- samoupravující se seznamy – viz. proj. V. Koubka
- sylabus – viz výtah / web – cíl: zajímavé myšlenky / koncepty → do šířky, spíše než do hloubky
- DÚ – ∅

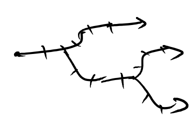
Persistentní datové struktury

- struktury, které dovolují přístup a modifikaci svých dřívějších verzí
- použití:
  - textové editory – operace UNDO/REDO
  - funkcionální jazyky
  - distribuované dat. str.
  - algoritmy výpočetní geometrie

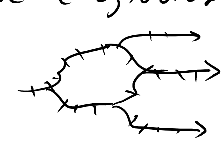
typy

- částečně persistentní
  - lze modifikovat poslední verzi a přistupovat k libovolné dřívější (RO)

- plně persistentní
  - lze přistupovat a měnit libovolnou předchozí verzi → nové verze



- splývavá persistence
  - lze kombinovat různé verze a vytvořit z nich novou verzi



- funkcionální persistence
  - dřívější verze jsou nedotknutelné (RO), ale můžu s libovolnou verzí nakládat jak chci, pokud ji nezměním. (mohu ji zakomponovat do nové verze)

triviální řešení:

1) před každou operací celou strukturu překopíruji a na nové kopii provedu změny → další verze

problém: čas & prostor

2) uchovávám si historii všech provedených operací nad dat. str., chci-li přistoupit k nějaké verzi, "přehraji" si historii až po danou verzi

problém: čas, prostor OK

3) kombinace 1) & 2) - historii rozdělím na intervaly a uchovávám kompletní verze po každém intervalu.
→ stačí přehrát změny v rámci jednoho intervalu.

trade-off: čas vs. prostor

↑ nic moc

částečná persistence

- pole - každý prvek si pamatuje všechny svoje změny → záznam změny: (verze, nová hodnota)
  → bin. vyhledávací strom změn uspořádaný dle čísla verze

  čas: každá operace $O(\log n)$ - krát delší

  prostor: úměrný počtu jednotlivých elementárních změn (přiřazení)

lze lépe:

- datové struktury sestávající ze souboru navzájem prolinkovaných záznamů jednoho typu

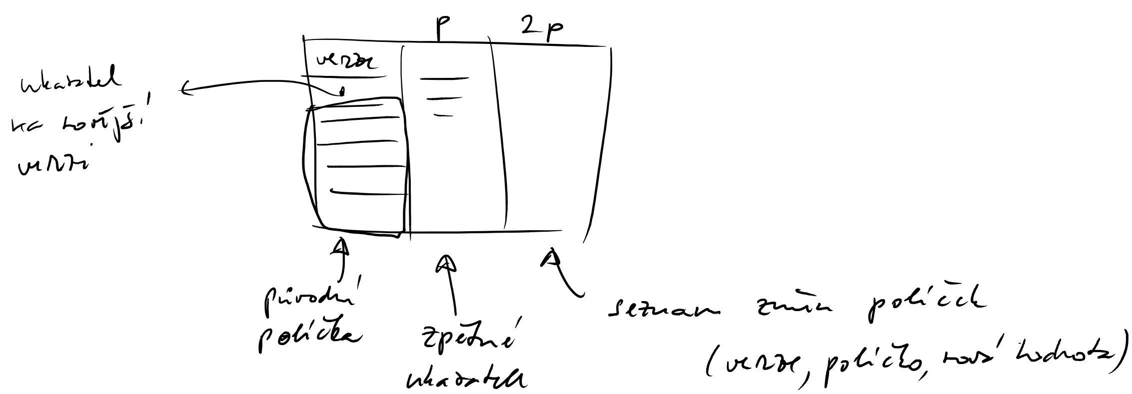
např.: spojové seznamy, vyhledávací stromy, ...

- každý záznam má konstantní počet políček, každé políčko je buď ukazatel, nebo jednoduchá hodnota (real, integer, char, ...)

<u>Tvrzení</u>: Pokud DS má tu vlastnost, že pro nějaké pevné p, na každý záznam odkazuje nejvýše p jiných záznamů, pak lze strukturu udělat částečně perzistentní tak, že čas na operaci je $O(1)$-krát delší (amortizovaně) a paměťové nároky jsou úměrné počtu všech elementárních operací provedených nad touto d.s.

- elementární operace = přiřazení hodnoty políčku v záznamu.

<u>Dk</u>: <u>implementace</u>: rozšířený záznam



k d.s. se přistupuje přes přístupový ukazatel (na "kořen")

→ pole indexované verzí, které udává odkaz na kořen dané verze.

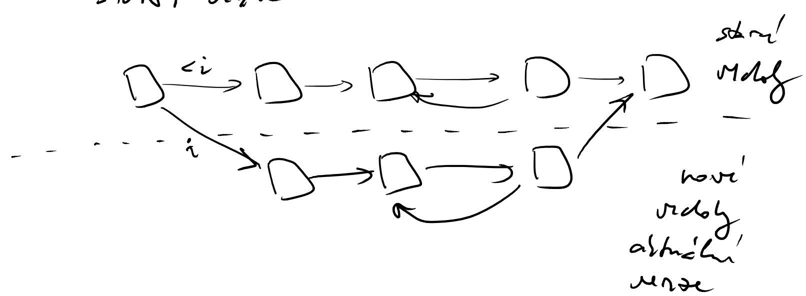
- čtení políčka v kontextu verze $\underline{i}$
  - v aktuálním záznamu projdu seznam změn a vrátím nejnovější hodnotu políčka verze $\leq \underline{i}$.

• zápis políčka v kontextu nejnovější verze i

• pokud je v aktuálním záznamu místo pro další změnu, přidej záznam o změně.
(Pokud se mění ukazatel, zaktualizuj zpětné ukazatele v odkazovaných záznamech.)

jinak vytvoř nový záznam s aktuálním číslem verze, políčkům přiřaď nejnovější hodnoty, zaktualizuj zpětné ukazatele všem záznamům, na které ukazujeme,
a rekurzivně zaktualizuj všechny ukazatele, které mají ukazovat na nový vrchol
(jejich seznam udávají naše zpětné ukazatele)

→ vznikne tak množina nových vrcholů, které ukazují na sebe a případně na starší verze



Analýza:

$\Phi = c \cdot \sum_{\text{záznamy}}$ # záznamů o změně v nejnovější verzi

amortizovaný čas $\le \underbrace{c}_{\text{skutečný čas}} + \underbrace{c}_{\text{změna } \Phi \text{ = přidání záznamu o prvotní změně}}$

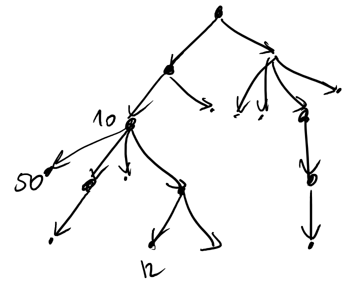
$$\underbrace{lc}_{\text{tvorba nových vrcholů}} + (\underbrace{-2cpl}_{\text{změna } \Phi \text{ při tvorbě nových vrcholů}} + \underbrace{cpl}_{\text{tvorba nových vrcholů}}) \le 2c$$

tvorba nových vrcholů nutno vytvořit pl nových záznamů o aktualizaci

l … # nových vrcholů

Úplná perzistence

Strom verzí



linearizace

– průchod do hloubky, při první návštěvě vrcholu i vypiš b_i (levá závorka), při posledním opuštění i vypiš e_i ($\approx$ pravá závorka)

. . . . b_{10} . . . $b_{12}\, e_{12}$ . . . e_{10} . . . .

$b_{50}\, e_{50}$

$X=27$ . . . . $X=13$ $X=27$ . . . . $X=$…

- při vytvoření nové verze i z verze j, vlož $b_i e_i$ bezprostředně za b_j (při přidání v. 50 pod v. 10)

- každá elementární změna políčka záznamu odpovídá nové verzi, pokud např. nastavíme $X = 13$, pak verze b_{50} má za úkol nastavit $X=13$ a verze e_{50} má za úkol nastavit X na původní hodnotu, jakou jí má ve verzi před b_{50}, tj. ve verzi b_{10}

⇒ hodnota $X=13$ tak bude platná mezi verzemi b_{50} e_{50} (i po přidání dalších verzí odvozených z v. 50, pokud nemění hodnotu X)

⇒ pokud chci znát hodnotu X ve verzi i stačí najít nejbližší verzi $\leq b_i$ v lineárním uspořádání, která nastavuje tuto hodnotu a to je též hodnota X ve verzi i.

→ přidání nové verze i, která mění hodnotu X, znamená přidat verze b_i a e_i do lineárního seznamu verzí, kde b_i nastaví X na novou hodnotu a e_i nastaví původní hodnotu platnou před b_i. (Do d.s. přidám nejdříve e_i, pak b_i.)

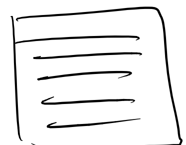
udržování lineárního seznamu verzí

chci operace:
- přidej nový vrchol b ihned za a
- rozhodni zda a je v seznamu před b

– $\exists$ d.s., která zajišťuje obě operace v čase $O(1)$ (amortizovaně, lze i v nejhorším případě)

---

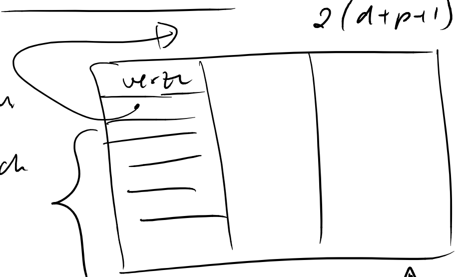
originální d.s. – záznamy



- d políček
- $\leq p$ odkazů na každý záznam v daném okamžiku

→ lze udělat plně perzistentní, čas na operaci se prodlouží $O(1)$-krát (amortizovaně), prostor úměrný počtu elementárních změn políček

DÚ: implementace



d původních políček

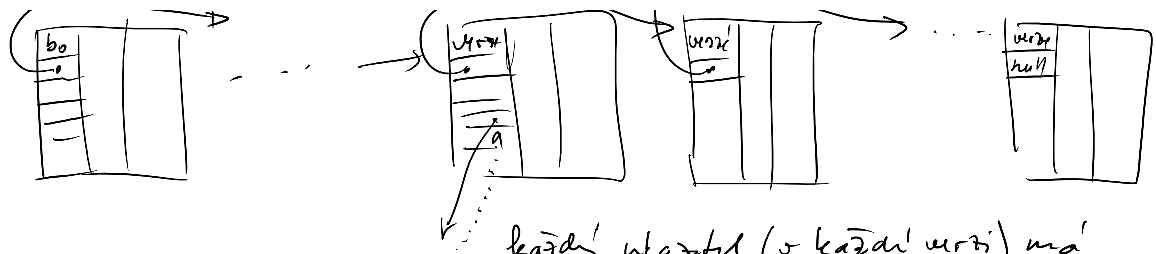
p zpětných odkazů

seznam změn políček (verze, políčko, hodnota)

→ zaznamenávají se i změny zpětných odkazů (rozdíl oproti částečné pers.)

≈ odkazy jsou obousměrné

→ rodina záznamů odpovídající různým verzím původního záznamu

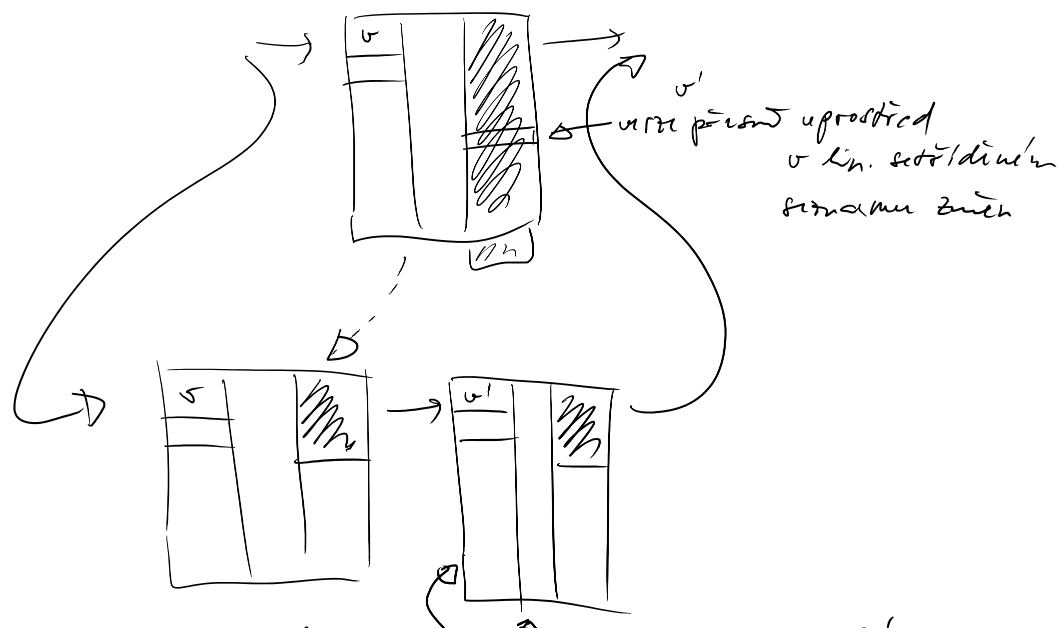


každý ukazatel (v každé verzi) má
k sobě zpětný ukazatel stejné verze

políčka ve verzi v

čtení: (jako u částečné per.) v aktuálním záznamu, který by měl obsahovat hledanou verzi políčka, projdu seznam změn políčka a použiji hodnotu z (lineárně) poslední verze předcházející nebo rovné v.

zápis: • pokud lze přidat zápis o změně, přidám zápis, [změním-li ukazatel, zaktualizuji zpětný ukazatel]
jinak musím stěpit aktuální záznam na dva

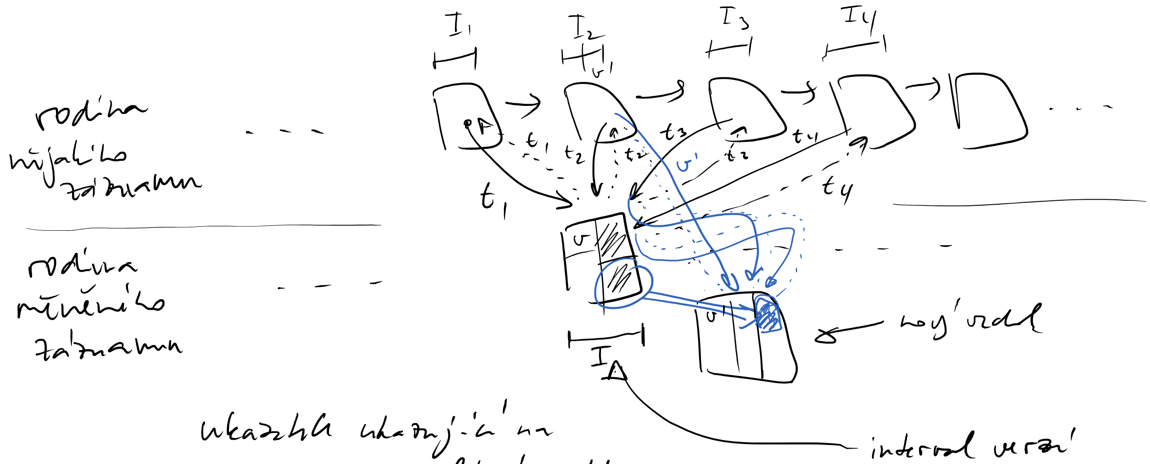


původní vrchol si ponechá všechny změny do verze $< v'$

nový vrchol si ponechá všechny změny po verzi v', přičemž iniciální hodnoty jsou nastaveny dle verze v'.

$\Rightarrow$ oba záznamy obsahují $\leq d+p+1$ změn.

co s ukazateli:



ukazatele ukazující na záznam v, které vznikly v čase $t_i \geq v'$ se přesměrují na nový záznam v' <u>přepsáním</u> jejich hodnoty

$I \subseteq I_1 \cup I_2 \cup I_3 \cup I_4$

$v' \in I_2$

pokud není shodou okolností žádný z těchto ukazatelů verze v', pak přidám do příslušného záznamu (I_2) <u>záznam o změně</u>, že od verze v' ukazuje daný ukazatel na nový vrchol

→ může dojít k přetečení tohoto záznamu a dalším rekurzivním štěpení

(rekurze se zastaví nejpozději v okamžiku, kdy každá verze má svůj vlastní záznam. (Změny hodnot ve verzi starší jako verze záznamu se aplikují přímo na inicializační hodnoty.))

- podobná procedura se použije pro ukazatele vycházející z <u>v</u>. (Ty, které se přesunou, přepíšeme odpovídající zpětné ukazatele, pak přidáme zpětné ukazatele na verzi v' pro iniciální hodnoty ukazatelů záznamu v'.)

<u>Pozn.:</u> rekurzi je třeba dělat průchodem do šířky:

1) rozštěpíme všechny záznamy, které v minulém kole přetekly

1) rozštípnu všechny záznamy, které v minulém kole přetekly

2) přidám potřební ukazatele na právě rozštípené záznamy. Pokud se nový ukazatel již nevejde do seznamu změn příslušného záznamu, uložím změnu do dočasného bufferu a poznamenám si, že záznam je nutné v dalším kole rozštípit.

- V daném kole může záznamu přibýt nejvýše $3d+3p+2$ noví ukazatele, za každý existující ukazatel nejvýše jeden nový.

  (ť. jeho verzi)

  $\Rightarrow$ buffer u každého záznamu je velký $O(1)$ a lze s ním pracovat v konstantním čase

  (po rozštípení může každý z nových záznamů mít jestě $\leq \frac{1}{2}(d+p)$ nezpracových záznamů v bufferu $\to$ další štěpení v příštím kole)

  resp. $\leq (d+p)$

analýza $\quad \Phi = c \sum_{\text{záznamy}} \max\left(0, \#\text{záznamů o změně} - (p+d+1)\right)$

- každý rozštěpený záznam vygeneruje $\leq p+d$ záznamů o změně ukazatelů a zpětných ukazatelů (ve verzi v') pro svoje inicializační hodnoty v ostatních záznamech. (Některé z nich můžou způsobit další štěpení.)
  (Ostatní ukazatele se připojují na místo.)

$\to$ dojde k $\underline{l}$ štěpení

- amort. čas zápisu $\leq c + c$

  skutečný čas, změna potenciálu

  $+cl \qquad + \left(-c\,2(d+p+1)\,l + c(d+p)\,l\right)$

$$+c\ell \quad + \left(-c2(d+p+1)\ell + c(d+p)\ell\right)$$

práce spojená se vznikem ℓ vrcholů a potřebnými aktualizacemi

pokles potenciálu v důsledku rozštěpení ℓ vrcholů

horní odhad na nárůst potenciálu v důsledku nových záznamů o změnách vzniklých v důsledku štěpení $\underline{\ell}$ vrcholů)

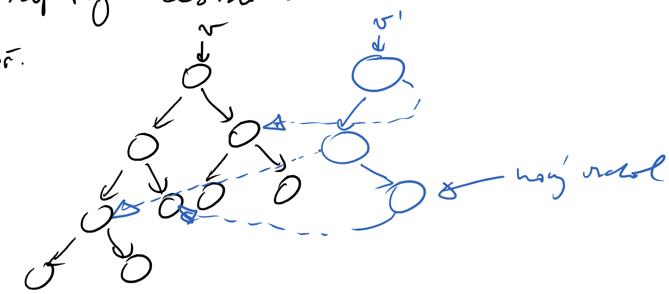
$$\le 2c$$

funkcionální perzistence

Př.: • červeno-černé stromy s aktualizací shora dolů bez rodičovského ukazatele.

— při Insertu, jak strom procházím a vyvažuji, kopíruji cestu do nové verze

př.

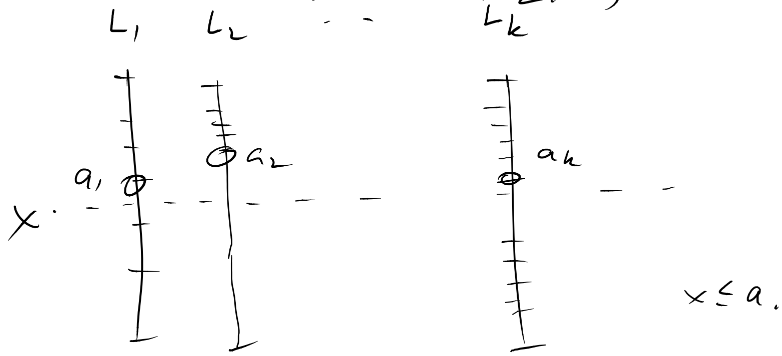


→ přibude $O(\log n)$ nových vrcholů

• čas na opravu $O(\log n)$, tj. asymptoticky jako předtím

Problém: Množiny $L_1, L_2, \dots L_k \subseteq \mathbb{N}$, $|L_i| \le n$. Chci d.s., která na dotaz $\underline{x}$ ($\in \mathbb{N}$), vrátí nejbližší menší nebo rovno číslo z každé množiny L_i.

→ ideálně – čas $O(\log(n) + k)$
prostor $O(\sum |L_i|)$

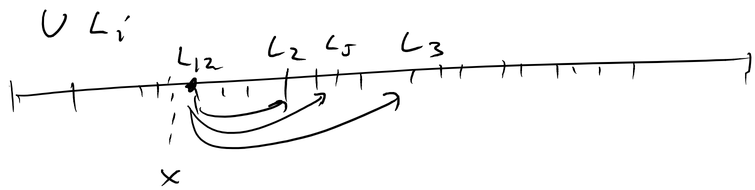


jednoduché řešení

1) každou množinu L_i reprezentuji setříděným polem → čas $O(k \cdot \log n)$
prostor $O(\sum |L_i|)$

2) množiny sjednotím, setřídím a pro každý prvek ve sjednocení si pamatuji ukazatel na nejbližší výskyt prvku z každé množiny, tj. k ukazatelů na prvek

$\bigcup L_i$



čas $O(\log n + k)$
prostor $O(k \cdot \sum_i |L_i|)$

Řešení – kaskádovité podíly ("fractional cascading")

definujme: $M_k = L_k$
a pro $i < k$, $M_i = L_i \cup \{$každý druhý prvek z $L_{i+1}\}$

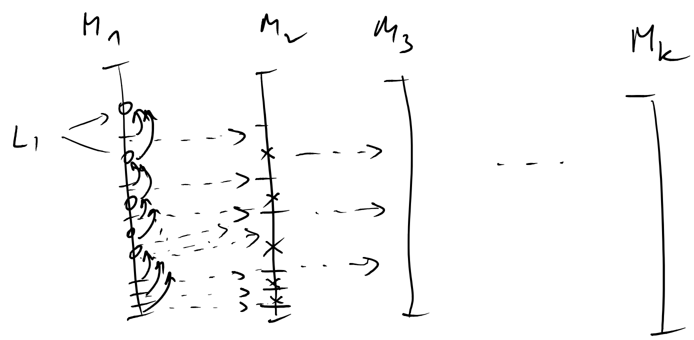
- $|M_i| \le \sum_{j \ge i}^{k} \frac{|L_j|}{2^{j-i}}$ Dk: indukcí na i. ■

$\Rightarrow \sum_i^k |M_i| \le \sum_i^k \sum_j^k \frac{|L_j|}{2^{j-i}} \le \sum_i^k \sum_j^k \frac{|L_j|}{\ldots}$

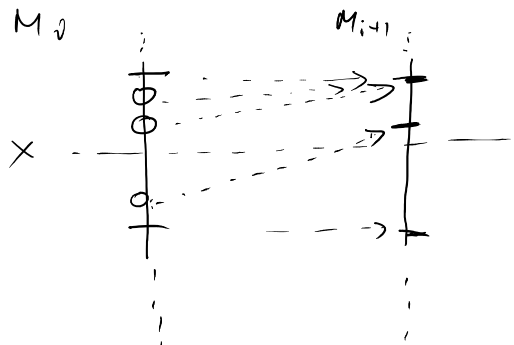
$$\Rightarrow \sum_{i=1}^{k} |M_i| \le \sum_{i=1}^{k} \sum_{j \ge i}^{k} \frac{|L_j|}{2^{j-i}} \le \sum_{i=1}^{k} \sum_{j=1}^{k} \frac{|L_i|}{2^{j}} \le 2 \sum_{i=1}^{k} |L_i|$$

- místo L_i reprezentuji setříděným polem M_i.

  \+ každý prvek z M_i má ukazatel na nejbližší větší prvek z L_i & na nejbližší větší nebo rovno prvek v reprezentaci M_{i+1}



- pomocí binárního vyhledávání / bin. vyhledávacího stromu, nalezneme nejmenší takový prvek M_1 větší x. To mi umožňuje nalézt nejbližší prvek k x v L_1, přeskočím do M_2 a naleznu interval pro x pomocí jednoho porovnání



pokračuji pro další M_i až do $i=k$

$\rightarrow$ čas $O(\log n + k)$

bin. hledání v M_1 — průchod přes k polí M_i

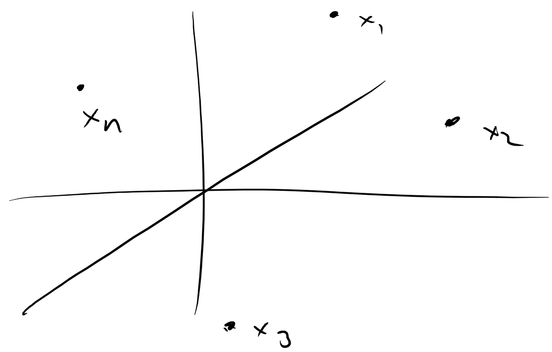
[illegible]

bin. hledání v M_1 pole M_i

prostor $O(\sum_i |M_i|) = O(\sum |L_i|)$. ✓

vícedimenzionální data

bod v $\mathbb{R}^d$

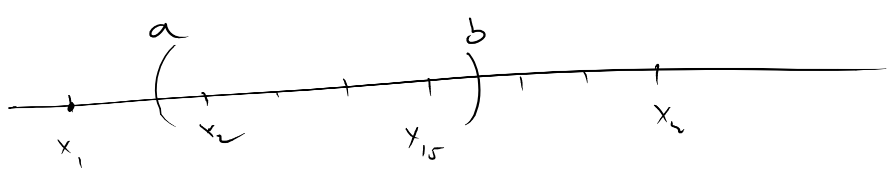


- vyhledávání, přibližné vyhledávání
- intervalové dotazy – které body jsou v obdélníku
  $(x_1, x_2) \times (y_1, y_2) \times \dots (z_1, z_2)$

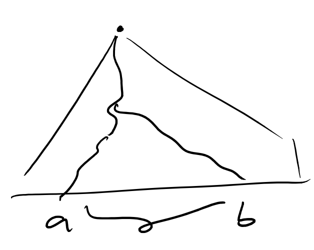
  – databáze, výpočetní geometrie...

→ kd-stromy

- $d=1$ dotaz (a, b)



bin. vyhl. strom



body mezi a a b

- u vyváženého stromu $O(\log n + k)$ čas na dotaz

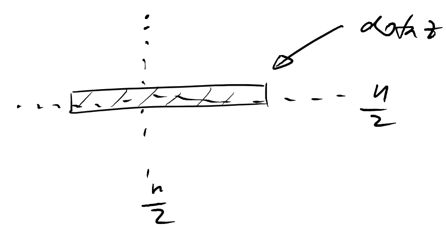
  ↑ počet bodů výstupu

- počtový dotaz (zajímá nás pouze počet bodů v daném intervalu)

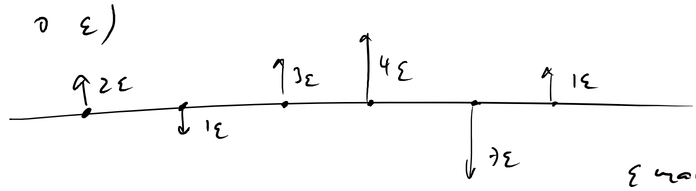
čas $O(\log n)$, pokud si ve vnitřních vrcholech udržuji počet listů v podstromu

• $d = 2$ ... 2-dim

Př.:



BÚNO: žádné dva body nemají totožnou žádnou souřadnici (každý bod můžeme posunout o ε)

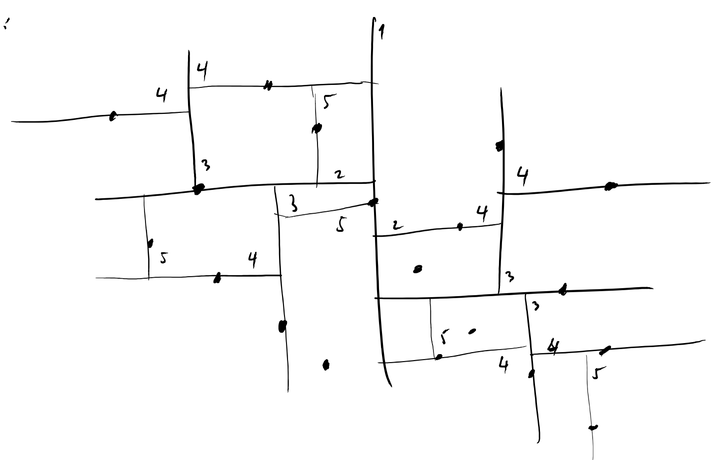


ε malé

2-dim kd-strom

- v lichých vrstvách se rozhodujeme podle x-ové souřadnice, v sudých podle y-ové.

Př.:



• výška stromu je $O(\log n)$ (přesně $\log n \pm 1$)

- každý uzel odpovídá určité oblasti v rovině (obdélník / nekonečný obdélník) tu lze spočítat při průchodu od kořene

• Find (vrchol v, interval R)

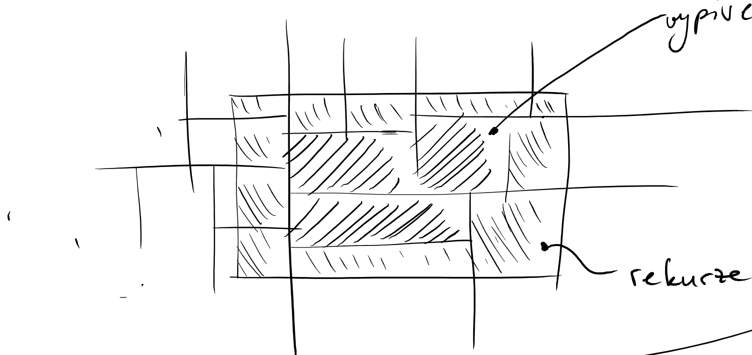
• Find (vrchol v, interval R)

Vstup – v ... vrchol podstromu kd-stromu, R ... interval zájmu

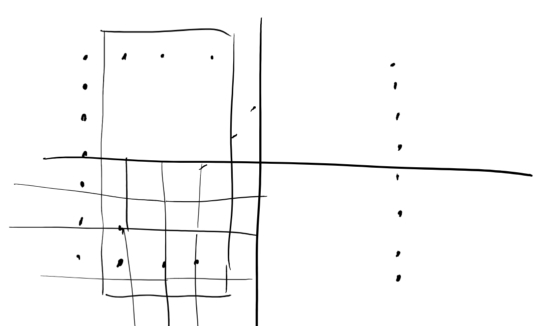
Výstup – vypíše všechny body v intervalu R, které jsou v podstromu v

1) Pokud je v list, vypiš příslušný bod pokud leží v R
2) Pokud oblast levého podstromu v_L je celá obsažena v R,
   vypiš všechny její body
   jinak pokud tato oblast protíná R, → Find(v_L, R)
3) Pokud oblast pravého podstromu v_R leží celá v R,
   vypiš všechny její body
   jinak pokud tato oblast protíná R → Find(v_R, R).
4) End.

Časová složitost:



Proč: Př.:



mřížka $\sqrt{n} \times \sqrt{n}$, každý bod posunut o zanedbatelné ε.

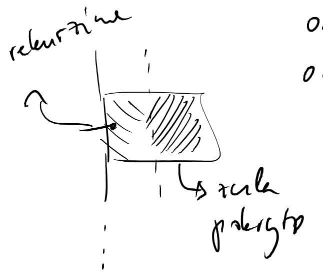
• čas $\Omega(\sqrt{n})$ i když obdélník může být prázdný ($k=0$)

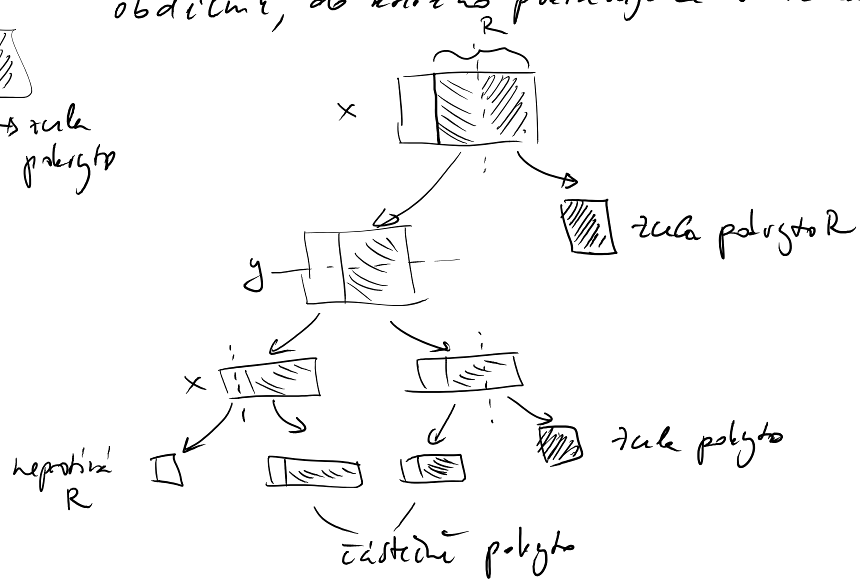
(hubený obdélník )

(hubený obdélník

• uvažujme levou hranici a neúplně pokryté obdélníky R k ní přilehlé. Rozhodování podle osy x odštěpí obdélník, který je zcela pokryt a jeden obdélník, do kterého pokračujeme v rekurzi.

→ nebo disjunktní s R





→ v každé druhé vrstvě se větvíme na 2, v ostatních vrstvách se nevětvíme

→ $\frac{1}{2} \log n$ větvení → velikost větvícího podstromu $\approx 2^{\frac{1}{2}\log n} = \sqrt{n}$.

→ čas $O(\sqrt{n})$ na větvení
+ $O(k)$ na výpis bodů ve zcela pokrytých obdélnících.

$\underline{d > 3}$ střídáme větvení dle jednotlivých dimenzí

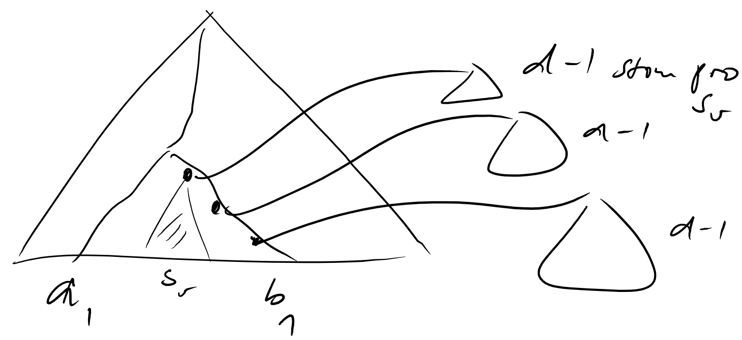
čas $O(n^{1-\frac{1}{d}} + k)$ na dotaz, prostor $O(n)$.
("stejná" analýza – až na každou d-tou vrstvu větvení na 2)

Intervalové stromy ("range trees")

• $d = 1$ ... stejné jako výše.

• $d > 1$

binární vyhledávací strom podle první souřadnice, každý vrchol ukazuje na intervalový strom dimenze $d-1$, kde jsou uloženy všechny prvky odpovídající podstromu podle první souřadnice.

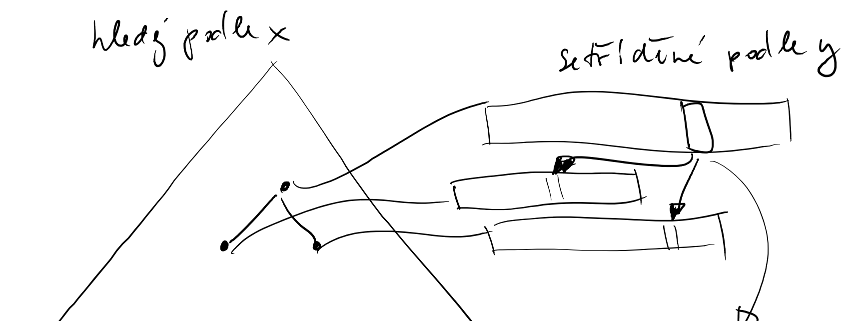


vyhledávání $O(\log^d n)$

prostor $O(n \log^{d-1} n)$

• vyhledávání pro $d=2$ lze zlepšit na $O(\log n)$ pomocí techniky kaskádování

→ pro obecné d čas $O(\log^{d-1} n)$ na intervalový dotaz



prvek ukazuje na nejbližší prvek v pomocné struktuře (setříděné podle y) pro každého ze synů

→ při průchodu hlavním stromem podle x se zároveň navigujeme v pomocném poli podle y.

poři podle y!

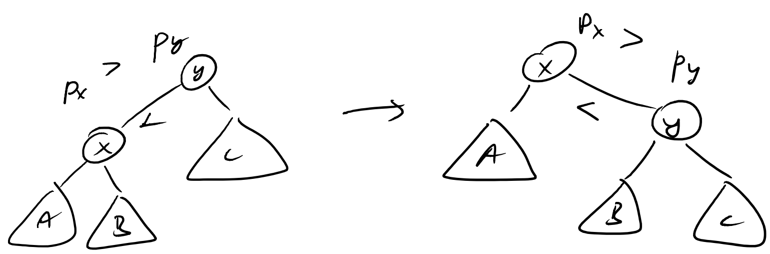
- Náhodné vyhledávací stromy

Treaps

- binární vyhledávací strom, kde každý vložený prvek má navíc přiřazenou prioritu a vrcholy stromu splňují, že otec má vždy vyšší prioritu než synové. (tj. strom zároveň tvoří haldu.)

Operace: Find(x) - jako u bin. vyhl. stromu

Insert(x) - pro prvek zvol náhodnou prioritu z [0; 1], vlož ho do nového listu, kam by měl patřit jako u bin. vyhl. stromu. Pokud nový vrchol x porušuje podmínku priorit vůči svému otci y, prováděj rotaci, dokud není vše v pořádku. (Vrchol x bublá nahoru.)



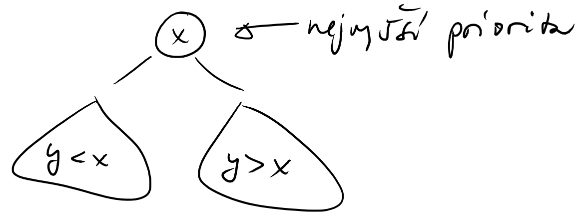
Delete(x) - sniž prioritu vrcholu x na $-\infty$ a pomocí rotací jej probublej až do listu. Smaž list x.

→ Treaps existují a lze přidávat nové prvky.

Pozorování: Tvar stromu je jednoznačně určen pořadím prvků a jejich prioritami.

Dk: 1) kořen je prvek x s nejvyšší prioritou. Levý podstrom je tvořen prvky menšími než kořen $\{y;\ y<x\}$ a pravý podstrom obsahuje prvky větší než x $\{y;\ y>x\}$.

Oba podstromy pak mají jednoznačně určený tvar svými hodnotami a prioritami. (indukce/rekurze)



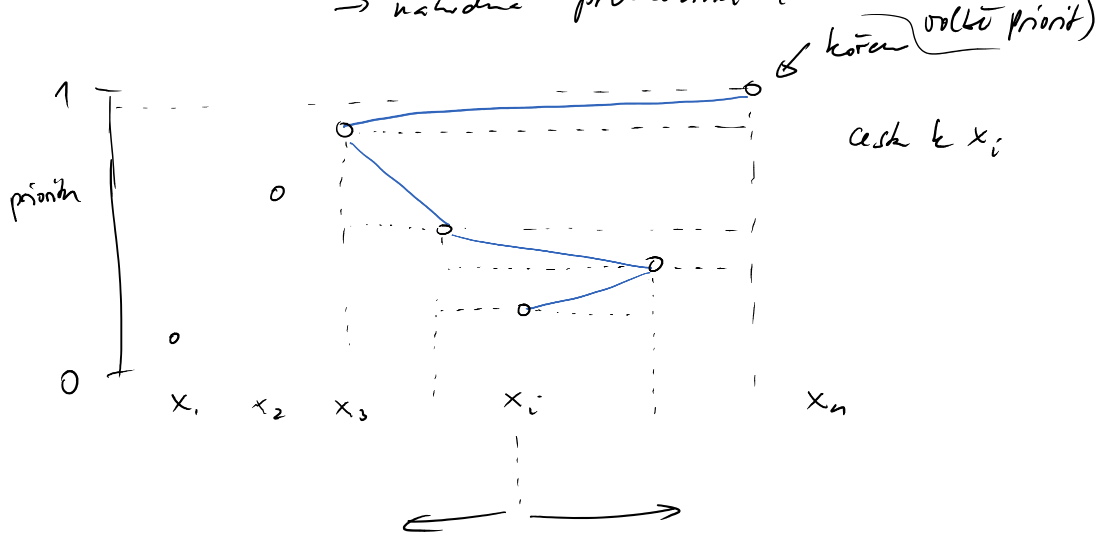
BÚNO: všechny priority různé.

- Čas na operaci nad Treap je úměrný hloubce stromu. Strom má tvar jako binární vyhl. strom, který vznikne při vkládání daných prvků do binárního vyhl. stromu v náhodném pořadí — každý prvek má stejnou šanci být kořen...

(bez vyvažování)

Treap s prvky $x_1 < x_2 < x_3 \dots < x_n$ (pevně danými)

- hloubka vrcholu x_i pro pevné i
  → náhodná proměnná (záleží na náhodné volbě priorit)



- Pro $\forall j < i$, x_j leží na cestě od kořene k x_i právě tehdy, když x_j má nejvyšší prioritu mezi $\{x_j, x_{j+1}, \dots x_i\}$
- pro $\forall j > i$, x_j leží na cestě od kořene k x_i $\iff$ x_j má nejvyšší prioritu mezi $\{x_i, x_{i+1}, \dots x_j\}$

• náhodná proměnná $Y_j = \begin{cases} 1 & x_j \text{ leží na cestě od kořene k } x_i \\ 0 & \text{jinak} \end{cases}$

$$\text{hloubka}(x_i) = \sum_{j=1}^{n} Y_j$$

$$\mathbb{E}[\text{hloubka}(x_i)] = \mathbb{E}\Big[\sum_{j=1}^{n} Y_j\Big] = \sum_{j=1}^{n} \mathbb{E}[Y_j]$$

"linearita očekávané hodnoty"

$$\mathbb{E}[Y_j] = 1 \cdot \Pr[X_j \text{ je na cestě}] + 0 \cdot \Pr[X_j \text{ není na cestě}]$$
$$= \Pr[Y_j = 1].$$

$j < i$:

$$\Pr[Y_j = 1] = \Pr[X_j \text{ má nejvyšší prioritu mezi } \{x_j, x_{j+1}, \dots x_i\}]$$
$$= \frac{1}{|\{x_j, x_{j+1}, \dots x_i\}|} = \frac{1}{i-j+1}$$

každý prvek v $\{x_j, x_{j+1}, \dots x_i\}$ má stejnou šanci, že bude mít nejvyšší prioritu.

$i < j$: obdobně

$$\Pr[Y_j = 1] = \frac{1}{|\{x_i, x_{i+1}, \dots x_j\}|} = \frac{1}{j-i+1}$$

$\Longrightarrow$

$$\mathbb{E}[\text{hloubka}(x_i)] = 1 + \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$
$$= \sum_{k=1}^{i} \frac{1}{k} + \sum_{k=1}^{n-i} \frac{1}{k}$$
$$\leq (\ln i) + 1 + (\ln n-i) + 1$$
$$\leq 2 + 2\ln n.$$

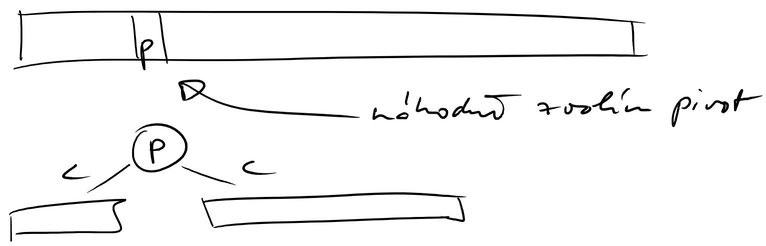
• Očekávaná hloubka vrcholu $\leq 2 + 2\ln n$

$\rightarrow$ $\underbrace{\text{čas}}_{\text{očekávaný}}$ na operaci $O(2 + 2\ln n)$

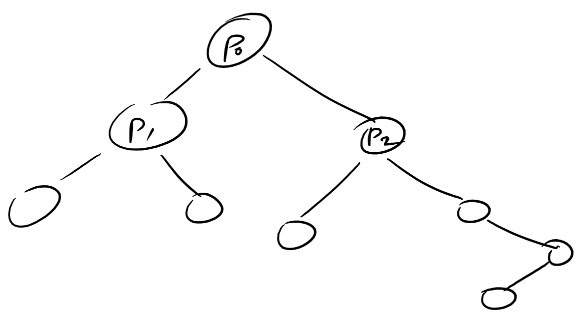
• provede-li $\underbrace{m}_{\text{počet zvolených operací}}$ operací pomocí prázdným stromem, ...

pak očekávaná doba provedení těchto operací je $O(n \cdot \log n)$, kde pravděpodobnost je přes výběr priorit.

Analýza Quicksortu:



strom rozhodování Quicksortu / výběr pivotů je izomorfní s náhodně vybraným stromem treap.



– každý prvek se během Quicksortu porovná se všemi pivoty na cestě ke kořeni

→ počet porovnání = hloubka prvku

→ očekávaná doba Quicksortu =

$$O\left(\mathbb{E}\left(\sum_{i=1}^{n} \text{hloubka}(x_i)\right)\right) =$$

$$O\left(\sum_{i=1}^{n} \mathbb{E}\left(\text{hloubka}(x_i)\right)\right) =$$

$$= O(n \cdot \log n)$$

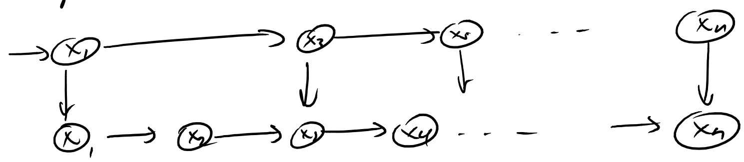
Skip list

uspořádaný seznam $x_1 < x_2 \dots x_n$

$$\to x_1 \to x_2 \to x_3 \to \dots \to x_n$$

doba vyhledání prvku $\leq n$

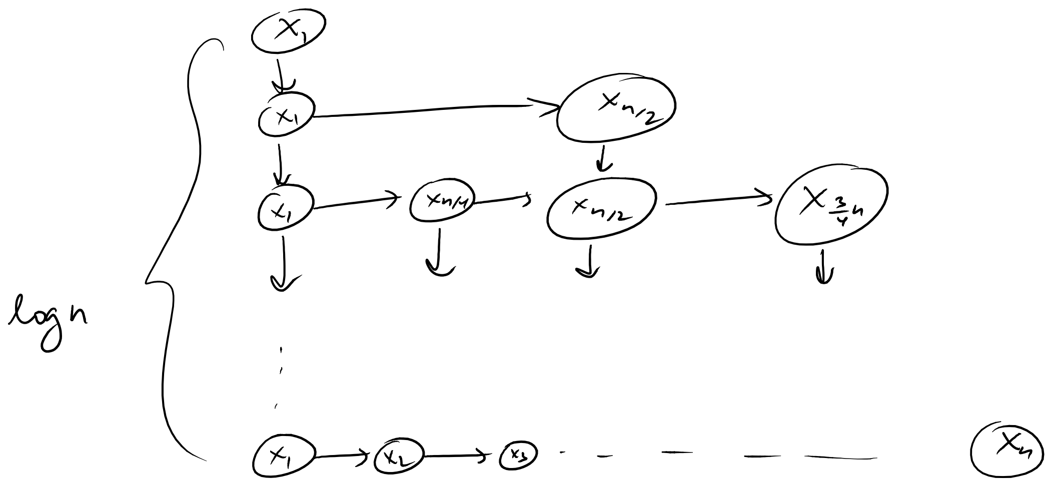
• zlepšení:



pomocí seznam, ve kterém je každý druhý prvek

doba vyhledání prvku $\leq \frac{n}{2} + 1$

• zobecnění



doba vyhledání $O(\log n)$, prostor $O(n)$

• pokud chci dělat dynamicky, není jasné, který prvek bude prostředním atd.

→ náhodně

Insert(x): naleznu místo, kam x patří, a pak ho nakopíruji a zařadím do vyšších hladin tak, že si vždy hodím korunou a s prstí $\frac{1}{2}$ přidám kopii na další hladinu a s prstí 1/2 skončím.

→ $\underbrace{k}_{\text{právě}}$ kopií má pravděpodobnost $2^{-(k+1)}$.

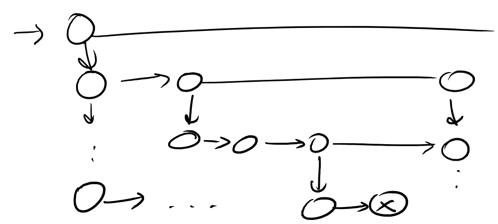
∘ vznikne struktura podobná té deterministické

$$\Pr[x \text{ má} \ge k \text{ kopií}] = \sum_{i \ge k} 2^{-(i+1)} = 2^{-k}$$

$$\Pr[\exists x \text{ s} \ge k \text{ kopiemi}] \le n \cdot 2^{-k}$$

$$\Rightarrow \Pr[\text{skip list má hloubku} \ge 2 \log n] \le n \cdot 2^{-2\log n} = \frac{1}{n}$$

Očekávaná doba vyhledání x : uvažujme cestu od x do kořene



s pst $\frac{1}{2}$ vrchol má kopii o patro výše

$\Rightarrow$ šli jsme nahoru

s pst $\frac{1}{2}$ vrchol nemá další kopii

$\Rightarrow$ šli jsme vlevo

• Jaká je pravděpodobnost p, že během $6 \log n$ kroků neuděláme $\ge 2 \log n$ kroků nahoru?

$$p \le \sum_{j=0}^{2\log n} \binom{6 \log n}{j} \cdot 2^{-6\log n} \le 2 \cdot \log n \binom{6 \log n}{2 \log n} \cdot 2^{-6 \log n}$$

$$1^n = \left(\frac{n-k}{n} + \frac{k}{n}\right)^n = \sum \binom{n}{k} \left(\frac{n-k}{n}\right)^{n-k} \left(\frac{k}{n}\right)^k$$

Binomická věta

$$\Rightarrow \quad 1 \ge \binom{n}{k} \left(\frac{n-k}{n}\right)^{n-k} \left(\frac{k}{n}\right)^k$$

$$\bullet \quad \binom{n}{k} \le \left(\frac{n}{n-k}\right)^{n-k} \left(\frac{n}{k}\right)^k$$

$$p \le 2 \cdot \log n \cdot \binom{6 \log n}{2 \log n} \cdot 2^{-6 \log n} \le 2 \cdot \log n \cdot \left(\frac{6}{4}\right)^{4 \log n} \left(\frac{3}{2}\right)^{2 \log n} \cdot 2^{-6 \log n}$$

$$= 2 \cdot \log n \left(\frac{3}{4}\right)^{6 \log n}$$

$$= 2 \cdot \lg n \left(\frac{3}{4}\right)^{6 \lg n}$$
$$\leq 2 \cdot \lg n \left(\frac{1}{2}\right)^{2 \lg n} = \frac{2 \lg n}{n^2}$$

$\Rightarrow$ pst., že hloubka je větší než $2 \lg n$ & že během $6 \lg n$ kroků nedosáhneme vrchní hladiny $\leq \frac{1}{n} + \frac{2 \lg n}{n^2} \leq \frac{2}{n}$.

$\Rightarrow$ s pstí $\geq 1 - \frac{2}{n}$ operace Insert / Find / Delete trvá $O(\lg n)$

[$\Rightarrow$ očekávaný čas $O(\log n)$.]

---

van Emde Boas trees $\approx$ '76

(vEB strom)

Slovníková struktura pro uložení $S \subseteq U$ s operacemi

- Insert
- Delete
- Find

$\}$ obyčejné operace

$+$ Pred (x) – najdou $\max\{y \in S;\ y \leq x\}$

Succ (x) $\min\{y \in S;\ x \leq y\}$

předchůdce & následník

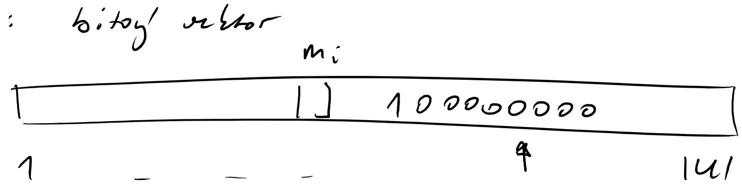
Řešení: binární vyhledávací strom – čas na op. $O(\log n)$, prostor $O(n)$

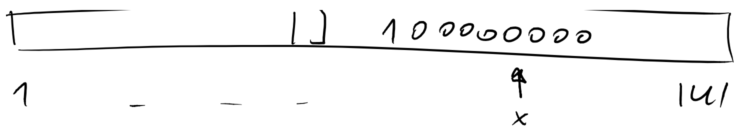
Otázka: lze lépe, pokud $|S| \approx |U|$?

Předpoklad: paměť organizovaná po slovech, každé obsahuje w bitů $w \approx O(\lg |U|)$

Operace nad slovem jako $+, -, *, /, \bmod$ v čase $O(1)$.

Idea: bitový vektor





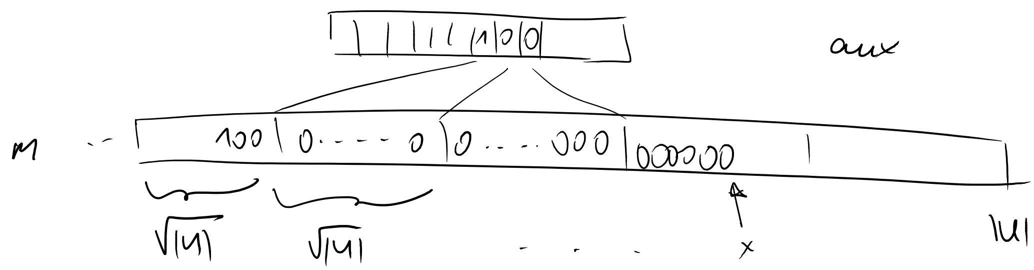
$$m_i = \begin{cases} 1 & i \in S \\ 0 & \text{jinak} \end{cases}$$

Insert, Delete, Find $\quad O(1)$ čas

Succ, Pred $\quad O(u)$ čas

nutno najít nejbližší jedničku, může být až na konci pole.

→ vylepšení ... přidáme bitový vektor <u>aux</u>



vektor M rozdělíme na bloky velikosti $\sqrt{|U|}$

$$aux_j = \begin{cases} 1 & j\text{-tý blok } M \text{ obsahuje jedničku} \\ 0 & \text{jinak} \end{cases}$$

Insert, Find $\quad O(1)$

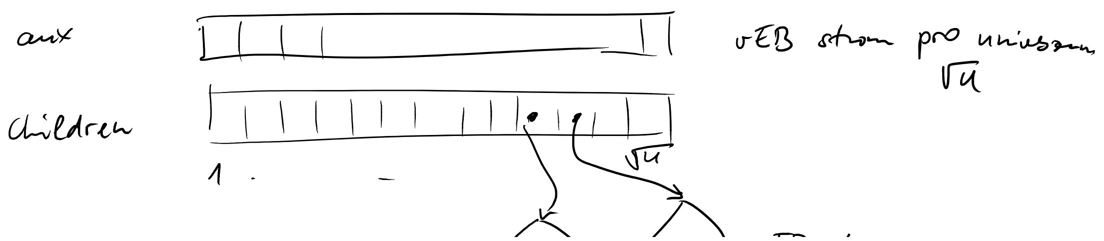
Delete, Succ, Pred $\quad O(\sqrt{u})$

Pred(x) ... prohledá blok x $\quad \left(j = x/\sqrt{u}\right)$

pokud neobsahuje předchůdce x, najdi s pomocí aux nejbližší neprázdný blok, v něm dohledá maximum → předchůdce x.

- <u>aux</u> je struktura řešící stejný problém, ale na univerzu velikosti $\sqrt{u}$.

zobecnění → vEB stromy <u>pro univerzum U</u>:



1 . –

vEB strom pro univerzum $\sqrt{u}$

min, max

- pokud vEB strom obsahuje ≤ 2 prvky, pak jsou to právě prvky min & max. Prvky min & max se již dále neukládají

  → vložení do prázdného stromu – čas $O(1)$

<u>Pozn:</u> Předpokládáme, že vEB strom je na začátku inicializován jako prázdný, před započetím operací s ním.

- aux obsahuje seznam neprázdných stromů children, $\forall j$: aux obsahuje j

  $\iff$ children(j) je neprázdný vEB strom.

<u>Insert(x)</u>:
- pokud je strom prázdný $min = max = x$
- pokud strom obsahuje 1 prvek, $min = \min(min, x)$ $max = \max(max, x)$.
- pokud je $x < min$ nebo $max < x$, prohodí x s min resp. max a pokračuj dále
- $j = x / \sqrt{u}$ $i = x \bmod \sqrt{u}$
- Pokud je children(j) prázdný (poznáš podle $min > max$) pak

  aux.Insert(j)
- children(j).Insert(i).

... čas $O(\log \log |U|)$ ... pokud provádíme aux.Insert(j), pak children(j).Insert(i) trvá $O(1)$.

⇒ vždy nejvýše jedna větev rekurze, která je na vEB strom nad univerzem $\sqrt{u}$.

rekurze, která je na ... strom nad univerzem $\sqrt{U}$.

$$\sqrt{\cdots\sqrt{\sqrt{U}}} = U^{\frac{1}{2^i}} \leq 10 \quad \Rightarrow \quad i = \log\log |U|.$$

pro univerzum velikosti ≤ 10 řešíme triviálně

- $\underline{\text{Find}(x)}$ – pokud $x \in \{\min, \max\} \rightarrow$ jasné
  jinak $\text{children}(x/\sqrt{U}).\text{Find}(x \bmod j)$.
  (pokud je $\text{children}(x/\sqrt{U})$ neprázdný)

... čas $O(\log\log |U|)$ ... rekurze na univerzum velikosti $\sqrt{U}$.

- $\underline{\text{Succ}(x)}$ – $j = x/\sqrt{U}$, $i = x \bmod \sqrt{U}$
  - pokud $i \leq \text{children}(j).\max$
    pak vrať $j * \sqrt{U} + \text{children}(j).\text{Succ}(i)$.

  $j = \text{aux.succ}(j)$;
  vrať $j * \sqrt{U} + \text{children}(j).\min$;

  Pozn: nutno ošetřit triviální případy, kdy daný strom je prázdný, $\text{succ}(x) = \max$, apod.

... čas $O(\log\log |U|)$ ... opět nejvýše jedno rekurzivní volání na univerzum $\sqrt{U}$.

- $\underline{\text{Pred}(x)}$ ... symetricky k $\text{Succ}(x)$.

- $\underline{\text{Delete}(x)}$ – pokud je $x \in \{\min, \max\}$, nahraď ho druhým nejmenším/největším prvkem
  ($\widetilde{\min} \leftarrow \text{Children}(\text{aux.succ}(0)).\min$
  — " — $.\text{delete}(\min)$)

  – $j = x/\sqrt{U}$, $i = x \bmod \sqrt{U}$
  pokud $\text{Children}(j)$ obsahuje poslední prvek
  tj. $i = \min = \max$
  pak odstraň i & $\text{aux.Delete}(j)$.
  jinak $\text{Children}(j).\text{Delete}(i)$.

... čas $O(\log \log |U|)$

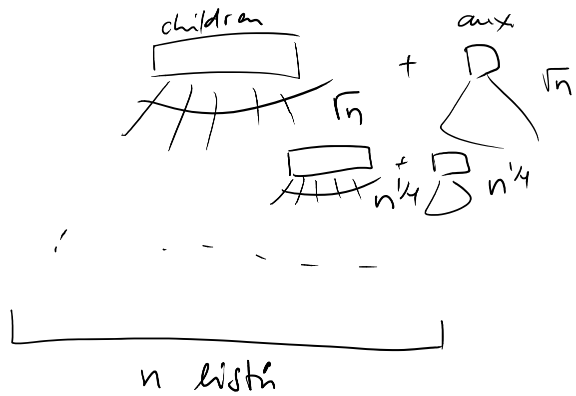
Prostor: $O(U)$ $\qquad |U| = n$

$i=0$

$i=1$

$\vdots$

$i = \log\log n$



na i-té hladině máme $n^{1-\frac{1}{2^i}}$ struktur aux

každá pro velikost $n^{\frac{1}{2^{i+1}}}$

→ celkově tyto struktury aux přidají

$\approx n^{1-\frac{1}{2^{i+1}}}$ svých listů

$$\Rightarrow \sum_{i=0}^{\log\log n} n^{1-\frac{1}{2^{i+1}}} \leq \frac{n}{c}$$

tyto aux ale potřebují svoje aux ... $\frac{n}{c^2}$ listů navíc

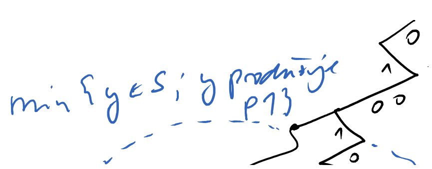
atd.

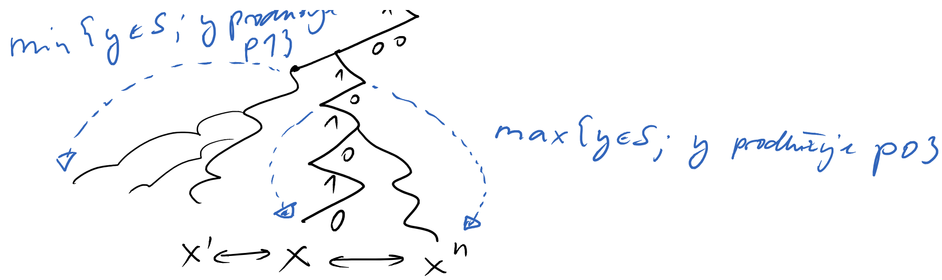
$$\Rightarrow \sum \frac{n}{c^i} \text{ listů } = O(n)$$

- x-fast trie
  - stejný problém jako vEB strom
    - lepší prostor ... $O(n \cdot \log|U|)$ $\qquad n = |S|$, $S \subseteq U$
    - čas o něco horší ... viz níže

idea: pro každý prvek $x \in S$ si uložím všechny prefixy x, je jich $\log|U|$

$x = 0100101010$





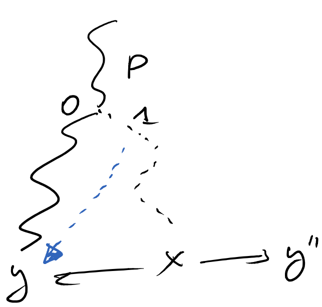
pokud p je jeden z prefixů,

- pokud $p0$ není prefix v S, pak si p pamatuje $\min\{y \in S, y$ má prefix $p1\}$
- pokud $p1$ není prefix v S tak p si pamatuje $\max\{y \in S; y$ má prefix $p0\}$

každý prvek v S si pamatuje nejbližší vyšší a nižší prvek v S.

→ při Find(x) nalezneme nejdelší prefix x, který je v S. Pokud to není x → false
jinak → true

při Succ(x) nalezneme nejdelší prefix p od x, který je v S. Pokud $p0$ je prefix x, pak p ukazuje na následníka x. Pokud $p1$ je prefix x, pak následník x je následník prvku, na který ukazuje p.



podobně pro Pred(x).

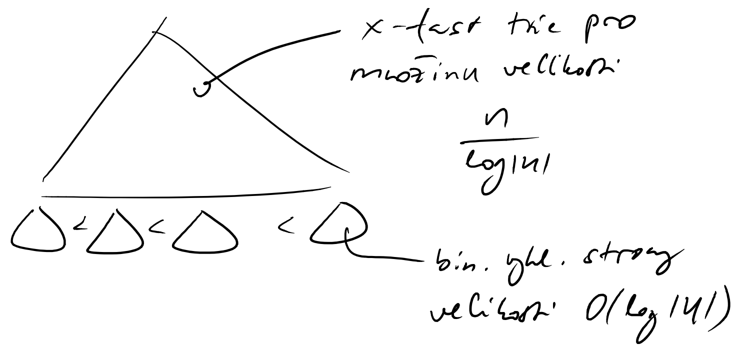
→ prefixy S stejné délky si pamatují v hašovací tabulce pro danou délku prefixu

→ nejdelší prefix x hledám binárním vyhledáváním na jeho délce → čas $O(\log \log |U|)$

Insert (x) ... musím vložit $O(\log |U|)$ prefixů do hašovacích tabulek a upravit odkazy pro prefixy, které tam již jsou
$O(\log |U|)$ práce

Delete (x) ... postupně od nejdelšího odstraňuji prefixy, dokud mohu, zaktualizuji odkazy → čas $O(\log |U|)$.

y-fast trie



- pro každý stromeček uložím do x-fast trie reprezentanta podstromečku (hodnotu mezi min a max sousedních podstromečků)
- při hledání Succ, Pred prozkoumám podstromečky se dvouma nejbližšíma reprezentantama.
- podstromečky udržuji velikosti mezi $\frac{1}{2} \log |U|$ a $2 \log |U|$.

  při Insertu štěpím dle potřeby
  při Deletu slévám dle potřeby

→ stromeček má potřebu se štěpit nejdříve po $\frac{1}{2} \log |U|$ Insertech

→ Stromeček má potřebu se slepit nejdříve

po $\frac{1}{2}\log|U|$ Insertech

a má potřebu se slít nejdříve

po $\frac{1}{4}\log|U|$ Deletech.

(slijí s libovolným sousedem a rozštěpím zpět, pokud mají $> \frac{3}{2}\log|U|$ prvků.)

po slití: rozštěpení vzniklé stromečky velikosti mezi $0.75\log|U|$ a $1.5\log|U|$.

⇒ každých $\frac{1}{4}\log|U|$ operací na podstromečku vyvolá nejvýše jednu operaci na X-fast trie reprezentantů. ≤ 3

(Podstromeček se může přečerpat kvůli svému sousedovi, ale pak to platí soused, a sám zůstane ve stavu, že potřebuje $\geq \frac{1}{4}\log|U|$ operací aby překročil limit.)

→ amortizovaně je čas určen operací na podstromečkách → $O(\log\log|U|)$

Insert, Delete, Succ, Pred — čas $O(\log\log|U|)$ amortizovaně.

Disjoint Set Union, Find:

Problém: chci udržovat komponenty souvislosti grafu, testovat, zda vrcholy jsou ve stejné komponentě a slévat komponenty.

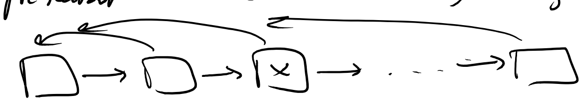
Operace Union(A, B) — slij A a B do jedné množiny, $A \cap B = \emptyset$

Find(x) — vrátí reprezentanta množiny ve které je x.

MakeSet(x) — vytvoř jednoprvkovou reprezentaci $\{x\}$

Možná implementace

1. Spojový seznam
   - každá množina jeden spojový seznam
   - reprezentant – hlava seznamu, každý si na ni ukazuje



Find(x) ... $O(1)$, Union(A,B) ... $O(n)$ čas

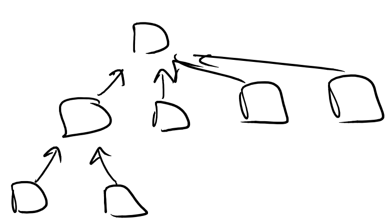
amortizovaně: n MakeSet
m Union, Find, MakeSet $(m \geq n)$
čas $O(n \cdot m)$

2. Vylepšení 1.: Při Union(A,B) připoj kratší seznam za delší (nutno přepočítat ukazatele na repr. prvku v kratším)
+ pamatovat si velikost

čas na Find .. $O(1)$, Union(A,B) ... $O(n)$

amortizovaně $O(m + n \cdot \log n)$
↑
každý prvek si přepíše reprezentanta pouze tehdy, když seznam, ve kterém je, se zdvojnásobí!
$\Rightarrow$ max $\log_2 n$ - krát

3. stromy

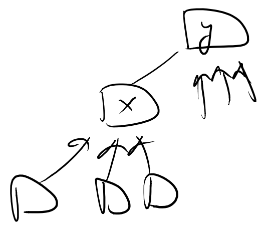


→ rank stromu

MakeSet(x) = x.rank = 0;
x.parent = x;

Find(x) =
x.parent = Find(x.parent);
return x.parent;

Union(x, y)
x = Find(x);
y = Find(y);
[illegible]



```
x = Find(x);
y = Find(y);
if x.rank < y.rank then
    x.parent = y,
else
    y.parent = x;
    if x.rank = y.rank then
        x.rank ++;
end if;
```

→ Find zkrátí prošlou cestu

časová složitost: n operací MakeSet z celkového

počtu m operací

amortizovaný čas $O(m \cdot \alpha(n))$

kde $\alpha(n)$ je inverzní fce. k

Ackermanově fci

pro $f: \mathbb{N} \to \mathbb{N}$ (nerostoucí) definujeme

$$f^*(n) = \min\{i, \underbrace{f f f \cdots f}_{i\text{-krát}}(n) \leq 1\}$$

$$\lambda_1 = \lceil \log_2 n \rceil$$

$$\lambda_{i+1}(n) = \lambda_i^*(n)$$

pak $\alpha(n) = \lambda_n(n)$

Ukážeme slabší výsledek: čas $O(m \cdot \log^* n)$

Pozorování:
- Rank vrcholu se zvýší pouze tehdy, pokud k němu připojím vrchol stejného ranku
- Kořen ranku r má $\geq 2^r$ vrcholů. (indukcí)
- Rank r má nejvýše $n/2^r$ jiných vrcholů. (*)
- Rank ostře roste podél každé cesty ke kořeni
- jakmile vrchol přestane být kořen, jeho rank se nemění.

potřebujeme analyzovat Find(x)

- analyzujeme pouze změny ukazatelů

na předka u vrcholů v hloubce 2 a více
(ostatní práce $O(m)$)

- takový vrchol už nemají pevný daný rank

rozdělme všechny vrcholy do skupin A_i

$$\ell_0 = 1 \qquad \ell_i = 2^{\ell_i}$$

$$A_i = \{ v ;\ v \text{ má rank } [\ell_{i-1}, \ell_i) \}$$

- $|A_i| \le \dfrac{2n}{\ell_i}$ (jednoduché z (*))

počet průchodů přes hranu
$u \to v$ kde $u, v \in A_i$
a $\underline{u}$ je pevně zafixovaný
je během všech operací Find
nejvýše ℓ_i, protože
přitom má u jiného předka s vyšším rankem
při každém dalším Find

→ čas strávený na těchto průchodech je

$$\sum_{i=1}^{\log^* n} \frac{2n}{\ell_i} \cdot \ell_i = O(n \log^* n)$$

počet průchodů přes hranu $u \to v$
kde $u \in A_i$ a $v \in A_j$, $i < j$
je nejvýše $\log^* n$ pro každou
operaci Find
→ celkem $O(m \log^* n)$

⟹ celkový čas $O(m \log^* n)$.

Operace s řetězci

abeceda .. Σ

$S \subseteq \Sigma^*$ $\quad m = \sum_{x \in S} |x| \quad n = |S|$

např.: $\Sigma = \{a, b, \dots, z\}$

$\Sigma = \{0, 1, 2, \dots 9\}$

$\Sigma = \{0, 1\}$

$\Sigma = \{A, G, T, C\}$

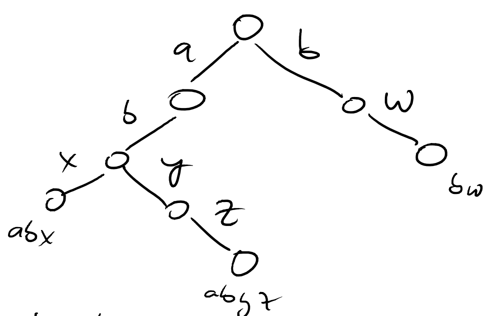
$\Sigma = \{0, 1, \dots, 255\}$

slovníkový problém pro S

- Insert (x)
- Delete (x)
- Member (x) $\quad$ (= Find (x))

řešení – Trie

$S = \{abx, abyz, bw\}$



Σ-ární strom, každá hrana ohodnocena symbolem z Σ

hrany vycházející z daného vrcholu ohodnoceny různě

s každým vrcholem asociuji řetězec $\alpha \in \Sigma^*$

získaný při průchodu od kořene k vrcholu

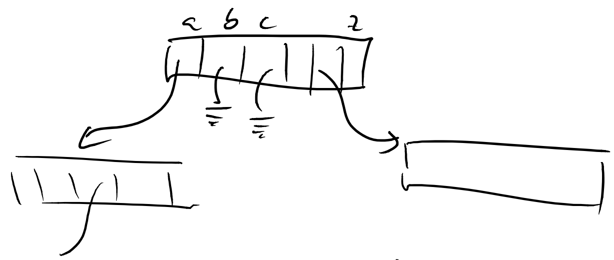
Member(x) – procházej strom od kořene a vyber vždy hranu ohodnocenou dalším znakem x. Pokud taková hrana neexistuje, $x \notin S$. Dosažený vrchol si pamatuje, zda přiřazené slovo je v S či není.

Insert(x) – procházím podobně jako u Member(x), jakmile dosáhnu chybějící hranu, hranu vytvořím, přidám nový list, přejdu do něj a pokračuji. Vrchol příslušející x označím, že je v S

Delete(x) – najdu vrchol příslušející x, odznačím, že je v S, pokud je to list, přejdu na jeho otce, list smažu, a opakuji dokud nedojdu na vrchol označený jako v S nebo vrchol, který má dalšího potomka.

Implementace:

1) každý vrchol má pole indexované Σ s ukazateli na své syny



– $O(|\Sigma|)$ prostoru na vrchol, $O(|\Sigma| \cdot n)$ celkově

Member(x) ... $O(|x|)$

Insert(x) ... $O(|\Sigma| \cdot |x|)$

Delete(x) ... –"–

2) každý vrchol má spojový seznam svých potomků

– celkový prostor $O(n)$

Member(x) ... $O(|\Sigma| \cdot |x|)$

Insert(x) ... $O(|\Sigma| \cdot |x|)$

Delete(x) ... –"–

lze trochu zrychlit, pokud je seznam potomků setříděn dle abecedy (poloviční čas)

3) binární vyhledávací strom potomků

operace $O(\log|\Sigma| \cdot |x|)$

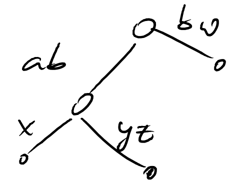
prostor $O(n)$

4) naivní tabulka potomků

Operace $O(|x|)$

prostor $O(n)$

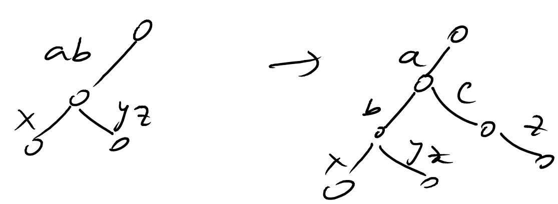
Komprimovaná trie



- hrany ohodnoceny slovy $\in \Sigma^*$
  hrany vycházející z jednoho vrcholu se liší v prvním znaku

  → při member(x) stačí porovnávat první znak, pokud si u vrcholů pamatuji délku označení hrany
  na konci však nutno ověřit, že jsem našel to, co jsem hledal

- při vkládání občas nutno rozdělit hranu   Insert(acz)



→ o něco kompaktnější reprezentace

Problémy s řetězci

$T \in \Sigma^m$ ... text

$P \in \Sigma^n$ ... hledané slovo

<u>úkol</u>: najdi výskyt / všechny výskyty P v T

<u>Klasika</u> Aho-Corasick – sestrojím konečný automat z P a spustím ho na T

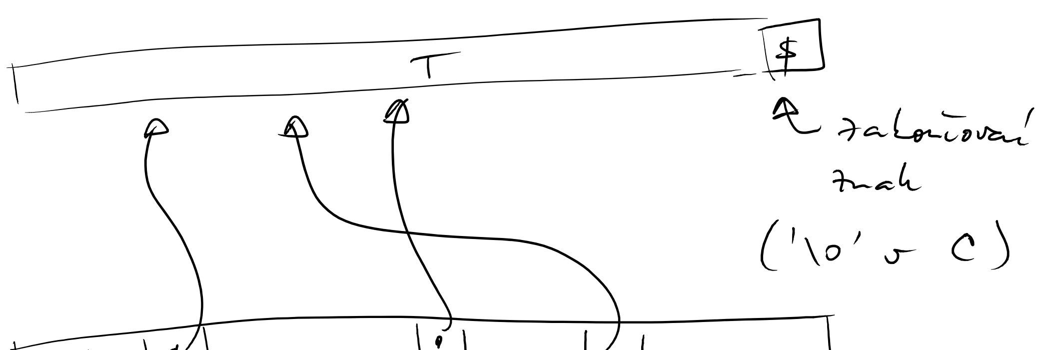
čas $\approx O(m+n)$.

<u>chceme lépe</u> – typicky $m \gg n$ a T je databáze, která se nemění

<u>řešení</u>: sufixové pole, sufixový strom

Sufixové pole

'\$' $<_{lex} \Sigma$



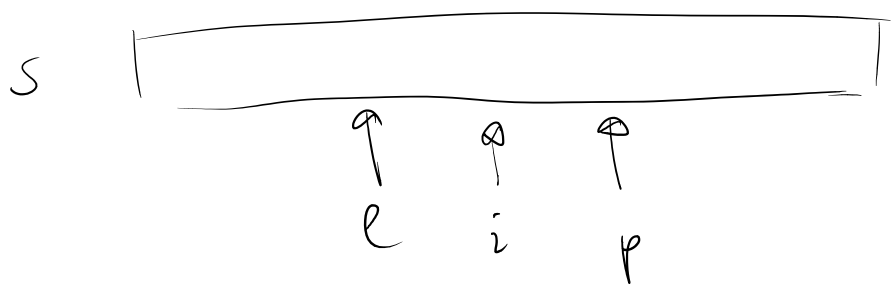
lexikograficky setříděné pole všech sufixů T,
tj. pole řetězců $T[1..m], T[2..m], \dots T[m..m]$

→ v poli lze binárně vyhledávat řetězec P

– hledám lexikograficky nejmenší řetězec větší-rovno P.

· pokud se P shoduje s tímto řetězcem na prvních $|P|$ pozicích, nalezl jsem výskyt P v T, bezprostředně následující řetězce též se shodující s P jsou další výskyty P.

binární vyhledávání – čas $O(n \cdot \log m)$



$l \leftarrow 1$, $p \leftarrow m$ // hledá s přesností ±1
dokud je $p > l+1$
  $i \leftarrow \left\lfloor \frac{l+p}{2} \right\rfloor$ (*)
  pokud $S[i] <_{lex} P$ pak $l \leftarrow i$
  jinak $p \leftarrow i$.
  return l
end.

... log m iterací, každé porovnání (*) stojí $O(m)$.

- lze zlepšit na $O(n + \log m)$

$u, v \in \Sigma^*$ $lcp(u,v)$ = délka nejdelšího společného prefixu

předpokládejme, že známe zadarmo

$lcp(S(l), S(i))$ a $lcp(S(p), S(i))$

pro všechny trojice l, p, i (které

se mohou vyskytnout během binárního

vyhledávání)

↳ těch je pouze $O(m)$, viz níže

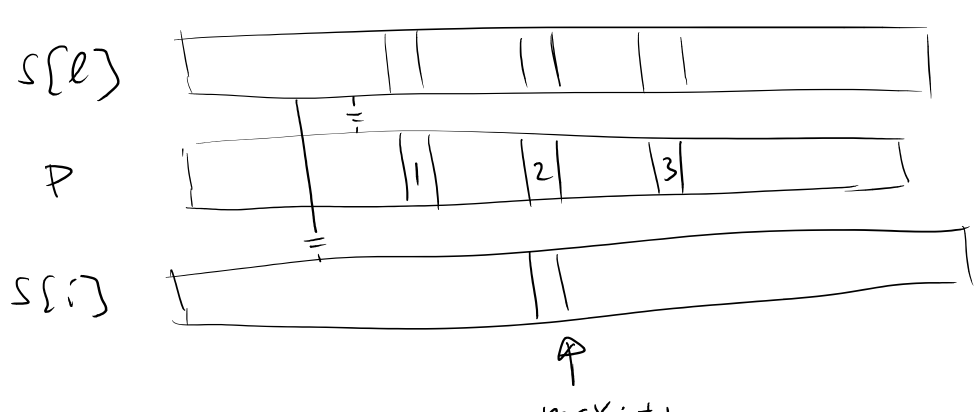
- během binárního vyhledávání si udržuji:

$max_l = lcp(S[l], P)$

$max_p = lcp(S[p], P)$

pokud $max_l > max_p$ porovnání (*) provedu následovně:

$max_i = lcp(S[l], S(i))$



tři možnosti:

3) $max_l > max_i \Rightarrow P <_{lex} S[i]$

neboť $S[\ell]$ a P se shodují
až do (3) a $S[i]$ se odliší v $max_i + 1$

$\Rightarrow$ $p \leftarrow i$, $max_p \leftarrow max_i$

2) $max_\ell = max_i$: najdi první rozdílný znak
mezi $S[i]$ a P za pozicí $max_i \to r$
pokud $S[i][r] < P[r]$

pak $\ell \leftarrow i$, $max_\ell \leftarrow r-1$

jinak $p \leftarrow i$, $max_p \leftarrow r-1$

1) $max_\ell < max_i$: $\ell \leftarrow i$, $max_\ell \leftarrow max_i$

pokud $max_\ell \leq max_p$ postupuj symetricky
jako při $max_\ell > max_p$.

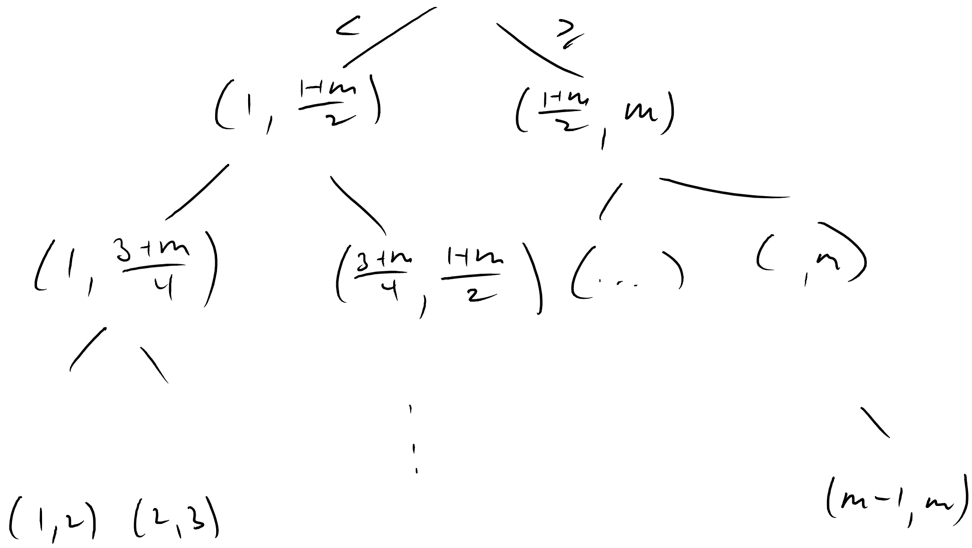
Pozn: pokud zjistím, že P je prefix $S[i]$,
vyhledávání zastavím a našel jsem
výskyt. Během binárního vyhledávání
tak můžu předpokládat, že P není
prefixem ani $S[\ell]$ ani $S[p]$.

Čas na (*) je pak v součtu $O(n + \log m)$,
neboť ℓ a p se posunují pouze doprava

- předpoklad, že známe $lcp(S[\ell], S[i])$ a $lcp(S[p], S[i])$:
  - ty si předpočítáme, je jich $O(m)$, takže
    je lze skladovat v rozumné paměti

ℓ p
$\downarrow$ $\downarrow$
$(1, m)$

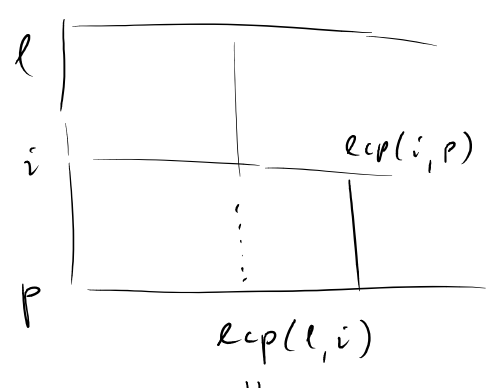
$<$ $\geq$



hranice l, p, i se posouvají jako při průchodu jednou z cest. Strom má $\leq m$ listů a $\leq m$ vnitřních uzlů $\Rightarrow$ $\leq 2m$ kombinací l, p, i.

potřebné hodnoty lcp lze spočítat odspodu nahoru, pokud známe $lcp(i, i+1)$ $\forall i$. Platí totiž

$$lcp(l,p) = \min_{i \in [l,p)} lcp(i, i+1)$$



$$\Downarrow$$

$$lcp(l,p) = \min(lcp(l,i), lcp(i,p))$$

- $lcp(i, i+1)$ $\forall i = 1 .. m-1$ lze spočítat v čase $O(m \log m)$

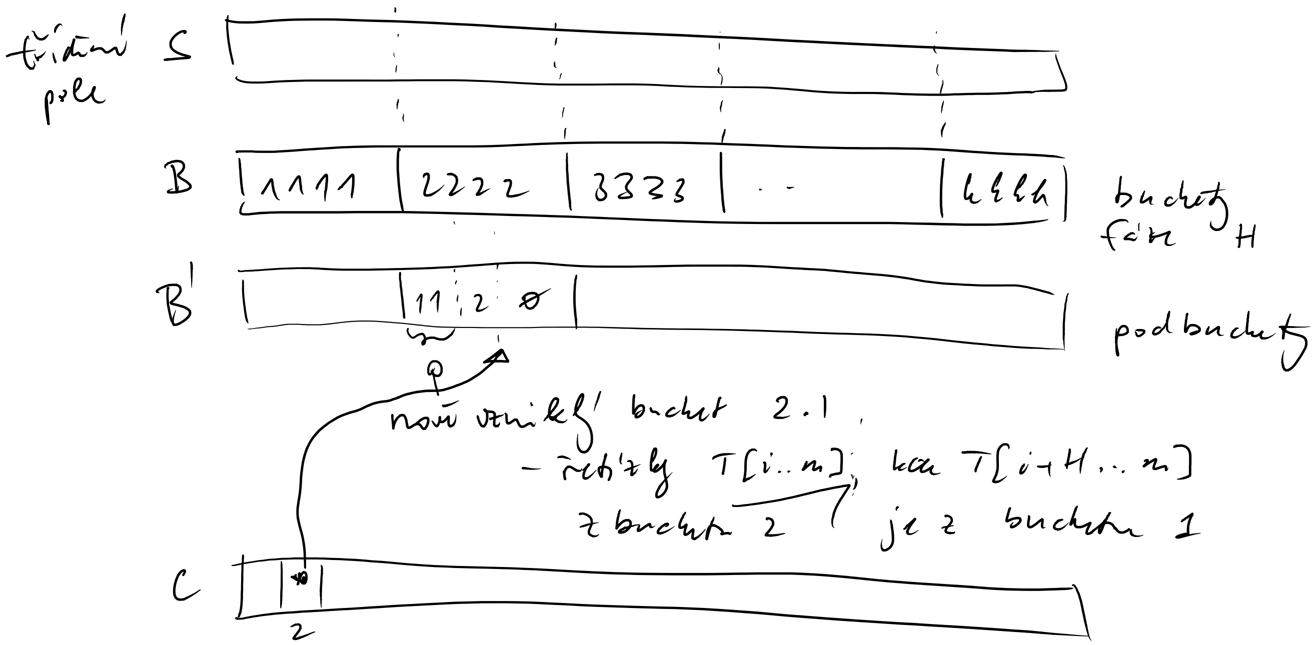
- sufixové pole lze zkonstruovat v čase $O(m \cdot \log m)$:
  - varianta bucket-sortu [Karp-Miller-Rosenberg '72]
  - $\log m$ fází
    - v nulté fázi rozdělím řetězce do bucketů podle prvního symbolu.
    - ve fázi H, přerozdělím řetězce do bucketů podle prvních $2H$ symbolů.
      - → na začátku fáze se řetězce v každém bucketu shodují v prvních H znacích, na konci v prvních $2H$ symbolech (buckety se podrozdělí)
      - → fáze trvá $O(m)$ čas.

↓

využívám, že již znám roztřídění řetězců podle H-symbolů. Pro porovnání $T[i,m]$ a $T[j,m]$ se stejnými prvními H symboly lze použít informaci o bucketech $T[i+H,m]$ a $T[j+H,m]$.

- skutečná implementace bere buckety v lex-pořadí, v rámci bucketu jde řetězcích $T[i..m]$ po řetězcích a přesune řetězec $T[i-H...m]$

(když $i-H>0$) v rámci jeho bucketu na začátek
do nově odděleného bucketu
[vše je uloženo v poli.]



ukazatel na ještě nezpracovaný záznam v daném bucketu, kde vznikl nový podbucket

– na konci fáze přečíslují vzniklé buckety od 1, 2, ...

Sufixový strom: trie sestavená z T[1..n], T[2..n], .. T[n,n]

– koncepčně přímočařejší než sufixové pole, ale sufixové pole je kompaktnější, z hlediska času obě řešení jsou srovnatelná.

- mnoho různých algoritmů na konstrukci suf. polí & stromů.
- komprimované trie, kde se místo řetězců na hranách pouze odkazují na podřetězce v T potřebují pouze $O(n)$ místa.

Užití:

1) Hledání P v T
2) zobecnění: Hledáme P v $T_1, T_2, \dots T_k$
3) nejdelší společný podřetězec (podinterval) T_1 a T_2 … $O(|T_1| + |T_2|)$
(ignorujeme "log"y)

a mnoho dalších

Samoupravující se seznamy

- množina prvků $x_1, \dots x_n$
- chci reprezentovat spojovým seznamem
- pstí $p_1, p_2, \dots p_n$, kde p_i je pravděpodobnost, že Find bude hledat x_i.

Otázka: Optimální uspořádání prvků v seznamu?
→ podle klesající pravděpodobnosti p_i

Bůno: $p_1 \geq p_2 \geq p_3 \dots \geq p_n$

očekávaný čas operace Find (x_i)

$$E[T] = \sum_{i=1}^{n} \underline{i \cdot p_i}$$

s pstí p_i hledám x_i a musím tedy přelézt i prvků.

Problém: většinou neznám $p_1, p_2, \dots p_n$ dopředu

MFR (Move-to-Front) strategie – po nalezení prvku x, přesuneme x na začátek seznamu.

T^k_{MFR} ... čas při k-tém Find při MFR strategii

Tvrzení: $\lim_{k\to\infty} E[T^k_{MFR}] \le 2 \cdot E[T]$

Dk: Nechť $a_1, a_2, \dots a_k$ je náhodná posloupnost prvků, na které se provádí Find

Platí:

$$E[T^k_{MFR}] = \sum_{i=1}^{n} Pr[a_k = i] \cdot E[\text{počet prvků před } x_i \text{ v čase } k]$$

$\forall i \quad Pr[a_k = i] = p_i$

Vezmeme pevné i.

$$E[\text{počet prvků před } x_i \text{ v čase } k] \le \sum_{\substack{j=1 \\ j \ne i}}^{n} 1 \cdot Pr[x_j \text{ je před } x_i \text{ v čase } k]$$

$$Pr[x_j \text{ je před } x_i \text{ v čase } k] \le$$

$$Pr[a_1 a_2 a_3 \dots a_{k-1} \text{ neobsahuje ani } x_i \text{ ani } x_j]$$

$$+ Pr[\text{poslední výskyt } x_i \text{ nebo } x_j \text{ v } a_1 \dots a_{k-1} \text{ je } x_j \mid x_i \text{ nebo } x_j \text{ se vyskytlo}]$$

je x_j | x_i nebo x_j se vyskytlo }

$$= (1 - p_i - p_j)^{k-1} + \frac{p_j}{p_i + p_j}$$

- uvažujme podposloupnost $a_1 a_2 \dots a_k$ sestávající pouze z x_i a x_j. Pravděpodobnost, že poslední v této podposloupnosti je x_i je $\frac{p_i}{p_i+p_j}$ a že je x_j je $\frac{p_j}{p_i+p_j}$

$$\Rightarrow E[T^a_{MFR}] = \sum_i p_i \cdot \sum_{j \neq i} \frac{p_j}{p_i + p_j} + (1 - p_i - p_j)^{k-1}$$

$$= \sum_{i \neq j} \frac{p_i p_j}{p_i + p_j} + \underbrace{\sum_{i \neq j} (1 - p_i - p_j)^{k-1}}$$

$$\leq 2 \sum_i p_i \underbrace{\sum_{j<i} \frac{p_j}{p_i + p_j}}_{\leq} + \quad -\text{"}-$$

$$\leq \underbrace{2 \sum_i p_i (i-1)}_{\leq E[T]} + \underbrace{\sum_{i \neq j} (1 - p_i - p_j)^{k-1}}_{\xrightarrow[k \to \infty]{} 0}$$

Pozn: Očekávaný čas MFR strategie je tedy nejvýše 2-krát delší než statický optimální (pro k velké)

- Dá se ukázat, že amortizovaný čas m operací

Find je nejvýše $O(OPT + n^2)$, kde
OPT je čas nejlepší možné strategie udržování
seznamu pro danou posloupnost operací Find.

Dynamizace datových struktur

- Statická dat. struktura
  - struktura, která nepodporuje modifikace (jako Insert/Delete), pouze zodpovídá dotazy

$P(n)$ ... čas na vytvoření struktury s $\underline{n}$ prvky ("preprocessing")

$Q(n)$ ... čas na dotaz na strukturu o $\underline{n}$ prvcích

Chceme udělat strukturu podporující

1) Insert ... "semidynamizace"
2) Insert & Delete "dynamizace"

Př: • range-trees dimenze d bez kaskádování

$$P(n) = O(n \cdot \log^d n)$$
$$Q(n) = O(\log^d n)$$

Insert? Delete?

→ generická metoda semidynamizace a dynamizace, kde čas na

Insert $O\left(\frac{P(n)}{n} \cdot \log n\right)$

Delete $O\left(\frac{P(n)}{n} + D(n) + \log n\right)$

Query $O(Q(n) \cdot \log n)$

Query $\quad O(Q(n) \cdot \log n)$

amortizovaně / v nejhorším případě

čas na "falešný Delete" ... označení prvku jako smazaného

<u>Idea</u>: (podobné binomiálním haldám)

množina A je reprezentovaná pomocí množin $A_0, A_1, \dots A_k$ $\quad (k = \lceil \log n \rceil)$

kde $A = \bigcup A_i$ $\qquad A_i \cap A_j \neq 0$ pro $i \neq j$

a $|A_i| = 2^i$ nebo $A_i = \emptyset$

a pro každou množinu A_i máme vytvořenu příslušnou dat. str. $S(A_i)$

• <u>Query (x)</u> – spustím Query (x) na $S(A_i)$ pro $i = 0, \dots, k$ a agreguji výsledek

→ lze použít pouze pro dotazy, kde lze výsledek nějakým způsobem agregovat.

<u>Např.</u>: $\text{Find}(x, A) = \bigvee_{i=1}^{k} \text{Find}(x, A_i)$

$$\text{Size}(A) = \sum_{i=1}^{k} \text{Size}(A_i)$$

apod.

- <u>Insert (x)</u> - vytvoř $A_0' = \{x\}$ a $S(\{x\})$.
  Dokud existují dvě množiny A_i a A_i' stejné velikosti slouč je dohromady $A_{i+1}' \leftarrow A_i \cup A_i'$ a vytvoř $S(A_{i+1}')$, zruš $S(A_i)$ & $S(A_i')$.

každá množina A_i je reprezentovaná spojovým seznamem prvků, tedy slučování lze provést v konstantním čase.

- předpoklady (rozumné) : 1) $\frac{P(n)}{n}$ je neklesající

  2) $\frac{S(n)}{n}$ — // —

  $S(n)$ je prostor potřebný pro uložení $S(A)$, $|A| = n$

  3) čas na zrušení $S(A)$ je menší než na její vytvoření.

Amortizovaný čas

n ... Insertů → nejvýše $\frac{n}{2^i}$ vytvoření $S(A_i)$ pro $|A_i| = 2^i$.

celkový čas $\leq \sum_{i=0}^{k} \frac{n}{2^i} S(2^i) \leq \sum_{i=0}^{k} n \cdot \frac{S(n)}{n}$ (P1)

$= S(n) \cdot \log n$

→ $O\left(\frac{S(n) \log n}{n}\right)$ za Insert

- Stejného času lze dosáhnout v nejhorším případě.

Idea: A_i a A_i' neslučuji během jednoho insertu, ale rozložím práci na budování $S(A_i \cup A_i')$ do 2^{i-1} částí, každá z nich trvající $\frac{P(2^{i-1})}{2^{i-1}}$ času. Při následujících insertech udělám vždy jednu část z každého probíhajícího slučování (těch je maximálně k)

- Dokud není $S(A_i \cup A_i')$ hotovo, používám na dotazy $S(A_i)$ a $S(A_i')$, jakmile je hotovo, $S(A_i)$ a $S(A_i')$ zahodím.

- Oproti zbrklé (amortizované) variantě má vytvoření struktury $S(A_i)$ pro $|A_i| = 2^i$ zpoždění 2^{i-1} insertů (indukcí podle i)

$\rightarrow$ $S(A_i' \cup A_i)$ bude hotovo dříve než mi přibude další množina velikosti 2^i.

Dynamizace: (Insert + Delete)

Podobně jako semi-dynamizaci, budu si udržovat trojice (A_i, D_i, F_i) $\quad i = 0, \dots, k$

kde $|A_i| + |D_i| + |F_i| = 2^i$

$2^{i-1} < |A_i| \leq 2^i$

$\bigcup A_i = A$ ... aktuální množina prvků v dat. str.

(všechny množiny jsou disjunktní) $|A| \leq n$ v každém okamžiku

Pravidla údržby

1) pokud mám (A_i, F_i, D_i) a (A_i', D_i', F_i') stejné úrovně, tj. $|A_i| + |F_i| + |D_i| = |A_i'| + |D_i'| + |F_i'|$ pak z nich vytvořím o úroveň vyšší

$(A_i \cup A_i', \emptyset, D_i \cup D_i' \cup F_i \cup F_i')$

a $S(A_i \cup A_i')$

2) pokud mám (A_i, F_i, D_i) na úrovni i, kde $|A_i| = |F_i| + |D_i| = 2^{i-1}$, pak zruším (A_i, F_i, D_i) a vytvořím

$(A_i, \emptyset, \emptyset)$ a $S(A_i)$

o úroveň níže

- A_i si udržuji v binárním vyhledávacím stromu

(- D_i a F_i si není třeba pamatovat, pro analýzu používejme jejich velikosti.)

- při Insert(x) vytvořím $(\{x\}, \emptyset, \emptyset)$ a aplikuji pravidla údržby

- při Delete(X), hledá $x \in A_i$ z (A_i, D_i, F_i) ve struktuře $S(A_i \cup D_i)$ příslušející k (A_i, D_i, F_i) označím $\underline{x}$ jako smazaný a přesunu $\underline{x}$ z A_i do D_i

(tj. $A_i \leftarrow A_i \setminus \{x\}$, $D_i \leftarrow D_i \cup \{x\}$)

a aplikuji pravidla údržby.

$\Rightarrow$ struktura S odpovídající (A_i, D_i, F_i) byla původně vytvořena pro $A_i \cup D_i$ a od té doby prvky D_i v ní byly označeny jako smazané.

Amortizovaná analýza

- každý prvek x při složení zaplatí dopředu za sloučení na k úrovních, na každé

$$\frac{P(2^i)}{2^i} \leq \frac{P(n)}{n} \rightarrow \frac{P(n)}{n} \log \text{ celkem}$$

to odpovídá času, který může v budoucnu být spotřebován při tvorbě $S(A_i)$, $x \in A_i$, a lze přisoudit tomuto prvku.

- V důsledku deletů, ale může prvek x vystupovat ve více tvorbách $S(A_i)$ než jen k. tj musíme účtovat deletům.

$\rightarrow$ prvek $\underline{x}$ si předplatí tvorbu $S(A_i)$ na k úrovních při svém Insertu

- v průběhu času musí vždy platit, že x má předplaceny všechny úrovně $j > i$, když x je v danou chvíli v A_i.

Pravidlo údržby 1) pak svůj čas vezme z času předplaceného prvky v A_i a A_i'

Pravidlo údržby 2) se vyvolá pouze v důsledku 2^{i-1} delete z množiny A_i, v okamžiku vyvolání tohoto pravidla mají prvky v A_i předplacený všechny úrovně $>i$, zároveň se ale přesunou o úroveň níže, čili potřebují, aby za ně někdo předplatil i-tou úroveň. To musí nařídit prvky z $D_i \cup F_i$, množiny D_i a F_i se v tento okamžik vyprázdní $\Rightarrow$ účtujeme jim předplacení jenom jednou.

Jelikož $|D_i \cup F_i| = |A_i|$, každý z $D_i \cup F_i$ musí přispět pouze $\frac{P(2^i)}{2^i} \leq \frac{P(n)}{n}$.

$\Rightarrow$ amortizovaný čas na Insert $\left(\frac{P(n)}{n} \cdot \log n\right)$

Delete $\left(\frac{P(n)}{n} + D(n) + \log n\right)$

předplacení

označení x v příslušné struktuře $S(A_i)$

údržba

A_i bin. vyhl. stromu

- Alternativně to odpovídá analýze s potenciálem

$$\Phi = \frac{P(n)}{n} \cdot \sum_{i=0}^{k} (k-i)\,|A_i| \; + |F_i| + |D_i|$$

Jelikož každý vložený prvek se smaže jenom jednou, lze počítat $\frac{P(n)}{n}$ z času delete do Insertu

Insert $\quad O\left(\frac{P(n)}{n} \cdot \log n\right)$

Delete $\quad O(D(n) + \log n)$.

- Pro implementaci lze použít následující pravidla:

$\mathrm{rank}(A_i) = \lceil \log |A_i| \rceil$ tj. $2^{\mathrm{rank}(A_i)-1} < |A_i| \le 2^{\mathrm{rank}(A_i)}$

1) pokud dvě množiny A_i a A_j mají stejný rank, vytvoř pro ně $S(A_i \cup A_j)$ a staré je doluvnaď

2) pokud při delete klesne rank A_i, postav novou $S(A_i)$ a starou zruš.

Tato pravidla odpovídají výše uvedenému popisu.

- Strukturu lze též udělat tak, aby se dosahovalo uvedených časů i v nejhorším případě.