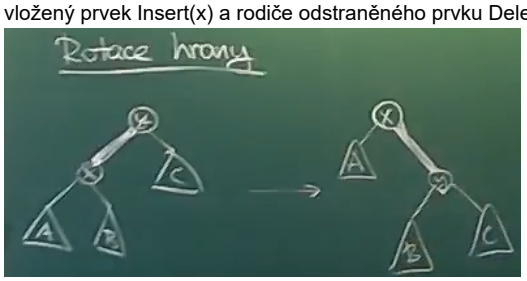
Zkouška

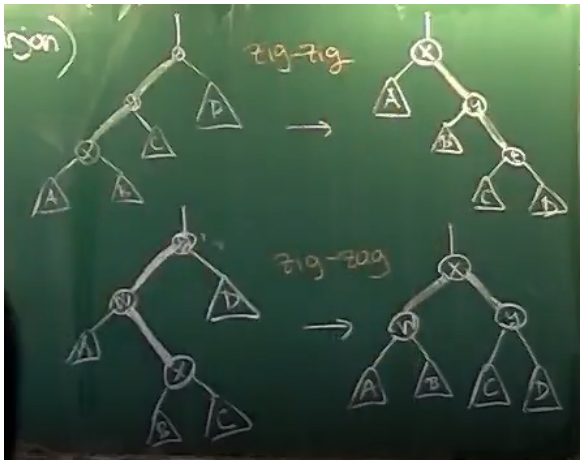
VELKÉ OTÁZKY

Definujte Splay strom. Vyslovte a dokažte větu o amortizované složitosti operace Splay.

Binární strom využívající splay operaci na vyvážení. Splay rotace vyrotuje rotovaný prvek do kořene stromu. Splay se používá na hledaný prvek Find(x), vložený prvek Insert(x) a rodiče odstraněného prvku Delete(x).



Je potřeba používat dvojité rotace vždy když to jde jinak by amortizovaná analýza nefungovala.



(pomůcka - nejdřív rotate parent x pak rotate x)

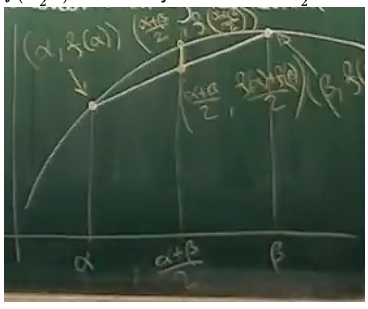
Amortizovaná analýza

$T(v)$ je podstrom s kořenem ve v
$s(v) = |T(v)|$ je velikost podstromu v (1 ... n)
$r(v) = log_2(s(v))$ je rank (0 ... log n)
Potenciál $\Phi = \sum_v r(v)$ (0 ... n log n)

(Přednáška 03 začátek)
Hlavní Lemma: Amortizovaná složitost operace $Splay(x) \leq 3(r'(x) - r(x)) + 1$ (max **log n**). Kde $r'(x)$ je rank po rotaci a $r(x)$ je rank před rotací.

Lemma A: $\forall \alpha, \beta \in \mathbb{R}^+ : log\frac{\alpha+\beta}{2} \geq \frac{log\,\alpha + log\,\beta}{2}$, (logaritmus průměru je větší roven průměru logaritmů)

Důkaz Lemma A: Logaritmus je konkávní funkce, takže pokud jdeme z libovolného bodu funkce $(\alpha, f(\alpha))$ půl cesty k druhému bodu funkce $(\beta, f(\beta))$, tak $f(\frac{\alpha+\beta}{2})$ bude vždy větší než $\frac{f(\alpha)+f(\beta)}{2}$.



Důsledek Lemma A: (ze začátku použito pravidlo pro dělení logaritmů na vzorec z Lemma A)
$log(\alpha + \beta) - log\,2 \geq \frac{log\,\alpha + log\,\beta}{2} \mid mult\ 2$
$2log(\alpha + \beta) - 2 \geq log\,\alpha + log\,\beta$

Značení: $r_0, \ldots, r_k$ $r_i =$ rank po i-tém kroku

Tvrzení B: i-tý krok splay operace má amortizovanou cenu $\leq 3(r_i - r_{i-1}) + 1$ ("+1" pouze v zig)

Plynutí hlavního lemma z Tvrzení B: Amortizovaná cena splay operace se intuitivně dá pochopit jako suma všech amortizovaných cen jednotlivých kroků splay operace.

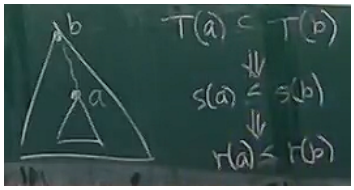
$$Splay(v) = 3(\sum_{i=1}^{k} r_i - r_{i-1}) + 1$$

Zde však nastane uvnitř sumy následující situace (např. pro k = 3):
$r_3 - r_2 + r_2 - r_1 + r_1 - r_0$, což je rovno $r_3 - r_0$, obecně tedy $r_k - r_0$,
tím pádem amortizovaná cena splay operace je:

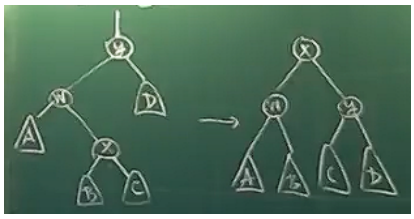
$$Splay(v) = 3(r_k - r_0) + 1$$

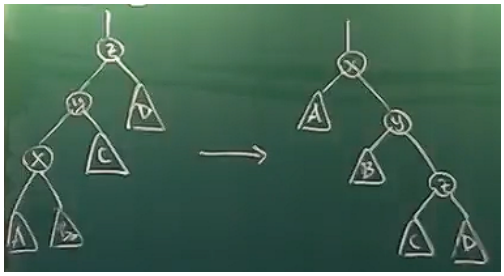
což je stejná jako v našem Hlavním Lemmatu. Nyní stačí dokázat Tvrzení B.

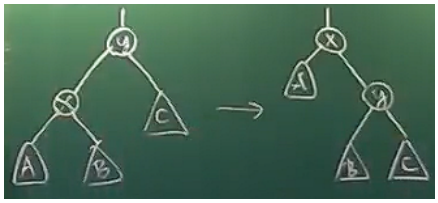
Pozorování C:



Důkaz Tvrzení B: Potřebujeme ukázat, že pro všechny druhy kroků (Zig, Zig-Zag, Zig-Zig) platí amortizovaná složitost z Tvrzení B.

- **Zig-Zag** - Reálná cena je 2, amort. cena (A = reálná cena+ změna potenciálu) je 2 + změna potenciálů, potenciál se však mohl změnit jenom vrcholům **x**, **w** a **y**
  - máme $A \leq 2 + r'(x) - r(x) + r'(y) - r(y) + r'(w) - r(w)$
  - chceme $A \leq 3(r'(x) - r(x))$
  - $r'(w) + r'(y) = log\ s'(w) + log\ s'(y) \leq$ (Lemma A) $2log(s'(w) + s'(y)) - 2$
  - Ale $s'(w) + s'(y)$ je celkový počet vrcholů v podstromech **w** a **y** (na obrázku níže to je ten vpravo) a to lze shora odhadnout počtem vrcholů v podstromu **x** ->
  - $\leq 2log(s'(x)) - 2 = 2r'(x) - 2$
  - Zatím tedy po dosazení máme: $A \leq 3r'(x) - r(x) - r(y) - r(w)$
  - Podle Pozorování C můžeme odhadnout ze spodu $r(y)$ jako $r(x)$ tj. $r(y) \geq r(x)$. Dále vynásobíme -1 -> $-r(y) \leq -r(x)$ a stejný postup uděláme pro $r(w)$. (oba vrcholy jsou nad **x**)
  - Po dosazení dostáváme: $A \leq 3r'(x) - 3r(x)$, což je to co jsme chtěli.
  - 
- **Zig-Zig** - (zde platí stejné věci jako u Zig-Zag)
  - máme zase $A \leq 2 + r'(x) - r(x) + r'(y) - r(y) + r'(z) - r(z)$
  - chceme zase $A \leq 3(r'(x) - r(x))$
  - $r(x) + r'(z) = log\ s(x) + log\ s'(z) \leq$ (Lemma A) $2log(s(x) + s'(z)) - 2$
  - Podle Pozorování C můžeme $s(x) + s'(z)$ shora odhadnout velikostí podstromu **x** = $s'(x)$ a z toho logaritmus je nový rank **x** = $r'(x)$
  - z toho vyplývá $2log(s(x) + s'(z)) - 2 \leq 2r'(x) - 2$
  - a po dosazení do pravé strany původní rovnosti $r(x) + r'(z) = log\ s(x) + log\ s'(z)$ dostáváme po upravení $r'(z) = 2r'(x) - r(x) - 2$
  - takže zatím máme $A \leq 3r'(x) - 2r(x) + r'(y) - r(y) - r(z)$
  - dále pomocí Pozorování C: $r'(y) \leq r'(x)$ -> $A \leq 4r'(x) - 2r(x) - r(y) - r(z)$
  - dále pomocí Pozorování C: $-r(y) \leq -r(x)$ -> $A \leq 4r'(x) - 3r(x) - r(z)$
  - dále pomocí Pozorování C: $-r(z) = -r'(x)$ -> $A \leq 3r'(x) - 3r(x))$
  - to však je to, co jsme chtěli.

- 

- **Zig** - reálná cena je 1
  - $A = 1 + r'(x) - r(x) + r'(y) - r(y)$
  - Podle Pozorování C $r'(y) \leq r'(x)$
  - Podle Pozorování C $-r(y) \leq -r(x)$
  - Pak máme $A = 1 + 2r'(x) - 2r(x)$
  - Ale $2r'(x) - 2r(x) \ \ \leq \ \ 3r'(x) - 3r(x)$, protože nový rank je vždy větší než starý rank
  - Takže máme to co jsme potřebovali.
  - 
  Tím pádem Tvrzení B je dokázané a Hlavní lemma také.

Definujte (a,b)-strom. Popište, jak na něm probíhají operace Find, Insert a Delete. Rozeberte jejich složitost v nejhorším případě.

Vnější vrcholy (nullptr) - pokud chybí child, přidá se prázdný node (na obrázkách jsou to kostičky)
Vícecestný vyhledávací strom - více klíčů ve vrcholu $x_1, \ldots, x_k$, rozděluje se na k+1 podstromů $T_0, \ldots, T_k$, kde pak platí $\forall y \in T_i \ \ x_i < y < x_{i+1}$. Tyto stromy odemykají nové způsoby vyvažování stromů.

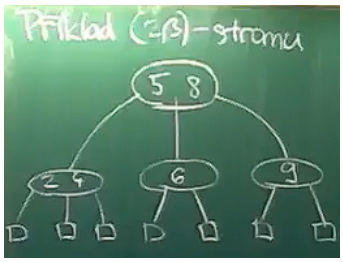
(a, b)-strom

Je to vícecestný vyhledávací strom s vnějšími vrcholy, vhodné nastavit $a \geq 2$, $b \geq 2a - 1$
Axiom 1,

- Každý vnitřní vrchol má *a* až *b* dětí. Kořen může mít 2 až b dětí.

Axiom 2,

- Všechny vnější vrcholy jsou stejně hluboko.



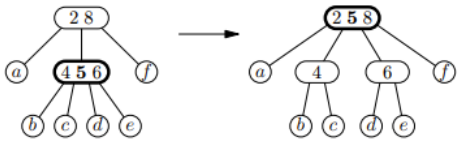
Lemma A: Hloubka (a, b)-stromu s n klíči je $O(log_a(n))$, $\Omega(log_b(n))$
// Důkaz Lemma A: Asi nemusíme vědět

Find

- Přijdeme do vrcholu a binárně vyhledáváme
- Pokud je rovný -> našli jsme
- Pokud je větší, tak se zanoříme do podstromu mezi předešlý a aktuální a postup opakujeme
- $O(log_a \ n \ * log \ b)$ (cena hledání * cena prohledání jednoho vrcholu)
- -> $O(\frac{log \ n}{log \ a} \ * log \ b)$ -> $O(log \ n \ * \ \frac{log \ b}{log \ a})$, zde ten druhý člen je O(1) dokud je $b \in poly(a)$
- -> $O(log \ n)$

Insert

- Jdeme co nejhlouběji a klíč přidáme do vrcholu a přidáme mu vnější vrchol jako dítě
- Pokud bude porušen Axiom 1 musíme vrchol štěpit - medián půjde do rodiče a vrchol se rozdělí na dva nové, nyní zkontrolujeme Axiom 1 rodiče a opakujeme



- Figure 3.2: Splitting an overfull node on insertion to a (2, 3)-tree
- $O(log_a\ n * b)$ (počet hladin * nejhorší případ štěpení na každé úrovni, kde je b klíčů)
- Při štěpení musíme mít alespoň tolik vrcholů, aby nově vzniklé vrcholy nebyly menší než a a proto volíme b alespoň 2a - 1

Delete

- Když mažeme, tak klíč nahradíme nejbližším vyšším klíčem (minimum v pravém podstromu)
- Nyní stačí obsloužit situaci po odebrání z nejhlubší vrstvy
- Pokud nám vrchol nepodteče máme hotovo
- Pokud ano, tak se podíváme na sourozence
- Pokud by po sloučení se sourozencem + klíčem, který nás v rodiči odděluje, nevzniklo přetečení uděláme to a máme hotovo
- Jinak si přidáme klíč, co nás v rodiči rozděluje, a místo něho tam dáme min/max ze sourozence (záleží jestli je to pravý (min) nebo levý (max) sourozenec)
- Časově to bude stejné jako insert, protože můžeme v nejhorším případě na každé vrstvě přeorganizovat celý vrchol

BONUS
Volba *a* a *b*

- b chceme co nejmenší 2a nebo 2a - 1
- jelikož časová složitost roste s větším *a*, tak to chceme taky nejmenší (pokud nepracujeme s čtením bloků, kdy načtení jednoho prvku je stejně I/O náročné jako načtení celého bloku)

Formulujte cache-aware a cache-oblivious algoritmy pro transpozici čtvercové matice. Rozeberte jejich časovou složitost a I/O složitost.

U transpozice matice je problém čtení matice po sloupcích, což je velmi neoptimální pro cache.

N ... Velikost matice
N^2 ... Počet prvků v matici
M ... Velikost cache
B ... Velikost bloku cache
d ... Velikost dlaždice

Cache-aware

Rozřežeme matice na dlaždice o velikosti $d * d$ (u pravé a dolní hrany nebudou přesně $d * d$), tak aby se nám vždy dvojice dlaždic vždy vešla do cache. Jelikož známe parametry cache můžeme vhodně zvolit $d = B$. S tím, že předpokládáme, že se do cache vejdou alespoň 2 bloky.
Počet dlaždic bude $= \lceil \frac{N}{d} \rceil^2 \ \leq \ (\frac{N}{d} + 1)^2 \ \in \ O(\frac{N^2}{d^2} + 1))$.

Dlaždice na diagonále pouze transponujeme a dlaždice mimo diagonálu transponujeme a prohodíme. Můžou nám nastat dva případy:
1, N je dělitelné B

- d = B a tím pádem počet prvků v dlaždici = B^2
- chceme aby $2B^2 \leq M$
- Obecně požadujeme štíhlou cache: $M \geq c * B^2$ (má alespoň tolik bloků kolik je položek v jednom bloku)
- Počet dlaždic zde bude bez +1, protože N je dělitelné B -> počet dlaždic = $\frac{N^2}{B^2}$
- Počet IO na jednu dlaždici je přesně B
- Počet I/O: $O(\frac{N^2}{B^2} * B) \ = \ O(\frac{N^2}{B})$

2, N není dělitelné B

- d = B i když mohou bloky přesahovat mimo dlaždici, maximálně na jeden řádek budeme potřebovat 2 bloky, tím pádem počet bloků v dlaždici $\leq 2B$
- chceme $M \geq 4B^2$
- počet dlaždic = $\frac{N^2}{B^2} + 1$

- I/O celkem: $O((\frac{N^2}{B^2}+1)*B) \;=\; O(\frac{N^2}{B}+B)$, zde "+B" nastane pouze pokud $N^2 < B$, tím pádem se nám celá matice vejde do jednoho bloku, ale potom už máme všechno v cache za cenu jednoho načtení -> $O(\frac{N^2}{B}+1)$.

Časová složitost zůstává stejná, musíme dělat operaci nad každý prvkem 2x -> $O(N^2)$

Cache-oblivious

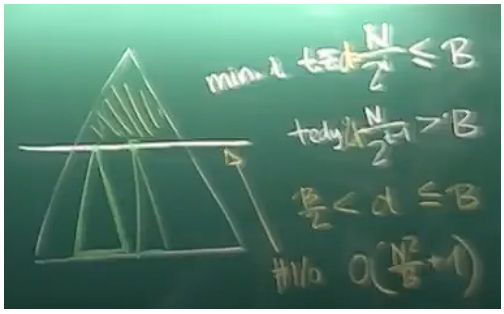
Styl "Rozděl a Panuj", rekurzivně dělíme matici na 4 podmatice o velikosti N/2. Na 2 podmatice na hlavní diagonále zavoláme znovu **Transpose (T)**, na jednu z ostatních dvou podmatic zavoláme **Transpose and swap (TS)**. Prvky se prohazují až na poslední hladině rekurze, když už jsou matice dostatečně malé.

Rozklad T -> 2T + TS
Rozklad TS -> 4TS

Časová složitost pomocí stromu rekurze

- každý vrchol má 3 nebo 4 děti
- na i-té hladině maximálně 4^i vrcholů a velikost $\frac{N}{2^i}$
- tím pádem počet hladin stromu je log N
- počet listů je $\leq 4^{log\,N} = N^2$ a zpracování jednoho listu je v konstantním čase -> $\theta(N^2)$
- počet vnitřních vrcholů $< počet\ listů \;=\; N^2$ -> $O(N^2)$
- Celkově tedy zpracování všech vrcholů stromu rekurze bude $\theta(N^2)$, což je optimální a stejné jako triviální algoritmus
- Jednoduše: I v rekurzivním přístupu uděláme pouze konstantní počet operací na prvek, kterých je N^2 -> $O(N^2)$

I/O složitost pomocí stromu rekurze

- Ve stromu rekurze se zaměříme na úroveň minimálního i takového, že $\frac{N}{2^i} \leq B$ (*i* takové, že se nám celý podstrom do cache už vejde) a díky tomu víme, že $\frac{N}{2^{i-1}} > B$ (*i* o jedna menší se ještě do cache nevejde)
- Tím můžeme $d = \frac{N}{2^i}$ a když vynásobíme 2, tak $2d = \frac{N}{2^{i-1}}$ a podle bodu výše můžeme určit, že následující nerovnost platí: $\frac{B}{2} < d \leq B$
- Tím pádem nám stejně jako u cache-aware analýzy platí $d \in \theta(B)$, takže až na nějakou multiplikativní konstantu je rozklad na dlaždice optimální
- Můžeme tedy navrhnout vhodnou cacheovací strategii, která sleduje, co algoritmus dělá a kdykoliv se dostane na i-tou hladinu, tak načteme do cache buď 1 (T) nebo 2 (TS) matice a už pracujeme jenom v cache
- Tímto způsobem můžeme vlastně jaksi simulovat cache-aware algoritmus a na i-té hladině je I/O složitost $O(\frac{N^2}{B}+1)$ (stejně jako v cache-aware)
- Na hladinách vyšších nemusíme dělat žádné I/O operace, protože s prvky operujeme pouze na nejnižší hladině rekurzivního stromu
- 

Definujte c-univerzální a k-nezávislé rodiny hešovacích funkcí. Formulujte a dokažte větu o střední délce řetězce v hešování s řetězci. Ukažte příklady c-univerzálních a k-nezávislých rodin pro hešování přirozených čísel.

Hešování přiřazuje prvky z univerza (obecný předpoklad čísla, $\{0, U-1\}$ = [U]) do množiny "přihrádek" B = [m], pomocí hešovací funkce h. Velikost podmnožiny univerza, jejíž prvky chceme v přihrádkách reprezentovat (X) značíme n.

Je vhodné vždy vybrat náhodnou hešovací funkci z rodiny.

c-univerzální

Systém funkcí H z U do [m] je c-univerzální pro c > 0 právě když:
$\forall\ x,\ y \in U,\ x \neq y:\ Pr_{h \in H}\ [h(x) = h(y)] \leq \frac{c}{m}$

Věta o střední délce řetězce ze zadání:
Lemma A: Nechť H je c-univ. systém funkcí z U do [m], $X = \{x_1, \ldots, x_n\} \subseteq U$ a $y \in U \backslash X$. Pak $\mathbb{E}_{h \in H}\ \ [počet\ i:\ h(x_i) = h(y)] \leq \frac{cn}{m}$ (strop střední hodnoty počtu prvků v přihrádce, kde hledáme y)

Důkaz Lemma A:

- Indikátory A_i = 1 pokud $h(x_i) = h(y)$, jinak 0
- $\mathbb{E}[A_i] = 0 * Pr[A_i = 0] + 1 * Pr[h(x_i) = h(y)]$ a jelikož je systém nejvýše c-univerzální tak ta pravděpodobnost je $\leq c/m$
- Celkový počet kolizí $A = \sum_i A_i$, z toho uděláme střední hodnotu, ale podle linearity střední hodnoty víme, že střední hodnota sumy je rovna sumě středních hodnot
- A jelikož indikátorů je nejvýše *n* , tak můžeme udělat horní odhad z $A \leq \frac{cn}{m}$.

Složitost nepovedeného Find - $\frac{cn}{m}$ viz. Lemma A
Složitost povedeného Insertu - shora omezené ↑
Složitost povedeného Find - shora omezené ↑
Složitost neúspěšného Insert - stejné jako ↑
Složitost delete = find
=> Složitost operací je $O(\frac{n}{m})$

k-nezávislé

Systém funkcí H z U do [m] je (k, c)-nezávislý ($k \geq 1$ a c > 0), právě když:
$\forall\, x_1, \dots x_k \in U$ navzájem různé
$\forall\, i_1, \dots i_k \in [m]$
platí že: $Pr_{h \in H}[h(x_1) = i_1 \,\&\, \dots \,\&\, h(x_k) = i_k] \leq \frac{c}{m^k}$

Jednoduše: Funkce nepreferuje některé přihrádky pro některé prvky (0 vždy do 0 atd.)

Příklad: U = [p], prvočíslo
$h_{a,h}(x) = ((ax + b)\ mod\ p)\ mod\ m$
Je 2-univ. a (2, 4)-nez.

Systém $\mathbb{L}$ (lineární, příklad 2-univ. a (2, 4)-nezávislé rodiny)

Pro p prvočíslo a $m < p$ definujme $\mathbb{L}$ systém z [p] -> [m], kde $\mathbb{L} := \{h_{a,b} / a, b \in [p]\}$ a $h_{a,b}(x) = ((ax + b)\ mod\ p)\ mod\ m)$

Věta A: Systém $\mathbb{L}$ je 2-univerzální

Důkaz věty A:
Uvažme $h(\_)$ bez $mod\ m$ pro 2 různé hodnoty x a y.
$ax + b \equiv r \pmod p$
$ay + b \equiv s \pmod p$
Pak $\exists\ bijekce\ mezi\ (a, b) \in [p]^2$ a $mezi\ (r, s) \in [p]^2$
-> BÚNO vybíráme rovnoměrně náhodně místo (a, b) dvojici (r, s), takže stačí spočítat:
$Pr_{r,s}[r \equiv s \pmod m]$ - ptáme se na pravděpodobnost, že je pár špatný (kolize)
Počet špatných párů pro pevné r

Představme si číselnou osu od 0 do p a rozdělme si je na $m - tice$ (m < p), r má stejné místo v každé $m - tici$, toto jsou ty místa do kterých se nechceme trefit s
$m - tic$ je $\lceil \frac{p}{m} \rceil$ a to je shora odhadnuto $\leq \frac{p+m-1}{m}$ a $(m - 1)$ je nevýše p, takže to celé lze shora odhadnout $\leq \frac{2p}{m}$
No a teď počet všech špatných párů je nejvýše $p - krát$ tolik (protože r může mít p různých hodnot) -> $\leq \frac{2p^2}{m}$
Takže $Pr_{r,s}[r \equiv s \pmod m] \leq \frac{2p^2}{m} / p^2 = \frac{2}{m}$ (počet špatných párů děleno počet všech párů p^2)
A to je definice 2-univerzality.

Je však možné dokázat i 1-univerzalitu. Všimněme si, když nám a vyjde 0, tak je funkce konstantní.
Definujme L', kde přidáme pravidlo, že $a \neq 0$. Tím se nám v důkazu přidá podmínka, že $r \neq s$ a po updatu výpočtu nám vyjde 1-univerzalita.

Věta B: Systém $\mathbb{L}$ je (2, 4)-nezávislý (Důkaz je v malé otázce)

Lemma M: Nechť H je (2, c)-nezávislý systém z U do [r] a $r > m$
Pak $H\ mod\ m := \{h\ mod\ m | h \in H\}$ je 2c-univerzální a (2, 4c)-nezávislý systém z H do $[m]$.

Důkaz Lemma M: Snad není potřeba

$\mathbb{L}_0\ (ax + b)\ mod\ p$ ... (2, 1)-nez. z [p] do [p]
$\mathbb{L}_0\ mod\ m$ ... 2-univerzální, (2, 4)-nez. a pro $p \geq 4m$ dokonce (2, 2)-nez.

Multiply-shift hešování - Systém $\mathbb{M}$

Máme 32-bit číslo x, náhodné liché číslo a, čísla vynásobíme a výsledek může mít až 64 bitů. Horních 32 bitů zahodíme a zbyde nám zase 32 bitové číslo, ze kterého si vezmeme horních $l - bitů$ s tím, že naše hešovací tabulka má velikost 2^l.

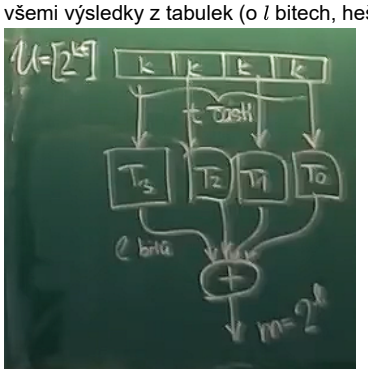
Zaveďme si notaci $X < i : j >$ jako bitové indexování (budou tam bity: $i, i + 1, \dots, j - 1$)

Systém $\mathbb{M}$
Pro w a $l \leq w$ definujeme: (w je počet bitů čísla)
Systém $\mathbb{M} := \{h_a |\ a \in [2^w]\ liché\}$, kde $h_a(x) := (ax) < w - l : w >$
je systém z $[2^w]$ do $[2^l]$

Systém $\mathbb{M}$ je 2-univerzální

Tabelační hešování - Systém $\mathbb{T}$?

Hešované číslo si rozdělíme na t částí po k bitech. Každou částí budeme indexovat v příslušné tabulce (kde máme připravené náhodné hodnoty) a se všemi výsledky z tabulek (o l bitech, hešovací tabulka bude o velikosti $m = 2^l$) uděláme XOR.



Výběr funkce nám trvá $\Theta(t * 2^k)$ (vygenerování tabulek)
Vyhodnocení nám trvá $\Theta(t)$

Tabelace pro aspoň $t \geq 2$ je 3-nezávislá, ale není 4-nezávislá (i když projevuje některé jeho vlastnosti).

POZOR existují 2 druhy polynomiálního hešování jedno je ze $\mathbb{Z}_p$ do $\mathbb{Z}_p$ a druhé z ze $\mathbb{Z}_p^k$ do $\mathbb{Z}_p$

Polynomiální hešování - Systém $\mathbb{P}_k$ ze $\mathbb{Z}_p$ do $\mathbb{Z}_p$

Systém $\mathbb{P}_k := \{h_{t^\rightarrow} \mid t^\rightarrow \in \mathbb{Z}_p^k\}$ (vezmeme náhodný polynom s k koeficienty)
Vyhodnocení $h_{t^\rightarrow}(x) := \sum_{i=0}^{k-1}\ t_i * x^i$

Věta C: Systém $\mathbb{P}_k$ je (k, 1)-nezávislý
Důkaz věty C: Malá otázka

Polynomiální hešování - Systém $\mathbb{R}$ ze $\mathbb{Z}_p^k$ do $\mathbb{Z}_p$ (Rolling hashes)

Systém $\mathbb{R} := \{h_a |\ a \in \mathbb{Z}_p\}$, kde $h_a(x^\rightarrow) := \sum_{i=0}^{d-1}\ x_{i+1} * a^i$

2-nezávislost nelze, protože to hešuje 0 do 0

Věta A: Systém $\mathbb{R}$ je d-univerzální s volbou v čase $\Theta(1)$ a vyhodnocením v čase $\Theta(d)$ (d je dimenze)

Důkaz věty A:
$Pr_a\ [h_a(x^\rightarrow) = h_a(y^\rightarrow)]$ pst. (pravděpodobnost) je rovna $\sum_{i=0}^{d-1}\ x_{i+1} * a^i = \sum_{i=0}^{d-1}\ y_{i+1} * a^i$. To se aritmetickými operacemi dá převést na $\sum_{i=0}^{d-1}\ (x^\rightarrow - y^\rightarrow)_{i+1} * a^i$, z čehož nám vyplývá, že se ptáme na pst., že a je kořenem nějakého nenulového polynomu ($x^\rightarrow - y^\rightarrow$) stupně $\leq d$ a tento polynom má nejvýše d kořenů, takže celá pravděpodobnost bude $\leq \frac{d}{p}$

Popište a analyzujte hešování s lineárním přidáváním.

Otevřená adresace

Otevřená adresace znamená, že na ukládání prvků nám slouží obyčejné pole a nemůžeme v každé buňce řetězit prvky do spojových seznamů, či jiných struktur. Pokud nám tedy nastane kolize musíme vybrat nějakou další buňku v poli, kam prvek dáme, konkrétně bude přístupová posloupnost:
$f : U\ x\ [m]$ -> $[m]$ a $\forall x\ f(x, \_)$ permutace na buňkách (postupně navštíví všechny)

Insert

- Postupně zkouší $f(x, 0), f(x, 1), \ldots$ dokud nenarazí na prázdnou buňku a tam prvek vloží

Find

- Postupně se koukne do $f(x, 0), f(x, 1), \ldots$ dokud nenarazí na prázdnou buňku (neúspěšný find) nebo dokud nenajde hledaný prvek (úspěšný find)

Delete

- Budeme si dělat pomníčky a označovat tak odebrané prvky, když jich bude moc tak to přehešujeme
- Kdybychom prvky odebírali mohli bychom vytvořit díru a pak by find nefungoval

Hešování s lin. přidáváním

Přístupová posloupnost definovaná jako: $f(x,i) := (h(x)+i)\ mod\ m$, kde $h:\ U$ -> [m]
Jednoduše pokud je kolize, tak se podíváme na další buňku a tak pokračujeme dokud nenajdeme volnou buňku. Toto však vytváří shlukování hodnot (clustering) a složitost blíží k lineární pro plné pole, avšak shluky jsou přátelské ke cache.

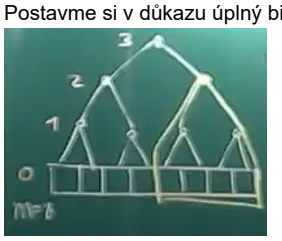
Uvažujme $m \geq (1+\varepsilon) * n$ (m je velikost tabulky, n počet prvků a ϵ nějaký násobek, kolikrát dáme více paměti než je prvků)
Jaká bude střední hodnota času na Find $\mathbb{E}[\ čas\ na\ Find\ ]$

- Pro úplně náhodnou hešovací funkci to bude $O(1/\varepsilon^2)$
- Pro 5-nezávislý systém to bude $O(1/\varepsilon^{13/6})$
- Existuje (ne každý) 4-nez. systém $\Omega(log\ n)$
- Existuje (ne každý) 2-nez. systém $\Omega(\sqrt{n})$
- Pro Mult.-Shift $\Omega(log\ n)$
- Pro Tabelační hešování $O(1/\varepsilon^2)$

Dokažme si složitost úplně náhodné hešovací funkce:
Hlavní Věta A:
Nechť m je mocnina 2, $n \leq m/3$, h je úplně náhodné hešovací funkce z U do [m]
Potom $\mathbb{E}[počet\ buněk\ navštívených\ při\ hledání\ x] \in O(1)$

BÚNO: jelikož s více prvky se struktura zpomaluje, tak uvažujme nejhorší možný případ $n = m/3$ (je to mírný švindl, protože m je mocnina 2, takže není dělitelná 3)

Důkaz hlavní věty A:
Postavme si v důkazu úplný binární strom nad buňkami



Zde si můžeme všimnout, že vnitřní vrcholy nám udávají intervaly (žluté obtažení), říkejme jim bloky. Takže vrchol ve výšce $t \approx$ blok buněk velikosti 2^t se začátkem v násobku 2^t.

Kritický blok - do bloku bylo zahešováno (NE vloženo) alespoň 2/3 jeho velikosti, pro velikost 2^t to je $2/3 * 2^t$.

// vsuvka
Černovova nerovnost pro horní chvost (horní chvost je v rozdělení od nějaké hodnoty dál)

- Nechť $x_1, \ldots, x_k$ jsou nezávislé náhodné veličiny s hodnotami $\{0,1\}$.
- $X := \sum_i\ x_i$
- $\mu := \mathbb{E}[X]$
- c > 1
- Potom

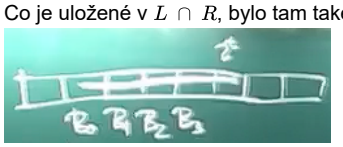
$$Pr\ [X > c * \mu] < (\frac{e^{c-1}}{c^c})^\mu$$

// vsuvka
Lemma B:
Nechť B je blok a $|B| = 2^t$
Pak $Pr[B\ je\ kritický] \leq q^{2^t}$, kde $q = (e/4)^{1/3} \approx 0.873$

// vsuvka
Důkaz lemma B:
Nechť $x_1, \ldots, x_n$ jsou prvky již vložené a $X_1, \ldots, X_n$ jsou indikátory jevů $h(x_i) \in B$ a střední hodnota takového indikátoru je
$\mathbb{E}[X_i] = Pr[jev\ nastal] = |B|/m = 2^t/m$
Také označme $X := \sum_i X_i$ jako počet prvků zahešovaných do B
Dále vytvořme $\mu = \mathbb{E}[X] = \sum_i\ \mathbb{E}[X_i]\ =\ \frac{n*2^t}{m}$ a víme, že n=m/3, takže -> $\frac{2^t}{3}$
Pak $Pr[B\ kritický] = Pr[X > 2 * \frac{2^t}{3}]$ (dvojku jsme přidali, aby platila definice kritičnosti)
Nyní můžeme použít **Černovovu nerovnost** -> $\leq (\frac{e^1}{2^2})^{2^t/3} = (e/4)^{2^t/3}$, což je to co tvrdí lemma.

Definujme běh jako posloupnost prvků od prvního (předním je prázdný) prvku až do posledního (za ním je prázdný) prvku (platí kruhově v poli). Platí, že co je do běhu zahešováno, tak se tam i nachází.

// vsuvka
Lemma C:
Nechť R je běh délky aspoň 2^{t+2} a $B_0, \ldots, B_3$ první 4 bloky velikosti 2^t proťaté R.
Pak alespoň jeden z nich je kritický.

// vsuvka
Důkaz lemma C:
$|B_0 \cap R| \geq 1$ Do B_0 zasáhnul běh alespoň jedním prvkem.
$|B_i \cap R| = 2^t$, pro $i = 1 \ldots 3$ V ostatních blokách to bude jejich velikost protože tam běh nekončí
$L = B_0 \cup \ldots \cup B_3$ Sjednocení prvních čtyř bloků
$|L \cap R| \geq 3 * 2^t + 1$ Plné bloky + 1 prvek v b_0
Co je uložené v $L \cap R$, bylo tam také zahešované.


Kdyby žádný B_i nebyl kritický, bylo by do L zahešováno nejvýše $4 * 2/3 * 2^t$ prvků, což je ale menší než $\leq 3 * 2^t + 1$. SPOR -> jeden z bloků je kritický.

// vsuvka
Lemma D:
Nechť R je běh obsahující h(x).
Pokud $|R| \in [2^{t+2}, 2^{t+3})$
Pak alespoň 1 z těchto 12 bloků velikosti 2^t je kritický: (Za použití Lemma C)

- blok obsahující h(x)
- 8 bloků před (nastane pokud blok s h(x) je poslední v běhu)
- 3 bloky za (nastane pokud blok s h(x) je první v běhu)

$Pr[|R| \in interval] \ \leq \ Pr[\exists\ blok\ z\ 12\ kritický] \ \leq \ 12 * Pr[blok\ velikosti\ 2^t\ je\ kritický] \ \leq \ 12 * q^{2^t}$
Pravděpodobnost délky běhu exponenciálně klesá s délkou běhu.

Nyní máme vše potřebné pro dokázání hlavní věty A:
Označme $|R|$ jako s (R je běh obsahující h(x), délkou běhu zde shora omezíme počet navštívených x z Hlavní Věty A)
$\mathbb{E}[\ s\ ]$ = $\mathbb{E}[\ s \mid s \leq 3] * Pr[s \leq 3] + \sum_{t \geq 0} \mathbb{E}[\ s \mid s \in [2^{t+2}, 2^{t+3}]] * Pr[s \in [2^{t+2}, 2^{t+3})]$ (rozdělení podmíněné pravděpodobnosti/střední hodnoty na bloky 0,1,2 a zbytek)

- $\mathbb{E}[\ s \mid s \in [2^{t+2}, 2^{t+3})]$ odhadneme ze shora jako $\leq 2^{t+3}$
- $Pr[s \in [2^{t+2}, 2^{t+3})]$ odhadneme pomocí Lemma D jako $\leq \ 12 * q^{2^t}$
- $\mathbb{E}[\ s \mid s \leq 3]$ bude maximálně 3
- $Pr[s \leq 3]$ bude maximálně 1
- Suma - $\sum_{t \geq 0} 2^{t+3} * 12 * q^{2^t} \ = \ 12 * 8 * \sum_{t \geq 0} 2^t * q^{2^t} \ \leq \ \sum_i i * q^i$ a pro q menší než jedno (což tady máme), tato suma konverguje ke konstantě, takže celá střední hodnota také konverguje ke konstantě, tím pádem střední hodnota je v $O(1)$ a máme hotovo.

Shrnutí: Dlouhé běhy jsou u úplně náhodné hešovací funkce, tak málo pravděpodobné, že střední hodnota počtu navštívených prvků, pokud je hešovací tabulka maximálně z 1/3 plná, bude konstantní.

Rekapitulace: Přednáška 8 čas 1:19:00

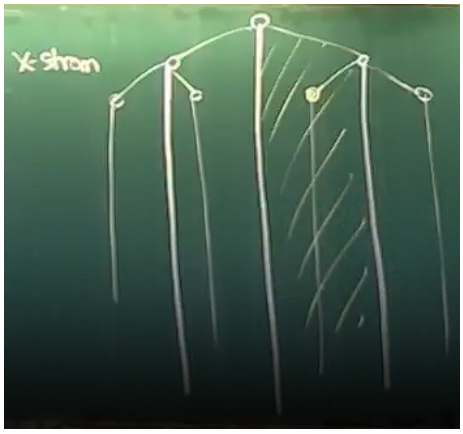
Definujte vícerozměrné intervalové stromy (range trees). Rozeberte časovou a prostorovou složitost konstrukce a obdélníkových dotazů (včetně zrychlení kaskádováním).

Geometrické DS (obecné info pro přehled, není potřeba)

- **objekty** - body v $\mathbb{R}^d$, úsečky, mnohoúhelníky/-stěny, křivky
- **dotazy** - na objekt, interval $[a_1, b_1]\ x\ [a_2, b_2] \ldots$, částečná shoda (např. 1. souřadnice fixní zbytek může být cokoliv), obecné oblasti (mnohoúhelníky, křivky atd.)
- **odpověď** - enumerace (vyjmenování všech objektů splňující dotaz), počet (kolik je objektů splňující dotaz), agregace (vybereme si oblast a na ní použijeme operaci, např. počet obyvatel na dané ploše na mapě)

k-rozměrné intervalové stromy

Začněme v rovině $\mathbb{R}^2$ s unikátními hodnotami pro souřadnice (nelze (2,1) a (2,0) současně).
Vytvořme si x-strom, což je binární strom podle x-ových souřadnic vrcholů a každý vrchol x-stromu má svůj pás určen strukturou x-stromu a také svůj y-strom s body, které padly do pásu.



(pro zvýrazněný bod platí, že řeže osu X na vybarvený pás)
$\forall$ bod náleží do $O(log\ n)$ y-stromů, kterých je stejně jako vrcholů v x-stromě (n), takže prostor bude $O(n\ log\ n)$

Dotaz na $[a_1, b_1]$ x $[a_2, b_2]$

- zeptáme se x-stromu na $[a_1, b_1]$ -> ten nám vrátí
  - vrcholy - vypisuje body pokud $y \in [a_1, b_1]$ (Toto trvá $O(log\ n)$, protože projdeme tolik vrcholů)
  - podstromy -> pásy -> v každém uděláme dotaz na y-strom $[a_2, b_2]$ (Zde máme $O(log\ n)$ pásů, kde na každém uděláme dotaz na y-strom a jeden dotaz na y-stromě trvá $O(log\ n + p)$, kde p je počet prvků, které vypsal)
- Celkově je složitost dotazu $O(log^2\ n + p)$

Build

- Udělejme si 2 seznamy všech bodů, kde jeden bude setříděn podle x B_x a jeden podle y B_y
- 1, m <- je medián x souřadnic ($O(1)$)
- 2, vyrobím vrchol x-stromu z m
  - levý potomek: rekurze na B_x, B_y vlevo od m ($O(n)$)
  - pravý potomek: rekurze na B_x, B_y vpravo od m ($O(n)$)
  - y-strom sestrojený z B_y ($O(n)$)
- Celkově složitost $O(n\ log\ n)$ (třídění je $n\ log\ n$, rekurze je taky $n\ log\ n$, protože pro každý z n vrcholů vytváříme 2-krát menší y-strom)

Kdybychom měli neunikátní hodnoty, tak si vytvoříme ještě jeden y-strom pro každý vrchol, kde budou setříděné všechny body s tímto hraničním x podle y souřadnice (jenom konstanta krát pomalejší, konkrétně 2x).

Obecně Dotaz nám bude trvat $O(log^k\ n)$, prostor bude $O(n * log^{k-1}\ n)$ a build seřazených polí bude taky $O(n * log^{k-1}\ n)$, kde k je rozměr stromu (2D, 3D ...).

Zrychlení Kaskádováním

- Ve statickém případě si můžeme y-stromy reprezentovat jako seřazené posloupnosti bodů
- Každý vrchol má 2 děti a posloupnosti těchto dětí jsou vždy podposloupnosti toho rodiče
- Dotaz v rodiči byl, kam padnou krajní body intervalu, tam vyhledáme jeho konce
- Jenomže stejné hledání bychom museli dělat i v dětech
- Proto si přepočítáme pro každý prvek v horním seznamu (parent), kam dopadne v dolní seznamu (child), pokud by tam nebyl, tak budeme pamatovat předchůdce
- Při vytváření podposloupností si rovnou můžeme předpočítat tyto odkazy a zaberou nám jenom konstanta-krát paměti navíc (pro každý prvek, kterých bude $O(n * log^{k-1}\ n)$ máme max 2 odkazy, neboli čísla, navíc)
- Nyní při hledání už nemusím hledat v nižších y-stromech a jenom se podívám na místo, kde jsou ty hranice (+ můžeme být o jedna vedle, tak si to v konstantním čase zkontrolujeme)
- Dotaz tedy bude mít složitost už jenom $O(n * log^{k-1}\ n)$.

Definujte suffixové pole a LCP pole. Popište a analyzujte algoritmy na jejich konstrukci.

Řetězcové DS

Notace

- Σ ... konečná abeceda
- Σ^* ... množina všech řetězců nad abecedou
- řecké písmenko ... řetězec
- $|\alpha|$ ... délka řetězce
- ϵ ... prázný řetězec (existuje jen jeden)
- $\alpha[0]\dots\alpha[|\alpha|-1]$ ... indexace znaků
- $\alpha[i:j]$ ... znaky $\alpha[i]\alpha[i+1]\dots\alpha[j-1]$
- $\alpha[:j]$ ... prefix
- $\alpha[i:]$ ... suffix
- porovnávání lexikograficky

Úloha: Předpočítat DS pro seno σ, $|\sigma| = S$, která umí najít všechny výskyty jehly δ, $|\delta| = J$

Mějme řetězec "jdou dlouhou rourou", seřaďme si všechny znaky: "d", "h", "j", "l", "o", "r", "u" nyní vytvoříme pole všech suffixů seřazené podle znaků a sekundárně podle délky:

- ϵ
- "dlouhourourou"
- "doudlouhourourou"
- "hourourou"
- atd.

Avšak tuto kvadraticky velkou tabulku lze reprezentovat snadněji. Stačí nám pro každý suffix si pamatovat index, kde začíná v původním řetězci a tomu se říká suffixové pole.

Suffixové pole (S)
Suffixové pole S pro řetězec α délky n. Je permutace na množině $\{0,\dots,n\}$ taková že:
$\forall i:\ \alpha[S[i]] < \alpha[S[i+1]]$ (permutace, která setřídí suffixy podle jejich lexikografického pořadí)

Pojďme toto vybudovat pro předešlou úlohu:
Očíslujme si znaky indexy:

j	**d**	**o**	**u**	**d**	**l**	**o**	**u**	**h**	**o**	**u**	**r**	**o**	**u**	**r**	**o**	**u**	ϵ
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Vybudujme suffixové pole (i s dalšími poli, které se budou vysvětleny později):

Suffix	**i**	**S[i]**	**R[i]**	**L[i]**
ϵ	0	17	4	0
dlouhourourou	1	4	2	1
doudlouhourourou	2	1	7	0
hourourou	3	8	14	0
jdoudlouhourourou	4	0	1	0
louhourourou	5	5	5	0
ou	6	15	8	2
oudlouhourourou	7	2	15	2
ouhourourou	8	6	3	2
ourou	9	12	10	5
ourourou	10	9	17	0
rou	11	14	12	3
rourou	12	11	9	0
u	13	16	16	1
udlouhourourou	14	3	11	1
uhourourou	15	7	6	1
urou	16	13	13	4
urourou	17	10	0	-

Jakmile máme suff. pole snadno najdeme jehlu: Uděláme dvojí binární vyhledávání najdeme nejbližší menší rovný a nejbližší větší suffix. Poté porovnání každého tohoto prvku s jehlou bude trvat délku jehly a každý prvek musíme vypsat. Takže celková složitost bude $O((log\ S) * J + p)$.

Tvrzení A: S jde vybudovat v čase $O(n)$
Důkaz A: Skew algoritmus níže.

Rankové pole (R)
Inverze suffixového pole, pro index v řetězci dostaneme index suffixu
je to permutace na $\{0, \ldots, n\}$ taková že: $\forall i: \ S[R[i]] = i$
Snadno vyplnit pomocí suffixového pole v $O(n)$ (například pro $i = 0$ je $S[0] = 17$, takže $R[17] = 0$)

LCP pole (longest common prefix) (L)
Pro 2 řetězce nám říká kolik znaků od začátku se shoduje.
Hodnota $LCP(\alpha, \beta) := max\{k \mid \alpha[: k] = \beta[: k]\}$ (k největší takové, že se k znaků od začátku u obou řetězců shoduje)
LCP pole: $L[0, \ldots, n-1]$: $L[i] := LCP(\alpha[S[i]], \alpha[S[i+1]])$
Také lze vybudovat v $O(n)$.

Lemma B: $LCP(\alpha[S[i] :], \alpha[S[j] :]) \ = \ min\{L[k] \mid i \leq k \leq j\}$ (Pokud hledáme nejdelší společný prefix pro nějaké dva suffixy, stačí vzít minimum všech $L[k]$ mezi nimi včetně)

Důkaz Lemma B: Jde snadno vymyslet zamyšlením se nad funkcí LCP a LCP pole.

Kasajův algoritmus

Dokáže vybudovat LCP v $O(n)$ pokud má S a R
Pozorování: Všimněme si dvou sousedních suffixů i a $i+1$. Pokud mají různé první znaky, pak jejich LCP je 0. Pokud mají stejné první znaky a pak jejich LCP = 1 + počet stejných znaků suffixů bez prvního písmene (označme i' a j'). Avšak počet stejných znaků i' a j' je podle Lemma B rovno $min\{L[k] \mid i' \leq k \leq j'\}$ nyní podle způsobu výpočtu LCP můžeme shora odhadnout minimum jako $\leq L[i']$ když dosadíme --> $L[i] \leq L[i'] + 1$ a převedeme jedničku $L[i'] \geq L[i] - 1$.

Postup:

- k <- aktuální dolní odhad = 0
- $\alpha[n] =$ \$ (ukončující znak)
- Pro $p = 0 \ldots n-1$: (udávající kde jsme v řetězci)
  - k <- $max(0, k-1)$
  - i <- $R[p]$ (index aktuálního suffixu v suffixovém poli)
  - q <- $S[i+1]$ (pozice v řetězci následujícího suffixu)
  - Dokud $\alpha[p+k] = \alpha[q+k]$:
    - k <- $k+1$
  - $L[i] = k$

(Dobře se to také dá pochopit z ručního postupu dole na stránce: https://www.geeksforgeeks.org/kasais-algorithm-for-construction-of-lcp-array-from-suffix-array/)

Časová složitost

- použijme k jako potenciál (leží vždy mezi 0 a n)
- každý krok vnitřní smyčky zvyšuje k o 1
- každý průchod vnější smyčkou sníží k nejvýše o 1 (za celý běh se sníží maximálně o n)
- a tím pádem zvýšení k může být maximálně $2n$, kdyby to bylo víc, tak by k mohlo být na konci vyšší než n, ale to nejde protože maximální LCP může být jenom n
- a tak časová složitost bude $O(n)$

Sestavení Suffixového pole S zdvojováním

Definujme $\alpha =_k \beta$ pokud se shoduje prvních k znaků (buď k nebo do konce)
Definujme $\alpha \leq_k \beta$
Budeme mít průchody pro $k = 2^0, 2^1, \ldots, 2^{\lceil log\, n \rceil}$
V každém průchodu budeme počítat

- S_k : pořadí suffixů podle $\leq_k$
- R_k : příslušné rankové pole

$R_k[i] =$ počet $j : \alpha[j :] <_k \alpha[i :]$

k=1

- Třídíme suffixy podle počátečního znaku -> obecný třídící algoritmus v čase $O(n \ log \ n)$

k=2k

- definujme si $\alpha_i := \alpha[i:]$
- chceme třídit podle následujícího porovnání $\alpha_i \leq_{2k} \alpha_j$ tohle platí když $(\alpha_i <_k \alpha_j) \vee (\alpha_i =_k \alpha_j \ \& \ \alpha_{i+k} \leq_k \alpha_{j+k})$
- což je stejné jako $(R_k[i] < R_k[j]) \vee (R_k[i] = R_k[j] \ \& \ R_k[i+k] \leq R[j+k])$ (*)
- (pokud bychom indexovali mimo pole, tak hodnota bude 0)
- zde probíhá třídění v čase $O(n \log n)$ a těchto třídění probíhá $log\ n$ (podle k, kterých je tolik), takže celkově $O(nlog^2\ n)$
- To (*) jde zrychlit, je to ekvivalentní s $(R_k[i], R_k[i+k]) \leq_{lex} (R_k[j], R_k[j+k])$ (lexikografické porovnání dvojic) -> to vede na přihrádkové řazení v čase $O(n \log n)$ během 2 průchodů do n+1 přihrádek
- Tím pádem nám spadne časová složitost na $O(n \log n)$

Sestavení Suffixového pole S pomocí Skew algoritmu (není ve skriptech, ale na přednášce bylo)

1, Přečíslujeme abecedu na $1...n$ (první běh $Sort(n, \sigma)$ pak $O(n)$)
2, Vytvoříme si 3 nové řetězce: (nakrájíme si celý řetězec na trojice a sestavíme z toho novou abecedu)

- $\forall i : \alpha_0[i]$ <- <$\alpha[3i], \alpha[3i+1], \alpha[3i+2]$>
- $\forall i : \alpha_1[i]$ <- <$\alpha[3i+1], \alpha[3i+2], \alpha[3i+3]$>
- $\forall i : \alpha_2[i]$ <- <$\alpha[3i+2], \alpha[3i+3], \alpha[3i+4]$>
- Slova délky $n/3$ nad abecedou $[n]^3$
- $O(n)$
  3, Rekurze na $\alpha_{01} := \alpha_0\alpha_1$ -> pole S_{01} (rekurzivní problém se zmenšuje o 1/3)
  4, Spočítáme R_0, R_1 tak že:
- $R_i[j] =$ kde v S_{01} leží $\alpha_i[j:]$
- $O(n)$
  5, Spočítáme S_2: suffixové pole pro α_2
  6, Slijeme S_{01} s S_2 -> Výsledné S

5, Krok podrobněji

- potřebujeme zjistit jestli $\alpha_2[i:] \leq \alpha_2[j:]$ to je v původní alpha
- $\alpha[3i+2:] \leq \alpha[3j+2:]$ to může rozepsat jako $\alpha[3i+2]\alpha[3i+3:] \leq \alpha[3j+2]\alpha[3j+3:]$
  - Potom $\alpha[3i+3:] = \alpha_0[i+1:] = R_0[i+1]$
  - Potom $\alpha[3j+3:] = \alpha_0[j+1:] = R_0[j+1]$
  - Pak stačí jenom porovnat $(\alpha[3i+2], R_0[i+1]) \leq (\alpha[3j+2], R_0[j+1])$
  - Teď stejně jako v předešlém algoritmu toto lze třídit přihrádkově v $O(n)$

6, Krok podrobněji

- Buď se se nám potká suffix $\alpha_0[i:] \leq \alpha_2[j:]$
  - a to je stejné jako porovnat $\alpha[3i:] \leq \alpha[3j+2:]$
  - odtrhneme jeden znak $\alpha[3i]\alpha[3i+1:] \leq \alpha[3j+2]\alpha[3j+3:]$
    - $\alpha[3i+1:] = \alpha_1[i:] = R_1[i]$
    - $\alpha[3j+3:] = \alpha_0[j+1:] = R_0[j+1]$
  - Takže stačí porovnat $(\alpha[3i], R_1[i]) \leq_{lex} (\alpha[3j+2], R_0[j+1])$
- Nebo $\alpha_1[i:] \leq \alpha_2[j:]$
  - a to je stejné jako porovnat $\alpha[3i+1:] \leq \alpha[3j+2:]$
  - odtrhneme jeden znak $\alpha[3i+1]\alpha[3i+2]\alpha[3i+3:] \leq \alpha[3j+2]\alpha[3j+3]\alpha[3j+4:]$
    - $\alpha[3i+3:] = \alpha_0[i+1:] = R_0[i+1]$
    - $\alpha[3j+4:] = \alpha_1[j+1:] = R_1[j+1]$
  - Takže stačí porovnat $(\alpha[3i+1], \alpha[3i+2], R_0[i+1]) \leq_{lex} (\alpha[3j+2]\alpha[3j+3], R_1[j+1])$
- Ve zkratce: Cokoliv se nám v tom porovnávacím algoritmu potká, tak umíme setřídit v konstantním čase pomocí Rankových polí.
- $O(n)$

Složitost

- T(n) = T(2/3*n) + $\Theta(n)$ -> T(n) = $\Theta(n)$ -> celkem $\Theta(n + Sort(n, \sigma))$

MALÉ OTÁZKY

Popište „nafukovací pole“ se zvětšováním a zmenšováním. Analyzujte jeho amortizovanou složitost.

Zvětšení

Na začátku kapacita C -> postupně přidáváme prvky, aktuální počet prvků značen N
Jakmile naplníme kapacitu => $C = N$, tak pole rozšíříme na C' a musíme všechny prvky přesunou, což bude mít $O(n)$ složitost
Dobře funguje $C' = 2C$

Tvrzení A: Zvolením $C' = 2C$ bude amortizovaně trvat insert $O(1)$.

Důkaz Tvrzení A:
C bude na počátku 1 -> 2 -> 4 -> 8 -> … -> 2^k, provede se *k* (re)alokací
Postupně složitost realokací je geometrická řada $\Theta(2^0 + 2^1 + \ldots + 2^k)$
Po poslední realokaci víme že: $2^{k-1} \leq n \leq 2^k$, vezměme si pouze levou nerovnost a vynásobme 2
=> $2^k \leq 2n$, nyní to má společný člen s pravou nerovností -> $n \leq 2^k < 2n$
Nyní geometrická řada realokací má součet $2^{k+1} + 1$ to je $\Theta(2^k)$ a jelikož víme, že $2^k < 2n$, tak víme, že 2^k je $\Theta(n)$.
Takže pokud postupně vložíme N prvků, tak všechny realokace stojí $\Theta(n)$ času, takže to vychází jako kdyby každý prvek přispěl konstantně.
AGREGAČNÍ METODA

Zmenšení

Podobný nápad, zmenšovat 2krát C -> C/2
Problém je kdy zmenšit. Naivně bychom mohli zmenšit, když $C = 2n$, to bychom ale mohli pomocí posloupnosti operací Delete (zmenšení), Insert, Insert (zvětšení), Delete pořád měnit velikost pole a už by to ani amortizovaně nevycházelo.
-> Je vhodné zvolit hranici než $n < c/4$ (& n > 0).

Důkaz, že to funguje:
Rozdělme historii operací datové struktury na epochy, které končí při realokaci.
U poslední epochy to je jednoduché, tam žádná realokace nenastala.
U první epochy bude $\Theta(1)$, protože začínáme s konstantní kapacitou. Nyní ostatní epochy.
Epochy vždy začíná situací, že $N = C/2 \pm 1$, na konci epochy je:

- Roztažení - $N = C$, za epochu muselo přibýt alespoň C/2 prvků, realokace nás bude stát $\Theta(C)$ a udělali jsme C/2 operací, cenu realokace rozdělíme mezi jednotlivé operace (přidám jim konstantu)
- Smrštění - $N = C/4$, za epochu se muselo smazat alespoň C/4 prvků, zde znovu přeúčtujeme cenu realokace všem operacím (oboje je lineární v závislosti na C)

Popište vyhledávací stromy s líným vyvažováním (BB[α] stromy). Analyzujte jejich amortizovanou složitost.

T(v) - podstrom z vrcholu v
s(v) - počet vrcholů v podstromu z vrcholu v
l(v) - levý podstrom vrcholu v
r(v) - pravý podstrom vrcholu v
Dokonale vyvážený strom - $\forall v, |s(l(v)) - s(r(v))| \leq 1$ (velikost pravého a levého podstromu je rozdílná maximálně o jedna), lze jakýkoliv strom dokonale vyvážit v $\Theta(n)$

Líně vyvážené stromy (BB[α] stromy)

α - konstanta, pro nás zvolená jako 2/3
Definice - Vrchol *v* je líně vyvážený (dále jen vyvážený) když: $\forall\ p\ dítě\ v:\ s(p) \leq \frac{2}{3}s(v)$. Strom je vyvážený právě když je každý jeho vrchol vyvážený.

// Asi není potřeba
Lemma A: Vyvážený strom s *n* vrcholy má hloubku $O(log\ n)$

// Asi není potřeba
Důkaz Lemma A: Uvažme cestu z kořene do nejhlubšího vrcholu $v_0, \ldots, v_k$, kde *k* je hloubka stromu. Pak víme: $s(v_0) = n$, $s(v_{i+1}) \leq s(v_i) * \frac{2}{3}$ (podle definice BB[α])
=>pak indukcí: $s(v_i) \leq n\ *\ (\frac{2}{3})^i$ (místo $s(v_i)$, použijeme velikost $s(v_0)$ krát počet násobení 2/3)
=> jakmile pravá strana klesne pod 1, už nemůže být pod aktuálním (i-tým) vrcholem žádný vrchol
-> ve vzorci: $n\ *\ (\frac{2}{3})^i \leq 1$ -> $n \leq (\frac{3}{2})^i$ -> $i \geq log_{3/2}\ n$, což je $O(log\ n)$.

Insert(x)

- Spustíme Find(x), pokud tam není prvek, tak ho přidáme
- Cestou zpět kontrolujeme, zda je stále strom vyvážený a updatujeme (počítadlo $s(v)$ v každém vrcholu)
- Pokud někde cestou najdeme nevyváženost, tak celý podstrom od tohoto vrcholu zničíme a postavíme ho dokonale vyvážený

Delete(x)

- Stejný jako insert, ale prvek smažeme místo vložení

Amortizovaná analýza
- $\varphi(v) = |s(l(v)) - s(r(v))|$ - příspěvek vrcholu *v* k potenciálu
- Potenciál $\Phi = \sum_v \varphi(v)$, pozorování $\Phi \geq 0$, $\Phi_0 = 0$
- Při Insertu měním každé s(v) nejvýše o 1, takže každé $\varphi(v)$ se maximálně také změní o 1 (2), takže změna celkového potenciálu bude změna potenciálu jednoho vrcholu krát délka cesty (ta je u insertu $O(log\ n)$)
- Takže insert bez vyvážení má reálnou cenu $O(log\ n)$, změnu potenciálu také $O(log\ n)$, tím pádem amortizovaná cena (reálná + změna potenciálu) bude také $O(log\ n)$.
- Teď ještě kolik bude cena vyvážení vrcholu *v*:
- $\exists\ p$ *dítě* $v:\ s(p) > s(v) * 2/3$
- $\exists\ p'$ *dítě* $v:\ s(p') < s(v) * 1/3$
- Z toho víme, že $\varphi(v) \geq 1/3 * s(v)$, po vyvážení bude $\varphi(v) \leq 1$ tím pádem nám potenciál lineárně klesl s velikostí podstromu, který přestavujeme, ale čas na přestavení je také lineární, takže přestavění se nám zaplatí ze změny potenciálu
- !! POKUD $\varphi(v) = 1$, PAK JE ROVEN 0 (Edge case, protože příspěvek k potenciálu v dokonale vyváženém stromě může být 1 a pak by nás mohlo vyvažování stát více potenciálu než máme)

Navrhněte operace Find, Insert a Delete na Splay stromu. Analyzujte jejich amortizovanou složitost.

T(v) ... Podstrom z vrcholu *v*
s(v) ... Počet vrcholů v T(v) $1 \leq s(v) \leq n$
r(v) ... Rank $log_2\ s(v)$ (dá se pochopit jako výška *v*) $0 \leq r(v) \leq log\ n$
Φ ... Potenciál definován jako $\sum_v\ r(v)\ 0 \leq \Phi \leq n\ log\ n$

Find(x)

- Provedeme klasický Find(x), jako v normálním binárním stromě
- Jakmile (najdeme prvek x)/(dojdeme nakonec do vrcholu y) tak uděláme Splay(x)/Splay(y)
- Amort. analýza - Pokud nám ve Splay operaci amortizace zaplatila projití té cesty, tak projít ji dvakrát (find + splay) nás bude stát amortizovaně stejně (naúčtujeme to splay) -> $O(log\ n)$

Insert(x)

- Find(x) - pokud ano, tak splay(x) a hotovo, když ne tak vložíme a uděláme splay(x)
- Amort. analýza - Vložením se změní rank všech vrcholů od vloženého až do kořene, takže to nebude, tak jednoduché jako u Find(x)
  - Označme vrcholy od nově vloženého $v_0, \ldots, v_k$, kde v_k je kořen
  - Změna potenciálu je $\Delta\Phi = \sum_{i=1}^{k}\ (r'_i - r_i)$, kterou musíme dokázat, že je malá
  - $s'_i = s_i + 1$, což lze shora odhadnout s_{i+1} , takže $s'_i \leq s_{i+1}$
  - tím pádem $r'_i \leq r_{i+1}$, a tak sumu shora odhadneme a r' nahradíme
  - tam se nám teleskopicky prvky sežerou a zbyde nám tam jenom $r'_k - r_1$, ale víme že r_1 je nezáporné, takže je to celé $\leq r'_k$, což je rank a ten je logaritmický ($\in O(log\ n)$), takže amortizovaně je změna potenciálu maximálně logaritmická a to se nám schová do $O(log\ n)$ zbytku insertu
- Složitost $O(log\ n)$

Delete(x)

- Find(x), pokud ne, tak splay(poslední nalezený vrchol) a chová se to stejně jako Find
- Pokud jsme našli, tak vrchol smažeme
  - List - prostě smažeme a potenciál snižujeme, takže dobře pro nás Splay(parent)
  - Jeden potomek - smažeme a napojíme potomka na parent a potenciál se zase sníží a uděláme Splay(parent)
  - Dva potomci - Najdeme minimum v pravém podstromě a vrchol vložíme místo mazaného vrcholu, z místa kde jsme minimum vzali, tak uděláme Splay(jeho parent)
- Složitost $O(log\ n)$

Vyslovte a dokažte věty o amortizované složitosti operací Insert a Delete na (a,2a-1)-stromech a (a,2a)-stromech.

První věta, která pracuje jenom s insertem:
Věta A:
Posloupnost m insertů na zpočátku prázdném (a, b)-stromu změní $O(m)$ vrcholů. (Neboli amortizovaná cena jednoho insertu je konstantní.)

Důkaz věty A:
Počet štěpení je $\leq$ počet vrcholů na konci $\leq m$

Důsledek věty A:
(a, b)-strom vkládáním rostoucí posloupnosti $x_1 < \ldots < x_n$ stavíme v $O(n)$ (Pokud si pamatujeme pointer na vrchol s max)

Věta B:
Posloupností *m* Insert a Delete na zpočátku prázdném (a, 2a)-stromu změní $O(m)$ vrcholů.

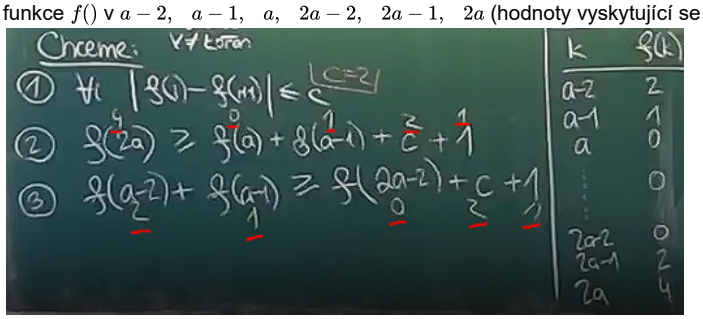
Toto neplatí u (a, 2a-1)-stromů, kde se nám může stát, že špatnou posloupností Insert a Delete budeme pořád slučovat a štěpit vrcholy dokola.

Důkaz věty B:
$f(k) =$ příspěvek vrcholu s k klíči ($k \in \{a-2, 2a\}$)
Reálná cena bude počet změněných vrcholů
Celkový potenciál $\Phi = \sum_{v \text{ vnitřní vrchol}} f(\text{počet klíčů ve } v)$, (počítáme vnitřní prvky bez kořene) je vždy nezáporný a začíná na 0

Chceme funkci $f()$ takovou aby:

- 1, Přidáním nebo odebráním prvku změní hodnotu f jenom o konstantu $\forall\, i\, |f(i) - f(i+1)| \;\leq\; c$
- 2, Štěpení zadarmo $f(2a) \;\geq\; f(a) + f(a-1) + c + 1$ - Rušení vrcholu s $2a$ klíči uvolníme $f(2a)$ potenciálu a chceme z něho zaplatit:
  - $f(a)$ - první nový vrchol s a prvky
  - $f(a-1)$ - druhý nový vrchol s $a-1$ prvky
  - c - přibyl prvek v rodiči
  - 1 - reálná cena operace
- 3, Slučování $f(a-2) + f(a-1) \;\geq\; f(2a-2) + c + 1$ - Rušení dvou vrcholů s $a-2$ a $a-1$ klíči uvolníme potenciál a chceme z něho zaplatit:
  - $f(2a-2)$ - vytvoření nového vrcholu s klíči ze zrušených vrcholů a klíče, který je v rodiči odděloval $a-2 \;\; + \;\; a-1 \;\; + \;\; 1$
  - c - ubyl prvek v rodiči
  - 1 - reálná cena operace

Zajímají nás hlavně hodnoty funkce $f()$ v $a-2, \quad a-1, \quad a, \quad 2a-2, \quad 2a-1, \quad 2a$ (hodnoty vyskytující se v našich požadavcích). Funkci lze vyhodnotit takto, aby nám to vycházelo:



(Červeně podtržené hodnoty jsou hodnoty po dosazení, c=2 protože nejvyšší skok o 2 je mezi funkčními hodnotami $2a-1$ a $2a$)

Analyzujte k-cestný Mergesort v cache-aware modelu. Jaká je optimální volba k?

k-cestný Mergesort neslévá 2 pole najednou, ale k polí

Analýza slévání *k* běhů

- Pořídíme si haldu, kde si budeme pamatovat vždy první prvek z *k* běhů
- Díky tomu budeme hned vědět ve kterém běhu se nachází aktuálně nejmenší prvek
- Práce s haldou bude mít časovou složitost $O(log\ k)$
- Toto se provede N-krát, takže celkově čas. složˇ. -> $O(N\ log\ k)$
- I/O složitost - Každý běh bude načítat $\frac{N/k}{B} + 1$ a běhů je *k*, takže když toto sečteme k-krát, tak I/O složitost bude $O(N/B\ + k)$

Rozšíření analýzy slévání o třídění *k* běhů

- Počet průchodů $= log_k\ n = \frac{log\ n}{log\ k}$
- Čas na průchod je $O(N\ log\ k)$, protože se každý prvek účastní právě jednoho slévání $O(log\ k)$
- Čas celkově -> $O(N\ log\ k * \frac{log\ N}{log\ k})$ (cena jednoho průchodu * počet průchodů) -> $O(N\ log\ N)$
- Tím pádem k-cestný Mergesort je časově optimální (stejný jako normální) pokud je *k* alespoň 2.
- I/O složitost na jeden průchod $O(N/B + 1)$, kde *k* se nám zmenší na 1, protože pokud potřebujeme jednu I/O navíc, tak v ní načteme další slévání, které hned použijeme, takže nám stačí +1 pro poslední slévání
- I/O celkově -> $O(N/B * \frac{log\ N}{log\ k} + 1 + k/B)$ (cena jednoho průchodu * počet průchodů)
- +1 u ceny jednoho průchodu pouze, když je to celé menší než blok, tím pádem nám stačí 1 I/O operace celkem (proto je +1 za počtem průchodů)

- +k/B za jedno načtení haldy na začátku (děleno B, protože je v paměti souvisle) to se však schová do asymptotiky -> nejvýše to je N/B, což se schová do N/B
- Nyní chceme zvolit vhodné *k*, s rostoucím *k* člen $N/B * \frac{log\ N}{log\ k}$ klesá, takže chceme *k*, co největší. Omezuje nás však velikost cache a potřebujeme, aby platilo $k * B + B + k \leq M$ (blok na každý běh + blok v zápisovém poly + halda), to lze shora odhadnout pomocí $2kB$, takže $2kB \leq M$ -> $k \leq \frac{M}{2B}$ a jelikož chceme *k*, co největší, tak $k = \frac{M}{2B}$ a v výsledná I/O složitost bude $O(N/B * \frac{log\ N}{log\ (M/2B)} + 1)$.

Vyslovte a dokažte Sleatorovu-Tarjanovu větu o kompetitivnosti LRU.

M ... Paměť strategie cache
B ... Blok cache
C ... Cena strategie cache

Kešovací strategie

BÚNO: B=1, cache má M bloků
Vstup je posloupnost požadavků $x_1, \ldots, x_n \in \mathbb{N}$
Výstup je posloupnost míst v cache $y_1, \ldots, y_n \in \{1, \ldots, M\}$

- Cena **C** - určuje jak je strategie dobrá, kolikrát jsme museli platit načtení z vnější paměti

Strategie

- Offline - znají celý vstup předem, je optimální
- Online - zná co bylo a aktuální požadavek, LRU (Least-Recently Used) - zahodí se nejdéle nepoužitý blok

Chceme A: $\exists k : C_{LRU} \leq k * C_{OPT}$ (existuje k tak, že cena LRU je jen o násobek horší než cena optima)

- LRU je *k*-kompetitivní -> TOTO VŠAK NEPLATÍ, vždy jde vymyslet dostatečně špatný vstup, aby LRU muselo platit za každý blok.
- Co když, ale LRU a OPT budou mít různou paměť *mrk* *mrk*

Věta (Sleator-Tarjan): Pro $M_{LRU} > M_{OPT} \geq 1$ a každou posloupnost požadavků:

$$C_{LRU} \leq \frac{M_{LRU}}{M_{LRU} - M_{OPT}} * C_{OPT} + M_{OPT}$$

Aditivní člen na konci je pro počáteční situaci, kdy OPT může mít už něco v cache, ale LRU ne. Zlomek zde slouží jako k-násobek u Chceme A, například dáme-li LRU 2x více paměti, stane se, až na aditivní konstantu, 2-kompetitivní. Toto nijak nemění asymptotiku cache algoritmů, stačí počítat s optimální strategií, která má 2x méně paměti než LRU.

Důkaz věty (Sleator-Tarjan): Mějme posloupnost přístupů $x_1, \ldots, x_n$, které rozdělíme na části, říkejme jim Epochy $E_1, E_2, \ldots, E_k$. Ty rozdělíme, tak aby LRU v každé epoše zaplatilo cenu takovou jaká je jeho paměť =M_{LRU} a zbytek dáme do epochy E_0 (v E_0 zaplatí maximálně M_{LRU}).

Nyní se zaměřme na nějakou nenultou epochu E_i, i > 0, kde nastane jedna z následujících situací:
1, všechny bloky, za něž platí LRU, jsou navzájem různé

- => aspoň M_{LRU} různých bloků
- =>OPT platí aspoň M_{LRU} - M_{OPT}
- Z předešlých dvou řádků vyplývá poměr cen:
- $$\frac{C_{LRU}}{C_{OPT}} \leq \frac{M_{LRU}}{M_{LRU} - M_{OPT}}$$

  2, LRU platí za tentýž blok 2x
- Jelikož jednomu bloku trvá vybublat z LRU, tak dlouho jako je jeho velikost, muselo se mezitím načíst alespoň M_{LRU} různých bloků, tím pádem platí argument z 1,

Posledně Epocha E_0:
1, LRU i OPT začínají s prázdnou cache

- C_{LRU} = počet různých bloků = C_{OPT}, což může být maximálně M_{LRU}, takže oba zaplatí stejně
  2, LRU začíná s neprázdnou cache -> tím lépe pro něj, platí méně
  3, OPT začíná s neprázdnou cache -> větu zachrání aditivní "$+C_{OPT}$" ve vzorečku

Popište systém hešovacích funkcí odvozený ze skalárního součinu. Dokažte, že je to 1-univerzální systém ze Z_p^k do Z_p.

Univerzum $U = \mathbb{Z}_p^d$
Do přihrádek $\mathbb{Z}_p$

Skalární součin systém $\mathbb{S}$
$\varphi := \{h_{a^\rightarrow} \| a^\rightarrow \in \mathbb{Z}_p^d\}$
$h_{a^\rightarrow}(x^\rightarrow) := a^\rightarrow * x^\rightarrow$

Věta: Systém $\mathbb{S}$ je 1-univerzální

Důkaz věty:
Chceme počítat pro $x^\rightarrow \neq y^\rightarrow$ pravděpodobnost kolize:
$Pr_{h\in\varphi}\left[h(x^\rightarrow) = h(y^\rightarrow)\right] \; = \; Pr_{a^\rightarrow \in \mathbb{Z}_p^d}\left[a^\rightarrow x^\rightarrow = a^\rightarrow y^\rightarrow\right]$, což je stejné jako pravděpodobnost, že $a^\rightarrow(x^\rightarrow - y^\rightarrow) = 0$, kde $x^\rightarrow - y^\rightarrow = z^\rightarrow$ rozdílné od 0

Tím pádem $\sum_{i=1}^{d}\; a_i z_i = 0$ a víme, že alespoň jedno z_i je nenulové (BÚNO to bude z_d)
Dále $(\sum_{i=1}^{d-1}\; a_i z_i) \; + \; a_d z_d = 0$, a víme, že postupně vybíráme členy a_i náhodně z tělesa, tak pak existuje právě jedno a_d takové, aby rovnice platila, no a pravděpodobnost, že zvolíme zrovna to a_d je $\frac{1}{p}$ (p je velikost tělesa).

Popište systém lineárních hešovacích funkcí. Dokažte, že je to 2-nezávislý systém ze Z_p do [m].

viz. Velká Otázka na hešování

Věta: Systém L je (2, 4)-nezávislý

Důkaz věty:
Uvažme $h(\_)$ bez $mod\; m$.
$ax + b \equiv r \pmod p$
$ay + b \equiv s \pmod p$
Pak $\exists\; bijekce\; mezi\; (a, b) \in [p]^2$ a $mezi\; (r, s) \in [p]^2$
-> BÚNO vybíráme rovnoměrně náhodně místo (a, b) dvojici (r, s), takže stačí spočítat:

$Pr_{r,s}\left[r \; \equiv \; y_1 \;\&\; s \; \equiv \; y_2 \; (mod\; m)\right]$ ($y_1\; a\; y_2$ jsou výsledky hešovací funkce s mod m)
$= \; Pr_r[r \; \equiv \; y_1] * Pr_s[s \; \equiv \; y_2]$ (protože výběr r a s je nezávislý)
Stačí spočítat třeba tu první pravděpodobnost, druhá vyjde stejně

Představme si číselnou osu od 0 do p a rozdělme si je na $m - tice$ (m < p), y_1 má stejné místo v každé $m - tici$, toto jsou ty místa do kterých se nechceme trefit r
$m - tic$ je $\lceil \frac{p}{m} \rceil$ a to je shora odhadnuto $\leq \frac{p+m-1}{m}$ a $(m-1)$ je nevýše p, takže to celé lze shora odhadnout $\leq \frac{2p}{m}$

-> Počet špatných bude $\leq \; \frac{2p}{m}$
-> $Pr[r\; špatné] \; \leq \; \frac{2p}{m}/p \; = \; \frac{2}{m}$ (Počet špatných děleno počet možností)
Takže $\; Pr_r[r \; \equiv \; y_1] * Pr_s[s \; \equiv \; y_2] \; \leq \; \frac{4}{m^2}$, což je přesně definice (2, 4)-nezávislosti.

Sestrojte k-nezávislý systém hešovacích funkcí ze Z_p do [m].

viz. Velká Otázka na hešování:

Polynomiální hešování - Systém $\mathbb{P}_k$ ze $\mathbb{Z}_p$ do $\mathbb{Z}_p$

Systém $\mathbb{P}_k := \{h_{t^\rightarrow} \mid t^\rightarrow \in \mathbb{Z}_p^k\}$ (vezmeme náhodný polynom s k koeficienty)
Vyhodnocení $h_{t^\rightarrow}(x) := \sum_{i=0}^{k-1}\; t_i * x^i$

Věta A: Systém $\mathbb{P}_k$ je (k, 1)-nezávislý
Důkaz věty A:
Dáno $x_1, \ldots, x_k$ různé $\in \mathbb{Z}_p$ a $y_1, \ldots, y_k \in \mathbb{Z}_p$

Podle definice k-nez. hledáme $Pr_{t^\rightarrow}\left[\forall i\; h_{t^\rightarrow}(x_i) = y_i\right]$, ale jakmile máme určené x_i, tak rovnice v pst. platí pro právě jedno $t^\rightarrow$, takže pst. je rovna $\frac{1}{p^k}$ (počet možností jak zvolit $t^\rightarrow$).

Nyní ale musíme sestavit polynomiální hešování a použít na něj $mod\; m$, protože hešujeme do [m] přihrádek. Na to ale potřebujeme větu, jak se chová $mod\; m$ pro (k, c)-nez. systémy.

Lemma o K-nez.:
Nechť H je (k, c)-nez. systém z U do [p] a $r \geq 2km$
Pak $H\; mod\; m := \{h\; mod\; m | h \in H\}$ je (k, 2c)-nez. systém z U do [m].

Díky použití tohoto lemma budeme mít systém funkcí $\mathbb{P}_k\; mod\; m$, který je (k, 2)-nezávislý.

Důkaz lemma o K-nez.:
Mějme $x_1, \ldots, x_k$ z Univerza, pak $y_1, \ldots, y_k$ z tělesa o velikosti r bez $mod\; m$ a $v_1, \ldots, v_k$ z finálního tělesa s $mod\; m$.
Ptáme se podle definice na $Pr[\forall i\; h(x_i)\; mod\; m = v_i] \leq$ (to jde ze shora odhadnout hešováním do mezilehlé množiny)

$$\leq \sum_{(y_1,\ldots,y_k)\in[r^k],\,\forall i\; y_i\equiv v_i} Pr[\forall i\; h(x_i)=y_i]$$

Tato pravděpodobnost je nejvýše (podle def. (k, c)-nez.) $\leq \frac{c}{r^k}$
Teď potřebujeme počet všech k-tic, aby pro ně platila podmínka ze sumy:

- Počet k-tic $\leq \lceil \frac{r}{m} \rceil^k \;\; \leq \;\; (\frac{r+m-1}{m})^k$

Takže celou pravděpodobnost odhadneme jako počet k-tic * pst pro jednu k-tici $\leq \frac{c}{r^k} * (\frac{r+m-1}{m})^k \;\; = \;\; \frac{c}{m^k} * (\frac{r+m-1}{r})^k$, zde shora odhadneme vnitřek závorky jako $\frac{r+m}{r} = 1 + \frac{m}{r}$ a nyní si "vzpomeňme" na nerovnost $e^x \geq 1 + x$ a použijme ji: $\frac{c}{m^k} * e^{mk/r}$ nyní můžu mk odhadnout shora jako $r/2$ pomocí podmínky z lemma pak nám vznikne $\frac{c}{m^k} * e^{1/2}$ a $e^{1/2}$ lze odhadnout shora jako 2 --> $\frac{2c}{m^k}$ ---> (2c, k)-nezávislost.

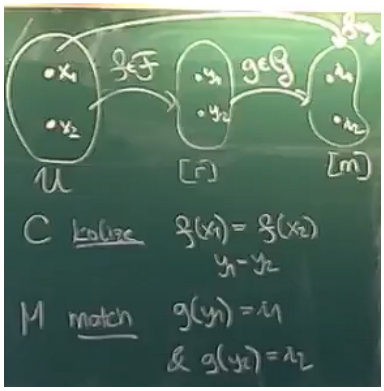
Sestrojte 2-nezávislý systém hešovacích funkcí hešující řetězce délky nejvýše L nad abecedou [a] do [m].

viz. Velká Otázka o hešování

Máme na výběr polynomiální hešování nebo skalární součin. U řetězců bude vhodnější použít poly. hešování, díky nutnosti náhodně vybrat jenom jeden skalár. Potřebujeme však tento výsledek poslat do přihrádek [m] pomocí modulo a na to potřebujeme Lemma:

Sestavme Lemma K o kompozici funkcí:
Nechť $\mathbb{F}$ je c-univerzální systém z U do [r] a $\mathbb{G}$ je (2, d)-nez. systém z [r] do [m] (viz. obrázek)
Pak $H := \mathbb{F} \circ \mathbb{G} = \{f \circ g |\; f \in \mathbb{F}, g \in \mathbb{G}\}$ (skládání je používáme tak, že aplikujeme f a pak g)
je (2, c')-nezávislý pro $c' = (\frac{cm}{r} + 1)d$

Na c' jsme se dostali následujícím způsobem:
Definujme si C kolizi, M match následovně



Chtěli bychom vědět $Pr[M]$, což nevíme protože nezávislost $g()$ nám říká pouze o případu když nenastane kolize. Použijme tedy podmíněnou pravděpodobnost

$Pr[M] = Pr[M|C] * Pr[C] + Pr[M|!C] * Pr[!C]$, tyto pravděpodobnosti můžeme shora odhadnout

- $Pr[C]$ - je pst. kolize hešování funkce f z U do [r] s c-univ., takže $\leq \frac{c}{r}$
- $Pr[M|!C]$ - (2, d)-nez. funkce g nám zase dává zde odhad $\leq \frac{d}{m^2}$
- $Pr[M|C]$ - zde buď:
  - i_1 a i_2 jsou různé indexy a pak je pst. 0, protože nám nemůže hešovací funkce g zahešovat stejný prvek do dvou různých
  - i_1 a i_2 jsou stejné indexy a pak, protože $g()$ je 2-nez., tak je i 1-nez. a to mám omezuje pst, že konkrétní prvek spadne do konkrétní přihrádky a pak pst. je $\leq \frac{d}{m}$
- $P[!C]$ - zde nám stačí odhad pst. ≤ 1
- Teď dáme odhady dohromady, tak aby se dělilo m^2:
  -

$$\frac{cd}{mr} + \frac{d}{m^2} = \frac{\frac{cdm}{r} + d}{m^2} = \frac{d(\frac{cm}{r} + 1)}{m^2}$$

  Takže $Pr[M] \leq \frac{c'}{m^2}$, kde $c' = (\frac{cm}{r} + 1)\, d$

Nyní Lemma K použijeme na systém $\mathbb{R}$ ze $\mathbb{Z}_p^l$ do $\mathbb{Z}_p$, kde dimenze bude délka řetězce (L, pokud by byl kratší, tak tam dáme nulové znaky tzv. padding) a systém lineárního hešování $\mathbb{L}$ ze $\mathbb{Z}_p$ do $[m]$. Zároveň pro lepší univerzalitu předpokládejme $p \geq 4 * L * m$
$\mathbb{R}$ je L-univerzální, takže c=L a poly. hešování posílá prvky do $\mathbb{Z}_p$ takže r=p
$\mathbb{L}$ je (2, 2)-nezávislé, takže d=2
Tedy podle lemmatu: $c' = (\frac{L*m}{p} + 1) * 2$ a podle předpokladu můžeme horní část zlomku odhadnout jako $L * m \leq 4p$, tím pádem c' bude maximálně $(1/4 + 1) * 2$ -> složení je (2, 2.5)-nezávislé => 2-nez.

Popište a analyzujte Bloomův filtr.

Je to reprezentace množiny, která má právo děla jednostranné chyby. Pokud odpoví na přítomnost prvku "Ne" určitě tam není, pokud odpoví "Ano", tak s malou pst to může být false positive (FP).

Příklad bloom filtru: Klasický obrázek Univerzum -> hešovací funkce -> m přihrádek, ale tentokrát to bude m-bitů a ne přihrádek.
Tudíž máme bity B[0] až B[m-1] a při **Insert(x)** nastaví B[h(x)] <- 1.
Při **Find(x)** přečte hodnotu B[h(x)].

Uvažme h z 1-univ. systému.
Vložili jsme prvky $x_1, \ldots, x_n \in U$ navzájem různé
Hledáme $y \in U,\ \forall i : y \neq x_i$

Pro fixní i je $Pr_h\left[h(y) = h(x_i)\right] \ \leq \ \frac{1}{m}$
Pst, že existuje i, které zkoliduje $Pr_h\left[\ \exists i : h(y) = h(x_i)\right] \ \leq \ \frac{n}{m}$ což je pravděpodobnost FP odpovědi (sice jevy nejsou nezávislé, ale lze je shora odhadnout nejhorším součtem pravděpodobností)

Pro dané ε (pst. FP) jak velké má být m, aby $Pr[FP] \leq \varepsilon$. Tedy $n/m \leq \varepsilon$ -> $m \geq \lceil \frac{n}{\varepsilon} \rceil$.
S růstem ε velmi strmě rostou nároky na paměť. Vylepšíme to více filtry.

Vícepásmový Bloomův filtr

k = počet pásem filtru
m = počet přihrádek jednoho pásma
$B_1 \ldots B_k$ k polí m bitů
$h_1 \ldots h_k$ nezávislé hešovací funkce z U do m přihrádek

Insert(x): Pro $i = 1, .., k:\ B_i[h_i(x)]$ <- 1.
Find(x): Jestli pro $i = 1, .., k:\ B_i[h_i(x)]$ je 1 pak tam prvek je, jinak není.

Nastavíme $m = 2n$ pak $Pr[i - t\acute{e}\ p\acute{a}smo\ FP] \leq 1/2$
Pak $Pr[v\check{s}echna\ p\acute{a}sma\ jsou\ FP] \leq 1/2^k$
Pro dané ε -> $k = \lceil log\frac{1}{\varepsilon} \rceil$, paměť $km = 2kn = 2n * log\ \frac{1}{\varepsilon}$ pak potřebujeme mnohem menší paměť pro stejně dobrou přesnost.

Časově ve vícepásmovém filtru bude lineární s k a v jedno pásmovém konstantní.

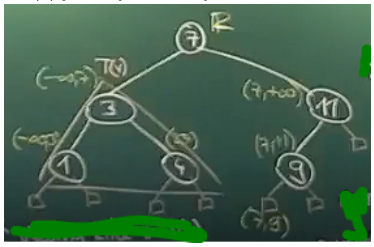
// Nevím jestli bude potřeba i optimalizační úloha na to jak zvolit m pro úplně náhodné hešovací funkce, ale je to $m = 1.44n$, případně je to 9. Přednáška čas 1:11:00

Ukažte, jak provádět 1-rozměrné intervalové dotazy na binárním vyhledávacím stromu.

Uvažujme 2 typy BVS:
Statické

- Na reprezentaci nám stačí setříděné pole, pro usnadnění přidejme -inf a +inf na začátek a konec
- Dostaneme interval [a, b], uděláme Find(a), musíme ošetřit, že a nemusí být v poli, poté snadno jdeme od indexu a až dokud klíč není větší než b
- Build: $O(n * log\ n)$
- Prostor: $O(n)$
- Dotaz: $O(log\ n + p)$, kde p je velikost odpovědi

Dynamické (Range tree)
Na reprezentaci použijeme BVS (vyvážený), nejdříve si ukážeme na statickém BVS a pak zdynamizujeme. Pracujme s variantou, kde máme prázdné externí vrcholy. (kostky na obrázku)
Definujme $int(v) := \{x \in \mathbb{R} \mid p\check{r}i\ hled\acute{a}n\acute{i}\ x\ navšt\acute{i}v\acute{i}me\ vrchol\ v\}$ pro každý vrchol
$int(ko\check{r}en) = \mathbb{R}$
$int(l(v)) = int(v)\ \cap\ (-inf, k(v))$ (k(v) je klíč ve v)
$int(p(v)) = int(v)\ \cap\ (k(v), inf)$ (k(v) je klíč ve v)
$int(v)$ je vždy otevřený interval

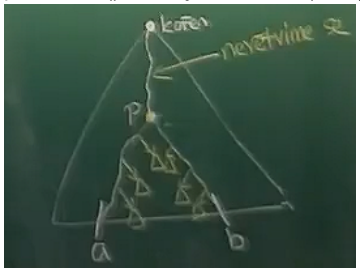


Pro daný vrchol, $int(v)$ odpovídá rozsahu klíčů v jeho podstromu.
Intervalový dotaz pomocí rekurzivního algoritmu RangeQuery(v-vrchol, Q-dotaz):

1. Pokud v je v externí: return
2. Pokud $int(v) \subseteq Q$: vypíšeme celý podstrom T(v) a return
3. Pokud $k(v) \in Q$: vypíšeme v
4. Q_l <- $Q \cap int(l(v))$, Q_p <- $Q \cap int(p(v))$
5. Pokud $Q_l \neq \emptyset$: RangeQuery($l(v), Q_l$)
6. Pokud $Q_p \neq \emptyset$: RangeQuery($p(v), Q_p$)

Lemma A: RangeQuery navštíví $O(log\ n)$ vrcholů a podstromů.

Důkaz Lemma A:
Nechť $Q = [a, b]$.
Uvažme cestu z kořene do a a z kořene do b, tak budou mít nějakého společného předchůdce p (viz. obrázek). Zde při cesta do a má složitost logaritmickou, to samé cesta do b, navíc vrcholy, které navštívíme mezi a a b (žluté podstromy na obrázku), tak těch bude také jenom logaritmicky. Tím pádem v O() notaci je složitost $O(log\ n)$.



Složitost

- dotaz $O(log\ n + p)$ (p je počet vypsaných vrcholů, $log\ n$ bude za cesty do a a b, p bude za vypsání podstromů mezi a a b)
- počítací dotaz $O(log\ n)$ (když si pamatujeme velikost podstromů)
- Build $O(n\ log\ n)$
- Prostor $O(n)$

Nyní ji zdynamizujeme velmi jednoduše například převedením na splay strom. Při rotacích budeme updatovat velikosti podstromů a Insert i Delete budou mít složitost $O(log\ n)$.

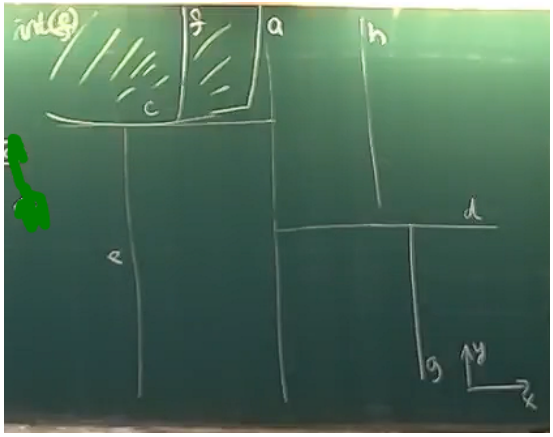
Definujte k-d stromy a ukažte, že 2-d intervalové dotazy trvají Ω(sqrt(n)).

viz. obecné info o geometrických datových strukturách ve Velké Otázce

k-dimenzionální stromy (k-d) v $\mathbb{R}^d$ (něco jiného něž range trees)

Jako BVS ale na i-té hladině porovnává $(i\ mod\ d)$-tou souřadnici.

Příklad v $\mathbb{R}^2$
zatím předpokládejme unikátní hodnoty souřadnic ((2,1) a (2,0) zde nemůže existovat)
Kořen dělí prostor podle x souřadnice, levý vrchol c dělí levou polorovinu a pravý vrchol d dělí pravou polorovinu (viz. obrázek).



(vlevo nahoře je žlutě $int(f)$)
RangeQuery() algoritmus z malé otázky o intervalových dotazech v 1-dim. stromě nám pořád funguje.

Build 2-d stromu

- 1, m <- medián x-ových (nebo y, podle patra) souřadnic v B
- 2, kořen s klíčem m
- 3, rekurze na body vlevo/vpravo

Složitost

- Build - $O(n \log n)$ (n na najití mediánu na $\log n$ vrstvách)
- Prostor - $O(n)$
- Dotaz - ? Příklad níže

Příklad špatného dotazu na 2-d stromě

- Mějme body, co leží na přímce $X = \{(i, i) \mid i \in \mathbb{N}, 1 \leq i \leq 2^k - 1\}$ (horní strop zvolen pro vhodné půlení intervalů podle mediánu na stejné kousky)
- Po půlení nám vyjde dokonale vyvážený binární strom
- Dotaz: $Q = \{0\}$ x $\mathbb{R}$
- Na sudých hladinách jdu jen jedním směrem na lichých oběma (viz. obrázek tučné vrcholy a hrany)
- 
- Takže počet navštívených listů = $2^{hloubka/2} = 2^{\log n/2} = \sqrt{n}$
- Takže dotaz nám trvá $\Omega(\sqrt{n})$
- (BONUS: potom v obecných k-stromech to bude $\Theta(n^{1-1/k})$)

Ukažte, jak dynamizovat k-rozměrné intervalové stromy (stačí Insert).

Líně vyvažované 2D intervalové stromy

Z líně vyvážených 1D stromů víme, že každý prvek přispívá $\log n$ do potenciálu, aby se to uamortizovalo
Zvolme potenciál jako $\Phi = (\sum$ potenciálů sekundárních stromů a potenciál primárního stromu $\log n)$ *(* $\log n$ protože každý vrchol v primárním má ještě sekundární strom na sestavení)

Insert

- V každém sekundárním stromu, kterého se dotkne zvýší potenciál o $\log n$ v $O(\log n)$ stromech (toto jsou klasické líně vyvažované 1D stromy)
- V primárním stromě zvýší potenciál $\log n$, ale jakmile potřebujeme rebuild nějakého podstromu primárního stromu (část primárního stromu + všechny sekundární stromy), tak potřebujeme rebuild i všech sekundárních stromů. Tím pádem se nám rebuild logaritmicky zpomalil a tak nám stačí když každý prvek bude přispívat do potenciálu $log^2\ n$.
- Navíc insert udělá nějakou práci, ale ta je logaritmická v primárním a logaritmická v sekundárním, což se nám schová do O(...).
- Celková amortizovaná cena je $O(log^2\ n)$
- 1D strom $O(log\ n)$, 2D strom $O(log^2\ n)$, 3D strom $O(log^3\ n)$ -> k-rozměrný strom $O(log^k\ n)$

Ukažte, jak použít suffixové pole a LCP pole na nalezení nejdelšího společného podřetězce dvou řetězců.

viz. Velká otázka na Suffixové pole a LCP

Složíme si oba řetězce za sebe a vložíme mezi ně znak, který se v řetězcích vyskytovat nemůže a je na začátku abecedy lexikograficky. Poté sestavíme suffixové a LCP pole. Všimněme si, že všechny suffixy z prvního řetězce budou mít na konci daný dělící znak, tudíž se nám nemůže stát, že by některý suffix z prvního řetězce byl stejný jako z druhého. Teď hledáme nějaké i a j, které mají maximální LCP. Toto nastane pro sousední suffixy, protože oba nám budou začínat daným hledaných nejdelším podřetězcem. Tím pádem mi stačí projít všechny dvojice v Suffixovém poli a hledat maximum LCP hodnoty toho prvního z dvojice. Pozor je potřeba kontrolovat, zda jsou oba z různých řetězců. To zjistíme snadno pomocí hodnoty Suffixového pole, protože známe pozici dělícího znaku ve spojeném řetězci.

(Toto není otázka) Něco málo k paralelním výpočtům:

Paralelní RAM

- P procesorů (RAMů)
- Globální paměť (nepočítáme)
- Lokální paměť (každého procesoru)
- Společný program, registr CPUID
- Každý procesor poběží asynchronně
- Posloupnost instrukcí vykonávané jedním procesorem říkáme proces
- CREW (buď libovolně mnoho čtení nebo 1 zápis)
- Race condition (všichni známe)
- Mutex (všichni známe) - lock, unlock
- Deadlock (všichni známe)

Zamykání v BVS

- zamykáme vždy vrchol, kterým procházíme -> problém, jakmile jeden procesor projde kořenem, tak všichni čekají je to sériové
- zamykáme okénkem o velikosti 2 - zamkneme aktuální vrchol, potom potomka, kam chceme, pak odemkneme aktuální vrchol a tak pokračujeme dále, dobrá strategie pro velký strom v poměru k počtu procesorů + mazání je problém

Definice korektnosti DS - Požadujeme serializovatelnost -lze seřadit operace do lineárního uspořádání které je:

- Konsistentní z pohledu DS
- Konzistentní s pořadím operací v každém procesu

Červeno-černé, AVL či Splay stromy nebudou fungovat dobře, kvůli nutnosti vracet se zpět stejnou cestou. (a, b)-strom je na tom lépe.

Ukažte, jak paralelizovat (a, b)-strom.

Insert
Přetečení budeme řešit už když dojdeme do vrcholu při vkládání, který má maximální počet klíčů (i kdyby nepřetekl po insertu). Problém by mohl nastat, když při štěpení bychom najednou neměli dost klíčů na vytvoření dvou vrcholů. To však můžeme opravit požadavkem na $b = 2a$ místo $2a - 1$.

Delete
Preventivně budeme slučovat vrcholy nebo přehazovat mezi dvěma. Problém nastane když potřebujeme pracovat mezi sourozenci, kdy nevíme jestli budeme potřebovat pravého nebo levého sourozence. Pokud bychom si řekli, že vždy pravého, abychom měli jasné konzistentní uspořádání primárně podle hloubky a sekundárně zleva doprava, tak nastane problém u nejpravějšího sourozence, protože nemá pravého sourozence. Řešení je odemknout ho, zamknout nejdříve levého a pak pravého (nebo to dělat preventivně).
Další problém nastane když bychom měli nahradit aktuální vrchol nejmenším vrcholem z pravého podstromu. Možnost je uzamknout celou cestu, což bude pomalé. Druhé řešení je označit si vrchol za smazaný (pomníček), kde potom není potřeba ani strom čistit, či přestavovat.

Navrhněte a analyzujte bez zámkovou implementaci zásobníku.

Bez zámkové paralelní DS

Atomické operace

- read/write
- exchange (adresa, x) - původní hodnota je vrácena a místo ní se tam vloží x
- fetch and add (adresa, x) - původní hodnota je vrácena a je k ní přičteno x
- compare and exchange (adresa, expected, new) - porovná hodnotu registru s očekávanou, pokud tam je, tak ji přepíše na novou, pak vrátí původní hodnotu
- CAS některé procesory neumí, ale umí:
  - LL - load linked (adresa) -> hodnota
  - SC - store conditional (adresa, new) -> procesor se sleduje jestli někdo zapisoval mezi LL a SC a podle toho SC uspěje
  - Lze tímto simulovat CAS, tohle umí pro max 2 registry najednou

Lock-free zásobník

Máme HEAD ukazující na začátek zásobníku, každý prvek má referenci na NEXT (je to spojový seznam).

Push(x)

- h = HEAD
- x.NEXT = h
- if CAS(HEAD, h, x):
  - True: HEAD = x

- False: Zkusíme to celé znovu

Pop

- h = HEAD
- n = h.NEXT
- if CAS(HEAD, h, n):
  - True: HEAD = n
  - False: Zkusíme to celé znovu

Problém může být, že se zasekneme ve smyčce vkládání na hodně dlouho (LIVE-LOCK). Naštěstí chování reálných PC je dostatečně náhodné na to, aby LIVE-LOCK vydržel dlouho.

Reálný problém, ale nastane když budeme mít dva procesy a následují posloupnosti instrukcí:

- Zásobník má ze začátku HEAD -> A -> B -> END
- P2: $\frac{1}{2}$pop (první 2 instrukce, takže získá pointer na HEAD -> A -> B)
- P1: pop (dostane A)
- P1: pop (dostane B)
- P1: push(A) (nyní vypadá zásobník HEAD -> A -> END)
- P2: $\frac{2}{2}$pop (CAS zkontroluje zda HEAD=A, provede pop a nastaví HEAD -> B)
- Ale to je špatně, zásobník má být prázdný, ale je tam B ABA Problém

-> Řešení:

- Obyčejných CAS o nevyřeší. CAS2, který by dokázal kontrolovat 2 místa (HEAD a HEAD.next) v paměti najednou a problém by vyřešil, tak skoro žádný hardware neumí.
- Dynamicky alokovat jednotlivé buňky s tím, že nám alokátor musí zajistit unikátní adresy pro všechny prvky. (vhodné pouze pro malé a jednoduché programy)
- DCAS je CAS, co umí kontrolovat dvě místa v paměti, ale musí být za sebou (64-bit PC -> 128 bitů). Mějme u každého ukazatele hned za ním v paměti i verzi ukazatele a pokaždé se bude měnit, tak zvětšíme číslo verze.
- LL+SC řeší problém díky kontrole přístupu k paměti

Push_DCAS(x)

- h = HEAD
- v = HEAD_version
- x.NEXT = h
- if DCAS(<HEAD, HEAD_version>, <h, v>, <x, v+1>):
  - True: HEAD = x
  - False: Zkusíme to celé znovu

Pop_DCAS

- h = HEAD
- v = HEAD_version
- n = h.NEXT
- if DCAS(<HEAD, HEAD_version>, <h, v>, <n, v+1>):
  - True: HEAD = n
  - False: Zkusíme to celé znovu

// Asi není potřeba, ale řekl bych, že je to důležitou součástí analýzy
Správa paměti (dynamicky alokované krabičky)

- Atomicky posouvat pointer do paměti, čímž alokujeme další paměť, ale není zde možnost znovu použít alokované buňky
- Problém jsou popnuté krabičky které po uvolnění z paměti, pokud měl nějaký jiný proces na ní referenci by mohl dělat n = h.NEXT přičemž by h byl nulptr
- Řešení: Free-List - Krabičky čekají v pomocné datové struktuře na uvolnění. Problém je jak zjistit, že můžeme dealokovat:
  - 1, Globální synchronizace - všechny procesy v jeden moment (kdy nemají lokální pointery na krabičky) na sebe počkají (nebo i postupně, pokud si uděláme kopii free-listu), problém je, že musíme mít celý "svět" pod kontrolou, používá se v Linux jádru
  - 2, Atomické počítání odkazů - problém může nastat, když projdeme celý free-list a zjistíme, že nemůžeme skoro nic uvolnit, to se však uamortizuje:
    - pro p procesů, každý maximálně r odkazů, takže projdeme free-list jenom když má alespoň $\geq 2rp$ prvků

    - z toho vychází, že můžeme uvolnit alespoň polovinu free-listu
    - poté bude pop vypadat následovně
        - h = HEAD
        - v = HEAD_version
        - h.ref++
        - if h != HEAD:
            - True: h.ref--; Zkusíme to celé znovu
            - False: Jdeme dál
        - n = h.NEXT
        - if DCAS(<HEAD, HEAD_version>, <h, v>, <n, v+1>):
            - True: HEAD = n
            - False: Zkusíme to celé znovu
- 3, Hazard pointery - představme si nástěnku, kam procesy dávají lístečky, že zrovna pracují s danou krabičkou, aby ho nikdo nerušil. Free-list zde vezme krabičku a koukne se jestli je na nástěnce, aby viděl, zda ji může uvolnit. Mějme pole hazard pointerů o p bloků pro p procesů, kde v každém bloku je r položek s pointery (každý proces může mít max r pointerů).
    - poté bude pop vypadat následovně
        - h = HEAD
        - v = HEAD_version
        - hp = h
        - if h != HEAD:
            - True: Zkusíme to celé znovu
            - False: Jdeme dál
        - n = h.NEXT
        - if DCAS(<HEAD, HEAD_version>, <h, v>, <n, v+1>):
            - True: HEAD = n; hp = 0
            - False: Zkusíme to celé znovu
    - Obsluha free-listu:
        - 1, Odpojíme atomicky free-list
        - 2, Zkopírujeme pole hazard-pointerů
        - 3, Pro všechny prvky:
            - Pokud prvek není v kopii hazard-pointeru, tak dealokujeme
            - Jinak vrátíme do nového Free-listu (ne do kopie)

Popište atomická primitiva a jejich vlastnosti. Vysvětlete problém ABA a jeho řešení.

viz. **Něco málo k paralelním výpočtům** a **Navrhněte a analyzujte bez zámkovou implementaci zásobníku.**