8. Q-Heaps

V minulé kapitole jsme zavedli výpočetní model RAM a nahlédli jsme, že na něm můžeme snadno simulovat vektorový počítač s vektorovými operacemi pracujícími v konstantním čase. Když už máme takový počítač, pojďme si ukázat, jaké datové struktury na něm můžeme vytvářet.

Svým snažením budeme směřovat ke strukturám, které zvládnou operace *Insert* a *Delete* v konstantním čase, přičemž bude omezena buďto velikost čísel nebo maximální velikost struktury nebo obojí. Bez újmy na obecnosti budeme předpokládat, že hodnoty, které do struktur ukládáme, jsou navzájem různé.

Značení: w bude vždy značit šířku slova RAMu a n velikost vstupu algoritmu, v němž datovou strukturu využíváme (speciálně tedy víme, že $w \geq \log n$).

Word-Encoded B-Tree

První strukturou, kterou popíšeme, bude vektorová varianta B-stromu. Nemá ještě tak zajímavé parametry, ale odvozuje se snadno a jsou na ní dobře vidět mnohé myšlenky používané ve strukturách složitějších.

Půjde o obyčejný B-strom s daty v listech, ovšem kódovaný vektorově. Do listů stromu budeme ukládat k-bitové hodnoty, vnitřní vrcholy budou obsahovat pouze pomocné klíče a budou mít nejvýše B synů. Strom bude mít hloubku h. Hodnoty všech klíčů ve vrcholu si budeme ukládat jako vektor, ukazatele na jednotlivé syny jakbysmet.

Se stromem zacházíme jako s klasickým B-stromem, přitom operace s vrcholy provádíme vektorově: vyhledání pozice prvku ve vektoru pomocí operace *Rank*, rozdělení a slučování vrcholů pomocí bitových posuvů a maskování, to vše v čase $\mathcal{O}(1)$. Stromové operace (*Find*, *FindNext*, *Insert*, *Delete*, ...) tedy stihneme v čase $\mathcal{O}(h)$.

Zbývá si rozmyslet, co musí splňovat parametry struktury, aby se všechny vektory vešly do konstantního počtu slov. Kvůli vektorům klíčů musí platit $Bk = \mathcal{O}(w)$. Jelikož strom má až B^h listů a nejvýše tolik vnitřních vrcholů, ukazatele zabírají $\mathcal{O}(h \log B)$ bitů, takže pro vektory ukazatelů potřebujeme, aby bylo $Bh \log B = \mathcal{O}(w)$. Dobrá volba je například $B = k = \sqrt{w}$, $h = \mathcal{O}(1)$, čímž získáme strukturu obsahující $w^{\mathcal{O}(1)}$ prvků o $\sqrt{w}$ bitech, která pracuje v konstantním čase.

Q-Heap

Předchozí struktura má zajímavé vlastnosti, ale často je její použití znemožněno omezením na velikost čísel. Popíšeme tedy o něco složitější konstrukci od Fredmana a Willarda [1], která dokáže totéž, ale s až w-bitovými čísly. Tato struktura má spíše teoretický význam (konstrukce je značně komplikovaná a skryté konstanty nemalé), ale překvapivě mnoho myšlenek je použitelných i prakticky.

Značení:

- $k = \mathcal{O}(w^{1/4})$ – omezení na velikost haldy,
- $r \leq k$ – aktuální počet prvků v haldě,

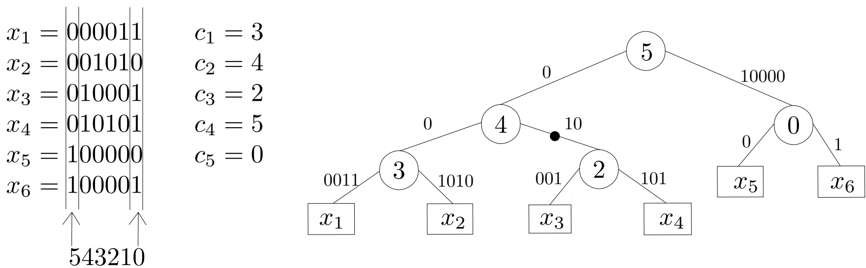
- $X = \{x_1, \ldots, x_r\}$ – uložené w-bitové prvky, očíslujeme si je tak, aby $x_1 < \ldots < x_r$,
- $c_i = \mathrm{MSB}(x_i \oplus x_{i+1})$ – nejvyšší bit, ve kterém se liší x_i a x_{i+1},
- $\mathrm{Rank}_X(x)$ – počet prvků množiny X, které jsou menší než x (přičemž x může ležet i mimo X).

Předvýpočet: Budeme ochotni obětovat čas $\mathcal{O}(2^{k^4})$ na předvýpočet. To může znít hrozivě, ale ve většině aplikací bude $k = \log^{1/4} n$, takže předvýpočet stihneme v čase $\mathcal{O}(n)$. V takovém čase mimo jiné stihneme předpočítat tabulku pro libovolnou funkci, která má vstup dlouhý $\mathcal{O}(k^3)$ bitů a kterou pro každý vstup dovedeme vyhodnotit v polynomiálním čase. Nadále tedy můžeme bezpečně předpokládat, že všechny takové funkce umíme spočítat v konstantním čase.

Iterování: Všimněte si, že jakmile dokážeme sestrojit haldu s k prvky pracující v konstantním čase, můžeme s konstantním zpomalením sestrojit i haldu s $k^{\mathcal{O}(1)}$ prvky. Stačí si hodnoty uložit do listů stromu s větvením k a konstantním počtem hladin a v každém vnitřním vrcholu si pamatovat minimum podstromu a Q-Heap s hodnotami jeho synů. Tak dokážeme každé vložení i odebrání prvku převést na konstantně mnoho operací s Q-Heapy.

Náčrt fungování Q-Heapu: Nad prvky $x_1, \ldots, x_r$ sestrojíme trii T a nevětvící se cesty zkomprimujeme (nahradíme hranami). Listy trie budou jednotlivá x_i, vnitřní vrchol, který leží mezi x_i a x_{i+1}, bude testovat c_i-tý bit čísla. Pokud budeme hledat některé z x_i, tyto vnitřní vrcholy (budeme jim říkat *značky*[1]) nás správně dovedou do příslušného listu. Pokud ale budeme hledat nějaké jiné x, zavedou nás do nějakého na první pohled nesouvisejícího listu a teprve tam zjistíme, že jsme zabloudili. K našemu překvapení však to, kam jsme se dostali, bude stačit ke spočítání ranku prvku a z ranků už odvodíme i ostatní operace.

Příklad: Trie pro zadanou množinu čísel. Ohodnocení hran je pouze pro názornost, není součástí struktury.



Lemma R: $\mathrm{Rank}_X(x)$ je určen jednoznačně kombinací:

(i) tvaru stromu T,

(ii) indexu i listu x_i, do kterého nás zavede hledání hodnoty x ve stromu,

[1] třeba turistické pro orientaci v lese

(iii) vztahu mezi x a x_i ($x < x_i$, $x > x_i$ nebo $x = x_i$) a
(iv) pozice $b = \mathrm{MSB}(x \oplus x_i)$.

Důkaz: Pokud $x = x_i$, je zjevně $\mathrm{Rank}_X(x) = i$. Předpokládejme tedy $x \neq x_i$. Hodnoty značek klesají ve směru od kořene k listům a na cestě od kořene k x_i se všechny bity v x_i na pozicích určených značkami shodují s bity v x. Přitom až do pozice b se shodují i bity značkami netestované. Sledujme tuto cestu od kořene až po b: pokud cesta odbočuje doprava, jsou všechny hodnoty v levém podstromu menší než x, a tedy se do ranku započítají. Pokud odbočuje doleva, jsou hodnoty v pravém podstromu zaručeně větší a nezapočítají se. Pokud nastala neshoda a $x < x_i$ (tedy b-tý bit v x je nula, zatímco v x_i je jedničkový), jsou všechny hodnoty pod touto hranou větší; při opačné nerovnosti jsou menší. ♡

Příklad: Vezměme množinu $X = \{x_1, x_2, \ldots, x_6\}$ z předchozího příkladu a počítejme $\mathrm{Rank}_X(011001)$. Místo první neshody je označeno puntíkem. Platí $x > x_i$, tedy celý podstrom je menší než x, a tak je $\mathrm{Rank}_X(011001) = 4$.

Rádi bychom předchozí lemma využili k sestrojení tabulek, které podle uvedených hodnot vrátí rank prvku x. K tomu potřebujeme především umět indexovat tvarem stromu.

Pozorování: Tvar trie je jednoznačně určen hodnotami $c_1, \ldots, c_{r-1}$ (je to totiž kartézský strom nad těmito hodnotami – blíže viz kapitola o dekompozicích stromů), hodnoty v listech jsou $x_1, \ldots, x_r$ v pořadí zleva doprava.

Kdykoliv chceme indexovat tvarem stromu, můžeme místo toho použít přímo vektor $(c_1, \ldots, c_r - 1)$, který má $k \log w$ bitů. To se sice už vejde do vektoru, ale pro indexování tabulek je to stále příliš (všimněte si, že $\log w$ smí být vůči k libovolně vysoký – pro w známe pouze dolní mez). Proto reprezentaci ještě rozdělíme na dvě části:

- $B := \{c_1, \ldots, c_r\}$ (množina všech pozic bitů, které trie testuje, uložená ve vektoru setříděně),
- $C : \{1, \ldots, r\} \to B$ taková, že $B[C(i)] = c_i$.

Lemma R': $\mathrm{Rank}_X(x)$ lze spočítat v konstantním čase z:

(i') funkce C,
(ii') hodnot $x_1, \ldots, x_r$,
(iii') $x[B]$ – hodnot bitů na „zajímavých" pozicích v čísle x.

Důkaz: Z předchozího lemmatu:

(i) Tvar stromu závisí jen na nerovnostech mezi polohami značek, takže je jednoznačně určený funkcí C.
(ii) Z tvaru stromu a $x[B]$ jednoznačně plyne list x_i a tyto vstupy jsou dostatečně krátké na to, abychom mohli předpočítat tabulku pro průchod stromem.
(iii) Relaci zjistíme prostým porovnáním, jakmile známe x_i.
(iv) MSB umíme na RAMu počítat v konstantním čase.

Mezivýsledky (i)–(iv) jsou opět dost krátké na to, abychom jimi mohli indexovat tabulku. ♡

Počítání ranků je téměř vše, co potřebujeme k implementaci operací *Find*, *Insert* a *Delete*. Jedinou další překážku tvoří zatřiďování do seznamu $x_1, \ldots, x_r$, který je moc velký na to, aby se vešel do $\mathcal{O}(1)$ slov. Proto si budeme pamatovat zvlášť hodnoty v libovolném pořadí a zvlášť permutaci, která je setřídí – ta se již do vektoru vejde. Řekněme tedy pořádně, co vše si bude struktura pamatovat:

Stav struktury:

- k, r – kapacita haldy a aktuální počet prvků (čísla),
- $X = \{x_1, \ldots, x_r\}$ – hodnoty prvků v libovolném pořadí (pole čísel),
- ϱ – permutace na $\{1, \ldots, r\}$ taková, že $x_i = X[\varrho(i)]$ a $x_1 < x_2 < \ldots < x_r$ (vektor o $r \cdot \log k$ bitech),
- B – množina „zajímavých“ bitových pozic (setříděný vektor o $r \cdot \log w$ bitech),
- C – funkce popisující značky: $c_i = B[C(i)]$ (vektor o $r \cdot \log k$ bitech),
- předpočítané tabulky pro různé funkce.

Nyní již ukážeme, jak provádět jednotlivé operace:

$\mathit{Find}(x)$:

1. $i \leftarrow \mathrm{Rank}_X(x)$.
2. Pokud $x_i = x$, odpovíme ANO, jinak NE.

$\mathit{Insert}(x)$:

1. $i \leftarrow \mathrm{Rank}_X(x)$.
2. Pokud $x = x_i$, hodnota už je přítomna.
3. Uložíme x do $X[{+}{+}r]$ a vložíme r na i-té místo v permutaci ϱ.
4. Přepočítáme c_{i-1} a c_i. Pro každou změnu c_j:
5. Pokud ještě nová hodnota není v B, přidáme ji tam.
6. Upravíme $C(j)$, aby ukazovalo na tuto hodnotu.
7. Upravíme ostatní prvky C, ukazující na hodnoty v B, které se vložením posunuly.
8. Pokud se na starou hodnotu neodkazuje žádné jiné $C(\cdot)$, smažeme ji z B.

$\mathit{Delete}(x)$:

1. $i \leftarrow \mathrm{Rank}_X(x)$ (víme, že $x_i = x$).
2. Smažeme x_i z pole X (například prohozením s posledním prvkem) a příslušně upravíme ϱ.
3. Přepočítáme c_{i-1} a c_i a upravíme B a C jako při Insertu.

Časová složitost: Všechny kroky operací po výpočtu ranku trvají konstantní čas, rank samotný zvládneme spočítat v $\mathcal{O}(1)$ pomocí tabulek, pokud známe $x[B]$. Zde

je ovšem nalíčen háček – tuto operaci nelze na Word-RAMu konstantním počtem instrukcí spočítat. Pomoci si můžeme dvěma způsoby:

a) Využijeme toho, že operace $x[B]$ je v AC^0, a vystačíme si se strukturou pro AC^0-RAM. Zde dokonce můžeme vytvářet haldy velikosti až $w \log w$. Také při praktické implementaci můžeme využít toho, že současné procesory mají instrukce na spoustu zajímavých AC^0-operací, viz např. pěkný rozbor v [2].

b) Jelikož B se při jedné Q-Heapové operaci mění pouze o konstantní počet prvků, můžeme si udržovat pomocné struktury, které budeme umět při lokální změně B v lineárním čase přepočítat a pak pomocí nich indexovat. To pomocí Word-RAMu lze zařídit, ale je to technicky dosti náročné, takže čtenáře zvědavého na detaily odkazujeme na článek [1].

Aplikace Q-Heapů

Jedním velice pěkným důsledkem existence Q-Heapů je lineární algoritmus na nalezení minimální kostry grafu ohodnoceného celými čísly. Získáme ho z Fredmanovy a Tarjanovy varianty Jarníkova algoritmu (viz kapitoly o kostrách) tak, že v první iteraci použijeme jako haldu Q-Heap velikosti $\log^{1/4} n$ a pak budeme pokračovat s původní Fibonacciho haldou. Tak provedeme tolik průchodů, kolikrát je potřeba zlogaritmovat n, aby výsledek klesl pod $\log^{1/4} n$, a to je konstanta. Všimněte si, že by nám dokonce stačila halda velikosti $\Omega(\log^{(k)} n)$ s operacemi v konstantním čase pro nějaké libovolné k.

Literatura

[1] M. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proceedings of FOCS'90*, pages 719–725, 1990.

[2] M. Thorup. On AC^0 Implementations of Fusion Trees and Atomic Heaps. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 699–707, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.