UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Vizualizace evoluce algoritmů pracujících nad vybranými datovými strukturami

Bc. Martin Šára

Diplomová práce
2014

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2013/2014**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martin Šára**
Osobní číslo:	**I12500**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vizualizace evoluce algoritmů pracujících nad vybranými datovými strukturami**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části práce je nutné provést přehled problematiky vybraných implementací abstraktního datového typu tabulka a prioritní fronta.
Primárním cílem diplomové práce je realizace vizualizací evolucí vybraných algoritmů nad následujícími datovými strukturami: binomická halda (binomial heap), Huffmanův strom (Huffman tree), B-strom (B-tree) a grid soubor (grid file).
Zmíněné vizualizace budou realizovány v rámci webové aplikace.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam odborné literatury:

SAMET, H. Foundations of Multidimensional and Metric Data Structures, San Francisco (CA), Morgan Kaufmann Publishers, 2006.
CORMEN, H. A KOL. Introduction to algorithms. Boston, MIT Press, 2001.
LEWIS, H. R., DENENBERG, L. Data structures and their algorithms. Berkley, Adison-Wesley, 1997.
GOODRICH, M.T., TAMASSIA, R. Algorithm Design. Hoboken (NJ), John Wiley & Sons, 2002.

Vedoucí diplomové práce: **prof. Ing. Antonín Kavička, Ph.D.**
Katedra softwarových technologií

Datum zadání diplomové práce: **31. října 2013**
Termín odevzdání diplomové práce: **16. května 2014**

Univerzita Pardubice
Česká republika

L.S.

prof. Ing. Simeon Karamazov, Dr.
děkan

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2013

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 14. 5. 2014 Bc. Martin Šára

Poděkování

Na tomto místě bych rád poděkoval všem, kteří mě při vypracovávání této práce podporovali. Zejména bych velmi rád poděkoval vedoucímu mé diplomové práce prof. Ing. Antonínu Kavičkovi, Ph.D za jeho cenné rady, připomínky a věnovaný čas.

Anotace

Diplomová práce se zabývá vizualizacemi datových struktur a jejich algoritmů. Konkrétně se jedná o vizualizace binomické haldy, Huffmanova stromu, B-stromu a grid souboru. V úvodní části práce je nejprve proveden přehled existujících vizualizací. V teoretické části práce jsou popsány abstraktní datové typy prioritní fronta a tabulka a problematika kódování textu. Teoretická část také obsahuje teoretický popis vybraných datových struktur. V praktické části práce je popsáno fungování jednotlivých algoritmů vybraných datových struktur. Hlavním výstupem práce jsou vizualizace čtyř vybraných datových struktur, které mohou být prováděny i ve webovém prohlížeči.

Klíčová slova

datové struktury, vizualizace, animace, binomická halda, Huffmanův strom, Huffmanovo kódování, B-strom, grid soubor

Title

Visualization of algorithm evolution working on selected data structures.

Annotation

This thesis deals with visualizations of data structures and their algorithms. The concrete structures are: binomial heap, Huffman tree, B-tree and grid file. At the beginning there is review of existing visualizations. The theoretical part describes abstract data type priority queue and dictionary and the problem of text encoding. There is also theoretical description of the selected data structures. The practical part describes working of each algorithm working on each selected data structure. The main output of this thesis is visualization of four selected data structures. These visualizations can be displayed in a web browser.

Keywords

data structures, visualization, animation, binomial heap, Huffman tree, Huffman coding, B-tree, grid file

Obsah

Seznam zkratek ... **10**
Seznam obrázků ... **11**
Seznam tabulek ... **12**
Úvod ... **13**
1.1 Cíle práce ... 14
1.2 Struktura práce ... 14
2 Přehled problematiky ... **15**
2.1 Webová aplikace Data Structure Visualizations ... 15
2.2 Webová aplikace Gnarley trees ... 16
2.3 Webová aplikace CSILM ... 17
2.4 Další zdroje ... 18
2.5 Shrnutí ... 20
3 ADT prioritní fronta ... **22**
3.1 Operace ... 22
3.1.1 Základní operace ... 23
3.1.2 Rozšiřující operace ... 23
3.2 Binomická halda ... 23
3.2.1 Binomický strom ... 24
3.2.2 Binomický les ... 25
3.2.3 Definice ... 25
3.2.4 Operace ... 26
4 Kódování textu ... **28**
4.1 Metody ... 29
4.2 Huffmanovo kódování ... 30
4.2.1 Huffmanův strom ... 31
4.2.2 Dvoufázové Huffmanovo kódování ... 32
4.2.3 Statické Huffmanovo kódování ... 33
4.2.4 Adaptivní Huffmanovo kódování ... 33
5 ADT tabulka ... **34**
5.1 Operace ... 35
5.2 B-strom ... 35

5.2.1 Parametrizace ... 36
5.2.2 Varianty ... 37
5.2.3 Operace ... 39
5.2.4 Štěpení a fúze uzlů ... 39
5.3 Grid soubor ... 40
5.3.1 Adresář ... 41
5.3.2 Stupnice ... 42
5.3.3 Operace ... 43
5.3.4 Strategie dělení adresáře a stupnic ... 43
5.3.5 Strategie slučování datových bloků ... 44
6 Implementace ... 45
6.1 Výběr technologie ... 45
6.1.1 Požadavky ... 45
6.1.2 Analyzované technologie ... 46
6.1.3 JavaFX ... 47
6.2 Implementace vizualizací ... 48
6.2.1 Filozofie ... 49
6.2.2 Koncept ... 49
6.2.3 Obecné funkcionality ... 51
7 Implementace binomické haldy ... 52
7.1 Paměťová reprezentace ... 54
7.2 Operace Link ... 55
7.3 Operace Merge ... 55
7.4 Operace Sjednocení ... 56
7.5 Operace Vlož ... 57
7.6 Operace Zpřístupni maximum ... 58
7.7 Operace Odeber maximum ... 58
7.8 Operace Zvyš prioritu ... 59
7.9 Operace Odeber ... 60
7.10 Grafické rozmístění prvků ... 60
8 Implementace Huffmanova stromu ... 62
8.1 Budování stromu ... 63
8.2 Tvorba kódů ... 64

8.3 Kódování ... 65
8.4 Dekódování ... 65
8.5 Grafické rozmístění prvků ... 66
9 Implementace B-stromu ... 68
9.1 Operace Vlož ... 69
9.2 Operace Najdi ... 72
9.3 Operace Odeber ... 73
9.4 Grafické rozmístění prvků ... 80
10 Implementace grid souboru ... 82
10.1 Operace Vlož ... 84
10.2 Operace Najdi ... 87
10.3 Operace Intervalové vyhledávání ... 88
10.4 Operace Odeber ... 89
Závěr ... 92
Literatura ... 93
Příloha A – Uživatelský manuál ... 95
Příloha B – UML diagram vizualizace B-stromu ... 102
Příloha C – Obsah přiloženého CD ... 103

Seznam zkratek

ADT	Abstract Data Type – abstraktní datový typ
UML	Unified Modeling Language – unifikovaný modelovací jazyk
ASCII	American Standard Code for Information Interchange
OS	Operating system – operační systém
IT	Information technology – informační technologie
HTML	HyperText Markup Language
OOP	Object-oriented programming – objektově orientované programování
JS	JavaScript
JDK	Java Development Kit
JRE	Java Runtime Environment
GUI	Graphical User Interface – grafické uživatelské rozhraní
CSS	Cascading Style Sheets – kaskádové styly
KiB	Kibibyte – předpona kibi znamená násobek 1024

Seznam obrázků

Obrázek 1 – Data Structure Visualizations – Java aplikace 15
Obrázek 2 – Data Structure Visualizations – Webový prohlížeč (JavaScript) 16
Obrázek 3 – Gnarley trees – Webový prohlížeč (Java applet) 17
Obrázek 4 – CSILM – Java applet 18
Obrázek 5 – OpenDSA – Webový prohlížeč 19
Obrázek 6 – JHAVÉ – Java aplikace 20
Obrázek 7 – Princip zásobníku a fronty 22
Obrázek 8 – Binomický strom B_0, B_1, B_2, B_3 a B_4 24
Obrázek 9 – Vznik binomického stromu řádu 3 24
Obrázek 10 – Příklad binomické min-haldy 26
Obrázek 11 – Příklad binomické max-haldy 26
Obrázek 12 – Prefixy kódů 31
Obrázek 13 – Kódovací stromy korespondující datům z tabulky 3 32
Obrázek 14 – B-strom 36
Obrázek 15 – B^+-strom 38
Obrázek 16 – B^*-strom 39
Obrázek 17 – Struktura grid souboru 41
Obrázek 18 – Grid adresář 42
Obrázek 19 – Stupnice grid souboru 42
Obrázek 20 – Systémy slučování datových bloků v grid souboru 44
Obrázek 21 – Dostupnost technologie Java ve webových prohlížečích 48
Obrázek 22 – UML diagram tříd konceptu animací 50
Obrázek 23 – UML diagram tříd binomické haldy 53
Obrázek 24 – Ilustrace paměťové reprezentace binomické haldy 54
Obrázek 25 – Binomická halda – ilustrace operace Link 55
Obrázek 26 – Binomická halda – ilustrace operace Merge 56
Obrázek 27 – Binomická halda – ilustrace operace Link v rámci operace Sjednocení 57
Obrázek 28 – Binomická halda před vložením prvku p20 57
Obrázek 29 – Binomická halda po vložení prvku p20 58
Obrázek 30 – Odebrání maxima z binomické haldy 58
Obrázek 31 – Binomická halda po odebrání maxima 59
Obrázek 32 – Binomická halda před zvýšením priority prvku 59
Obrázek 33 – Binomická halda po zvýšení priority prvku 60
Obrázek 34 – Zjednodušený UML diagram třídy BinHeapNode 61
Obrázek 35 – UML diagram tříd Huffmanova stromu 62
Obrázek 36 – Postup budování Huffmanova stromu 64
Obrázek 37 – Tvorba kódů pomocí Huffmanova stromu 65
Obrázek 38 – Průběh kódování vstupního textu pomocí Huffmanova stromu 65
Obrázek 39 – Dekódování textu pomocí Huffmanova stromu 66
Obrázek 40 – UML diagram tříd grafických prvků Huffmanova stromu 67
Obrázek 41 – UML diagram tříd B-stromu 68

Obrázek 42 – B-strom – štěpení kořene ... 69
Obrázek 43 – B-strom – vkládání prvku s klíčem 44 ... 70
Obrázek 44 – B-strom – štěpení uzlu ... 71
Obrázek 45 – B-strom – hledání prvku s klíčem 72 ... 72
Obrázek 46 – B-strom – před odebráním prvku s klíčem 57 ... 74
Obrázek 47 – B-strom – po odebrání prvku s klíčem 57 ... 74
Obrázek 48 – B-strom – před odebráním prvku s klíčem 15 ... 75
Obrázek 49 – B-strom – po odebrání prvku s klíčem 15 ... 75
Obrázek 50 – B-strom – před odebráním prvku s klíčem 15 (sloučení uzlů) ... 76
Obrázek 51 – B-strom – odebrání prvku s klíčem 15 – sloučení uzlů ... 76
Obrázek 52 – B-strom – po odebrání prvku s klíčem 15 (sloučení uzlů) ... 77
Obrázek 53 – B-strom – před přesunutím klíče z levého souseda ... 77
Obrázek 54 – B-strom – po přesunutí klíče z levého souseda ... 78
Obrázek 55 – B-strom – před sloučením uzlů ... 79
Obrázek 56 – B-strom – po sloučení uzlů ... 79
Obrázek 57 – Princip zarovnávání uzlů B-stromu ... 80
Obrázek 58 – UML diagram tříd grafické části B-stromu ... 81
Obrázek 59 – Příklad příslušnosti prvku do grid buňky ... 82
Obrázek 60 – UML diagram tříd grid souboru ... 83
Obrázek 61 – Grid soubor – vložení záznamu do prázdného bloku ... 84
Obrázek 62 – Grid soubor – před vložením záznamu do plného bloku ... 85
Obrázek 63 – Grid soubor – po vložení záznamu do plného bloku ... 86
Obrázek 64 – Grid soubor – vyhledání prvku ... 88
Obrázek 65 – Grid soubor – před odebráním prvku ... 90
Obrázek 66 – Grid soubor – po odebrání prvku ... 91
Obrázek 67 – Java nastavení ... 95
Obrázek 68 – Bezpečnostní varování ... 96
Obrázek 69 – Dialog pro dávkové načítání dat ... 97
Obrázek 70 – GUI – binomická halda ... 98
Obrázek 71 – GUI – Huffmanův strom ... 98
Obrázek 72 – GUI – B-strom ... 99
Obrázek 73 – GUI – grid soubor ... 100
Obrázek 74 – GUI – vizualizace uložených dat v grid souboru ... 101

Seznam tabulek

Tabulka 1 – Přehled asymptotických složitostí operací binomické haldy ... 27
Tabulka 2 – Nejfrekventovanější znaky českého textu ... 29
Tabulka 3 – Celkové počty bitů při konstantní a proměnné délce kódu ... 30

Úvod

Problematika datových struktur patří mezi velmi důležité oblasti informačních technologií. Jedná se o základní stavební kameny, bez kterých by žádné softwarové dílo nemohlo existovat. I kdyby vyvíjený softwarový systém přímo nevyužíval datové struktury, stejně by bez nich prakticky nemohl fungovat. Ať už se jedná o jakoukoli aplikaci, vždy potřebujeme na té nejnižší úrovni nějakým způsobem spravovat operační paměť, procesy atd. Tuto správu nelze realizovat jinak, než s využitím datových struktur, byť těch elementárních.

Bez datových struktur se tedy v oblasti informačních technologií vůbec neobejdeme. Důležitost datových struktur přitom netkví pouze v tom, že se využívají téměř všude. Podstatným faktem je oblast výkonu. Správná volba vhodné datové struktury pro uchování dat zásadně ovlivňuje výkon celé aplikace. Můžeme vyvinout sebelepší aplikaci, ale pokud bude využívat nevhodně zvolené datové struktury, bude její výkon zejména s přibývajícími daty postupně degradovat. Triviálním příkladem může být např. ukládání informací o objednávkách do obyčejného seznamu, namísto do některého vyhledávacího stromu. Při vývoji aplikace budeme ukládat desítky, maximálně stovky záznamů. Vše bude fungovat plynule a bez problémů. Ovšem po nasazení do reálného provozu a postupném zaplňování daty nastane chvíle, kdy přestane být aplikace použitelná.

Ačkoli je volba vhodné datové struktury takto významná, najde se řada vývojářů, kteří nejsou s problematikou datových struktur blíže seznámeni. Ti pak mohou ve svých aplikacích využívat nevhodně zvolené datové struktury, což může vést k nižšímu výkonu aplikace. Příkladem může být volba konkrétní implementace seznamu. Je velký rozdíl mezi dynamickou implementací (s explicitními referencemi) a implementací na poli. Obě implementace jsou vhodné k různým užitím. Pokud například předpokládáme přístup do seznamu pomocí indexů, je nevhodné zvolit dynamickou implementaci.

Z výše uvedeného plyne, že nestačí pouze perfektně ovládat programovací jazyky, abychom byli dobří vývojáři. Nutný je také dobrý přehled v oblasti datových struktur, jen pak můžeme vytvářet opravdu kvalitní software. Ať už se jedná o střední nebo vysoké školy, kde se vyučuje programování, je nutné dbát i na adekvátní výuku datových struktur a jejich algoritmů.

Smyslem této práce je právě podpora výuky datových struktur, zejména pak těch netriviálních. Práce je součástí projektu, jehož konečným výstupem bude databáze vizualizací vyučovaných datových struktur na naší fakultě. Všechny vizualizace budou prováděny přímo ve webovém prohlížeči a budou dostupné prostřednictvím přehledné webové aplikace. Tento rok jsou vypracovávány celkem tři práce, jejichž výstupem by mělo být 14 vizualizací a zmiňovaná webová aplikace, do které budou vizualizace umísťovány. V následujících letech budou dotvářeny další vizualizace, dokud nebude databáze kompletní.

1.1 Cíle práce

Tato práce je částí týmového projektu, jehož výstupem bude ucelená databáze vizualizací vyučovaných datových struktur. Cílem této práce, vedle dalších prací v projektu, je tedy vytvořit interaktivní podporu pro výuku vybraných datových struktur. Výstupem budou vizualizace evolucí algoritmů prováděných nad vybranými datovými strukturami. Konkrétně se jedná o vizualizace těchto struktur:

- binomická halda,
- Huffmanův strom,
- B-strom a
- grid soubor.

Vytvořené vizualizace budou vhodně doplňovat především výuku předmětu „Datové struktury a algoritmy“ v navazujícím magisterském studiu. V rámci studia předmětu jsou studenti seznamováni s jednotlivými strukturami. Součástí výkladu jsou samozřejmě i grafická znázornění evolucí jednotlivých algoritmů nad danými strukturami. Po probrání každé struktury budou mít studenti k dispozici interaktivní vizualizace, kde si budou moci vyzkoušet chování dané datové struktury s libovolnými daty.

Vizualizace budou prováděny přímo ve webovém prohlížeči. Dostupné budou prostřednictvím webové aplikace. Do této aplikace bude snadné přidávat další a další vizualizace. Součástí každé vizualizace bude i doprovodný text. Všechny budou mít podobný vzhled a ovládání. Zmiňovaná webová aplikace bude umístěna na veřejně dostupném webu, připadá v úvahu tedy i využití jinými technicky zaměřenými fakultami či širší veřejností se zájmem o problematiku datových struktur.

Shrneme-li cíle práce do bodů, pak to jsou tyto:

- podpora výuky datových struktur – didaktické účely,
- přehled existujících vizualizací a zjištění případných nedostatků,
- vizualizace evolucí algoritmů vybraných netriviálních datových struktur,
- přispění vytvořenými vizualizacemi do databáze vizualizací datových struktur.

Cílem této práce není vytvořit webovou aplikaci, do které budou jednotlivé vizualizace nahrávány. Cílem je pouze tyto vizualizace do aplikace dodat.

1.2 Struktura práce

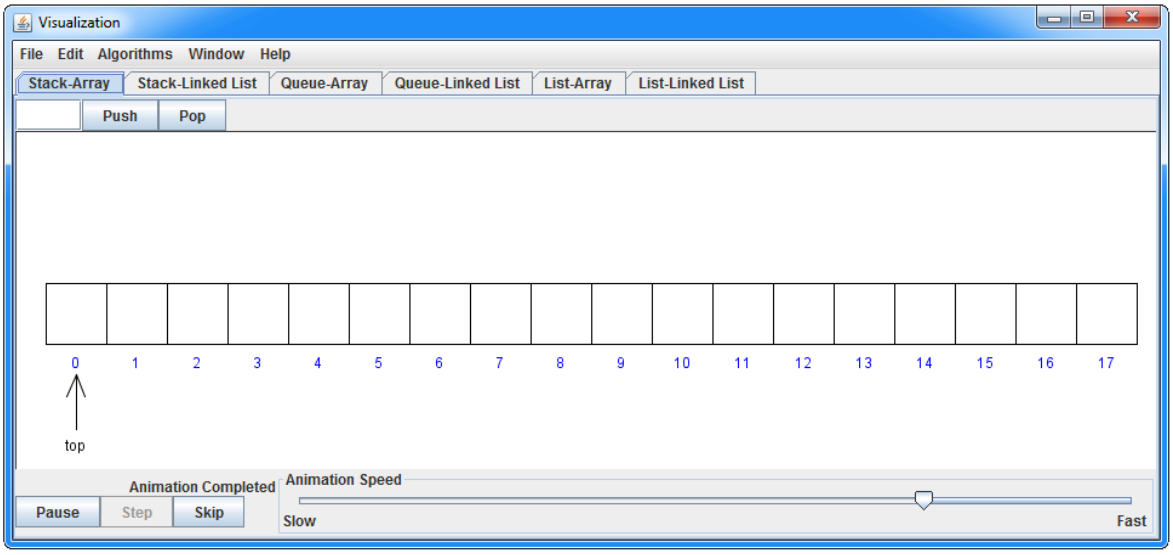
Práce se skládá celkem ze tří částí. První část reprezentuje kapitola 2, v níž je provedena analýza existujících vizualizací datových struktur. Druhou část pokrývají kapitoly 3–5, ve kterých jsou postupně představeny jednotlivé abstraktní datové typy a vybrané datové struktury. Tato část se zabývá pouze teoretickými principy. Kapitoly 6–10 představují implementační část práce. Nejprve je zvolena implementační technologie a obecně představena implementace animací. Následuje představení fungování jednotlivých algoritmů vybraných datových struktur a některých dalších implementačních detailů.

2 Přehled problematiky

Na Internetu můžeme nalézt celou řadu vizualizací nejrůznějších datových struktur. Může se jednat o větší databáze vlastněné technicky zaměřenými fakultami, o menší databáze vytvořené v rámci projektů, či o zcela samostatné vizualizace vypracované jednotlivci. Některé vizualizace běží přímo uvnitř webového prohlížeče, jiné jsou dostupné ke stažení. V následujících kapitolách budou podrobněji popsány vybrané databáze vizualizací datových struktur. Nakonec budou zmíněny i jiné zajímavé materiály podporující výuku datových struktur a algoritmů.

2.1 Webová aplikace Data Structure Visualizations

Snad nejrozsáhlejší ucelená databáze vizualizací je vlastněná Univerzitou San Francisco[1]. Nachází se zde vizualizace různých implementací seznamů, front a zásobníků, dále vyhledávacích stromů, hashovacích tabulek, řadících algoritmů, prioritních front, různých grafových algoritmů atd. Vizualizace jsou realizovány přímo ve webovém prohlížeči pomocí JavaScriptu (obrázek 2). Některé vizualizace jsou ovšem dostupné pouze v Java aplikaci (obrázek 1). Ta však již není nadále aktualizována.



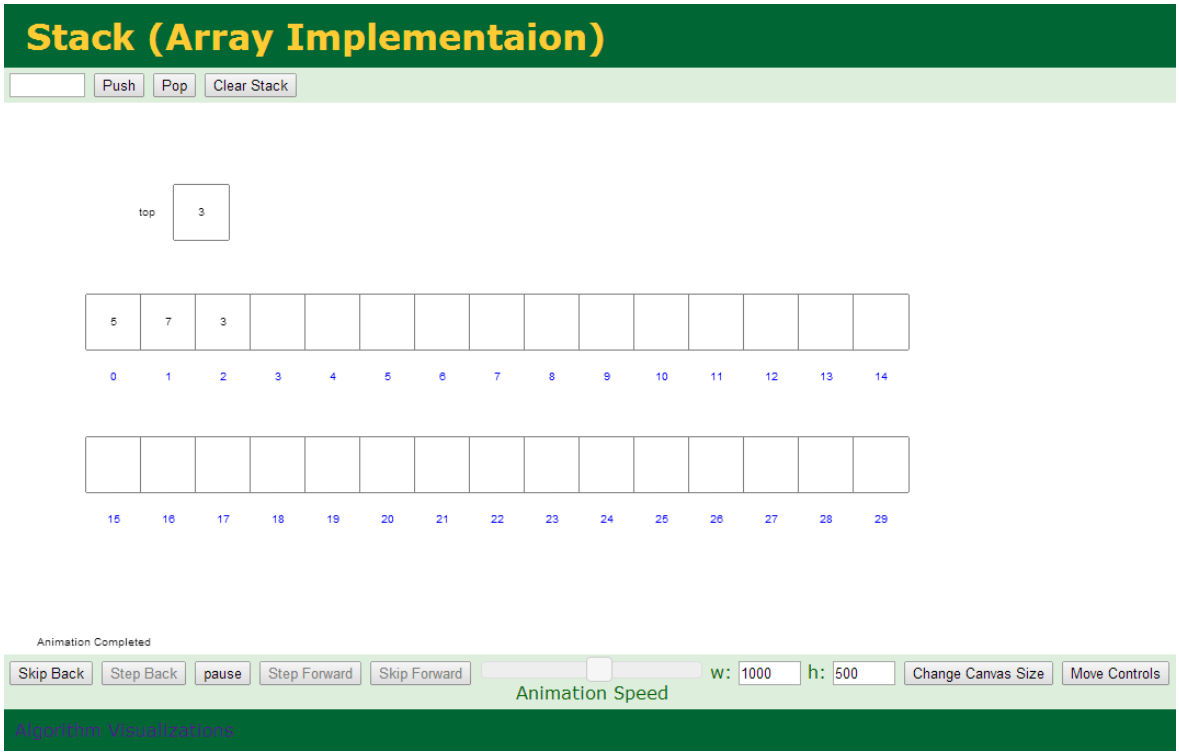
Obrázek 1 – Data Structure Visualizations – Java aplikace

Zdroj: http://www.cs.usfca.edu/~galles/visualization/java/visualization.jar

Vizualizace všech datových struktur disponují stejným vzhledem a podobným ovládáním. Přívětivost uživatelského rozhraní by však mohla být lepší – po každé operaci je nutné klikat do vstupního pole, absentují jakékoli klávesové zkratky. Součástí těchto struktur nejsou žádné přednastavené instance dat, ani možnost generovat data náhodná. Vizualizace jsou plynule animovány, avšak animace lze pozastavovat pouze v krajních bodech – až po provedení určitých úkonů. Rychlost probíhající animace lze měnit. V rámci probíhající operace je možné provádět i krokování zpět. Po dokončení animace (celé operace)

[1] http://www.cs.usfca.edu/~galles/visualization/Algorithms.html

je možné celé operace odvolávat zpět. U datových struktur chybí jakýkoliv popis a princip fungování.



Obrázek 2 – Data Structure Visualizations – Webový prohlížeč (JavaScript)
Zdroj: http://www.cs.usfca.edu/~galles/visualization/StackArray.html

Shrnutí:

- \+ rozsáhlost,
- \+ nevyžaduje doplněk v prohlížeči,
- \+ krokování i zpět,
- – ovládání,
- – bez jakéhokoli popisu,
- – žádná předpřipravená data, ani možnost generování dat.

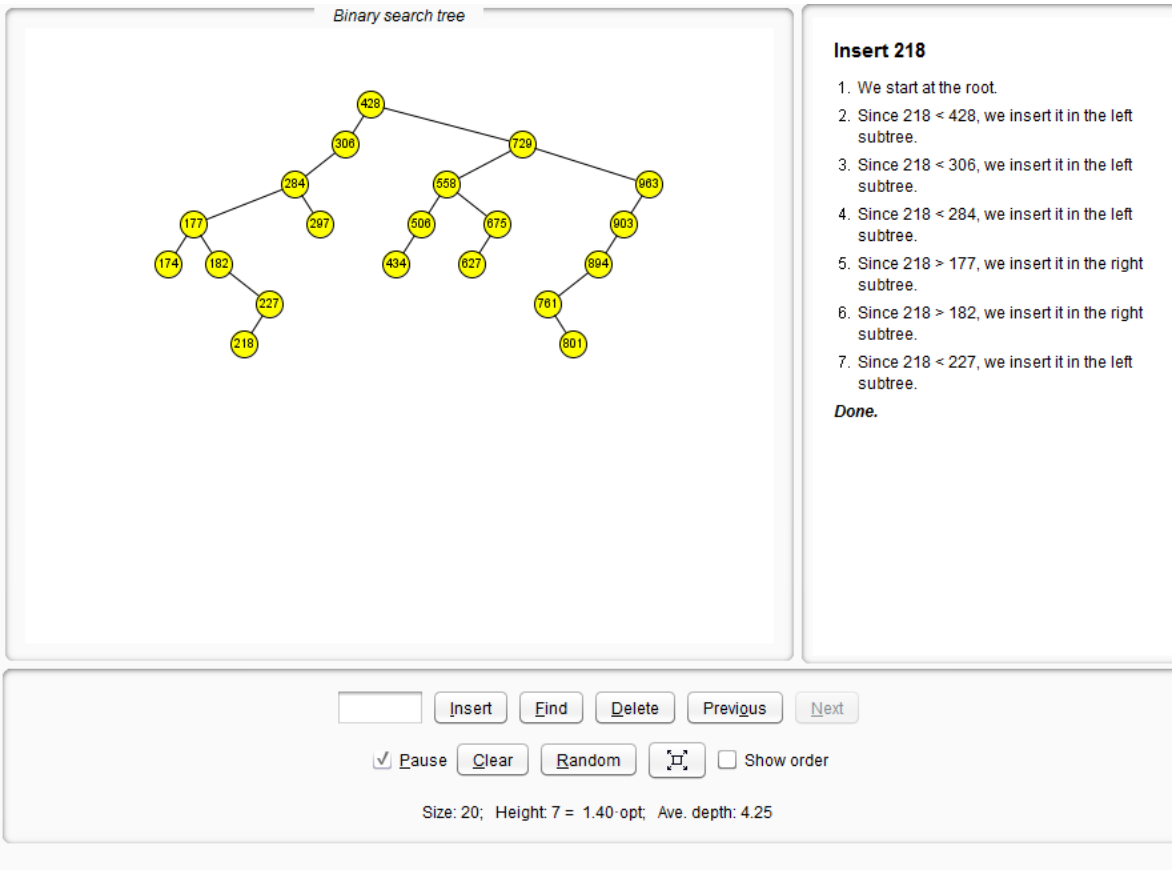
2.2 Webová aplikace Gnarley trees

Tento slovenský projekt[2] je zaměřen zejména na vizualizace vyvážených stromů, prioritních front, některých grafových algoritmů a datových struktur pro práci s řetězci. Vizualizace jsou prováděny ve webovém prohlížeči pomocí Java appletů.

Každá vizualizace datové struktury obsahuje krátký popis, ve kterém se nachází základní myšlenka, výhody a nevýhody. Všechny vizualizace disponují stejným vzhledem a podobným ovládáním. Škoda, že nelze při operacích se strukturami používat klávesové zkratky. Při spuštění appletů je vždy struktura naplněna náhodnými daty. Předpřipravené instance dat nejsou připraveny, ale alespoň lze využít generování dat náhodných. Při provádění dílčích operací je také vypisován doprovodný komentář popisující

[2] http://people.ksp.sk/~kuko/gnarley-trees/

průběh algoritmů. Animace nelze pozastavovat kdekoli, ale pouze v krajních bodech – až po provedení určitých úkonů. Rychlost animací nelze vůbec regulovat.



Obrázek 3 – Gnarley trees – Webový prohlížeč (Java applet)

Zdroj: http://people.ksp.sk/~kuko/gnarley-trees/?page_id=13

Shrnutí:

- \+ popis datových struktur,
- \+ komentáře probíhajících algoritmů,
- \+ generování dat,
- – klávesové zkratky,
- – ovládání rychlosti animace.

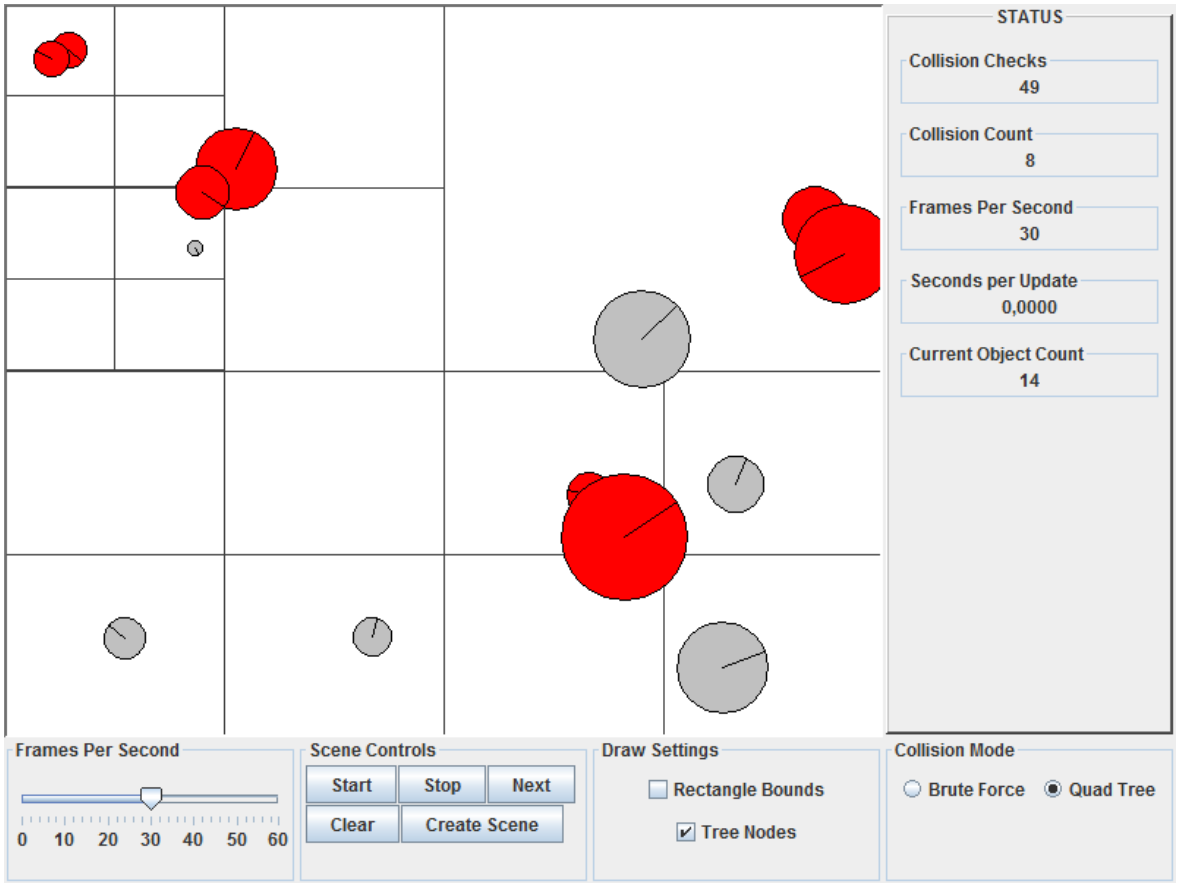
2.3 Webová aplikace CSILM

Další sbírkou nejrůznějších vizualizací datových struktur a algoritmů disponuje Utah State Unversity[3,4]. Na webové stránce se dají nalézt vizualizace seznamů, grafových algoritmů, samovyvažujících stromů, ale například i KMP algoritmu a dalších. Přehlednost webu bohužel není zcela nejlepší, a tak web na první pohled nepůsobí příliš přesvědčivě.

Vizualizace jednotlivých datových struktur nejsou vůbec jednotné – využívány jsou různé technologie (Java applety, Adobe Flash), vzhled a ovládání je u každé struktury jiný.

[3] http://csilm.usu.edu/lms/nav/index.jsp
[4] http://csilm.usu.edu/lms/nav/toc.jsp?sid=__shared&cid=usu@mills&cf=activity

U většiny datových struktur a algoritmů se nachází stručný popis, někde se objevuje i bližší výklad.



Obrázek 4 – CSILM – Java applet

Zdroj:
http://csilm.usu.edu/lms/nav/activity.jsp?sid=__shared&cid=emready@advanced_algorithms&lid=14

Shrnutí:

+ různorodé datové struktury a algoritmy,
+ někde podrobnější výklad,
– nepřehledné stránky,
– nejednotné vizualizace.

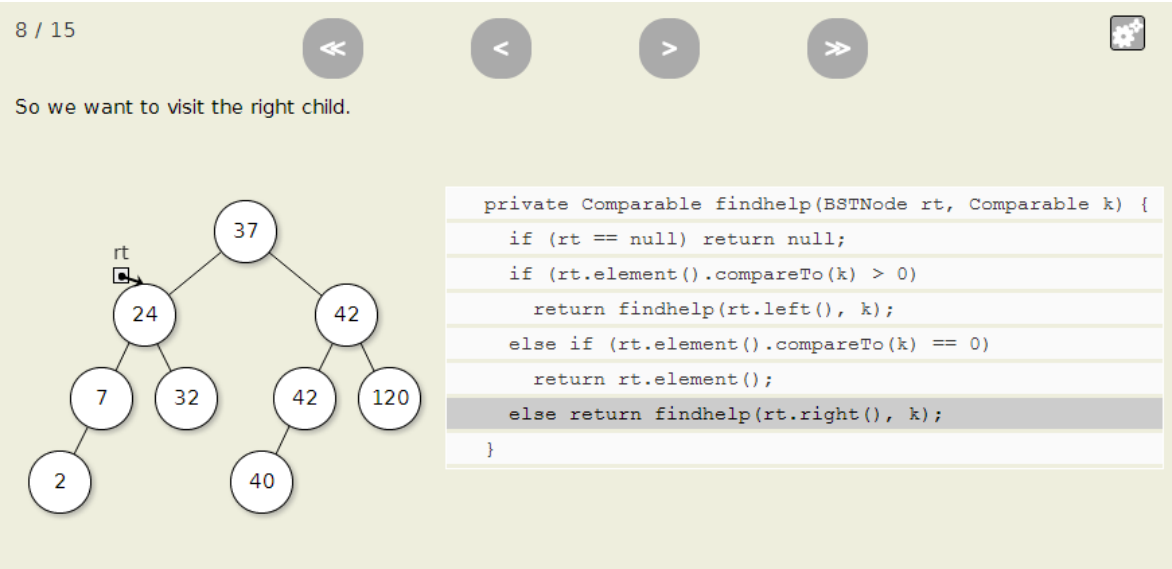
2.4 Další zdroje

V této kapitole budou okrajově zmíněny další zajímavé materiály, které mohou pomoci při studiu datových struktur a algoritmů.

Prvním zajímavým studijním materiálem je webová stránka CS Animated[5]. Na stránce se nachází animované přednášky s mluveným komentářem týkající se jak obecné teorie datových struktur a algoritmů, tak i konkrétních datových struktur.

[5] http://www.csanimated.com/

Velmi pěkným projektem je OpenDSA[6]. Jedná se o plnohodnotný výukový materiál s podrobným výkladem, ilustračními příklady i s částmi zdrojových kódů. Několik příkladů je plně interaktivních (můžeme vkládat vlastní data), jiné příklady jsou realizovány s připravenými daty. Průběh algoritmu bývá komentován, v některých případech dochází v průběhu vizualizace i ke zvýrazňování zdrojového kódu, což usnadňuje pochopení algoritmu. Součástí materiálů jsou také průběžné kontrolní otázky. Prozatím jsou dostupné čtyři kapitoly: lineární struktury, binární stromy, řadící algoritmy a hešování.



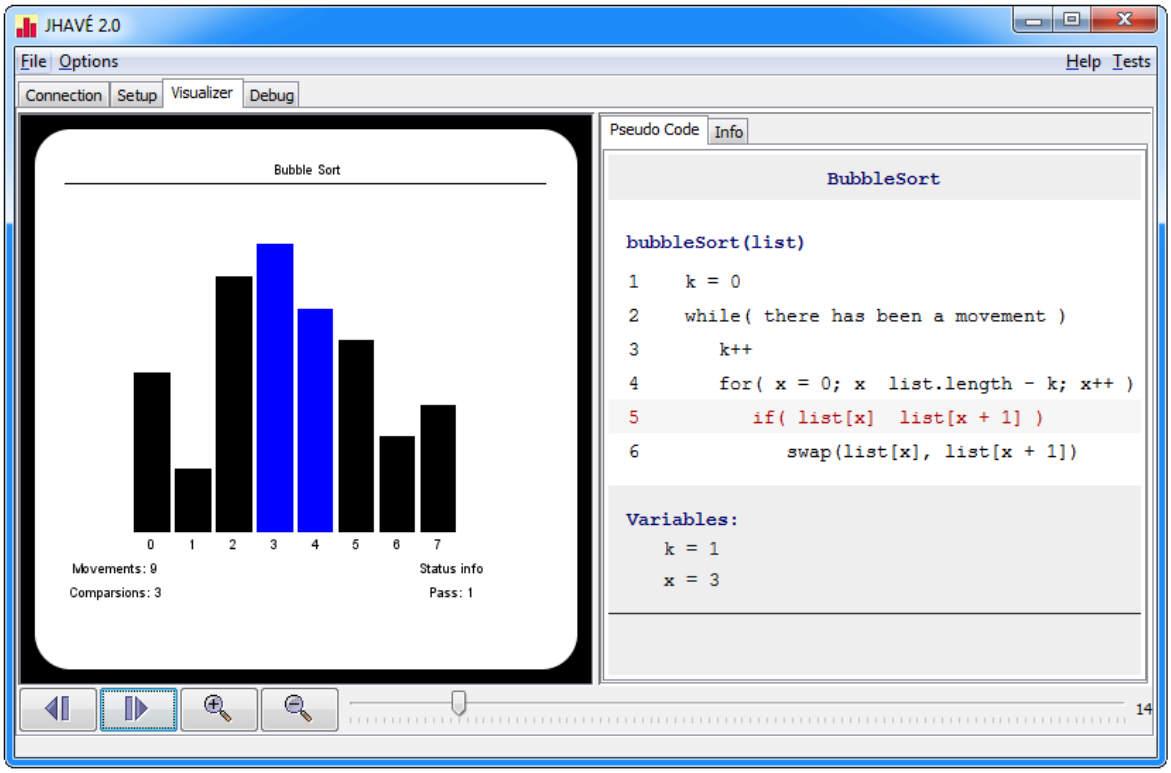
Obrázek 5 – OpenDSA – Webový prohlížeč

Zdroj: http://algoviz.org/OpenDSA/Books/OpenDSA/html/BST.html

Zcela odlišným projektem je JHAVÉ[7]. Jedná se o Java aplikaci, která je spouštěna pomocí technologie Java Web Start. Úkolem této aplikace je pouze vykreslování jednotlivých snímků vizualizace. Po spuštění aplikace je nutné se připojit k serveru, poté vybrat některou nabízenou datovou strukturu, nebo algoritmus a zadat vstupní data. Následně je vizualizace na serveru vypočítána a je poslána klientovi, kde je zobrazována. Zobrazovány jsou postupně jednotlivé kroky algoritmu se současným zvýrazňováním zdrojového kódu. Průběžně jsou také zobrazovány okna s kontrolními otázkami. Ukázka aplikace JHAVÉ zobrazující průběh bublinkového řazení se nachází na obrázku 6.

Posledním představovaným zdrojem informací je web AlgoViz.org[8]. Jedná se o rozsáhlý katalog, kde jsou shromažďovány veškeré zdroje týkající se vizualizace a animace datových struktur a algoritmů. V katalogu lze vyhledávat dle typu struktury, jazyka, použité vizualizační technologie atd. Samozřejmostí je i vyhledávání dle klíčových slov. Kromě základních informací a obrázku vizualizace, je součástí každé vizualizace také její hodnocení.

[6] http://algoviz.org/OpenDSA/Books/OpenDSA/html/
[7] http://jhave.org/
[8] http://algoviz.org/avcatalog



Obrázek 6 – JHAVÉ – Java aplikace

Zdroj: http://jhave.org/code/jhave.jnlp

2.5 Shrnutí

V předchozích kapitolách byly představeny a zhodnoceny tři rozsáhlejší databáze vizualizací datových struktur a algoritmů. Ani jedna z představených databází však neobsahovala všechny běžné datové struktury. V rámci jedné databáze neměly některé vizualizace dokonce ani stejný vzhled, ani ovládání. Setkat se můžeme i s mnoha menšími kolekcemi, ať už kvalitnějšími, či méně kvalitními. Vedle výše uvedených databází existuje i spousta zcela samostatných vizualizací. Pokud tedy nenajdeme požadovanou vizualizaci přímo v některé databázi, je velmi pravděpodobné, že se dá nalézt i samostatně. Například vizualizací B-stromu najdeme desítky.

Během testování dílčích vizualizací byly odhaleny některé nedostatky. Jedná se zejména o absenci:

- klávesových zkratek,
- předpřipravených dat a
- generátoru náhodných dat.

V některých vizualizacích bylo vždy nutné kliknout do vstupního pole, poté kliknout na tlačítko požadované operace a tak dále. Střídání stálého klikání myší a zadávání vstupních dat na klávesnici je velmi zdlouhavé. Zrychlení manipulace s vizualizacemi by se dosáhlo využitím klávesových zkratek.

Začínat vždy s úplně prázdnou strukturou může být také nežádoucí. V některých strukturách začnou probíhat náročnější reorganizace až po vložení určitého počtu prvků. Řešením je možnost dávkového vložení dat do struktury. Data mohou být buď náhodně generována, nebo staticky předpřipravena. Výhodou předpřipravených dat může být vizualizace určitých kritických manévrů a zároveň možnost jejich opětovné replikace. Předpřipravenými daty nedisponovala žádná vizualizace.

Zjištěny byly i další drobné nedostatky. Některé vizualizace např. vůbec neobsahovaly jakýkoli doprovodný text. Žádná vizualizace ve formě animace neumožňovala animaci pozastavit v jakémkoli čase.

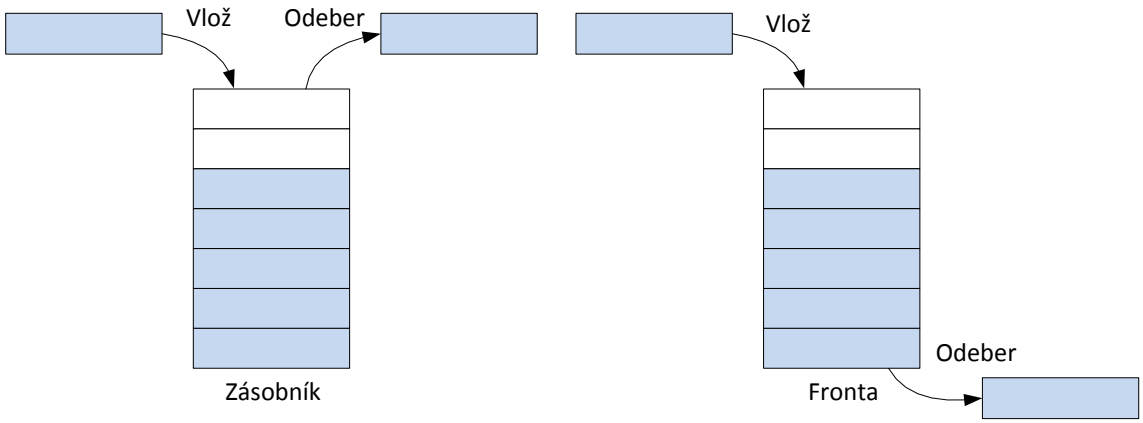
S přihlédnutím k získaným zkušenostem a zjištěným nedostatkům budou naimplementovány vizualizace vybraných datových struktur. Po provedení analýzy bylo zjištěno, že v podstatě neexistuje žádná ucelená databáze, kde by se nacházely vizualizace většiny běžných datových struktur. Překvapení zároveň bylo, že např. vizualizace grid souboru nebyla nalezena vůbec nikde.

3 ADT prioritní fronta

Prioritní fronty jsou často využívané struktury v mnoha různých oblastech. Setkat se s nimi můžeme např. v oblasti počítačových sítí, kde se uplatňují v algoritmech při řízení využití šířky pásma nebo v algoritmech zajišťující kvalitu některých síťových služeb. Dalším využitím prioritních front je oblast diskrétní simulace, kde zajišťují zpracování událostí v chronologickém pořadí. Prioritní fronty mohou být využity také při implementaci některých grafových algoritmů.

Prioritní fronta je množina s lineárním uspořádáním, ve které se pořadí jednotlivých prvků určuje dle jejich priority. Prioritu prvku definujeme jako libovolný prvek z uspořádané množiny. Nejčastěji se jedná o množinu čísel, setkat se ale můžeme např. i s množinou znaků, nebo s datem a časem. (Lewis a Denenberg 1991).

Vložíme-li do prioritní fronty prvky v libovolném pořadí, pak při postupném odebírání dostaneme prvky seřazené dle jejich priority nezávisle na čase vložení. Existují dvě specifické implementace ADT prioritní fronta, u kterých je priorita prvků brána jako čas vložení prvku do struktury. Jedná se o „klasickou“ frontu – struktura typu FIFO – nejvyšší prioritu mají prvky, které se nacházejí ve frontě nejdéle. Druhou specifickou implementací je zásobník – struktura typu LIFO – nejvyšší prioritu mají prvky, které se v zásobníku nacházejí nejkratší dobu. (Kavička 2010). Stylizované zobrazení zásobníku a fronty se nachází na následujícím obrázku.



Obrázek 7 – Princip zásobníku a fronty

Zdroj: vlastní

3.1 Operace

Následující dvě podkapitoly definují jak základní, tak i o rozšiřující operace ADT prioritní fronta. Rovněž budou uvedeny příklady datových struktur, které jsou implementacemi ADT prioritní fronta.

3.1.1 Základní operace

Abstraktní datový typ prioritní fronta definuje na třídě prvků s prioritou následující základní operace (Lewis a Denenberg 1991):

- *VYTVOŘ* – vytvoří prázdnou prioritní frontu,
- *JEPRÁZDNÁ* – vrátí `true`, pokud fronta neobsahuje žádné prvky,
- *VLOŽ* – vloží prvek do prioritní fronty,
- *ZPŘÍSTUPNIMAX* – zpřístupní prvek s maximální prioritou,
- *ODEBERMAX* – odebere a vrátí prvek s maximální prioritou.

Mezi implementace, které disponují výše uvedenými operacemi, patří například (Kavička 2010):

- prioritní fronta nad uspořádaným/neuspořádaným lineárním seznamem,
- dvojseznamová prioritní fronta,
- dvojúrovňová prioritní fronta,
- prioritní fronta na haldě.

3.1.2 Rozšiřující operace

Existují i tzv. *mergable heaps* – slučovatelné haldy (Kučera 2009) – které vedle základních operací definují i další operace (Cormen aj. 2009):

- *ZVYŠPRIORITU* – zvýší prioritu prvku,
- *ODEBER* – odebere a vrátí libovolný prvek z prioritní fronty,
- *SJEDNOCENÍ* – sloučí dvě prioritní fronty do jedné.

Mezi implementace prioritních front s rozšiřujícími operacemi patří (Kavička 2011):

- binomická halda,
- Fibonacciho halda,
- párovací halda.

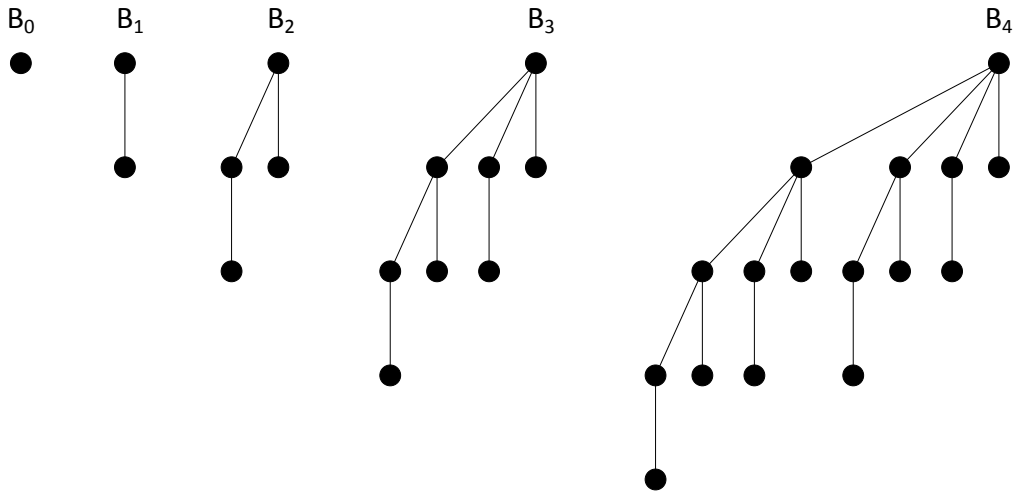
V následující kapitole bude podrobněji probrána problematika binomické haldy, jakožto jedna z vybraných implementací ADT prioritní fronta.

3.2 Binomická halda

Binomická halda se řadí mezi implementace ADT prioritní fronta s rozšiřujícími operacemi – jedná se o tzv. slučovatelnou haldu. Pro pochopení problematiky binomické haldy je nutné nejprve zavést několik základních pojmů.

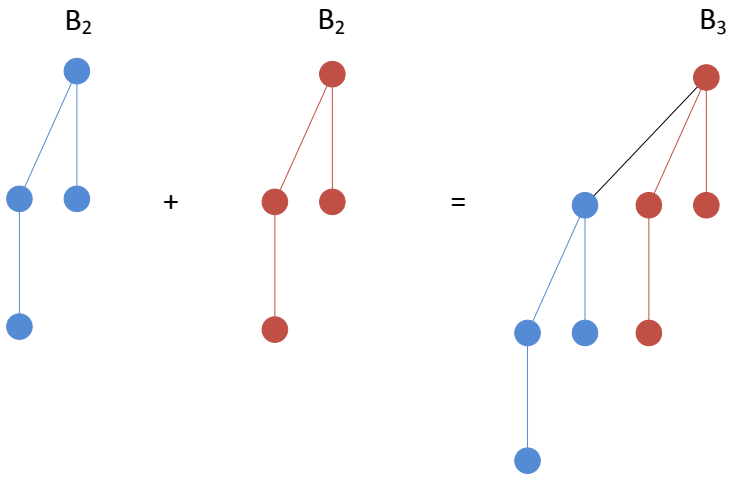
3.2.1 Binomický strom

Binomický strom B_n je uspořádaný strom definovaný rekurzivně. Strom řádu 0 (B_0) se skládá právě z jednoho prvku. Binomický strom řádu *n* lze sestrojit ze dvou binomických stromů o řádu $n - 1$. (Cormen aj. 2001). Vznik binomického stromu spojením dvou binomických stromů stejných řádů je znázorněn na obrázku 9. Na obrázku níže jsou zobrazeny binomické stromy řádu 0 až 4.



Obrázek 8 – Binomický strom B_0, B_1, B_2, B_3 a B_4

Zdroj: vlastní



Obrázek 9 – Vznik binomického stromu řádu 3

Zdroj: vlastní

Další vlastnosti binomického stromu (Cormen aj. 2001):

- Binomický strom B_n disponuje celkem 2^n prvků.
- Výška binomického stromu B_n je *n*.
- V hloubce *i* se nachází právě $\binom{n}{i}$ prvků.
- Kořen binomického stromu o řádu *n* disponuje právě *n* syny.

- Budeme-li číslovat syny binomického stromu B_n zleva, pak 1. syn je kořen binomického stromu stupně $n-1$, 2. syn je kořen binomického stromu stupně $n-2$,… poslední syn je stupně 0.

Například kořen binomického stromu řádu 4 má 4 potomky, jeho první potomek je řádu 3, druhý potomek řádu 2, třetí řádu 1 a čtvrtý nultého řádu. V B_4 se nachází celkem $2^4 = 16$ prvků a jeho výška je 4. Například v hloubce 2 se v binomickém stromu řádu 4 vyskytuje celkem $\binom{4}{2} = 6$ potomků.

3.2.2 Binomický les

Binomický les je les binomických stromů, ve kterém se nenachází žádné dva binomické stromy stejných řádů. Binomický les obsahující k prvků lze sestrojit pouze jedním způsobem. Zapíšeme-li číslo k ve dvojkové soustavě, pak každá jednička tohoto čísla určuje binomický strom obsažený v daném binomickém lese. (Lewis a Denenberg 1991). Názornější vysvětlení poskytne následující příklad.

Mějme 22 prvků, ze kterých chceme vytvořit binomický les. Převod čísla 22 do dvojkové soustavy je zapsán v následující rovnici:

$$(22)_{10} = (\overbrace{1}^{4}\overbrace{0}^{3}\overbrace{1}^{2}\overbrace{1}^{1}\overbrace{0}^{0})_2$$

Z rovnice je patrné, že binomický les skládající se z 22 prvků bude obsahovat binomické stromy o řádech 1, 2 a 4. Pro ověření lze zkontrolovat počet všech prvků pomocí počtu prvků v jednotlivých binomických stromech:

- B_1 obsahuje $2^1 = 2$ prvky,
- B_2 obsahuje $2^2 = 4$ prvky,
- B_4 obsahuje $2^4 = 16$ prvků.

Tyto tři binomické stromy obsahují celkem 22 prvků, což odpovídá původnímu počtu.

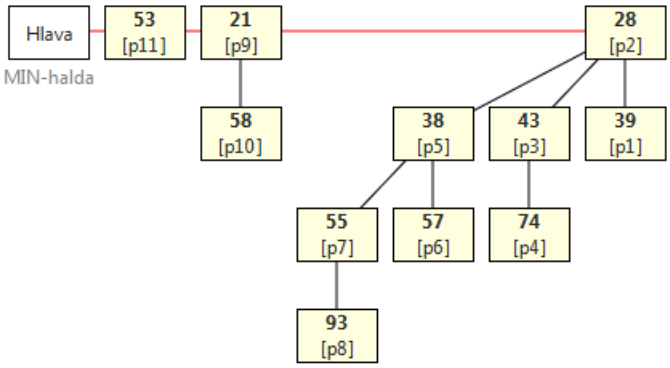
3.2.3 Definice

Binomická halda je označení pro binomický les, pro který platí následující pravidla (Lewis a Denenberg 1991):

- stromy tohoto lesa jsou uspořádány vzestupně dle svých řádů,
- každý strom tohoto lesa dodržuje haldové uspořádání.

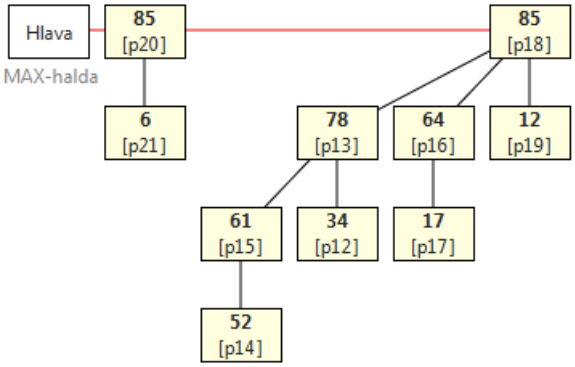
Haldovým uspořádáním se rozumí takové uspořádání, kdy priorita rodičovského prvku je větší nebo rovna prioritě všech jeho potomků. V takovémto případě je halda nazývána jako *max-halda*. Je-li priorita rodiče menší nebo rovna než priorita všech jeho potomků,

hovoříme o *min-haldě*. Na obrázcích 10 a 11 jsou uvedeny příklady min-haldy, resp. max-haldy.



Obrázek 10 – Příklad binomické min-haldy

Zdroj: vlastní



Obrázek 11 – Příklad binomické max-haldy

Zdroj: vlastní

3.2.4 Operace

Binomická halda patří mezi tzv. slučovatelné haldy (mergable heap), disponuje tedy operací, pomocí které je možné haldu sloučit s jinou binomickou haldou. Tato operace je pouze pomocná a je využívána ostatními operacemi. Binomická halda implementuje jak základní operace ADT prioritní fronta (vlož, zpřístupni maximum a odeber maximum), tak i rozšiřující operace – zvyš prioritu prvku, odeber libovolný prvek a sjednocení s jinou haldou. V následující tabulce se nachází přehled asymptotických složitostí jednotlivých operací binomické haldy, přičemž Θ notace označuje asymptotickou výpočetní složitost a *O* notace horní asymptotickou výpočetní složitost (Cormen aj. 2001).

Tabulka 1 – Přehled asymptotických složitostí operací binomické haldy

Operace	Složitost
Vytvoř	$\Theta(1)$
Vlož	$O(\log_2 n)$
Zpřístupni maximum	$\Theta(\log_2 n)$
Odeber maximum	$O(\log_2 n)$
Zvyš prioritu	$O(\log_2 n)$
Odeber	$\Theta(\log_2 n)$
Sjednocení	$\Theta(\log_2 n)$

Zdroj: Cormen aj. (2001)

Na nejhorší možné instanci dat mají operace binomické haldy logaritmickou asymptotickou výpočetní složitost. Podrobný popis všech operací prováděných v rámci binomické haldy bude probrán v kapitole 7, která se věnuje vlastní implementaci.

4 Kódování textu

S textem se v oblasti informačních technologií setkáváme na každém kroku. Na první pohled nás jistě napadnou různé dokumenty, poznámky, elektronické knihy apod. Může se ale jednat například i o zdrojové kódy rozsáhlých aplikací. Během procházení webu jsou jednotlivé webové stránky přenášeny také v textové formě. Při vzdálené správě systémů pomocí příkazové řádky se jednotlivé příkazy či celé skripty také přenášejí ve formě prostého textu. Text je hlavní komunikační kanál mezi člověkem a počítačem. Díky textu je nám umožněno uchovávat nabyté znalosti a další informace.

Cílem kódování textu je reprezentace zdrojového textu pomocí nejkratší možné posloupnosti bitů. Snažíme se tak efektivně využívat datová úložiště. Úsporná reprezentace textu se uplatňuje při archivaci velmi rozsáhlých textů, kdy množství ušetřeného místa v datovém úložišti může být značné. Výhodou úsporné reprezentace textu je také snížení počtu přenášených dat. Může se jednat o přenos v rámci jednoho systému (načítání textu z vnější paměti do vnitřní), nebo o přenos textu mezi dvěma systémy (přenos webové stránky z webového serveru do webového prohlížeče). (Goodrich a Tamassia 2006). Cílem kódování není znemožnění porozumění textu – šifrování. Touto problematikou se zabývá kryptografie.

Na začátku výkladu je nutné definovat některé používané pojmy:

- abeceda M – neprázdná konečná množina znaků, pouze tyto znaky může vstupní text obsahovat;
- vstupní text (řetězec) s – pole o délce n, jehož prvky jsou takové, že $\mathrm{s[i]} \in \mathrm{M}$, $\mathrm{i} = 0, \dots, \mathrm{n} - 1$;
- znak – prvek množiny M;
- kód – posloupnost bitů přiřazená znaku;
- výstupní kód – výsledek transformace vstupního textu pomocí kódovacího algoritmu;
- kódování (komprese) – transformace vstupního textu na výstupní kód;
- dekódování (dekomprese) – transformace kódu zpět na text, dekódovaný text musí přesně odpovídat originálnímu textu.

Nejjednodušším způsobem je uchovávání textu v homogenním souvislém poli. Každému znaku je přiřazen kód konstantní délky. Tento princip je využíván v datových typech pro ukládání řetězců ve vyšších programovacích jazycích. Pro kód o délce n bitů existuje celkem 2^n různých kódů. Máme-li definovánu abecedu M o velikosti m, pak je vyžadováno nejméně $\log_2 m$ bitů k uložení jednoho znaku. Je-li abeceda M např. rovna množině znaků ASCII, jejíž velikost je 256 znaků, pak je potřeba k uložení jednoho znaku právě $\log_2 256 = 8$ bitů. Máme-li vstupní text o délce i znaků definovaný nad množinou ASCII znaků, pak k jeho uložení potřebujeme celkem $i \cdot \log_2 256$ bitů. (Lewis a Denenberg 1991).

4.1 Metody

Úspěšnost, chcete-li kompresní poměr, různých metod závisí na vstupních datech. Kódování textu je založeno na následujícím principu. Předpokládá se, že se ve vstupním textu znaky dané abecedy nevyskytují se stejnou pravděpodobností. Je-li tento předpoklad splněn, pak lze vstupní text zakódovat pomocí kratší posloupnosti bitů. (Lewis a Denenberg 1991). Dle Mensy ČR (2014) je nejfrekventovanějším znakem v česky psaném textu znak *o*. Seznam pěti nejfrekventovanějších znaků se nachází v tabulce 2 (Mensa ČR 2014).

Tabulka 2 – Nejfrekventovanější znaky českého textu

Pořadí	Znak	Pravděpodobnost výskytu
1.	o	8,6664 %
2.	e	7,6952 %
3.	n	6,5353 %
4.	a	6,2193 %
5.	t	5,7268 %

Zdroj: Mensa ČR (2014)

Druhým nejfrekventovanějším znakem je znak *e*, překvapivě frekvence znaku *a* je až na čtvrtém místě. Z tabulky je patrné, že pravděpodobnost výskytu jednotlivých znaků v českém textu není stejná, tudíž lze české texty efektivně kódovat. Nutno podotknout, že pravděpodobnost výskytu jednotlivých znaků není v textech různých jazyků stejná (Lewis a Denenberg 1991). Různá pravděpodobnost výskytu znaků je i v různých typech textu psaného jedním jazykem. Rozdílná frekvence výskytu znaku je např. v české poezii a v české próze.

Po veškerých metodách kódování textu vyžadujeme, aby byly bezztrátové. Ačkoli mohou mít ztrátové metody lepší kompresní poměr, je nepřijatelné, aby rekonstruovaný text přesně neodpovídal originálu. (Lewis a Denenberg 1991). Ztrátové metody nacházejí uplatnění při kompresi obrázků a videa, kde částečné zanedbání obrazové informace nevadí. Ovšem text obsahující pozměněné znaky je pro čtenáře znehodnocený.

Přístupů a algoritmů pro kódování textu existuje celá řada. Uvedeme dva odlišné přístupy (Lewis a Denenberg 1991):

1) kódování jednotlivých znaků,
2) kódování posloupnosti znaků.

Algoritmus, který přiřazuje kódy jednotlivým znakům, se nazývá Huffmanovo kódování a bude podrobně rozebrán v následující kapitole.

Kódování posloupností znaků realizuje Lempel-Ziv kódování. Principem je přiřazení krátkého kódu často se opakujícím posloupnostem znaků. Lempel-Ziv kódování není předmětem této práce, proto nebude dále rozebíráno. Podrobný výklad se nachází například v publikaci autorů Lewise a Denenberga (1991).

Dále lze algoritmy dělit dle použitelnosti pro různé zdroje textu. Zdroje textu mohou být dva (Lewis a Denenberg 1991):

1) v paměti – text je neměnný, máme možnost jej kompletně analyzovat;
2) generovaný proud – text je neukončený, stále přibývají nové znaky, nemáme možnost text kompletně analyzovat.

Například pro text generovaný proudem nelze využít dvoufázové Huffmanovo kódování, neboť je třeba nejprve zjistit frekvence výskytů jednotlivých znaků v celém textu.

4.2 Huffmanovo kódování

Principem Huffmanova kódování je přiřazení co nejkratší posloupnosti bitů nejfrekventovanějším znakům ve vstupním textu. Původní kódy konstantní délky jsou nahrazeny kódy novými, které mají proměnlivou délku. Nejkratší kód obdrží nejfrekventovanější znak. V závislosti na vstupním textu může Huffmanovo kódování ušetřit 20–90 % místa potřebného k uložení tohoto textu. (Cormen aj. 2009).

Úspora místa bude názornější z následujícího příkladu (Cormen aj. 2009). Mějme abecedu $M = \{a, b, c, d, e, f\}$. Velikost abecedy je 6 znaků. Jelikož $\log_2 6 \doteq 2{,}59$, lze znaky této abecedy zakódovat pomocí 3bitových kódů. Vstupní text nad množinou znaků M čítá celkem 100 000 znaků. Frekvence jednotlivých znaků jsou zachyceny v následující tabulce.

Tabulka 3 – Celkové počty bitů při konstantní a proměnné délce kódu

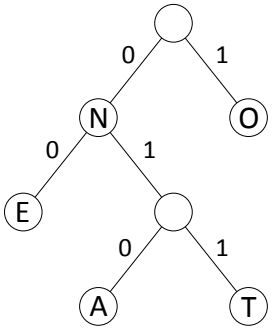
	a	**b**	**c**	**d**	**e**	**f**	**Σ**
Frekvence znaku (tisíce)	45	13	12	16	9	5	100
Kód konstantní délky	000	001	010	011	100	101	–
Počet bitů při konst. délce kódu (tisíce)	$3 \cdot 45$	$3 \cdot 13$	$3 \cdot 12$	$3 \cdot 16$	$3 \cdot 9$	$3 \cdot 5$	**300**
Kód proměnné délky	0	101	100	111	1101	1100	–
Počet bitů při prom. délce kódu (tisíce)	$1 \cdot 45$	$3 \cdot 13$	$3 \cdot 12$	$3 \cdot 16$	$4 \cdot 9$	$4 \cdot 5$	**224**

Zdroj: Cormen aj. (2009)

Z tabulky výše je patrné, že při využití kódů konstantní délky potřebuje k uložení daného textu 300 tisíc bitů (≅ 37 KiB), zatímco při využití kódů proměnlivé délky potřebujeme

jen 224 tisíc bitů (≅ 27 KiB). Zakódováním vstupního textu ušetříme cca 10 KiB místa, což odpovídá asi 25 %. Jedná se pouze o ukázkový příklad, v praxi může být úspora vyšší, ale i nižší. Vše záleží na charakteristice vstupního textu.

S využíváním kódů proměnlivé délky souvisí jeden problém. Jedná se o vhodnou volbu jednotlivých kódů. Musíme zajistit, aby kód jednoho znaku nebyl prefixem kódu jiného znaku. Problematiku prefixů kódů znázorňuje obrázek 12.



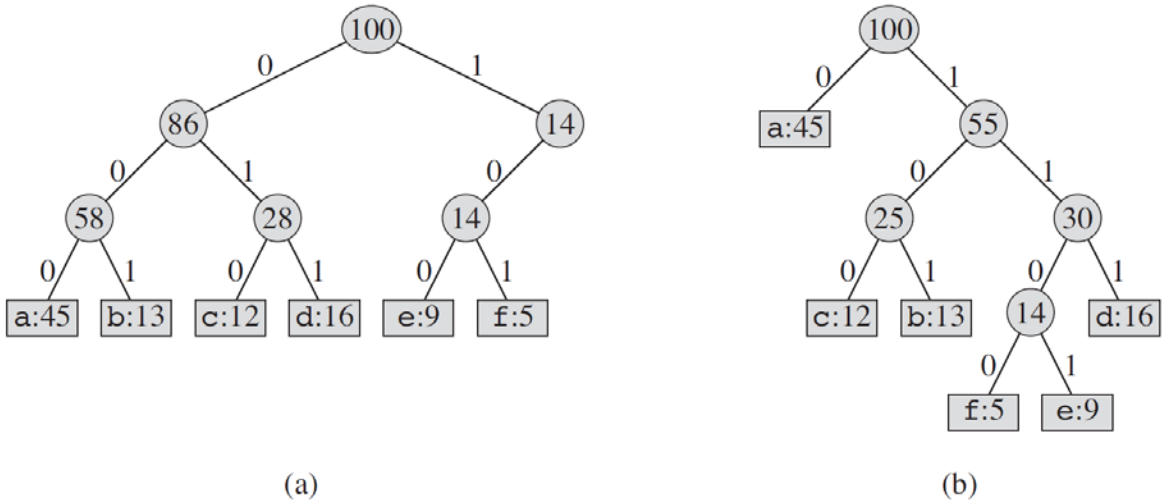
Obrázek 12 – Prefixy kódů

Zdroj: vlastní

Kód(O) = 1, kód(N) = 0, kód(E) = 00, kód(A) = 010 a kód(T) = 011. Pokud zakódujeme vstupní řetězec NON, dostaneme na výstupu kód 010. Tento kód je však již přiřazen znaku A. Problém vznikl díky tomu, že se znak N nenachází v listu, ale ve vnitřním uzlu stromu. Budeme-li všechny znaky ukládat výhradně v listech, pak tato situace nemůže nastat.

4.2.1 Huffmanův strom

Huffmanův strom je reprezentován plným binárním stromem. Plným binárním stromem rozumíme takový binární strom, jehož každý vnitřní uzel disponuje právě dvěma syny. (Cormen aj. 2009). Při průchodu tímto stromem jsou leví potomci reprezentováni bitem 0 a praví potomci bitem 1. V listech jsou uchovávány znaky s jejich četnostmi, v uzlech je uchováván součet četností znaků v příslušných podstromech.



Obrázek 13 – Kódovací stromy korespondující datům z tabulky 3

Zdroj: Cormen aj. (2009)

Na obrázku výše jsou uvedeny dva kódovací stromy. Strom nalevo (a) disponuje kódy konstantní délky, nejedná se o plný binární strom. Kódy stromu napravo (b) jsou proměnné délky, a jelikož všechny uzly (kromě listů) mají právě dva potomky, jedná se plný binární strom.

Text zakódovaný pomocí Huffmanova kódovacího stromu má nejkratší možnou reprezentaci textu. Neexistuje žádný jiný kódovací strom, který by měl kratší reprezentaci textu – Huffmanova optimalita. (Lewis a Denenberg 1991).

Konstrukce Huffmanova stromu bude popsána v implementační kapitole 8. Následující tři podkapitoly čerpají z publikace autorů Lewise a Denenberga (1991).

4.2.2 Dvoufázové Huffmanovo kódování

Během dvoufázového Huffmanova kódování se musí nejprve v prvním průchodu analyzovat vstupní text a až v druhém průchodu probíhá samotné kódování textu. V první fázi algoritmu se zjistí četnost jednotlivých znaků ve vstupním textu. Dle zjištěných četností se vybuduje Huffmanův kódovací strom. V druhé fázi se vstupní text prochází znovu a jednotlivé znaky jsou nahrazovány příslušnými kódy. Spolu se zakódovaným textem je nutné uložit i kódovací strom, jinak bychom nebyli schopni text zpět dekódovat.

Dvoufázové kódování nelze použít pro texty z generovaných proudů, poněvadž nejsme schopni nekončící proud textu předem analyzovat. Kódování textů z generovaných proudů umožňují následující dvě metody.

Dvoufázové Huffmanovo kódování bude podrobně popsáno v kapitole 8.

4.2.3 Statické Huffmanovo kódování

Pro statické Huffmanovo kódování máme k dispozici předpřipravený kódovací strom. Tento strom je vybudován dle charakteristik určitých typů vstupních textů. Budeme-li chtít kódovat rozsáhlé české texty, vybudujeme Huffmanův kódovací strom dle frekvencí výskytů jednotlivých znaků v českém textu. Dle tohoto stromu bude možné téměř optimálně kódovat i jiné české texty. U statického Huffmanova kódování stačí ukládat pouze zakódovaný text, neboť kódovací strom je pevně dán.

4.2.4 Adaptivní Huffmanovo kódování

Na počátku adaptivního Huffmanova kódování máme prázdný kódovací strom. Každý znak abecedy má frekvenci rovnu nule. Po každém zpracování dalšího znaku aktualizujeme stávající kódovací strom tak, aby byl pro již zpracovanou část textu optimální. Aktualizaci kódovacího stromu lze provést v čase úměrném délce kódu přidávaného znaků.

Stejně jako u statického kódování, není ani u adaptivního kódování potřeba spolu s kódovaným textem ukládat kódovací strom. Dekódovací algoritmus začíná s prázdným stromem, který je v průběhu dekódování aktualizován stejným způsobem jako při kódování.

5 ADT tabulka

V praxi se s implementacemi abstraktního datového typu tabulka setkáváme velmi často. Na rozdíl od ADT pole, kde se k jednotlivým prvkům přistupuje přes celočíselný index, ADT tabulka definuje přístup k datům pomocí jednoznačných klíčů. Motivací pro využívání klíčů je ukládání přidružených informací k těmto klíčům a zároveň co nejrychlejší přístup k těmto datům pomocí jejich klíčů. Implementace ADT tabulka nemusejí být pouze struktury uchovávané v operační paměti. Může se jednat i o struktury vhodné k uchovávání dat na externích paměťových médiích. Mnohdy mohou být ukládané informace velmi rozsáhlé, tudíž jejich stálé udržování v operační paměti není efektivní, někdy dokonce ani možné.

Oblast, která je prakticky celá založena na filozofii ADT tabulka, je oblast databázových systémů. S databázovými systémy se v dnešní době setkáváme na každém kroku. Může se jednat např. o evidenci obyvatel, registr vozidel, kartotéku u lékaře, sortiment internetového obchodu, firemní údaje o zaměstnancích, databáze knih a jejich výpůjček, různé rezervační systémy apod. Například v databázi knih může být přístupovým klíčem ISBN, v registru vozidel se může jednat o státní poznávací značku.

Samozřejmě se ADT tabulka nevyužívá jen v oblasti databázových systémů. ADT tabulka se využívá všude tam, kde požadujeme přístup k datům přes klíč. Např. během Huffmanova kódování vytváříme tabulku kódů, kde klíčem je znak a jeho přidruženou hodnotou je odpovídající kód znaku. Dalším příkladem může být tabulka symbolů z oblasti kompilátorů, ve které jsou uchovávány všechny identifikátory a informace o nich.

Implementací ADT tabulka existuje celá řada. Jednotlivé implementace se liší zejména časovými složitostmi prováděných operací. Některé implementace jsou vhodné pro uchování dat na externích paměťových médiích, jiné jsou vhodné pro uchovávání dat v operační paměti. V zásadě lze implementace ADT tabulka kategorizovat následovně (Kavička 2010):

- tabulka na poli,
- tabulka na seznamu,
- tabulka na vyhledávacím stromu,
- tabulka na implicitní kosočtvercové síti,
- rozptýlená tabulka (hashovací).

„Tabulka je množina s lineárním uspořádáním, přičemž uspořádání je určováno jednoznačnými klíčovými hodnotami (klíči) prvků.“ (Kavička 2010). Klíčem může být libovolný datový typ. Klíče mohou být i vícerozměrné (multidimenzionální), pak se k datům přistupuje pomocí tohoto složeného klíče. Kupříkladu k bodu uloženému v prostoru můžeme přistupovat přes vícerozměrný klíč skládající se ze souřadnic x, y a z.

V této práci bude podrobněji probrána implementace ADT tabulka vhodná k ukládání na externích paměťových médiích – B-strom. Druhou podrobněji probranou strukturou bude

grid soubor, který je také určen k uchovávání dat na externích paměťových médiích a umožňuje k datům přistupovat pomocí vícerozměrných klíčů.

5.1 Operace

ADT tabulka definuje na množině prvků s klíči pět operací (Lewis a Denenberg 1991):

- *VYTVOŘ,*
- *JEPRÁZDNÁ,*
- *VLOŽ,*
- *ODEBER,*
- *NAJDI.*

Operace *VYTVOŘ* vytvoří prázdnou tabulku, do které je možné vkládat prvky s klíči.

Operace *JEPRÁZDNÁ* testuje, zda tabulka obsahuje prvky. Pokud žádné prvky neobsahuje, vrátí operace *JEPRÁZDNÁ* `true`.

Operace *VLOŽ* vloží do tabulky prvek `P` s klíčem `K`.

Operace *ODEBER* odstraní z tabulky prvek s klíčem `K`.

Operace *NAJDI* najde dle klíče `K` prvek `P` a vrátí jej.

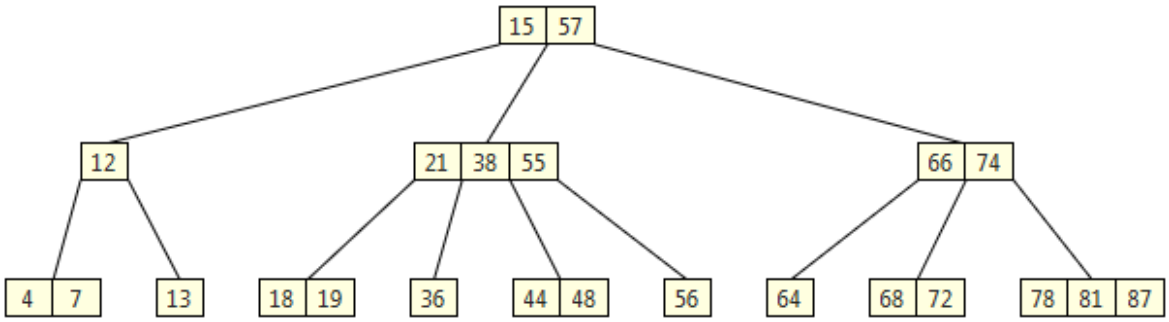
V následujících podkapitolách bude přiblížena problematika dvou vybraných datových struktur – B-stromu a grid souboru.

5.2 B-strom

B-strom je datová struktura vyvinutá pány Rudolf Bayer a Ed McCreight v Boeing Research Labs v roce 1971. Význam počátečního písmena B není přesně znám. Může se jednat o počáteční písmeno slova Boeing, Bayer, či balanced – vyvážený. (Lewis a Denenberg 1991, B-tree 2014).

B-strom je implementací ADT tabulka. Jedná se o dokonale vyvážený vyhledávací strom. Dokonale vyvážený strom je takový strom, jehož všechny listy se nacházejí ve stejné hloubce. B-stromy jsou vhodné k uchovávání dat na externích paměťových médiích.

Filozofie B-stromu je podobná jako u binárního vyhledávacího stromu. Na rozdíl od binárního vyhledávacího stromu je v uzlech B-stromu uloženo více klíčů, tím pádem mají uzly i více potomků. Na obrázku 14 se nachází příklad B-stromu.



Obrázek 14 – B-strom

Zdroj: vlastní

„Pro každý klíč *K* z interního vrcholu platí, že je větší nebo roven všem klíčům z jeho levého přilehlého podstromu a menší než všechny klíče z jeho pravého přilehlého podstromu.“ (Kavička 2011).

Vlastnosti B-stromu jsou následující (Cormen aj. 2009):

- každý uzel *x* obsahuje *x.n* klíčů uložených v neklesajícím pořadí ($x.klíč_1, x.klíč_2, \ldots, x.klíč_n$); $x.klíč_1 \leq x.klíč_2 \leq \ldots \leq x.klíč_{x.n}$;
- každý uzel *x* obsahuje *x.n+1* potomků ($x.syn_1, x.syn_2, \ldots, x.syn_{n+1}$);
- klíče uzlu *X* separují rozsahy klíčů uložené v podstromech; pokud K_i je libovolný klíč uložený v podstromu s kořenem $X.syn_i$, pak:
$$K_1 \leq X.klíč_1 \leq K_2 \leq X.klíč_2 \leq \ldots \leq K_i \leq X.klíč_i \leq X.klíč_{x.n} \leq K_{x.n+1};$$
- všechny listy se nacházejí ve stejné hloubce, hloubka listů se nazývá výška stromu;
- minimální a maximální počty klíčů v uzlech (řád B-stromu) jsou definovány odlišně dle různých autorů.

Definice B-stromu není v literatuře bohužel zcela jednotná. Největší rozdíly jsou v definici řádu B-stromu. Řád B-stromu určuje minimální a maximální počty klíčů (a tedy i potomků) v uzlech. Jednotnost však není ani u pojmu list. (B-tree 2014). Vzhledem k těmto nejednoznačnostem existuje celá řada různých pojetí a implementací B-stromu.

Jelikož jsou B-stromy využívány zejména pro uchovávání dat na externích paměťových médiích, bývají minimální a maximální počty klíčů voleny tak, aby velikost jednotlivých uzlů odpovídala paměťové velikosti bloku zařízení. Obvykle jsou počty klíčů v uzlech v řádu stovek. (Cormen aj. 2009).

5.2.1 Parametrizace

Různí autoři definují řád B-stromu – minimální a maximální počty klíčů – odlišně. V závislosti na určité definici mohou být uzly prázdnější (obsahovat menší počet klíčů), nebo plnější (obsahovat větší počet klíčů). Například autoři Bayer a McCreight, Cormen a další definují řád B-stromu jako *minimální počet uložených klíčů* v uzlech (kromě kořene), naproti tomu Knuth definuje řád B-stromu jako *maximální počet potomků* (B-tree 2014). Podrobně budou představeny definice dle Cormena aj. a dle Lewise a Denenberga.

Cormen aj. (2009) definují počty klíčů v uzlech pomocí parametru t, jež nazývají minimálním stupněm (minimum degree) B-stromu. Tento parametr je libovolné celé číslo ≥ 2, přičemž minimální a maximální počty klíčů jsou definovány takto:

- každý uzel kromě kořene obsahuje minimálně $t - 1$ klíčů, tudíž disponuje minimálně t potomky; pokud je strom neprázdný, pak kořen obsahuje minimálně 1 klíč;
- každý uzel obsahuje maximálně $2t - 1$ klíčů, tudíž disponuje maximálně $2t$ potomky.

Položíme-li parametr $t = 2$, pak dostáváme B-strom nejnižšího řádu, jehož vnitřní uzly mohou mít 2, 3 nebo 4 potomky. Tento strom se nazývá jako 2-3-4 strom.

Lewis a Denenberg (1991) definují B-strom zobecněním 2-3 stromu na (a, b)-strom, jehož parametry a a b jsou specificky zvoleny. Pro definování B-stromu je tedy nutné nejprve definovat 2-3 strom:

- všechny jeho listy se nacházejí ve stejné hloubce;
- uzly obsahují 1 nebo 2 klíče, tím pádem 2 nebo 3 potomky;
- vnitřní uzel obsahuje buď:
    - jeden klíč a dva potomky (2-uzel);
    - dva klíče a tři potomky (3-uzel);
- klíč v rodiči separuje dva podstromy – levý a pravý:
    - v levém podstromu se nacházejí klíče menší nebo rovny než separující klíč;
    - v pravém podstromu se nacházejí klíče větší než separující klíč.

Zobecněním 2-3 stromu je (a, b)-strom. Pomocí parametrů a a b definujeme minimální a maximální počty potomků v (a, b)-stromu:

- $\forall a \in \mathbf{N}; a \geq 2;$
- $\forall b \in \mathbf{N}; b \geq 2a - 1;$
- $\exists c \in \mathbf{N}; a \leq c \leq b;$
- pak uzel (a, b)-stromu je buď listem, nebo obsahuje c potomků;
- výjimkou je kořen, pro který platí: $2 \leq c \leq b$.

A konečně se dostáváme k definici B-stromu. B-strom řádu b označujeme jako (a, b)-strom, jehož parametr $b = 2a - 1$. Zvolíme-li parametr $a = 100$, pak $b = 199$. Vložíme-li do takového B-stromu milion prvků, pak bude jeho výška $\log_{100} 10^6 = 3$. Budeme-li v tomto stromu hledat libovolný prvek dle klíče, pak potřebujeme pouze 4 přístupy na paměťové médium. (Lewis a Denenberg 1991).

5.2.2 Varianty

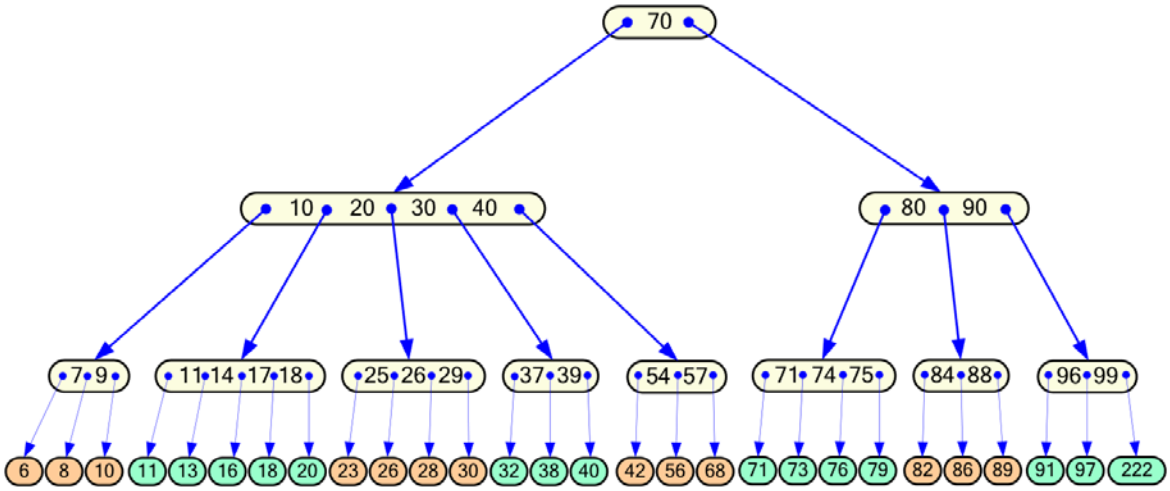
Existuje více variant B-stromů. V rámci této kapitoly budou představeny následující dvě varianty:

- B^+-strom a
- B^*-strom.

V „obyčejném“ B-stromu jsou klíče spolu s přidruženými daty uchovávány jak ve vnitřních uzlech, tak v listech.

Organizaci B-stromu lze upravit tak, aby bylo jeho uspořádání vhodnější pro uložení dat na externím paměťovém médiu. Budeme-li v interních listech uchovávat pouze separující klíče (bez přidružených dat), jsme schopni do jednoho paměťového bloku zařízení uložit více těchto separujících klíčů. Čím více klíčů uložíme v jednom paměťovém bloku, tím méně potřebujeme provést zpřístupnění paměťových bloků s rostoucím počtem uložených záznamů.

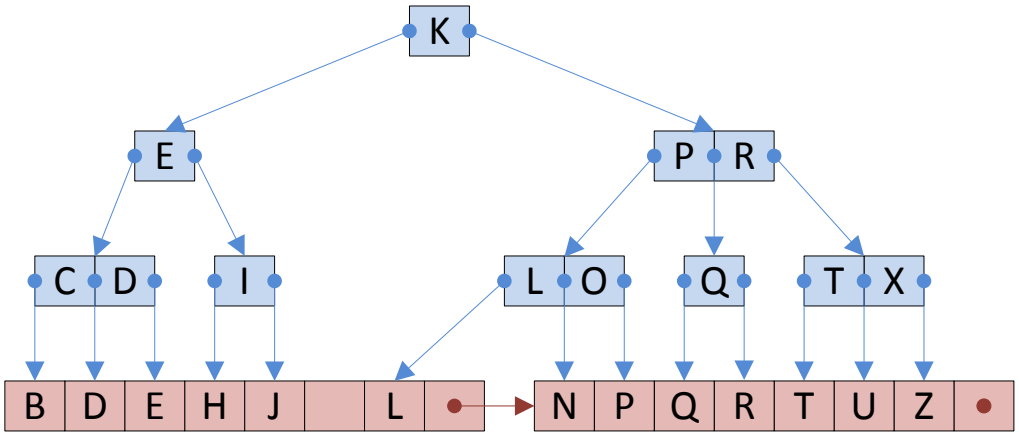
Varianta, ve které se ukládají klíče spolu s přidruženými daty výhradně v listech, se nazývá B^+-strom. V interních listech B^+-stromu se nacházejí pouze separující klíče a odkazy na potomky. (Cormen aj. 2009). Hierarchii vnitřních uzlů nazýváme jako přístupovou strukturu (index) ke zpřístupňování listů, ve kterých jsou uložena kompletní data. Klíče nacházející se v listech se nemusí nutně vyskytovat v přístupové struktuře. (Kavička 2011). Struktura B^+-stromu je znázorněna na obrázku 15.



Obrázek 15 – B^+-strom

Zdroj: Kavička (2011)

Další variantou je B^*-strom. Definice této varianty se dle různých autorů liší. Podle Cormena aj. (2009) je B^*-strom takový B-strom, který vyžaduje, aby každý vnitřní uzel byl zaplněn alespoň ze dvou třetin (u „obyčejného“ B-stromu mohou být vnitřní uzly zaplněny pouze z poloviny). Naproti tomu Lewis a Denenberg (1991) popisují B^*-strom tak, že listy tohoto stromu jsou lineárně zřetězeny, což umožňuje sekvenční průchod všemi uloženými záznamy. Záznamy jsou pak na výstupu samozřejmě seřazeny vzestupně dle svých klíčů. Zřetězení listů je patrné z následujícího obrázku.



Obrázek 16 – B*-strom

Zdroj: přepracováno dle Lewis a Denenberg (1991)

5.2.3 Operace

Vyhledávání prvků (dle jejich klíčů) je v B-stromu založeno na stejném principu jako u binárního vyhledávacího stromu. Hledáme-li prvek s klíčem *K*, pak proběhne traverz od kořene do listů odpovídajícími větvemi (dle hodnoty hledaného klíče). V listu je následně uskutečněn lineární traverz všemi uloženými klíči. Pokud se v listu vyskytuje prvek s odpovídajícím klíčem, je prvek vrácen.

Při operacích *VLOŽ* a *ODEBER* je proveden traverz k příslušnému prvku. V průběhu tohoto traverzu může docházet k reorganizacím struktury B-stromu. Během vkládání prvků se musí ošetřit situace, aby žádný uzel neobsahoval větší počet klíčů, než je dovoleno. Naopak při odebírání prvků se musí zajistit, aby žádný uzel neobsahoval méně klíčů, než je v konkrétním B-stromu požadováno. Při vkládání hovoříme o přetečení uzlu, při odebírání o podtečení uzlu.

Přetečení uzlu je řešeno přesunutím klíče do rodiče a rozdělením uzlu na dva nové. Toto dělení uzlů bývá označováno jako *štěpení*. Podtečení uzlu může být řešeno přesunem klíčů z jiných uzlů, nebo spojením s jiným uzlem. Spojení s jiným uzlem bývá označováno jako *fúze* uzlů.

5.2.4 Štěpení a fúze uzlů

Nyní budou představeny principiálně dva odlišné přístupy řešení přetečení, resp. podtečení uzlů:

a) vložení, resp. odebrání klíče z příslušného listu, následné řešení přetečení (podtečení) *při zpětném průchodu* od listu ke kořenu,
b) řešení potenciálního přetečení (podtečení) uzlu již *při průchodu od kořene do* adekvátního *listu*.

Nevýhodou prvního přístupu může být opětovné zpřístupňování paměťových bloků při zpětném průchodu, což může být časově náročná operace na externích paměťových médiích. Tuto nevýhodu odstraňuje druhý přístup, který ovšem může vést ke „zbytečným“ reorganizacím. Při traverzu do adekvátního listu mohou být na cestě potenciálně přeplněné (poloprázdné) uzly, které jsou reorganizovány tak, aby se při případném dělení (spojování) uzlů nedostaly do nepřípustného stavu. List na konci traverzu však nemusí být vůbec potřeba dělit (spojovat), pak není ani nutné jakkoli reorganizovat uzly na cestě k tomuto listu. Tento druhý přístup bývá označován jako proaktivní (preventivní) štěpení a fúze uzlů.

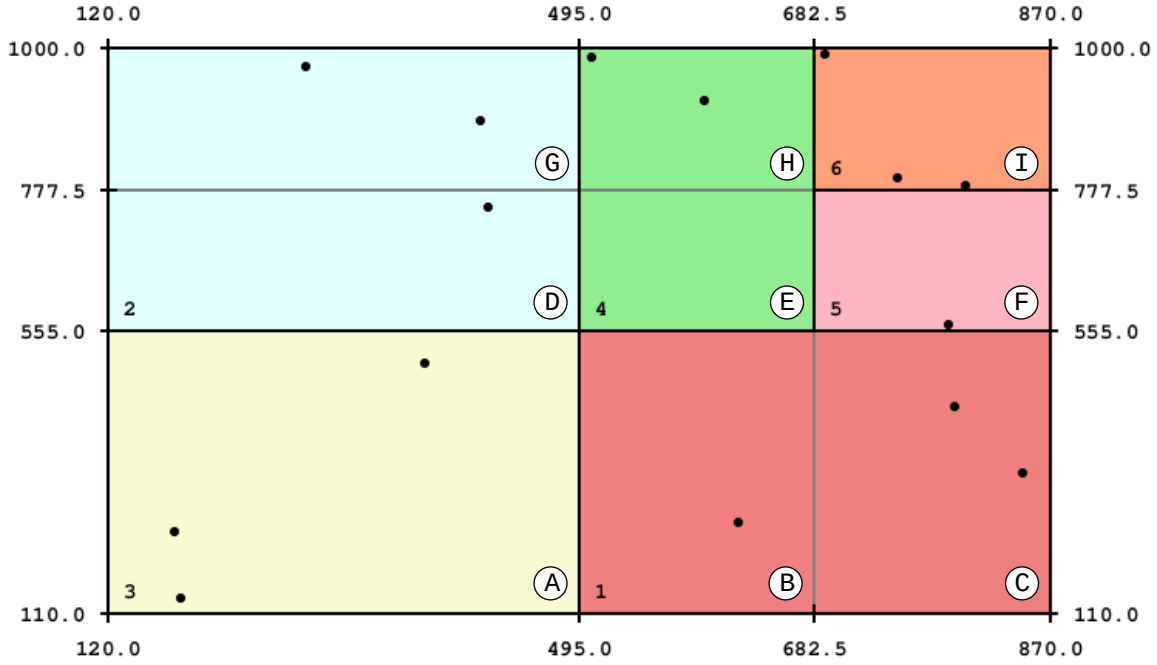
V rámci této práce bude v kapitole 9 představena implementace B-stromu dle Cormena aj. (2009) s proaktivním štěpením a fúzí uzlů.

5.3 Grid soubor

Grid soubor je struktura pro uchovávání multidimenzionálních dat. Motivací grid souboru je přístup k záznamům dle vícerozměrného klíče pouze pomocí dvou diskových operací, a dále efektivní vyhledávání záznamů, jejichž klíče náleží určitému rozsahu hodnot. Klíče těchto dat nejsou tedy jednoduché, ale jsou složené. Do grid souboru můžeme například ukládat informace o bodech v prostoru. Složený klíč těchto bodů by se pak skládal ze souřadnic x, y a z. Koncepce grid souboru byla představena v roce 1984 pány Nievergeltem a Hinterbergerem (1984). Celá tato kapitola a její podkapitoly čerpají téměř veškeré informace z knihy Foundations of multidimensional and metric data structures (Samet 2006).

Dle Sameta (2006) řadíme grid soubor do tzv. bucket metod. Bucketem (česky kyblík) rozumíme vyhrazený prostor pro určitou skupinu dat. Data jsou tříděna do skupin v závislosti na hodnotě jejich klíče. Velikost každého bucketu (dále již datový blok) je dána maximálním možným počtem uložených záznamů. Obvykle se počet záznamů volí tak, aby odpovídal velikosti paměťového bloku zařízení. Grid soubor je konkrétně kategorizován jako jedna z implementací grid directory metod. Princip grid directory metod spočívá v rozdělení prostoru klíčů do více oblastí – grid buněk.

Na obrázku níže je naznačena organizace grid souboru, do kterého jsou ukládány prvky s dvourozměrným klíčem – dle souřadnice x a y. Barevně jsou odlišeny jednotlivé datové bloky, přičemž každý blok je identifikován svým číslem (1–6).



Obrázek 17 – Struktura grid souboru

Zdroj: vlastní

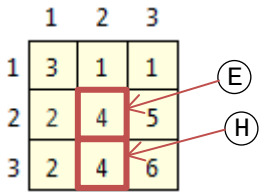
V každém datovém bloku se v tomto případě mohou vyskytovat maximálně tři záznamy. Všechny klíče se nacházejí v dvourozměrném souřadnicovém systému, kde koordináty $x \in \langle 120; 870)$ a $y \in \langle 110; 1000)$. Tento prostor je naznačenou mřížkou rozdělen celkem do 9 grid buněk (A–I). Všimněte si, že některé grid buňky mohou sdílet stejné datové bloky. Například datový blok 1 je využíván grid buňkou B i C.

Grid soubor se skládá celkem ze tří částí:

- grid adresáře (grid directory),
- lineárních stupnic (linear scales) a
- datových bloků (buckets).

5.3.1 Adresář

Pokud budou klíče uchovávaných prvků *d*-dimenzionální, pak interpretujeme grid adresář jako *d*-rozměrné pole. Prvky tohoto pole označujeme jako grid buňky (grid cells). V každé grid buňce se nachází ukazatel na datový blok s uloženými záznamy. Grid adresář je tedy zodpovědný za mapování jednotlivých grid buněk do datových bloků. Grid adresář odpovídající grid souboru na obrázku 17 se nachází na obrázku níže (adresář je vzhledem k vizualizaci na obrázku 17 svisle překlopen). Jednotlivé buňky adresáře obsahují čísla příslušných datových bloků.



Obrázek 18 – Grid adresář

Zdroj: vlastní

Během vkládání prvků do grid souboru nastane dříve či později situace, kdy bude vkládaný prvek patřit do datového bloku, který je již plný. V této situaci musíme adresář (stejně tak jako lineární stupnice) rozdělit. Strategie dělení adresáře a stupnic bude probrána dále.

5.3.2 Stupnice

Adresář bez stupnic je obyčejné nicneříkající pole. Přidružením stupnic ke každé dimenzi adresáře získáme informace o krajních hodnotách jednotlivých grid buněk. Jestliže je grid adresář *d*-dimenzionální pole, pak grid soubor disponuje právě *d* stupnicemi v podobě jednorozměrného pole. Hodnoty stupnic na obrázku 19 odpovídají mezním hodnotám a hodnotám řezů mřížky grid souboru, jehož podoba je zachycena na obrázku 17.

	1	2	3	4
x	120.0	495.0	682.5	870.0

	1	2	3	4
y	110.0	555.0	777.5	1000.0

Obrázek 19 – Stupnice grid souboru

Zdroj: vlastní

Nyní máme k dispozici kompletní aparát pro ukládání a vyhledávání prvků v grid souboru. Budeme-li hledat například prvek s klíčem $x = 700$ a $y = 600$, jsme schopni pomocí lineárních stupnic a adresáře dohledat datový blok, ve kterém je hledaný prvek uložen (pokud existuje).

Pokud budeme hledat prvek s výše uvedenými souřadnicemi, budeme postupovat následovně. Na stupnici *x* najdeme první větší hodnotu, než je 700, tj. 870. Index této hodnoty je 4. Snížíme-li hodnotu tohoto indexu o 1, dostaneme hodnotu indexu v dimenzi *X* v grid adresáři. Analogicky zjistíme hodnotu pro dimenzi *Y*. Do adresáře tedy přistoupíme na index [3, 2]. V této grid buňce se nachází identifikátor datového bloku 5, ve kterém je hledaný prvek uložen (pokud existuje). Korektnost přístupu do datového bloku lze vizuálně zkontrolovat na obrázku 17. Hledané souřadnice [700, 600] se opravdu nacházejí v datovém bloku 5.

5.3.3 Operace

V grid souboru jsou na množině prvků s vícedimenzionálním klíčem definovány tři následující operace:

- *NAJDI*,
- *VLOŽ*,
- *ODEBER*.

Během operace *NAJDI* je *d*-dimenzionální klíč porovnáván s *d* lineárními stupnicemi. Na základě zjištěných indexu na stupnicích se přistoupí na určité místo do grid adresáře. Odtud jsme odkázání do příslušného datového bloku, kde se hledaný prvek vyskytuje (pokud existuje).

Operace *VLOŽ* najde výše naznačenou strategií datový blok, do kterého má být vkládaný prvek uložen. Pokud nalezený datový blok obsahuje maximální možný počet záznamů, musí operace *VLOŽ* zajistit rozdělení grid adresáře a stupnic tak, aby mohl být prvek adekvátně uložen. Strategie dělení adresáře a stupnic bude představena v kapitole 5.3.4.

Operace *ODEBER* nejprve najde odebíraný prvek opět dle výše naznačené strategie. Pokud odebíraný prvek existuje, pak je z datového bloku odstraněn. Oproti vkládání prvku musíme ošetřit situaci, aby některý datový blok nezůstal moc prázdný. Strategie a podmínky slučování datových bloků budou probrány v kapitole 5.3.5.

5.3.4 Strategie dělení adresáře a stupnic

Nastane-li během vkládání situace, že máme vložit prvek do plného datového bloku, musíme rozdělit adresář a stupnice tak, aby mohl být prvek bez problémů uložen. V této kapitole nebude popsán konkrétní algoritmus dělení, ale různé možnosti tohoto dělení.

Podoba grid adresáře a lineárních stupnic je závislá na zvolené strategii dělení. Typicky nás zajímá:

- v které dimenzi provést dělení a
- lokace dělícího bodu.

Nejjednodušší volbou dělené dimenze je volba dle pevně daného plánu. Může se jednat například o pravidelné střídání dělených dimenzí. Můžeme však dělení v jedné dimenzi upřednostňovat před dělením v jiných dimenzích. Pokud pak budeme provádět pouze částečně specifikované vyhledávání (např. hledáme klíče v jedné dimenzi v určitém rozsahu a v ostatních dimenzích všechny klíče), pak budou výsledky hledání specifikované pouze nad častěji dělenou dimenzí přesnější.

Lokace dělícího bodu nemusí vždy být nutně uprostřed děleného intervalu. Dělící bod může být volen tak, aby odpovídal určité množině dat – například měsíce a týdny na časové ose. (Nievergelt a Hinterberger 1984).

5.3.5 Strategie slučování datových bloků

V této kapitole bude podrobněji představeno slučování datových bloků. Na začátku je nutné poznamenat, že slučování se nemusí v grid souboru vyskytovat pouze u datových bloků, ale i v grid adresáři. Obvykle není slučování v rámci grid adresáře požadováno, tudíž zde nebude uváděno.

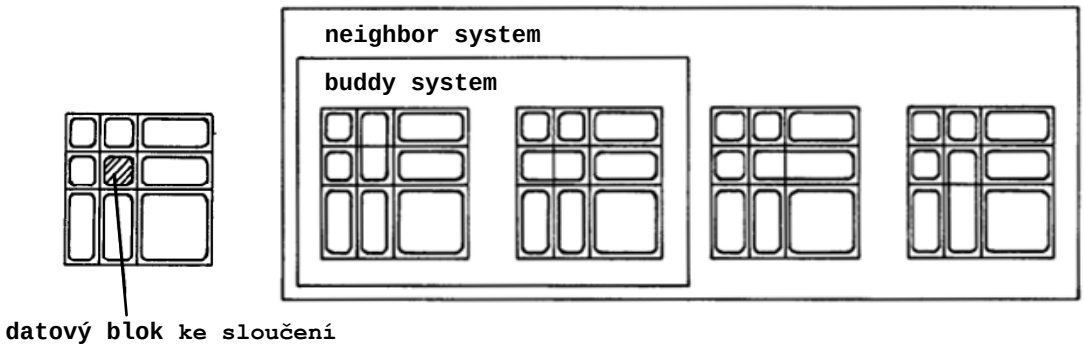
Během slučování datových bloků nás zajímají odpovědi na následující tři otázky:

- které dvojice bloků můžeme sloučit;
- která z těchto dvojic je preferovaná;
- při jaké hranici obsazenosti datového bloku máme sloučení provést.

Které datové bloky jsou kandidáti ke sloučení, určuje zvolený systém, a to buď:

- „buddy systém“ nebo
- „neighbor systém".

V „buddy systému“ může být datový blok sloučen pouze s jedním sousedem v každé dimenzi. Z obrázku 20 je patrné, že se jedná buď o levého, nebo o horního souseda z grid adresáře (pro $d = 2$). Tento systém slučování je doporučován jako standardní.



Obrázek 20 – Systémy slučování datových bloků v grid souboru
Zdroj: Nievergelt a Hinterberger (1984)

V „neighbor systému“ může být datový blok sloučen s dvěma sousedy v každé dimenzi, tzn. jak s levým a horním, tak i s pravým a dolním sousedem v grid adresáři (pro $d = 2$). Rozšíření „buddy systému“ na „neighbor systém“ rovněž zachycuje obrázek 20.

Kterou dvojici při slučování upřednostníme před jinou, záleží na zvolené strategii dělení dimenzí. Pokud upřednostňujeme dělení v jedné dimenzi před ostatními, nesmí strategie slučování toto dělení vracet zpět.

Sloučení datového bloku provedeme v závislosti na jeho aktuální obsazenosti. Hranice pro sloučení je udávána v procentech. Na základě experimentů je doporučována hranice 70 %. Hodnoty nad 80 % vedou ke špatnému výkonu. (Nievergelt a Hinterberger 1984).

Implementace grid souboru bude představena v kapitole 0.

6 Implementace

V této kapitole budou nejprve popsány požadavky na implementační technologii. Následně budou analyzovány některé technologie, pomocí kterých by bylo možné vizualizace evolucí algoritmů pracujících nad vybranými datovými strukturami implementovat. Blíže bude popsána zvolená technologie – JavaFX. V následující části bude představen koncept vizualizací a jejich obecný implementační rámec. Implementace vybraných datových struktur a jejich vizualizací bude předmětem až následujících kapitol.

6.1 Výběr technologie

Před začátkem implementování jednotlivých datových struktur a jejich vizualizací bylo nutné nejprve zvolit vhodnou implementační technologii. V této kapitole budou postupně popsány požadavky na volenou technologii, dále budou stručně představeny uvažované technologie, a nakonec bude blíže popsána zvolená technologie – JavaFX.

6.1.1 Požadavky

Volba vhodné implementační technologie je zcela zásadní. Stěžejním požadavkem na implementační technologii je realizace vizualizací vybraných datových struktur ve webové aplikaci. Tímto požadavkem se okruh možných technologií poněkud zúžil. Potřebujeme technologii, která poběží ve webovém prohlížeči a současně bude efektivně umožňovat vizualizaci (animaci) datových struktur. V potaz může být bráno i využití naimplementovaných datových struktur v jiných aplikacích.

Vezmeme-li v potaz cílovou skupinu uživatelů – tedy zejména studenti informačních technologií – vyvstává zde další požadavek na volenou technologii. Spousta studentů IT nepoužívá současně nejrozšířenější operační systém Windows, ale často se v tomto prostředí setkáváme s některými distribucemi Linuxu, ale například i s Mac OS. Dalším požadavkem je tedy pochopitelně podpora co nejvíce platforem – multiplatformnost. Ačkoli by u cílové skupiny neměl být problém s využíváním moderních technologií, budeme vyžadovat, aby byla zvolená technologie dostatečně rozšířená a „zažitá“.

Vzhledem k účelu práce – podpora výuky datových struktur – očekáváme její využívání i několik let dopředu. Zvolená technologie tedy musí být jak „zažitá“, tak musí mít i budoucnost. Nebudeme volit technologii, která již ustupuje novým technologiím.

Shrneme-li výše popsané požadavky, jedná se o tyto:

1) běh ve webovém prohlížeči v rámci webové aplikace,
2) snadná tvorba vizualizací (animací),
3) vhodnost pro implementaci datových struktur (jejich případné další využití, OOP),
4) multiplatformnost,
5) rozšířenost technologie,
6) budoucnost technologie.

6.1.2 Analyzované technologie

Vzhledem k výše uvedeným požadavkům byly vybrány následující technologie, které byly podrobněji analyzovány:

- HTML5 + JavaScript,
- Dart,
- Adobe Flash,
- Apache Flex,
- Microsoft Silverlight,
- JavaFX.

Dle prvního požadavku (běh ve webovém prohlížeči) se jako první možné řešení nabízí využít nativní jazyky pro webové prohlížeče – HTML a JavaScript. Implementace touto cestou by splňovala téměř všechny kladené požadavky. Na straně klienta navíc není vyžadován jakýkoliv další software kromě kompatibilního webového prohlížeče. Otázka multiplatformnosti, rozšířenosti a budoucnosti této technologie také není problém. Jistým problémem by mohlo být řešení některých nekompatibilit napříč různými webovými prohlížečemi. (Wähner 2012). Co se týče podpory animací, jsou na tom jiné technologie lépe. Navzdory všem výše uvedeným výhodám nebyla implementace touto cestou zvolena, a to zejména kvůli tomu, že JavaScript je prototypově-založený (prototype-based) jazyk. (Details of the object model 2014). Většina dnešních moderních jazyků je třídně-založená (class-based). (Class-based programming 2014). Implementace datových struktur v JavaScriptu tedy není dle mého názoru nejlepší volbou.

Nedostatky JavaScriptu, uvedené výše, odstraňuje velmi mladý programovací jazyk Dart. První stabilní verze byla představena 14. 11. 2013. Hlavním cílem Dartu je nahradit ve webových prohlížečích poměrně starý JavaScript (první verze byla představena v roce 1995). (Soukup 2014). Dart je na rozdíl od JavaScriptu, tak jako většina moderních jazyků, class-based. Ačkoli se jedná o velmi mladou technologii, neměl by být problém s rozšířeností – Dart lze kompilovat do JavaScriptu, a tak může běžet i v současných webových prohlížečích. Vedle možnosti kompilace do JS by se postupně měla rozšířit i nativní podpora v prohlížečích. Tato technologie však nebyla také zvolena, a to proto, že v době zadání této práce ještě nebyla ani v první stabilní verzi. (Dart 2014).

Další uvažovanou technologií je Adobe Flash. Rozšířenost této technologie je značná, avšak její budoucnost není zcela jasná. Najdou se oblasti, kde najde své uplatnění i nadále, avšak obecným trendem je nahrazování této technologie novým standardem HTML5 – zejména v oblasti přehrávání videa na webu. (Bright 2012). Dalším nemilým faktem je ustoupení od podpory mobilních operačních systémů. Distribuovat Flash aplikace na mobilní platformy však lze pomocí Adobe AIR (Adobe Integrated Runtime). (Ulanoff 2011). Perspektivnější technologií než Adobe Flash se zdá být následující popisovaná technologie – Apache Flex (dříve Adobe Flex). (Jackson 2011).

Velmi zajímavou technologií je Apache Flex. Jedná se o open source aplikační framework, který umožňuje jak vývoj klasických desktopových aplikací, tak webových aplikací, ale i aplikací pro mobilní zařízení. Vývoj aplikací ve Flexu slibuje jeden stejný kód pro všechny cílové platformy – Windows, Linux, Mac OS, Android, iOS i BlackBerry. Podpora pro Windows Phone prozatím chybí. Pro běh aplikací je vyžadováno běhové prostředí Adobe AIR. Kdyby nebyla zvolena JavaFX, Apache Flex by byl hned dalším kandidátem. (Apache Flex 2014).

Poslední uvažovanou technologií je Microsoft Silverlight. Hlavním problémem této technologie je její multiplatformnost. Jelikož se jedná o produkt firmy Microsoft, je tato technologie cílena zejména na rodinu operačních systémů Windows. Oficiálně je kromě OS Windows také podporován Mac OS. Pro Linux existují open source implementace Silverlightu. Na poli mobilních zařízení je podporována pouze platforma Windows Phone. Navíc se dle některých zdrojů jedná o technologii, která již nemá budoucnost. (Wähner 2012, Epperson 2013).

Nyní zbývá podrobněji představit technologii JavaFX, jakožto zvolenou implementační technologii.

6.1.3 JavaFX

JavaFX je dalším evolučním krokem vývoje platformy Java. Jedná se o sadu nástrojů pro vývoj bohatých multiplatformních grafických uživatelských rozhraní (GUI). Grafická rozhraní aplikací tvořených pomocí JavaFX využívají potenciálu moderních grafických karet – jsou hardwarové akcelerované. V první verzi, která byla uvolněna 4. prosince 2008, se aplikace psaly pomocí jazyku JavaFX Script. Od verze 2.0 byla JavaFX kompletně přepsána do jazyku Java, z čehož plyne zásadní výhoda – je možné využívat existující Java knihovny a Java API. (Dea 2011). Do budoucna by měla JavaFX kompletně nahradit knihovnu Swing. Poslední stabilní verzí je JavaFX 2.2. Tato verze je standardní součástí JRE a JDK 7 od aktualizace 6 (7u6). Od JDK 8 bude dostupná nová verze JavaFX 8.

Mezi hlavní vlastnosti JavaFX patří tyto:

- hardwarově akcelerovaná grafika;
- tvorba GUI pomocí značkovacího jazyka FXML;
- přes 60 grafických komponent včetně grafů a webové komponenty;
- podpora CSS stylování (snadná kompletní změna vzhledu aplikace);
- podpora multimédií, animací a různých grafických efektů;
- možnost využití ve Swing aplikacích.

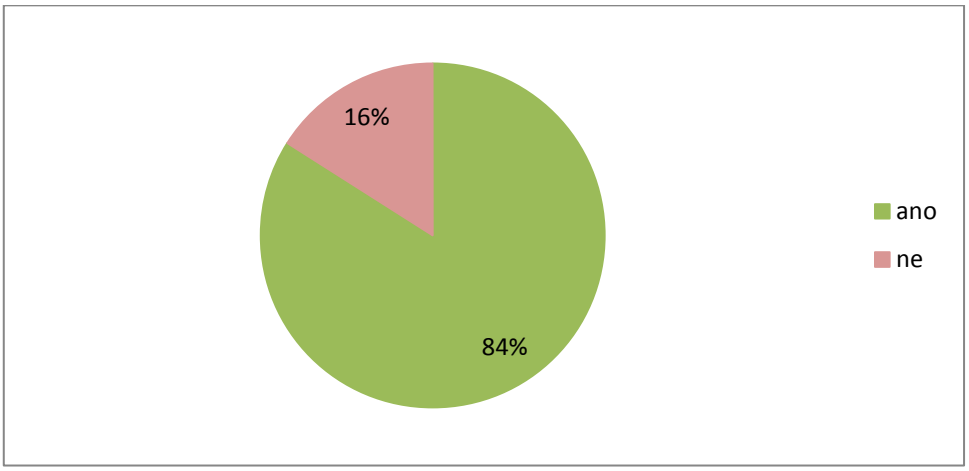
JavaFX, stejně jako standardní Java SE, je nezávislá na platformě. Lze ji provozovat jak na operačních systémech Windows, Linux, tak i na Mac OS. Na těchto platformách může běžet jako standardní desktopová aplikace, ale i jako aplikace spuštěná přímo ve webovém prohlížeči. Podporovány jsou nejčastěji používané prohlížeče – Internet Explorer, Mozilla Firefox, Google Chrome a Safari. Co se týče mobilních platforem, není prozatím JavaFX

na žádné oficiálně podporována. Nicméně již dnes jsou možnosti, jak JavaFX aplikace provozovat na operačním systému Android[9] a iOS[10]. (JavaFX FAQ 2014).

Několik důvodů, proč byla zvolena právě JavaFX:

- běh ve webovém prohlížeči i jako standardní desktopová aplikace,
- multiplatformnost,
- vynikající podpora animací,
- možnost využívat standardní Java knihovny,
- možnost dalšího využití naimplementovaných datových struktur,
- rozšířenost (viz níže),
- budoucnost a rozvoj technologie.

Podpora Javy ve webových prohlížečích je dle Java-support Statistics (2014) následující:



Obrázek 21 – Dostupnost technologie Java ve webových prohlížečích
Zdroj: Java-support Statistics (2014)

Data pochází z ledna 2014 a jsou získána z Google Analytics. Velikost analyzovaného vzorku není bohužel známa, nicméně jiná data nejsou k dispozici. Budeme-li předpokládat, že velikost vzorku je dostatečná, není 84% dostupnost Javy ve webových prohlížečích vůbec špatný výsledek. Data ovšem postrádají informaci o nainstalované verzi, tudíž nelze s jistotou říci, že v celých 84 % je dostupná i JavaFX.

6.2 Implementace vizualizací

Cílem práce je vizualizovat vybrané datové struktury a jejich vývoj v průběhu provádění jednotlivých algoritmů. Pod pojmem vizualizace rozumíme názorné grafické zobrazení. Z vizualizace datové struktury by měla být patrná její organizace. Tato organizace však nemusí nutně být v souladu s konkrétní paměťovou reprezentací, ale může např. respektovat grafické znázornění využívané při teoretickém přestavování dané datové struktury.

[9] https://bitbucket.org/javafxports/android/wiki/Home
[10] http://www.robovm.org

Vizualizací evoluce dané datové struktury je i provádění jednotlivých změn po skocích. V této práci nebude vizualizace evoluce prováděna po skocích, ale bude plynulá – bude se tedy jednat o animace.

6.2.1 Filozofie

Před začátkem implementace vizualizací bylo nutné zvolit přístup, kterým budou vizualizace prováděny. V zásadě byly uvažovány dva zcela odlišné přístupy:

- využití vláken,
- generování událostí.

První přístup – využití vláken – byl vyzkoušen jako první, ale následně od něj bylo upuštěno. Princip tohoto způsobu spočívá ve „vpuštění“ vlákna do datové struktury a jeho následné pozastavování v určitých částech algoritmů. Mezi jednotlivými pozastaveními vlákna docházelo v jiném vláknu ke grafickým změnám vizualizované datové struktury. Problémy tohoto přístupu jsou uspávání vlákna ve struktuře, řešení předčasného ukončení vizualizace (reset) a další. Nepěkným důsledkem tohoto přístupu je také poměrně těsná vazba mezi datovou strukturou a její vizualizací, proto je v práci využit druhý způsob.

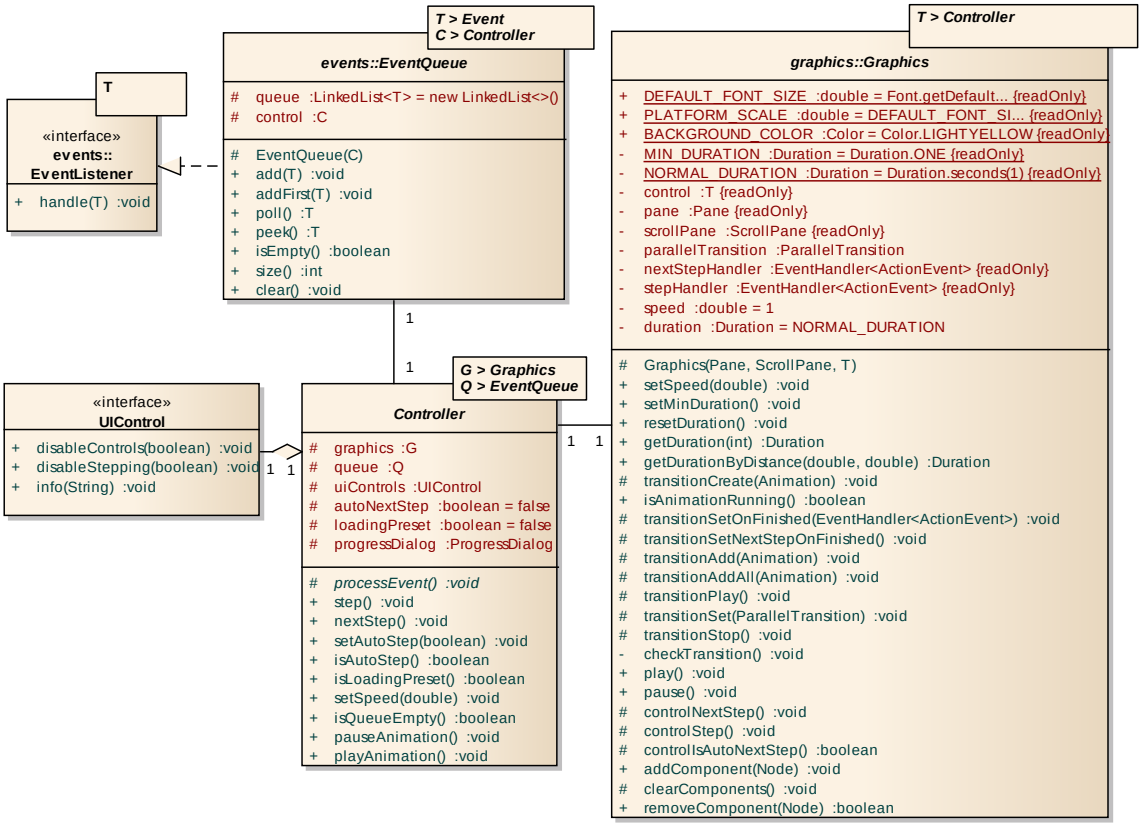
Druhý přístup – generování událostí – odstraňuje nedostatky výše uvedeného způsobu, a to zejména úzké propojení mezi datovou strukturou a její vizualizací. Princip tohoto přístupu spočívá v generování událostí v různých částech algoritmů dané datové struktury. K datové struktuře je možné zaregistrovat posluchače generovaných událostí. Struktura je tedy i samostatně funkčním celkem. Zaregistrovaný posluchač událostí postupně ukládá přijaté události do fronty. Po odstartování vizualizace je vybrána první událost, která je adekvátně obsloužena (animována). V závislosti na nastavení krokování se z fronty automaticky vybere další událost a opět se zpracuje. Takto se pokračuje, dokud není fronta prázdná – vizualizace určité operace byla dokončena. Po provedení další operace je fronta opět naplněna a pokračuje se výše uvedeným způsobem dále.

Jak již bylo uvedeno výše, vizualizace v této práci budou založeny na principu generování událostí. Pro všechny vizualizované struktury byl vytvořen společný koncept, který bude představen v následující kapitole.

6.2.2 Koncept

Pro všechny vizualizované datové struktury byl vytvořen společný koncept, který usnadňuje následnou vizualizaci jednotlivých datových struktur. Tento koncept je součástí samostatného projektu, který je následně importován formou knihovny do jednotlivých projektů. V tomto projektu se také nachází další pomocné třídy – např. některé grafické elementy, pomocné animační třídy apod.

Na UML diagramu tříd níže se nachází struktura navrženého konceptu.



Obrázek 22 – UML diagram tříd konceptu animací

Zdroj: vlastní

Jádrem všech animací je třída `Controller`. Tato třída je zodpovědná za výběr událostí z fronty `EventQueue` a předání vybrané události ke zpracování. Obsloužení vybrané události zajistí implementace metody `processEvent()`. Dále tato třída zajišťuje aktivaci a deaktivaci ovládacích prvků uživatelského rozhraní (prostřednictvím rozhraní `UIControl`). Metody `step()` a `nextStep()` zajišťují krokování či plynulou animaci. Pomocí této třídy je také pozastavována probíhající animace a nastavována její rychlost. Potomci této třídy již obsahují konkrétní datovou strukturou, na kterou registrují posluchače událostí – konkrétního potomka třídy `EventQueue`. Vizualizovaná datová struktura je pomocí třídy `Controller` také ovládána.

Jak název napovídá, potomci třídy `EventQueue` ukládají do fronty přijaté události z konkrétní datové struktury. Současně se zde řeší počáteční rozběhnutí vizualizace (zpracování první události z fronty po přijetí určité události).

Třída `Graphics` je předkem všech tříd, které provádějí animace konkrétních datových struktur. Nejdůležitějším atributem této třídy je atribut `parallelTransition` – reference na probíhající animaci. Pomocí této reference můžeme probíhající animaci pozastavovat či měnit její rychlost. Dalším důležitým atributem je `scrollPane` a v něm obsažený `pane`, kam jsou vkládány všechny grafické elementy.

6.2.3 Obecné funkcionality

Ještě před přistoupením k popisu jednotlivých implementací vizualizací, je nutné představit některé společné funkce, kterými všechny vizualizace (animace) disponují. Bližší popis uživatelského rozhraní se nachází v příloze. Všechny vizualizace disponují:

- možnost načítat předpřipravená data,
- možnost generování náhodných dat,
- informační dialog s popisem ovládání (klávesové zkratky),
- ovládání rychlosti animace,
- možnost plynulé animace či provádění animace po krocích,
- možnost pozastavit animaci v libovolném čase.

Během studování popisů jednotlivých algoritmů se současně doporučuje využívat vytvořené vizualizace. Na začátku popisu každé datové struktury se nachází informace o tom, s jakými daty a s jakým nastavením byly popisované příklady prováděny.

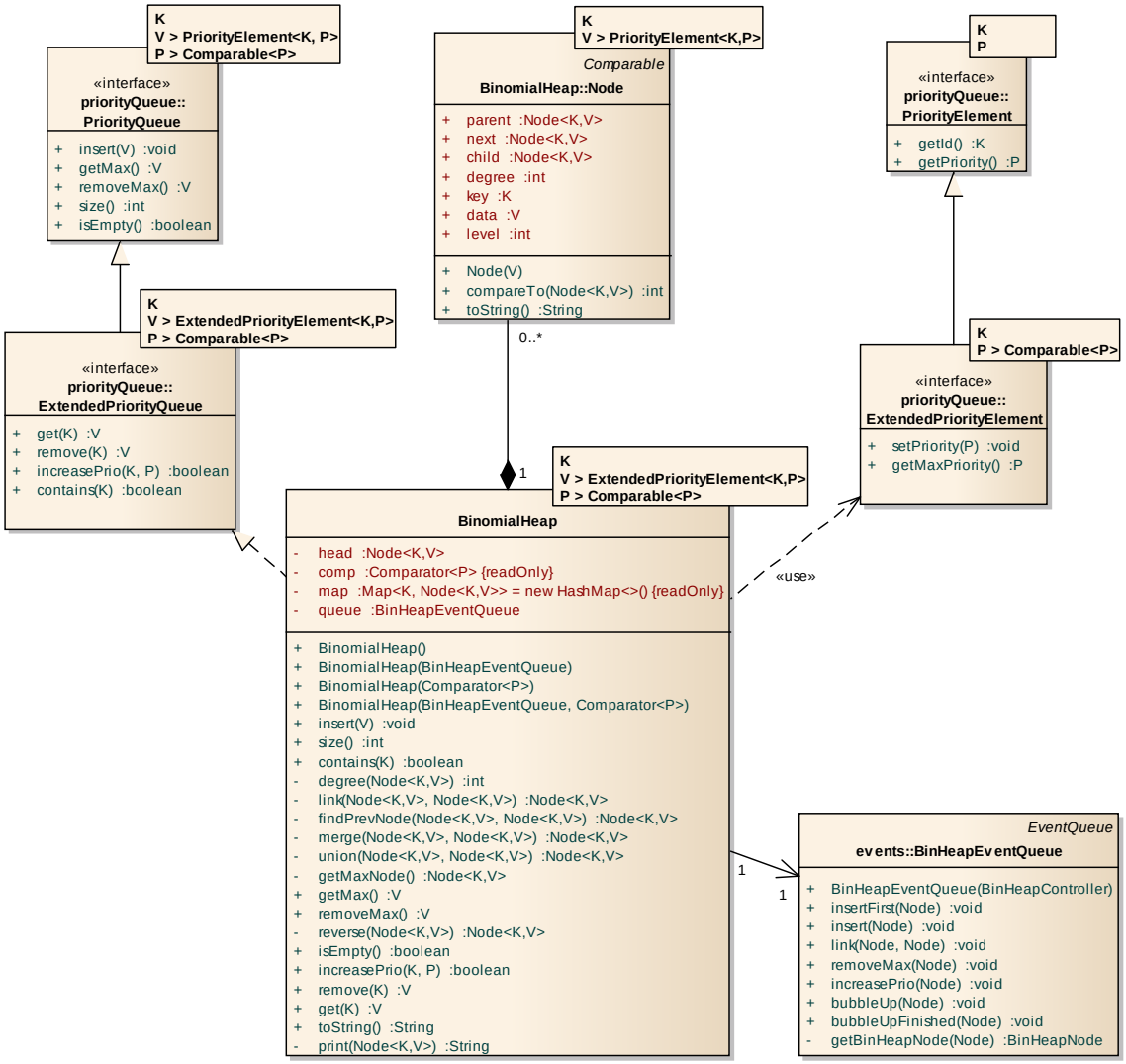
7 Implementace binomické haldy

Tato kapitola se bude podrobně zabývat implementací binomické haldy. Nejprve bude prezentována struktura a paměťová reprezentace implementované binomické haldy, poté bude popsáno fungování jednotlivých algoritmů.

Definice binomické haldy téměř přesně určuje veškeré implementační detaily. Implementační „volnost“ oproti jiným strukturám (např. B-strom a grid soubor) je tedy poměrně malá. Binomická halda disponuje hlavou, prvky haldy jsou provázány pomocí explicitních referencí. Kořeny jednotlivých binomických stromů jsou uchovávány ve spojovém seznamu. Jistou implementační modifikací může být udržování explicitní reference na prvek s maximální prioritou.

Implementovaná halda může být jak min-haldou, tak max-haldou. Implicitně je vytvořena min-halda. Chceme-li vytvořit max-haldu, musíme při její konstrukci předat příslušnou instanci třídy `Comparator`.

Na následujícím UML diagramu tříd je vyobrazena struktura implementované binomické haldy.



Obrázek 23 – UML diagram tříd binomické haldy

Zdroj: vlastní

Binomická halda je tzv. slučitelná halda, tudíž disponuje i operacemi zvýšení priority a odebrání libovolného prvku. Proto implementovaná binomická halda implementuje rozhraní `ExtendedPriorityQueue`. Do haldy musí být vkládány prvky, které implementují rozhraní `ExtendedPriorityElement`. Tyto prvky v sobě navíc nesou informaci o maximální možné prioritě a umožňují prioritu měnit (pouze zvyšovat). Vyžadovány jsou tři typové parametry:

- K – specifikuje datový typ klíče,
- V – specifikuje datový typ ukládaných prvků,
- P – specifikuje datový typ priority.

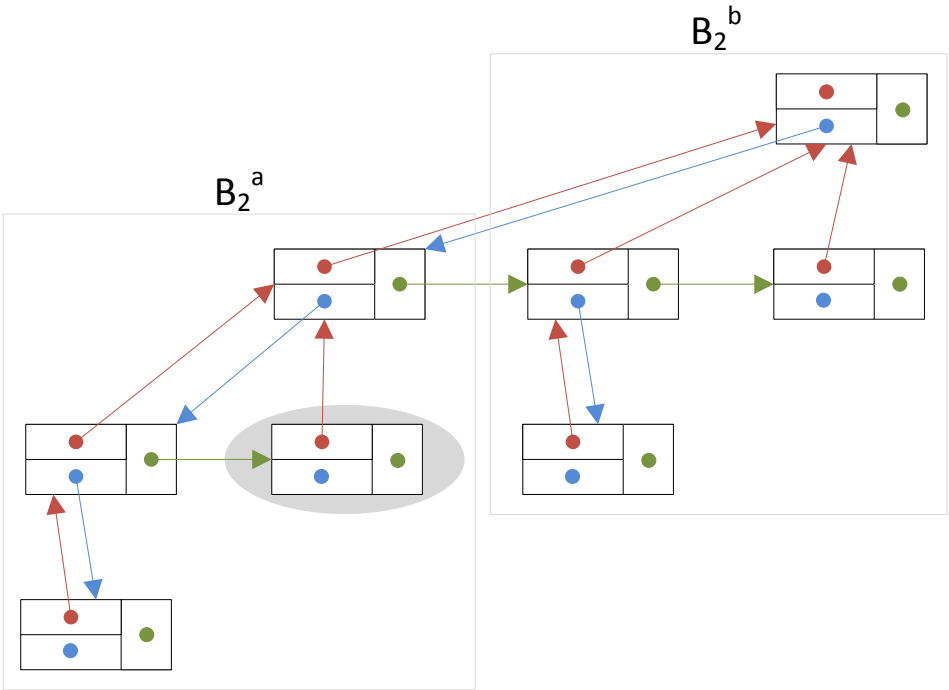
Nejdůležitější uchovávanou informaci ve třídě `BinomialHeap` je její hlava `head`, ve které se udržuje reference na první binomický strom. Aby bylo možné vytvářet jednak min-haldu, jednak max-haldu, disponuje halda komparátorem `comp`, pomocí kterého jsou prvky haldy porovnávány. Při operacích zvyšování priority prvku nebo odebírání prvku

z haldy se přistupuje k libovolnému prvku v haldě. Jelikož operace vyhledání libovolného prvku v binomcké haldě je poměrně časově náročná, obvykle se využívá přístupová struktura. Ta zajistí okamžitý přístup k libovolnému prvku přes klíč. V této implementaci se využívá hešovací tabulka, konkrétně instance třídy `HashMap`, která je součástí Java Collections Frameworku. Pro účely animace obsahuje halda frontu událostí `queue`, kam jsou v průběhu provádění algoritmů vkládány jednotlivé události.

Vnitřní třída `Node` reprezentuje uzel (prvek) haldy. Z těchto uzlů se staví dílčí binomické stromy. V uzlu je kromě reference na rodiče, následující prvek a prvního potomka uchováván také jeho stupeň (vezmeme-li tento uzel jako kořen binomického stromu). Součástí uzlu je i hloubka zanoření, ta však slouží pouze pro účely textových výpisů. Uzel samozřejmě disponuje klíčem a přidruženými daty.

7.1 Paměťová reprezentace

Realizace binomické haldy je založena pouze na explicitních referencích na ostatní prvky haldy. Ilustraci paměťové reprezentace binomické haldy zobrazuje obrázek 24. Červeně jsou znázorněny reference na rodiče, modře reference na prvního syna a zeleně reference na další prvek.



Obrázek 24 – Ilustrace paměťové reprezentace binomické haldy

Zdroj: vlastní

Na výše uvedeném obrázku je důležité si povšimnout toho, že podbarvený prvek nedisponuje referencí na další prvek, ačkoli se vizuálně napravo od tohoto prvku nachází jiný prvek. Podbarvený prvek je binomický strom řádu 0, který je součástí binomického stromu řádu 2 (B_2^a). Tento binomický strom B_2^a byl spojen s druhým binomickým stromem řádu 2 (B_2^b) a vznikl tak zobrazený binomický strom řádu 3. Během operace

spojení dvou binomických stromů se upravují pouze reference týkající se kořenů stromů B_2^a a B_2^b.

V binomické haldě uchováváme pouze hlavu (referenci na první binomický strom), kořeny dalších binomických stromů jsou uchovány v lineárně zřetězeném seznamu. Zřetězení se realizuje pomocí reference na další prvek.

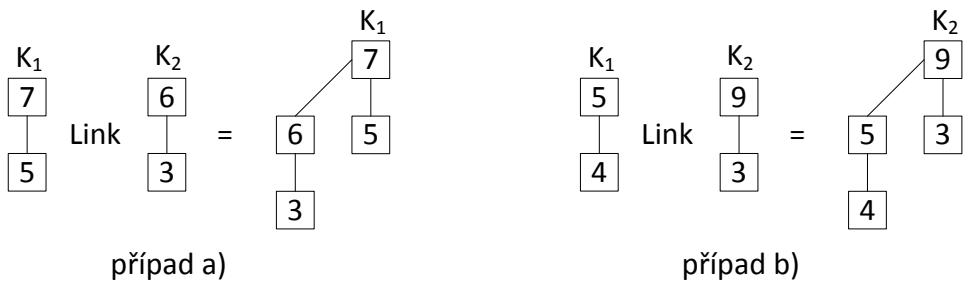
Následující kapitoly budou pojednávat o jednotlivých operacích binomické haldy. Nejprve budou rozebrány pomocné operace *LINK*, *MERGE* a *SJEDNOCENÍ*, které jsou využívány dalšími operacemi. Pokud nebude uvedeno jinak, halda bude automaticky považována za max-haldu. Implementace této haldy byla provedena dle studijních materiálů (Kavička 2011).

7.2 Operace Link

Při operaci *LINK* dochází ke spojení dvou haldově uspořádaných binomických stromů stejných řádů s kořeny K_1 a K_2 do nového binomického haldově uspořádaného stromu řádu vyššího. Haldové uspořádání nového stromu je zajištěno pomocí dvou následujících pravidel:

a) je-li priorita kořene K_1 vyšší nebo rovna prioritě kořene K_2, pak se K_2 stane synem K_1;
b) v opačném případě se K_1 stane synem K_2.

Následující obrázek demonstruje oba případy, které mohou při operaci *LINK* nastat. Operace *LINK* je aplikována na dva binomické stromy řádu 1.



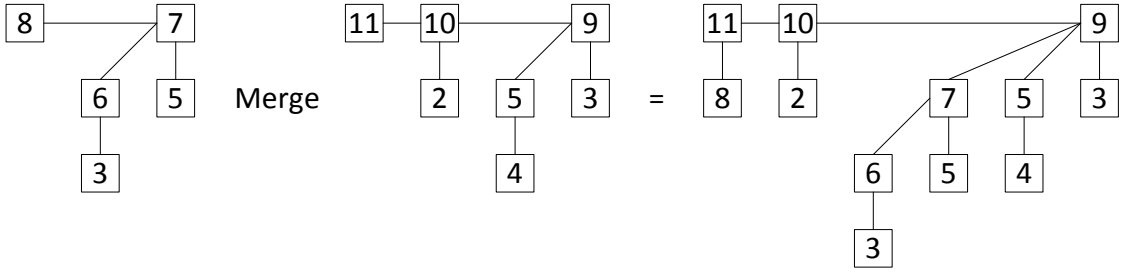
Obrázek 25 – Binomická halda – ilustrace operace Link

Zdroj: vlastní

7.3 Operace Merge

Druhá pomocná operace *MERGE* se využívá pro spojení dvou binomických lesů L_1 a L_2. Binomické stromy v těchto lesích musejí být vzestupně uspořádány dle svých řádů. Samotné slučování probíhá podobnou strategií jako u algoritmu mergesort. Z binomických lesů L_1 a L_2 se postupně odebírají binomické stromy tak, aby byl vždy odebrán binomický strom s nižším řádem. Pokud se nelze rozhodnout, z kterého lesa strom odebrat (v obou lesích se nacházejí binomické stromy stejných řádů), jsou tyto dva stromy spojeny pomocí operace *LINK* a zařazeny do výsledného lesa, stejně tak jako ostatní odebrané stromy.

Výsledný les však nemusí být nutně binomický – mohou se v něm vyskytovat binomické stromy stejných řádů. Následující obrázek zachycuje spojení dvou binomických hald operací *MERGE*.



Obrázek 26 – Binomická halda – ilustrace operace Merge

Zdroj: vlastní

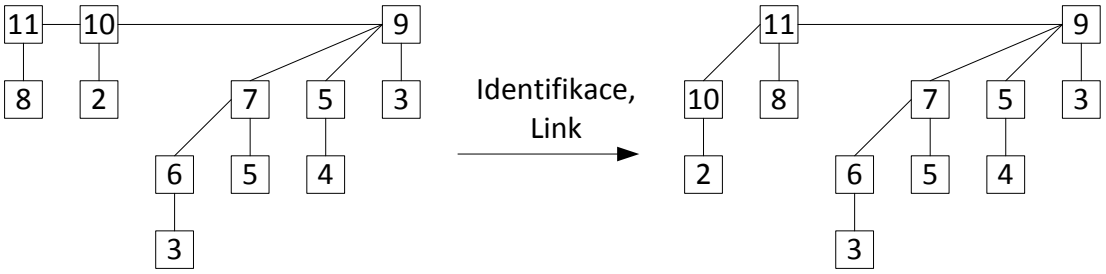
Průběh algoritmu dle obrázku 26 je následující:

1) Na počátku obou hald je binomický strom 0. řádu, proto se provede operace *LINK* a výsledný strom B_1 je zařazen do výsledné haldy.
2) Z druhé haldy je odebrán binomický strom řádu 1, protože na začátku první haldy se nachází binomický strom řádu vyššího (2). Priorita odebraného kořene je 10. Strom je zařazen nakonec výsledné haldy.
3) Na počátku obou hald se nyní opět vyskytují binomické stromy stejných řádů (B_2). Stromy jsou spojeny pomocí operace *LINK* a jsou opět zařazeny nakonec výsledné haldy.
4) Obě vstupní haldy jsou nyní prázdné, algoritmus končí.

7.4 Operace Sjednocení

Operace *SJEDNOCENÍ* sloučí dvě vstupní binomické haldy do jedné binomické haldy využitím operací *LINK* a *MERGE*. Produktem operace *MERGE* je potenciálně nebinomický les (viz obrázek 26 výše). Následuje sekvenční průchod tohoto lesa, ve kterém zjišťujeme výskyty binomických stromů stejných řádů. Případný výskyt binomických stromů stejných řádů eliminujeme aplikováním operace *LINK*. Po provedení těchto operací opět získáme binomickou haldu.

Budeme pokračovat v příkladu naznačeném na obrázku 26. Na tomto obrázku je zachycena první fáze operace *SJEDNOCENÍ* – spojení dvou binomických hald pomocí operace *MERGE*. Na následujícím obrázku se nachází ilustrace druhé fáze operace *SJEDNOCENÍ* – aplikace operace *LINK* na binomické stromy stejných řádů.



Obrázek 27 – Binomická halda – ilustrace operace Link v rámci operace Sjednocení
Zdroj: vlastní

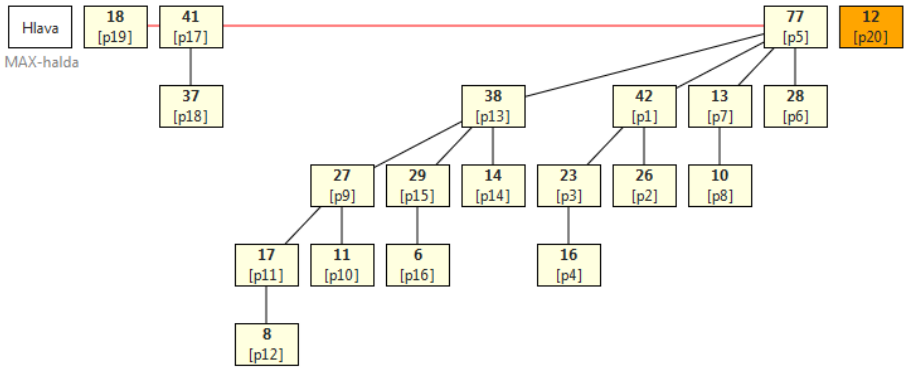
Na vstupu se nachází nebinomický les – obsahuje dva binomické stromy 1. řádu. Hodnoty priorit jejich kořenů jsou 11 a 10. Na tyto dva stromy je aplikována operace *LINK*. Na výstupu dostáváme binomickou haldu.

Následující kapitoly již popisují operace definované rozhraním binomické haldy.

7.5 Operace Vlož

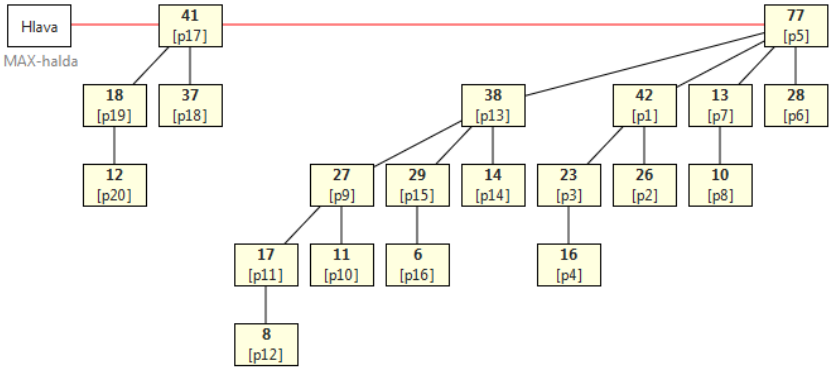
Operace vlož je jednoduše implementována s využitím operace *SJEDNOCENÍ*. Z vkládaného prvku se vytvoří nová binomická halda disponující jedním binomickým stromem (řádu nula). Původní halda se s nově vytvořenou haldou spojí pomocí operace *SJEDNOCENÍ*.

Mějme haldu zobrazenou na následujícím obrázku (konkrétně se jedná o max-haldu po vložení 19. prvku z předpřipravené sady číslo 6). Na obrázku je zvýrazněn vkládaný prvek *p20* s prioritou12.



Obrázek 28 – Binomická halda před vložením prvku p20
Zdroj: vlastní

Na začátku původní haldy se nachází binomický strom 0. řádu. Vkládaný prvek je také binomický strom 0. řádu. Během operace *MERGE* (v rámci operace *SJEDNOCENÍ*) jsou tyto binomické stromy spojeny. Následuje sekvenční průchod všemi binomickými stromy a případná aplikace operace *LINK*. Výsledný stav po vložení prvku *p20* zachycuje obrázek 29.



Obrázek 29 – Binomická halda po vložení prvku p20

Zdroj: vlastní

7.6 Operace Zpřístupni maximum

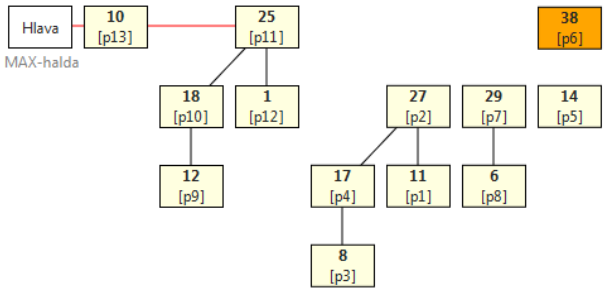
Pokud je v konkrétní implementaci haldy stále udržována reference na prvek s maximální prioritou, je tento prvek ihned vrácen. V opačném případě se provede sekvenční průchod všemi lineárně zřetězenými kořeny haldově uspořádaných stromů a je nalezen a vrácen prvek s maximální prioritou.

Tato implementace neudržuje referenci na prvek s maximální prioritou. Operace *ZPŘÍSTUPNIMAX* není vizualizována, neboť se jedná o triviální operaci.

7.7 Operace Odeber maximum

Během odebírání maxima z haldy se nejprve provede operace *ZPŘÍSTUPNIMAX*. Binomický strom, jehož kořenem je zpřístupněný prvek s maximální prioritou, je odebrán z binomické haldy. Následně se z tohoto binomického stromu oddělí kořen. Potomci odděleného kořene představují binomické stromy, které jsou zřetězeny tak, aby představovaly binomickou haldu. Následuje aplikování operace *SJEDNOCENÍ* na původní a nově vzniklou haldu.

Na obrázku níže se nachází binomická halda s daty ze sady 1. Na této max-haldě je provedena operace *ODEBERMAX* – maximem je zvýrazněný prvek *p6*.

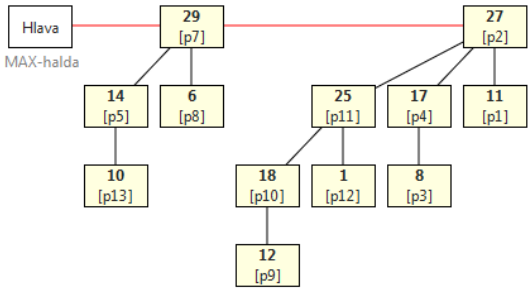


Obrázek 30 – Odebrání maxima z binomické haldy

Zdroj: vlastní

Nejprve se z binomické haldy odebere binomický strom s kořenem *p6*, následně je prvek *p6* (kořen) oddělen od potomků. Oddělení potomci *p2*, *p7* a *p5* jsou uspořádány do nové

binomické haldy. Na původní a novou binomickou haldu je aplikována operace *SJEDNOCENÍ*, jejíž výsledek je zobrazen na obrázku níže.



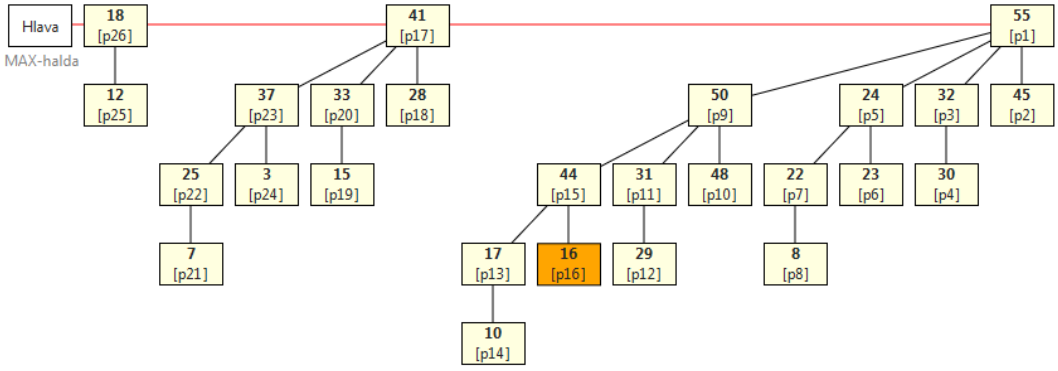
Obrázek 31 – Binomická halda po odebrání maxima

Zdroj: vlastní

7.8 Operace Zvyš prioritu

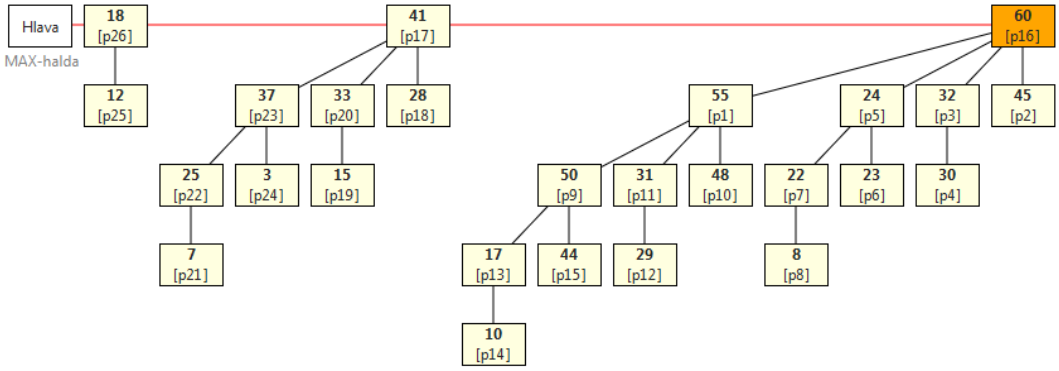
Prioritu prvku lze pouze zvýšit (v případě max-hald). Zvýšení priority prvku probíhá postupnými záměnami tohoto prvku s jeho rodičem, dokud je priorita rodiče nižší, než nová priorita prvku.

Následující dva obrázky zachycují zvýšení priority prvku *p16* z priority 16 na 60. Prvek *p16* je postupně zaměňován se svým rodičem, až se nakonec dostane nejvýše – do kořene binomického stromu.



Obrázek 32 – Binomická halda před zvýšením priority prvku

Zdroj: vlastní



Obrázek 33 – Binomická halda po zvýšení priority prvku

Zdroj: vlastní

7.9 Operace Odeber

Operaci odebrání libovolného prvku lze jednoduše naimplementovat pomocí dvou výše popsaných operací:

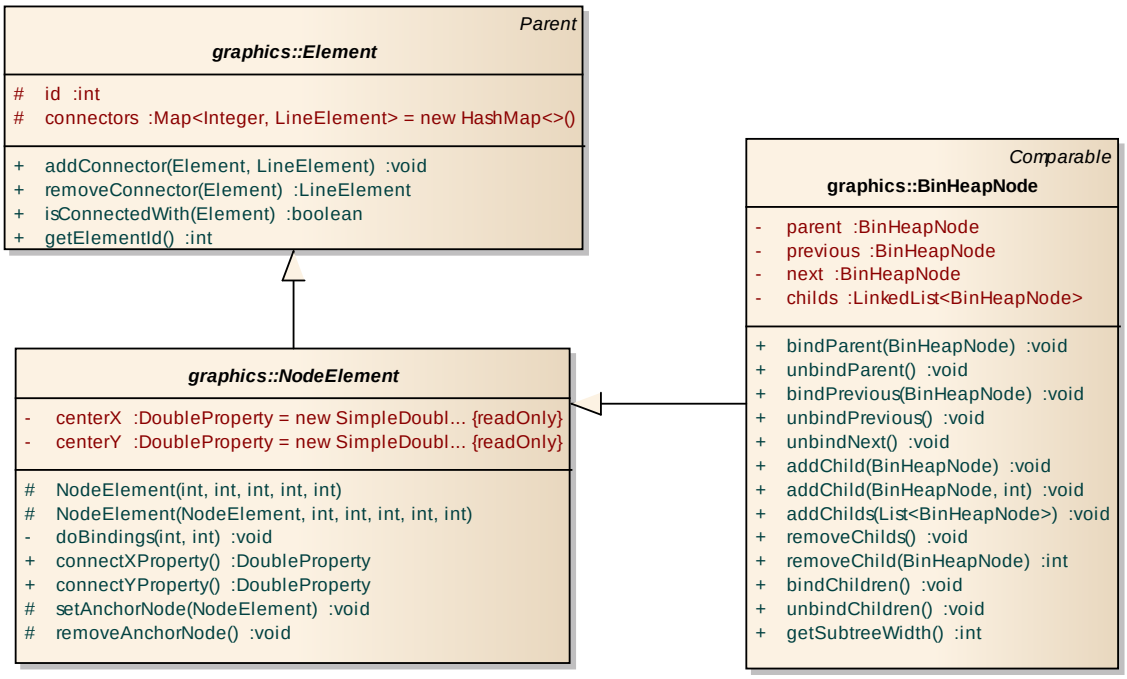
- *ZVYŠPRIORITU* a
- *ODEBERMAX*.

Odebíranému prvku se nejprve nastaví nejvyšší možná priorita. Této priority prvky v haldě prakticky nikdy nenabývají. Následuje odebrání prvku s maximální prioritou.

7.10 Grafické rozmístění prvků

Grafické zarovnání prvků binomické haldy je řešeno pomocí vázání (bindingu) pozic prvků. Každý prvek je navázán k rodičovskému prvku. Rodičovský a navázaný prvek mají udržovánu vzájemnou relativní polohu stále stejnou. Pokud se hýbe s rodičem, současně se pohybuje i navázaný prvek. Takto jsou na sebe navázány všechny grafické uzly binomické haldy. Při reorganizacích struktury dochází k dočasnému rozpoutání vazeb. Po dokončení reorganizačních animací jsou vazby opět obnoveny tak, aby při dalších operacích zachovávala binomická halda svůj „tvar“.

Na obrázku 34 se nachází zjednodušený UML diagram třídy `BinHeapNode`, která zajišťuje grafickou reprezentaci uzlu binomické haldy. V digramu jsou zobrazeny pouze atributy a metody související s pozicováním uzlů, ostatní atributy a metody jsou kvůli zjednodušení vypuštěny.



Obrázek 34 – Zjednodušený UML diagram třídy BinHeapNode

Zdroj: vlastní

Každý grafický uzel uchovává reference na všechny své potomky (nejedná-li se o list). Kořeny jednotlivých binomických stromů dále obsahují odkaz na předchozí uzel (nalevo) a následující uzel (napravo). Všechny uzly, kromě kořenů, také disponují referencí na svého rodiče.

Metody navazující uzly na sebe (`bindParent`, `bindPrevious` a `bindChildren`) využívají metodu `setAnchorNode`, jejíž kód je uveden níže.

```
/**
 * Navazani pozice na vztazny element.
 * @param anchor vztazny element
 */
protected final void setAnchorNode(NodeElement anchor) {
      // zachovani celych cisel (animace pracuji s realnymi cisly)
      long dx = Math.round(getTranslateX() - anchor.getTranslateX());
      long dy = Math.round(getTranslateY() - anchor.getTranslateY());

      translateXProperty().bind(
            Bindings.add(dx, anchor.translateXProperty())
      );

      translateYProperty().bind(
            Bindings.add(dy, anchor.translateYProperty())
      );
}
```

Při rušení vazby se pouze volá `translateXProperty().unbind();` resp. `translateYProperty().unbind();`

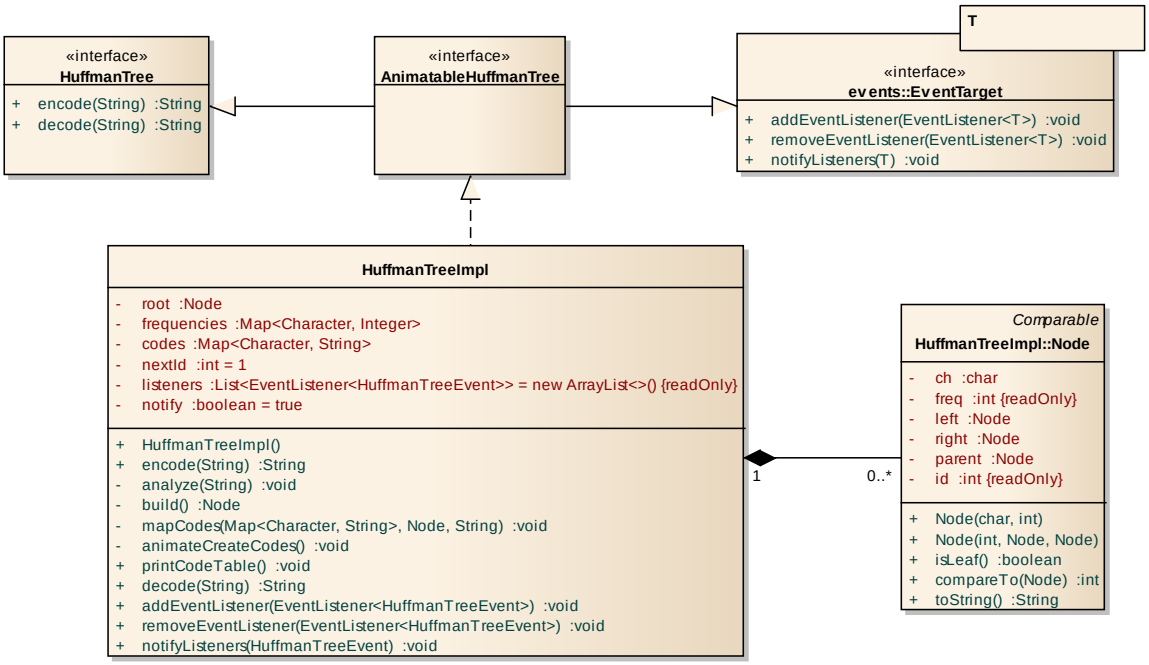
Výše využívané vázání (binding) proměnných JavaFX podporuje již v základu.

8 Implementace Huffmanova stromu

V této kapitole bude popsána implementace Huffmanova kódovacího stromu. Nejprve bude představena struktura, poté bude následovat popis jednotlivých částí algoritmů: proces budování stromu, sestavení kódovací tabulky, průběh kódování, a také průběh dekódování.

Cílem této práce je implementace Huffmanova kódovacího stromu, konkrétně dvoufázového Huffmanova kódování. Stejně jako u binomické haldy, není u dvoufázového Huffmanova kódování více implementačních možností. Na rozdíl od ostatních implementovaných struktur v této práci, je Huffmanovo kódování poměrně statické – proběhne vybudování stromu, tvorba kódovací tabulky, kódování, dekódování, a tím život struktury končí. Nelze dále přidávat nebo odebírat prvky apod.

Na obrázku 35 se nachází zjednodušený UML diagram tříd Huffmanova kódování.



Obrázek 35 – UML diagram tříd Huffmanova stromu

Zdroj: vlastní

Přes rozhraní `AnimatableHuffmanTree` implementuje třída `HuffmanTreeImpl` jak rozhraní Huffmanova kódování, tak i rozhraní umožňující registraci posluchače událostí. V Huffmanově stromu uchováváme jeho kořen, tabulku frekvencí výskytů jednotlivých znaků ve vstupním textu a tabulku přiřazení kódů k jednotlivým znakům. Další atributy jsou využívány k animačním účelům.

Vnitřní uzel Huffmanova stromu obsahuje referenci na levého a pravého potomka a celkovou frekvenci znaků v daném podstromu (nejedná-li se o list). V listech je navíc nastaven znak, kterému přísluší daná frekvence. V listech samozřejmě nejsou nastaveny reference na potomky. Reference na rodiče a ID uzlu jsou zavedeny z animačních důvodů.

Následovat bude popis jednotlivých částí Huffmanova kódování. Implementace Huffmanova kódování byla provedena dle studijních materiálů (Kavička 2011).

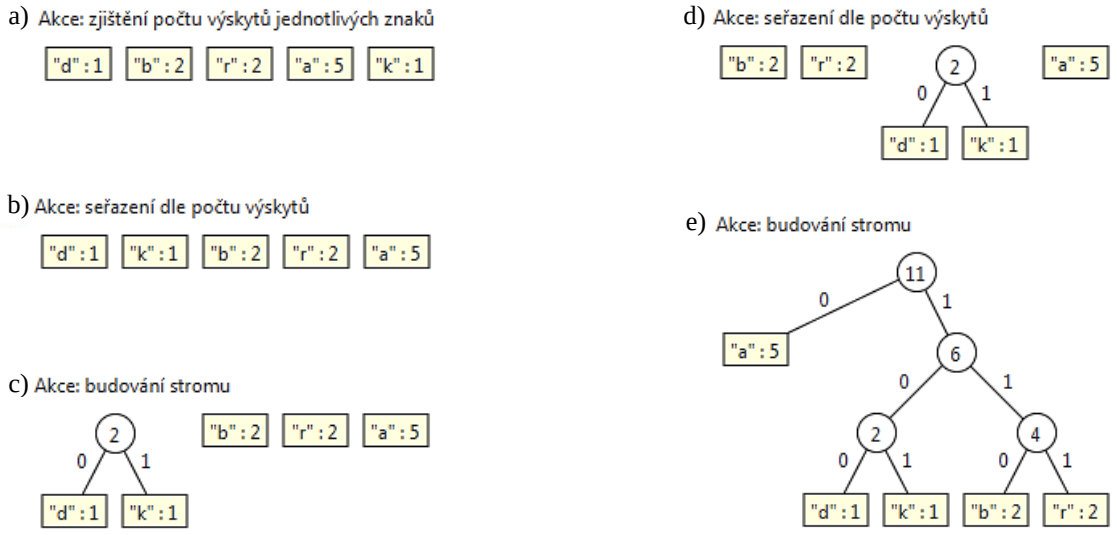
8.1 Budování stromu

Před začátkem budování stromu musíme nejprve zanalyzovat vstupní text. Během průchodu vstupním textem počítáme výskyty jednotlivých znaků. Na základě zjištěných frekvencí výskytů vytvoříme budoucí uzly Huffmanova stromu. Tyto uzly vložíme do prioritní fronty, přičemž prioritou každého prvku je jeho frekvence výskytu v textu. Prvky s nejnižším počtem výskytů se nacházejí na začátku fronty.

Po analýze vstupního textu přistoupíme k samotnému budování stromu. Budování je založeno na následujícím principu. Postupně odebíráme z prioritní fronty uzly po dvou. Těmto uzlům vždy vytvoříme společného rodiče, jehož frekvence je rovna součtu frekvencí obou uzlů. Nový uzel (rodič s dvěma potomky) se vloží zpět do prioritní fronty. Jakmile se v prioritní frontě vyskytuje pouze jeden uzel, algoritmus končí. Jediný zbylý uzel ve frontě je kořenem Huffmanova kódovacího stromu. Postup budování Huffmanova stromu je nastíněn v následujícím pseudokódu:

```
uzly = seznam všech znaků s frekvencemi výskytu;
fronta = prioritní fronta;

vlož všechny uzly do fronty;

dokud fronta obsahuje více než jeden uzel:
      levý = odeber prvek z fronty;
      pravý = odeber prvek z fronty;
      rodič = vytvoř rodiče prvků levý a pravý;
      vlož rodiče do fronty;

kořen = odeber prvek z fronty;
```

Budování Huffmanova stromu pro vstupní řetězec „abrakadabra“ zachycuje obrázek níže.



Obrázek 36 – Postup budování Huffmanova stromu

Zdroj: vlastní

8.2 Tvorba kódů

Během tvorby kódů je vytvořena tabulka, kde klíčem je znak a hodnotou klíče je kód daného znaku. Pro vytvoření kódů je zapotřebí navštívit všechny listy stromu. Popisem cesty z kořene do listu se vytvoří výsledný kód znaku. Cesta začíná vždy v kořenu. Půjdeme-li do levého potomka, zapíšeme do výstupního kódu `0`, půjdeme-li do pravého potomka, zapíšeme `1`. Takto pokračujeme, dokud se nedostaneme do listu.

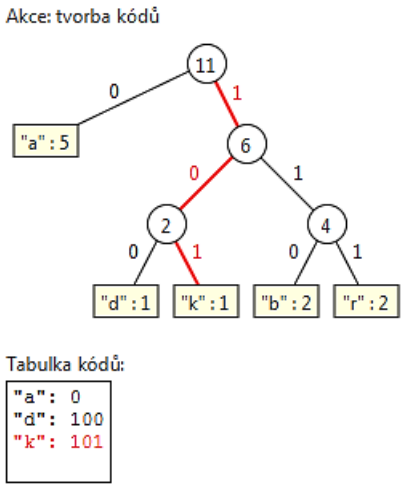
Proces tvorby kódů lze řešit dvěma odlišnými přístupy:

- rekurzí,
- pre-order prohlídkou.

V této implementaci se kódy jednotlivých znaků tvoří rekurzivním voláním funkce. Animace tvorby kódů využívá pre-order prohlídku. Pseudokód tvorby kódů pomocí rekurzivního volání funkce je naznačen níže:

```
funkce vytvořKódy(tabulkaKódů, aktuálníUzel, kód):
      pokud aktuálníUzel je list pak
            vlož do tabulkaKódů aktuálníKód.znak => kód;
      jinak
            vytvořKódy(tabulkaKódů, aktuálníUzel.levýSyn, kód + „0“);
            vytvořKódy(tabulkaKódů, aktuálníUzel.pravýSyn, kód + „1“);
konec;
```

Na obrázku 37 je zachycena tvorba kódu odpovídající znaku „k“.

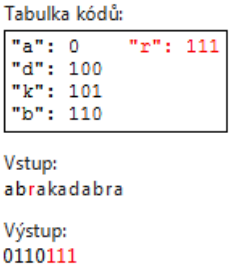


Obrázek 37 – Tvorba kódů pomocí Huffmanova stromu

Zdroj: vlastní

8.3 Kódování

Pokud jsme vybudovali kódovací strom, a z něj následně tabulku kódů, je zakódování vstupního textu snadné. Postupně procházíme vstupní text znak po znaku a každý znak nahradíme příslušným kódem z tabulky kódů. Průběh kódování vstupního textu „abrakadabra“ se nachází na následujícím obrázku. Odpovídající Huffmanův kódovací strom je zobrazen na obrázku 37 výše.



Obrázek 38 – Průběh kódování vstupního textu pomocí Huffmanova stromu

Zdroj: vlastní

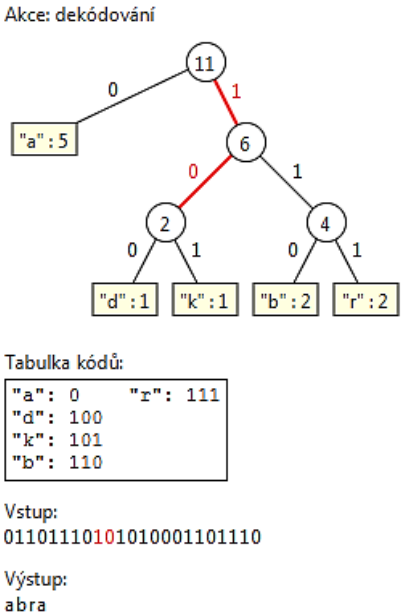
Těmito třemi operacemi je dokončeno kódování textu. Výsledný kód se spolu s kódovacím stromem uloží. Následné dekódování probíhá dle uloženého kódovacího stromu a výstupního kódu.

8.4 Dekódování

Dekódování je proces, kdy z výstupního kódu a kódovacího stromu zpět získáme původní text. Průběh dekódování je zhruba následující:

```
aktuálníUzel = kořen;
kód = posloupnost bitů vzniklá kódováním;

pro každý bit kódu dělej:
      pokud je bit „0“ pak
            aktuálníUzel = aktuálníUzel.levýSyn;
      jinak
            aktuálníUzel = aktuálníUzel.pravýSyn;

      pokud je aktuálníUzel list pak
            zapiš na výstup aktuálníUzel.znak;
            aktuálníUzel = kořen;
```

Dekódování každého znaku začíná vždy v kořenu. Pokud je na vstupu bit `0`, pak jdeme na levého potomka, jinak na pravého. Pokud jsme dotraverzovali do listu, pak znak uložený v tomto listu zapíšeme na výstup. Algoritmus končí po zpracování všech vstupních bitů. Průběh dekódování zakódovaného textu „abrakadabra“ ilustruje obrázek níže.



Obrázek 39 – Dekódování textu pomocí Huffmanova stromu

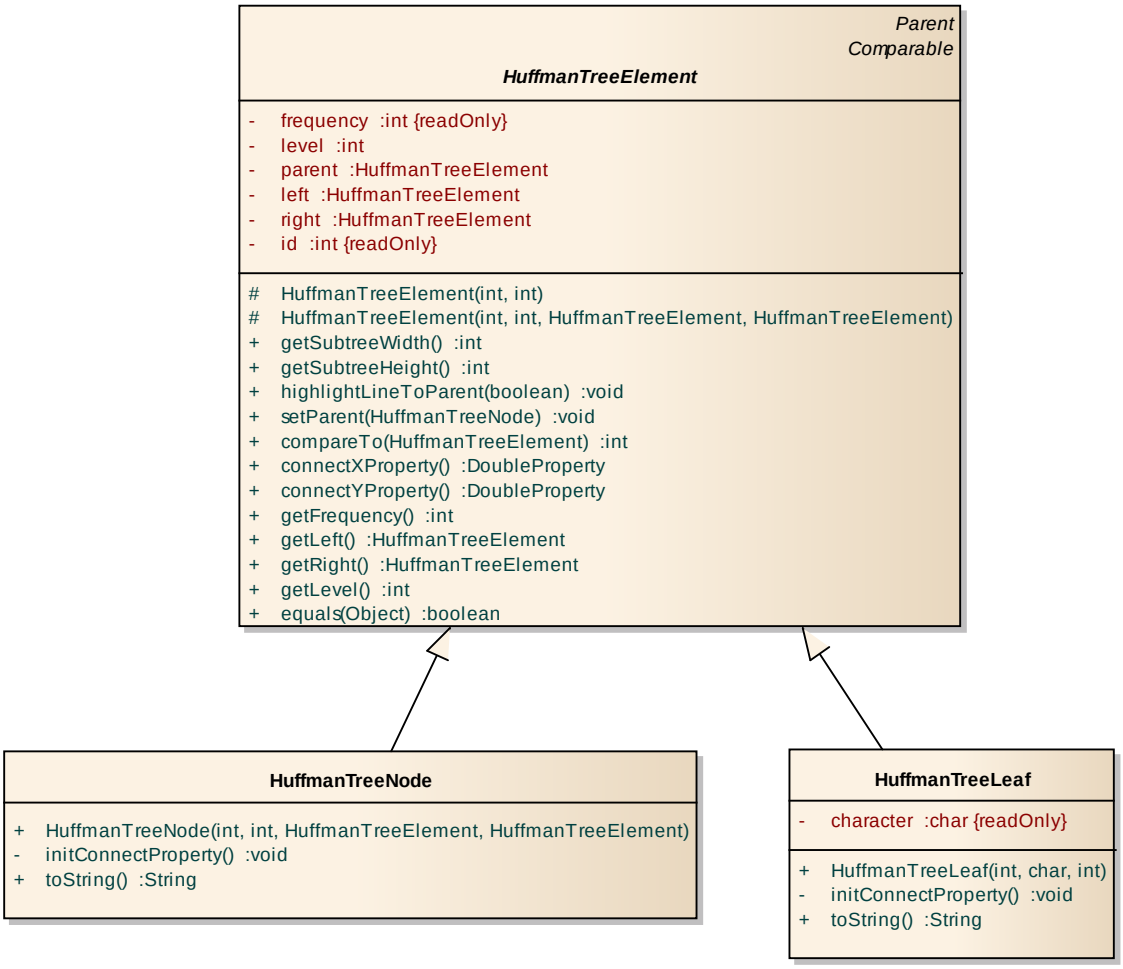
Zdroj: vlastní

Na vstupu jsou zvýrazněny bity `1` a `0`. Následuje bit `1`, tudíž z aktuálního uzlu půjdeme do pravého potomka. Dostáváme se tedy do listu, ve kterém se nachází znak „k“. Na výstupu se tedy nyní bude nacházet text „abrak“.

8.5 Grafické rozmístění prvků

Jelikož animace Huffmanova kódování je na rozdíl od ostatních struktur statická (proběhne pouze vybudování stromu, vytvoření kódů, kódování a dekódování), není zarovnávání prvků v animační scéně nikterak složité. Na začátku jsou všechny prvky vloženy přímo do animační scény. Během postupného budování stromu jsou prvky přesouvány z animační scény do scény rodičovského prvku. Tento způsob organizace prvků zajistí, že jakmile budeme pohybovat s rodičem, budou se hýbat i prvky v něm obsažené, tedy jeho potomci,

také potomci těchto potomků atd. Na obrázku 40 se nachází UML diagram tříd grafických prvků Huffmanova stromu.



Obrázek 40 – UML diagram tříd grafických prvků Huffmanova stromu

Zdroj: vlastní

Každý grafický prvek Huffmanova stromu disponuje frekvencí, hloubkou zanoření, rodičem, levým a pravým potomkem. Třída `HuffmanTreeNode` reprezentuje vnitřní uzel stromu, třída `HuffmanTreeLeaf` představuje list stromu, ve kterém se navíc nachází uchovávaný znak.

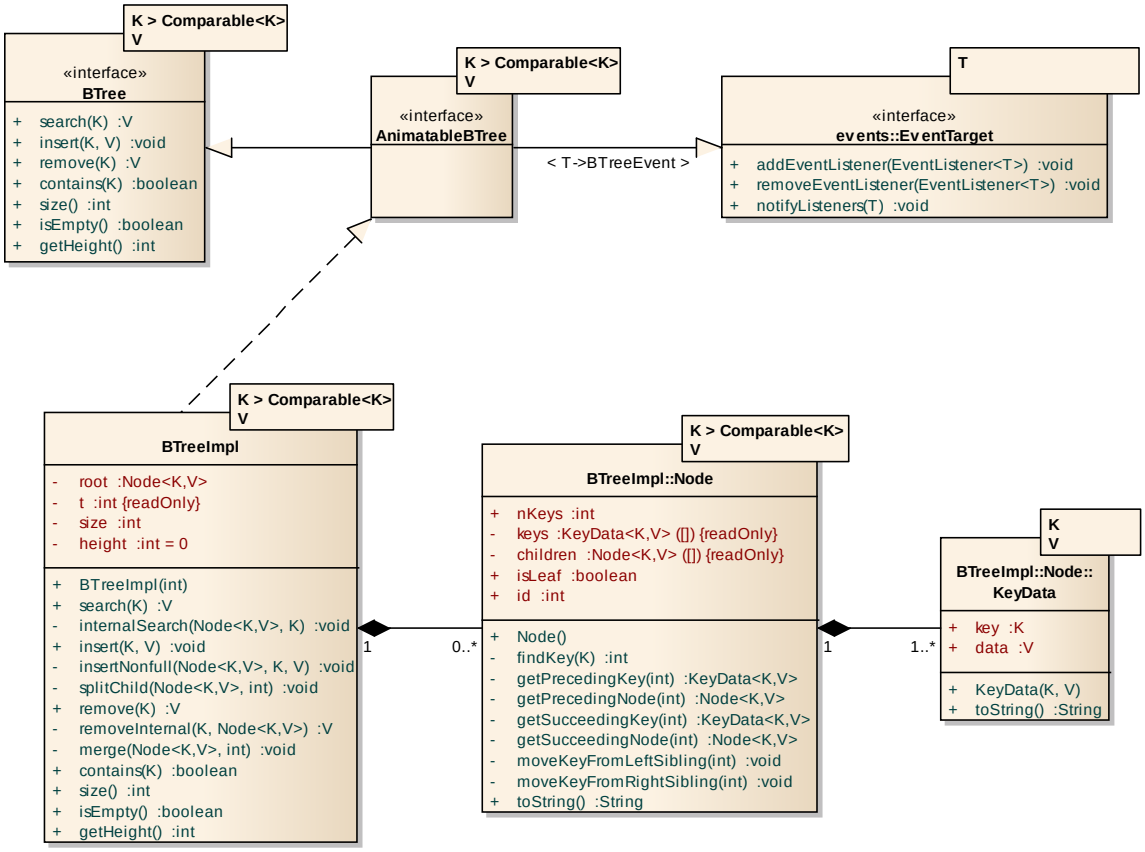
9 Implementace B-stromu

V této kapitole bude představena implementace B-stromu dle Cormena aj. (2009). Uzly budou štěpeny a fúzovány proaktivně (viz kapitola 5.2.4). Minimální a maximální povolený počet klíčů v B-stromu je dle Cormena aj. (2009) definován pomocí parametru *t*:

- $t \in \mathbf{N}, t \geq 2$;
- každý uzel (kromě kořene) obsahuje nejméně $\boldsymbol{t} - \mathbf{1}$ klíčů;
- každý uzel (kromě kořene) obsahuje nejméně $\boldsymbol{t}$ potomků;
- každý uzel obsahuje nejvíce $\mathbf{2}\boldsymbol{t} - \mathbf{1}$ klíčů;
- každý uzel obsahuje nejvíce $\mathbf{2}\boldsymbol{t}$.

Bude-li tedy parametr $t = 2$, pak může uzel B-stromu obsahovat 1–3 klíče a 2–4 potomky.

Na UML diagramu tříd níže je zachycena struktura implementovaného B-stromu. Prostřednictvím rozhraní `AnimatableBTree` implementuje třída `BTreeImpl` jak metody pro manipulaci s prvky, tak metody pro zaregistrování posluchačů generovaných událostí.



Obrázek 41 – UML diagram tříd B-stromu

Zdroj: vlastní

Součástí B-stromu je kořen `root`, parametr `t`, počet uložených prvků `size` a výška stromu `height`.

Stěžejní částí je vnitřní třída `Node` reprezentující uzel B-stromu. Jsou zde uchovávány jednak klíče s přidruženými daty, jednak reference na potomky. V poli `keys` o dimenzi $\mathbf{2t-1}$ jsou uloženy klíče a přidružená data (instance třídy `KeyData`). V poli `children` o dimenzi $\mathbf{2t}$ se nacházejí reference na potomky. Dále každý uzel disponuje aktuálním počtem uložených klíčů (`nKeys`) a značkou, zda se jedná o list (`isLeaf`). Pro potřeby vizualizace byla zavedena proměnná `id` k jednoznačné identifikaci každého uzlu.

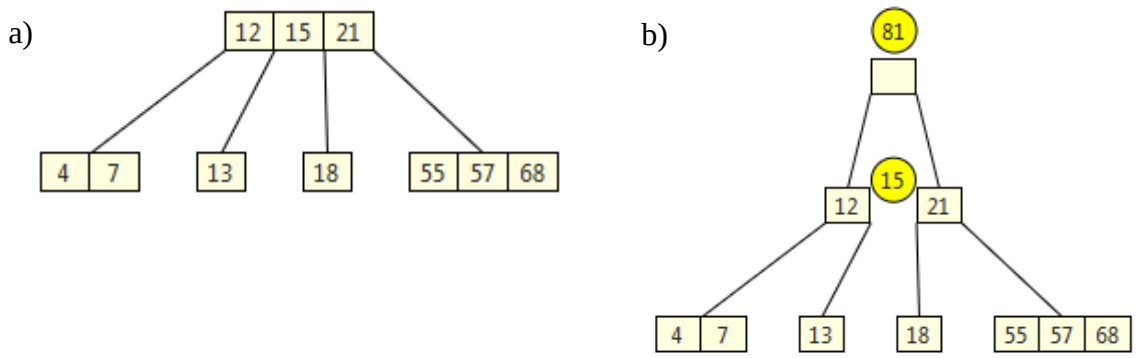
V následujících kapitolách bude popsáno fungování operací *VLOŽ*, *NAJDI* a *ODEBER*. Časové složitosti těchto operací jsou $O(th) = O(t \log_t n)$. (Cormen aj. 2009). Pokud nebude uvedeno jinak, budou ukázky prováděny na B-stromu s parametrem $t = 2$ s daty z přednastavené sady číslo 1.

9.1 Operace Vlož

Operace *VLOŽ* zajistí vložení prvků s klíčem *K* do adekvátního listu B-stromu. Adekvátním listem je myšlen takový list, do kterého se traverzuje na základě porovnávání klíčů uložených ve vnitřních uzlech se vkládaným klíčem. Podrobnější popis traverzu do adekvátního listu dle klíče bude popsán v rámci popisu operace *NAJDI* (kapitola 9.2). Představovaný proces vkládání prvku do B-stromu je proaktivní, tedy prvek je do B-stromu vložen během jednoho průchodu od kořene do adekvátního listu. Proces vložení prvku je rozdělen do dvou metod:

- *INSERTNONFULL* – vložení klíče do neplného uzlu a
- *SPLITCHILD* – rozdělení potomka do dvou uzlů.

Na počátku operace *VLOŽ* je nejprve nutné ošetřit případné přeplnění kořene. K němu může dojít v případě, že kořen obsahuje maximální možný počet klíčů, tedy přesně $2t-1$. Nastala-li tato situace, rozdělíme původní kořen do dvou uzlů a separující klíč (prostřední) vložíme do nového uzlu. Nový uzel se stane novým kořenem B-stromu. Vkládání pokračuje voláním metody *INSERTNONFULL*. Štěpení kořene je ilustrováno na obrázku 42. Pokud do stromu na obrázku 42a budeme vkládat jakýkoliv prvek, bude kořen rozštěpen tak, jak je naznačeno na obrázku 42b. Při štěpení uzlu je vždy vyjmut prostřední klíč a přesunut na odpovídající místo do rodiče.



Obrázek 42 – B-strom – štěpení kořene

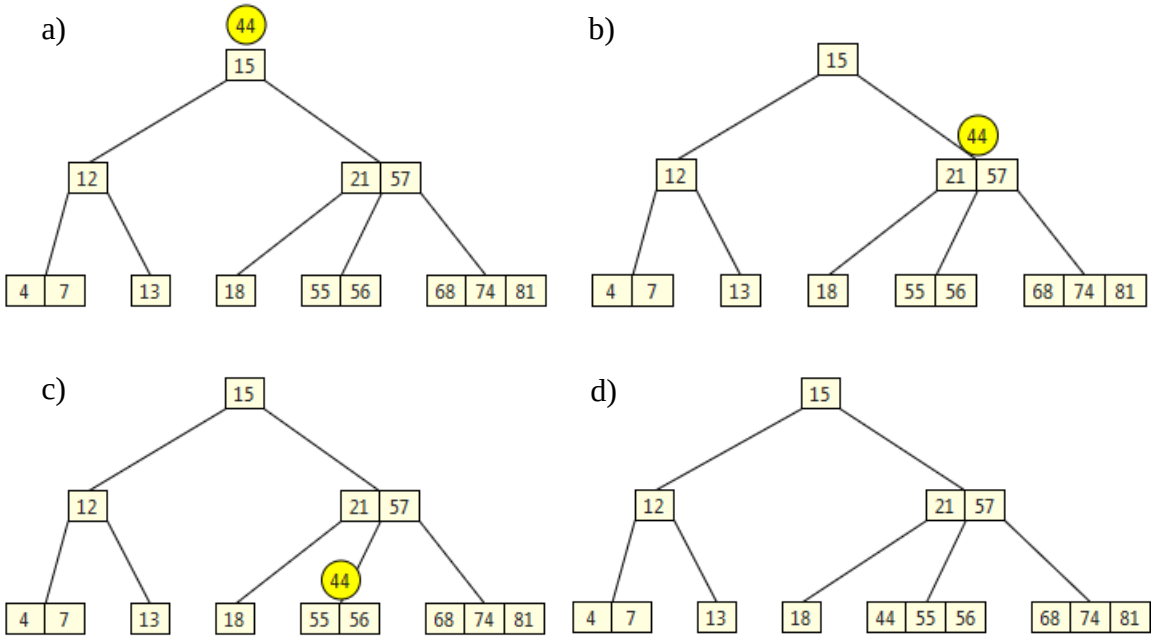
Zdroj: vlastní

Pokud kořen nemůže potenciálně přetéci, pokračuje vkládání ihned voláním metody *INSERTNONFULL*.

V případě že jsme dotraverzovali do listu, zajistí metoda *INSERTNONFULL* vložení klíče na správnou pozici do tohoto listu. V opačném případě porovnává uložené klíče se vkládaným klíčem a pokračuje rekurzí do odpovídajícího potomka. Pseudokód metody *INSERTNON-FULL* vypadá následovně:

```
funkce insertNonFull(uzel, klíč, data)
      pokud uzel je list pak
            přesuň všechny klíče větší než klíč o jedno místo doprava;
            vlož klíč s daty na uvolněnou pozici;

      jinak
            potomek = najdi potomka, do kterého náleží klíč;
            pokud potomek.početKlíčů = 2*t-1 pak
                  rozděl potomka; //volání splitChild

            insertNonFull(potomek, klíč, data);
konec;
```

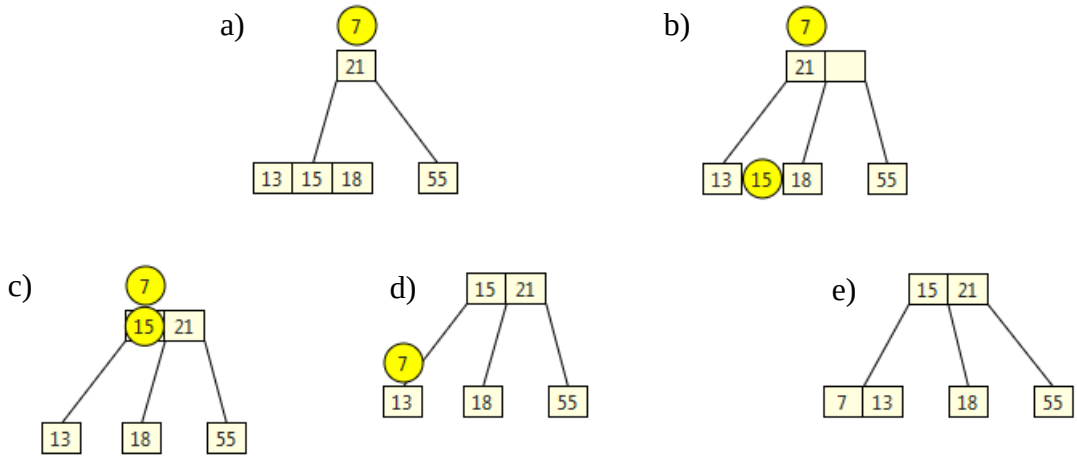
Budeme-li do B-stromu na obrázku 43a vkládat prvek s klíčem 44, bude postup následující. Jelikož kořen není plný (nehrozí jeho přetečení), není jej třeba štěpit. Porovnáme klíč vkládaného prvku s klíčem kořene. Jelikož 44 > 15, pokračujeme do pravého podstromu (obrázek 43b). Následuje porovnání klíče vkládaného prvku s klíči 21 a 57: 44 < 57 a 44 > 21, tudíž bude traverz pokračovat do „prostředního“ potomka (obrázek 43c). Nyní se již nacházíme v listu, do kterého prvek s klíčem 44 patří. V listu přesuneme všechny klíče větší než vkládaný klíč o jedno místo doprava a klíč vložíme na uvolněnou pozici (obrázek 43d).



Obrázek 43 – B-strom – vkládání prvku s klíčem 44

Zdroj: vlastní

Zbývá představit fungování metody *SPLITCHILD*. Tato metoda zajistí rozdělení uzlu do dvou nových uzlů a přesunutí prostředního klíče z děleného uzlu do rodičovského uzlu. Proces štěpení uzlu je znázorněn na obrázku 44.



Obrázek 44 – B-strom – štěpení uzlu

Zdroj: vlastní

Vkládaný prvek s klíčem 7 patří do levého podstromu (7 < 21) (obrázek 44a). Uzel v levém podstromu je však plný. Vložením prvku do tohoto listu by došlo k narušení maximálního povoleného počtu klíčů uchovávaných v uzlu. Proto je uzel rozdělen na jeden uzel s klíčem 13 a druhý uzel s klíčem 18 (obrázek 44b). Prostřední klíč 15 z původního uzlu je přesunut do rodiče (obrázek 44c). Vkládání pokračuje traverzem do levého přilehlého podstromu klíče 15 (7 < 15) (obrázek 44d). Klíče v listu jsou posunuty doprava a prvek s klíčem 7 je vložen na uvolněné místo (obrázek 44e). Pokud bychom zapsali průběh štěpení uzlu v pseudokódu, vypadal by následovně:

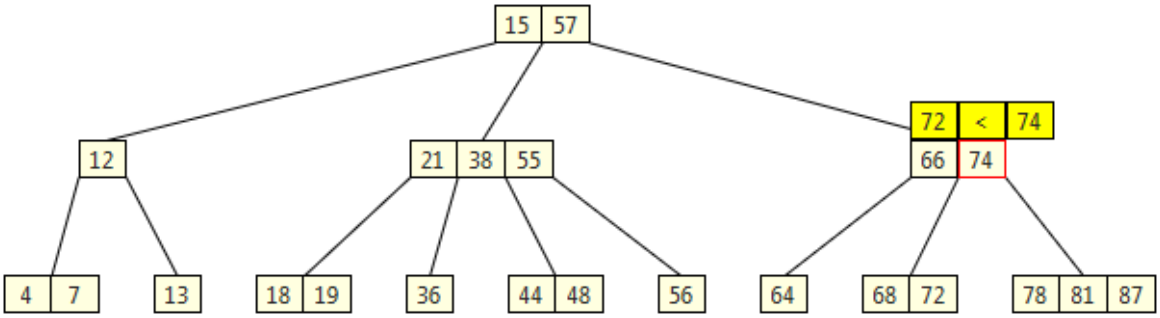
```
rodič = rodič děleného uzlu;
index = index děleného uzlu (potomka) v rodiči;

novýUzel = vytvoř nový uzel;
novýUzel.jeList = dělenýUzel.jeList;
novýUzel.početKlíčů = t-1; //rovno minimálnímu počtu klíčů

přesuň posledních t-1 klíčů z dělenýUzel do novýUzel;

pokud dělenýUzel není list pak
      přesuň posledních t potomků z dělenýUzel do novýUzel;

dělenýUzel.početKlíčů = t-1;

přesuň potomky od indexu index+1 v rodič o jednu pozici doprava;
vlož novýUzel do rodič na pozici index+1;

přesuň klíče od indexu index v rodič o jednu pozici doprava;
vlož prostřední klíč z dělenýUzel do rodič na index index;

rodič.početKlíčů = rodič.početKlíčů + 1;
```

9.2 Operace Najdi

Princip vyhledávání prvků v B-stromu je založen na stejném principu jako u binárního vyhledávacího stromu. Na rozdíl od binárního vyhledávacího stromu, kde je možné pokračovat buď do levého, nebo do pravého potomka, je možných cest v B-stromu více (v závislosti na počtu uložených klíčů v konkrétním uzlu). Obsahuje-li uzel *n* klíčů, pak můžeme pokračovat právě do jednoho z jeho *n+1* podstromů.

Vyhledávání prvku začíná vždy v kořenu. V každém navštíveném uzlu postupně porovnáváme uložené klíče s hledaným klíčem. Pokud je hledaný klíč menší než první uložený klíč, pokračujeme do nejlevějšího podstromu. Jinak hledáme první větší klíč. Jakmile takový klíč najdeme, traverz pokračuje do jeho levého přilehlého podstromu. Pokud je hledaný klíč větší než všechny uložené klíče, pokračujeme do nejpravějšího podstromu. Pokud jsme tímto způsobem dotraverzovali až do listu, následuje sekvenční průchod uložených klíčů a postupné porovnávání s hledaným klíčem. Shoduje-li se hledaný klíč s některým uloženým klíčem v listu, bylo hledání úspěšné. V opačném případě se hledaný klíč v B-stromu nenachází.

Na obrázku 45 je zachyceno porovnávání hledaného klíče s uloženými klíči v rámci hledání prvku s klíčem 72. Jelikož je hledaný klíč 72 menší než klíč 74, bude traverz pokračovat do levého přilehlého podstromu klíče 74. Tento podstrom je již listem, ve kterém bude následně hledaný klíč 72 úspěšně nalezen.



Obrázek 45 – B-strom – hledání prvku s klíčem 72

Zdroj: vlastní

Níže je uveden pseudokód operace *NAJDI*. Vyhledávání prvku vždy začíná voláním `najdi(kořen, klíč)`.

```
funkce najdi(aktuálníUzel, hledanýKlíč):
      index = 0;
      dokud hledanýKlíč > aktuálníUzel.klíče[index] dělej:
            index = index + 1;

      pokud hledanýKlíč = aktuálníUzel.klíče[index] pak
            vrať aktuálníUzel.klíče[index];
      jinak pokud aktuálníUzel.jeList pak
            klíč nenalezen;
      jinak
            najdi(aktuálníUzel.syn[index], hledanýKlíč);
konec;
```

9.3 Operace Odeber

Operace *ODEBER* je poněkud komplikovanější než operace *VLOŽ*. Při vkládání je prvek vždy vložen do listu, kdežto při odebírání může být prvek odebrán jak z listu, tak i z vnitřního uzlu. Při vkládání se musely plné uzly při traverzu z kořene do listu štěpit, aby nemohl některý uzel přetéci. Při odebírání se musí zajistit, aby žádný uzel při traverzu z kořene k odebíranému prvku nemohl podtéci. Tato podmínka bude splněna, pokud budou uzly obsahovat alespoň t klíčů. Pokud bude z uzlu, kde se nachází právě t klíčů, odebrán další klíč, bude ještě obsahovat $t - 1$ klíčů, což stále vyhovuje kladeným podmínkám o minimálním počtu klíčů v uzlech.

Během odebírání prvku z B-stromu může nastat několik různých situací. Jedná se o tyto:

1) klíč odebíraného prvku se nachází v aktuálním uzlu;
   a) aktuální uzel je list – klíč je odebrán z listu;
   b) aktuální uzel je vnitřní uzel;
      i) pokus se odebíraný klíč nahradit nejpravějším (největším) klíčem z levého přilehlého podstromu (dále NEJPRAV);
      ii) jinak se pokus odebíraný klíč nahradit nejlevějším (nejmenším) klíčem z pravého přilehlého podstromu (dále NEJLEV);
      iii) jinak spoj přilehlého levého a pravého potomka odebíraného klíče a do spojeného uzlu přesuň z rodiče odebíraný klíč (dále SPOJ);

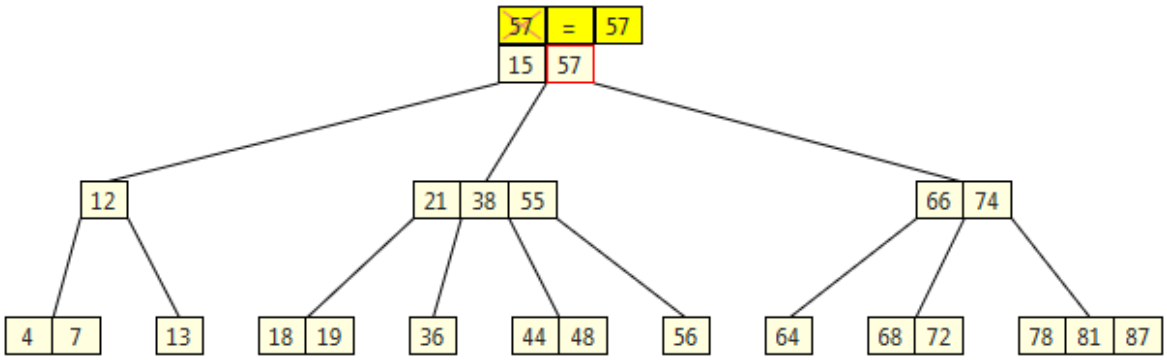
2) klíč odbíraného prvku se nenachází v aktuálním uzlu;
   a) aktuální uzel je list – odebíraný klíč nenalezen;
   b) pokud potomek, ve kterém by se mohl vyskytovat odebíraný klíč, obsahuje méně než t klíčů;
      i) pokus se přesunout klíče do potomka z jeho levého souseda (LEVÝSOUSED);
      ii) jinak se pokus přesunout klíče do potomka z jeho pravého souseda (PRAVÝSOUSED);
      iii) jinak spoj potomka s jeho sousedem (SPOJSOUSED).

Jak je vidět, oproti vkládání, kde se může objevit pouze rozdělení uzlu a přesun klíče do rodiče, je odebírání poněkud složitější proces.

Předpokládejme, že výše naznačená struktura možných situací je tělem metody, která obstarává odebírání prvků z B-stromu. Během řešení výše naznačených situací může být odebírání prvků rekurzivně voláno. Stejně jako u vkládání, začíná odebírání prvku od kořene, tedy voláním `odeber(kořen, klíč)`. Následovat bude ilustrace a řešení některých situací. Nutno podotknout, že se jedná o tzv. proaktivní řešení, tedy uzly na cestě k odebíranému prvku jsou ošetřeny během jednoho průchodu od kořene do příslušného listu.

Situace NEJPRAV

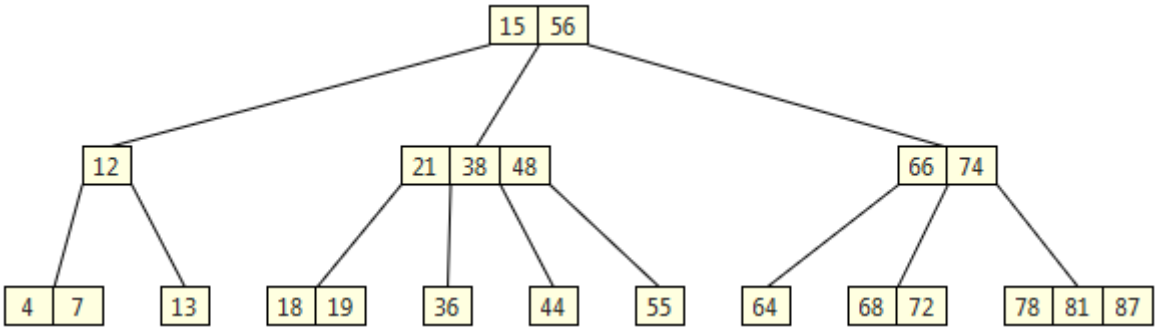
Chtějme odebrat prvek s klíčem 57 z B-stromu na obrázku 46. Fakt, že se odebíraný klíč nachází přímo v kořenu, nikterak neovlivňuje vzniklou situaci. Jelikož potomek v levém přilehlém podstromu odebíraného klíče disponuje alespoň *t* klíči, můžeme odebíraný klíč nahradit klíčem z tohoto podstromu.



Obrázek 46 – B-strom – před odebráním prvku s klíčem 57

Zdroj: vlastní

Nejpravější (největší) klíč z levého přilehlého podstromu odebíraného klíče (konkrétně klíč 56) je zkopírován na pozici odebíraného klíče (57). Odebraný prvek s klíčem 57 je vrácen. Odebírání však ještě nemůžeme ukončit – musíme z podstromu odebrat duplikovaný klíč 56. To učiníme opětovným voláním operace *ODEBER*, nyní však s hodnotou klíče 56 (samozřejmě mohou být během této rekurze řešeny i další kritické situace). Uspořádání B-stromu po dokončení odebrání prvku s klíčem 57 se nachází na obrázku níže.



Obrázek 47 – B-strom – po odebrání prvku s klíčem 57

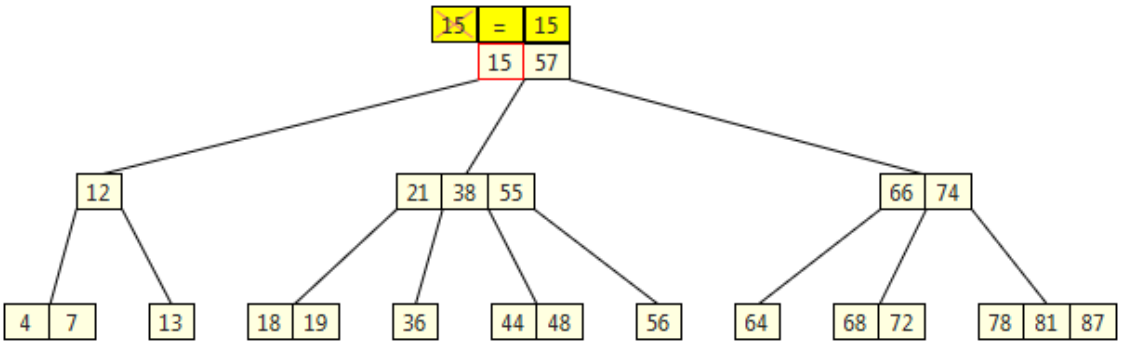
Zdroj: vlastní

Situace NEJLEV

Tato situace je podobná předchozí situaci. Pouze se změní to, že nelze vzít klíč z levého přilehlého podstromu, tudíž se vezme klíč z pravého přilehlého podstromu. Tentokrát se však bude jednat o nejlevější (nejmenší) klíč.

Budeme-li z B-stromu zobrazeného níže odebírat prvek s klíčem 15, nemůžeme tento prvek nahradit prvkem z levého přilehlého podstromu. Levý přilehlý potomek neobsahuje

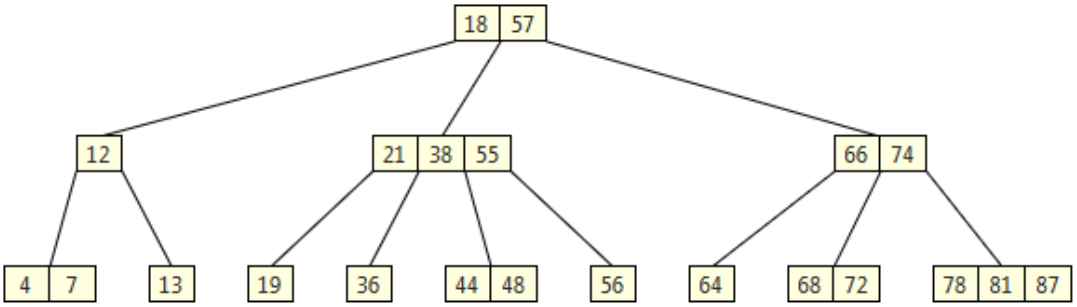
alespoň *t* klíčů (tedy minimálně 2 klíče), obsahuje pouze jeden, a to klíč 12. Proto musíme odebíraný klíč nahradit klíčem z pravého přilehlého podstromu, tedy klíčem 18.



Obrázek 48 – B-strom – před odebráním prvku s klíčem 15

Zdroj: vlastní

Analogicky jako u předchozí situace je provedeno rekurzivní odstranění duplikovaného klíče. Na rozdíl od předchozího příkladu nedochází k dalším reorganizacím, protože v listu, kde se nachází klíč 18, je stále alespoň *t* klíčů. B-strom po odebrání prvku s klíčem 15 zobrazuje obrázek 49.

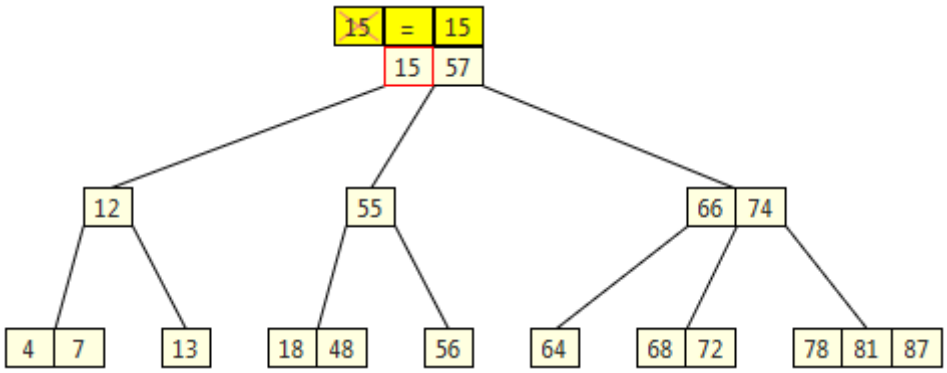


Obrázek 49 – B-strom – po odebrání prvku s klíčem 15

Zdroj: vlastní

Situace SPOJ

Tato situace nastane za předpokladu, že se nepodařilo přesunout klíč ani z levého, ani z pravého přilehlého podstromu odebíraného klíče (kořeny těchto podstromů nedisponovali dostatečným počtem klíčů). Řešením je spojení kořenů těchto podstromů do jednoho uzlu. Situace je ilustrována na obrázku níže.

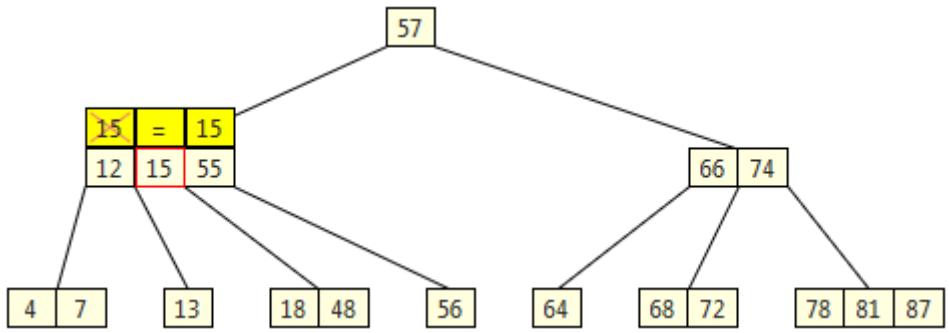


Obrázek 50 – B-strom – před odebráním prvku s klíčem 15 (sloučení uzlů)

Zdroj: vlastní

Postup řešení vzniklé situace je následující. Z uzlu, ve kterém se nachází odebíraný klíč, přesuneme separující klíč (odděluje dva slučované podstromy) do kořene levého podstromu. Přesunovaný separující klíč je samozřejmě totožný s odebíraným klíčem. Následně přesuneme všechny klíče z kořene pravého podstromu za separující klíč do kořene levého podstromu. Separující klíč přesunutý z rodičovského uzlu se nyní nachází uprostřed mezi původními klíči a klíči z kořene pravého podstromu. V obou kořenech slučovaných podstromů se totiž před sloučením nacházel stejný počet klíčů, a to přesně $t - 1$ klíčů (jinak by tato situace vůbec nenastala). Zbývá správně upravit reference na potomky v rodičovském uzlu a přesunout potomky z kořene pravého podstromu do kořene levého podstromu. Tímto postupem zmizel původní pravý přilehlý podstrom odebíraného prvku – byl sloučen. Algoritmus však ještě nekončí – odebíraný prvek byl pouze přesunut do potomka – musíme dále pokračovat rekurzivním odstraněním odebíraného prvku.

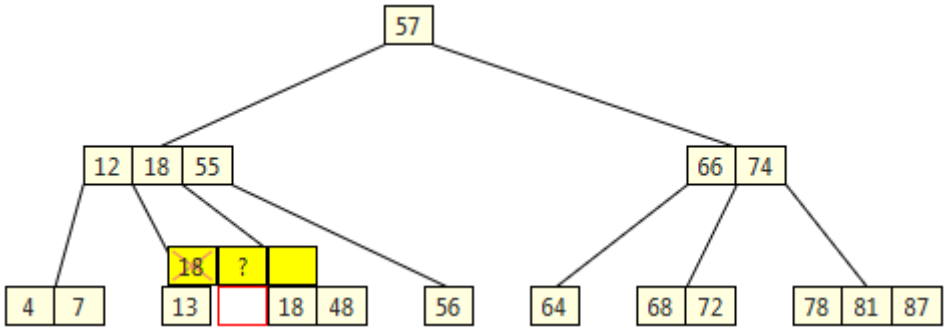
Průběh odebírání prvku s klíčem 15 z výše uvedeného B-stromu zachycuje obrázek 51. Sloučeny byly uzly s klíči 12 a 55. Do sloučeného uzlu byl také z rodiče přesunut separující (odebíraný) klíč 15.



Obrázek 51 – B-strom – odebrání prvku s klíčem 15 – sloučení uzlů

Zdroj: vlastní

Po sloučení následuje rekurzivní odebrání odebíraného klíče 15. Během této rekurze opět dojde k dalším reorganizacím (NEJLEV), jak je naznačeno na obrázku níže.



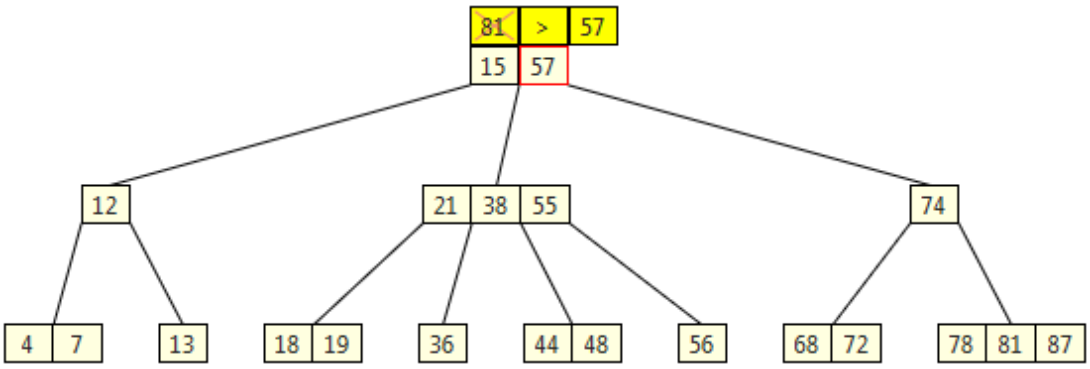
Obrázek 52 – B-strom – po odebrání prvku s klíčem 15 (sloučení uzlů)

Zdroj: vlastní

Situace LEVÝSOUSED

V tomto případě se prvek s odstraňovaným klíčem nenachází ve zpracovávaném uzlu. Zpracovávaný uzel se však nachází na cestě k odebíranému prvku a obsahuje pouze $t - 1$ klíčů – mohl by potenciálně podtéci. Levý soused zpracovávaného uzlu obsahuje alespoň t klíčů, tudíž si z něj můžeme klíč „vypůjčit“. Konkrétní reorganizace probíhá následovně. Nejprve je nutné identifikovat klíč v rodiči separující zpracovávaný uzel a jeho levého souseda. Separující klíč je následně přesunut na nejlevější pozici do zpracovávaného uzlu. Místo uvolněné separujícím klíčem je zaplněno nejpravějším klíčem z levého souseda zpracovávaného uzlu. Nejpravější podstrom v levém sousedu zpracovávaného uzlu je nastaven jako nejlevější podstrom zpracovávaného uzlu.

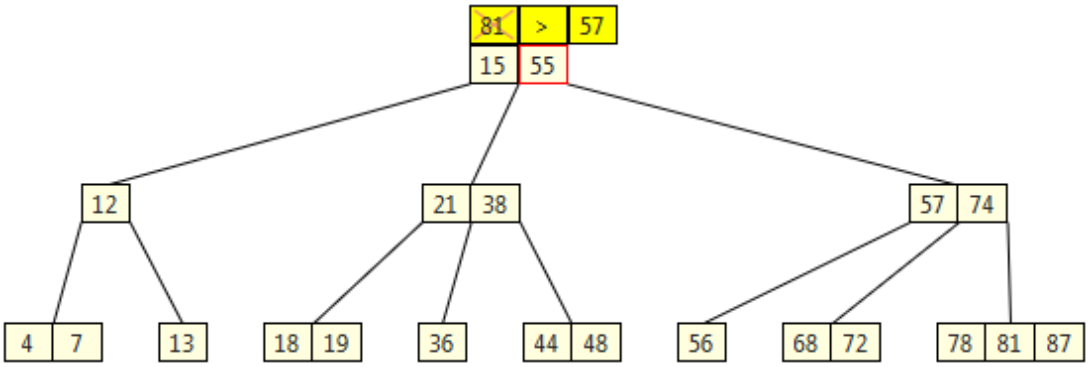
Z konkrétního příkladu bude postup reorganizace jasnější. Z B-stromu níže odebíráme prvek s klíčem 81.



Obrázek 53 – B-strom – před přesunutím klíče z levého souseda

Zdroj: vlastní

Traverzujeme přes uzel, ve kterém se nachází pouze jeden klíč – s hodnotou 74, tudíž obsahuje pouze $t - 1$ klíčů. Z rodiče přesuneme klíč 57 do zpracovávaného uzlu. Z levého souseda přesuneme nejpravější klíč – 55 – na uvolněné místo do rodiče (po klíči 57). Podstrom s kořenem, jehož klíč má hodnotu 56, nastavíme jako levého přilehlého potomka nejlevějšího klíče (57) zpracovávaného uzlu (obrázek 54).



Obrázek 54 – B-strom – po přesunutí klíče z levého souseda

Zdroj: vlastní

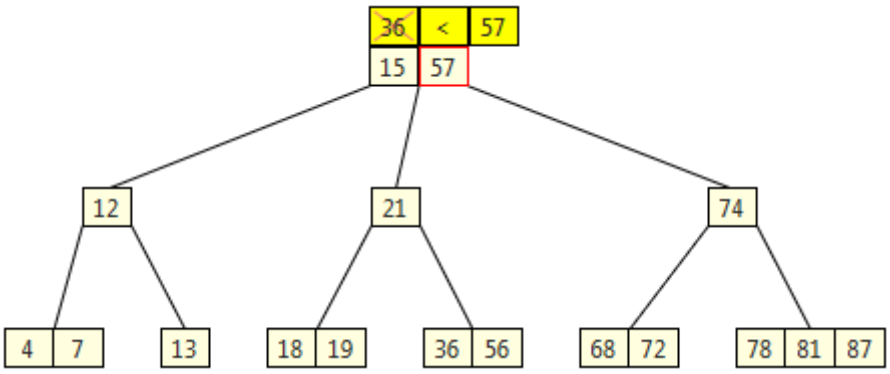
Situace PRAVÝSOUSED je analogická tomuto případu s tím rozdílem, že klíč je „vypůjčen“ z pravého souseda zpracovávaného uzlu.

Situace SPOJSOUSED

Posledním kritickým manévrem v rámci odebírání prvku je situace, kdy se na cestě k odebíranému prvku opět vyskytne uzel, ve kterém se nachází pouze $t - 1$ klíčů. Specifikem této situace je ten fakt, že si nemůžeme „vypůjčit“ klíč ani z levého, ani z pravého souseda, poněvadž obsahují pouze $t - 1$ klíčů.

Situaci vyřešíme spojením zpracovávaného uzlu s jeho pravým, případně s levým sousedem. Spojení s levým sousedem nastane, pouze pokud je zpracovávaný uzel kořenem nejpravějšího podstromu (poslední podstrom) ve svém rodiči. V následujícím popisu bude bráno v úvahu, že se nejedná o poslední podstrom. Nejprve musíme v rodiči identifikovat klíč separující zpracovávaný uzel a jeho pravého souseda. Separující klíč přesuneme na poslední pozici do zpracovávaného uzlu. Za separující klíč také přesuneme veškeré klíče (spolu s jejich potomky) z pravého souseda zpracovávaného uzlu.

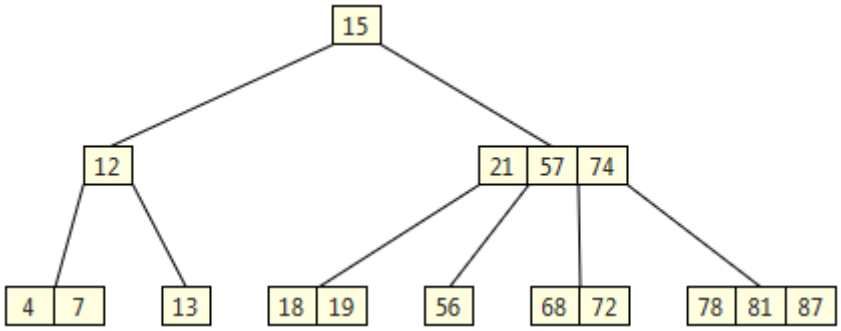
Výše popsaná situace se vyskytuje na obrázku 55. Odebíraný prvek s klíčem 36 se nachází v podstromu, jehož kořen obsahuje pouze jeden klíč – 21. Oba sousedi tohoto kořene obsahují také pouze jeden klíč (12 a 74), proto musíme dva kořeny sloučit. Jelikož kořen s hodnotou klíče 21 není kořenem posledního podstromu ve svém rodiči (tj. v tomto případě kořen s klíčem 74), sloučíme zpracovávaný kořen (21) s jeho pravým sousedem (74).



Obrázek 55 – B-strom – před sloučením uzlů

Zdroj: vlastní

V rodiči zpracovávaného uzlu identifikujeme klíč separující tyto dva kořeny – 57 – a přesuneme jej do zpracovávaného uzlu. Následně přesuneme všechny klíče (74) a potomky z pravého souseda do zpracovávaného uzlu. Stav B-stromu po sloučení uzlů a po odebrání prvku s klíčem 36 se nachází na obrázku 56.



Obrázek 56 – B-strom – po sloučení uzlů

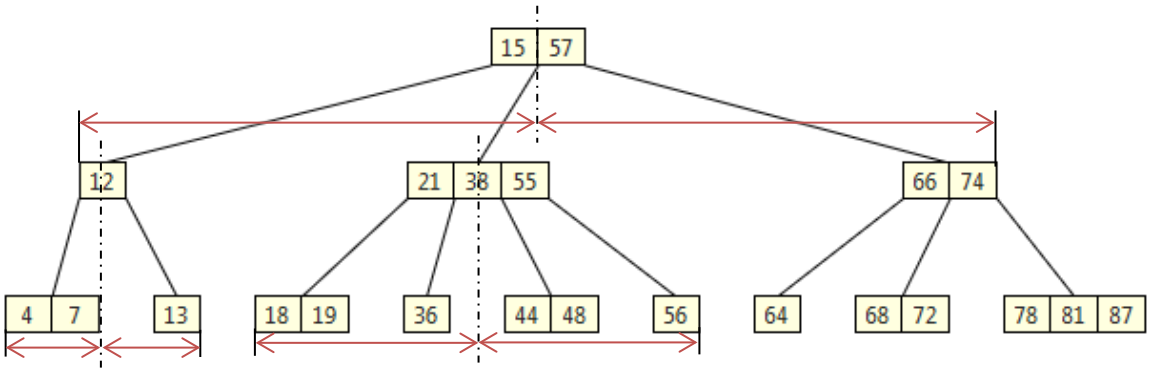
Zdroj: vlastní

Výše byly představeny různé kritické manévry při odebírání prvku z B-stromu. Stejně jako u vkládání, se musí i u odebírání specifickým způsobem ošetřit kořen B-stromu. Ošetření kořene při odebírání prvků nastává až po provedení výše naznačených reorganizací. Mohou nastat dvě kritické situace. Buď již byly odebrány všechny klíče (strom je prázdný), nebo byl v průběhu reorganizací z kořene přesunut poslední klíč do jiného uzlu, tudíž kořen neobsahuje žádné klíče. Odebrání prvku včetně ošetření kořene je zachyceno v následujícím pseudokódu.

```
odeber(kořen, klíč); // odebrání klíče z B-stromu

// ošetření kořene
pokud kořen.početKlíčů = 0 pak
      pokud kořen.jeList pak
            kořen = prázdný uzel;
      jinak
            výškaStromu = výškaStromu - 1;
            kořen = kořen.prvníSyn;
```

9.4 Grafické rozmístění prvků

Grafické zarovnání prvků je založeno na následujícím principu. Nejprve se zarovnají všechny listy, aby mezi sebou měly stejné mezery. Následně se postupně procházejí jednotlivé listy s cílem identifikovat první a poslední list stejného rodiče. Dle pozice prvního a posledního listu je zarovnán rodič (vycentrován).

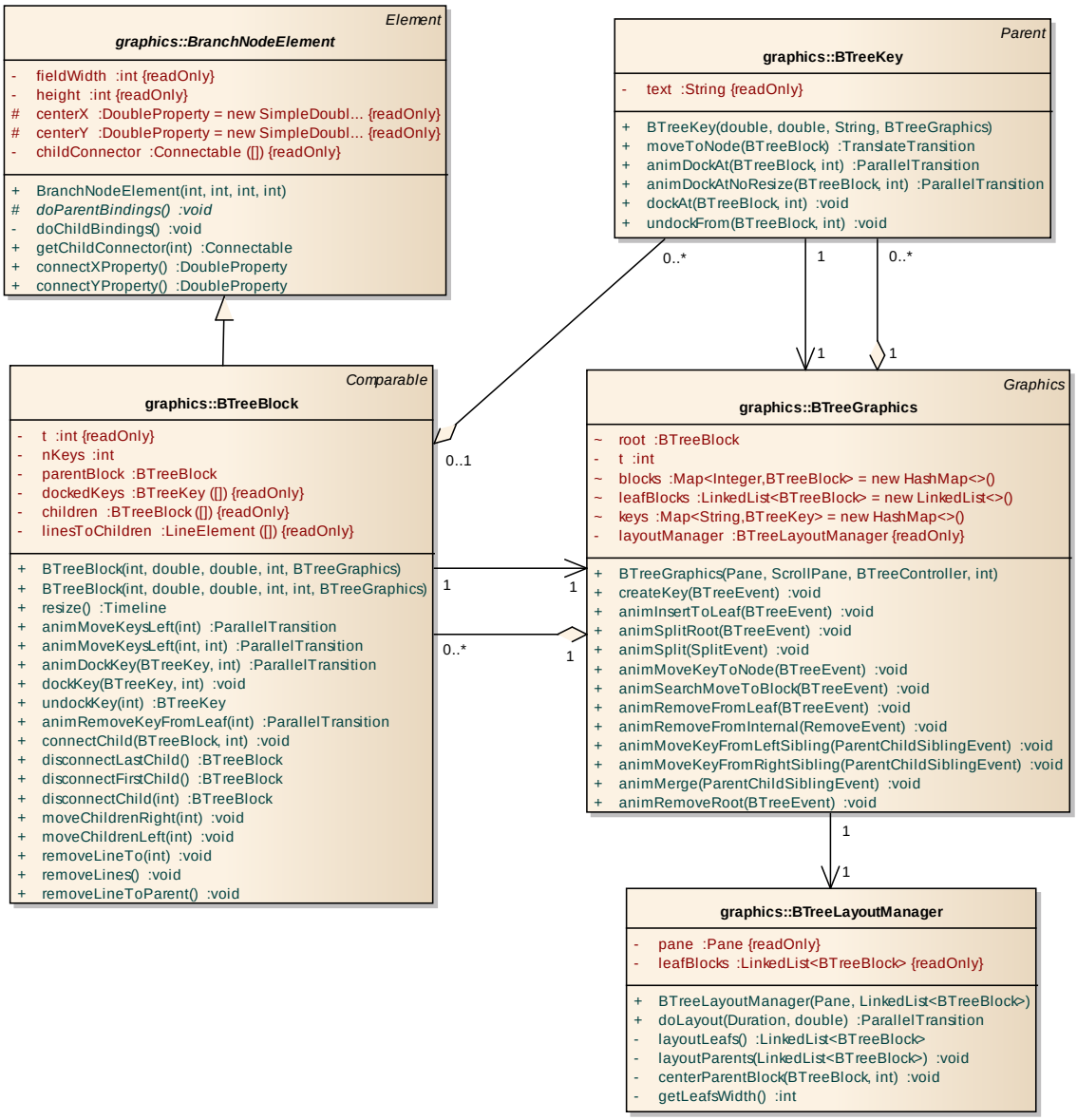


Obrázek 57 – Princip zarovnávání uzlů B-stromu

Zdroj: vlastní

Jakmile dojde k zarovnání celé úrovně, pokračuje se na další. Jako vstupní uzly (v prvním průchodu to jsou listy) jsou nyní brány zarovnaní rodiče. Takto se pokračuje, dokud není zarovnán kořen. Filozofie zarovnávání uzlů je naznačena na obrázku 57.

Na následujícím zjednodušeném UML diagramu tříd se nacházejí klíčové třídy, které se podílejí na grafickém vzhledu B-stromu. Jádrem grafické části je třída `BTreeGraphics`, které adekvátně reaguje na události generované B-stromem. Dále zajišťuje pomocí instance třídy `BTreeLayoutManager` grafické zarovnání zobrazených prvků. Blok (uzel) B-stromu `BTreeBlock` může obsahovat více uložených klíčů, tudíž je potomkem třídy `BranchNodeElement`. V bloku jsou uchovávány uložené klíče `BTreeKey`. Třídy `BTreeBlock` a `BTreeKey` disponují metodami pro provádění různých reorganizací – přesun klíčů, fúze a štěpení uzlů apod.



Obrázek 58 – UML diagram tříd grafické části B-stromu

Zdroj: vlastní

10 Implementace grid souboru

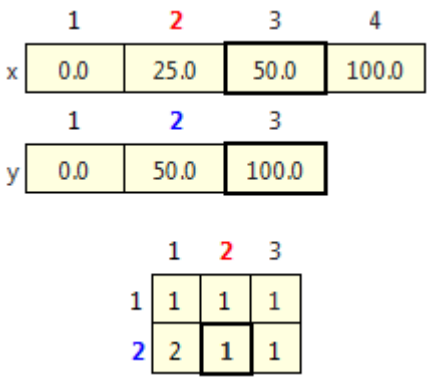
V této kapitole bude představena implementace grid souboru. Grid soubor je naimplementován pro vkládání prvků s dvoudimenzionálními klíči – vkládanými prvky mohou být například body nacházející se v rovině. Zvolená implementace respektuje následující pravidla:

- pravidelné střídání dělených dimenzí,
- dělící bod určen jako střed děleného intervalu,
- slučování datových bloků „buddy systémem“,
- po odebrání prvku požadováno alespoň 70% zaplnění datového bloku.

Dále je nutné specifikovat, jaké klíče budou patřit do které grid buňky (tedy i do příslušného datového bloku). Mějme lineární stupnici S s n hodnotami k_1, k_2, …, k_n, grid adresář A a dimenzi D. Jako A_D označme dimenzi D adresáře A. Pokud dimenzi D přísluší stupnice S, pak i-tá grid buňka ($i = 1 \dots n$) může v A_D obsahovat klíče K_D takové, že:

$$K_D \in \langle k_i;\ k_{i+1}).$$

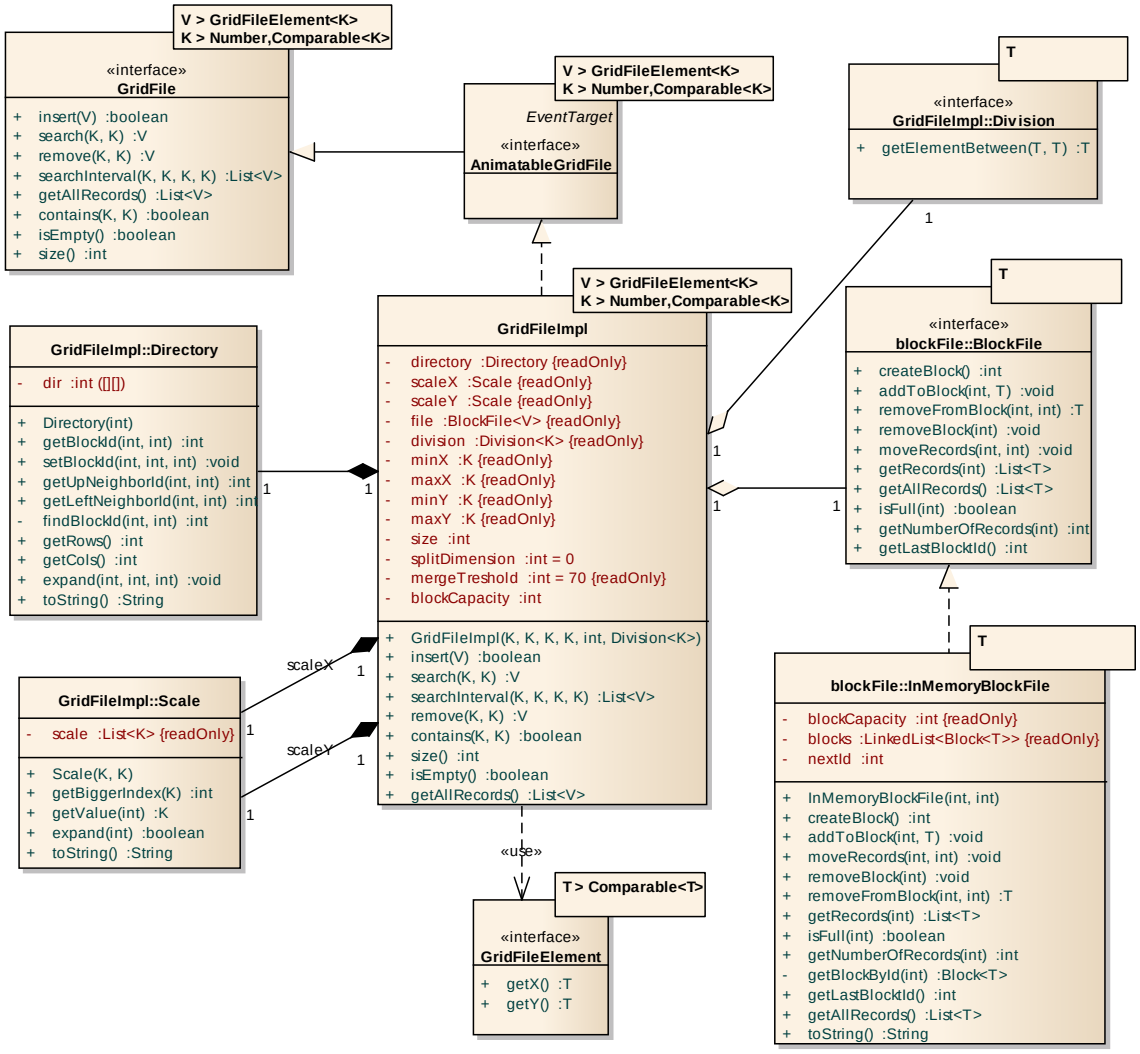
Z výše uvedené definice je patrné, že např. prvek s klíčem $x = 25$ a $y = 99$ odpovídá grid buňce na pozici [2, 2], viz níže.



Obrázek 59 – Příklad příslušnosti prvku do grid buňky

Zdroj: vlastní

Jelikož cílem implementace je především vizualizace organizace grid souboru, nebyla při implementaci vyžadována fyzická práce se souborem. Implementovaný grid soubor je však připraven i pro fyzickou práci se souborem – stačí pouze zaměnit implementaci rozhraní `BlockFile`.



Obrázek 60 – UML diagram tříd grid souboru

Zdroj: vlastní

Současná implementace využívá k ukládání dat instanci třídy `InMemoryBlockFile`, což není nic jiného, než implementace blokově orientovaného souboru v operační paměti. Při konstrukci grid souboru je požadováno předání instance typu `Division`. Pomocí této instance jsou určovány dělicí body při dělení lineárních stupnic. V této prací je dělení uskutečňováno vždy v půlce intervalu. Prvky, které jsou do grid souboru vkládány, musí implementovat rozhraní `GridFileElement` – disponují souřadnicemi x a y.

Grid soubor obsahuje adresář `directory` a dvě lineární stupnice `scaleX` a `scaleY`. Z vizualizačních důvodů jsou meze – minimální a maximální souřadnice x a y – grid souboru nastaveny pevně. Obecně nemusí být hranice grid souboru pevně dány, do grid souboru tak mohou být ukládány libovolné klíče.

V následujících kapitolách bude postupně popsáno vkládání, hledaní a odebírání prvků včetně příslušných reorganizací – dělení a spojování datových bloků. Implementace je

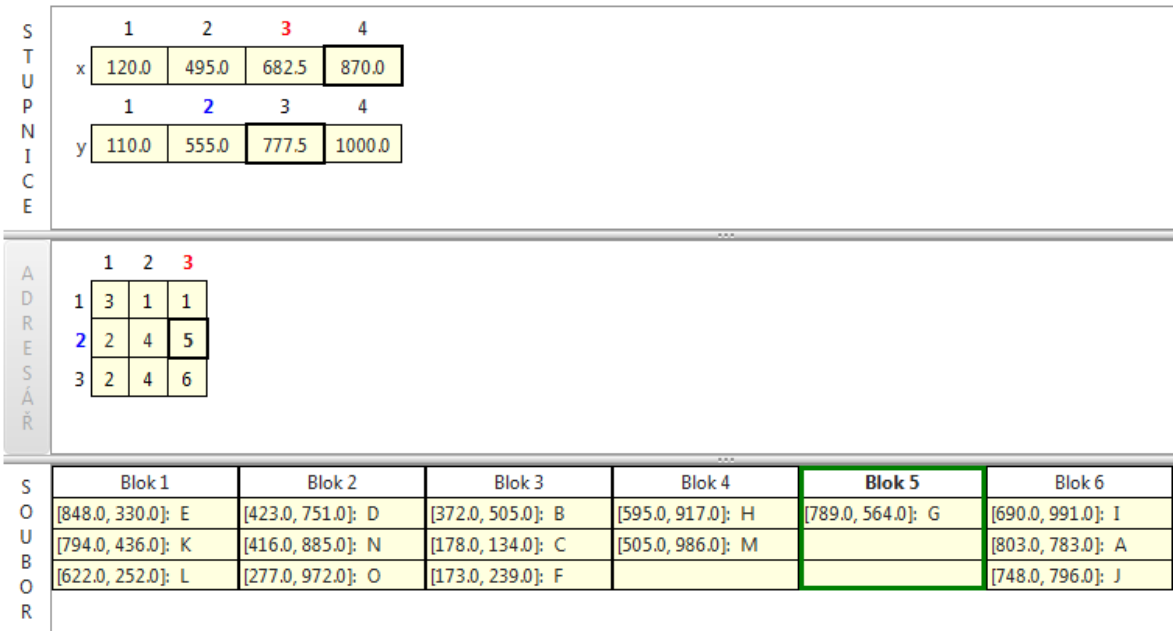
částečně inspirována diplomovou prací pana Michka (2011). Případné ukázky budou prováděny na datech z přednastavené sady číslo 1 s maximální kapacitou bloku 3 záznamy.

10.1 Operace Vlož

Při vkládání prvku je nejprve v lineárních stupnicích pro každou dimenzi vyhledán index dle klíče vkládaného prvku. Pokud máme k dispozici všechny indexy z lineárních stupnic, můžeme dle nich přistoupit do konkrétní buňky v grid adresáři. Z této buňky zjistíme identifikátor datového bloku. Mohou nastat dvě situace:

- vkládaný prvek se do datového bloku vejde,
- datový blok je již plný.

Na obrázku níže je ilustrováno vkládání prvku s klíčem $x = 700$ a $y = 600$.



Obrázek 61 – Grid soubor – vložení záznamu do prázdného bloku

Zdroj: vlastní

Na stupnici X najdeme první větší hodnotu, než je klíč $x = 700$, na stupnici Y první větší hodnotu, než je klíč $y = 600$. Indexy nalezených hodnot snížíme o jedničku a použijeme je pro přístup do grid adresáře. V adresáři se na pozici [3, 2] nachází identifikátor datového bloku 5. Datový blok číslo 5 načteme do paměti. Jelikož se v tomto bloku nachází ještě volné místo, můžeme vkládaný prvek zapsat do tohoto bloku. Vkládání je ukončeno.

Druhou situaci zachycuje obrázek 62 – v tomto případě vkládáme do grid souboru prvek s klíčem $x = 700$ a $y = 800$. Analogicky, jako v příkladu výše, načteme příslušný datový blok (6) do paměti, ten je ovšem plný.

STUPNICE

	1	2	3	4
x	120.0	495.0	682.5	870.0

	1	2	3	4
y	110.0	555.0	777.5	1000.0

ADRESÁŘ

	1	2	3
1	3	1	1
2	2	4	5
3	2	4	6

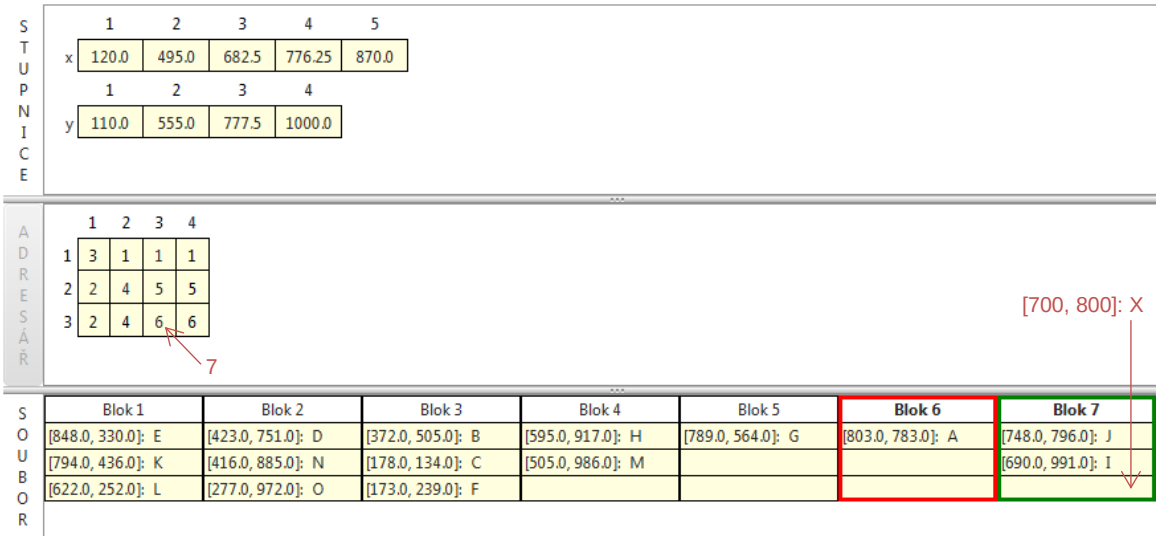
SOUBOR

Blok 1	Blok 2	Blok 3	Blok 4	Blok 5	**Blok 6**
[848.0, 330.0]: E	[423.0, 751.0]: D	[372.0, 505.0]: B	[595.0, 917.0]: H	[789.0, 564.0]: G	[690.0, 991.0]: I
[794.0, 436.0]: K	[416.0, 885.0]: N	[178.0, 134.0]: C	[505.0, 986.0]: M		[803.0, 783.0]: A
[622.0, 252.0]: L	[277.0, 972.0]: O	[173.0, 239.0]: F			[748.0, 796.0]: J

Obrázek 62 – Grid soubor – před vložením záznamu do plného bloku

Zdroj: vlastní

Abychom mohli prvek do grid souboru uložit, musíme rozdělit lineární stupnici a adresář. V tomto případě je na řadě dimenze 1, tedy stupnice X. V bloku 6 se mohou vyskytovat klíče, jejichž hodnota $x \in \langle 682{,}5;\ 870)$. Tento prostor musíme rozdělit. Prostor je dělen tak dlouho, dokud nejsou klíče rozděleny tak, aby se po skupinách vešly do jednotlivých datových bloků. V tomto případě stačí do lineární stupnice X přidat pouze jeden nový dělící bod – 776,25. Zároveň musíme rozdělit i adresář, a to tak, že jej „roztáhneme“ ve směru dělené dimenze. Identifikátory datových bloků v nových buňkách jsou zkopírovány ze sousedních buněk v příslušné dimenzi. Nyní je nutné přesunout prvky s klíči, které už do původního datového bloku nepatří, do nově alokovaného datového bloku 7 (J, I). V grid adresáři na pozici [3, 3] nastavíme identifikátor nově alokovaného datového bloku. Prvek s klíčem $x = 700$ a $y = 800$ je vložen do bloku 7. Grid soubor bude po rozdělení (zjemnění) stupnice X a adresáře vypadat následovně:



Obrázek 63 – Grid soubor – po vložení záznamu do plného bloku

Zdroj: vlastní

Může nastat i situace, kdy bude prvek příslušet do plného datového bloku, ale přesto neproběhne dělení. To se stane, pokud některý ze sousedních datových bloků v grid adresáři je stejný jako ten, do kterého vkládáme. Současně však musí platit, že některé klíče budou přesunuty do tohoto souseda. Tato situace by například nastala, pokud bychom do grid souboru na obrázku 63 vkládali prvek s klíčem $x = 500$ a $y = 150$ (grid buňka [2, 1]).

Průběh vkládání prvku je zachycen v následujícím pseudokódu. Vložení je iniciováno zavoláním funkce `vlož(prvek)`. Pokud padne prvek do plného datového bloku, předá se zpracování funkci `vložDoPlného`, ve které probíhá případné dělení stupnic a adresáře.

```
funkce vlož(prvek):
      indexX = stupniceX.dejIndex(prvek.X);
      indexY = stupniceY.dejIndex(prvek.Y);
      blokId = adresář.dejIdBloku(indexX, indexY);
      blok = soubor.dejBlok(blokId);

      pokud blok.jePlný pak
            vložDoPlného(blokId, prvek, indexX, indexY);
      jinak
            soubor.vložZáznam(blokId, prvek);
konec;
```

```
funkce vložDoPlného(plnýId, prvek, indexX, indexY):
      novýId = soubor.alokujNovýBlok();

      dokud dělit(indexX, indexY) dělej
            pokud dělenáDimenze = 1 pak
                  rozděl stupnici X;
                  rozděl adresář v dimenzi X;
                  aktualizuj indexX;
            jinak
                  rozděl stupnici Y;
                  rozděl adresář v dimenzi Y;
                  aktualizuj indexY;

            změň dělenou dimenzi;

      z plnýId přesuň do novýId ty klíče, které už do plnýId nepatří;
      nastav v adresář na pozici [indexX, indexY] novýId;
      soubor.vložZáznam(novýId, prvek);
konec;

funkce dělit(indexX, indexY):
      blokId = adresář.dejIdBloku(index, indexY);
      blok = soubor.dejBlok(blokId);

      fyzickýPočet = blok.početZáznamů;
      logickýPočet = počet patřících záznamů do blok (po proběhlém dělení);

      pokud fyzickýPočet = logickýPočet pak
            dělit dál; // vrať ano
      jinak
            nedělit; // vrať ne
konec;
```

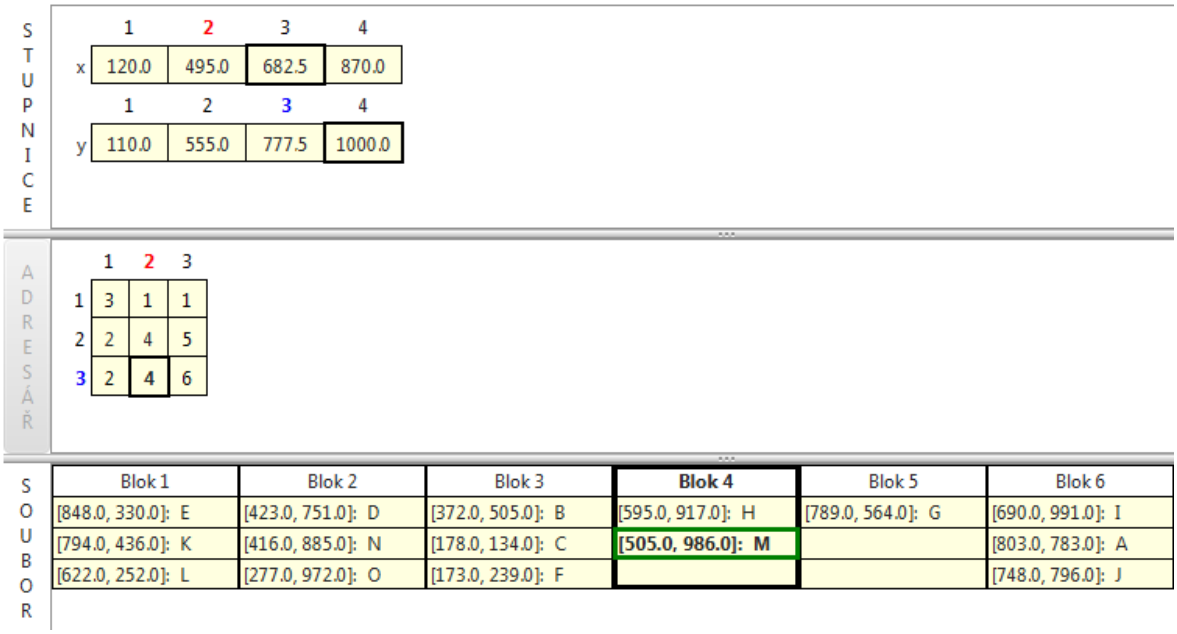
10.2 Operace Najdi

Grid soubor je schopný vyhledat prvek s využitím nejvýše dvou diskových operací. První přístup je do grid adresáře (pokud se celý nenachází v operační paměti) a druhý již do příslušného datového bloku s hledaným prvkem. Grid soubor také umožňuje efektivní intervalové vyhledávání, během kterého jsou nalezeny prvky, jejichž klíče jsou z určeného intervalu.

Princip vyhledávání je následující. Dle hodnoty hledaného klíče zjistíme příslušné indexy na lineárních stupnicích. Pomocí zjištěných indexů přistoupíme do grid buňky v adresáři. V této buňce se již nachází identifikátor datového bloku, ve kterém by se měl hledaný prvek nacházet (pokud existuje). Zbývá načíst odpovídající datový blok do operační paměti a sekvenčně projít všechny uložené záznamy. Pokud projdeme všechny záznamy a nenalezneme prvek s odpovídajícím klíčem, pak se hledaný prvek v grid souboru nenachází. Níže se nachází pseudokód vyhledávání určitého prvku.

```
funkce najdi(x, y):
      indexX = stupniceX.dejIndex(x);
      indexY = stupniceY.dejIndex(y);
      blokId = adresář.dejIdBloku(indexX, indexY);
      blok = soubor.dejBlok(blokId);

      záznamy = blok.dejZáznamy();

      pro každý záznam ze záznamy dělej:
            pokud záznam.X = x a záznam.Y = y pak
                  vrať záznam;

      vrať nenalezeno;
konec;
```

Ilustrace vyhledání prvku s klíčem $x = 505$ a $y = 986$ se nachází na obrázku níže.



Obrázek 64 – Grid soubor – vyhledání prvku

Zdroj: vlastní

10.3 Operace Intervalové vyhledávání

Intervalové vyhledávání funguje obdobně s tím rozdílem, že nepřistupujeme pouze do jednoho datového bloku. U intervalového vyhledávání máme specifikován minimální a maximální klíč v obou dimenzích. Pro tyto čtyři hodnoty zjistíme příslušné indexy na lineárních stupnicích. Zjištěné čtyři indexy určí obdélníkovou oblast v grid adresáři. V této oblasti projdeme všechny grid buňky. Současně zpřístupňujeme odpovídající datové bloky a procházíme jejich záznamy. Každý záznam porovnáváme s hledaným intervalem. Pokud záznam vyhovuje hledanému intervalu, pak tento záznam přidáme do množiny výsledků. Vizualizace intervalového vyhledávání nebyla předmětem práce. Pseudokód intervalového vyhledávání je následující:

```
funkce najdiInterval(minX, maxX, minY, maxY):
      indexXMin = stupniceX.dejIndex(minX);
      indexXMax = stupniceX.dejIndex(maxX);
      indexYMin = stupniceY.dejIndex(minY);
      indexYMax = stupniceY.dejIndex(maxY);

      pro všechny i od indexXMin do indexXMax dělej:
            pro všechny j od indexYMin do indexYMax dělej:
                  blokId = adresář.dejIdBloku(i, j);
                  blok = soubor.dejBlok(blokId);
                  záznamy = blok.dejZáznamy();

                  pro každý záznam ze záznamy dělej:
                        pokud záznam.X = x a záznam.Y = y pak
                              přidej záznam do výsledků;

      vrať výsledky;
konec;
```

10.4 Operace Odeber

Při odebírání musíme prvek nejdříve vyhledat. To provedeme stejným způsobem, který byl představen v předchozí kapitole. Pokud byl prvek v grid souboru nalezen, pak jej odebereme z příslušného datového bloku. Nyní musíme zkontrolovat, zda datový blok, ze kterého jsme odebrali prvek, má stále požadovanou obsazenost. Pokud zjistíme, že zaplnění bloku je pod požadovanou hranicí, pokusíme se jej sloučit s některým jeho sousedem (dle grid adresáře).

Implementovaný systém slučování bloků je tzv. „buddy systém“ (viz kapitola 5.3.5), tedy pokusíme se daný blok sloučit s jeho levým, případně horním sousedem v grid adresáři. Nejprve se pokusíme sloučit blok s jeho levým sousedem. Pokud levý soused existuje a součet záznamů ve slučovaném a sousedním bloku nepřesahuje maximální kapacitu datového bloku, pak provedeme sloučení s tímto levým sousedem. Jestliže nebyla podmínka pro levého souseda splněna, opakujeme výše uvedený postup pro horního souseda. Pokud nebyla podmínka splněna ani pro horního souseda, pak blok nelze sloučit („buddy systémem“) a algoritmus končí. Z toho vyplývá, že datový blok, který se nachází v grid adresáři na pozici [0, 0], nemůže být nikdy sloučen (má totiž pouze pravého a dolního souseda).

Při odstraňování bloků z blokově orientovaného souboru máme k dispozici více strategií, jak zacházet s uvolněnými bloky. V zásadě můžeme:

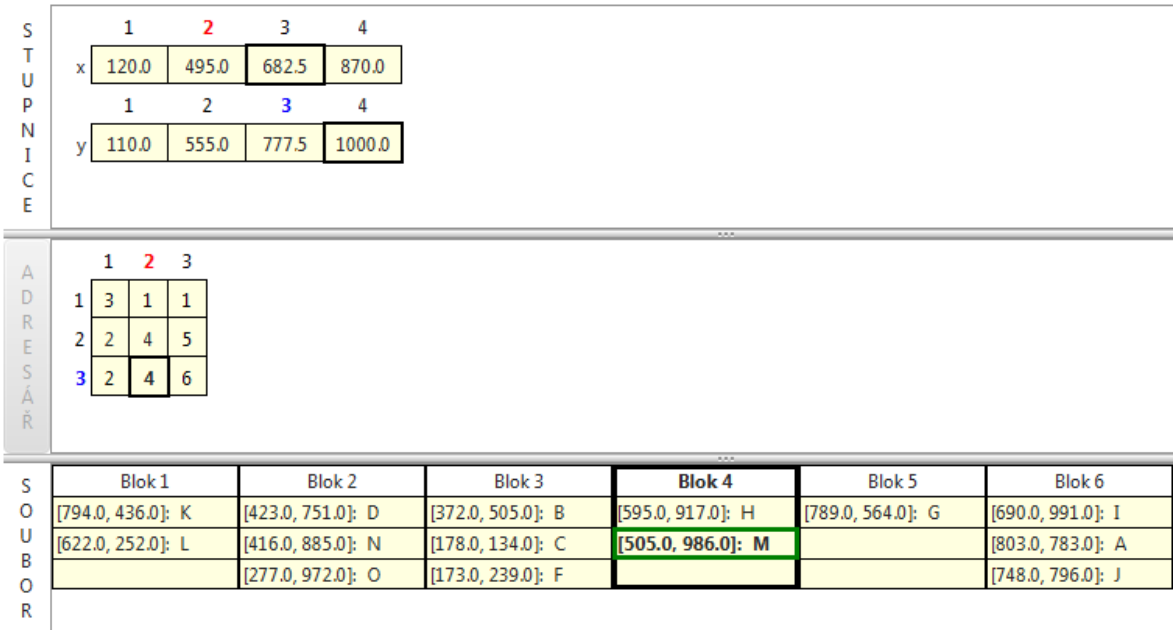
- volné bloky ponechávat v souboru a uchovávat seznam těchto volných bloků,
- volné bloky odstraňovat, tedy zmenšovat blokově orientovaný soubor.

Jak zacházet s uvolněnými bloky, záleží na povaze určité aplikace. V této implementaci jsou volné bloky odstraňovány. Volný blok odstraníme tak, že do něj přesuneme záznamy z posledního bloku a až poslední blok ze souboru odstraníme („odřízneme jej“).

Nyní víme, se kterým sousedem blok sloučit a víme, jak zacházet s prázdnými bloky. K dokončení sloučení bloků zbývá pár jednoduchých kroků:

1) ze sousedního bloku (v grid adresáři) přesuneme záznamy do slučovaného bloku,
2) do sousedního bloku přesuneme záznamy z posledního bloku,
3) odstraníme poslední blok ze souboru,
4) aktualizujeme identifikátory změněných bloků v grid adresáři.

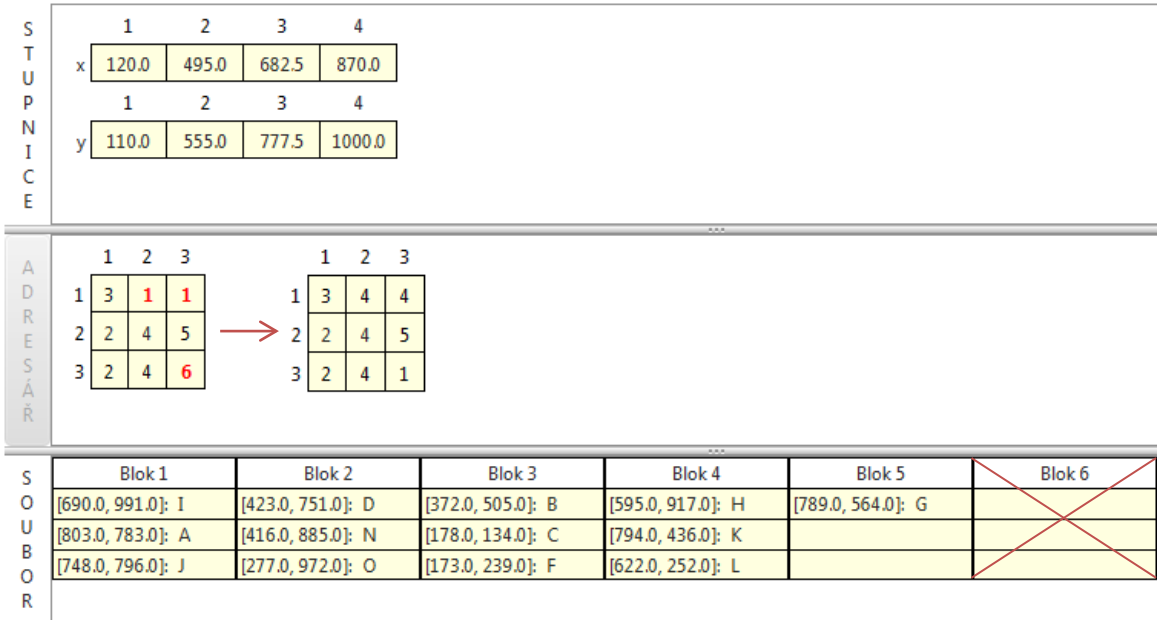
Odebírejme z grid souboru na obrázku níže prvek s klíčem $x = 505$ a $y = 986$.



Obrázek 65 – Grid soubor – před odebráním prvku

Zdroj: vlastní

Jelikož je požadována obsazenost bloku alespoň 70 %, měl by být blok 4 po odebrání prvku sloučen. V grid adresáři vidíme dva adepty: levý sousední blok (2) a horní sousední blok (1). Přednost má levý soused, ale s ním nelze Blok 4 sloučit, protože součet záznamů v těchto blocích je větší než kapacita bloku. S horním sousedním blokem se nám sloučení již podaří, protože počet záznamů v obou blocích nepřesahuje kapacitu bloku. Sloučení proběhne tak, že záznamy z bloku 1 budou přesunuty do bloku 4 a záznamy z posledního bloku (6) budou přesunuty do bloku 1. Nakonec odstraníme blok 6 ze souboru a aktualizujeme identifikátory změněných bloků v grid adresáři. Strukturu grid souboru po odebrání prvku s klíčem $x = 505$ a $y = 986$ zachycuje obrázek 66.



Obrázek 66 – Grid soubor – po odebrání prvku

Zdroj: vlastní

Závěr

Diplomová práce se zabývala vizualizacemi evolucí algoritmů vybraných datových struktur. Nejprve byla provedena analýza existujících vizualizací. Bylo zjištěno, že neexistuje databáze, která by obsahovala všechny vizualizace běžných datových struktur. Během analýzy byly také zjištěny některé nedostatky. Mezi hlavní nalezené nedostatky patří zejména nepříliš komfortní ovládání a absence předpřipravených či generovaných dat. S přihlédnutím ke zjištěným nedostatkům proběhla implementace vizualizací vybraných datových struktur: binomické haldy, Huffmanova stromu, B-stromu a grid souboru.

V teoretické části práce byly postupně popsány abstraktní datové typy prioritní fronta a tabulka a následně jejich vybrané implementace. Součástí teoretické části práce je také kapitola o kódování textu s podrobnějším zaměřením na Huffmanovo kódování. V této části se nacházejí pouze obecné informace a filozofie vybraných datových struktur. Popis konkrétních implementací a fungování jednotlivých algoritmů je obsahem praktické části.

V praktické části práce byla nejprve provedena volba vhodné implementační technologie. Technologií zvolenou pro vizualizaci animací se stala JavaFX. Následně byly představeny dva odlišné přístupy vizualizace datových struktur. Zvolen byl přístup, který je založen na generování událostí a jejich následném zpracovávání. Pomocí zvolené technologie a přístupu byly následně naimplementovány vizualizace evolucí algoritmů vybraných datových struktur. Součástí praktické části práce je popis vybraných datových struktur z implementačního hlediska s důrazem na fungování jednotlivých algoritmů.

Výstupem práce jsou čtyři vizualizace vybraných datových struktur: binomické haldy, Huffmanova stromu, B-stromu a grid souboru. Tyto vizualizace jsou součástí webové aplikace, jejíž hlavním cílem je interaktivní podpora výuky datových struktur a algoritmů. Vytvořené vizualizace odstraňují nedostatky všech analyzovaných vizualizací – zejména umožňují pohodlné ovládání klávesovými zkratkami a disponují jak předpřipravenými daty, tak i generátorem náhodných dat. Výsledné vizualizace jsou plynule animovány a umožňují jak regulaci rychlosti animace, tak i její pozastavování v libovolném čase.

Jako autor práce jsem měl možnost se hlouběji seznámit s problematikou datových struktur a algoritmů. Blíže jsem nastudoval vybrané datové struktury a poznal do detailů principy jejich fungování. Během tvoření vizualizací jsem nabyl zkušenosti jak se zobrazováním datových struktur, tak i s tvorbou animací. Nemalým přínosem bylo bližší seznámení se s technologií JavaFX, která poskytuje velmi pěkné nástroje k tvorbě uživatelských rozhraní, animací, práci s multimédii apod. Tato technologie má dle mého názoru velmi perspektivní budoucnost.

Veškeré cíle práce byly splněny. V teoretické části byl proveden přehled vybraných abstraktních datových typů a jejich některých implementací. V rámci praktické části byly vytvořeny vizualizace vybraných datových struktur, které je možné provádět přímo uvnitř webového prohlížeče. Vytvořené vizualizace jsou součástí webové aplikace, která je dostupná jak studentům naší univerzity, tak i širší veřejnosti se zájmem o datové struktury.

Literatura

Apache Flex [online]. 2014 [cit. 4. 5. 2014]. Dostupné z: http://flex.apache.org/

BRIGHT, Peter. Adobe lays out the future for Flash: a platform for the next 5-10 years. In: *Ars Technica* [online]. 23. 2. 2012 [cit. 4. 5. 2014]. Dostupné z: http://arstechnica.com/business/2012/02/adobe-lays-out-the-future-for-flash-a-platform-for-the-next-5-10-years/

B-tree. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2014, 17. 4. 2014 [cit. 26. 4. 2014]. Dostupné z: http://en.wikipedia.org/wiki/B-tree

Class-based programming. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001-2014, 21. 2. 2014 [cit. 3. 5. 2014]. Dostupné z: http://en.wikipedia.org/wiki/Class-based_programming

CORMEN, Thomas H. aj. *Introduction to algorithms*. 2nd ed. Cambridge: MIT Press, 2001, ISBN 02-620-3293-7.

CORMEN, Thomas H. aj. *Introduction to algorithms*. 3rd ed. Cambridge: MIT Press, 2009, ISBN 978-0-262-03384-8.

Dart [online]. 2014 [cit. 3. 5. 2014]. Dostupné z: https://www.dartlang.org/

DEA, Carl. *JavaFX 2.0: Introduction by Example*. New York: Apress, 2011. ISBN 978-143-0242-581.

Details of the object model. In: *Mozilla Developer Network* [online]. 2014, 7. 3. 2014 [cit. 3. 5. 2014]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model

EPPERSON, Kraettli L. Microsoft Silverlight is Dead; Long Live Silverlight!. In: *TechVentureGeek* [online]. 28. 3. 2013 [cit. 4. 5. 2014]. Dostupné z: http://techventuregeek.com/software-reviews/microsoft-silverlight-is-dead-long-live-silverlight/#.U2WGufk_t8F

GOODRICH, Michael T. a Roberto TAMASSIA. *Data structures and algorithms in Java*. 4th ed. Hoboken, New Jersey: Wiley, 2006. ISBN 04-717-3884-0.

JACKSON, Joab. Adobe donates Flex to Apache. In: *TechWorld* [online]. 17. 11. 2011 [cit. 5. 5. 2014]. Dostupné z: http://www.techworld.com.au/article/407714/adobe_donates_flex_apache/

JavaFX 2 Documentation [online]. 2014 [cit. 6. 5. 2014]. Dostupné z: http://docs.oracle.com/javafx/2/

JavaFX FAQ. *Oracle* [online]. 2014 [cit. 5. 5. 2014]. Dostupné z: http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html

Java-support Statistics. *W3resource* [online]. 2014 [cit. 5. 5. 2014]. Dostupné z: http://www.w3resource.com/browsers/java-support.php

KAVIČKA, Antonín. *Sylaby přednášek předmětu „Datové struktury a algoritmy“ v magisterském studiu*. Univerzita Pardubice, 2011.

KAVIČKA, Antonín. *Sylaby přednášek předmětu „Datové struktury“ v bakalářském studiu*. Univerzita Pardubice, 2010.

KUČERA, Luděk. *Algovize: aneb procházka krajinou algoritmů*. 1. vyd. Praha: Univerzita Karlova, 2009. ISBN 978-80-902938-5-4.

LEWIS, Harry R. a Larry DENENBERG. *Data Structures and Their Algorithms*. Harvard University: HarperCollinsPublisber, 1991. ISBN 0-673-39736-X.

MENSA ČR, 2014. Frekvenční tabulka písmen české abecedy. In: *Mensa ČR* [online]. 2014 [cit. 23. 4. 2014]. Dostupné z: http://www.mensa.cz/volny-cas/hlavolamy/detska-sifrovaci-liga/frekvencni-tabulka-pismen

MICHEK, Tomáš. *Datové struktury pro uchovávání geografických dat*. Pardubice, 2011. Diplomová práce. Univerzita Pardubice.

NIEVERGELT, J. a H. HINTERBERGER. *The Grid File: An Adaptable, Symmetric Multikey File Structure*. University of Toronto: Institut für Informatik, ETH and K. C. Sevcik, 1984. Dostupné z: http://infolab.usc.edu/csci587/Fall2011/papers/Grid-Index.pdf

SAMET, Hanan. *Foundations of multidimensional and metric data structures*. San Francisco: Morgan Kaufmann, 2006. ISBN 978-012-3694-461.

SOUKUP, Tomáš. Dart 1.0. Jazyk pro web, který chce nahradit Javascript. In: *Živě.cz* [online]. 14. 11. 2013 [cit. 3. 5. 2014]. Dostupné z: http://www.zive.cz/bleskovky/dart-10-jazyk-pro-web-ktery-chce-nahradit-javascript/sc-4-a-171324/default.aspx

ULANOFF, Lance. It's Official: Flash Mobile Player is Dead. In: *Mashable* [online]. 9. 11. 2011 [cit. 4. 5. 2014]. Dostupné z: http://mashable.com/2011/11/09/its-official-flash-mobile-player-is-dead/

WÄHNER, Kai. When to use JavaFX 2 instead of HTML5 for a Rich Internet Application (RIA)?. In: *DZone* [online]. 23. 4. 2012 [cit. 3. 5. 2014]. Dostupné z: http://java.dzone.com/articles/when-use-javafx-2-instead

WEAVER, James L. aj. *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology*. New York: Apress, 2012, ISBN 978-1-4302-6873-4.

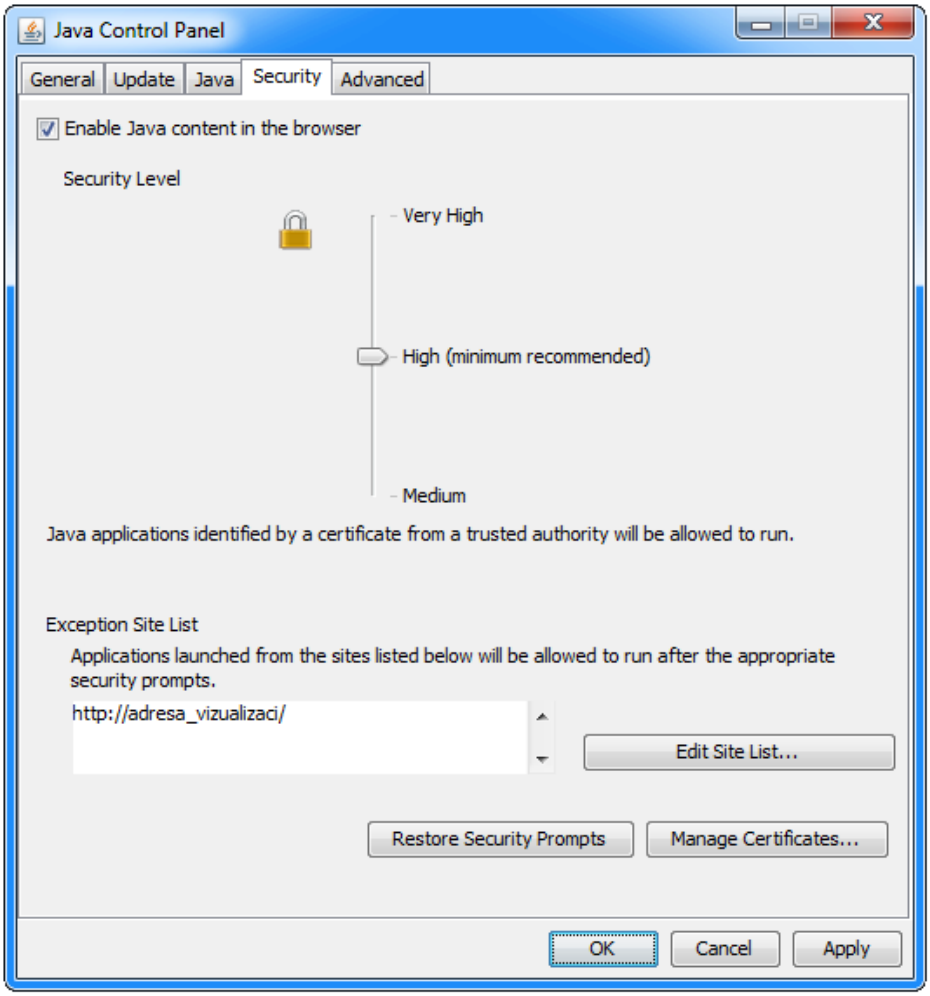
Příloha A – Uživatelský manuál

Pro běh všech vizualizací je vyžadována Java SE 7u45 (JavaFX 2.2) a vyšší. Od verze 7 aktualizace 51 jsou všechny nepodepsané Java Web Start aplikace ve výchozím nastavení blokovány. Vizualizace nejsou podepsány certifikační autoritou, tudíž jejich běh musí být povolen v Java nastavení. Povolit běh nepodepsaných aplikací lze provést dvěma způsoby:

- snížit úroveň zabezpečení, nebo
- přidat bezpečnostní výjimku.

V operačním systému Windows 7 je postup následující (testováno na Java 7u51 a 7u55):

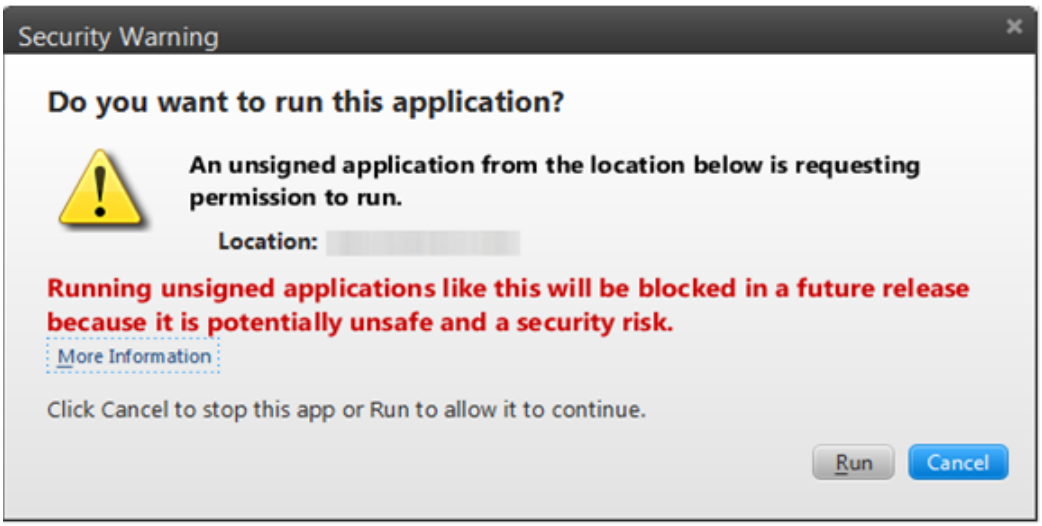
- Start – Všechny programy – Java – Configure Java,
- na záložce Security:
    - snížit Security Level na Medium, nebo
    - přidat URL adresu vizualizací do Exception Site List.



Obrázek 67 – Java nastavení

Zdroj: vlastní

Po provedení výše uvedených úkonů by se při spuštění vizualizací měla zobrazit následující bezpečnostní hláška:



Obrázek 68 – Bezpečnostní varování
Zdroj: vlastní

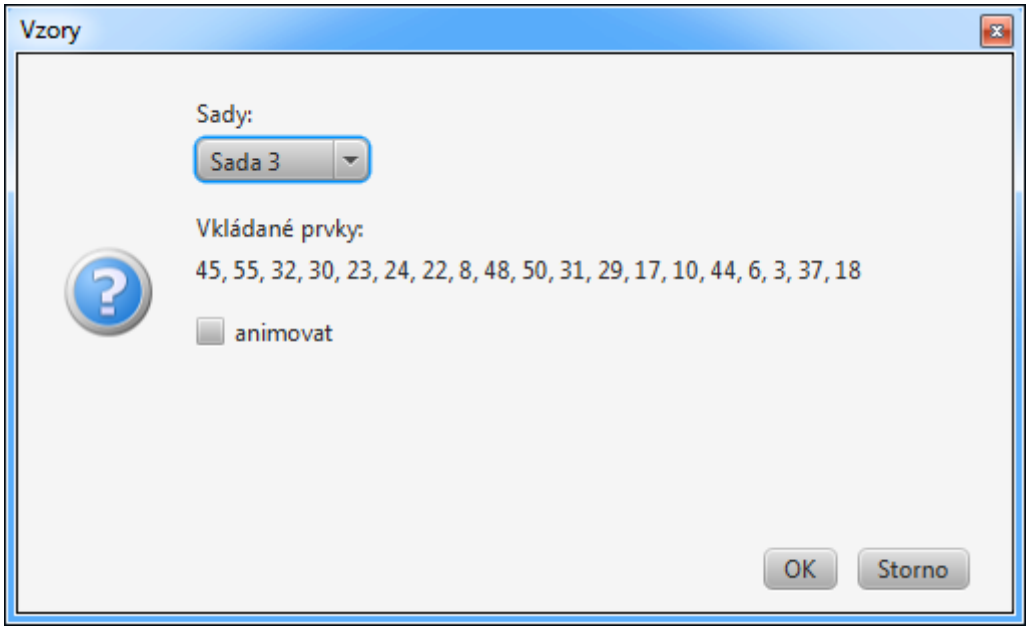
Po stisknutí tlačítka Run by měly vizualizace bez problémů fungovat.

Z neznámého důvodu nelze po aktualizaci 55 spouštět Java Web Start aplikace z lokálně uložené kopie – ani přes soubor JNLP, ani z lokálně otevřené webové stránky s vizualizací (problém s přístupem ke zdrojům). Lokálně je možné vizualizace spustit po umístění na lokální webový server, nebo přes standardní soubor JAR.

Každá vizualizace disponuje podobným uživatelským rozhraním. Společné ovládací prvky všech vizualizací jsou:

- Krokovat – zapíná/vypíná krokování animace,
- Krok – provede další krok (pokud je zapnuto krokování),
- Rychlost – ovládání rychlosti animace,
- Reset – vymazání všech dat,
- Vzory – zobrazí dialog se vzorovými daty,
- Nápověda ( ) – zobrazí nápovědu (ovládání a klávesové zkratky).

Na obrázku níže je zobrazen dialog pro dávkové načítání dat. Na výběr jsou statické předpřipravené sady dat a sady, jejichž data jsou generována náhodně. Pokud je zatržena volba animovat, proběhne plynulá animace, jinak budou data nahrána téměř okamžitě.



Obrázek 69 – Dialog pro dávkové načítání dat

Zdroj: vlastní

Společné klávesové zkratky jsou:

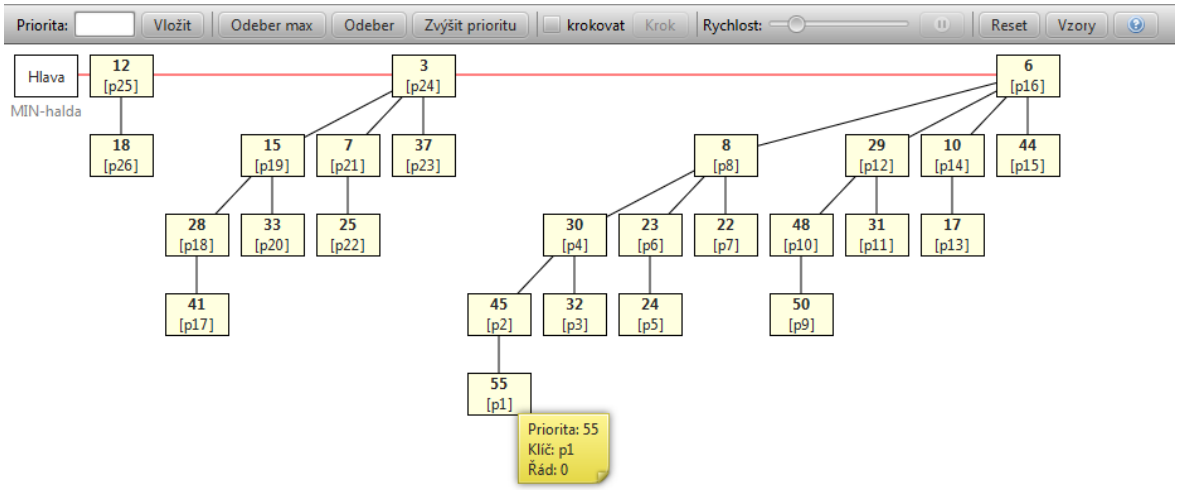
- F5 – zapnout/vypnout krokování,
- F6 – další krok,
- F7 – snížit rychlost animace,
- F8 – zvýšit rychlost animace,
- MEZERNÍK – pozastavení/spuštění animace.

Specifické klávesové zkratky pro každou vizualizaci se nachází v dialogu nápovědy (tlačítko).

Binomická halda

Implicitně je vytvořena binomická min-halda. Priorita vkládaných prvků musí být z oboru celých čísel. Klíče jednotlivých prvků jsou generovány automaticky (jsou zobrazeny v hranatých závorkách). Podrobnější informace o jednotlivých uzlech se zobrazí po najetí kurzoru myši. Zobrazena je priorita prvku, klíč a řád daného binomického stromu. Typ haldy (min-halda/max-halda) lze zvolit při resetu. Podporovány jsou následující operace:

- Vložit – vloží prvek s prioritou v poli Priorita do haldy,
- Odeber max – odebere prvek s maximální prioritou z haldy,
- Odeber – odebere libovolný prvek z haldy (zobrazeno okno pro zadání klíče prvku),
- Zvýšit prioritu – zvýší prioritu libovolnému prvku v haldě (zobrazeno okno pro zadání klíče a následně pro zadání nové priority).



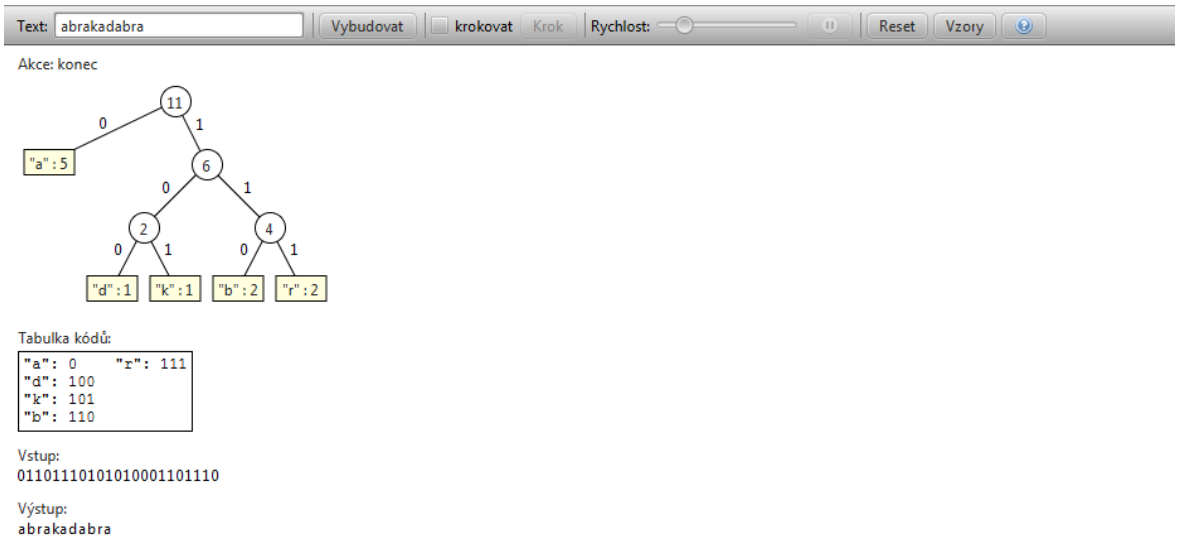
Obrázek 70 – GUI – binomická halda

Zdroj: vlastní

Huffmanův strom

Huffmanův strom disponuje pouze jednou operací:

- Vybudovat – zahájí postupné budování Huffmanova stromu pro text zadaný v poli Text. Následuje vytvoření kódů, zakódování textu a dekódování.



Obrázek 71 – GUI – Huffmanův strom

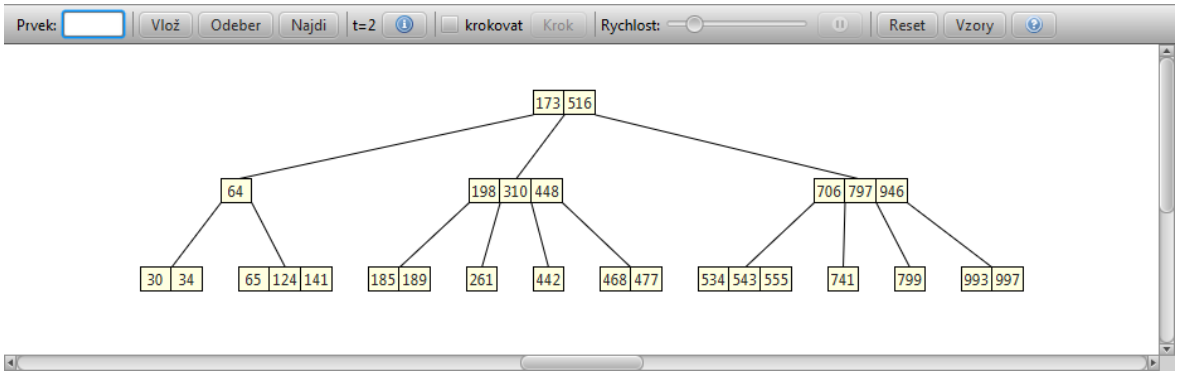
Zdroj: vlastní

B-strom

Na začátku je vytvořen B-strom s parametrem $t = 2$. Informace o aktuálním nastavení parametru t se zobrazuje vedle tlačítka Najdi. Parametr t lze změnit při resetu. Klíče vkládaných prvků musí být z oboru celých čísel. B-strom lze přibližovat a oddalovat zadržením tlačítka CTRL a rotací kolečka myši v animační scéně. Pod tlačítkem s ikonou se nachází informace o implementaci tohoto B-stromu. Zobrazen je popis

parametru *t*, minimální a maximální počty klíčů, resp. potomků a popis štěpení a fúze uzlů. Operace B-stromu jsou následující:

- Vložit – vloží prvek s klíčem v poli Prvek do B-stromu,
- Odeber – odebere prvek s klíčem v poli Prvek z B-stromu,
- Najdi – vyhledá prvek s klíčem v poli Prvek v B-stromu.



Obrázek 72 – GUI – B-strom

Zdroj: vlastní

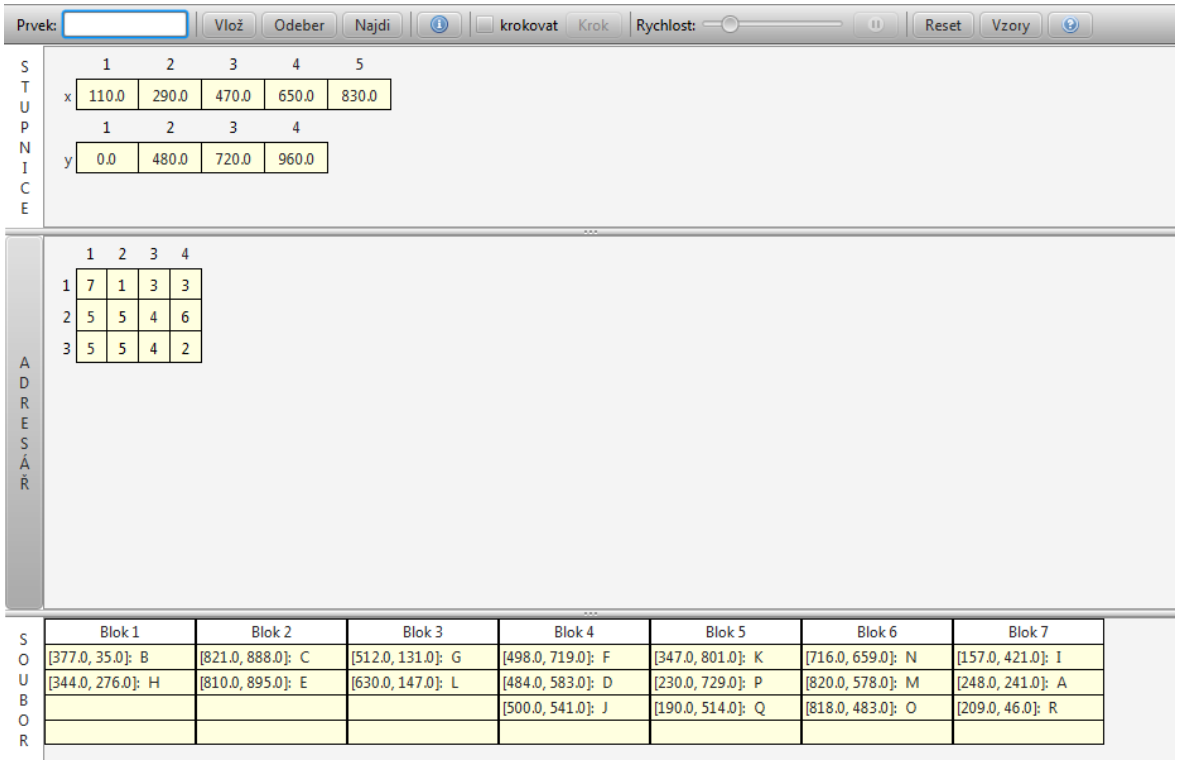
Grid soubor

Po spuštění vizualizace je vytvořen grid soubor s kapacitou bloku 3 a rozsahem $x \in \langle 0; 100)$ a $y \in \langle 0; 100)$. Tyto vlastnosti lze změnit při resetu. Informace o aktuálním nastavení lze zobrazit stisknutím tlačítka s ikonou . Zobrazena je také informace o použité strategii dělení lineárních stupnic a o použité strategii slučování bloků. Vstupní pole Prvek vyžaduje zadání prvku ve tvaru:

- pro operaci Vlož – souřadnice_x␣souřadnice_y␣název, např.: 12.5 33.1 bod 123,
- pro operaci Odeber a Najdi – souřadnice_x␣souřadnice_y, např.: 12.5 33.1.

Podporovány jsou následující operace:

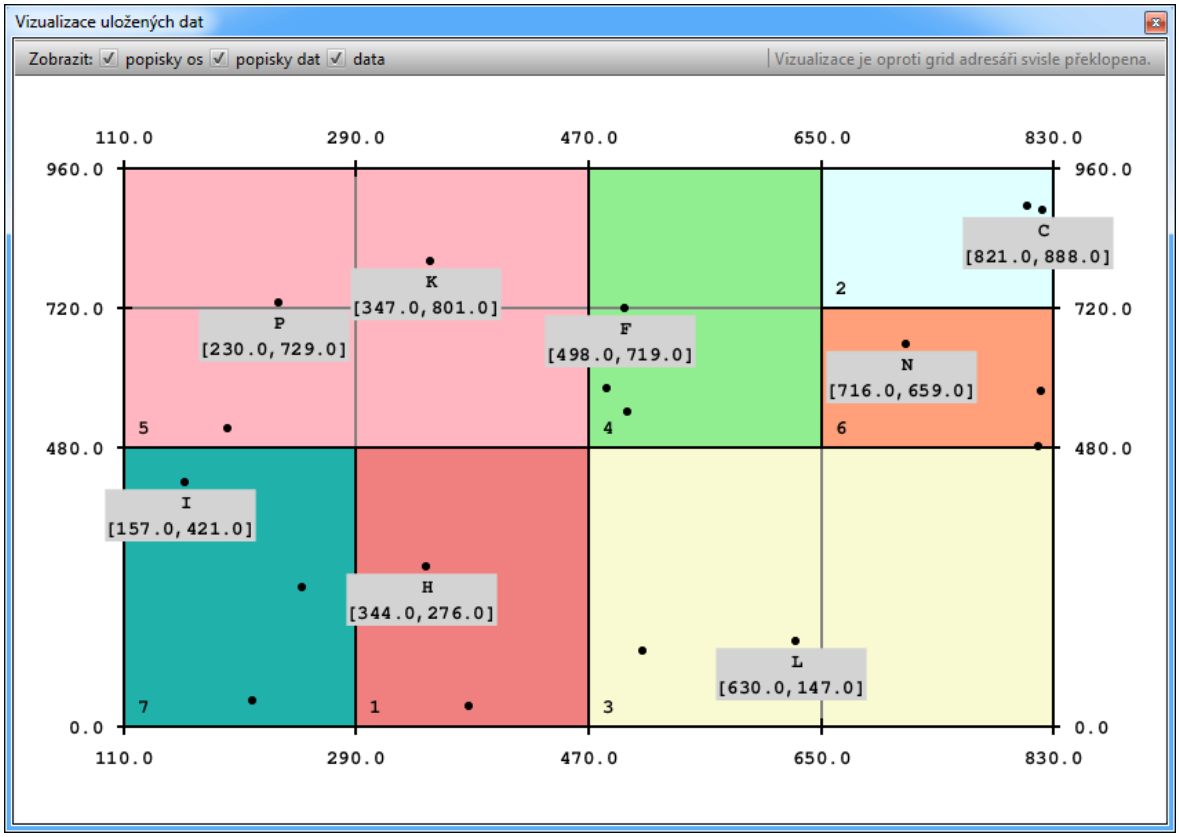
- Vlož – vloží prvek do grid souboru,
- Odeber – odebere prvek se zadanými souřadnice z grid souboru,
- Najdi – pokusí se vyhledat prvek se zadanými souřadnicemi v grid souboru.



Obrázek 73 – GUI – grid soubor

Zdroj: vlastní

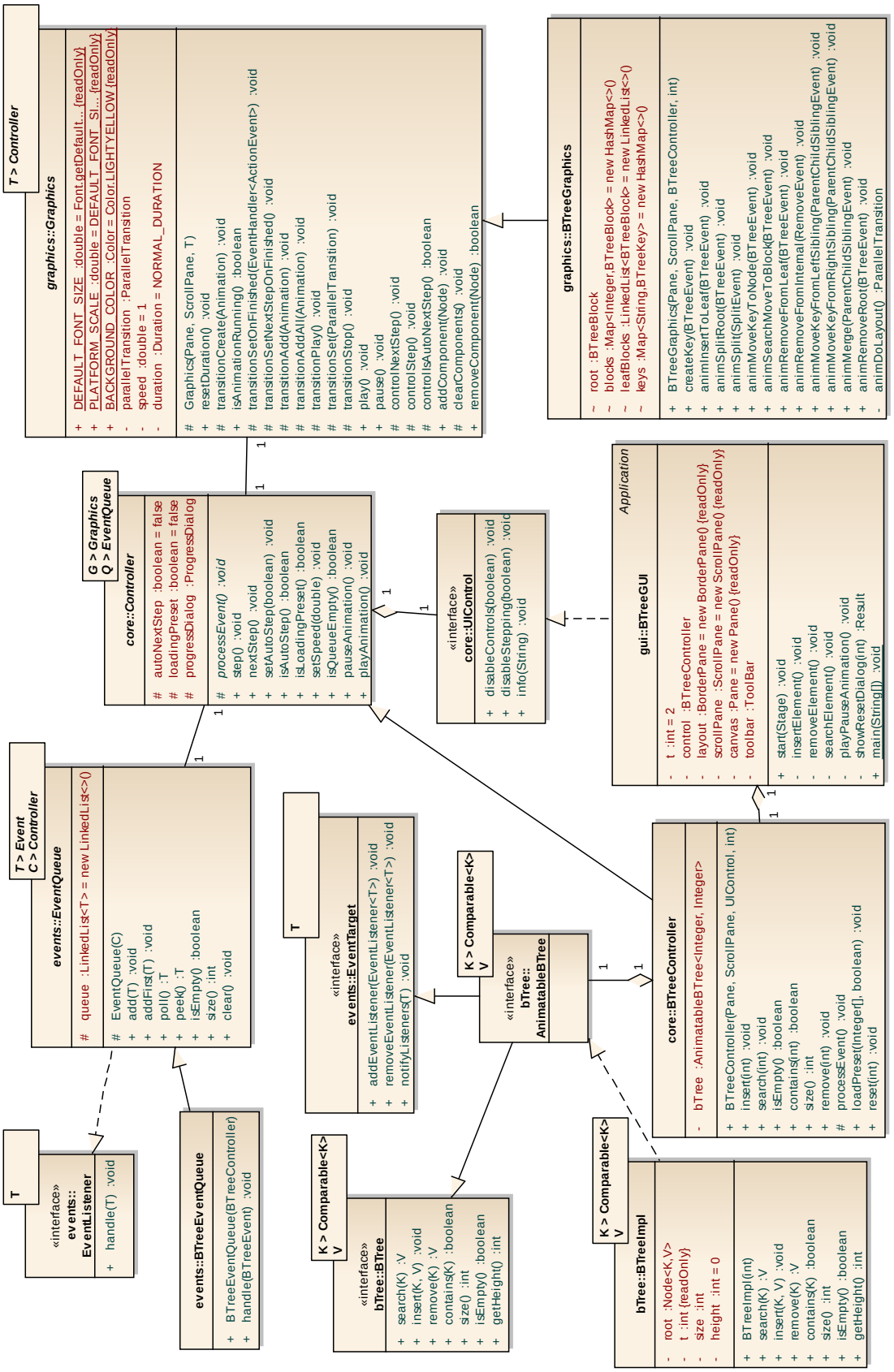
Stisknutím tlačítka ADRESÁŘ se zobrazí vizualizace uložených dat. Zobrazeny jsou jednotlivé záznamy v rovinné kartézské soustavě souřadnic. Zachycena je i struktura grid adresáře a příslušnost jednotlivých záznamů do datových bloků. Jednotlivé datové bloky jsou barevně odlišeny. Jejich čísla korespondují s čísly uloženými v grid adresáři – vizualizace je ovšem oproti grid adresáři svisle překlopena – číslo v levém horním rohu v grid adresáři se nachází v levém dolním rohu vizualizace. S vizualizací lze libovolně manipulovat – posunovat, přibližovat a oddalovat (viz nápověda grid souboru).



Obrázek 74 – GUI – vizualizace uložených dat v grid souboru

Zdroj: vlastní

Příloha B – UML diagram vizualizace B-stromu



Příloha C – Obsah přiloženého CD

Obsah přiloženého CD má následující strukturu:

- Projekty – obsahuje projekty vizualizací v NetBeans IDE 7.4,
    - BinomialHeap – projekt vizualizace binomické haldy,
    - BTree – projekt vizualizace B-stromu,
    - DataStructuresVisualizations – společný projekt všech vizualizací,
    - GridFile – projekt vizualizace grid souboru,
    - HuffmanTree – projekt vizualizace Huffmanova stromu.
- Vizualizace – obsahuje vytvořené vizualizace (soubor JAR, JNLP a testovací webovou stránku),
    - BinomialHeap – vizualizace binomické haldy,
    - BTree – vizualizace B-stromu,
    - GridFile – vizualizace grid souboru,
    - HuffmanTree – vizualizace Huffmanova stromu.
- SaraM_VizualizaceEvoluce_AK_2014.pdf – tento dokument.