UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Vizualizace evoluce algoritmů pracujících nad vybranými implementacemi tabulky

Bc. Viktor Krejčíř

Diplomová práce
2015

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: **2014/2015**

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Viktor Krejčíř**
Osobní číslo:	**I12478**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vizualizace evoluce algoritmů pracujících nad vybranými implementacemi tabulky**
Zadávající katedra:	**Katedra softwarových technologií**

Z á s a d y   p r o   v y p r a c o v á n í :

V úvodní části práce je nutné provést přehled problematiky implementací abstraktního datového typu tabulka využívajících vybrané hierarchické a lineární implementující typy. Primárním cílem diplomové práce je realizace vizualizací evolucí vybraných algoritmů nad následujícími datovými strukturami: binární vyhledávací strom (binary search tree), AVL strom (AVL tree), 2-3 strom (2-3 tree), quad strom (quad tree) a hierarchie seznamů (skip list). Zmíněné vizualizace budou realizovány v rámci webové aplikace.

Rozsah grafických prací:

Rozsah pracovní zprávy:

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

Seznam odborné literatury:

1. SAMET, H. Foundations of Multidimensional and Metric Data Structures, San Francisco (CA), Morgan Kaufmann Publishers, 2006.
2. CORMEN, H. A KOL. Introduction to algorithms. Boston, MIT Press, 2001.
3. LEWIS, H. R., DENENBERG, L. Data structures and their algorithms. Berkley, Adison-Wesley, 1997.
4. 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 2014**
Termín odevzdání diplomové práce: **15. května 2015**

prof. Ing. Simeon Karamazov, Dr.
děkan

L.S.

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

V Pardubicích dne 15. listopadu 2014

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 27. 8. 2015 Bc.Viktor Krejčíř

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

Tato diplomová práce se zabývá tématem vizualizace datových struktur a jejich příslušných algoritmů. Konkrétně se jedná o vizualizace binárního vyhledávacího stromu, AVL stromu, 2-3 stromu, quad stromu a hierarchie seznamů (skip-list). V úvodní části práce je uveden přehled existujících řešení vizualizací. Následující teoretická část poskytuje pohled na samotné datové struktury, jejich možnosti a využití. Závěrečná praktická část popisuje podrobněji fungování jednotlivých algoritmů a způsoby jejich implementace. Výsledkem této práce jsou mimo jiné samotné vizualizace pěti datových struktur spouštěné ve webovém prohlížeči.

Klíčová slova

datové struktury, vizualizace, JavaFX, binární vyhledávací strom, AVL strom, 2-3 strom, quad-strom, hierarchie seznamů

Title

Visualization of algorithm evolution used in selected table implementations

Annotation

This thesis deals with a theme of data structure visualization as well as their corresponding algorithms. Specifically – the visualizations of binary search tree, AVL tree, 2-3 tree, quad tree and skip list. In the introductory part of the thesis, there is a review of an existing visualization solutions. Following theoretical part spots on data structures, their possibilities and utilization. The last, practical part, describes the processes of individual algorithms in detail with a way of their implementation. The result of this work are, inter alia, the actual visualizations of five data structures runnable in a web browser environment.

Keywords

data structures, visualization, JavaFX, binary search tree, AVL tree, 2-3 tree, quad-tree, skip list

Obsah

Seznam zkratek **8**

Seznam obrázků **9**

Úvod **10**

1 Přehled problematiky **11**

1.1 Webový portál Interactive Data Structure Visualizations 11

1.2 Webová aplikace Data Structures Visualizations 12

1.3 Webová aplikace VisualGo 14

1.4 Další zdroje 15

1.5 Shrnutí 17

2 ADT Tabulka **18**

2.1 Operace 19

2.2 Binární vyhledávací strom 19

2.3 AVL strom 21

2.4 2-3 strom 22

2.5 Quad strom 24

2.6 Hierarchie seznamů 26

3 Implementační technologie **27**

3.1 Výběr technologie 27

3.1.1 Požadavky 27

3.1.2 Vybrané analyzované technologie 28

3.1.3 JavaFX 29

3.2 Implementace vizualizací 30

3.2.1 Princip návrhu vizualizací 31

4 Implementace komponent společných pro všechny projekty **32**

4.1 Komponenta Animation Control 32

4.2 Systém pro obsluhu událostí 34

4.3 Obecný layout manager pro binární strom 34

5 Implementace Binárního vyhledávacího stromu **36**

5.1 Operace Najdi 37

5.2 Operace Vlož 38

5.3 Operace Odeber 38

6 Implementace AVL stromu ... 40
6.1 Operace Najdi ... 40
6.2 Operace Vlož ... 40
6.3 Operace Odeber ... 41
6.4 Realizace vyvažování AVL stromu ... 43
7 Implementace 2-3 stromu ... 45
7.1 Operace Najdi ... 46
7.2 Operace Vlož ... 46
7.3 Operace Odeber ... 47
7.4 Animace změny struktury ... 48
8 Implementace Quad stromu ... 49
8.1 Operace Najdi ... 50
8.2 Operace Vlož ... 51
8.3 Operace Odeber ... 52
9 Implementace hierarchie seznamů (skip listu) ... 53
9.1 Operace Najdi ... 53
9.2 Operace Vlož ... 54
9.3 Operace Odeber ... 54
Závěr ... 55
Literatura ... 57
Příloha A – UML diagram tříd Hierarchie seznamů ... 58
Příloha B – Obsah přiloženého CD ... 59

Seznam zkratek

ADT	Abstraktní datový typ
BVS	Binární vyhledávací strom
CSS	Cascade Style Sheet
ERD	Entity-relationship diagram
GUI	Graphic User Interface
GWT	Google Web Tools
HTML	Hypertext Markup Language
JDK	Java Development Kit
JRE	Java Runtime Environment
OOP	Object Oriented Programming
RIA	Rich Internet Application
UML	Unified Modeling Language

Seznam obrázků

Obrázek 1: Interactive Data Structure Visualization - okno Java appletu.......................... 11
Obrázek 2: Data Structures Visualizations - klasická Java aplikace 13
Obrázek 3: Data Structure Visualizations - webová verze 14
Obrázek 4: VisualGo - prostředí vizualizace BVS 15
Obrázek 5: Záznam přednášky z kurzu Design and Analysis of Algorithms..................... 16
Obrázek 6: Prostředí vizualizace v OpenDSA 17
Obrázek 7: Binární vyhledávací strom 19
Obrázek 8: Zdegenerovaný BVS do podoby lineárního seznamu...................................... 20
Obrázek 9: Princip RL rotace 22
Obrázek 10: 2-3 strom 23
Obrázek 11: Regionové zobrazení Quad-stromu... 25
Obrázek 12: Stromové zobrazení Quad stromu.. 25
Obrázek 13: Skip-List... 26
Obrázek 14: UML diagram tříd komponenty Animation Control..................................... 33
Obrázek 15: UML diagram tříd BVS 36
Obrázek 16: Ilustrace paměťové reprezentace BVS.. 37
Obrázek 17: Vložení prvku s klíčem 30 do BVS 38
Obrázek 18: Ilustrace odebrání vnitřního uzlu s klíčem 20 39
Obrázek 19: Ilustrace evoluce algoritmu při operaci Vlož.. 40
Obrázek 20: Odstranění prvku z AVL stromu - případ I... 41
Obrázek 21: Odstranění prvku z AVL stromu - případ II 41
Obrázek 22: Odstranění prvku z AVL stromu - případ III... 42
Obrázek 23: Odstranění prvku z AVL stromu - případ IV... 42
Obrázek 24: Odstranění prvku z AVL stromu - případ V 42
Obrázek 25: Zobrazení pravé rotace.. 43
Obrázek 26: Zobrazení levé rotace... 44
Obrázek 27: UML diagram tříd 2-3 stromu 45
Obrázek 28: Rozdělení dvou-klíčového uzlu při vkládání klíče 46
Obrázek 29: Odstraňování klíče v případě existence dvou-klíčového sourozence 47
Obrázek 30: Odstraňování klíče v případě neexistence dvou-klíčového sourozence 48
Obrázek 31: UML diagram tříd Quad stromu 50
Obrázek 32: Změna stavu struktury po vložení prvku [80,80].. 51
Obrázek 33: Odstranění hodnoty [80,80] a následná redukce struktury 52
Obrázek 34: Paměťová reprezentace Skip listu.. 53

Úvod

Společně s dalšími pracemi, je i tato součástí projektu tvorby databáze vizualizací vybraných datových struktur. Ty budou sloužit nejen studentům FEI Univerzity Pardubice k lepšímu pochopení problematiky méně triviálních datových struktur. V oblasti programování jakékoliv aplikace se totiž bez jejich znalosti neobejdeme. Je velice důležité podporovat povědomí o pokročilejších strukturách, neboť často i přes složitější implementaci disponují lepšími vlastnostmi než ty, které jsou v praxi častěji používané.

Základní strukturou v této práci je Binární vyhledávací strom (BVS), jeho implementace je poměrně jednoduchá, stejně jako princip jeho činnosti. Dalo by se říci, že je ovšem jakýmsi „odrazovým můstkem" k dokonalejším strukturám. První z nich je AVL strom, který odstraňuje základní nedokonalost prostého BVS a představuje tak pokročilejší stromovou vyhledávací strukturu. 2-3 strom je z hlediska implementace pravděpodobně nejnáročnější strukturou, která je v této práci obsažena, nicméně se dokáže programátorovi, který ji využije, odměnit například dokonalým vyvážením stromu. Quad-strom je svým využitím poměrně specifický, ovšem jak je dále uvedeno, i on bývá poměrně často využíván v nejrůznějších aplikacích zabývajících se dvourozměrnými daty. Poslední z popisovaných a implementovaných datových struktur je poměrně méně známá Hierarchie seznamů neboli Skip-list. Ta nabízí zajímavou alternativu k tradičním stromovým vyhledávajícím strukturám.

Vzhledem k technologii, která byla zvolena k implementaci bude možné spouštět jednotlivé vizualizace přímo v okně prohlížeče. V případě, že uživatel bude požadovat spouštění aplikací bez internetového připojení, je k dispozici i jejich stažení na disk.

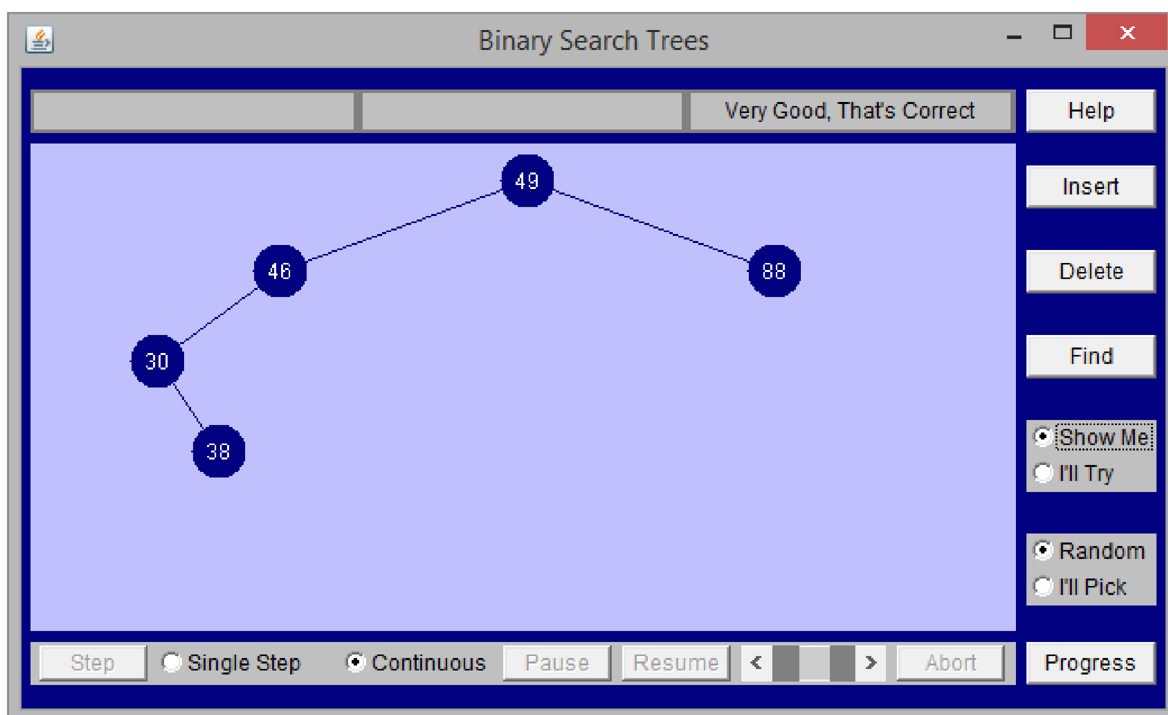
V první kapitole je uveden přehled stávajích řešení ve smyslu vizualizace datových struktur doplněný o komentář a hodnocení autorem této práce. V následující části práce je přednesen teoretický popis datových struktur, kterými se tato práce zabývá. Další kapitoly jsou pak věnovány samotné implementaci – od výběru technologie až k realizaci samotných implementací jednotlivých projektů.

1 Přehled problematiky

Existuje celá řada ucelených databází, které nám nabízí nejrůznější výběr vizualizací datových struktur. Ty největší jsou zpravidla spravovány univerzitami, nicméně na internetu najdeme i množství menších projektů. Většina z těch významnějších je realizována jako webová aplikace za použití nejrůznějších technologií – ať už se jedná o aplikace v Javascriptu, Flashi či Javě – jejich výhoda je jasná. Přístupnost odkudkoli a nulové nároky na uživatele z hlediska potřeby instalace a konfigurace.

1.1 Webový portál Interactive Data Structure Visualizations

Tato webová prezentace Dr. Duane J. Jarca z Univerzity George Washingtona[1] obsahuje hned několik vizualizací nejen datových struktur odvozených z binárního stromu či grafu, ale i animaci dvou řadicích algoritmů. Každá ze struktur či algoritmů je okomentována několika odstavci vysvětlující princip dané problematiky. Uživatel si zde také může zobrazit zdrojový kód řešené problematiky v ne příliš rozšířeném programovacím jazyce Ada[2].



Obrázek 1: Interactive Data Structure Visualization - okno Java appletu.

Zdroj: vlastní

Samotnou implementací je využití Java Appletu startujících ne jako vložený (embedded) objekt, ale jako samostatné okno, které se uživateli objeví po kliknutí na tlačítko na stránce. Toto řešení by mohlo některé uživatele zmást (není příliš populární využívat

[1] http://www.student.seas.gwu.edu/~idsv/idsv.html
[2] http://langpop.com/

vyskakujících oken u webové prezentace), navíc to s sebou nese i funkční nevýhodu v podobě nemožnosti opustit danou podstránku, ze které byla vizualizace spuštěna. Odpadá tím tedy základní myšlenka aplikace v samostatném okně - nezávislost.

Aplikace jako taková umožňuje jak plynulou, tak krokovou animaci vývoje algoritmu pro všechny základní operace. Obsahuje též generátor dat, avšak uživatel samozřejmě může zadávat i vlastní hodnoty dat. Zajímavou edukativní funkci, která zajistila tomuto souboru aplikací místo v tomto přehledu, je možnost nechat uživatele vyzkoušet, jaký bude další krok v průběhu algoritmu. Pokud uživatel zvolí špatnou volbu (ukáže myší na špatný prvek nebo vybere špatné místo pro vložení prvku) aplikace jej upozorní a dá mu k dispozici několik dalších pokusů. Pokud ani poté neuspěje, je mu znázorněno, jaký je korektní postup.

Původní účel těchto aplikací byl tak pravděpodobně nejen studentům vysvětlit princip fungování algoritmů ve spojitosti (nejen) s datovými strukturami, ale také je z těchto algoritmů vyzkoušet, což dokazuje koneckonců i okno „Progress" uchovávající výsledky interakce uživatele s aplikací.

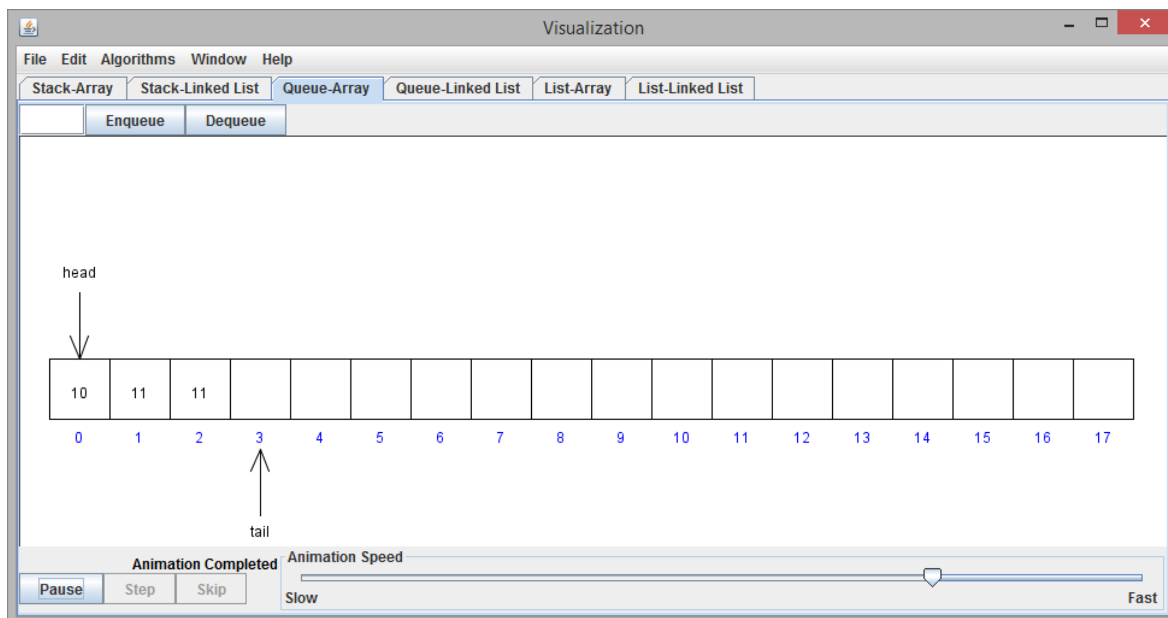
Shrnutí:

+ Interakce s uživatelem	- Uživatelské rozhraní
+ Vysvětlení algoritmu komentářem i kódem	- Malý výběr struktur

1.2 Webová aplikace Data Structures Visualizations

I tato aplikace je realizována v rámci univerzity, tentokrát se jedná o University of San Francisco, jejíž aplikaci[3] spravuje David Galles. Na první pohled zaujme množství vizualizací, které jsou na tomto webu dostupné – od elementárních struktur jakou jsou zásobníky či fronty, přes nejrůznější druhy stromových struktur až po grafové či geometrické algoritmy. K této nabídce je navíc k dispozici ke stažení ucelený soubor vizualizací v podobě samostatné desktopové Java aplikace.

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

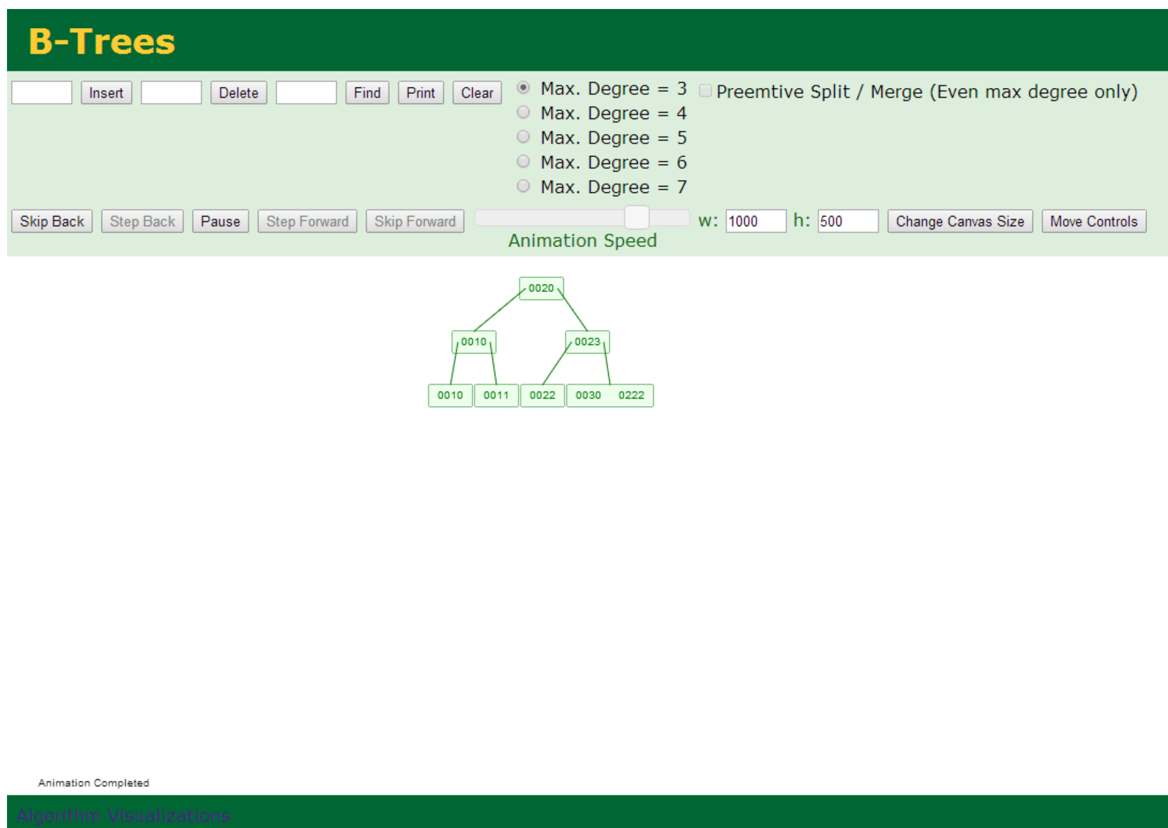


Obrázek 2: Data Structures Visualizations - klasická Java aplikace

Zdroj: vlastní

Ta zahrnuje pouze část z těch algoritmů, které jsou na webu dostupné (avšak i některé navíc). Je to zejména proto, že tato aplikace byla vytvořena dříve a v současné době již není autorem aktivně vyvíjena a doplňována o nové animace. Její bezesporu největší nevýhodou je fakt, že nejen že neobsahuje předpřipravenou sadu dat, ze které by bylo možno danou strukturu generovat, ale ani nedisponuje generátorem náhodných hodnot, které by usnadnilo uživateli vybudování komplexnější struktury.

Stejnou nevýhodou bohužel disponují i webové verze animací – ty jsou realizovány pouze za pomocí JavaScriptu a k jejich spuštění tak není potřeba žádný zvláštní plugin. Vizualizace disponují jak plynulou animací, tak postupným krokováním (dokonce s možností krokovat i zpětně). Další výtkou by také mohly být chybějící komentáře jak u jednotlivých kroků animace, tak u dat. struktur jako celku.



Obrázek 3: Data Structure Visualizations - webová verze

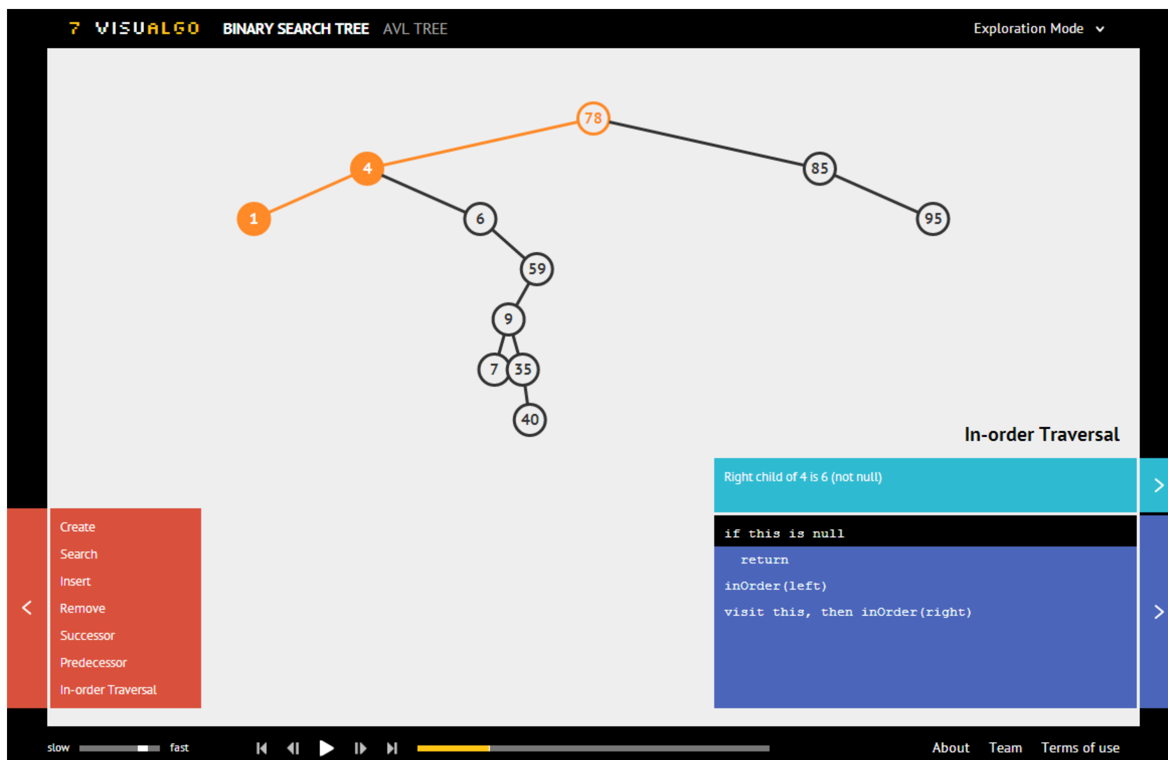
Zdroj: vlastní

Pro vývojáře, kteří se zabývají vizualizací algoritmů za pomocí technologií Javascript a HTML5 nabízí autor také kompletní zdrojové kódy všech publikovaných vizualizací.[4]

1.3 Webová aplikace VisualGo

Tato aplikace[5] ze singapurské univerzity vytvořená týmem vedeným Dr. Stevenem Halimem stejně jako předchozí nabízí animace realizované pomocí Javascriptu, proto ani tato nevyžaduje instalaci dodatečného softwaru či pluginů do počitače. Jednotlivých datových struktur k vizualizaci tu sice není příliš mnoho (autoři se pravděpodobně především soustředili na problematiku znázornění algoritmů obecně – napříč obory), nicméně jsou graficky opravdu velmi pěkně zpracovány. Animace jsou plynulé s možností krokování (i zpětně), každý krok je navíc vysvětlen jak v podobě stručného popisu, tak v podobě pseudokódu (což je pro technicky zdatnější uživatele či studenty poměrně přínosná funkce). Nechybí ani možnost nechat si datovou strukturu vygenerovat z náhodných dat. Pro začátečníky je zde obsažen i tzv. „tutorial mode“ vysvětlující princip a ovládání aplikace.

[4] http://www.cs.usfca.edu/~galles/visualization/source.html
[5] http://www.comp.nus.edu.sg/~stevenha/visualization/index.html



Obrázek 4: VisualGo - prostředí vizualizace BVS

Zdroj: vlastní

Shrnutí:

+ grafické zpracování
+ tutorial a komentáře

- malý výběr z oboru dat. struktur

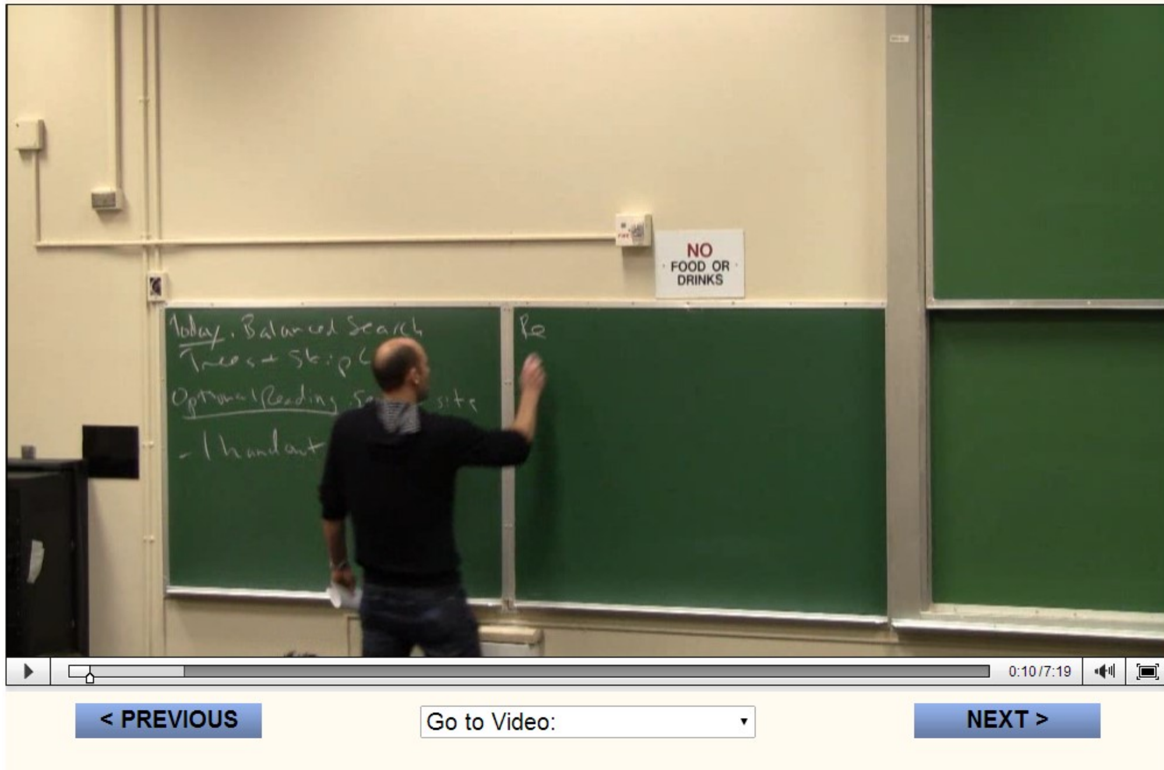
1.4 Další zdroje

V této podkapitole budou zmíněny některé další projekty, které mohou studentům pomoci při studiu problematiky datových struktur a s nimi souvisejících algoritmů.

Velice užitečným pomocníkem při studiu (zejména pro začátečníky) může být webová stránka CSAnimated[6] obsahující přednášky s mluveným komentářem zahrnující vysvětlení jak datových struktur, tak i dalších problematik (např. řadicích algoritmů).

Podobným studijním materiálem jsou i zveřejněné přednášky z kurzu *Design and Analysis of Algorithms* ze Stanfordské univerzity. Tento kurz z oboru počítačových věd (computer science) je spíše zaměřen na studium algoritmů jako takových (zahrnující nejrůznější grafové či řadicí algoritmy), nicméně může se ukázat užitečným i pro studenty věnující se vyloženě datovým strukturám. Nevýhodou může být požadavek na pokročilejší znalost jazyků, avšak ten je vyvážen faktem, že se jedná o opravdové přednášky odborníků.

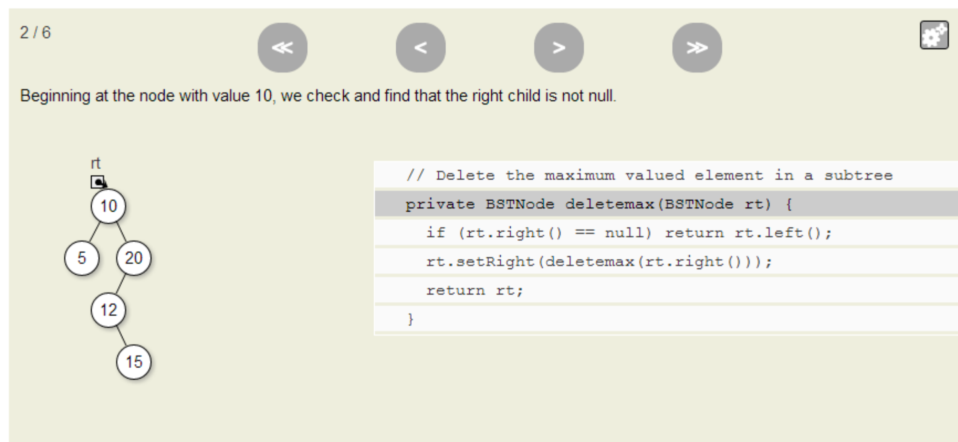
[6] http://www.csanimated.com/



Obrázek 5: Záznam přednášky z kurzu Design and Analysis of Algorithms

Zdroj: vlastní

Skutečně uceleným zdrojem je také projekt OpenDSA. Sami tvůrci jej popisují jako „interaktivní knihu“, která obsahuje jak komplexní popis datových struktur, tak i jejich interaktivní vizualizace. Jednotlivé kroky algoritmů jsou patřičně okomentovány jak v klasické formě, tak i v podobě pseudokódu. Potěší také zveřejněné zdrojové kódy daných datových struktur. Aplikace také nabízí uživateli v průběhu výuky několik testů umožňující vyzkoušet si nabyté znalosti dané problematiky.



Obrázek 6: Prostředí vizualizace v OpenDSA

Zdroj: vlastní

1.5 Shrnutí

V předchozích kapitolách byl představen a stručně zhodnocen výběr existujících řešení dané problematiky vizualizace datových struktur. Zajímavým faktem je, že i ty nejrozsáhlejší databáze nepokrývaly potřebu vysvětlení všech běžně používaných struktur. Některé implementace byly obsaženy vždy, ať už se jednalo o základní frontu/zásobník nebo binární vyhledávací strom – některé však v databázi zahrnuty nebyly (Quad strom nebo skip-list). Nicméně vedle těchto obsáhlejších, univerzitami spravovaných, projektů samozřejmě existují i ty menší, často jsou to práce jednotlivce vysvětlující konkrétní algoritmus či datovou strukturu.

Během tohoto průzkumu byly odhaleny i některé nedostatky v uvedených projektech. Jednalo se především o nepřehledná uživatelská rozhraní či nemožnost vygenerovat data či vytvořit strukturu z předpřipravené sady dat. Tento problém se může zdát malicherným, nicméně u komplexnějších struktur, kdy algoritmy často reorganizují data až při jejich větším objemu, toto může představovat pro uživatele problém a zdlouhavou či nepohodlnou obsluhu celé aplikace.

2 ADT Tabulka

Je heterogenní množina s lineárním uspořádáním sestávající se ze dvou elementárních typů – klíč a hodnota. Implementace tohoto ADT jsou v praxi velice časté. Tabulku můžeme rozlišovat buď podle dynamiky (pevně stanovená velikost tabulky nebo proměnný počet prvků) nebo podle způsobu následné implementace:

- Tabulka na poli
- Tabulka na seznamu
- Tabulka na vyhledávacím stromu
- Tabulka na implicitní kosočtvercové síti
- Tabulka s rozptýlenými položkami (hashovací)

Tabulky na poli využívají principu přímé adresace – každá položka tabulky má své místo, které je jednoznačně určené jejím klíčem. Svým využitím může připomínat elementární datovou strukturu pole – místo celočíselných indexů můžeme však použít klíče libovolného typu. Klíč také může být multidimenzionální – například sestávající se ze složek x a y určující jednoznačnou polohu prvku ve dvourozměrném prostoru.

Mezi výhody tohoto řešení patří rychlý přístup k odpovídající hodnotě (za předpokladu, že pole klíčů je uspořádané) a poměrně snadná implementace. Naopak mezi úskalí implementací založených na tomto způsobu patří nutnost využít celý rozsah klíče a k možné realizaci „řídkého pole“. Pokud budeme např. potřebovat evidovat databázi telefonní sítě, kdy klíčem bude unikátní telefonní číslo a hodnotou jméno majitele linky, budeme muset vzít v potaz všechny možné hodnoty tel. čísla, což ve výsledku vede k potřebě alokovat prostor pro obrovské množství položek. Za předpokladu, že by v dané síti navíc nebylo mnoho aktivních zákazníků, vznikal by zbytečně veliký počet neobsazených hodnot v celé tabulce.

Toto můžeme eliminovat nahrazením klasického pole spojovým seznamem. Využijeme tak jenom ten rozsah hodnot klíčů, který potřebujeme. Daní za toto rozhodnutí je nutnost veškeré operace nad touto implementací provádět sekvenčně, což u rozsáhlejších tabulek může být velice neefektivní.

Jedním z optimálních řešení, jak docílit lepších výsledků pro operaci s tabulkou je možnost využití některého z vyhledávacích stromů (BVS, AVL, …). Těm budou věnovány následující kapitoly v této práci.

2.1 Operace

ADT tabulka definuje na množině klíčů následující operace:

- Vytvoř
- JePrázdná
- Vlož
- Odeber
- Najdi

Operace „Vytvoř“ vytvoří prázdnou tabulku a zpřístupní další operace pro práci s ní.

„Je prázdná“ umožňuje uživateli či programátorovi zjistit, zdali tabulka obsahuje nějaké prvky.

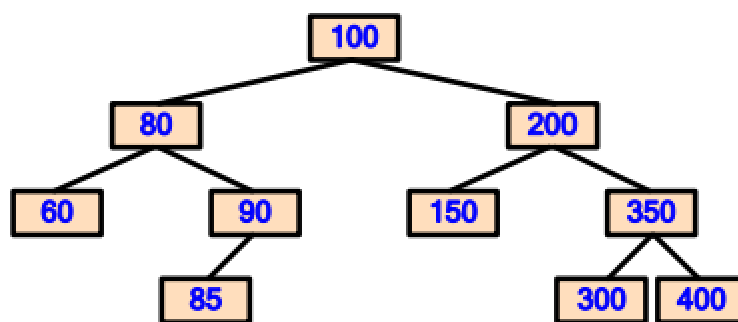
Operace „Vlož“ provede vložení prvku P s klíčem K do vytvořené tabulky.

Požadovaný prvek můžeme dle klíče K nalézt pomocí operace „Najdi“.

Poslední z operací – „Odeber“ odstraní daný prvek pomocí jeho klíče K.

2.2 Binární vyhledávací strom

Tento vyhledávací strom je „označení pro binární strom, pro jehož každý vrchol x charakterizovaný klíčem K_x platí: (i) je-li y vrchol z levého podstromu vrcholu x, pak $K_y < K_x$ a (ii) je-li y vrchol z pravého podstromu vrcholu x, pak $K_y > K_x$ “.



Obrázek 7: Binární vyhledávací strom

Zdroj: vlastní

Princip vyhledávání v BVS je prostý: postupně procházíme strom od jeho kořene a porovnáváním vyhledávané hodnoty s hodnotou klíče v daném prvku stromu volíme přechod do levého (vyhledávaná hodnota je menší než klíč prvku) nebo pravého (vyhledávaná hodnota je větší než klíč prvku) podstromu. Tímto způsobem pokračujeme až do nalezení patřičného klíče, kdy jeho pomocí zpřístupníme prvek, který reprezentuje.

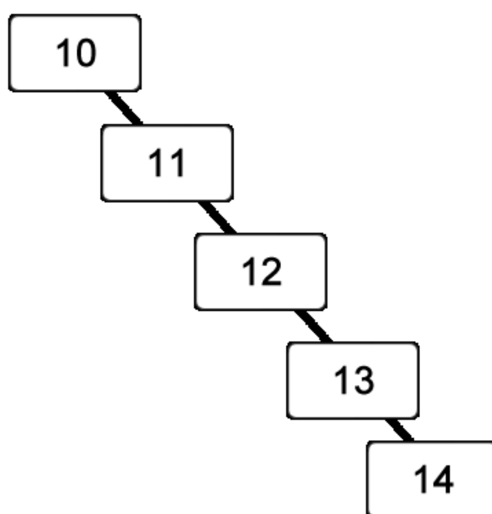
Vkládání probíhá vyhledáním vhodného místa pro vložení stejným způsobem, jaký byl popsán výše. Následně je prvek vložen jako list do tohoto stromu.

Odebrání prvku zajistíme následujícím způsobem. Pokud je odebíraný prvek listem, je prostě odebrán z celé struktury a neovlivní tak ostatní prvky ve stromu. Za předpokladu, že jím není, je prvek nahrazen nejpravějším prvkem z levého podstromu nebo nejlevějším prvkem z pravého podstromu (samozřejmě tehdy, pokud existuje).

Operace „JePrázdný“ je u všech stromových struktur stejná – testuje se přítomnost kořenu (jakožto vstupního bodu). Za předpokladu, že není nalezen, je struktura prohlášena za prázdnou.

Výhodou BVS je jednoduchá implementace v podobě buď dynamického stromu, nebo stromu na poli a současně jednoduchý a účinný mechanismus vyhledávání prvku. Za připomenutí také stojí, že složitost operací Vlož, Odeber a Najdi je $O(\log_2 n)$. Mohou však nastat i případy, které efektivitu BVS nepříznivě ovlivní.

Vkládáním určitým prvků můžeme dosáhnout stavu, kdy jeden z podstromů má významně větší hloubku než jiné – dojde k nevyváženosti stromu. Tato situace může v nejhorším případě zdegenerovat BVS do podoby lineárního seznamu. Následující obrázek tento případ ilustruje – v tomto případě byly postupně vkládány prvky 10, 11, 12, 13 a 14 (v tomto pořadí). Protože každý z prvků byl vložen do nejpravějšího podstromu, došlo k popisované situaci. Složitost operací v této formě stromu může dosáhnout až O(n).



Obrázek 8: Zdegenerovaný BVS do podoby lineárního seznamu

Zdroj: vlastní

2.3 AVL strom

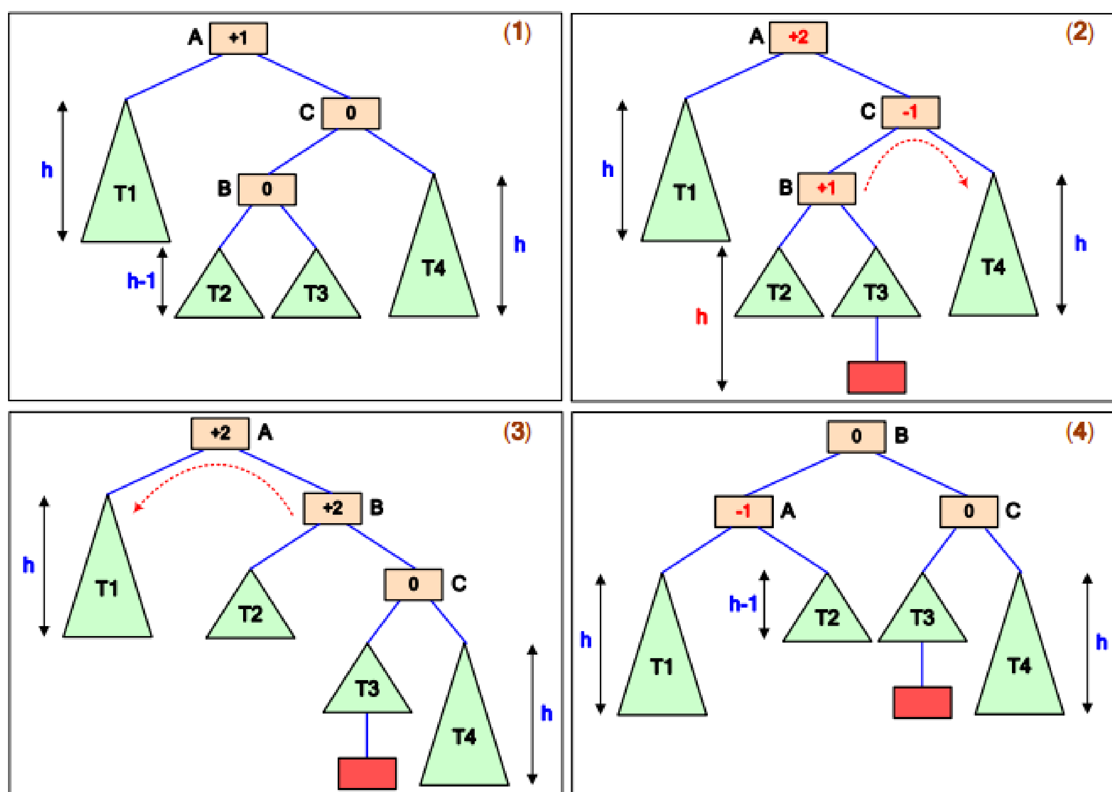
Poprvé byla tato struktura popsána pány G. M.Adelski-Velským a Jamesem M. Landisem v roce 1962, z počátečních písmen jejich příjmení tato struktura dostala i svůj název. Jeho základem je dříve uvedený BVS, nicméně tento strom obsahuje samovyvažovací mechanismus, který zabraňuje degradaci stromu. Pro přehlednost shrňme jeho základní vlastnosti:

- Jedná se o binární strom (vrchol obsahuje maximálně dva potomky)
- V levém podstromu jsou pouze hodnoty s menší hodnotou klíče, než je jeho kořen
- V pravém podstromu jsou pouze hodnoty s větší hodnotou klíče, než je jeho kořen
- Maximální výška pravého a levého podstromu se liší nejvýše o 1

První tři vlastnosti jsou společné s BVS, ovšem právě čtvrtá vlastnost – podmínka vyvážení stromu zabezpečuje, že nedojde k degradaci. AVL strom využívá stejný způsob průchodu stromem při operacích k vyhledávání, vkládání či odebírání.

Každý prvek stromu obsahuje hodnotu informující o jeho vyváženosti. Tato hodnota se spočítá odečtením výšky levého podstromu od výšky podstromu pravého. Přípustné hodnoty splňující podmínku vyváženosti stromu jsou $\{-1,0,1\}$. Při vkládání nebo odebírání (tedy operacích, které ovlivňují počet prvků stromu) jsou vždy hodnoty vyváženosti aktualizovány směrem od přidaného/odebraného prvku ke kořenu stromu. V případě, že některá z hodnot bude vyšší než 1, případně nižší než -1, dojde k úpravě organizace stromu prostřednictvím tzv. rotací.

Jejich principem je cyklická záměna ukazatelů v levém nebo pravém směru, případně jejich kombinace (mluvíme tak o pravo-levé, levo-pravé, dvojité pravé či dvojité levé rotaci). Po každé této operaci provedeme opětnou aktualizaci vyváženosti v každém ovlivněném vrcholu a v případě další potřeby celý postup opakujeme. Příklad dvojité (pravo-levé) rotace uvádí následující obrázek.



Obrázek 9: Princip RL rotace

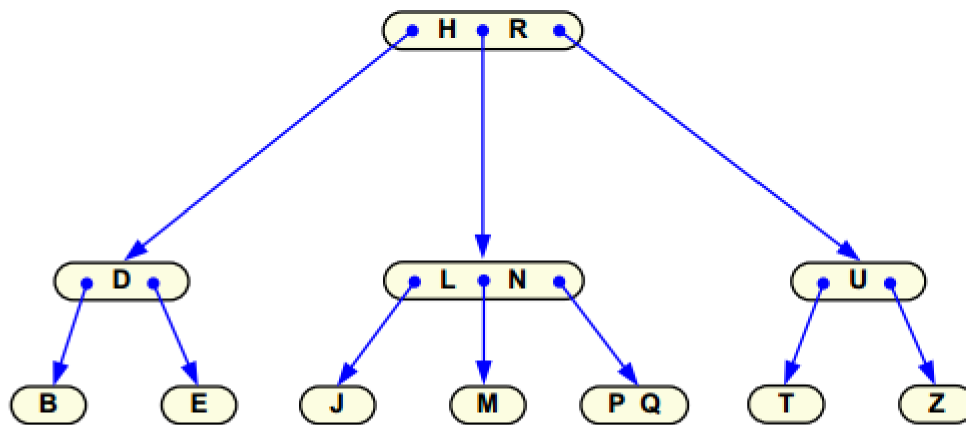
Zdroj: vlastní

2.4 2-3 strom

Je dokonale vyvážený strom (tzn. všechny jeho listy leží ve stejné hloubce) na rozdíl od předchozích struktur. Za jeho autora by se dal považovat John Hopcroft, který jej poprvé navrhnul v roce 1970 jako vylepšení vyvážených binárních stromů. 2-3 strom byl následně zobecněn do podoby B-stromu Rudolfem Bayerem a Edem McCreightem.

Základní vlastnosti 2-3 stromu jsou:

- Všechny cesty od kořene k listům jsou stejně dlouhé
- V případě B+stromové implementace jsou data zapsána v úrovni listů, vnitřní uzly tak slouží pouze k traverzování stromu
- Listy jsou řazeny podle klíče zleva (od minimu k maximu)
- Každý uzel má buď jednu nebo dvě klíčové položky. V případě jediného klíče má dva potomky, v případě dvou klíčů právě tři potomky (samozřejmě tehdy, pokud se jedná o vnitřní uzel).
- Je vlastně konkrétní implementací B-stromu, respektive B+-stromu s parametry 2 a 3.



Obrázek 10: 2-3 strom

Zdroj: vlastní

Operace „Najdi“ funguje podobně jako u předchozích dvou stromů. V případě 2-3 stromu je jediným rozdílem fakt, že při procházení můžeme porovnávat hledanou hodnotu klíče K se dvěma klíči (K_1, K_2). V tomto případě mohou nastat následující případy:

- $K < K_1$ – v tomto případě pokračujeme levým podstromem klíče K_1
- $K > K_1$ a současně $K < K_2$ – dále pokračujeme přechodem do levého podstromu klíče K_2 (prostřední podstrom z pohledu celého vrcholu)
- $K > K_2$ – přechod do pravého podstromu klíče K_2

Před vložením nového prvku nejdříve vyhledáme vhodného listu (L), kam budeme prvek vkládat. Za předpokladu, že je L 2-vrcholem, nově vkládaný prvek do něj zahrneme (vytvoříme tak z něj 3-vrchol) a algoritmus se ukončí. V případě, že je L vrcholem se dvěma klíčovými položkami, dojde k jeho rozštěpení na dva 2-vrcholy (L_1, L_2) a jejich klíče se aktualizují následujícím způsobem:

- Klíč vrcholu L_1 je zvolen jako nejmenší z trojice (K_1,K_2, K)
- Klíč vrcholu L_2 je zvolen jako největší z trojice (K_1,K_2, K)
- Zbylý klíč označíme jako $K_{medián}$ a přesuneme jej do otce původního vrcholu L (pokud existuje)

Za předpokladu, že by vložením klíče $K_{medián}$ do předka došlo k „přetečení“ počtu klíčů v prvku (tzn, modifikovaný prvek již je 3-vrcholem), stejný postup se opakuje i u tohoto vrcholu. Pokud tímto cyklem dosáhneme kořene celého stromu a dojde k jeho rozštěpení, je vytvořen nový kořenový 2-vrchol a výška celého stromu se tak zvýší o 1.

V rámci odebírání prvku nejdříve rozlišíme, zdali odebíráme list nebo vnitřní vrchol stromu. V prvním případě je list odstraněn, v druhém je nahrazen svým in-order následníkem (tzn prvkem s nejbližší vyšší hodnotou klíče).

U vrcholu, ze kterého byl odebrán prvek (*V*) je možnost, že už neobsahuje žádný prvek (dochází k podtečení počtu prvků ve vrcholu). Za této podmínky mohou nastat následující případy:

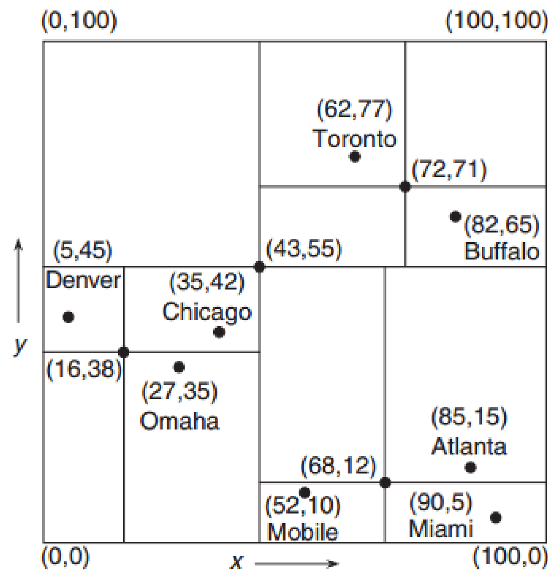
- *V* je kořenem, v tom případě je odstraněn a pokud disponoval synem, tento se stává kořenem místo něj. Pokud tomu tak není, je strom prázdný.
- *V* má bratra V‘, který se nachází po jeho levici či pravici, má dva prvky a společně s V otce s klíčem Ko, který je odděluje. Prvek s klíčem Ko je přesunut do v a do prvku Otec(V) je přesunut prvek z v‘, jehož klíč je přilehlý k vrcholu V. Představují-li V a V‘ vnitřní vrcholy, potom je rovněž přesunuta reference na příslušného syna z V‘ do V. Vrcholy V‘ a V se oba stanou 2-vrcholy a pro tento případ je algoritmus ukončen.
- *V* disponuje bratrem po své levici nebo pravice, který obsahuje pouze jeden klíč. Ko je opět klíčem otcovského vrcholu, který oba vrcholy V a V‘ separuje. Prvek, který je svázán s klíčem Ko a prvek z v‘ jsou sloučeny do nově vytvořeného 3-vrcholu nahrazující V a V‘. Následně je bod V aktualizován jako svůj otcovský prvek a všechny předchozí body se znovu opakují.

2.5 Quad strom

Tato struktura se využívá často pro uchovávání k souboru dat (často geografických), které jsou identifikovány dvěma nezávislými klíči v podobě koordinát ve 2D prostoru. Quad strom využívají např. i některé herní enginy k detekci kolizí objektů na scéně[7]. Hlavní vlastností této struktury je (jak už název napovídá) fakt, že každý z vnitřních vrcholů tohoto stromu má právě 4 potomky. Jestliže ne-listový vrchol stromu V reprezentuje čtvercovou oblast v dvojrozměrném prostoru, poté jeho potomci V_1, V_2, V_3 a V_4 (často pojmenovávané jako severovýchodní, jihovýchodní, jihozápadní a severozápadní) představují čtvercové podoblasti v podobě kvadrantů oblasti vrcholu V.

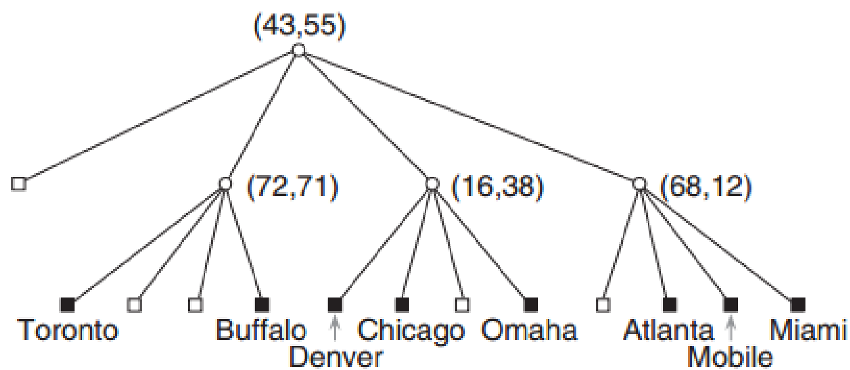
V případě vizualizace můžeme Quad strom znázornit jak plošným zobrazením, na kterém jsou zřetelně vidět oddělené jednotlivé regiony, tak hierarchickým (stromovým), jenž přináší pohled na hierarchickou reprezentaci jednotlivých oblastí.

[7] http://gamedevelopment.tutsplus.com/tutorials/quick-tip-use-quadtrees-to-detect-likely-collisions-in-2d-space--gamedev-374



Obrázek 11: Regionové zobrazení Quad-stromu

Zdroj: Hanan Samet (1)



Obrázek 12: Stromové zobrazení Quad stromu

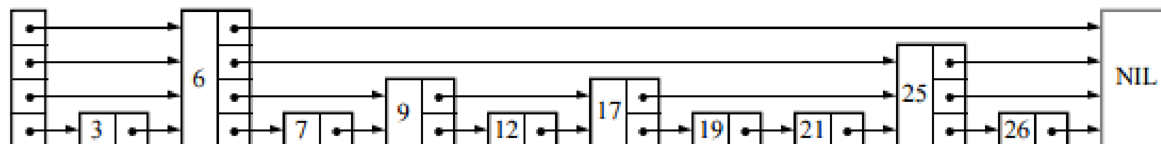
Zdroj: Hanan Samet (2)

Při vkládání nového prvku do stromu je nutné zjistit, jestli je příslušný region, do nějž vkládáme prázdný nebo ne. Pokud ano, je prvek do regionu vložen – v opačném případě je region rozdělen na čtyři potomky a prvek je vložen do jednoho z nich. Identifikace, do kterého potomka má být prvek vložen je provedena pomocí dvourozměrného klíče (2).

Odebírání probíhá pouze na úrovni listových vrcholů – pouze tato obsahují patřičná data. Za předpokladu, že má odebíraný prvek více než jednoho bratra, je odebrán a struktura se jiným způsobem nemění. Pokud má právě jednoho bratra, je tento vyměněn se svým otcem (posune se v hierarchii o jednu úroveň výše).

2.6 Hierarchie seznamů

Další strukturou je Hierarchie seznamů neboli Skip-list. Patří mezi ty mladší – poprvé byla vytvořena Walterem Pughem v roce 1989. Sestává se z různě řídkých spojových seznamů, seřazených hierarchicky podle vrstev – horní vrstva je nejřidší a obsahuje pouze některé prvky, nejspodnější je naopak nejhustější a obsahuje všechny prvky struktury.



Obrázek 13: Skip-List

Zdroj:W.Pugh (1)

Vyhledávání funguje postupně od nejmenšího prvku nejvyšší vrstvy směrem dolů. Pohybujeme se horizontálně, dokud není klíč hledaného prvku větší nebo roven porovnávanému klíči. Pokud je porovnávaná hodnota struktury větší než hodnota hledaného klíče nebo dosáhneme konce seznamu, celý proces průchodu se opakuje na nižší vrstvě. Takto můžeme postupovat až na nejnižší vrstvu v případě, že je hledaný prvek obsažen pouze v ní.

Při vkládání využíváme stejný způsob procházení jako při vyhledávání prvku – tímto průchodem najdeme vhodné místo pro vložení prvku. Prvek je nejprve vložen do nejspodnější vrstvy, následně je formou pravděpodobnosti rozhodnuto o vložení reference do vyššího seznamu. Často se volí poloviční nebo čtvrtinová pravděpodobnost na vložení reference do seznamu.

Odstranění probíhá jako u klasického spojového seznamu – prostým odpojením prvku z něj. Je potřeba však dbát na fakt, že v případě mazání musíme odstranit referenci na prvek ze všech úrovní struktury Skip-list.

.

3 Implementační technologie

V této kapitole bude uveden přehled existujících platforem a technologií umožňující implementaci daného tématu a požadavky na ně. Následně bude blíže představena vybraná technologie – JavaFX. Na závěr této kapitoly bude představena filozofie návrhu implementace a obecný koncept návrhu vizualizací.

3.1 Výběr technologie

Tato podkapitola se zaměřuje na konkrétní způsob výběru implementační platformy. Budou zde představeny vybrané analyzované technologie, popsány jejich specifika a následně zhodnocena míra využitelnosti v oblasti vizualizací datových struktur. Závěrem bude podrobněji popsána technologie JavaFX, která byla k samotné realizaci zvolena.

3.1.1 Požadavky

Zvolit správnou technologie je klíčový krok před každou tvorbou libovolné aplikace. Hlavním požadavkem je realizace vizualizací vybraných datových struktur prostřednictvím webové aplikace. Právě potřeba využít soubor technologií umožňujících tvorbu webové aplikace poněkud zužuje portfolio dostupných platforem – ty by měly poskytovat jak potřebné nástroje k vytvoření znovupoužitelných datových struktur, tak efektivně umožňovat jejich zobrazení a celkovou vizualizaci v tomto prostředí.

Dílčím požadavkem je také dostupnost aplikací na různých platformách. Jakkoli se tato problematika řeší spíše u desktopových aplikací spouštěných na různých operačních systémech, i v případě webového prostředí se často musíme mít na pozoru. Velká řada technologií totiž požaduje instalaci některého z pluginů, který už ze strany klientského systému nemusí být vždy podporován. Tato omezení mohou platit i na straně různých vykreslovacích jader webových prohlížečů v případě, že se programátor rozhodne využít klasickou kombinaci jazyků HTML + Javascript.

Neméně důležitým požadavkem je také předpokládaná podpora dané platformy. Je to dáno především cílem této práce – nabídnout budoucím studentům názorný výukový materiál ke studiu datových struktur a jejich algoritmů. Musíme tedy vzít v potaz i případné budoucí plány tvůrců platformy a garanci jejich podpory.

Krátké shrnutí výše popsaných požadavků:

- Běh aplikace ve webovém prostředí
- Potřebné nástroje pro tvorbu vizualizací a animace jako takové
- Vhodná platforma pro implementaci vlastních datových struktur
- Multiplatformnost
- Budoucnost technologie

3.1.2 Vybrané analyzované technologie

Na základě výše uvedených požadavků byly vybrány následující technologie, které byly podrobněji zhodnoceny.

- HTML + Javascript
- Dart
- Adobe Flash
- Apache Flex
- Microsoft Silverlight
- JavaFX

Kombinace značkovacího jazyka HTML společně s Javascriptem se nabízí hned jako první řešení při tvorbě webové aplikace. Mezi výhody patří jak snadná spustitelnost aplikace bez instalace jakéhokoliv doplňku, tak rozšířenost technologie a příslib do budoucna (v současné době je trendem nahrazování velikých RIA řešení právě kombinací těchto dvou jazyků). Za těmito výhodami se však skrývá i několik úskalí. Prvním z nich je různá podpora napříč internetovými prohlížeči – zvláště u poslední verze HTML a některých funkcí v Javascriptu (což se v druhém případě dá částečně eliminovat použitím některého z frameworků), druhým je nepříliš dobrá použitelnost Javascriptu pro tvorbu komplexnějších struktur. Je to především dáno tím faktem, že Javascript je z principu prototype-based jazyk.

Řešením problematiky Javascriptu by mohl být poměrně mladý open-source jazyk Dart, který má za cíl právě Javascript v moderních aplikacích nahradit. Nabízí klasický class-based přístup k objektově orientovanému programování a stále se rozšiřující komunitu vývojářů, kteří přispívají do ekosystému tohoto jazyka nejrůznějšími doplňky. K rychlému rozšíření Dartu by měl i přispět fakt, že tento jazyk lze v současné době kompilovat právě do Javascriptu, a je tedy možnost jej spustit v libovolném prohlížeči (ovšem za cenu snížení výkonu v porovnání s během v nativním virtuálním stroji). Tato technologie nebyla zvolena z toho důvodu, že při zadání práce a provedení této analýzy nebyla stále ještě ve své první stabilní verzi.

Další variantou bylo použití známé a rozšířené (leč poněkud kontroverzní) platformy Adobe Flash. Jeho první verze vyšla už v roce 1996, kdy jej od začínající firmy FutureWave převzala firma Macromedia, která byla v roce 2005 koupena společností Adobe. Flash byl používán velmi dlouhou dobu jako v podstatě jediná platforma umožňující tvorbu bohatých interaktivních aplikací, animací a přehrávání videa na webu. Jeho vlastník – Macromedia a později Adobe, jeho unikátnosti využívali tím způsobem, že tvorba animací této platformy byla umožněna pouze za pomocí proprietálního stejnojmenného softwaru. Ústup ze slávy by se dal přibližně datovat do období rozšíření chytrých telefonů, kdy jednotlivé mobilní platformy Flash buď nepodporovaly vůbec (iOS, Windows Mobile/Phone) nebo byla oficiální podpora ze strany vlastníka platformy od určité verze ukončena (Android). V reakci na tyto kroky se stalo trendem portovat stávající aplikace ve Flashi na jiné technologie (často právě na HTML+Javascript, jak je uvedeno

výše). Z pohledu současného vývojáře a uživatele se dá posuzovat, že tato platforma je pozvolna na ústupu a její budoucnost je při nejmenším velmi nejistá.

Aplikační framework Apache Flex pojí s Flashem společná historie – původně vlastněn firmou Macromedia, poté odkoupen společností Adobe, která jej však v roce 2011 uvolnila jako open-source a darovala nadaci Apache Software Foundation. O další vývoj tohoto Frameworku se tak nyní stará komunita sdružená kolem tohoto projektu. Flex využívá značkovacího jazyka MXML ke tvorbě uživatelského rozhraní a skriptovacího jazyka ActionScript k vytvoření samotné logiky aplikace. Jako běhové prostředí je z důvodu přenositelnosti na mobilní platformy využíván Adobe AIR, případně Adobe Flash (z důvodu zachování kompatibility).

Poslední zvažovanou technologií byl Microsoft Silverlight. Jeho první verze vyšla v září roku 2007 a měl se stát přímým konkurentem Flashe a Flexu. Využívá značkovacího jazyka XAML a jazyků zahrnutých v .NET frameworku (C#, F#, VB.NET). Jeho většímu rozšíření však brání podpora pouze systémů Microsoft Windows a Apple MacOs X. Existuje sice jeho open-source implementace v podobě projektu Moonlight, nicméně podpora některých funkcí Silverlightu chybí a původní aplikace v něm tak nemusí běžet zcela správně. Silverlight vzhledem ke své kvalitní podpoře DRM však našel uplatnění u různých zpoplatněných streamovacích služeb či virtuálních videopůjčoven (Topfun.cz). Otazníkem je také budoucnost této platformy, Microsoft přislíbil podporu poslední vydané verze do roku 2021, avšak žádné další verze by pravděpodobně neměly následovat.

3.1.3 JavaFX

JavaFX je sada nástrojů a knihoven především k tvorbě grafických uživatelských rozhraní v rámci platformy Java. První verze byla uvedena v roce 2008, která umožňovala tvorbu GUI pomocí vlastního jazyka JavaFX Script. Následovalo několik minoritních verzí přinášejících např. podporu stylování grafických komponent pomocí CSS, nové layout managery či komponenty sloužící k vykreslování grafů. Od verze 2 byla celá JavaFX kompletně přepsána do Javy (kvůli možnosti využívání všech existujících knihoven pro Javu a Java API) a JavaFX Script byl nahrazen značkovacím jazykem FXML (napodobující stejný princip jako XAML u Silverlightu nebo MXML u Flexu). JavaFX byla také začleněna do klasického JDK od verze 7u6.

Stejně jako Java je i JavaFX multiplatformní. Jedinou výjimkou jsou zařízení fungující na některé z variant ARM architektury, kde je jejich podpora teprve v přípravě. Významnou novinkou, kterou JavaFX přináší do Java platformy je možnost nasazení vytvořené aplikace jako klasickou desktopovou nebo vloženou (embedded) ve webovém prohlížeči. Tím se v podstatě smazávají rozdíly v tvorbě webové a desktopové aplikace a současně je tím umožněno tvořit skutečně plnohodnotné webové RIA projekty. Třetí možností deploymentu je pomocí tzv. web-startu (spouštění samostatné aplikace přímo z web. stránek) – jedná se však spíše o způsob distribuce, umožňující zpřístupnit nejaktuálnější verzi plnohodnotné aplikace běžící ve vlastním okně a navozující pocit, že uživatel pracuje s klasickou desktopovou verzí.

V této implementaci byla využita verze JavaFX dodávaná současně s JDK8. Soubor knihoven dříve označovaný jako FX SDK je tak nyní distribuován jako součást proprietárního frameworku Oracle JDK8. V případě open source knihoven OpenJDK je JavaFX vyvíjena jako projekt pod názvem OpenJFX[8], který se od původní verze od Oraclu liší vynecháním proprietárních technologií společnosti Oracle.

Vývoj projektů za použití technologie JavaFX má však i některé stinné stránky. O spuštění aplikace ve webovém prohlížeči se stará doplněk třetí strany označovaný jako NPAPI. Ten je distribuován současně s instalací Java Runtime Environment (JRE) knihoven. Tento doplněk byl donedávna podporován všemi běžnými prohlížeči, v dubnu 2015 však společnost Google ohlásila ukončení podpory tohoto doplňku[9] ve 42. verzi svého prohlížeče Chrome. Spouštění NPAPI doplňku je však možné nadále aktivovat v pokročilém nastavení prohlížeče. Ve verzi 45 však bude odstraněna i tato možnost. Je ovšem nutno podotknout, že javovský plugin není jediným, který bude takto blokován – stejný osud potká i doplněk pro Microsoft Silverlight či pro populární herní platformu Unity. Podle oznámení na oficiálním blogu[10] je motivací k tomuto kroku trend nárůstu počtu mobilních zařízení, které ať už z výkonnostních či licenčních důvodů tyto externí doplňky nepodporují.

Na závěr shrňme několik důvodů, proč byla vybrána právě platforma JavaFX

- Multiplatformnost
- Vynikající podpora animací ve spojení s hardwarovou akcelerací grafiky
- Spouštění jak ve webovém prostředí tak jako desktopová aplikace
- Možnost využití stávajících Java knihoven
- Schopnost oddělit návrh GUI a grafiky od samotné logiky programu (možnost sdílet jednotné grafické prvky)
- Budoucnost technologie a její neustálý rozvoj

3.2 Implementace vizualizací

Vizualizovat vývoj algoritmů nad vybranými strukturami je hlavním cílem této práce. K jeho dosažení byla vybrána podle kritérií popsaných v předchozí kapitole JavaFX. Samotná vizualizace by měla být schopna přehrát celou animaci plynule, tak po jednotlivých krocích, což umožňuje obsluhujícímu uživateli lépe proniknout do zkoumané problematiky a svým vlastním tempem si celý proces přehrát. Dalším praktickým požadavkem je zpětná animace, která v určitých situacích může být také přínosem. V neposlední řadě by uživatel měl mít možnost změnit samotnou rychlost animace.

[8] https://wiki.openjdk.java.net/display/OpenJFX/Main
[9] https://productforums.google.com/forum/#!topic/chrome/TYbj21PkcAQ
[10] http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html

3.2.1 Princip návrhu vizualizací

Samotný princip animace musí být nějakým způsobem synchronizován s probíhajícím procesem algoritmu v datové struktuře. Animace na změny provedené tímto procesem ve struktuře musí také nějakým způsobem reagovat. Jedním ze způsobů realizace je použitím paralelního vlákna animace, který poběží souběžně s procesem v datové struktuře a pomocí synchronizačních nástrojů bude reagovat na proběhlé změny. Toto řešení však není příliš elegantní z pohledu samotné implementace – musí se v tomto případě řešit uspávání animačního vlákna při přerušení animace, jeho předčasné ukončení a také správná synchronizace souběhu výpočetního a animačního vlákna. Nevýhodou by byla také velice těsná vazba s implementovanou strukturou.

Místo toho byl zvolen odlišný přístup – metoda generování událostí. Základní filozofií tohoto návrhu je generování jednotlivých událostí samotnou strukturou a jejich ukládání do fronty. Animační jádro pak z této fronty postupně jednotlivé nastalé události vybírá a provede jejich animaci. V závislosti na požadavku uživatele, zdali požaduje plynulou či krokovou animaci jsou jednotlivé kroky animace přehrány automaticky nebo ručně samotným uživatelem. Po dokončení vizualizace je fronta prázdná a uživatel může spustit novou animaci. V případě implementace zpětné animace bylo nutné frontu vyprazdňovat až před vytvořením další animace. Je to z toho důvodu, aby mělo animační jádro k dispozici již obsloužené události. V praxi se tedy z fronty během animace neodebírá, jen se posunuje ukazatel na aktuální událost požadovaným směrem.

4 Implementace komponent společných pro všechny projekty

V této kapitole je objasněna implementace komponent, které sdílí všechny vizualizace. Jedná se především o způsob implementace animací podporující zpětný průchod, návrh sytému pro obsluhu událostí a nástin společného tzv. *Layout manageru*, který se stará o správné rozmístění vykreslovaných prvků struktur.

4.1 Komponenta Animation Control

Jedním z dodatečných požadavků zadání této diplmové práce bylo umožňění zpětného průchodu animace. Protože výchozí podpora této funkcionality ve frameworku JDK je velice omezená, bylo třeba implementovat systém, který toto zajistí.

V tomto místě je nutno poznamenat, že animace celé operace algoritmu se skládá z přechodů (transitions). Přechodem se rozumí činnost, která v průběhu času změní stav objektu. V této práci je využito hned několik druhů přechodů – od toho nejběžnějšího, manipulující s polohou objektu (translate transition), až k těm, které ovlivňují viditelnost animovaného objektu (fade transition) nebo jeho velikost (scale transition).

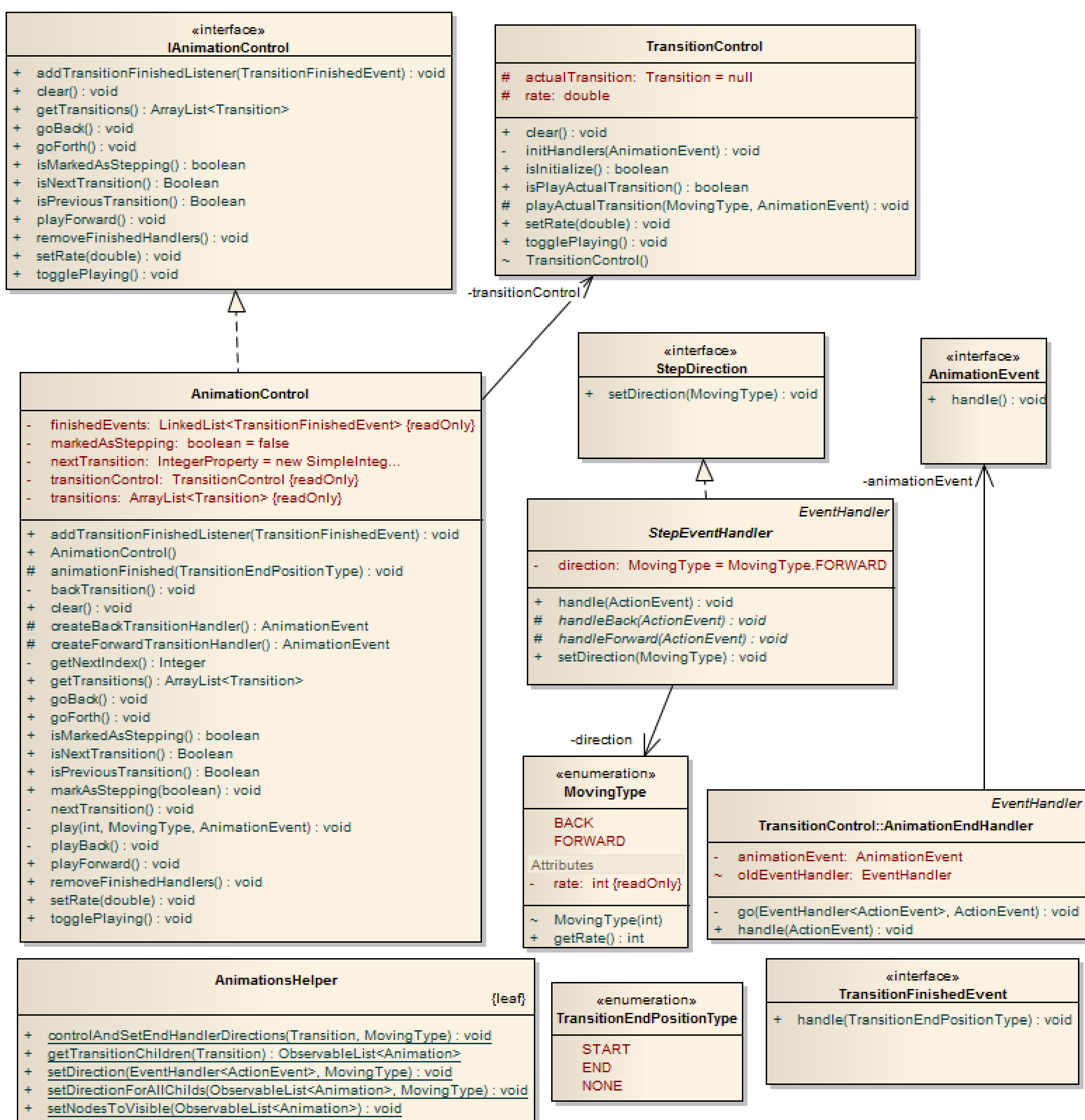
Samotná základní knihovna JavyFX umožňuje pouze nastavit násobek rychlosti provádění daného přechodu pomocí atributu *rate*. Ten může nabývat libovolných číselných hodnot – v případě, že jej nastavíme na hodnotu 0, přechod se zastaví. Nastavování tohoto násobiče je využito právě při implementaci zpětného průběhu – nastavíme jej na hodnotu -1, což znamená, že přechod se odehraje ve zpětném směru s jednotkovou rychlostí.

Hlavní problematikou ovšem bylo zajistit zpětný průběh celé animace (ne, tedy jen jednotlivého přechodu). Bylo třeba zajistit, aby při skončení přechodu ve zpětném směru bylo zajištěno přehrání přechodu předcházejícího, taktéž ve zpětném směru. A právě tento případ řeší nově implementovaný správce animace – *Animation Control*.

Výchozím bodem této komponenty je stejnojmenná třída *AnimationControl*, která nabízí API pro ovládání běhu celé animace. Vývojáři, který tuto komponentu chce využít ve svém projektu, umožňuje spouštět či zastavovat celou animaci, měnit její směr a ovlivňovat chování, jak po proběhnutí celé animace, tak při skončení jednoho z přechodů v jakémkoli směru pomocí mechanismu specifických handlerů. Její základní součástí je kolekce objektů typu *Transition*, která uchovává jednotlivé přechody v pořadí, ve kterém budou přehrány v dopředném směru. Tato kolekce je zpřístupněna pomocí veřejné metody *getTransitions*, což umožňuje přidávat či odebírat jednotlivé přechody do celé animace. Dalšími zajímavými metodami jsou například *markAsStepping* s parametrem booleanovské hodnoty, která zajišťuje nastavení krokování (v případě, že jejímu parametru přiřadíme hodnotu *true*, nebude další přechod v animaci přehrán, dokud jej výslovně nevyvoláme pomocí *goForth*) nebo *setRate*, jehož prostřednictvím dokážeme nastavit výše zmíněný atribut rychlosti animace.

Aby se zabránilo situaci „probliknutí“, kdy je prvek přidán do vykreslovacího panelu, přičemž není žádoucí, aby byl viditelný před patřičným přechodem, po němž se má

viditelným stát (tedy buď přechodem, který mění průhlednost objektu – *FadeTransition* nebo jeho velikost – *ScaleTransition* z nulové hodnoty na jednotkovou), je nutno vkládat animovaný prvek do panelu s nastavením viditelnosti na hodnotu *false*. Implementovaná animační komponenta tento neprůhledný prvek nastaví na viditelný až těsně před samotným provedením přechodu, který jej obsahuje. Zamezí se tak skokovému nastavení velikosti či průhlednosti na krátkou dobu před animováním daného elementu a tak nepříjemnému „bliknutí".



Obrázek 14: UML diagram tříd komponenty Animation Control

Zdroj: vlastní

4.2 Systém pro obsluhu událostí

K realizaci systému pro obsluhu událostí byla vybrána implementace tzv. sběrnice událostí - EventBus z knihovny Guava, která poskytuje i další užitečné funkce pro usnadnění některých často vytvářených operací při tvorbě projektů (např statická metoda toStringHelper v třídě MoreObjects, která značně zjednodušuje psaní toString metod nejen pro potřeby ladění).

Pro celou aplikaci existuje právě jedna sběrnice typu EventBus, která je inicializovaná v *Controller* třídě daného projektu. Reference na objekt sběrnice je následně předána konstruktorem do třídy animované struktury. V průběhu algoritmu struktury jsou pomocí metody *post* do sběrnice odesílány události jako specifické objekty. Ve výše zmíněném *Controlleru* je na sběrnici registrován objekt, který bude tyto objekty zpracovávat. Tento přístup nám umožňuje velice volnou vazbu mezi producentem událostí a jejich zpracovatelem (konzumentem). Na straně příjemce zpráv tak můžeme mimo animační třídy zaregistrovat i prostý výpis zachytávaných událostí, což je při testování a ladění animace velice užitečná vlastnost.

Samotná obsluha událostí je prováděna přiřazením anotace Subscribe k metodě, která obsahuje vstupní parametr stejného typu, jako je objekt události.

Tvůrci implementace EventBus z balíku Guava na svých webových stránkách[11] nabízejí stručný a rychle pochopitelný návod pro využití tohoto systému.

4.3 Obecný layout manager pro binární strom

Aby bylo možné korektně zobrazit hierarchickou strukturu stromu v grafické podobě, je nutno vytvořit správce rozmístění jednotlivých grafických komponent na vykreslovaném panelu (tzv. layout manager). Autor toho obecného, jenž se pak v různých obměnách a úpravách používá ve čtyř z pěti animací struktur této práce, je Vojtěch Müller, který jej široce využívá i ve své diplomové práci. Celá tato komponenta je k dispozici ve společné knihovně MT-common, která je též k dispozici na portále GitHub[12].

Obecná implementace tohoto správce bere v potaz pouze strom ve své binární podobě. Beze změny je tedy v této práci použit u struktur binárního vyhledávacího a AVL stromu. U animací Quad-stromu a 2-3 stromu musel být algoritmus rozmístění grafických prvků upraven tak, aby splňoval požadavky na zobrazení těchto struktur. V případě Skip listu se využívá jiný způsob organizace grafických elementů.

Základní třídou je BinaryTreeLayoutManager, která zprostředkovává operace s rozmístěním všech prvků. Při inicializaci objektu tohoto typu musíme konstruktorem specifikovat nastavení rozmístění prvků pomocí TreeLayoutSettings objektu, který definuje velikosti elementů a minimální vzdálenost mezi nimi, a panel, na který se bude

---

[11] https://code.google.com/p/guava-libraries/wiki/EventBusExplained
[12] https://github.com/Maxxa/MT-common

struktura vykreslovat. Dalším volitelným parametrem je způsob přepočtu prvků – tzn., zdali bude správce přepočítávat celkovou šířku stromu podle maximálního možného či aktuálního počtu potomků v listové úrovni.

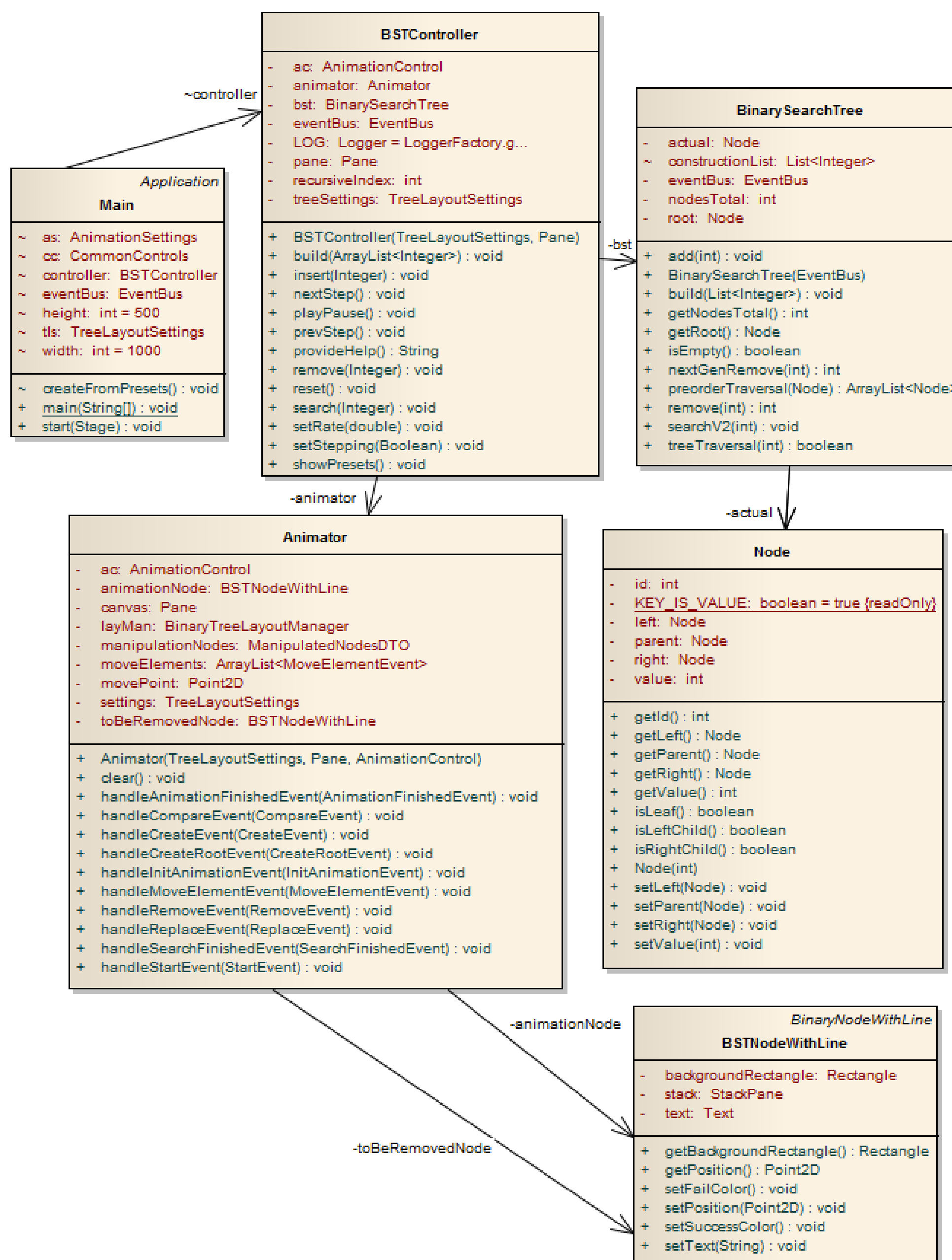
Pro přidání prvku do layout manageru pomocí metody add je třeba definovat vkládaný prvek jako takový, dále unikátní id jeho rodiče (v případě, že se jedná o kořen stromu, je předána hodnota null) a jeho pozici vůči rodiči (jestli se jedná o levého či pravého potomka). Volitelně také můžeme určit možnost, zdali prvek vložit i do samotného kreslicího panelu či nikoli (v případě, že z jakýchkoli důvodů chceme jeho samotné vložení realizovat později). Pokud tento parametr není zadán, je do panelu prvek přidán automaticky po definování své polohy v něm.

Odebírání probíhá prostým předáním unikátního identifikátoru elementu jako parametr metodě removeElement. Za zmínku stojí užitečná metoda getElementInfo, díky které můžeme získat nejen grafickou reprezentaci prvku jako takového (což je nejčastější využití při zpřístupňování elementu k animaci), ale např. i referenci na rodič prvku reprezentovanou atributem idParent.

Na závěr této podkapitoly je nutno poznamenat, že tento layout manager nijak nezabezpečuje rozmístění spojnic mezi vrcholy stromu.

5 Implementace Binárního vyhledávacího stromu

V této kapitole bude naznačena implementace Binárního vyhledávací stromu (dále jako BVS).

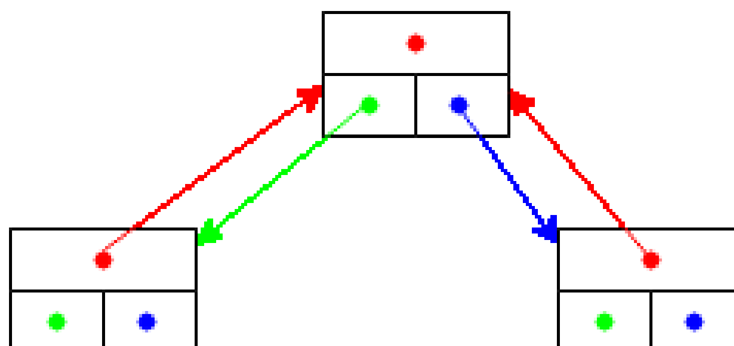


Obrázek 15: UML diagram tříd BVS

Zdroj: vlastní

Implementace BVS je založena pouze na přímých referencích mezi jednotlivými prvky. Ty jsou realizovány jak ve směru rodič-potomek, tak obráceně (při vytváření uzlu je specifikován jeho rodič).

Tyto reference se mění pouze ve dvou případech – při odebírání prvku, kdy jsou patřičné odkazy odstraněny a při jeho výměně (swapu), kdy před samotným odebráním vyměňujeme vnitřní prvek za svého in-order následovníka.



Obrázek 16: Ilustrace paměťové reprezentace BVS

Zdroj: Vlastní

5.1 Operace Najdi

Jak bylo uvedeno v kapitole 2.1, vyhledávání v BVS probíhá postupným porovnáváním hledané hodnoty s aktuálním klíčem a následným přechodem do levého či pravého podstromu (dle výsledku porovnání).

Realizace tohoto principu je poměrně jednoduchá. Začíná se přiřazením kořene aktuálním prvku. Proběhne porovnání hodnoty klíče aktuálního prvku s hledanou hodnotou a v případě neshody se zpřístupní patřičný potomek (pokud existuje) a celý proces se opakuje. Ukončovací podmínkou algoritmu je buď nalezení hodnoty klíče (hledaná hodnota se rovná hodnotě klíče aktuálního prvku) nebo pokus zpřístupnit neexistujícího potomka (v porovnávání není možné pokračovat – tedy prvek nemůže být nalezen).

Vzhledem k tomu, že je tato struktura implementována jako dynamický strom s explicitními referencemi, je velice prosté zpřístupnit si daného potomka přes patřičnou metodu *getLeftChild* nebo *getRightChild*.

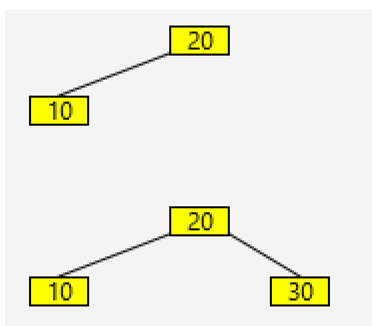
Abychom mohli patřičnou operaci animovat, jsou v tomto případě vytvářeny dva typy událostí. Prvním z nich je *CompareEvent*, který obsahuje hledanou hodnotu a unikátní identifikátor prvku, jehož klíč je porovnáván (vzhledem k tomu, že každý klíč je ve stromě unikátní, je tento identifikátor současně také hodnotou porovnávaného klíče). Tím druhým je *SearchFinishedEvent*, který animační jádro informuje o výsledku vyhledávání – tzn., zdali byl daný klíč ve struktuře nalezen či nikoli.

Animace celého traverzování je realizována způsobem přesouvání prvku pomocí *TranslateTransition* přechodů na pozice porovnávaných prvků, které jsou získávané z layout manageru.

5.2 Operace Vlož

Proces vložení prvku do stromu je poměrně prostý. Nejdříve je pomocí algoritmu pro vyhledávání nalezeno místo pro vkládaný uzel. Nový prvek je vytvořen a jsou nastaveny patřičné reference. Současně s tím je vytvořena událost *CreateEvent*, který obsahuje klíč nově vloženého uzlu a jeho rodiče (který je požadován při vkládání do layout manageru).

V animační třídě *Animator* je následně vytvořen nový grafický element, který je vložen do layout manageru a pomocí přechodu změny průhlednosti prvku je animováno jeho zobrazení v kreslicím panelu. Stejným způsobem je zobrazena i spojnice k rodičovskému prvku.



Obrázek 17: Vložení prvku s klíčem 30 do BVS

Zdroj: vlastní

Za situace vkládání již existujícího elementu, je vyhozena výjimka, která je následně zpracována v třídě *BSTController* a uživateli je zobrazeno chybové hlášení.

5.3 Operace Odeber

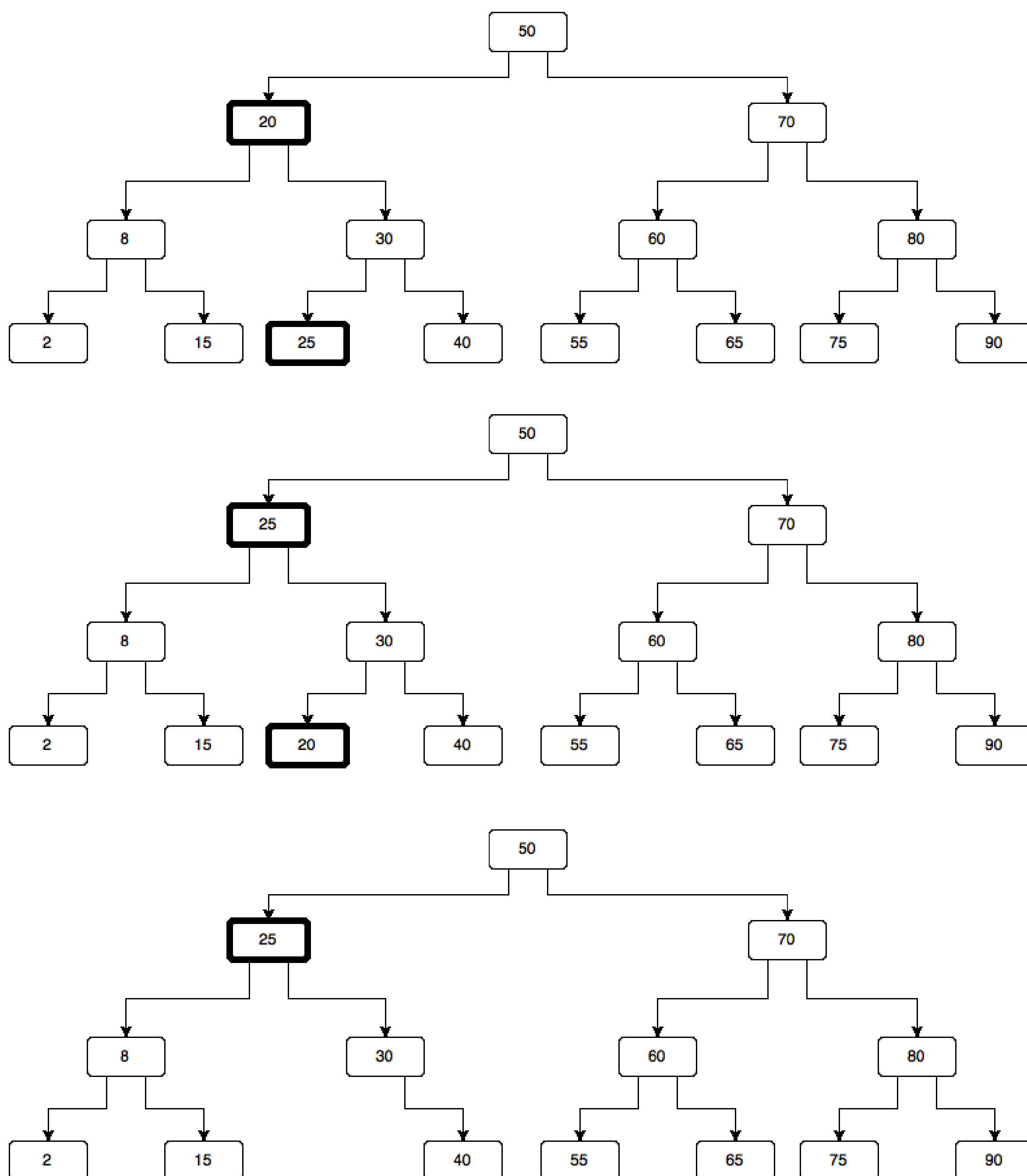
Stejně jako u operace Vlož je nedříve nutno lokalizovat odebíraný prvek pomocí vyhledávacího algoritmu. Za předpokladu, že požadovaný prvek není ve struktuře nalezen je opět zobrazeno chybové hlášení uživateli o odebírání neexistujícího prvku.

Při odebírání prvku může dojít k případu, kdy nalezený prvek není listem (má tedy jednoho či více potomků). Aby mohl být odebrán a současně neporušil integritu struktury, je nutno jej vyměnit se svým in-order následníkem (což je nejlevější prvek z pravého podstromu). Tím se z principu stane listem, který je připraven k odebrání. Animační komponentě je toto předáno jako událost *ReplaceEvent* obsahující klíč jak odebíraného prvku, tak následníka. Správce rozložení grafických elementů je toto schopen interpretovat a vyměnit pozice těchto prvků podle jejich klíčů.

Ve struktuře je prvek odstraněn okamžitě – reference jsou nastaveny na hodnotu null, tím dojde k jeho odpojení ze struktury a smazání garbage collectorem. Přitom je animační část informována o této operaci pomocí *RemoveEventu*.

Abychom zajistili možnost zpětného průchodu, není prvek okamžitě odstraněn z kreslicího panelu. Dojde pouze k animaci skrytí elementu a jeho spojnice s rodičem, přičemž je

současně označen jako „to remove“ (tzn. prvek označen k odebrání) a samotné smazání z grafické reprezentace a panelu je provedeno při začátku následující animace.



Obrázek 18: Ilustrace odebrání vnitřního uzlu s klíčem 20

Zdroj: vlastní

6 Implementace AVL stromu

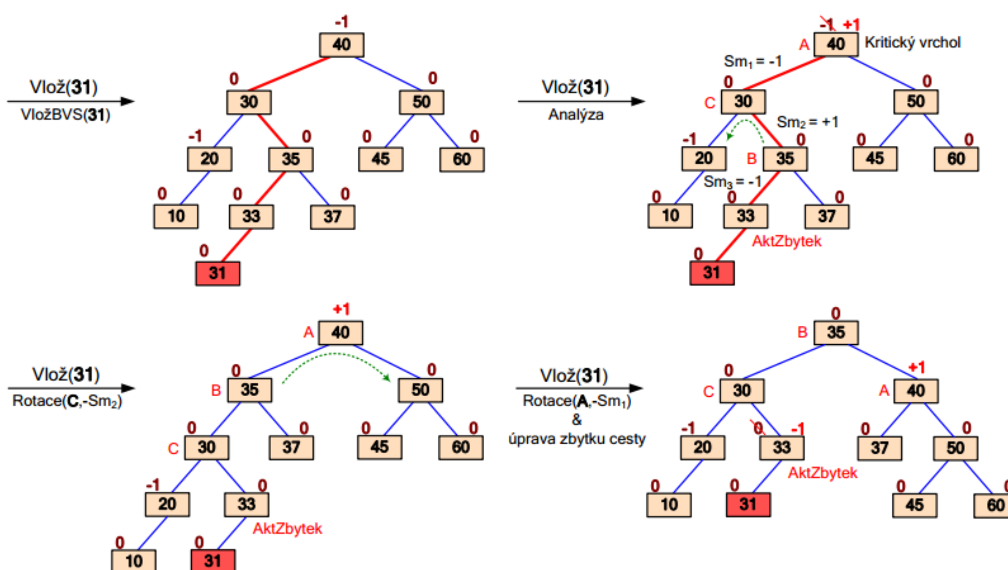
AVL strom je ze své podstaty velice podobný klasickému BVS. Až na princip vyvažování stromu při strukturálních změnách je vlastně jeho implementace totožná s klasickým BVS. V této kapitole tak bude dán důraz zejména na implementační rozdíly v důsledku jeho vyvažovacího mechanismu.

6.1 Operace Najdi

Protože operace *Najdi* nemění organizaci struktury, je vyhledávání v AVL stromu zcela identické jako v případě klasického BVS. Opět probíhá od kořene směrem k listům, je porovnávána hodnota vkládaného klíče s příslušnou hodnotou prvku stromu. V závislosti na výsledku tohoto porovnávání pak následně probíhá přechod do pravého či levého podstromu prvku. O výsledku vyhledávání je uživatel na konci graficky informován.

6.2 Operace Vlož

První fáze vložení nového prvku do struktury probíhá stejně jako v případě BVS. Nejdříve je vyhledána pomocí operace *Najdi* příslušná pozice, poté je vytvořen nový uzel stromu a prvek do něj vložen. V tomto okamžiku dochází ke změně uspořádání struktury, a je proto nutné aktualizovat všechny vrcholy stromu na cestě od vkládaného prvku směrem ke kořeni. Jednou z variant implementace je ukládání zpětné cesty během samotného vyhledávání – to v tomto případě není třeba. Protože každý prvek obsahuje referenci na svého rodiče, je velice triviální procházet strukturu zpětně (od listů) a průběžně tak aktualizovat vyváženost vrcholů. Ukončovací podmínkou je dosažení kořenového vrcholu (reference na rodičovský prvek je rovna hodnotě *null*). Po této aktualizaci se provede vyvážení stromu, které je uvedeno v kapitole 6.4.



Obrázek 19: Ilustrace evoluce algoritmu při operaci Vlož

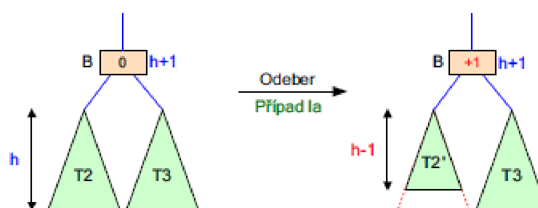
Zdroj - vlastní

6.3 Operace Odeber

I v případě odebírání prvku ze struktury je nejdříve postupováno stejně jako v případě BVS. Za předpokladu, že je odebíraný uzel vnitřním prvkem, je vyměněn se svým in-order následovníkem a následně odstraněn. V případě, že je listem, je jednoduše odstraněn. Protože opět došlo ke změně organizace struktury, je třeba opět vyvážení prvků analyzovat a přistoupit k případnému vyvážení stromu.

Po odebrání prvku může dojít k pěti případům, které je nutno vyvažováním řešit. Každý z těchto případů má dvě symetrické varianty (v závislosti na tom, zdali odebíraným byl levý, resp. pravý potomek rodiče).

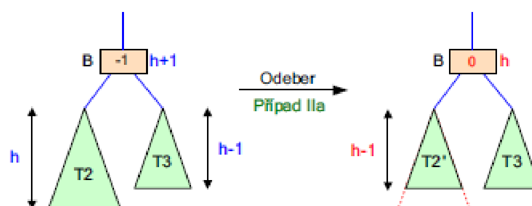
I. Příslušný podstrom byl před odebráním prvku vyvážen (rozdíl výšek podstromu byl 0), odebráním dochází ke snížení výšky levého či pravého podstromu o 1.



Obrázek 20: Odstranění prvku z AVL stromu - případ I

Zdroj - vlastní

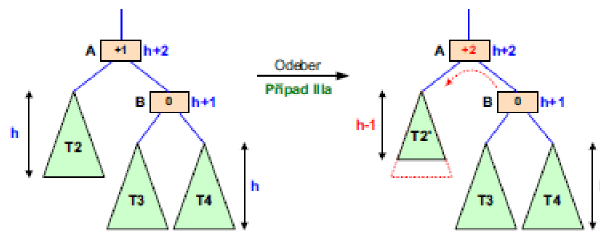
II. Po odebrání prvku je rozdíl výšek podstromů roven 0, je nutné další zkoumání směrem ke kořeni stromu (mohlo dojít k situaci nevyvážení v některém z rodičovských vrcholů).



Obrázek 21: Odstranění prvku z AVL stromu - případ II

Zdroj - vlastní

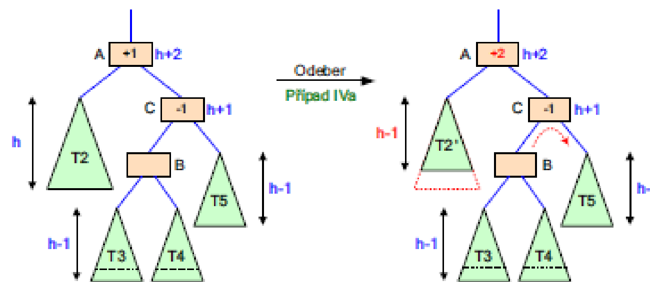
III. Po odebrání prvku došlo ke zvýšení rozdílu mezi výškami podstromu na hodnotu 2, respektive -2, přičemž sousední podstrom je vyvážený.



Obrázek 22: Odstranění prvku z AVL stromu - případ III

Zdroj - vlastní

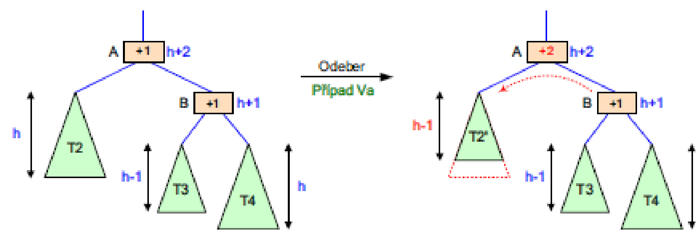
IV. Odebrání prvku způsobilo nevyvážení podstromu, přičemž sousední podstrom není zcela vyvážen v opačném směru (viz následující obrázek).



Obrázek 23: Odstranění prvku z AVL stromu - případ IV

Zdroj - vlastní

V. Po odebrání prvku došlo ke zvýšení rozdílu mezi výškami podstromu na hodnotu 2, respektive -2, přičemž sousední podstrom je nevyvážený ve stejném směru. I v tomto případu je nutné další zkoumání směrem ke kořeni, mohlo totiž dojít k nevyvážení stromu na vyšších úrovních.



Obrázek 24: Odstranění prvku z AVL stromu - případ V

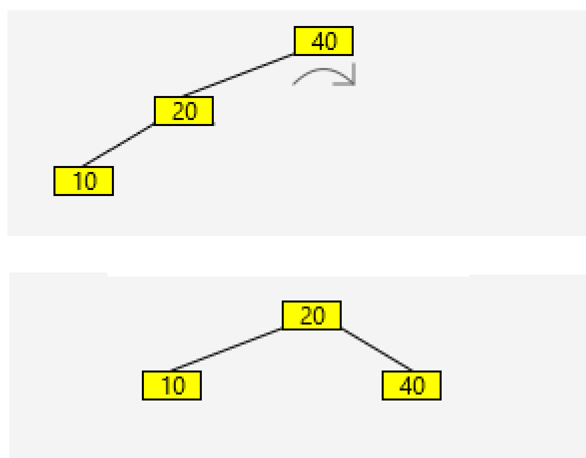
Zdroj - vlastní

6.4 Realizace vyvažování AVL stromu

V předchozí části bylo uvedeno, že stav vyváženosti AVL stromu je zajištěn rotacemi, tedy způsobem přemístění prvku v jednom či druhém směru, aby došlo ke snížení rozdílu mezi výškami podstromů jeho rodiče. Je pochopitelné, že tato operace nastává pouze při změně struktury (operacích Odeber a Vlož).

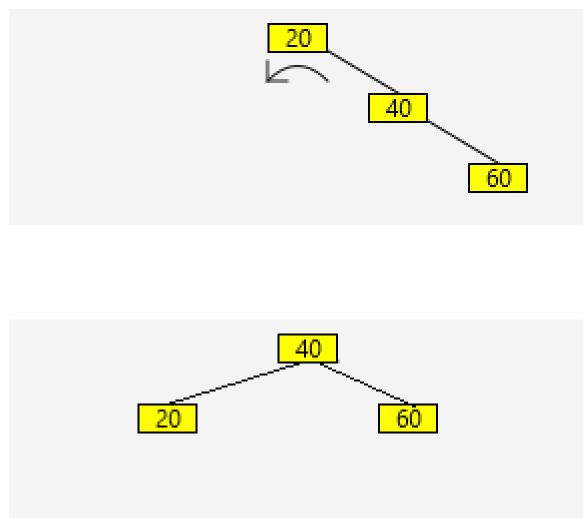
Po úspěšné změně struktury (prvek byl vložen či odebrán) je pomocí metody *recursiveBalancing* analyzována vyváženost daného vrcholu. V případě, že je daný vrchol nevyvážený, tzn. rozdíl mezi výškami jeho levého a pravého podstromu je 2 nebo -2, je iniciována rotace potřebným směrem. Za předpokladu, že vyvažovaný uzel není kořenem, je metoda *recursiveBalancing* rekurzivně zavolána na jeho rodičovský vrchol a celý algoritmus se opakuje.

Samotná levá, resp. pravá, rotace je na straně struktury implementována jako prosté přepojení referencí prvků.



Obrázek 25: Zobrazení pravé rotace

Zdroj: vlastní



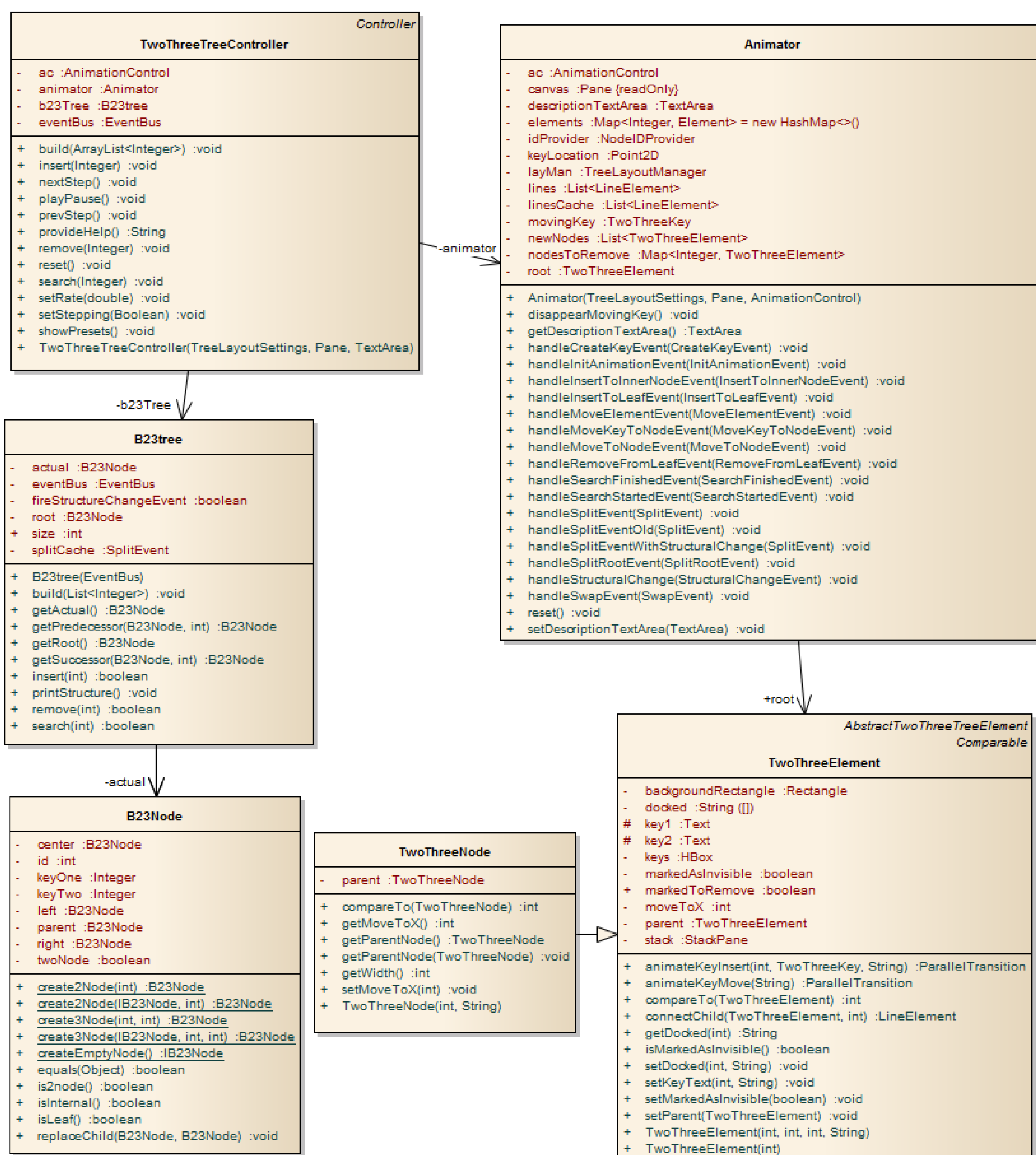
Obrázek 26: Zobrazení levé rotace

Zdroj: vlastní

Podstatně komplikovanější je animace těchto rotací, protože se na rozdíl od některých triviálnějších operací mění větší část struktury. K aktualizaci zobrazení grafické reprezentace struktury je nutno nejdříve analyzovat změnu struktury jako takové. Implementace je vytvoření popisu struktury od kořenového prvku podstromu, ve kterém k rotaci došlo. Pomocí utilitární třídy *TreeStructureBuilder* je tento popis vytvořen a následně předán jako parametr události *RotationLeftEvent* resp. *RotationRightEvent* (v závislosti na směru rotace). Animační část projektu se poté postará o korektní přepojení spojnic vrcholů a pomocí layout manageru vytvoří animaci posunu přesouvaných prvků. Drobným detailem, jenž zlepšuje ilustraci směru rotace, je přidání šipkového ukazatele.

7 Implementace 2-3 stromu

Vzhledem ke komplexnosti této struktury byla její implementace nejnáročnější. V rámci operací, ve kterých dochází ke změně struktury, dochází k tak zásadním změnám vztahů mezi prvky, že bylo vytvořeno i textové pole informujících uživatele o proběhlých akcích v rámci aktualizace stavu struktury. Bylo také nutno upravit správce rozmístění grafických elementů tak, aby podporoval možnost, že každý prvek může mít až tři potomky. Stejně jako v případě všech stromových struktur v této práci, je implementace založená na explicitních referencích mezi jednotlivými uzly (dynamický strom).



Obrázek 27: UML diagram tříd 2-3 stromu

Zdroj: vlastní

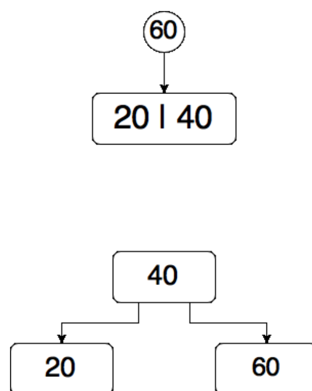
7.1 Operace Najdi

Realizace této operace je v mnoha ohledech podobná jako u předchozích struktur. Rozdílem je princip porovnávání u prvku se dvěma klíči – v případě, že porovnávaná hodnota je větší než první s klíčů, ale současně menší než druhý klíč, probíhá přesun na prostředního z potomků. Samotná animace je analogická jako u BVS či AVL stromu (opět dochází k přesouvání klíče mezi jednotlivými uzly v pořadí, v jakém probíhá porovnávání hodnoty. Pokud je klíč nalezen, je navíc ještě pro větší přehlednost zvýrazněn v elementu, který jej obsahuje.

7.2 Operace Vlož

Po vyhledání uzlu, do kterého se bude vkládat nový klíč, je třeba rozhodnout, zdali je v daném prvku na vkládaný klíč místo. Za předpokladu, že cílový element má před samotným vložením pouze jeden klíč, je tento vložen na volnou pozici. S tím souvisí i animace přesunu stávajícího klíče prvku a uvolnění pozice pro nově vkládaný.

Může ovšem dojít k situaci, kdy na vkládaný klíč jednoduše ve specifikovaném uzlu není místo (jedná se o prvek se dvěma klíči). V tomto případě se musí prvek rozdělit na dva jedno-klíčové a klíč s prostřední hodnotou z trojice je přesunut do rodičovského uzlu. V něm může dojít ke stejné situaci a celý postup se opakuje až ke kořeni celého stromu. Změna struktury tak může být potencionálně velmi radikální, v případě rozdělování u kořenového elementu se dokonce výška celého stromu zvýší o 1 (kvůli vložení nového kořenu).



Obrázek 28: Rozdělení dvou-klíčového uzlu při vkládání klíče

Zdroj: vlastní

Z implementačního hlediska je nejprve nalezený prvek identifikován podle počtu klíčů jako jedno-klíčový (v kódu označen jako 2node) nebo dvou-klíčový (3node)[13]. Za

[13] 2node a 3node jsou po implementační stránce shodné objekty, jediným rozdílem je, že prvek s jedním klíčem (2node) má hodnotu druhého klíče nastavenu na null. Současně s tím je nastaven i booleanovský příznak is2node na true.

předpokladu, že se jedná o jedno-klíčovou variantu, je tento konvertován na dvou-klíčový, klíč je vložen na patřičnou pozici a algoritmus je ukončen (je vhodné zmínit, že proces vkládání klíče začíná u listových prvků, tudíž v tomto případě není nutno nikterak upravovat reference na potomky uzlu). K rozdělení (splitu) dojde v případě, že je vyhledaný prvek již dvou-klíčovým. Je zavolána metoda *splitNode*, která vytvoří trojici vzájemně propojených jedno-klíčových vrcholů – konvertovaný rozdělovaný prvek, nově vzniklý prvek a pomocný uzel s klíčem, který bude přesunut do rodiče. Současně s tím vytvoří událost *SplitElementEvent*, informující objekt *Animator* o proběhlém dělení prvku. Pokud byl dělený prvek navíc prvkem vnitřním, jsou odpovídajícím způsobem upraveny reference na příslušné potomky.

7.3 Operace Odeber

I při odebírání prvku může dojít k různým situacím. V případě, že je prvek ve struktuře obsažen (dojde k jeho vyhledání) je zjištěno, jestli je prvkem vnitřním nebo listem. Vnitřní prvek je opět (stejně jako u předchozích stromů) vyměněn se svým in-order následníkem a proces odstraňování tak začíná opět od listové úrovně. Zde je nejjednodušším případem stav, kdy je klíč odebírán z prvku, který má klíče dva. Klíč je z uzlu odebrán, který je následně převeden na jedno-klíčový (2node).

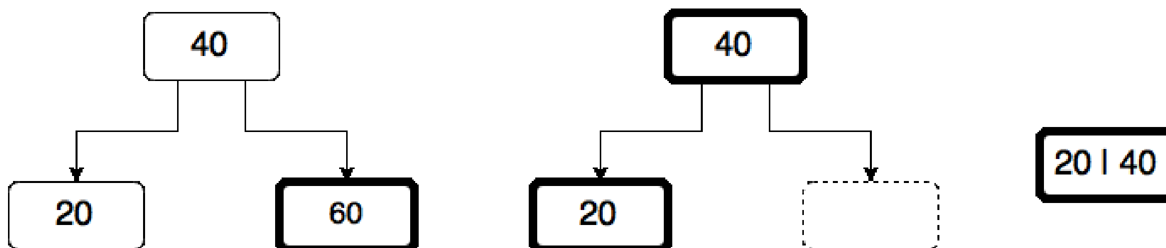
Může se stát, že prvek obsahuje pouze odebíraný klíč. Algoritmus poté zjišťuje, zdali prvek měl sourozence, který obsahuje klíče dva (každý uzel obsahuje reference jak na potomky, tak i na rodiče, tím je zajištěno zpřístupnění potomků rodiče a prostým porovnáním následně zjištěn sourozenec a jeho vlastnosti). Pokud ano, je klíč oddělující tyto dva elementy přesunut na místo odstraňovaného klíče a současně je přilehlý klíč z dvouklíčového stromu přesunut místo něj do rodičovského prvku. Dvou-klíčový sourozenec je následně převeden na jedno-klíčový.

Obrázek 29: Odstraňování klíče v případě existence dvou-klíčového sourozence

Zdroj: vlastní

Pokud ovšem žádný takový sourozenec prvku neexistuje, je třeba provést operaci Sloučení (merge prvku). To je realizováno tak, že je vytvořen nový prvek, do kterého je následně přesunut klíč ze sourozence prvku a klíč z rodiče, který oba uzly odděloval. Veškeré

referencejsou následně aktualizovány, rodič nově vzniklého 3nodu je analyzován a algoritmus se případně opakuje směrem ke kořeni.



Obrázek 30: Odstraňování klíče v případě neexistence dvou-klíčového sourozence

Zdroj: vlastní

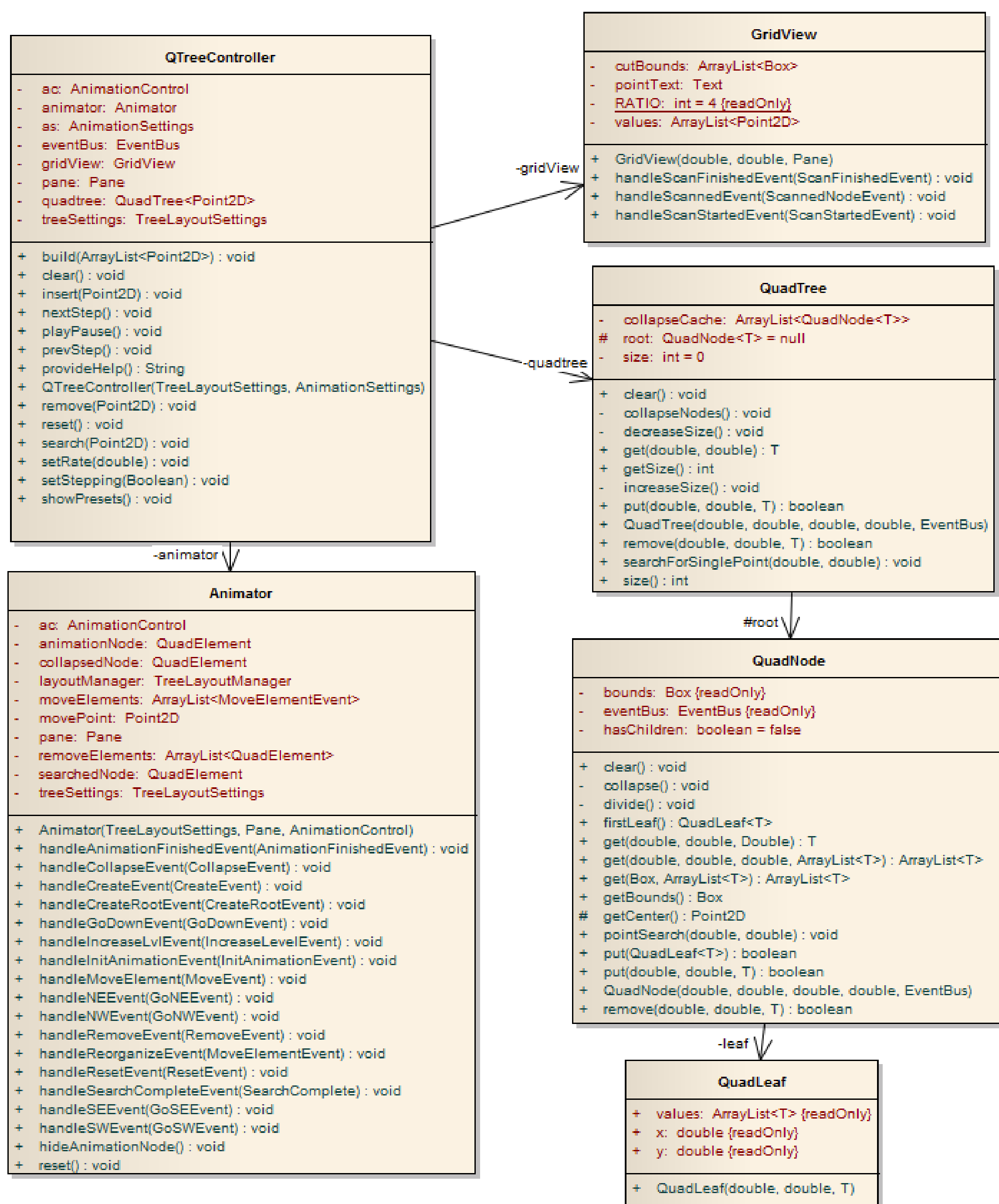
Případné konverze mezi jednotlivými druhy uzlů v závislosti na počtu klíčů jsou realizovány vytvořením nového prvku, kterému je předán unikátní identifikátor původního elementu. Všechny případné reference jsou přepojeny dodatečně.

7.4 Animace změny struktury

V případě, že došlo k zásadním změnám celé struktury (ať už při dělení či sjednocování prvků) je vytvořena událost *StructureChangedEvent*, která stejně jako při upozornění na rotaci v AVL stromu popisuje nový stav struktury, jediným rozdílem oproti AVL je, že u 2-3 stromu dochází k aktualizaci celého stromu, ne pouze jeho části (podstromu). Událost je zachycena v handler metodě třídy *Animator*, kde je nejdříve zjištěno, jestli nedošlo k vytvoření nebo odebráních některých z elementů. Je provedeno příslušné přidání nově vzniklých prvků, resp. skrytí a označení prvků k odstranění. Poté jsou aktualizovány polohy stávajících elementů struktury a nakonec příslušející klíče prvků. Současně jsou popsány změny v rozvržení stromu do textového pole (kvůli lepší přehlednosti).

8 Implementace Quad stromu

Realizace této struktury byla poměrně specifická oproti ostatním. To je způsobeno faktem, že se jedná o strom pracující s dvourozměrnými (2D) daty. To se odrazilo jak na úpravě uživatelského rozhraní (mimo možnosti zadávat 2D data i schopnost přepnutí na druhý způsob zobrazení stromu), tak i na layout manageru, který v tomto případě musí podporovat právě čtyři potomky u každého uzlu. Zobrazení této struktury má tendenci značně růst do šířky při vkládání vyšších úrovní – například strom o výšce 5 může v listové úrovni obsahovat až 256 prvků. Není proto doporučeno vkládat příliš mnoho úrovní do struktury, její stromové zobrazení se tak může stát velice snadno hůře přehledným. Jednotlivé uzly stromu jsou označeny bodem, který představuje střed kvadrantu dělené plochy.



Obrázek 31: UML diagram tříd Quad stromu

Zdroj: vlastní

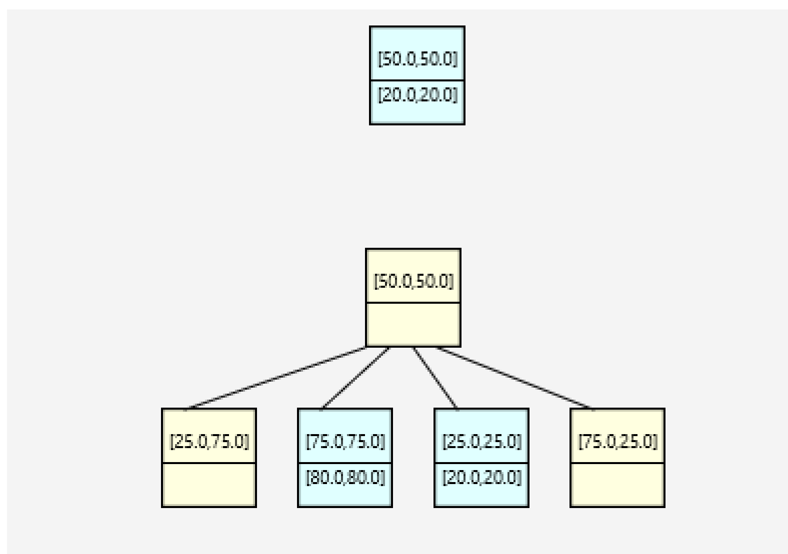
8.1 Operace Najdi

Vyhledávání je opět velice podobné předchozím implementacím, animovaný prvek je postupně přesouván mezi porovnávanými elementy, ve směru od kořene k listům, přičemž je porovnávána jak x-ová, tak y-ová souřadnice hledané hodnoty vůči odpovídajícím souřadnicím středu kvadrantu, který je uchováván v porovnávaném uzlu. Tím je určen

směr průchodu (a současně i potomek, do kterého bude algoritmus přecházet v dalším kroku). Výsledek hledání je vyhodnocen analogicky jako u ostatních struktur – za přepdokladu, že v listovém uzlu je obsažena hledaná hodnota, je výsledek vyhledávání kladný, v opačném záporný a je odpovídajícím způsobem znázorněn uživateli v animační vrstvě.

8.2 Operace Vlož

U Quad stromu jsou nové hodnoty vkládány vždy do listů, vnitřní vrcholy slouží pouze k definování cesty při jejich vyhledávání. Nejdříve je tedy vyhledán list, do nějž bude zadaná hodnota vložena. Za předpokladu, že je list prázdným, je do něj vložena a algoritmus je ukončen. Pokud by však došlo k situaci, kdy je požadovaný listový prvek obsazen jinou hodnotou, tedy v daném kvadrantu by v případě prostého vložení byly dva různé prvky, dojde k vytvoření nové úrovně – původní vkládaný list se stane rodičem pro čtveřici nových potomků (z pohledu plošného zobrazení dojde k rozdělení plochy listu na čtyři kvadranty)[14]. Následně je hodnota z listu přesunuta do jednoho z nově vzniklých potomků, stejně jako nově vkládaná hodnota. Tím dochází k postupnému růstu celého stromu v závislosti na postupném dělení kvadrantů. Vlastností této implementace je bohužel fakt, že při vkládání hodnot, jejichž hodnoty souřadnic jsou si blízké, může dojít k poměrně rychlému růstu stromu do hloubky. Vložení prvku je uživateli zobrazeno změnou podbarvení grafického elementu.



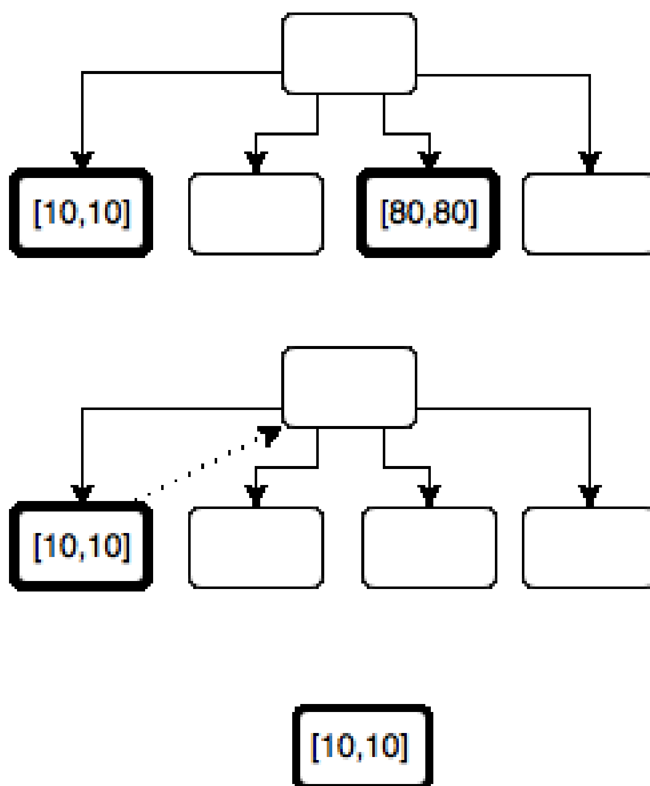
Obrázek 32: Změna stavu struktury po vložení prvku [80,80]

Zdroj: vlastní

[14] Je vytvořena událost *IncreaseLevelEvent*, která informuje animační vrstvu o potřebě vytvoření nových potomků a napojení na dosavadní list (který je specifikován jako parametr události).

8.3 Operace Odeber

Jak již bylo naznačeno dříve, hodnoty jsou uloženy pouze v listových element. Tento fakt dělá operaci odebrání poměrně triviální. Po vyhledání hodnoty a identifikaci listu, který ji obsahuje, dojde k jejím prostému smazání. Listový element jako takový je zachován, stále představuje kvadrant dělené plochy, nyní ovšem neobsazený. Jediným zvláštním případem je situace, kdy dojde k odstranění předposlední hodnoty v dané úrovni (tzn. po provedení operace odebrání, zůstane hodnota pouze v jednom ze sourozeneckých prvků – kvadrantů). V tomto případě je kvůli snížení paměťové náročnosti struktury hodnota v posledním obsazeném prvku přemístěna do svého rodiče a dojde ke smazání čtveřice nyní již prázdných kvadrantů a dealokace paměťového prostoru. Animační objekt je o této akci informován pomocí události CollapseElementEvent, jejíž zachycení způsobí skrytí a pozdější odstranění reprezentujících grafických prvků ze struktury.

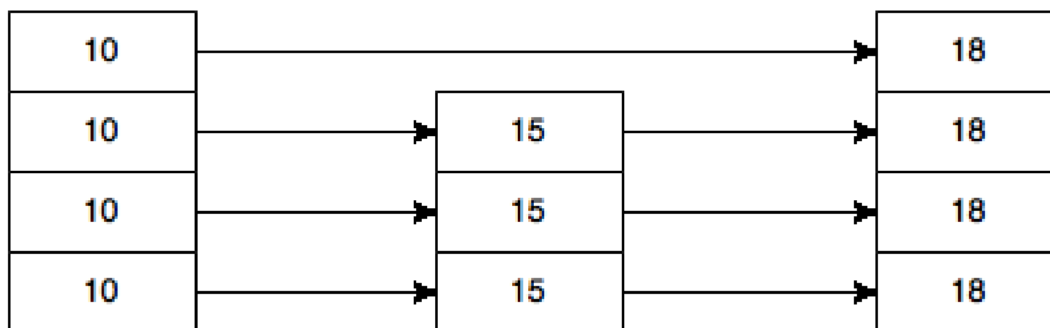


Obrázek 33: Odstranění hodnoty [80,80] a následná redukce struktury

Zdroj: vlastní

9 Implementace hierarchie seznamů (skip listu)

Tato struktura byla implementována poněkud odlišně než ostatní. Jedním z rozdílů je fakt, že jako jediná není organizovaná jako stromová hierarchie. Druhým rozdíl je ten, že tato struktura byla implementovaná jako první bez nutnosti použití jakékoli externí knihovny. Není zde dokonce ani obsažen systém EventBus z balíku Guava, ale prostá fronta událostí. V neposlední řadě je třeba zmínit mírně odlišnou paměťovou reprezentaci, která místo přímých referencí jednotlivých uzlů využívá elementární datovou strukturu typu pole k implementaci jednotlivých seznamů. Nicméně prvky jsou v horizontálním směru stále spojeny pomocí referencí (na následujícím obrázku znázorněny šipkami).



Obrázek 34: Paměťová reprezentace Skip listu

Zdroj: vlastní

Správce rozmístění grafických elementů je v zásadě tabulkového typu. Jednotlivé grafické prvky jsou rozmístěny jako buňky tabulky. Jejich pozice je uchovávána jako souřadnice z pohledu řádků a sloupců, prvky vlastní struktury začínají prvním řádkem a prvním sloupcem, nultý sloupec reprezentuje popisky úrovní. Při pohybu grafického elementu ve znázornění struktury se inkrementují čítače představující aktuální polohu v řádku, resp. sloupci. Poté dojde k přepočtu na souřadnicový systém vykreslovacího panelu a je tak jednoznačně identifikována pozice grafického prvku na kreslicí ploše. UML diagram tříd je pro svou rozsáhlost zařazen jako příloha této práce.

9.1 Operace Najdi

Vyhledávání ve struktuře začíná zpřístupněním prvního prvku na nejvyšší úrovni (headeru). Poté pokračuje procházení po nejvyšší úrovni, dokud není splněna podmínka, že hledaná hodnota je vyšší než hodnota následujícího prvku v úrovni vpravo nebo následující hodnota není null. Po dosažení této podmínky se postupuje o úroveň níže, a horizontální posun se opakuje. Tímto způsobem je dosaženo nejnižší úrovně, dokud není požadovaný prvek nalezen.

9.2 Operace Vlož

Vkládání opět začíná vyhledáním místa pro vložení nového seznamu. V průběhu hledání jsou ovšem do pomocné struktury uloženy reference na prvky následujícího seznamu. To umožňuje napojení na patřičné elementy v horizontálním směru na jednotlivých úrovních. Počet prvků seznamu je určen pomocí generátoru pseudonáhodných čísel, jehož výstupem je hodnota v rozsahu 1 – 7. Nakonec je nový seznam (implementovaný jako pole) alokován a v cyklu jsou jednotlivé prvky napojeny na svoje následovníky v horizontálním směru. Celá operace vkládání je animována a uživateli je zobrazena informace o počtu generovaných prvků. Pokud není nový seznam vkládán na konec struktury, je třeba taktéž animovat posun všech následujících grafických elementů o jedno místo doprava.

9.3 Operace Odeber

Princip odebírání je aplikován na seznam jako celek. Po nalezení odstraňovaného seznamu je tento ze struktury odpojen a následně dealokován. Aby bylo možné vzniklou mezeru po odstranění opět propojit, je během vyhledávání opět do pomocné struktury uložena reference na prvky následujícího seznamu. Stejně jako u všech předchozích struktur jsou grafické elementy reprezentující jednotlivé uzly nejprve skryty a přímo odstraněny až při začátku následující animace. Opět je třeba zobrazit i stav, kdy je seznam odebírán zevnitř struktury a je tedy nutno realizovat posunutí všech následujících elementů, tentokrát o jedno místo doleva.

Závěr

Tématem diplomové práce byla vizualizace evolucí algoritmů různých datových struktur. Nejprve byl proveden rozbor stávajících řešení této problematiky. Jedním ze závěrů této analýzy bylo zjištění, že žádné z existujících řešení nenabízí přehled vizualizací všech běžně užívaných datových struktur. Současně s tím postrádají některé užitečné funkcionality, zejména z hlediska ergonomie ovládání aplikace (někdy chyběla i možnost použití předpřipravené sady dat pro rychlé vygenerování animované struktury). Tyto poznatky byly následně využity při vlastní implementaci vizualizací zadaných datových struktur. Bylo také nutno dodržet konvence rozmístění ovládacích prvků a ovládání aplikací jako takových v souladu s ostatními pracemi vznikajících na Univerzitě Pardubice, zabývajících se podobnými tématy.

Teoretická část práce se zabývala přiblížením daných implementací abstraktního datového typu tabulka, které jsou předmětem této práce. V této části jsou obsaženy použe obecné informace o těchto strukturách, případně jejich využití. Samotný popis principu jednotlivých algoritmů, jejich implementace je především obsažen v následující praktické části.

Implementační část práce nejdříve popisuje platformu a technologie zvolenou pro realizaci vizualizací a dat. struktur jako takových. Po představení dostupných prostředků k implementaci vizualizací byla představena JavaFX jako nejvhodnější platforma k tomuto účelu. Dále byly uvedeny možnosti samotné realizace – v tomto případě byl vybrán systém obsluhy událostí jako filozofie implementace asynchronního způsobu programování. Za použití zvolených technologií byly následně implementovány vizualizace evolucí alogirtmů zadaných datových struktur. Tato prakticky orientovaná část také zahrnuje popis jednotlivých algoritmů a způsob jejich implementace.

Výsledkem této práce je krom tohoto dokumentu také pětice javovských aplikací umožňující uživateli přehlednou orientaci v problematice určitých datových struktur. Nabízí možnost vybudování struktur z předpřipravených sad dat, možnost kdykoli animaci pozastavit, krokovat a v neposlední řadě i změnit směr animace.

Autor této práce měl příležitost se podrobněji seznámit s problematikou pokročilých datových struktur a jejich algoritmů. Při tvorbě samotných vizualizací nabyl některé nové zkušenosti, zejména ve smyslu asynchronního programování a poznání technologie JavaFX, která představuje vítané rozšíření samotného univerza platformy Java o užitečné knihovny pro práci s grafikou a animace.

Mimo těchto znalostí si autor osvojil některé techniky a metodiky nasazení (deploymentu) společně se zabezpečením výsledné aplikace (podepisování javovských archivů). I přes určité problémy související s omezováním podpory doplňků třetích stran v nejpoužívanějším prohlížeči v době psaní této práce[15] je JavaFX velice zajímavou technologií k implementaci aplikací zabývajících se grafikou či práci s multimédii jako takovými.

[15] http://www.w3counter.com/globalstats.php?year=2015&month=7

Literatura

1. **Pugh, William.** *Skip lists: a probabilistic alternative to balanced trees.* 1990. doi: 10.1145/78973.78977.

2. **Samet, Hanan.** *Foundations of multidimensional and metric data structures.* Boston : Elsevier/Morgan Kaufmann, 2006. 9780123694461.

3. **Mautner, Pavel.** ADT Tabulka. [Online] 2014. http://www.kiv.zcu.cz/~mautner/Pt/Pt4_ADT_tabulka.pdf.

4. **Samet, Hanan.** Storing a Collection of Polygons Using Quadtrees. *ACM Transactions on Graphics.* 1985, Sv. IV, 3.

5. **Finkel, Raphael a Bentley, Jon Louis.** Quad trees a data structure for retrieval on composite keys. *Acta Informatica.* 1974, Sv. IV, 1-9.

6. **Papadakis, Thomas.** *Skip Lists and Probabilistic Analysis of Algorithms.* Waterloo (Ontario) : University of Waterloo, 1993.

7. **Drakos, Nikos.** Chapter 4 - SkipLists. *Open Data Structures.* [Online] http://opendatastructures.org/versions/edition-0.1e/ods-java/4_Skiplists.html.

8. **Cormen, Thomas H.** *Introduction to algorithms 3rd ed.* Cambridge : MIT Press, 2009. ISBN 9780262533058.

9. **Lewis, Harry a Denenberg, Larry.** *Data structures & their algorithms.* New York, NY : HarperCollins Publishers, 1991. ISBN 067339736x.

10. **Goodrich, Michael a Tamassi, Roberto.** *Algorithm design: foundations, analysis, and Internet examples.* New York, NY : Wiley, 2002. ISBN 0471383651-.

11. **Oracle.** JavaFX 2 Documentation. [Online] http://docs.oracle.com/javafx/2/.

12. **Wähner, Kai.** When to use JavaFX 2 instead of HTML5 for a Rich Internet Application. [Online] 2012. Dostupné z: https://dzone.com/articles/when-use-.

13. **Oracle.** JavaFX FAQ. [Online] http://www.oracle.com/technetwork/java/javafx/overview/faq-1446554.html.

14. **Google.** Dart. [Online] https://www.dartlang.org/.

15. **Dea, Carl a Coffin, David.** *JavaFX 2.0 introduction by example.* New York, NY : Apress, 2011.

Příloha A – UML diagram tříd Hierarchie seznamů

Zdroj: vlastní

Příloha B – Obsah přiloženého CD

- Zdrojové kódy k implementovaným strukturám
    - BST – binární vyhledávací strom – projekt Eclipse
    - AVL – AVL strom – projekt Eclipse
    - QuadTree – quad strom – projekt Eclipse
    - 23tree – 2-3 strom – projekt Eclipse
    - SkipList – projekt Netbeans
    - Guiprovider – projekt zajišťující jednotné GUI – projekt Eclipse
    - MT-common – projekt zajišťující společnou grafickou funkcionalitu
    - MT-common/lib/guava-18.0.jar – knihovna Guava obsahující využívanou komponentu EventBus.

Všechny projekty struktur v Eclipse vyžadují závislé projekty resp. knihovny Guiprovider, MT-common a Guava. Projekt Skip-Listu v Netbeans je spustitelný samostatně bez jakýchkoliv závislostí.

- Vizualizace datových struktur – soubory .jar a .jnlp
- KrejcirV_VizualizaceEvoluce_AK_2015.pdf – tento dokument.