VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PRAKTICKÁ EFEKTIVITA KONTEJNERŮ

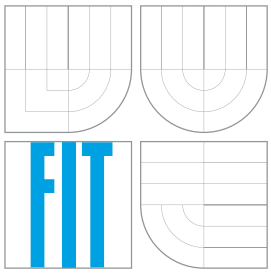
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN HALÁMKA

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PRAKTICKÁ EFEKTIVITA KONTEJNERŮ

PRACTICAL EFFICIENCY OF CONTAINERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN HALÁMKA
AUTHOR

VEDOUCÍ PRÁCE — Mgr. LUKÁŠ HOLÍK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá praktickým porovnáním výkonu datových kontejnerů a struktur jazyka C^{++} pro reprezentaci množin. Toho bylo dosaženo testováním jejich časové složitosti v průměrném případě. Testované struktury byly úspěšně porovnány. Na základě zjištěných údajů je umožněna snažší volba struktury pro konkrétní případ.

Abstract

This thesis is about practical comparrison of performance of containers and data structures in C^{++} for reprezentation of sets. This goal was reached by testing their average case time complexity. Tested structures were succesfully compared. Information provided makes choosing data structure for particular case easier.

Klíčová slova

binomiální halda, datová struktura, fibonacciho halda, kontejner, strom, vektor

Keywords

avl-tree, b-tree, binomial heap, container, data structure, deque, hash table, list, rb-tree, rope, scapegoat, sg-tree, skip list, splay-tree, treap, vector

Citace

Jan Halámka: Praktická efektivita kontejnerů, bakalářská práce, Brno, FIT VUT v Brně, 2015

Praktická efektivita kontejnerů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Lukáše Holíka

.....
Jan Halámka
22. května 2015

Obsah

Kapitola 1

Úvod

Tato práce se zabývá praktickým porovnáním výkonu datových kontejnerů (datový kontejner určuje způsob jakým jsou data uložena v paměti počítače a jak se s nimi dá manipulovat) při práci s množinami.

Pro porovnání kontejnerů existují matematické nástroje jako asymptotická analýza. Tato práce si neklade (a ani nemůže) za cíl tyto prostředky nahradit, ani se jim vyčerpávajícím způsobem věnovat. Jsou sice okrajově zmíněny, ale cílem práce je spíše doplnit informace, které se jimi získat nedají.

Mezi takové informace patří třeba praktické porovnání výkonu Hashovací tabulky. Ta má sice lineární asymptotické složitosti operací, ale v drtivé většině případů z praxe výkonem překonává i struktury s o třídu lepší asymptotickou složitostí.

Práce se zabývá kontejnery (a jejich modifikacemi) používanymi v C^{++}. Především těmi v rozšířených knihovnách (standardní a Boost), ale dostane se i na některé méně obvyklé z odborné literatury (například Rope, Binomiální a Fibonacciho halda). Pro názornost budou zařazeny i jednoduché struktury, které se pro reprezentaci množin obvykle nepoužívají (Vector a List) a které by měly ve většině případů podávat výrazně nejhorší výsledky (kvůli vysokým asymptotickým složitostem).

Budou zjištěny a porovnány teoretické vlastnosti všech vybraných kontejnerů. Dále bude navržena sada testů, vybrány a popsány konkrétní varianty implementace testovaných struktur (některé méně obvyklé bude třeba upravit, či naprogramovat). Nakonec budou uvedeny naměřené výsledky a jejich vyhodnocení.

Cílem práce je poskytnout čtenáři informace užitečné při rozhodování, zda se mu vyplatí investovat prostředky potřebné k použití složitějších datových struktur a případně kterou z nich zvolit, tak aby pro něj byla co nejvýhodnější, podle kritérií, která si sám stanoví.

Kapitola 2

Výkonnost kontejnerů

K porovnání kontejnerů lze přistoupit ze dvou hledisek, prostorového (kolik místa v paměti struktura vyžaduje) a časového (jak rychle lze provést operace nad ní). Vzhledem k dlouhodobému trendu výrazného poklesu ceny pamětí bych se v rámci této práce chtěl zaměřit na časové hledisko.

Při porovnávání časové výkonnosti datových struktur se zpravidla využívá vyhodnocení na základě následujících kritérií:

- Asymptotická časová složitost
- Amortizovaná časová složitost
- Časová složitost v průměrném případě
- Praktické porovnání

2.1 Asymptotická časová složitost

Asymptotická časová složitost udává, jakým způsobem se bude měnit chování operace nad kontejnerem v závislosti na počtu jeho prvků.

Používá se tzv. Omikron (Θ) notace, která udává, že průběh funkce zobrazující čas potřebný k provedení operace v závislosti na počtu prvků kontejneru je asymptoticky ohraničen funkcí v omikron notaci z obou stran (předpokládáme zanedbání vlivu konstant).

Formálně lze tento vztah vyjádřit jako: $f(x) \in \Theta(g(x))$, právě tehdy když $\exists(C, C_1 > 0)$, $x_0 : \forall(x > x_0)|Cg(x)| < |f(x)| < |C_1 g(x)|$, kde $f(x)$, $g(x)$ jsou reálné funkce a C, C_1 konstanty viz. [5].

Asymptotická složitost nám tedy říká, jak dlouho bude v nejhorším případě trvat provedení každé jednotlivé operace. A nejlépe se hodí použít tam, kde potřebujeme eliminovat výkonnostní propady a zaručit tak dobrou odezvu u provedení nejhoršího případu.

2.2 Amortizovaná časová složitost

Podle [1] amortizovaná analýza vychází z myšlenky, že i když jsou některé operace nad strukturou mimořádně drahé, nemusí se objevovat dostatečně často na to, aby snížili hodnotu přístupu k datové struktuře jako celku. Je to dáno tím, že tyto drahé operace nám

později umožní provést následujících mnoho operací levněji, než by bylo za normálních okolností možné. Přístup jako celek se tedy při provedení velkého počtu operací může prokázat jako účinnější.

Nejjednodušší amortizovanou datovou strukturou je Vector blíže popsaný v sekci 3.1.1. Který v případě potřeby alokace paměti vždy zvětší svoji velikost na dvojnásobek. Toto chování zaručuje, že k uložení n hodnot bude potřeba maximálně $\log_2(n)$ alokací paměti a zabere maximálně $2n$ místa v paměti.

Výpočet amortizovaného času na provedení operace se dá vyjádřit jako: $\frac{T_{(n)}}{n}$, kde $T_{(n)}$ odpovídá nejhoršímu času provedení sekvence operací a n jejich počtu v sekvenci.

Na rozdíl od Asymptotické složitosti je Amortizovaná složitost zaměřena na posloupnosti operací. Hodí se ji sledovat, pokud předpokládáme časté opakování sekvencí operací.

2.3 Časová složitost v průměrném případě

Někdy se také označuje jako očekávaná složitost. Tento přístup počítá s tím, že existuje velká množina průměrných vstupů, pro které má algoritmus dobrou složitost a velmi malá množina vstupů, pro které má špatnou složitost. Neboli, že algoritmus bude s vysokou pravděpodobností (např. $1 - 1/n$, kde n je počet prvků), vyhodnocen s dobrou (časovou) složitostí.

Na rozdíl od Amortizované složitosti, která se zabývá posloupností operací se složitost v průměrném případě týká jednotlivých operací a doba provedení závisí na vstupních datech. Při jejím testování je tedy nutno zvolit správnou testovací množinu.

Struktury spoléhající na tento přístup jsou například Hashovaná tabulka viz. 3.1.5, nebo Skiplist viz. 3.3.5.

2.4 Praktické porovnání

Proč vůbec prakticky testovat a porovnávat, když máme k dispozici kvalitní matematické nástroje jako asymptotickou a amortizovanou složitost?

Slabou i silnou stránkou obou výše zmíněných metod je to, že počítají s chováním kontejnerů pro tak velké množství prvků, že se konstanty stávají nepodstatnými. Tyto konstanty mohou ovšem mít zcela zásadní vliv zvláště u menšího počtu prvků. (Mějme funkci $f(x) = 1000000N$ a funkci $g(x) = (n + 100)^2$ asymptotické složitosti by pak odpovídaly: $\Theta f(x) = N$ a $\Theta g(x) = N^2$, můžeme si povšimnout, že i když má $f(x)$ lepší asymptotickou složitost bude se pro kontejner s 1 až 990 000 prvky chovat hůře, než $g(x)$.)

Dalším důvodem je porovnání výkonu struktur se stejnou asymptotickou složitostí.

A konečně výrobci procesorů své čipy optimalizují pro určité operace, díky čemuž by se mohly i teoreticky výkonnější struktury ukázat jako méně vhodné.

Kapitola 3

Kontejnery

V této kapitole si projdeme vybrané kontejnery a jejich teoretické vlastnosti. Tabulka v sekci 3.4 obsahuje teoretické složitosti operací nad těmito kontejnery.

3.1 Standardní kontejnery

Kontejnery vyskytující se ve standardní knihovně (standart C^{++}11).

3.1.1 Vector

V podstatě odpovídá poli s podporou vkládání nových prvků. Garantuje alokaci celistvého bloku paměti. Někdy bývá označován také jako dynamické, či bufferované pole. Vkládání nových prvků má lineární asymptotickou složitost. Pokud budeme vkládat prvky vždy na konec vectoru, tak bychom se měli dostat až na konstantní amortizovanou složitost, díky bufferování. Mazání prvku má lineární časovou složitost a nelze ji amortizovat. Vyhledání (a přístup k) prvku má buď lineární, nebo logaritmickou časovou složitost (v závislosti na tom, jestli je vector seřazen). Díky výše zmíněné garanci alokace celistvého bloku paměti umožňuje vector konstantní indexování obdobně jako pole.

3.1.2 Deque

Deque může být implementována buď pomocí obousměrného seznamu, nebo bufferovaného pole. V obou případech umožňuje rychlou práci s prvky na obou koncích a přístup k prvkům pomocí indexu. Negarantuje použití celistvého bloku paměti. Implementace pomocí bufferovaného pole se provádí několika různými způsoby:

- jako kruhový buffer
- jako pole, které je plněno od středu ke krajům
- jako několik menších polí

Vkládání nových prvků má lineární asymptotickou složitost. Vkládání maxima, ale i minima má konstantní amortizovanou složitost. To samé platí pro mazání a vyhledávání.

3.1.3 List

Klasický obousměrný seznam. Lineární asymptotická složitost vkládání, mazání i vyhledání prvku. Na rozdíl od vectoru a deque pro každý prvek alokuje zvláštní místo v paměti.

3.1.4 Set

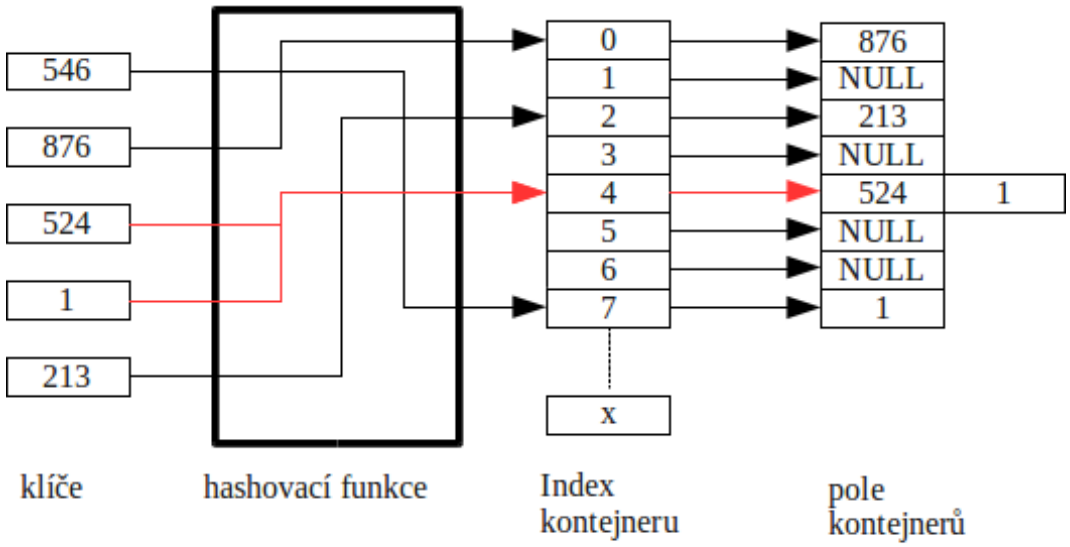
Množina implementovaná pomocí binárního vyhledávacího stromu.

Operace vložení, smazání a vyhledání prvku mají logaritmickou asymptotickou složitost.

Často se používá RB-tree v sekci:3.2.2, ale existují i varianty založené na jiných stromových strukturách - viz. sekce 3.2.1 až 3.3.1.

3.1.5 Hash table

Asociativní pole jehož prvky jsou další datové kontejnery (často obyčejné seznamy). Na data je aplikována tzv. hashovací funkce, která z dat vypočítá index do pole. S prvkem se dále pracuje ve struktuře na vypočteném indexu. V ideálním případě se na každé pozici v poli nachází právě jeden prvek. Na obrázku 3.1 je ilustrace hashovací tabulky.



Obrázek 3.1: Hashovací tabulka

Asymptotická složitost vyhledání prvku je vysoká (lineární). Přesto se očekává velmi rychlé vyhledání prvku (odpovídající konstantní asymptotické složitosti). Totéž platí i pro přidání a smazání. Výkon závisí na kvalitě hashovací funkce.

3.2 Pokročilé kontejnery

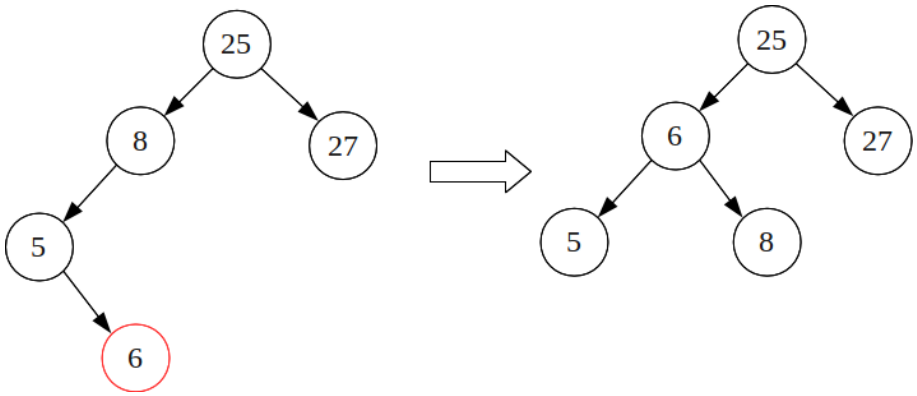
Intrusivní datové kontejnery vyskytující se v knihovně boost. Podle [6] by tyto kontejnery měly často podávat lepší výkony, než kontejnery ze standardní knihovny. Od standardních kontejnerů se liší tím, že v paměti neodkládají kopie objektů, ale pouze ukazatele na ně.

3.2.1 AVL-tree

Binární samovyvažovací vyhledávací strom. Platí pro něj následující:

- Hodnoty levého podstromu jsou nižší, než hodnota kořene.
- Hodnoty pravého podstromu jsou vyšší, než hodnota kořene.
- Výška levého podstromu se od výšky pravého podstromu liší maximálně o 1.

V případě, že by vložení nového prvku porušilo třetí pravidlo dojde k rotaci jako na obrázku 3.2. Vložení smazání i vyhledání prvku má logaritmickou asymptotickou složitost.



Obrázek 3.2: AVL-tree před a po vložení prvku, který porušuje vyvažovací pravidlo.

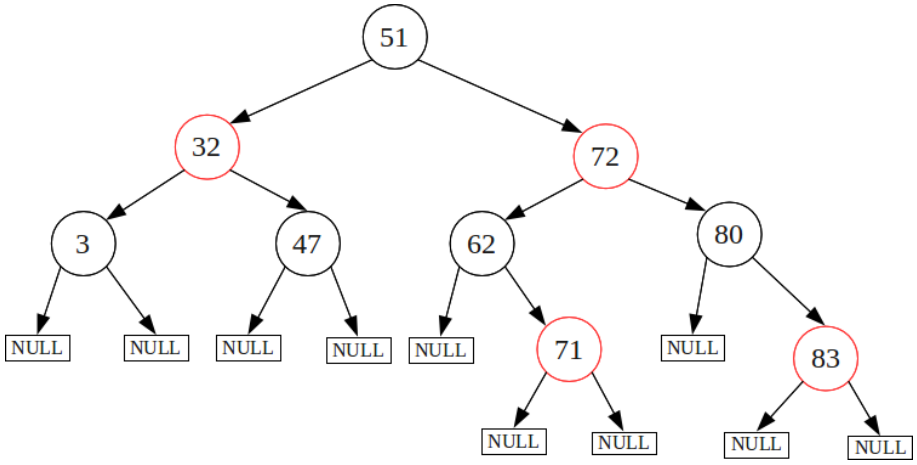
Implementace převzata z boost::intrusive::avltree

3.2.2 RB-tree

Red Black tree je binární vyhledávací samovyvažovací strom. V každém uzlu je kromě dat uložen jeden bit navíc - informace o jeho barvě. Tato vlasnost slouží k zajištění přibližného vyvážení stromu (respektive omezuje jak maximálně může být strom nevyvážený).

Pravidla pro uspořádání RB-tree:

- Hodnoty levého podstromu jsou nižší, než hodnota kořene.
- Hodnoty pravého podstromu jsou vyšší, než hodnota kořene.
- Každý uzel je buď černý, nebo červený.
- Kořen a NULL na konci listů jsou černé.
- Potomci červeného uzlu jsou černé.
- Každá cesta mezi NULL a kořenem obsahuje stejný počet černých uzlů.



Obrázek 3.3: Red Black-tree

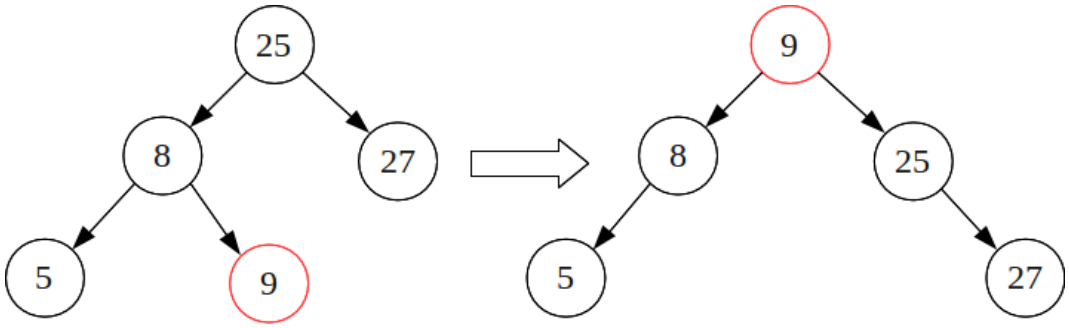
Příklad Red Black tree je na obrázku 3.3.

Pokud se strukura stromu změní, tak je vytvořen nový strom a jeho uzly znovu obarveny a uspořádány. Vše je navrženo tak, aby toto znovuvyvažovaní a přebarvování probíhalo efektivně.

Vlastnosti Red Black tree zaručují logaritmickou asymptotickou složitost operací vložení, smazání a vyhledání prvku.

3.2.3 Splay-tree

Splay-tree je samovyvažovací binární vyhledávací strom, který navíc přemísťuje prvky které právě vyhledal do stromu, aby se k nim dalo znovu rychle přistoupit. Příklad takové operace je na obrázku 3.4.

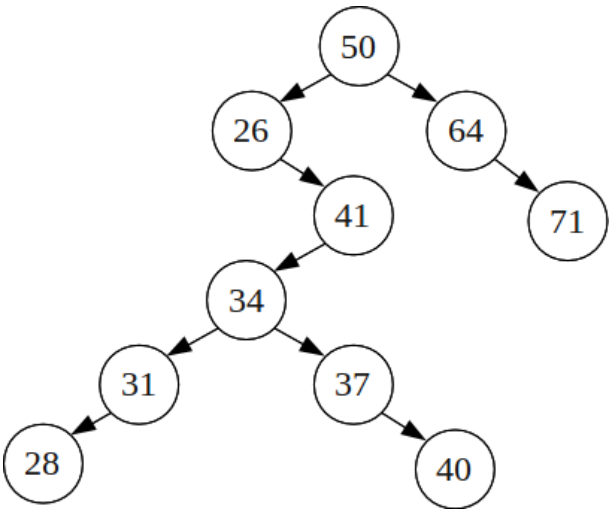


Obrázek 3.4: Splay-tree před a po vyhledání zvýrazněného uzlu.

Splay-tree má velkou nevýhodu v tom, že jeho výška může teoreticky být i lineární. Má tedy výrazně horší (lineární) asymptotickou složitost operací vložení, smazání a vyhledání prvku. Nicméně amortizovaná složitost těchto operací je logaritmická. V praxi se používá tam, kde je třeba k několika málo prvkům přistupovat výrazně častěji, než ke zbytku, například u routerů.

3.2.4 SG-tree

Scapegoat tree je binární vyhledávací samovyvažovací strom, který kromě počtu uzlů (n) uchovává i počítadlo q, které hlídá horní hranici počtu uzlů ($\frac{q}{2} \leq n \leq q$) a v každém uzlu jeho hloubku h pro niž platí, že: $h \leq \log_{\frac{3}{2}} q$. Tím je zaručena logaritmická hloubka a tedy i logaritmická asymptotická složitost operací vložení, odstranění a vyhledání prvku i když se na první pohled nemusí zdát moc vyvážený, jak je vidět na obrázku: 3.5.



Obrázek 3.5: Scapegoat tree s 10 uzly a hloubkou 5.

Vkládání prvku probíhá stejně jako u běžného binárního vyhledávacího stromu s vyjímkou situace, kdy by hloubka nového uzlu u překročila $\log_{\frac{3}{2}} q$. Pokud nastane tato situace tak je na cestě od uzlu u ke kořeni nalezen uzel s(scapegoat). Pro který platí, že poměr velikosti podstromu s na cestě k u a velikosti uzlu s je $> \frac{2}{3}$. Tento podstrom poté přeorganizujeme do podoby dokonale vyváženého binárního stromu, tím pádem snížíme výšku tohoto podstromu alespoň o 1 a nedojde k porušení pravidla.

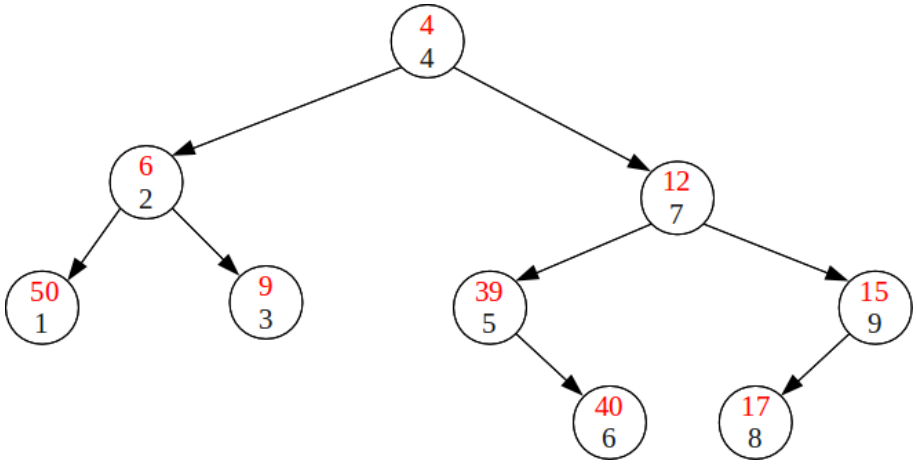
3.2.5 Treap

Treap je pravděpodobnostní stromová struktura. Název vznikl jako kombinace anglických slov "tree"(strom) a "heap"(halda) a kombinuje obě tyto struktury. Každé hodnotě v Treap je náhodně přiřazena určitá priorita (měla by být unikátní), která pomáhá vyvažování stromu. Uzly jsou uspořádány tak, že:

- Hodnoty levého podstromu jsou nižší, než hodnota kořene.
- Hodnoty pravého podstromu jsou vyšší, než hodnota kořene.
- Priority obou podromů jsou vyšší (nebo nižší), než hodnota kořene.

Při vkládání nových prvků dochází k rotacím, aby platila výše zmíněná pravidla.

Vkládání, mazání a vyhledání prvku má očekávanou logaritmickou složitost (asymptotická je lineární).



Obrázek 3.6: Treap s hodnotami 1 až 9 a zvyrazněnými prioritami.

3.3 Nestandardní kontejnery

Méně obvyklé kontejnery neimplementované ve standardních knihovnách.

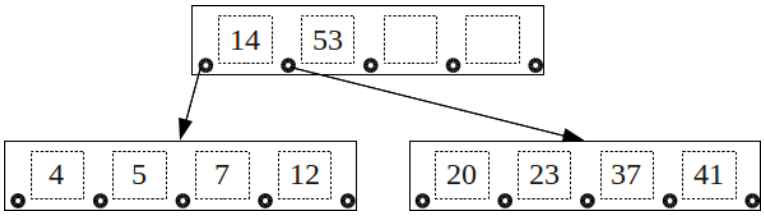
3.3.1 B-tree

Stromová struktura, kde má každý nelistový uzel dva až X (řád B-tree) potomků a o jedna méně hodnot. Tyto hodnoty určují rozdělení na potomky uzlu (N-tý ukazatel odkazuje na podstrom, kde jsou všechny hodnoty vyšší, než N-1. hodnota a zároveň nižší, než N-tá hodnota). B-tree řádu r splňuje následující:

- Každý uzel má maximálně r potomků.
- Každý nelistový uzel s vyjímkou kořene má alespoň $\frac{r}{2}$ potomků.
- Kořen má buď žádného, nebo alespoň dva potomky.
- Nelistový uzel s x potomky obsahuje $x - 1$ hodnot.
- Všechny listy jsou na stejné úrovni.

Příklad B-tree najdete na obrázku 3.7.

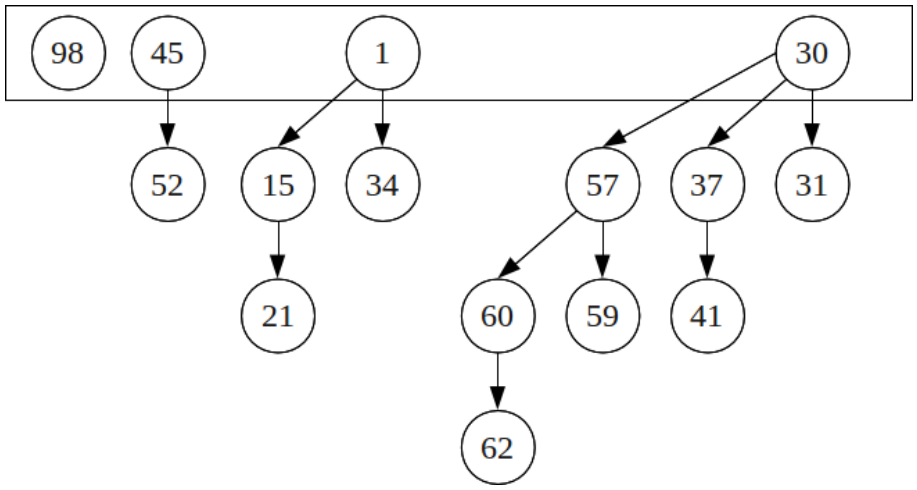
Díky variabilnímu počtu potomků nedochází k vyvažování tak často jako v případě ostatních samovyvažovacích stromů. B-tree je optimalizován pro práci s velkými bloky dat, běžně se používá v databázích a souborových systémech. Operace přidání smazání i vyhledání prvku mají logaritmickou asymptotickou složitost.



Obrázek 3.7: B-tree pátého stupně.

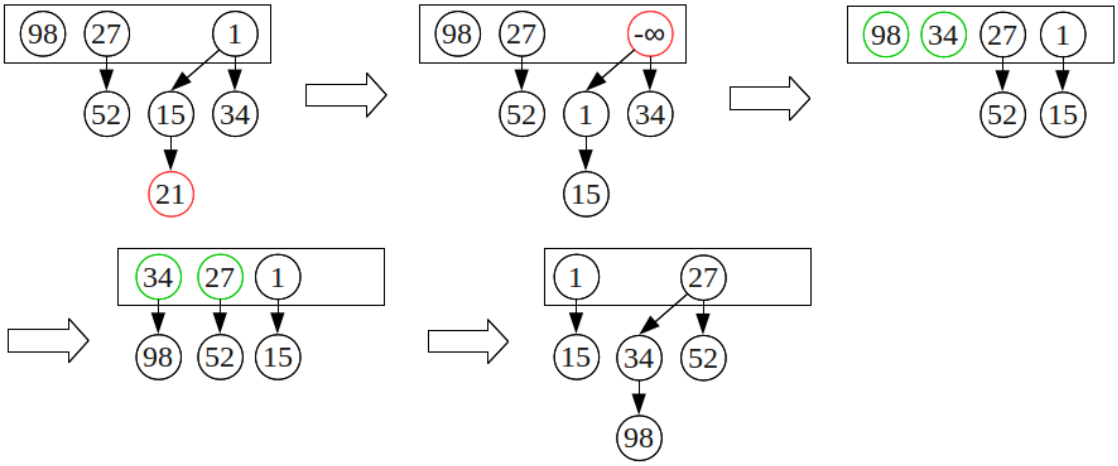
3.3.2 Binomiální halda

Seznam různě velkých binomiálních stromů uspořádaných podle velikosti. Počty uzlů stromů odpovídají mocninám dvojky. Prvky ve stromech jsou uspořádány tak, že v kořeni je vždy nejmenší (nebo největší) prvek ze stromu. Na strukturu binominální haldy se můžete podívat na obrázku 3.8. Hodí se zejména k implementaci prioritních front.



Obrázek 3.8: Binominální halda obsahující stromy nultého až třetího stupně.

Mazání prvku probíhá zajímavým způsobem viz 3.9. Nejprve je nalezen hledaný prvek, poté je jeho hodnota snížena na minimální hodnotu. Tím pádem je přesunut do kořene stromu. A následně je odstraněno minimum. Odstranění minima probíhá tak, že jeho potomci jsou zařazeni mezi potomky kořene, odstraní se a proběhne slévání. Vložení nového prvku proběhne tak, že je kořeni vytvořen nový podstrom stupně nula. A proběhne slévání – podstromy stejného stupně jsou spojeny do stromu se stupněm o jedna vyšším.

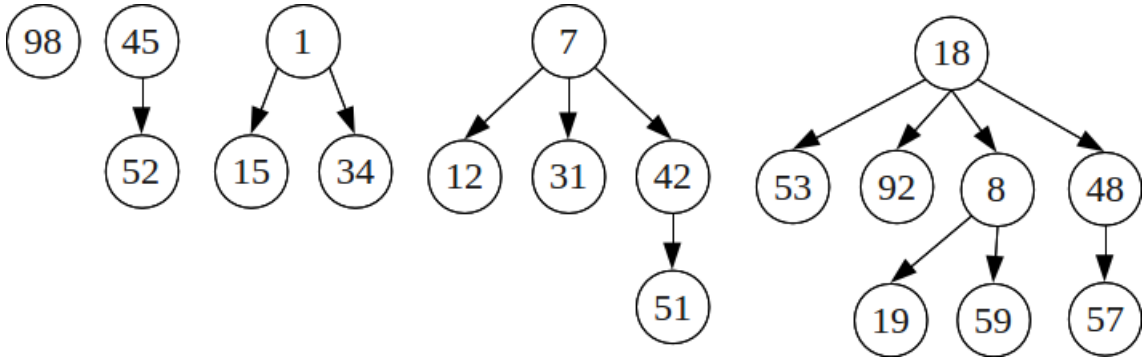


Obrázek 3.9: Odstranění zvýrazněného prvku z binomiální haldy.

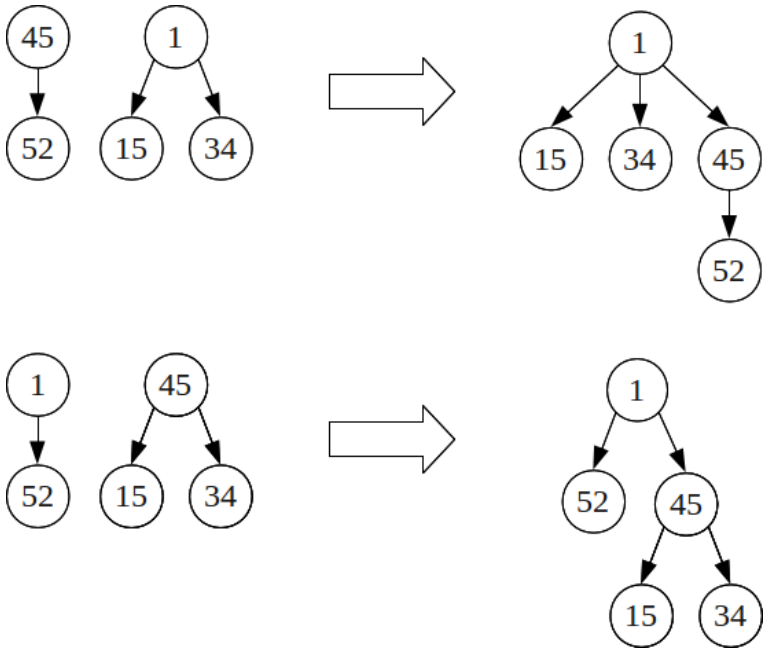
Výkonnostní propady pro malé množiny. Problematické vyhledání prvku (lineární složitost). Velmi dobrá doba vložení nového prvku (konstantní amortizovaná složitost). Efektivní práce s minimem.

3.3.3 Fibonacciho halda

Seznam různě velkých fibonacciho stromů, velmi podobná binomiální haldě v sekci 3.3.2. Počty uzlů fibonacciho stromu odpovídají prvkům fibonnaciho posloupnosti, jak je vidět na obrázku 3.10. Slévání uvnitř haldy probíhá buď pro dva stromy nultého stupně, nebo pro stromy, jejichž stupeň se liší o jedna. Na rozdíl od binomiálních stromů, kde má strom určitého stupně pevně danou strukturu, se dva fibonacciho stromy třetí a vyšší úrovně vyskytují v různých variantách. Jak k tomu dochází je ilustrováno na obrázku 3.11.



Obrázek 3.10: Fibonacciho stromy nultého až čtvrtého stupně.

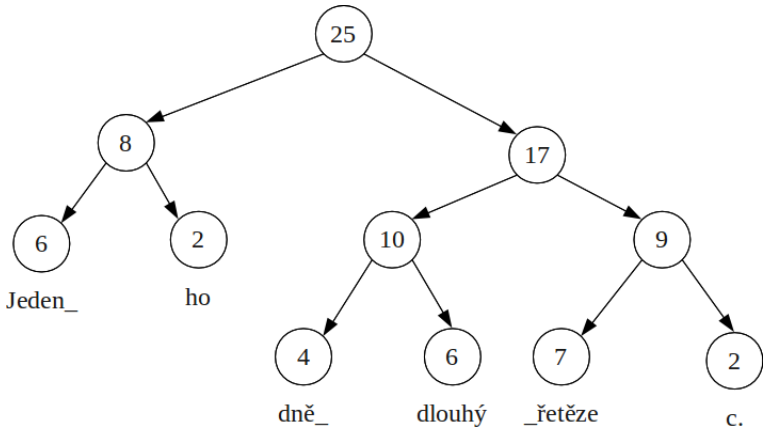


Obrázek 3.11: Vznik dvou různě strukturovaných fibonacciho stromů třetího stupně.

Operace vložení, hledání minima, snížení hodnoty klíče a spojování stromů probíhají v konstantním amortizovaném čase. Operace mazání pracuje s logaritmickou asymptotickou časovou složitostí. Používá se k vyhledávání minimální kostry grafu a k určení nejkratší cesty v grafu jako součást Jarníkova a Dijkstrova algoritmu.

3.3.4 Rope

Binární strom jehož uzly obsahují informaci o počtu prvků v podstromech, jeho listy obsahují kontejnery s uloženými prvky. Používá se zpravidla pouze na práci s řetězci. Podporuje řetězcové operace jako vyhledání podřetězce, konkatenaci a indexování. Pro srovnání s ostatními strukturami bude potřeba provést úpravy. Struktura rope je zřejmá z obrázku 3.12.



Obrázek 3.12: Typická struktura Rope

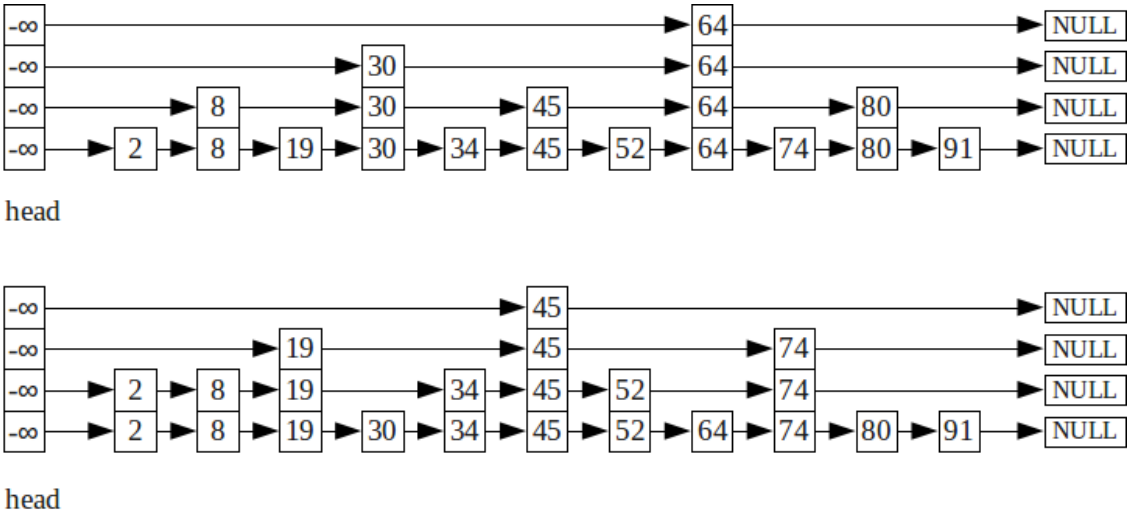
3.3.5 Skip list

Výceúrovňový seznam. Ukázku skip listů naleznete na obrázku 3.13. Existuje několik variant:

- s pevnou hloubkou (počet úrovní hlavičky a tím i celého listu je omezen)
- s variabilní hloubkou (hloubka skip listu není omezena)
- indexovatelný skip list (každý uzel obsahuje údaj o počtu prvků nejnižší úrovně mezi ním a následujícím prvkem stejné úrovně)

U skip listu s pevnou hloubkou jsou jednotlivé prvky tvořeny poli ukazatelů na následující prvky příslušné úrovně. U skip listu s variabilní hloubkou je každý uzel samostatný a obsahuje i ukazatele na sousední úrovně (pokud je v nich obsažena stejná hodnota, kterou obsahuje).

Při vkládání do skip listu je prvek s 50-ti procentní pravděpodobností povýšen na vyšší úroveň(je-li povýšen, tak opět existuje 50-ti procentní pravděpodobnost na povýšení). Při vyhledávání se postupuje z nejvyšší úrovně, díky tomu nabývá skip list logaritmické složitosti vyhledávání v seřazené sekvenci v průměrném případě (asymptotická složitost je lineární). Některé operace mohou výrazně zhoršit strukturu skip listu. Zejména se jedná o odebrání více víceúrovňových prvků za sebou. V nejhorším případě se tak můžeme dostat až na lineární časovou složitost. Proti tomuto jevu ovšem existují protiopatření. Často se používá modifikace, která opraví strukturu skip listu během operace, která má vždy lineární složitost (průchod všemi prvky).



Obrázek 3.13: Ideální a náhodně generovaný skip list

3.4 Srovnání kontejnerů

V tabulce 3.1 jsou asymptotické složitosti jednotlivých operací. Pokud se amortizovaná složitost (nebo složitost v průměrném případě) liší od asymptotické, tak je uvedena za středníkem. Operace, které nemají pro danou strukturu smysl jsou označeny znakem X. Informace označené jednou hvězdičkou platí pro práci s maximem, dvěma hvězdičkami pro práci s minimem a třemi hvězdičkami pro práci s minimem i maximem.

	Vložení	Smazání	Vyhledání	Indexování
Vector	$\Theta(n)$; $\Theta(1)$*	$\Theta(n)$; $\Theta(1)$*	$\Theta(\log(n))$	$\Theta(1)$
Deque	$\Theta(n)$; $\Theta(1)$***	$\Theta(n)$; $\Theta(1)$***	$\Theta(\log(n))$	$\Theta(1)$
List	$\Theta(n)$; $\Theta(1)$***	$\Theta(n)$; $\Theta(1)$***	$\Theta(n)$	$\Theta(n)$
Hash table	$\Theta(n)$; $\Theta(1)$	$\Theta(n)$; $\Theta(1)$	$\Theta(n)$; $\Theta(1)$	X
AVL-tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	X
RB-tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	X
Splay-tree	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	X
SG-tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	X
Treap	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	X
B-tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	X
Binomiální halda	$\Theta(1)$	$\Theta(n)$; $\Theta(\log(n))$**	$\Theta(n)$	X
Fibonacciho halda	$\Theta(1)$	$\Theta(n)$; $\Theta(\log(n))$**	$\Theta(n)$	X
Rope	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$
Skip list	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	$\Theta(n)$; $\Theta(\log(n))$	X

Tabulka 3.1: Přehled teoretických výkonností operací nad kontejnery

Kapitola 4

Testovací metodika

V následující kapitole je shrnut návrh testovací metody. Vzhledem k rozsahu práce je potřeba testování trochu omezit. Rozhodl jsem se, že budu testovat výkonnost struktur při reprezentaci množin. Budou tedy použity seřazené varianty vectoru a seznamu. A hodnoty ve strukturách budou unikátní.

4.1 Požadované informace

Při testování bych se chtěl zaměřit především na vyjádření časové náročnosti operací v závislosti na počtu prvků. Mělo by se tak prakticky ověřit, jak moc korelují naměřené hodnoty s hodnotami teoretickými. Kromě průměru budu zkoumat i nejdelší čas provedení operace.

Budu proto měřit dobu provedení operace nad strukturou. Bude vždy změřen velký počet operací provedených po sobě na dané struktuře, celkový čas provedení bude vydělen počtem opakování, abychom zjistili dobu potřebnou na jedno provedení operace. A vyjádřím růst potřebného času v závislosti na počtu prvků struktury. Jednotlivé struktury poté budou mezi sebou porovnány.

4.2 Prostředky

Pro ukládání do struktur bude použit datový typ unsigned int. Typicky budou ve strukturách uloženy ukazatele na jiné místo v paměti a vzhledem k tomu, že inteeger má velikost odpovídající ukazateli, tak by nemělo dojít k nežádoucímu zkreslení.

Hodnoty budou generovány pomocí několika různých rozložení pravděpodobnosti (normální, rovnoměrné, seřazená sekvence a seřazená sekvence v obráceném pořadí).

K měření času v maximální možné přesnosti bude použit modul chrono, integrovaný do standardní knihovny C^{++} od verze 11.

4.3 Návrh metody

Testování by mělo probíhat v následujících krocích pro počty prvků(N) od 100 do 1 000 000 s nárůstem na desetinásobek pro každou ze struktur:

- Bude změřen čas vložení N prvků.
- Bude změřen čas vyhledání N prvků.

- Pokud to struktura podporuje tak bude změřen čas indexování N prvků.
- Bude změřen čas odebrání N prvků ze struktury.

Kapitola 5

Implementace

V této kapitole bude popsána konkrétní implementace testovaných struktur.

5.1 Vector

V testech je použit vector implementovaný ve standardní knihovně, respektive jeho vzestupně seřazená varianta, která umožní využít logaritmického vyhledání prvku.

5.2 Deque

V testech je použita deque implementovaná ve standardní knihovně. Je tvořena více obyčejnými poli s pevnou velikostí a jedním dynamickým polem. Dynamické pole obsahuje ukazatele na obyčejná pole, ve kterých jsou uložena data, čímž je vyřešeno rychlé (konstantní) indexování. Stejně jako u vectoru bude použita vzestupně seřazená varianta, umožňující použití algoritmu pro logaritmické vyhledání prvku. Oproti vectoru by měla být rychlejší při práci s prvky ve velkých množinách. Protože pracuje s menšími bloky paměti.

5.3 List

V testech je použit obousměrný, seřazený seznam ze standardní knihovny. Kromě obvyklých operací nad seznamem byla přidána i operace pro indexování, spočívající v průchodu seznamem od začátku po "indexovaný"prvek. Tato hodnota bude sloužit k ilustraci výhodnosti používání struktur, které jsou k indexování optimalizovány.

5.4 Set

Implementaci množiny ze standardní knihovny jsem se rozhodl nepoužít. Místo toho jsou použity stromové implementace množiny blíže popsané v sekcích 3.2.1 až 3.2.5 z knihovny boost::intrusive a B-tree ze sekce 3.3.1 jehož implementace byla převzata z [7].

5.5 Hash table

Hashovací tabulka pro testování byla převzata ze standardní knihovny včetně integrované hashovací funkce.

5.6 Binomiální halda

Binomiální haldu jsem se rozhodl implementovat si sám podle popisu v [2]. Protože binomiální halda, kterou reprezentuje třída Binom, je v podstatě seznam binomiálních stromů, tak bylo potřeba vytvořit i třídu Tree, která je reprezentuje. Tato třída obsahuje data, ukazatel na svůj nadřazený strom, svůj stupeň a seznam podstromů. Seznam stromů v třídě Binom je řazen podle jejich stupně od nejnižšího po nejvyšší.

Přidání nového prvku "x"do haldy v kódu vypadá přibližně takto:

```
1   void vloz_prvek(x){
2     Tree novy = new Tree(x);
3     seznam_stromu->push_front(novy);
4     Merge();
5   }
6
7   void Merge(){ //slévání stromů
8     iterator it, it2;
9     it = seznam_stromu->begin();
10    it2 = it + 1;
11    while (it2 != seznam_stromu->end()){
12      if (it->level == it2->level){ //dva stromy stejné úrovně -> spojit do jednoho
13        if(it->data < it2->data){
14          it->pridej_podstrom(it2);
15          seznam_stromu->odstran(it2);
16        }
17        else{
18          it2->pridej_podstrom(it);
19          seznam_stromu->odstran(it);
20        }
21      }
22      it++;
23      it2++;
24    }
25  }
```

Funkce Merge() se stará o slévání stromů uvnitř haldy. Prochází seznam stromů a pokud narazí na dva stejné úrovně, tak porovná data v jejich kořeni a sloučí je do jednoho. Tato operace je pro haldy velmi významná, probíhá po každém přidání/odebrání prvku.

Vyhledání prvku vypadá tak, že je postupně prohledán každý podstrom. Pokud je hodnota prohledávaného uzlu větší, než hledaná hodnota, tak už není třeba prohledávat potomky uzlu. Nicméně je stále třeba porovnat stromy na stejné úrovni.

Smazání prvku oproti popisu sekci 3.3.2 nemění hodnotu nalezeného uzlu, pouze ji považuje za nové minimum. Probíhá v následujících krocích:

- je nalezen Tree s hodnotou, která má být odstraněna
- jeho hodnota je vyměněna s hodnotou nadřazeného uzlu
- 2. krok probíhá tak dlouho, dokud se odstraňovaná hodnota nedostane do kořene
- podstromy uzlu s odstraňovanou hodnotou jsou přidány do seznamu stromů
- odkaz na tento strom je odstraněn ze seznamu stromů
- seznam stromů je seřazen podle jejich úrovní a proběhne Merge()

5.7 Fibonacciho halda

Fibonacciho haldu jsem si také vytvořil sám. Vzhledem k tomu, že je binomiální haldě velmi podobná byla implementace jednoduchá. De-facto stačilo sdědit třídu Binom a změnit jeden řádek kódu v metodě Merge(). A to sice podmínku pro slévání stromů. Místo slévání stromů stejného stupně se slévají stromy, jejichž stupně se liší o 1, nebo jsou oba nulové.

5.8 Rope

Při modifikaci struktury rope mne zaujala možnost kombinace bufferovaného pole a stromu. Chtěl jsem zachovat možnost rychlého vyhledání a dobrého indexování. Dospěl jsem k následujícím vlastnostem:

- data jsou uložena v bufferovaných polích v listových uzlech stromu
- maximální velikost pole v je omezena vztahem: $v = 2^h$, kde h je hloubka uzlu
  - díky tomu je hloubka omezena na $log_2(N)$, kde N je počet prvků Rope
- dojde-li k zaplnění maximální kapacity uzlu v dané hloubce, tak bude rozdělen na dva s hloubkou o 1 vyšší, každý bude obsahovat polovinu původního počtu prvků

Díky logaritmicky omezené hloubce je zachována logaritmická asymptotická složitost vyhledání prvku a indexování. Asymptotická složitost vložení a smazání prvku zůstává lineární.

Pseudokód indexování v Rope:

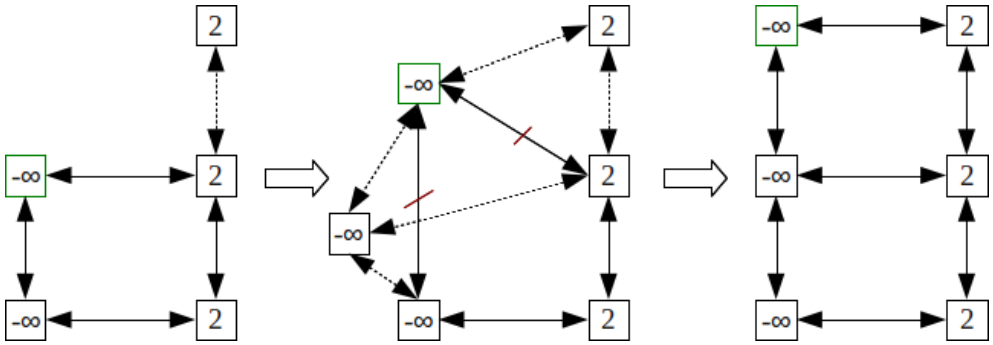
```
1  Get_at(x){
2    if(levy_potomek == NULL) // jedná se o listový uzel -> vrátit hodnotu z pole
3      return data[x];
4    else if(levy_potomek->size() > x) //hledaní v levem podstromu
5      return levy_potomek->Get_at(x);
6    else //index je za hranicemi leveho potomka -> hledat v pravem podstromu
7      return pravy_potomek->Get_at(x-levy_potomek->size());
8  }
```

5.9 Skip list

Při implementaci skip listu jsem postupoval podle popisu v článku[4]. Jen jsem před pevnou hloubkou hlavičky dal přednot variabilní. Každý prvek testované varianty skip listu obsahuje čtyři ukazatele (na následující a předchozí prvek obdobně jako u obousměrného senzamu a navíc ukazatele o úroveň výš a o úroveň níž) a hodnotu. Je implementován pomocí dvou tříd. Třídy Skiplist, která strukturu zapouzdřuje, obsahuje informaci o celkovém počtu prvků a ukazatel na prvek představující hlavičku skip listu a třídy Skipitem, která představuje prvek skip listu.

Vložení prvku je poměrně komplikované, protože je již při vkládání potřeba vytvořit prvky patřící do vyšší úrovně. Navíc je třeba speciálně ošetřit případ kdy má být některý prvek povýšen nad současnou úroveň hlavičky - je potřeba povýšit i ji - viz obrázek 5.1. Smazání prvku je na druhou stranu poměrně jednoduché.

```
1  void Remove(x){
2    if(right->data < x) //pokud prvek napravo obsahuje menší hodnotu, než x, tak posun doprava
3      right->Remove(x);
4    else if(right->data == x){ //pokud prvek napravo obsahuje x, tak ho odstraníme
5      right->right->left = this;
6      Skipitem *del = right;
7      right = right -> right;
8      delete del;
9      if(down != NULL) //odstranili jsme prvek v jeho nejvyšší úrovni, ještě musíme dolů
10       down->Remove(x);
11   } //vpravu je hodnota vyšší, než x, nebo NULL, zkusíme jít dolů
12   else if(down != NULL)
13     down->Remove(x);
14 }
```



Obrázek 5.1: Povýšení hlavičky Skip listu zároveň s prvkem

5.10 Generátor náhodných čísel

Protože některé ze struktur(Skiplist, Treap) volají rand(), tak bylo potřeba vytvořit si vlastní jednoduchý generátor náhodných čísel. Použil jsem kongruentní generátor z knihy:[3]. Má dostatečnou periodu na to, aby nedocházelo ke generování opakujících se čísel pro generované velikosti množin. Byla přidána možnost resetovat generátor na původní hodnotu seedu, aby generoval stejnou sekvenci znovu od začátku. V závislosti na inicializaci umí generovat čísla v rovnoměrném a normálním (průměr pěti hodnot) rozložení pravděpodobnosti, vzestupnou a sestupnou sekvenci.

5.11 Test průměrné složitosti

Test probíhá následovně:

- z povinného parametru je určen testovaný kontejner
- z volitelného parametru je určeno generované rozložení pravděpodobnosti (defaultně normální)
- je vytvořen kontejner
- pro každé N od 100 do 1 000 000 (pro méně výkonné struktury méně) s nárůstem na 10-ti násobek
  - je vygenerován prvek
  - je zaznamenán čas t1
  - prvek je vložen do kontejneru
  - je zaznamenán čas t2
  - rozdíl časů je přičten k celkovému součtu vkládání
  - je-li větší, než předchozí tak je zapamatován
- generátor je resetován a je provedeno to samé i pro vyhledávání prvku (poté pro indexování, pokud je kontejner podporuje) a pro mazání
- nakonec jsou změřená data vypsána.

Kapitola 6

Výsledky

Výsledky testování. V tabulkách se nacházejí naměřené časové údaje v nanosekundách. V závislosti na způsobu generování množiny (pomocí normálního a rovnoměrného rozložení pravděpodobnosti, vzestupné a sestupné sekvence), typu a počtu provedených operací. Pro každý měřený údaj jsou v tabulce uvedeny dvě položky průměrný čas provedení dané operace (avg) a nejhorší čas provedení dané operace (worst). Testy, které trvaly déle, než 10 minut byly předčasně přerušeny a položka v tabulce označena znakem "X".

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	640	6 999	622	12 000	694	92 000	742	2 628 000	712	4 565 999
Deque	1 370	3 999	2 106	31 999	2 834	34 999	3 360	36 999	4 049	33 999
List	1 710	6 000	15 478	124 000	114 627	632 999	X	X	X	X
Hash table	800	6 999	636	50 999	722	605 999	631	4 715 999	737	75 196 999
AVL-tree	1 490	6 000	1 637	25 000	1 946	34 999	2 134	20 000	2 651	30 000
RB-tree	1 440	5 000	1 571	21 999	1 935	25 000	1 969	12 000	2 563	26 999
Splay-tree	1 860	9 000	2 075	15 999	1 902	55 999	2 088	26 999	2 589	29 000
SG-tree	2 040	6 999	2 474	9 000	3 188	26 999	3 767	19 000	4 886	31 999
Treap	920	9 000	861	3 999	849	9 000	855	20 000	856	23 000
B-tree	2 010	6 999	3 143	16 999	3 647	20 999	4 075	21 999	4 658	38 000
Binomiální halda	10 770	48 999	14 709	148 999	12 573	127 999	13 635	458 999	X	X
Fibonacciho halda	20 460	92 000	12 464	106 999	12 236	242 000	12 950	476 999	X	X
Rope	719	7 999	1 823	33 999	9 915	41 000	73 676	386 000	X	X
Skip list	680	5 000	1 090	16 999	1 368	21 999	1 711	24 000	X	X

Tabulka 6.1: Vkládání do množiny generované pomocí normálního rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	620	12 000	647	15 999	676	133 000	574	248 999	707	4 582 000
Deque	1 290	9 000	3 412	33 000	2 878	25 999	3 368	20 000	3 989	36 999
List	1 650	6 999	14 884	81 000	112 064	586 000	X	X	X	X
Hash table	750	6 000	597	50 000	694	524 000	621	4 231 999	747	75 277 999
AVL-tree	1 560	5 000	1 750	10 000	2 261	163 000	2 143	20 000	2 648	25 999
RB-tree	1 440	5 000	1 550	6 999	1 894	46 999	2 082	27 999	2 582	26 999
Splay-tree	1 850	7 999	2 092	23 000	2 431	68 999	2 395	58 000	2 799	41 999
SG-tree	1 920	6 999	5 942	1 879 000	3 169	33 000	4 481	27 999	5 139	45 000
Treap	4 670	98 999	1 975	139 999	923	20 000	875	25 999	863	76 000
B-tree	2 640	10 000	3 066	12 999	3 863	33 999	4 076	29 000	4 670	33 999
Binomiální halda	12 570	62 000	13 452	88 999	12 538	128 999	13 728	453 000	X	X
Fibonacciho halda	26 540	103 999	13 900	137 999	12 025	87 000	12 859	471 999	X	X
Rope	599	5 000	1 797	7 999	14 027	107 999	132 266	608 000	X	X
Skip list	910	5 000	956	6 999	1 401	50 999	1 800	26 999	X	X

Tabulka 6.2: Vkládání do množiny generované pomocí rovnoměrného rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	650	10 999	520	10 000	538	33 999	480	233 999	X	X
Deque	1 450	2 000	3 717	20 999	3 725	31 999	3 247	15 999	3 821	41 999
List	2 210	6 999	20 984	91 000	171 522	630 000	X	X	X	X
Hash table	1 220	10 999	733	67 000	657	460 000	546	3 005 999	535	25 075 000
AVL-tree	4 380	46 000	4 202	20 999	1 740	31 999	1 887	10 000	2 048	29 000
RB-tree	1 550	5 000	1 924	10 999	2 366	101 999	2 461	16 999	2 765	25 000
Splay-tree	2 490	12 000	2 976	81 000	2 970	572 000	3 535	12 734 999	4 211	101 389 999
SG-tree	1 300	6 000	1 388	31 999	1 390	19 000	1 288	13 999	1 288	19 000
Treap	2 410	24 000	1 991	16 999	870	83 999	857	15 999	861	20 000
B-tree	1 610	7 999	1 934	31 999	2 620	34 999	2 906	20 999	3 469	27 999
Binomiální halda	10 520	58 000	11 979	91 000	12 577	128 999	13 641	444 000	X	X
Fibonacciho halda	11 470	40 000	13 054	96 999	12 362	97 999	13 178	464 000	X	X
Rope	340	1 999	344	6 000	384	63 999	408	484 000	X	X
Skip list	1 240	33 999	1 198	6 000	1 383	25 000	1 185	25 999	X	X

Tabulka 6.3: Vkládání do množiny generované pomocí vzestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	959	15 999	6 214	63 999	63 964	153 000	615 151	1 678 000	X	X
Deque	900	3 999	1 572	12 000	2 146	33 000	2 583	16 999	3 184	64 999
List	1 160	6 000	5 818	27 999	51 119	402 999	X	X	X	X
Hash table	1 070	9 000	1 026	87 999	841	349 000	543	2 923 999	549	24 386 999
AVL-tree	1 930	7 999	2 053	10 999	1 710	18 000	1 879	21 999	2 064	26 999
RB-tree	1 510	3 999	1 780	6 999	2 239	26 999	2 391	18 000	2 687	60 000
Splay-tree	1 950	7 999	2 538	62 000	3 428	665 000	3 966	11 353 000	4 751	81 206 000
SG-tree	1 700	9 000	1 442	12 000	1 634	20 999	1 219	13 999	1 230	36 999
Treap	1 000	9 000	872	10 000	930	95 000	879	60 999	909	73 999
B-tree	3 660	12 999	3 935	19 000	2 114	20 999	2 225	25 000	2 276	33 999
Binomiální halda	12 660	76 000	12 776	129 999	12 532	154 999	13 697	436 999	X	X
Fibonacciho halda	12 050	55 999	11 845	78 999	11 797	87 000	12 299	497 000	X	X
Rope	729	1 999	396	7 999	459	96 999	418	485 000	X	X
Skip list	490	9 000	580	9 000	424	23 999	472	33 000	X	X

Tabulka 6.4: Vkládání do množiny generované pomocí sestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	529	2 999	605	16 999	559	12 000	522	15 999	661	20 708 000
Deque	1 440	3 999	2 750	154 000	5 353	13 999	4 424	21 999	6 058	118 409 000
List	1 470	999	14 286	41 999	135 617	436 000	X	X	X	X
Hash table	260	9 000	263	999	283	999	306	9 000	407	19 000
AVL-tree	850	10 999	1 042	12 000	1 251	24 000	1 555	12 999	2 124	27 999
RB-tree	920	78 999	1 029	7 999	1 232	25 999	1 521	12 999	2 112	31 999
Splay-tree	1 160	9 000	1 493	141 000	1 309	15 000	1 677	25 000	2 223	30 000
SG-tree	1 860	31 000	2 459	23 000	2 989	23 000	3 715	19 000	4 795	30 000
Treap	410	5 000	420	1 999	452	15 000	403	6 999	404	21 999
B-tree	1 270	148 999	3 958	15 999	2 196	23 000	2 549	18 000	3 051	29 000
Binomiální halda	1 950	174 999	23 905	154 000	240 432	645 999	2 992 139	9 176 000	X	X
Fibonacciho halda	2 450	999	26 767	207 999	258 142	736 000	3 186 386	8 080 000	X	X
Rope	710	999	1 519	3 000	6 271	45 000	5 593	329 000	X	X
Skip list	670	24 999	812	10 000	901	21 999	1 473	24 000	X	X

Tabulka 6.5: Mazání prvku z množiny generované pomocí normálního rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	529	2 999	549	14 999	560	150 000	523	170 999	556	17 708 000
Deque	1 490	12 000	2 786	10 999	3 331	12 000	3 836	12 999	4 358	118 087 999
List	1 410	3 000	13 930	38 999	135 170	346 000	X	X	X	X
Hash table	250	999	263	999	288	3 000	307	5 000	407	21 999
AVL-tree	1 020	10 999	1 202	24 000	1 218	15 999	1 540	15 000	2 105	26 999
RB-tree	870	1 999	1 037	1 999	1 295	15 999	1 635	18 000	2 097	31 000
Splay-tree	1 220	1 999	1 464	1 999	1 361	121 000	1 719	29 000	2 222	38 999
SG-tree	1 870	3 000	3 114	10 999	3 023	16 999	3 788	30 000	4 807	43 999
Treap	5 110	262 000	397	999	404	3 999	409	15 999	405	20 999
B-tree	1 530	3 000	1 537	3 000	2 023	15 999	2 574	20 999	3 070	30 000
Binomiální halda	2 570	6 999	24 561	82 000	241 956	730 999	3 026 695	8 770 000	X	X
Fibonacciho halda	5 530	19 000	22 267	96 000	259 940	830 000	3 227 799	9 171 999	X	X
Rope	439	999	1 152	7 999	10 448	43 000	100 525	422 000	X	X
Skip list	400	999	567	999	957	23 000	1 554	30 000	X	X

Tabulka 6.6: Mazání prvku z množiny generované pomocí rovnoměrného rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	1 189	5 999	14 940	113 999	30 093	243 000	76 646	747 000	X	X
Deque	1 430	1 999	2 327	5 000	2 545	20 999	2 715	16 999	3 201	15 000
List	1 570	3 000	12 957	34 999	124 896	391 000	X	X	X	X
Hash table	250	999	534	10 999	255	16 999	253	18 000	255	15 000
AVL-tree	2 250	3 000	2 619	3 999	1 124	21 999	1 269	12 000	1 415	23 000
RB-tree	1 020	10 000	1 018	10 999	1 163	15 999	1 280	12 999	1 430	23 000
Splay-tree	1 160	1 999	1 306	10 999	1 147	16 999	1 302	33 999	1 447	34 999
SG-tree	1 290	7 999	1 277	66 000	1 374	679 000	1 286	6 662 000	1 288	66 342 999
Treap	1 040	1 999	870	1 999	399	999	403	6 999	405	18 000
B-tree	1 380	3 000	1 553	3 000	2 157	25 000	2 452	15 999	2 942	26 999
Binomiální halda	1 500	3 999	9 738	58 999	181 816	541 000	1 402 132	4 677 999	X	X
Fibonacciho halda	2 250	5 000	21 948	64 999	136 042	368 000	1 671 190	5 151 000	X	X
Rope	900	1 999	5 493	33 999	24 220	77 999	227 256	687 000	X	X
Skip list	420	999	261	999	255	11 999	199	67 000	X	X

Tabulka 6.7: Mazání prvku z množiny generované pomocí vzestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	320	999	324	999	379	3 999	425	6 000	X	X
Deque	840	1 999	2 137	12 000	2 381	10 000	2 976	15 000	3 494	53 000
List	2 260	12 999	14 718	36 000	147 869	400 000	X	X	X	X
Hash table	410	999	652	999	246	20 000	250	9 000	260	12 999
AVL-tree	1 110	3 000	1 314	1 999	1 291	20 000	1 272	21 999	1 435	36 000
RB-tree	890	999	991	1 999	1 157	20 999	1 273	15 999	1 423	20 999
Splay-tree	910	1 999	987	1 999	1 137	19 000	1 260	7 999	1 414	24 000
SG-tree	1 900	16 999	1 457	83 999	1 481	1 183 999	1 445	8 267 000	1 436	83 294 000
Treap	410	999	407	999	404	6 000	424	18 000	443	25 000
B-tree	2 440	5 000	3 247	36 000	2 442	15 999	2 872	27 999	3 330	36 999
Binomiální halda	2 690	6 000	31 451	82 999	329 214	824 999	4 130 044	10 160 999	X	X
Fibonacciho halda	3 060	12 000	32 324	96 000	328 695	815 999	4 004 848	9 025 999	X	X
Rope	599	999	2 327	4 999	21 763	69 000	202 963	663 000	X	X
Skip list	1 010	5 000	8 228	33 000	87 775	282 999	1 027 758	3 260 999	X	X

Tabulka 6.8: Mazání prvku z množiny generované pomocí sestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	441	6 999	454	12 000	459	221 000	520	14 000	508	125 000
Deque	1 070	3 999	3 237	31 999	2 383	16 999	3 549	22 999	3 841	28 000
List	1 390	6 000	14 386	124 000	149 639	665 000	X	X	X	X
Hash table	320	6 999	299	50 999	354	1 999	331	12 000	437	20 000
AVL-tree	1 860	6 000	2 596	25 000	2 367	13 999	2 803	24 000	3 600	30 000
RB-tree	1 950	5 000	2 026	21 999	2 320	15 999	2 761	26 999	3 605	29 000
Splay-tree	3 540	9 000	3 084	15 999	2 503	732 000	2 939	10 699 999	4 015	221 959 999
SG-tree	2 890	6 999	3 671	9 000	4 599	15 999	5 628	24 000	7 092	34 999
Treap	1 030	9 000	664	3 999	662	9 000	644	6 000	644	12 000
B-tree	1 930	6 999	2 513	16 999	2 867	15 000	3 171	15 000	3 680	29 000
Binomiální halda	28 170	48 999	35 584	148 999	91 914	1 062 999	717 928	9 107 999	X	X
Fibonacciho halda	48 790	92 000	39 677	106 999	114 428	885 999	725 322	11 232 000	X	X
Rope	430	1 999	1 018	33 999	6 153	26 999	5 000	10 999	X	X
Skip list	369	1 000	782	16 999	1 042	21 999	1 334	22 999	X	X

Tabulka 6.9: Vyhledání prvku ve množině generované pomocí normálního rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	770	12 000	444	13 999	467	174 000	519	680 000	726	1 622 000
Deque	1 670	30 000	4 014	171 000	3 436	1 183 999	3 931	11 873 999	3 836	28 000
List	1 410	3 999	14 523	62 000	150 806	666 999	X	X	X	X
Hash table	330	999	298	999	306	25 999	335	15 999	429	18 000
AVL-tree	2 040	10 999	3 803	25 999	2 373	15 999	2 895	27 999	3 603	26 999
RB-tree	1 900	7 999	2 061	3 000	2 323	24 000	2 781	16 999	3 610	31 000
Splay-tree	2 620	15 000	2 918	90 000	2 502	754 000	2 935	10 019 999	4 018	222 318 999
SG-tree	2 790	10 000	5 000	295 000	4 652	18 000	5 790	30 000	7 098	50 999
Treap	2 590	31 999	648	3 000	648	5 000	646	12 999	650	18 000
B-tree	2 500	6 999	2 424	31 000	2 795	15 000	3 230	67 999	3 694	25 000
Binomiální halda	31 170	95 000	30 138	143 000	90 571	1 085 999	754 643	11 173 999	X	X
Fibonacciho halda	33 920	178 999	41 288	166 999	95 641	630 000	707 598	8 554 000	X	X
Rope	310	999	382	999	448	6 000	519	8 999	X	X
Skip list	440	999	583	999	1 043	26 000	1 456	20 999	X	X

Tabulka 6.10: Vyhledání prvku ve množině generované pomocí rovnoměrného rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	310	999	330	999	390	16 999	427	23 000	X	X
Deque	1 630	1 999	1 769	3 999	2 086	12 999	2 652	12 000	3 468	21 999
List	1 060	25 000	8 664	62 999	88 862	657 999	X	X	X	X
Hash table	230	999	273	999	313	6 000	277	10 999	278	15 000
AVL-tree	4 430	20 999	2 812	51 999	1 981	12 999	2 197	13 999	2 392	24 000
RB-tree	2 260	12 000	1 902	10 000	2 096	18 000	2 356	23 000	2 605	15 000
Splay-tree	2 480	13 999	2 494	81 000	2 001	659 999	2 231	7 056 999	2 348	76 744 999
SG-tree	1 890	19 000	2 012	60 999	1 912	586 999	1 945	6 225 999	1 962	66 116 999
Treap	1 930	20 999	1 598	15 000	643	3 999	644	9 000	647	23 000
B-tree	2 160	6 000	2 013	6 000	2 065	16 999	2 086	25 999	2 121	21 999
Binomiální halda	7 440	48 000	8 749	116 000	31 156	214 999	27 869	339 000	X	X
Fibonacciho halda	14 840	105 000	9 492	92 999	31 273	167 999	39 238	373 999	X	X
Rope	520	999	537	999	554	999	573	6 000	X	X
Skip list	480	999	648	2 999	871	26 999	822	19 999	X	X

Tabulka 6.11: Vyhledání prvku ve množině generované pomocí vzestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	300	999	327	999	362	6 000	425	5 000	X	X
Deque	870	1 999	2 528	12 000	2 465	12 000	3 048	20 999	3 575	69 999
List	3 230	9 000	20 810	62 999	211 085	662 999	X	X	X	X
Hash table	620	9 000	761	10 000	273	999	277	13 999	282	25 999
AVL-tree	2 320	15 000	2 198	3 999	2 120	15 999	2 373	27 999	2 612	26 999
RB-tree	1 680	9 000	1 931	3 000	2 211	13 999	2 532	19 000	2 845	24 000
Splay-tree	1 830	7 999	1 861	67 999	2 015	657 000	2 216	7 222 000	2 412	79 310 000
SG-tree	2 630	38 000	2 145	76 000	2 056	739 999	2 081	7 562 000	2 102	77 102 000
Treap	710	7 999	1 245	3 000	642	5 000	647	15 999	662	20 999
B-tree	3 110	6 999	1 880	3 999	2 766	13 999	2 894	18 000	3 507	27 999
Binomiální halda	24 760	112 999	32 048	167 999	88 745	1 244 000	767 869	12 687 000	X	X
Fibonacciho halda	23 810	77 000	41 120	183 000	122 041	1 170 000	903 528	13 098 999	X	X
Rope	310	999	376	2 000	474	1 999	492	6 999	X	X
Skip list	1 549	31 000	9 738	90 000	88 788	368 000	103 842	403 000	X	X

Tabulka 6.12: Vyhledání prvku ve množině generované pomocí sestupné sekvence.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	210	999	197	999	204	10 000	194	9 000	190	6 000
Deque	290	999	275	3 000	291	3 000	293	9 000	289	6000
List	860	3 000	6 213	40 000	60 622	154 000	X	X	X	X
Rope	200	1 999	212	999	202	3 000	212	7 999	X	X

Tabulka 6.13:
Přístup k prvku na určité pozici (indexu) v množině generované pomocí normálního rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	190	999	234	999	194	5 000	194	3 000	191	18 000
Deque	260	999	314	19 000	294	6 999	291	10 999	309	21 000
List	840	1 999	6 242	20 999	60 550	178 999	X	X	X	X
Rope	240	999	207	999	199	999	210	6 000	X	X

Tabulka 6.14:
Přístup k prvku na určité pozici (indexu) v množině generované pomocí rovnoměrného rozložení pravděpodobnosti.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	180	999	192	999	214	999	193	10 999	X	X
Deque	420	999	482	999	297	3 000	293	3 000	286	6 999
List	880	1 999	6 448	33 000	60 316	147 999	X	X	X	X
Rope	220	999	217	999	221	6 999	211	9 000	X	X

Tabulka 6.15:
Přístup k prvku na určité pozici (indexu) v množině tvořené vzestupnou sekvencí.

	100		1 000		10 000		100 000		1 000 000	
	avg	worst	avg	worst	avg	worst	avg	worst	avg	worst
Vector	240	999	195	999	192	999	193	10 000	X	X
Deque	260	999	287	999	287	999	289	6 999	359	53 000
List	1 270	3 000	6 302	20 000	60 422	151 000	X	X	X	X
Rope	230	999	201	999	230	6 000	207	7 999	X	X

Tabulka 6.16:
Přístup k prvku na určité pozici (indexu) v množině tvořené sestupnou sekvencí.

Kapitola 7

Vyhodnocení

V této kapitole se nachází stručné vyhodnocení výkonu struktur v testu podle tabulek v přechozí kapitole 6.

7.1 Vector

Vector si oproti předpokladům v testu vedl velmi dobře. V mnoha případech skončil nedaleko za hashovací tabulkou. Výkonostní propady nastaly při vkládání prvků sestupné sekvence a při mazání prvků vzestupné sekvence (vždy přístup k prvnímu prvku a posun všech prvků o jedno pole).

7.2 Deque

Deque podávala překvapivě výrazně horší výsledky, než vector. Na druhou stranu u ní nedocházelo k žádným výkonnostním propadům. Indexování jen o málo horší, než Vector.

7.3 List

List se podle předpokladů v testu ukázal jako velmi neefektivní. Všechny testované kontejnery (s vyjímkou hald a výkonnostních propadů) podávaly lepší výkony ve všech testovaných operacích.

7.4 Hash table

Zdaleka nejlepší výsledky v průměrném případě. V některých případech se objevili situace, kdy provedení operace zabralo velký časový úsek. Při vkládání dat došlo ve dvou případech k situaci, že vložení jednoho prvku zabralo více, než 10 % času potřebného k provedení 1 000 000 operací viz údaje v tabulkách 6.1 a 6.2.

7.5 AVL-tree

Vyrovnané výkony s ostatními stromovými kontejnery. Lepší práce se sekvencemi, než s náhodně generovanými množinami.

7.6 RB-tree

Vyrovnané výkony s ostatními stromovými kontejnery.

7.7 Splay-tree

V testu pomalá (zbůsobeno tím, že navržený test tomuto kontejneru příliš nevyhovuje, kdyby bylo prováděno více operací hned za sebou nad jedním prvkem, tak by si vedl lépe). Nejdelší v testu naměřená délka provedení jedné operace viz tabulka 6.10

7.8 SG-tree

Nejpomalejší ze stromových kontejnerů v testu. Výrazně horší práce s náhodně generovanými množinami. Oproti ostatním stromům pomalé vyhledávání.

7.9 Treap

Velmi dobré výsledky. Výkonostní propad u malých množin generovaných pomocí rovnoměrného rozložení pravděpodobnosti. S nárůstem velikosti množin se zkracuje průměrná doba na provedení všech operací.

7.10 B-tree

Vkládání a mazání prvku pomalejší, než většina stromových kontejnerů. Vyhledání je přibližně stejné.

7.11 Binomiální a Fibonacciho halda

Velmi špatný výkon. I v předpokládaných silných oblastech (mazání vzestupné sekvence, což odpovídá práci s minimem).

7.12 Rope

Dobré vyhledání prvku. Obecně dobrý výkon u nejmenších testovaných množin. U větších množin ztrácí a nevyplatí se. Velmi dobrá doba přístupu k prvku přes index (srovnatelná s Vectorem).

7.13 Skip list

Třetí nejrychlejší kontejner v testu (po Hash table a Treap). Výkonostní propad při vyhledávání a mazání prvků sestupné sekvence.

Kapitola 8

Závěr

Cíl práce byl definován tak, že by jako celek měla čtenáři poskytnout informace usnadňující volbu datového kontejneru vhodného na řešení jeho problému.

Jako prostředek k dosažení tohoto cíle byly ve stručnosti představeny matematické nástroje pro vyjádření výkonnosti kontejneru. Jedná se o asymptotickou časovou analýzu, která říká jak nejhůře může dopadnout operace prováděná nad kontejnerem. Na tomto přístupu je založena většina stromových struktur, které dosahují poměrně dobré - logaritmické - časové složitosti pro většinu operací. Amortizovanou časovou analýzu, která se týká nejhoršího možného provedení stanovené sekvence operací. Příkladem amortizované datové struktury je například Vector, který při potřebě alokace paměti vždy alokuje prostor navíc, aby sekvence vkládání více prvků probíhala rychleji. A časovou složitost v průměrném případě u které zanedbáváme, že některé provedení operace může být mnohem delší. Zajímá nás totiž jak se daný kontejner bude chovat pro drtivou většinu případů. Příkladem struktury, jež je na tomto principu založena je hashovaná tabulka.

Dále byla představeny skupiny datových struktur včetně jejich popisu, teoretických vlastností a případů, kdy je vhodné je použít. Jedná se o datové kontejnery ze standardní knihovny C^{++} a knihovny boost::intrusive. A nestandardní kontejnery B-tree, Binomiální a Finacciho haldu, Rope (používá se zejména na práci s řetězci) a Skip list. Každá z těchto struktur byla popsána včetně údajů o existujících modifikacích, teoretických složitostech jejich operací a obvyklému způsobu využití. Od každé byla vybrána konkrétní varianta dále použitá v testech a popsána její vnitřní implementace.

Jenoduchý testovací nástroj byl navržen tak, aby porovnal výše zmíněné kontejnery, především v závislosti na jejich chování v průměrném případě. Výsledky testování byly uspořádány do tabulek a nejdůležitější informace z nich vyplývající zmíněny ve vyhodnocení.

Jak již bylo zmíněno práce je zaměřena především na zkoumání chování kontejnerů v průměrném případě. Kromě toho obsahuje i údaj o nejdelším provedení operace, který by měl odpovídat složitosti asymptotické. Vhodným pokračováním by bylo rozšířit sadu testů tak, aby byla prověřena i amortizovaná časová složitost.

Literatura

[1] Borodin, A.; El-Yaniv, R.: *Online Computation and Competitive Analysis*. Cambridge University Press, 1998, iSBN 0-521-56392-5.

[2] Horowitz, E.; Sahni, S.; Mehta, D. P.: *Fundamentals in Data Structures in C++*. Silicon Press, 2007, iSBN 9780929306377.

[3] Knuth, D. E.: *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN 0-201-89684-2.

[4] Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM*, ročník 33, č. 6, Červen 1990: s. 668–676, ISSN 0001-0782, doi:10.1145/78973.78977. URL http://doi.acm.org/10.1145/78973.78977

[5] WWW stránky: Big O notation. `http://en.wikipedia.org/wiki/Big_O_notation`.

[6] WWW stránky: Boost.Intrusive. `http://www.boost.org/doc/libs/1_58_0/doc/html/intrusive.html`. URL http://www.boost.org/doc/libs/1_58_0/doc/html/intrusive.html

[7] WWW stránky: cpp-btree. `http://code.google.com/p/cpp-btree/`. URL http://code.google.com/p/cpp-btree/