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

Persistentní datové struktury

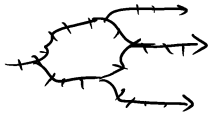
- struktury, které dovolují přístup a modifikaci nejen dřívější verzi

- použití: - textový editor – 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 → nová verze

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



- funkcionální persistence
  - dřívější verze jsou nedotknutelné (RO), ale můžu s libovolnou verzí zacházet jak chci, pokud ji nemutuju. (musím ji zachovat do nové verze)

triviální řešení:

1) před každou operací celou strukturu překopíruju a na nové kopii provedu změny → nová verze

problém: čas & prostor

2) uchovávám si historii všech provedených operací nad dat. str., chci-li přistoupit k nějaké verzi, "přetočím" 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
↑
míc mac

částečná perzistence

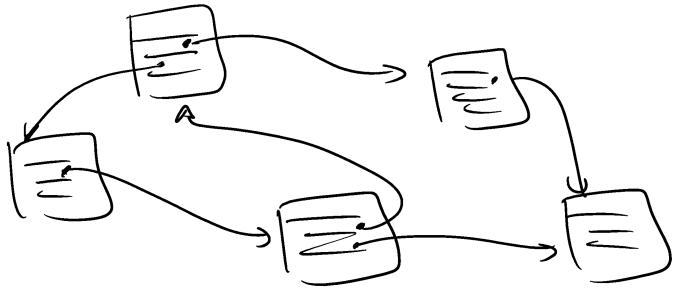
- pole - každý prvek si pamatuje všechny svoje změny → záznam změny: (verze, nová hodnota)
→ bin. vyhledávací strom změn uspořádaný dle čísla verze

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

prostor: úměrný počtu jednotlivých elementárních změn (přiřazení)

lze lépe:

- datové struktury sestávající ze záznamů obsahujících proměnný počet záznamů jednoho typu



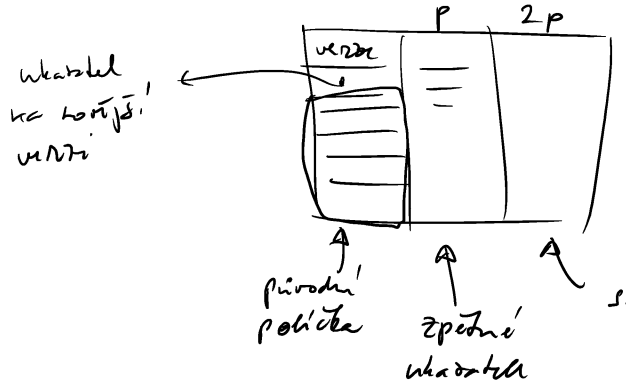
např.: spojové seznamy, vyhledávací stromy, ...

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

Tvrzení: 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 políčka v záznamu.

Dk: implementace: rozšířený záznam



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

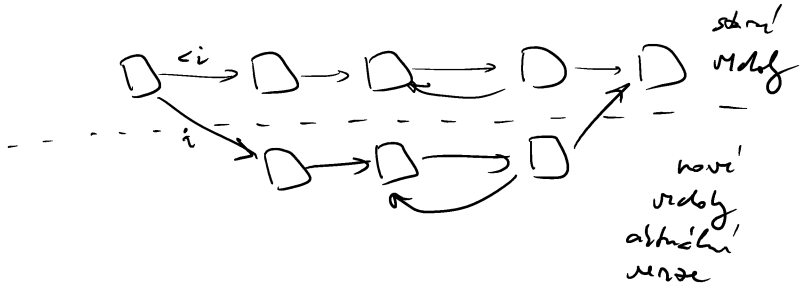
→ pole indexované verzí, které udává odkaz na kořen dané verze.

· čtení políčka v kontextu verze i

· v aktuálním záznamu projdu seznam změn a vrátím nejnovější hodnotu políčka verze $\leq i$.

- 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)

→ vzniknou tak množiny nových vrcholů, které ukazují na sebe a případně na starší verze



Analýza:

$\Phi = c \cdot \sum$ # záznamů o změnách v nejnovější verzi
záznamů

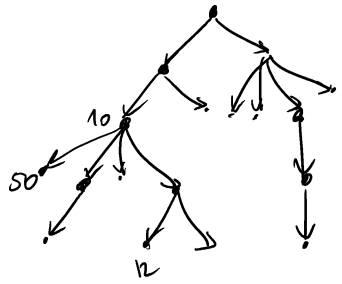
amortizovaný čas $\leq$ čas skutečný $+$ změna Φ = přidání záznamu o provedení změnách

$$lc + (-2cpl + cpl) \leq 2c$$

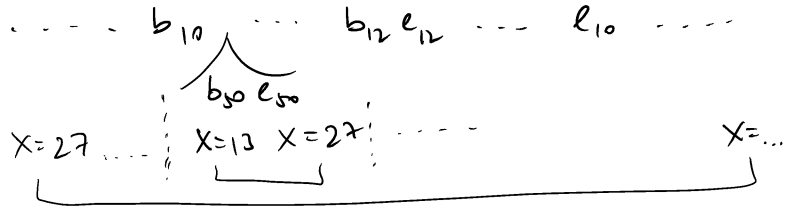
- lc — tvorba nových vrcholů (l ... # nových vrcholů)
- $-2cpl$ — změna Φ při tvorbě nových vrcholů
- cpl — tvorba nových vrcholů nelze vytvořit pl nových záznamů o vrcholech

Úplná perzistence

Strom verzí



linearizace – průchod do hloubky, při: první navštívení vrcholu i vypiš b_i (levá závorka), při: posledním opuštění i vypiš e_i (= pravá závorka)



- při vytvoření nové verze i z verze j, vlož $b_i e_i$ bezprostředně za b_j (př. přidání v. 50 pod v. 10)
- každá elementární změna položky záznamu odpovídá nová verze, pokud např. nastavíme $X = 13$, pak verze b_{50} má za úkol nastavit $X = 15$ a verze e_{50} má za úkol nastavit X na původní hodnotu, jakou ji 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 odpovídajících v. 50, pokud nemění hodnotu X)

  ⇒ pokud chci znát hodnotu X ve verzi i stačí najít nejbližší verzi $\leq b_i$, s lineárním uspořádáním, která nastavuje tuto hodnotu a to je hledaná 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

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

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

---

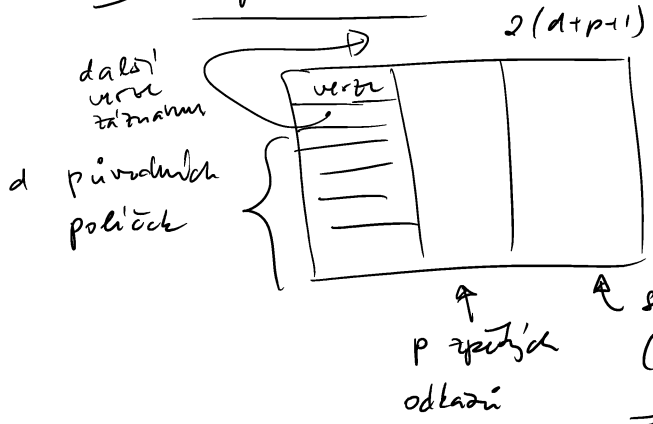
originální d.s. – záznamy

. d políček

. $\leq p$ odkazů na každý záznam v daném okamžiku

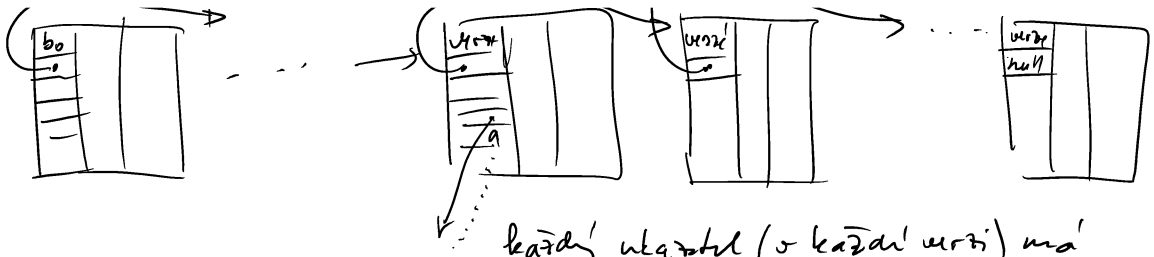
→ lze udělat plně perzistentní, často na operaci se prodlouží $O(1)$ – krát (amortizovaně), prostor úměrný počtu elementárních změn políček

Dk: implementace



→ zaznamenávají se i změny zpětných odkazů (rozdíl oproti částečné perz.)
≈ odkaz jsou obousměrné

→ rodina záznamů odpovídající různým verzím původního záznamu

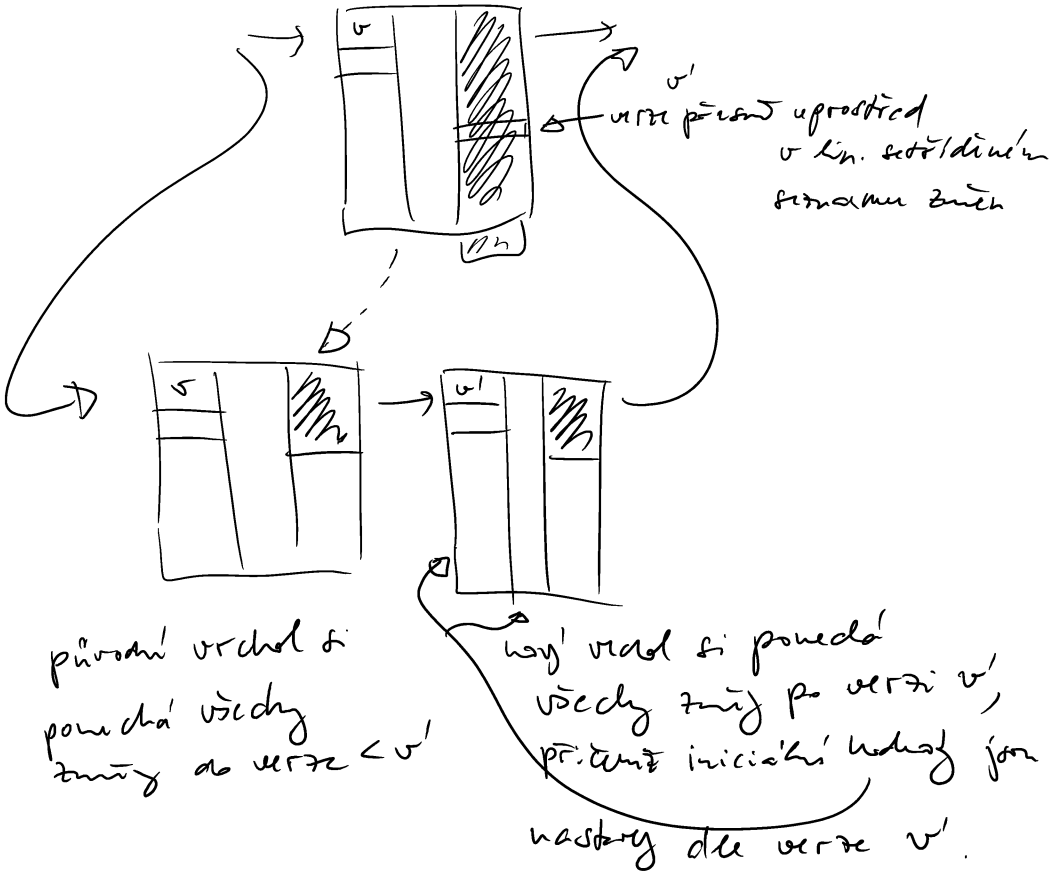


každý ukazatel (v každé verzi) má k sobě zpětný ukazatel stejné verze

pošlu se verzí v

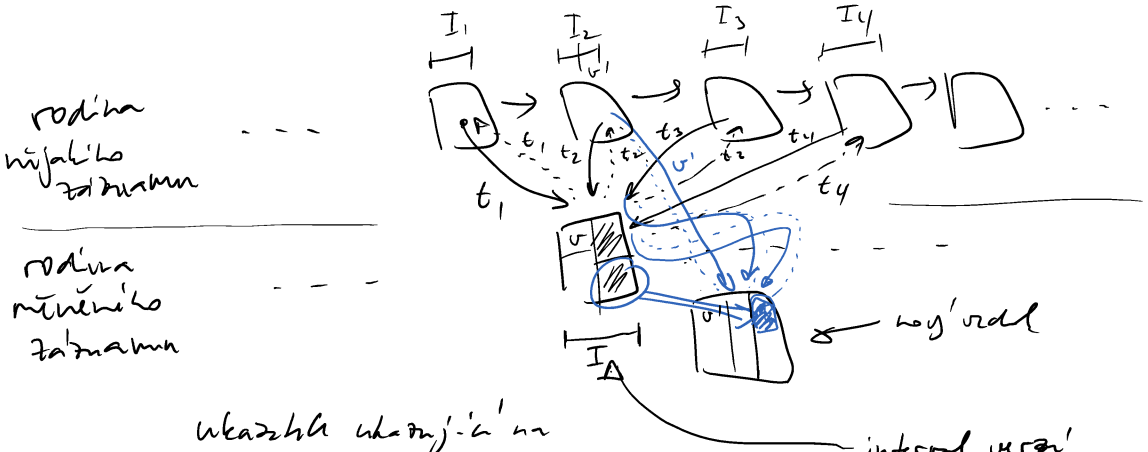
čtení: (jako u částečné pers.) v aktuálním záznamu, který by měl obsahovat hledanou verzi pošlu, projdu seznam změn pošlu a použiji hodnotu z (lineární) verze předchozí nebo rovné b.

zápis:
- pokud lze přidat zápis o změně, přidám zápis, [nižším-li ukazatel, zabraňujíc zpětný ukazatel]
- jinak musím štěpit aktuální záznam na dva



⇒ 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
přeskřížují na nový záznam v'

Přepsáním jejich hodnoty

pokud není shoden okolností žádný z těchto ukazatelů
verze v', pak přidám do příslušného záznamu (I_2)
Záznam o změně, že od verze v', ukazuje
daný ukazatel na nový vrchol
→ může dojít k přečtení dobrého záznamu
a dalším rekurzivním sestupem!
(rekurze se zastaví nejpozději v okamžiku,
kdy každá verze má svůj vlastní
záznam. (Změny hodnot ve verzi
dítěte jako verze záznamu se aplikují
přímo na inicializační hodnoty.))

- podobná procedura se použije pro ukazatele
vycházející z v. (Těm, které se přesunou,
přepíšeme odpovídající zpětné ukazatele, pak
přidáme zpětné ukazatele na verzi v' pro
inicializační hodnoty ukazatelů záznamu v'.)

Pozn.: rekurzi je třeba dělat průchodem do šířky:

1) rozstřílíme všechny záznamy, které v mínulém
kole přečetla

1) rozštěpíme všechny záznamy, které v minulém kole přetekly

2) přidáme potřebná ukazatele na právě rozštěpené záznamy. Pokud se nový ukazatel již nevejde do seznamu svého příslušného záznamu, uložíme zprávu do dočasného bufferu a poznamenáme si, že záznam je nutné v dalším kole rozštěpit.

- V daném kole může záznamem přibýt nejvýše $3A+3p+2$ nových ukazatelů, že každý existující ukazatel nejvýše jeden nový. (tj. jeho vrcholu)

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

  (po rozštěpení může každý z nových záznamů mít jeden $\leq \frac{1}{2}(d+p)$ nezpracovaných záznamů v bufferu $\rightarrow$ další štěpení v příštím kole
  resp. $\leq (d+p)$)

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

- 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 mohou způsobit další štěpení)
  (Ostatní ukazatele se přepíšou na místě.)

$\rightarrow$ obojí $\leq l$ štěpení

- amort. čas zápisu $\leq c + c$ (skutečný čas) + (změna potenciálu)

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

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

$c\ell$: práce spojená se vznikem ℓ vrcholů a potřebnými aktualizacemi

$-c2(d+p+1)\ell$: pokles potenciálu v důsledku rozštěpení ℓ vrcholů

$c(d+p)\ell$: horní odhad na nárůst potenciálu v důsledku nových záznamů o změnách vzniklých v důsledku štěpení ℓ vrcholů)

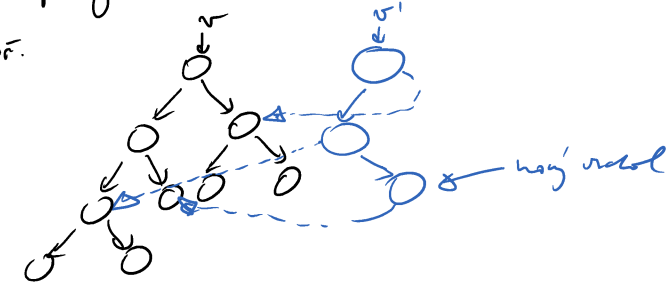
$\leq 2c$

funkcionální perzistence

Př.: • červeno-černý strom s aktualizací shora dolů bez rodičovského ukazatele.

— při insertu, jak strom procházím a upravuji: kopíruji cestu do nové verze

Př.:

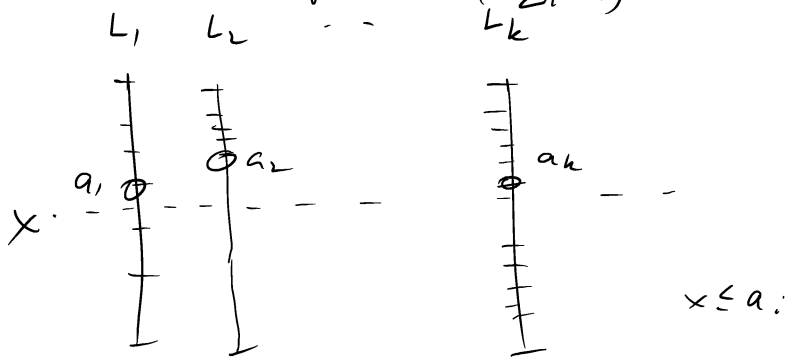


→ přibude $O(\lg n)$ nových vrcholů

• Úloha na operaci $O(\log n)$, jí asymptoticky jako předtím

Problém: Množiny $L_1, L_2, \ldots L_k \subseteq \mathbb{N}$, $|L_i| \leq n$. Chci d.s., která na dotaz $x \in \mathbb{N}$ vrátí nejbližší větší nebo rovno číslo z každé množiny L_i.

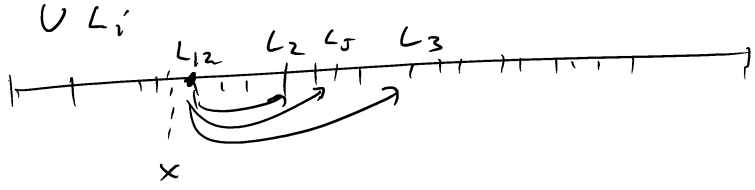
→ ideálně – čas $O(\log(n) + k)$
prostor $O(\sum |L_i|)$



jednodušší řešení

1) každou množinu L_i reprezentuj: 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



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

Řešení – kaskádovité podílů ("fractional cascading")

definujeme: $M_k = L_k$

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

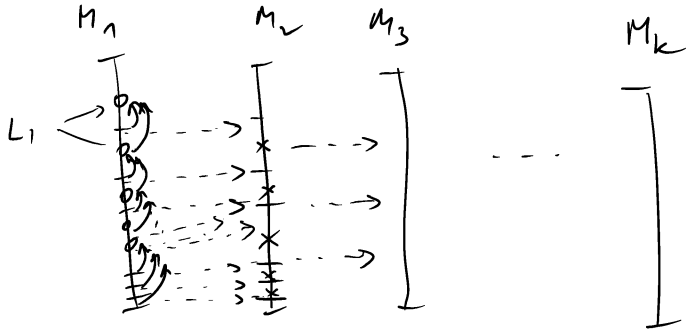
- $|M_i| \le \sum_{j \ge i}^{k} \frac{|L_j|}{2^{j-i}}$ Dk: indukce na i.

$\Rightarrow \sum |M_i| \le \sum \sum \frac{|L_j|}{2^{j-i}} \le \sum \sum |L_i|$

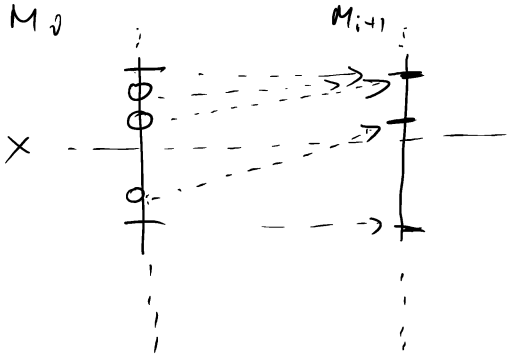
$$\Rightarrow \sum_{i=1}^{k} |M_i| \le \sum_{i=1}^{k} \sum_{j \ge i}^{k} \frac{|L_j|}{2^{j-i}} \le \sum_{i=1}^{k} \sum_{j=1}^{k} \frac{|L_i|}{2^j}$$

$$\le 2 \sum_{i=1}^{k} |L_i| \quad \blacksquare$$

- Místa L_i reprezentují 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, nalezeme mezi klíči prvky M_1 hledané x. To mi dovolí nalézt nejbližší prvek k x v L_1, přeskočím do M_2 a opakuji hledání pro x pomocí jednoho porovnání



pokračuji pro další M_i až do $i=k$

→ čas $O(\log n + k)$

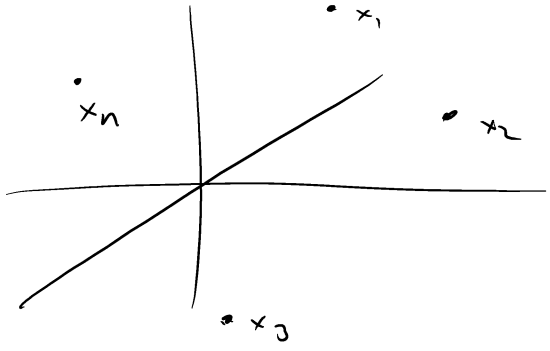
- $\log n$: bin. hledání v M_1
- k: průchod přes k polí M_i

bin. hledání v M_1 pak M_2

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

Vícedimenzionální data

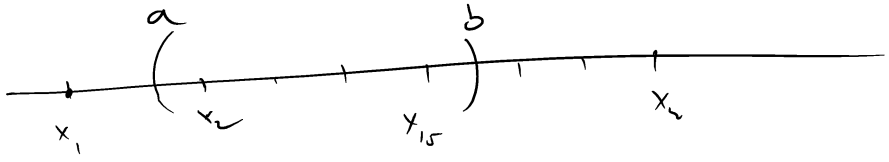
body v $\mathbb{R}^d$



- 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. vyh. strom

body mezi a a b

- u vyvážených stromů $O(\lg n + k)$ čas na dotaz

počet bodů výstupu

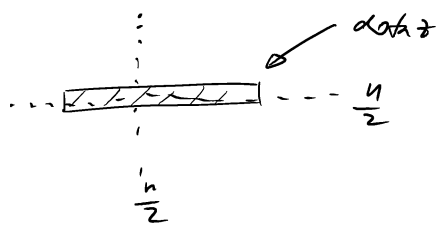
- počítání dotaz (zajímá nás pouze počet bodů v daném

intervalu)

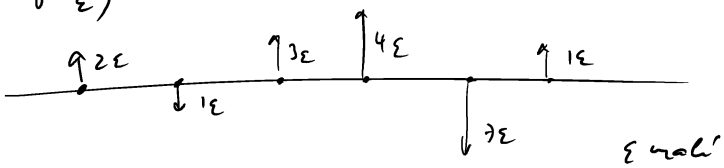
čas $O(\log n)$, pokud si ve vnitřních uzlech udržujeme počet listů v podstromu

- $d = 2$ ... 2-dim

Př:



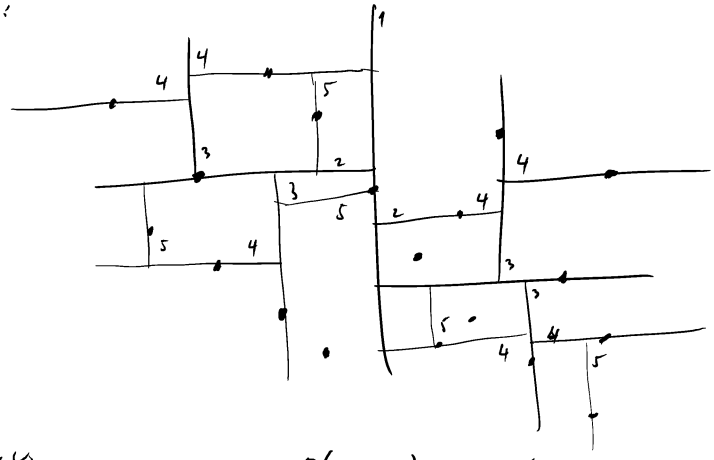
Bývá: žádné dva body nemají totožnou žádnou souřadnici (každý bod můžeme posunout o ε)



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 / nekonvexní obdélník)
  to lze spočítat při průchodu od kořene

- Find (vrátí v. interval R)

• Find (uzel v, interval R)

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

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

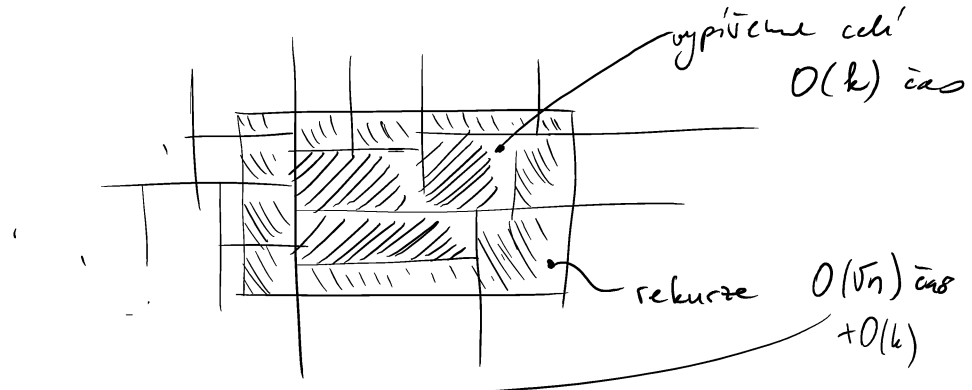
1) pokud je v list, vypiš příslušný bod pokud leží v R

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)

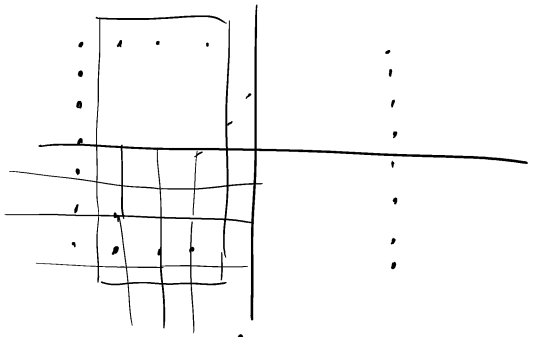
3) pokud oblast pravého podstromu v_R leží celá v R,
vypiš všechny její body
jinak pokud tato oblast protíná R → Find(v_R, R).

4) Konec.

Časová složitost:



Prop: P:

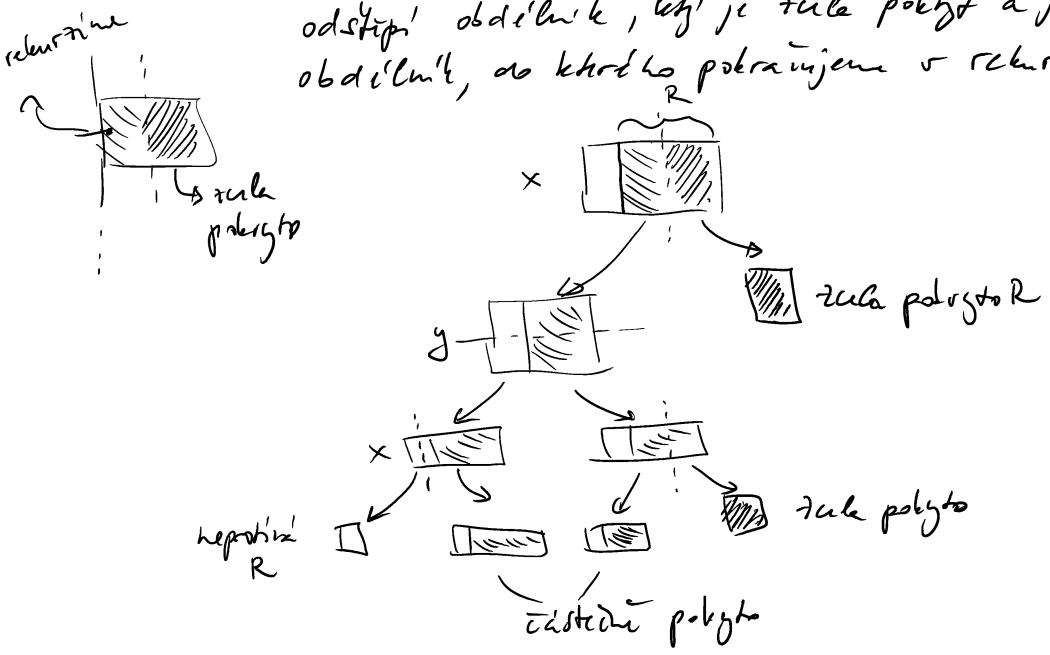


mřížka $\sqrt{n} \times \sqrt{n}$, každý bod posunut o zanedbatelné ε.

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

(hrubší obdélník

• uvažujme levou hranici a nejlepší pokrytí obdélníky R k ní přiléhající. Rozhodování podle os x → nebo disjunktní SR
odstupňí obdélníků, když je rovná pokryt a jedem obdélníků, do kterého pokračujeme v rekurzi.



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

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

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

$d > 3$ stříkáním větvení do jednotlivých dimenzí

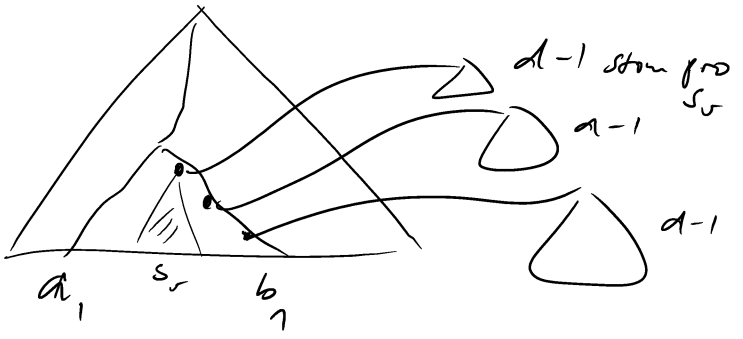
čas $O(n^{1-\frac{1}{d}} + k)$ na dotaz, prostor $O(n)$.
("stejná" analýza - až na každou d-tou vrstvu větvení na 2)

Intervalové stromy ("range trees")

• $d = 1$ ... stejné jako výše.

- $d > 1$

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



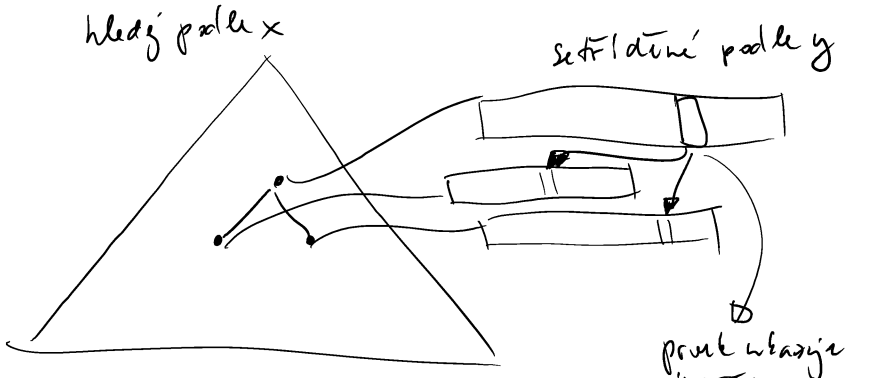
vyhledávání $O(\log^d n)$

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

- vyhledávání pro $d=2$ lze zlepšit

  na $O(\lg n)$ pomocí techniky kaskádování

  → pro obecné d čas $O(\lg^{d-1} n)$ na interních datech



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

poli podle y!

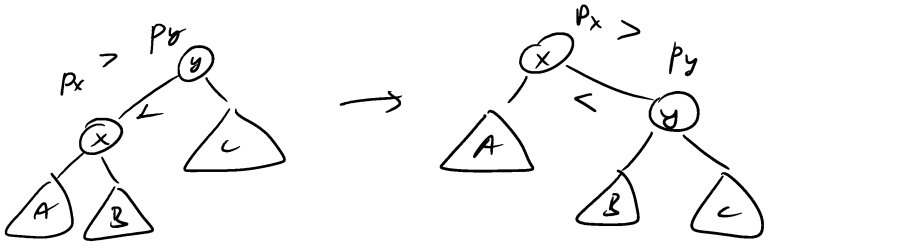
- Náhodné vyhledávací stromy

Treaps

- binární vyhledávací stromy, kde každý vložený prvek má navíc přiřazenou prioritu a všechny stromy 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 rovnou náhodnou prioritu $\in [0;1]$, vložit ho do nového listu, kam by měl padnout jako u bin. vyhl. stromu. Pokud náš vrchol (x) porušuje podmínku priority vůči svému otci (y), provádíme rotaci, dokud není vše v pořádku. (Vrchol x bublá nahoru.)



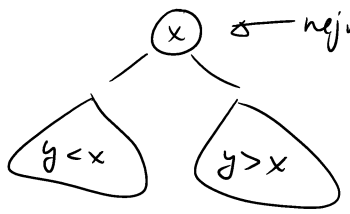
Delete (x) – sníž prioritu vrcholu x na $-\infty$ a pomocí rotací ji probublej až do listu. Smaž list x.

→ Treaps existují a lze přidávat nové prvky.

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

Dk: 1) kořenem je prvek s nejvyšší prioritou. Náš 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 stejnými hodnotami a prioritou (indukcí/rekurzí)



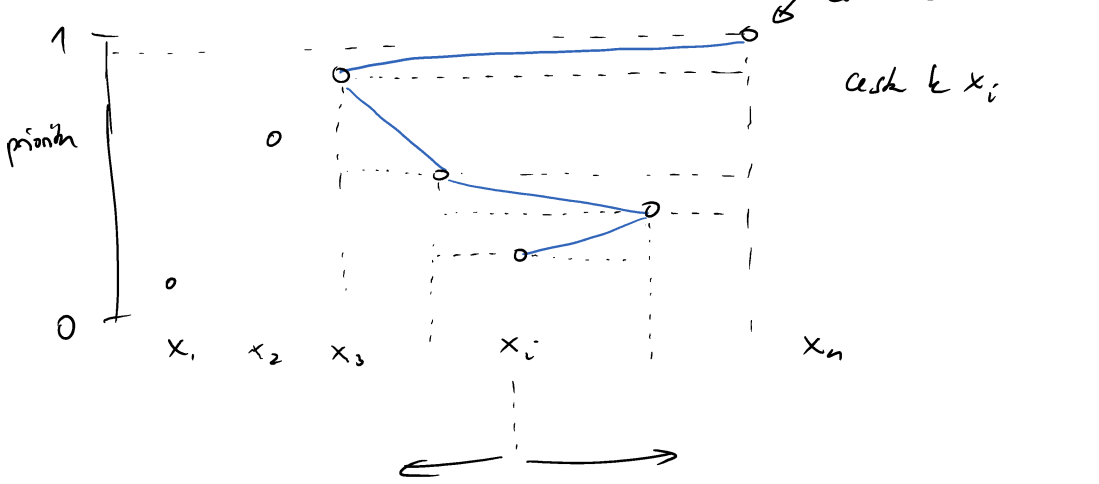
Důkaz: všechny priority různé.

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

(bez zjednodušení)

Treap s prvky $x_1 < x_2 < x_3 \dots < x_n$ (prvním dohromady)

- 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 nejmenší prioritu mezi $\{x_j, x_{j+1}, \dots x_i\}$
- pro $\forall j > i$, x_j leží na cestě od kořene k x_i
  $\Leftrightarrow$ x_j má nejmenší prioritu mezi $\{x_i, x_{i+1} \dots x_j\}$

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

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

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

"linearita očekávané hodnoty"

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

$j < i$:

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

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

$i < j$: obdobně

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

$\Rightarrow$

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

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

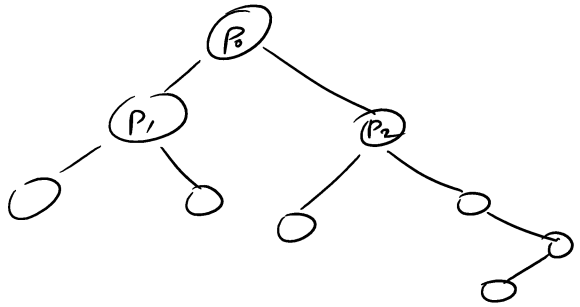
$\rightarrow$ čas na operaci $O(2 + 2 \ln n)$ (očekávaný)

• provedu-li m operací počínaje prázdným stromem, pomocí vhodných operací, ...

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

Analýza Quicksortu:

strom rozhodování Quicksortu / volby pivotů je izomorfní s náhodně vybraným vrcholem treap.



- každý prvek se během Quicksortu porovná se všemi pivoty na cestě ke kořeni
  → počet porovnání = hloubka prvku

→ očekávaná doba Quicksortu =

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

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

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

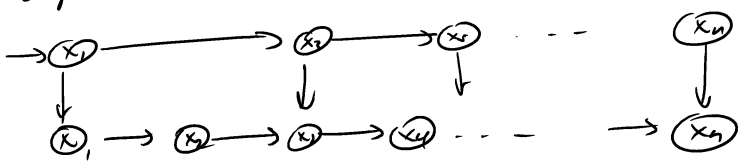
Skip list

uspořádaný seznam $x_1 < x_2 \dots x_n$

→ ⊙ → x_1 → x_2 → ... → x_n

doba vyhledání prvku $\leq n$

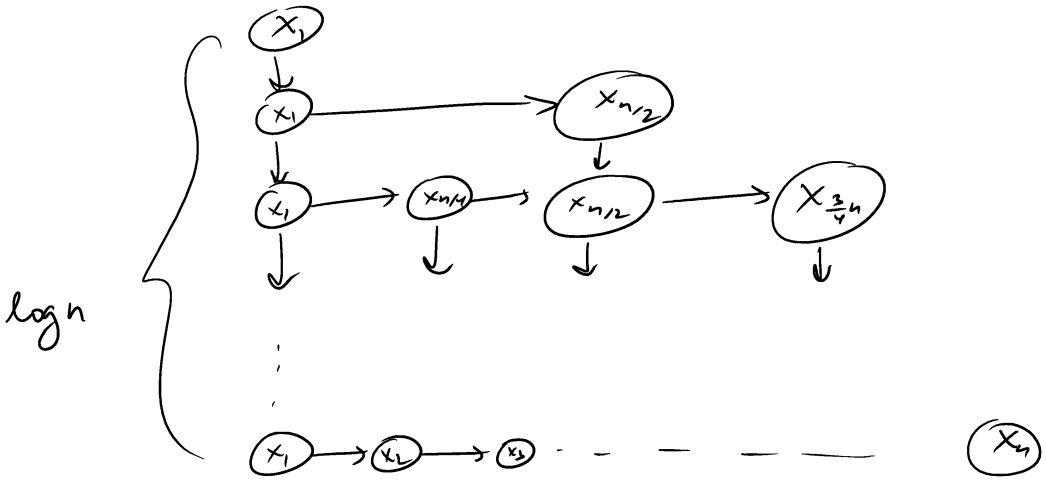
- zlepšení:



pomocný 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 kam prostředek atd.

→ náhodně

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

→ k kopií má pravděpodobnost $2^{-(k+1)}$.

prázdný

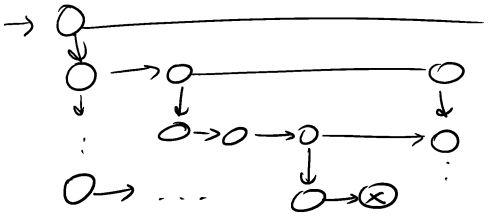
- vznikne struktura podobná tě deterministické

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

$$Pr[\exists x \; s \geq k \text{ kopiemi}] \leq n \cdot 2^{-k}$$

$$\Rightarrow Pr[\text{Skip list má výšku} \geq 2 \log n] \leq n \cdot 2^{-2\log n} = \frac{1}{n}$$

Očekávaná doba vyhledání x : uvažujme cestu od x ke kořeni



s pravd. $\frac{1}{2}$ vrchol má kopii o patro výše
$\Rightarrow$ šli jsme nahoru
s pravd. $\frac{1}{2}$ vrchol nemá další kopii
$\Rightarrow$ šli jsme vlevo

- Jaká je pravděpodobnost, že během $6 \log n$ kroků nedošli méně než $2 \log n$ kroků nahoru?

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

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

Binomická věta

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

$$\cdot \binom{n}{k} \leq \left(\frac{n}{n-k}\right)^{n-k} \left(\frac{n}{k}\right)^k$$

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

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

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

$\Rightarrow$ pst. že hloubka je určitě než $2 \lg n$ &
že během $6 \lg n$ kroků nedosáhneme
druhou hladinu $\leq \frac{1}{n} + \frac{2 \lg n}{n^2} \leq \frac{2}{n}$.

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

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

---

van Emde Boas trees $\simeq$ '76

(vEB stromy)

slovníková struktura pro uložení $S \subseteq U$

s operacemi

Insert
Delete
Find
} obyčejné operace

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

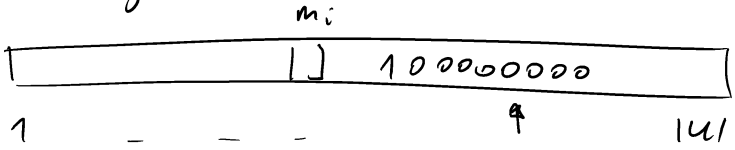
předchůdce & následník

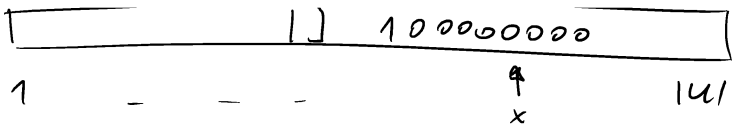
Řešení: binární vyhledávací stromy – čas na op. $O(\log n)$
prostor $O(n)$

Otázka: Lze líp, 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





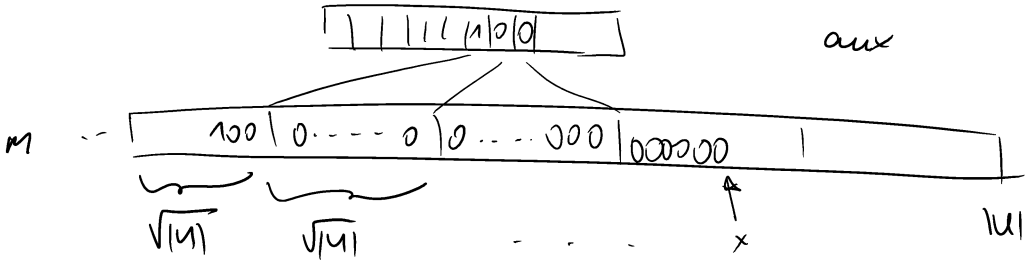
$m_i = \begin{cases} 1 & i \in S \\ 0 & \text{jinak} \end{cases}$

Insert, Delete, Find $O(1)$ čas

Succ, Pred $O(u)$ čas

musíme najít nejbližší jedničku, může být až na konci pole!

→ vylepšení — přidáme bitový vektor aux



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 $O(1)$

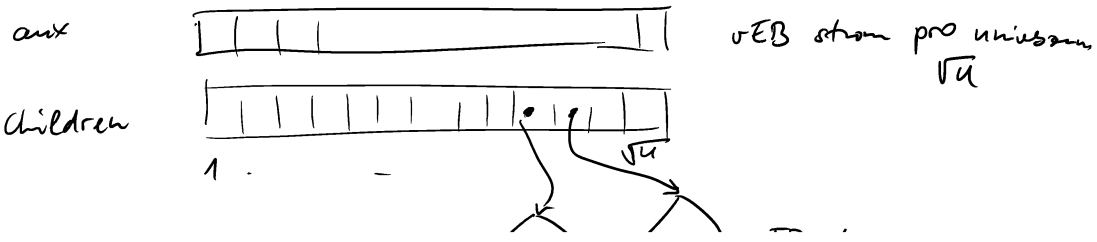
Delete, Succ, Pred $O(\sqrt{u})$

Pred(x) ... prohledej blok x $(j = x/\sqrt{u})$

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

- aux je strukturou řešící stejný problém, ale na univerzu velikosti $\sqrt{u}$.

Zobecnění → vEB stromy pro univerzum 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 nerekurzivně

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

Pozn: Předpokládáme, že VEB strom je na začátku naincializován jako prázdný, před započetím operací s ním.

- aux obsahuje seznam neprázdných stromů children, tj.: aux obsahuje j
  ⇔ children(j) je neprázdný VEB strom.

Insert(x): • 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 $< x$, pokud
x s min resp. max se
pokračuje dále
• $j = x / \sqrt{u}$ $i = x \bmod \sqrt{u}$
• pokud je children(j) prázdný (poznáme 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 $\sqrt{u}$.

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

pro univerzum velikosti ≤ 10 řešíme triviálně

- **Find(x)** – pokud $x \in \{min, max\}$ → jsme
  jinak children$(x/\sqrt{u})$. Find$(x \bmod j)$.
  (pokud je children$(x/\sqrt{u})$ neprázdný)

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

- **succ(x)** – $j = x/\sqrt{}$, $i = x \bmod \sqrt{u}$
  - pokud $i \leq$ children(j).max
    pak vrať $j*\sqrt{u}$ + children(j).succ(i).
  - j = aux.succ(j);
    vrať $j*\sqrt{u}$ + children(j).min;

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

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

- **Pred(x)** ... symetrický ke succ(x).

- **Delete(x)** – pokud je $x \in \{min, max\}$, nahraď
  ho druhým nejmenším/největším prvkem
  ($\tilde{=}$ min = children(aux.succ(0)).min
  — " — . delete(min))
  – $j = x/\sqrt{u}$, $i = x \bmod \sqrt{u}$
  pokud children(j) obsahuje pouhý prvek
  tj. $i = min = max$
  pak odstraň i & aux.Delete(j).
  jinak children(j).Delete(i).

... čas $O(\log \log |U|)$

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

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

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

$\rightarrow$ celkově tyto struktury aux přidají

$\simeq n^{1-\frac{1}{2^{i+1}}}$ synů listů

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

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

atd.

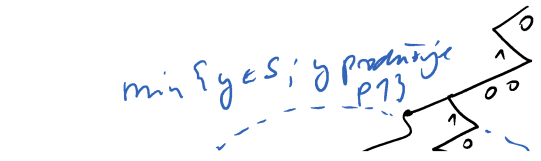
$\Rightarrow \sum \frac{n}{c^i}$ listů $= O(n)$

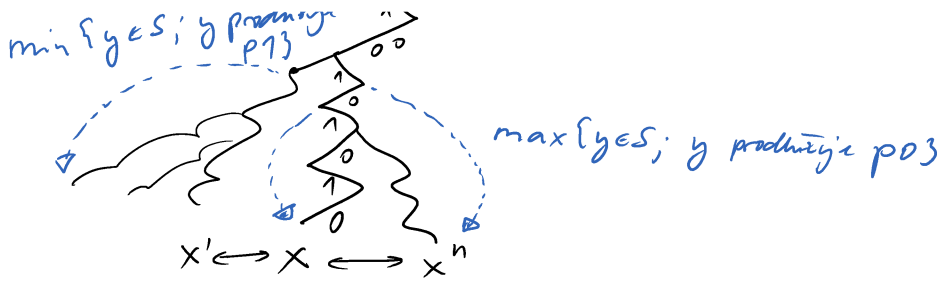
- x-fast trie

  - stejný problém jako u EB stromy
    - lepší prostor ... $O(n \cdot \log |U|)$ $\quad n = |S|$, $S \subseteq U$
    - čas o něco horší ... viz níže

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

$x = 0100101010$



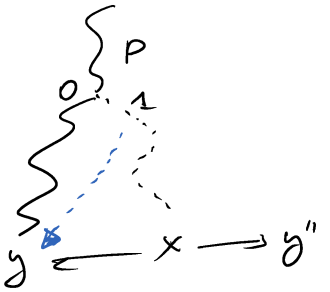


pokud p je jeden z prefixů,
- pokud p0 není prefix v S, pak si p pamatuje $\min\{y \in S, y$ má prefix $p1\}$
- pokud p1 není prefix v S tak p si pamatuje $\max\{y \in S, y$ má prefix $p0\}$

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

→ při: Find(x) nalezne nejbližší prefix x, který je v S. Pokud to není x → false
jinak → true

při: Succ(x) nalezne nejbližší 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

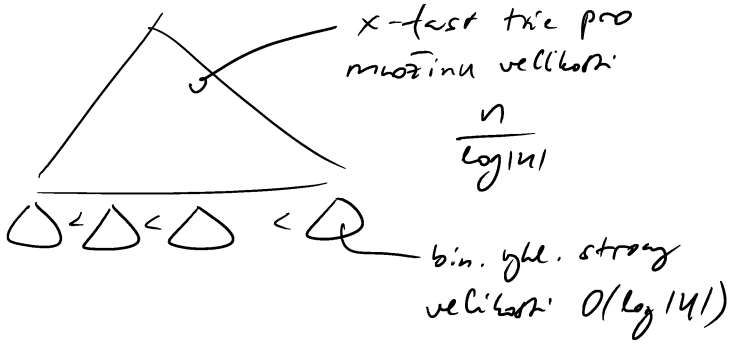
→ nejdelší prefix x hledám binárním vyhledáváním na jeho délce → čas $O(\log \log |U|)$

Insert(x) ... musím vložit $O(\log |U|)$ prefixů do hašovacích tabulek a upravit odkazy pro prefixy, kterými jsem již jednou $O(\log |U|)$ prošel

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

---

y-fast trie



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

  při Insertu štěpím dle potřeby
  při Delete slévám dle potřeby

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

→ stromeček má potřebu se slepit nejpříve

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

a má potřebu se slízat nejdříve

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

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

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

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

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

→ amortizovaný čas určen operací na podstromečcích → $O(\log\log|U|)$

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

Disjoint Set Union, Find:

Problém: chceme 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 $\{x\}$

Možné implementace

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

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

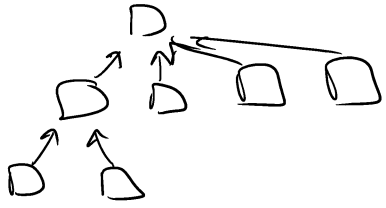
amortizovaně: n Makeset
m Union, Find, Makeset $(m \geq n)$
Čas $O(n \cdot m)$

2. Vylepšení 1.: Při Union(A,B) připoj kratší seznam za delší (musíme přenášet ukazatele na repr. prvek 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 právě tehdy, když seznam, ve kterém je, se zdvojnásobí.
$\Rightarrow$ max $\log_2 n$ - krát

3. stromy

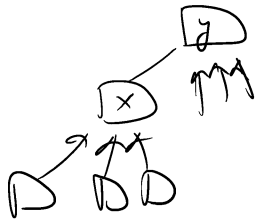


→ rank stromu

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);




```
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 prověřenou cestu

časová složitost: n operací Makeset z volného počátku m operací amortizovaně čas $O(m \cdot \alpha(n))$ kde $\alpha(n)$ je inverzní fce k Ackermannově fci

pro $f : \mathbb{N} \to \mathbb{N}$ (neklesající) definujme

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

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

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

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

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

Pozorování:
- Rank vrcholu se zvýší právě 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 vrcholu v hloubka 2 a více
(ostatní práci $O(n)$)

- takoví vrcholy už nejsi pevně dají rank

rozdělme všechny vrcholy do skupin A_i:

$$l_0 = 1 \qquad l_i = 2^{l_i}$$

$$A_i = \{v,\ v \text{ má rank } [l_{i-1}, l_i)\}$$

- $|A_i| \le \frac{2n}{l_i}$ (jednoduché) z (*)

počet průchodů přes hranu

$u \to v$ kde $u, v \in A_i$

a u je pevně zafixovaný

je během všech operací Find

nejvýše l_i, protože

při každém dalším Find

přijde na u jiného předka s vyšším rankem

→ čas strávený na těchto průchodech je

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

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 $O(n \log^* n)$

⇒ celkový čas $O(n \log^* n)$.

Operace s řetězci

abeceda .. Σ

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

např: $\Sigma = \{a, b, \ldots, z\}$

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

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

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

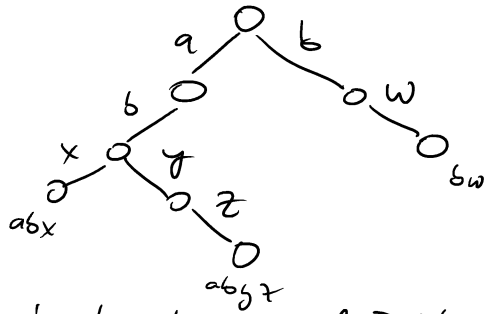
$\Sigma = \{0, 1, \ldots, 255\}$

slovníkový problém pro S

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

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 asociován řetězec $\alpha \in \Sigma^*$ získaný při průchodu od kořene k vrcholu

Member(x) – prochází strom od kořene a vybere 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.

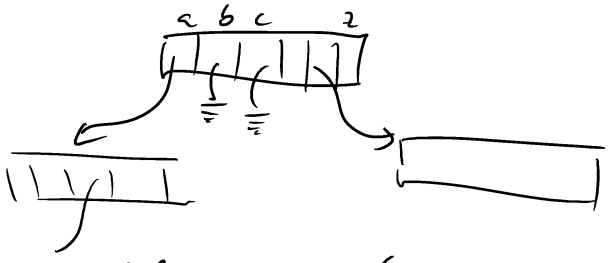
Insert(x) – prochází podobně jako u Member(x), jakmile dosáhnu chybějící hranu, hranu vytvořím, přidám nový list, přejdu do něj a pokračuji. Vrchol přidružený x označím,

že je v S

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

Implementace:

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



– $O(|\Sigma|)$ prostoru na vrchol, $O(|\Sigma| \cdot m)$ celkem

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

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

Delete(x) . —11—

2) každý vrchol má spojový seznam svých potomků

– celkový prostor $O(m)$

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

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

Delete(x) .. —11—

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

3) binární vyhledávací strom potomků

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

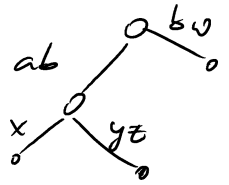
prostor $O(n)$

4) hašovací tabulka potřetí

operace $O(|x|)$

prostor $O(n)$ ✓

Komprimovaný trie

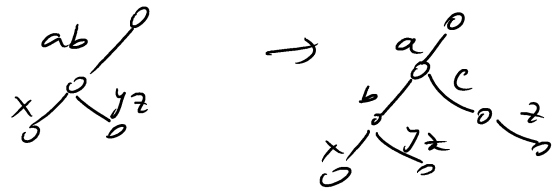


- hrany obsahují slova $\in \Sigma^*$
  hrany vycházející z jednoho uzlu se liší v prvním znaku

  → při member(x) stačí porovnávat první znak, pokud si u uzlů pamatují délku označující hrany na konci však musíme ověřit, že jsme našli to, co jsme hledali

- při vkládání občas musíme rozdělit hranu

  insert(acz)



→ o něco kompaktnější reprezentace

Problémy s řetězci

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

$P \in \Sigma^n$ ... hledané slovo

úkol: najdi výskyt / všechny výskyty P v T

Klasika Aho–Corasick – sestrojím konečný automat z P a spustím ho na T

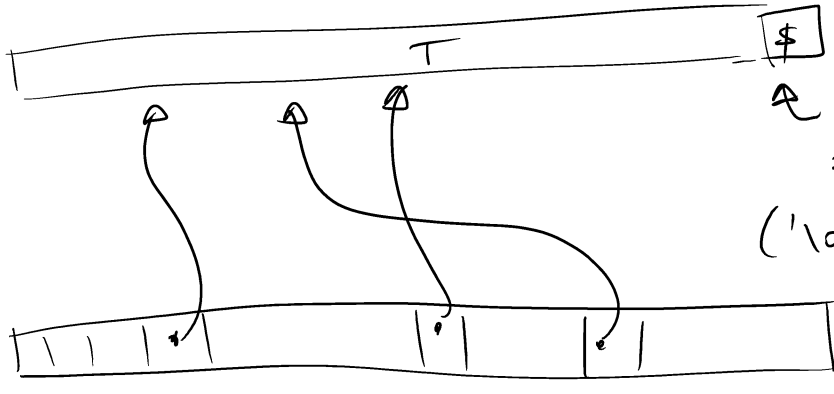
čas $\approx O(m+n)$.

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

řešení: sufixová pole, sufixový strom

Sufixová pole

'$' $\notin \Sigma$



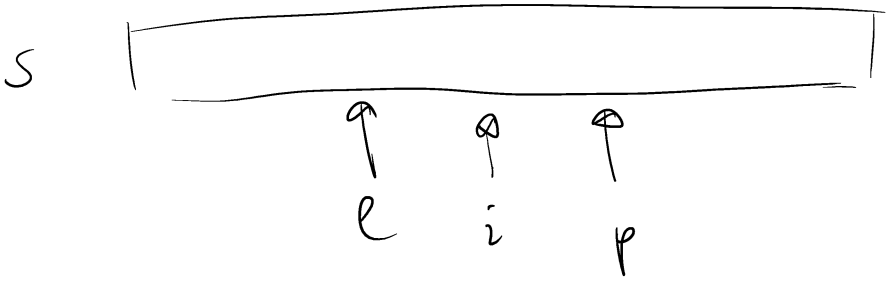
lexikograficky setříděné pole všech sufixů T, tj. pole řezů $T[1..m]$, $T[2..m]$, ... $T[m..m]$

→ v poli lze binárně vyhledávat řetízek P

- hledám lexikograficky nejmenší řetízek větší - rovný P.

· pokud se P shoduje s tímto řetězcem na prvních |P| pozicích, nalezl jsem výskyt P v T, bezprostředně následující řetězy 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 $p > l+1$

$i \leftarrow \lfloor \frac{l+p}{2} \rfloor$

pokud $S[i] <_{lex} P$ pak $l \leftarrow i$ (*)

jinak $p \leftarrow i$.

return l

end.

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

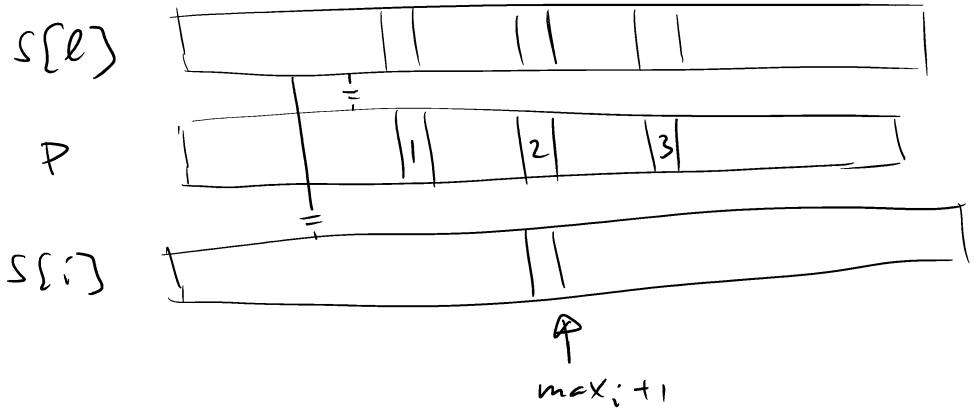
- lze zlepšit na $O(n + \log m)$

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

předpokládejme, že známe zadarmo
$lcp(S(\ell), S(i))$ 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 udržujeme:
$max_\ell = lcp(S[\ell], P)$
$max_p = lcp(S[p], P)$

pokud $max_\ell > max_p$ porovnání (*) provedu následovně:
$max_i = lcp(S[\ell], S(i))$



tři možnosti:

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

neboť $S[\ell]$ a P se shodují
až do (3) a $S[i]$ se odlišují 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 $max_i \rightarrow r$
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 $max_\ell > max_p$.

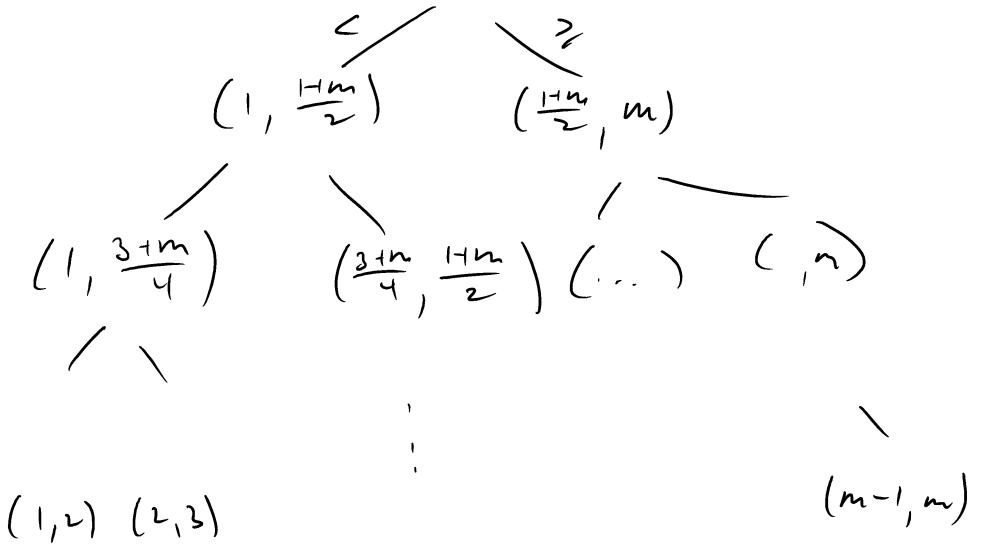
Pozn.: pokud zjistím, že P je prefix $S[i]$,
vyhledávání zastavím a našel jsem
výskyt. Běžný binárního vyhledávání
tak mírně předpokládat, že P není
prefixem ani $S[\ell]$ ani $S[p]$.

Čas na (*) je pak v součtu $O(n + \log m)$,
neboť v P se posunuji pouze doprava

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

ℓ p
i i
$(1, m)$

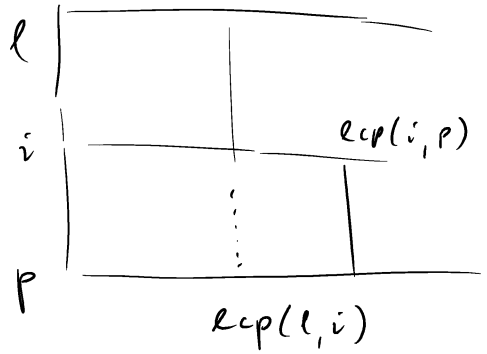
$<$ / \ $>$



hrany l, p, i se posuzují jako při průchodu jedním z vrcholů. 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

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



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

- $lcp(i, i+1)$ $\forall i = 1 \dots n-1$ lze spočítat

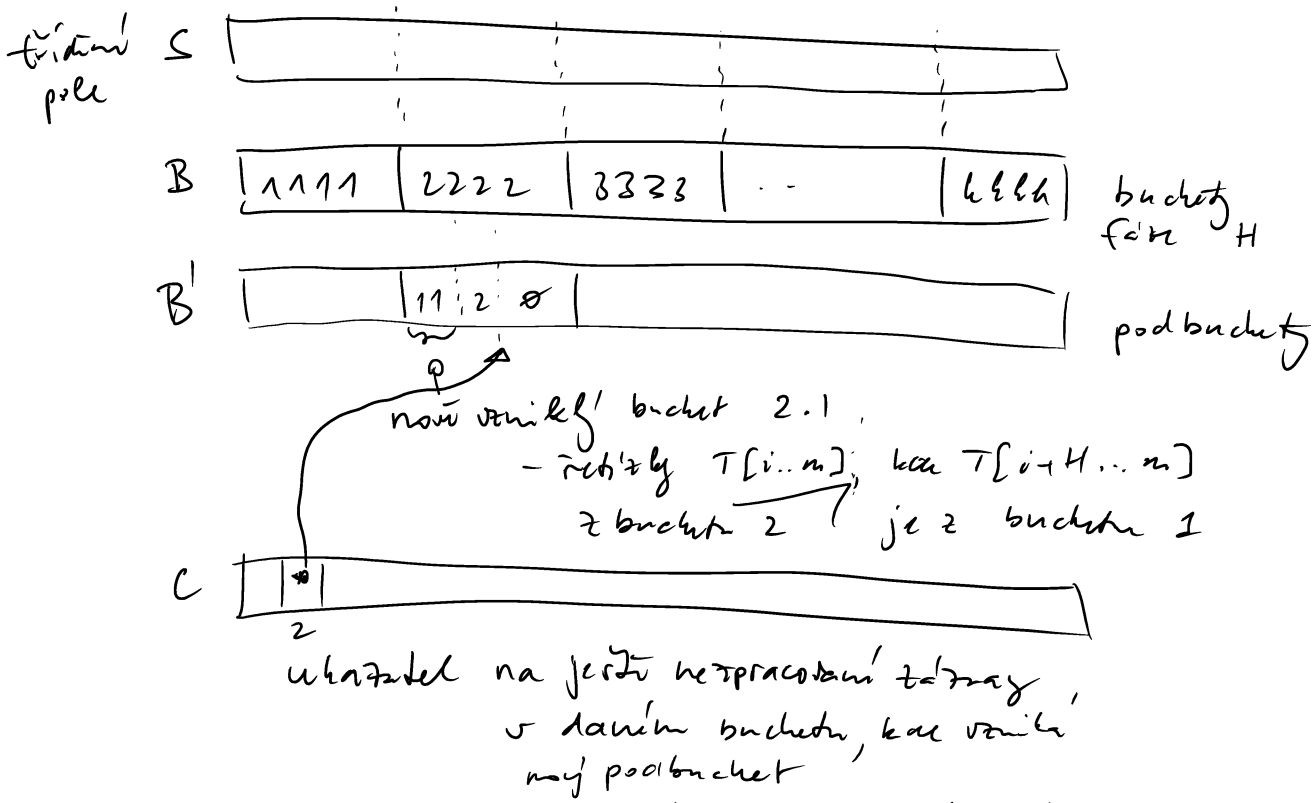
v čase $O(n \lg n)$

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

      ↓

      využívám, že již znám rozdělení řetězů podle H-symbolů
      Pro pozornání $T[i,n]$ a $T[j,n]$ se stejnými prvními H symboly lze použít informaci o bucketech $T[i+H,n]$ a $T[j+H,n]$.
    - složitější implementace bere buckety v lex-pořadí, v rámci bucketu jsou řetězce $T[i..n]$ po řetězcích a přidáváme řetězce $T[i-H...n]$

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



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

Sufixové stromy: trie sestavené z $T[1..n], T[2..n], \ldots T[n,n]$

– koncepčně přímočařejší než sufixové pole, ale sufixové pole je kompaktnější, z hlediska časové složitosti jsou srovnatelné.

- mnoho různých algoritmů na konstrukci suf. polí & stromů.
- komprimované trie, kde se místo řetězců na hranách pouze odkazuje 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")

a mnoho dalších

Samoupravující se stromy

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

Otázka: Optimální uspořádání prvků v seznamu?

→ podle klesající pravděpodobnosti p_i

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

očekávaný čas operace Find (x_i)

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

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

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

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

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

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

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

Platí:

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

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

Sečtěme přes i.

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

$Pr(x_j$ je před x_i v čase $k] \leq$

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

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

je $x_j \mid x_i$ nebo x_j se vyskytlo $\emptyset$

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

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

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

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

$$\leq 2 \sum_i p_i \sum_{j<i} \frac{p_j}{p_i + p_j} + \text{—}''\text{—}$$

$$\leq 2 \sum_i p_i (i-1) + \sum_{i \neq j} (1 - p_i - p_j)^{k-1}$$

$\leq E[T]$; $\sum_{i \neq j} (1 - p_i - p_j)^{k-1} \to 0$ pro $k \to \infty$

Pozn.: Očekávaný čas MFR strategie je tedy nejvýše 2-krát delší než 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
slovníku pro danou posloupnost operací Find.

Dynamizace datových struktur

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

$P(n)$ ... čas na vytvoření struktury s n prvky ("preprocessing")
$Q(n)$ ... čas na dotaz na strukturu o n prvcích

chci z ní 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 $O(Q(n) \cdot \log n)$

amortizovaně / v nejhorším případě

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

Idea: (podobná binomiálním haldám)

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

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

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

a pro každou množinu A_i máme vyvořenou přidruženou dod. str. $S(A_i)$

- Query (x) – spočítám 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.

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

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

apod.

• Insert (x) - vytvoř $A_0' = \{x\}$ a $S(\{x\})$.
Dokud existují dvě množiny A_i a A_i'
stejné velikosti sluč 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 reprezentována spojným seznamem prvků, tedy sloučení lze provést v konstantním čase.

• předpoklady (rozumné) : 1) $\frac{P(n)}{n}$ je nejhorší

2) $\frac{S(n)}{n}$ —//—

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

3) čas na tvoření $S(a)$ je menší než na její vytvoření.

Amortizovaný čas

n ... Insertů $\Rightarrow$ 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

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

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

- Stejného času lze dosáhnout v nejhorším případě.

Idea: A_i a A'_i rozhodují během jednoho insertu, ale rozdílnou prací na budování $S(A_i \cup A'_i)$ do 2^i částí, každá z nich trvající $\frac{P(2^{i-1})}{2^i}$ času. Při následujícím insertu uděláme vždy jednu část z každého probíhajícího slučování (tím je maximálně k)

- Dokud není $S(A_i \cup A'_i)$ hotovo, použijeme na dotazy $S(A_i)$ a $S(A'_i)$, jakmile je hotovo, $S(A_i)$ a $S(A'_i)$ zahodíme.
- $S(A'_i \cup A_i)$ bude hotovo dříve než mi přibude další množina velikosti 2^i.

Dynamizace: (Insert + Delete)

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

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 úrovně 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 zrušíme

(A_i, F_i, D_i) a vytvoříme

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

o úroveň níže

- A_i si udržuji v binárním vyhledávacím stromě

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

- př.: Insert(x) vytvořím $(\{x\}, \emptyset, \emptyset)$ a aplikuji pravidla údržby
- př.: Delete(x), kde $x \in A_i$ z (A_i, D_i, F_i) v 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.

⇒ 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 delecí, ale může prvek x vystupovat ve víc tvorbách $S(A_i)$ než jen k. Tyto musíme účtovat deletím.

⇒ prvek x si předplatí tvorbu $S(A_i)$ na k úrovních při svém insertu

- v průběhu času musí vědět platit, že x má předplacený všechny úrovně $j > i$, když x je v dosud dvíli v A_i.

Pravidlo údržby 1) pak smaže z času předplaceného prvky v A_i a A_i'

Pravidlo údržby 2) se vyvolá pouze v důsledku 2^{i-1} delších z množiny A_i; v okamžiku vyvolání tohoto pravidla mají prvky v A_i předplacený vždy úroveň $> i$, takovou se ale přesunou o úroveň níže, čili potřebují, aby za ně někdo předplatil i tuto úroveň. To musí učinit prvky z $D_i \cup F_i$, množiny D_i a F_i se v tento okamžik vyprázdní $\Rightarrow$ účtujeme jim předplatné jednou.

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

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

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

- předplatné
- označení x v příslušné struktuře $S(A_i)$
- údržba bin. vyh. stromu A_i

- Alternativně to odpovídá analýze

s potenciálem

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

Delším každý složený prvek se smaže jenom jednou,
lze počítat $\frac{P(n)}{n}$ z času dělek 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:

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

1) pokud dvě množiny A_i a A_j mají stejný rank, vytvoří pro ně $S(A_i \cup A_j)$ a sníží jim dohromady

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

Tato pravidla odpovídají výše uvedenému popisu.

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