Datové struktury I

1 Hashování

Základ – slovníkový problém, dáno univerzum U, úkolem reprezentovat množinu $S \subseteq U$ a na ní operace MEMBER, INSERT, DELETE. V případě, že $|S| << |U|$, nechci mít celé pole velikosti U v paměti – mám funkci $h : U \to \{0, \ldots, m-1\}$ a mn. S reprezentuji polem s m políčky tak, že $x \in S$ je uložen na indexu $h(x)$. Problém – kolize: $x \neq y$, $h(x) = h(y)$. Ozn. $|S| = n$, $|U| = N$. Load factor – $\alpha = \frac{n}{m}$.

1.1 Separované řetězce

Neuspořádané

Řešení kolizí – řetězení ve spojácích: pro každé políčko zvlášť spoják všech prvků, co mají takový hash, algoritmy je procházejí. Řetězce jsou prosté – nic v nich není 2x. Nejhorší případ – jeden seznam; paměť. náročnost – $\forall$ seznam $O(1 + l(i))$, kde $l(i)$ je jeho délka.

Pro složitost algoritmů – **očekávaná délka řetězců**: za předpokladu rychlého spočítání h, rovnoměrného rozdělení h, rovnoměrného náhodného výběru z S. Délka i-tého řetězce budiž $\mathbf{l}(i)$. Potom

$$P(\mathbf{l}(i) = l) = p_{n,l} = \binom{n}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-l}$$

což je ale jen aproximace, pro případ, že $N >> n^2 m$ ale lze použít. Ve skutečnosti $\forall i \; \exists \frac{N}{m}$ prvků x, že $h(x) = i$ a přesný výpočet je

$$p_{n,l} = \frac{\binom{\frac{N}{m}}{l}\binom{N-\frac{N}{m}}{n-l}}{\binom{N}{n}}$$

Potom očekávaná délka řetězce je (rozepíšu faktoriál a vytknu $\frac{n}{m}$, pak změním rozsah sumace $1 \ldots n$, pak $0 \ldots n-1$):

$$E(l) = \sum_{l=0}^{n} l p_{n,l} = \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-1-l} = \frac{n}{m}(\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m} = \alpha$$

Pro rozptyl potřebuju $E(l^2) = E(l(l-1)) + E(l)$, získám podobně (2× vytknu):

$$E(l) = \sum_{l=0}^{n} l(l-1)\binom{n}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-l} = \frac{n(n-1)}{m^2} \sum_{l=0}^{n-2} \binom{n-2}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-2-l} = \frac{n(n-1)}{m^2}$$

z toho rozptyl

$$\mathbf{var}(l) = E(l^2) - (E(l))^2 = \frac{n}{m}(1 - \frac{1}{m})$$

a očekávaný nejhorší případ (**délka nejdelšího řetězce**)

$$ENP = \sum_j j P(\max_j \mathbf{l}(i) = j) = \sum_j P(\max_i \mathbf{l}(i) \geq j)$$

kde

$$P(\max_i \mathbf{l}(j) \geq j) \leq \sum_i P(\mathbf{l}(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}$$

Najdeme mezní hodnotu j_0, pro které $n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1$ (z odhadu) – pak z $\frac{n}{m} \leq 1$ a $(\frac{j}{2})^{\frac{j}{2}} \leq j!$ plyne $j_0 \leq \min\{j | n \leq (\frac{j}{2})^{\frac{j}{2}}\} =: (k+1)$. Pak $(\frac{k}{2})^{\frac{k}{2}} < n \leq (\frac{k+1}{2})^{\frac{k+1}{2}}$, z prvního zlogaritmování vezmu levou a z druhého pravou stranu a podělením dostanu

$$\frac{k \log(\frac{k}{2})}{4 \log(\frac{k+1}{2})} < \frac{\log n}{\log \log n}$$

a protože pro $k \geq 3$ platí $\frac{1}{2} < \frac{\log \frac{k}{2}}{\log \frac{k+1}{2}}$, mám $k < 8\frac{\log n}{\log \log n}$, tj. $j_0 = O(\frac{\log n}{\log \log n})$.

Celkem tedy pro $\alpha \leq 1$ je $ENP = O(j_0) = O(\frac{\log n}{\log\log n})$ (pro $0.5 \geq \alpha \leq 1$ je $\Theta(\frac{\log n}{\log\log n})$):

$$ENP = \sum_j P(\max_i \mathbf{l}(i) \geq j) \leq \sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty}(n(\frac{n}{m})^{j-1}\frac{1}{j!}) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = \cdots \leq j_0 + \frac{1}{j_0}$$

Testy jsou porovnání argumentu hledání s prvkem, nebo zjištění, že řetězec je prázdný – jejich počet je další odhad efektivity. Rozlišujeme úspěšné a neúspěšné hledání (a z neúspěšného se dá spočíst úspěšné).

Neúspěšné hledání ($p_{n,0}$ - jeden test stejně provedu; do řetězce nepatří žádný z n prvků):

$$E(T) = p_{n,0} + \sum_l p_{n,l} = (1 - \frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$$

Úspěšné hledání – pro všechny prvky v 1 řetězci dohromady $1 + 2 \cdots + \mathbf{l}(i)$ testů, z toho pro všechny úplně $m\sum_l \frac{l(l+1)}{2}p_{n,l}$ a z toho pro 1 prvek $\frac{m}{2n}(\sum_{l=1} nl(l-1)p_{n,l} + \sum_{l=1}^{n} lp_{n,l}) = \frac{m}{2n}\left(\frac{n(n-1)}{m^2} + \frac{2m}{n}\right) \approx 1 + \frac{\alpha}{2}$. Jiná úvaha – obecně se tak dá provádět – počet testů pro úspěšné vyhledávání je roven počtu testů provedených při vložení prvku: $\frac{1}{n}\sum_{i=0} n - 1\left(1 + \frac{i}{m}\right)$ a vyjde to samé.

Uspořádané

To samé, jen prvky jsou v řetězcích uspořádané podle klíče. Počet testů pro úspěšné hledání je úplně stejný, pro neúspěšné končím dřív ($e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$).

1.2 Hashování s přemísťováním

Nevýhoda sep. řetězců – nutnost alokovat další paměť, to je neefektivní. Proto zavedu do tabulky další pomocné položky a řetězce nacpu přímo do ní. V případě přemísťování mám v tabulce navíc jednoduše odkaz na předch. a násl. prvky v řetězci. Pokud vkládám a už tam je něco z jiného řetězce, tak to něco přehodím jinam. Algoritmy jsou jednoduché, jen při DELETE prvního prvku řetězce je nutné dát druhý (je-li) na jeho místo.

Očekáváný počet testů je stejný jako pro hashování se separovanými řetězci. Nevýhoda: přemísťování v tabulce je náročnější než 1 test – INSERT a DELETE jsou pomalejší.

1.3 Hashování se 2 ukazateli

Místo ukazatele na předch. prvek používá odkaz na začátek řetězce „begin“ – řetězec už nemusí začínat na indexu svého hashe. Algoritmy místo přesouvání mění „begin“ (ten je na j-tém políčku vyplněn, právě když existuje řetězec prvků s hashem j; INSERT všechno cpe na konec řetězce, zakládá-li nový, do „begin“ píše, kde; DELETE jen upravuje odkazy na násl., nebo „begin“).

Počet testů – o něco větší, kvůli začátku řetězce jinde; úspěšné: $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$, neúspěšné přibližně $1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2 + \alpha) - 2$.

1.4 Srůstající (coalesced) hashování

Má jen 1 položku v tabulce navíc (ukazatel na další prvek „next“), řetězce obsahují hodnoty s různými hashi. Prvek s vkládáme vždy do řetězce, obsahujícího $h(s)$-té políčko v tabulce. Různé metody: standardní (bez pomocné paměti) LISCH, EISCH a bezpřívlastkové (s pomocnou pamětí) LICH, VICH, EICH.

Bez pomocné paměti – LISCH a EISCH

LISCH je „late insertion“, tedy přidává se za poslední prvek řetězce. EISCH („early“) přidává za první prvek. Alg. MEMBER stejný pro oba (jen projití řetězce po „next“). Alg. INSERT: u LISCH projití celého řetězce (v případě že není prázdný, jinak jednoduše vložím na správné políčko) s testy na přítomnost prvku, potom vložení na lib. volné místo v tabulce & připojení na konec řetězce. Pro EISCH – přepojení ukazatelů „next“ a vložení do řetězce za 1. prvek (pokud řetězec je neprázdný).

Algoritmy DELETE nejsou známy, kromě primitivních. Problém – zachování náhodného usp. prvků v řetězcích, pokud nedodržím, zhorší to očekávané doby. Možné také prvky jen označit jako odstraněné a jejich místa použít při

vkládání dalších (ale zpomaluje hledání). EISCH je o něco rychlejší na úspěšné vyhledání (větší pravděpodobnost práce s novým prvkem), očekávaný počet testů stejný.

Počet testů v neúspěšném případě – mějme prvky $s_1, \ldots, s_n$ uložené (stejně pravděpodobně) na (libovolných) adresách $a_1, \ldots, a_n$ a hledáme s_{n+1}. Počet testů při takovém hledání budiž $C(a_1, a_2, \ldots, a_n; a_{n+1})$, potom očekávaný počet testů je (pro $c_{n,l}$ počet řetězců délky l, které přispívají celkem $1 + 2 + \cdots + l = l + \binom{l}{2}$ porovnáními):

$$\frac{\sum_{\text{všechny posl. } a_i} C(a_1, \ldots, a_n; a_{n+1})}{m^{n+1}} = \frac{c_{n,0} + \sum_{l=1}^{n} l c_{n,l} + \sum_{l=1}^{n} \binom{l}{2} c_{n,l}}{m^{n+1}}$$

a $c_{n,0}$ je počet prázdných řádků, tedy $c_{n,0} = (m-n)m^n$, $lc_{n,l}$ je součet délek všech řetězců v reprezentacích n-prvk. množin, tedy $\sum_{l=1}^{n} lc_{n,l} = nm^n$. Poslední člen := S_n, 2 možnosti vzniku řetězce délky l přidáním dalšího prvku do $n-1$-prvkové množiny – buď přidávám do řetězce (délky $l-1$), nebo řetězec (pův. délky l) nezměním:

$$S_n = \sum_l \binom{l}{2}(m-l)c_{n-1,l} + \sum_l \binom{l}{2}(l-1)c_{n-1,l-1} = mS_{n-1} - \sum_l l^2 c_{n-1,l} = (m+2)S_{n-1} + (n-1)m^{n-1}$$

(úpravy: rozepsání rozdílů, vykrácení, rozpis l^2 jako $l^2 - l + l = \binom{l}{2} + l$), pak získám nerekurentní vztah:

$$S_n = (m+2)^{n-1}S_0 + \sum_{i=0}^{n-1}(m+2)^i(n-1-i)m^{n-1-i} = (m+2)^{n-1}\sum_{i=1}^{n-1} i\left(\frac{m}{m+2}\right)^i = \frac{1}{4}(m(m+2)^n - m^{n+1} - 2nm^n$$

(úpravy: $S_0 = 0$, obrácení sumace a vytknutí $m+2^{n-1}$, použití vztahu $T_n^c = \sum_{i=1}^{n} ic^i = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}$ spočítaného z $(c-1)T_n^c = nc^{n+1} + (\sum_{i=2}^{n} -c^i) - c)$. Tedy očekávaný počet testů je:

$$1 + \frac{1}{4}\left(\left(1 + \frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$$

Úspěšný případ – pro LISCH podobně jako u separovaných řetězců, tj. počet testů při vkládání prvku. Metoda EISCH nesplňuje předpoklady. Porovnání klíčů při neúspěšném vyhledávání je (bez hrabání na prázdná políčka) $\frac{n}{m} + \frac{1}{4}\left(\left(1 + \frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) = \frac{1}{4}\left(\left(1 + \frac{2}{m}\right)^n - 1 + \frac{2n}{m}\right)$. Očekávaný počet testů při úspěšném je $1 + \sum_{i=0}^{n-1}(\textit{por. při neúsp.})$ a to je

$$1 + \sum_{i=0}^{n-1}\frac{1}{4}\left(\left(1 + \frac{2}{m}\right)^i - 1 + \frac{2i}{m}\right) = 1 + \frac{m}{8n}\left(\left(1 + \frac{2}{m}\right)^n - 1 - \frac{2n}{m}\right) + \frac{n-1}{4m} \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

Pro metodu EISCH potom vychází (bez důkazu)

$$\frac{m}{n}\left(\left(1 + \frac{1}{m}\right)n - 1\right) \approx \frac{1}{\alpha}(e^{\alpha} - 1)$$

Všechny odhady mají odchylku $O(\frac{1}{m})$.

S pomocnou pamětí – LICH, VICH, EICH

Paměť je dělená na (hash-funkcí) přímo adresovatelnou a pomocnou část (bez přístupu hash-funkcí). Při kolizích nejdříve bereme řádky z pomocné části, pak teprve z přímo adresovatelné, tj. oddalujeme srůstání řetězců – chování se podobá separovaným do určitého okamžiku. Varianty – „early“, „late“, „varied“. Také nemá přirozené efektivní DELETE.

LICH vždy přidává na konec řetězce, EICH v případě neprázdného řetězce vždy za 1. prvek a VICH vždy za poslední prvek v pomocné paměti nebo (pokud žádné v pomocné paměti nejsou) za 1. prvek řetězce (tj. chová se na pomocné paměti jako LICH a v přímo adresovatelné části jako EICH). Algoritmy až na VICH jsou stejné jako pro standardní, rozhodující je výběr volného řádku pro vložení: např. „z nejvyšší adresy“ může zaručit používání pomocné paměti.

Složitost – použití n – počtu uložených prvků, m – velikosti přímé paměti, m' – celk. velikosti paměti a $\alpha = \frac{n}{m'}$ – faktor zaplnění, $\beta = \frac{m}{m'}$ – adresovací faktor. λ budiž jediné nezáp. řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$. Pokud je $\alpha \leq \lambda\beta$, pak pro všechny verze vychází očekávaný počet testů $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$ v neúspěšném případě a $1 + \frac{\alpha}{2\beta}$ v úspěšném (chyba: $O(\log \frac{m'}{\sqrt{m'}})$. V případě, že $\alpha \geq \lambda\beta$ (začínají srůstat řetězce), se metody liší a vychází divnosti. V neúspěšném případě je VICH a LICH lepší než EICH, v úspěšném vede VICH před EICH a LICH (vždy o jednotky procent). Doporučená $\beta = 0.86$. Pro hledání volného řádku se hodí např. spojový seznam volných řádků.

1.5 Lineární přidávání

Nemá žádné položky pro práci s tabulkou, nalezení dalšího prvku je přímo v algoritmu. Nejjednodušší – lineární přidávání: v případě kolize naleznu nejbližší vyšší volné políčko a vložím tam. Předpokládám „cyklickou“ paměť. Problém: shluky – při velkém zaplnění se dost zpomaluje, nepodporuje efektivní DELETE (a primitivní způsoby taky rychlé nejsou). Pro test přeplnění – dobré uchovávát počet uložených prvků nebo mít zarážku (nikdy neobsazované pole).

Počet testů – $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right)$ v neúspěšném a $\frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)\right)$ v úspěšném případě (bez důkazu).

1.6 Dvojité hashování

Vylepšení předchozího, aby nevznikaly shluky tak rychle: výběr následujícího řádku bude závislý na předchozím, rovnoměrně rozmístěný. Použiji 2. hash-fci h_2, vždy hledám takové i od 0, že $(h_1(x)+i\cdot h_2(x)) \mod m$ je volné políčko, tj. při operaci INSERT i hledání postupně přičítám $h_2(x)$ a modulím. Nutné je, aby $h_2(x) \not| m$, abych měl prosté posloupnosti. Idea, že jde pro každé x o náh. permutaci, není úplně přesná, ale v praxi stačí, aby z $h_1(x)=h_1(y)$ plynulo, že $h_2(x)$ a $h_2(y)$ budou odlišné. Nevýhoda – zas nemá efektivní DELETE. Přeplnění se řeší pořád stejně.

Nutné funkce volit správně (i lineární přidávání je spec. příp. dvojitého hashování, kdy $h_2\equiv 1$), pak vychází znatelně rychlejší než lin. přidávání; sice nelze splnit předpoklad náhodnosti, použitý v teoretické analýze, ale přiblížit se mu ano.

Počet testů – (teoreticky, na zákl. předpokladu náhodnosti) **v neúspěšném případě**: označme $q_i(n,m)$ pravděpodobnost, že při zaplnění $\frac{n}{m}$ je pro nějaké x prvních $i-1$ políček, kam bych ho mohl vložit, plných. Potom $q_i(n,m)=\frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}$ a tedy $q_i(n,m)=\frac{n}{m}q_{i-1}(n-1,m-1)$. Potom očekávaný počet testů (v urč. případě prvních $j-1$ obsazených, j volné) je

$$C(n,m)=\sum_{j=0}^{n}(j+1)(q_j(n,m)-q_{j+1}(n,m))=\sum_{j=0}^{n}(q_j(n,m))=1+\frac{n}{m}C(n-1,m-1)=\frac{m+1}{m-n+1}$$

(předposlední z rekurentního vztahu pro q_j, poslední krok dokázat indukcí).

Testy v úspěšném případě – stejná metoda jako u dřívějších analýz, takže vychází

$$\frac{1}{n}\sum_{i=0}^{n-1}C(i,m)=\frac{1}{n}\sum_{i=0}n-1\frac{m+1}{m-i+1}\approx\frac{1}{\alpha}\ln\left(\frac{m+1}{m-n+1}\right)\approx\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

1.7 Srovnání a dodatky

Podle počtu testů:

	neúspěšné	úspěšné
1.	separované uspořádané řetězce	separované (usp. i neusp.) řetězce, přemísťování
2.	separované řetězce, přemísťování	dva ukazatele
3.	dva ukazatele	VICH
4.	VICH, LICH	LICH
5.	EICH	EICH
6.	LISCH, EISCH	EISCH
7.	dvojité hashování	LISCH
8.	lineární přidávání	dvojité hashování
9.		lineární přidávání

VICH je při vhodném α lepší než dva ukazatele. Lineární přidávání nepoužívat pro $\alpha>0.7$, dvojité hashování pro $\alpha>0.9$. Separované řetězce a obecné srůstající hashování používají víc paměti, přemísťování a dvojité hashování zas víc času, tj. nelze říct, které je jednoznačně lepší.

Pro hledání volných řádků se většinou používá seznam (zásobník). Přeplnění se většinou řeší držením α v rozumném intervalu ($\langle 1/4, 1\rangle$) a přehashováním do jinak velké tabulky ($2^i \cdot m$) při pře- nebo podtečení; v praxi se doporučuje přehashování odkládat (např. pomocnými tabulkami) a provádět při nečinnosti systému.

DELETE ve strukturách co to nepodporují se řeší označením políčka jako smazaného s možností využití při vkládání, v případě 1/2 polí blokované tímto způsobem přehashováním. Pro srůstající hashování se toto používat nemusí, máme metody na zachování náhodnosti rozdělení dat.

V praxi je výhodné, známe-li něco o rozdělení vstupních dat, aby ho hashovací funkce kopírovala (většinou to ale nejde), jinak musíme předpokládat rovnoměrnost, což zaručeno zdaleka není. Nutnost rovnoměrného rozdělení vstupních dat lze obejít, viz níže.

1.8 Universální hashování

Idea: mít místo pevné hash-funkce nějakou množinu H, z níž funkci náhodně rovnoměrně vyberu. Pak se analýza dělá přes všechny $h \in H$ a platí pro všechny $S \subset U$ (S je daná pevně a h se k ní volí; $|U| = N$). Pro spočítatelnost a formalizaci nutné analytické zadání funkcí h a známá velikost množiny, což obejdu očíslováním funkcí $H = \{h_i; i \in I\}$ a počítáním s indexy (očekávaná hodnota je průměr přes I). Při použití skutečné velikosti H v odhadech bychom dostali horší výsledky.

Definice: Systém funkcí $H = \{h_i; i \in I\} : U \to \{0, \ldots, m-1\}$ je c-universální, pokud $\forall x, y \in U, x \neq y : |\{i \in I; h_i(x) = h_i(y)\}| \leq \frac{c|I|}{m}$.

Existence c-universálních systémů

Předpokládejme, že universum má tvar $U = \{0, 1, \ldots, N-1\}$ kde N je nějaké prvočíslo a vezmeme funkce typu

$$h_{a,b}(x) = ((ax + b) \mod N) \mod m$$

Jsou dobře použitelné, protože se dají počítat rychle. Tato množina funkcí je universální – vyjdeme z Frobeniovy věty o jednoznačnosti řešení soustav lin. rovnic (protože N je prvočíslo, můžeme pracovat v $\mathbb{Z}_N$, coz je těleso). Rovnice

$$h_{a,b}(x) = h_{a,b}(y)$$

je ekvivalentní s

$$\exists i \in \{0, \ldots, m-1\} \ \wedge \ \exists r, s \in \{0, \ldots, \lceil \frac{N}{m} \rceil - 1\} : (ax + b \equiv i + rm) \mod N$$
$$(ay + b \equiv i + sm) \mod N$$

Počet řešení soustavy je omezený číslem $m \cdot \lceil \frac{N}{m} \rceil^2$ (i – m hodnot, r, s – $\lceil \frac{N}{m} \rceil$ hodnot pro daná x, y). Pak je systém c-univerzální pro $c = \left(\lceil \frac{N}{m} \rceil\right)^2 / \left(\frac{N}{m}\right)^2$ (rozšířím vzorec pro počet řešení $\frac{N^2}{m^2}$). Jeho velikost opdovídá N^2.

Vlastnosti

Vyrobíme si pomocnou funkci

$$\delta_i(x, y) = 1 \text{ pro } h_i(x) = h_i(y), x \neq y \text{ a } = 0 \text{ jinak}$$

Chceme potom spočítat součet $\delta_i(x, S) = \sum_{y \in S} \delta_i(x, y)$. Z výsledku vidíme očekávanou délku řetězce pro libovolnou (jednu) množinu dat. Tohle pak sečtu přes všechny mé hash-funkce a z c-univerzality dostanu

$$\sum_{i \in I} \delta_i(x, S) = \sum_{y \in S} \sum_{i \in I} \delta_i(x, y) \leq \sum_{y \in S, x \neq y} c\frac{|I|}{m} = \begin{cases} (|S| - 1)c\frac{|I|}{m} & \text{pro } x \in S \\ |S|c\frac{|I|}{m} & \text{jinak} \end{cases}$$

Z toho dopočítám (podělením $|I|$) horní odhad očekávaného $\delta_i(x, S)$, výsledek: očekávaný čas operací MEMBER, INSERT a DELETE v c-univ. hashování je $O(1 + c\alpha)$ (kde faktor naplnění $\alpha = \frac{|S|}{m}$). Čas n po sobě jdoucích operací na původně prázdné tabulce je $O(n(1 + \frac{c}{2}\alpha))$. To není lepší hodnota než mají separované řetězce ($O(1 + \alpha)$), ale tady nechci rovnoměrné rozdělení dat (rovnoměrný výběr i můžu ovlivnit).

Mějme nějakou očekávanou hodnotu délky řetězce, chci vědět kolik hash-funkcí jí porušuje kolikrát. Z Markovovy nerovnosti se tohle dá spočítat. Máme dánu očekávanou hodnotu $\delta_i(x,S) := \mu$ a $t \geq 1$. Označme $I' = \{i \in I | \delta_i(x,S) \geq t\mu\}$. Potom z $\mu > \frac{\sum_{i\in I'} \delta_i(x,S)}{|I|} \geq \frac{|I'|}{I} t\mu$ dostaneme $\frac{|I'|}{|I|} < \frac{1}{t}$. Pravděpodobnost vybrání takovéto „špatné“ funkce je tedy menší než $\frac{1}{t}$.

Výběr vhodného i není úplně jednoduchý, protože funkcí může být např až N^2, tj. nelze provést jednoduchým zavoláním náh. generátoru, nýbrž např. náhodným vybráním každého bitu čísla. Chceme kvůli tomu najít co nejmenší c-univerzální systémy.

Dolní odhady velikosti univ. systémů

Očíslujme hash-funkce z I a induktivně definujme množiny U_i jako největší podmnožiny U_{i-1} takové, že $h_{i-1}(U_i)$ je jednoprvková. Platí $|U_i| \geq \lceil \frac{U_{i-1}}{m} \rceil$, tedy $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ – velikost těchto množin klesá s logaritmem a $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$. Takže velikost univ. systému roste alespoň úměrně logaritmu velikosti univerza.

Dokážeme **existenci 5-univerzálního systému**: Zvolme $t \in \mathbb{N}$ a k němu vezměme t-té prvočíslo p_t tak, že $t \ln p_t \geq m \ln N$. Definujme systém funkcí $H = \{g_{c,d,l}(x) | t < l \leq 2t, c, d \in \{0, 1, \ldots, p_{2t} - 1\}\}$ kde $((c(x \mod p_l) + d) \mod p_{2t}) \mod m$. Zřejmě $|H| = tp_{2t}^2$. Z teorie čísel plyne, že $p_t = O(t \log t)$, tedy $\log |H| = \log t + 2 \log p_{2t} = O(\log t + 2 \log 2t + 2 \log \log 2t) = O(\log t) = O(\log m + \log \log N)$, tj. splňuje odhad.

Spočteme odhad $|G = \{(c,d,l); h_{c,d,l}(x) = h_{c,d,l}(y)\}|$, když si množinu rozdělíme na $G_1 = \{(c,d,l) \in G; x \mod p_l \neq y \mod p_l\}$ a $G_2 = \{(c,d,l) \in G; x \mod p_l = y \mod p_l\}$. Pro $(c,d,l) \in G_1$ existují $i \in \{0, \ldots, m-1\}$ a $r, s \in \{0, \ldots, \lceil \frac{p_{2t}}{m} \rceil - 1\}$, že $(c(x \mod p_l) + d \equiv i + rm) \mod p_{2t}$ a $(c(y \mod p_l) + d \equiv i + sm) \mod p_{2t}$. Z toho dostávám regulární (z vl. G_1) matici nad $\mathbb{Z}_{p_{2t}}$, podle Frobeniovy věty dostávám jedno (c,d) pro každé (l,i,r,s) – celk. počet prvků je

$$|G_1| \leq tm(\lceil \frac{p_{2t}}{m} \rceil)^2 \leq \frac{tp_{2t}^2}{m} \left(1 + \frac{m}{p_{2t}}\right)^2 = \frac{|H|}{m} \left(1 + \frac{m}{p_{2t}}\right)^2 \leq 4\frac{|H|}{m}$$

Pro G_2 si označím $L = \{l; t < l \leq 2t, x \mod p_l = y \mod p_l\}$. Potom $(c,d,l) \in G_2$ jen když $l \in L$, takže

$$|G_2| \leq |L| p_{2t}^2 \leq \frac{tp_{2t}^2}{m} = \frac{|H|}{m}$$

za podmínky $|L| < t/m$, která je splněna z $\prod_{l \in L} p_l \leq |x - y| \leq N$ (P dělí $|x-y|$) a $P > p_t \not| L|$ ($l > t$), což dává $|L| \leq \ln N / \ln p_t$ a dále z definice. Celkem tedy $|G| = |G_1| + |G_2| \leq 5\frac{|H|}{m}$. Za dalších podmínek se dá dokázat i 3.25-univerzálnost takového systému.

Dolní odhad c: Platí: $c > 1 - \frac{m}{N}$. Spočítáme $\sum_{h \in H} \sum_{x,y \in U} \delta_h(x,y)$: pro pevnou h vezmu $\sum_{x,y \in U} \delta_h(x,y) = \sum_{i=0}^{m-1} u_{i,h}(u_{i,h} - 1) \geq \frac{(\sum_{i=0}^{m-1} u_{i,h})^2}{m} - N = \frac{N^2}{m} - N$ (z Cauchy-Schwarze, $u_{i,h} = |\{x \in U, h(x) = i\}|$). Tedy $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \geq \frac{|H|N(N-m)}{m}$. Zároveň platí $\sum_{h \in H} \sum_{x,y \in U} \delta(x,y) \leq \sum_{x,y \in U} c\frac{|H|}{m} = N^2 c \frac{|H|}{m}$, z toho výsledek.

1.9 Perfektní hashování

Idea – nalézt hash-funkci, která nedělá kolize pro danou množinu. Nevýhoda: nelze přirozeným způsobem realizovat INSERT, tj. v praxi se nesmí moc často vyskytovat. Tabulka by neměla být o mnoho větší než množina a funkce h rychle spočitatelná a její realizace nezabírat moc paměti (žádná zadávání tabulkou). Soubor funkcí $H : U \to \{0, \ldots, m-1\}$ je (N,m,n)-perfektní, pokud $\forall S \subseteq U$ takové, že $|S| = n$ existuje $h \in H$ perfektní pro S.

Odhady velikosti: každá funkce h je perfektní pro $\sum\{\prod_{j=0}^{n-1} |h^{-1}(i_j); 0 \leq i_0 < \cdots < i_{n-1} < m\}$, z Cauchy-Schwarze je max. pro $\forall i : |h^{-1}(i)| = \frac{N}{m}$, takže je perfektní pro max. $\binom{m}{n}(\frac{N}{m})^n$ množin. Z toho

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}\left(\frac{N}{m}\right)^n}$$

Druhý odhad, přes množiny jako u c-univ. systémů při $|H| = \{h_1, \ldots, h_t\}$, dostávám pak $|U_t| \leq 1$ a $N/m^t \leq 1$, tedy $t \geq \frac{logN}{logm}$.

Existence: přes matice: $M(H)$ typu $N \times t$. $M(H)_{x,i} = h_i(x)$. Pak žádná funkce z H není perfektní, když $M(H)$ nemá prostý sloupec. Takových matic je max. $\left(m^n - \prod_{i=0}^{n-1}(m-i)\right)^t \cdot m^{(N-n)t}$ (počet všech funkcí minus počet prostých (repr. podmaticemi s n řádky), to celé krát libovolné doplnění na N řádek). Podmnožin U velikosti n je

pak $\binom{N}{n}$, čímž vynásobeno mám počet matic, odp. (N,m,n)-perfektnímu systému. Všech matic je m^{Nt} – dám do nerovnosti, vydělím m^{Nt}, pak zlogaritmuju a zintegruju :) a dostávám podmínku existence $t \geq n(\ln N)e^{\frac{n^2}{m}}$.

Konstrukce funkce: chceme splnit rychlou spočitatelnost a paměťovou nenáročnost. Předpokládám univerzum prvočíselné velikosti a funkce typu

$$h_k(x) = (kx \mod N) \mod m$$

Ozn. $b_i^k = |\{x \in S; (kx \mod N) \mod m = i\}|$. Potom pokud $h_k(x)$ není perfektní, pak nějaké $b_i^k = 2$ a mám $\sum_{i=0}^{m-1}(b_i^k)^2 \geq n+2$.

Odhadnu výraz $\sum_{k=1}^{N-1}((\sum_{i=0}^{m-1}(b_i^k)^2) - n) = \sum_{x\neq y\in S}|\{k; 1 \leq k < N, h_k(x) = h_k(y)\}$ – tj. existují i,r,s taková, že $kx \equiv i + rm \mod N$ a $ky \equiv i + sm \mod N$, z toho $k(y-x) \equiv (s-r)m \mod N$ a protože je $\mathbb{Z}_N$ těleso, $\forall t \in \{-\lfloor \frac{N}{m} \rfloor, \ldots, \lfloor \frac{N}{m} \rfloor\}$ $\exists! k$ takové, že $k(y-x) \equiv tm \mod N$ a takových t je tedy nejvýše $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$. Dostávám tedy odhad $2(N-1)\frac{n(n-1)}{m}$ a z něj (při zachování podmínky perfektnosti $h_k(x)$) $\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{m} + n$.

Dá se dokázat trochu slabší předpoklad, že $P(k; \sum_{i=0}^{m-1}(b_i^k)^2 < \frac{3n(n-1)}{m} + n) \geq 1/4$, který je základem pravděpodobnostního algoritmu. Pak mám deterministický algoritmus, který pro $m = n(n-1)+1$ nalezne perfektní h_k v čase $O(nN)$ $(\sum(b_i^k)^2 < 3n)$ a pravděpodobnostní, který pro $m = 2n(n-1)$ najde perfektní h_k v čase $O(n)$ $(\sum(b_i^k)^2 < 4n)$.

Mám tedy konstrukci perfektní hash-funkce, ta ale nesplňuje požadavek na malou tabulku ($m = \Theta(n^2)$). To se dá vyřešit tak, že po nalezení této funkce najdeme všechny neprázdné množiny $S_i = \{s \in S; h_k(s) = i\}$ a pro jim odpovídající $c_i = |S_i|(|S_i|-1)+1$ (pro dvojnásobek v pravděp. metodě) najdeme k_i takové, že h_{k_i} je perfektní funkce pro S_i do tabulky velikosti c_i. Definujme $d_i = \sum_{j=0}^{i-1} c_j$, potom pokud $h_k(x) = l$, pak $g(x) = d_l + h_{kl}(x)$ je perfektní, její hodnota spočitatelná v čase $O(1)$ a hashuje do tabulky velikosti $O(3n)$ ($O(6n)$ s pravděp. případě), je naleznutelná v čase $O(nN)$ ($O(n)$) a pro její uložení do paměti jsou potřeba hodnoty k a k_i, vyžadující $O(n \log N)$ paměti. Pro výpočet $g(x)$ potřebuji 2 násobení, 2 modulo a 1 sčítání (pro d_i v paměti), tabulka má velikost $\sum c_i \leq \sum(b_i^k)^2 < 3n$.

„Malá“ funkce: Toto stále nesplňuje požadavek na málo paměti pro uložení. Víme, že pro $m \in \mathbb{N}$ je počet prvočísel, která ho dělí $O(\frac{\log m}{\log\log m})$, z toho úvahou o dělitelích $D = \prod_{1\leq i<j\leq n}(s_j - s_i) \leq N^{n^2}$ na n-prvkové S a vzorce $p_t \leq 2t\ln t$ dostanu, že existuje p o velikosti $O(\ln D) = O(n^2 \ln N)$ takové, že $\phi_p(x) = x \mod p$ je perfektní pro S. Deterministické nalezení trvá $O(n^3 \log n \log N)$ (test perfektnosti každého $O(n\log n)$), proto použijeme pravděpodobnostní alg (mezi $4cn^2 \ln N$ přir. čísly je aspoň 1/2 prvočísel, která vyhovují) – první najde prvočíslo a pak testuje perfektnost, očekávaný počet testů je $O(\ln(4cn^2 \ln N))$, celkem složitost algoritmu je pak $O(n\log n(\log n + \log\log N))$. Najde zhruba až 2× větší prvočíslo než deterministický. Tuhle funkci použijeme ke konstrukci hash-funkcí – nalezneme prvočíslo q_0 aby $\phi_{q_0}(x) = x \mod q_0$ byla perfektní pro S, položíme $S_1 = \{\phi_{q_0}(s) | s \in S\}$, pak najdeme prvočíslo $q_1 \in \langle n(n-1)+1, 2n(n-1)+2\rangle$ a k němu existuje $l \in \langle 1, q_0 - 1\rangle$ takové, že $h_l(x) = ((lx \mod q_0) \mod q_1)$ je perfektní pro S_1, položím $S_2 = \{h_l(s) | s \in S_1\}$ a najdu perfektní g pro S_2 do tabulky s méně než $3n$ řádky. Pak $f(x) = g(h_l(\phi_{q_0}(x)))$ je perfektní. Funkce q_0 je určena 1 číslem o vel. $O(n^2 \log N)$, h_l 2 čísly o velikosti $O(n^2)$ a $O(q_0)$, g je určená $n+1$ čísly o $O(q_1)$, tj. celkem zadání vyžaduje $O(n\log n + \log n + \log\log N)$.

1.10 Dynamické perfektní hashování

Perfektní hashování nepodporuje efektivní INSERT a DELETE, to se snaží tyto metody řešit (dynamizace pravděpodobnostních algoritmů). Máme množiny funkcí $H_t : U \to \{0, \ldots, t-1\}$ pro $t \in \{1, \ldots, N-1\}$ tvaru $h_k(x) = (kx \mod N) \mod t$ pro $k = 1, \ldots, N-1$. Volíme-li náhodně k, pak $h_k \in H_s$ s pravd. $\geq 1/2$ splňuje

$$\sum_{i=0}^{s-1}(b_i^k)^2 < \frac{8n^2}{s} + 2n$$

kde $s = \frac{4}{3}\sqrt{6}(1+c)|S|$ (pro fixované $c > 1$). Označme $S_i = \{t \in S; h_k(t) = i\}$ a pak zvolíme-li $j(i)$ náhodně, je $h_{j(i)} \in H_{2(b_i^k)^2}$ perfektní s pravd. 1/2 – předp. že takové $j(i)$ máme $\forall i \in \{1, \ldots, N-1\}$. Reprezentujme S_i funkcemi $h_{j(i)}$, pro každou zvláštní tabulkou T_i (pro $i \in \{0, \ldots, s-1\}$).

Algoritmus MEMBER jen počítá hash-fce, INSERT zjistí, zda $x \in S$, není-li, najde příslušnou T_i (pomocí $i = (kx \mod N) \mod s$) a vloží. Pokud je funkce stále perfektní, je vše OK, jinak alg. spočítá novou perfektní hash-funkci pro novou S_i a vytvoří T_i. Když k nesplňuje pravděpodobnostní rovnici, přehashuje se vše – spočte se nové s a nalezne správné k, rozdělí se do množin S_i a pro každé i se najde $j(i)$ a vytvoří tabulka T_i. DELETE zjišťuje po odstranění, zda k splňuje pravděp. rovnici, pokud ne, provede přehashování.

Složitost – lineárně paměti (bez uložení funkcí), MEMBER $O(1)$ v nejhorším případě a INSERT, DELETE $O(1)$ amortizovaně (bez důkazu).

Druhá možnost: hypergrafy

Hashuje do tabulky velikosti n. $r-$hypergraf – hrany jsou $r-$prvkové podmnožiny mn. vrcholů. Cyklus – hypergraf, kde každý vrchol leží alespoň ve 2 různých hranách. Mějme tedy *acyklický* r-hypergraf s n hranami. Nalezeneme funkci $g : V \to \{0, \dots, m-1\}$ takovou, že $h : E \to \{0, \dots, m-1\} : h(e) = \sum_{i=1}^{r} g(v_i) \mod m$ je prostá na $\{0, \dots, m-1\}$. To lze provést tak, že zvolím bijekci $h : E \to \{0, \dots, m-1\}$ a pak je-li $g(v_i)$ definováno pro $i = 2, \dots, r$, definuji $g(v_1) = h(e) - \sum_{i=2}^{r} g(v_i) \mod m$. Protože v každém acyklickém $r-$hypergrafu existuje vrchol ležící jen v 1 hraně, lze toto použít ke kontrukci g indukcí. Pak naleznu funkce $f_1, \dots, f_r : U \to V$ takové, že $E = \{(f_1(x), \dots, f_r(x)); x \in S\}$ je acyklický hypergraf. Celková hashovací funkce je definovaná

$$f(x) = \sum_{i=1}^{r} g f_i(x)$$

Nejvýhodnější je, když f_i jsou náhodná zobrazení náhodně zvolená, nebo náhodná z nějakého c-univerzálního systému. Vyžaduje $O(rn + |V|)$ času a $O(n \log n + r \log |V|)$ paměti (bez důkazu).

1.11 Externí hashování

Jiný cíl – minimalizace počtu operací s externí pamětí, ve které jsou uložená všechna data. Vezmeme prostou funkci $U \to \{0,1\}^k$. Pro mn. S. Slovu $\alpha \in \{0,1\}^*$ kratšímu než k přiřadíme $h_S^{-1}(\alpha) = \{s \in S; \alpha \text{ je prefix } h(S)\}$. Potom α je *kritické slovo*, když $h_S^{-1}(\alpha)$ se vejde do 1 stránky ext. paměti, ale všechny prefixy α toto neumí. Pro každé $s \in S$ existuje právě 1 krit. slovo; def. $d(S) = \max\{\text{délka}(\alpha); \alpha \text{ krit. slovo}\}$. Pak na stránce příslušející α uložíme $h_S^{-1}(\alpha)$. Přiřazení stránky kritickému slovu α řeší funkce – adresář. Adresář může být lexikograficky podle prefixů uspořádaná tabulka adres stránek. Ukládáme ho pak taky na jedné stránce ext. paměti. Potom MEMBER jsou max. 3 operace a INSERT/DELETE max. 6 operací s ext. pamětí. Očekávaný počet použitých stránek je $\frac{n}{b \ln 2}$ a velikost adresáře $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$ – to je víc než lineární růst, tj. nelze používat donekonečna (bez důkazu).

2 Stromy

Hashování nezachovává uspořádání a může často zklamat – motivace k používán stromů. Kromě operací MEMBER, INSERT a DELETE mám ještě MIN, MAX, ORD (k-tý největší) a SPLIT (podle x, které vyhodí, je-li ve stromě) a JOIN (dvě verze, s prvkem navíc, nebo bez něho).

2.1 (a, b) stromy

Pro $a \leq b$ přirozená je (a, b) strom, když

1. každý vnitřní vrchol v stromu T různý od kořene t má alespoň a a nejvíc b synů
2. všechny cesty od kořene k listům mají stejnou délku

Tato definice je ale pro praktické účely příliš obecná – budeme chtít $a \geq 2$ a $b \geq 2a - 1$ a podmínku navíc:

3. kořen je buď list nebo má alespoň 2 a nejvíc b synů

Takový (a, b) strom existuje pro každý přirozený počet listů, jeho výška je mezi $\log_b n$ a $1 + \log_a(\frac{n}{2})$, tedy $O(\log n)$. Indukcí: strom o výšce h má $2a^{h-1} \leq n \leq b^h$ listů (přidáním h-té hladiny do stromu s k listy dostaneme strom s $ka \leq n \leq kb$ listy. Strom pak *reprezentuje* nějakou množinu S, když mám bijekci mezi uspořádáním S a lexikografickým uspořádáním listů. Každý vnitřní vrchol v obsahuje informaci o počtu synů $\rho(v)$, pole ukazatelů na syny S_v a pole H_v prvků z U takových, že i-tý je největší v S reprezentovaný v podstromě i-tého syna. Listy mají jen svůj prvek. Pro každý prvek kromě největšího existuje vnitřní vrchol, který ho obsahuje – to se používá při vynechávání listů (což ale není moc přehledné). Můžu mít i odkaz na otce, nebo si otce můžu pamatovat při průchodech dolů (na vrcholech, ke kterým jsem nedošel od kořene, otce nepotřebuju).

Pro algoritmy máme pomocný VYHLEDEJ, který do hloubky projde stromem a vrátí nejbližší větší k nějakému prvku (nebo prvek sám, je-li ve stromě). INSERT: vyhledám místo kam, pokud tam prvek není, vytvořím nový list, připojím na správné místo do S_v a postupně nahoru štěpím, je-li potřeba, extrémně rozštěpím kořen. DELETE: najdu prvek, najdu, kam je pověšený a to jedno políčko v S_v a H_v zruším, opravím ρ, pokud dostanu méně než a synů v uzlu, spojím s bezprostř. bratrem (má-li ten právě a synů), nebo přesunu nějaký z bratra do mého uzlu. Všechny algoritmy pracují v $O(\log |S|)$ v nejhorším případě.

JOIN (bez prvku navíc): Přepokládá $\max S_1 < \min S_2$. Je-li $h(T_1) \geq h(T_2)$, najde v T_1 hladinu o 1 nad připojení a v ní největší prvek / vytvoří nadkořen T_1 v případě rovnosti, slije spojí do něj prvky obou kořenů a případně provede štěpení. Jinak hledá a připojuje v T_2. Potřebuje čas $O(|h(T_1) - h(T_2)|)$.

SPLIT: Prochází postupně dolů, rozděluje uzly (podstromy s prvky $< x$, resp. $\geq x$) a hází výsledky do 2 zásobníků. Pokud oddělí více než 1 krajní prvek, hodí na zásobník strom, jehož kořen je právě oddělená část uzlu, jinak na zásobník dává podstrom onoho krajního prvku. Tak pokračuje až k listům, pokud tam najde přímo x, tak ho vyhodí. Stromy ze zásobníků spojí postupným voláním JOIN – v 1 zásobníku jsou max. 2 stromy stejné výšky, je jich celkem $k \leq 2 \log_a |S|$ a jejich vyprázdnění trvá $O(\sum_{i=1}^{k-1}(h(T_i) - h(T_{i+1}) + 1)) = O(h(T_1) + k)$.

ORD: S takovouto reprezentací efektivně nejde, musíme navíc $\forall$ uzel udržovat pole P_v s počty vrcholů v jeho jednotl. podstromech a to při vkládání a odebírání průběžně aktualizovat. Pak v ORD procházím do hloubky a postupně přičítám velikosti přeskočených podstromů (pokud bych se přičtením dalšího dostal k 0, jdu na nižší hladinu).

Pro vnitřní paměť se doporučuje $a = 2 - 3$ a $b = 2a$, pro vnější $a \approx 100$ a $b = 2a$. Při přístupu více uživatelů ke struktuře je problém s aktualizačními operacemi – zamykání celého stromu není efektivní: použije se *vyvažování shora dolů*: algoritmus INSERT zamkne uzel, jeho otce a syny. Pak pokud je počet synů $= b$, rozštěpí ho (předem), nebude se pak už štěpit zpátky. Aby tohle fungovalo (abych měl $\geq a$ synů všude), je nutné $b \geq 2a$. Podobně funguje DELETE – najde-li uzel s a syny, provede „preventivně“ slití nebo přesun. Toto provádí víc slití a štěpení než původní algoritmy, ale asymptoticky je to furt stejné. Pro takovéto struktury na ext. paměti se doporučuje $a \approx 100$ a $b = 2a + 2$.

A-sort

Aplikace (a, b) stromů v třídících algoritmech – vhodné pro částečně předtříděné posloupnosti, jinak oproti klasickým algoritmům nemá žádné výhody. Pro algoritmus je nutné znát list s nejmenším prvekm FIRST, cestu k němu od kořene a pro každý list ukazatele na následující v uspořádání NEXT.

Postup: Odzadu (od „předtříděně největšího“) vkládat prvky do stromu modifikovaným A-INSERTem a pak přečíst posloupnost listů (jít po NEXT). A-INSERT pracuje tak, že místo pro vložení prvku hledá od FIRST (jde postupně nahoru po otcích a hledá, kde nejdřív může slézt zas k listům).

Složitost: Pomalejší než běžné na libovolná data (asymptoticky stejné), ale rychlejší na částečně předtříděná. Vezmeme F – počet inverzí v posloupnosti. Celk. potřebuju $O(n)$ pro načtení prvků, $O(n)$ pro všechna štěpení dohromady ze všech běhů A-INSERTu a na každé vložení $O(h)$ pro nalezení místa, kde h je výška, kam se z FIRST dostanu, přeskočím tak $f_i \geq a^{h-2}$ vrcholů (menších než vkládaný) a $h \in O(\log f_i)$. Součet f_i za $\forall i$ dává $n \log(\prod_{i=1}^{n} f_i)^{\frac{1}{n}} \leq n \log \frac{\sum_{i=1}^{n} f_i}{n} = n \log \frac{F}{n}$.

Protože se nepoužívá DELETE, hodí se na toto $(2, 3)$ stromy. Pro míru $F \leq n \log n$ má složitost $O(n \log \log n)$, v urč. případech i rychlejší než Quicksort.

Hladinově propojené (a, b)-stromy s prstem

Navíc oproti běžným mají uzly položky OTEC, LEVY, PRAVY (na odpovídající uzly na stejné hladině – sousedy v uspořádání) a PRST (na nějaký list). Používá zobecněné vyhledávání z A-sortu – začíná od PRSTu p, hledám-li x a $p > x$, jdu do otce t a zkouším jeho podstromy, případně se přesunu do LEVY(p), pokud ani tam x být nemůže, jdu o hladinu výš atd.; na druhou stranu zrcadlově. Najdu-li vrchol, v jehož podstromu by měl x ležet, použiji normální operaci VYHLEDEJ.

Implementace operací zahrnují ještě algoritmus SET-PRST(x), nastavující PRST na největší list menší nebo rovný x. Jsou výhodné pro řady operací, které se motají v blízkém okolí 1 prvku.

Amortizované odhady

Nejpomalejší jsou štěpení, slití a přesuny, obecně může INSERT volat $\log(|S|)$-krát štěpení a DELETE $\log(|S|)$-krát slití a 1 přesun. Začínáme-li s prázdným stromem a měříme na nějaké posloupnosti operací, zjistíme, že jde o $O(1)$. Ozn. $c = \min\{\min\{2a-1, \lceil\frac{b+1}{2}\rceil\} - a, b - \max\{2a-1, \lceil\frac{b+1}{2}\}\}$. Pak pro $b \geq 2a$ a $a \geq 2$ a posloupnost $\mathcal{P}$ z n operací INSERT a DELETE označme St_h, Sp_h a P_h počet štěpení, slití a přesunů ve výšce h. Potom platí:

1. $P \leq n$ a $(2c-1)St + cSp \leq n + c + \frac{c(n-2)}{a+c-1}$
2. $St_h + Sp_h + P_h \leq \frac{2(c+2)^n}{(c+1)^h}$

Z 1. dostanu $St + Sp \leq \frac{3n}{2} + 1$ a z toho

$$\lim_{n\to\infty} \frac{P + St + Sp}{n} \leq \frac{5}{2}$$

Doporučené hodnoty – pro $b = 2a-1$ lze nalézt jednoduše posloupnosti operací, kde počet slití a štěpení je $O(n \log n)$, to samé pro paralelní operace a $b = 2a+1$, takže proto se doporučuje $2a$, resp. $2a+2$.

V hladinově propojeném stromě platí, že posloupnost n operací MEMBER, INSERT, DELETE a PRST vyžaduje $O(\log n +$ čas na vyhledání prvků).

2.2 Vyhledávání v uspořádaném poli

Množina S reprezentována v setříděném poli, vyhledání – zjištění, zda je x v rozsahu S, pak je buď na kraji nebo existují d, h, $d < h$, že $A[d] < x < A[h]$, zmenšuji interval (d, h) a testuji, zda jsem x nenašel. Končím, když ho najdu, nebo když $d+1 \geq h$. Používá pomocnou funkci NEXT, která vrací hodnotu $d < n < h$ pro zmenšení intervalu. Zprac. dotazu je $O(1)$ a pokud čas výpočtu NEXT je $O(1)$, celkem mám $O($počet volání NEXT$)$. Nevýhoda: INSERT a DELETE nejdou efektivně, snaha najít je vede na binární stromy.

Unární hledání – NEXT je $d+1$. Max počet dotazů je $|S|$, očekávaný $\frac{|S|}{2}$, tedy $O(|S|)$.

Binární hledání – NEXT je $\lceil\frac{d+h}{2}\rceil$. Každý dotaz zmenší rozsah zhruba na 1/2, tedy pro k potřebných dotazů je $|S| \leq 2^{k-3}$, tedy složitost $O(\log|S|)$ a očekávaný čas pro rovnoměrné rozdělení stejný.

Interpolační hledání – NEXT je $d + \lceil\frac{x-A[d]}{A[h]-A[d]}(h-d)\rceil$. V nejhorším případě až $O(\frac{|S|}{2})$, ale pro rovnoměrné rozdělení dat je očekávaný čas $O(\log\log|S|)$ – založeno na faktu, že NEXT závisí i na x. Pokud rozložení prvků není rovnoměrné, ale známe ho, můžeme toto upravit, aby zas vyšla stejná očekávaná složitost.

Zobecněné kvadratické hledání – NEXT složitější: hledání děleno na bloky (testů), v rámci nichž jsou testy na sobě závislé. 1. dotaz v bloku – interpolační, tím se zjistí o kolik se seknul a stanoví se velikost kroku c pro další dotazy. Ostatní dotazy v bloku je střídání unárních a binárních. Blok končí, pokud $h - d < c$. Velikost kroku klesá zhruba jako odmocňování. Používá pomocné proměnné BLOK (udává přítomnost v rámci bloku), TYP (příští dotaz unární/binární), SMER (krok jde vlevo/vpravo), KROK (velikost kroku pro unární dotazy, na zač. bloku se stanoví na $\lfloor\sqrt{h-d}\rfloor$), jejich hodnoty se předávají napříč voláními.

Přes svou komplikovanost je spočítání NEXT $O(1)$. Velikost kroku klesá zhruba jako odmocňování, tj. počet bloků je $\leq$ nejmenší i takové, že $\sqrt[2^i]{n} \geq 1$, z toho $i \leq \lceil\log\log n\rceil$. Pro počet operací v 1 bloku uvažuju náh. veličinu X, která určuje počet i z intervalu $\langle d, h\rangle$ takových, že $A[i] \leq x$. Díky rovnoměrnému rozdělení x je X binomicky rozdělená s $p = \frac{x-A[d]}{A[h]-A[d]}$, její stř. hodnota je $p(h-d)$ a rozptyl $p(1-p)(h-d)$. Vím, že $p_{2i+1} \leq P(|X - p(h-d)| \geq i\sqrt{h-d})$, protože před $2i+1$-tým dotazem musí následovat i unárních dotazů, z nichž každý nalezne $\sqrt{h-d}$ políček s prvky menšími než x.

Potom z Čebyševovy nerovnosti ($P(|Y-\mu| \geq t) \leq \frac{\sigma^2}{t^2}$) plyne $p_{2i+1} \leq \frac{p(1-p)(h-d)}{i^2 n} \leq \frac{1}{4i^2}$. Celkem očekávaný počet dotazů v 1 bloku je $\sum_{i=0}^{\infty} i(p_i - p_{i+1}) = \sum_{i=0}^{\infty} p_i \leq 3 + 2\sum_{i=0}^{\infty}\frac{1}{4i^2} = 3 + \frac{\pi^2}{12} \approx 3.8$. Celk. očekávaný čas dotazů je $\leq 4\log\log n$ a tedy složitost $O(\log\log n)$.

Nejhorší odhad: počet binárních dotazů je stejný jako v binárním hledání, k tomu stejný počet unárních dotazů (max. o 1 na blok víc) a interpolačních dotazů stejně jako bloků – celkem $2(3 + \log(|S|-1)) + 2(1 + \log\log|S|)$ a tedy $O(\log|S|)$.

2.3 Binární vyhledávací stromy

Binární vyhledávací strom T je úplný strom (vše kromě listů má 2 syny), kde mám bijekci mezi množinou S a VNITŘNÍMI vrcholy takovou, že když v je vnitřní vrchol stromu, tak všechny vrcholy podstromů levého syna jsou

$\leq v$ a pravého $> v$. Listy pak reprezentují jednotlivé intervaly mezi vnitřními vrcholy. Můžeme je vynechat, ale s nimi je to (jak pro koho) logičtější.

MEMBER je běžný, INSERT: najít list reprezentující interval, kam vkládám, udělat z něj normální uzel se vkládanou hodnotou a dát mu dva listy s podintervaly. DELETE: najdu vrchol, má-li jednoho syna-lista, pak druhý syn ho nahradí na jeho místě, jinak najdeme a dáme na jeho místo nejmenší větší vnitřní vrchol, kteréhožto levý syn je list a pravého dáme na jeho místo. SPLIT: procházím stromem a hledám x, ost. prvky házím cestou do dvou stromů T_1, T_2, ve kterých si vždy uchovávám ukazatel na list, místo kterého vkládám (odkrojím syna ve kterém hledám dál, místo něj vložím list, na který si pamatuju ukazatel). JOIN: s prvkem navíc, triviální – spojím stromy jako 2 syny nového prvku.

Normální struktura nepodporuje efektivní ORD, podobně jako u (a, b)-stromů je nutné přidat navíc položky, které určují počet listů v podstromu určeném každým vrcholem. ORD je pak běžný – jde do pravých a přičítá levé podstromy dokud může, jinak jde do levého podstromu (a nepřičte nic).

Analýza algoritmů

Korektnost vyhledávání: z definice LHR a PHR (nejbližší levější/pravější na vyšší úrovni) a λ, π (hodnoty klíčů LHR a PHR nebo $-\infty$, resp. $+\infty$, nejsou-li LHR/PHR definovány). Potom je-li T' podstrom $t \in T$, pak T' reprezentuje $S \cap (\lambda(t), \pi(t))$ (a je to největší interval nezastoupený mimo T'). Důkaz indukcí od kořene po synech. Pak pro vyhledání vrcholu x platí $\lambda(t) < x < \pi(t)$, vyšetřuji-li při hledání vrchol t. Z toho je korektní i MEMBER a INSERT, u DELETE musím dokázat případ s přehazováním vrcholů (dostávám bin. strom reprezentující $S \setminus \{x\}$). Korektnost MIN, MAX, JOIN je zřejmá, u SPLIT plyne z korektnosti hledání a toho, že moje označené listy jsou nejlevější, resp. nejpravější. Korektnost ORD plyne z toho, že v každém kroku je k-tý prvek představován tolikátým v pořadí vrcholů akt. vyšetřovaného podstromu, kolik mi zbývá přičíst.

Zpracování 1 vrcholu je vždy $O(1)$ a alg. se pohybuje po nějaké cestě od kořene k listu, která má $O(h)$, kde h je výška stromu.

Vyvážené stromy

Chceme-li zajistit, že výška bude $O(\log |S|)$, dáme pro strom další podmínky, které bude muset splňovat a operace je zachovávat. Je velká pravděpodobnost, že i bez vyvažování strom zůstane $O(\log |S|)$ vysoký a operace na něm můžou tak (bez vyvažování) běhat i rychleji. Existují i pravděpodobnostní postupy, nahrazující vyvažování znáhodněním posloupností operací.

Další možnost jsou samoopravující struktury – operace samy bez dalších uchovávaných dat obstarávají vyvažování, existuje strategie, která zajistí dobré chování bez ohledu na data. Nebo se sleduje chování struktury, a když začne být příliš pomalá, vytvoří se nová – vyvážená. Poslední možnost je upravit dat. strukturu podle známého rozdělení dat.

Pro vyvažovací operace se používá pomocný alg. ROTACE(u, v) – mám v, jeho pravého syna u a podstromy (zleva) A, B, C. Přehodím v pod u, upravím ukazatel v otci a přeházím podstromy. Symetricky = zpátky. Další operace – DVOJROTACE(u, v, w): mám u, jeho levého syna v a jeho pravého syna w, seřadím je tak, že v je otec obou, t vpravo a w vlevo. Přitom opět upravím ukazatele v nadř. uzlu a přepojím podstromy. Taky existuje symetrický případ. Při obou lze aktualizovat i počty listů v podstromě, pracují v $O(1)$.

AVL-stromy

Nejstarší vyvážené stromy, dodnes oblíbené, jednoduše definované, ale detailně technicky složité. Podmínka: *Výška pravého a levého podstromu lib. vrcholu se liší max. o 1.* Ozn. výšku vrcholu $\eta(v)$, na jejím základě určuji $\omega(v)$ – rozdíl výšek levého a pravého podstromu ($\in \{-1, 0, 1\}$). Uchovávat potřebuji jenom ω.

Odhad výšky (η(kořen)) – podstrom AVL stromu je vždy AVL strom, vezmeme rekurzivní vztahy pro největší a nejmenší množinu v AVL stromu výšky i:

$$mn(i) = mn(i-1) + mn(i-2) + 1 \qquad mx(i) = 2mx(i-1) + 1$$

Indukcí dokážeme, že $mx(i) = 2^i - 1$ a $mn(i) = F_{i+2} - 1$, kde F_i je i-té Fibonacciho číslo (pro ty platí vzorec $F_{i+2} = F_{i+1} + F_i$. Víme, že $\lim_{i\to\infty} F_i = \sqrt{5}(\frac{1+\sqrt{5}}{2})^{-n}$ a z toho zlogaritmováním plyne pro AVL-strom o výšce i s n

prvky:

$$\log\left(\frac{c_1}{\sqrt{5}}\right) + (i+2)\log\left(\frac{1+\sqrt{5}}{2}\right) < \log(n+1) < i$$

A tedy $0.69i < \log(n+1) < i$, takže $i = \Theta(\log n)$.

Operace MEMBER je stejná jako pro nevyvážené, INSERT už se musí po (normálním) vložením zabývat vyvažováním. Jde zpět a hledá, který nejnižší vrchol x nemá po vložení vyvážené ω (buď $\omega(x) = 1$ a přidávám do levého podstromu, nebo $\omega(x) = -1$ a přidávám vpravo), cestou upravuje ω, na něm provede vhodnou rotaci nebo dvojrotaci a končí. Korektnost je založena na faktu, že zvětší-li se výška nějakého podstromu, není potom vyvážený ($\omega \neq 0$). Důkaz rozborem případů podle neupravené hodnoty ω(otec) (vkládám-li do levého, ω klesá – změnila/nezměnila se výška? musím pokračovat, dělat rotace?: pro $\omega = 1$ končím, $\omega = 0$ rekurze, $\omega = -1$ jsou 2 případy (ω(můj vrchol) $= -1, 1, 0$ není povolená, pro -1 dělám rotaci a 1 dvojrotaci).

Operace DELETE je vyvažuje podobně jako INSERT, ale potřebuje víc operací (až $O(\log |S|)$ rotací). Založeno na faktu, že přesune-li DELETE vrchol, ten dostává původní η nebo o 1 menší. Důkaz – když se někde snížilo η, rozborem případů podle neupravené hodnoty ω v otci: pro ω(otec) $= 1$ jsou 3 možnosti podle ω(pravý syn), pro 1 rekurze, pro 0 rotace, pro -1 dvojrotace a pak opět 3 případy podle nového kořene, ve všech rekurze. Pro ω(otec) $= 0$ skončím, -1: rekurze.

Červeno-černé stromy

Červeno-černý strom má tyto čtyři povinné vlastnosti:

1. Každý uzel má definovanou barvu, a to černou nebo červenou.
2. Každý list je černý.
3. Každý červený vrchol musí mít oba syny černé.
4. Každá cesta od libovolného vrcholu k listům v jeho podstromě musí obsahovat stejný počet černých uzlů.

Pro červeno-černé stromy se definuje *černá výška uzlu* ($\mathbf{bh}(x)$) jako počet černých uzlů na nejdelší cestě od uzlu k listu.

Garantování výšky – Podstrom libovolného uzlu x obsahuje alespoň $2^{\mathbf{bh}(x)} - 1$ interních uzlů. Díky tomu má červeno-černý strom výšku vždy nejvýše $2\log(n+1)$ (kde n je počet uzlů). (Důkaz prvního tvrzení indukcí podle $\mathbf{h}(x)$, druhého z prvního a třetí vlastnosti červeno-černých stromů)

Algoritmy INSERT a DELETE – Jde také o vložení a následné vyvažování Bez porušení vlastností červeno-černých stromů lze kořen vždy přebarvit načerno, můžeme pro ně předpokládat, že *kořen stromu* je *vždy černý.*

INSERT vypadá následovně:

- Nalezení místa pro vložení a přidání nového prvku jako v obecných bin. vyhl. stromech, nový prvek se přebarví načerveno.
- Pokud je jeho otec černý, můžeme skončit – vlastnosti stromů jsou splněné. Pokud je červený, musíme strom upravovat (tady předpokládám, že otec přidávaného uzlu je levým synem, opačný připad je symetrický):
- Je-li i strýc červený, přebarvit otce a strýce načerno a přenést chybu o patro výš (je-li děd černý, končím, jinak můžu pokračovat až do kořene, který už lze přebarvovat beztrestně).
- Je-li strýc černý a přidaný uzel je levým synem, udělat pravou rotaci na dědovi a přebarvit uzly tak, aby odpovídaly vlastnostem stromů.
- Je-li strýc černý a přidaný uzel je pravým synem, udělat levou rotaci na otci a převést tak na předchozí případ.

DELETE se provádí takto:

- Odstraním uzel stejně jako v předchozím případě. Opravdu odstraněný uzel (z přepojování) má max. jednoho syna. Pokud odstraňovaný uzel byl červený, neporuším vlastnosti stromů, stejně tak pokud jeho syn byl červený – to řeším jeho přebarvením načerno.
- V opačném případě (tj. syn odebíraného – x – je černý) musím udělat násl. úpravy (přep. že x je levým synem svého nového otce, v op. případě postupuji symetricky):
- x prohlásím za „dvojitě černý“ a této vlastnosti se pokouším zbavit.

- Pokud je bratr x (buď w) červený, pak má 2 černé syny – provedu levou rotaci na rodiči x, prohodím barvy rodiče x a uzlu w a převedu tak situaci na jeden z násl. případů:
- Je-li w černý a má-li 2 černé syny, prohlásím x za černý a přebarvím w načerveno, rodiče přebarvím buď na černo (a končím) nebo na „dvojitě černou“ a propaguji chybu (mohu dojít až do kořene, který lze přebarovat beztrestně).
- Je-li w černý, jeho levý syn červený a pravý černý, vyměním barvy w s jeho levým synem a na w použiji pravou rotaci, čímž dostanu poslední případ:
- Je-li w černý a jeho pravý syn červený, přebarvím pravého syna načerno, odstraním dvojitě černou z x, provedu levou rotaci na w a pokud měl původně w (a x) červeného otce, přebarvím w načerveno a tohoto (teď už levého syna w) přebarvím načerno.

MIN a MAX jsou stejné jako pro nevyvážené. JOIN (s prvkem navíc): mám-li černou výšku u obou stejnou, není co řešit, pokud ne, projdu po tom s větší $\mathbf{bh}(x)$ do patra, kde se výšky rovnají, půjčím si přísl. podstrom a slepím s ním, vrátím celek zpátky a aplikuji vyvažování, jako kdybych vložil 1 prvek (poruším výšku max. o 1). SPLIT je podobný jako pro (a, b)-stromy: rozhazuji podstromy do zásobníků, odkud je pak slepuji operací JOIN. Každý alg. pracuje jen s vrcholy na 1 cestě od kořene k listům a s každým dělá konstantně činností, takže všechny algoritmy mají logaritmickou složitost. DELETE volá max. 2 rotace nebo 1 rotaci a 1 dvojrotaci, INSERT zase max. 1 rotaci (i když jdou rekurzivně).

Pro vyrobení ORD je zas potřeba statistik počtu listů v podstromě a funguje to stejně.

Váhově vyvážené stromy (BB-α)

Na ústupu, ale občas se ještě používají. Mějme $1/4 < \alpha < \sqrt{2}/2$, označme $p(T)$ počet listů ve stromě T. Pak strom je BB-α, když

$$\alpha \leq \frac{p(T_{\text{levý}(v)})}{p(T_v)} \leq 1 - \alpha$$

pro T_v jako podstrom určený (každým) vrcholem v. O BB-α stromech platí, že

$$\text{výška}(T) \leq 1 + \frac{\log(n+1) - 1}{\log \frac{1}{1-\alpha}}$$

Takže jsou také vyvážené a operace mají zaručenou logaritmickou hloubku, vyvažuje se na nich také rotacemi a dvojrotacemi. Vždy totiž existuje $\alpha \leq d \leq 1 - \alpha$ takové, že když mám strom, jehož oba podstromy splňují vlastnosti a navíc $p(T_l)/p(T) \leq \alpha$ a $p(T_l)/p(T) - 1$ nebo $p(T_l) + 1/p(T) + 1$ vyhovuje, vezmu $\rho = \frac{p(T')}{p(T_r)}$, kde T' je určen levým synem T_r, a pro $\rho \leq d$ provedu ROTACE(T, T_r), jinak DVOJROTACE(T, T_r, T') a dostanu BB-α strom (bez důkazu). Opačný případ je popsaný symetricky.

Mají pěknou vlastnost, kvůli které se používaly: pro $\forall\alpha$ existuje $c > 0$ takové, že každá posloupnost k operací INSERT a DELETE volá max. $c \cdot k$ rotací a dvojrotací.

3 Haldy

Problém na uspořádaném univerzu, jehož uspořádání se v čase mění, nevyžaduje se efektivní operace MEMBER (často se předpokládá s argumentem operace informace o uložení prvku). Požadují se malé nároky na paměť; rychlost ostatních operací. Vhodné je znát i skutečnou rychlost na nějakém hardwaru. Operace krom běžných: mám $f : S \to \mathbb{R}$, která vyrábí uspořádání a mohu ji měnit: INCREASE, DECREASE změní velikost f na nějakém daném prvku s se známým uložením o $+a$, $-a$. Další operace: DELETEMIN – smazání prvku s nejmenší hodnotou f.

Pro uložení v haldě se používá lokální „podmínka haldy“: $\forall v \in S : f(\text{otec}(v)) \leq f(v)$, případně v duální podobě. Pro operaci DELETE také budeme požadovat přímé zadání uložení prvku, navíc definujeme operace MAKEHEAP (vytvoření haldy při známé množině a f) a MERGE (z 2 hald vytvoření jedné, reprezentující $S_1 \cup S_2$ a $f_1 \cup f_2$, aniž by ověřovala disjunktnost).

3.1 Regulární haldy

Pro d-regulární strom ($d \in \mathbb{N}$) s kořenem r platí, že existuje pořadí synů vnitřních vrcholů takové, že očíslování prohledáváním z r do šířky splňuje:

1. každý vrchol má nejvýše d synů
2. když vrchol není list, tak všechny vrcholy s menším f mají právě d synů
3. má-li vrchol méně než d synů, pak všechny vrcholy s většími čísly jsou listy

Potom takový strom s n vrcholy má max. jeden ne-list, který nemá právě d synů, jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$. Čísla synů vrcholu s číslem k jsou $(k-1)d+2, \ldots, kd+1$, číslo otce je $1 + \lfloor \frac{k-2}{d} \rfloor$. Množina *je reprezentovaná* haldou, když přiřazení prvků vrcholům haldy je bijekce, splňující podmínku haldy. Takto vytvořená halda umožňuje i efektivní reprezentaci v poli.

Není známa efektivní operace MERGE. Máme pomocné operace UP, DOWN, posunující prvek níž/výš ve struktuře, dokud není splněna podmínka haldy („probublávání“). INSERT jen vloží za poslední a spustí UP, DELETE nahradí odstranění posledním listem a volá UP nebo DOWN podle potřeby, DELETEMIN odstraní kořen, nahradí posl. listem a volá DOWN, MIN jen vrátí kořen, INCREASE a DECREASE změní hodnotu f u prvku a zavolají UP, resp. DOWN. Operace MAKEHEAP vytvoří libovolný strom a pak postupně od posledního ne-listu ke kořeni volá na všechno UP.

U všech operací je korektnost zajištěna podmínkou haldy (a tím, že UP a DOWN zaručí její splnění), u MAKEHEAP se uvažuje jiná formulace podmínky – $\forall v \in S : \forall w, w$ syn $v : f(v) \leq f(w)$, kterou splňuje každý list, a po proběhnutí DOWN(v) všechny vrcholy s vyšším číslem než v, takže postupně všechny.

Složitost operací

Běh DOWN chce $O(d)$ a UP $O(1)$ v každém cyklu, takže celkem jde o $O(d \log |S|)$ a $O(\log |S|)$. Haldu lze vytvořit opakovaným INSERTem v čase $|S| \log |S|$, ale pro větší množiny je rychlejší MAKEHEAP: potřebuje čas $O(\sum_{i=0}^{k-1} d^i(k-1)d)$, ozn. $A := \sum_{i=0}^{k-1} d^{i+1}(k-1)$ a spočítám $dA - A$, z toho dostanu $O(d^2|S|)$ jako nejhorší případ.

Aplikace

Třídění haldou – vytvoření haldy a postupné volání MIN a DELETEMIN. Lze ukázat, že pro $d = 3, d = 4$ je výhodnější než $d = 2$, empiricky je pro posl. do 1000000 délky $d = 6$ nebo $d = 7$ nejlepší; pro delší posloupnosti je možné d zmenšit.

Dijkstra – normální Dijkstrův alg., jen prvky uchovávám v haldě, tříděné podle aktuálního d (horního odhadu vzdálenosti). Složitost $O((m+n)\log n)$, pro $d = \max\{2, \frac{m}{n}\}$ je to $O(m \log_d n)$ a pro husté grafy ($m > x^{1+\varepsilon}$) je lineární v m.

3.2 Leftist haldy

Operace jsou založeny na alg. MERGE a DECREASE. Jde o bin. strom (T, r). Označme npl(v) délku nejkratší cesty z v do listu/vrcholu s max. 1 synem. Splňuje podmínky:

1. má-li vrchol 1 syna, pak je vždy levý
2. má-li 2 syny l, p , pak npl(p)≤npl(l)
3. podmínka haldy na klíče prvků (ex. přiřazení prvků vrcholům stromu)

Pro leftist haldu se definuje pravá cesta (posl. pravých synů) a pokud máme takovou cestu délky k z vrcholu v, víme, že podstrom v do hloubky k je úplný binární strom. Délka pravé cesty z každého vrcholu je tedy rovna nejvýše log(|podstromu|).

Hlavní algoritmus – MERGE: rekurzivní. Testuje prázdnost jednoho ze stromů, pak se volá rekurzivně na podstrom pravého syna toho s menším klíčem v kořeni dohromady s celým druhým a výsledek připojí místo onoho pravého syna. Pokud neplatí podmínka na npl, syny vymění. INSERT je to samé co vytvoření jednoprvkové haldy a zavolání MERGE.

MIN je typické vypsání kořene, DELETEMIN je zMERGEování synů kořene (a jeho zahození). MAKEHEAP je vytvoření hald z jednotl. prvků, jejich nacpání do fronty a potom v cyklu: vyberu dva z fronty, zmerguju a hodím výsledek zpátky; dokud mám ve frontě víc než 1 haldu – to je potom výsledek.

INCREASE a DECREASE se dělají jinak, mám pomocnou operaci OPRAV, která odtrhne podstrom a dopočítá všem vrcholům správné npl. Po odtržení vrcholu a příp. přehození pravého syna doleva jde nahoru dokud provádí změny npl (možno až do kořene), vztahuje npl odspoda a příp. prohazuje syny. DECREASE se pak udělá snížením hodnoty ve vrcholu, zavoláním OPRAV, tj. jeho odříznutím od zbytku haldy, a MERGE podstromu a zbytku. INCREASE: zapamatuju si levý a pravý podstrom vrcholu s mým prvkem a provedu na něj OPRAV (vyhodím ho), potom vyrobím nový vrchol s mým prvkem se zvednutou hodnotou a jako samostatnou haldu ho zMERGEuju s levým podstromem. Pravý podstrom zMERGEuju se zbytkem haldy a nakonec s tím zMERGEuju výsledek MERGE levého podstromu a mého prvku.

Složitost

1 běh MERGE bez rekurze je $O(1)$, hloubka rekurze je omezena pravými cestami, takže je to $O(\log(|S_1| + |S_2|))$. Z toho plyne logaritmovost INSERT a DELETEMIN. Pro MAKEHEAP – uvažuje se, kolikrát projdou haldy frontou: po k projití frontou mají velikost 2^{k-1} a tedy fronta obsahuje $\lceil \frac{|S|}{2^{k-1}} \rceil$ hald, jedno projití frontou pro všechny haldy tedy trvá $O(k\lceil \frac{|S|}{2^{k-1}} \rceil)$. Celkem dostávám $O(|S| \sum_{k=1}^{\infty} \frac{k}{2^{k-1}}) = O(|S|)$ (součet řady je 4).

OPRAV chodí jen po pravé cestě, takže má logaritmickou složitost. INSERT, INCREASE a DECREASE se díky ní dostanou taky na $O(\log |S|)$, protože jejich části kromě MERGE a OPRAV mají konstantní složitost.

3.3 Amortizovaná složitost

„Bankovní paradigma“ – mám nějakou funkci w, která popisuje vhodnost jednotlivých konfigurací D. Potom amortizovaná složitost operace o – $am(o) = t(o) + h(D') - h(D)$. Pokud máme nějaký odhad amortizované složitosti operací o_i – c, platí:

$$\sum_{i=1}^{n} am(o_i) = \sum_{i=1}^{n} t(o_i) + w(D_n) - w(D_0) \leq \sum_{i=1}^{n} c(o_i)$$

takže pro $w(D) \geq 0$ platí

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) + w(D_0)$$

a tedy složitost nejhoršího případu celé posloupnosti operací může být mnohem „rychlejší“ než posloupnost nejhorších případů jednotlivých operací.

3.4 Binomiální haldy

Binomiální stromy se definují rekurentně jako H_i, kde H_0 je jednoprvkový a H_{i+1} vznikne z dvou H_i, kdy se kořen jednoho stane dalším (krajním) synem kořenu druhého. Pak strom H_i má 2^i prvků, jeho kořen má i synů, jeho výška je i a podstromy určené syny kořene jsou právě $H_{i-1}, \ldots, H_0$.

Binomiální halda reprezentující S je seznam stromů T_i takový, že celkový počet vrcholů v těchto stromech je $|S|$ a je dáno jednoznačné přiřazení prvků vrcholům, respektující podmínku haldy; každý strom je izomorfní s nějakým H_i a dva $T_i, T_j, i \neq j$ nejsou izomorfní. Toto existuje pro každé přirozené $|S|$, což plyne z existence dvojkového zápisu čísla.

Operace zase založené na MERGE. **MERGE** pracuje stejně jako binární sčítání – za pomoci operace **SPOJ** (slepení dvou stromů, přilepím jako syna toho, který má v kořeni vyšší klíč) slepí stromy stejného řádu, přenáší výsledky do dalšího spojování (přenos + obě haldy mající strom daného řádu = vyplivnutí 1 stromu na výsledek a spojení zbývajících dvou). Složitost – $O(\log |S_1| + \log |S_2|)$, protože 1 krok (SPOJ) je konstantní.

INSERT je MERGE s jednoprvkovou haldou. **MIN** je prohledání kořenů & vypsání nejmenšího. **DELETEMIN** je MIN + odebrání stromu s nejmenším a přidání (MERGE) jeho synů do haldy. **INCREASE a DECREASE** se dělají úplně stejně jako u regulárních hald. Přímo není podporováno DELETE, jen jako DECREASE + DELETEMIN. MAKEHEAP se provádí opakováním INSERT.

Složitost – halda má nejvýše $\log |S|$ stromů, takže MIN a DELETEMIN mají tuto složitost. Výška všech stromů je $\leq \log |S|$, což dává složitost INCREASE $O(\log^2 |S|)$ a DECREASE $O(\log |S|)$. Pro MERGE se použije odhad amortizované složitosti přičítání jedničky k bin. číslu, což je $O(1)$, tedy celkem $O(|S|)$.

Líná implementace

Vynecháme předpoklad neexistence 2 izomorfních stromů v haldě a budeme „vyvažování“ provádět jen u operací MIN a DELETEMIN, kdy se stejně musí projít všechny stromy. MERGE je pak prosté slepení seznamů hald. Vyvažování se provádí operací **VYVAZ**, která sloučí izomorfní stromy (podobně jako MERGE z pilné implementace).

Složitost INSERT a MERGE je $O(1)$, ale DELETEMIN a MIN v nejhorším případě $O(|S|)$. Amortizovaně je to ale lepší. Hodnocení konfigurace budiž počet stromů v haldě. INSERT a MERGE nemění, resp. mění o 1, takže jsou stále $O(1)$. Operace VYVAZ potřebuje $O(k + \sum_{i=0}^{k} |O_i|)$, kde O_i jsou všechny stromy izomorfní s H_i. Bez operace VYVAZ potřebuje MIN $O(h)$ a DELETEMIN $O(h+\log|S|)$, kde h je počet stromů. Dohromady vychází amortizovaná složitost pro MIN: $am(o) = t(o) - w(D) + w(D') = O(h - h + \log|S|)$, protože výsledný počet stromů odpovídá pilné implementaci. Pro DELETEMIN dostanu $O(h + \log|S| - h + \log|S|) = O(\log|S|)$.

3.5 Fibonacciho haldy

Mají amortizovanou čas. složitost pro INSERT a DECREASE $O(1)$ a pro DELETEMIN $O(\log|S|)$, takže se často používají v grafových algoritmech. Praktické porovnání rychlosti s jinými haldami není dosud přesně prostudováno. Definují se jako množiny stromů, jejichž některé vrcholy jsou speciálně označeny, a které splňují podmínku haldy; *musely ale vzniknout posloupností operací z prázdné haldy.* Vrchol je označený, když není kořen a byl mu předtím někdy odříznut syn. Když se vrchol stane kořenem, označení se zapomene. Strom má rank i, má-li jeho kořen i synů (podobné jako izomorfismus s H_i u binomiálních hald).

MERGE, INSERT, MIN a DELETEMIN jsou stejné jako v líné implementaci binomiálních hald, jen požadavek na isomorfismus s H_i je nahrazen požadavkem na daný rank. Pole v MIN a DELETEMIN se prochází do max. ranku $\lfloor c\log(\sqrt{5}|S|+1)\rfloor$, kde $c = \log^{-1}(\frac{3}{2})$. Pomocné operace VYVAZ1 a SPOJ jsou také stejné.

DECREASE, INCREASE a DELETE vycházejí z leftist hald. Používají pomocnou operaci VYVAZ2, která prochází od daného vrcholu ke kořeni a dokud nalézá označené vrcholy, odtrhává je i s jejich podstromy, ruší jejich označení a vkládá do haldy jako zvláštní stromy. DECREASE pak odtrhne podstrom určený snižovaným vrcholem (není-li to už kořen), zruší u něj případné označení a vloží ho zvlášť do haldy, potom volá na odtržené místo VYVAZ2. INCREASE provede to samé, jen ještě roztrhá podstrom zvedaného vrcholu (odtrhne všechny syny, zruší jejich příp. označení a vloží jako samostatné stromy do haldy) a vloží zvednutý vrchol do haldy zvlášť. DELETE je to samé co INCREASE, bez přidání vrcholu zpět do haldy.

Korektnost a složitost

SPOJ podobně jako u binomiálních hald vyrobí z 2 stromů ranku i jeden strom ranku $i+1$. Je nutné dokázat, že stromy ve Fibonacciho haldě mají rank maximálně $c\log(\sqrt{5}|S|+1)$. Operace VYVAZ2 zajistí, že od každého vrcholu kromě kořenů byl odtržen max. 1 syn – když odtrhnu dalšího, odtrhu i tento vrchol a propaguju operaci nahoru.

Složitost operací MERGE a INSERT je $O(1)$, MIN má $O(h)$ pro h počet stromů v haldě (nemění označení vrcholů), DELETEMIN $O(h+$počet synů $v)$, když v je vrchol s nejmenším klíčem (může odznačit některé vrcholy), DECREASE $O(1+c)$, kde c je počet odznačených vrcholů (navíc označí max. 1 vrchol), INCREASE a DELETE $O(1+c+d)$, kde navíc d je počet synů zvedaného nebo odstraňovaného vrcholu (také označí navíc max. 1 vrchol).

Pro výpočet amortizované složitosti volíme hodnotící funkci w jako počet stromů v haldě + 2× počet označených vrcholů. Mám-li daný max. počet synů všech vrcholů v lib. haldě reprezentující n-prvkovou množinu – $\rho(n)$, mohu říct, že amortizovaná složitost MERGE, INSERT a DECREASE je $O(1)$ a MIN, DELETEMIN, INCREASE a DELETE $O(\rho(n))$.

Pro odhad je potřeba znát fakt, že i-tý nejstarší syn libovolného vrcholu má aspoň $i-2$ synů (plyne ze vzniku syna operací SPOJ a odtrhávání VYVAZ2). Potom je vidět, že podstrom nějakého vrcholu s i syny má vždy alespoň F_{i+2} vrcholů (indukcí – podstrom má $1 + \sum_{k=1}^{i} F_k = F_{k+2}$ vrcholů, z vlastností Fibonacciho čísel). Můžu tedy pro n-prvkovou množinu vzít nejmenší F_i takové, že $n < F_i$. Pak má každý vrchol stupeň menší než $i-2$. Protože $F_i = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{1}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{1}\right)^i$, odhadnu druhou půlku rovnice shora jako 3/8 a zlogaritmováním a převáděním dostanu silnější podmínku na n:

$$\log(\sqrt{5}n + \frac{3\sqrt{5}}{8}) \leq i\log(\frac{1+\sqrt{5}}{2})$$

Z toho odhadnu, že když $\frac{\log(\sqrt{5}n+1)}{\log_2 \frac{3}{2}} < i$, pak $n < F_i$ a tedy $\rho(n) < i-2$. Potom tedy $\rho(n)$ je $O(\log n)$ a z toho dostávám složitost operací a korektnost DELETEMIN a MIN.

Aplikace v Dijkstrově algoritmu – dává to složitost $O(m + n \log n)$, takže by to mělo být lepší pro velké, ale řídké grafy oproti d-regulárním haldám. O prakticky zjištěném „zlomu“ nevíme.

3.6 Třídění

Existuje mnoho algoritmů, známe i (za určitých podmínek) dolní odhad složitosti.

HEAPSORT je třídění pomocí (např.) d-regulárních hald, akorát s duální podmínkou haldy a funkcí DELETE-MAX (která ale funguje stejně jako DELETEMIN). Pak se postupně odebírají maxima a setříděná posloupnost se staví na jejich místě od konce pole.

MERGESORT je snad nejstarší, pracuje s frontou, do které na počátku nahází „předtříděné“ rostoucí úseky, potom je v cyklu vybírá a jejich slití vrací zpátky, dokud nemá jen 1 posloupnost. Slití je vybrání vždy menšího na výstup a na konci dokopírování zbytků. MERGE vyžaduje $O(n+m)$, jsou-li 2 posloupnosti dlouhé n a m prvků. První rozházení vyžaduje lineární čas, potom každé projití všech prvků slitím vyrábí posloupnosti délky $\geq 2^{i-1}$, tedy počet projití všech prvků je $\lceil \log n \rceil$ a celk. složitost $O(n \log n)$. Je adaptivní na předtříděné posloupnosti a při omezeném počtu rostoucích úseků dosahuje lineární složitosti.

QUICKSORT je asi nejpoužívanější, v průměru má při rovnoměrném rozložení nejnižší očekávaný čas. Vybere prvek a a vytvoří posloupnosti $a_1 \dots a_{k-1}$ prvků menších než a a $a_{k+1} \dots a_n$ větších než k. Na ty sám sebe zavolá rekurzivně, do výsledku zapíše za sebe obě setříděné poloviny. Procedura (bez rekurze) vyžaduje čas k nebo $n-k$ v 1 běhu, tj. pro a medián, kdyby $n-k=k$, by měl celý algoritmus složitost $O(n \log n)$. Medián lze nalézt v lin. čase, ale pak by byly MERGESORT i HEAPSORT rychlejší, proto se jako a bral např. první prvek posloupnosti. Potom má procedura očekávaný čas $O(n \log n)$, ale nejhorší případ $O(n^2)$ – pro neznámé rozdělení nevhodné. Volba náhodně by celý běh také mohla dost zpomalit, proto se bere medián 3-5 pevně zvolených prvků.

Očekávaný čas QUICKSORTu: prvky a_i, a_j jsou porovnávány max. 1×, a to když a_i nebo a_j je pivot a předtím ani jeden z nich pivot nebyl. Vezmeme si náhodnou veličinu $X_{i,j}$, která má hodnotu 1, když QUICKSORT porovnal během výpočtu b_i a b_j z nějaké výsledné setříděné posloupnosti $(b_i)_{1 \leq i \leq n}$. Potom $\mathbf{E}X_{i,j} = p_{i,j}$, kde $p_{i,j}$ je pravděpodobnost porovnání. Potom celk. počet porovnání v celém běhu je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}X_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}$$

Pro výpočet $p_{i,j}$ uvažujeme „strom rekurze“, kde každý vrchol odpovídá jednomu rek. volání procedury – potom v levém podstromě jsou operace na úsecích prvků menších než pivot $a_i, \dots a_{l-1}$ této posloupnosti a v pravém na větších $a_{l+1}, \dots a_j$. Celkem tedy ke každému vrcholu patří $j-i+1$ prvků a b_j nebo b_i je v nějakém běhu pivot, právě když jemu odp. prvek je první z těchto $j-i+1$ prvků. Takže $p_{i,j} = \frac{2}{j-i+1}$. Počet porovnání pak vyjde

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \int_1^n \frac{1}{x} \mathrm{d}x = 2n \ln n$$

SELECTIONSORT třídí vybíráním nejmenšího prvku a jeho prohozením s levým krajním v nesetříděném úseku. **INSERTIONSORT** vkládá do setříděného úseku další prvek a postupným vyměňováním ho řadí na správné místo.

Porovnání algoritmů

Algoritmus	Čas nejhůř	Čas ∅	Porov. nejhůř	Porov. ∅	PP	Paměť	AD
QUICKSORT	$\Theta(n^2)$	$9n \log n$	$n^2/2$	$1.44n \log n$	A	$n + \log n + c$	N
HEAPSORT	$20n \log n$	$\leq 20n \log n$	$2n \log n$	$2n \log n$	A	$n + c$	N
MERGESORT	$12n \log n$	$\leq 12n \log n$	$n \log n$	$n \log n$	N	$2n + c$	A
A-SORT	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	$O(n \log(F/n))$	A	$5n + c$	A
SELECTIONSORT	$2n^2$	$2n^2$	$n^2/2$	$n^2/2$	A	$n + c$	N
INSERTIONSORT	$O(n^2)$	$O(n^2)$	$n^2/2$	$n^2/4$	N	$n + c$	A

c je nějaká konstanta, F značí počet inverzí v posloupnosti, PP – přímý přístup, AD – adaptivní na předtříděné.

Pro krátké posloupnosti je do délky 22 vhodný SELECTIONSORT, do 15 INSERTIONSORT, jinak QUICKSORT, což vede na hybridní algoritmus. Pro A-SORT jsou nejvhodnější $(2,3)$-stromy. Poměr časů QUICKSORT, MERGESORT, HEAPSORT je v průměru $1 : 1.33 : 2.22$, platilo to ale v roce 1984 :-).

Vylepšení MERGERSORTu

Nedosahuji optimálních výsledků, pokud slévané posloupnosti ve frontách nejsou přibližně stejně dlouhé. Proto provedu úvahu: mějme algoritmus, který slévá rostoucí posloupnosti a uvažujme jeho „slévací“ strom T (kde posloupnost $P(v)$ odp. vrcholu v (délky $l(P(v))$) vznikne slitím posloupností z jeho synů). Součet časů pro MERGESORT je pak $O(\sum\{l(P(v))|v \text{ vnitřní vrchol } T\}) = O(\sum\{d(t)l(P(t))|t \text{ list } T\}$. Dále pracujeme jen s délkami posloupností, vytvoříme algoritmus **OPTIM**, který při slévání sumu minimalizuje – na začátku dá každé jednoprvkové posl. hodnotu c, která odpovídá hodnotě jejího prvku (?). Pro slévání vybírá posloupnosti (stromy) s nejmenším c a slitému stromu přiřadí $c_1 + c_2$. Nakonec zbyde v množině stromů jen jeden, a ten je optimální. Pro třídění fronty podle c se používá BUCKETSORT. Celkem pracuje v čase $O(\sum_{i=1}^{n} l(P_i))$ na posloupnosti rostoucích úseků $P_1, \ldots, P_n$.

3.7 Rozhodovací stromy

Práce většiny třídících algoritmů je založená na porovnávání, takže ji lze popsat binárním stromem, jehož vnitřní vrcholy jsou ohodnoceny porovnáním dvojic prvků. Posloupnost dat (odp. permutaci π na $\{1, \ldots, n\}$) prochází stromem – začíná v kořeni a ve vrcholu ohodnoceném porovnáním a_i a a_j jde pro $\pi(i) < \pi(j)$ do levého syna, jinak do pravého. Končí, když dojde do listu.

Aby byl algoritmus korektní, musí pro každé dvě různé permutace existovat dva listy, tj. listů musí být $\geq n!$. Délka cesty reprezentuje dobu výpočtu algoritmu, takže výška stromu je dolní odhad doby výpočtu, což umožňuje získat dolní odhad složitosti třídících algoritmů. Konkrétní strom je daný algoritmem, nezáleží ale na konkrétních prvcích, jen jejich vztazích (proto je možné uvažovat jen permutace $\{1, \ldots, n\}$). Pro plýtvající algoritmus mohou existovat listy, neodpovídající žádné permutaci, tj. porovnává stejnou dvojici prvků $2\times$ (a jedna z možností už nemůže nastat).

Rozhodovací strom třídícího algoritmu ***A*** *pro n-prvkové posloupnosti* je binární strom, jehož vrcholy jsou ohodnocené porovnáními, a v němž pro každou permutaci π na $\{1, \ldots, n\}$ platí, že posloupnost při třídění permutace π algoritmem **A** je stejná jako posloupnost porovnání při průchodu permutace tímto stromem.

Pro rovnoměrné rozdělení je očekávaný čas průměrná délka cesty od kořene k listům, nejhorší čas je výška stromu. Definujeme $S(n)$ jako minimum délek nejdelších cest z kořene do listu přes všechny stromy s $n!$ listy a $A(n)$ minimum průměrných délek. Potom $S(n) \geq \log(n!)$ a ze Stirlingova vzorce $n! = \sqrt{2\pi n}(\frac{n}{e})^n(1+\frac{1}{12n}+O(\frac{1}{n^2})$ a faktu, že $\frac{1}{\ln 2} = \log e$ dostanu odhad

$$S(n) \geq \log n! \geq \left(n + \frac{1}{2}\right) \log n - \frac{n}{\ln 2}$$

Pro odhad $A(n)$ definujeme $B(k)$ jako minimum součtů délek všech cest z kořene do listů přes všechny stromy s k listy. Dokážeme, že $B(k) = k \log k$, proto se stačí omezit na úplné binární stromy (u všech ost. dostanu větší hodnoty) a postupovat indukcí (ze dvou podstromů s k_1, k_2 listy dělám jeden s $k = k_1 + k_2$ listy). Dostanu nerovnost $k_1 + k_2 + k_1 \log k_1 + k_2 \log k_2 \geq k \log k$ a převedu ji na funkci $f(x) = x \log x + (k - x) \log(k - x) + k - k \log k$, najdu, že její minimum je 0, takže původní nerovnost platí.

3.8 Přihrádkové třídění

Algoritmus **BUCKETSORT** třídí jen přirozená čísla z intervalu $\langle 0, m\rangle$ a to zavedením $m + 1$ množin, do kterých je rozhází a nakonec tyto spojí do výsledku. Třídění je stabilní pro opakující se prvky, inicializace množin a projití při konkatenaci potřebují $O(m)$, rozházení prvků pak $O(n)$, takže celkem $O(n + m)$.

Sofistikovanější verze **HYBRIDSORT** třídí reálná čísla z $\langle 0, 1\rangle$, má dané α (počet přihrádek v poměru k n), rozhazuje do $k = \alpha n$ přihrádek a ty pak třídí haldou. Nejhorší možný čas je $O(n \log n)$, protože nejhůře se může stát, že všechny prvky nacpu do 1 přihrádky. Očekávaný čas pro nezávislé rovnoměrně rozdělené prvky je $O(n)$ – pravděpodobnost velikostí množin se řídí binomickým rozdělením s parametrem $1/k$, střední hodnota pak je $k +$

$k\left(\frac{n(n-1)}{k^2}+\frac{n}{k}\right)=O(n)$. I kdybychom použili algoritmus s kvadratickou složitostí ve třídění jednotl. přihrádek, zůstane očekávaná složitost lineární.

WORDSORT je modifikace pro třídění slov: Příprava: rozhodí slova do množin L_i podle jejich délky, pak vytvoří $\{(j, a_i[j])\}$, kterou setřídí podle druhé složky BUCKETSORTem a výsledek setřídí podle první složky (stejně). Výsledek (dvojice) pak rozstrká do množin S_i podle pořadí písmena ve slově, když odstraní duplicity.

Pak v hlavním cyklus postupuju od největší možné délky a pro každé i nejdřív podle i-tého písmena rozhodím všechna slova délky $\geq i$ do množin T_x. Potom podle množiny S_i vyberu všechna neprázdná T_x a seřadím je za sebe.

Výpočet délek slov, inicializace L_i a zařazení slov do L_i vyžadují čas $O(l)$, kde L je součet délek všech řetězců. Vytvoření seznamu dvojic a jeho třídění vyžaduje také $O(l)$. Založení S_i a přeházení dvojic do nich také $O(l)$. Inicializace T_x je $O(|\Sigma|)$, kde $|\Sigma|$ je velikost abecedy. V hlavním cyklu v každém kroku potřebuji dvakrát čas $O(l_i)$ (součet délek slov dlouhých i nebo víc). Celkem tedy $O(l+|\Sigma|)$.

3.9 Pořadové statistiky

Chceme nalézt k-tý nejmenší prvek nějaké množiny. První algoritmus **FIND** se chová podobně jako QUICKSORT: zvolí si pivot, rozdělí posloupnost na množiny prvků větších a menších než pivot a pokračuje v jedné z nich, v té, ve které by se k-tý nejmenší měl nacházet (podle jejich velikostí). Takto jde, dokud neoddělí množinu velikosti 1. V nejhorším případě má složitost $O(n^2)$. Očekávaný čas odhadneme na $4n$ a dokážeme indukcí z rekurzivního vzorce $T(n,i)=n+\frac{1}{n}(\sum_{k=1}^{i-1}T(n-k,i-k)+\sum_{k=i+1}^{n}T(k,i)$.

Alogritmus **SELECT** najde k-tý nejmenší i v nejhorším případě v lineárním čase. Má threshold $n=100$, pod kterým množinu přímo třídí, pro větší rozdělí do 5-tic a z pro každou z nich najde (přímo) medián. Pak z mediánů pětic najde medián rekurzivně. Prvky rozdělí na větší a menší než medián mediánů a pokračuje rekurzivně na jedné z množin. Korektní je podobně jako FIND, složitost plyne z toho, že v každém kroku (který už přímo netřídí) se zbavím alespoň $\lfloor\frac{3n}{10}\rfloor-1$, takže mi zbyde $\leq\frac{8n}{11}$, důkaz počtu kroků z rekurentního vzorce indukcí.

FIND bývá rychlejší než SELECT pro většinu případů. Je známo, že medián lze nalézt na $3n$ porovnání a že dolní odhad počtu porovnání je $2n$.