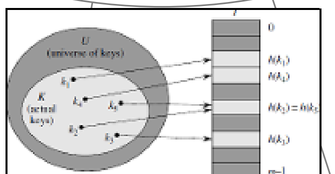
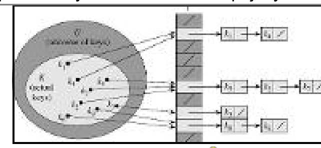
Hašování - Operace: SEARCH, INSERT, DELETE
Univerzum klíčů U a **K**⊆U je množina použitých klíčů |K| = n
Hašovací funkce h: U→{0,1,..,m−1} mapuje univerzum U do (menší) tabulky T [0,..,m−1], |U| > m
Kolize je situace: $h(k_i) = h(k_j)$, pro $k_i \neq k_j$; $k_i, k_j \in K$
Faktor naplnění: α = n/m

snaha o efektivní prostorovou reprezentaci a malou časovou náročnost operací



DELETE se ve strukturách, které ho nepodporují, řeší označením políčka jako smazaného s možností využití při vkládání. V případě, že polovina polí je blokovaná tímto způsobem, se vše přehashuje.

Hašování separovanými řetězci (chaining)
při kolizi vytvoříme v tabulce spojový seznam



- nevýhodou obou je nerovnoměrné využití paměti někde můžou být dlouhé seznamy jinde nic
- **Hašování s uspořádanými řetězci**
  seznamy jsou uspořádány podle velikosti
  -rychlejší op.SEARCH
- **Očekávaná délka řetezce**
  E(l) = α = n / m
  var(l) = n/m (1 - 1/m)
  - délky řetězců mají binomiální rozdělení

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

  - dk:

    $$E(l) = \sum_{l=0}^{n} l \cdot P(\ell(i) = l) = \sum_{l=0}^{n} l \binom{n}{l} (\frac{1}{m})^l (1 - \frac{1}{m})^{n-l} = \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^l (1 - \frac{1}{m})^{n-1-l} = \frac{n}{m} (\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m}.$$

    $$E(l^2) = E(l(l-1)) + E(l),$$

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

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

    $$var(l) = E(l - E(l))^2 = E(l^2) - (E(l))^2 = \frac{n}{m}(1 + \frac{n-1}{m}) - (\frac{n}{m})^2 = \frac{n}{m}(1 - \frac{1}{m}).$$
- **Očekávaná délka nejdelšího řetězce** (ocekavany nejhorsi pripad)
  E(NP) = O (log n / log log n)

Hašování s přemisťováním
každé políčko má dva ukazatele (**prev, next**), jejichž pomocí tvoří kolizní položky řetězec
pokud kolizní položka zabírá místo položce, co na místo dle haše patří, je přemístěna

- nevýhoda: přemísťování při INSERTu zpomaluje

řádek	key	next	previous
P(0)			
P(1)	1	9	
P(2)			
P(3)	73	6	
P(4)	11	5	9
P(5)	161		4
P(6)	53		3
P(7)	7		
P(8)	28		
P(9)	141	4	1

Hashování se dvěma ukazateli
-podobné, políčko má ukazatele následník a začátek řetězce (**begin, next**)
-prvky se nepřemisťují, místo toho může být začátek přesměrován pomocí druhého ukazatele

- zrychlí INSERT a DELETE ale zpomalí SEARCH

Srůstající hashování (coalesced)
-jen 1 ukazatel v tabulce navíc - odkaz na další prvek NEXT

- Pocet testu od nejlepsiho: VICH,LICH,EICH,EISCH,LISCH
- **standardní,bez pomocné paměti (LISCH, EISCH)**
  LISCH(late-insertion standard) přidává se za poslední prvek řetězce, projití celého řetězce s testy na přítomnost prvku, potom vložení na libovolné volné místo v tabulce a připojení na konec řetězce
  EISCH(early-insertion standard) v případě neprázdného řetězce vždy za 1. prvek, vložení na nějaké volné místo v tabulce a jen přepojení ukazatelů NEXT

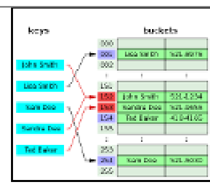
řádek	key	next	řádek	key	next
P(0)			P(0)		
P(1)	1	9	P(1)	1	9
P(2)			P(2)		
P(3)	73	6	P(3)	73	6
P(4)	28		P(4)	28	7
P(5)	7	4	P(5)	7	
P(6)	53		P(6)	53	
P(7)	161	5	P(7)	161	5
P(8)	11	7	P(8)	11	4
P(9)	141	8	P(9)	141	8

- **s pomocnou pamětí (LICH, VICH, EICH)**
  paměť na dvě části: (hash-funkcí) přímo adresovatelná a pomocná část (bez přístupu hash-funkcí)
  Kolize ukládáme do pomocné části, pak teprve do přímo adresovatelné(oddalujeme srůstání řetězců). Chování se tak až do určitého okamžiku podobá separovaným.
  LICH(late-insertion) vždy přidává na konec řetězce
  VICH(varied-insertion) v pomocné paměti jako LICH, v přímo adresovatelné části jako EICH
  EICH(early-insertion) v případě neprázdného řetězce vždy za 1. prvek

řádek	key	next	řádek	key	next	řádek	key	next
P(0)			P(0)			P(0)		
P(1)	1	10	P(1)	1	10	P(1)	1	9
P(2)			P(2)			P(2)		
P(3)	73	11	P(3)	73	11	P(3)	73	11
P(4)	28	9	P(4)	28	7	P(4)	28	7
P(5)	7	4	P(5)	7		P(5)	7	
P(6)			P(6)			P(6)		
P(7)	161	5	P(7)	161	5	P(7)	161	5
P(8)	11	7	P(8)	11	4	P(8)	11	4
P(9)	31		P(9)	31	8	P(9)	31	10
P(10)	141	8	P(10)	141	9	P(10)	141	8
P(11)	53		P(11)	53		P(11)	53	

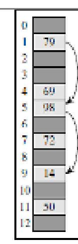
- **Očekávaná délka řetězce(poč.testů) při zaplněné tabulce**:
  3 a s pomocnou pamětí 2
- **Význam pomocné paměti**: pomáhá tomu aby se v tabulce neblokovalo zbytěčně místo - protože prvky samozřejmě obsazují mista pro prvky z jiné skupiny 0..m
- Nepodporují efektivní **DELETE**
- β = |přímá paměť|/|celk.pam.| je adresovací faktor

Hašování s lineárním přidáváním (probing)
-tabulka má jen klíč, kolizní prvky se dávají na první volné místo:
h(k, i) = (h'(k) + i) mod m



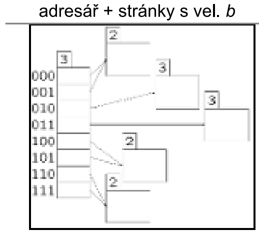
- spec. případ dvojitého pro $h_2(x) = 1$
- použitelné do zaplnění 75% pak moc velké shluky
- DELETE se řeší přehashováním celého "clusteru" dat

Dvojité hašování
-podobné, jakmile selže h_1 použije se 0,1,...-násobek h_2:
$h(k, i) = (h_1(k) + ih_2(k)) \bmod m$



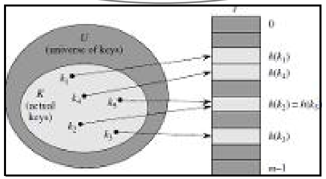
- $nsd(h_2(x), m)=1$ (h_2 a m musí být nesoudělné)
- i<m jinak přeplníme T
- je vhodné volit: $h_1(x)=h_1(y) \Rightarrow h_2(x) \neq h_2(y)$
- **MEMBER/INSERT**: hledame nejmensi i, ze $T[h_1(x) + ih_2(x) \bmod m]$ je volne
- **DELETE** nefunguje protože každý klíč může mít různé sekvence hledání volného místa.

Externí - Fagin (rozšiřitelné)
adresář + stránky s vel. *b*

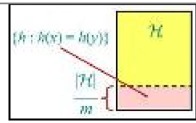


- cíl: minimalizace počtu operací s externí pamětí
- **MEMBER**
  - načteme adresář a podle prefixu h(x) najdeme stránku
  - načteme stránku do int.paměti a najdeme x
  - vyžaduje **nejvýše 3 op.**s externí pamětí
- **INSERT**
  - **načteme** adresář a podle prefixu h(x) **najdeme** stránku
  - načteme stránku nebo vytvoříme novou když není
  - přidáme x
  - když jí přeplníme, zvětšíme adresář o 1 bit a rozštěpíme stránku na 2
  - uložíme stránku a případně i zdrojnásobený adr.
- **DELETE**
  - **načteme** adresář a podle prefixu h(x) **najdeme** stránku +stránku kandidáta (liší o poslední bit)
  - načteme stránku, smažeme x
  - pokud |stránky a kandidáta|≤b sloučíme je, uložíme a aktualizujeme adresář
  - otestujeme jestli se adresář nedá zkrátit a příp. zkrátíme
- INSERT a DELETE vyžadují **nejvýše 6 op.**s externí pamětí
- **oč.počet stránek je n/(b ln2)**
- **oč.velikost adresáře** $e/(b \ln 2)\, n^{1+1/b}$
  - více než lin.růst, tj. nelze používat donekonečna
- Neumí se vypořádat s příliš mnoho kolizemi => hašovací funkci je třeba vybírat uvážlivě.

Hašování - Operace: SEARCH, INSERT, DELETE
Univerzum klíčů U a **K**⊆U je množina použitých klíčů |K| = n
Hašovací funkce h: U→{0,1,..,m−1} mapuje univerzum U do (menší) tabulky T [0,..,m−1], |U| > m
Kolize je situace: $h(k_i) = h(k_j)$, pro $k_i \neq k_j$; $k_i, k_j \in K$
Faktor naplnění: α = n/m



Univerzání hashování (c-universalni system)
abychom si vylepšili naše šance že h bude fungovat dobře, zaveďme si obecnější třídu funkcí H⊆{h|h: U→{0,1,..,m−1}} kde:
∀x≠y∈U: |{h∈H: h(x)=h(y)}|≤c|H|/m



než začneme hashovat, vybereme z nich náhodnou funkci h∈H (ta se stane atributem konkrétní instance hashovací tabulky)

Alternativní definice:
H⊆{h|h: U→{0,1,..,m−1}} kde:
∀x≠y∈U: P(h(x)=h(y))≤c/m

oč.počet testů: O(1+cα)

Existence:
předpokládejme: U={0,1,..,N−1} a fce:
$h_{a,b}(k)$ = ((ak + b) mod N) mod m

dk:

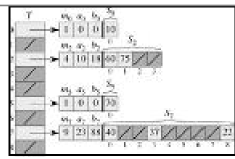
$|H| = N^2$, což je počet dvojic (a, b).
Nechť (x, y) jsou libovolné, ale pevné. Zároveň platí $x \neq y$.
Kolize nastane v případech, když :
$h_{ab}(x) = h_{ab}(y)$, neboli
$ax + b = q + rm \pmod N$
$ay + b = q + sm \pmod N$
kde (a, b) jsou neznámé a parametry (q, r, s) nabývají všech hodnot takových, že
$q \in \{0, .., m-1\} \wedge r, s \in \{0, .., \lceil N/m \rceil - 1\}$.
N je prvočíslo, tedy $\mathbb{Z}_N$ je těleso a pro každou trojici parametrů (q, r, s) má soustava právě jedno řešení (a, b).
Počet kolidujících funkcí je přesně tolik, jako počet trojic (q, r, s), který je $m \cdot \lceil N/m \rceil^2$.

$$|\{h_{ab} : h_{ab}(x) = h_{ab}(y)\}| \le m \left\lceil \frac{N}{m} \right\rceil^2 = \frac{\lceil \frac{N}{m} \rceil^2}{(\frac{N}{m})^2} \frac{N^2}{m} = c\frac{|H|}{m}$$

$\Rightarrow$ tento univerzální systém lze zkonstruovat. Velikost H jeN^2 a c $= \frac{\lceil \frac{N}{m} \rceil^2}{(\frac{N}{m})^2}$

Konstrukce

Perfektní hashování - najdeme hash fukci pro danou množ. co nepřipouští kolize. Nevýhoda této metody je, že nelze dost dobře implementovat operaci INSERT, proto se dá prakticky použít pouze tam, kde předpokládáme hodně operací MEMBER a jen velmi málo operací INSERT. Kolize se potom dají řešit třeba malou pomocnou tabulkou, kam se ukládají kolidující data. Příklad perfektního hašování pomocí dvouúrovňového univerzálního hašování:



(N,m,n)-perfektní soubor funkcí:
H⊆{h|h: U→{0,1,..,m−1}} kde:
∀S⊆U že |S|=n existuje h∈H perfektní pro S (bez kolizí)

veta o dolnim odhadu velikosti systemu

$$|H| \ge \max\{\frac{\binom{N}{n}}{\binom{m}{n}(\frac{N}{m})^n}, \frac{\log N}{\log m}\}.$$

veta o existenci perfektniho hashovaciho systemu

konstrukce

$h_k(x) = (kx \mod N) \mod m \quad$ pro $k = 1, 2, \ldots, N-1$.

(a,b) stromy

každý vnitřní vrchol má:
ρ(v) - počet synů
S_v - pole ukazatelů na syny
H_v - pole největších prvků z každého podstromu

Pravidla:
- kazdy vnitrni vrchol (mimo kořene) má a≤x≤b synů
- všechny cesty od kořene k listům jsou stejně dlouhé
- kořen je bud listem nebo ná 2≤x≤b synů
- a≥2 a b≥2a-1

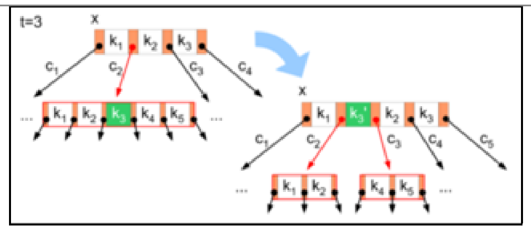
Hloubka: m-min počet listů, každý vrchol až na kořen má ≥ a synů, d-hloubka
$m \geq a^{d-1}$; $\log_a m \geq d-1$; $d \leq 1+\log_a m$ => řádově O(log n)

vhodný pro externí paměť protože více dat v uzlech snižuje přístupy do paměti

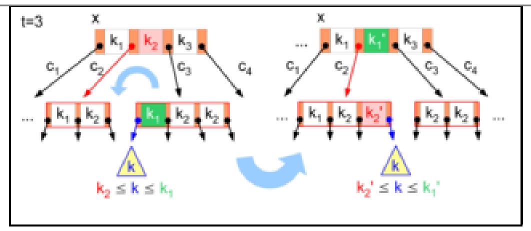
vyžadují čas **O(log |S|)**

Vyhledej: jdeme dolu dokud nedojdeme k nejbližšímu (>= hledanému **x**)

Insert:
-pomocí **Vyhledej** zkusím prvek najít,
-pokud tam prvek není, vytvořím nový list (pod otce nalezeného), připojím na správné místo do S_v a případně upravím H_v
-postupně nahoru štěpím, je-li potřeba, extrémně rozštěpím kořen.

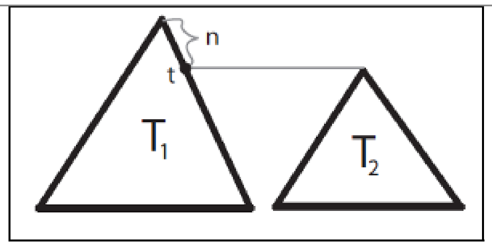


Delete
-pomocí Vyhledej zkusím prvek najít
- to jedno políčko v Sv a Hv zruším, opravím ρ
-pokud dostanu méně než a synů v uzlu, spojím s bezprostř. bratrem (má-li ten právě a synů), nebo přesunu nějaký z bratra do mého uzlu.



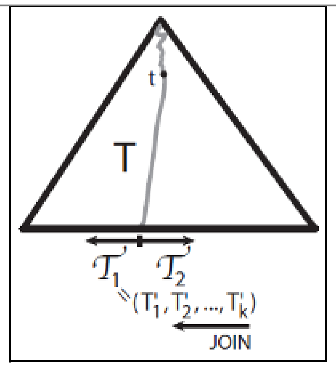
vyžaduje čas **O(|h(T1) - h(T2)|)**

Join
Přepokládá max S1 < min S2.
Je-li h(T1) >= h(T2),
-najde v T1 hladinu o 1 nad připojení a v ní největší prvek nebo
-vytvoří nadkořen T1 v případě rovnosti,
 -slije spojí do něj prvky obou kořenů a případní provede štěpení.
Jinak hledá a připojuje v T2.



vyžaduje čas **O (log|S|)**

Split: Prochází postupně dolů, rozděluje uzly (podstromy s prvky < x, resp >= x) a hází výsledky do 2 zásobníků.
Pokud oddělí více než 1 krajní prvek, hodí na zásobník strom, jehož kořen je právě oddělená část uzlu, jinak na zásobník dává podstrom onoho krajního prvku. Tak pokračuje až k listům, pokud tam najde přímo x, tak ho vyhodí. Stromy ze
zásobníků spojí postupným voláním JOIN.



A-sort - je aplikací (a,b) stromů v třídících algoritmech, vhodnou pro částečně předtříděné posloupnosti. Jinak proti klasickým algoritmům nemá žádné výhody.

maximum inverzí je:
$$\binom{n}{2} = \frac{n(n-1)}{2}$$

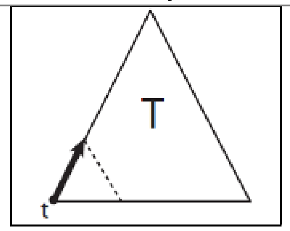
počet inverzí definujeme jako: **F=|{ (x_i, x_j) | i < j, x_i > x_j }|**

částečně předtříděné posloupnosti = čím menší počet inverzí ve vstupním poli tím lépe

rozšíříme definici (a,b) stromu:
- listy stromu propojíme do seznamu
- uložíme ukazatel a cestu od nejmenšího (nejlevějšího) listu do kořene

algoritmus: Odzadu (od "předtříděně největšího") vkládat prvky do stromu modifikovaným **A-INSERT**em a pak přečíst posloupnost listů (jít po NEXT).

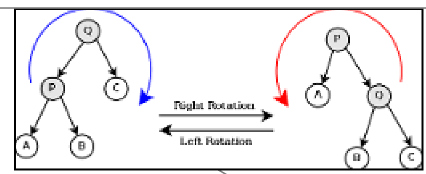
A-INSERT(x) pracuje tak, že místo pro vložení prvku hledá od FIRST (jde postupně nahoru po otcích a hledá, kde nejdřív může slézt zas k listům).
1. začneme v nejmenším listu a jdeme nahoru dokud není v podstromu list **t** >= vkládanému **x**
2.jdeme dolu dokud nedojdeme k nejbližšímu listu **t** >= vkládanému **x**
3. proveď klasický zbytek (a,b)-INSERT



složitost prům./nejh.: O(n log(F/n)) kde F je počet inverzí vstupu

Nepoužívá DELETE, hodí se na toto (2,3) stromy. Pro míru F ≤ n log n, má složitost O(n log log n), v urč. případech i rychlejší než Quicksort.

(pravá/levá) rotace - pomocné operace pro vyvažování, proveditelné v konst.čase



BVS - uzel má dva syny
levy podstrom obsahuje mensi nez klic
pravy podstrom vetsi

AVL stromy

Pravidla: - ∀uzel platí: **výška** jeho **levého a pravého podstromu se liší nejvýše o 1**, uchováváme si v uzlu o tom info {-1,0,1}

Hloubka: $a_0=0, a_1=1, a_2=2$... pro h ≥ 2:
(min.počet vrcholů stromu výšky h) = $a_h=1+a_{h-1}+a_{h-2} > 2^{h-1/2}+2^{h-2/2} = 2^{h/2}(2^{-1/2}+2^{-1}) \sim 2^{h/2} * 1{,}21 > 2^{h/2}$ a tedy $h \leq 2\log a_h$ =>
hloubka O(log n)

Insert (max 2 rot.)
- postupujeme od nově přidaného uzlu směrem nahoru a cestou opravujeme balance uzlů podle hloubky podstromů
- pokud se balance uzlu změnila na 2 nebo −2 (silně nevyvážený vrchol) - > je nutná reorganizace stromu … operace rotace (LR a RL rotace jde brat jako jednu)
- zrotovaný podstrom má stejnou výšku jako původní, takže není potřeba postupovat dále nahoru ke kořeni stromu (**tzn. rotace 1x a dost**)
- časová složitost je nejvýše rovna výšce stromu, tzn. O(log N)

Delete (může mít rotace až do kořene)
- vyhledat uzel s rušenou hodnotou a odebrat ho jako v BVS:
● má- li 0 nebo 1 syna - > vypustit přímo tento uzel U
● má- li 2 syny, nahradit jeho hodnotu maximem z levého podstromu (nebo minimem z pravého podstromu) a vypustit ze stromu tento náhradní uzel U
● případného syna uzlu U přepojit na otce uzlu U místo U samotného
- postupujeme od otce zrušeného uzlu směrem nahoru ke kořeni stromu, v každém uzlu přepočítáváme balanci
- pokud vznikne silně nevyvážený vrchol (hodnota 2 nebo −2), provedeme v tomto uzlu rotaci − z balance uzlu a jeho synů vyplývá potřebný druh rotace (LL, LR, RR, RL), při rotaci se opraví údaje o výšce a balanci dotčených uzlů
- cestou se může provádět **rotace až log n krát**
- časová složitost je rovna výšce stromu, tzn. O(log N)

Find,Insert,Delete vždy v O(log n)

! ! RB stromy

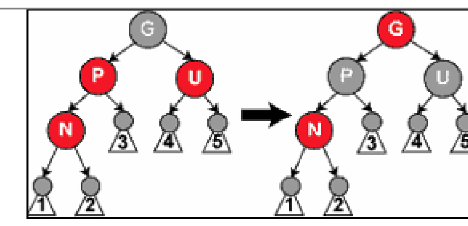
Find,Insert,Delete vždy v O(log n)

máme více případů ale zase jen max 2 rotace (delete)

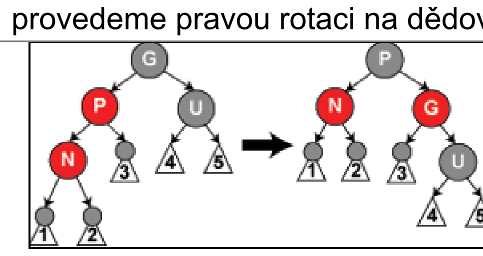
Pravidla:
- **listy** jsou **černé**
- pokud má **červený** uzel otce, otec je **černý**
- Na cestě **od kořene do lib. uzlu** s jedním nebo žádným synem je **stejný počet černých** uzlů

Hloubka: k-poč.černých vrcholů,n-počet vrcholů
nejmenší strom má všechny vrcholy černé=>hloubka k-1 a pocet vrcholu je $1+2+...+2^{k-1} = 2^k-1$;
největší: střídavě černé a červ. =>hloubky 2k-1 a pocet vrcholu je $1+2+...+2^{2k-1} = 2^{2k}-1$;
$2^k-1 \leq n \leq 2^{2k}-1 \Rightarrow k \leq \log_2(n) + 1 \leq 2k$
a dále k ≤ hloubka ≤ 2k => hloubka O(log n)

Insert (max 1 rot.)
- vložíme **nový vrchol N** jako do standatního BVS (jako list) a **obarvíme červeně**, první a 3 pravidlo jsme neporušili
- pokud je **otec černý** tak ani druhé (->konec)
- pokud je **otec červený**
  - **strýc** N **je červený** → **přebarvíme otce a strýce na cerno** a **dědu** na **červeno**

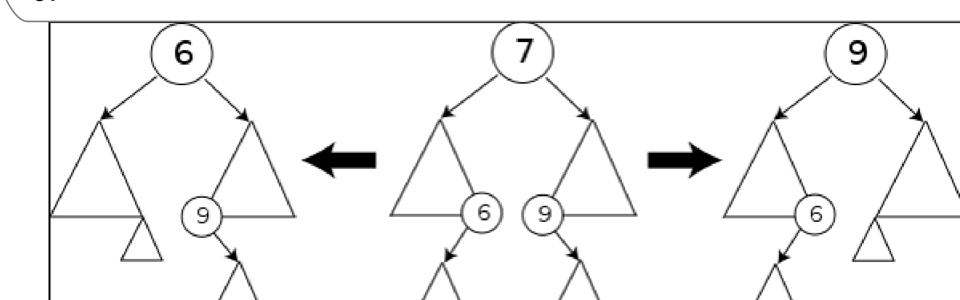


    posuneme chybu na dědu
  - **strýc** N **je černý**
    - **N je levý syn**: provedeme pravou rotaci na dědovi a přebarvíme



      pravidla splněna,konec
    - **N je pravý syn**: levá rotace na otci → přechozí případ

Delete (max 2 rot.)
- pokud **mazaný vrchol** ma vice nez 1 syna, tak že do něj zkopírujem **maximum z levého podstromu (min. z pravého)**, který má vždy jenom max. jednoho syna, a zkopirovany vrchol pak mazeme místo nej
- pokud je **červený**, nahradime ho jeho **černým synem N** a ten přebarvíme na červeno (=>konec)
- pokud je **smazaný vrchol černý** (ne kořen) je porušeno pravidlo 3:

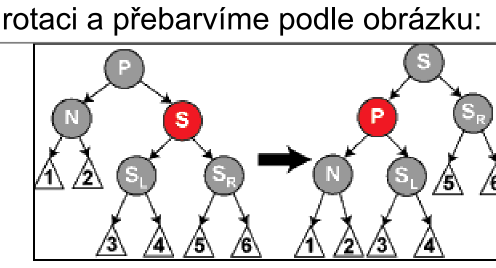


- pokud je jeho **syn N červený**, přebarvíme ho načerno (=>konec)
- jinak je **N černý** a označíme ho za dvojnásobně černý a tuto barvu posouváme výše dokud se jí nezbavíme (pomocí bodů níže)
- pokud je dvojnásobně černý kořen, přebarvíme ho na černý (->konec)
- jinak označíme P jako nového rodiče N a S jako bratra N (musí být interní)

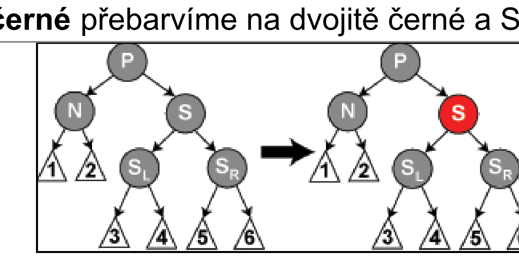
3-téměř červenočerný strom

situace převedena na 2,3 nebo 4

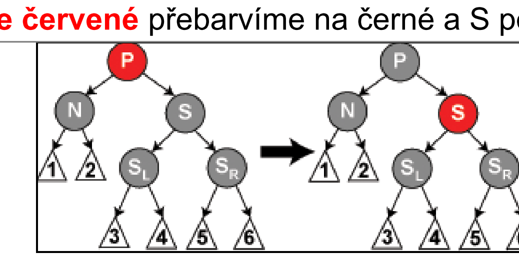
1. **S je červený** (=>oba jeho synové jsou černí a P je také černé) provedeme LL rotaci a přebarvíme podle obrázku:



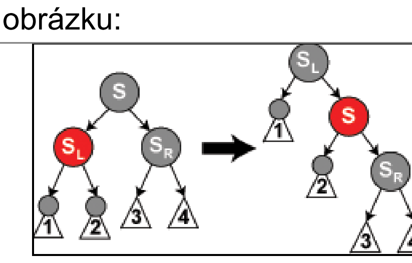
2. S je černé a
- **má dva černé syny**
  - pokud **P je černé** přebarvíme na dvojitě černé a S podle obrázku



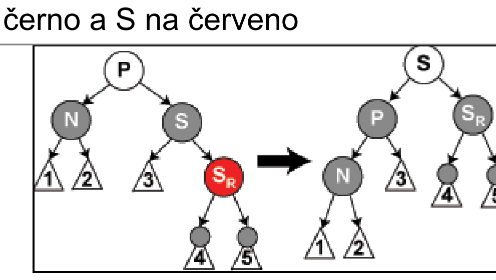
    pokračujeme až ke kořeni zkoušením podmínek 1,2…
  - pokud **P je červené** přebarvíme na černé a S podle obrázku



    konec
- jeho **levý syn červený** a **pravý černý**, provedeme RR rotaci a přebarvíme podle obrázku:



  situace se nám převedla na další případ
- jeho **pravý syn je červený**, uděláme LL rotaci, přebarvíme SR na černo, z N odebereme dvojnásobnou černou, pokud je P červené přebarvíme na černo a S na červeno



  konec

haldy - stromová struktura kde platí buď:
otec≥syn (tzv. max-heap)
nebo
otec≤syn (tzv. min-heap)

d-regulární haldy

d-reg.strom má max 1 vrchol, který není list a má synů < d

Pravidla: - ∀ vrchol má nejvýše d synů
- vrchol není list => ∀vrchol s menším čísl… synů
- vrchol má synů < d => ∀vrcholy s většímy čísly jsou listy

Implemetace v poli: prvek je na pozici *k* a potomci vždy na:
(k-1)d+2 až (k-1)d+(d+1)
rodiče na 1+floor((k−2)/d)

Hloubka: zřejmě ceil $\log_d$ (n(d-1) +1)

INSERT: **přidám na konec haldy** a když nesedí prohazujeme s rodičem
(tedy **posouvám nahoru**) O($\log_d$ n)

DELETE/ DELETEMIN: odeberu, **nahradím posledním prvkem** a pokud nesedí prohazuji s potomky kteří porušují vlastnost víc (tedy **posouvám nahoru nebo dolů**) O(d $\log_d$n)

DECREASE: **zvětším číslo o a** a když nesedí prohazujeme s rodičem
(tedy **posouvám nahoru**) O($\log_d$ n)

Vytvoření haldy z n prvků: vytvoříme z prvků lib. d-reg.strom a postupně
posouváme dolů každý ne-list
nejh.čas O(d^2 n)

Dijkstra

Dijkstra - hledame nejkratsi cesty z vrcholu *s* do ostatních v grafu (aplikace haldy)

Algoritmus: stavy vrcholu: Neviden, Viden, Hotov
D(*):= ∞, D(s):=0
Z(*):=N, Z(s):=V
while ∃v: Z(v) = V
  vybereme *v* že: Z(v) = V, D(v) = min (*zde záleží na použité dat. struktuře*)
  Z(v) = H
  for ∀w : (v,w)∈E(G)
    D(w):=min(D(w),D(w) + l(v,w))
    if Z(w)=N ⇒ Z(w):=V

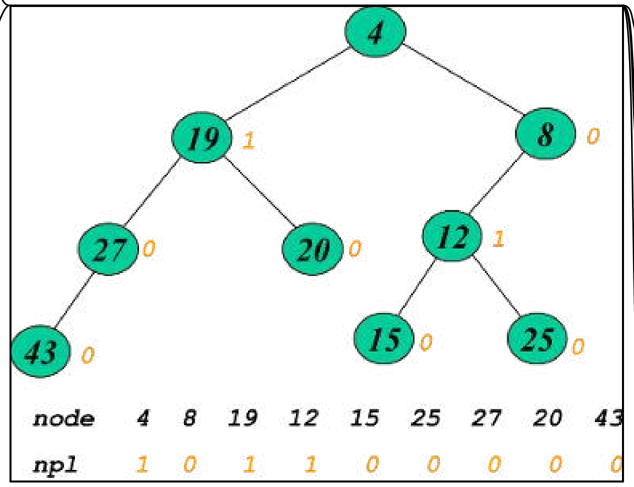
Složitost: záleží na datove strukture pouzite pro ∀v: Z(v)=V a D(v) =min, pokud pouzijeme pole tak je O(|V|^2 + |E|) = O(|V|^2)
pokud ***fibonacciho haldu*** je O(|V| log |V| +|E|)
pokud ***d-regulární*** tak: O(|V| log |V| +|E| log |V|)

Funguje pouze s nezápornými hranami

Leftist haldy

npl(v) - (null path lenght) je **delka nejkratsi cesty** z vrcholu v **do vrcholu co nemá 2 syny**

leftist haldy: binarni strom
-vrchol má 1 syna => je to levý syn
-vrchol má 2 syny => npl(levého syna)≥npl(pravého)
-platí usporadni na halde



MERGE(T1,T2) (rekurzivní):
-Testuje prázdnost obou stromů
-volá MERGE(podstrom pravého syna stromu T1 s menším klíčem v kořeni, T2)
-výsledek připojí místo onoho pravého syna.
-Pokud neplatí podmínka na npl, vymění syny T1.

INSERT(x) je vytvoření jednoprvkové haldy a zavolání MERGE

DECREASE(s,a) se udělá snížením hodnoty ve vrcholu **s** o **a**, zavoláním OPRAV, tj. jeho odříznutím od zbytku haldy, a MERGE podstromu a zbytku.

OPRAV(T,v) odtrhne podstrom a dopočítá všem vrcholům správné npl. Po odtržení vrcholu a příp. přehození pravého syna doleva jde nahoru dokud provádí změny npl (možno až do kořene), vztahuje npl odspoda a příp. prohazuje syny.

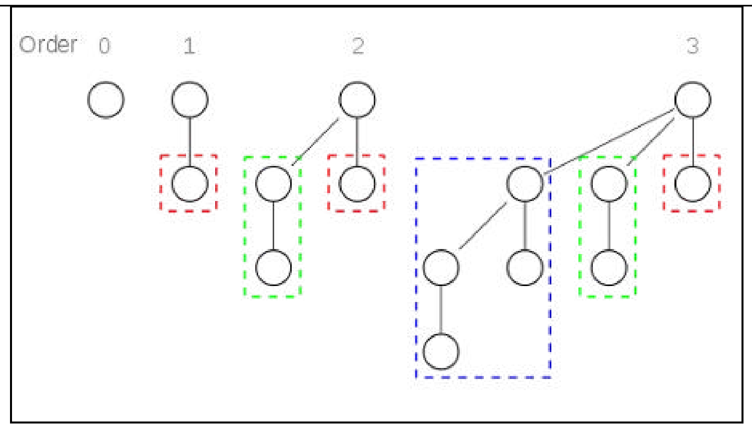
v nejh.případě O(log n)

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

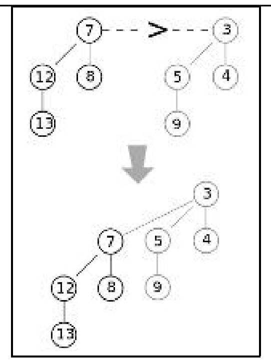
MAKEHEAP potřebuje jen O(n)

Binomiální haldy

binomiální haldy: les binomiálních stromů - každý řád max.jednou,
-binom.strom řádu *i* se skládá z kořene a synů ze stromů izomorf. řádu *0,...,i-1* - chová se jako malá halda
-používá se ukazatel na strom s min.prvkem O(1)



MERGE: pracuje stejně jako binární sčítání - za pomoci operace SPOJ (slepení dvou stromů, přilepím jako syna toho, který má v kořeni vyšší klíč) **slepí jen stromy stejného řádu**, přenáší výsledky do dalšího spojování (přenos + obě haldy mající strom daného řádu = vyplivnutí 1 stromu na výsledek a spojení zbývajících dvou). Složitost - O(log |S1| + log |S2|), protože 1 krok (SPOJ) je konstantní.



INSERT je vytvoření jednoprvkové haldy a zavolání MERGE

DECREASE INCREASE a DECREASE změní hodnotu f u prvku a zavolají probublávání nahoru nebo dolů

není podporováno DELETE, jen jako DECREASE + DELETEMIN.

vše vyžaduje čas **O(log n)**
jen INCREASE O($\log^2$n)

umožňuje rychlou implementaci Insert O(log n) (amotizovaně O(1)) a Merge (dvou hald) O(log n)

Fibonacciho haldy

fibonacciho haldy - les haldových stromů vzniklých posloupnosti operací pro fib.haldy, vychází z binomiálních

-při odebírání prvků lze oddelit z nekořenového prvku max 1 syna, jinak se oddeli cely uzel do noveho stromu
-při odebrání minima se počet stromů naopak snižuje - spojují se

v je vrchol a má i synů => pak podstrom v má alespoň F_i+1 vrcholů

MERGE, INSERT, MIN a DELETEMIN jsou stejné jako v líné implementaci binomiálních hald, jen požadavek na isomorsmus s Hi je nahrazen požadavkem na daný rank.

MERGE, INSERT, DECREASE mají amort.složitost **O(1)**

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

MIN,DELETEMIN,INCREASE,DELETE mají amort.sl. **O(log n)**

Insert je amortizovaně O(1) Delete O(log n)

třídící algoritmy

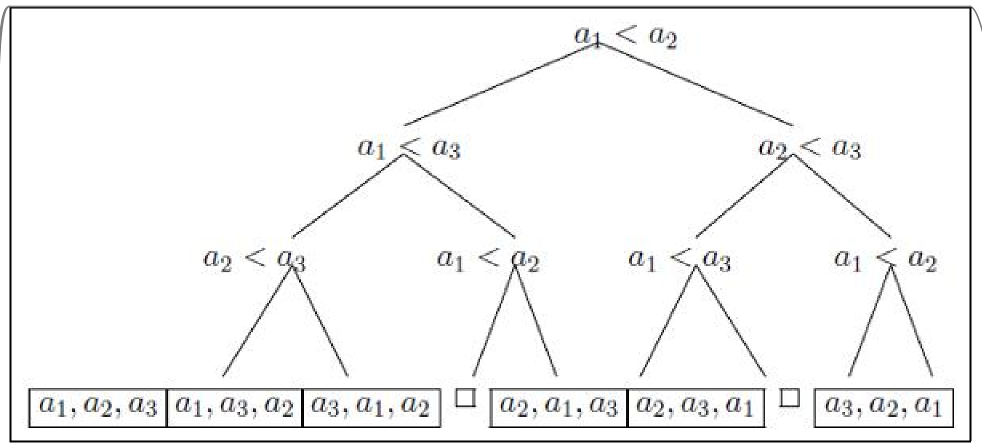
Každý třídicí algoritmus, jehož jedinou primitivní operací s prvky vstupní posloupnosti je porovnání, vyžaduje v nejhorším i v očekávaném případě alespoň *cn log n* času pro nějakou konstantu c>0.
V **nejhorším případě** použije alespoň ceil((n+½) $\log_2 n -$ n/ln2) porovnání.

Očekávaný počet porovnání při rovnoměrném rozdělení vstupních posloupností je alespoň (n+½) $\log_2 n$ − n/ln2 .

- algoritmy, ktoré "porušujú" dolný odhad zložitosti jsou třeba BUCKETSORT

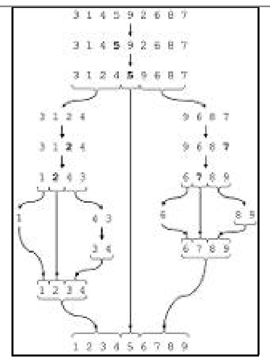
Rozhodovací strom pro alg.A - je bin.strom s vnitřními vrcholy ohodnocenými porovnáními: ai≤aj, i≠j∈1...n když ∀permutaci π na {1,...,n} platí, že posl. při třídění permutace π algoritmem A je stejná jako posloupnost porovnání při průchodu permutace tímto stromem.

A je třídící alg. pro posloupnosti dlouhé *n* s jedinou primit.operací: porovnání.



porovnávací algoritmy

Quick sort - vezme prvek(pivot) ze sekvence a rozdělí jí na 2 podle toho kdo je větší nebo menší než pivot, ty pak rekurzivně setřídí



- + univerzální
- − v základu nevhodný pro předtříděné

```
qsort [] = []
qsort (pivot:tail) =
  qsort (filter (< pivot) tail) ++
  [pivot] ++
  qsort (filter (>= pivot) tail)
```

- **nejhorší složitost** - pivot špatně: T(n)=T(n-1)+Θ(n)=**Θ(n^2)**
  průměr složitost - pivot náhodně: **O(n log n)**
  nejlepší složitost - pivot medián: T(n)=2T(n/2)+O(n) ⇒ a=2 b=2 d=1 ; $2=2^1$ ⇒ Θ(n log n)
  O(n) paměti
- pro kazdy pevne zadratovany algoritmus volby pivota existuje nejaky vstup, pro ktery alg. bezi v O(n^2)
- zatimco pri nahodne volbe pivota/hledani medianu je čas horší (multipl.konstanta) než MERGESORT / HEAPSORT pro vsechny vstupy (nahodna volba i hledani medianu chce cas), takže se typicky bere median ze 3-5 pevne danych prvku
- pri zohledneni multiplikativni konstanty jde o nejrychlejsi znamy alg. pro trideni zalozeny na vzajemnem porovnavani prvku

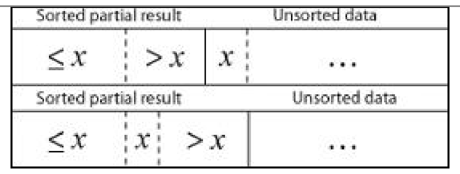
Heapsort - nasypeme prvky do max-haldy a pak je po jednom odebirame
O(n log n)

sekvenční třídění

Select sort - najde nejmenší prvek a vloží ho na konec, opakuje
O(n^2) - vždy

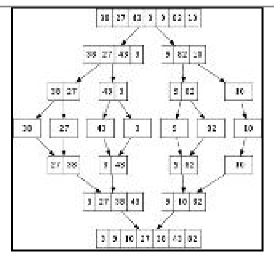
- + vhodný pro hodně přeházené
- − nevhodný pro předtříděné
- − nevhodný pro větší data

Insert sort - vzdy vezmeme prvek z nesetridene casti pole a vlozi ho do setridene casti
O(n^2) - v prům.i nejhorším případě v nejlepším O(n)
O(n+1) až O(1) - paměti



- + vhodný pro předtříděné
- − nevhodný pro hodně přeházené
- + stabilní
- − nevhodný pro větší data

(Adaptabilní) Mergesort
- projdi posloupnost a rostoucí úseky hazej do fronty
- pak je slejvej dokud nemáš jenom 1



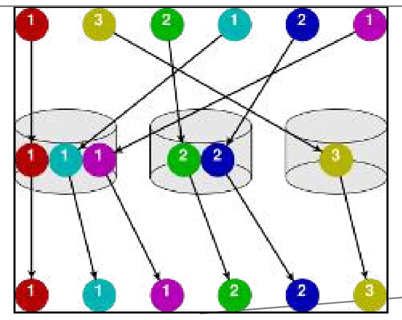
- + stabilní
- − základní MS paměťově náročný
- + adaptabilní na předtříděné
- − základní MS nevhodný pro malinká data
- **nejh. O(n log n)**
- **nejl. O(n)**
- **Slévání nestejně dlouhých posloupností** - pro slevani posloupností se pokusime vytvorit optimalni strom posloupností dílčích slévání (=chceme aby slévané posl. byly stejné dlouhé)

OPTIM
- Vstup: l(P1),..,l(Pn) (délky posl.)
- mame množinu stromů (na začátku jednoprvkových), vždy vezmeme 2 s nejmenším l v kořeni a spojíme je dohromady kořenem s hodnotou l_1+l_2 a výsledný vrátíme do množiny stromů
- Výstup: bin.strom porovnání množin Pi
- **pokud je vstup neklesající potřebuje O(n)**
- vstup se musí předtřídit např.BUCKETSORTEM (O(n+N))

přihrádkové třídění - lineární třídění

O(n+počet přihrádek) nemusí být rychlejší než klasické algoritmy

Bucket sort - nacpe n čísel z intervalu <0,N> do m přihrádek
O(n+N) čas./paměť složitost



- pozor na wikipedii tomuhle rikaji pidgenhole sort
- + stabilní

WordSort - založen na bucket-sortech, řeší lexikografické uspořádání slov

- **lexikograficke usporadani**: a<b⇔∃i=0,..,min(|a|,|b|) že∀j=0,..,i: $a_j=b_j$ platí |a|=i < |b| nebo i < min(|a|,|b|) a $a_{i+1}=b_{i+1}$
- **algoritmus**
  1. slova roztridi bucketsortem do množin L_i podle jejich delek — zvladnu v O(L)
  2. vytvori dvojice **[pořadí ve slově**;**písmeno]** — (napr. pro slovo ahoj vznikne [1,a], [2,h], [3,o], [4,j])
  3. dvojice setridi podle **písmen** a pak takto setříděný seznam ještě setřídím podle **pořadí ve slově** — (oboji je bucket-sortem, takze je to stabilni)
  4. takto setridene dvojice rozdelim do mnozin (nazveme je S_i) podle stejneho poradi pismene ve slove a **odstraní duplicity** — (napr. [1,a] a [1,d] jdou do stejne mnoziny, ale [2,a] do dalsi; de facto se jen odstrani duplicity)

  (kroky 2.–4.) vyžaduje O(L)

  5. hlavní cyklus, kde i = délka nejdelšího slova,...,1:
     I. L_i spojim s T rozdelim do mnozin T_x podle i-teho pismena x — inicializace Tx je O(|Σ|)
     II. T_x(jen neprázdná) podle množiny S_i seradim za sebe do T — v každém kroku potřebuji dvakrát čas O(součet délek slov dlouhých i nebo víc)
     — tedy pouzivam bucket-sort, takze je operace stabilni a zachovava poradi podle uz projitych kroku
  6. na konci pridam na zacatek T prazdna slova z L_0 a mam vysledek
- **čas. složitost** je **O(L+|Σ|)** kde L je součet délek všech slov

- **hledání**
  - ❗ **Vyhledávání v uspořádaném poli**
    - **základní algoritmus**
      - next = next(d,h)
      - **while** (s[next]≠x **and** d < h)
        - **if** (s[next] < x) d = next+1
        - **else** h = next-1
        - next = next(d,h)
      - rekni ANO pokud s[next] == x, jinak rekni NE
    - **varianty funkce NEXT**
      - **unární** $d+1$
        - jedeme popořadě
        - max.počet dotazu |S|, očekávaný |S|/2 tedy **složitost nejh./prům.** je **O(|S|)**
      - **binární** $\lceil\frac{d+h}{2}\rceil$
        - půlíme interval
        - **počet dotazů nejh./prům.** je **O(log|S|)**
      - **interpolační** $d+\lceil\frac{x-A(d)}{A(h)-A(d)}(h-d)\rceil$
        - přes kraje protáhneme přímku a píchneme tam, kde by na ní mělo být x
        - **počet dotazů prům.** je **O(log log|S|)**
        - **počet dotazů nejh.** je **O(|S|/2)**
      - **zobecněné kvadratické**
        - přes kraje protáhneme přímku a píchneme tam, kde by na ní mělo být x
        - ```
          if nastav then
           parita := true
           next := nextInterp(d,h)
           nahoru := s[next] < x
           nastav := false
           return next
          end if
          if not parita then
           next := nextBin(d; h)
           parita := true
           return next
          end if
          delka := h - d + 1
          next := nahoru ? d + √delka : h + √delka
          if s[next] < x xor nahoru then
           nastav := true
          else
           parita := false
          end if
          return next
          ```
        - **počet dotazů prům.** je **O(log log|S|)**
        - **počet dotazů nejh.** je **O(log|S|)**
  - **Pořádkové statistiky**
    - **Find(X,k)**
      -pivot zvolime z X
      -Rozdělime X na L(=menší než pivot), P(=větší než pivot)
      -Rekurzivně se zavoláme na jednu z L, P (tu, ve ktere se ma vyskytovat hledany prvek)
      -pokud neni ani v jedne, nasli jsme prvek :)
      - nejhorsi pripad $O(n^2)$
      - ocekavany cas O(4n)
      - je vetsinou rychlejsi nez select, ale v ridkych pripadech muze byt kvadraticky
    - **Hledání k-tého nejmenšího prvku v lin.čase (Blum et al.)**
      Select(X,k)
      -Pokud n ≤ 5 -> vyřešíme přímo
      -Vstup rozdělíme na pětice $P_1 \ldots P_{n/5}$ (poslední může být neúplná), to zvládneme v O(n)
      -∀ i: m_i = median(P_i), to zvládneme v O(n)
      -pivot = Select ($m_1,\ldots,m_{n/5}$, n/10) (medián mediánů - T(n/5))
      -Rozdělime X na L(=menší než pivot), S(=stejné jako pivot), P(=větší než pivot)
      -Rekurzivně se zavoláme na jednu z L, S, P (tu, ve ktere se ma vyskytovat hledany prvek - T(7n/10))
      -pokud neni ani v jedne, nasli jsme prvek :)

      T(n)=Θ(n)+T(n/5)+T(7n/10) =Θ(n)
      - ⌛ **nejh./nejl./prum. pripad O(n)**
          ```