Triedenie a vyhľadávanie

Začnime príkladom. Máme súbor dát, ktoré chceme preusporiadať. Pre každý prvok d_i už poznáme jeho cieľovú pozíciu π_i – teda permutáciu π čísel $1, \ldots, N$, ktorá určuje, kam má ktorý prvok patriť. Otázka znie:

> Ako tieto dáta efektívne usporiadať podľa π?

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.

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:

$$\text{I/O zložitosť} \leq \text{časová zložitosť},$$

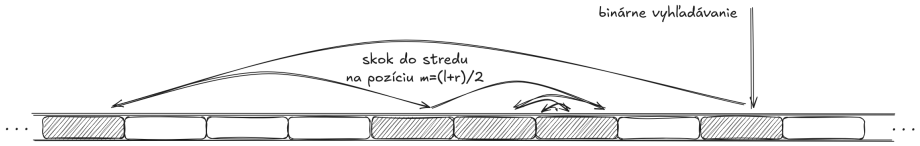
pretože v najhoršom prípade môže každá operácia spôsobiť cache miss a tým jeden prenos. Naopak,

$$\text{I/O zložitosť} \geq \frac{\text{minimálny počet prístupov}}{B},$$

pretože každý prenos načíta aspoň jeden blok veľkosti B.

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.

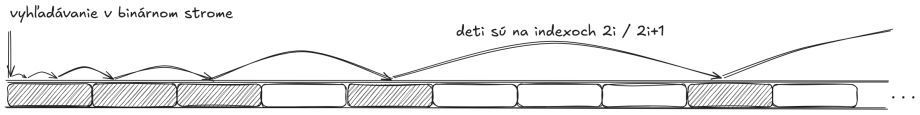
Celkový počet I/O operácií teda vieme odhadnúť ako

$$O(\log(N/B)) = O(\log N - \log B)$$

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.

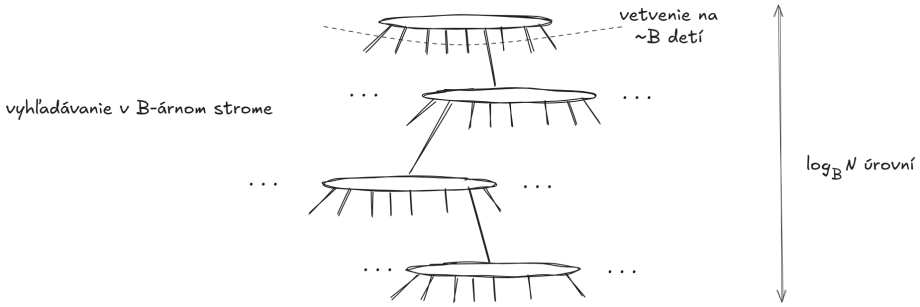
Slabé. To sa naozaj nedá lepšie?

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.



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.

[1] Predpokladajme, že prvky majú konštantnú veľkosť, takže do jedného bloku veľkosti B sa zmestí $\Theta(B)$ prvkov.

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

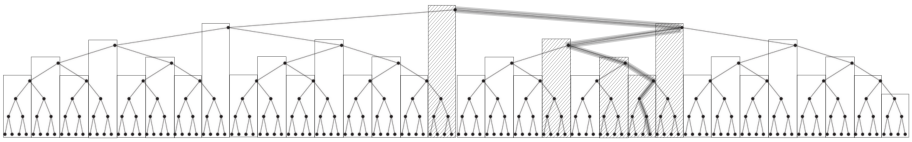
$$k_0 = -\infty < k_1 < k_2 < \cdots < k_B < k_{B+1} = \infty,$$

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.

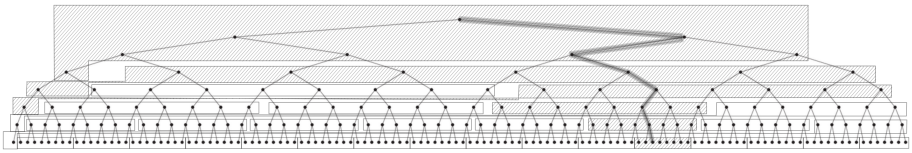
Keďže spolu potrebujeme $\log N$ bitov a z jedného bloku ich získame najviac $\log B$, potrebujeme aspoň

$$\Omega(\log N / \log B)$$

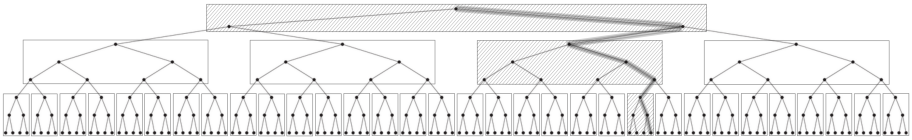
diskových operácií.



(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 ≤ 7 vrcholov. Podstromy majú výšku zhruba $\Theta(\log B)$ a cesta z koreňa do listu prejde $O(\log N / \log B)$ takýchto podstromov.

Obr. 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

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 exter-

nej 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. Najľahšie 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.

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?

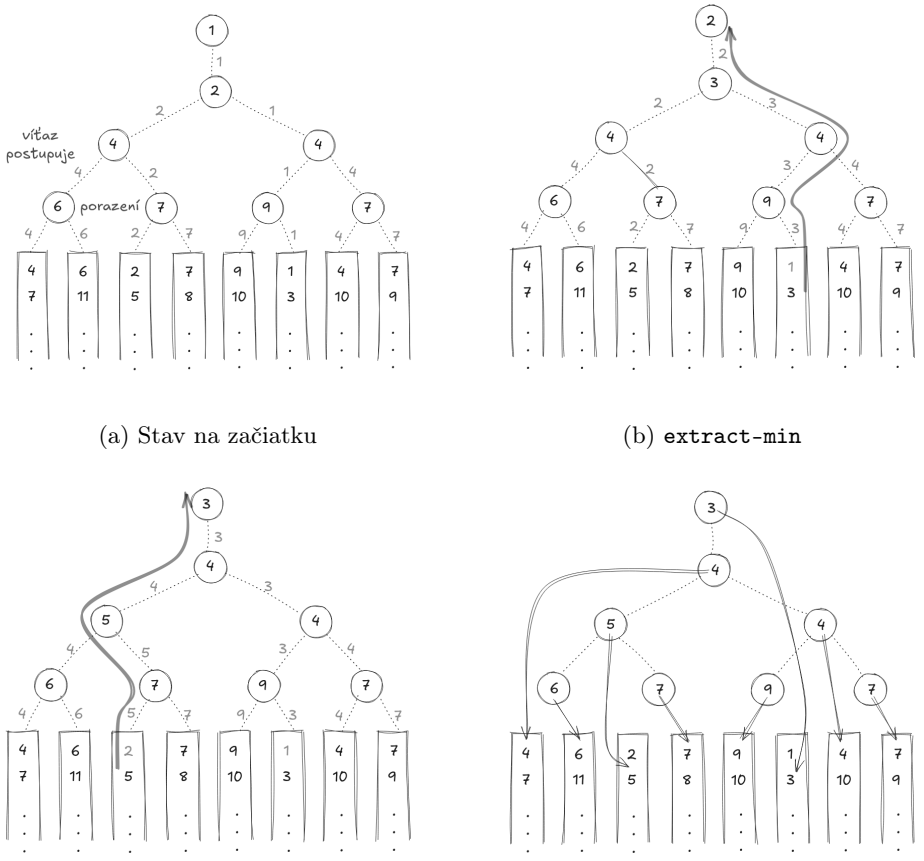
Triviálne riešenie je porovnávať prvky zo všetkých behov priamo, čo by viedlo k zložitosti $O(kN)$ – a to je pre veľké k samozrejme nepoužiteľné. Lepšie je použiť haldu, ktorá nám umožní v každom kroku nájsť najmenší prvok v čase $O(\log k)$. Tým dosiahneme zložitosť $O(N \log k)$,

Ešte o niečo lepšie riešenie a v praxi najpoužívanejšie je použiť tzv. *turnajové stromy* (*loser tree*), pozri obr. 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.



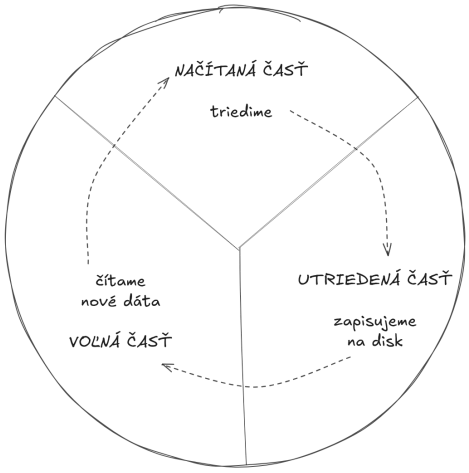
(a) Stav na začiatku

(b) `extract-min`

(c) `extract-min`

(d) Reprezentácia v pamäti.

Obr. 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 prie-

bežne 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.