UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

VIZUALIZÁCIA HÁLD A INTERVALOVÝCH STROMOV

Bakalárska práca

2012

Katarína Kotrlová

UNIVERZITA KOMENSKÉHO V BRATISLAVE FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

Vizualizácia háld a intervalových stromov

Bakalárska práca

Študijný program: Informatika
Študijný odbor: 2508 Informatika
Školiace pracovisko: Katedra Informatiky FMFI
Školiteľ: Mgr. Jakub Kováč

Bratislava, 2012 **Katarína Kotrlová**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Katarína Kotrlová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: 9.2.1. informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Vizualizácia háld a intervalových stromov

Cieľ: Cieľom práce je vizualizovať rôzne dátové štruktúry, konkrétne rôzne druhy prioritných front (ako je ľavicová halda, skew halda, či párovacia halda) a intervalových stromov (fínske stromy).

Kľúčové slová: vizualizácia, dátové štruktúry

Vedúci: Mgr. Jakub Kováč
Katedra: FMFI.KI - Katedra informatiky

Dátum zadania: 27.09.2011

Dátum schválenia: 04.10.2011

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Poďakovanie patrí môjmu školiteľovi za ochotu a poctivé vedenie pri práci.

Čestné prehlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracovala samostatne s použitím citovaných zdrojov.

. . . . . .

Abstrakt

V práci postupne rozoberáme niekoľko vybraných dátových štruktúr, skúmame ich zložitosti a možné vylepšenia. Konkrétnejšie sa zaoberáme druhmi prioritných front (d-árna halda, ľavicová halda, skew halda, párovacia halda) a intervalovými stromami. Neskôr sa zameriavame na vizualizáciu a samotný vizualizačný program, ktorý je navrhnutý ako edukačný prostriedok. Práca nadväzuje na bakalársku prácu s Java appletom Jakuba Kováča z roku 2007 na tejto univerzite. Ako prílohu prikladáme CD so zdrojovými súbormi a spustiteľnou aplikáciou.

Kľúčové slová: dátové štruktúry, vizualizácia, intervalové stromy, zlúčiteľné prioritné fronty

Abstract

In this thesis we describe several data structures, examine their complexity and possible improvements. In particular we explore various kinds of priority queues (d-ary heap, leftist heap, skew heap, pairing heap) and interval trees. Then, we focus on their visualization and the visualization program itself, which is devices as an educational tool. This work is a continuation of the bachelor's thesis and Java applet of Jakub Kováč from the year 2007 in this university. In an appendix we provide a CD with source codes and an executable application.

Keywords: data structures, visualization, interval trees, meldable priority queues

Obsah

Úvod

Motiváciou pre spracovanie témy zaoberajúcej sa algoritmami a dátovými štruktúrami je potreba ušetriť čas. Aby sme vedeli narábať s veľkým množstvom dát efektívne, potrebujeme pre ne vytvoriť niečo, čo sa o to postará. Abstraktnými dátovými štruktúrami označujeme abstraktnú realizáciu ukladania a pracovania s dátami. Ideou je čo najviac na nich zefektívniť operácie, čo sa týka pamäťovej a časovej zložitosti.

Existuje množstvo dátových štruktúr. To, prečo ich je tak veľa zrejme zodpovedá na otázku množstva rôznych požiadaviek na typy operácií, druh dát, uprednostňovanie času pred pamäťou a naopak. Jednotlivé typy sa líšia v usporiadaní dát a v operáciách, ktoré môžeme na nich vykonávať. Obsahujú štandardné algoritmy pre dopyty ako *vytvor dátovú štruktúru*, *vlož prvok*, *nájdi najmenší prvok*, *vymaž najmenší prvok* a niektoré nami popisované štruktúry aj algoritmus pre *spoj*. Pre tieto operácie rozoberáme zložitosti a uvádzame modifikácie pre efektívnejšie prevedenie, ktoré sa snažia zložitosti zlepšovať, avšak často na úkor jednoduchosti implementácie. Väčšina z nich má štruktúru *stromu*. V niektorých prípadoch sa snažíme minimalizovať počet úkonov pri vykonaní operácie zvolením vhodného smeru - zhora nadol, zdola nahor (*skew halda*).

Občas potrebujeme vykonať veľa operácií po sebe. Vtedy sa nepozeráme na najhorší, najlepší či priemerný prípad jednej operácie, ale zaujímavejší je priemer času vykonania postupnosti operácií. Toto voláme *amortizovaná časová zložitosť*. Amortizáciu bližšie popisujeme pri *skew halde* a *intervalových stromoch*.

V súčasnosti je málo programov, ktoré by prinášali komplexnejší prehľad využívaných dátových štruktúr, na niektoré dokonca vizualizácia doposiaľ neexistuje. Je veľa appletov, ktoré implementujú niektoré algoritmy, avšak ich nedostatkom býva neprehľadnosť vizualizácie a hlavne okrem základných operácií dátovej štruktúry takmer žiadna interaktivita.

Výstupný program s názvom Gnarley trees je vytváraný v spolupráci s ďalšími nadšencami (Jakub Kováč, Viktor Tomkovič, Tatiana Tóthová, Pavol Lukča). Myšlienkou je podať čo najjednoduchšie informáciu o rozdieloch medzi dátovými štruktúrami a použitými algoritmami. Má hlavne edukačný charakter, obsahuje textovú časť vysvetľujúcu jednotlivé kroky pri vykonávaní operácií. Takisto existuje stránka zaoberajúca sa tým, čo sme spracovali na portáli `http://people.ksp.sk/~kuko/gnarley-trees` . Jej účelom je stručne popísať operácie, časovú zložitosť a význam každej implementovanej dátovej štruktúry. Spolu s appletom je navyše obohatená o doplňujúce otázky overujúce správne pochopenie problematiky.

Časť I

Teoretický popis

Kapitola 1

Základné pojmy

1.1 Označenia

O čitateľovi sa predpokladá, že má základné znalosti z oblasti dátových štruktúr, najmä stromovitých. Tiež by mal mať aspoň intuitívny pohľad na časové zložitosti a ich označenia.
Definície *stromu*, *zakoreneného stromu*, *binárneho stromu*, *rozšíreného binárneho stromu* a *úplného binárneho stromu* používame podľa definícií uvedených v [4]. Takisto aj vlastnosti stromov a vrcholov ako výšku, hĺbku, stupeň, či definíciu *externých vrcholov* v *rozšírenom binárnom strome*.
Vrcholy budeme obvykle značiť v, w. Pripomenieme, že synov vrcholu v v binárnom strome označujeme $left(v)$ a $right(v)$, rodiča označujeme $p(v)$. Ak máme dátovú štruktúru s vyšším stupňom vrcholov, $left(v)$ je smerník na prvého syna a $right(v)$ je smerním na pravého brata. Naše dáta budú reprezentovať *kľúče*. Kľúč je prirodzené číslo, ktoré má každý vrchol nesúci informáciu o dátach. Ako $key(v)$ označíme kľúč vrchola v, teda hodnotu vo vrchole v.

Les. *Les* je n-tica stromov. V našom prípade budú mať stromy v lese rovnaké definície.

Pravá (resp. ľavá) cesta z vrcholu v**.** Je cesta najpravejšieho (resp. najľavejšieho) vrcholu v podstrome zakorenenom vo vrchole v. Prejdeme ju tak, že sa z vrcholu v budeme hýbať len do jeho pravého (resp. ľavého) syna, pokým bude existovať.

Amortizovaná časová zložitosť. S *amortizovanou časovou zložitosťou* sa stretneme pri *skew halde* a *intervalových stromoch*. Na rozdiel od *asymptotickej zložitosti* dáva presnejší popis dlhodobejšieho chovania systému. Neuvádza najhorší možný prípad

zložitosti operácie, ale priemer zložitostí postupnosti operácií v najhoršom prípade. Toto môže zavážiť napríklad ak postupnosť operácií garantuje, že keď nastane najhorší prípad, dosť dlhú dobu potom nenastane.

1.2 Pseudokód

Pseudokódy sú založené na jazyku C++. Príkazy požívame štandardným spôsobom.
Z programátorskeho hľadiska je strom smerník na jeho koreň, teda ak hovoríme o strome v, tak tým myslíme strom, ktorý je zakorenený vo vrchole v.
Typy premenných, ktoré používame sú obvykle *int*, *Node* - vrchol, ktorého sa môžeme pýtať na jeho synov, rodiča, či kľúč.
Funkcie a procedúry mávajú popisný názov. Z hľadiska zrozumiteľnosti, niekedy namiesto príkazu použijeme krátky popis.
Program 1.1 vykreslí postupne vrchol v s kľúčmi od 1 po k.

Program 1.1 Kreslenie vrcholu v

```
void changeKey(Node v, int k) {
    for (int i = 1; i <= k; i++){
        key(v) = k;
        vykresli(v);
    }
}
```

Kapitola 2

Haldy

Definícia. Halda je vo všeobecnosti *zakorenený strom*, ktorý spĺňa podmienku, ak vrchol $p(v)$ je otcom vrcholu v, potom $key(p(v)) \leq key(v)$[1]. Otec je teda vždy menší ako jeho synovia a v koreni haldy sa nachádza najmenší prvok[2].

Operácie. Každá dátová štruktúra je daná jej definíciou a operáciami. Štandardné operácie, ktoré haldy podporujú a ktorými sa budeme zaoberať, sú:

- **createHeap** – vytvorí prázdnu haldu;
- **insert** (x) – vloží vrchol s kľúčom x;
- **findMin** – vráti minimum t.j. hodnotu kľúča v koreni;
- **deleteMin** – odstráni vrchol s najmenším kľúčom, t.j. koreň;
- **decreaseKey** (v, Δ) – zníži kľúč vrcholu v o delta;

Niektoré haldy navyše implementujú **meld** (i, j) – spojí haldu i s haldou j.

2.1 Binárna halda

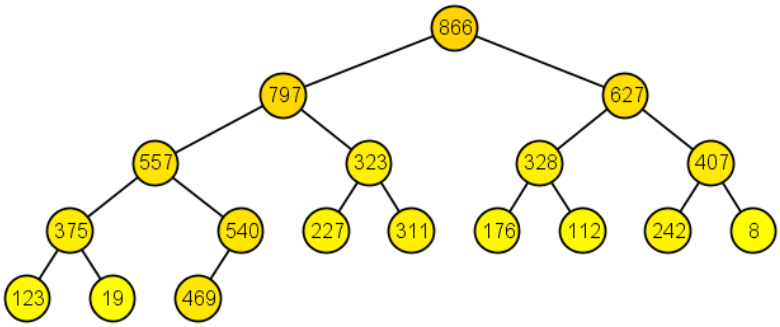
2.1.1 Definícia

Definícia. *Binárna halda* je *úplný binárny strom* spĺňajúci podmienku haldy, ktorého stupeň vrcholov je najviac dva. Najčastejšie sa reprezentuje v poli, koreň je na mieste 0 a synovia i-teho prvku sú v poli na miestach $(2 \cdot i + 1)$ až $(2 \cdot i + 2)$. Pokiaľ koreň

[1] Bez ujmy na všeobecnosti budeme uvažovať o *min haldách*. Podobnými úvahami by sme text mohli rozšíriť o *max haldy* s najväčším prvkom v koreni.

[2] V koreni sa nachádza vždy prvok s najväčšou prioritou. Pri min haldách dochádza mierne k zmätku, keďže vrchol s menším kľúčom má väčšiu prioritu ako vrchol s väčším kľúčom.

obsahuje hodnotu *null* halda je prázdna.



Obr. 2.1: *Binárna halda*

V koreni sa nachádza najmenší prvok. Tiež vieme s istotou povedať, že druhý najmenší prvok sa nachádza v hĺbke najviac dva, tretí v hĺbke najviac tri atď.

Pokiaľ je porušená podmienka haldy potrebujeme ju upraviť. Preto si definujeme ešte procedúru *bubbleup(v)*, ktorá, ak $key(p(v)) > key(v)$, vymieňa vrchol v so svojim otcom, až pokým nie je podmienka haldy opäť splnená.

Program 2.1 Procedúra bubbleup(v)

```
void bubbleup(v) {
    while (key(p(v)) > key(v) && p(v) != null)
        vymeň(p(v),v);
}
```

2.1.2 Operácie

Operácia *insert*(v)**.** Vložíme vrchol v na najbližšie voľné miesto, tak, aby sa neporušila úplnosť stromu. V praxi to znamená, že sa pridá nový prvok na koniec poľa. Takto vložený prvok môže porušovať podmienku haldy, takže ešte musí "prebublať"smerom hore na správne miesto. Využijeme procedúru *bubble(v)*, teda vrchol v sa vymieňa so svojím otcom, až pokým nie je podmienka haldy splnená.

Operácia *deleteMin***.** Minimum sa nachádza v koreni haldy. Operácia deleteMin najprv vymení koreň haldy s posledným vrcholom a potom minimum, ktoré sa teraz nachádza na konci haldy, odstráni. Koreň haldy po výmene nemusí spĺňať podmienku haldy a preto musí "prebublať"nadol. Podobne, ako procedúru *bubbleup(v)* si môžeme definovať procedúru *bubbledown(v)*. Vrchol v sa vymieňa so svojím menším synom, až pokým nie je splnená podmienka haldy.

Program 2.2 Procedúra bubbledown(v)

```
void bubbledown(v) {
    while (left(v) != null && v > min(left(v),right(v))
        vymeň(v, min(left(v),right(v)));
}
```

Operácia *decreaseKey*(v, Δ)**.** Po znížení hodnoty kľúča vrcholu v tento vrchol nemusí spĺňať podmienku haldy a preto musí opäť "prebublať"nahor, k čomu nám poslúži procedúra bubbleup(v).

2.1.3 Časová zložitosť

Procedúra *bubbleup(v)* beží v čase $O(d(\mathrm{v}))$, keďže z definície hĺbky vrcholu v je zrejmé, že v sa môže vymeniť som svojím otcom najviac d(v)-krát, kým sa z neho stane koreň. Procedúra *bubbledown(v)* beží v čase $O(v(\mathrm{v}))$, keďže z definície výšky vrcholu v v strome je zrejmé, že v sa môže vymeniť som svojím synom najviac h(v)-krát, kým sa z neho stane list.

Ak n je počet vrcholov haldy, operácia *insert*(v) má časovú zložitosť $O(log(n))$. Prvok vždy vkladáme na koniec haldy, teda sa z neho stane list, ktorý musí prebublať nahor. Hĺbka listu je výška haldy, teda $lg(n)$.
Operácia *deleteMin* má časovú zložitosť $O(log(n))$. Po výmene prvok vždy prebubláva z koreňa smerom nadol, teda maximálne celú vzdialenosť listov od koreňa, čo je výška koreňa.

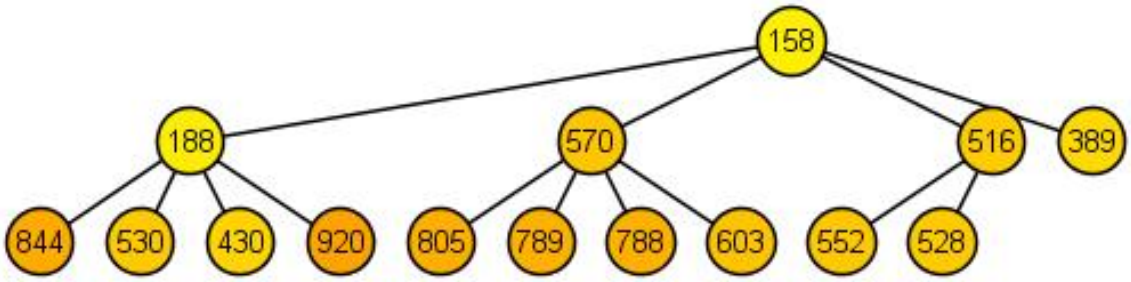
2.2 d-árna halda

2.2.1 Definícia

Definícia. D-árna halda je *úplný d-árny strom*, spĺňajúci podmienku haldy. Môžeme ju považovať za zovšeobecnenie binárnej haldy. Rozdiel je iba v stupni vrcholov, pri d-árnej halde sú vrcholy stupňa d. Čo sa týka uloženia v poli, koreň je na mieste 0 a synovia i-teho prvku sú v poli na miestach $(d \cdot i + 1)$ až $(d \cdot i + d)$.

2.2.2 Operácie

Operácie *insert(x)*, *deleteMin* a *decreaseKey*(v, Δ) sa nijak nelíšia od binárnej haldy. Tak isto ani procedúra *bubbleup(v)*. Mierne sa zmení iba procedúra *bubbledown(v)*. Keďže vrcholy majú stupeň d, pri prebublávaní nadol, aby bola zachovaná podmienka



Obr. 2.2: *Príklad d-árnej haldy pre* $d = 4$

haldy, sa vyberá najmenší z d synov.

Program 2.3 Procedúra bubbledown(v)

```
void bubbledown(v) {
while (v > min(syn[1],...,syn[d]) && v nie je list)
    vymeň(v, min(syn[1],...,syn[d]));
}
```

2.2.3 Časová zložitosť.

Ak n je počet prvkov v halde, potom operácie *insert* a *decreaseKey* majú časovú zložitosť $O(log_d(n))$, pretože $log_d(n)$ je hĺbka stromu, teda vrchol by sa po $log_d(n)$ krokoch dostal ku koreňu. Operácia deleteMin má zložitosť $O(d \cdot log_d(n))$, pretože sa navyše pri prebublávaní musí hľadať najmenší syn spomedzi d synov.

2.3 Ľavicová halda

2.3.1 Definícia.

Definícia. Ľavicová halda je *halda* so stupňom vrcholov nanajvýš dva, pričom pre každý vrchol si pamätáme hodnotu *rank*. *Rank* je najkratšia vzdialenosť vrcholu k *externému vrcholu*. V definícií haldy uvažujeme o *rozšírenom binárnom strome*, ale externé vrcholy nie sú súčasťou haldy. Ich rank je 0. Rank vrcholu v je daný rekurzívne ako $rank(v) = 1 + min\{rank(left(v)), rank(right(v))\}$ Pre ľavicovú haldu špeciálne platí, že rank pravého syna je vždy menší alebo rovný ako rank ľavého syna.

2.3.2 Operácie

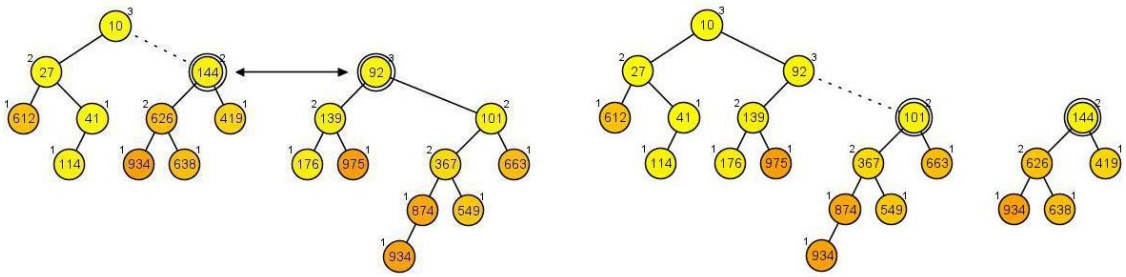
Operácia $meld(i, j)$**.** Haldy sa spájajú pozdĺž pravej cesty. V prvej fáze postupne prechádzame odvrchu nadol celú pravú cestu haldy i a porovnávame kľúče s koreňom

haldy j. Ak narazíme na kľúč vrcholu v v halde i, ktorý je väčší ako kľúč v koreni w haldy j, haldu zakorenenú vo w a haldu j vymeníme. Teda z vrcholu w sa stane pravý syn otca v a z podstromu zakoreneným vrcholom v sa stane halda j (obr. 2.3). Kľúč prázdnej haldy považujeme za nekonečno. Takto pokračujeme, až kým nedôjdeme na koniec pravej cesty haldy i.

Program 2.4 1.fáza operácie $meld(i, j)$

```
void meld1(i,j) {
    Node w = i;
    while (w != null) {
        if (key(w) > key(j)) {
            vymeňPodstromy(w, j);
        }
        w = right(w);
    }
}
```

Potom nasleduje fáza úpravy rankov. Ranky sa mohli zmeniť len na pravej, spájacej ceste, preto ich pozdĺž tejto cesty zdola nahor upravíme.
Nakoniec pre ľavicovú haldu musí byť dodržané pravidlo o veľkosti rankov synov. Preto opäť prejdeme pravú cestu a pokiaľ je niekde pravidlo porušené, bratov vymeníme.



Obr. 2.3: *Spájanie pozdĺž pravej cesty na ľavicovej halde.* V danom momente spájame podstrom 144 a 92 (vľavo). Aby sme zachovali podmienku haldy, pravý syn vrcholu 10 bude menší prvok z dvojice 144, 92. Preto tieto podstromy vymeníme a pokračujeme v spájaní podstromov 101 a 144 (vpravo).

Operácia $insert(x)$**.** Akonáhle vieme vykonať operáciu $meld(i, j)$, pridať operáciu $insert(x)$ je jednoduché: Vytvorí sa nová jednoprvková halda obsahujúca iba vrchol s kľúčom x a tá sa spojí s pôvodnou pomocou funkcie $meld$.

Operácia *deleteMin*. Najprv sa vymaže koreň haldy v a potom sa zavolá $meld(left(v), right(v))$.

Operácia $decreaseKey(v, \Delta)$. Implementuje sa rovnako ako v binárnej halde, teda po znížení kľúča sa vrchol "prebuble"nahor. Táto operácia nie je pre ľavicový haldu štandardná. Naše prevedenie nie je najefektívnejšie, pretože negarantuje logaritmickú hĺbku vrcholu.

2.3.3 Časová zložitosť

Veľkým plusom ľavicovej haldy je spájanie v logaritmickom čase. Toto sa dosiahne vďaka tomu, že cesta, pozdĺž ktorej sa dve haldy spájajú, sa udržuje čo najkratšia. Zabezpečuje to práve vlastnosť rankov synov, ktorá vlastne hovorí, že ľavá cesta je vždy aspoň taká dlhá, ako pravá cesta. Toto platí o každom podstrome haldy a z toho nám potom vyplýva, že vrcholov na pravej ceste je maximálne $\lg n$. Operácie $insert(x)$ a *deleteMin* majú rovnakú zložitosť ako $meld(i, j)$.

2.3.4 Modifikácie

Existuje "lenivá"verzia ľavicovej haldy [6], ktorá odkladá vymazávanie a spájanie na neskôr. Časová zložitosť týchto dvoch operácií sa stane konštantnou, na úkor operácie *findMin* (zložitosť je stále $O(\log n)$ ale iba v amortizovanom zmysle).

2.4 Skew halda

Skew halda je jednou zo samoupravujúcich sa háld. To znamená, že negarantuje dobrú časovú zložitosť pre najhorší prípad, ale stará sa o to, aby v budúcnosti robila operácie efektívnejšie. Pri takomto druhu haldy sa pozeráme na *amortizovanú časovú zložitosť*. Skew halda je odvodená z ľavicovej haldy. Jediný rozdiel je, že pre ňu nedefinujeme *rank*.

2.4.1 Definícia

Definícia. Skew halda je *halda* so stupňom vrcholov nanajvýš dva. Je charakteristická samoupravovaním počas operácie $meld(i, j)$.

2.4.2 Operácie

Operácia $meld(i, j)$**.** Prvá fáza operácie $meld(i, j)$ na skew halde je totožná s ľavicovou haldou. Postupne prechádzame pozdĺž pravej cesty haldy i a v prípade potreby haldy vymeníme. Keďže ranky tu neexistujú, druhá fáza spájania ľavicových háld sa preskočí a prejdeme k poslednej fáze. Táto časť obsahuje kľúčovú úpravu haldy, ktorá zabezpečuje efektívne spájanie. Postupujeme po pravej spájacej ceste haldy, ktorá vznikla v prvom kroku. Začneme v predposlednom vrchole[3] smerom nahor až po koreň a každému vrcholu po ceste vymeníme synov.

Operácie $insert(x)$**,** *deleteMin* **a** $decreaseKey(v, \Delta)$**.** Zvyšné operácie sú definované rovnako ako pri ľavicovej halde.

2.4.3 Časová zložitosť

Amortizovaná časová zložitosť pre $meld(i, j)$ je $O(\log n)$. To isté platí aj pre $insert(x)$, *deleteMin* a $decreaseKey(v, \Delta)$ [6].

2.4.4 Modifikácie

Ku skew halde tiež existujú alternatívy – top-down, bottom-up. Bottom-up prístup ohraničuje všetky operácie až na *deleteMin* na $O(1)$. *DeleteMin* má v tomto prípade časovú zložitosť $O(\log n)$.

2.5 Párovacia halda

Párovacia halda je ďalším druhom samoupravujúcej sa haldy. Opäť sa budeme pozerať na amortizovanú časovú zložitosť jej operácií.

2.5.1 Definícia

Definícia. Párovacia halda je *všeobecná halda*, teda počet synov nie je obmedzený.

Základná procedúra, ktorú budeme pri popise párovacej haldy používať je spájanie (*linking*) dvoch háld, pričom sa halda s väčším kľúčom v koreni napojí pod tú s menším kľúčom. Nový vrchol sa napája vždy ako prvý syn.

[3]Keďže posledný vrchol nemá pravého syna, nemá zmysel mu vymieňať synov, nanajvýš by sme tým predĺžili pravú cestu.

2.5.2 Operácie $meld(i, j)$, $insert(v, x)$ a $decreaseKey(v, \Delta)$

Operácia $meld(i, j)$ jednoducho spojí ("zlinkuje") dané dve haldy. $Insert(x)$ podobne len prilinkuje novú jednoprvkovú haldu. Operácia $decreaseKey(v, \Delta)$ najprv zníži hodnotu vrcholu v a keďže môže byť porušená podmienka pre haldu, strom zakorenený vo vrchole v sa odtrhne a prilinkuje ku zvyšku. Časové zložitosti pre všetky tieto operácie sú $O(1)$. Najzaujímavejšie na párovacích haldách je *deleteMin*. Po odstránení koreňa ostane les jeho detí. Môžeme zvoliť niekoľko prístupov ako z detí opäť vytvoríme jeden nový strom.

2.5.3 Operácia *deleteMin*

Naivné riešenie spájania detí do nového stromu hovorí, že si vyberieme jedno dieťa a ostatné k nemu prilinkujeme. V prípade, že si však zvolíme ako prvé dieťa minimum, po $(n-1)$ prilinkovaniach nám vznikne strom, ktorého koreň má $(n-1)$ detí. Postupnosť n operácií vloženia a vymazania minima nám v najhoršom prípade bude trvať $\Omega(n^2)$. Takéto riešenie má teda v najhoršom prípade až lineárnu zložitosť.

O niečo lepší nápad je deti najskôr v prvom prechode zlinkovať po pároch a v druhom prechode prilinkovať zvyšné stromy s jedným vybratým. Po každom odstránení minima a vložení prvku má koreň aspoň 2-krát menej detí. Takýto algoritmus už garantuje amortizovanú časovú zložitosť $O(\sqrt{n})$ (a existuje postupnosť n operácií, ktorá trvá $\Omega(n\sqrt{n})$).

Keď si dáme väčší pozor na to, ako deti párujeme, môžeme dosiahnnuť ešte lepšie výsledky. [3] dokázali, že ak v prvom prechode párujeme synov v poradí, v akom boli prilinkovaní od najmladšieho a v druhom prechode ich linkujeme sprava doľava, operácie *insert*, *meld*, *decreaseKey* a *deleteMin* majú amortizovanú zložitosť $O(\log n)$. Skutočná časová zložitosť týchto operácií je dodnes otvorený problém.

Ak *root* je koreň haldy, setParent(w,v) nastaví v ako rodiča vrcholu w a setRight(w,v) nastaví v ako pravého brata w, *left(v)* vráti smerník na najľavejšieho syna a *right(v)* vráti smerník na pravého brata v, tak Program 2.4 a Program 2.5 nám ozrejmí prístup pre párovanie zľava doprava a potom linkovanie sprava doľava (LRRL). Funkcia *link(i,j)* vracia haldu, ktorá vznikne po zlinkovaní haldy i a haldy j.

Podobný prístup majú *back-front* a *front-back* varianty, akurát pri druhom prechode dodržiavame rovnaký smer linkovania ako v prvom prechode pri párovaní.

Ďalší možný prístup je *multipass*, kde pri odstraňovaní minima postupne párujeme zvyšné stromy (na viacero prechodov), až kým nám neostane jediný strom. Iná alternatíva je *lazy* variant. Po odstránení minima ponecháme les synov. Pri ďalšom dopyte na minimum tento les prechádzame a popri hľadaní minima deti párujeme.

Program 2.5 vymazanie minima, párovanie

```
void deleteMin() {
    if (pocetDeti(root) > 0){
        Node w = left(root);
        Node wr = null, wrr = null;
        if (w != null){
            wr = right(w);
        }
        if (wr != null){
            wrr = right(wr);
        }
        // linkovanie do párov
        for (int k = 1; k <= pocetDeti(root) / 2; k++){
            w = link(w, wr);
            setParent(w, root);
            setRight(w, wrr);
            w = wrr;
            if (w != null){
                wr = right(w);
            }
            if (wr != null){
                wrr = right(wr);
            }
        }
    }
}
```

Program 2.6 vymazanie minima, linkovanie párov

```
void delMinLink() {
    if (pocetDeti(root) > 0){
        w = rightmostChild(root);
        if (w != null){
            wr = right(w);
        }
        if (wr != null){
            wrr = right(wr);
        }
        for (int k = 1; k < pocetDeti(root); k++){
            w = link(w, wr);
            setParent(w, root);
            setRight(w, wrr);
            if (w != null){
                wr = right(w);
            }
            if (wr != null){
                wrr = right(wr);
            }
        }
    }
    root = left(root);
}
```

Kapitola 3

Intervalové stromy

Niekedy potrebujeme odpovedať na dopyty týkajúce sa intervalov prvkov v danom poli. Môžu to byť otázky typu aký je súčet prvkov na intervale, aké je maximum z intervalu, apod. Rýchlu odpoveď nám dajú *intervalové stromy*. Intervalových stromov existuje viacero druhov. Obvykle ich rozlišujeme podľa toho, aké informácie si v nich pamätáme. Napríklad v strome pre súčty si každý vrchol pamätá súčet na svojom intervale, v strome pre maximá si pamätá maximum intervalu atď.

Operácie. Operácie, ktoré *intervalové stromy* podporujú, sú:

- **insert** (x) – vloží vrchol s kľúčom x;
- **find** (i, j) – vráti minimum/maximum/sumu/... intervalu $< i, j >$;
- **changeKey** (v, k) – zmení kľúč listu v na k;

3.1 Intervalový strom

3.1.1 Definícia.

Definícia. *Intervalový strom* je *dokonale vyvážený úplný binárny strom*, ktorý reprezentuje pole o k prvkoch. Má 2^n listov, pričom $n = \lceil log(k) \rceil$. Každý z k listov predstavuje dáta v poli, zvyšné listy neobsahujú žiadne dáta. Listy reprezentujú v poli interval dĺžky 1. Všetky ostatné vrcholy reprezentujú interval, ktorý vznikne zložením intervalov ich synov.

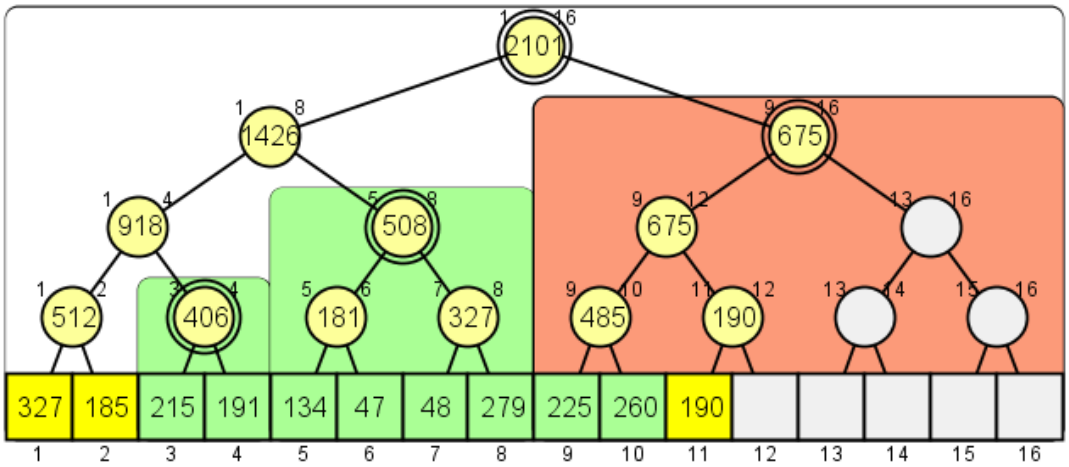
Zároveň intervaly vrcholov jednej hladiny na seba nadväzujú (vždy smerom zľava doprava). Z toho vyplýva, že zložením intervalov z vrcholov jednej hladiny dostaneme interval, ktorý si pamätáme v koreni.

3.1.2 Operácie

Bez ujmy na všeobecnosti budeme uvažovať o minimovom strome, teda vo vrcholoch stromu si pamätáme minimum z intervalu, ktorý vrchol reprezentuje. Operácie sú, s drobnými zmenami, ľahko aplikovateľné aj na iné druhy intervalových stromov.

Operácia *insert*(x)**.** Vloží prvok s hodnotou x na koniec poľa, teda do najbližšieho voľného listu. Môže sa však stať, že pole je práve zaplnené (počet listov je rovný počtu prvkov v poli), a preto budeme potrebovať strom rozšíriť. Zdvojnásobíme teda počet listov a vyrobíme k nim príslušný intervalový strom. Teraz už máme kam nový prvok vložiť a nasleduje úprava kľúčov vrcholov v strome. Od listu, kde sme prvok vložili až po koreň musíme skontrolovať, či platí *key(v) = min(left(key(v)), right(key(v)))*.

Operácia ***changeKey***(v, x)**.** Zmení hodnotu prvku z poľa v liste v na x. Avšak rovnako ako pri vkladaní sa musíme uistiť, či na ceste z listu do koreňa obsahujú vrcholy správne hodnoty.



Obr. 3.1: *Intervalový strom.* Hľadáme minimum na intervale 3 až 10, vrcholy 406 a 508 už boli nájdené ako jedny z reprezentantov.

Operácia *find*(i, j)**.** V našom prípade nájde minimum z intervalu $< i, j >$. Úlohou je nájsť vrcholy, ktoré interval $< i, j >$ reprezentujú (obr. 3.1). Budeme prehľadávať strom do hĺbky (DFS) a v každom vrchole sa pýtať v akom je vzťahu s hľadaným intervalom.
Môžu nastať tri prípady vzájomnej polohy intervalu, ktorý reprezentuje vrchol momentálne uvažovaný DFS (a) a intervalu, ktorý hľadáme (b).

1. Nepretínajú sa - v prehľadávaní vynecháme podstrom zakorenený v tomto vrchole.

2. Pretínajú sa ale interval (a) nie je vnorený do intervalu (b) - pokračujeme v prehľadávaní v podstrome.

3. Interval (a) je vnorený do intervalu (b) - našli sme vrchol, ktorý reprezentuje časť intervalu, ktorý hľadáme, takže nepokračujeme v hľadaní v podstrome.

Z vrcholov, ktoré sme našli v treťom prípade vyberieme maximum.

3.1.3 Časová zložitosť.

Hĺbka stromu s počtom listov n je $1 + \lg n$. Upravovanie stromu v prípade zmeny alebo vloženia nového prvku teda trvá $O(\log n)$. Časová zložitosť operácie $insert(x)$ je v najhoršom prípade $O(n)$. Je to práve vtedy, keď počet prvkov v poli dosiahol počet listov a musí sa alokovať 2-krát väčší priestor. Vytvorenie prázdneho stromu s počtom listov n trvá $O(n)$. Avšak ak sa pozeráme na zložitosť v amortizovanom zmysle, realokovanie sa deje len raz za n vykonaní $insert(x)$. Preto je amortizovaná časová zložitosť $O(\log n)$.
Operácia $changeKey(v, x)$ potrebuje po sebe iba upratať cestu ku koreňu, čo sa deje v $O(\log n)$.
Operácia $find(i, j)$ potrebuje tiež logaritmický čas od počtu listov. Vysvetlíme si to nasledovným spôsobom. Iba v jednom z troch prípadov vzájomnej polohy pokračujeme v prehľadávaní v podstrome. Zvyšné dva prípady budeme považovať za *ľahké*, keďže je prehľadávanie v podstrome skončené, trvá čas $O(1)$.

Pozrime sa na to, ako môže vyzerať interval, ktorý reprezentuje vrchol v. Na obr. 3.2 je zelenou znázornený interval, ktorý hľadáme a celé pole je interval, ktorý reprezentuje vrchol v.

Môže nastať 5 prípadov umiestnenia hľadaného intervalu. Každý z týchto prípadov sa po vyhodnotení vzájomnej polohy rozdelí na dva. V prípadoch a) a b) sú intervaly umiestnené na kraji, v prípadoch c), d) a e) sú intervaly v strede.

- V prípade a) je vyhľadávanie v oboch podstromoch skončené, teda trvá $O(1)$.
- Prípade b) sa rozdelil opäť na prípad intervalu na kraji a *ľahký* prípad, ktorým sme vylúčili polovicu vrcholov, ktoré nemusíme prehľadávať.
  Vidíme, že ak sú intervaly umiestnené na kraji, po každom zostúpení hlbšie vylúčime polovicu z vrcholov v podstromoch.
- Prípad c) sa rozdelí na interval na kraji a *ľahký* prípad.
- Prípad d) sa rozdelí na interval v strede a *ľahký* prípad.

Obr. 3.2: *Intervaly.*

- V prípade e) je nám ostanú 2 prípady na kraji, čo znamená, že pri nasledujúcich zostupoch opäť budeme prehľadávať nanajvýš jeden podstrom.

Vo všetkých z týchto piatich prípadov okrem e) sme zostúpením do hĺbky o 1 zahodili polovicu vrcholov, do ktorých sa nemusíme pozrieť. Všimnime si, že prípad e) môže nastať len jediný krát. To znamená, že ak počet vrcholov v strome je n najviac po $2 \cdot log(n)$ krokoch budeme mať prehľadaný celý strom.
Keď si počas prehľadávania priamo pamätáme minimum z doposiaľ nájdených reprezentantov intervalu, operácia $find(i, j)$ trvá $O(log(n))$.

3.1.4 Alternatívy.

Pri hľadaní intervalu si môžeme zvoliť viaceré postupy. Jeden sme už popísali vyššie a teraz sa pozrieme na ďalší, o niečo jednoduchší.
V prvom rade musíme zmeniť uloženie stromu v poli. Prvok na indexe 0 aj prvok na indexe n v poli bude 0. Súčet na intervale ¡i,j¿ zistíme tak, že budeme postupne prechádzať z indexu $i-1$ a z indexu $j+1$ smerom do koreňa až po vrchol p. Vrchol p je prvý spoločný predok na ceste z vrcholu $i-1$ a z vrcholu $j+1$ do koreňa. Keď na ceste z indexu $i-1$ vojdeme do vrcholu zľava, našli sme jedného reprezentanta intervalu. Tiež keď na ceste z indexu $j+1$ vojdeme do vrcholu sprava, našli sme jedného reprezentanta intervalu.

3.2 Fínsky strom

Fínsky strom je veľmi podobný *intervalovému stromu* pre súčty. Avšak zakladá na tom, že je zbytočné si pamätať všetky tri hodnoty kľúčov otca a jeho dvoch synov, keď z

dvoch vieme určiť tretiu. Preto si budeme pamätať vždy len kľúč rodiča a ľavého syna.

3.2.1 Definícia

Definícia. *Fínsky strom* je intervalový strom reprezentovaný v poli veľkosti $n = 2^k$ indexovaného od 1. Na indexe 1 je uložený prvý prvok, na indexe 2 je uložený súčet prvého a druhého prvku, na indexe 3 tretí prvok, na indexe 4 súčet prvých štyroch prvkov atď. Na indexe n je uložený súčet posledných 2^k hodnôt, kde k je pozícia prvého jednotkového bitu v binárnom zápise čísla n.

3.2.2 Operácie

Operácia *insert*(x)**.** Vloží prvok s hodnotou x na najbližší voľný index poľa. Môže sa však stať, že pole je práve zaplnené. Vtedy pole jednoducho zdvojnásobíme. Teraz potrebujeme upraviť niektoré prvky v poli. Konkrétne tie, ktoré obsahujú vo svojom intervale miesto, kde sme vložili nový prvok. Toto nám robí procedúra *insert*.

Program 3.1 Operácia vloženie

```
void insert(int index, int prvok) {
  rozdiel = prvok - strom[index];
  while (index<=n) {
    strom[index] += rozdiel;
    index = index + (index & -index);   // bitový and
  }
}
```

Výraz $(index + (index \,\&\, -index))$ *robí to, že sa v pomyslenom strome intervalov posunie o úroveň vyššie. Teda ak sme v intervale o veľkosti 2, tak sa dostaneme do intervalu veľkosti 4, ktorý daný interval obsahuje (tento interval je jednoznačný). Samotný výpočet robí to, že v čísle index vezme najpravejšiu jednotku a znova ju pričíta.* [7]

Operácia *changeKey*(i, x)**.** Zmení hodnotu prvku zo vstupného poľa na indexe i na x. Operácia sa implementuje ako *insert(i,x)*.

Operácia *find*(i, j)**.** Nájde súčet intervalu $< i, j >$. V skutočnosti nájdeme dve prefixové sumy po i a po j, ktorých rozdiel je potom súčet prvkov na hľadanom intervale. Ako zistíme prefixový súčet? Pomôže nám nasledujúci pseudokód.

Program 3.2 Prefixový súčet

```
int prefSucet(index) {
  int sucet = 0;
  while (index > 0) {
    sucet = sucet + strom[index];
    index = index & (index-1);
  }
  return sucet;
}
```

Vo výraze (index & (index−1)) *z funkcie prefSoucet() sa vynuluje najpravejší jednotkový bit v indexe. Tým sa dostaneme na prvý interval, ktorý sme ešte nepričítalii. Akonáhle máme index == 0, môžeme ukončiť výpočet, lebo už máme interval celý sčítaný.* [7]

Časť II

Vizualizácia

Kapitola 4

Softvér

Softvér bol vytvorený počas bakalárskej práce Jakuba Kováča . Od vtedy prešiel niekoľkými zásadnými zmenami a boli do neho doprogramované ďalšie dátové štruktúry a vylepšenia. Program je napísaný v jazyku Java. Ako vývojové prostredie sme použili Eclipse.

4.1 Popis

4.1.1 Časti okna aplikácie.

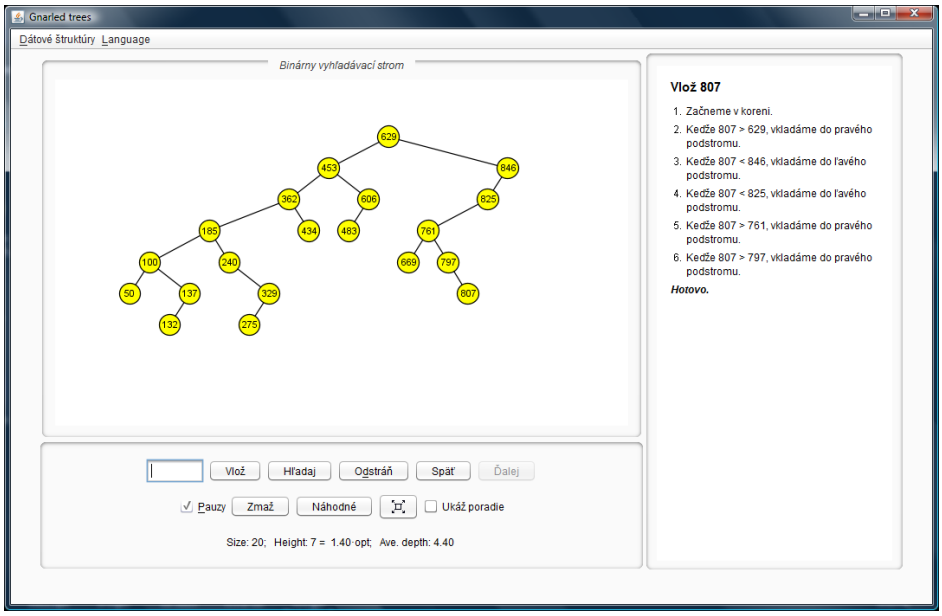
Na obr. 4.1 vidíme ukážku aplikácie. Každá dátová štruktúra má vlastnú sadu tlačidiel podľa operácií a nastavení, ktoré podporuje.

- Hore sa nachádza **menu**. Kliknutím na *Dátové štruktúry* sa nám rozbalí ponuka doposiaľ implementovaných dátových štruktúr. Softvér je dostupný v dvoch jazykoch - anglický a slovenský.

- Vpravo je **textová plocha**. Vypisujú sa na nej komentáre, čo práve s dátovou štruktúrou deje. Obvykle po každom kliknutí *Ďalej* sa vyzualizovaná operácia posunie o jeden krok a zároveň sa vypíše komentár, čo sa stalo, alebo čo bude nasledovať.

- V strede je **vykresľovacia plocha**, kde sa všetko odohráva. Je tu vykreslená dátová štruktúra a animujú sa tu kroky operácií. Vykresľovacia plocha sa dá ovládaním myšou priblížiť, či oddialiť, tiež posunúť a pri väčšine dátových štruktúr sa dajú označiť vrcholy.

- V spodnej časti sú **ovládacie prvky** - textové pole, do ktorého sa vpisujú vkladané prvky, tlačidlá pre každú operáciu, možnosť posúvať kroky operácií, checkbox na ovládanie rýchlosti operácií, tlačidlo na vygenerovanie a vloženie



Obr. 4.1: *Aplikácia*

náhodných prvkov a tlačidlo na zoom, prípadne ďalšie nastavenia konkrétnej dátovej štruktúry.

- Pod ovládacími prvkami sú ešte **informácie o dátovej štruktúre**, koľko prvkov obsahuje, prípadne aká je jej výška, priemerná hĺbka vrcholov a pod.

4.1.2 Ďalšia funkcionalita.

Pokiaľ do textového poľa napíšeme viac prvkov naraz a klikneme na *Vlož*, začnú sa všetky postupne vkladať. Pre jednoduchšie ovládanie sú pre štandardné operácie definované klávesové skratky.

Kapitola 5

Úvod

Vizualizačný softvér je prvotne zameraný pre študentov alebo ich učiteľov ako edukačný prostriedok. Preto sme ho prispôsobili na základe potrieb tejto skupiny používateľov. Vizualizualizácia pomáha pochopiť fungovanie dátových štruktúr z tej vonkajšej vrstvy.

Dobré vizualizácie oživujú algoritmy tým, že reprezentujú ich rôzne stavy a animujú prechody medzi nimi. Ilustrujú dátové štruktúry prirodzeným, abstraktným spôsobom namiesto toho, aby sa zameriavali na pamäťové adresy a funkčné volania.[5]

Prednosti a funkcie, ktoré náš softvér podporuje:

- interakcia
- možnosť zvolenia si vlastného tempa
- prídavné informácie o tom, čo sa práve deje
- farebné odlíšenia a animácia
- možnosť náhodného generovania prvkov
- jednoduchosť ovládania
- konzistentnosť
- pohodlný prístup (stránka)
- flexibilnosť

5.1 Existujúce vizualizácie

Množstvo naprogramovaných algoritmov je dostupných na portáli `algoviz.org`. Nami implementované dátové štruktúry sme porovnávali s dostupnými appletmi. Skew a ľavicová halda sú prístupné na `http://www.cse.yorku.ca/~aaw/Pourhashemi`, prípadne `http://people.cis.ksu.edu/~rhowell/viewer/heapviewer.html`. Avšak už všeobecne applety nespĺňajú ani základné požiadavky pre pútavú a zrozumiteľnú vizualizáciu. Na párovaciu haldu ani intervalové stromy sme doposiaľ žiadnu vizualizáciu nenašli a teda predpokladáme, že žiadna dostupná neexistuje.

5.2 Základné princípy vizualizácie

Vo väčšine prípadov sa **vrcholy** vykresľujú ako farebné kruhy ohraničené čiernou. Ak má vrchol kľúč, vypíše sa ako číslo do stredu vrcholu. Dodatočné informácie o vrchole sú buď charakterizované farbou vrcholu alebo, ak ide napríklad o *rank* pri ľavicovej halde, prípadne interval pri intervalových stromoch, sú vypísané tesne pri vrchole. Označený vrchol sa vizualizuje ako zakrúžkovaný.
Celé stromy sa vykresľujú tak, že koreň je vo vrchnej časti a ostatné vrcholy sú pod jeho úrovňou. Vrcholy s rovnakou hĺbkou sú na rovnakej úrovni.
Všetky vrcholy, ktoré sa pridávajú alebo vymieňajú nikdy nepriskočia "zrazu". Každý presun je animovaný. Akékoľvek zmeny v strome sú animované.

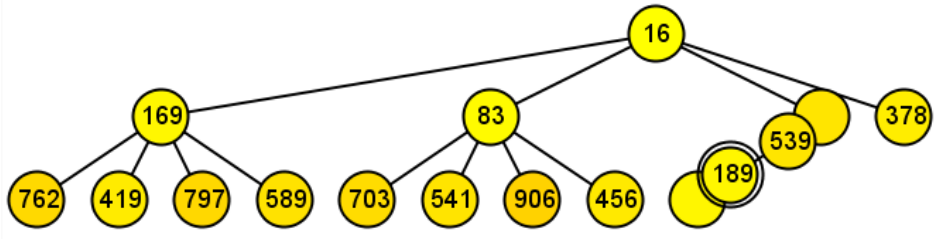
Kapitola 6

Vizualizácia háld

6.1 Spoločné vlastnosti.

Farba vrcholov. Pri haldách znázorňuje ich prioritu. Čím je odtieň tmavší, tým väčší je kľúč vrcholu. Teda pri *min haldách* je vrchol s najväčšou prioritou najsvetlejší, pri *max haldách* najtmavší.

Procedúry *bubbleup(v)* a *bubbledown(v).* Na základe podmienky pre haldu sa animuje "prebublávanie"postupne ako výmena vrcholov. V rámci šetrenia času sa výmena vrcholov v programe implementuje ako výmena kľúčov vrcholov. Vrchol, ktorý práve prebubláva je označený. Pred každou výmenou je animácia pozastavená.



Obr. 6.1: *Procedúra bubbleUp na d-árnej halde so stupňom vrcholov 4:* Vrchol s kľúčom 189 sa vymieňa s vrcholom s kľúčom 539, pretože má väčšiu prioritu.

6.2 D-árna halda

Operácia *insert(v).* Nový vrchol sa zaradí na koniec haldy. Potom sa už len vizualizuje procedúra *bubbleup(v).*

Operácia *deleteMin().* Označí sa minimum, teda koreň haldy a vymení sa s posledným vrcholom. Potom sa staré minimum odtrhne a putuje mimo vykresľovaciu

plochu. Vrchol v koreni prebuble nadol.

Operácia *decreaseKey(*v, Δ*).* Keďže haldy nepodporujú vyhľadávanie, na vykonanie operácie *decreaseKey(*v, Δ*)* musí byť vrchol, ktorému znižujeme hodnotu označený. Označuje sa kliknutím naň. Po znížení kľúča vizualizujeme procedúru *bubbleup(v)*.

6.3 Ľavicová halda

Každý vrchol má okrem kľúča zaznačený aj svoj *rank*.

Operácia *meld*(i, j) Vedľa seba sú štandardným spôsobom vykreslené dve haldy i a j, ktoré spájame. Keď porovnávame vrchol v z pravej cesty haldy i, koreň haldy j je na rovnakej úrovni ako vrchol v. Takto je zretelnejšie, ktoré vrcholy porovnávame a že skutočne prechádzame pravú cestu. Pokiaľ treba vrcholy vymeniť, jednoducho sa vymenia. V ďalšom kroku upravíme ranky. Nakoniec postupne od spodu označujeme vrcholy pravej spájacej cesty a pýtame sa, či je porušená požiadavka na ranky synov. Ak je, synov vymeníme.[1]

Operácie *insert*(v) **a** *deleteMin***.** Prebiehajú rovnako ako operácia *meld*(i, j). Pri vkladaní je halda j vrchol v a po vymazaní minima je halda i ľavý syn a halda j pravý syn.

Operácia *decreaseKey* (v, Δ)**.** Je vizualizovaná rovnako ako pri *d-árnej halde*.

6.4 Skew halda

Vizualizácia Skew haldy prebieha rovnako ako Ľavicovej haldy s rozdielom, že sa neupravujú ranky a v tretej fáze sa synovia vždy vymenia.

6.5 Párovacia halda

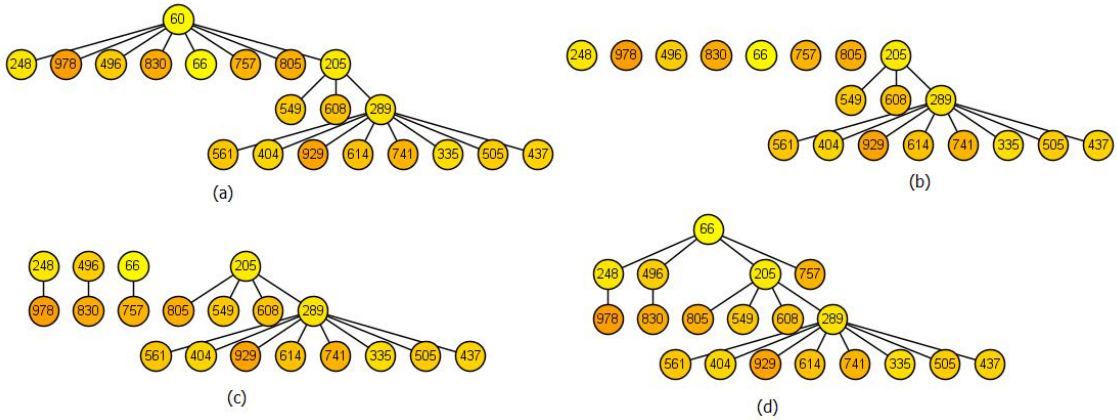
Párovacia halda využíva na vykresľovanie vrcholov Walkerov algoritmus. Tento algoritmus je spracovaný v triede pre vykresľovanie všeobecných stromov a prvotne bol pridaný pre *union-find problém*. Rozdiel oproti bežnému vykresľovaniu je v tom, že rozloženie vrcholov v strome je čo najtesnešie, čo robí dátovú štruktúru prehľadnejšou

[1] Druhý a tretí krok sa dajú robiť súčasne, avšak z hľadiska prehľadnosti vizualizácie a správneho pochopenia sú v našom programe implementované po sebe.

a šetrí to priestor.
Procedúra *linking* napojí jednu haldu na koreň druhej ako prvého syna.

Operácia *meld*(i, j) **a** *insert*(x)**.** Tieto dve operácie iba využijú *linkovanie*.

Operácia *decreaseKey*(v, Δ)**.** Označenému vrcholu najprv znížime hodnotu a po odtrhnutí sa prilinkuje.



Obr. 6.2: *Vymazanie minima z párovacej haldy.* (a) pôvodná halda, (b) halda po odstránení minima, t.j. koreňa stromu; (c) situácia po prvom prechode, keď zlinkujeme dvojice stromov zľava doprava, (d) halda po spájaní: v druhom prechode postupujeme sprava doľava a postupne linkujeme zvyšné stromy od najpravejšieho koreňa k najľavejšiemu.

Operácia *deleteMin*. Na párovanie sme implementovali *naivný* algoritmus a algoritmus *zľava doprava, sprava doľava*. Pri *naivnom* algoritme oddelíme jedného syna a ostatné postupne prilinkujeme. Pri algoritme *zľava doprava sprava doľava* sa haldy zľava popárujú a zlinkujú. Potom oddelíme najpravejšiu haldu a sprava ostatné postupne prilinkujeme (pozri obr. 6.2).

Kapitola 7

Vizualizácia intervalových stromov

7.1 Intervalový strom

Strom sme vykreslili štandardným úrovňovým spôsobom, avšak listy, ktoré predstavujú prvky v poli sme vykreslili hranato tesne vedľa seba. Z toho je jasnejšie, že ide o prvky v poli. Farebne sme odlíšili zaplnené listy (sýta žltá), vrcholy stromu, ktoré sú zatiaľ prázdne (svetlo sivá) a zvyšné vrcholy, ktoré obsahujú informáciu o (napríklad) minime svojich synov (svetlo žltá). Svetlo sivé vrcholy neobsahujú kľúč, čo značí, že intervaly, ktoré reprezentujú sú prázdne. Pri každom vrchole sú dve čísla, ktoré hovoria, aký interval vrchol reprezentuje a pod každým listom je index, na akom mieste sa prvok nachádza. Toto nám prispieva k prehľadnosti stromu a rýchlejšiemu orientovaniu v ňom.

Operácia *insert(v).* Pokiaľ je pole práve zaplnené, vytvorí sa nový sivý koreň a k nemu sa pripojí sivý pravý podstrom. Keď už je pole rozšírené, môžeme pridať nový vrchol na najľavejší sivý list. Po pridaní sa postupne prechádza cesta nahor ku koreňu, pričom sa označuje vrchol, ktorý práve uvažujeme a upravujú sa hodnoty vo vrcholoch.

Operácia *changeKey(v, x).* Listy, teda prvky v poli sa dajú označiť. Označenému prvku zmeníme hodnotu a prejdeme cestu ku koreňu podobne ako pri vkladaní.

Operácia *find(i, j).* Vrcholy, ktoré obsahuje interval, v ktorom hľadáme minimum sú v poli zelené. Hľadanie reprezentantov prebieha od koreňa. Podľa toho, v akom vzťahu je interval, ktorý reprezentuje vrchol momentálne uvažovaný DFS a interval, ktorý hľadáme, je pozadie celého podstromu zafarbené. Použili sme intuitívne farby pre rozoznanie situácie. V prvom prípade, keď sa nepretínajú, je to červená, pričom do hĺbky už nepokračujeme a vraciame sa v rekurzií. V druhom prípade, keď majú

spoločný prienik ale nie sú vnorené je to tiež červená, ale pokračujeme do hĺbky. V treťom prípade, keď sme našli reprezentanta je to zelená farba. Reprezentant s podstromom ostáva označený až do konca prehľadávania, teda na konci je zreteľné, ktoré vrcholy hľadaný interval pokrývajú.

Záver

V práci sme popísali niekoľko dátových štruktúr, menovite d-árnu haldu, ľavicovú haldu, skew haldu, párovaciu haldu a intervalové stromy. Zamerali sme sa na teoretický popis, ktorý je doplnený popisom prevedenia vizualizácie. Podstatnou časťou je výstupná aplikácia, ktorá dátové štruktúry implementuje. V rámci porovnania s dostupnými existujúcimi vizualizáciami je softvér na dobrej úrovni a ponúka niekoľko nadštandardných ovládacích prvkov. Nie len možnosť zasahovania do prebiehajúcej operácie, ale samotné vykresľovanie a pútavosť je to, čo študentovi, či učiteľovi uľahčí robotu. Vyzdvihli by sme ešte fakt, že dostupná vizualizácia párovacej haldy ani intervalového stromu zatiaľ neexistuje.

Počas budúcej práce na projekte by sme chceli rozšíriť softvér o ďalšie dátové štruktúry, hlavne fínsky strom, lenivú verziu ľavicovej haldy a tiež v párovacej halde doplniť ďalšie možnosti pre párovanie. Ďalej by sme chceli doplniť stručné informácie o dátových štruktúrach aj na stránku, kde sa aplikácia prezentuje.

Dopĺňame ešte informáciu, že zdrojové súbory sú dostupné na stránke GIT-u `http://github.com/katarinka/alg-vis`

Literatúra

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*, volume 2. The MIT Press, 2001.

[2] Peter M. Fenwick. A New Data Structure for Cumulative Frequency Tables. *SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(3), 327–336*, march, 1994.

[3] Michael L. Fredman, Robert Sedgewick, Daniel Dominic Sleator, Robert Endre Tarjan. The Pairing Heap: A New Form Of Self-Adjusting Heap. Algorithmica. 1986.

[4] Kubo Kováč. *Vyhľadávacie stromy a ich vizualizácia.* 2007.

[5] CLIFFORD A. SHAFFER, MATTHEW L. COOPER, ALEXANDER JOEL D. ALON, MONIKA AKBAR, MICHAEL STEWART, SEAN PONCE, and STEPHEN H. EDWARDS. Algorithm Visualization: The State of the Field. ACM Transactions on Computing Education. 2010.

[6] Daniel Dominic Sleator, Robert Endre Tarjan. Self-adjusting heaps. SIAM J. COMPUT, 1986.

[7] Karel Tesař. Recepty z programátorské kuchařky. Intervalové stromy. http://ksp.mff.cuni.cz/tasks/24/cook3.htm.