Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Vizualizace evoluce algoritmů datových struktur uchovávajících bodová multi-dimenzionální data

Bc. Miloš Samek

Diplomová práce

2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2017/2018

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Miloš Samek**
Osobní číslo:	**I16238**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Vizualizace evoluce algoritmů datových struktur uchovávajících bodová multidimenzionální data**
Zadávající katedra:	**Katedra softwarových technologií**

Zásady pro vypracování:

V úvodní části práce je nutné provést přehled problematiky vybraných implementací abstraktního datového typu tabulka a principů uchovávání bodových multidimenzionálních dat. Primárním cílem diplomové práce je realizace vizualizací evolucí vybraných algoritmů nad následujícími datovými strukturami: rozsahový strom (range tree), quad strom (quad tree) a prioritní vyhledávací strom (priority search tree). Dalším cílem je implementace vizualizací různých typů lineárních průchodů (one-dimensional ordering) prostorem - Hilbertova křivka, Z-křivka, Peanova křivka.
Zmíněné vizualizace budou realizovány v rámci webové aplikace.

Rozsah grafických prací:	**10**
Rozsah pracovní zprávy:	**60**
Forma zpracování diplomové práce:	**tištěná**

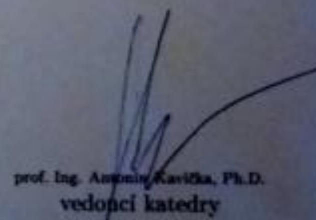
Seznam odborné literatury:

SAMET, Hanan. Foundations of multidimensional and metric data structures. San Francisco: Morgan Kaufmann, 2006, xxvii, 993 s. ISBN 978-012-3694-461.
CORMEN, Thomas H. Introduction to algorithms. 3rd ed. Cambridge: MIT Press, c2009, xix, 1292 s. ISBN 978-0-262-03384-8.
LEWIS, Harry R a Larry DENENBERG. Data structures. 1997. vyd. New York, NY: HarperCollins Publishers, c1991, xv, 509 p. ISBN 06-733-9736-X.
GOODRICH, Michael T a Roberto TAMASSIA. Algorithm design: foundations, analysis, and Internet examples. 2002. vyd. New York: Wiley, c2002, xii, 708 p. ISBN 04-713-8365-1.

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

Datum zadání diplomové práce:	**30. října 2017**
Termín odevzdání diplomové práce:	**18. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan



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

V Pardubicích dne 15. listopadu 2017

Prohlašuji:

Tuto práci jsem 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.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 11. 5. 2018

Bc. Miloš Samek

PODĚKOVÁNÍ

Tímto bych rád poděkoval prof. Ing. Antonínu Kavičkovi, PhD. za poskytnutí cenných rad a připomínek v průběhu vypracování této práce. Také bych chtěl poděkovat svým rodičům za podporu, kterou mi během mého studia věnovali.

ANOTACE

Diplomová práce se zabývá tématem vizualizace datových struktur uchovávající multidimenzionální bodová data a jejich operací. Konkrétně se jedná o vizualizace prioritního vyhledávacího stromu, rozsahového stromu a dvou typů quad stromu. Sekundárním cílem bylo zpracování vizualizací lineárních průchodů prostorem, a to Peanovy křivky, Hilbertovy křivky a Z-křivky.

KLÍČOVÁ SLOVA

Datové struktury, vizualizace, rozsahový strom, quad strom, prioritní vyhledávácí strom, lineární průchody prostorem, Z-křivka, Peanova křivka, Hilbertova křivka

TITLE

Visualization of algorithms evolution of data structures storing multidimensional point data

ANNOTATION

The primary topic of this thesis is visualization of data structures used to store multidimensional point data and their operations. Specifically, this thesis covers visualisations of the priority search tree, range tree and two quad tree specifications. The secondary goal was to create visualisations of linear orderings of Peano curve, Hilbert curve and Z-curve.

KEYWORDS

Data structures, visualization, range tree, quad tree, priority search tree, Z-curve, linear ordering, Peano curve, Hilbert curve

OBSAH

Seznam obrázků ... **9**

Seznam tabulek ... **11**

Seznam zkratek ... **12**

Úvod ... **13**

1 Datové struktury pro uchovávání multidimenzioanlních dat ... **14**

1.1 Typy multidimenzionálních dotazů ... 14

1.2 Výpočetní složitost operací ... 15

2 Abstraktní datový typ tabulka ... **16**

2.1 Operace ... 17

2.2 Výpočetní složitosti implementací ... 18

3 Vybrané datové struktury ... **19**

3.1 Struktura pseudokódu ... 19

3.2 Datová sada ... 20

3.3 Prioritní vyhledávací strom ... 20

3.3.1 Operace vybuduj ... 21

3.3.2 Operace vlož ... 24

3.3.3 Operace najdi ... 27

3.3.4 Operace odeber ... 28

3.4 Rozsahový strom ... 32

3.4.1 Operace vybuduj ... 32

3.4.2 Operace vlož ... 35

3.4.3 Operace najdi ... 37

3.4.4 Operace odeber ... 39

3.5 Quad Strom ... 42

3.5.1 Typy implementací ... 42

3.5.2 Využití v praxi ... 42

3.5.3 Bodový Quad Strom ... 43

3.5.3.1 Operace vybuduj ... 43

3.5.3.2 Operace vlož ... 45

3.5.3.3 Operace najdi ... 47
3.5.3.4 Operace odeber ... 48
3.5.4 Bodově oblastní Quad strom ... 48
3.5.4.1 Operace vybuduj ... 49
3.5.4.2 Operace vlož ... 50
3.5.4.3 Operace najdi ... 52
3.5.4.4 Operace odeber ... 53

4 Linearní průchody prostorem ... 56

4.1 Peanova křivka ... 56

4.2 Z-křivka ... 57

4.3 Hilbertova křivka ... 58

5 Implementace ... 59

5.1 Použité technologie ... 59

5.2 Uživatelské prostředí ... 60

5.3 Databáze ... 62

5.4 Návrh tříd ... 63
5.4.1 Třídy pro prioritní vyhledávací strom ... 64
5.4.2 Třídy pro rozsahový strom ... 65
5.4.3 Třídy pro Bodový Quad strom ... 66
5.4.4 Třídy pro Bodově oblastní Quad Strom ... 67
5.4.5 Třídy pro lineární průchody prostorem ... 68
5.4.6 Třídy pro komunikaci mezi backendem a frontendem ... 68

Závěr ... 70

Použitá literatura ... 71

Přílohy ... 73

Příloha A – Peanova křivka ... 74

Příloha B – Z-křivka ... 75

Příloha C – Hilbertova křivka ... 76

Příloha D – CD s vytvořenou aplikací ... 77

Příloha E – Uživatelská příručka ... 78

SEZNAM OBRÁZKŮ

Obrázek 1 Základní typy dotazů, zdroj: [vlastní]..15
Obrázek 2 Sada dat, zdroj: [vlastní]..20
Obrázek 3 Volba kořenu PST, zdroj: [vlastní]..21
Obrázek 4 Dělení vstupních dat pomocí mediánu – PST, zdroj: [vlastní]............................22
Obrázek 5 Vybudovaná struktura – PST, zdroj: [vlastní]..22
Obrázek 6 Začátek Operace vlož s označeným místem – PST, zdroj: [vlastní]....................24
Obrázek 7 Vložený prvek – PST, zdroj: [vlastní]..25
Obrázek 8 Stav struktury po vložení prvku – PST, zdroj: [vlastní]....................................25
Obrázek 9 Operace Najdi – PST, zdroj: [vlastní]..27
Obrázek 10 Operace Odeber PST – původní stav stromu, zdroj: [vlastní]..........................29
Obrázek 11 Operace Odeber PST – stav po levé rotaci zdroj: [vlastní]..............................29
Obrázek 12 Operace Odeber PST – stav po druhé levé rotaci, zdroj: [vlastní]....................29
Obrázek 13 Operace Odeber PST – stav po pravé rotaci, zdroj: [vlastní]..........................30
Obrázek 14 Vybudování hlavní navigační struktury rozsahového stromu, zdroj: [vlastní] 33
Obrázek 15 Vybudovaný rozsahový strom, zdroj: [vlastní]..34
Obrázek 16 První krok operace vlož na rozsahovém stromu, zdroj: [vlastní]......................35
Obrázek 17 Rozsahový strom po vložení prvku, zdroj: [vlastní]......................................36
Obrázek 18 Operace najdi v rozsahovém stromu, zdroj: [vlastní]......................................38
Obrázek 19 Operace odeber Rozsahový strom – označen prvek pro odebraní, zdroj: [vlastní] 39
Obrázek 20 Stav Rozsahového stromu po odebrání prvku, zdroj: [vlastní]........................40
Obrázek 21 Ukázka vybudování prvního patra Bodového Quad Stromu, zdroj: [vlastní]..43
Obrázek 22 Dělení 2D prostoru v Bodovém Quad Stromu, zdroj: [vlastní]........................44
Obrázek 23 Vybudovaný Bodový Quad Strom, zdroj: [vlastní]..44
Obrázek 24 Vložení prvku do Bodového Quad stromu, zdroj: [vlastní]............................45
Obrázek 25 Hledání prvku v bodovém Quad Stromu, zdroj: [vlastní]..............................47
Obrázek 26 Dělení 2D prostoru pomocí půlení intervalů, zdroj: [vlastní]..........................49
Obrázek 27 Budování struktury Bodově oblastního Quad stromu, zdroj: [vlastní]............49
Obrázek 28 Vybudovaná struktura Bodově oblastního Quad Stromu, zdroj: [vlastní].......50
Obrázek 29 Stav bodově oblastního stromu po vložení prvku, zdroj: [vlastní]...................51
Obrázek 30 Znázornění operace najdi v bodově oblastním quad stromu, zdroj: [vlastní]..53

Obrázek 31 Označení odebíraného prvku v Bodově oblastním Quad stromu, zdroj: [vlastní] 54
Obrázek 32 Stav Bodově oblastního stromu po odebrání prvku, zdroj: [vlastní] 54
Obrázek 33 První dvě iterace Peanovy křivky, zdroj: [11] 57
Obrázek 34 První dvě iterace Z-křivky, zdroj: [vlastní] 58
Obrázek 35 První dvě iterace Hilbertovy křivky, zdroj: [11] 58
Obrázek 36 Uživatelské prostředí, zdroj: [vlastní] 60
Obrázek 37 Kreslící plátno, zdroj: [vlastní] 61
Obrázek 38 Návrh databáze, zdroj: [vlastní] 62
Obrázek 39 Rozhraní pro datové struktury, zdroj: [vlastní] 63
Obrázek 40 Třídy pro PST, zdroj: [vlastní] 64
Obrázek 41 Třída pro rozsahový strom, zdroj: [vlastní] 65
Obrázek 42 Třída pro bodový quad strom, zdroj: [vlastní] 66
Obrázek 43 Třída pro bodově oblastní Quad strom, zdroj: [vlastní] 67
Obrázek 44 Třídy pro linearní průchody prostorem, zdroj: [vlastní] 68
Obrázek 45 Ukázka komunikace backend-frontend, zdroj: [vlastní] 69
Obrázek 46 Uživatelská příručka – úvodní obrazovka 78
Obrázek 47 Uživatelská příručka – detail vykonané operace 79
Obrázek 48 Uživatelská příručka – kreslící plátno PST 79
Obrázek 49 Uživatelská příručka – PST s operací najdi 80
Obrázek 50 Uživatelská příručka – kontextové menu 80
Obrázek 51 Uživatelská příručka – budování struktury z datové sady 81
Obrázek 52 Uživatelská příručka – operace nad datovou strukturou 81
Obrázek 53 Uživatelská příručka – vytvoření prvku 82
Obrázek 54 Uživatelská příručka – vykreslený průchod 82
Obrázek 55 Uživatelská příručka – volba možnosti průchodu 83
Obrázek 56 Uživatelská příručka – vybraný bod 83

SEZNAM TABULEK

Tabulka 1 Asymptotická složitost operací datových struktur [zdroj: vlastní] 15

Tabulka 2 Operace nad ADT [zdroj: vlastní]..17

SEZNAM ZKRATEK

ADT Abstraktní datový typ

PST Prioritní vyhledávací strom

UID Unikátní identifikátor

UML Unified modeling Language

PAAS Platforma jako služba

ÚVOD

Při uchovávání dat v programování se bez vhodných datových struktur a jejich znalosti neobejdeme. Je vhodné podporovat povědomí o pokročilejších datových strukturách, které obvykle ukládají data efektivněji. Vizualizované operace nad datovými strukturami pomáhají snadnějšímu pochopení problematiky hierarchických stromových struktur.

Primární cíl diplomové práce byl stanoven realizací vizualizací evolucí vybraných algoritmů nad datovými strukturami uchovávající multidimenzionální bodová data. Multidimenzionální bodová data mohou představovat geografická data, která tvoří hodnoty souřadnic pro zeměpisnou šířku a zeměpisnou délku. Postupně budou představeny datové struktury: prioritní vyhledávací strom, rozsahový strom a quad strom, který bude charakterizován ve dvou typech. První typ bude bodový quad strom, který je označován jako obdoba binárního vyhledávacího stromu pro multidimenzionální data. Druhým typem bude bodově oblastní quad strom, který se vyznačuje datovými prvky pouze na pozici listů.

Lineární průchody prostorem slouží k zajištění efektivního průchodu vyznačeného prostoru. Využívány jsou v mnoha oblastech informatiky, přičemž v oblasti datových struktur se používají při zefektivňování vybudování struktury ze sady předem známých dat. V rámci diplomové práce budou popsány tři typy průchodů v následujícím pořadí: Peanova křivka, Hilbertova křivka a Z-křivka.

Praktickou částí diplomové práce bylo vytvoření webové aplikace, která obsahuje vzorové implementace výše zmíněných struktur a lineárních průchodů prostorem. Webová aplikace bude zpracována pomocí nejvyužívanějších java frameworků v praxi.

1 DATOVÉ STRUKTURY PRO UCHOVÁVÁNÍ MULTIDIMENZIOANLNÍCH DAT

Výrazem multidimenzionální data, jsou obecně myšlena data, která obsahují jednoznačné informace o objektu, jeho rozměr či jeho umístění v prostoru. Tyto informace jsou většinou reprezentovány vektory a objekty multidimenzionálních dat ležících v k-dimenzionálním prostoru, kde ***k*** je přirozené číslo určující počet rozměrů.

Obvykle jsou multidimenzionální data rozdělována do dvou základních kategorií. První kategorie jsou data bodová. Tyto data například leží ve dvourozměrném dimenzionálním prostoru a mohou označovat geografické souřadnice. Druhou kategorii jsou data objektová, která mají nějaký tvar a velikost jako například čáry, trojúhelníky a jiné [1].

Datové struktury pro uchovávání multidimenzionálních dat jsou nástrojem podporující vyhledávání a úpravu (vkládání, změnu, vymazání) v multidimenzionálních datech.

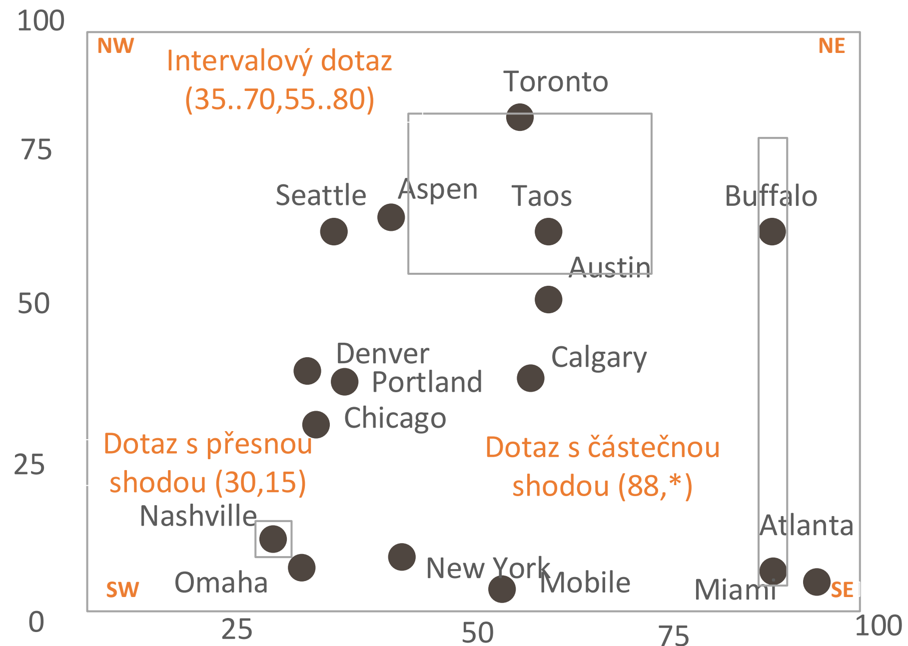
V některých literaturách bývají datové struktury pro uchovávání multidimenzionálních dat označovány jako multidimenzionální přístupové metody, prostorové přístupové metody nebo také prostorové indexové struktury.

1.1 Typy multidimenzionálních dotazů

Při aplikaci vyhledávacích dotazů v datových strukturách uchovávající multidimenzionální data je pracováno s různými vstupními daty a taktéž s různými předpoklady výsledného dotazování. Tři základní typy dotazů jsou graficky zpracovány na Obrázku 1. Přehled dotazů nad datovými struktury:

- Dotaz s přesnou shodou: Všechny dimenze klíčového atributu musí platit.
- Dotaz s částečnou shodou: Pouze ***t*** z ***k*** dimenzí klíčových atributů musí platit, kde ***k*** je počet všech dimenzí a ***t*** počet dimenzí platných. Ostatní dimenze mohou mít libovolné hodnoty.
- Intervalový dotaz: Pro každou dimenzi je určen interval platných hodnot.
- Dotaz s nejlepší shodou: Nalezne nejbližšího souseda bodu, nebo oblasti specifikovaného podmínkami dotazu.

- Hledání k nejbližších sousedů: Generalizace dotazu s nejlepší shodou, přičemž hledá ***k*** výsledků, kde k je maximální počet výsledků.



Obrázek 1 Základní typy dotazů, zdroj: [vlastní]

1.2 Výpočetní složitost operací

V Tabulce 1 můžeme naleznout porovnání multidimenzionálních struktur vůči výpočetní složitosti jejich hlavních operací. Vybrané datové struktury byly převzaty z literárního zdroje [1]. Všechny vybrané hierarchické stromové datové struktury jsou ukládány pouze operační paměti. Některé z uvedených datových struktur budou dále přiblíženy v následujících kapitolách diplomové práce.

Strom	Vyhledávání	Vkládání	Odebírání
Rozsahový	O ($\log^2 n + k$)	O (n*log n)	O (n*log n)
Prioritní vyhledávací	O ($n * \log_2 n + k$)	O ($\log_2 n$)	O ($\log_2 n$)
Bodový Quad	O ($2 * n^{1/2}$)	O ($\log_4 n$)	O ($\log_4 n$)
Bodově oblastní Quad	O(n)	O ($\log_2 n$)	O ($\log_2 n$)
K-D	O ($\log_2 n + k$)	O ($\log_2 n$)	O ($\log_2 n$)

Tabulka 1 Asymptotická složitost operací datových struktur, zdroj: [vlastní]

2 ABSTRAKTNÍ DATOVÝ TYP TABULKA

Definicí abstraktního datového typu tabulka můžeme označit jako „množina s lineárním uspořádáním, přičemž uspořádání je určováno jednoznačnými klíčovými hodnotami (klíči) prvků“. Z toho vyplývá, že v datových strukturách implementující ADT tabulku přistupujeme k datům pomocí jednoznačně definovaného klíče, na rozdíl od ADT pole, kde k prvkům přistupujeme přes celočíselný index [2].

Jednoznačně definovaný klíč může představovat libovolný datový typ. Klíče mohou být i vícerozměrné(multidimenzionální), kde se k datům přistupuje pomocí složeného klíče. Například v tabulkách určených pomocí geografické lokace, lze přistupovat pomocí souřadnic x, y [3].

Existuje celá řada implementací ADT tabulky, liší se zejména výpočetní složitostí prováděných operacích. Některé implementace jsou vhodné pro časté vkládání, některé pro časté hledání. Ukládané informace mohou být velmi rozsáhlé přičemž, jejich stále udržování v operační paměti není efektivní, občas dokonce i nemožné. Z toho důvodu jsou některé implementace vhodné pro uchování dat na externích paměťových mediích, jiné jsou vhodné pro uchovávání dat v operační paměti. Níže jsou zmíněné některé z implementací:

- tabulka na poli,
- tabulka na seznamu,
- tabulka na binárním vyhledávacím stromu,
- rozptýlená tabulka (hashovací),
- a jiné.

2.1 Operace

V následující podkapitole budou ukázány operace nad abstraktním datovým typem tabulkou. V Tabulce 2 jsou vypsány dostupné operace, jejichž funkčnost a parametry jsou charakterizovány níže. Operace, jsou úkony, které lze v rámci dané datové struktury nad daty provádět. Pro každou operaci lze vypočítat asymptotickou složitost udávající rychlost, s kterou se daná operace provede [2][3].

ADT Tabulka
A. Třída prvků s klíči B. Třída konečných tabulek
A. Vytvoř Zruš JePrázdná (↑Boolean) Mohutnost (↑PočetPrvků) Prohlídka (↓TypProhlídky, ↓Akce) Vlož (↓Prvek) Odeber (↓Klíč, ↑Prvek) Najdi (↓Klíč, ↑Prvek)
B. Sjednocení (↓TabulkaA, ↓TabulkaB, ↑TabulkaC)

Tabulka 2 Operace nad ADT, zdroj: [vlastní]

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

Operace ***jePrazdná*** testuje, zda tabulka obsahuje prvky. Pokud žádné prvky neobsahuje, operace vrací logickou nepravdu.

Operace ***mohutnost*** vrací aktuální počet prvků v tabulce.

Operace ***prohlídka*** prochází tabulku prvek po prvku dle zadaného typu prohlídky.

Operace ***vlož*** přidává do tabulky prvek P s klíčem K.

Operace ***odeber*** z tabulky odebírá prvek P dle klíče K.

Operace ***najdi*** dle zadaného klíče K, hledá prvek P a pokud je naleznut vrací jej.

2.2 Výpočetní složitosti implementací

Výpočetní složitost je způsob klasifikace počítačových algoritmů a určuje náročnost algoritmu tak, že zjišťuje, jakým způsobem se bude chování algoritmu měnit v závislosti na změně velikosti vstupních dat [3]. Výpočetní složitost, bude představena při operacích *Vlož*, *Odeber* a Najdi.

Tabulka na poli (homogenním souvislém)

Výpočetní složitosti vybraných operací na utříděném poli se liší, podle toho, zda je pole utříděné nebo neutříděné. Utříděné pole je vhodné využít při častém vykonání operace *Najdi* s výpočetní složitostí $O(\log_2 n)$, přičemž operace *Vlož* a *Odeber* mají lineární výpočetní složitost O(n). Naopak implementace na neutříděném poli je vhodné použít při častém vkládání prvků do struktury se složitostí O(1), přičemž operace *Vlož* a *Odeber* disponují lineární výpočetní složitostí O(n).

Tabulka na seznamu

Stejně jako implementace na poli je vhodnost využít implementace rozdílná podle intenzity použití hlavních operací. Implementace na neutříděném seznamu je vhodná při mnohokrát opakované operaci *Vlož* s výpočetní složitostí O(1), kde ostatní operace mají složitost lineární O(n). Implementace na utříděném seznamu, má u všech hlavních operací stejnou výpočetní složitost, a to lineární O(n).

Tabulka na binárním vyhledávacím stromu

Při implementaci tabulky na binárním vyhledávacím stromu, zpracovává struktura operace *Vlož*, *Odeber* a *Najdi* s logaritmickou výpočetní složitostí $O(\log_2 n)$, ale pouze s dodržením podmínky nezdegenerovaného stromu. Při nedodržení podmínky degenerace se výpočetní složitosti posouvají k výpočetně horšímu spojovému seznamu.

Rozptýlená tabulka (hashovací)

Rozptýlená tabulka přináší při hlavních operacích *Vlož*, *Odeber* a *Najdi* nejoptimálnější asymptotickou výpočetní složitost O(1). Nicméně, výpočetní složitost nemusí platit, pokud by byla nevhodně zvolená rozptylová funkce [2][3].

3 VYBRANÉ DATOVÉ STRUKTURY

V této kapitole bude proveden teoretický rozbor vybraných datových struktur. Jako první bude vždy vybraná datová struktura představena a poté ukázány základní operace konstrukce, přidání prvku, vyhledání a odebrání prvku.

Při zpracování operací bude pro znázornění použito nejprve slovní vyjádření operace, následně vizuální představení operace, které bude doplněno pseudokódem.

3.1 Struktura pseudokódu

Pseudokód, který bude využíván k lepšímu pochopení představení operací bude založen na tzv. „českém pascalu". Stejně jako jazyk pascal není pseudokód závislý na velikosti písmen. Uvození bloku příkazu bude ohraničeno klíčovými slovy **začátek** a **konec**. Přiřazení je stejně jako v pascalu pomocí dvojznaku „**:=**", kde vlevo od dvojznaku stojí vždy název proměnné, do které bude hodnota přiřazena. Větvení pro omezení vykonaní určitého příkazu či bloku příkazů bude označeno klíčovým slovem **jestliže**. Příkaz větvení může tedy vypadat takto:

jestliže x > 1 **pak** příkaz;

Podmínka ve větvení, může být skládána pomocí logických operátorů, tudíž je nezbytné, aby výsledkem byla jednoznačně reprezentovaná hodnota *logické pravdy* / *logické nepravdy*. Do úplného větvení lze použít klíčového slova **jinak**.

Cykly rozdělujeme do dvou typů. První typ je cyklus s označením pomocí klíčového slova **pro**.

pro i **od** 1 **do** 99 **opakuj** příkaz;

Cyklus pro je určen pro případy, kdy jsou použity příkazy ve známém počtu opakování. Druhý obecnější příkaz je označen klíčovým slovem **dokud**. Příkaz, **dokud** obsahuje vstupní parametr logického výrazu, který se před každým zopakováním cyklu otestuje a pokud je výsledkem *logická pravda,* cyklus je znovu opakován.

dokud x > 0 **opakuj** příkaz;

Poslední nedílnou součásti je ternární operátor, který se nejčastější používá pro zkrácení zápisu vyhodnocení podmínky.

proměnná := **podmínka** ? výraz1 : výraz2;

3.2 Datová sada

Pro grafické zpracování operací byla navržena sada o 16 prvcích fiktivních dat. Data označují Americká města a klíče jsou smyšlené zeměpisné šířky a délky. Sada je graficky zpracována na Obrázku 2.

Toronto 57 77	Denver 31 45	Buffalo 84 65	Omaha 35 16
Chicago 35 42	Atlanta 85 15	Miami 90 5	Mobile 52 10
Calgary 55 42	New York 41 17	Seattle 38 57	Portland 33 44
Aspen 40 60	Nashville 30 20	Austin 60 50	Taos 65 64

Obrázek 2 Sada dat, zdroj: [vlastní]

3.3 Prioritní vyhledávací strom

Prioritní vyhledávací strom je datová struktura, která slouží k uchovávání dvourozměrných dat. Struktura vychází z kombinace binárního vyhledávacího stromu a binární haldy. Navrhnutá struktura slouží pro řešení problémů s částečně nekonečnými rozsahy ve dvourozměrném prostoru. Cílem navrhnuté struktury, bylo snížení vyhledávacího času O(n) na O(s + log n), kde *n* je počet prvků ve stromu a *s* je počet naleznutých prvků [1][6].

První souřadnici (X – zeměpisnou šířku) dělíme podle pravidel binárního vyhledávacího stromu, pro kterou byla zavedena pomocná hranice. Ta nám slouží k vybudování vyvážené struktury pro známá vstupní data. V případě nezavedení hranice může nastat situace, kdy prvek s nejvyšší Y souřadnicí, který se umístí do kořene stromu, může být zároveň prvkem s nejvyšší nebo nejnižší X souřadnicí. Druhá souřadnice (Y – zeměpisnou délku) je organizována pomocí pravidel binární haldy (konkrétně podle pravidel tzv. max-haldy). Pravidla, která platí v prioritním vyhledávacím stromu:

- každý prvek má maximálně dva syny,

- levý podstrom uzlu obsahuje pouze klíče X-ové souřadnice menší, než je klíč mediánu uchovaný v tomto uzlu,
- pravý podstrom uzlu obsahuje pouze klíče X-ové souřadnice větší nebo rovné klíči mediánu tohoto uzlu,
- klíč Y-ové souřadnice uzlu je vždy větší nebo roven klíči jeho podstromů,
- hranice je tvořena pomocí prvku, který je mediánem zpracovávaných dat [6][7].

3.3.1 Operace vybuduj

Operaci *Vybuduj*, lze provést pomocí sady předem známých vstupních dat. Na rozdíl od opakovaného zpracování operace *Vlož* zajišťuje operace po sestrojení vyváženou stromovou strukturu. Při budování datové struktury budou dodržována pravidla max-haldy a prioritního vyhledávacího stromu.

Nejprve je vstupní sada dat seřazena pomocí zeměpisné délky a prvek s nejvyšší hodnotou zeměpisné délky, se stává kořenem stromu. Následně jsou data znovu seřazena, tentokrát pomocí zeměpisné šířky prvků. Poté je nalezen medián, který označuje hraniční hodnotu kořene stromu a rozděluje vstupní data do dvou skupin. První skupina, která leží před mediánem je vložena do levého podstromu, poté je druhá skupina vložena do pravého podstromu. Předchozí postup pro oba podstromy je cyklicky opakován do doby, než v každé skupině zůstane pouze jeden prvek. Jakmile je ve skupinách maximálně jeden prvek, výsledný prvek se stává listem prioritního vyhledávacího stromu.

Graficky postup operace je znázorněn na následujících třech obrazcích.

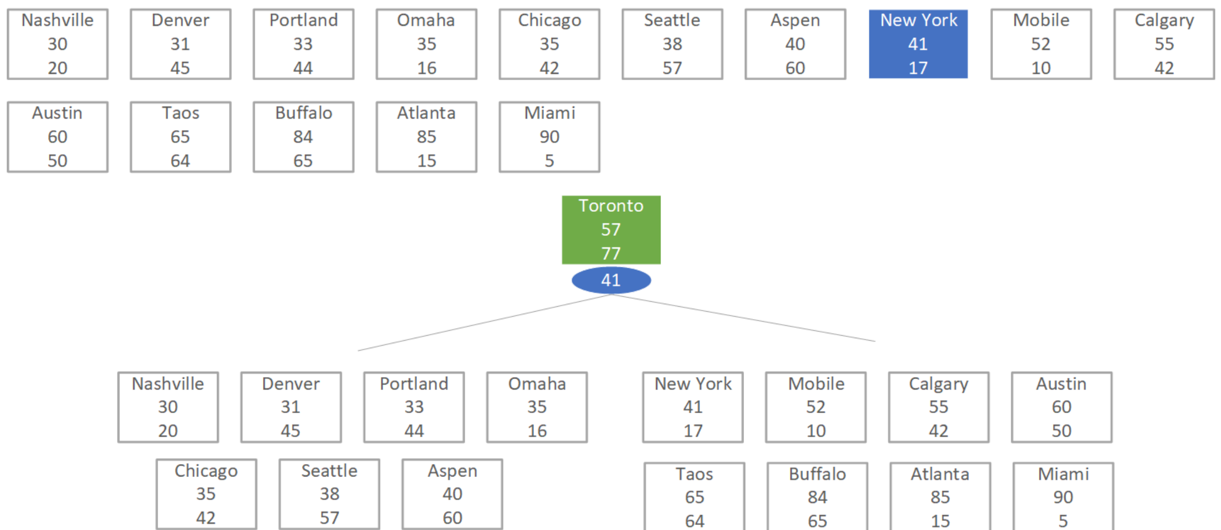
Miami 90 5	Mobile 52 10	Atlanta 85 15	Omaha 35 16	New York 41 17	Nashville 30 20	Chicago 35 42	Calgary 55 42	Portland 33 44	Denver 31 45
Austin 60 50	Seattle 38 57	Aspen 40 60	Taos 65 64	Buffalo 84 65	Toronto 57 77				

Toronto 57 77

Obrázek 3 Volba kořenu PST, zdroj: [vlastní]

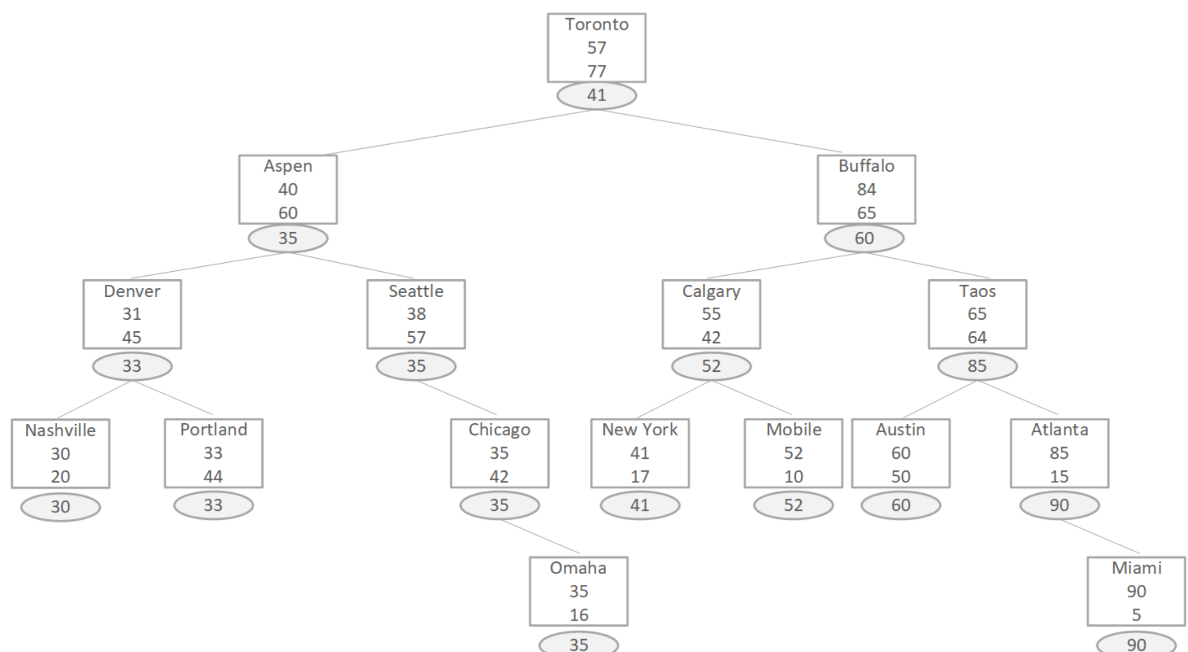
V prvním kroku byl zvolen kořen stromu Toronto s nejvyšší zeměpisnou délkou (Obrázek 3), která činí hodnotu 77. Ve druhém kroku, byly prvky seřazeny pomocí hodnot zeměpisné šířky a data byla rozdělena do dvou skupin pomocí mediánu (Obrázek 4), který je v modelové situaci

prvek New York. Pro vybudování vyvážené stromové struktury byla kořenu zvolena hranice mediánu s hodnotou 41.



Obrázek 4 Dělení vstupních dat pomocí mediánu – PST, zdroj: [vlastní]

Postup dělení prvků do skupin je v následně cyklicky opakován do okamžiku, než v každé skupině zbyde pouze jeden prvek. Poslední prvek ze skupiny se stává listem stromu. Vybudovaná struktura z ukázané sady dat je představena na Obrázku 5.



Obrázek 5 Vybudovaná struktura – PST, zdroj: [vlastní]

Pseudokód vybudování struktury

```
1.  Vybuduj (vstupniData)
2.  zacatek
3.    jestlize !jePrazdny() OR delka(vstupniData) == 0 pak konecOperace;
4.    pocetPrvku := delka(vstupniData);
5.
6.    //volba korene stromu
7.    vstupniData := seradData(vstupniData, podleY);
8.    Koren := vstupniData[delka(vstupniData) - 1];
9.    odeberPrvek(vstupniData, delka(vstupniData) - 1);
10.
11.   //volba hranice
12.   koren.hranice := delka(vstupniData) == 0 ? koren.X : vstupniData[median].X));
13.
14.   //vybudovani leveho a praveho podstromu
15.   vstupniData := seradData(vstupniData, podleX);
16.
17.   vybudujPodstrom(vstupniData[0..median - 1], root, TypPrvku.LevyPotomek, 1);
18.   vybudujPodstrom(vstupniData[median..delka(vstupniData) - 1], root,
19.                   TypPrvku.PravyPotomek, 1);
20.  konec
```

V operaci *vybudujPodstrom*, je nejprve vybrán poslední prvek s nejvyšší zeměpisnou délkou a poté jsou pomocí mediánu vstupní data rozdělena a rekurzivně zpracována do levého a pravého podstromu.

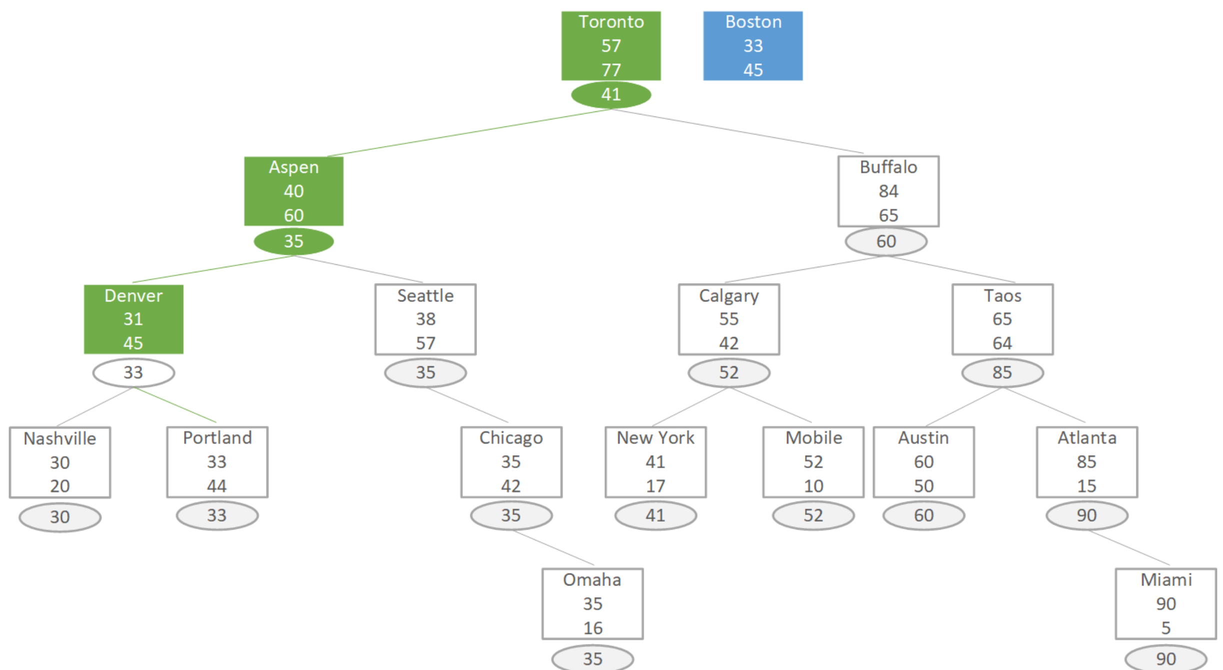
```
1.  vybudujPodstrom (vstupniData, otecPrvku, typPrvku, uroven)
2.  zacatek
3.    jestlize delka(vstupniData) == 0 pak konecOperace;
4.    vstupniData := seradData(vstupniData, podleY);
5.    zpracovavanyPrvek := null;
6.    jestlize typPrvku == levySyn pak
7.      zacatek
8.        levySyn := vstupniData[delka(vstupni  Data) - 1];
9.        levySyn.level := uroven;
10.       levySyn.otecPrvku := otecPrvku;
11.       otecPrvku.levySyn := levySyn;
12.       odeberPrvek(vstupniData, delka(vstupniData) - 1);
13.       zpracovavanyPrvek := levySyn;
14.     konec
15.     jinak
16.     zacatek
17.       pravySyn := vstupniData[delka(vstupniData) - 1];
18.       pravySyn.level := uroven;
19.       pravySyn.otecPrvku := otecPrvku;
20.       otecPrvku.pravySyn := pravySyn;
21.       odeberPrvek(vstupniData, delka(vstupniData) - 1);
22.       zpracovavanyPrvek := pravySynSyn;
23.     konec
24.
25. //volba hranice
26.   zpracovanayPrvek.hranice := delka(vstupniData) == 0 ? zpracovanayPrvek.X :
27.   vstupníData[median].X);
28.
29. //tvoreni podstromu
30.   vstupniData: = seradData(vstupniData, podleX);
31.
32.   vybudujPodstrom(vstupniData[0..median - 1], zpracovavanyPrvek,
33.                   TypPrvku.LevyPotomek, uroven + 1);
34.
35.   vybudujPodstrom(vstupniData[median..delka(vstupniData) - 1],
36.                  zpracovavanyPrvek, TypPrvku.PravyPotomek, uroven + 1);
37. konec
```

3.3.2 Operace vlož

Při operaci *Vlož* je nejprve ověřeno, jestli stromová struktura není prázdná. Pokud je stromová struktura prázdná, vkládaný prvek se stává kořenem.

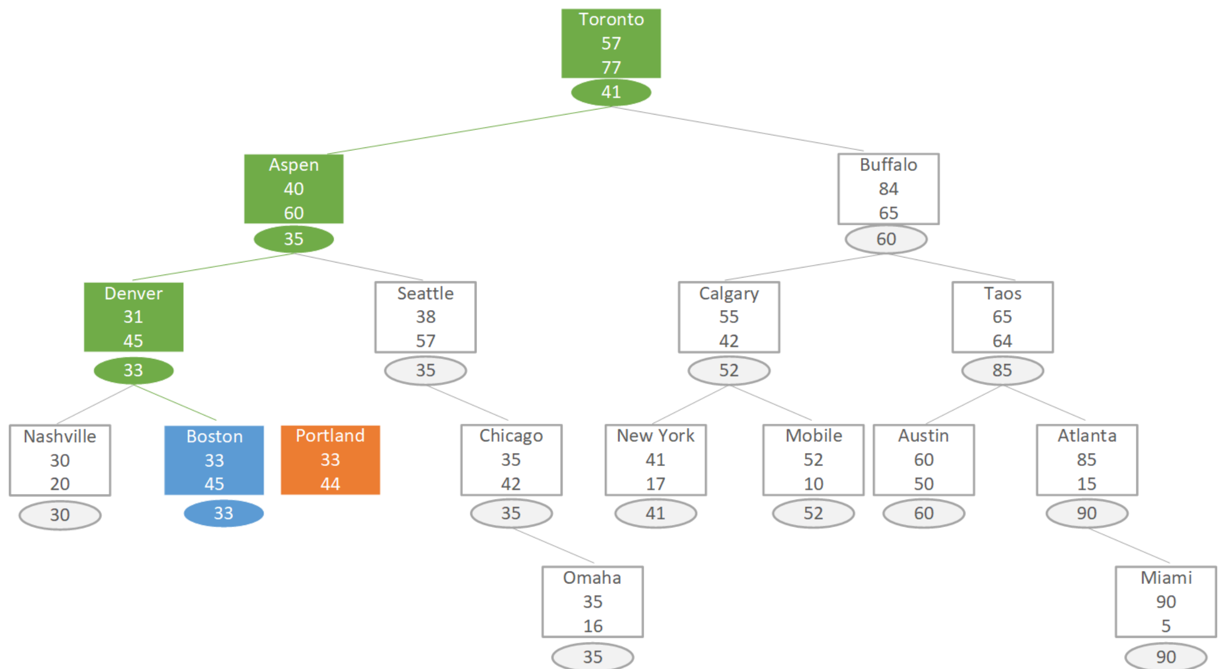
Při naplněné stromové struktuře je pro navigaci do levého či pravého podstromu postupováno pomocí pravidel binárního vyhledávacího stromu. Pro určení úrovně vkládaného prvku, jsou použita pravidla haldy. Algoritmus začíná pomocí traverzování prvků, do doby, než zeměpisná délka vkládaného prvku je větší než procházeného prvku, nebo pokud bylo naleznuto vhodné prázdné místo ve stromové struktuře. Pokud byl nalezen takový procházený prvek, který má menší zeměpisnou délku, tak na původní místo procházeného prvku, je vložen vkládaný prvek a algoritmus pokračuje s prvkem procházeným.

Na následujících třech obrazcích bude znázorněna operace *Vlož* pro prvek Boston s klíčem 33;45.



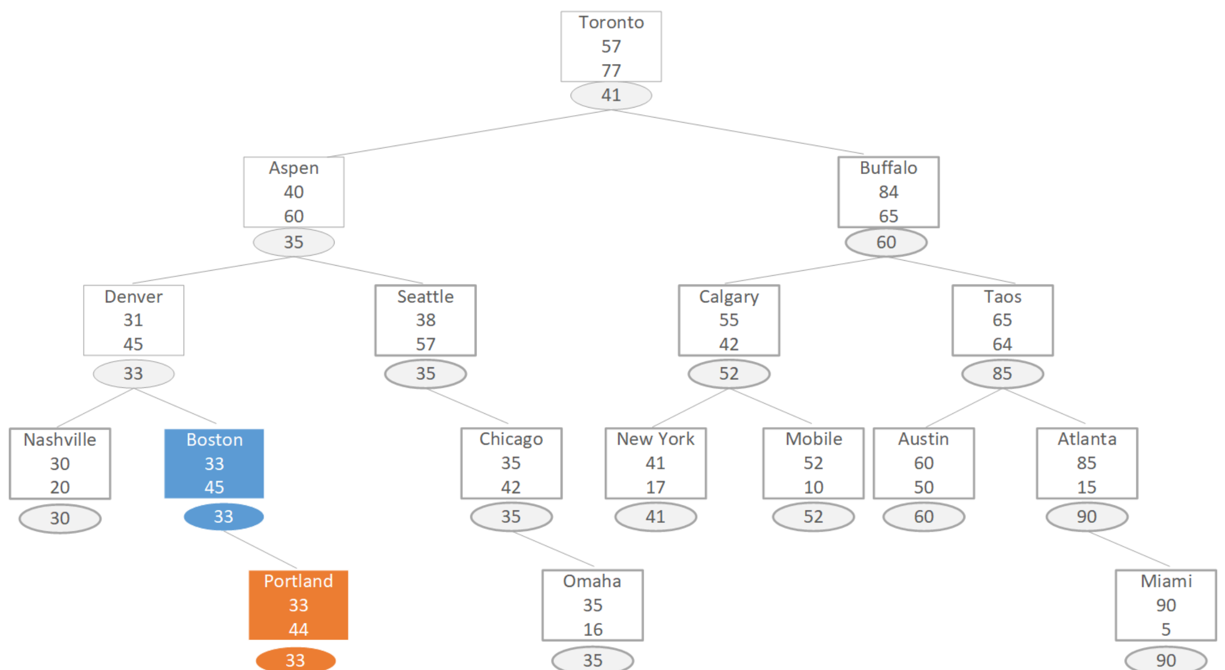
Obrázek 6 Začátek Operace vlož s označeným místem – PST, zdroj: [vlastní]

Na Obrázku 6 je zobrazeno traverzování prvku na správnou pozici ve struktuře. Podle podmínek haldového uspořádání prvek Portland již neodpovídá. Na následujícím Obrázku 7 je prvek Portland nahrazen vkládaným prvkem Boston.



Obrázek 7 Vložený prvek – PST, zdroj: [vlastní]

Nyní se prvek Boston nachází na správné pozici ve struktuře. Pokud by prvek Portland měl nějaké potomky, byly by přiřazeny vloženému prvku. Operace *Vlož* dále pokračuje s prvkem, který byl nahrazen ve struktuře (Obrázek 8).



Obrázek 8 Stav struktury po vložení prvku – PST, zdroj: [vlastní]

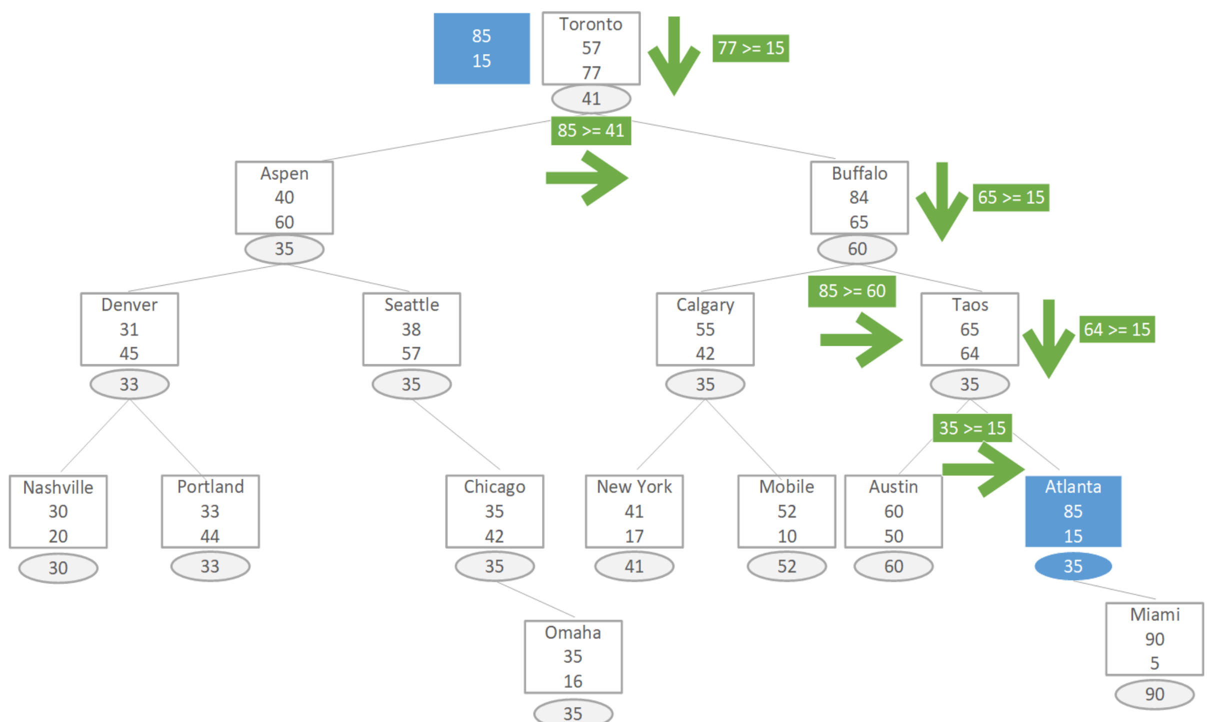
Pseudokód vložení prvku do struktury

```
1.  Vloz (vkladanyPrvek, aktualniPrvek, uroven)
2.  zacatek
3.    jestlize koren == null pak
4.      zacatek
5.        koren := vkladanyPrvek;
6.        pocetPrvku++;
7.        konecOperace koren;
8.      konec
9.
10. jestlize (vkladanyPrvek.Y > aktualniPrvek.Y) pak
11.   zacatek
12.   //prehozeni aktualne zpracovaneho prvku za novy prvek a aktualne zpracovany pr-
13.   vek bude vlozen jako syn vlozeneho prvku
14.     jestlize (aktualniPrvek.Otec.LevySyn != null AND
15.               aktualniPrvek.Otec.LevySyn == aktualniPrvek)
16.     zacatek
17.       aktualniPrvek.Otec.LevySyn := vkladanyPrvek;
18.       vkladanyPrvek.Otec := aktualniPrvek.Otec;
19.     konec
20.     jinak
21.     zacatek
22.       aktualniPrvek.Otec.PravySyn := vkladanyPrvek;
23.       vkladanyPrvek.Otec := aktualniPrvek.Otec;
24.     konec
25.     //pokud aktualne prehazovany prvek mel nektere potomky, zachováme vazby
26.     jestlize  (aktualniPrvek.LevySyn != null) pak
27.     zacatek
28.       vkladanyPrvek.LevySyn := aktualniPrvek.LevySyn;
29.       aktualniPrvek.LevySyn.Otec := vkladanyPrvek.LevySyn;
30.     konec
31.
32.     jestlize (aktualniPrvek.PravySyn != null) pak
33.     zacatek
34.       vkladanyPrvek.PravySyn := aktualniPrvek.PravySyn;
35.       aktualniPrvek.PravySyn.Otec := vkladanyPrvek.PravySyn;
36.     konec
37.     //vlozeni prehozene prvku jako syna vkladaneho prvku
38.     konecOperace vloz(aktualniPrvek, vkladanyPrvek, level + 1);
39.   konec
40.   jinak
41.   zacatek
42.     jestlize (vkladanyPrvek.X > aktualniPrvek.X) pak
43.     zacatek
44.      //traverzovani do leva
45.      jestlize (aktualniPrvek.LevySyn != null) pak
46.       konecOperace vloz (vkladanyPrvek, aktualniPrvek.LevySyn, uroven + 1);
47.
48.     //vytvoreni leveho potomka
49.     aktualniPrvek.LevySyn := vkladanyPrvek;
50.     vkladanyPrvek.Otec := aktualniPrvek;
51.     konecOperace;
52.     konec
53.     jinak
54.     zacatek
55.       //traverzovani do prava
56.       jestlize (aktualniPrvek.PravySyn != null) pak
57.         konecOperace vloz(vkladanyPrvek, aktualniPrvek.PravySyn, uroven + 1);
58.       //vytvoreni praveho potomka
59.       aktualniPrvek.PravySyn := vkladanyPrvek;
60.       vkladanyPrvek.Otec := aktualniPrvek;
61.       konecOperace;
62.     konec
63.   konec
64. konec
```

3.3.3 Operace najdi

V operaci *Najdi* podobně jako v operaci *Vlož* je traverzováno od kořene stromu, až po hledaný prvek. Při traverzování jsou porovnávány zeměpisné souřadnice aktuálního prvku a hledaných souřadnic. V případě nalezení shodného prvku je aktuální prvek hledaným a operace končí. Jestliže hledané souřadnice neodpovídají bude rozhodnuto, jakým způsobem bude traverzování pokračovat. Pokud hodnota zeměpisné délky aktuálního prvku je menší, než hledaná zeměpisná délka dochází k ukončení operace podle pravidel binární haldy a prvek lze označit jako nenalezený. Při splněných podmínkách binární haldy operace pokračuje, podle hranice aktuálního prvku do levého (větší hranice) či pravého (menší hranice) podstromu.

Postup operace Najdi je znázorněn na Obrázku 9, kde klíčem jsou souřadnice 85;15.



Obrázek 9 Operace Najdi – PST, zdroj: [vlastní]

Pseudokód hledání prvku ve struktuře

```
1.  Najdi (hledanaSouradnice)
2.  zacatek
3.    aktualniPrvek := root;
4.    dokud (aktualniPrvek != null) opakuj
5.    zacatek
6.      jestlize (aktualniPrvek.X == hledanaSouradnice.X
7.               AND aktualniPrvek.Y == hledanaSouradnice.Y) pak
8.      zacatek
9.       //prvek nalezen
10.       konecOperace aktualniPrvek;
11.     konec
```

```
12.
13.     jestlize(hledanaSouradnice.Y <= aktualniPrvek.Y) pak
14.       zacatek
15.         jestlize(aktualniPrvek.Hranice <= hledanaSouradnice.X) pak
16.           zacatek
17.             //traverzujeme vpravo podle hranice
18.             aktualniPrvek := aktualniPrvek.PravySyn;
19.           konec
20.           jinak
21.           zacatek
22.             //traverzujeme vlevo podle hranice
23.             aktualniPrvek := aktualniPrvek.LevySyn;
24.           konec;
25.       konec;
26.       jinak konecOperace null; //prvek nenalezen, pravidla haldy
27.   konec
28.  konecOperace null; //prvek nenalezen strom, strojim projit po listy
29. konec
```

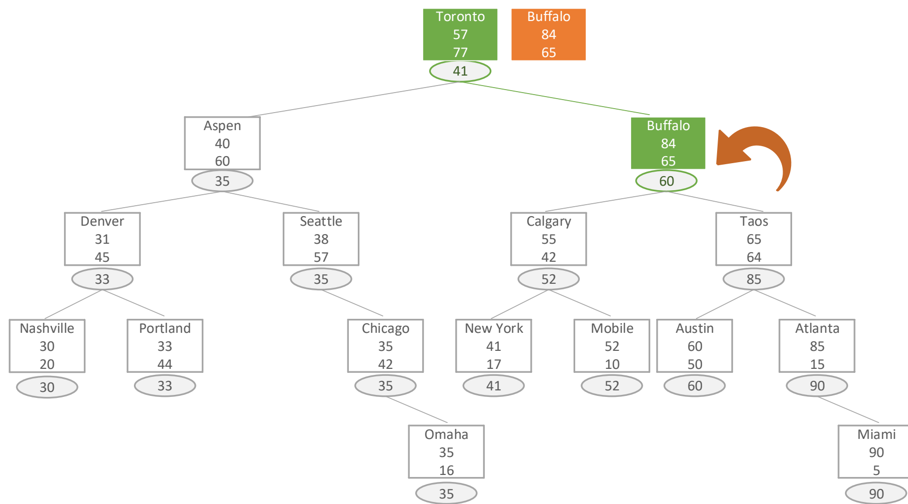
3.3.4 Operace odeber

Operaci *Odeber* lze rozdělit do dvou fázi. V první fázi je aplikována operace *Najdi,* a pokud byl prvek nalezen, následuje další fáze. V druhém případě je operace ukončena z důvodu nenaleznutí odebíraného prvku ve struktuře.

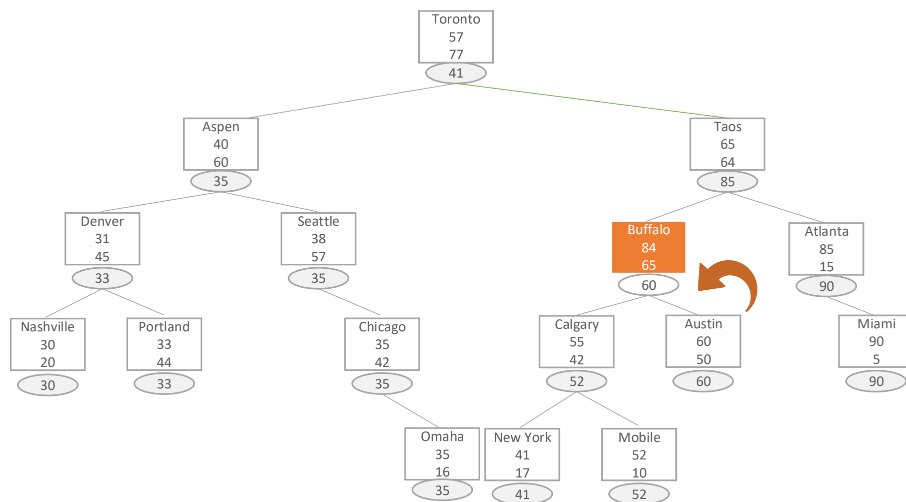
Při odebírání prvku je klasifikována pozice prvku ve stromu. V případě, že odebíraný prvek je listem stromu, prvek je přímo odebírán a ve stromové struktuře nejsou vykonány žádné reorganizační akce. Další možná situace je taková, že synové odebíraného prvku jsou listy stromu. V tomto případě je prvek odebrán a je nahrazen synem s vyšší hodnotou zeměpisné délky.

V ostatních případech je odebíraný prvek, přemísťován pomocí rotací směrem od kořene k listům. Operace *Rotace* je vykonávána do doby splnění podmínek u jednoho ze dvou výše zmíněných případů. Volba druhu rotace je vždy určena tak, aby se otcem odebíraného prvku stal syn, který má větší hodnotu zeměpisné délky.

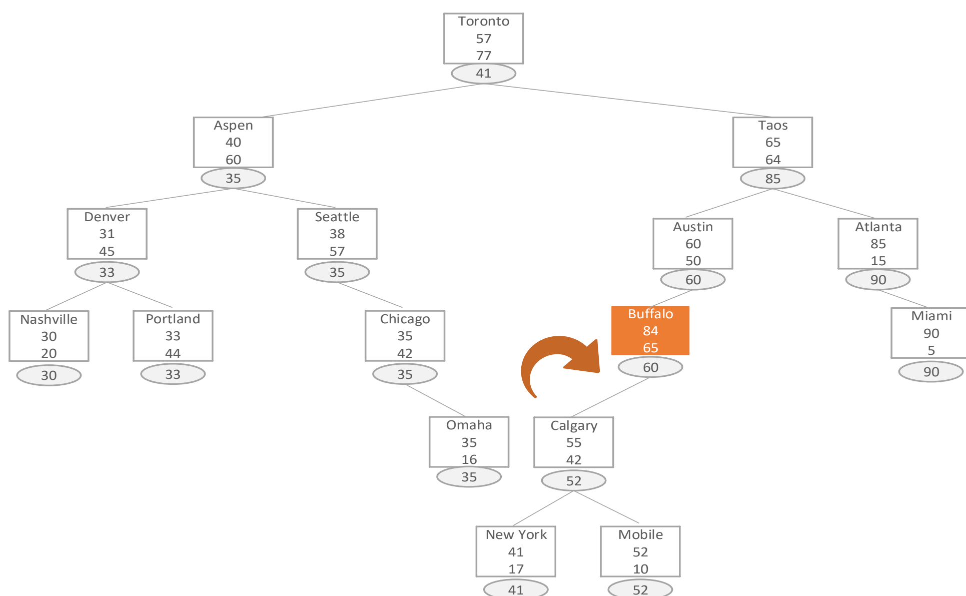
Operace *Odeber* je graficky znázorněna na následujících čtyřech obrázcích, kde se odebírá prvek se souřadnicemi zeměpisné šířky = 80 a zeměpisné délky = 65 (Buffalo). Pomocí levých a pravých rotací je odebíraný prvek přemísťován do stavu, kdy potomci prvku jsou listy stromu. Následně dochází k odstranění prvku způsobem, že prvek je nahrazen potomkem s vyšší hodnotou zeměpisné délky.



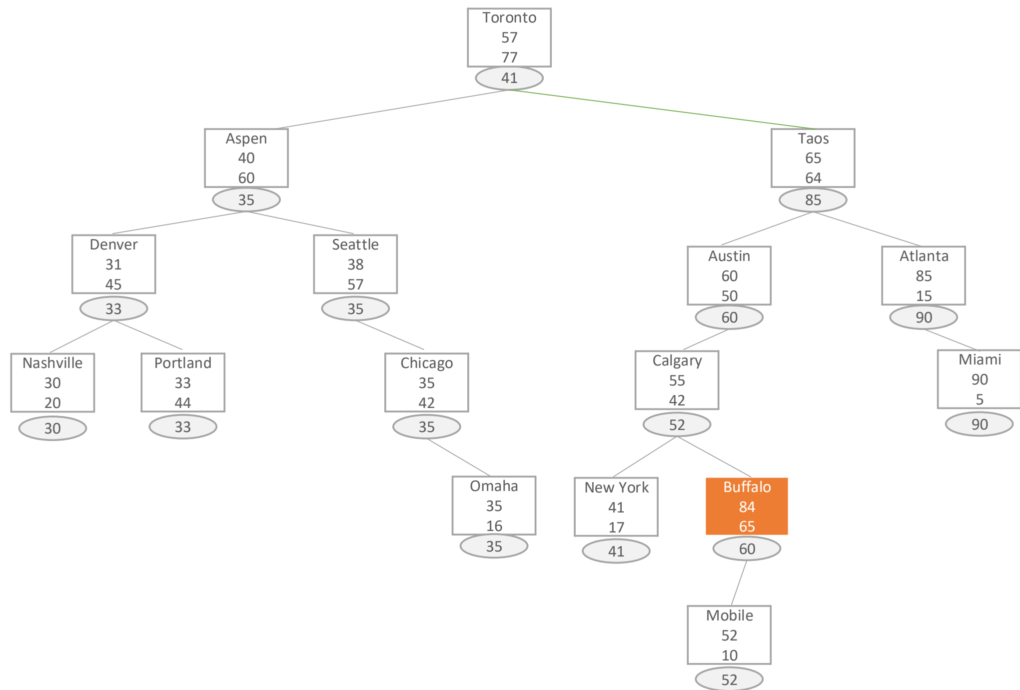
Obrázek 10 Operace Odeber PST – původní stav stromu, zdroj: [vlastní]



Obrázek 11 Operace Odeber PST – stav po levé rotaci zdroj: [vlastní]



Obrázek 12 Operace Odeber PST – stav po druhé levé rotaci, zdroj: [vlastní]



Obrázek 13 Operace Odeber PST – stav po pravé rotaci, zdroj: [vlastní]

Po poslední operaci *Rotaci* se prvek dostal do stavu, který splňuje podmínky pro odebrání prvku ze struktury. Prvek je odebrán a je nahrazen prvkem Mobile, který byl původně jeho synem.

Pseudokód odebrání prvku ve struktuře

```
1. Odeber (hledanaSouradnice)
2. zacatek
3.   hledanyPrvek := najdi(hledanaSouradnice);
4.   jestlize (hledanyPrvek != null) pak
5.     zacatek
6.      konecOperace odeberPrvek(hledanyPrvek);
7.     konec
8. konec
9.
10. odeberPrvek (odebiranyPrvek)
11.  zacatek
12.    jestlize jeListem(odebiranyPrvek) pak
13.     zacatek
14.      //je-li prvek listem, strom neni potreba reorganizovat a prvek je odebran
15.       odebranyPrvek := odeberList(odebiranyPrvek);
16.       konecOperace odebranyPrvek;
17.     konec
18.
19.    dokud(odebiranyPrvek.LevySyn != null || odebiranyPrvek.PravySyn != null) opakuj
20.    zacatek
21.     //pro zjednodušení a čitelnost podmínka upravena pro lepší čtení,
22.     //pristupujeme vzdy k odebiranemuPrvku a jeho potomkum
23.      jestlize LevySyn == null OR (LevySyn.LevySyn == null AND LevySyn.Pravy-
24.      Syn == null AND PravySyn == null) OR (PravySyn.LevySyn == null AND PravySyn.Pravy-
25.      Syn == null AND LevySyn == null) OR (PravySyn.LevySyn == null AND PravySyn.Pravy-
26.      Syn == null AND LevySyn.LevySyn == null AND LevySyn.PravySyn == null) pak
27.      zacatek
28.        jestlize jeListem(odebiranyPrvek) pak
29.        zacatek
30.         konecOperace  odeberList(odebiranyPrvek);
31.        konec
```

```
32.         jestlize LevySyn != null AND PravySyn != null pak
33.           zacatek
34.             jestlize LevySyn.Y > PravySyn.Y pak
35.             zacatek
36.               LevySyn.Otec := odebiranyPrvek.Otec;
37.               LevySyn.PravySyn := PravySyn;
38.               PravySyn.Otec := LevySyn;
39.               jestlize odebiranyPrvek == odebiranyPrvek.Otec.LevySyn pak
40.                 odebiranyPrvek.Otec.LevySyn := odebiranyPrvek.LevySyn;
41.               jinak
42.                 odebiranyPrvek.Otec.PravySyn := odebiranyPrvek.LevySyn;
43.               konec
44.             jinak
45.             zacatek
46.               PravySyn.Otec := odebiranyPrvek.Otec;
47.               PravySyn.LevySyn := LevySyn;
48.               LevySyn.Otec := PravySyn;
49.               jestlize odebiranyPrvek == odebiranyPrvek.Otec.LevySyn pak
50.               zacatek
51.                 jestlize LevySyn.X < PravySyn.X pak
52.                 LevySyn.LevySyn := PravySyn;
53.                 jinak
54.                 LevySyn.PravySyn := PravySyn;
55.               konec
56.               jinak
57.               zacatek
58.                 jestlize LevySyn.X < PravySyn.X pak PravySyn.LevySyn := LevySyn;
59.                 jinak PravySyn.PravySyn := LevySyn;
60.               konec
61.             konec
62.           konec
63.           jinak
64.           zacatek
65.           //prvek s jedním potomkem, potomek na pozici otce
66.             jestlize LevySyn == null pak
67.               konecOperace odeberPrvek_PravySyn(odebiranyPrvek)
68.             jestlize PravySyn == null pak
69.               konecOperace odeberPrvek_LevySyn(odebiranyPrvek)
70.           konec
71.        konec
72.        jinak
73.        zacatek
74.        //prvek bez upravy stromu nelze odebrat, provadime rotace stromu
75.          jestlize LevySyn != nul && PravySyn != null pak
76.          zacatek
77.            jestlize LevySyn.Y > PravySyn.Y pak pravaRotace(odebiranyPrvek);
78.            jinak levaRotace(odebiranyPrvek);
79.          konec
80.          jinak
81.          jestlize LevySyn == null pak levaRotace(odebiranyPrvek);
82.          jinak pravaRotace(odebiranyPrvek);
83.          Konec
84.       konec
85. konec
```

3.4 Rozsahový strom

Rozsahový strom je datová struktura, která efektivně slouží k uchování dvou a více rozměrných dat. Datová struktura byla vytvořena jako alternativa k-d stromu. Ve srovnání rozsahový strom nabízí efektivnější dotazovací operace v časové složitosti O ($\log^d n + k$), kde ***n*** je počet prvků ve stromu, ***d*** je dimenze každého bodu a ***k*** je počet naleznutých bodů, ale méně efektivní operace pro vkládání či odebírání dat ze struktury v časové složitosti O ($n*\log^{d-1}n$) [6].

Rozsahový strom se vyznačuje tím, že datové prvky se nachází pouze na úrovni listů, kde jsou mezi sebou zřetězeny a zároveň seřazeny podle jedné z dimenzí. Ostatní prvky rozsahového stromu slouží jako navigační. Navigační prvky mají uloženou informaci o intervalu, do kterého patří všechny jejich potomci, taktéž každý navigační prvek obsahuje strom druhé dimenze [1]. Pravidla, která platí v rozsahovém stromu:

- každý prvek má maximálně dva syny,
- pouze listy obsahují data,
- navigační prvky ukládají interval všech svých potomků,
- každý navigační prvek má odkaz na sekundární strom,
- listy jsou zřetězeny.

3.4.1 Operace vybuduj

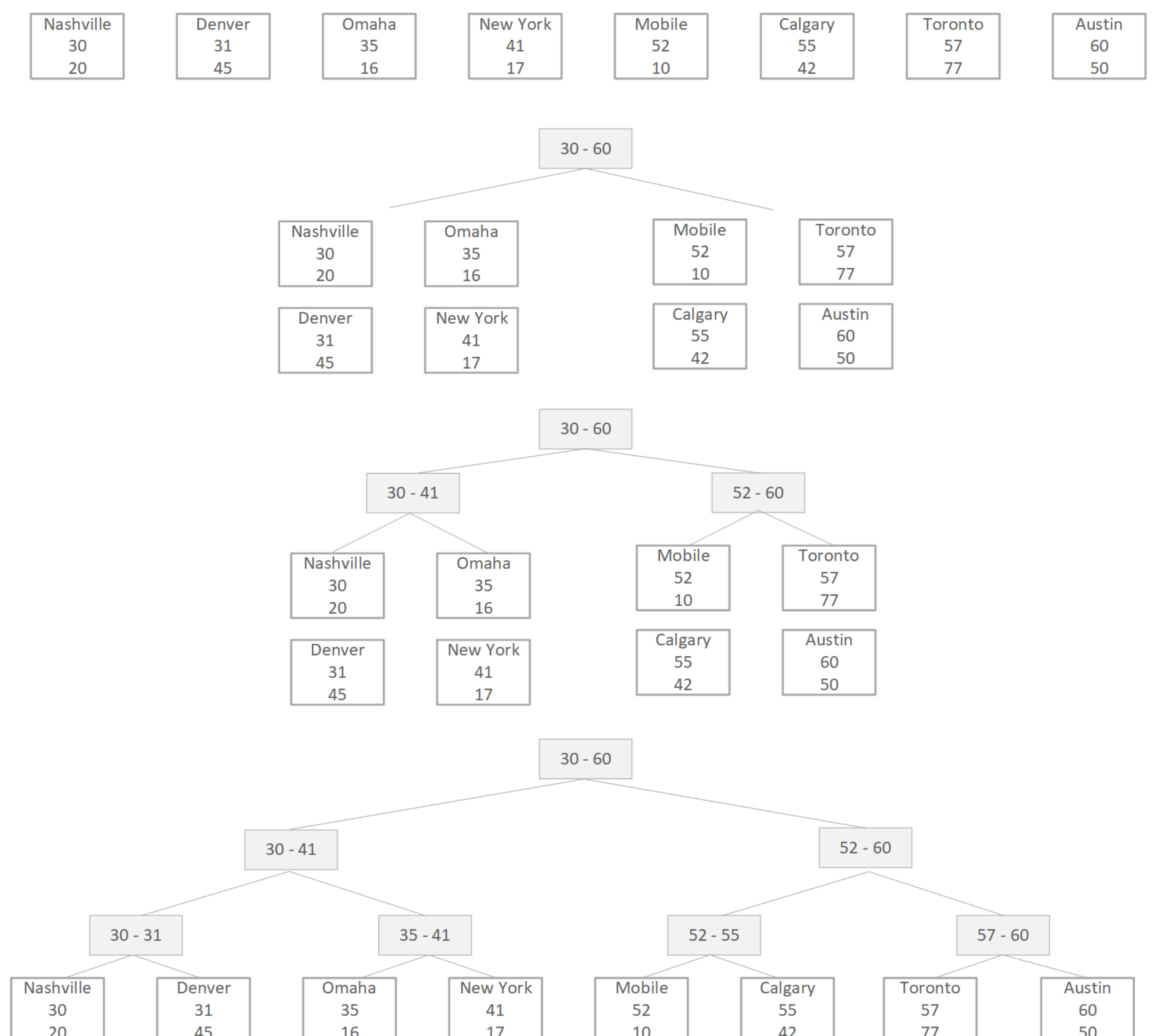
Operaci *Vybuduj*, lze provést pouze s předem známou vstupní sadou dat. Operace zajišťuje vybudovat stejně jako u prioritního vyhledávacího stromu vyváženou stromovou strukturu. Při budování, nejprve tvoříme navigační strukturu, přičemž každý navigační prvek obsahuje referenci na sekundární binární vyhledávací strom.

Operace začíná ověřením velikosti vstupní sady dat. Je-li vstupní sada dat větší než jeden prvek, dochází k postupnému tvoření navigační strukturu hlavního stromu. Navigační prvek, vzniká z nejmenšího možného intervalu zeměpisné šířky z aktuální sady zpracovaných dat. Nadále jsou data rozdělovány pomocí zeměpisné šířky na polovinu a jsou rekurzivně vkládány do levého (první polovina) a pravého (druhá polovina) podstromu navigačního prvku. Následující postup se opakuje do doby, než se nachází v každé polovině maximálně jeden prvek. Jakmile zůstává v sadě jeden prvek, vzniká list, který drží již konkrétní prvek s jeho klíčem a daty. Taktéž, každý navigační prvek obsahuje referenci na sekundární binární vyhledávací strom. Na

rozdíl od hlavní navigační datové struktury, sekundární je tvořena pomocí zeměpisné délky. Postup tvoření sekundárního vyhledávacího stromu je obdobný jako u hlavního stromu.

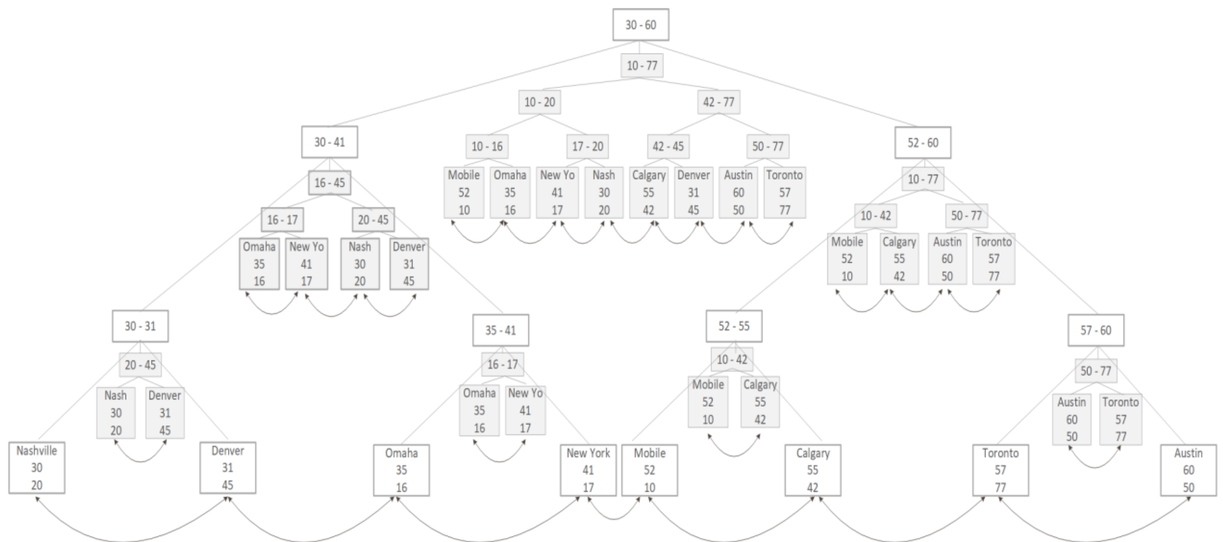
V grafickém představení operací bude použita poloviční sada dat z důvodu tvorby rozsáhlejší navigační datové struktury při budování sekundární dimenze prvků. Taktéž při ukázce operace *Vybuduj* bude ukázáno pouze vybudování hlavní navigační struktury. Sekundární dimenze se tvoří stejným postupem a liší se pouze v záměně zeměpisné šířky za zeměpisnou délku.

Na Obrázku 14 lze vidět postup tvoření hlavní navigační struktury rozsahového stromu při datové sadě o velikosti osmi prvků. Při postupu dochází k cyklickému dělení skupin prvků do doby, než jsou vytvořeny listy stromu.



Obrázek 14 Vybudování hlavní navigační struktury rozsahového stromu, zdroj: [vlastní]

Obrázek 15 zobrazuje kompletní vybudovanou strukturu Rozsahového stromu po operaci *Vybuduj* ze vstupní sady dat o velikosti osmi prvků.



Obrázek 15 Vybudovaný rozsahový strom, zdroj: [vlastní]

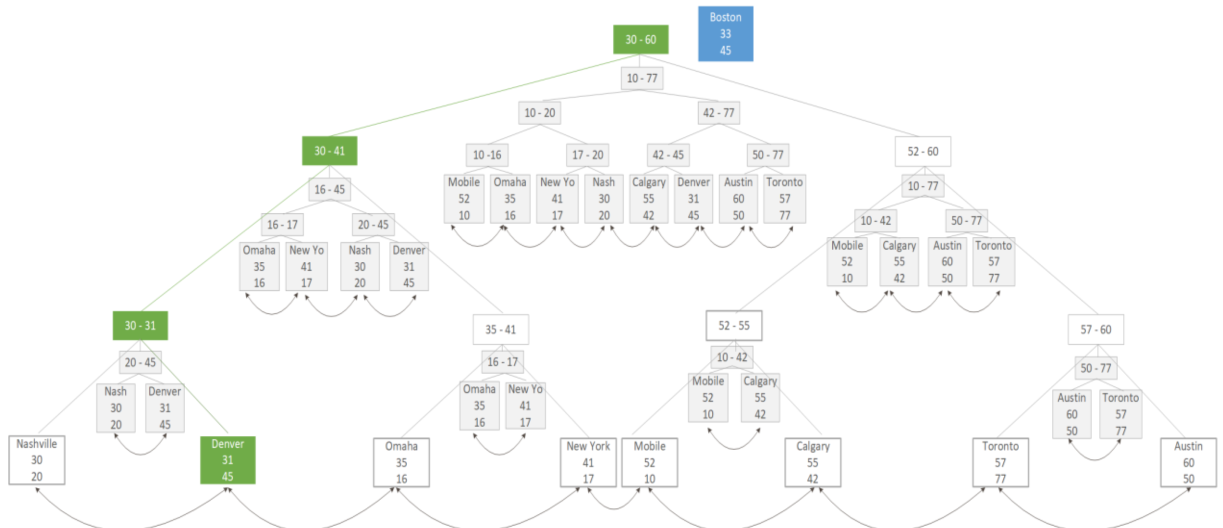
Pseudokód vybudování struktury

```
1.  Vybuduj (vstupniData)
2.  zacatek
3.    jestlize !jePrazdny() || delka(vstupniData) == 0 pak konecOperace;
4.    pocetPrvku := delka(vstupniData);
5.    //vytvoreni korene stromu
6.    koren := vybuduj(vstupniData, null, 0, podleX);
7.  konec
8.
9.   Vybuduj (vstupniData, otec, uroven, typRazeni)
10.  zacatek
11.   novyPrvek := null;
12.   jestlize delka(vstupniData) > 1 pak
13.   zacatek
14.     novyPrvek := vytvorNavigacniPrvek(vstupniData, otec, uroven, typRazeni);
15.   konec
16.   jinak
17.   zacatek
18.     //vytvoreni noveho listu a vazeb
19.     novyPrvek := vytvorList(vstupniData, posledniVlozenyList);
20.   konec
21.   //pokud tvorime navigacni prvek v hlavni stromove strukture,
22.   // vytvorime sekundarni strom
23.   jestlize !jeListem(novyPrvek) AND typRazeni == podleX pak
24.     zacatek
25.       novyPrvek.sekundarniStrom := vybuduj(vstupniData, null, 0, podleY);
26.       novyPrvek.sekundarniStrom.Otec := novyPrvek;
27.     konec
28. konec
29.
30. VytvorNavigacniPrvek (vstupniData, otec, uroven, typRazeni)
31. zacatek
32.   vstupniData := seradData(vstupniData, typRazeni);
33.   //podle typu razeni, zjišťujeme zeměpisnou delku, nebo sirku
34.   x := ziskejNejmensiHodnotu(vstupniData, typRazeni);
35.   y := ziskejNejvetsiHodnotu(vstupniData, typRazeni);
36.   navigacniPrvek := novy Prvek(x, y, otec, uroven, typRazeni);
37.   rozdelData(vstupniData, prvniPolovina, druhaPolovina);
38.   navigacniPrvek.LevySyn := vybuduj(prvniPolovina, navigacniPrvek, uroven + 1,
39.                                   typRazeni);
40.   navigacniPrvek.PravySyn := vybuduj(druhaPolovina, navigacniPrvek, uroven + 1,
41.                                   typRazeni);
42. konec
```

3.4.2 Operace vlož

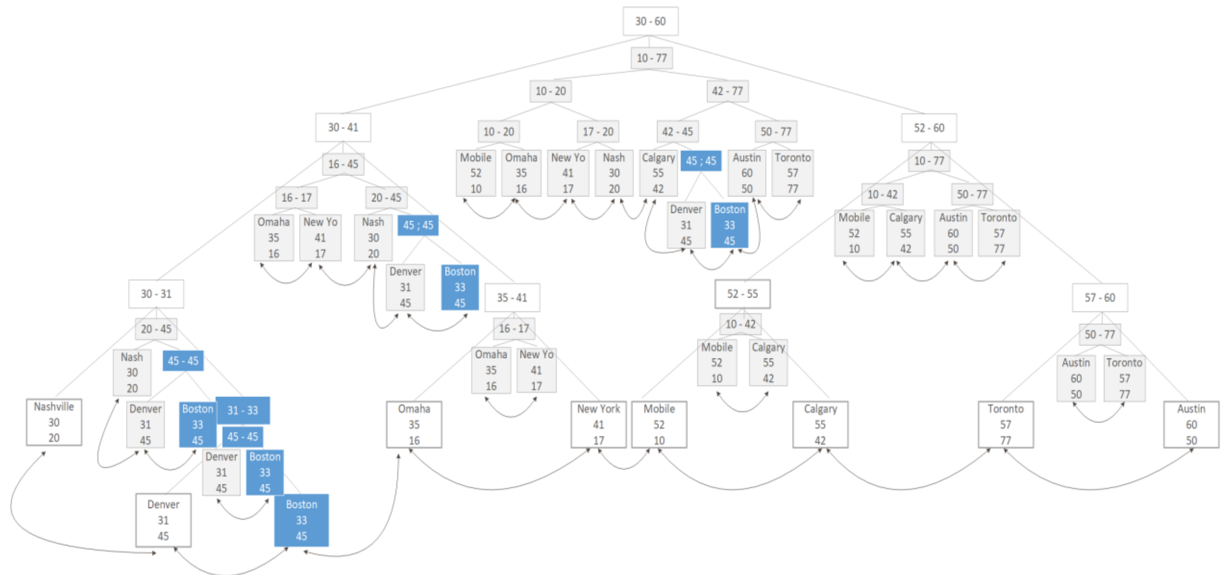
Při operaci *Vlož* je postupně stromová struktura traverzována směrem od kořene k listům. Při průchodu jsou kontrolovány synové navigačního vrcholu a pomocí jejich intervalů je rozhodnuto, jestli následující prvek bude z levého či pravého podstromu navigačního prvku. Pro každý navštívený navigační prvek je do jeho sekundárního vyhledávacího stromu taktéž vložen nový prvek s obdobným postupem, jako při vkládání do hlavní stromové struktury a je upraven interval navigačního vrcholu. Traverzování končí, jakmile je dosažena pozice listu (plnohodnotný prvek stromu). Aktuální prvek je nahrazen novým navigačním prvkem, který byl vytvořen pomocí klíčů z aktuálního prvku a nově vkládaného prvku a oba prvky jsou vloženy jako synové nového navigačního prvku.

Grafické znázornění operace *Vlož* zobrazuje na Obrázku 16 cestu ke správné pozici vkládaného prvku Boston se souřadnicemi 33;45. Modrou barvou je označen nově vkládaný prvek a zelenou cesta k hledanému prvku.



Obrázek 16 První krok operace vlož na rozsahovém stromu, zdroj: [vlastní]

Na druhém obrázku (Obrázek 17) je vytvořen nový navigační prvek na původní pozici listu, pro který byl vložen původní prvek a nově vkládaný prvek. Obdobně byl do všech sekundárních dimenzí procházených navigačních prvků vložen prvek Boston.



Obrázek 17 Rozsahový strom po vložení prvku, zdroj: [vlastní]

Pseudokód vložení prvku do struktury

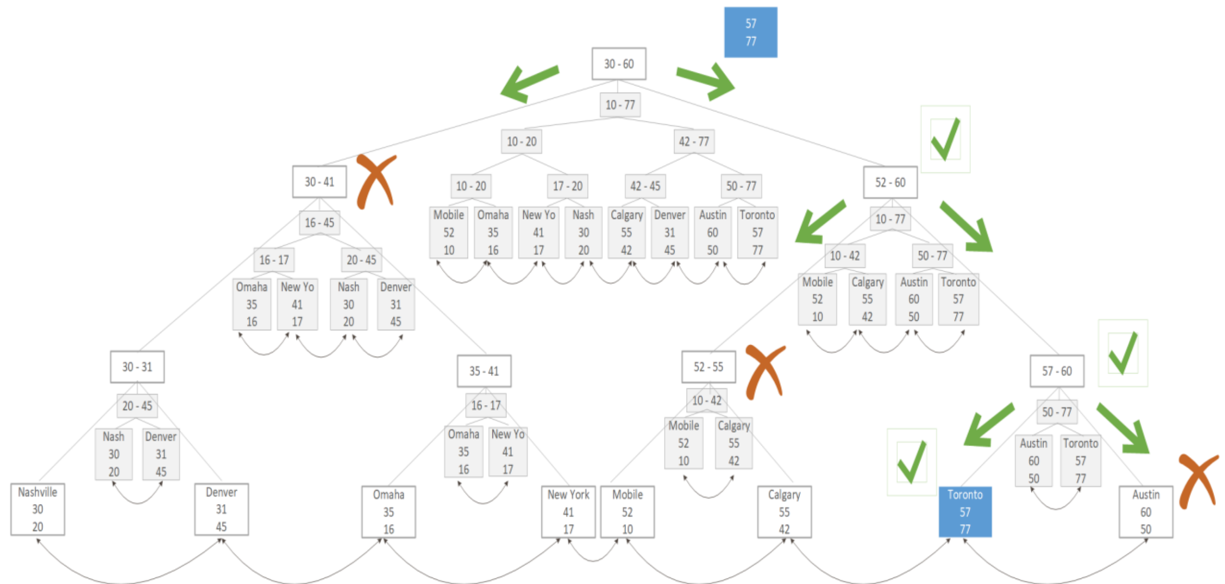
```
1. Vloz (vkladanyPrvek)
2. zacatek
3.   jestlize koren == null pak
4.   zacatek
5.     koren := vkladanyPrvek;
6.     pocetPrvku++;
7.     konecOperace;
8.   konec
9.
10.   vloz(vkladanyPrvek, otecPrvku, uroven, podleX);
11.   pocetPrvku++;
12. konec
13.
14.
15. Vloz (vkladanyPrvek, otecPrvku, uroven, typRazeni)
16. zacatek
17.   jestlize jeListem(otecPrvku) pak
18.     vlozPrvekDoListu(vkladanyPrvek, otecPrvku,uroven, typRazeni)
19.   jinak
20.     vlozPrvek(vkladanyPrvek, otecPrvku, uroven, typRazeni)
21.  konec
22.
23.
24. vlozPrvek (vkladanyPrvek, otecPrvku, uroven, typRazeni)
25. zacatek
26.   //pri traverzovani prvku upravujeme rozsahy intervalu
27.   UpravRozsahNavigacnihoPrvku (vkladanyPrvek, otecPrvku, typRazeni);
28.   Souradnice := typRazeni == podleX ? vkladanyPrvek.X : vkladanyPrvek.Y;
29.   levySyn := otecPrvku.levySyn;
30.   pravySyn := otecPrvku.pravySyn;
31.
32.
```

```
33.   //traverzujeme strom podle podminek
34.   jestlize NeniListem(levySyn) AND souradnice <= levySyn.Y pak
35.     vloz(node, levySyn, uroven, typRazeni);
36.   jinak
37.   jestlize NeniListem(pravySyn) AND souradnice >= pravySyn.X pak
38.     vloz(node, pravySyn, uroven, typRazeni);
39.   jinak
40.   jestlize JeListem(levySyn) AND
41.   (typRazeni == podleX ? (levySyn.X >= souradnice) : (levySyn.Y >= souradnice) pak
42.     vloz(node, levySyn, uroven, typRazeni);
43.   jinak
44.   jestlize JeListem(pravySyn) AND
45.   (typRazeni == podleX ? (souradnice >= pravySyn.X) : (souradnice >= pravySyn.Y) pak
46.     vloz(node, pravySyn, uroven, typRazeni);
47.   jinak
48.     vloz(node, levySyn, uroven, typRazeni);
49.
50.   //jestlize traverzujeme hlavnim stromem, musime vlozit prvek take do
51.   //sekundarniho stromu
52.   jestlize typRazeni == podleX pak
53.     vloz(vkladanyPrvek, otecPrvku.SekundarniStrom, uroven, podleY);
54. konec
55.
56.
57.   VlozPrvekDoListu (vkladanyPrvek, otecPrvku, uroven, typRazeni)
58.   zacatek
59.     jestlize (typRazeni == podleX) pak
60.     zacatek
61.       //vytvoreni noveho navigacniho prvku a posunuti otcePrvku a aktualne
62.       //vkladanehoPrvku na spravne pozice
63.       navigacniPrvek := vytvorPrvek(vkladanyPrvek, otecPrvku, uroven,
64.                                     typRazeni);
65.       //jakmile tvorime novy navigacni prvek, musime vytvorit taktez sekundarni strom
66.       VytvorSekundarniStrom(navigacniPrvek, vkladanyPrvek, otecPrvku,
67.                             uroven, typRazeni);
68.     konec
69.     jinak
70.     zacatek
71.       //vytvoreni noveho navigacniho prvku a posunuti otcePrvku a
72.       //aktualne vkladanehoPrvku na spravne pozice
73.       vytvorPrvek(vkladanyPrvek, otecPrvku, uroven, typRazeni);
74.      konec
75.   konec
```

3.4.3 Operace najdi

Operace *Najdi* hledá prvek pomocí postupného procházení stromu od kořene stromu směrem k listům, přičemž jednoduchá operace hledání je vykonávána pouze v hlavní navigační struktuře stromu. V každém navigačním vrcholu je testováno, jestli vyhledávaná souřadnice je součástí intervalu vrcholu. Pokud souřadnice není v intervalu vyhledávání, dojde k ukončení bez nalezeného prvku. Jakmile je prvek součástí intervalu, operace pokračuje do obou prvků podstromu a je rekurzivně testováno, jestli je prvek součástí intervalu do doby, než je dotraverzováno do listu stromu. Při dosažení je testováno, jestli souřadnice hledaného prvku odpovídají souřadnicím listu a pokud souřadnice odpovídají, prvek je nalezen a navrácen.

Ukázka operace najdi v grafickém provedení je na Obrázku 18 pro souřadnice 57;77.



Obrázek 18 Operace najdi v rozsahovém stromu, zdroj: [vlastní]

Pseudokód hledání prvku ve struktuře

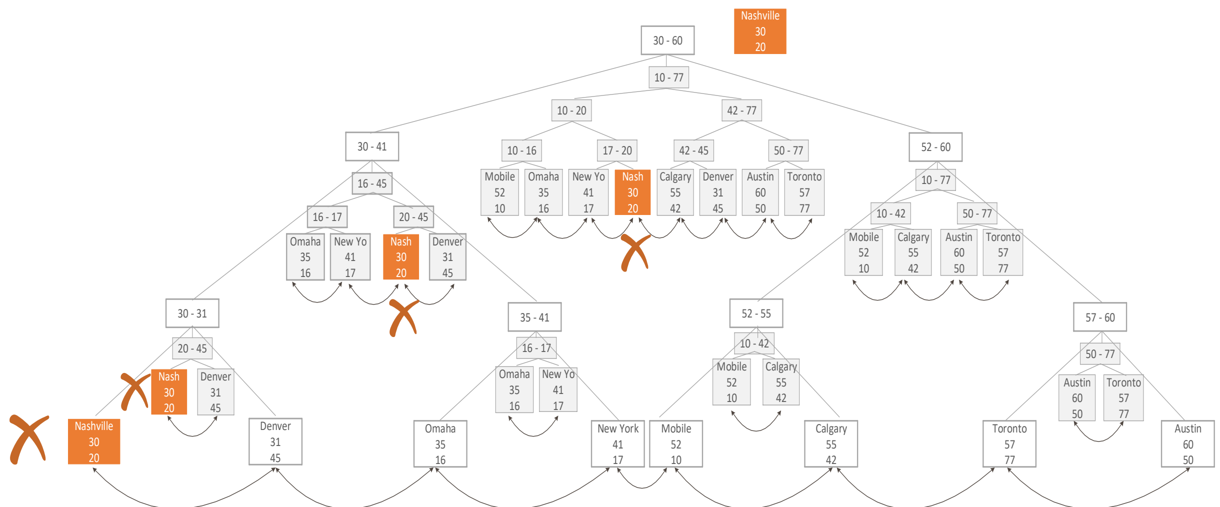
```
1.  najdi(hledanaSouradnice)
2.  zacatek
3.    aktualniPrvek := koren;
4.
5.    dokud(aktualniPrvek != null) opakuj
6.    zacatek
7.      //hledany prvek je levym potomkem, aktualne prochazeneho prvku
8.      jestlize(JeListem(aktualniPrvek.LevySyn) AND
9.      hledanaSouradnice == aktualniPrvek.LevySyn.souradnice) pak
10.       konecOperace aktualniPrvek.LevySyn;
11.
12.     //hledany prvek je pravym potomkem, aktualne prochazeneho prvku
13.     jestlize(JeListem(aktualniPrvek.PravySyn) AND
14.     hledanaSouradnice == aktualniPrvek.PravySyn.souradnice) pak
15.      konecOperace aktualniPrvek.PravySyn;
16.
17.     //prvek nebyl nalezen traverzujeme stromem
18.     jestlize(NeniListem(aktualniPrvek.LevySyn) AND
19.     aktualniPrvek.LevySyn.X <= hledanaSouradnice.X AND
20.     aktualniPrvek.LevySyn.Y > hledanaSouradnice.X pak
21.       //traverzujeme vlevo
22.       aktualniPrvek := aktualniPrvek.LevySyn;
23.     jinak
24.     zacatek
25.       jestlize(JeListem(aktualniPrvek.PravySyn) OR
26.       aktualniPrvek.X > hledanaSouradnice.X OR
27.       aktualniPrvek.Y <= hledanaSouradnice.X pak
28.         konecOperace null;
29.
30.       //traverzujeme v pravo
31.       aktualniPrvek := aktualniPrvek.PravySyn;
32.     konec
33.   konec
34.
35. //prvek nebyl nalezen
36.   konecOperace null;
37. konec
```

3.4.4 Operace odeber

V Operaci *Odeber je* nejprve procházeno stromem od kořene směrem k listům. Přičemž, na každém hlavním navigačním prvku je testováno, jestli sekundární strom neobsahuje referenci na odebíraný prvek. Z toho vyplývá, že při operaci odebírání je prvek mazán z hlavní stromové struktury a zároveň, ze všech pomocných sekundárních stromových struktur. Jakmile je do traverzováno k listu stromu, odebíraný prvek je odstraněn a bratr odebíraného prvku je přemístěn na pozici otce.

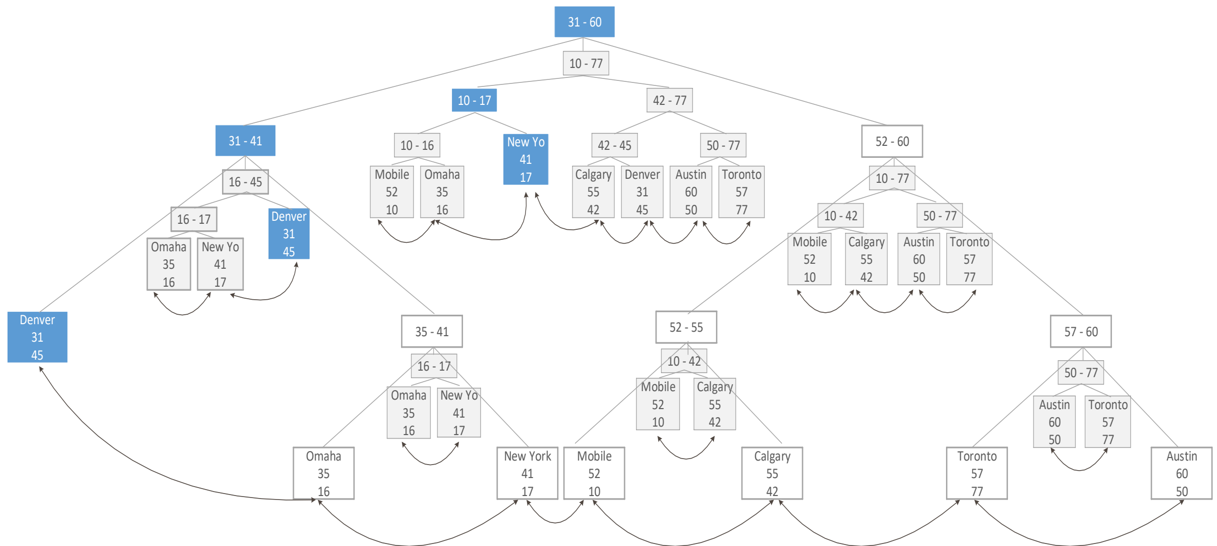
Po odebrání prvku, je strom procházen pomocí odkazu aktuálně přemístěného prvku (bratr odebraného prvku) na otce směrem ke kořenu stromu a podle potřeby jsou upravovány intervaly navigačních prvků.

Při grafickém zpracování není ukázáno traverzování k odebíranému prvku od kořenu stromu, protože operace Najdi je graficky zpracována v kapitole 3.4.3 na Obrázku 18. Na obrázku jsou nejprve označeny všechny instance prvku Nashville v hlavní i vedlejších navigačních strukturách.



Obrázek 19 Operace odeber Rozsahový strom – označen prvek pro odebraní, zdroj: [vlastní]

Druhým krokem operace *Odeber* je odebrání všech instancí odebíraného prvku a upravení struktury stromu, viditelné na Obrázku 20. Po odebrání dochází ke zrušení přebytečných navigačních prvků a úpravy intervalů od přesunutého prvku ke kořenu stromu.



Obrázek 20 Stav Rozsahového stromu po odebrání prvku, zdroj: [vlastní]

Pseudokód odebrání prvku ve struktuře

```
1. Odeber(hledanaSouradnice)
2. zacatek
3.   odebiranyPrvek := odeberPrvek(hledanaSouradnice, root, podleX, 0);
4.   jestlize(odebiranyPrvek != null) pak
5.   zacatek
6.     konecOperace odebiranyPrvek;
7.   konec
8. konec
9.
10. OdeberPrvek(hledanaSouradnice, aktualniPrvek, typRazeni, uroven)
11. zacatek
12.   jestlize(JeListem(aktualniPrvek) AND
13.   aktualniPrvek.souradnice == hledanaSouradnice) pak
14.     odeberList(aktualniPrvek, typRazeni, uroven);
15.
16.   //pokud prvek neni listem traverzujeme az po listy
17.   jestlize typRazeni == podleX pak
18.   zacatek
19.     jestlize aktualniPrvek.X <= hledanaSouradnice.X AND
20.     hledanaSouradnice.Y <= aktualniPrvek.Y pak
21.     zacatek
22.       //pokud prochazime strom v hlavni navigacni strukture, musime take
23.       //projit vedlejsi
24.       odeberPrvek(hledanaSouradnice, aktualniPrvek.SekundarniStrom, podleY,
25.                     uroven + 1);
26.       odebranyPrvek := odeberPrvek(hledanaSouradnice, aktualniPrvek.LevySyn,
27.                              podleX, uroven + 1);
28.       jestlize(odebranyPrvek != null) konecOperace odebranyPrvek;
29.       odebranyPrvek := odeberPrvek(hledanaSouradnice, aktualniPrvek.PravySyn,
30.                              podleX, uroven + 1);
31.       jestlize(odebranyPrvek != null) konecOperace odebranyPrvek;
32.      konec
33.   konec
34.   jinak
35.   zacatek
36.     jestlize aktualniPrvek.X <= hledanaSouradnice.X AND
37.     hledanaSouradnice.Y <= aktualniPrvek.Y pak
38.     zacatek
39.       odebranyPrvek := odeberPrvek(hledanaSouradnice, aktualniPrvek.LevySyn,
40.                              podleY, uroven + 1);
41.       jestlize(odebranyPrvek != null) konecOperace odebranyPrvek;
```

```
42.       odebranyPrvek := odeberPrvek(hledanaSouradnice, aktualniPrvek.PravySyn,
43.                                    podleY, uroven + 1);
44.       jestlize(odebranyPrvek != null) konecOperace odebranyPrvek;
45.     konec
46.   konec
47.   konecOperace null;
48. konec
49.
50. odeberList(aktualniPrvek, typRazeni, uroven)
51. zacatek
52.   jestlize aktualniPrvek.Otec == null pak
53.     odeberListBezOtce(aktualniPrvek, typRazeni, level);
54.   jestlize aktualniPrvek.Otec.Otec == null pak
55.     odeberListBezSekundarnihoOtce(aktualniPrvek, typRazeni, level);
56.   jestlize aktualniPrvek.Otec == aktualniPrvek.Otec.Otec.LevySyn pak
57.     odeberListOtec_OtecOtecLevySyn(aktualniPrvek, typRazeni, level);
58.   jestlize aktualniPrvek == aktualniPrvek.Otec.LevySyn pak
59.     odeberListPrvek_OtecLevySyn(aktualniPrvek, typRazeni, level);
60.   jestlize aktualniPrvek == aktualniPrvek.Otec.PravySyn pak
61.     odeberListPrvek_OtecPravySyn(aktualniPrvek, typRazeni, level);
62.
63.   //po odebrani prvku, traverzujeme od otce odebiraneho prvku smerem ke korenu
64.   //a kontrolujeme upravujeme interval navigacnich prvku
65.   dokud(aktualniPrvek!= null) opakuj
66.   zacatek
67.     UpravRozsahNavigacnihoPrvku(aktualniPrvek, typRazeni);
68.     aktualniPrvek := aktualniPrvek.Otec;
69.   konec
70. konec
```

3.5 Quad Strom

Datová struktura Quad strom je struktura, kde každý prvek stromu, který není listem má čtyři syny. Prvek *p* stromu *t* reprezentuje čtvercový region *r*, čtyři synové prvku *p* reprezentují severovýchodní, jihovýchodní, jihozápadní a severozápadní kvartál regionu r [1]. Pravidla, která platí pro všechny typy Quad stromů:

- rozkládají prostor do buněk,
- každá buňka má maximální kapacitu, pokud je kapacita dosažena, buňka je rozdělena,
- struktura stromu následuje prostorovou dekompozici Quad stromu.

3.5.1 Typy implementací

Datová struktura Quad strom může být klasifikována podle typu dat které reprezentují, podle oblastí, bodů čar či křivek. Z tohoto důvodu existuje několik typů implementací. Jednoduše jdou tyto typy rozdělit do dvou kategorií. První kategorie, do kterého spadá typ Bodový Quad strom, splňuje podmínku, že každá buňka v stromu je plnohodnotná na rozdíl od druhé kategorie, kde pouze listy drží data a ostatní prvky jsou navigační [6]. Typy:

- oblastní quad strom (region quad tree),
- bodový quad strom (point quad tree),
- trie – based quadtrees
    - MX quadtree,
    - bodově oblastní quad strom (point region quad treee)[1].

3.5.2 Využití v praxi

Datová struktura Quad Strom je z vybraných rozebraných datových struktur uchovávající multidimenzionální data nejvíce využívána a implementována. Některá z využití:

- efektivní detekce kolizí ve 2D obrazu,
- zpracování obrazu,
- prostorové indexování v databázích,
- zobrazení „cullingu“ v počítačové grafice.

3.5.3 Bodový Quad Strom

Bodový Quad strom, který je adaptací binárního vyhledávacího stromu pro dvourozměrná bodová data, při kterém platí, že jednotlivé regiony jsou děleny pomocí konkrétních bodů. V praxi již Bodové Quad Stromy nejsou využívány, protože byly nahrazeny K-D stromy, které disponují efektivnější výpočetní složitostí hlavních operací [1][6].

3.5.3.1 Operace vybuduj

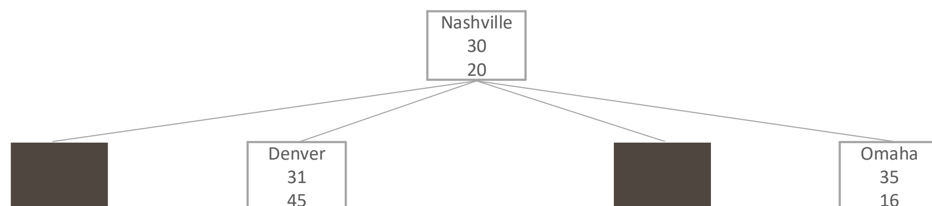
Při budování Bodového Quad stromu na rozdíl od předchozích stromových struktur využíváme operaci *Vlož* a není vytvořen žádný specifický algoritmus pro vložení prvku do struktury.
Jediná odlišnost mezi operací *Vybuduj* a operací *Vlož* je taková, že prvky před vložením struktury jsou seřazeny pomocí Z-křivky, která zajišťuje optimalizaci vkládání prvků do struktury. Výhody seřazení dat pomocí Z-křivky jsou popsány v článku od Sameta [8]. Z-křivka někdy taktéž označována jako Mortonův rozklad bude popsána v kapitole 4.2.
Grafický postup operace *Vybuduj* je zobrazen na následujícím Obrázku 21. Vstupní sada dat je nejprve seřazena a poté ukázána situace po vybudovaní prvního patra stromu.

Miami 90 5 | Mobile 52 10 | Atlanta 85 15 | Omaha 35 16 | New York 41 17 | Nashville 30 20 | Chicago 35 42 | Calgary 55 42 | Portland 33 44 | Denver 31 45
Austin 60 50 | Seattle 38 57 | Aspen 40 60 | Taos 65 64 | Buffalo 84 65 | Toronto 57 77

Z-Order sort

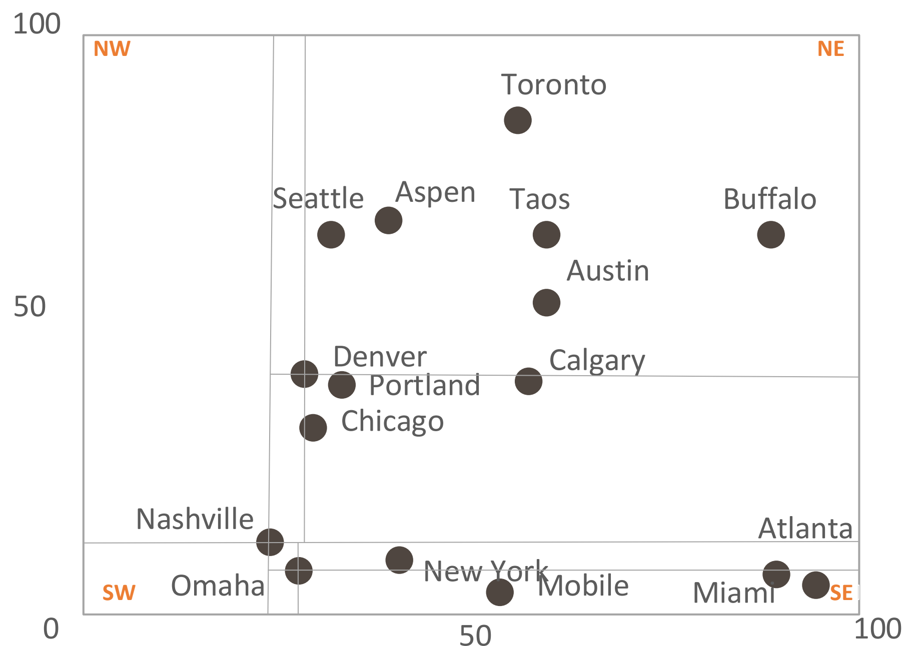
Nashville 30 20 | Denver 31 45 | Omaha 35 16 | New York 41 17 | Mobile 52 10 | Chicago 35 42 | Portland 33 44 | Seattle 38 57 | Aspen 40 60 | Calgary 55 42
Austin 60 50 | Toronto 57 77 | Atlanta 85 15 | Miami 90 5 | Taos 65 64 | Buffalo 84 65

Opakováné volání operace Vlož



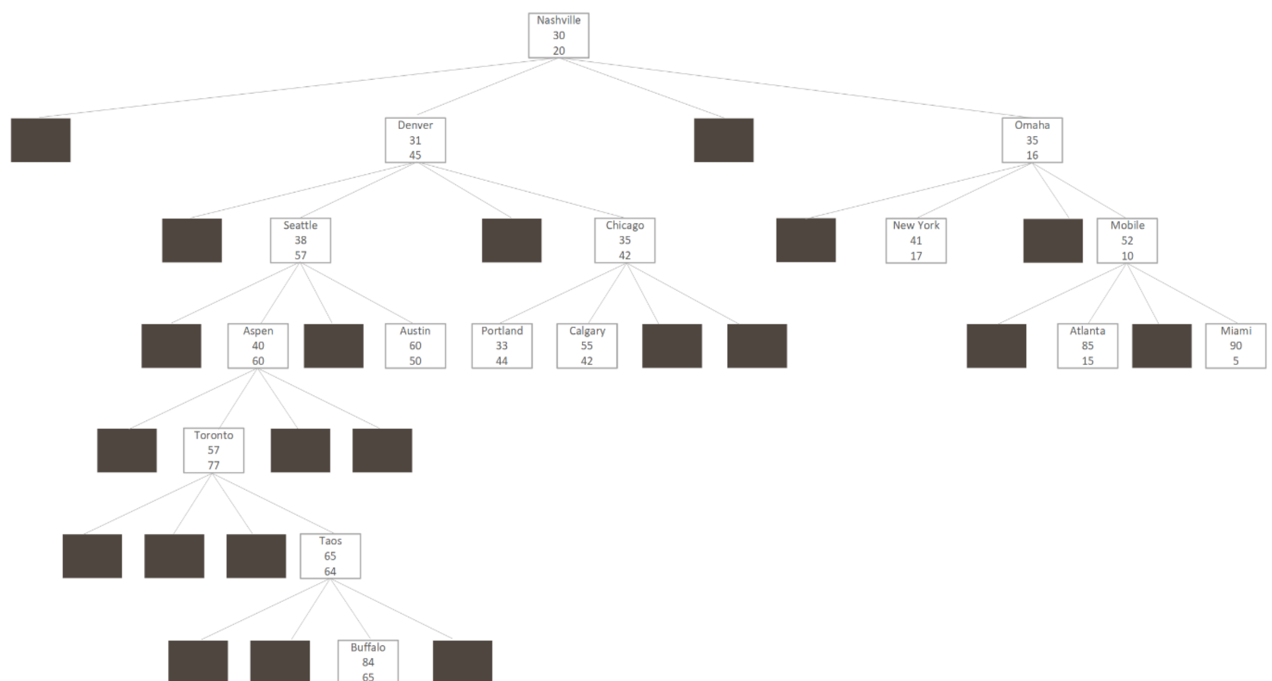
Obrázek 21 Ukázka vybudování prvního patra Bodového Quad Stromu, zdroj: [vlastní]

Dělení 2D prostoru pomocí hranic hledaných souřadnic je zobrazeno na Obrázku 22. Prostor je dělen pro kořen a první patro stromu. Dělení probíhá do doby, než je v každém regionu maximálně jeden prvek.



Obrázek 22 Dělení 2D prostoru v Bodovém Quad Stromu, zdroj: [vlastní]

Jakmile je celý 2D prostor rozdělen do stavu, kdy v každém děleném regionu je maximálně jeden prvek, dochází k vybudování struktury, která je představena na Obrázku 23.



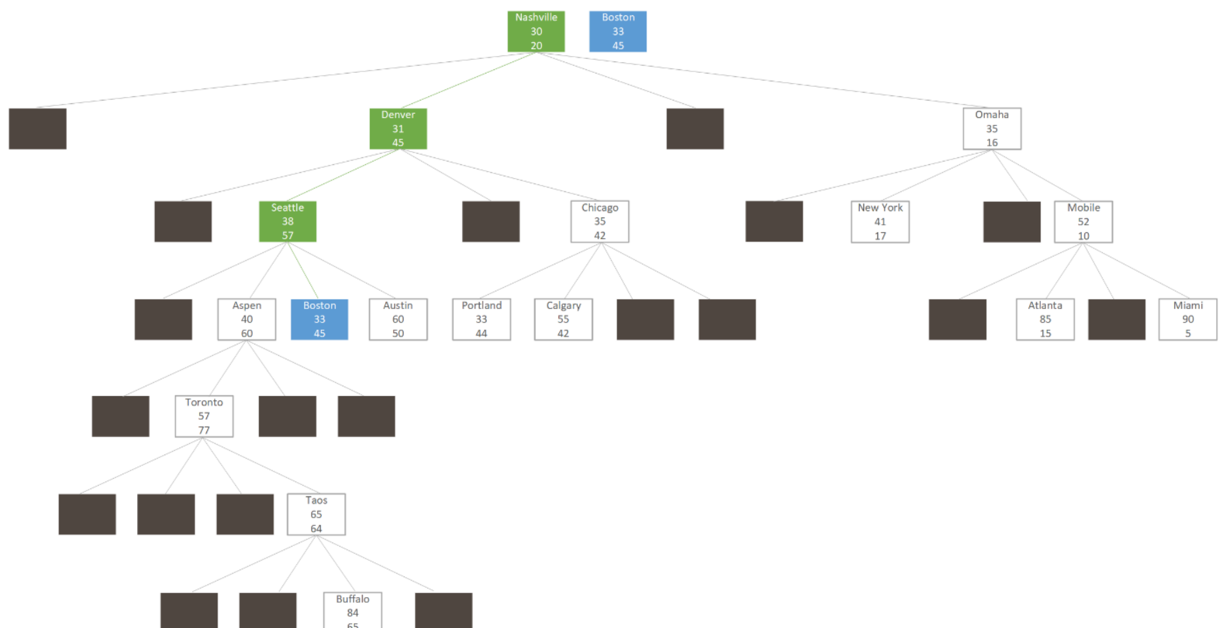
Obrázek 23 Vybudovaný Bodový Quad Strom, zdroj: [vlastní]

Pseudokód vybudování struktury

```
1. vybuduj(vstupniData)
2. zacatek
3.   jestliže (!jePrazdny()) || (delka(vstupniData) == 0) pak konecOperace;
4.   VstupniData := seradDataPodleZOrderHodnot(vstupniData);
5.   pro i od 1 do delka(vstupniData) - 1 opakuj
6.   zacatek
7.     vloz(vstupniData[i],root,0);
8.   konec
9. konec
```

3.5.3.2 Operace vlož

Na rozdíl od ostatních implementací Quad Stromu, kde dochází k tvoření regiónu pomocí půlení intervalu v této implementaci, rozdělujeme regiony pomocí konkrétního bodu prvku. Analogicky se dá označit operace *Vlož* za stejnou jako operace v binárním vyhledávacím stromu s rozdílem určování pozice prvku. V binárním vyhledávacím stromu existují pouze dva potomci, kteří jsou označováni jako levý a pravý potomek prvku, přičemž v Quad stromu označujeme potomky regiony, které jsou vždy 4. Podle obou klíčových atributů vkládaného prvku je určeno, do jakého z jednoho ze čtyř kvadrantů bude prvek vložen. Pokud je určený region prázdný, operace je ukončena a prvek přidán, jinak operace rekurzivně pokračuje s aktuálním prvkem v určeném regionu.
Vložení prvku Boston do původní struktury Bodového Quad stromu přibližuje Obrázek 24, kde je zelenou barvou označena cesta ke správnému místu umístění a modrou vkládaný prvek.



Obrázek 24 Vložení prvku do Bodového Quad stromu, zdroj: [vlastní]

Nejprve před představením operace *Vlož*, bude ukázán algoritmus pro zjištění regionu při průchodu stromem. Tento algoritmus bude využit při operacích *Vlož*, *Odeber* i *Najdi*.

```
1.  zjistiRegion (vkladanyPrvek, otecPrvku)
2.  zacatek
3.    jestlize(vkladanyPrvek.X >= otecPrvku.X) pak // 2 x 4 region
4.    zacatek
5.      jestlize(vkladanyPrvek.Y >= otecPrvku.Y) pak
6.        konecOperace 2; //severovychodni region
7.      jinak //4
8.        konecOperace 4; //jihovychodni region
9.     konec
10.    jinak // 1 x 3 region
11.    zacatek
12.      jestlize(vkladanyPrvek.Y >= otecPrvku.Y) pak
13.        konecOperace 1; //severozapadni region
14.      jinak
15.        konecOperace 3; //jihozapadni region
16.    konec
17.  konec
```

Následuje algoritmus pro vložení prvku do Bodového Quad Stromu.

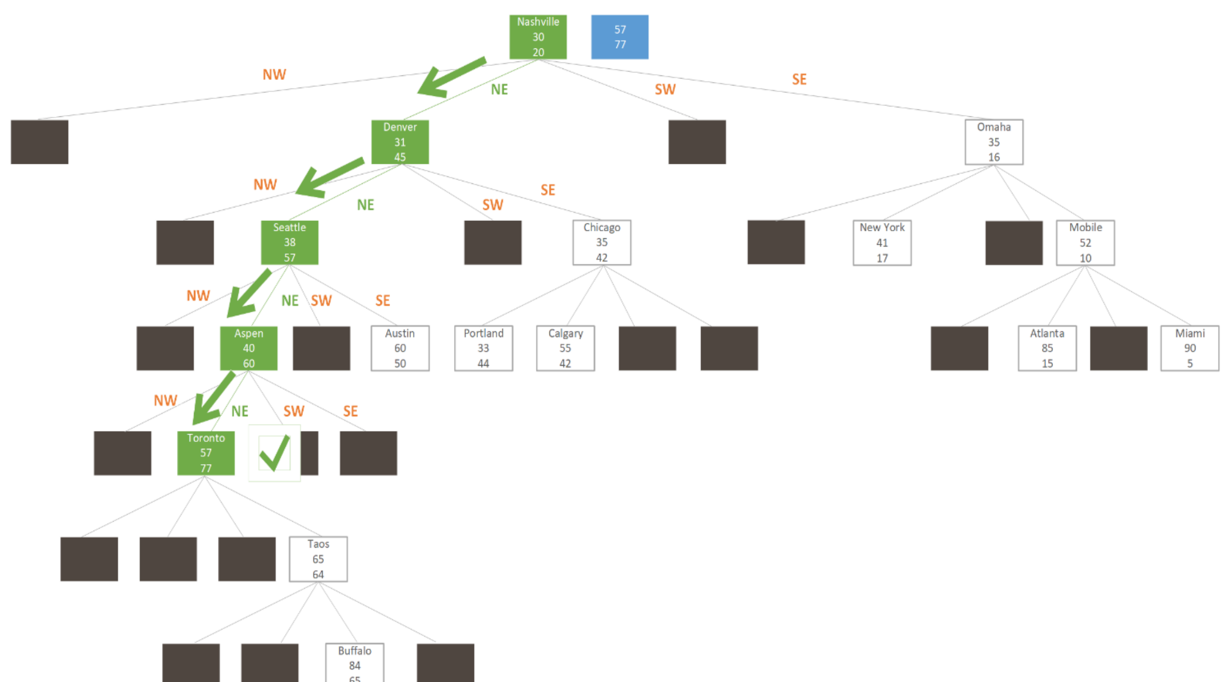
Pseudokód vložení prvku do struktury

```
1. vloz(vkladanyPrvek)
2. zacatek
3.   vlozenyPrvek := vlozPrvek(vkladanyPrvek, root, 0);
4.   jestlize vlozenyPrvek != null pak
5.   konecOperace vlozenyPrvek;
6. konec
7.
8.
9. vlozPrvek(vkladanyPrvek, otecPrvku, uroven)
10. zacatek
11.   //prazdny strom, prvek se stava korenem
12.   jestlize otecPrvku == null pak
13.   zacatek
14.     koren := vkladanyPrvek;
15.     pocetPrvku++;
16.     konecOperace koren;
17.   konec
18.
19.   indexRegionu := zjistiRegion(vkladanyPrvek, otecPrvku);
20.
21.   jestlize otecPrvku.Potomek[indexRegionu] == null pak
22.   zacatek
23.     otecPrvku.Potomek[indexRegionu]  := vytvorNovyList(vkladanyPrvek);
24.   konec
25.   jinak
26.   zacatek
27.   //jestlize ve stromu neni volne misto, traverzujeme pomoci
28.   //jiz zjisteneho regionu o uroven nize
29.     konecOperace vlozPrvek(otecPrvku.Potomek[indexRegionu], otecPrvku, uroven + 1);
30.   konec
31. konec
```

3.5.3.3 Operace najdi

Při operaci *Najdi* je postupováno od kořene stromu směrem k listům. Nejprve je zkoumáno, jestli hledané souřadnice neodpovídají aktuálně zpracovávanému prvku. Pokud souřadnice odpovídá, aktuálně zpracovaný prvek je označen jako hledaný prvek a je navrácen. Jestliže souřadnice neodpovídají, je pomocí obou souřadnic stejně jako při operaci *Vlož* určen region, do kterého budeme rekurzivně postupovat do doby, než je nalezen hledaný prvek a souřadnice odpovídají.

Operace Najdi je představena na následujícím Obrázku 25, kdy zeleně označený region znamená směr traverzování.



Obrázek 25 Hledání prvku v bodovém Quad Stromu, zdroj: [vlastní]

```
1.  najdi(hledanaSouradnice)
2.  zacatek
3.    nalezenyPrvek := najdiPrvek(hledanaSouradnice, root);
4.    konecOperace nalezenyPrvek;
5.  konec
6.
7.  najdiPrvek(hledanaSouradnice, aktualniPrvek)
8.  zacatek
9.    jestlize (aktualniPrvek == null) pak konecOperace null;
10.
11.   jestlize(hledanaSouradnice == aktualniPrvek.souradnice) pak
12.   zacatek
13.     //hledany prvek je nalezen, souradnice se shoduji
14.     konecOperace aktualniPrvek;
15.   konec;
16.   //traverzujeme stromem podle regionu
17.   indexRegionu := zjistiRegion(hledanaSouradnice, aktualniPrvek);
18.   konecOperace najdiPrvek(hledanaSouradnice, aktualniPrvek.Potomek[indexRegionu]);
19. konec
```

3.5.3.4 Operace odeber

Jak již bylo zmíněno v úvodu k Bodovému Quad stromu, efektivita dynamických operací byla překonána stromově datovou strukturou K-D stromem. Reálně využití této datové struktury vede ke statickým aplikacím a z tohoto důvodu dynamické operace povedou k přebudování stromové struktury stromu.
Operace bude ověřovat existenci hledaného prvku, kde můžou nastat dva různé scénáře. V prvním scénáři, pokud hledaný prvek bude listem stromu. prvek bude odebrán a struktura stromu nebude měněna. Jakmile naleznutý prvek nebude listem stromu, celá stromová struktura bude znova přebudována bez požadovaného prvku pomocí operace *Vybuduj*.
V této kapitole grafické představení operace *Odeber* nebude ukázáno. Jestliže prvek je listem, operace využívá operaci *Najdi* a poté prvek odebere, grafické zpracování Operace *Najdi* nalezneme v kapitole 3.5.3.3. Pokud prvek není listem, je struktura znovu vybudována podle kapitoly 3.5.3.1.

Pseudokód odebrání prvku ve struktuře

```
1. Odeber(hledanaSouradnice)
2. zacatek
3.   hledanyPrvek = najdi(hledanaSouradnice)
4.   jestlize hledanyPrvek != null) pak
5.     zacatek
6.       jestlize(JeListem(hledanyPrvek)) pak
7.         zacatek konecOperace odeberList(hledanyPrvek);
8.     konec
9.     jinak
10.     zacatek
11.       listPrvku: = prohlidka(koren);
12.       odeberPrvek(listPrvku, hledanyPrvek);
13.       //po odebrani prvku z listu, znovu vybuduj strukturu
14.       vybuduj(listPrvku);
15.     konec
16.    konec
17.   konecOperace null;
18. konec
```

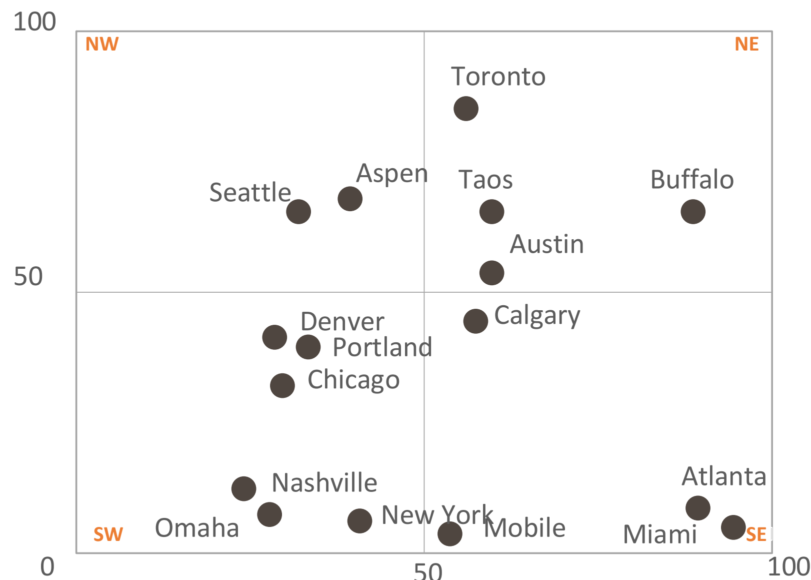
3.5.4 Bodově oblastní Quad strom

Při bodově oblastní implementaci na rozdíl od bodového Quad stromu dochází k dělení území na jednotlivé kvadranty pomocí metody půlení intervalu. Vyznačuje se tím, že datové prvky se nacházejí pouze na úrovni listů, obdobně jako tomu je u rozsahového stromu. Ostatní vrcholy stromu tvoří navigační strukturu se čtyřmi potomky (regiony). Tyto čtyři regiony sousedí s daným vrcholem, a pokud se v některém z nich nachází více než jeden prvek, je tento region dále cyklicky dělen [1][9].

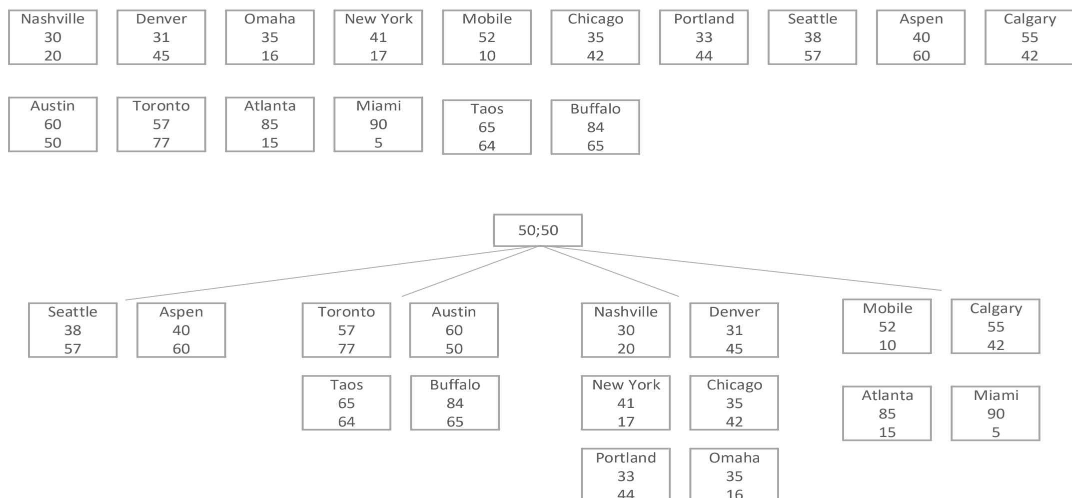
3.5.4.1 Operace vybuduj

Stejně jako u Bodového Quad stromu operace *Vybuduj* využívá operaci *Vlož* pro vybudování struktury. Na začátku operace jsou vstupní data seřazena pomoci Z-křivky, která zajišťuje efektivnější vkládání prvků do stromové struktury. Výhody seřazení dat pomocí Z-křivky jsou popsány v článku od Sameta [8]. Z-křivka, někdy označována jako Mortonův rozklad bude popsána v kapitole 4.2.

Při představení operace vybuduj byla hodnota vertikální i horizontální hranice zvolena hodnotou 100. Nejprve jsou data seřazena pomocí pořadí Z-křivky a poté jsou postupně pomocí operace Vlož a půlení intervalů vkládána do struktury. Na Obrázku 27 je zobrazena operace vybuduj, kde jsou prvky rozděleny do 4 regionů, které odpovídají dělení z Obrázku 26. Operace následně pokračuje do vybudování kompletní stromové struktury.

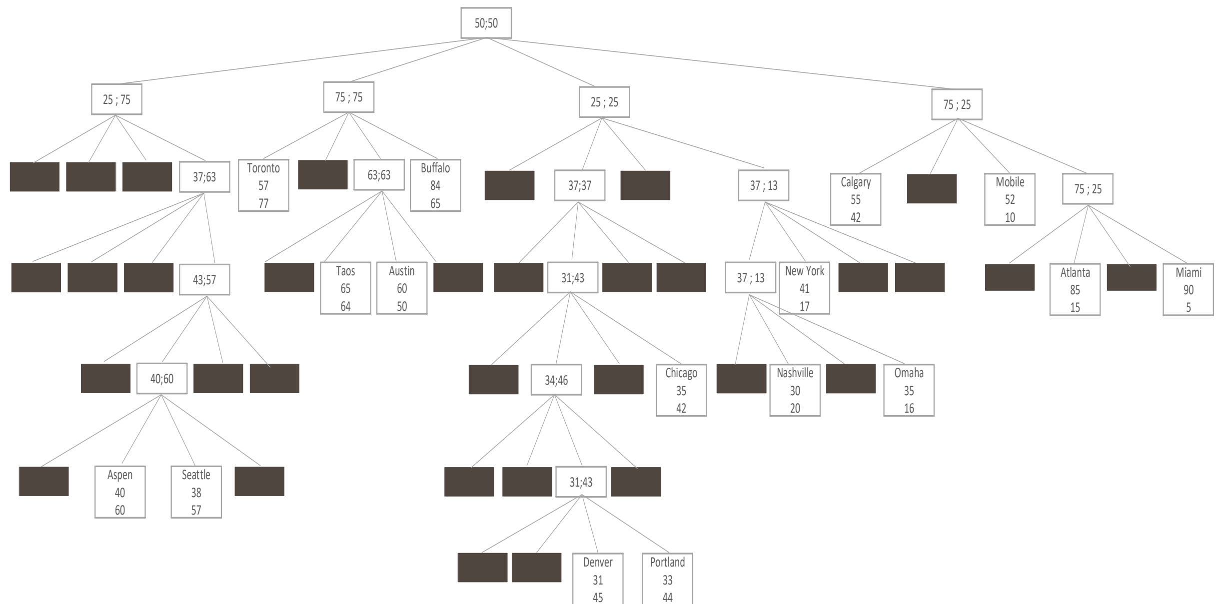


Obrázek 26 Dělení 2D prostoru pomocí půlení intervalů, zdroj: [vlastní]



Obrázek 27 Budování struktury Bodově oblastního Quad stromu, zdroj: [vlastní]

Výsledná struktura je prezentována na Obrázku 28. Zde můžeme vidět tři druhy prvků. Navigační prvek, prázdný prvek a datový prvek.



Obrázek 28 Vybudovaná struktura Bodově oblastního Quad Stromu, zdroj: [vlastní]

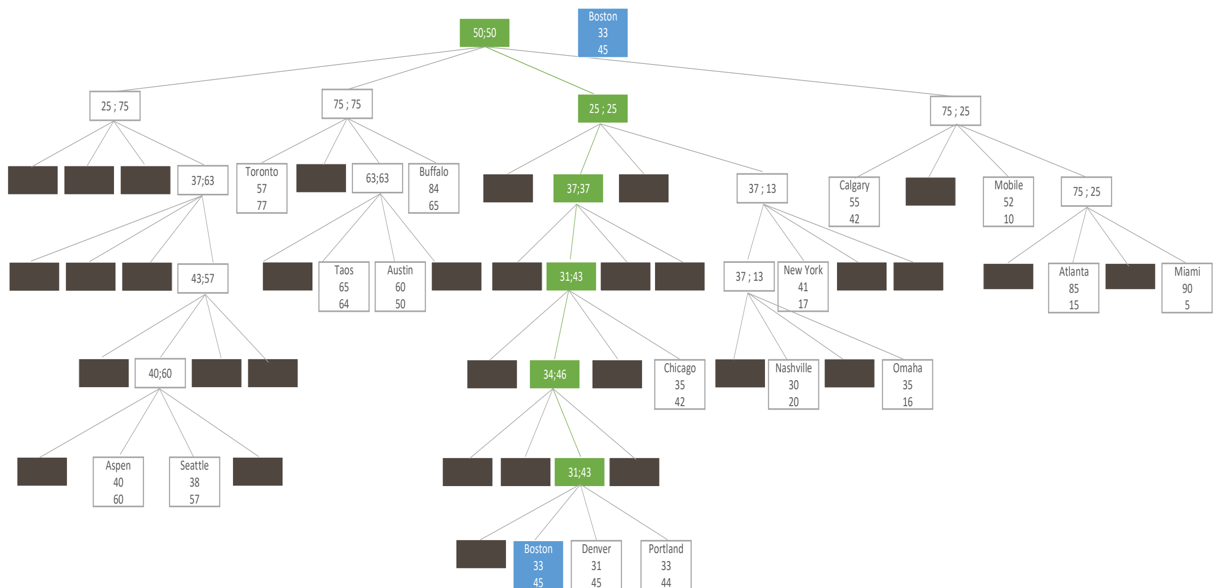
Pseudokód vybudování struktury

```
10.    vybuduj(vstupniData)
11.    zacatek
12.      jestliže (!jePrazdny()) || (delka(vstupniData) == 0) pak konecOperace;
13.      vstupniData := seradDataPodleZOrderHodnot(vstupniData);
14.      pro i od 1 do delka(vstupniData) - 1 opakuj
15.      zacatek
16.        vloz(vstupniData[i],koren,0);
17.      konec
18.    konec
```

3.5.4.2 Operace vlož

Nejprve je zjišťována naplněnost struktury. Jestliže je strom prázdný, prvek se stává kořenem stromu a operace je ukončena. Ve druhém případě se ve stromu nachází již nějaký prvek a operace probíhá směrem od kořene stromu k prázdnému místu ve stromě nebo k listu stromu. Postupně jak je strom procházen je rozhodnuto, do kterého regionu bude algoritmus pokračovat. Podmínky pro rozhodnutí jsou ukázány v pseudokódu. Jestliže traverzování dojde k listu stromu, aktuální list se stane navigačním prvkem a pomocí půlení intervalu jsou určeny obě souřadnice nového navigačního prvku a pro nový navigační prvek je rekurzivně aplikována operace *Vlož* pro původní list stromu a vkládaný prvek.

Obrázek 29 představuje operaci *Vlož*, kde modrou je označen nově vložený prvek a zelenou cesta, kterou prvek traverzoval od kořene stromu na místo, které odpovídá podmínkám operace.



Obrázek 29 Stav bodově oblastního stromu po vložení prvku, zdroj: [vlastní]

Před představením operace *Vlož*, bude ukázán algoritmus pro zjištění regionu při průchodu stromem. Tento algoritmus bude využit při operacích *Vlož*, *Odeber* i *Najdi*.

```
18. zjistiRegion (vkladanyPrvek, otecPrvku)
19. zacatek
20.   jestlize(vkladanyPrvek.X >= otecPrvku.X) pak // 2 x 4 region
21.   zacatek
22.     jestlize(vkladanyPrvek.Y >= otecPrvku.Y) pak
23.       konecOperace 2; //severovychodni region
24.     jinak //4
25.       konecOperace 4; //jihovychodni region
26.    konec
27.    jinak // 1 x 3 region
28.    zacatek
29.      jestlize(vkladanyPrvek.Y >= otecPrvku.Y) pak
30.        konecOperace 1; //severozapadni region
31.      jinak
32.        konecOperace 3; //jihozapadni region
33.    konec
34.  konec
```

Pseudokód vložení prvku do struktury

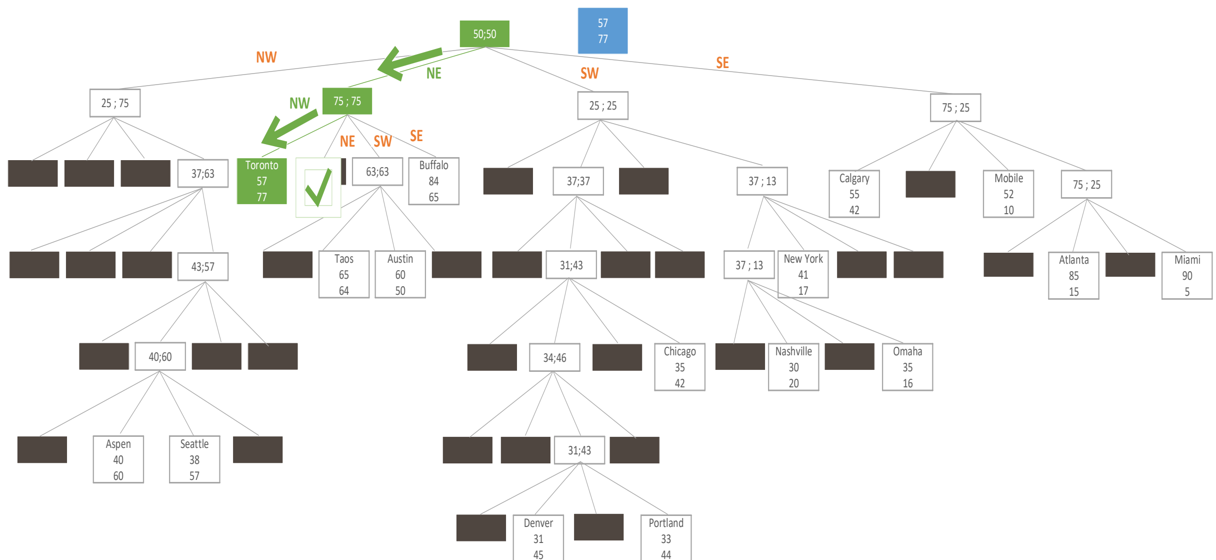
```
1.  vloz(vkladanyPrvek)
2.  zacatek
3.    vlozenyPrvek := vlozPrvek(vkladanyPrvek, koren, 0);
4.    jestlize vlozenyPrvek != null pak
5.      konecOperace vlozenyPrvek;
6.  konec
7.
8.
9.  vlozPrvek(vkladanyPrvek, aktualniPrvek, uroven)
10. zacatek
11.   //prazdny strom, prvek se stava korenem
12.   jestlize aktualniPrvek == null pak
13.   zacatek
14.     koren := vkladanyPrvek;
15.     pocetPrvku++;
16.     konecOperace koren;
17.   konec
```

```
18.
19.   jestlize jeListem(aktualniPrvek) pak
20.   zacatek
21.     //nastane jenom pokud aktualniPrvek je koren a zaroven listem
22.     tempPrvek := aktualniPrvek;
23.     koren := vytvorNovyNavigacniPrvek(horizontalniHranice / 2, vertikalniHranice / 2);
24.     vlozPrvek(tempPrvek, aktualniPrvek.Potomek[indexRegionu], uroven + 1);
25.     konecOperace vlozPrvek(tempPrvek, vkladanyPrvek, uroven + 1);
26.   konec
27.   jinak
28.   zacatek
29.     indexRegionu := zjistiRegion(vkladanyPrvek, otecPrvku);
30.     //funkce vypocita souradnice pomoci puleni intervalu ve spravnem regionu
31.     xCord := vypocitejSouradnice_X(aktualniPrvek, indexRegionu, uroven,
32.                                    horizontalniHranice);
33.     yCord := vypocitejSouradnice_Y(aktualniPrvek, indexRegionu, uroven,
34.                                    vertikalniHranice);
35.     jestlize aktualniPrvek.Potomek[indexRegionu] == null pak
36.     zacatek
37.       aktualniPrvek.Potomek[indexRegionu] := vytvorNovyList(vkladanyPrvek);
38.     konec
39.     jinak
40.     zacatek
41.       jestlize JeListem(aktualniPrvek.Potomek[indexRegionu]) pak
42.       zacatek
43.         //vytvorime novy navigacni prvek a vkladanyPrvek a puvodni list
44.         // vlozime jako potomky
45.         tempPrvek := aktualniPrvek.Potomek[indexRegionu];
46.         aktualniPrvek.Potomek[indexRegionu] := vytvorNovyNavigacniPrvek(xCord, yCord);
47.         vlozPrvek(tempPrvek, aktualniPrvek.Potomek[indexRegionu], uroven + 1);
48.         konecOperace vlozPrvek(tempPrvek, vkladanyPrvek, uroven + 1);
49.       konec
50.       jinak
51.       zacatek
52.         //traverzujeme stromem do spravneho regionu
53.         konecOperace vlozPrvek(aktualniPrvek.Potomek[indexRegionu], vkladanyPrvek,
54.                                uroven + 1);
55.       konec
56.     konec
57. konec
```

3.5.4.3 Operace najdi

Operace *Najdi* podle zadaných souřadnic začíná směrem od kořenu stromu k listům. V každém navigačním vrcholu je nejprve zkoumáno, ve kterém z jeho regionů hledaný prvek leží a podle toho se určí následující směr hledání. Průchod struktury končí při dosažení datového prvku neboli listu. Jestliže souřadnice listu odpovídají zadaným souřadnicím, prvek je nalezen a navrácen. V opačném případě je jisté, že hledaný prvek se ve struktuře nenachází.

Operace najdi pro klíč 57;77 je představena na Obrázku 30.



Obrázek 30 Znázornění operace najdi v bodově oblastním quad stromu, zdroj: [vlastní]

Pseudokód hledání prvku ve struktuře

```
20. najdi(hledanaSouradnice)
21. zacatek
22.   nalezenyPrvek := najdiPrvek(hledanaSouradnice, koren);
23.   konecOperace nalezenyPrvek;
24. konec
25.
26. najdiPrvek(hledanaSouradnice, aktualniPrvek)
27. zacatek
28.   jestlize(jeListem(aktualniPrvek)) pak
29.   zacatek
30.     jestlize(hledanaSouradnice == aktualniPrvek.souradnice) pak
31.     zacatek
32.       //hledany prvek je nalezen, souradnice se shoduji
33.       konecOperace aktualniPrvek;
34.     konec;
35.     Jinak konecOprace null; //hledany prvek nenalezen
36.    konec;
37.
38.   //traverzujeme stromem podle regionu
39.   indexRegionu := zjistiRegion(hledanaSouradnice, aktualniPrvek);
40.   konecOperace najdiPrvek(hledanaSouradnice, aktualniPrvek.Potomek[indexRegionu]);
41. konec
```

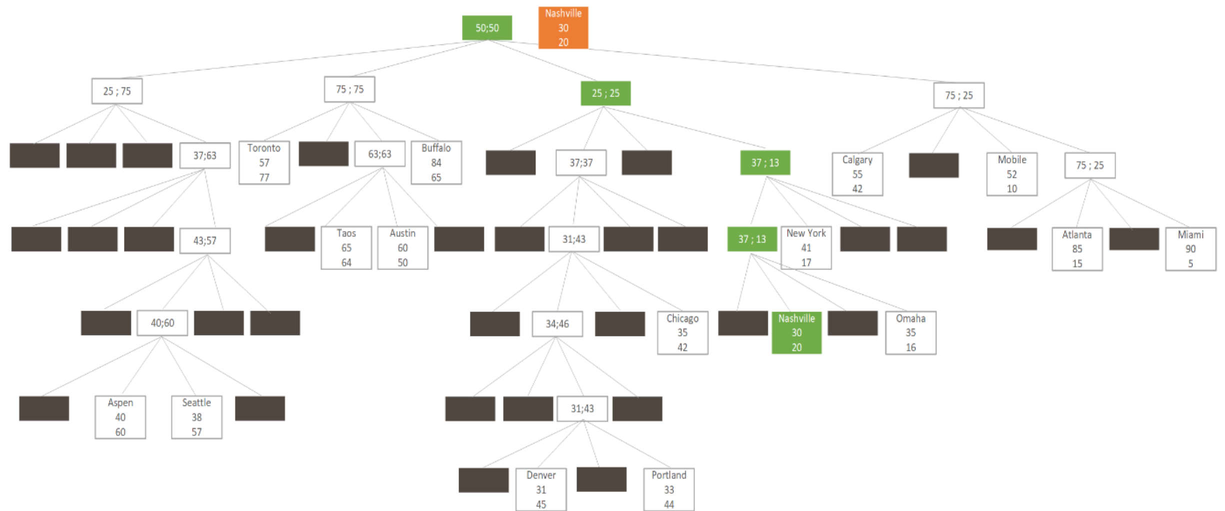
3.5.4.4 Operace odeber

Stejně jako v ostatních datových strukturách, při vykonávání operace *Odeber* je nejprve zjištěno, jestli se odebíraný prvek v datové struktuře nachází. Pokud tomu tak není, operace je ukončena.

V opačném případě je postupně traverzováno od kořene stromu po konkrétní hledaný list. Odebíraný prvek, je v této datové struktuře vždy list stromu a z tohoto důvodu může být prvek odebrán. Po odebrání prvku je zjištěno, kolik platných sourozenců po prvku zůstalo. Pokud po odebrání zůstal právě jeden sourozenec, struktura stromu musí být upravena. Otec

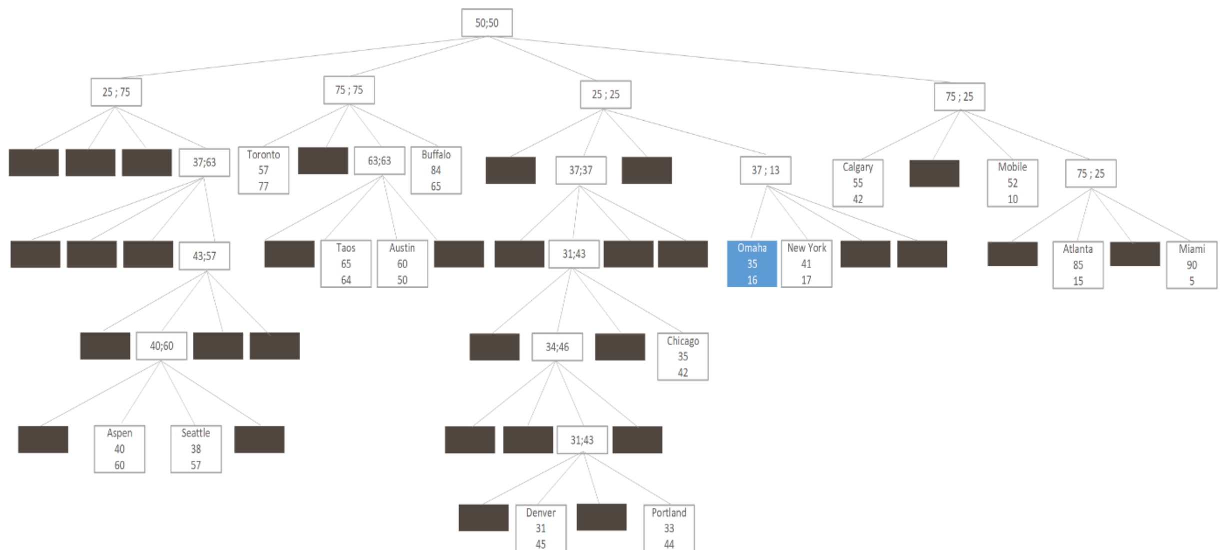
odebíraného prvku (navigační prvek) bude ze stromové struktury odebrán a na jeho místo bude přesunut poslední sourozenec odebraného prvku.

Grafické zpracování bude ukázáno při odstraňování prvku Nashville. V prvním kroku zpracování grafické operace (Obrázek 31) je nejprve nalezen odebíraný prvek Nashville.



Obrázek 31 Označení odebíraného prvku v Bodově oblastním Quad stromu, zdroj: [vlastní]

V následujícím kroku (Obrázek 32) je odebrán prvek Nashville a sourozenec prvku Omaha je přesunut místo navigačního prvku, který měl již posledního potomka.



Obrázek 32 Stav Bodově oblastního stromu po odebrání prvku, zdroj: [vlastní]

Pseudokód odebrání prvku ve struktuře

```
1. Odeber(hledanaSouradnice)
2. zacatek
3.   odebiranyPrvek := odeberPrvek(hledanaSouradnice, root, null, 0);
4.   jestlize(odebiranyPrvek != null) pak
5.   zacatek
6.     konecOperace odebiranyPrvek;
7.    konec
8. konec
9.
10. odeberPrvek(hledanaSouradnice, aktualniPrvek, otecPrvku, indexPrvku)
11. zacatek
12.   jestlize aktualniPrvek == null konecOperace null;
13.
14.   jestlize jeListem(aktualniPrvek) pak
15.   zacatek
16.     jestlize hledanaSouradnice == aktualniPrvek.souradnice pak
17.     zacatek
18.       jestlize otecPrvku.Potomek[indexPrvku] == aktualniPrvek pak
19.       zacatek
20.         //prvek nalezen a odebran;
21.         otecPrvku.Potomek[indexPrvku] := null;
22.         //pokud po odebrani prvku ma navigacniPrvek jen 1 platny vrchol,
23.         // navigacni prvek je odebran
24.         jestlize pocetPotomku(otecPrvku) == 1 pak
25.           odeberNeplatnyNavigacniPrvek(otecPrvku);
26.         konecOperace aktualniPrvek;
27.       konec
28.     konec
29.     jinak
30.     zacatek
31.       //prvek ve stromu nebyl nalezen, nelze odebrat
32.       konecOperace null;
33.     konec
34.   konec
35.   jinak
36.   zacatek
37.     //traverzovani stromem podle souradnic hledaneho prvku
38.     indexRegionu := zjistiRegion(hledanaSouradnice, aktualniPrvek);
39.     konecOperace odeberPrvek(hledanaSouradnice, aktualniPrvek.Potomek[indexRegionu],
40.                           aktualniPrvek, indexRegionu);
41.   konec
42. konec
```

4 LINEARNÍ PRŮCHODY PROSTOREM

Lineární průchody prostorem jsou v informatice využívány v různých oblastech, ale zejména tam, kde je důležité linearizovat vícerozměrná data. Vícerozměrná data mohou být například matice, obrazy, tabulky a jiné. Zpracování datových operací jako je násobení matic, budování struktur, úprava struktur lze zjednodušit volbou efektivního způsobu procházení dat pomocí některého z lineárních průchodů [14].

Průchody slouží jako způsob mapování multidimenzionálního prostoru do jednorozměrného. Fungují jako vlákno, které prochází každým buněčným prvkem v prostoru tak, aby každá buňka byla navštívena právě jednou. Výsledným průchodem křivky je lineárně seřazená sada dat v multidimenzionálním prostoru. Existuje velké množství způsobů, jak prostor procházet a některé způsoby budou rozebrány v následujících podkapitolách. Společné vlastnosti, které průchody splňují:

- patří mezi fraktály,
- jsou si podobné,
- jsou invariantní vůči velikosti,
- jsou nekonečně dlouhé.

V diplomové práci bude zkoumáno mapování pouze z 2D prostoru.

4.1 Peanova křivka

Peanova křivka objevena italským matematikem v roce 1890 Giuseppem Peanem byla první, která vyplňuje dvourozměrný prostor. V praxi je využívána například při převodu snímku ve stupních šedi do černobílého [12]. Konstrukce křivky v několika bodech:

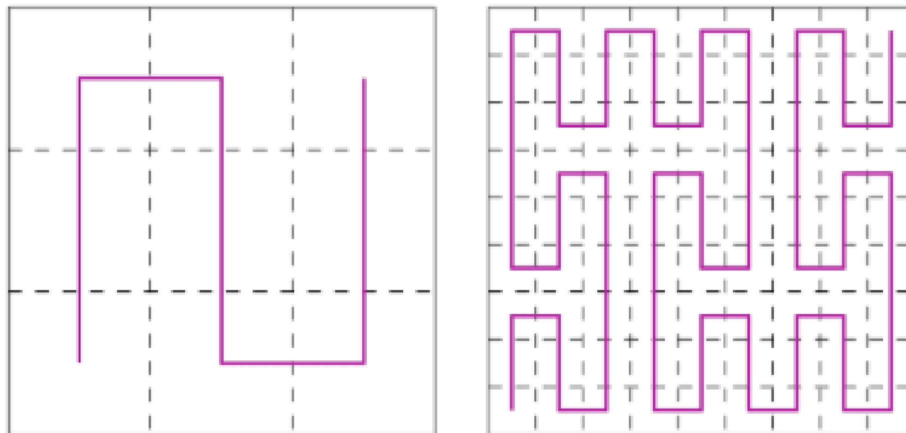
- Při každé iteraci křivky je každý sub region dělen na devět stejných částí.
- Každý sub region obsahuje vhodně transformovanou kopii Peanovy křivky.
- Každý sub region musí být situován tak, aby jednotlivé regiony byly spojeny do souvislé křivky. Pro situování regionu jsou využívány horizontální a vertikální kopie.

Při konstrukci existují různé varianty Peanovy křivky, které se liší určením počátečního bodu lineárního průchodu prostorem. Pro všechny varianty nicméně platí společné vlastnosti:

- v prostoru se nikdy neprotíná,

- invariantní vůči měřítku,
- nevyplňuje prostor neomezeně, ale vyplňuje prostor daný první iterací křivky [10].

Algoritmus pro výpočet souřadnic konkrétního bodu Peanovy křivky je přiložen do přílohy A. Algoritmus byl vypracován pomocí zdrojů [11] a veřejného repositáře společnosti Google umístěného na githubu [17].



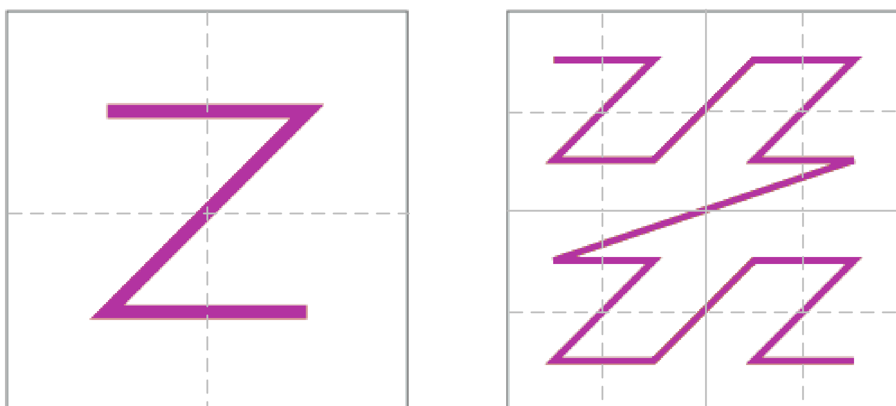
Obrázek 33 První dvě iterace Peanovy křivky, zdroj: [11]

4.2 Z-křivka

Z-křivka, která byla objevena a popsána v roce 1966 angličanem Guyem Macdonald Mortonem taktéž někdy označována jako Mortonův rozklad, je křivka vyplňující prostor, která udává lineární pořadí průchodu vícerozměrným prostorem. Křivka bývá využívána pří zefektivnění operace *Vybuduj* u Quad Stromu, nebo také R-Stromu. Křivku též využívají některé grafické karty k ukládání textur pro zvýšení prostorové referenční lokality během rasterizace [11]. Na rozdíl od ostatních lineárních průchodů prostorem lze hodnotu konkrétního bodu jednoduše vypočítat pomocí prokládání souřadnicových hodnot v binární reprezentaci. Vlastnosti křivky:

- křivka vyplňující prostor,
- nikdy se neprotíná,
- do každé liché hrany směřuje křivka „doprava"

Algoritmus pro výpočet souřadnic konkrétního bodu Z-křivky přiložen do přílohy B. Algoritmus byl vypracován pomocí zdrojů [11] a veřejného repositáře na githubu [18].

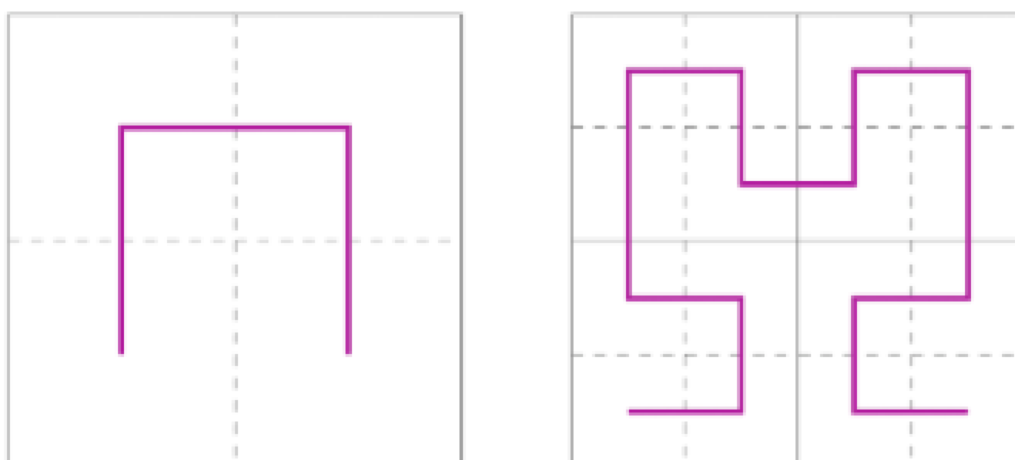


Obrázek 34 První dvě iterace Z-křivky, zdroj: [vlastní]

4.3 Hilbertova křivka

Hilbertova křivka, která byla objevena a pojmenována podle německého matematika Davida Hilberta v roce 1891 je dvourozměrná varianta Peanovy křivky. Na rozdíl od Peanovy křivky, která při každé iteraci (Obrázek 35) dělí sub regiony do devíti menších, dělí sub regiony po čtyřech. Bývá hojně využívána v databázích pro tvoření indexů prostorových databází, kde při hledání záznamu v blízké zeměpisné poloze mohou určit prioritu pro průzkum [13].

Algoritmus pro výpočet souřadnic konkrétního bodu Hilbertovy křivky je přiložen do přílohy C. Algoritmus byl vypracován pomocí zdrojů [11] a veřejného repositáře společnosti Google umístěného na githubu [17].



Obrázek 35 První dvě iterace Hilbertovy křivky, zdroj: [11]

5 IMPLEMENTACE

V následující kapitole budou nejprve popsány hlavní použité technologie při zpracování webové aplikace. Technologie budou obecně představeny a porovnány jejich výhody či nevýhody. Poté, bude ukázán návrh aplikace a na závěr budou popsány hlavní části implementace.

5.1 Použité technologie

Při zpracování praktické části webové aplikace pro vizualizaci evolucí algoritmů, byly použity jedny z nejpoužívanějších technologií v praktickém světě.

Pro backend, který se stará o srdce aplikace, byl využit nejpoužívanější java framework spring boot. Spring boot je moderní webový framework, který slouží k jednoduchému vytvoření samostatných aplikací. Též usnadňuje vystavení aplikací na aplikační server, poskytuje základní rozvržení konfiguračního Maven souboru a není potřeba žádná konfigurace pomocí XML [14].

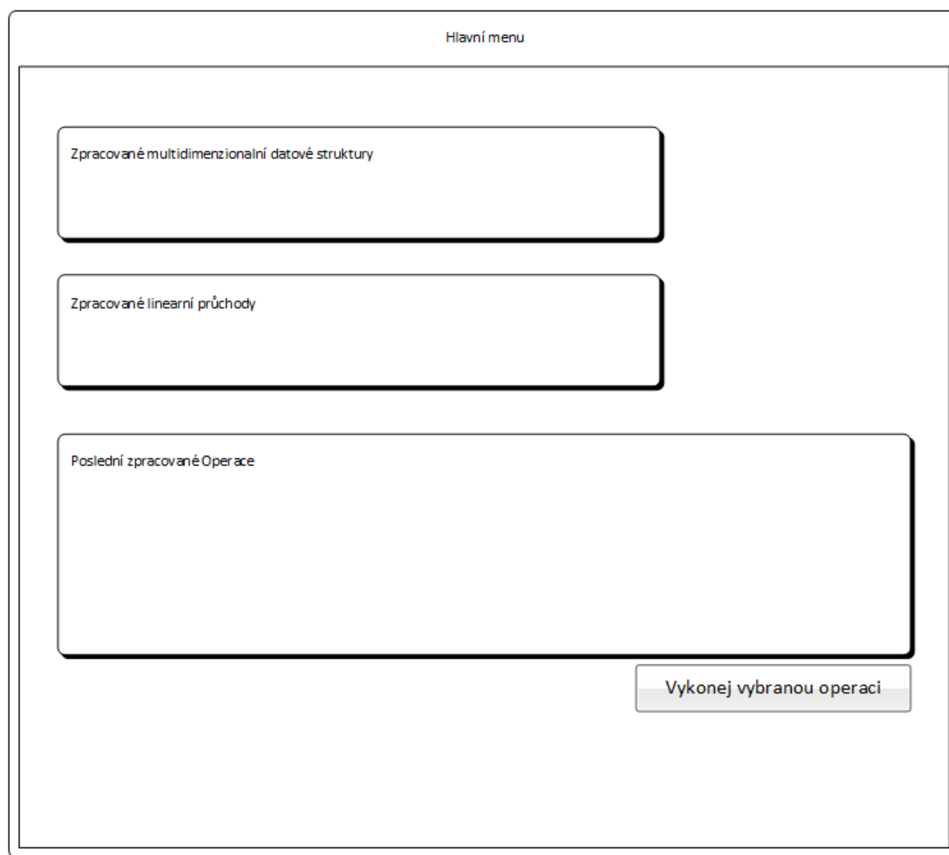
Pro tzv. frontend, byl použit také java framework Vaadin. Vaadin se využívá pro tvoření webových aplikací, které se chovají a zobrazují stejně jako desktopové aplikace. Naprogramovaný java kód je za pomocí překladače GWT překládán do JavaScriptu a ten je následně interpretován v internetovém prohlížeči. Jednou z výhod frameworku, je dostupný designer pro tvorbu formulářů bez znalosti webových programovacích jazyků jako je HTML a CSS.

Jednou z hlavních částí výsledné webové aplikace bylo vytvoření animační části evoluce algoritmů. Pro zpracování byla využita obalovací komponenta pro JavaScriptovou knihovnu Processing.Js. Knihovna slouží pro tvorbu vizualizací, digitálního umění a interaktivních animací ve webovém prohlížeči. Výhodou použití obalovací komponenty je programování javovského kódu, který je následně kompilován do JavaScriptu. Nevýhodou tohoto zpracování je možnost v Javě využít pouze některých dostupných tříd z balíčku GWT a čas kompilace při vývoji animační částí [15].
Pro práci s databází byla využita technologická část java frameworku spring konkrétně technologie Hibernate. Technologie zajišťuje objektové mapování a dotazování do databáze, bez nutnosti použití SQL jazyk. Kromě toho podporuje například lazy-loading, který je při práci s větší sadou dat velice užitečný.

5.2 Uživatelské prostředí

Navrhovanou webovou aplikaci by bylo vhodné rozdělit do třech kategorií. První kategorií by byla úvodní stránka aplikace. Druhou kategorií, již konkrétní pohledy zpracovaných funkčností a poslední třetí kategorií jsou vyskakovací formuláře.

Na Obrázku 36, můžeme vidět úvodní obrazovku webové aplikace. Na vrchu formuláře, se nachází hlavní menu, ve kterém jsou všechny zpracované funkčnosti aplikace. Pod hlavním menu, se nachází základní informace o webové aplikaci a přehled kategorizovaných vypracovaných funkčností. Poslední částí úvodní obrazovky jsou poslední zpracované operace. Webová aplikace ukládá posledních 10 vykonaných operacích na datových strukturách, které lze znovu jednoduše opakovat. Při vybrání operace, lze zobrazit detail operace, kde můžeme vidět informace o operaci, typu datové struktury a jeho prvky.



Obrázek 36 Uživatelské prostředí, zdroj: [vlastní]

Na dalším Obrázku 37 můžeme vidět již kreslící pohled. Pro všechny struktury je pohled stejný. Na vrchu formuláře se nachází hlavní menu, pro výběr jiné funkčnosti nebo návrat na úvodní obrazovku. Pod hlavním menu je co největší kreslící plátno, které lze vytvořit pro využití co největší možné kapacity obrazovky. Pod plátnem se nachází, již jen tlačítko pro vybrání operace u struktury.

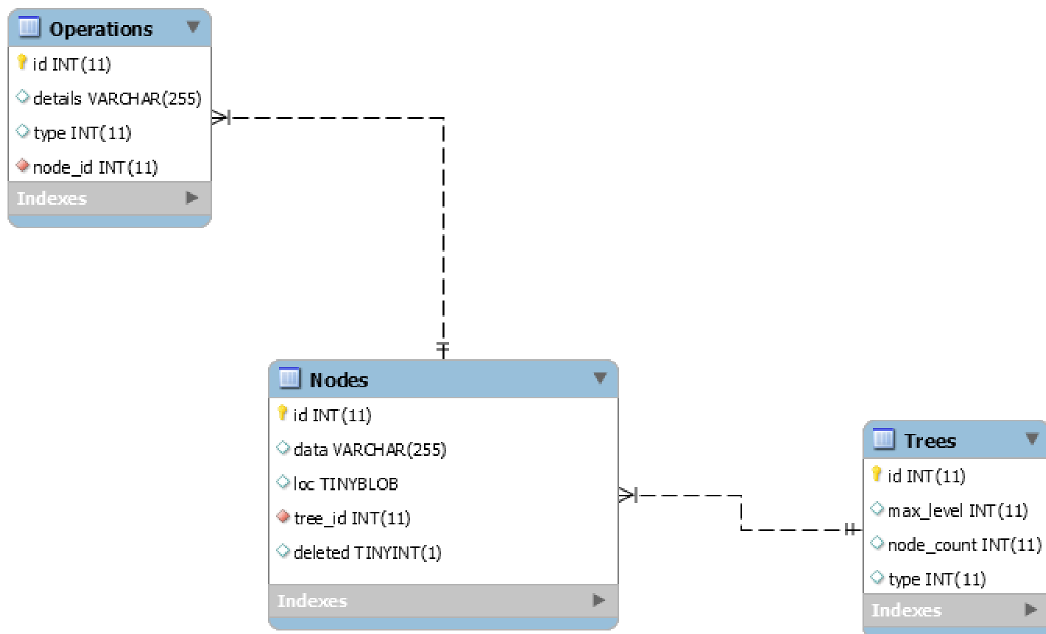
Obrázek 37 Kreslící plátno, zdroj: [vlastní]

Poslední částí vizuální části aplikace jsou vyskakovací formuláře, které jsou vyvolány při zobrazení detailu stromu, vytvoření datové struktury z předem vypracované sady dat a editace a vykonání operace na datové struktuře.

5.3 Databáze

Na následujícím Obrázku 38 je zobrazena definice ukládání zpracovaných operací ve webové aplikaci. Model zobrazuje vazbu mezi operacemi, prvky stromu a stromem. Model se skládá ze tří entit a dvou vazeb. Struktura databáze, byla vytvořena technologií Hibernate, která z předem vytvořených objektů vytvořila databázový model s vazbami.

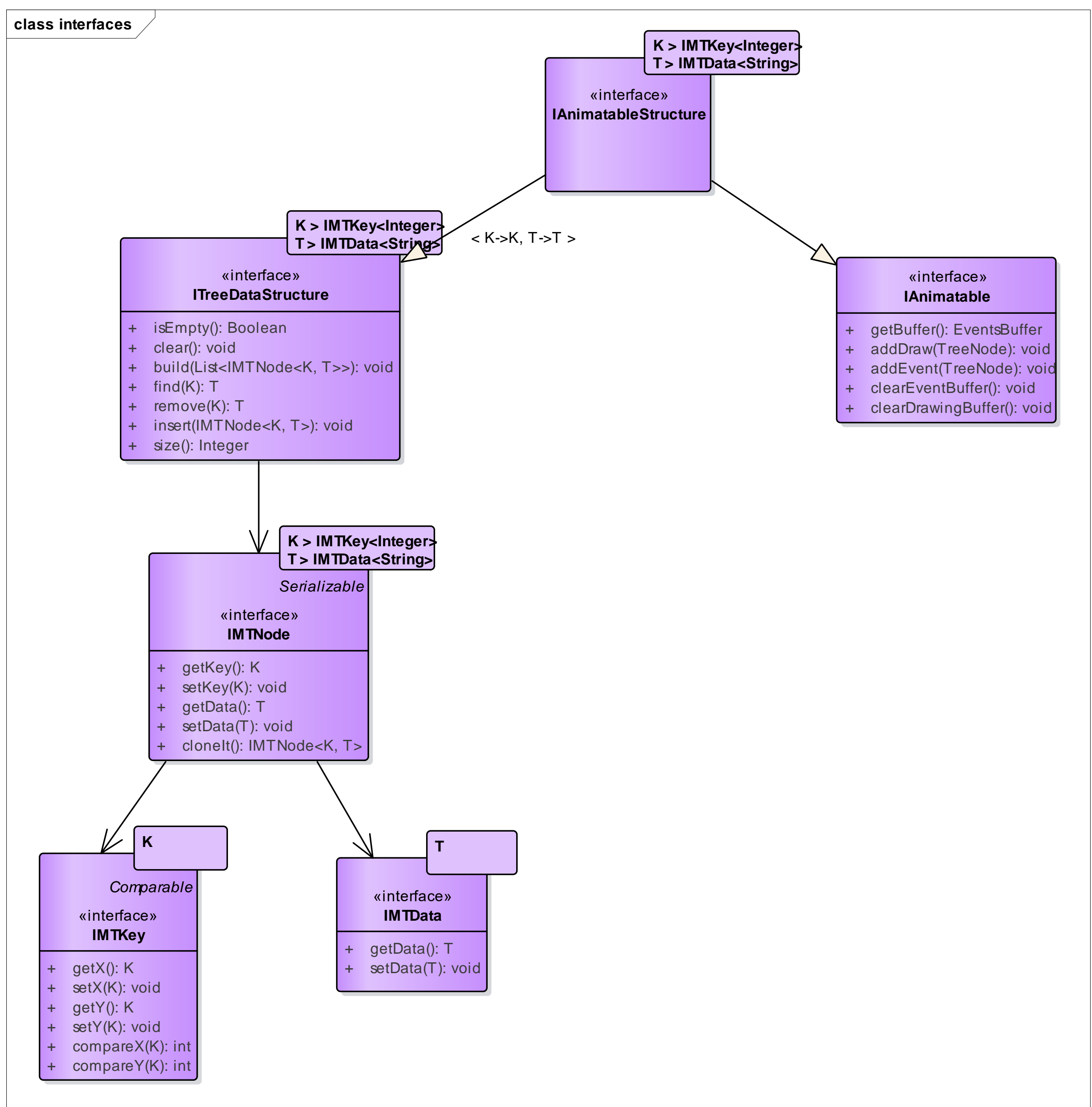
Pro testovací účely byla testovaná aplikace vystavena na platformě PaaS Heroku, kde je architektura databáze pro volné použití PostgreSQL.



Obrázek 38 Návrh databáze, zdroj: [vlastní]

5.4 Návrh tříd

V návrhu tříd budou popsány třídy pro implementaci datových struktur uchovávající multidimenzionální bodová data. Všechny implementované datové struktury jsou naimplementovány pomocí sad rozhraní, které jdou rozdělit do dvou kategorii. Rozhraní jsou ukázána na Obrázku 39. První skupina uvozuje komunikační kanál pro společné operace nad strukturou a zpracovávaným prvkem. Druhá sada, kde je pouze jedno rozhraní IAnimatable, zajišťuje komunikační kanál pro vizuální zpracování hlavních operací nad implementací.

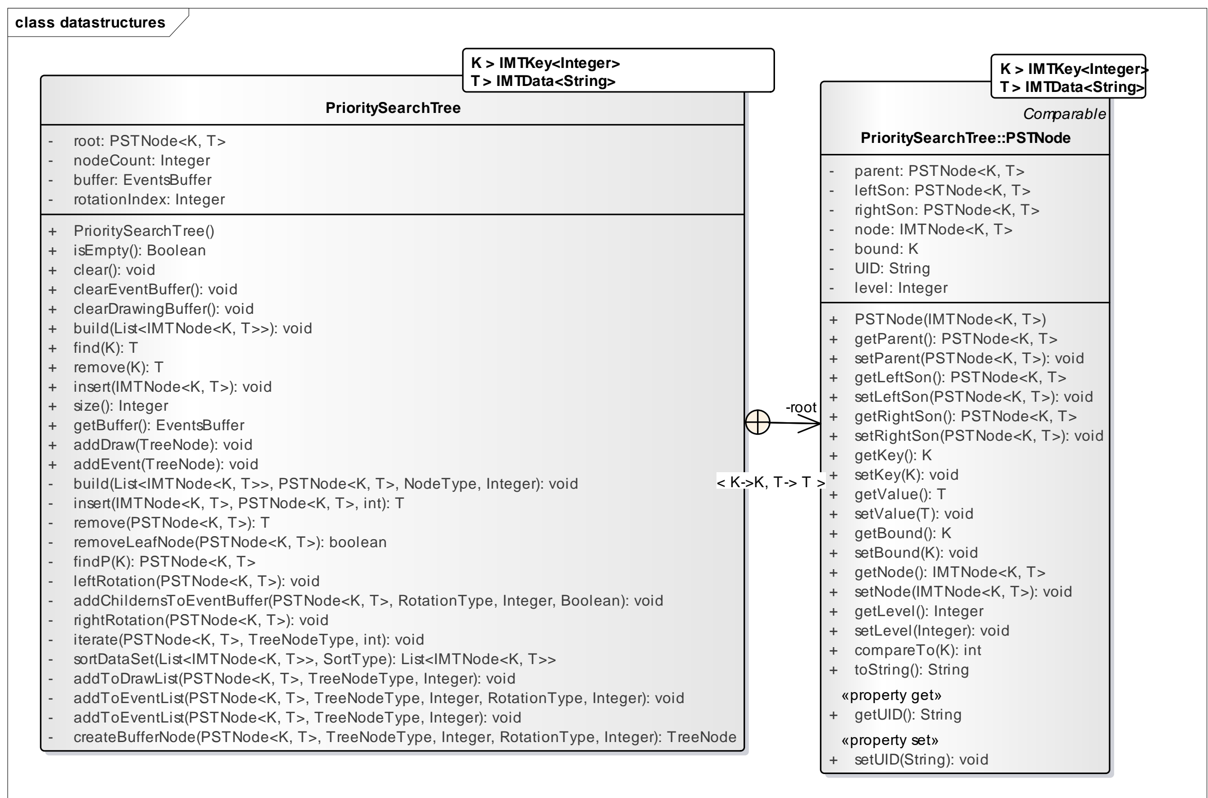


Obrázek 39 Rozhraní pro datové struktury, zdroj: [vlastní]

5.4.1 Třídy pro prioritní vyhledávací strom

Pro datovou strukturu prioritní vyhledávací strom byla navržena třída *PrioritySearchTree* a vnitřní třída *PSTNode*, která implementuje prvek struktury. Atributy, kterými disponuje třída *PrioritySearchTree* jsou *root* jako kořen struktury, *nodeCount* neboli počet prvků ve struktuře, buffer pro vizualizaci a pomocný atribut *rotationIndex*, který slouží pro identifikaci následující rotace při vizuálním zpracování struktury. Třída *PSTNode* obsahuje atributy *parent*, *leftSon* a *rightSon*, které obsahují reference na další prvky stromu. Dalším atributem je node, který obsahuje klíč a datovou část prvku. Posledním funkčním atributem je atribut *bound*, který označuje hranici prvku, slouží jako pomocný atribut při budování struktury a zajišťuje možnost vybudování vyvážené datové struktury. Poslední dva atributy *UID* a *level* jsou pomocné a slouží ve vizualizační části výsledné aplikace. Třída a její atributy operace je zobrazena na Obrázku 40.

Implementované operace prioritního vyhledávacího stromu jsou popsány v pseudokódu v kapitole 3.3 v teoretické části diplomové práce.



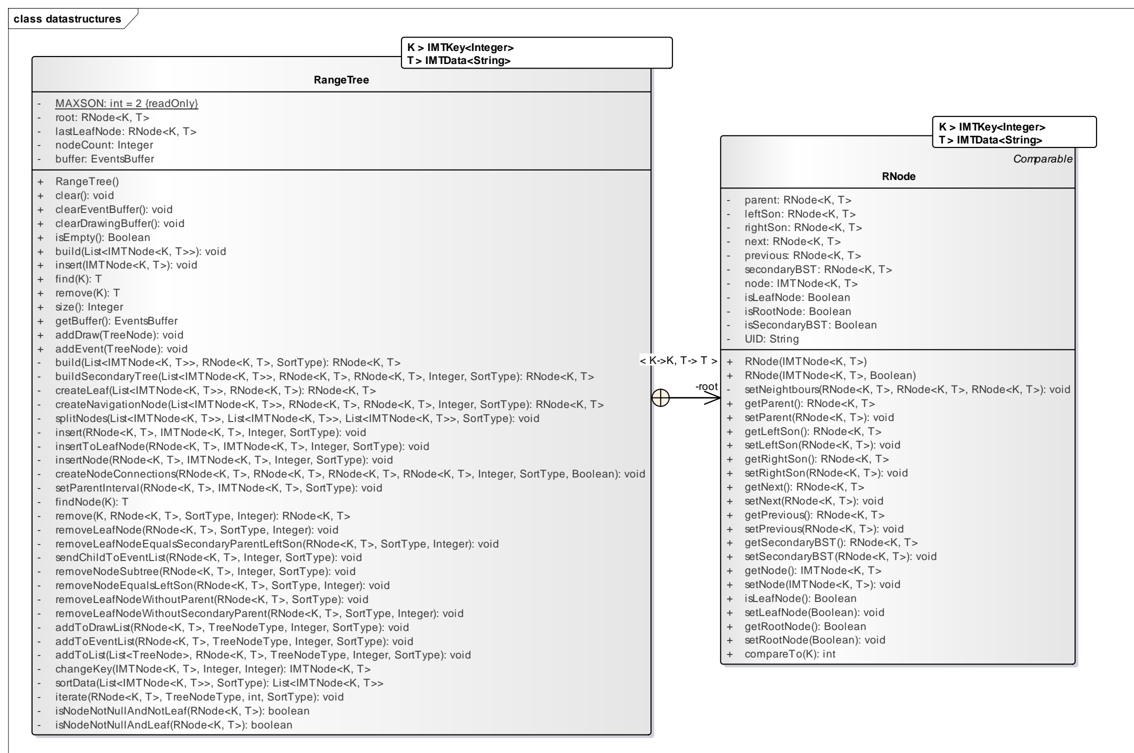
Obrázek 40 Třídy pro PST, zdroj: [vlastní]

5.4.2 Třídy pro rozsahový strom

Pro datovou strukturu rozsahový strom byla implementována třída *RangeTree* s vnitřní třídou *RNode*. Implementace obsahuje všechny atributy jako třída *PrioritySearchTree* a navíc byl pouze přidán atribut *lastLeafNode*, který značí poslední přidaný list do struktury. Vnitřní třída pro reference mezi ostatními prvky struktury obsahuje základní atributy *parent*, *leftSon*, *rightSon*, ke kterým byly přidány atributy *next*, *previous*. Přidané atributy slouží k zřetězení seznamu na úrovni listů. Další atributy *isLeafNode*, *isRootNode*, *isSecondaryBST* se používají k jednodušší identifikaci vybraného prvku. Posledním atributem je *secondaryBST*, který drží referenci na strom druhé dimenze. Atribut je využíván pouze v navigačních prvcích stromu. Návrh tříd je zobrazen na Obrázku 41.

Pomocné atributy pro identifikaci byly přidány z důvodu rozlišení navigačních a datových prvků ve struktuře. Navigační vrcholy neuchovávají klíče souřadnic x a y, ale uchovávají interval v rámci jedné z vybraných souřadnic. Interval označuje všechny prvky, které se nachází v podstromu vybraného prvku.

Implementované operace rozsahového stromu jsou popsány v pseudokódu v kapitole 3.3 v teoretické části diplomové práce.

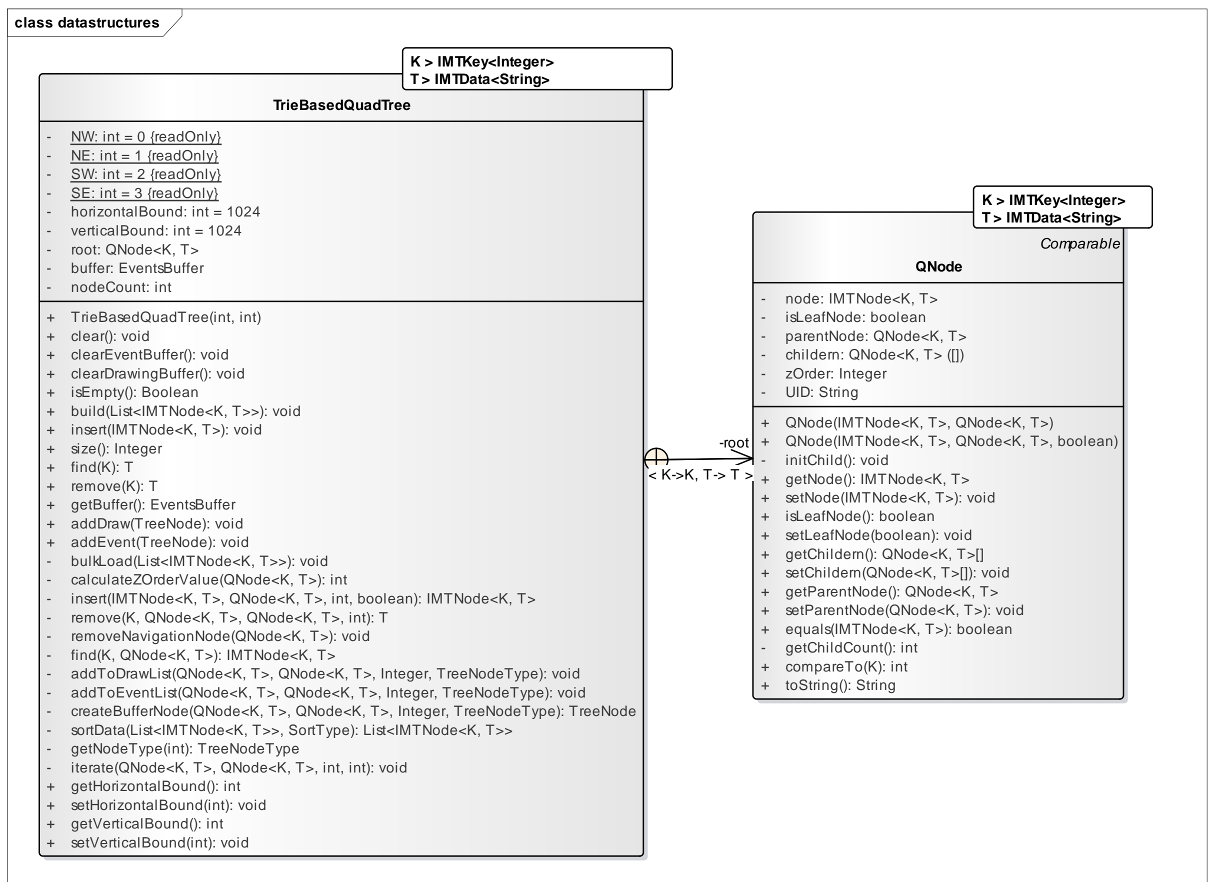


Obrázek 41 Třída pro rozsahový strom, zdroj: [vlastní]

5.4.3 Třídy pro Bodový Quad strom

Na rozdíl od předchozích zpracovaných implementací se Quad Strom vyznačuje tím, že každý prvek má vždy přiřazené čtyři syny, pokud prvek není listem. Třída pro implementaci bodového Quad Stromu je *PointBasedQuadTree* s vnitřní třídou pro prvek QNode. Globální konstanty slouží k identifikaci regionu uloženého prvku. Další atributy jsou *root* pro odkaz na kořen, *nodeCount* pro počet prvků a *buffer* pro vizualizace. Třída *QNode* obsahuje atribut node pro uchování dat a klíče, *isLeafNode* pro identifikaci, jestli je prvek list, *zOrder* pro označení hodnoty Z-křivky. Posledním atributem *childern* je statické pole o velikosti čtyři pro uchování odkazu ke všem synům prvku. Návrh tříd je zobrazen na Obrázku 42.

Implementované operace bodového Quad stromu jsou popsány v pseudokódu v kapitole 3.4.1 v teoretické části diplomové práce.

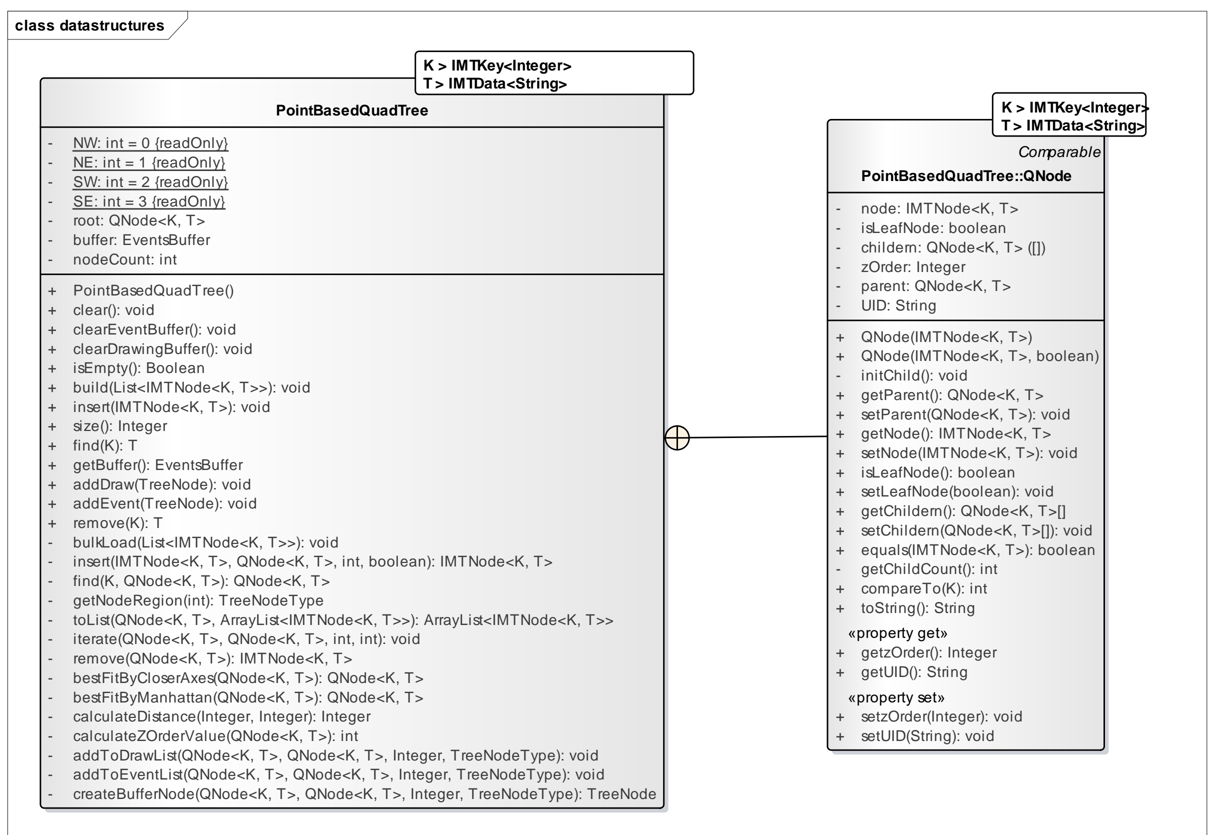


Obrázek 42 Třída pro bodový quad strom, zdroj: [vlastní]

5.4.4 Třídy pro Bodově oblastní Quad Strom

Rozložení Bodově oblastního Quad Stromu je závislé na volbě vertikální a horizontální hranice dělení prvků. Při operaci vkládání či operaci Vybudování je struktura tvořena pomocí půlení intervalu vybraného regionu a z tohoto důvodu volba hranice ovlivňuje kořen stromu a všechny jeho prvky. Třída TrieBasedQuadTree disponuje atributy pro *horizontalni* a *vertikalni* hranici, odkaz na *kořen*, *nodeCount* pro počet prvků a buffer pro tvorbu vizualizace. Vnitřní třída QNode je stejná jako při implementaci bodového Quad Stromu.

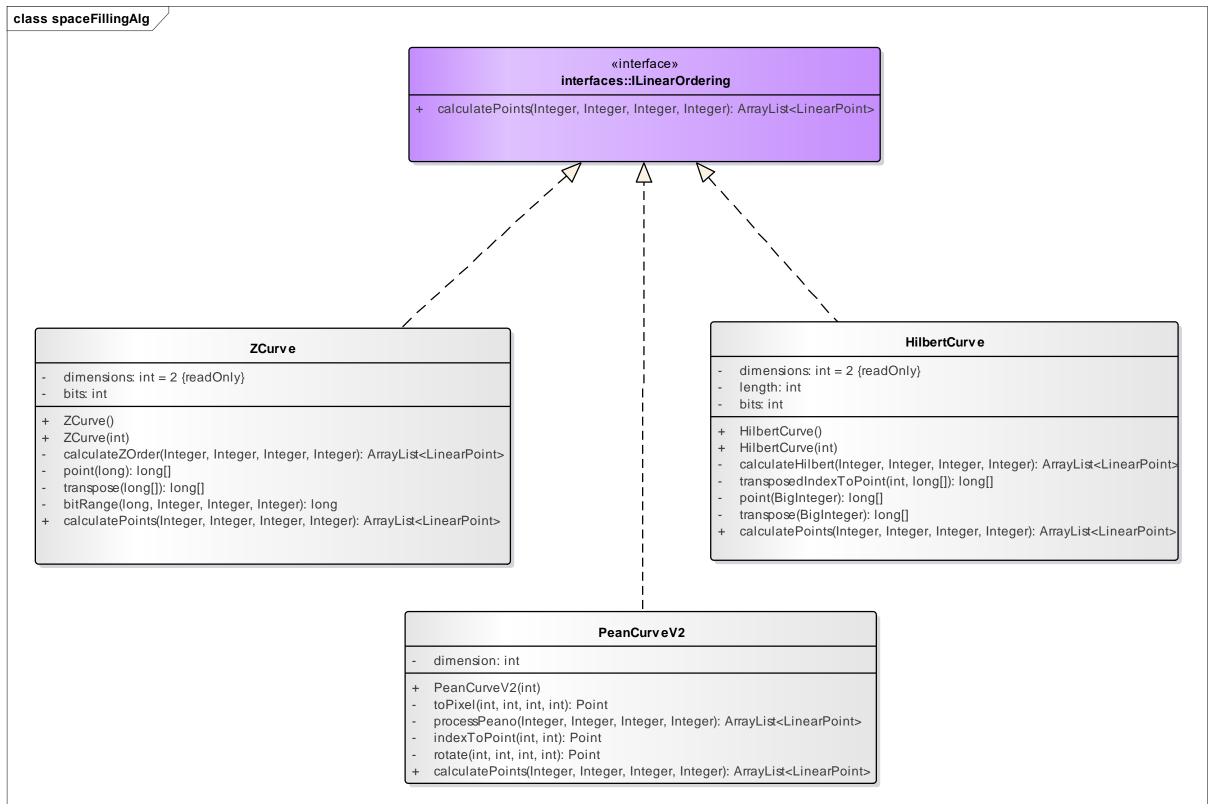
Implementované operace bodově oblastního QuadStromu stromu jsou popsány v pseudokódu v kapitole 3.4.2 v teoretické části diplomové práce.



Obrázek 43 Třída pro bodově oblastní Quad strom, zdroj: [vlastní]

5.4.5 Třídy pro lineární průchody prostorem

Tři třídy pro lineární průchody prostorem implementují rozhraní ILinearOrdering. Jediná veřejná metoda pro komunikaci je metoda calculatePoints se vstupními parametry pro velikost plátna *width* a *height*, *bits* které označují počet regionů a *marginem* pro určení velikosti okraje při vykreslení na plátno. Metoda vrací list seřazených bodů pro následné vykreslení lineárních bodů prostorem na 2D plátně.

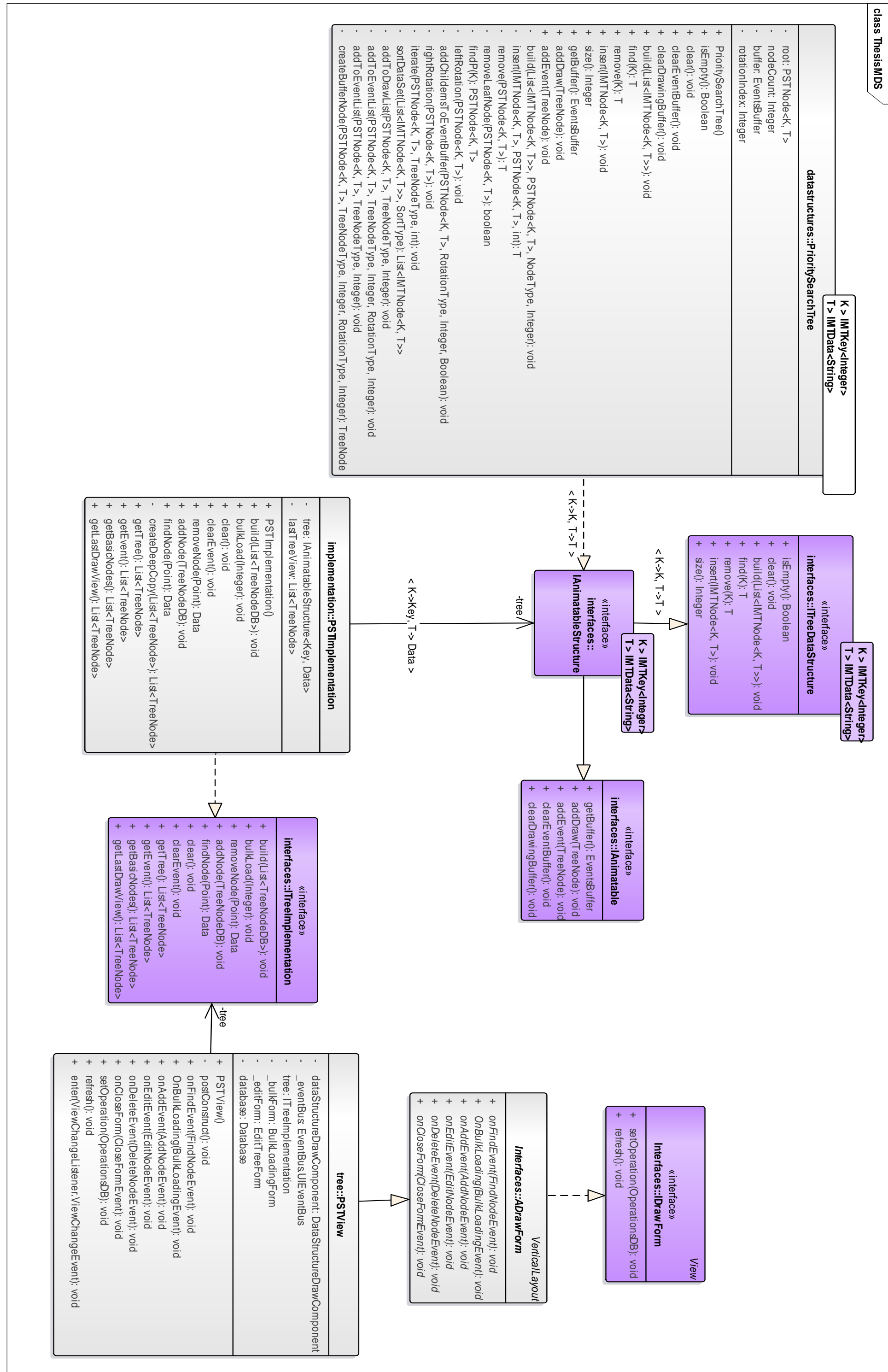


Obrázek 44 Třídy pro linearní průchody prostorem, zdroj: [vlastní]

5.4.6 Třídy pro komunikaci mezi backendem a frontendem

Na následujícím UML diagramu (Obrázek 45) je ukázána komunikace mezi naimplementovanými datovými strukturami a pohledy webové aplikace. Konkrétní příklad znázorňuje komunikaci prioritního vyhledávacího stromu a pohledu pro vykreslování. Komunikace je postavena na stylu programování proti rozhraní a z tohoto důvodu jsou pří ostatních implementacích vyměněny pouze třídy *PrioritySearchTree* a *PSTImplementation*. Po vykonání některé z operací je pomocí rozhraní předána informace o stavu struktury a všech krocích, které potřebujeme vědět pro správnou vizualizaci struktury. Surová data jsou zpracována a pomocí jsonu předána

komponentě *DataStructureDrawComponent*, která zapouzdřuje JavaScriptovou knihovnu Processing.JS pro tvorbu animací.



Obrázek 45 Ukázka komunikace backend-frontend, zdroj: [vlastní]

ZÁVĚR

Tématem diplomové práce byla vizualizace evolucí algoritmů různých datových struktur uchovávající multidimenzionální bodová data a vizualizace lineárních průchodů v prostoru konkrétně v prostoru dvoudimenzionálním.

V teoretické částí byly nejprve představeny obecně datové struktury pro uchovávání multidimenzionálních bodových dat a jejich operace. Všechny zkoumané implementace mají společného předka a tím je abstraktní datový typ tabulka. Po představení tabulky, dochází k představení konkrétních realizací hierarchických stromových struktur prioritního vyhledávacího stromu, rozsahového stromu a quad stromu. V této části byly informace o konkrétních strukturách, případně jejich využití a poté byly představeny hlavní operace způsobem slovního popsání, grafického znázornění a popisu v pseudokódu. Poslední část teoretické části se zabývala sekundárním cílem diplomové práce, a to lineárními průchody prostoru. V této části byly rozebrány křivky Peanova, Hilbertova a Z-křivka.

V praktické části diplomové práce byly nejprve obecně představeny hlavní vybrané technologie pro zpracování výsledné webové aplikace. Po představení technologických prostředků byly ukázány klíčové stránky uživatelského prostředí a architektura databáze. V poslední částí byly popsány hlavní třídy a rozhraní pro datové struktury a lineární průchody prostorem za pomoci jazyku UML.

Výsledná webová aplikace je vystavena na cloudové službě PaaS (platforma jako služba) od Heroku, která poskytuje zdarma uložiště a databázi s omezenou velikostí a výkonem. Pro ukázku operací nad datovými strukturami uchovávající multidimenzionální bodová data, jsou připraveny tři datové sady o velikosti osmi, dvanácti a šestnácti prvcích.

POUŽITÁ LITERATURA

[1] SAMET, Hanan. Foundations of multidimensional and metric data structures. Boston: Elsevier/Morgan Kaufmann, c2006. ISBN 978-0123694461.

[2] CORMEN, Thomas H. Introduction to algorithms. 3rd ed. Cambridge: MIT Press, c2009, xix, 1292 s. ISBN 978-0-262-03384-8.

[3] LEWIS, Harry R a Larry DENENBERG. Data structures. 1997. vyd. New York, NY: HarperCollins Publishers, c1991, xv, 509 p. ISBN 06-733-9736-X.

[4] GOODRICH, Michael T. a Roberto TAMASSIA. Algorithm design: foundations, analysis, and Internet examples. New York: Wiley, c2002. ISBN 04-713-8365-1.

[5] McCreight, Edward (May 1985). ""Priority search trees"". SIAM Journal on Scientific Computing. 14 (2): 257-276.

[6] MEHTA, Dinesh P. a Sartaj. SAHNI. Handbook of data structures and applications. Boca Raton, Fla.: Chapman & Hall/CRC, c2005. ISBN isbn1-58488-435-5.

[7] LUEKER, George S. A data structure for orthogonal range queries. In: 19th Annual Symposium on Foundations of Computer Science (sfcs 1978) [online]. IEEE, 1978, 1978, s. 28-34 [cit. 2018-04-27]. DOI: 10.1109/SFCS.1978.1. Dostupné z: http://ieeexplore.ieee.org/document/4567959/

[8] FINKEL, R. A. a J. L. BENTLEY. Quad trees a data structure for retrieval on composite keys. Acta Informatica [online]. 1974, 4(1), 1-9 [cit. 2018-04-27]. DOI: 10.1007/BF00288933. ISSN 0001-5903. Dostupné z: http://link.springer.com/10.1007/BF00288933http://www.cs.umd.edu/~hjs/pubs/acmgis99.pdf.

[9] ORENSTEIN, Jack A. Multidimensional tries used for associative searching. Information Processing Letters [online]. 1982, 14(4), 150-157 [cit. 2018-04-27]. DOI: 10.1016/0020-0190(82)90027-8. ISSN 00200190. Dostupné z: http://linkinghub.elsevier.com/retrieve/pii/0020019082900278

[10] SAMET, Hanan. The Quadtree and Related Hierarchical Data Structures. ACM Computing Surveys [online]. 16(2), 187-260 [cit. 2018-04-27]. DOI: 10.1145/356924.356930. ISSN 03600300. Dostupné z: http://portal.acm.org/citation.cfm?doid=356924.356930

[11] BADER, Michael. Space-Filling Curves [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013 [cit. 2018-04-27]. Texts in Computational Science and Engineering. ISBN 978-3-642-31045-4.

[12] Morton, G. M. (1966), A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd.

[13] HILBERT, David. Über die stetige Abbildung einer Linie auf ein Flächenstück. Mathematische Annalen – 38. 1891 Dostupné online [PDF]. (anglicky)

[14] Information about framework. Spring boot [online]. [cit. 2018-04-27]. Dostupné z: https://projects.spring.io/spring-boot/

[15] FRY, Ben a Casey REAS. Information about library. Processingjs.org: a port of the Processing Visualization Language [online]. [cit. 2018-04-27]. Dostupné z: http://processingjs.org/

[16] SIPSER, Michael. Introduction to the theory of computation. Canada: Thomson, 2005. ISBN isbn0-619-21764-2.

[17] WITTEN a NEAL. Using Peano Curves for Bilevel Display of Continuous-Tone Images. IEEE Computer Graphics and Applications[online]. 1982, 2(3), 47-52 [cit. 2018-04-27]. DOI: 10.1109/MCG.1982.1674228. ISSN 0272-1716. Dostupné z: http://ieeexplore.ieee.org/document/1674228/

[18] Hilbert & Peano Space Filling curves. Github.com: Google Open Source [online]. [cit. 2018-04-27]. Dostupné z: https://github.com/google/hilbert

[19] CORTESI, Aldo. Z-Order curve. Github.com: Open source repository [online]. [cit. 2018-04-27]. Dostupné z: https://github.com/cortesi/scurve

PŘÍLOHY

Příloha A – Peanova křivka..74
Příloha B – Z-křivka..75
Příloha C – Hilbertova křivka 76
Příloha D – CD s vytvořenou aplikací 77
Příloha E – Uživatelská příručka..78

PŘÍLOHA A – PEANOVA KŘIVKA

```
private Point indexToPoint(int t, int n) {
    if ((t < 0) || (t > n * n)) {
        return null;
    }
    Point outPoint = null;
    int x = 0, y = 0;
    for (int i = 1; i < n; i = i * 3) {
        int s = t % 9;

        //rx jsou souradnice v subregionu
        int rx = s / 3;
        int ry = s % 3;

        if (rx == 1) {
            ry = 2 - ry;
        }

        //otacim do doby nez jsou spravne souradnice
        if (i > 1) {
            outPoint = rotate(i, x, y, s);
            x = outPoint.x;
            y = outPoint.y;
        }

        x += rx * i;
        y += ry * i;

        t = t / 9;

    }

    return new Point(x, y);
}

private Point rotate(int n, int x, int y, int s) {
    if (n == 1) {
        // Special case
        return new Point(x, y);
    }

    n = n - 1;
    switch (s) {
        case 0:
            return new Point(x, y); // normal
        case 1:
            return new Point(n - x, y);// flip horizontal
        case 2:
            return new Point(x, y);// normal
        case 3:
            return new Point(x, n - y); // flip vertical
        case 4:
            return new Point(n - x, n - y);// flip vertical and fliph horizontal
        case 5:
            return new Point(x, n - y);// flip vertical
        case 6:
            return new Point(x, y);// normal
        case 7:
            return new Point(n - x, y);// flip horizontal
        case 8:
            return new Point(x, y);// normal
    }

    return null;
}
```

PŘÍLOHA B – Z-KŘIVKA

```
//The Z-order curve is generated by interleaving the bits of an offset.
private long[] point(long index) {
    long[] returnPoint = new long[dimensions];
    Integer iWidth = bits * dimensions;
    for (int i = 0; i < iWidth; i++) {
        long actualPoint = bitRange(index, iWidth, i, i + 1) << ((iWidth - i - 1) /
dimensions);
        returnPoint[i % dimensions] += actualPoint;
    }

    return transpose(returnPoint);
}

private long[] transpose(long[] array) {
    for (int i = 0; i < array.length / 2; i++) {
        long temp = array[i];
        array[i] = array[array.length - i - 1];
        array[array.length - i - 1] = temp;
    }

    return array;
}

private long bitRange(long x, Integer width, Integer start, Integer end) {
    return x >> (width - end) & Math.round(Math.pow(2, (end - start) - 1));
}
```

PŘÍLOHA C – HILBERTOVA KŘIVKA

```
private long[] point(BigInteger index) {
    return transposedIndexToPoint(bits, transpose(index));
}

private long[] transposedIndexToPoint(int bits, long[] x) {
    final long N = 2L << (bits - 1);
    // Note that x is mutated by this method (as a performance improvement
    // to avoid allocation)
    int n = x.length; // number of dimensions
    long p, q, t;
    int i;
    // Gray decode by H ^ (H/2)
    t = x[n - 1] >> 1;
    // Corrected error in Skilling's paper on the following line. The
    // appendix had i >= 0 leading to negative array index.
    for (i = n - 1; i > 0; i--) {
        x[i] ^= x[i - 1];
    }
    x[0] ^= t;
    // Undo excess work
    for (q = 2; q != N; q <<= 1) {
        p = q - 1;
        for (i = n - 1; i >= 0; i--)
            if ((x[i] & q) != 0L)
                x[0] ^= p; // invert
            else {
                t = (x[0] ^ x[i]) & p;
                x[0] ^= t;
                x[i] ^= t;
            }
    } // exchange
    return x;
}

private long[] transpose(BigInteger index) {
    long[] x = new long[dimensions];
    byte[] b = index.toByteArray();
    for (int idx = 0; idx < 8 * b.length; idx++) {
        if ((b[b.length - 1 - idx / 8] & (1L << (idx % 8))) != 0) {
            int dim = (length - idx - 1) % dimensions;
            int shift = (idx / dimensions) % bits;
            x[dim] |= 1L << shift;
        }
    }
    return x;
}
```

PŘÍLOHA D – CD S VYTVOŘENOU APLIKACÍ

Pro korektní spuštění vytvořené výsledné webové aplikace je nutné rozbalit celý obsah souboru DiplomovaPrace.rar do adresáře. V rozbalené složce se nacházejí následující položky:

- složka ***src*** - zdrojové soubory webové aplikace
- soubor ***.gitignore*** – textový soubor, který obsahuje soubory, které nebudou případně verzované
- soubor ***pom.xml*** – konfigurační soubor Mavenu

SPUŠTĚNÍ VÝSLEDNÉ WEBOVÉ APLIKACE

Pro výsledné spuštění výsledné webové aplikace doporučuji použít programovací prostředí IntelliJ IDEA 2017.3.1 x64 s podporou Mavenu, který slouží pro správu a řízení projektu. Pro správnou funkčnost aplikace je potřeba mít lokální databázi platformy MySQL s vytvořenou konkrétní databází „thesis“. Pro změnění cest k lokální databázi, či názvu databáze lze modifikovat konfigurační soubor „application-local.properties“. Stejná pravidla platí pro vzdálenou databázi s konfiguračním souborem „application-remote.properties“. Při prvním spuštění aplikačního serveru je nutno nastavit v konkrétním konfiguračním souboru hodnotu „spring.jpa.hibernate.ddl-auto=create“, která vytvoří strukturu potřebných tabulek. Při následujících spuštění lze konfiguraci změnit na hodnotu update.
Jakmile splníme tyto kroky stačí již pomocí Mavenu vyčistit projekt „Maven clean“ a poté „Maven install“, který zkompiluje potřebné soubory pro správné spuštění aplikace. Jakmile je zkompilováno, lze výslednou webovou aplikaci pustit přes klasický build v prostředí a aplikace je spuštěna v integrovaném aplikačním serveru programového prostředí.

Druhou možností spuštění výsledné webové aplikace je vystavená webová aplikace na internetovém odkazu https://thesismds.herokuapp.com/. Aplikace je vystavena na PaaS platformě Heroku zdarma. Rychlost výsledné aplikace odpovídá slabému aplikačnímu serveru dostupném v Americe.

PŘÍLOHA E – UŽIVATELSKÁ PŘÍRUČKA

Tato příručka popisuje funkčnost výsledné webové aplikace a jejího rozložení. Na následujícím obrázku (Obrázek 46) lze vidět úvodní stránku webové aplikace. Nejprve je na stránce zobrazeno navigační menu, které slouží k přechodu mezi všemi podporovanými strukturami a lineárními průchody prostorem. Výčet dostupných struktur a průchodů:

- prioritní vyhledávací strom (Priority search tree),
- rozsahový strom (Range tree),
- bodový quad strom, (Quad tree – Point based)
- bodově oblastní quad strom (Quad tree – Trie based),
- hilbertova křivka (Hilbert),
- z-křivka (ZOrder),
- peanova křivka (Peano).

V další části jsou rozděleny datové struktury a lineární průchody prostorem do kategorií. V poslední části je zobrazeno maximálně posledních 10 vykonaných operacích nad vybranými datovými strukturami, které lze provést znovu.

Obrázek 46 Uživatelská příručka – úvodní obrazovka

Při kliknutí na buňku ve sloupci tabulky „show details“ dochází k vyvolání vyskakovacího formuláře (Obrázek 47), kde se lze vidět detaily stromu. Obrázek je pouze ilustrací, pro úsporu místa je přesunut „Tree detail“ na pravou část ilustrace. Na obrázku jsou nejprve ukázány podrobnosti o operaci, které následuje zobrazený detail prvku, nad kterým byla operace vykonána

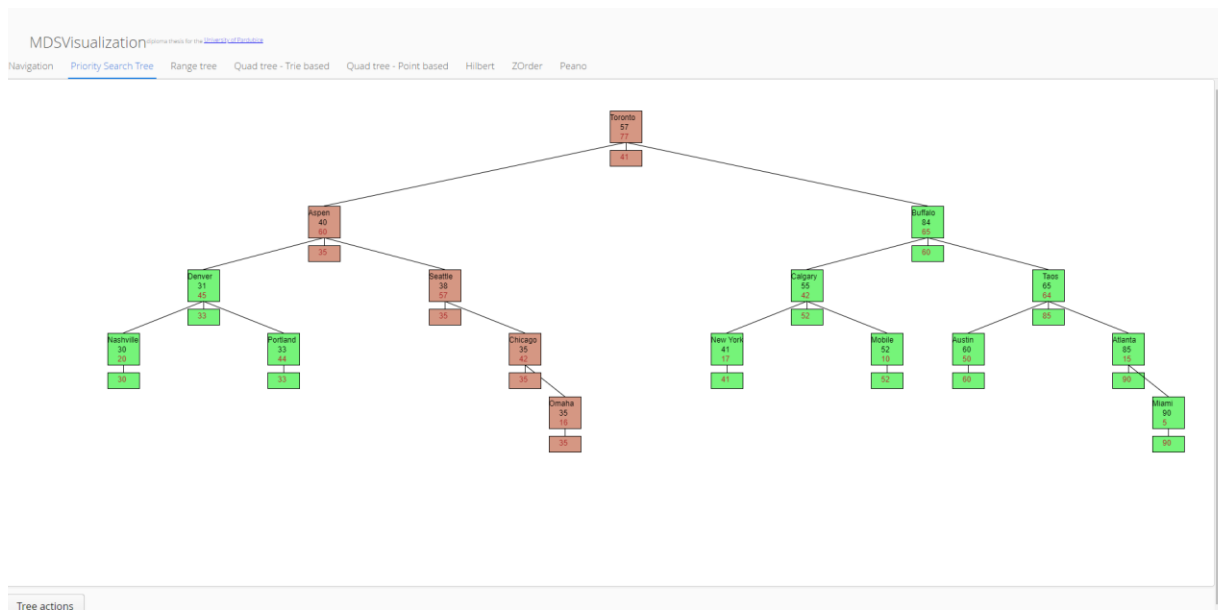
(při operaci *Vybuduj* je za prvek označen kořen stromu). V poslední části je zobrazen detail o konkrétním stromu se seznamem prvků.

Obrázek 47 Uživatelská příručka – detail vykonané operace

Po přepnutí na určitou datovou strukturu (Obrázek 48), lze vidět konkrétní pohled na kreslící plátno a tlačítko „Tree actions“, které slouží k vykonání operací nad konkrétní datovou strukturou. Po vybudování struktury jsou prvky vykresleny zelenou barvou.

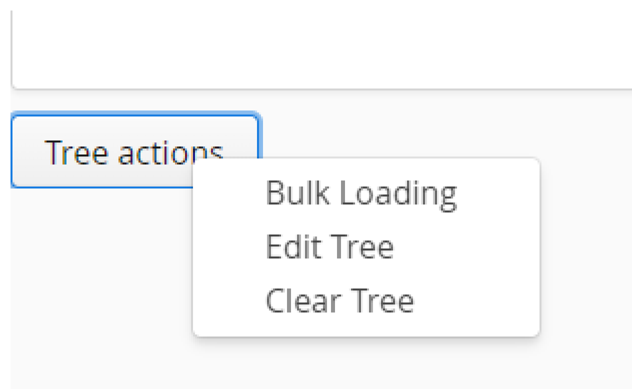
Obrázek 48 Uživatelská příručka – kreslící plátno PST

Pokud je provedena operace, tak prvky, které byly zapojeny při vykonání operace jsou označeny barvou červenou (Obrázek 49). Rychlost operace lze korigovat tlačítky + (zvětšení rychlosti animace) a – (snížení rychlosti animace) na klávesnici. Zastavení a opětovné spuštění animace lze vykonat pomocí stisku levého tlačítka myši na kreslící plátno.



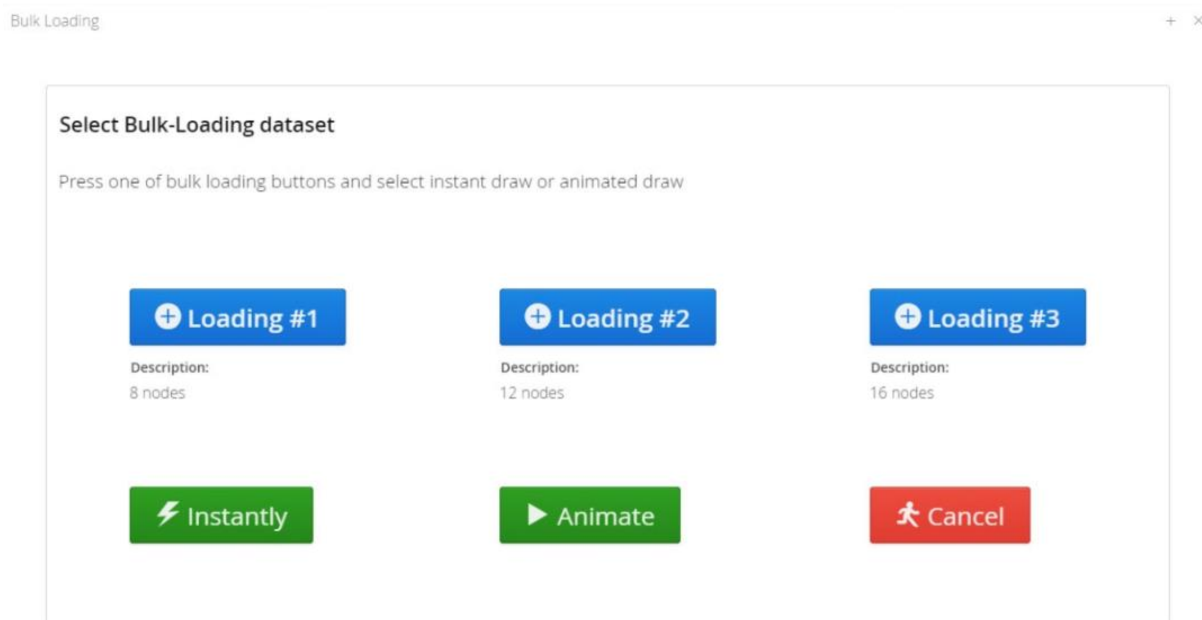
Obrázek 49 Uživatelská příručka – PST s operací najdi

Pří stisku tlačítka „Tree actions“ dochází k vyvolání kontextového menu (Obrázek 50), kde jsou tři volby. První volba slouží k naplnění stromu z předem připravené vstupní sady dat. Druhá k editaci konkrétního stromu a poslední volba nám dovoluje smazat aktuální strom a vyčistit kreslící plátno. První dvě volby budou popsány na následujících stránkách.



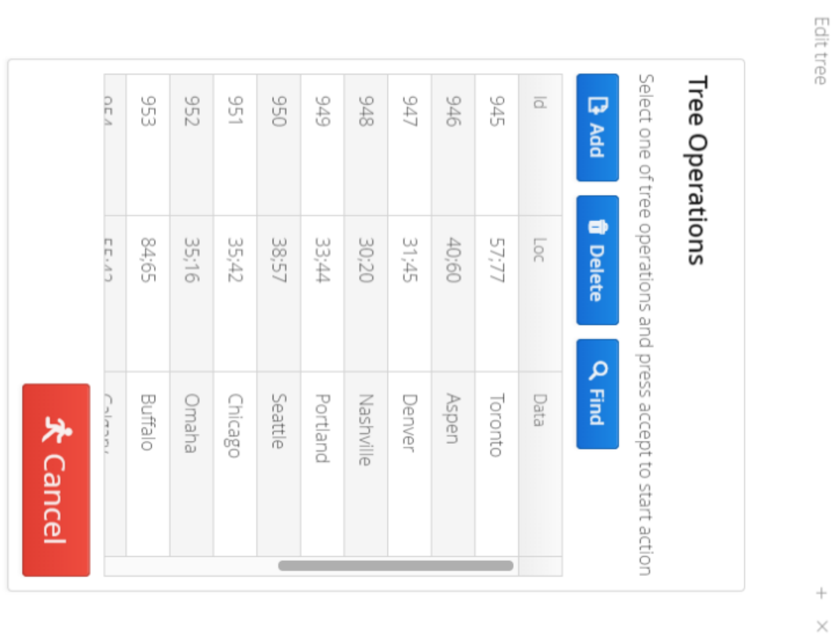
Obrázek 50 Uživatelská příručka – kontextové menu

Po zvolení volby „Bulk loading“ v kontextové nabídce je zobrazen formulář pro vybudování stromu z předem známých vstupnéch dat. Vstupní sady dat jsou tři. Sady jsou rozděleny podle počtu prvků. Nejmenší sada činí 8 prvků, prostřední 12 a největší 16 prvků. Po zvolení vybrané sady lze vybrat mezi okamžitým vybudováním struktury bez animace, nebo vybudováním postupném s animací, které je zpracováno směrem od kořene stromu k listům.



Obrázek 51 Uživatelská příručka – budování struktury z datové sady

Následující volbou „Edit Tree“ vyvoláváme operace nad konkrétní datovou strukturou (Obrázek 52). Tlačítko „Add“ slouží k přidání prvku do struktury (Obrázek 53). Tlačítko „Delete“ pro smazání vybraného prvku z tabulky a tlačítko „Find“ pro vyhledání vybraného prvku v tabulce.



Obrázek 52 Uživatelská příručka – operace nad datovou strukturou

Create node

Location X:

Location Y:

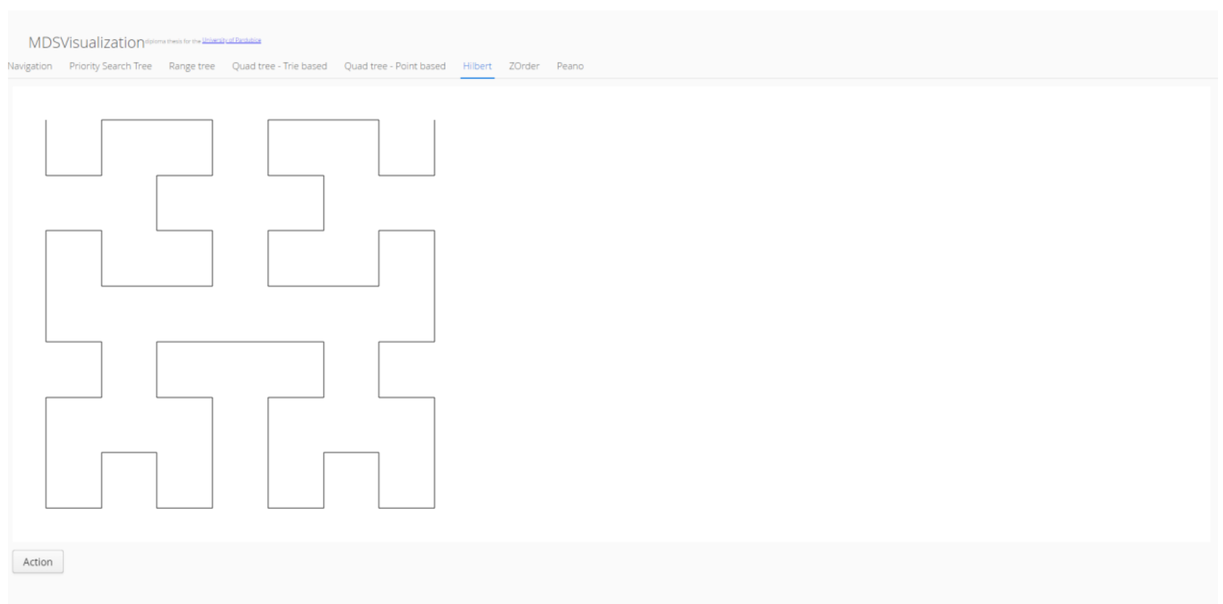
Data:

+ Add node

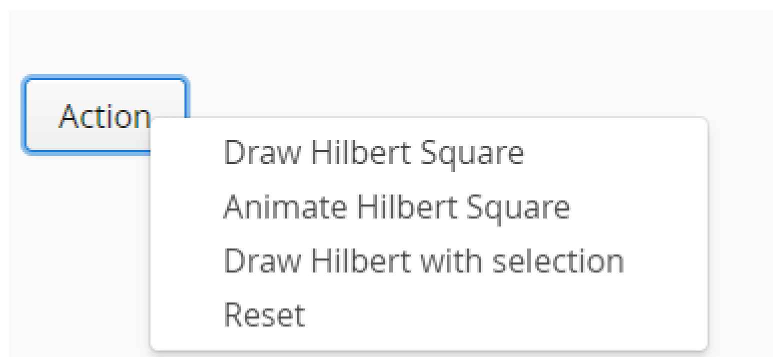
Cancel

Obrázek 53 Uživatelská příručka – vytvoření prvku

Sekundárním cílem diplomové práce bylo zpracování vizualizací lineárních průchodů prostorem. Jak již bylo zmíněno na začátku příručky ve výsledné webové aplikaci byly zpracovány tři průchody. Webová aplikace disponuje třemi možnostmi vykreslení průchodů (Obrázek 54). Vykreslení průchodu okamžitě, animované vykreslení průchody a interaktivní vykreslení s možností zjištění pořadí vykresleného bodu. Na Obrázku 54 je zobrazena třetí iterace vykreslení Hilbertovy křivky. Po opakovaném stisknutí tlačítka vykresli je zvýšena aktuální iterace o jeden a křivka znovu vykreslena.

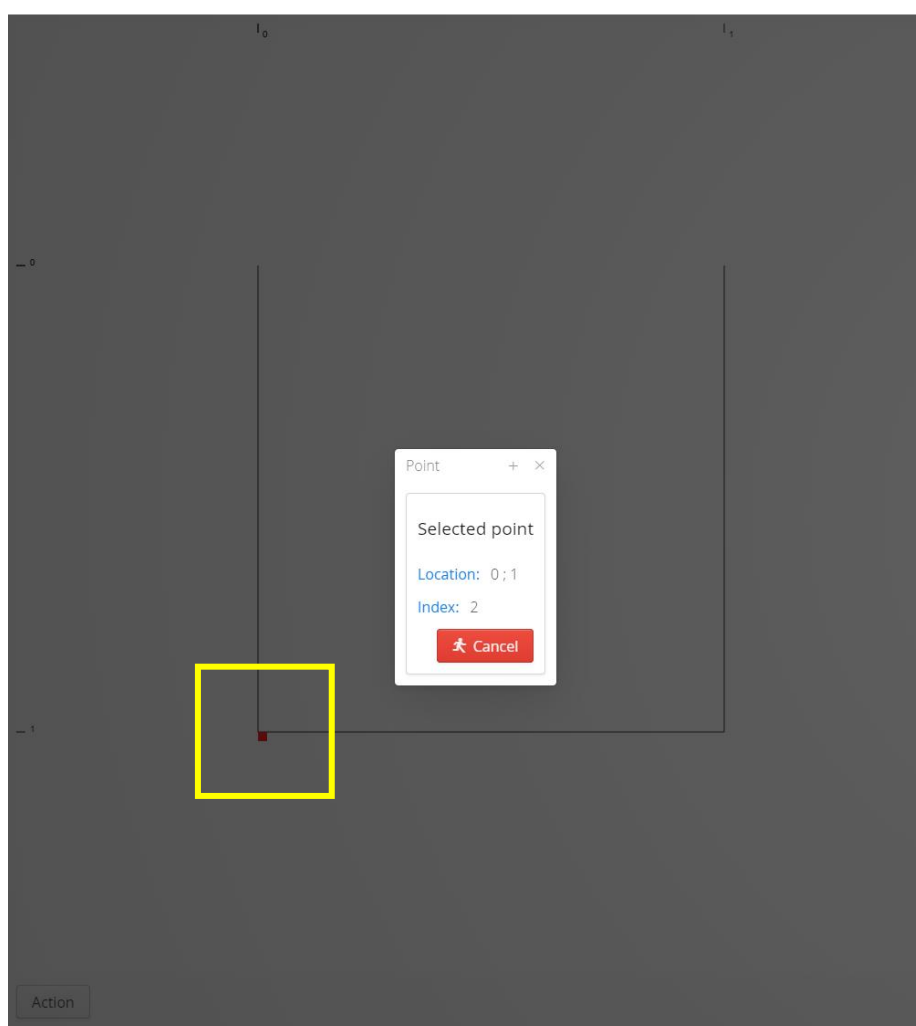


Obrázek 54 Uživatelská příručka – vykreslený průchod



Obrázek 55 Uživatelská příručka – volba možnosti průchodu

Poslední částí je interaktivní možnost vybrání konkrétního bodu na vykresleném průchodu pro zjištění souřadnic a pořadí bodu na křivce (Obrázek 56).



Obrázek 56 Uživatelská příručka – vybraný bod