V praxi se často setkáváme s následujícím problémem, který je modifikací slovníkového problému.
Zadání problému: $U$ je univerzum. Je dána množina $S \subseteq U$ a funkce $f : S \to \mathbb{R}$, kde $\mathbb{R}$ jsou reálná čísla. Máme navrhnout reprezentaci $S$ a $f$, která umožňuje operace:

**INSERT**$(s, a)$ – přidá k množině $S$ prvek $s$ tak, že $f(s) = a$;
**MIN** – nalezne prvek $s \in S$ s nejmenší hodnotou $f(s)$;
**DELETEMIN** – odstraní prvek $s \in S$ s nejmenší hodnotou $f(s)$;
**DELETE**$(s)$ – odstraní prvek $s \in S$;
**DECREASE**$(s, a)$ – zmenší hodnotu $f(s)$ o $a$ (tj. $f(s) := f(s) - a$);
**INCREASE**$(s, a)$ – zvětší hodnotu $f(s)$ o $a$ (tj. $f(s) := f(s) + a$).

Při operaci **INSERT**$(s, a)$ se předpokládá, že $s \notin S$, tento předpoklad operace **INSERT** neověřuje. Při operacích **DELETE**$(s)$, **DECREASE**$(s, a)$ **a INCREASE**$(s, a)$ se předpokládá, že $s \in S$ a operace navíc dostává informaci, jak nalézt $s$ a s ním spojená data v reprezentaci $S$ a $f$. Haldy jsou typ struktury, která se používá pro řešení tohoto problému.

Halda je stromová struktura, kde vrcholy reprezentují prvky z $S$ a splňují lokální podmínku na $f$. Obvykle se používá následující podmínka nebo její symetrická verze:

(usp) Pro každý vrchol $v$, když $v$ reprezentuje prvek $s \in S$ a otec$(v)$ reprezentuje $t$, pak $f(t) \leq f(s)$.

Probereme několik verzí hald a budeme předpokládat, že vždy splňují tuto podmínku a že operace **DELETE**$(s)$, **DECREASE**$(s, a)$ a **INCREASE**$(s, a)$ zadávají také ukazatel na vrchol reprezentující $s \in S$. Navíc budeme uvažovat operace

**MAKEHEAP**$(S, f)$ – operace vytvoří haldu reprezentující množinu $S$ a funkci $f$.
**MERGE**$(H_1, H_2)$ – předpokládáme, že halda $H_i$ reprezentuje množinu $S_i$ a funkci $f_i$ pro $i = 1, 2$ a $S_1 \cap S_2 = \emptyset$. Operace vytvoří haldu $H$ reprezentující $S_1 \cup S_2$ a $f_1 \cup f_2$. Operace neověřuje disjunktnost $S_1$ a $S_2$.

## Regulární haldy

Předpokládejme, že $d > 1$ je přirozené číslo. $d$-regulární strom je kořenový strom $(T, r)$ takový, že existuje pořadí synů jednotlivých vnitřních vrcholů takové, že očíslování vrcholů prohledáváním do šířky (kořen $r$ je číslován 1) splňuje

(1) každý vrchol má nejvýše $d$ synů;
(2) když vrchol není list, tak všechny vrcholy s menším číslem mají právě $d$ synů;
(3) když vrchol má méně než $d$ synů, pak všechny vrcholy s větším číslem jsou listy.

Toto očíslování se nazývá přirozené očíslování $d$-regulárního stromu. přirozené očíslování budeme označovat $o$, to znamená, že pro vrchol $v$ je $o(v)$ jeho číslo/pořadí v tomto očíslování.

**Tvrzení.** *$d$-regulární strom má nejvýše jeden vnitřní vrchol, který má méně než $d$ synů. Když $d$-regulární strom má $n$ vrcholů, pak jeho výška je $\lceil \log_d (n(d-1)+1) \rceil - 1$. Když pro vrchol $v$ je $o(v) = k$, pak vrchol $w$ je syn vrcholu $v$, právě když $o(w) \in \{(k-1)d + 2, (k-1)d+3, \dots, kd+1\}$, a vrchol $u$ je otcem vrcholu $v$, právě když $o(u) = 1 + \lfloor \frac{k-2}{d} \rfloor$.*

Řekneme, že množina $S$ s funkcí $f$ je reprezentována $d$-regulární haldou $H$ a bijekcí key, když $H$ je $d$-regulární strom $(T, r)$ a bijekce key z vrcholů stromu $T$ na množinu $S$ splňuje podmínku (usp).

Implementace $d$-regulární haldy $H$. Nechť $o$ je přirozené očíslování $d$-regulárního stromu $(T, r)$, pak halda $H$ je reprezentovaná polem $H[1..|S|]$, kde $H(i) = (\text{key}(v), f(\text{key}(v)))$ a $o(v) = i$. Algoritmy budeme popisovat pro stromy, protože je to názornější. Přeformulovat je pro pole, je snadné. Pro jednoduchost zápis $f(v)$ pro vrchol $v$ označuje $f(s)$, kde $s \in S$ je reprezentován vrcholem $v$. U $d$-regulárního stromu předpokládáme, že známe přirozené očíslování a fráze 'poslední vrchol', 'předcházející vrchol' atd. se vztahují k tomuto očíslování.

Není známa efektivní implementace operace **MERGE** pro $d$-regulární haldy. Efektivní implementace ostatních operací jsou založeny na dvou pomocných operacích **UP**$(v)$ a **DOWN**$(v)$. Operace **UP**$(v)$ posunuje prvek $s$ reprezentovaný vrcholem $v$ směrem ke kořeni, dokud vrchol reprezentující prvek $s$ nesplňuje podmínku (usp). Na druhou stranu operace **DOWN**$(v)$ posunuje prvek $s$ směrem k listům dokud není splněna podmínka (usp).

**Algoritmy.**

**UP**$(v)$:
**while** $v$ není kořen a $f(v) < f(\text{otec}(v))$ **do**
vyměň $\text{key}(v)$ a $\text{key}(\text{otec}(v))$, $v := \text{otec}(v)$
**enddo**

**DOWN**$(v)$:
**if** $v$ není list **then**
$w$ :=syn vrcholu $v$ reprezentující prvek s nejmenší hodnotou $f(w)$
**while** $f(w) < f(v)$ a $v$ není list **do**
vyměň $\text{key}(v)$ a $\text{key}(w)$, $v := w$
$w$ :=syn vrcholu $v$ reprezentující prvek s nejmenší hodnotou $f(w)$
**enddo**
**endif**

**INSERT**$(s)$:
$v$ :=nový poslední list, $\text{key}(v) := s$, **UP**$(v)$

**MIN**:
**Výstup** key (kořen $T$)

**DELETEMIN**:
$v$ :=poslední list, $r$ :=kořen
$\text{key}(r) := \text{key}(v)$, odstraň $v$
**DOWN**$(r)$

**DELETE**$(s)$,
$v$ :=vrchol reprezentující $s$, $w$ :=poslední list
$\text{key}(v) := \text{key}(w)$, odstraň $w$
**if** $f(t) < f(s)$ **then UP**$(v)$ **else DOWN**$(v)$ **endif**

**DECREASE**$(s, a)$:
$v$ :=vrchol reprezentující $s$
$f(s) := f(s) - a$, **UP**$(v)$

**INCREASE**$(s, a)$:
$v$ :=vrchol reprezentující $s$
$f(s) := f(s) + a$, **DOWN**$(v)$

**MAKEHEAP**$(S,f)$:
$T := d$-regulární strom s $|S|$ vrcholy
zvolme libovolnou reprezentaci $S$ vrcholy stromu $T$
$v :=$poslední vrchol, který není list
**while** $v$ je vrchol $T$ **do**
**DOWN**$(v)$, $v :=$vrchol předcházející vrcholu $v$
**enddo**

**Korektnost algoritmů.**

U operace **INSERT** je podmínka (usp) splněna pro všechny vrcholy s výjimkou nově vytvořeného listu a operace **UP** zajistí její splnění. Při operaci **DELETEMIN** je podmínka (usp) splněna pro všechny vrcholy s výjimkou kořene a operace **DOWN** zajistí její splnění. U operací **DELETE**$(s)$, **DECREASE**$(s,a)$ a **INCREASE**$(s,a)$ je podmínka (usp) splněna pro všechny vrcholy s výjimkou vrcholu $v$ a provedení operace **UP** nebo **DOWN** zajistí její splnění. Pro operaci **MAKEHEAP** budeme uvažovat duální formulaci podmínky (usp):

(d-usp) když $s$ je prvek reprezentovaný vrcholem $v$, pak $f(s) \le f(t)$ pro všechny prvky reprezentované syny $t$ vrcholu $v$.

Když každý vrchol splňuje podmínku (d-usp), pak splňuje i podmínku (usp). Každý list splňuje podmínku (d-usp), a proto když algoritmus **MAKEHEAP** pracuje s vrcholem $v$, tak podmínku (d-usp) splňují všechny vrcholy $w$ takové, že $o(w) > o(v)$. Po provedení podprocedury **DOWN**$(v)$ je podmínka (d-usp) splněna i pro vrchol $v$ (tj. pro všechny vrcholy $w$ takové, že $o(w) \ge o(v)$). Odtud plyne korektnost algoritmu pro operaci **MAKEHEAP**, protože algoritmus končí provedením operace **DOWN** na kořen.

**Efektivita operací.**

Jeden běh cyklu v operaci **UP** vyžaduje čas $O(1)$ a v operaci **DOWN** čas $O(d)$. Proto v nejhorším případě operace **UP** vyžaduje čas $O(\log_d |S|)$ a **DOWN** čas $O(d \log_d |S|)$. Operace **MIN** zřejmě vyžaduje čas $O(1)$, operace **INSERT** a **DECREASE** vyžadují čas $O(\log_d |S|)$ a operace **DELETEMIN**, **DELETE** a **INCREASE** čas $O(d\log_d |S|)$. Haldu můžeme vytvořit i tak, že opakujeme $|S|$-krát operaci **INSERT**, to vyžaduje čas $O(|S| \log_d(|S|))$. Spočítáme složitost operace **MAKEHEAP**. Operace **DOWN**$(v)$ na vrchol ve výšce $h$ vyžaduje v nejhorším případě čas $O(hd)$. Vrcholů v hloubce $i$ je nejvýše $d^i$. Předpokládejme, že strom má výšku $k$, pak operace **MAKEHEAP** vyžaduje čas $O\left(\sum_{i=0}^{k-1} d^i (k-i) d\right) = O\left(\sum_{i=0}^{k-1} d^{i+1}(k-i)\right)$. Označme $A = \sum_{i=0}^{k-1} d^{i+1}(k-i)$, pak

$$
\begin{aligned}
dA - A = &\sum_{i=0}^{k-1} d^{i+2}(k-i) - \sum_{i=0}^{k-1} d^{i+1}(k-i) = \\
&\sum_{i=2}^{k+1} d^i (k-i+2) - \sum_{i=1}^{k} d^i (k-i+1) = \\
&d^{k+1} + \sum_{i=2}^{k} d^i (k-i+2-k+i-1) - dk = \\
&d^{k+1} + \sum_{i=2}^{k} d^i - dk = d^{k+1} + d^2 \frac{d^{k-1}-1}{d-1} - dk.
\end{aligned}
$$

Tedy $A = \frac{d^{k+1}}{d-1} + \frac{d^{k+1}-d^2}{(d-1)^2} - dk$. Protože $k = \lceil \log_d(|S|(d-1)+1) \rceil - 1$, dostáváme,

že $d^{k+1} \leq d^2\left((d-1)|S|+1\right)$, a proto $A \leq 2d^2|S|$. Tedy **MAKEHEAP** vyžaduje v nejhorším případě jen čas $O\left(d^2|S|\right)$.

**Aplikace: Třídění.**

Pro setřídění posloupnosti prvků můžeme použit následující algoritmus.

$d$-**HEAPSORT**$(x_1, x_2, \ldots, x_n)$:
**MAKEHEAP**$(\{x_i \mid i = 1, 2, \ldots, n\}, f)$
$i = 1$
**while** $i \leq n$ **do**
$y_i$ :=**MIN**, **DELETEMIN**, $i := i + 1$
**enddo**
**Výstup**: $y_1, y_2, \ldots, y_n$

Předpokládáme, že $x_1, x_2, \ldots, x_n$ je posloupnost čísel a $f$ v tomto případě bude identická funkce.

Teoreticky lze ukázat, že $d = 3$ a $d = 4$ jsou lepší než $d = 2$. Experimenty ukázaly, že optimální algoritmus pro posloupnosti délek do 1 000 000 by měl být $d = 6$ nebo $d = 7$.

**Nalezení nejkratších cest v grafu z daného bodu.**

Vstupem pro tuto aplikaci je orientovaný ohodnocený graf $(X, R, c)$ a vrchol $z \in X$, $c$ je funkce z $R$ do množiny kladných reálných čísel.
Úkol: nalézt pro každý bod $x \in X$ délku nejkratší cesty ze $z$ do $x$ – délka cesty je součet ohodnocení hran na dané cestě funkcí $c$.
Řešení:

**Dijkstrův algoritmus**
$d(z) := 0$, $U := \{z\}$
**for every** $x \in X \setminus \{z\}$ **do** $d(x) := +\infty$ **enddo**
**while** $U \neq \emptyset$ **do**
najdi vrchol $u \in U$ s nejmenší hodnotou $d(u)$, odstraň ho z $U$
**for every** $(u, v) \in R$ **do**
**if** $d(u) + c(u, v) < d(v)$ **then**
**if** $d(v) = +\infty$ **then** vlož $v$ do $U$ **endif**
$d(v) := d(u) + c(u, v)$
**endif**
**enddo**
**enddo**

Když $U = \emptyset$, pak $d(x)$ jsou délky nejkratších cest ze $z$ do $x$. Když $U$ reprezentujeme jako $d$-regulární haldu, pak se provede nejvýše $|X|$ operací **INSERT**, **MIN** a **DELETEMIN** a $|R|$ operací **DECREASE** a $|U| \leq |X|$. Pro $d = 2$ dostáváme, že algoritmus vyžaduje čas $O(|X| \log(|X|) + |R| \log(|X|))$. když položíme $d = \max\left\{2, \lfloor\frac{|R|}{|X|}\rfloor\right\}$, pak algoritmus vyžaduje čas $O(|R| \log_d |X|)$. Když $|R| > |X|^{1+\varepsilon}$ pro $\varepsilon > 0$, pak $\log_d |X| = O(1)$ a algoritmus je lineární (tj. vyžaduje čas $O(|R|)$).

## Leftist haldy

Mějme binární strom $(T, r)$, to znamená, že $r$ je kořen, každý vrchol má nejvýše dva syny a u každého syna víme, zda je to pravý nebo levý syn. Pro vrchol $v$ označme $\mathrm{npl}(v)$ délku

nejkratší cesty z $v$ do vrcholu, který má nejvýše jednoho syna. Takže např. list má npl rovno 0.

Mějme $S \subseteq U$ a funkci $f : S \to \mathbb{R}$. Pak binární strom $(T, r)$ takový, že

(1) když vrchol $v$ má jen jednoho syna, pak je to levý syn,
(2) když vrchol $v$ má dva syny, pak

$$\text{npl}\,(\text{pravy}\,(v)) \le \text{npl}\,(\text{levy}\,(v))\,,$$

(3) existuje jednoznačná korespondence mezi prvky z $S$ a vrcholy $T$, která splňuje podmínku (usp)

je leftist halda reprezentující množinu $S$ a funkci $f$.

Struktura vrcholu $v$:
ukazatelé otec $(v)$, levy $(v)$ a pravy $(v)$ na otce vrcholu $v$, na levého syna $v$ a na pravého syna $v$. Když ukazatel není definován, pak píšeme, že jeho hodnota je $NIL$;
npl $(v)$ – proměnná s hodnotou npl $(v)$;
key $(v)$ – prvek reprezentovaný vrcholem $v$;
$f(v)$ – proměnná obsahující hodnotu $f(\text{key}\,(v))$.

Základní vlastnost leftist haldy.
Posloupnost vrcholů $v_0, v_1, \ldots, v_k$ se nazývá pravá cesta z vrcholu $v$, když $v = v_0$, $v_{i+1}$ je pravý syn $v_i$ pro každé $i = 0, 1, \ldots, k-1$ a $v_k$ nemá pravého syna. Pak podstrom vrcholu $v$ do hloubky $k$ je úplný binární strom, a tedy podstrom vrcholu $v$ má alespoň $2^{k+1} - 1$ vrcholů. Tedy délka pravé cesty z vrcholu $v$ je

$$O\,(\log\,(\text{velikost podstromu určeného vrcholem } v))\,.$$

Základní operace pro leftist haldy je **MERGE**.

**Algoritmy.**

**MERGE**$(T_1, T_2)$:
**if** $T_1 = \emptyset$ **then Výstup**$= T_2$ konec **endif**
**if** $T_2 = \emptyset$ **then Výstup**$= T_1$ konec **endif**
**if** key (kořen $T_1$) $>$ key (kořen $T_2$) **then**
zaměň $T_1$ a $T_2$
**endif**
$T'$ :=**MERGE**(podstrom pravého syna kořene $T_1, T_2$)
pravy (kořen $T_1$) := kořen $T'$, otec (kořen $T'$) := kořen $T_1$
**if** npl (pravy (kořen $T_1$)) $>$ npl (levy (kořen $T_1$)) **then**
zaměň levého a pravého syna kořene $T_1$
**endif**
npl (kořen $T_1$) := npl (pravy (kořen $T_1$)) + 1

**INSERT**$(x)$:
Vytvoř haldu $T_1$ reprezentující $\{x\}$
**MERGE** $(T, T_1)$

**MIN**:
**Výstup**: key (kořen $T$)

**DELETEMIN**:
$T_1$ :=podstrom levého syna kořene $T$
$T_2$ :=podstrom pravého syna kořene $T$
**MERGE**$(T_1, T_2)$

**MAKEHEAP**$(S, f)$:
$Q$ :=prázdná fronta
**for every** $s \in S$ **do**
  vytvoř leftist haldu $T_s$ reprezentující $\{s\}$
  vlož $T_s$ do $Q$
**enddo**
**while** $|Q| > 1$ **do**
  vezmi haldy $T_1$ a $T_2$ z vrcholu $Q$ (odstraň je)
  **MERGE**$(T_1, T_2)$ vlož do $Q$
**enddo**

**Efektivnost algoritmů.**

Každý běh algoritmu **MERGE** (bez rekurzivního volání) vyžaduje čas $O(1)$. Počet rekurzivních volání je součet délek pravých cest, proto algoritmus **MERGE** vyžaduje čas $O(\log(|S_1| + |S_2|))$, kde $S_i$ je množina reprezentovaná haldou $T_i$ pro $i = 1, 2$. Proto algoritmy **INSERT** a **DELETEMIN** vyžadují v nejhorším případě čas $O(\log(|S|))$. Operace **MIN** vyžaduje čas $O(1)$. Pro odhad složitosti **MAKEHEAP** budeme uvažovat, že na začátku algoritmu je na vrcholu fronty speciální znak, který se jen přenese na konec fronty. Odhadneme čas, který potřebují **while-**cykly mezi dvěma přeneseními speciálního znaku. Předpokládejme, že se speciální znak přenesl $k$-krát. V tomto okamžiku, až na jednu haldu, všechny haldy ve frontě mají velikost $2^k$. Proto ve frontě $Q$ je $\left\lceil \frac{|S|}{2^k} \right\rceil$ hald a každá operace **MERGE** vyžaduje $O(k)$ času. Tedy **while**-cykly vyžadují čas $O\left(k\frac{|S|}{2^k}\right)$. Dostáváme, že operace **MAKEHEAP** vyžaduje čas

$$O\left(\sum_{k=1}^{\infty} k\frac{|S|}{2^k}\right) = O\left(|S|\sum_{k=1}^{\infty} \frac{k}{2^k}\right) = O(|S|).$$

Implementace operací **DECREASE** a **INCREASE** pomocí operací **UP** a **DOWN** jako v $d$-regulárních haldách není efektivní, protože délka cesty z kořene do listu v leftist haldě může být až $|S|$. Proto navrhneme složitější, ale efektivní algoritmus pro tyto operace.

Nejprve popíšeme pomocnou operaci **Oprav**$(T, v)$, která vytvoří leftist haldu z binárního stromu $T'$ vzniklého z leftist haldy $T$ odtrhnutím podstromu s kořenem ve vrcholu $v$.

**Oprav**$(T, v)$:
$t :=$ otec$(v)$, npl$(t) := 0$
**if** pravy$(t) \neq v$ **then** levy$(t) :=$ pravy$(t)$ **endif**
pravy$(t) := NIL$
**while** se zmenšilo npl$(t)$ a $t$ není kořen **do**
  $t :=$ otec$(t)$
  **if** npl(pravy$(t)$) $>$ npl(levy$(t)$) **then**
    vyměň levy$(t)$ a pravy$(t)$
  **endif**

$\text{npl}(t) := \text{npl}(\text{pravy}(t)) + 1$
**enddo**

Po provedení procedury **Oprav** mají všechny vrcholy správné číslo npl a navíc jsou splněny podmínky položené na leftist haldu. Tedy po provedení **Oprav** je $T$ opět leftist halda. Když $t$ je poslední vrchol, u kterého se zmenšilo npl, pak vrcholy, kde se zmenšilo npl tvoří pravou cestu z vrcholu $t$. To znamená, že **while**-cyklus se prováděl nejvýše $\log(|S|)$-krát a každý běh **while**-cyklu vyžadoval čas $O(1)$. Proto algoritmus **Oprav** vyžaduje čas $O(\log(|S|))$.

**Ostatní algoritmy.**

**DECREASE**$(s, a)$:
$v :=$ prvek reprezentující $s$
$T_1 :=$ podstrom $T$ určený vrcholem $v$, $f(v) := f(v) - a$
odtrhni podstrom $T_1$ od stromu $T$
$T_2 :=$**Oprav**$(T, v)$, $T :=$**MERGE**$(T_1, T_2)$

**INCREASE**$(s, a)$:
$v :=$ prvek reprezentující $s$
$T_1 :=$ podstrom $T$ určený vrcholem $\text{levy}(v)$
$T_2 :=$ podstrom $T$ určený vrcholem $\text{pravy}(v)$
$T_3 :=$ leftist halda reprezentující prvek $s$
odtrhni od stromu $T$ podstrom určený vrcholem $v$
$f(v) := f(v) + a$, $T_4 :=$**Oprav**$(T, v)$, $T_1 :=$**MERGE**$(T_1, T_3)$
$T_2 :=$**MERGE**$(T_2, T_4)$, $T :=$**MERGE**$(T_1, T_2)$

**DELETE**$(s, a)$:
$v :=$ prvek reprezentující $s$
$T_1 :=$ podstrom $T$ určený vrcholem $\text{levy}(v)$
$T_2 :=$ podstrom $T$ určený vrcholem $\text{pravy}(v)$
odtrhni od stromu $T$ podstrom určený vrcholem $v$
$T_3 :=$**MERGE**$(T_1, T_2)$, $T_4 :=$**Oprav**$(T, v)$
$T :=$**MERGE**$(T_3, T_4)$

Odtržení podstromu určeného vrcholem $v$ od stromu $T$ znamená provedení akce

**if** $v = \text{levy}(\text{otec}(v))$ **then**
$\text{levy}(\text{otec}(v)) := NIL$
**else**
$\text{pravy}(\text{otec}(v)) := NIL$
**endif**
$\text{otec}(v) := NIL$

**Shrnutí výsledků.**

**Věta.** *V leftist haldách existuje implementace operace* **MIN***, která v nejhorším případě vyžaduje čas $O(1)$, implementace operací* **INSERT***,* **DELETEMIN***,* **DELETE***,* **DECREASE***,* **INCREASE** *a* **MERGE***, které vyžadují v nejhorším případě čas $O(\log(|S|))$, a implementace operace* **MAKEHEAP***, která vyžaduje čas $O(|S|)$, kde $S$ je reprezentovaná množina.*

## Amortizovaná složitost

Předpokládejme, že $h$ je funkce, která ohodnucuje konfigurace. Když na konfiguraci $D$ aplikujeme operaci $o$ a dostaneme konfiguraci $D'$, pak amortizovaná složitost $am(o)$ operace $o$ je $am(o) = t(o) + h(D') - h(D)$, kde $t(o)$ je čas potřebný pro provedení operace $o$. Předpokládejme, že

$$D_0 \xrightarrow{o_1} D_1 \xrightarrow{o_2} D_2 \xrightarrow{o_3} \dots \xrightarrow{o_n} D_n.$$

Dále předpokládejme, že známe odhady amortizované složitosti operací: $am(o_i) \leq c(o_i)$ pro všechna $i = 1, 2, \dots, n$. Pak

$$\sum_{i=1}^{n} am(o_i) = \sum_{i=1}^{n} (t(o_i) + h(D_i) - h(D_{i-1})) =$$
$$h(D_n) - h(D_0) + \sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i).$$

Z toho plyne, že

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n) + h(D_0).$$

Obvykle $h(D) \geq 0$ pro všechny konfigurace $D$ nebo $h(D) \leq 0$ pro všechny konfigurace $D$. Když $h(D) \geq 0$ pro všechny konfigurace $D$, pak můžeme psát

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) + h(D_0),$$

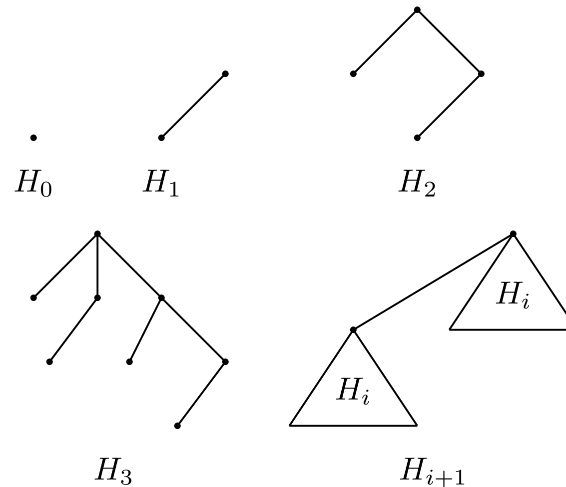
když $h(D) \leq 0$ pro všechny konfigurace $D$, pak můžeme psát

$$\sum_{i=1}^{n} t(o_i) \leq \sum_{i=1}^{n} c(o_i) - h(D_n).$$

To znamená, že odhad amortizované složitosti dává také odhad na časovou složitost, který je vhodný pro posloupnost operací. Odhad amortizované složitosti bývá lepší než odhad složitosti v nejhorším případě, a tedy i odhad složitosti posloupnosti operací v nejhorším případě nalezený pomocí amortizované složitosti je menší než součet složitostí jednotlivých operací v nejhorším případě. Zdůvodnění tohoto jevu je, že je nepravděpodobné, aby dvě operace za sebou vyžadovaly největší čas. To hraje zvlášť významnou roli, když nejhorších případů je málo (jinými slovy, to znamená, že jev, že nastane nejhorší případ, je málo pravděpodobný).

## Binomiální haldy

Definujme rekurentně binomiální stromy $H_i$ pro $i = 0, 1, \dots$. Jsou to kořenové stromy takové, že $H_0$ je jednoprvkový strom a strom $H_{i+1}$ vznikne ze dvou disjunktních stromů $H_i$, kde kořen jednoho se stane dalším synem kořene druhého z nich. Viz Obr. 1




Obr. 1

Nejprve uvedeme základní vlastnosti těchto stromů.

**Tvrzení.** *Pro každé přirozené číslo $i = 0, 1, \ldots$ platí:*

(1) *strom $H_i$ má $2^i$ vrcholů;*
(2) *kořen stromu $H_i$ má $i$ synů;*
(3) *délka nejdelší cesty z kořene do listu ve stromu $H_i$ je $i$;*
(4) *podstromy určené syny kořene stromu $H_i$ jsou izomorfní po řadě se stromy*

$$H_0, H_1, \ldots, H_{i-1}.$$

*Důkaz.* Tvrzení platí pro strom $H_0$ a jednoduchou indukcí dokážeme toto tvrzení i pro ostatní stromy. □

Binomiální halda $\mathcal{H}$ reprezentující množinu $S$ je soubor stromů $\{T_1, T_2, \ldots, T_k\}$ takový, že

počet vrcholů v těchto stromech je roven velikosti $S$ a je dáno jednoznačné přiřazení prvků z $S$ vrcholům stromů takové, že platí podmínka (usp);
každý strom $T_i$ je izomorfní s nějakým stromem $H_j$;
$T_i$ není izomorfní s žádným $T_j$ pro $i \neq j$.

Z binárního zápisu přirozených čísel plyne, že pro každé přirozené číslo $n > 0$ existuje prostá posloupnost $i_1, i_2, \ldots, i_k$ přirozených čísel taková, že $n = \sum_{j=1}^{k} 2^{i_j}$. Z toho plyne, že pro každou neprázdnou množinu $S$ existuje binomiální halda reprezentující $S$.

Operace pro binomiální haldy jsou založeny stejně jako pro leftist haldy na operaci **MERGE**. Operace **MERGE** pro binomiální haldy je analogií sčítání přirozených čísel v binárním zápise.

**MERGE**$(\mathcal{H}_1, \mathcal{H}_2)$:
(komentář: $\mathcal{H}_i$ reprezentuje množinu $S_i$ pro $i = 1, 2$ a $S_1 \cap S_2 = \emptyset$)
$i := 0$, $T :=$prázdný strom, $\mathcal{H} := \emptyset$
**while** $i < \log(|S_1| + |S_2|)$ **do**
  **if** existuje $U \in \mathcal{H}_1$ izomorfní s $H_i$ **then**
    $U_1 := U$

**else**
$U_1$ :=prázdný strom
**endif**
**if** existuje $U \in \mathcal{H}_2$ izomorfní s $H_i$ **then**
$U_2 := U$
**else**
$U_2$ :=prázdný strom
**endif**
**case**
(stromy $T$, $U_1$, $U_2$ jsou prázdné) **do**
nic
(existuje právě jeden neprázdný strom $V$ mezi stromy $T$, $U_1$ a $U_2$) **do**
vložme $V$ do $\mathcal{H}$, $T$ :=prázdný strom;
(existují právě dva neprázdné stromy $V_1$ a $V_2$ mezi stromy $T$, $U_1$ a $U_2$) **do**
$T$ :=**spoj**$(V_1, V_2)$
(všechny stromy $T$, $U_1$ a $U_2$ jsou neprázdné) **do**
vložme $T$ do $\mathcal{H}$, $T$ :=**spoj**$(U_1, U_2)$
**endcase**
$i := i + 1$
**enddo**
**if** $T \neq$prázdný strom **then** vložme $T$ do $\mathcal{H}$ **endif**
**Výstup**:$\mathcal{H}$

**spoj**$(T_1, T_2)$:
**if** $f\,(\text{kořen } T_1) > f\,(\text{kořen } T_2)$ **then**
vyměníme stromy $T_1$ a $T_2$
**endif**
kořen $T_2$ připojíme jako dalšího syna kořene $T_1$

Je vidět, že když oba stromy $T_1$ a $T_2$ jsou izomorfní s $H_i$, pak výsledný strom operace **spoj** je izomorfní se stromem $H_{i+1}$. Korektnost operace **MERGE** plyne z tohoto pozorování a z faktu, že $\mathcal{H}_j$ obsahuje strom izomorfní s $H_i$, právě když v binárním zápise čísla $|S_j|$ je na $i$-tém místě zprava 1, a že $T$ je neprázdný strom, když se provádí posun řádu při sčítání. Protože každý běh cyklu vyžaduje čas $O\,(1)$, algoritmus **MERGE** vyžaduje čas $O\,(\log\,(|S_1| + |S_2|))$. Implementace dalších algoritmů je podobná jako pro leftist haldy.

**INSERT**$(x)$:
Vytvoříme haldu $\mathcal{H}_1$ reprezentující $\{x\}$
**MERGE**$\mathcal{H}, \mathcal{H}_1)$

**MIN**:
Prohledáme prvky reprezentované kořeny stromů v $\mathcal{H}$ a nalezneme mezi nimi nejmenší prvek

**DELETEMIN**:
Prohledáme prvky reprezentované kořeny stromů v $\mathcal{H}$ a nalezneme mezi nimi strom $T$, jehož kořen reprezentuje nejmenší prvek
$\mathcal{H}_1 := \mathcal{H} \setminus \{T\}$, vytvoříme haldu $\mathcal{H}_2$ z podstromů $T$ určených syny kořene $T$
**MERGE**$(\mathcal{H}_1, \mathcal{H}_2)$

Z podmínky (usp) je zřejmé, že nejmenší prvek v $S$ je reprezentován v kořeni nějakého stromu haldy. Tím dostáváme korektnost operace **MIN**. Z tvrzení plyne, že $\mathcal{H}_2$ v operaci **DELETEMIN** je binomiální halda, a odtud plyne korektnost operace **DELETEMIN**. Operace **DECREASE** se implementuje pomocí operace **UP** a operace **INCREASE** pomocí operace **DOWN** stejně jako pro regulární haldy. Tato struktura nepodporuje přímo operaci **DELETE** (lze provést příkazy **DECREASE**($\infty$) a pak **DELETEMIN**)) a operace **MAKEHEAP** se provádí iterací operace **INSERT**.

Následující věta popisující efektivitu operací v této struktuře využívá faktu, že binomiální halda reprezentující množinu $S$ má tolik stromů, kolik je jedniček v binárním zápise $|S|$, což je nejvýše $\log(|S|)$, dále že operace **MERGE** simuluje sčítání čísel $|S_1|$ a $|S_2|$ v binárním zápise a má tedy odpovídající složitost, a konečně že amortizovaná složitost přičítání 1 k binárnímu číslu je $O(1)$.

**Věta.** *V binomiálních haldách algoritmy operací* **INSERT**, **MIN**, **DELETEMIN**, **DECREASE** *a* **MERGE** *vyžadují čas* $O(\log(|S|))$, *algoritmus operace* **INCREASE** *vyžaduje čas* $O\left(\log^2(|S|)\right)$ *a algoritmus operace* **MAKEHEAP** *vyžaduje čas* $O(|S|)$.

Z tvrzení plyne, že výška stromů v binomiální haldě je $\leq \log(|S|)$, ale počet synů je také $\leq \log(|S|)$ a tento odhad se nedá zlepšit. Odtud dostáváme složitost operací **DECREASE** a **INCREASE** v nejhorším případě.

Z těchto výsledků je vidět, že ostatní haldy mají efektivnější chování než binomiální haldy. Význam binomiálních hald spočívá v tom, že Fibonacciho haldy jsou jejich zobecněním. Na Fibonacciho haldách lze krásně ilustrovat princip, že pro řadu akcí je výhodné s nimi počkat a neprovádět je okamžitě. Na tomto principu pracuje i následující modifikace binomiálních hald.

### Líná implentace operací binomiální haldy.

Následující algoritmy jsou založeny na ideji, že 'vyvažování' stačí provádět jen při operacích **MIN** a **DELETEMIN**, kdy stejně musíme prohledat všechny stromy. Z tohoto důvodu zeslabíme podmínky na binomiální haldy.

Líná binomiální halda $\mathcal{H}$ reprezentující množinu $S$ je soubor stromů $\{T_1, T_2, \dots, T_k\}$ takový, že

> počet vrcholů v těchto stromech je roven velikosti $S$ a je dáno jednoznačné přiřazení prvků z $S$ vrcholům stromů takové, že platí podmínka (usp);
> každý strom $T_i$ je izomorfní s nějakým stromem $H_j$.

V líné binomiální haldě vynecháváme předpoklad na neizomorfnost stromů. Tento fakt se projeví ve velmi jednoduchém algoritmu pro operaci **MERGE**.

**MERGE**($\mathcal{H}_1, \mathcal{H}_2$):
Provedeme konkatenaci seznamů $\mathcal{H}_1$ a $\mathcal{H}_2$.

Algoritmus pro operaci **INSERT** se nezmění, jen provede tuto implementaci operace **MERGE**. Operace **MIN** a **DELETEMIN** použijí následující pomocnou proceduru **vyvaz**. Vstupem pro tuto operaci je soubor seznamů $\{O_i \mid i = 0, 1, \dots, k\}$, kde seznam $O_i$ obsahuje jen stromy izomorfní s $H_i$. Procedura **vyvaz** pak z těchto seznamů stromů vytvoří binomiální haldu.

**vyvaz**($\{O_i \mid i = 0, 1, \dots, k\}$):
$i := 0$, $\mathcal{H} := \emptyset$

**while** $i \leq k$ nebo $O_i \neq \emptyset$ **do**
  **while** $|O_i| > 1$ **do**
    vezmeme dva různé stromy $T_1$ a $T_2$ z $O_i$ a odstraníme je z $O_i$
    **spoj**$(T_1, T_2)$ vložíme do $O_{i+1}$
  **enddo**
  **if** $O_i \neq \emptyset$ **then**
    strom $T \in O_i$ odstraníme z $O_i$ a vložíme do $\mathcal{H}$
  **endif**
  $i := i + 1$
**enddo**
**Výstup**: $\mathcal{H}$

**MIN**:
Prohledáme všechny stromy v $\mathcal{H}$, nalezneme nejmenší prvek reprezentovaný v kořeni nějakého stromu a stromy rozdělíme do množin $O_i$ obsahujících všechny stromy v izomorfní s $H_i$.
**vyvaz**$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

**DELETEMIN**:
Prohledáme všechny stromy v $\mathcal{H}$, nalezneme nejmenší prvek reprezentovaný v kořeni nějakého stromu $T \in \mathcal{H}$, stromy rozdělíme do množin $O_i$ obsahujících všechny stromy izomorfní s $H_i$ různé od $T$ a dále pro každého syna kořene stromu $T$ dáme podstrom určený tímto synem do $O_i$ jestliže tento podstrom $T$ je izomorfní s $H_i$.
**vyvaz**$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

Amortizovaná složitost operací **INSERT** a **MERGE** při líné implementaci je $O(1)$ a amortizovaná složitost operací **MIN** a **DELETEMIN** je $O(\log(|S|))$. Ohodnocením líné binomiální haldy bude počet stromů v této haldě (přesněji dvojnásobek počtu stromů). Amortizovaná složitost je čas operace plus ohodnocení výsledné struktury minus ohodnocení počáteční struktury. Líná implementace operací **MERGE** a **INSERT** vyžaduje čas $O(1)$ a operace **MERGE** nemění počet stromů, kdežto operace **INSERT** přidá jeden strom. Odtud amortizovaná složitost obou operací je omezena konstantou, a tedy je $O(1)$. Protože každý běh vnitřního **while**-cyklu v operaci **vyvaz** vyžaduje čas $O(1)$ a zmenší počet stromů v seznamech $O_i$ o 1, dostaneme, že operace **vyvaz** vyžaduje čas $O\left(k + \sum_{i=0}^{k} |O_i|\right) = O(k + |\mathcal{H}|)$, kde $k$ je počet seznamů. Z Tvrzení plyne, že $k, |\langle| \leq \log(|S|)$. Operace **MIN** bez podprocedury **vyvaz** vyžaduje čas $O(|\mathcal{H}|)$ a operace **DELETEMIN** bez podprocedury **vyvaz** vyžaduje čas $O(\mathcal{H} + i)$, kde $T$ je izomorfní s $H_i$. Podle tvrzení je $i \leq \log(|S|)$, a tedy operace **MIN** vyžaduje čas $O(2|\mathcal{H}| + \log(|S|))$ a operace **DELETEMIN** vyžaduje čas $O(2|\mathcal{H}| + 2\log(|S|))$. Protože ohodnocení binomiální haldy je nejvýše $2\log(|S|)$, dostaneme, že odhad amortizované složitosti operací **MIN** a **DELETEMIN** je

$$O(2|\mathcal{H}| - 2|\mathcal{H}| + 4\log(|S|)) = O(\log(|S|)).$$

Protože si funkci ohodnocení volíme, můžeme použít takové multiplikativní koeficienty, aby jednotka času odpovídala jednotce v amortizované složitosti. Proto lze $|\mathcal{H}|$ od sebe odečíst.

## Fibonacciho haldy

Zhruba řečeno, Fibonacciho halda je množina stromů, kde některé vrcholy různé od kořenů stromů jsou označené a existuje jednoznačná korepondence mezi prvky $S$ a vrcholy stromů taková, že splňuje podmínku (usp). Bohužel toto je jen přibližné vyjádření. Existují takovéto struktury, které nevznikly z prázdné Fibonacciho haldy pomocí posloupnosti operací implementovaných navrženými algoritmy. Přitom důkaz efektivity Fibonacciho hald se dost výrazně opírá o fakt, že halda takto vznikla. Proto nejdříve popíšeme algoritmy pro operace a pak řekneme, že Fibonacciho halda je struktura vzniklá z prázdné Fibonacciho haldy pomocí posloupnosti operací, které byly realizovány navrženými algoritmy.

Budeme předpokládat, že Fibonacciho halda je seznam stromů, kde některé vrcholy různé od kořene jsou označeny. Vrchol je označen, právě když není kořen a byl mu někdy dřív v tomto stromě odtržen nějaký jeho syn. Řekneme, že strom má rank $i$, když kořen má $i$ synů.

Algoritmy pro operace **MERGE**, **INSERT**, **MIN** a **DELETEMIN** jsou analogické jako pro línou implementaci v binomiálních haldách. Jen požadavek, aby strom byl izomorfní s $H_i$, je nahrazen požadavkem, že má rank $i$. Algoritmy pro operace **DECREASE**, **INCREASE** a **DELETE** jsou založeny na algoritmech pro tyto operace v leftist haldách. V algoritmech předpokládáme, že $a = \log^{-1}\left(\frac{3}{2}\right)$.

**Algoritmy.**

**MERGE**$(\mathcal{H}_1, \mathcal{H}_2)$:
Provedeme konkatenaci seznamů $\mathcal{H}_1$ a $\mathcal{H}_2$.

**INSERT**$(x)$:
Vytvoříme haldu $\mathcal{H}_1$ reprezentující $\{x\}$
**MERGE**$(\mathcal{H}, \mathcal{H}_1)$

**MIN**:
Prohledáme všechny stromy v $\mathcal{H}$, nalezneme nejmenší prvek reprezentovaný v kořeni nějakého stromu a stromy rozdělíme do množin $O_i$, kde $O_i$ je množina všech stromů v haldě s rankem $i$.
**vyvaz1**$\big(\{O_i \mid i = 0, 1, \dots, \lfloor a \log\left(\sqrt{5}|S| + 1\right)\rfloor\}\big)$

**DELETEMIN**:
Prohledáme všechny stromy v $\mathcal{H}$, nalezneme nejmenší prvek reprezentovaný v kořeni nějakého stromu $T \in \mathcal{H}$, stromy rozdělíme do množin $O_i$, kde $O_i$ obsahuje všechny stromy s rankem $i$ různé od $T$ a dále každý podstrom $T$ určený synem kořene $T_i$ dáme do $O_i$, právě když má rank $i$.
**vyvaz1**$\big(\{O_i \mid i = 0, 1, \dots, \lfloor a \log\left(\sqrt{5}|S| + 1\right)\rfloor\}\big)$

**vyvaz1**$(\{O_i \mid i = 0, 1, \dots, k\})$:
$i := 0$, $\mathcal{H} := \emptyset$
**while** $i \le k$ nebo $O_i \ne \emptyset$ **do**
  **while** $|O_i| > 1$ **do**
    vezmeme dva různé stromy $T_1$ a $T_2$ z $O_i$ a odstraníme je z $O_i$
    **spoj**$(T_1, T_2)$ vložíme do $O_{i+1}$
  **enddo**
  **if** $O_i \ne \emptyset$ **then**

strom $T \in O_i$ odstraníme z $O_i$ a vložíme do $\mathcal{H}$
**endif**
$i := i + 1$
**enddo**
**Výstup**: $\mathcal{H}$

**spoj**$(T_1, T_2)$:
**if** $f$ (kořen $T_1$) $>$ $f$ (kořen $T_2$) **then**
vyměníme stromy $T_1$ a $T_2$
**endif**
kořen $T_2$ připojíme jako dalšího syna kořene $T_1$

**DECREASE**$(s, z)$:
$T$ :=strom v $\mathcal{H}$ reprezentující $s$
$v$ :=vrchol ve stromu $T$ reprezentující $s$
**if** $v$ je kořen **then**
$f(v) := f(v) - z$
**else**
odtrhneme podstrom $T'$ určený vrcholem $v$
**vyvaz2**$(T, v)$
pokud $v$ byl označen, zrušíme označení vrcholu $v$
$f(v) := f(v) - z$, $T'$ vložíme do $\mathcal{H}$
**endif**

**INCREASE**$(s, z)$:
$T$ :=strom v $\mathcal{H}$ reprezentující $s$
$v$ :=vrchol ve stromu $T$ reprezentující $s$
**if** $v$ není list **then**
odtrhneme podstrom $T'$ určený vrcholem $v$
**if** $v$ není kořen **then** **vyvaz2**$(T, v)$ **endif**
pokud $v$ byl označen, zrušíme označení vrcholu $v$
$f(v) := f(v) + z$,
**for every** $u$ syn vrcholu $v$ **do**
zrušíme označení vrcholu $u$
do $\mathcal{H}$ vložíme podstrom $T'$ určený vrcholem $u$
**enddo**
do $\mathcal{H}$ vložíme strom mající jen vrchol $v$
**else**
$f(v) := f(v) + z$
**endif**

**DELETE**$(s)$:
$T$ :=strom v $\mathcal{H}$ reprezentující $s$
$v$ :=vrchol ve stromu $T$ reprezentující $s$
odtrhneme podstrom $T'$ určený vrcholem $v$
**if** $v$ není kořen **then** **vyvaz2**$(T, v)$ **endif**
zrušíme označení u všech synů vrcholu $v$ a
do $\mathcal{H}$ vložíme všechny podstromy $T'$ určené syny $v$

**vyvaz2**$(T, v)$:
$u :=$ otec $v$
**while** $u$ je označen **do**
  $u' :=$ otec $(u)$, zrušíme označení $u$
  odtrhneme podstrom $T$ určený vrcholem $u$ a vložíme ho do $\mathcal{H}$, $u := u'$
**enddo**
**if** $u$ není kořen $T$ **then** označíme $u$ **endif**

Všimněme si, že když stromy $T_1$ a $T_2$ mají rank $i$, pak procedura **spoj**$(T_1, T_2)$ vytvoří strom s rankem $i + 1$. Aby algoritmy pro operace **MIN** a **DELETEMIN** byly korektní, musíme ukázat, že všechny stromy ve Fibonacciho haldě $\mathcal{H}$ reprezentující množinu $S$ mají rank nejvýše $a \log\left(\sqrt{5}|S| + 1\right)$. Jen tak zajistíme, aby výsledná halda reprezentovala $S$, respektive $S \setminus \{$prvek s nejmenší hodnotou $f\}$. Operace **vyvaz2** zajišťuje, že od každého vrcholu stromu různého od kořene byl v tomto stromě odtržen podstrom nejvýše jednoho syna – v tom případě je tento prvek označen a když se mu odtrhává podstrom dalšího syna, bude odtržen i celý podstrom tohoto vrcholu (tím se stane kořenem stromu). Když se později stane tento vrchol zase vrcholem různým od kořene, celý proces se opakuje.

**Složitost operací.**

Nejdříve spočítáme časovou složitost jednotlivých operací:

**MERGE** časová složitost $O(1)$, nevzniká žádný nový strom, označené vrcholy se nemění;
**INSERT** časová složitost $O(1)$, přibyl jeden strom, označené vrcholy se nemění;
**MIN** časová složitost $O(|\mathcal{H}|)$, po provedení operace různé stromy v haldě mají různé ranky, žádný nový vrchol nebyl označen;
**DELETEMIN** časová složitost $O(|\mathcal{H}| +$ počet synů $v)$, kde $v$ reprezentoval prvek s nejmenší hodnotou $f$, po provedení operace různé stromy v haldě mají různé ranky, žádný nový vrchol nebyl označen;
**DECREASE** časová složitost $O(1 + c)$, kde $c$ je počet vrcholů, které přestaly být označené, bylo přidáno $1 + c$ nových stromů a byl označen nejvýše jeden nový vrchol;
**INCREASE** časová složitost $O(1 + c + d)$, kde $c$ je počet vrcholů, které přestaly být označené, $d$ je počet synů vrcholu $v$, bylo přidáno $1 + c + d$ nových stromů a byl označen nejvýše jeden nový vrchol;
**DELETE** časová složitost $O(1 + c + d)$, kde $c$ je počet vrcholů, které přestaly být označené, $d$ je počet synů vrcholu $v$, bylo přidáno $c + d$ nových stromů a byl označen nejvýše jeden nový vrchol.

Abychom spočítali amortizovanou složitost, musíme nejdříve navrhnout funkci ohodnocující konfigurace. Nechť ohodnocení konfigurace je počet stromů v konfiguraci plus dvojnásobek počtu označených vrcholů. Nechť $\rho(n)$ je maximální počet synů vrcholu ve Fibonacciho haldě reprezentující $n$-prvkovou množinu. Pak amortizovaná složitost operací je:
**MERGE** amortizovaná složitost je $O(1)$;
**INSERT** amortizovaná složitost je $O(1)$;
**MIN** amortizovaná složitost je $O(\rho(n))$;
**DELETEMIN** amortizovaná složitost je $O(\rho(n))$;
**DECREASE** amortizovaná složitost je $O(1)$;
**INCREASE** amortizovaná složitost je $O(\rho(n))$;
**DELETE** amortizovaná složitost je $O(\rho(n))$.

Abychom spočítali odhad $\rho(n)$, využijeme toho, že Fibonacciho halda vznikla z prázdné

haldy pomocí popsaných algoritmů. Nejprve jedno technické lemma.

**Lemma.** *Nechť $v$ je vrchol stromu ve Fibonacciho haldě a nechť $u$ je $i$-tý nejstarší syn vrcholu $v$, pak $u$ má aspoň $i-2$ synů.*

*Důkaz.* Když se $u$ stával synem $v$, aplikovala se operace **spoj** na stromy s kořeny $u$ a $v$. Přitom vrcholy $u$ a $v$ měly stejný počet synů. Podle předpokladů měl vrchol $v$ alespoň $i-1$ synů (jinak by $u$ nebyl $i$-tý nejstarší syn), a protože od $u$ se mohl odtrhnout jen jeden syn, dostáváme, že $u$ musí mít alespoň $i-2$ synů. □

**Tvrzení.** *Nechť $v$ je vrchol stromu ve Fibonacciho haldě, který má právě $i$ synů, pak podstrom určený vrcholem $v$ má aspoň $F_{i+2}$ vrcholů.*

*Důkaz.* Tvrzení dokážeme pomocí indukce podle maximální délky cesty z vrcholu $v$ do některého listu. Tato délka je 0, právě když $v$ je list. V tom případě $v$ nemá syna a podstrom určený vrcholem $v$ má jediný vrchol. Protože $1 = F_2 = F_{0+2}$, tak tvrzení platí. Mějme vrchol $v$, který má $k$ synů, a nechť maximální délka cesty z vrcholu $v$ do listů je $j$. Předpokládejme, že tvrzení platí pro všechny vrcholy, pro něž maximální délka cesty z nich do listů je menší než $j$. Tedy tvrzení platí pro všechny syny vrcholu $v$. Pak pro $i > 1$ má $i$-tý nejstarší syn vrcholu $v$ podle předchozího lemmatu alespoň $i-2$ synů a podle indukční hypotézy podstrom určený tímto synem má alespoň $F_i$ vrcholů. Odtud dostáváme, že podstrom určený vrcholem $v$ má alespoň

$$1 + F_2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=1}^{k} F_i$$

vrcholů, protože $F_1 = F_2$ (první 1 je za vrchol $v$, první $F_2$ je za nejstarší vrchol). Indukcí dokážeme, že

$$1 + \sum_{i=1}^{n} F_i = F_{n+2}$$

pro všechna $n \geq 0$. Skutečně, pro $n = 0$ platí

$$1 + \sum_{i=1}^{0} F_i = 1 = F_2 = F_{0+2}$$

a pro $n = 1$ máme

$$1 + \sum_{i=1}^{1} F_i = 1 + F_1 = 2 = F_3 = F_{1+2}.$$

Dále indukcí dostáváme, že

$$1 + \sum_{i=1}^{n} F_i = 1 + \sum_{i=1}^{n-1} F_i + F_n = F_{n+1} + F_n = F_{n+2}.$$

Když shrneme tato fakta, dostáváme, že podstrom určený vrcholem $v$ má alespoň $F_{k+2}$ vrcholů, a tvrzení je dokázáno. □

Vezměme nejmenší $i$ takové, že $n < F_i$. Protože posloupnost $\{F_i\}_{i=1}^{\infty}$ je rostoucí, plyne z předchozího tvrzení, že každý vrchol ve Fibonacciho haldě reprezentující $n$ prvkovou

množinu má méně než $i-2$ synů (když vrchol $v$ Fibonacciho haldy má $i-2$ synů, pak podstrom vrcholu $v$ reprezentuje množinu alespoň s $F_i$ prvky). Proto $\rho(n) < i-2$. K odhadu velikosti $i$ použijeme explicitní vzorec pro $i$-té Fibonacciho číslo:

$$F_i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i.$$

Protože $0 > \frac{1-\sqrt{5}}{2} > -\frac{3}{4}$ a protože $\sqrt{5} > 2$, je $|\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i| < \frac{3}{8}$ pro všechna $i = 1, 2, \ldots$, a tedy

$$\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{3}{8} < F_i < \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i + \frac{3}{8}.$$

Odtud dostáváme, že když $i$ splňuje

$$n \le \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{3}{8},$$

pak $n < F_i$. Převedením $\frac{3}{8}$ na druhou stranu výrazu, jeho vynásobením $\sqrt{5}$ a zlogaritmováním dostaneme následující ekvivalenci:

$$\log_2\left(\sqrt{5}n + \frac{3\sqrt{5}}{8}\right) \le i \log_2\left(\frac{1+\sqrt{5}}{2}\right) \quad \Leftrightarrow$$
$$n \le \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^i - \frac{3}{8}.$$

Z $\frac{3\sqrt{5}}{8} < 1$ a z $\frac{3}{2} < \frac{1+\sqrt{5}}{2}$ plyne, že

$$\frac{\log_2\left(\sqrt{5}n + \frac{3\sqrt{5}}{8}\right)}{\log_2\frac{1+\sqrt{5}}{2}} < \frac{\log_2\left(\sqrt{5}n + 1\right)}{\log_2\frac{3}{2}}.$$

Tedy platí následující implikace:

$$\frac{\log_2\left(\sqrt{5}n + 1\right)}{\log_2\frac{3}{2}} < i \quad \Longrightarrow \quad \frac{\log_2\left(\sqrt{5}n + \frac{3\sqrt{5}}{8}\right)}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} < i.$$

Proto když $\frac{\log_2\left(\sqrt{5}n+1\right)}{(\log_2 3)-1} < i$, pak $n < F_i$, a tedy $\rho(n) < i-2$.

Výsledky shrneme do následující věty:

**Věta.** *Ve Fibonacciho haldě, která reprezentuje $n$ prvkovou množinu, má každý vrchol stupeň menší než*

$$\frac{\log_2\left(\sqrt{5}n + 1\right)}{(\log_2 3) - 1} - 2 = O(\log n).$$

*Amortizovaná složitost operací* **INSERT**, **MERGE** *a* **DECREASE** *je* $O(1)$ *a amortizovaná složitost operací* **MIN**, **DELETEMIN**, **INCREASE** *a* **DELETE** *je* $O(\log n)$. *Operace* **MIN** *a* **DELETEMIN** *jsou korektní.*

Pro úplnost dokážeme, že $F_i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}}$.
Pro $i = 1$ platí

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^1 - \left(\frac{1-\sqrt{5}}{2}\right)^1}{\sqrt{5}} = \frac{1+\sqrt{5}-1+\sqrt{5}}{2\sqrt{5}} = \frac{2\sqrt{5}}{2\sqrt{5}} = 1 = F_1.$$

Pro $i = 2$ platí

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} = \frac{1+2\sqrt{5}+5-1+2\sqrt{5}-5}{4\sqrt{5}} = \frac{4\sqrt{5}}{4\sqrt{5}} = 1 = F_2.$$

Indukční krok:

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(\frac{1+\sqrt{5}}{2}\right)^2 - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(\frac{1-\sqrt{5}}{2}\right)^2}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(\frac{3+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(\frac{3-\sqrt{5}}{2}\right)}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2}\left(1+\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}\left(1+\frac{1-\sqrt{5}}{2}\right)}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2} + \left(\frac{1+\sqrt{5}}{2}\right)^{i-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-1}}{\sqrt{5}} =$$

$$\frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-2} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-2}}{\sqrt{5}} + \frac{\left(\frac{1+\sqrt{5}}{2}\right)^{i-1} - \left(\frac{1-\sqrt{5}}{2}\right)^{i-1}}{\sqrt{5}} =$$

$$F_{i-2} + F_{i-1} = F_i.$$

Tedy indukcí dostáváme požadovaný vztah.

**Hledání nejkratších cest.**

Vrátíme se k Dijkstrově algoritmu. Množinu $U$ budeme reprezentovat pomocí Fibonacciho haldy. Protože ohodnocení hald je nezáporné a ohodnocení počáteční haldy je 0, dává odhad amortizované složitosti také odhad časové složitosti. Proto časová složitost Dijkstrova algoritmu v nejhorším případě je $O(|X|(1+\log|X|)+|R|) = O(|R|+|X|\log|X|)$. Stejný výsledek dostaneme i pro konstrukci nejmenší napnuté kostry grafu.

Otázka je, kdy použít Fibonacciho haldu a kdy použít $d$-regulární haldy v Dijkstrově algoritmu nebo v algoritmu konstruujícím nejmenší napnutou kostru. Lze říci, že Fibonacciho halda by měla být výrazně lepší pro větší, ale řídké grafy (tj. grafy s malým počtem hran). Dá se předpokládat, že $d$-regulární haldy budou lepší (díky svým jednodušším algoritmům) pro husté grafy (tj. grafy, kde počet hran je $|X|^{1+\varepsilon}$ pro vhodné $\varepsilon > 0$). Problém je, pro které hodnoty nastává zlom. Nevím o žádných experimentálních nebo teoretických výsledcích tohoto typu.

Historický přehled: Binární neboli 2-regulární haldy zavedl Williams 1964. Jejich zobecnění na $d$-regulární haldy pochází od Johnsona 1975. Leftist haldy definoval Crane 1972 a detailně popsal Knuth 1975. Binomiální haldy navrhl Vuillemin 1978, Brown 1978 je implementoval a prokázal jejich praktickou použitelnost. Fibonacciho haldy byly zavedeny Fredmanem a Tarjanem 1987.

## Třídicí algoritmy

Jeden ze základních problémů datových struktur je následující:
$U$ je totálně uspořádané univerzum.
Vstup: Prostá posloupnost $\{a_1, a_2, \dots, a_n\}$ prvků z univerza $U$.
Výstup: Rostoucí posloupnost $\{b_1, b_2, \dots, b_n\}$ taková, že $\{a_i \mid i = 1, 2, \dots, n\} = \{b_i \mid i = 1, 2, \dots, n\}$. Tento problém se nazývá třídění. V mnoha aplikacích datových struktur je nutné ho řešit.

Jsou tři základní algoritmy, které řeší třídicí problém: **HEAPSORT**, **MERGESORT**, **QUICKSORT**. **HEAPSORT** byl první algoritmus používající haldy (binární regulární haldy byly definovány při návrhu **HEAPSORT**u). Byl popsán jako jedna z aplikací regulárních hald. Je mu stále věnována velká pozornost a bylo navrženo několik jeho modifikací. Řekneme si více o implementaci třídění na místě.

Třídicí algoritmy se často používají jako podprocedura při řešení jiných úloh. V takovém případě je obvykle vstupní posloupnost uložena v poli v pracovní paměti programu a požadavkem je setřídit ji bez použití další paměti pouze s výjimkou omezeného (malého) počtu pomocných proměnných. Pro řešení tohoto problému se hodí **HEAPSORT** implementovaný pomocí $d$-regulárních hald, které jsou reprezentovány polem, v němž je uložena vstupní posloupnost. Použijeme algoritmus s jedinou změnou – budeme požadovat duální podmínku na uspořádání a nahradíme operace **MIN** a **DELETEMIN** operacemi **MAX** a **DELETEMAX**. V algoritmu vždy umístíme odebrané maximum na místo prvku v posledním listu haldy (tj. prvku, který ho při operaci **DELETEMAX** nahradil) místo toho, abychom ho vložili do výstupní posloupnosti.

Nejstarší z uvedených algoritmů je **MERGESORT** a je starší než je počítačová éra. Jeho verze se používaly už při mechanickém třídění. Popíšeme jednu jeho iterační verzi.

**MERGESORT**$(a_1, a_2, \dots, a_n)$:
$Q :=$ prázdná fronta, $i := 1$
**while** $i \le n$ **do**
  $j := i$
  **while** $i < n$ a $a_{i+1} > a_i$ **do** $i := i + 1$ **enddo**
  posloupnost $P = (a_j, a_{j+1}, \dots, a_i)$ vložíme do $Q$
  $i := i + 1$
**enddo**
**while** $|Q| > 1$ **do**

vezmeme $P_1$ a $P_2$ dvě posloupnosti z vrcholu $Q$
odstraníme $P_1$ a $P_2$ z $Q$
**MERGE**$(P_1, P_2)$ vložíme na konec $Q$
**enddo**
**Výstup**: posloupnost z $Q$

**MERGE**$(P_1 = (a_1, a_2, \ldots, a_n), P_2 = (b_1, b_2, \ldots, b_m))$:
$P :=$ prázdná posloupnost, $i := 1$, $j := 1$, $k := 1$
**while** $i \leq n$ a $j \leq m$ **do**
**if** $a_i < b_j$ **then**
$c_k := a_i$, $i := i + 1$, $k := k + 1$
**else**
$c_k := b_j$, $j := j + 1$, $k := k + 1$
**endif**
**enddo**
**while** $i \leq n$ **do**
$c_k := a_i$, $i := i + 1$, $k := k + 1$
**enddo**
**while** $j \leq m$ **do**
$c_k := b_j$, $j := j + 1$, $k := k + 1$
**enddo**
**Výstup:** $P = (c_1, c_2, \ldots, c_{n+m})$

Všimněme si, že všechny posloupnosti v $Q$ jsou rostoucí a že množina $\{a_i \mid i = 1, 2, \ldots, n\}$ je sjednocením všech prvků z posloupností v $Q$ vždy na začátku běhu cyklu **while** $|Q| > 1$. Každý průběh tohoto cyklu zmenší počet posloupností v $Q$ o 1. Protože počet posloupností ve frontě $Q$ je nejvýše délka vstupní posloupnosti, je algoritmus **MERGESORT** korektní.

Složitost podprocedury **MERGE**. Určení prvku $c_k$ vyžaduje čas $O(1)$ (nejvýše jedno porovnání) a maximální hodnota $k$ je $n + m$. Tedy podprocedura **MERGE** vyžaduje čas $O(n + m)$ (a provede nejvýše $n + m$ porovnání), kde $n$ a $m$ jsou délky vstupních posloupností.

Složitost procedury **MERGESORT**. První cyklus vyžaduje čas $O(n)$, kde $n$ je délka vstupní posloupnosti. Před první běh cyklu **while** položíme na vrchol $Q$ speciální znak $\natural$, který se vždy jen přenese z vrcholu $Q$ na její konec. Protože mezi dvěma přenosy znaku $\natural$ projde každý prvek podprocedurou **MERGE** právě jednou, vyžadují běhy cyklu **while** mezi dvěma přenosy $\natural$ čas $O(n)$. Všechny posloupnosti na počátku mají délku $\geq 1$, a proto po $i$-tém přenosu $\natural$ mají délku $\geq 2^{i-1}$ a počet přenosů $\natural$ je nejvýše $\lceil \log_2 n \rceil$. Tedy algoritmus **MERGESORT** vyžaduje čas $O(n \log n)$.

Nyní popíšeme algoritmus **QUICKSORT**. Je to nejvíce používaný algoritmus, protože pro obecně danou posloupnost má nejlepší očekávaný čas.

**Quick**$(a_i, a_{i+1}, \ldots, a_j)$:
**if** $i = j$ **then**
**Výstup**: $(a_i)$
**else**
zvolíme $k$ takové, že $i \leq k \leq j$, $a := a_k$,
vyměníme $a_i$ a $a_k$, $l := i + 1$, $q := j$
**while** true **do**

**while** $a_l < a$ **do** $l := l + 1$ **enddo**
**while** $a_q > a$ **do** $q := q - 1$ **enddo**
**if** $l > q$ **then**
ukončit běh cyklu
**else**
vyměníme $a_l$ a $a_q$, $l := l + 1$, $q := q - 1$
**endif**
Komentář: Situace, kdy $l = q$, nenastává.
**enddo**
**if** $i + 1 = l$ **then**
**Výstup**($a$, Quick $(a_{q+1}, a_{q+2}, \ldots, a_j)$)
**else**
**if** $j = q$ **then**
**Výstup**(Quick $(a_{i+1}, \ldots, a_{l-1})$, $a$)
**else**
**Výstup**(Quick $(a_{i+1}, \ldots, a_{l-1})$, $a$, Quick $(a_{q+1}, \ldots, a_j)$)
**endif**
**endif**
**endif**

**QUICKSORT**($a_1, a_2, \ldots, a_n$):
**Výstup**(Quick $(a_1, a_2, \ldots, a_n)$)

Algoritmus **Quick** třídí posloupnost $(a_{i+1}, a_{i+2}, \ldots, a_j)$ tak, že posloupnost $(a_i, a_{i+1}, a_{l-1})$ obsahuje všechny prvky vstupní posloupnosti $< a = a_k$ a posloupnost $(a_{q+1}, a_{q+2}, \ldots, a_j)$ obsahuje všechny prvky vstupní posloupnosti $> a = a_k$. Na tyto posloupnosti pak zavolá sám sebe a do výsledné posloupnosti uloží setříděnou první posloupnost, pak prvek $a$ a nakonec setříděnou druhou posloupnost. Korektnost procedury **Quick** i algoritmu **QUICKSORT** je tedy zřejmá, protože $l \leq j$ a $i \leq q$.

Procedura **Quick** bez rekurzivního volání vyžaduje čas $O(j - i)$. Tedy kdyby $a_k$ byl medián posloupnosti $(a_i, a_{i+1}, \ldots, a_j)$ (tj. prostřední prvek), pak by algoritmus **QUICKSORT** vyžadoval čas $O(n \log n)$. Jak uvidíme později, medián lze nalézt v lineárním čase, ale použít jakoukoliv známou proceduru pro jeho nalezení má za důsledek, že **MERGESORT** a **HEAPSORT** budou rychlejší (nikoliv asymptoticky). Proto je třeba volit prvek $a_k$ (tento prvek se nazývá pivot) co nejrychleji. Původně se bral první nebo poslední prvek. Při rovnoměrném rozdělení vstupu je pak očekávaný čas algoritmu $O(n \log n)$ a algoritmus je obvykle rychlejší než algoritmy **MERGESORT** a **HEAPSORT**. Nevýhoda je, že pro určité rozdělení dat se takový algoritmus chová špatně (to znamená, že vyžaduje kvadratický čas). Proto tuto verzi algoritmu není vhodné použít pro úlohy, kdy rozdělení dat bude pro takovou volbu nevýhodné. Lze to napravit tak, že budeme volit $k$ náhodně. Bohužel, použití pseudonáhodného generátoru vyžaduje čas, a pak už algoritmus zase nemusí být rychlejší než algoritmy **MERGESORT** a **HEAPSORT** (a navíc náhodně zvolený prvek není skutečně náhodný, ale to v tomto případě nevadí). Důsledkem je návrh brát pivota jako medián tří nebo pěti pevně zvolených prvků posloupnosti. Praxe ukázala, že tento výběr pivota je nejpraktičtější, dá se provést rychle a zajišťuje dostatečnou náhodnost.

Protože při každém volání má **Quick** kratší vstupní posloupnost, lze ukázat, že při každé volbě pivota je nejhorší čas algoritmu **QUICKSORT** $O(n^2)$, a pokud je pivot vybrán jednoduchým a rychlým způsobem, pak existuje konfigurace, která vyžaduje čas $O(n^2)$

(když se volí náhodně, tak pro každou konfiguraci existuje taková volba). Nyní ukážeme, že očekávaný čas je $O(n \log n)$. Následná analýza je pro náhodně zvoleného pivota (bez dalšího předpokladu na vstupní data) nebo pro případ, kdy pivot je pevně zvolen a data jsou rovnoměrně rozdělena.

Ukážeme dva výpočty očekávaného času. Jeden je založen na několika jednoduchých pozorováních a druhý na rekurzivním počítání. Hlavní idea v obou výpočtech je založena na pozorování:

Očekávaný čas algoritmu **QUICKSORT** je
$O$ (očekávaný počet porovnání v algoritmu **QUICKSORT**) .

Tento fakt plyne přímo z popisu algoritmu. Spočítáme očekávaný počet porovnání pro algoritmus **QUICKSORT**.

**První výpočet.**

Prvky $a_i$ a $a_j$ algoritmus **QUICKSORT** porovná při třídění posloupnosti $(a_1, a_2, \ldots, a_n)$ nejvýše jednou. Když algoritmus **QUICKSORT** porovnává prvky $a_i$ a $a_j$, pak pro nějaký běh podprocedury **Quick** je $a_i$ nebo $a_j$ pivot. Přitom v předchozích bězích **Quick** $a_i$ ani $a_j$ nebylo pivotem, protože pivot se vždy vyřadí z následujících volání této podprocedury.

Nechť $(b_1, b_2, \ldots, b_n)$ je výsledná posloupnost. Označme $X_{i,j}$ boolskou proměnou, která má hodnotu 1, když **QUICKSORT** provedl porovnání mezi prvky $b_i$ a $b_j$, jinak má hodnotu 0. Předpokládejme, že je to náhodná veličina. Když $p_{i,j}$ je pravděpodobnost, že $X_{i,j} = 1$, pak očekávaná hodnota $X_{i,j}$ je

$$\mathbf{E}(X_{i,j}) = 0(1 - p_{i,j}) + 1p_{i,j} = p_{i,j}.$$

Protože počet porovnání při běhu algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j}$$

a protože očekávaná hodnota součtu náhodných proměnných je součet očekávaných hodnot, dostáváme, že očekávaný počet porovnání v algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}(X_{i,j}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}.$$

Abychom spočítali $p_{i,j}$, popíšeme chování algoritmu **QUICKSORT** pomocí modifikace stromu výpočtu. Bude to binární strom, v němž každý vrchol odpovídá jednomu běhu podprocedury **Quick** a vrchol $v$ bude vnitřní vrchol, když odpovídající podprocedura volila pivota, který ohodnotí vrchol $v$. V podstromu levého syna vrcholu $v$ budou právě všechna následující rekurzivní volání podprocedury **Quick** nad posloupností, která předchází pivotu. Analogicky v podstromu pravého syna vrcholu $v$ budou právě všechna následující rekurzivní volání procedury **Quick** nad posloupností, která následuje po pivotu. Listy stromu jsou označeny prvky, které jsou v posloupnosti, s níž je voláno odpovídající **Quick**. Když vrchol $v$ odpovídá volání **Quick** s posloupností $(a_i, a_{i+1}, \ldots, a_j)$, pak vrcholy v podstromu levého syna $v$ jsou ohodnoceny prvky z posloupnosti $(a_{i+1}, a_{i+2}, \ldots, a_{l-1})$ a vrcholy v podstromu pravého syna vrcholu $v$ jsou ohodnoceny prvky z posloupnosti $(a_{q+1}, \ldots, a_j)$ (po přerovnání posloupnosti). Dále platí $\{a_l \mid i \leq l \leq j\} = \{b_l \mid i \leq l \leq j\}$.

Očíslujeme vrcholy tohoto stromu prohledáváním do šířky, za předpokladu, že levý syn vrcholu předchází pravému synu. Nechť $(c_1, c_2, \dots, c_n)$ je posloupnost prvků $\{a_i \mid 1 \le i \le n\}$ v pořadí daném tímto očíslováním. Pak platí, že $X_{i,j} = 1$, právě když první prvek v posloupnosti $(c_1, c_2, \dots, c_n)$ z množiny $\{b_l \mid i \le l \le j\}$ je buď $b_i$ nebo $b_j$. Protože posloupnost $(c_1, c_2, \dots, c_n)$ se chová jako náhodná s rovnoměrným rozložením vzhledem k posloupnosti $(b_1, b_2, \dots, b_n)$, tak pravděpodobnost tohoto jevu je $\frac{2}{j-i+1}$, tedy $p_{i,j} = \frac{2}{j-i+1}$ pro $1 \le i < j \le n$. Odtud očekávaný počet porovnání algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \le$$
$$2 \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{1}{k} = 2n \left( \sum_{k=2}^{n} \frac{1}{k} \right) \le 2n \int_{1}^{n} \frac{1}{x} dx =$$
$$2n \ln n.$$

**Druhý výpočet.**

Označme $QS(n)$ očekávaný počet porovnání provedený algoritmem **QUICKSORT** při třídění $n$-členné posloupnosti. Pak platí

$$QS(0) = QS(1) = 0 \text{ a}$$
$$QS(n) = \frac{1}{n} \left( \sum_{k=0}^{n-1} n - 1 + QS(k) + QS(n-k-1) \right) =$$
$$n - 1 + \frac{2}{n} \left( \sum_{k=0}^{n-1} QS(k) \right).$$

Z toho dostáváme, že

$$nQS(n) = n(n-1) + 2 \sum_{k=0}^{n-1} QS(k) \text{ a}$$
$$(n+1) QS(n+1) = (n+1) n + 2 \sum_{k=0}^{n} QS(k)$$

a tedy

$$QS(n+1) = \frac{2n}{n+1} + \frac{n+2}{n+1} QS(n).$$

Protože $\frac{i-1}{i} \le 1$ pro každé $i \ge 1$, dostaneme, že

$$QS(n) = \sum_{i=2}^{n} \frac{n+1}{i+1} \frac{2(i-1)}{i} \le 2(n+1) \left( \sum_{i=2}^{n} \frac{1}{i+1} \right) =$$
$$2(n+1) \left( \sum_{i=3}^{n+1} \frac{1}{i} \right) \le 2(n+1) \left( \left( \int_{i=1}^{n+1} \frac{1}{x} dx \right) - \frac{1}{2} \right) =$$
$$2n \ln(n+1) + 2 \ln(n+1) - n - 1.$$

Pro dostatečně velká $n$ platí

$$2n \ln(n+1) + 2 \ln(n+1) - n \le 2n \ln n.$$

**Porovnání algoritmů.**

Nyní porovnáme složitost algoritmů **HEAPSORT**, **MERGESORT**, **QUICKSORT**, **A-sort**, **SELECTIONSORT**, **INSERTIONSORT**. Připomeňme, že **SELECTION-SORT** třídí posloupnost tak, že jedním průchodem nalezne její nejmenší prvek, který vyřadí a vloží do výsledné posloupnosti. Tento proces pak opakuje se zbytkem původní posloupnosti (tato idea byla základem algoritmu **HEAPSORT**). **INSERTIONSORT** třídí tak, že do již setříděné části posloupnosti vkládá další prvek, který pomocí výměn zařadí na správné místo, a tento proces opakuje.

**QUICKSORT** v nejhorším případě vyžaduje čas $\Theta\left(n^2\right)$, očekávaný čas je $9n\log n$, v nejhorším případě vyžaduje $\frac{n^2}{2}$ porovnání, očekávaný počet porovnání je $1.44n\log n$, používá $n+\log n+$konstanta paměti. Používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
**HEAPSORT** v nejhorším případě vyžaduje čas $20n\log n$, očekávaný čas je $\leq 20n\log n$, v nejhorším i v očekávaném případě vyžaduje $2n\log n$ porovnání, používá $n+$ konstanta paměti. Používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
**MERGESORT** v nejhorším případě vyžaduje čas $12n\log n$, očekávaný čas je $\leq 12n\log n$, v nejhorším i v očekávaném případě vyžaduje $n\log n$ porovnání, používá $2n+$ konstanta paměti. Používá sekvenční přístup k paměti a verze, kterou jsme uvedli, je adaptivní na předtříděné posloupnosti, které se skládají z malého počtu dlouhých setříděných úseků.
**A-sort** vyžaduje čas $O\left(n\log\frac{F}{n}\right)$ v nejhorším i v očekávaném případě, kde $F$ je počet inverzí ve vstupní posloupnosti, stejně tak počet porovnání je $O\left(n\log\frac{F}{n}\right)$ v nejhorším i v očekávaném případě, používá $5n+$ konstanta paměti. Používá přímý přístup k paměti a je adaptivní na předtříděné posloupnosti s malým počtem inverzí.
**SELECTIONSORT** v nejhorším i v očekávaném případě potřebuje čas $2n^2$, počet porovnání v nejhorším i v očekávaném případě je $\frac{n^2}{2}$, používá $n+$ konstanta paměti. Používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
**INSERTIONSORT** v nejhorším i v očekávaném případě vyžaduje čas $O\left(n^2\right)$, počet porovnání v nejhorším případě je $\frac{n^2}{2}$, v očekávaném případě $\frac{n^2}{4}$, používá $n+$ konstanta paměti. Používá sekvenční přístup k paměti a má verzi, která je adaptivní na předtříděné posloupnosti s malým počtem inverzí.

Čas prezentovaný ve výsledcích byl spočítán pro model RAM (viz Mehlhorn).

Očekávaný čas pro **HEAPSORT** je prakticky stejný jako nejhorší čas. Byly navrženy verze, které optimalizují počet porovnání, ale většinou mají větší nároky na čas, a proto až na výjimky nejsou vhodné. Situace pro **MERGESORT** je komplikovanější, hodně závisí na konkrétní verzi algoritmu. Algoritmus **MERGESORT** je nejvhodnější pro externí paměti, protože používá sekvenční přístup, pro interní paměť kvůli vetší prostorové náročnosti (například je dvojnásobná proti **HEAQPSORT**u) není doporučován. Také se hodí pro návrh paralelních algoritmů. Pro třídění krátkých posloupností je doporučeno místo **QUICKSORTU** pro posloupnosti délky $\leq 22$ použít **SELECTIONSORT** a pro posloupnosti délky $\leq 15$ použít **INSERTIONSORT.** To vede k návrhu algoritmu, který pro dlouhé posloupnosti pracuje jako **QUICKSORT,** a když volá rekurzivně sám sebe na krátkou posloupnost, pak použije **SELECTIONSORT** nebo **INSERTIONSORT**. V algoritmu **A-sort** se doporučuje použít $(2,3)$-strom. Poměr časů v klasických počítačích vyžadovaných algoritmy **QUICKSORT**, **MERGESORT** a **HEAPSORT** je $1 : 1.33 : 2.22$ (viz Mehlhorn). To však nemusí být pravda pro RISK-architekturu ani pro cache-paměti apod.

V algoritmu **MERGESORT** jsme použili frontu, která řídila slučování posloupností. Tato metoda je uspokojující a dává optimální výsledek, pokud posloupnosti ve frontě jsou stejně dlouhé. Pokud se jejich délky hodně liší, nedosáhneme optimálního výsledku. Budeme řešit následující problém, který se poprvné vyskytl při návrhu Huffmanova kódu.

Vstup: Množina rostoucích navzájem disjunktních posloupností.
Úkol: Pomocí operace **MERGE** co nejrychleji spojit všechny tyto posloupnosti do jediné rostoucí posloupnosti.

Předpokládejme, že máme postup, který vytvoří jedinou posloupnost. Tento postup určuje úplný binární strom $T$ (tj. strom, kde každý vnitřní vrchol má dva syny) takový, že vstupní posloupnosti ohodnocují listy a každá posloupnost vzniklá slučováním ohodnocuje některý vnitřní vrchol tak, že platí:

> když $P(v)$ je posloupnost ohodnocující vrchol $v$ a $v_1$ a $v_2$ jsou synové $v$, pak $P(v) =$**MERGE**$(P(v_1), P(v_2))$.

Pro posloupnost $P$ označme $l(P)$ její délku. Pak součet časů, které v tomto procesu vyžaduje podprocedura **MERGE**, je $O\left(\sum \{l(P(v)) \mid v \text{ je vnitřní vrchol stromu } T\}\right)$. Indukcí lehce dostaneme, že

$$\sum \{l(P(v)) \mid v \text{ vnitřní vrchol stromu } T\} = \sum_{t \text{ list } T} d(t)\, l(P(t)),$$

kde $d(t)$ je hloubka listu $t$.

Když $T$ je úplný binární strom takový, že listy jsou ohodnoceny rostoucími navzájem disjunktními posloupnostmi, pak algoritmus **Slevani** spojí tyto posloupnosti do jediné rostoucí posloupnosti a procedury **MERGE** vyžadují čas

$$O\left(\sum_{t \text{ list } T} d(t)\, l(P(t))\right).$$

**Slevani**$(T, \{P(l) \mid \ l \text{ list } T\})$
**while** $P(\text{kořen } T)$ není definováno **do**
vezměme vrchol $v$ takový, že $P(v)$ není definováno a pro oba syny $v_1$ a $v_2$ vrcholu $v$ jsou $P(v_1)$ a $P(v_2)$ definovány, položme $P(v)$ :=**MERGE**$(P(v_1), P(v_2))$
**enddo**

Nyní můžeme přeformulovat původní problém:
Vstup: $n$ čísel $x_1, x_2, \ldots, x_n$
Výstup: úplný binární strom $T$ s $n$ listy a bijekce $\phi$ z množiny $\{1, 2, \ldots, n\}$ do listů $T$ taková, že $\sum_{i=1}^{n} d(\phi(i))\, x_i$ je minimální, kde $d(\phi(i))$ je hloubka listu $\phi(i)$.

Řekneme, že dvojice $(T, \phi)$ je optimální strom vzhledem k $x_1, x_2, \ldots, x_n$, když součet $\sum_{i=1}^{n} d(\phi(i))\, x_i$ je nejmenší možný.

V přeformulované úloze už nepracujeme s posloupnostmi, ale jen s jejich délkami. To znamená, když pro původní úlohu byly vstupem posloupnosti $P_1, P_2, \ldots, P_n$, pak pro přeformulovanou úlohu jsou vstupem jen jejich délky $l(P_1), l(P_2), \ldots, l(P_n)$. Vytvořený strom je pak použit v algoritmu **Slevani** (kde posloupnost $P_i$ nahradí při ohodnocení svoji délku).

Pro úplný binární strom $T$ s $n$ listy a bijekci $\phi$ z množiny $\{1,2,\ldots,n\}$ do listů stromu $T$ definujme

$$\mathrm{Cont}\,(T,\phi)=\sum_{i=1}^{n} d\,(\phi\,(i))\,x_i,$$

kde $d\,(\phi\,(i))$ je hloubka listu $\phi\,(i)$, tj. délka cesty z kořene do listu $\phi\,(i)$ pro $i=1,2,\ldots,n$. Chceme zkonstruovat úplný binární strom s $n$ listy, který minimalizuje hodnotu Cont. Zavedeme proto pojem kořenový les – to je disjunktní sjednocení kořenových stromů. Vrchol se nazývá list lesa, když je listem některého stromu, jejichž sjednocením les vznikl. Velikost lesa $V$, značíme ji $|V|$, je počet stromů, jejichž sjednocením $V$ vznikl. Uvažujme následující algoritmus.

**Optim**$(x_1,x_2,\ldots x_n)$:
$V$ je kořenový les vzniklý disjunktním sjednocením $n$ jednoprvkových stromů,
$\phi$ je bijekce mezi $\{1,2,\ldots,n\}$ a listy lesa $V$
**for every** $v\in V$ **do** $c\,(v):=x_{\phi^{-1}(v)}$ **enddo**
**while** $|V|>1$ **do**
vezmeme z $V$ dva stromy $T_1$ a $T_2$ s nejmenším ohodnocením, odstraníme je z $V$,
vytvoříme strom $T$ jako disjunktní sjednocení $T_1$, $T_2$ a nového vrcholu $v$,
$v$ je kořen $T$ a jeho dva synové jsou kořeny stromů $T_1$ a $T_2$,
$c\,(T)=c\,(T_1)+c\,(T_2)$, $T$ vložíme do $V$
**enddo**
**Výstup**: $(V,\phi)$.

**Věta.** *Pro danou posloupnost čísel* $(x_1,x_2,\ldots,x_n)$ *algoritmus* **Optim** *nalezne optimální strom pro množinu* $x_1,x_2,\ldots,x_n$ *a pokud je posloupnost* $(x_1,x_2,\ldots,x_n)$ *neklesající, pak vyžaduje čas* $O\,(n)$.

*Důkaz.* Nejprve si všimněme, že v každém okamžiku je $\phi\,(i)$ list lesa $V$ pro každé $i\in\{1,2,\ldots,n\}$. Algoritmus končí, když $|V|=1$, tedy, když $V$ je strom. Když v daném okamžiku $V$ vznikl disjunktním sjednocením stromů $T_1,T_2,\ldots,T_k$ a na stromy $T_1$ a $T_2$ použijeme popsaný proces, pak dostaneme les $V'$ vzniklý ze stromů $T,T_3,\ldots,T_k$, tedy $|V'|+1=|V|$. Protože na začátku $V$ obsahoval jen $n$ jednoprvkových stromů, tak algoritmus po konečném počtu kroků skončí. Ukázali jsme, že každý běh cyklu **while do** zmenší počet stromů o jeden, ale nezmění množinu listů. Proto výsledný les $V$ je strom s $n$ listy a $\phi$ je bijekce z $\{1,2,\ldots,n\}$ do množiny listů $V$. Zbývá ukázat, $(V,\phi)$ je optimální strom vzhledem k $x_1,x_2,\ldots,x_n$. Dokážeme to indukcí podle $n$. Když $n=2$, tak tvrzení zřejmě platí. Předpokládejme, že tvrzení platí pro každou posloupnost čísel $(y_1,y_2,\ldots,y_{n-1})$ a nechť $x_1\leq x_2\leq\cdots\leq x_n$ je posloupnost čísel. Bez újmy na obecnosti můžeme předpokládat, že v prvním kroku algoritmus **Optim** zvolil stromy $\phi\,(1)$ a $\phi\,(2)$. Uvažujme množinu $(y_1,y_2,\ldots,y_{n-1})$, kde $y_i=x_{i+2}$ pro $i=1,2,\ldots,n-2$, $y_{n-1}=x_1+x_2$. Nechť $V'$ je strom získaný ze stromu $V$ odstraněním listů $\phi\,(1)$ a $\phi\,(2)$ a nechť $\psi$ je bijekce z množiny $\{1,2,\ldots,n-1\}$ taková, že $\psi\,(i)=\phi\,(i+2)$ pro $i=1,2,\ldots,n-2$ a $\psi\,(n-1)$ =otec listu $\phi\,(1)$. Pak můžeme předpokládat, že algoritmus **Optim**$(y_1,y_2,\ldots,y_{n-1})$ zkonstruoval $(T',\psi)$. Podle indukčního předpokladu je to optimální strom pro $(y_1,y_2,\ldots,y_{n-1})$. Nechť $(U,\theta)$ je optimální strom vzhledem k $(x_1,x_2,\ldots,x_n)$. Zvolme vnitřní vrchol $u$ ve stromě $U$ s největší hloubkou. Pak synové $u_1$ a $u_2$ vrcholu $u$ jsou listy. Nechť $i,j\in\{1,2,\ldots,n\}$ takové, že $\theta\,(i)=u_1$, $\theta\,(j)=u_2$. Můžeme předpokládat, že když $i,j\in\{1,2\}$, pak $i=1$ a $j=2$. Definujme $\eta$ z $\{1,2,\ldots,n\}$ do listů $U$ tak, že $\eta\,(1)=u_1$, $\eta\,(2)=u_2$, $\eta\,(i)=\theta\,(1)$,

$\eta(j) = \theta(2)$ a $\eta(k) = \theta(k)$ pro všechna $k \in \{3, 4, \ldots, n\} \setminus \{i, j\}$. Pak $\eta$ je bijekce a

$$\operatorname{Cont}(U, \eta) - \operatorname{Cont}(U, \theta) = (d(u_1) - d(\theta(1)))(x_1 - x_i) + (d(u_2) - d(\theta(2)))(x_2 - x_j).$$

Z volby $u$ plyne, že $d(u_1) \geq d(\theta(1))$, $d(u_2) \geq d(\theta(2))$, $x_1 \leq x_i$ a $x_2 \leq x_j$. Odtud

$$(d(u_1) - d(\theta(1)))(x_1 - x_i) + (d(u_2) - d(\theta(2)))(x_2 - x_j) \leq 0$$

a protože $(U, \theta)$ je optimální strom pro $(x_1, x_2, \ldots, x_n)$, dostáváme, že $(U, \eta)$ je také optimální strom pro $(x_1, x_2, \ldots, x_n)$. Odstraněním listů $u_1$ a $u_2$ ze stromu $U$ dostaneme strom $U'$. Definujme $\tau$ z $\{1, 2, \ldots, n-1\}$ předpisem $\tau(i) = \eta(i+2)$ pro $i = 1, 2, \ldots, n-2$ a $\tau(n-1) = u$. Pak $\tau$ je bijekce z $\{1, 2, \ldots, n-1\}$ do množiny listů $U'$ a protože $(T', \psi)$ je optimální strom pro $(y_1, y_2, \ldots, y_{n-1})$, dostáváme, že

$$\operatorname{Cont}(T', \psi) \leq \operatorname{Cont}(U', \tau).$$

Protože platí

$$\operatorname{Cont}(T, \phi) = \operatorname{Cont}(T, \psi) + x_1 + x_2 \text{ a}$$
$$\operatorname{Cont}(U, \eta) = \operatorname{Cont}(U', \tau) + x_1 + x_2$$

dostáváme, že $(T, \phi)$ je optimální strom pro $(x_1, x_2, \ldots, x_n)$. Předpokládejme, že $x_1 \leq x_2 \leq \cdots \leq x_n$ a že algoritmus postupně vytváří víceprvkové stromy $T_1, T_2, \ldots, T_k$. Pak indukcí okamžitě dostáváme, že $c(T_1) \leq c(T_2) \leq \cdots \leq c(T_k)$. Tedy stačí, když použijeme následující strukturu: rostoucí posloupnost prvků $x_1, x_2, \ldots, x_n$, ukazatel, který v každém okamžiku ukazuje na nejmenší prvek této posloupnosti, který je reprezentován listem v jednoprvkovém stromě (pak před ukazatelem jsou prvky, které jsou reprezentovány listem ve víceprvkovém stromě a za ukazatelem jsou prvky reprezentované listem v jednoprvkovém stromě) a frontu víceprvkových stromů (to znamená, že stromy odebíráme zepředu a ukládáme je dozadu). Udržovat tyto struktury vyžaduje čas $O(1)$ stejně jako nalezení dvou stromů s nejmenším ohodnocením. Tedy algoritmus **Optim** zkonstruuje optimální strom v čase $O(n)$. □

Při aplikaci na naši původní úlohu musíme ještě setřídit vstupní posloupnost (délek) pro přeformulovanou úlohu. Tato posloupnost je tvořena přirozenými čísly a délku maximální posloupnosti nalezneme v čase úměrném součtu délek posloupností. Na její setřídění pak můžeme použít algoritmus **BUCKETSORT** (popíšeme si ho v následující přednášce), který vyžaduje čas $O(n+m)$, kde $n$ je počet posloupností a $m$ je maximální délka posloupnosti.

**Věta.** *Uvedený algoritmus spojí disjunktní rostoucí posloupnosti $P_1, P_2, \ldots, P_n$ o délkách $l(P_1), l(P_2), \ldots, l(P_n)$ do jediné rostoucí posloupnosti v čase $O\left(\sum_{i=1}^{n} l(P_i)\right)$.*

## Rozhodovací stromy

Většina obecných třídicích algoritmů používá jedinou primitivní operaci mezi prvky vstupní posloupnosti, a to jejich vzájemné porovnání, jiné operace nejsou používány. To znamená, že práci takového algoritmu pro $n$-prvkové posloupnosti lze popsat binárním

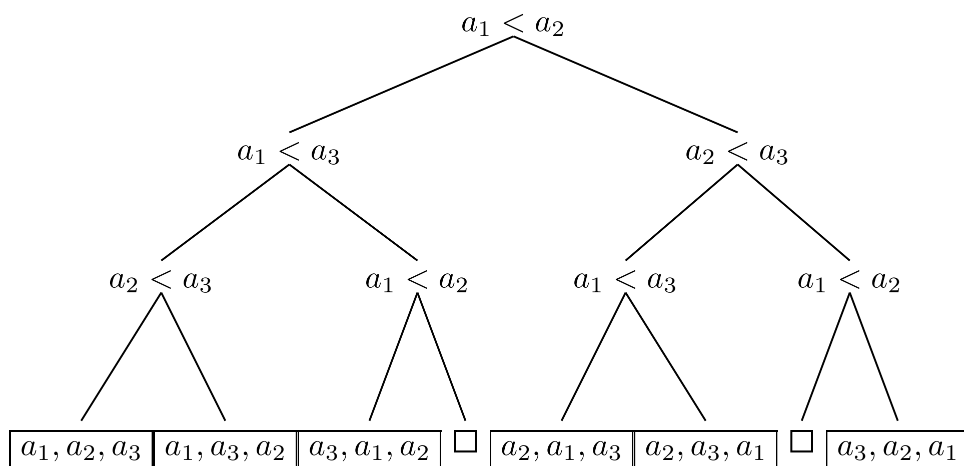
stromem, jehož vnitřní vrcholy jsou ohodnoceny porovnáním dvou prvků vstupní posloupnosti (např. $a_i < a_j$). Bez újmy na obecnosti předpokládáme, že vstupní posloupnost je permutace $\pi$ množiny $\{1, 2, \ldots, n\}$, a tato permutace prochází stromem takto:

> Začíná v kořeni stromu. Když je ve vnitřním vrcholu $v$ ohodnoceném porovnáním $a_i \leq a_j$, pak $\pi(i) < \pi(j)$ znamená, že pokračuje v levém synu vrcholu $v$, a $\pi(j) < \pi(i)$ znamená pokračování v pravém synu vrcholu $v$. Proces třídění končí, když se dostane do listu.

Aby byl algoritmus korektní, musí platit, že dvě různé permutace skončí v různých listech. To znamená, že definovaný strom pak musí mít alespoň $n!$ listů. Délka cesty z kořene do listu, kde skončila permutace $\pi$, dává dolní odhad na čas potřebný k setřídění posloupnosti $\pi$. To nám umožňuje získat dolní odhad času potřebného k setřídění posloupnosti. Korektnost těchto úvah plyne z pozorování, že když porovnání je jediná primitivní operace, pak algoritmus není závislý na prvcích vstupní posloupnosti, ale jen na jejich vzájemném vztahu. Proto stačí uvažovat pouze permutace $n$-prvkové množiny, protože zachycují všechny možné vztahy v $n$-prvkové posloupnosti. Dále je třeba si uvědomit, že vztah mezi stromem pro $n$-prvkové posloupnosti a stromem pro $(n+1)$-prvkové posloupnosti je dán konkrétním algoritmem a nedá se popsat obecně.

V nevhodném algoritmu se může stát, že v některém listu neskončí žádná permutace. To se stane, když strom pro $n$-prvkové posloupnosti má více než $n!$ listů.

Následující obrázek ilustruje naše úvahy na **SELECTIONSORT**u pro 3-prvkové posloupnosti. Listy jsou ohodnoceny permutacemi množiny $\{1, 2, 3\}$, které v nich skončí nebo jsou prázdné.




Obr. 1

**Definice.** Mějme třídicí algoritmus **A**, který jako jedinou primitivní operaci pro prvky vstupní posloupnosti používá porovnání. Řekneme, že binární strom $T$, jehož vnitřní vrcholy jsou ohodnoceny porovnáními $a_i \leq a_j$ pro $i, j = 1, 2, \ldots, n$, $i \neq j$, je rozhodovacím stromem algoritmu **A** pro $n$-prvkové posloupnosti, když pro každou permutaci $\pi$ $n$-prvkové množiny platí

> posloupnost porovnání při třídění posloupnosti $\pi$ algoritmem **A** je stejná jako posloupnost porovnání při průchodu posloupnosti $\pi$ stromem $T$.

Pak korektnost algoritmu zajišťuje, že dvě různé permutace množiny $\{1, 2, \ldots, n\}$ skončí v různých listech stromu $T$. Dolním odhadem pro čas algoritmu **A** v nejhorším případě je délka nejdelší cesty z kořene do listu, protože algoritmus **A** při třídění permutace použije tolik porovnání, jako je délka její cesty stromem $T$. Proto při rovnoměrném rozdělení vstupních posloupností je očekávaný čas algoritmu **A** průměrná délka cesty z kořene do listu. Tato fakta motivují hledání následujících veličin.

Definujme
$S(n)$ jako minimum přes všechny stromy $T$ s alespoň $n!$ listy z délek nejdelších cest z kořene do listu $T$ a
$A(n)$ jako minimum přes všechny stromy $T$ s alespoň $n!$ listy z průměrných délek cest z kořene do listu v $T$.
Naším cílem je spočítat dolní odhad těchto veličin.

Když nejdelší cesta z kořene do listu v binárním stromě $T$ má délku $k$, pak $T$ má nejvýše $2^k$ listů. Proto $n! \leq 2^{S(n)}$. Odtud plyne $S(n) \geq \log_2 n!$. Připomeňme si Stirlingův vzorec pro faktoriál:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right).$$

Protože pro $n \geq 1$ je $\frac{1}{12n}, \frac{1}{n^2} \geq 0$, můžeme předpokládat, že $\left(1 + \frac{1}{12n} + O\left(\frac{1}{n^2}\right)\right) \geq 1$ pro všechna $n \geq 1$. Po zlogaritmování vzorce dostáváme

$$\log_2 n! \geq \frac{1}{2}\log_2 n + n\left(\log_2 n - \log_2 e\right) + \log_2 \sqrt{2\pi} \geq \left(n + \frac{1}{2}\right)\log_2 n - n\log_2 e.$$

Protože platí

$$e^1 = e = 2^{\log_2 e} = \left(e^{\ln 2}\right)^{\log_2 e} = e^{\ln 2 \log_2 e},$$

dostáváme, že $\frac{1}{\ln 2} = \log_2 e$, a tedy

$$S(n) \geq \log_2 n! \geq \left(n + \frac{1}{2}\right)\log_2 n - \frac{n}{\ln 2}.$$

Pro binární strom $T$ označme $B(T)$ součet všech délek cest z kořene do nějakého listu a položme

$$B(k) = \min\left\{B(T) \mid T \text{ je binární strom s } k \text{ listy}\right\}.$$

Když ukážeme, že $B(k) \geq k\log_2 k$, pak bude

$$A(n) \geq \frac{B(n!)}{n!} \geq \frac{n!\log_2 n!}{n!} = \log_2 n! \geq \left(n + \frac{1}{2}\right)\log_2 n - \frac{n}{\ln 2}.$$

Takže dokažme, že $B(T) \geq k \log_2 k$ pro každý binární strom $T$ s $k$ listy. Když ve stromě $T$ vynecháme každý vrchol, který má jen jednoho syna a tohoto syna spojíme s jeho předchůdcem, dostaneme úplný binární strom $T'$ s $k$ listy takový, že $B(T') \leq B(T)$. Proto se stačí omezit na úplné binární stromy. Když $T$ je úplný binární strom s jedním vrcholem, pak $B(T) = 0 = 1 \log_2 1$, když $T$ je úplný binární strom s dvěma listy, pak $B(T) = 2 = 2 \log_2 2$. Tedy platí $B(1) \geq 1 \log_2 1$ a $B(2) \geq 2 \log_2 2$. Předpokládejme, že $B(i) \geq i \log_2 i$ pro $i < k$, a nechť $T$ je úplný binární strom s $k$ listy. Nechť $T_1$ a $T_2$ jsou podstromy určené syny kořene $T$ a nechť $T_i$ má $k_i$ listů pro $i = 1, 2$. Pak $1 \leq k_1, k_2$ a $k_1 + k_2 = k$, tedy $k_1, k_2 < k$, a podle indukčního předpokladu $B(k_i) \geq k_i \log_2 k_i$. Odtud

$$\begin{aligned} B(T) = & k_1 + B(T_1) + k_2 + B(T_2) \geq k + B(k_1) + B(k_2) \geq \\ & k + k_1 \log_2 k_1 + k_2 \log_2 k_2. \end{aligned}$$

Tedy stačí ukázat, že

$$k + k_1 \log_2 k_1 + k_2 \log_2 k_2 \geq k \log_2 k$$

pro všechna $k_1, k_2 > 0$ taková, že $k = k_1 + k_2$. Toto je ekvivalentní s tvrzením, že pro $k > 0$ platí

$$f(x) = x \log_2 x + (k - x) \log_2 (k - x) + k - k \log_2 k \geq 0,$$

kde $x \in (0, k)$. Abychom to dokázali, všimněme si, že $f\left(\frac{k}{2}\right) = 0$. Nyní spočítáme derivaci $f$.

$$f'(x) = \log_2 x + \log_2 e - \log_2 (k - x) - \log_2 e = \log_2 \frac{x}{k - x}.$$

Když $x \in \left(0, \frac{k}{2}\right)$, pak $f'(x) < 0$ a $f$ je na tomto intervalu klesající, když $x \in \left(\frac{k}{2}, k\right)$, pak $f'(x) > 0$ a $f$ je na tomto intervalu rostoucí. Odtud plyne, že $f(x) \geq 0$ pro $x \in (0, k)$. Tím jsme dokázali, že $A(n) \geq \left(n + \frac{1}{2}\right) \log_2 n - \frac{n}{\ln 2}$. Shrneme naše výsledky.

**Věta.** *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* $\left\lceil \left(n + \frac{1}{2}\right) \log_2 n - \frac{n}{\ln 2} \right\rceil$ *porovnání a očekávaný počet porovnání při rovnoměrném rozdělení vstupních posloupností je alespoň* $\left(n + \frac{1}{2}\right) \log_2 n - \frac{n}{\ln 2}$.

V tomto výsledku lze oslabit předpoklady. Věta platí i za předpokladu, že třídicí algoritmus nepoužívá nepřímé adresování a celočíselné dělení. Tato metoda pro nalezení dolního odhadu se používá i pro vyčíslování algebraických funkcí a při algoritmickém řešení geometrických úloh. Na druhé straně klasický algoritmus **BUCKETSORT** ukazuje, že se nelze předpokladů ve větě úplně zbavit. V následujících algoritmech předpokládáme, že $Q_i$ jsou spojové seznamy, nový prvek se vkládá na konec seznamu a konkatenace seznamů závisí na jejich pořadí. V seznamech máme okamžitý přístup k prvnímu a poslednímu prvku (pomocí ukazatelů na tyto prvky).

**BUCKETSORT**$(a_1, a_2, \ldots, a_n, m)$:
Komentář: Vstup je přirozené číslo $m$ a posloupnost přirozených čísel $a_1, a_2, \ldots, a_n$ z intervalu $< 0, m >$. Cílem je setřídit posloupnost $a_1, a_2, \ldots, a_n$.
**for every** $i = 0, 1, \ldots, m$ **do** $Q_i = \emptyset$ **enddo**
**for every** $i = 1, 2, \ldots, n$ **do**
  $a_i$ vlož na konec seznamu $Q_{a_i}$
**enddo**

$i := 0$, $P := \emptyset$
**while** $i \le m$ **do**
  $P$ :=konkatenace $P$ a $Q_i$, $i := i+1$
**enddo**
$P$ je neklesající posloupnost prvků $a_1, a_2, \ldots, a_n$.

Algoritmus nevyžaduje, aby prvky ve vstupní posloupnosti byly různé. Ve výstupní posloupnosti se daný prvek opakuje tolikrát, kolikrát se opakoval ve vstupní posloupnosti, se zachováním pořadí (algoritmus je stabilní). Konkatenace dvou seznamů a vložení prvku do seznamu vyžadují čas $O(1)$. Proto první a třetí cyklus vyžadují čas $O(m)$ a druhý cyklus čas $O(n)$. Tedy algoritmus vyžaduje $O(n+m)$ času a paměti. Když $m = O(n)$, tak pro algoritmus neplatí tvrzení věty. Důvod je, že nejsou splněny předpoklady, protože druhý cyklus používá nepřímé adresování.

Nyní uvedeme dvě sofistikovanější verze tohoto algoritmu. V první předpokládáme, že $a_1, a_2, \ldots, a_n$ je posloupnost reálných čísel z intervalu $< 0, 1 >$ a $\alpha$ je pevně zvolené kladné reálné číslo.

**HYBRIDSORT**$(a_1, a_2, \ldots, a_n)$:
$k := \alpha n$
**for every** $i = 0, 1, \ldots, k$ **do** $Q_i := \emptyset$ **enddo**
**for every** $i = 1, 2, \ldots, n$ **do**
  $a_i$ vlož na konec seznamu $Q_{\lceil k a_i \rceil}$
**enddo**
$i := 0$, $P := \emptyset$
**while** $i \le k$ **do**
  **HEAPSORT**$(Q_i)$
  $P$ :=konkatenace $P$ a $Q_i$, $i := i+1$
**enddo**
$P$ je rostoucí posloupnost prvků $a_1, a_2, \ldots, a_n$.

**Věta.** *Algoritmus* **HYBRIDSORT** *setřídí posloupnost reálných čísel z intervalu* $< 0, 1 >$ *v nejhorším případě v čase* $O(n \log n)$. *Když prvky* $a_i$ *mají rovnoměrné rozložení a jsou na sobě nezávislé, pak očekávaný čas algoritmu* **HYBRIDSORT** *je* $O(n)$.

*Důkaz.* První dva cykly v algoritmu vyžadují čas $O(n)$, $i$-tý běh třetího cyklu vyžaduje nejvýše čas $O(1 + |Q_i| \log |Q_i|)$. Proto třetí cyklus vyžaduje čas

$$O\left(\sum_{i=0}^{k} (1 + |Q_i| \log |Q_i|)\right) = O\left(\sum_{i=0}^{k} (1 + |Q_i| \log n)\right) = O\left(k + \left(\sum_{i=0}^{k} |Q_i|\right) \log n\right) = O(n \log n)$$

a celkový čas **HYBRIDSORT**u je nejvýše $O(n \log n)$.

Označme $X_i = |Q_i|$, pak můžeme předpokládat, že $X_i$ je náhodná proměnná. Protože pravděpodobnost, že $x \in Q_i$, je $\frac{1}{k}$, dostáváme, že

$$\operatorname{Prob}(X_i = q) = \binom{n}{q} \left(\frac{1}{k}\right)^q \left(1 - \frac{1}{k}\right)^{n-q}.$$

Očekávaný čas vyžadovaný třetím cyklem se pak rovná

$$E\left(\sum_{i=0}^{k} 1+X_i\log X_i\right) = k+\sum_{i=0}^{k}\sum_{q=1}^{n} q\log q\binom{n}{q}\left(\frac{1}{k}\right)^q(1-frac1k)^{n-q}\leq$$
$$k+k\sum_{q=2}^{n} q^2\binom{n}{q}\left(\frac{1}{k}\right)^q\left(1-\frac{1}{k}\right)^{n-q}=$$
$$k+k\left(\frac{n\,(n-1)}{k^2}+\frac{n}{k}\right)=O\,(n)\,,$$

protože $k=\alpha n$ a

$$q^2\binom{n}{q}=(q\,(q-1)+q)\binom{n}{q}=n\,(n-1)\binom{n-2}{q-2}+n\binom{n-1}{q-1}.$$

(Jedná se vlastně o známý výpočet druhého momentu binomického rozdělení.) □

Nyní použijeme modifikaci **BUCKETSORT**u k setřídění slov. Máme totálně uspořádanou abecedu a chceme lexikograficky uspořádat slova $a_1,a_2,\ldots,a_n$ nad touto abecedou. Připomeňme, že když $a=x_1x_2\ldots x_n$ a $b=y_1y_2\ldots y_m$ jsou dvě slova nad totálně uspořádanou abecedou $\Sigma$, pak $a<b$ v lexikografickém uspořádání, právě když existuje $i=0,1,\ldots,\min\{n,m\}$ takové, že $x_j=y_j$ pro každé $j=1,2,\ldots,i$ a buď $n=i<m$ nebo $i<\min\{n,m\}$ a $x_{i+1}<y_{i+1}$. Předpokládejme, že $a_i=a_i^1a_i^2\ldots a_i^{l(i)}$, kde $a_i^j\in\Sigma$ a $l\,(i)$ je délka $i$-tého slova $a_i$.

**WORDSORT**$(a_1,a_2,\ldots,a_n)$:
**for every** $i=1,2,\ldots,n$ **do** $l\,(i)$ :=délka slova $a_i$ **enddo**
$l:=\max\{l\,(i)\mid i=1,2,\ldots,n\}$
**for every** $i=1,2,\ldots,l$ **do** $L_i:=\emptyset$ **enddo**
**for every** $i=1,2,\ldots,n$ **do**
$a_i$ vložíme do $L_{l(i)}$
**enddo**
Komentář: Pro každé $i$ množina $L_i$ obsahuje všechna slova z množiny $\{a_1,a_2,\ldots,a_n\}$ o délce $i$.
$P:=\left\{\left(j,a_i^j\right)\mid 1\leq i\leq n,\,1\leq j\leq l\,(i)\right\}$
$P_1$ :=**BUCKETSORT**$(P)$ podle druhé komponenty
$P_2$ :=**BUCKETSORT**$(P_1)$ podle první komponenty
**for every** $i=1,2,\ldots,l$ **do** $S_i:=\emptyset$ **enddo**
$(i,x)$ :=první prvek $P_2$
**while** $(i,x)\neq NIL$ **do**
$(i,x)$ vložíme do $S_i$
$(i,x)$ :=následník $(i,x)$ v $P_2$
**enddo**
Komentář: V $S_i$ jsou všechny dvojice $(i,x)$ takové, že $x$ je $i$-tým písmenem abecedy a když $x<y$, pak $(i,x)$ je před $(i,y)$.
**for every** $s\in\Sigma$ **do** $T_s:=\emptyset$ **enddo**
$T:=\emptyset$, $i:=l$
**while** $i>0$ **do**
$T:=L_i$ konkatenace s $T$, $a$ :=první slovo v $T$

**while** $a \neq NIL$ **do**
  $s :=$ $i$-té písmeno $a$, vložme $a$ do $T_s$
  $a :=$následník $a$ v $T$
**enddo**
$(i,x)$ :=první prvek v $S_i$, $T := \emptyset$
**while** $(i,x) \neq NIL$ **do**
  $T := T$ konkatenace s $T_x$, $T_x := \emptyset$
  $(i,x)$ :=následník $(i,x)$ v $S_i$
**enddo**
$i := i - 1$
**enddo**
$T$ je setříděná posloupnost slov $a_1, a_2, \ldots, a_n$.

Uvažujme běh posledního cyklu pro dané $i$. Po jeho skončení jsou v $T$ všechna slova z množiny $a_1, a_2, \ldots, a_n$, která mají délku alespoň $i$, a když slovo $a_q$ je před $a_r$ v seznamu $T$, pak existuje $j = i-1, i, \ldots, l$ takové, že $a_r^k = a_q^k$ pro každé $k = i, i+1, \ldots, j$ a buď $l(r) = j \leq l(q)$ nebo $j < \min\{l(r), l(q)\}$ a $a_r^{j+1} < a_q^{j+1}$. To dostáváme indukcí podle $i$ (viz **BUCKETSORT**). Jediný a hlavní rozdíl proti **BUCKETSORT**u je, že neprocházíme všechny přihrádky $T_x$, ale pouze neprázdné přihrádky. To nám zajišťuje množina $S_i$, viz Komentář.

Označme $L = \sum_{i=1}^{n} l(i)$. První cyklus (spočítání délek slov) vyžaduje čas $O(L)$. Druhý cyklus vyžaduje čas $O(l) = O(L)$ a třetí cyklus čas $O(n) = O(L)$. Vytvoření seznamu $P_1$ vyžaduje čas $O(L)$ a jeho setřídění čas $O(L+l) = O(L)$, protože $P_1$ i $P_2$ mají nejvýše $L$ prvků. Další cyklus (založení seznamů $S_i$) vyžaduje čas $O(l)$ a následující cyklus vytvářející seznamy $S_i$ čas $O(L)$. Cyklus zakládající seznamy $T_x$ vyžaduje čas $O(|\Sigma|)$. Běhy dalšího cyklu jsou indexovány $i = 1, 2, \ldots, l$. Pro každé $i$ označme $m_i$ počet slov z množiny $\{a_1, a_2, \ldots, a_n\}$, která mají délku alespoň $i$. Pak $L = \sum_{i=1}^{l} m_i$ a první vnitřní cyklus v $i$-tém běhu vnějšího cyklu vyžaduje čas $O(m_i)$ a druhý vnitřní cyklus v $i$-tém běhu vnějšího cyklu vyžaduje čas $O(|S_i|) = O(m_i)$. Tedy celkový čas algoritmu je $O(L+m)$, kde $m = |\Sigma|$ a $L$ je součet délek všech slov z množiny $a_1, a_2, \ldots, a_n$.

**Hledání $k$-tého prvku.**

Na závěr popíšeme dva algoritmy pro hledání $k$-tého nejmenšího prvku v dané podmnožině totálně uspořádaného univerza. První z nich využívá stejný princip jako **QUICKSORT**. Přesné znění problému:

Vstup: množina prvků $M = \{a_1, a_2, \ldots, a_n\}$ a číslo $i$ takové, že $1 \leq i \leq n$.
Výstup: prvek $a_k$ takový, že $|\{j \mid 1 \leq j \leq n,\ a_j \leq a_k\}| = i$.
Když $i = \frac{n}{2}$, pak $a_k$ se nazývá medián.

**Praktický algoritmus.**

**FIND**$(M = (a_1, a_2, \ldots, a_n), i)$:
zvolme $a \in M$, $M_1 := \{b \in M \mid b < a\}$, $M_2 := \{b \in M \mid b > a\}$
**if** $|M_1| > i - 1$ **then**
  **FIND**$(M_1, i)$
**else**
  **if** $|M_1| < i - 1$ **then**
    **FIND**$(M_2, i - |M_1| - 1)$
  **else**

$a$ je hledaný prvek
**endif**
**endif**

Korektnost algoritmu je zřejmá. V nejhorším případě voláme **FIND** $i$-krát a jedno volání vyžaduje čas $O(|M|)$. Tedy **FIND** v nejhorším případě vyžaduje čas $O\left(n^2\right)$. Dobré volby prvku $a$ mohou algoritmus značně zrychlit. Zde platí stejná diskuse jako pro **QUICK-SORT**. Spočítáme očekávaný čas, když prvek $a$ byl vybrán náhodně. Pak pravděpodobnost, že je $k$-tým nejmenším prvkem, je $\frac{1}{n}$, kde $n = |M|$. Označme $T(n,i)$ očekávaný čas algoritmu **FIND** pro nalezení $i$-tého nejmenšího prvku v $n$-prvkové množině $M$. Platí

$$T(n,i) = n + \frac{1}{n}\left(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)\right),$$

protože procedura **FIND** bez rekurzivního volání sama sebe vyžaduje čas $O(n)$. Předpokládejme, že $T(m,i) \leq 4m$ pro každé $m < n$ a každé $i$ takové, že $1 \leq i \leq m$. Pak

$$\begin{aligned} T(n,i) =& n + \frac{1}{n}\left(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)\right) \leq \\ & n + \frac{1}{n}\left(\sum_{k=1}^{i-1} 4(n-k) + \sum_{k=i+1}^{n} 4k\right) = \\ & n + \frac{4}{n}\left(\frac{(2n-i)(i-1)}{2} + \frac{(n+i+1)(n-i)}{2}\right) = \\ & n + \frac{4}{n}\left(\frac{n^2 + 2ni - n - 2i^2}{2}\right). \end{aligned}$$

Výraz v čitateli zlomku nabývá svého maxima pro $i = \frac{n}{2}$ a jeho maximalní hodnota je $\frac{3}{2}n^2 - n = \frac{3n^2-2n}{2}$. Tedy

$$T(n,i) \leq n + \frac{4}{n}\left(\frac{3n^2 - 2n}{4}\right) = n + 3n - 2 = 4n - 2 < 4n$$

Protože tento odhad platí také pro $n = 1$ a $n = 2$, ukázali jsme, že $T(n,i) \leq 4n$ pro všechna $n$ a všechna $i$ taková, že $1 \leq i \leq n$. Shrneme získané výsledky o algoritmu **FIND**.

**Věta.** *Algoritmus* **FIND** *nalezne $i$-tý nejmenší prvek v $n$ prvkové totálně uspořádané množině. V nejhorším případě vyžaduje čas $O\left(n^2\right)$, ale když se pivot volí náhodně nebo když všechny vstupní množiny mají stejnou pravděpodobnost, pak očekávaný čas je $O(n)$.*

Pro velmi malá $i$ nebo pro $i$ velmi blízká $n$ pracuje rychleji přímý přirozený algoritmus (udržuje si posloupnost $i$ nejmenších nebo $n-i$ největších prvků a k ní přidává další tak, že ten prvek, který překročil danou hranici, je zapomenut). Tento algoritmus však není efektivní pro obecná $i$.

**Teoretický algoritmus.**

Následující algoritmus nalezne $i$-tý nejmenší prvek v lineárním čase. Vstup je podmnožina $M$ totálně uspořádaného univerza $U$ a přirozené číslo $i$ takové, že $1 \leq i \leq |M|$.

**SELECT**$(M, i)$:
$n := |M|$
**if** $n \leq 100$ **then**
setřídíme množinu $M$ a najdeme $i$-tý nejmenší prvek $m$
**else**
rozdělíme $M$ do $\lceil \frac{n}{5} \rceil$ navzájem disjunktních pětiprvkových podmnožin $A_1, A_2, \ldots, A_{\lceil \frac{n}{5} \rceil}$ (poslední z podmnožin může mít méně než 5 prvků).
**for every** $j = 1, 2, \ldots, \lceil \frac{n}{5} \rceil$ **do**
najdeme medián $m_j$ množiny $A_j$
**enddo**
$\bar{m} :=$**SELECT**$\left(\{m_j \mid j = 1, 2, \ldots, \lceil \frac{n}{5} \rceil\}, \lceil \frac{n}{10} \rceil\right)$
$M_1 := \{m \in M \mid m < \bar{m}\}$, $M_2 := \{m \in M \mid \bar{m} < m\}$
**if** $|M_1| > i - 1$ **then**
$m :=$**SELECT**$(M_1, i)$
**else**
**if** $|M_1| < i - 1$ **then**
$m :=$**SELECT**$(M_2, i - |M_1| - 1)$
**else**
$m := \bar{m}$
**endif**
**endif**
**Výstup**: $m$
**endif**

Korektnost algoritmu je zřejmá, zbývá vyšetřit složitost. Nejprve ukážeme, že

**Lemma.** *Když* $n \geq 100$, *pak* $|M_1|, |M_2| \leq \frac{8n}{11}$.

*Důkaz.* Pro $j \leq \lfloor \frac{n}{5} \rfloor$, když $m_j < \bar{m}$, pak $|A_j \cap M_1| \geq 3$, když $m_j > \bar{m}$, pak $|A_j \cap M_2| \geq 3$, když $m_j = \bar{m}$, pak $|A_j \cap M_1| = |A_j \cap M_2| = 2$. Protože $|\{j = 0, 1, \ldots, \lfloor \frac{n}{5} \rfloor \mid m_j < \bar{m}\}|, |\{j = 0, 1, \ldots, \lfloor \frac{n}{5} \rfloor \mid m_j > \bar{m}\}| \geq \lfloor \frac{n}{10} \rfloor$, dostáváme $|M_1|, |M_2| \geq \lfloor \frac{3n}{10} \rfloor - 1$. Protože platí $M_1 \cap M_2 = \emptyset$ a $M_1 \cup M_2 = M \setminus \{\bar{m}\}$ a protože $\frac{8n}{11} + \lfloor \frac{3n}{10} \rfloor - 1 \geq \frac{113n}{110} - 2 \geq n$ když $n > 100$, dostáváme požadovaný odhad. □

Maximální čas vyžadovaný algoritmem **SELECT**$(M, i)$ pro $|M| = n$ označme $T(n)$. Když $n \leq 100$, pak zřejmě existuje konstanta $a$ taková, že $T(n) \leq an$. Když $n > 100$, pak $\lceil \frac{n}{5} \rceil \leq \frac{21n}{100}$, a protože **SELECT**$(M, i)$ pro $|M| > 100$ bez rekurentních volání vyžaduje čas $O(|M|)$, dostáváme pro $n > 100$, že $T(n) \leq T\left(\frac{21n}{100}\right) + T\left(\frac{8n}{11}\right) + bn$ pro nějakou konstantu $b$. Zvolme $c \geq \max\left\{a, \frac{1100b}{69}\right\}$. Ukážeme, že $T(n) \leq cn$. Když $n \leq 100$, tak tvrzení platí, protože $a \leq c$. Když $n > 100$, pak $\lceil \frac{21n}{100} \rceil, \lceil \frac{8n}{11} \rceil < n$, a proto

$$T(n) \leq c\frac{21n}{100} + c\frac{8n}{11} + bn = \left(\frac{1031c}{1100} + b\right) n \leq cn.$$

Tedy

**Věta.** *Algoritmus* **SELECT** *nalezne i-tý nejmenší prvek v lineárním čase.*

Algoritmus **FIND** je ve velké většině případů rychlejší než algoritmus **SELECT**, proto se v praxi doporučuje používat **FIND**, i když existují případy (velmi řídké), kdy potřebuje

kvadratický čas. Je známo, že lze nalézt medián $n$-prvkové množiny s méně než $3n$ porovnáními, a že každý algoritmus hledající medián a používající porovnání jako jedinou primitivní operaci mezi prvky množiny vyžaduje více než $2n$ porovnání.

Historické resume: Algoritmus **HEAPSORT** navrhl v roce 1964 Williams a vylepšil Floyd (rovněž 1964). Návrh na použití $d$-regulárních hald je folklor stejně tak jako algoritmus **MERGESORT**. Algoritmy **QUICKSORT** a **FIND** zavedl Hoare (1962). Analýza operace **MERGE** a hledání optimálního stromu pochází od Huffmana (1952) a lineární implementaci algoritmu navrhl van Leeuwen (1976). Analýza rozhodovacích stromů je folklor. Algoritmus **HYBRIDSORT** navrhli Meijer a Akl (1980), vylepšená verze **BUCKETSORT**u (nazvaná **WORDSORT**) pochází od Aho, Hopcrofta a Ullmana (1974), algoritmus **SELECT** byl navržen Blumem, Floydem, Prattem, Rivestem a Tarjanem (1972).