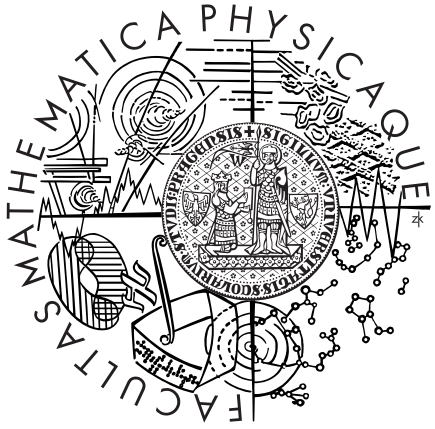
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Jan Neuvirth

Min–max haldy

Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Alena Koubková, CSc.

Studijní program: Informatika

Studijní obor: ISS softwarové systémy

Praha 2006

Děkuji vedoucí mé diplomové práce RNDr. Aleně Koubkové za její vedení a cenné rady, které mi v průběhu práce poskytla.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 17.4.2006 Jan Neuvirth

Obsah

Název práce: *Min–max haldy*
Autor: *Jan Neuvirth*
Katedra: *Katedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Alena Koubková, CSc.*
E-mail vedoucího: *Alena.Koubkova@mff.cuni.cz*

Abstrakt: *Diplomová práce se věnuje detailnímu popisu oboustranných prioritních front, které vycházejí z běžně používané datové struktury halda. Zároveň se snažíme zjistit, jak se tyto datové struktury chovají na velkých datech. Vzájemně pak poměřujeme kvality těchto datových struktur na základě dosažených výsledků v experimentálních testech. Nesnažíme se kategoricky vyhlásit nejlepší min–max haldu, raději chceme čtenáři poskytnout dostatečné množství užitečných informací, které by nám dovolily vybrat použití takové struktury, která by v daných podmínkách vykazovala nejlepší chování.*

Klíčová slova: *halda, oboustranná prioritní fronta, min–max halda, složitost*

Title: *Min–max heaps*
Author: *Jan Neuvirth*
Department: *Department of software engineering*
Supervisor: *RNDr. Alena Koubková, CSc.*
Supervisor's e-mail address: *Alena.Koubkova@mff.cuni.cz*

Abstract: *The object of the diploma thesis is detailed description of double-ended priority queues. Heap data structure is commonly used as their base. Simultaneously we are trying to uncover behavior of these data structures while working with big data and measure its quality on the basis of obtained results in experimental tests. The aim of the thesis is not to indicate the best min–max heap, but to offer a reader a satisfactory amount of useful information, which enable him to choose an optimal structure in certain conditions.*

Keywords: *heap, double-ended priority queue, min–max heap, complexity*

1 Úvod

Prioritní fronta je datová struktura, jejíž elementy mají přiřazen údaj reprezentující prioritu daného elementu. Takto definované prioritní fronty mají v informatice a potažmo v celém softwarovém průmyslu široké využití, například v nejrůznějších simulacích, externím třídění či operačních systémech.

Formálně lze pak na prioritní frontu nahlížet jako na datovou strukturu, která implementuje následující operace nad prvky zde uloženými: `Find-Min`, `Delete-Min`, `Insert`, `Increase-Key`, `Decrease-Key` a `Delete`.

Nejznámější prioritní frontou, která je v praxi běžně používána, je bezesporu klasická binární halda. V běžné praxi bývá však nutné efektivně implementovat zároveň kromě výše zmíněných operací i operace k nim symetrické, tj. zejména operace `Find-Max` a `Delete-Max`.

Obyčejná binární halda však neumožňuje současnou efektivní implementaci všech těchto operací. V případě max–haldy má například operace `Find-Min` dokonce lineární složitost oproti požadované konstantní. Tato práce si tedy klade za jeden z hlavních cílů popis alternativních datových struktur, které vycházejí z datové struktury halda a které efektivně implementují všechny výše zmíněné datové operace. Tyto datové struktury pak budeme nazývat souhrnným názvem min–max haldy.

Jedním z typických použití min–max hald, respektive oboustranných prioritních front (popis pojmu lze nalézt v kapitole 2), je bezesporu *externí quick sort*. Klasický quick sort, jak jej známe, rozděluje tříděná data na tři skupiny. Prostřední skupina obsahuje takzvaného *pivota*. Tento *pivot* pak rozděluje ostatní prvky do levé respektive pravé skupiny podle toho, zda je prvek menší či větší než *pivot*. Pravá i levá skupina je pak setříděna rekurzivním způsobem.

Při externím quick sortu však nejsou všechna data efektivně uložena v interní paměti. V interní paměti je uložena prioritní fronta, která slouží jako pivot běžného quick sortu. Ostatní prvky se načítají z disku, porovnávají s pivotem a pak vkládají do prioritní fronty nebo se zpátky odkládají na disk pro další fáze třídění. Myšlenka *externího quick sortu* je pak založena

na následující strategii:

1. Načti tolik prvků, kolik jde, do některé z min–max hald. Tyto prvky se tedy načtou do interní paměti.

2. Čti zbývající prvky.

3. a) Pokud je prvek $\leq$ než nejmenší prvek v prioritní frontě, bude tento prvek tvořit levou skupinu klasického quick sortu.

   b) Pokud je prvek $\geq$ než největší prvek v prioritní frontě, bude prvek tvořit pravou skupinu quick sortu.

   c) V opačném případě odstraň minimum nebo maximum z fronty (zde je možné střídat volbu, či vybírat náhodně). Je–li odstraněno maximum, přidej jej do pravé skupiny, jinak do levé skupiny. Vlož do prioritní fronty načtený prvek.

4. Vyprázdni prioritní frontu. Výstupem bude setříděná posloupnout.

5. Setřiď stejným způsobem vytvořenou levou a pravou skupinu, tj. rekurzivně.

Další uplatnění nacházejí prioritní fronty v dokumentografických informačních systémech (*DIS*). Úkolem *DIS* je nalézt v množině (textových) dokumentů ty, které svým obsahem vyhovují požadavkům uživatele *DIS* - tazatele. Zde se prioritní fronty užívají pro úkoly typu vybrání malého počtu nejvíce relevantních záznamů z poměrně velkého množství nalezených hitů (záznamy relevance). Takto nalezené záznamy jsou uživateli předkládány v pořadí klesající relevance. Jedním ze způsobů jak vyřešit tento problém je ponechání k nejvíce relevantních záznamů a smazání ostatních méně významných. Jakmile je prioritní fronta vybudována, tj. obsahuje všechny relevantní klíče, může být výsledných k relevantních záznamů poskytnut uživateli.

Výše zmíněné příklady budou pro nás příklady motivační a myšlenka na ně nás bude svým způsobem provázet celou prací.

V kapitole 2 zmiňujeme některé základní pojmy, které budeme dále v textu často používat.

Kapitola 3 se pak věnuje detailnímu popisu jednotlivých typů porovnávaných oboustranných prioritních front. Zaměříme se zde zejména na popis činnosti jednotlivých hald, tj. popíšeme technická úskalí ukrývající se za operacemi `Insert`, `Delete-Min` a `Delete-Max`. Rovněž zmíníme složitosti jednotlivých operací. Vše budeme dokumentovat na názorných obrázcích. Části zdrojových kódů budeme popisovat v komentovaném pseudokódu.

V kapitole 4 se budeme věnovat způsobu realizace jednotlivých testů provedených na experimentálních implementacích jednotlivých oboustranných prioritních front. Obecně si popíšeme jednotlivé testy. Rovněž zmíníme i způsob vzájemného poměřování kvalit implementovaných datových struktur.

Kapitola 5 pak obsahuje výsledky realizovaných testů. Formou přehledných grafů diskutujeme nad dosaženými výsledky a hodnotíme, proč tomu tak je.

V závěrečné kapitole 6 shrnujeme teoretické i prakticky ověřené znalosti oboustranných prioritních front.

2 Základní pojmy

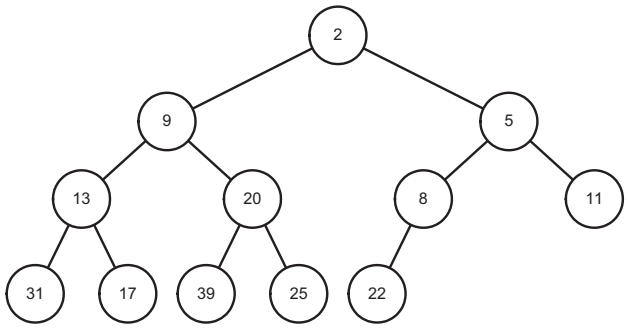
Binární halda je binární strom, který splňuje všechny níže uvedené vlastnosti:

1. V každé hladině tohoto binárního stromu vyjma hladiny poslední je maximální možný počet uzlů. Tj. v k–té hladině je vždy 2^{k-1} uzlů.

2. V poslední hladině jsou uzly umístěny co nejvíc vlevo. Tj. pokud budeme procházet předposlední hladinou binární haldy zleva doprava, bude mít prvních l uzlů vždy dva následníky, pak může mít jeden uzel levého následníka a zbývající uzly předposlední hladiny již následníky nemají.

3. Pro každý uzel platí, že hodnota v něm uložená je menší než hodnota uložená v libovolném jeho následníkovi.

V informatice bývá zvykem haldu ukládat do pole. Pokud budeme haldu postupně procházet od kořene po hladinách zleva doprava, můžeme jednotlivým prvkům přiřadit čísla od 1 do N. Následníky uzlu j jsou pak uzly s čísly $2j$ a $2j+1$. V další části práce budeme proto předpokládat, že je halda včetně jakékoliv její modifikace uložena vždy v poli indexovaném od nuly.



Obr. 2.1: Binární halda

Oboustranná prioritní fronta je datová struktura, která podporuje následující operace:

`Find_Min(`Q`)` – nalezení prvku s minimální hodnotou klíče
`Find_Max(`Q`)` – nalezení prvku s maximální hodnotou klíče

`Insert(`Q`, `x`)` – vložení nového prvku x do Q

`Delete_Min(`Q`)` – odebrání prvku s minimální hodnotou klíče

`Delete_Max(`Q`)` – odebrání prvku s maximální hodnotou klíče

Pokud budeme v další části textu zmiňovat *prioritní frontu*, budeme mít vždy na mysli *oboustrannou prioritní frontu*.

3 Přehled srovnávaných typů min–max hald

3.1 Symetrická min–max halda

V následující kapitole se budeme věnovat popisu symetrické min–max haldy, jak ji definovali Arvind a Rangan v [1].

Definice 1. (vlastnost P1) Nechť L je binární strom s ohodnocenými vrcholy. Pro uzel X nechť T_x označuje podstrom s kořenem X. Uzel X splňuje vlastnost P1, jestliže

1. Prvek s maximální hodnotou z hodnot uložených v uzlech či listech podstromu T_x vyjma hodnoty uložené přímo v kořenu X je uložen v pravém následníku kořene X.

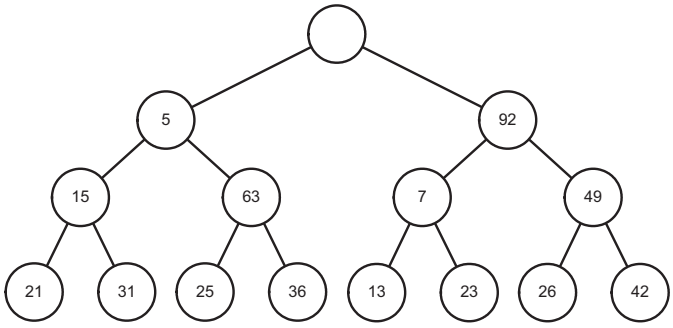
2. Analogicky prvek s minimální hodnotou z hodnot uložených v uzlech či listech podstromu T_x vyjma hodnoty uložené přímo v kořenu X je uložen v levém následníku kořene X.

Definice 2. Symetrická min–max halda (SMM) je binární halda, která splňuje následující vlastnosti:

1. Kořen neobsahuje žádný element. Je to pouze pomocný uzel.

2. Každý uzel v haldě splňuje vlastnost P1.

Příklad symetrické min–max haldy je uveden na obrázku 3.1



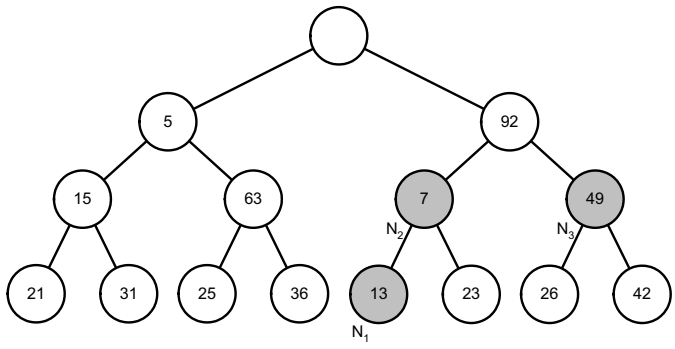
Obr. 3.1: SMM halda

Nechť N je uzel SMM haldy a nechť X je jeho rodič. Uzel Y nechť představuje bratra uzlu X a P rodiče uzlů X a Y. X je pak označen jako

levý sourozenec Y, pokud je X levým synem rodiče P. Analogicky definujeme i pravého sourozence jako pravého syna rodičovského uzlu P.

Dále řekneme, že uzel X je *Lnode(N)*, jestliže X je levým synem svého rodičovského uzlu, jinak jej nazveme *Rnode(N)*. Obdobně řekneme, že uzel Y je *Lnode(N)*, pokud je Y levým synem rodiče P, nebo *Rnode(N)*, pokud je Y pravým synem.

Pro názornost uvažujme příklad na obrázku 3.2. Nechť N_1 je uzel s hodnotou 13, N_2 uzel s hodnotou 7 a N_3 uzel s hodnotou 49. Potom *Lnode*(N_1) je uzel N_2 a *Rnode*(N_1) je uzel N_3.



Obr. 3.2: Vztahy *Lnode(N)* a *Rnode(N)*

Definice 3. (vlastnost P2) Řekneme, že uzel X splňuje vlastnost P2, pokud *Lnode(X)* není definován, nebo jestliže hodnota uložená v uzlu X je větší nebo rovna hodnotě uložené v uzlu *Lnode(X)*.

Definice 4. (vlastnost P3) Řekneme, že uzel X splňuje vlastnost P3, pokud *Rnode(X)* není definován, nebo jestliže hodnota uložená v uzlu X je menší nebo rovna hodnotě uložené v uzlu *Rnode(X)*.

Pro symetrickou min–max haldu pak platí následující jednoduché lemma uvedené v [1].

Lemma 3.1. *Nechť L je binární halda taková, že v každém uzlu vyjma kořene je uložena právě jedna hodnota. Potom L je SMM halda, pokud splňuje následující podmínky:*

1. *Každý uzel L splňuje vlastnosti P2 a P3*

2. *Hodnota uložená v pravém sourozenci je větší než hodnota v levém sourozenci.*

Důkaz. Důkaz proveden indukcí. □

3.1.1 Operace INSERT

Operace vkládání prvku do SMM haldy je velmi podobná operaci vkládání u klasické binární haldy. Nový prvek je vložen na poslední místo v haldě a následně je probublán na své pravé místo, přičemž se kontroluje splnění vlastností P1, P2 a P3.

Algoritmus 1 SMM Insert

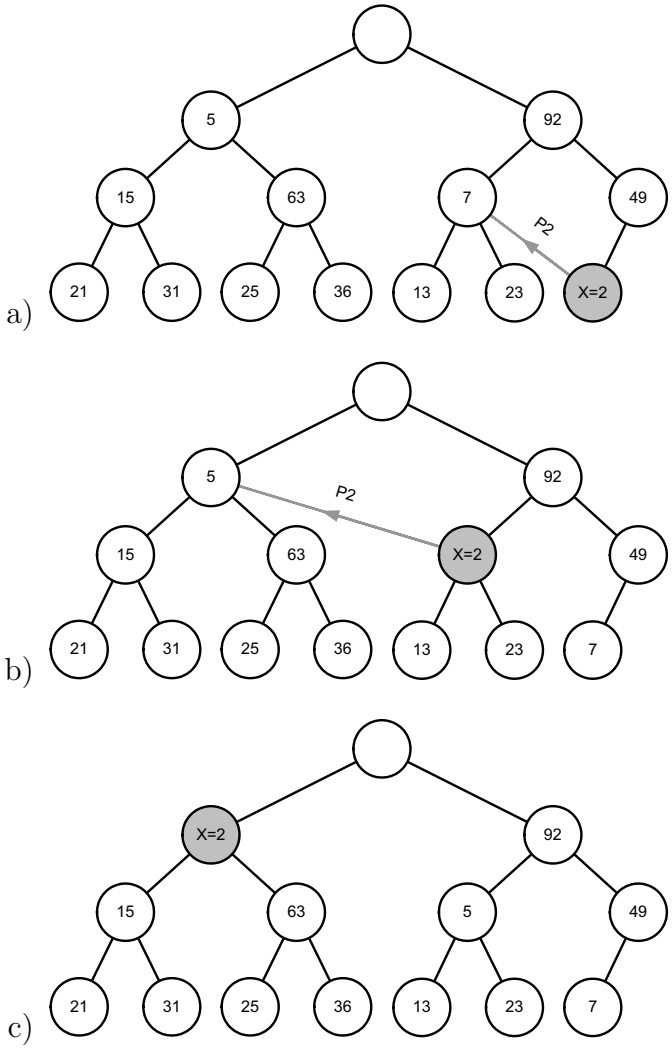
```
last++                    {X nový uzel, x klíč prvku, last počet prvků v haldě}
if x < left_sibling(X) then
  swap(X, Lnode(X))
end if
not_inserted ← true
while not_inserted do
  if x < key(Lnode(X)) then
    swap(X, Lnode(X))
  else if x > key(Rnode(X)) then
    swap(X, Rnode(X))
  else
    not_inserted ← false
  end if
end while
```

Průběh operace INSERT si názorně ukážeme na sérii obrázků 3.3. Předpokládejme, že do SMM haldy vkládáme prvek A s hodnotou 2. Nový prvek vložíme samozřejmě nejprve na poslední volné místo v haldě. Je snadné nahlédnout, že prvek A nyní nesplňuje vlastnost P2, neboť levý syn praotce A má hodnotu větší než je hodnota uložená v A. Proto se provede jednoduché prohození těchto uzlů.

Dostaneme tak situaci popsanou obrázkem 3.3.b. Vidíme nyní, že vlastnost P1 nemůže být porušena. Podobně nemůže být porušena ani vlastnost

P3. Uzel A ale opět nesplňuje vlastnost P2, a tudíž dojde opět k prohození prvků.

Korektnost algoritmu plyne z předchozího lemmatu 3.1.



Obr. 3.3: Vložení prvku

3.1.2 Operace DELETE–MIN

Operace odebrání minima z SMM haldy je opět velmi podobná operaci odebrání minima z klasické binární haldy. Prvek na pozici 1 nám zde označuje uzel s minimální hodnotou. Na tuto pozici vložíme poslední prvek v haldě a snížíme čítač počtu prvků o 1. Nyní musíme zkontrolovat, zda nově vložený

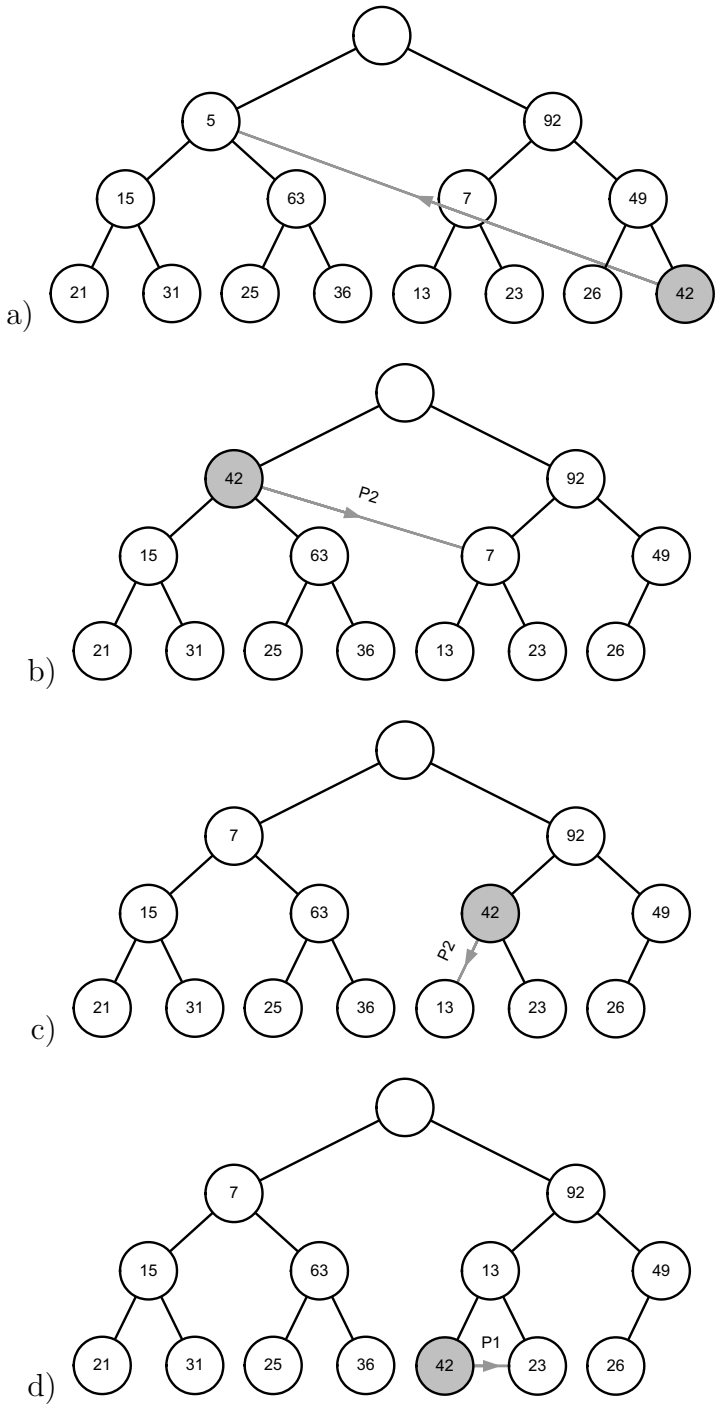
prvek na pozici minima neporušuje vlastnosti P1 či P2. Vlastnost P3 v tomto případě nemůže být nikdy porušena, proto testování této vlastnosti při odebrání minima zanedbáme. Testujeme tedy vždy nejprve hodnotu v sourozenci. Pokud není vlastnost P1 porušena testujeme vlastnost P2, tedy levého syna a levého syna sourozence. Celou situaci si můžeme názorně představit na obrázku 3.5.

Algoritmus 2 SMM Delete-Min

```
E ← 1, min ← h[E], h[E] ← h[last − −]    {E pomocný index do haldy}
continue ← true
while continue do
  if h[E] > h[E + 1] then
    swap(E,E + 1), E ← E + 1                          {vlastnost P1}
  else
    minor = h[E ∗ 2] < h[(E + 1) ∗ 2]?E ∗ 2 : (E + 1) ∗ 2
    if  h[E] > h[minor]  then
      swap(E,minor), E ← minor                        {vlastnost P2}
    else
      continue ← false
    end if
  end if
end while
return  min
```

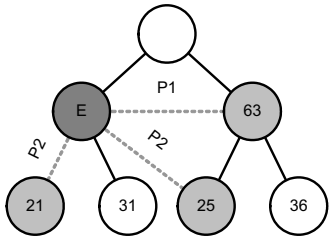
Vše si podrobně rozebereme na následujícím příkladu. Ze SMM haldy uvedené na obrázku 3.4.a budeme odebírat minimum. Na místo minima dosadíme poslední prvek haldy, jehož hodnota je 42. Dosazením za minimum se neporuší vlastnost P1, neboť hodnota uložena v jeho sourozenci je určitě větší. Zbývá tedy otestovat vlastnost P2. Vybereme minimum z hodnot levého syna a levého syna sourozence. V našem případě testujeme hodnotu 42 oproti hodnotě 7. Vlastnost P2 je tedy porušena a musí následovat prohození těchto uzlů.

Tento postup opakujeme tak dlouho, dokud prvek s hodnotou 42 neprobublá směrem dolů na své místo. Přehledný průběh odebrání minima včetně



Obr. 3.4: Odebrání minima

doplňujících popisků označujících vlastnost, která je porušena je pak rovněž uveden na sérii obrázků 3.4. V posledním kroku na obrázku 3.4.d se ještě



Obr. 3.5: Test vlastností P1 a P2

uplatní pravidlo P1 a provede se prohození prvků 42 a 23.

Přikládáme podrobný popis algoritmu 2 odebrání minima z SMM haldy.

3.1.3 Operace DELETE–MAX

Operace odebrání prvku s maximální hodnotou klíče je analogií předchozí operace `Delete-Min`. Místo vlastnosti P2 se testuje konzistence pravidla P3.

3.1.4 Složitost

Časová složitost operací `Insert`, `Delete-Min` a `Delete-Max` je logaritmická, tedy $O(\log n)$, kde n je počet prvků v haldě. Operace `Find-Min` a `Find-Max` má evidentně konstantní časovou složitost $O(1)$. Podrobnosti lze nalézt v [1] včetně jednoduchého náznaku důkazu.

3.2 Reflektovaná min–max halda

V následujícím odstavci se zaměříme na další z tzv. min–max prioritních front. Podrobně se seznámíme s reflektovanou min–max haldou (dále jako RMM). Reflektovanou haldou se blíže zabývali Makris, Tsakalidis a Tsichlas v [14]. My jsme aplikovali jejich výsledky na binární haldu.

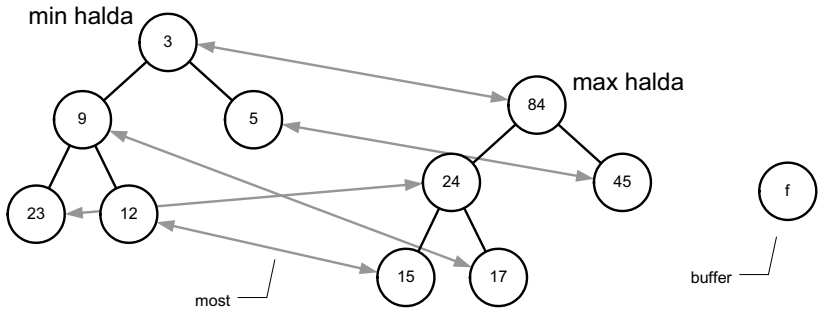
Základem k pochopení základních principů činnosti RMM haldy jsou dvě haldy, jejichž uzly jsou vzájemně propojeny prostřednictvím mostů. RMM halda Q je tedy složena ze dvou hald, Q_{min} a Q_{max}. Jedna halda, jak již jejich označení napovídají, reprezentuje min haldu a druhá max haldu. Označme si nyní prvek s minimální uloženou hodnotou v min haldě jako e_{min}. Obdobně si označme maximální prvek max haldy jako e_{max}. Nechť n pak označuje celkový počet prvků uložených v RMM haldě Q. Jestliže je n sudé, pak je $n/2$ prvků uloženo v Q_{min} a $n/2$ prvků v Q_{max}. Pokud je n liché, existuje zde takzvaný volný uzel F, který nepatří ani do jedné z hald.

RMM halda pak musí splňovat následující tři vlastnosti:

P1. Q_{min} a Q_{max} obsahují vždy stejný počet prvků, tedy $|Q_{min}| = |Q_{max}|$

P2. Každý uzel $u \epsilon Q_{min}$ má most u_b ukazující na nějaký uzel $v \epsilon Q_{max}$ a naopak. Zároveň musí platit, že $u_b = v$ a $v_b = u$.

P3. Pro hodnoty každého uzlu $u \epsilon Q_{min}$ a jeho most platí, že $u < u_b$. Analogicky pro prvky z Q_{max}.

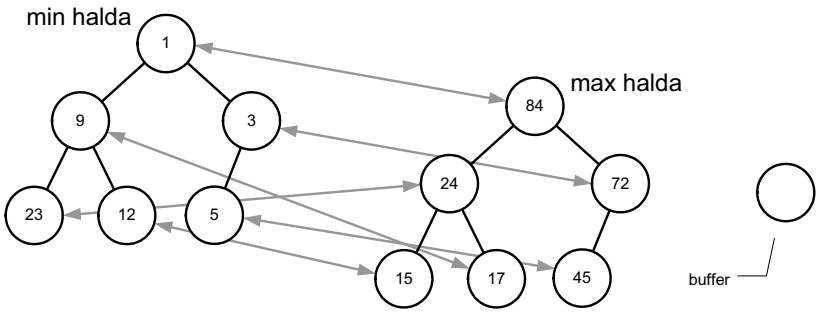


Obr. 3.6: RMM halda

Příklad RMM haldy včetně propojení prostřednictvím mostů je uveden na obrázku 3.6. Tato halda vznikla postupným vkládáním prvků 5, 84, 12, 15, 3, 45, 23, 24, 9 a 17.

3.2.1 Operace INSERT

Nový prvek X je do RMM haldy vkládán následujícím způsobem. Pokud je celkový počet prvků uložených v Q sudý, je nový prvek vložen jako volný uzel. Pokud je hodnota vkládaného prvku menší než e_{min} či větší než e_{max}, provede se příslušné prohození těchto prvků. Pokud je celkový počet prvků v Q lichý, spojí se nově vkládaný prvek prostřednictvím mostu s volným uzlem F. Bez újmy na obecnosti předpokládejme, že hodnota ve volném uzlu F je $<$ než hodnota v prvku X. V tom případě vložíme prvek F do haldy Q_{min} a prvek X do haldy Q_{max}. Je zřejmé, že takto budovaná min–max halda neporušuje ani jednu z vlastností P1, P2 či P3.



Obr. 3.7: RMM halda – operace Insert

Operaci si ještě názorně popíšeme na situaci, kdy budeme do haldy přidávat postupně prvky s klíči 1 a 72. Budeme vycházet ze RMM haldy znázorněné na obrázku 3.6. Nejprve přidáváme prvek s hodnotou 1. Buffer je prázdný, naplníme jej. Min prvek v min haldě je větší než prvek v bufferu, a tudíž je prohodíme. Nyní vkládáme prvek s hodnotou 72. Do min haldy tedy vložíme prvek z bufferu s hodnotou 3 a do max haldy prvek s hodnotou 72. Oba prvky spojíme mostem. Prvky následně probublají na své stabilní místo. Výsledek vložení je na obrázku 3.7.

Pro kompletnost uvádíme popis algoritmu 3 v pseudokódu.

Algoritmus 3 RMM Insert

```
if is_odd(inserted_items) then
  F ← X, inserted_items++          {X vkládaný prvek, F volný uzel}
  if F < e_min then
    swap(e_min,F)
  else if e_max < F then
    swap(e_max,F)
  end if
else
  make_links(X,F), inserted_items++
  if X < F then
    insert_min(X),insert_max(F)    {X vložen do Q_min, F do Q_max}
  else
    insert_min(F), insert_max(X)   {X vložen do Q_max, F do Q_min}
  end if
  F ← NULL, inserted_items++
end if
```

3.2.2 Operace DELETE–MIN

Nechť Y označuje uzel, na který ukazuje most příslušící k e_{min}. Na klasické min haldě Q_{min} zavoláme operaci `delete-min`, čímž odstraníme minimum z haldy. Následně nad max haldou Q_{max} provedeme operaci `delete` na prvek Y, tj. provedeme operaci `increase-key` na prvek Y, aby měl maximální hodnotu, čímž se přesune do kořene max haldy, a pak je odstraněn opět běžnou operací max haldy `delete-max`. Nakonec opět vložíme prvek Y do SMM haldy prostřednictvím operace `Insert` (viz odstavec 3.2.1).

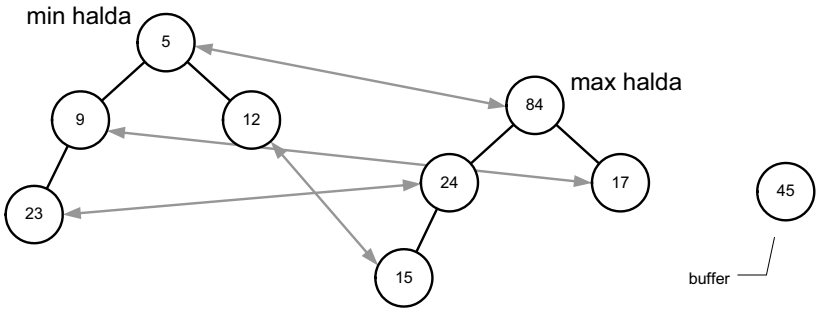
Celou situaci si ještě ozřejmíme na názorném příkladu. Vycházejme přitom z haldy uvedené na obrázku 3.6. Z haldy odstraňujeme minimum, jehož hodnota je 3. Minimum je prostřednictvím mostu spojeno s max haldou s prvkem 84. Prostřednictvím operace `delete-min` odstraníme z min haldy minimum. Prvek v kořenu je tak nahrazen posledním prvkem v min haldě, tj. prvkem s klíčem 12. Prvek následně propadne z kořene na své

stabilní místo. Při propadávání se nám patřičně upravují i mosty.

Algoritmus 4 RMM Delete–Min

$min \leftarrow e_{min}$	
$Y \leftarrow$ get_link(e_{min})	{korespondující prvek z Q_{max}}
delete_min()	{odstraň minimum z min haldy}
delete(Y)	{odstraň korespondující prvek z Q_{max}}
$inserted\_items- = 2$	{odstranili jsme dva prvky}
Insert(Y)	{znovu vlož do RMM haldy}
return min	

Podobně z max haldy odstraníme prvek, jenž byl spojen mostem s min prvkem v min haldě, tj. odstraňujeme prvek s klíčem 84. Prvek odstraňujeme prostřednictvím operace delete, tj. provedeme `increase-key` klíče na ∞, čímž jej přesuneme do kořene max haldy, patřičně se nám opět posouvají mosty. V tomto případě se žádné přemístění neprovede. Následně takto přemístěný prvek v kořenu opět nahradíme posledním prvkem v max haldě a necháme propadnout na své stabilní místo. Tj. do kořene přemístíme prvek s klíčem 17.



Obr. 3.8: RMM halda – operace Delete–Min

Odstraněný prvek s klíčem 84 nakonec opět vložíme do reflektované haldy. Jelikož je buffer prázdný, vložíme jej zde. Prvek v bufferu je však větší než maximální prvek v max haldě, a tudíž je oba prohodíme. Výsledek operace pak názorně vidíme na obrázku 3.8.

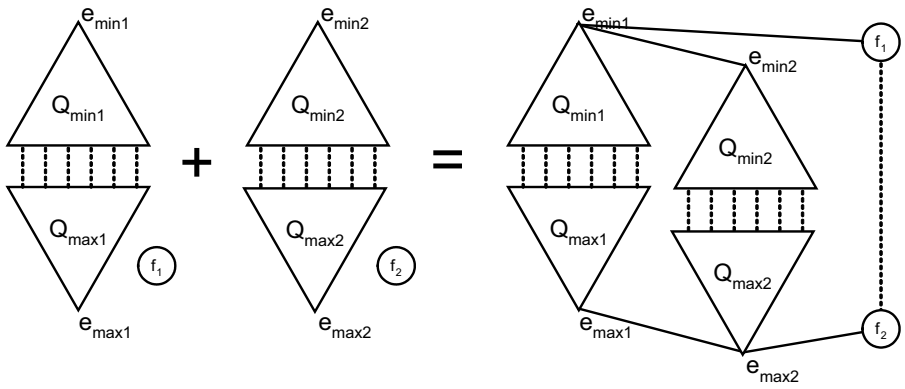
Opět uvádíme celý postup algoritmu 4 v komentovaném pseudokódu.

3.2.3 Operace DELETE–MAX

Operace odebrání prvku s maximálním klíčem z RMM haldy je doslovně symetrií předchozí operace `Delete-Min`. Nebude ji proto zde dále rozvádět.

3.2.4 Operace MELD

Kromě výše uvedených operací podporuje RMM halda implementovaná podle [3] či [9] i operaci `Meld`, tj. sloučení dvou prioritních front. Nejprve se provede sloučení dvou min a dvou max hald. Pokud budou po operaci slití volné dva uzly, provede se 2x operace Insert na již sloučené haldy. Je zřejmé, že sloučením dvou RMM hald není porušena žádná z vlastností P1, P2 a P3. Situaci sloučení dvou hald si můžeme ilustrovat na obrázku 3.9, který je převzat z [14].



Obr. 3.9: RMM halda – operace Meld

3.2.5 Složitost

Časové složitosti popisovaných operací uvedené v [14] vychází z implementace reflektované haldy prostřednictvím hald uváděných v [3] a [9]. Tyto haldy zejména umožňují implementaci operace `Meld` v konstantní složitosti.

Práce [14] rovněž uvádí zajímavé tvrzení 3.2.

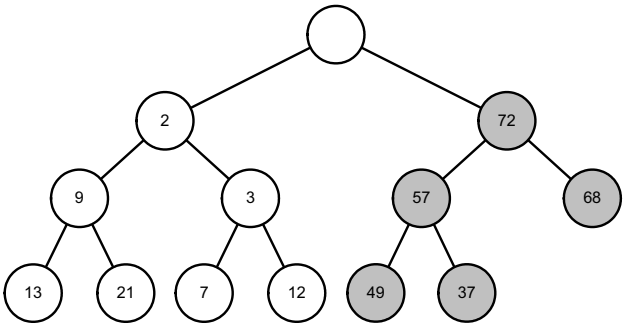
Věta 3.2. *Existuje implementace min–max prioritní fronty taková, že operace* ***`Find-Min`****,* ***`Find-Max`****,* ***`Insert`*** *a* ***`Meld`*** *mají v nejhorším případě konstantní časovou složitost, a zároveň operace* ***`Decrease-Key`****,* ***`Increase-Key`****,*

Delete-Min *a* ***Delete-Max*** *vyžadují časovou složitost v nejhorším případě rovnu* $O(\log n)$*.*

V případě naší implementace prostřednictvím binárních hald je časová složitost operací `Find-Min` a `Find-Max` evidentně $O(1)$. Časová složitost operace Insert je $\log(n/2)$, neboť střídavě volá 2x `insert` běžné binární haldy, jeden do Q_{min} a druhý do Q_{max}, do hald poloviční velikosti než je celkový počet prvků uložených v RMM haldě Q. Složitost operací `Delete-Min` a `Delete-Max` je stejná, neboť to jsou symetrické operace. Zaměříme se tedy pouze na operaci `Delete-Min`. Ta se skládá z dílčích volání operací `delete-min` (tj. běžné odebrání minima z obyčejné haldy), `delete` z max haldy (tj. odebrání prvku x z haldy, což znamená volání operace `increase-key(x,`∞`)` a následného `delete-max`) a `Insert`. Složitosti těchto dílčích operací jsou $O(\log n/2)$, složitost `Insert` je $\log(n/2)$. Výsledná složitost je tedy $O(4 \log n/2)$.

3.3 Dvoukoncová halda (*deap*)

V následující kapitole se blíže seznámíme s další strukturou založenou na oblíbené datové struktuře halda, která implementuje obousměrnou prioritní frontu. Zaměříme se na popis takzvané dvoukoncové haldy, anglicky zkráceně jako *deap*, jak ji uvádí Carlsson v [4].



Obr. 3.10: Dvoukoncová halda

Dvoukoncovou haldu si můžeme představit jako klasickou haldu s tím rozdílem, že kořen neobsahuje žádný prvek. V levém podstromu určeném tímto kořenem pak máme uloženou min haldu, kdežto v pravém podstromu max haldu. Datová struktura *deap* zavádí jisté vztahy mezi jednotlivými prvky v těchto min a max haldách. Pro deap pak musí platit následující vztahy:

1. Jakýkoliv list v min haldě je menší než jemu odpovídající list v max haldě. Zároveň tedy musí platit, že jakýkoliv uzel v min haldě je menší než jemu odpovídající uzel v max haldě, pokud existuje (tj. jestli $n{+}1 \geq k + a$). Tedy pro uzel k platí, že hodnota v k je $<$ než hodnota v uzlu $k + a$, kde $a = 2^{\log pos(k)-1}$

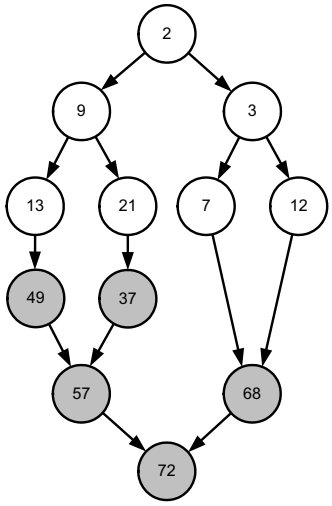
2. Pokud neexistuje korespondující uzel v max haldě ve stejné hladině, musí platit, že hodnota v k je $<$ než hodnota v prvku $(k + a)$ div 2, kde pro a opět platí $a = 2^{\log pos(k)-1}$.

Příklad dvoukoncové haldy je uveden na obrázku 3.10.

3.3.1 Operace INSERT

V následujícím odstavci uvádíme obecnou implementaci operace `Insert`, která se rovněž používá při operaci `Delete-Min`, potažmo pak v operaci `Delete-Max`.

Operace je obecná, neboť nově vkládaný prvek nevkládáme vždy na poslední místo haldy. Vkládáme jej na první volné místo v posledním patře haldy (toto volné místo totiž může vzniknout například předchozím použitím operace `Delete-Min`, více viz odstavec 3.3.2).



Obr. 3.11: Dvoukoncová halda – Hasseho diagram

Pokud nově vkládaný prvek je vložen na pozici v min haldě, musí se porovnat s jemu odpovídajícím prvkem v max haldě. Pokud je hodnota nového prvku větší než hodnota korespondujícího prvku v max haldě, provede se jejich výměna a nový element je vložen do max haldy. Takto vložený prvek následně probublá haldou na své stabilní místo technikou popsanou například v [5]. Nový element je následně konečně vložen na správné místo.

Pokud je nově vkládaný prvek vložen nejprve do max haldy, provede se analogie k výše uvedenému, tj. porovnání s korespondujícím prvkem v min haldě a následné probublání na stabilní pozici.

Celý algoritmus vychází samozřejmě z logických vztahů vyplývajících z Hasseho diagramu, který uvádíme na obrázku 3.11. Pro kompletnost uvádíme popis celého algoritmu 5 v pseudokódu.

Algoritmus 5 Deap Insert

```
pos ← find_free_position()
if pos in min heap then
  l ← correspondent item in max heap
  if h[pos] > h[l] then
    swap(pos,l), pos ← l
    while h[pos] > h[parent(pos)] do
      swap(pos, parent(pos)), pos ← parent(pos)
    end while
  end if
else
  analogy as above but in max heap
end if
```

3.3.2 Operace DELETE–MIN

Operace odebrání minima z *deap* haldy je poměrně jednoduchá. Minimum je k nalezení na druhém místě, pokud je halda uložena v poli indexovaném od 1. Následně postupně nahrazujeme uzel od tohoto minima nejmenším z obou synů. Tento postup opakujeme tak dlouho, až narazíme na list. Do tohoto listu pak vložíme poslední prvek uložený v haldě pomocí operace `Insert` uvedené v předchozím odstavci 3.3.1.

Následuje krátký a názorný popis algoritmu 6 v pseudokódu.

Algoritmus 6 Deap Delete-Min

```
hole ← 1, min ← h[hole]
while hole is not a leaf do
  if h[hole * 2] < h[hole * 2 + 1] then
    minor ← hole * 2                          {menší ze synů}
  else
    minor ← hole * 2 + 1
  end if
  swap(E,minor), hole ← minor
end while
Insert(hole, h[last − −])                     {zaplň vzniklou díru}
return min
```

3.3.3 Operace DELETE–MAX

Datová struktura deap je symetrická struktura. Proto je operace `Delete-Max` opět dokonalou symetrií výše popsané operace `Delete-Min` v odstavci 3.3.2.

3.3.4 Operace CREATE

Dvoukoncovou haldu lze vybudovat rychlejším způsobem než jen prostřednictvím opakovaného použití operace `Insert`. *Deap* je vybudována zespodu (postupně od listů) podobně jako se buduje běžná binární halda. *Deap* budujeme opakovaným voláním operací `Delete-Min` a `Delete-Max`. Postupně od konce haldy bereme jeden uzel za druhým. Na tento uzel následně voláme mírně modifikovanou operaci `Delete-Min` či `Delete-Max` podle toho, ve které haldě se testovaný uzel nachází. Modifikace `Delete` operací spočívá v tom, že nesestupujeme od kořene *deapu*, ale od právě testovaného uzlu. Jakmile již nemůže být provedena žádná výměna prvků, nastupuje opět mírně modifikovaná operace `Insert`, která na uvolněné místo vloží testovaný uzel. Následné prohazování prvků směrem ke kořenu haldy se zastaví na hranici určené hladinou původně testovaného uzlu.

Další podrobnosti lze nalézt v práci [16]. Carlsson se v práci [4] věnuje operaci `Create` jen okrajově a nevěnuje pozornost speciálním případům, které

mohou při fázi budování haldy nastat.

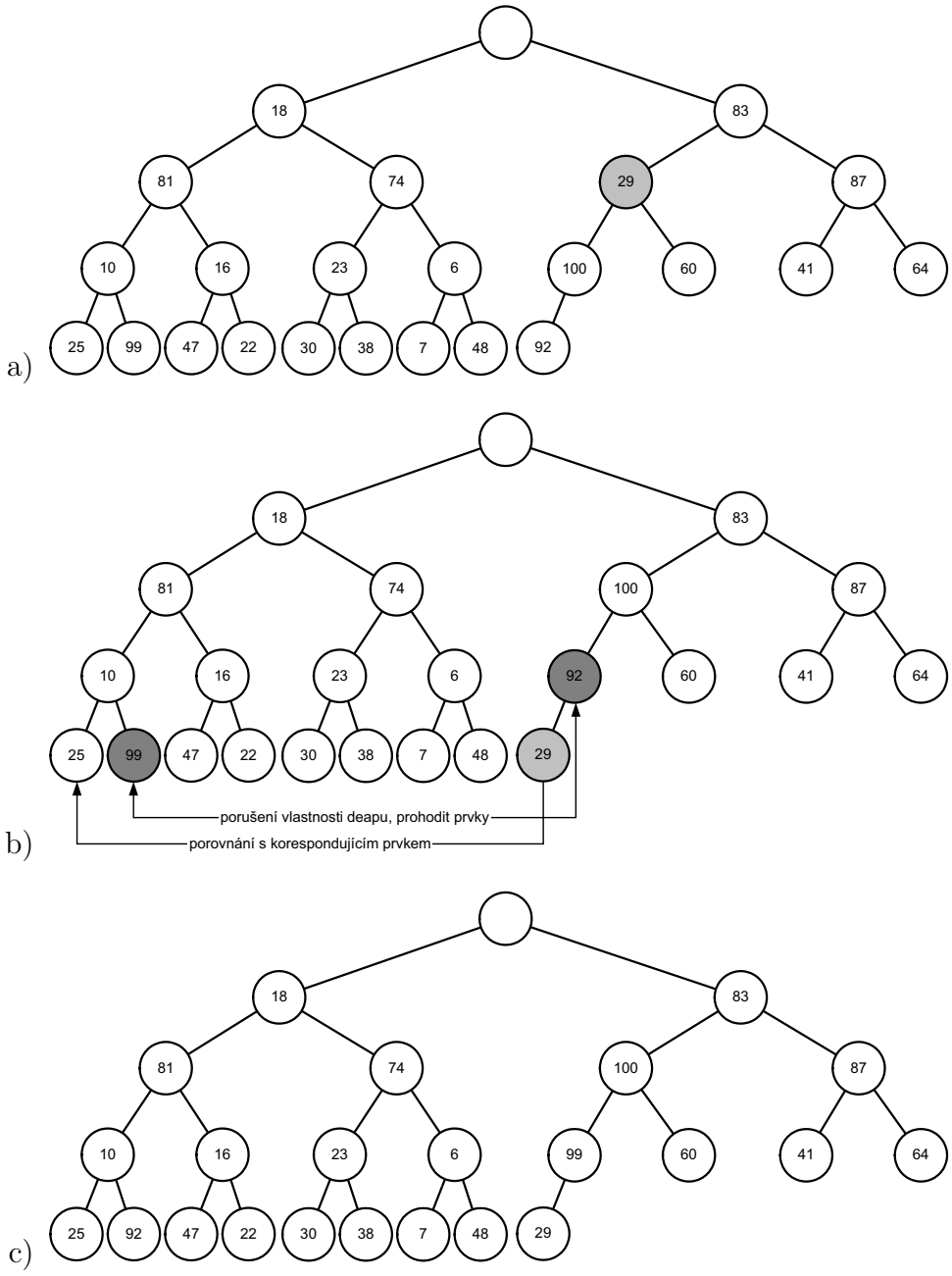
Algoritmus 7 Deap Create

```
for all j position in deap starting from last do
  if j in min heap then
    do Delete-Min, but start from element at j
  else
    do Delete-Max, but start from element at j
  end if
end for
```

Celou operaci `Create` si ještě popíšeme na názorném příkladu. Představme si situaci na obrázku 3.12.a, kdy již dle algoritmu 7 zpracováváme uzel s indexem 5 a hodnotou 29. Prostřednictvím modifikované operace `Delete-Max` prvek s touto hodnotou propadne až na spodní hladinu na pozici 23. Následně se na tuto pozici zavolá modifikovaná operace `Insert`. Provede se porovnání s korespondujícím prvkem v min haldě, tj. s prvkem na pozici 15, jehož hodnota 25 je menší než 29, a proto se žádné prohození neprovede. Nyní již může prvek probublávat na své stabilní místo. Jelikož jsme zůstali v max haldě, máme již stabilní místo. Dostáváme tak situaci popsanou obrázkem 3.12.b.

Situaci, kterou jsme popisovali, jsme vybrali záměrně, protože nám odhalí speciální případ, který může při fázi budování haldy nastat. Tento případ může nastat pouze v případě, že původně zpracováváme prvek, který je alespoň o dvě hladiny výš, než kam nakonec prvek vkládáme. V našem příkladu to byl prvek na pozici 5. Pokud se totiž podrobně podíváme na obrázek 3.12.b, zjistíme, že při procesu propadávání prvku směrem dolů, se porušila vlastnost, že každý prvek v min haldě je menší než jemu odpovídající prvek v max haldě. Jedná se o prvek na pozici 16 s hodnotou 99 a jemu korespondující prvek na pozici 11 s hodnotou 92. Tento speciální případ je odstraněn mírnou modifikací `Insert` operace. Stačí navíc testovat situaci, kdy je daný prvek, kam vkládáme, listem a jediným synem svého rodiče. V takovém případě je třeba navíc přidat porovnání jeho rodiče s bratrem korespondujícího prvku, viz popis v obrázku 3.12.b. V našem příkladu se na začátku `Insert` ope-

race ještě prohodí tyto dvě hodnoty. Nakonec dostáváme situaci popsanou obrázkem 3.12.c.



Obr. 3.12: Operace Create

3.3.5 Složitost

Dle Carlssona v [4] používá operace `Insert` $\lfloor \log \lfloor \log(n \; div \; i) \rfloor \rfloor + 1$ porovnání, pokud binárně hledáme vhodné místo pro prvek na pozici i. Podobně je na tom operace `Delete-Min`. Samotná operace používá $\lfloor \log(size + 1) div \; i \rfloor$ porovnání. Je však potřeba k ní ještě připočítat složitost operace `Insert`. Celkově pak podle [4] dostáváme následující složitosti:

`Insert`	$1 + \lfloor \log \lfloor \log(n \; div \; 2) \rfloor \rfloor + 1 \leq \log \log n + 2$
`Delete-Min`	$\lfloor \log((n+1) \; div \; 2) \rfloor + \lfloor \log \lfloor \log((n+1) \; div \; 2) \rfloor \rfloor + 1 \leq$ $\log n + \log \log n$
`Delete-Max`	$\lfloor \log((n+1) \; div \; 3) \rfloor + \lfloor \log \lfloor \log((n+1) \; div \; 2) \rfloor \rfloor + 1 \leq$ $\log n + \log \log n$
`Create`	$\sum_{i=1}^{3n/4} (\lfloor \log((n+1) \; div \; i) \rfloor +$ $\lfloor \log \lfloor \log((n+1) \; div \; i) \rfloor + 1 \rfloor + 1) \leq 2.07 \ldots n$

Carlsson, Chen a Strothotte v práci [6] dále zpřesňují odhad časové složitosti fáze budování haldy. Uvádějí, že časová složitost operace `Create` trvá minimálně $2.07n - 2 \log n$. Horní odhad časové složitosti v nejhorším případě je pak opraven z předchozích $2.07 \ldots n$ na $2.49 \ldots n$.

3.4 Diamantová halda

Další ze skupiny modifikací datových struktur založených na binární haldě je datová struktura představená Changem a Du v [7]. Tato struktura opět implementuje obousměrnou prioritní frontu. Byla vtipně pojmenována jako diamantová halda, neboť, jak si později ukážeme, grafická reprezentace logických vztahů vložených elementů, Hasseho diagram, vytváří strukturu podobnou perfektnímu diamantu.

Definice 5. Pole klíčů D_0,D_1,...,D_{M-1} nazveme *perfektní diamantová halda*, jestli jsou splněny následující vlastnosti:

P1. $M = 2 * (2^k - 1)$ pro nějaké číslo $k \geq 0$ a pro $0 \leq j < M$

P2. Jestli j je sudé a $j/2$ je liché, pak $D_{j/2-1} \leq D_j$

P3. Jestli j je sudé a $j/2$ je sudé, pak $D_{j/2-2} \leq D_j$

P4. Jestli j je liché a $(j-1)/2$ je liché, pak $D_{(j-1)/2} \geq D_j$

P5. jestli j je liché a $(j-1)/2$ je sudé, pak $D_{(j-3)/2} \geq D_j$

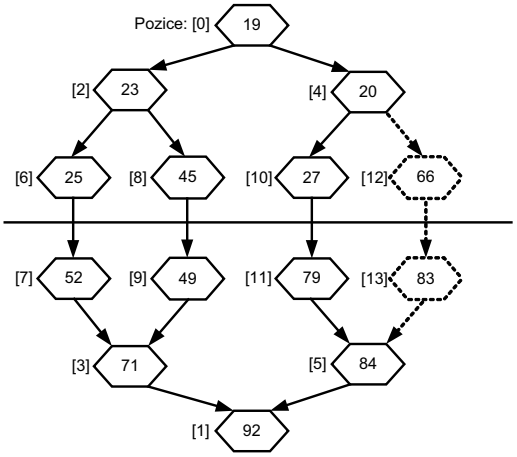
P6. Jestli j je sudé, pak $D_j \leq D_{j+1}$

Perfektní diamantovou haldu si můžeme představit jako pár min a max hald, které jsou logicky položeny jedna na druhou. Fyzicky jsou pak v poli uloženy průběžně vedle sebe. Min halda je uložena na lichých pozicích tohoto pole, kdežto max halda je uložena na pozicích sudých. Vlastnost P1 říká, že obě min i max haldy jsou binární stromy až do úrovně k. Implicitní vztah rodič–syn v min či max haldě pak zachycují vlastnosti P2–P5. Vlastnosti P2 a P3 pro min haldu, vlastnosti P4 a P5 pro max haldu. Vlastnost P6 pak konečně určuje uspořádání mezi min a max haldou.

Definice 6. Pole klíčů K_0, K_1, ..., K_{N-1} nazveme *kompletní diamantová halda*, jestli existuje perfektní diamantová halda D_0,D_1,...,D_{M-1} taková, že platí $M/2 \leq N \leq M$ a $K_i = D_i$ pro $0 \leq i < N$.

Kompletní diamantová halda je tedy dvoj halda vložená do diamantové haldy. Logické vztahy v takto komplikovaných haldách bývá zvykem

znázorňovat pomocí Hasseho diagramu. Na obrázku 3.13 je proto znázorněna diamantová halda s 12 elementy vložená do kompletní diamantové haldy s 14 prvky. Na obrázku pro názornost rovněž uvádíme indexy do pole, ve kterém jsou jednotlivé elementy uloženy.



Obr. 3.13: Diamantová halda – Hasseho diagram

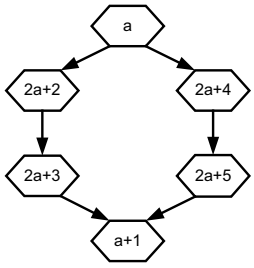
Pro úplnost ještě uvedeme další vlastnost, která je zřejmá z Hasseho diagramu. Hodnoty klíčů prvků uložených v diamantové haldě na jakékoliv cestě z horního k dolnímu konci by měly být vzestupně setříděné. Vlastnost P6 uvedená v definici 5 je v případě Hasseho diagramu použita jen pro hraniční uzly mezi min a max haldou. V našem případě by měla být nahrazena podmínkou (více viz [13]):

P6'. Jestli j je sudé, pak $D_j \leq D_{j+1}$ pro $N/2 - 1 \leq j < N$

3.4.1 Vztah předchůdce a následník

Předpokládejme perfektní diamantovou haldu K_0, K_1, ..., K_{N-1}. Z definice plyne, že K_i je v min haldě, pokud je i sudé, jinak je v max haldě. Rozhraní nebo možná lépe řečeno přechod mezi min a max haldou diamantové haldy si můžeme nejlépe znázornit pomocí hexagonu. Tento hexagon vytváří jednotlivé uzly min a max haldy. Například na obrázku 3.13 vytvářejí uzly na pozicích 2, 6 ,7, 3, 9 a 8 hexagon. Obdobně vytváří pozice 4, 10, 11, 5, 13, 12 hexagon další. Na základě těchto hexagonů nyní stanovíme podmínky definující vztahy *okamžití předchůdci* a *okamžití následníci* daného uzlu.

Pokud se K_i nachází v min haldě, může mít jednoho okamžitého předchůdce, levého okamžitého následníka (*LIS*) a pravého okamžitého následníka (*RIS*). Pokud je obdobně K_i v max haldě, může mít jednoho okamžitého následníka a opět pravého (*RIP*) a levého (*LIP*) okamžitého předchůdce. Pokud je K_i na poslední hladině min haldy, pak má jen jednoho okamžitého následníka na poslední hladině max haldy. Všechny tyto vztahy plynou bezprostředně z logických vazeb znázorněných v Hasseho diagramu.



Obr. 3.14: Vyjádření vztahu předchůdce a následník

Na obrázku 3.14 jsou pak výše uvedené vztahy vyjádřeny pomocí indexů do pole, ve kterém je diamantová halda uložena. Například pokud je K_j *LIS* prvku K_i v min haldě, pak platí, že $j = 2 * i + 2$ a $i = j/2 - 1$. Na druhou stranu pokud je K_i *RIP* elementu K_j v max haldě, platí, že $j = (i - 3)/2$ a $i = j * 2 + 3$. Pokud je K_i na poslední hladině min haldy, je jeho bezprostředním následníkem prvek K_{i+1} v poslední hladině max haldy.

V následujících kapitolách předpokládejme, že máme k dispozici kompletní diamantovou haldu $K_0, K_1, \ldots, K_{N-1}$.

3.4.2 Operace INSERT

Nový prvek s klíčem K vložíme nejprve na konec haldy, tj. $K_N = K$. Následně prvek K probublá směrem nahoru (*bubble–up*) či propadne směrem dolů v Hasseho diagramu (*trickle–down*) na své stabilní místo. Stabilní pozice je v tomto případě dosaženo, pokud platí, že všechny cesty z pozice 0 do pozice 1, jsou korektně vzestupně setříděné, viz Hasseho diagram na obrázku 3.13.

Zda provádět operaci *bubble–up* či *trickle–down* můžeme rozhodnout na začátku vkládání nového prvku.

3.4.3 Operace DELETE–MIN

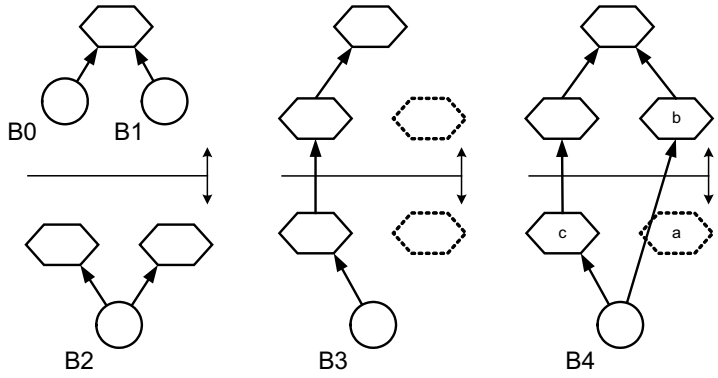
Hodnota minimálního klíče je rovna K_0. Abychom mohli smazat prvek s tímto minimálním klíčem, nahradíme prvek K_0 prvkem K_{N-1}. Následně musí nový prvek na pozici K_0 propadnout dolů na své stabilní místo. Operaci propadnutí si popíšeme později.

3.4.4 Operace DELETE–MAX

Hodnota maximálního klíče je rovna K_1. Tento prvek podobně jako v případě operace `Delete-Min` nahradíme prvkem s klíčem K_{N-1}. Prvek na nové pozici K_0 následně probublá směrem nahoru na své stabilní místo.

3.4.5 Dílčí operace Bubble–Up

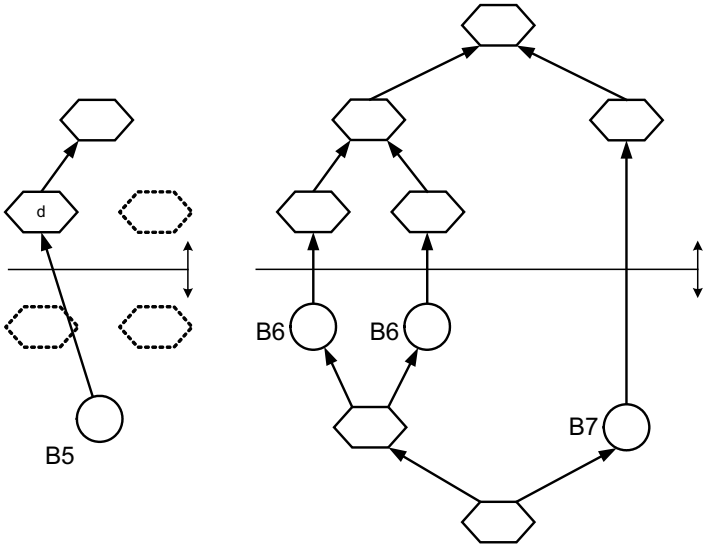
Ve všech předchozích případech se můžeme vždy jednoznačně rozhodnout, zda při procesu přemísťování prvků se budeme pohybovat v Hasseho diagramu směrem nahoru či dolů. Bez újmy na obecnosti předpokládejme, že se budeme pohybovat směrem nahoru. Nový prvek bude tedy probublávat. Připomeňme si, že každá pozice v Hasseho diagramu má korespondující pozici v poli, ve kterém ukládáme diamantovou haldu.



Obr. 3.15: Dílčí případy operace *bubble–up* (1)

V našem příkladu se budeme snažit najít další vhodnou pozici (bublinu) pro prvek, který hledá své stabilní místo. Na obrázcích 3.15 a 3.16 máme znázorněny všechny situace, které mohou při procesu propadávání nastat. Kolečko na obrázku znázorňuje další možné místo při procesu porovnání,

tj. bublinu. Pokud jsou bubliny v horní části obrázku, tj. v min haldě, jediné možné přijatelné místo pro další porovnání je bezprostřední předchůdce bubliny (situace B0 a B1). Pokud se bublina i oba jeho *LIP* a *RIP* nacházejí v dolní části, tj. v max haldě, je vybráno místo většího z obou předchůdců (situace B2).



Obr. 3.16: Dílčí případy operace *bubble–up* (2)

Pokud bublina dorazí na rozhraní mezi min a max haldou, může nastat celkově pět různých situací, které si nyní popíšeme (situace B3–B7). Pro případ B3 je prvek *LIP* jedinou volbou k dalšímu posunutí bubliny. Pro případ B6 a B7 je jedinou možností aktuální pozice bubliny plus jedna.

Případy B4 a B5 jsou trochu složitější. Nechť K je klíč dané bubliny. Případu B4 lze dosáhnout pouze v případě provedení operace `Delete-Max`, a proto klíč K musí být předchozím klíčem na pozici a. Proto K není menší než hodnota klíče uložená na pozici b, a proto musíme K porovnat pouze s *LIP* (pozice c). Podobně k situaci B5. K není menší než hodnota klíče na pozici d. Což má za následek, že bublina již našla své stabilní místo.

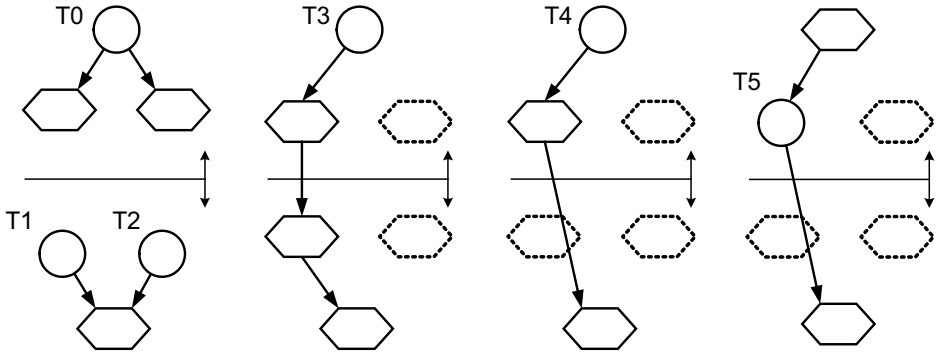
Pro úplnost uvádíme popis algoritmu 8 operace *bubble–up* v pseudokódu.

Algoritmus 8 Diamond Bubble–Up

```
while pos > 0 do
  if pos&1 == 0 then
    if pos&2 == 2 then
      predecessor ← (pos >> 1) − 1                          {B0}
    else
      predecessor ← (pos >> 1) − 2                          {B1}
    end if
  else
    LIP ← (pos << 1) + 1
    RIP ← LIP + 2
    if RIP <= last then
      if h[LIP] < h[RIP] then
        predecessor ← RIP                                   {B2}
      else
        predecessor = LIP                                   {B2}
      end if
    else if LIP <= last then
      predecessor = LIP                                     {B3 or B4}
    else
      predecessor = pos − 1                                 {B5, B6 or B7}
    end if
  end if
  if h[pos] < h[predecessor] then
    swap(pos, predecessor)
    pos ← predecessor
  else
    break
  end if
end while
```

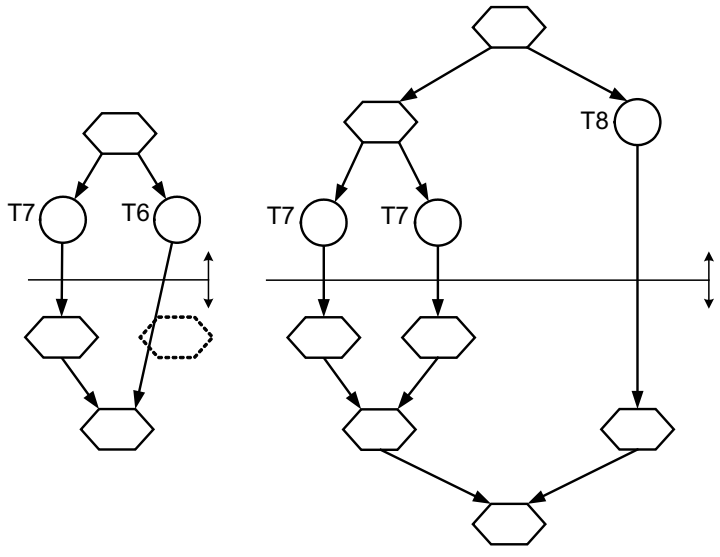
3.4.6 Dílčí operace Trickle-Down

Operace propadnutí směrem dolů v Hasseho diagramu je analogií předchozí operace *bubble–up* uvedené v 3.4.5. Popis situací včetně názorného schématu, které mohou nastat při operaci *trickle–down*, uvádíme na obrázcích 3.17 a 3.18.



Obr. 3.17: Dílčí případy operace *trickle–down* (1)

V Hasseho diagramu se pohybujeme směrem dolů. Stejně jako v případě *bubble–up* operace můžeme spojit dílčí případ T4 s případem T3 a situaci T8 s T7.



Obr. 3.18: Dílčí případy operace *trickle–down* (2)

3.4.7 Operace CREATE

Diamantovou haldu lze vybudovat v lineárním čase pomocí techniky popsané Floydem v [10]. Halda bude budována od svého prostředku ke svému hornímu a dolnímu konci.

3.4.8 Složitost

Chang a Du v [7] uvádějí časovou složitost pro operace `Insert`, `Delete-Min` a `Delete-Max` rovnu $O(\log n)$. Z konstrukce datové struktury je tato složitost zřejmá. Minimum a maximum je samozřejmě k nalezení v konstantním čase $O(1)$.

Chang a Du rovněž uvádějí, že pokud známe celkový počet prvků, který budeme chtít do haldy uložit, lze diamantovou haldu vybudovat v čase $O(n)$. Důkaz je proveden rekurzivní rovnicí. Nechť $C(n)$ označuje počet porovnání nutných pro vybudování diamantové haldy. Nechť $T(k)$ označuje počet porovnání nutných k vybudování diamantové haldy, která má $2k$ pater. Každá perfektní diamatová halda se skládá ze dvou dílčích diamantových hald o $2 * (k - 1)$ patrech a horního a dolního konce, viz Hasseho diagram na obrázku 3.13. Dostáváme tak následující rovnice.

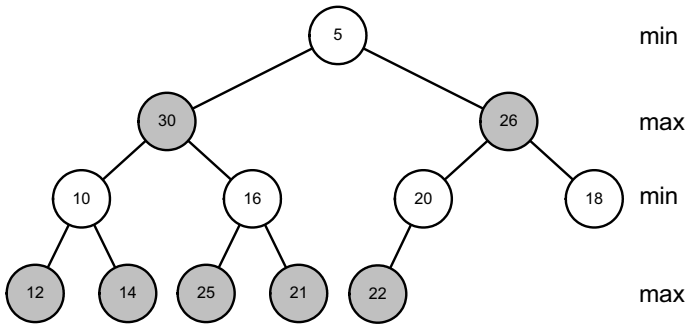
$$
\begin{aligned}
&T(0) = 0 \\
&T(k) = 2 * T(k-1) + 6 * k - 5 \\
&T(k) = 7 * 2^k - 6 * k - 7 \\
&C(n) = 7/2 * n - 6 * \log(n/2 + 1)
\end{aligned}
$$

Odtud konečně dostáváme složitost pro vybudování diamantové haldy rovnu O(n).

3.5 Min–max halda

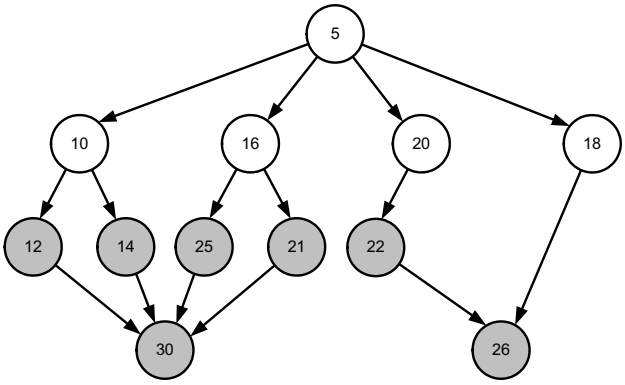
Na min–max haldu lze nahlížet jako na klasickou binární haldu, kde sudá hladina reprezentuje min haldu a lichá hladina max haldu. Na jednotlivých hladinách se tedy pravidelně střídají prvky s malými a velkými klíči. Touto datovou strukturou se podrobně věnovali Atkinson, Sack, Santoro a Strothotte v [2].

Kořen stromu obsahuje prvek s nejmenším klíčem. Kořen se tedy nachází na min hladině. Každý uzel na min hladině pak splňuje vlastnost, že je prvkem s nejmenší hodnotou klíče v podstromu, který je určen tímto uzlem. Obdobné pravidlo platí pro uzly na max hladině. V případě, že počet prvků uložených v haldě je větší než dva, je pak maximální prvek největší ze synů kořene. Příklad min–max haldy je uveden na obrázku 3.19.



Obr. 3.19: Min–max halda

Tato struktura může být zobecněna pro operaci `Find(`k`)`, tj. nalezení k–tého nejmenšího prvku. Nalézt takový prvek pak lze v konstantním čase. Implementovat v takto zobecněné struktuře lze v logaritmickém čase rovněž i operaci `Delete(`k`)`, tedy odstranění k–tého nejmenšího prvku. Hovoříme pak o tzv. min–max–medián haldě. Podrobnosti o této modifikaci lze rovněž nalézt v [2].



Obr. 3.20: Hasseho diagram min–max haldy

3.5.1 Operace INSERT

Nový element je do min–max haldy tradičně vložen na poslední místo. Konstrukce algoritmu pak vychází z Hasseho diagramu, jenž je uveden na obrázku 3.20. Listy min–max haldy jsou v Hasseho diagramu umístěny v prostřední hladině. Aby bylo zachováno min–max uspořádání po vložení prvku s novým klíčem, je třeba nechat tento prvek probublat či propadnout Hasseho diagramem na správnou pozici. Jakmile se rozhodneme pro směr pohybu prvku, zůstává tento směr již neměnný. Z obrázku 3.20 je rovněž patrné, že při cestě diagramem se pohybujeme vždy jen po prarodičích. Není tedy nutné porovnávat hodnoty klíčů syna s rodičem, neboť leží v různých haldách. Toto zjištění pak urychlí samotný průchod haldou.

Pro názornost přikládáme popis algoritmu 10 a jeho dílčí části algoritmu 9 v pseudokódu včetně popisu dílčí operace `BubbleUpMin`, jenž má za úkol probublat prvek po min hladinách. `BubbleUpMax` je pak analogií předchozího, prvek probublává po max hladinách.

Algoritmus 9 MM BubbleUpMin

if $h[i] < h[\text{grandparent}(i)]$ **then**
  swap(i, grandparent(i))
  BubbleUpMin(grandparent(i))
end if

Algoritmus 10 MM Insert

```
h[++last] ← element, i ← last
if pos is on min level then
  if h[i] > h[parent(i)] then
    swap(i, parent(i))
    BubbleUpMax(parent(i))
  else
    BubbleUpMin(i)
  end if
else
  if h[i] < h[parent(i)] then
    swap(i, parent(i))
    BubbleUpMin(parent(i))
  else
    BubbleUpMax(i)
  end if
end if
```

3.5.2 Operace DELETE-MIN

Princip odebrání minima z min–max haldy je opět velmi podobný jako v případě odebrání minima z běžné haldy. Prvek s minimální hodnotou klíče se nachází v kořenu haldy. Nejmenší prvek je tedy nahrazen posledním prvkem v haldě. Následně musí prvek propadnout na své stabilní místo, tj. pozice, ve které nebude porušena žádná vlastnost min–max haldy.

Při propadávání prvku směrem dolů v Hasseho diagramu se musí v každém kroku porovnat vždy prvky z nejbližší min i max hladiny. Vybere se vždy prvek s nejmenší hodnotou uloženého klíče. V případě, že je vybraný prvek synem právě porovnávaného prvku, procedura propadávání končí, neboť jsme narazili na prvek z opačné hladiny, než ve které se právě nacházíme. Pokud je nejmenší prvek vnukem zkoumaného uzlu, procedura propadávání může i nemusí pokračovat.

Následuje názorný popis celého algoritmu 11 v pseudokódu.

Algoritmus 11 MM Delete-Min

```
i ← 0, min ← h[i], h[i] ← h[last − −]
while A[i] has children do
  m ← index of smallest of the children and grandchildren of A[i]
  if A[m] is a grandchildren of A[i] then
    if A[m] < A[i] then
      swap(i,m)
      if A[m] > A[parent(m)] then
        swap(m,parent(m))
      end if
    else
      break
    end if
  else if A[m] < A[i] then
    swap(i,m)
  else
    break
  end if
end while
return min
```

3.5.3 Operace DELETE-MAX

V případě, že je v haldě uloženo dva a více prvků, je prvek s maximální hodnotou klíče jedním ze synů kořene. Tento prvek je pak z haldy odstraněn a nahrazen opět posledním prvkem v haldě. Následně musí prvek stejně jako u operace `Delete-Min` propadnout na své stabilní místo. Logika propadávání je stejná jako v předchozí operaci, pouze jsou prohozeny relační operátory.

3.5.4 Operace CREATE

Min–max halda může být vytvořena v lineárním čase použitím modifikovaného Floydova algoritmu, který je uveden v [10]. Floydův algoritmus buduje haldu odspodu. Když algoritmus testuje podstrom s kořenem s indexem i, jsou oba podstromy určené tímto kořenem uspořádané. Další krok algo-

ritmu testuje uspořádání mezi kořenem s indexem i a jeho syny. V min–max haldě je třeba navíc zachovat uspořádaní mezi min a max hladinami. Proto je třeba v algoritmu 12 rozlišovat mezi min a max hladinou. Algoritmus 12 vychází z modifikovaného algoritmu 11.

Algoritmus 12 MM Create

```
for i = last to 0 do
  if i is on min level then
    while A[i] has children do
      m ← index of smallest of the children and grandchildren of A[i]
      if A[m] is a grandchildren of A[i] then
        if A[m] < A[i] then
          swap(i,m)
          if A[m] > A[parent(m)] then
            swap(m,parent(m))
          end if
        else
          break
        end if
      else if A[m] < A[i] then
        swap(i,m)
      else
        break
      end if
    end while
  else
    analogy as above but on max levels except that relational operators
    are reversed
  end if
end for
```

3.5.5 Složitost

V [2] je přesně vyčíslena složitost datové struktury založená na datových výměnách a počtu porovnání. Pro jednotlivé operace jsou pak tyto složitosti uvedeny v tabulce 3.1.

Tab. 3.1: Složitost min–max haldy

	Výměny	Porovnání
`Create`	n	$2.15n$
`Insert`	$0.5\log(n+1)$	$\log(\log(n+1))$
`Delete-Min`	$\log(n)$	$1.5\log(n)+\log(\log n)$
`Delete-Max`	$\log(n)$	$1.5\log(n)+\log(\log n)$

4 Realizace testů

Tato kapitola se zabývá podrobným popisem způsobu vzájemného poměřování kvalit či záporů jednotlivých typů prioritních front, které jsme popisovali v předchozí části textu. Zaměříme se zejména na popis typů navržených testů a na způsob vzájemného porovnání struktur prostřednictvím spotřebovaného výpočetního času procesoru a počtu porovnání a výměn prvků uložených v jednotlivých haldách. Výsledky testů včetně jejich grafické reprezentace a podrobné diskuse nad dosaženými údaji pak uvádíme v následující kapitole 5.

4.1 Měřící a testovací nástroje

4.1.1 Testovací prostředí

Realizace všech testů byla provedena na stroji s následující testovací konfigurací:

Hardware

CPU AMD Athlon 64 3000+
MB MSI K8N Neo2 Platinum-54G
2x512MB DDR Kingmax CL2.5 PC433 v dual channel módu

OS

Fedora Core 4 (x86_64)
verze jádra 2.6.13–1

Překladač

g++ 4.0.1 (Red Hat 4.0.1–5) , překlad bez optimalizací kódu

4.1.2 Měření času

Jedním z hlavních porovnávacích kritérií, které jsme použili pro vzájemné poměřování kvalit popisovaných typů prioritních front, je celkový čas spotřebovaný danými operacemi použitými v těchto typech datových struktur.

Samozřejmě jsme okamžitě zavrhli jakýkoliv primitivní manuální způsob měření času. Takto získané hodnoty nemají žádnou vypovídací hodnotu, neboť v sobě bezpochyby započítávají i procesorový čas spotřebovaný jinými než testovacími procesy měření.

Dnešní běžné operační systémy poskytují v zásadě dva poměrně přesné měřící prostředky, prostřednictvím kterých lze měřit čas spotřebovaný konkrétním procesem.

První způsob je čítač cyklů procesoru. Procesor obsahuje zpravidla 64 bitový registr, který se v pravidelných velmi malých intervalech inkrementuje. Čas spotřebovaný procesem by pak odpovídal počtu těchto cyklů od začátku běhu programu. Nevýhodou tohoto způsobu měření je to, že se do výsledné hodnoty započítává i přepínání kontextu procesoru při pravidelném přeplánovávání běžících procesů. Tento registr je tedy globální a započítává se zde i čas strávený procesorem v jiných než jen testovacích procesech. Tento způsob měření se dá použít pouze na velmi krátkých úlohách běžících mimo jiné na systému nepříliš zatíženém během paralelních procesů. Zde dává čítač cyklů velmi přesné výsledky.

Druhý způsob je použití časovače operačního systému. Hodnota časovače je pravidelně aktualizována právě plánovačem operačního systému. Každý proces běžící v operačním systému si pak drží ve zvláštní proměnné hodnotu tohoto časovače. Měření času prostřednictvím časovače operačního systému poskytuje například knihovní funkce `clock()`, která dle dokumentace a manuálových stránek vrací přibližný čas spotřebovaný běžícím procesem. Způsob měření použitím této funkce by pak vypadal jako rozdíl časů získaných opakovaným voláním funkce `clock()` před začátkem a po konci oblasti, ve které bychom prováděli měření.

Vzhledem k tomu, že naše testy nejsou krátkodobé úlohy, rozhodli jsme se k měření času použít časovač operačního systému.

4.2 Typy testů

Každý test je složen z deseti různě dlouhých náhodných posloupností operací `Insert`, `Delete-Min` a `Delete-Max`. Délky těchto posloupností volíme podle typu testu. Obecně začínáme na posloupnosti 1 milión operací s krokem jeden

milión či 500 tisíc. Limitem celkové velikosti testovací posloupnosti operací je volná operační paměť počítače, v našem případě je limitním faktorem posloupnost o délce 10 miliónů Insertů. Pokud bychom chtěli testy měřit na posloupnostech ještě větší délky, začali bychom se potýkat s problémy neustálého odkládání operační paměti na disk.

Každý test pouštíme desetkrát, tedy desetkrát vygenerujeme testovací posloupnosti operací a následně je pouštíme na prioritní fronty, které jsou předmětem testování. Způsob generování náhodné posloupnosti je popisován v kapitole 4.3 a dodatku D. V následujících odstavcích si krátce popíšeme jednotlivé typy testů.

4.2.1 Budování haldy pomocí Insert operací

V tomto testu jsme zkoumali, jak moc je operace vkládání nového prvku použitelná pro budování celé haldy. Vyšli jsme přitom z prázdné haldy, kterou jsme postupně opakovaným vkládáním nových prvků vytvářeli. Vše jsme testovali na poměrně velkých datech. Minimální testovací hranicí pro nás byl jeden milión prvků, maximální hranicí pak prvků deset miliónů. Na tomto testu si tak můžeme názorně představit, jak je v praxi použitelné budování haldy pouze prostřednictvím `Insert` operací. Některé typy popisovaný min–max hald nabízejí možnost budovat haldu odspodu, postupně od listů, v lineárním čase. Takto budovaná halda by měla být vybudována v celkově rychlejším čase. My jsme se však rozhodli tuto operaci neimplementovat, neboť se snažíme měřit kvality a zápory prioritních front pouze na operacích, které jsou společné pro všechny testované datové struktury. Tento způsob budování haldy je univerzálně použitelný ve všech typech námi porovnávaných prioritních front.

4.2.2 Heapsort

Testování prioritních front prostřednictvím heapsortu je dalším testem, který nachází uplatnění v běžné praxi. V našem případě je tento test spojením testů fáze budování haldy prostřednictvím `Insert` operací a následného vyprázdnění haldy prostřednictvím `Delete-Min` či `Delete-Max` operací. Test pouštíme opakovaně na datech různé velikosti. Horní hranicí je měření

heapsortu na pěti miliónech prvcích, dolní hranicí je heapsort na 500 tisíc prvků.

4.2.3 Porovnání Insert operací

Test je spouštěn na již vybudované haldě. Slouží k experimentálnímu ověření rychlosti `Insert` operace. Dosažené výsledky evidentně odpovídají předchozímu testu uvedeném v odstavci 4.2.1.

4.2.4 Porovnání Delete–Min operací

Tento test spouštíme na již předem vybudované haldě. Do výsledného času se tedy nezapočítává fáze budování haldy, neboť nás zajímá pouze náročnost samotné operace `Delete-Min`.

4.2.5 Porovnání Delete–Max operací

Tento test provádíme na již předem naplněné haldě. Do výsledného času se tedy nezapočítává fáze budování haldy, neboť nás zajímá pouze náročnost samotné operace `Delete-Max`. Na první pohled by se mohlo zdát, že testování náročnosti `Delete-Max` operace je zbytečné, neboť máme k dispozici výsledky předchozího měření operace `Delete-Min`. Nemusí tomu tak jednoznačně být. V některých případech námi popisovaných datových struktur je sice operace `Delete-Max` dokonalou symetrií operace `Delete-Min`, v některých případech je však nepatrně složitější a nese si sebou tak i nezanedbatelnou režii, která může ovlivnit negativně výsledek celého měření.

4.2.6 Přidání a odebrání prvků v již vybudované haldě

Tento test má zřejmě nejblíže k běžné praxi. Do již předem vybudované haldy postupně přidává či odebírá prvky. Opět se do výsledku měření nezapočítává fáze budování haldy. Test je opakovaně spouštěn na posloupnosti přidávání, odebírání minima a maxima, jejichž množství jsou v různých poměrech.

4.3 Generování testovacích dat

Testovací data jsou náhodné posloupnosti pevné délky operací `Insert`, `Delete-Min` a `Delete-Max`. Pro potřeby opakovaného generování náhodných dat byla implementována jednoduchá utilita, která vytváří náhodné posloupnosti operací prioritních front. Popis této aplikace uvádíme v dodatku D.

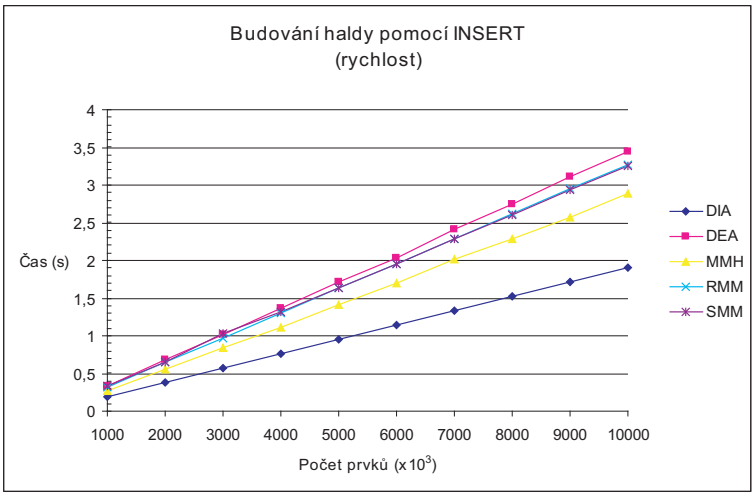
Náhodné hodnoty v testovací posloupnosti generujeme prostřednictvím generátoru náhodných čísel ze standardní knihovny C. Náhodné číslo získáme voláním funkce `rand()`, která vrací dle manuálových stránek pseudo–náhodné celé číslo v rozmezí `0` až `RAND_MAX`.

5 Výsledky měření

U každého typu testu uvádíme vždy tři grafy, z jejichž průběhu můžeme odvodit některé zajímavé závěry. První graf spojnicově zobrazuje pro každou prioritní frontu závislost naměřeného času na délce vstupní posloupnosti operací. Další graf spojnicově znázorňuje průměrný počet provedených porovnání. Pro poslední graf, který zobrazuje průměrný počet provedených výměn, jsme záměrně zvolili sloupcový typ grafu. Tři z pěti testovaných prioritních front zde dosahují velmi podobných výsledků. Pokud bychom naměřené hodnoty zobrazovali ve spojnicovém grafu, výsledné křivky by většinou splývaly.

5.1 Budování haldy postupnými Inserty

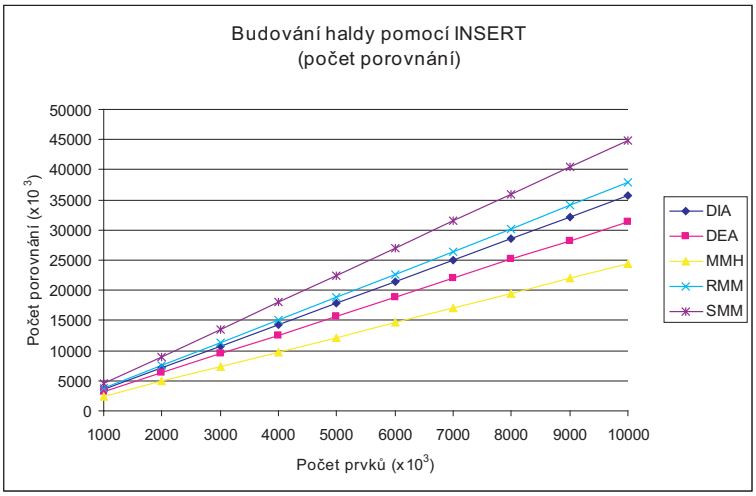
Test jsme postupně provedli na posloupnosti `Insert` operací od jednoho miliónu po deset miliónů vložení s krokem jeden milión prvků. Každý test jsme provedli desetkrát a spočítali průměrně dosažené časy včetně příslušných směrodatných odchylek. Výsledky sepsané do podoby tabulek jsou k nahlédnutí v příloze práce. Na přiložených grafech je pak graficky znázorněn průměrný čas fáze budování haldy prostřednictvím `Insert` operací včetně průměrného počtu porovnání prvků a jejich výměn.



Obr. 5.1: Budování haldy postupnými Inserty (průměrný čas)

Jako nejrychlejší se jeví budování diamantové haldy. Poměrně stejně jsou

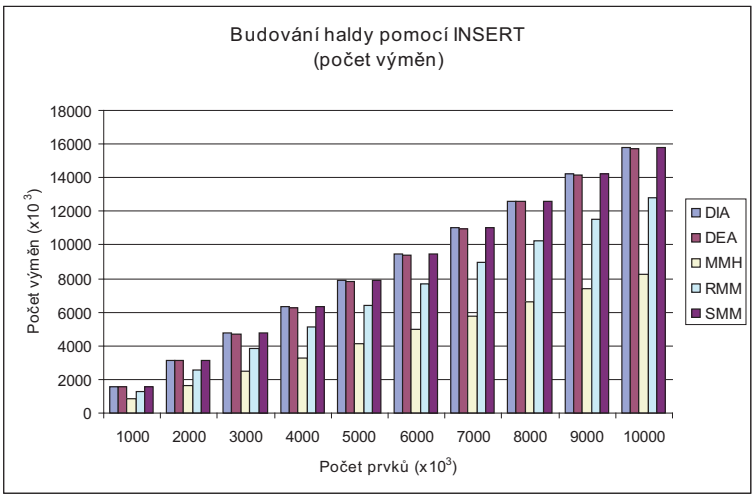
na tom pak symetrická a reflektovaná halda, které si jsou ve fázi budování haldy dost podobné.



Obr. 5.2: Budování haldy postupnými Inserty (průměrný počet porovnání)

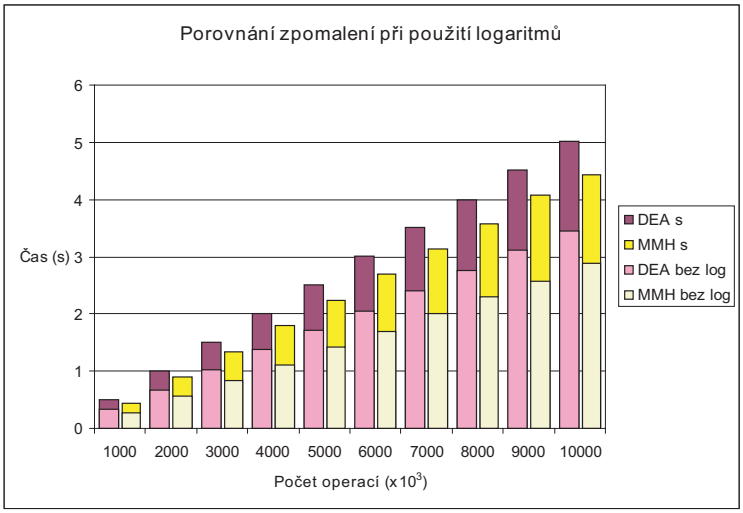
Podobných výsledků dosáhla i min–max halda. Nejvíce ve fázi budování haldy prostřednictvím `Insert` operací zahálí dvoukoncová halda (*deap*). Celkově v porovnání s ostatními jsou však rozdíly zanedbatelné. Tyto dvě prioritní fronty potřebují při operaci `Insert` vědět, do které hladiny prvek vkládají. První implementace pro jednoduchost zjišťovala tuto hladinu prostřednictvím výpočtu přes logaritmy. Tato implementace však byla příliš pomalá. Jelikož naše haldy jsou binární stromy, bylo nasnadě použít bitové operace, resp. bitový posun, pro zjištění hladiny daného prvku. Jak jsme si experimentálně ověřili, výpočet u obou struktur se bez použití logaritmů urychlil v průměru o více než 30%.

Pořadí v čase víceméně odpovídá dosaženým výsledkům v průměrném počtu porovnání. Jedinou výjimku tvoří v umístění v průměrném počtu porovnání diamantová a dvoukoncová halda. *Deap* se v průměrně dosaženém čase umístila na posledním místě, ačkoliv v praxi její operace Insert dosahuje nejmenšího počtu porovnání. Naproti tomu diamantová halda má jednoznačně nejrychlejší Insert operaci, ačkoliv dosahuje průměrného počtu porovnání ve srovnání s ostatními prioritními frontami. Vysvětlení je prosté. Diamantová halda používá k přístupu k jednotlivým prvkům a potažmo i



Obr. 5.3: Budování haldy postupnými Inserty (průměrný počet výměn)

v celé logice výpočtu velmi rychlé bitové operace. Umožňuje to konstrukce struktury vycházející z Hasseho diagramu, který min a max haldu rozděluje na sudé a liché prvky. *Deap* pak ve srovnání s ostatními používá většího počtu poměrně drahé operace výměny prvků.



Obr. 5.4: Porovnání rychlosti použití logaritmů

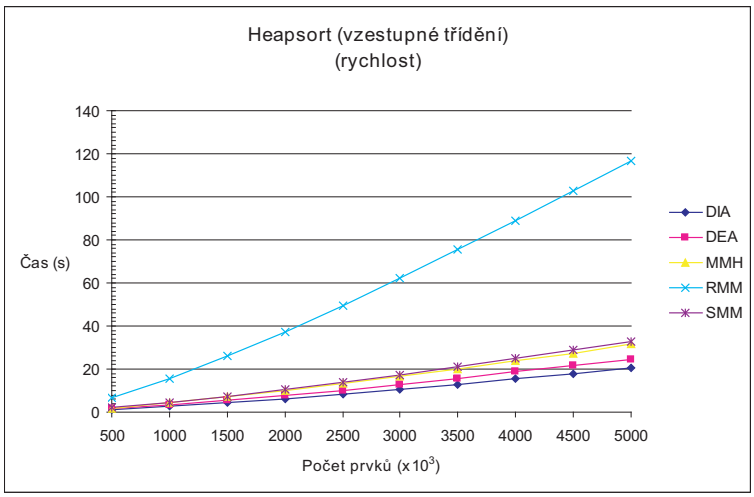
Pokud bychom si vybírali prioritní frontu na základě kritéria rychlosti budování haldy, rozhodně bychom se rozhodli pro diamantovou haldu, která dosahuje velmi dobrých výsledků v tomto typu experimentálního

testu. V další části textu se rovněž dočteme, že tento typ prioritní haldy nedosahuje příznivých výsledků pouze v operaci `Insert`, nýbrž i v operaci `Delete-Min` a `Delete-Max`. O tom ale později. Některé z námi poměřovaných hald však umožňují vybudování haldy v lineárním čase. Kupříkladu dvoukoncová halda je jednou z nich. Jak jsme však uvedli dříve, zaměřili jsme se na měření kvalit prioritních front z pohledu stejných operací. Je zřejmé, že v praxi bychom haldu *deap* konstruovali lineárně a nikoli prostřednictvím `Insert` operací.

5.2 Heapsort

Pokusili jsme se experimentálně ověřit chování popisovaných hald na jednom z nejznámějších třídících algoritmů. Opakovaně jsme třídili deset různě dlouhých posloupností prvků. Nejmenší tříděná posloupnost měla 500 tisíc prvků, největší pak 5 miliónů prvků.

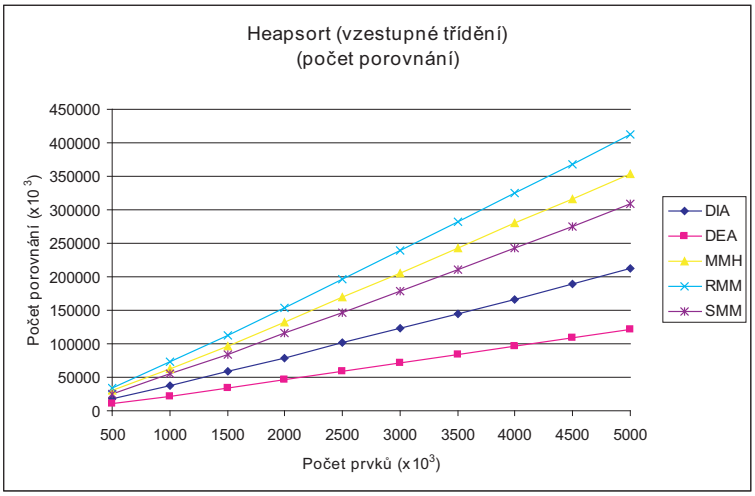
Na přiložených grafech lze názorně vidět, že si opět nejlépe vedla diamantová halda. V tomto testu pak zcela propadla reflektovaná halda. V následující části textu se proto pokusíme odhalit, proč se tak stalo.



Obr. 5.5: Heapsort (průměrný čas)

Samotný heapsort se skládá ze dvou částí, vytvoření haldy a jejího vyprázdnění prostřednictvím operací `Delete-Min` či `Delete-Max`. V předchozí části textu jsme se zabývali měřením fáze budování haldy. Ze srovnání

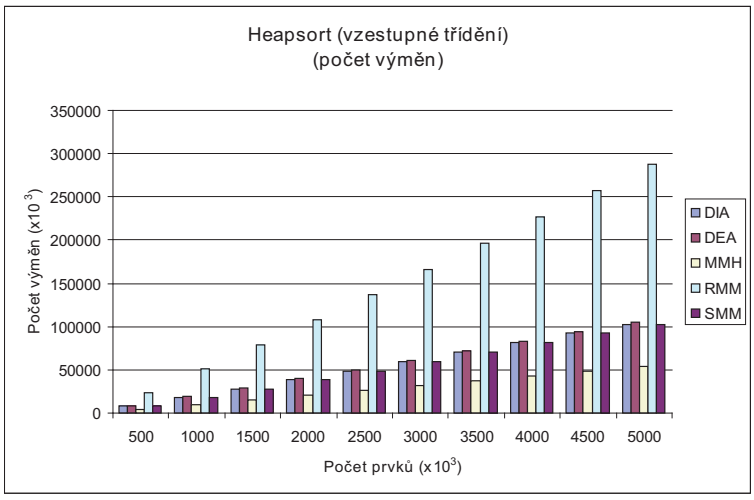
obou grafů je zřejmé, že zpomalení způsobuje fáze vyprázdnění haldy.



Obr. 5.6: Heapsort (průměrný počet porovnání)

Reflektovaná halda se totiž skládá ze dvou oddělených min a max hald. Tyto haldy, respektive jejich prvky, jsou vzájemně propojeny prostřednictvím tzv. mostů. Když proto odstraňujeme minimum či maximum z RMM haldy, je nutné odebrat z druhé haldy korespondující prvek. Takto odebraný prvek musíme do haldy opět vložit. Máme tady minimálně dvojnásobnou režii navíc, která způsobuje největší zpomalení výpočtu. Další zpomalení je způsobeno opakovaným vkládáním prvků do haldy a samozřejmě přepočítáváním mostů. Jisté zrychlení pak spatřujeme v tom, že min a max halda je poloviční velikosti než ostatní testované haldy. Výška haldy je však jen o jedna menší, tudíž ve finále je toto zrychlení zanedbatelné.

Na stejných testovacích datech jsme spustili i heapsort, který třídil data na výstupu sestupně, tj. místo operace `Delete-Min` se pro řazení používala operace `Delete-Max`. Domnívali jsme se, že v některých případech, je použití operace `Delete-Max` dražší než použití operace `Delete-Min`. Praktické testy však tuto domněnku zcela vyvrátily. Dosažené výsledky jsou téměř totožné s naměřenými hodnotami heapsortu řadícího data vzestupně. Neuvádíme zde proto tuto sérii grafů, které jsou k nahlédnutí na přiloženém médiu.



Obr. 5.7: Heapsort (průměrný počet výměn)

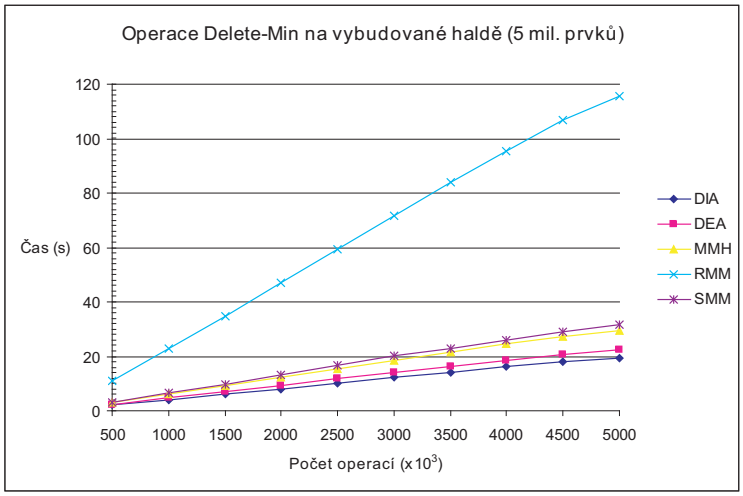
5.3 Porovnání Insert operací

Na již předem vybudované haldě o 5ti miliónech prvcích jsme pouštěli různě dlouhé posloupnosti `Insert` operací. Dosažené výsledky odpovídají výsledkům uvedeným v kapitole 5.1, kde jsme se věnovali fázi budování haldy. Nebudeme je zde proto již dále diskutovat. Všechny grafy a tabulky jsou však k dispozici na přiloženém médiu.

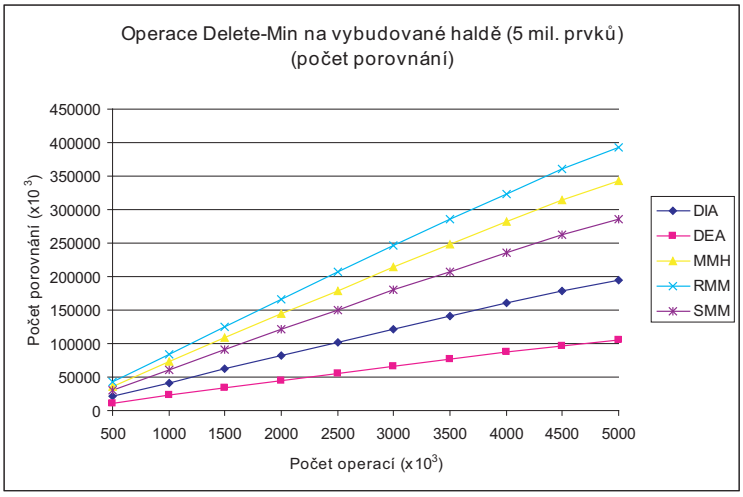
5.4 Porovnání Delete–Min a Delete–Max operací

Testy měření `Delete-Min` operací dopadly podle předpokladů, které jsme získali předchozím testem, kde jsme měřili použitelnost popisovaných prioritních front při třídění dat. Víceméně vyrovnaných výsledků dosáhly dvojice diamantová halda, *deap* a min–max halda se symetrickou haldou. Min–max halda používá ve výsledku více porovnání než halda symetrická. Naproti tomu SMM halda častěji prohazuje prvky. Proto se zde částečně stírají tyto rozdíly a ve výsledku dosahují podobných časů. Výměna je však dražší operace než porovnání a proto je min–max halda mírně rychlejší. Obdobné zdůvodnění můžeme hledat i u porovnání výsledků diamantové haldy a *deapu*. Diamantová halda je však kromě výše uvedeného značně urychlena trikovými operacemi. Jako zcela nepoužitelná se jeví reflektovaná halda z

důvodů zmíněných v kapitole 5.2.

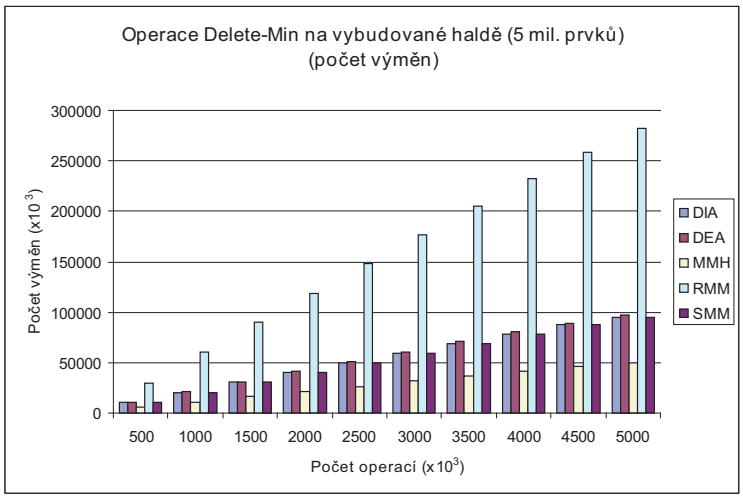


Obr. 5.8: Delete-Min (průměrný čas)



Obr. 5.9: Delete-Min (průměrný počet porovnání)

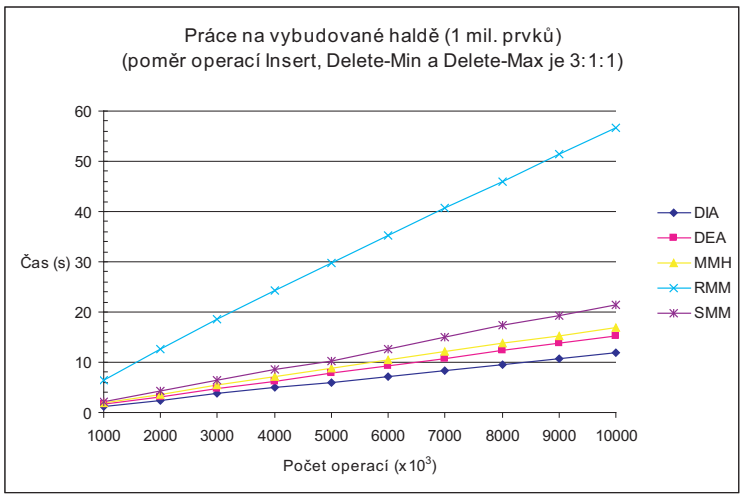
Testy měření rychlosti `Delete-Max` operací dopadly téměř stejně jako předchozí testy `Delete-Min` operace. Grafy a tabulky jsou proto k nahlédnutí na přiloženém médiu a do výsledné práce je zbytečně neuvádíme.



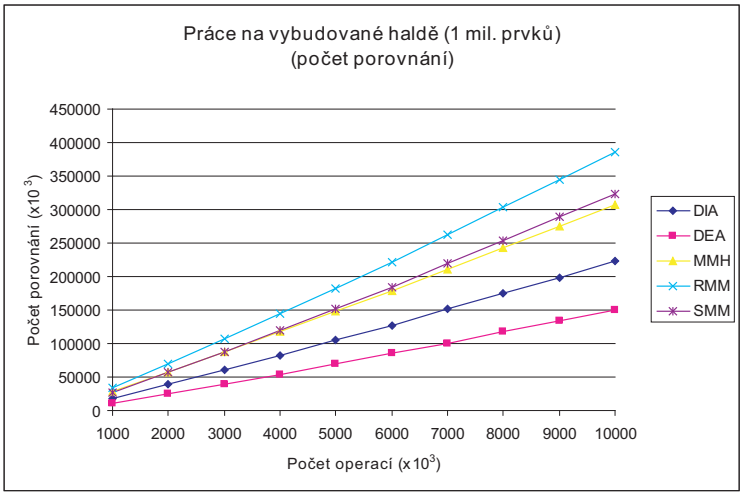
Obr. 5.10: Delete-Min (průměrný počet výměn)

5.5 Přidání a odebrání prvků v již vybudované haldě

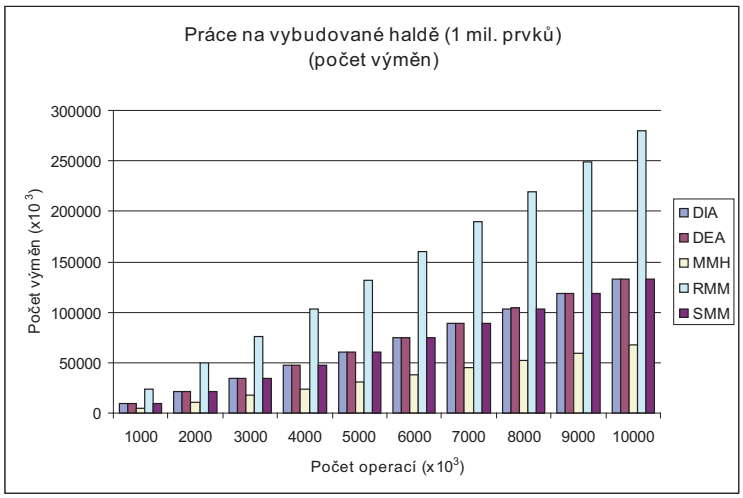
Tento test se nejvíce blíží běžné praxi v použití prioritních front. Na již předem vybudované haldě o jednom či třech miliónech prvcích jsme prováděli posloupnost jednoho až deseti miliónů operací s krokem jeden milión. Poměr mezi množstvím `Insert`, `Delete-Min` a `Delete-Max` operací byl 3:1:1. Výsledky jsou opět jednoznačně nakloněny diamantové haldě. Podle očekávání pak opět propadla RMM halda ze stejných důvodů, jako jsme již uvedli výše. Pro úplnost přikládáme grafy s dosaženými výsledky.



Obr. 5.11: Náhodná posloupnost operací (průměrný čas)

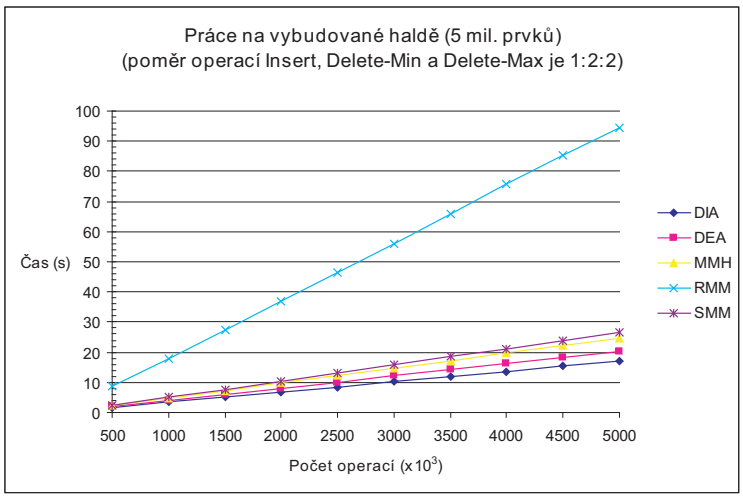


Obr. 5.12: Náhodná posloupnost operací (průměrný počet porovnání)

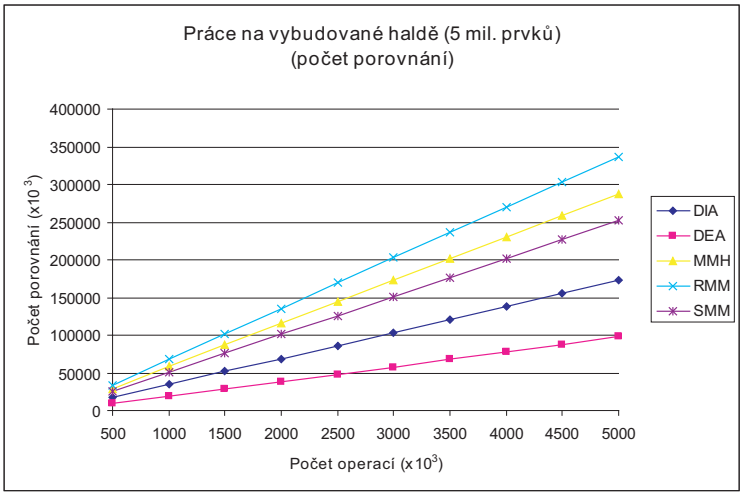


Obr. 5.13: Náhodná posloupnost operací (průměrný počet výměn)

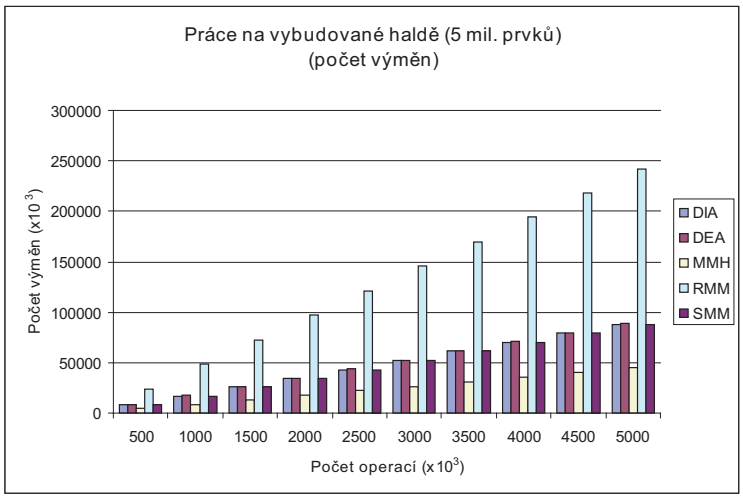
Stejný typ testu jsme provedli na haldě o 5ti miliónech prvcích. Poměr operací `Insert`, `Delete-Min` a `Delete-Max` byl tentokrát ale 1:2:2, tj. zaměřili jsme se na situace, kdy se bude z haldy více odebírat než vkládat. Nejmenší posloupnost operací měla 500 tisíc prvků, největší 5 miliónů prvků. Dosažené výsledky jsou srovnatelné s předchozím testem, který měl poměr mezi operacemi `Insert`, `Delete-Min` a `Delete-Max` roven 3:1:1. Jedinou výjimkou je umístění symetrické haldy před min–max haldou v průměrném počtu porovnání. Toto je způsobeno zejména tím, že v operacích `Delete`, které jsou v tomto testu více zastoupené, vybírá min–max halda nejmenší či největší prvek vždy z dvou hladin, což celkový počet provedených porovnání zvýší.



Obr. 5.14: Náhodná posloupnost operací (průměrný čas)



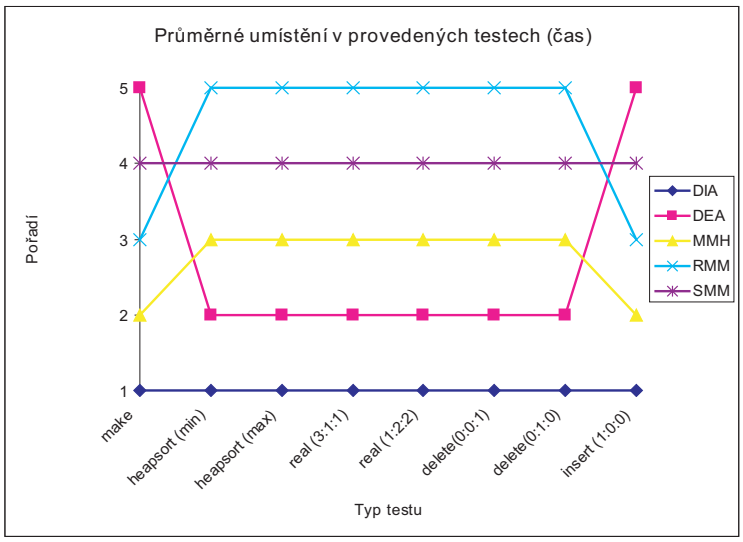
Obr. 5.15: Náhodná posloupnost operací (průměrný počet porovnání)



Obr. 5.16: Náhodná posloupnost operací (průměrný počet výměn)

5.6 Závěrečné zhodnocení

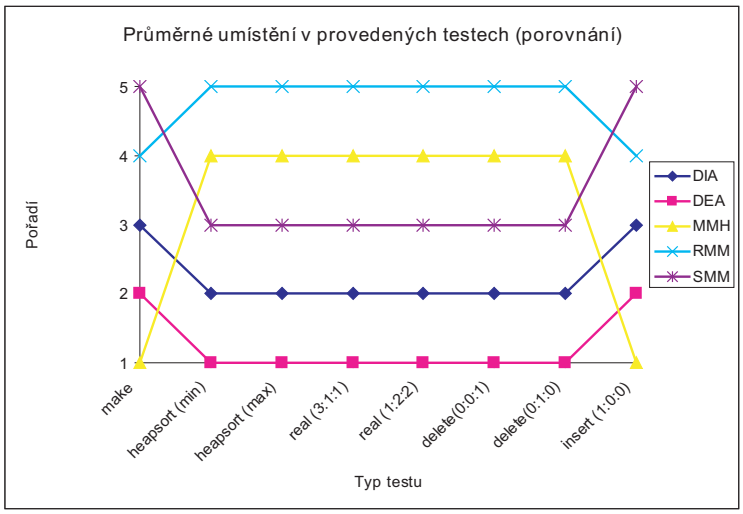
Na závěr kapitoly věnující se porovnávání implementovaných datových struktur uvádíme souhrnný přehled dosažených výsledků.



Obr. 5.17: Průměrné umístění v čase

V množství spotřebovaného procesorového času vyhrála jednoznačně diamantová halda. Dobře si vedla i dvoukoncová halda (*deap*). Ta však propadla v testech zaměřených pouze na rychlost `Insert` operací. Průměrně si

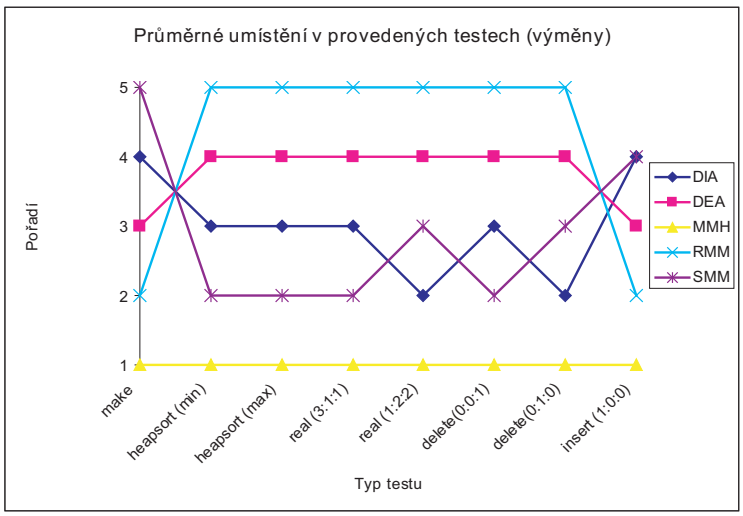
pak vedla min–max halda a symetrická min–max halda. Nejhůře pak dopadla reflektovaná halda i přesto, že má poměrně rychlé `Insert` operace. Doplatila tak na velmi pomalé `Delete-Min` a `Delete-Max` operace.



Obr. 5.18: Průměrné umístění v počtu porovnání

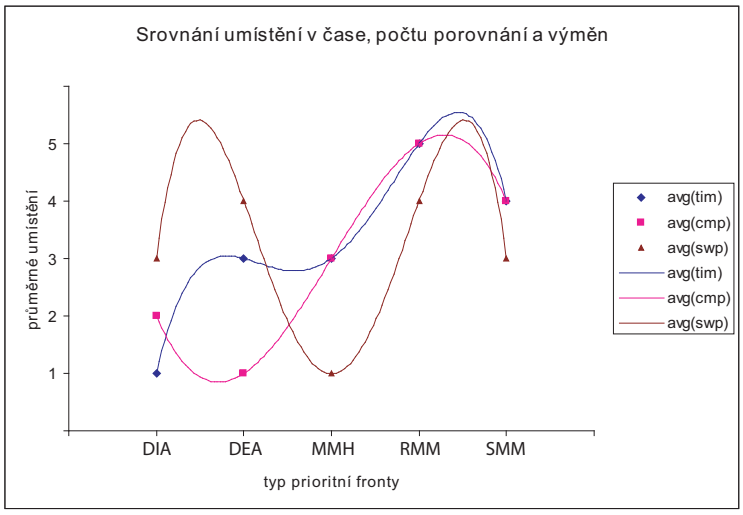
Poměrně zajímavé výsledky dostaneme, pokud navzájem porovnáme dosažené průměrné pořadí datových struktur v čase, počtu porovnání a počtu výměn. Porovnání je k dispozici na grafu 5.20. Pro názornost je průměrným pořadím proložena křivka. V běžném povědomí odborné veřejnosti převládá názor, že datová struktura, které dosahuje nízkých hodnot v počtu porovnání musí být i celkově rychlejší. Tato domněnka se potvrdila pouze z části. Obecně toto samozřejmě nemůžeme tvrdit. Dosažené snížení počtu porovnání je téměř vždy na úkor jistého zesložitění celkového algoritmu. V běžné praxi však tuto režii navíc nemůžeme opomenout. V našem případě dosahovala nejmenšího počtu porovnání dvoukoncová halda, diamantová halda a symetrická halda. Nejvyššího pak reflektovaná halda.

Diamantová halda a *deap* dosáhly velmi slušných výsledků v testech zaměřených na čas. V našem konkrétním případě na základě experimentálního ověření chování struktur na velkých datech tedy můžeme potvrdit, že struktury s menším počtem porovnání jsou většinou i rychlejší. Jedinou výjimku zde tvoří diamantová halda, která se v časech umístila před *deap*,



Obr. 5.19: Průměrné umístění v počtu výměn

ačkoliv používá většího množství porovnání. Můžeme to odůvodnit tím, že diamantová halda používá ve větší míře než ostatní prioritní fronty velké množství nejrůznějších trikových operací, jak jsme již zmínili dříve, většinou velmi rychlých bitových posunů, které celkový algoritmus velmi urychlí. Na opačném konci stojí reflektovaná halda. Ta, jak jsme již uvedli výše, používá spoustu dílčích operací s haldou, které celý algoritmus zesložiťují, a tím i výrazně zpomalují.



Obr. 5.20: Vzájemné porovnání času, výměn a porovnání

6 Závěr

Tato práce podala ucelený přehled pěti datových struktur, které spadají do kategorie oboustranných prioritních front odvozených od datové struktury halda. Provedli jsme implementaci těchto datových struktur a experimentálně ověřili jejich chování na velkých datech. Sestavili jsme proto sérii testů, které jsme postupně pouštěli na jmenované datové struktury. Výsledky měření jsme sepsali do podoby přehledných a názorných grafů, z jejichž průběhů jsme odvodili některé zajímavé závěry.

Vedlejším výsledkem této práce je kromě sady statistických dat popisující kvality a zápory popisovaných datových struktur jejich implementace v programovacím jazyku C++, která je k dispozici na přiloženém médiu. Implementace byla navržena dostatečně obecně pomocí techniky použití šablon. Prioritní fronty tak mohou uchovávat libovolný datový typ, který má korektně předefinován operátor porovnání $<$. Implementace samozřejmě mohla být provedena programátorsky daleko čistěji například prostřednictvím použití polymorfismu. Ten jsme však zavrhli z důvodu nezanedbatelné režie související s metodikou volání virtuálních funkcí, které by výsledné naměřené hodnoty mohly zhoršit či zkreslit.

Podobnou práci přinesli Chong a Sahni v [8]. Ve své studii se věnovali porovnání třech obecných způsobů, jak efektivně implementovat oboustrannou prioritní frontu z jednoduchých front. Vzájemně porovnali struktury založené na totální, duální či listové korespondenci, kdy vedle sebe stojí vždy dvě haldy, jedna min a jedna max. Jejich experimentální závěry pak hovoří jednoznačně ve prospěch listové korespondence. V naší práci jsme se vesměs zabývali prioritními frontami, kterým stačí k uložení dat pouze jedna nějakým způsobem modifikovaná halda. Výjimku zde tvořila právě reflektovaná halda, která se ve většině testů nemohla pochlubit přívětivými výsledky, kromě poměrně velmi svižné `Insert` operace. Ta odpovídá totální korespondenci, popisované Chongem a Sahnim v [8].

Dodatky

A Seznam algoritmů

B Tabulky

V následující kapitole uvádíme některé tabulky naměřených průměrů a jejich směrodatných odchylek u všech provedených testů. Ostatní tabulky jsou k dispozici na přiloženém médiu v adresáři `results/xls`. Záhlaví sloupců mají následující význam:

`heap`	typ oboustranné prioritní fronty
`len`	délka vstupní posloupnosti operací
`avg(t)`	průměrný čas v sekundách
`var(t)`	směrodatná odchylka naměřených časů
`avg(cmp)`	průměrný počet porovnání
`var(cmp)`	směrodatná odchylka počtu porovnání
`avg(swp)`	průměrný počet výměn prvků
`var(swp)`	směrodatná odchylka počtu výměn

V tabulkách jsou použity zkratky s následujícím významem:

`SMM`	symetrická min–max halda
`DIA`	diamantová halda
`DEA`	*deap*, dvoukoncová halda
`RMM`	reflektovaná min–max halda
`MMH`	min–max halda

Tab. B.2: Budování haldy postupnými Inserty

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	1000000	0,191	0,005676462	3577581,7	1714,552553	1577613	1714,171131
	2000000	0,381	0,005676462	7155396,1	1490,03277	3155425,7	1489,905221
	3000000	0,571	0,005676462	10734596,4	2393,821594	4734630,2	2397,084609
	4000000	0,759	0,005676462	14310892,8	2985,562363	6310920,4	2986,906731
	5000000	0,956	0,005163978	17888756,1	3587,475078	7888789,1	3589,545668
	6000000	1,144	0,006992059	21467847,3	3125,182414	9467881,4	3127,443088
	7000000	1,331	0,005676462	25047671,9	3873,010499	11047702	3873,267441
	8000000	1,526	0,00843274	28626274,3	3302,859066	12626305,5	3301,498793
	9000000	1,718	0,006324555	32203171,9	5086,647039	14203206,4	5085,068298
	10000000	1,905	0,005270463	35782808,4	4099,675304	15782843,3	4098,141694

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	1000000	0,331	0,005676462	3139527,2	1398,729566	1572661,1	1563,789589
	2000000	0,676	0,009660918	6279469,3	1733,376349	3145674,2	2209,641037
	3000000	1,014	0,010749677	9439850,8	2565,54598	4690872,5	2665,566992
	4000000	1,372	0,012292726	12559004,8	3501,193631	6291354	3976,269748
	5000000	1,715	0,011785113	15712748,2	3182,863588	7842613,7	3071,356995
	6000000	2,039	0,0166333	18876980,8	2920,756782	9379417,1	2854,463423
	7000000	2,405	0,007071068	22008070,7	3001,717399	10973216,5	3682,838798
	8000000	2,751	0,032128215	25121635	2488,383902	12587214,8	3042,485599
	9000000	3,106	0,025905812	28264409,2	4501,324072	14152065,4	4580,905422
	10000000	3,443	0,023118055	31430001,7	3129,234804	15689585,5	4220,482338

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	1000000	0,322	0,00421637	3780861,8	1474,900358	1280900,7	1474,533075
	2000000	0,649	0,007378648	7561935,7	1735,330007	2561972,5	1735,992528
	3000000	0,975	0,005270463	11344152,4	2564,354075	3844194,3	2568,591229
	4000000	1,303	0,004830459	15123163	2779,316503	5123197,3	2780,460955
	5000000	1,629	0,00875595	18905282	2745,11716	6405322,2	2749,109626
	6000000	1,958	0,00421637	22688115,6	3373,697681	7688157,8	3375,571484
	7000000	2,284	0,005163978	26470489,5	2511,858376	8970526,8	2511,848934
	8000000	2,613	0,006749486	30252228,2	2660,749844	10252265,6	2660,863152
	9000000	2,945	0,008498366	34032143,1	3942,988811	11532186,7	3940,037593
	10000000	3,276	0,012649111	37815827,3	3619,378954	12815870,5	3621,383997

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	1000000	0,326	0,005163978	4491172,1	3167,832994	1577613	1714,171131
	2000000	0,65	0,004714045	8983299,2	2839,178199	3155425,7	1489,905221
	3000000	1,027	0,067503086	13477469,5	3898,217634	4734630,2	2397,084609
	4000000	1,315	0,041965594	17966167,9	4999,818607	6310920,4	2986,906731
	5000000	1,629	0,009944289	22459308,9	5554,625409	7888789,1	3589,545668
	6000000	1,954	0,006992059	26951943,5	6641,121663	9467881,4	3127,443088
	7000000	2,281	0,00875595	31449375,7	6134,656289	11047702	3873,267441
	8000000	2,611	0,005676462	35940057,7	7187,854371	12626305,5	3301,498793
	9000000	2,934	0,005163978	40430976,4	7245,680816	14203206,4	5085,068298
	10000000	3,259	0,009944289	44923075,6	7099,793208	15782843,3	4098,141694

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	1000000	0,275	0,007071068	2438446,4	727,2835608	824342	814,4877054
	2000000	0,555	0,005270463	4877375,2	776,3900509	1648578,1	987,4067551
	3000000	0,838	0,00421637	7316382,5	853,7664591	2473171,7	1186,850927
	4000000	1,117	0,006749486	9754398,1	1438,717365	3296413	2187,828756
	5000000	1,413	0,021108187	12193778,8	1392,987341	4121268	1621,686776
	6000000	1,703	0,025407785	14631751,7	2132,449351	4945791,6	2002,832228
	7000000	2,011	0,034140234	17072430,1	2022,070196	5771591,6	1663,273025
	8000000	2,291	0,046055522	19511091,1	1425,251046	6596446,2	1715,005915
	9000000	2,577	0,029078438	21949204,2	1722,329288	7419896,2	2370,934593
	10000000	2,889	0,055667665	24387985,1	2650,241182	8243230,9	2439,206631

Tab. B.3: Heapsort - vzestupné třídění

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	1000000	1,151	0,007378648	17865254,8	1077,839794	8640884,2	1004,518547
	2000000	2,625	0,007071068	37730358,5	2095,388654	18281736,3	1896,320065
	3000000	4,382	0,010327956	58345094,4	2445,365058	28297389,8	2185,451532
	4000000	6,366	0,021186998	79460200	1946,149989	38562682,9	1869,255375
	5000000	8,461	0,018529256	100895533	3025,683587	48989471,8	2595,753447
	6000000	10,68	0,020439613	122687655	3377,778422	59593376	3334,572536
	7000000	13,01	0,043320511	144728400	1665,212979	70322909,2	2133,642363
	8000000	15,45	0,034058773	166922518	4945,47576	81127256,1	4140,37949
	9000000	18,01	0,045276926	189249034	3586,661128	91998474,6	2989,702297
	10000000	20,57	0,041486276	211792824	3311,402591	102980710	3190,436222

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	1000000	1,411	0,005676462	10515571,8	815,2121469	8860760	829,1000476
	2000000	3,265	0,068839588	22031101,1	970,5192711	18721211,9	1202,64897
	3000000	5,461	0,042804465	33919224,6	1802,335226	28927356,9	1600,040934
	4000000	7,573	0,036530049	46061384,9	1416,551717	39441769,9	2176,738
	5000000	10,16	0,092068815	58320887,8	2254,072798	50068800,7	1788,723757
	6000000	12,66	0,092430154	70837525,3	1878,413163	60854305,3	2108,761514
	7000000	15,72	0,125437014	83471473,8	2659,036367	71810886,4	3215,796642
	8000000	18,62	0,077143445	96124439,3	2824,361365	82884928,9	3135,252868
	9000000	21,45	0,116809436	108809122	2884,648488	93978011,6	2622,308915
	10000000	24,38	0,106228266	121642730	2903,600916	105138651	3085,139012

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	1000000	6,574	0,006992059	34563663	2161,679028	23870260,7	1868,909667
	2000000	15,59	0,020976177	73127671,2	2690,922799	50740134,4	2215,603916
	3000000	26,05	0,031710496	113206886	2246,472049	78720411,9	2569,212783
	4000000	37,46	0,077056977	154255822	2609,565952	107480340	2801,121809
	5000000	49,57	0,046951512	196006201	3898,536963	136707545	3591,512632
	6000000	62,19	0,072755298	238413948	4606,763927	166439626	3761,305815
	7000000	75,3	0,057387571	281289018	3574,761443	196557762	4267,390263
	8000000	88,83	0,133732735	324515694	4480,809854	226966604	4335,980895
	9000000	102,6	0,094991228	368066025	6857,400787	257556342	6275,048813
	10000000	116,7	0,104056609	412014439	5911,255055	288417435	5018,938012

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	1000000	1,967	0,004830459	25845893,7	1032,404965	8640876,2	990,8469778
	2000000	4,455	0,010801234	54690966,7	3348,259616	18281887,1	2162,372229
	3000000	7,324	0,012649111	84661309,2	2500,39005	28297511,2	2451,325981
	4000000	10,57	0,027507575	115380121	2567,436943	38562524,5	1903,448056
	5000000	13,85	0,029230882	146579592	5025,030322	48989444,9	2317,276392
	6000000	17,34	0,023664319	178314808	3364,797582	59593433,7	2740,672061
	7000000	21,03	0,040606513	210425339	3609,704305	70322796,9	2866,734164
	8000000	24,81	0,051650535	242763102	3996,530024	81127320,6	3309,941127
	9000000	28,75	0,059851668	275300340	5161,330902	91998462	3148,616204
	10000000	32,71	0,035605867	308160903	5917,890104	102980511	3433,284116

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	1000000	1,864	0,012649111	29695193,1	1290,882855	4554162,8	1118,15938
	2000000	4,292	0,03190263	62894840,9	1183,998259	9682566,2	1699,75827
	3000000	7,097	0,057551909	97187964,5	1908,321383	14895760,7	1256,214158
	4000000	10,05	0,032058973	132778631	1441,139842	20215629,5	1282,767256
	5000000	13,28	0,033681515	169088694	3711,778317	25724110,3	1588,8206
	6000000	16,62	0,033681515	205712961	1822,824576	31336316,8	1652,69005
	7000000	20,13	0,047422451	242569749	3623,972749	37012477,4	1844,865078
	8000000	23,75	0,045472825	279580331	2629,978327	42729761,3	930,8166128
	9000000	27,48	0,025841397	316709219	2873,563581	48416724,4	2087,246469
	10000000	31,39	0,042804465	354314767	3185,642729	54105671,1	2423,338031

Tab. B.4: Heapsort - sestupné třídění

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	1000000	1,135	0,007071068	17865417,4	1826,643078	8641009,8	1652,160256
	2000000	2,603	0,006749486	37730762,7	2434,403511	18282199,7	2312,613819
	3000000	4,349	0,007378648	58345459,8	1990,996892	28298056,2	1868,957035
	4000000	6,295	0,014337209	79460929,9	1788,529902	38563495,8	1534,253115
	5000000	8,396	0,022705848	100894378	2461,180015	48988216	2410,156242
	6000000	10,58	0,019888579	122688100	4022,696724	59593654,1	3584,510425
	7000000	12,92	0,035276684	144727983	4254,722585	70321777,6	3514,950298
	8000000	15,33	0,020027759	166923508	4773,069393	81128972,6	4280,112361
	9000000	17,82	0,026853512	189248531	3564,25763	91998213,8	4068,845395
	10000000	20,37	0,025905812	211785869	3562,174642	102974524	3658,608564

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	1000000	1,393	0,011595018	10518637,5	1350,253412	8681781,9	1575,739717
	2000000	3,141	0,02514403	22037930,5	1686,083117	18363678,6	1892,173659
	3000000	5,19	0,042426407	33928897,8	1158,197911	28443513,3	1204,054728
	4000000	7,611	0,096200023	46074408,8	1189,567503	38726748,2	1747,79962
	5000000	9,957	0,054375239	58340345,4	2369,035537	49265193,7	2672,364498
	6000000	12,64	0,142968528	70857324,8	3202,63606	59887575,8	3337,28888
	7000000	15,18	0,180151171	83451507	3202,738342	70556481,2	3438,633152
	8000000	17,86	0,065319726	96152466,1	2729,796999	81457494,2	2871,674146
	9000000	20,75	0,110176626	108862807	4869,521652	92477054,3	5160,360798
	10000000	23,99	0,082704293	121681200	3791,297866	103530841	4099,11325

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	1000000	6,616	0,012649111	34563473,9	1729,909018	23870097,2	1825,12044
	2000000	15,71	0,027568098	73128261,5	2460,69499	50740163,5	2569,409108
	3000000	26,19	0,041912607	113206950	2865,0776	78720120,4	2220,772798
	4000000	37,63	0,053551637	154257267	2692,379388	107482319	3844,052094
	5000000	49,87	0,090896767	196004381	4133,143169	136705162	4700,13625
	6000000	62,52	0,071250731	238413486	2601,379346	166439473	2867,605149
	7000000	75,78	0,07748835	281288357	3082,321052	196555642	3025,036125
	8000000	89,29	0,086564042	324517040	4160,675287	226965475	4560,105146
	9000000	103,1	0,108837289	368065451	6930,551509	257554877	6420,929086
	10000000	117,3	0,062182527	412009182	4070,341483	288412382	3610,643248

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	1000000	1,924	0,00843274	25845870,5	1991,346571	8641009,8	1652,160256
	2000000	4,367	0,012516656	54691510,1	2343,902133	18282199,7	2312,613819
	3000000	7,194	0,013498971	84659448,5	1953,431184	28298056,2	1868,957035
	4000000	10,41	0,030713732	115381253	2263,666838	38563495,8	1534,253115
	5000000	13,62	0,030477679	146578061	4206,680607	48988216	2410,156242
	6000000	17,06	0,036530049	178317848	5370,228048	59593654,1	3584,510425
	7000000	20,71	0,04794673	210424094	5807,068816	70321777,6	3514,950298
	8000000	24,46	0,08617811	242765158	7340,452212	81128972,6	4280,112361
	9000000	28,29	0,07007932	275300692	5829,541926	91998213,8	4068,845395
	10000000	32,24	0,022827858	308155901	6594,02967	102974524	3658,608564

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	1000000	1,791	0,003162278	28446912,4	956,4299591	4341154,7	726,8100852
	2000000	4,072	0,006324555	60390260,1	1849,496535	9108853,7	1198,391241
	3000000	6,719	0,00875595	93857626,1	1402,543598	14168848,5	1434,067041
	4000000	9,641	0,023781412	127790218	2287,031431	19364517,1	1588,809649
	5000000	12,76	0,017638342	162156905	2457,548103	24552276,5	2163,375821
	6000000	16,04	0,028205594	197374819	2788,600145	29791228,9	1447,832591
	7000000	19,48	0,019321836	233221348	2480,477071	35086432,3	1687,881647
	8000000	23,03	0,034009803	269560525	4625,007014	40433710,4	2175,525541
	9000000	26,67	0,036040101	306270573	2769,31248	45893553,2	3011,130862
	10000000	30,33	0,042018514	343176061	2527,278563	51446926,2	2128,939423

Tab. B.5: Přidání a odebrání prvků v haldě o 1 mil. prvků

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	1000000	1,256	0,005163978	18491325,8	7354,430045	9709958,6	7824,156968
	2000000	2,482	0,007888106	38969175,9	9593,125634	21353479,5	10495,86425
	3000000	3,696	0,00843274	60365825,3	9306,723019	33904047,3	10386,97313
	4000000	4,885	0,008498366	82300276,1	9304,05525	46991314	11131,43028
	5000000	6,054	0,006992059	104572038	16772,11627	60412508,4	19131,5374
	6000000	7,252	0,020439613	127565003	23263,18722	74398128,8	28402,12579
	7000000	8,399	0,00875595	151207671	15069,6489	88857513,1	18068,64064
	8000000	9,602	0,023475756	174996652	19426,90346	103457062	26704,39812
	9000000	10,79	0,022632327	198891380	24771,759	118137774	31252,43469
	10000000	11,87	0,039384148	222926503	24802,87135	132981393	30606,32465

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	1000000	1,595	0,007071068	11470268,9	7388,523735	9815235,1	7713,939409
	2000000	3,156	0,00843274	24802070,2	9410,923816	21441129,8	9938,30296
	3000000	4,704	0,006992059	39028543,5	8893,721159	33967974,6	10001,52749
	4000000	6,251	0,014491377	53945353,6	10297,43521	47275662,1	10061,30541
	5000000	7,774	0,013498971	69182821,4	17463,61335	60936161,8	16903,4986
	6000000	9,283	0,013374935	84840689,9	27313,06243	74982194,6	27511,87073
	7000000	10,79	0,008232726	100840256	17059,78186	89326711,5	16815,27097
	8000000	12,28	0,013703203	116982388	26493,71727	103804970	27714,20958
	9000000	13,77	0,011547005	133205564	31889,03344	118357036	32314,17871
	10000000	15,21	0,023094011	149593641	30135,47698	133073865	32081,55525

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	1000000	6,418	0,016865481	33531158,8	8174,378682	23454992	8305,118997
	2000000	12,58	0,017126977	69482794,1	7644,198504	49229776,9	8124,633441
	3000000	18,55	0,015634719	106625995	9117,59568	76097119,6	7930,67247
	4000000	24,2	0,018287822	144476123	7588,769583	103596722	7794,214495
	5000000	29,68	0,022632327	182810018	15765,93445	131499869	15455,06945
	6000000	35,14	0,076194196	222240587	22101,502	160274883	20312,38103
	7000000	40,64	0,116599981	262736065	14439,22603	189879534	14149,30905
	8000000	45,95	0,064773108	303458132	19395,78828	219666251	17233,00754
	9000000	51,31	0,134808506	344383773	26947,96652	249604579	23399,18937
	10000000	56,55	0,193531508	385464395	25179,58769	279676403	22970,30006

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	1000000	2,097	0,050122073	26634780,8	11945,23209	9709963,3	7830,777243
	2000000	4,402	0,189490545	56253557,1	15814,78175	21353469	10506,61714
	3000000	6,401	0,361338561	87254532,3	18175,95393	33904047,4	10374,52753
	4000000	8,567	0,348617906	119064173	17409,45686	46991318,2	11146,17021
	5000000	10,34	0,155809713	151392994	28381,00119	60412466,4	19089,42384
	6000000	12,58	0,216692306	184789343	33995,08283	74398028	28379,66754
	7000000	14,92	0,339417802	219152387	14463,6364	88857488,9	18096,93509
	8000000	17,3	0,789514619	253796476	29558,79286	103457055	26665,55734
	9000000	19,36	0,72670107	288554320	47499,50633	118137777	31227,15682
	10000000	21,5	1,472257903	323474019	46962,33152	132981297	30626,30214

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	1000000	1,849	0,007378648	28138905,1	3810,724645	4969918,9	3940,86055
	2000000	3,649	0,017288403	57549629,3	3587,793691	10915291,2	5130,925535
	3000000	5,396	0,024129281	87504305,4	2883,460236	17305758,4	4967,630627
	4000000	7,083	0,009486833	117751365	2295,072984	23951071,4	5284,629772
	5000000	8,739	0,009944289	148177627	5242,938087	30748589,6	9343,277265
	6000000	10,4	0,009944289	179361653	12259,1919	37818093,5	13719,58934
	7000000	12,06	0,008232726	211291559	10251,55016	45108927,9	8819,508886
	8000000	13,71	0,012516656	243311145	7193,068769	52460070,7	12348,46409
	9000000	15,34	0,015811388	275429774	8911,272998	59843624,1	15201,11737
	10000000	16,92	0,025582112	307571768	10431,41137	67302409	15093,38467

Tab. B.6: Přidání a odebrání prvků v haldě o 3 mil.prvků

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	1000000	1,58	0,008164966	18796880,3	4645,893026	9393603,5	4959,48022
	2000000	3,125	0,007071068	38680871,7	6170,485557	19899101,6	6571,448905
	3000000	4,637	0,004830459	59334821,3	8152,921917	31193908	9190,161865
	4000000	6,169	0,011005049	80569265,1	8261,622863	43085304,4	9338,794036
	5000000	7,59	0,00942809	102265856	14019,78718	55449080,8	15683,44368
	6000000	9,094	0,010749677	124316505	16536,76847	68171992,9	19675,63693
	7000000	10,54	0,029363621	147361321	12318,79675	81663320,9	13904,70483
	8000000	11,98	0,037785947	170697236	15153,76792	95430410,1	19708,14647
	9000000	13,37	0,00942809	194266906	15930,45544	109409102	19374,18432
	10000000	14,8	0,013984118	218060878	16280,5464	123621219	20311,96904

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	1000000	1,983	0,004830459	11305117,7	4696,798533	9509092,4	4631,016596
	2000000	3,93	0,006666667	23692506,6	6283,524121	20209015,8	5762,231759
	3000000	5,849	0,007378648	36847139,7	9094,841335	31728140,7	8476,542967
	4000000	7,741	0,003162278	50584860,8	10876,20563	43862523,2	10393,86787
	5000000	9,591	0,005676462	64782821,2	15331,17948	56482200,5	15014,90517
	6000000	11,45	0,009189366	79320775,7	21210,61623	69451419,9	20811,00123
	7000000	13,25	0,009189366	94412368,3	16114,64152	82898021,9	16378,24839
	8000000	15,08	0,006666667	109782042	20788,11926	96609676,1	21269,68335
	9000000	16,87	0,009718253	125357078	22126,73947	110516564	21935,49718
	10000000	18,64	0,014142136	141172990	20444,6453	124659361	21432,04605

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	1000000	8,061	0,009944289	34956255,1	5040,404822	24251727,7	5382,871622
	2000000	16,12	0,011005049	71113604,9	7287,785694	49748985,6	7273,217631
	3000000	24,02	0,024855136	108145517	9234,910087	76131017,4	8559,02744
	4000000	31,72	0,020027759	145847705	7360,418584	103176080	8310,739662
	5000000	39,23	0,056813535	184107250	13957,6558	130755355	13664,40105
	6000000	46,65	0,093118801	222789937	15449,44693	158730550	16091,5752
	7000000	53,82	0,116852233	262926146	16942,77014	187880221	14672,57026
	8000000	61,18	0,116242085	303463528	12444,81641	217376775	11472,08893
	9000000	68,15	0,151745401	344349439	20070,12117	247167042	15485,11712
	10000000	75,32	0,215241466	385502010	11581,85693	277198920	13513,84484

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	1000000	2,503	0,054375239	27087901,8	6978,437947	9393611,3	4954,57248
	2000000	4,921	0,098820826	55808594,6	10006,91687	19899091,3	6558,486581
	3000000	7,417	0,157201216	85686137,5	14678,87017	31193919,3	9183,201524
	4000000	9,881	0,116184719	116435273	12005,19755	43085302,7	9366,521803
	5000000	12,52	0,28190621	147891866	22112,98571	55449088,4	15688,4729
	6000000	14,99	0,33006565	179853471	25811,95596	68172049,3	19618,82421
	7000000	17,65	0,575843343	213319419	17755,47323	81663364,8	13970,47609
	8000000	19,5	0,122401888	247263677	25836,03724	95430428,4	19717,75085
	9000000	22,08	0,199836044	281531933	30828,80794	109409174	19399,89954
	10000000	25,25	0,727094675	316081527	30969,72298	123621245	20287,1504

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	1000000	2,298	0,006324555	29751226,3	1888,513175	4801442,2	2476,7296
	2000000	4,537	0,006749486	60095937,7	2552,560373	10171926,3	3100,755286
	3000000	6,734	0,006992059	90848992	3551,181243	15943572,9	5107,110848
	4000000	8,894	0,006992059	121901569	3193,759243	22016149,4	4613,973154
	5000000	10,99	0,013333333	153190140	4559,297364	28324186,8	7938,991187
	6000000	13,1	0,022010099	184671302	11558,92884	34810776,1	9675,916143
	7000000	15,15	0,008164966	217220458	8271,089852	41674120,2	7824,440481
	8000000	17,24	0,012472191	249957336	6368,126099	48668355,2	10048,12687
	9000000	19,25	0,010593499	282867837	9897,47778	55766725,6	9026,157731
	10000000	21,28	0,015951315	315878415	8410,227329	62979191,4	10226,08788

Tab. B.7: Insert, Delete–Min, Delete–Max v poměru 1:2:2

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	500000	2,043	0,004830459	9434181,2	1441,194859	8572463,8	1407,269365
	1000000	4,069	0,003162278	18933425,3	3529,307331	17231061,3	3664,853553
	1500000	6,031	0,003162278	28502783,2	3684,493718	25982806	3532,551045
	2000000	8,082	0,00421637	38180965	2214,552225	34815029,4	2141,498551
	2500000	10,12	0,007071068	47978960,2	6135,318498	43718602,7	6548,512198
	3000000	12,11	0,005676462	57892569,9	6358,242829	52682150,8	6074,528071
	3500000	14,16	0,004714045	67873599,1	5086,278763	61697541,4	4914,091963
	4000000	16,21	0,007071068	77838012,7	5250,389616	70740629,3	5490,346337
	4500000	18,22	0,00942809	87828368,3	9941,684611	79858464,5	10200,70068
	5000000	20,16	0,039567102	97786032,9	15366,90501	88986524,1	15689,56146

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	500000	1,696	0,005163978	17176870,7	1669,06794	8488680	1677,314785
	1000000	3,39	0	34396712,2	3986,833269	17074537,4	4125,881622
	1500000	5,037	0,004830459	51643286,9	3717,113978	25744823,9	4008,061944
	2000000	6,769	0,007378648	68873144,3	2616,421732	34471020,9	2717,393009
	2500000	8,483	0,006749486	86116463,4	5587,919195	43272135,2	6120,103518
	3000000	10,18	0,00843274	103373313	6182,228422	52146242,4	6657,735855
	3500000	11,92	0,007378648	120654665	3890,575691	61101124,2	4287,092716
	4000000	13,68	0,011785113	137927001	4191,535387	70100905,8	4676,008359
	4500000	15,42	0,013540064	155231695	9464,144376	79190394,8	10753,60457
	5000000	17,1	0,012649111	172444723	14697,12141	88269384,9	16628,65712

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	500000	8,871	0,011005049	33945572,6	1728,405829	24192040,5	1727,387253
	1000000	18	0,019888579	67877173,6	3880,149029	48443429,5	3763,750501
	1500000	27,26	0,013374935	101767819	3224,171434	72722785,9	3844,242114
	2000000	36,8	0,030110906	135552343	2949,559864	96990281,5	3457,896286
	2500000	46,49	0,033399933	169271151	5012,145743	121275354	6183,668257
	3000000	56,12	0,056920998	202927187	6967,130001	145564498	6712,859066
	3500000	65,93	0,059113826	236530649	6179,58318	169854801	6519,861413
	4000000	75,62	0,058461763	270033648	3974,673769	194088930	4259,173754
	4500000	85,23	0,102464086	303460109	8602,696551	218291633	9678,722671
	5000000	94,56	0,130996183	336621136	12169,27939	242298680	16106,91466

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	500000	2,574	0,011737878	25120288,8	2641,558707	8488675,2	1680,948132
	1000000	5,142	0,019888579	50329461,7	5608,742304	17074538,1	4117,046175
	1500000	7,666	0,024129281	75598537,2	4766,001138	25744831,8	3997,09872
	2000000	10,37	0,044284434	100848655	4823,939331	34471017,8	2720,497863
	2500000	13,16	0,064472216	126124459	9889,980657	43272141,5	6057,871009
	3000000	15,82	0,092309865	151421950	10281,31229	52146250,6	6640,628203
	3500000	18,51	0,142999611	176758963	7413,015617	61101221,8	4304,699107
	4000000	21,16	0,215625705	202078311	6791,51714	70100893,3	4633,078747
	4500000	23,78	0,12580408	227462622	14091,26598	79190372,9	10788,55872
	5000000	26,58	0,029135698	252682489	22960,60766	88269345	16650,74382

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	500000	2,423	0,009486833	29195627,9	743,8734137	4338757,7	704,1107946
	1000000	4,856	0,013498971	58304790,6	2020,070417	8722860,4	2252,513224
	1500000	7,219	0,007378648	87303153,2	1631,312818	13151437	2119,369141
	2000000	9,737	0,021628171	116126779	2388,86747	17610643,4	1646,02877
	2500000	12,22	0,021499354	144833095	2781,790449	22108877,7	3377,013113
	3000000	14,68	0,025905812	173434871	2433,526725	26648390,5	3312,191529
	3500000	17,22	0,021705094	201942120	2606,33714	31231242,4	2723,253618
	4000000	19,79	0,029981476	230332268	2706,617988	35838065,2	2282,170059
	4500000	22,32	0,04351245	258603452	3885,031051	40490767,5	5113,303156
	5000000	24,78	0,036878178	286585167	6125,770737	45138408,2	8082,590382

Tab. B.8: Insert operace na haldě o 5ti mil. prvcích

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	500000	0,188	0,006324555	1582816,3	642,2189052	768745,9	582,9555825
	1000000	0,377	0,008232726	3164403	1227,064881	1536437,3	1300,613022
	1500000	0,568	0,006324555	4735305,5	1457,084399	2320217,5	1433,651999
	2000000	0,764	0,005163978	6291902,5	1244,998104	3126462,5	1257,054162
	2500000	0,961	0,003162278	7847985,9	2157,266915	3933108,7	2635,486421
	3000000	1,156	0,006992059	9406476,1	1863,666366	4741207,7	2467,784479
	3500000	1,35	0,006666667	10968438,1	2515,767365	5539491,4	3116,220653
	4000000	1,54	0,006666667	12549437,2	3313,299859	6306900,8	3234,003635
	4500000	1,73	0,006666667	14130329	2663,945111	7073977	2421,792679
	5000000	1,921	0,003162278	15715133	2671,99738	7844401	2246,716122

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	500000	0,096	0,005163978	1789729,6	535,5276111	789729,7	535,6369106
	1000000	0,196	0,006992059	3578332,4	1061,515081	1578332,7	1061,665782
	1500000	0,286	0,005163978	5367849,4	1125,869067	2367849,9	1126,212182
	2000000	0,387	0,006749486	7155357,8	1309,129295	3155358,1	1309,221775
	2500000	0,478	0,006324555	8943270,7	1673,664048	3943271,8	1673,567301
	3000000	0,587	0,021108187	10733660,5	1418,935771	4733661,7	1418,577225
	3500000	0,671	0,005676462	12521111,9	1685,472532	5521112,7	1685,270044
	4000000	0,774	0,024129281	14310511,9	3097,378298	6310512,7	3097,11669
	4500000	0,865	0,005270463	16098853,7	2089,644949	7098855	2089,261433
	5000000	0,962	0,00421637	17891863,2	2016,336105	7891864	2016,380806

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	500000	0,164	0,005163978	1891286,1	550,7993081	641286,3	550,9569554
	1000000	0,32	0	3781533,2	1134,588599	1281533,7	1134,847817
	1500000	0,481	0,003162278	5672598,2	1460,588603	1922598,8	1461,031584
	2000000	0,643	0,004830459	7561952,5	1421,290435	2561952,8	1421,348796
	2500000	0,804	0,005163978	9451792,5	1737,016551	3201793,7	1736,517015
	3000000	0,965	0,005270463	11343506,6	1241,25279	3843508	1240,585346
	3500000	1,126	0,005163978	13232700,2	1963,213567	4482701,4	1962,954417
	4000000	1,283	0,004830459	15123730,4	2589,1141	5123731,3	2588,812514
	4500000	1,447	0,004830459	17014660,7	1462,086869	5764662,1	1462,036209
	5000000	1,61	0,004714045	18907704	1449,880531	6407705	1449,538854

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	500000	0,165	0,007071068	2246331,6	822,1827588	789729,7	535,6369106
	1000000	0,365	0,07412452	4491501,9	1729,608398	1578332,7	1061,665782
	1500000	0,492	0,007888106	6737282,9	2148,879525	2367849,9	1126,212182
	2000000	0,694	0,111175537	8982875	2370,819971	3155358,1	1309,221775
	2500000	0,834	0,027968236	11226629,5	2532,744987	3943271,8	1673,567301
	3000000	1,021	0,101811372	13472910,7	2195,019921	4733661,7	1418,577225
	3500000	1,306	0,242725268	15718056,7	2966,340396	5521112,7	1685,270044
	4000000	1,328	0,039944406	17965572,2	5079,600483	6310512,7	3097,11669
	4500000	1,484	0,017763883	20210834,4	2944,859861	7098855	2089,261433
	5000000	1,672	0,059777365	22461992,9	3412,45651	7891864	2016,380806

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	500000	0,148	0,006324555	1219686,9	370,3111154	412113,1	371,2356095
	1000000	0,297	0,004830459	2439166,6	661,0505444	824372,2	743,8044546
	1500000	0,45	0,006666667	3658432,3	790,8900823	1236193	951,7800866
	2000000	0,597	0,012516656	4877244,8	498,694919	1648090,4	1006,404624
	2500000	0,74	0	6095938,8	849,488709	2059185,9	905,51501
	3000000	0,896	0,006992059	7316251,8	854,539096	2471951,3	1024,84742
	3500000	1,045	0,011785113	8534301,4	1048,72369	2883116,1	1031,85652
	4000000	1,198	0,007888106	9754470,8	1504,714428	3295761	2194,711674
	4500000	1,342	0,006324555	10973487	854,3588889	3707729,2	1252,726431
	5000000	1,496	0,00843274	12194028,8	1000,216288	4121174,8	918,6314459

Tab. B.9: Delete–Min operace na haldě o 5ti mil. prvcích

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	500000	2,35	0	11292537,2	453,8841751	10474405,6	439,5637231
	1000000	4,674	0,006992059	22426196	545,2936008	20810367,3	548,945059
	1500000	6,949	0,007378648	33516004,2	589,1220964	31084821,3	704,6313536
	2000000	9,395	0,005270463	44602245,1	728,3734466	41323956,2	1009,088896
	2500000	11,75	0,006992059	55513824,2	768,5420252	51293007,5	1190,846781
	3000000	14,05	0,009660918	66180472,7	874,0644586	61053228,5	1139,192526
	3500000	16,38	0,006749486	76769649,4	948,4046488	70808627	1278,684133
	4000000	18,61	0,014298407	87051502,3	944,7532364	80169302,5	1160,170605
	4500000	20,69	0,008232726	96988549	964,9485882	89230652,1	1296,59241
	5000000	22,3	0,018856181	105926583	1190,050905	97293563,7	1524,508886

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	500000	2	0	20948203	972,515524	10347787,2	806,4988531
	1000000	4,008	0,007888106	41651954,2	1658,342332	20550200,3	1409,195992
	1500000	5,96	0,006666667	62088289,9	2139,841347	30594464,8	1746,418163
	2000000	8,051	0,003162278	82241701,5	2547,771325	40471749	1993,216496
	2500000	10,14	0,011352924	102092472	2123,86978	50163689,9	1711,967319
	3000000	12,24	0,007071068	121819263	2391,41014	59852853,3	2046,483762
	3500000	14,28	0,01197219	141065951	2866,382096	69273718,4	2369,01734
	4000000	16,28	0,016193277	159856061	2672,270414	78466821,3	2289,884958
	4500000	18,11	0,008232726	177876624	2793,343176	87276127,4	2384,008119
	5000000	19,51	0,013165612	193898457	3192,187895	95086776,3	2689,370271

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	500000	11,16	0,015776213	41978226,1	1251,975901	30113069,6	938,321338
	1000000	22,72	0,030110906	83606107,6	1975,74679	59986525,7	1739,992404
	1500000	34,61	0,051434964	124885603	1784,234838	89655930,1	1593,05991
	2000000	46,91	0,042018514	165849239	3041,30904	119135557	2196,093997
	2500000	59,42	0,076514341	206469524	2653,495188	148380570	2002,978196
	3000000	71,69	0,0967758	246343710	2811,983857	176980567	2740,64955
	3500000	83,79	0,107372664	285483143	3090,46631	205079225	2914,443069
	4000000	95,6	0,166679999	323724053	2664,170465	232493781	2964,64137
	4500000	106,8	0,185771664	360415091	2533,559955	258750335	3139,902872
	5000000	115,5	0,182622866	393105962	3277,847573	282007916	3984,948639

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	500000	3,201	0,007378648	30739769,5	874,0516448	10347782,3	802,6745639
	1000000	6,468	0,075395255	61182269,5	1165,970769	20550199	1417,118987
	1500000	9,587	0,025841397	91290180,6	1701,614149	30594440,4	1718,530651
	2000000	13,1	0,046055522	121039982	2131,486649	40471675,1	1906,432529
	2500000	16,6	0,067733136	150422509	2248,296266	50163629,8	1680,636969
	3000000	20,04	0,041952354	179473302	2173,486763	59852786,4	2106,895357
	3500000	23,01	0,320249902	207872609	2566,073276	69273615,6	2648,712064
	4000000	26,01	0,048085572	235586138	2524,069521	78467260	2732,120788
	4500000	29,07	0,069673843	262142243	2593,09245	87276754,5	3066,548775
	5000000	31,47	0,062396581	285698040	3305,905934	95087736,8	3637,637842

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	500000	3,009	0,005676462	36507064	1222,860217	5362158,4	682,1585348
	1000000	6,053	0,006749486	72553579,9	1596,051687	10691943,8	865,1207751
	1500000	9,048	0,012292726	108323767	1790,837386	16019261,3	937,9137665
	2000000	12,22	0,009944289	143850681	2087,619019	21263931,3	1153,413971
	2500000	15,38	0,011005049	179136515	1912,153414	26412855,7	1198,315211
	3000000	18,5	0,012649111	214176139	2345,373718	31527636,3	1507,472502
	3500000	21,6	0,013498971	248493083	2151,052262	36417160,7	1617,935451
	4000000	24,62	0,010749677	281735036	3280,09834	41164480,8	1889,864945
	4500000	27,41	0,021628171	313634122	3350,250155	45868324,6	1891,941284
	5000000	29,58	0,020548047	342120143	3550,175176	49983552,2	1917,251088

Tab. B.10: Delete–Max operace na haldě o 5ti mil. prvcích

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DEA	500000	2,303	0,004830459	11292508,6	444,7941347	10398695,7	507,2431041
	1000000	4,584	0,005163978	22432646,6	694,1479509	20667930,3	632,9574411
	1500000	6,781	0,007378648	33541998,5	849,2620653	30719808,5	963,7205969
	2000000	9,142	0,013165612	44618361,2	926,3547197	40574040,9	929,9711405
	2500000	11,49	0,008232726	55531784,1	712,705713	50392756,4	788,7665758
	3000000	13,8	0,00843274	66197589,4	931,9255097	60180969,9	1094,251286
	3500000	16,03	0,013498971	76784411,5	1055,286612	69616946	969,2023066
	4000000	18,22	0,020439613	87079744,5	1219,444978	78930077	994,6022097
	4500000	20,21	0,01490712	97023019,6	1408,260724	87808346,4	1220,162211
	5000000	21,78	0,012472191	105967493	1504,557905	95688481,3	1255,341304

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
DIA	500000	1,99	0,006666667	20947288	1234,011435	10346908	1005,678433
	1000000	3,98	0,004714045	41651333,8	1366,252279	20549495,2	1138,468738
	1500000	5,925	0,005270463	62088440,6	1292,156354	30594232,9	1061,625007
	2000000	8,004	0,006992059	82241670,4	1389,329511	40471470,8	1234,610132
	2500000	10,08	0,010540926	102092614	1967,438947	50163490,3	1477,080307
	3000000	12,17	0,012516656	121818822	2029,223226	59852239,8	1631,387194
	3500000	14,2	0,012516656	141065640	1709,260191	69273182,9	1373,038192
	4000000	16,2	0,02321398	159855912	2071,382493	78466384,8	1647,135885
	4500000	18,01	0,014757296	177877218	2542,065881	87276228,5	1957,269825
	5000000	19,42	0,023687784	193899530	2299,009886	95087225,7	1911,308455

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
RMM	500000	11,19	0,023593784	41978175,6	995,5668625	30112991,7	678,0732589
	1000000	22,78	0,032930904	83606345	1865,612619	59986791,2	1391,09244
	1500000	34,72	0,064678693	124887524	1651,306473	89656866,3	1504,021945
	2000000	47,04	0,084301048	165851970	2759,204155	119137062	2257,283953
	2500000	59,58	0,096729864	206471831	3215,464385	148381787	2802,04719
	3000000	71,87	0,090455637	246345533	3338,794841	176981881	2221,477839
	3500000	84,07	0,11120052	285484907	3717,953505	205079689	2779,381468
	4000000	95,84	0,109082028	323725587	4500,680645	232494155	3365,035349
	4500000	107,1	0,19851952	360416290	4744,584128	258751448	3666,126477
	5000000	115,7	0,095248214	393106329	5856,276812	282007600	4075,037506

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
SMM	500000	3,13	0,004714045	30739414,9	1004,886776	10346908	1005,678433
	1000000	6,266	0,009660918	61182316	1059,582413	20549495,2	1138,468738
	1500000	9,328	0,006324555	91291252,6	1314,200238	30594232,9	1061,625007
	2000000	12,59	0,012692955	121040835	1583,994935	40471470,8	1234,610132
	2500000	15,84	0,011785113	150423881	2186,067664	50163490,3	1477,080307
	3000000	19,06	0,017288403	179474041	1908,896365	59852239,8	1631,387194
	3500000	22,22	0,015238839	207873470	1717,314428	69273182,9	1373,038192
	4000000	25,31	0,014298407	235586758	2140,209242	78466384,8	1647,135885
	4500000	28,17	0,024244129	262143146	2665,48424	87276228,5	1957,269825
	5000000	30,4	0,017288403	285698981	2497,186775	95087225,7	1911,308455

heap	len	avg(t)	var(t)	avg(cmp)	var(cmp)	avg(swp)	var(swp)
MMH	500000	2,944	0,005163978	35699639,4	710,3920983	5217605,4	529,6697504
	1000000	5,902	0,007888106	71160968	884,9849967	10316529,1	662,657017
	1500000	8,804	0,011737878	106172465	856,1435757	15272690,4	546,2358465
	2000000	11,91	0,010749677	140769735	1355,005633	20154265,8	615,5024686
	2500000	15,01	0,017029386	174888894	1966,10711	24951931,2	625,4301275
	3000000	18,03	0,01490712	208212591	1965,069589	29686650,3	800,0107013
	3500000	20,99	0,021628171	240850566	1914,156098	34462743,1	978,385115
	4000000	23,94	0,016329932	273012576	2256,786959	39096605,1	1306,523925
	4500000	26,66	0,033065591	303792334	2653,062717	43416510,1	1288,37507
	5000000	28,82	0,046427961	330984715	2654,029021	47326454,3	1231,104026

C Struktura přiloženého média

Na přiloženém médiu jsou k dispozici zejména zdrojové soubory implementace testovaných prioritních front, včetně skriptů pro automatizované testování a skriptů pro zpracování naměřených hodnot. Dále jsou přiloženy všechny obrázky, tabulky a grafy, včetně těch, které se již do výsledné práce nevešly. Soubory jsou uloženy v následující adresářové struktuře:

```
dev/
    src/            - implementace prioritních front
    scripts/        - skripty pro testování a zpracování výsledků
    tests/          - adresářová struktura pro testy
    results/
        processed/  - dílčí výsledky jednotlivých testů
        statistica/ - směrodatné odchylky z programu Statistica
        xls/        - tabulky a grafy pro Microsoft Excel
doc/                - dokumentace práce
    img/            - obrázky
```

D Dokumentace implementace

V této části příloh se budeme krátce věnovat obecnému popisu zdrojových kódů, způsobu jejich přeložení do binární podoby. Zmíníme rovněž i nástroje určené k vytvoření množiny testovacích dat, způsob spouštění jednotlivých testů na těchto datech a prostředky ke zpracování naměřených výsledků.

D.1 Zdrojové soubory

Zdrojové kódy jsou umístěny na přiloženém médiu v adresáři `dev/src/`. Zdrojové soubory poměřovaných datových struktur jsou pojmenovány jako `<zkratka struktury>_cpp.[cc|h]`. Jednotlivé datové struktury jsou implementovány obecně jako šablony. Mohou tedy uchovávat libovolné datové prvky, nad kterými je nutné správně předefinovat operátor porovnání $<$. Každá prioritní fronta pro potřeby měření pak poskytuje zejména tyto hlavní metody.

`Insert(T *item)`
: Metoda pro vložení prvku do haldy.

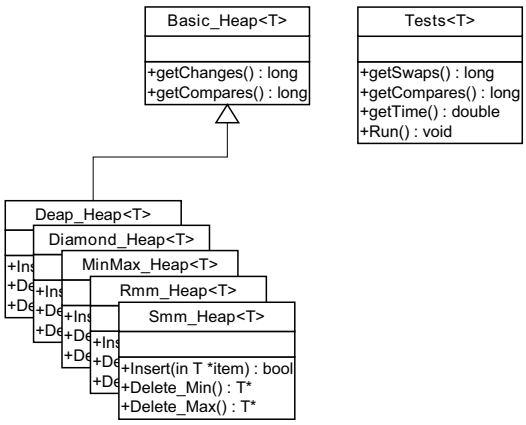
`T* Delete_Min()`
: Metoda odebere minimum z prioritní fronty.

`T* Delete_Max()`
: Metoda odebere maximum z prioritní fronty.

`long int getChanges()`
: Vrací celkový počet výměn od začátku práce s haldou.

`long int getCompares()`
: Vrací celkový počet porovnání od začátku práce s haldou.

Pro názornost přikládáme na obrázku D.1 jednoduchý class diagram s architekturou testovací aplikace.



Obr. D.1: Class diagram

D.1.1 Překlad

Pro správný překlad zdrojových souborů je nutný překladač g++. Výsledek překladu se uloží do adresáře `dev/bin/`. Překlad se provádí z příkazové řádky prostřednictvím následujících příkazů:

`make -f Makefile32 clean all`
: Provede překlad, který bude měřit množství spotřebovaného procesorového času.

`make -f Makefile32_COMPARISONS clean all`
: Provede překlad, který zapne direktivu `-d_COMPARISONS_`, tj. místo množství spotřebovaného procesorového času se bude měřit pouze počet porovnání a výměn.

D.1.2 Spuštění aplikace

Běh aplikace je modifikovatelný prostřednictvím parametrů předaných aplikaci při startu.

Použití

`./Heaps <TY1|TY2> <SMM|RMM|DIA|DEA|MMH> <cesta k souboru s test. daty> [volitelně cesta k souboru s daty pro vytvoření haldy]`

- `<TY1|TY2>` udává typ spouštěného testu. TY2 znamená, že je třeba

před samotným měřením vybudovat haldu ze souboru, který je předán aplikaci jako poslední parametr aplikace.

- `<SMM|RMM|DIA|DEA|MMH>` označuje prioritní frontu, kterou budeme testovat.

- Předposlední parametr je cesta k souboru s testovacími daty, který obsahuje posloupnoust vkládání a odebírání prvků z prioritní fronty.

D.2 Testovací skripty

Všechny testovací skripty jsou umístěny v adresáři `dev/scripts/`. Jsou určeny pro shell `bash`.

D.2.1 Generování testovacích dat

Pro potřeby hromadného generování testovacích dat byl napsán jednoduchý program, který na základě vstupních parametrů vygeneruje na standardní výstup jednu sadu testovacích dat. Program se nachází v podadresáři `random`.

Použití

```
./Random -c <velikost dat> [-s <hnízdo pseudo posloupnosti>]
[-m <maximální hodnota v posloupnosti>] -r <insert:del-min:
del-max poměr operací> [-e]
```

Přepínač `-e` na výstupu setřídí data podle použité operace, tj. nejprve vypíše operace `Insert`, pak `Delete-Min` a nakonec `Delete-Max`.

Pro potřeby opakovaného spouštění testů na stejných datech je uživateli k dispozici skript `generate.sh` ve stejném adresáři. Tento skript generuje na základě vstupních parametrů sadu testovacích dat stejné velikosti do předem specifikovaného adresáře.

Použití

```
./generate.sh <počet opakování> <soubor s délkami testovacích
dat> <poměr mezi operacemi insert:delete-min:delete-max> <prefix
výstupních souborů> <adresář pro výstup>
```

D.2.2 Spouštění testů

Testy se spouští prostřednictvím skriptu `run_tests.sh`. Tento skript čte testovací data umístěná v adresářích `tests/simple/`, `tests/preheap/` a `tests/preheap_tests/` a postupně na všech testuje implementované datové struktury. Každý test se pak ještě pouští dvakrát, jednou pro získání množství spotřebovaného procesorového času a jednou pro získání počtu porovnání a výměn.

Výsledky s naměřenými hodnotami se ukládají do adresáře `results/`. Pro jeden konkrétní test se vytvoří celkem pět souborů, tj. pro každou datovou strukturu jeden. Obsah takového souboru vypadá následovně:

```
Leng:  1000000      - délka testované posloupnosti
Time:  0.33000      - naměřený čas
Comp:  3137208      - počet porovnání
Swap:  1570583      - počet výměn
```

D.3 Zpracování výsledků

Soubory s výsledky testů je třeba zpracovat skriptem `process_results.sh`. Ten shlukuje výsledky opakovaného spouštění testů pro konkrétní datovou strukturu do jednoho souboru. Výsledek ukládá do adresáře `results/processed/`. Takto zpracované výsledky je možné předat k dalšímu zpracování, například statistickému programu Statistica pro výpočet směrodatných odchylek či tabulkovému procesoru pro vytvoření názorných grafů reprezentujících dosažené výsledky.

Provedené a zpracované testy jsou uloženy v adresáři `results/xls/` a `results/statistica/`. Jsou zde uvedeny všechny v textu uváděné tabulky a grafy včetně dalších zajímavých grafů a tabulek, které se již do výsledné práce nevešly. Pro úplnost jsou však uvedeny právě na přiloženém médiu.

Směrodatné odchylky jsme počítali statistickým programem Statistica. Běžně používaný tabulkový procesor Microsoft Excel bohužel směrodatné odchylky počítá zcela špatně, jak jsme si experimentálně ověřili. Nedokázal správně spočítat směrodatnou odchylku z deseti stejných čísel, jenž by měla být rovna 0. Chyba je zřejmě způsobena tím, že Microsoft Excel interně

počítá na 15 platných číslic. Výpočet tak jednoduše přeteče. K možnému dalšímu statistickému zpracování naměřených dat jej proto nelze doporučit. Použili jsme jej pouze pro generování grafů a výsledných tabulek, které se v práci objevily.

Literatura

[1] A. Arvind a C. P. Rangan. Symmetric min–max heap: A simpler data structure for double-ended priority queue. *Information Processing Letters 69*, pages 197–199, 1999.

[2] M. Atkinson, J. Sack, N. Santoro a T. Strothotte. Min-max heaps and generalized priority queues. *Communications of the ACM 29*, pages 996–1000, 1986.

[3] G. S. Brodal. Fast meldable priority queues. *Workshop on Algorithms and Data Structures*, pages 282–290, 1995.

[4] S. Carlsson. The deap – a double-ended heap to implement double-ended priority queues. *Information Processing Letters 26*, pages 33–36, 1987.

[5] S. Carlsson. A variant of heapsort with almost optimal number of comparisons. *Information Processing Letters 24*, pages 247–250, 1987.

[6] S. Carlsson, J. Chen a T. Strothotte. A note on the construction of the data structure "deap". *Information Processing Letters 31*, pages 315–317, 1989.

[7] S. Chang a M. Du. Diamond deque: A simple data structure for priority deques. *Information Processing Letters 46*, pages 231–237, 1993.

[8] K. Chong a S. Sahni. Correspondance–based data structures for double-ended priority queues. *The ACM Journal of Experimental Algorithmics 5, Article 2*, 2000.

[9] R. Fagerberg. A note on worst case efficient meldable priority queues. *Technical report, PP–1996–12*, 1999.

[10] R. Floyd. Treesort 3 algorithm 245. *Communications of the ACM 7*, 1964.

[11] G. Gonnet. Handbook of algorithms and data structures. 1984.

[12] E. Horowitz, S. Sahni a D.Mehta. Fundamentals of data structures in c ++. 1995.

[13] D. Knuth. The art of computer programming, vol. 3, sorting and searching. 1973.

[14] C. Makris, A. Tsakalidis a K. Tsichlas. Reflected min–max heaps. *Information Processing Letters 86*, pages 209–214, 2003.

[15] S. Olariu, C. Overstreet a Z. Wen. A mergeable double–ended priority queues. *The Computer Journal 34*, pages 423–427, 1991.

[16] S. Skov a J. Olsen. A comparative analysis of three different priority deques. *CPH STL Report 2001–14*, 2001.