \rfloor$ dolárov.

3. *Upratovanie.* Nech $t$ je počet koreňov pred upratovaním. Celé upratovanie trvá čas $O(t)$, respektíve stojí $t$ dolárov.

   Označme $\ell$ počet koreňov, ktoré sa pripoja pod iné a $r$ je počet tých, ktoré ostanú koreňmi. Zjavne $t = \ell + r$.

   Každý z tých $\ell$ koreňov má uložený 1\$, a keďže po upratаní už koreňmi *nebudú*, tieto peniaze môžeme použiť na zaplatenie práce. *Jednu časť upratovania teda pokryje halda sama zo svojich úspor.*

   Po upratаní zostane najviac $r \leq \lceil \lg n \rceil \leq \lfloor \lg n \rfloor + 1$ koreňov. To je tak málo, že *túto druhú časť výdavkov môžeme pohodlne uhradiť z peňazí, ktoré sme dostali na samotnú operáciu.*

Prvá fáza sa zaplatí sama, na nové korene prispievame najviac $\lfloor \lg n \rfloor$ dolárov a samotné upratovanie pokryjeme z odložených dolárov odchádzajúcich koreňov plus z ďalších najviac $\lfloor \lg n \rfloor + 1$ dolárov pridelených na operáciu. Preto celkovo postačí $2\lfloor \lg n \rfloor + 1$ dolárov vyčlenených pre `extract-min`.

Tým je dokázané, že pridelené rozpočty (2\$ na `insert`, 1\$ na `merge` a $2\lfloor \lg n \rfloor + 1$ na `extract-min`) postačujú na zaplatenie všetkej práce a pritom sa invariant vždy zachováva. □

Dôkaz účtovníckou metódou je veľmi názorný, pretože si vieme presne odsledovať kam sa každý dolár uloží a na čo sa neskôr použije. Poďme si však ukázať ten istý dôkaz v jazyku potenciálovej metódy.

Pripomeňme, že amortizovanú zložitosť počítame ako skutočnú zložitosť plus rozdiel potenciálov. Potenciál $\Phi(H)$ si môžeme predstaviť ako celkovú sumu našetrených dolárov v danej chvíli: ak potenciál rastie, šetríme, ak klesá, znamená to, že prácu platíme z úspor. Dôležité je, že potenciál dátovej štruktúry vždy začína na nule (pri prázdnej halde) a nikdy nesmie klesnúť pod nulu – inak by sme minuli viac, než máme na účte.

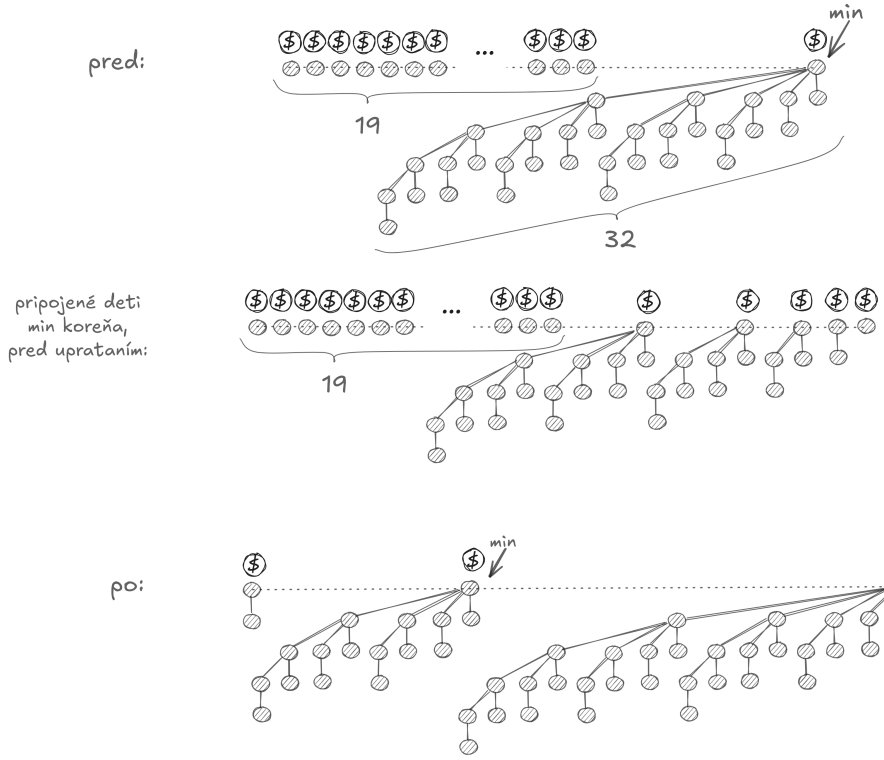
■ **Dôkaz #2.** Zvoľme si potenciál haldy $\Phi(H)$ ako *počet binomiálnych stromov*, teda počet koreňov.

Operácia `merge` má $T_{\text{skutočný}} = O(1)$ a $\Delta\Phi = 0$; `insert` má $T_{\text{skutočný}} = O(1)$ a $\Delta\Phi = +1$ (pribudne nový koreň). Obe tieto operácie teda trvajú $O(1)$ amortizovane.

Pri `extract-min`, nech $t_{\text{pred}}$ označuje počet stromov *na začiatku* a $t_{\text{po}}$ počet stromov *na konci* operácie.

Skutočný čas operácie je

$$T_{\text{skutočný}} = O(t_{\text{pred}} + \log n),$$




| | #stromov | reálna práca | vyúčtovanie |
|---|---|---|---|
| pred: | 20 | odstránenie minima<br>+ pripojenie detí: 1$ | 1$ zaplatí samotný koreň<br>5$ z peňazí na operáciu<br>vložíme deťom na účet |
| po pripojení detí: | $-1+5=24$ | upratovanie<br>+ nájdenie minima: 24$ | 21$ zaplatia bývalé korene<br>3$ zaplatíme z peňazí na operáciu |
| po uprataní: | $-21=3$ | | |
| spolu: | | 25$ | 8$ z peňazí na operáciu<br>17$ z našetrených peňazí |

Obr. 5.6: Konkrétny príklad operácie `extract-min` spolu s vyúčtovaním. Na vstupe je halda s 19 stromami rádu 0 a jedným rádu 5 (spolu $19+2^5=51$ vrcholov). Po odstránení minima a pripojení jeho 5 detí (1$) bude treba upratať 24 binomiálnych stromov. Reálna práca teda stojí 25$. Na operáciu však dostaneme pridelených iba $2\lfloor \lg n \rfloor + 1 = 13$$. Kľúčové je, že počet stromov klesne z 20 pred operáciou na iba 3 po nej. Rozdiel 17 stromov predstavuje 17 našetrených dolárov, ktoré vieme použiť na zaplatenie zvyšku.

pretože k pôvodným $t_{\text{pred}}$ stromom môže pribudnúť najviac $\log n$ detí odstráneného koreňa a všetky tieto stromy je potrebné počas upratovania spracovať.

Zmena potenciálu je

$$\Delta\Phi = (t_{\text{po}} - t_{\text{pred}})\$.$$

Amortizovaný čas je teda

$$T_{\text{amort}} = T_{\text{skutočný}} + \Delta\Phi = O[(\cancel{t_{\text{pred}}} + \log n) + (t_{\text{po}} - \cancel{t_{\text{pred}}})] = O(\log n),$$

za predpokladu, že 1\$ postačí na zaplatenie $O(1)$ inštrukcií. Počet stromov $t_{\text{pred}}$ sa vyruší a zároveň platí $t_{\text{po}} \leq \log n$. □

## Zhrnutie

V časti o lenivej binomiálnej halde sme videli, že klasickú štruktúru možno ďalej zefektívniť. Kým pri bežnej binomiálnej halde trvá spájanie $O(\log n)$ (čo je už samo o sebe pokrok oproti binárnej halde), stále to nie je ideálne. Myšlienka lenivého prístupu je jednoduchá: spájanie spravíme okamžite v čase $O(1)$, ale „upratovanie“ (zlúčenie stromov rovnakého rádu) odložíme na neskôr.

Takto síce získame $O(1)$ čas pre operácie `merge` aj `insert`, no v halde sa hromadí neporiadok (viaceré stromy rovnakého rádu). To znamená, že neskoršie upratovanie môže stáť až lineárny čas. Z amortizovaného hľadiska sa však ukáže, že cena je iba logaritmická: pred každou drahou operáciou predchádzalo veľa lacných, takže si dokážeme potrebné zdroje postupne „našetriť“.

Lenivá binomiálna halda je pekný príklad toho, že ak si dovolíme odložiť časť práce na neskôr a pozrieme sa na zložitosť z pohľadu amortizovanej analýzy, môžeme získať podstatne rýchlejšie priemerné časy operácií, ako keby sme sa snažili štruktúru neustále udržiavať úplne upratanú.

A to nie je všetko. V nasledujúcej kapitole sa pozrieme na to, ako zlepšiť operáciu `decrease-key`.

| **Operácia** | **Binomiálna halda** | **Lenivá binomiálna halda** |
|---|---|---|
| insert | $O(\log n)$ | $O(1)$ |
| merge | $O(\log n_1 + \log n_2)$ | $O(1)$ |
| get-min | $O(1)$ | $O(1)$ |
| extract-min | $O(\log n)$ | $O(\log n)$ amort. |
| decrease-key | $O(\log n)$ | $O(\log n)$ |

# Kapitola 6

# Fibonacciho halda

V predchádzajúcej kapitole sme videli, že binomiálna halda a jej lenivá verzia umožňujú veľmi efektívne operácie spájania a vkladania. Otázka teda znie: Dá sa to lepšie? Vieme aj `decrease-key` urýchliť na konštantný čas?

Na prvý pohľad to vyzerá beznádejne: stromy môžu mať logaritmickú hĺbku, takže ak chceme znížený kľúč „prebublať“ až do koreňa, bude to trvať pomerne dlho. Napriek tomu odpoveď znie: áno, dá sa to.

Mohli by sme si hneď ukázať finálne riešenie, no bolo by to trochu neuspokojivé. Prečo práve takto? Prečo tak komplikovane? Ako na to vôbec niekto prišiel? Aby sme pochopili motiváciu, ukážeme si najskôr jeden *neúspešný pokus*. Je poučný a zároveň nám pomôže pochopiť, aké prekážky treba pri operácii `decrease-key` prekonať.

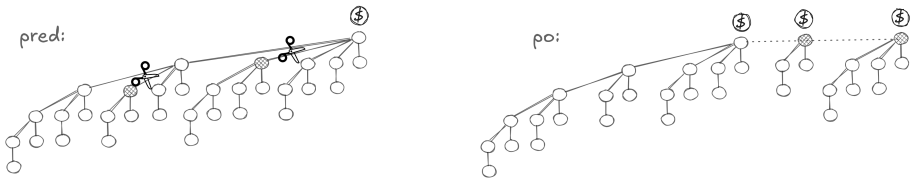
## 6.1 Takto to nejde

Prvý nápad, ako sa vyhnúť pomalému bublaniu, je naplno prijať lenivý prístup a odkladanie práce na neskôr. Ak v nejakom vrchole $x$ znížime kľúč a ten sa stane menším než kľúč jeho rodiča, poruší sa haldovité usporiadanie. S vrcholom $x$ teda musíme niečo urobiť. Čo keby sme ho jednoducho *odstrihli* aj s celým jeho podstromom a presunuli ho medzi korene? Pri najbližšom upratovaní sa stromy s rovnakým rádom opäť pospájú do väčších a štruktúra sa obnoví.

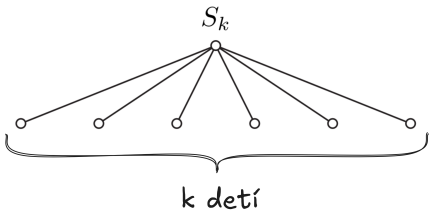
Jednoduché. Za túto operáciu si vyúčtujeme 2$. Jeden dolár pokryje samotnú prácu (odrezanie vrcholu a jeho pripojenie medzi korene) a druhý dolár uložíme novému koreňu na účet, aby sme zachovali invariant. Zvyšok dôkazu pre operáciu `extract-min` bude prebiehať úplne rovnako ako v predchádzajúcej kapitole (viď amortizovaná analýza lenivej binomiálnej haldy).

Je tento dôkaz správny? Ak nie, kde presne sa to pokazí? A ak táto štruktúra nedosahuje želaný výkon, aká je skutočná zložitosť `extract-min`? Skúste sa nad tým na chvíľu zamyslieť, než si pozriete riešenie.

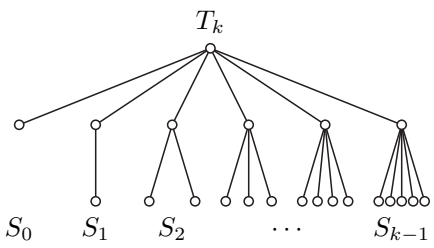
V prvom rade si všimnime, že po odrezaní nejakého podstromu už zvyšok pôvodného stromu nespĺňa definíciu binomiálneho stromu. Predstavme si extrémny




Obr. 6.1: Prvý nápad: Pri `decrease-key` vrchol jednoducho odstrihneme a pridáme medzi korene. Na obrázku je príklad `decrease-key` na dva označené vrcholy.




(a) Strom $S_k$ definujeme ako koreň a pod ním $k$ vrcholov.



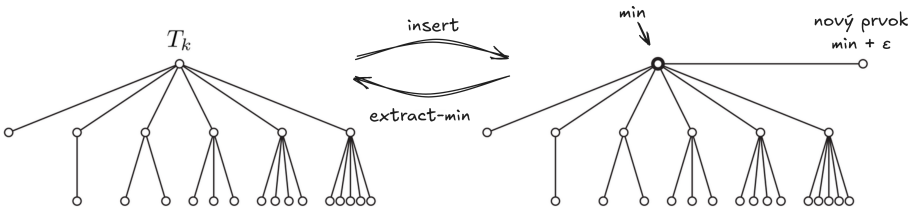

(b) Strom z $T_k$ rádu $k$ sa skladá z koreňa a jeho deti sú $S_0, S_1, \ldots, S_{k-1}$. Tento strom má rádovo $\Theta(k^2)$ vrcholov.

príklad: začneme s binomiálnym stromom rádu $k$ a koreňu postupne (pomocou operácie `decrease-key`) odstrihneme všetkých vnukov. Výsledkom je strom $S_k$ (pozri obr. 6.2a).

Zatiaľ čo v binomiálnej halde mal strom rádu $k$ výšku $k$ a obsahoval $2^k$ vrcholov, strom $S_k$ má výšku 1 a obsahuje len $k + 1$ vrcholov. Tento problém pretrvá aj po upratаní. V binomiálnej halde spájame stromy rovnakého rádu a výsledkom je strom dvojnásobnej veľkosti. Po rezoch však veľkosť stromov už nezodpovedá ich rádu a zlučovanie preto nezaručuje zdvojnásobenie veľkosti. Dôsledky sú zásadné: nedokážeme garantovať, že maximálny rád bude logaritmický, že každý strom má len logaritmický počet detí, alebo že štruktúra sa skladá len z logaritmického počtu stromov.

„A nedá sa ten dôkaz nejako zachrániť?“ povie si optimista. Možno keby sme upravili invarianty, šetrili inak alebo úplne zmenili analýzu, dalo by sa *nejakým spôsobom* dokázať, že zložitosť `extract-min` je logaritmická. Odpoveď je však opäť nie.

Tu je konkrétny protipríklad (pozri obr. 6.3): Nech $T_k$ je strom, ktorý sa skladá z koreňa a jeho deťmi sú stromy $S_0, S_1, \ldots, S_{k-1}$ (pozri obr. 6.2b). Takýto strom $T_k$ má $\Theta(k^2)$ vrcholov a vieme ho zostrojiť pomocou $O(k^3)$ operácií. Všimnime si, že ak pripojíme $S_k$ pod koreň $T_k$, dostaneme strom $T_{k+1}$ s rádom




Obr. 6.3: Protipríklad, ktorý ukazuje, že pri naivnom odstrihávaní vrcholov trvá operácia `extract-min` aspoň $\Omega(\sqrt{n})$, a to aj v amortizovanom zmysle.

o jeden vyšším.

Uvažujme teraz nasledujúcu postupnosť operácií: Začneme so stromom $T_k$, kde koreň má hodnotu 0 a všetky ostatné vrcholy majú kľúče väčšie než, povedzme, milión. Do haldy vložíme nový kľúč, ktorý je len o $\varepsilon$ väčší než hodnota v koreni, ale zároveň menší než hodnoty všetkých jeho synov. Výsledkom je halda zložená zo stromu $T_k$ a jedného nového vrcholu, teda $T_0$.

Ak teraz zavoláme `extract-min`, najmenším prvkom je koreň $T_k$. Ten odstránime a jeho synov $S_0, S_1, \ldots, S_{k-1}$ vložíme medzi korene. Pri následnom upratovaní sa postupne pospájajú všetky tieto stromy s $T_0$. Kľúč sme zvolili tak, aby $T_0$ vyhral pri každom porovnaní, takže stromy $S_0, \ldots, S_{k-1}$ sa jeden po druhom pripoja pod neho. Výsledkom je séria spojení: $T_0 + S_0 \to T_1$, $T_1 + S_1 \to T_2$, a tak ďalej, až kým opäť nedostaneme strom $T_k$ s rovnakým tvarom, len s koreňom o trochu väčším.

Dostali sme sa späť do pôvodného stavu, ale algoritmus pritom vykonal $\Theta(k) = \Theta(\sqrt{n})$ operácií, čo je priveľa. Túto dvojicu operácií môžeme opakovať ľubovoľne dlho, a preto je amortizovaná zložitosť `insert` + `extract-min` aspoň $\Omega(\sqrt{n})$. Ak by sme chceli dokázať, že je to menej, museli by sme zvyšnú prácu do $\sqrt{n}$ zakaždým doplácať z našetrených dolárov. To sa však nedá robiť donekonečna – skôr či neskôr by sme zbankrotovali.

Mimochodom, dá sa pomerne jednoducho ukázať, že $\sqrt{n}$ je nielen dolný, ale aj horný odhad pre zložitosť operácie `extract-min`. Dostávame tak haldu, ktorá podporuje všetky operácie v čase $O(1)$, až na `extract-min`, ktorá má zložitosť $O(\sqrt{n})$.

Poučenie, ktoré si z tohto neúspešného pokuku odnášame, je jasné: nechceme mať stromy s vysokým rádom, ktoré pritom obsahujú len málo vrcholov. Ak má byť štruktúra rýchla, musíme zabezpečiť, aby každý strom obsahoval *exponenciálne veľa vrcholov vzhľadom na svoj rád*. Inak povedané: pre strom rádu $k$ potrebujeme, aby mal aspoň $c^k$ vrcholov pre nejakú konštantu $c > 1$. Z toho priamo plynie, že maximálny rád stromu v halde s $n$ prvkami je najvýš $\log_c n$ a po upratаní sa celá halda skladá z najviac $\log_c n$ stromov. Práve tieto vlastnosti sú kľúčové v dôkaze, že operácia `extract-min` má v lenivej binomiálnej halde logaritmickú amortizovanú zložitosť.

Skúsme to teda z opačnej strany: ku každému vrcholu si budeme pamätať veľkosť jeho podstromu a *rád* zadefinujeme ako logaritmus tejto veľkosti (zaokrúhlený nadol). Týmto zaručíme, že veľkosť podstromu rastie exponenciálne s rádom jeho koreňa.

Má to však háčik: pri každom `decrease-key`, keď nejaký vrchol odstrihneme, by sme museli aktualizovať veľkosti všetkých predkov na ceste ku koreňu. To je znova *logaritmicky veľa práce* – presne toľko, ako keby sme vrchol prebublávali nahor.

## 6.2 Riešenie: Kaskádové rezy

Prvý známy spôsob, ako zvládnuť operáciu `decrease-key` v konštantnom a zároveň `extract-min` v logaritmickom čase, navrhli Michael L. Fredman a Robert E. Tarjan v roku 1984. Ich technika dostala názov *kaskádové rezy* (*cascading cuts*).

Odvtedy vzniklo viacero ďalších prístupov, ktoré dosahujú rovnaký cieľ (Driscoll et al. 1988; Kaplan a Robert Endre Tarjan 2008; Elmasry 2010; Haeupler, Sen a Robert Endrendre Tarjan 2011; Chan 2013; Kaplan, Robert Endre Tarjan a Zwick 2014; Hansen et al. 2017; Elmasry, Jensen a Katajainen 2008; Brodal 1996; Brodal a Okasaki 1996; Brodal, Lagogiannis a Robert Endre Tarjan 2012). My si však ukážeme práve pôvodný nápad Fredmana a Tarjana.

Definujeme *rád* vrcholu ako *počet jeho detí*. Myšlienka je nasledovná: Pri `decrease-key` vrchol odstrihneme a pripojíme do zoznamu koreňov. Zároveň si však pre jeho otca poznačíme, že sme mu odrezali syna. Budeme dodržiavať invariant, že každý vrchol okrem koreňov má najviac jedného odrezaného syna. Každý vrchol (okrem koreňa), ktorému odrežeme dvoch synov, odrežeme tiež a pripojíme ho ku koreňom.

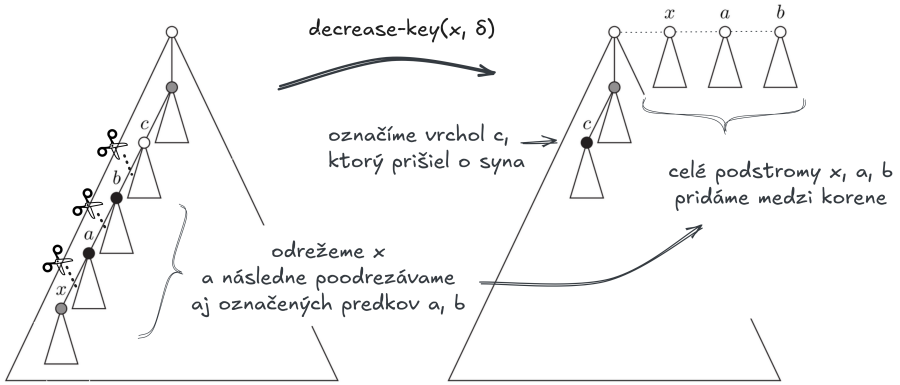
Jedna operácia `decrease-key` teda môže spustiť celú kaskádu rezov (pozri obr. 6.4). V najhoršom prípade môže takáto kaskáda trvať až logaritmický čas (úmerný hĺbke vrcholu). Ukážeme si však, že amortizovaná zložitosť ostáva konštantná.

Následne si ukážeme, že vďaka invariantu, že každý vrchol prišiel najviac o jedného syna, zostávajú stromy dostatočne „husté“ a ich veľkosť bude exponenciálna vzhľadom na počet detí koreňa.

Na prvý pohľad to môže pôsobiť paradoxne: pri operáciach `decrease-key` odrezávame podstromy, pritom sa chceme vyhnúť stromom s veľkým rádom a malým počtom vrcholov. Riešením má byť odrezať *ešte viac* podstromov?

Keď odrežeme veľa podstromov, strom sa prirodzene zmenší, a preto by mal mať aj menší rád. Potrebujeme teda, aby sa tento „signál“ – informácia o odstránených vrcholoch – prenášal smerom nahor, ku koreňu. A práve na to slúžia kaskádové rezy: ak odstránime príliš veľa vrcholov, budeme nútení postupne odstrániť aj syna samotného koreňa. Keďže *rád* vrcholu definujeme ako *počet jeho detí*, rád koreňa (a teda aj celého stromu) sa primerane zníži.

To je aspoň intuícia, ktorá stojí za kaskádovými rezmi. Teraz sa už môžeme pozrieť na formálne dôkazy, ktoré ukážu, že operácia `decrease-key` je skutočne




Obr. 6.4: Priklad operácie `decrease-key`$(x, \delta)$. Čierne vrcholy $a$ a $b$ majú poznačené, že už prišli o jedného syna, biele vrcholy majú všetkých synov a v prípade šedých vrcholov na obrázku je to nepodstatné pre túto operáciu. Odstrihnutie $x$ spôsobí kaskádu: $a$ stratí druhého syna, takže ho odrežeme a tým pádom aj $b$ stratí druhého syna a aj tento podstrom odrežeme. Zastavíme sa až na prvom bielom vrchole $c$, ktorý, keďže prišiel o syna, ofarbíme na čierno.

konštantná z amortizovaného hľadiska a že `extract-min` si zachováva logaritmickú zložitosť.

**Veta 6.1.** *Operácia* `decrease-key` *s kaskádovými rezmi trvá* $O(1)$ *amortizovane.*

■ **Dôkaz #1.** Použijeme účtovnícku metódu. Za každú operáciu si vypýtame 4$.

**Invariant.** Každý označený vrchol (teda taký, ktorému už bol odstrihnutý jeden syn) má nasporené 2$ na zaplatenie prípadného budúceho kaskádového rezu. Navyše tak ako v predošlej kapitole, každý koreň má odložený 1$, ktorým prispieva na upratovanie.

**Vyúčtovanie.** Pri jednej operácii `decrease-key`:

- 1$ použijeme na samotnú prácu ($O(1)$ úkonov: odstrihnutie, pripojenie medzi korene, označenie otca),
- 1$ odložíme na účet tohto nového koreňa,
- 2$ pošleme jeho otcovi.

Ak mal otec doteraz všetkých synov, teraz má na účte 2$, invariant zostáva zachovaný a operácia končí.

Ak mu už jeden syn chýbal, mal uložené 2$ a práve dostal ďalšie 2$ – spolu 4$. Tie mu postačia na zaplatenie vlastného rezu:

- 1$ použije na samotnú prácu,
- 1$ si nechá ako nový koreň,
- a zvyšné 2$ posunie *svojmu* otcovi.

Takto pokračuje celá kaskáda: Každý označený vrchol si odrezanie pokryje z vlastných nasporených peňazí a zároveň pošle 2$ vyššie. Proces pokračuje, až kým nenarazíme na koreň, alebo na neoznačený vrchol, ktorý iba označíme a prispejeme mu 2$. V oboch prípadoch sa operácia končí a invarianty ostanú zachované. □

Ten istý dôkaz pomocou potenciálovej metódy:

■ **Dôkaz #2.** Zvoľme si potenciál haldy ako

$$\Phi(H) = \textit{počet koreňov} + 2 \times \textit{počet označených vrcholov}.$$

Ak počas jednej operácie odrežeme $k$ vrcholov, reálna práca zaberie $O(k)$ času. Pozrime sa, ako sa zmení potenciál:

| | |
|---|---|
| $+k$ | pretože pribudne $k$ nových koreňov, |
| $-2 \times (k-1)$ | pretože okrem prvého vrcholu boli všetky ostatné označené a po odrezaní už nie sú, |
| $+2$ | pretože na konci kaskády jeden označený vrchol pribudol. |

Celkovo potenciál klesne o $k-4$, takže

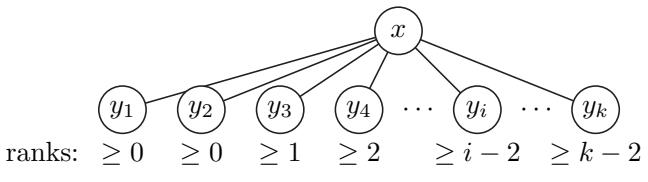
$$T_{\text{amort}} = T_{\text{skutočný}} + \Delta\Phi = O(k) - (k-4)\$ = O(1).$$ □

Z hľadiska amortizovanej analýzy sú teda kasakádové rezy „zadarmo“ – vrcholy dokážu zaplatiť samy za seba.

## 6.3 Veľkosť stromov

Ostáva nám ešte ukázať, že pri takomto urezávaní bude mať koreň rádu $k$ stále exponenciálne veľa vrcholov v závislosti od $k$.

**Lema 6.1.** *Nech vrchol $x$ má synov $y_1, \ldots, y_m$, pričom ich označíme v poradí, v akom sa pripojili ku $x$. Potom platí rád$(y_i) \geq i-2$, teda tretí syn má aspoň jedného syna, štvrtý syn aspoň dvoch, a tak ďalej.*

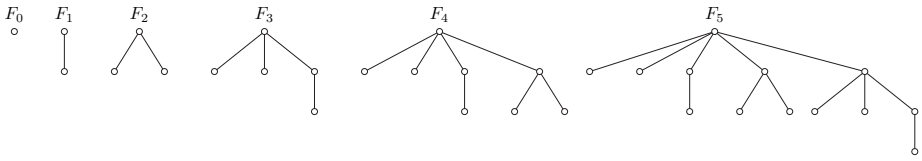


■ **Dôkaz.** Vráťme sa naspäť do momentu, keď sme pripájali $y_i$ pod $x$. V tom čase mal vrchol $x$ už aspoň $i-1$ detí (konkrétne $y_1, \ldots, y_{i-1}$; prípadne aj ďalšie, ktoré sa neskôr odrezali, ale minimálne tieto zostali zachované).

Pri spájaní však vždy spájame iba stromy s rovnakým rádom, a teda *$y_i$ v tom čase muselo mať tiež aspoň $i-1$ detí*. Odvtedy mu mohol ubudnúť nanajvýš jeden syn (v opačnom prípade by sme ho takisto odrezali) a preto v súčasnosti platí rád$(y_i) \geq i-2$. □

Túto lemu môžeme aplikovať rekurzívne na všetky vrcholy stromu. Zamyslime sa, aká je najmenšia možná veľkosť stromu, ktorého koreň má rád $n$ (t. j. má $n$ detí). Takýto minimálny strom označme $F_n$.

Pre $n = 0, 1, 2$ je $F_n$ len koreň a jeho $n$ synov. A čo $F_3$? Podľa lemy musí mať tretí syn rád aspoň 1, teda má aspoň jedného syna.



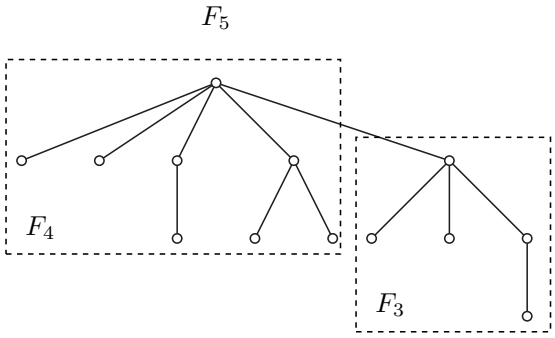

V strome $F_4$ má koreň štyroch synov; tretí má rád aspoň 1 a štvrtý aspoň 2.
V strome $F_5$ pribudne piaty syn s rádom aspoň 3.
Našťastie, najmenšie stromy rádov 1, 2 a 3 sme už identifikovali.

Spočítajme počet vrcholov:

| strom | $F_0$ | $F_1$ | $F_2$ | $F_3$ | $F_4$ | $F_5$ |
|---|---|---|---|---|---|---|
| #vrcholov | 1 | 2 | 3 | 5 | 8 | 13 |

Hmmmm... Náhoda? Nemyslím si.

Veľkosť stromu $F_n$ je $(n+1)$-vé Fibonacciho číslo (odtiaľ pochádza názov Fibonacciho halda). Dôvod je jedoduchý: Najmenší strom rádu $n$ sa skladá z koreňa a jeho $n$ detí. Prvých $n-1$ synov vyzerá rovnako ako v strome $F_{n-1}$. Posledný, $n$-ty syn, má podľa lemy rád aspoň $n-2$, a teda najmenší taký strom je práve $F_{n-2}$. Z toho vyplýva, že strom $F_n$ vznikne pripojením $F_{n-2}$ ku $F_{n-1}$, čo je rovnaká rekurencia, akú spĺňajú Fibonacciho čísla.



Fibonacciho čísla rastú exponenciálne rýchlo. Asi najjednoduchšie to vidno z rekurencie

$$F_n = F_{n-1} + F_{n-2} \geq 2 \times F_{n-2},$$

čo znamená, že $F_n \geq 2^{n/2} = (\sqrt{2})^n$. Presný odhad je $F_n = \Theta(\phi^n)$, kde $\phi = (1+\sqrt{5})/2 \approx 1.618$ je hodnota zlatého rezu.

Odtiaľ vyplývajú nasledovné tvrdenia:

**Veta 6.2.** *Každý vrchol rádu $k$ má pod sebou* exponenciálne veľa *vrcholov v závislosti od $k$; presnejšie, aspoň $\Omega(\phi^k)$ vrcholov. (Rád vrcholu je definovaný ako počet jeho detí.)*

**Veta 6.3.** *Vo Fibonacciho halde majú operácie nasledujúcu amortizovanú zložitosť:*

- *`insert`, `merge` a `decrease-key`* – $O(1)$,
- *`extract-min`* – $O(\log n)$.

■ **Dôkaz.** Veľkosť stromu rádu $k$ je aspoň $\phi^k$, takže maximálny možný rád je logaritmický (pri základe $\phi$ namiesto 2):

$$\log_\phi n \leq \lg n / \lg \phi \leq 1.45 \lg n.$$

Každý koreň má teda najviac $O(\log n)$ detí a po upratаní ostane najviac $O(\log n)$ stromov. Zvyšok argumentu je rovnaký ako pri lenivej binomiálnej halde (líšia sa len konštanty). □

# Referencie


Brodal, Gerth Stølting (1996). „Worst-case efficient priority queues“. In: SODA.

Brodal, Gerth Stølting, George Lagogiannis a Robert Endre Tarjan (2012). „Strict fibonacci heaps“. In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, s. 1177–1184.

Brodal, Gerth Stølting a Chris Okasaki (1996). „Optimal purely functional priority queues“. In: *Journal of Functional Programming* 6.6, s. 839–857.

Driscoll, James Robert et al. (1988). „Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation“. In: *Communications of the ACM* 31.11, s. 1343–1354.

Elmasry, Amr (2010). „The violation heap: A relaxed Fibonacci-like heap“. In: *International Computing and Combinatorics Conference*. Springer, s. 479–488.

Elmasry, Amr, Claus Jensen a Jyrki Katajainen (2008). „Two-tier relaxed heaps“. In: *Acta Informatica* 45.3, s. 193–210.

Haeupler, Bernhard, Siddhartha Sen a Robert Endrendre Tarjan (2011). „Rank-pairing heaps“. In: *SIAM Journal on Computing* 40.6, s. 1463–1485.

Hansen, Thomas Dueholm et al. (2017). „Hollow heaps“. In: *ACM Transactions on Algorithms (TALG)* 13.3, s. 1–27.

Chan, Timothy Moon-Yew (2013). „Quake heaps: A simple alternative to Fibonacci heaps“. In: *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Springer, s. 27–32.

Kaplan, Haim a Robert Endre Tarjan (2008). „Thin heaps, thick heaps“. In: *ACM Transactions on Algorithms (TALG)* 4.1, s. 1–14.

Kaplan, Haim, Robert Endre Tarjan a Uri Zwick (2014). „Fibonacci heaps revisited“. In: *arXiv preprint arXiv:1407.5750*.

# Kapitola 7

# Najlacnejšia kostra

ukážeme si aplikáciu Fibonacciho haldy, ukážeme si rýchlejší algoritmus ako tie, čo poznáte. napínavý príbeh, ktorého rozuzlenie stále nepoznáme. fascinujúci príbeh, ktorý začína na území a v období Prvej česko-slovenskej republiky

Problém najlacnejšej kostry (*Minimum Spanning Tree*, MST) patrí k najzákladnejším úlohám v teórii grafov. Dostaneme neorientovaný ohodnotený graf a našou úlohou je nájsť podgraf, ktorý spája všetky vrcholy, je stromom (t. j. neobsahuje cyklus) a má minimálny možný súčet hrán.

Predpokladám, že poznáte Kruskalov alebo Primov algoritmus, takže možno máte pocit, že táto úloha je prakticky vyriešená, takže v tejto kapitole vás vyvediem z omylu. Výskumníci postupne objavili rýchlejšie a rýchlejšie algoritmy[1], ktoré najlacnejšiu kostru nájdu v takmer-lineárnom čase. Jeden taký algoritmus si tu ukážeme. Dodnes však nevieme, aká je zložitosť tohto problému. Nevieme, či existuje lineárny algoritmus V skutočnosti

## 7.1 Malá historická vsuvka

Z teoretického hľadiska však

Hľadanie minimálnej kostry je fascinujúci príbeh, ktorý začína v období Prvej česko-slovenskej republiky.

Borůvkov popis je veľmi technický a ťažkopádny – neexistovali totiž ešte pojmy ako graf, cesta, strom, komponent... Treba si uvedomiť, že v tej dobe ešte neexistovala teória grafov ako samostatné odvetvie matematiky. Prvú učebnicu teórie grafov napísal Dénes König až o 10 rokov neskôr. Takisto ešte neexistovala vedná disciplína menom informatika. Bolo to 10 rokov predtým ako Alan Turing definuje Turingove stroje, a dávno predtým ako Hartmanis a Stearns zavedú výpočtovú zložitosť v 1965 a Donald Knuth spopularizuje analýzu algoritmov v 1969.

[1] tu máme na mysli teoretické riešenia s lepšou asymptotickou zložitosťou, nie implementácie, ktoré sú rýchle v praxi

| MST algoritmus | Časová zložitosť |
|---|---|
| Borůvka 1926 | $O(m \log n)$ |
| Jarník 1930; Prim 1957; Dijkstra 1959 | $O(n^2)$ |
| ... s binárnou haldou | $O(m \log n)$ |
| ... s $d$-árnou haldou | $O(m \log_{\max(2,m/n)} n)$ |
| ... s Fibonacciho haldou | $O(m + n \log n)$ |
| Kruskal 1956 | $O(m \log n)$ |
| ...s Union-Find Robert Endre Tarjan 1979 | $O(m\alpha(m,n))$, ak sú hrany utriedené |
| Yao 1975 | $O(m \log \log n)$ |
| Cheriton a Robert Endre Tarjan 1976 | $O(m \log \log n)$; viacero špeciálnych prípadov v $O(m)$ |
| Fredman a Robert Endre Tarjan 1987 | $O(m \log^* n)$ |
| Gabow et al. 1986 | $O(m \log(\log^* n))$ |
| Dixon, Rauch a Robert E Tarjan 1992; King 1995 | verifikácia v $O(m)$ |
| Karger, Klein a Robert Endre Tarjan 1995 | $O(m)$, ale randomizovane |
| Chazelle 2000; Pettie 1999 | $O(m\alpha(m,n))$ |
| Pettie a Ramachandran 2002 | optimálna v porovnávacom modeli |

Tabuľka 7.1: Krátka história vývoja čoraz rýchlejších algoritmov pre hľadanie najlacnejšej kostry.

V zime 1925/26 sa Borůvka stretol s Jindřichom Saxelom Západomorovské elektrárny. Elekrifikácia južnej Moravy.

## 7.2 Klasické riešenia

Modro-červený meta-algoritmus

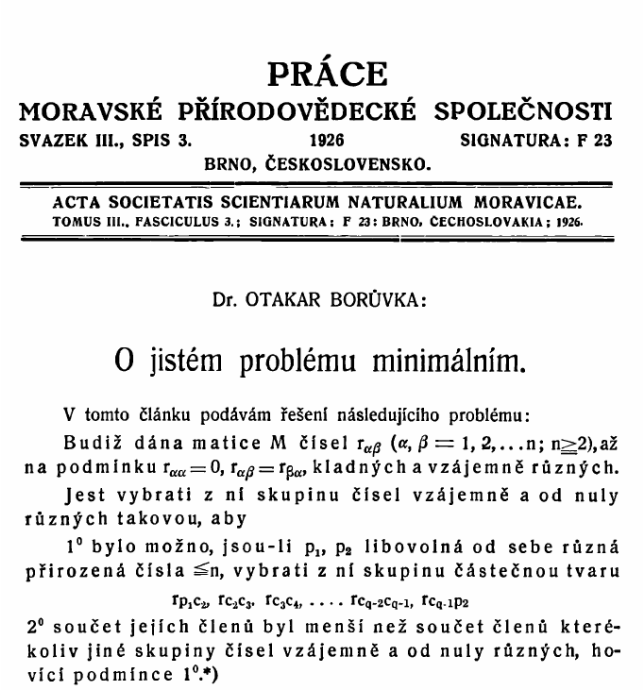
Pravidlo rezu: Vyberieme rez $C$, ktorého najlacnejšia hrana nie je modrá a zafarbíme ju namodro. Pravidlo cyklu: Vyberieme cyklus $C$, ktorého najdrahšia hrana nie je červená a zafarbíme ju načerveno.

## 7.3 Fredmanov-Tarjanov algoritmus

V tejto sekcii si ukážeme len o chlp horší ako lineárny algoritmus bežiaci v čase $O(m \log^* n)$. Algoritmus strieda „Jarníkove“ kroky, kde začíname z rôznych vrcholov a zárodky minimálnej kostry rozrastáme pripájaním najlacnejších susedov, a „Borůvkove“ kroky, kde tieto stromy skontrahujeme do vrcholov.

Pomalá časť pri Jarníkovom algoritme aj s Fibonacciho haldou je výber minima. Problémom je veľkosť haldy: ak do nej nasúkame $\Omega(n)$ prvkov, výber minima trvá $O(\log n)$ a túto zložitosť už veľmi zlepšiť nevieme (pretože inak by sme vedeli triediť lepšie ako v $O(n \log n)$).

Fredman a Tarjan prišli s myšlienkou, ako sa tomuto problému vyhnúť: obmedzíme veľkosť haldy na maximálne $k$ prvkov. Začneme budovať kostru



**PRÁCE**
**MORAVSKÉ PŘÍRODOVĚDECKÉ SPOLEČNOSTI**
SVAZEK III., SPIS 3. 1926 SIGNATURA: F 23
BRNO, ČESKOSLOVENSKO.

ACTA SOCIETATIS SCIENTIARUM NATURALIUM MORAVICAE.
TOMUS III., FASCICULUS 3.; SIGNATURA: F 23: BRNO, ČECHOSLOVAKIA; 1926.

Dr. OTAKAR BORŮVKA:

## O jistém problému minimálním.

V tomto článku podávám řešení následujícího problému:

Budiž dána matice M čísel $r_{\alpha\beta}$ ($\alpha, \beta = 1, 2, \ldots n$; $n \geqq 2$), až na podmínku $r_{\alpha\alpha} = 0$, $r_{\alpha\beta} = r_{\beta\alpha}$, kladných a vzájemně různých.

Jest vybrati z ní skupinu čísel vzájemně a od nuly různých takovou, aby

1° bylo možno, jsou-li $p_1$, $p_2$ libovolná od sebe různá přirozená čísla $\leqq n$, vybrati z ní skupinu částečnou tvaru

$$r_{p_1c_2},\ r_{c_2c_3},\ r_{c_3c_4},\ \ldots.\ r_{c_{q-2}c_{q-1}},\ r_{c_{q-1}p_2}$$

2° součet jejích členů byl menší než součet členů kterékoliv jiné skupiny čísel vzájemně a od nuly různých, hovící podmínce 1°.*)

Obr. 7.1: Úplne prvý článok, ktorý formuloval problém hľadania minimálnej kostry a navrhol prvé riešenie, je od Otakara Borůvku z roku 1926.

## Příspěvek k řešení otázky ekonomické stavby elektrovodných sítí

Dr. Otakar Borůvka

Ve své práci „O jistém problému minimálním“ *) odvodil jsem obecnou větu, jíž jest ve zvláštním případě řešena tato úloha:

V rovině (v prostoru) jest dáno n bodů, jejichž vzájemné vzdálenosti jsou vesměs různé. Jest je spojiti sítí tak, aby

1. každé dva body byly spojeny buď přímo anebo prostřednictvím jiných,
2. celková délka sítě byla co nejmenší.

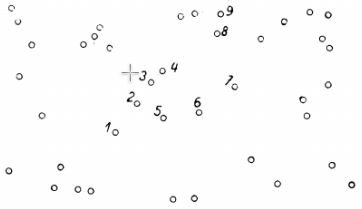
Jest zřejmé, že řešení této úlohy může míti v elektrotechnické praksi při návrzích plánů elektrovodných sítí jistou důležitost; z toho důvodu je zde stručně na příkladě vyložím. Čtenáře, jenž by se o věc blíže zajímal, odkazuji na citované pojednání.

*) Vyjde v nejbližší době v Pracích Moravské přírodovědecké společnosti.

*) Kesl: Theorie a prakse oteplování a přechodového odporu dotyků vypinače (čís. 45).

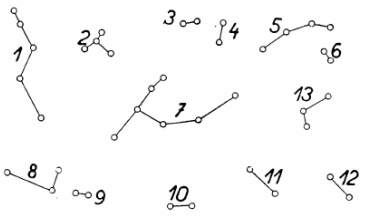


Řešení úlohy provedu v případě 40 bodů daných v obr. 1.



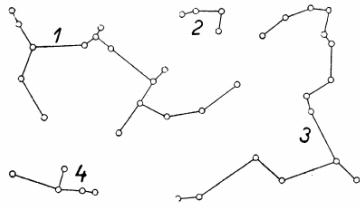
Obr. 1.

Každý z daných bodů spojím s bodem nejbližším. Tedy na př. bod 1 s bodem 2, bod 2 s bodem 3, bod 3 s bodem 4 (bod 4 s bodem 3), bod 5 s bodem 2, bod 6 s bodem 5, bod 7 s bodem 6, bod 8 s bodem 9, (bod 9 s bodem 8) atd. Obdržím řadu polygonálních tahů 1, 2,.... 13 (obr. 2).



Obr. 2.

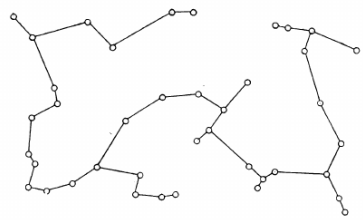
Každý z nich spojím nejkratším způsobem s tahem nejbližším. Tedy na př. 1 s tahem 2, (tah 2 s tahem 1), tah 3 s tahem 4, (tah 4 s tahem 3) atd. Obdržím řadu polygonálních tahů 1, 2,.... 4 (obr. 3).



Obr. 3.

Každý z nich spojím nejkratším způsobem s tahem nejbližším. Tedy tah 1 s tahem 3, tah 2 s tahem 3 (tah 3 s tahem 1), tah 4 s tahem 1. Obdržím konečně jediný polygonální tah (obr. 4), jenž řeší danou úlohu.

Matematický ústav Masarykovy university v Brně, v lednu 1926.



Obr. 4.

Obr. 7.2: Sprievodný článok v časopise *Elektrotechnický obzor*. V skratke, píše sa v ňom zhruba toto: „Počujte, keby ste chceli budovať nejaké siete – elektrické alebo vodovodné, to je jedno – a potrebujete prepojiť všetky mestá, tak ja som už vyriešil, že ako to spraviť za čo najmenšiu cenu. Mrknite na tento matický článok [link na *O jistém probému minimálním*].“ Mimochodom, pri publikácií sa stala tlačová chyba a obrázok 4 je obrátený.

z jedného vrcholu Jarníkovým algoritmom. Akonáhle sa halda „preplní“, zastavíme a spustíme Jarníkov algoritmus z *iného* vrcholu – začneme odznovu s prázdnou haldou. Takto pokračujeme, až kým nie je každý vrchol zaradený do niektorého stromu. Celkový čas tohto kroku je $O(m + n \log k)$, pretože halda má maximálnu veľkosť $k$.

Následne prejdeme celý graf a každý z vytvorených stromov (čiastočných kostier) kontrahujeme do jedného vrcholu. Ak medzi dvoma stromami vedie viacero hrán, ponecháme len tú najlacnejšiu. Táto kontrakcia sa dá urobiť v lineárnom čase $O(m)$.

Tieto dva kroky: vytvorenie častí kostry Jarníkovým algoritmom a následná kontrakcia tvoria jednu fázu algoritmu (pozri obrázok 7.3). Výsledkom je menší graf, na ktorom proces opakujeme, až kým neprepojíme všetky vrcholy.

Otázka teraz znie, aké $k$ by sme mali zvoliť. Chceme také „akurát“ – ani príliš malé, ani príliš veľké. Na jednej strane, čím je $k$ menšie, tým je menšie $n \log k$, čiže jedna fáza je rýchlejšia. Naopak, čím je $k$ väčšie, tým sa stromy v jednej fáze viac rozrastú, takže ich bude menej. Tým pádom v nasledujúcej fáze bude menej vrcholov a celkovo budeme mať menej fáz.

Vhodný kompromis je voľba je $k = 2^{2m/n}$, pretože vtedy je $n \log k = n \log(2^{2m/n}) = n \times (2m/n) = \Theta(m)$. To znamená, pri tejto voľbe bude jedna fáza trvať lineárny čas.

Koľko fáz bude treba?

Označme $n$, $m$ počet vrcholov a hrán v súčasnej a $n'$ a $m'$ počet vrcholov a hrán v *nasledujúcej* fáze. Počet vrcholov $n'$ je rovný počtu stromov, ktoré v jednej fáze vytvoríme. A aký bude počet hrán $m'$? Na konci fázy má každý strom $T$ aspoň $k$ hrán začínajúcich v $T$ (pričom každá hrana má dva konce). To znamená, že v nasledujúcej fáze bude počet hrán aspoň $m' \geq k \times n'/2$. Ak nerovnosť preusporiadame, dostaneme $2m'/n' \geq k$. Z toho ale vyplýva, že v nasledujúcej fáze si zvolíme $k' = 2^{2m'/n'} \geq 2^k$, t.j. exponenciálne väčšie!

Nech $k_i$ je hodnota $k$ v $i$-tej fáze. Potom $k_0 = 2^{2m/n} = \Omega(1)$, $k_1 \geq 2^{k_0}$, $k_2 \geq 2^{2^{k_0}}$, $k_3 \geq 2^{2^{2^{k_0}}}$, atď. V momente, keď hodnota $k_i$ presiahne aktuálny počet hrán, znamená to, že *všetky* hrany sa už zmestia do haldy a tým pádom v Jarníkovom kroku pospájame všetky vrcholy a algoritmus končí.

Celkovo teda bude najviac $\log^* m = \log^* n$ fáz a celkový čas je $O(m \log^* n)$.

# Referencie

Borůvka, Otakar (1926). „O jistém problému minimálním.“ In: *Práce Moravské Přírodovědecké Společnosti* 3.3.

Dijkstra, Edsger Wybe (1959). „A Note on Two Problems in Connexion with Graphs“. In: *Numerische Mathematik* 1, s. 269–271.

Dixon, Brandon, Monika Rauch a Robert E Tarjan (1992). „Verification and sensitivity analysis of minimum spanning trees in linear time“. In: *SIAM Journal on Computing* 21.6, s. 1184–1192.

Fredman, Michael Lawrence a Robert Endre Tarjan (1987). „Fibonacci heaps and their uses in improved network optimization algorithms“. In: *Journal of the ACM (JACM)* 34.3, s. 596–615.

Gabow, Harold N et al. (1986). „Efficient algorithms for finding minimum spanning trees in undirected and directed graphs“. In: *Combinatorica* 6.2, s. 109–122.

Chazelle, Bernard (2000). „A minimum spanning tree algorithm with inverse-Ackermann type complexity“. In: *Journal of the ACM (JACM)* 47.6, s. 1028–1047.

Cheriton, David a Robert Endre Tarjan (1976). „Finding minimum spanning trees“. In: *SIAM journal on computing* 5.4, s. 724–742.

Jarník, Vojtěch (1930). „O jistém problému minimálním. (Z dopisu panu O. Borůvkovi)“. In: *Práce Moravské Přírodovědecké Společnosti* 6.4.

Karger, David R, Philip N Klein a Robert Endre Tarjan (1995). „A randomized linear-time algorithm to find minimum spanning trees“. In: *Journal of the ACM (JACM)* 42.2, s. 321–328.

King, Valerie (1995). „A simpler minimum spanning tree verification algorithm“. In: *Workshop on Algorithms and Data Structures*. Springer, s. 440–448.

Kruskal, Joseph Bernard (1956). „On the shortest spanning subtree of a graph and the traveling salesman problem“. In: *Proceedings of the American Mathematical society* 7.1, s. 48–50.

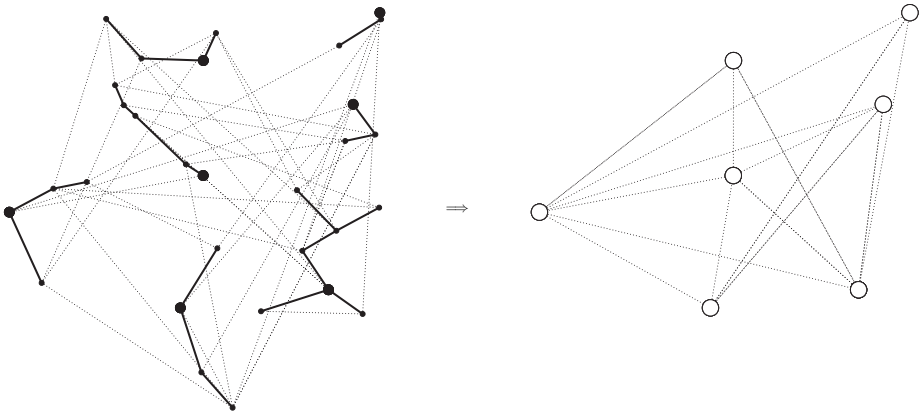
Pettie, Seth (1999). *Finding Minimum Spanning Trees in $O(m\alpha(m,n))$ Time*.

Pettie, Seth a Vijaya Ramachandran (2002). „An optimal minimum spanning tree algorithm“. In: *Journal of the ACM (JACM)* 49.1, s. 16–34.

Prim, Robert Clay (1957). „Shortest connection networks and some generalizations“. In: *The Bell System Technical Journal* 36.6, s. 1389–1401.

Tarjan, Robert Endre (1979). „Applications of path compression on balanced trees“. In: *Journal of the ACM (JACM)* 26.4, s. 690–715.

Yao, Andrew Chi-chih (1975). „An $\Theta(|E| \log\log |V|)$ algorithm for finding minimum spanning trees“. In: *Information Processing Letters* 4.1, s. 21–23.



Obr. 7.3: Jedna fáza Fredmanovho-Tarjanovho algoritmu: Jarníkovým algoritmom spúšťaným z rôznych vrcholov (vľavo) až kým sa halda nepreplní vytvoríme časti minimálnej kostry (tučné hrany). Nasleduje Borůvkov krok: Každý komponent skontrahujeme do jedného vrcholu (vpravo); ak boli časti kostry v pôvodnom grafe spojené viacerými, hranami, v novom grafe budú ponecháme len tú najlacnejšiu. Výsledkom je menší graf,

# Kapitola 8

# Radixová halda

Zložitosť Dijkstrovho algoritmu môžeme zapísať ako

$$O\big(n \times (T_{\texttt{insert}} + T_{\texttt{extract-min}}) + m \times T_{\texttt{decrease-key}}\big).$$

V predchádzajúcich kapitolách sme videli, že vhodná dátová štruktúra vie tento čas výrazne ovplyvniť:

- jednoduché pole: $O\big(n \times (1+n) + m \times 1\big) = O(n^2 + m) = O(n^2)$,
- binárna halda: $O\big(n \times (\log n + \log n) + m \times \log n\big) = O(m \log n)$,
- $d$-árna halda: $O\big(n \times (\log_d n + d \log_d n) + m \times \log_d n\big) = O(m \log_{m/n} n)$,
- Fibonacciho halda: $O\big(n \times (1 + \log n) + m \times 1\big) = O(m + n \log n)$ (teoreticky výborné, v praxi často pomalé kvôli vysokým konštantám, ktoré sa schovávajú v $O$).

Predpokladajme teraz, že všetky dĺžky hrán sú *celočíselné* z rozsahu $[0, 1, \ldots, C]$. Dokážeme využiť túto informáciu a implementovať Dijkstru ešte rýchlejšie?

Extrémny prípad: ak majú všetky hrany dĺžku 1, môžeme použiť prehľadávanie do šírky (BFS), ktoré nájde najkratšiu cestu v lineárnom čase.

Ak sú dĺžky len malé celé čísla, môžeme *hranu* dĺžky $c$ nahradiť *cestou* dĺžky $c$ (t.j. $c$ jednotkovými hranami) a následne opäť použiť BFS. Graf sa tým „nafúkne“ približne $C$-násobne a výsledná zložitosť bude $O(C \times m)$.

Iný nápad: všetky dočasné vzdialenosti ležia v intervale $[0, 1, \ldots, n \times C]$, takže si môžeme udržiavať jednoduché pole „chlievikov“, kde na pozícii $d$ držíme všetky vrcholy s aktuálnou vzdialenosťou $d$. Pole prechádzame zľava doprava: prázdne políčka preskakujeme, a keď narazíme na neprázdne, vrcholy v ňom majú (spomedzi nespracovaných) minimálnu vzdialenosť. Relaxácie len presúvajú vrchol medzi chlievikmi v konštantnom čase. Okrem lineárneho prechodu po poli sú všetky úkony konštantné, takže celková zložitosť je $O(m + n \times C)$.

> Dá sa to lepšie? (Lineárna závislosť od $C$ činí algoritmus neprakický pre väčšie $C$.)

Všimnime si dve kľúčové vlastnosti, ktoré súvisia s tým, ako Dijkstrov algoritmus využíva svoju haldu:

1. **Neklesajúce vzdialenosti pri `extract-min`:** Dijkstrov algoritmus vždy spracúva vrcholy v poradí rastúcich vzdialeností. Akonáhle spracujeme vrchol s vzdialenosťou $d(x)$, žiaden neskorší spracovávaný vrchol už nebude mať menšiu vzdialenosť.

2. **Obmedzený rozsah vzdialeností nespracovaných vrcholov:** Nech $x$ je naposledy spracovaný vrchol. Každý vrchol, ktorý ešte nebol spracovaný, má aktuálnu vzdialenosť v rozsahu

   $$[d(x), d(x)+1, \ldots, d(x)+C]$$

   alebo je zatiaľ nedosiahnuteľný (vzdialenosť $\infty$).

   Prečo to platí? V momente, keď sme sa do vrcholu $x$ dostali, existovala cesta s dĺžkou najviac $d(x)$. Každá hrana má dĺžku najviac $C$, takže susedia tohto vrcholu môžu mať vzdialenosť najviac $d(x)+C$. Počas ďalších iterácií sa tieto vzdialenosti môžu len *zmenšovať* (keď nájdeme kratšiu cestu), zatiaľ čo hodnota $d(x)$ pre posledný spracovaný vrchol už iba rastie.

Z týchto pozorovaní vyplývajú dve kľúčové dôsledky:

1. **Monotónna halda:** Stačí nám dátová štruktúra, ktorá predpokladá, že do haldy nikdy nepridáme vrchol s menším kľúčom, než je aktuálne minimum. Tento typ haldy nazývame *monotónna halda*.

2. **Malá pamäť:** Keďže všetky vzdialenosti sa v danom momente nachádzajú v intervale $[d(x), \ldots, d(x)+C]$ (plus $\infty$ pre nespracované vrcholy), malo by nám stačiť $O(C)$ pamäte.

## 8.1 Myšlienka

Použijeme *radixovú haldu*: udržiavame niekoľko „chlievikov“ (bucketov) s *exponenciálne* rastúcimi rozsahmi kľúčov

$$1, 1, 2, 4, 8, 16, 32, \ldots$$

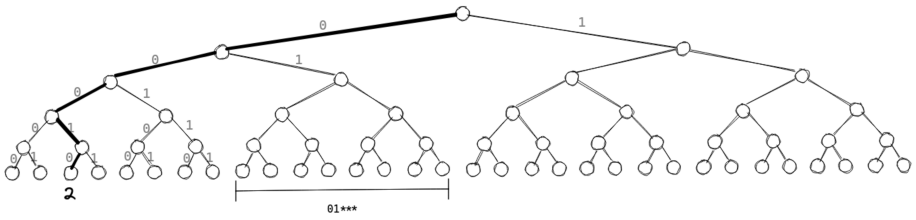
a zároveň si pamätáme poslednú vybranú hodnotu *last*. [1]

Začnime s hračkárskym príkladom, na ktorom si ukážeme, ako štruktúra funguje a jej hlavnú myšlienku, potom si povieme o implementácii.

Predstavme si, že všetky možné 5-bitové čísla (od 0 po 31, v binárnom zápise `00000` … `11111`) usporiadame ako binárny strom: každý bit určuje smer – ak

[1] Pôvodná verzia (Ahuja et al. 1990) zaraďovala prvky podľa rozdielu od *last* do fixných kategórií; tu opisujeme jednoduchší a veľmi praktický variant „radix heap“, pekne vysvetlený aj na `http://ssp.impulsetrain.com/radix-heap.html`.

je 0, ideme doľava, ak je 1, ideme doprava. Pozor! Tento imaginárny/implicitný strom je len pomôcka pri vysvetľovaní – strom sa v skutočnosti nikdy nevytvára ani neukladá v pamäti.




Každé 5-bitové číslo zodpovedá jednej ceste od koreňa k listu v tomto imaginárnom strome. Napríklad číslo 2 (`00010`) predstavuje hrubou čiarou vyznačenú cestu na obrázku vyššie.

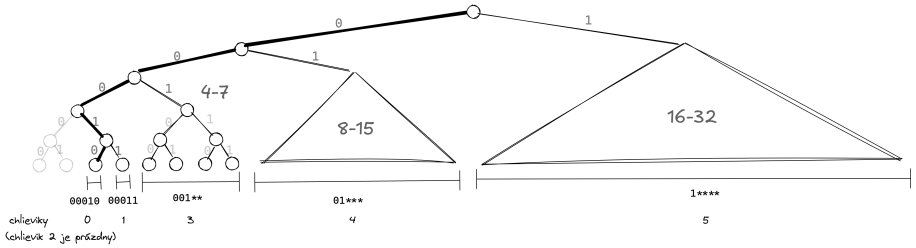
Kratší prefix zodpovedá podstromu – teda rozsahu čísel. Napríklad rozsah `01000` ... `01111` (alebo `01***`, kde `*` je zástupný znak pre `0` alebo `1`) je znázornený na obrázku.

Čísla rozdelíme do chlievikov podľa najvýznamnejšieho bitu (MSB, most significant bit), v ktorom sa hodnota $x$ a *last* líšia. V našom imaginárnom strome MSB zodpovedá prvému bodu, kde sa cesty od koreňa k listu rozchádzajú – *last* ide doľava a $x$ doprava.

Príklad: Nech $last = 2$. Potom

$$MSB(2, 13) = MSB(\texttt{00010}, \texttt{01101}) = 4,$$

takže 13 patrí do chlievika č. 4:




Nech $last = 2$. Všetky čísla v halde budú teda $\geq 2$ a rozdelíme ich do nasledujúcich chlievikov:

- #0: `00010`: samotné číslo 2
- #1: `00011`: číslo 3 (líši sa v 1. bite sprava)
- #2: —— (čísla líšiace sa v 2. bite sú menšie ako 2, preto je chlievik prázdny)

- #3: `001**`: čísla 4–7 (líšia sa v 3. bite)
- #4: `01***`: čísla 8–15 (líšia sa v 4. bite)
- #5: `1****`: čísla 16–31 (líšia sa v 5. bite)

Vo všeobecnosti: Pre prvok $x$ vezmime binárnu reprezentáciu čísel $x$ a *last*. Nech $i$ je najvýznamnejší bit (MSB), v ktorom sa $x$ a *last* líšia (budeme počítať od 1 sprava; definujeme $i = 0$, ak $x = last$). Prvok $x$ potom vložíme do $i$-teho chlievika.

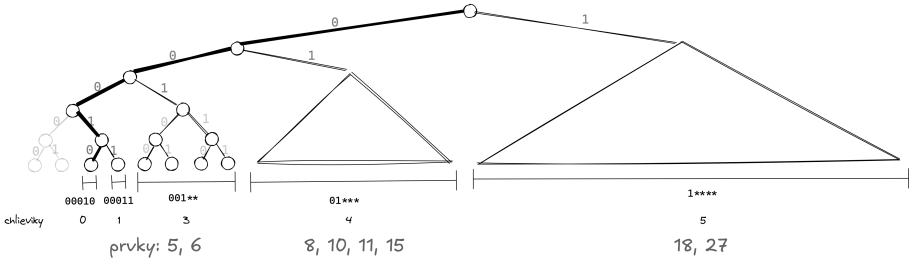
Inými slovami, stačí vypočítať $x \oplus last$ (bitový XOR) a nájsť pozíciu najvyššie nastaveného bitu. (XOR dvoch čísel má jednotky práve na pozíciách, kde sa bity líšia, a nuly tam, kde sú rovnaké.)

Ak chceme vložiť nový prvok, jednoducho ho vložíme do príslušného chlievika. Ak potrebujeme zmeniť hodnotu existujúceho prvku (poznáme jeho pozíciu v halde), najjednoduchšie je ho odstrániť a znova vložiť tam, kam podľa novej hodnoty patrí.

Ako funguje operácia `extract-min`? Ak nultý chlievik nie je prázdny, jednoducho z neho odstránime a vrátime najmenší prvok. Ak je prázdny, postupujeme zľava doprava, kým nenájdeme prvý neprázdny chlievik – čo zaberie najviac $O(\log C)$ krokov.

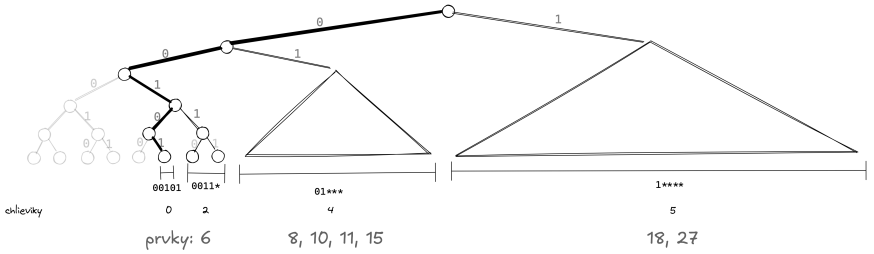
Keď takýto chlievik nájdeme, prejdeme všetky jeho prvky a určíme nové minimum (všimnime si, že čísla v rámci jedného chlievika nemusia byť zoradené!). Toto minimum odstránime a všetky ostatné prvky z daného chlievika presunieme do ich správnych chlievikov podľa novej hodnoty *last*.

Povedzme, že halda momentálne obsahuje nasledujúce čísla: 5, 6 (v chlieviku 3), 8, 10, 11, 15 (v chlieviku 4), 18 a 27 (v chlieviku 5), pričom $last = 2$ (hodnota *last* je vyznačená hrubou čiarou):

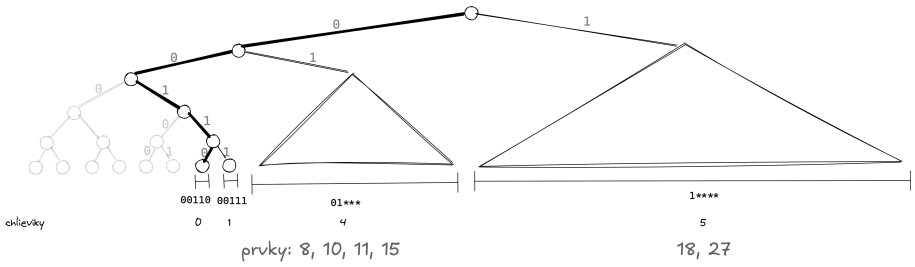



Ak vykonáme operáciu `extract-min`, prejdeme chlieviky 0, 1 a 2, a ako prvý neprázdny nájdeme chlievik č. 3. Z neho vyberieme minimum – hodnotu 5 – a vrátime ju ako výsledok. Zvyšný prvok 6 z tohto chlievika presunieme do nového chlievika na základe aktualizovanej hodnoty $last = 5$.

Po vykonaní operácie `extract-min` bude halda vyzerať takto:



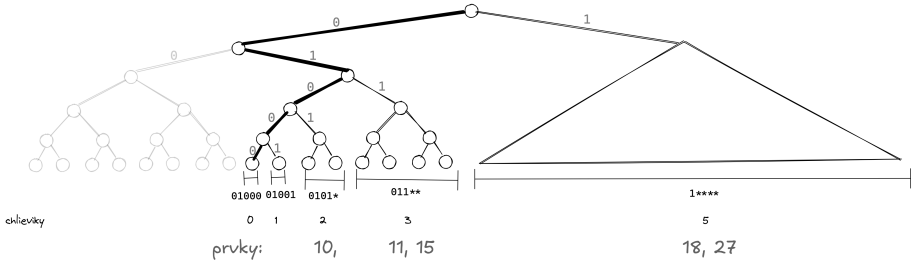

Ak opäť vykonáme `extract-min`, vrátime hodnotu 6 z druhého chlievika a halda bude vyzerať nasledovne:




Nakoniec, ak znovu vykonáme operáciu `extract-min`, prvý neprázdny chlievik bude číslo 4. Predtým sa hodnota *last* nachádzala v rozsahu `00***`, no v tomto ľavom podstrome už nie sú žiadne ďalšie prvky. Najbližší neprázdny je podstrom `01***`, teda chlievik 4.

Z tohto chlievika zistíme, že 8 je nové minimum, a všetky ostatné prvky (10, 11 a 15) prerozdelíme do chlievikov 0 až 3 na základe novej hodnoty $last = 8$ (viď obr. nižšie).

Všimnime si, že všetky prvky v chlieviku 4 začínajú `01***`, takže sa môžu líšiť len v bitoch 1 až 3. Zároveň pre neskoršie chlieviky (v našom hračkárskom príklade ide len o chlievik 5) sa nič nemení – stále sa líšia od novej hodnoty *last* vo vyšších bitoch, takže ich nie je potrebné presúvať.



Vo všeobecnosti, ak je prvý neprázdny chlievik číslo $k$, znamená to, že predchádzajúci *last* mal v $k$-tom bite hodnotu `0`. Ľavý podstrom je teda prázdny; prejdeme preto pravý podstrom, v $k$-tom chlieviku nájdeme minimum a ostatné prvky z tohto chlievika rozdelíme do chlievikov $0, \ldots, k-1$ podľa novej hodnoty *last*. Neskoršie chlieviky sa týmto krokom neovplyvnia.

Časové náklady:

- `insert` a `decrease-key` sú zjavne $O(1)$,
- `extract-min` trvá $O(\log C + B)$, kde $B$ je veľkosť práve spracovávaného chlievika. V najhoršom prípade môže byť $B = \Theta(n)$, avšak amortizovane ostáva čas $O(\log C)$.

Prečo? Pozrime sa na životný cyklus jedného prvku: najprv ho vložíme do nejakého chlievika a odtiaľ sa už môže posúvať len doľava. Operácia `decrease-key` ho posúva doľava a každé prerozdelenie chlievika počas `extract-min` ho tiež posunie len doľava. Keďže máme iba $O(\log C)$ chlievikov, každý prvok sa môže presunúť nanajvýš $O(\log C)$-krát.

Môžeme si teda predstaviť, že za každú operáciu `insert` účtujeme $\log C\$$, z ktorých sa všetky neskoršie presuny zaplatia. Zamyslite sa nad tým: najhorší prípad zložitosti `insert` je síce $O(1)$, ale ak mu priradíme amortizovanú cenu $O(\log C)$ (teda výrazne vyššiu), môžeme tým pokryť aj náklady na `extract-min`.

Je to pekný príklad toho, ako „preplatením“ jednej operácie vieme zaplatiť za inú, čo sa nám oplatí, ak chceme dosiahnuť lepší odhad celkovej zložitosti algoritmu. Namiesto pesimistického odhadu $n \times (T_{\texttt{insert}} + T_{\texttt{extract-min}}) = n \times (O(1) + O(n)) = O(n^2)$, dostaneme vďaka kreatívnemu účtovníctvu $n \times (O(\log C) + O(\log C)) = O(n \log C)$.

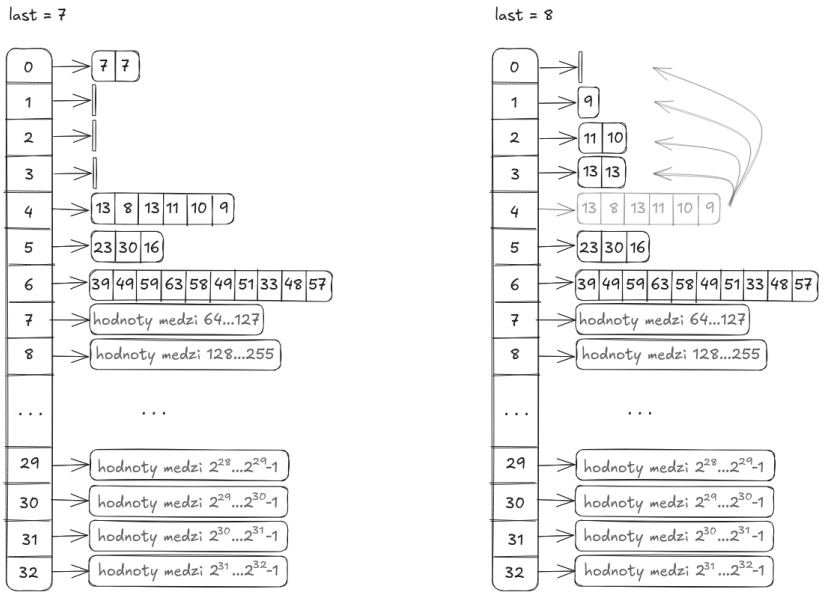
## 8.2 Implementácia

Každý chlievik bude obyčajný (neutriedený) vektor. (Nepoužívajte spájané zoznamy — sú neefektívne z hľadiska cache.) Celá halda bude reprezentovaná ako obyčajné pole, ktoré obsahuje 33 alebo 65 chlievikov – podľa toho, či pracujeme s 32-bitovými alebo 64-bitovými číslami.

Pri odstraňovaní prvku z vektora si dajte pozor: nechcete používať funkciu, ktorá odstráni prvok a následne presunie všetky prvky za ním – to by malo lineárnu zložitosť! (Ako to spraviť v $O(1)$?)

Na určenie správneho chlievika pre číslo $x$ nám stačí len zopár bitových operácií. Vypočítame XOR medzi *last* a $x$ a následne zistíme pozíciu najvyššieho nastaveného bitu (MSB – most significant bit). Napríklad: `01001110`$\oplus$`01011000` = `00010110`. XOR má `1` na pozíciách, kde sa bity dvoch čísel líšia, a `0` inde. Pozícia najvyššej `1` v tomto výsledku (v našom prípade piaty bit sprava) určuje číslo chlievika, do ktorého $x$ patrí.

Intelovské a AMD procesory majú na tento účel priamo hardvérovú inštrukciu




Obr. 8.1: Implementácia radixovej haldy pre 32-bitové čísla. Vpravo stav haldy po tom ako trikrát zavoláme `extract-min`. Dvakrát vyberieme 7 priamo z nultého chlievika, tretíkrát nájdeme prvý neprázdny chlievik (ten štvrtý), z ktorého vyberieme minimum 8 a všetky ostatné prvky prerozdelíme do skorších chlievikov.

– `BSR` (Bit Scan Reverse)[2], ktorá vráti index najvyššieho nastaveného bitu. (Túto inštrukciu mala už staroveké 386.) Na ARM procesoroch zase máme príbuznú inštrukciu `CLZ` (Count Leading Zeros), ktorá spočíta počet núl pred prvou `1`.

Môžete si dohľadať, ako spočítať pozíciu najvyššieho bitu vo vašom jazyku a kompilátore. Vo väčšine moderných kompilátorov sa nájde spôsob, či už cez štandardizované funkcie alebo cez „vstavané funkcie kompilátora“, tzv. „builtin“ a „intrinsic“ funkcie, ktoré sa prekladajú priamo na procesorové inštrukcie. Napríklad:

| | |
|---|---|
| C++20 | štandardizovaná funkcia `std::countl_zero`[3] |
| staršie C/C++: | podľa kompilátora |
| – GCC a Clang | `__builtin_clz`, resp. `__builtin_clzll`[4] |
| – MSVC | `#include <intrin.h>`, `_BitScanReverse`, `_BitScanReverse64`[5] |
| Java | `Integer/Long.numberOfLeadingZeros`[6] |
| Rust | `u32/u64::leading_zeros`[7] |

(pozor, niektoré operácie môžu byť *nedefinované* pre 0!)

Pamätajte, že pri Dijkstrovom algoritme potrebujete vedieť rýchlo nájsť vrchol `x` v halde (pri `decrease-key`), takže bude treba si udržiavať a aktualizovať tabuľku s pozíciami vrcholov v halde.

## Referencie

Ahuja, Ravindra K et al. (1990). „Faster algorithms for the shortest path problem“. In: *Journal of the ACM (JACM)* 37.2, s. 213–223.

[2] `https://c9x.me/x86/html/file_module_x86_id_20.html`

# Časť III

# Stromy

# Kapitola 9

# Lenivý (scapegoat) strom

Existuje mnoho druhov „snaživých“ vyvažovaných stromov, ktoré si pre každý vrchol pamätajú nejakú dodatočnú informáciu – napríklad výšku, čiernu/červenú farbu alebo počet uzlov v podstrome – a pomocou rotácií sa neustále snažia udržať strom vyvážený. Naopak, v tejto kapitole si však ukážeme úplne iný prístup: Scapegoat stromy sú úplne lenivé a nerobia vôbec nič, kým nemusia.

Je to podobné, ako keď máme doma neporiadok: Pokiaľ je ešte relatívne malý, vieme potrebné veci nájsť pomerne rýchlo. Ale keď nám už prerastie cez hlavu, musíme si upratať. Zároveň nás môže hriať dobrý pocit, ako sme odďaľovaním upratovania ušetrili kopu času.[1]

Možno si v tomto momente kladiete (uprávnenú) otázku:

> Prečo sa vôbec zaoberať ďalším typom vyhľadávacieho stromu? To nám nestačia existujúce riešenia ako AVL alebo červeno-čierne stromy?

Uvediem tri dôvody, prečo sú scapegoat stromy zaujímavé:

**#1. Lenivá metóda.** Pre mnohé druhy stromovitých štruktúr sú základnou metódou vyvažovania rotácie. V neskorších kapitolách si však ukážeme dátové štruktúry, kde nie je možné efektívne rotovať vrcholy. Otázka potom znie: Má niekto plán B? My ho mať budeme Elegantné riešenie: nič nerobiť, až kým strom nie je príliš nevyvážený a vtedy od základov prebudovať časť štruktúry.

**#2. Výška stromu.** Perfektne vyvážený binárny strom má výšku $\lceil \lg n \rceil$. Ako sú na tom iné vyvažované stromy?

- Červeno-čierne stromy garantujú maximálnu výšku $2 \lg n$.
- AVL stromy dosahujú maximálnu výšku okolo $1.44 \lg n$.
- Priemerná výška náhodného stromu (treapu) je okolo $2.988 \lg n$

[1] Disclaimer: Toto nie je odporúčanie do reálneho života. Chýbajú implementačné detaily.

Otázka znie: Ako veľmi sa vieme priblížiť k ideálnej výške $\lg n$?

Scapegoat stromy umožňujú dosiahnuť výšku $(1+\varepsilon)\lg n$ pre ľubovoľne malé $\varepsilon > 0$, pričom insert a delete bude stále trvať $O(\log n)$. Samozrejme, je to za určitú cenu: čím viac sa snažíme priblížiť k optimálnej výške, tým častejšie budeme musieť prebudovávať podstromy a konštanta skrytá v $O$ bude väčšia. Scapegoat stromy nám však poskytujú možnosť zobchodovať efektívnosť aktualizácií za rýchlosť vyhľadávania.

**#3. Dodatočná pamäť.** Väčšina vyvažovaných stromov si musí pre každý uzol pamätať nejakú dodatočnú informáciu:

- AVL stromy si udržiavajú rozdiely výšok ľavého a pravého podstromu,
- červeno-čierne stromy si pamätajú farbu každého vrcholu,
- treapy si ukladajú náhodné priority vrcholov.

Otázka znie: Dá sa vyvažovať aj bez toho?

Prekvapujúca odpoveď znie ÁNO, hoci my si kvôli jednoduchosti ukážeme verziu, ktorá využíva dodatočnú pamäť.

## 9.1 Ako fungujú scapegoat stromy

Základná myšlienka je veľmi jednoduchá: necháme strom rásť, nerobíme žiadne rotácie ani vyvažovanie, pokiaľ to nie je nevyhnutné. Keď sa strom stane príliš nevyváženým, nájdeme „vinníka“ – vrchol, ktorý je zodpovedný za nevyváženosť – a celý jeho podstrom prebudujeme na perfektne vyvážený binárny vyhľadávací strom.[2]

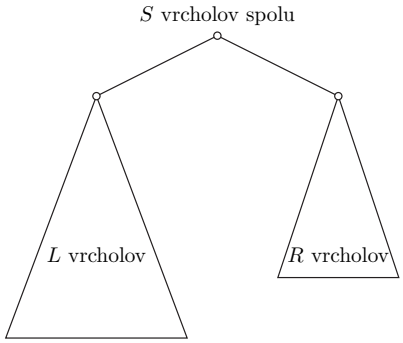
Prebudovanie podstromu s $k$ vrcholmi síce trvá lineárny čas $O(k)$, ale ukážeme si, že takéto prebudovania nastávajú len zriedka a že celkový čas potrebný na všetky prebudovania je v amortizovanom zmysle stále $O(\log n)$ na operáciu. Dôležité je, že ak je nevyvážená iba malá časť stromu, prebudujeme len túto lokálnu časť a zvyšok stromu zostáva nedotknutý.

### Kritérium vyváženosti

Vezmime si ľubovoľný vrchol $v$ v strome a jeho podstrom, ktorý obsahuje celkovo $S$ vrcholov (vrátane samotného $v$). Nech $L$ je počet vrcholov v ľavom podstrome

[2] Odtiaľ pochádza aj anglický názov !scapegoat tree! – *scapegoat* znamená „obetný baránok“, teda niekto, na koho zvalíme (možno i neprávom) všetku vinu. Scapegoat strom môže byť mierne nevyvážený na viacerých miestach, no my si vždy vyberieme práve jeden vrchol, na ktorý to zvalíme.

a $R$ počet vrcholov v pravom podstrome $v$ ako na obrázku:




Ak by bol vrchol $v$ perfektne vyvážený, ľavý aj pravý podstrom by mali mať zhruba rovnakú veľkosť (polovicu celého podstromu)

$$L \approx R \approx \frac{1}{2}S.$$

To je pomerne ťažko dosiahnuteľné, takže sa v scapegoat stromoch uspokojíme so slabším kritériom: Budeme hovoriť, že *vrchol $v$ je v rovnováhe,* ak platí

$$\begin{aligned} L &\leq \frac{2}{3}S \quad \text{a zároveň} \quad R \leq \frac{2}{3}S \\ \text{ekvivalentne: } R &\geq \frac{1}{3}S - 1 \quad \text{a zároveň} \quad L \geq \frac{1}{3}S - 1 \\ \text{ekvivalentne: } L &\leq 2 \times R + 2 \quad \text{a zároveň} \quad R \leq 2 \times L + 2 \end{aligned}$$

Budeme hovoriť, že *strom je vyvážený, ak je každý jeho vrchol $v$ rovnováhe.* Formálne: pre každý vrchol $v$ a každého jeho syna $s$ platí:

$$\text{size}(s) \leq 2/3 \times \text{size}(v),$$

teda žiadny syn nie je väčší ako dve tretiny celého podstromu svojho rodiča.

Z kritéria rovnováhy by malo byť zjavné, že ak je strom vyvážený, jeho výška bude logaritmická vzhľadom na počet vrcholov $N$. Prečo?

- Koreň má pod sebou všetkých $N$ vrcholov.
- Každý jeho syn má pod sebou najviac $\frac{2}{3}N$ vrcholov.
- Každá ďalšia úroveň znižuje počet vrcholov v podstrome o faktor aspoň 2/3, teda
- každý vnuk má najviac $\frac{2}{3} \times \frac{2}{3} \times N = (\frac{2}{3})^2 \times N$ vrcholov.
- každý pravnuk má najviac $\frac{2}{3} \times \frac{2}{3} \times \frac{2}{3} \times N = (\frac{2}{3})^3 \times N$ vrcholov.

- Všeobecne: podstrom v hĺbke $h$ obsahuje najviac $(\frac{2}{3})^h \times N$ vrcholov.

Otázka znie: aká je maximálna hĺbka stromu? Hľadáme najväčšie $h$, pre ktoré ešte existuje aspoň jeden vrchol v podstrome, teda

$$\begin{aligned} \left(\frac{2}{3}\right)^h \times N &\geq 1 \\ N &\geq \left(\frac{3}{2}\right)^h \\ \log_{3/2} N &\geq h \end{aligned}$$

Maximálna výška vyváženého stromu je teda: $h \leq \log_{3/2} N$, čo je približne $1.7095 \lg N$. Rozmyslite si, ako by sa maximálna výška zmenila, ak by sme namiesto dvoch tretín zvolili inú konštantu $\alpha \in (\frac{1}{2}, 1)$.

## Vkladanie

Algoritmus vkladania do Scapegoat stromu začína rovnako ako v obyčajnom binárnom vyhľadávacom strome: prechádzame od koreňa smerom nadol, nájdeme správne miesto podľa usporiadania kľúčov a vložíme nový vrchol ako list.

Rozmyslite si, že jediné veľkosti podstromov, ktoré sa pri vkladaní nového vrcholu zmenia, sú tie na ceste od koreňa k novo vloženému vrcholu. Preto po vložení prejdeme cestu späť do koreňa, prepočítame veľkosti podstromov a skontrolujeme, či všetky vrcholy na tejto ceste stále spĺňajú podmienku rovnováhy. (Vo väčšine prípadov budú a môžeme tu skončiť.)

Ak však nájdeme vrchol, ktorý je nevyvážený, prípadne viac takých vrcholov, vyberieme ten najvyšší a celý jeho podstrom prebudujeme na perfektne vyvážený binárny vyhľadávací strom.

Rozmyslite si, ako presne takúto rekonštrukciu efektívne vykonať. Cieľom je, aby prebudovanie podstromu s $k$ vrcholmi prebehlo v čase $O(k)$.

Celková zložitosť jednej operácie bude $O(\log n)$ a „raz za čas“ ešte plus $O(k)$ navyše, ak prebudujeme podstrom veľkosti $k$. Formálnu analýzu vykonáme čoskoro, ale už teraz si môžeme všimnúť intuitívny dôvod, prečo bude rekonštrukcia v amortizovanom zmysle takmer „zadarmo“: Po prebudovaní je konkrétny vrchol perfektne vyvážený, jeho podstromy majú zhruba $\frac{1}{2}k$ vrcholov. Aby sa takýto vrchol opäť stal nevyváženým, musíme pridať *lineárne veľa* vrcholov (napríklad $\approx \frac{1}{2}k$ vrcholov len na jednu stranu). Tým pádom však môžeme nákladné prebudovanie rozpočítať na lineárne veľa predchádzajúcich operácií. Ukážeme, že ak každá operácia vkladania uloží dopredu nejaké „peniaze do fondu opráv“, tak kým sa nejaký podstrom stane opäť nevyvážený, stihne si nasporiť dostatok zdrojov na zaplatenie celej rekonštrukcie. Tým bude zaručené, že amortizovaný čas každej operácie zostane $O(\log n)$.

### Vymazávanie

Vymazávanie v scapegoat stromoch je ešte lenivejšie ako vkladanie. Pri vymazávaní vrcholu $x$ ho fyzicky neodstránime zo stromu, len ho *označíme* ako vymazaný. Pozor: Operáciu find musíme potom upraviť tak, aby ignorovala vrcholy označené ako vymazané. Ak počet živých (nevymazaných) vrcholov klesne pod polovicu všetkých uzlov, prebudujeme celý strom.

Intuitívne: Prebudovanie celého stromu síce trvá $O(n)$, ale predtým muselo nastať aspoň $n/2$ operácií `delete`, takže amortizovaný čas jednej operácie `delete` zostáva $O(\log n)$.

## 9.2 Analýza časovej zložitosti

Pre každý vrchol $v$ si zadefinujeme $\Delta_v$ ako rozdiel veľkostí medzi ľavým a pravým podstromom, v absolútnej hodnote.

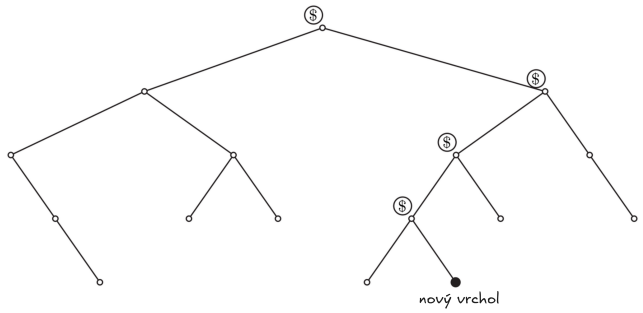
Budeme dodržiavať nasledovný invariant:

> Každý vrchol $v$ má vždy našetrených *aspoň* $\Delta_v - 1$ dolárov.

Keď vrchol pridáme ako list, nemusí mať našetrené nič (pretože $\Delta_v = 0$). Rovnako, po prebudovaní perfektne vyváženého podstromu nemusí mať žiadny vrchol v ňom našetrené nič, lebo rozdiel veľkostí podstromov je najviac 1. Avšak čím je vrchol viac nevyvážený, tým musí mať aj viac našetrené.

Ukážeme, že ak za každú operáciu `insert` dostaneme $2\log_{3/2} N\$$, vystačí nám to:

- $\log_{3/2} N$ dolárov minieme hneď:
  - na nájdenie správneho miesta,
  - pripojenie nového listu,
  - návrat späť ku koreňu,
  - prepočítavanie veľkostí podstromov a kontrolu rovnováhy na ceste.
- $\log_{3/2} N$ dolárov si odložíme na neskôr:
  - každý vrchol na ceste od nového listu po koreň dostane 1 dolár.



Pri vkladaní sa zmení $\Delta_v$ iba pre vrcholy na ceste k novému vrcholu – a to najviac o 1. Zároveň do každého takéhoto vrcholu uložíme práve 1 dolár, takže invariant ostáva zachovaný. Ak strom po vložení zostáva vyvážený, operácia týmto končí.

Ukážme teraz, že ak dôjde k prebudovaniu nevyváženého podstromu, dokážeme túto operáciu zaplatiť z už našetrených mincí.

Povedzme, že prebudovaný podstrom má veľkosť $k$ vrcholov. Keďže došlo k porušeniu rovnováhy, jeden z podstromov mal *viac ako* $\frac{2}{3}k$ vrcholov a druhý *menej ako* $\frac{1}{3}k - 1$ vrcholov. To znamená, že nevyvážený vrchol mal pred rekonštrukciou $\Delta_v \geq \frac{1}{3}k+1$ a teda podľa nášho invariantu musel mať našetrených aspoň $\frac{1}{3}k$ dolárov.

Po prebudovaní budeme mať perfektne vyvážený podstrom a vrcholy v ňom nemusia mať nič našetrené. takže môžeme všetky našetrené doláre použiť na zaplatenie práce. A keďže prebudovanie podstromu veľkosti $k$ trvá $O(k)$ času, máme dostatok financií na pokrytie celej rekonštrukcie.

A čo operácia delete?

Na jednu operáciu delete nám stačí $\log_{3/2} N + 1$ dolárov:

- $\log_{3/2} N$ dolárov zaplatíme ihneď za nájdenie a označenie vrcholu ako vymazaného,
- zvyšný jeden dolár si odložíme do prasiatka na neskôr.

Dodržíme tak invariant, že

> Strom, v ktorom je vymazaných $D$ vrcholov má našetrených práve $D$ dolárov.

Ak vymažeme polovicu vrcholov, strom má našetrených $N/2$ dolárov – a tie stačia na zaplatenie prebudovania celého stromu, ktoré trvá $O(N)$ času.

Pre korektnosť ešte musíme dodať, že $N$ tu označuje počet všetkých vrcholov – vrátane tých, ktoré sme označili ako vymazané, ale zatiaľ fyzicky neodstránili. V analýze dátových štruktúr však vždy posudzujeme zložitosť v závislosti od skutočného počtu prvkov v strome $n = N - D$. Avšak keďže vymazaných vrcholov môže byť najviac polovica, $N \leq 2n$ a teda výška stromu je najviac $\log_{3/2}(2n) < \log_{3/2} n + 2$, takže všetky naše odhady ostávajú logaritmické.

## Zhrnutie

| *Operácia* | *Čas v najhoršom prípade* | *Amortizovaný čas* |
|---|---|---|
| `find` | $O(\log n)$ | (scapegoat strom garantuje logaritmickú hĺbku vždy) |
| `insert` | $O(n)$ | $O(\log n)$ |
| `delete` | $O(n)$ | $O(\log n)$ |

# Úlohy

- Ako sa zmení maximálna výška stromu, ak namiesto konštanty $2/3$ zvolíme inú konštantu $\alpha \in (\frac{1}{2}, 1)$? Aké kritérium rovnováhy máme zvoliť, ak chceme strom s výškou $1.5 \lg n$ alebo $1.1 \lg n$?

- Ak si povieme, že pri prebudovaní podstromu veľkosti $k$ vykonáme $c \times k$ inštrukcií, potom predpokladáme, že 1$ zaplatí $3c$ inštrukcií. Ak namiesto konštanty $2/3$ zvolíme inú konštantu $\alpha \in (\frac{1}{2}, 1)$, koľkokrát viac (alebo menej) dolárov potrebujeme ušetriť pri každom `insert`e, aby sme vedeli prebudovanie podstromu stále zaplatiť?

- Rozmyslite si, ako algoritmus vkladania upraviť tak, aby sme si nemuseli v strome pamätať žiadnu informáciu navyše.

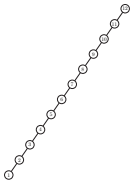
  Hint: Ak hĺbka nového uzla presiahne: $\log_{3/2} n$, kde $n$ je aktuálny počet vrcholov v strome, vieme, že strom je príliš vysoký a musíme zasiahnuť. Ako nájdeme vinníka, keď nemáme predpočítané veľkosti podstromov? Ak je vinníkov viac, budeme musieť vybrať toho najnižšieho – rozmyslite si, či to stačí.

# Kapitola 10

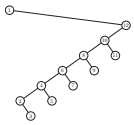
# Splay strom

Splay stromy sú podľa mňa jedna z najzázračnejších dátových štruktúr, aké kedy uvidíte.

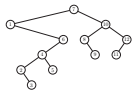
Klasické vyvažované stromy si pamätajú dodatočné informácie, nakoľko sú jednotlivé podstromy nevyvážené a vždy keď sa strom príliš odchýli od ideálneho tvaru, snažia sa ho naprávať, vyvažovať. Naproti tomu, splay strom pracuje úplne „naslepo“ – nemá úplne žiadnu dodatočnú informáciu. Nevie, ktoré časti stromu sú perfektne vybalansované a ktoré sú nakrivo. Napriek tomu dosahuje logaritmickú amortizovanú zložitosť.



(a) `insert` $1, \ldots, 12$

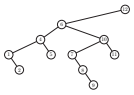


(b) `find` 1



(c) `find` 7

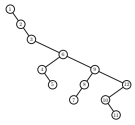
(d) `delete` 3



(e) `find` 13

(f) `find` 9

(g) `insert` 3



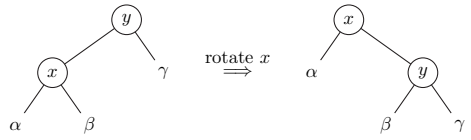
(h) `find` 1

## 10.1 Splayovanie

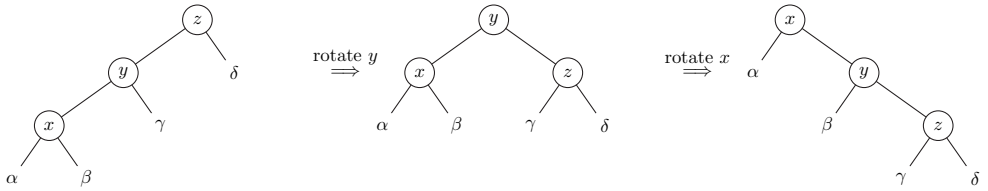
Kľúčovou operáciou splay stromu je – prekvapujúco – operácia splay($x$). Všetky ostatné operácie (find, insert, delete, atď.) budeme implementovať pomocou

splay.

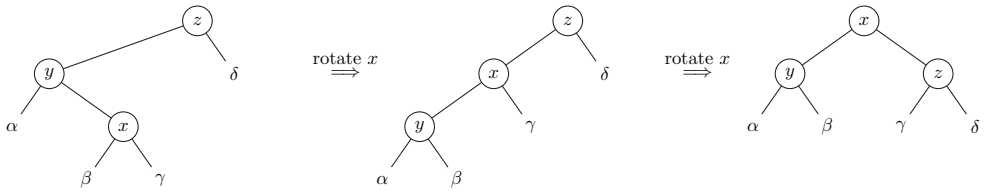
Splay($x$) sa začína klasickým vyhľadávaním: začneme v koreni, ak sme našli $x$, končíme, ak je hodnota vo vrchole príliš veľká, ideme vľavo, ak je príliš malá, ideme vpravo, kým sa dá. Na konci buď nájdeme vrchol $x$, alebo skončíme vo vrchole, ktorého hodnota je buď najbližšia menšia alebo najbližšia väčšia ako $x$. Tento vrchol následne „vybubleme“ až do koreňa stromu. Tento proces prebieha pomocou sérií rotácií, ale nie hocijakých: používame špeciálne pravidlá, ktoré zabezpečia dobrú amortizovanú časovú zložitosť.




(a) Prípad „cik“: Ak je otec $x$ koreň, jednoduchou rotáciou dostaneme $x$ do koreňa a končíme.




(b) Prípad „cik-cik“: Cesta $x$–$y$–$z$ je dvakrát vpravo ako na obrázku (alebo symetricky dvakrát vľavo). Rotujeme najskôr $y$, potom $x$. Výsledkom je, že cesta $x$–$y$–$z$ sa otočí.




(c) Prípad „cik-cak“: Cesta $x$–$y$–$z$ je cik-cakovito doľava–doprava ako na obrázku (alebo symetricky doprava–doľava). V tomto prípade dvakrát zrotujeme $x$; výsledkom je, že predkovia $y$ a $z$ skončia ako dve deti $x$.

Obr. 10.2: Pri splayovaní rozlišujeme tri rôzne situácie („cik“, „cik-cik“, „cik-cak“) a podľa situácie volíme, ktoré vrcholy rotujeme.

Nech $y$ je rodič vrcholu $x$ a $z$ jeho starý rodič (ak existuje). Rozlišujeme tri možné situácie:

1. Prípad „cik“: Ak vrchol $x$ nemá starého rodiča, teda jeho rodič $y$ je už koreň, vykonáme jedinú rotáciu, ktorá dostane $x$ do koreňa (pozri obr. 10.2a). Tento prípad nastáva najviac raz, na konci bublania, keďže $x$ sa dostane do koreňa, splayovanie sa končí.

2. Prípad „cik-cik“: Ak $x$ aj $y$ sú obaja ľavými synmi (alebo naopak obaja pravými), hovoríme o prípade „cik-cik“ (pozri obr. 10.2b). V takomto prípade spravíme rotáciu najprv na vrchole $y$, a až potom na $x$.

3. Prípad „cik-cak“: Ak $x$ je ľavý syn a $y$ pravý (alebo symetricky, $x$ je pravý a $y$ ľavý syn svojho otca), máme situáciu „cik-cak“ (pozri obr. 10.2c). Vtedy urobíme dve rotácie $x$. Najskôr sa $x$ posunie na miesto svojho rodiča, a následne ešte vyššie, na miesto starého rodiča. Výsledkom je, že $x$ sa posunie o dva úrovne vyššie a cesta $x$–$y$–$z$ sa otočí.

V prípade „cik-cik“ aj „cik-cak“ spravíme dve rotácie a vrchol $x$ sa dostane o 2 úrovne vyššie. Proces opakujeme, až kým sa $x$ nedostane do koreňa.

Na obrázkoch 10.3 a sú dve ukážky, ako prebieha splayovanie vrchola na konkrétnom strome.

## 10.2 Ostatné operácie

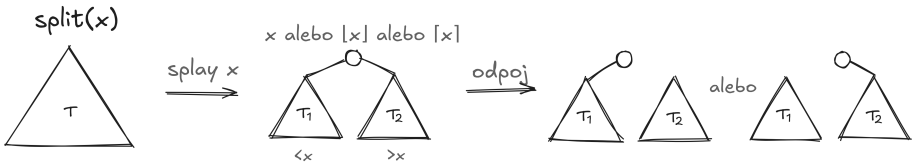
Všetky operácie v splay strome sa implementujú jednoducho, pomocou operácie splay:

- `find`($x$): Vysplayujeme $x$ a pozrieme sa na koreň – ak je tam $x$, našli sme ho; v opačnom prípade sa v strome nenachádza.

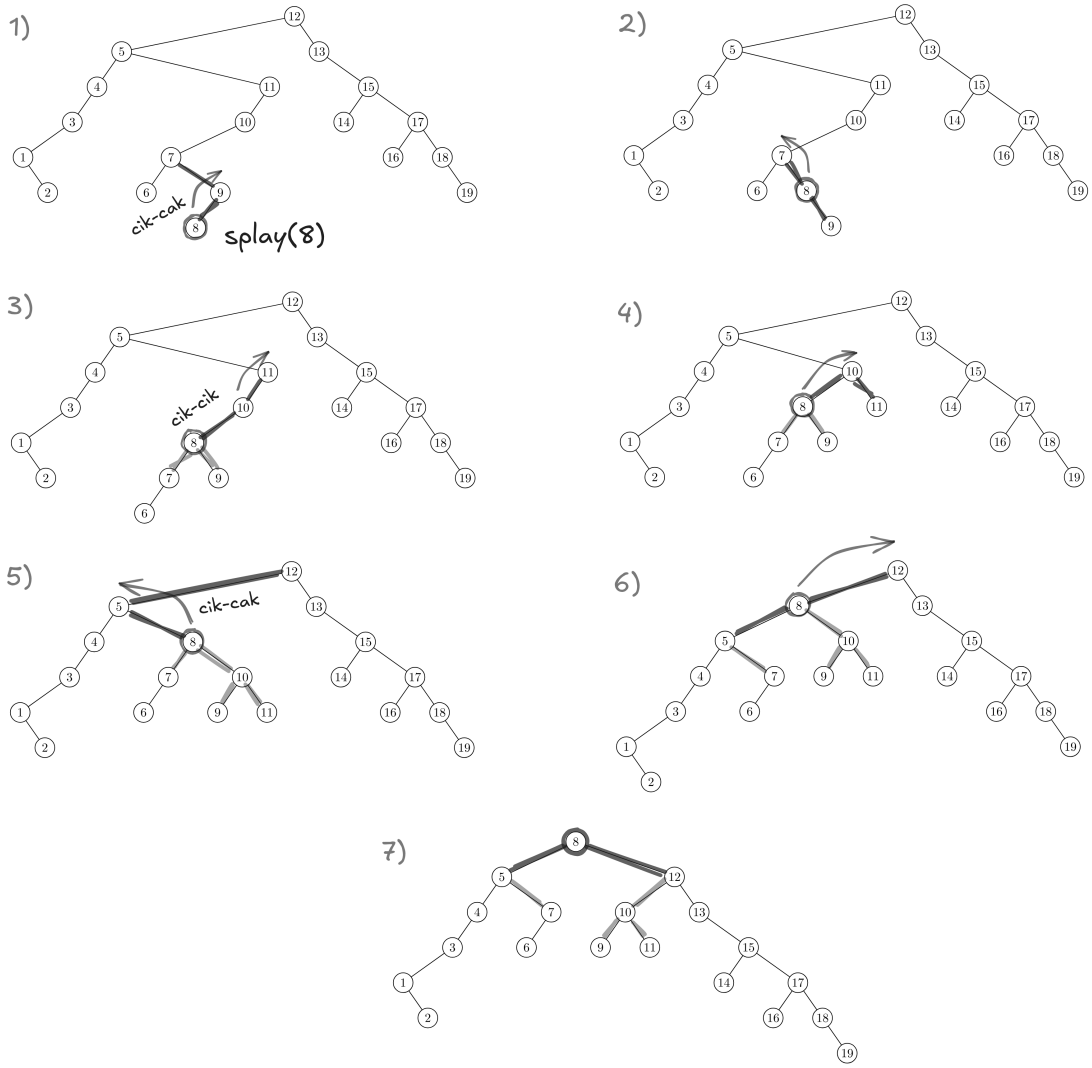
- `min`: Vysplayujeme $-\infty$. Do koreňa sa dostane najbližší väčší prvok, čo je minimum celého stromu.

- `max`: Vysplayujeme $+\infty$. Do koreňa sa dostane najbližší menší prvok, čo je maximum celého stromu.

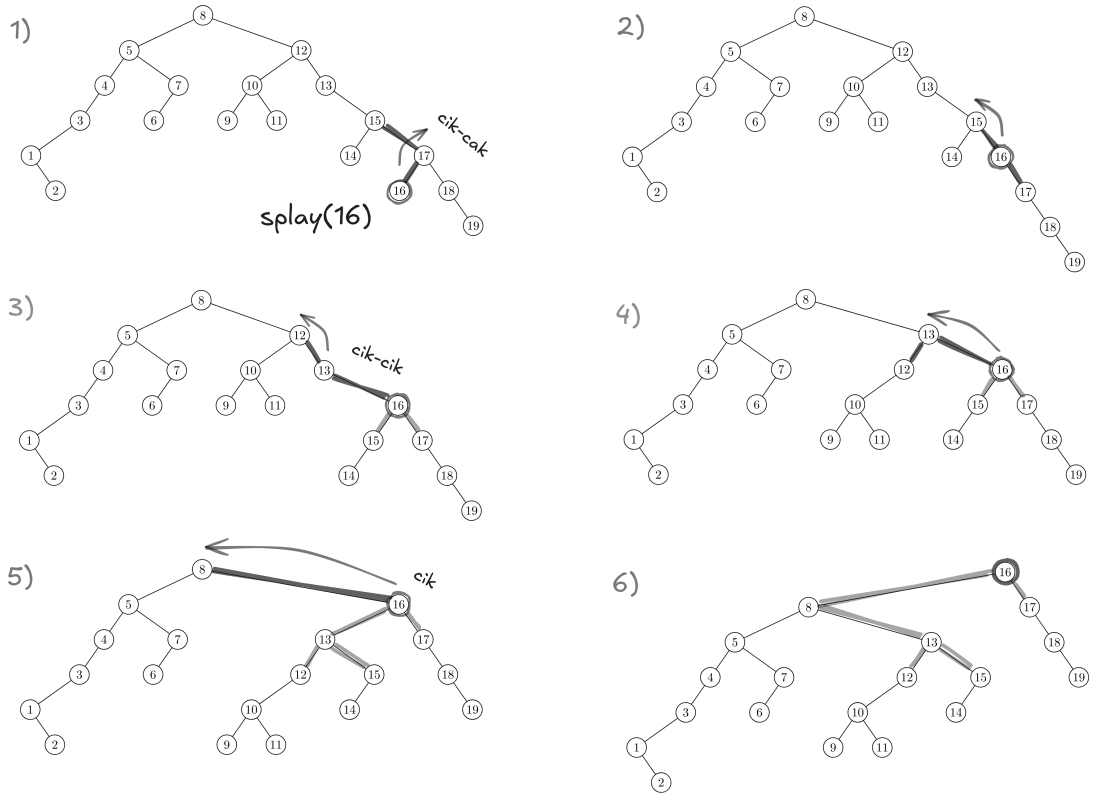
- `split`($x$): Chceme rozdeliť strom na dve časti – jeden strom s prvkami $\leq x$ a jeden strom s prvkami $> x$. Stačí vysplayovať $x$, čím sa do koreňa dostane samotné $x$, alebo najbližšia menšia alebo najbližšia väčšia hodnota. Celý ľavý podstrom koreňa bude $< x$, celý pravý podstrom koreňa bude $> x$ a samotný koreň porovnáme s $x$ a pripojíme na správnu stranu. Stačí zmazať jedinú hranu: od koreňa k ľavému alebo pravému synovi.



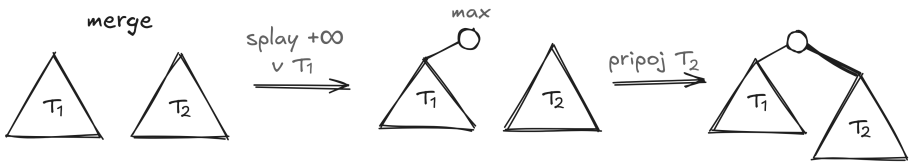

- `merge`($T_1, T_2$): Máme dané dva stromy spolu s garanciou, že všetky prvky v $T_1$ sú menšie ako všetky prvky v $T_2$ a chceme ich spojiť do jedného stromu. Riešenie je elegantné: v $T_1$ vysplayujeme $+\infty$. Tým sa do koreňa dostanem maximum a koreň *nebude* mať pravého syna. Tým pádom $T_2$ môžeme jednoducho pripojiť ako pravého syna koreňa.



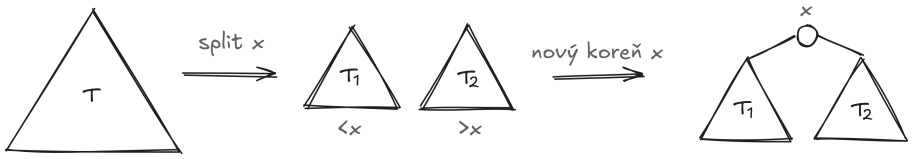

Obr. 10.3: Splayovanie vrcholu 8 prebehne tromi dvoj-rotáciami: 1) Cesta 8—9—7 je „cik-cak“, takže dvakrát zrotujeme 8. 3) Cesta 8—10—11 je „cik-cik“, takže najskôr zrotujeme 10, potom 8. 5) 8—5—12 je opäť „cik-cak“, takže dvakrát zrotujeme 8. 7) Na konci je vrchol 8 v koreni. Všimnite si na prvom obrázku cestu z 8 do koreňa: 8—9—7—10—11—5—12. Táto cesta sa postupne rotáciami transformovala na podstrom vyznačený šedou na poslednom obrázku.



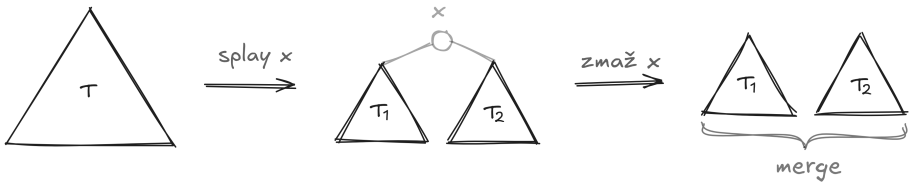

Obr. 10.4: Splayovanie vrcholu 16: Cesta do koreňa 16—17—15—13—12—8 sa skladá z 16—17—15, čo je „cik-cak“, 15—13—12, čo je „cik-cik“ a posledného kroku 12—8 „cik“. Takto 16 prebubleme až do koreňa.




- `insert`$(x)$: Stačí spraviť `splay`$(x)$. Ak je $x$ v koreni, strom už hodnotu obsahuje a môžeme skončiť. V opačnom prípade rozdelíme strom ako pri operácii `split` na strom $T_1$ s prvkami $< x$ a $T_2$ s prvkami $> x$. Ako výsledok vrátime strom, kde $x$ je nový koreň a jeho ľavý a pravý syn sú $T_1$ a $T_2$.




- `delete`$(x)$: Stačí vysplayovať $x$, čím sa $x$ dostane do koreňa, ktorý vymažeme. Ostanú nám jeho dva podstromy, $T_1$ a $T_2$, na ktoré zavoláme `merge`.




## 10.3 Jednoduchá analýza

V tejto časti si dokážeme, že splay stromy majú logaritmickú amortizovanú zložitosť. Budeme sa sústrediť iba na analýzu samotnej operácie splay, keďže všetky ostatné operácie vieme implementovať pomocou zopár volaní splay a zvyšná práca je konštantná.

Pre zjednodušenie predpokladajme, že v strome sa nachádzajú kľúče $1, 2, \ldots, n$ a že vždy splayujeme kľúč, ktorý v strome naozaj je.

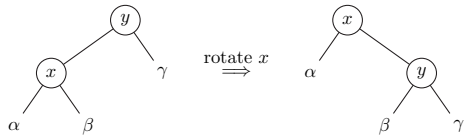
Pre každý vrchol $x$ si zavedme nasledovné označenia:
$size(x) = s_x$ je počet vrcholov v podstrome zakorenenom vo vrchole $x$,
$rank(x) = r_x = \lfloor \lg s_x \rfloor$ je hodnota odvodená od veľkosti podstromu.
Budeme udržiavať nasledovný invariant:

> Každý vrchol má na svojom „účte“ nasporených práve $r_x$\$.

Spomeňme najskôr niekoľko postrehov o rankoch.

- Pri rotácii vrcholu $x$ s jeho otcom $y$ sa menia iba ich ranky – $r_x$ a $r_y$. Všetky ostatné ranky zostávajú nezmenené, pretože žiadne iné podstromy okrem $x$ a $y$ sa nezmenia – iba $x$ sa stane otcom $y$ a $y$ dostane jedného zo synov $x$.

- Zároveň platí, že *po* rotácii má vrchol $x$ rovnaký rank, ako mal jeho otec *pred* rotáciou:




  Vrchol $y$ vľavo a vrchol $x$ vpravo majú pod sebou tie isté vrcholy: $x$, $y$ a podstromy $\alpha, \beta, \gamma$.

- Listy majú veľkosť podstromu $s_x = 1$, a teda rank $r_x = 0$.

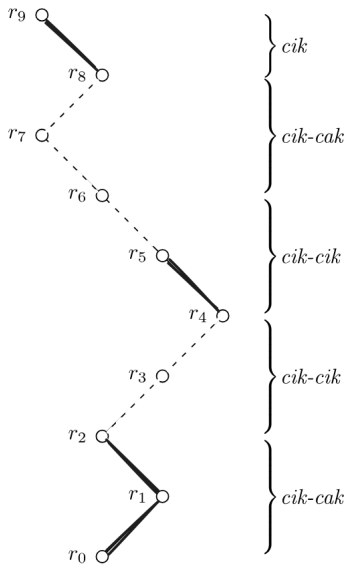
- Rank otca je vždy aspoň taký veľký, alebo väčší ako rank jeho syna (podstrom obsahuje všetky synove vrcholy a ešte nejaké navyše). To znamená, že na ľubovoľnej ceste od nejakého vrcholu ku koreňu tvoria ranky neklesajúcu postupnosť.

- Najväčší rank dosahuje koreň, kde $r_{\text{koreň}} = \lfloor \lg s_{\text{koreň}} \rfloor = \lfloor \lg n \rfloor$.

- Vrchol s rankom $r$ má pod sebou aspoň $2^r$ vrcholov, ale menej ako $2^{r+1}$ vrcholov.

Ešte predtým ako sa pustíme do formálneho dôkazu, skúsim ponúknuť dva intuitívne dôvody, prečo by mal byť čas splayovania logaritmicky.

**Intuícia I.** Predstavme si cestu od vrcholu $x$ až ku koreňu a označme ranky na ceste $r_0, r_1, \ldots, r_k$:

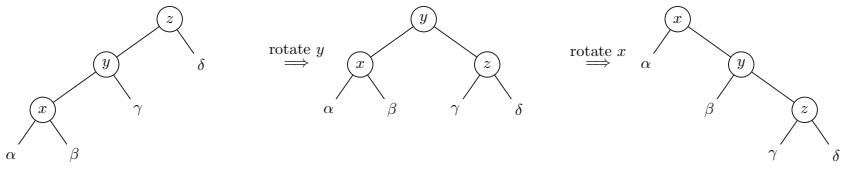



Hrany, kde $r_i < r_{i+1}$ sme označili nahrubo a hrany, kde $r_i = r_{i+1}$ sme vyznačili prerušovanou čiarou.

Zložitosť splayovania je úmerná dĺžke tejto cesty – $O(k)$, čo môže byť v najhoršom prípade až $O(n)$. Avšak ranky $r_i$ sú celé čísla z rozsahu $0, 1, 2, \ldots, \lfloor \lg n \rfloor$, takže pozdĺž tejto cesty sa môže rank zvýšiť nanajvýš $\lg n$-krát. Inými slovami, hrubých hrán je len logaritmicky veľa a treba zaplatiť hlavne za tie prerušované, kde rank nestúpa. My si však ukážeme, že kroky, kde prechádzame cez prerušované hrany zaplatia samy za seba z našetrených peňazí. Z pohľadu amortizovanej analýzy sú teda tieto kroky v podstate zadarmo.

**Intuícia II.** Čím je strom horšie vyvážený, tým viac má našetrené, naopak, čím lepšie je vybalansovaný, tým menej potrebuje mať našetrené. Našetrené doláre sú vlastne taký fond opráv; keď sa strom stáva horšie vyvážený, treba prispieť viac, naopak, ak ho trochu napravíme, uvoľnia sa nám nejaké zdroje, ktorými môžeme prácu zaplatiť.

Vezmime napríklad prípad „cik-cik“ a predstame si, že rank $x$ je $r$ a rank $z$ je o 2 väčší.



Čo z toho vyplýva? Veľkosť podstromu $x$ je medzi $[2^r, 2^{r+1})$ a veľkosť podstromu $z$ je aspoň $2^{r+2}$ – to znamená, že v podstromoch $\gamma$ a $\delta$ je veľa vrcholov (aspoň $2^{r+1}$), ktoré sme pri hľadaní vylúčili a vďaka tomu bolo hľadanie rýchle. Po rotáciach sa bude táto časť stromu možno horšie vyvážená, ale ukážeme si, že stačí za to do fondu opráv prispieť len $O(1)$ dolárov a takýchto krokov je najviac len logaritmicky veľa.

Naopak, predstavme si, rank $z$ je rovnaký ako rank $x$. To znamená, že podstromy $\delta$ a $\gamma$ sú v porovnaní s $\alpha, \beta$ malé – väčšina vrcholov je sústredená práve pod $x$. V tomto prípade sa však vyváženosť stromu *zlepší*: po rotáciách sa celé podstromy $\alpha, \beta$ posunú vyššie, a väčšina prvkov pod $x$ tak bude mať menšiu hĺbku než predtým. Tým pádom budeme môcť operácie zaplatiť z fondu opráv.

**Veta 10.1 (O prístupe).** *Na operáciu* splay$(x)$ *postačí* $3(r_{\text{koreň}} - r_x) + 1\$$, *pričom invariant ostáva zachovaný: každý vrchol $x$ má na účte nasporených $r_x\$$. Zjavne $r_{\text{koreň}} = \lfloor \lg n \rfloor$ a $r_x \geq 0$, takže na operáciu* splay$(x)$ *stačí* $3 \lg n + 1\$$.

**Lema 10.1.** *Označme $r'_x$ rank vrcholu $x$ po jednom kroku splayovania (teda po jednej dvojitej rotácii alebo po záverečnej jednoduchej rotácii).*

*Potom na každý prípad „cik-cik“/„cik-cak“ stačí $3(r'_x - r_x)\$$, a na posledný prípad „cik“ $3(r'_x - r_x) + 1\$$.*

Z lemy priamo plynie veta o prístupe: Označme $r_x, r'_x, r''_x, \ldots, r_x^{(k)}$ rank $x$ na začiatku, po prvom kroku, po druhom kroku, …, po $k$ krokoch. Keď sčítame náklady všetkých krokov v rámci splay$(x)$ podľa Lemy 1, dostaneme teleskopickú sumu

$$\begin{aligned} &3(r'_x - r_x) + 3(r''_x - r'_x) + \cdots + 3(r_x^{(k-1)} - r_x^{(k-2)}) + 3(r_x^{(k)} - r_x^{(k-1)}) + 1 \\ &\quad = 3\left(r_x^{(k)} - \cancel{r_x^{(k-1)}} + \cancel{r_x^{(k-1)}} - \cancel{r_x^{(k-2)}} + \cdots + \cancel{r''_x} - \cancel{r'_x} + \cancel{r'_x} - r_x\right) + 1 \\ &\quad = 3(r_x^{\text{posledný}} - r_x) + 1\$, \end{aligned}$$

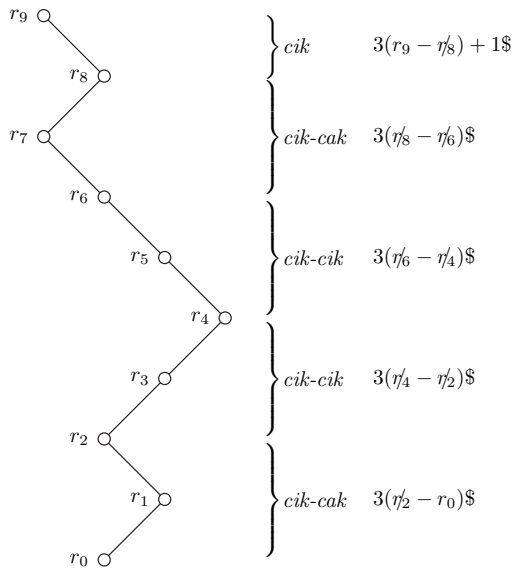
pričom $r_x^{\text{posledný}} = r_{\text{koreň}}$ je rank koreňa na konci splayovania.

Alternatívny pohľad je, keď si uvedomíme, že rank vrcholu $x$ *po* rotácii sa rovná ranku jeho rodiča *pred* rotáciou, resp. rank $x$ *po* každej dvojrotácii sa rovná ranku jeho starého rodiča *pred* týmito rotáciami. To znamená, že ak si ranky na ceste od vrcholu $x$ po koreň označíme $r_0, r_1, r_2, \ldots, r_{\text{koreň}}$, tak podľa Lemy 1 na operáciu splay$(x)$ stačí

$$3(\cancel{r_2} - r_0) + 3(\cancel{r_4} - \cancel{r_2}) + 3(\cancel{r_6} - \cancel{r_4}) + \cdots + 3(r_{\text{koreň}} - \cancel{r_\ell}) + 1 \leq 3(r_{\text{koreň}} - r_0) + 1.$$

dolárov (pozri obr. 10.5).

■ **Dôkaz lemy.** Budeme analyzovať jednotlivé prípady. Nech $x$ je vrchol, ktorý splayujeme, a $y$, $z$ sú jeho otec a starý otec. Označme $r_x, r_y, r_z$ ranky týchto vrcholov *pred* rotáciami a $r'_x, r'_y, r'_z$ ranky *po* príslušnej (dvojitej alebo záverečnej jednoduchej) rotácii. Ranky všetkých ostatných vrcholov sa pri danom kroku nemenia. S týmto zápisom teraz rozoberieme prípady „cik“ a „cik-cik“ (dôkaz pre posledný prípad „cik-cak“ je podobný a prenechávame ho ako cvičenie čitateľom).




Obr. 10.5: Dôkaz Vety o prístupe priamo vyplýva z Lemy 1. Ak označíme ranky vrcholov od splayovaného vrcholu ku koreňu $r_0, r_1, r_2, \ldots$, tak podľa lemy stačí, ak na každú dvojrotáciu dostaneme $3(r_{k+2} - r_k)\$$, teda $3\times$ rozdiel rankov vrcholu a jeho starého otca, a na poslednú „cik“ rotáciu stačí $3(r_{k+1} - r_k) + 1\$$. Tieto rozdiely sa pozdĺž cesty ku koreňu nasčítajú tak, že všetky prostredné členy vypadnú a ostane rozdiel medzi rankom koreňa a rankom splayovaného vrcholu $+1$.

**Prípad „cik“:** Pred rotáciou má dvojica $(x, y)$ uložených spolu $r_x + r_y$ dolárov (ostatné vrcholy ignorujeme, ich ranky sa nemenia). Po rotácii chceme zachovať invariant, takže spolu musia mať $r'_x + r'_y$ dolárov. Okrem toho potrebujeme aspoň 1\$ na zaplatenie samotnej rotácie. Požadovaný prídel je teda

$$(r'_x + r'_y) - (r_x + r_y) + 1.$$

Platí, že po rotácii sa $x$ posunie na pozíciu otca, takže $r'_x = r_y$, a zároveň $r'_x \leq r'_y$ (rank syna nie je väčší než rank otca). Preto

$$(r'_x + r'_y) - (r_x + r_y) + 1 = r'_y - r_x + 1 \leq (r'_x - r_x) + 1.$$

Teda na prípad *cik* stačí $(r'_x - r_x) + 1$ dolárov.

**Prípad „cik-cik“:** Rozlíšime dve situácie podľa vzťahu $r_x$ a $r_z$.

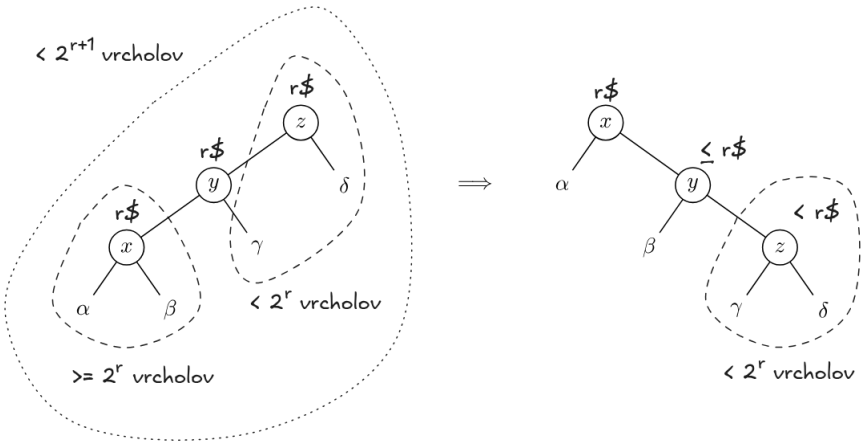
**a)** $\boldsymbol{r_x = r_z}$**.** V tomto prípade $x$, $y$, aj $z$ majú všetky rovnaký rank $r$ a keďže na konci sa $x$ dostane na miesto $z$ pred rotáciami, $r'_x = r_z$. Takže platí $r_x = r_y = r_z = r'_x = r$ a na tento krok máme pridelených $3(r'_x - r_x) = 0$ dolárov. To znamená, že amortizovane by mal byť tento krok „zadarmo“

– nedostaneme naň žiadne financie a všetko treba uhradiť z našetrených peňazí.

Spomeňme si, že $rank(x) = \lfloor \lg size(x) \rfloor$, takže veľkosť podstromu $x$ je niekde v rozsahu $[2^r, 2^{r+1})$. Vrcholy $y$ a $z$ majú rovnaký rank, takže aj tieto podstromy majú veľkosť v rozsahu $[2^r, 2^{r+1})$. Špeciálne podstrom $z$ má *menej* ako $2^{r+1}$ vrcholov, zatiaľčo podstrom $x$ má aspoň $2^r$ vrcholov. Z toho ale vyplýva, že veľkosť $|\gamma| + |\delta| + 1 < 2^r$ vrcholov a teda rank $z$ *po* dvoch rotáciach bude *menej* ako $r$!

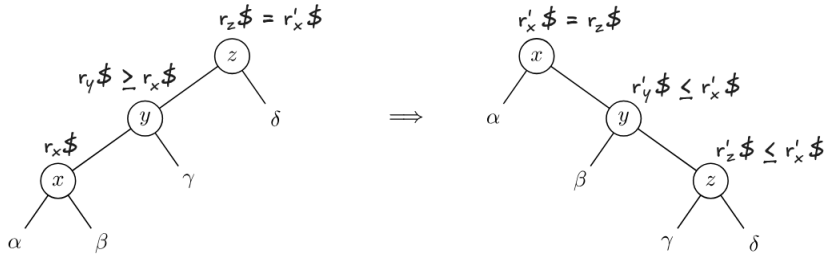
O ranku $y$ na konci nevieme povedať veľa, ale určite nie je väčší ako $r$ (platí $r'_z \leq r'_y \leq r'_x$).

Takže pred dvoma rotáciami mali vrcholy $x, y, z$ spolu našetrených $3r\$$ a po rotáciách im stačí *menej*, pretože minimálne vrcholu $z$ sa znížil rank. Všetku prácu v tomto kroku môžeme teda zaplatiť z úspor vrcholu $z$ a invariant zostane zachovaný.




**b)** $\boldsymbol{r_x < r_z}$**.** Pred rotáciami má trojica spolu $r_x + r_y + r_z$ dolárov, po rotáciách potrebuje $r'_x + r'_y + r'_z$, takže na zaplatenie tohto kroku potrebujeme 1\$ na prácu, plus rozdiel rankov, aby sme dorovnali úspory podľa invariantu:

$$\text{treba} \quad (r'_x + r'_y + r'_z) - (r_x + r_y + r_z) + 1\$.$$



Platí $r'_x = r_z$ (rank koreňa podstromu ostáva rovnaký; $z$ je koreň pred rotáciami, $x$ po nich). Ďalej $r_x \leq r_y \leq r_z$ a $r'_x \geq r'_y \geq r'_z$. Odtiaľ jednoduchými úpravami dostaneme

$$\begin{aligned}(\cancel{r'_x} + r'_y + r'_z) - (r_x + r_y + \cancel{r_z}) + 1 &= (r'_y - r_x) + (r'_z - r_y) + 1 \\ &\leq (r'_x - r_x) + (r'_x - r_x) + 1 \\ &= 2(r'_x - r_x) + 1.\end{aligned}$$

To znamená, že na zaplatenie tohto kroku potrebujeme $2(r'_x - r_x) + 1$ dolárov. Avšak v tomto prípade predpokladáme $r_x < r_z = r'_x$. Rozdiel $r'_x - r_x$ je teda aspoň 1 dolár a teda $2(r'_x - r_x) + 1 \leq 3(r'_x - r_x)$, čo sme chceli dokázať.

Ak to zhrnieme: Na tento krok je vyčlenených $3(r'_x - r_x)$ dolárov: z toho $(r'_x - r_x) \geq 1\$$ zaplatí samotné operácie a zvyšných $2(r'_x - r_x)\$$ použijeme na dorovnanie úspor tak, aby invariant ostal zachovaný.

**Prípad „cik-cak“:** Dôkaz je veľmi podobný prípadu „cik-cik“, takže ho prenechávame ako cvičenie čitateľom. Opäť treba rozlíšiť dve situácie: a) ranky $x$, $y$, aj $z$, sú všetky rovnaké – toto sú tie kroky „navyše“, kedy sa po ceste ku koreňu nezvyšuje rank avšak analýzou rankov pred a po rotáciach zistíme, že v tomto prípade je na konci strom lepšie vyvážený a má menší súčet rankov ako pred rotáciami a teda túto časť cesty môžeme zaplatiť z našetrených peňazí. Prípad b) je, že $r_x < r_z$ – rank na ceste ku koreňu stúpne. Takýchto krokov môže byť len logaritmicky veľa. Analýzou rankov pred a po rotáciach, s využitím vhodných rovností a nerovností medzi rankami sa dá dokázať, že z pridelených $3(r'_x - r_x)$ použiť $1\$$ na zaplatenie odvedenej práce a $2(r'_x - r_x)$ nám stačí na zachovanie invariantu – potrebujeme vlastne dokázať, že dvomi rotáciami sa síce môže strom stať menej vyvážený, ale nie príliš a do „fondu opráv“ stačí prispieť $2(r'_x - r_x)$ dolárov. □

## 10.4 Všeobecná analýza

Priraďme každému vrcholu $x$ kladnú váhu $w_x \in \mathbb{R}^+$. Nech $s_x$ označuje *váhu* podstromu $x$, t.j. súčet váh všetkých vrcholov v tomto podstrome. Rank vrcholu definujeme tentokrát ako

$$\text{rank}(x) = r_x = \lg s_x,$$

teda ako logaritmus váhy podstromu bez dolnej celej časti – rank teraz môže byť ľubovoľné, aj záporné, reálne číslo.

Rovnako ako vyššie budeme udržiavať nasledujúci invariant: každý vrchol bude mať na účte našetrených $r_x\$$. Inými slovami, celkový potenciál stromu bude

$$\Phi = \sum_x r_x = \sum_x \lg s_x.$$

**Veta 10.2 (Zovšeobecnená veta o prístupe).** *Operácia* splay$(x)$ *má amortizovanú zložitosť* $3(r_{koreň} - r_x) + 1$ *pre ľubovoľnú voľbu váh* $w_x \in \mathbb{R}^+$. *Inými slovami, ak* $W = \sum_x w_x$ *je celková váha stromu, potom zložitosť* splay$(x)$ *je* $3(\lg W - \lg w_x) + 1 = O(1 + \lg(W/w_x))$

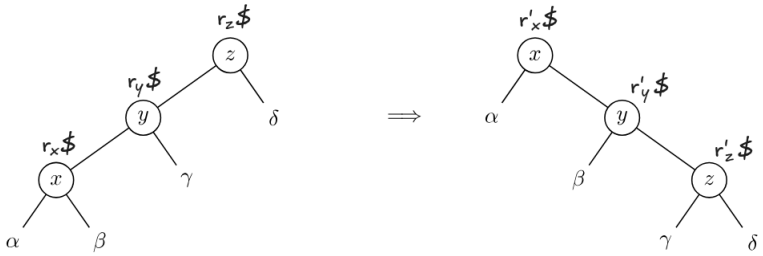
**Lema 10.2.** *Každý cik-cik/cik-cak prípad má amortizovanú zložitosť* $3(r'_x - r_x)$, *posledný prípad cik má zložitosť* $3(r'_x - r_x)+1$ *(pre ľubovoľnú voľbu váh* $w_x \in \mathbb{R}^+$*).*

■ **Dôkaz lemy.** Amortizovaný čas je skutočný čas plus zmena potenciálu:

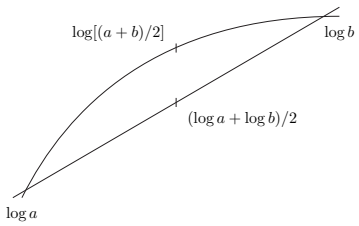
$$T_{\text{amort}} = T_{\text{skutočný}} + \Delta\Phi.$$

**Prípad „cik-cik“:** Skutočný čas sú 2 rotácie, rozdiel potenciálov $\Delta\Phi$ je

$$\begin{aligned}(r'_x + r'_y + r'_z) - (r_x + r_y + r_z) &= r'_y + r'_z - r_x - r_y \\ &\leq r'_x + r'_z - 2r_x\end{aligned}$$




Využijeme konkávnosť logaritmickej funkcie:




priemer logaritmov je menší alebo rovný logaritmu priemeru, takže sŭčet logaritmov (2× priemer) je menší alebo rovný 2× logaritmu priemeru:

$$r_x + r'_z = \lg s_x + \lg s'_z \leq 2\lg[(s_x + s'_z)/2].$$

Teraz si všimnime, že $s_x + s'_z \leq s'_x$ ($s_x$ je počet vrcholov $|\alpha| + |\beta| + 1$, $s'_z$ je počet vrcholov $|\gamma| + |\delta| + 1$, zatiaľčo $x$ je po rotáciách koreňom celého podstromu, takže obsahuje všetky tieto podstromy + vrcholy $x$, $y$ a $z$). Takže

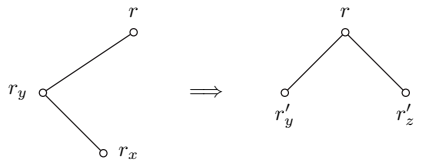
$$r_x + r'_z \leq 2\lg[(s_x + s'_z)/2] \leq 2\lg(s'_x/2) = 2r'_x - 2.$$

Dostávame nerovnosť $r'_z \leq 2r'_x - r_x - 2$, ktorú dosadíme vyššie:

$$\Delta\Phi \leq r'_x + (2r'_x - r_x - 2) - 2r_x = 3(r'_x - r_x) - 2,$$
$$T_{\text{amort}} = 2 + \Delta\Phi \leq 3(r'_x - r_x).$$

**Prípad „cik-cak“:** Dôkaz je podobný. Amortizovaný čas je

$$T_{\text{amort}} = T_{\text{skutočný}} + \Delta\Phi = 2 + r'_y + r'_z - r_x - r_y \leq 2 + r'_y + r'_z - 2r_x.$$




Z konkávnosti logaritmu a z nerovnosti medzi veľkosťami podstromov

$$r'_y + r'_z = \lg s'_y + \lg s'_z \leq 2\lg[(s'_y + s'_z)/2] \leq 2\lg[s'_x/2] = 2r'_x - 2,$$

čo dosadíme vyššie a dostávame

$$T_{\text{amort}} \leq 2 + 2r'_x - 2 - 2r_x = 2(r'_x - r_x).$$

**Prípad „cik“:** Prenechávame čitateľovi. □

## Dôsledky

**Vyváženosť:** Pre $w_x = 1$ dostávame $r_{\text{koreň}} = \lg n$, takže amortizovaná zložitosť je $O(\lg n)$.

**Statická optimálnosť alias Veta o entropii:** Nech $f_x \geq 1$ je frekvencia, s ktorou splayujeme $x$, $m$ je celkový počet operácií, $p_x = f_x/m$; potom zložitosť $m$ operácií je $O(n \log n + m + \sum f_x \lg(m/f_x)) = O(m + m\sum p_x \lg(1/p_x))$; teda splay($x$) má amortizovanú zložitosť približne $O(1 + \lg(1/p_x))$. Hodnota $H = p_x \lg(1/p_x)$ je entropia pravdepodobnostného rozdelenia. Z teórie informácie vyplýva, že ak poznáme $f_x$, najlepší statický strom dosahuje zložitosť zhruba $mH$ – splay strom dosahuje konštantný násobok a to *bez znalosti* $f_x$!

Dôkaz: zvoľme $w_x = f_x$, potom $r_{\text{koreň}} = \lg m$ a $r_x = \lg f_x$, dosadíme do vety o prístupe.

**Veta o statickom prste:** Zvoľme si prst – vrchol $p$; amortizovaná zložitosť splay($x$) je približne $O(\lg(2 + |x - p|))$, kde $|x - p|$ je vzdialenosť (počet prvkov) medzi $x$ a $p$. Presnejšie ľubovoľných $m$ operácií bude mať zložitosť $O(n \log n + m + \sum \lg(2 + |x - p|))$. Inými slovami, ak často pristupujeme k prvkom blízko $p$, prístup je rýchly.

Dôkaz: zvoľme $w_x = 1/(|x - p| + 1)^2$; $s_{\text{koreň}} < 2\sum_{k=1}^{\infty} 1/k^2 = \pi^2/6 = O(1)$, dosadíme do vety o prístupe.

**Veta o pracovnej množine:** Nech $t_i(x)$ je počet rôznych prvkov (vrátane $x$), ktoré sme splayovali odkedy sme naposledy vysplayovali $x$ pred časom $i$; potom splay($x$) trvá $O(1 + \lg t_i(x_i))$ amortizovane. Inými slovami, ak stále pristupujeme iba k malej „pracovnej“ množine prvkov, čas je logaritmický od veľkosti pracovnej množiny.

Náčrt dôkazu: Váhy budeme meniť; v čase $i$ zvoľme $w_x = 1/t_i(x)^2$. Potom $s_{\text{koreň}} = \sum_{k=1}^{\infty} 1/k^2 = \pi^2/6$, čiže splay($x_i$) trvá $O(1 + \lg(O(1)/t_i(x_i)^{-2})) = O(1 + \lg t_i(x_i))$. Treba ešte overiť, že sme s meniacimi sa váhami nepodvádzali; ako sa zmenia váhy? Všetkým prvkom, ktoré sme splayovali od posledného splay($x_i$) sa $t_i(y)$ zvýši o 1; vrcholom, ktorých sme sa odvtedy nedotkli sa váha nezmení a prvok $x_i$ bude mať váhu 1. Inými slovami, ak je $t_i(x_i) = k$, potom $t_{i+1}(y)$ sa zmení takto: $k \to 1$, $1 \to 2$, $2 \to 3$,$\ldots k-1 \to k$. Teda $w_{x_i}$ vzrastie o $< 1$ a ostatné váhy klesnú alebo ostanú nezmenené, teda $\Delta\Phi < 1$.

## Ďalšie vlastnosti splay stromov

**Veta o skenovaní:** Ak pristupujeme postupne k prvkom $1, 2, 3, \ldots, n$, celkový čas je $O(n)$.

**Veta o dynamickom prste:** prístup ku $x_i$ trvá $O(\lg(2 + |x_i - x_{i-1}|))$, teda prístup blízko predošlému prvku je rýchly. Z tejto vety vyplýva veta o skenovaní aj veta o statickom prste. Dôkaz je veľmi ťažký.

**Hypotéza o obojsmernej fronte:** Ak splay strom používame ako deque, teda vkladáme a vyberáme prvky iba zo začiatku alebo konca, čas bude $O(m)$ (amortizovane). Zatiaľ najlepší dokázaný odhad je $O(m\alpha(m))$.

**Hypotéza o split strome:** Split strom je dátová štruktúra, ktorá podporuje operácie make($x_1, \ldots, x_n$) – vytvorenie stromu a split($x$) – vráti $x$ a rozdelí strom na 2 split stromy s prvkami $< x$ a $> x$. Existuje algoritmus, kde make a $n\times$split trvá $O(n)$; predpokladá sa, že splay strom dosahuje rovnakú zložitosť. Zatiaľ najlepší dokázaný odhad je $O(n\alpha(n))$.

**Zjednotená hypotéza:** Zovšeobecňuje vlastnosť pracovnej množiny a dynamického prsta: ak sme nedávno pristupovali k prvku, ktorý je blízko, prístup bude rýchly: $O(\lg \min_y[t_i(y) + |x_i - y| + 2])$ amortizovane.

**Hypotéza o dynamickej optimálnosti:** Splay strom je len konštantný násobok od najlepšieho možného BST algoritmu, ktorý pozná celú postupnosť prístupov dopredu.

# Kapitola 11

# Dynamická optimálnosť

# Časť IV

# Geometria

# Kapitola 12

# $k$-d strom

# Kapitola 13

# Rozsahový strom

# Kapitola 14

# Perzistentné dátové štruktúry

TODO: Motivácia: funkcionálne jazyky, redux (react), ľahké kopírovanie/snapshotovanie; zdieľanie v aplikáciách kde máme viacero paralelných vlákien, niektoré chcú čítať dáta, niektoré chcú zapisovať (meniť)

```
import qualified Data.Set as S

main = do
  let a = S.fromList [1, 2, 3]
  let b = S.insert 5 a
  let c = S.delete 1 b
  let d = S.delete 1 a
  print a
  print b
  print c
  print d
```

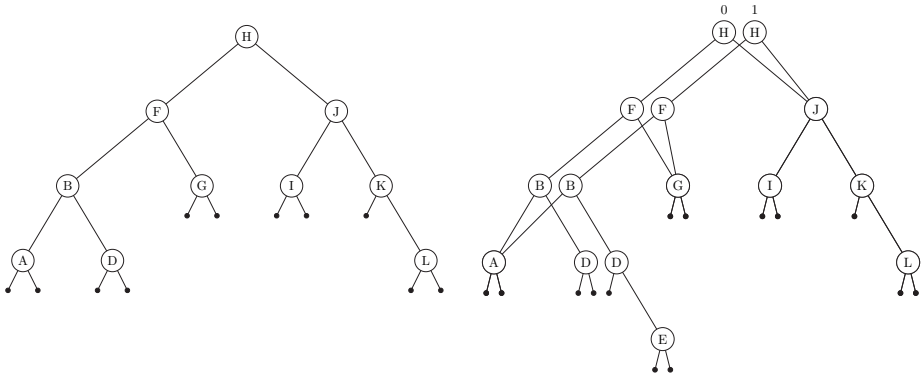
$a = \{1, 2, 3\}$, $b = \{1, 2, 3, 5\}$, $c = \{2, 3, 5\}$, $d = \{2, 3\}$.

**Riešenie #−2: Kópia celej dátovej štruktúry.**

**Riešenie #−1: Pamätáme si zmeny.**

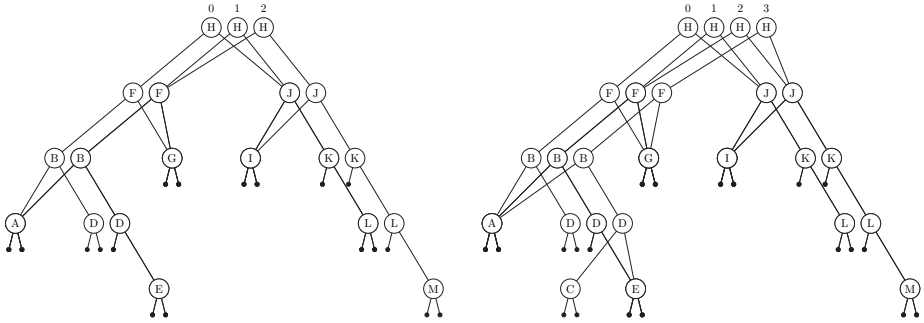
**Riešenie #0: Nejaká kombinácia kopírovania a ukladania zmien.** Základné princípy perzistencie si najlepšie ukážeme na jednoduchom a dobre známom príklade: *binárnom vyhľadávacom strome* (BVS).

Predpokladajme, že máme obyčajný binárny vyhľadávací strom, ktorý podporuje operácie `insert`, `delete` a `find`. Každá z týchto operácií mení iba vrcholy na ceste od koreňa k listu.




(a) Počiatočný stav binárneho stromu.

(b) V čase 1 vložíme nový prvok `E`.




(c) V čase 2 vložíme nový prvok `M`.




(d) V čase 3 vložíme nový prvok `C`.

Obr. 14.1: Funkcionálny binárny vyhľadávací strom: pôvodné dáta nikdy nemeníme; namiesto toho vždy vytvoríme nový vrchol s pozmenenými údajmi.

**Riešenie #1: Kopírovanie cesty** Pri každej operácii skopírujeme iba tie vrcholy, ktoré sa skutočne zmenili – nemusíme kopírovať celú dátovú štruktúru. A ktoré to sú? Pozri príklad na obr. 14.1. Keď do stromu na obr. 14.1a vložíme prvok `E`, musíme

1. pridať nový vrchol `E`,
2. vytvoriť novú verziu vrcholu `D`: v klasickej (neperzistentnej) dátovej štruktúre by sme jednoducho zmenili smerník na pravého syna tak, aby ukazoval na vrchol `E`; keďže si však chceme zachovať históriu, ponecháme si aj starú verziu `D` (bez pravého syna), aj novú verziu `D`, ktorá má pravého syna `E`,
3. vytvoriť novú verziu vrcholu `B`: nová verzia bude mať pravého syna – novú verziu vrcholu `D`; ľavý syn môže naďalej ukazovať na starú verziu vrcholu

`A`, pretože tento podstrom sa vôbec nezmenil; všimnite si, že nová verzia `B` sa líši od starej: nová verzia má vnuka `E`, zatiaľ čo stará verzia žiadneho vnuka nemá,

4. vytvoriť novú verziu vrcholu `F`, ktorý bude ukazovať na novú verziu `B` (pravý syn sa nezmenil), a

5. vytvoriť novú verziu koreňa `H`, ktorá ukazuje na novú verziu ľavého podstromu a starú verziu pravého podstromu (pretože táto časť stromu sa nezmenila).

Všimnite si, že sa snažíme zdieľať čo najviac štruktúry medzi starou a novou verziou. Výsledkom je, že kopírujeme iba vrcholy na ceste od koreňa po novo pridaný vrchol. Každému vrcholu na tejto ceste sa zmení práve jeden podstrom, zatiaľ čo druhý ostane nezmenený a môže byť zdieľaný.

Na obr. 14.1c a 14.1d je zobrazený stav štruktúry po tom, ako vložíme prvky `M` a `C`. Podobným spôsobom vieme implementovať aj ďalšie operácie, ako je vymazávanie, a dokonca aj vyvažovanie pomocou rotácií.

Všimnite si, že na konci máme štyri korene – teda štyri rôzne verzie stromu, zodpovedajúce časom $0, 1, 2, 3$. Ak chceme zistiť, či bol prvok $x$ prítomný v strome v čase $t$, jednoducho začneme vyhľadávanie v koreni vo verzii $t$.

Toto riešenie je navyše plne *funkcionálne*: žiadne dáta sa nikdy nemenia, iba vznikajú nové verzie. Tento prístup umožňuje meniť nielen poslednú verziu stromu, ale úplne ľubovoľnú historickú verziu; dokonca umožňuje kombinovať rôzne verzie, napríklad pri spájaní alebo delení stromov (pozri, ako sme tieto operácie definovali v kapitole o splay stromoch).

Čas aj pamäť sú úmerné počtu vykonaných zmien. Ak udržiavame strom vyvážený, každá operácia trvá $O(\log n)$ času a vytvorí navyše $O(\log n)$ nových vrcholov, teda spotrebuje $O(\log n)$ dodatočnej pamäte.

**Dá sa to lepšie?** Vo striktne funkcionálnom modeli, teda v situácii, keď *žiadne* dáta nesmieme meniť, odpoveď zrejme znie: nie. Každá operácia totiž nutne mení (presnejšie: vytvára nové verzie) $\Theta(\log n)$ vrcholov na ceste od koreňa k listu, a tomuto sa jednoducho nedá vyhnúť.

Preto teraz zmeníme model. Dovolíme si *mutácie* – teda prepísanie existujúcich dát – a zároveň si úlohu mierne zjednodušíme. Namiesto plnej perzistentnosti sa zameriame na *čiastočnú perzistentnosť*.

Budeme predpokladať, že zmeny dátovej štruktúry (vloženie alebo vymazanie prvku) môžeme vykonávať *iba v najnovšej verzii*. Staršie verzie sú nemenné a slúžia len na čítanie. Napriek tomu si chceme pamätať *celú históriu* štruktúry a budeme chcieť vedieť odpovedať na otázku *find*$(x, t)$: nachádzal sa prvok $x$ v strome vo verzii (v čase) $t$?

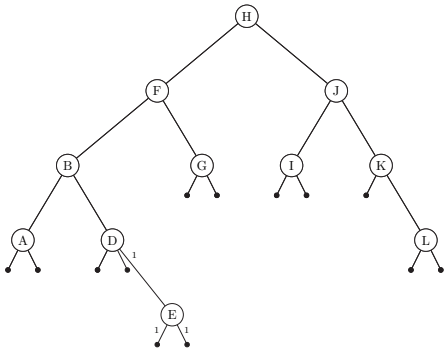
**Riešenie #2: Veľké vrcholy** Základná myšlienka je presne opačná ako pri kopírovaní cesty. Namiesto toho, aby sme pri každej zmene vytvárali nové vrcholy, necháme každý vrchol „rásť“ a pamätať si svoju vlastnú históriu.

Konkrétne: v klasickom binárnom vyhľadávacom strome má každý vrchol smerníky `left` a `right` na ľavého a pravého syna. Vo veľkých vrcholoch budeme mať namiesto týchto smerníkov dve *polia* `left` a `right`, ktoré si budú pamätať zoznam dvojíc

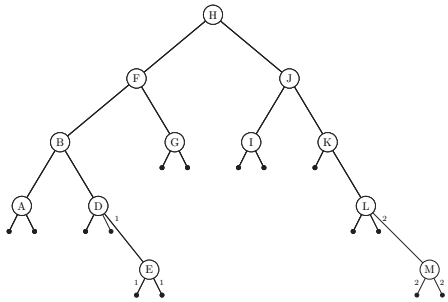
$$(\text{čas}, \text{hodnota})$$

ktoré hovoria: „od tohto času smerník ukazuje sem“.

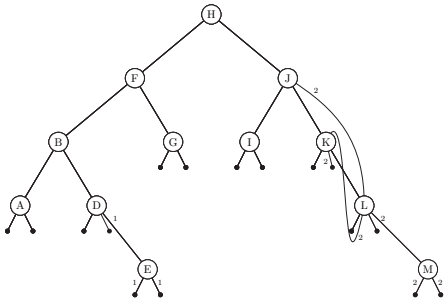
Ako prebieha aktualizácia? Keď vkladáme alebo vymazávame prvok v najnovšej verzii, jednoducho do poľa `left` alebo `right` v niektorých vrcholoch pridáme *nový* záznam s aktuálnym časom. Zvyšok štruktúry ostáva nezmenený. Pozri obr. 14.2.



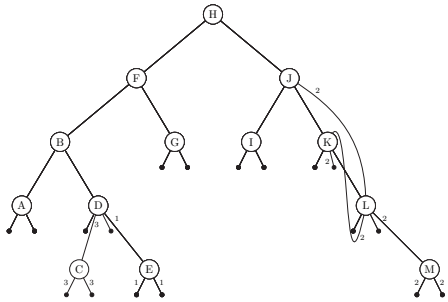
(a) Strom v čase 1 po vložení `E`.



(b) V čase 2 vložíme `M`.



(c) Keďže podstrom `K`—`L`—`M` je nevyvážený, môžeme v čase 2 ešte zrotovať vrchol `L`.



(d) V čase 3 vložíme prvok `C`.

Obr. 14.2: Čiastočne perzistentný BVS s veľkými vrcholmi. Každý vrchol si pamätá celú históriu pre ľavý a pravý smerník. Napríklad na konci, na obr. (d) vrchol D obsahuje dve polia: `left`: $\boxed{(0,\perp)}\boxed{(3,\mathrm{C})}$ a `right`: $\boxed{(0,\perp)}\boxed{(1,\mathrm{E})}$.

Dodatočná pamäť je úmerná počtu zmien, ktoré vykonáme. Ak použijeme

obyčajný binárny vyhľadávací strom, tak vloženie zmení len jeden smerník a vymazanie zmení najviac dva. Môžeme však použiť aj vyvažované stromy – napríklad AVL stromy alebo červeno-čierne stromy – kde vyvažovanie mení iba $O(1)$ hodnôt amortizovane.

Červeno-čierne stromy sú v tomto ohľade obzvlášť výhodné: garantujú konštantný počet rotácií aj v najhoršom prípade a zmenia nanajvýš logaritmicky veľa farieb. Históriu farieb si však pamätať nemusíme, pretože farba sa pri vyhľadávaní nepoužíva – farbu môžeme pokojne natvrdo prepísať. Z pohľadu perzistencie nás zaujímajú len zmeny smerníkov.

Výsledkom je, že každá aktualizácia pridá iba konštantné množstvo novej pamäte, čo je optimálne a predstavuje výrazné zlepšenie oproti riešeniu s kopírovaním cesty, kde každá operácia vyžadovala $O(\log n)$ nových vrcholov.

Na druhej strane za túto úsporu pamäte zaplatíme miernym spomalením vyhľadávania. Pri vyhľadávaní začneme v koreni, porovnávame hľadaný prvok s kľúčom vo vrchole a pokračujeme doľava alebo doprava. Rozdiel je v tom, že pri každom kroku musíme zistiť, kam daný smerník ukazoval *v čase* $t$.

To vieme určiť binárnym vyhľadávaním v histórii príslušného vrcholu. Ak má vrchol zaznamenaných $O(t)$ zmien, trvá tento krok $O(\log t)$. Keďže prejdeme $O(\log n)$ vrcholov, celkový čas vyhľadávania sa zhorší z $O(\log n)$ na

$$O(\log n \times \log t).$$

Máme teda jedno riešenie (kopírovanie cesty), ktoré má dobrý čas, ale zlú pamäť a jedno riešenie (veľké vrcholy), ktoré má dobrú pamäť, ale zlý čas. Prirodzene sa teda natíska otázka: *Existuje riešenie, ktoré by spojilo výhody oboch prístupov?*

Odpoveď je áno.

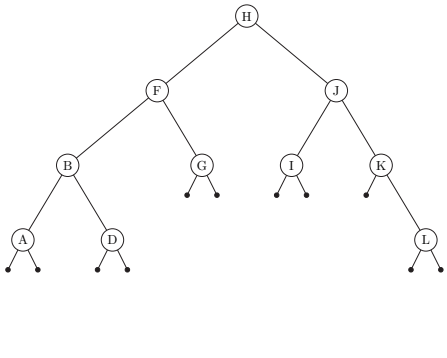
**Riešenie #3: Limitované vrcholy** Myšlienka je prekvapivo jednoduchá a dá sa chápať ako kompromis medzi predchádzajúcimi dvoma prístupmi. Namiesto toho, aby sme si vo vrchole pamätali *celú* históriu zmien (ako pri veľkých vrcholoch), povolíme každému vrcholu uchovávať len *obmedzený počet* historických záznamov.

Konkrétne, každý vrchol bude mať okrem svojej aktuálnej hodnoty iba jedno extra políčko, kde si pamätá najviac jednu zmenenú hodnotu (pozri obr. 14.3).

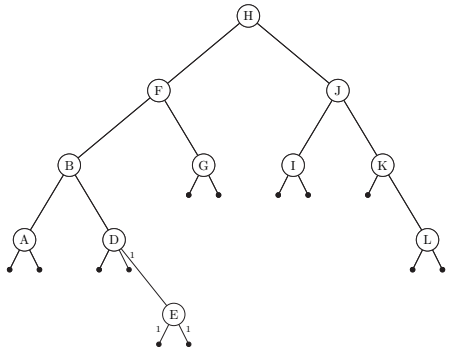
Ak potrebujeme spraviť vo vrchole zmenu, sú dve možnosti:

a) Extra políčko je prázdne. V tom prípade zmenu jednoducho zapíšeme do tohto políčka. Kokrétne potrebujeme špecifikovať 1. ktorá hodnota sa mení (ľavý alebo pravý smerník), 2. odkedy zmena platí a 3. aká je nová hodnota.

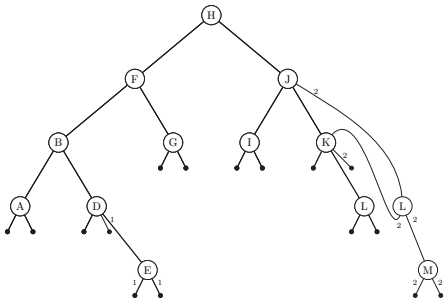
b) Extra políčko je už zaplnené. V tomto prípade vytvoríme *novú kópiu* vrcholu, do ktorej zapíšeme aktuálne hodnoty všetkých políčok. Starý vrchol zostane zachovaný a bude slúžiť pre staršie verzie stromu, nový vrchol sa stane súčasťou novších verzií. Samozrejme, musíme následne upraviť aj smerník v otcovi, aby ukazoval na novú verziu vrcholu.



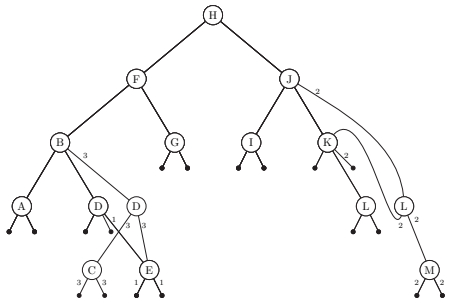

(a) Počiatočný stav binárneho stromu.




(b) V čase 1 vložíme nový prvok `E`.




(c) V čase 2 vložíme `M` a zrotujeme vrchol L. Pre vrchol L to znamená 2 zmeny, takže vyrobíme rovno novú verziu.




(d) V čase 3 vložíme prvok `C`. Keďže vrchol D už jednu zmenu absolvoval, vyrobíme novú verziu D.

Obr. 14.3: Čiastočne perzistentný BVS s limitovanými vrcholmi. Každý vrchol má jedno extra políčko, kde si pamätá jednu zmenenú hodnotu, napríklad na obr. (d) má vrchol J ľavý smerník na I, pravý smerník na K a navyše si pamätá zmenu $(r, 2, \mathrm{L})$, teda že pravý smerník od času 2 ukazuje na vrchol L.

Ak sú teda políčka na zmeny zaplnené, vrcholy kopírujeme podobne ako pri kopírovaní cesty. Rozdiel je v tom, že ku kopírovaniu dochádza len *zriedkavo*. Každý vrchol, ktorý kopírujeme, musel byť predtým zaplnený (takže si dokázal našetriť...), čo využijeme pri amortizovanej analýze.

Tvrdíme, že počet kopírovaných vrcholov je amortizovane konštantný.

Budeme predpokladať, že skopírovanie jedného vrcholu stojí 1$ a každá operácia (vloženie alebo vymazanie) vykoná len $O(1)$ lokálnych zmien smerníkov. Ukážeme, že každá operácia si vystačí s $O(1)$ dolármi.

Dôkaz je jednoduchý: Invariant, ktorý budeme dodržiavať je, že každý *plný*

vrchol má našetrený 1$. Inými slovami, definujeme potenciál

$$\Phi(T) = \#\text{plných vrcholov.}$$

Pozrime sa teraz na jednotlivé prípady:

a) Vrchol je prázdny. Zapíšeme zmenu do extra políčka. Tým sa vrchol stane plným, čo zaplatíme prideleným dolárom.

b) Vrchol je plný. Vrchol skopírujeme, čo zaplatíme dolárom, ktorý mal tento vrchol našetrený. Následne musíme rekurzívne zmeniť smerník v otcovi, čo môže vyvolať ďalší krok rovnakého typu, avšak stále nám ostáva dolár pridelený na operáciu.

Inými slovami, ak jedna zmena vytvorí kaskádu kopírovania $k$ vrcholov (skutočná cena je $k$) a na konci sa zaplní jeden nový vrchol, potenciál sa zmení o

$$\Delta\Phi = -k + 1,$$

pretože pri každom kopírovaní vytvoríme prázdny vrchol a potenciál klesne o 1 a kvôli jednému poslednému zaplnenému vrcholu stúpne o 1.

Keďže zmien je len konštantne veľa a na každú nám stačí jeden dolár, celková zložitosť je amortizovane konštantná.

**Zrhnutie.**

| | *pamäť* | *find*$(x,t)$ |
|---|---|---|
| kopírovanie celej DŠ | $+O(n)$ | $O(\log n)$ |
| ukladanie zmien | $+O(1)$ | $O(n)$ |
| kopírovanie cesty | $+O(\log n)$ | $O(\log n)$ |
| veľké vrcholy | $+O(1)$ | $O(\log n \times \log t)$ |
| limitované vrcholy | $+O(1)$ | $O(\log n)$ |

## 14.1 Všeobecná konštrukcia

pointer machine

## 14.2 Problém najbližšej pošty

Predstavme si, že máme danú mapu mesta a na nej sú zakreslené všetky pozície, kde sa nachádza pošta. Tieto dáta sa nemenia (alebo sa menia len veľmi zriedka), takže si môžeme dovoliť ich dôkladne predspracovať.

Jedného dňa si idem mestom a spomeniem si, že potrebujem poslať list. Kde sa nachádza najbližšia pošta?

Formálne: dané sú body $p_1, p_2, \ldots, p_n$ v rovine. Máme čas na ich predspracovanie. Následne prichádzajú dotazy:

Je daný bod $q$. Ktorý z bodov $p_i$ je najbližší ku $q$?

Budeme predpokladať, že vzdialenosť meriame pomocou euklidovskej vzdialenosti.

Samozrejme, pre každý dotaz vieme v lineárnom čase prejsť všetky body, spočítať vzdialenosť od bodu $q$ ku každému $p_i$ a vybrať ten najbližší. Ak je však bodov aj dotazov veľa, takéto riešenie je neakceptovateľné. Otázka teda znie: Dá sa využiť fakt, že body $p_i$ sú statické a dotazy prichádzajú až neskôr?

Predstavme si nasledujúcu myšlienku. Rozdelíme rovinu na oblasti (nazvime ich *rajóny*) tak, že každá pošta bude mať svoj vlastný rajón, v ktorom je práve ona najbližšia. Môžeme si to predstaviť aj vizuálne: každej pošte priradíme jednu farbu a celú mapu mesta zafarbíme podľa toho, ktorá pošta je v danom bode najbližšie.

Samozrejme, nechceme túto úlohu riešiť bod po bode pre nespočítateľne nekonečne veľa bodov roviny. Dúfajme preto, že tieto rajóny majú nejakú rozumnú štruktúru a dajú sa opísať konečným (a nie príliš veľkým) množstvom údajov.

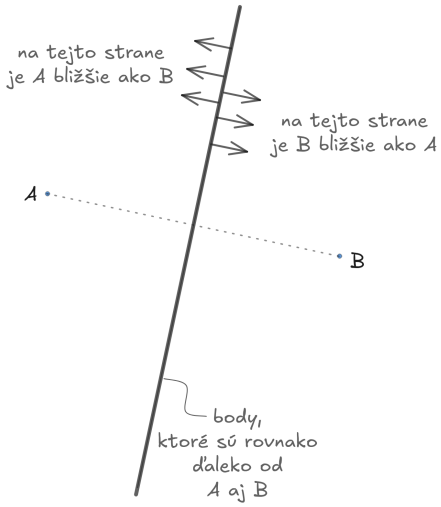
Ak by sme takúto „ofarebnú mapu“ mali, odpoveď na dotaz by sa výrazne zjednodušila: problém nájdenia najbližšej pošty by sa zredukoval na inú otázku:

Do ktorého rajónu patrí bod $q$?

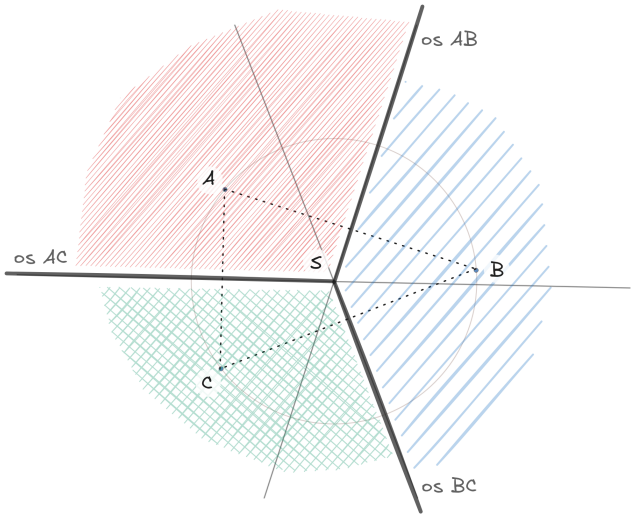
Skúsme sa teraz zamyslieť nad tým, ako takéto rajóny vlastne vyzerajú. Zoberte si papier a ceruzku a skúste si nakresliť niekoľko príkladov: začnite s dvoma bodmi, potom s troma, štyrmi, ... Čo môžete povedať o tvaroch hraníc medzi jednotlivými oblasťami?

Začnime úplne jednoducho.

**Dva body.** Ak máme iba dve pošty $A$ a $B$, hranica medzi ich rajónmi je množina bodov, ktoré majú od oboch pošt rovnakú vzdialenosť. V rovine ide o priamku – kolmicu na úsečku $AB$ v jej strede (a.k.a „os úsečky $AB$“). Rovina sa teda rozdelí na dve polroviny.

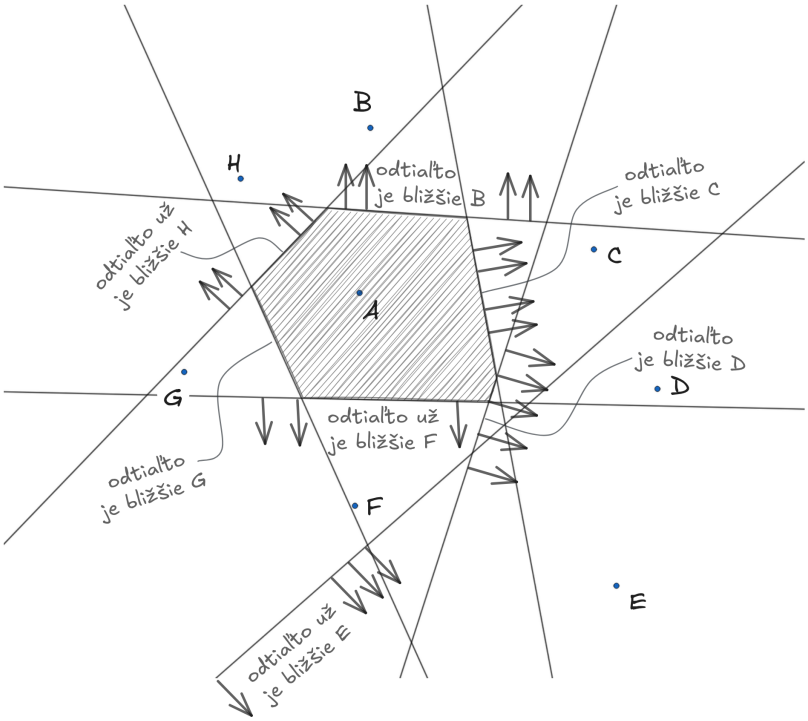



**Tri body.** Pri troch bodoch sa situácia skomplikuje len o málo. Každá dvojica bodov určuje takúto deliacu priamku. Tieto tri osi úsečiek sa stretnú v jednom bode, ktorý je stredom opísanej kružnice trojuholníka $ABC$. Toto je bod, ktorý je rovnako ďaleko od všetkých troch bodov. Hranice rajónov budú polpriamky začínajúce v tomto strede, idúce donekonečna.




**Všeobecný prípad.** Pre ľubovoľný počet bodov môžeme rozdelenie roviny na oblasti získať takto: Vezmime ľubovoľný bod $p_i$; spočítajme osi úsečiek so všetkými ostatnými bodmi $p_j$. Každá os nám rovinu rozdelí na dve polroviny –

tam, kde je bližšie $p_i$ a tam, kde je bližšie iný bod $p_j$. Oblasť, kde $p_i$ je najbližší zo všetkých teda musí byť priesečníkom všetkých týchto polrovín.




Výsledkom je rozdelenie roviny na konvexné mnohouholníky (prienikom konvexných oblastí je konvexná oblasť), ktoré spolu presne vypĺňajú celú rovinu (žiadne diery, žiadne prekryvy).

Toto rozdelenie má dokonca meno.

> *Voronoiov diagram* množiny bodov $p_1, \ldots, p_n$ je rozdelenie roviny na oblasti, kde každá oblasť obsahuje práve tie body, ktoré sú najbližšie k jednému konkrétnemu bodu $p_i$.

Jednoduchý spôsob spočítať každú oblasť ako priesečník polrovín by zabral až $O(n^2 \log n)$ času, ale existujú oveľa efektívnejšie algoritmy, ktoré dokážu Voronoiov diagram spočítať v $O(n \log n)$.

## 14.3 Lokalizácia bodu v rovine

Ak už máme Voronoiov diagram predpočítaný, odpoveď na dotaz „kde je najbližšia pošta?“ sa zredukuje na problém nájsť, v ktorej oblasti sa bod $q$ nachádza (tzv. *planar point location*):

> Dané je rozdelenie roviny na mnohouholníkové oblasti. Tieto oblasti môžeme vopred predspracovať. Následne budú prichádzať dotazy typu: pre daný bod $q$ nájdi oblasť, v ktorej sa nachádza.

Táto úloha je zaujímavá aj sama o sebe. Predstavme si napríklad používateľa, ktorý má otvorených viacero aplikácií, alebo webovú stránku rozdelenú na množstvo interaktívnych prvkov (tlačidlá, menu, panely, odkazy). Pri každom kliknutí myšou potrebujeme rýchlo zistiť, *do ktorej oblasti obrazovky klik patril* a ktorá časť rozhrania má na túto udalosť reagovať. Podobné problémy sa objavujú aj v geografických informačných systémoch, v počítačovej grafike či v simuláciách.

Prvé riešenia Kirkpatrick 1983 a Edelsbrunner, Guibas a Stolfi 1986 sú pomerne komplikované. My si ukážeme elegantné riešenie od Sarnak a Tarjan 1986.

Prvá myšlienka pochádza už od Dobkin a Lipton 1976.

**Dobkinov/Liptonov nápad: rozdelenie na pásy.** Rozdeľme rovinu na zvislé pásy podľa všetkých $x$-ových súradníc vrcholov rozdelenia. V jednom pevnom páse sa už žiadne dve hrany nestretávajú ani nerozpájajú. Všetky vrcholy ležia na hraniciach pásov. Vnútri pásov sú iba rovné úsečky spájajúce ľavú a pravú hranicu pásu. Vďaka tomu môžeme v každom páse uvažovať zvislé poradie hrán zhora nadol.

Riešenie potom vyzerá nasledovne:

- Najskôr nájdeme pás, do ktorého patrí bod $q$ (binárnym vyhľadávaním podľa $x$).
- V tomto páse si udržiavame zoznam hrán usporiadaný zhora nadol; stačí nájsť, medzi ktorými dvoma hranami leží bod $q$ (binárnym vyhľadávaním podľa $y$).

Tento prístup dáva veľmi rýchle dotazy: stačia dve binárne vyhľadávania, takže čas na dotaz je $O(\log n)$. Cena za jednoduchosť je však vysoká: pri prechode z pásu do pásu sa vertikálne poradie hrán mení, a ak si ho budeme pre každý pás ukladať zvlášť, skončíme s pamäťou $O(n^2)$ a predspracovaním $O(n^2 \log n)$.

Pásy sú si veľmi podobné. Keď sa posúvame zľava doprava, hranice pásov vždy prechádzajú cez nejaký vrchol subdivízie: v tomto $x$-ovom momente niektoré hrany v aktuálnom páse končia, iné naopak začínajú. Znamená to, že usporiadaný zoznam hrán v susedných pásoch sa líši len *málo*.

Namiesto toho, aby sme každý pás triedili od nuly, urobíme nasledovné:

- pre úplne prvý pás si zistíme vertikálne poradie všetkých hrán, ktoré ho pretínajú, a uložíme si ho do vyhľadávacej štruktúry (napr. vyváženého BVS),
- pri prechode do ďalšieho pásu iba *odoberieme* hrany, ktoré v deliacej priamke končia, a *pridáme* hrany, ktoré v nej začínajú.

Keďže v jednom vrchole začne/končí v priemere len konštantne veľa hrán, prechod do ďalšieho pásu znamená v priemere len $O(1)$ aktualizácií v BVS, teda časovo $O(\log n)$ na pás. Presnejšie: všetkých zmien je spolu rovný dvakrát počtu hrán, čo je $O(n)$, keďže rozdelenie roviny je planárny graf. Celkovo tak vieme všetky pásy spracovať v čase $O(n \log n)$.

Tu však narazíme na zdanlivo drobný, ale zásadný problém: pri dotaze potrebujeme vedieť vyhľadávať v *konkrétnom* páse, do ktorého patrí bod $q$. Teda nechceme mať iba *poslednú* verziu BVS po tom, čo prejdeme všetky pásy, ale chceme si zapamätať aj všetky *staršie* verzie – jednu pre každý pás.

A to je presne úloha pre perzistentné dátové štruktúry!

Os $x$ si predstavme ako čas: pri zvyšujúcom sa $x$ sa náš BVS postupne mení (hrany pribúdajú/ubúdajú), no vďaka perzistencii si zároveň zachováme prístup ku každej predchádzajúcej verzii. Inými slovami, pre každý pás budeme mať koreň ukazujúci na jeho vlastnú verziu stromu.

Ak použijeme perzistentný BVS realizovaný *limitovanými vrcholmi*, jedna aktualizácia (insert/delete) nás stojí iba $O(1)$ pamäť navyše. Keďže spolu spravíme iba $O(n)$ aktualizácií, celková pamäť bude lineárna.

Celé predspracovanie zvládneme v $O(n \log n)$ (zostrojenie prvého pásu + $O(n)$ zmien pri prechodoch medzi pásmi) a odpovedať na otázky budeme vedieť stále v $O(\log n)$ (binárne vyhľadanie pásu podľa $x$ + vyhľadanie v príslušnej verzii BVS podľa $y$).

Takto sme nahradili kvadratické kopírovanie „poradia hrán pre každý pás“ jednou dynamickou štruktúrou, ktorá sa pri posune po osi $x$ mení, ale všetky jej historické verzie ostávajú k dispozícii.

## Referencie

Dobkin, David a Richard J Lipton (1976). „Multidimensional searching problems“. In: *SIAM Journal on Computing* 5.2, s. 181–186.

Edelsbrunner, Herbert, Leonidas J Guibas a Jorge Stolfi (1986). „Optimal point location in a monotone subdivision“. In: *SIAM Journal on Computing* 15.2, s. 317–340.

Kirkpatrick, David (1983). „Optimal search in planar subdivisions“. In: *SIAM Journal on Computing* 12.1, s. 28–35.

Sarnak, Neil a Robert E Tarjan (1986). „Planar point location using persistent search trees“. In: *Communications of the ACM* 29.7, s. 669–679.

# Kapitola 15

# Hľadanie najbližšieho suseda

# Časť V

# Hešovanie

# Časť VI

# Stringológia

# Kapitola 16

# Sufixový strom

**Motivácia #1: Ako vyhľadávať vzorky v texte?** Najjednoduchšie riešenie je, samozrejme, *triviálne vyhľadávanie*: porovnáme vzorku so všetkými pozíciami v texte, čo trvá $O(m \times n)$, kde $n$ je dĺžka textu a $m$ dĺžka vzorky.

Klasické algoritmy ako Knuth-Morris-Prattov dokážu vzorku predspracovať v čase $O(m)$ a následne prejsť text v lineárnom čase $O(n)$. Praktické algoritmy typu Boyer-Moore či jeho varianty často fungujú ešte lepšie, najmä na prirodzených textoch.

Tieto prístupy však majú spoločnú jednu vec: *predspracúvajú vzorku* a hľadajú ju v pôvodnom texte. To je ideálne, ak vzorka zostáva rovnaká a text sa mení.

Čo ak sme v opačnej situácii? Text $T$ je obrovský a fixný, a chceme v ňom vyhľadávať veľa rôznych vzoriek. Predstavte si napríklad Wikipédiu, ktorá obsahuje desiatky gigabajtov textu a v ktorej chceme rýchlo nájsť dané slová alebo frázy. Alebo ľudskú DNA – približne 3 miliardy „báz“ (`A`, `C`, `G`, `T`) – a chceme v nej nájsť gén dlhý tisíce až desaťtisíce znakov.

Dokázali by sme si vopred predspracovať *text* tak, aby sme ho pri vyhľadávaní nemuseli prechádzať celý?

Ukážeme si, že s pomocou sufixových stromov dokážeme (po úvodnom predspracovaní) vyhľadávať v čase $O(m)$ (!), teda v čase úmernom dĺžke vzorky, nie celého textu.

**Motivácia #2: najdlhší spoločný podreťazec.** Problém najdlhšieho spoločného podreťazca dvoch reťazcov bol dlhé roky považovaný za ťažký: najlepšie známe riešenia mali zložitosť $O(n \log n)$ a existovali dohady, že lineárny čas je možno nemožný. V roku 1970 Donald Knuth dokonca vyslovil hypotézu, že lineárny algoritmus neexistuje.

Ukážeme si, že ak poznáme sufixové stromy, je táto úloha úplne jednoduchá a elegantné riešenie s lineárnou zložitosťou sa objaví takmer samo.

Keďže oba problémy, o ktorých sme sa zmienili, sú pomerne zložité, skúsme sa najskôr pozrieť na dve jednoduchšie úlohy. Predstavme si, že máme dané

texty

$$T_1, \ldots, T_d.$$

Chceme si ich predspracovať tak, aby sme dokázali riešiť nasledujúce otázky:

- *Vyhľadávanie prefixov.* Pre danú vzorku $P$ chceme rýchlo zistiť, ktoré z textov $T_i$ začínajú práve na $P$.
- *Najdlhší spoločný začiatok.* Chceme nájsť dvojicu textov, ktoré majú najdlhší spoločný prefix.

Tieto dve úlohy sú výrazne jednoduchšie než pôvodné problémy, no ich riešenia nás navedú na cestu k sufixovým stromom a poliam. Skúste sa nad nimi najprv zamyslieť – tieto problémy dokážete vyriešiť aj sami. Až potom pokračujte v čítaní.

**Vyhľadávanie prefixov.** Ako rýchlo nájsť všetky texty začínajúce na vzorku $P$? Začnime od najjednoduchšieho riešenia:

- *Triviálne riešenie.* Bez akéhokoľvek predspracovania stačí porovnať vzorku so všetkými textami. To trvá

  $$O(m \times d),$$

  kde $m$ je dĺžka vzorky a $d$ je počet textov.
- *Triedenie + binárne vyhľadávanie.* Ak si texty vopred zotriedime lexikograficky, môžeme následne hľadať pomocou binárneho vyhľadávania. Pri jednom porovnaní dvoch reťazcov prejdeme najviac $m$ znakov, no množinu prehľadávaných textov zmenšíme na polovicu, takže výsledná zložitosť je

  $$O(m \times \log d).$$

- *Písmenkový strom (trie).*[1] Ak si texty uložíme do prefixového stromu, pri vyhľadávaní jednoducho zídeme po ceste zodpovedajúcej vzorke $P$. Ak cesta existuje, všetky listy v podstrome pod touto cestou predstavujú texty začínajúce na $P$. Dĺžka zostupu je práve $m$, takže čas hľadania je

  $$O(m).$$

**Najdlhší spoločný začiatok.** Pre jednoduchosť predpokladajme, že každý text má dĺžku práve $n$. Ako nájsť dve slová s najdlhším spoločným prefixom?

- *Triviálne riešenie.* Môžeme porovnať každú dvojicu textov. Takých dvojíc je $d^2$ a porovnanie dvoch reťazcov trvá v najhoršom prípade $n$, čiže celková zložitosť je

  $$O(d^2 \times n).$$

[1] Tieto stromy sú známe pod veľa rôznymi menami: v angličtine trie (zo slova re*trie*val), prefixový strom, písmenkový strom, lexikografický strom...

- *Triedenie.* Ak si texty najprv zotriedime lexikograficky, stačí porovnať len dvojice, ktoré stoja v utriedenom poradí vedľa seba. Každý text teda porovnávame s najviac dvoma susedmi, takže riešenie beží v čase

$$O(d \times n).$$

- *Písmenkový strom (trie).* Keď texty vložíme do prefixového stromu, každý text zodpovedá jednej ceste z koreňa do niektorého listu. Dva texty majú spoločný začiatok presne tam, kde sa ich cesty zhodujú. Najdlhší spoločný prefix medzi ľubovoľnými dvoma textami teda nájdeme tak, že v strome hľadáme *najhlbší vrchol, ktorý má aspoň dvoch potomkov.*

  Tento vrchol predstavuje najdlhšiu cestu od koreňa, ktorá je spoločná pre aspoň dva texty, a jeho hĺbka je dĺžkou najdlhšieho spoločného začiatku. Vrchol ľahko nájdeme v lineárnom čase (od veľkosti stromu).

## 16.1 Štruktúra sufixového stromu

Ako sme videli, písmenkový strom je mimoriadne vhodný na úlohy týkajúce sa prefixov. Ponúka sa teda na prvé počutie šialená myšlienka:

> Každý podreťazec je predsa prefixom nejakého sufixu.

Čo keby sme teda vyrobili písmenkový strom, do ktorého vložíme *všetky sufixy* nášho textu?

Vezmime si napríklad text `MISSISSIPPI$`, ktorý má presne 12 neprázdnych sufixov:

$$\texttt{\$}, \texttt{I\$}, \texttt{PI\$}, \texttt{PPI\$}, \texttt{IPPI\$}, \ldots, \texttt{ISSISSIPPI\$}, \texttt{MISSISSIPPI\$}.$$

Ak zostavíme písmenkový strom obsahujúci všetky tieto reťazce, dostaneme strom zobrazený na obr. 16.1.

Na obr. 16.2 a 16.3 sú dva väčšie príklady: sufixové stromy pre text

`SHE␣SELLS␣SEASHELLS␣BY␣THE␣SEASHORE$`

a pre úsek DNA sekvencie

`ATAGACCGCCATTACATAGATGAGTATAGAGACT$`.

Na koniec textu vždy pridáme špeciálny symbol `$`, ktorý sa nikde inde v reťazci nevyskytuje. Tým zabezpečíme, že každý sufix skončí v samostatnom liste (a nie vo vnútornom vrchole).

Všimnite si, že každá cesta od koreňa k listu zodpovedá jednému sufixu, a teda každá cesta začínajúca v koreni zodpovedá nejakému podreťazcu daného reťazca.

Očividný problém však je, že všetky sufixy majú spolu dĺžku $\Theta(n^2)$. Takýto strom by zaberal príliš veľa pamäte – kvadratickú pamäť si pri textoch dlhých

miliardy znakov nemôžeme v žiadnom prípade dovoliť. Zároveň štruktúru, ktorá sama o sebe obsahuje $\Theta(n^2)$ znakov, nedokážeme zostrojiť rýchlejšie než v čase $O(n^2)$, takže aj časová zložitosť by bola zásadný problém.

Na druhej strane, ako vidíme na obr. 16.1–16.3, takéto sufixové triese obsahujú množstvo dlhých úsekov, ktoré sa vôbec nerozdeľujú. Prirodzené riešenie je teda jasné: každú takú maximalnú cestu bez vetvenia skomprimujeme a uložíme ako jedinú hranu (pozri obr. 16.4–16.6).

Takáto štruktúra sa všeobecne nazýva komprimovaný (alebo kompaktný) písmenkový, lexikografický či prefixový strom; stretnete sa tiež s názvami ako *radixový strom* alebo *Patricia trie*.

Všimnite si, že keďže do stromu vkladáme $n$ reťazcov (všetky sufixy), počet listov je $n$ a teda celý strom má iba lineárne veľa vrcholov aj hrán. To je dobrá správa.

Na druhej strane, zlá správa je, že štruktúra stále zaberá kvadratickú pamäť, pretože sme do nej vložili spolu $\Theta(n^2)$ znakov.

Lenže moment! Všetky reťazce, ktorými sú označené hrany, sú predsa len nejaké úseky pôvodného textu $T$. Takže namiesto ukladania kópií týchto podreťazcov si na každej hrane stačí zapamätať, *odkiaľ–pokiaľ* daný úsek v texte siaha (pozri obr. 16.7).

Takto si pri každej hrane v strome stačí zapamätať dve čísla – dva indexy do textu $T$, ktoré určujú začiatok a koniec príslušného úseku. V tejto reprezentácii má teda každá hrana len konštantnú veľkosť a celková pamäťová zložitosť je zrazu *lineárna*!

**Zovšeobecnený sufixový strom.** Štruktúru môžeme prirodzene zovšeobecniť aj pre množinu viacerých „dokumentov“

$$\mathcal{D} = \{T_1, T_2, \ldots, T_d\}.$$

Jednoducho zostrojíme sufixový strom, ktorý obsahuje *všetky sufixy všetkých dokumentov* (pozri malý príklad na obr. 16.8).

Predstavte si napríklad Wikipédiu, ktorá obsahuje milióny rôznych článkov, knižnicu všetkých kníh sveta, alebo databázy genómov (GenBank, Ensembl) so sekvenciami DNA/RNA najrozličnejších organizmov.

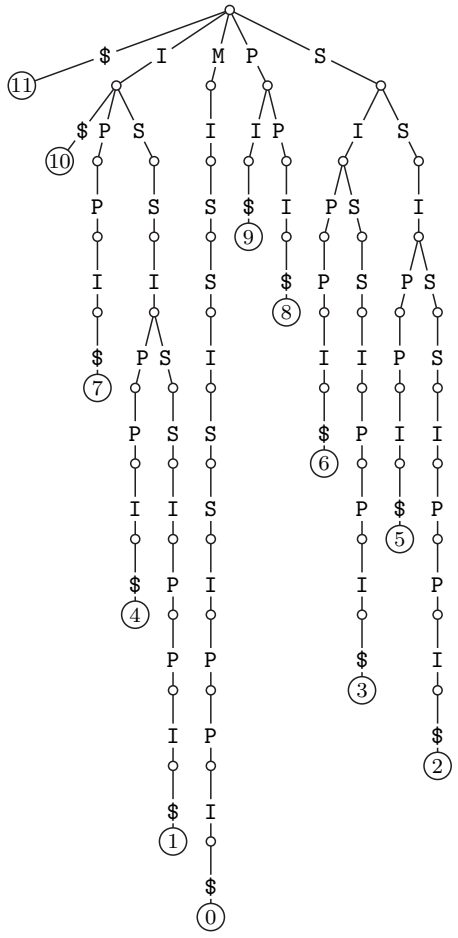
Zovšeobecnený sufixový strom môžeme zostrojiť tak, že každý dokument ukončíme iným špeciálnym znakom, ktorý sa v žiadnom texte nenachádza. Alternatívne môžeme zostrojiť obyčajný sufixový strom pre zreťazený text

$$T_1\texttt{\#}T_2\texttt{\#}T_3\texttt{\#}\cdots T_d\texttt{\#\$},$$

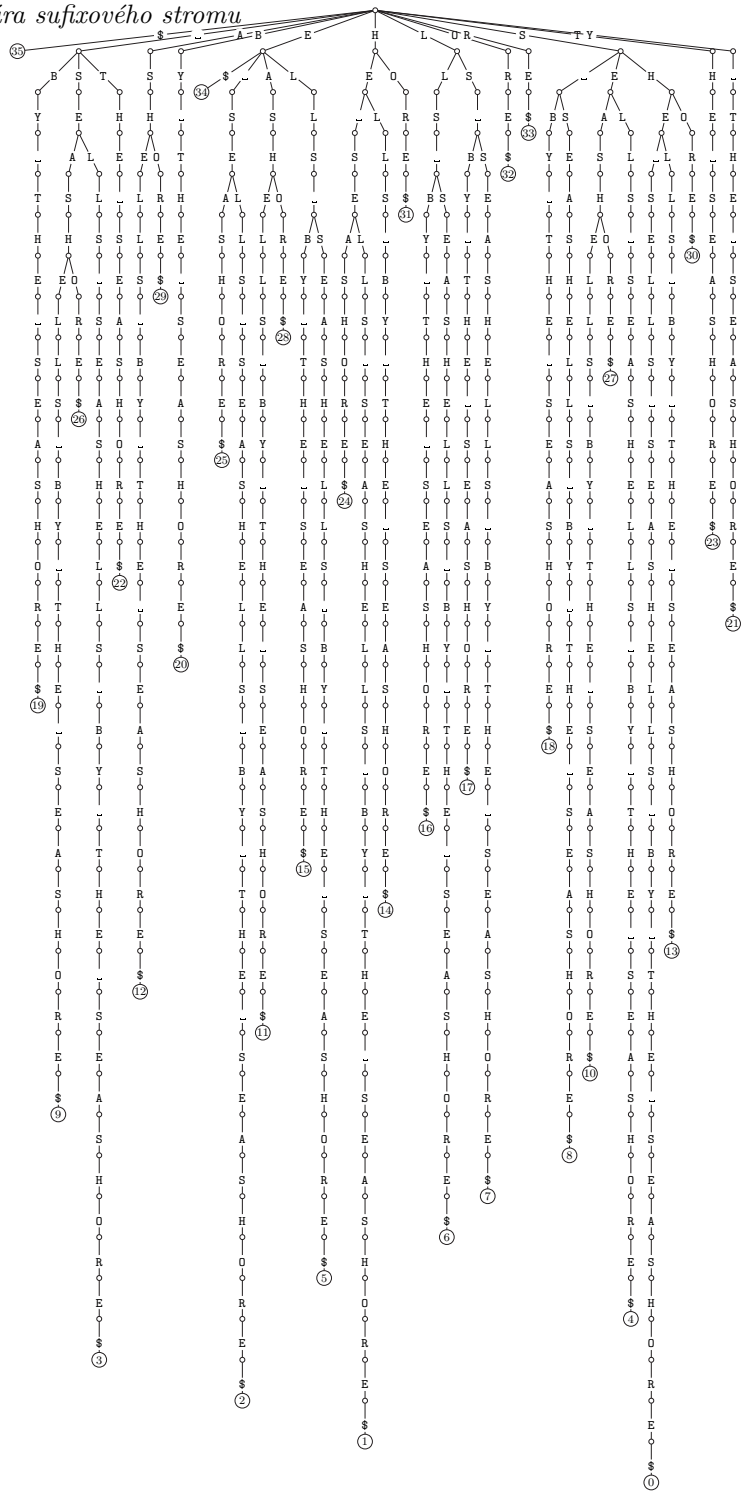
kde `#` a `$` sú dva špeciálne ukončovacie symboly, ktoré sa v žiadnom texte nenachádzajú. Jediný drobný rozdiel je v tom, že listy a hrany musia navyše špecifikovať, ktorého dokumentu sa daný sufix (alebo jeho časť) týka.

**~~Konštrukcia?~~ Aplikácie.** Sufixové stromy sa dajú zostrojiť v čase $O(n)$; ako to dosiahnuť, si však vysvetlíme až v kapitole o sufixových poliach. Najprv si

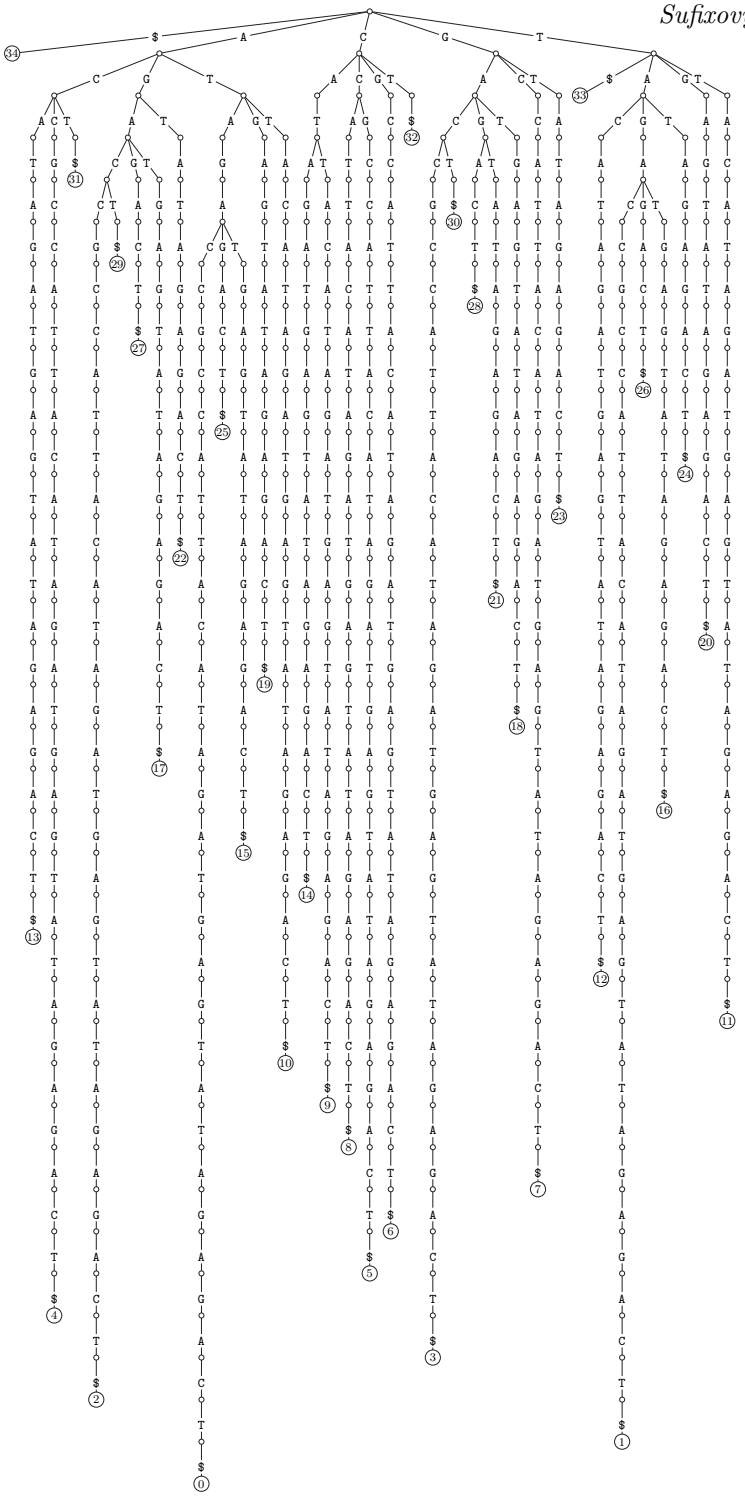
však ukážeme, aké užitočné sufixové stromy sú a „milión“ rôznych aplikácií, kde sa dajú využiť. Hovorí sa, že keď má človek kladivo, veľa problémov začne pripomínať klince. A sufixový strom je naozaj veľmi veľké kladivo na najrôznejšie problémy v stringológii.



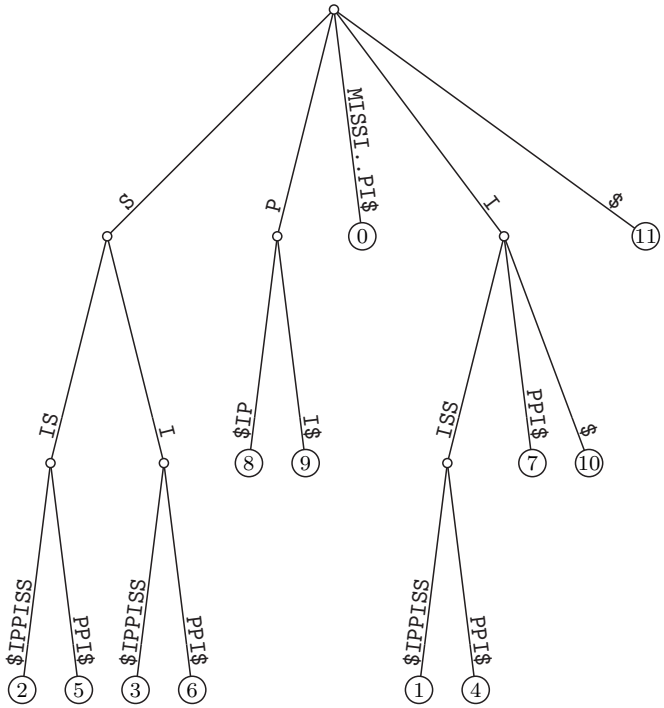

Obr. 16.1: Sufixový strom pre text „MISSISSIPPI$“.



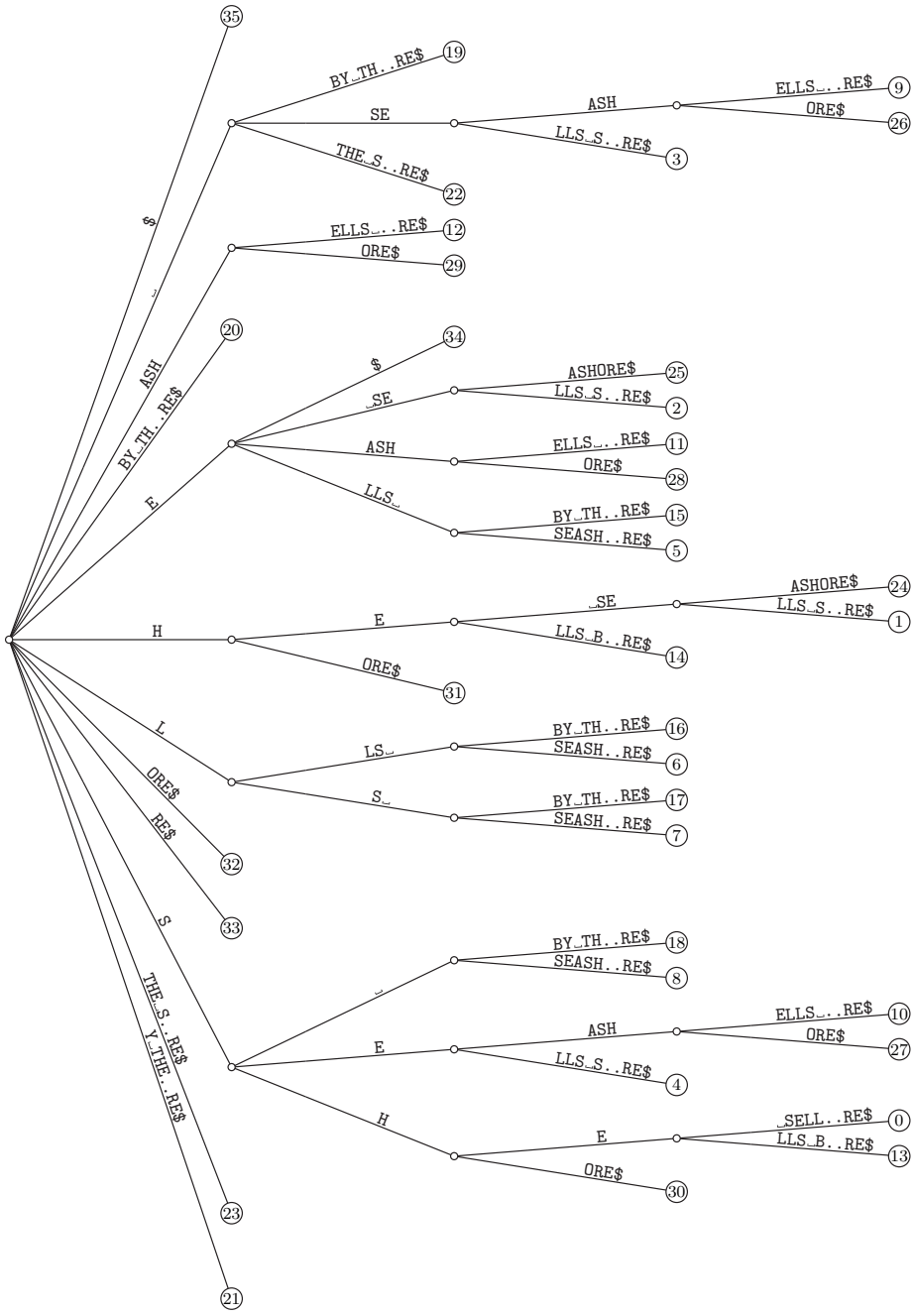
Obr. 16.2: Sufixový strom pre text „SHE␣SELLS␣SEASHELLS␣BY␣THE␣SEASHORE$“.



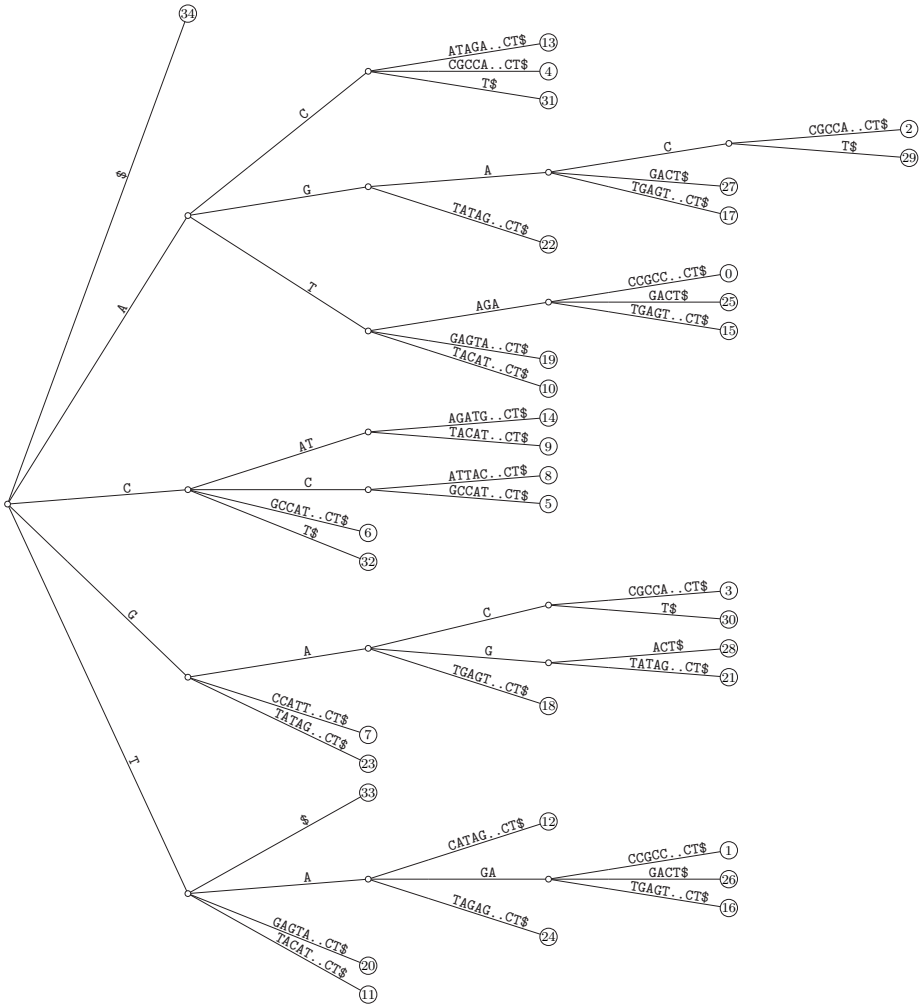
Obr. 16.3: Sufixový strom pre text „ATAGACCGCCATTACATAGATGAGTATAGAGACT$“.



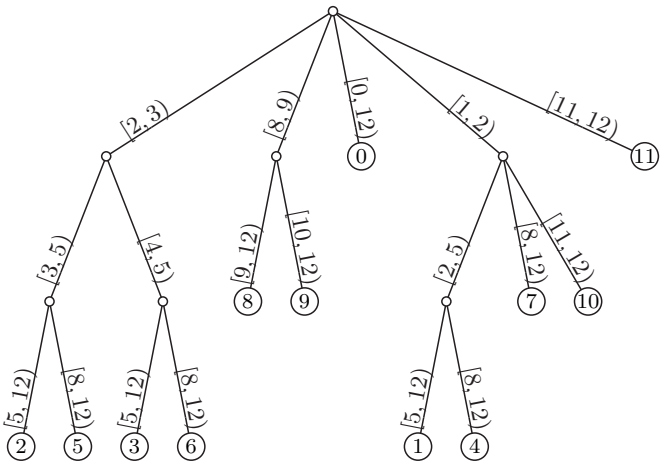

Obr. 16.4: Sufixový strom pre text „`MISSISSIPPI$`“. Cesty, ktoré sa nedelili, sme skontrahovali do jednej hrany. Keďže v strome je $n$ sufixov, teda $n$ listov, strom má iba lineárne veľa vrcholov aj hrán. Stále však máme problém, že písmená na jednotlivých hranách zaberajú spolu kvadratický priestor.



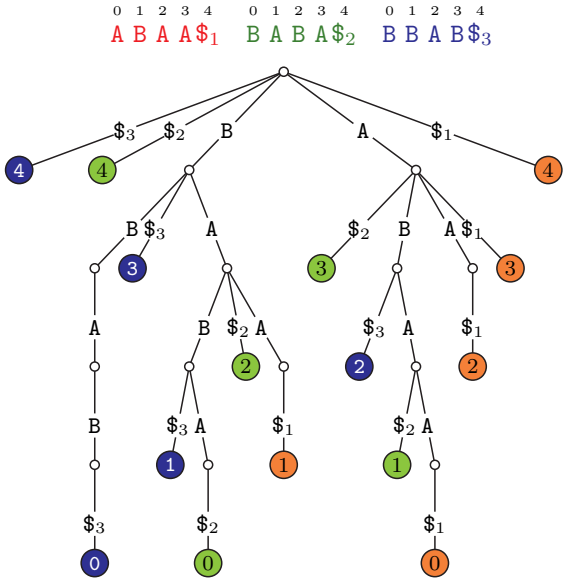

Obr. 16.5: Sufixový strom (s komprimovanými cestami) pre text „SHE␣SELLS␣SEASHELLS␣BY␣THE␣SEASHORE$“.




Obr. 16.6: Sufixový strom (s komprimovanými cestami) pre úsek DNA sekvencie „`ATAGACCGCCATTACATAGATGAGTATAGAGACT$`“.




Obr. 16.7: Sufixový strom pre „MISSISSIPPI$“. Skontrahujeme hrany a namiesto reťazcov na každej hrane použijeme indexy [odkiaľ, pokiaľ) sa nachádza v pôvodnom texte. Takto vieme každú hranu reprezentovať v konštantnej pamäti a tým pádom celý sufixový strom zaberá len lineárnu pamäť.




Obr. 16.8: Zovšeobecnený sufixový strom môže obsahovať viacero textov (dokumentov) naraz. Listy v tomto prípade predstavujú sufix alebo pozíciu v niektorom texte.

## 16.2 Aplikácie

### #1 Vyhľadávanie v texte

Majme text $T$ a vzorku $P$. Chceme zistiť, či sa $P$ nachádza v $T$, prípadne nájsť jej prvý výskyt alebo všetky výskyty.

Sufixový strom to umožňuje veľmi jednoducho:

- Stačí zísť z koreňa pozdĺž cesty určenej reťazcom $P$. Ak sa cesta skončí úspešne v nejakom vrchole, všetky listy v danom podstrome zodpovedajú výskytom $P$ v texte.
- Ak chceme nájsť *prvý* výskyt, predpočítame si pre každý vrchol ukazovateľ na list s najmenším číslom sufixu (alebo priamo pozíciu prvého výskytu). Urobíme to jedným prechodom stromu zdola nahor v čase $O(n)$: pri postorder prechode uložíme do každého vrcholu minimum z jeho detí.
- Ak chceme nájsť *počet* výskytov, stačí si predpočítať počet listov v každom podstrome, opäť jedným postorder prechodom.
- Ak chceme nájsť *všetky* výskyty, jednoducho prehľadáme celý podstrom. Ak je počet výskytov $k$, podstrom má veľkosť $O(k)$.

Napríklad, vyhľadajme reťazec `BA` vo všeobecnom sufixovom strome na obrázku 16.8. Začneme v koreni a postupne sledujeme hrany označené `B` a `A`. V podstrome, do ktorého sa takto dostaneme, sa nachádzajú štyri listy: červená 1, zelená 0 a 2 a modrá 1. Tieto listy presne zodpovedajú štyrom výskytom reťazca `BA` v textoch `ABAA`, `BA BA` a `BBAB`.

Výsledné zložitosti:

| | |
|---|---|
| *predpočítanie* | $O(n)$ |
| *prvý výskyt / počet výskytov* | $O(m)$ |
| *všetky výskyty* | $O(m+k)$ |

kde $k$ je počet výskytov vzorky v texte $T$.

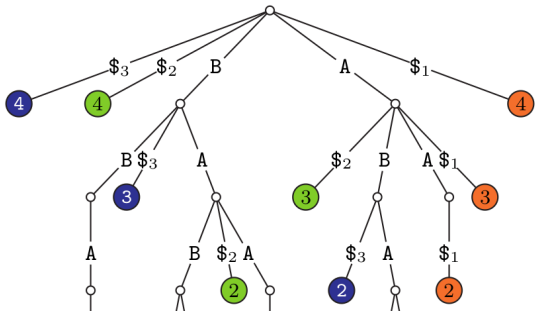
### #2 $n$-gramy

Pod $n$-gramom rozumieme postupnosť $n$ písmen. Napríklad existuje 8 rôznych trigramov pozostávajúcich iba z písmen `A` a `B`. Ktoré z nich sa vyskytujú v reťazcoch `ABAA`, `BABA` a `BBAB`?

Jeden spôsob, ako to zistiť, je prechádzať reťazce a priebežne si ukladať množinu všetkých trigramov, ktoré sme videli.

Alternatívne sú všetky trigramy priamo viditeľné v zovšeobecnenom sufixovom strome týchto reťazcov: stačí si predstaviť, že strom „rozrežeme“ v hĺbke 3. Potom môžeme odpoveď jednoducho odčítať zo všetkých ciest dĺžky 3 od koreňa:

ABA, BAA, BAB a BBA.




## #3 Najdlhší opakujúci sa podreťazec

Aký je najdlhší podreťazec v reťazci MISSISSIPPI, ktorý sa v ňom vyskytuje aspoň dvakrát?

Odpoveď: ISSI.

Ako to však zistíme vo všeobecnosti? Toto je problém, ktorý sme spomínali v úvode kapitoly.

Pred zavedením sufixových stromov bol dlho považovaný za ťažký: existuje triviálny $O(n^3)$ algoritmus a o niečo lepší $O(n^2)$ algoritmus založený na dynamickom programovaní. Dlhý čas nebolo jasné, či sa dá nájsť aj lineárny algoritmus a dokonca sa objavili hypotézy, že žiadny takýto neexistuje.

Prekvapivo však s pomocou sufixových stromov dostaneme riešenie, ktoré je úplne jednoduché:

> *najdlhší opakujúci sa podreťazec zodpovedá najhlbšiemu vnútornému vrcholu sufixového stromu.*

Vnútorný vrchol sufixového stromu vždy reprezentuje reťazec, ktorý sa v texte vyskytuje aspoň dvakrát: pod daným vrcholom sa totiž nachádzajú aspoň dva listy, teda dva sufixy začínajúce rovnakým prefixom. Počet listov v podstrome zároveň udáva počet výskytov tohto podreťazca.

Pre každý vrchol si môžeme predpočítať jeho *textovú hĺbku (string depth)*, t.j. *počet znakov* na ceste od koreňa do daného vrcholu. (Pozor, nejde o klasickú hĺbku vrcholu v zmysle počtu hrán, ale o dĺžku textu uloženého na hranách po ceste od koreňa.)

Najdlhší opakujúci sa podreťazec zodpovedá vnútornému vrcholu s maximálnou hodnotou *textovou hĺbkou*. Takýto vrchol vieme nájsť v čase $O(n)$ jedným prechodom stromu.

Skúste si rozmyslieť, ako by ste riešili mnohé ďalšie príbuzné úlohy ako:

- najdlhší opakujúci sa podreťazec, ktorého výskyte sa neprekrývajú,
- najkratší podreťazec, ktorý sa v texte vyskytuje len raz,
- najčastejšie sa vyskytujúci reťazec dĺžky aspoň $k$.

## #4 Najdlhší spoločný podreťazec

Podobná úloha ako predošlá, ale tentoraz majme dva texty $T_1$ a $T_2$ a chceme nájsť ich najdlhší spoločný podreťazec.

Stačí zostrojiť zovšeobecnený sufixový strom pre množinu $\{T_1, T_2\}$. Listy teraz *ofarbíme dvoma farbami* podľa toho, ku ktorému textu príslušný sufix patrí. Cieľom je nájsť taký vnútorný vrchol, pod ktorým sa nachádzajú listy oboch farieb, a ktorý zároveň maximalizuje svoju *textovú hĺbku*.

Pri prechode stromom zdola nahor si pre každý vrchol predpočítame, či jeho podstrom obsahuje iba listy jednej farby (a ktorú), alebo listy oboch farieb. Vnútorné vrcholy, ktoré majú pod sebou listy oboch farieb, reprezentujú všetky spoločné podreťazce oboch textov. Najdlhší z nich je ten s najväčšou hodnotou textovou hĺbkou.

## #5 Maximálne *repeaty*

Majme text $T$ a chceme nájsť všetky *maximálne repeaty*, t.j. také dvojice výskytov podreťazca, že

$$T[i \ldots i+k] = T[j \ldots j+k],$$

ale podreťazec sa už nedá predĺžiť ani doľava, ani doprava:

$$T[i-1] \neq T[j-1] \qquad \text{a} \qquad T[i+k+1] \neq T[j+k+1].$$

Intuitívne hľadáme všetky opakujúce sa podreťazce, ktoré sú „maximálne“ v tom zmysle, že ich ďalšie rozšírenie by už viedlo k nezhode.

V sufixovom strome je riešenie opäť jednoduché. Každý opakujúci sa podreťazec zodpovedá vnútornému vrcholu. Aby bol repeat maximálny, stačí skontrolovať, či jeho výskyty nemožno predĺžiť o jeden znak doľava alebo doprava.

Na tento účel si pri konštrukcii stromu stačí pri každom liste, ktorý reprezentuje sufix začínajúci na pozícii $i$, poznačiť znak, ktorý sa nachádza bezprostredne *pred ním,* t.j. znak $T[i-1]$ (pre $i = 0$ môžeme použiť špeciálny symbol).

Potom pre každý vnútorný vrchol, pod ktorým sa nachádzajú najmenej dva takto označené listy, môžeme ľahko overiť, či sa daný reťazec dá (alebo nedá) predĺžiť doľava či doprava. Vnútorné vrcholy, ktoré zároveň spĺňajú tieto obmedzenia, zodpovedajú práve maximálnym repeatom v texte.

## LCA a RMQ

V nasledujúcich aplikáciách budeme potrebovať vedieť riešiť dve základné podúlohy: *LCA (Lowest Common Ancestor)* a *RMQ (Range Minimum Query).*

**LCA:** Majme daný zakorenený strom a dva jeho vrcholy $u$ a $v$. Úlohou je nájsť ich *najnižšieho spoločného predka*, t.j. taký vrchol, ktorý je predkom oboch a zároveň leží najhlbšie v strome. Ak si predstavíme cestu od koreňa k $u$ a $v$, tak $\mathrm{LCA}(u, v)$ je posledný vrchol na ich spoločnej ceste.

**RMQ:** Majme pole čísel $A[0 \ldots n-1]$. Dotaz $\mathrm{RMQ}(i,j)$ má vrátiť pozíciu najmenšej hodnoty v intervale $A[i \ldots j]$.

Ako tieto dve úlohy efektívne riešiť si ukážeme v nasledujúcej kapitole. Zatiaľ budeme predpokladať, že ak si vstupný strom/pole vhodne predspracujeme, dokážeme odpovedať na otázky o LCA a RMQ v konštantnom čase.

## #6 Najdlhší spoločný prefix

Majme dve pozície $i$ a $j$ v texte $T$ a chceme zistiť najdlhší spoločný prefix sufixov $T[i \ldots]$ a $T[j \ldots]$, t.j. dĺžku najdlhšej počiatočnej zhody:

$$\mathrm{LCP}(i,j) = \max\{\, k : T[i \ldots i+k-1] = T[j \ldots j+k-1] \,\}.$$

**Triviálne riešenie.** Jednoduchý spôsob je porovnávať znaky jeden po druhom, kým nenájdeme prvý nesúlad. Toto trvá $O(k)$, kde $k = \mathrm{LCP}(i,j)$.

**Riešenie pomocou LCA.** V sufixovom strome má každý sufix svoj list. Najdlhší spoločný prefix dvoch sufixov zodpovedá *textovej hĺbke* ich najnižšieho spoločného predka:

$$\mathrm{LCP}(i,j) = \text{string-depth}\big(\mathrm{LCA}(\mathrm{list}(i), \mathrm{list}(j))\big).$$

Ak teda vieme LCA počítať v čase $O(1)$, vieme aj $\mathrm{LCP}(i,j)$ určiť v čase $O(1)$.

## #7 Približné výskyty

Majme text $T$ dĺžky $n$, vzorku $P$ dĺžky $m$ a chceme nájsť všetky pozície, kde sa $P$ „približne“ nachádza v $T$, pričom tolerujeme, ak sa vzorka od výskytu líši najviac v $k$ znakoch.

**Triviálne riešenie.** Priložíme vzorku $P$ ku každej pozícii v texte a spočítame, v koľkých znakoch sa líši. Toto trvá $O(n \times m)$, pretože pre každú pozíciu porovnávame po znakoch až kým nedosiahneme $m$.

**Rýchlejšie riešenie pomocou sufixového stromu.** Môžeme dosiahnuť čas $O(n \times k)$, ak využijeme sufixový strom a najmä možnosť počítať najdlhší spoločný prefix dvoch sufixov v čase $O(1)$.

Myšlienka je jednoduchá: skúsime priložiť vzorku $P$ ku každej pozícii $i$ v texte ako predtým, ale namiesto porovnávania znakov jeden po druhom sa vždy v konštantnom čase posunieme na *najbližšiu chybu*. To urobíme pomocou výpočtu

$$\mathrm{LCP}(P[j \ldots], T[i+j \ldots]).$$

Ak je napríklad priloženie presné na prvých $x$ znakov, LCP nám v $O(1)$ povie hodnotu $x$, a my okamžite preskočíme o $x$ znakov ďalej, až k najbližšej možnej chybe.

Opakujeme to najviac $(k+1)$-krát, keďže pri viac než $k$ chybách môžeme pozíciu rovno zamietnuť. Na každej pozícii teda strávime $O(k)$ času a celkovo tak dostávame časovú zložitosť $O(n \times k)$.

Poznámka: Tento algoritmus je skôr teoretický.

## #8 Počítanie dokumentov

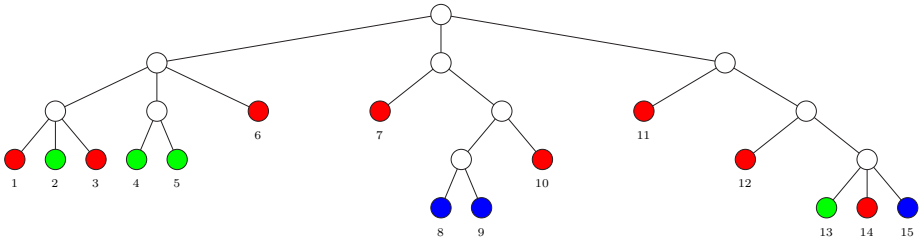
Majme množinu dokumentov

$$\mathcal{D} = \{T_1, \ldots, T_d\}.$$

Označme $n_i$ dĺžku $i$-teho textu a $N = \sum_i n_i$ celkovú dĺžku všetkých textov dokopy. Pre danú vzorku $P$ chceme spočítať v koľkých rôznych dokumentoch sa vyskytuje.

Pri ôsmej aplikácií snáď neprekvapí, že budeme pracovať so zovšeobecneným sufixovým stromom pre danú množinu dokumentov. Každý list zafarbíme farbou dokumentu, z ktorého pochádza príslušný sufix.

Ak úlohu preformulujeme do reči stromov: Máme $d$ rôznych farieb a pre každý vrchol v strome chceme určiť, *koľko rôznych farieb* (teda dokumentov) sa nachádza v jeho podstrome.




**Priamočiare riešenie.** Pre vrchol, v ktorom končí vyhľadávaná vzorka, stačí prehľadať celý jeho podstrom a zozbierať farby všetkých listov. To trvá

$$O(m + \#\text{počet listov v podstrome}) = O(m + \#\text{výskytov}).$$

Dá sa to lepšie? V dokumentoch sa hľadaná vzorka môže vyskytovať veľakrát a teda počet výskytov môže byť oveľa–oveľa väčší ako je počet rôznych dokumentov, ktoré ju obsahujú.

**Prvý pokus o zrýchlenie.** Pre každý vrchol môžeme predpočítať množinu farieb v jeho podstrome. To by však znamenalo pamäť aj čas na predpočítanie

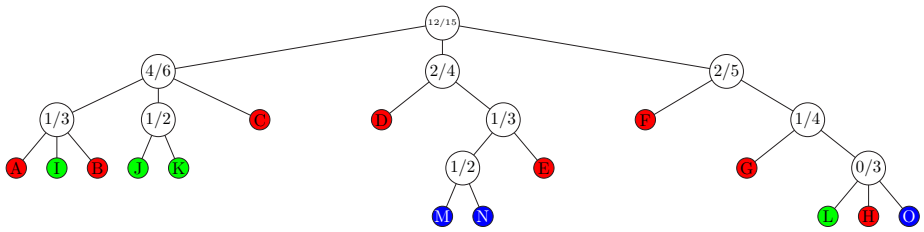
$$O(N \times d),$$

čo je dosť veľa.

Ak je $d$ malé, praktické riešenie je reprezentovať množiny ako bitvektory: jednotkový bit na pozícii $j$ znamená, že podstrom obsahuje $j$-ty dokument. Pri zjednocovaní množín detí potom stačí z-OR-ovať jednotlivé bitvektory.

**Lepšie riešenie pomocou LCA.** Finta je v tom, že sa vyhneme explicitnému ukladaniu množín. Fixujme si nejaký konkrétny podstrom (vrchol $v$). Dva listy $a$ a $b$ patria do podstromu práve vtedy, keď ich najnižší spoločný predok $\mathrm{LCA}(a, b)$ leží v podstrome $v$.

To nám umožní počítať opakovania farieb nasledovne: ak sa v podstrome nachádza napríklad $r$ listov červenej farby, tak existuje presne $r - 1$ po sebe idúcich dvojíc červených listov, ktorých LCA leží v tomto podstrome.




Namiesto toho, aby sme zisťovali počet rôznych farieb, spočítame najprv *počet opakovaní* farieb. Vo vrchole s podstromom obsahujúcim $r$ listov rovnakej farby má táto farba $r-1$ "extra výskytov". Ak vieme pre všetky farby spočítať, kde sa takéto opakovania nachádzajú, vieme pre daný vrchol určiť:

$$\#\text{farieb v podstrome } v \;=\; (\#\text{listov v podstrome } v) - (\#\text{opakovaní v podstrome } v).$$

Ako zistíme opakovania?

1. Pre každú farbu si vezmime všetky listy tejto farby a zotrieďme ich podľa poradia v Eulerovskom prechode (zľava doprava).
2. Pre každú dvojicu po sebe idúcich listov $a, b$ danej farby spočítame $\mathrm{LCA}(a, b)$.
3. Pre vrchol $u = \mathrm{LCA}(a, b)$ zaznačíme, že v jeho podstrome začína jedno opakovanie.
4. Nakoniec urobíme jeden DFS prechod zdola nahor. V každom vrchole sčítame všetky hodnoty svojich detí a tak zistíme celkový počet opakovaní v jeho podstrome.

Potom už len odpočítame počet opakovaní od počtu listov a získame počet rôznych farieb. Celú štruktúru vieme predpočítať v čase $O(n)$, ak máme k dispozícii LCA v čase $O(1)$. Po tomto predspracovaní vieme pre každý dotaz zodpovedať v čase $O(1)$.

## #9 Hľadanie dokumentov

Uvažujme opäť zovšeobecnený sufixový strom nad množinou dokumentov $\mathcal{D} = \{T_1, \ldots, T_d\}$. Tentokrát budeme chcieť nielen *počet* dokumentov, ktoré obsahujú

vzorku $P$, ale aj vypísať, ktoré to sú. Tak ako pri predchádzajúcej úlohe, si môžeme vrchol zodpovedajúci sufixu z dokumentu $T_i$ označiť farbou $i$ a úlohou bude vypísať všetky farby v danom podstrome.

Tak ako v predchádzajúcej aplikácii môžeme bez predpočítania jednoducho prejsť celý podstrom a zozbierať farby listov. Časová zložitosť tohto riešenia je však

$$O(m + \#\text{vrcholov v podstrome}).$$

My by sme ideálne chceli algoritmus, ktorý má zložitosť

$$O(m + \#\text{rôznych farieb v podstrome}).$$

**Kľúčová myšlienka.** Zavedieme si pole $A[1..n]$, kde $A[i]$ je index *predošlého listu s rovnakou farbou* ako list $i$. Ak je list $i$ prvý zo svojej farby, nech $A[i] = 0$.

Pozri príklad na obr. 16.9.

Listy zodpovedajúce výskytom vzorky tvoria *súvislý interval* $[i, j]$ v tomto poli. Chceme vypísať všetky *rôzne farby* v tomto intervale.

Farba na pozícii $k$ sa v intervale $[i, j]$ vyskytuje prvýkrát práve vtedy, keď

$$A[k] < i.$$

Takže problém sa redukuje na nasledovný:

$$\text{Vypísať všetky } k \in [i, j] \text{ také, že } A[k] < i.$$

Tieto pozície vždy prvým výskytom vzorky $P$ v každom dokumente. Ako ich nájsť efektívne?

Nad poľom $A$ si predpočítame dátovú štruktúru pre *range minimum query* (RMQ) v čase $O(n)$, s dotazmi v čase $O(1)$.

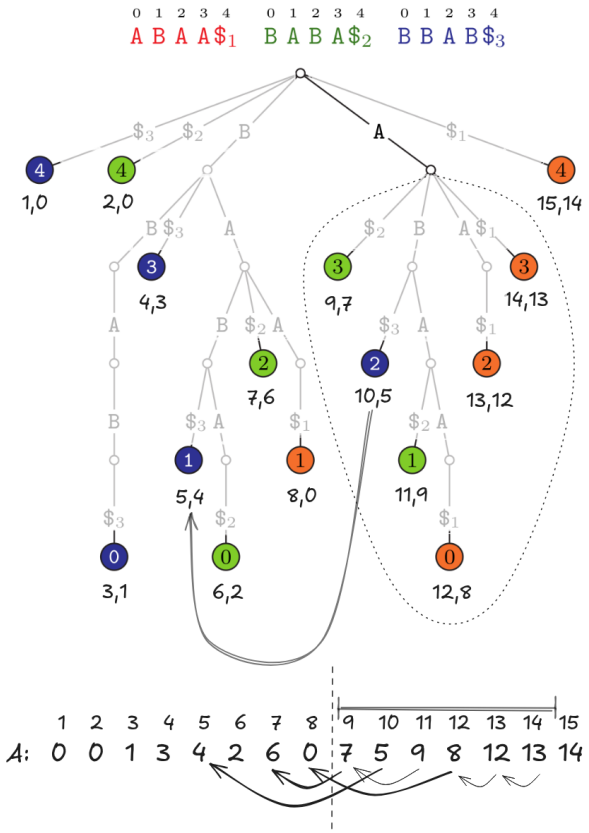
Potom postupujeme rekurzívne:

1. Nájdeme
$$k = \text{argmin}(A[i..j]).$$
2. Ak $A[k] < i$, tak pozícia $k$ reprezentuje nejaký dokument – vypíšeme farbu listu $k$.
3. Rekurzívne pokračujeme v intervaloch $[i, k-1]$ a $[k+1, j]$.

Rekurzia sa zastaví vždy, keď minimum v intervale už nespĺňa podmienku $A[k] < i$.

**Výsledná zložitosť.** Predspracovanie poľa $A$ (výpočet $A$ a RMQ) trvá $O(n)$.

Ak si predstavíme strom rekurzívnych volaní (koreň je minimum intervalu, synovia sú minimá ľavého a pravého podintervalu), prechádzame vlastne časť tzv. Kartézskeho stromu pre dané pole. Každý vnútorný vrchol je jedna farba, ktorú vypíšeme a listy reprezentujú časti poľa, kde rekurziu môžeme ukončiť, pretože všetky prvky sú už väčšie alebo rovné $i$. Keďže listov je len o 1 viac ako vnútorných vrcholov a každý vrchol zodpovedá len jednému volaniu RMQ, čo je $O(1)$, celkový čas je $O(\#\text{dokumentov})$.




Obr. 16.9: Pre každý list si poznačíme dvojicu $(i, p)$, kde $i$ je poradové číslo listu zľava doprava a $p$ je číslo *predošlého listu s rovnakou farbou*. Napríklad pod modrým vrcholom 2 máme poznačené $(10, 5)$, pretože je to 10-ty vrchol v poradí a najbližší modrý list vľavo je 5-ty v poradí (ako znázorňuje šípka medzi modrými listami $2 \to 1$). Tieto dvojice tvoria pole $A$ dolu.

Povedzme, že v strome hľadáme vzorku `A` – prejdeme od koreňa po hrane označenej `A` a zakrúžkovaný podstrom zodpovedá všetkým výskytom `A` vo všetkých dokumentoch. Tento podstrom zodpovedá intervalu od 9-teho po 14-ty vrchol.

Všimnite si, že prvý výskyt v každom dokumente sú práve tie listy, ktorých predchodca je *skôr ako* 9-ty v poradí. Naopak, opakované výskyty majú predchodcu v intervale $[9, 14]$.

Úloha sa nám teda redukuje na nájdenie tých pozícii $k$ v intervale $[9, 14]$, kde $A[k] < 9$.

## Zhrnutie

Sufixový strom je písmenkový strom (trie), ktorý obsahuje všetky sufixy daného reťazca. Zovšeobecnený sufixový strom obsahuje všetky sufixy viacerých reťazcov. Zaberá iba $O(n)$ pamäte, ak použijeme kompaktnú reprezentáciu, pričom cesty, ktoré sa nedelia, skomprimujeme do jednej hrany a každú hranu reprezentujeme len dvoma indexmi do pôvodného textu. Možno ho zostrojiť v čase $O(n)$ (konštrukciu si ukážeme v kapitole o sufixových poliach).

Je veľmi užitočný v stringológii, pretože odhaľuje veľa štruktúry o podreťazcoch daného textu a mnoho problémov nad reťazcami sa dá pomocou neho preložiť na problémy nad stromami. Preto sa oplatí mať poruke malý „prekladový slovník“ medzi svetom reťazcov a svetom stromov:

| *Stringológia* | | *Stromy* |
|---:|:---:|---|
| pozícia $i$ v texte | $\longleftrightarrow$ | list $i$ v sufixovom strome |
| $i$-ty sufix | $\longleftrightarrow$ | cesta od koreňa k listu $i$ |
| podreťazec $T[i..j]$ | $\longleftrightarrow$ | začiatok cesty od koreňa k listu $i$ dĺžky $j-i+1$; |
| výskyty vzorky $P$ | $\longleftrightarrow$ | listy v podstrome pod cestou označenou $P$ |
| všetky podreťazce dĺžky $k$ v texte | $\longleftrightarrow$ | „prerezanie“ stromu v hĺbke $k$ |
| dokument | $\longleftrightarrow$ | farba listu |
| dokumenty obsahujúce $P$ | $\longleftrightarrow$ | rôzne farby listov pod cestou $P$ |
| spoločný prefix dvoch podreťazcov | $\longleftrightarrow$ | spoločná cesta od koreňa smerom k dvom vrcholom |
| najdlhší spoločný prefix dvoch sufixov | $\longleftrightarrow$ | LCA príslušných dvoch listov. |

# Kapitola 17

# LCA a RMQ

V tejto kapitole sa budeme venovať dvom na prvý pohľad úplne odlišným a nesúvisiacim algoritmickým problémom, ktoré sa nám zišli pri riešení úloh na sufixových stromoch.

Prvý z nich je LCA – Lowest Common Ancestor, teda najnižší spoločný predok. Úloha znie nasledovne: Máme dopredu daný koreňový strom, v ktorom sa vrcholy nemenia. Máme tiež čas na jeho predspracovanie, teda môžeme si pripraviť dátové štruktúry a pomocné informácie. Následne budeme opakovane dostávať dotazy, v ktorých dostaneme vždy dvojicu vrcholov a našou úlohou bude čo najrýchlejšie nájsť ich najnižšieho spoločného predka. Ako napovedá už názov, najnižší spoločný predok je taký vrchol v strome, ktorý je predkom oboch vrcholov a zároveň sa nachádza najhlbšie (teda najbližšie k týmto vrcholom).

Druhý problém, ktorému sa budeme venovať, je RMQ – Range Minimum Query, teda dotazy na minimum v rozsahu. V tomto prípade máme dané pole čísel $A[0 \ldots n-1]$ a cieľom je opakovane zodpovedať dotazy typu: „Kde sa nachádza najmenšia hodnota v poli medzi indexmi $i$ a $j$?“ Inými slovami: „Nájdi, kde je minimum v úseku $A[i \ldots j]$.“ Rovnako ako pri LCA, aj tu predpokladáme, že pole sa nemení (je statické), a že máme čas na predspracovanie údajov. Po ňom nasledujú dotazy, ktoré chceme zodpovedať čo najrýchlejšie.

Ako by ste takéto úlohy riešili?

## 17.1 Jednoduché riešenia

Postupne sa pozrime na niekoľko základných, čoraz lepších riešení, pričom budeme porovnávať čas zodpovedania dotazu, pamäťovú náročnosť a čas predspracovania.

Začnime problémom RMQ.

**RMQ #1. Žiadne predspracovanie:** Najjednoduchší prístup je nerobiť nič navyše. Pre každý dotaz RMQ$(i, j)$ jednoducho prejdeme celé pole medzi $i$ a $j$ a nájdeme minimum.

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(n)$ | $O(n)$ | žiadne |

**RMQ #2. Predpočítame všetko:** Ďalšia možnosť je predpočítať si minimum pre *každú možnú* dvojicu indexov $(i, j)$ a uložiť si ho do tabuľky. Potom môžeme každý dotaz na RMQ zodpovedať v konštantnom čase jednoduchým pohľadom do tabuľky.

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(1)$ | $O(n^2)$ | $O(n^2)$ |

**RMQ #3. Intervalový strom:** Nad poľom si zostrojíme binárny strom (každý vrchol bude zodpovedať nejakému intervalu pod ním). Pre každý vrchol spočítame minimum z jeho dvoch synov (ľavý a pravý syn zodpovedá prvej a druhej polovici intervalu), tým pádom bude mať každý vrchol predpočítané minimum z intervalu pod ním. Keď budeme chcieť nájsť minimum z intervalu $(i, j)$, odpoveď poskladáme z najviac $O(\log n)$ podstromov, ktoré dokopy pokrývajú celý interval.

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(\log n)$ | $O(n)$ | $O(n)$ |

Dá sa to lepšie?

Oddýchnime si a pozrime sa na LCA. Prvé dva nápady pre RMQ fungujú aj pre LCA:

**LCA #1. Žiadne predspracovanie:** Pre každý dotaz LCA$(u, v)$ jednoducho prejdeme a zapíšeme si cestu od $u$ aj od $v$ ku koreňu. Potom tieto cesty prejdeme v opačnom smere od koreňa k $u/v$; obe cesty najskôr začínajú spoločne no a my potrebujeme nájsť prvé miesto, kde sa odpoja.

Druhá možnosť je, že si predpočítame hĺbku každého vrcholu. Pri otázke na LCA$(u, v)$ najskôr dorovnáme hĺbky (toho, kto je hlbšie, posunieme po rodičoch vyššie, kým sa nedostane na rovnakú úroveň) a následne oboch postupne *naraz* posúvame smerom nahor, kým sa nestretnú (prvý spoločný vrchol je práve LCA).

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(h)$, kde $h$ je výška stromu, najviac $O(n)$ | $O(n)$ | žiadne, resp. $O(n)$ |

**LCA #2. Predpočítame všetko:** Pre každú dvojicu vrcholov $u$ a $v$ si spočítame výsledok a uložíme do tabuľky. Každý dotaz potom vieme vyriešiť jedným pohľadom do tabuľky. (Skúste si rozmyslieť, ako túto tabuľku spočítať efektívne.)

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(1)$ | $O(n^2)$ | $O(n^2)$ |

OK. Vedeli by sme nájsť riešenie, ktoré má čas lepší ako lineárny a pamäť lepšiu ako kvadratickú?

Hint: Binárne vyhľadávanie.

Hint #2: Predstavme si, že naším prvým krokom bude vyrovnať hĺbky oboch vrcholov. To znamená, že hlbšie položený vrchol „vystúpa“ po strome nahor, kým sa nedostane na rovnakú úroveň ako ten druhý. Ako sa to dá lepšie ako v lineárnom čase a zároveň lepšie ako s kvadratickou pamäťou?

**LCA #3 Binárny rebrík.** Myšlienka je nasledovná: Predpokladajme, že vrcholy už máme dorovnané na rovnakej úrovni a predstavme si, že si pod seba napíšeme obe cesty od vrcholov smerom ku koreňu – napíšeme si zoznam predkov, krok za krokom, až po koreň. Ako nájdeme miesto, kde sa tieto dve cesty prvýkrát „zbehnú“?

Jednoducho, pomocou binárneho vyhľadávania: Pozrieme sa do stredu týchto ciest.

- Ak sú v tomto bode predkovia rovnakí, vieme, že LCA je v tejto výške alebo ešte nižšie (bližšie k vrcholom) – LCA treba hľadať v prvej polovici.
- Ak sú predkovia rozdielni, znamená to, že k zhodnému predkovi sme sa ešte nedostali, a LCA musí byť vyššie v strome – LCA trbea hľadať v druhej polovici.

Takýmto spôsobom dokážeme v logaritmickom počte krokov nájsť najnižší bod, kde sa obe cesty spájajú.

Ostáva ešte vyriešiť dve otázky: Ako dorovnať výšky vrcholov, aby sme mohli hľadať LCA v synchronizovaných úrovniach? A ako sa pri binárnom vyhľadávaní pozrieť do stredu? Postupovať zakaždým krok po kroku by bolo pomalé. Potrebujeme mať možnosť „skákať“ rýchlejšie – nie po jednom, ale po väčších skokoch.

Riešením je predpočítať si pre každý vrchol skoky o 1 predka, o 2 predkov, o 4, o 8, o 16, atď., skoky dĺžky $2^k$ pre každú mocninu dvojky. Pre každý vrchol $v$ a každé $k$ od 1 po $\lg n$ si uložíme

$$\mathrm{up}[v][k] = \text{predok vo vzdialenosti } 2^k \text{ od } v.$$

(Táto technika sa nazýva *binary lifting*.)

Dorovnanie hĺbok potom dokážeme v logaritmickom čase: Napríklad ak jeden vrchol je v hĺbke 47 a druhý v hĺbke 68, potrebujeme druhým skočiť o 21 vyššie, čo zvládneme jednoducho troma skokmi o $16 + 4 + 1$.

Namiesto klasického binárneho vyhľadávania, ktoré sa pozerá vždy do stredu použijeme variant, ktorý sa zakaždým pozrie na najbližšiu menšiu mocninu dvojky. Takto sa celý dotaz LCA dá vyriešiť v čase $O(\log n)$, pričom predspracovanie trvá $O(n \log n)$ a zaberá rovnaké množstvo pamäte.

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(\log n)$ | $O(n \log n)$ | $O(n \log n)$ |

Hmmm... síce sme dorovnali tretie riešenie RMQ (logaritmický čas), ale pamäť a predspracovanie je horšie! Dá sa to lepšie?

## 17.2 RMQ – ešte lepšie riešenie

Späť ku RMQ. V predchádzajúcej časti sme si ukázali viacero riešení – od naivného lineárneho po logaritmické s intervalovým stromom. Zaujímavé však

je, že všetky doterajšie riešenia sú oveľa všeobecnejšie, než by sa na prvý pohľad mohlo zdať. *Nevyužili sme pritom žiadnu špeciálnu vlastnosť operácie minimum!* Každý z uvedených prístupov by úplne rovnako fungoval aj pre iné operácie, napríklad:

- výpočet súčtu alebo súčinu prvkov v intervale,
- bitové operácie ako AND, OR, XOR,
- dokonca aj pre zložitejšie štruktúry ako:
  - matice, kde v intervale počítame súčin matíc (napr. na ľavo asociovaný),
  - alebo funkcie $X \to X$ reprezentované tabuľkou, pričom chceme poskladať funkcie v intervale $((f \circ g)(x) = g(f(x)))$.

Naše doterajšie riešenia fungujú pre ľubovoľnú asociatívnu operáciu, pre fanúšikov algebry: pre ľubovoľný monoid. Nevyužívajú žiadnu špeciálnu vlastnosť minima.

Môžeme si teda položiť otázku: *Čo ak nás zaujíma konkrétne práve minimum?* Existujú efektívnejšie riešenia ktoré naplno využívajú vlastnosti operácie min? A čím je vlastne minimum špeciálne?

Nuž, operácia min má hneď niekoľko príjemných vlastností. Je asociatívna (nezáleží na zátvorkách), komutatívna (nezáleží na poradí) a okrem toho má ešte jednu tajnú zbraň:

$$\min(x, x) = x \qquad \text{pre každé } x.$$

V preklade: ak aj nejaký prvok započítame viackrát, výsledné minimum sa nezmení! Ak by ste chceli machrovať v krčme (alebo na skúške), tak táto vlastnosť sa volá *idempotencia*.

V treťom riešení, kde sme použili intervalový strom, sme celý interval rozložili na niekoľko disjunktných podintervalov (vždy s dĺžkami, ktoré sú mocninami dvojky). Pre tieto podintervaly sme mali už vypočítanú odpoveď a stačilo jednotlivé medzivýsledky skombinovať. V najhoršom prípade však bolo potrebných až $O(\log n)$ podintervalov – práve preto, že boli disjunktné a museli pokrývať celý rozsah presne.

Pri operácii minimum máme výhodu: môžeme použiť aj intervaly, ktoré sa prekrývajú (výsledok to nezmení).

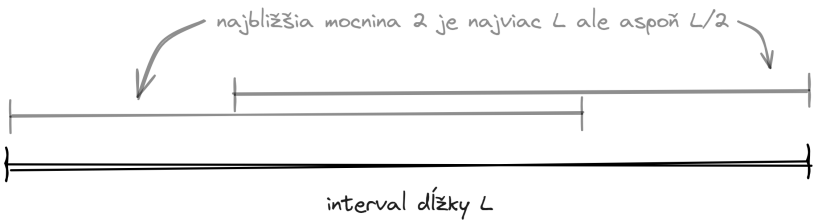
**RMQ #4 Riedka tabuľka.** Myšlienka riešenia je prekvapivo jednoduchá: predpočítame si minimá pre všetky intervaly, ktorých dĺžka je mocninou dvojky. Konkrétne pre každú začiatočnú pozíciu $i \in [0 \ldots n)$ a pre každé $k$ od 0 po $\lg n$ si spočítame

$$M[k][i] = \min A[i \ldots i + 2^k - 1].$$

Finta spočíva v tom, že *každý* interval $[i, j]$ (dĺžky $\ell = j - i + 1$) vieme pokryť *len dvoma* intervalmi dĺžky $2^k$. Ako to funguje? Najskôr si nájdeme najväčšiu mocninu dvojky, ktorá sa ešte do nášho intervalu zmestí:

$$2^k \le \ell \qquad \text{teda} \qquad k = \lfloor \lg \ell \rfloor.$$

Hodnota $2^k$ je určite aspoň polovica dĺžky intervalu, $\ell/2 \leq k \leq \ell$, takže celý interval pokryjeme jedným intervalom, ktorý začína na $i$ ($A[i \ldots i + 2^k - 1]$) a jedným, ktorý končí na $j$ ($A[j - 2^k + 1 \ldots j]$).




Výsledok spočítame ako

$$\mathrm{RMQ}(i, j) = \min(M[k][i], M[k][j - 2^k + 1]).$$

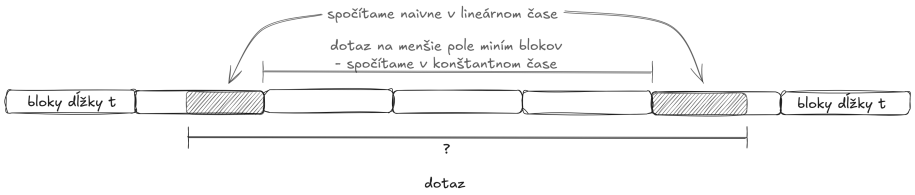
Poznámka k implementácii: hodnotu $k$ prosím vás nepočítajte pomocou matematickej funkcie logaritmus. Počítače prirodzene pracujú v dvojkovej sústave a vďaka tomu môžeme získať $k$ oveľa efektívnejšie ako pozíciu najvyššieho jednotkového bitu. Moderné procesory majú na tento účel špeciálnu inštrukciu s názvom CLZ (Count Leading Zeros), ktorá vráti počet núl na začiatku binárneho zápisu čísla. Hodnotu $k = \lfloor \lg \ell \rfloor$ dostaneme ako $k = \text{dĺžka registra} - 1 - \mathrm{CLZ}(\ell)$.

| Čas na dotaz | Pamäť | Predspracovanie |
|---|---|---|
| $O(1)$ | $O(n \log n)$ | $O(n \log n)$ |

Super – podarilo sa nám vyriešiť problém RMQ v konštantnom čase a zaplatili sme za to len „trochu horšou“ pamäťovou zložitosťou $O(n \log n)$. Dobrá správa: Vyhrali sme, zjavne lepší ako konštantný čas už dosiahnuť nevieme. Zlá správa: Lenže teoreticky by sa stále mohla dať zlepšiť pamäť. A tak ostáva v pozadí trýznivá otázka: Dá sa to lepšie?

**RMQ #5. Delenie na bloky.** Tu je jeden praktický nápad, ak nám záleží na úspornejšom využití pamäte a zároveň nepotrebujeme úplne najrýchlejšie dotazy. Môžeme urobiť rozumný kompromis a získať lepšiu pamäť za cenu trochu horšieho času: Pole veľkosti $n$ si rozdelíme na bloky veľkosti $t$ (povedzme $t = 10$). Dostaneme tak $n/t$ blokov. Z každého bloku spočítame minimum a zostrojíme nové, $t$-krát menšie pole týchto miním. Na toto menšie pole aplikujeme predchádzajúce riešenie – budeme tak vedieť v konštantnom čase odpovedať na RMQ dotazy medzi *celými blokmi*.

A čo s intervalmi, ktoré zasahujú čiastnočne do vnútra blokov? V konštantnom čase nájdeme minimum menšieho intervalu zarovnaného na bloky, a zvyšné kúsky na začiatku a na konci intervalu (najviac $2t$ prvkov) jednoducho prejdeme naivne v lineárnom čase.




Celkový čas je $O(t)$, na druhej strane pamäťová náročnosť je $t$-krát menšia. Jeden extrém je voľba $t = 1$, čo je v podstate to isté ako riešenie s riedkou tabuľkou s časom $O(1)$ a pamäťou $O(n \log n)$). Opačný extrém sú bloky dĺžky $\log n$ – čas narastie na $O(\log n)$, ale pamäť klesne na lineárnu. Dostávame tak alternatívne, úplne nové riešenie, ktoré má rovnakú zložitosť ako intervalové stromy. Avšak môžeme si zvoliť aj ľubovoľné $t$ „medzi tým“, podľa toho, či chceme lepší čas alebo lepšiu pamäť. Dostávame tak celé spektrum riešení a tzv. trade-off (niečo za niečo) medzi časom a pamäťou. Napríklad pre $t = \sqrt{\log n}$ bude čas $O(\sqrt{\log n})$ a pamäť $O(n\sqrt{\log n})$.

**Rekapitulácia.** Skúsme si zrekapitulovať, kde sme a čo už vieme. Riešenia, ktoré sme zatiaľ vymysleli sú v tabuľke nižšie:

| **Riešenia RMQ** | **Čas na dotaz** | **Pamäť** | **Predspracovanie** |
|---|---|---|---|
| #1 bez predspracovania | $O(n)$ | $O(n)$ | žiadne |
| #2 všetko predpočítané | $O(1)$ | $O(n^2)$ | $O(n^2)$ |
| #3 intervalový strom | $O(\log n)$ | $O(n)$ | $O(n)$ |
| #4 riedka tabuľka | $O(1)$ | $O(n \log n)$ | $O(n \log n)$ |
| #5 delenie na bloky | $O(t)$ | $O(n \log n/t)$ | $O(n \log n/t)$ |
| napr. pre $t = \sqrt{\log n}$ | $O(\sqrt{\log n})$ | $O(n\sqrt{\log n})$ | $O(n\sqrt{\log n})$ |

| **Riešenia LCA** | **Čas na dotaz** | **Pamäť** | **Predspracovanie** |
|---|---|---|---|
| #1 bez predspracovania | $O(n)$ | $O(n)$ | žiadne |
| #2 všetko predpočítané | $O(1)$ | $O(n^2)$ | $O(n^2)$ |
| #3 binárny rebrík | $O(\log n)$ | $O(n)$ | $O(n)$ |

## 17.3 Vzťah LCA a RMQ

Z doterajšieho vývoja by sa mohlo zdať, že problém RMQ je jednoduchší ako LCA. Veď pri RMQ pracujeme len s obyčajným poľom čísel, zatiaľ čo pri LCA riešime dotazy na predkov v stromovej štruktúre, čo už na prvý pohľad vyzerá zložitejšie. A naozaj, pre RMQ sme našli oveľa efektívnejšie algoritmy: $O(1)$ čas, $O(n \log n)$ pamäť alebo $O(\log n)$ čas, $O(n)$ pamäť, zatiaľčo naše doteraz najlepšie riešenie LCA s binárnym rebríkom má $O(\log n)$ čas a $O(n \log n)$ pamäť a celé pôsobí o čosi „komplikovanejšie“ ako riešenia RMQ.

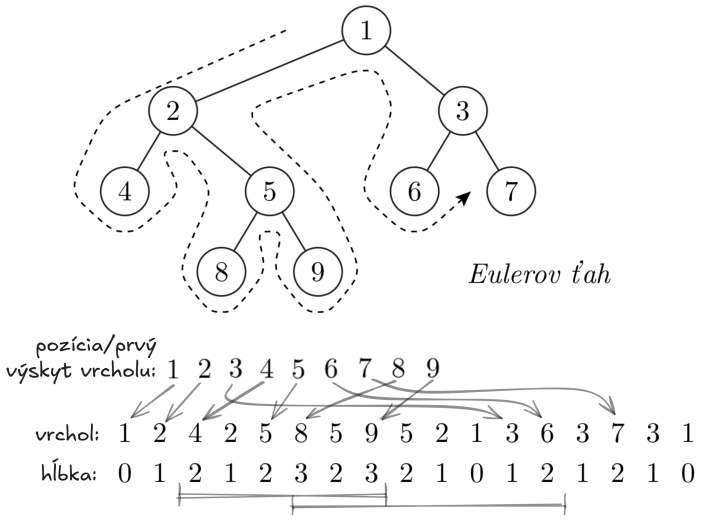
Navyše, netrpezlivým čitateľom už možno vŕta v hlave otázka: *„Prečo sa vôbec obom problémom venujeme v jednej kapitole? Veď spolu vôbec nesúvisia. . . “*

No tu prichádza prekvapenie: LCA a RMQ sú oveľa užšie späté, než sa na prvý pohľad zdá. Ukážeme si, že oba problémy sú algoritmicky ekvivalentné – teda rovnako „ťažké“ (alebo ak chcete: rovnako „ľahké“) a jeden vieme efektívne vyriešiť pomocou druhého.

## Redukcia LCA na RMQ

Predstavme si, že máme daný strom z úlohy LCA a našou úlohou je opakovane odpovedať na dotazy typu: „Nájdi najnižšieho spoločného predka vrcholov $u$ a $v$.“ Ukážeme si, že takúto úlohu vieme pretransformovať na problém RMQ. Cieľom tejto časti bude ukázať, ako zo stromu „vyrobiť“ pole, a ako dotazy typu LCA v tomto strome preložiť na dotazy RMQ, vďaka ktorým už potom nájdeme LCA jednoducho a efektívne. Predstavme si, že niekto za nás už problém RMQ vyriešil. My riešime LCA, ale v našom riešení môžeme pokojne použiť „knižnicu na RMQ“ ako čiernu krabičku.

Ako na to? Strom „rozvinieme do poľa“ pomocou tzv. Eulerovského prechodu okolo stromu (pozri príklad na obrázku) – ide o modifikované prehľadávanie do hĺbky, pričom pri každej návšteve vrcholu si zapíšeme jeho *hĺbku*. Okrem toho si spravíme dve tabuľky, ktoré nám umožnia prevádzať medzi vrcholmi a pozíciami v tomto poli: pre každú pozíciu v poli hĺbok si poznačíme *vrchol*, ktorému zodpovedá a naopak, pre každý vrchol si uložíme *pozíciu* v poli hĺbok, keď sme vrchol *prvýkrát navštívili*. Všimnite si, že vrcholy môžeme navštíviť viackrát, napríklad z koreňa 1 vôjdeme do 2, zídeme do 4, vrátime sa späť do 2, a cez 2 prejdeme ešte raz po tom, čo prejdeme cez podstrom 5 8 9.



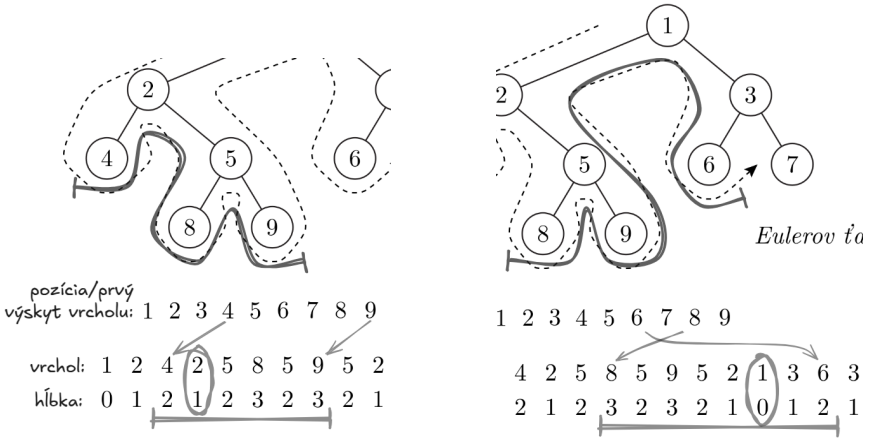

Pole hĺbok si predspracujeme na hľadanie RMQ. Ako teraz nájdeme LCA

dvoch vrcholov? Pre vrcholy $u, v$ platí:

$$\mathrm{LCA}(u,v) = \mathrm{vrchol}[\underset{\text{hĺbok}}{\mathrm{RMQ}}(\text{pozícia } u, \text{pozícia } v)]$$

alebo slovne: najskôr premeníme vrcholy na pozície v poli hĺbok. Potom spočítame RMQ na tomto intervale (pozor: potrebujeme pozíciu minima, nie len jeho hodnotu) a nakoniec túto pozíciu prevedieme naspäť na zodpovedajúci vrchol, čo je najnižší spoločný predok.

Ukážme si to na dvoch príkladoch: $\mathrm{LCA}(4,9)$ a $\mathrm{LCA}(8,6)$. V prvom príklade (pozri obr. vľavo) sa vrcholy 4 a 9 nachádzajú prvýkrát na pozíciach 2 a 7. Všimnite si, že tento interval zodpovedá presne časti Eulerovského ťahu medzi 4 a 9 a obsahuje hĺbky vrcholov, ktoré sme medzi tým navštívili. Z nich najmenšiu hĺbku 1 (index 3) má vrchol číslo 2, čo je naozaj $\mathrm{LCA}(4,9)$. V druhom príklade (pozri obr. vpravo) sa vrcholy 8 a 6 nachádzajú na pozíciach 5 a 12 a tento interval zodpovedá časti Eulerovského ťahu medzi 8 a 6. Najvyšší vrchol, ktorý po ceste navštívime, je koreň 1.




Skúste si rozmyslieť, že celú redukciu (v rámci predspracovania) zvládneme v lineárnom čase. (Vstup sa nám pritom trochu nafúkne, ale iba trochu – aká je dĺžka Eulerovského ťahu?)

Skvelé! Vďaka tejto redukcií už vieme aj LCA riešiť v konštantnom čase a $O(n \log n)$ pamäti! Stačí LCA tranformovať na RMQ a na RMQ použiť riešenie s riedkou tabuľkou a predpočítanými intervalmi dĺžky mocnín dvoch.

## Redukcia RMQ na LCA

Teraz sa môžeme pozrieť na opačný smer redukcie: ako previesť problém RMQ na problém LCA. Máme pole čísel a chceme odpovedať na dotazy typu RMQ (minimum v intervale). Ako to prevedieme na dotaz typu LCA v strome?

Použijeme tzv. kartézsky strom – je to binárny strom, ktorý:

- rešpektuje inorder poradie – ak prvky vypíšeme v inorder poradí (ľavý podstrom → koreň → pravý podstrom), dostaneme pôvodné pole

- a zároveň spĺňa vlastnosť min-haldy: každý vrchol má menšiu hodnotu ako jeho deti.

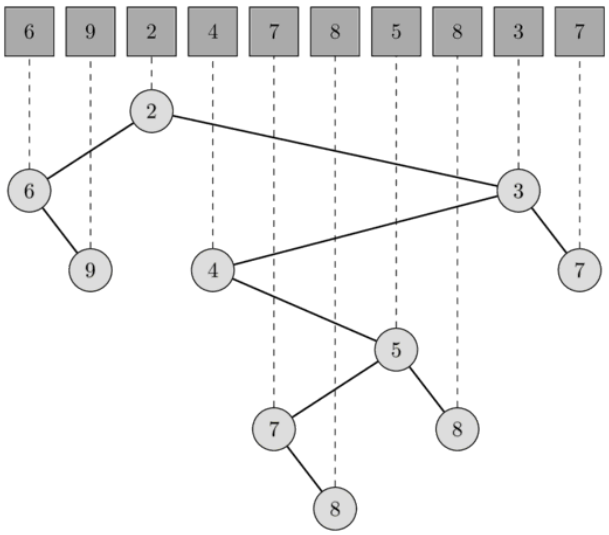
Ak sú všetky hodnoty v poli rôzne, potom je kartézsky strom nad týmto poľom jednoznačne určený:

- minimum poľa musí byť koreň,

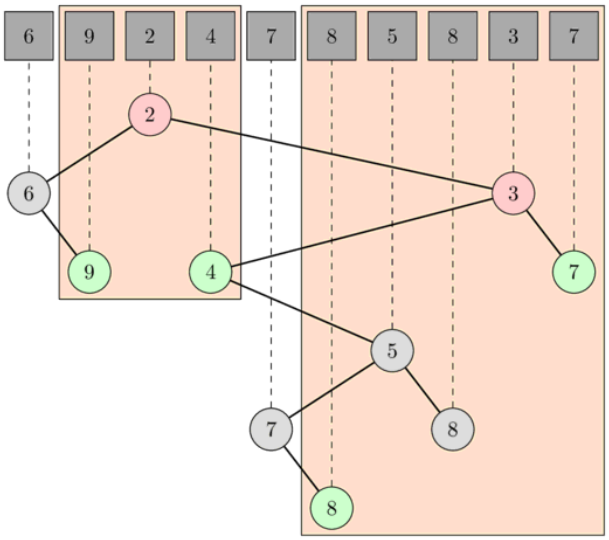
- ľavý podstrom sa skonštruuje rekurzívne z prvkov, ktoré ležia naľavo od minima a

- pravý podstrom sa skonštruuje z prvkov, napravo od minima.

V prípade, že sa v poli nachádzajú opakujúce sa hodnoty, strom už nie je jednoznačne určený, ale stále platí, že *niektorý* prvok s minimálnou hodnotou musí byť v koreni a zvyšok sa skonštruuje rekurzívne (pozri príklad na obrázku).




K tomuto stromu opäť pripravíme dve pomocné tabuľky, ktoré nám umožnia prevádzať medzi pozíciami v poli a vrcholmi v strome.




Ak teraz príde otázka na minimum z rozsahu $[\ell, r]$, stačí nájsť najnižšieho spoločného predka zodpovedajúcich vrcholov. Na obrázku je príklad hľadania RMQ$(1, 3)$ a RMQ$(5, 9)$. Všimnite si, že vrchol zodpovedajúci najmenšiemu prvku v rozsahu musí byť *predkom* všetkých vrcholov v intervale. Toto vyplýva z konštruckie zhora nadol – najmenší prvok daného intervalu spracujeme skôr; bude to koreň daného podstromu, prvky vľavo (vrátane ľavej hranice) budú v ľavom podstrome a prvky vpravo (vrátane pravej hranice) budú v pravom podstrome. Zároveň musí byť aj *najnižším* predkom, pretože inak by vrcholy zodpovedajúce $\ell$ a $r$ boli oba v ľavom alebo oba v pravom podstrome a prvky zodpovedajúce takémuto vrcholu sú mimo zadaného intervalu.

TODO: lineárna konštrukcia

## 17.4 Optimálne riešenie: konštantný čas a lineárna pamäť

Doteraz sme si ukázali mnoho prístupov k problému RMQ, pričom najrýchlejší v konštantnom čase potreboval pamäť $O(n \log n)$. Od toho momentu nás trápi otázka: *Dá sa dosiahnuť ešte lepší algoritmus – konštantný čas a iba lineárna pamäť?*

Odpoveď je prekvapivo áno. A cesta k nemu je ešte prekvapivejšia a zdanlivo úplne nelogická:

*RMQ najskôr zredukujeme na LCA a potom naspäť na RMQ.*

(Prosím?!) Áno, čítate správne: pole, na ktorom chceme riešiť RMQ, najskôr transformujeme na kartézsky strom a úlohu premeníme na LCA. Následne spravíme

Eulerovský prechod tohto stromu, vďaka čomu vznikne úplne nové pole (pole hĺbok), pričom RMQ pôvodného poľa vieme riešiť pomocou RMQ zodpovedajúcich intervalov v novom poli.

Znie to ako šialenstvo, pretože sme práve úlohu RMQ pretransformovali na *inú* úlohu RMQ – a ešte k tomu na väčšom poli. Namiesto toho, aby sme si pomohli, sme si zdalo by sa pridali prácu. Vrátili sme sa tam, odkiaľ sme prišli, len s väčším vstupom.

A predsa je tu jeden podstatný detail: nové pole má veľmi špecifickú štruktúru, keďže vzniklo z Eulerovského prechodu binárneho stromu:

- hodnoty zodpovedajú hĺbkam vrcholov, tzn., že sú to celé čísla od 0 po $n$,
- pole začína nulou a končí nulou, pretože Eulerovský prechod sa začína a končí v koreni, ktorý má hĺbku 0,
- Každé dva susedné prvky sa líšia presne o $\pm 1$, keďže pri pohybe stromom buď zostupujeme o úroveň nižšie (hĺbka $+1$), alebo sa vraciame späť nahor (hĺbka $-1$).

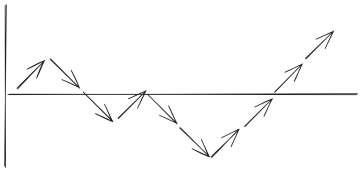
Zatiaľčo pôvodné pole mohlo obsahovať ľubovoľne veľké, aj reálne, aj záporné hodnoty, nové pole už obsahuje iba prvky od 0 po $n$. Zatiaľčo v pôvodnom poli sa susedné prvky mohli líšiť neobmedzene, v novom poli sa líšia vždy o 1. Táto redukovaná úloha sa preto nazýva tiež RMQ$\pm 1$.

Optimálne riešenie RMQ$\pm 1$ začína podobne ako riešenie #5 rozdelením celého poľa na bloky dĺžky zhruba $\log n$. Z každého bloku spočítame minimum a tieto minimá uložíme do poľa $B$, ktoré má dĺžku približne $n/\log n$. Na pole $B$ aplikujeme riešenie s riedkou tabuľkou, ktoré síce používa $O(n \log n)$ pamäte, ale teraz pracuje na $\log n$-krát menšom poli, takže celkové predspracovanie zaberie

$$O\left(\frac{n}{\log n} \times \log n\right) = O(n).$$

a odpovede na dotazy medzi celými blokmi budú v konštantnom čase. Rozdiel oproti predchádzajúcim riešeniam je v tom, ako budeme riešiť prečnievajúce časti, ktoré zasahujú iba čiastnočne do vnútra blokov.

Postupnosť hodnôt v jednom bloku si môžeme lepšie predstaviť ako graf hĺbok. Jeden blok môže vyzerať napríklad takto:



Všimnite si, že ak nás zaujímajú iba minimá v intervaloch v rámci jedného bloku, nezáleží na tom, v akej hĺbke blok začína. Nezáleží ani na tom, aké konkrétne hodnoty blok obsahuje. Záleží len na *tvare* tejto krivky, nie na jej zvislej polohe.

Ak celý blok zvýšime alebo znížime o konštantu, pozície miním v ľubovoľnom intervale sa nezmenia. Každý blok môžeme bez straty informácie normalizovať tak, aby začínal nulou.

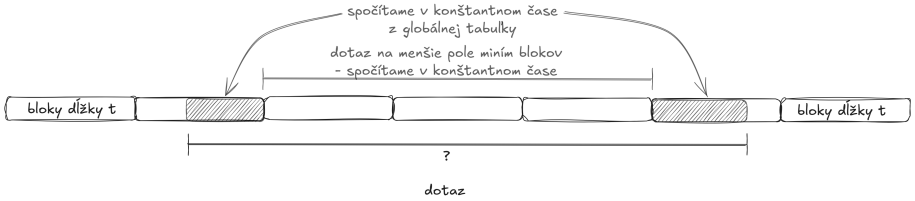
**Koľko existuje rôznych blokov?** V skutočnosti relatívne málo. Blok dĺžky $t$ vieme opísať ako postupnosť krokov $\uparrow$, $\downarrow$ dĺžky $t-1$ a teda celkový počet rôznych tvarov blokov je $2^{t-1}$. Príklad: majme pole dĺžky jeden milión a rozdelíme ho na bloky dĺžky 11 – takto získame zhruba 90-tisíc blokov. Avšak keď sa pozrieme na počet *rôznych tvarov* blokov po normalizácii, zistíme, že existuje iba 1024 rôznych tvarov! Medzi vyše 90 000 blokmi sa teda neustále opakuje len 1024 rôznych tvarov. Vďaka tomu si môžeme dovoliť pre každý možný tvar bloku vopred predpočítať všetky možné odpovede na všetky RMQ dotazy v rámci tohto bloku. Pre blok dĺžky 11 existuje $11 \times 12/2 = 66$ možných podintervalov, krát 1024 rôznych tvarov – to je len zhruba 68-tisíc hodnôt, ktoré si môžeme predpočítať. Zhruba 68-tisíc hodnôt je úplne zanedbateľné množstvo v porovnaní s pôvodným poľom veľkosti milión. Predpočítané hodnoty si uložíme do jednej globálnej tabuľky vďaka ktorej budeme vedieť spočítať minimum ľubovoľného intervalu v rámci jedného bloku v konštantnom čase.

**Finálny algoritmus (Bender a Farach-Colton).** Rozdelíme pole na bloky dĺžky $t = \lfloor \frac{1}{2} \log n \rfloor$. Vznikne tak približne $2n/\log n$ blokov. Z každého bloku spočítame minimum a tieto minimá uložíme do poľa $B$ dĺžky $O(n/\log n)$. Na pole $B$ aplikujeme riešenie #4, riedku tabuľku. Výsledkom je štruktúra, ktorá dokáže nájsť minimum medzi celými blokmi v konštantnom čase a lineárnej pamäti.

Jednotlivé bloky znormalizujeme tak, aby začínali od nuly a zakódujeme ich ako binárne reťazce (napríklad `0` znamená klesanie a `1` stúpanie). Počet všetkých možných tvarov blokov je

$$2^{t-1} \approx 2^{\frac{1}{2}\log n} = n^{1/2} = \sqrt{n}.$$

Počet možných podintervalov v jednom bloku je $\binom{t}{2} = O(\log^2 n)$. Pre každý tvar bloku a každý podinterval si spočítame pozíciu minima – dostaneme globálnu tabuľku veľkosti $O(\sqrt{n} \cdot \log^2 n) = o(n)$.




Ako odpovedáme na dotazy RMQ$(\ell, r)$? Najskôr interval $[\ell, r]$ v pôvodnom poli transformujeme na zodpovedajúci interval $[i, j]$ v poli RMQ$\pm 1$, ktoré vzniklo

Eulerovským prechodom kartézskeho stromu. Ak celý interval $[i, j]$ leží v rámci jedného bloku, predpočítanú odpoveď pre daný tvar bloku a konkrétny podinterval jednoducho nájdeme v globálnej tabuľke. V opačnom prípade si interval rozdelíme na tri časti: kúsok bloku na začiatku (ak $i$ nezačína presne na hranici bloku), kúsok bloku na konci (ak $j$ nekončí presne na hranici bloku) a stredná časť, ktorá sa skladá iba z celých blokov. Na začiatok a koniec použijeme globálnu tabuľku, kde nájdeme príslušný tvar bloku a konkrétny podinterval, na stred použijeme pole $B$ a riešenie s riedkou tabuľkou. Tieto medzivýsledky porovnáme a vyberieme ten najmenší z nich. Tým získame pozíciu minima v poli RMQ$\pm$1 – tú ešte musíme preložiť naspäť na pozíciu v pôvodnom poli.

Každú z týchto častí vyriešime v konštantnom čase – stačí sa pozrieť na správne miesto v príslušnej tabuľke, takže celý dotaz zodpovieme v $O(1)$. Navyše celková pamäť je len lineárna.

# Kapitola 18

# Sufixové pole

V predchádzajúcej kapitole sme videli, že sufixové stromy dokážu riešiť ohromné množstvo úloh nad reťazcami aj množinami reťazcov – rýchlo, elegantne a často v lineárnom čase. Ich slabinou je však pamäťová náročnosť. Aj pri veľmi úspornej implementácii zaberajú typicky 10–20 bajtov na znak a pri textoch dlhých miliardy znakov je to jednoducho priveľa.

Pozrime sa na konkrétny príklad. Ľudský genóm má približne 3 miliardy ($3 \times 10^9$) báz – znakov zo štvorpísmenovej abecedy `A`, `C`, `G`, `T`. Ak si ho uložíme po 1 bajte na znak, zaberá zhruba 3 GB. Pri zhustenej reprezentácii (2 bity na znak) dokonca iba okolo 750 MB.

Ak však nad týmto reťazcom vybudujeme sufixový strom, výsledná štruktúra bude zvyčajne potrebovať 30–60 GB pamäte, teda až približne 80× viac než samotný text. A teraz si predstavte, že chceme naraz analyzovať desiatky či stovky genómov... Pri podobných vstupoch nás veľmi rýchlo začne limitovať veľkosť RAM.

Samozrejme, vďaka virtuálnej pamäti vieme pracovať aj s oveľa väčšími dátami, no ak RAM zaplníme, systém začne stránky odkladať na disk (*page swapping*). Pri ďalšom použití ich bude musieť opäť načítať, zatiaľ čo iné stránky vysunie na disk, aby uvoľnil miesto. Toto neustále presúvanie je extrémne drahé.

Vysoká pamäťová náročnosť tak prirodzene viedla výskumníkov k hľadaniu úspornejších alternatív.

Vráťme sa k úvodným úlohám z kapitoly o sufixových stromoch. Tam sme videli, že úlohy na prefixoch vieme riešiť pomocou písmenkových stromov (trie), ale tesne na druhom mieste bolo jednoduché riešenie: *zotriediť reťazce lexikograficky*. A tak, podobne ako sufixový strom je „písmenkový strom zo všetkých sufixov“, myšlienka sufixového poľa je jednoduchá: vezmime všetky sufixy, zotrieďme ich podľa abecedy a uložme si výsledné poradie.

Napríklad, ak zotriedime všetky sufixy slova `MISSISSIPPI$`, dostaneme:

```
11  $
10  I$
 7  IPPI$
 4  ISSIPPI$
 1  ISSISSIPPI$
 0  MISSISSIPPI$
 9  PI$
 8  PPI$
 6  SIPPI$
 3  SISSIPPI$
 5  SSIPPI$
 2  SSISSIPPI$
```

Samozrejme, nebudeme si pamätať kópie všetkých $n$ sufixov explicitne – to by zaberalo $\Theta(n^2)$ pamäte. Úplne stačí pamätať si pôvodný reťazec a každý sufix reprezentovať jedným číslom: indexom jeho začiatku. Sufixové pole je preto jednoducho pole $n$ celých čísel – presne ľavý stĺpec v príklade vyššie.

Pamäťová zložitosť sufixového poľa je teda oveľa lepšia: potrebujeme len jedno celé číslo na každý znak textu. V našom príklade s 3 miliardovým DNA reťazcom nám stačia 32-bitové indexy, takže celé sufixové pole zaberie približne 12 GB (plus pôvodný reťazec, ktorý pri 2 bitoch na znak zaberie asi 0.75 GB). To je obrovský rozdiel oproti 60 GB sufixovému stromu.

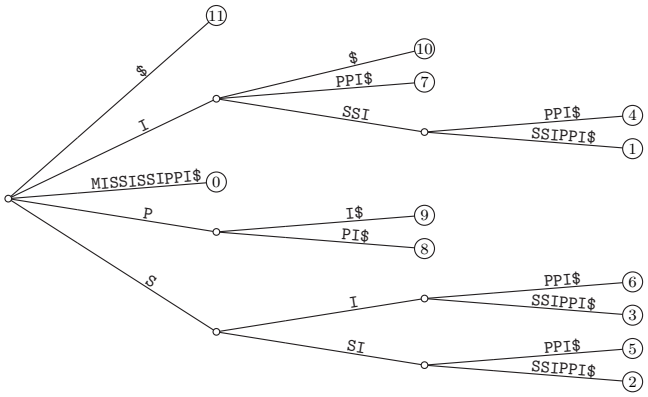
## 18.1 Vzťah sufixových stromov a sufixových polí

Hoci sufixové pole (*suffix array*, SA) pôsobí oveľa jednoduchšie než plnohodnotný sufixový strom, v skutočnosti sú tieto dve štruktúry takmer ekvivalentné. Väčšinu aplikácií sufixových stromov vieme s pomocou sufixových polí vyriešiť tiež, ak si k nemu doplníme ešte jedno pole: *LCP pole* (Longest Common Prefix), ktoré ku každému dvojici *susedných* sufixov v sufixovom poli uchováva dĺžku ich najdlhšieho spoločného prefixu.

Ak si porovnáme sufixový strom a sufixové pole spolu s LCP poľom, ukáže sa, že:

- sufixové pole aj LCP vieme odvodiť zo sufixového stromu v lineárnom čase: všimnite si, že ak máme v každom vrchole stromu zotriedené znaky, tak prechodom listov zhora nadol dostaneme utriedenú postupnosť sufixov; a ak si pri prehľadávaní budeme pamätať textovú hĺbku, vieme zároveň vypisovať hodnoty LCP (ako vysoko sme sa pri prehľadávaní museli vrátiť, než sme sa zanorili do ďalšieho podstromu);

- naopak, aj sufixový strom sa dá zrekonštruovať z SA+LCP v lineárnom čase: stačí strom začať budovať zhora nadol; LCP pole nám prezradí, do akej hĺbky sa máme vrátiť, a kde máme odpojiť novú vetvu;




| *LCP* | *SA* | *sufixy* |
|---|---|---|
| 0 | 11 | `$` |
| 1 | 10 | `I$` |
| 1 | 7 | `IPPI$` |
| 4 | 4 | `ISSIPPI$` |
| 0 | 1 | `ISSISSIPPI$` |
| 0 | 0 | `MISSISSIPPI$` |
| 1 | 9 | `PI$` |
| 0 | 8 | `PPI$` |
| 2 | 6 | `SIPPI$` |
| 1 | 3 | `SISSIPPI$` |
| 3 | 5 | `SSIPPI$` |
| – | 2 | `SSISSIPPI$` |

Obr. 18.1: Vľavo sufixový strom, vpravo sufixové pole a LCP pole pre reťazec `MISSISSIPPI$`.

- LCP pole zodpovedá textovým hĺbkam vnútorných vrcholov; napríklad vrchol na ceste „`ISSI`“ sa nachádza v kompaktnom sufixovom strome, pretože je to najdlhší spoločný začiatok sufixov `ISSIPPI$` a `ISSISSIPPI$`; po štvrtom písmene sa sufixy oddelia;

- podstromy v sufixovom strome zodpovedajú intervalom v SA; napríklad ak prejdeme cestu „`I`“, všetky výskyty tejto vzorky sú listy daného podstromu: 10, 7, 4, 1; v sufixovom strome to zodpovedá intervalu $1 \ldots 4$ – vďaka triedeniu sa všetky sufixy začínajúce na „`I`“ dostanú vedľa seba.

## 18.2 Vyhľadávanie

Najjednoduchší spôsob, ako pomocou sufixového poľa nájsť, či sa vzorka $P$ nachádza v texte $T$, je obyčajné binárne vyhľadávanie. V sufixovom poli sú všetky sufixy utriedené lexikograficky, takže všetky sufixy začínajúce na $P$ tvoria jeden súvislý úsek. Stačí teda pomocou binárneho vyhľadávania nájsť ľavú a pravú hranicu tohto úseku.

Binárne vyhľadávanie trvá $O(\log n)$ krokov, avšak každé porovnanie dvoch reťazcov môže trvať až $O(m)$, preto je celková zložitosť

$$O(m \log n).$$

To je horšie než pri sufixových stromoch, kde sme vyhľadávali v čase $O(m)$.

**Dá sa to zrýchliť?** Ukáže sa, že áno – ak sme ochotní použiť trochu viac pamäte. Kľúčovou pomocnou štruktúrou je pole $LCP$, kde $\mathrm{lcp}(i, j)$ označuje dĺžku najdlhšieho spoločného prefixu $i$-teho a $j$-teho sufixu v sufixovom poli.

LCP pole nám umožní pri porovnávaní preskakovať časti reťazcov, ktoré sme už raz porovnali.

Hľadajme vzorku $P$. Počas binárneho vyhľadávania si budeme udržiavať dva indexy: $\ell$ a $r$, pričom bude platiť, že hľadaná vzorka sa nachádza niekde medzi tým, presnejšie:

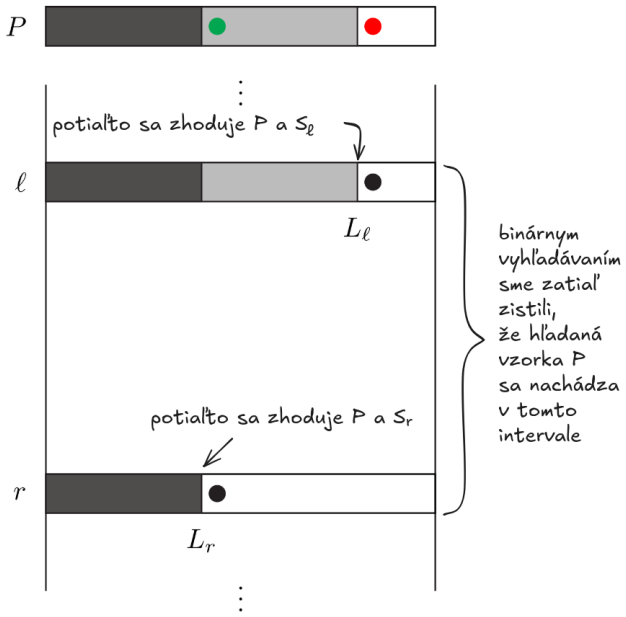
$$S[SA[\ell]] < P \leq S[SA[r]].$$

Keďže povedať $S[SA[i]]$ je celkom nálož, budeme hovoriť jednoducho $i$-ty sufix (v utriedenom poradí) a značiť ho $S_i$.

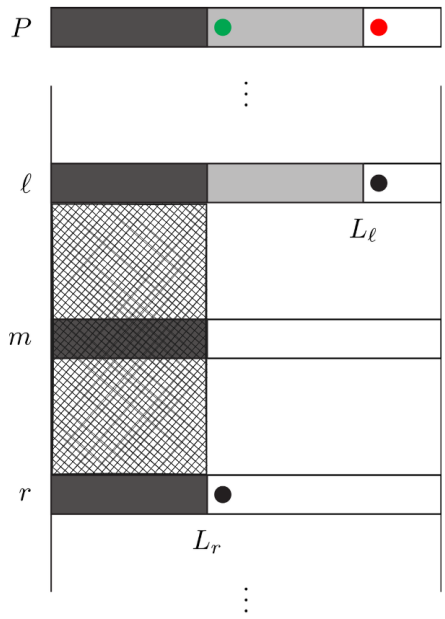
Zároveň si uchovávame dve hodnoty:

$$L_\ell = \mathrm{lcp}(P, S_\ell), \qquad L_r = \mathrm{lcp}(P, S_r),$$

teda dĺžku spoločného prefixu $P$ a sufixu na ľavej a pravej hranici. (Tieto prefixy znázorňujú šedé úseky na obrázku. Za šedou zhodou nasleduje znak, v ktorom sa sufixy líšia – červený znak pri $\ell$ a zelený pri $r$.)



Pozrime sa teraz dostredu, na pozíciu $m$.

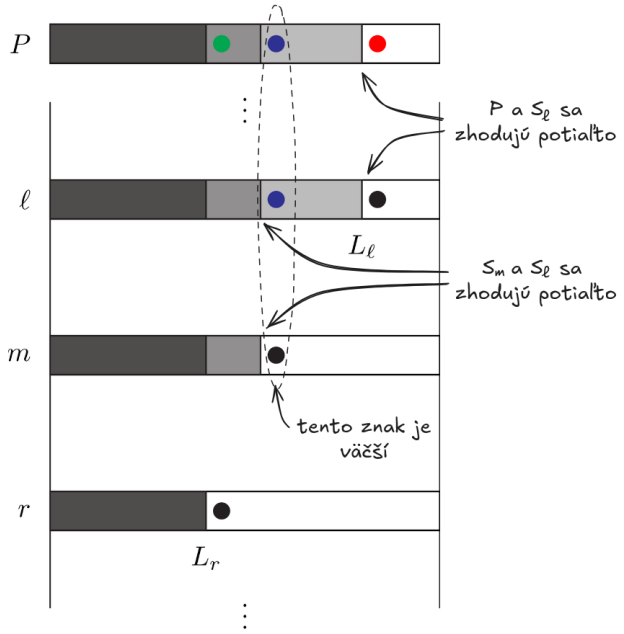



Keďže $\ell$-tý a $r$-tý sa zhodujú s $P$ v prvých $\min(L_\ell, L_r)$ znakoch, aj všetky reťazce medzi nimi (keďže sú zotriedené lexikograficky), sa musia začínať na tie isté znaky (ako znázorňuje vyšrafovaná oblasť). Tieto znaky už teda nemusíme porovnávať.

Ba čo viac, pozrime sa teraz na $p = \text{lcp}(S_\ell, S_m)$ (bez újmy na všeobecnosti predpokladajme, že $L_\ell \geq L_r$, teda že $S_l$ má dlší spoločný začiatok s hľadanou vzorkou ako $S_r$ – v opačnom prípade budeme postupovať symetricky a pozrieme sa na $p = \text{lcp}(S_r, S_m)$).

Sú tri možné prípady:

**Prípad 1:** $p < L_\ell$**.** To znamená, že $S_\ell$ a $P$ majú *dlhší spoločný začiatok* ako $S_\ell$ s $S_m$.

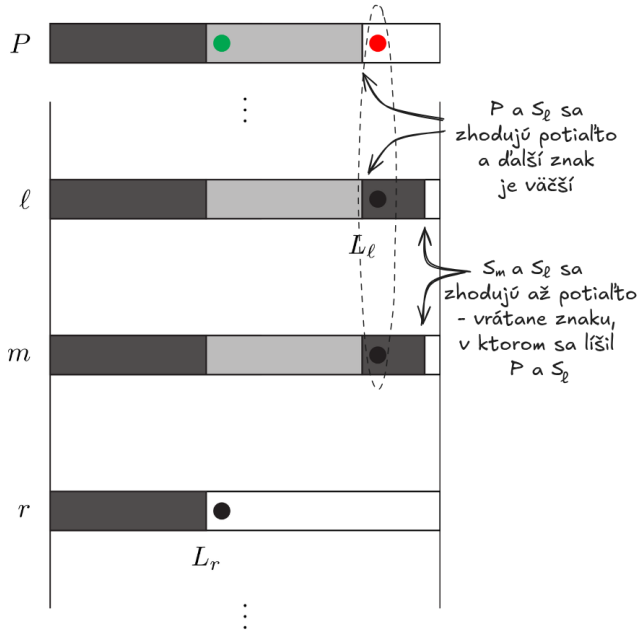



Čo z toho vyplýva? Nuž vyplýva z toho, že po $p$-ty znak sa $P$, $\ell$-tý, aj $m$-tý sufix rovnajú, ale $(p+1)$-vý znak má $S_m$ odlišný(!), konkrétne väčší, ako znak, ktorý zdieľajú $S_\ell$ a $P$.

Takže $P < S_m$ a treba hľadať v hornej polovici poľa. Zároveň platí, že $L_m = p$.

Všimnite si, čo sa stalo: nemuseli sme porovnať ani jeden znak a zistili sme, ktorou cestou sa v binárnom vyhľadávaní máme vydať.

**Prípad 2:** $p > L_\ell$**.** To znamená, že $S_\ell$ s $S_m$ majú *dlhší spoločný začiatok* ako $S_\ell$ a $P$.




Čo z toho vyplýva? Nuž, zjavne $S_m$ obsahuje ten istý znak, v ktorom sa líšili $S_\ell$ od $P$, takže výsledok porovnania bude ten istý.

Keďže $P > S_\ell$, tak $P > S_m$ a v tomto prípade treba hľadať v dolnej polovici poľa. Zároveň platí, že $L_m = L_\ell$.

Opäť sme zistili, ktorým smerom pokračovať v hľadaní bez toho, aby sme porovnali čo i len jediný znak!

**Prípad 3:** $p = L_\ell$**.** V tomto prípade nevieme okamžite povedať výsledok, takže začneme porovnávať $S_m$ a $P$ znak po znaku. Samozrejme, prvých $p$ znakov preskočíme. Na koniec podľa výsledku porovnania zistíme, či máme pokračovať v hornej alebo dolnej polovici a zároveň spočítame $L_m$ – aký dlhý je spoločný prefix $L_m$ a $P$.

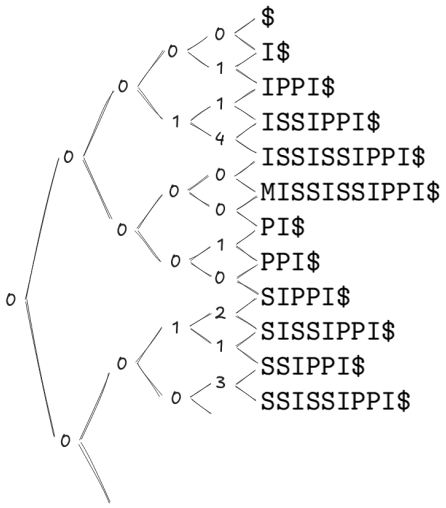
**Zložitosť.** Predpokladajme, že všetky hodnoty LCP si predpočítame dopredu. Potom prípad 1 a 2 trvá iba konštantný čas, takže všetky tieto kroky spolu trvajú iba $O(\log n)$ času. A koľko trvá prípad 3? Tvrdím, že dokopy za celé vyhľadávanie spravíme len $O(m+\log n)$ porovnaní, pretože sa nikdy nevraciame na pozície, ktoré sme už porovnali; keďže vzorka má dĺžku $m$, môžeme sa iba $m$-krát posunúť doprava. Okrem toho, ak nájdeme nezhodu, ostávame na rovnakej pozícii, avšak pri nezhode sa porovnávanie končí, takže týchto krokov je len $\log n$. Výsledná zložitosť je teda

$$O(m + \log n),$$

čo je skoro rovnako dobré ako $O(m)$ pri sufixových stromoch – keďže aj pre texty dlhé miliardy znakov je $\lg n \approx 30$, v praxi je často $m > \log n$ ten dominantný člen.

Na druhej strane, za toto rýchlejšie vyhľadávanie zaplatíme horšou pamäťou. Musíme si predpočítať nielen LCP susedných sufixov, ale aj LCP dvojíc, ktoré sú ďalej od seba.

Našťastie netreba LCP každej dvojice – stačí si prejsť rozhodovací strom pri binárnom vyhľadávaní a zapamätať si iba tie hodnoty, ktoré potrebujeme. Praktické riešenie je (aspoň pomyselne) „zaokrúhliť“ dĺžku sufixového poľa na najbližšiu väčšiu mocninu 2 a vybudovať nad LCP poľom úplný binárny strom. LCP dvoch vzdialenejších sufixov spočítame ako minimum LCP dvoch detí. Binárne vyhľadávanie potom prispôsobíme tak, že začneme s $r$ mocninou dvojky:




## 18.3 Konštrukcia LCP poľa

Pole LCP (Longest Common Prefix) je pre sufixové pole rovnako dôležité, ako bola textová hĺbka pri sufixových stromoch. Definujme

$$\mathrm{lcp}[i] = \mathrm{lcp}(S[\mathrm{SA}[i]],\ S[\mathrm{SA}[i-1]]), \qquad i = 1, \ldots, n,$$

teda dĺžka najdlhšieho spoločného prefixu dvoch po sebe idúcich sufixov v utriedenom poradí. Hodnota lcp[0] sa zvykne definovať ako 0.

Použitím LCP poľa sa mnohé operácie nad sufixovým poľom dramaticky zrýchlia– napríklad zrýchlené binárne vyhľadávanie na $O(m+\log n)$, vyhľadávanie opakujúcich sa podreťazcov, najdlhší opakujúci sa substring a množstvo ďalších úloh.

Otázka znie: *Ako rýchlo LCP pole skonštruovať, ak už máme sufixové pole?*

Ukážeme si klasický algoritmus od Kasaiho, ktorý to zvládne v čase $O(n)$.

## 18.4 Kasaiho algoritmus

Kľúčová myšlienka algoritmu je jednoduchá:

> Ak poznáme $\mathrm{lcp}(i, j)$, potom $\mathrm{lcp}(i+1, j+1)$ je určite *aspoň* $\mathrm{lcp}(i, j) - 1$.

Inými slovami: Dva susedné sufixy $T[i\,..]$ a $T[j\,..]$ zdieľajú určitý spoločný prefix dĺžky $h$. Keď ich „o znak posunieme doprava“ a porovnávame $T[i+1\,..]$ a $T[j+1\,..]$, tak spoločný prefix je dlhý aspoň $h-1$.

Takže ak pri výpočte ďalšieho lcp vždy začneme porovnávať nie od nuly, ale od $h-1$, dohromady vykonáme najviac $n$ porovnaní znakov navyše.

Celý algoritmus funguje v troch krokoch:

1. *Vytvoríme inverzné sufixové pole.* Pre každý index $i$ v texte chceme poznať jeho pozíciu v sufixovom poli:

   $$\mathrm{rank}[i] = \text{pozícia sufixu } T[i..] \text{ v sufixovom poli.}$$

   To vieme zostrojiť v čase $O(n)$.

2. *Prejdeme sufixy v poradí od najdlhšieho po najkratší (nie v utriedenom poradí!).* Teda v poradí $i = 0, 1, 2, \ldots, n-1$. Pomocou `rank[i]` zistíme, ktorý je predchádzajúci sufix v SA:

   $$j = \mathrm{SA}[\,\mathrm{rank}[i] - 1\,],$$

   a chceme spočítať $\mathrm{lcp}(i, j)$.

3. *Porovnávame znaky.* Hodnotu $h$, ktorú si uchovávame medzi iteráciami, vždy najskôr znížime o 1 (ak $h > 0$), a potom od tejto pozície ďalej porovnávame znaky, kým sú rovnaké:

   $$\text{while } T[i+h] = T[j+h], \text{ zväčši } h.$$

   Výsledkom je

   $$\mathrm{lcp}[\mathrm{rank}[i]] = h.$$

   Po skončení iterácie prechádzame na $i+1$.

Algoritmus je uvedený nižšie:

```
h = 0
for i = 0 .. n-1:
    if rank[i] = 0:
        LCP[0] = 0
        continue
    j = SA[rank[i] - 1]
    while T[i+h] == T[j+h]: h++
    LCP[rank[i]] = h
    if h > 0: h--
```

**Prečo je časová zložitosť lineárna?** V celom algoritme sa vykonávajú dva typy prác:

1. Pri každom kroku robíme konštantnú prácu (prístup do polí SA, rank).
2. V cykle `while` porovnávame znaky $T[\cdot]$.

Trik je v tom, že premenná $h$ za celý beh nikdy nenarastie na viac než $n$. Zároveň len $n$-krát klesne o 1. Z toho vyplýva, že nemôže stúpnuť viac ako $2n$-krát. Pri každom porovnaní máme buď zhodu (vtedy $h$ stúpne, takže týchto krokov je najviac $2n$), alebo nezhodu (vtedy porovnávanie končí a ideme na ďalšiu dvojicu, takže týchto krokov je najviac $n$). Celkový čas algoritmu je teda lineárny.

## 18.5 Konštrukcia sufixového poľa

Chceme zotriediť $n$ sufixov. Ako na to?

### Quicksort?

Najjednoduchší nápad je uložiť si všetkých $n$ sufixov a zotriediť ich sortom zo štandardnej knižnice. To však znamená triediť reťazce priemernej dĺžky $n/2$, takže celkový čas bude $O(n^2 \log n)$ a pamäť $O(n^2)$. Pre dlhšie texty absolútne nepoužiteľné.

Lepšie riešenie je využiť, že v štandardných knižniciach často vieme na triedenie zadať *vlastnú* porovnávaciu funkciu. Potom stačí triediť indexy $0 \ldots n-1$, akurát pri porovnaní $i$ a $j$ porovnáme sufixy $T[i..]$ a $T[j..]$. Časová zložitosť bude rovnaká, ale pamäť bude lineárna.

Pre niektoré texty (napríklad v prirodzenom jazyku, ktoré nemajú príliš veľa dlhých opakovaní) to môže byť dokonca celkom praktický algoritmus. Ak totiž označíme $L$ ako najdlhší spoločný prefix ľubovoľných dvoch sufixov, potom pri každom porovnaní stačí porovnať len $L$ znakov a teda skutočná zložitosť je $O(L \times n \log n)$. Samozrejme, v najhoršom prípade je $L = n - 1$; zložitosť je stále $O(n^2 \log n)$ v najhoršom prípade, avšak ak je $L$ malé, tento čas môže byť zvládnuteľný.

Dá sa to lepšie?

### Radixsort?

Triedime predsa reťazce – na to je vhodnejší radixsort ako quicksort. V najhoršom prípade dostávame zložitosť $O(n^2)$ – pre texty dlhé miliardy znakov stále nepoužiteľné.

Všimnite si, že doteraz sme uvažovali iba všeobecné algoritmy, ktoré dokážu utriediť ľubovoľnú postupnosť reťazcov. Ak chceme lepší algoritmus, musíme využiť, že triedime veľmi špeciálne reťazce – všetky sufixy daného textu.

## Manber–Myersov $n \log n$ algoritmus

Kľúčová myšlienka je prekvapivo jednoduchá:

> *Sufix sufixu je opäť sufix.*

Ak máme sufixy utriedené podľa prvých $K$ znakov, vieme ich veľmi jednoducho zotriediť podľa prvých $2K$ znakov. To nám umožní „zdvojnásobiť“ počet porovnaných znakov v každej fáze.

Algoritmus pracuje v $\log n$ fázach. V $k$-tej fáze budeme mať sufixy zotriedené podľa prvých $2^k$ znakov.

Označme:

$$r[i] = \text{poradové číslo sufixu } s_i \text{ v triedení podľa prvých } 2^k \text{ znakov.}$$

Ak majú dva rôzne sufixy rovnaký prefix dĺžky $2^k$, ich ranku sa priradí rovnaká hodnota. Takto stačí v každej fáze zotriediť nie celé reťazce, ale len trojice čísel:

$$(r[i],\ r[i + 2^k],\ i),$$

kde:

- $r[i]$ určuje poradie podľa prvých $2^k$ znakov,
- $r[i + 2^k]$ určuje poradie „druhej polovice prefixu“,
- $i$ je samotná pozícia sufixu.

Na začiatku (pre $k = 0$) sufixy triedime podľa prvého znaku – stačí jednoducho utriediť znaky abecedy.

V každej ďalšej fáze potom všetky sufixy zotriedime podľa dvojice $(r[i], r[i + 2^k])$.

Ak použijeme quicksort, jedna fáza bude trvať $O(n \log n)$, takže celkovo bude konštrukcia trvať maximálne $O(n \log^2 n)$ času.

Avšak keďže triedime celé čísla v rozsahu $0 \ldots n$, môžeme použiť radix sort a implementovať jednu fázu v lineárnom čase. Celkový čas potom bude $O(n \log n)$.

### Zopár praktických vylepšení:

1. V prvej fáze nemusíme triediť podľa jediného písmena. Namiesto toho sufixy pred-triedime podľa prvých povedzme 64 alebo 128 písmen.
2. Nemusíme dokončiť všetkých $\lg n$ fáz – ak už sú všetky sufixy rozlíšené a jednoznačne zotriedené, môžeme skončiť.
3. Môžeme si pamätať úseky s rovnakým rankom a sústrediť sa len na ich triedenie.

## Kärkkäinenov–Sandersov lineárny algoritmus

2003 bol dobrý rok. Dovoľte malú historickú vsuvku. Sufixové stromy vynašiel Peter Weiner už v roku 1973 a zároveň predstavil aj prvý lineárny algoritmus na ich konštrukciu. McCreight (1976) a neskôr Ukkonen (1995) objavili jednoduchšie a praktickejšie lineárne algoritmy. Tieto riešenia však ticho predpokladali, že veľkosť abecedy je konštantná. Až Farach (1997) publikoval prvý algoritmus, ktorý bol optimálny pre ľubovoľnú abecedu (kde „znaky“ môžu byť napríklad čísla v rozsahu 1 až $n^{O(1)}$).

Sufixové polia ako pamäťovo úspornejšiu alternatívu sufixových stromov predstavili Manber a Myers v roku 1990. Pomerne dlho však nikto nepoznal *priamu* lineárnu konštrukciu sufixového poľa – priamu v zmysle, že by nebolo potrebné najskôr zostrojiť celý sufixový strom a až z neho odvodiť pole.

A potom prišiel rok 2003.

V tom istom roku vyšli *tri* nezávislé články, ktoré po viac ako desaťročí priniesli *tri rôzne* lineárne algoritmy na konštrukciu sufixových polí.

V tejto kapitole si ukážeme algoritmus Kärkkäinena a Sandersa (často sa označuje ako *skew* algoritmus[1] alebo tiež DC3 (podľa použitého „difference cover“ modulo 3).

Začnime myšlienkou, ktorá pochádza z Farachovho algoritmu na konštrukciu sufixových stromov. Bez toho, aby sme zachádzali do detailov, Farach rozdelil všetky sufixy na dve skupiny: tie, ktoré začínajú na *párnych* pozíciách, a tie, ktoré začínajú na *nepárnych* pozíciách.

Najskôr si rekurzívne zostrojil sufixový strom pre sufixy na nepárnych indexoch. Z tohto stromu potom dokázal odvodiť poradie sufixov na párnych pozíciách. Nakoniec obe množiny zlúčil a získal kompletný sufixový strom pre celý text.

Skúsme podobný prístup použiť pri konštrukcii sufixového poľa. Vezmime si ako príklad reťazec `MISSISSIPPI$`. Predstavme si, že sa nám už podarilo zotriediť všetky sufixy začínajúce na *nepárnych* pozíciách:

```
11  $
 7  IPPI$
 1  ISSISSIPPI$
 9  PI$
 3  SISSIPPI$
 5  SSIPPI$
```

Ako teraz zotriediť sufixy na *párnych* pozíciách?

```
 0  MISSISSIPPI$
 2  SSISSIPPI$
 4  ISSIPPI$
 6  SIPPI$
 8  PPI$
10  I$
```

[1] “skew” by sme mohli preložiť ako „asymetrický“, keďže algoritmus pracuje s asymetrickým rozdelením indexov.

Kľúčové pozorovanie: každý párny sufix pozostáva z *prvého písmena* a za ním nasleduje *nepárny sufix* – a tieto nepárne sufixy už máme zotriedené! Napríklad:

- `SIPPI$` začína písmenom `S` a za ním nasleduje sufix 7, ktorý je v utriedenom poradí druhý;
- `SSISSIPPI$` začína písmenom `S`, ale za ním nasleduje sufix 3, ktorý je v poradí až piaty.

Takže každý párny sufix môžeme reprezentovať dvojicou:

(prvé písmeno, poradie nasledujúceho nepárneho sufixu).

Pre náš príklad to vyzerá takto:

| | |
|---|---|
| 0 | (`M`, 3) |
| 2 | (`S`, 5) |
| 4 | (`I`, 6) |
| 6 | (`S`, 2) |
| 8 | (`P`, 4) |
| 10 | (`I`, 1) |

Ak už teda poznáme poradie nepárnych sufixov, poradie párnych sufixov dokážeme získať pomocou jednoduchého dvojfázového radix sortu podľa tejto dvojice:

| | | |
|---|---|---|
| 10 | (`I`, 1) | `I$` |
| 4 | (`I`, 6) | `ISSIPPI$` |
| 0 | (`M`, 3) | `MISSISSIPPI$` |
| 8 | (`P`, 4) | `PPI$` |
| 6 | (`S`, 2) | `SIPPI$` |
| 2 | (`S`, 5) | `SSISSIPPI$` |

Ako však dostaneme poradenie nepárnych sufixov?

Rekurzívne. Použijeme nasledovný trik: Zoberieme náš reťazec, odhodíme jeho prvé písmeno a zvyšok si rozdelíme na dvojice. Každú dvojicu budeme považovať za jeden nedeliteľný „super-znak“:

| `IS` | `SI` | `SS` | `IP` | `PI` | `$$` |
|---|---|---|---|---|---|

Na takto vytvorený reťazec sa teraz pozeráme ako na nový reťazec zo šiestich znakov. Všimnite si, že jeho sufixy zodpovedajú práve nepárnym sufixom pôvodného reťazca. Ak teda dokážeme zotriediť sufixy tohto „zbaleného“ reťazca, dostaneme poradie všetkých nepárnych sufixov pôvodného textu.

Nakoniec nám ostáva už „iba“ jeden krok: zobrať dve utriedené sufixové polia – jedno s nepárnymi sufixami a druhé s párnymi – a zlúčiť ich do jedného spoločného, úplne utriedeného sufixového poľa.

Ukazuje sa však, že toto zlučovanie vôbec nie je také jednoduché, ako by sa mohlo zdať. Hoci existuje riešenie v lineárnom čase (vďaka práci Kim et al.), technicky je pomerne komplikované.

A tu prichádza elegantný trik algoritmu DC3. Namiesto delenia sufixov na párne a nepárne spravíme rozdelenie *asymetricky*: na sufixy začínajúce na indexoch deliteľných tromi (približne tretina všetkých sufixov) a na sufixy začínajúce na indexoch nedeliteľných tromi (zvyšné dve tretiny).

Predstavme si, že sufixy na pozíciách $i \not\equiv 0 \pmod 3$ už máme zotriedené:

```
11  $
10  I$
 7  IPPI$
 4  ISSIPPI$
 1  ISSISSIPPI$
 8  PPI$
 5  SSIPPI$
 2  SSISSIPPI$
```

Ako zotriediť zostávajúce sufixy, teda tie na pozíciách $i \equiv 0 \pmod 3$? Rovnakým spôsobom ako v predošlom prístupe: pre každý taký sufix si vytvoríme dvojicu

$$(\text{prvé písmeno}, \text{pozícia o 1 kratšieho sufixu v už utriedenom poli})$$

a tieto dvojice zotriedime radix sortom:

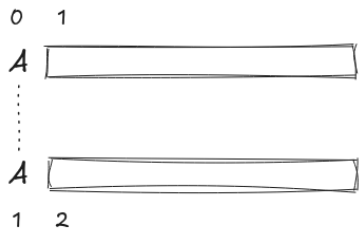
```
0  MISSISSIPPI$  (M, 5)
3  SISSIPPI$     (S, 4)
6  SIPPI$        (S, 3)
9  PI$           (P, 2)
```

Zotriedením týchto dvojíc dostaneme poradie sufixov 0, 9, 6, 3.

Teraz máme dve utriedené sufixové polia a potrebujeme ich zlúčiť do jedného kompletného poradia všetkých sufixov. Ako na to?

Utriedené polia zlučujeme podobne ako v mergesorte. Kľúčové je vedieť rýchlo porovnať sufix s indexom $i \equiv 0 \pmod 3$ voči sufixu s indexom $j \equiv 1$ alebo $2 \pmod 3$.

**Prípad 0 vs. 1 modulo 3.** Ak porovnávame sufixy začínajúce na indexoch $i \equiv 0 \pmod 3$ a $j \equiv 1 \pmod 3$, stačí porovnať ich prvé písmená. Ak sú rovnaké, posúvame sa o jeden znak ďalej.

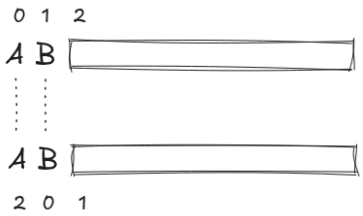



Zvyšok týchto dvoch sufixov začína na indexoch

$$i + 1 \equiv 1 \pmod 3, \qquad j + 1 \equiv 2 \pmod 3,$$

teda oba patria medzi indexy nedeliteľné tromi a ich vzájomné poradie už poznáme!

**Prípad 0 vs. 2 modulo 3.** Podobne, ak porovnávame sufixy na indexoch $i \equiv 0 \pmod 3$ a $j \equiv 2 \pmod 3$, najskôr porovnáme prvé písmeno. Ak sú rovnaké, porovnáme druhé písmeno.




Ak aj tie sú rovnaké, zvyšné časti sufixov začínajú na indexoch

$$i + 2 \equiv 2 \pmod 3, \qquad j + 2 \equiv 1 \pmod 3,$$

a teda opäť sme sa dostali na dva indexy nedeliteľné tromi, ktorých vzájomné poriadie už poznáme.

Tu vidíme, prečo sa oplatilo rozdeľovať úlohu *asymetricky*. V oboch prípadoch sa po jednom až dvoch krokoch presunieme na indexy *nedeliteľné tromi*. Tieto indexy patria do jednej spoločnej množiny, ktorú už máme zotriedenú. Vďaka tomu dokážeme rozhodnúť výsledok každého porovnania v konštantnom čase a zlučovanie oboch utriedených polí prebehne jednoduchým lineárnym prechodom.

Ako spočítame sufixové pole pre indexy nedeliteľné tromi?

Opäť použijeme podobný trik ako v predchádzajúcom prístupe. Zoberieme náš pôvodný reťazec od indexu 1 (teda $T[1 \ldots n]$), prípadne ho na konci doplníme znakmi `$`, aby mal dĺžku deliteľnú tromi. Zaň zapíšeme ešte jednu kópiu reťazca od indexu 2 (teda $T[2 \ldots n]$, opäť doplnenú `$` podľa potreby). Spolu tak dostaneme nový text dĺžky približne $2n$.

Tento nový reťazec si následne rozdelíme na postupné trojice znakov. Každú trojicu budeme považovať za jeden nedeliteľný „super-znak“:

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| ISS | ISS | IPP | I$$ | SSI | SSI | PPI | $$$ |

Dĺžka tohto reťazca je teda $\approx 2n/3$.

Všimnite si, že sufixy z prvej polovice tohto nového reťazca presne zodpovedajú sufixom, ktoré začínajú na indexoch so zvyškom 1 (mod 3). Rovnako sufixy z druhej polovice zodpovedajú sufixom na indexoch so zvyškom 2 (mod 3).

Presnejšie, ak si reťazec porovnáme s tým pôvodným,

```
0 1 2 3 4 5 6 7 8 9 10 11
M I S S I S S I P P I $
```

vidíme, že sufixy 0, 1, 2, 3 z prvej polovice zodpovedajú indexom 1, 4, 7, 10 a sufixy 4, 5, 6, 7 z druhej polovice zodpovedajú indexom 2, 5, 8, 11 v pôvodnom reťazci.

(Sufixy v prvej polovici síce obsahujú „čosi navyše“ na konci – druhú kópiu textu – ale z hľadiska poradia a porovnávania je všetko za znakmi `$` irelevantné.)

Ak teda rekurzívne spočítame sufixové pole

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 7 | `$$$` | | | | | | | |
| 3 | `I$$` | `SSI` | `SSI` | `PPI` | `$$$` | | | |
| 2 | `IPP` | `I$$` | `SSI` | `SSI` | `PPI` | `$$$` | | |
| 1 | `ISS` | `IPP` | `I$$` | `SSI` | `SSI` | `PPI` | `$$$` | |
| 0 | `ISS` | `ISS` | `IPP` | `I$$` | `SSI` | `SSI` | `PPI` | `$$$` |
| 6 | `PPI` | `$$$` | | | | | | |
| 5 | `SSI` | `PPI` | `$$$` | | | | | |
| 4 | `SSI` | `SSI` | `PPI` | `$$$` | | | | |

vieme z neho spätne zrekonštruovať pôvodné indexy a sufixové pole pre indexy nedeliteľné tromi:

| | |
|---|---|
| 11 | `$` |
| 10 | `I$` |
| 7 | `IPPI$` |
| 4 | `ISSIPPI$` |
| 1 | `ISSISSIPPI$` |
| 8 | `PPI$` |
| 5 | `SSIPPI$` |
| 2 | `SSISSIPPI$` |

No moment – a nie je to celé podvod?

To sa naozaj môžeme len tak rozhodnúť, že trojica znakov bude jeden „super-znak“ a tváriť sa, že nový reťazec má iba tretinovú dĺžku?

Pôvodne mal text abecedu veľkosti $\sigma$. Trojice znakov však pochádzajú z množiny všetkých trojíc Trojice znakov však pochádzajú z množiny všetkých trojíc $\Sigma^3 = \{(a, b, c) : a, b, c \in \Sigma\}$, ktorá má $\sigma^3$ prvkov. A keď sa v rekurzii zanoríme ešte o krok hlbšie, budeme pracovať s abecedou veľkosti $(\sigma^3)^3 = \sigma^9$, v ďalšom volaní dokonca $((\sigma^3)^3)^3 = \sigma^{27}$ „super-super-super znakov“.

Veľkosť abecedy nám teda rastie prudko exponenciálne a tieto „super-znaky“ sú čoraz zložitejšie objekty. Tradične pritom predpokladáme, že dva znaky vieme porovnať v konštantnom čase. Ale ako máme porovnávať tieto „super-$\cdots$-super“ znaky?

Našťastie má tento problém jednoduché riešenie:

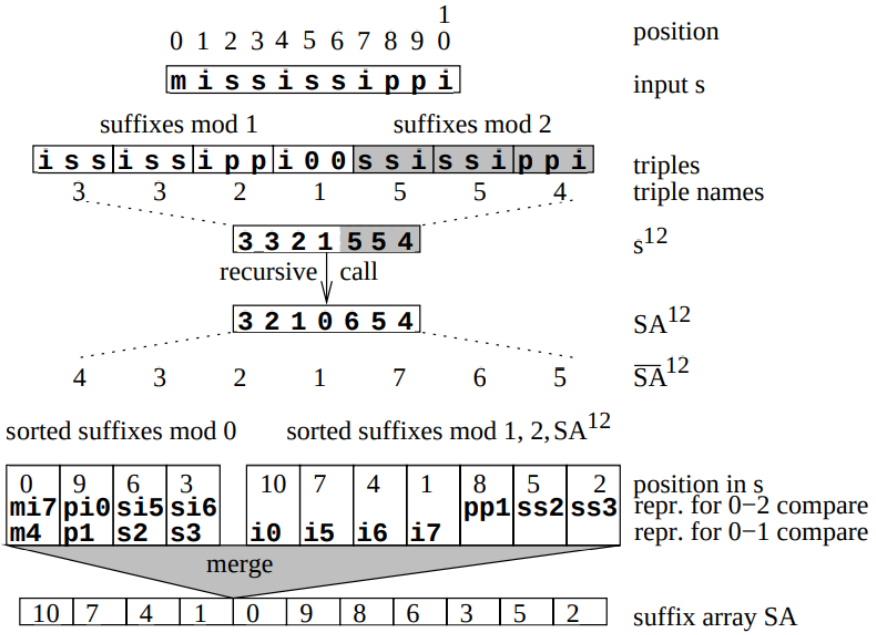
> Každý reťazec dĺžky $n$ môže obsahovať najviac len $n$ rôznych znakov.

Inými slovami: aj keby bola pôvodná abeceda obrovská, v konkrétnom vstupe sa reálne vyskytne najviac $n$ rôznych symbolov. A presne to isté platí aj pre naše „super-znaky“: hoci teoreticky pochádzajú z veľkej množiny $\Sigma^3$, v skutočnosti sa v celom zreťazenom reťazci objaví nanajvýš $n$ rôznych trojíc.

Namiesto toho, aby sme pracovali s celou abecedou veľkosti $\sigma^3$, spravíme jednoduchú vec: všetky trojice, ktoré sa v texte objavia, *zotriedime* (ako?) a v tomto poradí im priradíme nové mená – čísla $0, 1, 2, \ldots$

Tým:

- zachováme správne poradie sufixov (triedenie trojíc je stabilné),
- veľkosť abecedy sa v každom kroku rekurzie zmenší na najviac $n$, takže pri rekurzii exponenciálne nevybuchne,
- dva „super-znaky“ budeme vedieť porovnať v konštantnom čase (iba porovnaním ich čísel),



# Kapitola 19

# FM-index

Túto časť sme začali so sufixovými stromami a ukázali si, ako vieme daný text predspracovať tak, že veľa iných problémov sa výrazne zjednoduší. Okrem iného vieme vyhľadávať vzorky v čase úmernom dĺžke hľadaného reťazca. Jediná nevýhoda sufixových stromov je, že žerú strašne veľa pamäte.

To nás priviedlo k hľadaniu úspornejších štruktúr. Sufixové polia predstavovali veľký praktický pokrok, pretože zaberajú podstatne menej miesta. My sa však nechceme uspokojiť len s tým, že štruktúra je 3- až 4-krát menšia. V duchu nášho hesla „Dá sa to *ešte* lepšie?“ pátrame ďalej.

Pokračujme v našom príklade s ľudským genómom. Ľudská DNA je reťazec dĺžky približne 3 miliardy znakov nad abecedou `A`, `C`, `G`, `T`. Samotný genóm teda zaberie asi 3 GB (ak použijeme 1 bajt na znak), alebo približne 750 MB, ak využijeme zhustenú reprezentáciu s 2 bitmi na znak.

Sufixový strom, aj keby sme sa snažili byť maximálne úsporní, potrebuje približne 10–20 bajtov na znak. Pre ľudský genóm by teda zaberal 30–60 GB pamäte. Sufixové pole je na tom podstatne lepšie: potrebujeme jedno číslo na každý znak. Ak máme text do 4 miliárd znakov, vystačíme si s 32-bitovým `int`om, čo sú 4 bajty na znak. Celé pole teda zaberie asi 12 GB (plus samotný reťazec 0.75 GB). Ak by sme pridali aj LCP pole, je to ďalších 8 bajtov (2 `int`y) na znak.

A to stále hovoríme len o výslednej štruktúre, nepočítajúc dočasnú pamäť počas konštrukcie. Pri spracovaní sa tak ľahko dostaneme na hranicu veľkosti RAM. Akonáhle sa RAM zaplní, operačný systém začne *page-ovať* – odkladať časti pamäte na disk. No a čítanie či zápis na disk je rádovo pomalšie než prístup do hlavnej pamäte.

V tejto kapitole si ukážeme, ako dosiahnuť pamäťovo *ešte úspornejšie* riešenie. A dokonca niečo až neuveriteľné: vstupný text môžeme *skomprimovať* a pritom stále umožniť rýchle vyhľadávanie!

Tieto dva ciele pôsobia na prvé počutie protichodne. Ak ste si niekedy otvorili skomprimovaný súbor, videli ste, že text je úplne nečitateľný. Ak poznáte niektoré kompresné algoritmy, viete, že využívajú opakovania v texte a nahrádzajú ich rôznymi „skratkami“ či odkazmi na iné časti. To však komplikuje vyhľadávanie.

Navyše sa často používajú kódovania, kde rôzne znaky zaberajú rôzne veľa bitov, takže sa posúvajú pozície a nie je jednoduché určiť, kde presne začína $i$-ty znak.

Pointou kompresie je totiž stlačiť vstup a vyjadriť tú istú informáciu čo najúspornejšie, zatiaľ čo pri indexovaní potrebujeme uložiť dáta štruktúrovane a pridať pomocné informácie, ktoré uľahčia vyhľadávanie. Aj na druhý pohľad to teda vyzerá ako nekompatibilné ciele.

Napriek tomu si ukážeme, že sa to dá. Predstavíme si štruktúru, ktorá už vo svojej najjednoduchšej podobe dokáže uchovať celý ľudský genóm na približne 1.5 GB pamäte. V ďalších častiach si potom ukážeme, ako jej jednotlivé komponenty reprezentovať ešte efektívnejšie.

| **Dátová štruktúra** | **Pamäť, ktorú zaberie ľudský genóm** |
|---|---|
| pôvodný reťazec | 3mld písmen, 0.75 GB (ale nevieme vyhľadávať) |
| sufixový strom | 30–60 GB [kap. 16] |
| sufixové pole | 12 GB [kap. 18] |
| FM-index | 1.5 GB [v tejto kapitole] |
| ... s kompresiou | $<$ 0.75 GB [kap. 22] |

Výsledná štruktúra, *FM-index*, je založená na tzv. *Burrows–Wheelerovej transformácii* (BWT). Ide o zaujímavú operáciu, ktorá sa využíva v rôznych kompresných algoritmoch, napríklad v známom `bzip2`. Transformácia sama osebe ešte text nekomprimuje, ale „premieša“ písmená tak, že výsledok sa dá oveľa ľahšie komprimovať. Dôležité je, že ide o *vratnú* operáciu – z transformovaného textu vieme pôvodný text jednoznačne zrekonštruovať (čo sa pri dekompresii celkom hodí).

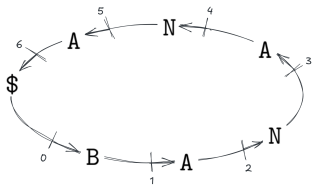
Na nasledujúcich stranách si postupne vysvetlíme:

1. čo je Burrows-Wheelerova transformácia,
2. ako sa využíva v kompresii a prečo je taká užitočná,
3. a ako dokážeme z transformovaného textu obnoviť pôvodný.

A až potom príde to najzaujímavejšie: ukážeme si, ako v takto transformovanom texte vieme vyhľadávať a ako celú dátovú štruktúru reprezentovať tak, aby bola nielen pamäťovo úsporná, ale aj rýchla.

## 19.1 Burrows-Wheelerova transformácia

Predstavme si, že písmená textu navlečieme ako korálky na retiazku; *$i$-ta rotácia* textu $T$ je reťazec, ktorý vznikne tak, že začneme čítať od $i$-teho znaku, prejdeme cez ukončovací symbol `$` a pokračujeme dookola, až kým sa nevrátime na začiatok. Formálne: $i$-ta rotácia je jednoducho reťazec $T_{i\ldots n-1}T_{0\ldots i-1}$.




Ako príklad vezmime reťazec `BANANA$` (tak ako v predchádzajúcich kapitolách, aj tu pridávame na koniec jedinečný ukončovací znak). Všetky jeho rotácie sú:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | B | A | N | A | N | A | $ |
| 1 | A | N | A | N | A | $ | B |
| 2 | N | A | N | A | $ | B | A |
| 3 | A | N | A | $ | B | A | N |
| 4 | N | A | $ | B | A | N | A |
| 5 | A | $ | B | A | N | A | N |
| 6 | $ | B | A | N | A | N | A |

Zotrieďme teraz všetky riadky lexikograficky (teda podľa abecedy):

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | $ | B | A | N | A | N | A |
| 5 | A | $ | B | A | N | A | N |
| 3 | A | N | A | $ | B | A | N |
| 1 | A | N | A | N | A | $ | B |
| 0 | B | A | N | A | N | A | $ |
| 4 | N | A | $ | B | A | N | A |
| 2 | N | A | N | A | $ | B | A |


Takto usporiadaná $n \times n$ matica všetkých rotácií sa nazýva *Burrows–Wheelerova matica (BWM)*.

*Burrows-Wheelerovu transformáciu* textu $T$ (budeme ju označovať $T^{\mathrm{bwt}}$) definujeme ako *posledný stĺpec tejto matice*. Konkrétne, pre $T = \texttt{BANANA\$}$ platí, že

$$T^{\mathrm{bwt}} = \texttt{ANNB\$AA}.$$

Samozrejme, v praxi nechceme zostavovať celú maticu – tá má kvadraticky veľa prvkov, čo by bolo pamäťovo neúnosné a výpočtovo pomalé. Otázka teda znie: ako vieme vypočítať $T^{\mathrm{bwt}}$ efektívne? Skúste porozmýšľať sami, so znalosťami s predchádzajúcich kapitol by to nemal byť problém.

Všimnite si, že BWT je vždy len *permutáciou* znakov pôvodného textu. Prečo?

Stačí sa pozrieť na pôvodnú maticu *pred zotriedením*. Posledný stĺpec bol `$BANANA`, čo je zjavne permutácia pôvodného reťazca. Matica je dokonca symetrická: $i$-ty stĺpec je rovnaký ako $i$-ty riadok, čo je $i$-ta rotácia textu. Keď riadky zotriedime, v každom stĺpci sa písmenká iba poprehadzujú a zloženie dvoch permutácií (rotácia plus triedenie) je opäť permutácia.

## Použitie v kompresii

Prečo je BWT taká cool? Ukážme si to na *trochu väčšom* príklade. A keď hovorím trochu väčšom, myslím naozaj veľkom: vezmime všetky diela Williama Shakespeara. Od Sonetov, cez Antónia a Kleopatru, Hamleta, Kráľa Leara, Macbetha, Rómea a Júliu, Veselé panie z Windsoru, až po báseň Venuša a Adonis. Pre jednoduchosť text znormalizujeme: odstránime interpunkciu, necháme iba malé písmená `a-z` a medzery. Dostaneme tak necelých 5 MB textu, takmer milión slov (z toho približne 27 500 rôznych).

Predstavme si teraz tento 5-miliónový reťazec a všetky jeho rotácie, zotriedené podľa abecedy. Napríklad:

> Miliónty riadok (404 643-ta rotácia) začína takto:
> `adam so had you need i scarce can speak to thank you`
> `for myself` ... [ďalších zopár miliónov znakov] ... a končí
> `orlando i thank you most for him`

(z komédie *Ako sa vám páči*).

Slávna Hamletova replika na písmeno „T“ je zhruba 4.5-miliónta v abecednom poradí:

> Riadok 4 537 084 (931 406-ta rotácia):
> `to be or not to be ..... enter hamlet hamlet`

Prvý riadok začína `$` a pôvodným textom. Druhý riadok je 2 846 296-ta rotácia a začína sa preklepom: `␣a␣a␣farm␣house`...

Posledný riadok sa začína písmenami `zzy`, ktoré pochádzajú zo stredu slova „dizzy“:

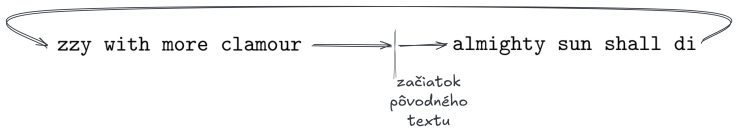
> Riadok 4 980 782 (4 340 678-ma rotácia):
> `zzy with more clamour neptunes ear ..... shall di`

z tragédie *Troilos a Kressida*.

A teraz tri hádanky:

1. Predstavte si riadky, ktoré začínajú písmenami `attle`. Aký bude posledný znak v takom riadku?
2. Aké budú posledné znaky v riadkoch, ktoré začínajú na `fe␣`?
3. Aké budú posledné znaky v riadkoch, ktoré začínajú na `a␣`?

Treba si uvedomiť, že pri rotáciách reťazca je na konci text, ktorý sa v pôvodnom tvare nachádza *tesne pred* textom na začiatku:




Takže aké písmeno môže byť pred `attle`?

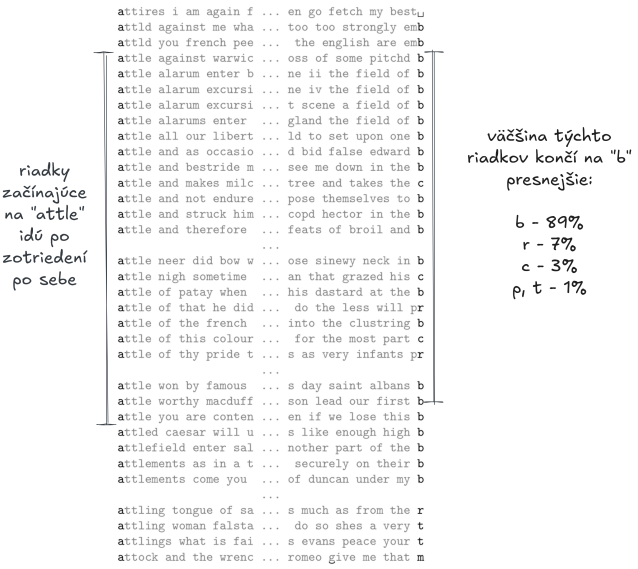
S najväčšou pravdepodobnosťou

- `b` (zo slov „battle“, „embattle“ – boj, zoradiť do boja), alebo
- `c` (cattle – dobytok).

Ak ste Shakespeare, alebo máte bohatšiu slovnú zásobu, sú aj ďalšie možnosti:

- `r` (zo slov „prattle“ – tárať, „rattle“ – hrkot/rinčanie, „berattle“ – urážať),
- `t` („tattle“ – klebetiť)
- `p` zo slova „pattle“? V skutočnosti má v hre *Henry V* kapitán Fluellen waleský prízvuk, čo Shakespeare zachytil foneticky vo vete „fought a most prave pattle here in France“.

Ak spočítame výskyty, reťazec `attle` sa v Shakespearovi nachádza 170-krát. Z toho 151-krát je predtým `b`, 12-krát `r` a 5-krát `c`; po jednom výskyte majú `p` a `t`.



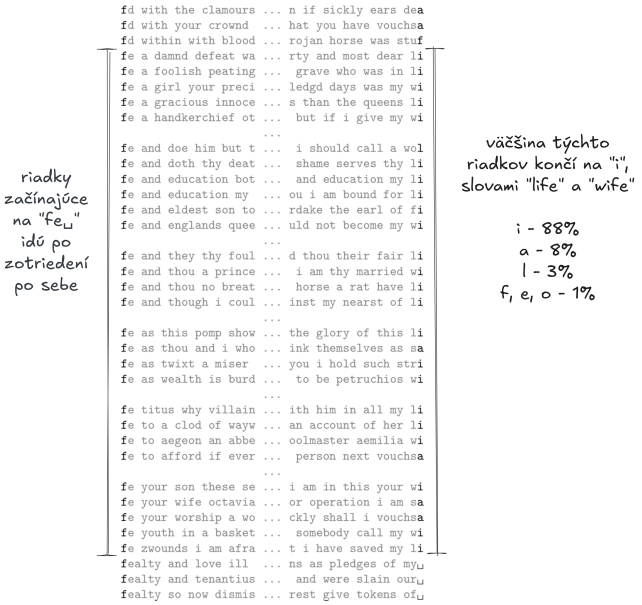

Obr. 19.1: Malý výsek Burrows-Wheelerovej matice zo Shakespearovského textu. V riadkoch začínajúcich na `attle` sa v poslednom stĺpci nachádza najmä `b`, zriedka `r`, `c`, `p`, `t` a nič iné.

A aké písmeno sa môže nachádzať pred `fe␣`?
Ako prvé nám asi napadne

- `i` zo slov „life“, „wife“, či „knife“... a Shakespeare k tomu ešte dodá „strife“, „Fife“ (miesto v Škótsku), „greife“ („grief“)
- `a` zo slov „safe“/„unsafe“, a Shakespeare pridá aj „vouchsafe“ – uráčiť, „chafe“ – trieť/dráždiť

V Shakespearovskej angličtine so starým pravopisom pribudne ešte veľa ďalších slov ako „deafe“, „selfe“, „halfe“, „greefe“, „roofe“, „proofe“, „staffe“, „theefe“, „wiffe“, „wolfe“, atď.

Napriek tomu, z 1842 výskytov reťazca `fe␣` je 1615-krát predchádzajúce `i` a 152-krát `a`. Iba 51-krát `l`, 10-krát `f`, 6-krát `e` a 4-krát `o`. Ďalšie možnosti sa nevyskytujú.




Obr. 19.2: Ďalší výsek Burrows-Wheelerovej matice Shakespearovského textu, riadky začínajúce na `fe␣`.

A posledná hádanka: aký znak bude predchádzať `a␣`?

Toto je možno trochu chyták. Najčastejšie viacpísmenové slovo končiace na „a“ je „sea“ – more (286 výskytov). Ďalej sú to citoslovcia ako „ha“ (233) alebo „yea“ (200), a, samozrejme, aj množstvo ženských mien (počet výskytov závisí najmä od mien hlavných postáv v jednotlivých hrách – nie nutne od frekvencie mien v bežnej populácii): Cleopatra (264 výskytov), Emilia (233), Desdemona (226), Helena (194), Portia (175), Isabella (159), Olivia (147), Julia (145), Viola (142), a mnoho ďalších až po Angelica, Julietta, či Polyxena (po jednom výskyte).

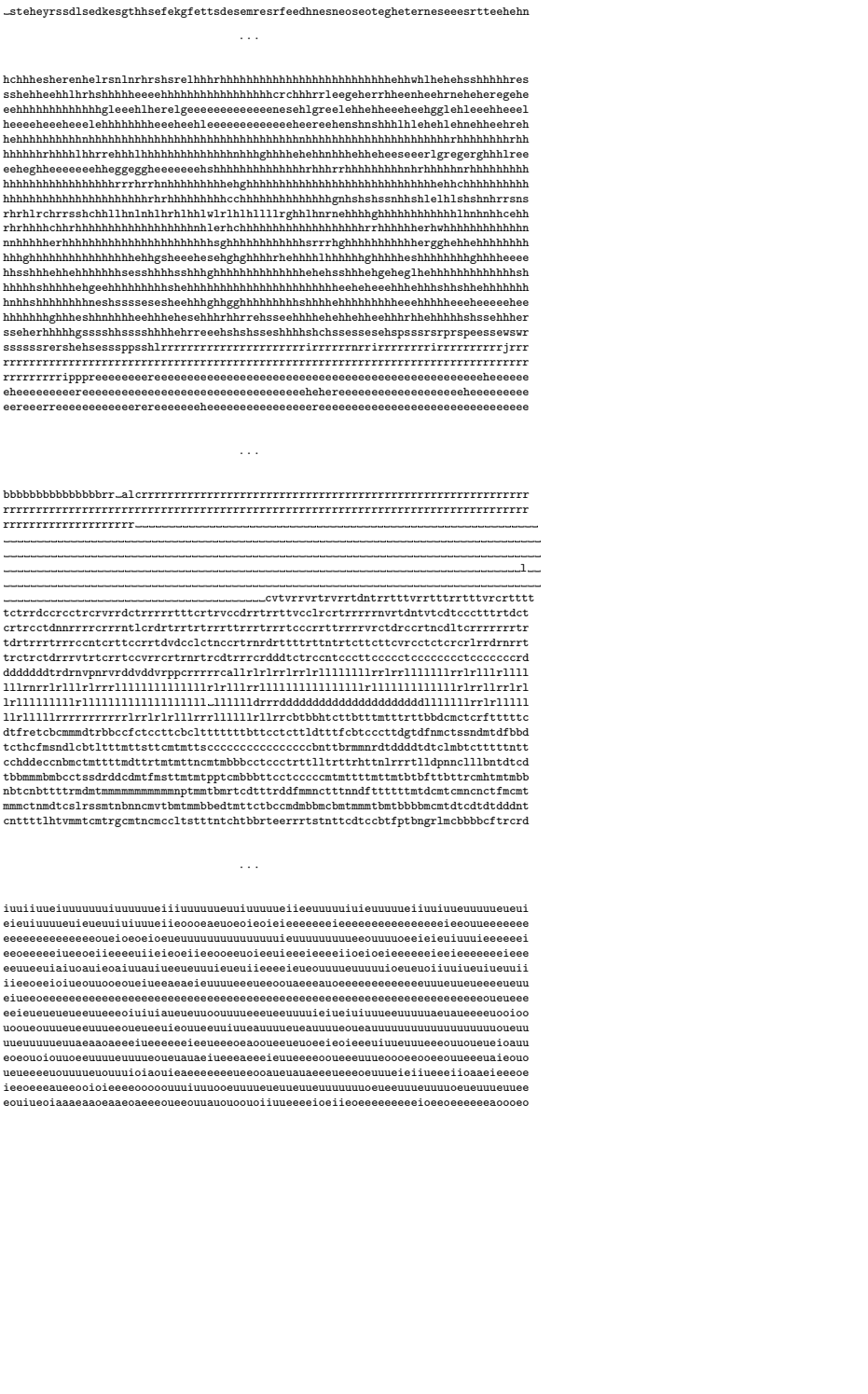
Jednoznačne najčastejšie slovo, ktoré končí na `a`, je však samotný neurčitý člen „a“. To znamená, že posledným znakom v riadku začínajúcom na `a␣` je *medzera*. V skutočnosti je to až 16 089 riadkov z 22 151 (72%).

Tým, že riadky zotriedime, zabezpečíme, že rotácie začínajúce rovnakými písmenami budú *vedľa seba*.

A teraz pointa. Ako bude vyzerať BWT Shakespearovho textu? Označme ho $SH^{\mathrm{bwt}}$.

Burrows-Wheelerova transformácia je definovaná ako posledný stĺpec Burrows-Wheelerovej matice, teda posledné písmeno z každého riadku (z každej rotácie), ktoré sme utriedili podľa abecedy. Tým, že riadky zotriedime, dosiahneme, že rotácie začínajúce rovnakými písmenami, sa ocitnú *vedľa seba.*

$SH^{\mathrm{bwt}}$ je takmer 5-miliónový reťazec, ktorý je permutáciou pôvodného textu. Pre lepšiu predstavu, tu je zopár úryvkov z BTW Shakespearovho textu:



```
eiuuueueioeoouoeeeeuueeuoouieoooouoieueueeeeuoeeuaioiiieaoauuuiuoouiioeaiiieouooeoo
ueeoeeueueueoeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeieuaeeeuoeuoaeeueoueoooouuoeoooueeeooo
eoeuuueeueeueeeeuueiueouueeeeeeueeuueeuuueeeeueoouеuueeoueueieuouueuiuiuieiuaeeeueeu
eeaeeouiueeioooioouuiuiuiuuuuuauiieeiuoeeeeeoueuuioeeeeaeieoeueeeeeeueeoeeeaeieoeu
euuuuuouuuuueoueuueuuuuuueueuuuuuuuuuuuueuuuuuueuuiiuuuuuuuueuiueiuuuiiuiiuuei

...

nnazzanuuuuuuuuueauuuuuuuuiuuuuuauuiauauiuauiiaiiuuaiuauaaiiii
```

Začiatok $SH^{\text{bwt}}$, `␣steheyrssdlsedk`..., síce vyzerá náhodne – prvé riadky totiž začínajú medzerou, takže posledný znak v riadku je posledné písmeno nejakého slova. Keď sa však dostaneme k riadkom začínajúcim na `a␣`, drvivá väčšina bude končiť medzerou, a v $SH^{\text{bwt}}$ tak vznikne dlhý úsek (22 151 znakov), z ktorých 72% sú medzery. Keď sa dostaneme k riadkom začínajúcim na `attle`, v poslednom stĺpci budú samé `b`-čka (občas `r`, `c`, `p`, `t` a žiadne iné písmená). Riadky začínajúce na `fe␣` zas vytvoria dlhý blok, v ktorom budú najmä `i`-čka, sem-tam `a` alebo `l` a úplne zriedkavo iné tri písmená, ale nič viac.

Riadky, ktoré začínajú slovom „Romeo“, budú končiť *vždy* medzerou, pretože žiadne iné slovo neobsahuje podreťazec `romeo`. V $SH^{\text{bwt}}$ to vytvorí súvislý úsek 317 po sebe idúcich medzier.

Vo všeobecnosti bude $SH^{\text{bwt}}$ obsahovať úseky s opakovanými písmenami a ešte všeobecnejšie, dlhšie úseky, v ktorých sa vyskytuje len veľmi málo rôznych znakov. Tento jav sa objavuje nielen pri Shakespearovi, ale aj pri iných textoch: v prirodzených i umelých jazykoch, pri biologických sekvenciách plných opakovaní a podobne.

Ide o to, že znak na danej pozícii sa často dá ľahko uhádnuť z kontextu, teda z toho, čo nasleduje za ním. Keď rotácie zotriedime, podobné kontexty sa dostanú vedľa seba. Výsledkom sú dlhé bloky, kde sa buď opakuje ten istý znak, alebo sa strieda veľmi malá abeceda.

Takýto text sa komprimuje oveľa ľahšie než pôvodný reťazec. Ako na to?

Najjednoduchší nápad je použiť tzv. run-length encoding (RLE). Myšlienka je priamočiara: ak sa v texte $n$-krát po sebe opakuje ten istý znak $c$ (povedzme pre $n \geq 3$), tak reťazec

$$\underbrace{ccc\ldots ccc}_{n} \quad \text{zakódujeme ako dvojicu} \quad (n, c).$$

Aby sme pri dekompresii vedeli správne interpretovať bajty, musíme si niekam poznačiť, či daný bajt reprezentuje znak alebo dĺžku behu. Jednoduchý spôsob je využiť najvyšší (prvý) bit každého bajtu: ak je nastavený na `0`, zvyšných 7 bitov kóduje znak, ak je nastavený na `1`, zvyšných 7 bitov udáva počet opakovaní.

Už pri takomto jednoduchom postupe vieme Shakespearov text skomprimovať na takmer polovičnú veľkosť. Tento výsledok je fascinujúci najmä preto, že sme ešte ani nepoužili sofistikovanejšie techniky kompresie – iba sme využili veľmi „výhodné“ usporiadanie znakov, ktoré vyprodukovala Burrows-Wheelerova transformácia.

Ďalšia možnosť, ako využiť štruktúru, ktorú BWT odhalí, je použiť kódovanie *s premenlivou dĺžkou kódov*. Čím častejšie sa znak vyskytuje, tým kratší kód by

mal mať, a naopak – zriedkavé znaky môžu mať dlhšie kódy. Týmto spôsobom sa dosiahne, že priemerný počet bitov na znak bude menší, než keby sme každému znaku priradili kód rovnakej dĺžky. Príklady takéhoto prístupu sú Huffmanovo kódovanie, aritmetické kódovanie, alebo modernejšie ANS (asymetrické numerické systémy).

Ak sa pozrieme na normalizovaný text všetkých Shakespearových diel, najčastejšie znaky sú: medzera ␣ (19.3%), `e` (9.6%), `t` (7.1%), `o` (6.6%), `a` (6.2%), `i` (5.0%). Tieto by sme teda mali kódovať čo najkratšími kódmi. Naopak, znaky ako `z`, `q`, `j`, či `x`, ktoré sa vyskytujú veľmi zriedka, môžu dostať dlhšie kódy.

Tento klasický prístup však vieme v kombinácii s BWT ešte vylepšiť. Po prvé, v transformovanom texte $T^{\text{bwt}}$ jednotlivé úseky zodpovedajú konkrétnym *kontextom*, v ktorých sa znaky vyskytujú. Čím lepšie poznáme tento kontext, tým presnejšie dokážeme odhadnúť pravdepodobnosť jednotlivých znakov – podobne ako keď sme hádali, ktoré písmeno sa objaví pred daným podreťazcom. Po druhé, rôzne úseky transformovaného textu $T^{\text{bwt}}$ zodpovedajú *odlišným kontextom*, a teda aj rozdelenie frekvencií znakov sa v nich môže zásadne líšiť.

Napríklad v riadkoch, ktoré začínajú medzerou, posledný znak určite nebude medzera. A hoci písmená `o`, `a`, `i` patria medzi najčastejšie v angličtine, *nepatria* medzi tie, ktoré sa najčastejšie objavujú na konci slova. Na konci slov sa omnoho častejšie vyskytujú písmená `e`, `s`, `t`, `d`, `r`.

Ako túto informáciu využiť?

Jedna možnosť je rozdeliť text na bloky a každý z nich zakódovať samostatným Huffmanovým kódom. Ak majú jednotlivé bloky rôzne frekvencie písmen, môžeme im priradiť aj odlišné kódy, čím dosiahneme lepšiu kompresiu.

Ďalšou možnosťou je použiť *adaptívne* Huffmanovo kódovanie (pozri Vitter 1987). Hlavná myšlienka je jednoduchá: pri postupnom spracúvaní textu priebežne počítame frekvencie znakov (buď od úplného začiatku, alebo len v rámci okna posledných $w$ znakov). Podľa týchto frekvencií, ktoré sa neustále menia, si dynamicky udržiavame Huffmanov kód, ktorý sa prispôsobuje aktuálnym frekvenciám znakov.

Algoritmus `bzip2` (Seward 1996) používa ešte iný trik: na $T^{\text{bwt}}$ aplikuje tzv. Move-to-front (MTF) transformáciu. Myšlienka je jednoduchá: počas kódovania si udržiavame zoznam všetkých znakov a $i$-ty znak kódujeme číslom $i$. Zároveň vždy keď zakódujeme nejaký znak, presunieme ho na začiatok zoznamu („move to front“).

Napríklad ak kódujeme reťazec

```
D A D D D D C D C D D B D D D
```

a začíname so zoznamom `␣ A B C D E ..`, postup vyzerá takto:

- písmeno `D` je štvrté (počítame od 0), takže ho zakódujeme ako `4` a presunieme na začiatok zoznamu: `D ␣ A B C E ..`,
- následne `A` zakódujeme ako `2` a presunieme na začiatok: `A D ␣ B C E ..`,
- ďalšie `D` zakódujeme ako `1` a `D` sa dosane opäť na začiatok zoznamu: `D A ␣ B C E ..`,

- nasledujúce tri `D`-čka zakódujeme ako `0`, lebo `D` je na začiatku,
- atď.

Výsledná zakódovaná postupnosť je:

`4 2 1 0 0 0 4 1 1 1 0 4 1 0 0.`

Dôležité vlastnosti MTF transformácie sú tieto: Ak sa v texte $n$-krát po sebe opakuje to isté písmeno, po aplikovaní MTF sa zakóduje podľa aktuálnej pozície a následne $n-1$ núl. Podstatné je, že v rôznych častiach textu sa môžu opakovať rôzne písmená, no po MTF budú tieto behy vždy vyzerať ako postupnosť núl.

Všeobecnejšie: ak nejaký úsek obsahuje iba $r$ rôznych znakov, ich prvé výskyty zakódujeme možno väčšími číslami, no potom sa hneď presunú na začiatok zoznamu. Zvyšok úseku potom kódujeme len číslami $0, 1, 2, \ldots, r-1$. Takže ak sa v rôznych častiach textu vyskytujú malé podmnožiny znakov, po MTF sa tam budú objavovať takmer výlučne malé čísla $0, 1, 2, \ldots$.

MTF je teda elegantný spôsob, ako všetky dlhé úseky s rôznymi malými abecedami namapovať na dlhé úseky tvorené malými číslami $0, 1, 2, \ldots$. Takto spracovaný text sa následne výborne hodí na Huffmanovo kódovanie.

V skutočnosti `bzip2` pozostáva zo štyroch krokov:

1. Burrows-Wheelerova transformácia usporiada text tak, že vzniknú dlhé úseky s malou abecedou,
2. Move-to-front transformácia tieto úseky ďalej prevedie na postupnosti prevažne veľmi malých čísel, najčastejšie núl,
3. Run-length encoding skomprimuje jednopísmenové behy do tvaru (počet, znak), a nakoniec
4. Huffmanovo kódovanie zakóduje výslednú postupnosť pomocou prefixového kódu, pričom častejšie symboly dostanú kratšie kódy.

## Inverzná transformácia

Začnime jednoduchým príkladom: vezmime slovo „bedač“, kde sú všetky písmená rôzne. Ak vytvoríme všetky rotácie reťazca `BEDAC$` a zotriedime ich lexikograficky, dostaneme maticu, ktorej posledný stĺpec je $T^{\mathrm{bwt}} =$ `CD$AEB`.

```
5  $ B E D A C
3  A C $ B E D
0  B E D A C $
4  C $ B E D A
2  D A C $ B E
1  E D A C $ B
```

Otázka znie: ako sa vieme od `CD$AEB` dostať naspäť k pôvodnému textu? Nápoveda: predstavme si Burrows-Wheelerovu maticu. $T^{\mathrm{bwt}}$ je jej posledný

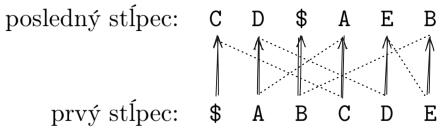
stĺpec. Okrem neho existuje ešte jeden špeciálny stĺpec, ktorý dokážeme z $T^{\mathrm{bwt}}$ veľmi ľahko zrekonštruovať. Ktorý?

Ak hovoríte, že ten špeciálny stĺpec je prvý,máte pravdu. Ako sme spomínali vyššie, každý stĺpec Burrows-Wheelerovej matice je permutáciou znakov pôvodného textu. Špeciálne prvý stĺpec vznikne tak, že sa znaky utriedia podľa abecedy, pretože celé riadky triedime lexikograficky.

To znamená, že poznáme

| | | | | | | |
|---|---|---|---|---|---|---|
| posledný stĺpec: | C | D | $ | A | E | B |
| prvý stĺpec: | $ | A | B | C | D | E |

Čo ďalej? Uvedomme si, že znak v poslednom stĺpci je vždy ten, ktorý sa v príslušnom riadku nachádza *hneď pred* znakom v prvom stĺpci.




Ak pôjdeme odzadu, vieme, že posledný znak pôvodného textu je `$`. Podľa tabuľky sa pred ním nachádza `C`. Pred `C`-čkom je zase `A`, a tak ďalej. Takto postupujeme, až kým neprídeme späť na začiatok textu – pred `B` je znovu `$`, čo nám signalizuje, že máme končiť.

Jediné, čo si potrebujeme rozmyslieť, je: ako pre daný znak v poslednom stĺpci nájdeme jeho pozíciu v prvom stĺpci. (Mimochodom, proces by sa dal robiť aj opačne a dekódovať pôvodný reťazec od začiatku do konca, ale mapovanie prvého stĺpca na posledný by sa robilo trochu komplikovanejšie.)

A čo v prípade opakujúcich sa znakov?

Vezmime si úvodný príklad, reťazec `BANANA$`:

| | | | | | | |
|---|---|---|---|---|---|---|
| $_0$ | B | A | N | A | N | A$_0$ |
| A$_0$ | $ | B | A | N | A | N$_0$ |
| A$_1$ | N | A | $ | B | A | N$_1$ |
| A$_2$ | N | A | N | A | $ | B$_0$ |
| B$_0$ | A | N | A | N | A | $_0$ |
| N$_0$ | A | $ | B | A | N | A$_1$ |
| N$_1$ | A | N | A | $ | B | A$_2$ |

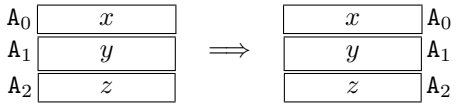
Tvrdím, že ak si jednotlivé písmená označíme poradovými číslami, aby sme ich vedeli od seba odlíšiť, tak všetky výskyty rovnakého písmena v prvom stĺpci *budú zoradené v rovnakom poradí* ako tie isté písmená v poslednom stĺpci. Napríklad všetky `A`-čka v prvom stĺpci budú zoradené v tom istom poradí ako `A`-čka v poslednom stĺpci.

Prečo je to tak?

Zamerajme sa iba na riadky Burrows–Wheelerovej matice, ktoré začínajú písmenom `A`. Keď tieto riadky zotriedime, dostanú sa vedľa seba. O ich vzájomnom

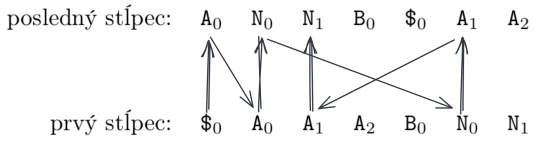
poradí však rozhoduje zvyšok riadku, teda text, ktorý nasleduje *po* úvodnom A-čku. Ak máme riadky A$x$, A$y$, A$z$, ich poradie bude dané poradím reťazcov $x, y, z$.

Čo sa stane s riadkami, ktoré končia na A? Tie presne zodpovedajú predchádzajúcej skupine riadkov, len posunutej o jednu pozíciu doľava: z A$x$, A$y$, A$z$ sa stanú $x$A, $y$A, $z$A. Tieto riadky síce v utriedenej matici nemusia stáť vedľa seba, no ich vzájomné poradie je opäť určené poradím $x, y, z$.




Výsledkom je, že poradie všetkých A-čiek v prvom stĺpci presne zodpovedá poradiu všetkých A-čiek v poslednom stĺpci. A rovnako to platí aj pre všetky ostatné písmená.

Vďaka tomu presne vieme, ktorý znak v poslednom stĺpci zodpovedá ktorému znaku v prvom stĺpci, a môžeme použiť ten istý postup ako predtým: postupne sa vracať po znakoch smerom späť a zrekonštruovať pôvodný text.




## 19.2 FM-index

Keď už vieme, ako funguje Burrows-Wheelerova transformácia a jej použitie pri kompresii, poďme sa pozrieť na to, ako sa dá v transformovanom texte $T^{\text{bwt}}$ vyhľadávať.

### Vyhľadávanie

Ukážme si postup na konkrétnom príklade: Máme text

$$T = \texttt{ATAGACCGCCATTACATAGATGAGTATAGAGACT\$},$$

spočítame si jeho Burrows-Wheelerovu transformáciu

$$T^{\text{bwt}} = \texttt{TTGGTGTTG\$TCGCACGACAAAATACACTAAAGAA}.$$

Celá Burrows-Wheelerova matica je na obr. 19.3 – pripomeňme, že ju tu uvádzame len na ilustráciu – v praxi ju nikdy celú nezostrome. Zaujíma nás z nej len prvý stĺpec $F$ a posledný stĺpec $L = T^{\text{bwt}}$, zvyšné znaky matice nemáme k dispozícii, preto sú znázornené šedou.

V skutočnosti aj prvý stĺpec

$$F = \texttt{\$AAAAAAAAAAAAAACCCCCCGGGGGGGTTTTTTTT}$$

nemáme uložený ako $n$ znakov – je to zbytočne veľa pamäte. Namiesto toho si stačí pamätať počty znakov: 1 ukončovací znak, 13 `A`-čok, 6 `C`-čok, 7 `G`-čok a 8 `T`-čok. Ako uvidíme ochvíľu, ešte lepšie uložiť si tabuľku začiatkov:

| *znaky* | `$` | `A` | `C` | `G` | `T` | koniec |
|---|---|---|---|---|---|---|
| *začínajú od riadku* | 0 | 1 | 14 | 20 | 27 | 35 |

Na obr. 19.3 sme si tiež očíslovali výskyty jednotlivých znakov, podobne ako pri inverznej BWT, aby sme vedeli, ktorý znak v poslednom stĺpci zodpovedá ktorému znaku v prvom stĺpci. Napríklad `A 3` znamená, že ide o tretie `A`-čko v danom stĺpci (číslujeme od 0), resp., že pred týmto `A` sú v texte ešte ďalšie tri.

Predstavme si, že chceme vyhľadať všetky výskyty reťazca `TAG`. Postup je tiež znázornený na obr. 19.3. Budeme postupovať odzadu a hľadať čoraz dlhšie sufixy hľadaného reťazca. Najskôr nájdeme všetky riadky, ktoré začínajú posledným písmenom `G`, potom nájdeme riadky, ktoré začínajú na `AG`, až nakoniec nájdeme všetky riadky, ktoré začínajú na `TAG`.

Riadky začínajúce na `G` vieme odčítať priamo z tabuľky: sú riadky 20–26.

Ak sa pozrieme na posledný stĺpec v týchto riadkoch, vidíme, že väčšina končí na `A`, niektoré na `C` a `T`. Nás samozrejme zaujímajú práve riadky, kde predchádzajúce písmeno pred `G` je `A`, teda riadky, ktoré majú v prvom stĺpci `G` a v poslednom stĺpci `A`.

---

Jeden krok vyhľadávacieho algoritmu. Predstavme si, že už sme našli interval riadkov, ktoré začínajú na podreťazec $P$ (vyšrafovaná oblasť v strede). To znamená, že riadky nad tým začínajú reťazcom menším ako $P$ a riadky pod tým začínajú reťazcom väčším ako $P$. Predpokladajme ďalej, že chceme nájsť riadky, ktoré začínajú na `c`$p$, kde `c` je jeden znak.

Pozrime sa na riadky, ktoré *končia na* `c`. Riadky končiace nultým a prvým `c` sú *nad* naším intervalom. To znamená, že v rotáciach za $\texttt{c}_0$ a $\texttt{c}_1$ nasleduje reťazec menší ako $p$. Podobne riadky končiace piatym a šiestym `c` sú *pod* naším intervalom, takže za $\texttt{c}_5$ a $\texttt{c}_6$ nasleduje reťazec väčší ako $p$.

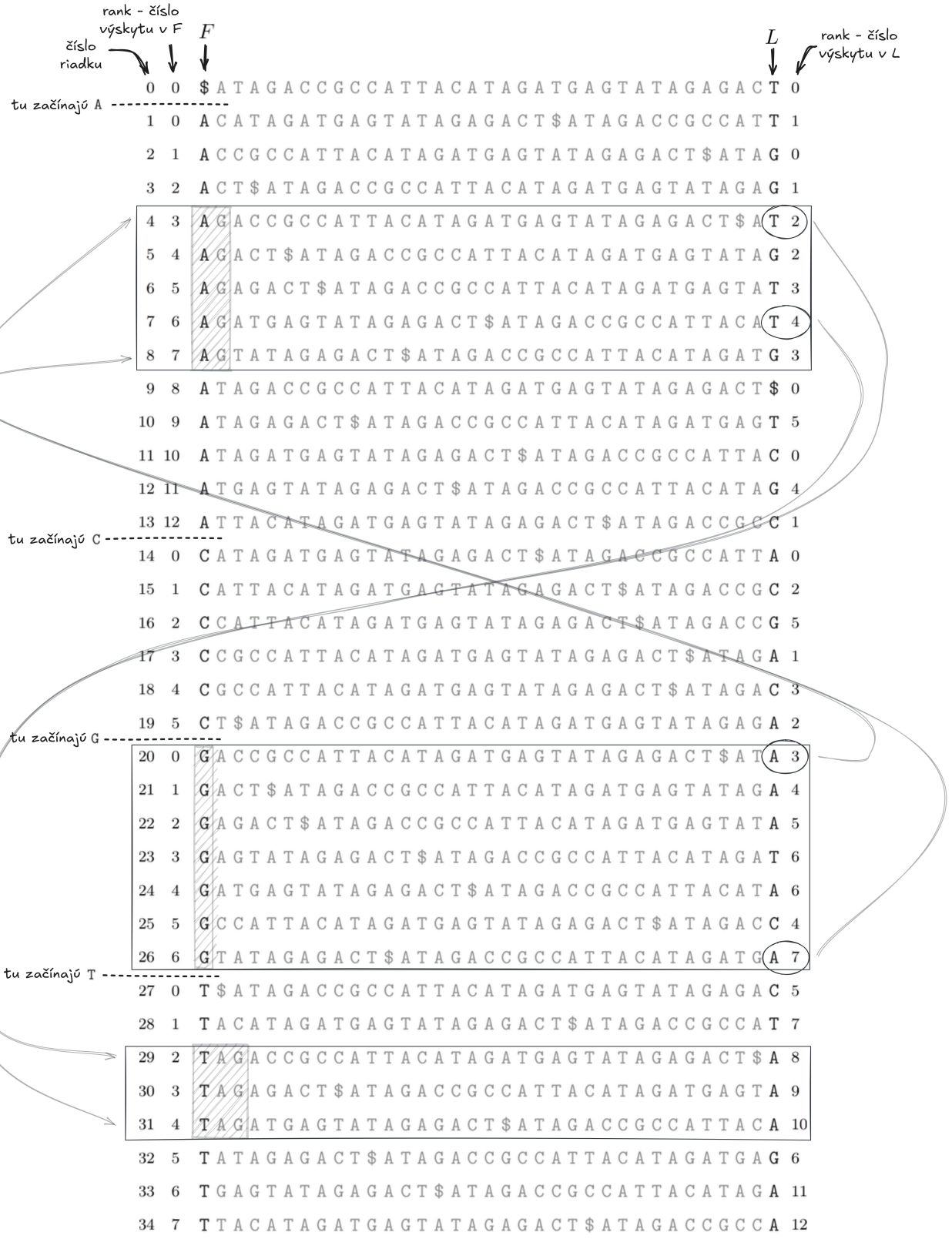
Riadky končiace druhým, tretím a štvrtým `c` sa nachádzajú v našom intervale, tzn. začínajú na $p$. To znamená, že v rotáciách za $\texttt{c}_2$, $\texttt{c}_3$ a $\texttt{c}_4$ nasleduje $p$ a interval od druhého po štvrté `c` je náš hľadaný interval.

## Ranky

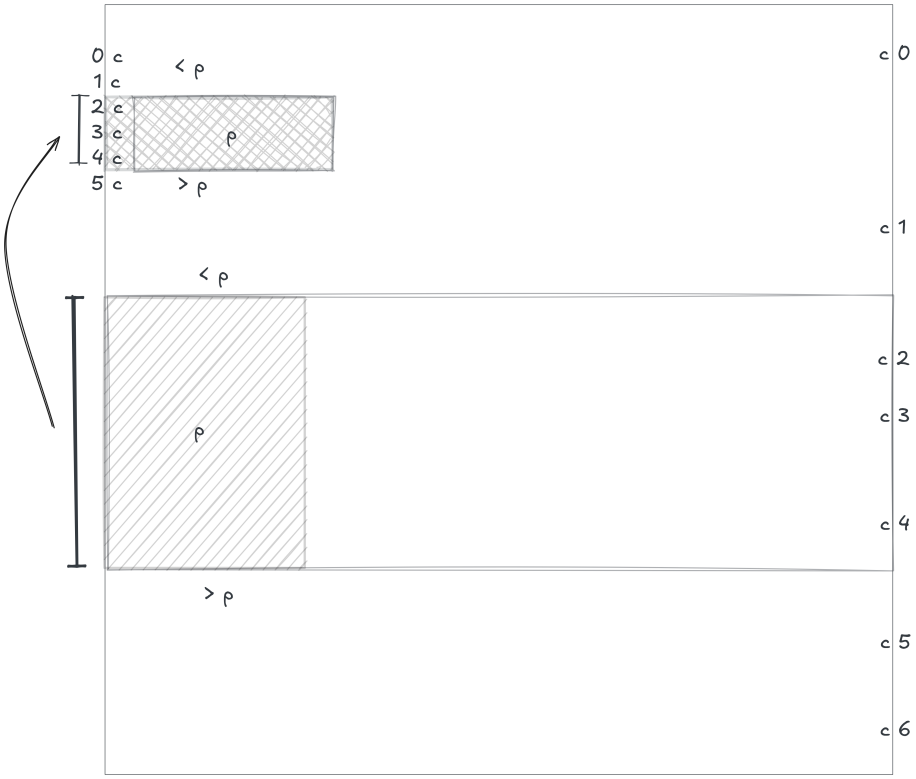
## Pozície v texte

## Výsledná štruktúra

Na prípravu jedného FM-indexu budeme potrebovať: vstupný text $T$. Spočítame sufixové pole $SA(T)$. Zo sufixového poľa odvodíme posledný stĺpec Burrows-Wheelerovej matice $L$, ktorý si uložíme. Zo samotného sufixového poľa si necháme iba malú podmnožinu a zvyšok zahodíme. Spočítame frekvencie jednotlivých znakov a pole čiastočných súčtov, čím dostaneme tabuľku $F$ pre prvý stĺpec.




Obr. 19.3: BWT textu $T =$ `ATAGACCGCCATTACATAGATGAGTATAGAGACT$`.
Prvý stĺpec: $F =$ `$AAAAAAAAAAAAAACCCCCCGGGGGGGTTTTTTTT`,
Posledný stĺpec: $L =$ `TTGGTGTTG$TCGCACGACAAAATACACTAAAGAA`.




Obr. 19.4: Jeden krok vyhľadávacieho algoritmu. Predstavme si, že už sme našli interval riadkov, ktoré začínajú na podreťazec $P$ (vyšrafovaná oblasť v strede) a chceme nájsť riadky, ktoré začínajú na c$p$, kde c je jeden znak. Potrebujeme zistiť, *ktoré* c v poslednom stĺpci sa nachádzajú v našom intervale. V tomto príklade sú to $\mathtt{c}_2$ až $\mathtt{c}_4$. Preto interval od druhého po štvrté c je hľadaný interval, ktorý začína na c$p$.

Následne vybudujeme štruktúru pre $\text{rank}_{\mathtt{c}}(L, i)$, vďaka ktorej vieme rýchlo povedať pre ľubovoľný znak $\mathtt{c}$ a pozíciu $i$, koľko $\mathtt{c}$-čiek sa nachádza v $L$ pred $i$.

Výsledný FM-index sa skladá zo štyroch častí:

- $L = T^{\text{bwt}}$, posledného stĺpca Burrows-Wheelerovej matice,
- $F$, prvého stĺpca,
- štruktúry pre $\text{rank}_{\mathtt{c}}(L, i)$,
- podmnožiny sufixového poľa $SA(T)$.

Koľko miesta to zaberie?

- $L = T^{\text{bwt}}$ – $n$ znakov, rovnako ako vstupný text (zatiaľ – avšak už sme videli, že $L$ je reťazec, ktorý sa dá dobre komprimovať)
- $F$ – jedno číslo pre každý znak abecedy; abeceda je väčšinou malá (napr. menej ako 256 znakov), takže toto je väčšinou úplne zanedbateľná hodnota,
- $\text{rank}_{\mathtt{c}}(L, i)$ – zatiaľ sme si ukázali riešenie s pamäťou $n|\Sigma|/b$, kde $b$ je konštanta, v nasledujúcej kapitole si ukážeme lepšie riešenia,
- podmnožina sufixového poľa – $2n/s$ čísel, kde $s$ je nejaká konštanta.

Ukážme si to na príklade ľudskej DNA z úvodu. Máme teda 4-písmenovú abecedu a zvoľme $s = 64$ a $b = 128$. Potom

- $L$ zaberie 750 MB (ako pôvodný reťazec),
- $F$ sú 4 čísla (zanedbateľných 16 bajtov).

3 miliardy 4-bajtových čísiel zaberie 12 GB, tzn.

- štruktúra pre rank zaberie $12\,\text{GB} \times 4/128 = 375\,\text{MB}$,
- $1/64$ sufixového poľa zaberie $12\,\text{GB} \times 2/64 = 375\,\text{MB}$

Podtrženo a sečteno: spolu len 1.5 GB, čiže dvakrát veľkosť pôvodného reťazca. Všimnite si, že na rozdiel od sufixových stromov a polí si nepotrebujeme pamätať pôvodný reťazec $T$ (vieme ho rekonštruovať z $L = T^{\text{bwt}}$).

Z 30–60 GB sufixového stromu, cez 12 GB sufixové pole sme sa teda dostali na 1.5 GB FM-index. Dosiahli sme 20–40× menej pamäte a to sme ešte nekomprimovali $L$ a použili sme len veľmi jednoduché riešenie pre rank. S kompresiou a ďalšími vylepšeniami vieme dosiahnuť dátovú štruktúru, ktorá zaberá len 30–50% pamäte pôvodného reťazca a navyše v nej vieme rýchlo vyhľadávať. Ako na to, si postupne ukážeme v ďalších kapitolách.

# Referencie

Seward, Julian (1996). *bzip2 and libbzip2*. URL: `https://sourceware.org/bzip2/manual/manual.html`.

Vitter, Jeffrey Scott (1987). „Design and analysis of dynamic Huffman codes“. In: *Journal of the ACM (JACM)* 34.4, s. 825–845.

# Časť VII

# Úsporné dátové štruktúry

# Kapitola 20

# Rank a select

V tejto kapitole si vybudujeme jeden z najzákladnejších stavebných kameňov pre úsporné dátové štruktúry: *bitvektor* s operáciami `access`, `rank` a `select`.

Majme bitvektor $B \in \mathtt{0},\mathtt{1}^n$. Na ňom budeme definovať tri základné operácie:

- access($i$) (prístup) vráti bit $B[i]$.
- $\mathrm{rank}_{\mathtt{1}}(i)$ vráti počet jednotiek v prefixe $B[0 \ldots i)$ pred pozíciou $i$. Analogicky $\mathrm{rank}_{\mathtt{0}}(i) = i - \mathrm{rank}_{\mathtt{1}}(i)$ udáva počet núl.
- $\mathrm{select}_{\mathtt{1}}(k)$ vráti najmenšie $i$, pre ktoré $\mathrm{rank}_{\mathtt{1}}(i+1) = k$, t. j. pozíciu $k$-tej jednotky. Analogicky definujeme $\mathrm{select}_{\mathtt{0}}$ pre nuly.[1]

Na prvý pohľad to nevyzerá ako nič zvláštne – prístup do poľa, počet jednotiek, pozícia jednotky. Lenže cieľom bude navrhnúť tieto operácie tak, aby boli čo najrýchlejšie (v konštantnom čase) a zároveň zaberali *čo najmenej pamäte*.

**Motivácia #1: FM-index.** V kapitole o FM-indexe sme narazili na potrebu rýchlo spočítať, koľkokrát sa daný znak `c` vyskytuje v prefixe $L[0 \ldots i)$. Pre binárnu abecedu je táto úloha presne operácia rank nad bitvektorom. Keď tento prípad zvládneme rýchlo a pamäťovo úsporne, v ďalšej kapitole sa pozrieme na väčšie abecedy.

**Motivácia #2: Zhustené pole.** Predstavme si pole $A$ dĺžky $m$, ktoré obsahuje len $n < m$ reálnych prvkov a zvyšok je prázdny. Každý prvok zaberá $\ell$ bitov a predpokladajme, že v týchto $\ell$ bitoch vieme reprezentovať aj špeciálnu hodnotu „prázdny“ (napríklad `null`). Ako môžeme takéto riedke pole reprezentovať úsporne a pritom efektívne?

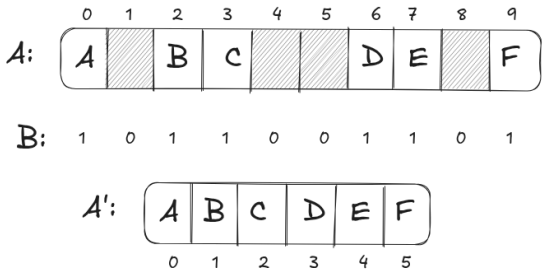
Jednoduché riešenie: obyčajné pole. Najpriamejší prístup je alokovať $m \times \ell$ bitov, teda klasické pole dĺžky $m$. Nevýhoda je zrejmá – ak je pole riedke,

[1] Pozor na off-by-one chyby: v celej knihe budeme dôsledne používať nulové indexovanie a intervaly tvaru $[0 \ldots i)$, takže rank($i$) sa vždy počíta len po pozíciu $i$, nie vrátane nej

väčšina priestoru sa spotrebuje na prázdne políčka. Na druhej strane, ak je $n$ len o trochu menšie ako $m$, teda prázdnych miest je málo, je to dobré riešenie.

Druhá možnosť je uložiť si iba tie prvky, ktoré skutočne existujú, ako slovník $i \mapsto A_i$. Takýto slovník môžeme implementovať buď ako utriedené pole (v ktorom budeme binárne vyhľadávať), alebo ako hešovaciu tabuľku. Takéto riešenie zaberá približne $n \times (\lg m + \ell)$ bitov, v prípade heštabuľky ešte krát $(1 + \varepsilon)$. Toto je výborné riešenie, ak je pole veľmi riedke, teda $n \ll m$.

Keď si o chvíľu ukážeme úspornú implementáciu pre operácie `rank` a `select`, pribudne nám ešte tretia možnosť: Vytvoríme si bitvektor $B$ dĺžky $m$, v ktorom si pre každú pozíciu uložíme informáciu, či je dané políčko obsadené (`1`) alebo prázdne (`0`). Zároveň si alokujeme pole $A'$ dĺžky $n$, do ktorého zapíšeme všetky neprázdne prvky $A_i$, natlačené tesne vedľa seba, bez medzier.




Ak potom chceme zistiť, či je $A[i]$ obsadené, jednoducho sa pozrieme na bit $B[i]$. Ak $B[i] = \texttt{1}$, vieme pomocou operácie $\mathrm{rank}_1(i)$ zistiť, koľko neprázdnych prvkov sa nachádza pred pozíciou $i$, a teda aj index v poli $A'$, kde sa nachádza požadovaná hodnota:

$$A[i] = A'[\mathrm{rank}_1(i)].$$

Naopak, operácia $\mathrm{select}_1$ nám umožňuje nájsť, kde sa v pôvodnom poli nachádza $j$-ty neprázdny prvok, teda slúži na prepočet indexov opačným smerom:

$$A'[j] = A[\mathrm{select}_1(j)].$$

Takéto riešenie zaberá $(1 + \varepsilon)m + n \times \ell$ bitov.

Ktoré riešenie je najlepšie? Zaléží...

Predpokladajme, že implementácia `rank` a `select`, resp. heštabuľky zaberie $\varepsilon = 10\%$ pamäte navyše. Vezmime napríklad $n = 2^{20} \approx 1,000,000$ prvkov, pričom každý z nich má veľkosť $\ell = 256$ bitov (teda 32 bajtov). Samotné dáta teda zaberajú $n \times \ell = 2^{20} \times 256 = 32\,\mathrm{MB}$. Porovnajme teraz, koľko pamäte zaberie celá štruktúra pri rôznych hustotách:

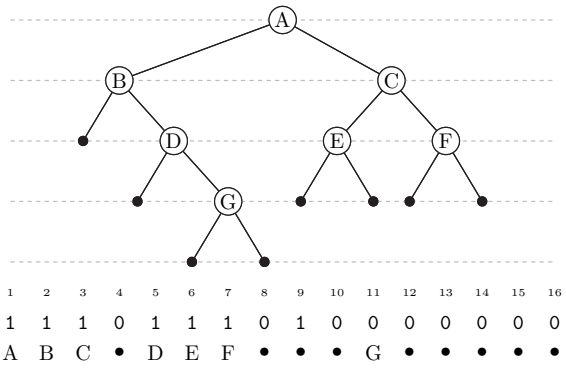
| pamäť | riedke pole $m \times \ell$ | heštabuľka $(1+\varepsilon) \times n \times (\lg m + \ell)$ | zhustené pole s bitvektorom $(1+\varepsilon)m + n \times \ell$ |
|---|---|---|---|
| $m = 1.1n$ | 35.2 MB | 37.97 MB | 32.15 MB |
| $m = 2n$ | 64 MB | 38.1 MB | 32.28 MB |
| $m = 10n$ | 320 MB | 38.4 MB | 33.38 MB |
| $m = 50n$ | 1.56 GB | 38.73 MB | 38.88 MB |
| $m = 100n$ | 3.13 GB | 38.86 MB | 45.75 MB |

Inými slovami, *dodatočná pamäť* slovníka $i \mapsto A_i$ je prinajmenšom $n \lg m$ bitov navyše – toľko potrebujeme len na uloženie indexov. V prípade zhusteného poľa s bitvektorom je dodatočná pamäť približne $(1+\varepsilon) \times m$ bitov, teda len o málo viac než samotná dĺžka pôvodného poľa. Z toho vyplýva, že použitie slovníka sa oplatí až vtedy, keď $m \gg n \lg m$, teda keď je pole skutočne veľmi riedke.

**Motivácia #3: Úsporné stromy.** Predstavme si, že máme statický binárny strom a chceme ho uložiť čo najúspornejšie. Tradičná reprezentácia – teda každému vrcholu uložiť smerníky na ľavého a pravého syna a rodiča – je síce pohodlná, ale z pohľadu pamäte veľmi neefektívna. Každý smerník zaberá 64 bitov, takže len samotné odkazy pridávajú až $3 \times 64 = 192$ bitov na každý jeden vrchol.

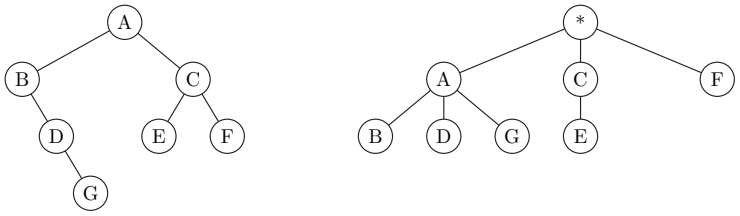
Dá sa to lepšie?

Áno: Doplňme strom o tzv. *vonkajšie vrcholy* (•), ktorými nahradíme všetky nulové smerníky. Každý vnútorný vrchol má teda presne dvoch synov – buď vnútorných, alebo vonkajších. Teraz stačí prejsť strom po úrovniach a zapísať si `1` ak je daný vrchol vnútorný, `0` ak je vonkajší:




Týmto spôsobom dokážeme binárny strom reprezentovať ako obyčajný bitvektor, ktorý má dĺžku $2n + 1$ bitov. Navyše, vďaka operáciám `rank` a `select` nad týmto bitvektorom sa dokážeme po strome pohybovať – z rodiča na deti, z detí na rodiča. (Ako presne, necháme ako úlohu pre čitateľa.)

Alternatívny prístup je previesť binárny strom na *zakorenený usporiadaný strom* pomocou tzv. reprezentácie *left-child–right-sibling*: V tejto schéme interpretujeme ľavého syna ako *prvého potomka* a pravého syna ako *nasledujúceho súrodenca*.




```
(  (  (  )  (  )  (  )  )  (  (  )  )  (  )  )
*  A  B  B  D  D  G  G  A  C  E  E  C  F  F  *
```

(Napríklad v pôvodnom binárnom strome koreň $A$, jeho pravý syn $C$ a *jeho* pravý syn $F$ tvoria súrodencov – synov nového koreňa označeného $(*)$. Ľavý syn $B$ sa v tejto reprezentácii stáva *prvým* synom vrcholu $A$, a jeho praví potomkovia sa premenia na jeho súrodencov.)

Nakoniec spravíme Eulerovský prechod okolo nového stromu. Vždy, keď do vrcholu prvýkrát vchádzame, zapíšeme si ľavú zátvorku, `(` a keď z neho naposledy odchádzame, zapíšeme pravú zátvorku `)`. Takto dostaneme *dobre uzátvorkovanú postupnosť*, ktorá presne vystihuje tvar pôvodného stromu.

Tieto dve transformácie – *left-child–right-sibling* a *Eulerovský prechod* – sú izomorfizmy: každému binárnemu stromu zodpovedá práve jeden zakorenený usporiadaný strom, a každej takejto štruktúre prislúcha práve jedna dobre uzátvorkovaná postupnosť. Naopak, z každej správne uzátvorkovanej postupnosti vieme strom jednoznačne zrekonštruovať. Zároveň postupnosť zátvoriek môžeme reprezentovať ako bitový vektor, kde `(` je `0` a `)` je `1`. Opäť prenecháme na čitateľa, aby domyslel, ako operácie na stromoch premeniť na operácie na bitvektore.

Vidíme teda, že bitvektory a operácie na nich nie sú len akýsi podivný umelý problém, ktorým sa teoretický informatici zaoberajú počas dlhých zimných večerov, ale tvoria praktický nástroj, na ktorom je založených veľa ďalších úsporných dátových štruktúr. Poďme teraz porozmýšľať, ako takúto dátovú štruktúru navrhnúť.

## 20.1 Úsporný rank

Začnime dvoma triviálnymi riešeniami:

**#0. Žiadne predspracovanie:** Rank spočítame v lineárnom čase. Katastrofa, ale OK, musel som to spomenúť.

**#1. Predpočítame všetko:** Spočítame si tabuľku $R[i] = \text{rank}_1(i)$ – každý rank takto vieme povedať v konštantnom čase. Avšak pamäť je katastrofálna: napríklad ak použijeme 64-bitové čísla, tak pole rankov bude 64×väčšie ako pôvodný bitvektor!

Vieme vymyslieť niečo lepšie?

**#2. Predpočítame menej:** Ak chceme zmenšiť potrebnú pamäť, čo tak predpočítať si rank len pre každú $k$-tu pozíciu? Každý rank zaberá $\lceil \lg n \rceil$ bitov, ale ak si zapamätáme iba každé $k$-te, celkovo ranky zaberú $(n/k) \times \lg n$ bitov.

Napríklad, ak zvolíme $k = 10 \lg n$, ranky spotrebujú len 10% pamäte navyše oproti samotnému bitvektoru.

Lenže na druhej strane, ak chceme, aby ranky zaberali menej ako $n$ bitov, *potrebujeme* $k > \lg n$, ale tým pádom sa nám zhorší časová zložitosť. Ak si pamätáme len každý $(\log n)$-tý rank, tie zvyšné musíme dopočítať v lineárnom čase od $k$. Či?

**#3. Praktické riešenie:** Poďme sa najskôr porozprávať o reálnej implementácii na reálnom počítači. Pretože $k = 10 \lg n$ znie hrozivo, ale ak máme povedzme miliardové pole, tak $\lg 10^9 < 30$, takže máme jeden rank pre každých 300 bitov. Pre predstavu, jeden register je 64 bitov a jedna cache line je 64B = 512 bitov.

Navyše súčasné počítače majú inštrukciu `popcount`, ktorá spočíta počet jednotkových bitov v registri. Jediná inštrukcia v konštantnom čase. Takže 300 bitov zvládneme 5-timi inštrukciami, ktoré môžu bežať dokonca paralelne. Dokonca najnovšie počítače (v čase písania) s architektúrou AVX-512 (napríklad Intel Ice Lake) majú vektorové inštrukcie `VPOPCNTDQ`, ktoré spočítajú počet jednotiek v 8-mich 64-bitových číslach *naraz*, dokonca s prípadnou maskou.

Ak chceme spočítať len počet jednotiek po nejakú pozíciu $i$, stačí bity od tejto pozície vynulovať pomocou bitového `AND`u. Konkrétne stačí zobrať `1`, posunúť ju o $i$ doľava a odčítať 1, teda `(1 << i) - 1`. Dostaneme v dvojkovej sústave:

$$\begin{array}{rl} \texttt{1 << i} & \texttt{00001}\overbrace{\texttt{000 ... 000}}^{i} \\ \texttt{(1 << i) - 1} & \texttt{00000111 ... 111} \end{array}$$

Keď ľubovoľné číslo z`AND`ujeme s takouto maskou, ostanú nám len bity na jednotkových pozíciach, teda prvých $i$ pozícií (sprava).

Jedno veľmi praktické riešenie teda je, že vezmeme 512 bitov (64 bajtov, 1 cache line). Z toho prvých 64 bitov bude predpočítaný rank, teda počet jednotiek pred touto pozíciou a zvyšných 448 bitov budú bity pôvodného bitvektoru. Dôležité nemáme dve oddelené polia: pôvodný bitvektor a predpočítané ranky, ale všetky hodnoty sú uložené prerývane v jednom poli. Takto nám stačí namiesto dvoch prístupov na dve rôzne miesta v pamäti načítať naraz len jednu cache line. Výsledný rank spočítame ako súčet predpočítaného ranku a hodnoty dopočítanej maximálne 7-mimi operáciami `popcount`, s prípadným maskovaním. Takéto riešenie zaberá $\approx 14\%$ pamäte navyše oproti samotnému bit-

vektoru. Dokonca ak máme bitvektor dĺžky menej ako 4 miliardy, stačí nám len 32 bitov na každý rank a zvyšných 480 bitov tvoria bity z bitvektoru, čo je len $1/15 \approx 6.7\%$ navyše.

Čas je prakticky konštantný. (Teda ak predpokladáme model počítača, v ktorom máme registre dĺžky $\Theta(\log n)$ a `popcount` vieme spočítať v konštantnom čase.)

Dá sa to ešte lepšie? Čo keby sme chceli ešte úspornejšie riešenie?

Predstavme si, že máme $n = 10\,\text{GB}$ dlhý bitvektor. To znamená, že na ranky potrebujeme aspoň 34 bitov ($\lceil \lg(10 \cdot 2^{30}) \rceil$). Ak bitvektor nasekáme na bloky dĺžky $b = 512$, potrebujeme aspoň

$$\frac{n}{b} \times 34 = n \times 34/512 = 680\,\text{MB} \approx 6.6\% \times n$$

Vieme sa dostať pod 6%? Mohli by sme ešte predĺžiť veľkosť bloku, lenže tým zároveň algortimus spomaľujeme...

**#4. Ešte úspornejšie riešenie:** Môžeme však použiť dvoj-úrovňové riešenie! Nasekáme bitvektor na superbloky dĺžky $s = 2^{14} - 1 = 16383$ bitov. Pre každý začiatok superbloku spočítame rank po danú pozíciu. Ostáva nám vyriešiť, ako spočítame rank v rámci jedného superbloku.

Odpoveď je, že môžeme každý superblok opäť nasekať na menšie bloky dĺžky $b = 512$ (a v rámci bloku dĺžky 512 už rank spočítame pomocou bitových operácií). Vtip je však v tom, že pre tieto bloky stačí predpočítať rank iba *od začiatku superbloku*, čo je číslo od 0 po 16383, na ktoré nám stačí 14 bitov!

Koľko pamäte takéto riešenie spotrebuje? Máme $n/s$ superblokov a na každý rank potrebujeme 34 bitov, plus máme $n/b$ blokov a na každý blok predpočítame „malý rank“ od začiatku superbloku, na ktorý potrebujeme len 14 bitov. Spolu:

$$\frac{n}{s} \times 34 + \frac{n}{b} \times 14 = n \times \left( \frac{34}{2^{14} - 1} + \frac{14}{512} \right) \approx 301\,\text{MB} \approx 2.94\% \times n$$

Ak by sme zvolili bloky dĺžky $b = 1024$, oplatí sa zvoliť superbloky dĺžky $s = 2^{15} - 1 = 32767$. Pri tejto voľbe by ranky zaberali

$$n \times \left( \frac{34}{2^{15} - 1} + \frac{15}{1024} \right) \approx 161\,\text{MB} \approx 1.57\% \times n$$

**Teória.** V teórii nás zaujímajú *úsporné* riešenia, kde dodatočná pamäť je *asymptoticky menšia* ako minimálna pamäť pre danú štruktúru – v našom prípade $o(n)$. To znamená, že chceme dosiahnuť takú pamäť, že podiel

$$\frac{\text{dodatočná pamäť na ranky}}{\text{pamäť na bitvektor}} \to 0 \qquad \text{pre } n \to \infty$$

je zanedbateľný, resp. blíži sa k nule pre $n$ idúce do nekonečna.

Vo všeobecnosti v dvojúrovňovom riešení ranky zaberajú

$$n \times \left( \frac{\lceil \lg n \rceil}{s} + \frac{\lceil \lg s \rceil}{b} \right) \text{ bitov.}$$

Ak zvolíme napríklad superbloky dĺžky $s = \lg^2 n$ a bloky dĺžky $b = \frac{1}{2} \lg n$, tak ranky budú zaberať

$$n \times \left( \frac{\cancel{\lceil \lg n \rceil}}{\lg n \cdot \cancel{\lg n}} + \frac{\lceil \lg(\lg^2 n) \rceil}{\frac{1}{2} \lg n} \right) = n \times O\Bigg( \underbrace{\frac{1 + \lg \lg n}{\lg n}}_{\to 0 \text{ pre } n \to \infty} \Bigg) = o(n)$$

A čo sa týka počítania ranku v rámci jedného bloku (dĺžky $\frac{1}{2} \lg n$), tak aj keby sme neuvažovali model s operáciou `popcount`, tak pri takejto malej dĺžke si vieme predpočítať jednu globálnu tabuľku. Všetkých možných bitvektorov dĺžky $b$ je totiž $2^b$, čo pre našu voľbu $b = \frac{1}{2} \lg n$ znamená $\sqrt{n}$. Aj keby sme si teda pre každý bitvektor a pre každú pozíciu v ňom spočítali rank, zaberie to iba $O(\sqrt{n} \times \lg^2 n)$ bitov, čo je zanedbateľné oproti $n$.

## 20.2 Úsporný select

Operácia $\text{select}_1(j)$ je v istom zmysle inverzná ku $\text{rank}_1(i)$. Pri jeho riešení budeme vyzbrojení technikami, ktoré sme použili na rank, napriek tomu je tento problém zložitejší. Pri ranku nám stačilo vyriešiť $\text{rank}_1$ a počet núl sme mohli počítať vďaka vzťahu $\text{rank}_0(i) = i - \text{rank}_1(i)$. Naproti tomu ak vieme nájsť $j$-tu jednotku, to nám ešte nič nehovorí o tom, kde sa nachádza $j$-ta nula. Naše úsilie budeme musieť zdvojnásobiť a vyrobiť jednu štruktúru pre $\text{select}_0$ a jednu pre $\text{select}_1$.

Obvyklé triviálne riešenia stále fungujú (a stále sú katastrofálne zlé):

**#0. Žiadne predspracovanie:** Hľadanie trvá $O(n)$.

**#1. Predpočítame všetko:** Zaberá $O(n \log n)$ bitov pamäte navyše. A my chceme $o(n)$.

V skutočnosti, ak nás zaujíma iba $\text{select}_1$, je to $O(n_1 \log n)$ bitov, kde $n_1$ je počet jednotiek v poli. Táto reprezentácia sa však oplatí iba ak je počet jednotiek dosť malý – $n_1 = o(n/\log n)$.

**#2. Binárne vyhľadávanie na rankoch:** Jedno veľmi praktické riešenie, ak už máme vybudovanú štruktúru pre ranky, je použiť binárne vyhľadávanie a najskôr medzi rankami superblokov nájsť ten správny superblok, potom medzi rankami blokov nájsť ten správny blok a nakoniec v rámci bloku dohľadať tú správnu jednotku. Časová zložitosť bude $O(\log n)$, čo nie je na zahodenie a výhodou je, že okrem štruktúry pre rank už nepotrebujeme žiadne ďalšiu pamäť navyše.

**#3. Predpočítame menej:** Keďže predpočítať všetko má príšernú pamäťovú zložitosť, ale predpočítanie vo všeobecnosti pomáha, môžeme ušetriť pamäť tak, že si toho predpočítame menej. Ak si zapamätáme iba pozíciu každej $k$-tej jednotky, pamäťová zložitosť bude $O(n/k \times \log n)$, čo, povedzme, pre $k = \log^2 n$, je sublineárne $o(n)$.

Človek by čakal, že takto zlepšíme časovú zložitosť, ale v najhoršom prípade tomu tak nie je. Človek by tiež čakal, že takto nasekáme bitvektor na menšie bloky, použijeme dvoj-úrovňové riešenie ako pri rankoch a môžeme si odfajknúť select a rozlúčiť sa. Bohužiaľ, nie je to také ľahké.

Totiž zatiaľčo pri rankoch sme bitvektor nasekali na rovnaké bloky dĺžky $b$, pri selecte budú mať bloky rôzne dĺžky. Dokonca veľmi dramaticky rôzne: napríklad ak vieme, že $\text{select}_1(0) = 0$, tak $\text{select}_1(1)$ môže byť hneď vedľa na pozícii 1, alebo aj úplne na konci $\text{select}_1(1) = n-1$, ak sú medzitým samé nuly.

Našťastie nás zachráni, že pre riedke polia, kde je veľmi málo jednotiek, môžeme použiť aj triviálne riešenie – a síce poznačíme si pozície všetkých jednotiek.

**#4. Plus všetky pozície v dlhých blokoch:** Predpočítajme si pozíciu každej $k$-tej jednotky, kde $k = \log^2 n$. Tieto pozície nám rozdelia bitvektor na menšie bloky. Bloky môžu mať veľmi rôznorodé dĺžky, ale v každom bloku (možno okrem posledného) sa nachádza presne $k$ jednotiek.

Ak je blok dlhší ako $k^2 = \log^4 n$, budeme ho volať *dlhý* blok – ostatné budeme volať *krátke*. Zjavne dlhých blokov môže byť najviac $n/k^2$ a v každom je najviac $k$ jednotiek – to je dokopy najviac $n/k = n/\log^2 n$ jednotiek. Ak si aj pre každú z nich zapamätáme jej pozíciu, zaberie to dokopy len $O(n/\log n) = o(n)$ pamäte. Navyše pre každý z $n/k$ blokov si potrebujeme poznačiť, či je dlhý alebo krátky a pre dlhé bloky smerník na pole všetkých pozícii – na toto opäť môžeme použiť bitvektor s predpočítaným rankom, ale aj priamočiarejšie riešenia budú zaberať len $o(n)$ pamäte.

Keď takto vyriešime všetky dlhé bloky, ostáva nám doriešiť tie krátke. Každý z nich má dĺžku najviac $\log^4 n$, takže ak použijeme binárne vyhľadávanie, dostaneme riešenie s časovou zložitosťou $O(\log(\log^4 n)) = O(\log \log n)$.

**#5. Dvoj-úrovňové riešenie.** Budeme postupovať rovnako ako v predchádzajúcom riešení – predpočítame si pozíciu každej $k$-tej jednotky pre $k = \log^2 n$. Pre dlhé bloky si predpočítame pozície všetkých jednotiek. Pre krátke bloky použijeme tú istú myšlienku ešte raz.

Vyderžaj pioner.

Pre každý *krátky* blok si predpočítame každú $\ell$-tú jednotku pre $\ell = (\log \log n)^2$. Vtip je opäť v tom, že pozíciu budeme počítať od začiatku bloku a keďže všetky krátke bloky majú dĺžku najviac $\log^4 n$, táto pozícia zaberá len $4 \log \log n$ bitov. Spolu všetky tieto pozície zaberú najviac

$$\frac{n}{\ell} \times 4 \log \log n = O\left(\frac{n \cancel{\log \log n}}{\log \log n \times \cancel{\log \log n}}\right) = o(n) \text{ bitov.}$$

Tieto pozície nám rozdelia každý krátky blok na menšie *minibloky*. Minibloky budú mať opäť rôznu dĺžku. Budeme hovoriť že minibloky dlhšie ako $\ell^2 = (\log\log n)^4$ sú *dlhé* a tie ostatné sú krátke. Pre dlhé minibloky si môžeme opäť uložiť všetky pozície jednotiek – takýchto dlhých miniblokov je len $n/\ell^2$ a každý obsahuje najviac $\ell$ jednotiek – to je spolu $n/\ell = n/(\log\log n)^2$ jednotiek. Každá pozícia zaberá len $\log\log n$ bitov, takže si to môžeme dovoliť. Celková pamäť pre dlhé minibloky je

$$O\left(\frac{n}{\ell^2} \times \cancel{\ell} \times \log\log n\right) = O\left(\frac{n}{(\log\log n)^{\cancel{2}}} \times \cancel{\log\log n}\right) = O\left(\frac{n}{\log\log n}\right) = o(n).$$

Nakoniec nám ostanú krátke minibloky dĺžky najviac $\ell^2 = (\log\log n)^4$ a tie sa už zmestia do registra, resp. všetkých možných bitvektorov takejto dĺžky je len veľmi málo, takže si vieme predpočítať jednu globálnu tabuľku, ktorá nám povie pre každý bitvektor a každé $j$ pozíciu $j$-tej jednotky.

Takto dokážeme rank aj select spočítať v konštantnom čase a v $o(n)$ dodatočnej pamäti.

## 20.3 Komprimovaný bitvektor

Ako sme videli v predchádzajúcej časti, ak máme reťazec bitov, vieme si k nemu uložiť pomocné informácie a vďaka nim podporovať operácie rank a select. Celá táto sranda je veľmi praktická a stojí nás len malú pamäť, rádovo jednotky percent z pamäte, ktorú zaberá pôvodný reťazec.

V nasledujúcej časti skúsime byť ešte ambicióznejší. Vedeli by sme daný bitvektor *skomprimovať* tak, že stále budeme vedieť podporovať operácie access, rank a select? A za akú cenu?

Samozrejme, mohli by sme zo šuflíka vytiahnuť ľubovoľný kompresný algoritmus, a celý bitvektor skomprimovať – ale ako potom budeme vedieť zistiť hodnotu $i$-teho bitu bez toho, aby sme celý reťazec dekomprimovali?

Ďalší nápad by mohol byť rozsekať vstupný bitvektor na bloky (nápad „sekať na bloky“ nás už viackrát zachránil). Každý blok zakódujeme zvlášť. Budeme si musieť zapamätať nejakú pomocnú infomáciu, aby sme rýchlo vedeli nájsť $k$-ty blok, avšak potom stačí na zistenie $i$-teho bitu dekomprimovať iba jediný blok a nie celý reťazec. Navyše, myšlienka blokov ide pekne dokopy s algoritmami pre rank a select, ktoré tiež delia bitvektor na bloky.

Stojíme však pred neľahkou úlohou. Na jednej strane chceme bloky čo najdlhšie, aby sme mali čo najlepšiu kompresiu a potrebovali čo najmenej dodatočnej pamäte pre rank a select, na druhej strane chceme bloky krátke, aby sme vedeli rýchlo dekódovať $i$-ty bit.

Tu si ukážeme jednu jednoduchú metódu, ako bitový reťazec skomprimovať, pričom sa budeme blížiť entropii daného reťazca. Dátová štruktúra sa volá RRR podľa mien jej vynálezcov: Raman, Raman a Rao.

Myšlienka je jednoduchá:

- Vezmime blok $r$ bitov.

- Spočítajme $c$, počet jednotiek v ňom.
- Predstavme si, že máme utriedený zoznam všetkých bitvektorov dĺžky $r$, ktoré obsahujú $c$ jednotiek a spočítajme poradie $p$ nášho bloku v tomto zozname.
- Náš blok budeme kódovať dvojicou $(c, p)$.

Ukážme si to na príklade. Predstavme si, že máme na vstupe:

| 0110000 | 0001000 | 0110010 | 0000000 | 0000010 | 0000101 |
|---|---|---|---|---|---|

a zvolíme si blok dĺžky 7. Pomocná tabuľka je na obrázku 20.1. Prvý blok má 2 jednotky a medzi takými blokmi je 14-ty v poradí (počítame od nuly). Druhý blok má 1 jednotku a je 3-tí v poradí. Tretí blok má 3 jednotky a je to 17-ty blok medzi 7-bitovými blokmi s 3-oma jednotkami. Štvrtý blok sú samé nuly, teda $c = 0$ a žiadne poradie nepotrebujeme kódovať, pretože existuje len jeden taký blok, atď. Výsledný kód bude:

| $c$ | 2 | 1 | 3 | 0 | 1 | 2 |
|---|---|---|---|---|---|---|
| $p$ | 01110 | 011 | 010001 | | 001 | 00001 |

Na zápis každého $c$ potrebujeme 3 bity (môže nadobúdať $0 \ldots 7$, teda $2^3$ rôznych hodnôt), takže dokopy táto reprezentácia zaberie $6 \times 3 + 5 + 3 + 6 + 3 + 5 = 40$ bitov, zatiaľčo pôvodná postupnosť mala $6 \times 7 = 42$ bitov.

To nie je žiadna sláva, lenže na lepšiu kompresiu treba zväčšiť dĺžku bloku $r$.

Vo všeobecnosti, ak máme bloky dĺžky $r$, tak $c$ nadobúda hodnoty medzi 0 a $r$, čo je $r + 1$ možných rôznych hodnôt. Aby sme tieto hodnoty zapísali, potrebujeme $\lceil \lg(r+1) \rceil$ bitov. Je teda výhodné sústrediť sa práve na bloky dĺžky o 1 menej ako nejaká mocnina dvojky (v opačnom prípade plytváme bitmi).

Keď potom narazíme na blok, ktorý má $c$ jednotiek, tak existuje presne $\binom{r}{c}$ bitvektorov s $c$ jednotkami – jednoduchá kombinatorika: z $r$ miest potrebujeme *vybrať podmnožinu* $c$ pozícií, na ktorých sú jednotky, takže odpoveď je kombinačné číslo $\binom{r}{c}$, počet výberov $c$ pozícií z $r$. Na reprezentáciu $X$ rôznych možností potrebujeme $\lceil \lg X \rceil$ bitov, takže na poradie $p$ budeme potrebovať práve $\lceil \lg \binom{r}{c} \rceil$ bitov.

Pre lepšiu predstavu si spočítajme, koľko to je konkrétne, pre rôzne dĺžky blokov.

Pre $r = 15$ sa má situácia nasledovne. Potrebujeme 4 bity na zápis $c$ a počet bitov na zápis pozície je $\lceil \lg \binom{15}{c} \rceil$:

| $r = 15$, $c$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lceil \lg \binom{15}{c} \rceil$ | 0 | 4 | 7 | 9 | 11 | 12 | 13 | 13 | 13 | 13 | 12 | 11 | 9 | 7 | 4 | 0 |
| $4 + \left\lceil \lg \binom{15}{c} \right\rceil$ | 4 | 8 | 11 | 13 | 15 | 16 | 17 | 17 | 17 | 17 | 16 | 15 | 13 | 11 | 8 | 4 |

Z tabuľky vidíme, že na blokoch, ktoré majú menej ako 4 jednotky, alebo viac ako 11 jednotiek, budeme šetriť pamäť. Bloky so 4 alebo 11 jednotkami budú

| | *Žiadna jednotka* | | |
|---|---|---|---|
| 0 | 0000000 | → | $\varepsilon$ |

| | *Jedna jednotka* | | |
|---|---|---|---|
| 0 | 0000001 | → | 000 |
| 1 | 0000010 | → | 001 |
| 2 | 0000100 | → | 010 |
| 3 | 0001000 | → | 011 |
| 4 | 0010000 | → | 100 |
| 5 | 0100000 | → | 101 |
| 6 | 1000000 | → | 110 |

| | *Dve jednotky* | | |
|---|---|---|---|
| 0 | 0000011 | → | 00000 |
| 1 | 0000101 | → | 00001 |
| 2 | 0000110 | → | 00010 |
| 3 | 0001001 | → | 00011 |
| | ... | | ... |
| 14 | 0110000 | → | 01110 |
| 15 | 1000001 | → | 01111 |
| 16 | 1000010 | → | 10000 |
| | ... | | ... |
| 20 | 1100000 | → | 10100 |

| | *Tri jednotky* | | |
|---|---|---|---|
| 0 | 0000111 | → | 000000 |
| 1 | 0001011 | → | 000001 |
| 2 | 0001101 | → | 000010 |
| | ... | | ... |
| 8 | 0011010 | → | 001000 |
| 9 | 0011100 | → | 001001 |
| 10 | 0100011 | → | 001010 |
| 11 | 0100101 | → | 001011 |
| 12 | 0100110 | → | 001100 |
| 13 | 0101001 | → | 001101 |
| 14 | 0101010 | → | 001110 |
| 15 | 0101100 | → | 001111 |
| 16 | 0110001 | → | 010000 |
| 17 | 0110010 | → | 010001 |
| 18 | 0110100 | → | 010010 |
| 19 | 0111000 | → | 010011 |
| 20 | 1000011 | → | 010100 |
| | ... | | ... |
| 31 | 1100010 | → | 011111 |
| 32 | 1100100 | → | 100000 |
| 33 | 1101000 | → | 100001 |
| 34 | 1110000 | → | 100010 |

Obr. 20.1: Všetky 7-bitové reťazce môžeme rozdeliť do skupín podľa počtu jednotiek (na obrázku sú len skupiny s 0–3 jednotkami). V rámci každej skupiny si vypíšeme reťazce s daným počtom jednotiek a očíslujeme. Keďže existuje 7 reťazcov s 1 jednotkou, na zapísanie poradia v tejto skupine nám stačia 3 bity. Reťazcov s 2-oma jednotkami je $\binom{7}{2} = 21$, čo vieme zapísať 5 bitmi a reťazcov s 3-omi jednotkami je $\binom{7}{3} = 35$, na čo potrebujeme 6 bitov. Tabuľky pre 4–7 jednotiek budú vyzerať podobne, symetricky – stačí bity znegovať a očíslovať opačne.

zaberať rovnako 15 bitov v pôvodnom aj komprimovanom reťazci. Bloky, ktoré majú 5–10 jednotiek, sa nám ešte predĺžia o 1 alebo 2 bity. (Všimnite si, že tabuľka je symetrická podľa stredu, keďže kombinačné čísla sú symetrické – v tabuľke máme vlastne len zlogaritmovaný riadok Pascalovho trojuholníka.)

Ak zvolíme bloky dĺžky 31, budeme potrebovať 5 bitov na zápis $c$ a na zápis pozície $p$ budeme potrebovať:

| $r=31$, $c$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lceil \lg \binom{31}{c} \rceil$ | 0 | 5 | 9 | 13 | 15 | 18 | 20 | 22 | 23 | 25 | 26 | 27 | 28 | 28 | 28 | 29 |
| $5 + \left\lceil \lg \binom{31}{c} \right\rceil$ | 5 | 10 | 14 | 18 | 20 | 23 | 25 | 27 | 28 | 30 | 31 | 32 | 33 | 33 | 33 | 34 |

druhá polovica tabuľky je symetrická. Takže bloky s menej ako 10 alebo viac ako 21 jednotiek z 31 sa skracujú a bloky s 11–20-timi jednotkami sa mierne predlžujú.

Nakoniec ak zvolíme bloky dĺžky 63, na zápis $c$ potrebujeme 6 bitov a na zápis pozície:

| $r=63$, $c$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lceil \lg \binom{63}{c} \rceil$ | 0 | 6 | 11 | 16 | 20 | 23 | 27 | 30 | 32 | 35 | 37 | 40 | 42 | 44 | 46 | 47 |
| $6 + \left\lceil \lg \binom{63}{c} \right\rceil$ | 6 | 12 | 17 | 22 | 26 | 29 | 33 | 36 | 38 | 41 | 43 | 46 | 48 | 50 | 52 | 53 |

| $c$ | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\lceil \lg \binom{63}{c} \rceil$ | 49 | 50 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 58 | 59 | 59 | 60 | 60 | 60 | 60 |
| $6 + \left\lceil \lg \binom{63}{c} \right\rceil$ | 55 | 56 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 64 | 65 | 65 | 66 | 66 | 66 | 66 |

Vidíme, že bloky s menej ako 23 a viac ako 40 jednotkami sa skracujú a bloky s 24–39 jednotkami sa predlžujú.

Aká dobré je takáto kompresia?

Predstavme si, že vygenerujeme úplne náhodný reťazec, ktorý má len 5% jednotiek (každý jeden bit vygenerujeme úplne nezávisle tak, že si hodíme mincou, na ktorej padá jednotka s 5% pravdepodobnosťou). Shannonova entropia takéhoto zdroja je $H(5\%) \approx 0.286$, teoreticky sa takýto reťazec nedá skomprimovať lepšie ako na $\approx 29\%$. Ak zvolíme RRR kódovanie a bloky dĺžky 63, vieme skrátiť (očakávane) reťazec na $\approx 33.7\%$. Pre rôzne počty jednotiek a rôzne veľkosti blokov:

| | | *očakávaná kompresia s RRR* | | |
|---|---|---|---|---|
| *#jednotiek* | *entropia* | $r=127$ | $r=63$ | $r=31$ |
| 5% | 0.286 | 31.5% | 33.7% | 37.6% |
| 10% | 0.469 | 49.4% | 51.2% | 54.2% |
| 20% | 0.722 | 74.3% | 75.8% | 78% |
| 30% | 0.881 | 90.2% | 91.4% | 93.3% |
| 40% | 0.971 | 99.1% | 100.3% | 102% |
| 50% | 1 | 102% | 103% | 105% |

(posledný riadok je úplne náhodný reťazec, kde každý bit je s rovnakou pravdepodobnosťou `0` alebo `1`, ktorý je nekomprimovateľný.)

## Súvis s entropiou

Ak označíme $n$ dĺžku celého bitvektoru, $n_0$ a $n_1$ počet núl, resp. počet jednotiek, tak všetky poradia zaberajú

$$\sum_i \left\lceil \lg \binom{r}{c_i} \right\rceil \leq \sum_i \lg \binom{r}{c_i} + n/r$$

a

$$\sum_i \lg \binom{r}{c_i} = \lg \prod_i \binom{r}{c_i} \leq \lg \binom{n}{n_1}.$$

Totiž $\prod_i \binom{r}{c_i}$ je počet spôsobov, ako vybrať $c_1$ pozícií jednotiek v prvom bloku a zároveň $c_2$ pozícií v druhom bloku a tak ďalej. Dokopy vyberieme $n_1$ pozícií z $n$, avšak každý blok má predpísaný počet jednotiek. Hodnota $\binom{n}{n_1}$ je proste počet výberov $n_1$ pozícií z $n$ bez ďalších obmedzení, tzn. väčšie číslo.

Zo Stirlingovej aproximácie $\lg n! = n \lg n - n \lg e + O(\lg n)$ potom vyplýva

$$\begin{aligned}
\lg \binom{n}{n_1} &= \lg n! - \lg n_1! - \lg n_0! \\
&= n \lg n - \cancel{n \lg e} - n_1 \lg n_1 + \cancel{n_1 \lg e} - n_0 \lg n_0 + \cancel{n_0 \lg e} + O(\lg n) \\
&= n_1 \lg \frac{n}{n_1} + n_0 \lg \frac{n}{n_0} + O(\log n) \\
&= nH(S) + O(\log n)
\end{aligned}$$

Samozrejme, v RRR-kódovaní máme ešte navyše všetky $c$-čka, ktoré zaberajú $\lceil \lg(r+1) \rceil$ bitov pre každý blok a blokov je $n/r$. Spolu je to teda približne $n \times (\lg r)/r$, čo pre malé bloky nie je úplne zanedbateľné: pre $r = 31$ to je 16% z $n$, pre $r = 63$ takmer 10% z $n$.

V teórií si môžeme povedať, že zvolíme $r = \log n$ a tým pádom $n \times (\lg r)/r = o(n)$. V praxi na konštantách záleží zaujímajú nás hodnoty pre konkrétne $n$, nie chovanie keď $n \to \infty$.

## Ako kódovať a dekódovať bloky

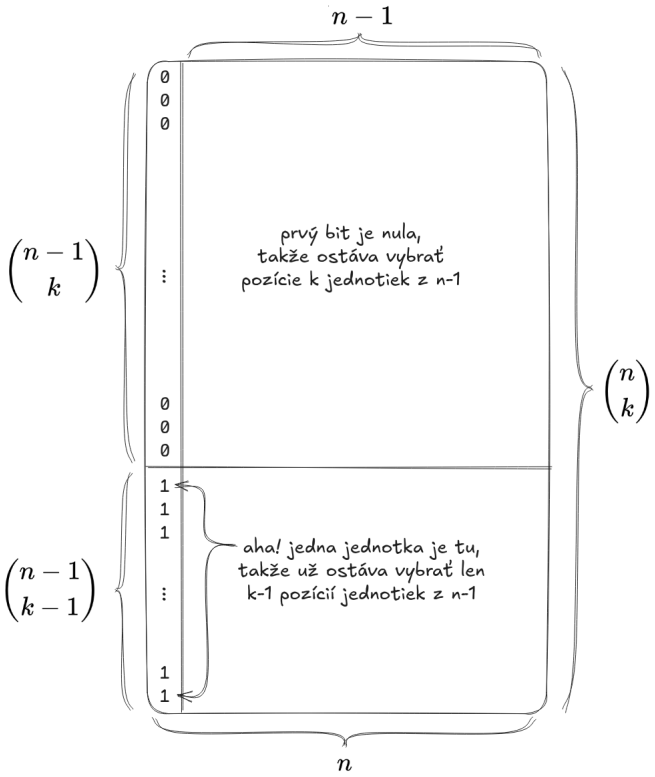
Poďme sa teraz pozrieť, ako implementovať jednotlivé operácie. V prvom rade budeme potrebovať vedieť jednotlivé bloky kódovať a dekódovať. Prezrite si ešte raz obrázok 20.1 vpravo. Otázka znie, ako napríklad vieme zistiť, že reťazec `0101100` je 15-ty (z 35) v poradí medzi reťazcami dĺžky 7 s 3-omi jednotkami? A naopak, ako vyzerá v poradí 23-tí reťazec dĺžky 7 s 3-omi jednotkami?

Skúste porozmýšľať sami a nájsť vhodný algoritmus na kódovanie a dekódovanie.

Ako pomôcka by vám malo stačiť nasledovné pozorovanie: Počet reťazcov dĺžky $n$ s $k$ jednotkami je presne $\binom{n}{k}$, pretože chceme vybrať $k$ pozícií jednotiek z $n$. Sústreďme sa na prvý bit. Ten je buď `0` alebo `1`.

a) Ak je to `0`, stále ostáva vybrať $k$ pozícií, ale zo zvyšných $n-1$ (to je $\binom{n-1}{k}$ možností).

b) Ak je to `1`, ostáva už vybrať iba $k-1$ pozícií zo zvyšných $n-1$ (to je $\binom{n-1}{k-1}$ možností).




To je známy vzorec pre kombinačné čísla

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1},$$

resp. známy fakt, že v Pascalovom trojuholníku je každé políčko rovné súčtu dvoch susedných políčok nad ním.

Keďže predpokladáme, že máme $\binom{n}{k}$ reťazcov zotriedených lexikograficky, tak prvých $\binom{n-1}{k}$ začína nulovým bitom a ďalších $\binom{n-1}{k-1}$ začína jednotkovým bitom.

Riešenie je jednoduché, rekurzívne.

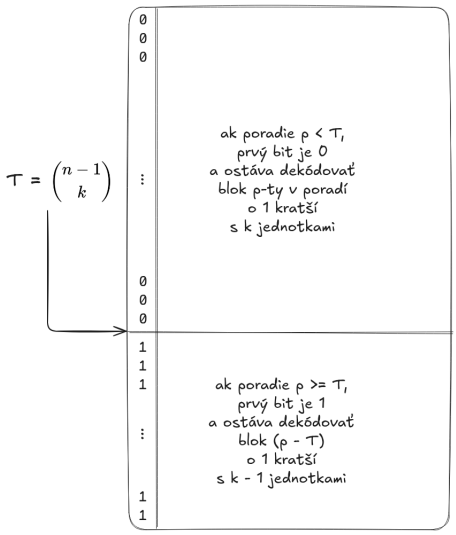
**Dekódovanie.** Príklad: Ktorý reťazec je 23-tí v poradí medzi reťazcami dĺžky 7 s 3-omi jednotkami? Spočítajme $\binom{6}{3} = 20$, tzn. prvých 20 reťazcov začína nulou (pamätajte, že poradie číslujeme od nuly). To znamená že 23-tí reťazec začína

`1`, dokonca je tretí medzi tými, čo začínajú `1`, konkrétne tretí medzi reťazcami dĺžky 6 s 2-oma jednotkami. Dekódujeme zvyšok.

Spočítame $\binom{5}{2} = 10$, to znamená prvých 10 reťazcov začína na nulu, náš tretí reťazec je jeden z nich. Ďalší bit je `0`, dekódujeme zvyšok: tretí medzi reťazcami dĺžky 5 s 2-oma jednotkami.

Spočítame $\binom{4}{2} = 6$, ďalší bit je `0`, dekódujeme zvyšok: tretí medzi reťazcami dĺžky 4 s 2-oma jednotkami. Keďže $\binom{3}{2} = 3$ (a poradie počítame od nuly!), prvý bit je `1` a zvyšok je nultý v poradí – teda úplne na začiatku. Ostávajú 3 bity a jedna jednotka; na začiatku tohto zoznamu je `001`.

Výsledok: 23-tí v poradí je reťazec `1001001`. Pred ním je 20 reťazcov tvaru `0******` a 3 reťazce tvaru `1000***`.



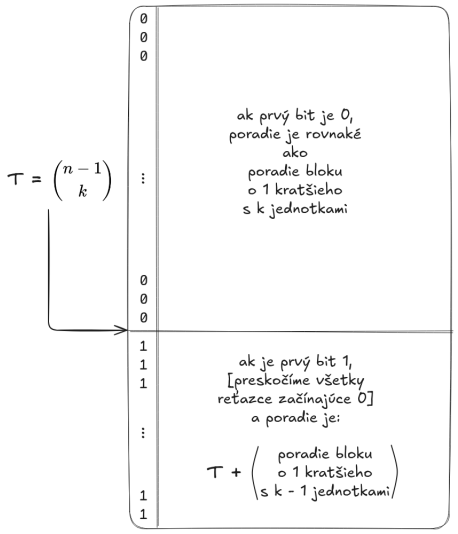

**Kódovanie.** Príklad: ako zakódujeme reťazec `0101100`? Nuž prvý bit je `0`, takže reťazec sa nachádza v prvej časti zoznamu (skôr ako $\binom{6}{3} = 20$). Prvý bit zahodíme a ostáva zakódovať 6-bitový reťazec `101100` s 3-oma jednotkami. Pred týmto reťazcom sú všetky tie, ktoré začínajú na nulu `0*****` – takých je $\binom{5}{3} = 10$. To znamená, že `101100` sa nachádza za všetkými týmito desiatimi, až medzi reťazcami, čo začínajú na `1`. Koľkatý presne?

Zahodíme prvý bit a ostáva zistiť, koľkatý je reťazec `01100` – ten začína na nulu, takže poradie je rovnaké ako poradie `1100` dĺžky 4, s 2-oma jednotkami. Pred týmto sú všetky reťazce `0***` začínajúce na nulu, s 2-oma jednotkami a také sú 3. Tie preskočíme a ostane nám `100`, čo je v poradí druhý reťazec (číslujeme od 0!) medzi 3-bitovými s jednou jednotkou – konkrétne pred ním sú už iba `001` a `010`.

Výsledná pozícia `0101100` je teda $10 + 3 + 2 = 15$. Dostali sme ju vlastne tak, že sme spočítali, koľko reťazcov je *pred* ním a to sú:

- reťazce začínajúce `00*****`, ktorých je $\binom{5}{3} = 10$,

- reťazce začínajúce `0100***`, ktorých je $\binom{3}{2} = 3$, a
- reťazce začínajúce `01010**`, ktoré sú $\binom{2}{1} = 2$.




## Výsledná štruktúra RRR

Budeme mať „minibloky“ dĺžky $r$, povedzme $r = 63$. Pri väčších miniblokoch treba počítať s tým, že kódovanie/dekódovanie bude trochu pomalšie, navyše, potrebujeme počítať s kombinačnými číslami a to najväčšie, $\binom{r}{r/2} \approx 2^r/\sqrt{\pi r/2}$ sa už nemusí zmestiť do 64-bitového registra, takže budeme potrebovať artimetiku s veľkými číslami.

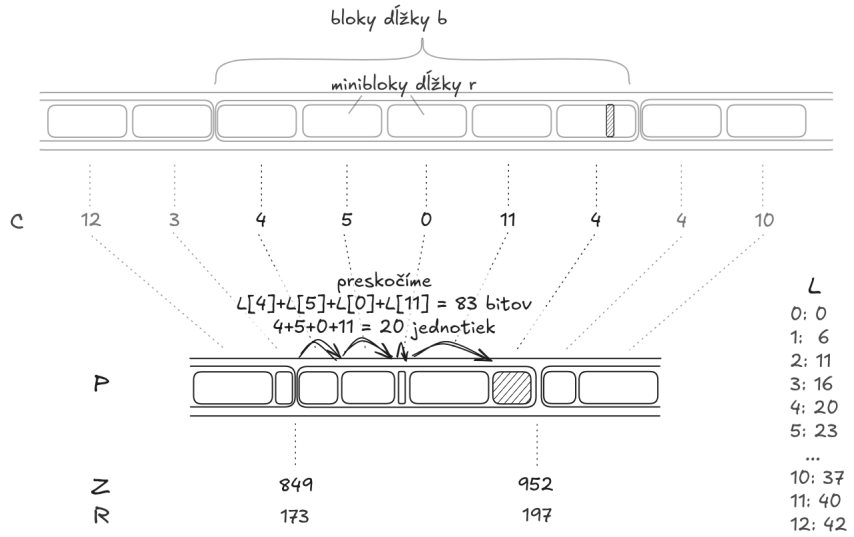
Každý miniblok budeme kódovať ako dvojicu $(c_i, p_i)$, kde $c_i$ = #jednotiek v $i$-tom bloku a $p_i$ = poradie medzi reťazcami dĺžky $r$ s $c$ jednotkami. Tieto hodnoty budeme mať uložené oddelene v dvoch poliach $C$ a $P$.

V zhustenom poli $C$ každá hodnota $c_i$ zaberá $\lg(r+1)$ bitov, napríklad pre $r = 63$ je sú hodnoty 6-bitové. Takže do tohto poľa sa indexuje ľahko.

Pole $P$ obsahuje prvky s premenlivým počtom bitov. Budeme preto potrebovať pomocné pole, aby sme vedeli povedať, na ktorej pozícií začínajú bity zodpovedajúce $k$-temu bloku.

Celý bitvektor rozdelíme na väčšie bloky, povedzme dĺžky $b = 16 \times 63 = 1008$. Pre každý začiatok bloku si spočítame rank $R[i]$ – koľko jednotiek bolo doteraz a pozíciu $Z[i]$, kde začína zakódovaný $i$-ty blok.

Okrem toho sa nám pri výpočtoch bude hodiť predpočítaná tabuľka kombinačných čísiel (Pascalov trohuholník) a predpočítané dĺžky kódov pre daný počet jednotiek $c_i$ – t.j. hodnoty $L[c] = \lceil \lg \binom{r}{c} \rceil$.




Obr. 20.2: Príklad štruktúry RRR. Úplne hore je zobrazený pôvodný vektor rozdelený na bloky a minibloky (len pre ilustráciu, po zakódovaní si ho, samozrejme, nemusíme pamätať). Celý RRR sa skladá z polí $C$ – počty jednotiek v každom minibloku, $P$ – kódy miniblokov, predstavujú poradie medzi reťazcami dĺžky $r$ s $c_i$ jednotkami, $Z$ – pomocné pole pre indexy začiatkov blokov (keďže minibloky $P$ majú rôzne dĺžky), $R$ – predpočítané ranky, počet jednotiek po začiatok bloku, $L$ – hodnoty $\lceil \lg \binom{r}{c} \rceil$.

Ak teraz budeme chcieť napríklad zistiť hodnotu $i$-teho bitu, nech

$$\ell = \underbrace{\lfloor i/b \rfloor}_{\text{číslo bloku}}, \qquad k = \underbrace{\lfloor (i - \ell b)/r \rfloor}_{\text{číslo minibloku v rámci bloku}}, \qquad j = \underbrace{i \bmod r}_{\text{index v rámci minibloku}}.$$

Potom $i$-ty bit je $j$-ty bit v rámci $k$-teho minibloku vnútri $\ell$-tého bloku. V tabuľke $Z[\ell]$ nájdeme, kde začína $\ell$-tý blok. Potom preskočíme $k$-miniblokov – ich dĺžky vieme tak, že čítame pole $C$, kde je počet jednotiek a z poľa $L$ sa dozvieme, aká dĺžka kódu tomu zodpovedá. Tak sa dostaneme ku $k$-temu minibloku, ten dekódujeme a prečítame $j$-ty bit.

Ak chceme zistiť $\text{rank}_1(i)$, postupujeme podobne – v tabuľke $R[\ell]$ nájdeme rank po $\ell$-tý blok. Ako preskakujeme $k$-miniblokov, zároveň si nasčítavame hodnoty $C$, teda počet jednotiek v týchto miniblokoch. Nakoniec $k$-ty blok dekódujeme a spočítame rank po $j$-ty bit.

# Kapitola 21

# Wavelet strom

V predchádzajúcich kapitolách sme riešili, ako efektívne pracovať s bitvektormi – zaviedli sme operácie rank a select, a ukázali sme si, ako ich možno implementovať úsporne, až po optimálne riešenie s $n + o(n)$ bitmi pamäte, dokonca ako ich komprimovať.

Bitvektor je však len najjednoduchší prípad, kde pracujeme s abecedou veľkosti $\sigma = 2$. Čo ak ale máme všeobecnú abecedu – napríklad znaky alebo dokonca celé čísla, ktorých môže byť stovky, tisíce, alebo aj milióny? Ako sa dajú operácie access, rank a select rozšíriť z bitvektoru na text s veľkou abecedou?

Napríklad majme DNA reťazec nad abecedou $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$.

Jedna naivná možnosť je pripraviť si samostatný bitvektor pre každý symbol, kde jednotky označujú výskyty daného písmena: pre DNA by sme teda mali bitvektory $B_{\mathtt{A}}$, $B_{\mathtt{C}}$, $B_{\mathtt{G}}$, $B_{\mathtt{T}}$. Pri takomto riešení by sme však pre každý znak textu potrebovali štyri bity namiesto dvoch – teda dvojnásobok minimálnej teoretickej hranice. Takéto riešenie preto nie je pamäťovo úsporné.

Lepšie je rozšíriť naše praktické riešenie pre bitvektor na viacpísmenovú abecedu. Text rozdelíme na bloky a na začiatok každého bloku si uložíme hlavičku, ktorá obsahuje rank pre každé písmeno – teda počet výskytov `A`, `C`, `G`, `T` všetkých predchádzajúcich blokov. Respektíve, v skutočnosti môžeme trochu pamäte ušetriť a jedno písmeno vynechať, pretože jeho rank sa dá dopočítať z pozície a rankov zvyšných písmen. Napríklad

$$\mathrm{rank}_{\mathtt{T}}(i) = i - \mathrm{rank}_{\mathtt{A}}(i) - \mathrm{rank}_{\mathtt{C}}(i) - \mathrm{rank}_{\mathtt{G}}(i).$$

Zvyšné pozície v rámci bloku potom môžeme dopočítať priamo pomocou bitových operácií, ako `AND`, `XOR`, `SHIFT` a `POPCNT` (implementáciu nechávame ako cvičenie pre čitateľa). Vďaka tomu dokážeme zodpovedať dotaz typu $\mathrm{rank}_{\mathtt{C}}(i)$ či $\mathrm{rank}_{\mathtt{G}}(i)$ veľmi rýchlo, pričom celý text ostáva natlačený v kompaktnom binárnom formáte. Pamäť navyše bude trojnásobná oproti pamäti navyše pri bitvektore, avšak stále $o(n)$.

Keďže však pamäťové nároky rastú lineárne s veľkosťou abecedy, toto riešenie je pre väčšie abecedy nepraktické. Poďme sa pozrieť, ako úlohu riešiť lepšie.

## 21.1 Štruktúra wavelet stromu

Teraz sa pozrime, ako môžeme rozšíriť náš prístup z binárnej abecedy na všeobecnú abecedu veľkosti $\sigma$. Myšlienka je prekvapivo jednoduchá: ak vieme riešiť prípad $\sigma = 2$, môžeme sa k väčšej abecede dostať rekurzívne, postupným delením abecedy na polovice.

Začnime tým, že abecedu $\Sigma$ rozdelíme na dve disjunktné časti $\Sigma = \Sigma_0 \cup \Sigma_1$. Pre každý znak $S[i]$ z textu $S$ si zapíšeme bit

$$B[i] = \begin{cases} 0 & \text{ak } S[i] \in \Sigma_0, \\ 1 & \text{ak } S[i] \in \Sigma_1, \end{cases}$$

Tým získame nový bitvektor $B$, ktorý opisuje, do ktorej „polovice“ abecedy každý symbol patrí. Tento bitvektor si predspracujeme pre rýchle operácie rank a (ak treba, tak aj) select.

Ďalej vytvoríme dva nové reťazce:

- $S_0$ – obsahuje všetky znaky z $S$, ktoré patria do $\Sigma_0$, v rovnakom poradí, v akom sa vyskytovali v $S$;
- $S_1$ – analogicky pre znaky z $\Sigma_1$.

Zrejme platí $|S_0| + |S_1| = |S|$.

Ďalej pokračujeme rekurzívne: každú z dvoch abecied $\Sigma_0$ a $\Sigma_1$ opäť rozdelíme na polovice, pre každú vytvoríme nový bitvektor a nové podreťazce, a tak ďalej, až kým nedosiahneme abecedy veľkosti 1. Na týchto listoch sú už všetkye úlohy triviálne, pretože všetky symboly v liste sú rovnaké.

Tu je príklad wavelet stromu pre text

> *Mama má Emu. Ema má mamu. Paula má lampu. Umelú. Mama úpela. Plela. Má úpal.*

respektíve trochu znormalizovaný: s veľkými písmenami bez dĺžňov a mäkčeňov sa zmestíme do abecedy veľkosti 8:

```
MAMA␣MA␣EMU.␣EMA␣MA␣MAMU.␣PAULA␣MA␣LAMPU.␣UMELU.␣
MAMA␣UPELA.␣PLELA.␣MA UPAL.
```

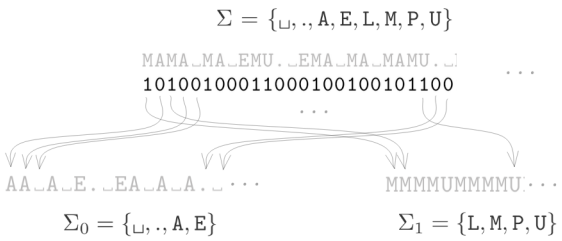
Pri konštrukcii wavelet stromu abecedu $\Sigma = \{\text{␣}, ., \texttt{A}, \texttt{E}, \texttt{L}, \texttt{M}, \texttt{P}, \texttt{U}\}$ rozdelíme na dve polovice:

$$\Sigma_0 = \{\text{␣}, ., \texttt{A}, \texttt{E}\} \qquad \text{a} \qquad \Sigma_1 = \{\texttt{L}, \texttt{M}, \texttt{P}, \texttt{U}\}.$$

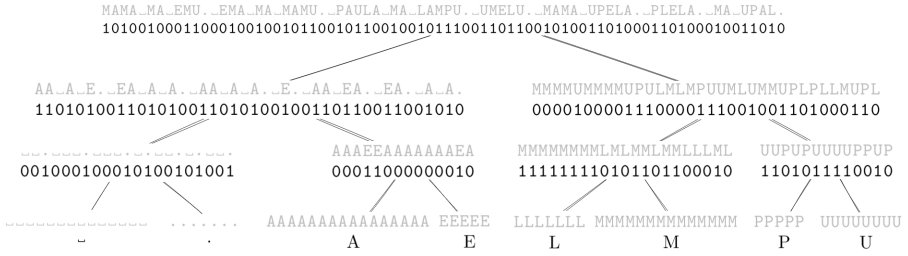
Zostrojíme bitvektor, kde namiesto medzery, bodky, `A`, `E`, je nula (tieto písmená pôjdu doľava) a namiesto `L`, `M`, `P`, `U`, je jednotka (tieto písmená pôjdu doprava):

```
MAMA␣MA␣EMU.␣EMA␣MA␣MAMU.␣
10100100011000100100101100 ...
```

Celý text tak rozdelíme na dva kratšie, s polovičnou abecedou.

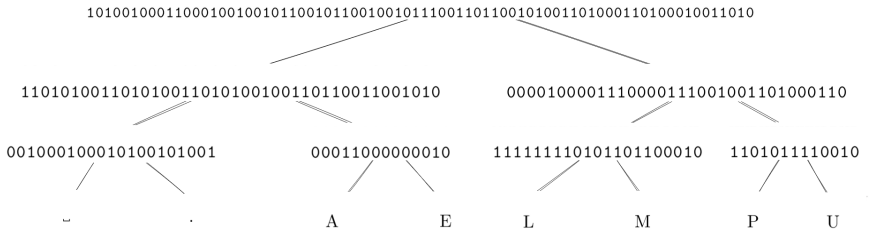



A rekurzívne postupujeme v delení abecedy na polovicu. Celý wavelet strom pre náš príklad bude vyzerať nasledovne:




Každý list reprezentuje jedno písmeno (abecedu veľkosti 1) a každý vnútorný vrchol reprezentuje množinu písmen $\Sigma_v = \Sigma_{\text{left}(v)} \cup \Sigma_{\text{right}(v)}$ a reťazec $S_v = S|\Sigma_v$, teda obmedzenie pôvodného textu len na symboly z danej podabecedy.

A čo je najlepšie? Ukážeme, že text nad bitvektormi, na obrázku vyznačený šedou, *nebudeme potrebovať* a *nebude uložený*. Štruktúra v pamäti bude vyzerať nejak takto:




**Koľko pamäte zaberá wavelet strom?** Nuž v koreni máme presne $n$ bitov. Na druhej úrovni máme... Hmmm... Každé písmeno pôvodného textu ide buď doľava alebo doprava, takže spolu sú je druhej úrovni opäť $n$ písmen, tzn. vo vrcholoch stromu bude $n$ bitov. A takto môžeme pokračovať: na každej úrovni bude presne $n$ bitov. A keďže abecedu delíme vždy na polovicu, dostávame perfektný strom s výškou $\lg \sigma$. To znamená, že celkovo všetky bitvektory zaberajú

$$n \lg \sigma \text{ bitov.}$$

Uvedomme si, že ak máme abecedu veľkosti $\sigma$, tak na zápis jedného znaku potrebujeme $\lg \sigma$ bitov. To znamená, že celý pôvodný text zaberal tiež $n \lg \sigma$ bitov! Pri wavelet strome potrebujeme ešte trochu pamäte navyše na smerníky, avšak väčšinou ide o zanedbateľnú veľkosť a pamäť potrebná na celý wavelet strom je zhruba rovnaká ako pamäť pre pôvodný text.

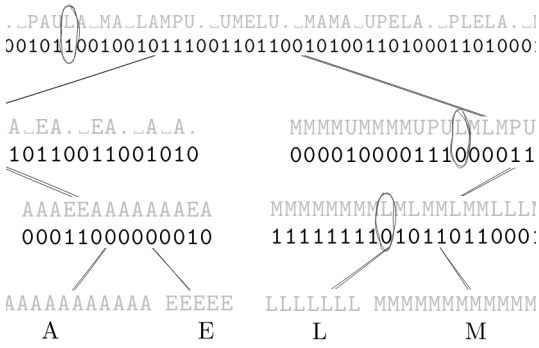
Na wavelet strom sa dá pozerať aj z iného uhla: každý znak $c \in \Sigma$ si môžeme predstaviť ako binárny kód svojej pozície v abecede. Tento kód presne určuje cestu z koreňa do listu stromu, ktorý symbol $c$ reprezentuje – bit 0 znamená „choď doľava“, bit 1 znamená „choď doprava“.

Strom v našom príklade zodpovedá kódovaniu

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ␣ | 000 | A | 010 | L | 100 | P | 110 |
| . | 001 | E | 011 | M | 101 | U | 111 |

A každá úroveň stromu zodpovedá jednému bitu binárneho zápisu znaku. Bitvektor vo vrchole len určuje, ktoré znaky majú v danom bite hodnotu `0` a ktoré `1`. Ak sa teda pozrieme na celú cestu z koreňa po list, postupne čítame bity binárneho kódu daného znaku.

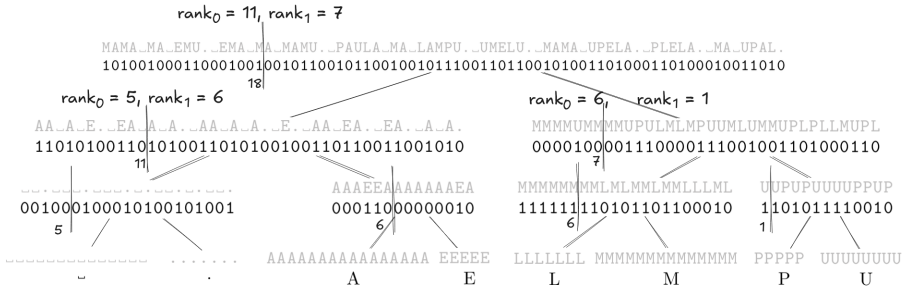
Všimnime si napríklad prvé `L` v texte:




Znak `L` má kód `100`, takže tomuto písmenu prislúcha cesta „doprava, doľava, doľava“, a každý jeden výskyt písmena `L` prispeje práve bitmi `1`, `0`, `0`, do bitvektorov na tejto ceste.

Všeobecne: každý výskyt každého znaku prispeje presne toľkými bitmi, aká je dĺžka jeho kódu. Toto je druhý užitočný spôsob, ako nahliadnuť, že bitvektory zaberú presne takú istú pamäť, ako pôvodný text. Ak si predstavíme pôvodný text kódovaný nejakým binárnym kódom, tak, wavelet strom len ukladá tieto kódy „vertikálne“.

## 21.2 Operácie na wavelet strome




**Operácia `rank`.** Napríklad pri výpočte $\text{rank}_c(i)$ (počet výskytov znaku $c$ pred pozíciou $i$): začneme v koreni s intervalom $[0, i)$ a zisťujeme, či $c$ patrí do ľavej alebo pravej polovice abecedy. Ak $c \in \Sigma_j$, spočítame

$$i' = \text{rank}_j(B, i),$$

kde $B$ je bitvektor v aktuálnom vrchole a $\text{rank}_j$ je operácia `rank` na bite $j \in \{0, 1\}$. Potom rekurzívne pokračujeme na príslušného syna (smerom vľavo alebo vpravo) a hľadáme $\text{rank}_c(i')$. Na liste dostaneme výsledok.

Inými slovami, *dotaz rank sleduje v strome cestu, ktorú by prešiel samotný znak*, pričom na každej úrovni prepočítava aktuálnu pozíciu v zodpovedajúcom podreťazci.

**Operácia `select`.** Ak chceme podporovať aj operáciu `select`, stačí bitvektory v každom vrchole predspracovať aj pre `select`. Výpočet $\text{select}_c(i)$ je potom presne inverzný k výpočtu $\text{rank}_c(i)$.

Začneme v liste zodpovedajúcom písmenu $c$. Zaujíma nás, kde sa nachádza $i$-ty výskyt znaku $c$ v pôvodnom texte. Rekurzívne sa teda posúvame smerom nahor po ceste k koreňu. Nech pre rodiča daného vrcholu platí, že $c \in \Sigma_j$. Potom si spočítame

$$i' = \text{select}_j(B, i),$$

kde $B$ je bitvektor rodiča, a pokračujeme rekurzívne volaním $\text{select}_c(i')$ v jeho nadriadenej úrovni. Po návrate do koreňa dostaneme pozíciu $i$-teho výskytu znaku $c$ v pôvodnom texte $S$.

**Operácia `access`.** Pomocou týchto dvoch operácií vieme dokonca rekonštruovať znak na ľubovoľnej pozícii textu, bez potreby uchovávať samotný reťazec $S$. Operácia `access`(i) teda funguje nasledovne:

Začneme v koreni s pozíciou $i$. Nech $j = B[i]$ – teda či znak na tejto pozícii patrí do ľavej alebo pravej polovice abecedy. Spočítame

$$i' = \text{rank}_j(B, i),$$

a pokračujeme s hľadaním $S_j[i']$ v príslušnom podstrome (ľavom, ak $j = 0$, pravom, ak $j = 1$). Na liste stromu už jednoznačne poznáme písmeno, takže vrátime príslušný znak abecedy.

Inými slovami: *ak zostrojíme wavelet strom, samotný text už vôbec nepotrebujeme – dá sa z neho kedykoľvek zrekonštruovať.*

## 21.3 Implementačné detaily a zložitosť

Na záver ešte ostáva rozhodnúť niekoľko praktických detailov:

- ako budeme deliť abecedu v každom vrchole,
- ako budeme reprezentovať samotný strom,
- a ako budeme ukladať jednotlivé bitvektory.

Najprirodzenejším riešením je deliť abecedu v každom kroku na dve približne rovnaké polovice. Takto dostaneme perfektne vyvážený binárny strom, ktorého výška je $\lg \sigma$. Každý znak textu sa teda objaví práve raz na každej úrovni stromu, čo znamená, že celková dĺžka všetkých bitvektorov v jednej úrovni je vždy presne $n$.

Ak bitvektory z jednotlivých úrovní jednoducho zreťazíme za sebou do jedného veľkého poľa, dostaneme veľmi kompaktnú reprezentáciu, ktorú vieme spracovávať aj sekvenčne (napríklad pri konštrukcii). Na každý znak tak pripadá $\lg \sigma$ bitov informácie, preto celková pamäťová náročnosť je

$$n \lg \sigma + o(n \lg \sigma) \text{ bitov},$$

kde člen v malom $o$ zodpovedá pomocným štruktúram pre rýchly `rank` a `select`.

Operácie `access`, `rank` a `select` prechádzajú od koreňa po list, resp. od listu po koreň, a teda trvajú $O(\log \sigma)$ krokov. Keďže na každej úrovni spravíme len konštantné množstvo práce a všetky bitvektory vieme obsluhovať v čase $O(1)$, celková časová zložitosť týchto operácií je $O(\log \sigma)$.

Výsledkom je veľmi elegantná dátová štruktúra: wavelet strom generalizuje bitvektor s operáciami `rank` a `select` na ľubovoľnú abecedu, pričom si zachováva rovnaké asymptotické vlastnosti.

## 21.4 Komprimovaný wavelet strom

Druhou, ešte úspornejšou možnosťou je reprezentovať všetky bitvektory vo wavelet strome pomocou RRR štruktúry, teda komprimovane podľa ich entropie. Takto získame *komprimovaný wavelet strom*, ktorý zaberá iba $nH(S)+o(n \log \sigma)$ bitov, kde $H(S)$ je empirická entropia textu $S$.

Pozrime sa, prečo to platí. Nech $n_0$ a $n_1$ označujú počty núl a jednotiek v bitvektore koreňa, a nech $n_{00}, n_{01}$ sú počty núl a jednotiek v ľavom podstrome, zatiaľ čo $n_{10}, n_{11}$ sú ich počty v pravom podstrome.

RRR reprezentácia v koreni potrebuje približne

$$n_0 \lg \frac{n}{n_0} + n_1 \lg \frac{n}{n_1} = n_{00} \lg \frac{n}{n_0} + n_{01} \lg \frac{n}{n_0} + n_{10} \lg \frac{n}{n_1} + n_{11} \lg \frac{n}{n_1} \qquad (*)$$

bitov. Bitvektory v oboch deťoch potom zaberú

$$n_{00} \lg \frac{n_0}{n_{00}} + n_{01} \lg \frac{n_0}{n_{01}} + n_{10} \lg \frac{n_1}{n_{10}} + n_{11} \lg \frac{n_1}{n_{11}}. \qquad (\dagger)$$

Ak oba tieto výrazy $(*)$ a $(\dagger)$ sčítame, krásne sa teleskopicky spoja – napríklad

$$n_{00} \left( \lg \frac{n}{n_0} + \lg \frac{n_0}{n_{00}} \right) = n_{00} \lg \left( \frac{n}{\cancel{n_0}} \times \frac{\cancel{n_0}}{n_{00}} \right) = n_{00} \lg \left( \frac{n}{n_{00}} \right)$$

a rovnako pre ďalšie tri členy. Po algebraickom usporiadaní dostaneme, že súčet prvých dvoch úrovní spolu zaberá

$$n_{00} \lg(n/n_{00}) + n_{01} \lg(n/n_{01}) + n_{10} \lg(n/n_{10}) + n_{11} \lg(n/n_{11}).$$

Nie je ťažké nahliadnuť, že ak budeme takto pokračovať až po listy, dostaneme všeobecný tvar:

$$\sum_c n_c \log \frac{n}{n_c},$$

kde $n_c$ je počet výskytov znaku $c$ v texte. Tento súčet je práve $nH(S)$, kde $H(S)$ je empirická entropia textu $S$.

**Reprezentácia komprimovaného wavelet stromu.** Pri nekomprimovanom wavelet strome majú všetky bitvektory rovnakú dĺžku $n$, takže ich môžeme jednoducho zreťaziť do jedného poľa. Pri kompresnom variante to už neplatí: každý bitvektor je uložený pomocou RRR štruktúry, ktorá zaberá len toľko bitov, koľko zodpovedá jeho entropii. Dĺžky jednotlivých bitvektorov teda nie sú rovnaké a preto potrebujeme navyše informáciu, kde sa ktorý z nich v pamäti nachádza.

Existujú dve základné možnosti, ako takúto štruktúru reprezentovať:

- **Explicitná stromová reprezentácia.** Každý vrchol obsahuje smerníky na svojho otca a synov (ľavého aj pravého), spolu s vlastným RRR bitvektorom. Takéto riešenie je veľmi prehľadné a umožňuje jednoduchú navigáciu, no prináša určitý overhead v podobe smerníkov a rozbitia dátovej lokality v pamäti.

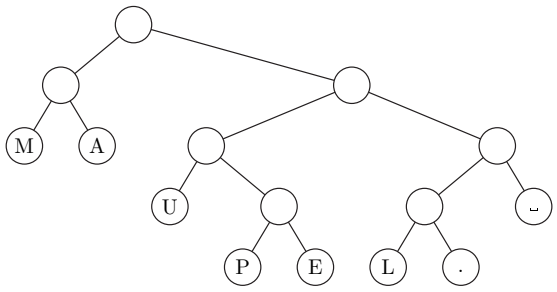
- **Zreťazená reprezentácia.** Všetky RRR štruktúry uložíme sekvenčne za sebou v jednom veľkom poli bitov a pre každý vrchol si zapíšeme jeho *offset*, teda pozíciu, kde sa jeho bitvektor v tomto poli začína. Tieto offsety môžeme uložiť v samostatnom poli dĺžky rovnej počtu vrcholov. Takto dostaneme veľmi kompaktné a cache-priateľské usporiadanie, vhodné najmä pre statické texty a indexy.

## 21.5 Wavelet strom v tvare Huffmanovho kódu

Doteraz sme predpokladali, že abecedu $\Sigma$ delíme vždy na polovicu, teda že každý znak má kód rovnakej dĺžky $\lceil \log \sigma \rceil$. Takýto strom je perfektne vyvážený, čo zabezpečuje rovnaký počet krokov pre všetky znaky.

Môžeme však ísť ešte ďalej: nemusíme vyžadovať, aby mal každý znak kód rovnakej dĺžky. Namiesto rovnomerného delenia abecedy môžeme použiť *Huffmanov kód*, ktorý priraďuje kratšie kódy častejšie sa vyskytujúcim znakom a dlhšie kódy tým zriedkavejším. Tým dostaneme *Huffmanov wavelet strom*.

| *znak* | *frekvencia* | *kód* |
|---|---|---|
| `A` | 16 | `01` |
| `M` | 14 | `00` |
| `␣` | 14 | `111` |
| `U` | 8 | `100` |
| `.` | 7 | `1101` |
| `L` | 7 | `1100` |
| `E` | 5 | `1011` |
| `P` | 5 | `1010` |




Obr. 21.1: Huffmanov kód pre text `MAMA␣MA␣EMU.␣EMA␣MA␣MAMU.␣PAULA␣MA␣LAMPU.␣UMELU.␣MAMA␣UPELA.␣PLELA.␣MA UPAL.` Optimálny kód priradí častým znakom ako `A` a `M` *kratší* kód a ako menej sa vyskytujúcim `E` a `P`.

Takto zostrojený strom má zaujímavé vlastnosti:

- Ak aj jednotlivé bitvektory reprezentujeme priamo (teda bez kompresie pomocou RRR), celková pamäťová náročnosť bude už len $nH(S)$, teda presne zodpovedá entropii textu $S$.

- Výška stromu nie je konštantná, takže v najhoršom prípade môže byť čas jednej operácie až $O(\sigma)$, ak existuje veľmi zriedkavý znak s extrémne dlhým kódom.

- V priemernom prípade však výška stromu (a teda aj čas na operácie `access`, `rank`, `select`) zodpovedá priemernej dĺžke kódov v Huffmanovom kóde, čo je práve $O(H(S))$, pričom $H(S) \leq \log \sigma$.

Inými slovami, Huffmanov wavelet strom sa automaticky prispôsobí rozdeleniu frekvencií symbolov v texte: časté znaky majú kratšie cesty, zriedkavé dlhšie, a celková veľkosť štruktúry sa blíži entropickému optimu. Tým spája výhody kompresie a indexovania v jednej elegantnej dátovej štruktúre.

## 21.6 FM-index ešte raz

## 21.7 Geometria

Wavelet stromy boli pôvodne vynájdené ako dátová štruktúra na reprezentáciu textu – na riešenie úloh na reťazcoch. Preto vás možno príjemne prekvapím, keď vám prezradím, že wavelet stromy majú využitie aj na riešenie úloh v geometrii!

Ako je to možné? Pri reťazcoch máme jednorozmernú postupnosť, ktorá sa skladá z diskrétnych znakov. Každý znak je vybraný z nejakej konečnej abecedy. Naproti tomu v geometrii máme viac rozmerov a body môžu byť ľubovoľné reálne čísla!

Vrátime sa konkrétne k úlohe počítania bodov v obdĺžniku (*orthogonal range query*), ktorú sme riešili v 13. kapitole pomocou kd-stromov a rozsahových stromov. Na vstupe je teda daných $n$ bodov, máme čas na predpočítanie a následne budú prichádzať otázky „koľko bodov je v danom obdĺžniku?“

Ako sa na riešenie tejto úlohy dá použiť wavelet strom?? Kde máme aký text?

Riešením je nasledujúca finta: Vezmeme všetky $x$-ové súradnice bodov a zotriedime ich. Vezmeme všetky $y$-ové súradnice bodov a aj tie zotriedime. Každému bodu potom môžeme priradiť *nové* súradnice, ktoré budú celé čísla od 1 do $n$.

Môžeme si to predstaviť aj tak, že cez každý bod vedieme vodorovnú a zvislú priamku a tieto priamky nám celú plochu rozdelia na diskrétnu mriežku. Vodorovné aj zvislé priamky si očíslujeme zľava doprava / zhora nadol od 1 po $n$.

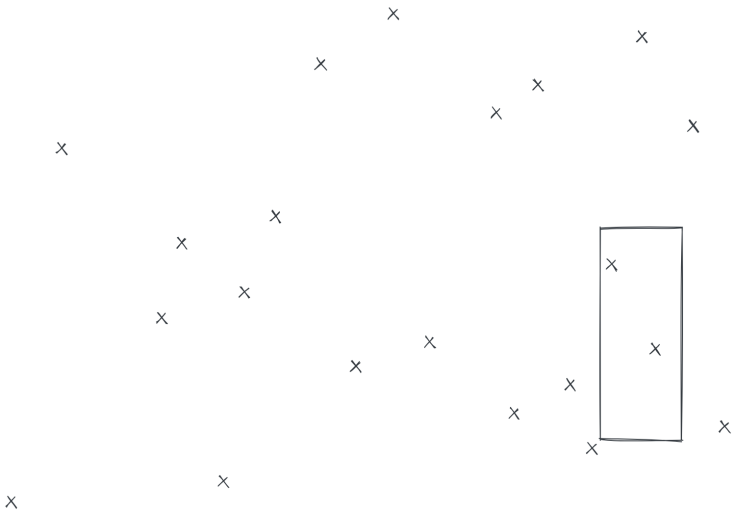
Ak potom dostaneme otázku na nejaký obdĺžnik, stačí pre súradnice jeho krajných bodov pomocou binárneho vyhľadávania zistiť, do ktorých políčok mriežky patria. V skutočnosti totiž nezáleží na nekonečnej presnosti, a či je bod trošku viac vľavo alebo trochu vpravo. Záleží iba na tom, do ktorého políčka mriežky bod patrí.

Takto dokážeme spojitú úlohu s reálnymi číslami redukovať na úlohu na diskrétnej mriežke, kde súradnice bodov sú celé čísla od 1 do $n$.

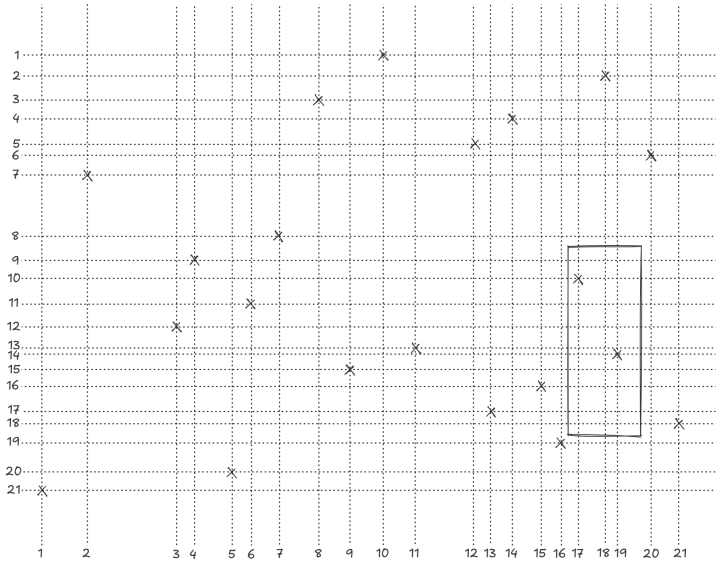
Ak si body zoradíme zľava doprava a v tomto poradí vypíšeme poradia $y$-ových súradníc, dostaneme reťazec dĺžky $n$ nad abecedou $\{1, 2, \ldots, n\}$. Pozícia v texte reprezentuje $x$-ovú súradnicu, a jednotlivé znaky reprezentujú $y$-ovú súradnicu.

Z tohto reťazca postavíme wavelet strom. Keďže abeceda je veľkosti $n$, bude mať výšku $\lg n$ a zaberať $O(n \lg n)$ bitov pamäte – to je $O(n)$ registrov(!). Listy $1, 2, \ldots, n$ predstavujú $y$-ové súradnice a všimnite si, že každý vnútorný vrchol reprezentuje nejaký rozsah $y$-ových súradníc, teda vodorovný „slíž“. Koreň reprezentuje celú plochu, ľavý syn hornú polovicu, pravý syn dolnú polovicu.

Algoritmus pre počítanie bodov v obdĺžniku, resp. počet znakov z daného rozsahu $[c_{\min}, c_{\max}]$ v podreťazci $T[i..j]$ je jednoduchý: Začneme v koreni a rekurzívne hľadáme v ľavom a pravom podstrome, pričom si udržiavame pozíciu v reťazci $[i..j]$. Pri zostupe o úroveň nižšie sa veľkosť abecedy zmenší vždy na

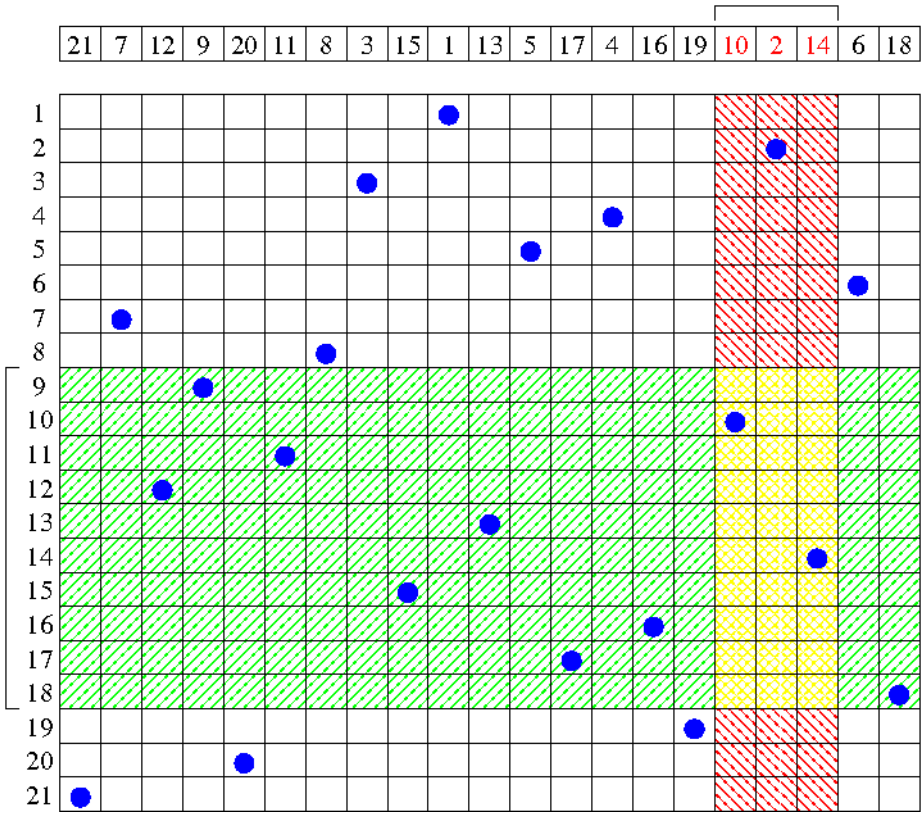


(a) Geometrická úloha: Aký je počet bodov v danom obdĺžniku?

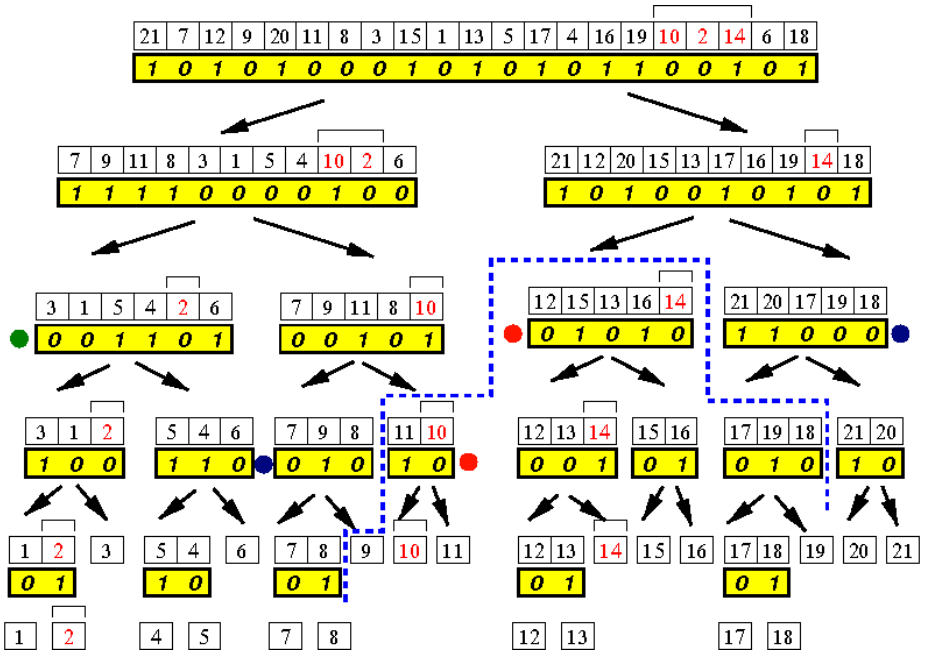



(b) Množinu bodov budeme reprezentovať ako mriežku. Body zotriedime zľava doprava a zdola nahor a súradnice očíslujeme číslami od 1 do $n = 21$. Keď sa pýtame, koľko bodov je v danom obdĺžniku na obrázku, úlohu môžeme preformulovať ako: koľko bodov má (v nových súradniciach) $x$-ovú súradnicu medzi $17 \ldots 19$ a $y$-ovú medzi $9 \ldots 18$ (vrátane)?

Obr. 21.2




Obr. 21.3: Body z obrázku 2 môžeme reprezentovať ako reťazec. Stačí vypísať $y$-ové súradnice bodov zľava doprava: 21, 7, 12,.... Úlohou potom zistiť, koľko znakov v podreťazci na pozíciach 17...19, sú znaky v rozsahu 9...18?




polovicu. Vyhľadávanie môžeme skončiť skôr, ak v prehľadávanom vrchole

1. v intervale $[i..j]$ nie sú žiadne znaky,
2. vrchol zodpovedá rozsahu znakov $[c_1, c_2]$, ktorý sa neprekrýva s $[c_{\min}, c_{\max}]$,
3. alebo naopak, ak vrchol zodpovedá rozsahu znakov $[c_1, c_2]$, ktorý je celý vnútri $[c_{\min}, c_{\max}]$ – v tomto prípade treba všetky znaky na pozíciach $i..j$ započítať.

Ako sme si ukázali v kapitole 13, každý interval vieme poskladať z najviac $2 \lg n$ podstromov, takže tento algoritmus navštívi iba $O(\log n)$ vrcholov a v každom vrchole spraví len konštantne veľa práce (pri prechode na nižšiu úroveň treba počítať ranky).

# Kapitola 22

# FM-index ešte raz

# Časť VIII

# Externé dátové štruktúry

# Kapitola 23

# Triedenie a vyhľadávanie

Začnime príkladom. Máme súbor dát, ktoré chceme preusporiadať. Pre každý prvok $d_i$ už poznáme jeho cieľovú pozíciu $\pi_i$ – teda permutáciu $\pi$ čísel $1, \ldots, N$, ktorá určuje, kam má ktorý prvok patriť. Otázka znie:

Ako tieto dáta efektívne usporiadať podľa $\pi$?

Úplne najpriamočiarejšie riešenie je napísať obyčajný cyklus, ktorý každý prvok jednoducho presunie na svoje miesto:

```
for i in 0..N:
  a[pi[i]] = d[i]
```

Na prvý pohľad to vyzerá nevinne – čítame prvky, zapisujeme ich inde, hotovo. Lenže ak sú dáta uložené na disku, môže to byť jedna z najhorších vecí, aké môžete svojmu počítaču spraviť. Ak si spomínate na klasické pevné disky (hard disky – tie prastaré technológie s rotujúcimi platňami, kde sa čítacia hlava musí najskôr presunúť na správnu stopu a potom čakať, kým sa pod ňu dostane správny sektor), viete, že *sekvenčné čítanie* je veľmi rýchle – ale *náhodný prístup* (random access) je extrémne pomalý.

Súčasné disky (rok 2025) dokážu pri sekvenčnom čítaní prenášať stovky megabajtov za sekundu, kdežto pri náhodnom prístupe je to často len v jednotkách až desiatkach MB/s. Sekvenčný prístup je teda typicky *10- až 100-krát rýchlejší* než náhodný. Moderné SSD disky túto priepasť síce výrazne zmenšili – no aj pri nich zostáva sekvenčný prístup násobne rýchlejší než úplne náhodný.

## 23.1 Model externej pamäti

Aby sme mohli analyzovať algoritmy v prostredí, kde nie je prístup do pamäte rovnomerne rýchly, zavedieme *model externej pamäti*, známy tiež ako *I/O model*, *external memory model*, *disk access model* alebo niekedy aj *cache-aware model*.

- Máme *neobmedzenú externú pamäť* (disk) a *rýchlu cache* veľkosti $M$ (typicky hlavná pamäť).

- *Počítať* môžeme len s údajmi, ktoré sa nachádzajú v cache; všetko ostatné musí byť najprv načítané z externej pamäte.
- Externá pamäť je rozdelená na *bloky* po $B$ slovách (alebo bajtoch). Keď čítame alebo zapisujeme, vždy sa prenáša celý blok – nie jednotlivé slová.
- Popri klasickej časovej a pamäťovej zložitosti algoritmu nás preto bude zaujímať aj jeho *I/O zložitosť*: počet prenosov blokov (čítaní + zápisov) medzi externou pamäťou a cache.

Inými slovami, I/O model meria, *koľkokrát musíme ísť na disk*. Každý takýto prístup je extrémne drahý a preto práve tento počet často určuje reálnu rýchlosť algoritmu.

Je zrejmé, že:

$$\text{I/O zložitosť} \leq \text{časová zložitosť},$$

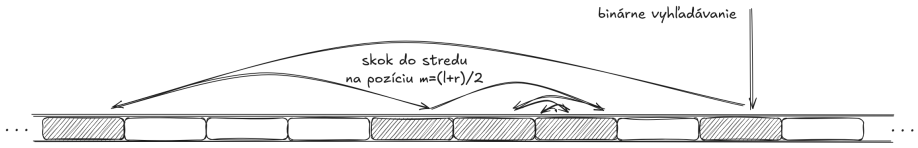
pretože v najhoršom prípade môže každá operácia spôsobiť cache miss a tým jeden prenos. Naopak,

$$\text{I/O zložitosť} \geq \frac{\text{minimálny počet prístupov}}{B},$$

pretože každý prenos načíta aspoň jeden blok veľkosti $B$.

## 23.2 Vyhľadávanie

**Riešenie #1: binárne vyhľadávanie.** Prvý nástrel: klasické riešenie na hľadanie v utriedenom poli je, samozrejme, *binárne vyhľadávanie*.




Hľadaný prvok nájdeme pomocou $O(\log N)$ porovnaní a dokonca vieme, že tento algoritmus je optimálny – hľadanie s menším počtom porovnaní (v najhoršom prípade) jednoducho nie je možné. V našom prípade nás však nezaujíma počet porovnaní, ale *počet prístupov na disk*, a v tomto ohľade je binárne vyhľadávanie, jemne povedané, suboptimálne.

Binárne vyhľadávanie totiž veľmi „skáče“: takmer na každé porovnanie bude treba načítať nový blok z pamäte. Takto postupne zmenšujeme interval, v ktorom hľadáme, až kým jeho dĺžka neklesne na veľkosť jedného bloku. Vtedy sa celý zvyšný interval zmestí do najviac dvoch susedných blokov (aj keď nie sú presne zarovnané, v najhoršom prípade v jednom bloku začne a v druhom skončí), ktoré načítame do RAM – a ďalej už netreba siahať na disk.

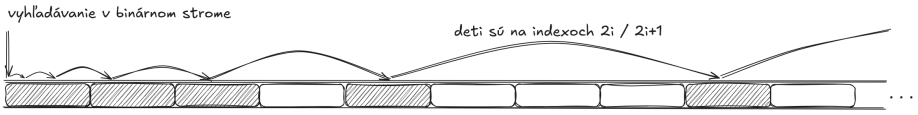
Celkový počet I/O operácií teda vieme odhadnúť ako

$$O(\log(N/B)) = O(\log N - \log B)$$

prístupov na disk.

Dá sa to lepšie?

**Riešenie #2: binárny vyhľadávací strom.** Druhý klasický prístup je zostrojiť si binárny vyhľadávací strom. Predpokladajme, že je perfektne vyvážený a že si ho uložíme na disk *po úrovniach zľava doprava*, podobne ako pri reprezentácii binárnej haldy. Ak sme vo vrchole na pozícii $i$, jeho deti sa nachádzajú na pozíciách $2i$ a $2i + 1$.




Akú má takýto prístup I/O zložitosť?

Prvých približne $\lg B$ skokov sa ešte odohráva v rámci jedného bloku, pretože všetky uzly najvyšších úrovní sa zmestia do jedného bloku. Lenže dĺžka skokov sa exponenciálne zväčšuje a od určitého momentu sú už skoky väčšie ako veľkosť bloku $B$. Každý ďalší zostup o úroveň nižšie znamená načítanie nového bloku z disku.

Celková I/O zložitosť teda opäť vychádza na

$$O(\log(N/B)) = O(\log N - \log B)$$

prístupov na disk.

Slabé. To sa naozaj nedá lepšie?

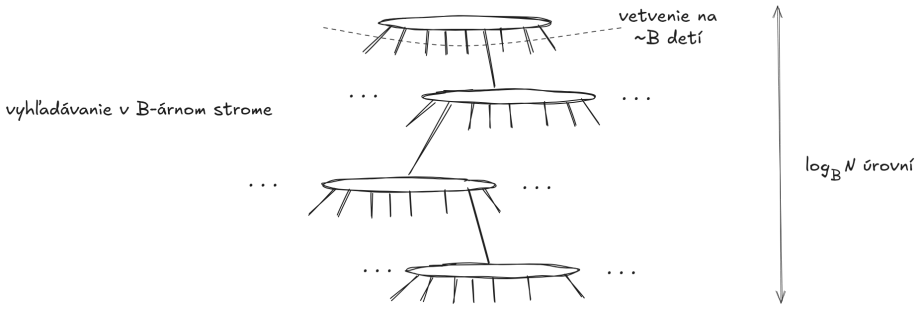
Dá – stačí si uvedomiť, že nikde nie je napísané, že strom musí byť *binárny*.

**Riešenie #3: B-strom.** Namiesto binárneho stromu použijeme zhruba $B$-árny, kde $B$ zodpovedá veľkosti bloku.[1] Každý vrchol tak bude uložený v $O(1)$ susedných blokoch, a keďže strom má výšku $\log_B N$, stačí nám len

$$O(\log_B N) = O(\log N / \log B)$$

prístupov na disk.

[1] Predpokladajme, že prvky majú konštantnú veľkosť, takže do jedného bloku veľkosti $B$ sa zmestí $\Theta(B)$ prvkov.




V praxi väčšinou zvolíme vetvenie tak, aby zodpovedalo maximálnemu počtu prvkov, ktoré sa do jedného bloku ešte zmestia, a celé rozloženie vrcholov zarovnáme na začiatky blokov. Vyhneme sa tak nepríjemnej situácii, keď vrchol „prečnieva“ do dvoch susedných blokov a museli by sme ich načítať oba.

Ďalšie praktické vylepšenie, najmä ak reprezentujeme slovník, v ktorom má každý kľúč pridruženú väčšiu hodnotu, je tzv. $B^+$*-strom*. Myšlienka je jednoduchá: všetky dáta uložíme iba v listoch, zatiaľčo vnútorné vrcholy budú obsahovať len kópie kľúčov, ktoré nás budú navádzať k cieľu. Na rozdiel od klasického $B$-stromu, kde každý vrchol obsahuje aj kľúč, aj hodnotu, teraz vnútorné uzly obsahujú *len* kľúče. Vďaka tomu sa do jedného bloku zmestí viac kľúčov, strom je plytší a vyhľadávanie ešte efektívnejšie.

Navyše, všetky reálne dáta sa nachádzajú v listoch, ktoré sú na disku uložené v poradí podľa kľúčov. Prechod cez interval hodnôt zľava doprava je tak možné vykonať jednoduchým sekvenčným čítaním blokov – a to je presne to, čo má disk najradšej.

Mimochodom, rozdiel medzi $O(\log N - \log B)$ a $O(\log N / \log B)$ je v praxi zásadný. Napríklad ak máme $N = 10^9$ prvkov a blok veľkosti $B = 10^3$, tak $\lg N \approx 30$ a $\lg B \approx 10$. Binárne vyhľadávanie preto potrebuje približne $30 - 10 = 20$ prístupov na disk, zatiaľčo $B$-strom iba $30/10 = 3$(!) To je obrovský rozdiel, ak uvážime, že každý prístup na disk môže trvať milisekundy (pre SSD stovky–desiatky mikrosekúnd), zatiaľčo prístup do RAM trvá desiatky nanosekúnd.

**A nedá sa to ešte lepšie?** Ak sa bavíme o hľadaní *pomocou porovnávaní* (teda bez hešovania, písmenkových stromov, alebo iných štruktúr, ktoré využívajú vnútornú reprezentáciu kľúčov), potom odpoveď znie: *nie*.

Pre jednoduchosť predpokladajme, že hľadaný kľúč sa v našom súbore nenachádza a chceme zistiť jeho pozíciu, tzn. najbližší menší a najbližší väčší prvok. V tomto modeli je každý krok len otázka typu „je hľadaný prvok menší alebo väčší než tento?“, čo nám poskytuje práve jeden bit informácie.

Aby sme našli presnú pozíciu hľadaného prvku v utriedenom poli dĺžky $N$, musíme rozlíšiť medzi $N + 1$ možnosťami, čo si vyžaduje aspoň $\log_2 N$ bitov informácie. Z toho priamo plynie dolná hranica:

$$\Omega(\log N)$$

porovnaní.

Ak namiesto jednotlivých prvkov čítame celé bloky, v ktorých sa nachádza $B$ kľúčov

$$k_0 = -\infty < k_1 < k_2 < \cdots < k_B < k_{B+1} = \infty,$$

tak jediné, čo sa po načítaní bloku dozvieme, je, *do ktorého intervalu* $(k_i, k_{i+1})$ hľadaný prvok patrí. To je $B + 1$ rôznych možností, a teda z jedného načítania môžeme získať najviac $O(\log B)$ bitov informácie.

Keďže spolu potrebujeme $\log N$ bitov a z jedného bloku ich získame najviac $\log B$, potrebujeme aspoň

$$\Omega(\log N / \log B)$$

diskových operácií.

## 23.3 Triedenie

V klasickom RAM modeli sme zvyknutí, že bežné triediace algoritmy ako *heapsort*, *quicksort* alebo *mergesort* triedia $N$ prvkov v čase $O(N \log N)$, čo je optimálne v porovnávacom modeli. Ak by sme ich priamo použili aj v externej pamäti, znamenalo by to $O(N \log N)$ diskových operácií – a to je priveľa. Prirodzene, chceli by sme čosi lepšie.

Dobrá správa je, že všetky tieto algoritmy sa dajú pre účely externého triedenia upraviť. Najťažšie je to pri *heapsorte* – tam potrebujeme úplne inú verziu haldy, ktorá je efektívna aj v externej pamäti. Tomuto problému sa budeme venovať v neskoršej kapitole.

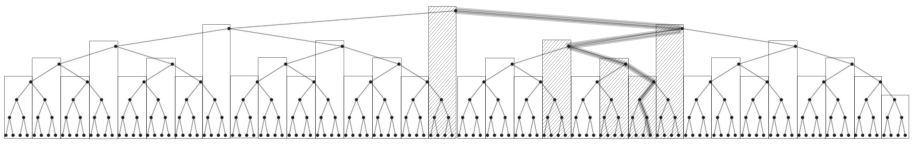
*Quicksort* a *mergesort* sú na tom lepšie, pretože prirodzene pracujú *sekvenčne*. Lineárny prechod $N$ prvkov si vyžaduje len $O(\lceil N/B \rceil + 1)$ prístupov na disk, čo je v podstate ideálne – čítame aj zapisujeme celé bloky naraz.

Vezmime si teda *mergesort*. V klasickej verzii začína s jednotlivými prvkami a postupne ich spája do utriedených behov dĺžky $2, 4, 8, 16, \ldots$. To je však pri externom triedení úplne zbytočné. Oveľa rozumnejšie je načítať naraz *plnú RAM*, teda $\Theta(M)$ prvkov, zotriediť ich interne a výsledok zapísať späť na disk ako jeden utriedený beh. Tento krok stojí len $O(N/B)$ diskových operácií a získame tak približne $N/M$ utriedených behov dĺžky $M$.

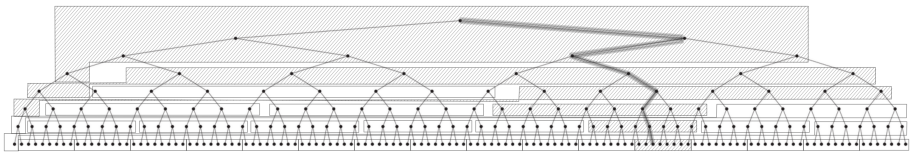
Ostáva už len pospájať tieto dlhé behy. Ak budeme v každej fáze spájať vždy po dvoch, potrebujeme $\lceil \log(N/M) \rceil$ fáz. Každá fáza prechádza všetky dáta sekvenčne (čítanie dvoch behov a zapisovanie výsledku), takže jedna fáza stojí $O(N/B)$ diskových operácií. Celková I/O zložitosť teda bude:

$$O\left(\frac{N}{B} \log \frac{N}{M}\right).$$

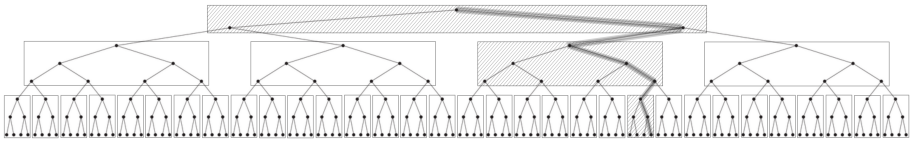
**Dá sa to ešte lepšie?** Veru áno. V predchádzajúcom riešení sme spájali behy *binárne*, vždy po dvoch, ale nič nám nebráni spájať ich viac naraz – vlastne to, *koľko* ich môžeme spájať súčasne, určuje kapacita hlavnej pamäte.



(a) Pri binárnom vyhľadávaní v utriedenom poli je strom implicitný; prvky sú usporiadané ako pri inorder prechode zľava doprava. V jednom bloku sa ocitnú súvislé úseky, teda akoby sme strom rozsekali na vertikálne slíže.



(b) BFS usporiadanie – implicitný binárny strom uložíme po riadkoch. Po zhruba $\log B$ úrovniach je na jednej úrovni viac ako $B$ vrcholov a teda prechod na každú nižšiu úroveň spôsobí cache miss.



(c) B-strom si môžeme predstaviť aj tak, že každý „super-vrchol“ je zložený z malého binárneho podstromu. Do jedného bloku vložíme perfektne vyvážený úplný podstrom $\leq 7$ vrcholov. Podstromy majú výšku zhruba $\Theta(\log B)$ a cesta z koreňa do listu prejde $O(\log N / \log B)$ takýchto podstromov.

Obr. 23.1: Pri vyhľadávaní to vyzeralo, že sme uvažovali rôzne algoritmy (binárne vyhľadávanie vs. binárny strom vs. B-árny strom). V skutočnosti sa menilo len rozloženie dát v pamäti. Každé vyhľadávanie porovnávaním zodpovedá nejakému binárnemu rozhodovaciemu stromu – či už je strom implicitný a pozície vrcholov vieme vypočítať podľa vzorca, alebo je explicitne uložený a medzi vrcholmi skáčeme pomocou smerníkov. Hlavná otázka teda v skutocnosti znie: ktoré vrcholy máme uložiť spolu do blokov tak, aby sme minimalizovali počet I/O operácií. Na obrázku sú bloky veľkosti 8

Do pamäte sa zmestí približne $M/B$ blokov, takže môžeme naraz spájať až $M/B-1$ utriedených behov. Z týchto blokov si vyhradíme $M/B-1$ ako *vstupné buffery*, do ktorých načítame po jednom bloku z každého z behov, a posledný voľný blok použijeme ako *výstupný buffer*.

Mergovanie potom prebieha nasledovne: z aktuálnych blokov vždy vyberieme

najmenší prvok a zapíšeme ho do výstupného bufferu. Keď sa výstupný buffer zaplní, celý ho v jednom kroku zapíšeme na disk. Ak sa niektorý vstupný buffer vyprázdni, načítame z príslušného behu ďalší blok.

Takýmto spôsobom dokážeme spojiť až $M/B-1$ behov naraz, pričom jedna fáza stále vyžaduje len $O(N/B)$ diskových operácií. Počet fáz sa tak výrazne zníži: pri binárnom spájaní (po dvoch) sa po každej fáze počet behov zmenší na polovicu, pri $k$-násobnom spájaní sa zmenší $k$-krát. Počet fáz je teda približne $\log_{M/B}(N/M)$ a celková I/O zložitosť externého mergesortu je

$$O\left(\frac{N}{B}\log_{M/B}\frac{N}{M}\right).$$

(Ak $N/M = (M/B)^k$, tak $N/B = (M/B)^{k-1}$, takže táto zložitosť sa často zapisuje aj v ekvivalentnom tvare $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$.)

V praxi, ak máme bloky veľkosti povedzme 4 kB a operačnú pamäť s kapacitou niekoľkých gigabajtov, dokážeme naraz spájať rádovo *tisícku behov*. Uvedomte si, že každý z týchto behov má sám o sebe niekoľko gigabajtov, pretože vznikol tak, že sme do pamäte RAM načítali maximum dát, ktoré sa tam zmestili, a tieto sme interne zotriedili. Výsledkom je, že aj pre vstupy veľkosti rádovo terabajtov stačia v praxi len 2 (slovom: *dva*) prechody diskom: v prvom vytvoríme utriedené postupnosti dĺžky $M$, a v druhom ich všetky spojíme dokopy do jedného veľkého, utriedeného súboru. Faktor $\log_{M/B}\frac{N}{M}$ v praxi často znamená, že každý blok dvakrát načítame a dvakrát zase vypíšeme.

**Praktické vylepšenia.** Ako vlastne spojíme $k$ postupností naraz?

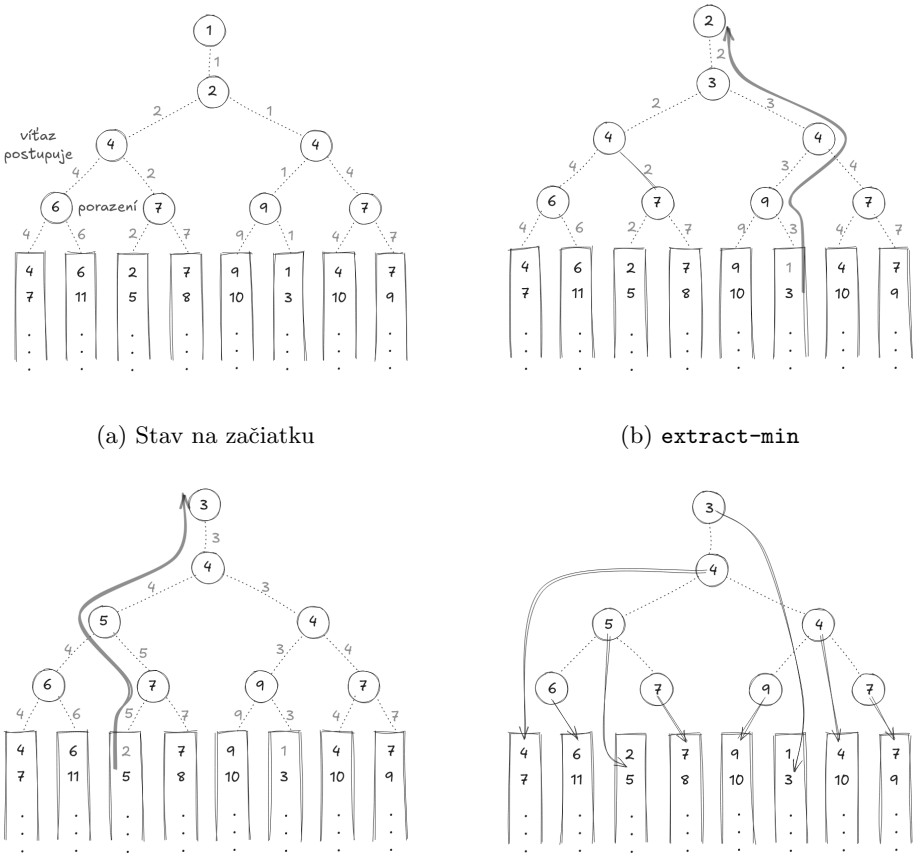
Triviálne riešenie je porovnávať prvky zo všetkých behov priamo, čo by viedlo k zložitosti $O(kN)$ – a to je pre veľké $k$ samozrejme nepoužiteľné. Lepšie je použiť haldu, ktorá nám umožní v každom kroku nájsť najmenší prvok v čase $O(\log k)$. Tým dosiahneme zložitosť $O(N \log k)$,

Ešte o niečo lepšie riešenie a v praxi najpoužívanejšie je použiť tzv. *turnajové stromy* (*loser tree*), pozri obr. 23.2. Myšlienka je, že namiesto obyčajnej haldy si udržiavame binárny strom, ktorý v každom vrchole uchováva „porazeného“ z porovnania svojich dvoch detí. Pri vybraní minima nám stačí prejsť cestu od listu do koreňa a spraviť $\lg k$ porovnaní, zatiaľčo v halde pri bublaní nadol spravíme v najhoršom prípade dvakrát toľko porovnaní.

Druhé vylepšenie je prejsť na *trojpoľný systém*.[2]

V základnej verzii algoritmu, ako sme ho popísali vyššie, najskôr načítavame dáta do pamäte (disk pracuje, CPU sa nudí), potom triedime (CPU maká, disk stojí), a nakoniec zapisujeme výsledok späť na disk (a CPU opäť nič nerobí). Disk aj procesor sa teda väčšinu času striedajú v nečinnosti.

[2]Trojpoľný systém zaviedli v Európe v stredoveku, hoci Číňania ho poznali ešte pred naším letopočtom. Myšlienka bola, že pole sa rozdelilo na tri časti: na jednej sa na jar seje *jarina* (jačmeň, ovos), na druhej časti sa na jeseň zaseje *ozimina* (pšenica) a tretia časť leží úhorom. Jačmeň aj ovos vyžadujú menej živín ako pšenica a dobre rastú aj na pôde, ktorá je po zime ochudobnená. V úhore sa pásol dobytok, ktorý pôdu prirodzene hnojil a do pôdy sa dostával dusík a organická hmota. Ďalší rok sa časti vymenili: tam, kde bol úhor, sa vysiala ozimina, na mieste oziminy jarina a jarina sa nechala úhorom.



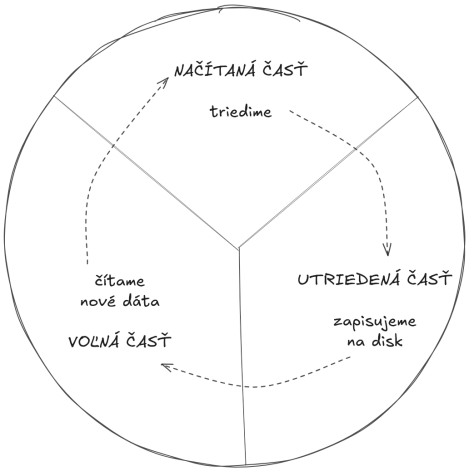

(a) Stav na začiatku

(b) `extract-min`

(c) `extract-min`

(d) Reprezentácia v pamäti.

Obr. 23.2: Spájame 8 postupností na obrázku. Potrebujeme vždy porovnať prvky zo začiatku každého behu a vybrať minimum. a) Predstavme si, že prvky zo začiatku každého behu hrajú turnaj. V prvom kole sa stretnú 4 vs. 6, 2 vs. 7, atď. Prvky 6 a 7 prehrajú (zapíšeme ich do vrcholu stromu) a 4 a 2 pokračujú do ďalšieho kola (zapísali sme ich na hrane do rodiča). Vo finále sa stretnú 2 vs. 1, čo vyhrá 1 (zo 6. postupnosti), ktorá je minimum. b) Čo sa stane, keď 1 odstránime? Na jej miesto nastúpi ďalší prvok zo 6. behu, číslo 3. Nemusíme opakovať celý turnaj, stačí si prejsť cestu predchádzajúceho víťaza: Nad 9 a 4 vyhrá aj 3, takže sa dostane do finále, kde ju porazí 2. c) Keď odstránime 2, ktorá pochádza z 3. behu, na jej miesto nastúpi 5. Vyhrá nad 7, ale 4 ju porazí. Tým pádom 4 postúpi do finále, kde ju porazí 3. d) Reprezenácia *loser tree* v pamäti: pamätáme si len hodnoty porazených a z ktorej postupnosti pochádzajú.




Oveľa lepšie je rozdeliť pamäť na tri časti. V prvej budeme načítavať nové dáta z disku, v druhej triediť tie, ktoré sú už načítané, a v tretej budeme zapisovať utriedené bloky späť. Keď jedna fáza skončí, časti si jednoducho vymenia úlohy: pamäť, ktorú sme práve naplnili, sa začne triediť; pamäť, ktorá sa práve zotriedila, sa začne zapisovať; a pamäť, ktorú sme práve zapísali, sa uvoľní a môže znova slúžiť na načítanie.

Takto dokážeme efektívne *prelínať I/O operácie s výpočtom* – využijeme, že čítanie a zápis medzi RAM a diskom môžu prebiehať paralelne, bez toho, aby do nich musel CPU aktívne zasahovať.

Pri treťom vylepšení sa pozrieme na „hluché“ miesta, ktoré vznikajú počas spájania behov. V predchádzajúcom algoritme, ako sme ho opísali, keď sa výstupný buffer zaplní, celý ho zapíšeme na disk – CPU pritom čaká, kým sa zápis dokončí, sa uvoľní sa miesto a až potom pokračuje ďalej.

V praxi samozrejme máme výstupné buffery aspoň dva a stredame medzi nimi. Keď sa jeden zaplní, procesor pokračuje vo výpočtoch s druhým, kým prvý sa paralelne zapisuje na disk. Kým sa druhý buffer zaplní, prvý je už zvyčajne zapísaný, uvoľnený a pripravený na nové dáta.

Podobný problém však vzniká aj na opačnej strane – pri vstupných bufferoch. Keď sa niektorý z nich vyprázdni, musíme počkať, kým sa z disku načíta ďalší blok, čo opäť spôsobí prestoje. Mohli by sme použiť dvojicu bufferov pre každý beh: keď sa jeden vyprázdni, pokračujeme s druhým, a medzitým do prvého načítame nový blok. Lenže tým by sme efektívne znížili počet behov, ktoré vieme spájať naraz, zhruba na polovicu.

Lepší nápad je trochu prefíkanejší. Z každého vstupného bloku si zapamätáme jeho minimum a číslo behu, z ktorého pochádza. Tým získame krátku pomocnú postupnosť, ktorú si zotriedime a vďaka nej budeme vedieť predvídať budúcnosť! Budeme vedieť poradie, v akom sa jednotlivé buffery vyprázdnia, a teda aj v akom poradí budeme potrebovať načítavať nové bloky.

Potom nám stačí rezervovať len niekoľko voľných blokov, kam budeme priebežne

načítavať budúce dáta. Vždy, keď sa niektorý buffer vyprázdni, jeho náhradník už čaká pripravený v pamäti: prázdny buffer okamžite vymeníme za plný, s plným pokračujeme v spájaní a do prázdneho medzitým načítame ďalší blok, ktorý budeme potrebovať najbližšie.

# Kapitola 24

# Štruktúry nezávislé na cache

V predchádzajúcej kapitole sme pracovali s modelom externej pamäte, v ktorom algoritmus explicitne poznal parametre pamäťovej hierarchie: veľkosť rýchlej pamäte $M$ a veľkosť bloku $B$. Takýto model sa niekedy nazýva aj *cache-aware* – algoritmy sú si „vedomé“ cache, poznajú jej veľkosť aj veľkosť bloku a tieto konštanty môžu použiť pri optimalizácií rozloženia dát v pamäti.

V tejto kapitole sa zamyslíme nad otázkou:

Čo keby sme parametre $M$ a $B$ nepoznali?

Vedeli by sme aj napriek tomu navrhnúť efektívne algoritmy?

Takéto algoritmy sa nazývajú *cache-oblivious*, čo by sme mohli preložiť ako *nezávislé na [parametroch] cache*. Výhodou takýchto algoritmov je, že, keďže nepoznajú parametre cache, musia fungovať dosť dobre pre *ľubovoľnú* veľkosť bloku a veľkosť cache.

V skutočnosti sme sa už s jedným cache-oblivious algoritmom stretli. Pamätáte sa?

Bolo to už v prvej kapitole, pri transponovaní matice. Prvé riešenia rozdelili maticu na bloky a transponovali ju po blokoch. Síce sme vo finále vymieňali tie isté prvky, ale v inom poradí, ktoré bolo oveľa priateľskejšie ku cache. Pri prechode po stĺpcoch sa totiž z celej cache line použije iba jediný prvok a ide sa ďalej – kým sa vrátime k vedľajšiemu prvku, môže táto cache line vypadnúť z cache a musíme ju znovu načítať. Ak maticu rozdelíme na malé bloky, ktoré sa celé zmestia do cache, prvky, ktoré sú blízko seba v pamäti vymieňame aj blízko seba v čase, kým sú ešte stále v cachi. Počet načítaní do cache je tak menší.

Na aké veľké bloky máme maticu rozdeliť? To nie je úplne jasné – odpoveď môže závisieť od veľkosti cache, od veľkosti cache line, ale možno aj od veľa iných parametrov: od rýchlosti prístupu do pamäte, od prefetchera, od cachovacieho algoritmu (kam sa uloží cache line, ktoré cache liny kolidujú, ako sa zvolí cache line, ktorá sa vyhodí, ak je plno). Nakoniec sme veľkosť bloku zvolili skusmo a zadrátovali ju do programu.

Keďže moderné počítače obsahujú viacero úrovní cache (L1, L2, L3), následne sme zistili, že ešte lepší spôsob je deliť maticu na veľké bloky a tie deliť na menšie, respektíve najlepší algoritmus bol taký, čo delí maticu na štyri kvadranty a tie rekurzívne vymieňa (opäť ich deliac na 4). Tento algoritmus nezávisel od veľkosti cache – vďaka rekurzii fungoval dosť dobre pre ľubovoľnú veľkosť. Skúste si rozmyslieť, že počet cache missov bude najviac $O(N^2/B)$, čo je asymptoticky optimálne, pretože potrebujeme $\Theta(N^2)$ prístupov do pamäte.

Čo iné problémy?

V predchádzajúcej kapitole sme riešili vyhľadávanie a triedenie. Optimálne riešenie prvej úlohy je B-strom, resp. $B^+$-strom, lenže ten vyžaduje znalosť $B$. Optimálny triediaci algoritmus je externý mergesort, ktorý v prvej fáze triedi úseky dĺžky $M$ a v druhej fáze spája $M/B-1$ behov naraz – lenže toto riešenie opäť vyžaduje znalosť $B$ aj $M$. Samozrejme, mohli by sme použiť jednoduché binárne vyhľadávanie, alebo binárny mergesort – tieto algoritmy nezávisia od $B$ ani $M$, ale, ako sme videli v predchádzajúcej kapitole, majú oveľa horšiu zložitosť. Existujú algoritmy, ktoré sú efektívne, aj keď nepoznajú $B$ ani $M$?

Na prvé počutie (aj an druhé) to znie takmer nemožne. Napriek tomu prichádza prekvapenie: v mnohých základných úlohách dokážeme v cache-oblivious modeli dosiahnuť *rovnakú asymptotickú I/O zložitosť* ako v klasickom I/O modeli, kde algoritmus parametre $M$ a $B$ pozná a aktívne ich využíva! V tejto kapitole si ukážeme cache-oblivious vyhľadávanie, najskôr v statickom poli, ktoré sa nemení, potom aj cache-oblivious alternatívu k dynamickým B-stromom. V nasledujúcej kapitole sa pozrieme na haldy a triedenie.

## 24.1 Cache oblivious model

Skôr ako sa pustíme do riešenia úloh, poďme si presnejšie popísať model, s ktorým budeme pracovať.

Ide o variant I/O modelu, to znamená, stále predpokladáme, že

- dáta sú uložené v externej pamäti neobmedzenej veľkosti,
- avšak procesor môže počítať iba s dátami v cache.
- Externá pamäť je rozdelená na bloky veľkosti $B$ a prístup do pamäte je vždy po blokoch.

Avšak na rozdiel od klasického *cache-aware* modelu, sa cache-oblivious model líši v nasledujúcich aspektoch:

- Algoritmus nepozná hodnoty $M$ ani $B$. V analýze I/O zložitosti sa síce tieto parametre objavujú, ale samotný algoritmus na nich nijako nezávisí.
- Pamäť (cache) je spravovaná automaticky. Predpokladáme ideálny *offline* algoritmus správy cache, ktorý presne vie, aké dáta budú v budúcnosti potrebné. (Optimálna stratégia je vždy vyhodiť ten blok, ktorý bude znovu použitý v budúcnosti *najneskôr*, tzv. Béládyho algoritmus).

Samozrejme, v reálnych systémoch takýto ideálny algoritmus nemáme k dispozícii, pretože správu cache musíme robiť *online*, bez znalosti budúcnosti. Našťastie sa ukazuje, že jednoduché stratégie používané v praxi, ako FIFO alebo LRU, sú od optimálneho riešenia pomerne blízko.

Presnejšie platí nasledovné tvrdenie: FIFO aj LRU stratégia s cache veľkosti $2M$ vykonajú najviac $O(1)$-krát viac pamäťových prístupov ako optimálny off-line algoritmus, ktorý má k dispozícii cache veľkosti $M$.

Okrem toho v cache-oblivious modeli predpokladáme, že cache je *plne asociatívna*. To znamená, že ľubovoľný pamäťový blok môže byť uložený na ľubovoľné miesto v cache. To je pri stránkovaní (medzi diskom a RAM) pravda. Pri cache (L1, L2, L3 ) v procesoroch je situácia o niečo komplikovanejšia. Väčšina praktických cache nie je plne asociatívna, ale iba *k-asociatívna* pre malé $k$ (typicky $k = 4$, 8 alebo 16). To znamená, že každý blok hlavnej pamäte má v cache iba $k$ možných pozícií, kam sa môže uložiť. Ak sa viacero blokov mapuje do tej istej množiny pozícií, môžu sa navzájom vytláčať, aj keď je cache ako celok ešte zďaleka nie je plná. Z pohľadu teórie však tento detail môžeme abstrahovať, keďže plne asociatívna cache sa dá simulovať pomocou hešovania.

Cache-oblivious model má jednu veľmi príjemnú vlastnosť: keďže algoritmus *nepozná* parametre cache ($M$ ani $B$), je automaticky robustný voči ich voľbe. Ten istý algoritmus funguje dobre pre malú aj veľkú cache, pre rôzne veľkosti blokov, a dokonca aj pre viacúrovňovú pamäťovú hierarchiu (L1, L2, L3, RAM). Nemusíme ho nijako ladiť ani prepisovať pri zmene hardvéru.

Na druhej strane má tento prístup aj svoju cenu. Niektoré cache-oblivious algoritmy majú v praxi pomerne veľkú konštantu skrytú v $O(\cdot)$ a zrejme nemôžeme očakávať, že algoritmus, ktorý nepozná parametre bude lepší ako taký, čo ich pozná.

Napriek tomu, je štúdium cache-oblivious algoritmov užitočné aj z praktického hľadiska: cache-oblivious algoritmus môže slúžiť ako inšpirácia pre cache-aware riešenia.

Z hľadiska teórie je zaujímavé študovať hranice možného.

## 24.2 Van Emde Boasovo usporiadanie

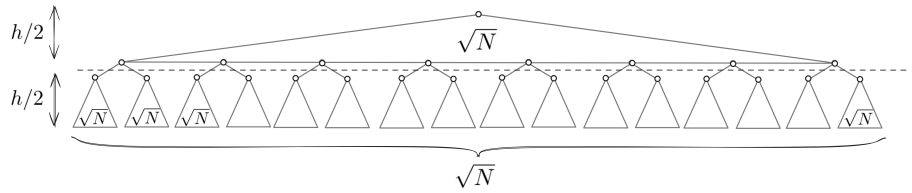
Ako prvý netriviálny príklad cache-oblivious algoritmu si ukážeme, ako vyhľadávať v *statickej* množine veľkosti $N$ v čase $O(\log_B N)$ prístupov do pamäte, teda s rovnakou asymptotickou I/O zložitosťou, akú dosahuje B-strom v externom pamäťovom modeli. Rozdiel je v tom, že tentoraz *nepoznáme* veľkosť bloku $B$.

Použijeme úplný binárny vyhľadávací strom, avšak do pamäte ho neuložíme „naivným“ spôsobom (napr. po úrovniach), ale podľa tzv. *van Emde Boasovho usporiadania (vEB layout)*.

**Základná myšlienka.** Nech má strom výšku $h$ (teda $N = 2^h - 1$ vrcholov). Strom rozdelíme približne v polovici výšky:

- *horný strom* pozostáva z prvých zhruba $h/2$ úrovní,

- *spodné stromy* sú podstromy výšky zhruba $h/2$ zavesené pod listami horného stromu.



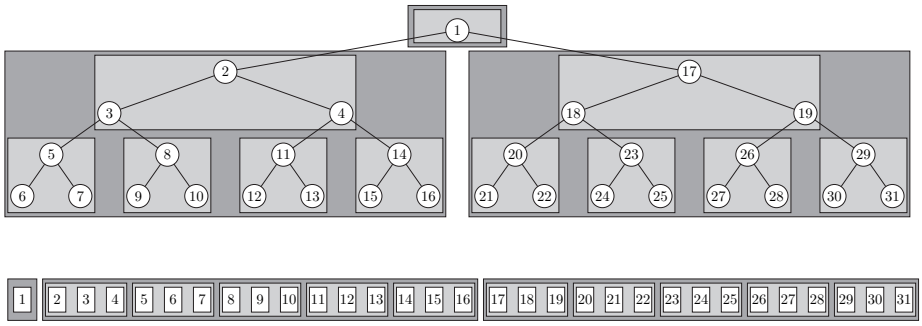

Horný strom má približne $2^{h/2} = \sqrt{N}$ vrcholov. Pod ním sa nachádza zhruba $\sqrt{N}$ spodných stromov, z ktorých každý má opäť veľkosť približne $\sqrt{N}$.

**Rekurzívne uloženie.** Van Emde Boasovo usporiadanie funguje rekurzívne:

1. najprv uložíme celý horný strom do pamäte (rekurzívne v vEB poradí),
2. potom za ním uložíme všetky spodné stromy, každý opäť rekurzívne vo van Emde Boasovom usporiadaní.

Týmto spôsobom dostaneme lineárne uloženie stromu do poľa, ktoré má veľmi silnú lokalitu prístupu: vrcholy, ktoré sú blízko seba v strome, sú aj blízko seba v pamäti.

Tu je príklad pre konkrétny 31 vrcholový strom. Prvé delenie je nie vo výške $h/2$ ale najbližšia menšia mocnica 2:




Napríklad ak by sme mali 4-miliardový strom výšky 32, rozdelíme ho na

**Prečo to funguje.** Pri vyhľadávaní v binárnom strome postupujeme z koreňa k listu po ceste dĺžky $O(\log N)$. Van Emde Boasovo usporiadanie zabezpečí, že každých $O(\log B)$ krokov po strome spôsobí najviac dva nové cache missy. Celkový počet prístupov do pamäte je teda $O(\log N / \log B) = O(\log_B N)$, presne rovnako ako pri B-strome.

Pozrime sa na to bližšie. VEB usporiadanie je definované rekurzívne, takže strom sa na rôznych úrovniach rozkladá na menšie a menšie podstromy. V

analýze sa priblížime na tú úroveň rekurzie, kde sa jednotlivé podstromy už celé zmestia do jedného cache bloku.

Formálnejšie: uvažujme takú úroveň rozkladu, kde má každý podstrom veľkosť $s$ spĺňajúcu

$$\sqrt{B} < s \leq B.$$

Keďže ide o úplné binárne stromy, zodpovedajúca výška takéhoto podstromu $h$ spĺňa

$$\frac{1}{2} \log B < h \leq \log B.$$

Každý takýto podstrom je v pamäti uložený ako *súvislý úsek* dĺžky najviac $B$, takže sa celý zmestí do jedného bloku (prípadne dvoch, ak nie je zarovnaný ideálne).

Pri vyhľadávaní v celom strome prechádzame cestu od koreňa k listu dĺžky najviac $\log N$. Táto cesta sa skladá z úsekov, ktoré ležia v jednotlivých podstromoch výšky $h$. Počet takýchto podstromov, ktorými prejdeme, je teda najviac

$$\frac{\log N}{h} = O\left(\frac{\log N}{\log B}\right).$$

Pre každý z týchto podstromov potrebujeme načítať do cache najviac konštantný počet blokov (najviac dva). Celkový počet prístupov do pamäte je preto

$$O\left(\frac{\log N}{\log B}\right) = O(\log_B N),$$

čo zodpovedá optimálnej I/O zložitosti vyhľadávania.

Zhrnuté: van Emde Boasovo usporiadanie zabezpečí, že vyhľadávacia cesta sa rozpadne na malé kúsky, z ktorých každý má výbornú lokálnosť prístupu k pamäti, a to *bez znalosti* veľkosti bloku $B$.

## 24.3 Usporiadaný súbor

Uvažujme nasledujúci problém. Chceme udržiavať *zoznam prvkov v danom poradí*. Podporované operácie sú:

- vložiť nový prvok medzi dva dané prvky v zozname,
- vymazať daný prvok zo zoznamu.

Ak máme k dispozícii smerníky, riešenie je jednoduché: obojsmerný spájaný zoznam umožňuje vkladanie aj mazanie v čase $O(1)$, za predpokladu, že poznáme miesto, kde operáciu vykonávame. Ak je zoznam navyše *utriedený* a chceme podporovať aj vyhľadávanie, môžeme použiť vyvážený binárny vyhľadávací strom. Ten nám poskytne vkladanie, mazanie aj vyhľadávanie v čase $O(\log N)$.

V oboch prípadoch však používame pointery a dynamickú pamäť, čo nie je veľmi vhodné pre externú pamäť.

Položme si teda ťažšiu otázku:

> Dá sa takýto zoznam reprezentovať *v obyčajnom poli*, bez pointerov, a zároveň podporovať vkladanie a mazanie efektívne?

Ak by sme zoznam uložili ako jeden súvislý úsek v poli, tak každé vloženie alebo vymazanie by si v najhoršom prípade vyžiadalo posun $O(N)$ prvkov, čo je neakceptovateľné.

Na druhej strane, ak by sme medzi susednými prvkami ponechali medzery, mohli by sme vkladať nové prvky bez presúvania. Avšak ak má medzera veľkosť $k$, dokážeme vykonať len $O(\log k)$ vložených prvkov, kým sa zaplní. Problém je, že medzery sa postupne vyčerpávajú a niekde ich bude priveľa, inde primálo.

Na prvý pohľad sa teda zdá, že táto úloha je beznádejne ťažká.

Prekvapivo však existuje relatívne efektívne riešenie. Dá sa udržiavať $N$ prvkov v poli veľkosti $O(N)$ tak, že medzi každými dvoma susednými prvkami je medzera dĺžky najviac $O(1)$, a zároveň vkladanie aj mazanie trvá len $O(\log^2 N)$ amortizovane!

Navyše sa dá ukázať, že za týchto podmienok nie je možné dosiahnuť asymptoticky lepší čas než $\Omega(\log^2 N)$ amortizovane.

Ako bonus má tento algoritmus ešte jednu veľmi príjemnú vlastnosť: každá operácia upravuje iba *súvislý úsek poľa* pomocou konštantného počtu sekvenčných prechodov. Vďaka tomu sa dá prirodzene implementovať aj v *cache-oblivious modeli*, pričom amortizovaná I/O zložitosť je $O\left(\frac{\log^2 N}{B}\right)$.

Základná myšlienka riešenia je nasledovná. Pole, v ktorom udržiavame prvky zoznamu, rozdelíme na malé *bloky* dĺžky $O(\log N)$. Tieto bloky budeme považovať za listy implicitného binárneho stromu.

Nad blokmi si *len konceptuálne* (kvôli vysvetleniu a analýze) vybudujeme úplný binárny strom. Tento strom *nie je súčasťou dátovej štruktúry* a nikde ho explicitne neukladáme – slúži len ako analytický nástroj.

Každý vrchol stromu zodpovedá nejakému súvislému intervalu v pôvodnom poli:

- listy zodpovedajú jednotlivým blokom,
- rodič pokrýva zjednotenie intervalov svojich dvoch synov,
- koreň zodpovedá celému poľu.

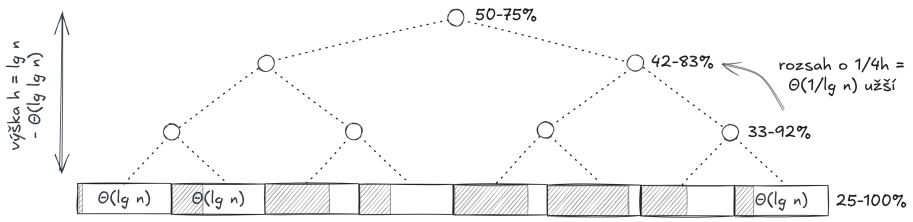
Pre každý vrchol $v$ definujme jeho *hustotu* ako

$$\text{hustota}(v) = \frac{\#\text{prvkov v intervale}}{\text{dĺžka intervalu}}.$$

Nech $d$ je hĺbka vrcholu $v$ v tomto implicitnom strome (koreň má hĺbku $d = 0$) a nech $h = O(\log N)$ je maximálna hĺbka stromu (výška mínus 1).

Cieľom algoritmu je udržiavať hustotu každého vrcholu v určitom povolenom intervale, ktorý závisí od jeho hĺbky. Konkrétne sa budeme snažiť zabezpečiť, aby pre každý vrchol $v$ platilo:

$$\frac{1}{2} - \frac{1}{4} \cdot \frac{d}{h} \;\leq\; \text{hustota}(v) \;\leq\; \frac{3}{4} + \frac{1}{4} \cdot \frac{d}{h}.$$




Pozrime sa, čo toto obmedzenie znamená.

Pre koreň stromu je $d = 0$, takže jeho hustota musí ležať medzi $\frac{1}{2}$ a $\frac{3}{4}$. Inými slovami, celé pole by malo byť zaplnené približne na 50–75%.

Na opačnom konci, pre listy stromu platí $d = h$ a povolený interval hustoty je $\frac{1}{4}$ a 1. To znamená, že jednotlivé bloky môžu byť zaplnené veľmi voľne: od 25% až po úplné zaplnenie.

Pre vrcholy na medziľahlých úrovniach dostávame postupne sa meniace hranice: dolná hranica sa pohybuje od 50% do 25% a horná od 75% do 100%. Čím je vrchol vyššie v strome (bližšie ku koreňu), tým sú obmedzenia na hustotu prísnejšie. Čím je vrchol nižšie, tým sú hranice voľnejšie.

Dôležité je, že tieto hranice interpolujeme *lineárne* podľa hĺbky. Keďže maximálna hĺbka stromu je $h = O(\log N)$, posun o jednu úroveň mení povolený interval hustoty o $1/4h = \Theta(1/\log N)$.

Toto postupné zvoľňovanie podmienok je dôležité – vďaka nemu čím je vrchol vyššie, tým menej často ho musíme upravovať (a medzičasom si zvládne našetriť).

Pozrime sa teraz, ako v tejto štruktúre prebieha vkladanie a vymazávanie prvkov.

**Vkladanie.** Nový prvok chceme vložiť na konkrétnu pozíciu v zozname. Najskôr ho vložíme do príslušného listového bloku a v rámci tohto bloku posunieme prvky o jednu pozíciu, ak je to potrebné. Keďže blok má dĺžku $O(\log N)$, tento krok je lacný.

Ak však po vložení dôjde k tomu, že sa blok *preplní* (t.j. jeho hustota presiahne povolený interval), začneme sa v implicitnom binárnom strome posúvať smerom nahor. Postupne zväčšujeme interval, ktorý berieme do úvahy, a hľadáme *najnižší* vrchol $v$ taký, že po započítaní nového prvku bude jeho hustota opäť v povolených medziach.

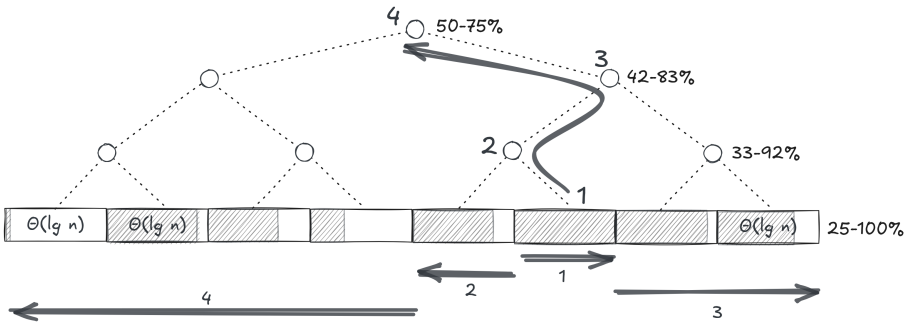
Keď takýto vrchol nájdeme, vezmeme celý interval, ktorý mu zodpovedá, a *prepíšeme ho nanovo*: všetky prvky v tomto intervale rovnomerne rozdistribuujeme po celej jeho dĺžke. Tým sa hustota nielen v tomto vrchole, ale aj vo všetkých jeho potomkoch dostane do povoleného rozsahu.

**Vymazávanie.** Vymazávanie funguje úplne analogicky. Najskôr prvok odstránime z príslušného bloku a v bloku posunieme zvyšné prvky. Ak tým dôjde k porušeniu dolnej hranice hustoty, opäť sa posúvame v implicitnom strome

nahor, až kým nenájdeme najnižší vrchol, ktorého hustota po vymazaní ostáva v povolenom intervale. Interval tohto vrcholu následne nanovo rovnomerne zaplníme.

**Dôležité technické poznámky.** Implicitný binárny strom, o ktorom hovoríme, existuje iba *v našej analýze*. V samotnej implementácii ho explicitne neukladáme. V praxi iba skenujeme pole doľava alebo doprava (podľa toho, či by sme sa v strome pohybovali do ľavého alebo pravého syna), pričom si priebežne udržiavame:

- aktuálnu dĺžku intervalu,
- počet prvkov v ňom (a teda hustotu),
- hĺbku, na ktorej sa nachádzame,
- a z nej vyplývajúci povolený rozsah hustoty.




Ďalšia dôležitá poznámka je, že *listy budú vždy v povolených medziach hustoty*. Naopak, vnútorné vrcholy nemusia byť v každom okamihu korektné. Napríklad, ak by sme do štruktúry vkladali prvky striktne zľava doprava, môžeme zaplniť všetky listové bloky až na 100%. V takom prípade by koreň ani iné vnútorné vrcholy zjavne nespĺňali svoje obmedzenia na hustotu. To však nevadí – tieto porušenia opravíme až v momente, keď dôjde k prvému preplneniu bloku a vyvolá sa redistribúcia na vyššej úrovni.

Platí však opačná implikácia:

> Ak je vrchol v hĺbke $d$ v povolených medziach hustoty, potom sú v povolených medziach hustoty aj *všetky* vrcholy v jeho podstrome.

Navyše, tesne po redistribúcii platí ešte silnejšie tvrdenie. Vrcholy nižšie v strome nie sú len „tak-tak“ v medziach, ale majú *výraznú rezervu*. Ich hustota je ďalej od hraníc povoleného intervalu o približne $O(1/h) = O(1/\log N)$.

**Analýza zložitosti.** Pozrime sa teraz bližšie na amortizovanú zložitosť *vkladania*. Zameriame sa na najdrahšiu operáciu v algoritme – redistribúciu veľkého intervalu.

Ak sa počas vkladania dostaneme do vrcholu $v$ a rozhodneme sa redistribuovať celý interval, ktorý mu zodpovedá, pričom tento interval má dĺžku $k$, potom reálna cena tejto operácie je $O(k)$, pretože musíme celý interval prejsť a prvky nanovo rovnomerne rozmiestniť.

Aby sme dokázali, že algoritmus je efektívny, musíme ukázať, že túto cenu vieme zaplatiť z úspor, ktoré sme si nazbierali počas predchádzajúcich vkladaní.

Po redistribúcii intervalu zodpovedajúceho vrcholu $v$ platí:

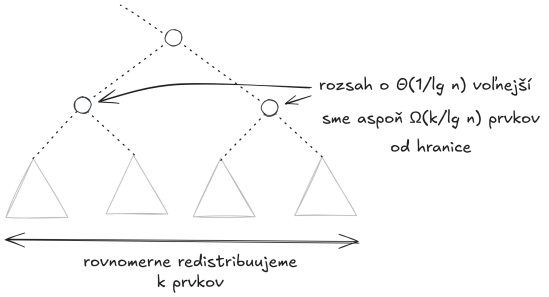
- vrchol $v$ je v povolených medziach hustoty,
- všetky vrcholy v jeho podstrome sú tiež v povolených medziach,
- dokonca majú *rezervu* – ich hustota je vzdialená od hraníc aspoň o $\Omega(1/\log N)$.

Pozrime sa na bezprostredných synov vrcholu $v$. Keďže interval $v$ má dĺžku $k$, jeho synovia majú intervaly dĺžky približne $k/2$. Rezerva hustoty $\Omega(1/\log N)$ v týchto vrcholoch zodpovedá aspoň $\Omega(k/\log N)$ ďalším vkladaniam, ktoré môžu prebehnúť v ich podstromoch, kým sa niektorý zo synov opäť dostane na hranicu a vyvolá ďalšiu redistribúciu na úrovni $v$.

Inými slovami:

> Ak práve redistribuujeme interval dĺžky $k$ vo vrchole $v$, potom sa do tohto vrcholu určite nedostaneme znova skôr, než prebehne aspoň $\Omega(k/\log N)$ ďalších vkladaní v jeho podstrome.

Toto pozorovanie nám umožňuje použiť jednoduchú účtovnícku schému.




Dohodnime sa, že pri každom vložení nového prvku uložíme na účet každého vrcholu na ceste od príslušného listu až ku koreňu $\lg N$ dolárov.

Keďže výška implicitného stromu je $O(\log N)$, jedno vloženie nás stojí spolu $O(\log N) \times \lg N = O(\log^2 N)$ dolárov.

Teraz sa pozrime na redistribúciu vo vrchole $v$ s intervalom dĺžky $k$. Medzi dvoma po sebe idúcimi redistribúciami v tomto vrchole prebehne aspoň

$\Omega(k/\log N)$ vkladaní. Každé z nich prispelo na účet vrcholu $v$ sumou $O(\log N)$. Na účte vrcholu $v$ sa teda medzičasom nazbiera

$$\Omega\left(\frac{k}{\log N}\right) \cdot O(\log N) = \Omega(k)$$

dolárov, čo stačí na zaplatenie redistribúcie intervalu dĺžky $k$.

Každú redistribúciu vieme zaplatiť z kreditov, ktoré sa nahromadili medzi dvoma takýmito operáciami. Preto má vkladanie amortizovanú časovú zložitosť $O(\log^2 N)$.

Dôkaz pre vymazávanie je podobný.

## 24.4 Cache-oblivious B-strom

Teraz už máme všetky stavebné kamene na to, aby sme si poskladali plnohodnotnú *cache-oblivious alternatívu k B-stromom.*

> Skombinujeme *usporiadaný súbor* (packed array) s *binárnym vyhľadávacím stromom* uloženým vo *van Emde Boasovom usporiadaní.*

**Štruktúra.** Prvky uložíme v utriedenom poradí do dátovej štruktúry pre usporiadaný súbor, ktorú sme si práve popísali. Táto štruktúra obsahuje aj prázdne pozície (medzery), ktoré nám umožňujú efektívne vkladanie a mazanie.

Nad týmto poľom si zostrojíme úplný binárny strom:

- listy stromu zodpovedajú jednotlivým pozíciám v usporiadanom súbore (vrátane prázdnych),
- vnútorné vrcholy reprezentujú zjednotenia intervalov svojich podstromov.

Tento strom si uložíme do poľa vo van Emde Boasovom (vEB) usporiadaní. Medzi listom stromu a príslušnou pozíciou v poli vieme skákať v oboch smeroch pomocou jednoduchých prepočtov indexov.

V každom vnútornom vrchole stromu si udržiavame *maximum prvkov v jeho podstrome.* Vďaka tomu môžeme strom používať ako vyhľadávací strom – presne tak, ako v klasickom $B^+$-strome.

**Vyhľadávanie.** Vyhľadávanie prebieha zostupom v binárnom strome: v každom vrchole porovnáme hľadaný kľúč s maximom ľavého podstromu a rozhodneme sa, či pokračovať vľavo alebo vpravo.

Keďže strom je uložený vo vEB-usporiadaní, prechod z koreňa do listu vyžaduje len $O(\log_B N)$ I/O operácií, hoci algoritmus nepozná ani $B$, ani $M$.

**Vkladanie a vymazávanie.** Pri vkladaní alebo vymazávaní najskôr pomocou stromu nájdeme správnu pozíciu v usporiadanom súbore. Samotná operácia v usporiadanom súbore môže spôsobiť redistribúciu intervalu dĺžky $k$, presne tak, ako sme si ukázali v predchádzajúcej časti.

Keď sa takýto interval prepíše, musíme aktualizovať aj informácie vo vnútorných vrcholoch stromu – konkrétne maximá podstromov. To urobíme jedným *inorder prechodom* nad zodpovedajúcim intervalom listov a všetkými ich predkami.

**I/O analýza.** Ukážeme, že vďaka vEB-usporiadaniu si prepísanie intervalu dĺžky $k$ spolu s aktualizáciou všetkých vrcholov nad ním vyžiada len

$$O\left(\frac{k}{B} + \log_B N\right) \text{ I/O operácií.}$$

Keď si spomenieme, že v usporiadanom súbore je amortizovaná hodnota $k = O(\log^2 N)$, dostávame pre vkladanie aj vymazávanie amortizovanú I/O zložitosť

$$O\left(\frac{\log^2 N}{B} + \log_B N\right).$$

**Záver.** Dostali sme plnohodnotnú dynamickú dátovú štruktúru, ktorá:

- podporuje vyhľadávanie v $O(\log_B N)$ I/O,
- podporuje vkladanie a vymazávanie amortizovane v $O\left(\frac{\log^2 N}{B} + \log_B N\right)$ I/O,
- nepozná parametre cache $M$ a $B$,
- a dosahuje asymptoticky rovnaké záruky ako klasické B-stromy v I/O modeli.

Inými slovami: získali sme *cache-oblivious B-strom*.

**Detailná I/O analýza.** Aby sme odhadli počet I/O operácií pri prepísaní intervalu dĺžky $k$ (vrátane aktualizácie všetkých vrcholov nad ním), priblížme si van Emde Boasovo usporiadanie na tú úroveň rekurzie, kde sa celé podstromy zmestia do jedného bloku cache, teda majú veľkosť $\leq B$.

Analýzu rozdelíme na tri časti podľa toho, v ktorej časti stromu sa nachádzame.

**#1 Vrcholy v spodných dvoch úrovniach vEB-podstromov.** Podstromy na najnižšej úrovni vEB-rekurzie sú v pamäti uložené ako súvislé úseky a každý z nich má veľkosť najviac $B$. Na ich načítanie teda potrebujeme najviac dva cache bloky.

Pri inorder prechode sa pohybujeme nasledovne: prechádzame celý jeden podstrom na najnižšej úrovni, potom niekoľko vrcholov v jeho rodičovskom podstrome (o jednu úroveň vyššie), potom opäť celý ďalší podstrom na spodnej

úrovni, potom zasa niekoľko vrcholov na druhej úrovni atď., až kým neprejdeme celý podstrom druhej úrovne.

Ak má cache aspoň štyri bloky ($M \geq 4B$), zmestia sa do nej súčasne dva takéto podstromy zo spodných dvoch úrovní. Kľúčové je, že podstrom na druhej úrovni nevytlačíme z cache skôr, než prejdeme všetky jeho detské podstromy. Výsledkom je, že $O(k)$ vrcholov v spodných dvoch úrovniach prejdeme s celkovou I/O zložitosťou $O(k/B)$.

**#2 Všetky vrcholy nad nimi, až po LCA.** Teraz sa pozrime na vrcholy, ktoré ležia nad najnižšími vEB-podstromami, ale ešte pod najnižším spoločným predkom (LCA) všetkých dotknutých listov.

Podstromy, ktoré sa zmestia do jedného bloku, majú veľkosť aspoň $\sqrt{B}$. Ak sa pozrieme o dve úrovne vyššie, dostaneme stromy veľkosti aspoň $B$. Z toho vyplýva, že koreňov takýchto podstromov môže byť nanajvýš $\lceil k/B \rceil$.

Aj keby každý z týchto vrcholov spôsobil jeden cache miss, celková I/O zložitosť tejto časti je

$$O\left(1 + \frac{k}{B}\right).$$

**#3 Cesta od LCA ku koreňu.** Napokon ostáva aktualizovať vrcholy na ceste od LCA až ku koreňu stromu. Táto cesta má dĺžku najviac $\log N$. Vďaka van Emde Boasovmu usporiadaniu vieme takúto cestu prejsť v $O(\log_B N)$ I/O operáciách.

**Zhrnutie.** Prepísanie intervalu dĺžky $k$ spolu s aktualizáciou všetkých dotknutých vrcholov si teda vyžiada

$$O\left(\frac{k}{B} + \log_B N\right) \text{ I/O operácií.}$$

Po dosadení amortizovanej hodnoty $k = O(\log^2 N)$ dostávame výslednú zložitosť cache-oblivious B-stromu.

**Odstránenie členu $O(\log^2 N/B)$.** V predchádzajúcej analýze sa v amortizovanej zložitosti objavili dva členy:

$$O\left(\frac{\log^2 N}{B}\right) \qquad \text{a} \qquad O(\log_B N)$$

Ktorý z nich je väčší?

To, samozrejme, závisí od počtu prvkov $N$ a od veľkosti bloku $B$ (a konštánt skrytých v $O(\cdot)$), avšak $O(\log_B N) = O(\lg N / \log B)$, takže kým v druhom člene delíme iba logaritmom $B$, v prvom člene delíme celým $B$.

Pre lepšiu predstavu si vezmime napríklad $B = 1000$ a $N =$ miliarda ($10^9 \approx 2^{30}$). Potom $\log_B N = 3$, zatiaľčo $\lg^2 N \approx 30 \times 30 = 900$, avšak po delení veľkým $B$ dostávame výsledok menší ako 1.

Vo všeobecnosti je $O(\log_B N)$ dominantný člen pre dostatočne veľké bloky, konkrétne ak $B = \Omega(\log N \log\log N)$.

Prirodzene sa však pýta otázka:

## 24.5 Vieme sa členu $O(\log^2 N/B)$ zbaviť?

Odpoveď je áno, za cenu mierne zložitejšej štruktúry a na úkor horšej zložitosti prechádzania súvislého úseku.

Prvky rozdelíme do $\Theta(N/\log N)$ skupín, pričom každá skupina má veľkosť $\Theta(\log N)$ a tie uložíme do obojsmerného spájaného zoznamu. Každú skupinu budeme reprezentovať ako obyčajné pole, ktoré je zaplnené na 25–100 %. Ľubovoľná operácia v rámci jednej skupiny (vkladanie, vymazávanie, vyhľadávanie) trvá

$$O\left(\frac{\log N}{B}\right) = O(\log_B N) \text{ I/O operácií.}$$

Nad týmito skupinami teraz zostrojíme cache-oblivious B-strom popísaný vyššie, pričom v jeho vnútorných vrcholoch si budeme pamätať maximá jednotlivých skupín. Tento strom slúži iba na navigáciu medzi skupinami.

**Amortizácia.** Kľúčové pozorovanie je, že vkladanie alebo vymazávanie v samotnom B-strome potrebujeme vykonať iba vtedy, keď sa niektorá skupina *preplní* alebo *podtečie*. To sa však stane až po $\Omega(\log N)$ operáciách nad danou skupinou.

Ak si teda každá operácia odloží $O((\log N)/B)$ dolárov, potom po $\Omega(\log N)$ operáciách sa nazbiera presne $O((\log^2 N)/B)$, čo stačí na zaplatenie jednej aktualizácie v B-strome. Týmto spôsobom sa člen $O((\log^2 N)/B)$ rozpustí do amortizácie a výsledná zložitosť každej operácie je len $O(\log_B N)$.

**Nevýhoda.** Táto úprava má však aj svoju daň. Pri obyčajnom usporiadanom poli vieme prejsť interval $k$ prvkov pomocou $O(k/B)$ I/O operácií (po zaplatení $O(\log_B N)$ na nájdenie začiatku a konca).

V poslednej štruktúre musíme pri prechode každou skupinou veľkosti $\Theta(\log N)$ zaplatiť jeden cache miss za skákanie v spájanom zozname. Prechod intervalu $k$ prvkov teda trvá $O(k/B + k/\log N)$, čo je o niečo horšie než v prípade súvislého poľa.

Na druhej strane, pre základné operácie *vyhľadávanie, vkladanie a vymazávanie* dosahujeme cache-oblivious zložitosť $O(\log_B N)$, teda rovnakú ako klasické B-stromy, a to aj bez znalosti $B$ a $M$.

# Kapitola 25

# Haldy

# Kapitola 26

# Štruktúry optimalizované na zápis

Asi chcem spomenut:
LSM tree - basic + BF + leveling/tiering merge
LSM tree + forward + ghost = COLA Bender, Farach-Colton, Fineman, Fogel, Kuszmaul, Nelson 07 Monkey Dostoevsky RocksDB?

# Časť IX

# Appendix

# Kapitola 27

# Ako správne benchmarkovať

V tejto kapitole sa porozprávame o tom, ako správne merať rýchlosť algoritmov a dátových štruktúr. Možno si poviete:

> Na tom predsa nie je nič zložité: spustím stopky, spustím program, keď skončí, stopnem čas – a je to.

Ak máte podobný názor, je to ten najlepší dôvod pokračovať v čítaní, táto kapitola je napísaná presne pre vás Ukážeme si, že realita je omnoho – omnoho zradnejšia: existuje množstvo spôsobov, ako sa pri benchmarkovaní streliť do nohy a prísť k výsledkom, ktoré majú s realitou pramálo spoločné.

## 27.1 Naivný prístup

Skúsme si meranie ukázať na konkrétnom príklade: Koľko trvá vyhľadávanie v heš-tabuľke? Konkrétne vezmime si napríklad heš-tabuľku zo štandardnej knižnice v C++, `std::unordered_map`, a zmerajme, koľko času potrebuje funkcia `find`, aby našla (alebo nenašla) požadovaný prvok.

Ešte predtým, než začneme merať, skúste si tipnúť, aký čas očakávate. Aspoň rádovo: sú to nanosekundy? mikrosekundy? milisekundy? desiatky či stovky?

Tu je náš prvý benchmark:

```
#include <unordered_map>
#include <random>
#include <ctime>
#include <iostream>

int main() {
    std::unordered_map<int, int> map;
    std::mt19937 rng(42);
```

```
    std::uniform_int_distribution<int> dist(0, 1'000'000);

    for (int i = 0; i < 1'000'000; ++i) {
        map[dist(rng)] = i;
    }

    int query = dist(rng); // nahodne cislo

    clock_t start = clock();
    auto value = map.find(query);
    clock_t end = clock();

    double duration = double(end - start) / CLOCKS_PER_SEC;
    std::cout << "Lookup time: " << duration << " seconds\n";
}
```

Skompiloval som (`g++ bench1.cpp -O3`), spustil som – a čo som dostal za výsledok?

Výstup bol zakaždým iný: raz `Lookup time: 2e-06 seconds`, inokedy `5e-06`, potom zase `3e-06`, ... a podobne. Čiže 2–5$\mu$s?

Ovšem, nemôžeme predsa merať len raz a očakávať presnú hodnotu. Zopakujme pokus miliónkrát a spočítajme priemerný čas jedného vyhľadávania.

Nový kód vyzerá takto:

```
    double duration = 0;
    for (int i = 0; i < 1'000'000; ++i) {
        clock_t start = clock();
        auto value = map.find(dist(rng));
        clock_t end = clock();
        duration += double(end - start) / CLOCKS_PER_SEC;
    }
    std::cout << "Lookup time: " << duration / 1'000'000 << " seconds\n";
```

Spustil som trikrát, a dostal som iné hodnoty:

```
Lookup time: 1.27537e-06 seconds
Lookup time: 1.24568e-06 seconds
Lookup time: 1.2248e-06 seconds
```

Takže koľko? 2–5$\mu$ alebo 1.2$\mu$s?

Skúsme spraviť malý experiment: zakomentujme riadok s volaním funkcie `find`. Inými slovami, len „naprázdno“ spustíme a hneď zastavíme stopky, bez toho, aby sa vykonala akákoľvek reálna práca. Výsledok?

```
Lookup time: 1.15427e-06 seconds
Lookup time: 1.25004e-06 seconds
Lookup time: 1.19033e-06 seconds
```

Prosím??

Prezradím, v čom je problém (alebo aspoň jeden z problémov): funkcia `clock()` je úplne nevhodná na mikromerania, pretože nemá dostatočnú presnosť. Je to skoro, ako keby ste chceli merať veľkosť vírusu voľným okom pomocou pravítka – číslo síce dostanete, ale s realitou nebude mať veľa spoločného.

Namiesto toho treba v C++ použiť funkciu `high_resolution_clock` z knižnice `std::chrono`, ktorá je na takéto účely priamo určená:

```
using namespace std::chrono;

    nanoseconds duration{0};
    for (int i = 0; i < 1'000'000; ++i) {
        auto start = high_resolution_clock::now();
        auto value = map.find(dist(rng));
        auto end = high_resolution_clock::now();
        duration += duration_cast<nanoseconds>(end - start);
    }
    std::cout << "Lookup time: " << (duration.count() / 1'000'000) << " nanoseconds\n";
```

A keď už sme pri tom, je trochu hlúpe stále dokola spúšťať a zastavovať stopky a snažiť sa merať takéto krátke časy jednotlivo. Lepšie bude zmerať čas celého cyklu naraz a výsledok vydeliť počtom opakovaní.

```
    auto start = high_resolution_clock::now();
    for (int i = 0; i < 1'000'000; ++i) {
        auto value = map.find(dist(rng));
    }
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
```

Pre istotu spustime obe verzie. Pripravení?

Prvá verzia, ktorá meria každý `find` zvlášť: 23–25 nanosekúnd. Druhá verzia, ktorá meria celý cyklus naraz? 5–6 nanosekúnd.

Dámy a páni, to je viac než štvornásobný rozdiel v meraniach – a v tomto bode to už začína byť trochu na zbláznenie. Ktorému meraniu má človek veriť??

Skúsme ešte zmerať samotné generovanie náhodných čísel. Pôvodne sme totiž chceli merať len čas operácie find. Generovanie náhodného vstupu (volanie `dist(rng)`) je v našom cykle navyše, to by sme mali odpočítať.

Koľko to teda trvá?

Chvíľka napätia – upravím, skompilujem, spustím...

Výsledok: 5–6 nanosekúnd.

Čože??? OK, koniec. Stačilo. Na toto nemám nervy.

## 27.2 Nula celých nula celých nula nula

Asi sa zhodneme, že najlepšie bude použiť presné stopky a merať celkový čas milión opakovaní, nie každú iteráciu jednotlivo. Lenže týmto spôsobom sme namerali, že vyhľadávanie náhodnej hodnoty `value = map.find(dist(rng))` trvá 5–6 nanosekúnd, zatiaľčo iba samotné generovanie náhodného čísla `dist(rng)` trvá 5–6 nanosekúnd. Z toho vyplýva, že samotná funkcia `find` trvá 0 nanosekúnd, čo je – s dovolením – zjavná blbosť. Ako je to možné?

Nastal čas zavŕtať sa trochu hlbšie. Ak chceme pochopiť, čo sa deje, poďme si pozrieť, čo vlastne meriame. Pozrieme sa na vygenerovaný kód v assembleri.

Jedna možnosť je vypýtať si výstup priamo z kompilátora:

```
g++ bench.cpp -O3 -S -o bench.asm
```

alebo, o čosi príjemnejšia možnosť: použiť online nástroj Compiler Explorer[1], kde skopírujeme svoj kód, zvolíme verziu kompilátora a úroveň optimalizácií a hneď vedľa v reálnom čase uvidíme vygenerovaný assembler. Tento nástroj sa navyše snaží farebne vyznačiť časti C++ kódu a zodpovedajúce časti assemblerového výstupu, vďaka čomu sa budeme vedieť lepšie orientovať v kóde.

Netreba rozumieť všetkému, čo sa na úrovni assembleru deje, ale všimnime si aspoň riadky `call`, ktoré reprezentujú volanie nejakej funkcie.

Hmmmm, čo tam nájdeme?

Napríklad funkciu `std::__detail::_Prime_rehash_policy::_M_need_rehash` – tá sa volá, keď sa heš-tabuľka preplní a potrebuje zväčšiť. Ďalej tam vidíme `std::chrono::_V2::system_clock::now` – to sú naše stopky.

Medzi dvoma volaniami `system_clock::now` je dlhočizný názov, ktorý obsahuje `uniform_int_distribution` a `mersenne_twister_engine` – to bude generátor náhodných čísel. Časť kódu, ktorá vykoná naše meranie vyzerá takto:

```
        call    std::chrono::_V2::system_clock::now()
        mov     r12, rax
.L61:
        xor     esi, esi
        mov     edx, 1000000
        mov     rdi, rbp
        call    int std::uniform_int_distribution<int>::...
        sub     ebx, 1
        jne     .L61
        call    std::chrono::_V2::system_clock::now()
```

Riadok `.L64:` definuje návestie – začiatok cyklu. V registri `ebx` je počet opakovaní, ktoré ostávajú. Na konci cyklu `sub ebx, 1` zmenší počítadlo o jedna a `jne .L64` znamená: ak ešte nie sme na nule, skáč späť na začiatok cyklu.

Moment... a kde je funkcia `find`?

[1] dostupný na stránke `https://godbolt.org/`

Nič také tam nie je.

Ale ako je to možné?

Nuž, je to jednoduché, milý Watson, kompilátor toto volanie jednoducho „odoptimalizoval“ – inými slovami vymazal, vyhodil, vyšmaril preč. Kompilátor totiž nevie, že chceme merať čas, ktorý trvá funkcia `find`. Pozrel sa na náš program, všimol si, že hodnotu `value` nikde nepoužívame, a preto usúdil, že je zbytočné ju vôbec počítať.

V tom je kompilátor veľmi „nápomocný“: vytvoril pre nás program, ktorý je ekvivalentný (pretože nič nerobí) – ale nič nerobí oveľa rýchlejšie

Možno ste teraz zmätení a vravíte si:

> No dobre, ale ako to, že kompilátor neodoptimalizoval úplne všetko? Načo potom generuje náhodné čísla? A načo vôbec vytvára tú heš-tabuľku, keď ju – podľa všetkého – vôbec nepoužívame?

Veľmi dobrá otázka. Časť odpovede je, že kompilátor nemôže všetko odoptimalizovať, aj keby chcel, pretože nemože zmeniť chovanie programu. Napríklad vytváranie heš-tabuľky alokuje pamäť a to môže zlyhať (napríklad ak by bola pamäť plná). Takéto zlyhanie je pozorovateľný vedľajší efekt – program, ktorý by alokáciu preskočil, by sa správal inak (nikdy by nezlyhal). A preto optimalizátor nemá právo celú tabuľku zahodiť.

Taktiež vypísanie výsledku na obrazovku je pomerne viditeľný efekt, ktorý nemôže kompilátor odstrániť.

Druhá časť odpovede asi bude, že kompilátor nie je až taký chytrý a najmä, ak nejakú funkciu neinlinuje, nevidí, čo sa deje vo „vnútri“. Funkcia, ktorá generuje náhodné čísla mení vnútorný stav generátora, číta a zapisuje do pamäte. Kompilátor, si ju zrejme netrúfne len tak vyhodiť.

Späť k meraniu času: Ako zabezpečíme, aby nám kompilátor našu funkciu neodoptimalizoval?

Jednoduché riešenie je výsledok funkcie proste použiť. Napríklad v našom prípade môžeme spočítať, koľkokrát sme hľadaný prvok v heš-tabuľke našli a na konci tento počet vypísať na obrazovku. Potom už kompilátor nebude môcť nič vymazať – ak by to urobil, zmenil by správanie programu. Namiesto pôvodného príkazu

```
auto value = map.find(dist(rng));
```

použijeme

```
value += map.find(dist(rng)) != map.end();
```

A voilà, 70–90 nanosekúnd – konečne čas, ktorý sa aspoň trochu podobá realite. Pre istotu si ešte skontrolujeme assembler… Volanie inštrukcie `call find`

tam síce stále nie je, ale to preto, že funkcia bola inlinovaná – kompilátor dosadil jej kód priamo do cyklu.

Mimochodom, toto je tá „správna“ verzia, ktorá stopuje celý cyklus od začiatku do konca. Podľa verzie, ktorá neustále stláča stopky a snaží sa merať každé jedno hľadanie zvlášť, je to 360–480 nanosekúnd. To je ukážkový príklad, ako zlé meranie dokáže výsledok “nafúknuť” niekoľkonásobne, hoci samotný algoritmus ostáva úplne rovnaký.

Druhá možnosť, ako zabrániť optimalizácii je použiť funkciu `DoNotOptimize` z knižnice Google Benchmark. Táto funkcia povie kompilátoru, že výsledok operácie sa môže hocikedy použiť mimo aktuálneho kódu, a preto ho nesmie vyhodiť ani presúvať.

Podobné možnosti existujú aj v iných jazykoch.

Napríklad v Jave (v knižnici Java Microbenchmark Harness) existuje špeciálna trieda `Blackhole`, ktorá poskytuje metódu `blackhole.consume(value)`. Táto metóda hodnotu „skonzumuje“, ale zároveň JIT kompilátoru neprezradí, či je ďalej potrebná alebo nie. Týmto spôsobom JVM zabezpečí, že kód sa neodoptimalizuje preč.

Rust má zase funkciu `black_box`. Táto funkcia sa správa ako identita – jednoducho vráti svoj vstup – ale snaží sa kompilátoru naznačiť, aby bol maximálne pesimistický, pokiaľ ide o jej správanie. V praxi to znamená dve veci: 1. kompilátor by mal predpokladať, že funkcia `black_box` svoj vstup naozaj potrebuje a preto ho musí vyhodnotiť a 2. kompilátor by zároveň nemal predpokladať nič o tom, čo funkcia `black_box` vracia ako výsledok.

Tu je príklad priamo z dokumentácie `black_box`:

```
use std::hint::black_box;

fn contains(haystack: &[&str], needle: &str) -> bool {
    haystack.iter().any(|x| x == &needle)
}

pub fn benchmark() {
    let haystack = vec!["abc", "def", "ghi", "jkl", "mno"];
    let needle = "ghi";
    for _ in 0..1_000_000 {
        black_box(contains(
            black_box(&haystack),
            black_box(needle),
        ));
    }
}
```

Všimnite si, že funkcia `black_box` je tu použitá až trikrát. Ak by sme ju nepoužili vôbec, kompilátor môže spraviť napríklad tieto optimalizácie:

- keďže vstupy funkcie `contains` sa nemenia a je to čistá funkcia bez vedľajších efektov, výsledok stačí vypočítať raz, ešte pred začiatkom cyklu,
- ostane nám prázdny cyklus, ktorý sa môže úplne zmazať,
- vstupy funkcie sú konštantné, kompilátor môže funkciu inlinovať a postupne zjednodušovať, až zistí, že výsledok je vždy `true` – celú funkciu nahradí konštantou
- a keďže výsledok sa vôbec nepoužíva, môže pokojne zmazať aj celú funkciu `benchmark`.

Ak by sme použili `black_box` iba na výsledok, kompilátor stále môže optimalizovať vnútro funkcie a zistí, že `black_box(contains(&haystack, needle))` je to isté ako `black_box(true)`.

Ak by sme pridali `black_box` aj na vstup `needle`, teda

```
black_box(contains(&haystack, black_box(needle)))
```

kompilátor už nedokáže úplne odoptimalizovať celú funkciu.

Stále ju však môže

- inlinovať,
- dosadiť konštanty za `haystack`,
- cyklus, ktorý vo všeobecnosti môže prechádzať ľubovoľne dlhé pole, odrolovať a postupne vyskúšať, či je `needle` jedna z piatich známych hodnôt v `haystack`u,
- aj samotné porovnávanie stringov môže špeciálne zoptimalizovať, pretože neporovnávame ľubovoľne dĺhé reťazce, ale päť 3-znakových literálov.

Dostaneme tak program, ktorý je rýchlejší, pretože vie niečo špeciálne o vstupoch.

Na tomto príklade vidíme ďalšie, ešte zákernejšie spôsoby, ako sa pri meraní môžeme popáliť – ak nameriame, že naša funkcia trvá nulový čas, asi nám dopne, že niečo nie je v poriadku. Ak však kompilátor zoptimalizuje náš benchmark iba čiastočne, pretože predpokladá čosi špeciálne o vstupoch, šanca, že na to prídeme je oveľa menšia.

Túto časť zakončíme ešte jedným príkladom, ktorý sa skutočne stal. Úlohou bolo odmerať zvlášť čas úspešných hľadaní vs. čas neúspešných hľadaní. Tieto časy sa môžu dosť líšiť (samozrejme, aj v závislosti od zaplnenosti tabuľky), keďže pri chýbajúcom prvku musíme preveriť všetky možné miesta, kde by sa mohol nachádzať, zatiaľčo pri úspešnom hľadaní časť môžeme skončiť skôr.

Ako na to?

Jeden nápad je proste generovať náhodné čísla, kým nenájdeme také, ktoré sa v hešmape nenachádza a zmerať len tieto prípady:

```
for (int i = 0; i < N; ++i) {
    do { key = dist(rng); } while (map.find(key) != map.end());
    auto start = high_resolution_clock::now();
    auto value += map.find(key) != map.end();
    auto end = high_resolution_clock::now();
    duration += duration_cast<nanoseconds>(end - start);
}
```

Vidíte tu nejaký problém? (Okrem toho, že zase meriame mrňavé časy zvlášť, čo je zle.)

Nuž, ak kompilátor vie, že `find` je čistá funkcia v zmysle: na rovnakom vstupe dá vždy rovnaký výsledok a nemá žiadne vedľajšie efekty, potom si dokáže odvodiť, že výraz `map.find(key) != map.end()` bude *vždy* nepravdivý (iba vtedy skončí `while`-cyklus vyššie). Inými slovami, výraz, ktorý sa snažíme zmerať, nemusí vôbec znovu počítať, pretože ho raz už vypočítal a výsledok bude rovnaký.

Náš program je ekvivalentný tomuto:

```
for (int i = 0; i < N; ++i) {
    auto result;
    do {
        key = dist(rng);
        result = map.find(key) != map.end();
    } while (result == true);
    auto start = high_resolution_clock::now();
    auto value += result; // vzdy false, teda 0
    auto end = high_resolution_clock::now();
    duration += duration_cast<nanoseconds>(end - start);
}
```

Oveľa lepšie je vygenerovať si všetky vstupy predom, alebo, oveľa jednoduchšie, napríklad vložiť do mapy iba párne čísla a testovať nepárne. Nekonečne inteligentný kompilátor by aj toto dokázal prekuknúť, ale ešte tam nie sme. Ak si chcete byť istý, radšej používajte funkcie ako `DoNotOptimize`, `black_box`, `Blackhole.consume`.

Ako vidíme, niekedy vie byť aj „obyčajné meranie“ rýchlosti pomerne záludná záležitosť a treba si dať pozor na to, čo všetko kompilátor dokáže optimalizovať.

## 27.3 Reprodukovateľnosť výsledkov

Mytkowicz et al. 2009 Berger 2019 Lemire 2019

## Referencie

Berger, Emery (2019). *Performance matters*. Strange Loop Conference, St. Louis, Missouri, USA. URL: `https://www.youtube.com/watch?v=r-TLSBdHe1A`.

Lemire, Daniel (2019). *Benchmarking is hard: processors learn to predict branches*. Daniel Lemire's blog. URL: `https://lemire.me/blog/2019/10/16/benchmarking-is-hard-processors-learn-to-predict-branches/`.

Mytkowicz, Todd et al. (2009). „Producing wrong data without doing anything obviously wrong!“ In: *ACM Sigplan Notices* 44.3, s. 265–276.