FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Praktická výkonnost různých implementací prioritní fronty
Student:	Radoslav Hašek
Vedoucí:	doc. Ing. Ivan Šimeček, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

1) Nastudujte [1,2,4] abstraktní datovou strukturu prioritní fronta.
2) Nastudujte [1,2,3,5] různé možnosti implementace prioritní fronty.
3) Nastudujte [6,7,8] a popište možnosti a přínos paralelizace prioritní fronty.
4) Jednotlivé varianty z bodu 2) implementujte v jazyce C++.
5) Porovnejte výkonnost jednotlivých implementací pro různé typy (velikost, různý poměr prováděných operací) vstupních dat na školním serveru STAR.
6) Diskutujte dosažené výsledky.
–
[1] Introduction to Algorithms; Cormen, Leserson, Rivest, Stein; ISBN 978-0-262-03384-8
[2] The Algorithm Design Manual; Steven S. Skiena; ISBN 978-1848000698
[3] On the efficiency of pairing heaps and related data structures; Michael L. Fredman; https://dl.acm.org/citation.cfm?id=320214
[4] Algorithms (4th Edition); R. Sedgewick, K. Wayne; ISBN 978-0321573513
[5] Theoretical and practical efficiency of priority queues; Claus Jensen; http://www.diku.dk/~jyrki/PE-lab/Claus/thesis.pdf
[6] A Parallel Priority Queue with Constant Time Operations; Brodaly Träff, Zaroliagis; https://www.cs.au.dk/~gerth/papers/jpdc98.pdf
[7] A Survey on Parallel Algorithms for Priority Queue Operations; Lixin Yu; https://pdfs.semanticscholar.org/aa47/5ab0de18955ad295e606b5efac1d61cfa7f3.pdf[8] Praktická výkonnost různých implementací prioritní fronty; Šimon Schierreich; BP FIT ČVUT;2019

Elektronicky schválil/a doc. Ing. Jan Janoušek, Ph.D. dne 1. září 2020 v Praze.

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Praktická výkonnost různých implementací prioritní fronty

Radoslav Hašek

Katedra teoretické informatiky
Vedoucí práce: doc. Ing. Ivan Šimeček, Ph.D.

5. května 2021

Poděkování

Poděkování patří doc. Ing. Ivanu Šimečkovi Ph.D. za cenné rady a připomínky poskytnuté během psaní této práce a také všem ostatním učitelům na FITu, díky kterým jsem se dostal až do tohoto bodu.

Děkuji také své rodině, která mě ve studiu vždy podporovala.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Hašek, Radoslav. *0.0.0.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Teoretická část této práce popisuje abstraktní datovou strukturu prioritní fronta, její podporované operace a řadu možných implementací, které jsou podrobně rozebrány po teoretické stránce. Dále jsou uvedeny příklady užití prioritní fronty. Práce rovněž nastiňuje několik možností paralelního přístupu k této struktuře.

V rámci praktické části byla většina z možných realizací prioritní fronty implementována v jazyce C++ a na školním serveru STAR byla změřena jejich praktická výkonnost v několika scénářích. Práce uvádí výsledky těchto měření a analyzuje je. V okomentovaných grafech jsou jednotlivé implementace přímo porovnány.

Na základě těchto výsledků je možné vybrat vhodnou variantu prioritní fronty pro konkrétní použití a zejména pro velká vstupní data tak ušetřit výpočetní čas. Z měření vyplynulo že implementačně jednodušší nebo asymptoticky pomalejší datové struktury mohou být v praxi výkonnější.

Klíčová slova prioritní fronta, datová struktura, pole, spojový seznam, vyhledávací strom, halda, výkonnost, paralelizace

Abstract

The theoretical part of this thesis describes an abstract data structure called priority queue, it's supported operations and a number of possible implementations which are described in detail from the theoretical point of view. After that, examples of priority queue applications are listed. This thesis also mentions several possible approaches to parallelization of this structure.

In the practical part, most of the described priority queue realizations have been implemented in C++. Their practical performance has been measured on the STAR school cluster in several test cases. This thesis shows results of these measurements and analyses them. The implementations are compared on several commented graphs.

These results help to choose a suitable priority queue variant for a specific purpose in order to save computing time, especially for large input data. The measurements have shown that simpler or asymptotically slower data structures can be faster in practice.

Keywords priority queue, data structure, array, linked list, search tree, heap, performance, parallelization

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Řada problémů vyžaduje přístup k prvkům nikoli v pořadí, ve kterém byly vloženy (neboli princip FIFO), ani v opačném pořadí (LIFO), ale podle jejich důležitosti. Člověk chce nejdříve splnit ten nejdůležitější úkol, procesor se snaží poskytnout čas procesům s největší prioritou atp.

Prioritní fronta je narozdíl od klasické fronty nebo zásobníku pro problémy tohoto typu užitečná, neboť právě takový přístup k prvkům nabízí. Určitě je žádoucí, aby byla co nejefektivnější, jenže existuje řada způsobů, jak ji implementovat, z nichž některé se od sebe svou efektivitou značně odlišují. Tato práce nabídne jejich popis a právě srovnání jejich efektivity v různých situacích. Ukáže, zda je výhodné použít velmi složité datové struktury, které mají příznivou asymptotickou složitost ale velké skryté konstanty, a které jsou náchylné na chyby často způsobující i pád programu, nebo jestli nepostačují podstatně jednodušší struktury, které mohou být ve specifických případech i rychlejší.

Bude nahlédnuto i do možností paralelního přístupu k prioritní frontě, což je oblast zatím prozkoumaná jen málo.

Struktura práce

První kapitola tvoří teoretickou část. Je zde představena abstraktní datová struktura prioritní fronta, operace které podporuje a jejich teoretické složitosti. V další části kapitoly je podrobně rozebráno množství datových struktur, které se dají použít při implementaci prioritní fronty, od těch nejjednodušších až po ty pokročilé. Následují příklady využití prioritní fronty. Na konci kapitoly jsou popsány možnosti a přínos paralelizace této struktury.

Datové struktury představené v první kapitole byly implementovány v jazyce C++. Zatímco první kapitola poskytla abstraktnější popis datových struktur, kapitola druhá vysvětluje některé zajímavé implementační detaily.

Ukazuje jakých vlastností jazyka bylo využito, jaké operace byly nakonec implementovány atd.

Třetí kapitola se zabývá měřením výkonnosti jednotlivých implementací prioritní fronty. Na začátku jsou popsány metody měření, použitý hardware a nastavení překladače. Potom následují popisy jednotlivých testů a jejich okomentované výsledky zobrazené v grafech. Vyhodnocuje se, jak která implementace obstála v jednotlivých testech a která se tedy hodí pro různá použití.

Kapitola 1

Analýza

1.1 Prioritní fronta

Prioritní fronta je abstraktní datová struktura umožňující přístup k prvkům na základě nějakého klíče – priority. Hodnoty těchto klíčů musí tvořit úplně uspořádanou množinu, aby bylo vždy možné rozhodnout o tom, který klíč je větší [2].

Pokud je jako první vydán prvek s největší prioritou, hovoří se o maximové prioritní frontě, pokud naopak ten s nejmenší prioritou, jedná se o minimovou prioritní frontu [1].

V celé práci se pracuje s maximovou verzí, pokud není řečeno jinak.

1.1.1 Operace

Maximová prioritní fronta Q podporuje minimálně tyto dvě operace [3]:

- `insert(Q, k)` vloží do Q nový prvek s klíčem k.
- `extractMax(Q)` odstraní z Q prvek s největší prioritou a vrátí ho. Otázka je, jak bude tato operace pracovat s prvky, jejichž klíče se rovnají. Buďto lze ze všech prvků s maximálním klíčem vrátit ten, který byl vložen nejdříve, nebo nechat toto pořadí nedefinované. Dále bude uvažovaná varianta s nedefinovaným pořadím prvků se stejnými klíči.

Volitelně může podporovat i další operace jako jsou například [4]:

- `size(Q)` vrátí počet prvků v Q.
- `findMax(Q)` vrátí prvek z Q s největší prioritou, ale neodstraní ho.
- `merge(Q1, Q2)` sloučí Q_1 a Q_2, neboli vloží do Q_1 všechny prvky Q_2. Tato operace může změnit strukturu Q_2.

- `increaseKey(Q, x, k)` změní prvku x klíč na k za předpokladu, že k je větší než stávající klíč prvku x.
- `delete(Q, x)` odstraní prvek x z Q.

Jak již bylo řečeno, v prioritní frontě bude povolena přítomnost více prvků se stejným klíčem, přičemž nebude nijak garantováno jejich pořadí. Pokud by poslední dvě zmíněné operace byly definované jako `increaseKey(Q, k, k')` a `delete(Q, k)`, kde k je klíč nějakého prvku a k' je jeho nový klíč, při přítomnosti více prvků se stejným klíčem bychom neměli žádnou jistotu, na který z nich bude tato operace aplikovaná. Je tedy nutné těmto operacím předávat ukazatel na prvek.

1.2 Datové struktury použitelné pro její implementaci

Nyní bude představena celá řada možností, jak prioritní frontu implementovat, a bude ukázáno, jak jsou v nich realizovány jednotlivé operace. Určitě jsou důležité asymptotické složitosti všech operací, byť hlavní jsou rozhodně `insert(Q, k)` a `extractMax(Q)`. Pro ně je ale známý dolní odhad složitosti a lze tedy říci, která datová struktura je umožňuje provádět v optimálním čase. K určení dolní meze pomůže tato věta:

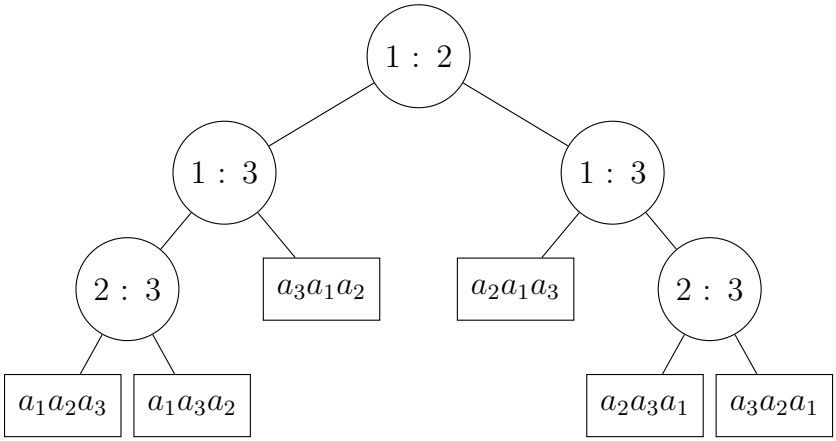
Věta 1: *Každý deterministický algoritmus v porovnávacím modelu, který třídí n-prvkovou posloupnost, použije v nejhorším případě $\Omega(n \log n)$ porovnání [3].*

Důkaz. Důkaz této věty je uveden např. v [1], [3], [5]. V tomto textu bude pouze nastíněna hlavní myšlenka převzatá z těchto zdrojů, ale nebudou dokazována pomocná lemmata, ani nebudou definované všechny pojmy.

Chceme seřadit vstupní posloupnost $a_1, a_2, ..., a_n$. Jediný způsob, jak lze získat informaci o pořadí prvků ve vstupních datech, je jejich porovnání. Režii spojenou se všemi ostatními operacemi (jako je přesouvání) neuvažujeme, ty mohou výsledný čas pouze zvýšit.

Sestavíme si rozhodovací strom, neboli binární strom, jehož listy obsahují jednotlivé permutace vstupní posloupnosti. Vnitřní uzly mají tvar $(i : j)$, kde $1 \leq i, j \leq n$, a odpovídají porovnání prvků s těmito indexy. Tento strom reprezentuje běh algoritmu. Pokud algoritmus provede porovnání dvou prvků a_i a a_j a zjistí, že $a_i \leq a_j$, jde do levého syna, v opačném případě jde do pravého syna. Jakmile vstoupí do listu, znamená to, že seřadil vstupní posloupnost a výsledkem je její permutace obsažená v tomto listu.

Každá cesta z kořene do listu odpovídá posloupnosti porovnání vykonaných algoritmem, pokud se v listu nachází permutace, která seřadí vstupní posloupnost. Bez provedení všech těchto porovnání nemá algoritmus dost informací na to, aby prohlásil, že tato permutace je ta správná.



Obrázek 1.1: Příklad rozhodovacího stromu pro tříprvkovou posloupnost.

Existuje celkem $n!$ permutací vstupní posloupnost, tudíž náš rozhodovací strom alespoň $n!$ listů. Žádnou nelze vynechat, protože potom bychom vždy dokázali najít vstup, který algoritmus nebude schopný korektně seřadit. Časové složitosti algoritmu v nejhorším případě odpovídá délka nejdelší cesty z kořene do listu.

Lemma 1: *Binární strom hloubky h má nejvýše 2^h listů.*

Díky tomuto lemmatu můžeme říci, že pro počet listů l rozhodovacího stromu platí

$$n! \leq l \leq 2^h$$

Což po zlogaritmování dává

$$h \geq \log(n!) = \Omega(n \log n)$$

Hloubka stromu je tedy aspoň $n \log n$ a to je také délka nejdelší cesty z kořene do listu. □

S pomocí této věty je možné dokázat, že platí:

Věta 2: *Alespoň jedna z operací* `insert(Q, k)` *a* `extractMax(Q)` *na prioritní frontě má asymptotickou složitost* $\Omega(\log n)$ *[4].*

Důkaz. Pro spor mějme prioritní frontu Q, kde obě zmíněné operace mají složitosti $o(\log n)$.

Dále mějme posloupnost n čísel. Pokud bychom všechna čísla z této posloupnosti vložili do Q a následně n krát zavolali `extractMax(Q)`, dostali bychom seřazenou posloupnost. To znamená, že bychom toto seřazení zvládli v čase

$$\Theta(n) \cdot o(\log n) + \Theta(n) \cdot o(\log n) = \Theta(n) \cdot o(\log n)$$

a jelikož platí, že $\Theta(n) \cdot o(\log n) \in o(n \cdot \log n)$, došli jsme ke sporu s předchozí větou [4]. □

U všech dále popisovaných datových struktur bude udržován ukazatel na největší prvek, i když to nebude explicitně zmíněno. Není to nutné a nijak to nemění asymptotické složitosti operací `insert(Q, k)` a `extractMax(Q)` (bude to mít vliv pouze na skryté konstanty), ale ve všech případech bude díky tomu mít operace `findMax(Q)` konstantní časovou složitost. Bude totiž stačit vrátit hodnotu, na kterou tento ukazatel ukazuje. Tato operace už nebude dále popisována.

Stejně tak si bude daná struktura vždy zvlášť pamatovat a aktualizovat čítač prvků a dále nebude vysvětlována operaci `size(Q)`.

1.2.1 Pole

Pole je spojitá datová struktura fixní velikosti umožňující přístup k prvkům na základě jejich indexu nebo adresy v konstantním čase. Je prostorově velice efektivní, protože nepotřebuje žádné ukazatele ani jiné režijní informace [2].

Implementace prioritní fronty pomocí pole je ta vůbec nejjednodušší varianta. Problémem je ale ona fixní velikost. Možností by bylo pro tento účel vytvořit pole s předem danou velkou kapacitou, ale to by bylo plýtvání místem a stejně by nebylo zaručeno, že velikost bude dostatečná. Proto bude nyní představeno takzvané dynamické pole [2].

Dynamické pole umožňuje změnu velikosti za běhu programu. Začne se s polem dané velikosti a pokud se toto pole celé zaplní, vytvoří se nové pole o velikosti k násobku velikosti původní (často se jako k používá číslo 2), do něho se překopírují všechny stávající prvky, staré pole se dealokuje a pracuje se už jen s novým polem. Ve chvíli, kdy se musí pole velikosti n zvětšit, zabere operace vkládání očividně $\Theta(n)$ času. V ostatních případech zabere konstantní čas.

Nyní ale bude ukázáno, že složitost operace vkládání je poněkud překvapivě amortizovaně konstantní.

Věta 3: *Přidání n prvků do prázdného dynamického pole trvá $\Theta(n)$ času [2].*

Důkaz. Bez újmy na obecnosti zvolme $k = 2$. Začíná se s polem velikosti 1. Zajímá nás, kolikrát musíme zkopírovat prvek ať už do stávajícího pole nebo do nového pole dvojnásobné velikosti během n vkládání.

Abychom mohli vložit celkem n prvků, bude nutné provést $\log_2 n$ zvětšení pole. První vložený prvek bude nutné zkopírovat při prvním, druhém, čtvrtém, osmém atd. vkládání. Ale ne všechny prvky budou kopírovány tak často. Například prvek vložený na pozici $n/2 + 1$ bude zkopírován jen jednou (při svém vkládání), protože pole už je v tu dobu na dostatečné kapacitě na pojmutí všech n prvků.

To znamená, že polovina prvků bude zkopírována jednou, čtvrtina dvakrát, osmina třikrát atd. a celkový počet kopírování je roven

$$\sum_{i=1}^{log_2 n} i \cdot n \,/\, 2^i \;= n \sum_{i=1}^{log_2 n} i \,/\, 2^i \;\leq n \sum_{i=1}^{\infty} i \,/\, 2^i \;= 2n$$

Suma na pravé straně nerovnosti je řada konvergující k číslu 2 [2]. □

Vložení n prvků vyžaduje $\Theta(n)$ operací kopírování. Časová složitost operace vkládání do dynamického pole je tedy $O^*(1)$.

Nyní budou představeny dva způsoby, jak s polem pracovat.

1.2.1.1 Neseřazené pole

Prvky prioritní fronty jsou udržovány v dynamickém poli. Pro přehled o tom, kde se v poli nachází maximum, si stačí pamatovat jeho index.

insert(Q, k) Vkládání do neseřazeného pole je velice jednoduché, nový prvek je pouze vložen na konec, což zabere, jak již bylo řečeno, $O^*(1)$ času. Poté jsou porovnány klíče stávajícího maximální prvku s prvkem nově vloženým a případně je upraven index maxima. Celá operace má tedy časovou složitost $O^*(1)$.

extractMax(Q) Díky indexu největšího prvku je známo, kde se maximum nachází. Aby nebylo nutné všechny prvky, které se v poli nachází dál než současné maximum, posouvat o jednu pozici doleva, zkopíruje se na jeho pozici poslední prvek a pole se zmenší o 1. Potom se nalezne nové maximum, což se nedá udělat jinak než projitím všech prvků. Celková časová složitost `extractMax(Q)` je $O(n)$.

merge(Q1, Q2) Pro sloučení dvou neseřazených polí a_1 a a_2 stačí alokovat nové pole o velikosti $n + m$ ($n = \texttt{size}(a_1)$ a $m = \texttt{size}(a_2)$) a překopírovat do něj všechny prvky z a_1 a za ně prvky z a_2. Index nového maxima může být buďto index maxima a_1 nebo index maxima a_2 zvýšený o $\texttt{size}(a_1)$. Na to stačí jedno porovnání. Operaci `merge(Q1, Q2)` lze tedy zvládnout za $O(n+m)$ času.

increaseKey(Q, x, k) Prvky v neseřazeném poli mezi sebou neudržují žádný invariant, stačí tedy klíč prvku x přepsat. Pokud je větší než klíč současného maxima, přepíše se ukazatel na maximum. Časová složitost je $O(1)$.

delete(Q, x) Mazání libovolného prvku probíhá naprosto stejně jako mazání maxima s dvěma výjimkami. Pokud daný prvek nebyl největší, nemusí se hledat nové maximum a postačuje tak konstantní čas. Pokud byl maximální prvek na konci a byl jím nahrazen vymazaný prvek, je nutné ještě aktualizovat ukazatel na maximum.

1.2.1.2 Seřazené pole

V neseřazeném poli musí být vždy nové maximum nalezeno projitím všech prvků. Tento nedostatek se dá odstranit tím, že prvky pole budou po každé operaci udržovány seřazené. To ale na druhou stranu zvýší náročnost vkládání.

Pole bude seřazené vzestupně, protože fakt, že maximální prvek bude na poslední pozici a nikoliv na té první, je výhodný. Ukazatel na maximální prvek je tedy vždy ukazatel na poslední prvek pole.

insert(Q, k) Nový prvek se vloží na poslední pozici, což zabere $O^*(1)$ času. Poté je nutné opravit seřazení pole. Zkontroluje se tedy, jestli je nový prvek menší než předchozí prvek. Pokud je větší nebo roven nebo bylo dosaženo začátku pole, algoritmus skončí. Pokud je menší, tyto dva prvky jsou prohozeny a pokračuje se porovnáním s dalším předchozím prvkem. Nový prvek takto „probublá“ na správné místo. V nejhorším případě (pokud je vložen prvek, který je menší než všechny stávající), bude provedeno n porovnání a prohození. Časová složitost této operace je tedy $O(n)$.

extractMax(Q) Smazání maxima ze seřazeného pole je triviální, protože největší prvek je vždy na poslední pozici. Stačí pouze zmenšit pole o 1, což znamená, že operace má konstantní časovou složitost.

merge(Q1, Q2) Nejjednodušší přístup ke slučování dvou seřazených polí a a b s prvky $a_1 \leq a_2 \leq ... \leq a_n$ a $b_1 \leq b_2 \leq ... \leq b_m$ je zavolat m krát operaci vkládání na a. To by ale vedlo na složitost $O(m \cdot n)$. Jak bude níže ukázáno, jde to rychleji s pomocí algoritmu slévání.

Použijí se dva ukazatele i a j. Na začátku se nastaví i na a_1 a j na b_1 a naalokuje se pole velikosti $n+m$. V každém kroku jsou porovnány prvky, na které ukazují i a j, menší u nich se zkopíruje do nového pole a příslušný ukazatel se posune o jeden prvek. Tento postup je opakován, dokud se nedojde na konec jedno z původních polí. Jakmile se tak stane, celý zbytek druhého pole (to bude obsahovat vždycky minimálně jeden prvek) se zkopíruje do výstupního pole.

Poznámka: tento postup je použit například jako subrutina v algoritmu Mergesort v [1].

Věta 4: *Algoritmus slévání vyrobí ze dvou seřazených polí a a b v čase $O(n + m)$ seřazené pole obsahující všechny prvky a a b.*

Důkaz. Pro spor předpokládejme, že výstupní pole není korektně seřazené, tj. obsahuje dva za sebou jdoucí prvky x a y, kde $x > y$. To, že prvky jsou za sebou znamená, že na ně v jednu chvíli ukazovaly ukazatele i a j a algoritmus se musel rozhodnout, který z nich vložit do výstupního pole. V takové situaci je ale vždy vybrán menší prvek, což je v tomto případě y a taková situace tudíž nemohla nastat, čímž jsme došli ke sporu.

Na každý prvek původních polí bude některý z ukazatelů ukazovat pouze jednou. Prvek následně přesuneme do výstupu a už s ním nijak nemanipulujeme. S každým prvkem tedy strávíme konstantně mnoho času. Celková časová složitost je $O(n + m)$. □

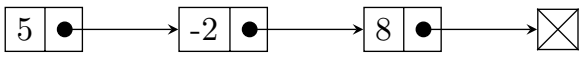
increaseKey(Q, x, k) Po zvětšení klíče je nutné pro prvek najít nové místo. Použije se stejný postup jako v případě vkládání nového prvku, ale prvek se tentokrát bude posouvat doprava. Časová složitost je rovněž $O(n)$.

delete(Q, x) Smazaný prvek nelze jednoduše nahradit posledním prvkem jako v případě neseřazeného pole. Místo toho musí být všechny následující prvky posunuty o jednu pozici doleva. To vede na lineární časovou složitost v nejhorším případě.

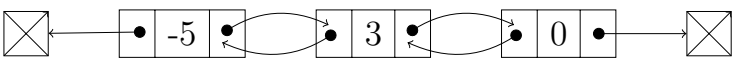
1.2.2 Spojové seznamy

Spojový seznam je jednoduchá spojová datová struktura. To dle [2] znamená, že hodnoty nejsou v paměti uloženy bezprostředně za sebou jako ve spojitých strukturách, ale jsou uchovávány na různých místech v takzvaných uzlech, které kromě ní obsahují také ukazatel na jeden nebo více dalších uzlů.

Pro tuto práci jsou zajímavé dva hlavní typy spojových seznamů a to jednosměrně zřetězený a obousměrně zřetězený spojový seznam. V jednosměrně zřetězeném seznamu obsahuje každý uzel ukazatel na uzel následující a pro přístup k celému seznamu si stačí pamatovat adresu prvního uzlu. Obousměrně zřetězený seznam se pak skládá z uzlů, které obsahují i druhý ukazatel – na předchozí uzel [2]. Pokud jsou udržovány ukazatele na začátek i na konec seznamu, může být procházen v obou směrech.



Obrázek 1.2: Jednosměrně zřetězený seznam.



Obrázek 1.3: Obousměrně zřetězený seznam.

U pole bylo dlouze rozebíráno, jak umožnit jeho proměnlivou velikost pomocí tzv. dynamického pole. Nic takového ale není u spojových seznamů

potřeba, ty se mohou libovolně zvětšovat a zmenšovat bez nutnosti jakéhokoli přesouvání stávajících prvků a velikost je omezena jen dostupnou pamětí. To patří také mezi jejich největší výhody oproti jednoduššímu poli. Další výhodou je například fakt, že vkládání na libovolnou pozici v seznamu je snazší, jak bude rozebráno dále.

Mezi největší nevýhody spojových seznamů patří nemožnost efektivního přístupu k libovolnému prvku a horší využití skrytých pamětí (cache). Kvůli ukazatelům taktéž zabírá více paměti než pole [2].

U pole stačilo znát index největšího prvku, což by zde bylo velmi neefektivní, takže bude udržován ukazatel na uzel s největší hodnotou.

1.2.2.1 Neseřazený spojový seznam

Bude uvažován obousměrně zřetězený seznam, což usnadní operaci mazání maxima. Stejně jako v neseřazeném poli nebude mezi prvky udržován žádný invariant.

insert(Q, k) Nový prvek lze vkládat na začátek nebo na konec seznamu se stejnou časovou složitostí. V tomto případě budou nové prvky vkládané na začátek.

Vytvoří se nový uzel a jeho ukazatel na další uzel se nastaví na stávající začátek seznamu (jehož ukazatel na předchozí uzel bude taktéž upraven). Pokud je nově vložená hodnota větší než dosavadní největší hodnota, aktualizuje se ukazatel na maximum. Na to očividně stačí konstantní množství času.

extractMax(Q) Uzel s maximální hodnotou se ze seznamu vypojí tak, že se upraví ukazatele jeho sousedů. Díky tomu, že je použit obousměrně zřetězený seznam, je známý předchůdce uzlu a není nutné ho hledat. Pokud bylo maximum na začátku nebo konci seznamu, aktualizují se i odpovídající ukazatele. Poté se musí najít nové maximum, což nelze udělat jinak, než projitím celého seznamu. Operace `extractMax(Q)` má tedy časovou složitost $O(n)$.

merge(Q1, Q2) Při slučování dvou neseřazených seznamů je stačí jednoduše spojit za sebe a ukazatel na maximum nastavit na větší z maxim obou seznamů. Časová složitost je $O(1)$.

increaseKey(Q, x, k) Stačí pouze změnit hodnotu daného uzlu a podívat se, jestli se v něm nyní nenachází maximum, což trvá $O(1)$ času.

delete(Q, x) Operace mazání libovolného prvku se realizuje úplně stejně jako `extractMax(Q)`. Pokud není odstraňováno maximum, stačí $O(1)$ času, jinak je potřeba $O(n)$.

1.2.2.2 Seřazený spojový seznam

Stejně jako v poli je možné i ve spojovém seznamu udržovat prvky seřazené. Opět se bude jednat o obousměrně zřetězený seznam, který bude řazen sestupně. Pokud by nebylo potřeba implementovat i operaci `delete(Q, x)`, stačilo by použít jednosměrně zřetězený seznam.

insert(Q, k) Při vkládání prvku je pro něj potřeba najít vhodnou pozici. Seznam bude procházen od začátku a jakmile bude nalezen uzel u, který nemá následníka nebo jehož následník má hodnotu menší než nově vkládaný prvek, nový uzel se vloží za u. V nejhorším případě bude prohledán celý seznam, časová složitost je tedy $O(n)$.

extractMax(Q) Největší prvek se odstraní za konstantní čas odpojením prvního uzlu a aktualizováním ukazatele na začátek seznamu.

merge(Q1, Q2) Pro sloučení dvou seřazených spojových seznamů je možné použít stejný algoritmus jako v případě seřazených polí, tentokrát si ale vystačí s konstantním množstvím pomocné paměti [4].

increaseKey(Q, x, k) Uzel se zvětšenou hodnotou je potřeba přesunout doleva na správnou pozici, což zabere $O(n)$ času.

delete(Q, x) Při mazání libovolného prvku stačí přepojit ukazatele sousedních uzlů, ostatní prvky se nemusí přesouvat. Složitost operace je díky použití obousměrně zřetězeného seznamu vždy $O(1)$ (jinak by bylo nutné najít předchozí uzel projitím celého seznamu).

1.2.3 Stromy

Stromy jsou další skupinou datových struktur použitelných pro implementaci prioritní fronty. Nejdříve bude ujasněna terminologie, [6] uvádí následující:

Definice 1 (Strom)**:** *Jako **strom** je označen souvislý graf, který neobsahuje žádnou kružnici.*

Definice 2 (Zakořeněný graf)**:** *Dvojice (G, v), kde graf G = (V, E) a v je zvolený vrchol, se nazývá **zakořeněný graf** s kořenem v.*

A pro účely prvních tří podkapitol se bude hodit ještě jedna definice z [3]:

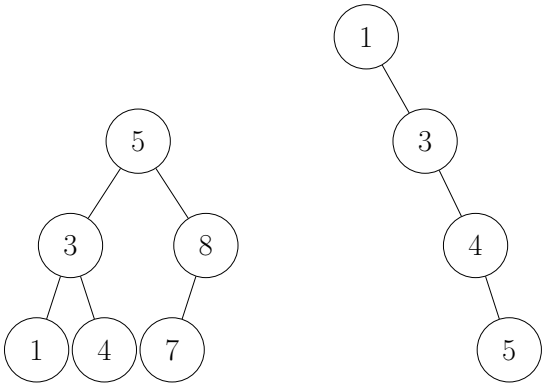
Definice 3 (Binární strom)**:** *Strom se nazývá **binárním**, pokud je zakořeněný a každý vrchol má nejvýše dva syny, u nichž se rozlišuje, který je levý a který pravý.*

1.2.3.1 Binární vyhledávací strom

Definice 4 (Binární vyhledávací strom)**:** ***Binární vyhledávací strom*** *je binární strom, jehož každý uzel obsahuje klíč a který splňuje podmínku, že klíč v každém uzlu u je větší než klíče všech uzlů v levém podstromu u a zároveň je menší než klíče všech uzlů v pravém podstromu u [5].*

BVS je nejjednodušší strukturou popisovanou v této kapitole. Je možné ho naimplementovat jako spojovou datovou strukturu. Ukazatel na maximum bude vždy ukazovat na nejpravější uzel stromu, v něm se nachází největší klíč. [2]

Definice 4 je poměrně benevolentní, to znamená že BVS mohou vypadat různě.



Obrázek 1.4: Příklady binárních vyhledávacích stromů

Při pohledu na tyto obrázky je vidět, že BVS může v extrémním případě zdegradovat až na spojový seznam. Není těžké si rozmyslet, že vnitřní struktura bude mít velký vliv na rychlost operací.

insert(Q, k) Vložení nového prvku do prázdného stromu je triviální. Pouze se vytvoří nový uzel s daným klíčem.

V opačném případě je vkládaný klíč porovnán s klíčem kořene a na základě výsledku tohoto porovnání je stejným způsobem vložen do levého (nový klíč je menší nebo roven klíči kořene) nebo do pravého (nový klíč je větší) podstromu.

Je tedy potřeba až tolik zanoření, kolik je hloubka stromu. A ta může být nejvýše rovna počtu uzlů. Časová složitost vkládání do BVS je tedy $O(n)$ [2].

extractMax(Q) Obecně při mazání uzlu z BVS mohou nastat tři případy podle počtu synů, které uzel má. Maximum ovšem nikdy nemá pravého syna (ten by obsahoval ještě větší klíč), takže případ s dvěma syny v tomto případě není třeba uvažovat.

Pokud nemá uzel žádného syna, je rovnou smazán.

V případě jednoho syna je uzel odstraněn a nahrazen svým synem.

V obou situacích je poté nutné aktualizovat ukazatel na maximum. Novým maximem bude předchůdce současného největšího prvku. Tím je nejpravější prvek jeho levého podstromu a pokud levý podstrom neexistuje, pak je předchůdcem uzlu jeho rodič. Pokud ani rodič neexistuje, znamená to, že strom obsahoval jen jeden prvek.

Co se týče složitosti operace `extractMax(Q)`, samotné smazání znamená pouze přepsat několik ukazatelů, ale najít nové maximum může (jak je uvedeno v [1]) trvat dobu lineární s hloubkou stromu, což je v nejhorším případě až $O(n)$ času. Pokud by nebyl explicitně udržován ukazatel na maximum, stejně by bylo nutné ho vždy před mazáním nejdříve najít, takže by nedošlo k asymptotickému zrychlení.

merge(Q1, Q2) Klíče z obou BVS se uloží do polí ve vzestupném pořadí. Toho se dá docílit tak, že se strom projde *in-order*, tedy začne se v kořeni a rekurzivně jsou nejprve uloženy klíče z levého podstromu, potom klíč z kořene a nakonec klíče z pravého podstromu. Obě pole jsou následně sloučeny algoritmem slévání z kapitoly 1.2.1.1.

Tímto postupem vznikne seřazené pole obsahující klíče z obou stromů, ze kterého se dá dle [4] v čase $O(m + n)$ postavit BVS následujícím způsobem.

Prázdné pole reprezentuje prázdný strom. Prostřední prvek pole se vloží do kořene. Poté je stejným způsobem rekurzivně postaven levý podstrom z první a pravý podstrom z druhé poloviny pole.

increaseKey(Q, x, k) Zvětšení klíče libovolného uzlu se dá implementovat jako volání operace `delete(Q, x)` a následné volání `insert(Q, k)` s novým klíčem. Obě tyto operace mají v nejhorším případě složitost lineární s počtem uzlů a tedy i operace zvětšení klíče má časovou složitost $O(n)$.

delete(Q, x) Při mazání libovolného uzlu u může oproti mazání maxima nastat ještě jedna situace - uzel může mít dva syny. Tentokrát není možné u smazat přímo, protože syny by nebylo kam připojit. Místo toho je nalezen jeho následník (nejlevější uzel v pravém podstromu - [3]), který má určitě nejvýše jednoho syna. Klíč následníka je zkopírován do u a následník je smazán tak, jak bylo popsáno u operace `extractMax(Q)`. S touto operací má i `delete(Q, x)` shodnou časovou složitost.

Jak již bylo zmíněno, od hloubky BVS se odvíjí rychlost základních operací `insert(Q, k)` a `extractMax(Q)`. Hloubka může být mezi $\log n$ a n. Dolní hranice plyne přímo z lemma 1 zlogaritmováním obou stran nerovnice. Žádoucí je tedy udržet co nejmenší hloubku stromu a dosáhnout tak až logaritmické

časové složitosti. Nabízí se například udržovat strom tzv. *dokonale vyvážený*, což je pojem definovaný v [3].

Definice 5 (Dokonale vyvážený strom)**:** *Binární vyhledávací strom je* ***dokonale vyvážený****, pokud pro každý jeho vrchol v platí*

$$||L(v)| - |P(v)|| \leq 1$$

Jinými slovy počet vrcholů levého a pravého podstromu se smí lišit nejvýše o 1.

Takový strom má hloubku $\lfloor \log_2 n \rfloor$, ovšem tento požadavek je velmi striktní a po každém vkládání nebo mazání je nutné zajistit, aby byl BVS i nadále dokonale vyvážený. To s sebou bohužel nese nepříjemnou skutečnost, jak uvádí [3].

Věta 5: *Pro každou implementaci operací insert a delete udržujících strom dokonale vyvážený platí, že insert nebo delete trvá* $\Omega(n)$ *pro nekonečně mnoho různých* n*.*

Důkaz. Nejprve si představíme, jak bude vypadat dokonale vyvážený BVS s klíči $1, ..., n$, kde $n = 2^k - 1$. Tvar stromu je určen jednoznačně: Kořenem musí být prostřední z klíčů (jinak by se levý a pravý podstrom kořene lišily o více než 1 uzel). Levý a pravý podstrom proto mají právě $(n-1)/2 = 2^{k-1} - 1$ uzlů, takže jejich kořeny jsou opět určeny jednoznačně a tak dále. Navíc si všimneme, že všechna lichá čísla jsou umístěna v listech stromu.

Nyní na tomto stromu provedeme následující posloupnost operací:

```
insert(n + 1), delete(1), insert(n + 2), delete(2), ...
```

Po provedení i-té dvojice operací bude strom obsahovat hodnoty $i+1, ..., i+n$. Podle toho, zda je i sudé nebo liché, se budou v listech nacházet buď všechna sudá, nebo všechna lichá čísla. Pokaždé se proto všem uzlům změní, zda jsou listy, na což je potřeba upravit $\Omega(n)$ ukazatelů. Tedy aspoň jedna z operací `insert(Q, k)` a `delete(Q, x)` trvá $\Omega(n)$ [3]. □

V dalších kapitolách budou ukázány méně striktní požadavky na vyvážení BVS, které skutečně povedou na logaritmickou složitost základních operací.

1.2.3.2 AVL strom

AVL stromy jsou podle [7] nejstarším a stále nejvíce používaným typem vyvažovaných stromů, hloubka stromu s n uzly je nejvýše $1,44 \cdot \log_2 n$ (důkaz alespoň asymptotické hloubky je uveden níže).

Definice 6 (AVL strom)**:** *Strom je hloubkově vyvážený, pokud pro každý vnitřní uzel platí, že hloubka levého podstromu a hloubka pravého podstromu se liší nejvýše o 1. Takový strom je často označován jako* ***AVL strom*** *[7].*

Věta 6: *AVL strom na n vrcholech má hloubku* $\Theta(\log n)$.

Důkaz. Nejprve pro každé $h \geq 0$ stanovíme A_h, což bude minimální možný počet vrcholů v AVL stromu hloubky h, a dokážeme, že tento počet roste s hloubkou exponenciálně.

Očividně platí, že $A_0 = 1$ (samotný kořen) a $A_1 = 2$ (kořen s jedním synem). Uvažujme jak vypadá minimální AVL strom pro větší h. Jeho kořen musí mít dva podstromy, jeden z nich hloubky $h-1$ a druhý hloubky $h-2$. Oba tyto podstromy musí být minimální AVL stromy dané hloubky. Musí tedy platit $A_h = A_{h-1} + A_{h-2} + 1$.

Platí, že $A_h \geq 2^{h/2}$ (nebude zde dokazováno). Jinými slovy víme, že platí $A_h \geq c^h$ pro konstantu $c = \sqrt{2}$. Proto AVL strom o n vrcholech může mít nejvýše $\log_c n$ hladin – kdyby jich měl více, obsahoval by více než $c^{\log_c n} = n$ vrcholů.

Na druhou stranu největší možný AVL strom hloubky h je úplný binární strom s $2^{h+1} - 1$ vrcholy. Tudíž minimální možná hloubka AVL stromu je $\Omega(\log n)$ [3]. □

Tentokrát bude u každého uzlu podstatná i *hloubka stromu*, jehož je tento uzel kořenem. Uzel bez synů má hloubku 0 a prázdný strom má hloubku -1.

insert(Q, k) Nový prvek se vloží stejným způsobem jako v BVS. Je ale možné, že tato operace rozbila hloubkové vyvážení a bude potřeba ho opravit. K zjištění stavu vyvážení se používá *znaménko*, které se spočítá pro každý uzel jako $sign(u) = h(t(u), h(l(u))$ [3].

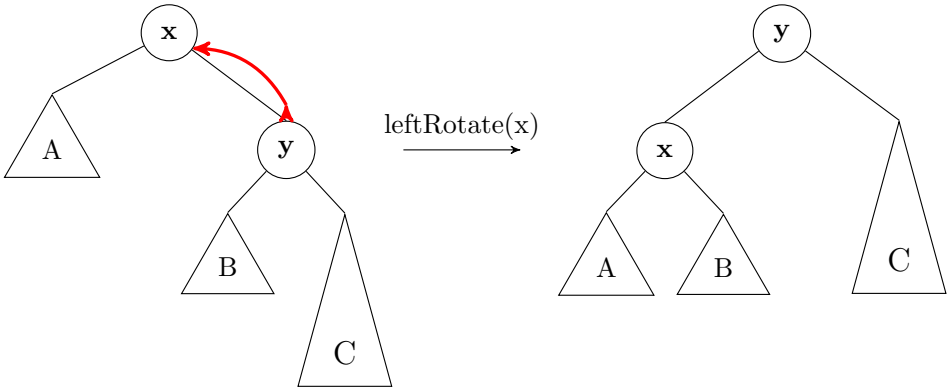
Nyní budou uzly procházeny od nově vloženého směrem ke kořenu (což se velmi snadno dělá při návratu z rekurze) a budou se počítat znaménka vrcholů po cestě. Pokud je $sign(u)$ v intervalu $[-1, 1]$, je příslušný strom s kořenem u hloubkově vyvážený a stačí aktualizovat údaj o hloubce a přesunout se do rodiče u. Hloubka se vždy přepočítává tímto způsobem:

$$height(u) = max(h(l(u)), h(r(u))) + 1$$

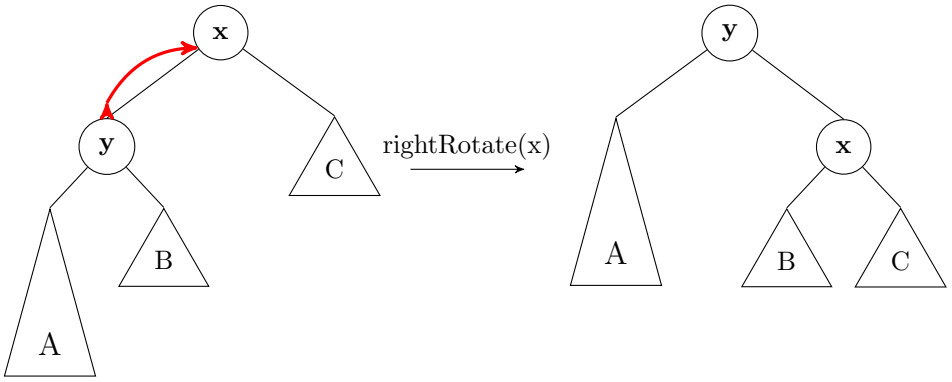
Pokud není znaménko v tomto intervalu, znamená to, že je potřeba vyvažovat. Vyvažování se v AVL stromu řeší tzv. *rotacemi*. Existují dva typy rotací – levá (L) a pravá (R).

V některých situacích ale jednoduché rotace problém nevyřeší. Tehdy je nutné použít dvojité LR nebo RL rotace. Na obrázku 1.7 je znázorněna LR rotace. Nejprve je provedena L rotace v uzlu y a následně R rotace v uzlu x. RL rotace by probíhala obdobně jen s opačnou dvojicí operací.

Typ použité rotace závisí na hodnotě znaménka, jak je rozepsáno v [3].



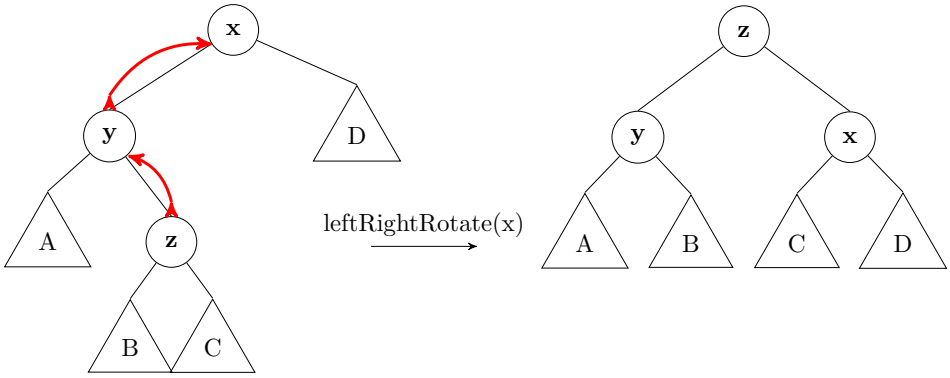
Obrázek 1.5: Levá (L) rotace.



Obrázek 1.6: Pravá (R) rotace.

- Pokud $sign(u) < -1$:
  - Pokud $sign(l(u)) = 1$, použije se LR rotace
  - Jinak se použije R rotace
- Pokud $sign(u) > 1$:
  - Pokud $sign(r(u)) = -1$, použije se RL rotace
  - Jinak se použije L rotace

V minulé kapitole bylo uvedeno, že rychlost vkládání do binárního vyhledávacího stromu závisí na jeho hloubce. Ta je u AVL stromu $\Theta(\log n)$. Prvek byl tedy vložen za $O(\log n)$ času. Při návratu z rekurze byly navštíveny uzly na cestě do kořene a v každém uzlu byla přepočítána hloubka a případně provedena některá rotace. Hloubky podstromů jsou ale uložené a není nutné je pokaždé počítat. Tím pádem strávil algoritmus v každém uzlu konstantní množství času a složitost celé operace je $O(\log n)$.



Obrázek 1.7: LR rotace.

extractMax(Q) a delete(Q, x) Logika je podobná jako v případě operace `insert(Q, k)`, nejprve se uzel smaže standardním způsobem a poté se budou směrem ke kořenu přepočítávat hloubky a kontrolovat vyvážení. Pro opravení vyvážení se používají rotace stejným způsobem, jako při vkládání. To zabere rovněž $O(\log n)$ času [3].

merge(Q1, Q2) Slučování dvou AVL stromů bude probíhat stejně jako v případě BVS. Strom vzniklý konstrukcí z pole klíčů je dokonale vyvážený a tím pádem je i hloubkově vyvážený. Časová složitost je opět $O(n + m)$.

increaseKey(Q, x, k) I tato operace funguje stejně jako u BVS, avšak díky větší efektivitě operací `insert(Q, k)` a `delete(Q, x)` bude časová složitost pouze $O(\log n)$.

1.2.3.3 Červeno-černý strom

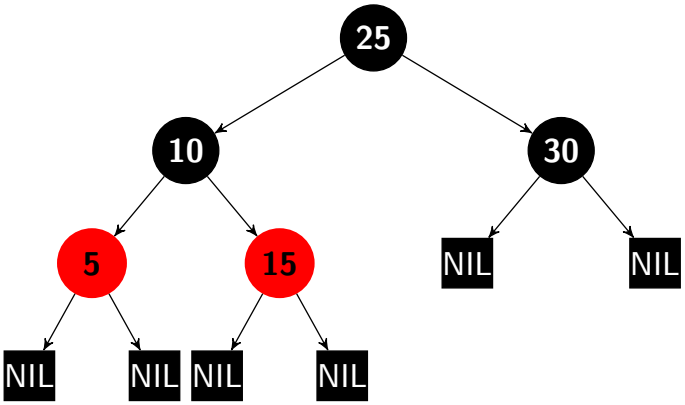
Použití AVL stromu samozřejmě není jediným způsobem, jak udržet BVS vyvážený. Podle [8] je jejich nevýhodou nutnost provést velké množství rotací při mazání. Nyní bude představen červeno-černý strom, který používá jinou strategii k obnovení vyvážení, byť asymptotické složitosti jsou stejné. Jeho definice uvedená v [1] zní následovně:

Definice 7 (Červeno-černý strom)**:** *Binární vyhledávací strom, jehož všechny uzly jsou obarveny buď červenou nebo černou barvou, se nazývá* ***červeno-černým stromem****, pokud splňuje tyto vlastnosti:*

1. *Kořen je černý.*
2. *Všechny listy (NIL) jsou černé.*
3. *Oba synové červeného uzlu jsou černí.*

4. *Pro každý uzel u platí, že všechny cesty z u do listu obsahují stejný počet černých uzlů.*

Je potřeba si vysvětlit, co znamená onen *NIL* zmíněný v definici. Jedná se o speciální uzel, který neobsahuje klíč. Listy stromu (a také rodič kořene) budou výhradně *NIL* uzly. Ostatní uzly jsou nazvány *vnitřními uzly*. To je změna oproti dříve představeným stromovým strukturám. Tento pohled ale zjednoduší popis některých operací.



Obrázek 1.8: Příklad červeno-černého stromu včetně NIL listů

Definice není na první pohled tak jasná jako v případě AVL stromu. K udržování správné struktury nestačí pouze rotace. Popis i implementace základních operací (vkládání a mazání) je o něco delší, jak bude záhy ukázáno.

Nyní ale bude dokázáno, že výše uvedená definice rovněž zaručuje příznivou hloubku stromu. Věta i důkaz jsou k nalezení v [1]. Pokud se mluví o *černé hloubce* uzlu u (bude značeno jako $bh(u)$), myslí se tím počet černých uzlů (vyjma u) na cestě z u do listu.

Věta 7: *Červeno-černý strom s n vnitřními uzly má hloubku nejvýše* $2\log(n+1)$.

Důkaz. Nejdříve dokážeme indukcí podle hloubky, že podstrom T_u zakořeněný v uzlu u má alespoň $2^{bh(u)} - 1$ vnitřních uzlů.

Pokud je hloubka T_u nulová, musí u být NIL. V tom případě má T_u určitě alespoň $2^{bh(u)} - 1 = 2^0 - 1 = 0$ vnitřních uzlů.

V indukčním kroku budeme uvažovat podstrom T_u, který má kladnou hloubku a u má dva syny. Červený syn bude mít černou hloubku $bh(u)$ a černý syn bude mít černou hloubku $bh(u) - 1$. Výška syna je určitě menší než výška samotného u, takže na něj můžeme aplikovat indukční předpoklad. Podle něj by oba synové měli mít alespoň $2^{bh(u)-1} - 1$ vnitřních uzlů. To znamená, že T_u má minimálně $(2^{bh(u)-1} - 1) + (2^{bh(u)-1} - 1) + 1 = 2^{bh(u)} - 1$ vnitřních uzlů, čímž je tento dílčí důkaz hotov.

Pro zbytek hlavního důkazu si označíme v jako výšku celého stromu. Podle podmínky 4 z definice 7, je aspoň polovina uzlů (kromě kořene) na cestě z kořene do listu černá. Černá výška kořene je tedy minimálně h / 2. Díky předchozímu dílčímu tvrzení můžeme říci, že pro počet uzlů n ve stromu platí:

$$n \geq 2^{h/2} - 1$$

$$n + 1 \geq 2^{h/2}$$

$$\log(n + 1) \geq h \,/\, 2$$

$$h \leq 2 \log(n + 1)$$

A tím je důkaz hotov. □

insert(Q, k) Nový uzel x se vloží místo nějakého listu standardním algoritmem na vkládání do BVS. Poté mu bude nastavena *červená* barva a jako syny mu budou dány dva NIL uzly. Teď je ale nutné opravit strukturu, aby platily všechny podmínky uvedené v definici 7. Podmínka 2 určitě není porušena, protože byly přidány dva nové černé listy. Rovněž tak podmínka 4 určitě stále platí, přidáním červeného uzlu místo listu se nezměnila černá hloubka žádného uzlu. Podmínky 1 a 3 ale být porušeny mohly. Postup opravy se dělí do několika případů tak, jak jsou uvedeny v [1]. *Strýcem* uzlu x se chápe sourozenec jeho rodiče a značí se u. Rodič x je pak značen jako p.

Pokud je x kořenem stromu, stačí ho přebarvit na černo a algoritmus může skončit. Stejně tak, pokud je p černý, je hotovo. Nyní bude rozebrán nejsložitější případ, kdy je p červený a není tak dodržena podmínka 3.

1. *u je červený.* Uzly p a u se obarví černě a rodič p červeně. To lze bez problémů udělat, protože v daném podstromu se tím žádná podmínka neporuší. Tím se vyřešil problém, že x i p jsou červení. Tyto změny ale mohly způsobit porušení podmínky 3 výše ve stromu. Proto se bude celý postup opravy aplikovat na rodiče p, který bude novým x. To tedy znamená, že se algoritmus přesune o dvě úrovně výše.

2. *u je černý a x je pravým synem p.* V tomto případě se algoritmus přesune do rodiče. Ukazatel x se tedy přepíše na p. Následně v x dojde k provedení levé rotace (rotace v červeno-černém stromu vypadají stejně jako rotace v AVL stromu). Uzel x je nyní určitě levým synem svého rodiče. Dále se pokračuje případem 3.

3. *u je černý a x je levým synem p.* Do tohoto případu se dá dostat rovnou nebo přes případ 2. Každopádně se uzel p obarví černě a rodič p červeně. Poté se provede pravá rotace v rodiči p.

Celý dosavadní postup počítá s tím, že p je levým synem svého rodiče. Pokud by byl pravým synem, všechny operace proběhnou symetricky. To znamená, že se změní směry rotací atd.

Co se týče analýzy této operace, tak $O(\log n)$ času zabere samotné vložení uzlu. Dále [8] uvádí, že při vkládání se provede $O(\log n)$ přebarvení uzlu a pouze konstantní množství rotací (nejvýše jedna). Celková časová složitost operace je tedy $O(\log n)$.

extractMax(Q) a delete(Q, u) Tyto dvě operace budou popsány dohromady, protože mazání maxima se od mazání ostatních prvků nijak neliší. Pouze je nutné najít jeho předchůdce a aktualizovat ukazatel na maximum. Mazaný uzel je výjimečně označený jako u, aby nedocházelo ke zmatení ve značení.

Podle [1] se nejprve postupuje jako při mazání z obyčejného BVS. Tento algoritmus ale pracuje s počtem synů. Proto musí být modifikován tak, aby NIL uzly nepočítal jako syny.

Nechť je x uzel, ve kterém mohlo dojít k porušení podmínek. Může se jednat o NIL uzel, který nahradil u, v případě, že u neměl žádného syna. Nebo může x být jediný syn u. Pokud měl u dva syny, potom se jedná o pravého syna jeho následníka (což může být i NIL). Na začátku mazání je důležité si také poznamenat původní barvu u (nebo barvu jeho následníka, pokud má u dva syny).

Pokud tato původní barva byla červená, nemohlo dojít k porušení žádné podmínky a operace je dokončena. Pokud ale byla černá, je situace složitější.

Nadále bude p označovat rodiče x a s sourozence x. Pokud je x kořen stromu nebo je červený, je operace dokončená. Jinak mohou nastat 4 případy [1]:

1. *s je červený.* V tomto případě je určitě p černý a oba synové s jsou černí. Prohodí se tedy barvy p a s a následně se provede levá rotace v p. Nový sourozenec s (jeden ze synů původního s) je nyní černý a může nastat některý z dalších případů.

2. *s je černý a oba jeho synové jsou také černí.* Uzel s bude přebarven na červeno a celý postup opravy se opakuje pro nové $x = p$.

3. *s je černý, jeho levý syn je červený a pravý syn je černý.* Zamění se barvy s a jeho levého syna. Poté bude provedena pravá rotace v s. Nové s má nyní černou barvu a jeho pravý syn je červený. Tím pádem algoritmus pokračuje případem 4.

4. *s je černý a jeho pravý syn je červený.* Barva s se nastaví na barvu p. Následně dojde k obarvení p a a pravého syna s černě. Dalším krokem je levá rotace v p. Nakonec se nastaví x na kořen stromu, čímž celý algoritmus v další iteraci skončí.

Tento postup platí, pokud je x levým synem svého rodiče. V opačném případě jsou všechny operace symetrické.

Podle [8] se při mazání se provede $O(\log n)$ přebarvení uzlu a nejvýše dvě rotace. Celková časová složitost mazání libovolného uzlu i mazání maxima je $O(\log n)$.

merge(Q1, Q2) Strom se zkonstruuje stejným způsobem jako v případě klasického BVS. Zbývá se ale rozhodnout, jak uzly obarvit. Pokud by všechny uzly byly černé, nemusela by platit podmínka 4. Proto se uzly v poslední hladině obarví červeně (kromě případu kdy je v poslední hladině i kořen). To zaručí platnost všech podmínek.

Časová složitost `merge(Q1, Q2)` je lineární vzhledem k počtu prvků obou stromů.

increaseKey(Q, x, k) Zvětšení klíče se opět realizuje pomocí smazání uzlu a jeho vložení s novým klíčem. Díky časové složitosti těchto dvou operací zabere zvětšení klíče $O(\log n)$ času.

1.2.3.4 (a-b)-strom

Zatím byli představeni pouze zástupci binárních stromů. Existují ale i stromy, které dovolují uzlu mít více klíčů a synů, aby si udržely příznivou hloubku. Jedná se o (a-b)-stromy, které jsou definované v [3].

Definice 8 (Obecný vyhledávací strom)**:** ***Obecný vyhledávací strom*** *je zakořeněný strom s určeným pořadím synů každého uzlu. Uzly se dělí na vnitřní a vnější, přičemž platí:*

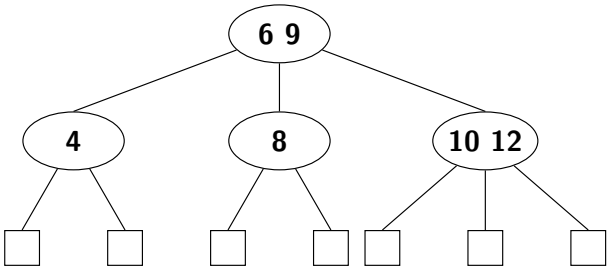
Vnitřní uzly mají jeden nebo více klíčů. Uzel s klíči $x_0 < ... < x_k$ má $k+1$ synů označených $s_0, ..., s_k$. Pro účely definice platí $x_0 = -\infty$ a $x_{k+1} = \infty$. Klíče slouží jako oddělovače hodnot v podstromech, to znamená, že pro každý klíč h z $T(s_i)$, pro $i \in 0, ..., k$, platí že $x_i < h < x_{i+1}$. $T(s_i)$ označuje podstrom zakořeněný v s_i.

Vnější uzly neobsahují žádné klíče a nemají žádné syny, jedná se tedy o listy stromu [3].

Definice 9 ((a-b)-strom)**:** *Jako* ***(a-b)-strom*** *s parametry $a \geq 2, b \geq 2a - 1$ se označuje obecný vyhledávací strom, pro který platí:*

- *Kořen má 2 až b synů, ostatní vnitřní uzly mají a až b synů.*
- *Všechny vnější uzly (listy) jsou ve stejné hloubce.*

Mezi často popisované varianty patří (2-3)-stromy [5] nebo (2-4)-stromy [8]. Dříve zmíněné červeno-černé stromy se někdy definují s pomocí (2-4)-stromů, jak je uvedeno v [8].



Obrázek 1.9: Příklad (2-3)-stromu. Bílé čtverečky reprezentují vnější uzly.

I tu tohoto typu stromu je důležité si ukázat, že garantuje logaritmickou hloubku, [3] říká, že:

Věta 8: *(a-b)-strom s n klíči má hloubku* $\Theta(\log n)$.

Důkaz probíhá velmi podobně jako důkaz hloubky AVL stromu a nebude zde uveden. Je k dispozici v [3], přesné odhady jsou pak popsány v [9].

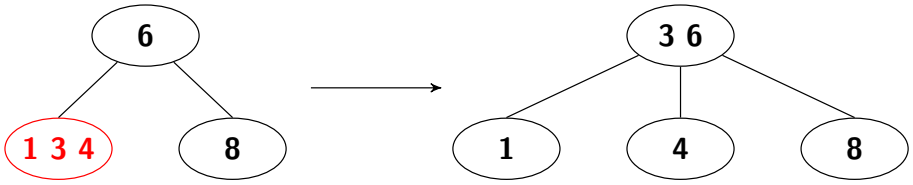
insert(Q, k) Jelikož v této práci je v prioritní frontě umožněna přítomnost více stejných klíčů, je nutné přizpůsobit si definici 8. Mezi klíči v uzlu bude tedy platit neostrá nerovnost. Dále bude platit, že pokud $x_i = k$, tak se při vkládání klíče k algoritmus zanoří do podstromu $T(s_{i+1})$ (pravého syna z pohledu x_i) a nikoliv $T(s_i)$.

Dle [3] se postupuje jako při vkládání do BVS s rozdílem, že mohou být i více než 2 možnosti, kam se zanořit. Nakonec algoritmus skončí v nějakém listu l. Pokud by byl místo l vložen nový vnitřní uzel s jediným klíčem a dvěma listy jako syny, mohlo by dojít k porušení podmínky o minimálním počtu synů i o stejné hloubce listů.

Místo vytváření nového uzlu se vloží nový klíč a jeden nový syn (list) do otce l (dále jako p). Pokud v p i nadále platí podmínka, že má maximálně b synů, algoritmus končí. V opačném případě se uzel p rozdělí na tři části - prostřední klíč x_m, levou část (levá půlka klíčů a synové mezi nimi) a pravou část. Z levé a pravé části vzniknou nové uzly u_1 a u_2. Následně se vloží do otce p na příslušné místo klíč x_m. Tím vzniknou místa pro dva nové syny. Na tyto místa se vloží u_1 a u_2. Tím ale mohlo dojít k přeplnění otce p, což je nutné opět zkontrolovat a případně opravit stejným způsobem.

Pokud se takto dojde až do kořene a kořen se rozdělá na dva uzly, vznikne nový kořen a celý strom se prohloubí o 1.

Strom má logaritmickou hloubku a algoritmus stráví v každé úrovni konstantní množství času (musí sice najít správnou pozici pro syna mezi všemi klíči, ale a a b jsou konstanty). Časová složitost operace `insert(Q, k)` je tedy i podle [9] $O(\log n)$.



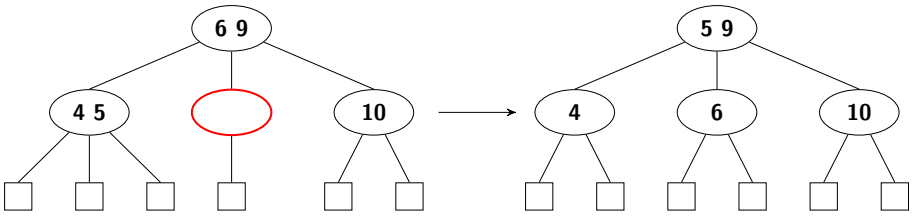
Obrázek 1.10: Příklad dělení uzlu v (2-3)-stromu (bez vnějších uzlů). Nejlevější uzel má příliš mnoho synů a je rozdělen.

delete(Q, x) Tato operace se díky možné přítomnosti více stejných klíčů ve stromu značně komplikuje. Dalším problémem je, že v kapitole 1.1.1 bylo řečeno, že operaci `delete` se předává ukazatel, zde však je v uzlu i více než 1 klíč. Nejdřív bude podle [3] popsáno mazání daného *klíče* k (nikoliv uzlu) z (a-b)-stromu, který vyžaduje unikátní klíče, a na konci bude uvedeno několik možností, jak tento postup upravit pro použití v tomto případě.

Nejdříve bude tedy mazaný klíč ve stromu vyhledán. Pokud se nachází v předposlední hladině, z příslušného uzlu se rovnou smaže klíč k a jeden syn. V opačném případě ho přímo smazat nelze, protože uzel by přišel o místo pro syna, kterým je vnitřní uzel. Místo toho se bude postupovat stejně jako u binárních stromů, to znamená, že se klíč nahradí svým následníkem, který se určitě nachází v předposlední hladině, a smaže se tento následník.

V obou případech se může stát, že v uzlu u, ze kterého byl klíč odebrán, zbude méně než a klíčů. V takovém případě se algoritmus podívá do levého bratra l uzlu (nebo do pravého, pokud levý bratr neexistuje). Nechť uzly u a s v otci odděluje klíč o.

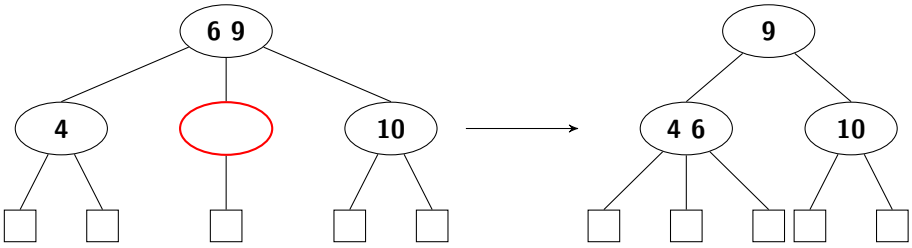
Nyní se podle počtu klíčů v l rozhodne, co dál. Pokud má l více než a synů, lze z něj odpojit nejpravějšího syna c a nejpravější klíč m. Klíč m se přesune do otce a o vloží se jako první klíč do u. Tím se v u vytvořilo místo pro nejlevějšího syna, na které se připojí c, čímž byl opraven uzel u a v l nedošlo k porušení žádné podmínky.



Obrázek 1.11: Příklad doplňování uzlu v (2-3)-stromu. Prostřední syn kořene má nedostatek synů a půjčí si syna od levého bratra.

Pokud má ale l pouze a synů, tento postup by způsobil nedostatek synů v l. Dojde proto ke sloučení uzlů u a l a do výsledného uzlu v se přidá ještě klíč o. Z otce byli tedy odebráni dva synové a jeden klíč, takže zbývá místo

pro jednoho syna. Na toto místo se připojí v. Teď ale může nastat nedostatek synů otce, bude na něj tedy aplikován stejný postup.



Obrázek 1.12: Příklad slučování uzlů v (2-3)-stromu. Prostřední syn kořene má nedostatek synů a je sloučen se svým levým bratrem.

Tímto způsobem lze dojít až do kořene. Pokud byli slučováni jeho synové, mohlo se stát, že kořen nyní neobsahuje žádný klíč a má jen jednoho syna. V takovém případě se kořen smaže a novým kořenem se stane právě tento syn. Hloubka stromu se pak sníží o 1.

Smazat libovolný klíč lze za $O(\log n)$ času dle [9].

Zbývá vyřešit problém zmíněný na začátku této sekce. Není možné předávat pouze ukazatel na uzel, algoritmus potřebuje vědět i který klíč má smazat. Jedno z možných řešení by bylo speciálně pro (a-b)-stromy předefinovat operaci `delete` tak, aby jako parametr měla klíč a nikoliv ukazatel. V kapitole 1.1.1 již bylo zmíněno, jaké toto řešení přináší nevýhody. Předávání jak ukazatele na uzel, tak i klíče má úplně stejná úskalí, navíc by bylo nutné přidat do uzlů ukazatele na rodiče. Bude tedy uvažována varianta s předávání klíče a smíříme se s faktem, že se může smazat libovolný prvek s daným klíčem.

extractMax(Q) Mazání maxima je jednodušší v tom, že není nutné vyhledávat správného syna, ale stačí se rovnou zanořit do nejpravějšího syna. Jakmile se algoritmus dostane do nejpravějšího uzlu stromu, smaže mu poslední klíč. Za povšimnuté stojí, že vrcholy, které pak budou procházeny při cestě směrem ke kořeni nikdy nemají pravého syna. Časová složitost mazání maxima je tedy rovněž $O(\log n)$.

merge(Q1, Q2) U binárních stromů vždy stačila konstrukce dokonale vyváženého stromu ze seřazeného pole a jeho případnou lehkou úpravou. Dokonale vyvážený binární strom bohužel často není korektní (a-b)-strom (např. už při volbě $a > 2$). Pro sloučení dvou (a-b)-stromů se tedy bude opakovaně volat $insert(Q_1, k)$ pro každé $k \in Q_2$, což vede k linearitmické časové složitosti vzhledem k počtu prvků obou stromů.

increaseKey(Q, x, k) I zde nastává úplně stejný problém jako v případě mazání. Uvažována bude tedy varianta `increaseKey(Q, k, k')`, kdy bude

prvku s klíčem k nastaven klíč k' s tím, že nelze garantovat, na který z prvků s klíčem k, bude změna aplikována.

Jako u všech ostatních stromů i zde se použije dvojice operací mazání a vkládání, což zabere $O(\log n)$ času.

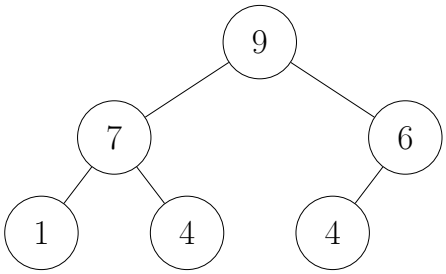
1.2.4 Haldy

Poslední velkou skupinou datových struktur použitelných pro implementaci prioritní fronty, které budou v této práci představeny, jsou haldy. Jak říká [7], jedná se o velmi používané struktury a někdy se dokonce pojmy *halda* a *prioritní fronta* zaměňují. Jednotlivé druhy hald se svojí vnitřní strukturou liší, ale všechny mají formu jednoho nebo více binárních, případně n-nárních stromů.

Bude se hodit vysvětlit si tzv. *haldové uspořádání* z [3]. V maximové verzi tato podmínka říká, že pokud je v uzel stromu a s je jeho syn, platí $k(v) \geq k(s)$.

1.2.4.1 Binární halda

Binární halda je nejjednodušším příkladem haldy. Dle [1] se na ni dá dívat jako na téměř úplný binární strom. Všechny hladiny stromu jsou kompletně zaplněny, až na poslední hladinu, která se plní zleva doprava.

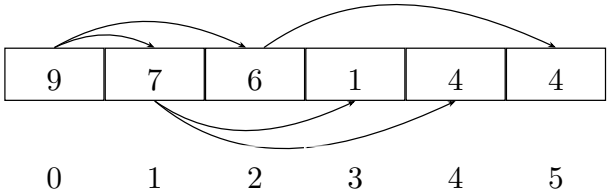


Obrázek 1.13: Binární halda se 6 uzly

Samozřejmě je možné binární haldu implementovat jako strom, avšak dá se reprezentovat i jednodušeji jako pole. Pokud se použije pole, pro uzel nacházející se v něm na pozici n (při indexování od 0) platí, že jeho rodič se nachází na pozici $\lfloor (n-1) \,/\, 2 \rfloor$, levý syn na $2 \cdot n + 1$ a pravý syn na pozici $2 \cdot n + 2$. Šipky na obrázku 1.14 reprezentují vztahy rodič - syn. Některá literatura, například [2] nebo [1], indexuje od 1, což v důsledku znamená, že pozice rodiče a synů se počítají lehce jinak.

Je dobré si uvědomit, jakou hloubku bude halda s n prvky mít.

Věta 9: *Binární halda s n prvky má hloubku $h = \lfloor \log n \rfloor$ [2].*



Obrázek 1.14: Halda z obrázku 1.13 reprezentovaná v poli.

Důkaz. Platí že

$$\sum_{i=0}^{h} 2^i \ = 2^{h+1} - 1 \geq n$$

Nerovnost plyne z faktu, že poslední hladina nemusí být zcela zaplněná a uzlů tedy může být méně. Z této nerovnice rovnou vidíme, že $h = \lfloor \log n \rfloor$ [2]. □

insert(Q, k) Vkládaný klíč se vloží do nového uzlu a ten je připojen na konec haldy. To znamená na první volnou pozici v poslední hladině, případně na první pozici nové hladiny, pokud je ta předchozí zaplněná. Efektivně to znamená přidat klíč na konec pole.

Tím ale mohlo být porušeno haldové uspořádání. To se opraví tak, že pokud je k větší než klíč otce nového uzlu, klíče se prohodí. Tím se ale problém nemusel vyřešit, takže se algoritmus přesune do rodiče a postup se opakuje. Naopak, pokud nerovnost daná haldovým uspořádáním platí nebo už se algoritmus nachází v kořeni, je hotovo. Na tuto proceduru bude dále odkazováno jako na `bubbleUp(x)`, kde x je uzel haldy.

Vložení prvku do pole trvá $O^*(1)$ času, jak už bylo dokázáno. Provedení `bubbleUp(x)` potom trvá $O(\log n)$. Při vložení nového maxima je nutné ho „probublat“ až do kořene, přičemž v každém uzlu se provede konstantní množství operací. Celková časová složitost je tedy logaritmická [5].

extractMax(Q) Maximum se vždy nachází v kořeni. Ten nebude odstraněn přímo, místo toho do něj bude zkopírován klíč z posledního uzlu u a smaže se u.

Tím opět mohla vzniknout nevalidní halda, což se opraví podobně jako v případě `insert(Q, k)`. Tentokrát se ale algoritmus bude dívat do synů. Buď x aktuální uzel a s ten z jeho synů, který má větší klíč. Pokud má s větší klíč než x, klíče se prohodí a stejný postup se opakuje pro s. Pokud už haldové uspořádání platí, nebo už žádný syn neexistuje, algoritmus končí. Tento postup bude dále označován jako `bubbleDown(x)`.

Smazání posledního prvku zabere pouze konstantní množství času, procedura `bubbleDown(x)` je na tom z hlediska složitosti stejně jako `bubbleUp(x)` a časová složitost operace je tedy opět logaritmická [5].

merge(Q1, Q2) Pokud jsou obě haldy reprezentované polem, tyto pole se pouze spojí za sebe do jednoho a z výsledného pole se postaví validní binární halda podle algoritmem `heapBuild` [2].

Zkusme se na pole podívat zezadu od posledního ($n - 1.$) prvku. Tento prvek reprezentuje uzel, který nemá žádné syny, tvoří tedy sám o sobě validní binární haldu. To samé platí pro posledních $\lceil n \,/\, 2 \rceil$ prvků, protože ty jsou také listy. Při procházení pole směrem dopředu časem narazíme na první prvek, který má alespoň jednoho syna. V tomto místě může být haldové uspořádání porušeno, ale synové jsou určitě validní binární haldy. Přesně na to se hodí procedura `bubbleDown(x)`. Stačí tedy provést $\lfloor n \,/\, 2 \rfloor$ netriviálních volání `bubbleDown(x)`.

Na první pohled to vypadá, že tato operace bude vyžadovat linearitmicky mnoho času. To je sice správný, avšak ne těsný odhad.

Věta 10: *Algoritmus `heapBuild` zavolaný na pole o velikosti n postaví binární haldu v čase $O(n)$.*

Důkaz. U každého prvku můžeme vykonat až $\lfloor \log n \rfloor$ operací. U většiny prvků ale zdaleka tolik času nestrávíme. Připomeňme si, že rychlost `bubbleDown(x)` ja závislá na hloubce haldy, na kterou je volána. Většina těchto hald je ale velmi malá. V úplném binárním stromu na n uzlech je $n/2$ listů (stromů výšky 0), $n \,/\, 4$ uzlů výšky 1, $n \,/\, 8$ výšky 2 atd. To velmi připomíná situaci jako při dokazování věty 3. I tady platí podobná nerovnost. Čas nutný pro postavení haldy je je:

$$\sum_{h=0}^{\lfloor logn \rfloor} \lceil n \,/\, 2^{h+1} \rceil h \leq n \sum_{h=0}^{\lfloor logn \rfloor} h \,/\, 2^h \leq 2n$$

[2] □

Z toho plyne časová složitost slučování dvou binárních hald $O(n + m)$.

Byly popsány i efektivnější způsoby slučování binárních hald. V [10] autoři přišli s algoritmem, kterému stačí $O(\log n \cdot \log m)$ času (při implementaci pomocí stromu, ne pole), v [11] je pak popsán způsob, jak operaci provést v čase $O(n + min(\log m \cdot \log \log m, \log n \cdot \log m))$. To je však dosti nad rámec této práce, zde popsaná binární halda si vystačí s výše popsaným způsobem.

increaseKey(Q, x, k) Haldy obecně nejsou narozdíl od stromů struktury příliš vhodné pro vyhledávání libovolného prvku. Pokud ale je ukazatel na prvek k dispozici, pro zvětšení klíče stačí klíč přepsat a poté zavolat `bubbleUp(x)`. Na to stačí $O(\log n)$ času.

delete(Q, x) Je několik způsobů, jak smazat libovolný uzel, dá se například postupovat obdobně jako při mazání maxima. To znamená, že se do x zkopíruje klíč z posledního uzlu, který se poté smaže. Následně se zkontroluje platnost

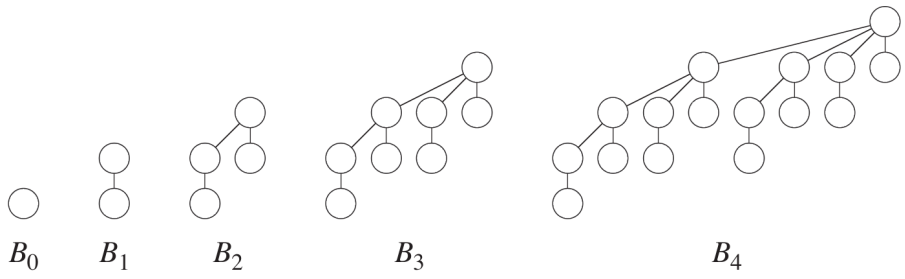
haldového uspořádání a případně se zavolá `bubbleUp(x)` nebo `bubbleDown(x)` podle výsledku porovnání $k(x)$ s klíči rodiče a synů. To zabere $O(\log n)$ času.

1.2.4.2 Binomiální halda

Binomiální halda je implementačně složitější než obyčejná binární halda, ale - jak bude vidět dále - nabízí efektivnější operaci slučování. Původně byla v [12] představená jako *binomiální fronta*, protože byla vytvořená speciálně pro použití jako prioritní fronta.

Nejdříve bude definována pomocná struktura, ze které je binomimální halda postavená.

Definice 10 (Binomiální strom řádu k)**: *Binomiální strom řádu*** *k, který bude značen jako B_k, je definovaný induktivně. B_0 je samotný kořen. B_k, kde $k > 0$, je definován jako B_{k-1}, k jehož kořenu je jako nejpravější syn připojený další B_{k-1}. [12]*



Obrázek 1.15: Příklady binomiálních stromů. Převzato z [1].

Definice 11 (Binomiální halda)**: *Binomiální halda*** *je dle [1] množina binomiálních stromů, pro kterou platí:*

- *V každém uzlu stromu se nachází klíč.*
- *V každém stromu mezi sebou uzly udržují haldové uspořádání.*
- *Pro každé $k \geq 0$ obsahuje halda maximálně jeden B_k.*

Maximum se vždy nachází v kořeni jednoho ze stromů.

Pro další analýzu se budou hodit dvě lemmata uvedená a dokázaná v [3]. V tomto textu nebudou jejich důkazy uvedeny.

Lemma 2: *Binomiální strom řádu k má 2^k vrcholů.*

Lemma 3: *Binomiální strom s n vrcholy má hloubku $O(\log n)$ a počet synů kořene je taktéž $O(\log n)$.*

merge(Q1, Q2) Trochu neobvykle bude nejdříve popsána operaci slučování dvou binomiálních hald, protože další operace ji využívají. K pochopení toho, jak tato operace funguje, je dobré si ukázat, že binomiální haldy mají souvislost s binárním zápisem čísel [3].

Lemma 4: *Binomiální halda s n prvky obsahuje binomiální strom řádu k právě tehdy, když je ve dvojkovém zápisu čísla n k-tý nejnižší bit roven 1.*

Důkaz. Binomiální halda H (neboli binomiální stromy, z nichž se skládá) obsahuje celkem $\sum_{i=0}^{k} b_i 2^i = n$ vrcholů, kde k je maximální řád stromu v H a b_i je rovno 1, pokud H obsahuje B_i a 0 jinak. Číslo $b_k b_{k-1} \cdots b_0$ je zápis čísla n ve dvojkové soustavě. Navíc je zápis čísla ve dvojkové soustavě jednoznačný pro každé n a tím pádem i hodnoty b_i (neboli to, které stromy jsou v H přítomny) jsou určeny jednoznačně [3]. □

Z tohoto lemmatu rovněž plyne, že v binomiální haldě s n prvky se nachází nejvýše $\lceil \log_2 n \rceil$ binomiálních stromů [3].

Pro pochopení slučování binomiálních hald je dobré si připomenout, jak funguje školní sčítání binárních čísel a a b „pod sebou“. Do každého řádu může přijít *carry* z nižšího řádu. Bit výsledného čísla se spočítá jako

$$c_i = (a_i + b_i + carry_i) \mod 2$$

a nový přenos do vyššího řádu

$$carry_{i+1} = (a_i + b_i + carry_i) \,/\, 2.$$

V praxi to znamená, že jsou tři vstupy do každého řádu. Pokud jsou všechny tři 0, napíše se do výsledku 0 a do přenosu taktéž. V případě jednoho vstupu s hodnotou 1 jde na výstup 1 a do přenosu 0. Pokud jsou dva vstupy nenulové, zapisuje se 0 a do přenosu jde 1. Pokud jsou 1 ve všech třech vstupech, na výstupu i v přenosu budou jedničky [3].

Díky platnosti lemmatu 4 lze naprosto stejně postupovat i při slučování binomiálních hald. Případy s žádným nebo jedním kladným sčítancem v řádu jsou jasné. Co to ale znamená sečíst dva kladné bity v řádu i a vytvořit tak přenos do vyššího řádu? To znamená sloučit dva stromy B_k^1 a B_k^2 a vytvořit tak jeden strom řádu $k + 1$. To znamená, že jeden z nich se připojí jako nejpravějšího syna kořene druhého. Což přesně odpovídá definici 10. Pouze musí být porovnány klíče kořenů a stromy se musí propojit tak, aby bylo dodrženo haldové uspořádání.

Složitost očividně závisí na počtu stromů v obou slučovaných haldách. Těch je, jak již bylo řečeno, logaritmicky mnoho vzhledem k počtu prvků. V každém řádu stromů se provede konstantně mnoho operací. Složitost celé operace `merge(Q1, Q2)` je tedy $O(\log(m + n))$ [4].

insert(Q, k) Po zadefinování operace `merge(Q1, Q2)` je vkládání nového prvku triviální. Z vkládaného klíče je vytvořena binomiální haldu o jednom prvku (neboli samotný strom B_0), která je následně sloučena s Q.

Analýza složitosti až tak jednoduchá není. Na první pohled lze říct, že díky logaritmické složitosti operace slučování zabere `insert(Q, k)` $O(\log n)$ času. To ale není těsný odhad. Dle [7] je složitost dokonce $O^*(1)$. Důkaz se nachází například v [3].

extractMax(Q) Strom, v jehož kořeni u se nachází maximum, se z haldy odtrhne. Následně je ze synů u, což jsou binomiální stromy, vyrobena korektní binomiální halda Q'. Poté jsou Q a Q' sloučeny.

Odtrhnutí stromu zabere $O(1)$ času, synů u je $O(\log n)$ a sloučení obou hald trvá $O(\log n)$. Celá operace má tedy logaritmickou časovou složitost [3].

increaseKey(Q, x, k) Zvětšování klíče bude probíhat podobně jako u výše představené binární haldy. Klíč v uzlu se přepíše a následně se opraví haldové uspořádání „probubláním“ klíče nahoru. Přinejhorším bude k novým maximem, což by jako důsledek lemmatu 3 znamenalo provedení $O(\log n)$ porovnání a prohození.

delete(Q, x) Smazání libovolného uzlu se dá provést jako volání dvojice operací `increaseKey(Q, x, ∞)` a `extractMax(Q)`. Obě tyto operace mají logaritmickou složitost a tím pádem se celé mazání provede v čase $O(\log n)$ [4].

1.2.4.3 Fibonacciho halda

Fibonacciho halda byla dle [13] zavedena jako vylepšení binomiální haldy. Principiálně z ní vychází, ale dovoluje porušit strukturu haldy i jednotlivých stromů a do korektního stavu je opravit až be chvíli, kdy to bude nutné. Tím se dá dosáhnout konstantní složitosti operace vkládání a slučování a amortizovaně konstantní složitosti `increaseKey(Q, x, k)` za cenu horší složitosti mazání (jak maxima, tak i libovolného prvku), které v nejhorším případě zabere $\Theta(n)$ času. Je rovněž implementačně složitější. V [3] je formálně definována takto:

Definice 12 (Fibonacciho halda)**:** ***Fibonacciho halda*** *se skládá ze souboru stromů $\mathcal{T} = T_1, \cdots, T_l$, kde:*

- *Uchovávané prvky jsou uloženy ve vrcholech stromů T_i.*
- *Pro každý strom $T_i \in \mathcal{T}$ platí haldové uspořádání.*

Pro každý vrchol v je definován ještě *řád vrcholu v* jako počet synů v, *řád stromu T* je pak roven řádu kořene T. Každý vrchol může být *označen*. Na

tento pojem bude odkazováno, ale jeho význam bude vysvětlen až při popisu operace `increaseKey(Q, x, k)`.

Na první pohled je vidět, že z definice oproti binomiální haldě vypadla podmínka o počtu stromů stejného řádu v haldě, může jich tedy být neomezeně mnoho. Samotné stromy jsou rovněž velmi volně definované. Později ale bude ukázáno, že úplně libovolnou strukturu mít nemohou.

merge(Q1, Q2) Pro popis dalších operací je nutné vědět, jak se dvě Fibonacciho haldy slučují. Jedná se o triviální operaci, kdy jsou spojové seznamy stromů obou hald spojené za sebe a je aktualizován ukazatel na maximum [14].

insert(Q, k) Z klíče k je vytvořena nová halda (neboli samotný strom řádu 0, jehož kořen není označený) a provede se sloučení s Q. Jak je uvedeno v [13], má operace `merge(Q1, Q2)` a tím pádem i `insert(Q, k)` časovou složitost $O(1)$.

extractMax(Q) Bude popsán postup podle [14]. Strom, v jehož kořeni se nachází maximum, se odtrhne ze seznamu stromů. Všem jeho synům se vynuluje ukazatele na otce, zruší se jejich případné označení a vytvoří se z nich halda. Ta se následně sloučí se zbytkem původní haldy.

Takto by ale Fibonacciho halda zdegradovala na úplně obyčejný neseřazený spojový seznam. Proto bude v dalším kroku prováděna tzv. *konsolidace* (nazvaná původně *linking step*), která bude slučovat stromy stejného řádu, aby se zmenšil počet stromů v haldě. Nejprve bude provedena analýza maximálního řádu stromu v haldě.

Nechť F_k je k-té Fibonacciho číslo. Dále počet vrcholů stromu T bude značen jako $|T|$. V [3] jsou uvedena a dokázána dvě lemmata.

Lemma 5: *Po každé provedené operaci platí, že je-li $T \in \mathcal{T}$ řádu k, potom $|T| \geq F_{k+2}$.*

Lemma 6: *Řád každého stromu ve Fibonacciho haldě je nejvýše $2\lceil \log_2 n \rceil$.*

První lemma bude ponecháno bez důkazu. Druhé pak plyne přímo z lemmatu 5 a z faktu, že pro $k \geq 0$ je $F_{k+2} \geq 1,6^k$ [3].

Při provádění konsolidace bude vytvořeno $2\lceil \log_2 n \rceil + 1$ spojových seznamů. Tento počet vychází z lemmatu 6 Postupně se z haldy odpojí všechny stromy a umístí se do seznamů dle jejich řádu. Strom řádu i se bude nacházet v i-tém seznamu. Poté algoritmus postupuje od nejmenšího po největší řád a stromy po dvou z odpovídajícího seznamu vybírá. Pokud byly v seznamu i alespoň dva stromy, sloučí je (stejným způsobem jako se slučovaly binomiální stromy), a výsledný strom vloží do seznamu $i + 1$. Pokud se v seznamu nacházel pouze jeden strom, přidá ho zpátky do haldy. Tento postup je opakován, dokud není seznam vyprázdněn. Poté se algoritmus přesune do dalšího řádu.

Na konec se projdou kořeny všech nových stromů a najde se nové maximum.

V [3] je provedena podrobná analýza, ze které vyplývá, že složitost je v nejhorším případě je $O(n)$, ale amortizovaná cena operace je pouze $O^*(\log n)$.

increaseKey(Q, x, k) Efektivita této operace je jednou z výhod Fibonacciho haldy oproti dříve představeným haldám. Uzel se změněným klíčem zde nebude „bublat“ směrem vzhůru.

Místo toho je dle [4] zrušeno případné označení x, podstrom zakořeněný v x je odtržen a přidán ho do haldy. Nyní se poprvé využije možnost označení uzlu. Označí se otec uzlu x (dále bude značen jako o). Pokud už ale o označený byl, označení se zruší a odtrhne se i podstrom zakořeněný v o a následně je připojen zpět do haldy. Takto se pokračuje i s otcem o, dokud se nenarazí na neoznačený uzel nebo na kořen celého stromu. Tento postup zabrání tomu, aby stromy příliš ztratily svoji strukturu.

Podobně jako u `extractMax(Q)`, je i zde složitost operace popsaná v [3]. Zvětšení klíče má amortizovanou časovou složitost $O^*(1)$.

delete(Q, x) Mazání libovolného prvku se podle [13] realizuje jako zvětšení klíče x na ∞ a následné smazání maxima. Složitosti těchto dvou operací již byla uvedena, takže není těžké si rozmyslet, že i složitost `delete(Q, x)` je $O^*(\log n)$.

1.2.4.4 Párovací halda

Fibonacciho halda je poměrně implementačně složitá a byť asymptotická analýza vychází dobře, v praxi není tak rychlá jako jiné typy hald [15]. Proto byla představena párovací halda, která je principiálně podstatně jednodušší. Její analýza je však velmi složitá a dodnes není plně dokončená, i když už se jí věnovala řada prací jako např. [16] nebo [17].

Na rozdíl od předchozích struktur je dle [15] párovací halda reprezentována jediným zakořeněným stromem, ve kterém platí haldové uspořádání, maximum je tedy v kořeni.

merge(Q1, Q2) Dva stromy (libovolně velké) budou slučovány podobným způsobem jako v případě binomiální nebo Fibonacciho haldy. Na základě porovnání klíčů v kořenech je jeden strom připojen jako nejlevější syn druhého. To vede na konstantní časovou složitost [15].

insert(Q, k) Vytvoří se nový strom skládající se pouze z kořene, který je poté sloučen se zbytkem haldy. I vkládání je tedy operace s konstantní časovou složitostí [15].

extractMax(Q) Mazání maxima je o něco složitější. Podle [17] je nejdříve odebrán kořen stromu. Tak zbude seznam jeho synů a jejich podstromů.

Synové se následně zleva doprava po dvojicích (odtud název párovací halda) slučují. Výsledné stromy se v dalším kroku slučují zprava doleva do jediného stromu, který bude reprezentovat haldu po odebrání maxima.

Časová složitost `extractMax(Q)` je stejně jako v případě Fibonacciho haldy $O^*(\log n)$ [15].

Poznámka: existují i odlišné strategie pro slučování synů, k dispozici jsou například v [17].

delete(Q, x) Mazání libovolného prvku (jiného než maxima) je popsáno například v [17]. Uzel x je odtržen ze stromu, z jeho synů se vyrobí korektní párovací halda stejným způsobem jako při mazání maxima. Ta je nakonec sloučena se zbytkem Q. Časová složitost je rovněž $O^*(\log n)$.

increaseKey(Q, x, k) Nejdřív se změní klíče na k. Pokud je x kořen, algoritmus končí. V opačném případě mohlo dojít k porušení haldového uspořádání. Celý podstrom s kořenem x je tedy odtržen a sloučen se zbytkem haldy.

Analýza této operace stále není hotová. V [16] byla dokázána dolní mez $\Omega^*(\log \log n)$. Naopak horní mez je podle [18] $O^*(2^{2\sqrt{\log \log n}})$, což je funkce striktně nižšího řádu než $\log n$. Ani jedna z mezí avšak není těsná.

1.2.4.5 Striktní Fibonacciho halda

Představení Fibonacciho haldy bylo průlomem, jelikož tento typ haldy dosáhl časové složitosti $O^*(1)$ pro všechny operace kromě mazání (a mazání maxima), kde je potřeba $O^*(\log n)$ času. Jejich nevýhodou je ale komplikovanost v porovnání s dříve představenými haldami. Dle [19] nejsou v praxi tak efektivní jako jiné teoreticky pomalejší struktury. Od té doby se mnoho vědců snažilo najít haldu, jejíž operace by dosahovaly stejné časové složitosti jako operace na Fibonacciho haldě, ale v nejhorším případě a nikoliv amortizovaně. Při tomto výzkumu vznikla celá řada často velmi složitých datových struktur, které se tomuto cíli přibližovaly.

Striktní Fibonacciho halda představená v [19] je první halda, která tohoto cíle dosáhla bez použití polí. Nejdříve bude princip jejího fungování popsán na abstraktní úrovni dle [19].

Celá striktní Fibonacciho halda je reprezentována jediným zakořeněným stromem, v jehož každém uzlu se nachází jediný klíč a v němž platí haldové uspořádání. *Velikostí stromu* se značí počet jeho uzlů. *Stupeň uzlu* je počet jeho synů. Každý uzel je *aktivní* nebo *pasivní*. Aktivní uzel s pasivním rodičem se nazývá *aktivní kořen*. *Hodnost* aktivního uzlu je definována jako počet jeho aktivních synů. Každému aktivním uzlu je přiřazeno nezáporné číslo *ztráta*. *Celková ztráta* haldy je součet ztrát všech aktivních uzlů. Pasivní uzel

je *připojitelný*, pokud jsou všichni jeho synové pasivní. Všechny uzly kromě kořene jsou udržovány v obyčejné frontě Q'. Uzel má *pozici* p, pokud se v Q' nachází na p. pozici. Halda počítá s unikátností klíčů, tudíž pro použití v prioritní frontě definované v této práci je nutné ke klíči přidat ještě identifikátor. Pro účely analýzy (ale nikoliv implementace) je potřeba ještě jedna konstanta $R = 2\log n + 6$, kde n je počet prvků v haldě [19]. Nyní budou představeny invarianty, které se v haldě udržují.

1. Aktivní synové uzlu se vždy nachází nalevo od pasivních synů. Kořen je pasivní a jeho připojitelní pasivní synové se nachází úplně vpravo. Pro každý aktivní uzel platí, že i-tý nejpravější aktivní syn má součet ztráty a stupně alespoň $i\ -1$. Aktivní kořen má nulovou ztrátu.

2. Počet aktivní uzlů je nejvýše $R + 1$.

3. Celková ztráta je nejvýše $R + 1$.

4. Maximální stupeň kořene je $R + 3$. Nechť x je nějaký uzel kromě kořene a p je jeho pozice v Q'. Pokud je x pasivní uzel nebo aktivní uzel s kladnou ztrátou, jeho stupeň je nejvýše $2\log(2n - p) + 9$; jinak je x aktivní uzel s nulovou ztrátou a může mít stupeň až $2\log(2n - p) + 10$.

Tyto invarianty zajistí platnost následující věty. I s jejím důkazem jsou uvedeny v [19].

Věta 11: *Všechny uzly ve striktní Fibonacciho haldě mají hodnost nejvýše R.*

Nyní bude představeno několik transformací, kterými lze opravit strukturu, pokud dojde k jejímu porušení provedením operací [19]. Unikátnost klíčů zajišťuje, že se ve stromu po provedení těchto transformací nikdy neobjeví cyklus.

- *Připojení.* Nejjednodušší transformace, která odpojí uzel x ze seznamu synů svého otce a s i jeho podstromem ho připojí pod uzel y. Pokud je x aktivní, bude připojen jako nejlevější syn y. Jinak se stane nejpravějším synem.

- *Redukce aktivních kořenů.* Nechť jsou x a y aktivní kořeny stejné hodnosti. Nejprve se porovnají jejich klíče. BÚNO řekněme, že klíč x je větší. Uzel y se připojí pod x a tím se zvětší hodnost x o jedna. Pokud je nejpravější syn z uzlu x pasivní, připojí se z pod kořen stromu. Tím se sníží počet aktivních kořenů o 1 a stupeň kořene se může zvětšit o 1.

- *Redukce stupně kořene.* Budiž x, y a z tři nejpravější připojitelní synové kořene. S užitím tří porovnání se seřadí klíče těchto uzlů. BÚNO předpokládejme, že $k(z) < k(y) < k(x)$. Uzly x a y se označí jako aktivní. Uzel z se připojí pod y a y se připojí pod x. Dále se x připojí

jako nejlevější syn kořene. Poté bude uzlům x a y nastavena ztráta 0 a hodnosti po řadě 1 a 0. Díky tomu se stupeň kořene sníží o 2 a počet aktivních kořenů se zvýší o 1.

- *Jednouzlová redukce ztráty* se aplikuje, když existuje aktivní uzel x se ztrátou ≥ 2. Nechť je y rodičem x. Uzel x je připojen pod kořen stromu a je označen jako aktivní s nulovou ztrátou, čímž se hodnost y sníží o 1. Pokud y není aktivní kořen, jeho ztráta se zvětší o 1. Jelikož se ztráta x snížila minimálně o 2, celková ztráta klesla alespoň o 1.

- *Dvouuzlová redukce ztráty* přijde na řadu ve chvíli, kdy dva aktivní uzly x a y se stejnou hodností mají ztrátu rovnou 1. BÚNO předpokládejme, že $k(x) > k(y)$. Rodiče y označme jako z. Nejdříve se y připojí pod x, čímž dojde ke zvýšení hodnosti x. Ztráta x a y se následně nastaví na nulu. Tímto se stupeň i hodnost z sníží o 1. Pokud není z aktivní kořen, jeho ztráta stoupne o 1.

merge(Q1, Q2) BÚNO předpokládejme, že velikost Q_1 je nejvýše velikost Q_2.Všechny uzly v Q_1 se označí jako pasivní, což lze provést za $O(1)$ času změnou jediné hodnoty [19]. Nechť je u větší z kořenů a v menší z nich. Uzel v se připojí pod u. Fronta Q' bude zřetězení Q'_1, v a Q'_2, kde Q'_i je obyčejná fronta. která je součástí haldy Q_i. Nakonec budou prováděny redukce aktivních kořenů a redukce stupně kořene, dokud bude některá z těchto transformací možná. Sloučit dvě striktní Fibonacciho haldy trvá $O(1)$ času [19].

insert(Q, k) Vytvoří se nová halda s jedním uzlem obsahujícím klíč k, která je poté sloučena s Q. Díky efektivitě slučování má i tato operace konstantní časovou složitost v nejhorším případě [19].

extractMax(Q) Prvek s největším klíčem se nachází v kořeni (dále značen jako r). Algoritmus nejdříve najde uzel x jako toho ze synů r, který obsahuje největší klíč. Pokud je x aktivní, udělá z něj pasivní uzel, čímž se ze všech jeho aktivních synů stanou aktivní uzly. Všechny ostatní syny r připojí pod x. Pasivní připojitelné syny x přesune v seznamu synů x doprava. Uzel x bude odstraněn z Q' a smazán. Poté se dvakrát provede následující: uzel y nacházející se na začátku Q' se přesune na konec této fronty; připojí se dva nejpravější synové y k x, pokud jsou pasivní. Potom bude provedena jedna redukce ztráty a nakonec bude prováděna v libovolném pořadí redukce aktivních kořenů a redukce stupně kořene, dokud bude jedna z těchto transformací možná. Tímto postupem je dosaženo časové složitosti $O(\log n)$ [19].

increaseKey(Q, x, k) Nechť je r kořen haldy. Nejdříve se uzlu x nastaví klíč k. Pokud je x kořen, operace je dokončená. Pokud je $k(x) > k(r)$, prvky v těchto uzlech se vymění. Rodiče x bude dále značen jako y. Uzel x bude

připojen pod kořen. Pokud byl x aktivní uzel ale nikoliv aktivní kořen, stane se z x aktivní kořen s nulovou ztrátou a hodnost y se sníží o 1. Pokud je to možné, provede se redukce ztráty. Poté se bude provádět 6 redukcí aktivního kořene a 4 redukce stupně kořene, dokud je to možné. Časová složitost operace `increaseKey(Q, x, k)` je $O(1)$ [19].

delete(Q, x) Jako u některých předchozích typů hald, i zde se zvýší klíč x na ∞ a smaže se maximum. Mazání libovolného prvku má tedy časovou složitost $O(\log n)$ [19].

1.2.5 Tabulka časových složitostí

Nyní budou přehledně shrnuty asymptotické časové složitosti operací v 13 datových strukturách, které byly představeny výše. V kapitole 3 pak bude vidět, jestli je dobré se při výběru implementace prioritní fronty řídit čistě asymptotickými složitostmi nebo nikoliv.

Operace `findMax` a `size` mají na všech strukturách shodně časovou složitost $O(1)$, jak již bylo řečeno na začátku kapitoly 1.2, proto nebudou uvedeny. Pokud je u operace `merge` uvedeno n, myslí se tím součet velikostí obou front.

	insert	extractMax	merge
Neseřazené pole	$O^*(1)/O(n)$	$O(n)$	$O(n)$
Seřazené pole	$O(n)$	$O(1)$	$O(n)$
Neseřaz. spoj. seznam	$O(1)$	$O(n)$	$O(1)$
Seřaz. spoj. seznam	$O(n)$	$O(1)$	$O(n)$
BVS	$O(n)$	$O(n)$	$O(n)$
AVL strom	$O(\log n)$	$O(\log n)$	$O(n)$
Červeno-černý strom	$O(\log n)$	$O(\log n)$	$O(n)$
(a-b)-strom	$O(\log n)$	$O(\log n)$	$O(n \log(n))$
Binární halda	$O(\log n)$	$O(\log n)$	$O(n)$
Binomiální halda	$O^*(1)/O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacciho halda	$O(1)$	$O^*(\log n)/O(n)$	$O(1)$
Párovací halda	$O(1)$	$O^*(\log n)/O(n)$	$O(1)$
Striktní Fib. halda	$O(1)$	$O(\log n)$	$O(1)$

Tabulka 1.1: Tabulka asymptotických časových složitostí operací na představených strukturách.

[a] $O(1)$ v případě, že se maže jakýkoli jiný prvek než maximum.

	increaseKey	delete
Neseřazené pole	$O(1)$	$O(n)^{\mathrm{a}}$
Seřazené pole	$O(n)$	$O(n)$
Neseřaz. spoj. seznam	$O(1)$	$O(n)^{\mathrm{a}}$
Seřaz. spoj. seznam	$O(n)$	$O(1)$
BVS	$O(n)$	$O(n)$
AVL strom	$O(\log n)$	$O(\log n)$
Červeno-černý strom	$O(\log n)$	$O(\log n)$
(a-b)-strom	$O(\log n)$	$O(\log n)$
Binární halda	$O(\log n)$	$O(\log n)$
Binomiální halda	$O(\log n)$	$O(\log n)$
Fibonacciho halda	$O^*(1)/O(n)$	$O^*(\log n)/O(n)$
Párovací halda	$O^*(2^{2\sqrt{\log \log n}})$	$O^*(\log n)/O(n)$
Striktní Fib. halda	$O(1)$	$O(\log n)$

Tabulka 1.2: Pokračování tabulky 1.1.

1.3 Využití

V předchozích částech práce byla představena prioritní fronta a podrobně rozebrána řada možných implementací. Nyní bude uvedeno několik příkladů použití, na kterých bude vidět význam této abstraktní datové struktury.

1.3.1 Heapsort

Prioritní fronta se dá velmi jednoduše použít k řazení. Algoritmus heapsort funguje tak, že všechny prvky vstupní posloupnosti se vloží do (maximové) prioritní fronty a následně se opakovaně volá mazání maxima [5]. Je možné použít libovolnou implementaci, ale jak už název algoritmu napovídá, často se užívá (binární) halda, přičemž algoritmus pak s haldou pracuje odlišným způsobem, než by se na první pohled nabízelo [5].

Vstupní posloupnost je uložena v poli. Na toto pole se zavolá algoritmus `heapBuild` popsaný v kapitole 2.8. Poté se bude volat `extractMin`. Díky tomu, že maximum se vždycky prohodí s posledním prvkem aktuálně uvažovaného pole, zbyde v poli na konci sestupně seřazená vstupní posloupnost [1].

Postavit z pole haldu trvá $O(n)$ času. N-krát vyjmout maximum z binární haldy zabere $O(n \log n)$ času. Celková časová složitost algoritmu heapsort je tedy $O(n \log n)$, což znamená, že se jedná o asymptoticky optimální řadící algoritmus [3]. V praxi se ukázalo, že algoritmus *quicksort* je rychlejší, byť jeho časová složitost v nejhorším případě je kvadratická [1].

1.3.2 Dijkstrův algoritmus

Dijkstrův algoritmus slouží k hledání nejkratších cest z jednoho vrcholu v_0 v orientovaném grafu, jehož hrany jsou ohodnoceny nezápornými čísly [1]. Délka hrana e bude značená jako $l(e)$.

Každý vrchol v může být buďto *nenalezený*, *otevřený* nebo *uzavřený* a je mu přiděleno *ohodnocení* ($h(v)$). Aby bylo možné nejkratší cestu zrekonstruovat, je nutné si pamatovat ještě předchůdce v. Na začátku jsou všechny vrcholy nenalezené a jejich ohodnocení je ∞, s výjimkou v_0, který je otevřený a jeho ohodnocení je 0 [3]. V prioritní frontě si algoritmus bude ukládat otevřené vrcholy.

Dokud existují nějaké otevřené vrcholy, vybere z nich vrchol v, který má nejmenší ohodnocení. Pro každého následníka w vrcholu v vykoná následující: pokud platí, že $h(w) > h(v) + l(v, w)$, nastaví $h(w)$ na $h(v) + l(v, w)$, otevře w a jeho předchůdce nastaví na v. Po projití všech následníků v uzavře [3].

V Dijkstrově algoritmu se uplatní minimová prioritní fronta. Otevření vrcholu znamená zavolání operace `insert`, aktualizace ohodnocení vrcholu je volání `decreaseKey` a uzavření vrcholu znamená provedení `extractMax`. Při použití neseřazeného pole dosáhne časové složitosti $O(n^2)$, což je výhodné pro husté grafy [1]. Při použití binární haldy ovšem postačuje $O((n + m) \log n)$ času a s Fibonacciho haldou dokonce pouze $O(m + n \log n)$ (n značí počet vrcholů a m počet hran grafu)[3].

1.3.3 Jarníkův algoritmus

Jarníkův (nebo také Primův) algoritmus dokáže najít minimální kostru ohodnoceného grafu. Jedná se o hladový algoritmus, který začíná se stromem o jednom (libovolném) vrcholu a žádné hraně. V každém kroku je poté do stromu přidána nejlehčí hrana vedoucí mezi některým vrcholem stromu a zbytkem grafu. Takto budou vybrány hrany tvořící nějakou minimální kostru [9].

Problém spočívá ve zvolení efektivního způsobu výběru nejlehčí hrany vedoucí do zbytku grafu. V prioritní frontě budou udržovány *sousední vrcholy* - vrcholy ležící mimo strom, které jsou s ním spojeny alespoň jednou hranou. Každému sousednímu vrcholu bude přiřazeno *ohodnocení* udávající váhu nejlehčí hrany, kterou je připojen ke stromu. V každém kroku je tedy vybrán soused u s nejnižším ohodnocením a je připojen ke stromu příslušnou nejlehčí hranou. Poté je nutné do prioritní fronty přidat sousedy vrcholu u, nebo upravit jejich ohodnocení, pokud v ní už byly [3].

Takto popsaný Jarníkův algoritmus připomíná Dijkstrův algoritmus, se kterým má i shodnou časovou složitost [3].

1.3.4 Huffmanovo kódování

Huffmanův kód je optimální prefixový kód používaný pro bezztrátovou kompresi. Symboly, které se ve vstupu objeví nejčastěji budou zakódovány pomocí

menšího počtu bitů. Naopak symboly, které se v něm vyskytují zřídka, budou mít kódové slovo delší. Kód lze zkonstruovat pomocí hladového algoritmu [2].

Algoritmus staví binární strom, jehož listy jsou symboly původní abecedy. Na začátku je z každého symbolu abecedy vytvořen strom o jednom uzlu. Tyto stromy jsou vloženy do minimové prioritní fronty, kdy priorita je dána četností výskytu symbolů ve stromu. Poté jsou v každé iteraci z fronty vybrány dva stromy s nejmenší četností, které jsou následně spojeny vytvořením nového kořene. Na konci v prioritní frontě zbyde pouze jeden finální strom. Kódové slovo symbolu je dáno cestou do daného listu z kořene. Dekódování potom probíhá procházením stromu [1].

Při použití binární haldy je časová složitost postavení stromu $O(n \log n)$ [1]. Komprese Huffmanovým kódem je velmi efektivní, úspora se většinou pohybuje mezi 20% a 90% [1], ovšem má i několik nevýhod. Pro zakódování jsou nutné dva průchody vstupem (jeden na postavení stromu a druhý na samotné zakódování), kódovací tabulka musí být explicitně uložena spolu se zakódovaným textem a Huffmanův kód sice dokáže využít četnosti jednotlivých znaků, ale existují i algoritmy, které rozpoznají i delší opakující se vzory [2].

1.3.5 Další příklady využití

Kromě zde uvedených příkladů má prioritní fronta i řadu dalších použití. V diskrétních simulacích je potřeba udržovat si množinu čekajících událostí, ze kterých se dá vybrat ta nejdéle čekající a vložit nově vytvořené [9]. Stejně tak operační systém musí přidělovat omezené prostředky různým procesům, které mohou mít různou prioritu [1]. Za zmínku stojí ještě algoritmus A*, který stejně jako Dijkstrův algoritmus hledá nejkratší cesty v grafu, ale využívá heuristiky [20].

1.4 Možnosti a přínos paralelizace prioritní fronty

Pojem paralelizace prioritní fronty může být chápán dvěma způsoby. Může jít o použití několika procesorů ke zrychlení operací, které manipulují s jedním prvkem prioritní fronty. Tomuto tématu se věnovalo v minulosti několik prací, příkladem je práce [21]. V ní byla popsána prioritní fronta pro model CREW PRAM (definován v [22]), podporující nalezení maxima v konstantním čase na jednom procesoru a ostatní operace (vytvoření, vkládání, slučování, nalezení a mazání maxima, mazání libovolného prvku a změnu klíče) v konstantním čase při použití $O(\log n)$ procesorů.

Nebo se může jednat o paralelizaci ve smyslu současného přístupu (vkládání, mazání, ...) ke k prvkům, kde k je konstanta [23]. Dále bude podrobně rozebrána jedna strategie, jak tyto operace efektivně realizovat a bude odkázáno na několik dalších.

1.4.1 n-Bandwidth-Heap a n-Bandwidth-Leftist-Heap

Nejprve je třeba definovat *paralelní prioritní frontu* podle [24].

Definice 13 (Paralelní prioritní fronta)**:** ***(Maximová) paralelní prioritní fronta*** *Q je abstraktní datová struktura skládající se z prvků označených přirozenými čísly, která podporuje následující operace:*

- `Insert(<i1,...,in,>, Q)` *vloží prvky $i_1, ..., i_n$ do Q.*
- `Deletemin(Q, n)` *smaže a vrátí n největších prvků z Q.*
- `Makequeue(S, Q)` *zkonstruuje Q z množiny prvků S.*

A volitelně:

- `Meld(Q1, Q2, Q)` *sloučí Q_1 a Q_2 do Q.*

V [24] byly představeny hned dvě datové struktury *n-Bandwidth-Heap* (krátce n-H) a *n-Bandwidth-Leftist-Heap* (n-L) vycházející z binární respektive levicové haldy (leftist heap)[b]. Změna spočívá v přítomnosti n klíčů v uzlu. Uzly mezi sebou udržují takzvané *rozšířené haldové uspořádání*, tzn. že největší klíč uložený v uzlu je větší nebo roven všem klíčům uloženým v potomcích tohoto uzlu (v maximové verzi). Nechť $m = f(n) \cdot n$, kde $f(n)$ je rostoucí funkce z n do $\mathbb{N}$, je počet klíčů v haldě. Hloubka n-H nebo délka nejpravější cesty v n-L bude značena jako $h \in O(\log \frac{m}{n})$.

Dále jsou v [24] popsány i algoritmy umožňující efektivní realizaci operací paralelní prioritní fronty. N-Bandwidth-Leftist-Heap narozdíl od n-Bandwidth-Heap podporuje slučování. Pro obě haldy jsou důležité dvě základní operace - `PARALLEL-SORT(`S`)` a `PARALLEL-MERGE(`E_1`,`E_2`)`, kde S je množina prvků k seřazení, E_1 a E_2 jsou seřazené posloupnosti, které budou slity. S pomocí n procesorů v modelu CREW PRAM může být n klíčů seřazeno v čase $\Theta(\log n)$ a slévání dvou vektorů s kardinalitou k_1 respektive k_2 lze zvládnout v čase $\Theta(\frac{k_1+k_2}{n} + \log\log n)$ [24].

Operace s n-H Pro každou cestu z kořene haldy do listu platí, že konkatenace množin prvků z jednotlivých uzlů v této cestě tvoří seřazenou množinu. Při vkládání nebo mazání ale dojde ke změně prvků v kořeni nebo listu, což poruší rozšířené haldové uspořádání. Aby se toto uspořádání obnovilo, je definována subrutina `REARRANGE`, která jako parametry přijímá cestu v haldě a uzel (kořen nebo list na této cestě). tato subrutina zavolá `PARALLEL-MERGE` na všechny klíče v uzlech na cestě. Výsledná posloupnost je rozdělena na celky po n prvcích, které jsou vloženy zpět do uzlů. Časová složitost této subrutiny je $O(|\pi| + \log\log n)$, kde $|\pi|$ je délka cesty [24].

[b] Nebyla popsána v této práci, popis se nachází například v [25].

Při vkládání n prvků do n-H jsou nejdříve prvky vloženy do nejlevějšího prázdného listu (jako u klasické binární haldy). Následně jsou prvky v tomto uzlu seřazeny pomocí `PARALLEL-SORT`. Na konci je cesta z kořene do tohoto listu opravena pomocí `REARRANGE`. Časová složitost celého vkládání je dána zejména seřazením vkládaných klíčů a následnou opravou cesty, celkem tedy $O(h + \log n)$ [24].

Pro popis dalších operací je potřeba si definovat *minimální cestu*. Ta je definovaná rekurzivně. Kořen haldy patří do minimální cesty. Pokud uzel patří do minimální cesty a X a Y jsou jeho synové a Y je buďto prázdný nebo je největší klíč v X menší nebo roven nejmenšímu klíči v Y, pak X patří do minimální cesty [24]. Dále je potřeba další subrutina `ADJUST`. Ta pro každý uzel P na minimální cestě zkontroluje, zda existuje jeho sourozenec P'. Pokud ano, provede se paralelní slévání klíčů P a P' a výsledná posloupnost se rozdělí na poloviny a vloží zpět do P a P'. To se dá zvládnout za $O(h + \log\log n)$ času [24].

Mazání rovněž začíná stejně jako v klasické binární haldě. Do kořene se zkopírují prvky z nejpravějšího obsazeného listu z poslední hladiny. Následně se provede volání `ADJUST` na minimální cestu a `REARRANGE` rovněž na minimální cestu s tím, že druhým argumentem je kořen. Nakonec jsou vráceny prvky původního kořene. Časová složitost mazání n prvků je $O(h + \log\log n)$ [24].

Poslední popisovanou operací je `Makequeue(S, Q)`. Nejdříve je postavena nová halda, jejíž uzly obsahují seřazené klíče (neboli $\frac{m}{n}$ volání `PARALLEL-SORT`). V tuto chvíli ještě nemusí platit rozšířené haldové uspořádání. To se opraví tak, že se celá halda projde od nejnižší úrovně až po kořen a v každé úrovni se pro každý uzel P najde minimální cesta z P do kořene a zavolá se na ni `ADJUST` a `REARRANGE`. Postavení haldy zabere $O(\frac{m}{n} \log n)$ času [24].

Operace s n-L Struktura n-H neumožňuje stejně jako klasická binární halda efektivní slučování, naproti tomu operace s n-L jsou postavené na slučování dvou těchto struktur, je tedy nutné si nejdříve popsat tuto operaci. Nejprve jsou sloučeny cesty cesty z kořenů do nejpravějších uzlů obou hald. Ostatní uzly jsou k výsledné cestě vhodně připojeny. Nakonec je provedeno přepočítání hodností, aby se obnovila správná struktura levicové haldy. To je velmi vágní popis, ale vzhledem k tomu, že v této práci nebyla levicová halda popsána, na podrobný popis bude pouze odkázáno do [24]. Sloučení dvou n-L zabere $O(h + \log\log n)$ času [24].

Při vkládání n prvků do n-L Q je vytvořena nová n-L s jedním uzlem, která je sloučena s Q. Mazání n největší prvků probíhá tak, že se smaže kořen a jeho levý a pravý podstrom (dva korektní n-L) jsou sloučeny. Při konstruování n-L se vytvoří $\frac{m}{n}$ hald o jednom uzlu, které jsou poté postupně sloučeny do jedné. Časové složitosti těchto operací jsou stejné jako v případě n-H [24].

1.4.2 Další přístupy

V [26] byla představena paralelní prioritní fronta pracující na modelu EREW PRAM (definice dostupná v [22]). Jedná se opět o dvě struktury, z nichž jedna podporuje efektivní slučování. Tyto struktury jsou podobné haldám popsaným v 1.4.1, ale zvládnou provést vložení k prvků nebo smazání k největších prvků v čase $O(\log\log\frac{n}{k} + \log k)$ [23].

Dále byl v [27] představen způsob, jak efektivně namapovat prioritní frontu reprezentovanou tzv. *slope* haldou na hyperkrychli a optimální paralelní algoritmy pro vkládání a mazání maxima. Práce popisuje dvě implementace těchto operací. První z nich dosahuje na prioritní frontě o n prvcích, kde v každém uzlu je uloženo b prvků, při mazání b nejmenších prvků nebo vkládání b prvků zrychlení $O(min(\log n, \frac{b\log n}{\log b+\log\log n}))$, kde $b = O(n^{1/c})$ pro $c > 1$. Zejména operace jednoho vkládání nebo jednoho mazání jsou optimální a při použití $p = O(\frac{\log n}{\log\log n})$ procesorů vyžadují pouze $O(\frac{\log n}{p} + \log p)$ času. Druhá implementace využívá větší množství procesorů a na jedné hyperkrychli dosahuje téměř optimálního zrychlení. Vkládání $\log n$ seřazených prvků nebo mazání $\log n$ největších prvků zvládne za $O(\log\log n^2)$ času na $O(\frac{\log^2 n}{\log\log n})$ procesorech. Na silnějším modelu zřetězené hyperkrychle pak vykoná druhá implementace $\log n$ operací v čase $O(\log\log n)$ použitím $O(\frac{\log^2 n}{\log\log n})$ procesorů, což znamená, že dosáhne optimálního zrychlení [27].

Za zmínku stojí ještě paralelní prioritní fronta představená v [28], která umožňuje provedení paralelního vkládání seřazené posloupnosti prvků, paralelní zvětšení klíče seřazené posloupnosti prvků, smazání maxima a smazání libovolného prvku v konstantním čase. Tato prioritní fronta může být implementována na EREW PRAM a umožňuje provést jakoukoli posloupnost n operací v čase $O(n)$ a při $O(n\log n)$ celkové práce, kde m je počet vkládaných nebo upravovaných prvků. Tvůrci zamýšlené použití této struktury je především paralelní implementace Dijkstrova algoritmu, který by na CREW PRAM běžel v čase $O(n)$ při $O(m\log n)$ celkové práce (n značí počet vrcholů a m počet hran grafu) [28].

Výše uvedené přístupy rozhodně nejsou jediné strategie pro implementaci paralelní prioritní fronty, v různých pracích bylo publikováno ještě několik dalších strategií.

Kapitola 2

Implementace

V rámci praktické části práce byla implementována prioritní fronta s pomocí datových struktur popsaných v kapitole 1.2. Byl použit jazyk C++, který je pro tento účel vhodný zejména díky své efektivitě. Ze třinácti představených datových struktur jich bylo implementováno 12, striktní Fibonacciho halda se mezi nimi nenachází vzhledem k její značné komplikovanosti a praktické nedostupnosti vzorové implementace v žádném jazyce. Popis možné implementace je ale uveden dále v podkapitole 2.12.

Z operací definovaných v kapitole 1.1.1 se v kódu nachází tyto:

- `insert(Q, k)`
- `extractMax(Q)`
- `size(Q)`
- `findMax(Q)`
- `merge(Q1, Q2)`

To znamená, že `increaseKey(Q, x, k)` a `delete(Q, x)` nebyly implementovány. V některých případech by to s tímto rozhraním ani nebylo možné, konkrétně pro (a-b)-stromy (jak bylo vysvětleno v kapitole 1.2.3.4) a pro struktury realizované pomocí pole (seřazené pole, neseřazené pole, binární halda), jejichž prvky se v paměti přesouvají.

Implementace datových struktur vychází z jejich abstraktnějších popisů v kapitole 1.2 s několika zjednodušeními, které jsou způsobeny právě nepřítomností dvou výše uvedených operací. To znamená, že se mohly v některých případech mírně zjednodušit i ostatní operace. Žádná z těchto drobných změn ale nemá vliv na asymptotické složitosti.

Praktická část práce často využívá standardní knihovnu jazyka C++, např. STL kontejnery, knihovnu `<algorithm>` nebo další.

2.1 Struktura projektu

Základem je třída `PriorityQueue`, která v konstruktoru přijímá ukazatel na abstraktní třídu `DataContainer`. Slouží tedy jako obal pro nějakou konkrétní implementaci jako je např. `SortedArrayContainer` nebo `AVLTreeContainer`, což jsou podtřídy `DataContainer`. Všechny tyto třídy jsou šablonové a mají typový parametr `ValueType`. Je možné vytvořit prioritní fronty z prvků datového typu, který má definované operátory menší, menší nebo rovno a rovno.

V projektu se ale nachází ještě dvě další podtřídy `DataContainer`. Jedná se o `StdPQContainer` (volá metody `std::priority_queue` a slučování je vyřešeno jako opakované vkládání prvků) a `BoostFibHeapContainer` (obaluje `boost/heap/fibonacci_heap`). Tyto třídy budou pro srovnání rovněž zahrnuty do měření popsaného v kapitole 3, aby nebyly pouze porovnávány vlastní implementace mezi sebou.

Dále jsou v projektu přítomny ještě třídy `EmptyQueueException` (výjimka vyhazovaná při pokusu o získání prvku z prázdné fronty), `Tester` (obsahuje testy pro datové struktury) a `PerformanceMeter`, která měří výkonnost jednotlivých datových struktur dle metrik popsaných v kapitole 3.

2.2 Pole

Třídy reprezentující seřazené i neseřazené pole jsou v podstatě obalové třídy nad `std::vector`. V neseřazené variantě je potřeba ještě index největšího prvku.

2.3 Spojové seznamy

I zde bylo využito kontejnerů z knihovny STL. V případě neseřazené varianty spojového seznamu bylo využito obousměrně zřetězeného seznamu `std::list` s tím, že je udržován `const_iterator` na uzel s maximální hodnotou.

Pro seřazenou variantu postačuje použití `std::forward_list`, neboli jednosměrně zřetězeného seznamu, jelikož nikdy nebude mazán jiný než první uzel. Zde není žádný iterátor potřeba.

2.4 Binární vyhledávací strom

Všechny struktury popsané v podkapitole 1.2.3 jsou implementované jako spojové struktury, přičemž u BVS obsahuje každý uzel kromě klíče i ukazatel na levého a pravého syna a na rodiče. V případě, že syn nebo rodič neexistují, ukazatel má hodnotu `nullptr`.

Pro celý strom je potřeba znát adresu kořene a nejpravějšího uzlu (s maximálním klíčem). Veškerá manipulace se stromem poté znamená aktualizaci

ukazatelů v uzlech, většina operací je realizovaná rekurzivně. Při implementaci BVS (i jeho vyvažovaných variant) nebylo potřeba brát v úvahu situaci, kdy se maže uzel se dvěma syny, tedy ten nejkomplikovanější případ.

2.5 AVL strom

Reprezentace AVL stromu je podobná jako v případě klasického BVS. V uzlu pouze přibyla informace o hloubce. Tu tedy není potřeba počítat pokaždé znovu, je ale nutné ji vždy udržovat aktuální.

Pochopitelně bylo nutné implementovat logiku vyvažování stromu a levou a pravou rotaci (LR a RL rotace jsou realizovány dvěma voláními jednoduchých rotací).

2.6 Červeno-černý strom

V každém uzlu se kromě standardních tří ukazatelů udržuje ještě informace o barvě. Na rozdíl od předchozích struktur není chybějící syn nebo rodič reprezentován nulovým ukazatelem. Místo toto byl zaveden speciální uzel NIL, který neobsahuje klíč.

Není nutné vytvářet více instancí NIL uzlu, postačuje jen jedna instance, na kterou bude ukazovat více vnitřních uzlů. Ukazatele na rodiče tohoto jediného NIL uzlu je možné dle potřeby upravovat. Konkrétně ve třídě `RedBlackTreeContainer` je NIL reprezentován statickou členskou proměnnou (stejného datového typu jako ostatní uzly). Klíčové slovo `static` bylo použito z důvodu přístupu k NIL z vnořené struktury. Díky tomu není tato třída thread-safe, měření prováděné v rámci této práce je ovšem realizované čistě sekvenčně, takže nedojde ke zkreslení výsledků.

U popisu operace slučování v kapitole 1.2.3.3 bylo řečeno, že uzly v poslední hladině budou obarveny červeně, ale z tohoto vysvětlení není úplně jasné, jak toho efektivně docílit. Obarvování lze zajistit už při konstrukci stromu. Z počtu uzlů se spočítá maximální hloubku. Při samotné konstrukci se poté jako parametr rekurzivní funkce předává i hloubka zanoření. Podle tohoto údaje algoritmus ví, do jaké hladiny vkládá a může tedy zvolit správnou barvu.

2.7 (a-b)-strom

Struktura (a-b)-stromu se od ostatních stromů značně odlišuje. Příslušná třída `ABTreeContainer` sice byla implementována pro obecné parametry, ale pro jejich změnu je potřeba třídu znovu zkompilovat. Tématem práce není hledání nejvhodnějších parametrů (a-b)-stromu, všechna měření budou provedena na často používaném (2-3)-stromu.

Vnější uzly jsou reprezentovány nulovými ukazateli. V uzlu je uložen seznam klíčů a seznam ukazatelů na syny (pro oboje byl využit `std::vector`).

Ukazatel na rodiče v uzlu není přítomen. Jak již bylo ukázáno, při opravě struktury po vkládání nebo mazání mohou vznikat nebo zanikat uzly a udržování ukazatele na rodiče by bylo složitější než v dříve představených stromech. Na rozdíl od ostatních spojových struktur není zvlášť udržován ukazatel na uzel s největším klíčem, ale přímo hodnota tohoto klíče, jelikož ukazatel na nejpravější uzel ve stromu může být některými operacemi zneplatněn. Právě z důvodu absence ukazatele na rodiče se případné dělení uzlů provádí při návratu z rekurze.

Použitá implementace se z velké části drží pseudokódů z [3]. To znamená, že při vkládání je volána pomocná metoda, která vrací buďto Ø (v kódu `std::nullopt`) nebo trojici obsahující klíč a dva uzly (`std::tuple`).

2.8 Binární halda

Binární halda je reprezentována polem (`std::vector`), nikoliv spojovou strukturou. Používá se indexování od 0. Z kapitoly 1.2.4.1 stojí za připomenutí, že v tomto případě se index rodiče n-tého prvku spočítá jako $\lfloor (n-1) \,/\, 2 \rfloor$, index levého syna je $2 \cdot n + 1$ a pravý syn se nachází na pozici $2 \cdot n + 2$. Slučování se realizuje dříve popsaným algoritmem `heapBuild`.

2.9 Binomiální halda

Uzel binomiálního stromu je v paměti reprezentován jako struktura, která kromě klíče obsahuje ještě ukazatel na rodiče, na nejpravějšího syna a na levého sourozence.

Pro celou haldu si stačí pamatovat ukazatel na kořen nejmenšího stromu. Všechny stromy pak budou tvořit spojový seznam. K tomu lze využít ukazatel na levého sourozence kořenů. Tím vznikne jednosměrně zřetězený seznam, takže je dobré si pamatovat i ukazatel na předchůdce stromu s maximálním klíčem, ať ho není nutné hledat.

Při mazání uzlu se vyrábí binomiální halda ze synů. Synové už se ve spojovém seznamu nachází, avšak jsou ve špatném pořadí. Je proto nutné otočit směr seznamu. K tomu postačuje jeden průchod seznamem.

2.10 Fibonacciho halda

V řadě zdrojů včetně původního článku představujícího Fibonacciho haldu [14] se uvádí implementace souboru stromů a souboru synů každého vrcholu pomocí kruhového obousměrně zřetězeného spojového seznamu. V implementaci vytvořené v rámci této práce je ale použit po vzoru [3] klasický obousměrně zřetězený seznam (`std::list`).

Pro celou haldu je udržován ukazatel na T_1, T_l (neboli první a poslední strom haldy) a na uzel s maximálním klíčem, což je rovněž kořen nějakého

stromu. V každém uzlu je spolu s klíčem uložen řád uzlu, ukazatele na rodiče, nejlevějšího syna a levého a pravého sourozence. V této implementaci úplně odpadlo označování vrcholů, protože podstromy jiného uzlu než maxima se nikdy nebudou odtrhávat. Tím se i mírně zjednodušila operace mazání maxima, kdy není odpadá krok s rušením označení synů.

2.11 Párovací halda

V každém uzlu párovací haldy se nacházejí ukazatele na nejlevějšího syna a na pravého sourozence. To znemožňuje efektivně implementovat operace `increaseKey(Q, x, k)` a `delete(Q, x)` (pro upravení ukazatele na syna v otci x by musel být prohledán celý strom), což ale v tomto případě nevadí. Pokud by měly být používány i tyto dvě operace, by byl přidán ještě ukazatel na levého sourozence, který by v případě uzlů, které levého sourozence nemají, sloužil jako ukazatel na otce [9]. V [9] je popsán ještě mírně komplikovanější způsob, jak reprezentovat strom pomocí dvou ukazatelů v uzlu při stejných časových složitostech těchto operací.

Za zmínku stojí konkrétní podoba implementace operace mazání maxima. Algoritmus potřebuje dvě pole. Synové kořene se odpojí ze seznamu sourozenců a jsou vloženy do prvního pole P_1. Pokud byl synů lichý počet, vloží se na konec P_1 ještě prázdný strom. Nyní přijde na řadu slučování po stromů po dvojicích. Výsledné stromy se budou vkládat do druhého pole (P_2). Stromy v P_2 se nakonec sloučí během jediného průchodu polem. To znamená, že pro k od l do 1 se bude slučovat $P_2[k]$ s $P_2[k-1]$ a výsledek se bude ukládat do $P_2[k-1]$. Na konci se tak bude v $P_2[0]$ nacházet výsledný strom.

2.12 Striktní Fibonacciho halda

Níže bude uvedena jedna z možných implementací zmíněná v [19].

Každý *uzel* bude kromě klíče obsahovat ještě ukazatele na levého a pravého sourozence, rodiče a nejlevějšího syna a ukazatel na tzv. *aktivní záznam*. Informace o tom, zda-li je uzel aktivní nebo pasivní tedy není uložena přímo v uzlu.

Aktivní záznam obsahuje booleovskou proměnnou, která určuje jestli jsou uzly ukazující na tento záznam aktivní nebo pasivní. Dále se zde nachází i čítač referencí, který je užitečný pro uvolnění paměti v případě, že záznam už není potřeba.

Dále je potřeba *seznam hodností*, který bude v sestupném pořadí obsahovat jeden uzel pro každou možnou hodnost r. V každém z těchto uzlů se nachází ukazatel na oba sourozence, 2 ukazatele na záznam v *seznamu oprav* (bude představen dále) - jeden na aktivní uzel hodnosti r s kladnou ztrátou a jeden na aktivní kořen hodnosti r. Poslední položkou bude čítač referencí.

Další položkou, která bude ukládána, je již zmíněný *seznamu oprav*, v jehož záznamu se nachází ukazatel na uzel stromu, ukazatel na levého a pravého sourozence (jedná se o kruhový spojový seznam) a ukazatel na příslušný záznam v *seznamu hodností* (dle hodnosti uzlu). Účel *seznamu oprav* je ukládání aktivních uzlů, nad kterými by se dala provést některá z transformací zmíněných výše.

K práci s haldou je nutné znát její velikost, ukazatel na kořen, ukazatel na *aktivní záznam*, ukazatel na nejlevějšího pasivního nepřipojitelného syna kořene, ukazatel na uzel, který se nachází a prvním místě v Q', ukazatele na nejpravější záznamy v *seznamu hodností* a *seznamu oprav* a ukazatel na nejlevější záznam v *seznamu hodností* s kladnou ztrátou.

Kapitola 3

Měření

Dále byla změřena praktická výkonnost implementovaných variant prioritní fronty v několika scénářích.

3.1 Technika měření

V prvním testu bylo do prioritní fronty vloženo n prvků *vzestupně seřazené posloupnosti*. Každý nově vložený prvek měl tedy maximální klíč. Poté byla n-krát volaná operace mazání maxima.

Druhý test je podobný, ale tentokrát byly vloženy prvky *sestupně seřazené posloupnosti*, žádný vložený prvek tedy nebyl novým maximem (kromě prvku vloženého na začátku). Nakonec byly všechny prvky z fronty opět vyjmuty.

Následně byl proveden test *náhodného vkládání a výběru*, kdy byl n-krát s 50% pravděpodobností do prioritní fronty vložen prvek s náhodným klíčem. Jinak bylo smazáno maximum. Pokud by měli dojít k mazání maxima, ale fronta byla prázdná, rovněž se vložil náhodný prvek. Na konci byly vyjmuty všechny zbývající prvky. V tomto scénáři se s velkou pravděpodobností nikdy nebude ve frontě vyskytovat velké množství prvků.

V dalším scénáři - *převažující vkládání* - tomu bude jinak. Poměr operací vkládání a mazání nyní bude 4:1. Na konec se opět smažou zbývající prvky.

Na konec bude změřena výkonnost operace *slučování*. Zde se naplní n prioritních front 100 prvky (čas potřebný k naplnění není zahrnut do měření) a následně se všechny sloučí do jedné.

Pro každou hodnotu n proběhly všechny testy třikrát a jako výsledný čas je brán medián jednotlivých časů. Ve všech testech byl jako prvek prioritní fronty použit datový typ `uint32_t`.

3.2 Testovací server a nastavení překladače

Měření bylo provedeno na školním serveru STAR, aby se snížila míra ovlivnění měření jinými běžícími procesy. Server měl následující konfigurací:

- CPU: Intel® Xeon® CPU E5-2620 v2 @ 2,1 GHz (6 jader, 12 vláken)
- RAM: 32 GB
- OS: CentOS 7

Použitým překladačem je `g++` ve verzi 8.2.1 s přepínači `-Wall -pedantic -std=c++17 -O3`.

V průběhu tvorby implementace se ukázalo, že přepínač `-O3` má značný vliv na výsledky měření. U některých struktur nebyl rozdíl mezi stupni optimalizace 0 a 3 znatelný, někde ale šlo o desetinásobné zrychlení. Zkoumání vlivu optimalizace ale není cílem této práce.

3.3 Diskuze výsledků

Nyní budou rozebrány výsledky měření dle výše popsaných scénářů. Grafy kompletních výsledků jsou k dispozici v příloze A. Vzhledem k počtu měřených datových struktur ale nejsou grafy na první pohled přehledné a jsou v práci uvedené hlavně pro úplnost.

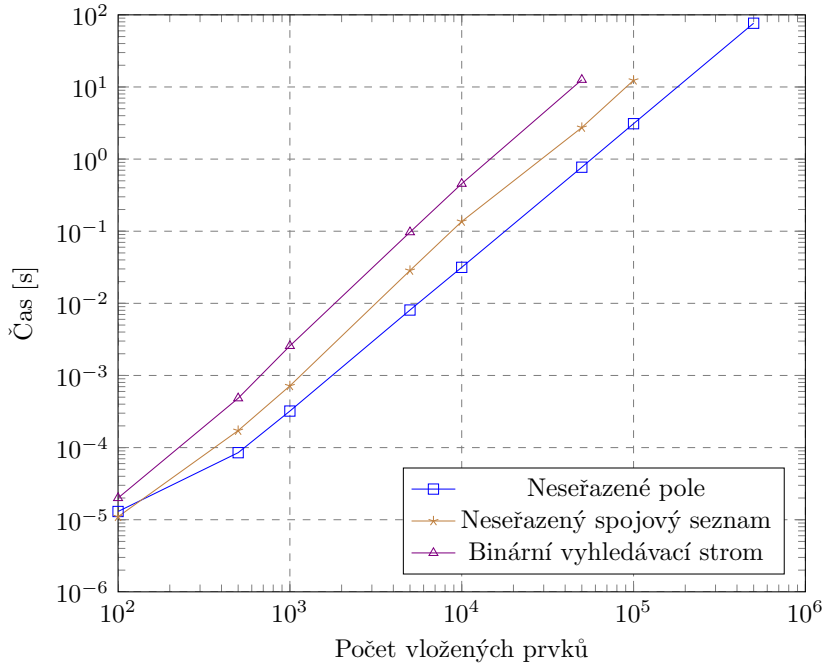
3.3.1 Vzestupně seřazená posloupnost

V tomto scénáři se jasně vyprofilovaly dvě skupiny implementací.

Zdaleka nejhůře dopadly prioritní fronty realizované jako *neseřazené pole*, *neseřazený spojový seznam* a *binární vyhledávací strom*. U prvních dvou struktur bylo sice vkládání velmi rychlé, avšak při každém vyjmutí maxima bylo nutné projít všechny prvky a najít nové maximum, což se ukázalo jako velmi časově náročné. Binární vyhledávací strom v tomto případě zdegeneroval na spojový seznam, který musel být celý prohledán při vkládání. Výsledky nejpomalejších struktur se nachází v grafu 3.1.

Naopak nejlépe si vedly *seřazené* varianty pole a spojového seznamu, což je vidět v grafu 3.2. Při vkládání vzestupně seřazené posloupnosti nebylo třeba přesouvat žádné dříve vložené prvky. Tyto datové struktury se tedy hodí pro implementaci prioritní fronty ve specifickém případě, kdy je vkládána (téměř) seřazená posloupnost prvků.

Výsledné časy ostatních implementací se od sebe liší už méně. Za zmínku stojí, že dobrého výsledku dosáhla například většina hald a *std::priority_queue*, naopak horší čas byl naměřen u *vyvažovaných variant BVS* a obou implementací *Fibonacciho haldy*.



Obrázek 3.1: Nejpomalejší implementace v testu vzestupně seřazené posloupnosti.

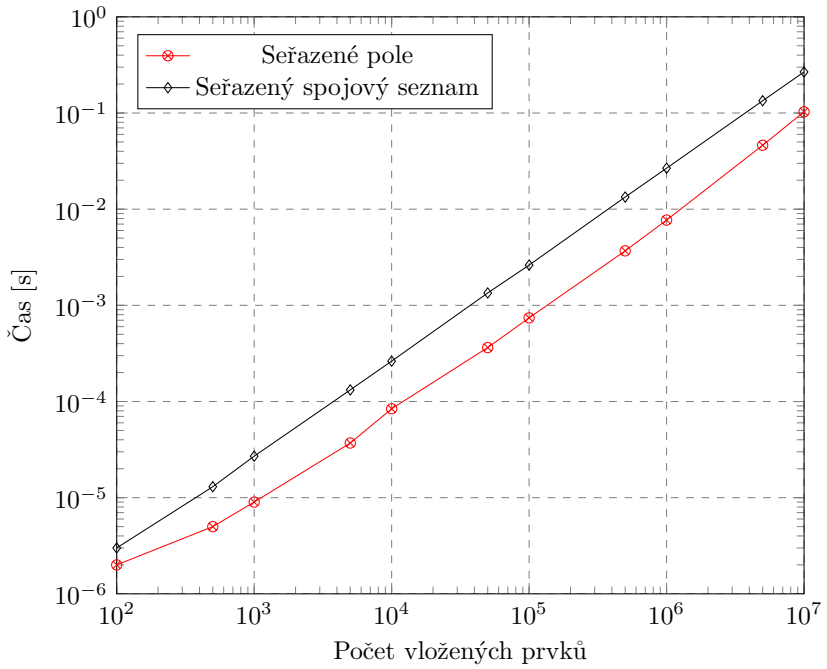
3.3.2 Sestupně seřazená posloupnost

Jak je vidět na obrázku 3.3, opět se jako pomalé ukázaly implementace pomocí *neseřazeného pole*, *neseřazeného spojového seznamu* a *binárního vyhledávacího stromu*. Podobně špatně si ale tentokrát vedly i *seřazené pole* a *neseřazený spojový seznam*, dokonce jsou pomalejší než jejich neseřazené verze. Na rozdíl od scénáře s vkládáním vzestupně seřazené posloupnosti bylo při vkládání nutné nový prvek přesunout na začátek.

Z tohoto testu nevyšel jasný vítěz. Se srovnatelným časem skončily *haldy* (kromě Fibonacciho) a *std::priority_queue*. Vyvažované stromy a Fibonacciho halda (výše popsaná implementace i ta z knihovny `boost`) byly opět o něco pomalejší. Vybrané výsledky jsou znázorněné v grafu 3.4.

3.3.3 Náhodné vkládání a výběr

V tomto scénáři se s velkou pravděpodobností v prioritní frontě nenacházelo mnoho prvků. Rozdíl v naměřených časech je podle očekávání mezi jednotlivými implementacemi o poznání menší. Jako nejpomalejší se ukázaly prioritní fronty postavené na *neseřazeném poli* a zejména *neseřazeném spojovém seznamu*. Jejich *seřazené* varianty ale dopadly naopak nejlépe ze všech implementací. V grafu 3.5 jsou znázorněné obě varianty. Z ostatních implemen-



Obrázek 3.2: Nejrychlejší implementace v testu vzestupně seřazené posloupnosti.

tací stojí za zmínění (2-3)-strom a implementace Fibonacciho haldy vytvořená v rámci této práce. Jejich časy se blíží neseřazenému poli. Zbytek datových struktur dopadl srovnatelně.

3.3.4 Převažující vkládání

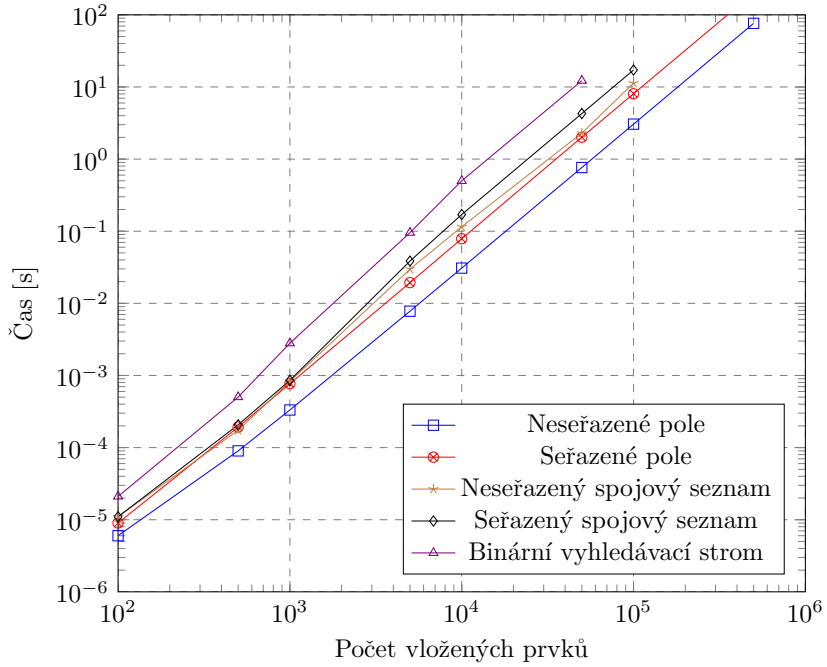
Zde už jsou díky přítomnosti většího množství prvků vidět větší rozdíly. V tomto testu propadly zejména *spojové seznamy* a za nimi *pole*(nebudeme ukazovat na grafu, jedná se o podobnou situaci jako v předchozích testech). Vůbec nejlépe si vedly *binární halda* a *std::priority_queue*.

V grafu 3.6 lze s odstupem pozorovat ještě *červeno-černý strom*. V těsném závěsu za ním skončila většina ostatních struktur. Pouze *(2-3)-strom* a verze *Fibonacciho haldy* implementovaná v rámci této práce jsou znatelně pomalejší, což je rovněž trend pozorovaný i v předchozích scénářích.

3.3.5 Slučování

Časy naměřené v rámci testu slučování se pro jednotlivé implementace značně liší.

Téměř identických časů dosáhly *BVS*, *AVL strom* a *červeno-černý strom*. Za nimi skončily *neseřazené* varianty pole a spojového seznamu a *binární*



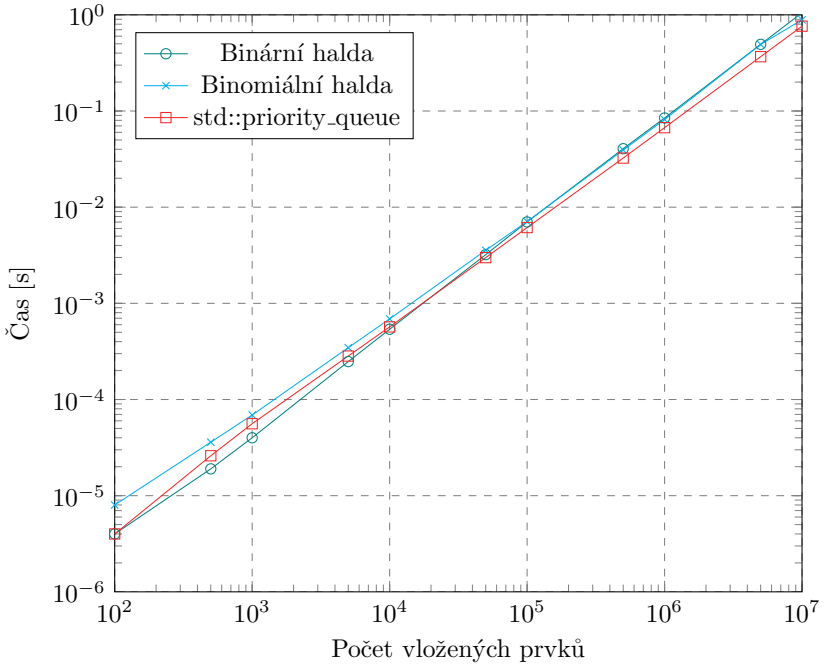
Obrázek 3.3: Nejpomalejší implementace v testu sestupně seřazené posloupnosti.

halda, jak je vidět v grafu 3.7. O něco lépe dopadl poslední ze stromů - *(2-3)-strom*, třebaže jeho operace slučování byla implementována pouze jako opakované vkládání a nikoliv jako postavení stromu ze seřazeného pole. To odpovídá zjištění z [4], kde v podobném testu také dopadl lépe strom s touto variantou slučování. Roli zde hraje fakt, že nedochází ke slučování dvou prioritních front s velkým množstvím prvků, ale velkého množství menších. To znamená, že značná část výsledného stromu byla postavena mnohokrát.

Lépe si vedla jednak *std::priority_queue* a také *neseřazené pole*. Vítězi jsou ale varianty prioritní fronty, které mají konstantní, případně logaritmickou časovou složitost operace slučování. Jedná se o *neseřazený spojový seznam* a komplikovanější typy hald (*binomiální*, *párovací*, a obě implementace *Fibonacciho*). V grafu 3.8 je z nich pro přehlednost znázorněn pouze spojový seznam, ostatní zmiňované struktury se nachází těsně za ním.

3.3.6 Obecné závěry

Za zmínku stojí, jak si vedly implementace vytvořené v rámci praktické části této práce v porovnání se dvěma knihovními implementacemi, které byly zahrnuty. Binární haldu lze přímo srovnat s std::priority_queue [29]. V grafu 3.9 je vidět, jak tyto struktury skončily v testu s převažujícím vkládáním. Obě binární haldy dosáhly podobných časů, byť v jiných testech operací vkládání

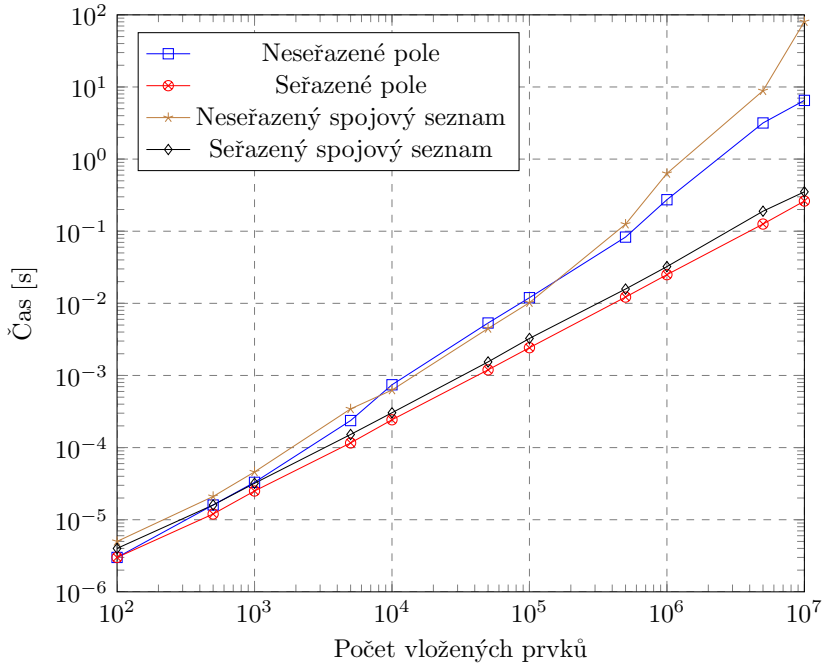


Obrázek 3.4: Nejrychlejší implementace v testu sestupně seřazené posloupnosti.

a mazání maxima knihovní implementace mírně vítězí. Implementace Fibonacciho haldy z knihovny `boost` se ukázala ve všech testech jako rychlejší (s výjimkou testu slučování, kde ale rozdíl není velký).

Zajímavé je rovněž srovnání vyhledávacích stromů. Obyčejný BVS propadl v testech vkládání seřazených posloupností, pro náhodné hodnoty ale dopadl srovnatelně s vyvažovanými variantami. Není to překvapivé, jelikož očekávaná hloubka BVS postaveného z n různých náhodných klíčů je $O(n)$ [1] - v tomto testu se sice ve stromu mohou objevit duplikáty, ale na výsledkách měření je vidět, že hloubka stromu se držela u této meze. (2-3)-strom si pak vždy vedl o něco hůře než AVL a červeno-černý strom.

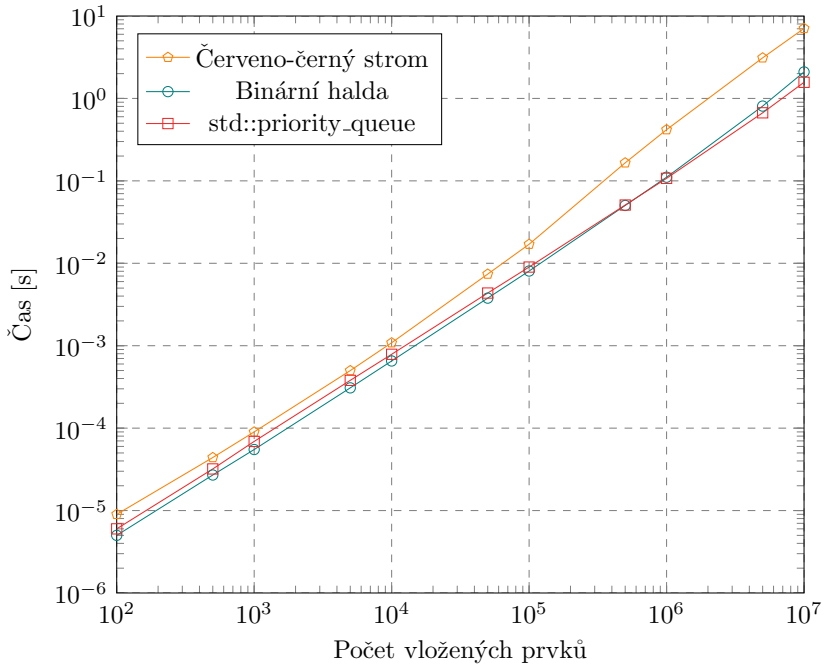
Z výsledků plyne, že ve specifických případech jsou nejrychlejší jednoduché datové struktury jako pole nebo spojové seznamy, jindy ale výrazně zaostávají. S výjimkou slučování se ukázalo že režie spojená se zvětšováním pole je menší než manipulace s ukazateli spojového seznamu. Pro náhodné prvky je možno za vítěze prohlásit binární haldu. Její nevýhoda je ale neefektivní slučování. Tento problém odpadá u složitějších typů hald. Například binomiální nebo párovací halda jsou kompromisy, které dosahují dobrých časů při vkládání a mazání i při slučování. Přestože vyhledávací stromy nejsou na rozdíl od hald primárně učené pro použití jako prioritní fronta, zejména červeno-černý strom za nimi ve většině testů nezaostával.



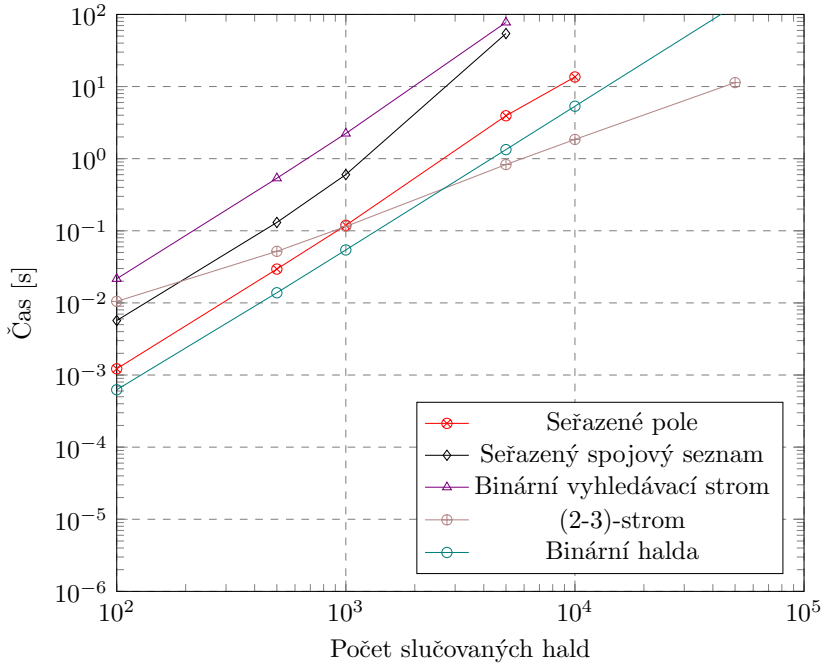
Obrázek 3.5: Vybrané výsledky testu náhodného vkládání a mazání.

Jak bylo řečeno, do testů nebyla zahrnuta *striktní Fibonacciho halda*. Z měření provedených v [30] vyplývá, že je velmi pomalá. V testu řazení a použití v Dijkstrově algoritmu skončila dokonce jako vůbec nejpomalejší ze všech testovaných hald. Na jejím příkladě je vidět, že datové struktury nelze posuzovat čistě dle asymptotických složitostí operací.

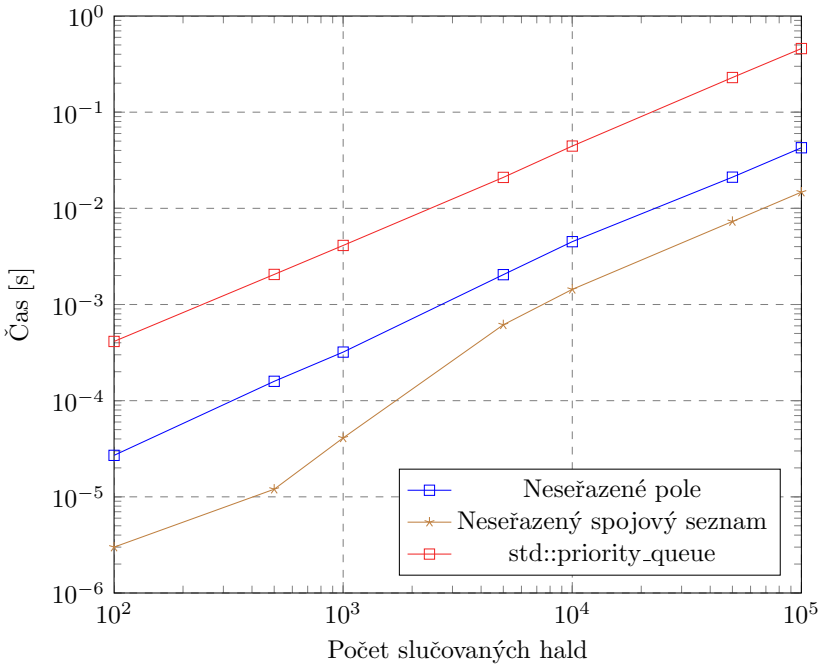
Stejnou problematikou se ve své práci zabýval Šimon Schierreich [4]. Při prostudování výsledků jeho práce vyšlo najevo, že výkonnost stejných typů prioritní fronty ve srovnatelných scénářích se někdy liší od časů naměřených v této práci. Může to být způsobeno odlišným nastavením překladače nebo implementačními detaily. Nikdy ale nejde o řádové rozdíly a z této práce nelze vyvodit žádná zjištění, která by byla přímo v rozporu ze zjištěními v práci Šimona Schierreicha.



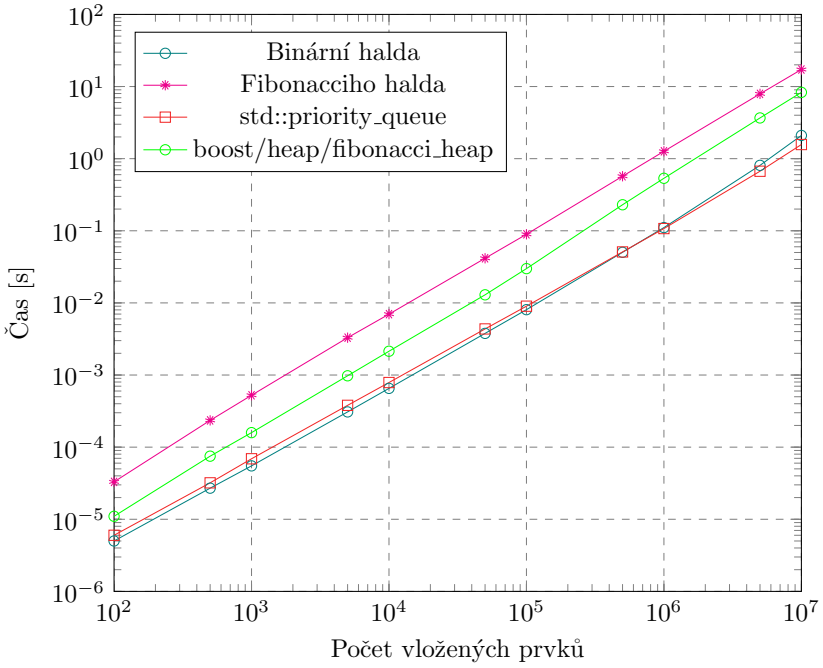
Obrázek 3.6: Vybrané výsledky testu s převažujícím vkládáním.



Obrázek 3.7: Nejpomalejší implementace v testu slučování.



Obrázek 3.8: Nejrychlejší implementace v testu slučování.



Obrázek 3.9: Srovnání s knihovními implementacemi.

Závěr

Cílem práce bylo nastudovat abstraktní datovou strukturu prioritní fronta, nastudovat a implementovat její varianty a změřit jejich praktickou výkonnost. Dalším cílem bylo popsat možnosti paralelizace prioritní fronty.

V práci bylo detailně popsáno množství různých implementací a s jednou výjimkou se je všechny podařilo implementovat. Striktní Fibonacciho halda se ukázala jako velmi složitá a díky nedostupnosti její vzorové implementace v jakémkoli programovacím jazyce se ji nepodařilo implementovat.

V rámci praktické části proběhlo i měření výkonnosti implementovaných datových struktur v několika scénářích. Ukázalo se, že při výběru implementace prioritní fronty se nelze řídit pouze asymptotickými složitostmi operací, protože v praxi se mohou na výkonnosti negativně projevit skryté konstanty. Práce podrobně rozebrala výsledky všech měření. Velmi dobrých výsledků dosáhly zejména haldy, jmenovitě binární nebo párovací halda. Ve specifických případech překvapily svou rychlostí velmi jednoduché datové struktury jako je seřazené pole.

Práce rovněž poskytuje stručný přehled dosavadního výzkumu na poli paralelizace prioritní fronty, což není v českém prostředí příliš diskutované téma.

Na práci se dá navázat nastudováním a implementováním jiných datových struktur vhodných pro použití jako prioritní fronta. Rovněž je možno implementovat i operace pro změnu klíče a mazání libovolného prvku a provést měření jejich výkonnosti, například při použití v Dijkstrově algoritmu nebo jiných reálných aplikacích nebo algoritmech. Zajímavé by bylo také naimplementovat striktní Fibonacciho haldu, která zde byla pouze popsána. Přestože implementace vytvořené v rámci této práce jsou šablonové, měření proběhlo pouze na jednom primitivním datovém typu. Dalo by se tedy provést měření i pro různé uživatelské datové typy.

Literatura

[1] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms*. The MIT Press, třetí vydání, 2009, ISBN 978-0-262-03384-8.

[2] Skiena, S. S.: *The Algorithm Design Manual*. Springer London Ltd, druhé vydání, 2008, ISBN 978-1848000698.

[3] Mareš, M.; Valla, T.: *Průvodce labyrintem algoritmů*. CZ.NIC, první vydání, 2017, ISBN 978-80-88168-19-5.

[4] Schierreich, S.: *Praktická výkonnost různých implementací prioritní fronty*. Bakalářská práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

[5] Sedgewick, R.; Wayne, K.: *Algorithms*. Pearson Education, čtvrté vydání, 2011, ISBN 978-0-321-57351-3.

[6] Bender, E. A.; Williamson, S. G.: *Lists, Decisions and Graphs - With an Introduction to Probability*. University of California San Diego, 2010. Dostupné z: `http://cseweb.ucsd.edu/~gill/BWLectSite/Resources/LDGbookCOV.pdf`

[7] Brass, P.: *Advanced Data Structures*. Cambridge University Press, 2008, ISBN 9780521880374.

[8] Goodrich, M. T.; Tamassia, R.; Goldwasser, M. H.: *Data Structures and Algorithms in Java*. Wiley, 6 vydání, 2014, ISBN 978-1-118-80314-1.

[9] Mehlhorn, K.; Sanders, P.: *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2007, ISBN 978-3540779773.

[10] Sack, J. R.; Strothotte, T.: An algorithm for merging meaps. *Acta Informatica*, ročník 22, 1985: str. 171–186. Dostupné z: `https://doi.org/10.1007/BF00264229`

[11] Kuszmaul, C. L.: Efficient Merge and Insert Operations for Binary Heaps and Trees. 2000. Dostupné z: https://www.nas.nasa.gov/assets/pdf/techreports/2000/nas-00-003.pdf

[12] Vuillemin, J.: A data structure for manipulating priority queues. *Communications of the ACM*, ročník 21, č. 4, 1978: str. 309–315. Dostupné z: https://doi.org/10.1145/359460.359478

[13] Melka, J.: *Fibonacciho haldy - jejich varianty a alternativní datové struktury*. Diplomová práce, Univerzita Karlova v Praze, Matematicko-fyzikální fakulta, 2012.

[14] Fredman, M. L.; Tarjan, R. E.: Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, ročník 34, č. 3, 1987: str. 596–615, ISSN 0001-0782. Dostupné z: https://dl.acm.org/doi/10.1145/28869.28874

[15] Fredman, M. L.; Sedgewick, R.; Sleator, D. D.; aj.: The Pairing Heap: A New Form of Self-Adjusting Heap. *Algorithmica*, ročník 1, č. 1, 1986: s. 111–129. Dostupné z: https://doi.org/10.1007/BF01840439

[16] Fredman, M. L.: On the Efficiency of Pairing Heaps and Related Data Structures. *Journal of the ACM*, ročník 46, č. 4, 1999, ISSN 0004-5411. Dostupné z: https://dl.acm.org/doi/10.1145/320211.320214

[17] Stasko, J. T.; Vitter, J. S.: Pairing heaps: experiments and analysis. *Communications of the ACM*, ročník 30, č. 3, 1987: str. 234–249, ISSN 0001-0782. Dostupné z: https://dl.acm.org/doi/10.1145/214748.214759

[18] Pettie, S.: Towards a final analysis of pairing heaps. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, 2005, ISBN 0-7695-2468-0, str. 174–183. Dostupné z: https://doi.org/10.1109/SFCS.2005.75

[19] Brodal, G. S.; Lagogiannis, G.; Tarjan, R. E.: Strict Fibonacci Heaps. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, New York, NY, USA: Association for Computing Machinery, 2012, ISBN 9781450312455, str. 1177–1184. Dostupné z: https://doi.org/10.1145/2213977.2214082

[20] Hart, P. E.; Nilsson, N. J.; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, ročník 4, č. 2, 1968: s. 100–107. Dostupné z: https://doi.org/10.1145/1056777.1056779

[21] Brodal, G.: Priority queues on parallel machines. *Parallel Computing*, ročník 25, č. 8, 1999: s. 987–1011, ISSN 0167-8191. Dostupné z: https://doi.org/10.1016/S0167-8191(99)00032-0

[22] Eppstein, D.; Galil, Z.: Parallel Algorithmic Techniques for Combinatorial Computation. In *Annual Review of Computer Science*, ročník 3, USA: Annual Reviews Inc., 1988, ISBN 0824332032, str. 233–283. Dostupné z: https://dl.acm.org/doi/10.5555/53594.53603

[23] Yu, L.: A Survey on Parallel Algorithms for Priority Queue Operations. 12 2003. Dostupné z: https://www.researchgate.net/publication/2870764_A_Survey_on_Parallel_Algorithms_for_Priority_Queue_Operations

[24] Pinotti, M. C.; Pucci, G.: Parallel Priority Queues. *Inf. Process. Lett.*, ročník 40, 1991: s. 33–40. Dostupné z: https://www.semanticscholar.org/paper/Parallel-Priority-Queues-Pinotti-Pucci/25acf77ba056a4617a80355c8f5b7ce0ccad6d97

[25] Tarjan, R. E.: *Data Structures and Network Algorithms*. USA: Society for Industrial and Applied Mathematics, 1983, ISBN 0898711878.

[26] Chen, D. Z.; Hu, X. S.: FAST AND EFFICIENT OPERATIONS ON PARALLEL PRIORITY QUEUES. *Parallel Processing Letters*, ročník 06, č. 04, 1996: s. 451–467. Dostupné z: https://doi.org/10.1142/S012962649600042X

[27] Das, S. K.; Pinotti, M. C.; Sarkar, F.: Optimal and Load Balanced Mapping of Parallel Priority Queues in Hypercubes. *IEEE Trans. Parallel Distrib. Syst.*, ročník 7, č. 6, Červen 1996: str. 555–564, ISSN 1045-9219. Dostupné z: https://doi.org/10.1109/71.506694

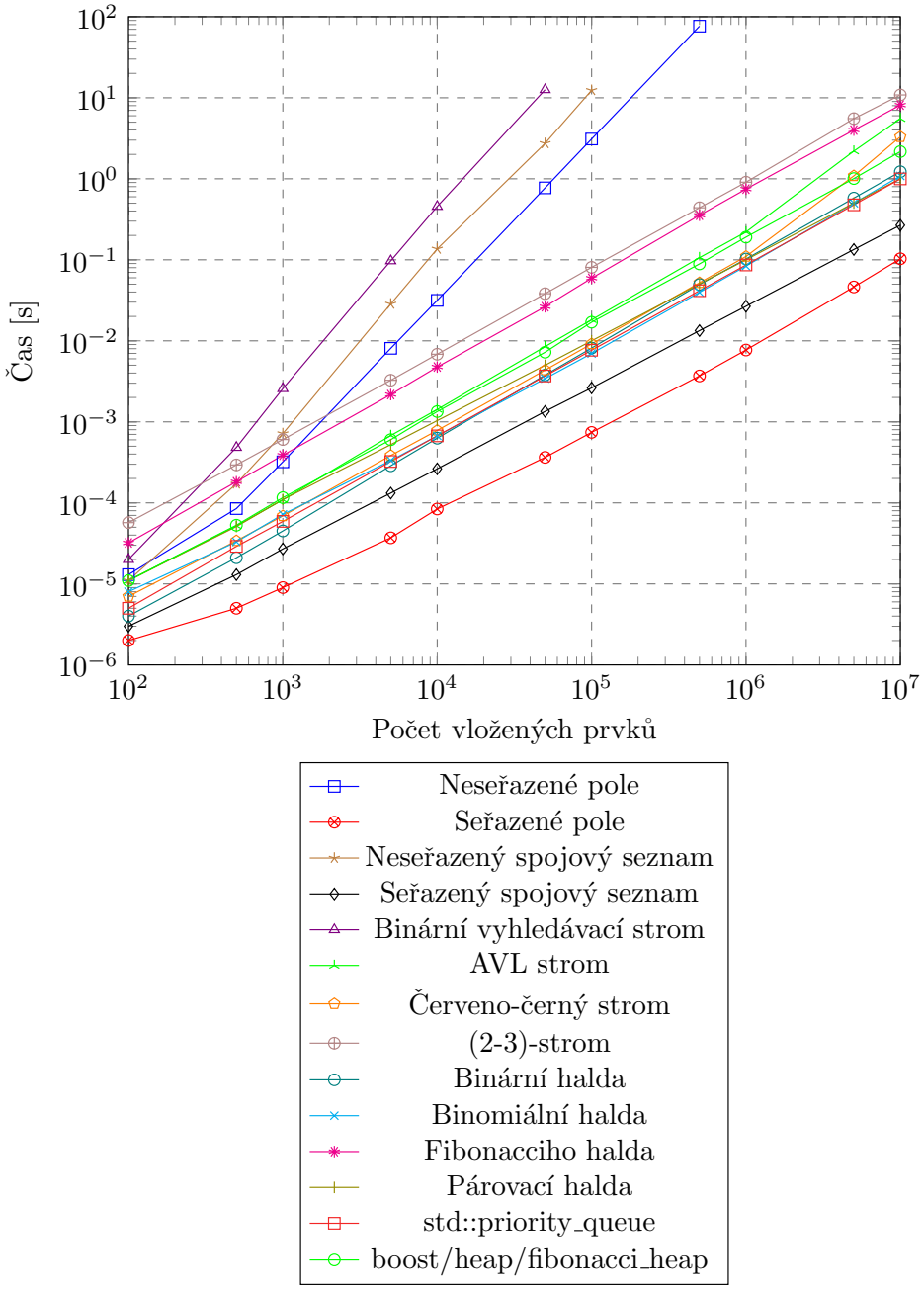
[28] Brodal, G. S.; Träff, J. L.; Zaroliagis, C. D.: A Parallel Priority Queue with Constant Time Operations. *J. Parallel Distrib. Comput.*, ročník 49, č. 1, Únor 1998: str. 4–21, ISSN 0743-7315. Dostupné z: https://doi.org/10.1006/jpdc.1998.1425

[29] std::priority_queue [online] 2021 [cit. 2021-04-17]. https://en.cppreference.com/w/cpp/container/priority_queue.

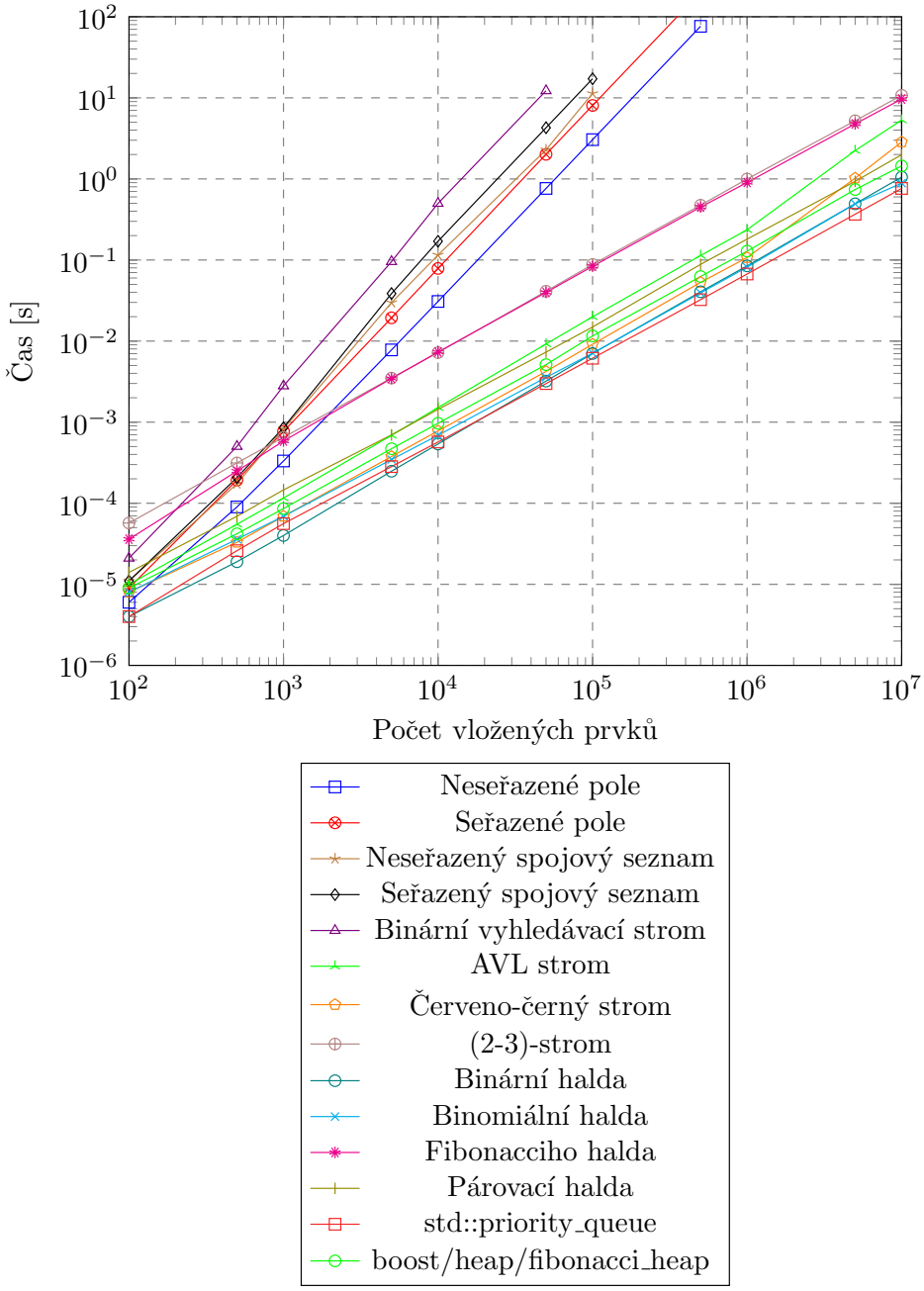
[30] Larkin, D. H.; Sen, S.; Tarjan, R. E.: A Back-to-Basics Empirical Study of Priority Queues. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, USA: Society for Industrial and Applied Mathematics, 2014, s. 61–72. Dostupné z: https://dl.acm.org/doi/10.5555/2790174.2790181

Příloha A

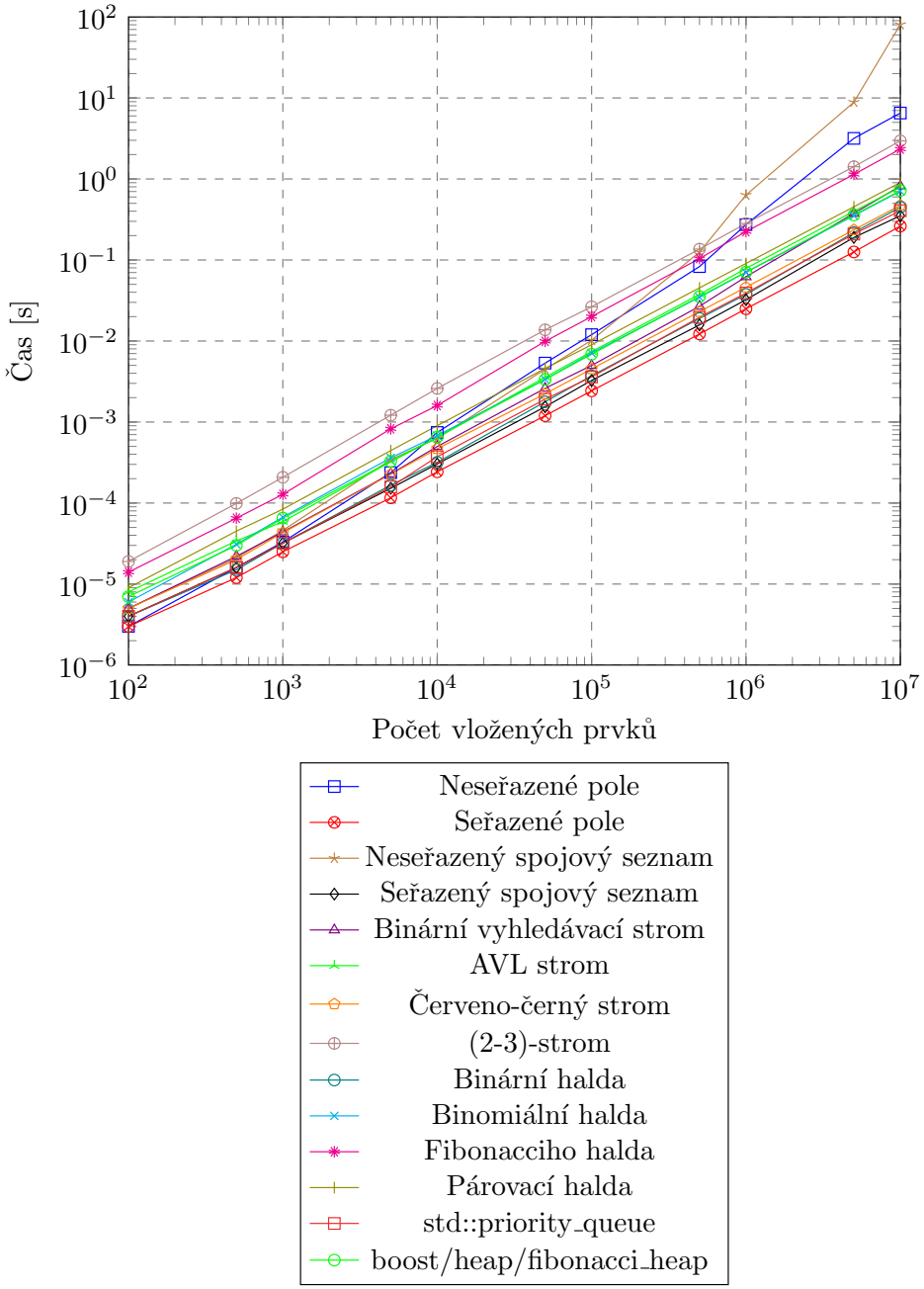
Kompletní výsledky měření



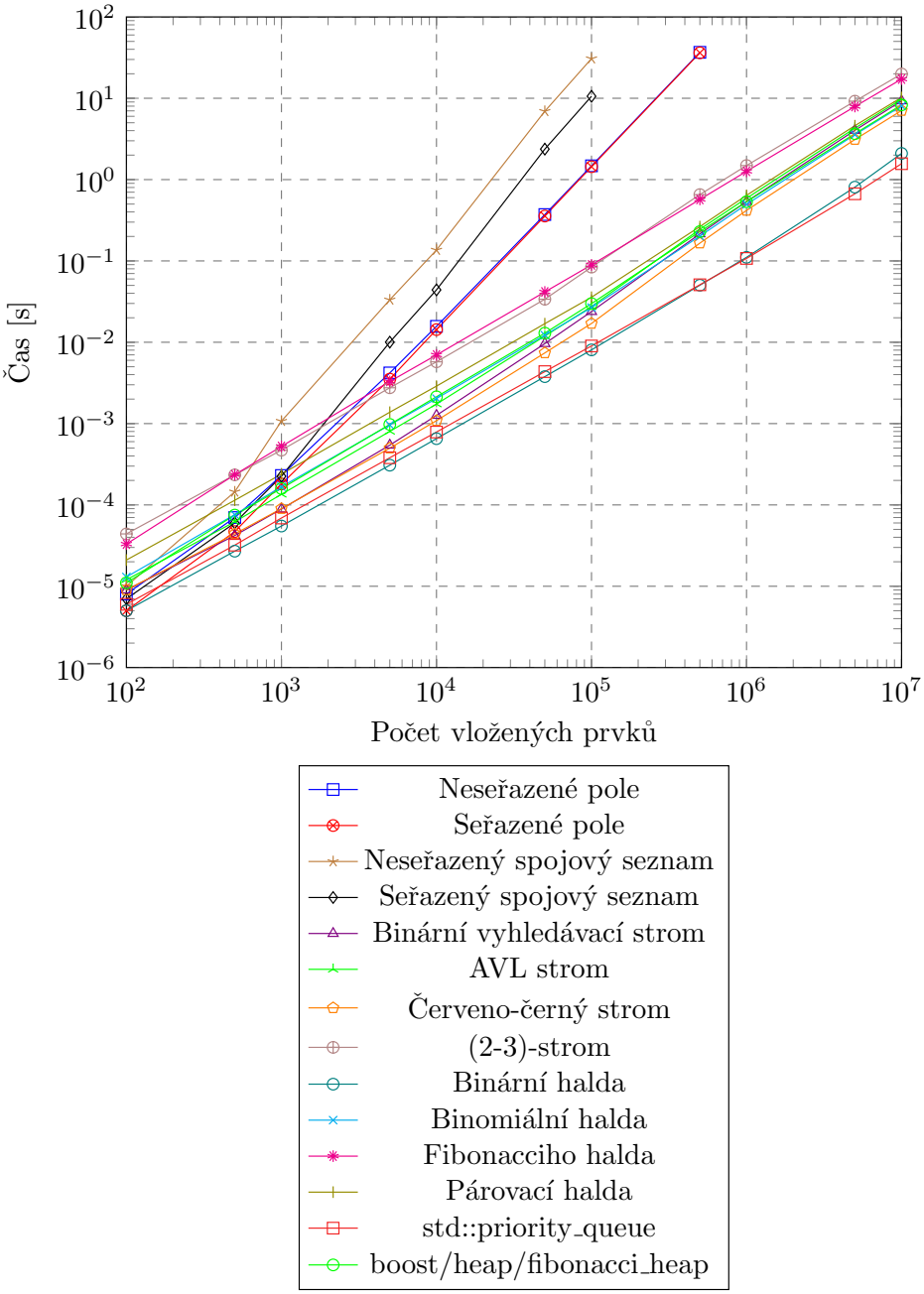
Obrázek A.1: Výsledky testu vzestupně seřazené posloupnosti.



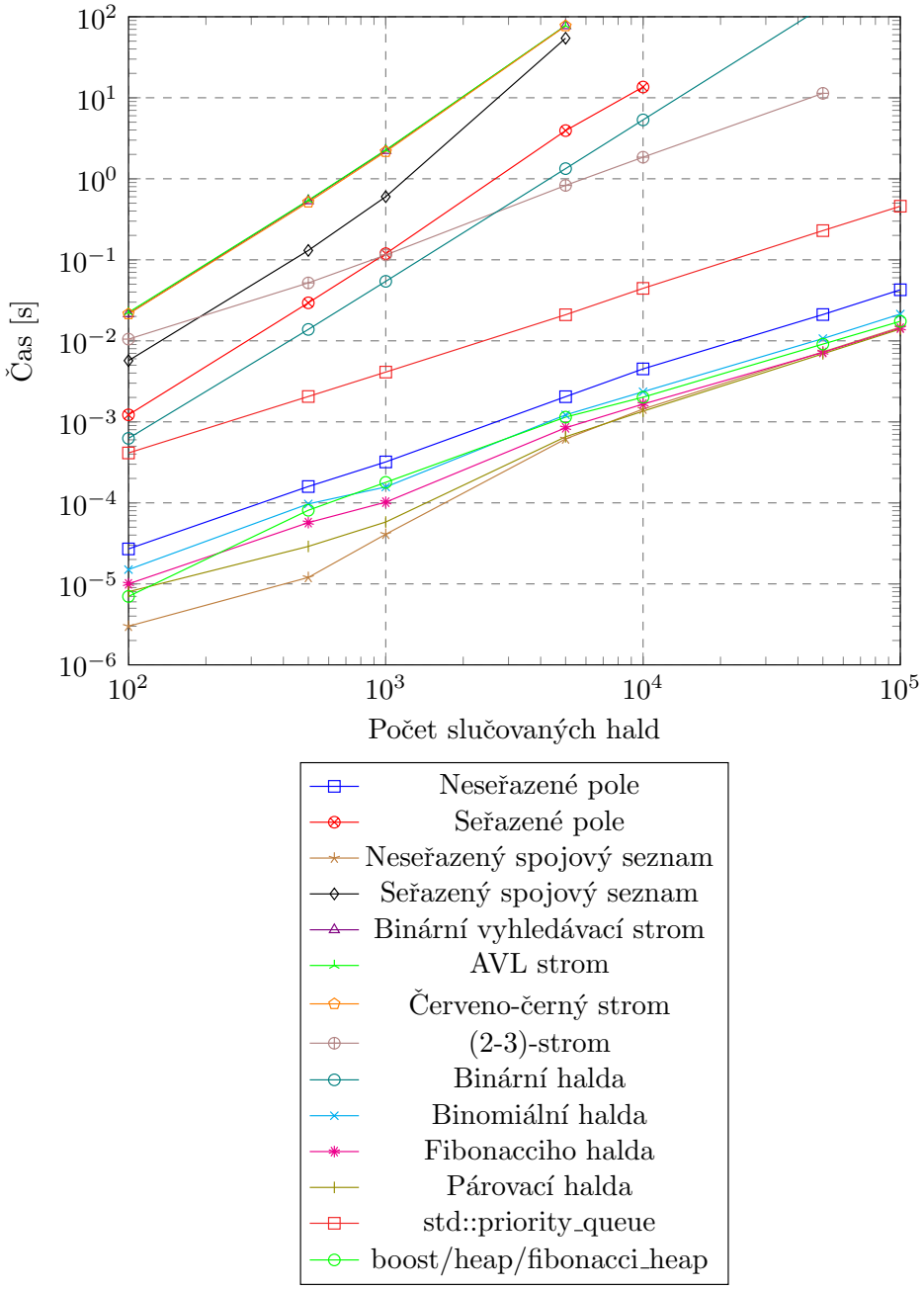
Obrázek A.2: Výsledky testu sestupně seřazené posloupnosti.



Obrázek A.3: Výsledky testu náhodného vkládání a výběru (1:1).



Obrázek A.4: Výsledky testu náhodného vkládání a výběru s převažujícím vkládáním (4:1).



Obrázek A.5: Výsledky testu slučování.

Příloha B

Seznam použitých zkratek

FIFO First In, First Out

LIFO Last In, First Out

BVS Binární vyhledávací strom

PRAM Parallel random-access machine

EREW Exclusive read exclusive write

CREW Concurrent read exclusive write

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src` ..... zdrojové kódy
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `data` ..... kompletní výsledky měření
  - `results.csv` ..... výsledky ve formátu CSV
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF