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 ∞).

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 ∞ pre nespracované vrcholy), malo by nám stačiť $O(C)$ pamäte.

Myšlienka

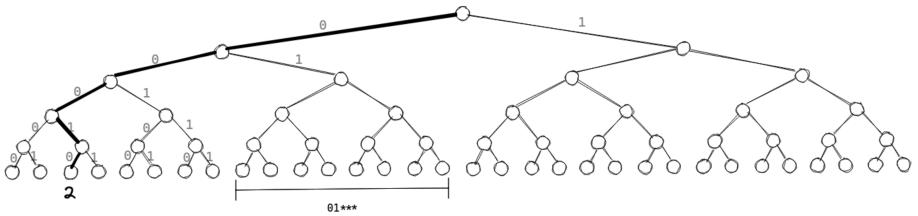
Použijeme *radixovú haldu*: udržiavame niekoľko „chlievikov“ (bucketov) s *exponenciálne* rastúcimi rozsahmi kľúčov

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

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 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.

[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`.

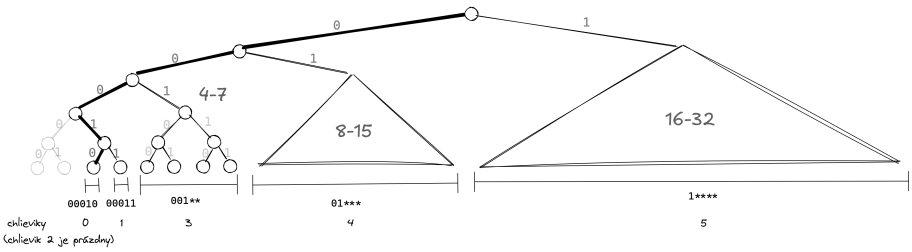
Kratší prefix zodpovedá podstromu – teda rozsahu čísel. Napríklad rozsah `01000` ... `01111` (alebo `01***`, kde `*` je zástupný znak pre `0` alebo `1`) je znázornený na obrázku.

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 ≥ 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.

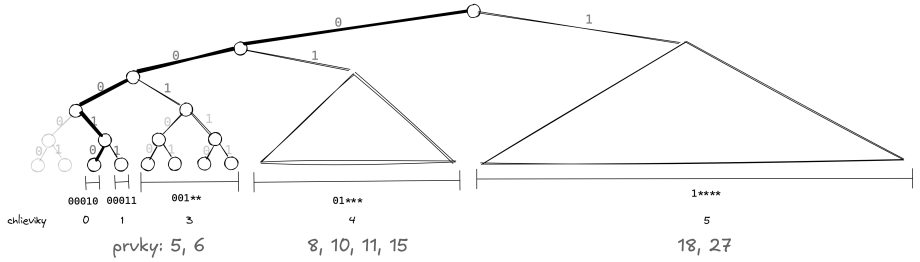
Inými slovami, stačí vypočítať $x \oplus last$ (bitový XOR) a nájsť pozíciu najvyššie nastaveného bitu. (XOR dvoch čísel má jednotky práve na pozíciách, kde sa bity líšia, a nuly tam, kde sú rovnaké.)

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.

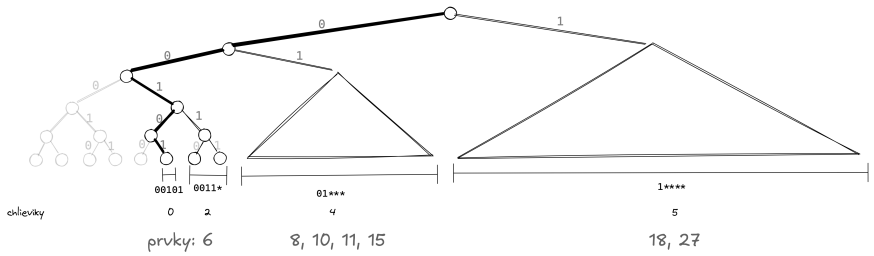
Keď takýto chlievik nájdeme, prejdeme všetky jeho prvky a určíme nové minimum (všimnime si, že čísla v rámci jedného chlievika nemusia byť zoradené!). Toto minimum odstránime a všetky ostatné prvky z daného chlievika presunieme do ich správnych chlievikov podľa novej hodnoty *last*.

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):

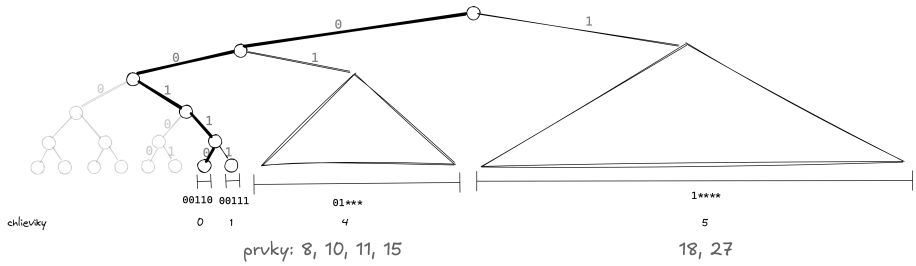


Ak vykonáme operáciu `extract-min`, prejdeme chlieviky 0, 1 a 2, a ako prvý neprázdny nájdeme chlievik č. 3. Z neho vyberieme minimum – hodnotu 5 – a vrátime ju ako výsledok. Zvyšný prvok 6 z tohto chlievika presunieme do nového chlievika na základe aktualizovanej hodnoty $last = 5$.

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



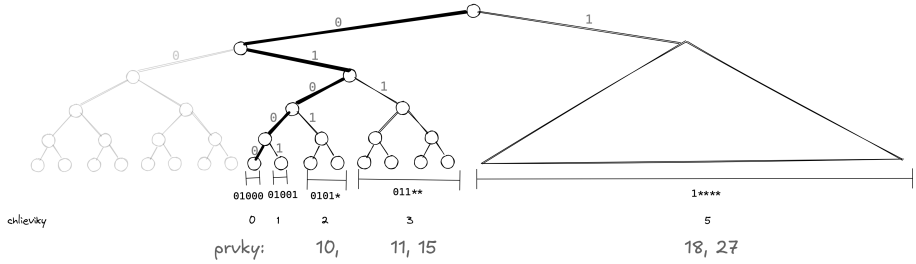
Ak opäť vykonáme `extract-min`, vrátime hodnotu 6 z druhého chlievika a halda bude vyzerať nasledovne:



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 $n \times (O(\log C) + O(\log C)) = O(n \log C)$.

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.

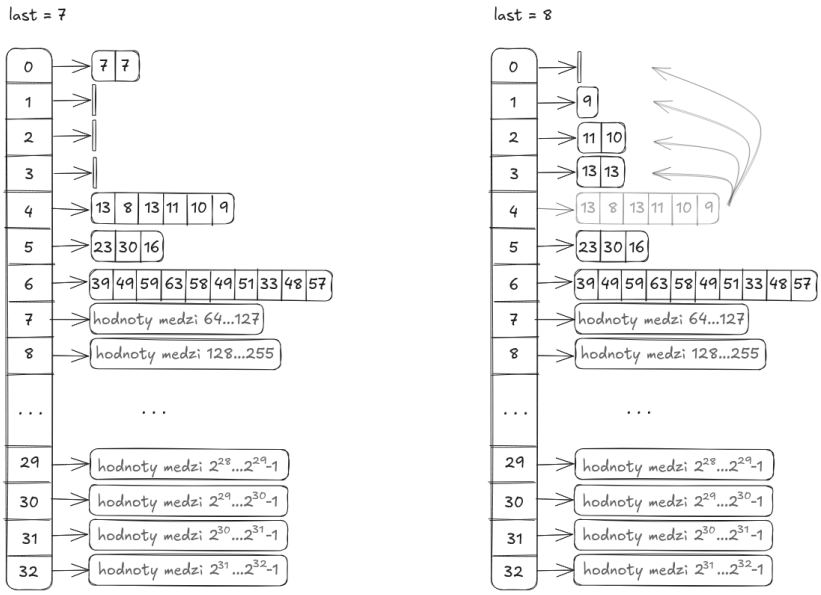
Pri odstraňovaní prvku z vektora si dajte pozor: nechcete používať funkciu, ktorá odstráni prvok a následne presunie všetky prvky za ním – to by malo lineárnu zložitosť! (Ako to spraviť v $O(1)$?)

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 – `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. Na-

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



Obr. 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.

príklad:

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.

[3] `https://en.cppreference.com/w/cpp/numeric/countl_zero.html`
[4] `https://gcc.gnu.org/onlinedocs/gcc/Bit-Operation-Builtins.html`
[5] `https://learn.microsoft.com/en-us/cpp/intrinsics/bitscanreverse-bitscanreverse64?view=msvc-170`
[6] `https://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html#numberOfLeadingZeros-int-`
[7] `https://doc.rust-lang.org/std/primitive.u64.html#method.leading_zeros`