FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název: Praktická výkonnost různých implementací prioritní fronty
Student: Šimon Schierreich
Vedoucí: doc. Ing. Ivan Šimeček, Ph.D.
Studijní program: Informatika
Studijní obor: Teoretická informatika
Katedra: Katedra teoretické informatiky
Platnost zadání: Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Nastudujte abstraktní datovou strukturu prioritní fronta [1,2,4].
2) Nastudujte různé možnosti implementace prioritní fronty [1,2,3,5].
3) Jednotlivé varianty z bodu 2) implementujte v jazyce C++.
4) Porovnejte výkonnost jednotlivých implementací pro různé typy vstupních dat, jejich velikost a různý poměr prováděných operací na školním serveru STAR.
5) Diskutujte dosažené výsledky.

Seznam odborné literatury

[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

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 7. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Praktická výkonnost různých implementací prioritní fronty

Šimon Schierreich

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

15. května 2018

Poděkování

Rád bych na tomto místě poděkoval několika lidem, kteří umožnili, aby tato práce vznikla.

V prvé řadě patří mé díky doc. Ing. Ivanu Šimečkovi, PhD. za odvahu, s jakou se pustil do naší spolupráce, za množství komentářů a ochotu při každé konzultaci.

Další, kdo si zaslouží být na těchto řádcích zmíněn, jsou RNDr. Tomáš Valla, Ph.D. a RNDr. Ondřej Suchý, Ph.D., totiž za to, že mi jako první otevřeli cestu k nekonečným dobrodružstvím teoretické informatiky.

Nic by ovšem nemohlo vzniknout, nebýt mých rodičů, kteří ve mě po celou dobu studia věřili a všemožně mě podporovali. Jim patří můj největší dík.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické 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. § 46 odst. 6 tohoto zákona 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 Lysolajích dne 15. května 2018 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2018 Šimon Schierreich. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Schierreich, Šimon. *Praktická výkonnost různých implementací prioritní fronty.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce se zabývá měřením reálné výkonnosti vybraných implementací prioritní fronty pro rozličná vstupní data a různý poměr prováděných operací.

Implementace jednotlivých variant proběhla v programovacím jazyce C++ a byla měřena na školním výpočetním svazku STAR s procesorem Intel Core i7-950, 24 GB DDR3 RAM a dvojicí grafických karet GeForce GTX 590 a GeForce GTX 470.

Vytvořená řešení byla otestována podle několika typických scénářů, došlo k diskuzi naměřených hodnot a jejich porovnání s teoretickými hranicemi výkonnosti.

Na základě naměřených hodnot je možné optimalizovat kritické části algoritmů využívající zkoumanou datovou strukturu a ušetřit tak výpočetní čas a zprostředkovaně i spotřebovanou energii.

Klíčová slova prioritní fronta, halda, binární vyhledávací strom, Fibonacciho halda, párovací halda, striktní Fibonacciho halda, měření výkonnosti

Abstract

In this thesis we measure the practical efficiency of selected implementations of priority queue for different input data and various ratios of performed operations.

The solution is implemented in the C++ programming language and the performance is measured on the school computing cluster named STAR with Intel Core i7-950 processor, 24 GB DDR3 RAM and two graphic cards GeForce GTX 590 and GeForce GTX 470.

The created solutions are tested using some typical scenarios, measured values are discussed and compared with the theoretical bounds of complexity.

Due to the measured values one can optimize critical parts of the algorithms that use the examined data structure in order to save computing resources and consumed energy.

Keywords priority queue, heap, binary search tree, Fibonacci heap, pairing heap, strict Fibonacci heap, performance measurement

Obsah

Úvod **1**
Cíle práce 1
Struktura práce 1

1 Prioritní fronta **3**
1.1 Podporované operace 3
1.2 Teoretická hranice složitosti 4
1.3 Využití 5

2 Implementace prioritní fronty **9**
2.1 Neřazené pole 9
2.2 Řazené pole 12
2.3 Neřazený spojový seznam 13
2.4 Řazený spojový seznam 13
2.5 Binární vyhledávací strom 14
2.6 Červeno-černý strom 17
2.7 Binární halda 22
2.8 Binomiální halda 24
2.9 Fibonacciho halda 29
2.10 Párovací halda 31
2.11 Striktní Fibonacciho halda 33

3 Měření výkonnosti **39**
3.1 Test řazením 39
3.2 Test vkládání a výběru 43
3.3 Test spojování 44
3.4 Diskuze dosažených výsledků 45

Závěr **47**

Literatura **49**

A Slovník **53**

B Obsah přiloženého CD **55**

Seznam obrázků

2.1 Základní operace prioritní fronty implementované neřazeným polem. 11
2.2 Příklady různých binárních vyhledávacích stromů 15
2.3 Příklady vkládání do červeno-černého stromu, při kterých je porušena některá z jeho podmínek 19
2.4 Ukázka rotací uzlů binárního stromu 20
2.5 Binomiální stromy různých řádů 25
2.6 Příklad slévání dvou binomiálních stromů řádu $k = 2$ 27
2.7 Slévání dvou binomiálních hald 28
2.8 Struktura Fibonacciho haldy 29
2.9 Reprezentace stromů v párovací haldě. 31
2.10 Mazání maxima z párovací haldy 32
2.11 Transformace striktních Fibonacciho hald 36

3.1 Výsledky testu řazení opačně seřazenou posloupností 40
3.2 Výsledky testu řazení náhodnými daty 41
3.3 Výsledky testu řazení seřazenou posloupností 42
3.4 Výsledky testu vkládání a výběru 43
3.5 Výsledky testu spojování 44

Seznam tabulek

2.1 Přehled asymptotických složitostí základních operací pro jednotlivé implementace 38
2.2 Přehled asymptotických složitostí rozšiřujících operací pro jednotlivé implementace 38

3.1 Konfigurace měřicího serveru 39

Úvod

Prioritní fronta je hlavním proudem výzkumníků v oblasti návrhu a studia datových struktur lehce přehlížená datová struktura. Zaslouží si to snad svou zdánlivou jednoduchostí.

I když je myšlenka, na které je prioritní fronta postavená, v zásadě jednoduchá, její efektivní implementace již tak triviální není. Ze zkušenosti se totiž zdá, že čím lepší časové složitosti ta která varianta dosahuje, tím je i její implementace složitější. A větší implementační složitost s sebou nese, krom větší pravděpodobnosti programátorské chyby, i více instrukcí, které je potřeba kvůli jedné operaci provést.

Nabízí se tedy otázka, zda-li se pro běžné využití vyplatí asymptoticky rychlejší, ale implementačně řádově složitější varianty prioritní fronty.

Cíle práce

Hlavním cílem této práce je nalézt nejrychlejší variantu prioritní fronty pro aplikaci v několika vybraných algoritmech. Prakticky to znamená, že dojde k implementaci a následné optimalizaci vybraných variant prioritní fronty, měření jejich reálné výkonnosti nad různými daty a porovnání jejich výkonnosti.

Po přečtení této práce a na základě znalosti problematické domény, by měl být programátor schopný vybrat nejrychlejší možnou implementaci prioritní fronty.

Struktura práce

V kapitole 1 bude detailně rozebrána prioritní fronta, dojde k představení všech operací, které tato datová struktura podporuje. Nesmí chybět i detailní analýza teoretických hranic složitostí jednotlivých operací. V závěru dojde k představení několika konkrétních aplikací.

Druhá kapitola představí různé varianty prioritní fronty lišící se zejména asymptotickou časovou a paměťovou složitostí jednotlivých podporovaných operací a v neposlední řadě i složitostí implementace. Dojde také k implementaci zmíněných variant v jazyce C++ .

Poslední kapitola se bude skládat z měření reálné výkonnosti dříve implementovaných variant prioritní fronty a diskuze dosažených výsledků.

Kapitola 1

Prioritní fronta

Prioritní fronta je abstraktní datová struktura, která umožňuje výběr vložených prvků v pořadí určeném hodnotou jejich klíčů. Lisí se tak od jiných ADT, jako jsou například fronta (FIFO) a zásobník (LIFO), kde jsou prvky vybírány na základě času vložení, nebo tabulky (slovníku, mapy), kde jsou prvky vybírány na základě rovnosti klíčů.[1, s. 373]

Každý prvek vložený do prioritní fronty musí obsahovat klíč, který slouží k určení priority[2]. Tyto klíče navíc musí tvořit úplně uspořádanou množinu, jinak by nebylo možné určit správné pořadí, ve kterém budou jednotlivé prvky z prioritní fronty vybírány.[1]

Prioritní fronta se objevuje ve dvou hlavních variantách v závislosti na tom, jakým způsobem jsou klíče chápány. Konkrétně se jedná o maximovou a minimovou prioritní frontu, přičemž v případě maximové je na začátku fronty vždy prvek s největší hodnotou klíče, naopak u minimové je to ten s nejmenší hodnotou klíče.[2]

V dalším textu bude, pokud nebude řečeno jinak, uvažována vždy maximová prioritní fronta.

1.1 Podporované operace

Mezi základní operace podporované ADT prioritní fronta patří následující:[3]

- `enqueue(Q,x)` – Provede vložení nového prvku x do prioritní fronty Q.
- `front(Q)` – Vrátí prvek z fronty Q s nejvyšší prioritou. Prvek ve frontě nadále zůstává.
- `dequeue(Q)` – Odstraní z fronty Q prvek s nejvyšší prioritou.

Pro některé aplikace jsou často užitečné i další operace:[4]

- `change_key(Q,x,k)` – Změní klíč prvku x z fronty Q na hodnotu k. Hodnota k musí být vždy větší nebo rovna stávající hodnotě klíče prvku x.
- `delete(Q,x)` – Odstraní z fronty Q prvek x bez ohledu na jeho prioritu.
- `merge(Q1, Q2)` – Provede sloučení dvou prioritních front Q_1 a Q_2 do jedné.

Jak vidno, prioritní fronta nepodporuje žádný způsob prohledávání uložených prvků. Operacím `delete` a `change_key` je tedy třeba předat ukazatel na prvek, se kterým mají pracovat.[2]

1.2 Teoretická hranice složitosti

Obecně složitost jednotlivých operací prioritní fronty záleží na její konkrétní implementaci. Je ovšem možné, díky znalosti spodní hranice složitosti obecného, na porovnání založeného řadícího algoritmu, určit spodní hranice složitosti kombinací jednotlivých operací.

Ze všeho nejdříve budiž připomenuta spodní hranice asymptotické složitosti řazení.

Definice 1 (Spodní hranice složitosti řazení)**.** *Any comparison sort algorithm requires $\Omega(n \cdot \log n)$ comparisons in the worst case.[2, s. 193]*

Důkaz předchozí definice na těchto stránkách nebude uveden, čtenář ho ovšem může najít ve zdrojové literatuře.

Věta 1 (Spodní hranice složitosti prioritní fronty)**.** *Při vložení n náhodných prvků do prioritní fronty a jejich následném vybrání bude složitost kombinace těchto dvou operací $\Omega(n \log n)$.*

K důkazu předchozí věty bude využita právě uvedená definice 1 spodní hranice složitosti řazení.

Důkaz. Mějme prioritní frontu podporující operace `enqueue` a `dequeue` popsané v kapitole 1.1 a n náhodných prvků, které mají být do fronty vloženy. Následně dojde k vyjmutí vložených prvků. Složitost vložení všech n prvků pak bude $\Omega(n \cdot g(n))$, kde $g(n)$ je funkce složitosti vložení jednoho prvku, a složitost vybrání všech vložených prvků je $\Omega(n \cdot h(n))$, kde $h(n)$ je funkce složitosti vyjmutí jednoho prvku.

Po provedení této operace dojde k seřazení všech prvků podle hodnoty jejich klíčů. Složitost řazení musí být pro nejhorší případ nejlépe $\Omega(n \log n)$, jinak by bylo možné pomocí takové prioritní fronty vytvořit obecný řadící algoritmus fungující v čase lepším, než $\Omega(n \log n)$, což by byl ale spor s definicí 1. □

Věta 1 ještě neříká nic o tom, jaké jsou hranice složitosti pro jednotlivé operace.

Věta 2. *Alespoň jedna z operací* `enqueue` *a* `dequeue` *musí mít složitost* $\Omega(\log n)$.

Důkaz. Uvažujme implementaci prioritní fronty takovou, že $g(n) \in o(\log n)$ a zároveň $h(n) \in o(\log n)$, kde $g(n)$ a $h(n)$ jsou složitosti operací `enqueue`, resp. `dequeue`. Potom po provedení n vložení a stejného počtu vyjmutí z takové prioritní fronty dostaneme celkovou složitost

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

Jelikož platí, že $\Omega(n) \cdot o(\log n)$ je asymptoticky menší, než $\Omega(n \cdot \log n)$, došli jsme ke sporu s dříve dokázanou větou 1 a platí tedy původní tvrzení, tedy že alespoň jedna z operací `enqueue` a `dequeue` musí mít složitost $\Omega(\log n)$. □

1.3 Využití

Prioritní fronta najde využití všude tam, kde nějaký algoritmus potřebuje zpracovávat prvky v daném pořadí, ale počet prvků není předem znám a dochází k jejich dynamickému přidávání v průběhu zpracovávání.

Typická situaci vypadá tak, že na začátku existuje množinu prvků, ten s největší prioritou je z ní odebrán a začne jeho zpracovávání. Při zpracovávání může dojít k vytvoření nových prvků, které jsou do množiny také přidány. Následně se pokračuje opět s prvkem s největší prioritou. Toto se opakuje, dokud nedojde ke zpracování celé množiny.[5]

Nejčastěji se prioritní fronta využívá v oblastech jako jsou plánování procesů, teorie front, prohledávání grafů, při kompresi dat nebo třeba výpočetní teorii čísel.[6]

V další části této kapitoly bude představeno několik konkrétních aplikací prioritní fronty.

1.3.1 Dijkstrův algoritmus

Dijkstrův algoritmus[7] slouží k hledání nejkratší cesty mezi dvěma vrcholy P a Q v grafu s kladně ohodnocenými hranami.

Všechny vrcholy grafu jsou rozděleny do třech množin:

- A Vrcholy, pro které je délka nejkratší cesty z P známá.
- B Množina vrcholů, z kterých bude v příští iteraci jeden z vrcholů přidán do množiny A.
- C Zbývající vrcholy.

Na začátku jsou všechny vrcholy v množině C. Jako první je do množiny A přidán počáteční vrchol P a poté jsou opakovaně prováděny následující kroky:

1. Nechť R označuje poslední vrchol vložený do množiny A.
2. Pro všechny sousedy $S \in B$ vrcholu R dojde, v případě, že je délka cesty z P do S přes R kratší, než aktuální nejkratší cesta z P do S, k přepočítání délky nejkratší cesty.
3. Všichni sousedé $S \in C$ vrcholu R jsou vloženy do množiny B a je nastavena odpovídající délka nejkratší cesty z P.
4. Vrchol T z množiny B s nejmenší vzdáleností od P je přesunut do množiny A.
5. Pokud neplatí, že vrchol T z předchozího bodu je právě Q, pokračuje se opět s krokem 1.

Po provedení tohoto algoritmu dojde k nalezení nejkratší cesty z P do Q.[7]

Z dynamické množiny B jsou v případě Dijkstrova algoritmu vrcholy vybírány podle toho, ke kterému vede nejkratší cesta z počátečního vrcholu P, což je často implementováno právě prioritní frontou.[4]

Při lineárním prohledávání množiny B je asymptotická složitost Dijkstrova algoritmu $\mathcal{O}(|V^2|)$, zatímco při použití prioritní fronty realizované Fibonacciho haldou je to $\mathcal{O}(|E| + |V| \log |V|)$.[2]

1.3.2 Huffmanovo kódování

Huffmanovo kódování[8] je algoritmus využívaný pro bezztrátovou kompresi. Podle charakteru vstupních dat dokáže ušetřit mezi 20% a 90% objemu.[2]

Základem pro kompresi nějakého textu S je vytvoření tzv. Huffmanova stromu, jehož konstrukce probíhá následovně.

Nechť C je množina všech různých znaků v textu S a pro každé $c \in C$ platí, že se jedná o objekt obsahující atribut $c.freq$, který udává frekvenci jeho výskytů v textu S. Navíc ať Q značí minimovou prioritní frontu, ve které jsou jednotlivé znaky uspořádány podle frekvence výskytu.

Na začátku tvorby Huffmanova stromu jsou do prioritní fronty Q vloženy všechny znaky $c \in C$. Následně dojde k vytvoření nového uzlu z a vybrání dvou prvků x, y z fronty Q s největší prioritou. Prvek x je nastaven jako levý potomek uzlu z, zatímco y se stane pravým potomkem. Atribut $z.freq$ je pak roven součtu $x.freq + y.freq$. Nakonec dojde k vložení uzlu z do fronty Q. Po provedení $|C| - 1$ iterací předchozího algoritmu vzniká Huffmanův strom a jeho kořen se nachází na začátku prioritní fronty Q.[2]

Listy vzniklého Huffmanova stromu jsou vždy původní znaky c z množiny C. Kód pro každý znak je potom tvořen cestou z kořene do odpovídajícího listu, přičemž za každý sestup do levého potomka je ke kódu přidána binární 0, resp. 1 za sestup do pravého potomka.

Asymptotická složitost tvorby Huffmanova stromu pro n znaků je, při použití prioritní fronty, $\mathcal{O}(n \log n)$. Nahrazením prioritní fronty van Emde Boas stromy lze dosáhnout asymptotické složitosti dokonce $\mathcal{O}(n \log \log n)$.[2]

1.3.3 Heapsort

Heapsort je nestabilní in-place řadící algoritmus postavený na operacích ADT halda.

Jak popsali již Fredman a Tarjan v [4], ve skutečnosti je halda ekvivalentní prioritní frontě. Dokonce jsou tyto dva pojmy některými autory zaměňovány.

Všechny hodnoty, které mají být seřazeny, jsou na začátku algoritmu vloženy do prioritní fronty. Následně jsou z fronty všechny vložené prvky vybrány, což probíhá, díky vlastnostem prioritní fronty, v pořadí od největšího po nejmenší. Tedy vzniká seřazená posloupnost hodnot.[9]

Jak vyplývá z kapitoly 1.2 a jak uvádí i [2], asymptotická složitost řazení haldou je $\mathcal{O}(n \log n)$. V praxi se ovšem častěji využívá spíše algoritmus quicksort, který má sice v nejhorším případě asymptotickou složitost $\mathcal{O}(n^2)$, nicméně pro průměrný případ pracuje, dle naměřených hodnot, výrazně rychleji.[2]

1.3.4 Diskrétní simulace

V případě diskrétních simulací je potřeba udržovat množinu čekajících událost, kde každá proběhne v předem určený čas a může vytvořit několik dalších budoucích událostí. Tyto události je vhodné udržovat právě v prioritní frontě Q, ve které za prioritu slouží čas provedení.

Hlavní smyčka programu pak v daný čas vybere všechny události s patřičnou prioritou, provede je a případně zařadí nově vygenerované události do Q.[10]

Kapitola 2

Implementace prioritní fronty

Jak bylo naznačeno již v předchozí kapitole, prioritní frontu lze implementovat několika způsoby, které se od sebe liší nejen časovou a paměťovou složitostí, ale například i složitostí implementace.

Na dalších stránkách budou jednotlivé implementace rozebrány a nakonec dojde i k jejich porovnání z hlediska asymptotických složitostí. Krom toho byly všechny datové struktury v rámci práce na kapitole implementovány v jazyce C++ . Ten byl vybrán zejména *z důvodu vysokého výkonu, kompilace do strojového kódu a manuální správy paměti. Automatická správa paměti a interpretovaný kód by mohli nepříznivě ovlivnit výkon datových struktur.*[11]

Zdrojové kódy jednotlivých implementací jsou k nalezení na přiloženém CD ve složce `/src/impl`. K dispozici je i `Makefile` sloužící ke kompilaci.

2.1 Neřazené pole

Neřazené pole je velice jednoduše implementovatelná a pochopitelná varianta prioritní fronty.

Všechny prvky jsou uloženy v datové struktuře pole a navíc je udržován ukazatel na největší prvek.

2.1.1 Vkládání a hledání maxima

Při vkládání nového prvku dojde k přidání vkládaného prvku na konec pole. Pokud je navíc hodnota klíče nového prvku větší, než hodnota klíče stávajícího maxima, dojde k aktualizaci ukazatele na největší prvek. Tato operace má, za předpokladu, že má pole dostatečnou kapacitu, časovou složitost $\mathcal{O}(1)$, jelikož dochází pouze k vložení nového prvku do pole, jednomu porovnání a případné aktualizaci jednoho ukazatele.[1]

V případě, že by v poli nezbývalo žádné volné místo, bude třeba ho rozšířit. To většinou probíhá vytvořením nového pole s větší kapacitou, do kterého jsou všechny prvky původního pole překopírovány. Následně je staré pole zrušeno.

Je snadné nahlédnout, že v tomto případě bude složitost vkládání nového prvku lineární s počtem jeho prvků.

Často je ovšem uváděno, že vložení nového prvku do pole je operace s konstantní časovou složitostí. Jak je to možné, když bylo právě ukázáno, že v nejhorším případě bude mít vkládání lineární složitost?

Abstraktní datová struktura pole je v teorii běžně uvažována jako struktura s takřka neomezenou kapacitou, což v praxi pokulhává. Pokud by docházelo k alokování tak velkých polí, aby se z programátorova hlediska zdála neomezená, došlo by velmi rychle k vyčerpání veškeré volné paměti.

Z toho důvodu se využívá tzv. nafukovacího pole, což je pole s omezenou kapacitou, která se ve chvíli, kdy mu dojde kapacita, zvětší, většinou dvojnásobně, což je ovšem vykoupeno právě větší časovou náročností. Při opakovaném vkládání hodnot do takové struktury se pak bude časová složitost jedné samotné operace zdát opravdu konstantní, jak bude konečně ukázáno dále.

Věta 3 (O složitosti vkládání do nafukovacího pole)**.** *Vkládání prvků do nafukovacího pole má amortizovanou složitost $\mathcal{O}^*(1)$.*

Důkaz. K důkazu bude využita tzv. penízková metoda, která je blíže popsána v [2], [12] či [13]. Při každém vložení nového prvku budou na virtuální účet pole přidány dvě mince. Při kopírování prvků budou pak tyto mince spotřebovávány, konkrétně jedna mince za zkopírování každého jednoho prvku.

Na začátku je pole prázdné a má kapacitu 1. Po vložení nového prvku se počet mincí na účtu zvýší o dvě. Jelikož je v tuto chvíli pole zcela zaplněno, bude jeho kapacita zdvojnásobena a stávající prvek bude nakopírován do zvětšeného pole. Za to bude zaplacena jedna mince a na účtu tak ještě jedna mince zůstává. Kapacita pole je v tuto chvíli 2.

Následně je vložen další prvek, který opět přidá dvě mince na účet. Podobně jako v předchozím případě je v tuto chvíli pole maximálně zaplněné, dojde tedy k jeho rozšíření. Za kopírování dvou stávajících prvků budou zaplaceny dvě mince ze tří a kapacita pole se zvětší na 4.

Po provedení n vložení tedy dojde celkově k přičtení $2n$ mincí. Zároveň vždy ve chvíli, kdy $n = 2^k, k \in \mathbb{Z}$ dojde k odečtení 2^k mincí. Nechť $n = 2^i$, potom bude celkový počet mincí na účtě po provedení n operací vkládání roven

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

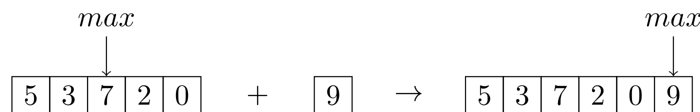
.

Bylo dokázáno, že hodnota účtu nikdy neklesne pod nulu, tedy na provedení série n vložení bude spotřebováno vždy $\mathcal{O}(n)$ času. Amortizovaně je tedy na jedno vložení potřeba pouze $\mathcal{O}^*(1)$ času. □

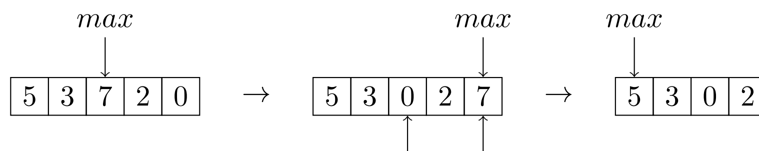
Získání prvku s největší prioritou je triviální, stačí vrátit ten, na který ukazuje maximový ukazatel.[1]

2.1.2 Mazání

Odebrání největšího prvku sestává ze dvou kroků. Nejdříve dojde k samotnému odebrání největšího prvku, což lze realizovat prohozením prvku s nejvyšší prioritou s posledním vloženým a následnou dekrementací počítadla počtu prvků v poli. Toto je realizovatelné v konstantním čase. Ve druhém, a o poznání náročnějším, kroku je třeba nalézt nové maximum. To nelze provést jinak, než prohledáním celého pole, tedy s asymptotickou složitostí $\mathcal{O}(n)$.[1]



(a) Nový prvek je přidán na konec pole. Jelikož je jeho hodnota větší, než stávající maximum, je aktualizován maximový ukazatel.



(b) Maximum je při odebírání prohozeno s posledním prvkem v poli. Následně dojde k jeho odtržení. Nové maximum je pak nalezeno lineárním prohledáním celého pole.

Obrázek 2.1: Základní operace prioritní fronty implementované neřazeným polem.

Smazaní libovolného prvku lze rozdělit na dva různé případy. Nastane-li varianta, kdy je mazán prvek s nejvyšší prioritou, bude složitost rovna odebírání největšího prvku, tedy $\mathcal{O}(n)$. V opačném případě bude prvek rovněž mazán jako v případě odstraňování největšího prvku, ovšem s tím rozdílem, že není třeba aktualizovat maximový ukazatel, složitost tedy bude konstantní.

2.1.3 Změna klíče a spojování

Změna klíče se velice podobá vložení nového prvku. Hodnota klíče měněného prvku je aktualizována a v případě, že je nová hodnota větší, než aktuálně největší klíč, dojde k aktualizaci maximového ukazatele. Operaci lze opět provést v konstantním čase.

Poslední operace, která byla představena, je sloučení dvou prioritních front. V případě neřazeného pole stačí obě pole zřetězit, což nelze provést jinak, než vložením všech prvků jedné fronty do druhé. Celková složitost je tedy $\mathcal{O}(n)$, kde n je součet počtu prvků obou front.

Paměťová složitost implementace prioritní fronty pomocí neřazeného pole je rovna počtu jejích prvků, tedy $\Theta(n)$.

2.2 Řazené pole

Mnoho algoritmů a datových struktur lze výrazně urychlit předzpracováním vstupních dat jejich řazením. Přímo se tedy nabízí zkusit vylepšit předchozí variantu implementace pomocí seřazeného pole.

Podobně jako v předchozím případě bude k implementaci třeba využít datové struktury pole a počítadlo prvků v poli. Pořadí prvků v poli bude navíc takové, že na nejnižším indexu bude prvek s nejnižší hodnotou klíče, tedy obráceně, než by možná bylo intuitivní.

2.2.1 Vkládání a hledání maxima

Při vkládání nového prvku bude inkrementováno počítadlo prvků a na první prázdné místo bude nový prvek vložen. Následně vložený prvek „probublá“ na svou pozici podobně, jako je tomu v případě algoritmu bublinkového řazení [14, s. 107]. Prvek je tedy prohazován s levým sousedem dokud platí, že hodnota jeho klíče je menší, než hodnota klíče souseda. Počet prohození je v nejhorším případě roven počtu prvků v prioritní frontě, a to ve chvíli, kdy je hodnota klíče nového prvku nejmenší. Asymptotická časová složitost vkládání je tedy $\mathcal{O}(n)$.

Získání prvku s nejvyšší prioritou sestává z pouhého navrácení prvku na indexu $n - 1$, kde n je hodnota čítače počtu prvků.[1]

2.2.2 Mazání maxima

Mazaní prvku, jehož priorita je největší, je v tomto případě podobně jednoduché, jako jeho získání. Stačí dekrementovat čítač počtu prvků. Celá operace proběhne, jak lze snadno nahlédnout, v konstantní čase.[1]

2.2.3 Změna klíče

Při aktualizaci hodnoty klíče nějakého prvku se musí změněný prvek dostat na správné místo v poli tak, aby bylo zachováno jeho řazení. K tomu využijeme stejný postup, jako v případě vkládání nového prvku, totiž dříve popsané „probublání“ na správné místo. Ze stejných důvodů, jako u vkládání nového prvku, bude asymptotická složitost rovna $\mathcal{O}(n)$.

2.2.4 Mazání obecného prvku

Mazání libovolného prvku lze realizovat i za použití řazeného pole. Nejdříve dojde k aktualizaci hodnoty klíče daného prvku na hodnotu $+\inf$, což způsobí jeho přesunutí na konec pole. Následným odebráním největšího prvku se původního prvku snadno zbavíme. Časová složitost takové operace sestává z časové složitosti změny klíče a smazání největšího prvku, tedy $\mathcal{O}(n)+\mathcal{O}(1) = \mathcal{O}(n)$.

2.2.5 Spojování

Sloučení dvou front realizovaných pomocí seřazeného pole je možné například pomocí procedury `merge` popsané v [2, s. 31], která běží v čase $\mathcal{O}(n)$, kde n je součet počtu prvků obou polí.

Paměťová složitost realizace prioritní fronty pomocí řazeného pole je, stejně jako u neřazeného pole, lineární s počtem prvků prioritní fronty.

2.3 Neřazený spojový seznam

Kapitola 2.2 přinesla zrychlení u operace odebrání největšího prvku, ovšem za cenu zpomalení v případě operace vkládání nového prvku. U všech doplňujících operací je pak složitost stejná, nebo dokonce horší, než v případě neřazeného pole.

Některé neduhy neřazeného pole lze vyřešit použitím spojového seznamu. Z důvodu zachování složitosti bude použit oboustranně zřetězený spojový seznam a ukazatele na první a poslední prvek a na prvek s nejvyšší prioritou.

Výhody spojového seznamu se projeví u operace spojování dvou prioritních front. Zatímco u neřazeného pole bylo potřeba provést n vložení, v případě spojového seznamu stačí za poslední prvek první fronty napojit první prvek druhé fronty. Ukazatel na poslední prvek se přepíše hodnotou ukazatele z druhé prioritní fronty a maximový ukazatel bude nastaven na větší z maxim původních front. Všechny tyto operace jsou konstantní, slití tedy proběhne v čase $\Theta(1)$.

Paměťová složitost implementace prioritní fronty pomocí neřazeného spojového seznamu je sice asymptoticky stejná, jako v případě neřazeného pole, reálně potřebuje ale více paměti z důvodu udržování ukazatelů na předchozí a následující prvek.

2.4 Řazený spojový seznam

Jestliže neřazený spojový seznam přinesl zrychlení u operace `merge`, nabízí se vyzkoušet, zda nebude mít podobný efekt i použití řazeného spojového seznamu.

Struktura spojového seznamu bude v tomto případě stejná, jako tomu bylo u neřazeného spojového seznamu. Hodnoty naopak budou uspořádány v intuitivním pořadí od té s největším klíčem, po nejmenší, tedy opačně, než u řazeného pole z kapitoly 2.2.

2.4.1 Vkládání a hledání maxima

Při vkládání nového prvku je potřeba nejdříve ve spojovém seznamu najít jeho správné místo. To lze provést procházením celého seznamu od největšího po

nejmenší prvek dokud je splněna podmínka, že hodnota prvku ve spojovém seznamu je větší, než hodnota vkládaného prvku. Ve chvíli, kdy podmínka přestane platit, nebo algoritmus dojde na konec seznamu, je nový prvek zařazen na patřičné místo. V nejhorším případě dojde k projití celého seznamu, časové složitost je tedy $\mathcal{O}(n)$.

Největší prvek se vždy nachází na začátku spojového seznamu, jeho získání je tedy konstantní operace.

2.4.2 Mazání

Smazání největšího prvku spočívá v pouhém odtržení prvního prvku a změny ukazatele na počátek seznamu, tedy opět konstantní operace.

Mazání obecného prvku spočívá v pouhém odtržení prvku ze spojového seznamu. Pouze je třeba dát pozor na situace, kdy je mazaný prvek na začátku či konci seznamu, v těchto případech je navíc třeba aktualizovat ukazatel na začátek, respektive konec seznamu. Časová složitost tedy zůstává $\Theta(1)$.

2.4.3 Změna klíče a spojování

Změnu klíče lze realizovat odtržení prvku ze spojového seznamu, což lze udělat v konstantním čase, aktualizací jeho hodnoty a jeho opětovným vložením zpět do seznamu. Jelikož má vložení nového prvku lineární složitost, bude mít i změna klíče asymptotickou složitost $\mathcal{O}(n)$.

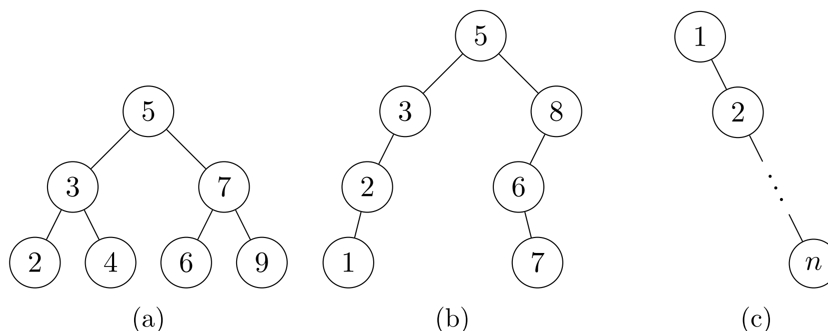
Spojování dvou seřazených probíhá nalezením většího prvku z počátků obou seznamů, jeho odtržení a zařazení do výsledného seznamu. Poté se pokračuje stejným způsobem až do chvíle, než je jeden ze seznamů prázdný. V tu chvíli je aktuálně první prvek neprázdného seznamu zařazen za poslední prvek nově vznikajícího seznamu. Časová složitost algoritmu je $\mathcal{O}(\min(n_1, n_2))$, kde n_1 a n_2 označuje velikosti původních seznamů. Velkou výhodou použití řazených spojových seznamů je, že k jejich spojení je třeba pouze konstantní množství pomocné paměti.

2.5 Binární vyhledávací strom

Binární vyhledávací strom (BVS) je, jak název napovídá, binární strom, kde každý uzel obsahuje ukazatele na pravého a levého syna, ukazatel na rodiče a klíč. Navíc platí podmínka, že v levém podstromě se vždy nachází prvky, jejichž hodnota klíče je menší, než hodnota klíče rodiče. Naopak v pravém podstromě jsou všechny prvky s hodnotou klíče větší.[2]

Díky předchozím vlastnostem lez pomocí BVS poměrně efektivně implementovat i prioritní frontu, jelikož pozice největšího prvku je v každé chvíli známá, nachází se vždy v nejpravějším uzlu.[1]

Všechny základní operace nad binárním stromem jsou svou složitostí závislé na jeho výšce. V případě úplného[1] BVS s n uzly (obr. 2.2a) je hloubka stromu $\mathcal{O}(\log n)$. Pokud je ale binární strom degenerovaný na lineární řetězec n uzlů (obr. 2.2c), může být složitost operací až $\mathcal{O}(n)$.



Obrázek 2.2: Příklady různých binárních vyhledávacích stromů

2.5.1 Vkládání a hledání maxima

Vkládání nového prvku začíná u kořene stromu. S ním je porovnána hodnota klíče nového prvku a na základě výsledku se rekurzivně pokračuje s pravým (je-li klíč nového prvku větší), resp. levým podstromem. Ve chvíli, kdy podstrom neexistuje, je nový prvek uložen jako odpovídající potomek posledního navštíveného uzlu. Složitost takové operace bude v nejhorším případě $\mathcal{O}(n)$.[2]

Pozice největšího prvku, jak bylo již řečeno, je v každou chvíli známa. Je ovšem zbytečné pokaždé procházet celý strom až k nejpravějšímu uzlu, implementace bude tedy ukazatel na nejpravější prvek udržovat zvlášť. Díky tomu bude složitost získání maxima konstantní.

2.5.2 Mazání

Při mazání obecného prvku z BVS je potřeba rozlišovat tři situace. Je-li mazaný prvek list, dojde k jeho prostému odtržení. V případě, že má pouze jeden podstrom, je odstraněn a nahrazen svým synem. Nejkomplikovanější je situace v případě, že má uzel oba dva podstromy. V takovém případě je nutné nalézt následníka[2] mazaného uzlu, ten odstranit a jeho hodnotu přesunout do mazaného uzlu. Hledání následníka má v nejhorším případě složitost $\mathcal{O}(n)$, tedy i složitost mazání libovolného prvku z BVS bude $\mathcal{O}(n)$.[12]

[1] Úplný binární strom je takový binární strom, kde pro každý uzel v platí, že hloubka levého a pravého podstromu se liší maximálně o jedna[12].

[2] Následníkem nějakého uzlu v je nejlevější uzel jeho pravého podstromu.

V případě odstraňování největšího prvku z binárního stromu je situace o něco jednodušší. Nejdříve je potřeba si uvědomit, že pro největší prvek nikdy nenastane situace, kdy by obsahoval oba podstromy.

Věta 4. *Uzel v jehož hodnota klíče $k(v)$ je v rámci stromu T největší je nejpravějším uzlem, tedy nemá pravý podstrom.*

Důkaz. Nechť existuje uzel v takový, že hodnota jeho klíče $k(v)$ je v rámci stromu T největší a zároveň existuje jeho pravý podstrom $P(v)$. Potom ale pro libovolný uzel $u \in P(v)$ nutně platí $k(u) > k(v)$. Uzel v tedy nemůže být v rámci stromu T největší, což je spor s předpokladem. Uzel v tedy nemůže mít pravý podstrom. □

Jak vyplývá z předchozí věty, při mazání největšího prvku řešíme vždy variantu bez podstromů, respektive s jedním, konkrétně levým, podstromem. Na první pohled by se mohlo zdát, že takové mazání lze provádět v konstantním čase. To je pravda jen částečně, totiž pouze v případě, že byl mazaný prvek listem. Tehdy opravdu není potřeba hledat následníka, novým maximem se stává rodič mazaného prvku a dochází k pouhému přepsání několika ukazatelů. Ovšem obsahuje-li mazané maximum levý podstrom, není novým maximem jeho kořen, ale předchůdce[3] mazaného prvku. Jeho hledání může v nejhorším případě opět vyžadovat prohledání všech prvků, složitost mazání maxima je tedy $\mathcal{O}(n)$.

2.5.3 Změna klíče a spojování

Změna hodnoty klíče libovolného prvku sestává ze dvou kroků. Nejprve dojde k jeho smazání, následně pak k jeho opětovnému vložení s novou hodnotou. Jak mazání, tak vkládání, mají, jak bylo ukázáno výše, lineární složitost, proto bude i asymptotická složitost změny hodnoty klíče lineární.

Poslední operací, kterou by měla implementace prioritní fronty podporovat, je spojení dvou prioritních front do jedné. Při této operaci jsou nejdříve oba BVS *in-order* projity a jejich prvky jsou nakopírovány do pomocných polí A_1, resp. A_2. Tyto jsou pak slity do jednoho pole A pomocí algoritmu popsaného v kapitole 2.2. Konstrukce binárního vyhledávacího stromu ze seřazeného pole pak zabere lineární čas. Kořen výsledného stromu bude prostřední prvek pole A. Jeho pravý a levý podstrom jsou pak rekurzivně vytvořeny z prvků napravo, resp. nalevo od nalezeného kořene.

Není bez zajímavosti, že pomocí binárního vyhledávacího stromu je možné poměrně úsporně implementovat podobnou ADT, totiž *double-ended* prioritní frontu[15][16]. Ta je velice podobná popisované prioritní frontě, umožňuje ovšem efektivně pracovat jak s maximem, tak s minimem zároveň.

[3] Aneb nejpravější prvek levého podstromu.

2.6 Červeno-černý strom

Jak bylo ukázáno v kapitole 2.5, složitost základních operací binárního vyhledávacího stromu závisí na jeho výšce, která může být v nejhorším případě i lineární s počtem prvků. Červeno-černé stromy, poprvé představené v [17] jako symetrické B-stromy, jsou jedna z mnoha variant[4] vyvažovaných binárních vyhledávacích stromů, což znamená, že udržují hloubku stromu rovnou $\mathcal{O}(\log n)$.[2]

2.6.1 Struktura

Červeno-černý strom je binární vyhledávací strom, který u každého uzlu udržuje jeden bit informace navíc, totiž jeho barvu[19]. Pomocí omezení barev, které se mohou vyskytovat na každé jedné cestě z kořene do listu, je v červeno-černých stromech zajištěno, že každá cesta bude maximálně dvakrát delší, než libovolná z ostatních cest.[2]

Každý uzel bude tedy obsahovat pět atributů, totiž barvu, klíč, ukazatele na levého a pravého syna, ukazatel na rodiče a odpovídající data. Pokud některý ze synů uzlu neexistuje, bude mít ukazatel hodnotu `NIL`. Tyto hodnoty jsou ovšem považovány také za uzly binárního vyhledávacího stromu a platí pro ně tedy i všechny podmínky z definice 2.[2]

Definice 2 (Červeno-černý strom)**.** *Červeno-černý strom je binární strom, který splňuje následující podmínky:*

1. *Každý uzel je buď červený nebo černý.*
2. *Kořen stromu je vždy černý.*
3. *Každý list*[5] *je černý.*
4. *Pokud je uzel červený, pak oba jeho synové musí být černí.*
5. *Pro každý uzel platí, že počet černých uzlů na všech cestách do listů jeho podstromů je stejný.*

2.6.2 Hloubka červeno-černého stromu

Jak bylo uvedeno hned na začátku této kapitoly, nejdůležitější vlastností, kterou u binárních vyhledávacích stromů sledujeme, je jeho výška. Proto bude, ještě před analýzou fungování jednotlivých operací, uveden důkaz, že jeho hloubka je asymptoticky opravdu logaritmická.

[4] Mezi další zástupce vyvažovaných BVS patří například AVL stromy[12, s. 183], B-stromy[2, s. 505], 2-3 stromy[5, s. 424] či splay stromy[18].

[5] Listy červeno-černého stromu tvoří vždy `NIL` hodnoty.

V důkazu využijeme další vlastnost uzlů červeno-černých stromů, totiž tzv. *černou hloubku*, značíme $\mathrm{bh}(x)$. Ta přímo vyplývá z bodu 5 definice 2, jedná se o počet černých uzlů na libovolné cestě z uzlu do kořene. Černá hloubka celého stromu je pak rovna černé hloubce jeho kořene.

Nyní již k samotnému důkazu logaritmické hloubky.

Věta 5 (O hloubce červeno-černých stromů)**.** *Červeno-černý strom s n vnitřními uzly má hloubku maximálně $2\log(n+1)$.*

Důkaz. Nejdříve budiž ukázáno, že libovolný podstrom s kořenem x obsahuje alespoň $2^{\mathrm{bh}(x)} - 1$ vnitřních uzlů.

Pokud je hloubka x rovna nule, tak platí, že x je list. O stromu s takovým kořenem může být bezpochyby řečeno, že obsahuje $2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 0$ vnitřních uzlů. Nyní nechť x má kladnou hloubku a jedná se o vnitřní uzel nějakého stromu s oběma syny. Každý ze synů má hloubku buď $\mathrm{bh}(x)$ nebo $\mathrm{bh}(x)-1$, podle toho, jestli je jejich barva červená nebo černá. Jelikož hloubka libovolného syna x je určitě alespoň o jedna menší, než hloubka x, indukční předpoklad říká, že každý ze synů má minimálně $2^{\mathrm{bh}(x)-1} - 1$ vnitřních uzlů. Strom s kořenem x pak bude mít $(2^{\mathrm{bh}(x)-1} - 1) + (2^{\mathrm{bh}(x)-1} - 1) + 2 = 2^{\mathrm{bh}(x)} - 1$ vnitřních uzlů, čímž je dokázán předpoklad o vnitřních uzlech.

Nyní nechť h je hloubka červeno-černého stromu. Podle vlastnosti 4 z definice 2 musí být alespoň polovina uzlů na libovolné cestě z kořene do listu, nepočítaje kořen, černá. Tedy černá hloubka kořene takového stromu musí být minimálně $h/2$ a tím pádem platí nerovnost $n \geq 2^{h/2} - 1$, jejíž úprava pak vede na požadovaný důkaz hloubky:

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

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

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

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

.[2]

□

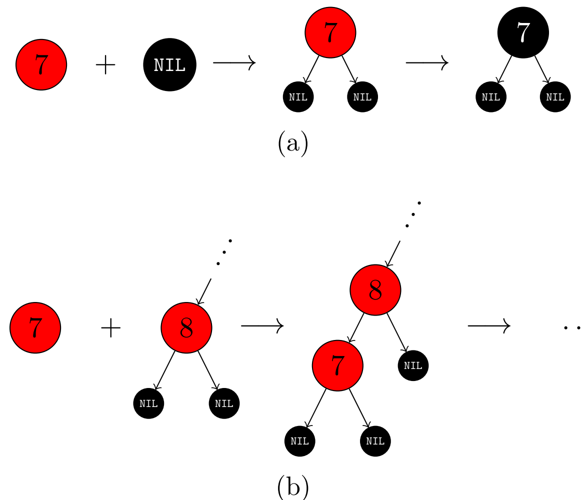
Se znalostí maximální hloubky červeno-černého stromu mohou být konečně analyzovány všechny podporované operace.

2.6.3 Vkládání

Je zřejmé, že i když mají červeno-černé stromy ve své podstatě podobnou strukturu, jako obecný BVS, modifikující operace se budou nutně lišit. Když by tomu tak nebylo, došlo by k porušení některé z vlastností z definice 2 a tím pádem by přestala platit i záruka maximální hloubky z věty 5.

Jak tedy provést vložení nového prvku z do červeno-černého stromu? Nejdříve je nový prvek obarven červenou barvou[6] a následně vložen do stromu stejně, jako v případě běžného binárního stromu. Po provedení předchozích kroků je třeba ověřit, že nebyla porušena některá z podmínek z definice 2.

Je nasnadě, že podmínka 1 nemůže být vložením nového prvku nikdy porušena. Stejně tak podmínka 3 bude stále splněna, jelikož nově vložený uzel implicitně obsahuje dva černé `NIL` syny, které jsou nutně listy. Poslední podmínkou, která zůstává neporušena, je 5. Přidáním nového červeného uzlu se totiž nikdy nenavýší počet černých uzlů na libovolné cestě od kořene k listům. Vkládaný uzel sice nahrazuje jeden černý `NIL` uzel, nicméně, jak již bylo řečeno, obsahuje dva černé syny a sám k černé hloubce nic nepřidává. Problém budou tedy způsobovat pouze podmínky 2, tedy že kořen musí být černý, a 4, která říká, že červený uzel musí mít oba syny černé.



Obrázek 2.3: Příklady vkládání do červeno-černého stromu, při kterých je porušena některá z jeho podmínek

Jednodušší situace nastává, pokud se vložený uzel z stane kořenem stromu. V takovém případě dojde k jeho pouhému přebarvení na černo (obr. 2.3a).

Vložením červeného uzlu pod černý uzel dokonce nedojde, jak vyplývá z předchozích odstavců, k porušení žádné podmínky.

V případě, kdy je nový uzel z zařazen jako jeden ze synů nějakého červeného uzlu y (obr. 2.3b), bude situace o něco komplikovanější.

Nejdříve je ale třeba definovat další pojem, který bude hned vzápětí využit. Uzel u bude zván *strýc* uzlu z, pokud platí, že se jedná o sourozence rodiče uzlu z.

[6] Černá barva není použita proto, že by vždy došlo k porušení podmínky 5.

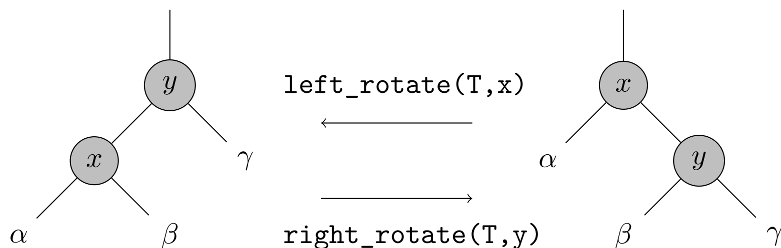
Nechť u označuje strýce nově vkládaného uzlu z a rodič z je levým synem svého rodiče. Z červeno-černé podmínky 1 vyplývá, že u může mít buď červenou, nebo černou barvu. Podle barvy strýce u jsou rozlišovány následující situace:

1. Je-li u červený, dojde k přebarvení rodiče z a u na černou barvu, zatímco předek u je přebarven na červenou. V tuto chvíli může být některá z červeno-černých podmínek stále porušena, proto se s opravováním pokračuje s předkem strýce u.

2. Je-li u černý a zároveň z je pravý syn, bude provedena levá rotace v předkovi uzlu z (viz obrázek 2.4), ukazatel z bude nově ukazovat na předka z, ze kterého se nově stane levý syn. Pokračuje se s bodem 3.

3. Je-li u černý a zároveň z je levý syn, bude třeba provést pravou rotaci v prapředkovi z a zároveň změnit jeho barvu na červenou. Krom toho bude přebarven i rodič z, totiž na černou.

V případě, že by byl rodič uzlu z pravým synem svého rodiče, budou nastávat stejné situace, jako v případě, že je levým synem. Pouze se při řešení těchto situací bude vždy namísto levého syna pracovat s pravým a obráceně. Také rotace budou probíhat na opačnou stranu.



Obrázek 2.4: Ukázka fungování rotací uzlů binárního stromu. Symboly α, β, γ představují podstromy jednotlivých uzlů.[2]

Složitost předchozího algoritmu je logaritmická s počtem uzlů stromu. Nastane-li situace 2 nebo 3, budou provedeny dvě rotace a dál se nikam nepokračuje. Rotace jsou navíc konstantní operace, jelikož dochází pouze k přepojení několika ukazatelů. V případě situace 1 se po provedení přebarvení pokračuje na hladině o 2 nižší, než je hladina aktuálního uzlu. Jak bylo dokázáno dříve, hloubka stromu je logaritmická, tedy i počet přebarvení bude nejhůře logaritmický.[2]

2.6.4 Mazání

Další modifikující operací nad červeno-černými stromy je mazání nějakého prvku. Nejdříve bude ukázáno mazání obecného prvku, následně bude věnován prostor i speciálnímu případu, kterým je mazání maxima.

Ve své nejzákladnější struktuře se mazání z červeno-černého stromu neliší od mazání z obecného binárního vyhledávacího stromu. Ovšem při každé možnosti počtu synů mazaného uzlu může být porušena některá z vlastností z definice 2, proto bude potřeba strukturu stromu pomocí rotací a přebarvování uzlů opravovat.

V případě, že mazaný uzel z nemá žádného potomka[7], pak je odtržen. Byl-li uzel červený, nedojde k porušení žádné vlastnosti. Jediným potenciálním problémem by mohla být změna černé hloubky podstromu obsahujícího mazaný uzel z. Ta se ale odstraněním červeného uzlu nikdy nemění. Přesně opačná situace nastává, pokud je mazaný uzel černý. V tom případě dojde ke změně černé hloubky a je tím pádem porušena vlastnost 5 z definice 2. Je tedy potřeba provést opravu. Ta bude zavolána na `NIL` list, který nahradí mazaný uzel z.

Má-li pouze jednoho ze synů, je uzel odtržen a nahrazen svým jediným synem s. Jestliže barva z byla černá, potom je opět třeba strom opravit. Algoritmus na opravu bude volán na uzel s, který nahradil původní z.

Nejsložitější situací zůstává ten případ, kdy má uzel oba syny. V takovém případě je nutné najít následníka y, vyměnit ho se z a provést odtržení z, který v tu chvíli má nutně pouze jednoho nebo žádné potomky[8]. Pokud byla původní barva následníka s černá, poté opět mohlo dojít k porušení některé z podmínek a je potřeba provést opravu na jeho potomkovi, který může být buď `NIL` list, nebo kořen jeho pravého podstromu.[2]

Nyní již k několikrát zmiňované opravě vlastností stromu. Nechť x označuje uzel, ve kterém došlo k porušení vlastností stromu, w je sourozenec x a p je jejich společný rodič. Jestliže je x kořenem stromu nebo je jeho barva červená, je jeho barva nastavena na černou a algoritmus končí. V opačném případě, tedy když x není kořenem a jeho barva je černá, je prováděno následující:[2]

1. Je-li barva w červená, je jeho barva změněna na černou, p je přebarven na červenou, dojde k provedení levé rotace popsané v kapitole 2.6.3 na uzlu p. Nový sourozenec uzlu x je díky rotaci jeden ze synů původního sourozence w, který je navíc určitě černý. V tuto chvíli se případ 1 mění na jeden z případů 2,3 nebo 4.

2. Je-li barva w černá a navíc jsou oba jeho synové také černí, dojde k přebarvení w na červeno a dále se pokračuje s $x <= p$.

[7] Nepočítaje `NIL` listy.

[8] Budiž připomenuta definice následníka. Jedná se o nejlevější uzel v pravém podstromu, tedy bude vždy obsahovat pouze pravý, nebo dokonce žádný podstrom.

3. Je-li barva w černá, levý syn w je červený a pravý syn w je černý, dojde k přehození barvy uzlu w a jeho levého syna, následně je provedena pravá rotace v uzlu w. Novým sourozencem uzlu x se v tuto chvíli stává černý uzel s červeným pravým synem, čímž se z případu 3 stává případ 4.

4. Je-li barva w černá a pravý syn w je červený, potom je barva w nastavena na barvu p, barva p je, stejně jako barva pravého syna w, nastavena na černou a dojde k levé rotaci v uzlu p. Následně je algoritmus ukončen.

Předchozí operace se opakují, dokud není porušena podmínka, že uzel x není kořen a jeho barva je černá. Je dobré si všimnout, že jediný případ, kdy dojde k opakování je, pokud nastane případ 2.[2]

Stejně jako jiné operace, má i mazání obecného prvku z n prvkového červeno-černého stromu, díky garanci jeho hloubky, logaritmickou složitost.[2]

Při mazání maxima x existuje stejná garance, kterou dává věta 4, tedy že mazané maximum nebude mít pravý podstrom. Na základě této znalosti může být řečeno, že problém nastává pouze ve chvíli, kdy je barva maxima černá. V opačném případě se totiž černá hloubka žádného podstromu obsahujícího x nemění a stejně tak žádná další z vlastností červeno-černých stromů nemůže být porušena. Vyřešení problému, který nastává při smazání černého uzlu, bude řešen stejně, jako v případě mazání obecného prvku a proto je i jeho složitost stejná.

2.6.5 Hledání maxima, spojování a změna klíče

Operace hledán maxima, spojování dvou červeno-černých stromů i změna klíče nějakého prvku budou probíhat identicky, jako tomu bylo v případě binárních vyhledávacích stromů z kapitoly 2.5.

2.7 Binární halda

Binární halda byla představena v [9] jako datová struktura pro algoritmus heapsort, který byl podrobněji rozebrán již v kapitole 1.3.3. Článek se ovšem zabývá spíše praktickou aplikací, než teoretickým popisem, s tím přišli až další.

Na binární haldu lze hledět jako na úplný binární strom, tedy platí, že všechny jeho vrstvy jsou plně zaplněny, krom poslední, která nemusí být kompletní, ale je zaplňována zleva. Navíc tento binární strom splňuje tzv. haldovou podmínku, která určuje, že hodnota klíče každého uzlu je maximálně taková, jako je hodnota klíče jeho rodiče[9]. Výjimku tvoří kořen stromu, který žádného rodiče nemá, nicméně z předchozího vyplývá, že ten bude vždy nutně největším prvkem celého stromu.[2]

[9]Budiž opět připomenuto, že v této práci je uvažována pouze maximová prioritní fronta, a tudíž i haldová podmínka je uvedena pro maximovou binární haldu.

Jak bylo řečeno, binární halda je úplný binární strom, a jako takovou ji lze jednoduše implementovat pomocí pole. Kořen binární haldy, což je, jak plyne z haldové podmínky, zároveň i uzel s největším klíčem, je při této implementaci vždy na indexu 1. Díky úplnosti takového stromu lze jednoduše zjistit i pozice synů libovolného uzlu. Pro libovolný uzel uložený na nějakém indexu i, kde $i \geq 1$, najdeme levého syna, pokud tedy existuje, na pozici $2i$ a pravého syna, pokud tedy existuje, na $2i + 1$. I pozice rodiče je snadno zjistitelná, konkrétně se jedná o $\lfloor i/2 \rfloor$.[2]

2.7.1 Vkládání a hledání maxima

Vkládání nového prvku do prioritní fronty sestává z vytvoření nového listu l a jeho připojením na první volnou pozici poslední vrstvy binárního stromu reprezentujícího haldu. V této chvíli může ovšem dojít k porušení haldové podmínky. Je tedy nutné ověřit, že hodnota klíče nového prvku je menší, než hodnota klíče rodiče. Pokud by nastala situace, kdy by byla haldová podmínka porušena, dojde k prohození listu l a jeho rodiče. Toto prohazování nově vloženého prvku s jeho rodičem bude probíhat dokud je porušena haldová podmínka. V nejhorším případě je klíč l větší, než klíč kořenu celé haldy, a dojde tedy k prohození na každé úrovni binárního stromu. Už dříve bylo uvedeno, že hloubka úplného binárního stromu je $\mathcal{O}(\log n)$, tedy i složitost vkládání nového prvku bude v nejhorším případě $\mathcal{O}(\log n)$.[1]

Zjištění maxima je konstantní operace, jelikož to se vždy, pro haldu uloženou v poli A, nachází na pozici $A[1]$.

2.7.2 Mazání maxima

Další algoritmicky zajímavou operací je mazání maxima z binární haldy. To probíhá prohozením kořene r, ve kterém je maximum uloženo, s posledním vloženým listem l. Následně je uzel r ze stromu odtržen. V tuto chvíli může opět nastat situace, kdy je porušena haldová podmínka, totiž uzel l, který se díky prohození s r stal novým kořenem, nemusí být větší, než kořeny jeho podstromů. Je tedy potřeba, podobně jako při vkládání nového prvku, zařídit splnění haldové podmínky. V tomto případě nebude docházet k prohazování s rodičem, nýbrž s kořenem toho podstromu uzlu l, jehož hodnota klíče je větší. To bude opět probíhat až do chvíle, kdy je klíč l větší, než klíče levého i pravého syna. V nejhorším případě, například, když je l nejmenším prvek celé fronty, a ze stejných důvodů, jaké byly uvedeny při analýze vkládání nového prvku, může dojít až $\mathcal{O}(\log n)$ prohození.[1]

2.7.3 Změna klíče

Změna klíče je velice podobná vkládání nového prvku. Po navýšení hodnoty je opět potřeba ověřit, že mezi měněným prvkem x a jeho rodičem je dodržena

haldová podmínka. Pokud by tomu tak nebylo, dojde ke stejnému prohození, jako u zmíněného vkládání. Složitost změny klíče bude v nejhorším případě také stejná, tedy $\mathcal{O}(\log n)$.

2.7.4 Mazání obecného prvku

Další operací, která má být prioritní frontou podporována, je mazání obecného prvku. To lze provést poměrně přímočaře využitím již dříve popsaných operací. Nejdříve je změněna hodnota mazaného klíče na $+\infty$. Jelikož se takový prvek nutně dostane do kořene, stačí tento smazat pomocí operace `dequeue`. Asymptotická složitost změny klíče i mazání maxima z binární haldy je $\mathcal{O}(\log n)$. Celková složitost mazání obecného prvku bude tedy $\mathcal{O}(\log n) + \mathcal{O}(\log n)$, což se rovná $\mathcal{O}(\log n)$.

2.7.5 Spojování

Spojování dvou binárních hald je nejčastěji realizováno prostým zřetězením polí realizujících jednotlivé haldy a následným vybudováním nové haldy nad nově vzniklým polem. Jak je uvedeno v [20] a dokázáno v [21], to lze provést v čase $\Theta(n_A)$, kde n_A je součet velikostí obou hald. Později byly v literatuře popsány i rychlejší varianty spojování binárních hald, viz například [22] nebo [23].

Jelikož je binární halda udržována v poli, není potřeba ukládat ukazatele na podstromy a rodiče tak, jak tomu bylo například u binárního vyhledávacího stromu. Díky tomu je binární halda paměťově celkem úspornou datovou strukturou.

2.8 Binomiální halda

Vlastnostmi velmi podobnou datovou strukturou, jako je v kapitole 2.7 představená binární halda, je i binomiální halda. Její hlavní výhodou je, jak bude později dokázáno, podpora spojování dvou hald v logaritmickém čase. Zároveň je pro její udržování potřeba méně paměti, než u většiny vyvažovaných binárních vyhledávacích stromů[24].

2.8.1 Binomiální strom

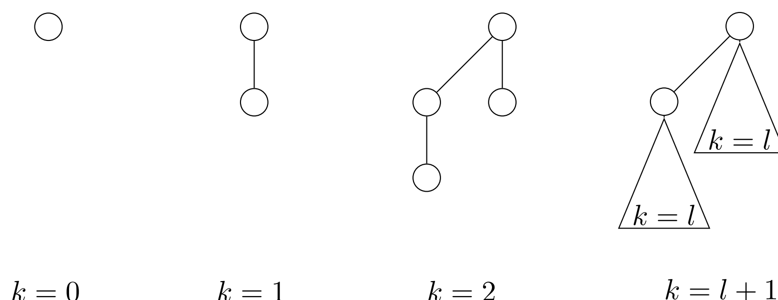
Před samotným popisem a analýzou operací binomiální haldy je potřeba definovat podpůrné struktury, které pro svou práci binomiální halda používá.[24]

Definice 3 (Binomiální strom)**.** *Zakořeněný strom T je binomiální stromem řádu k, pokud splňuje následující:*

1. *Pokud je řád k roven nule, obsahuje T pouze kořen.*

2. *Pokud je řád k různý od nuly, pak má T kořen s právě k synů, které jsou po řadě binomiálními stromy řádů $k-1,\ldots,0$.[12]*

Každý uzel binomiálního stromu se bude skládat z ukazatele na nejlevějšího ze svých dětí, ukazatele na svého pravého sourozence, ukazatele na rodiče, svého stupně a pochopitelně hodnoty klíče. Pokud některý z cílů ukazatelů neexistuje, bude ukazatel směřovat na hodnotu `NIL`.[2]



$k=0$ $k=1$ $k=2$ $k=l+1$

Obrázek 2.5: Binomiální stromy různých řádů

Jak mohou takové binomiální stromy vypadat ukazuje obrázek 2.5.

2.8.2 Vlastnosti binomiální haldy

Pomocí binomiálních stromů je pak již možné definovat samotnou binomiální haldu.

Definice 4 (Binomiální halda)**.** *Binomiální halda se skládá ze souboru binomiálních stromů $\mathcal{T}=T_1,\ldots,T_l$, kde:*

1. *Pro každý strom $T_i\in\mathcal{T}$ platí haldová podmínka.*
2. *V $\mathcal{T}$ se žádný řád stromu nevyskytuje více než jednou.*
3. *Binomiální stromy z $\mathcal{T}$ jsou v souboru uspořádány vzestupně podle řádu.[12]*

Pro analýzu podporovaných operací je třeba znát počet stromů, z kterých se binomiální halda o n prvcích skládá. Nejdříve úkrok stranou k počtu prvků v jednom binomiálním stromu, který bude využit při důkazu počtu stromů.

Věta 6 (O počtu prvků v binomiálním stromu)**.** *Binomiální strom řádu k obsahuje právě 2^k prvků.*

Důkaz. Důkaz bude proveden indukcí podle řádu stromu. Pro strom řádu $k=0$ platí triviálně $2^k=2^0=1$. U stromu řádu $k=n+1$ budeme vycházet z definice 3, konkrétně bodu 2. Ten uvádí, že každý binomiální strom řádu k

je tvořen kořenem s k podstromy, jejichž řád je popořadě $k-1,\dots,0$. Z indukčního předpokladu plyne, že všechny podstromy kořene obsahují přesně 2^i prvků, kde $i \in \{0,\dots,k-1\}$. Strom řádu k bude tedy obsahovat přesně

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

prvků, čímž byla věta dokázána. □

Se znalostí počtu prvků může být vysloveno, a následně dokázáno, tvrzení o počtu stromů v binomiálním stromu.

Věta 7 (O počtu stromů binomiální haldy)**.** *Binomiální halda o n prvcích se skládá z nejvýše $\lfloor \log_2(n) \rfloor + 1$ binomiálních stromů.*

Důkaz. Vlastnost přímo vyplývá z věty 6 a definice 4. Má-li každý binomiální strom 2^k prvků a je svým řádem v rámci haldy unikátní, bude se ve výsledné haldě vyskytovat právě tehdy, když v binárním zápisu počtu prvků celé haldy je k-tý nejmenší bit rovný 1. Jelikož počet bitů potřebných k binární reprezentaci libovolného kladného decimálního čísla je roven $\lfloor \log_2(n) \rfloor + 1$, je i počet stromů v libovolné binomiální haldě roven $\lfloor \log_2(n) \rfloor + 1$.[12] □

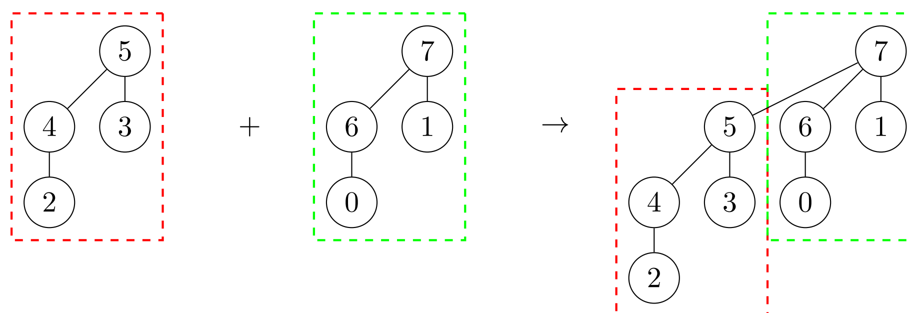
Předchozí vlastnosti budou využity v důkazech asymptotických složitostí jednotlivých operací.

2.8.3 Spojování

Nejdůležitější operací binomiální haldy, která je využívána i ve většině ostatních operací, je sloučení dvou binomiálních hald, budiž proto uvedena a rozebrána jako první.

Prvním problémem, který je potřeba vyřešit, je slévání dvou binomiálních stromů stejného řádu k. Při něm je potřeba napojit kořen jednoho stromu jako prvního syna kořene druhého stromu. Kvůli zachování podmínky 1 z definice 4 musí nutně dojít k připojení kořene s menším prvkem pod kořen s větším prvkem. Po provedení slití dostaneme binomiální strom řádu $k+1$.[12] Slití dvou binomiálních stromů je konstantní operace a ilustruje ho obrázek 2.6.

Nyní již k samotnému slévání dvou binomiálních hald. Algoritmus probíhá postupným procházením obou hald od nejnižšího k nejvyššímu řádu. Pokud ani jedna fronta neobsahuje strom daného řádu, ve výsledné haldě se také žádný strom patřičného řádu neobjeví. Objevuje-li se strom nějakého řádu k pouze v jedné z původních hald, dojde k jeho překopírování do výsledného stromu. Poslední možnost je, že obě haldy strom daného řádu obsahují. V takovém případě dojde k jejich sloučení připojením kořene s menší hodnotou pod kořen s větší hodnotou, čímž vznikne strom řádu $k+1$. Ten je potom uvažován v další iteraci.[24] Příklad jednoduchého slévání dvou binomiálních hald je na obrázku 2.7.



Obrázek 2.6: Příklad slévání dvou binomiálních stromů řádu $k = 2$

Počet kroků algoritmu slučování závisí na počtu prvků v jednotlivých haldách. Bylo dokázáno, že tento je nejhůře logaritmický s počtem prvků haldy. Tedy algoritmus slučování bude mít nejhůře $(\lfloor \log_2(\max(n_1, n_2)) \rfloor + 1) + 1$ iterací. Jeho složitost je tedy také logaritmická.

2.8.4 Vkládání a hledání maxima

Vložení nového prvku do binomiální haldy B probíhá ve dvou krocích. Nejdříve je nově vkládaný prvek označen za binomiální haldu tvořenou jedním binomiálním stromem řádu 0. Tato halda je následně sloučena s původní haldou B. Asymptotická složitost přidávání nového prvku je rovna složitosti slévání dvou hald, tedy $\mathcal{O}(\log n)$.

Konstantní operací zůstává, stejně jako u binární haldy, získání maxima. Při každé operaci prováděné nad binomiální haldou je udržován ukazatel na největší kořen, tedy stačí tento vrátit.

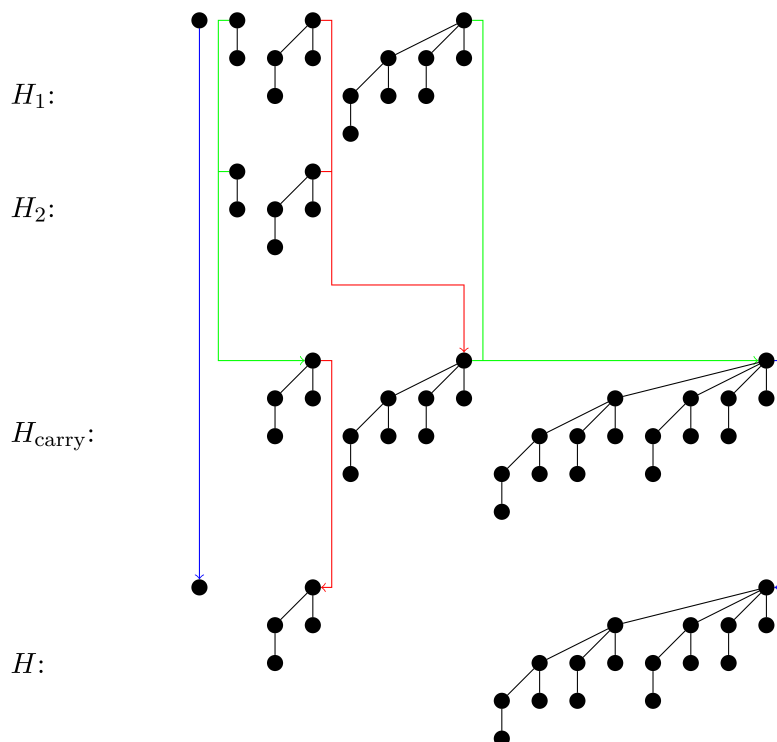
2.8.5 Mazání maxima

Při mazání maxima bude opět využit dříve popsaný algoritmus na slévání dvou binomiálních hald. Nejdříve dojde k odtržení stromu T, jehož kořen je v rámci haldy H maximální. Následně je ze všech podstromů kořene T vytvořena nová binomiální halda, která je nakonec slita s původní haldou H. Vytvoření haldy z podstromů kořene T zabere nejvýše tolik času, kolik jich existuje, tedy $\mathcal{O}(\log n)$. Slévání hald má pak logaritmickou složitost, tedy celková složitost odstranění největšího prvku bude $\mathcal{O}(\log n)$.[12]

2.8.6 Změna klíče

Operace změny klíče a odstranění libovolného prvku jsou pak velice podobné, jako u binární haldy z kapitoly 2.7.

V případě změny klíče může dojít k porušení haldové podmínky, je tedy potřeba tuto opravit probubláním prvku na správnou pozici. V nejhorším případě bude nová hodnota klíče větší, než hodnota klíče kořene stromu, ve



Obrázek 2.7: Haldy H_1 a H_2 jsou slévány do nové haldy H. Algoritmus začíná se stromy řádu 0. Ten existuje pouze u haldy H_1, je tedy přímo překopírován do výsledné haldy H. V druhém kroku se pracuje se stromy řádu 1. Ty existují u obou hald, jsou tedy spojeny a vzniká přenos řádu 2. Ve výsledné haldě se žádný strom řádu 1 neobjevuje. Ve třetí iteraci se na scéně objevují stromy řádu 2, které jsou opět přítomné u obou hald, navíc ale existuje přenos z předchozího kroku. Ten je uložen do výsledné haldy H, stromy z hald H_1 a H_2 jsou opět sloučeny a vzniká další přenos, tentokrát řádu 3. S tím se algoritmus v dalším kroku vypořádá stejně, jako tomu bylo u stromů řádu 1. V posledním kroku zbývá akorát přenos z minulé iterace, který je zkopírován do výsledné haldy.

kterém se prvek nachází, dojde tedy k jeho přesunutí do kořene. Hloubka každého binomiálního stromu je logaritmická s počtem jeho prvků, tedy i změna klíče bude logaritmická operace.

2.8.7 Mazání obecného prvku

Mazání obecného prvku x pak sestává z nastavení jeho klíče na $+\infty$. Díky tomu se x stane maximem celé haldy, které je následně odstraněno procedurou `dequeue`. Obě operace mají logaritmickou složitost, tedy i mazání bude mít složitost $\mathcal{O}(\log n)$.

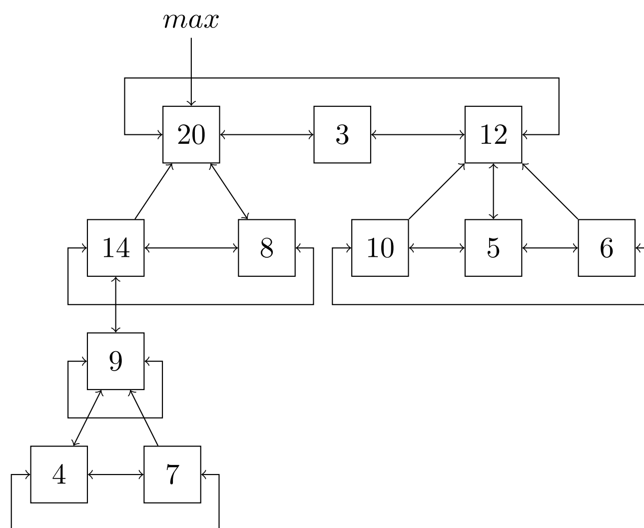
2.9 Fibonacciho halda

Fibonacciho halda je datová struktura podobná binomiální haldě, jež umožňuje provádět rychlé spojování dvou hald a zároveň velká část jí podporovaných operací běží amortizovaně[10] v konstantním čase.[2]

Datová struktura byla poprvé publikována v roce 1987 jako rychlá implementace prioritní fronty pro Dijkstrův algoritmus, který je blíže popsán v kapitole 1.3.1. Jedná se také o historicky první prioritní frontu, která se, i když jen amortizovaně, blíží teoretickým limitům rychlosti jednotlivých operací.[4]

Fibonacciho halda se skládá z kolekce haldově uspořádaných stromů. Každý uzel stromu se pak skládá z ukazatele na rodiče a ukazatele na jednoho ze svých synů. Pokud syna nebo rodiče nemá, mají ukazatele speciální hodnotu `NIL`. Potomci každého uzlu jsou pak propojeni pomocí obousměrného cyklického spojového seznamu. Dále každý uzel obsahuje informaci o počtu synů, která se nazývá *rank*, a bit indikující, zda je uzel tzv. označený (*marked*). Kořeny jednotlivých stromů tvořících Fibonacciho haldu jsou propojeny stejně, jako sourozenci ve stromech. Navíc je v každou chvíli udržován ukazatel na strom s největším kořenem.[4] Struktura Fibonacciho haldy je ukázána na obrázku 2.8.

Během všech prováděných operací je navíc potřeba udržovat kořeny stromů tvořící haldu neoznačené. Tedy zejména při vkládání nového prvku, odstraňování maxima případně změně klíče.[12]



Obrázek 2.8: Struktura Fibonacciho haldy. Každý prvek obsahuje ukazatel na svého rodiče, pravého a levého sourozence a jednoho ze svých potomků. Pro zjednodušení jsou vynechány hodnoty *rank* a *mark*.[4]

[10] Amortizovaná složitost vyjadřuje průměrnou časovou složitost jedné operace při posloupnosti jejích opakování.[13]

2.9.1 Hledání maxima, spojování a vkládání

Pro získání maxima stačí vrátit prvek, na který ukazuje maximový ukazatel.[4]

Spojení dvou hald H_1 a H_2 do nové haldy H spočívá v propojení spojového seznamu kořenů jejich stromů a nastavení maximového ukazatele na větší z původních hald H_1 a H_2.[2]

Pro vložení nového prvku x do haldy H je třeba vytvořit jednoprvkovou haldu H', která je následně spojena s původní haldou H a v případě, že je nový prvek větší, než největší prvek v H, je přepojen maximový ukazatel.[4]

Všechny předchozí operace mají díky použití obousměrně zřetězeného spojového seznamu pro udržování seznamu stromů asymptotickou složitost $\mathcal{O}(1)$.[4]

2.9.2 Mazání

Časově nejnáročnější operací, kterou lze nad Fibonacciho haldou provádět, je mazání maxima[4]. Je to také operace, kdy jsou stromy tvořící haldu odloženě konsolidovány[2].

Před samotným představením mazání maxima z haldy je třeba ukázat fungování algoritmu konsolidace stromů haldy, který je při operaci `dequeue` používán. Ten spočívá v opakovaném hledání dvou stromů, jejichž kořen má stejný *rank*, a jejich spojení do nového stromu T_n při zachování haldového uspořádání. Strom T_n bude mít *rank* o jedna větší, než původní stromy. Ve chvíli, kdy neexistují žádné dva stromy mající stejný *rank*, je ze stromů vytvořen nový seznam kořenů a dojde k nastavení maximového ukazatele.

Při mazání maxima x z haldy H dojde nejdříve k jeho odtržení. Odstranění prvku z obousměrně zřetězeného spojového seznamu je konstantní operace, stačí přepojit několik ukazatelů. Následně dojde k zapojení všech synů x do seznamu kořenů haldy H a je proveden algoritmus konsolidace.

Mazání obecného prvku opět spoléhá na změnu klíče prvku na nějakou hodnotu $+\infty$ a následné smazání maxima z haldy.

Časová složitost mazání maxima i mazání obecného prvku je amortizovaně $\mathcal{O}^*(\log n)$.[2]

2.9.3 Změna klíče

Zásadně rozdílnou operací oproti binomiální haldě je v případě Fibonacciho hald změna klíče prvku x. Již v předchozích kapitolách věnujících se haldám a zmiňujících haldové uspořádání stromu bylo ukázáno, že změna klíče může toto uspořádání porušit. Řešením většinou bylo probublání daného prvku na správnou pozici ve stromě. V tomto případě ovšem dojde k odtržení podstromu zakořeněném v měněném prvku a jeho vložení do seznamu stromů tvořících haldu. Navíc dojde k nastavení označení rodiče y uzlu x. Byl-li y již dříve označený, je odtržen stejným způsobem jako x a opět se ověřuje, není-li jeho rodič také označen. Jak vidno, celý proces může vyústit v sérii odtrhávání, která se zastaví v některém neoznačeném uzlu na cestě do kořene stromu.[12]

Změna klíče má amortizovanou časovou složitost $\mathcal{O}^*(1)$.[11]

2.10 Párovací halda

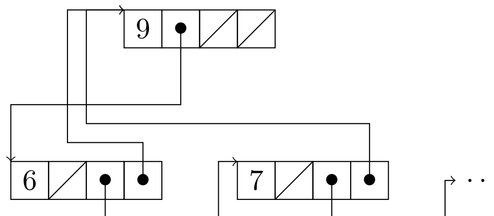
I když jsou Fibonacciho haldy velice efektivním druhem hald, jejich implementace je značně komplikovaná a v praxi nejsou tak rychlé, jak by bylo potřeba. Proto byly vyvinuty párovací haldy (originálně *pairing heaps*), které sice nemají tak dobré teoretické vlastnosti[25][26] jako Fibonacciho haldy, zároveň jsou ale mnohem snadněji implementovatelné a v praxi často rychlejší.[27]

Na párovací haldy lze hledět jako na líné samovyvažovací binomiální haldy, které ale zároveň sdílí jednoduchost implementace a praktickou rychlost skew hald[28]. Vyvažování probíhá pouze při provádění operace `dequeue`.[27]

2.10.1 Struktura haldy

Každá párovací halda je reprezentována endogenním[11] haldově uspořádaným stromem, což je zakořeněný strom, kde každý uzel je prvkem haldy, a zároveň v něm platí haldová podmínka.

Pro efektivní implementaci je použita v případě párovacích hald tzv. *child-sibling* reprezentace, tedy každý uzel obsahuje levý ukazatel na prvního ze svých potomků a pravý na staršího souseda. Aby byly efektivní i operace `change_key` a `delete`, bude potřeba udržovat u každého uzlu i ukazatel na jeho rodiče.



Obrázek 2.9: Reprezentace stromů v párovací haldě.

2.10.2 Spojování

Nejzákladnější operací, kterou musí párovací halda podporovat, je spojení dvou haldově uspořádaných stromů. To probíhá připojením kořene stromu, jehož klíč je menší, pod kořen druhého stromu. Při implementaci stromů podle obrázku 2.9 se jedná o konstantní operaci.

[11] Pojem pochází z [29] a znamená, že není rozlišován uzel stromu a prvek haldy.

2.10.3 Vkládání a hledání maxima

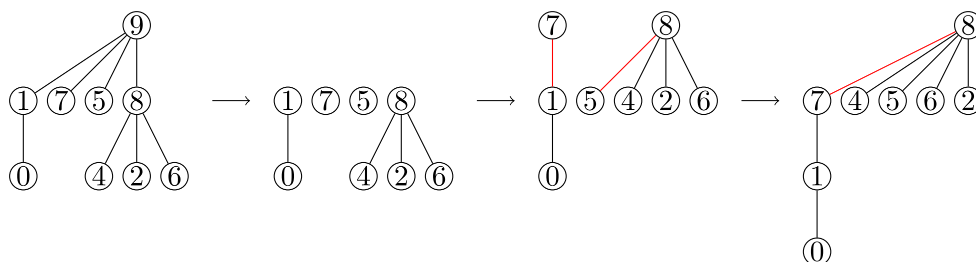
Vložení nového prvku probíhá velice podobně, jako u binomiálních hald. Tedy je vytvořena nová halda s jedním prvkem, která je následně spojena se stávající haldou. Vytvoření i spojení jsou konstantní operace, tedy i vkládání bude mít konstantní složitost.

Jelikož je párovací halda reprezentována haldově uspořádaným stromem, bude se maximum nacházet vždy v kořeni. Získání největšího prvku je tedy také konstantní operace.

2.10.4 Mazání

Nejkomplikovanější operací z hlediska implementace je pak odstranění maxima. Nejdříve je odstraněn maximální prvek x, který je určitě kořenem stromu reprezentujícího haldu. Původní strom se rozpadne na množinu S haldově uspořádaných stromů, které jsou vždy zakořeněné v jednom ze synů původního x. Všechny tyto stromy jsou následně spojeny do nového haldově uspořádaného stromu, je ovšem potřeba dát pozor na pořadí, v jakém ke spojení dojde. Amortizovaná složitost je podle [27] logaritmická s počtem prvků v haldě, což bylo později i dokázáno v [26].

K udržení potřebné amortizované složitosti jsou nejdříve všechny stromy $s \in S$ po dvojicích zleva doprava spojeny do párů, které jsou následně v opačném směru spojeny do výsledné haldy. Odstranění maxima ilustruje obrázek 2.10



Obrázek 2.10: Mazání maxima z párovací haldy. Nejdříve je odtržen její kořen x. Ze synů x vznikne les haldově uspořádaných stromů, který je po dvojicích zleva doprava spojen. Následně jsou při zpětném průchodu spojeny všechny vzniklé páry do jedné výsledné haldy.

Mazání obecného prvku je podobné mazání maxima. Obecný prvek x je odtržen ze stromu a následně smazán. Z jeho podstromů je pak stejným postupem, jako v případě mazání maxima, zkonstruována nová halda, která je následně spojena s původní haldou. Složitost mazání obecného prvku je stejná, jako mazání maxima.

2.10.5 Změna klíče

Změna klíče prvku je realizována jeho samotnou změnou. Pokud je měněný prvek kořenem haldy, není již dál potřeba cokoliv dělat. V opačném případě je takový prvek společně se svými podstromy odtržen a spojen s původní haldou. Podle [26] je časová složitost změny klíče rovna $\mathcal{O}(2^{2\sqrt{\log\log n}})$.

2.11 Striktní Fibonacciho halda

Poslední představenou implementací prioritní fronty bude striktní Fibonacciho halda. Jedná se o první datovou struktura, která má časové složitosti stejné, jako Fibonacciho halda, ovšem ne amortizovaně, ale pro nejhorší případ. Tedy vkládání nového prvku, nalezení maxima, sloučení hald a změna klíče mají v nejhorším případě složitost $\mathcal{O}(1)$ a smazání maxima i smazání obecného prvku mají složitost v nejhorším případě $\mathcal{O}(\log n)$.[30]

2.11.1 Struktura haldy

Jako první bude ukázána struktura a vlastnosti striktní Fibonacciho haldy a budou definovány všechny další podpůrné struktury.

Každá striktní Fibonacciho halda je reprezentována jedním haldově uspořádaným stromem, kde každý uzel uchovává jeden prvek haldy. Dále velikost stromu je počet jeho uzlů a stupeň uzlu x je rovný počtu jeho synů, značeno $deg(x)$. Každý uzel stromu je označen buď jako **aktivní** (*active*) nebo **pasivní** (*passive*).[30]

Aktivní uzel s pasivním rodičem pak bude zván **aktivní kořen** (*active root*). Krom toho je u každého aktivního uzlu udržována jeho **hodnost** (*rank*), která představuje počet aktivních synů, a je mu také přiřazeno nezáporné celé číslo zvané **ztráta** (*loss*). Celková ztráta celé haldy je pak součet ztrát všech aktivních uzlů.[30]

Pasivní uzel je zván **spojovatelný** (*linkable*), jestliže jsou všechny jeho synové také pasivní.[30]

Všechny prvky haldy, krom kořene, jsou udržovány ve frontě Q. Libovolný prvek, který není kořenem, bude uvažován na pozici p, pokud je p-tý ve frontě Q.

Pro analýzu složitosti striktních Fibonacciho hald se bude hodit také hodnota R, která je definována jako $2 \cdot \log(n) + 6$, kde n značí počet prvků haldy.

Nyní několik invariantů, které musí v každé striktní Fibonacciho haldě platit.[30]

Definice 5 (Struktura striktní Fibonacciho haldy)**.** *Pro striktní Fibonacciho haldy platí:*

1. *Kořen stromu tvořícího haldu je vždy pasivní.*

2. *Pro všechny uzly platí následující uspořádání synů: aktivní synové jsou vždy nalevo od pasivních synů a spojovatelné pasivní uzly jsou vždy napravo od ostatních pasivních uzlů.*

3. *Pro i-tý nejpravější aktivní uzel platí, že součet hodnosti a ztráty je minimálně $i-1$.*

4. *Aktivní kořen má nulovou ztrátu.*

Definice 6 (Počet aktivních kořenů)**.** *Počet aktivních kořenů ve striktní Fibonacciho haldě je maximálně $R+1$.*

Definice 7 (Celková ztráta haldy)**.** *Celková ztráta haldy je maximálně $R+1$.*

Definice 8 (Stupně uzlů)**.** *Pro stupně uzlů ve striktní Fibonacciho haldě platí:*

1. *Maximální stupeň kořene je $R+3$.*

2. *Nechť x je libovolný uzel haldy, který není kořen, a p označuje jeho pozici ve frontě Q. Je-li x pasivní uzel nebo aktivní uzel s kladnou ztrátou, je jeho stupeň maximálně $2 \cdot \log(2n-p)+9$, jinak může být maximální stupeň uzlu o jedna větší.*

Z definice 8 vyplývá, že libovolný uzel má stupeň maximálně $2\cdot\log(n)+12$. Dále platí následující věta o vztahu mezi R a maximální hodností:

Věta 8. *Jsou-li splněny podmínky z definice 5 a L označuje celkovou ztrátu, potom maximální hodnost je nejvýše $\log(n)+\sqrt{2L}+2$.*

Důkaz. Důkaz předchozí věty není pro tuto práci zásadní, proto bude čtenář pouze odkázán na jeho originální znění, které je možné najít v [30, s. 3]. □

Z definice 7 je známo, že $L \leq R+1$. R. Po dosazení do výrazu z věty 8 vychází $\log(n)+\sqrt{2(R+1)}+2 \leq R$, což je pro $R = 2\cdot\log(n)+6$ splněno a vyplývá tvrzení, že každý uzel má hodnost nejvýše R.[30]

2.11.2 Transformace

Je zřejmé, že modifikující operace nad haldou mohou některou z dříve uvedených vlastností striktní Fibonacciho haldy porušit. Je tedy třeba dále představit transformace, které se v případě porušení vlastností postarají o nápravu.

Základní úpravou je připojení (*link*) uzlu x a jeho podstromu pod jiný uzel y. Nejdříve dojde k odstranění x ze seznamu synů jeho původního rodiče a připojení pod y. Pokud je x označen jako aktivní, stane se z něj nejlevější syn y, v opačném případě se stává nejpravějším synem.[30]

První transformací, kterou je možné použít k nápravě vlastností haldy je **redukce aktivního kořene** (*active root reduction*). Nechť x a y jsou aktivní

kořeny stejné hodnosti r a klíč x je b.ú.n.o větší, než klíč y, potom je uzel y připojen k uzlu x a hodnost x se zvětší o jedna. Pokud je navíc nejpravější syn z uzlu x pasivní, potom je třeba ho připojit pod kořen celého stromu. Při této transformaci je počet aktivních kořenů snížen o jedna, zatímco stupeň kořene se o jedna zvětší.[30]

Předchozí transformace navýšila stupeň kořene, což může být nežádoucí. Proto další transformace, kterou halda podporuje, je **redukce stupně kořene** (*root degree reduction*). Ať x, y a z jsou tři nejpravější pasivní spojovatelní synové kořene haldy a zároveň b.ú.n.o platí $x.key < y.key < z.key$. Potom jsou x a y označeny jako aktivní uzly, z je připojen k y a y je připojen k x, který se stává nejlevějším synem kořene. Uzlům x i y je dále nastavena nulová ztráta a jednotková, respektive nulová, hodnost.[30]

Poslední možná transformace je **redukce ztráty** (*loss reduction*), kterou lze rozdělit na dva podpřípady.

V případě, že existuje uzel aktivní uzel se ztrátou minimálně 2, potom je aplikována **jedno-uzlová redukce ztráty** (*one-node loss reduction*). Nechť y je otcem x. Potom je x připojen ke kořeni a stává se aktivním kořenem s nulovou ztrátou. Zároveň je hodnost y snížena o jedna. Pokud navíc y není aktivním kořenem, je jeho ztráta zvětšena také o jedna. Jelikož původní hodnota ztráty x byla zmenšena minimálně o 2 a ztráta y byla navýšena o 1, celkově je ztráta haldy snížena minimálně o jedna.[30]

Druhou možností je tzv. **dvou-uzlová redukce ztráty** (*two-node loss reduction*). Ta je provedena v případě, že existují dva uzly x a y s hodností r a jednotkovou ztrátou, pro které b.ú.n.o platí, že $x.key < y.key$. Dále nechť z označuje rodiče uzlu y. Po připojení uzlu y pod x dojde k zvětšení hodnosti x o jedna. Ztráta x i y je ponížena o 1, stejně jako stupeň a hodnost z. Jestliže z není aktivním kořenem, je jeho ztráta zvýšena také o jedna.[30]

Nyní, již se znalostí všech transformací a vlastností uzlů, budou představeny i všechny podporované operace.

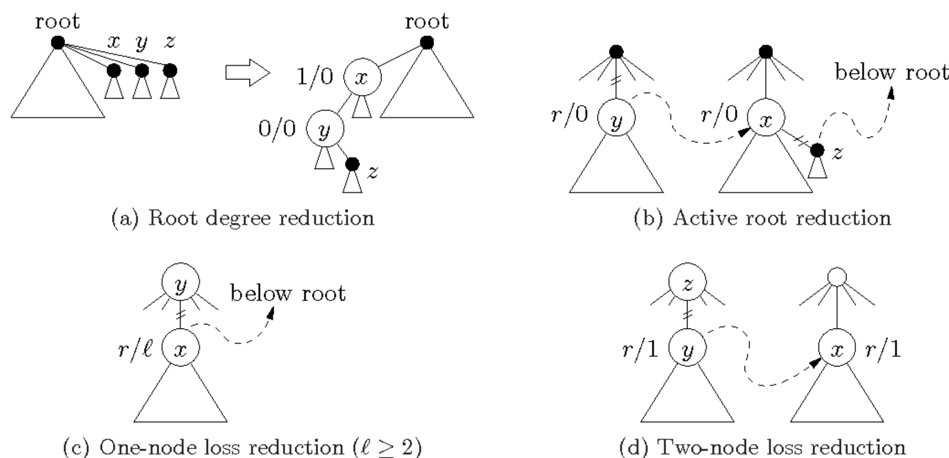
2.11.3 Vkládání

Při vkládání nového prvku je třeba vytvořit nový strom tvořený jediným uzlem, který bude navíc označen jako pasivní. Tento bude následně sloučen se stávající haldou pomocí algoritmu `merge`.[30] Složitost takové operace závisí na složitosti algoritmu slučování hald, ta je $\mathcal{O}(1)$, tedy i vkládání prvku bude konstantní operace.

2.11.4 Spojování

Předchozí operace se odkazovala na algoritmus slučování dvou Fibonacciho hald, bylo by tedy korektní tento algoritmus představit a řádně zanalyzovat.

Pro spojení dvou hald x a y je nejdříve potřeba označit všechny uzly stromu, jehož velikost je menší, jako pasivní, což proběhne, díky struktuře



Obrázek 2.11: Transformace striktních Fibonacciho hald sloužící k redukci stupně kořene, počtu aktivních kořenů a celkové ztráty. Černá, resp. bílá barva uzlu ukazuje na pasivní, resp. aktivní uzly. Hodnoty r/l představují hodnost/ztrátu aktivních uzlů. Převzato z [30].

stromů, v konstantním čase. Dále nechť u označuje kořen s větším klíčem a v kořen s menším klíčem. Potom je v připojen jako syn u, dále jsou do nové fronty Q vloženy všechny prvky fronty haldy s menší velikostí, uzel v, a následně i všechny prvky fronty druhé haldy. Nakonec je třeba provádět redukci aktivního kořene a redukci stupně kořene až do chvíle, kdy k transformacím zbývají nějaké vhodné vrcholy.[30]

2.11.5 Mazání maxima

Při mazání maxima ze stromu s kořenem z se začíná s hledáním syna x, jehož klíč je v rámci potomků z největší. Pokud je x aktivní uzel, potom musí být označen jako pasivní a zároveň se všechny jeho aktivní synové stávají aktivními kořeny. Všichni ostatní synové uzlu z jsou následně připojeny pod x. Všichni spojovatelní pasivní synové uzlu x se posunou v seznamu na nejpravější pozice, x je odstraněn z fronty Q a uzel z je nakonec zničen. Poté se dvakrát opakuje následující: počáteční uzel y z Q je zařazen na konec fronty, dva nejpravější synové y jsou, pokud jsou pasivní, připojeny k x. Po zopakování předchozího algoritmu je, pokud je to možné, provedena redukce ztráty, která je následována redukcí aktivních kořenů a redukcí stupně kořene. Ty už mohou probíhat v libovolném pořadí a jsou prováděny dokud existuje uzel, nad kterým je možné transformace provádět.[30]

2.11.6 Hledání maxima

K získání maxima stačí vrátit prvek tvořící kořen haldy.[30]

2.11.7 Změna klíče

Při změně klíče prvku uloženého v uzlu x ve stromu s kořenem z dojde nejdříve k samotné změně klíče. Pokud je x kořenem stromu, není potřeba dále nic dělat. Jinak, když klíč x je větší, než klíč z, je potřeba provést přehození ukazatelů na prvky uzlů x a z. Dále nechť y je rodič uzlu x. Uzel x je připojen ke kořeni stromu. Pokud byl x aktivním uzlem, ale ne aktivním kořenem, stane se aktivním kořenem s nulovou ztrátou a hodnost y se sníží o jedna. Jestliže je y aktivním uzlem, ale ne aktivním kořenem, pak je jeho ztráta zvětšena o jedna. Pokud je to možné, je potřeba provést redukci ztráty. Nakonec by mělo proběhnout ještě šest redukcí aktivních kořenů a čtyři redukce stupně kořene.[30]

2.11.8 Mazání obecného prvku

Vymazání obecného prvku sestává, stejně jako u většiny ostatních hald, ve změně jeho klíče na $+\infty$ a následného odmazání maxima.[30] Změna klíče má sice v případě striktních Fibonacciho hald složitost $\mathcal{O}(1)$, nicméně mazání maxima je logaritmická operace, tedy celková složitost mazání obecného prvku bude $\mathcal{O}(\log n)$.[30]

V rámci uplynulé kapitoly bylo představeno jedenáct možných implementací prioritní fronty. Jsou mezi nimi struktury od základních, zejména polí a spojových seznamů, až po velice komplikované, jako je většina pokročilých hald.

Poslední věcí, která bude v kapitole obsažena, bude přehled asymptotických složitostí představených datových struktur. Ten se nachází v tabulkách 2.1 a 2.2.

Po teoretické stránce se vítězem pomyslného souboje implementací prioritní fronty stává striktní Fibonacciho halda, která je jedinou strukturou, jejíž asymptotické složitosti se rovnají hranicím složitosti představeným v kapitole 1.2.

Tabulka 2.1: Přehled asymptotických složitostí základních operací pro jednotlivé implementace

	`enqueue`	`front`	`dequeue`
neřazené pole	$\mathcal{O}^*(1)/\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
řazené pole	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
neřazený spojový seznam	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
řazený spojový seznam	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
binární vyhledávací strom	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
červeno-černý strom	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
binární halda	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
binomiální halda	$\mathcal{O}^*(1)/\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Fibonacciho halda	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}^*(\log n)$
párovací halda	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}^*(\log n)/\mathcal{O}(n)$
striktní Fibonacciho halda	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Tabulka 2.2: Přehled asymptotických složitostí rozšiřujících operací pro jednotlivé implementace

	`change_key`	`delete`	`merge`
neřazené pole	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
řazené pole	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
neřazený spojový seznam	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
řazený spojový seznam	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
binární vyhledávací strom	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
červeno-černý strom	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
binární halda	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
binomiální halda	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Fibonacciho halda	$\mathcal{O}^*(1)$	$\mathcal{O}^*(\log n)$	$\mathcal{O}(1)$
párovací halda	$\mathcal{O}^*(2^{2\sqrt{\log \log n}})$	$\mathcal{O}^*(\log n)/\mathcal{O}(n)$	$\mathcal{O}(1)$
striktní Fibonacciho halda	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$

Kapitola **3**

Měření výkonnosti

Poslední kapitola se bude věnovat samotnému měření výkonností jednotlivých implementací prioritní fronty popsaných a realizovaných v kapitole 2.

Rychlost jednotlivých variant prioritní fronty bude měřena na školním výpočetním svazku STAR, který má následující parametry:

Tabulka 3.1: Konfigurace měřicího serveru

Procesor:	Intel Core i7-950, 3.06 GHz, 8 MB cache
Operační paměť:	24 GB DDR3
Grafická karta:	GeForce GTX 590 a GeForce GTX 470
Operační systém:	CentOS Linux 7 64bit

Server je přímo určen na provádění výpočtů a měření doby jejich běhu, nebude tedy třeba řešit prioritizování procesů a podobné problémy. O to se postará systém sám.

Pro zajištění co nejpřesnějších výsledků proběhl každý test několikrát s různými daty a nakonec byl pro diskuzi výsledků použit medián dosažených časů.

Z důvodů lepší čitelnosti jsou všechny hodnoty v grafech zobrazeny na logaritmické škále.

3.1 Test řazením

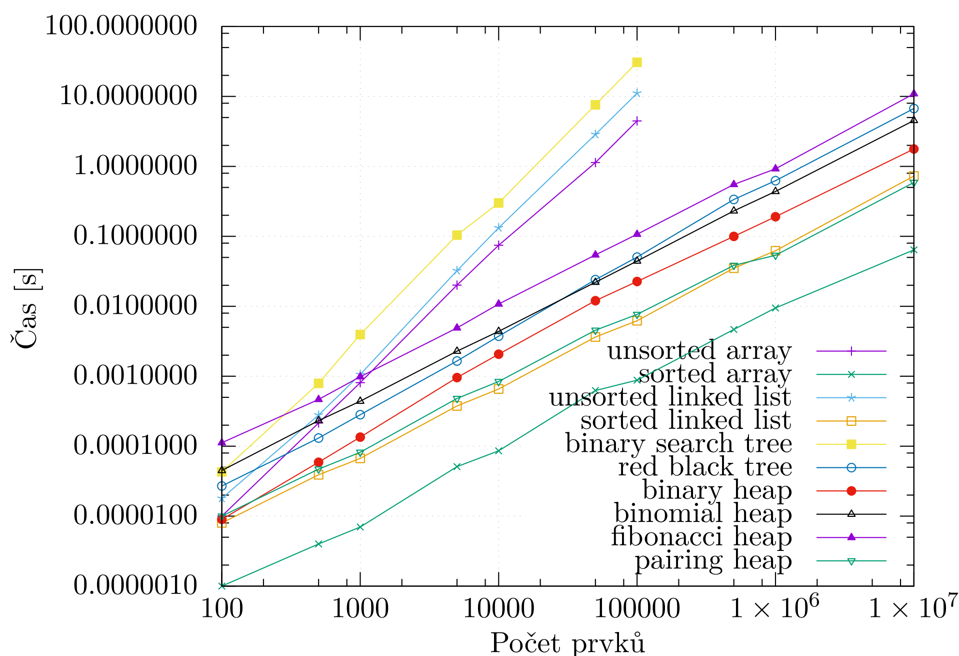
Jako první proběhne test řazením. Ten se zaměřuje zejména na rychlost provádění operací `enqueue` a `dequeue` a přímo vychází z algoritmu Heapsort popsaného v kapitole 1.3.3.

Nejdříve je prioritní fronta naplněna všemi prvky. Ty jsou následně po jednom vybrány v pořadí od největšího po nejmenší, tedy dojde k seřazení zadané posloupnosti.

Samotný test obsahuje tři fáze. Nejdříve dojde ke změření rychlosti na téměř sestupně seřazené posloupnosti, následně budou struktury testovány náhodnými daty. Poslední test pak bude obsahovat vzestupně seřazenou posloupnost.

3.1.1 Opačně seřazená posloupnost

V prvním testu byly jednotlivé fronty spuštěny se vzestupně uspořádanou posloupností kladných celých čísel, tedy opačně, než v jakém pořadí jsou z maximové haldy vybírány.



Obrázek 3.1: Výsledky testu řazení opačně seřazenou posloupností

V tomto testu, jehož výsledky jsou vidět na obrázku 3.1, se ukázala jako nejrychlejší implementace prioritní fronty seřazené pole. A není se co divit, opačně seřazená posloupnost je přesně typ dat, který se pro řazené pole implementované tak, jak bylo popsáno v kapitole 2.2, hodí nejvíce. Prvky jsou do pole vkládány odzadu. Jelikož jsou vzestupně uspořádané, dostane se na konec pole vždy největší hodnota, není tedy třeba s ním nikam hýbat. Vložení jednoho prvku bude tedy probíhat v čase $\mathcal{O}^*(1)$, mazání dokonce v $\mathcal{O}(1)$.

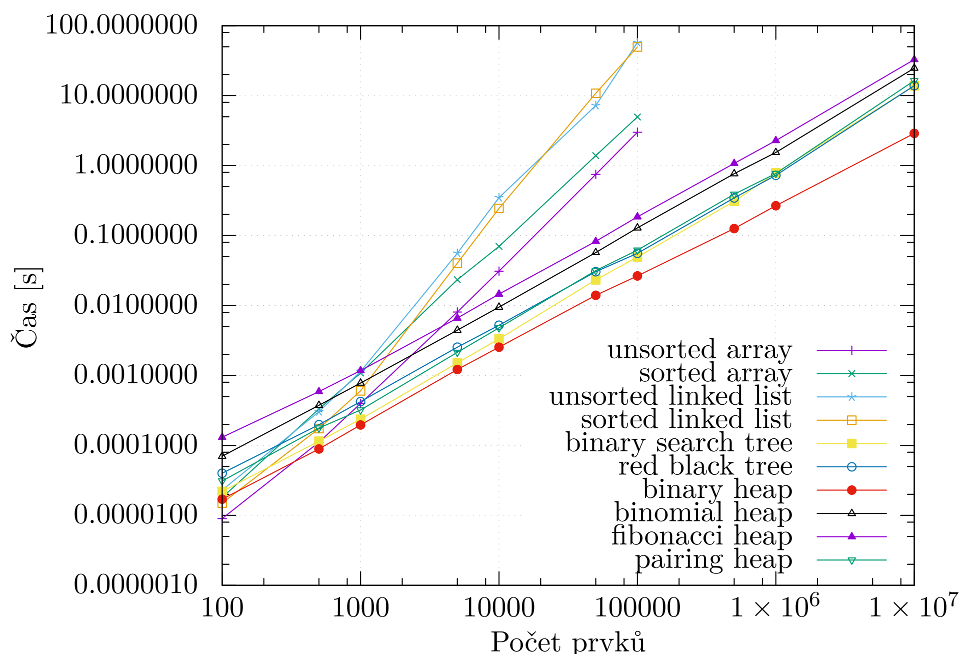
Podobně dobře dopadl i řazený spojový seznam, ten z podobných důvodů, jako tomu bylo u řazeného pole, a párovací halda, která tak dostála své pověsti snadno implementovatelné a v praxi poměrně rychlé datové struktury.

Nejhorší se ukázala býti implementace pomocí binárního vyhledávacího stromu. V případě jakkoliv seřazené posloupnosti je strom deformován na

lineární řetězec, tedy jeho hloubka je rovna $\mathcal{O}(n)$, a čas potřebný na projití testu se tak neúměrně navyšuje.

3.1.2 Náhodná data

V druhém testu byla do prioritních front nejdříve vložena a následně vybrána náhodná data. Díky použití stejné hodnoty `seed` pro inicializace generátoru náhodných čísel pro každou z front bylo zaručeno, že každá implementace dostane identická data.



Obrázek 3.2: Výsledky testu řazení náhodnými daty

Na testu náhodnými daty (obr. 3.2) se ukazuje, jak výrazně výkonnější jsou při obecném užití stromy oproti jiným implementacím. Nejlepší výkonnosti dosahuje binární halda, v těsném závěsu se pak drží párovací halda společně s vyhledávacími stromy.

Není také bez zajímavosti, jak moc podobných výsledků dosahují binomiální a Fibonacciho halda. Je známý fakt, že jsou-li nad Fibonacciho haldou používány pouze operace `enqueue` a `dequeu`, jsou stromy tvořící Fibonacciho haldu binomiálními stromy. U Fibonacciho haldy je třeba na provedení jedné operace vkládání či mazání maxima vykonat více procesorových instrukcí, což se také na čase projeví. Pro velké hodnoty se zdá ovšem býti tendence taková, že Fibonacciho halda tu binomiální předežene.

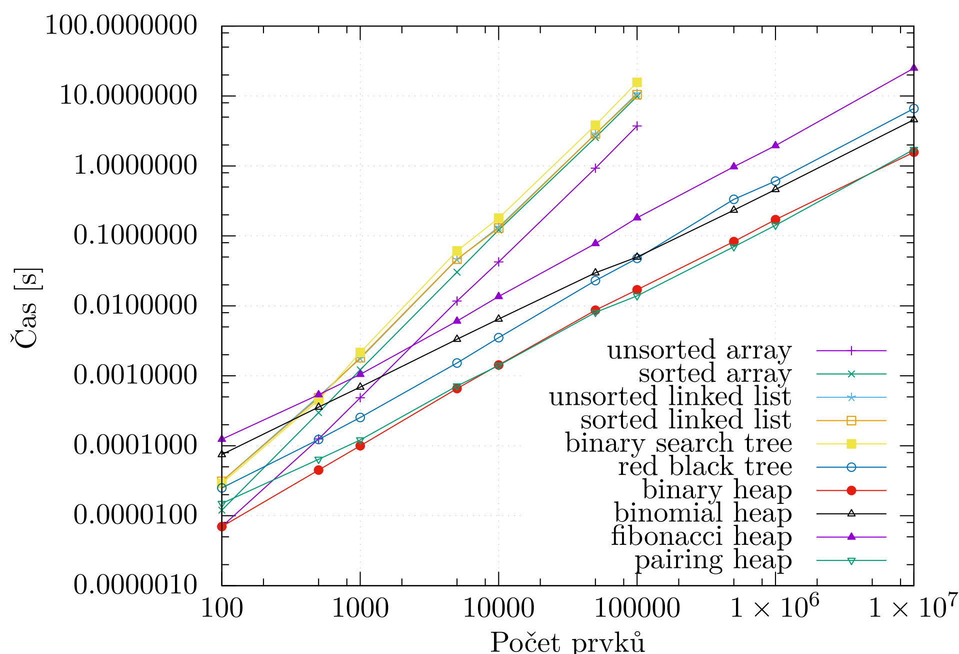
Na druhé straně se také ukazuje, že v případě náhodných dat jsou výsledky řazených i neřazených polí dosti podobné. Zatímco řazené pole má

lineární složitost vkládání nového prvku a konstantní složitost odebírání maxima, u neřazeného je to přesně naopak. Výsledný výkon se pak dosti podobá. Podobná situace je i v případě implementace pomocí spojového seznamu.

V případě binárního vyhledávacího stromu jsou u náhodných dat výsledky mnohem lepší, než u minulého testu, jelikož strom není tak výrazně degenerovaný.

3.1.3 Seřazená posloupnost

Poslední test spočíval v řazení již seřazené posloupnosti.



Obrázek 3.3: Výsledky testu řazení seřazenou posloupností

Výsledky měření zachycuje obrázek 3.3. Vůbec nejlepších výsledků dosáhla binární a párovací halda. V případě seřazené posloupnosti nebude u binární haldy docházet při žádném vložení k nápravě haldové vlastnosti haldy, tedy složitost jednoho vložení bude $\mathcal{O}(1)$.

Nedaleko za binární haldou se pohybují teoreticky výkonnější binomiální a Fibonacciho halda.

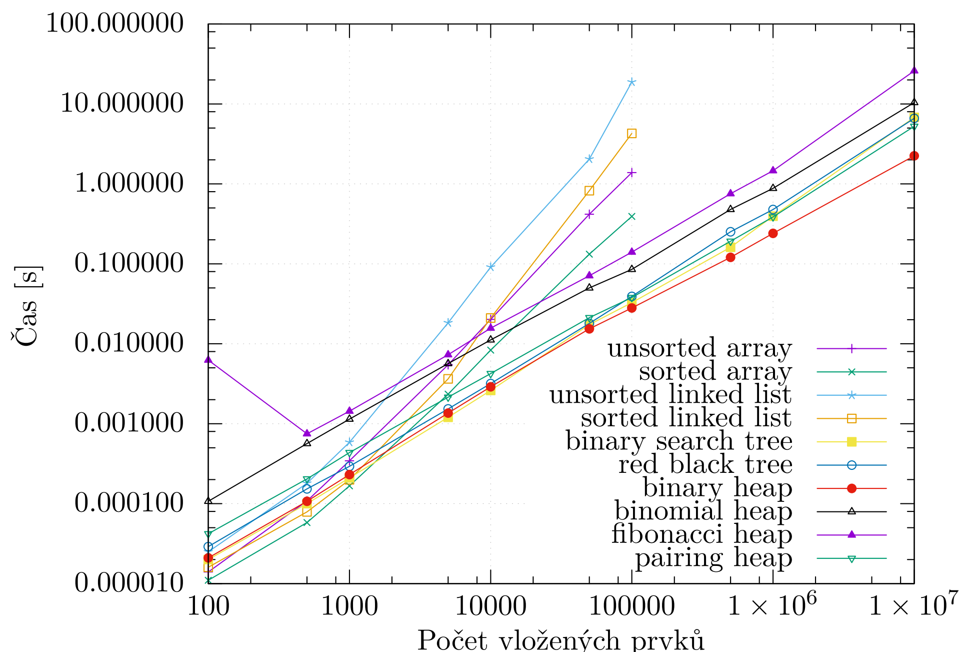
Tradičně špatného výkonu dosahuje binární vyhledávací strom. Ten je v tomto případě, stejně jako při opačně seřazené posloupnosti, degenerován na lineární řetězec uzlů, tedy bude jeho hloubka lineární s počtem prvků. Na případu červeno-černého stromu se jasně ukazuje, jak výrazný rozdíl ve výkonu dělá vyvažování.

Oproti opačně seřazené posloupnosti si také výrazně pohoršil řazený spojový seznam a řazené pole. V tomto případě je pořadí prvků přesně opačné, než by bylo třeba, dojde tedy při každém jednom vložení k projití všech prvků. Složitost vložení n prvků tedy bude $\mathcal{O}(n^2)$.

3.2 Test vkládání a výběru

Druhý z testů, který nad implementacemi prioritní fronty proběhl, byl test vkládání a výběru. Ten spočíval ve střídavém provádění operací `enqueue` a `dequeue` v poměru přibližně 3:2. Ve chvíli, kdy počet vložených prvků dosáhl dané hranice, byly z fronty vybrány i zbývající prvky.

Test tak svými vlastnostmi simuluje použití prioritních front například při diskrétních simulacích (viz kapitola 1.3.4). Výsledky reprezentuje obrázek 3.4.



Obrázek 3.4: Výsledky testu vkládání a výběru

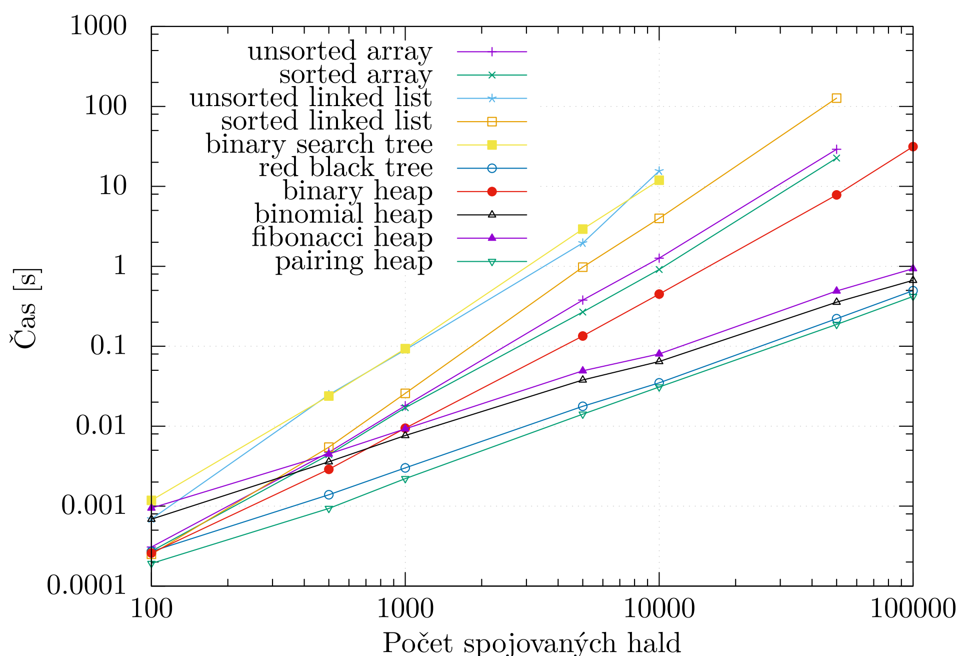
V tomto testu poměrně výrazně vede binární halda. Zajímavých hodnot dosahuje, zejména pro menší hodnoty, také binární vyhledávací strom, s přibývajícím množstvím prvků je ovšem z grafu jasně patrná tendence směrem ke zpomalení.

Výsledky také ukazují, že v případě použití pouze operací `enqueue` a `dequeue` se nevyplatí využívat Fibonacciho nebo binomiální haldu. Ty se svými vlastnostmi specializují spíše na případy, kdy je často prováděno slévání hald.

Na chvostu zůstávají, jako již tradičně, jednoduché implementace pomocí pole a spojového seznamu.

3.3 Test spojování

Jako poslední proběhne test spojování. Ten spočívá ve vytvoření velkého množství front, jejich naplnění a následném spojování pomocí operace `merge`. Nakonec byla z výsledné prioritní fronty všechna data vybrána.



Obrázek 3.5: Výsledky testu spojování

V tomto testu (obr. 3.5) se konečně projevila síla binomiálních a Fibonacciho hald, které předstihly i jinak dobře si vedoucí binární haldu. Je tak v praxi ověřen teoretický předpoklad, totiž že binomiální, resp. Fibonacciho, haldu se hodí použít tam, kde je často využívána operace spojování hald.

Překvapivě dobrých výsledků dosáhli červeno-černé stromy, a to i přes to, že jejich spojování bylo implementováno pouze pomocí vložení všech prvků jedné haldy do druhé, tedy s asymptotickou složitostí $\mathcal{O}(n \log n)$. Naopak u binárního vyhledávacího stromu, kde byl použit lineární algoritmus, jsou výsledky o dost horší.

Velice špatně v tomto testu dopadl neřazený spojový seznam, který by měl být ve spojování teoreticky mnohem lepší, než například neřazené pole.

3.4 Diskuze dosažených výsledků

Jednoduché implementace prioritní fronty pomocí pole nebo spojového seznamu se vyplatí použít pouze v případě nadmíru příznivých dat. V tu chvíli budou, díky své jednoduchosti, excelovat. Pro obecné případy se vůbec nehodí.

Obecný binární vyhledávací strom si naopak v obecném případě až tak špatně nevede. Existují ovšem případy, kdy dojde k jeho degeneraci, a potom jeho výkonnost rapidně klesá. Je tedy lepší využít některou z variant samovyvažovacích binárních vyhledávacích stromů. Režie potřebná k vyvažování se ve výsledcích neprojevila.

Poslední velkou rodinou datových struktur použitých k implementaci prioritní fronty jsou haldy. V případě testů, kde byly použity pouze operace vkládání a výběru prvků, dosahovaly nejlepších výsledků binární a párovací halda. A to i přes to, že jejich teoretické složitosti nejsou tak dobré, jako například u Fibonacciho hald. Zásluhu na tom má především složitost implementace a z toho plynoucí větší násobící konstanty u složitostí, které u v práci měřeného počtu prvků/operací nutně projeví.

Naopak při častém používaní operace spojování dat se projevily dobré vlastnosti binomiální a Fibonacciho haldy. Tedy v aplikacích, kde často dochází ke slučování prioritních front, se zjevně vyplatí použít právě zmíněné.

3.4.1 Datová senzitivita

Na dalších řádcích bude rozebrána datová citlivost jednotlivých implementací, tedy to, jak je výkon každé prioritní fronty ovlivněn složením dat.

Jako nejstabilnější a zároveň nejvýkonnější implementace se jeví většina hald. Binomiální, Fibonacciho, párovací i binární halda mají téměř identické výsledky v každém z proběhlých testů, dokonce i jejich pořadí, seřadíme-li je dle rychlosti, je pokaždé téměř stejné.

Z pohledu datové senzitivity vychází dobře i červeno-černé stromy. Naopak obecný vyhledávací strom v případě nepříznivých data degeneruje na lineární řetězec uzlů a tím se ztrácí i jeho efektivita.

Velké rozdíly v rychlosti byly zaznamenány u spojových seznamů a polí, zejména těch řazených. Ty dosahovaly fantastických výsledků v případě příznivých dat (řazené posloupnosti), u ostatních testů jsou pak jejich výsledky značně neuspokojivé.

3.4.2 Striktní Fibonacciho halda

Pozorný čtenář si jistě všiml, že v žádném z výsledků měření není zahrnuta striktní Fibonacciho halda. Tu se bohužel z důvodu velké komplexnosti implementovat nepodařilo. Z měření, které provedl Larkin, Sen a Tarjan v [31] ovšem vyplývá, že i když má striktní Fibonacciho halda po teoretické stránce ideální vlastnosti, v praxi je velice pomalá.

Závěr

V rámci práce se podařilo nastudovat a implementovat hned několik zajímavých datových struktur. Práce v tomto ohledu pochopitelně není, a nemůže být, vyčerpávající, na některé další vhodné tak bylo v textu alespoň odkázáno.

Měření rychlosti implementovaných prioritních front bylo provedeno hned podle několika možných testovacích scénářů. Po přečtení výsledků by měl být čtenář schopný pro svou konkrétní aplikaci vybrat z představených datových struktur tu ideální implementaci prioritní fronty.

Výsledky dosažené v měřeních se nijak zásadně neliší od výsledků dosažených v pracech jiných autorů. Velice dobré výkonnosti dosahuje zejména párovací halda, která je ve školním prostředí značně opomíjená. Důvody jsou ovšem zřejmé, teoretická analýza její složitosti byla až donedávna jedním z otevřených problémů informatiky, což není pro základní kurzy algoritmů příliš šťastné. Binární halda, která je v akademických podmínkách jednou z prvních představených komplexnějších datových struktur, ovšem dosáhla ve většině testů podobně dobrých výsledků, nelze tedy rozhodně říci, že by ve škole byly vyučovány pouze struktury s příznivou teoretickou analýzou, které jsou v praxi nepoužitelné. Ostatní pokročilé haldy, tedy zejména Fibonacciho a binomiální, dosahovaly výsledků dle očekávání, tedy dobře si vedly v případě častého spojování hald, v ostatních případech se projevila jejich implementační složitost.

Na odvedenou práci lze navázat zejména nastudováním dalších datových struktur, které mohou prioritní frontu také implementovat. Krom toho je možné porovnat dosažené výsledky s implementací prioritní fronty ze standardní knihovny jazyka C++ . V neposlední řadě by bylo namístě rozšířit testy i na další operace, případně měřit jednotlivé implementace s větším množstvím dat. Zároveň v práci nedošlo k implementaci striktní Fibonacciho haldy, tato výzva tedy také zůstává otevřená.

Literatura

[1] Skiena, S. S.: *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, druhé vydání, 2008, ISBN 1848000693, 9781848000698.

[2] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; aj.: *Introduction to Algorithms*. The MIT Press, třetí vydání, 2009, ISBN 0262033844, 9780262033848.

[3] Jenses, C.: *Theoretical and practical efficiency of priority queues*. Diplomová práce, University of Copenhagen, Květen 2006. Dostupné z: http://www.diku.dk/~jyrki/PE-lab/Claus/index_Master.html

[4] Fredman, M. L.; Tarjan, R. E.: Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM*, ročník 34, č. 3, Červenec 1987: s. 596–615, ISSN 0004-5411, doi:10.1145/28869.28874. Dostupné z: http://doi.acm.org/10.1145/28869.28874

[5] Sedgewick, R.; Wayne, K.: *Algorithms*. Addison-Wesley Professional, čtvrté vydání, 2011, ISBN 032157351X, 9780321573513.

[6] Thorup, M.: Equivalence Between Priority Queues and Sorting. *J. ACM*, ročník 54, č. 6, Prosinec 2007, ISSN 0004-5411, doi: 10.1145/1314690.1314692. Dostupné z: http://doi.acm.org/10.1145/1314690.1314692

[7] Dijkstra, E. W.: A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, ročník 1, č. 1, Prosinec 1959: s. 269–271, ISSN 0029-599X, doi:10.1007/BF01386390. Dostupné z: http://dx.doi.org/10.1007/BF01386390

[8] Huffman, D. A.: A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, ročník 40, č. 9, Září 1952: s. 1098–1101,

ISSN 0096-8390, doi:10.1109/JRPROC.1952.273898. Dostupné z: `http://ieeexplore.ieee.org/document/4051119/`

[9] Williams, J. W. J.: Algorithm 232 - Heapsort. *Communications of the ACM*, ročník 7, č. 6, Červen 1964: s. 347–348, ISSN 0001-0782, doi:10.1145/512274.512284. Dostupné z: `http://doi.acm.org/10.1145/512274.512284`

[10] Mehlhorn, K.; Sanders, P.: *Algorithms and Data Structures: The Basic Toolbox*. Springer Publishing Company, Incorporated, první vydání, 2008, ISBN 9783540779773.

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

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

[13] Tarjan, R. E.: Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods*, ročník 6, č. 2, 1985: s. 306–318, doi:10.1137/0606031. Dostupné z: `https://doi.org/10.1137/0606031`

[14] Knuth, D. E.: *The Art of Computer Programming: Volume 3: Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., druhé vydání, 1998, ISBN 0-201-89685-0.

[15] Brass, P.: *Advanced Data Structures*. New York, NY, USA: Cambridge University Press, první vydání, 2008, ISBN 0521880378, 9780521880374.

[16] Sahni, S.: *Data Structures, Algorithms and Applications in Java*. New York, NY, USA: McGraw-Hill, Inc., první vydání, 1999, ISBN 007109217X.

[17] Bayer, R.: Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, ročník 1, č. 4, Prosinec 1972: s. 290–306, ISSN 1432-0525, doi:10.1007/BF00289509. Dostupné z: `https://doi.org/10.1007/BF00289509`

[18] Sleator, D. D.; Tarjan, R. E.: Self-adjusting Binary Search Trees. *J. ACM*, ročník 32, č. 3, Červenec 1985: s. 652–686, ISSN 0004-5411, doi:10.1145/3828.3835. Dostupné z: `http://doi.acm.org/10.1145/3828.3835`

[19] Guibas, L. J.; Sedgewick, R.: A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, Říjen 1978, ISSN 0272-5428, s. 8–21, doi:10.1109/SFCS.1978.3.

[20] Floyd, R. W.: Algorithm 245: Treesort 3. *Communications of the ACM*, ročník 7, č. 12, Prosinec 1964: s. 701–, ISSN 0001-0782, doi:10.1145/355588.365103. Dostupné z: `http://doi.acm.org/10.1145/355588.365103`

[21] Suchenek, M. A.: Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program. *Fundamenta Informaticae*, ročník 120, č. 1, Leden 2012: s. 75–92, ISSN 0169-2968. Dostupné z: `http://dl.acm.org/citation.cfm?id=2385656.2385660`

[22] Sack, J.-R.; Strothotte, T.: An Algorithm for Merging Heaps. *Acta Informatica*, ročník 22, č. 2, Červen 1985: str. 171–186, doi:10.1007/BF00264229. Dostupné z: `https://doi.org/10.1007/BF00264229`

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

[24] Vuillemin, J.: A Data Structure for Manipulating Priority Queues. *Commun. ACM*, ročník 21, č. 4, Duben 1978: s. 309–315, ISSN 0001-0782, doi:10.1145/359460.359478. Dostupné z: `http://doi.acm.org/10.1145/359460.359478`

[25] Fredman, M. L.: On the Efficiency of Pairing Heaps and Related Data Structures. *J. ACM*, ročník 46, č. 4, Červenec 1999: s. 473–501, ISSN 0004-5411, doi:10.1145/320211.320214. Dostupné z: `http://doi.acm.org/10.1145/320211.320214`

[26] Pettie, S.: Towards a Final Analysis of Pairing Heaps. In *Proceedings of the 46th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '05, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2468-0, s. 174–183, doi:10.1109/SFCS.2005.75. Dostupné z: `https://doi.org/10.1109/SFCS.2005.75`

[27] Fredman, M. L.; Sedgewick, R.; Sleator, D. D.; aj.: The pairing heap: A new form of self-adjusting heap. *Algorithmica*, ročník 1, č. 1, Listopad 1986: s. 111–129, ISSN 1432-0541, doi:10.1007/BF01840439. Dostupné z: `https://doi.org/10.1007/BF01840439`

[28] Sleator, D. D.; Tarjan, R. E.: Self Adjusting Heaps. *SIAM J. Comput.*, ročník 15, č. 1, Únor 1986: s. 52–69, ISSN 0097-5397, doi:10.1137/0215004. Dostupné z: `http://dx.doi.org/10.1137/0215004`

[29] Tarjan, R.: *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983, doi:10.1137/1.9781611970265. Dostupné z: `https://epubs.siam.org/doi/abs/10.1137/1.9781611970265`

[30] Brodal, G. S.; Lagogiannis, G.; Tarjan, R. E.: Strict Fibonacci Heaps. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1245-5, s. 1177–1184, doi:10.1145/2213977.2214082. Dostupné z: http://doi.acm.org/10.1145/2213977.2214082

[31] 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*, Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, s. 61–72. Dostupné z: http://dl.acm.org/citation.cfm?id=2790174.2790181

Příloha A

Slovník

ADT Abstraktní datový typ.

b.ú.n.o Bez újmy na obecnosti.

BVS Binární vyhledávací strom.

FIFO First in, first out.

LIFO Last in, first out.

Příloha B

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  thesis.pdf ...................... text práce ve formátu PDF
```