Při práci s daty z uspořádaného univerza je často třeba pracovat s dalšími operacemi založenými na uspořádání. Příkladem takových úloh jsou grafové algoritmy. Částečně to umožňují binární vyhledávací stromy nebo (a,b)-stromy a také haldy. Haldy jsou velmi jednoduché, a proto jsou několikrát rychlejší než binární vyhledávací stromy nebo (a,b)-stromy. Cena za to je, že haldy nepodporují operaci **MEMBER**. Další nevýhodou hald, které jsme probírali, je fakt, že pracují jen s minimem nebo maximem, probírané haldy nepracovaly s oběma extrémy najednou. Cílem tohoto textu je se seznámit s datovými strukturami, které pracuje s více operacemi založenými na uspořádání.

Nejprve si zadefinujeme naši úlohu: Máme univerzum U, které se skládá z přirozených čísel $\{0,1,\ldots,m-1\}$ pro nějaké přirozené číslo m. Chceme nalézt datovou strukturu reprezentující množiny $S \subseteq U$ a umožňující operace **MEMBER**, **MIN**, **MAX**, **DELETE**, **INSERT**, **DELETEMIN**, **DELETEMAX**, **SUCCESSOR** a **PREDECESSOR**.

Nejprve popíšeme přidané operace **SUCCESSOR** a **PREDECESSOR**. Pro reprezentovanou množinu $S \subseteq U$ a prvek $x \in U$ definujeme, že $\mathbf{SUCCESSOR}(x) = \min\{t \in S \mid x \leq t\}$, když existuje $t \in S$ takové, že $x \leq t$, nebo $\mathbf{SUCCESSOR}(x) = \infty$, když takové $t \in S$ neexistuje. Zde ∞ formálně znamená největší prvek univerza (všimněme si, že formálně platí $\min \emptyset = \infty$). Dále $\mathbf{PREDECESSOR}(x) = \max\{t \in S \mid t \leq x\}$, když existuje $t \in S$ takové, že $t \leq x$, v opačném případě $\mathbf{PREDECESSOR}(x) = -\infty$, kde $-\infty$ znamená nejmenší prvek univerza (platí $\max \emptyset = -\infty$).

Emde Boasova struktura.

Popíšeme si strukturu, která řeší tuto úlohu. Tato struktura na rozdíl např. od vyhledávacích stromů dává omezení na velikost univerza. Je založena na vlastnostech celočíselných operací mod a $\div$. Připomínáme, že když m, n, p a q jsou přirozená čísla taková, že $m = np + q$ a $q < n$, pak $m \div n = p$ a $q = m \bmod n$. Čísla p a q jednoznačně reprezentují číslo m vzhledem k pevně danému n. To vede k pozorování, že když $m = nl$, pak můžeme množinu $S \subseteq U$ reprezentovat ve dvou univerzech, U' velikosti l a U'' velikosti n pomocí množin $S' = \{s \div n \mid s \in S\} \subseteq U'$ a $S'' = \{s \bmod n \mid s \in S\} \subseteq U''$. To nám umožňuje čínská věta o zbytcích. Bohužel pro daný prvek $t \in S$ bychom potřebovali spojit jeho reprezentaci jak v U' tak v U''. Navíc tato reprezentace by porušovala uspořádání. Klíčem k řešení problému je reprezentovat $S \subseteq U$ pomocí množiny $S' = \{s \div n \mid s \in S\} \subseteq \{0,1,\ldots,l-1\}$ v univerzu $U' = \{0,1,\ldots,l-1\}$ a množin $S''_i = \{s \bmod n \mid s \in S,\ s \div n = i\}$ pro $i = 0,1,\ldots,l-1$ v univerzu $U'' = \{0,1,\ldots,n-1\}$.

Tuto reprezentaci chceme použít rekurzivně. Proto je přirozené požadovat, aby n a l byla přibližně stejně velká (tj. blízká $\sqrt{m}$, kde m je velikost univerza). Pak bude rekurzivních kroků přibližně $\log\log m$. Z tohoto důvodu budeme předpokládat, že univerzum U je tvaru $\{0,1,\ldots,n-1\}$, kde $n = 2^k$ pro nějaké přirozené číslo k. Datová struktura je definována rekurzivně a budeme jí říkat k-struktura (k je určeno velikostí univerza). Předpokládejme, že máme reprezentovat $S \subseteq U$. Nejdřív položme $k' = \lceil \frac{k}{2} \rceil$ a $k'' = \lfloor \frac{k}{2} \rfloor$.
Když $S \subseteq U$ je neprázdná, pak k-struktura reprezentující množinu obsahuje

(1) číslo $|S|$;

(2) rostoucí dvousměrný seznam t_S všech prvků množiny S;
(3) když $|S| \geq 2$, pak pole $P_S[0,..,n-1]$ ukazatelů, kde

$$P_S(i) = \begin{cases} \text{prvek } i \text{ v seznamu } t_S & \text{když } i \in S, \\ NIL & \text{když } i \notin S; \end{cases}$$

(4) když $|S| \geq 2$, pak k'-strukturu TOP_S reprezentující množinu $\{s \div 2^{k''} \mid s \in S\} \subseteq \{0, 1, \ldots 2^{k'} - 1\}$
a pole ukazatelů $BOTTOM_S[0,..,2^{k'}-1]$ na k''-struktury, kde struktura, na kterou ukazuje $BOTTOM_S(i)$, reprezentuje množinu $\{s \bmod 2^{k''} \mid s \in S,\ s \div 2^{k''} = i\}$, když je tato množina neprázdná, a $BOTTOM_S(i) = NIL$, když neexistuje $s \in S$ takové, že $s \div 2^{k''} = i$.

To znamená, že prázdnou množinu reprezentuje prázdný ukazatel "NIL". Dále ukazatel $BOTTOM_S(i)$ budeme ztotožňovat s k''-strukturou, na kterou ukazuje. Pak S lze považovat za ukazatel na strukturu reprezentující množinu S.

Algoritmy realizující operace **MEMBER**, **MIN** a **MAX** jsou jednoduché. Operace **MEMBER** vychází z toho, že $s \notin S$, právě když $P_S(s) = NIL$, a operace **MIN** a **MAX** využívají toho, že t_S je rostoucí dvousměrný seznam všech prvků v S.

MEMBER(x)
if $|S| > 1$ **then**
  if $P_S(x) \neq NIL$ **then**
    Výstup: $x \in S$
  else
    Výstup: $x \notin S$
  endif
endif
if $|S| = 1$ **then**
  if x je prvek t_S **then**
    Výstup: $x \in S$
  else
    Výstup: $x \notin S$
  endif
endif
if $S = NIL$ **then Výstup**: $x \notin S$ **endif**

MIN
if $|S| > 0$ **then**
  Výstup: první prvek v seznamu t_S
else
  Výstup: neexistuje
endif

MAX
if $|S| > 0$ **then**

Výstup: poslední prvek v seznamu t_S
else
Výstup: neexistuje
endif

DELETEMIN
if $|S| = 1$ **then** $S := NIL$, odstranit seznam t_S **endif**
if $|S| = 2$ **then**
$|S| := 1$, **DELETE**(první prvek) v t_S
odstranit P_S, TOP_S a $BOTTOM_S$
endif
if $|S| > 2$ **then**
$x :=$ první prvek v seznamu t_S, $|S| := |S| - 1$
$x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
DELETEMIN v $BOTTOM_S(x')$
if $BOTTOM_S(x') = NIL$ **then**
DELETEMIN v TOP_S, $BOTTOM_S(x') := NIL$
endif
DELETE(x) v seznamu t_S, $P_S(x) := NIL$
endif

DELETEMAX
if $|S| = 1$ **then** $S := NIL$, odstranit seznam t_S **endif**
if $|S| = 2$ **then**
$|S| := 1$, **DELETE**(poslední prvek) v t_S
odstranit P_S, TOP_S a $BOTTOM_S$
endif
if $|S| > 2$ **then**
$x :=$ poslední prvek v seznamu t_S, $|S| := |S| - 1$
$x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
DELETEMAX v $BOTTOM_S(x')$
if $BOTTOM_S(x') = NIL$ **then**
DELETEMAX v (TOP_S), $BOTTOM_S(x') := NIL$
endif
DELETE(x) v seznamu t_S (odzadu), $P_S(x) := NIL$
endif

Je lehce vidět, že algoritmy pro operace **MEMBER**, **MIN** a **MAX** jsou korektní a že spotřebují konstantní čas.

Algoritmy, které realizují operace **DELETEMIN** a **DELETEMAX** jsou složitější, protože musí také upravit rekurzivní struktury.

Všimněme si, že **DELETE** v seznamu t_S vyžaduje vždy $O(1)$ času (protože t_S je dvousměrný seznam a odstraňujeme první nebo poslední prvek). Pokud $|S| \leq 2$, pak celá operace **DELETEMIN** a **DELETEMAX** bez rekurzivního volání vyžaduje $O(1)$ času. Když

algoritmus volá dvě rekurzivní volání sebe sama, pak jedno je použito na případ, že reprezentovaná množina má nejvýše jeden prvek, a proto vyžaduje čas $O(1)$. Protože když nepočítáme rekursivní volání sebe sama, tak algoritmus vyžaduje $O(1)$ času, tedy dostáváme, že celkový čas algoritmu je úměrný délce řetězce rekurzivních volání sebe sama.

Korektnost algoritmů plyne z pozorování, že když $x = \min(S)$ ($x = \max(S)$), pak x' je nejmenší (největší) prvek v množině reprezentované TOP_S a x'' je nejmenší (největší) prvek v množině reprezentované v $BOTTOM_S(x')$.

Nyní popíšeme algoritmy realizující operace **INSERT** a **DELETE**. Algoritmus **DELETE** zjistí, zda $x \in S$ a když ano, pak x odstraní z P_S, t_S a ze struktury $BOTTOM_S(x \div 2^{k''})$ odstraní prvek $x \bmod 2^{k''}$. Když potom $BOTTOM_S(x \div 2^{k''})$ reprezentuje prázdnou množinu, tj. $BOTTOM_S(x \div 2^{k''}) = NIL$, odstraní prvek $x \div 2^{k''}$ ze struktury TOP_S. Algoritmus **INSERT** je inverzní. Když $x \notin S$, pak $x \bmod 2^{k''}$ přidá do $BOTTOM_S(x \div 2^{k''})$ a pokud tato struktura reprezentovala prázdnou množinu, tak přidá prvek $x \div 2^{k''}$ do struktury TOP_S. Problém je, kam dát x v seznamu t_S. K tomu použijeme operaci **SUCCESSOR**(x) a pak upravíme P_S, t_S a $|S|$.

INSERT(x)
if $|S| = 0$ **then**
  vytvořme seznam $t_S = \{x\}$, $|S| = 1$, stop
endif
if $|S| = 1$ **then**
  if x není prvek t_S **then**
    vložme x do seznamu t_S a vytvořme pole ukazatelů P_S
    for every $u \in t_S$ **do**
      $P_S(u)$ je ukazatel na prvek u v t_S
    enddo
    vytvořme TOP_S reprezentující $\{s \div 2^{k''} \mid s \in S\}$
    vytvořme pole ukazatelů $BOTTOM_S$
    for every $u \in t_S$ **do**
      INSERT$(u \bmod 2^{k''})$ do $BOTTOM_S(u \div 2^{k''})$
    enddo
    $|S| = 2$
  endif
else
  if $P_S(x) = NIL$ **then**
    $x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
    if SUCCESSOR$(x) \neq \infty$ **then**
      $y :=$ **SUCCESSOR**(x)
    endif
    if $BOTTOM_S(x') = \emptyset$ **then**
      INSERT(x') do TOP_S
    endif

INSERT(x'') do $BOTTOM_S(x')$
if SUCCESSOR$(x) \neq \infty$ **then**
INSERT(x) do seznamu t_S před y
else
INSERT(x) jako poslední prvek seznamu t_S
endif
$P_S(x) :=$ ukazuje na prvek x v seznamu t_S, $|S| := |S| + 1$
endif
endif

DELETE(x)
if $|S| = 1$ a x je v seznamu t_S **then**
$S := NIL$, odstranit seznam t_S
endif
if $|S| = 2$ a x je v seznamu t_S **then**
$|S| := 1$, **DELETE**(x) v seznamu t_S
odstranit P_S, TOP_S a $BOTTOM_S$
endif
if $|S| > 2$ a $P_S(x) \neq NIL$ **then**
$|S| := |S| - 1$, $x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
DELETE(x'') v $BOTTOM_S(x')$
if $BOTTOM_S(x') = NIL$ **then**
DELETE(x') v TOP_S
endif
DELETE(x) v seznamu t_S pomocí $P_S(x)$, $P_S(x) := NIL$
endif

Všimněme si, že operace **DELETE** na nejvýše dvouprvkovou množinu a operace **INSERT** na jednoprvkovou množinu vyžadují čas $O(1)$ a nevolají samy sebe. Operace **DELETE** a **INSERT** na seznam t_S vyžadují čas $O(1)$, protože známe pozici x v seznamu t_S pomocí ukazatele $P_S(x)$. Proto operace **DELETE** a **INSERT**, když nepočítáme rekursivní volání sebe sama, vyžadují $O(1)$ času. Když **DELETE** použije dvě rekurzivní volání sebe sama, pak jedno je použito na případ, že reprezentovaná množina má nejvýše dva prvky, když **INSERT** použije dvě rekurzivní volání sebe sama, pak jedno je použito na případ, že reprezentovaná množina má nejvýše jeden prvek, proto celkový čas algoritmů je úměrný délce maximálního řetězců rekurzivních volání sebe sama.

Nyní popíšeme algoritmus pro operaci **SUCCESSOR**(x). Algoritmus popíšeme jen pro případ, že $x \notin S$, v opačném případě je výsledek x a algoritmus je jasný. Všimněme si, že když existuje $s \in S$ takové, že $s > x$, pak buď existuje $s \in S$ takové, že $s > x$ a $s \div 2^{k''} = x \div 2^{k''}$, a v tom případě

$$\min\{s \in S \mid s > x\} = (x \div 2^{k''})2^{k''} + \min\{s \in S \mid s > x,\ s \div 2^{k''} = x \div 2^{k''}\}$$

nebo neexistuje $s \in S$ takové, že $s > x$ a $s \div 2^{k''} = x \div 2^{k''}$ a v tom případě existuje $s' \in S' = \{s \div 2^{k''} \mid s \in S\}$ takové, že $x' < s'$. Pak pro $y' = \min\{s \div 2^{k''} \mid s \in S,\ s \div 2^{k''} >$

$x \div 2^{k''}\}$ platí

$$\min\{s \in S \mid s > x\} = y'2^{k''} + \min\{s \bmod 2^{k''} \mid s \in S,\ s \div 2^{k''} = y'\}.$$

Následující algoritmus právě realizuje tyto výpočty. Algoritmus pro operaci **PREDECESSOR**(x) je založen na symetrických vztazích.

SUCCESSOR(x)
if $S = NIL$ **then** **SUCCESSOR**$(x) := \infty$, stop **endif**
if $|S| = 1$ **then**
  nalezneme **SUCCESSOR**(x) pomocí prohledání t_S, stop
endif
if $P_S(x) \neq NIL$ **then**
  Výstup: **SUCCESSOR**$(x) := x$
else
  $x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
  $z' :=$ **MAX** v $BOTTOM_S(x')$
  if $TOP_S.P(x') = NIL$ nebo $x'' > z'$ **then**
    if **SUCCESSOR**$(x') \neq \infty$ v TOP_S **then**
      $y' :=$ **SUCCESSOR**(x') v TOP_S
      $y'' :=$ **MIN** v $BOTTOM_S(y')$
      Výstup: **SUCCESSOR**$(x) := y'2^{k''} + y''$
    else
      Výstup: **SUCCESSOR**$(x) := \infty$
    endif
  else
    $y'' :=$ **SUCCESSOR**(x'') v $BOTTOM_S(x')$
    Výstup: **SUCCESSOR**$(x) := x'2^{k''} + y''$
  endif
endif

PREDECESSOR(x)
if $S = NIL$ **then** **PREDECESSOR**$(x) := -\infty$, stop **endif**
if $|S| = 1$ **then**
  nalezneme **PREDECESSOR**(x) pomocí prohledání t_S
  stop
endif
if $P_S(x) \neq NIL$ **then**
  Výstup: **PREDECESSOR**$(x) := x$
else
  $x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
  $z' :=$ **MIN** v $BOTTOM_S(x')$
  if $TOP_S.P(x') = NIL$ nebo $x'' < z'$ **then**
    if **PREDECESSOR**$(x') \neq -\infty$ v TOP_S **then**
      $y' :=$ **PREDECESSOR**(x') v TOP_S

$y'' :=$ **MAX** v $BOTTOM_S(y')$
Výstup: **PREDECESSOR**$(x) := y'2^{k''} + y''$
else
Výstup: **PREDECESSOR**$(x) := -\infty$
endif
else
$y'' :=$ **PREDECESSOR**(x'') v $BOTTOM_S(x')$
Výstup: **PREDECESSOR**$(x) := x'2^{k''} + y''$
endif
endif

Korektnost algoritmů plyne z poznámky před algoritmem. Dále si všimněme, že každý běh algoritmu **SUCCESSOR** nebo **PREDECESSOR** volá sám sebe nejvýše jednou a protože operace **MIN** a **MAX** vyžadují čas $O(1)$, tak algoritmy **SUCCESSOR** a **PREDECESSOR**, když nepočítáme rekursivní volání sebe sama, vyžadují $O(1)$ času. Proto celkový čas algoritmů je úměrný délce řetězce rekurzivních volání sebe sama.

Dále si všimněme, že když některý algoritmus pracuje v k-struktuře a volá sám sebe, tak rekurzivně volaná verze algoritmu bude pracovat buď v k'-struktuře nebo v k''-struktuře. Když použije dvě volání sebe sama, tak jedno volání vyžadovalo totální čas $O(1)$. To znamená, že délka řetězce rekuzivních volání je nejvýše $\lfloor \log k \rfloor$. Protože $k = \log n$, tak algoritmy pro **DELETEMIN**, **DELETEMAX**, **INSERT**, **DELETE**, **SUCCESSOR** a **PREDECESSOR** běží v čase $O(\log \log n)$.

Ještě odhadneme velikost reprezentace. Seznam t_S a pole P_S vyžadují velikost paměti $O(|U|)$. Z toho dostaneme, když $|U| = 2^k$ a $P(k)$ je velikost paměti, kterou potřebujeme pro reprezentaci podmnožiny U, pak platí

$$P(k) = O(2^k) + (\lceil \frac{k}{2} \rceil + 1)P(\lceil \frac{k}{2} \rceil).$$

Když tuto rekurzi budeme postupně dosazovat do vzorečku, dostaneme

$$P(k) = O(2^k) + \sum_{i=1}^{\log k} O((\lceil \frac{k}{2^i} \rceil + 1)2^{\frac{k}{2^i}})$$

a odtud dostáváme, že $P(k) = O(2^k \log k)$, a protože $k = \log |U|$, tak dostáváme

Věta. *k-struktura reprezentuje podmnožiny univerza U, kde $U = \{0, 1, \ldots, n-1\}$ a $n = 2^k$ pro nějaké přirozené číslo k, a vyžaduje $O(n \log \log n)$ paměti. Operace* **MEMBER**, **MIN** *a* **MAX** *vyžadují čas $O(1)$ a operace* **INSERT**, **DELETE**, **DELETEMIN**, **DELETEMAX**, **SUCCESSOR** *a* **PREDECESSOR** *vyžadují čas $O(\log \log n)$.*

To znamená, že tato implementace k-struktury se hodí jen pro reprezentaci malých univerz (např. když U je nosná množina grafu apod.).

Algoritmus pro operaci **PREDECESSOR**(x) lze realizovat ještě jiným způsobem. Nejprve spočítáme **SUCCESSOR**(x). Když výsledek je x, pak i **PREDECESSOR**$(x) =$

x, jinak je to prvek v seznamu t_S, který předchází **SUCCESSOR**(x) (prvnímu prvku předchází $-\infty$ a ∞ předchází největší prvek v S).
Tato struktura se s výhodou používá pro grafové algoritmy. Tam se setkáváme s různými modifikacemi této úlohy. Jedna z podstatných modifikací je následující. Univerzum jsou všechna přirozená čísla, ale reprezentovaná množina S vždy splňuje podmínku, že $\max(S) - \min(S) \leq n$ pro fixované n. Pak místo S reprezentujeme množinu $\{s \bmod n \mid s \in S\}$ v univerzu $\{0, 1, \ldots, n-1\}$ a známe prvek s nejmenším ohodnocením, nechť je to prvek t a pamatujeme si jeho skutečné ohodnocení – označme si ho min. Pak můžeme použít tuto strukturu pro řešení dané úlohy (kde univerzum má tvar $U = \{0, 1, \ldots, n-1\}$). Pole P_S bude cyklus a my musíme hlídat nejmenší prvek a jeho skutečnou velikost. Skutečné ohodnocení prvku s je

$$\begin{cases} \min + (\text{ohodnocení}(s) - \text{ohodnocení}(t) \bmod n) & \text{když } s \geq t, \\ \min + n + \text{ohodnoceni}(s) & \text{když } s < t. \end{cases}$$

Protože nároky na paměť jsou důležitější než nároky na čas, hledaly se modifikace této struktury, které by měly menší nároky na prostor i za cenu, že se zhorší časová složitost. První pozorování bylo, že nároky této datové struktury na prostor jsou zaviněné polem P_S a seznamem t_S. Tyto struktury jsou tam proto, aby operace **MEMBER**, **MIN** a **MAX** vyžadovaly konstantní čas. Na druhou stranu je lehké najít algoritmy pro operace **MEMBER**, **MIN** a **MAX**, které vyžadují čas $O(\log \log m)$, kde m je velikost univerza, a nepoužívají pole P_S a seznam t_S. Jsou to jednoduché modifikace algoritmu pro operaci **SUCCESSOR**.

To vedlo k následující modifikaci definice k-struktury. Mějme neprázdnou množinu $S \subseteq U$. Pak modifikovaná k-struktura reprezentující S obsahuje

(1) číslo $|S|$;
(2) když $|S| = 1$, pak S je reprezentována seznamem t_S obsahující její jediný prvek;
(3) když $|S| \geq 2$, pak S je reprezentována k'-strukturou TOP_S reprezentující množinu $\{s \div 2^{k''} \mid s \in S\} \subseteq \{0, 1, \ldots 2^{k'} - 1\}$ a
polem ukazatelů $BOTTOM_S[0, .., 2^{k'} - 1]$ na k''-struktury, kde $BOTTOM_S(i)$ ukazuje na k''-struktura reprezentující množinu $\{s \bmod 2^{k''} \mid s \in S,\ s \div 2^{k''} = i\}$, když je tato množina neprázdná a $BOTTOM_S(i) = NIL$, když je tato množina prázdná.

Nyní operace **MEMBER**, **MIN** a **MAX** využívají následujících faktů:

$x \in S$, právě když prvek $x \div k''$ je ve struktuře TOP_S a prvek $x \bmod k''$ je ve struktuře $BOTTOM_S(x \div k'')$,

minimální prvek v S je

$$2^{k''}(\mathbf{MIN}(TOP_S)) + \mathbf{MIN}(BOTTOM_S(\mathbf{MIN}(TOP_S)))$$

a maximální prvek v S je

$$2^{k''}(\mathbf{MAX}(TOP_S)) + \mathbf{MAX}(BOTTOM_S(\mathbf{MAX}(TOP_S))).$$

MEMBER(x)
if $|S| = 0$ **then Výstup**: $x \notin S$, stop **endif**
if $|S| = 1$ **then**
  if x je v seznamu t_S **then**
    Výstup: $x \in S$, stop
  else
    Výstup: $x \notin S$, stop
  endif
endif
$x' := x \div 2^{k''}$, $x'' := x \bmod 2^{k''}$
if x' v TOP_S **then**
  if x'' v $BOTTOM_S(x')$ **then**
    Výstup: $x \in S$
  else
    Výstup: $x \notin S$
  endif
else
  Výstup: $x \notin S$
endif

MIN
if $|S| = 0$ **then** neexistuje, stop **endif**
if $|S| = 1$ **then** prvek v t_S, stop **endif**
$y' := \mathrm{MIN}(TOP_S)$, $y'' :=$ **MIN**$(BOTTOM_S(y'))$
$y := y'2^{k''} + y''$

MAX
if $|S| = 0$ **then** neexistuje, stop **endif**
if $|S| = 1$ **then** prvek v t_S, stop **endif**
$y' := \mathrm{MAX}(TOP_S)$, $y'' :=$ **MAX**$(BOTTOM_S(y'))$
$y := y'2^{k''} + y''$

Algoritmy pro operace **DELETEMIN** a **DELETEMAX** se vlastně zjednodušší.

DELETEMIN
if $|S| = 1$ **then** odstraníme t_S a $|S|$, stop **endif**
$y :=$ **MIN**(TOP_S), **DELETEMIN** v $BOTTOM_S(y)$
if $BOTTOM_S(y) = NIL$ **then**
  DELETEMIN(TOP_S)
endif
$|S| := |S| - 1$
if $|S| = 1$ **then**
  vytvořme seznam t_S obsahující prvek **MIN**(S)
  zruš TOP_S a $BOTTOM_S$
endif

DELETEMAX
if $|S| = 1$ **then** odstraníme t_S a $|S|$, stop **endif**
$y := \mathbf{MAX}(TOP_S)$, **DELETEMAX** v $BOTTOM_S(y)$
if $BOTTOM_S(y) = NIL$ **then**
DELETEMAX(TOP_S)
endif
$|S| := |S| - 1$
if $|S| = 1$ **then**
vytvořme seznam t_S obsahující prvek **MIN**(S)
zruš TOP_S a $BOTTOM_S$
endif

Při operaci **INSERT** se mění instrukce pro $|S| = 1$, vytvoří se jen seznam t_S pro množinu S a vynechají se příkazy pro P_S a když $|S| > 1$ tak se vynechají příkazy pro t_S a P_S. V případě, že po úspěšné operaci **INSERT** je $|S| = 2$, tak se zruší seznam t_S. Algoritmus pro operaci **DELETE**, když má odstranit prvek $x \in S$, tak když $|S| > 1$, odstraní prvek $x \bmod 2^{k''}$ ze struktury $BOTTOM_S(x \div 2^{k''})$, a pak postupuje stejně jako v algoritmech pro **DELETEMIN** a **DELETEMAX**. Algoritmy pro **SUCCESSOR** a **PREDECESSOR** se nemění. Stejný argument jako pro klasickou k-strukturu zajistí, že délka rekurzivních volání sama sebe je $\lceil \log k \rceil = O(\log \log |U|)$.

Tedy zbývá spočítat nároky na paměť. Zde má rekurze jiný tvar. Označme $\sigma(n)$ prostor potřebný pro vytvoření modifikované k-struktury, když velikost univerza je $n = 2^k$. Označme $k' = \lceil \frac{k}{2} \rceil$ a $k'' = \lfloor \frac{k}{2} \rfloor$. Pak platí:

$$\sigma(n) = \sigma(2^k) = \sigma(2^{k'}) + 2^{k'}\sigma(2^{k''}) + k$$

protože zápis velikosti $|S|$ vyžaduje nejvýše k bitů. Když budeme předpokládat, že b a c jsou kladné konstanty takové, že $b+1 < c$ a $\sigma(|U|) \leq b|U| - c$ pro $|U| = 2$. Pak dostaneme, že $(c-b)2^{k'} \geq 2^{k'} > k$ a tedy bude platit

$$\begin{aligned}\sigma(n) =& \sigma(2^{k'}) + 2^{k'}\sigma(2^{k''}) + k \leq \\ & b2^{k'} - c + b2^{k'}2^{k''} - c2^{k'} + k = \\ & bn + (b-c)2^{k'} + k - c \leq bn - c.\end{aligned}$$

Teď se jedná o splnění iniciálních podmínek. Když $|U| = 2$, pak reprezentace $S \subseteq U$ vyžaduje nejvýše 8 bitů a tedy stačí položit $b = 9$ a $c = 10$, pak $2b - c \geq 8$. Tedy můžeme shrnout

Věta. *Modifikovaná k-struktura reprezentuje podmnožiny univerza $U = \{0, 1, \ldots, n-1\}$, kde $n = 2^k$ pro nějaké přirozené číslo k, a vyžaduje $O(n)$ paměti. Pak operace* **MEMBER**, **MIN**, **MAX**, **INSERT**, **DELETE**, **DELETEMIN**, **DELETEMAX**, **SUCCESSOR** a **PREDECESSOR** *vyžadují čas $O(\log \log n)$.*

Důležitá otázka je, zda lze tuto strukturu modifikovat tak, aby vyžadovala $O(|S|)$ prostoru místo $O(|U|)$. Odpověď je že to lze, ale časová složitost bude horší. Základní idea je, že se

nahradí struktury TOP_S a $BOTTOM_S$ hašovací tabulkou (která odkazuje na neprázdné struktury $BOTTOM_S(i)$), ale to se ještě musí kompresovat. Výsledek pak je struktura, která sice vyžaduje $O(n)$ prostoru a operace **MIN**, **MAX** vyžadují $O(\log\log|U|)$ času, ale operace **MEMBER**, **SUCCESSOR** a **PREDECESSOR** mají jen očekávanou složitost rovnou $O(\log\log|U|)$ a operace **INSERT** a **DELETE** mají jen očekávanou amortizovanou složitost $O(\log\log|U|)$.

Tuto datovou strukturu navrhl van Emde Boas v roce 1977. Modifikovanou verzi navrhli van Emde Boas, Kaas a Zijlstra také v roce 1977.

Dvoukoncové haldy

Hodně často používanou strukturou jsou haldy. V zimním semestru jsme si ukázali 'jednokoncové' haldy, mohli jsme pracovat buď s minimem nebo maximem. Ukážeme si teď 'dvoukoncové' haldy, které jsou zobecněním d-regulárních hald (takto lze zobecnit i jiný typ hald než regulární haldy). Tyto haldy budou podporovat následující operace:
MIN, **MAX**, **DELETEMIN**, **DELETEMAX**, **DECREASEKEY**, **INCREASEKEY**, **INSERT**, **DELETE**.
Při operacích **DECREASEKEY**, **INCREASEKEY** a **DELETE** zadavatel úlohy dává adresu prvku, s nímž máme pracovat. Při operaci **INSERT** předpokládáme, že zadavatel zajistil, aby vkládaný prvek v haldě nebyl reprezentován.

Dále budeme uvažovat jen haldy, které jsou buď strom nebo soubor stromů. Řekneme, že halda je min-halda (resp. max-halda) když vrchol reprezentuje vždy prvek větší (resp. menší) nebo roven než je prvek reprezentovaný otcem. Budeme předpokládat, že min-halda podporuje operace spojené s minimem a nikoliv operace spojené s maximem, kdežto max-halda podporuje operace spojené s maximem a nikoliv s minimem. Naším cílem bude zkonstruovat haldu podporující operace jak s minimem tak s maximem. Dále předpokládáme, že haldy podporují operace **INSERT**, **INCREASEKEY**, **DECREASEKEY**, a **DELETE**.

Dvoukoncové regulární haldy.
Máme univerzum U, podmnožinu $S \subseteq U$ a hodnotící funkci f z S do lineárně uspořádané množiny. Popíšeme min-max-haldu reprezentující S a funkci f. Bude tvořena d-regulární min-haldou a stejně velkou d-regulární max-haldou a proměnnou $free$.

Když $|S|$ je sudé, pak $free$ je prázdné, když $|S|$ je liché, pak $free$ obsahuje prvek z množiny S takový, že $\min S = \min S \setminus \{free\}$ a $\max S = \max S \setminus \{free\}$. Dále máme disjunktní množiny S_1 a S_2 takové, že $S \setminus \{free\} = S_1 \cup S_2$. Tedy $|S_1| = |S_2| = \lfloor \frac{|S|}{2} \rfloor$, Nechť T_1 je d-regulární min-halda reprezentující S_1 a T_2 je d-regulární max-halda reprezentující S_2 takové, že:

(•) pro každé i je prvek $s \in S_1$ reprezentovaný i-tým listem v haldě T_1 menší než prvek $s' \in S_2$ reprezentovaný i-tým listem v haldě T_2.

Předpokládáme, že listy jsou očíslovány v lexikografickém uspořádaní binární haldy.

Nyní popíšeme operace v této haldě. Budeme používat operace definované v zimním semestru pro binární haldu, jen budeme rozlišovat, jestli jsou pro min-haldu (to je halda

T_1) nebo max-haldu (to je T_2). Rozdíl je v pomocné operaci **D-DOWN**, která může projít oběma haldami. V algoritmu pro operaci **DELETE** použijeme tuto proceduru **D-DOWN** místo procedury **DOWN** pro jednokoncové haldy.

Algoritmy **UP** a **DOWN** pro max-haldu budeme nazývat **M-UP** a **M-DOWN**. Procedura **M-UP** vyměňuje prvky reprezentované vrcholem a jeho otcem, když otec reprezentuje menší prvek a proceduru **M-DOWN** vyměňuje prvek reprezentovaný vrcholem s největším prvkem reprezentovaný jeho syny, pokud tento prvek je menší. Procedura **D-DOWN**(x) se liší podle toho, zda x je v min-haldě nebo max-haldě. Když x je v min-haldě, tak provede **DOWN**(x) a když skončí v i-tém listě, tak porovná prvky reprezentované v i-tém listě min-haldy a v i-tém listě max-haldy. Pokud v min-haldě i-tý list reprezentuje větší prvek, tak je vymění a provede na i-tý list max-haldy proceduru **M-UP**. Když x je v max-haldě tak provede **M-DOWN**(x). Když skončí v i-tém listě, tak porovná prvky reprezentované v i-tém listě max-haldy a v i-tém listě min-haldy. Pokud v min-haldě i-tý list reprezentuje větší prvek, tak je vymění a provede na i-tý list min-haldy proceduru **UP**. Další operace jsou stejné jako v d-regulárních haldách.

D-DOWN(x)
if x je prvek v T_1 **then**
  DOWN(x, T_1)
  if x je reprezentován i-tým listem T_1 **then**
    y je reprezentován i-tým listem T_2
    if $f(x) > f(y)$ **then**
      i-tý list T_1 reprezentuje y, i-tý list T_2 reprezentuje x
      M-UP(x, T_2)
    endif
  endif
else
  M-DOWN(x, T_2)
  if x je reprezentován i-tým listem T_2 **then**
    y je reprezentován i-tým listem T_1
    if $f(x) < f(y)$ **then**
      i-tý list T_1 reprezentuje x, i-tý list T_2 reprezentuje y
      UP(x, T_1)
    endif
  endif
endif

DELETEMIN
if $free \neq \emptyset$ **then**
  key(kořen T_1) := $free$, **D-DOWN**(kořen T_1), $free := \emptyset$
else
  DELETEMIN(T_1)
  $free$:= prvek reprezentovaný posledním listem T_2
  DELETE(poslední list T_2)
endif

DELETEMAX
if $free \neq \emptyset$ **then**
  key(kořen T_2) := $free$, **D-DOWN**(kořen T_2), $free := \emptyset$
else
  DELETEMAX(T_2)
  $free$:= prvek reprezentovaný posledním listem T_1
  DELETE(poslední list T_1)
endif

MIN
Výstup: prvek reprezentovaný kořenem T_1

MAX
Výstup: prvek reprezentovaný kořenem T_2

INSERT(x)
$y := key$(kořen T_1), $z :=$ key(kořen T_2)
if $x < y$ **then**
  vyměníme x a y
endif
if $z < x$ **then**
  vyměníme x a z
endif
if $free = \emptyset$ **then**
  $free := x$
else
  $a := \min\{x, free\}$, $b := \max\{x, free\}$
  INSERT(a, T_1), **INSERT**(b, T_2)
  $free := \emptyset$
endif

DECREASEKEY(x, a)
$f(x) := a$
if $x \in T_1$ **then**
  UP(x, T_1)
else
  D-DOWN(x, T_2)
endif

INCREASEKEY(x, a)
$f(x) := a$
if $x \in T_2$ **then**
  UP(x, T_2)
else

D-DOWN(x, T_1)
endif

DELETE(x)
x je reprezentován v T_i, $j := 3 - i$
DELETE(x, T_i)
if $free \neq \emptyset$ **then**
$y :=$ prvek reprezentovaný posledním listem T_j
DELETE(poslední listT_j)
if $i = 1$ a $free \leq y$ nebo $i = 2$ a $y \leq free$ bf then
INSERT$(free, T_i)$
else
INSERT(y, T_i), **Insert**$(free, T_j)$
endif
$free := \emptyset$
else
$free :=$ prvek reprezentovaný posledním listem T_j
DELETE(poslední list T_j)
endif

Je lehké vidět, že tuto strukturu lze reprezentovat pomocí dvou stejně velkých polí reprezentujících d-regularní haldy T_1 a T_2 a proměnnou free. Algoritmy využijí znalosti umístění synů a otce v tomto poli.

Shrneme získané výsledky.

Věta. *V navrhnuté min-max haldě operace* **MIN** *a* **MAX** *vyžadují čas* $O(1)$*, operace* **DELETEMIN**, **DELETEMAX**, **INSERT**, **DECREASEKEY**, **INCREASEKEY**, **DELETE** *vyžadují čas* $O(\log n)$*, kde* n *je velikost reprezentované množiny. Tuto haldu lze reprezentovat pomocí dvou polí.*

Lze ukázat, že očekávaný čas operace **INSERT** je $O(1)$ (stejně jako v regulárních haldách). Poněkud komplikovanější je algoritmus na konstrukci této haldy v lineárním čase.

Algoritmus pro vytvoření haldy popíšeme neformálně. Předpokládejme, že vstup je posloupnost prvků $a_1, a_2, \ldots, a_n$ spolu s hodnotící funkcí f. Když n je liché, pak nalezneme prostřední prvek mezi prvky a_1, a_2 a a_3. Tento prostřední prvek vyměníme s prvkem a_n a vložíme ho do proměnné $free$ a prvek a_n odstraníme. Tedy můžeme předpokládat, že $n = 2m$ je sudé. Nyní prvky $a_1, a_2, \ldots, a_m$ vložíme do pole Min a prvky $a_{m+1}, a_{m+2}, \ldots, a_n$ vložíme do pole Max. Pak položíme $k := \lfloor \frac{m-2}{d} \rfloor + 1$ a pro $i = k, k+1, \ldots, m$ porovnáme prvky $Min(i)$ a $Max(i)$ a pokud $f(Max(i)) < f(Min(i))$ tak je vyměníme. Dál budeme pro $i = k, k-1, \ldots, 1$ v klesajícím pořadí provádět modifikaci operací **D-DOWN**$(Min(i))$ a **D-DOWN**$(Max(i))$. Tato operace skončí, když se aplikuje ve struktuře T_1 a při provádění operace **M-UP** se dostala do prvku umístěného v poli $Max(l)$ pro $l < i$ nebo když se aplikuje ve struktuře T_2 a při provádění operace **UP** se dostala do prvku umístěného v poli $Min(l)$ pro $l < i$.

Tento postup zajistí, že když se provádí tyto operace pro i, pak pro každé $j > i$ platí, že $f(Min(j))$ je menší než hodnota f od synů vrcholu $Min(j)$ v haldě T_1 a $f(Max(j))$ je větší než hodnota f od synů vrcholu $Max(j)$ v haldě T_2. Proto po skončení pole Min reprezentuje min-haldu a pole Max reprezentuje max-haldu, které reprezentují disjunktní podmnožiny množiny S o velikosti $\lfloor \frac{n}{2} \rfloor$ a případně zbývající prvek je v proměnné $free$. Navíc, když vrchol i je ve výšce h, pak tato operace vyžaduje čas $O(h)$. Podle stejného výpočtu jako pro d-regulární haldy dostaneme, že tato operace vyžaduje čas $O(n)$.

Tuto strukturu (pro binární haldy) navrhl Williams v roce 1964, když definoval binární haldy. Podrobně jsou spočítané Knuthem v jeho monografii Art of Programming III.

Symetrická min-max halda.
Nyní popíšeme tzv. symetrickou min-max haldu. Binární halda T ohodnocená prvky z množiny S je symetrickou min-max haldou reprezentující množinu S, když platí:

(1) různé prvky množiny S ohodnocují různé vrcholy haldy a kořen haldy není ohodnocen žádným prvkem z S;
(2) pro každý vnitřní vrchol v, když s je ohodnocení levého syna vrcholu v a t je ohodnocení pravého syna vrcholu v, pak platí $s \leq u \leq t$ pro každý prvek u z S, který ohodnocuje některý vrchol w v podstromu určeném vrcholem v a $v \neq w$.

Tedy pro každý vrchol v stromu T platí: když S_v je množina prvků z S reprezentovaných nějakým vrcholem w v podstromu T určeném vrcholem v, $v \neq w$, pak levý syn vrcholu v reprezentuje minimum množiny S_v a pravý syn v reprezentuje maximum množiny S_v. Proto minimum množiny S je reprezentováno levým synem kořene a maximum je reprezentováno pravým synem kořene.

Operace jsou prakticky stejné jako v binární haldě, jen operace **UP**(x) a **DOWN**(x) jsou složitější. Nejprve popíšeme operaci **UP**(x). Když otec vrcholu reprezentujícího x je kořen, operace končí. V opačném případě, když x je menší než prvek reprezentovaný levým synem děda vrcholu reprezentujícího x, pak si x s tímto prvkem vymění místo. Když x je větší než prvek reprezentovaný pravým synem děda vrcholu reprezentujícího x, pak si x s tímto prvkem vymění místo. Když si x nevyměnilo místo s žádným prvkem, pak operace končí, v opačném případě se akce zopakuje s x.

Nyní popíšme operaci **DOWN**(x). Předpokládejme, že x je reprezentován vrcholem v. Když v je list, operace končí. Když v není list, pak setřídí množinu obsahující x a prvky reprezentované syny vrcholu v. Nyní upravíme reprezentaci prvků v této množině tak, aby platilo:

(•) když v je levým synem svého otce, pak reprezentuje nejmenší prvek z této množiny, levý syn v reprezentuje prostřední prvek množiny a pravý syn v reprezentuje největší prvek množiny;
(•) když v je pravým synem svého otce, pak levý syn v reprezentuje nejmenší prvek množiny, pravý syn v reprezentuje prostřední prvek množiny a v reprezentuje největší prvek množiny.

Když vrchol v nereprezentuje x, pak provedeme stejnou akci na vrchol reprezentující prvek x. Operace končí, když v reprezentuje x.

Formální popis těchto pomocných procedur:

UP(x)
$v :=$ vrchol reprezentující x
if otec(v) je kořen **then** stop **endif**
$y := key(\text{levy}(\text{ded}(v)))$
if $x < y$ **then**
  $key(\text{levy}(\text{ded}(v))) := x$, $key(v) := y$, **UP**(x)
endif
$z := key(\text{pravy}(\text{ded}(v)))$
if $z < x$ **then**
  $key(\text{pravy}(\text{ded}(v))) := x$, $key(v) := z$, **UP**(x)
endif

DOWN(x)
$v :=$ vrchol reprezentující x
if v je list **then** stop **endif**
$s := key(\text{levy}(v))$, $t := key(\text{pravy}(v))$
if v je levý syn svého otce **then**
  $key(v) := \min\{s, x, t\}$, $key(\text{pravy}(v)) := \max\{s, x, t\}$
  $key(\text{levy}(v)) :=$ median z množiny $\{s, x, t\}$
  if $key(v) \neq x$ **then DOWN**(x) **endif**
else
  $key(v) := \max\{s, x, t\}$, $key(\text{levy}(v)) := \min\{s, x, t\}$
  $key(\text{pravy}(v)) :=$ median z množiny $\{s, x, t\}$
  if $key(v) \neq x$ **then DOWN**(x) **endif**
endif

Algoritmy pro ostatní operace jsou stejné jako pro binární haldy.

Reflex-halda.
Na závěr si ukážeme metodu, jak zobecnit jednokoncovou haldu, která připouští jen operace **MIN**, **DELETEMIN**, **INSERT**, **MERGE**, **DECREASEKEY** a **INCREASEKEY** a nikoliv **MAX** a **DELETEMAX**, na dvoukoncovou haldu (tj. haldu, jež má také operace **MAX** a **DELETEMAX**). Předpokládejme, že máme 'stromovou' minhaldu $Q_{\min}$ (tj. provádí operace **MIN**, **DELETEMIN**, **INSERT**, **MERGE**, **DECREASEKEY** a **INCREASEKEY**) a halda používá jen ukazatele na syny a otce. Nechť $Q_{\max}$ je duální halda, tj. reprezentuje operace **MAX**, **DELETEMAX**, **INSERT**, **MERGE**, **DECREASEKEY** a **INCREASEKEY**. Budeme předpokládat kvůli jednodušímu popisu, že používáme jen binární haldu, tj. ukazatelé jsou levy a pravy.

Reflex-halda je zobecněním min-max-haldy definované Williamsem. Reflex-halda reprezentující množinu S, kterou tady chceme ukázat, používá jednu $Q_{\min}$ o velikosti $\lfloor \frac{|S|}{2} \rfloor$ jednu $Q_{\max}$ o velikosti $\lfloor \frac{|S|}{2} \rfloor$ a proměnnou $free$. Když $|S|$ je sudé, pak $free$ je prázdné, když $|S|$ je liché pak $free$ obsahuje prvek z S takový, že $\min S = \min S \setminus \{free\}$ a $\max S =$

$\max S \setminus \{free\}$. Dále $Q_{\min}$ a $Q_{\max}$ reprezentují disjunktní množiny S_1 a S_2 takové, že $|S_1| = |S_2| = \lfloor \frac{|S|}{2} \rfloor$ a $S_1 \cup S_2 = S \setminus \{free\}$. Navíc pro každý vrchol v haldy $Q_{\min}$ je dán ukazatel v_b na vrchol haldy $Q_{\max}$ a pro každý vrchol u haldy $Q_{\max}$ je dán ukazatel u_b na vrchol haldy $Q_{\min}$ takový, že pro každý vrchol v haldy $Q_{\min}$ platí $(v_b)_b = v$ a prvek reprezentovaný vrcholem v je menší než prvek reprezentovaný vrcholem v_b.

Operace **INSERT**(x) je velmi podobná operaci **INSERT** v haldě navržené Williamsem. Nejprve se zkontroluje, zda x není menší než nejmenší prvek nebo větší než největší. Pokud toto nastane, tak se x s příslušným prvkem vymění. Pak když $free$ je prázdné, tak se x vloží do $free$, když $free$ není prázdné, tak se v min-haldě i max-haldě vytvoří nový vrchol, tyto prvky budou spárovány ukazateli a vloží se na ně hodnoty x a $free$ (tím se $free$ vyprázdní) a operací **UP** se prvky přesunou na správné místo. Operace **DELETE**(x)**,** když $free$ je prázdné, tak přesune klíč spárovaného vrcholu do $free$ a upraví tvar haldy, když $free$ je neprázdné, tak hodnota $free$ nahradí odstraněný prvek a zase se operacemi **DOWN** a **UP** zajistí správný tvar haldy. Při operacích **UP** a **DOWN** se ukazatelé mezi prvky pohybují současně s prvky (nikoliv s vrcholy). To zajišťuje splnění podmínky, že vrchol min-haldy v reprezentuje prvek menší než vrchol v_b. Ostatní operace se provedou pomocí těchto operací.

Význam této metody je, že když máme haldu, která provede **INSERT** v konstantním čase, tak i tato halda bude vyžadovat konstantní čas a že analogicky lze provést operaci **MERGE**. To znamená

Věta. *Existuje-li min-halda, na kterou lze aplikovat tuto metodu a splňuje požadavky, pak vytvořená reflex-halda provede v nejhorším případě operace* **MIN**, **MAX**, **INSERT** *a* **MERGE** *v čase* $O(1)$ *a operace* **DELETE**, **DELETEMIN**, **DELETEMAX**, **DECREASEKEY**, **INCREASEKEY** *v čase* $O(\log n)$, *kde* n *je velikost reprezentované množiny.*

Metoda konstrukce reflex-haldy lze zobecnit i na haldy tvořené souborem stromů. V tomto zobecnění není nutné, aby každý vrchol min-haldy byl spojen s vrcholem max-haldy. Stačí jen, aby každý list min-haldy byl spojen s některým vrcholem max-haldy a každý list max-haldy byl spojen s některým vrcholem min-haldy, a aby platilo, že když vrchol v v min-haldě je spojen s vrcholem w v max-haldě, tak $f(\text{key}(v)) \leq f(\text{key}(w))$. Zde se může místo procedur **UP** a **DOWN** může použít metoda odtrhávání podstromů.

Symetrickou min-max haldu navrhli Arvind a Pandu Rangan v roce 1999, reflected min-max haldu navrhli Makris, Tsakalidis a Tsichlas v roce 2003.

Modifikace Fibonacciho hald

Při standardní implementaci Fibonacciho haldy má každý vrchol, alespoň čtyři ukazatele. Také se ukázalo, že operace **DECREASEKEY** má sice amortizovanou složitost $O(1)$, ale v nejhorším případě má složitost $\Omega(n)$. Proto multiplikativní konstanta schovaná v notaci O je velká, takže se nedoporučují pro praktické použití. To vedlo ke snaze nalézt haldy,

které jsou rychlejší, vyžadují méně paměti (přesněji, používají méně ukazatelů) a asymptoticky mají stejné chování jako Fibonacciho haldy. Tady ukážu modifikace Fibonacciho hald splňující tyto požadavky.

Tenká halda.

Tenká halda bude soubor kořenových stromů, kde každý vrchol v má definovaný rank(v) (rank(v) zde není počet synů vrcholu v, i když souvisí s tímto počtem). Je dána bijekce z vrcholů těchto stromů na reprezentovanou množinu taková, že pro každý vrchol v různý od kořene platí $f(v) \geq f(\text{otec}(v))$. Tedy kořen stromu má nejmenší ohodnocení mezi prvky reprezentované tímto stromem. Stromy jsou v jednosměrném seznamu a název haldy má ukazatel na první a poslední strom v tomto seznamu (aby šla dobře provádět konkatenace seznamů) a ukazatel na kořen stromu, který má nejmenší ohodnocení (aby operace **MIN** vyžadovala konstantní čas). Tedy se jedná o min-haldu.

Stromy budou reprezentovány pomocí seznamů synů jednotlivých vrcholů. Struktura vrcholů stromů tenké haldy:

- (•) ukazatel levy(v) má hodnotu NIL, když v je kořen, ukazuje na otce v, když v je první prvek v seznamu synů otce v, na předchůdce v v seznamu synů otce v, když v není ani kořen ani není první prvek v seznamu synů otce v;
- (•) ukazatel pravy(v) ukazuje na kořen následujícího stromu v seznamu stromů, když v je kořen, ale není poslední prvek v seznamu stromů, na prvek následující za vrcholem v v seznamu synů otce vrcholu v, když v není poslední prvek v seznamu synů otce v, a má hodnotu NIL, když v je poslední kořen v seznamu stromů nebo poslední prvek v seznamu synů otce v;
- (•) ukazatel prvni(v) ukazuje na první prvek v seznamu synů vrcholu v, pokud v není list a má hodnotu NIL, když v je list;
- (•) proměnnou key(v) dávající prvek reprezentovaný vrcholem v;
- (•) proměnnou rank(v) obsahující hodnotu ranku vrcholu v.

Vrcholy stromů budou splňovat následující podmínky:

- (h1) když vrchol v má k synů, pak synové budou mít hodnotu ranku $k-1, k-2, \ldots, 0$ a syn vrcholu v s rankem i bude $(k-i)$-tým prvkem seznamu synů vrcholu v (tj. rank(prvni(v)) $= k-1$, a když w je syn vrcholu v, pak
  rank(levy(w)) $=$ rank(w) $+ 1$ když w není první prvek v seznamu synů vrcholu v,
  rank(w) $=$ rank(pravy(w)) $+1$, když w není poslední prvek v seznamu synů vrcholu v,
  rank(w) $= 0$ když w je poslední prvek v seznamu synů v);
- (h2) když vrchol v má k synů, pak bude mít rank k nebo $k+1$, v prvním případě to bude normální vrchol v druhém případě to bude tenký vrchol;
- (h3) kořen každého stromu bude normální vrchol.

Tedy každý vrchol stromu v má nejvýše tři ukazatelé. Tato struktura umožňuje v konstantním čase testovat, zda vrchol je kořen stromu (tj. zda levy(v) $= NIL$), zda je prvním synem svého otce (tj. zda prvni(levy(v)) $= v$) a zda je tenký (tj. zda rank(v) $= 1$ a prvni(v) $= NIL$ nebo rank(v) $=$ rank(prvni(v)) $+ 2$).

Rank stromu je rank jeho kořene. Pro dva stromy T_1 a T_2 o stejném ranku je definována operace **LINK**(T_1, T_2), která je spojí do jednoho stromu. Vezme strom, jehož kořen má větší ohodnocení a jeho kořen udělá prvním synem druhého stromu a rank kořene druhého stromu zvětší o 1 (viz analogické operace u binomiálních hald a Fibonacciho hald). Tato operace vyžaduje $O(1)$ času stejně jako pro binomiální nebo Fibonacciho haldy. Všimněme si, že když strom má rank i a všechny jeho vrcholy jsou normální, pak je izomorfní s binomiálním stromem H_i. Proto tenký vrchol, má podobný semantický význam jako označený vrchol ve Fibonacciho haldě.

Lemma. *Když v je vrchol stromu v tenkém haldě a má i synů, pak jeho podstrom má alespoň F_{i+2} vrcholů, kde F_i je i-té Fibonacciho číslo.*

Důkaz. Tvrzení dokážeme indukcí podle počtu synů. Když vrchol nemá syna, pak jeho podstrom má jeden vrchol a $F_2 = 1$. Když má jednoho syna, pak jeho podstrom má alespoň dva vrcholy a $F_3 = 2$. Dále si stačí uvědomit, když vrchol má k synů, pak jeho synové mají rank od $k-1$ do 0 a vrchol s rankem i má nejvýše i a alespoň $i-1$ synů. Tedy z indukce dostáváme, že podstrom má alespoň $1 + \sum_{i=0}^{k-1} F_{i+2} = \sum_{i=1}^{k+1} F_i = F_{k+2}$ vrcholů. □

Nyní popíšeme operace v tenkých haldách. Mějme dvě tenké haldy H_1 a H_2, které reprezentují disjunktní množiny. Operace **MERGE**(H_1, H_2) provede konkatenci seznamů stromů a aktualizuje ukazatele nově vytvořené haldy. Operace **INSERT**(x) vytvoří novou haldu reprezentující $\{x\}$ a pak provede na obě haldy operaci **MERGE**. Operace **MIN** získá výsledek použitím ukazatele na kořen stromu reprezentujícího prvek s nejmenším ohodnocením. Operace **DELETEMIN** stejně jako operace **MIN** nalezne prvek, který má odstranit a po jeho odstranění provede konzolidaci seznamu stromů a seznamu podstromů určených syny odstraněného vrcholu.

Popíšeme konzolidaci. Z Lemmatu plyne, že rank nemůže být větší než $1+1.44 \log n$. Tedy vytvořme pole O o velikosti $1 + \lceil 1.44 \log n \rceil$. Procházejme pole stromů a strom s rankem k se pokusme vložit na místo $O(k)$. Pak procházíme pole synů odstraněného vrcholu x. Když syn je tenký tak zmenšíme jeho rank o 1 a pak se ho pokusíme vložit na místo $O(k)$, kde k je jeho aktualizovaný rank. Pokus o vložení vrcholu v s rankem k je následující akce: Když je místo $O(k)$ volné, pak tam v vložíme, v opačném případě provedeme operaci **LINK** na podstrom určený vrcholem v a na podstrom určeným vrcholem v $O(k)$, nově získaný strom zkusíme vložit na místo $O(k+1)$ a místo $O(k)$ se uprázdní. Toto opakujeme dokud se nám nepodaří strom vložit do pole O.

Nyní popíšeme algoritmy pro operace **DECREASEKEY** a **INCREASEKEY**, které se výrazně liší od algoritmů pro Fibonacciho haldy. Nejprve popíšeme základní tělo obou algoritmů. Uvažujme algoritmus pro **DECREASEKEY**(v, d), tj. chceme zmenšit ohodnocení prvku reprezentovaného vrcholem v o hodnotu d.

(1) vrchol v není kořen a $f(\text{otec}(v)) \leq f(v) - d$, pak zmenšíme ohodnocení prvku reprezentovaného vrcholem v a skončíme;
(2) vrchol v je kořen, pak zmenšíme ohodnocení vrcholu v a pokud po $f(v) - d < f(w)$, kde na w je kořen stromu s nejmenším ohodnocením, máme na něj ukazatel, pak změníme tento ukazatel, aby ukazoval na vrchol v a skončíme;

(3) vrchol v není kořen a $f(\text{otec}(v)) > f(v) - d$, a nechť $y = \text{levy}(v)$, pak odstraníme vrchol v ze seznamu, pokud byl tenký vrchol zmenšíme jeho rank a vložíme ho do seznamu stromů v haldě a aktualizujeme ukazatel haldy na kořen stromu s nejmenším ohodnocením, požadavky na vrcholy v tenké haldě mohou být porušeny jen ve vrcholu y, proto na vrchol y provedeme vyvažovací operace.

Teď popíšeme algoritmus pro **INCREASEKEY**(v, d).

(1) pokud $f(v) + d \leq \min\{f(w) \mid w \text{ je syn } v\}$, pak zvětšíme $f(v)$ o d a skončíme;
(2) když $f(v)+d > \min\{f(w) \mid w \text{ je syn } v\}$, pak položíme $y := levy(v)$, každého syna w vrcholu v testujeme, zda je tenký, pokud ano provedeme $\text{rank}(w) := \text{rank}(w) - 1$, a odtrhneme jeho podstrom a vložíme ho do seznamu stromů, pak odtrhneme i vrchol v, zvětšíme jeho ohodnocení, položíme $\text{rank}(v) := 0$ a vložíme ho jako jednoprvkový strom do seznamu stromů; protože podmínka na vrcholy v tenké haldě může být porušena jen ve vrcholu y, tak provedeme na vrchol y vyvažovací operace.

Nyní popíšeme vyvažovací operaci, která je stejná jak pro operaci **DECREASEKEY** tak pro operaci **INCREASKEY**. Když je porušena podmínka (h1) z definice tenké haldy, pak buď $z = \text{pravy}(y) \neq NIL$ a platí $rank(y) - 2 = rank(z)$ nebo $\text{pravy}(y) = NIL$ a $rank(y) = 1$. Když y je normální vrchol, pak prvního syna vrcholu y odstraníme ze seznamu synů vrcholu y a vložíme ho do seznamu synů otce vrcholu y hned za vrchol y. Teď všechny vrcholy splňují podmínky na tenkou haldu (y je teď tenký vrchol), a proto končíme. Když y je tenký, pak zmenšíme jeho rank o 1. Nyní y splňuje podmínku (h1) na tenkou haldu, ale vrchol $\text{levy}(y)$ ji nemusí splňovat. Proto položíme $y := \text{levy}(v)$ a opakujeme vyvažovací operaci na vrchol y.

Když je porušena podmínka (h2) z definice tenké haldy, pak buď $\text{prvni}(y) \neq NIL$ a $\text{rank}(y) = \text{rank}(\text{prvni}(y)) + 3$ nebo $\text{prvni}(y) = NIL$ a $\text{rank}(y) = 2$. Pak y není kořen stromu. Položme $z := \text{levy}(y)$, $\text{rank}(y) := \text{rank}(y) - 2$ a odtrhneme y ze seznamu synů otce y a vložíme ho do seznamu stromů. Nyní podmínky na tenkou haldu mohou být porušeny jen ve vrcholu z, proto položíme $y := z$ a opakujeme vyvažovací operaci na vrchol y.

Když je porušena podmínka (h3) z definice tenké haldy, pak y je kořen stromu, ale je tenký. Nyní stačí položit $\text{rank}(y) := \text{rank}(y) - 1$ a skončit.

Položme potenciální funkci rovnou počtu stromů plus dvojnásobku počtu tenkých vrcholů. Pak stejnou analýzou jako pro Fibonacciho haldy dostaneme, že amortizovaná složitost operací **MERGE**, **INSERT**, **DECREASEKEY** a **MIN** je $O(1)$ a amortizovaná složitost operací **DELETEMIN INCREASEKEY** a **DELETE** je $O(\log n)$, (protože počet synů vrcholu v v nějaké stromu tenké haldy, je podle Lemmatu nejvýše $\lfloor 1.44 \log n \rfloor$), kde operace **DELETE**(x) se provede tak, že pomocí operace **DECREASEKEY** zmenšíme ohodnocení x na $-\infty$ a pak provedeme **DELETEMIN**.

Protože při testování, zda jsou splněné podmínky pro tenkou haldu, rank vrcholu y roste, tak operace **DECREASEKEY** v nejhorším případě vyžaduje čas $O(\log n)$ – z Lemmatu plyne, že rank je $O(\log n)$ (u Fibonacciho hald vyžaduje čas $\Omega(n)$). To ukazuje, že tato modifikace asi bude rychlejší než Fibonacciho haldy.

Když není splněna podmínka (h2), to znamená, že v tom případě platí buď prvni$(y) \neq NIL$ a rank(y) = rank(prvni(y)) + 3 nebo prvni$(y) = NIL$ a rank$(y) = 2$, pak opravu tenké haldy lze provést ještě jiným způsobem, protože pravy$(y) = z \neq NIL$. Když z je normální vrchol, pak položíme rank(y) := rank$(y) - 1$, rank(z) := rank$(z) + 1$ a vrcholy y a z vyměníme v seznamu synů otce y a končíme (nyní jsou oba vrcholy y a z tenké). Když z je tenký vrchol, pak položíme u := levy(y), rank(y) := rank$(y) - 2$, rank(z) := rank$(z) - 1$ a provedeme operaci **LINK** na podstromy vrcholů y a z. Kořen vzniklého stromu pak vložíme do seznamu synů otce y místo vrcholů y a z. Podmínky na tenkou haldu mohou byt porušeny jen ve vrcholu u, proto provedeme vyvažovací operaci na vrchol u.

Když teď položíme potenciálovou funkci rovnou součtu počtu stromů a počtu tenkých vrcholů, pak dostaneme, že tato verze tenké haldy má stejnou amortizovanou složitost jako původní verze.

Tlustá halda.

Obecně se věří, že když stromy budou mít menší výšku, tak datové struktury budou efektivnější. To vedlo k modifikaci podmínek na haldy, aby počet následovníků vrcholu rostl rychleji než počet synů. Důsledkem bylo, že podmínka (2) v tenkých haldách byla nahrazena podmínkou

(h2a) když vrchol v má k synů, pak rank$(v) \in \{k-1, k\}$.

Lemma. *Když kořenový strom splňuje podmínky (h1), (h2a) a (h3), pak podstrom každého jeho vrcholu v, který má k synů, má alespoň 2^k vrcholů.*

Důkaz. Tvrzení dokážeme indukcí podle ranku. Tvrzení platí pro každý list a pro každý vrchol, který má jen jednoho syna. Tedy pokud rank$(v) = 1$, pak pro vrchol v tvrzení platí. Předpokládejme, že když strom splňuje podmínky (h1), (h2a) a (h3), pak tvrzení platí pro vrcholy v, takové, že rank$(v) < k$ nebo rank$(v) = k$ a výška v je menší než l, pro $k > 1$ a $l \geq 1$. Mějme strom T, který splňuje (h1), (h2a) a (h3) a v něm vrchol v takový, že rank$(v) = k$ a výška v je l. Nechť u je první syn vrcholu v, pak u má alespoň $k - 1$ synů. Nechť T_u je podstrom stromu T určený vrcholem u. Pak $k - 1 \leq$ rank$(u) \leq k$ a výška u je menší než l (v obou stromech T i T_u). Tedy u splňuje indukční předpoklady, a proto T_u má alespoň 2^{k-1} vrcholů. Ve stromu $T \setminus T_u$ snížíme rank(v) o 1, pak strom $T \setminus T_u$ také splňuje podmínky (h1), (h2a) a (h3) a vrchol v ve stromě $T \setminus T_u$ splňuje indukční hypotézu. Tedy podstrom stromu $T \setminus T_u$ určený vrcholem v má alespoň 2^{k-1} vrcholů. Protože $2^k = 2^{k-1} + 2^{k-1}$, podstrom stromu T určený vrcholem v má alespoň 2^k vrcholů. Tedy tvrzení je dokázáno. □

Proto definujeme tlustou haldu jako jednosměrný seznam stromů splňujících podmínky (h1), (h2a) a (h3) a podmínku, že $f(v) \geq f($otce $v)$ pro každý vrchol v. Řekneme, že v je tlustý vrchol, když rank(v) = rank(prvni(v)), jinak v je normální vrchol. Reprezentace stromů je stejná jako pro tenké haldy.

Operace **INSERT**, **MERGE**, **MIN** a **DELETEMIN** jsou stejné jako pro tenké haldy, algoritmus pro operaci **DECREASEKEY** kopíruje druhý algoritmus pro tenké haldy.

Popíšeme vyvažující operaci.

(1) Vrchol y porušuje podmínku (h1). Pak buď $\text{rank}(y) = \text{rank}(\text{pravy}(y)) + 2$ nebo $\text{rank}(y) = 1$ a $\text{pravy}(y) = NIL$.
  (1A) Když y je tlustý vrchol, pak vrchol $\text{prvni}(y)$ odstraníme ze seznamu synů vrcholu y a vložíme ho do seznamu synů otce y před vrchol y a $\text{rank}(y)$ snížíme o 1 a skončíme.
  (1B) Když y je normální vrchol, pak snížíme $\text{rank}(y)$ o 1 a položíme $y := \text{levy}(y)$ a provedeme vyvažovací operace na vrchol y.
(2) Vrchol y porušuje podmínku (h2a). Tedy $\text{rank}(y) = \text{rank}(\text{prvni}(y)) + 2$ nebo $\text{rank}(y) = 1$ a $\text{prvni}(y) = NIL$. Pak y je buď kořen nebo $\text{pravy}(y) \neq NIL$.
  (2A) Když y je kořen stromu, pak snížíme $\text{rank}(y)$ o 1 a končíme.
  (2B) Když y není kořen a $w := \text{pravy}(y)$ je tlustý. Pak snížíme $\text{rank}(y)$ o 1, zvětšíme $\text{rank}(w)$ o 1, vyměníme y a w v seznamu synů otce vrcholu y a končíme.
  (2C) Když y není kořen a $w := \text{pravy}(y)$ je normální. Pak položíme $z := \text{levy}(y)$, snížíme $\text{rank}(y)$ o 1 a provedeme **LINK** na podstromy určené vrcholy y a w a kořen vzniklého stromu nahradí v seznamu synů otce y vrcholy y a w a pak položíme $y := z$ a provedeme test na vrchol y.
(3) Vrchol y porušuje podmínku (h3). Zvětšíme $\text{rank}(y)$ o 1 a končíme.

Zde hlavní rozdíl je v definici potenciálové funkce. Ohodnocení je počet stromů plus dvakrát počet normálních vrcholů, které nejsou kořenem stromu. Pak se úplně stejně jako pro tenké haldy dostane

Věta. *V tlusté haldě popsané algoritmy mají amortizovaný čas* $O(1)$ *pro operace* **MIN**, **INSERT**, **MERGE**, **DECREASEKEY**, *amortizovaný čas* $O(\log(n)$ *pro operace* **DELETEMIN** *a* **DELETE**. *Čas v nejhorším případě je O(1) pro operace* **MIN**, **INSERT**, **MERGE** *a čas* $O(\log n)$ *pro operaci* **DECREASEKEY**.

Operace **DELETE**(x) provedeme příkazy

DELETEMIN$(x, -\infty)$, **DELETEMIN**

Operace **DELETEMIN** a **DELETE** vyžadují v nejhorším případě čas $O(n)$. To lze vylepšit tak, že budeme mít zaznamenán počet stromů a při každé operaci, když počet stromů převýší $\log n$ provedeme konsolidaci. Pak všechny operace, kromě operace **MIN** vyžadují v nejhorším případě čas $O(\log n)$ a amortizovaná složitost se nemění. To lze ještě vylepšit tak, že i operace **INSERT** bude v nejhorším případě vyžadovat čas $O(1)$. Stačí, když seznam stromů rozdělíme podle ranku stromů a stromy se stejným rankem budou tvořit disjunktní dvojice s vyjímkou posledního "lichého" stromu. Nyní při operaci **INSERT** přidávaný strom zařadíme do seznamu (buď ho spárujeme s lichým stromem nebo ho přidáme jako lichý strom). Pokud stromů bude více než $\log n$, provedeme na jednu dvojici **LINK** a vzniklý strom vrátíme do seznamu (buď jako lichý strom nebo ho sparujeme s lichým stromem). Tedy platí

Věta. *V prezentovaných haldách operace* **MIN** *a* **INSERT** *v nejhorším případě pracují v čase* $O(1)$, *operace* **MERGE**, **DECREASEKEY**, **DELETEMIN** *a* **DELETE** *vyžadují*

v nejhorším případě čas $O(\log n)$ *a navíc operace* **DECREASEKEY** *a* **MERGE** *mají amortizovanou složitost* $O(1)$.

Operaci **INCREASEKEY** lze provést stejně jako v tenkých haldách. Bude mít amortizovanou složitost i časovou složitost v nejhorším případě $O(logn)$ a při rovnoměrném rozdělení vstupů lze ukázat, že očekávaná amortizovaná složitost je $O(1)$.

Tuto modifikaci Fibonacciho hald navrhli Kaplan a Tarjan v roce 2008.