# Datové struktury

## Obsah

# 1 Úvod

*Základní* problém: Reprezentace množin a operace s nimi. V řadě úloh a algoritmů je tento podproblém rozhodující pro složitost řešení, protože tyto operace se mnohokrát opakují. Proto je třeba navrhnout

pro tyto úlohy co nejefektivnější algoritmy (každý ušetřený čas mnohonásobným opakováním začne hrát důležitou roli). To vede k detailní analýze složitosti v závislosti na vnějších okolnostech. Nelze říct, že některý algoritmus je nejlepší, protože za určitých okolností může být 'méně efektivní' algoritmus výhodnější.

Cílem této přednášky není pouze seznámit vás s algoritmy pro řešení těchto problémů, protože s těmi jste se seznámili už v přednášce 'Algoritmy a datové struktury'. Hlavním cílem je ukázat vám prostředky a metody, jak měřit a zjišťovat jejich efektivitu, a tím vám ukázat prostředky, které vám umožní rozhodnout se v dané situaci pro určitý algoritmus. Proto hlavní náplní této přednášky bude počítání efektivity algoritmů. Budeme počítat za zjednodušených předpokladů, protože neumím říct (a ani to nelze, protože vždy záleží na konkrétních okolnostech), které sofistikovanější metody budou v praxi vhodné pro řešení vašeho problému. Cílem přednášky je seznámit vás s možnostmi, jak řešit tyto problémy, a se základními metodami pro jejich řešení.

Skripta byla napsána profesorem Koubkem; pan profesor je dal k dispozici Karlovi Bílkovi a Markovi Vašutovi, kteří z nich „svévolně" udělali tuto upravenou verzi v LaTeXu. Některé části tedy nemusí být správné. Delší důkazy, ve kterých mi v původních skriptech chyběla struktura, jsem přepsal na sérii drobnějších lemmat, doufám, že to bude jasnější.

# 2 Hašování

## 2.1 Slovníkový problém

Nejprve si zadefinujeme asi nejzákladnější problém, který řešíme v datových strukturách.

Řešíme tzv. slovníkový problém: Dáno univerzum $U$, máme reprezentovat $S \subseteq U$ a navrhnout algoritmy pro následující operace
**MEMBER**$(x)$ – zjistí, zda $x \in S$, a nalezne jeho uložení
**INSERT**$(x)$ – když $x \notin S$, pak vloží $x$ do struktury reprezentující $S$
**DELETE**$(x)$ – když $x \in S$, pak odstraní $x$ ze struktury reprezentující $S$.
Efektivita algoritmu: časová složitost, prostorová složitost;
vyšetřené buď v nejhorším případě nebo v průměrném případě nebo amortizovaně.

Literatura:
K. Mehlhorn: Data Structures and Algorithms 1: Sorting and Searching, Springer 1984
`http://www.mpi-sb.mpg.de/~mehlhorn/DatAlgbooks.html`

J. S. Vitter, W.-Ch. Chen: Design and Analysis of Coalesced Hashing, Oxford Univ. Press, 1987

## 2.2 Hašování obecně

Pomocí bitového pole můžeme rychle implementovat operace **MEMBER**, **INSERT** a **DELETE**.

Nevýhoda: když je velké univerzum, pak je prostorová složitost v nejlepším případě ohromná, ve špatném případě nelze pole zadat do počítače.
Hašování chce zachovat rychlost operací, ale odstranit paměťovou náročnost. První publikovaný článek o hašování je od Dumney z roku 1956, první analýza hašování pochází od Petersona z roku 1957, ale existuje technická zpráva od IBM o hašování z roku 1953.

Základní idea: Dáno univerzum $U$ a množina $S \subseteq U$ tak, že $|S| << |U|$. Máme funkci $h : U \to \{0, 1, \ldots, m-1\}$ (taky *hešovací funkce*) a množinu $S$ reprezentujeme tabulkou (polem) s $m$ řádky tak, že $s \in S$ je uložen na řádku $h(s)$.

$m$ (jako *memory*) si tedy budeme značit velikost tabulky; $n$ je velikost $|S|$.

Nevýhoda: mohou existovat různá $s, t \in S$ taková, že $h(s) = h(t)$ - tento jev se nazývá *kolize*.

Hlavní problém, kterému se věnuje zbytek kapitoly: řešení kolizí.

## 2.3 Hašování se separovanými řetězci

Základní řešení: použijeme pole o velikosti $[0..m-1]$ a $i$-tá položka pole bude spojový seznam obsahující všechny prvky $s \in S$ takové, že $h(s) = i$. Toto řešení se nazývá *hašování se separovanými řetězci*.

Příklad: $U = \{1, 2, \ldots, 1000\}$, $S = \{1, 7, 11, 53, 73, 141, 161\}$ a funkce je $h(x) = x \bmod 10$. Pak

$$P(0) = P(2) = P(4) = P(5) = P(6) = P(8) = P(9) = \emptyset,$$
$$P(7) =< 7 >, \quad P(3) =< 53, 73 >, \quad P(1) =< 1, 141, 11, 161 > .$$

Seznamy nemusí být uspořádané.

### 2.3.1 Algoritmy operací

**MEMBER**$(x)$
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t \neq x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

**INSERT**$(x)$
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t \neq x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t \neq x$ **then** vložíme $x$ do $i$-tého seznamu **endif**

**DELETE**$(x)$
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t \neq x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t = x$ **then** odstraníme $x$ z $i$-tého seznamu **endif**

### 2.3.2 Nejhorší případy

V následující analýze předpokládáme, že hodnota funkce $h(x)$ je spočitatelná v čase $O(1)$.

V nejhorším případě operace vyžadují čas $O(|S|)$ (všechny prvky jsou v jednom seznamu).
Požadovaná paměťová náročnost $O(m+|S|)$ (předpokládáme, že reprezentace prvku $s \in S$ vyžaduje paměť $O(1)$)
Paměť není efektivně využitá.

Další kapitoly 2.3 jsou věnovány **očekávaným případům**

### 2.3.3 Očekávané případy - předpoklady

Pro výpočet očekávaných případů si zavedeme předpoklady:

1. $h$ je rychle spočitatelná (tj. $O(1)$) a neměnná během výpočtu;
2. všechny $h^{-1}(i)$ jsou stejně velké, tj. $h$ rozděluje univerzum $U$ rovnoměrně (tj. $-1 \leq |h^{-1}(i)| - |h^{-1}(j)| \leq 1$ pro $i, j \in \{0, 1, \ldots, m-1\}$ – rozdíl 1 kvůli tomu, že jde o celá čísla);
3. $S$ je náhodně vybraná z univerza $U$ (tj. pro dané $n = |S|$ jsou všechny podmnožiny $U$ o velikosti $n$ reprezentovanou množinou $S$ se stejnou pravděpodobností);
4. každý prvek z $U$ má stejnou pravděpodobnost být argumentem operace **INSERT**, **DELETE**, **MEMBER**
5. velikost reprezentované množiny je výrazně menší než velikost univerza.

Použité značení: $|S| = n$ (*number*), $m$ =počet řetězců (*memory*), $|U| = N$,
$\ell(i)$ =délka $i$-tého řetězce, $\alpha = \frac{n}{m}$ faktor naplnění (load factor)

### 2.3.4 Jednoduché důsledky předpokladů

1. $\text{Prob}(h(x) = i) = \frac{1}{m}$ pro všechna $x \in U$ a všechna $i = 0, 1, \ldots, m-1$ – tj. prvky jsou rovnoměrně rozloženy do „slotů“
2. $\text{Prob}(\ell(i) = l) = p_{n,l} = \binom{n}{l}(\frac{1}{m})^l(1 - \frac{1}{m})^{n-l}$ pro všechna $i = 0, 1, \ldots, m-1$ – tj. délky řetězců mají binomiální rozdělení.

Vysvětlení: $i$-tý řetězec má délku $l$, právě když je $l$ prvků z $S$ zahešováno do $i$ a zbytek ne – tj. existuje podmnožina $A \subseteq S$ taková, že $|A| = l$ (těchto možností je $\binom{n}{l}$), pro každé $x \in A$ platí $h(x) = i$ (pravděpodobnost tohoto jevu je $(\frac{1}{m})^l$) a pro každé $x \in S \setminus A$ platí $h(x) \neq i$ (pravděpodobnost tohoto jevu je $(1 - \frac{1}{m})^{n-l}$). To znamená, že jev má binomiální rozdělení.

Zde jsme nepřesní kvůli možnému rozdílu 1 ve velikostech množin. Obecně pro náhodně zvolené $x \in U$ a dané $i$ je $\text{Prob}(h(x) = i) = \frac{|h^{-1}(i)|}{|U|}$ a když existují dvě různá $i, j \in \{0, 1, \ldots, m-1\}$ taková, že $|h^{-1}(i)| \neq |h^{-1}(j)|$, pak obecně $\frac{|h^{-1}(i)|}{|U|} \neq \frac{1}{m}$. Toto nastane i v případě, když jsme již zvolili nějaký prvek v $h^{-1}(i)$. Protože však předpokládáme, že $n, m << |U|$ tak ve všech uvažovaných případech je $Prob(h(x) = i)$ přibližně $\frac{1}{m}$, a můžeme tuto pravděpodobnost aproximovat hodnotou $\frac{1}{m}$.

### 2.3.5 Očekávaná délka řetězců

**Věta.** *Očekávaná délka řetězců je* $\frac{n}{m}$.

*Důkaz.*

$$\begin{aligned}
E(l) =& \sum_{l=0}^{n} l p_{n,l} = \sum_{l=0}^{n} l \binom{n}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \\
& \sum_{l=0}^{n} l \frac{n!}{l!(n-l)!} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \\
& \frac{n}{m} \sum_{l=1}^{n} \frac{(n-1)!}{(l-1)!(n-l)!} (\frac{1}{m})^{l-1} (1-\frac{1}{m})^{n-l} = \\
& \frac{n}{m} \sum_{l=1}^{n} \binom{n-1}{l-1} (\frac{1}{m})^{l-1} (1-\frac{1}{m})^{(n-1)-(l-1)} = \\
& \frac{n}{m} \sum_{l=0}^{n-1} \binom{n-1}{l} (\frac{1}{m})^{l} (1-\frac{1}{m})^{n-1-l} = \\
& \frac{n}{m} (\frac{1}{m} + 1 - \frac{1}{m})^{n-1} = \frac{n}{m}.
\end{aligned}$$

Toto je standardní elementární výpočet očekávané hodnoty binomiálního rozdělení. □

---

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

*Důkaz.*

$$\begin{aligned}
E(l^2) =& E(l(l-1)) + E(l), \\
E(l(l-1)) =& \sum_{l=0}^{n} l(l-1) \binom{n}{l} (\frac{1}{m})^l (1-\frac{1}{m})^{n-l} = \\
& \frac{n(n-1)}{m^2} \sum_{l=2}^{n} \binom{n-2}{l-2} (\frac{1}{m})^{l-2} (1-\frac{1}{m})^{(n-2)-(l-2)} = \\
& \frac{n(n-1)}{m^2} \sum_{l=0}^{n-2} \binom{n-2}{l} (\frac{1}{m})^{l} (1-\frac{1}{m})^{n-2-l} = \\
& \frac{n(n-1)}{m^2}, \\
E(l^2) =& \frac{n(n-1)}{m^2} + \frac{n}{m} = \frac{n}{m}(1 + \frac{n-1}{m}).
\end{aligned}$$

□

---

**Věta.** *Rozptyl řetězců je* $nm(1-\frac{1}{m})$.

*Důkaz.*

$$\begin{aligned}\mathrm{var}(l) =& E(l - E(l))^2 = E(l^2) - (E(l))^2 = \\ & \frac{n}{m}(1 + \frac{n-1}{m}) - (\frac{n}{m})^2 = \frac{n}{m}(1 - \frac{1}{m}).\end{aligned}$$

□

---

Shrneme výsledky:

**Očekávaná délka řetězců je $\frac{n}{m}$ a rozptyl délky řetězců je $\frac{n}{m}(1 - \frac{1}{m})$.**

Toto jsou standardní elementární odvození druhého momentu a rozptylu binomiálního rozdělení.

### 2.3.6 Očekávaný nejdelší řetězec

Spočítáme $E(NP)$ očekávanou délku maximálního řetězce ($NP$ jako nejhorší případ :))

**Lemma.** *$E(NP) = \sum_j \mathrm{Prob}(\max_i \ell(i) \geq j)$, kde $\ell(i)$ je délka $i$-tého řetězce.*

*Důkaz.* Platí, že

$$\mathrm{Prob}(\max_i \ell(i) = j) = \mathrm{Prob}(\max_i \ell(i) \geq j) - \mathrm{Prob}(\max_i \ell(i) \geq j + 1).$$

Pak můžeme počítat:

$$\begin{aligned}E(NP) =& \sum_j j \,\mathrm{Prob}(\max_i \ell(i) = j) = \\ & \sum_j j(\mathrm{Prob}(\max_i \ell(i) \geq j) - \mathrm{Prob}(\max_i \ell(i) \geq j + 1)) = \\ & \sum_j j \,\mathrm{Prob}(\max_i \ell(i) \geq j) - \sum_j j \,\mathrm{Prob}(\max_i \ell(i) \geq j + 1) = \\ & \sum_j j \,\mathrm{Prob}(\max_i \ell(i) \geq j) - \sum_j (j-1) \,\mathrm{Prob}(\max_i \ell(i) \geq j) = \\ & \sum_j (j - j + 1) \,\mathrm{Prob}(\max_i \ell(i) \geq j) = \\ & \sum_j \mathrm{Prob}(\max_i \ell(i) \geq j).\end{aligned}$$

Vysvětlení: Při čtvrté rovnosti se v druhé sumě zvětšil index, přes který sčítáme, o 1, v páté rovnosti se k sobě daly koeficienty při stejných pravděpodobnostech ve dvou sumách.

□

---

**Lemma.** $\text{Prob}(\max_i(\ell(i)) \geq j) \leq \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\}$.

*Důkaz.*

$$\begin{aligned}\text{Prob}(\max_i(\ell(i)) \geq j) = \\ &\text{Prob}(\ell(1) \geq j \vee \ell(2) \geq j \vee \cdots \vee \ell(m-1) \geq j) \leq \\ &\sum_i \text{Prob}(\ell(i) \geq j) \leq m\binom{n}{j}(\frac{1}{m})^j = \\ &\frac{\prod_{k=0}^{j-1}(n-k)}{j!}(\frac{1}{m})^{j-1} \leq n(\frac{n}{m})^{j-1}\frac{1}{j!}.\end{aligned}$$

Vysvětlení: První nerovnost plyne z toho, že pravděpodobnost disjunkce jevů je menší než součet pravděpodobností jevů, druhá nerovnost plyne z toho, že $i$-tý řetězec má délku alespoň $j$, jakmile existuje podmnožina $A \subseteq S$ taková, že $|A| = j$ (těchto možností je $\binom{n}{j}$) a pro každé $x \in A$ platí $h(x) = i$ (pravděpodobnost tohoto jevu je $(\frac{1}{m})^j$). Protože pravděpodobnost je pro všechna $i$ stejná a $i$ nabývá $m$ hodnot, dostáváme druhou nerovnost. Následující rovnost plyne z rozepsání binomických koeficientů. Poslední nerovnost dostaneme nahrazením $n-k$ hodnotou $n$.

Že pravděpodobnost je menší než 1 je triviální.

□

---

**Lemma** (Stirlingův vzorec pro faktoriály (bez důkazu))**.** $j! = \sqrt{2j\pi}\left(\frac{j}{e}\right)^j\left(1 + \frac{1}{12j} + \frac{1}{288j^2} + O(j^{-3})\right)$

**Lemma** (Pomocné lemma)**.** *Když* $(\frac{q}{e})^q \leq n$*, pak* $q \leq (1+o(1))\frac{\ln n}{\ln\ln n}$.

*Důkaz.* Všimněme si nejdřív, že platí

$$\begin{aligned}\left((\ln(\frac{\ln n}{\ln\ln n}) - 1\right)\frac{\ln n}{\ln\ln n} =&(\ln\ln n - \ln\ln\ln n - 1)\frac{\ln n}{\ln\ln n} = \\ &\ln n - \frac{\ln n(\ln\ln\ln n)}{\ln\ln n} - \frac{\ln n}{\ln\ln n} = \\ &\ln n(1 - \frac{\ln\ln\ln n}{\ln\ln n} - \frac{1}{\ln\ln n}) = (1+o(1))\ln n,\end{aligned}$$

protože $\lim_{n\mapsto\infty}\frac{\ln\ln\ln n}{\ln\ln n} = 0 = \lim_{n\mapsto\infty}\frac{1}{\ln\ln n}$. Odtud plyne

$$\lim_{n\mapsto\infty}\left(\frac{\frac{\ln n}{\ln\ln n}}{e}\right) = e^{(1+o(1))\ln n} = n.$$

Protože $(\frac{q}{e})^q$ je rostoucí funkce, tak dostáváme, že $q \leq (1+o(1))(\frac{\ln n}{\ln\ln n})$. □

---

Označme si $j_0 = \min\{j \mid n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1\}$

**Lemma** (Omezení $j_0$)**.** *Pokud* $\alpha = \frac{n}{m} \leq 1$*, pak* $j_0 \leq \frac{(1+o(1))\log n}{\log\log n}$

*Důkaz.*

$$j_0 = \min\{j \mid n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1\} \leq \min\{j \mid n \leq j!\} \leq \min\{j \mid n \leq (\frac{j}{e})^j\} \leq (1+o(1))\frac{\ln n}{\ln\ln n}.$$

První nerovnost je jen převedení na druhou stranu a ignorování zlomku menšího než 1, druhá nerovnost plyne ze Stirlingova vzorce, třetí plyne z pomocného lemmatu. □

---

**Věta.** *Pokud $\alpha = \frac{n}{m} \leq 1$, horní odhad odhad očekávané délky maximálního řetězce je $O(\frac{\log n}{\log\log n})$*

*Důkaz.*

$$\begin{aligned}
E(NP) = &\sum_j \text{Prob}(\max_i(\ell(i)) \geq j) \leq \\
&\sum_j \min\{1, n(\frac{n}{m})^{j-1}\frac{1}{j!}\} = \\
&\sum_{j=1}^{j_0} 1 + \sum_{j=j_0+1}^{\infty} \left(n(\frac{n}{m})^{j-1}\frac{1}{j!}\right) \leq j_0 + \sum_{j=j_0+1}^{\infty}\frac{n}{j!} = \\
&j_0 + \frac{n}{j_0!}\sum_{j=j_0+1}^{\infty}\frac{j_0!}{j!} \leq j_0 + \sum_{j=j_0+1}(\frac{1}{j_0+1})^{j-j_0} = \\
&j_0 + \frac{\frac{1}{j_0+1}}{-\frac{1}{j_0+1}+1} = j_0 + \frac{1}{j_0} = O(j_0).
\end{aligned}$$

Vysvětlení: Při druhé rovnosti jsme použili, že $n(\frac{n}{m})^{j-1}\frac{1}{j!} \leq 1$, právě když $j \leq j_0$. Při druhé nerovnosti jsme použili, že $\frac{n}{m} \leq 1$, při třetí nerovnosti jsme použili, že $\frac{n}{j_0!} \leq 1$ a

$$\frac{j_0!}{j!} = \frac{1}{\prod_{k=j_0+1}^{j} k} \leq (\frac{1}{j_0+1})^{j-j_0}.$$

□

---

**Věta** (bez důkazu)**.** *Když $0.5 \leq \alpha \leq 1$, je $O(\frac{\log n}{\log\log n})$ zároveň i dolní odhad $E(NP)$.*

Shrneme získaný výsledek

Za předpokladu $\alpha = \frac{n}{m} \leq 1$ je při hašování se separovanými řetězci horní odhad očekávané délky maximálního řetězce $O(\frac{\log n}{\log\log n})$.
Když $0.5 \leq \alpha \leq 1$, je to zároveň i dolní odhad.

### 2.3.7 Očekávaný počet testů

Test je porovnání argumentu operace s prvkem na daném místě řetězce nebo zjištění, že vyšetřovaný řetězec je prázdný.

Budeme rozlišovat dva případy:
*úspěšné vyhledávání* – argument operace je mezi prvky reprezentované množiny,
*neúspěšné vyhledávání* – argument operace není mezi prvky reprezentované množiny.

**Věta** (Neúspěšné vyhledávání)**.** *Při hašování se separovanými řetězci je očekávaný počet testů při neúspešném vyhledávaní přibližně* $e^{-\alpha} + \alpha$

*Důkaz.* Očekávaný počet testů:

$$E(T) = \text{Prob}(\ell(i) = 0) + \sum_l l \,\text{Prob}(\ell(i) = l) =$$
$$p_{n,0} + \sum_l l p_{n,l} =$$
$$(1 - \frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha.$$

Vysvětlení: Zjištění, zda řetězec je prázdný, vyžaduje jeden test, tj. $\text{Prob}(\ell(i) = 0)$ není s koeficientem 0, ale 1. Protože pravděpodobnosti jsou stejné pro všechny řetězce, nemusíme specifikovat řetězec, který vyšetřujeme, stačí psát obecně $i$. $\sum_l l p_{n,l}$ jsme spočítali při výpočtu očekávané délky řetězce. □

---

**Věta** (Úspěšné vyhledávání)**.** *Při hašování se separovanými řetězci je očekávaný počet testů při úspešném vyhledávaní přibližně* $1 + \frac{\alpha}{2}$

*Důkaz.* Zvolme jeden řetězec prvků o délce $l$. Počet testů při vyhledání všech prvků v tomto řetězci je

$$1 + 2 + \cdots + l = \binom{l+1}{2}.$$

Očekávaný počet testů při vyhledání všech prvků v nějakém řetězci je

$$\sum_l \binom{l+1}{2} \text{Prob}(\ell(i) = l) = \sum_l \binom{l+1}{2} p_{n,l}.$$

Očekáný počet testů při vyhledání všech prvků v tabulce je $m \sum_l \binom{l+1}{2} p_{n,l}$.

Očekávaný počet testů pro vyhledání jednoho prvku je

$$\frac{m}{n} \sum_{l=0}^{n} \binom{l+1}{2} p_{n,l} = \frac{m}{2n} \Big( \sum_{l=0}^{n} l^2 p_{n,l} + \sum_{l=0}^{n} l p_{n,l} \Big) =$$
$$\frac{m}{2n} \Big( \sum_{l=1}^{n} l(l-1) p_{n,l} + 2 \sum_{l=1}^{n} l p_{n,l} \Big) =$$
$$\frac{m}{2n} \Big( \frac{n(n-1)}{m^2} + \frac{2n}{m} \Big) = \frac{n-1}{2m} + 1 \approx$$
$$1 + \frac{\alpha}{2}.$$

□

Jiný postup.

*Důkaz.* Předpokládejme, že počet testů při úspěšném vyhledávání prvku $x \in S$ je 1+počet porovnání klíčů při neúspěšném vyhledávání $x$ v operaci **INSERT**$(x)$. Pak počet porovnání klíčů je délka řetězce, a proto očekávaný počet porovnání klíčů je očekávaná délka řetězce. Tedy očekávaný počet testů při úspěšném vyhledávání $x$ je 1+očekávaná délka řetězce v okamžiku vkládání $x$, neboli

$$\frac{1}{n}\sum_{i=0}^{n-1}(1+\frac{i}{m}) = 1 + \frac{n-1}{2m}.$$

□

Při hašování se separovanými řetězci je očekávaný počet testů při neúspešném vyhledávaní přibližně $e^{-\alpha} + \alpha$ a při úspěšném vyhledávání přibližně $1 + \frac{\alpha}{2}$.

Následující tabulka dává přehled očekávaného počtu testů pro různé hodnoty $\alpha$

| $\alpha$ | 0 | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 |
|---|---|---|---|---|---|---|---|
| neúsp. vyh. | 1 | 1.005 | 1.019 | 1.041 | 1.07 | 1.107 | 1.149 |
| úspěš. vyh. | 1 | 1.05 | 1.1 | 1.15 | 1.2 | 1.25 | 1.3 |

| $\alpha$ | 0.7 | 0.8 | 0.9 | 1 | 2 | 3 |
|---|---|---|---|---|---|---|
| neúsp. vyh. | 1.196 | 1.249 | 1.307 | 1.368 | 2.135 | 3.05 |
| úspěš. vyh. | 1.35 | 1.4 | 1.45 | 1.5 | 2 | 2.5 |

Všimněte si, že očekávaný počet testů při neúspěšném vyhledávání je menší než očekávaný počet testů při úspěšném vyhledávání, když $\alpha \leq 1$. Na první pohled vypadá tento výsledek nesmyslně, ale důvod je, že počet testů při úspěšném vyhledávání průměrujeme proti $n$, kdežto při neúspěšném vyhledávání proti $m$.

Ilustrujeme to na následujícím příkladu:

Nechť $n = \frac{m}{2}$ a nechť polovina neprázdných řetězců má délku 1 a polovina má délku 2.
Očekávaný počet testů při neúspěšném vyhledávaní:

- 1 test pro prázdné řetězce a řetězce délky 1 – těchto případů je $\frac{5m}{6}$
- 2 testy pro řetězce délky 2 – těchto případů je $\frac{m}{6}$.

Očekávaný počet testů je $\frac{1}{m}(1\frac{5m}{6} + 2\frac{m}{6}) = \frac{7}{6}$.
Očekávaný počet testů při úspěšném vyhledávání:

- 1 test pro prvky na prvním místě řetězce – těchto případů je $\frac{2n}{3}$
- 2 testy pro prvky, které jsou na druhém místě řetězce – těchto případů je $\frac{n}{3}$.

Očekávaný počet testů je $\frac{1}{n}(1\frac{2n}{3} + 2\frac{n}{3}) = \frac{4}{3}$.

Velikost $\alpha$ je doporučována menší než 1, ale nemá být hodně malá, protože by paměť nebyla efektivně využita.

## 2.4 Hašování s uspořádanými separovanými řetězci

Vylepšení metody: hašování s uspořádanými řetězci. Rozdíl proti původní metodě – řetězce jsou uspořádané ve vzrůstajícím pořadí. Protože řetězce obsahují tytéž prvky, je počet očekávaných testů při úspěšném vyhledávání stejný jako u neuspořádaných řetězců. Při neúspěšném vyhledávání končíme, když argument operace je menší než vyšetřovaný prvek v řetězci, tedy končíme dřív.

### 2.4.1 Očekávaný počet testů

Očekávaný počet testů při neúspěšném vyhledávání pro hašování s uspořádanými řetězci je přibližně $e^{-\alpha} + 1 + \frac{\alpha}{2} - \frac{1}{\alpha}(1 - e^{-\alpha})$.
Očekávaný počet testů při úspěšném vyhledávání pro hašování s uspořádanými řetězci je přibližně $1 + \frac{\alpha}{2}$.

### 2.4.2 Algoritmy

**MEMBER**$(x)$
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t < x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t = x$ **then** $x \in S$ **else** $x \notin S$ **endif**

**INSERT**$(x)$
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t < x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t \neq x$ **then**
  **if** $x < t$ **then**
    vložíme $x$ do $i$-tého seznamu před prvek $t$
  **else**
    vložíme $x$ do $i$-tého seznamu za prvek $t$
  **endif**
**endif**

**DELETE**($x$)
Spočítáme $i := h(x)$, $t := NIL$
**if** $i$-tý seznam je neprázdný **then**
  $t$ :=první prvek $i$-tého seznamu
  **while** $t < x$ a $t \neq$poslední prvek $i$-tého seznamu **do**
    $t$ :=následující prvek $i$-tého seznamu
  **enddo**
**endif**
**if** $t = x$ **then** odstraníme $x$ z $i$-tého seznamu **endif**

## 2.5 Motivace pro neseparované řetězce

Nevýhody hašovaní se separovanými řetězci –
  nevyužití alokované paměti (nehospodárné)
  používání ukazatelů (cache).

Řešení: využít pro řetězce původní tabulku. Pak řádky tabulky musí mít strukturu, která umožňuje prohledávat řetězce, a velikost reprezentované množiny může být nejvýše rovna velikosti tabulky.

Položky tabulky:

- key,
- odkaz na uložená data,
- položky pro práci s tabulkou.

Předpokládáme, že data jsou velká, v tom případě se ukládají mimo tabulku. V tabulce je jen odkaz na uložená data. Při popisu práce s tabulkou tuto část budeme vynechávat (tj. data budou pouze klíč).

Podle řešení kolize dělíme dál hašování:

- hašování s přemísťováním
- hašování s dvěma ukazateli,
- srůstající hašování,
- dvojité hašování,
- hašování s lineárním přidáváním.

## 2.6 Hašování s přemísťováním

### 2.6.1 Nepřesný popis

*(pozn.: sekce „nepřesný popis“ jsou opravdu nepřesné)*

V tabulce na řádku $i$ buď nic není (= do $i$ se nic nezahešovalo), nebo tam začíná řetězec od něčeho, co koliduje v $h(i)$, nebo je tam prostředek nějakého jiného řetězce (taky jenom tehdy, pokud se do $i$ nic nezahešovalo).

Pokud tam při vkládání nic neni, dám to tam.

Pokud tam je prostředek jiného řetězce, přemístím ten prostředek (od toho název metody) do jiného volného místa a vkládaný prvek dám do nově uvolněného místa.

Pokud tam je začátek správného řetězce, dojedu na konec a dám to za něj.

### 2.6.2 Ilustrace

Položky pro práci s tabulkou: next, previous
položka next – číslo řádku tabulky obsahující následující položku seznamu
položka previous – číslo řádku tabulky obsahující předcházející položku seznamu.

Protože velikost tabulky omezuje velikost reprezentované množiny, může nastat přeplnění. O řešení případného přeplnění pojednáme později na str. 29. Stejný způsob řešení přeplnění se používá i v dalších metodách, kde velikost tabulky omezuje velikost reprezentované množiny.

Příklad: $U = \{1, 2, \ldots, 1000\}$, $h(x) = x \bmod 10$,
uložená množina $S = \{1, 7, 11, 53, 73, 141, 161\}$,
řetězce: $P(1) = (1, 141, 11, 161)$, $P(3) = (73, 53)$, $P(7) = (7)$.
Hašovací tabulka:

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

Tabulka vznikla následující posloupností operací:
**INSERT**(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53),
**INSERT**(7), **INSERT**(161).

### 2.6.3 Algoritmy

**MEMBER**($x$)
Spočítáme $i := h(x)$
**if** $i.previous \neq$prázdné nebo $i.key =$prázdné **then Výstup**: $x \notin S$, stop **endif**
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

**DELETE**($x$)
Spočítáme $i := h(x)$
**if** $i.previous \neq$prázdné nebo $i.key =$prázdné **then** stop **endif**
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key = x$ **then**
  **if** $i.previous \neq$prázdné **then**
    $(i.previous).next := i.next$
    **if** $i.next \neq$prázdné **then** $(i.next).previous := i.previous$ **endif**
    $i.key := i.next := i.previous :=$ prázdné
  **else**
    **if** $i.next \neq$prázdné **then**
      $i.key := (i.next).key$, $i.next := (i.next).next$
      **if** $((i.next).next) \neq$prázdné **then** $((i.next).next).previous := i$ **endif**
      $(i.next).key := (i.next).next := (i.next).previous :=$ prázdné
    **else**
      $i.key :=$ prázdné
    **endif**
  **endif**
**endif**

**INSERT**($x$)
Spočítáme $i := h(x)$
**if** $i.key = NIL$ **then** $i.key := x$, stop **endif**
**if** $i.previous \neq NIL$**then**
  **if** neexistuje prázdný řádek tabulky **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je volný řádek tabulky
    $j.key := i.key$, $j.previous := i.previous$, $j.next := i.next$, $(j.previous).next :=$
$j$
    **if** $j.next \neq NIL$ **then** $(j.next).previous := j$ **endif**
      $i, key := x$, $i.next := i.previous :=$prázdné
    **endif**
  **endif**
**else**
  **while** $i.next \neq NIL$ a $i.key \neq x$ **do** $i := i.next$ **enddo**
  **if** $i.key \neq x$ **then**
    **if** neexistuje prázdný řádek tabulky **then**
      **Výstup**: přeplnění
    **else**
      nechť $j$ je volný řádek tabulky
      $i.next := j$, $j.key := x$, $j.previous := i$
    **endif**
  **endif**
**endif**

V příkladu provedeme **INSERT**(28), nový řádek je 4. řádek

– výsledná hašovací tabulka

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

### 2.6.4 Očekávaný počet testů

Očekávaný počet testů je stejný jako pro hašování se separovanými řetězci:
úspěšné vyhledávání: $\frac{n-1}{2m} + 1 \approx 1 + \frac{\alpha}{2}$
neúspěšné vyhledávání: $(1 - \frac{1}{m})^n + \frac{n}{m} \approx e^{-\alpha} + \alpha$,
kde $m$ = velikost tabulky, $n$ = velikost $S$, tj. počet uložených prvků, $\alpha = \frac{n}{m}$ = faktor zaplnění.

### 2.6.5 Diskuse

Nevýhodou hašování s přemísťováním je v operaci **INSERT** případ, že *previous* $h(i)$-tého řádku je neprázdný. Pak přemísťujeme položku na $h(i)$-tém řádku na volný řádek a to vyžaduje více času – operace s přemístěním položky. Toto odstraňuje další implementace hašování se separujícími řetězci.

## 2.7 Hašování s dvěma ukazateli

### 2.7.1 Nepřesný popis

V řádku $i$ je úplně jedno, co je v položce *key*, resp. nijak to nesouvisí s výsledkem hešovací funkce – co souvisí je položka *begin*, co říká, kde řetězec k danému číslu začíná.

Nemusíme nic přemísťovat, stačí ukazovat na správný začátek.

### 2.7.2 Ilustrace

Položky pro práci s tabulkou – next, begin
Položka next – číslo řádku tabulky obsahující následující položku seznamu
Položka begin – číslo řádku tabulky obsahující první položku seznamu s touto adresou

Stejná data jako v minulém případě

Hašovací tabulka:

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

Tabulka vznikla následující posloupností operací:
**INSERT**(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53), **INSERT**(7), **INSERT**(161).

### 2.7.3 Algoritmy

**MEMBER**($x$)
Spočítáme $i := h(x)$
**if** $i.begin$ =prázdné **then Výstup**: $x \notin S$, stop **else** $i := i.begin$ **endif**
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

**DELETE**($x$)
Spočítáme $i := h(x)$
**if** $i.begin$ =prázdné **then** stop **else** $j := i$, $i := i.begin$ **endif**
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $j := i$, $i := i.next$ **enddo**
**if** $i.key = x$ **then**
  **if** $i = j.begin$ **then**
    **if** $i.next \neq$prázdné **then**
      $j.begin := i.next$
    **else**
      $j.begin$ :=prázdné
    **endif**
  **else**
    $j.next := i.next$
  **endif**
  $i.key := i.next$ :=prázdné
**endif**

**INSERT**($x$)
Spočítáme $i := h(x)$
**if** $i.begin$ =prázdné **then**
  **if** $i.key$ =prázdné **then**
    $i.key := x$, $i.begin := i$
  **else**
    **if** neexistuje prázdný řádek tabulky **then**
      **Výstup**: přeplnění
    **else**
      nechť $j$ je volný řádek tabulky
      $j.key = x$, $i.begin := j$
    **endif**
  **endif**
**else**
  $i := i.begin$
  **while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
  **if** $i.key \neq x$ **then**
    **if** neexistuje prázdný řádek tabulky **then**
      **Výstup**: přeplnění
    **else**
      nechť $j$ je volný řádek tabulky
      $i.next := j$, $j.key := x$
    **endif**
  **endif**
**endif**

V příkladu provedeme **INSERT**(28), nový řádek je 4. řádek
– výsledná hašovací tabulka

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

### 2.7.4 Očekávaný počet testů

Algoritmus při práci s položkami je rychlejší než při hašování s přemísťováním, ale začátek řetězce v jiném místě tabulky přidává jeden test.

Očekávaný počet testů:
úspěšný případ: $1 + \frac{(n-1)(n-2)}{6m^2} + \frac{n-1}{2m} \approx 1 + \frac{\alpha^2}{6} + \frac{\alpha}{2}$
neúspěšný případ: $\approx 1 + \frac{\alpha^2}{2} + \alpha + e^{-\alpha}(2+\alpha) - 2.$

## 2.8 Srůstající hašování - přehled

### 2.8.1 Nepřesný popis

Máme řetězce jako v minulých dvou algoritmech, ale pokud se trefím doprostřed řetězce, tak ho nepřemísťuji, ale srostu s ním.

Protože nic nepřemísťujeme, není potřeba *previous*.

Typy:

- Standardní – normální tabulka.
  - LISCH - vkládám na konec řetězce
  - EISCH - vkládám hned za prvek
- „Nestandardní“ – bez písmene navíc :) – tabulka rozšířena o pomocnou tabulku, kam ukládám jako první
  - LICH - vkládám na konec řetězce
  - EICH - vkládám hned za prvek
  - VICH - komplikovanost, viz dál

### 2.8.2 Přehled

Srůstající hašování se dělí podle práce s pamětí na standardní a na srůstající hašování s pomocnou pamětí (které se nazývá jen srůstající hašování) a podle způsobu přidávání dalšího prvku.

Popíšeme metody:
Standardní srůstající hašování: LISCH, EISCH,
Srůstající hašování: LICH, VICH, EICH.

Všechny metody pro práci s tabulkou používají jen položku next – číslo řádku tabulky obsahující následující položku seznamu.

Základní idea: řetězec začíná na svém místě, ale pokud už tam byl uložen nějaký údaj, pak řetězec tohoto údaje sroste s řetězcem začínajícím na tomto řádku. To znamená, že prvky řetězce, který začíná na tomto místě, budou uloženy v řetězci, který už je uložen na tomto místě, ale jen od tohoto místa dál.

## 2.9 Metody EISCH a LISCH

### 2.9.1 Popis

- EISCH – early-insertion standard coalesced hashing
- LISCH – late-insertion standard coalesced hashing.

Organizace tabulky je stejná jako v předchozích případech.
Základní ideje: LISCH přidává nový prvek na konec řetězce,
EISCH přidává nový prvek $x$ do řetězce na řádek $h(x)$ (pokud je prázdný) nebo hned za prvek na řádku $h(x)$

### 2.9.2 Ilustrace

$U = \{1, 2, \ldots, 1000\}$, $h(x) = x \bmod 10$
množina $S = \{1, 7, 11, 53, 73, 141, 171\}$ je uložena v hašovací tabulce

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

Tabulka pro metodu LISCH vznikla následující posloupností operací:
**INSERT**(1), **INSERT**(141), **INSERT**(11), **INSERT**(73), **INSERT**(53),
**INSERT**(161), **INSERT**(7).
Pro metodu EISCH tabulka vznikla následující posloupností operací:
**INSERT**(1), **INSERT**(161), **INSERT**(11), **INSERT**(73), **INSERT**(53), **INSERT**(7), **INSERT**(141).

Provedeme **INSERT**(28), přidáváme do čvrtého rádku, výsledná tabulka vlevo je pro metodu LISCH, vpravo pro metodu EISCH.

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

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

### 2.9.3 Algoritmy

Algoritmus operace **MEMBER** je pro obě metody stejný.

**MEMBER**($x$)
Spočítáme $i := h(x)$
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

Metoda LISCH:

**INSERT**$(x)$
Spočítáme $i := h(x)$
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key \neq x$ **then**
  **if** neexistuje prázdný řádek tabulky **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je prázdný řádek $j.key := x$, $i.next := j$
  **endif**
**endif**

Metoda EISCH:

**INSERT**$(x)$
Spočítáme $k := i := h(x)$
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key \neq x$ **then**
  **if** neexistuje prázdný řádek tabulky **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je volný řádek tabulky
    $j.next := k.next$, $k.next := j$, $j.key := x$
  **endif**
**endif**

Efektivní operace **DELETE** není známá, ale i primitivní algoritmy pro operaci **DELETE** mají rozumnou očekávanou časovou složitost.

### 2.9.4 Očekávaný počet testů neúspěšného vyhledávání ($s_{n+1} \notin S$)

Popis situace: Uložena množina $S = \{s_1, s_2, \ldots, s_n\}$ do tabulky velikosti $m$, je dán prvek $s_{n+1}$ a máme zjistit, zda $s_{n+1} \in S$. Označme $a_i = h(s_i)$ pro $i = 1, 2, \ldots, n+1$, kde $h$ je použitá hašovací funkce.

Předpoklad: všechny posloupnosti $a_1, a_2, \ldots, a_{n+1}$ jsou **stejně pravděpodobné**. Výběr prázdného řádku je pevně daný, to znamená, že při stejně obsazených řadcích dostaneme vždy stejný prázdný řádek.

**Definice.** *$C(a_1, a_2, \ldots, a_n; a_{n+1})$ označuje počet testů pro zjištění, že $s_{n+1} \notin S$ – tj. to, co chceme spočítat, pro konkrétní posloupnost $a_n$.*

**Lemma.** *Očekávaný počet testů při neúspěšném vyhledávání v množině $S$ je*

$$\frac{\sum_{a_1,a_2,\ldots,a_{n+1}} C(a_1, a_2, \ldots, a_n; a_{n+1})}{m^{n+1}}$$

*Důkaz.* Sčítá se přes všechny posloupnosti $a_1, a_2, \ldots, a_{n+1}$ – a těch je $m^{n+1}$. □

---

**Definice.** *Řetězec délky $l$ v množině $S$ je maximální posloupnost adres $(b_1, b_2, \ldots, b_l)$ taková, že $b_i.next = b_{i+1}$ pro $i = 1, 2, \ldots, l-1$. Existence řetězce je dána posloupností a (při jiném pořadí by vzniknul jinak, naopak dvě množiny se stejnou posloupností a mají stejný řetězec).*

**Lemma.** *Jeden řetězec délky $l > 0$ v jedné dané posloupnosti $\{a_n\}$ přispěl k součtu $\sum C(a_1, a_2, \ldots, a_n; a_{n+1})$ počtem testů $1 + 2 + \cdots + l = l + \binom{l}{2}$.*

*Důkaz.* Když v množině $\{a_n\}$ tento řetězec existuje a adresa $a_{n+1}$ je $i$-tý prvek v řetězci, pak počet testů je $l - i + 1$ – počet testů znamená „projet“ kompletní zbytek řetězce. □

---

**Definice.** *$c_n(l)$ = počet všech řetězců délky $l$ ve všech reprezentacích $n$-prvkových množin (ztotožňujeme dvě množiny, které měly stejnou posloupnost adres při ukládaní prvků).*

**Lemma.** $\sum C(a_1, a_2, \ldots, a_n; a_{n+1}) = c_n(0) + \sum_{l=1}^{n} l c_n(l) + \sum_{l=1}^{n} \binom{l}{2} c_n(l)$

*Důkaz.*

$$\begin{aligned}\sum C(a_1, a_2, \ldots, a_n; a_{n+1}) &= c_n(0) + \sum_{l=1}^{n} (l + \binom{l}{2}) c_n(l) \\ &= c_n(0) + \sum_{l=1}^{n} l c_n(l) + \sum_{l=1}^{n} \binom{l}{2} c_n(l),\end{aligned}$$

kde $c_n(0)$ je počet prázdných řádků ve všech reprezentacích.

$c_n(0)$ je zde proto, že test na prázdnost je $O(1)$.[1]

Sčítáme všechny možné $C$ jako součet přes všechny možné řetězce *vůbec*, ve všech množinách.

□

---

**Lemma.** $c_n(0) = (m - n)m^n$

*Důkaz.* Reprezentace $S$ má $m - n$ prázdných řádků, všech posloupností $n$-adres je $m^n$, proto

$$c_n(0) = (m - n)m^n.$$

(pozor, opravdu počítáme všechny možné prázdné řetězce vůbec) □

---

**Lemma.** $\sum_{l=1}^{n} l c_n(l) = nm^n$

*Důkaz.* $\sum_{l=1}^{n} l c_n(l)$ je celková délka všech řetězců ve všech tabulkách reprezentujících všechny $n$-prvkové množiny, a proto

$$\sum_{l=1}^{n} l c_n(l) = nm^n.$$

□

---

[1] poznámka studenta - tohle mi není trochu jasné :(

**Lemma** (Rekurentní vztah pro $c_n(l)$)**.** $c_{n+1}(l) = (m-l)c_n(l) + (l-1)c_n(l-1)$.

*Důkaz.* Přidáváme prvek s adresou $a_{n+1}$. Pak řetězec délky $l$ v reprezentaci $S$ zůstal stejný, když adresa $a_{n+1}$ neležela v tomto řetězci, v opačném případě se délka řetězce zvětšila na $l+1$. Proto přidání jednoho prvku vytvořilo z řetězce délky $l$ celkem $m-l$ řetězců délky $l$ a $l$ řetězců délky $l+1$.

Vysčítáním přes všechny $n$-prvkové posloupnosti adres dostáváme

$$c_{n+1}(l) = (m-l)c_n(l) + (l-1)c_n(l-1).$$

□

---

**Lemma** (součet binomických hodnot)**.** $(m-l)\binom{l}{2} + l\binom{l+1}{2} = (m+2)\binom{l}{2} + l$

*Důkaz.*

$$\begin{aligned}(m-l)\binom{l}{2} + l\binom{l+1}{2} =& \frac{1}{2}(l^2m - lm - l^3 + l^2 + l^3 + l^2) = \\ & \frac{1}{2}(l^2m - lm + 2l^2) = \\ & \frac{1}{2}(l^2m - lm + 2(l^2 - l)) + l = \\ & (m+2)\binom{l}{2} + l.\end{aligned}$$

□

---

Označme $S_n = \sum_{l=1}^{n} \binom{l}{2} c_n(l)$ poslední sčítanec.

**Lemma** (Rekurentní vztah pro $S_n$)**.** $S_n = (m+2)S_{n-1} + (n-1)m^{n-1}$

*Důkaz.*

$$\begin{aligned}S_n =& \sum_{l=1}^{n} \binom{l}{2} c_n(l) = \\ & \sum_{l=1}^{n} \left(\binom{l}{2}(m-l)c_{n-1}(l) + \binom{l}{2}(l-1)c_{n-1}(l-1)\right) = \\ & \left(\sum_{l=1}^{n} \binom{l}{2}(m-l)c_{n-1}(l)\right) + \left(\sum_{l=0}^{n-1} \binom{l+1}{2} l c_{n-1}(l)\right) = \\ & \binom{n}{2}(m-n)c_{n-1}(n) + \\ & \left(\sum_{l=1}^{n-1}\left(\binom{l}{2}(m-l) + \binom{l+1}{2} l\right)c_{n-1}(l)\right) + \binom{1}{2} 0 c_{n-1}(0) = \\ & \sum_{l=1}^{n-1} \binom{l}{2}(m+2)c_{n-1}(l) + \sum_{l=1}^{n-1} l c_{n-1}(l) = \\ & (m+2)S_{n-1} + (n-1)m^{n-1},\end{aligned}$$

kde jsme použili, že $c_{n-1}(n) = 0$ a lemma o součtu binomických hodnot. □

---

**Lemma** (První vztah pro $S_n$)**.** $S_n = (m+2)^{n-1} \sum_{i=1}^{n-1} i\big(\frac{m}{m+2}\big)^i$

*Důkaz.* Rekurence pro $S_n$ dává

$$\begin{aligned}
S_n =&(m+2)S_{n-1} + (n-1)m^{n-1} = \\
&(m+2)^2 S_{n-2} + (m+2)(n-2)m^{n-2} + (n-1)m^{n-1} = \\
&(m+2)^3 S_{n-3} + (m+2)^2(n-3)m^{n-3} + \\
&(m+2)(n-2)m^{n-2} + (n-1)m^{n-1} = \\
&(m+2)^{n-1} S_0 + \sum_{i=0}^{n-1} (m+2)^i (n-1-i)m^{n-1-i} = \\
&(m+2)^{n-1} \sum_{i=0}^{n-1} (n-1-i)\big(\frac{m}{m+2}\big)^{n-1-i} = \\
&(m+2)^{n-1} \sum_{i=1}^{n-1} i\big(\frac{m}{m+2}\big)^i,
\end{aligned}$$

kde jsme využili, že $S_0 = 0$. □

---

**Definice.** $T_c^n = \sum_{i=1}^n ic^i$ *pro* $n = 1, 2, \ldots$ *a* $c \neq 1$

**Lemma** (Vztah pro $T_c^n$)**.**

$$T_c^n = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{c-1}$$

*Důkaz.* Z $cT_c^n = \sum_{i=1}^n ic^{i+1}$ plyne

$$\begin{aligned}
(c-1)T_c^n =& cT_c^n - T_c^n = \sum_{i=2}^{n+1} (i-1)c^i - \sum_{i=1}^{n} ic^i = \\
&nc^{n+1} + \big(\sum_{i=2}^{n} ((i-1)c^i - ic^i)\big) - c = \\
&nc^{n+1} + \big(\sum_{i=2}^{n} -c^i\big) - c = \\
&nc^{n+1} - \sum_{i=1}^{n} c^i = nc^{n+1} - \frac{c^{n+1} - c}{c-1} = \\
&\frac{nc^{n+2} - (n+1)c^{n+1} + c}{c-1}.
\end{aligned}$$

Tedy platí

$$T_c^n = \frac{nc^{n+2} - (n+1)c^{n+1} + c}{(c-1)^2}.$$

□

---

**Lemma** (Druhý vztah pro $S_n$)**.** $S_n = \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)$

*Důkaz.* Protože $\frac{m}{m+2} \neq 1$, dostáváme dosazením vztahu pro $T_c^n$ do prvního vztahu pro $S_n$, že

$$\begin{aligned}
S_n =& (m+2)^{n-1}\frac{(n-1)\big(\frac{m}{m+2}\big)^{n+1} - n\big(\frac{m}{m+2}\big)^n + \frac{m}{m+2}}{\big(\frac{m}{m+2}-1\big)^2} = \\
& \frac{1}{4}(m+2)^{n+1}\big[(n-1)\big(\frac{m}{m+2}\big)^{n+1} - n\big(\frac{m}{m+2}\big)^n + \frac{m}{m+2}\big] = \\
& \frac{1}{4}\big[(n-1)m^{n+1} - n(m+2)m^n + m(m+2)^n\big] = \\
& \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big).
\end{aligned}$$

□

---

**Věta** (Odhad nejhoršího případu)**.** *Očekávaný počet testů při neúspěšném vyhledávání je* $\frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$

*Důkaz.* Z předchozích lemmat

$$\begin{aligned}
& \frac{\sum_{a_1,a_2,\ldots,a_{n+1}} C(a_1, a_2, \ldots, a_n; a_{n+1})}{m^{n+1}} = \\
& \frac{(m-n)m^n + nm^n + \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)}{m^{n+1}} = \\
& \frac{m^{n+1} + \frac{1}{4}\big(m(m+2)^n - m^{n+1} - 2nm^n\big)}{m^{n+1}} = \\
& 1 + \frac{1}{4}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) \sim 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha).
\end{aligned}$$

□

---

Tento odhad je stejný pro obě metody – LISCH i EISCH, protože mají stejné posloupnosti adres (liší se jen pořadím prvků v jednotlivých řetězcích).

Očekávaný počet testů při neúspěšném vyhledávání je $\frac{1}{4}(e^{2\alpha}-1-2\alpha)$

### 2.9.5 Úspěšný případ ($s_j \in S$)

Očekávaný počet testů při úspěšném vyhledávání v modelu LISCH spočítáme stejnou metodou jako pro hašování se separujícími řetězci.

**Lemma.** *Pro úspěšné vyhledání prvku $s_j \in S$ je počet testů roven 1+počet porovnání klíčů při neúspěšném vyhledávání při operaci* **INSERT**$(s_j)$**.**

*Důkaz.* Při úspěšném vyhledávání procházím stejné prvky, jako při vkládání. Jednička je za poslední porovnání $s_j$, které jsem při vkládání dělat nemusel. □

---

**Lemma** (verze 0)**.** *Když $s_j$ je vložen na místo $h(s_j)$, nebyl porovnáván žádný klíč a test pro úspěšné vyhledávání bude* 1*.*

*Důkaz.* Plyne jednoduše z předchozího lemmatu. Pozor, v předchozí části jsem měl 1 test na prázdnost, zde beru v úvahu pouze porovnání klíčů.[2] □

---

**Lemma** (verze $> 0$)**.** *Když $h(s_j)$ byl na na $i$-tém místě v řetězci délky $l$, pak bylo při operaci* **INSERT**$(s_j)$ *použito $l - i + 1$ porovnání klíčů a teď se použije $l - i + 2$ testů.*

*Důkaz.* Při neúspěšném vyhledávání jsem musel projet celý zbytek řetězce. □

---

**Lemma.** *Očekávaný počet porovnání klíčů při neúspěšném vyhledávání je pro $i$-prvkovou množinu $\frac{1}{4}\big((1+\frac{2}{m})^i - 1 + \frac{2i}{m}\big)$.*

*Důkaz.* Stejně, jako v předchozí sekci (s tím rozdílem, že u prázdných řádků nepočítám 1[3]) dostaneme, že očekávaný počet porovnání klíčů při neúspěšném vyhledávání je

$$\frac{1}{m^{i+1}}\Big(\sum_{l=1}^{i}\big(l + \binom{l}{2}\big)c_i(l)\Big) =$$
$$\frac{1}{m^{i+1}}\Big(im^i + \frac{1}{4}\big(m(m+2)^i - m^{i+1} - 2im^i\big)\Big) =$$
$$\frac{1}{4}\big((1+\frac{2}{m})^i - 1 + \frac{2i}{m}\big).$$

□

---

**Lemma.** *Tedy očekávaný počet testů při úspěšném vyhledávání v $n$-prvkové množině je roven $1 + n$-tina součtu očekávaného počtu porovnání klíčů při neúspěšném vyhledávání v $i$-prvkové množině, kde $i$ probíhá čísla $0, 1, \ldots, n-1$.*

*Důkaz.* Skutečně, každý prvek z $n$-prvkové množiny, podle kterých průměruji úspěšné vyhledávání, tam musel být vložen, tj.

$$\frac{\sum_{i=0}^{n-1} 1 + \text{očekávaný počet testů v i-prvkové množině}}{n}$$

a jedničku můžu strčit dopředu. □

[2] poznámka studenta - toto mi opět není jasné

[3] jak píšu výše, není mi jasné proč

---

**Lemma.** *Součet očekávaných počtů porovnání klíčů při neúspěšném vyhledávání v i-prvkové množině, kde i probíhá čísla* $0, 1, \ldots, n-1$*, je*

$$\frac{m}{8}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) + \frac{n^2-n}{4m}$$

.

*Důkaz.*

$$\sum_{i=0}^{n-1} \frac{1}{4}\big[(1+\frac{2}{m})^i - 1 + \frac{2i}{m}\big] = \frac{1}{4}\frac{(1+\frac{2}{m})^n - 1}{1+\frac{2}{m}-1} - \frac{n}{4} + \frac{\binom{n}{2}}{2m} = \frac{m}{8}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) + \frac{n^2-n}{4m}.$$

□

---

**Věta.** *Očekávaný počet testů v úspěšném případě pro n-prvkovou množinu je*

$$1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$$

*Důkaz.*

$$1 + \frac{m}{8n}\big((1+\frac{2}{m})^n - 1 - \frac{2n}{m}\big) + \frac{n-1}{4m} \sim 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}.$$

□

---

Pro LISCH očekávaný počet testů v úspěšném případě pro $n$-prvkovou množinu je $1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$.

**Věta.** *Pro metodu EISCH je očekáváný počet testů v úspěšném případě*

$$\frac{m}{n}\big((1+\frac{1}{m})^n - 1\big) \sim \frac{1}{\alpha}(e^{\alpha} - 1).$$

Výpočet je komplikovanější, musí se použít složitější metoda (metoda EISCH dává nový prvek hned za místo, kde má být uložen).

Chyba aproximace pro tyto odhady je $O(\frac{1}{m})$.

## 2.10 Metody LICH, EICH, VICH

### 2.10.1 Popis

- LICH – late-insertion coalesced hashing
- EICH – early-insertion coalesced hashing
- VICH – varied-insertion coalesced hashing.

Základní idea: Metody používají pomocnou paměť. Tabulka je rozdělená na adresovací část a na pomocnou paměť, která není dostupná pomocí hašovací funkce, ale pomáhá při řešení kolizí. Metody se liší operací **INSERT**. Všechny metody při kolizi nejprve použijí řádek tabulky z pomocné části a teprve, když je pomocná část zaplněna, používají adresovací část.

Metoda LICH: při **INSERT**u vkládá prvek vždy na konec řetězce.

Metoda EICH: při **INSERT**u vkládá prvek $x$ do řetězce vždy na místo hned za řádkem $h(x)$.

Metoda VICH: Při **INSERT**u, když nový řádek je z pomocné části, tak je vložen s novým prvkem na konec řetězce, když je pomocná část paměti vyčerpána, tak se řádek s novým prvkem vkládá do řetězce za poslední řádek z pomocné části tabulky. Když řetězec neobsahuje žádný řádek z pomocné paměti, tak se řádek s novým prvkem $x$ vkládá hned za řádek $h(x)$.

Idea: pomocná část má zabránit rychlému srůstání řetězců.

Tyto metody nepodporují přirozené efektivní algoritmy pro operaci **DELETE**.

### 2.10.2 Ilustrace

$U = \{1, 2, \ldots, 1000\}$, $h(x) = x \bmod 10$,
$S = \{1, 7, 11, 53, 73, 141, 161\}$. Tabulka má 12 řádků a má tvar

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

Hašovací tabulka vznikla posloupnostmi operací:
Pro metodu LICH:
**INSERT**(1), **INSERT**(73), **INSERT**(141), **INSERT**(53), **INSERT**(11),
**INSERT**(161), **INSERT**(7).
Pro metodu EICH:

**INSERT**(1), **INSERT**(73), **INSERT**(161), **INSERT**(53), **INSERT**(11),
**INSERT**(141), **INSERT**(7),

ale nedodržovalo se, že se nejdřív zaplňují řádky z pomocné části. Při dodržování tohoto pravidla takováto tabulka nemůže vzniknout.
Pro metodu VICH:
**INSERT**(1), **INSERT**(73), **INSERT**(141), **INSERT**(53), **INSERT**(161),
**INSERT**(11), **INSERT**(7).

Aplikujeme operace **INSERT**(28) a **INSERT**(31), nové řádky budou řádky číslo 4 a 9. Tabulka vytvořená pomocí metody LICH je na levé straně, metodou VICH je v prostředku a metodou EICH je na pravé straně.

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

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

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

### 2.10.3 Algoritmy

Algoritmus operace **MEMBER** je pro tyto metody stejný jako pro LISCH a EISCH

**MEMBER**($x$)
Spočítáme $i := h(x)$
**while** $i.next \neq$prázdné a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key = x$ **then** **Výstup**: $x \in S$ **else** **Výstup**: $x \notin S$ **endif**

Algoritmus operace **INSERT** je pro metodu LICH stejný jako pro metodu LISCH a pro metodu EICH je stejný jako pro metodu EISCH s jediným doplňkem, pokud existuje prázdný řádek v pomocné části, tak $j$-tý řádek je z pomocné části. Tento předpoklad je i pro algoritmus **INSERT** pro metodu VICH.

Metoda LICH

**INSERT**($x$)
Spočítáme $i := h(x)$
**if** $i.next = NIL$ **then** $i.next = x$, stop **endif**
**while** $i.next \neq NIL$ a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key \neq x$ **then**
  **if** neexistuje prázdný řádek tabulky **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je prázdný řádek, $j.key := x$, $i.next := j$
  **endif**
**endif**

Metoda EICH

**Insert**$(x)$
Spočítáme $k := i := h(x)$
**if** $i.next = NIL$ **then** $i.next = x$, stop **endif**
**while** $i.next \neq NIL$ a $i.key \neq x$ **do** $i := i.next$ **enddo**
**if** $i.key \neq x$ **then**
  **if** neexistuje prázdný řádek tabulky **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je volný řádek tabulky
    $j.next := k.next$, $k.next := j$, $j.key := x$
  **endif**
**endif**

Metoda VICH

**INSERT**$(x)$
Spočítáme $i := h(x)$
**if** $i.next = NIL$ **then** $i.next = x$, stop **endif**
**while** $i.next \neq NIL$ a $i.key \neq x$ **do**
  **if** $k$ není definováno a $i.next < m$ **then** $k := i$ **endif**
**Poznámka**: Podmínka pro $k$ je splněna, když jsme byli na začátku nebo v pomocné části, podmínka na $i.next$ je splněna, když $i.next$ není v pomocné části.
  $i := i.next$
**enddo**
**if** $i.key \neq x$ **then**
  **if** neexistuje prázdný řádek **then**
    **Výstup**: přeplnění
  **else**
    nechť $j$ je volný řádek, $j.key := x$
    **if** $k$ není definováno **then**
      $i.next := j$
    **else**
      $j.next := k.next$, $k.next := j$
    **endif**
  **endif**
**endif**

### 2.10.4 Očekávaný počet testů

*(pozn. studenta - opravdu nevím, jestli je nutné se tohle učit.)*

Značení: $n$ – velikost uložené množiny,
$m$ – velikost adresovací části tabulky,
$m'$ – velikost tabulky,
$\alpha = \frac{n}{m'}$ – faktor zaplnění,
$\beta = \frac{m}{m'}$ – adresovací faktor,
$\lambda$ – jediné nezáporné řešení rovnice $e^{-\lambda} + \lambda = \frac{1}{\beta}$.

Očekávaný počet testů pro metodu LICH
neúspěšný případ:
  $e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, když $\alpha \leq \lambda\beta$,

$\frac{1}{\beta} + \frac{1}{4}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1)(3 - \frac{2}{\beta} + 2\lambda) - \frac{1}{2}(\frac{\alpha}{\beta} - \lambda)$, když $\alpha \geq \lambda\beta$
úspěšný případ:
$1 + \frac{\alpha}{2\beta}$, když $\alpha \leq \lambda\beta$,
$1 + \frac{\beta}{8\alpha}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1 - 2(\frac{\alpha}{\beta} - \lambda))(3 - \frac{2}{\beta} + 2\lambda) + \frac{1}{4}(\frac{\alpha}{\beta} + \lambda) + \frac{\lambda}{4}(1 - \frac{\lambda\beta}{\alpha})$, když $\alpha \geq \lambda\beta$.

Očekávaný počet testů pro metodu EICH
neúspěšný případ:
$e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, když $\alpha \leq \lambda\beta$,
$e^{2(\frac{\alpha}{\beta}-\lambda)}(\frac{3}{4} + \frac{\lambda}{2} - \frac{1}{2\beta}) + e^{\frac{\alpha}{\beta}-\lambda}(\frac{1}{\beta} - 1) + (\frac{1}{4} - \frac{\alpha}{2\beta} + \frac{1}{2\beta})$, když $\alpha \geq \lambda\beta$
úspěšný případ:
$1 + \frac{\alpha}{2\beta}$, když $\alpha \leq \lambda\beta$,
$1 + \frac{\alpha}{2\beta} + \frac{\beta}{\alpha}((e^{\frac{\alpha}{\beta}-\lambda} - 1)(1 + \lambda) - (\frac{\alpha}{\beta} - \lambda))(1 + \frac{\lambda}{2} + \frac{\alpha}{2\beta}))$, když $\alpha \geq \lambda\beta$.

Očekávaný počet testů pro metodu VICH
neúspěšný případ:
$e^{-\frac{\alpha}{\beta}} + \frac{\alpha}{\beta}$, když $\alpha \leq \lambda\beta$,
$\frac{1}{\beta} + \frac{1}{4}(e^{2(\frac{\alpha}{\beta}-\lambda)} - 1)(3 - \frac{2}{\beta} + 2\lambda) - \frac{1}{2}(\frac{\alpha}{\beta} - \lambda)$, když $\alpha \geq \lambda\beta$
úspěšný případ:
$1 + \frac{\alpha}{2\beta}$, když $\alpha \leq \lambda\beta$,
$1 + \frac{\alpha}{2\beta} + \frac{\beta}{\alpha}((e^{\frac{\alpha}{\beta}-\lambda} - 1)(1 + \lambda) - (\frac{\alpha}{\beta} - \lambda))(1 + \frac{\lambda}{2} + \frac{\alpha}{2\beta})) + \frac{1-\beta}{\alpha}(\frac{\alpha}{\beta} - \lambda - e^{\frac{\alpha}{\beta}-\lambda} + 1)$, když $\alpha \geq \lambda\beta$.

Chyba aproximace pro tyto odhady je $O(\log \frac{m'}{\sqrt{m'}})$.

## 2.11 Hašování s lineárním přidáváním

### 2.11.1 Popis

Tabulka má jedinou položku – key

Základní idea: Při operaci **INSERT**$(x)$ vložíme $x$ na řádek $h(x)$, když je prázdný, v opačném případě nalezneme nejmenší $i$ takové, že řádek $h(x) + i \bmod m$ je prázdný, a tam vložíme $x$. Tato metoda byla motivována snahou o co největší využití paměti.

Komentář: Metoda vyžaduje minimální velikost paměti.

V tabulce se vytvářejí shluky použitých řádků, a proto při velkém zaplnění metoda vyžaduje velký počet testů.

Metoda nepodporuje efektivní implementaci operace **DELETE**.

Při vyhledávání je třeba testovat, zda nevyšetřujeme podruhé první vyšetřovaný řádek, a pro zjištění přeplnění je vhodné mít uložen počet vyplněných řádků v tabulce. Pro standarní paměti není výhodná. Při použití cache-paměti se výrazně mění její hodnocení. Důvodem je, že v tomto případě hraje klíčovou roli nikoliv počet testů, ale počet přechodů mezi různými úrovněmi paměti. Protože tabulka je reprezentovaná polem, tak je tento počet menší než u jiných metod. Proto se tato metoda doporučuje pro počítače s cache-pamětí.

### 2.11.2 Algoritmy

**MEMBER**$(x)$
Spočítáme $i := h(x)$, $h := i$
**if** $i.key = x$ **then Výstup** $x \in S$, stop **endif**
**if** $i.key$ =prázdný **then** Výstup: $x \notin S$, stop **endif**
$i := i + 1$
**while** $i.key \neq$prázdný a $i.key \neq x$ a $i \neq h$ **do** $i := i + 1 \bmod m$ **enddo**
**if** $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

**INSERT**$(x)$
Spočítáme $i := h(x)$, $j := 0$
**while** $i.key \neq$prázdný a $i.key \neq x$ a $j < m$ **do** $i := i + 1 \bmod m$, $j := j + 1$ **enddo**
**if** $j = m$ **then Výstup**: přeplnění, stop **endif**
**if** $i.key$ =prázdný **then** $i.key := x$ **endif**

### 2.11.3 Ilustrace

Máme universum $U = \{1, 2, \ldots, 1000\}$, hašovací funkci $h(x) = x \bmod 10$ a množinu $S = \{1, 7, 11, 53, 73, 141, 161\}$. Tato množina je uložena v levé tabulce. Provedeme operaci **INSERT**(35). Výsledek je uložen v pravé tabulce.

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

| řádek | key |
|---|---|
| P(0) | |
| P(1) | 1 |
| P(2) | 11 |
| P(3) | 73 |
| P(4) | 141 |
| P(5) | 161 |
| P(6) | 53 |
| P(7) | 7 |
| P(8) | 35 |
| P(9) | |

Tabulka vznikla posloupností operací:
**INSERT**(1), **INSERT**(11), **INSERT**(73), **INSERT**(141), **INSERT**(161),
**INSERT**(53), **INSERT**(7).

### 2.11.4 Očekávaný počet testů

Očekávaný počet testů pro neúspěšný případ: $\approx \frac{1}{2}(1 + \left(\frac{1}{1-\alpha}\right)^2)$.
Očekávaný počet testů pro úspěšný případ: $\approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$.

## 2.12 Dvojité hašování

### 2.12.1 Popis

Základní nevýhoda předchozí metody je způsob výběru dalšího řádku. Je velmi determinován a důsledkem je vznik shluku řádků, který vede k výraznému zpomalení metody.
Idea jak odstranit tuto nevýhodu: Použijeme dvě hašovací funkce $h_1$ a $h_2$ a při operaci **INSERT**$(x)$ nalezneme nejmenší $i = 0, 1, \ldots$ takové, že $(h_1(x) + ih_2(x)) \bmod m$ je prázdný řádek, a tam uložíme prvek $x$.
Tabulka má jedinou položku – key.
Požadavky na korektnost: Pro každé $x$ musí být $h_2(x)$ a $m$ nesoudělné (jinak prvek $x$ nemůže být uložen na libovolném řádku tabulky).

Předpoklad pro výpočet očekávaného počtu testů: posloupnost $\{h_1(x) + ih_2(x)\}_{i=0}^{m-1}$ je náhodná permutace množiny řádků tabulky.

Nevýhoda: Uvedená metoda nepodporuje operaci **DELETE**.
Poznámka: Metoda hašování s lineárním přidáváním je speciální případ dvojitého hašování, kde $h_2(x) = 1$ pro každé $x \in U$.

### 2.12.2 Algoritmy

**MEMBER**$(x)$
Spočítáme $i := h_1(x)$, $h := h_2(x)$, $j := 0$
**while** $i.key \neq$prázdný a $i.key \neq x$ a $j < m$ **do** $i := i + h \bmod m$, $j := j + 1$ **enddo**
**if** $i.key = x$ **then Výstup**: $x \in S$ **else Výstup**: $x \notin S$ **endif**

**INSERT**$(x)$
Spočítáme $i := h_1(x)$, $h := h_2(x)$, $j := 0$
**while** $i.key \neq$prázdný a $i.key \neq x$ a $j < m$ **do** $i := i + h \bmod m$, $j := j + 1$ **enddo**
**if** $j = m$ **then Výstup**: přeplnění, stop **endif**
**if** $i.key =$prázdný **then** $i.key := x$ **endif**

### 2.12.3 Ilustrace

Mějme universum $U = \{1, 2, \ldots, 1000\}$. Hašovací funkce jsou $h_1(x) = x \bmod 10$ a $h_2(x) = 1 + 2(x \bmod 4)$, když $x \bmod 4 \in \{0, 1\}$, $h_2(x) = 3+2(x \bmod 4)$, když $x \bmod 4 \in \{2, 3\}$. Množina je $S = \{1, 7, 11, 53, 73, 141, 161\}$. Tato množina je uložena v levé tabulce. Aplikujme **INSERT**$(35)$. Pak $h_2(35) = 9$, tedy posloupnost pro $x = 35$ je

$$(5, 4, 3, 2, 1, 0, 9, 8, 7, 6).$$

Výsledek je uložen v pravé tabulce.

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

| řádek | key |
|---|---|
| P(0) | 11 |
| P(1) | 1 |
| P(2) | 35 |
| P(3) | 73 |
| P(4) | 141 |
| P(5) | 7 |
| P(6) | 53 |
| P(7) | 161 |
| P(8) | |
| P(9) | |

Tabulka vznikla posloupností operací:
**INSERT**(1), **INSERT**(73), **INSERT**(53), **INSERT**(141), **INSERT**(161),
**INSERT**(11), **INSERT**(7).

### 2.12.4 Očekávaný počet testů - neúspěšný případ

**Definice.** $q_i(n,m)$ – *když tabulka má $m$ řádků a je v ní obsazeno $n$ řádků, tak je to pravděpodobnost, že pro každé $j = 0, 1, \ldots, i-1$ je řádek $h_1(x) + jh_2(x)$ obsazen.*

**Pozorování.** $q_0(n,m) = 1$

*Důkaz.* Krajní případ - je určitě obsazen pro $j = 0 \ldots -1$ □

---

**Pozorování.** $q_1(n,m) = \frac{n}{m}$

*Důkaz.* Bez druhé hešovací funkce, tj. zkusím jen jednou. □

---

**Pozorování.** $q_2(n,m) = \frac{n(n-1)}{m(m-1)}$

*Důkaz.* První hešovací funkce se nestrefí, druhá také ne (a druhá funkce je nesoudělná s $m$ a je náhodná) □

---

**Lemma** (Obecný odhad $q_i$)**.** *Obecně platí*

$$q_i(n,m) = \frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)}$$

*Důkaz.* Zobecnění předchozích pozorování. □

---

**Definice.** *$C(n,m)$ – očekávaný počet testů v neúspěšném vyhledávání, když tabulka má $m$ řádků a $n$ jich je obsazeno (tj. to, co chci spočítat)*

**Lemma.** $C(n,m) = \sum_{j=0}^{n}(j+1)(q_j(n,m) - q_{j+1}(n,m))$

*Důkaz.* Pro každé $j$ vezmu pravděpodobnost, že jen pro $i \leq j$ je $h_1(x) + jh_2(x)$ obsazen, pro všechny další je volný; pro každé takové $j$ je počet testů $j+1$ □

---

**Lemma.** $C(n,m) = \sum_{j=0}^{n} q_j(n,m)$

*Důkaz.* Předchozí lemma + úprava indexů □

---

**Lemma.** $C(0,m) = 1$ *pro každé* $m$

*Důkaz.* Vyplývá z předchozího lemmatu + $q_0(0,m) = 1$ □

---

**Lemma.** $q_j(n,m) = \frac{n}{m} q_{j-1}(n-1,m-1)$ *pro všechna* $j, n > 0$ *a* $m > 1$

*Důkaz.* Z obecného odhadu $q_i$

$$q_i(n,m) = \frac{\prod_{j=0}^{i-1}(n-j)}{\prod_{j=0}^{i-1}(m-j)} = \frac{n\prod_{j=0}^{i-1}((n-1)-j)}{m\prod_{j=0}^{i-1}((m-1)-j)} = \frac{n}{m} q_{j-1}(n-1,m-1)$$

□

---

**Lemma.** $C(n,m) = 1 + \frac{n}{m} C(n-1,m-1)$

*Důkaz.*

$$C(n,m) = \sum_{j=0}^{n} q_j(n,m) = 1 + \frac{n}{m}(\sum_{j=0}^{n-1} q_j(n-1,m-1)) = 1 + \frac{n}{m} C(n-1,m-1).$$

□

---

**Lemma.** *Očekávaný počet dotazů při neúspěšném vyhledávání v tabulce s $m$ řádky, z nichž $n$ je obsazeno, je* $C(n,m) = \frac{m+1}{m-n+1}$

*Důkaz.* Indukcí.

Když $n = 0$, pak $C(0, m) = \frac{m+1}{m-0+1} = 1$ a tvrzení platí.

Předpokládáme, že tvrzení platí pro $n - 1 \geq 0$ a pro každé $m \geq n - 1$ a dokážeme tvrzení pro $n$ a $m \geq n$. Platí

$$\begin{aligned} C(n,m) =& 1 + \frac{n}{m} C(n-1, m-1) = \\ & 1 + \frac{n((m-1)+1)}{m((m-1)-(n-1)+1)} = \\ & 1 + \frac{n}{m-n+1} = \frac{m+1}{m-n+1}. \end{aligned}$$

□

---

**Věta.** *Očekávaný počet dotazů při neúspěšném vyhledávání je přibližně* $\frac{1}{1-\alpha}$

*Důkaz.* Z předchozího lemmatu $C(n, m) = \frac{m+1}{m-n+1} \sim \frac{1}{1-\alpha}$. □

---

Očekávaný počet dotazů při neúspěšném vyhledávání je přibližně $\frac{1}{1-\alpha}$.

### 2.12.5 Úspěšný případ

Použijeme opět stejnou metodu ze separujících řetězců, proto jen stručně.

**Věta.** *Očekávaný počet dotazů při úspěšném vyhledávání je přibližně* $\frac{1}{\alpha} \ln(\frac{1}{1-\alpha})$

*Důkaz.* Počet dotazů při vyhledávání $x$ pro $x \in S$ je stejný jako byl počet dotazů při vkládání $x$ do tabulky. Tedy očekávaný počet dotazů při úspěšném vyhledávání v tabulce s $m$ řádky, z nichž $n$ je obsazeno, je

$$\begin{aligned} \frac{1}{n} \sum_{i=0}^{n-1} C(i,m) =& \frac{1}{n} \sum_{i=0}^{n-1} \frac{m+1}{m-i+1} = \\ & \frac{m+1}{n} \Big( \sum_{j=1}^{m+1} \frac{1}{j} - \sum_{j=1}^{m-n+1} \frac{1}{j} \Big) \approx \\ & \frac{1}{\alpha} \ln\Big(\frac{m+1}{m-n+1}\Big) \approx \frac{1}{\alpha} \ln\Big(\frac{1}{1-\alpha}\Big). \end{aligned}$$

□

---

Očekávaný počet dotazů při úspěšném vyhledávání je přibližně $\frac{1}{\alpha} \ln(\frac{1}{1-\alpha}$.

Následující tabulka ukazuje tyto hodnoty v závislosti na velikosti $\alpha$.

| hodnota $\alpha$ | 0.5 | 0.7 | 0.9 | 0.95 | 0.99 | 0.999 |
|---|---|---|---|---|---|---|
| $\frac{1}{1-\alpha}$ | 2 | 3.3 | 10 | 20 | 100 | 1000 |
| $\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$ | 1.38 | 1.70 | 2.55 | 3.15 | 4.65 | 6.9 |

## 2.13 Porovnání efektivity hašovacích algoritmů

### 2.13.1 Neúspěšné vyhledávání

- Hašování s uspořádanými řetězci
- Hašování s řetězci=Hašování s přemísťováním
- Hašování s dvěma ukazateli
- VICH=LICH
- EICH
- LISCH=EISCH
- Dvojité hašování
- Hašování s lineárním přidáváním

### 2.13.2 Úspěšné vyhledávání

Pořadí metod hašování podle očekávaného počtu testů:

- Hašování s uspořádanými řetězci=Hašování s řetězci=Hašování s přemísťováním
- Hašování s dvěma ukazateli,
- VICH
- LICH
- EICH
- EISCH
- LISCH
- Dvojité hašování
- Hašování s lineárním přidáváním

Poznámka: Metoda VICH při neúspěšném vyhledávání pro $\alpha < 0.72$ a při úspěšném vyhledávání pro $\alpha < 0.92$ vyžaduje menší očekávaný počet testů než metoda s dvěma ukazateli.

Při neúspěšném vyhledávání jsou metody VICH a LICH stejné a jsou o 8% lepší než EICH a o 15% než metody LISCH a EISCH. Při úspěšném vyhledávání je VICH nepatrně lepší než LICH a EICH o 3% lepší než EISCH a o 7% lepší než LISCH.

### 2.13.3 Očekávaný počet testů při úplně zaplněné tabulce

- Metoda s přemísťováním: neúspěšné vyhledávání 1.5, úspěšné vyhledávání 1.4.
- Metoda s dvěma ukazateli: úspěšné i neúspěšné vyhledávání 1.6.
- VICH: neúspěšné vyhledávání 1.79, úspěšné vyhledávání 1.67.
- LICH: neúspěšné vyhledávání 1.79, úspěšné vyhledávání 1.69.
- EICH: neúspěšné vyhledávání 1.93, úspěšné vyhledávání 1.69.
- EISCH: neúspěšné vyhledávání 2.1, úspěšné vyhledávání 1.72.

- LISCH: neúspěšné vyhledávání 2.1, úspěšné vyhledávání 1.8.

Metodu s lineárním přidáváním je dobré použít jen pro $\alpha < 0.7$, metodu s dvojitým hašováním pro $\alpha < 0.9$, pak čas pro neúspěšné vyhledávání rychle narůstá.

### 2.13.4 Vliv $\beta = \frac{m}{m'}$ při srůstajícím hašování

Při úspěšném vyhledávání je optimální hodnota $\beta = 0.85$, při neúspěšném vyhledávání je optimální hodnota $\beta = 0.78$. V praxi se doporučuje použít hodnotu $\beta = 0.86$ (uvedené výsledky byly pro tuto hodnotu $\beta$).

### 2.13.5 Komentář

Metody se separujícími řetězci a srůstající hašování používají více paměti (při srůstajícím hašování součet adresovací a pomocné části). Metoda s přemísťováním a metoda dvojitého hašování vyžadují více času – na přemístění prvku a na výpočet druhé hašovací funkce.

## 2.14 Další otázky

### 2.14.1 Jak nalézt volný řádek

Za nejlepší metodu se považuje mít seznam (zásobník) volných řádků a z jeho vrcholu brát volný řádek a po úspěšné operaci **DELETE** tam zase řádek vložit (pozor při operaci **DELETE** ve strukturách, které nepodporují **DELETE**).

### 2.14.2 Jak řešit přeplnění

Standardní model: Dána základní velikost tabulky $m$ a pracuje se s tabulkami s $2^i m$ řádky pro vhodné $i = 0, 1, \ldots$. Vhodné $i$ znamená, že faktor zaplnění $\alpha$ je v intervalu $< \frac{1}{4}, 1 >$ (s výjimkou $i = 0$, kde se uvažuje pouze horní mez). Při překročení meze se zvětší nebo zmenší $i$ a všechna data se přehašují do nové tabulky.
Výhoda: Po přehašování do nové tabulky je počet operací, které vedou k novému přehašovávání, roven alespoň polovině velikosti uložené množiny.
Praktické použití: Nedržet se striktně mezí, používat malé pomocné tabulky při přeplnění a posunout velké přehašování na dobu klidu (aby systém nenechal uživatele v normální době čekat).

### 2.14.3 Jak řešit DELETE v metodách, které ho nepodporují

Použít ideu tzv. 'falešného **DELETE**'. Odstranit prvek, ale řádek neuvolnit (i v klíči nechat nějakou hodnotu, která bude znamenat, že řádek je prázdný, položky podporující práci s tabulkami neměnit). Řádek nebude v seznamu volných řádků, ale operace **INSERT**, když testuje tento řádek, tam může vložit nový prvek. Když je alespoň polovina použitých řádků takto blokována, je vhodné celou strukturu přehašovat. Pravděpodobnostní analýzu tohoto modelu neznám.

### 2.14.4 Otevřené problémy

Jak využít ideje z hašování s uspořádanými řetězci pro ostatní metody řešení kolizí (jmenovitě pro srůstající hašování).

Jakou metodu použít pro operaci **DELETE** ve srůstajícím hašování (problém je zachovat náhodnost uložené množiny a tím platnost odhadu na složitost operací).

Jak nalézt druhou hašovací funkci pro metodu dvojitého hašování, aby vzniklé posloupnosti adres při operaci **INSERT** se chovaly jako náhodné.

### 2.14.5 Předpoklady a jejich splnitelnost

Připomeňme si předpoklady pro předchozí uvedené výsledky o hašování:

1. Hašovací funkce se rychle spočítá (v čase $O(1)$);
2. Hašovací funkce rovnoměrně rozděluje univerzum (to znamená, že pro dvě různé hodnoty $i$ a $j$ hašovací funkce platí $-1 \leq |h^{-1}(i)| - |h^{-1}(j)| \leq 1$);
3. Vstupní data jsou rovnoměrně rozdělená.

Diskutujme splnitelnost těchto předpokladů.
Předpoklad 1) je jasný.
Předpoklad 2) – je výhodné, když rozdělení univerza hašovací funkcí kopíruje známé rozdělení vstupních dat. Toto se použilo při návrhu překladače pro FORTRAN (Lum 1971). V překladači byla použita metoda separovaných řetězců a hašovací funkce, která preferovala obvyklé názvy identifikátorů. Výsledky byly měřeny, když se překladač FORTRANu použil pro standardní výpočet. Získané výsledky se porovnávaly s teoretickými výpočty za našich předpokladů. V následující tabulce můžete porovnat výsledky získané teoretickými výpočty a naměřené hodnoty. Porovnání výsledků:

| hodnota $\alpha$ | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 |
|---|---|---|---|---|---|
| experiment | 1.19 | 1.25 | 1.28 | 1.34 | 1.38 |
| teorie | 1.25 | 1.30 | 1.35 | 1.40 | 1.45 |

Závěr: Podmínky 1) a 2) můžeme splnit, když známe rozložení vstupních dat, můžeme dosáhnout ještě lepších výsledků.

Nevýhoda: Rozložení vstupních dat nemůžeme ovlivnit a obvykle ho ani neznáme. Je reálné, že rozdělení vstupních dat bude nevhodné pro použitou hašovací funkci. Důsledek – na počátku 70. let se začalo ustupovat od hašování. Hledal se postup, který by se vyhnul uvedenému problému s bodem 3).

Řešení navrhli Carter a Wegman (1977), když přišli s metodou univerzálního hašování, která obchází požadavek 3). To vedlo k novému rozsáhlému používání hašování.

Nalezenému řešení je věnován následující text.

## 2.15 Univerzální hašování

### 2.15.1 Základní idea

Místo jedné funkce máme množinu $H$ funkcí z univerza do tabulky velikosti $m$ takových, že pro každou množinu $S \subseteq U$, $|S| \leq m$ se většina funkcí chová dobře vůči $S$ (tj. hašovací funkce má jen málo kolizí v množině $S$). Hašovací funkci zvolíme náhodně z $H$ (s rovnoměrným rozdělením) a hašujeme pomocí takto zvolené funkce.

Tedy ještě jednou – nemám jednu hašovací funkci, ale mám *více* hašovacích funkcí, pro každou *množinu* $S$ se mi *většina* z nich chová rozumně – a z nich *náhodně* jednu vyberu; tedy není nutné náhodné rozložení $S$.

### 2.15.2 Modifikace ideje

Ověřování vlastností vyžaduje znalost velikosti množiny $H$.

Mám ale problém - rychlá vyčíslitelnost $h(x)$ vyžaduje analytické zadání funkcí v $H$, ale zjištění rovnosti dvou analyticky zadaných funkcí na univerzu $U$ je problematické. Řešením problému je použití indexové množiny $I$, kterou si oindexuji funkce v $H$.

To znamená, že $H = \{h_i \mid i \in I\}$ a dvě funkce jsou různé, když mají různé indexy. Pak velikost systému, tj. velikost $H$, bude velikost indexové množiny.

Místo zvolení hašovací funkce budeme volit náhodně index s rovnoměrným rozložením a když zvolíme index $i$, pak budeme pracovat s hašovací funkcí $h_i$. Očekávaná hodnota náhodné proměnné $f$ z množiny $I$ do reálných čísel bude průměr přes $I$, tj. $\frac{\sum_{i \in I} f(i)}{|I|}$.

### 2.15.3 Formální definice $c$-univerzálních systémů

Nechť $U$ je univerzum. Soubor funkcí $H = \{h_i \mid i \in I\}$ z univerza $U$ do množiny $\{0, 1, \ldots, m-1\}$ se nazývá *$c$-univerzální* ($c$ je kladné reálné číslo), když

$$\forall x, y \in U,\ x \neq y \text{ platí } |\{i \in I \mid h_i(x) = h_i(y)\}| \leq \frac{c|I|}{m}.$$

Omezuji tedy počet kolizních funkcí pro libovolnou množinu.

Jako ekvivalentní definici lze použít toto tvrzení: systém funkcí $H$ z univerza $U$ do množiny $\{0, 1, \ldots, m-1\}$ je $c$-univerzální, když vybereme-li $h \in H$ s rovnoměrným rozdělením, pak pro každá dvě různá $x, y \in U$ platí

$$\text{Prob}(h(x) = h(y)) \leq \frac{c}{m}.$$

Problémy: existence $c$-univerzálních systémů,
vlastnosti $c$-univerzálních systémů (zda splňují požadované ideje).

### 2.15.4 Existence univerzálních systémů

Bez újmy na obecnosti můžu vzít univerzum $U$, které bude vypadat $U = \{0, 1, \ldots, N-1\}$ pro prvočíslo $N$. (Stačí si uvědomit, že každé univerzum můžeme považovat za univerzum tvaru $\{0, 1, \ldots, N-1\}$ pro nějaké $N$ a že mezi čísly $N$ a $2N$ vždy existuje nějaké prvočíslo.)

Definujme si množinu funkcí $H$ pro univerzum $U$ pro nějaké $m$.

**Definice.** *$H = \{h_{a,b} \mid (a,b) \in U \times U\}$, kde $h_{a,b}(x) = ((ax+b) \bmod N) \bmod m$ (tj. indexová množina je $U \times U$ a její velikost je $N^2$).*

Výhoda: funkce z množiny $H$ umíme rychle vyčíslit. Nevýhoda: indexová množina je velikost univerza na druhou.

**Lemma.** *Pro $x, y \in U$ taková, že $x \neq y$, existuje maximálně $m\left(\lceil \frac{N}{m} \rceil\right)^2$ dvojic $(a,b) \in U \times U$ takových, že $h_{a,b}(x) = h_{a,b}(y)$.*

*Důkaz.* Zvolme $x, y \in U$ taková, že $x \neq y$. Chceme nalézt $(a,b) \in U \times U$ takové, že $h_{a,b}(x) = h_{a,b}(y)$.

Musí existovat $i \in \{0, 1, \ldots, m-1\}$ a $r, s \in \{0, 1, \ldots, \lceil \frac{N}{m} \rceil - 1\}$ tak, že platí

$$(ax + b \equiv i + rm) \bmod N$$
$$(ay + b \equiv i + sm) \bmod N$$

($i$ je ono shodné modulo, $r$ a $s$ jsou zbytky po modulu, tedy $ax + b \bmod N = i + rm$ a $i + rm < N$, podobně s $y$)

Když $x$, $y$, $i$, $r$ a $s$ jsou konstanty a $a$ a $b$ jsou proměnné, je to systém lineárních rovnic v tělese $\mathbb{Z}/ \bmod N$, kde $\mathbb{Z}$ jsou celá čísla. Matice soustavy

$$\begin{pmatrix} x & 1 \\ y & 1 \end{pmatrix}$$

je regulární, protože $x \neq y$. Jelikož $\mathbb{Z}/ \bmod N$ je těleso (protože $N$ je prvočíslo), tak pro fixovaná $x$, $y$, $i$, $r$ a $s$ existuje právě jedno řešení této soustavy.

Tedy, pro daná $x$ a $y$, $i$ nabývá $m$ hodnot, $r$ a $s$ nabývají $\lceil \frac{N}{m} \rceil$ hodnot.

Závěr: pro každá $x, y \in U$ taková, že $x \neq y$, existuje maximálně $m\left(\lceil \frac{N}{m} \rceil\right)^2$ dvojic $(a,b) \in U \times U$ takových, že $h_{a,b}(x) = h_{a,b}(y)$. □

---

**Věta.** *Množina $H$ je $c$-univerzální pro*

$$c = \frac{\left(\lceil \frac{N}{m} \rceil\right)^2}{\left(\frac{N}{m}\right)^2}.$$

*Důkaz.* Skutečně, pro každé $x, y \in U$, $x \neq y$, je počet $(a,b) \in U \times U$ takových, že $h_{a,b}(x) = h_{a,b}(y)$, nejvýše roven

$$m\left(\lceil \frac{N}{m} \rceil\right)^2 = \frac{\left(\lceil \frac{N}{m} \rceil\right)^2}{\left(\frac{N}{m}\right)^2} \frac{N^2}{m} = \frac{\left(\lceil \frac{N}{m} \rceil\right)^2}{\left(\frac{N}{m}\right)^2} \frac{|I|}{m}.$$

□

---

**Pozorování.** *Dokázali jsme existenci $c$-univerzálních systémů pro $c$ blízké* 1.

### 2.15.5 Vlastnosti univerzálního hašování

Předpoklad: $H = \{h_i \mid i \in I\}$ je $c$-univerzální systém funkcí. (nemusí být totožný s $H$ z poslední kapitoly)

**Definice.** *Pro $i \in I$ a prvky $x, y \in U$ označme*

$$\delta_i(x,y) = \begin{cases} 1 & \text{když } x \neq y \text{ a } h_i(x) = h_i(y), \\ 0 & \text{když } x = y \text{ nebo } h_i(x) \neq h_i(y). \end{cases}$$

$\delta_i(x,y)$ je tedy 1, pokud $x, y$ v $h_i$ kolidují, jinak je 0.

**Definice.** *Pro množinu $S \subseteq U$, $x \in U$ a $i \in I$ definujme*

$$\delta_i(x,S) = \sum_{y \in S} \delta_i(x,y).$$

$\delta_i(x,S)$ tedy říká, s kolika prvky v $S$ $x$ koliduje při použití funkce $h_i$; je to tedy horní odhad řetězce, pokud $S$ je reprezentovaná množina.

**Lemma.** $\frac{1}{|I|} \sum_{i \in I} \delta_i(x,S)$ *je očekávaná délka řetězce při pevném $S$ při náhodném výběru $i$.*

*Důkaz.* Plyne jednoduše z definice. □

---

**Lemma.** *Očekávaný počet testů u operací* **MEMBER**, **INSERT** *a* **DELETE** *je $O(1+$očekávané délka řetězce$)$*

*Důkaz.* Není mi zcela jasná +1, jinak ale opravdu musíme projet celý řetězec. □

---

**Lemma.** *Pro fixovanou množinu $S \subseteq U$ a pro fixované $x \in U$ platí* $\sum_{i \in I} \delta_i(x,S) = \begin{cases} (|S|-1)c\frac{|I|}{m} & \text{když } x \in S, \\ |S|c\frac{|I|}{m} & \text{když } x \notin S. \end{cases}$

*Důkaz.* V důkazu „vtipně“ přehodíme, přes co vlastně sčítáme, a použijeme definici $c$-univerzálního systému.

Sečteme $\delta_i(x,S)$ přes všechna $i \in I$:

$$\sum_{i \in I} \delta_i(x,S) = \sum_{i \in I} \sum_{y \in S} \delta_i(x,y) = \sum_{y \in S} \sum_{i \in I} \delta_i(x,y) =$$
$$\sum_{y \in S, y \neq x} |\{i \in I \mid h_i(x) = h_i(y)\}| \leq$$
$$\sum_{y \in S, y \neq x} c\frac{|I|}{m} = \begin{cases} (|S|-1)c\frac{|I|}{m} & \text{když } x \in S, \\ |S|c\frac{|I|}{m} & \text{když } x \notin S. \end{cases}$$

□

---

**Lemma.** *Očekávaná délka řetězce pro fixovanou množinu $S \subseteq U$ a fixované $x \in U$ přes $i \in I$ s rovnoměrným rozdělením je nejvýše* $\begin{cases} c\frac{|S|-1}{m} & \text{když} \quad x \in S, \\ c\frac{|S|}{m} & \text{když} \quad x \notin S. \end{cases}$

*Důkaz.* Z předchozích lemmat:

$\delta_i(x, S)$ dává odhad na velikost řetězce $h_i(x)$ při reprezentaci množiny $S$ pomocí funkce $h_i$, tedy očekávaná délka řetězce pro fixovanou množinu $S \subseteq U$ a fixované $x \in U$ přes $i \in I$ s rovnoměrným rozdělením je nejvýše

$$\frac{1}{|I|} \sum_{i \in I} \delta_i(x, S) \leq \begin{cases} c\frac{|S|-1}{m} & \text{když} \quad x \in S, \\ c\frac{|S|}{m} & \text{když} \quad x \notin S. \end{cases}$$

□

---

**Věta.** *Očekávaný počet testů při operacích* **MEMBER**, **INSERT** *a* **DELETE** *při c-univerzálním hašování je $O(1 + c\alpha)$, kde $\alpha$ je faktor naplnění (tj. $\alpha = \frac{|S|}{m}$).*

*Důkaz.* Plyne jednoduše z předchozích lemmat. □

---

**Věta.** *Očekávaný čas pro pevnou posloupnost n operací* **MEMBER**, **INSERT** *a* **DELETE** *aplikovaných na prázdnou tabulku pro c-univerzální hašování je $O((1 + \frac{c}{2}\alpha)n)$, kde $\alpha = \frac{n}{m}$.*

*Důkaz.* Já (upravující student) nevidím, proč by to mělo platit :( □

---

Očekávaný počet testů je $O(1 + c\alpha)$

**Význam výsledku** Vzorec se jen o multiplikativní konstantu $c$ liší od vzorce pro hašování se separovanými řetězci. Přitom $c$ může být jen o málo menší než 1 a ve všech známých příkladech je $c \geq 1$. Takže, co jsme dosáhli?

Rozdíl je v předpokladech. Zde je předpoklad 3) nahrazen předpokladem, že index $i \in I$ je vybrán s rovnoměrným rozdělením, a není žádný předpoklad na vstupní data. **Výběr indexu $i$ můžeme ovlivnit, ale výběr vstupních dat nikoliv.** Můžeme zajistit rovnoměrné rozdělení výběru $i$ z $I$ nebo se k tomuto rozdělení hodně přiblížit.

### 2.15.6 Markovova nerovnost

Předpoklady: Je dána množina $S \subseteq U$, prvek $x \in U$. Očekávaná velikost $\delta_i(x, S)$ je $\mu$ a $t \geq 1$.

Předpokladejme, že $i$ je z $I$ vybráno s rovnoměrným rozdělením.

**Věta.** *Pro $t > 1$ platí: pravděpodobnost, že $\delta_i(x, S) \geq t\mu$ pro $i \in I$, je menší než $\frac{1}{t}$. .*

*Důkaz.* Označme $I' = \{i \in I \mid \delta_i(x, S) \geq t\mu\}$. Pak platí

$$\mu = \frac{\sum_{i \in I} \delta_i(x,S)}{|I|} > \frac{\sum_{i \in I'} \delta_i(x,S)}{|I|} \geq \frac{\sum_{i \in I'} t\mu}{|I|} = \frac{|I'|}{|I|} t\mu$$

Odtud

$$|I'| < \frac{|I|}{t}.$$

Tedy pravděpodobnost, že $\delta_i(x, S) \geq t\mu$, je menší než $\frac{1}{t}$. □

---

Poznámka: Toto tvrzení platí obecně a nazývá se Markovova nerovnost. Uvedený důkaz ilustruje jednoduché tvrzení pro konečný případ.

### 2.15.7 Výběr funkce ze systému

Hlavní problém: Zajištění rovnoměrného rozdělení výběru $i$ z $I$.

Provedení výběru: Budeme vybírat index z množiny $I$, budeme ho vybírat jako binární číslo tak, že každou pozici binárního čísla náhodně vybereme.

Totéž formálněji: Zakódovat indexy z množiny $I$ do čísel $0, 1, \ldots, |I| - 1$. Zvolit náhodně číslo $i$ z tohoto intervalu s rovnoměrným rozdělením a pak použít funkci s indexem, jehož kód je $i$. Abychom vybrali $i$, nalezneme nejmenší $j$ takové, že $2^j - 1 \geq |I| - 1$. Pak čísla v intervalu $\{0, 1, \ldots, 2^j - 1\}$ jednoznačně korespondují s posloupnostmi 0 a 1 délky $j$. Budeme vybírat náhodně posloupnost 0 a 1 délky $j$. Když takto vybraná posloupnost neodpovídá prvku z $I$, tak vygenerujeme jinou posloupnost (a tuto vynecháme). Pokud použijeme náhodný generátor 0 a 1, pak takto získáme náhodný prvek z $I$. Tedy k výběru náhodné funkce potřebujeme náhodný generátor 0 a 1 s rovnoměrným rozdělením.

Závada: Skutečný náhodný generátor pro rovnoměrné rozdělení je prakticky nedosažitelný (některé fyzikální procesy). K dispozici je pouze pseudogenerátor.

Jeho nevýhoda: Čím je $j$ větší, tím je posloupnost pravidelnější (tj. méně náhodná).

Důsledky: Hledáme co nejmenší $c$-univerzální systémy. Nejprve ale nalezneme dolní odhady na jejich velikost obecně. (Velikostí je zde stále myšleno počet funkcí v $|H|$, vše ostatní – $c, |S|, m$ – je fixováno!)

### 2.15.8 Dolní odhady na velikost

Předpoklady: Nechť $U$ je univerzum velikosti $N$ a nechť $H = \{h_i \mid i \in I\}$ je $c$-univerzální systém funkcí hašujících do tabulky velikosti $m$. Můžeme předpokládat, že

$$I = \{0, 1, \ldots, |I| - 1\}.$$

**Věta.** *Když $H = \{h_i \mid i \in I\}$ je $c$-univerzální systém pro univerzum $U$ o velikosti $N$ hašující do tabulky s $m$ řádky, pak*

$$|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1).$$

*Důkaz.* Indukcí definujme množiny $U_0, U_1, \ldots$ tak, že: $U_0 = U$.
Nechť $U_1$ je největší podmnožina $U_0$ vzhledem k počtu prvků taková, že $h_0(U_1)$ je jednoprvková množina.

Nechť $U_2$ je největší podmnožina $U_1$ vzhledem k počtu prvků taková, že $h_1(U_2)$ je jednoprvková množina. Nechť $U_3$ je největší podmnožina $U_2$ vzhledem k počtu prvků taková, že $h_2(U_3)$ je jednoprvková množina. Obecně, nechť $U_i$ je největší podmnožina $U_{i-1}$ vzhledem k počtu prvků taková, že $h_{i-1}(U_i)$ je jednoprvková množina.

Protože hašujeme do tabulky velikosti $m$, platí $|U_i| \geq \lceil \frac{|U_{i-1}|}{m} \rceil$. Protože $|U_0| = N$, dostáváme indukcí, že $|U_i| \geq \lceil \frac{N}{m^i} \rceil$ pro každé $i$. Zvolme $i = \lceil \log_m N \rceil - 1$. Pak $i$ je největší přirozené číslo takové, že $\frac{N}{m^i} > 1$. Tedy $U_i$ má aspoň dva prvky, zvolme $x, y \in U_i$ taková, že $x \neq y$. Pak $h_j(x) = h_j(y)$ pro $j = 0, 1, \ldots, i-1$. Tedy

$$i \leq |\{j \in I \mid h_j(x) = h_j(y)\}| \leq \frac{c|I|}{m}.$$

□

---

Když $H = \{h_i \mid i \in I\}$ je $c$-univerzální systém pro univerzum $U$ o velikosti $N$ hašující do tabulky s $m$ řádky, pak $|I| \geq \frac{m}{c}(\lceil \log_m N \rceil - 1)$.

**Pozorování.** *Posloupnosti* 0 *a* 1 *při náhodné volbě* $i$ *z* $I$ *musí mít délku alespoň* $\lceil (\log m - \log c + \log\log N - \log\log m) \rceil$ *(zde všechny logaritmy jsou o základu* 2*).*

### 2.15.9 Malý univerzální systém - definice

Zkonstruujeme $c$-univerzální systém takový, že logaritmus z velikosti jeho indexové množiny pro velká univerza je až na aditivní konstantu menší než $4(\log m + \log\log N)$, kde $N$ je velikost univerza a $m$ je počet řádků v tabulce.

Mějme velikost tabulky $m$ a univerzum $U = \{0, 1, \ldots, N-1\}$ pro nějaké přirozené číslo $N$ (nemusí být prvočíslo).

**Definice.** *Nechť* $p_1, p_2, \ldots$ *je rostoucí posloupnost všech prvočísel.*

**Definice.** *Nechť* $t$ *je nejmenší číslo takové, že* $t \ln p_t \geq m \ln N$.

**Lemma.** $t < m \ln N$, *když* $p_t > 3$.

*Důkaz.* Pokud $p_t > 3$, je $\ln p_t \geq 1$. □

---

**Definice.** *Definujme*

$$H_1 = \{g_{c,d}(h_\ell) \mid t < \ell \leq 2t,\ c, d \in \{0, 1, \ldots, p_{2t} - 1\}\},$$

*kde* $h_\ell(x) = x \bmod p_\ell$ *a* $g_{c,d}(x) = ((cx + d) \bmod p_{2t}) \bmod m$.

V další sekci ukážeme, že když $m(\ln m + \ln\ln m) < N$, pak $H_1$ je 3.25-univerzální systém. Nejdříve ale ukážeme, že indexová množina je dostatečně malá.

Připomeneme si známou větu o velikosti prvočísel, bez důkazu (zde ln je přirozený logaritmus, tj. o základu $e$).

**Věta.** *Pro každé* $i = 1, 2, \ldots$ *platí* $p_i > i \ln i$ *a pro* $i \geq 6$ *platí* $p_i < i(\ln i + \ln\ln i)$. □

**Pozorování.** *Pro $i \geq 6$ platí $p_i < 2i \ln i$. (z dosazení do věty o velikosti prvočísel)*

**Věta.** $|I| < 16m^4 \log^4 N$ *pro dostatečně velké* $t$.

*Důkaz.* (pozn. studenta - díky logaritmům je tento důkaz dost nepřehledný :( logaritmy jsou dvojkové)

Indexová množina $H_1$ je

$$I = \{(c, d, \ell) \mid c, d \in \{0, 1, \ldots, p_{2t} - 1,\ t < \ell \leq 2t\}.$$

Tedy $|I| = tp_{2t}^2$. Odtud plyne $|I| \leq 16t^3 \ln^2 2t$ a tedy

$$\log(|I|) \leq 4 + 3 \log t + 2 \log \log t.$$

Pro dostatečně velké $t$ (takové, že $\log t \geq 2 \log \log t$, tj. $t \geq 16$)[4] platí, že $\log(|I|) \leq 4 + 4 \log t$.

$t \leq m \ln N$, když $p_t \geq 3$ (viz výše).

Po dosazení $\log(|I|) \leq 4 + 4(\log m + \log \log N)$, což už upravíme na požadovanou nerovnost. □

---

**Pozorování.** *Logaritmus z důkazu, tj.* $\log(|I|) \leq 4+4(\log m+\log \log N)$*, nám dává horní omezení velikosti binárního zápisu funkce, tj. počet nutných náhodných výběrů.*

### 2.15.10 Univerzalita malého systému $H_1$

Zvolme různá $x$ a $y$ z univerza $U$.

**Definice.** *Označme*

$$G_1 = \{(c, d, \ell) \mid g_{c,d}(h_\ell(x)) = g_{c,d}(h_\ell(y)),\ h_\ell(x) \neq h_\ell(y)\},$$
$$G_2 = \{(c, d, \ell) \mid g_{c,d}(h_\ell(x)) = g_{c,d}(h_\ell(y)),\ h_\ell(x) = h_\ell(y)\}$$

Budeme odhadovat $G_1, G_2$.

**Věta.** $G_1 \leq \frac{|I|}{m}(1 + \frac{m}{p_{2t}})^2$

*Důkaz.* Použijeme podobný trik, jako v kapitole s důkazem existence univerzálních systémů.

Když $(c, d, \ell) \in G_1$, pak existují $r, s \in \{0, 1, \ldots, \lceil \frac{p_{2t}}{m} \rceil - 1\}$ a $i \in \{0, 1, \ldots, m - 1\}$ taková, že

$$(c(x \bmod p_\ell) + d \equiv i + rm) \bmod p_{2t}$$
$$(c(y \bmod p_\ell) + d \equiv i + sm) \bmod p_{2t}.$$

Když $c$ a $d$ považujeme za neznámé, pak je to soustava lineárních rovnic s regulární maticí (protože $x \bmod p_\ell \neq y \bmod p_\ell$), a tedy pro každé $\ell$, $i$, $r$ a $s$ existuje právě jedna taková dvojice $(c, d)$ (připomínáme, že $\mathbb{Z}/ \bmod p_{2t}$ je těleso). Proto

$$|G_1| \leq tm(\lceil \frac{p_{2t}}{m} \rceil)^2 \leq \frac{tp_{2t}^2}{m}(1 + \frac{m}{p_{2t}})^2 = \frac{|I|}{m}(1 + \frac{m}{p_{2t}})^2.$$

□

[4] Nejsem si jist, proč platí, ale asi ano

---

**Věta.** $G_2 \leq \frac{|I|}{m}$

*Důkaz.* Označme $L = \{\ell \mid t < \ell \leq 2t, x \bmod p_\ell = y \bmod p_\ell\}$ a $P = \prod_{\ell \in L} p_\ell$. Protože $P$ dělí $|x - y|$, dostáváme, že $P \leq N$. Protože $p_t < p_\ell$ pro každé $\ell \in L$, dostáváme, že $P > p_t^{|L|}$. Tedy $|L| \leq \frac{\ln N}{\ln p_t} \leq \frac{t}{m}$ z definice $t$. Protože $(c, d, \ell) \in G_2$, právě když $\ell \in L$ a $c, d \in \{0, 1, \ldots, p_{2t} - 1\}$, shrneme, že

$$|G_2| \leq |L| p_{2t}^2 \leq \frac{t p_{2t}^2}{m} = \frac{|I|}{m}.$$

□

---

**Lemma** (Pomocné lemma)**.** *Když $t \geq 6$ a $m(\ln m + \ln \ln m) < N$, pak $m < \frac{p_t}{\ln t}$.*

*Důkaz.* Předpokládejme, že tvrzení neplatí. Pak $m \geq \frac{p_t}{\ln t}$. Z Věty o velikosti prvočísel plyne $m \geq \frac{p_t}{\ln t} > \frac{t \ln t}{\ln t} = t$. Když použijeme, že $m(\ln m + \ln \ln m) < N$, tak dostaneme, že

$$\ln m + \ln(\ln m + \ln \ln m) < \ln N,$$

a odtud plyne, že

$$t \ln p_t < t \ln(t(\ln t + \ln \ln t)) \leq m(\ln m + \ln(\ln m + \ln \ln m)) < m \ln N$$

a to je spor s definicí $t$. Tedy $m < \frac{p_t}{\ln t}$. □

---

**Pozorování.** $\ln 2t \geq \ln t \geq \ln \ln t \qquad$ *pro všechna $t \geq 1$*

**Lemma.** *$\frac{m}{p_{2t}}$ je menší, než $\frac{1}{2}$, a pokud $t$ konverguje k $+\infty$, tak konverguje k 0.*

*Důkaz.* Zkombinujeme Větu o odhadu velikosti prvočísel, Pomocné lemma a předchozí pozorování a dostaneme, že

$$\frac{m}{p_{2t}} \leq \frac{\frac{p_t}{\ln t}}{2t \ln 2t} < \frac{t(\ln t + \ln \ln t)}{2t \ln t \ln 2t} < \frac{1}{\ln 2t}(1 + \frac{\ln \ln t}{\ln t}).$$

Je zřejmé, že tento výraz je menší než $\frac{1}{2}$, a když $t$ konverguje k $+\infty$, pak tento výraz konverguje k 0. □

---

**Lemma.** $(1 + \frac{m}{p_{2t}})^2 \leq 1.5^2 = 2.25$

*Důkaz.* Plyne jednoduše z přechozího lemmatu. □

---

**Věta.** *$H_1$ je 3.25-univerzální*

*Důkaz.* Z předchozího plyne:

$$|\{i \in I \mid h_i(x) = h_i(y)\}| = |G_1| + |G_2| \leq$$
$$\frac{|I|}{m}(1 + \frac{m}{p_{2t}})^2 + \frac{|I|}{m} \leq \frac{|I|}{m}(1 + 2.25) = 3.25\frac{|I|}{m}.$$

□

---

Když $t \geq 6$ a $m \ln m \ln \ln m < N$, pak $H_1$ je 3.25-univerzální.

Bez jakýchkoliv předpokladů lze ukázat, že $H_1$ je 5-univerzální.

### 2.15.11 Odhad na velikost $c$

**Lemma** (Technické lemma)**.** *Mějme reálná čísla $b_i$ pro $i = 0, 1, \ldots, m-1$ a nechť $b = \sum_{i=0}^{m-1} b_i$. Pak*

$$\sum_{i=0}^{m-1} b_i(b_i - 1) \geq b(\frac{b}{m} - 1).$$

*Důkaz.* Z Cauchyho-Schwarzovy nerovnosti

$$(\sum_{i=0}^{m-1} x_i y_i)^2 \leq (\sum_{i=0}^{m-1} x_i^2)(\sum_{i=0}^{m-1} y_i^2)$$

plyne $(\sum_{i=0}^{m-1} b_i)^2 = b^2 \leq m(\sum_{i=0}^{m-1} b_i^2)$, stačí položit $x_i = b_i$ a $y_i = 1$, a tedy $\frac{b^2}{m} \leq \sum_{i=0}^{m-1} b_i^2$. Odtud

$$\sum_{i=0}^{m-1} b_i(b_i - 1) = \sum_{i=0}^{m-1} b_i^2 - \sum_{i=0}^{m-1} b_i = \sum_{i=0}^{m-1} b_i^2 - b \geq \frac{b^2}{m} - b = b(\frac{b}{m} - 1)$$

a lemma je dokázáno. □

---

**Lemma** (O obecné funkci)**.** *Pokud $f : U \to T$ je libovolná hašovací funkce (tj. ne z $H$), počet dvojic $u, v$ takových, že $u \neq v$ a $f(u) = f(v)$ (tj. kolidující dvojice) je větší, než $N(\frac{N-m}{m})$, kde $N$ je velikost $U$ a $m$ je velikost $T$.*

*Důkaz.* Když pro $t \in T$ označíme $k_t = |f^{-1}(t)|$, pak $|A| = \sum_{t \in T} k_t(k_t - 1)$. Z lemmatu plyne, že

$$|A| = \sum_{t \in T} k_t(k_t - 1) \geq N(\frac{N}{m} - 1) = N(\frac{N-m}{m}),$$

protože $\sum_{t \in T} k_t = N$.

Nerovnítko plyne z technického lemmatu. □

---

**Věta.** *Když $H$ je $c$-univerzální systém univerza $U$ o velikosti $N$ hašující do tabulky s $m$ řádky, pak $c \geq 1 - \frac{m}{N}$.*

*Důkaz.* Použijeme předchozí lemma o obecné funkci a při sčítání si „vtipně“ přehodíme, přes co vlastně sčítáme, a poté využijeme toho, že $H$ je $c$-univerzální.

Když $H = \{h_i \mid i \in I\}$ je $c$-univerzální systém funkcí z univerza $U$ o velikosti $N$ do tabulky o velikosti $m$, pak pomocí lemmatu dostáváme

$$\begin{aligned}
&|I|N(\frac{N-m}{m}) \leq \\
&\sum_{i \in I} |\{(x,y) \in U \times U \mid h_i(x) = h_i(y),\, x \neq y\}| = \\
&\sum_{(x,y) \in U \times U,\, x \neq y} |\{i \in I \mid h_i(x) = h_i(y)\}| \leq \\
&\sum_{(x,y) \in U \times U, x \neq y} c\frac{|I|}{m} = N(N-1)c\frac{|I|}{m}.
\end{aligned}$$

Odtud plyne, že $N - m \leq c(N-1)$, a tedy

$$c \geq \frac{N-m}{N-1} > \frac{N-m}{N} = 1 - \frac{m}{N}.$$

□

---

Když $H$ je $c$-univerzální, $c \geq 1 - \frac{m}{N}$.

### 2.15.12 Problémy univerzálního hašování

Použít jiné metody na řešení kolizí než separované řetězce. Jak to ovlivní použitelnost univerzálního hašování? Platí podobné vztahy jako pro pevně danou hašovací funkci? Jaký vliv na efektivnost má nepřítomnost operace **DELETE**?

Existuje $c$-univerzální hašovací systém pro $c < 1$? Jaký je vztah mezi velikostí $c$-univerzálního hašovacího systému a velikostí $c$? Lze zkonstruovat malý $c$-univerzální systém pro $c < 3.25$? Zde hraje roli fakt, že při $c = 3.25$ se očekávaná délka řetězce může pohybovat až kolem hodnoty 7.

Použití Čebyševovy nerovnosti místo Markovovy nerovnosti dává kvadratický odhad pravděpodobnosti, že délka řetězce je o $t$ větší než očekávaná hodnota. Za jakých okolností dává lepší odhad? Lze použít i vyšších momentů?

Jak použít Markovou nerovnost a očekávanou délku maximálního řetězce pro odhad očekávaného počtu voleb hašovací funkce?

Pro jaké parametry lze použít následující model?[5]

Je dána základní velikost tabulky $m$ a dále pro $j = 0, 1, \ldots$ čísla (parametry) $l_j$ a $c$-univerzální hašovací systémy $H_j = \{h_i \mid i \in I_j\}$ z univerza do tabulky s $m2^j$ řádky.

[5] pozn.studenta - vůbec nevím, o co jde :(

Množina $S \subseteq U$ je reprezentována následovně: je dáno $j$ takové, že když $j > 0$, pak $m2^{j-2} \leq |S| \leq m2^j$, když $j = 0$, pak $|S| \leq m$, a je zvolen index $i \in I_j$. Dále máme prosté řetězce $r_0, r_1, \ldots, r_{m2^j-1}$, jejichž délky jsou nejvýše $l_j$, a řetězec $r_k$ obsahuje prvky $\{s \in S \mid h_i(s) = k\}$.
Operace **INSERT**$(x)$ prohledá řetězec $r_{h_i(x)}$ a když tento řetězec neobsahuje prvek $x$, pak ho přidá. Když $m2^{j-2} \leq |S| \leq m2^j$ a délka řetězce $r_{h_i}(x)$ je nejvýše $l_j$, pak operace končí. Když $|S| > m2^j$, tak se nejdříve zvětší $j$ o 1. Pak se náhodně zvolí $i \in I_j$ a zkonstruují se řetězce reprezentující $S$. Když některý z nich má délku větší než $l_j$, tak se volba a konstrukce řetězců opakuje tak dlouho, dokud se nepovede zvolit $i \in I_j$ takové, že všechny zkonstruované řetězce mají délku nejvýše $l_j$. Operace **DELETE** se řeší analogicky.
Problém: Jak volit parametry $l_i$?

V případě řešení kolizí dvojitým hašováním nebo hašováním s lineárním přidáváním je třeba dát silnější podmínky na velikost $|S|$. V poslední době se této tématice věnuje pozornost a byla dosažena řada zajímavých výsledků.

## 2.16 Perfektní hašování

### 2.16.1 Idea

Jde o další řešení kolizí. Idea je nalézt pro *předem danou* množinu hašovací funkci, která nevytváří *žádné* kolize.

Nevýhoda: Metoda nepřipouští operaci **INSERT** (pro nový vstup nemůžeme zaručit, že nevznikne kolize). Metodu lze prakticky použít pro úlohy, kde lze očekávat hodně operací **MEMBER** a operace **INSERT** se téměř nevyskytuje (kolize se řeší pomocí malé pomocné tabulky, kam se ukládají kolidující data). Tato metoda se používá při navrhování kompilátorů.

### 2.16.2 Požadavky

Pro danou množinu $S \subseteq U$ chceme nalézt hašovací funkci $h$ takovou, že

1. pro $s, t \in S$ takové, že $s \neq t$, platí $h(s) \neq h(t)$ (tj. $h$ je *perfektní hašovací funkce* pro $S$);
2. $h$ hašuje do tabulky s $m$ řádky, kde $m$ je přibližně stejně velké jako $|S|$ (není praktické hašovat do příliš velkých tabulek – ztrácí se jeden ze základních důvodů pro hašování);
3. $h$ musí být rychle spočitatelná – jinak hašování není rychlé;
4. uložení $h$ nesmí vyžadovat moc paměti, nejvýhodnější je analytické zadání (když zadání $h$ bude vyžadovat moc paměti, např. když by byla dána tabulkou, pak se ztrácí důvod k použití stejně jako v bodě 2).

Kompenzace: Nalezení hašovací funkce může spotřebovat více času. Provádí se jen na začátku úlohy.

### 2.16.3 $(N, m, n)$-perfektní systém - definice

Mějme univerzum $U = \{0, 1, \ldots, N-1\}$.

**Definice.** *Soubor funkcí $H$ z $U$ do množiny $\{0, 1, \ldots, m-1\}$ se nazývá $(N, m, n)$-*perfektní*, když pro každou $S \subseteq U$ takovou, že $|S| = n$, existuje $h \in H$ perfektní pro $S$ (tj. $h(s) \neq h(t)$ pro každá dvě různá $s, t \in S$).*

Protože nevíme, zda taková $h$ existují, nejprve vyšetříme množiny perfektních hašovacích funkcí. Vyšetříme vlastnosti $(N, m, n)$-perfektních souborů funkcí.

### 2.16.4 Dolní odhady na velikost $(N, m, n)$-perfektního souboru

Předpokládejme, že $H$ je $(N, m, n)$-perfektní systém pro $U = \{0, 1, \ldots, N-1\}$ a nejprve nalezneme dolní odhady na velikost $|H|$.

**Lemma.** *Libovolná funkce $h$ z $U$ do množiny $\{0, 1, \ldots, m-1\}$ je perfektní pro maximálně $\binom{m}{n}(\frac{N}{m})^n$ množin.*

*Důkaz.* Ještě jednou – zjišťujeme počet množin $S \subseteq U$ takových, že $h$ je perfektní funkce pro $S$ a $|S| = n$.

Funkce $h$ je perfektní pro $S \subseteq U$, právě když pro každé $i = 0, 1, \ldots, m-1$ je $|h^{-1}(i) \cap S| \leq 1$. (pokud by bylo větší, nebyla by perfektní)

Odtud počet těchto množin je

$$\sum_{0 \leq i_0 < i_1 < \cdots < i_{n-1} < m} \prod_{j=0}^{n-1} |h^{-1}(i_j)|$$

Vysvětlení: vzali jsme si všechny možné podmnožiny $m$, velké $n$ – tahle množina nám říká, na kterých místech výsledné tabulky je něco zahešováno – a reprezentovali jsme si ji přes rostoucí posloupnost. Pro každé místo v tabulce, kde je něco zahešováno, jsem si vzal všechny možnosti, co tam můžou být (to je $|h^{-1}(i_j)|$).

Jinak řečeno, $h(S) = \{i_j \mid j = 0, 1, \ldots, n-1\}$.

Hledáme horní odhad této sumy; je maximální, když $|h^{-1}(i)| = \frac{N}{m}$ pro každé $i$. Posloupností $i$ je $\binom{m}{n}$, tedy $h$ může být perfektní nejvýše pro $\binom{m}{n}(\frac{N}{m})^n$ množin. □

---

**Věta.**

$$|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}(\frac{N}{m})^n}.$$

*Důkaz.* Víme, že $n$-prvkových podmnožin universa je $\binom{N}{n}$, a každá z funkcí v $H$ je perfektní pro maximálně $\binom{m}{n}(\frac{N}{m})^n$ množin, tedy skutečně $|H| \geq \frac{\binom{N}{n}}{\binom{m}{n}(\frac{N}{m})^n}$. □

---

Jiný odhad velikosti $(N, m, n)$-perfektního souboru.

**Věta.**

$$|H| \geq \frac{\log N}{\log m}$$

*Důkaz.* Velikost souboru funkcí nazvěme $t$, $H = \{h_1, \ldots, h_t\}$.

Definujme indukcí soubor množin $U_i$:

- $U_0 = U$

- pro $i > 0$ je $U_i$ největší podmnožina $U_{i-1}$, co do počtu prvků, taková, že $h_i$ je konstantní na $U_i$.

Pak $|U_i| \geq \frac{|U_{i-1}|}{m}$ pro všechna $i > 0$; z $|U_0| = N$ plyne $|U_i| \geq \frac{N}{m^i}$.

Pro každé $i = 1, 2, \ldots, t$ je $h_j(U_i)$ jednobodová množina pro každé $j \leq i$ (z definice množin $U$ – je tam konstantní).

Tedy, jakmile bychom pro nějaké $i$ měli množinu $S$, pro kterou $|S \cap U_i| \geq 2$, tak žádné $j \leq i$ není perfektní (protože pro tyto dva prvky by byla shodná).

Protože $H$ je $(N, m, n)$-perfektní, musí být $|U_t| \leq 1$ (vzali-li bychom si množinu, co by v sobě měla více, než 2 prvky z $U_t$, ani jedna z funkcí by tam nemohla být perfektní), a tedy $\frac{N}{m^t} \leq 1$.

Proto $t \geq \frac{\log N}{\log m}$. □

---

Z obou vět potom platí:

Když $H$ je $(N, m, n)$-perfektní soubor funkcí, pak

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

### 2.16.5 Existence $(N, m, n)$-perfektního souboru

Mějme univerzum $U = \{0, 1, \ldots, N-1\}$ a soubor funkcí $H = \{h_1, h_2, \ldots, h_t\}$ z univerza $U$ do množiny $\{0, 1, \ldots, m-1\}$. (funkce jsou libovolné)

**Definice.** *Reprezentujeme tento soubor pomocí matice $M(H)$ typu $N \times t$ s hodnotami $\{0, 1, \ldots, m-1\}$ tak, že pro $x \in U$ a $i = 1, 2, \ldots, t$ je v $x$-tém řádku a $i$-tém sloupci matice $M(H)$ hodnota $h_i(x)$.*

(Jedna matice je tedy rovna jednomu souboru funkcí. Pokud mám jedinou množinu, pro kterou neexistuje perfektní funkce, už to není $(N, m, n)$-perfektní systém.)

**Lemma.** *Pro pevnou množinu $S = \{s_1, s_2, \ldots, s_n\} \subseteq U$ je matic* bez *perfektní funkce nejvýše*

$$(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t}.$$

*Důkaz.* Žádná funkce z $H$ není perfektní pro množinu $S = \{s_1, s_2, \ldots, s_n\} \subseteq U$, právě když podmatice $M(H)$ tvořená řádky $s_1$, $s_2$, až $s_n$ a všemi sloupci nemá prostý sloupec. Takových matic je nejvýše

$$(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t}.$$

Vysvětlení: $m^n$ je počet všech funkcí z $S$ do $\{0, 1, \ldots, m-1\}$, $\prod_{i=0}^{n-1}(m-i)$ je počet prostých funkcí z $S$ do $\{0, 1, \ldots, m-1\}$, a tedy počet všech podmatic s $n$ řádky takových, že žádný jejich sloupec není prostý, je $(m^n - \prod_{i=0}^{n-1}(m-i))^t$. Tyto podmatice můžeme libovolně doplnit na matici typu $N \times n$ a pro každou matici je těchto doplnění $m^{(N-n)t}$. □

---

**Lemma.** *Počet matic, které nereprezentují $(N, m, n)$-perfektní systém, je menší nebo roven*

$$\binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t}.$$

*Důkaz.* Podmnožin $U$ velikosti $n$ je $\binom{N}{n}$, tedy počet všech matic, které nereprezentují $(N, m, n)$-perfektní systém, je menší nebo roven $\binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t}$ (podle posledního lemmatu) □

---

**Lemma** (Postačující podmínka)**.** *Když*

$$\binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t} < m^{Nt},$$

*pak nutně existuje $(N, m, n)$-perfektní systém.*

*Důkaz.* Všech matic je $m^{Nt}$, a tedy když $\binom{N}{n}(m^n - \prod_{i=0}^{n-1}(m-i))^t m^{(N-n)t} < m^{Nt}$, pak nutně existuje $(N, m, n)$-perfektní systém, protože nějaká matice, co ho reprezentuje, se „najde“. □

---

**Věta.** *Pokud $t \geq n(\ln N)e^{\frac{n^2}{m}}$, tak existuje $(N, m, n)$-perfektní soubor funkcí.*

*Důkaz.* Následující výrazy jsou ekvivalentní s postačující podmínkou:

$$\binom{N}{n}(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n})^t < 1 \Leftrightarrow t \geq \frac{\ln\binom{N}{n}}{-\ln(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n})}.$$

Zlomek vpravo odhadneme shora – protože se jedná o postačující podmínku, pokud bude $t$ větší, než tento horní odhad, pak $(N, m, n)$ systém bude existovat.

Čitatel odhadneme shora – $\ln\binom{N}{n} \leq n \ln N$.

Jmenovatel odhadneme zdola – protože $-\ln(1-x) \geq x$ pro $x \in (0, 1)$, dostáváme

$$-\ln(1 - \frac{\prod_{i=0}^{n-1}(m-i)}{m^n}) \geq \prod_{i=0}^{n-1}(1 - \frac{i}{m}) = e^{\sum_{i=0}^{n-1}\ln(1-\frac{i}{m})} \geq e^{\int_0^n \ln(1-\frac{x}{m})dx},$$

kde integrál můžeme odhadnout

$$m[(1-\frac{n}{m})(1-\ln(1-\frac{n}{m})) - 1] \geq m[(1-\frac{n}{m})(1+\frac{n}{m}) - 1] = -\frac{n^2}{m}.$$

Horní odhad zlomku je tedy $n(\ln N)e^{\frac{n^2}{m}}$.

Odtud dostáváme, že když $t \geq n(\ln N)e^{\frac{n^2}{m}}$, pak platí postačující podmínka, a tedy existuje $(N, m, n)$-perfektní soubor funkcí. □

---

Existence $(N, m, n)$-perfektního souboru funkcí ale nezaručuje splnění požadavků 2), 3) a 4) ze sekce 2.16.2.

Abychom uspěli, použijeme ideu z metody univerzálního hašování.

Pozn. studenta – v následujících několika kapitolách jsem si dovolil různé funkce, které stavíme, nazvat písmeny $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}$, abych v nich sám měl pořádek. Původně šlo o jednu kapitolu, ale ztrácel jsem se v tom.

### 2.16.6 Konstrukce perfektních hašovacích funkcí $\mathbf{A}, \mathbf{B}$

Předpoklady: $U = \{0, 1, \ldots, N-1\}$, kde $N$ je prvočíslo. Mějme pevné $S \subseteq U$ o velikosti $n$.

**Definice.**

$$h_k(x) = (kx \bmod N) \bmod m \qquad \text{pro } k = 1, 2, \ldots, N-1.$$

**Definice.** *Pro $i = 0, 1, \ldots, m-1$ a $k = 1, 2, \ldots, N-1$ označme*

$$b_i^k = |\{x \in S \mid (kx \bmod N) \bmod m = i\}|.$$

Význam $b_i^k$: Hodnoty $b_i^k$ lze považovat za veličiny, které ukazují odchylku od perfektnosti. Říkají, kolik prvků koliduje v $k$-té funkci do $i$-tého slotu.

**Pozorování.**

$$\text{když } b_i^k \geq 2, \text{ pak } (b_i^k)^2 - b_i^k \geq 2,$$

*protože $a^2 - a \geq 2$, když $a \geq 2$. Na druhou stranu*

$$b_i^k \leq 1 \text{ implikuje } (b_i^k)^2 - b_i^k = 0.$$

**Lemma** (Podmínka perfektnosti)**.** *Funkce $h_k$ je perfektní, právě když $\sum_{i=0}^{m-1} (b_i^k)^2 - n < 2$.*

*Důkaz.* Plyne z $\sum_{i=0}^{m-1} b_i^k = n$. □

---

**Lemma.** *Existuje $k$ takové, že $\sum_{i=0}^{m-1} (b_i^k)^2 \leq 2\frac{n(n-1)}{m} + n$.*

*Důkaz.* Odhadneme výraz $\sum_{k=1}^{N-1} \big((\sum_{i=0}^{m-1} (b_i^k)^2) - n\big)$.

$$\begin{aligned}
&\sum_{k=1}^{N-1} \big((\sum_{i=0}^{m-1} (b_i^k)^2) - n\big) = \\
&\quad \sum_{k=1}^{N-1} \big((\sum_{i=0}^{m-1} |\{x \in S \mid h_k(x) = i\}|^2) - n\big) = \\
&\quad \sum_{k=1}^{N-1} |\{(x, y) \mid x, y \in S,\, x \neq y,\, h_k(x) = h_k(y)\}| = \\
&\quad \sum_{x,y \in S, x \neq y} |\{k \mid 1 \leq k < N,\, h_k(x) = h_k(y)\}|.
\end{aligned}$$

První rovnítko je z definice; druhé platí proto, že umocněný výraz je počet všech kolidujících dvojic a $n$ je takových, že se rovnají; třetí je opět přehození sumy.

Teď použijeme podobný „trik“ s modulem, jako předtím.

Zvolme $x, y \in S$ taková, že $x \neq y$.

Pak $h_k(x) = h_k(y)$, právě když existuje $i = 0, 1, \ldots, m-1$ a $r, s = 0, 1, \ldots, \lfloor \frac{N}{m} \rfloor$ taková, že

$$(kx \equiv i + rm) \bmod N$$
$$(ky \equiv i + sm) \bmod N$$

a $i + rm, i + sm < N$ ($i$ je opět zbytek po modulu).

Odtud odečtením dostáváme, že $h_k(x) = h_k(y)$ implikuje $kx - ky \equiv (r - s)m \bmod N$.

Protože $0 < k < n$ a $x \neq y$, platí, že $kx - ky \neq 0$; tedy $h_k(x) = h_k(y)$ implikuje existenci

$$(r - s) = q = -\lfloor \frac{N}{m} \rfloor, -\lfloor \frac{N}{m} \rfloor + 1, \ldots, -1, 1, 2, \ldots, \lfloor \frac{N}{m} \rfloor$$

takového, že $k(x - y) = kx - ky \equiv qm \bmod N$.

Nechť např. $q > 0$; pro $x > y$ a pro jedno $q = 1, 2, \ldots, \lfloor \frac{N}{m} \rfloor$ existuje právě jedno $k$ takové, že $k(x - y) \equiv qm \bmod N$, protože $\mathbb{Z}_N$ je těleso (tato rovnice má jediné řešení – řešíme $k$, protože $x, y, q, m, N$ jsou zafixovány).

Naopak pro $q = -\lfloor \frac{N}{m} \rfloor, \ldots, -2, -1$ je rovnice $k(x - y) \equiv qm \bmod N$ ekvivalentní s rovnicí $k(x - y) \equiv N + qm \bmod N$, opět je právě jedno řešení.

Dostáváme, že pro $x, y \in S$, $x > y$, existuje nejvýše $2\lfloor \frac{N}{m} \rfloor = 2\lfloor \frac{N-1}{m} \rfloor$ různých $k = 1, 2, \ldots, N - 1$, že $h_k(x) = h_k(y)$ (jedno $k$ pro každé možné $q$).

Stejný odhad analogicky dostaneme, když $x < y$ (ale dostáváme jiná řešení).

Odtud

$$\sum_{k=1}^{N-1} \big( (\sum_{i=0}^{m-1} (b_i^k)^2) - n \big) \leq \sum_{x,y \in S, x \neq y} 2(\frac{N-1}{m}) = 2(N-1)\frac{n(n-1)}{m}.$$

Tedy platí, že existuje $k$ takové, že $\sum_{i=0}^{m-1} (b_i^k)^2 \leq 2\frac{n(n-1)}{m} + n$. □

---

**Lemma.** *Ukážeme, že existuje více než $\frac{N-1}{4}$ takových $k$, že platí*

$$\sum_{i=0}^{m-1} (b_i^k)^2 < 3\frac{n(n-1)}{m} + n.$$

*Důkaz.* Sporem.

V opačném případě dostáváme, že

$$\sum_{k=1}^{N-1}\left((\sum_{i=0}^{m-1}(b_i^k)^2)-n\right) \geq \frac{3(N-1)}{4}\frac{3n(n-1)}{m} = \frac{9(N-1)n(n-1)}{4m} > \frac{2(N-1)n(n-1)}{m},$$

a to je spor s předchozím výsledkem. Tedy při náhodném rovnoměrném výběru $k$ je

$$\mathrm{Prob}\{\sum_{i=0}^{m-1}(b_i^k)^2 < \frac{3n(n-1)}{m}+n \mid k \in \{1,2,\ldots,N-1\}\} \geq \frac{1}{4}.$$

□

---

**Věta.** *Když $n = m$, pak*

*(a) (nazvu funkce $\mathbf{A_D}$) existuje deterministický algoritmus, jenž v čase $O(nN)$ nalezne $k$ takové, že*

$$\sum_{i=0}^{m-1}(b_i^k)^2 < 3n;$$

*(b) (nazvu funkce $\mathbf{A_N}$) existuje pravděpodobnostní algoritmus, který nalezne v čase $O(n)$ takové $k$, že $\sum_{i=0}^{m-1}(b_i^k)^2 < 4n$ – očekávaný počet iterací výpočtu je nejvýše 4.*

*Dále*

*(c) (nazvu funkce $\mathbf{B_D}$) existuje deterministický algoritmus, jenž v čase $O(nN)$ pro $m = n(n-1)+1$ nalezne takové $k$, že $h_k$ je perfektní;*

*(d) (nazvu funkce $\mathbf{B_N}$) existuje pravděpodobnostní algoritmus, který pro $m = 2n(n-1)$ v čase $O(n)$ nalezne $k$ takové, že $h_k$ je perfektní – očekávaný počet iterací výpočtu je nejvýše 4.*

*Důkaz.* Neformálně:

Časy jsou jednoduché – pro deterministické musíme zkusit všechny možné $k$, pro nedeterministické zkusíme v průměru jen čtyřikrát. Zbytek v podstatě jen dosadíme do předchozích dvou lemmat různé velikosti $m$.

Formálně:

Mějme $n = m$. Protože spočítání $\sum_{i=0}^{m-1}(b_i^k)^2$ pro pevné $k$ vyžaduje čas $O(n)$, prohledáním všech možností nalezneme $k$ takové, že

$$\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{n}+n = 3n-2 < 3n,$$

v čase $O(nN)$. Tím je dokázáno a). Pravděpodobnostní algoritmus dokazující b) volí náhodně $k$ a v čase $O(n)$ ověří, zda $\sum_{i=0}^{m-1}(b_i^k)^2 \leq 3\frac{n(n-1)}{n}+n = 4n-3 < 4n$. Tuto akci opakuje, dokud požadavek není splněn. Protože pravděpodobnost, že $k$ splňuje požadavek, je alespoň $\frac{1}{4}$, tak očekávaný počet iterací akce je nejvýše

$$\sum_{i=0}^{\infty} i(\frac{3}{4})^{i-1}\frac{1}{4} = \frac{1}{4}\frac{1}{(1-\frac{3}{4})^2} = 4$$

a odtud plyne b).

Když $m = n(n-1)+1$, pak prohledáním všech možností nalezneme $k$ takové, že

$$\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{2n(n-1)}{n(n-1)+1} + n < n+2,$$

v čase $O(nN)$ a c) plyne z předchozí věty. Když $m = 2n(n-1)$, pak pro náhodně zvolené $k$ platí s pravděpodobností $\leq \frac{1}{4}$, že

$$\sum_{i=0}^{m-1}(b_i^k)^2 \leq \frac{3n(n-1)}{2n(n-1)} + n < n+2.$$

Algoritmus splňující tvrzení d) je stejný jako v případě b) (jen $m = 2n(n-1)$).

□

---

Hašovací funkce **A** nejsou perfektní.

Hašovací funkce **B** jsou perfektní, ale nesplňují požadavek 2) z 2.16.2 (platí $m = \Theta(n^2)$). Pro ten nalezneme funkci **C**

### 2.16.7 Konstrukce perfektní hašovací funkce C

Neformálně:

Použiji funkce **A**, ty mi budou někde kolidovat. Na každé z kolizních množin pak použiji funkce **B**, každou extra v extra tabulce a pak je všechny dám za sebe. Počet kolizí v **A** je omezen.

Formálně, deterministická verze:

1. Nalezneme $k$ takové, že pro $m = n$ platí $\sum_{i=0}^{m-1}(b_i^k)^2 < 3n$. Pro $i = 0, 1, \ldots, m-1$ nalezneme množiny $S_i = \{s \in S \mid h_k(s) = i\}$
2. Pro každé $i = 0, 1 \ldots, m-1$ takové, že $S_i \neq \emptyset$, nalezneme pro $m = 1 + |S_i|(|S_i|-1)$ takové $k_i$, že $h_{k_i}$ je perfektní na $S_i$. Definujme $c_i = 1 + |S_i|(|S_i|-1)$, když $S_i \neq \emptyset$, a $c_i = 0$, když $S_i = \emptyset$.
3. Pro $i = 0, 1, \ldots, m$ definujme $d_i = \sum_{j=0}^{i-1} c_j$ a pro $x \in U$ označme $h_k(x) = l$. Pak položíme $g(x) = d_l + h_{k_l}(x)$.

Formálně, nedeterministická verze, rozdíly podtrženy:

1. Nalezneme $k$ takové, že pro $m = n$ platí $\underline{\sum_{i=0}^{m-1}(b_i^k)^2 < 4n}$. Pro $i = 0, 1, \ldots, m-1$ nalezneme množiny $S_i = \{s \in S \mid h_k(s) = i\}$
2. Pro každé $i = 0, 1 \ldots, m-1$ takové, že $S_i \neq \emptyset$, nalezneme pro $\underline{m = 1 + 2|S_i|(|S_i|-1)}$ takové $k_i$, že $h_{k_i}$ je perfektní na $S_i$. Definujme $\underline{c_i = 2|S_i|(|S_i|-1)}$, když $S_i \neq \emptyset$, a $c_i = 0$, když $S_i = \emptyset$.
3. Pro $i = 0, 1, \ldots, m$ definujme $d_i = \sum_{j=0}^{i-1} c_j$ a pro $x \in U$ označme $h_k(x) = l$. Pak položíme $g(x) = d_l + h_{k_l}(x)$.

**Věta.** • *Zkonstruovaná funkce $g$ je perfektní.*
- *Hodnota $g(x)$ se pro každé $x \in U$ spočítá v čase $O(1)$.*
- *V deterministickém případě hašuje do tabulky velikosti $< 3n$ a je nalezena v čase $O(nN)$, v pravděpodobnostním případě hašuje do tabulky velikosti $< 6n$ a je nalezena v čase $O(n)$.*
- *Pro její zakódování jsou třeba hodnoty $k$ a $k_i$ pro $i = 0, 1, \ldots, m-1$. Tyto hodnoty jsou v rozmezí $1, 2, \ldots, N-1$, a tedy vyžadují $O(n \log N)$ paměti.*

*Důkaz.* • Protože $g(S_i)$ pro $i = 0, 1, \ldots, m-1$ jsou navzájem disjunktní a $h_{k_i}$ je perfektní na $S_i$, dostáváme, že $g$ je perfektní.
- Pro výpočet hodnoty $g(x)$ jsou třeba dvě násobení, dvojí výpočet zbytku při dělení a jedno sčítání (hodnoty $d_i$ jsou uloženy v paměti). Proto výpočet $g(x)$ vyžaduje čas $O(1)$.
- Dále $d_m$ je horní odhad na počet řádků v tabulce. Protože pro $S_i \neq \emptyset$ máme $|S_i|(|S_i|-1)+1 \leq |S_i|^2 = (b_i^k)^2$, dostáváme v deterministickém případě $d_m = \sum_{i=0}^{m-1} c_i \leq \sum_{i=0}^{m-1} (b_i^k)^2 < 3n$ a $k$ nalezneme v čase $O(nN)$. Protože $k_i$ nalezneme v čase $O(|S_i|N)$, lze $g$ zkonstruovat v čase $O(nN + \sum_{i=0}^{m-1} |S_i|N) = O(nN + N\sum_{i=0}^{m-1} |S_i|) = O(2nN) = O(nN)$. V pravděpodobnostním případě je

$$d_m = \sum_{i=0}^{m-1} c_i \leq \sum_{i=0}^{m-1} (2|S_i|^2 - 2|S_i|) = 2\sum_{i=0}^{m-1} (b_i^k)^2 - 2\sum_{i=0}^{m-1} b_i^k < 8n - 2n = 6n$$

  (protože $|S_i| = b_i^k$ a $\sum_{i=0}^{m-1} b_i^k = n$).
- Protože $k$ nalezneme v čase $O(n)$ a $k_i$ v čase $O(|S_i|)$, dostaneme, že že $g$ nalezneme v čase $O(n)$.
- Zbytek je jasný.

□

---

Tedy zkonstruovaná hašovací funkce splňuje požadavky 1), 2) a 3) z 2.16.2, ale požadavek 4) není splněn.

### 2.16.8 Konstrukce perfektní hašovací funkce D

**Lemma.** *Nechť $q$ =počet prvočísel, která dělí $m$. Pak $q = O(\frac{\log m}{\log \log m})$.*

*Důkaz.* Mějme přirozené číslo $m$ a nechť $q$ je počet všech prvočísel dělících $m$ ($p_1, p_2, \ldots$ je rostoucí posloupnost všech prvočísel). Pak

$$m \geq \prod_{i=1}^{q} p_i > q! = e^{\sum_{i=1}^{q} \ln i} \geq e^{\int_1^q \ln x dx} = e^{q\ln(\frac{q}{e})+1} \geq (\frac{q}{e})^q.$$

Proto existuje konstanta $c$, že $q \leq c\frac{\ln m}{\ln \ln m}$ (viz Pomocné lemma v sekci 2.3.6). □

---

**Věta.** *Pro každou $n$-prvkovou množinu $S \subseteq U$ existuje prvočíslo $p$ o velikosti $O(n^2 \ln N)$ takové, že funkce $\phi_p(x) = x \bmod p$ je perfektní pro $S$. (nazvu funkce* **D***)*

*Důkaz.* Mějme opět pevně danou $S = \{s_1 < s_2 < \cdots < s_n\} \subseteq U$.

Označme $d_{i,j} = s_j - s_i$ pro $1 \leq i < j \leq n$. Pak $s_i \bmod p \neq s_j \bmod p$, právě když $d_{i,j} \neq 0 \bmod p$.

Označme $D = \prod_{1 \leq i < j \leq n} d_{i,j} \leq N^{(n^2)}$.

Pak počet prvočíselných dělitelů čísla $D$ je nejvýše $c\frac{\ln D}{\ln \ln D}$ (z minulého lemmatu).

Tedy mezi prvními $1 + c\frac{\ln D}{\ln \ln D}$ prvočísly existuje prvočíslo $p$ takové, že $s_i \bmod p \neq s_j \bmod p$ pro každé $1 \leq i < j \leq n$. Existuje proto, že jsme vzali o 1 větší než největší možné prvočíslo, které to dělí; nerovná se proto, že to nesmí dělit ani jeden rozdíl.

To znamená, že funkce $\phi_p(x) = x \bmod p$ je perfektní pro $S$.

Podle věty o velikosti prvočísel $p_t \leq 2t \ln t$ pro každé $t \geq 6$, tedy

$$\begin{aligned} p \leq & 2(1 + c\frac{\ln D}{\ln \ln D}) \ln(1 + c\frac{\ln D}{\ln \ln D}) \leq \\ & 4c\frac{\ln D}{\ln \ln D} \ln(2c\frac{\ln D}{\ln \ln D}) \leq \\ & 4c(\ln 2c)\frac{\ln D}{\ln \ln D} + 4c\frac{\ln D}{\ln \ln D} \ln(\frac{\ln D}{\ln \ln D}) = \\ & 4c \ln D + o(\ln D) = O(\ln D) = O(n^2 \ln N). \end{aligned}$$

□

---

**Věta.** *Pro danou množinu $S \subseteq U$ takovou, že $|S| = n$, deterministický algoritmus nalezne prvočíslo $p = O(n^2 \log N)$ takové, že $\phi_p(x) = x \bmod p$ je perfektní pro $S$, a pracuje v čase $O(n^3 \log n \log N)$.*

*Důkaz.* Test, zda funkce $\phi_p(x) = x \bmod p$ je perfektní pro $S$, vyžaduje čas $O(n \log n)$. Tedy systematické hledání nejmenšího $p$, že $\phi_p$ je perfektní pro $S$, vyžaduje čas $O(n^3 \log n \log N)$. □

---

**Věta.** *Pravděpodobnostní algoritmus nalezne prvočíslo $p = O(n^2 \log N)$ takové, že $\phi_p$ je perfektní, v očekávaném čase $O(n \log n(\log n + \log \log N))$.*

*Důkaz.* Nejmenší $p$ takové, že $\phi_p$ je perfektní pro $S$, je prvočíslo. Navrhneme pravděpodobnostní algoritmus pro nalezení $p$.

Pro dostatečně velké $n$ mezi prvními $9c \ln D$ čísly je alespoň polovina takových prvočísel $p$, že $\phi_p$ je perfektní pro $S$. Algoritmus pak opakuje následující krok, dokud nenalezne perfektní funkci

- vyberme náhodně číslo $p$ mezi prvními $9cn^2 \ln N$ čísly a otestujme, zda $p$ je prvočíslo a $\phi_p$ je perfektní

Odhadneme očekávaný počet neúspěšných kroků.

Náhodně zvolené číslo $p \leq 9cn^2 \ln N$ je prvočíslo s pravděpodobností $\Theta(\frac{1}{\ln(9cn^2 \ln N)})$ (použijeme Rabin-Millerův pravděpdobnostní algoritmus na testování prvočísel) a pro prvočíslo $p$ je $\phi_p$ perfektní s pravděpodobností $\geq \frac{1}{2}$. Tedy náhodně zvolené číslo $p \leq 9cn^2 \ln N$ splňuje test s pravděpodobností $\Theta(\frac{1}{\ln(9cn^2 \ln N)})$, a proto očekávaný počet neúspěšných testů je $O(\ln(9cn^2 \ln N))$. Tedy očekávaný čas algoritmu je $O(n \log n(\log n + \log \log N))$.

□

---

**Pozorování.** *Deterministický algoritmus nalezne nejmenší prvočíslo s požadovanou vlastností. Pravděpodobnostní algoritmus nalezne prvočíslo, které může být podstatně větší, ale jeho velikost je omezena $9cn^2 \log N$.*

### 2.16.9 Konstrukce perfektní hašovací funkce E

Neformálně: Nyní všechny funkce $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ „zkombinujeme“ dohromady.

1. První vezmeme funkci $D_D$.
2. Na zahešované množině najdeme funkci $B_D$ s nějakým prvočíslem jako velikostí tabulky.
3. Na zahešované množině najdeme funkci $C_D$.

Podobně nedeterministicky. Pokud to chápu správně, použijeme $B$ jakoby dvakrát; jednou v $B$ a jednou v $C$.

Všechno jsme to takhle zkombinovali, protože:

- $C$ se nám líbí, protože je perfektní a má malou tabulku; ale potřebuje moc paměti na uložení. Tak si zmenšíme prostor, ve kterém se pohybujeme.
- Pokud si předtím omezíme prostor pomocí $B$, tak je paměť, nutná k $C$, o něco menší, ale zase trvá příliš dlouho nalézt parametr k $B$ a je moc velký
- Proto si ještě pomůžu na začátku $D$, která nám prostor zmenší.

Formálně:

1. Nalezneme prvočíslo $q_0 \in O(n^2 \log N)$ takové, že $\phi_{q_0}(x) = x \bmod q_0$ je perfektní funkce pro $S$. Položme $S_1 = \{\phi_{q_0}(s) \mid s \in S\}$.
2. Nalezneme prvočíslo $q_1$ takové, že $n(n-1) < q_1 \leq 2n(n-1)$. Pak existuje $l \in \{1, 2, \ldots, q_0 - 1\}$ takové, že $h_l(x) = ((lx) \bmod q_0) \bmod q_1$ je perfektní pro $S_1 \subseteq \{0, 1, \ldots, q_0 - 1\}$. Položme $S_2 = \{h_l(s) \mid s \in S_1\}$.
3. Dále zkonstruujme perfektní hašovací funkci $g$ pro množinu $S_2 \subseteq \{0, 1, \ldots, q_1 - 1\}$ do tabulky s méně než $3n$ řádky. Položme $f(x) = g(h_l(\phi_{q_0}(x)))$. Konstruovaná hašovací funkce je $f$.

Výsledek: $f$ je perfektní hašovací funkce pro $S$, protože složení perfektních hašovacích funkcí je zase perfektní funkce, a tedy požadavek 1) je splněn.

$f$ hašuje $S$ do tabulky s méně než $3n$ řádky, a tedy splňuje požadavek 2).

Protože každá z funkcí $g$, $h_l$, $\phi_{q_0}$ se vyčíslí v čase $O(1)$, i vyčíslení funkce $f$ vyžaduje čas $O(1)$ a požadavek 3) je splněn.

Funkce $\phi_{q_0}$ je jednoznačně určena číslem $q_0 \in O(n^2 \log N)$. Funkce $h_l$ je určena čísly $q_1 \in O(n^2)$ a $l \in O(q_0)$. Funkce $g$ je určena $n + 1$ čísly velikosti $O(q_1)$. Tedy zadání $f$ vyžaduje paměť o velikosti

$$O(\log n + \log \log N + n \log n) = O(n \log n + \log \log N).$$

Lze říct, že požadavek 4) je splněn.

Výpočet $\phi_{q_0}$ vyžaduje čas $O(n^3 \log n \log N)$. Výpočet $h_l$ vyžaduje čas $O(n(n^2 \log N)) = O(n^3 \log N)$ (použité univerzum je $\{0, 1, \ldots, q_0\}$). Výpočet $g$ vyžaduje čas $O(nn^2) = O(n^3)$ (zde univerzum je $\{0, 1, \ldots, q_1\}$). Celkově výpočet $f$ vyžaduje čas $O(n^3 \log n \log N)$.

Lze použít i pravděpodobnostní algoritmy pro nalezení $g$, $h_l$ a $\phi_{q_0}$. Pak hašujeme do tabulky s méně než $6n$ řádky, ale očekávaný čas pro nalezení $f$ je $O(n \log n(\log n + \log \log N))$.

Tuto metodu navrhli Fredman, Komlós a Szemerédi.

### 2.16.10 Univerzální a perfektní hašování

Předchozí hlavní konstrukce perfektní hašovací funkce vycházela z idejí použitých v univerzálním hašování. Ukážeme, že to není náhodná shoda. Dokážeme, že každý $c$-univerzální systém funkcí umožňuje základní konstrukci perfektní hašovací funkce. Pro každé $m$ nechť $\mathcal{H}_m = \{h_i \mid i \in I\}$ je $c$-univerzální systém funkcí hašujících do tabulky velikosti $m$. Pro libovolnou, ale pevnou podmnožinu $S \subseteq U$ o velikosti $n$ definujme $b_j^i = \{s \in S \mid h_i(s) = j\}$ pro každé $j = 0, 1, \ldots, m-1$ a $i \in I$. Když $b_j^i \geq 2$, pak $(b_j^i)^2 - b_j^i \geq 2$, a když $b_j^i \leq 1$, pak $(b_j^i)^2 - b_j^i = 0$. Odtud dostáváme přirozené zobecnění důsledku z předchozí sekce, které využijeme stejným způsobem jako v předchozí sekci:

**Důsledek.** *Když $\sum_{j=0}^{m-1}(b_j^i)^2 \geq n+2$, pak $h_i$ není perfektní pro $S$, když $\sum_{j=0}^{m-1}(b_j^i)^2 < n+2$, pak $h_i$ je perfektní hašovací funkce pro $S$.*

Stejně jako v předchozí sekci spočítáme

$$\sum_{i\in I}\big(\sum_{j=0}^{m-1}(b_j^i)^2 - n\big) = \sum_{i\in I}|\{(s,t) \mid s,t \in S,\ s \neq t,\ h_i(s) = h_i(t)\}| =$$
$$\sum_{s,t\in S, s\neq t}|\{i \in I \mid h_i(s) = h_i(t)\}| \leq \sum_{s,t\in S,s\neq t}\frac{c|I|}{m} = \frac{cn(n-1)|I|}{m}.$$

Proto existuje $i \in I$ takové, že $\sum_{j=0}^{m-1}(b_j^i)^2 \leq \frac{cn(n-1)}{m} + n$. Nyní spočítáme analogické odhady, které tvoří základ pro pravděpodobnostní algoritmus. Mějme kladné číslo $a > 0$ a označme $I'$ množinu těch $i \in I$, že

$$\sum_{j=0}^{m-1}(b_j^i)^2 > \frac{(c+a)n(n-1)}{m} + n.$$

Dále předpokládejme, že $|I'| \geq b|I|$ pro nějaké kladné číslo $b$. Pak platí

$$\frac{cn(n-1)|I|}{m} \geq \sum_{i\in I}\big(\sum_{j=0}^{m-1}(b_j^i)^2 - n\big) \geq \sum_{i\in I'}\big(\sum_{j=0}^{m-1}(b_j^i)^2 - n\big) >$$
$$\sum_{i\in I'}\frac{(c+a)n(n-1)}{m} \geq \frac{(c+a)n(n-1)b|I|}{m}.$$

Odtud plyne, že $c > (c+a)b$, a proto $b < \frac{c}{c+a}$. Když tedy vybíráme $h_i \in \mathcal{H}_m$ náhodně s rovnoměrným rozdělením (vzhledem k $i \in I$), pak pravděpodobnost, že bude platit $\sum_{j=0}^{m-1}(b_j^i)^2 > \frac{(c+a)n(n-1)}{m} + n$, je menší než $\frac{c}{c+a}$. Stejně jako v předchozí sekci shrneme tato fakta do tvrzení

**Tvrzení.** *Pro přirozené číslo $m$ mějme $c$-univerzální systém funkcí $\mathcal{H}_m = \{h_i \mid i \in I\}$ hašujících do tabulky o velikosti $m$. Pro $m = n$ existuje deterministický algoritmus, který v čase $O(|I|n)$ nalezne $h_i \in \mathcal{H}_m$ takovou, že $\sum_{j=0}^{m-1}(b_j^i)^2 \leq c(n-1)+n$, a existuje pravděpodobnostní algoritmus, který pro kladné číslo $a$ v čase $O(n)$ nalezne $h_i \in \mathcal{H}_m$ takové, že*

$$\sum_{j=0}^{m-1}(b_j^i)^2 < (c+a)(n-1)+n,$$

*a očekávaný počet iterací při hledání $h_i$ je menší než $\frac{c+a}{a}$.*

*Pro $m = \frac{cn(n-1)}{2} + 1$ existuje deterministický algoritmus, který nalezne perfektní hašovací funkci $h \in \mathcal{H}_m$ pro množinu $S$ velikosti $n$ v čase $O(n|I|)$.*

*Pro $a > 0$ a pro $m = \frac{(c+a)n(n-1)}{2} + 1$ existuje pravděpodobnostní algoritmus, který nalezne perfektní hašovací funkci $h \in \mathcal{H}_m$ pro množinu $S$ v čase $O(n)$ a očekávaný počet iterací je menší než $\frac{c+a}{a}$.*

*Důkaz.* Když $n = m$, pak v čase $O(n)$ pro hašovací funkci ověříme, zda $\sum_{j=0}^{m-1}(b_j^i)^2 \leq c(n-1)+n$, respektive $\sum_{j=0}^{m-1}(b_j^i)^2 \leq (c+a)(n-1)+n$. V prvním případě víme, že taková funkce v souboru $\mathcal{H}_m$ existuje, a systematickým prohledáváním všech funkcí v daném $c$-univerzálním systému $\mathcal{H}_m$ ji nalezneme v čase $O(n|I|)$. Pro pravděpodobnostní algoritmus budeme vybírat funkci ze souboru $\mathcal{H}_m$ náhodně s rovnoměrným rozdělením. Pak očekávaný počet iterací než uspějeme je

$$\sum_{i=1}^{\infty} i(\frac{c}{c+a})^{i-1}(1-\frac{c}{c+a}) \leq \sum_{i=1}^{\infty} i(\frac{c}{c+a})^{i-1} - \sum_{i=1}^{\infty} i(\frac{c}{c+a})^{i} =$$
$$\sum_{i=0}^{\infty}(\frac{c}{c+a})^{i} = \frac{1}{1-\frac{c}{c+a}} = \frac{c+a}{a}.$$

Pro hledání perfektní hašovací funkce opět použijeme systematické prohledávání $c$-univerzálního systému, protože víme, že existuje funkce $h_i \in \mathcal{H}_m$ taková, že $\sum_{j=1}^{m-1}(b_j^i)^2 \leq \frac{cn(n-1)}{\frac{cn(n-1)}{2}+1} + n < n+2$, a tedy je tato funkce perfektní. To vyžaduje čas $O(n|I|)$.

Když máme $a > 1$ a $m = \frac{(c+a)n(n-1)}{2} + 1$ a když budeme volit funkce z $\mathcal{H}_m$ náhodně s rovnoměrným rozdělením, pak s pravděpodobností $\frac{a}{c+a}$ dostaneme funkci $h_i$ takovou, že

$$\sum_{j=0}^{m-1}(b_j^i)^2 \leq \frac{(c+a)n(n-1)}{\frac{(c+a)n(n-1)}{2}+1} + n < 2 + n.$$

Z důsledku plyne, že $h_i$ je perfektní. Analýza očekávaného počtu iterací je stejná jako u předchozího tvrzení pro pravděpodobnostní algoritmus. □

---

Další postup konstrukce perfektní hašovací funkce už nesouvisí s $c$-univerzálními systémy.

### 2.16.11 Dynamické perfektní hašování

Jedna z velkých nevýhod perfektního hašovaní je neznalost efektivních aktualizačních operací. Existují sice obecné metody na dynamizaci deterministických operací – viz letní přednáška, ale tato metoda v tomto případě neposkytuje efektivní dynamizační operace, protože deterministický algoritmus pro řešení perfektního hašování je pro aktualizační operace příliš pomalý. To vedlo k návrhu, který kombinuje pravděpodobnostní algoritmus pro perfektní hašování s obecnou metodou dynamizace a tyto metody jsou upraveny pro konkrétní situaci.

Nejprve uvedeme modifikaci výsledků z předchozí části, na kterých je tato metoda založena. Předpokládáme, že $U = \{0, 1, \ldots, N-1\}$ je univerzum, kde $N$ je prvočíslo, a že je dáno číslo $s < N$. Označme $\mathcal{H}_s = \{h_k \mid k = 1, 2, \ldots, N-1\}$ množinu funkcí z $U$ do $\{0, 1, \ldots, s-1\}$, kde $h_k(x) = (kx \bmod N) \bmod s$ pro každé $x \in U$. Když zvolíme náhodně $k = 1, 2, \ldots, N-1$, pak s pravděpodobností alespoň $\frac{1}{2}$ platí

$$\sum_{i=0}^{s-1}(b_i^k)^2 < \frac{4n^2}{s} + n.$$

Skutečně, když pro méně než $\frac{N-1}{2}$ hodnot $k$ platí $\sum_{i=0}^{m} -1(b_i^k)^2 \leq \frac{4n}{m}$, pak

$$\sum_{k=1}^{N-1}\left((\sum_{i=0}^{m-1}(b_i^k)^2) - n\right) \geq 2(N-1)\frac{n^2}{m}$$

a to je spor. Budeme předpokládat, že takové $k$ máme, a pak pro každé $i = 0, 1, \ldots, s-1$ předpokládáme, že náhodně zvolíme $j_i \in \mathcal{H}_{2(b_i^k)^2}$ takové, že $h_{j_i}$ je prostá na množině $S_i = \{s \in S \mid h_k(s) = i\}$ (z předchozího textu víme, že když zvolíme náhodně $j_i = 0, 1, \ldots, N-1$, pak s pravděpodobností alespoň $\frac{1}{4}$ je $h_{j_i}$ prostá na $S_i$). Pro jednoduchost předpokládáme, že množiny $S_i$ pro $i = 0, 1, \ldots, s-1$ uložíme do tabulek $T_i$ a tabulky $T_0, T_1, \ldots, T_{s-1}$ budou uloženy v tabulce $T$. Když $s = O(|S|)$, pak tato metoda vyžaduje $O(|S|)$ prostoru. Abychom určili $s$, zvolme $c > 1$ a položme $s = \sigma(|S|)$, kde $\sigma(n) = \frac{4}{3}\sqrt{6}(1+c)n$ pro každé $n$. Nyní popíšeme algoritmy. Zde $n$ je velikost reprezentované množiny, $s = \sigma(n)$ a $2m(j)$ je velikost tabulky $T_j$ pro $j = 0, 1, \ldots, s-1$.

### 2.16.12 Algoritmy

**INSERT**$(x)$:
$n := n + 1$
**if** $n \leq s$ **then**
  $j := h(x)$, $|S_j| := |S_j| + 1$
  **if** $|S_j| \leq m(j)$ a pozice $h_j(x)$ v $T_j$ je prázdná **then**
    vložíme $x$ do tabulky $T_j$ na pozici $h_j(x)$
  **else**
    **if** $|S_j| \leq m(j)$ a pozice $h_j(x)$ v $T_j$ je obsazená **then**
      vytvoříme seznam $S_j$ prvků v tabulce $T_j$
      vyprázdníme tabulku $T_j$
      zvolíme náhodně funkci $h_j \in \mathcal{H}_{m(j)^2}$
      **while** $h_j$ není prostá na množině $S_j$ **do**
        zvolíme náhodně funkci $h_j \in \mathcal{H}_{m(j)^2}$
      **enddo**
      **for every** $y \in S_j$ **do** vložíme $y$ do $T_j$ na pozici $h_j(y)$ **enddo**
    **else**
      $m(j) := 2m(j)$
      **if** není dost prostoru pro tabulku $T_j$ nebo

$$\sum_{i=0}^{\sigma(m)-1} (m(i))^2 \geq \frac{4n^2}{\sigma(n)} + n$$

      **then**
        **RehashAll**
      **else**
        alokujeme prostor pro novou prázdnou tabulku $T_j$
        vytvoříme seznam $S_j$ prvků ze staré tabulky $T_j$ a zrušíme ji
        zvolíme náhodně funkci $h_j \in \mathcal{H}_{m(j)^2}$
        **while** $h_j$ není prostá na množině $S_j$ **do**
          zvolíme náhodně funkci $h_j \in \mathcal{H}_{m(j)^2}$
        **enddo**
        **for every** $y \in S_j$ **do** vložíme $y$ do $T_j$ na pozici $h_j(y)$ **enddo**
      **endif**
    **endif**
  **else**
    **RehashAll**
  **endif**
**endif**

**RehashAll**:

projdeme tabulku $T$ a tabulky $T_i$ a vytvoříme seznam prvků z množiny $S$
$s := \sigma(n)$
zvolme náhodně $h \in \mathcal{H}_s$
**for every** $i = 0, 1, \ldots, s-1$ **do** $S_i := \{x \in S \mid h(x) = i\}$ **enddo**
**while** $\sum_{i=0}^{s-1} 2(|S_i|)^2 > \frac{8n^2}{s} + 2n$ **do**
  zvolme náhodně $h \in \mathcal{H}_s$
  **for every** $i = 0, 1, \ldots, s-1$ **do** $S_i := \{x \in S \mid h(x) = i\}$ **enddo**
**enddo**
Komentář: zde $S_i$ jsou množiny vytvořené náhodně zvolenou funkcí $h$
$n := 0$
**for every** $i = 0, 1, \ldots, s-1$ **do**
  $m(i) := 2|S_i|$
  zvolíme náhodně $h_i \in \mathcal{H}_{m(i)^2}$
  **while** $h_i$ není prostá na množině $S_i$ **do**
    zvolíme náhodně $h_i \in \mathcal{H}_{m(i)^2}$
  **enddo**
**enddo**
**for every** $x \in S$ **do INSERT**$(x)$ **enddo**

**DELETE**$(x)$:
$j := h(x)$, $n := n - 1$, $|S_j| := |S_j| - 1$
odstraníme $x$ z pozice $h_j(x)$ v tabulce $T_j$, pozice bude prázdná
**if** $n < \frac{m}{1+2c}$ **then RehashAll endif**

**MEMBER**$(x)$:
$j := h(x)$
**if** $x$ je na $h_j(x)$-té pozici v tabulce $T_j$ **then**
  **Výstup**: $x$ je prvek $S$
**else**
  **Výstup**: $x$ není prvkem $S$
**endif**

Algoritmy předpokládají, že při operaci **INSERT**$(x)$ prvek $x$ nepatří do $S$ a při operaci **DELETE**$(x)$ $x$ je prvkem $S$. Velikost reprezentované množiny je $n$.

Uvedu složitost této metody bez důkazu.

**Věta.** *Popsaná metoda vyžaduje lineární paměť (neuvažuje se paměť potřebná pro zakódování hašovacích funkcí), operace* **MEMBER** *v nejhorším případě vyžaduje čas* $O(1)$ *a očekávaná amortizovaná složitost operací* **INSERT** *a* **DELETE** *je také* $O(1)$.

Toto zobecnění Fredman-Komlós-Szemerédiho metody navrhli Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert a Tarjan.

Další nevýhoda Fredman-Komlós-Szemerédiho metody:
Navržená metoda pracuje pro $m < 3n$, ale nezajistí $m = n$. Lze říct, že paměť je efektivně využita? Existuje metoda, která by umožnila návrh perfektní hašovací funkce pro $m = n$? Z výsledků pro $(N, m, n)$-perfektní soubory funkcí plyne existence $(N, n, n)$-perfektního souboru pro $n^N > e^{n+ln(n)} \ln(N)$. Zmíníme se orientačně o parametrizované metodě, která navrhuje perfektní hašovací funkci pro $S \subseteq U$ a pro $|S| = n$. Parametr bude přirozené číslo $r$, které určuje, jaké hypergrafy jsou užity při konstrukci funkce. Proto nejdříve připomeneme několik definic.

Dvojice $(X, E)$, kde $X$ je množina a $E$ je systém $r$-prvkových podmnožin $X$, se nazývá *$r$-hypergraf*. Prvky v $E$ se nazývají *hrany* $r$-hypergrafu. *Cyklus* je hypergraf $(X, E)$, kde každý vrchol leží alespoň ve dvou

různých hranách. Naopak $r$-hypergraf $(X, E)$ se nazývá *acyklický*, když žádný jeho podhypergraf není cyklus.

Nyní popíšeme metodu, která je rozdělena do dvou kroků. Je dáno $S \subseteq U$ takové, že $|S| = n$.

Krok 1) Mějme $r$-hypergraf $(V, E)$, kde $|E| = n$. Nalezneme zobrazení

$$g : V \to \{0, 1, \ldots, n-1\}$$

takové, že funkce $h : E \to \{0, 1, \ldots, n-1\}$ definovaná $h(e) = \sum_{i=1}^{r} g(v_i) \bmod n$, kde $e = \{v_1, v_2, \ldots, v_r\}$, je prostá (místo sčítání modulo $n$ můžeme použít libovolnou grupovou operaci na množině $\{0, 1, \ldots, n-1\}$). Pro acyklický $r$-hypergraf lze funkci $g$ zkonstruovat následujícím postupem. Zvolíme bijekci $h : E \to \{0, 1, \ldots, n-1\}$ a pak definujeme $g$ následovně: když $e = \{v_1, v_2, \ldots, v_r\}$ a $g(v_i)$ je definováno pro $i = 2, 3, \ldots, r$, pak

$$g(v_1) = h(e) - \sum_{i=2}^{r} g(v_i) \bmod n.$$

Protože pro každý acyklický $r$-hypergraf existuje vrchol, který leží v jediné hraně, lze tento postup použít ke konstrukci $g$ pomocí indukce (a tedy máme algoritmus pro konstrukci $g$).

Krok 2) Nalezneme $r$ funkcí $f_1, f_2, \ldots, f_r : U \to V$ takových, že $(V, E)$, kde

$$E = \{\{f_1(x), f_2(x), \ldots, f_r(x)\} \mid x \in S\},$$

je acyklický $r$-hypergraf. Pak hašovací funkce $f$ je definována $f(x) = \sum_{i=1}^{r} g(f_i(x))$ pro každé $x \in U$. Z konstrukce vyplývá, že je perfektní na množině $S$.

Autoři dokázali, že nejvhodnější alternativa je, když zobrazení $f_1, f_2, \ldots, f_r$ jsou náhodná zobrazení náhodně zvolená. Bohužel taková zobrazení neumíme zkonstruovat, ale autoři ukázali, že pro tyto účely lze použít náhodný výběr funkcí z nějakého $c$-univerzálního souboru funkcí.

Autoři ukázali, že jejich algoritmus vyžaduje $O(rn + |V|)$ času a $O(n \log n + r \log |V|)$ paměti.

Tento metapostup navrhli Majewski, Wormald, Havas a Czech (1996).

Pro praktické použití je problematická reprezentace $r$-hypergrafu a i náhodná volba funkcí $f_1, f_2, \ldots, f_r$ (viz předchozí diskuze o $c$-univerzalitě). Z požadavků na perfektní hašovací funkci je opět problémem splnění požadavku 4). Nevím, jak je uvedená metoda prakticky použitelná a zda se někde používá.

## 2.17 Externí hašování

Navržený postup je také znám pod názvem Faginův algoritmus. Tímto problémem se jako první asi zabýval Larsson.

Řešíme jiný problém – uložení dat v externí paměťi. Hlavní problém – minimalizovat přístupy na externí paměť.

Předpoklady: Externí paměť je rozdělena na stránky, každá stránka obsahuje $b$ položek (dat) (předpokládáme, že $b > 1$, jinak to nemá smysl). Vždy v jednom kroku načteme celou stránku do interní paměti nebo celou stránku v interní paměti v jednom kroku zapíšeme na externí medium. Tyto operace jsou řádově pomalejší než operace v interní paměti.

Podobný problém se také řeší pri práci s cache-pamětí. V tom případě však neovlivňujeme, která stránka se bude načítat, kdežto v případě externí paměti to právě musíme řešit.

Náš cíl: Nalézt způsob ukládání dat do stránek externí paměti, aby se minimalizoval počet operací s externí pamětí.

Předpokládejme, že $h : U \to \{0,1\}^*$ je prosté zobrazení takové, že délka $h(u)$ je stejná pro všechny prvky univerza $U$. Označme $k$ délku $h(u)$ pro $u \in U$. Pak $h$ je hašovací funkce (to znamená, že hašovací funkce je přejmenování prvků). Nechť $S \subseteq U$, pak pro slovo $\alpha$ délky menší než $k$ definujme

$$h_S^{-1}(\alpha) = \{s \in S \mid \alpha \text{ je prefix } h(s)\}.$$

Řekneme, že $\alpha$ je *kritické slovo*, když $0 < |h_S^{-1}(\alpha)| \leq b$ a pro každý vlastní prefix $\alpha'$ slova $\alpha$ platí $|h_S^{-1}(\alpha')| > b$. Pro každé $s \in S$ existuje právě jedno kritické slovo $\alpha$, které je prefixem $h(s)$. Definujme $d(s)$ pro $s \in S$ jako délku kritického slova, které je prefixem $h(s)$ a

$$d(S) = \max\{\text{délka}(\alpha) \mid \alpha \text{ je kritické slovo}\} = \max\{d(s) \mid s \in S\}.$$

Množinu $S$ reprezentujeme tak, že je jednoznačná korespondence mezi kritickými slovy a stránkami externí paměti sloužícími k reprezentaci $S$. Na stránce příslušející kritickému slovu $\alpha$ je reprezentován soubor $h_S^{-1}(\alpha)$.

Problém: jak nalézt stránku kritického slova $\alpha$?
Řešení: Adresář je funkce, která každému slovu $\alpha$ o délce $d(S)$ přiřadí adresu stránky předpisem

1. když kritické slovo $\beta$ je prefixem $\alpha$, pak k $\alpha$ je přiřazena stránka korespondující s $\beta$, jinak je k $\alpha$ přiřazena stránka $NIL$ – speciální prázdná stránka.

Korektnost: Pro různá kritická slova $\beta$ a $\gamma$ platí $h_S^{-1}(\beta) \cap h_S^{-1}(\gamma) = \emptyset$, a tedy pro každé slovo $\alpha$ délky $d(S)$ existuje nejvýše jedno kritické slovo, které je prefixem $\alpha$. Když $\alpha$ je slovo délky $d(S)$, pak nastane jeden z těchto tří případů:

1. $h_S^{-1}(\alpha) \neq \emptyset$, pak $0 < |h_S^{-1}(\alpha)| \leq b$ a existuje právě jedno kritické slovo $\beta$, které je prefixem $\alpha$;
2. $h_S^{-1}(\alpha) = \emptyset$ a existuje prefix $\alpha'$ slova $\alpha$ takový, že $0 < |h_S^{-1}(\alpha')| \leq b$, pak existuje právě jedno kritické slovo, které je prefixem $\alpha'$ (a tedy také prefixem $\alpha$);
3. $h_S^{-1}(\alpha) = \emptyset$ a pro každý prefix $\alpha'$ slova $\alpha$ platí buď $h_S^{-1}(\alpha') = \emptyset$ nebo $|h_S^{-1}(\alpha')| > b$ (pak k $\alpha$ je přiřazena stránka $NIL$).

Mějme slovo $\alpha$ o délce $d(S)$. Označme $c(\alpha)$ nejkratší prefix $\alpha'$ slova $\alpha$ takový, že každému slovu $\beta$ o délce $d(S)$, které má $\alpha'$ za prefix, je přiřazená stejná stránka jako slovu $\alpha$. Všimněme si, že když $h_S^{-1}(\alpha) \neq \emptyset$, pak $c(\alpha)$ je kritické slovo. Platí silnější tvrzení, které tvrdí, že následující podmínky jsou ekvivalentní:

1. stránka přiřazená slovu $\alpha$ je různá od $NIL$;
2. $c(\alpha)$ je kritické slovo;
3. nějaký prefix $\alpha$ je kritické slovo.

Všimněme si, že znalost adresáře umožňuje nalézt slovo $c(\alpha)$ pro každé slovo o délce $d(S)$.

Lineární uspořádání na slovech délky $n$ nazveme *lexikografické*, když $\alpha < \beta$, právě když $\alpha = \gamma 0 \alpha'$ a $\beta = \gamma 1 \beta'$ pro nějaká slova $\gamma$, $\alpha'$ a $\beta'$. Lexikografické uspořádání vždy existuje a je jednoznačné.

Reprezentace adresáře: Je to seznam adres stránek o délce $2^{d(S)}$ takový, že adresa na $i$-tém místě odpovídá $i$-tému slovu délky $d(S)$ v lexikografickém uspořádání.

Příklad: $U$ je množina všech slov nad $\{0,1\}$ o délce 5, $h$ je identická funkce a $b = 2$. Reprezentujme množinu $S = \{00000, 00010, 01000, 10000\}$. Pak $d(00000) = d(00010) = d(01000) = 2$, $d(10000) = 1$, kritická slova jsou 00, 01 a 1 a adresář je (místo adresy stránky uvedeme množinu, která je na této stránce uložena)

$$00 \mapsto \{00000, 00010\}, \quad 01 \mapsto \{01000\}, \quad 10 \mapsto 11 \mapsto \{10000\}.$$

Tedy $c(00) = 00$, $c(01) = 01$ a $c(10) = c(11) = 1$. Když odstraníme prvek 10000, pak 1 přestane být kritické slovo a adresář bude mít tvar

$$00 \mapsto \{00000, 00010\}, \quad 01 \mapsto \{01000\}, \quad 10 \mapsto 11 \mapsto NIL.$$

Opět platí $c(00) = 00$, $c(01) = 01$ a $c(10) = c(11) = 1$. V adresáři je také uloženo $d(S)$.

### 2.17.1 Algoritmy

Uvedeme zde jen slovní popis operací. Předpokládáme, že adresář je uložen v externí paměti na jedné stránce.

**MEMBER**$(x)$
1) Spočítáme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix $\alpha$ slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k $\alpha$. Když je to stránka $NIL$, pak $x \notin S$ a konec, jinak pokračujeme krokem 2).
2) **Načteme** stránku příslušející k $\alpha$ do interní paměti. Prohledáme ji a pokud neobsahuje $x$, pak $x \notin S$ a konec. Když obsahuje $x$, pak provedeme požadované změny a stránku **uložíme** do externí paměti na její původní místo. Konec.

**INSERT**$(x)$
1) Spočítáme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix $\alpha$ slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k $\alpha$ a slovo $c(\alpha)$. Když stránka přiřazená k $\alpha$ je $NIL$, pokračujeme krokem 3), v opačném případě pokračujeme krokem 2).
2) **Načteme** stránku přiřazenou slovu $\alpha$. Když $x$ je uloženo na této stránce, pak skončíme. Když $x$ není na této stránce, pak tam přidáme slovo $x$. Pokud na stránce je nejvýše $b$ prvků, pak **uložíme** stránku na její původní místo a skončíme. Když na stránce je více než $b$ prvků, pak nalezneme nová kritická slova, která nám stránku rozdělí, a vytvoříme dvě stránky – jednu **uložíme** na místo původní stránky a druhou **uložíme** na novou stránku. Pokračujeme krokem 4).
3) Vytvoříme v interní paměti novou stránku, která obsahuje $x$, nalezneme novou stránku v externí paměti a tam **uložíme** vytvořenou stránku (všem slovům, která mají $c(\alpha)$ za prefix, bude přiřazena tato stránka) a pokračujeme krokem 4).
4) **Načteme** opět adresář do interní paměti, aktualizujeme adresy přiřazených stránek a případně zvětšíme adresář (to nastane, když nějaké nové kritické slovo má délku větší než $d(S)$, pak nové $d(S)$ je právě délka tohoto slova – obě kritická slova vzniklá v kroku 2) mají stejnou délku). Aktualizovaný adresář **uložíme** do externí paměti. Konec.

**DELETE**$(x)$
1) Spočítáme $h(x)$ a **načteme** adresář do interní paměti. Vezmeme prefix $\alpha$ slova $h(x)$ o délce $d(S)$ a nalezneme adresu stránky příslušející k $\alpha$ a slovo $c(\alpha)$. Když stránka přiřazená k $\alpha$ je $NIL$, pak skončíme. Označme $\beta'$ slovo, které má stejnou délku jako $c(\alpha)$ a liší se od $c(\alpha)$ pouze v posledním bitu. Když existuje slovo $\beta$ délky $d(S)$ takové, že $c(\beta) = \beta'$, pak stránka přiřazená k $\beta$ je *kandidát*.
2) **Načteme** stránku příslušnou k slovu $\alpha$ do interní paměti. Když tato stránka neobsahuje $x$, pak skončíme. Když tato stránka obsahuje $x$, pak odstraníme $x$ z této stránky. Když neexistuje kandidát nebo když nová stránka a stránka kandidáta dohromady obsahují více než $b$ prvků, pak novou stránku **uložíme** na její původní místo a skončíme.
3) Když nová stránka a stránka kandidáta mají dohromady $b$ prvků, pak **načteme** stránku kandidáta do interní paměti. V interní paměti tyto stránky spojíme do jedné a tuto stránku pak **uložíme** do externí paměti.
4) **Načteme** adresář, kde zaktualizujeme adresy stránek. Pokud jsme sloučili dvě stránky, musíme nalézt nové $c(\alpha)$ (je to nejkratší prefix $\alpha'$ slova $\alpha$ takový, že ke každému slovu $\beta$ o délce $d(S)$, které má $\alpha'$ za prefix, je přiřazena jedna z těchto adres: adresa stránky přiřazená k $\alpha$, adresa stránky kandidáta, $NIL$) a

každému slovu o délce $d(S)$, které má nové $c(\alpha)$ za prefix, bude přiřazena adresa nové (spojené) stránky. Otestujeme, zda se adresář nemůže zkrátit (to nastane, když adresy stránek přiřazené $(2i+1)$-ímu slovu a $(2i+2)$-ému slovu o délce $d(S)$ jsou stejné pro všechna $i$, pak se tato slova spojí a $d(S)$ se zmenší o 1). Upravený adresář **uložíme**. Konec.

Následující věta ukazuje, že jsme náš hlavní cíl splnili. Pro jednoduchost předpokládáme, že adresář je také uložen na externí paměti a že v interní paměti nemůže být uložen spolu s nějakou jinou stránkou.

**Věta.** *Operace* **MEMBER** *vyžaduje nejvýše tři operace s externí pamětí. Operace* **INSERT** *a* **DELETE** *vyžadují nejvýše šest operací s externí pamětí.*

V našem příkladu provedeme operaci **INSERT**(00001). Po přidání prvku stránka původně přiřazená k slovu 00 vypadá takto $\{00000, 00001, 00010\}$. Tuto stránku rozdělíme na stránky $\{00000, 00001\}$ a $\{00010\}$. Přitom kritické slovo první stránky je 0000 a druhé stránky je 0001. Takže $d(S) = 4$ a adresář vypadá

$$\begin{aligned}
&0000 \mapsto \{00000, 00001\}, 0001 \mapsto \{00010\},\\
&0010 \mapsto 0011 \mapsto NIL,\\
&0100 \mapsto 0101 \mapsto 0110 \mapsto 0111 \mapsto \{0100\},\\
&1000 \mapsto 1001 \mapsto 1010 \mapsto 1011 \mapsto \{10000\},\\
&1100 \mapsto 1101 \mapsto 1110 \mapsto 1111 \mapsto \{10000\}.
\end{aligned}$$

To znamená, že kromě adresy 00 se ostatní slova rozdělila na čtyři slova, ale adresy zůstaly stejné. Jen u slova 00 vzniklá slova dostala různé adresy.

V původním příkladu provedeme operaci **DELETE**(01000). Pak kandidát je 00 a po odstranění prvku 01000 nastane spojení těchto dvou stránek. Po aktualizaci adres dostane adresář tvar

$$00 \mapsto 01 \mapsto \{00000, 00010\}, 10 \mapsto 11 \mapsto \{10000\},$$

tj. k prvnímu a druhému slovu je přiřazena stejná stránka a stejně tak k třetímu a čtvrtému slovu. Takže můžeme adresář zmenšit. Pak $d(S) = 1$ a adresář má podobu

$$0 \mapsto \{00000, 00010\}, 1 \mapsto \{10000\}.$$

Vzniká otázka, jak je tato metoda efektivní. Hlavně jak efektivně využívá paměť. Platí

**Věta.** *Když velikost reprezentované množiny je $n$, pak očekávaný počet použitých stránek je $\frac{n}{b \ln 2}$ a očekávaná velikost adresáře je $\frac{e}{b \ln 2} n^{1+\frac{1}{b}}$.*

První tvrzení říká, že očekávaný počet prvků na stránce je $b \ln 2 \approx 0.69b$. Tedy zaplněno je asi 69% míst. Tento výsledek není překvapující a je akceptovatelný. Horší je to s adresářem, jak ukazuje následující tabulka

| velikost $S$ | $10^5$ | $10^6$ | $10^8$ | $10^{10}$ |
|---|---|---|---|---|
| 2 | $6.2 \cdot 10^7$ | $1.96 \cdot 10^8$ | $1.96 \cdot 10^{11}$ | $1.96 \cdot 10^{14}$ |
| 10 | $1.2 \cdot 10^5$ | $1.5 \cdot 10^6$ | $2.4 \cdot 10^8$ | $3.9 \cdot 10^{10}$ |
| 50 | $9.8 \cdot 10^3$ | $1.0 \cdot 10^6$ | $1.1 \cdot 10^8$ | $1.2 \cdot 10^{10}$ |
| 100 | $4.4 \cdot 10^3$ | $4.5 \cdot 10^4$ | $4.7 \cdot 10^6$ | $4.9 \cdot 10^8$ |

kde jednotlivé řádky odpovídají hodnotám $b$ uvedeným v prvním sloupci. Protože očekávaná velikost adresáře se zvětšuje rychleji než lineárně (exponent u $n$ je $1 + \frac{1}{b}$), tak nelze očekávat, že tuto metodu lze vždy použít. Výpočty i experimenty ukazují, že použitelná je do velikosti $|S| = 10^{10}$, když $b \approx 100$. V tomto rozmezí je nárůst adresáře jen kolem 5%. Pro větší $n$ je třeba, aby $b$ bylo ještě větší.

# 3 Vyhledávání v uspořádaném poli

pozn studenta - tato kapitola byla uprostřed stromů, ale tam mi nedávala smysl, takže jsem si jí dovolil přesunout sem.

## 3.1 Zadání úlohy

Máme podmnožinu $S$ lineárně uspořádaného univerza a $S$ je uložena v poli $A[1..|S|]$ tak, že pro $i < j$ je $A(i) < A(j)$. Pro dané $x \in U$ máme zjistit, zda $x \in S$ (operace **MEMBER**$(x)$).

## 3.2 Metaalgoritmus

Pokud $x < A(1)$ nebo $A(|S|) < x$, pak $x$ není prvkem $S$.

V opačném případě buď $x = A(1)$ nebo $x = A(|S|)$ nebo máme dvě hodnoty $d$ a $h$ takové, že $1 \leq d < d + 1 < h \leq |S|$ a $A(d) < x < A(h)$. Pak najdeme $n$ takové, že $d < n < h$, a dotazem zjistíme, zda $x = A(n)$ (pak končíme a $x \in S$) nebo $x < A(n)$ (pak položíme $h = n$) nebo $x > A(n)$ (pak položíme $d = n$) a proces opakujeme. Končíme, když $d + 1 \geq h$, pak $x \notin S$. Na začátku položíme $d = 1$ a $h = |S|$. Formální zápis algoritmu:

**MEMBER**$(x)$
**if** $x = A(1)$ **then**
  **Výstup**: $x \in S$ **stop**
**else**
  **if** $x < A(1)$ **then**
    **Výstup**: $x \notin S$ **stop**
  **else**
    $d = 1$
  **endif**
**endif**
**if** $x = A(|S|)$ **then**
  **Výstup**: $x \in S$ **stop**
**else**
  **if** $x > A(|S|)$ **then**
    **Výstup**: $x \notin S$ **stop**
  **else**
    $h = |S|$
  **endif**
**endif**
**while** $d + 1 < h$ **do**
  $n :=$ **next**$(d, h)$
  **if** $\mathbf{x} = A(n)$ **then**
    **Výstup**: $\mathbf{x} \in S$ **stop**
  **else**
    **if** $x < A(n)$ **then** $h := n$ **else** $d := n$ **endif**
  **endif**
**enddo**
**Výstup**: $x \notin S$ **stop**

V tomto metaalgoritmu je **next**$(d, h)$ funkce, která nalezne hodnotu $n$ takovou, že $d < n < h$. Korektnost

plyne z pozorování, že když $d+1=h$, pak $A(d) < x < A(h)$ implikuje, že neexistuje $i$ takové, že $x = A(i)$, a tedy $x \notin S$. Efektivita algoritmu záleží na fukci **next**. Zpracování dotazu vyžaduje čas $O(1)$ a počet dotazů je počet volání funkce **next**.

## 3.3 Typy funkce next

Unární vyhledávání: $\mathbf{next}(d,h) = d+1$, pak každý dotaz zvětší $d$ o 1, a tedy největší počet dotazů je $|S|$. Algoritmus v nejhorším případě vyžaduje čas $O(|S|)$ a očekávaný počet dotazů při rovnoměrném rozložení množiny $S$ a prvku $x$ je $\frac{|S|}{2}$ (tedy očekávaný čas je $O(|S|)$).

Poznámka: Duální přístup je, když $\mathbf{next}(d,h) = h-1$, výsledky se nezmění. Při aplikacích je někdy výhodné použít funkci $\mathbf{next}(d,h) = \min\{d+c, h-1\}$, kde $c$ je nějaká konstanta (pak krok není 1, ale $c$). Jak uvidíme později, jsou situace, kdy je výhodné takovéto unární vyhledávání použít.

Binární vyhledávání: $\mathbf{next}(d,h) = \lceil \frac{d+h}{2} \rceil$, pak každý dotaz zmenší rozdíl $h-d$ přibližně na polovinu. Počet dotazů je nejvýše $3+\log(|S|-2)$, algoritmus tedy v nejhorším případě vyžaduje čas $O(\log|S|)$ a očekávaný čas při rovnoměrném rozložení množiny $S$ a $x \in U$ je také $O(\log|S|)$.

Interpolační vyhledávání: $\mathbf{next}(d,h) = d + \lceil \frac{x-A(d)}{A(h)-A(d)}(h-d) \rceil$. V nejhorším případě musíme položit více než $\frac{|S|}{2}$ dotazů, a proto čas v nejhorším případě je $O(|S|)$, ale při rovnoměrném rozložení množiny $S$ a $x \in U$ je očekávaný čas $O(\log\log|S|)$. To je založeno na faktu, že hodnota **next** závisí i na velikosti $x$. Když $x$ je velké, tak hodnota **next** je posunuta do větších hodnot, když $x$ je malé, pak je posunuta do menších hodnot.

Poznámka: Když rozložení prvků není rovnoměrné, ale je známé, pak podle toho můžeme upravit funkci **next** a očekávaný čas algoritmu se nezmění.

Pro následující funkci **next** bude jednodušší spočítat očekávaný počet dotazů než pro interpolační vyhledávání, ale výsledek je asymptoticky stejný.

### 3.3.1 Zobecněné kvadratické vyhledávání

Pozn. studenta: Tohle vůbec nechápu a mám to jako TODO, jestli to stihnu.

Funkce **next** je zde definována složitější procedurou, jejíž výsledek závisí i na předchozích situacích a na výsledku dotazu. Procedura zadává dotazy v blocích. První dotaz v bloku je interpolační a procedura přitom zjistí velikost kroku a zda $x$ je menší nebo větší než první dotaz v bloku. Pak střídá unární a binární vyhledávání. Blok končí, když rozdíl mezi $h$ a $d$ je nejvýše velikost kroku. Krok v následujícím bloku klesne přibližně na odmocninu velikosti kroku v tomto bloku. Procedura používá boolské proměnné *blok*, *typ*, *smer*. Proměnná *blok* je inicializována hodnotou $false$ a určuje, zda se dotaz zadává v rámci stejného bloku nebo nikoliv. Proměnná *typ* určuje, zda příští dotaz je unární (když $typ = true$) nebo binární. Proměnná *smer* určuje, zda dotazy jsou menší než první dotaz v bloku ($smer = true$) nebo větší. Dále procedura používá proměnnou *krok* typu integer, která obsahuje velikost kroku v rámci bloku. Hodnoty těchto proměnných se předávají z jednoho volání procedury do dalšího volání (tj. jsou to globální proměnné, které se neinicializují voláním procedury **next**).

```
next(d, h)
if blok then
  if typ then
    if smer then
      next(d, h) := h − krok
      if A(next(d, h)) < x then
```

$blok := false$
**endif**
**else**
**next**$(d,h) := d + krok$
**if** $A($**next**$(d,h)) > x$ **then**
$blok := false$
**endif**
**endif**
$typ := false$
**else**
**if** $\min\{h - \lceil\frac{d+h}{2}\rceil, \lceil\frac{d+h}{2}\rceil - d\} < krok$ **then**
$blok := false$
**endif**
**next**$(d,h) := \lceil\frac{d+h}{2}\rceil$, $typ := true$
**endif**
**else**
$krok := \lfloor\sqrt{h-d}\rfloor$, **next**$(d,h) := d + \lceil\frac{x-A(d)}{A(h)-A(d)}(h-d)\rceil$,
**if** $A($**next**$(d,h)) > x$ **then**
$smer := true$
**else**
$smer := false$
**endif**
$typ := true$, $blok := true$
**endif**

Po dvou dotazech klesne $h - d$ buď pod $\sqrt{h-d}$ nebo pod $\frac{h+d}{2}$. Proto procedura v nejhorším případě použije nejvýše $8 + 2\log(|S|-1) + 2\log\log|S|$ dotazů, a tedy v nejhorším případě vyžaduje čas $O(\log|S|)$.

Nyní spočítáme očekávaný počet dotazů během jednoho bloku za předpokladu rovnoměrného rozdělení dat. Nechť $p_i$ je pravděpodobnost, že v rámci bloku se položí alespoň $i$ dotazů. Pak očekávaný počet dotazů v rámci bloku je

$$E(C) = \sum_{i\geq 1} i(p_i - p_{i+1}) = \sum_{i\geq 1} p_i.$$

Nyní odhadneme $p_i$. Označme $n + d$ argument prvního dotazu (interpolační vyhledávaní) v rámci bloku a nechť $krok = k$ v rámci bloku. Označme $X = |\{i \mid i > d,\ A(i) \leq x\}|$ na začátku bloku, pak $X$ je náhodná proměnná závislá na argumentu operace a bloku. Když se v bloku položí alespoň $i$ dotazů pro $i > 2$, pak $|X - n| \geq \lfloor\frac{i-2}{2}\rfloor k$, protože každý unární dotaz, jehož položení nezmění blok, nalezne dalších $k$ hodnot $i$ v rozdílu $|X - n|$. Tedy

$$p_i \leq \mathrm{Prob}(|X - n| \geq \lfloor\frac{i-2}{2}\rfloor k).$$

Použijeme Čebyševovu nerovnost pro náhodnou proměnnou $X$. Když $Y$ je náhodná proměnná s očekávanou (střední) hodnotou $\mu$ a rozptylem $\sigma^2$, pak Čebyševova nerovnost říká, že

$$\mathrm{Prob}(|Y - \mu| \geq t) \leq \frac{\sigma^2}{t^2} \qquad \text{pro každé } t > 0.$$

Uvažujme okamžik, kdy jsme na začátku nějakého bloku. Protože $S$ je vybraná s rovnoměrným rozdělením, je pravděpodobnost, že $A(i) < x$ pro $d < i < h$, rovna $p = \frac{x-A(d)}{A(h)-A(d)}$, a pak pravděpodobnost, že $X = j$, je $\binom{h-d}{j}p^j(1-p)^{h-d-j}$. To znamená, že $X$ je náhodná veličina s binomickým rozdělením s rozsahem $d - h$ a

pravděpodobností $p$, a tedy její očekávaná hodnota je

$$\mu = \sum_{j=0}^{h-d} j \binom{h-d}{j} p^j (1-p)^{h-d-j} = p(h-d)$$

a rozptyl má hodnotu

$$\sigma^2 = \sum_{j=0}^{h-d} (j-\mu)^2 \binom{h-d}{j} p^j (1-p)^{h-d-j} = p(1-p)(h-d).$$

Když si uvědomíme, že $k = \lfloor \sqrt{h-d} \rfloor$ a $n = p(h-d)$, pak dostáváme

$$p_i, p_{i+1} \leq \text{Prob}(|X-n| \geq \lfloor \frac{i-2}{2} \rfloor k) \leq \frac{4p(1-p)(h-d)}{(i-2)^2 k^2} \leq$$
$$\frac{4p(1-p)}{(i-2)^2} \leq \frac{1}{(i-2)^2},$$

protože pro $0 \leq p \leq 1$ je $p(1-p) \leq \frac{1}{4}$. Když shrneme tato pozorování, dostáváme, že

$$E(C) = \sum_{i \geq 1} p_i \leq 2 + 2 \sum_{i \geq 3} \frac{1}{(i-2)^2} = 2 + 2 \sum_{i \geq 1} \frac{1}{i^2} =$$
$$2 + 2\frac{\pi^2}{6} = 2 + \frac{\pi^2}{3} \approx 5.3$$

Závěr: očekávaný počet dotazů v bloku je menší než 6.

Když $E(T(n))$ je očekávaný počet dotazů pro operaci **MEMBER** a když $|S| = n$, pak platí

$$E(T(n)) \leq E(C) + E(T(\sqrt{n})).$$

Protože $E(T(1)) = 1$ a $E(T(2)) \leq 2$, dostáváme z rekurentního vzorce, že

$$E(T(n)) \leq 2 + E(C) \log\log n \qquad \text{pro } n \geq 2.$$

**Věta.** *Čas operace* **MEMBER** *v uspořádaném poli délky n při zobecněném kvadratickém vyhledávání je v nejhorším případě* $O(\log n)$*. Když rozdělení vstupních dat je rovnoměrné, pak očekávaný čas je* $O(\log\log n)$*.*

Nevýhoda této datové struktury spočívá v neexistenci přirozených efektivních implementací operací **INSERT**, **DELETE**, **SPLIT** a **JOIN**. Přirozené implementace těchto operací vyžadují čas $O(|S|)$, zhruba řečeno musíme pohybovat s téměř každým prvkem. Pokusem o řešení tohoto problému byl návrh binárních vyhledávacích stromů.

# 4 Stromy

## 4.1 Uspořádaný slovníkový problém

Jedná se o rozšíření základního slovníkového problému. Je dáno totálně uspořádané univerzum $U$ (tj. pro každé dva různé prvky $u, v \in U$ platí buď $u < v$ nebo $v < u$). Cílem je reprezentovat množinu $S \subseteq U$ a navrhnout algoritmy pro tyto operace:
**MEMBER**, **INSERT**, **DELETE**
**MIN** – nalezne nejmenší prvek v $S$,

**MAX** – nalezne největší prvek v $S$,
**SPLIT**$(x)$ – zkonstruuje reprezentace dvou množin $S_1 = \{s \in S \mid s < x\}$ a $S_2 = \{s \in S \mid s > x\}$ a oznámí, zda $x \in S$,
**JOIN** – používají se dvě verze této operace:
**JOIN2**$(S_1, S_2)$ – jsou dány reprezentace množin $S_1$ a $S_2$, které splňují $\max S_1 < \min S_2$, vytvoří se reprezentace množiny $S = S_1 \cup S_2$,
**JOIN3**$(S_1, x, S_2)$ – jsou dány reprezentace množin $S_1$ a $S_2$ a prvek $x \in U$ tak, že je splněno $\max S_1 < x < \min S_2$, vytvoří se reprezentace množiny $S = S_1 \cup \{x\} \cup S_2$.

Je vidět, že operace **JOIN2** a **JOIN3** lze pomocí operací **INSERT** a **DELETE** převést jednu na druhou. Proto často budeme popisovat pro danou strukturu jen jednu z nich. Občas se také používá operace
**ord**$(k)$ – předpokádáme, že $k \leq |S|$, a operace nalezne $k$-tý nejmenší prvek v $S$.
Zřejmě operace **MIN** a **MAX** jsou speciálním případem operace **ord**$(k)$, přesně **MIN** je operace **ord**$(1)$ a **MAX** je operace **ord**$(|S|)$.

## 4.2 $(a, b)$-stromy

### 4.2.1 Obecná definice

Důležitou datovou strukturou vhodnou pro řešení uspořádaného slovníkového problému jsou $(a, b)$-stromy. Tuto datovou strukturu lze použít pro interní i pro externí paměť. Je to struktura založená na stromech. Nejobecnější grafová definice $(a, b)$-stromu je:

Nechť $1 \leq a < b$ jsou kladná přirozená čísla. Pak kořenový strom $(T, t)$ se nazývá $(a, b)$-*strom*, když

1. když $v$ je vnitřní vrchol stromu $T$ různý od kořene $t$, pak má alespoň $a$ a nejvýše $b$ synů;
2. všechny cesty z kořene do libovolného listu mají stejnou délku.

### 4.2.2 Speciální případ – definice

Tato definice je příliš obecná a pro datové struktury se nehodí. Proto používáme její speciální případ (změny podtrženy).

Datová struktura $(a, b)$-strom je definována jen na těchto stromech: Nechť $a$ a $b$ jsou přirozená čísla taková, že $2 \leq a$ a $2a - 1 \leq b$. Pak kořenový strom $(T, t)$ nazveme $(a, b)$-*strom*, když platí

1. každý vnitřní vrchol $v$ stromu $T$ různý od kořene $t$ má alespoň $a$ a nejvýše $b$ synů;
2. kořen je buď list nebo má alespoň dva syny a nejvýše $b$ synů;
3. všechny cesty z kořene do libovolného listu mají stejnou délku.

### 4.2.3 Vlastnosti – velikost

Výhody našich $(a, b)$-stromů:
Když má $(a, b)$-strom *výšku* $h > 0$ (tj. délka každé cesty z kořene do libovolného listu je $h$), pak strom má alespoň $2a^{h-1}$ listů a nejvýše $b^h$ listů. Z toho jednoduše plyne:

**Tvrzení.** *Mějme přirozená čísla $a$ a $b$ taková, že $a \geq 2$ a $b \geq 2a - 1$. Pak pro každé kladné přirozené číslo $n$ existuje $(a, b)$-strom, který má přesně $n$ listů. Když $(a, b)$-strom má přesně $n$ listů, pak výška stromu je nejvýše $1 + \log_a(\frac{n}{2})$ a je alespoň $\log_b n$. Tedy výška stromu je $O(\log n)$.*

### 4.2.4 Vlastnosti – uspořádání na listech

Mějme kořenový strom $(T, t)$ takový, že pro každý vnitřní vrchol $v$ platí:
když $v$ má $\rho(v)$ synů, pak jsou očíslovány od 1 do $\rho(v)$. Řekneme, že vrchol $v$ je v *hloubce* $h$, když cesta z kořene $t$ do $v$ má délku $h$. Množina všech vrcholů v hloubce $h$ se nazývá *$h$-tá hladina*. *Lexikografické uspořádání* na $h$-té hladině je definováno rekurzivně:

$v \leq w$, právě když buď $\text{otec}(v) < \text{otec}(w)$ nebo $\text{otec}(v) = \text{otec}(w)$ a když $v$ je $i$-tý syn $\text{otec}(v)$ a $w$ je $j$-tý syn $\text{otec}(v)$, pak $i \leq j$.

Předpokládáme, že v $(a, b)$-stromu synové každého vnitřního vrcholu jsou uspořádány. Listy tvoří hladinu $h$, kde $h$ je hloubka $(a, b)$-stromu, a je na nich definováno lexikografické uspořádání.

Mějme lineárně uspořádané univerzum $U$ a množinu $S \subseteq U$. Pak $(a, b)$-strom $(T, t)$ *reprezentuje* množinu $S$, když má přesně $|S|$ listů a je dán izomorfismus mezi lexikografickým uspořádáním listů stromu $T$ a uspořádanou množinou $S$ (tj. bijekce $\text{key} : \text{list}(T) \to S$, která pro $s, t \in S$ splňuje $s \leq t$ v $U$, právě když $\text{key}^{-1}(s) \leq \text{key}^{-1}(t)$ v lexikografickém uspořádání na množině listů stromu $T$).

### 4.2.5 Jak reprezentujeme množinu?

Co je uloženo ve vnitřních vrcholech $(a, b)$-stromu $(T, t)$ reprezentujícího množinu $S \subseteq U$?

- $\rho(v)$ – počet synů vrcholu $v$,
- $S_v(1..\rho(v))$ – pole ukazatelů na syny vrcholu $v$ takové, že $S_v(i)$ je $i$-tý syn vrcholu $v$ pro $i = 1, 2 \ldots, \rho(v)$,
- $H_v(1..\rho(v) - 1)$ – pole prvků z $U$ takové, že $H_v(i)$ je největší prvek z $S$ reprezentovaný v podstromu $i$-tého syna vrcholu $v$
  alternativa: $H_v(i)$ je prvek z $U$ takový, že největší prvek reprezentovaný v podstromu $i$-tého syna vrcholu $v$ je menší nebo roven $H_v(i)$ a to je menší než nejmenší prvek reprezentovaný v podstromu $(i + 1)$-ního syna vrcholu $v$

Struktura listů:
listu $v$ je přiřazen prvek $\text{key}(v) \in S$.

Někdy je ve struktuře každého vrcholu $v$ $(a, b)$-stromu různého od kořene ještě ukazatel $\text{otec}(v)$ na otce vrcholu $v$.

**Pozorování.** *Když $H_v(i)$ jsou prvky z reprezentované množiny, pak pro každý prvek $s \in S$ kromě největšího existuje právě jeden vnitřní vrchol $v$ $(a, b)$-stromu a jedno $i$, že $H_v(i) = s$, a největší prvek v $S$ není prvek $H_v$ pro žádný vrchol $v$.*

Tento fakt se používá při implementaci, kde se vynechávají listy. Prvky z $S$ jsou reprezentovány v polích $H_v$ vnitřních vrcholů stromu a největší prvek je uložen zvlášť nebo je k množině $S$ přidán formální největší prvek (a ten je pak „uložen“ zvlášť). Je to prostorově efektivnější reprezentace množiny $S$, ale je technicky nepřehledná. Proto při práci s $(a, b)$-stromy používám verzi s listy.

Nyní uvedeme algoritmy pro $(a, b)$-stromy.

### 4.2.6 Algoritmy

Pomocný algoritmus

**Vyhledej**($x$)
$t$ :=kořen stromu $T$, $w := NIL$
**while** $t$ není list **do**
  $i := 1$
  **while** $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  **if** $H_t(i) = x$ **then** $w := t$ **endif**
$t := S_t(i)$ **enddo** **Výstup**: $t$ a $w$.

**MEMBER**($x$)
**Vyhledej**($x$)
**if** key($t$) $= x$ **then** **Výstup**: $x \in S$ **else** **Výstup**: $x \notin S$ **endif**

**INSERT**($x$)
**Vyhledej**($x$)
**if** key($t$) $\neq x$ **then**
  vytvoř nový list $t'$, key($t'$) $:= x$, $u :=$ otec($t$)
  **if** key($t$) $< x$ **then**
(komentář: $x > \max S$)
    $S_u(\rho(u) + 1) := t'$, $H_u(\rho(u)) :=$ key($t$), $\rho(u) := \rho(u) + 1$
  **else**
    najdi $i$, že $S_u(i) = t$
    $S_u(\rho(u) + 1) := S(\rho(u))$, $j := \rho(u) - 1$
    **while** $j \geq i$ **do**
      $S_u(j + 1) := S_u(j)$, $H_u(j + 1) := H_u(j)$, $j := j - 1$
    **enddo**
    $S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u) + 1$
  **endif**
  $t := u$
  **while** $\rho(t) > b$ **do** **Štěpení**($t$) **enddo**
**endif**

**Štěpení**($t$)
**if** $t$ je kořen stromu **then**
  vytvoř nový kořen $u$ s jediným synem $t$
**endif**
$u :=$ otec($t$), najdi $i$, že $S_u(i) = t$,
vytvoř nový vnitřní vrchol $t'$, $j := 1$
**while** $j < \lfloor \frac{b+1}{2} \rfloor$ **do**
  $S_{t'}(j) := S_t(j + \lceil \frac{b+1}{2} \rceil)$, $H_{t'}(j) := H_t(j + \lceil \frac{b+1}{2} \rceil)$, $j := j + 1$
**enddo**
$S_{t'}(\lfloor \frac{b+1}{2} \rfloor) := S_t(b + 1)$, $\rho(t) := \lceil \frac{b+1}{2} \rceil$, $\rho(t') := \lfloor \frac{b+1}{2} \rfloor$,
**if** $i < \rho(u)$ **then** $S_u(\rho(u) + 1) := S_u(\rho(u))$ **endif**
$j := \rho(u) - 1$, $\rho(u) := \rho(u) + 1$,
**while** $j > i$ **do**
  $S_u(j + 1) := S_u(j)$, $H_u(j + 1) := H_u(j)$, $j := j - 1$
**enddo**
$S_u(i + 1) := t'$, $H_u(i + 1) := H_u(i)$, $H_u(i) := H_t(\rho(t))$, $t := u$

**DELETE**$(x)$
**Vyhledej**$(x)$
**if** key$(t) = x$ **then**
  $u :=$ otec$(t)$, najdi $i$, že $S_u(i) = t$, a $j$, že $H_w(j) = x$, $k := i$
  **if** $w \neq u$ a $w \neq NIL$ **then** $H_w(j) := H_u(\rho(u) - 1)$ **endif**
  **while** $k < \rho(u) - 1$ **do**
    $H_u(k) := H_u(k + 1)$, $S_u(k) := S_u(k + 1)$, $k := k + 1$
  **enddo**
  **if** $i \neq \rho(u)$ **then** $S_u(\rho(u) - 1) := S_u(\rho(u))$ **endif**
  $\rho(u) := \rho(u) - 1$, odstraň $t$, $t := u$
  **while** $\rho(t) < a$ a $t$ není kořen **do**
    $y$ je bezprostřední bratr $t$
    **if** $\rho(y) = a$ **then Spojení**$(t, y)$ **else Přesun**$(t, y)$ **endif**
  **enddo**
**endif**

**Spojení**$(t, y)$
$u :=$ otec$(t)$, najdi $i$, že $S_u(i) = t$, $j := 1$
**if** $S_u(i - 1) = y$ **then** vyměň $t$ a $y$, $i := i - 1$ **endif**
**while** $j < \rho(y)$ **do**
  $S_t(\rho(t) + j) := S_y(j)$, $H_t(\rho(t) + j) := H_y(j)$, $j := j + 1$
**enddo**
$H_t(\rho(t)) := H_u(i)$, $S_t(\rho(t) + \rho(y)) := S_y(\rho(y))$, $\rho(t) := \rho(t) + \rho(y)$, odstraň $y$
**while** $i < \rho(u) - 1$ **do**
  $S_u(i + 1) := S_u(i + 2)$, $H_u(i) := H_u(i + 1)$, $i := i + 1$
**enddo**
$\rho(u) := \rho(u) - 1$
**if** $u$ je kořen a $\rho(u) = 1$ **then**
  odstraň $u$
**else**
  $t := u$
**endif**

**Přesun**$(t, y)$
$u :=$ otec$(t)$, najdi $i$ takové, že $S_u(i) = t$
**if** $S_u(i + 1) = y$ **then**
  $S_t(\rho(t) + 1) := S_y(1)$, $H_t(\rho(t)) := H_u(i)$,
  $H_u(i) := H_y(1)$, $j := 1$
  **while** $j < \rho(y) - 1$ **do**
    $S_y(j) := S_y(j + 1)$, $H_y(j) := H_y(j + 1)$, $j := j + 1$
  **enddo**
  $S_y(\rho(y) - 1) := S_y(\rho(y))$, $\rho(t) := \rho(t) + 1$, $\rho(y) := \rho(y) - 1$
**else**
  $S_t(\rho(t) + 1) := S_t(\rho(t))$, $j := \rho(t) - 1$
  **while** $j > 0$ **do**
    $S_t(j + 1) := S_t(j)$, $H_t(j + 1) := H_t(j)$, $j := j - 1$
  **enddo**
  $\rho(t) := \rho(t) + 1$, $S_t(1) := S_y(\rho(y))$, $H_t(1) := H_u(i - 1)$,
  $H_u(i - 1) := H_y(\rho(y) - 1)$, $\rho(y) := \rho(y) - 1$
**endif**

**MIN**
$t$ :=kořen stromu
**while** $t$ není list **do** $t := S_t(1)$ **enddo**
key($t$) je nejmenší prvek $S$

**MAX**
$t$ :=kořen stromu
**while** $t$ není list **do** $t := S_t(\rho(t))$ **enddo**
key($t$) je největší prvek $S$

**JOIN2**$(T_1, T_2)$
Předpoklad $T_i$ je $(a, b)$-strom reprezentující množinu $S_i$ pro $i = 1, 2$, které splňují $\max S_1 < \min S_2$ (tento předpoklad je silnější než požadavek, že $S_1$ a $S_2$ jsou disjunktní, ale algoritmus nekontroluje jeho splnění)
**if** výška $T_1$ je větší nebo rovna výšce $T_2$ **then**
  $t$ :=kořen $T_1$, $k := v(T_1) - v(T_2)$
  **while** $k > 0$ **do** $t := S_t(\rho(t))$, $k := k - 1$ **enddo**
  **Spojení**($t$,kořen $T_2$), $t :=$ otec($t$)
  **while** $\rho(t) > b$ **do Štěpení**($t$) **enddo**
**else**
  $t$ :=kořen $T_2$, $k := v(T_2) - v(T_1)$
  **while** $k > 0$ **do** $t := S_t(1)$, $k := k - 1$ **enddo**
  **Spojení**($t$,kořen $T_1$), $t :=$ otec($t$)
  **while** $\rho(t) > b$ **do Štěpení**($t$) **enddo**
**endif**

```
SPLIT(T, x)
Z_1, Z_2 prázdné zásobníky, t :=kořen T
while t není list do
  i := 1
  while H_t(i) < x a i < ρ(t) do i := i + 1 enddo
  t := S_t(i)
  if i = 2 then vlož podstrom vrcholu S_t(1) do Z_1 endif
  if i > 2 then
    vytvoř nový vrchol t_1, ρ(t_1) = i − 1,
    for every j = 1, 2, ..., i − 2 do
      S_{t_1}(j) := S_t(j), H_{t_1}(j) := H_t(j)
    enddo
    S_{t_1}(i − 1) := S_t(i − 1), vlož podstrom vrcholu t_1 do Z_1
  endif
  if i = ρ(t) − 1 then
    vlož podstrom S_t(ρ(t)) do Z_2    endif
  if i < ρ(t) − 1 then
    vytvoř nový vrchol t_2, ρ(t_2) := ρ(t) − i
    for every j = 1, 2, ..., ρ(t) − i − 1 do
      S_{t_2}(j) := S_t(i + j), H_{t_2}(j) := H_t(i + j)
    enddo
    S_{t_2}(ρ(t) − i) := S_t(ρ(t)), vlož podstrom t_2 do Z_2
  endif
enddo
if key(t) = x then
  Výstup: x ∈ S
else
  Výstup: x ∉ S
  if key(t) < x then
    vlož podstrom vrcholu t do Z_1
  else
    vlož podstrom vrcholu t do Z_2
  endif
endif
T_1 := vrchol Z_1, odstraň T_1 ze Z_1
while Z_1 ≠ ∅ do
  T' :=vrchol Z_1, odstraň T' ze Z_1, T_1 :=JOIN(T', T_1)
enddo
T_2 := vrchol Z_2, odstraň T_2 ze Z_2
while Z_2 ≠ ∅ do
T' := vrchol Z_2, odstraň T' ze Z_2, T_2 :=JOIN(T_2, T')
enddo
```

### 4.2.7 Korektnost algoritmů

Odkaz na otce vrcholu: buď je v každém vrcholu $v$ stromu $T$ přímo odkaz na otec($v$), nebo se v proceduře **Vyhledej** vkládají vrcholy do zásobníku a otec($v$) je vrchol v zásobníku před vrcholem $v$.

Při operaci **SPLIT** se zásobníky používají jednoprůchodově – nejprve se naplní a v této části algoritmu se nepoužije operace **pop**, pak se vyprázdní a v této fázi se nepoužívá operace **push**. V okamžiku, když jsou zásobníky naplněné, platí:

- v zásobnících jsou uloženy $(a, b)$-stromy reprezentující podmnožiny $S$;
- když $(a, b)$-stromy $T_i$ a $T_{i+1}$ reprezentují množiny $S_i$ a $S_{i+1}$ a jsou v zásobníku $Z_1$ (nebo $Z_2$) a strom $T_{i+1}$ následuje po stromu $T_i$, pak platí $\max S_i < \min S_{i+1} < x$ (nebo $\min S_i > \max S_{i+1} > x$) a výška $T_i$ je větší nebo rovna výšce $T_{i+1}$;
- když $T_i$ a $T_{i+1}$ jsou dva po sobě následující $(a, b)$-stromy v zásobníku $Z_j$ pro $j = 1, 2$, které mají stejnou výšku, pak následující strom v zásobníku $Z_j$ má ostře menší výšku.

Toto plyne z první fáze algoritmu operace **SPLIT** a zajišťuje korektnost druhé fáze algoritmu.

### 4.2.8 Časová analýza

Dále si všimněme, že podprocedury **Štěpení**, **Spojení** a **Přesun** vyžadují čas $O(1)$, a proto algoritmy pro operace **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** a pro první fázi algoritmu **SPLIT** vyžadují čas $O(1)$ pro práci v dané hladině. Protože hladin je nejvýše $\log_a |S|$, můžeme shrnout:

**Věta.** *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN2** *a* **SPLIT** *v* $(a, b)$*-stromech vyžadují v nejhorším případě čas* $O(\log_a |S|)$*, kde* $S$ *je reprezentovaná množina.*

Je třeba ještě odhadnout spotřebovaný čas ve druhé fázi algoritmu pro operaci **SPLIT**.

Nejprve si všimněme, že algoritmus **JOIN2**$(T_1, T_2)$ vyžaduje ve skutečnosti jen čas rovný $O($rozdíl výšek stromů $T_1$

Když po naplnění zásobník $Z_j$ pro $j = 1, 2$ obsahuje stromy $U_1, U_2, \ldots, U_k$ v tomto pořadí, pak $k \leq 2 \log_a |S|$ a vyprázdnění zásobníku $Z_j$ vyžaduje čas $O(\sum_{i=1}^{k-1}(u_i - u_{i+1} + 1) = O(u_1 + k)$, kde $u_i$ je výška stromu $U_i$ pro $i = 1, 2, \ldots, k$. Protože výška stromu $U_1$ je nejvýše rovna výšce stromu $T$, dostáváme, že druhá fáze algoritmu **SPLIT** vyžaduje čas $O(\log_a |S|)$ a důkaz je kompletní.

### 4.2.9 Pořádková statistika

Nyní popíšeme algoritmus pro operaci **ord**$(k)$. Tato operace se často nazývá $k$-tá pořádková statistika Tato operace není podporována navrženou strukturou, pro její efektivní implementaci musíme rozšířit strukturu vnitřního vrcholu $v$ o pole
$P_v(1..\rho(v) - 1)$, kde $P_v(i)$ je počet prvků $S$ reprezentovaných v podstromu $i$-tého syna vrcholu $v$.

Udržovat pole $P_v$ v aktuálním stavu znamená při úspěšném provedení aktualizační operace projít cestu z vrcholu do kořene a aktualizovat pole $P$. Uvedeme algoritmus pro nalezení $k$-té pořádkové statistiky.

```
ord(k)
if k > |S| then neexistuje k-tý nejmenší prvek, konec endif
t :=kořen stromu
while t není list do
  i := 1
  while k > P_t(i) a i < ρ(t) do
    k := k − P_t(i), i := i + 1
  enddo
  t := S_t(i)
enddo
key(t) je hledaný k-tý nejmenší prvek
```

Invariant algoritmu: V každém okamžiku platí, že původní $k$ se rovná aktuální $k$+počet prvků z $S$, které jsou v podstromech vrcholů stromu, které v lexikografickém uspořádání předcházejí $i$-tému synu vrcholu $t$. Korektnost algoritmu plyne z tohoto invariantu.

**Věta.** *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **SPLIT**, **JOIN2** *a* **ord**$(k)$ *pro všechna* $k$ *v rozšířené struktuře* $(a,b)$*-stromu vyžadují v nejhorším případě čas* $O(\log |S|)$*, kde* $S$ *je reprezentovaná množina.*

### 4.2.10 Hodnoty $a, b$

$(a,b)$-stromy se používají jak v interní tak v externí paměti. Jaké hodnoty $a$ a $b$ je vhodné používat?
Pro interní paměť jsou doporučené hodnoty $a = 2$, $b = 4$ nebo $a = 3$ a $b = 6$.
Pro externí paměť jsou doporučené hodnoty $a \approx 100$, $b = 2a$.

### 4.2.11 Paralelní verze

Když je množina reprezentovaná $(a,b)$-stromem uložena na serveru a má k ní přístup více uživatelů, vzniká problém s aktualizačními operacemi. Tyto operace mění strukturu $(a,b)$-stromu a v důsledku toho se v něm jiný uživatel může ztratit. Tento problém se dá řešit tak, že při aktualizačních operacích se uzavře celý strom.

Nevýhoda: ostatní uživatelé do něho nemají přístup a nemohou pracovat. Tzv. paralelní implementace operací **INSERT** a **DELETE** nabízí jiné, efektivnější řešení.

Předpoklad: $b \geq 2a$.
Při operaci **INSERT** jsou ve vyhledávací fázi vždy uzavřeny vrcholy $t$, otec$(t)$ a synové vrcholu $t$. Algoritmus zjistí, ve kterém synu vrcholu $t$ má pokračovat, a pak, když $\rho(t) = b$, provede **Štěpení** (proto je nutně $b \geq 2a$, abychom po této operaci měli zase $(a,b)$-strom). V algoritmu pak odpadne vyvažovací část (tj. **Štěpení** při cestě vzhůru ke kořeni).

Při operaci **DELETE** jsou ve vyhledávací fázi uzavřeny vrcholy $t$, otec$(t)$, bezprostřední bratr $y$ vrcholu $t$ a jejich synové. Když $\rho(t) = a$, pak po najiti vrcholu, kde se bude pokračovat, se provede buď **Přesun** (když $\rho(y) > a$) nebo **Spojení** (když $\rho(y) = a$). Stejně jako při operaci **INSERT** se vynechá vyvažovací část uzavírající původní algoritmus.

Tato úprava vyžaduje sice více **Štěpení**, **Spojení** a **Přesunů**, ale asymptoticky vychází čas stejný (jen je větší multiplikativní konstanta). Doporučené hodnoty $a$ a $b$ jsou $a \approx 100$ a $b = 2a + 2$ při uložení na serveru v externí paměti, ve vnitřní paměti se doporučuje $a = 2$, $b = 6$.

Operace **JOIN2** lze také paralelizovat, ale operaci **SPLIT** paralelizovat nelze.

### 4.2.12 $A$-sort

$(a,b)$-stromy dávají také zajímavé aplikace pro třídicí algoritmy. Použití $(a,b)$-stromů pro setřídění náhodné posloupnosti není vhodné, režie na udržování struktury $(a,b)$-stromu vede k tomu, že multiplikativní konstanta by byla o hodně větší než u klasických třídicích algoritmů. Také uložení $(a,b)$-stromu vyžaduje více paměti než je potřeba pro klasické algoritmy. Situace se podstatně změní, když vstupní posloupnost je předtříděná a je ji třeba jen dotřídit. Klasické algoritmy většinou nejsou schopné využít faktu, že posloupnost je předtříděná, a jejich časová náročnost je prakticky stejná (někdy i horší) jako u náhodné posloupnosti. Na rozdíl od nich algoritmus $A$-**sort** založený na $(a,b)$-stromech je schopen předtříděnost využít a má na předtříděných posloupnostech lepší výsledky než klasické algoritmy.

Modifikace $(a,b)$-stromů pro algoritmus $A$-**sort**. Máme $(a,b)$-strom reprezentující vstupní posloupnost, je dán ukazatel Prv na první list, listy $(a,b)$-stromu jsou propojeny do seznamu v rostoucím lexikografickém pořadí (ukazatel na následující prvek je Nasl) a je dána cesta z prvního listu do kořene (to znamená, že na cestě z prvního listu do kořene známe pro každý vrchol $v$ jeho otce). Nyní uvedeme algoritmus $A$-**sort**.

$A$-**sort**$(x_1, x_2, \ldots, x_n)$
$i := n - 1$, vytvoř jednoprvkový strom s vrcholem $t$
key$(t) := x_n$, Prv $:= t$
**while** $i \geq 1$ **do** $A$-**Insert**$(x_i)$, $i := i - 1$ **enddo**
$y_1 :=$ key(Prv)
**while** $i \leq n$ **do**
  $y_i :=$ key$(t)$, $i := i + 1$, $t :=$ Nasl$(t)$
**enddo**
**Výstup**: $(y_1, y_2, \ldots, y_n)$ setříděná posloupnost $(x_1, x_2, \ldots, x_n)$

$A$-**Insert**$(x)$
$t :=$ Prv
**while** $t \neq$ kořen $T$ a $H_t(1) < x$ **do** $t :=$ otec$(t)$ **enddo**
**while** $t \neq$ list **do**
  $i := 1$
  **while** $H_t(i) < x$ a $i < \rho(t)$ **do** $i := i + 1$ **enddo**
  **if** $i > 1$ **then** $v := S_t(i - 1)$ **else** $v := S_t(\rho(t))$ **endif**
  $t := S_t(i)$
**enddo**
**if** key$(t) \neq x$ **then**
  vytvoř nový list $t'$, key$(t') = x$,
  **if** $t$ je kořen **then**
    vytvoř nový kořen $u$, $\rho(u) := 2$
    **if** key$(t) > x$ **then**
      $H_u(1) := x$, $S_u(1) := t'$, $S_u(2) := t$,
      Prv $:= t'$, Nasl$(t') := t$, Nasl$(t) := NIL$
    **else**
      $H_u(1) :=$ key$(t)$, $S_u(1) := t$, $S_u(2) := t'$
      Prv $:= t$, Nasl$(t) := t'$, Nasl$(t') := NIL$
    **endif**
  **else**
    $u :=$ otec$(t)$
    **if** key$(t) < x$ **then**
(komentář: $x > \max S$)
      $S_u(\rho(u) + 1) := t'$, $H_u(\rho(u)) :=$ key$(t)$, $\rho(u) := \rho(u) + 1$
      Nasl$(t) := t'$, Nasl$(t') := NIL$
    **else**
      najdi $i$, že $S_u(i) = t$, $S_u(\rho(u) + 1) := S(\rho(u))$,
      $j := \rho(u) - 1$, Nasl$(v) := t'$, Nasl$(t') := t$
      **while** $j \geq i$ **do**
        $S_u(j + 1) := S_u(j)$, $H_u(j + 1) := H_u(j)$, $j := j - 1$
      **enddo**
      $S_u(i) := t'$, $H_u(i) := x$, $\rho(u) := \rho(u) + 1$,
      **if** $t =$ Prv **then** Prv $:= t'$ **endif**
    **endif**
    $t := u$
    **while** $\rho(t) > b$ **do Štěpení**$(t)$ **enddo**
  **endif**
**endif**

Korektnost algoritmu plyne z faktu, že key je izomorfismus uspořádání a seznam listů je v rostoucím

pořadí. Protože $v$ je vždy bezprostřední předchůdce $t$, je seznam korektně definován. Ukazatel otec($t$) je dán na cestě z vrcholu Prv do kořene, pro ostatní vrcholy se řeší stejným způsobem jako pro $(a, b)$-stromy.

### 4.2.13 $A$-sort – složitost

Algoritmus $A$-**sort** vyžaduje více času i více paměti než klasické třídicí algoritmy, ale jejich asymptotická složitost je stejná. Jeho výhoda je v použití na předtříděné posloupnosti.

**Definice.** *Pro posloupnost $(x_1, x_2, \ldots, x_n)$ prvků z totálně uspořádaného univerza $U$ definujme*

$$F = |\{(i, j) \mid i < j,\, x_j < x_i\}|.$$

Zřejmě $F = 0$, právě když posloupnost $(x_1, x_2, \ldots, x_n)$ je setříděná.

Dále $0 \leq F \leq \binom{n}{2}$ a $F = \binom{n}{2}$, právě když je posloupnost $(x_1, x_2, \ldots, x_n)$ klesající.

To vede k tomu brát $F$ jako míru předtříděnosti posloupnosti. Spočítáme složitost algoritmu $A$-**sort** v závislosti na $n$ a $F$.

**Pozorování.** *Algoritmus $A$-**sort** v nejhorším případě vyžaduje čas, který potřebuje $A$-**Insert**, plus $O(n)$.*

**Pozorování.** *Algoritmus $A$-**Insert**$(x)$ vyžaduje čas potřebný na nalezení místa, kam vložit $x$, plus $O$(počet volání* **Štěpení**).

**Pozorování.** *Protože každý běh procedury* **Štěpení** *vytvořil jeden vnitřní vrchol $(a, b)$-stromu a protože $a \geq 2$ a $(a, b)$-strom po skončení volání $A$-**Insert** má $n$ listů, je vnitřních vrcholů $(a, b)$-stromu $< n$.*

**Pozorování** (Plyne z minulého)**.** *Všechny běhy procedury $A$-**Insert** vyžadují čas na nalezení míst jednotlivých prvků plus $O(n)$.*

**Pozorování.** *Když procedura $A$-**Insert**$(x)$ při hledání místa pro prvek $x$ skončila ve výšce $h$ (tj. první cyklus se $h$-krát opakoval), pak nalezení místa pro prvek $x$ vyžadovalo čas $O(h)$.*

**Pozorování.** *Všechny prvky reprezentované $(a, b)$-stromem pod prvním vrcholem ve výšce $h-1$ jsou menší než $x$ a je jich alespoň $a^{h-1}$.*

**Pozorování.** *Když $x = x_i$, pak počet prvků reprezentovaných $(a, b)$-stromem při běhu procedury $A$-**Insert**$(x)$, které jsou menší než $x$, je počet $j$ takových, že $i < j$ a $x_j < x_i$. (Plyne z toho, v jakém pořadí vkládáme.)*

**Definice.** *Označme $f_i$ počet $j$ takových, že $i < j$ a $x_j < x_i$.*

**Věta.** *Algoritmus $A$-**sort** na setřídění $n$-členné posloupnosti vyžaduje v nejhorším případě čas $O(n + n \log \frac{F}{n})$, kde $F$ je míra setříděnosti vstupní posloupnosti.*

*Důkaz.* Z pozorování platí

$$a^{h-1} \leq f_i \implies h - 1 \leq \log_a f_i \implies h \in O(\log f_i).$$

Proto v nejhorším případě čas potřebný pro nalezení pozice $x_i$ je $O(\log f_i)$. Odtud plyne, že čas algoritmu potřebný k běhu algoritmu $A$-**sort** je

$$O((\sum_{i=1}^{n} \log f_i) + n).$$

Zřejmě $\sum_{i=1}^{n} f_i = F$ a nyní využijeme toho, že geometrický průměr je vždy menší nebo roven aritmetickému průměru, a odtud dostáváme

$$\sum_{i=1}^{n} \log f_i = \log \prod_{i=1}^{n} f_i = n \log (\prod_{i=1}^{n} f_i)^{\frac{1}{n}} \leq$$
$$n \log \frac{\sum_{i=1}^{n} f_i}{n} = n \log \frac{F}{n}.$$

□

---

Zhodnocení: Protože $A$-**sort** nepoužívá operaci **DELETE**, doporučuje se použít $(2,3)$-stromy. Když se budou třídit posloupnosti s mírou $F \leq n \log n$, pak algoritmus $A$-**sort** bude potřebovat v nejhorším případě čas $O(n \log \log n)$. Mehlhorn a Tsakalidis dokázali, že když $F \leq 0.02n^{1.57}$, pak algoritmus $A$-**sort** je rychlejší než algoritmus **Quicksort**.

### 4.2.14 Propojené stromy s prstem

*Hladinově propojený* $(a,b)$*-strom s prstem* je $(a,b)$-strom, kde struktura vnitřního vrcholu různého od kořene je rozšířena (proti klasickému $(a,b)$-stromu) o ukazatele otec($v$), levy($v$), pravy($v$), kde:

- levy($v$) ukazuje na největší vrchol (v lexikografickém uspořádání) ve stejné hladině jako $v$, který je menší než $v$ (když neexistuje, tak je to $NIL$),
- pravy($v$) ukazuje na nejmenší vrchol (v lexikografickém uspořádání) ve stejné hladině jako $v$, který je větší než $v$ (když neexistuje, tak je to $NIL$).

Navíc je dán ukazatel Prst na některý list.

Zde se liší hlavně vyhledávání, které je zobecněním postupu $A$-**sortu**. Začínáme od listu $p$, na který ukazuje Prst. Když $x$ je menší než prvek reprezentovaný tímto listem, pak se pokračuje v jeho otci $v$, a když $p$ byl $i$-tý syn $v$, tak se pomocí pole $H_v$ zjišťuje, zda $x$ nemá být reprezentován v podstromu jeho $j$-tého syna pro $j < i$. Když ne, pokračuje se ukazatelem levy($v$). Když $x$ není reprezentován ani v jeho podstromu, tak se celý postup opakuje o hladinu výš (zkoumá se otec vrcholu). Když $x$ je větší než prvek reprezentovaný listem $p$, je postup zrcadlově obrácený. Když se nalezne vrchol, v jehož podstromu má $x$ ležet, pak se aplikuje od tohoto vrcholu (místo od kořene) procedura **Vyhledej**.

Struktura kromě operací uspořádaného slovníkového problému ještě používá přidanou operaci **PRST**($x$), která nastaví ukazatel Prst na list, který reprezentuje nejmenší prvek větší nebo rovný $x$ (pokud $x > \max S$, tak ukazatel Prst bude ukazovat na největší list). Operace provedou vyhledání a pak pokračují klasickým způsobem.

Použití: Tato struktura je velmi výhodná pro úlohy, kde vždy skupina po sobě jdoucích operací pracuje v blízkém okolí nějakého $x \in U$. Pak vyhledání prvku je rychlejší než v klasickém $(a,b)$-stromu, viz $A$-**sort**.

### 4.2.15 Omezení štěpení, spojování a přesunů

Pozn. studenta - dost jsem popřesouval pořadí dokazování v této kapitole tak, aby mi dávalo logický smysl.

Vyvažovací operace **Štěpení, Spojování, Přesun** vyžadují čas $O(1)$, ale ve skutečnosti jsou nejpomalejší částí algoritmů pro operace **INSERT** a **DELETE**. Omezení jejich počtu vedlo k menší složitosti algoritmu $A$-**sort**. To motivovalo analýzu jejich použití.

Libovolný běh algoritmu **INSERT** volá podproceduru **Štěpení** nejvýše $\log(|S|)$-krát a libovolný běh algoritmu **DELETE** může nejvýše $\log(|S|)$-krát zavolat podproceduru **Spojení** a nejvýše jednou podproceduru **Přesun**. V obecném případě tyto odhady nejdou zlepšit. Pro vhodný typ $(a, b)$-stromu však amortizovaný počet vyvažovacích operací (začínáme-li s původně prázdným stromem) je konstantní.

Připomínáme, že *výška vrcholu* v kořenovém stromě je maximální délka cesty z něho do některého listu. (tj. je počítaná odspodu)

Důkaz omezení počtu štěpení je založen na bankovním principu – navrhneme kvantitativní ohodnocení $(a, b)$-stromu, nalezneme jeho horní odhad a popíšeme, jak toto ohodnocení mohou změnit vyvažovací operace. Srovnání těchto odhadů dá požadovaný výsledek.

Nechť $b \geq 2a$ a $a \geq 2$.

**Definice.** *Pro pevné $a$ a $b$ označme*

$$c = \min\{\min\{2a-1, \lceil \frac{b+1}{2} \rceil\} - a, b - \max\{2a-1, \lfloor \frac{b+1}{2} \rfloor\}\}.$$

**Definice.** *Mějme $(a, b)$-strom $T$, pro vnitřní vrchol $v$ různý od kořene definujme $b(v) = \min\{\rho(v) - a, b - \rho(v), c\}$, pro kořen $r$ definujme $b(r) = \min\{\rho(r) - 2, b - \rho(r), c\}$.*

$b$ se dá vnímat jako „vzdálenost“ počtu synů od krajních hodnot $a$ a $b$, ale s horním omezením $c$.

**Pozorování** (1)**.** *Pro vnitřní vrchol stromu $v$ různý od kořene platí*

1. *$b(v) \leq c$;*
2. *když $\rho(v) = a$ nebo $\rho(v) = b$, pak $b(v) = 0$;*
3. *když $\rho(v) = a - 1$ nebo $\rho(v) = b + 1$, pak $b(v) = -1$;*
4. *když $\rho(v) = 2a - 1$, pak $b(v) = c$;*
5. *Když $v'$ a $v''$ jsou dva různé vrcholy stromu různé od kořene takové, že $\rho(v') = \lceil \frac{b+1}{2} \rceil$ a $\rho(v'') = \lfloor \frac{b+1}{2} \rfloor$, pak $b(v') + b(v'') \geq 2c - 1$;*
6. *pro kořen $r$ platí $b(r) \leq c$.*

*Důkaz.* Důkazy 1,2,3,6 jsou triviální. Důkazy 4 a 5 jsou technické a vyplývají z definice $c$ (pozn. studenta - já to tam nevidím :( ) □

---

**Definice.** *Strom $(T, r)$ ohodnotíme*

$$b_h(T) = \sum \{b(v) \mid v \neq r \text{ vnitřní vrchol stromu ve výšce } h\}$$
$$b(T) = \sum_{h=1}^{\infty} b_h(T) + b(r).$$

**Definice.** *Řekneme, že $(T, r, v)$ je* parciální $(a, b)$-strom*, když $r$ je kořen stromu, $v$ je vnitřní vrchol $T$ a platí:*

- *když $v \neq r$, pak $a - 1 \leq \rho(v) \leq b + 1$ a $2 \leq \rho(r) \leq b$;*

- *když $v = r$, pak $2 \leq \rho(r) \leq b + 1$;*
- *když $t$ je vnitřní vrchol $T$ různý od $v$ a $r$, pak*

$$a \leq \rho(t) \leq b;$$

- *všechny cesty z kořene $r$ do nějakého listu mají stejnou délku.*

Tj. jde o a,b strom s jedním „pokaženým“ vrcholem.

Nyní rozložíme operace **INSERT** a **DELETE** do jednotlivých akcí se stromem a vyšetříme vliv těchto akcí na jeho ohodnocení. Důkazy lemmat jsou založené na následujícím pozorování

**Pozorování** (2)**.** *Mějme dva stromy $T$ a $T'$, které mají stejnou množinu vrcholů ve výšce $h$. Pak platí:*

1. *když každý vrchol ve výšce $h$ má stejný počet synů v obou stromech, pak $b_h(T) = b_h(T')$;*
2. *když všechny vrcholy ve výšce $h$ až na jeden vrchol mají stejný počet synů v obou stromech a počet synů u zbylého vrcholu se ve stromech $T$ a $T'$ liší nejvýše o 1, pak $b_h(T) \geq b_h(T') - 1$.*

**Lemma 1.** *Když $(T, r)$ je $(a, b)$-strom a když strom $T'$ vznikne z $T$ přidáním/ubráním jednoho syna vrcholu $v$ ve výšce 1 (tj. přidávaný/ubíraný syn je list), pak $(T', r, v)$ je parciální $(a, b)$-strom a platí*

$$\begin{aligned} b_1(T') \geq & b_1(T) - 1 \quad \text{a } b_h(T') = b_h(T) \text{ pro } h > 1; \\ b(T') \geq & b(T) - 1. \end{aligned}$$

*Důkaz.* Zhoršili jsme $b_1$ v nejspodnější hladině, na nic jiného jsme nešahali, nic jsme neštěpili. □

---

**Lemma 2.** *Nechť $(T, r, v)$ je parciální $(a, b)$-strom, $\rho(v) = b + 1$ a $v$ je ve výšce $l \geq 1$. Když $T'$ vznikne z $T$ operací* **Štěpení**$(v)$*, pak $(T', r, \text{otec}(v))$ je parciální $(a, b)$-strom a platí:*

$$\begin{aligned} b_l(T') \geq & b_l(T) + 2c, \; b_{l+1}(T') \geq b_{l+1}(T) - 1 \\ b_h(T') = & b_h(T) \text{ pro } h \neq l, l + 1; \quad b(T') \geq b(T) + 2c - 1. \end{aligned}$$

*Důkaz.* Nerovnítko o $l$-té hladině platí kvůli bodu 5 z pozorování 1, protože jsme $b$ z $-1$ změnili na $2c - 1$, o $l + 1$ hladině to platí proto, protože se zvětšil uzel o jedna (viz pozorování 2). Zbytek stromu je stejný. □

---

**Lemma 3.** *Nechť $(T, r, v)$ je parciální $(a, b)$-strom, $\rho(v) = a - 1$, $v$ je ve výšce $l \geq 1$ a $y$ je bezprostřední bratr $v$ takový že $\rho(y) = a$. Když $T'$ vznikne z $T$ operací* **Spojení**$(v, y)$*, pak $(T', r, \text{otec}(v))$ je parciální $(a, b)$-strom a platí:*

$$\begin{aligned} b_l(T') \geq & b_l(T) + c + 1, \; b_{l+1}(T') \geq b_{l+1}(T) - 1 \\ b_h(T') = & b_h(T) \text{ pro } h \neq l, l + 1; \quad b(T') \geq b(T) + c. \end{aligned}$$

*Důkaz.* Ve vrstvě $l$ se ze situace v bodech 2 a 3 z pozorování 1 dostaneme do situace v bodu 4.

Ve vrstvě $l + 1$ opět měníme jenom jeden prvek. □

---

**Lemma 4.** *Nechť $(T, r, v)$ je parciální $(a, b)$-strom, $\rho(v) = a - 1$, $v$ je výšce $l \geq 1$ a $y$ je bezprostřední bratr $v$ takový, že $\rho(y) > a$. Když $T'$ vznikne z $T$ operací* **Přesun**$(v, y)$*, pak $(T', r)$ je $(a, b)$-strom a platí:*

$$b_l(T') \geq b_l(T) \text{ a } b_h(T') = b_h(T) \text{ pro } h \neq l; \quad b(T') \geq b(T).$$

*Důkaz.* Nevím, ze kterého pozorování přesně to platí :( ale ... asi to platí, protože pokud jednu vzdálenost zhoršíme, jinou zlepšíme. Nebo tak něco. □

---

**Definice.** *Nechť $\mathcal{P}$ je posloupnost $n$ operací* **INSERT** *a* **DELETE***, aplikujme ji na prázdný $(a, b)$-strom. Označme*

- *$St_h$ – počet* **Štěpení** *ve výšce $h$ při aplikaci $\mathcal{P}$, $St = \sum_h St_h$*
- *$Sp_h$– počet* **Spojení** *ve výšce $h$ při aplikaci $\mathcal{P}$, $Sp = \sum_h Sp_h$*
- *$P_h$ – počet* **Přesunů** *ve výšce $h$ při aplikaci $\mathcal{P}$, $P = \sum_h P_h$.*

**Definice.** *Položme si $St_0 + Sp_0 =$* počet listů v $T_k \leq n$*, aby nám vycházely vzorečky dále (v 0 - tj. listové - hladině jinak nic neštěpíme/nespojujeme a $St$ či $Sp$ tam nedává smysl)*

**Definice.** *Označme $T_k(a, b)$-strom vzniklý provedením posloupnosti $\mathcal{P}$ na prázdný $(a, b)$-strom.*

Sečtením předchozích výsledků dostáváme

**Důsledek 5.** *Když položíme*

$$St_0 + Sp_0 = \text{počet listů v } T_k \leq n, \quad \text{pak}$$
$$b_h(T_k) \geq 2cSt_h + (c+1)Sp_h - St_{h-1} - Sp_{h-1} \text{ pro } h \geq 1.$$

*Důkaz.* Skutečně plyne z lemmat z částí pro hladiny, různé operace přispějí různě. □

---

**Důsledek 6.** *Dále $b(T_k) \geq (2c-1)St + cSp - n$, kde $n$ je délka posloupnosti $\mathcal{P}$.*

*Důkaz.* Skutečně plyne z lemmat z částí pro celé stromy.

□

---

Nyní odhadneme shora $b(T_k)$.

**Lemma 7.** *Když $T$ je $(a, b)$-strom s $m$ listy, pak $0 \leq b(T) \leq c + (m-2)\frac{c}{a+c-1}$.*

*Důkaz.* Pro $0 \leq j < c$ označme $m_j$ počet vnitřních vrcholů různých od kořene, které mají přesně $a + j$ synů, a $m_c$ označme počet vnitřních vrcholů různých od kořene, které mají alespoň $a+c$ synů. Když vrchol $v$ má $a + j$ synů, pak $b_T(v) \leq j$ a pro každý vnitřní vrchol $v$ platí $b_T(v) \leq c$. Tedy $b(T) \leq c + \sum_{j=0}^{c} jm_j$. Z vlastností stromů plyne

$$2 + \sum_{j=0}^{c}(a+j)m_j \leq \sum\{\rho(v) \mid v \text{ je vnitřní vrchol } T\} =$$

$$m + \sum_{j=0}^{c} m_j.$$

Odtud plyne

$$\sum_{j=0}^{c}(a+j-1)m_j \leq m - 2.$$

Protože $\frac{j}{a+j-1} \leq \frac{c}{a+c-1}$ pro každé $j$ takové, že $0 \leq j \leq c$, dostáváme

$$b(T) \leq c + \sum_{j=0}^{c} jm_j = c + \sum_{j=0}^{c} \frac{j}{a+j-1}(a+j-1)m_j \leq$$
$$c + \frac{c}{a+c-1}(m-2)$$

a lemma je dokázáno. □

---

**Věta** (1)**.**

$$P \leq n \quad \text{a} \quad (2c-1)St + cSp \leq n + c + \frac{c(n-2)}{a+c-1};$$

*Důkaz.* Protože každá operace **DELETE** použije nejvýše jednu operaci **Přesun** (a operace **INSERT** operaci **Přesun** nepoužívá) dostáváme, že

$$P \leq \text{počet operací } \textbf{DELETE} \leq n$$

a první nerovnost platí. Abychom dokázali druhou nerovnost, spojíme druhé tvrzení v Důsledku 5 a Lemma 7 ($T_k$ má nejvýše $n$ listů)

$$(2c-1)St + cSp - n \leq b(T_k) \leq c + (n-2)\frac{c}{a+c-1}$$

Odtud plyne požadovaná nerovnost. □

---

**Lemma 8.** *Pro každé $h \geq 1$ a pro každý $(a,b)$-strom $T$ s $m$ listy platí*

$$\sum_{l=1}^{h} b_l(T)(c+1)^l \leq (c+1)m.$$

*Důkaz.* Pro $0 \leq j < c$ a pro libovolné $h$ označme $m_j(h)$ počet vrcholů ve výšce $h$ různých od kořene, které mají přesně $a + j$ synů, a $m_c(h)$ počet vrcholů ve výšce $h$ různých od kořene, které mají alespoň $a + c$ synů. Pak máme

$$b_h(T) \leq \sum_{j=0}^{c} j m_j(h),$$
$$\sum_{j=0}^{c} (a+j) m_j(h) \leq \sum_{j=0}^{c} m_j(h-1) \text{ pro každé } h \geq 1,$$

kde dodefinováváme $\sum_{j=0}^{c} m_j(0) = m$. Tyto vztahy použijeme v následujícím odhadu. Platí

$$\begin{aligned}
\sum_{l=1}^{h} b_l(T)(c+1)^l &\leq \sum_{l=1}^{h} \big[(c+1)^l \big( \sum_{j=0}^{c} j m_j(l) \big) \big] \leq \\
&\sum_{l=1}^{h} \big[(c+1)^l \big( \sum_{j=0}^{c} m_j(l-1) - a \sum_{j=0}^{c} m_j(l) \big) \big] = \\
&(c+1) \sum_{j=0}^{c} m_j(0) - (c+1)^h a \sum_{j=0}^{c} m_j(h) + \\
&\sum_{l=1}^{h-1} (c+1)^{l+1} \big( \sum_{j=0}^{c} m_j(l) - \frac{a}{c+1} \sum_{j=0}^{c} m_j(l) \big) \leq \\
&(c+1)m,
\end{aligned}$$

kde rovnost jsme získali přerovnáním sčítanců tak, aby výrazy $\sum_{j=0}^{c} m_j(l)$ byly u sebe, a poslední nerovnost plyne z toho, že $\frac{a}{c+1} \geq 1$, a tedy druhý sčítanec v předchozím výrazu není kladný. □

---

**Lemma 9.** $St_h + Sp_h \leq \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k) \frac{(c+1)^l}{(c+1)^{h+1}}$

*Důkaz.* Výraz z Důsledku 5 upravíme (využíváme, že $c \geq 1$):

$$\begin{aligned}
St_h + Sp_h \leq & \frac{b_h(T_k)}{c+1} + \frac{St_{h-1} + Sp_{h-1}}{c+1} \leq \\
& \frac{b_h(T_k)}{c+1} + \frac{b_{h-1}(T_k)}{(c+1)^2} + \frac{St_{h-2} + Sp_{h-2}}{(c+1)^2} \leq \cdots \leq \\
& \sum_{i=0}^{h-1} \frac{b_{h-i}(T_k)}{(c+1)^{i+1}} + \frac{n}{(c+1)^h} = \\
& \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k) \frac{(c+1)^l}{(c+1)^{h+1}}.
\end{aligned}$$

□

---

**Věta** (2)**.** $St_h + Sp_h + P_h \leq \frac{2(c+2)n}{(c+1)^h}$.

*Důkaz.* Zkombinujeme předchozí dvě lemmata. Dostaneme

$$St_h + Sp_h \leq \frac{n}{(c+1)^h} + \sum_{l=1}^{h} b_l(T_k)\frac{(c+1)^l}{(c+1)^{h+1}} \leq \frac{n}{(c+1)^h} + \frac{n(c+1)}{(c+1)^{h+1}} = \frac{2n}{(c+1)^h}.$$

Protože $P_h \leq Sp_{h-1} - Sp_h \leq St_{h-1} + Sp_{h-1} \leq \frac{2n}{(c+1)^{h-1}}$ dostáváme, že

$$St_h + Sp_h + P_h \leq \frac{2n}{(c+1)^h} + \frac{2n}{(c+1)^{h-1}} = \frac{2n + 2n(c+1)}{(c+1)^h} = \frac{2n(c+2)}{(c+1)^h}$$

□

---

**Důsledek.** *Amortizovaný počet vyvažovacích operací splňuje*

$$\frac{P + St + Sp}{n} \leq \frac{5}{2}.$$

*Důkaz.* Z definice plyne, že $c \geq 1$, a protože $a \geq 2$, z věty (1) dostaneme

$$St + Sp \leq \frac{n}{c} + 1 + \frac{n-2}{a} \leq n + 1 + \frac{n-2}{2} \leq \frac{3n}{2}.$$

Amortizovaný počet vyvažovacích operací splňuje tedy

$$\frac{P + St + Sp}{n} \leq \frac{5}{2}.$$

□

---

### 4.2.16 Omezení štěpení, spojování a přesunů – diskuze

Věta vysvětluje, proč jsou doporučené hodnoty $b \geq 2a$ – pak je počet vyvažovacích operací během posloupnosti operací **INSERT** a **DELETE** lineární vzhledem k délce této posloupnosti. Pro $b = 2a - 1$ lze lehce nalézt posloupnost operací **INSERT** a **DELETE** o délce $n$ takovou, že její aplikace na prázdný $(a, b)$-strom vyžaduje počet vyvažovacích operací úměrný $n \log n$ (pro každé dostatečně velké $n$). Podobná věta platí i pro paralelní implementaci $(a, b)$-stromů, ale platí za předpokladu $b \geq 2a + 2$. Pro $b = 2a$ nebo $b = 2a + 1$ lze nalézt posloupnost, která je protipříkladem. Proto se doporučuje hodnota $b = 2a + 2$ pro paralelní implementaci $(a, b)$-stromu. Pro propojené $(a, b)$-stromy platí silnější verze.

**Věta.** *Předpokládejme, že $b \geq 2a$ a $a \geq 2$. Mějme hladinově propojený $(a, b)$-strom s prstem $T$, který reprezentuje $n$-prvkovou množinu. Pak posloupnost $\mathcal{P}$ operací* **MEMBER**, **INSERT**, **DELETE** *a* **PRST** *aplikovaná na $T$ vyžaduje čas*

$$O(\log(n) + \text{ čas na vyhledání prvků}).$$

Vysvětlení: Začínáme v libovolném propojeném $(a,b)$-stromě $T$, proto jeho struktura může být nevýhodná pro danou posloupnost operací $\mathcal{P}$. Abychom se dostali do vhodného režimu, může být třeba až $\log(n)$ vyvažovacích operací. Čas na vyhledávání nemůžeme ovlivnit, ten musí ovlivnit uživatel.

Aplikace: analýza hladinově propojených stromů s prstem umožnila návrh algoritmu, který pro dvě množiny $S_1$ a $S_2$ reprezentované propojenými $(a,b)$-stromy, kde $b \geq 2a$ a $a \geq 2$, zkonstruuje propojený $(a,b)$-strom reprezentující množinu $S_1 \cup S_2$ (nebo množinu $\Delta(S_1,S_2) = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$ nebo $S_1 \cap S_2$ nebo $S_1 \setminus S_2$) v čase $O(\log \binom{n+m}{m})$, kde $n = \max\{|S_1|,|S_2|\}$ a $m = \min\{|S_1|,|S_2|\}$. Detaily budou v letním semestru.

Vyvažování při operaci **INSERT** lze provádět tak, že operace **Štěpení**$(t)$ se provede, jen když oba bratři vrcholu $t$ mají $b$ synů. Jinak se provádí operace **Přesun**. Nevím o žádném seriózním pokusu tyto alternativy porovnat.

## 4.3 Binární vyhledávací stromy

Binární vyhledávací strom je struktura pro binární vyhledávání v uspořádaném poli roztaženém do roviny a vyhledávání odpovídá cestě ve stromě.

### 4.3.1 Formální definice

Předpokládáme, že $U$ je lineárně uspořádané univerzum a $S \subseteq U$. *Binární vyhledávací strom $T$ reprezentující* množinu $S$ je úplný binární strom (tj. každý vrchol je buď listem nebo má dva syny, levého a pravého), kde existuje bijekce mezi množinou $S$ a vnitřními vrcholy stromu taková, že

- když $v$ je vnitřní vrchol stromu $T$, kterému je přiřazen prvek $s \in S$, pak každému vnitřnímu vrcholu $u$ v podstromu levého syna vrcholu $v$ je přiřazen prvek z $S$ menší než $s$ a každému vnitřnímu vrcholu $w$ v podstromu pravého syna vrcholu $v$ je přiřazen prvek z $S$ větší než $s$.

Strukura vnitřního vrcholu $v$:

- ukazatel otec$(v)$ na otce vrcholu $v$
- ukazatel levy$(v)$ na levého syna vrcholu $v$
- ukazatel pravy$(v)$ na pravého syna vrcholu $v$
- atribut key$(v)$ – prvek z $S$ přiřazený vrcholu $v$.

Když $v$ je kořen stromu, pak hodnota ukazatele otec$(v)$ je $NIL$. List má ukazatele pouze na otce.

Každý list reprezentuje interval mezi dvěma sousedními prvky z $S$ – přesně, když $u$ je list a je levým synem vrcholu $v$, nalezneme vrchol na cestě z $u$ do kořene nejblíže $u$ takový, že je pravým synem vrcholu $w$. Pak $u$ reprezentuje interval $(\text{key}(w), \text{key}(v))$ a když vrchol $w$ neexistuje, pak $u$ reprezentuje interval $(-\infty, \text{key}(v))$ a prvek key$(v)$ je nejmenší prvek v $S$. Když $u$ je list a je pravým synem vrcholu $v$, nalezneme vrchol na cestě z $u$ do kořene nejblíže $u$ takový, že je levým synem vrcholu $w$. Pak $u$ reprezentuje interval $(\text{key}(v), \text{key}(w))$ a když takový vrchol $w$ neexistuje, pak $u$ reprezentuje interval $(\text{key}(v), +\infty)$ a prvek key$(v)$ je největší prvek v $S$.

Při implementaci binárních vyhledávácích stromů je výhodné vynechat listy (místo nich bude ukazatel $NIL$). Při návrhu algoritmů je však naopak výhodné pracovat s listy (vyhlíží to logičtější). Proto při návrhu algoritmů budeme předpokládat, že stromy mají listy reprezentující intervaly.

### 4.3.2 Algoritmy

Navrhneme algoritmy pro binární vyhledávací stromy realizující operace z uspořádaného slovníkového problému.

**Vyhledej**($x$)
$t$ :=kořen stromu
**while** $t$ není list a key($t$) $\neq x$ **do**
  **if** key($t$) $> x$ **then** $t$ := levy($t$) **else** $t$ := pravy($t$) **endif**
**enddo**

**MEMBER**($x$)
**Vyhledej**($x$)
**if** $t$ není list **then** **Výstup**: $x \in S$ **else** **Výstup**: $x \notin S$ **endif**

**INSERT**($x$)
**Vyhledej**($x$)
**if** $t$ je list **then**
$t$ se změní na vnitřní vrchol, key($t$) := $x$,
levy($t$) a pravy($t$) jsou nové listy, jejichž otcem je $t$
**endif**

**DELETE**($x$)
**Vyhledej**($x$)
**if** $t$ není list **then**
  **if** levy($t$) je list **then**
    odstraníme vrchol levy($t$), otec(pravy($t$)) := otec($t$)
    **if** $t$ = levy(otec($t$)) **then**
      levy(otec($t$)) := pravy($t$)
    **else**
      pravy(otec($t$)) := pravy($t$)
    **endif**
    odstraníme vrchol $t$
  **else**
    $u$ := levy($t$)
    **while** pravy($u$) není list **do**
      $u$ := pravy($u$)
    **enddo**
    key($t$) := key($u$), odstraníme vrchol pravy($u$),
    otec(levy($u$)) := otec($u$)
    **if** $u$ = levy(otec($u$)) **then**
      levy(otec($u$)) := levy($u$)
    **else**
      pravy(otec($u$)) := levy($u$)
    **endif**
    odstraníme vrchol $u$
  **endif**
**endif**

**MIN**
$t$ :=kořen stromu
**while** levý syn $t$ není list **do** $t :=$ levy$(t)$ **enddo**
**Výstup**: prvek reprezentovaný $t$ je nejmenší prvek v $S$

**MAX**
$t$ :=kořen stromu
**while** pravý syn $t$ není list **do** $t :=$ pravy$(t)$ **enddo**
**Výstup**: prvek reprezentovaný $t$ je největší prvek v $S$

**SPLIT**$(x)$:
$T_1$ a $T_2$ jsou prázdné stromy
$u_1 := u_2 := NIL$
$t :=$ kořen stromu $T$
**while** $t$ není list a key$(t) \neq x$ **do**
  **if** key$(t) > x$ **then**
    $u :=$ levy$(t)$, levy$(t) := NIL$, otec$(u) := NIL$
    **if** $T_2$ je prázdný strom **then**
      $T_2 :=$ podstrom vrcholu $t$
    **else**
      levy$(u_2) := t$, otec$(t) := u_2$
    **endif**
    $u_2 := t$
  **else**
    $u :=$ pravy$(t)$, pravy$(t) := NIL$, otec$(u) := NIL$
    **if** $T_1$ je prázdný strom **then**
      $T_1 :=$ podstrom vrcholu $t$
    **else**
      pravy$(u_1) := t$, otec$(t) := u_1$
    **endif**
    $u_1 := t$
  **endif**
  $t := u$
**enddo**
**if** key$(t) = x$ **then**
  otec(levy$(t)$) $:= u_1$, pravy$(u_1) :=$ levy$(t)$
  otec(pravy$(t)$) $:= u_2$, levy$(u_2) :=$ pravy$(t)$
  otec$(u_1) := NIL$, otec$(u_2) := NIL$, **Výstup**: $x \in S$
**else**
  **Výstup**: $x \notin S$
**endif**
Komentář: $T_1$ je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s < x\}$ a $T_2$ je binární vyhledávací strom reprezentující množinu $\{s \in S \mid s > x\}$.

**JOIN3**$(T_1, x, T_2)$ – předpokládáme, že když $T_i$ reprezentuje množinu $S_i$ pro $i = 1, 2$, pak $\max S_1 < x < \min S_2$
vytvořme nový vrchol $u$, key$(u) = x$, otec$(u) := NIL$,
otec(kořene $T_1$) $:= u$, otec(kořene $T_2$) $:= u$,
levy$(u) :=$kořen $T_1$, pravy$(u) :=$kořen $T_2$.

### 4.3.3 Korektnost

Abych dokázali korektnost algoritmu **Vyhledej** – jedná se o modifikaci vyhledávání v uspořádaném poli – popíšeme podrobněji vlastnosti binárního vyhledávacího stromu.

Nejprve rozšíříme universum o dva nové prvky, o nový nejmenší prvek $-\infty$ a o nový největší prvek $+\infty$.

Mějme binární vyhledávací strom $T$ reprezentující množinu $S$, pak pro vrchol $t$ stromu $T$ definujeme indukcí hodnoty $\lambda(t)$ a $\pi(t)$. Když $r$ je kořen, pak $\lambda(r) = -\infty$ a $\pi(r) = +\infty$. Když hodnoty $\lambda(t)$ a $\pi(t)$ jsou pro vrchol $t$ definovány, pak pro levého syna $u$ vrcholu $t$ definujeme $\lambda(u) = \lambda(t)$ a $\pi(u) = \text{key}(y)$ a pro pravého syna $w$ vrcholu $t$ definujeme $\lambda(w) = \text{key}(t)$ a $\pi(w) = \pi(t)$.

Pozn. studenta - nejspíš Lambda jako Levý, Pi jako Pravý.

Nyní dokážeme

**Lemma.** *Je-li $T'$ podstrom binárního vyhledávacího stromu $T$ určený vrcholem $t$, pak $T'$ reprezentuje množinu $S \cap (\lambda(t), \pi(t))$. Navíc interval $(\lambda(t), \pi(t))$ je největší interval, který obsahuje jenom prvky z $S$, které jsou reprezentovány vrcholy podstromu $T'$. Navíc, když $t$ je list, pak $< \lambda(t), \pi(t) >$ je interval repreyentovaný listem $t$.*

*Důkaz.* Tvrzení dokážeme indukcí. Zřejmě platí, když $t$ je kořen stromu $T$. Předpokládejme, že platí pro vrchol $t$ a dokážeme ho pro syny vrcholu $t$. Označme $t_l$ levého syna vrcholu $t$, $t_p$ pravého syna vrcholu $t$. Z definice binárního vyhledávacího stromu stromu plyne, že když $u$ je vnitřní vrchol v podstromu $T$ určeném vrcholem $t_l$ a když $v$ je vnitřní vrchol v podstromu $T$ určeném vrcholem $t_p$, pak $\text{key}(u) < \text{key}(t) < \text{key}(v)$. Nyní platnost tvrzení pro $t$ implikuje platnost tvrzení i pro vrcholy $t_l$ a $t_p$. □

---

Korektnost podprocedury **Vyhledej** plyne z následujícího invariantu:

**Lemma.** *Když při vyhledávání $x$ vyšetřujeme vrchol $t$, pak*

$$\lambda(t) < x < \pi(t).$$

Toto tvrzení se lehce dokáže indukcí z popisu algoritmu **Vyhledej**. Tedy operace **Vyhledej** je korektní a korektnost operací **MEMBER** a **INSERT** je teď zřejmá.

V operaci **DELETE**, když $\text{levy}(t)$ je list, pak korektnost je zřejmá. Když $\text{levy}(t)$ není list, pak algoritmus nalezne list $v$ takový, že $\pi(v) = x$. Pak pro $u = \text{otec}(v)$ platí $v = \text{pravy}(u)$ a $\lambda(v) = \text{key}(u)$ a $(\lambda(v), \pi(v)) \cap S = \emptyset$. Když $y = \text{key}(u)$, pak odstranění vrcholů $u$ a $v$ dává binární vyhledávací strom reprezentující $S \setminus \{y\}$. Protože $(y, x) \cap S = \emptyset$, tak příkaz $\text{key}(t) := y$ dává binární vyhledávací strom reprezentující $S \setminus \{x\}$ a proto operace **DELETE** je korektní.

Korektnost operací **MIN**, **MAX** a **JOIN3** plyne z definice binárního vyhledavacího stromu.

Korektnost operace **SPLIT** plyne z korektnosti algoritmu **Vyhledej** a z faktu, že $u_1$ je otec nejpravějšího listu stromu $T_1$ a $u_2$ je otec nejlevějšího listu stromu $T_2$.

Protože ke stromu $T_1$ se přidává část stromu $T$ reprezentující prvky, které jsou větší než prvky reprezentované v $T_1$, a ke stromu $T_2$ se přidává část stromu $T$ reprezentující prvky, které jsou menší než prvky reprezentované v $T_2$, korektnost algoritmu pro operaci **SPLIT** je jasná.

### 4.3.4 Časová složitost

Zpracování jednoho vrcholu vyžaduje čas $O(1)$ a algoritmus se pohybuje po jedné cestě z kořene do nějakého listu. Označme hloubka($T$) délku nejdelší cesty z kořene do nějakého listu. Pak dostáváme

**Věta.** *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *v binárním vyhledávacím stromě $T$ vyžadují čas $O$*(hloubka($T$)).

### 4.3.5 Pořádková statistika

Bohužel ani struktura binárních vyhledávacích stromů nepodporuje efektivní implementaci operace **ord**($k$). Pro její efektivní implentaci je vhodné rozšířit datovou strukturu tak, že u každého vrcholu $t$ je deklarován také údaj $p(t)$ – počet listů v podstromu určeném vrcholem $t$. Po provedení operací **INSERT**, **DELETE**, **JOIN3** a **SPLIT** je pak nutné aktualizovat tuto položku na cestě z vrcholu do kořene. Následující algoritmus pak realizuje operaci **ord**($k$).

```
ord(k)
t :=kořen stromu
if k ≥ p(t) then k-tý prvek neexistuje, stop endif
while true do
  if k > p(levy(t)) then
    k := k − p(levy(t)), t := pravy(t)
  else
    if k < p(levy(t) then
      t := levy(t)
    else
      key(t) je k-tý prvek reprezentované množiny, stop
    endif
  endif
enddo
```

Korektnost algoritmu plyne z následujícího invariantu: Když algoritmus má v daném okamžiku v proměnné $t$ vrchol $v$ a hodnota proměnné $k$ je $k'$, pak $k$-tý prvek v $S$ se rovná $k'$-tému prvku v intervalu reprezentovaném v podstromu stromu $T$ určeném vrcholem $v$. Protože na počátku algoritmu je $v$ kořen stromu a interval je $S$ (a $k' = k$), tak na počátku běhu algoritmu invariant platí. Předpokládejme, že platí v některém kroku. Nechť $u$ je levý syn $v$, $w$ je pravý syn $v$ a $I_a$ je interval reprezentovaný podstromem $T$ určeným vrcholem $a$. Pak $|I_u| = p(u) - 1$, $\max I_u < \text{key}(v) < \min I_w$ a $I_v = I_u \cup \{\text{key}(v)\} \cup I_w$. Odtud plyne, že když $k' < p(u)$, pak $k'$-tý prvek v intervalu $I_v$ je $k'$-tý prvek v intervalu $I_u$, když $k' > p(u)$, pak $k'$-tý prvek v intervalu $I_v$ je $(k' - p(u))$-tý prvek v intervalu $I_w$, a když $k' = p(u)$, pak $k'$-tý prvek v intervalu $I_v$ je key($v$). Odtud plyne invariant a korektnost algoritmu. Podle stejných argumentů jako v předchozím případě dostaneme, že časová složitost algoritmu je $O$(hloubka($T$)). Tedy můžeme tato fakta shrnout.

**Věta.** *Algoritmy pro operace* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3**, **SPLIT** *a* **ord**($k$) *pro všechna $k$ v rozšířených binárních vyhledávacích stromech vyžadují čas $O$*(hloubka($T$)), *kde $T$ je reprezentující strom.*

### 4.3.6 Diskuze

Tento výsledek motivuje používání binárních vyhledávacích stromů, které splňují další podmínku, která má zajistit, že hloubka($T$) = $O(\log |S|)$. V takovémto případě mluvíme o *vyvážených binárních vyhledávacích stromech.* Je však nutné přidat k operacím **INSERT**, **DELETE**, **JOIN3** a **SPLIT** další kroky,

které zaručí, že po jejich provedení strom opět splňuje požadované podmínky. To vede k požadavku, aby vyvažovací operace byly rychlé a provádělo se jich málo.

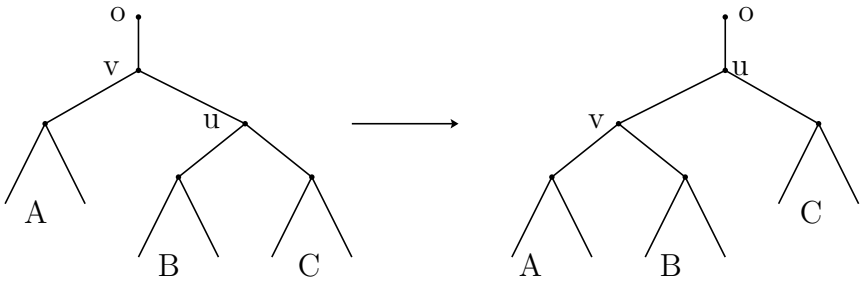
Při náhodné posloupnosti operací **INSERT** a **DELETE** je velká pravděpodobnost, že dostaneme náhodný binární vyhledávací strom. Je známo, že očekávaná hodnota proměnné hloubka($T$) je $O(\log |S|)$. Protože se nepoužívají vyvažovací operace, můžeme dostat lepší výsledek (časově) než pro vyvážené binární vyhledávací stromy. Tento problém se teď intenzivně studuje. Velká pozornost je věnována pravděpodobnostním modifikacím binárních vyhledávacích stromů. Hledají se však i další možnosti.

Studují se tzv. samoupravující struktury. Zde se pracuje s datovou strukturou bez dodatečných informací, ale operace nad touto strukturou provádí vyvažování v závislosti na argumentu operace. Dokázalo se, že existuje strategie vyvažování, která zajišťuje dobré chování bez ohledu na vstupní data. Další strategie je, že se jen zjišťuje, zda datová struktura nemá výrazně špatné chování, a pokud ho má nebo po dlouhé řadě úspěšných aktualizačních operací se vybuduje nová datová struktura (s optimálním chováním). Třetí, poměrně stará, strategie je založena na předpokladu, že známe rozdělení vstupních dat. Zde se datová struktura předem upravuje pro toto rozdělení. Ukazuje se, že tyto strategie mají úspěch. Další podrobnosti v letním semestru.

### 4.3.7 Rotace

Nyní si ukážme dvě operace se stromy, na nichž jsou založeny vyvažovací operace pro binární vyhledávací stromy. Obě operace vyžadují čas $O(1)$.

Mějme vrchol $v$ binárního vyhledávacího stromu $T$ a jeho syna $u$, který je vnitřní vrchol. Pak **Rotace**$(v, u)$ je znázorněna na obrázku 1 a provádí ji následující algoritmus.



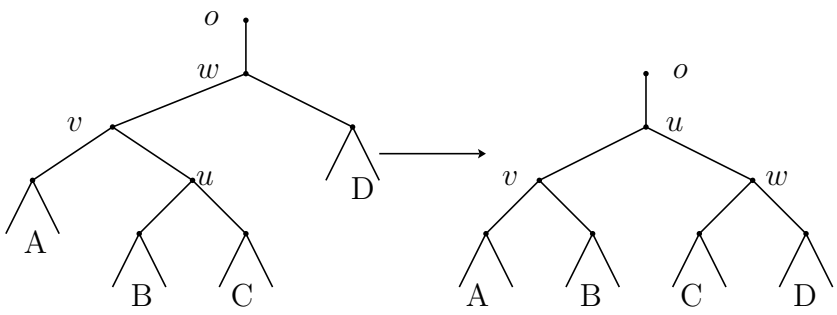

Obrázek 1: Rotace $(v, u)$

**Rotace**$(v, u)$
otec$(u)$ := otec$(v)$,
**if** $v$ = levy(otec$(v)$) **then**
  levy(otec$(v)$) := $u$
**else**
  pravy(otec$(v)$) := $u$
**endif**
otec$(v)$ := $u$
**if** $u$ = levy$(v)$ **then**
  otec(pravy$(u)$) := $v$, levy$(v)$ := pravy$(u)$, pravy$(u)$ := $v$
**else**
  otec(levy$(u)$) := $v$, pravy$(v)$ := levy$(u)$, levy$(u)$ := $v$
**endif**

Všimněme si, že při **Rotace** můžeme aktualizovat i funkci $p$. Pro vrchol $w \neq u, v$ se její hodnota nemění,

nová hodnota $p(u)$ je rovná původní hodnotě $p(v)$ a novou hodnotu $p(v)$ dostaneme jako $p(\text{levy}(v)) + p(\text{pravy}(v))$.

Mějme vrchol $w$ stromu $T$, jeho syna $v$ a jeho syna $u$ takového, že $u$ není list a $v$ je pravý syn vrcholu $w$, právě když $u$ je levý syn vrcholu $v$. Pak **Dvojita-rotace**$(w, v, u)$ je znázorněna na obrázku a provádí ji následující algoritmus.




Obrázek 2: Dvojita-rotace$(w, v, u)$

**Dvojita-rotace**$(w, v, u)$
otec$(u)$ := otec$(w)$
**if** $w$ = levy(otec$(w)$) **then**
  levy(otec$(w)$) := $u$
**else**
  pravy(otec$(w)$) := $u$
**endif**
otec$(v)$ := $u$, otec$(w)$ := $u$
**if** $v$ = levy$(w)$ **then**
  levy$(w)$ := pravy$(u)$, otec(pravy$(u)$) := $w$, pravy$(v)$ := levy$(u)$
  otec(levy$(u)$) := $v$, levy$(u)$ := $v$, pravy$(u)$ := $w$
**else**
  pravy$(w)$ := levy$(u)$, otec(levy$(u)$) := $w$, levy$(v)$ := pravy$(u)$
  otec(pravy$(u)$) := $v$, levy$(u)$ := $w$, pravy$(u)$ := $v$
**endif**

Také zde můžeme v čase $O(1)$ spočítat nové hodnoty $p$. Pro vrchol $x \neq u, v, w$ se hodnota nemění, nová hodnota $p(u)$ je rovná původní hodnotě $p(w)$ a nové hodnoty $p(v)$ a $p(w)$ získáme podle stejného vzorce jako v **Rotace**.

Další kapitoly by technicky měly patřit pod binární stromy, pro přehlednost jsem je ale nechal ve vlastních kapitolách.

## 4.4 AVL-stromy

### 4.4.1 Definice

Binární vyhledávací strom je *AVL-strom*, když pro každý vnitřní vrchol $v$ se délka nejdelší cesty z jeho levého syna do listu a délka nejdelší cesty z jeho pravého syna do listu liší nejvýše o 1.

Pro vnitřní vrchol $v$ stromu $T$ označme $\eta(v)$ délku nejdelší cesty z vrcholu $v$ do listu.

Struktura vnitřních vrcholů v AVL-stromech je rozšířena o hodnotu $\omega$:

- $\omega(v) = -1$, když
$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v) + 1;$$
- $\omega(v) = 0$, když
$$\eta(\text{levý syn vrcholu } v) = \eta(\text{pravý syn vrcholu } v);$$
- $\omega(v) = +1$, když
$$\eta(\text{levý syn vrcholu } v) + 1 = \eta(\text{pravý syn vrcholu } v).$$

Všimněme si, že hodnota $\eta(v)$ pro vnitřní vrcholy $v$ stromu $T$ není nikde uložena. Hodnoty $\eta$ jsme schopni spočítat z hodnot $\omega$, ale není to třeba. Stačí, když po aktualizačních operacích budeme umět aktualizovat hodnoty $\omega$ a upravit binární vyhledávací strom tak, aby byl opět AVL-strom.

### 4.4.2 Odhad výšky stromu

Odhad velikosti $\eta(\text{kořen } T)$ =výška stromu v závislosti na velikosti reprezentované množiny $S$.

**Pozorování.** *Když $T$ je AVL-strom a $v$ je vnitřní vrchol $T$, pak podstrom $T$ určený vrcholem $v$ je opět AVL-strom.*

**Definice.** *Označme $mn(i)$ velikost nejmenší množiny reprezentované AVL-stromem $T$ takovým, že*
$$\eta(\text{kořen } T) = i.$$

**Definice.** *Označme $mx(i)$ velikost největší množiny reprezentované AVL-stromem $T$ takovým, že*
$$\eta(\text{kořen } T) = i.$$

**Pozorování.** *Z definice AVL-stromu plynou rekurze*
$$mn(i) = mn(i-1) + mn(i-2) + 1,\ mx(i) = 2mx(i-1) + 1,$$
$$a\ mn(1) = mx(1) = 1, \quad mn(2) = 2, \quad mx(2) = 3.$$

**Lemma** (1)**.** $mx(i) = 2^i - 1$

*Důkaz.* Tento vzorec je splněn pro $i = 1, 2$. Dále
$$mx(i+1) = 2mx(i) + 1 = 2(2^i - 1) + 1 = 2^{i+1} - 1.$$
Tím je vzorec dokázán. □

---

Abychom spočítali $mn$, připomeneme si definici Fibonacciho čísel.

**Definice.** *Fibonacciho číslo $F_i$ je definováno rekurencí*
$$F_1 = F_2 = 1 \text{ a } F_{i+2} = F_i + F_{i+1} \text{ pro všechna } i \geq 3.$$

**Lemma.** *Platí* $F_i = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^i - \left(\frac{1-\sqrt{5}}{2}\right)^i}{\sqrt{5}}$ *pro všechna* $i \geq 1$

(dokážeme si v části o haldách).

**Lemma.** *Existují konstanty* $0 < c_1 < c_2$ *takové, že*

$$c_1(\frac{1+\sqrt{5}}{2})^i < \sqrt{5}F_i < c_2(\frac{1+\sqrt{5}}{2})^i.$$

*Důkaz.* Protože $-1 < \frac{1-\sqrt{5}}{2} < 0$ a $\frac{1+\sqrt{5}}{2} > 1$, dostáváme, že

$$\lim_{n\mapsto\infty} F_n\sqrt{5}(\frac{1+\sqrt{5}}{2})^{-n} = 1.$$

Proto skutečně existují konstanty $0 < c_1 < c_2$ takové, že $c_1(\frac{1+\sqrt{5}}{2})^i < \sqrt{5}F_i < c_2(\frac{1+\sqrt{5}}{2})^i$. □

---

**Lemma** (2)**.** $mn(i) = F_{i+2} - 1$

*Důkaz.* Protože $F_3 = 2$ a $F_4 = 3$, tvrzení platí pro $i = 1$ a $i = 2$. Dále

$$\begin{aligned} mn(i+2) =& mn(i+1) + mn(i) + 1 = \\ & F_{i+3} - 1 + F_{i+2} - 1 + 1 = F_{i+4} - 1. \end{aligned}$$

Z toho indukcí plyne požadovaný vztah. □

---

**Věta.** $i = \Theta(\log(n))$

*Důkaz.* Když AVL-strom $T$ o výšce $i$ reprezentuje množinu $S$ o velikosti $n$, pak platí

$$\frac{c_1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{i+2} - 1 < F_{i+2} - 1 \leq n \leq 2^i - 1.$$

Po zlogaritmování z toho okamžitě dostáváme

$$\log(\frac{c_1}{\sqrt{5}}) + (i+2)\log(\frac{1+\sqrt{5}}{2}) < \log(n+1) < i.$$

Protože $\log(\frac{1+\sqrt{5}}{2}) \approx 0.69 \approx \frac{1}{1.44}$ dostáváme, že pro dostatečně velká $n$ platí, že $0.69i < \log(n+1) \leq i$. Odtud plyne, že $\log(n+1) \leq i \leq 1.44\log(n)$, a tedy $i = \Theta(\log(n))$. □

---

$\eta(\text{kořen } T) = \Theta(\log(n))$

### 4.4.3 Algoritmy

Operace **MEMBER**$(x)$ pro AVL-stromy je stejná jako operace **MEMBER**$(x)$ pro nevyvážené binární vyhledávací stromy. Aktualizační operace pro AVL-stromy nejprve provedou příslušnou operaci pro nevyvážené binární vyhledávací stromy a pak následuje jejich vyvažovací část.

Při úspěšně provedené operaci **INSERT**$(x)$ v nevyvážených binárních stromech změníme vhodný list $t$ na vnitřní vrchol stromu reprezentující $x$ a přidáme k $t$ dva syny, kteří budou listy. Důsledkem je, že definujeme $\omega(t) = 0$. Protože se však zvětšila hodnota $\eta(t)$ (bylo $\eta(t) = 0$ a teď je $\eta(t) = 1$), zavoláme proceduru **Kontrola-INSERT**$(t)$, která zajistí správnou hodnotu funkce $\omega$ pro otce $t$. Navíc, když zjistí, že se zvětšila hodnota $\eta$ vrcholu otce $t$, pak zavolá sama sebe na vrchol otec $t$. Nejprve provedeme analýzu situace.

Mějme vrchol $t$, jeho $\eta(t) = a$ (ale $a$ neznáme), na začátku operace **INSERT** bylo $\eta(t) = a - 1$. V podstromu určeném vrcholem $t$ máme už správné hodnoty $\omega$. Vrchol $v$ je otcem $t$, $t = \text{levy}(v)$ a $\omega(v)$ má ještě původní hodnotu.

**Lemma.** *Když se hodnota $\eta(t)$ při operaci* **INSERT** *zvětšila a $t$ nebyl listem před operaci, pak po operaci neplatí $\omega(t) = 0$.*

*Důkaz.* Skutečně, aby se zvětšila, musí se změnit z nuly na jednu ze stran. □

---

Označme $u = \text{pravy}(v)$ – tj. $v$ má děti $t$, $u$, $\eta(t)$ jsme právě změnili z $a - 1$ na $a$ a jdeme řešit $v$:

1. $\omega(v) = 1$, tj. předtím byl pravý syn $u$ větší – $\eta(u) = a$.

   Stačí položit $\omega(v) = 0$, protože synové jsou stejně velcí, a $\eta(v) = a + 1$ se nezměnilo, takže změna se nepropaguje výš.

2. $\omega(v) = 0$, tj. předtím byl pravý syn $u$ stejný – $\eta(u) = a - 1$.

   Protože jsme zvětšili levou stranu, položíme $\omega(v) = -1$.

   Protože jsme zvětšili $\eta(v) = a + 1$, musíme zavolat proceduru **Kontrola-INSERT** na vrchol $v$.

3. $\omega(v) = -1$, pak máme problém – předtím byl levý syn větší a my jsme ho ještě zvětšili (tj. $\eta(u) = a-2$ a $\eta(v) = a + 1$ se změnilo).

   Takže $\omega(v) = -2$ a to je zakázané. Označme $t_1 = \text{levy}(t)$, $t_2 = \text{pravy}(t)$ ($t$ je naposledy upravovaný vrchol) a podle toho, jaké je $\omega(t)$ budeme postupovat dál ($\omega(t) = 0$ nenastane, viz Lemma).

   (a) $\omega(t) = -1$, můžeme provést jednoduchou rotaci **Rotace**$(v, t)$, tj. $t$ půjde „nahoru“, levý syn zůstane $t_1$, pravý syn bude $v$, který bude mít za levého syna $t_2$. Pak stačí položit $\omega(v) = \omega(t) = 0$, nic nepropagujeme výš.

   (b) $\omega(t) = 1$ – jednoduchou rotaci neprovedeme, protože by se „levá větev“ příliš zkrátila. Uděláme tedy **Dvojita-rotace**$(v, t, t_2)$ – $t_2$ půjde „nahoru“, zavěsí se za něj $v$ a $t$ a rozdělí si jeho „děti“. Musíme udělat analýzu pro všechny tři případy $\omega(t_2)$.

      i. $\omega(t_2) = 1 \implies \eta(t_3) = a - 3$ a $\eta(t_4) = a - 2$ a stačí položit $\omega(t) = -1$, $\omega(v) = \omega(t_2) = 0$, protože $\eta(t_2) = a$.

      ii. $\omega(t_2) = 0 \implies \eta(t_3) = \eta(t_4) = a - 2$ a stačí položit $\omega(t_2) = \omega(v) = \omega(t) = 0$, protože $\eta(t_2) = a$.

iii. $\omega(t_2) = -1 \implies \eta(t_3) = a - 2$ a $\eta(t_4) = a - 3$ a stačí položit $\omega(v) = 1$, $\omega(t_2) = \omega(t) = 0$, protože $\eta(t_2) = a$.

Když $t$ je pravý syn $v$, pak situace je symetrická.
Popíšeme proceduru **Kontrola-INSERT** (vychází z analýzy výše)

```
Kontrola-INSERT(t)
v := otec(t)
if t = levy(v) then
  Leva-Kontrola-INSERT(t)
else
  Prava-Kontrola-INSERT(t)
endif
```

```
Leva-Kontrola-INSERT(t)
if ω(v) = 1 then
  ω(v) := 0
else
  if ω(v) = 0 then
    ω(v) := −1, t := v, Kontrola-INSERT(t)
  else
    if ω(t) = −1 then
      Rotace(v, t), ω(v) := 0, ω(t) := 0
    else
      w := pravy(t), Dvojita-rotace(v, t, w),
      if ω(w) = 0 then
        ω(t) := 0, ω(v) := 0
      else
        if ω(w) = 1 then
          ω(v) := 0, ω(t) := −1
        else
          ω(v) := 1, ω(t) := 0
        endif
      endif
      ω(w) := 0
    endif
  endif
endif
```

```
Prava-Kontrola-INSERT(t)
if ω(v) = −1 then
  ω(v) := 0
else
  if ω(v) = 0 then
    ω(v) := 1, t := v, Kontrola-INSERT(t)
  else
    if ω(t) = 1 then
      Rotace(v, t), ω(v) := 0, ω(t) := 0
    else
      w := levy(t), Dvojita-rotace(v, t, w),
      if ω(w) = 0 then
        ω(t) := 0, ω(v) := 0
      else
        if ω(w) = 1 then
          ω(v) := 0, ω(t) := −1
        else
          ω(v) := 1, ω(t) := 0
        endif
      endif
      ω(w) := 0
    endif
  endif
endif
```

Všimněme si, že po provedení **Rotace** nebo **Dvojita-rotace** vyvažování v operaci **INSERT** končí. Tedy operace **INSERT** provádí nejvýše jednu proceduru **Rotace** nebo **Dvojita-rotace**. Korektnost vyvažovací operace je založena na faktu, že když se zvětší hodnota $\eta(t)$, pak nemůže být $\omega(t) = 0$.

Popíšeme vyvažovací operaci pro operaci **DELETE**. Předpokládejme, že $t$ je vrchol, jehož otec se odstranil (tj. bratr $t$ byl list) a hodnota $\eta(t)$ je menší než byla hodnota $\eta(\text{otec}(t))$. Proto zavoláme proceduru **Kontrola-DELETE**$(t)$. Tato procedura zajistí správnou hodnotu funkce $\omega$ pro otce $t$. Navíc, když zjistí, že se zmenšila hodnota $\eta$ vrcholu otce $t$, pak zavolá sama sebe na vrchol otec $t$. Popíšeme analýzu situace, na níž je založena korektnost procedury **Kontrola-DELETE**$(t)$.

V analýze je důležité, že když procedura **Kontrola-DELETE** přesune vrchol $x$ na místo vrcholu $y$, pak skutečná hodnota $\eta(x)$ je buď původní hodnota $\eta(y)$ nebo je přesně o 1 menší. Všimněte si, že to platí.

Dán vrchol $t$, jehož hodnota $\eta(t)$ se zmenšila (o 1). V podstromu určeném vrcholem $t$ jsou hodnoty $\omega$ aktualizovány, $v = \text{otec}(t)$ a $\omega(v)$ je původní. Předpokládejme $t = \text{levy}(v)$, $u = \text{pravy}(v)$ a $\eta(t) = a$ ($a$ je neznámé). Nastávají případy:

1. když $\omega(v) = 0$, tak jsme jen z „vyváženého“ vrcholu udělali nevyvážený, tedy $\eta(u) = a + 1$ a $\eta(v) = a + 2$.

   Stačí položit $\omega(v) = 1$ a skončit.

2. Když $\omega(v) = -1$, tak jsme strom, „vychýlený“ doleva „narovnali“ - tj. změníme číslo a propagujeme výš.

   Tedy $\eta(u) = a$ a $\eta(v) = a + 2$. Nyní položíme $\omega(v) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol $v$.

3. Když $\omega(v) = 1$, máme problém - ubíráme na už kratším konci. Opět vezmeme $u_1 = \text{levy}(u)$, $u_2 = \text{pravy}(u)$ a opět se podle $\omega(u)$ rozhodneme, co budeme dělat.

   (a) $\omega(u) = 1 \implies \eta(u_1) = a$, $\eta(u_2) = a + 1$. Provedeme **Rotace**$(v, u)$.
   Vrchol $u_1$ je druhým synem $v$ a platí $\eta(t) = \eta(u_1) = a$, $\eta(v) = \eta(u_2) = a + 1$ a $\eta(u) = a + 2$. Tedy položme $\omega(v) = \omega(u) = 0$ a zavolejme **Kontrola-DELETE** na vrchol $u$.

   (b) $\omega(u) = 0 \implies \eta(u_1) = \eta(u_2) = a + 1$. Provedeme **Rotace**$(v, u)$.
   Vrchol $u_1$ je druhým synem $v$ a platí $\eta(t) = a$, $\eta(u_1) = a+1 = \eta(u_2)$, $\eta(v) = a+2$, $\eta(u) = a+3$. Položme $\omega(v) = 1$, $\omega(u) = -1$ a končíme.

   (c) $\omega(u) = -1 \implies \eta(u_1) = a + 1$, $\eta(u_2) = a$. Provedeme **Dvojita-rotace**$(v, u, u_1)$. Opět řešíme více případů podle $\omega(u) = 1$
   $u_3 = \text{levy}(u_1)$, $u_4 = \text{pravy}(u_1)$

      i. $\omega(u_1) = -1 \implies \eta(u_3) = a$, $\eta(u_4) = a - 1$ a tedy $\eta(v) = \eta(u) = a + 1$ a $\eta(u_1) = a + 2$. Proto položíme $\omega(v) = \omega(u_1) = 0$, $\omega(u) = 1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol $u_1$.

      ii. $\omega(u_1) = 0 \implies \eta(u_3) = \eta(u_4) = a$ a tedy $\eta(v) = \eta(u) = a + 1$ a $\eta(u_1) = a + 2$. Proto položíme $\omega(v) = \omega(u_1) = \omega(u) = 0$ a zavoláme proceduru **Kontrola-DELETE** na vrchol $u_1$.

      iii. $\omega(u_1) = 1 \implies \eta(u_3) = a - 1$, $\eta(u_4) = a$ a tedy $\eta(v) = \eta(u) = a+1$ a $\eta(u_1) = a+2$. Proto položíme $\omega(u) = \omega(u_1) = 0$, $\omega(v) = -1$ a zavoláme proceduru **Kontrola-DELETE** na vrchol $u_1$.

```
Kontrola-DELETE(t)
v := otec(t)
if t = levy(v) then
  Leva-Kontrola-DELETE
else
  Leva-Kontrola-DELETE
endif
```

```
Leva-Kontrola-DELETE(t)
if ω(v) = 1 then
  u := pravy(v)
  if ω(u) ≥ 0 then
    Rotace(v, u)
    if ω(v) = 0 then
      ω(v) := 1, ω(u) := −1
    else
      ω(u) := ω(v) := 0, t := u, Kontrola-DELETE(t)
    endif
  else
    w := levy(u), Dvojita-rotace(v, u, w)
    if ω(w) = 1 then
      ω(u) := 0, ω(v) := −1
    else
      if ω(w) := 0 then
        ω(u) := 0, ω(v) := 0
      else
        ω(u) := 1, ω(v) := 0
      endif
    endif
    ω(w) := 0, t := w, Kontrola-Delete(t)
  endif
else
  if ω(v) = 0 then
    ω(v) := 1
  else
    ω(v) := 0, t := v, Kontrola-DELETE(t)
  endif
```

```
Prava-Kontrola-DELETE(t)
if ω(v) = −1 then
  u := levy(v)
  if ω(u) ≤ 0 then
    Rotace(v, u)
    if ω(u) = 0 then
      ω(v) := −1, ω(u) := 1
    else
      ω(u) := ω(v) := 0, t := u, Kontrola-DELETE(t)
    endif
  else
    w := pravy(u), Dvojita-rotace(v, u, w)
    if ω(w) = 1 then
      ω(u) := −1, ω(v) := 0
    else
      if ω(w) := 0 then
        ω(u) := 0, ω(v) := 0
      else
        ω(u) := 0, ω(v) := 1
      endif
    endif
    ω(w) := 0, t := w, Kontrola-Delete(t)
  endif
else
  if ω(v) = 0 then
    ω(v) := −1
  else
    ω(v) := 0, t := v, Kontrola-DELETE(t)
  endif
endif
```

V operaci **DELETE** se může stát, že procedury **Rotace** nebo **Dvojita-rotace** jsou volány až $\log(|S|)$-krát. To je výrazný rozdíl proti operaci **INSERT**. Proto operace **DELETE** je pomalejší než operace **INSERT**, i když asymptoticky jsou stejně rychlé. Korektnost se ověří přímo.

**Věta.** *Datová struktura AVL-strom umožňuje implementaci operací* **MEMBER**, **INSERT** *a* **DELETE**, *které vyžadují čas* $O(\log(|S|))$ *(kde* $S$ *je reprezentovaná množina). Operace* **INSERT** *zavolá nejvýše jednu proceduru* **Rotace** *nebo* **Dvojita-rotace**.

## 4.5 Červeno-černé stromy

### 4.5.1 Definice

Binární vyhledávací strom $T$ reprezentující množinu $S$, jehož vrcholy jsou obarveny červeně nebo černě (každý vrchol má právě jednu barvu) tak, že jsou splněny podmínky:

- listy jsou obarveny černě,
- když $v$ je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
- všechny cesty z kořene do listů mají stejný počet černých vrcholů

se nazývá *červeno-černý strom.*

Pozn studenta – já sám to vždycky chápu tak, že červené vrcholy jsou „špatné“ a ukazují nám odchylku od perfektní vyváženosti, takže jich tam nesmí být moc.

### 4.5.2 Vyváženost

Nejprve ukážeme, že červeno-černé stromy jsou vyvážené stromy, tj.

$$\text{hloubka}(T) = O(\log(|S|).$$

**Věta.** *Když červeno-černý strom $T$ reprezentuje množinu $S$, pak*

$$\text{hloubka}(T) \leq 2\log(2|S| + 2) = 1 + \log(|S| + 1).$$

*Důkaz.* Předpokládejme, že $T$ je červeno-černý strom, který má na cestě z kořene do listu právě $k$ černých vrcholů. Pak pro počet vrcholů $\#T$ stromu $T$ platí

$$2^k - 1 \leq \#T \leq 2^{2k} - 1.$$

Nejmenší takový strom má všechny vrcholy černě obarvené a je to úplný pravidelný binární strom o výšce $k - 1$, což dává dolní odhad. Největší takový strom má všechny vrcholy v sudých hladinách obarveny červeně a v lichých hladinách černě, je to úplný pravidelný binární strom o výšce $2k - 1$ a tím je dán horní odhad. Tedy $k \leq \log(1 + \#T) \leq 2k$. Protože velikost $S$ je počet vnitřních vrcholů, dostáváme, že $\#T = 2|S| + 1$. Z vlastností červeno-černých stromů plyne, že

$$k \leq \text{hloubka}(T) \leq 2k.$$

□

---

### 4.5.3 Popis algoritmů (kromě vyvažování)

Pro červeno-černé stromy navrhneme algoritmy realizující operace z uspořádaného slovníkového problému.

Operace **MEMBER** pro červeno-černé stromy je stejná jako pro nevyvážené binární vyhledávací stromy.

Operace **INSERT** a **DELETE** mají dvě části: nejprve se provede operace **INSERT** nebo **DELETE** pro nevyvážené binární vyhledávací stromy a pak následují vyvažovací operace, které zajistí, že výsledný strom splňuje podmínky pro červeno-černé stromy (stejné schéma jako pro AVL-stromy).

Schéma operací **JOIN** a **SPLIT** bude vycházet z jejich realizací v $(a, b)$-stromech.

V operaci **JOIN** prohledáváním nalezneme místo, kde se stromy dají spojit (a aplikujeme operaci **JOIN** pro nevyvážené binární vyhledávací stromy), a pak použijeme vyvažovací operace.

Algoritmus operace **SPLIT** rozdělí červeno-černý strom do několika menších podle cesty vyhledávající $x$ (podobně jako v $(a, b)$-stromech) a na tyto stromy pak aplikuje operaci **JOIN** a zkonstruuje hledané červeno-černé stromy. Algoritmy pro operace **MIN** a **MAX** jsou stejné jako pro nevyvážené binární vyhledávací stromy.

### 4.5.4 Vyvažovací operace

Nejprve popíšeme vyvažovací operace.

**Definice.** *Dvojice $(T, v)$ se nazývá 2-*parciální červeno-černý strom*, když $T$ je binární vyhledávací strom, každý vrchol je obarven červeně nebo černě, $v$ je vnitřní vrchol stromu $T$ obarvený červeně a platí:*

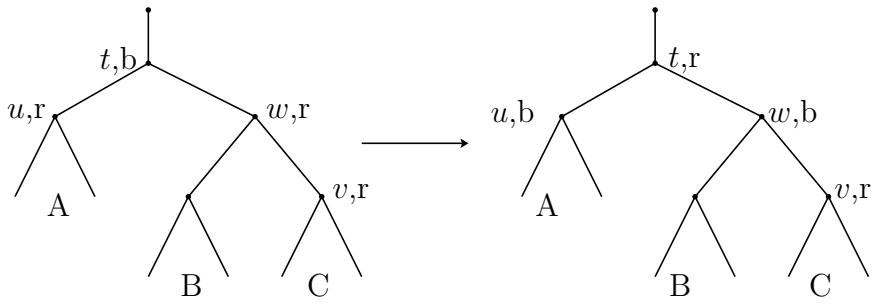
- *listy jsou obarveny černě,*
- *když $t$ je vrchol obarvený červeně, pak je buď kořen stromu nebo $t = v$ nebo jeho otec je obarven černě,*
- *všechny cesty z kořene do listů mají stejný počet černých vrcholů.*

Tj. jde opět o „o 1 špatný“ červeno-černý strom.

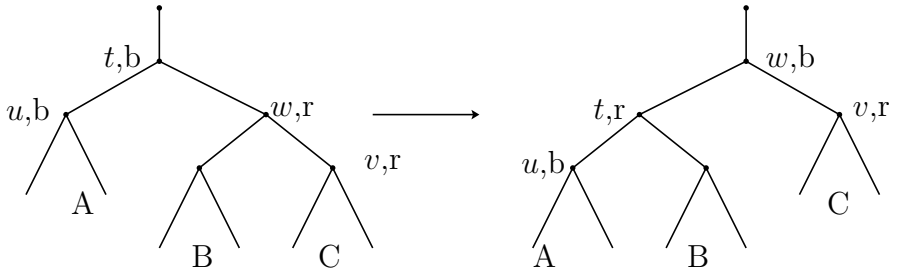
Vyvažování 2-parciálního červeno-černého stromu $(T', v)$ provádí procedura **Vyvaz-INSERT**$(v)$. Po jejím provedení buď dostaneme červeno-černý strom nebo je procedura **Vyvaz-INSERT** zavolána na vrchol $v'$ takový, že $(T', v')$ je 2-parciální červeno-černý strom a $v'$ je děd $v$ (tj. je o dvě hladiny blíž ke kořeni než vrchol $v$).

**Definice.** *Obarvení je realizováno rozšířením struktury vrcholu $v$ o boolskou proměnnou $b(v)$, kde $b(v) = 0$ znamená, že $v$ je obarven červeně, a $b(v) = 1$ znamená, že $v$ je obarven černě.*

Rozebereme případy. Na obrázku $b$ značí černou barvu a $r$ značí červenou barvu. Otec vrcholu $w$ je označen $t$.

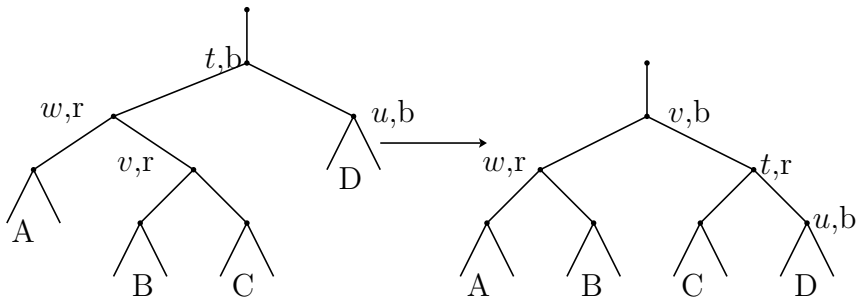



Obrázek 3




Obrázek 4

1. Pokud je otec černý, nic neřeším.




Obrázek 5

2. Pokud je otec kořen, nemusíme nic řešit a změníme otcovi barvu.

3. Tedy otec je určitě červený a není kořen.

4. Pokud je strýc také červený, přebarvíme otce i strýce („celou generaci“), ale kvůli počtům musíme přebarvit i děda a propagovat výše. (Viz obrázek 3)

5. Pokud je strýc červený, přebarvit generaci nemůžeme – místo toho uděláme rotaci. Pokud $t - w - v$ je „rovná“, uděláme jednoduchou rotaci (obrázek 4), pokud je „lomená“, uděláme dvojitou rotaci (obrázek 5); potom přebarvíme tak, aby v žádné cestě nepřibyl černý uzel.

Popíšeme formálně proceduru **Vyvaz-INSERT**$(v)$ (předpokládáme, že $v$ je obarven červeně). Pro zjednodušení $s(v) = levy$, když $v = \text{levy}(\text{otec}(v))$, a $s(v) = pravy$ pro $v = \text{pravy}(\text{otec}(v))$.

```
Vyvaz-INSERT(v).
if v není kořen T' a b(otec(v)) = 0 then
  if otec(v) je kořen then
    b(otec(v)) := 1
  else
    w := otec(v), u := bratr(w)
    if b(u) = 0 then
      v := otec(w), b(w) := 1, b(u) := 1
      b(v) := 0, Vyvaz-INSERT(v) (Viz Obrázek 3)
    else
      t := otec(w)
      if s(w) = s(v) then
        Rotace(t, w), b(t) := 0, b(w) := 1 (Viz Obrázek 4)
      else
        Dvojita-rotace(t, w, v), b(t) := 0, b(v) := 1 (Viz Obrázek 5)
      endif
    endif
  endif
endif
```

2-parciální červeno-černé stromy vznikají při operacích **INSERT** a **JOIN.** Při operaci **DELETE** se poruší struktura červeno-černých stromů jiným způsobem a vznikne 3-parciální červeno-černý strom.

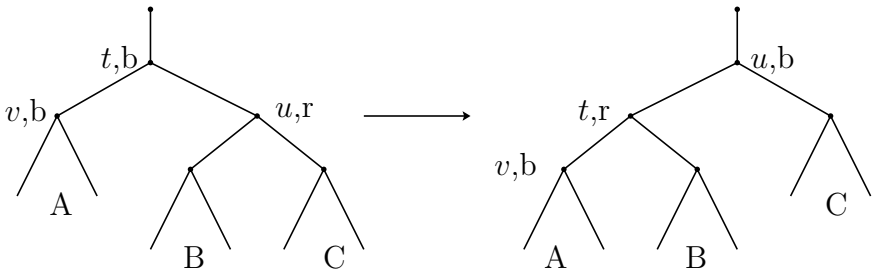
Řekneme, že dvojice $(T, v)$ je *3-parciální červeno-černý strom*, když $T$ je binární vyhledávací strom, každému vrcholu je přiřazena právě jedna z dvojice barev červená – černá, $v$ je vrchol ve stromu $T$ a platí následující podmínky:

- listy a vrchol $v$ jsou obarveny černě,

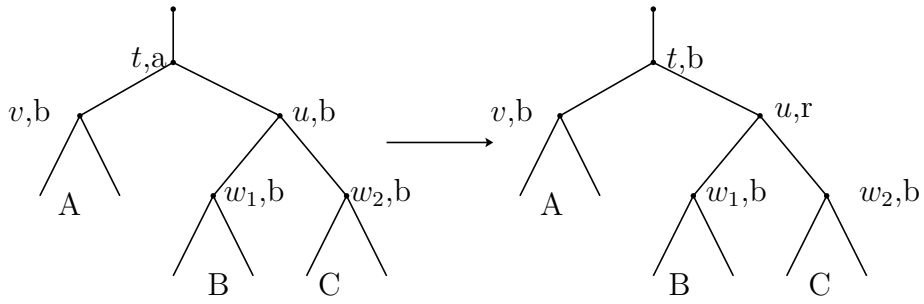
- když $t$ je vrchol obarvený červeně, pak je buď kořen stromu nebo jeho otec je obarven černě,
- existuje číslo $k$ takové, že všechny cesty z kořene do listů, které neobsahují vrchol $v$, obsahují právě $k$ černých vrcholů, a všechny cesty z kořene do listů procházející vrcholem $v$ obsahují $k-1$ černých vrcholů.

Rozdíl je v tom, že u 2-parciálních jsme „slevili“ na následnosti červených, naopak tady slevujeme na stejném počtu černých vrcholů.

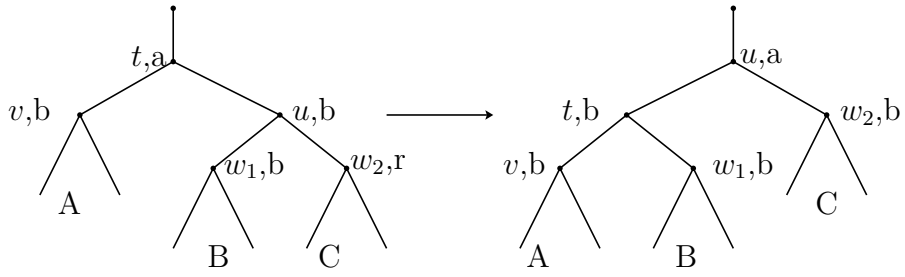
Rozebereme případy. V následujících obrázcích jsou vrcholy, které nemají specifikovanou barvu (mohou být jak červené tak černé). Tyto barvy budeme označovat $a$, $a'$. Důvod je, že se tato barva může přenést do cílového stromu, ale i na jiný vrchol. V tomto smyslu jsou tyto barvy určeny vstupním stromem a specifikují tyto barvy v cílovém stromě. V Obr. 7 se barva $a$ v cílovém stromě neobjevuje.




Obrázek 6

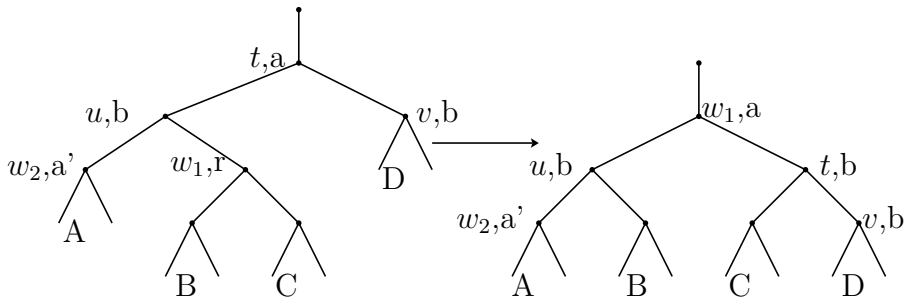



Obrázek 7




Obrázek 8

1. pokud máme červeného bratra, uděláme rotaci, abychom ho měli černého (algoritmus ale nekončí). Obr. 6




Obrázek 9

2. tedy bratr je černý. Pokud oba synovce černé, můžu bratra přebarvit na červeno (Obr. 7) a:

   (a) Pokud je otec červený, přebarvíme ho na černo a problémů jsme se zbavili: v cestách v podstromu $v$ je o jednoho černého víc, kdežto v cestách jeho bratra nic nepřibylo.

   (b) Pokud je otec černý, problémů jsme se nezbavili, ale naopak jsme je přidali i do cest v bratru, tedy můžeme je delegovat o stupeň výš.

3. Pokud nejsou oba synovce černé, ale synovec blíž „ke mě“ černý je, udělám takovou rotaci, aby se stal mým bratrem, a přebarvíme tak, aby se jednak problémy vyřešily, ale aby „vršek“ všeho zůstal stejný. (Obr. 8)

4. Pokud nejsou oba synovce černé a synovec blíž „ke mě“ je červený, udělám takovou rotaci, aby byl „nahoře“, a takové přebarvení, aby se problém vyřešil, ale „vršek“ byl pořád stejný. (Obr. 9)

Formálně opíšeme proceduru **Vyvaz-DELETE**$(v)$, která se použije na 3-parciální červeno-černý strom $(T, v)$, když $v$ není jeho kořen. Výsledkem procedury bude buď červeno-černý strom nebo zavolání procedury **Vyvaz-DELETE**$(v')$, kde $v'$ je otcem vrcholu $v$. Z faktu, že když $(T, v)$ je 3-parciální červeno-černý strom a $v$ je jeho kořen, pak $T$ je červeno-černý strom, plyne, že aplikací **Vyvaz-DELETE**$(v)$ na 3-parciální červeno-černý strom $(T, v)$ dostaneme červeno-černý strom.

**Vyvaz-DELETE**$(v)$
$u :=$ bratr$(v)$, $t :=$ otec$(v)$
**if** $b(u) = 0$ **then**
  **Rotace**$(t, u)$, $b(u) := 1$, $b(t) := 0$, $u :=$ bratr$(v)$
**endif**
(Viz Obr. 6, Komentář: nyní $b(u) = 1$)
$w_1$ je syn $u$ takový, že $s(v) = s(w_1)$, $w_2 :=$ bratr$(w_1)$
**if** $b(w_1) = b(w_2) = 1$ **then**
  $b(u) := 0$
  **if** $b(t) := 0$ **then**
    $b(t) := 1$
  **else**
    **if** $t$ není kořen stromu **then**
      $v := t$, **Vyvaz-DELETE**$(v)$
    **endif**
  **endif** (Viz Obr. 7)
**else**
  **if** $b(w_1) = 1$ **then**
(Komentář: $b(w_2) = 0$)
    **Rotace**$(t, u)$, $b(w_2) := 1$, $b(u) := b(t)$, $b(t) := 1$ (Viz Obr. 8)
  **else**
    **Dvojita-rotace**$(t, u, w_1)$, $b(w_1) := b(t)$, $b(t) := 1$ (Viz Obr. 9)
  **endif**
**endif**

### 4.5.5 Popis nevyvažovacích operací

Nyní popíšeme algoritmy realizující operace **INSERT**, **DELETE**, **JOIN3** a **SPLIT** pro červeno-černé stromy.

Předpokládejme, že $T$ je červeno-černý strom reprezentující množinu $S$ a provádíme operaci **INSERT**$(x)$ pro $x \notin S$. Když operace **INSERT**$(x)$ pro nevyvážené binární vyhledávací stromy vytvoří strom $T'$, kde vrchol $v$ reprezentuje $x$, pak $v$ obarvíme červeně a syny $v$ (jsou to listy) obarvíme černě. Dostáváme, že $(T', v)$ je 2-parciální červeno-černý strom, a pak aplikujeme proceduru **Vyvaz-INSERT**.

Operace **INSERT** v červeno-černých stromech volá nejvýše $2 + \log(|S|)$-krát proceduru **Vyvaz-INSERT** a provede nejvýše jednu rotaci nebo dvojitou rotaci.

Operace **DELETE** je řešena stejným způsobem jako operace **INSERT**, ale při operaci **DELETE** je porušena třetí podmínka v definici červeno-černých stromů a vyvažování je technicky náročnější.

Předpokádejme, že $T$ je červeno-černý strom. Když chceme provést operaci **DELETE**, pak nejprve provedeme algoritmus **DELETE** pro nevyvážené binární vyhledávací stromy. Při provádění jsme odstranili vrchol $u$ a jeho syna $w$, který je list. Na místo vrcholu $u$ se dostal jeho druhý syn $v$, který obarvíme černě. Pak jsou splněny první dvě podmínky v definici červeno-černých stromů a pokud vrchol $u$ nebo vrchol $v$ byl obarven červeně, pak je splněna i třetí podmínka. Pokud vrchol $u$ i vrchol $v$ byly obarveny černě, pak každá cesta z kořene do listu obsahující vrchol $v$ má o jeden černý vrchol méně než cesta z kořene do listu neobsahující vrchol $v$ (chybí černý vrchol $u$), a tedy $(T, v)$ je 3-parciální červeno-černý strom. Nyní aplikujeme proceduru **Vyvaz-DELETE**. Analýza poskytuje rychlý test na to, zda vznikne červeno-černý strom nebo 3-parciální červeno-černý strom (pak $v$ je list).

Popišme **JOIN3**$(T_1, x, T_2)$ . Mějme červeno-černé stromy $T_1$ a $T_2$ reprezentující množiny $S_1$ a $S_2$ a mějme prvek $x \in U$ takový, že $\max S_1 < x < \min S_2$. Nejprve zajistíme, že kořeny $T_1$ i $T_2$ jsou obarveny černě.

Předpokládejme, že $k_i$ je počet černých vrcholů na cestě z kořene do listů ve stromě $T_i$ pro $i = 1, 2$. Když $k_1 = k_2$, pak stačí provést **JOIN3**$(T_1, x, T_2)$ pro nevyvážené binární vyhledávací stromy (kořen obarvíme červeně). Problém je, když $k_1 \neq k_2$. Například předpokládejme, že $k_1 > k_2$. Pak začneme v kořeni stromu $T_1$ a jdeme po pravých synech dolů tak dlouho, až nalezneme černý vrchol $v$ takový, že všechny cesty z $v$ do listů v $T_1$ obsahují právě $k_2$ černých vrcholů. Pak provedeme **JOIN3** pro nevyvážené binární vyhledavací stromy na podstrom $T_1$ určený vrcholem $v$, na $x$ a na $T_2$. Kořen $w$ vzniklého stromu obarvíme červeně a tento strom vložíme do $T_1$ místo podstromu určeného vrcholem $v$. Pak $(T_1, w)$ je 2-parciální červeno-černý strom a aplikujeme proceduru **Vyvaz-INSERT**. Případ $k_2 > k_1$ se řeší symetricky.

Algoritmus pro operaci **SPLIT** je velmi podobný algoritmu pro $(a, b)$-stromy. Vyhledáváme vrchol reprezentující $x$. Když jsme ve vrcholu $t$ a pokračujme akcí $t :=$ levy$(t)$, pak dvojici key$(t)$ a podstrom $T$ určený pravým synem $t$ vložíme do zásobníku $Z_2$, když pokračujeme akcí $t :=$ pravy$(t)$, pak do zásobníku $Z_1$ vložíme dvojici podstrom $T$ určený levým synem $T$ a key$(t)$. Když key$(t) = x$, pak do $Z_1$ vložíme podstrom určený levým synem $t$ a do $Z_2$ podstrom určený pravým synem $t$. Když $t$ je list, pak do $Z_1$ i $Z_2$ vložíme jednoprvkové stromy. Ze zásobníku $Z_1$ pomocí operace **JOIN3** vytvoříme strom $T_1$ a ze zásobníku $Z_2$ pomocí operace **JOIN3** dostaneme strom $T_2$.

Nyní popíšeme algoritmy pro tyto operace.

**INSERT**$(x)$
**Vyhledej**$(x)$
**if** $t$ je list **then**
  $t$ se změní na vnitřní vrchol, key$(t) := x$
  pro vrchol $t$ vytvořme syny levy$(t)$ a pravy$(t)$
  $b(t) := 0$, $b($levy$(t)) := 1$, $b($pravy$(t)) := 1$, **Vyvaz-INSERT**$(t)$
**endif**

```
DELETE(x)
Vyhledej(x)
if t není list then
  vyv := false
  if levy(t) je list then
    v := pravy(t)
    if b(t) = 1 a b(v) = 1 then
      vyv := true
    endif
    odstraníme vrchol levy(t), otec(v) := otec(t)
    if t = levy(otec(t)) then
      levy(otec(t)) := v
    else
      pravy(otec(t)) := v
    endif
    b(v) := 1, odstraníme vrchol t
  else
    u := levy(t)
    while pravy(u) není list do u := pravy(u) enddo
    key(t) := key(u), v := levy(u)
    if b(u) = 1 a b(v) = 1 then
      vyv := true
    endif
    odstraníme vrchol pravy(u), otec(v) := otec(u)
    if u = levy(otec(u)) then
      levy(otec(u)) := v
    else
      pravy(otec(u)) := v
    endif
    b(v) := 1, odstraníme vrchol u
  endif
  if vyv then Vyvaz-DELETE(v) endif
endif
```

```
JOIN3(T1, x, T2)
if b(kořen T1) = 0 then b(kořen T1) := 1 endif
if b(kořen T2) = 0 then b(kořen T2) := 1 endif
k1 je počet černých vrcholů v T1 z kořene do listů
k2 je počet černých vrcholů v T2 z kořene do listů
if k1 ≥ k2 then
   t := kořen T1, i := k1 − k2
   while i > 0 do
      t := pravy(t)
      if b(t) = 1 then i := i − 1 endif
   enddo
   vytvoř vrchol u, b(u) := 0, key(u) := x
   if t není kořen T1 then
      otec(u) := otec(t), pravy(otec(t)) := u
   endif
   otec(t) := u, otec(kořen T2) := u
   pravy(u) := kořen T2, levy(u) := t, Vyvaz-INSERT(T1, u)
else
   t := kořen T2, i := k2 − k1
   while i > 0 do
      t := levy(t)
      if b(t) = 1 then i := i − 1 endif
   enddo
   vytvoř vrchol u, b(u) := 0, key(u) := x
   otec(u) := otec(t), levy(otec(t)) := u, otec(t) := u
   otec(kořen T1) := u, levy(u) := kořen T1
   pravy(u) := t, Vyvaz-INSERT(T2, u)
endif
```

```
SPLIT(x)
Z_1 a Z_2 jsou prázdné zásobníky, t := kořen T
while key(t) ≠ x a t není list do
  if key(t) > x then
    vlož (key(t), pravy(t)) do Z_2, t := levy(t)
  else
    vlož (levy(t), key(t)) do Z_1, t := pravy(t)
  endif
enddo
if key(t) = x then
  Výstup: x ∈ S, T_1 je podstrom T určený levy(t)
  T_2 je podstrom T určený pravy(t)
else
  Výstup: x ∉ S, T_1 a T_2 jsou jednoprvkové stromy
endif
while Z_1 ≠ ∅ do
  (t, x) je na vrcholu Z_1, odstraň (t, x) ze Z_1
  T' je podstrom T určený t, T_1 :=JOIN3(T', x, T_1)
enddo
while Z_2 ≠ ∅ do
  (x, t) je na vrcholu Z_2, odstraň (x, t) ze Z_1
  T' je podstrom T určený t, T_2 :=JOIN3(T_2, x, T')
enddo
```

### 4.5.6 Korektnost a složitost

Korektnost algoritmů je vidět z obrázků. Všimněme si při operaci **DELETE**, že když $u$ je obarven červeně, pak po provedení **Rotace**$(t, u)$ bude $(T, v)$ opět 3-parciální červeno-černý strom a vrchol $t$ bude obarven červeně. Pak z Obr. 5 je vidět, že dostaneme červeno-černý strom. Tedy můžeme shrnout:

**Věta.** *Algoritmy operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3** *a* **SPLIT** *pro červeno-černé stromy vyžadují v nejhorším případě čas* $O(\log(|S|))$*, kde* $S$ *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo* **Rotace** *a* **Dvojita-rotace**.

Všimněte si, že operace **JOIN3** ve skutečnosti vyžaduje čas $O(|k_1 - k_2| + 1)$. Protože $Z_1$ a $Z_2$ obsahují nejvýše $\log(|S|)$ položek, tak se odhad časové složitosti operace **SPLIT** provede stejným způsobem jako v $(a, b)$-stromech. V ostatních případech je odhad časové složitosti vidět z toho, že hloubka$(T) = O(\log(|S|))$ a akce na každé hladině vyžadují jen $O(1)$ času.

Pokud chceme mít i algoritmus pro operaci **ord**$(k)$, pak musíme rozšířit strukturu o funkci $p$. Pak lze použít přímo algoritmus pro **ord**$(k)$ v nevyvážených binárních vyhledávacích stromech. Připomeňme si, že procedury **Rotace** a **Dvojita-rotace** mohou aktualizovat funkci $p$ v čase $O(1)$. Proto dostáváme

**Věta.** *Algoritmy operací* **MEMBER**, **INSERT**, **DELETE**, **MIN**, **MAX**, **JOIN3**, **SPLIT** *a* **ord**$(k)$ *pro rozšířenou strukturu červeno-černých stromů vyžaduje v nejhorším případě čas* $O(\log(|S|))$*, kde* $S$ *je reprezentovaná množina. Operace* **INSERT** *a* **JOIN3** *zavolají nejvýše jednou buď* **Rotace** *nebo* **Dvojita-rotace** *a operace* **DELETE** *zavolá nejvýše dvakrát* **Rotace** *nebo jednou* **Rotace** *a* **Dvojita-rotace**.

Vzniká otázka, proč se tolik pozornosti věnuje procedurám **Rotace** a **Dvojita-rotace**. Sice vyžadují čas $O(1)$, ale jsou to nejsložitější akce vyžadující nejvíce času. V mnoha aplikacích (používají se hlavně ve

výpočetní geometrii), tvar stromu spolu s parametry nesou ještě další zakódované informace. Při změně tvaru stromu je třeba je přepočítat. **Rotace** a **Dvojita-rotace** mění tvar stromu, kdežto posun směrem ke kořeni pouze mění obarvení. V tomto případě pak **Rotace** nebo **Dvojita-rotace** vyžaduje čas $O(|S|)$ (obvykle je třeba prohlédnout celý strom) a nikoliv $O(1)$.

## 4.6 Váhově vyvážené stromy

V osmdesátých letech se ve výpočetní geometrii hodně používaly $BB(\alpha)$-stromy, proto se o nich alespoň orientačně zmíníme. Mějme reálné číslo $\alpha$ takové, že $\frac{1}{4} < \alpha \leq \frac{\sqrt{2}}{2}$. Pro strom $T$ označme $p(T)$ počet listů ve stromu $T$. Binární vyhledávací strom $T$ reprezentující množinu $S$ se nazývá $BB(\alpha)$-*strom*, když pro každý vnitřní vrchol $v$ platí:

$$\alpha \leq \frac{p(T_l)}{p(T_v)} = 1 - \frac{p(T_r)}{p(T_v)} \leq 1 - \alpha$$

kde $T_v$ je podstrom $T$ určený vrcholem $v$, $T_l$ je podstrom $T$ určený levým synem vrcholu $v$, $T_r$ je podstrom $T$ určený pravým synem vrcholu $v$. Platí

**Tvrzení.** *Když $T$ je $BB(\alpha)$-strom reprezentující $n$-prvkovou množinu, pak*

$$\text{hloubka}(T) \leq 1 + \frac{\log(n+1) - 1}{\log \frac{1}{1-\alpha}}.$$

Důsledek je, že $BB(\alpha)$-stromy patří do skupiny vyvážených binárních vyhledávacích stromů. Vyvažování se provadí opět pomocí **Rotace** a **Dvojita-rotace** a popisuje ho následující technické tvrzení.

**Tvrzení.** *Pro každé $\alpha$ existuje konstanta $d$ taková, že $\alpha < d < 1 - \alpha$ a pro každý binární vyhledávací strom $T$ s kořenem $t$ splňující podmínky*

1. *podstromy $T_l$ a $T_r$ stromu $T$ určené levým a pravým synem $t$ jsou $BB(\alpha)$-stromy;*
2. *$\frac{p(T_l)}{p(T)} < \alpha$, ale $\alpha \leq \frac{p(T_l)}{p(T)-1} \leq 1 - \alpha$ nebo $\alpha \leq \frac{p(T_l)+1}{p(T)+1} \leq 1 - \alpha$*

*platí:*
*když $\rho \leq d$ a provedeme* **Rotace**$(t, \text{pravy}(t))$*, nebo když $\rho > d$ a provedeme proceduru* **Dvojita-rotace**$(t, \text{pravy}(t), \text{l}$
*pak dostaneme $BB(\alpha)$-strom (zde $\rho = \frac{p(T')}{p(T_r)}$ a $T'$ je určen levým synem pravého syna kořene $t$).*

Toto tvrzení a jeho symetrické verze jednoznačně ukazují, jak vyvažovat $BB(\alpha)$-stromy při aktualizačních operacích (podstrom $BB(\alpha)$-stromu je $BB(\alpha)$-strom). Pak dostáváme:

**Věta.** *Implementace operací* **MEMBER**, **INSERT** *a* **DELETE** *v $BB(\alpha)$-stromech vyžaduje v nejhorším případě čas $O(\log(|S|))$, kde $S$ je reprezentovaná množina.*

Obliba $BB(\alpha)$-stromů byla zapříčiněna platností následující věty, která je analogií věty o vyvažovacích operacích pro $(a, b)$-stromy.

**Věta.** *Když $\alpha$ je reálné číslo takové, že $\frac{1}{4} < \alpha < 1 - \frac{\sqrt{2}}{2}$, pak existuje konstanta $c > 0$ závislá jen na $\alpha$ taková, že každá posloupnost operací* **INSERT** *a* **DELETE** *o délce $m$ aplikovaná na prázdný $BB(\alpha)$-strom volá nejvýše $cm$ procedur* **Rotace** *a* **Dvojita-rotace**.

## 4.7 Historický přehled:

$(a, b)$-stromy zavedli Bayer a McGreght (1972),
věty o počtu vyvažovacích operací pro $(a, b)$-stromy dokázali Huddleston a Mehlhorn (1982).
$A$-sort analyzovali Guibas, McGreight, Plass a Roberts (1977).

Analýza interpolačního vyhledávání pochází od Perla, Itai a Avniho (1978),
kvadratické vyhledávání analyzovali Perl a Reingold (1977).

Adelson-Velskij a Landis (1962) definovali AVL-stromy,
červeno-černé stromy definovali Guibas a Sedgewick (1978),
verze algoritmu **DELETE** pochází od Tarjana (1983). $BB(\alpha)$-stromy zavedli Nievergelt a Reingold (1973),
věty o jejich vyvažování dokázali Blum a Mehlhorn (1980).
Priorita AVL-stromů se odráží v jejim hojném používání, i když červeno-černé stromy jsou efektivnější.

# 5 Haldy

## 5.1 Úvodní definice

### 5.1.1 Motivace

V praxi se často setkáváme s následujícím problémem, který vzniká na uspořádaném univerzu, jehož uspořádání se však v průběhu času mění. Úloha se liší od slovníkového problému v tom, že se nevyžaduje efektivní operace **MEMBER**. Dokonce se předpokládá, že operace dostane spolu s argumentem informaci o uložení zpracovávaného prvku. Hlavním požadavkem je rychlost provedední ostatních operací a malé paměťové nároky. Přitom v praxi obvykle nestačí znát jen asymptotickou složitost, důležitou roli hraje skutečná rychlost, kterou však neumíme obecně spočítat, protože je závislá na použitém systému a hardwaru. Přesto je při použití následujících struktur dobré mít realistickou představu o skutečných rychlostech operací a podle toho si vybrat vhodnou strukturu.

### 5.1.2 Zadání

Zadání problému: Nechť $U$ je univerzum. Je dána množina $S \subseteq U$ a funkce $f : S \to \mathbb{R}$, kde $\mathbb{R}$ jsou reálná čísla (tato funkce realizuje uspořádání na univerzu $U$ – pro $u, v \in U$ platí $u \leq v$, právě když $f(u) \leq f(v)$; změna uspořádání se pak realizuje změnou funkce $f$). 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 $s \in S$ z množiny $S$,
**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 $s \notin S$, a 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 najít prvek $s$ v reprezentaci $S$ a $f$.

Haldy jsou typ struktury, která se používá pro řešení tohoto problému.

### 5.1.3 Definice haldy

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

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

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

**MAKEHEAP**$(S, f)$ – operace vytvoří haldu reprezentující množinu $S$ a funkci $f$,
**MERGE**$(H_1, H_2)$ – předpokládá, 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 $f_1 \cup f_2$, přičemž neověřuje disjunktnost $S_1$ a $S_2$.

## 5.2 Regulární haldy

První použité haldy byly binární neboli 2-regulární haldy. Tyto haldy jsou velmi oblíbené pro svou jednoduchost a názornost a pro velmi efektivní implementaci.

### 5.2.1 $d$-regulární strom

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

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 většími čísly jsou listy.

Toto očíslování se nazývá *přirozené očíslování* $d$-regulárního stromu.

### 5.2.2 Výška

**Tvrzení.** *Každý $d$-regulární strom má nejvýše jeden vrchol, který není list a má méně než $d$ synů.*

*Důkaz.* Plyne přímo z požadavku 2) na $d$-regulární strom. □

---

**Tvrzení.** *Když $d$-regulární strom má $n$ vrcholů, pak jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$.*

*Důkaz.* Má-li $d$-regulární strom výšku $k$, pak má alespoň $\Sigma_{i=0}^{k-1} d^i + 1$ a nejvýše $\Sigma_{i=0}^{k} d^i$ vrcholů. Proto

$$\frac{d^k - 1}{d-1} < n \leq \frac{d^{k+1}}{d-1} \quad , \quad d^k - 1 < n(d-1) \leq d^{k+1} - 1$$

a zlogaritmováním dostaneme

$$k < \log_d(n(d-1)+1) \leq k+1.$$

Odtud již plyne. □

---

### 5.2.3 Reprezentace pomocí pole

**Tvrzení.** *Nechť $o$ je přirozené očíslování vrcholů $d$-regulárního stromu. Když pro vrchol $v$ je $o(v) = k$, pak vrchol $w$ je syn vrcholu $v$, právě když $o(w) \in \{(k-1)d+2, (k-1)d+3, \ldots, kd+1\}$, a vrchol $u$ je otcem vrcholu $v$, právě když $o(u) = 1 + \lfloor \frac{k-2}{d} \rfloor$.*

*Důkaz.* Dokážeme indukcí podle očíslování.

Synové kořene mají čísla $2, 3, \ldots, d+1$, protože kořen má číslo 1. Když tvrzení platí pro vrchol s číslem $k$, pak synové vrcholu s číslem $k+1$ mají čísla $kd+2, kd+3, \ldots, kd+d+1$, což odpovídá číslům $(k+1-1)d+2, (k+1-1)d+3, \ldots, (k+1)d+1$, a tedy tvrzení platí. Poslední část pak plyne z toho, že když $i \in \{(k-1)d+2, (k-1)d+3, \ldots, kd+1\}$, pak $1 + \lfloor \frac{i-2}{d} \rfloor = k$. □

---

Všimněme si, že speciálně pro $d = 2$ mají synové vrcholu s číslem $k$ čísla $2k$ a $2k+1$ a otec vrcholu s číslem $k$ má číslo $\lfloor \frac{k}{2} \rfloor$. Tedy pro 2-regulární stromy je předpis pro nalezení synů a otce zvláště jednoduchý.

Řekneme, že množina $S$ s funkcí $f$ je reprezentována $d$-regulární haldou $H$, kde $H$ je $d$-regulární strom $(T, r)$, když přiřazení prvků množiny $S$ vrcholům stromu $T$ je bijekce splňující podmínku (usp). Toto přiřazení je realizováno funkcí key, která vrcholu přiřazuje jím reprezentovaný prvek.

Definice $d$-regulárního stromu umožňuje velmi efektivní implementace $d$-regulárních hald. Mějme množinu $S$ reprezentovanou $d$-regulární haldou $H$ s přirozeným očíslováním $o$ $d$-regulárního stromu $(T, r)$. Pak haldu $H$ můžeme reprezentovat polem $H[1..|S|]$, kde pro vrchol stromu $v$, pro který $o(v) = i$, je $H(i) = (\text{key}(v), f(\text{key}(v))$. Algoritmy budeme popisovat pro stromy, protože je to názornější. Přeformulovat je pro pole je snadné (viz očíslování synů a otce vrcholu $v$). Pro jednoduchost budeme pro vrchol $v$ psát $f(v)$ místo $f(\text{key}(v))$, neboli $f(v)$ bude označovat $f(s)$, kde $s$ 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.

### 5.2.4 Algoritmy

Pro $d$-regulární haldy není známa efektivní implementace operace **MERGE**. Efektivní implementace ostatních operací jsou založeny na 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). Operace **DOWN**$(v)$ je symetrická.

```
UP(v):
while v není kořen a f(v) < f(otec(v)) do
  vyměň key(v) a key(otec(v))
  v := 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 key($v$) a key($w$), $v := w$
    $w$ :=syn vrcholu $v$ reprezentující prvek s nejmenší hodnotou $f(w)$
  **enddo**
**endif**

**INSERT**($s$):
$v$ :=nový poslední list, key($v$) := $s$, **UP**($v$)

**MIN**:
**Výstup** key(kořen($T$))

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

**DELETE**($s$):
$v$ :=vrchol reprezentující $s$
$w$ :=poslední list
$t$ := key($w$), key($v$) := $t$, 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
zvol 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**

### 5.2.5 Korektnost

Ověříme korektnost algoritmů. Je zřejmé, že pomocné operace jsou korektní – skončí, když podmínku (usp) splňuje prvek $s$, který byl původně reprezentován vrcholem $v$. Korektnost operace **MIN** plyne přímo z podmínky (usp), protože kořen reprezentuje nejmenší prvek množiny $S$. 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 v tomto případě operace **DOWN** zajistí její splnění. Po provedení operací **DELETE**($s$), **DECREASE**($s, a$) a

**INCREASE**$(s, a)$ je podmínka (usp) splněna pro všechny vrcholy s výjimkou vrcholu $v$ a její splnění opět zajistí operace **UP** resp. **DOWN**. Pro operaci **MAKEHEAP** budeme uvažovat duální formulaci podmínky (usp):

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

Pokud každý vrchol splňuje podmínku (d-usp), pak splňuje i podmínku (usp). Zřejmě každý list splňuje podmínku (d-usp) a když operace **MAKEHEAP** provede proceduru **DOWN**$(v)$, pak je podmínka (d-usp) splněna pro všechny vrcholy s čísly alespoň tak velkými jako je číslo $v$. Operace **MAKEHEAP** končí provedením operace **DOWN** na kořen a odtud plyne její korektnost.

### 5.2.6 Složitost operací

Vypočteme časovou složitost operací: Jeden běh cyklu v operaci **UP** vyžaduje čas $O(1)$ a v operaci **DOWN** čas $O(d)$. Proto operace **UP** v nejhorším případě vyžaduje čas $O(\log_d |S|)$ a operace **DOWN** čas $O(d \log_d |S|)$. Operace **MIN** vyžaduje čas $O(1)$, **INSERT** a **DECREASE** vyžadují čas $O(\log_d |S|)$ a **DELETEMIN**, **DELETE** a **INCREASE** čas $O(d \log_d |S|)$.

Haldu můžeme vytvořit iterací operace **INSERT**, což vyžaduje čas $O(|S| \log_d(|S|))$. Ukážeme, že složitost operace **MAKEHEAP** je menší, ale pro malé haldy je výhodnější provádět opakovaně operaci **INSERT**.

**Věta. MAKEHEAP** *vyžaduje v nejhorším případě jen čas* $O(d^2|S|)$.

*Důkaz.* 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 vrchol v hloubce $i$ má výšku nejvýše $k - i$.

Tedy operace **MAKEHEAP** vyžaduje čas

$$O(\sum_{i=0}^{k-1} d^i(k-i)d) = O(\sum_{i=0}^{k-1} d^{i+1}(k-i)).$$

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

$$A = \frac{d^{k+1}}{d-1} + \frac{d^{k+1}-d^2}{(d-1)^2} - \frac{dk}{d-1}.$$

Protože $k = \lceil \log_d(|S|(d-1)+1) \rceil$, dostáváme, že $d^{k+1} \leq d^2((d-1)|S|+1)$, a proto $A \leq 2d^2|S|$.

Tedy **MAKEHEAP** vyžaduje v nejhorším případě jen čas $O(d^2|S|)$. □

### 5.2.7 Aplikace – heapsort

Třídění: prostou posloupnost čísel $x_1, x_2, \ldots, x_n$ lze setřídit následujícím algoritmem používajícím haldu ($f$ bude v tomto případě identická funkce).

$d$-**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**: $y_1, y_2, \ldots, y_n$

Teoreticky lze ukázat, že použití $d$-regulárních hald v algoritmu **HEAPSORT** pro $d = 3$ a $d = 4$ je výhodnější než $d = 2$. Experimenty ukázaly, že optimální algoritmus pro posloupnosti délek do 1 000 000 by měl používat $d = 6$ nebo $d = 7$ (v experimentech byl měřen skutečně spotřebovaný čas, nikoli počet porovnání a výměn prvků). Pro delší posloupnosti se optimální hodnota $d$ může zmenšit.

### 5.2.8 Aplikace – Dijkstra

Dalším příkladem je nalezení nejkratších cest v grafu z daného bodu. Řešme následující úlohu:

Vstup: orientovaný ohodnocený graf $(X, R, c)$, kde $c$ je funkce z $R$ do množiny kladných reálných čísel, a vrchol $z \in X$.
Úkol: nalézt pro každý bod $x \in X$ délku nejkratší cestu ze $z$ do $x$, kde délka cesty je součet $c$-ohodnocení hran na cestě.

**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 $u$ 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**

Korektnost algoritmu je založena na kombinatorickém lemmatu, které říká, že když odstraňujeme z $U$ prvek $x$ s nejmenší hodnotou $d(x)$, pak vzdálenost ze $z$ do $x$ je právě $d(x)$. Proto když $U = \emptyset$, pak $d(x)$ jsou délky nejkratších cest ze $z$ do $x$ pro všechna $x \in X$. Tedy práce s množinou $U$ vyžaduje nejvýše $|X|$ operací **INSERT**, **MIN** a **DELETEMIN** a $|R|$ operací **DECREASE** a vždy platí $|U| \leq |X|$.

Vypočteme časovou složitost **Dijkstrova algoritmu** za předpokladu, že $U$ reprezentujeme jako $d$-regulární haldu. Když $d = 2$, pak dostáváme, že algoritmus vyžaduje čas $O(|X| \log(|X|) + |R| \log(|X|))$. Když $d = \max\{2, \lfloor \frac{|R|}{|X|} \rfloor\}$, pak algoritmus vyžaduje čas $O(|R| \log_d |X|)$. V případě, že $(X, R)$ je hustý graf, tj. $|R| > |X|^{1+\varepsilon}$ pro $\varepsilon > 0$, pak $\log_d |X| = O(1)$ a algoritmus je lineární (tj. vyžaduje čas $O(|R|)$).

## 5.3 Leftist haldy

### 5.3.1 Úvod

Dalším typem hald, se kterými se seznámíme, jsou lefist haldy (neznáme vhodný český překlad, proto zůstáváme u anglického názvu). Je to velmi elegantní a jednoduchý typ hald. Všechny operace jsou stejně jako u regulárních hald založeny na dvou základních operacích, z nichž v tomto případě hlavní je **MERGE** a druhou je **DECREASE**. Použití **MERGE** při návrhu jiných operací je běžné i v dalších haldách. Operace **MERGE** využívá speciálních vlastností leftist hald a idea operace **DECREASE** je stejná jako ve Fibonacciho haldách. Nejprve formálně popíšeme strukturu leftist hald.

### 5.3.2 Defince

Mějme binární kořenový 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. pro list $l$ platí $\mathrm{npl}(l) = 0$.

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

$$\mathrm{npl}(\text{pravý syn } v) \leq \mathrm{npl}(\text{levý syn } v),$$

3. existuje jednoznačné přiřazení prvků $S$ vrcholům $T$, které splňuje podmínku (usp) (toto přiřazení je reprezentováno funkcí key, která vrcholu $v$ přiřadí prvek z množiny $S$ reprezentovaný vrcholem $v$)

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

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

### 5.3.3 Základní vlastnost

Uvedeme základní vlastnost leftist haldy, která umožňuje efektivní implementace operací. 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 má tedy alespoň $2^{k+1} - 1$ vrcholů. Proto platí

**Tvrzení.** *V leftist haldě je délka pravé cesty z každého vrcholu $v$ nejvýše rovna*

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

### 5.3.4 Algoritmy

Základní operací pro leftist haldy je **MERGE**. Tato operace je definována rekurzivně a hloubka rekurze je omezena právě délkami pravých cest.

**MERGE**$(T_1, T_2)$:
**if** $T_1 = \emptyset$ **then Výstup**$= T_2$ stop **endif**
**if** $T_2 = \emptyset$ **then Výstup**$= T_1$ stop **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**
  vyměň 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**
  vlož leftist haldu $T_s$ reprezentující $\{s\}$ 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**

### 5.3.5 Časová složitost

Vypočteme časovou složitost předchozích 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 $i = 1, 2$. Odtud dále plyne, že čas algoritmů **INSERT** a **DELETEMIN** je v nejhorším případě $O(\log(|S|))$. Operace **MIN** vyžaduje čas $O(1)$.

**Věta. MAKEHEAP** *má časovou složitost* $O(|S|)$

*Důkaz.* Budeme uvažovat, že na začátku algoritmu je na vrcholu fronty speciální znak, který se jen přenese na konec fronty.

Odhadneme čas, který spotřebují **while-**cykly mezi dvěma přeneseními speciálního znaku. Předpokládejme, že se speciální znak přenesl po $k$-té.

V tomto okamžiku mají všechny haldy ve frontě až na jednu velikost $2^{k-1}$.

Proto ve frontě $Q$ je $\lceil \frac{|S|}{2^{k-1}} \rceil$ hald a jelikož jedna operace **MERGE** vyžaduje $O(k)$ času, tak **while**-cykly vyžadují čas $O(k\frac{|S|}{2^{k-1}})$.

Můžeme tedy shrnout, že operace **MAKEHEAP** potřebuje čas

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

Řada $\sum_{k=1}^{\infty} \frac{k}{2^{k-1}}$ konverguje např. podle podílového d'Alambertova kritéria a lze jednoduše spočítat (např. stejnou metodou jako pro regulární haldy), že součet je 4, tedy čas je $O(|S|)$. □

---

### 5.3.6 Efektivní DECREASE a INCREASE

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 pro tyto operace efektivnější algoritmus založený na jiném principu. Tento princip je pak použit i pro Fibonacciho haldy.

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

```
Oprav(T, v):
t := otec(v), npl(t) := 0
if pravy(t) ≠ 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
   npl(t) := npl(pravy(t)) + 1
enddo
```

Po provedení operace **Oprav** mají všechny vrcholy správné číslo npl a podmínky kladené na leftist haldu jsou splněny. Tedy po provedení **Oprav** je $T$ opět leftist halda. Když $t$ je poslední vrchol, u kterého se zmenšilo npl, pak všechny 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 $O(\log(|S|))$.

Popíšeme ostatní algoritmy.

```
DECREASE(s, a):
v :=prvek reprezentující s
T1:=podstrom T určený vrcholem v, f(v) := f(v) − a
T2 :=Oprav(T, v), T :=MERGE(T1, T2)
```

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

Protože algoritmy **MERGE** a **Oprav** vyžadují čas $O(\log(|S|)$ a protože zbylé části algoritmů pro operace **DECREASE**, **INCREASE** a **DELETE** vyžadují $O(1)$ času, můžeme shrnout výsledky:

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

## 5.4 Amortizovaná složitost

Popíšeme bankovní paradigma pro počítání s amortizovanou složitostí.

### 5.4.1 Idea

Idea je taková, že si nějak ohodnotíme stavy (představa: účet v bance) a budeme dělat odhad upravené složitosti, ke které buď přičteme to, co si do účtu chceme nastřádat (tj. podle definice dále $h(D') > h(D), \text{spoření} = h(D') - h(D) > 0, am(o) = t(o) + \text{spoření}$), nebo naopak odečteme to, co jsme si nastřádali a chceme utratit (tj. $h(D') < h(D), \text{utrácení} = h(D) - h(D') > 0, am(o) = t(o) - \text{utrácení}$).

Odhad téhle upravené složitosti je pak i odhadem klasické složitosti.

### 5.4.2 Definice

Předpokládejme, že máme funkci $h$, která ohodnocuje konfigurace a kvantitativně vystihuje jejich vhodnost pro provedení operace $o$. Když na konfiguraci $D$ aplikujeme operaci $o$ a dostaneme konfiguraci $D'$, pak amortizovaná složitost $am(o)$ operace $o$ má vystihovat nejen časovou náročnost operace, ale i to, jak se změnila vhodnost konfigurace pro tuto operaci. Proto ji definujme jako $am(o) = t(o) + h(D') - h(D)$, kde $t(o)$ je čas potřebný pro provedení operace $o$. Předpokládejme, že chceme provést posloupnost operací $o_1, o_2, \ldots, o_n$ na konfiguraci $D_0$. Znázorníme si to takto:

$$D_0 \to^{o_1} D_1 \to^{o_2} D_2 \to^{o_3} \cdots \to^{o_n} D_n.$$

**Věta.** *Odhad amortizované složitosti je odhadem složitosti.*

*Důkaz.* Předpokládejme, že pro každé $i = 1, 2, \ldots, n$ máme odhad $c(o_i)$ amortizované složitosti operace $o_i$, tj. $am(o_i) \leq c(o_i)$ pro všechna $i = 1, 2, \ldots, n$. Pak

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

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 je $h(D) \geq 0$ pro všechny konfigurace $D$ nebo naopak $h(D) \leq 0$ pro všechny konfigurace $D$. Když $h(D) \geq 0$, pak

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

když $h(D) \leq 0$, pak

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

□

---

To znamená, že odhad amortizované složitosti dává také odhad na časovou složitost posloupnosti operací, který bývá lepší než odhad složitosti v nejhorším případě. Tato skutečnost vysvětluje řadu případů, kdy výsledky byly lepší než teoretický výpočet. Ukazuje se, že složitost posloupnosti operací v nejhorším případě je často podstatně menší než součet složitostí v nejhorším případě pro jednotlivé operace.

## 5.5 Binomiální haldy

### 5.5.1 Motivace

Další typ hald je motivován sčítáním přirozených čísel. Binomiální halda reprezentující $n-$prvkovou množinu se totiž chová podobně jako číslo $n$. Tento typ hald je také po zobecnění v jistém smyslu vzorem pro Fibonacciho haldy.

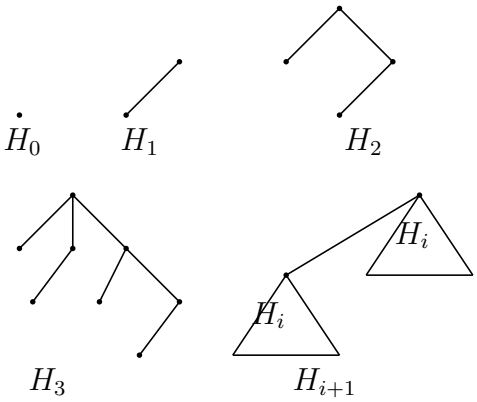
### 5.5.2 Definice binomiálního stromu

Pro $i = 0, 1, \ldots$ definujeme rekurentně binomiální stromy $H_i$. 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 stromu se stane dalším synem (nejlevějším nebo nejpravějším) kořene druhého stromu. Viz obrázek.

### 5.5.3 Vlastnosti binomiálního stromu

Nejprve uvedeme základní vlastnosti těchto stromů.

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




Obrázek 10: Binomiální stromy

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 $i$ (tj. výška $H_i$ je $i$),*
4. *podstromy určené syny kořene stromu $H_i$ jsou izomorfní se stromy $H_0, H_1, \ldots, H_{i-1}$.*

*Důkaz.* Tvrzení platí pro strom $H_0$ a jednoduchou indukcí se dokáže i pro další stromy. Skutečně, když $H_i$ má $2^i$ vrcholů, pak $H_{i+1}$ má $2(2^i) = 2^{i+1}$ vrcholů. Kořen stromu $H_{i+1}$ má o jednoho syna více než kořen stromu $H_i$ a nejdelší cesta do listu je o 1 delší. Protože podstrom syna, který přibyl kořeni stromu $H_{i+1}$, je izomorfní s $H_i$ a jinak se nic neměnilo, je důkaz kompletní. □ □

---

### 5.5.4 Definice binomiální haldy

**Definice.** Binomiální halda $\mathcal{H}$ *reprezentující množinu $S$ je soubor (seznam) stromů $\{T_1, T_2, \ldots, T_k\}$ takový, že*

1. *celkový počet vrcholů v těchto stromech je roven velikosti $S$ a existuje a je dáno jednoznačné přiřazení prvků z $S$ vrcholům stromů takové, že platí podmínka* (usp) *– toto přiřazení je realizováno funkcí* key, *která vrcholu stromu přiřazuje prvek jím reprezentovaný;*
2. *každý strom $T_i$ je izomorfní s nějakým stromem $H_j$;*
3. *$T_i$ 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 $S$. Tato halda obsahuje strom izomorfní s $H_i$, právě když v binárním zápise čísla $|S|$ je na $i$-tém místě zprava 1.

### 5.5.5 Algoritmy, korektnost

Operace pro binomiální haldy jsou stejně jako pro leftist haldy založeny 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**
    (existuje právě jeden neprázdný strom $V \in \{T, U_1, U_2\}$) **do**:
      vlož $V$ do $\mathcal{H}$, $T$ :=prázdný strom
    (existují právě dva neprázdné stromy $V_1, V_2 \in \{T, U_1, U_2\}$) **do**:
      $T$ :=**spoj**$(V_1, V_2)$
    (všechny stromy $T$, $U_1$ a $U_2$ jsou neprázdné) **do**:
      vlož $T$ do $\mathcal{H}$, $T$ :=**spoj**$(U_1, U_2)$
  **endcase**
  $i := i + 1$
**enddo**
**if** $T \neq$prázdný strom **then** vlož $T$ do $\mathcal{H}$ **endif**
**Výstup**:$\mathcal{H}$

**spoj**$(T_1, T_2)$:
**if** $f$(kořen $T_1$) $>$ $f$(kořen $T_2$) **then**
  vyměň stromy $T_1$ a $T_2$
**endif**
vytvoř nového syna $v$ kořene $T_1$
$v$ :=kořen $T_2$

Je vidět, že když oba stromy $T_1$ a $T_2$ jsou izomorfní s $H_i$, pak výsledný strom operace **spoj** je izomorfní s $H_{i+1}$.

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.

Implementace dalších algoritmů je podobná jako pro leftist haldy.

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

**MIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
**Výstup**: nejmenší z těchto prvků

**DELETEMIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
$\mathcal{H}_1 := \mathcal{H} \setminus \{T\}$
$\mathcal{H}_2 :=$ halda tvořená podstromy $T$ určenými 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 je dána korektnost operace **MIN**.

Z úvodního 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 v regulárních haldách.

Struktura binomiální haldy nepodporuje přímo operaci **DELETE** – ta se dá realizovat jedině jako posloupnost operací **DECREASE**$(s, \infty)$ a **DELETEMIN**. Operace **MAKEHEAP** se provádí opakováním operace **INSERT**.

### 5.5.6 Složitost

Výpočet časové složitosti operací pro binomiální haldy využívá několik známých faktů.

**Věta.** *Pro binomiální haldy algoritmy operací* **INSERT**, **MIN**, **DELETEMIN**, **DECREASE** *a* **MERGE** *vyžadují čas* $O(\log(|S|))$, *algoritmus operace* **INCREASE** *vyžaduje čas* $O(\log^2(|S|))$ *a algoritmus operace* **MAKEHEAP** *čas* $O(|S|)$.

*Důkaz.* Operace **MERGE** simuluje sčítání přirozených čísel v binárním zápise a má tedy stejnou složitost. Protože každý běh cyklu vyžaduje čas $O(1)$, algoritmus **MERGE** vyžaduje čas $O(\log(|S_1| + |S_2|))$.

Odhad složitosti vytváření haldy **MAKEHEAP** využívá známého faktu, že amortizovaná složitost přičítání 1 k binárnímu číslu je $O(1)$.

Odhad složitosti operací **MIN** a **DELETEMIN** je založen na pozorování, že binomiální halda reprezentující množinu $S$ má tolik stromů, kolik je jedniček v binárním zápise $|S|$, a to je nejvýše $\log(|S|)$.

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

□

---

Z těchto výsledků je vidět, že předchozí typy hald mají efektivnější chování než binomiální haldy. Význam binomiálních hald tak spočívá především v tom, že se dají dále zobecnit (tímto zobecněním jsou Fibonacciho haldy) a že na nich lze krásně ilustrovat princip, že s řadou úprav je výhodné počkat a neprovádět je okamžitě.

### 5.5.7 Líná binomiální halda

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

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

1. celkový počet vrcholů v těchto stromech je roven velikosti $S$ a existuje jednoznačné přiřazení prvků množiny $S$ vrcholům stromů, které splňuje podmínku (usp) – toto přiřazení je jako obvykle realizováno funkcí key;
2. každý strom $T_i$ je izomorfní s nějakým stromem $H_j$.

V líné binomiální haldě je vynechán předpoklad neizomorfnosti stromů tvořících haldu. Tento fakt se projeví ve velmi jednoduchém algoritmu pro operaci **MERGE**.

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

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

**vyvaz**$(\{O_i \mid i = 0, 1, \ldots, k\})$:
$i := 0$, $\mathcal{H} := \emptyset$
**while** existuje $O_i \neq \emptyset$ **do**
  **while** $|O_i| > 1$ **do**
    vezmi dva různé stromy $T_1$ a $T_2$ z $O_i$
    odstraň je z $O_i$
    **spoj**$(T_1, T_2)$ vlož do $O_{i+1}$
  **enddo**
  **if** $O_i \neq \emptyset$ **then**
    strom $T \in O_i$ odstraň z $O_i$ a vlož do $\mathcal{H}$
  **endif**,
  $i := i + 1$
**enddo**
**Výstup**: $\mathcal{H}$

**MIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
**Výstup**: nejmenší z těchto prvků
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ izomorfní s $H_i\}$
**vyvaz**$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

**DELETEMIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ izomorfní s $H_i$ různé od $T\} \cup$
$\{$podstrom $T$ určený nějakým synem kořene $T$ izomorfní s $H_i\}$
**vyvaz**$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

Časová složitost operací **INSERT** a **MERGE** při líné implementaci je $O(1)$, ale časová složitost operací

**MIN** a **DELETEMIN** je v nejhorším případě $O(|S|)$. Tento odhad je velmi špatný, ale ukážeme, že amortizovaná složitost má rozumné hodnoty. Připomínáme, že amortizovaná složitost je čas operace plus ohodnocení výsledné struktury minus ohodnocení počáteční struktury.

Konfiguraci ohodnotíme počtem stromů v haldě.

**Tvrzení.** *Amortizovaná složitost operací* **MERGE** *a* **INSERT** *je* $O(1)$.

*Důkaz.* Protože operace **MERGE** nemění počet stromů a protože operace **INSERT** přidá jen jeden strom, je amortizovaná složitost operací **MERGE** a **INSERT** stále $O(1)$. □

---

**Lemma.** *Operace* **vyvaz** *vyžaduje čas* $O(k + \sum_{i=0}^{k} |O_i|)$

*Důkaz.* Platí, protože každý běh vnitřního **while**-cyklu v operaci **vyvaz** vyžaduje čas $O(1)$ a zmenší počet stromů v seznamech $O_i$ o 1. □

---

**Pozorování.** *Operace* **MIN** *bez podprocedury* **vyvaz** *vyžaduje čas* $O(|\mathcal{H}|)$ *a operace* **DELETEMIN** *bez podprocedury* **vyvaz** *čas* $O(\mathcal{H} + i)$ *pro takové* $i$*, že* $T$ *je izomorfní s* $H_i$.

Ukážeme, že amortizovaná složitost operací **MIN** a **DELETEMIN** při líné implementaci binomiálních hald je $O(\log(|S|)$.

**Lemma.** *Operace* **MIN** *vyžaduje čas* $O(|\mathcal{H}|)$ *a operace* **DELETEMIN** *čas* $O(|\mathcal{H}| + \log(|S|))$

*Důkaz.* Podle tvrzení v části 5.5.3 je $i \leq \log(|S|)$, dále plyne z předchozího. □

---

**Věta.** *Amortizovaná složitost operace* **MIN** *a* **DELETEMIN** *je* $O(\log(|S|))$.

*Důkaz.* Ohodnocení klasické binomiální haldy je nejvýše $\log(|S|)$ (obsahuje tolik stromů, kolik je 1 v binárním zápise čísla $|S|$).

Z toho dostáváme, že amortizovaná složitost operace **MIN** je $O(|\mathcal{H}| - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$.

Také amortizovaná složitost operace **DELETEMIN** je $O(|\mathcal{H}| + \log(|S|) - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$.
□

---

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.

## 5.6 Fibonacciho haldy

### 5.6.1 Motivace

Význam Fibonacciho hald určuje fakt, že amortizovaná složitost operací **INSERT** a **DECREASE** v těchto haldách je $O(1)$ a amortizovaná složitost operace **DELETEMIN** je $O(\log(|S|)$. Proto se hodně používají v grafových algoritmech, kde umožňují v mnoha případech dosáhnout asymptoticky téměř lineární složitosti. Neznáme však žádné experimentální výsledky, které by porovnávaly použití Fibonacciho hald a např. $d$-regulárních hald v těchto grafových algoritmech v praxi. Takže neznáme podmínky, za kterých jsou Fibonacciho haldy lepší než třeba $d$-regulární haldy, ani nevíme, do jaké míry je to jen teoretický výsledek a do jaké míry jsou opravdu prakticky použitelné.

### 5.6.2 Velmi neformální definice

Fibonacciho halda je opět kolekce stromů. Jde tu ale o to, že ve Fibonacciho haldě je v každém stromě s $k$ syny alespoň $F_{k+2}$ prvků – toho dosáhneme tak, že se halda strašně „často“ rozpadá – každému vrcholu můžeme odebrat maximálně jednoho syna. Jakmile bychom mu chtěli odebrat dalšího, tak ho místo toho useknem a dáme jako další strom.

### 5.6.3 Méně neformální definice

Neformálně řečeno, je Fibonacciho halda množina stromů, jejichž některé vrcholy různé od kořenů jsou označeny, a kde existuje jednoznačná korepondence mezi prvky $S$ a vrcholy stromů (realizována funkcí key), která splňuje podmínku (usp).

Toto je však jen přibližné vyjádření. Existují totiž struktury, na které se tento popis hodí, ale nevznikly z prázdné Fibonacciho haldy aplikací posloupnosti haldových operací. Přitom důkaz efektivity Fibonacciho hald se dosti výrazně opírá o fakt, že halda vznikla z prázdné haldy aplikací algoritmů pro Fibonacciho haldy.

Proto nejprve popíšeme algoritmy pro tyto operace, a pak budeme definovat *Fibonacciho haldy* jako ty struktury vzniklé z prázdné haldy aplikací posloupnosti těchto algoritmů.

### 5.6.4 Algoritmy

V algoritmech předpokládáme, že Fibonacciho halda je seznam stromů, kde některé vrcholy různé od kořenů jsou označeny. Vrchol je označen, právě když není kořen a když mu byl někdy dříve odtržen některý jeho syn. Toto se nezaznamenává pro kořeny stromů. Proto když se vrchol stane kořenem (odtržením podstromu určeného tímto vrcholem), zapomene se tento údaj a začne se znovu zaznamenávat, až když vrchol přestane být kořenem. Řekneme, že strom má *rank* $i$, když jeho kořen má $i$ synů. Tento fakt nahrazuje test používaný v binomiálních haldách, že strom je izomorfní se stromem $H_i$.

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

| |
|---|
| **MERGE**$(\mathcal{H}_1, \mathcal{H}_2)$:<br>Proveď konkatenaci seznamů $\mathcal{H}_1$ a $\mathcal{H}_2$ |

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

**MIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
**Výstup**: nejmenší z těchto prvků
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ s rankem $i\}$
**vyvaz1**$(\{O_i \mid i = 0, 1, \ldots, \lfloor c \log(\sqrt{5}|S| + 1)\rfloor\})$

**DELETEMIN**:
Prohledej prvky reprezentované kořeny všech stromů v $\mathcal{H}$
$T :=$ strom, jehož kořen reprezentuje nejmenší prvek
stromy rozděl do množin $O_i = \{$všechny stromy v $\mathcal{H}$ s rankem $i$ různé od $T\} \cup$ $\{$podstrom $T$ určený některým synem kořene $T$ s rankem $i\}$
**vyvaz1**$(\{O_i \mid i = 0, 1, \ldots, \lfloor c \log(\sqrt{5}|S| + 1)\rfloor\})$

**vyvaz1**$(\{O_i \mid i = 0, 1, \ldots, k\})$:
$i := 0$, $\mathcal{H} := \emptyset$
**while** existuje $O_i \neq \emptyset$ **do**
  **while** $|O_i| > 1$ **do**
    vezmi dva různé stromy $T_1$ a $T_2$ z $O_i$
    odstraň je z $O_i$
    **spoj**$(T_1, T_2)$ vlož do $O_{i+1}$
  **enddo**
  **if** $O_i \neq \emptyset$ **then**
    strom $T \in O_i$ odstraň z $O_i$ a vlož ho 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**
  vyměň stromy $T_1$ a $T_2$
**endif**
vytvoř nového syna $v$ kořene $T_1$
$v :=$kořen $T_2$

**DECREASE**$(s, a)$:
$T :=$strom v $\mathcal{H}$, který obsahuje vrchol reprezentující $s$
$v :=$vrchol stromu $T$ reprezentující $s$
**if** $v$ není kořen **then**
  odtrhni podstrom $T'$ určený vrcholem $v$
  **vyvaz2**$(T, v)$
  **if** $v$ byl označen **then** zruš označení $v$ **endif**
  vlož $T'$ do $\mathcal{H}$
**endif**
$f(v) := f(v) - a$

**INCREASE**$(s, a)$:
$T$ :=strom v $\mathcal{H}$, který obsahuje vrchol reprezentující $s$
$v$ :=vrchol stromu $T$ reprezentující $s$
**if** $v$ není list **then**
  odtrhni podstrom $T'$ určený vrcholem $v$
  **if** $v$ není kořen **then vyvaz2**$(T, v)$ **endif**
  **if** $v$ byl označen **then** zruš označení $v$ **endif**
  zruš označení všech synů vrcholu $v$
  odtrhni podstromy $T'$ určené všemi syny $v$ a vlož je do $\mathcal{H}$
  do $\mathcal{H}$ vlož strom mající jen vrchol $v$
**endif**
$f(v) := f(v) + a$

**DELETE**$(s)$:
$T$ :=strom v $\mathcal{H}$, který obsahuje vrchol reprezentující $s$
$v$ :=vrchol stromu $T$ reprezentující $s$
**if** $v$ není list **then**
  zruš označení synů vrcholu $v$
  odtrhni podstromy určené všemi syny vrcholu $v$ a vlož je do $\mathcal{H}$
**endif**
**if** $v$ není kořen **then vyvaz2**$(T, v)$ **endif**
zruš vrchol $v$

**vyvaz2**$(T, v)$:
$u :=$ otec $v$
**while** $u$ je označen **do**
  $u' :=$ otec$(u)$, zruš označení $u$
  odtrhni podstrom $T'$ určený vrcholem $u$
  vlož $T'$ do $\mathcal{H}$, $u := u'$
**enddo**
**if** $u$ není kořen $T$ **then** označ $u$ **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 $c \log(\sqrt{5}|S| + 1)$. 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 tento vrchol stane kořenem stromu). Když se později stane tento vrchol zase vrcholem různým od kořene, celý proces se opakuje.

### 5.6.5 Složitost operací

Naším cílem bude odhadnout amortizovanou složitost těchto operací, protože složitost v nejhorším případě není použitelný výsledek. Abychom to mohli udělat, spočítáme parametry složitosti 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, označené vrcholy se nemění;

**DELETEMIN** – časová složitost $O(|\mathcal{H}| + \text{počet synů } v)$, kde $v$ reprezentoval prvek s nejmenší hodnotou $f$. Po provedení operace různé stromy v haldě mají různé ranky, žádný nový vrchol nebyl označen, některé označené vrcholy přestaly být označené;
**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 vrchol;
**INCREASE** – časová složitost $O(1 + c + d)$, kde $c$ je počet vrcholů, které přestaly být označené, $d$ je počet synů vrcholu $v$ reprezentujícího prvek, jehož hodnota se zvyšuje. Bylo přidáno nejvýše $1 + c + d$ nových stromů a byl označen nejvýše jeden vrchol;
**DELETE** – časová složitost $O(1 + c + d)$, kde $c$ je počet vrcholů, které přestaly být označené, $d$ je počet synů vrcholu $v$ reprezentujícího prvek, který se má odstranit. Bylo přidáno nejvýše $c + d$ nových stromů a byl označen nejvýše jeden vrchol.

Pro výpočet amortizované složitosti musíme nejprve navrhnout funkci ohodnocující konfigurace. Při vyšetřování líné implementace binomiálních hald se ukázalo, že vhodným ohodnocením je počet stromů v haldě. Když si ale prohlédneme algoritmus pro operaci **DECREASE**, vidíme, že zde je vhodné brát do ohodnocení i počet označených vrcholů, a to dokonce tak, aby se pokryl nejen čas, ale i přírůstek stromů. To vede k následujícímu ohodnocení konfigurace: ohodnocení je počet stromů v konfiguraci plus dvojnásobek počtu označených vrcholů.

**Tvrzení.** *Nechť $\rho(n)$ je maximální počet synů vrcholu ve Fibonacciho haldě reprezentující $n$-prvkovou množinu. Pak amortizovaná složitost operací* **MERGE**, **INSERT** *a* **DECREASE** *je $O(1)$ a operací* **MIN**, **DELETEMIN**, **INCREASE** *a* **DELETE** *je $O(\rho(n))$.*

*Důkaz.* Tohle mi *vůbec* není jasné, z čeho by mělo plynout. □

---

Abychom spočítali odhad $\rho(n)$, využijeme toho, že Fibonacciho halda vznikla z prázdné haldy pomocí popsaných algoritmů. Nejprve uvedeme 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.* V momentě, kdy se $u$ stával synem $v$, se aplikovala operace **spoj**, $u$ a $v$ byly kořeny stromů a měly stejný počet synů. Podle předpokladů měl vrchol $v$ alespoň $i-1$ synů (jinak by $u$ nebyl $i$-tý nejstarší syn), a protože se od $u$ 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 indukcí 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 $1 = F_2 = F_{0+2}$, tvrzení platí. Mějme nyní 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 tato délka je menší než $j$, tedy platí i 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 indukčního předpokladu 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 $F_1 = F_2$ (na levé straně první 1 je za vrchol $v$ a první $F_2$ je za nejstarší vrchol). Indukcí pak dostaneme, že

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

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

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

pro $n = 1$ máme

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

a z indukčního předpokladu a z vlastností Fibonacciho čísel plyne, 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 $F_{i+2}$ vrcholů, a tvrzení je dokázáno. □

---

Pozn. studenta – zde se naprosto ztrácím v toku vzorců. Snad aspoň píšu dobře, co se snažíme dokazovat.

**Lemma.** $\rho(n) < \frac{\log_2(\sqrt{5}n+1)}{(\log_2 3)-1} - 2$

*Důkaz.* Vezměme nyní 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 $\sqrt{5} > 2$, dostáváme, že $|\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^i| < \frac{3}{8}$ 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 plyne, že když $i$ splňuje

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

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

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

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

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

Tedy platí následující implikace

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

Proto když $\frac{\log_2(\sqrt{5}n+1)}{\log_2 3-1} < i$, 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*

$$\frac{\log_2(\sqrt{5}n+1)}{(\log_2 3)-1} - 2.$$

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

*Důkaz.* 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. □

### 5.6.6 Aplikace

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

Otázka je, kdy v Dijkstrově algoritmu nebo v algoritmu konstruujícím nejmenší napnutou kostru použít Fibonacciho haldu a kdy např. $d$-regulární haldy. 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 ani teoretických výsledcích tohoto typu.

### 5.6.7 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 navrhnl Vuillemin 1978, Brown 1978 je implementoval a prokázal jejich praktickou použitelnost. Fibonacciho haldy byly zavedeny Fredmanem a Tarjanem 1987.

# 6 Třídicí algoritmy

Pozn. studenta – zde jsem to již vzdal; tedy zbytek skript je téměř neupraven. Možná se k tomu někdy dostanu.

Jednou z nejčastěji řešených úloh při práci s daty je setřídění posloupnosti prvků nějakého typu. Proto velká pozornost byla a je věnována třídicím algoritmům řešícím tuto úlohu, která svým charakterem a svými požadavky na algoritmy je řazena do datových struktur. Byla navržena řada algoritmů, které se stále ještě analyzují a optimalizují. Analýzy jsou velmi detailní a algoritmy se studují za různých vstupních předpokladů. Kromě toho třídění je jedna z mála úloh, pro kterou alespoň za jistých předpokladů umíme spočítat dolní odhad složitosti.

Formulace úlohy:
Nechť $U$ je totálně uspořádané univerzum.
Vstup: Prostá posloupnost $\{a_1, a_2, \ldots, a_n\}$ prvků z univerza $U$.
Výstup: Rostoucí posloupnost $\{b_1, b_2, \ldots, b_n\}$ taková, že $\{a_i \mid i = 1, 2, \ldots, n\} = \{b_i \mid i = 1, 2, \ldots, n\}$.
Tento problém se nazývá *třídění*. V praxi se setkáváme s řadou jeho modifikací, a nichž asi nejběžnější je vynechání předpokladu, že vstupem je prostá posloupnost. Pak jsou dvě varianty řešení – buď se ve výstupní posloupnosti odstraní duplicity nebo výstupní posloupnost zachová četnost prvků ze vstupní posloupnosti.

Základní algoritmy, které řeší třídicí problém, jsou **QUICKSORT**, **MERGESORT** a **HEAPSORT**.

### 6.0.8 HEAPSORT

S algoritmem **HEAPSORT** jsme se seznámili při aplikacích hald. Byl to první algoritmus používající haldy (binární regulární haldy byly definovány právě při návrhu **HEAPSORTU**). Podíváme se detailněji na jednu z jeho implementací, která třídí takzvaně 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**. Zvolíme implementaci **HEAPSORTU** pomocí $d$-regulárních hald, které jsou reprezentovány polem, v němž je uložena vstupní posloupnost (viz odstavec Aplikace v kapitole o $d$-regulárních haldách). Použijeme algoritmus s jedinou změnou – budeme požadovat duální podmínku na uspořádání (to znamená, že prvek reprezentovaný vrcholem bude menší než prvek reprezentovaný jeho otcem) 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. Musíme si ale pamatovat, kde v poli končí reprezentovaná halda. Každá aplikace operace **DELETEMAX** zkrátí počáteční úsek pole reprezentujícího haldu o jedno místo a zároveň o toto místo zvětší druhou část, ve které je uložena již setříděná část posloupnosti.

**HEAPSORTU** je stále věnována velká pozornost a bylo navrženo několik jeho modifikací, snažících se např. minimalizovat počet porovnání prvků apod.

### 6.0.9 MERGESORT

Nejstarší z uvedených algoritmů je **MERGESORT**, který je starší než je počítačová éra, neboť některé jeho verze se používaly už při mechanickém třídění. Popíšeme jednu jeho iterační verzi, tzv. přirozený **MERGESORT**.

**MERGESORT**$(a_1, a_2, \ldots, a_n)$:
$Q$ je prázdná fronta, $i = 1$
**while** $i \leq n$ **do**
  $j := i$
  **while** $i < n$ a $a_{i+1} > a_i$ **do** $i := i + 1$ **enddo**
  posloupnost $P = (a_j, a_{j+1}, \ldots, a_i)$ vlož do $Q$
  $i := i + 1$
**enddo**
**while** $|Q| > 1$ **do**
  vezmi $P_1$ a $P_2$ dvě posloupnosti z vrcholu $Q$
  odstraň $P_1$ a $P_2$ z $Q$
  **MERGE**$(P_1, P_2)$ vlož na konec $Q$
**enddo**
**Výstup**: posloupnost z $Q$

**MERGE**$(P_1 = (a_1, a_2, \ldots, a_n), P_2 = (b_1, b_2, \ldots, b_m))$:
$P :=$ je 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 sjednocením všech jejich prvků je vždy na začátku běhu cyklu **while** $|Q| > 1$ **do** množina $\{a_i \mid i = 1, 2, \ldots, n\}$. Protože počet posloupností ve frontě $Q$ je nejvýše roven délce vstupní posloupnosti a každý průběh tohoto cyklu zmenší jejich počet o 1, je algoritmus **MERGESORT** korektní.

Spočítáme časovou složitost **MERGESORTU**. Nejprve vyšetříme složitost podprocedury **MERGE**. Protože určení prvku $c_k$ vyžaduje čas $O(1)$ (provede se nejvýše jedno porovnání) a protože maximální hodnota $k$ je $n + m$, dostáváme, že podprocedura **MERGE** vyžaduje čas $O(n + m)$ (nejvýše $n + m$ porovnání), kde $n$ a $m$ jsou délky vstupních posloupností.

Nyní vypočteme složitost hlavní procedury. Zřejmě první cyklus vyžaduje lineární čas. Vyšetříme druhý cyklus probíhající přes frontu $Q$. Předpokládejme, že před prvním během tohoto cyklu je na vrcholu $Q$ speciální znak $\natural$, který se vždy pouze přenese z vrcholu $Q$ na její konec. Protože mezi dvěma přenosy $\natural$ projde každý prvek vstupní posloupnosti podprocedurou **MERGE** právě jednou, vyžadují jednotlivé běhy cyklu čas $O(n)$, kde $n$ je délka vstupní posloupnosti (a zároveň součet všech délek posloupností v $Q$). Všechny posloupnosti v $Q$ mají na počátku délku $\geq 1$. Odtud jednoduchou indukcí dostaneme, že po $i$-tém přenosu znaku $\natural$ mají délku $\geq 2^{i-1}$. Proto počet přenosů je nejvýše $\lceil \log_2 n \rceil$, a tedy algoritmus **MERGESORT** vyžaduje čas $O(n \log n)$ (provede se nejvýše $n \log n$ porovnání).

Vzhledem k počtu porovnání je **MERGESORT** optimální třídicí algoritmus. Navíc v této verzi je adaptivní na předtříděné posloupnosti, které mají jen malý počet dlouhých setříděných úseků (běhů). Při konstantním počtu běhů má složitost $O(n)$. Jiná jeho verze, která začíná slévání vždy od jednoprvkových posloupností (tzv. přímý **MERGESORT**) tuto vlastnost nemá.

### 6.0.10 QUICKSORT

Nyní popíšeme patrně vůbec nejpoužívanější třídicí algoritmus, kterým je **QUICKSORT**. Důvodem je, že pro obecnou posloupnost je nejrychlejší, při rovnoměrném rozložení vstupních polsoupností má nejmenší očekávaný čas.

**Quick**$(a_i, a_{i+1}, \ldots, a_j)$:
**if** $i = j$ **then**
  **Výstup**: $(a_i)$
**else**
  zvol $k$ takové, že $i \leq k \leq j$, $a := a_k$, vyměň $a_i$ 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** $l \geq q$ **then**
      exit
    **else**
      vyměň $a_l$ a $a_q$, $l := l + 1$, $q := q - 1$
    **endif**
  **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**$(a_{i+1}, a_{i+2}, \ldots, a_{l-1}), a)$
    **else**

**Výstup**(**Quick**$(a_{i+1}, a_{i+2}, \ldots, a_{l-1}), a,$**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** setřídí posloupnost $(a_i, a_{i+1}, \ldots, a_j)$ tak, že pro prvek $a = a_k$ vytvoří posloupnost $(a_i, a_{i+1}, \ldots, a_{l-1})$ všech prvků menších než $a$ a posloupnost $(a_{q+1}, \ldots, a_j)$ všech prvků větších než $a$. Na tyto posloupnosti pak zavolá sám sebe a do výsledné posloupnosti uloží nejprve 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 (tj. prostřední prvek) posloupnosti $(a_i, a_{i+1}, \ldots, a_j)$, pak by algoritmus **QUICKSORT** v nejhorším případě vyžadoval čas $O(n \log n)$. Jak uvidíme později, medián lze sice nalézt v lineárním čase, ale použití jakékoli procedury pro jeho nalezení má za následek, že algoritmy **MERGESORT** a **HEAPSORT** budou rychlejší (nikoliv asymptoticky, ale multiplikativní konstanta bude v tomto případě vysoká). Proto je třeba vybrat prvek $a_k$ (tzv. pivot) co nejrychleji. Původně se bral první nebo poslední prvek posloupnosti. Při této volbě a při rovnoměrném rozdělení vstupů je očekávaný čas **QUICKSORTU** $O(n \log n)$ a algoritmus je obvykle rychlejší než **MERGESORT** a **HEAPSORT**. Avšak čas v nejhorším případě je kvadratický a dokonce pro určitá rozdělení vstupních dat je i očekávaný čas kvadratický. Proto tuto volbu pivota není vhodné používat pro úlohy, kdy neznáme rozdělení vstupních dat (mohlo by se stát, že je nevhodné). Jednoduše to lze napravit tak, že budeme volit $k$ náhodně. Bohužel použití pseudonáhodného generátoru také vyžaduje jistý čas, a pak už by algoritmus zase nemusel být rychlejší než algoritmy **MERGESORT** a **HEAPSORT** (navíc takto náhodně zvolený prvek není skutečně náhodný, ale to v tomto případě nevadí). Důsledkem je návrh vybírat pivota jako medián ze 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** jako argument kratší vstupní posloupnost, lze ukázat, že:

1. při každé volbě pivota je nejhorší čas algoritmu **QUICKSORT** $O(n^2)$,
2. pokud je pivot vybrán jednoduchým a rychlým způsobem (to platí, i když se volí náhodně), pak existují vstupní posloupnosti, které vyžadují čas $O(n^2)$,
3. očekávaný čas je $O(n \log n)$.

Následná analýza očekávaného případu 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 způsoby výpočty očekávaného času. Jeden je založen na několika jednoduchých pozorováních a není v něm mnoho počítání, druhý na rekurzivním výpočtu. Ten je početně náročnější, ale postup je standardní. Hlavní idea v obou případech spočívá v tom, že očekávaný čas algoritmu **QUICKSORT** je úměrný očekávanému počtu porovnání v algoritmu **QUICKSORT**. Tento fakt plyne přímo z popisu algoritmu. Budeme tedy počítat očekávaný počet porovnání pro algoritmus **QUICKSORT**.

První způsob výpočtu:
Každé dva prvky $a_i$ a $a_j$ algoritmus **QUICKSORT** porovná při třídění posloupnosti $(a_1, a_2, \ldots, a_n)$ nejvýše jednou, přičemž když porovnává $a_i$ a $a_j$, pak pro nějaký běh podprocedury **Quick** je $a_i$ nebo $a_j$ pivot, ale v předchozích bězích **Quick** $a_i$ ani $a_j$ nebyl pivotem (protože pivot je vždy vyřazen 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 $b_j$, a 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 součtem 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**. Vrchol $v$ bude vnitřním vrcholem, když odpovídající podprocedura volila pivota, a tento pivot bude ohodnocením $v$. V podstromu levého syna vrcholu $v$ budou právě všechna následující rekurzivní volání podprocedury **Quick** nad částí posloupnosti, která předchází pivotu. Analogicky v podstromu pravého syna vrcholu $v$ budou právě všechna následující rekurzivní volání procedury **Quick** nad částí posloupnosti, která následuje po pivotu. Listy stromu odpovídají volání procedury **Quick** nad jednoprvkovými posloupnostmi a každý takový jednotlivý prvek ohodnocuje příslušný list. Když vrchol $v$ odpovídá volání **Quick** nad posloupností $(a_i, a_{i+1}, \ldots, a_j)$, pak vrcholy v podstromu levého syna $v$ jsou ohodnoceny prvky z posloupnosti $(a_i, a_{i+1}, \ldots, a_{l-1})$ a vrcholy v podstromu pravého syna vrcholu $v$ jsou ohodnoceny prvky z posloupnosti $(a_{q+1}, \ldots, a_j)$ (po přerovnání). 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 $(c_1, c_2, \ldots, c_n)$ je posloupnost prvků $\{a_i \mid 1 \leq i \leq n\}$ v pořadí daném tímto očíslováním. Pak platí, že $X_{i,j} = 1$, právě když první prvek v posloupnosti $(c_1, c_2, \ldots, c_n)$ z množiny $\{b_l \mid i \leq l \leq j\}$ je buď $b_i$ nebo $b_j$. Pravděpodobnost tohoto jevu je $\frac{2}{j-i+1}$, tedy $p_{i,j} = \frac{2}{j-i+1}$ pro $1 \leq i < j \leq n$. Odtud očekávaný počet porovnání v algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \leq 2n(\sum_{k=2}^{n} \frac{1}{k}) \leq 2n \int_1^n \frac{1}{x} dx = 2n \ln n.$$

Druhý způsob výpočtu:
Označme $QS(n)$ očekávaný počet porovnání provedených algoritmem **QUICKSORT** při třídění $n$-členné posloupnosti. Pak platí

$$QS(0) = QS(1) = 0 \text{ a}$$

$$QS(n) = \frac{1}{n}\big(\sum_{k=0}^{n-1} n - 1 + QS(k) + QS(n-k-1)\big) = n - 1 + \frac{2}{n}(\sum_{k=0}^{n-1} QS(k)).$$

Z toho dostáváme, že

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

Přepíšeme ještě jednou tuto rovnici s $n+1$ místo $n$:

$$(n+1)QS(n+1) = (n+1)n + 2\sum_{k=0}^{n} QS(k).$$

Od této rovnice odečteme rovnici předchozí a po jednoduché úpravě získáme rekurentní vztah

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

Postupným dosazováním dostaneme řešení

$$\begin{aligned} QS(n) = & \sum_{i=2}^{n} \frac{n+1}{i+1} \frac{2(i-1)}{i} \leq 2(n+1)\Big(\sum_{i=2}^{n} \frac{1}{i+1}\Big) = 2(n+1)\Big(\sum_{i=3}^{n+1} \frac{1}{i}\Big) = \\ & 2(n+1)\Big(\sum_{i=2}^{n+1} \frac{1}{i} - \frac{1}{2}\Big) \leq 2(n+1)\Big(\Big(\int_{i=1}^{n+1} \frac{1}{x} dx\Big) - \frac{1}{2}\Big) = \\ & 2n\ln(n+1) + 2\ln(n+1) - n - 1. \end{aligned}$$

Pro dostatečně velká $n$ tedy platí

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

### 6.0.11 Porovnání třídicích algoritmů

Nyní porovnáme složitost algoritmů **HEAPSORT**, **MERGESORT**, **QUICKSORT**, **A-sort** (byl popsán v kapitole o $(a,b)-$stromech), **SELECTIONSORT** a **INSERTIONSORT**. Připomeňme si, že **SELECTIONSORT** třídí posloupnost tak, že jedním průchodem nalezne její nejmenší prvek, který vyřadí a vloží do výsledné posloupnosti (ve verzi, která třídí na místě, ho vymění s levým krajním prvkem pole). Tento proces pak opakuje se zbytkem původní posloupnosti. Tato idea byla základem algoritmu **HEAPSORT**. **INSERTIONSORT** třídí tak, že do již setříděného začátku posloupnosti vkládá další prvek, který pomocí výměn zařadí na správné místo, a tento proces (začíná druhým prvkem zleva) opakuje.

**QUICKSORT** v nejhorším případě vyžaduje čas $\Theta(n^2)$, očekávaný čas je $9n\log n$, v nejhorším případě provádí $\frac{n^2}{2}$ porovnání, očekávaný počet porovnání je $1.44n\log n$. Potřebuje $n + \log n + konst$ 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 provádí $2n\log n$ porovnání. Potřebuje $n + konst$ 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 provádí $n\log n$ porovnání (nejmenší možný počet). Potřebuje $2n + konst$ paměti, používá sekvenční přístup k paměti a má verzi, která je adaptivní na předtříděné posloupnosti s malým počtem běhů.
**A-sort** v nejhorším případě i v očekáném případě vyžaduje čas $O(n\log\frac{F}{n})$, kde $F$ je počet inverzí ve vstupní posloupnosti, v nejhorším i v očekávaném případě provádí $O(n\log\frac{F}{n})$ porovnání. Potřebuje $5n + konst$ paměti, používá přímý přístup k paměti a je adaptivní na předtříděné posloupnosti s malým počtem inverzí.
**SELECTIONSORT** v nejhorším i v očekávaném případě vyžaduje čas $2n^2$, počet porovnání v nejhorším i v očekávaném případě je $\frac{n^2}{2}$. Potřebuje $n+konst$ paměti, používá přímý přístup k paměti a není adaptivní na předtříděné posloupnosti.
**INSERTIONSORT** v nejhorším i v očekávaném případě vyžaduje čas $O(n^2)$, počet porovnání v nejhorším případě je $\frac{n^2}{2}$, v očekávaném případě $\frac{n^2}{4}$. Potřebuje $n + konst$ paměti, používá sekvenční přístup k paměti a má verzi, která je adaptivní na předtříděné posloupnosti s malým počtem inverzí.

Prezentované výsledky byly spočítány pro model RAM (viz Mehlhorn 1984).

Očekávaný čas pro **HEAPSORT** je prakticky stejný jako jeho nejhorší čas. Byly navrženy verze, které optimalizují počet porovnání, ale většinou mají větší nároky na čas, a proto až na výjimky nejsou pro

praktické použití vhodné. Situace pro **MERGESORT** je komplikovanější, hodně závisí na konkrétní verzi algoritmu. Algoritmus **MERGESORT** je nejvhodnější pro externí paměti se sekvenčním přístupem k datům, pro interní paměť kvůli velké prostorové náročnosti není doporučován (je např. dvojnásobná proti **HEAPSORTU** a téměř dvojnásobná proti **QUICKSORTU**). Také se hodí pro návrh paralelních algoritmů. Pro třídění krátkých posloupností je doporučováno místo **QUICKSORTU** pro posloupnosti délky $\leq 22$ použít **SELECTIONSORT** a pro posloupnosti délky $\leq 15$ **INSERTIONSORT.** To vede k návrhu optimalizovaného **QUICKSORTU**, který**,** 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 spotřebovaných algoritmy **QUICKSORT**, **MERGESORT** a **HEAPSORT** na klasických počítačích uvádí Mehlhorn (1984) jako $1 : 1.33 : 2.22$. To však nemusí být pravda pro současné procesory, paměti a operační systémy.

### 6.0.12 Slévání nestejně dlouhých posloupností

V algoritmu **MERGESORT** jsme použili frontu, která řídila proces slučování rostoucích posloupností. Tato metoda je uspokojující a dává optimální výsledek (ve smyslu časové náročnosti)**,** pokud posloupnosti ve frontě jsou stejně dlouhé. Pokud se ale jejich délky hodně liší, nedosáhneme tímto způsobem optimálního výsledku. Přitom různé verze tohoto problému se vyskytují v mnoha úlohách. Jednou z prvních úloh, kde jsme se s ním setkali, je konstrukce Huffmanova kódu – to je minimální redundantní kód, který byl nalezen v roce 1952. K optimálnímu řešení vede např. postup, který je kombinací 'mergeování' a optimalizace a používá metody dynamického programování. Nejprve formálně popíšeme abstraktní verzi tohoto problému.

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 z daných rostoucích posloupností vytvoří jedinou rostoucí posloupnost. Tento postup určuje úplný binární strom $T$, jehož listy jsou ohodnoceny vstupními posloupnostmi a každý vnitřní vrchol je ohodnocen posloupností, která je sloučením vstupních posloupností ohodnocujících listy v podstromu určeném tímto vrcholem. Tedy kořen je ohodnocen výstupní posloupností. Formálně pro každý vnitřní vrchol $v$ platí:

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

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

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

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

Když tedy $T$ je úplný binární strom, jehož listy jsou ohodnoceny navzájem disjunktními rostoucími posloupnostmi, pak následující algoritmus **Slevani** spojí tyto posloupnosti do jediné rostoucí posloupnosti a procedury **MERGE** budou vyžadovat celkový čas

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

**Slevani**$(T, \{P(l) \mid l \text{ je list } T\})$
**while** $P$(kořen $T$) není definováno **do**

$v :=$ vrchol $T$ 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
$P(v) :=$**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 $x_1, x_2, \ldots, x_n$.

V přeformulováné úloze už nepracujeme s posloupnostmi, ale jen s jejich délkami. To znamená, že když pro původní úlohu byly vstupem posloupnosti $P_1, P_2, \ldots, P_n$, pak pro přeformulovanou úlohu jsou vstupem jen délky $l(P_1), l(P_2), \ldots, l(P_n)$. Strom vytvořený pro přeformulovanou úlohu je použit v algoritmu **Slevani** tak, že posloupnost $P_i$ ohodnocuje list, který byl v přeformulované úloze ohodnocen délkou $l(P_i)$, a hledaná posloupnost v původní úloze ohodnocuje kořen stromu.

Mějme množinu $\{x_i \mid i = 1, 2, \ldots, n\}$. 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. K řešení použijeme následující algoritmus, který je upravenou verzí hladového algoritmu pro náš problém.

**Optim**$(x_1, x_2, \ldots x_n)$:
$V$ je množina $n$ jednoprvkových stromů
$\phi$ je bijekce mezi $\{1, 2, \ldots, n\}$ a množinou $V$
**for every** $v \in V$ **do** $c(v) := x_{\phi^{-1}(v)}$ **enddo**
**while** $|V| > 1$ **do**
vezmi z $V$ dva stromy $v_1$ a $v_2$ s nejmenším ohodnocením
odstraň je z $V$
vytvoř nový strom $v$ spojením stromů $v_1$ a $v_2$
$c(v) := c(v_1) + c(v_2)$, strom $v$ vlož do $V$
**enddo**
**Výstup**: $(T, \phi)$, kde $T$ je strom v množině $V$

Vytvoření nového stromu $v$ spojením stromů $v_1$ a $v_2$ znamená vytvoření nového vrcholu, který bude kořenem stromu $v$ a jehož synové budou kořeny stromů $v_1$ a $v_2$. To je analogické proceduře **spoj**.

**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 čase* $O(n)$.

*Důkaz.* Důkaz má dvě části. V první dokážeme korektnost algoritmu a ve druhé popíšeme reprezentaci množiny $V$ a vypočteme časovou složitost.

Nejprve připomeňme, že $\phi(i)$ je list $T$ pro každé $i \in \{1, 2, \ldots, n\}$. Protože na začátku $V$ obsahuje jen jednoprvkové stromy, tak tvrzení platí. Každý běh cyklu **while do** zmenší počet stromů $V$ o jeden, ale nezmění množinu listů. Proto $T$ je strom s $n$ listy, $\phi$ je bijekce z $\{1, 2, \ldots, n\}$ do množiny listů $T$ a algoritmus vždy končí. Dokážeme indukcí podle $n$, že zkonstruovaná dvojice $(T, \phi)$ je optimální strom vzhledem k $(x_1, x_2, \ldots, x_n)$. Když $n = 2$, tvrzení zřejmě platí. Předpokládejme, že 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 neklesající 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 $T'$ je strom získaný ze stromu $T$ 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 $\psi(n-1)$ je otec listu $\phi(1)$. Pak můžeme předpokládat, že algoritmus **Optim**$(y_1, y_2, \ldots, y_{n-1})$ zkonstruoval strom $(T', \psi)$, a 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 $u$ stromu $U$ takový, že délka cesty z kořene do vrcholu $u$ je největší mezi všemi vnitřními vrcholy stromu $U$. Nechť $u_1$ a $u_2$ jsou synové $u$, pak nutně $u_1$ a $u_2$ jsou listy stromu $U$. Nechť $i, j \in \{1, 2, \ldots, n\}$ takové, že $\theta(i) = u_1$, $\theta(j) = u_2$. Po eventuálním přejmenování 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

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

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 $(y_1, y_2, \ldots, y_{n-1})$, platí, že

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

Protože

$$\begin{aligned} \mathrm{Cont}(T, \phi) &= \mathrm{Cont}(T, \psi) + x_1 + x_2 \ , \\ \mathrm{Cont}(U, \eta) &= \mathrm{Cont}(U', \tau) + x_1 + x_2 \end{aligned}$$

pak závěr je, že $(T, \phi)$ je optimální strom pro $(x_1, x_2, \ldots, x_n)$.

Předpokládejme opět, že $x_1 \leq x_2 \leq \cdots \leq x_n$ a že v daném okamžiku jsou $v_1, v_2, \ldots, v_k$ postupně vytvořené víceprvkové stromy (tj. strom $v_i$ byl vytvořen před stromem $v_j$, když $i < j$). V tomto okamžiku je množina $V$ sjednocením množiny $\{v_1, v_2, \ldots, v_k\}$ a množiny jednoprvkových stromů, které nebyly ještě zpracovány. Nyní vytvoříme strom $w$ spojením stromů $t_1$ a $t_2$ s nejmenším ohodnocením. Z popisu algoritmu plyne, že když strom $v_i$ pro $i = 1, 2, \ldots, k$ vznikl spojením stromů $u_1$ a $u_2$, pak $\max\{c(u_1), c(u_2)\} \leq \min\{c(t_1), c(t_2)\}$, a proto $c(w) \geq c(v_i)$ pro každé $i = 1, 2, \ldots, k$. Pak indukcí okamžitě dostáváme, že $c(v_1) \leq c(v_2) \leq \cdots \leq c(v_k)$. Tedy stačí, abychom měli rostoucí posloupnost listů a v ní ukazatel na nejmenší list, který je ještě nezpracovaným jednoprvkovým stromem (tj. před ukazatelem jsou listy, které už nejsou stromy v množině $V$, za ukazatelem jsou listy, které jsou ještě jednoprvkové stromy v množině $V$) a frontu víceprvkových stromů (z níž stromy ke zpracování odebíráme zpředu a nově vytvořené ukládáme na konec). Udržovat tyto struktury vyžaduje čas $O(1)$ stejně jako nalezení dvou stromů s nejmenším ohodnocením. Můžeme tedy shrnout, že algoritmus **Optim** konstruuje optimální stromy v čase $O(n)$, kde $n$ je počet zadaných čísel $x_i$. □ □

---

Pro aplikaci na naši původní úlohu je třeba ještě setřídit vstupní posloupnost délek pro přeformulovanou úlohu. Tato posloupnost je tvořena přirozenými čísly a k jejímu setřídění můžeme použít algoritmus **BUCKETSORT** (bude popsán dále v textu), který vyžaduje čas $O(n + m)$, kde $n$ je počet posloupností a $m$ je maximální délka posloupnosti.

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

## 6.1 Rozhodovací stromy

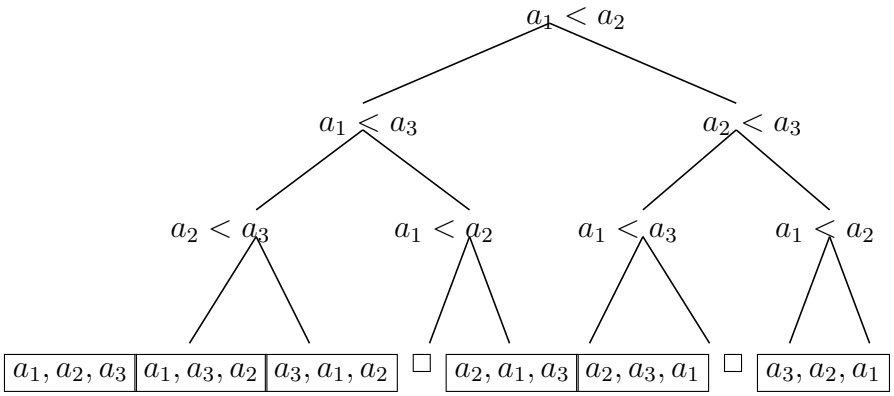
Většina obecných třídicích algoritmů používá jedinou primitivní operaci mezi prvky vstupní posloupnosti, a to jejich vzájemné porovnání. To znamená, že práci takového algoritmu lze popsat binárním stromem, jehož vnitřní vrcholy jsou ohodnoceny porovnáními dvojic prvků vstupní posloupnosti (např. $a_i < a_j$). Bez újmy na obecnosti předpokládejme, že vstupní posloupnost je permutace $\pi$ množiny $\{1, 2, \ldots, n\}$. Tato permutace prochází stromem takto:

1. ""Začíná v kořeni stromu. Když je ve vnitřním vrcholu $v$ ohodnoceném porovnáním $a_i \leq a_j$, pak když $\pi(i) < \pi(j)$, pokračuje v levém synu vrcholu $v$, a když $\pi(j) < \pi(i)$, pokračuje 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. Tedy strom popisující korektní algoritmus pro setřídění $n$-prvkovýh posloupností musí mít alespoň $n!$ listů. Délka cesty z kořene do listu, kde skončila permutace $\pi$, reprezentuje počet porovnání, které potřebuje daný algoritmus k setřídění dané posloupnosti $\pi$. Protože porovnání vyžaduje alespoň jednotku času, dostáváme tím i dolní odhad na čas potřebný k setřídění této posloupnosti algoritmem odpovídajícím danému stromu. Dolní odhad počtu porovnání i času pro daný algoritmus a všechny $n$-prvkové posloupnosti je pak délka nejdelší cesty z kořene do listu v odpovídajícím stromu. To nám umožňuje získat obecně platný dolní odhad času potřebného k setřídění $n-$prvkové posloupnosti, kterým je minimum přes všechny binární stromy s alespoň $n!$ listy z jejich maximálních délek cest z kořene do listu. Korektnost těchto úvah plyne z pozorování, že když porovnání je jediná primitivní operace, pak algoritmus není závislý na konkrétních 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 nastane, když strom pro $n$-prvkové posloupnosti má více než $n!$ listů, nebo, jinak řečeno, když porovnání dvou stejných prvků se na nějaké cestě vyskytne alespoň dvakrát.

Následující obrázek ilustruje naše úvahy na **SELECTIONSORTU** pro 3-prvkové posloupnosti. Listy jsou ohodnoceny permutacemi vstupní množiny $\{a_1, a_2, a_3\}$, které v nich skončí, nebo jsou prázdné.




Obrázek 11: SELECTIONSORT - postup

**Definice.** *Mějme třídicí algoritmus* **A**, *který jako jedinou primitivní operaci s prvky vstupní posloupnosti používá jejich 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*

1. *""posloupnost porovnání při třídění permutace $\pi$ algoritmem* **A** *je stejná jako posloupnost porovnání při průchodu permutace $\pi$ stromem $T$.*

Pak korektnost algoritmu zajišťuje, že dvě různé permutace množiny $\{1, 2, \ldots, n\}$ skončí v různých listech stromu $T$ a dolním odhadem pro čas algoritmu **A** v nejhorším případě je délka nejdelší cesty z kořene do listu. Při rovnoměrném rozdělení vstupních posloupností je očekávaný čas algoritmu **A** roven průměrné délce cesty z kořene do listu.

Definujme
$S(n)$ jako minimum přes všechny stromy $T$ s alespoň $n!$ listy z délek nejdelších cest z kořene do listu v $T$,
$A(n)$ 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 odhady 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, že $S(n) \geq \log_2 n!$. Připomeňme si Stirlingův vzorec pro faktoriál:

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

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

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

Protože

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

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

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

Dále pro binární strom $T$ označme $B(T)$ součet všech délek cest z kořene do listů a položme

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

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

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

Dokažme tedy, 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 listem, pak $B(T) = 0 = 1\log_2 1$, když $T$ je úplný binární strom se 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 a nechť $T_i$ má $k_i$ listů, kde $i = 1, 2$. 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

$$B(T) = k_1 + B(T_1) + k_2 + B(T_2) \geq k + B(k_1) + B(k_2) \geq k + k_1\log_2 k_1 + k_2\log_2 k_2.$$

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 $k = k_1 + k_2$. To 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 $f(\frac{k}{2}) = 0$ a počítejme derivaci $f$.

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

Nyní když $x \in (0, \frac{k}{2})$, pak $f'(x) < 0$ a $f$ je na tomto intervalu klesající, když $x \in (\frac{k}{2}, k)$, 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 $A(n) \geq (n + \frac{1}{2}) \log_2 n - \frac{n}{\ln 2}$. 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 $\lceil (n+\frac{1}{2}) \log_2 n - \frac{n}{\ln 2} \rceil$ porovnání a očekávaný počet porovnání při rovnoměrném rozdělení vstupních posloupností je alespoň $(n + \frac{1}{2}) \log_2 n - \frac{n}{\ln 2}$.*

Tato věta platí i pro širší třídu primitivních operací, proto v ní lze oslabit předpklady. Dolní odhad (v nejhorším i průměrném případě) bude platit i za předpokladu, že třídicí algoritmus nepoužívá nepřímé adresování a celočíselné dělení. (Na druhé straně následující klasický algoritmus **BUCKETSORT** ukazuje, že předpoklady ve větě nelze zcela vynechat.) Tato metoda pro nalezení dolního odhadu se používá i pro vyčíslování algebraických funkcí a při algoritmickém řešení geometrických úloh.

## 6.2 Přihrádkové třídění

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). Algoritmus **BUCKETSORT** třídí posloupnost přirozených čísel $a_1, a_2, \ldots, a_n$ z intervalu $< 0, m >$.

**BUCKETSORT**$(a_1, a_2, \ldots, a_n, m)$:
**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** $i \leq m$ **do**
  $P$ :=konkatenace $P$ a $Q_i$, $i := i + 1$
**enddo**
**Výstup:** $P$ 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 (tj. třídění 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 $O(n)$. Celkem algoritmus vyžaduje $O(n + m)$ času a paměti. Zřejmě když $m = O(n)$, tak pro tento algoritmus neplatí tvrzení věty z předchozího odstavce. Důvodem 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 navzájem různých 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 $Q_{\lceil ka_i \rceil}$
**enddo**
$i := 0$, $P := \emptyset$
**while** $i \leq k$ **do**
  **HEAPSORT**$(Q_i)$ $P$ :=konkatenace $P$ a $Q_i$, $i := i + 1$
**enddo**
**Výstup:** $P$ 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 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 čas celého třetího cyklu je

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

a celkový čas **HYBRIDSORTU** v nejhorším případě je nejvýše $O(n \log n)$.

Nyní odhadneme očekávaný čas. Položme $X_i = |Q_i|$. Pak $X_i$ je náhodná proměnná a protože pravděpodobnost, že $x \in Q_i$, je $\frac{1}{k}$, dostáváme, že

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

Očekávaný čas vyžadovaný třetím cyklem se pak rovná

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

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 2. momentu binomického rozdělení). □ □

---

Poznámka: V důkazu jsme použili odhad $q \log q \leq q^2$ a důsledkem toho je, že jsme dokázali, že očekávaná složitost **HYBRIDSORTU** zůstane lineární, i kdybychom v něm místo **HEAPSORTU** použili nějaký třídicí algoritmus s kvadratickou složitostí, např. **INSERTIONSORT**.

Nyní použijeme modifikaci **BUCKETSORTU** pro třídění slov. Máme totálně uspořádanou abecedu a chceme lexikograficky setřídit 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 $a_i = a_i^1 a_i^2 \ldots a_i^{l(i)}$, 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**
  $a_i$ vlož do $L_{l(i)}$
**enddo**
Komentář: Pro každé $i$ obsahuje $L_i$ všechna slova z množiny $\{a_1, a_2, \ldots, a_n\}$ délky $i$.
$P := \{(j, a_i^j) \mid 1 \leq i \leq n,\ 1 \leq j \leq l(i)\}$
$P_1$ :=**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**
  $(i, x)$ vlož do $S_i$
  **while**$(i, x)$ =následník $(i, x)$ v $P_2$ **do**
    $(i, x)$ :=následník $(i, x)$ v $P_2$
  **enddo**
  $(i, x)$ :=následník $(i, x)$ v $P_2$
**enddo**
Komentář: V $S_i$ jsou všechny dvojice $(i, x)$ takové, že $x$ je $i$-tým písmenem některého vstupního slova 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**
  $T :=$ konkatenace $L_i$ a $T$, $a$ :=první slovo v $T$
  **while** $a \neq NIL$ **do**
    $s := i$-té písmeno $a$, vlož $a$ do $T_s$
    $a$ :=následník $a$ v $T$
  **enddo**
  $(i, x)$ :=první prvek v $S_i$, $T := \emptyset$
  **while** $(i, x) \neq NIL$ **do**
    $T :=$ konkatenace $T$ a $T_x$, $T_x := \emptyset$
    $(i, x)$ :=následník $(i, x)$ v $S_i$
  **enddo**
  $i := i - 1$
**enddo**
**Výstup:** $T$ je setříděná posloupnost slov $a_1, a_2, \ldots, a_n$

Uvažujme jeden běh posledního cyklu algortimu pro určité $i$. 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 $a_r$ je před $a_q$ 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 plyne z vlastností algoritmu **BUCKETSORT** indukcí podle $i$. Jediný a hlavní rozdíl proti **BUCKETSORTU** je, že neprocházíme všechny přihrádky $T_x$, ale pouze neprázdné. To nám zajišťuje množina $S_i$ (viz Komentář).

Označme $L = \sum_{i=1}^{n} l(i)$ a připomeňme, že $l = \max\{l(i) \mid i = 1, 2, \ldots, n\}$. Pak první cyklus (výpočet délek slov) vyžaduje čas $O(L)$. Druhý cyklus (inicializace seznamů $L_i$) vyžaduje čas $O(l) = O(L)$ a třetí cyklus (zařazení slov do $L_i$ podle délek) čas $O(n) = O(L)$. Vytvoření seznamu $P$ vyžaduje čas $O(L)$ a jeho setřídění podle obou komponent čas $O(L + l) = O(L)$, protože $P$ i $P_1$ 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 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 $a_1, a_2, \ldots, a_n$.

## 6.3 Pořádkové statistiky

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**. Nejprve zadáme přesné znění naší úlohy (úloha i algoritmy se dají snadno přeformulovat pro případ, kdy hledáme $k-$tý největší prvek).

Pracujeme s totálně uspořádaným univerzem $U$.
Vstup: množina prvků $M = \{a_1, a_2, \ldots, a_n\} \subseteq U$ 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*.

**FIND**$(M = (a_1, a_2, \ldots, a_n), i)$:
zvol $a \in M$
$M_1 := \{b \in M \mid b < a\}$, $M_2 := \{b \in M \mid b > a\}$
**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**
    **Výstup:** $a$ je hledaný prvek
  **endif**
**endif**

Důkaz korektnosti algoritmu je založen na následujícím jednoduchém pozorování: mějme množinu $M$ a prvek $x$ a položme $M_1 = \{m \in M \mid m < x\}$. Když $k \leq |M_1|$, pak $k$-tý nejmenší prvek v $M_1$ je stejný jako $k$-tý nejmenší prvek v $M$. Když $k > |M_1|$, pak $(k - |M_1|)$-tý nejmenší prvek v $M \setminus M_1$ je $k$-tý nejmenší prvek v $M$. Zbývá vyšetřit složitost.

V nejhorším případě voláme **FIND** $n$-krát a jedno volání vyžaduje čas $O(|M|)$. Tedy časová složitost algoritmu **FIND** v nejhorším případě je $O(n^2)$. Dobré volby prvku $a$ mohou algoritmus značně zrychlit. V tomto případě platí stejná diskuse jako pro **QUICKSORT**. Spočítáme očekávaný čas za předpokladu, že 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

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

protože procedura **FIND** bez rekurzivního volání sebe sama 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

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

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

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

Protože tento odhad platí také pro $n = 1$ a $n = 2$, dokázali jsme indukcí, že $T(n,i) \leq 4n$ pro všechna $n$ a všechna $i$ taková, že $1 \leq i \leq n$. Platí tedy

**Věta.** *Algoritmus* **FIND** *nalezne $i$-tý nejmenší prvek v $n$ prvkové totálně uspořádané množině a v nejhorším případě vyžaduje čas $O(n^2)$. 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 $i$.

Následující algoritmus nalezne $i$-tý nejmenší prvek v lineárním čase i v nejhorším případě. Vstupem je opět 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**
  setřiď množinu $M$, $m :=$ $i$-tý nejmenší prvek $M$
**else**
  rozděl $M$ do 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**
    najdi medián $m_j$ množiny $A_j$
  **enddo**
  $\bar{m} :=$**SELECT**$(\{m_j \mid j = 1, 2, \ldots, \lceil \frac{n}{5} \rceil\}, \lceil \frac{n}{10} \rceil)$
  $M_1 := \{m \in M \mid m < \bar{m}\}$, $M_2 := \{m \in M \mid \bar{m} < m\}$
  **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** Důkaz korektnosti algoritmu je stejný jako u algoritmu **FIND**. Zbývá vyšetřit složitost. Nejprve dokážeme následující lemma.

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

*Důkaz.* Pro $j \leq \lfloor \frac{n}{5} \rfloor$ platí, že 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, že $|M_1|, |M_2| \geq \lfloor \frac{3n}{10} \rfloor - 1$. Dále platí $M_1 \cap M_2 = \emptyset$, $M_1 \cup M_2 = M \setminus \{\bar{m}\}$ 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 $O(|M|)$, platí, že $T(n) \leq T(\frac{21n}{100}) + T(\frac{8n}{11}) + bn$ pro nějakou konstantu $b$. Zvolme $c \geq \max\{a, \frac{1100b}{69}\}$. Ukážeme, že $T(n) \leq cn$ pro všechna $n$. Když $n \leq 100$, tak tvrzení zřejmě platí, protože $a \leq c$. Když $n > 100$, pak $\lceil \frac{21n}{100} \rceil, \lceil \frac{8n}{11} \rceil < n$, a protože z volby $c$ plyne $b \leq \frac{69}{1100}c$, dostáváme

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

Tedy

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

Algoritmus **FIND** je ve velké většině případů rychlejší než algoritmus **SELECT**, proto je v praxi doporučován, i když existují případy (velmi řídké), kdy potřebuje kvadratický čas. Je známo, že medián $n$-prvkové množiny lze nalézt 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.

### 6.3.1 Historický přehled

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 **BUCKETSORTU** (nazvaná **WORDSORT**) pochází od Aho, Hopcrofta a Ullmana (1974). Algoritmus **SELECT** byl navržen Blumem, Floydem, Prattem, Rivestem a Tarjanem (1972).