Datové struktury - haldy a třídicí algoritmy

I. Úvod

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.

Upravený slovníkový problém:

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:

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.

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:

Další 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.

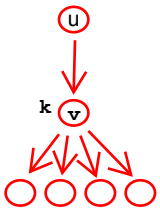
II. Regulární haldy

První použité haldy byly binární neboli 2-regulární haldy. Tyto haldy jsou velmi oblíbené pro svou jednoduchost a názornost a pro velmi efektivní implementaci.

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.



Tvrzení. *Každý d-regulární strom má nejvýše jeden vrchol, který není list a má méně než d synů. Když d-regulární strom má n vrcholů, pak jeho výška je $\lceil \log_d(n(d-1)+1) \rceil$. 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. První část tvrzení plyne přímo z požadavku 2) na d-regulární strom. 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

$$\frac{d^k - 1}{d-1} < n \leq \frac{d^{k+1}-1}{d-1}, \quad d^k - 1 < n(d-1) \leq d^{k+1} - 1$$

a zlogaritmováním dostaneme

$$k < \log_d(n(d-1)+1) \leq k+1.$$

Odtud plyne druhá část tvrzení. Třetí část pro čísla synů 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 $1 + \lfloor \frac{i-2}{d} \rfloor = k$. □

Číslování uzlů ve 2-reg. haldě:

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.

Def:

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.

Přechod od d-reg. str. k d-reg. haldě:

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

Zjednodušené značení:

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.

Na to máme např. leftis haldy

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 & DOWN:

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 $f(v) < f(\text{otec}(v))$ **do**
  vyměň key(v) a key$(\text{otec}(v))$ [vyměň uzly v a otec(v), tj. i hodnoty funkce f]
  $v :=$ 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 [Toto je ve shode s predpokladem, ze dostaneme pro prvek na vstupu jeste informaci, jak prvek najit]
$w :=$poslední list
$t :=$ key(w), key$(v) := t$, odstraň w [Prohodime nas prvek s poslednim listem a pak UP nebo DOWN]
if $f(t) < f(s)$ **then** **UP**(v) **else** **DOWN**(v) **endif** [Neporovnavam s otcem prvku s!]

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

[libovolne umisti prvky S do stromu T (viz str. 38 (resp. 42) v Tarjan - DS and Network algorithms)]

Korektnost algoritmů:

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):

Duální podmínka k (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.

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 $O(d \log_d |S|)$.

Složitost MAKEHEAP:

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**. 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

Vzdálenost od kořene

Vzdálenost od nejhlubšího listu

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

$$dA - A = \sum_{i=0}^{k-1} d^{i+2}(k-i) - \sum_{i=0}^{k-1} d^{i+1}(k-i) = \sum_{i=2}^{k+1} d^i(k-i+2) - \sum_{i=1}^{k} d^i(k-i+1) =$$

$$d^{k+1} + \sum_{i=2}^{k} d^i(k-i+2-k+i-1) - dk = d^{k+1} + \sum_{i=2}^{k} d^i - dk =$$

$$d^{k+1} + d^2\frac{d^{k-1}-1}{d-1} - dk.$$

[Vydělím (d-1), protože dA - A = (d-1)A]

Zanedbáme

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 $O(d^2|S|)$.

Aplikace

Příklad #1: 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: $y_1, y_2, \ldots, y_n$

Vhodné hodnoty d: 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.

Příklad #2: 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 $z \in X$. [funkce c: R -> R+; cena cesty]
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:
$d(z) := 0$, $U := \{z\}$ [d = distance; U je nějaká HALDA]
for every $x \in X \setminus \{z\}$ **do** $d(x) := +\infty$ **enddo** [Zatím je pro nás každý vrchol kromě kořene nedosažitelný]

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** [Zkoušíme vylepšit odhad nejkratší vzdálenosti do vrcholu v]
    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 Dijkstrova algoritmu:

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 $O(|R|)$).

III. Leftist haldy

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.

Definice hodnoty *npl*:

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 npl(v) délku nejkratší cesty z v do vrcholu, který má nejvýše jednoho syna, takže např. pro list l platí npl(l) $= 0$.

Definice Leftist haldy:

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

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

$$\text{npl}(\text{pravý syn } v) \leq \text{npl}(\text{levý syn } v),$$

(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:

Struktura vrcholu:

S vrcholem v jsou spojeny ukazatelé otec(v), levy(v) a 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
npl(v) – proměnná s hodnotou npl(v),
key(v) – prvek reprezentovaný vrcholem v,
$f(v)$ – proměnná obsahující hodnotu f(key(v)).

Uvedeme základní vlastnost leftist haldy, která umožňuje efektivní implementace operací.

Def:

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(velikost podstromu určeného vrcholem v).

Algoritmy a složitost operací

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 konec **endif**
if $T_2 = \emptyset$ **then Výstup**= T_1 konec **endif**
if key(kořen T_1) > key(kořen T_2) **then**
  zaměň T_1 a T_2
endif
[Výsledná halda musí splňovat podmínku (usp), bez této záměny by podmínka nemusela být splněná!!]
T' :=**MERGE**(podstrom pravého syna kořene T_1, T_2)
pravy(kořen T_1) := kořen T'
otec(kořen T') := kořen T_1
Připojím strom T'
if npl(pravy(kořen T_1)) > npl(levy(kořen T_1)) **then**
  vyměň levého a pravého syna kořene T_1
endif
Udržujeme platnou definici Leftist haldy
npl(kořen T_1) := npl(pravy(kořen T_1))+ 1
Opravím si hodnotu npl

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

Vytvořím jednoprvkové haldy

Postupně spojuji tyto haldy až mi zůstane jediná

Časové složitosti: 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 **DELETE-MIN** je v nejhorším případě $O(\log(|S|))$. Operace **MIN** vyžaduje čas $O(1)$. Pro odhad složitosti **MAKEHEAP** budeme uvažovat, že na začátku algoritmu je na vrcholu fronty speciální znak, který se jen přenese na konec fronty. Odhadneme čas, který 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

Složitost MAKEHEAP:

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

Viz pozn. níže

Pozn: 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.

Implementace 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.

Operace OPRAV: 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$(t) := 0$ [Zde opravuji strom T (konkrétně otce t uzlu v), ze kterého škubu v.]
if pravy$(t) \neq v$ **then** levy$(t) :=$ pravy(t) **endif**
pravy$(t) := NIL$
while se zmenšilo npl(t) a t není kořen **do**
  $t :=$ otec(t)
  if npl(pravy(t)) > npl(levy(t)) **then**
    vyměň levy(t) a pravy(t)
  endif
  npl$(t) :=$ npl(pravy(t)) + 1
enddo

Uzel t nesmí mít pouze pravého syna

Směrem ke kořeni opravuji hodnoty npl

Výstup: t

Korektnost procedury OPRAV:

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
T_1:=podstrom T určený vrcholem v, $f(v) := f(v) - a$
T_2 :=**Oprav**(T, v), T :=**MERGE**(T_1, T_2)

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.*

IV. Amortizovaná složitost

Popíšeme bankovní paradigma pro počítání s amortizovanou složitostí. Předpokládejme, že máme funkci h, která ohodnucuje 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:

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

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.

V. Binomiální haldy

Další typ hald je motivován sčítaním 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.

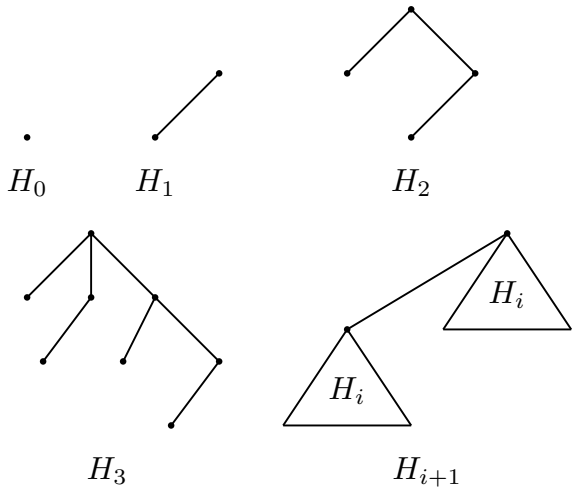
Def:

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 Obr. 1.

Nejprve uvedeme základní vlastnosti těchto stromů.

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

(1) *strom H_i má 2^i vrcholů,*
(2) *kořen stromu H_i má i synů,*
(3) *délka nejdelší cesty z kořene do listu ve stromu H_i je i (tj. výška H_i je i),*
(4) *podstromy určené syny kořene stromu H_i jsou izomorfní se stromy $H_0, H_1, \ldots, H_{i-1}$.*



Obr. 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. □

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

- 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;
- každý strom T_i je izomorfní s nějakým stromem H_j;
- T_i není izomorfní s žádným T_j pro $i \neq j$.

Vztah bin. čísel a binom. hald:

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.

Algoritmy a složitost operací

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:$\mathcal{H}$

Existuje v první haldě strom izomorfní s H_i? (Tzn. stačí projít stromy T_i haldy a hledat index i)

[Našli jsme H_i pouze v jedné z hald a nezůstalo nám nic z předchozího spojování]

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
Výstup: Kořen T_1

Korektnost MERGE:

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. Protože každý běh cyklu vyžaduje čas $O(1)$, algoritmus **MERGE** vyžaduje čas $O(\log(|S_1| + |S_2|))$.
Implementace dalších algoritmů je podobná jako pro leftist haldy.

INSERT(x):
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$(\mathcal{H}_1, \mathcal{H}_2)$

Korektnost MIN operace:

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 INCREASE a DECREASE:

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, ∞) a **DELETEMIN**. Operace **MAKEHEAP** se provádí opakováním operace **INSERT**.

Časová slož. MERGE:

Výpočet časové složitosti operací pro binomiální haldy využívá několik známých faktů. Operace **MERGE** simuluje sčítání přirozených čísel v binárním zápise a má tedy stejnou složitost. Odhad složitosti vytváření haldy využívá známého faktu, že amortizovaná složitost přičítání 1 k binárnímu číslu je $O(1)$.

Časová slož. MIN a DELMIN:

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 $\log(|S|)$.

Časová slož. DECREASE a INCREASE:

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. Můžeme tedy shrnout:

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* $O(|S|)$.

Neefektivnost Binom. hald:

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.

Líná implentace operací

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.

Def:

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

- 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;
- každý strom T_i je izomorfní s nějakým stromem H_j.

Stejné podmínky jako u binom. hald, pouze jedna podmínka byla odebrána.
Zmizela podminka ohledne vylouceni dvou izomorfnich stromu v H.

Pozn. k definici:

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 $\mathcal{H}_2$

INSERT:

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** [For cyklus]
  while $|O_i| > 1$ **do**
    vezmi dva různé stromy T_1 a T_2 z O_i
#1 odstraň je z O_i
    spoj(T_1, T_2) vlož do O_{i+1}
  enddo
  if $O_i \neq \emptyset$ **then**
#2 strom $T \in O_i$ odstraň z O_i a vlož do $\mathcal{H}$
  endif,
  $i := i + 1$
enddo
Výstup: $\mathcal{H}$

Spojujeme stromy v *O_i* a tím z nich efektivně vytváříme stromy pro seznam *O_{i+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 $\mathcal{H}$ izomorfní s $H_i\}$
vyvaz$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

MIN operace, jak jsme ji definovali poprvé.

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$(\{O_i \mid i = 0, 1, \ldots, \lfloor \log(|S|) \rfloor\})$

Časové složitosti:

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. Protože operace **MERGE** nemění počet stromů a protože operace **INSERT** přidá jen jeden strom, je

amortizovaná složitost operací **MERGE** a **INSERT** stále $O(1)$. Ukážeme, že amortizovaná složitost operací **MIN** a **DELETEMIN** při líné implementaci binomiálních hald je $O(\log(|S|)$. Protože každý běh vnitřního **while**-cyklu v operaci **vyvaz** vyžaduje čas $O(1)$ a zmenší počet stromů v seznamech O_i o 1, operace **vyvaz** vyžaduje čas $O(k + \sum_{i=0}^{k} |O_i|) = O(k+|\mathcal{H}|)$. 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. Podle tvrzení je $i \leq \log(|S|)$, a tedy operace **MIN** vyžaduje čas $O(|\mathcal{H}|)$ a operace **DELETEMIN** čas $O(|\mathcal{H}| + \log(|S|))$. Protože ohodnocení klasické binomiální haldy je nejvýše $\log(|S|)$ (obsahuje tolik stromů, kolik je 1 v binárním zápise čísla $|S|$), dostáváme, že amortizovaná složitost operace **MIN** je $O(|\mathcal{H}| - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$ a amortizovaná složitost operace **DELETEMIN** je $O(|\mathcal{H}| + \log(|S|) - |\mathcal{H}| + \log(|S|)) = O(\log(|S|))$.

Amortizovaná složitost MIN a DELMIN:

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.

VI. Fibonacciho haldy

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.

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 struktury vzniklé z prázdné haldy aplikací posloupnosti těchto algoritmů.

Algoritmy

Označování vrcholů:

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.

Def:

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$(\mathcal{H}_1, \mathcal{H}_2)$:

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 [Noví synové se ukládají na konec seznamu synů]
$v :=$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** [Zrušíme označení, protože v je nyní kořen T']
  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**

[Stoupáme po označených vrcholech a odoznačujeme a odtrháváme podstromy a vkládáme je do haldy.]

Vysvětlení operace vyvaz2 viz poslední řádka na této stránce.

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 ~~vyvaz1~~ 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.

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.

Ohodnocení konfigurace pro amort. složitost:

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. (matematicky: #stromu + 2*#pocet_oznacenych_vrcholu :-))

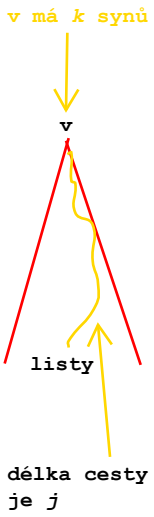
DEF:

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 $O(\rho(n))$.

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. □

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 $|\frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^i| < \frac{3}{8}$ pro všechna $i = 1, 2, \ldots$, a tedy

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

Odtud plyne, že když i splňuje

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

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

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

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

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

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

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.

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.

Kdy použít Fib. a kdy d-reg. haldu:

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.

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.

VII. Třídicí algoritmy

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:

Def:

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**.

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.

Požadavky na třídění:

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.

Duální podmínka uspořádání (chceme MAXHEAP):

Pozn.:

HEAPSORTU je stále věnována velká pozornost a bylo navrženo několik jeho modifikací, snažících se např. minimalizovat počet porovnání prvků apod.

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

Hledám vždy setříděný úsek posloupnosti a_i

Čas: O(n)

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

Slívám po dvou posloupnostech z Q dokud nejsem slitý na jedinou setříděnou PSP

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, $i := 1$, $j := 1$, $k := 1$

Sliji dvě neklesající PSP do PSP c_k:

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

Stacilo dat jednou za endif.

while $i \leq n$ **do**
  $c_k := a_i$, $i := i + 1$, $k := k + 1$
enddo

Posloupnost a_i může být delší než PSP b_i

while $j \leq m$ **do**
  $c_k := b_j$, $j := j + 1$, $k := k + 1$
enddo

Posloupnost b_i může být delší než PSP a_i

Výstup: $P = (c_1, c_2, \ldots, c_{n+m})$

Korektnost:

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.

Složitost procedury MERGE:

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.

Složitost procedury MERGESORT:

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 ≥ 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.

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** [Indexy i a j máme ze vstupu algoritmu]
  Výstup: (a_i)
else
  zvol k takové, že $i \leq k \leq j$, $a := a_k$, vyměň a_i a a_k, $l := i+1$, $q := j$
(#3)**while** true **do** pivot
 (#1)**while** $a_l < a$ **do** $l := l + 1$ **enddo** [Hledám prvky menší než pivot od začátku PSP a_i]
 (#2)**while** $a_q > a$ **do** $q := q - 1$ **enddo** [Hledám prvky větší než pivot od konce PSP a_i]
    if $l \geq q$ **then**
      exit [break while cyclus]
    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

```
Příklad:
================================
1 2 6 4 5 7 3 8 9
i       k       j

PSP je upravena před #1 takto:

5 2 6 4 1 7 3 8 9
i 1             q=j

Po #2 PSP vypadají indexy takto:

5 2 6 4 1 7 3 8 9
i   l       q   j

Po skončení while cyklu #3:

5 2 3 4 1 7 6 8 9
i       q l     j
```

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

Korektnost algoritmu:

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.

Původní (historicky) výběr pivota:

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

Náhodný pivot:

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).

Nejlepší výběr pivota:

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 $O(n \log n)$.

Předpoklady následující analýzy:

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~~ **výpočtu** 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).

Náhodná proměnná X_i,j:

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

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{i,j} \qquad \text{[Počet všech porovnání]}$$

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

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbf{E}(X_{i,j}) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j}. \qquad \text{[Očekávaný počet všech porovnání]}$$

Výpočet p_{i,j}:

Def strom výpočtu:

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 $\{a_l \mid i \le l \le j\} = \{b_l \mid i \le l \le j\}$.

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 $\{a_i \mid 1 \le i \le n\}$ v pořadí daném tímto očíslováním. Pak platí, že $X_{i,j} = 1$, právě když první prvek v posloupnosti $(c_1, c_2, \ldots, c_n)$ z množiny $\{b_l \mid i \le l \le j\}$ 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 $1 \le i < j \le n$. Odtud očekávaný počet porovnání v algoritmu **QUICKSORT** je

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{i,j} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n} \sum_{k=2}^{n-i+1} \frac{2}{k} \le 2n\left(\sum_{k=2}^{n} \frac{1}{k}\right) \le 2n \int_1^n \frac{1}{x} dx = 2n \ln n.$$

Druhý způsob výpočtu:
Označme $QS(n)$ očekávaný počet porovnání provedených algoritmem **QUICKSORT** při třídění n-členné posloupnosti. Pak platí

$$QS(0) = QS(1) = 0 \text{ a}$$

$$QS(n) = \frac{1}{n}\Big(\sum_{k=0}^{n-1} n - 1 + QS(k) + QS(n-k-1)\Big) = n - 1 + \frac{2}{n}\Big(\sum_{k=0}^{n-1} QS(k)\Big).$$

Všechny možné volby pivota. S pivotem vždy porovnám všechny zbývající prvky kromě sebe (tedy n-1 prvků) a musím započítat počet testů, které se provedou na "levé" a na "pravé" PSP (původní PSP je rozdělena přes pivota na "levou" a "pravou").

Z toho dostáváme, že

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

[Vynásobili jsme obě strany rovnice proměnnou n]

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

$$QS(n) = \sum_{i=2}^{n} \frac{n+1}{i+1}\,\underbrace{\frac{2(i-1)}{i}}_{\leq 1} \leq 2(n+1)\Big(\sum_{i=2}^{n}\frac{1}{i+1}\Big) \leq 2(n+1)\Big(\sum_{i=3}^{n+1}\frac{1}{i}\Big) =$$

$$2(n+1)\Big(\sum_{i=2}^{n+1}\frac{1}{i} - \frac{1}{2}\Big) \leq 2(n+1)\Big(\Big(\underbrace{\int_{i=1}^{n+1}\frac{1}{x}dx}_{\ln(n+1)}\Big) - \frac{1}{2}\Big)$$

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

Pro dostatečně velká n tedy platí

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

Porovnání třídicích algoritmů

Selection sort:

Nyní porovnáme složitost algoritmů **HEAPSORT**, **MERGESORT**, **QUICKSORT**, **A-sort** (byl popsán v kapitole o $(a,b)-$stromech), **SELECTIONSORT** a **INSERTION-SORT**. 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.

Výsledky pro model RAM:

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 ≤ 22 použít **SELECTIONSORT** a pro posloupnosti délky ≤ 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.

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.

MERGE rost. disj. PSPstí:

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

Strom slučování posloupností:

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:

> 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 $O(\sum\{l(P(v)) \mid v$ 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$(T, \{P(l) \mid l$ 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 ϕ z množiny $\{1, 2, \ldots, n\}$ do listů T

Def optimálního stromu:

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, ϕ) 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.

Def hodnoty Cont:

Mějme množinu $\{x_i \mid i = 1, 2, \ldots, n\}$. Pro úplný binární strom T s n listy a bijekci ϕ 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 [Jednotlivé listy se pak ohodnotí hodnotami x_i]
ϕ 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, ϕ), kde T je strom v množině V

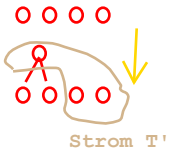
Spojuju stromy s nejmenšími hodnotami.

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 čas $O(n)$.*

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, ϕ je bijekce z $\{1, 2, \ldots, n\}$ do množiny listů T a algoritmus vždy končí. Dokážeme indukcí podle n, že zkonstruovaná dvojice (T, ϕ) je optimální strom vzhledem k $(x_1, x_2, \ldots, x_n)$.



Jen info, co nám dává indukční předpoklad:

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 ψ 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', ψ), a podle indukčního předpokladu je to optimální strom pro $(y_1, y_2, \ldots, y_{n-1})$. Nechť (U, θ) 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 η 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 η 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, θ) je optimální strom pro $(x_1, x_2, \ldots, x_n)$, dostáváme, že (U, η) 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 τ 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 τ je bijekce z $\{1, 2, \ldots, n-1\}$ do množiny listů U' a protože (T', ψ) je optimální strom pro $(y_1, y_2, \ldots, y_{n-1})$, platí, že

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

Protože

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

pak závěr je, že (T, ϕ) je optimální strom pro $(x_1, x_2, \ldots, x_n)$.

Druhá část důkazu (čas. slož.)

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 $l(P_1), l(P_2), \ldots, l(P_n)$ spojí do jediné rostoucí posloupnosti v čase $O(\sum_{i=1}^{n} l(P_i))$.*

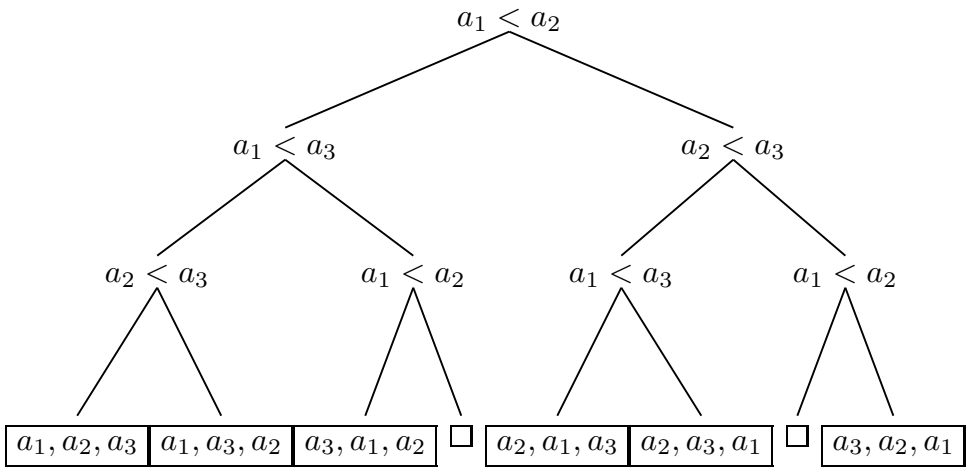
VIII. Rozhodovací stromy

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 π množiny $\{1, 2, \ldots, n\}$. Tato permutace prochází stromem takto:

> 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 π, reprezentuje počet porovnání, které potřebuje daný algoritmus k setřídění dané posloupnosti π. 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.



Obr. 1

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.

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 π n-prvkové množiny platí

> posloupnost porovnání při třídění permutace π algoritmem **A** je stejná jako posloupnost porovnání při průchodu permutace π stromem T.

Korektnost algo. a dolní odhad času:

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.

Odhad S(n)

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:

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

Protože pro $n \geq 1$ je $\frac{1}{12n}, \frac{1}{n^2} \geq 0$, můžeme předpokládat, že $(1 + \frac{1}{12n} + O(\frac{1}{n^2})) \geq 1$ 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

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

Odhad A(n)

Dále pro binární strom T označme $B(T)$ součet všech délek cest z kořene do listů a položme

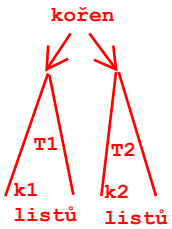
$$B(k) = \min\{B(T) \mid T \text{ je binární strom s } k \text{ listy}\}.$$

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

Důkaz indukcí:



$$B(T) = k_1 + B(T_1) + k_2 + B(T_2) \geq k + B(k_1) + B(k_2) \geq k + k_1\log_2 k_1 + k_2 \log_2 k_2.$$

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.

IX. Přihrádkové třídění

Vlastnosti struktury:

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.

Varianty Bucketsortu:

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 α 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, $i := i + 1$ [Skutečně HEAPSORT!]
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* $O(n)$.

Nejhorší případ:

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 [while cyklu]

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

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

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

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**.

Druhá modifikace BUCKETSORTU:

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 $a = x_1 x_2 \ldots x_n$ a $b = y_1 y_2 \ldots y_m$ jsou dvě slova nad totálně uspořádanou abecedou Σ, 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** [Spočítám si délky slov]
$l = \max\{l(i) \mid i = 1, 2, \ldots, n\}$ [Spočítám si nejdelší délku slova]
for every $i = 1, 2, \ldots, l$ **do** $L_i = \emptyset$ **enddo** [Inicializuji si seznamy pro slova dané délky]
for every $i = 1, 2, \ldots, n$ **do**
  a_i vlož do $L_{l(i)}$ [Podle délky vkládám slova do správných seznamů]
enddo
~~Komentář: Pro každé i obsahuje L_i všechna slova z množiny $\{a_1, a_2, \ldots, a_n\}$ délky i.~~
$P := \{(j, a_i^j) \mid 1 \le i \le n, 1 \le j \le l(i)\}$ [Seznam dvojic (i-tý znak slova, znak)]
P_1 :=**BUCKETSORT**(P) podle druhé komponenty [Mám setříženo podle písmenek tedy]
P_2 :=**BUCKETSORT**(P_1) podle první komponenty [Setříděno podle 1. a 2. 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** [Porovnávají se obě složky dvojice, nechceme vkládat vícekrát prvek (i,x)]
    (i, x) :=následník (i, x) v P_2
  enddo
  (i, x) :=následník (i, x) v P_2 [Tento řádek je tu navíc??? Ano!!]
enddo

Do S_i si uložím jaká různá písmenka se vyskytují jako i-tý znak ve všech slovech

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** [For cyklus od i:= l (el) downto 1 (jedna)]
  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, $T := \emptyset$ [Tady se maže T!]
  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 $a_1, a_2, \ldots, a_n$

Projdu všechna slova v T a rozhodím je do kbelíků podle i-tého písmenka.

```
Příklad
======================
a1 = angl (začátek slova anglie :))
a2 = ano

L_4 = {angl}, L_3 = {ano}

P = {(1,a),(2,n),(3,g),(4,l),
     (1,a),(2,n),(3,o)}
P1= {(1,a),(1,a),(4,l)
     (2,n),(2,n},
     (3,g),(3,o)}
P2= {(1,a),(1,a),
     (2,n),(2,n},
     (3,g),(3,o),(4,l)}

S1 = {(1,a)}
S2 = {(2,n)}
S3 = {(3,g),(3,o)}

1. běh while cyklu
T = {angl}
T_l = {angl}
```

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 ~~$j = i, i+1, \ldots, l$~~ 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).

Časová složitost: 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 $a_1, a_2, \ldots, a_n$.

X. 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 $|\{j \mid 1 \leq j \leq n, a_j < a_k\}| = i$.
Když $i = \frac{n}{2}$, pak a_k se nazývá medián.

FIND$(M = (a_1, a_2, \ldots, a_n), i)$:

```
zvol a ∈ M
M1 := {b ∈ M | b < a}, M2 := {b ∈ M | b > a}
if |M1| > i − 1 then
  FIND(M1, i)
else
  if |M1| < i − 1 then
    FIND(M2, i − |M1| − 1)
  else
    Výstup: a je hledaný prvek
  endif
endif
```

Korektnost: 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.

Časová 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

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

protože procedura **FIND** bez rekurzivního volání 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

Indukční předpoklad

$$T(n,i) = n + \frac{1}{n}\left(\sum_{k=1}^{i-1} T(n-k, i-k) + \sum_{k=i+1}^{n} T(k,i)\right) \leq n + \frac{1}{n}\left(\sum_{k=1}^{i-1} 4(n-k) + \sum_{k=i+1}^{n} 4k\right) =$$

$$n + \frac{4}{n}\left(\frac{(2n-i)(i-1)}{2} + \frac{(n+i+1)(n-i)}{2}\right) = n + \frac{4}{n}\left(\frac{n^2 + 2ni - n - 2i^2}{2}\right).$$

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

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

Protože tento odhad platí také pro $n = 1$ a $n = 2$, 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 $O(n)$.*

Neefektivní přirozený algoritmus:

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.

Druhý algo.:

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.*

Doporučení: 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.

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).