UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

Softwarová aplikace pro podporu výuky R-stromů

Jiří Hermann

Diplomová práce

2018

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2016/2017

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Jiří Hermann**
Osobní číslo:	**I14251**
Studijní program:	**N2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Softwarová aplikace pro podporu výuky R-stromů**
Zadávající katedra:	**Katedra softwarových technologií**

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

V úvodní části práce je nutné provést přehled teoretických základů vybraných typů R-stromů (R-strom, R* strom, R+ strom, prioritní R-strom, Hilbertův R-strom atd).

Primárním cílem diplomové práce je vytvoření softwarové aplikace pro podporu výuky datových struktur (a příslušných algoritmů) vycházejících z koncepce R-stromu.

Pro tento účel je potřebné vybudovat příslušné vývojové prostředí, v jehož rámci je možné jednak připravit pracovní prostor obsahující 2D objekty a jednak aplikovat požadované algoritmy nad R-stromy, které obsahující data o příslušných 2D objektech z dané pracovní oblasti.

Vyvinutá aplikace poskytne vizualizace evolucí jednotlivých algoritmů pracujících nad R-stromy.

Rozsah grafických prací:

Rozsah pracovní zprávy: **cca 50 stran**

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.**

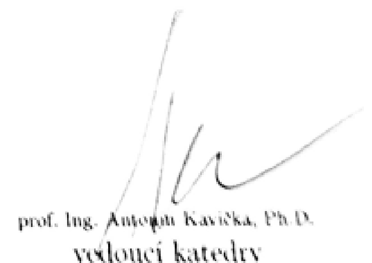
***GOODRICH, Michael T a Roberto TAMASSIA. Algorithm design: foundations, analysis, and Internet examples. 2002. vyd. New York: Wiley, c2002, xii, 708 p. ISBN 04-713-8365-1.**

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

Datum zadání diplomové práce: **31. října 2016**
Termín odevzdání diplomové práce: **17. května 2017**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.



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

V Pardubicích dne 15. listopadu 2016

Prohlášení autora

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

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

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 28.8.2018 Jiří Hermann

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat panu prof. Ing. Antonínu Kavičkovi, PhD. za pomoc při zpracování diplomové práce a za ochotu, vstřícný přístup a cenné rady.

ANOTACE

Cílem diplomové práce je vytvořit demonstrační program pro výuku datových struktur typu R-strom.

KLÍČOVÁ SLOVA

datové struktury, prostor vyplňující křivky, multidimenzionální data, 2D, indexový záznam

TITLE

Software application determined for a learning process related to R-trees

ANNOTATION

The aim of the diploma thesis is to create a demonstration program for teaching R-tree data structures.

KEYWORDS

data structure, space fillings curves, multidimensional data, 2D, index record

OBSAH

Úvod..13

1 Teoretický popis r-stromu..14

1.1 Obecný popis datových struktur typu R-strom 14

1.2 R-strom...15

1.2.1 Metoda Search 17

1.2.2 Metoda Insert 17

1.2.3 Metoda ChooseLeaf..18

1.2.4 Metoda AdjustTree 18

1.2.5 Metoda SplitNode...19

1.2.6 Metoda Kvadratického splitu..19

1.2.7 Metoda PickSeeds...20

1.2.8 Metoda PickNext 20

1.2.9 Metoda LineárníSplit...20

1.2.10 Metoda Delete...20

1.2.11 Metoda FindLeaf..21

1.2.12 Metoda ConsendenseTree...21

1.2.13 R-strom v databázových systémech...22

1.3 R*-strom...24

1.3.1 Optimalizační kritéria 25

1.3.2 Metoda ChooseLeaf..26

1.3.3 Metoda Split..26

1.3.4 ChooseSplitAxis 27

1.3.5 Metoda ChooseSplitIndex..27

1.3.6 Forced Re-instert..27

1.3.7 Metoda InsertData...28

1.3.8 Metoda Insert 28

1.3.9 Metoda OverFlowTreatment....28
1.3.10 Metoda ReInsert....28
1.3.11 R*-strom v databázových systémech....29
1.4 R+-strom....30
1.4.1 Metoda Search....30
1.4.2 Metoda Insert....30
1.4.3 Metoda Delete....32
1.4.4 Splity....32
1.4.5 Metoda Partition....36
1.4.6 Metoda Sweep....37
1.5 Hilbertův strom....37
1.5.1 Metoda Search....38
1.5.2 Metoda Insert....38
1.5.3 Metoda ChooseLeaf....38
1.5.4 Metoda AdjustTree....39
1.5.5 Metoda Delete....39
1.5.6 Metoda HadleOverflow....39
1.6 Pseudo PR-strom....40
1.6.1 Struktura Pseudo PR-stromu....41
1.7 Prostor vyplňující křivky (Space-filling curves)....44
1.7.1 Mortonův rozklad (Z-order curve)....44
1.7.2 Hilbertova křivka....46
2 Vzorové implementace r-stromu....48
2.1 Implementace R-stromu....48
2.1.1 Implementace algoritmu Insert....48
2.1.2 Implementace algoritmu AdjustTree....49
2.1.3 Implementace algoritmu ConsendeseTree....52

2.2 Implementace R*-stromu ... 54
2.2.1 Implementace algoritmu Insert ... 54
2.2.2 Implementace algoritmu overflowTreatment ... 54
2.2.3 Implementace metody reInsert ... 55
2.2.4 Implementace metody split ... 56
2.2.5 Pomocná metoda pro výpočet okrajové plochy ... 57
2.3 Implementace R+-stromu ... 58
2.3.1 Metoda Insert ... 58
2.3.2 Metoda Sweep ... 60
2.3.3 Metoda Partition ... 62
3 Demonstrace R-stromů ... 64
3.1 R-strom ... 64
3.2 R*-strom ... 68
3.3 R+-strom ... 69
4 ZÁVĚR ... 70
5 Použitá literatura ... 71
Příloha A – CD s vytvořenou aplikací ... 73
Příloha B – Uživatelská Dokumentace ... 74

SEZNAM ILUSTRACÍ A TABULEK

Obrázek 1 – Příklad průniku dvou MBR, kde se polygony neprotínají 15

Obrázek 2 – Ukázka R-stromu a MBR 16

Obrázek 3 – Rozložení objektů v R-stromu 16

Obrázek 4 – Demonstrace ukládání objektů do databáze 24

Obrázek 5 – Ukázka R*-stromu 25

Obrázek 6 – Ukázka R*-stromu rozložení uzlů 25

Obrázek 7 – Vkládání uzlu F do uzle 2 31

Obrázek 8 – Vkládání uzle F do uzle 2....31

Obrázek 9 – Finální vložení uzlu F do uzle 2 32

Obrázek 10 – Propagace rozděleni rodičovských uzlů a uzly potomků 33

Obrázek 11 – Vložení záznamu T do uzle 1 34

Obrázek 12 – Přetečení uzle 1 34

Obrázek 13 – Vyvolání metody sweep pro určení osy 35

Obrázek 14 – Partition podle osy y....35

Obrázek 15 – Partition podle osy x....36

Obrázek 16 – Rozdělení uzlu 1 do nových dvou uzlů 36

Obrázek 17 – PR-strom zobrazení struktury 40

Obrázek 18 – Rozdělení záznamů do prioritních uzlů a rekurzivních uzlů 42

Obrázek 19 – Počáteční rozložení záznamů 42

Obrázek 20 – Rozdělení podle extrémních souřadnic 43

Obrázek 21 – Rozdělení zbývajících záznamů podle Ts 44

Obrázek 22 – Rozložení bodů Z-order křivky 45

Obrázek 23 – Rozložení Z-order křivky ve větším měřítku 45

Obrázek 24 – Rozdělení prostoru Z-order křivkou 46

Obrázek 25 – Rozložení bodů pomocí Hilbertovy křivky 47

Obrázek 26 – Vkládání do prvního uzlu 64

Obrázek 27 – Vkládání do prvního uzlu hierarchie záznamů....64

Obrázek 28 – Vkládání rozkladu kořenového uzlu 65

Obrázek 29 – Vkládání rozkladu kořenového uzlu – hierarchie 65

Obrázek 30 – Naplnění 2 uzlů 66

Obrázek 31 – Naplnění 2 uzlů – hierarchie 66

Obrázek 32 – Rozdělení do 3 nových uzlů 67

Obrázek 33 – Rozdělení do 3 nových uzlů – hierarchie 67
Obrázek 34 – Vizualizace R*-stromu 68
Obrázek 35 – Rozložení uzlů R*-stromu..68
Obrázek 36 – Vizualizace R+-stromu..69
Obrázek 37 – Hiearchie uzlů v R+-stromu 69
Obrázek 38 – Design aplikace generovaní 74
Obrázek 39 – Komponenta pro výběr barvy..75
Obrázek 40 – Vyhledávání objektů 76
Obrázek 41 – Mazání objektů..77
Obrázek 42 – Nahrání dokumentu 77

Tabulka 1 – Rozřazení záznamů do uzlů 41
Tabulka 2 – Rozdělení záznamů do 57

SEZNAM ZKRATEK A ZNAČEK

MBR	minimal bound rectangle
2D	dvou dimensionální prostor

ÚVOD

Cílem diplomové práce je realizace datových struktur typu R-strom pro podporu jejich výuky. Tyto datové struktury vznikly z důvodu nutnosti ukládání prostorových dat do databáze. Jedná se především o geografické aplikace, které potřebují ukládat vícerozměrná data. Jelikož datových struktur založených na typu R-strom je velká řada, byly vybrány pro přehled tyto typy:

- R-strom
- R*-strom
- R+-strom
- Hilbertův strom
- PR-strom

Všechny vyjmenované datové struktury pracují v 2D prostoru, kde i vstupní data jsou 2D. Základním prvkem těchto datových struktur je uzel (node), který osahuje MBR (minimal bound rectangle) a ukazatel na potomky. MBR je obdélník, který ohraničuje vkládané datové objekty a jednotlivé uzly. Struktury, jak je patrné z názvu, mají stromovou strukturu a vkládané datové objekty se nacházejí v listových uzlech, kde vložená data jsou pojmenována jako IndexRecord.

Pro implementaci byly vybrány tyto tři typy:

- R-strom
- R*-strom
- R+-strom

Součástí diplomové práce je aplikace pro zobrazení a demonstraci práce s datovými strukturami, kam můžeme vkládat různé geometrické objekty a na základě vkládání objektů se nám vykreslí datová struktura. Jedná se o statická data, která jsou postupně vkládána do struktury, je zde i možnost vygenerování náhodných dat, která se pomocí prostor-plnících křivek seřadí a vloží do struktury.

1 TEORETICKÝ POPIS R-STROMU

1.1 Obecný popis datových struktur typu R-strom

Jak jsem již zmínil v úvodu, datové struktury typu R-strom vznikly na základě možnosti ukládat vícerozměrná data do databáze. Prostorová data zakrývají plochu v mutli-dimensionálním prostoru, ale nelze ji dobře reprezentovat v relační databázi. Máme prostorová data, například geografickou mapu České republiky, a budeme chtít vyhledat všechna města v okolí 50 km. Vymezíme si MBR, který bude mít souřadnice v okruhu 50 km. Klasická jednodimensionální databáze není vhodná pro vyhledávání multidimenzionálních dat. Jednodimensionální datové struktury jsou založeny na vyhledávání podle klíče, kterému je přiřazena hodnota, jako je například datová struktura B-strom, avšak díky implementaci datové struktury typu R-strom můžeme vytvořit dotaz vrať nám všechna města, která jsou v okruhu 50 km. Když předáme v parametru souřadnice ($x_{min}, y_{min}, x_{max}, y_{\max}$), vytvoří se MBR jako ohraničující obdélník v 2D prostoru a vrátíme všechna daná města, která se protínají s tímto ohraničením. Z důvodu čerpání z anglické literatury jsem ponechal anglické názvy metod a i z důvodu následné implementace vybraných datových struktur je ponechání anglických názvu lepší. [3] [6] [14]

Anglický název	Český název
Insert	Vlož
ChooseLeaf	Vyber listový uzel
AdjustTree	Uprav strom
SplitNode	Rozděl uzel
PickSeeds	Vyber dvojici
PickNext	Vyber další
Delete	Smaž
FindLeaf	Najdi list
ConsendenseTree	Kondenzace stromu
ChooseSplitAxis	Vyber osu pro rozdělení
ChooseSplitIndex	Vyber index rozdělení
Re-insert	Znovu vlož
InsertData	Vlož data
OverFlowTreatment	Ošetření přetečení
Partition	Rozdělení

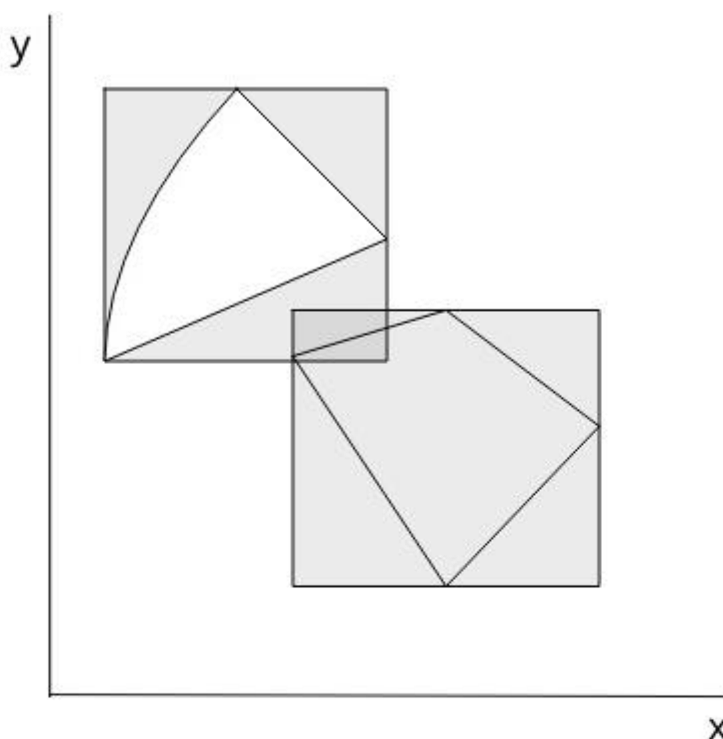
Sweep	Vyber
HandleOwerflow	Uprav přetečení

Table 1 - Názvy metod v angličtině a jejich český význam

1.2 R-strom

Datová struktura R-strom je výškově vyvážený strom odvozený od datové struktury B+ strom. Tato datová struktura se používá pro uchovávání geometrických objektů. Datová struktura R-strom se skládá z uzlů, kde prvním a hlavním uzlem je kořenový uzel, který se dál rozpadá na poduzly nebo jen uzly a poslední částí jsou listové uzly, ve kterých jsou uchovávány záznamy. Každý z uzlů obsahuje MBR (minimální ohraničující obdélník), který ohraničuje všechny potomky daného rodičovského uzlu. Listové záznamy obsahují ukazatele na objekty uložené v databázi. Tyto uzly jsou implementovány jako stránky na disku.

V R-stromu je dovoleno překrývání MBR. MBR může být obsažené i ve více uzlech, ale taky může být přidružený pouze k jednomu uzlu. To znamená, že prostorové prohledávání může navštívit hodně uzlů, než najde správný záznam. Může nastat situace, kdy se nám protnou dvě MBR, ale jejich geometrické objekty se navzájem neprotínají. R-strom se tyto průniky snaží minimalizovat filtrováním. [3] [6] [11] [14]



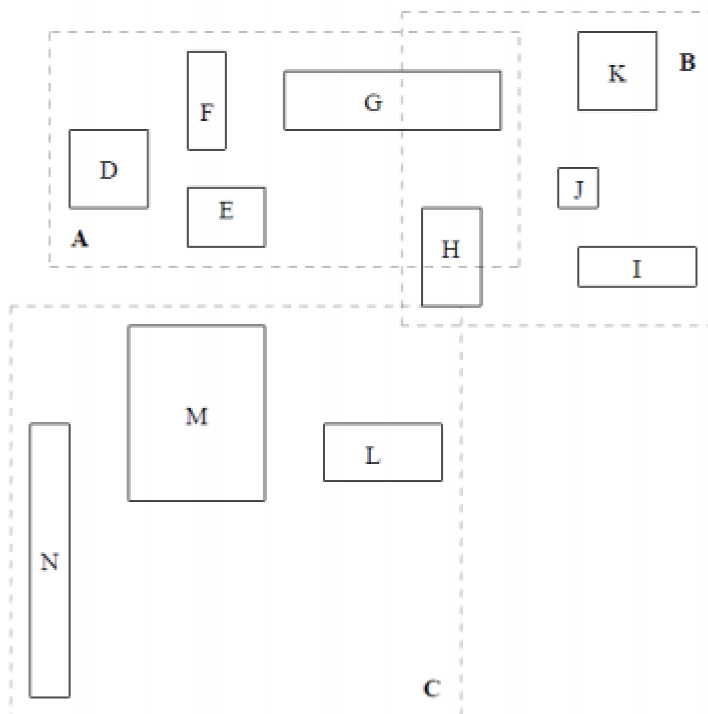
Obrázek 1 – Příklad průniku dvou MBR, kde se polygony neprotínají

Charakteristika R-stromu:

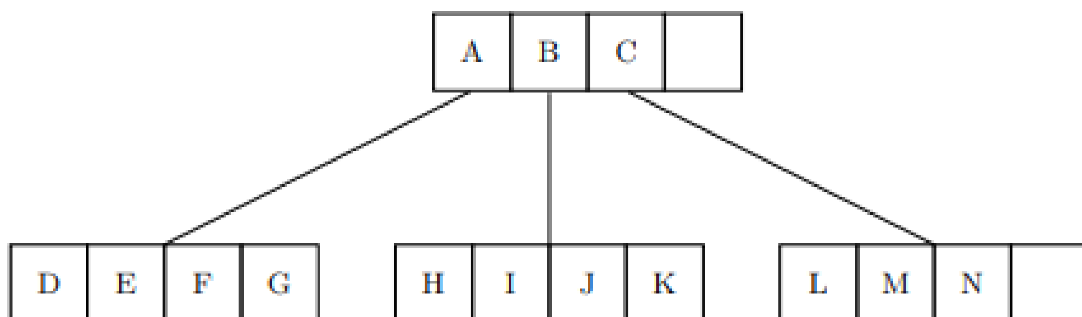
- Každý uzel obsahuje minimální počet m až M indexových záznamů, pokud se nejedná o kořen.

- Každý indexový záznam obsahuje (MBR, id), kde MBR je minimální ohraničující obdélník, který obsahuje n-dimensionální datový objekt, který reprezentuje id.
- Všechny listové uzly jsou na stejné úrovni.
- Každý nelistový uzel, nejedná se zde o kořen, má m až M potomků.
- Každý nelistový uzel také obsahuje (MBR, p), kde MBR je minimální ohraničující obdélník, který ohraničuje všechny potomky, p je pointer na potomka.
- Minimální počet potomků kořene je 2, pokud kořen není list.

Z charakteristiky R-stromu je patrné, že se jedná o vysoce vyvážený strom. R-strom je dynamická datová struktura. Na obrázku 2 můžeme vidět demonstraci MBR a jejich potomků. Kde minimální počet m = 2 a M = 4.



Obrázek 2 – Ukázka R-stromu a MBR [3]



Obrázek 3 – Rozložení objektů v R-stromu [3]

Výška stromu, kterou může R-strom nabývat, je:

$$h_{max} = [log_m N] - 1$$

Toto číslo je odvozeno ze všech uzlů, které obsahují minimální povolený počet potomků.

Maximální počet uzlů je:

$$\sum_{i=1}^{h_{max}} \left[\frac{N}{m^i}\right] = \left[\frac{N}{m}\right] + \left[\frac{N}{m^2}\right] + \cdots + 1$$

Nejhorší prostorové využití je:

$$\frac{m}{M}$$

Uzly budou obsahovat spíše více uzlů; čím více uzlů R-strom obsahuje, tím lépe je pokrytý a využitý prostor. S parametrem m můžeme libovolně manipulovat v rámci dosahování co nejlepších výsledků.

1.2.1 Metoda Search

Vyhledávácí algoritmus R-stromu začíná od kořene a jde až dolů k listovým uzlům. Jedná se o podobné vyhledávání, jako je například u datové struktury typu B-strom. Na rozdíl od B-stromu může dojít k situaci, že v R-stromu budeme potřebovat prohledávat více potomků navštíveného uzlu. To nám nezaručuje dobrý výkon při hledání. Avšak R-strom se stará, aby struktura stromu byla co nejčistší, a snaží se odstraňovat irelevantní časti indexového prostoru.

Vstupem vyhledávacího algoritmu je **R** reprezentace R-stromu s nenulovým kořenem a **S** je vyhledávaná podoblast. Výstupem je seznam všech listových uzlů, které se protnuly s vyhledávanou podoblastí **S**. V algoritmu je označen uzlový záznam písmenem E a pokrývající obdélník I_e.

Algoritmus 1

Krok 1: Jestliže R není listovým uzlem, pak prozkoumej, jestli všechny záznamy T uzlu R pronikají s vyhledávací oblastí S kde platí S $\cap$ I_e, *pro všechny tyto záznamy vyvolej metodu search.*

Krok 2: Pokud R je list, projdi všechny záznamy E a zjisti, jestli S $\cap$ I_e, *pokud ano, E je výsledný záznam.*

1.2.2 Metoda Insert

Vkládání indexových rekordů do stromu je podobné jako vkládání záznamů u B-stromu. Indexové rekordy jsou vkládány do listových uzlů. Může nastat situace, že chceme vložit

indexový rekord do listového uzlu, který již má plnou kapacitu. V tomto případě se musí daný uzel tzv. splitnout. Daný split uzlu se poté musí propagovat stromem směrem vzhůru.

Vstupem vkládacího algoritmu je kořenový uzel R R-stromu a nově vkládaný indexový záznam E.

Algoritmus 2

Krok 1: Vyvolej metodu ChooseLeaf, která vybere uzel L, do kterého budeme vkládat záznam E.

Krok 2: Pokud kapacita L je menší než maximální počet M, vlož záznam E do L, pokud však není, vyvolej metodu split a rozděl uzel L na uzly L a LL.

Krok 3: Vyvolej metodu AdjustTree na uzlu L a také na uzlu LL, pokud došlo k rozlomení.

Krok 4: Pokud split uzlu se dostal až ke kořeni, proveď rozdělní kořene, kde vytvoříme nový kořen R a vložíme do něj nové 2 potomky N a NN.

1.2.3 Metoda ChooseLeaf

Metoda ChooseLeaf nám vybírá takový listový uzel L, do kterého se nejvíce hodí nově vkládaný indexový záznam E.

Algoritmus 3

Krok 1: Pomocný uzel N si nastavíme jako kořenový uzel.

Krok 2: Pokud N je listový uzel, vrať N.

Krok 3: Pokud N není listový uzel, najdi takový záznam F v uzlu N tak, že I_F potřebuje nejmenší rozšíření, tak aby $I_E \subseteq I_F$. Remízu vyřešme výběrem záznamu s nejmenším obdélníkem.

Krok 4: Do N nastav ukazatel na F_p.

1.2.4 Metoda AdjustTree

Metoda AdjustTree nám zabezpečuje vyváženost R-stromu a také upravuje plochu pokrývajících MBR. AdjustTree začíná od uzlu L a probublává až ke kořeni R. Tímto průchodem kontroluje každého z rodičů předchozího uzlu a přepočítává oblast pokrývajících MBR. Pokud došlo k rozlomení daného uzlu, tak se daná oblast přepočítá a nastaví se nová velikost MBR.

Algoritmus 4

Krok 1: Do pomocného uzlu N nastav uzel L, pokud došlo k rozlomení uzlu, nastav do pomocného uzlu NN rozlomený uzel LL.

Krok 2: Pokud N je kořen, zastav algoritmus.

Krok 3: Do pomocného uzlu P nastav rodiče uzlu N, E_n je záznam v uzlu P, který obsahuje uzel N. Upravme I_{E_N} (ohraničující obdélník) tak, aby těsně obklopoval všechny záznamy z uzlu N.

Krok 4: Pokud došlo k rozlomení uzlu a máme pomocný uzel NN, vytvořme pro tento uzel pomocný uzel E_{NN} s $p_{E_{NN}}$ a $I_{E_{NN}}$, který uzavírá všechny záznamy uzlu NN. Pokud v uzlu P je volné místo, vlož tam záznam E_{NN}. Pokud je uzel P plný, vyvoláme metodu SplitNode, rozdělíme uzel P na P a PP, aby tyto uzly obsahovaly uzel E_{NN} a všechny ostatní uzly uzlu P.

Krok 5: Nastav N na P a NN na PP, a pokud nastalo rozlomení, opakuj Krok 2.

Představme si situaci, že nám nastane případ, kdy prostřední uzel obsahuje pouze jeden záznam, kde doporučená minimální velikost m je 2. Díky algoritmu AdjustTree nám nezůstane tento uzel osamocený, ale vloží se do nového uzlu.

1.2.5 Metoda SplitNode

Metoda SplitNode nám slouží k rozdělení přeplněného uzlu N na nové uzly N a NN, které obsahují potomky uzlu N, kde velikost potomků je M + 1. SplitNode by se měl provést tak, že oba nové uzly se vloží do R-stromu a při vyvolání metody Search nedojde k situaci, že bude prohledávat v obou nových uzlech, ale pouze v tom, do kterého zasahuje prohledávaná oblast. Algoritmus Search porovnává uzly s vyhledávanou oblastí, jestli daný uzel spadá do prohledávané oblasti, proto algoritmus SplitNode vytváří nové uzly a přizpůsobuje MBR nově vytvořeným uzlům. Podobně funguje Algoritmus ChooseLeaf, který vybírá takový uzel, do kterého se vejde nový indexový záznam. U obou algoritmů se prochází od vrchu, tedy od kořene stromu až k listovým uzlům a vždy se porovnává vstupní záznam s daným uzlem na dané úrovni, jestli je záznam uvnitř daného uzlu.

Máme zde dva algoritmy, které nám popisují, jak rozložit množinu záznamů M + 1 do nových uzlů o minimální velikosti m.

1.2.6 Metoda Kvadratického splitu

Kvadratický split se snaží rozdělit danou oblast na co nejmenší dvě nové podoblasti, avšak není zaručeno, že dané podoblasti budou opravdu nejmenší. Algoritmus vybírá dva záznamy (uzle), které by nejvíce plýtvaly prostorem.

Výpočet pokrytého místa se provede následovně: celkový obsah obou dvou uzlů minus obsah oblastí samostatných záznamů. Ostatní záznamy čekají na přidělení do těchto dvou skupin. V každém dalším kroku je porovnávaná velikost oblasti, o kterou se zvětší přidáním záznamu. Vybíráme takovou oblast, kde se její obsah nejvíce zvětší.

Algoritmus 5

Krok 1: Vyvoláním metody PickSeeds vyber první dva záznamy, které se stanou prvními prvky rozkladu, do nových dvou uzlů. Prvky jsou vloženy do nových uzlů.

Krok 2: Pokud jsou všechny záznamy přiřazeny skonči, pokud jeden z uzlů má tak málo záznamů, že není splněna podmínka i minimální velikosti m, musí se všechny ostatní záznamy vložit do aktuálního uzlu, aby byla podmínka splněna, poté skonči.

Krok 3: Vyvolání metod PickNext, který bude přiřazen do uzlu. Přiřaď ho do takového uzlu, kde bude s menším pokrývajícím MBR. Když nastanou remízy, vyřeš vybrání uzle s nejmenší plochou, kterou pokrývá MBR, a poté s nejmenším počtem záznamů obsažených v uzlu.

1.2.7 Metoda PickSeeds

Algoritmus vybírá první dva záznamy, které se stanou prvními prvky nově vytvořených uzlů.

Algoritmus 6

Krok 1: Pro každý pár záznamů E_1 *a* E_2 *vypočítejte celkovou plochu d, kterou pokrývají záznamy* E_1 *a* E_2.

$$d = S(J) - S\left(I_{E_1}\right) - S(I_{E_2})$$

Krok 2: Vyberme pár s největším d (největší pokrytou plochou).

1.2.8 Metoda PickNext

Metoda PickNext třídí zbylé záznamy do nově vytvořených uzlů.

Algoritmus 7

Krok 1: Pro každý záznam E vypočítáme plochu d_1 *a* d_2 *uzlu* N_1 *a* N_2 *a zjistíme, o jakou část se zvětšila.*

Krok 2: Vybereme záznam s maximálním rozdílem d_1 *a* d_2.

1.2.9 Metoda LineárníSplit

Metoda lineárního splitu je velice podobná jako KvadratickýSplit, je lineární k max. počtu prvků M a k počtu dimenzí. Tento algoritmus se liší pouze algoritmem PickSeeds. PickNext jednoduše vybírá jakýkoliv ze zbývajících prvků, který se pak vkládá do uzlu.

Algoritmus 8

Krok 1: Najdi takové MBR s nejpravějším levým okrajem a s nejlevějším pravým okrajem. Tento postup proveď pro všechny dimenze, ulož si tyto páry a jejich separaci.

Krok 2: Normalizuj separaci vydělením odpovídajícím rozměrem obdélník pokrývající všechny záznamy.

Krok 3: Vyber pár s největší normalizovanou separací podle kterékoli dimenze.

1.2.10 Metoda Delete

Metoda Delete vyhledává vstupní záznam, který se má smazat, a pokud ho najde, tak tento indexový záznam E smaže z R-stromu. Dochází zde k deformaci stromu, a proto je potřeba zajistit, aby strom byl vyvážený. To zabezpečíme pomocí algoritmu ConsendenseTree.

Algoritmus 9

Krok 1: Vyvolej metodu FindLeaf, prohledej celý strom a najdi takový listový uzel N, který obsahuje indexový záznam E. Pokud daný listový uzel neexistuje, skonči.

Krok 2: Odstraň záznam E z listového uzlu N.

Krok 3: Vyvolej metodu ConsendenseTree a předej jí uzel N.

Krok 4: Pokud kořen R-stromu má pouze jednoho potomka po předešlém kroku, vytvoř z tohoto potomka nový kořen.

1.2.11 Metoda FindLeaf

Metoda FindLeaf prohledává R-strom od kořene po listový uzel a hledá takový uzel N, ve kterém nachází indexový záznam E. Když takový uzel najde, vrací ho.

Algoritmus 10

Krok 1: Pokud uzel N není list, potom najdi záznamy T v N tak, že $I_E \sqsubseteq I_T$. Pro každý takový záznam vyvolej rekurzivně metodu FindLeaf a předej nalezeného potomka této metodě.

Krok 2: Pokud N je list, vrať ho, prohledej jeho potomky (indexové záznamy) a vrať shodný záznam.

1.2.12 Metoda ConsendenseTree

Metoda ConsendeseTree je založena na principu uchovávání vyváženosti R-stromu. Tento algoritmus odstraňuje deformace stromu při mazání indexových záznamů. Vkládáme do něj uzel L, ve kterém došlo k odstranění indexového záznamu. V případě, že tento uzel nesplňuje podmínku, že počet záznamů je větší než m, odstraň tento uzel a přesuň jeho záznamy. Tuto změnu šiř nahoru až ke kořeni, pokud je potřeba. Při probublávání až ke kořeni upravuj všechny MBR, aby ohraničovaly pouze obsahující uzly či záznamy.

Algoritmus 11

Krok 1: Nastav do pomocného uzle N vstupní uzel L, vytvoř prázdnou množinu Q, která bude obsahovat eliminované uzle.

Krok 2: Pokud N je kořen, pokračuj krokem 4. Pokud N není kořen, vytvoř pomocný kořen P, do kterého vlož předka uzlu N a vytvoř pomocný uzel E_N, který obsahuje uzel N.

Krok 3: Pokud uzel N obsahuje méně než m potomků, přidej uzel N do množiny Q a odstraň pomocný uzel E_N z P.

Krok 4: Pokud uzel N byl eliminován, uprav I_{E_N} tak, aby těsně ohraničoval všechny záznamy uzlu N.

Krok 5: Nastav do uzlu N rodiče P a pokračuj krokem 4.

Krok 6: Znovu vlož všechny záznamy z množiny Q do stromu R. Záznamy, které jsou typu uzel, musí být vloženy tak, že listy jejich podstromů budou na stejné úrovni jako listy hlavního stromu. Záznamy typu indexového záznamu jsou vkládány klasicky podle alg. Insert.

1.2.13 R-strom v databázových systémech

Databázové systémy nám nabízí možnost ukládání prostorových objektů do databáze. Tuto funkčnost nám například nabízí databáze PostgreSQL, která má rozšíření PostGIS, toto rozšíření umožňuje vkládat geografická data do databáze. Také databáze Oracle umožňuje vkládat geografická data. Tento demonstrační příklad jsem převzal z Oracle dokumentace. [13]

Ukázka vkládání prostorových objektů do databáze Oracle:

Vytvoření tabulky cola_market :

```
CREATE TABLE cola_markets (
  mkt_id NUMBER PRIMARY KEY,
  name VARCHAR2(32),
  shape SDO_GEOMETRY);
```

Do tabulky vložíme data:

```
INSERT INTO cola_markets VALUES(
  1,
  'cola_a',
  SDO_GEOMETRY(
    2003,  -- two-dimensional polygon
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,1003,3), -- one rectangle (1003 = exterior)
    SDO_ORDINATE_ARRAY(1,1, 5,7) -- only 2 points needed to
          -- define rectangle (lower left and upper right) with
          -- Cartesian-coordinate data
  )
);
```

První vkládaný objekt je obdélník, který popisuje první místo A prodeje coly:

```
INSERT INTO cola_markets VALUES(
  2,
  'cola_b',
  SDO_GEOMETRY(
    2003,  -- two-dimensional polygon
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon (exterior polygon ring)
    SDO_ORDINATE_ARRAY(5,1, 8,1, 8,6, 5,7, 5,1)
  )
);
```

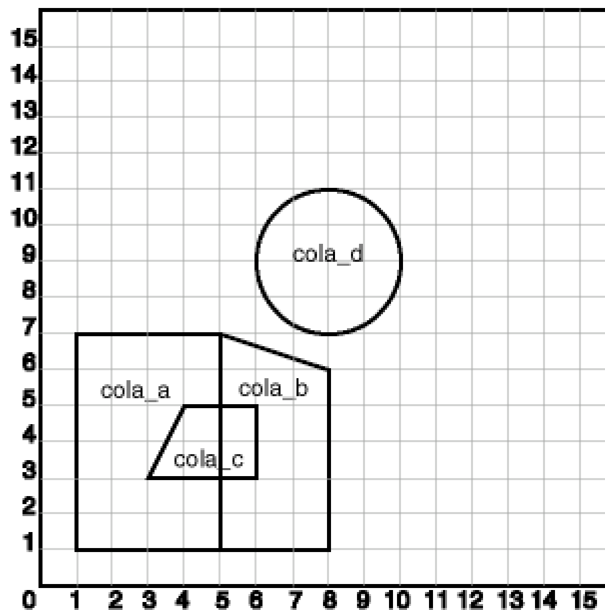
Druhým místem je místo B ve tvaru polygonu:

```
INSERT INTO cola_markets VALUES(
  3,
  'cola_c',
  SDO_GEOMETRY(
    2003,  -- two-dimensional polygon
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,1003,1), -- one polygon (exterior polygon ring)
    SDO_ORDINATE_ARRAY(3,3, 6,3, 6,5, 4,5, 3,3)
  )
);
```

Třetí místo je místo C ve tvaru polygonu:

```
INSERT INTO cola_markets VALUES(
  4,
  'cola_d',
  SDO_GEOMETRY(
    2003,  -- two-dimensional polygon
    NULL,
    NULL,
    SDO_ELEM_INFO_ARRAY(1,1003,4), -- one circle
    SDO_ORDINATE_ARRAY(8,7, 10,9, 8,11)
  )
);
```

Posledním místem je místo D, které má tvar kruhu.



Obrázek 4 – Demonstrace ukládání objektů do databáze [13]

Jak je vidět na obrázku, všechny 4 objekty se uložily do databáze jako grafické primitivy.

Zde příklad zobrazení průniku mezi dvěma geometrickými objekty:

```
SELECT SDO_GEOM.SDO_INTERSECTION(c_a.shape, c_c.shape, 0.005)
FROM cola_markets c_a, cola_markets c_c
WHERE c_a.name = 'cola_a' AND c_c.name = 'cola_c';
```

Průnik mezi objektem cola_a a cola b.

1.3 R*-strom

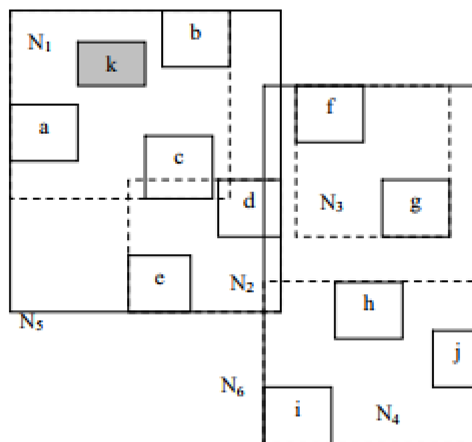
R-strom je datová struktura založená na optimalizačních heuristikách plochy, kterou pokrývají MBR z každého uzle. Datová struktur R*-strom se snaží kombinovat víc těchto heuristik, aby minimalizovala co nejvíce prostoru, který není pokryt. R*-strom používá snižování area, margin a overlap, tyto heuristiky budou popsány později. R*-strom má trochu odlišené dotazy a operace a celkové propočítávání zapouzdřování uzlů ohraničujících obdélníky MBR. Dále tato struktura je lepší v efektivnějším přístupu pro práci s bodovými a prostorovými daty a jeho implementace je jen o maličko náročnější než implementace R-stromu.

R*-strom je varianta R-stromu, který je náročnější na sestavení než R strom, kvůli znovuvkládání dat, ale jeho výsledkem jsou lepší dotazovací výsledky. R*-strom je založený na principu vyváženého B-stromu. Jako u R-stromu i R*-strom obsahuje hlavní uzel, tzv. kořenový uzel, od kterého se datová struktura dále rozvětvuje. Kořen obsahuje ukazatele p, který ukazuje na potomka, a MBR, který zapouzdřuje všechny potomky. Obdobně je tomu i u prostředních uzlů, které obsahují rovněž pointer p a MBR, ovšem listové uzly obsahují ukazatel p, který ukazuje na indexové záznamy, ve kterých jsou uložena prostorová data, a také obsahuje MBR, který pokrývá celkovou minimální oblast, ve které se nachází všechny záznamy. [2] [5] [11] [14]

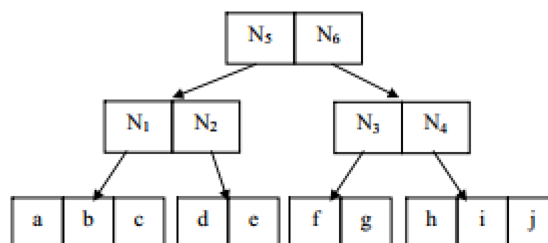
Charakteristika R*-stromu:

- Kořen má minimální 2 uzly, pokud se nejedná o kořen a zároveň listový uzel
- Pro každý střední uzel, který se nachází mezi kořenem a listovým uzlem, platí, že počet potomků je mezi m a M.
- Pro každý listový uzel platí, že počet indexových záznamů je mezi m a M.
- Všechny indexové záznamy jsou na stejné úrovni stromu.
- Strom je vysoce vyvážený.

Problém u R-stromu je v jeho dynamickém sestavování MBR, kde MBR zapouzdřují vždy danou úroveň uzlů. Jelikož každý index záznam, který vkládáme do stromů, může mít různou velikost a také různý tvar, může docházet k plýtvání místem, které není pokryto. Naším cílem je tento prostor co nejvíce minimalizovat a také celkově optimalizovat strom.



Obrázek 5 - Ukázka R*-stromu [12]



Obrázek 6 - Ukázka R*-stromu rozložení uzlů [12]

1.3.1 Optimalizační kritéria

Area

Cílem optimalizačního kritéria Area je co nejvíce snížit celkovou pokrytou oblast I uzlu N, který zapouzdřuje jeho potomky. Tak zvaný **dead space** (mrtvé místo) je tímto kritériem

minimalizován. Dále se zvyšuje výkon průchodu stromu, při kterém se porovnávají MBR a jejich plocha.

Overlap

Optimalizační kritérium Overlap je postaveno na výběru co nejmenšího překrývání dvou MBR. Snižuje to počet cest, které musíme projít, když hledáme indexový záznam.

Margin

Margin reprezentuje celkovou sumu délky hran ohraničujícího obdélníku MBR. Optimalizační kritérium Margin se snaží o co největší snížení okrajů MBR uzlu N.

1.3.2 Metoda ChooseLeaf

Pomocí této metody vybíráme uzel, do kterého se má vkládat nový indexový záznam. U předchozí datové struktury jsme při vyhledávání optimálního uzlu pro vkládání indexového záznamů dávali váhu pouze na porovnávání pokryté oblasti. U této metody pro R*-strom však testujeme nejenom Area, ale také Overlap a Margin. Nechť E_1 ... E_p jsou všechny záznamy aktuálního uzlu N, kde:

$$overlap = \sum_{i=1, i \neq k}^{p} area(E_k \cap E_i), 1 \leq k \leq p$$

Algoritmus 12

Krok 1: Nastav do pomocného uzle N kořen K.

Krok 2: Pokud uzel N je listový uzel, vrať N, pokud není, pak:

Pokud ukazatele p uzlu N ukazují na listové uzly, vyber ten listový uzel L, který potřebuje nejmenší zvýšení překryvu pro MBR při vložení nového indexového záznamu. Remízy vyřeš výběrem MBR, které potřebuje nejmenší zvětšení obsahu, a poté záznam s MBR nejmenšího obsahu.

Pokud ukazatele ukazují na prostřední uzly, pak vyber takové MBR uzlu N, které potřebuje nejmenší zvětšení obsahu, aby zahrnul nová data. Remízy vyřeš výběrem záznamu s MBR nejmenšího obsahu.

Krok 3: Nastav do uzlu N potomka vybraného z Kroku 2.

1.3.3 Metoda Split

Metoda split u R*-stromu je založena na nalezení dobrého rozdělení (splitu). Abychom nalezli dobrý split, vezmeme si všechny záznamy z uzlu N, který chceme rozdělit, a seřadíme je podle os od nejmenší souřadnice po nejvyšší. Jelikož pracujeme ve 2D prostoru, seřazené záznamy se budou řadit podle osy x a y. Pro každou skupinu je určeno M – 2m + 2 rozdělení M + 1 záznamů, které rozdělují do dvou skupin. Pro k-té rozdělení (k = 1, …, (M – 2m + 2)) je popsáno: první skupina obsahuje (m – 1 + k) záznamů a druhá obsahuje zbytek. Pro každé

rozdělení vypočítáváme číslo vhodnosti, které nám určuje dobré rozdělení uzlu. Využíváme tři metody:

1. Hodnota Area area [bb (první skupina)] + area [bb (druhá skupina)]
2. Hodnota Margin margin [bb (první skupina)] + margin [bb (druhá skupina)]
3. Hodnota Overlap area [bb (první skupina)] ∩ area [bb (druhá skupina)]

bb znamená ohraničující box množiny obdélníků.

Vhodné způsoby pro zpracování:

- minimum podle jedné osy
- minimální suma vhodnosti skrze jednu osu
- celkové minimum

Algoritmus 13

Krok 1: Vyvolání metody ChooseSplitAxis vybírá, na které ose dojde ke splitu.

Krok 2: Vyvolání metody ChooseSplitIndex rozřadí záznamy do skupin podle vybrané osy.

Krok 3: Rozděluje záznamy podle vybraných 2 skupin.

1.3.4 ChooseSplitAxis

Metoda ChooseSplitAxis seřazuje M + 1 záznamů uzlu N podle souřadnic a vrací nám osu, na které má být prováděn řez.

Algoritmus 14

Krok 1: Pro každou osu setřiď záznamy podle dolní a poté horní hodnoty jejich MBR a urči všechna rozdělení. Vypočítej S jako součet všech hodnot obvodu pro všechna rozdělení.

Krok 2: Vyber osu s nejmenší hodnotou S jako osu splitu.

1.3.5 Metoda ChooseSplitIndex

Metoda ChooseSplitIndex vybírá takové dvě skupiny, které mají nejmenší překrytí (overlap), a při shodě vybírá ty s nejmenší pokrytou plochou.

Algoritmus 15

Krok 1: Podle vybraného osy splitu vyber takové dvě skupiny, které mají nejmenší překrytí, remízy vyřeš výběrem dvou skupin s nejmenší pokrytou plochou.

1.3.6 Forced Re-instert

Obě datové struktury R-strom a R*-strom jsou nedeterministické v umisťování záznamu do struktury, což znamená, že při vložení dvakrát stejných dat do struktury vznikne dvakrát různý strom. R-strom hodně trpí kvůli vkládaným záznamům a jejich MBR, které byly vkládány na začátku. Ohraničující obdélníky těchto uzlů mohou snižovat efektivitu struktury v pozdějším

vkládáním nových záznamů a uzlů. K dosažení dynamické reorganizace R*-stromu nám pomůže přinutit záznam znovu vložit během Insert algoritmu.

1.3.7 Metoda InsertData

Slouží pouze ke vkládání dat a na dané úrovni.

Algoritmus 16

Krok 1: Vyvolej metodu Insert.

1.3.8 Metoda Insert

Vkládání nového záznamu do listového uzlu.

Algoritmus 17

Krok 1: Vyvolej metodu ChooseSubtree a vyber takový listový uzel N, do kterého může být vložen záznam E.

Krok 2: Pokud N má méně záznamů, než je počet záznamů M, vlož E do N. Pokud N má více záznamů, než je počet M, vyvolej metodu OverFlowTreatment.

Krok 3: Pokud byla vyvolána metoda OverFlowTreatment a nastal Split, šiř metodu VerFlowTreatment nahoru, pokud je to nutné. Pokud metoda OverFlowTreatment způsobila rozdělení kořene, vytvoř nový kořen.

Krok 4: Uprav všechny ohraničující obdélníky na cestě při vkládání tak, aby potomci byli zapouzdřeni a plocha pokrytí byla co nejmenší.

1.3.9 Metoda OverFlowTreatment

Metoda OverFlowTreatment nám provádí znovuvložení uzlu nebo rozdělení uzlu.

Algoritmus 18

Krok 1: Pokud nejsme na úrovni kořene a toto je první volání metody OverFlowTreatment, vyvolej znovu vložení záznamu. Pokud to není první volání metody OverFlowTreatment, vyvolej metodu Split.

1.3.10 Metoda ReInsert

Metoda ReInsert vybírá počet záznamů p, které mají být znovu vloženy do stromu. Dle experimentů s datovou strukturou se ukázalo, že optimální číslo p = 30 % z celkového počtu záznamů v uzlu.

Algoritmus 19

Krok 1: Pro všechny záznamy M + 1 uzlu N vypočítej vzdálenost mezi centroidy záznamů E a uzlu N.

Krok 2: Seřaď všechny záznamy E v sestupném řazení vůči výpočtu z kroku 1.

Krok 3: Odstraň prvních p záznamů z uzlu N a uprav MBR uzlu N.

Krok 4: Podle seřazení z kroku 2 začni s maximální vzdáleností nebo minimální vzdáleností a vyvolej metodu Insert ke znovuvložení odstraněných záznamů.

1.3.11 R*-strom v databázových systémech

Databázový systém SQLite má modul obsahující možnost vkládání prostorových dat. Tento modul se nazývá R*Tree Module a umožnuje nám vytvořit R*-strom v databázi. Tento modul je implementován jako virtuální tabulka. Každý index R*-stromu je virtuální tabulka s lichým počtem sloupců mezi 3 až 11, kde první sloupec je primární klíč s 64bitovým podepsaným celočíselným klíčem. Ostatní sloupce jsou dvojice, jeden pár je rozměr, který obsahuje minimální a maximální hodnoty pro danou dimenzi. Například jednorozměrný strom má 3 sloupce, dvourozměrný strom má sloupců pět atd. SQLite podporuje maximálně 5rozměrný strom. Tento příklad jsem čerpal z SQLite dokumentace. [16]

Vytvoření R*-indexu:

```
CREATE VIRTUAL TABLE <name> USING rtree(<column-names>);
```

Příklad vytvoření dvoudimensionálního R*-tree indexu:

```
CREATE VIRTUAL TABLE demo_index USING rtree(
   id,              -- Integer primary key
   minX, maxX,      -- Minimum and maximum X coordinate
   minY, maxY       -- Minimum and maximum Y coordinate
);
```

Vložení záznamů do tabulky demo_index:

```
INSERT INTO demo_index VALUES(
    1,                   -- Primary key -- SQLite.org headquarters
    -80.7749, -80.7747,  -- Longitude range
    35.3776, 35.3778     -- Latitude range
);
INSERT INTO demo_index VALUES(
    2,                   -- NC 12th Congressional District in 2010
    -81.0, -79.6,
    35.0, 36.2
);
```

Dotazy na R*-tree index:

```
SELECT * FROM demo_index WHERE id=1;
SELECT id FROM demo_index
 WHERE maxX>=-81.08 AND minX<=-80.58
   AND maxY>=35.00  AND minY<=35.44;
```

1.4 R+-strom

Datová struktura R+-strom je odvozena od datové struktury K-D-B-strom. Rozdíl mezi R-stromem a R+-stromem je takový, že R+-strom nepřekrývá MBR na rozdíl od R-stromu, kde se mohou MBR jakkoli překrývat. U této datové struktury může nastat, že vložený záznam bude vložen více než do jednoho uzlu, záznam tak bude rozdělen do dvou uzlů, ale MBR těchto uzlů se nebude překrývat. Dále R+-strom na rozdíl od R-stromu má lepší vyhledávací výkon, aby však byl dosažen lepší vyhledávací výkon, je potřeba minimalizovat pokrytí uzlů ve stromu a překryv. [10] [11] [12] [15]

Struktura R+-stromu:

- Pro kořen platí, že má alespoň 2 potomky, pokud není list.
- Pro listové uzly platí, že minimální počet indexových záznamů je m a maximální počet záznamů je M, listový uzel má ohraničující obdélník MBR zapouzdřující všechny záznamy a ukazatel p ukazující na indexové záznamy, a také platí, že pro uzel N_1 a potomka N_p může mezi ohraničujícími obdélníky MBR_1 a MBR_2 dojít k překrytí.
- Pro nelistové uzly platí to, že minimální počet potomků v uzlu je m a maximální počet M. Každý z nelistových uzlů obsahuje pointer p, který ukazuje na potomky, a ohraničující obdélník MBR, který zapouzdřuje všechny potomky. Pro každé dva uzly uzel N_1 a N_2 platí, že překryv mezi ohraničujícími obdélníky MBR_1 a MBR_2 je nulový.
- Všechny listové uzly jsou na stejné úrovni.

Hlavní snahou R+-stromu je minimalizovat pokrytí oblastí tím, že se minimalizuje pokrytí a není zde překryv mezi uzly. To nám umožňuje pokrývat co nejmenší oblast.

1.4.1 Metoda Search

Metoda Search se značně podobá metodě Search z R-stromu. Hlavní ideou je nejdříve prohledávaný prostor rozdělit na disjunktní podoblasti a pro každý z nich porovnat vyhledávanou oblast s disjunktní oblostí, ta oblast, která se překrývá vybere a sestupuje se od aktuálního uzlu směrem dolů, kde jsou uložené indexové záznamy. Na rozdíl od předešlých dvou algoritmů R+-strom nedovoluje překrývání uzlu a jejich MBR, což umožnuje lepší prohledávání.

Algoritmus 20

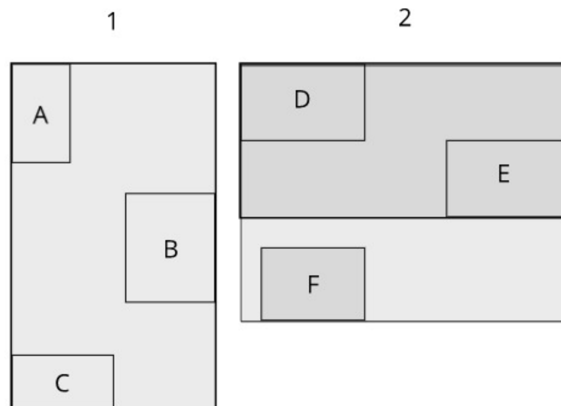
Krok 1: Pokud N není listový uzel, pak pro každého potomka uzlu N otestuj, jestli MBR potomka P nepřekrývá MBR hledané oblasti O. Pokud ano, vyvolej metodu Search (P, O).

Krok 2: Pokud N je listový uzel, otestuj všechny záznamy uzlu N a vrať všechny záznamy, které se překrývají s oblastí O.

1.4.2 Metoda Insert

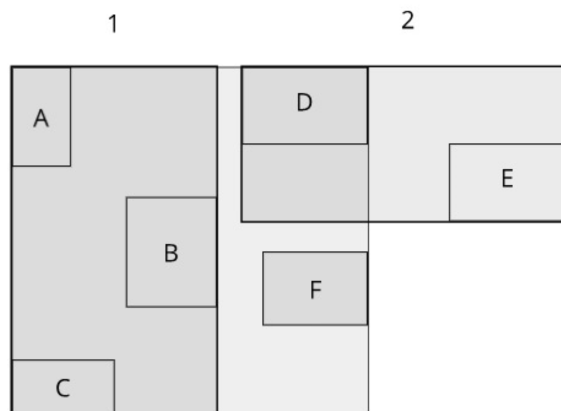
Jako u obou předešlých R-stromů, tak i u této datové struktury začíná vkládání objektu u nalezení správného listového uzlu, do kterého se indexový záznam vloží. Na rozdíl od R-

stromu není u R+-stromu dovoleno, aby se MBR u uzlů překrývaly, proto když by mělo dojít k překrytí MBR, musí se daný uzel rozdělit. To znamená, že záznam může patřit do 2 a více uzlů. Při rozdělení uzlu nebo záznamu do dvou uzlů musíme zabezpečit správně rozdělení jak rodičovských uzlů (postup nahoru), tak i uzlů potomků (postup dolů).



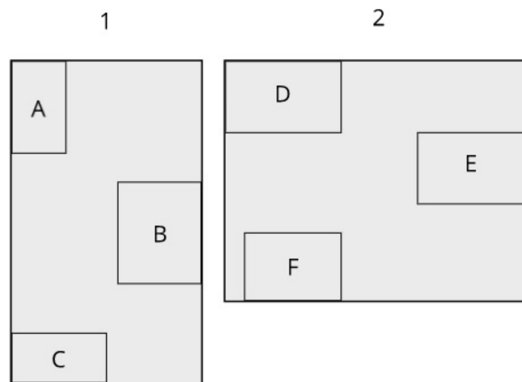
Obrázek 7 – Vkládání uzlu F do uzle 2

Na obrázku 7 vidíme vkládání uzle F do uzlu 2.



Obrázek 8 – Vkládání uzle F do uzle 2

Na obrázku 8 vidíme vkládání uzlu F do uzle 1.



Obrázek 9 – Finální vložení uzlu F do uzle 2

Metoda Insert vyhodnotila, že nejlepším kandidátem pro uložení uzlu F je uzel 2.

Algoritmus 21

Krok 1: Pokud N není listový uzel, potom pro každý záznam E uzlu N otestuj, jestli se překrývá MBR s MBR nově vkládaného záznamu I. Pokud ano, vyvolej metodu Insert (E, I), kde E je překrývající uzel a I je nově vkládaný záznam.

Krok 2: Pokud N je listový uzel, přidej záznam I do uzlu N. Pokud po přidání nového záznamu má uzel N více záznamů než M, vyvolej metodu SplitNode (N) a reorganizuj strom.

1.4.3 Metoda Delete

Metoda Delete funguje na principu nalezení shodného záznamu, který se nachází v listovém uzlu. Může ovšem dojít k situaci, kdy záznam, který chceme odstranit, se nachází ve více listových uzlech, proto musíme všechny dané listy projít a záznam odmazat. Často dochází po odmazání záznamu k deformaci stromu a je potřeba strom přeuspořádat.

Algoritmus 22

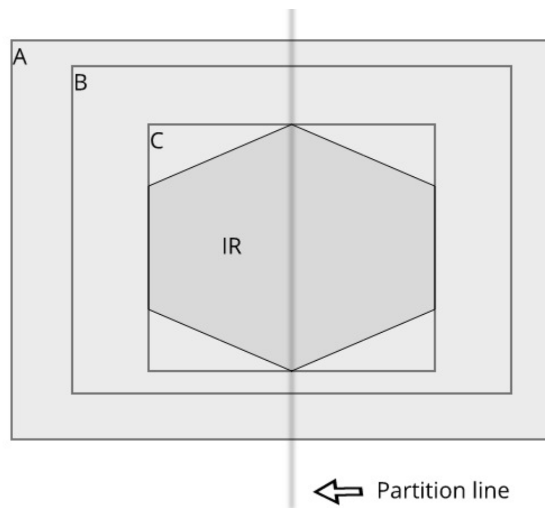
Krok 1: Pokud uzel N není list, pak pro každý záznam E z uzlu N otestuj, jestli se záznam E překrývá s obdélníkem mazané oblasti M. Pokud ano, vyvolej metodu Delete (E, M), kde E je uzel, který překrývá oblast M, a M je oblast ke smazání.

Krok 2: Pokud uzel N je list, odstraň z uzlu N záznam překrývající obdélník M. Uprav MBR uzlu N, aby zapouzdřoval všechny záznamy v uzlu N.

1.4.4 Splity

Při vkládáních nových záznamů do stromu nastávají situace, kdy začne být uzel přeplněn a je potřeba ho rozdělit (splitnout). V tomto případě využíváme metodu split. K co nejefektivnějšímu použití metody Split potřebujeme najít dobré rozdělení uzlů. Nejdříve uzel N, který chceme rozdělit na dva nové uzle, rozdělíme na dvě disjunktní oblasti. Poté najdeme dobré rozdělení (Partition), buďto vertikální, nebo horizontální a rozdělíme oblast na dvě podoblasti. Tímto získáme nejlepší rozdělení přeplněného uzlu a nové dvě podoblasti vložíme

do stromu. Nesmíme také zapomenout, že v R+-stromu není povoleno, aby se uzly překrývaly, proto když dochází ke splitu, musíme dbát na rekurzivní rozdělení rodičovských uzlů a uzlů potomků, jak je vidět na obrázku.



Obrázek 10 – Propagace rozděleni rodičovských uzlů a uzly potomků

Na obrázku vidíme rodičovský uzel A, potomek uzlu A je uzel B a potomek uzlu B je listový uzel C, který obsahuje indexový záznam IR. Může nastat situace, že uzel C se bude překrývat s jiným uzlem. Aby se nám uzel neprotínal s jiným uzlem, tento uzel rozdělíme do více uzlů a propagujeme tuto akci nahoru. Upravíme MBR jak uzlu B, tak i rodičovského uzlu A. Využíváme rekurzi pro rozdělení rodičovských uzlu A a B.

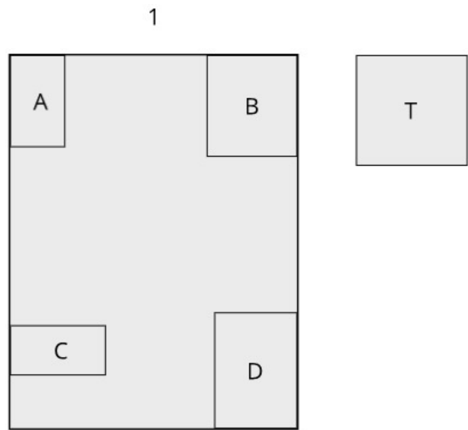
Algoritmus 23

Krok 1: Uzel N rozložíme podle metody Partition, ze které nám vzniknou dvě podoblasti O_1 *a* O_2*. Vytvoř pomocný uzel* $N_1(MBR_1, p_1)$ *a* $N_2(MBR_2, p_2)$*, které vnikly z podoblastí* O_1 *a* O_2*, kde platí, že* $MBR_N \cap MBR_1$ *a* $MBR_N \cap MBR_2$*.*

Krok 2: Vlož všechny uzle N_i *z uzle N do nových uzlů* N_t *pro t = 1,2, tak, že když platí, že* N_i *leží vně* N_1*, vlož* N_i *do uzle* N_1*, v opačném případě* N_i *leží vně* N_2 *, vlož uzel* N_i *do uzle* N_2*. Pokud však platí, že* $N_1 \cap N_i \neq N_1$ *nebo* $N_2 \cap N_i \neq N_2$*, proveď následující:*

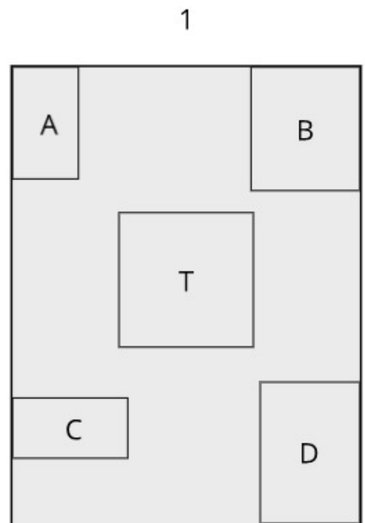
- *Pokud N je list, vlož* N_i *do obou nových uzlů.*
- *Jinak vyvolej metodu SplitNode pro rekurzivní split potomků. Kde* N_{i_1} *a* N_{i_2} *jsou dva uzly po splitu uzle* N_i*. Vlož tyto dva uzle do odpovídajícího uzle* N_t*.*

Krok 3: Pokud uzel N je kořen, vytvoř kořen nový s dvěma potomky N_1 *a* N_2*. Jinak nahraď kořenový uzel N uzlem NK a vlož do nového uzlu NK uzly* N_1 *a* N_2*, pokud má nový NK více záznamů než počet M, vyvolej metodu SplitNode. [7][8]*



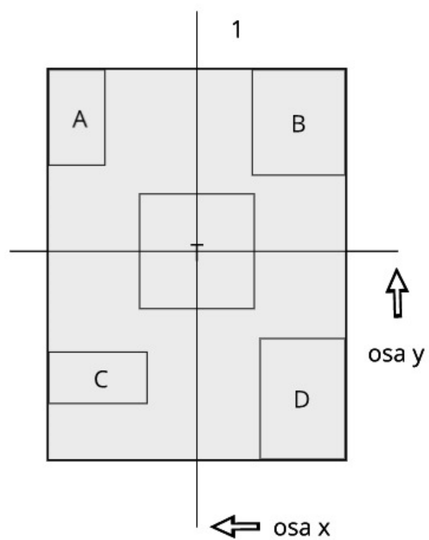
Obrázek 11 – Vložení záznamu T do uzle 1

Na obrázku 11 vidíme rozložení uzle 1, kde minimální velikost m je 2 a maximální velikost M jsou 4 a chceme vložit uzel T do uzle 1.

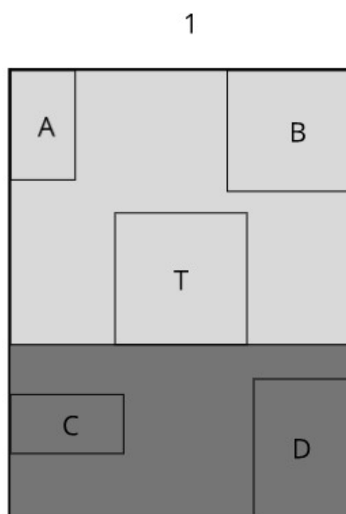


Obrázek 12 – Přetečení uzle 1

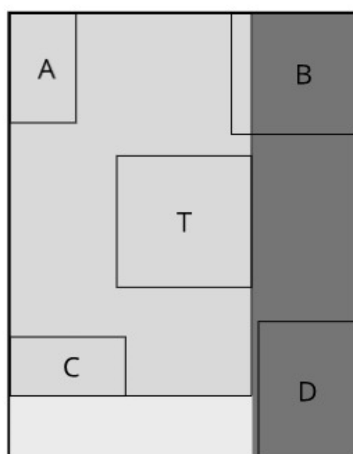
Na obrázku 12 vidíme přetečení uzle 1 a musíme tento uzel rozdělit. Nejdříve si zvolíme osu, podle které chceme uzel rozdělit. Vyvoláme metodu Partition, ze které získáme dvě skupiny nejideálnějších nových uzlů, které pak vložíme na místo uzle 1.



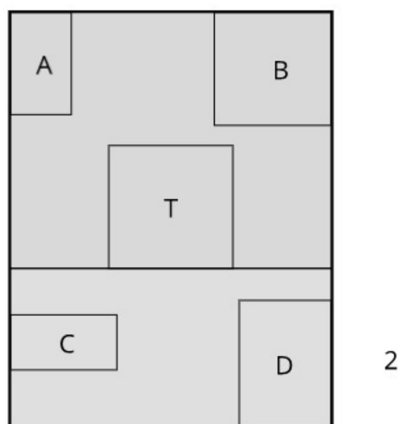
Obrázek 13 – Vyvolání metody sweep pro určení osy



Obrázek 14 – Partition podle osy y



Obrázek 15 – Partition podle osy x



Obrázek 16 – Rozdělení uzlu 1 do nových dvou uzlů

Na obrázku 14 a 15 vidíme, jak metoda Partition vybírá nejlepší dvě skupinu pro rozdělení do nových dvou uzlů. Na obrázku 16 vidíme výsledek algoritmu SplitNode.

1.4.5 Metoda Partition

K dosažení co nejlepšího splitu využíváme metodu Partition, tato metoda se nám stará o rozdělení splitu starého uzlu N do nových dvou uzlů N_1 a N_2, dosažené pomocí následujících kritérií:

1. Nejbližší soused
2. Minimální celkový prostor zabraný oblastí uzlu N_1 a N_2
3. Minimální počet splitů uzlů potomků splitovaného uzlu N

Metoda Partiton využívá metodu Sweep pro určení, podle jaké osy má uzle N splitnout.

Algoritmus 24

Krok 1: Pokud celkový rozkládaný prostor obsahuje méně nebo rovno fill-factor obdélníků, není prováděn žádný další rozklad, je vytvořen uzel N s těmito záznamy a algoritmus vrací (N, null)

Krok 2: Nechtě x a z jsou nejmenšími body osy x a osy y z MBR.

Krok 3: Vyvolej metodu Sweep(osa x, fillfactor, E), kde E jsou potomci z uzlu N, vypočítej tak hodnotu Cx na ose x.

Krok 4: Vyvolej metodu Sweep(osa y, fillfactor, E), kde E jsou potomci z uzlu N, vypočítej tak hodnotu Cy na ose y.

Krok 5: Vyber nejmenší hodnotu z hodnot Cx a Cy a podle nich rozděl MBR a jejich záznamy, kde je vytvořen uzel n, který obsahuje záznamy z první oblasti. Dále vytvoř uzel L, do kterého vlož záznamy z druhé podoblasti a vrať uzel N a L.

1.4.6 Metoda Sweep

Metoda Sweep seřazuje záznamy podle jednotlivých os. Vytváří dvě podoblasti, které naplňuje záznamy podle fill-factoru. Dle literatury je doporučeno nastavení fill-factoru od 50 % do 60 % z maximálního počtu záznamů M.

Algoritmus 25

Krok 1: Seřaď záznamy podle zadané osy (x nebo y), vytvoř první podoblast a do té vlož počet záznamů podle fill-factoru a do druhé podoblasti vlož zbytek.

Krok 2: Vypočítej celkovou cenu měřených vlastností použitých k utřiďování MBR. Vrať cenu pro každou osu.

1.5 Hilbertův strom

Hilbertův strom je hybridní datová struktura založená na základních principech R-stromu a B-stromu. Základním stavebním kamenem Hilbertova stromu je Hilbertovo číslo, které se nachází u každého objektu. Jedná se o prostřední bod objektu, tzv. centroid, který se nachází uprostřed každého objektu. Pro tento bod se vypočítá Hilbertovo číslo pomocí Hilbertovy křivky a dále se s každým objektem pracuje pomocí Hilbertova čísla. Hilbertova křivka bude vysvětlena později, ale v základním principu se jedná o plochu vyplňující křivku, která prochází celým definovaným prostorem a přepočítává jednotlivé body na Hilbertova čísla a prochází tak prostorem. Pokud dojde k přetečení uzlu a chceme rozdělit jeden uzel do 2 nových uzlů, nazýváme tuto metodu 1-to-2 rozdělení. Prozatím toto rozdělení odkládáme a čekáme, až nastane rozdělení uzlu 2-to-3. Obecně můžeme říci, že čekáme na s-to-(s+1) rozdělení uzlu. Pokud dojde k přetečení uzlu, pokusíme se do s – 1 záznamů vložit jeden ze záznamu (s – 1) do sourozenců. Pokud jsou všichni sourozenci plní, dochází k rozdělení s-to-(s+1) splitu. Tento s – 1 sourozenců jsou sourozenci daného uzlu.

Struktura Hilbertova stromu:

- Kořenový uzel obsahuje minimálně 2 uzle, pokud se nejedná o list.
- Každý listový uzel má minimální počet záznamů m a maximální počet M, uzel dále obsahuje pointer p na indexové záznamy a MBR.
- Nelistový uzel má minimální počet záznamů m a maximální počet M, uzel dále obsahuje ukazatel na potomky a MBR ohraničující všechny potomky.
- MBR obsahuje Hilbertovo číslo LHV (largest hilbert value).
- Každý uzel krom kořene obsahuje LHV.

1.5.1 Metoda Search

Metoda Search je velmi podobná metodě Search v R-stromu. Prohledáváme celý strom od kořene k listovým uzlům, kde hledáme shodu s požadovaným uzlem a vyhledávanou oblastí.

Algoritmus 26

Krok 1: Pokud uzel N je nelistový uzel, pak porovnej potomky s prohledávanou oblastí T, pokud potomek P má průnik s oblastí T, vyvolej metodu Search (P, T).

Krok 2: Pokud uzel N je listový uzel, vrať všechny uzly, které pokrývá oblast T.

1.5.2 Metoda Insert

Vkládání nového záznamu do Hilbertova R-stromu se liší v jeho uspořádání, na rozdíl od předešlých algoritmů, kde vkládání orientovalo podle MBR, u Hilbertova R-stromu se řídí podle Hilbertova čísla. Toto číslo obsahuje každé MBR, podle kterého pak můžeme jednotlivé záznamy porovnávat. Každý uzel obsahuje jen ty záznamy, které mají menší Hilbertovo číslo, než je číslo daného uzlu.

Algoritmus 27

Krok 1: máme záznam I, který chceme vložit do struktury, vyvoláme metodu ChooseLeaf (I_{MBR}, H), kde I_{MBR} *je MBR nově vkládaného záznamu a H je Hilbertovo číslo. Metodou ChooseLeaf získáme uzel, do kterého budeme záznam I vkládat.*

Krok 2: Pokud uzel N je prázdný, vložíme záznam I do uzle N. Pokud je uzel plný, vyvoláme metodu HandleOverflow(N, I), která nám rozdělí přeplněný uzel a vloží záznam I do požadovaného uzle.

Krok 3: Vyvolej metodu AdjustTree nad rodičem uzlu N.

Krok 4: Pokud došlo k rozlomení kořenového uzlu, vytvoř nový uzel a vlož do něj jeho dva nové potomky.

1.5.3 Metoda ChooseLeaf

Vrací uzel N, do kterého se má vložit nový záznam.

Algoritmus 28

Krok 1: Nastav do uzle N kořenový uzel K.

Krok 2: Pokud uzel N je list, vrať uzel N, pokud není, tak pro potomky uzle N otestuj, který potomek má nejmenší Hilbertovo číslo, ale větší než H vkládaného záznamu.

Krok 3: Pokud potomek P z uzle N není list, nastav do uzle N uzel P a vyvolej metodu ChooseLeaf.

1.5.4 Metoda AdjustTree

Metoda pro reorganizaci stromu. Začíná od listových uzlů a postupuje až ke kořeni, kde přepočítáme hodnotu jednotlivých MBR a velikost MBR daného uzlu na dané úrovni.

Algoritmus 29

Krok 1: Pokud N_p je kořen, skonči.

Krok 2: Do uzle N_p vložme předka uzlu N, pokud došlo ke splitnutí uzlu N, nechť NN je nový uzel. Vložíme uzel NN do uzle N_p, pokud je v něm místo. Pokud ne, vyvoláme metodu HadleOverflow(N_p, NN), pokud došlo ke splitu uzlu N_p, potom nechť je uzel LL novým uzlem.

Krok 3: Nechť ρ je množina všech uzlů předka P, přepočítejme hodnotu LHV a upravme MBR.

Krok 4: Nechť ϱ je množina rodičovských uzlů uzlu P, kde NN = PP, pokud byl uzel NP rozdělen. Opakujme krok 1.

1.5.5 Metoda Delete

V Hilbertově R-stromu se nemusíme starat o osiřelé uzly, které vzniknou například odmazáním v předkovi. Místo znovuvkládáním osamoceného uzle si půjčíme hodnoty LHV od sourozenců, nebo spojíme osamocený uzel s jeho sourozenci. Tento postup jsme schopni docílit díky čistému řazení Hilbertova R-stromu (díky LHV hodnotě). Můžeme si povšimnout, že v metodě Delete potřebujeme spolupracovat se sourozeneckými uzly S – 1.

Algoritmus 30

Krok 1: Najdi takový listový uzel L, který se shoduje se záznamem I, který chceme smazat.

Krok 2: Odstraň záznam I z uzlu L.

Krok 3: Pokud uzel L má méně záznamů než je m, tak vyber, jestli u všech sourozeneckých uzlů uzle L dochází k podtečení spoje ρ + 1 do ρ, kde ρ je množina uzlů, kde se nachází uzel L, a uprav uzly.

Krok 4: Z množiny uzlů ρ obsahujících uzel L vyvolej metodu AdjustTree.

1.5.6 Metoda HadleOverflow

Metoda HandleOverflow slouží při přetečení uzlu a jeho rozdělení, kde buď přesouvá záznamy z uzlu, nebo rozděluje záznamy v uzlu.

Algoritmus 31

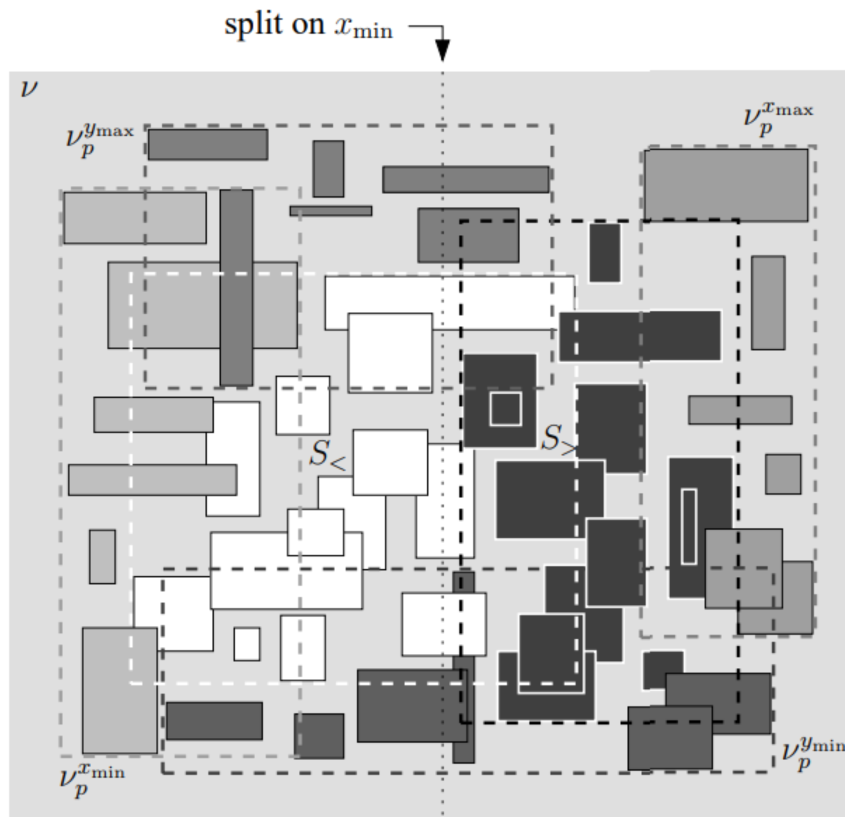
Krok 1: Nechť δ je množinou, která obsahuje všechny záznamy uzlu N.

Krok 2: Přidejme záznam I do množiny δ, pokud s – 1 spolupracujících sourozenců není plný. Přepočítej LHV pro uzel N.

Krok 3: Pokud množina δ je plná, vytvoř nový uzel NN a rovnoměrně množinu rozděl mezi uzly podle Hilbertova čísla a vrať uzel NN.

1.6 Pseudo PR-strom

V této diplomové práci se zabýváme 2D prostorem, proto se v této kapitole budeme bavit o dvoudimensionálním pseudo PR-stromu. Jako u R-stromu, tak i u pseudo-PR-tree, jehož vstupem je obdélník, platí, že každý listový uzel i prostřední uzel obsahuje MBR, který zapouzdřuje svoje potomky. Na rozdíl od R-stromu v pseudo PR-stromu nemusí být všechny vkládané záznamy ve stejné výšce. Tato datová struktura je založena na základech kd-stromu, kde vstupní uzel obsahuje minimální obdélník MBR a zde očekáváme 4 extra listové uzly, které jsou přidány níže pod každý střední uzel. Jsou to tzv. prioritní listy, které obsahují extrémní body v každé ze čtyř dimenzí, pro každou dimenzi máme ohraničující obdélník MBR, který zapouzdřuje potomky, kde pro každou extrémní hodnou rodičovského uzlu přiřadíme počet záznamů B. Pod pojmem extrémní hodnotou myslíme body min_x, min_y, max_x, max_y. [1] [4] [12]



Obrázek 17 – PR-strom zobrazení struktury [1]

Prvky PR-stromu:

- N – počet (dimensionálních) obdélníků uchovávaných ve stromu
- T – počet boxů reportovaných dotazem
- B – velikost diskového bloku

1.6.1 Struktura Pseudo PR-stromu

Představme si strukturu PR-stromu, nechť $S = \{R_1, \dots, R_n\}$, kde N je počet obdélníků v jedné rovině. Definice R_i^*:

$$R_i^* = (min_x(R_i), min_y(R_i), max_x(R_i), max_y(R_i))$$

Kde pro mapování R_i:

$$R_i = ((min_x(R_i), min_y(R_i)), (max_x(R_i), max_y(R_i)))$$

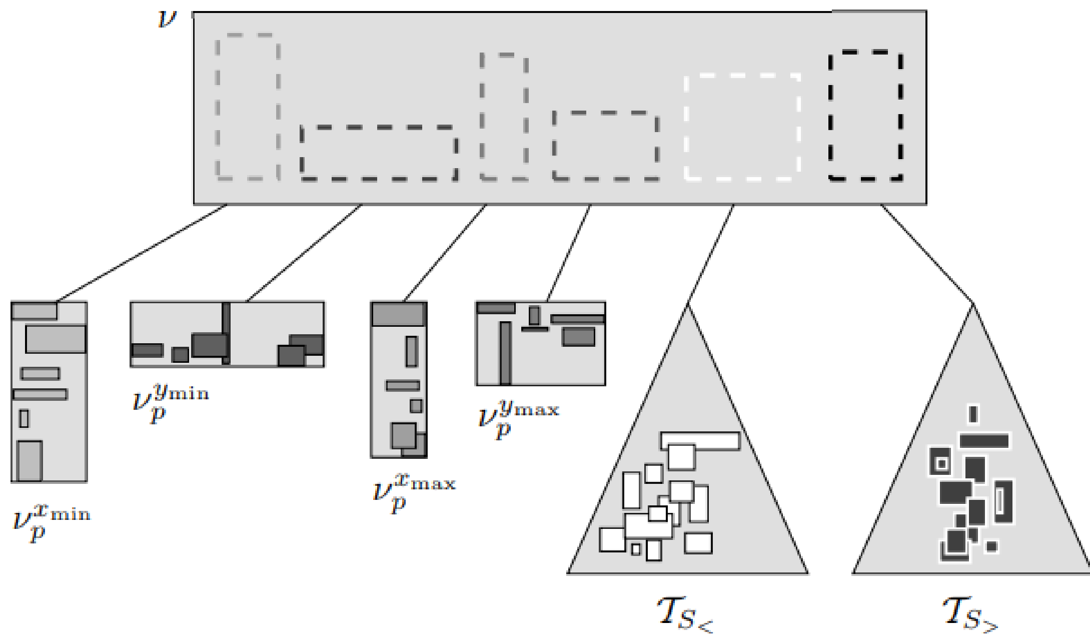
Jedná se o mapování pro bod ve čtyřech dimenzích. Takže máme počet záznamů B pro extrémní hodnoty obdélníku N a dále tu máme dva podstromy, které jsou tvořeny rekurzivně. Tyto dva podstromy T_s jsou založeny na množině S. Pokud množina S obsahuje méně obdélníků, než je počet B, potom T_s sestává pouze z jednoho listu, pokud však S obsahuje více listů, než jaký je počet B, potom T_s se skládá z uzlu v, který obsahuje 6 potomků, kde pro každého potomka v_s uzlu v platí, že jsou zapouzdřeny MBR uzlu v.

Prioritní uzly jsou sestaveny:

Souřadnice	Uzel v_p pro souřadnici	Počet záznamů B
min_x	$v_p^{min_x}$	B
min_y	$v_p^{min_y}$	B
max_x	$v_p^{max_x}$	B
max_y	$v_p^{max_y}$	B

Tabulka 1 – Rozřazení záznamů do uzlů

Z tabulky vidíme, že uzel $v_p^{min_x}$ obsahuje počet záznamů B s nejmenší souřadnicí min_x a takto to pokračuje pro další uzly. Tyto prioritní uzly obsahují extrémní obdélníky v množině S.



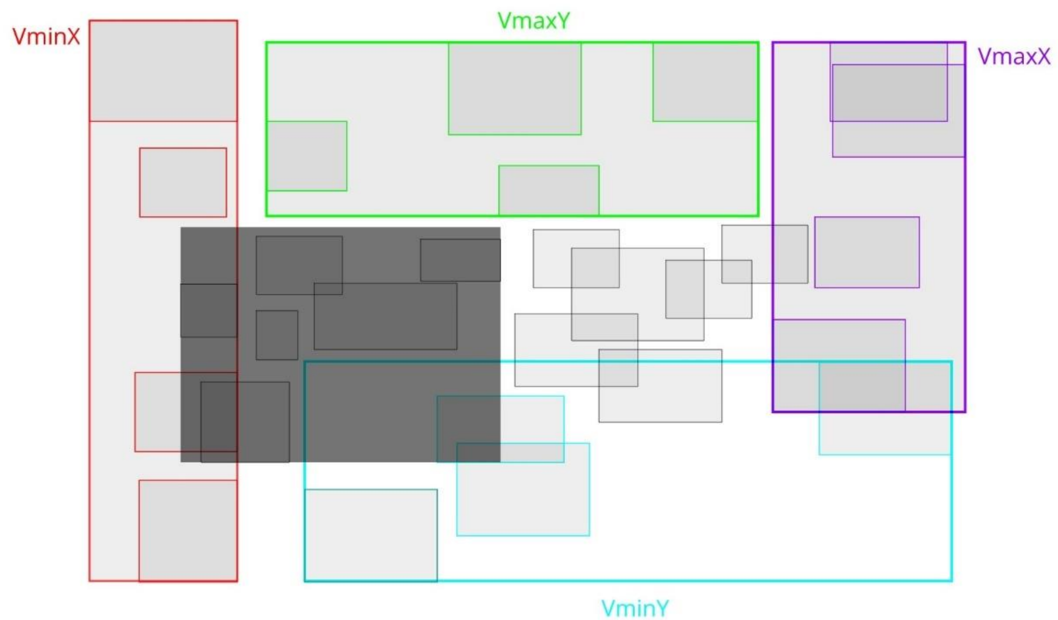
Obrázek 18 – Rozdělení záznamů do prioritních uzlů a rekurzivních uzlů [1]

Poté co vytvoříme prioritní uzly podle extrémních souřadnic, zbývá nám rozdělit množinu S_r, tuto množinu si rozdělíme na dvě podmnožiny, $S_<$ a $S_>$.

Na obrázku č.19 vidíme počáteční rozložení záznamů v prostoru.

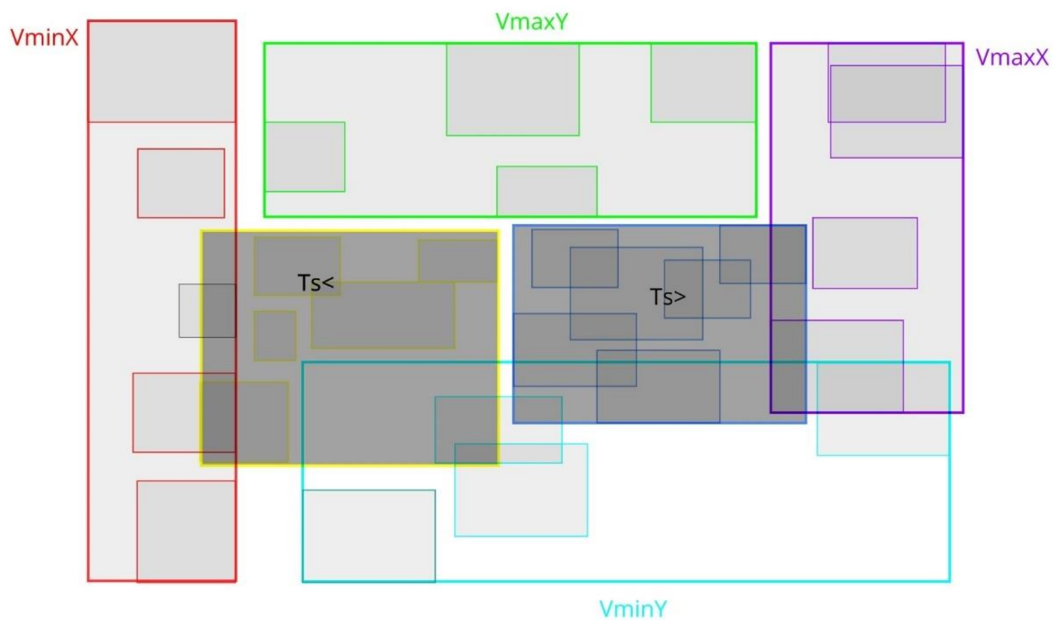
Obrázek 19 – Počáteční rozložení záznamů

Na následujícím obrázku vidíme rozložení do extrémních souřadnic, kde ke každé souřadnici jsou přiřazeny 4 záznamy. Zbývající záznamy jsou rozděleny podle Ts.



Obrázek 20 – Rozdělení podle extrémních souřadnic

Rozdělení podle Ts je založeno na metodě Round-Robin, kdy rozdělení zbývajících uzlů je nejdříve podle osy min_x, na další úrovni podle osy min_y, a takto se situace rekurzivně opakuje pro každou souřadnici v tomto pořadí (min_x, min_y, max_x, max_y) pomocí metody Round-Robin. Na obrázku 21 je vidět řez podle souřadnice min_x, kde je vybráno 6 záznamů, a dále pak zbytek je přiřazen pro Ts>.



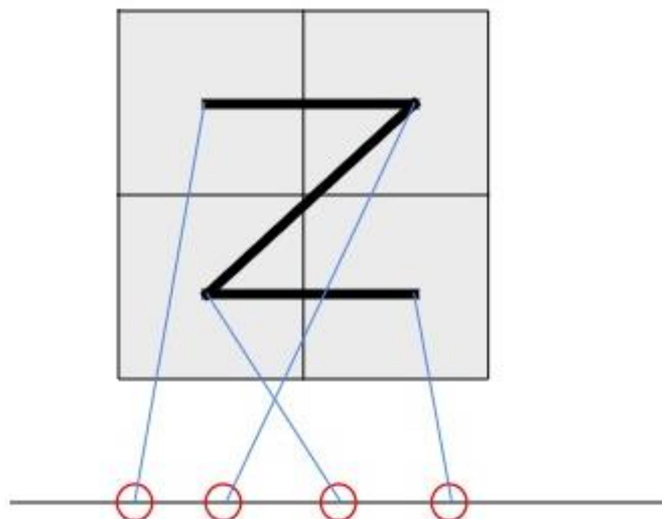
Obrázek 21 – Rozdělení zbývajících záznamů podle Ts

1.7 Prostor vyplňující křivky (Space-filling curves)

Prostor vyplňující křivky jsou křivky, které procházejí multidimenzionální prostor, kde převádění multidimenzionální prostor na jednodimensionální. Tyto křivky se používají zejména v datových strukturách, které podporují práci s multidimenzionálními daty. Jedná se o techniky indexování, kde index jednorozměrného bodu můžeme namapovat na index multidimenzionálního bodu. Zaměření této diplomové práce je na 2dimensionální prostor. Motivací pro využití prostor vyplňující křivky je možnost sestavení datových struktur typu R-strom ze statických dat, která mohou být náhodně vygenerovaná v prostoru. Díky těmto křivkám procházíme prostor a seřazujeme jednotlivé záznamy do množiny, ze které je pak vkládáme do datových struktur.

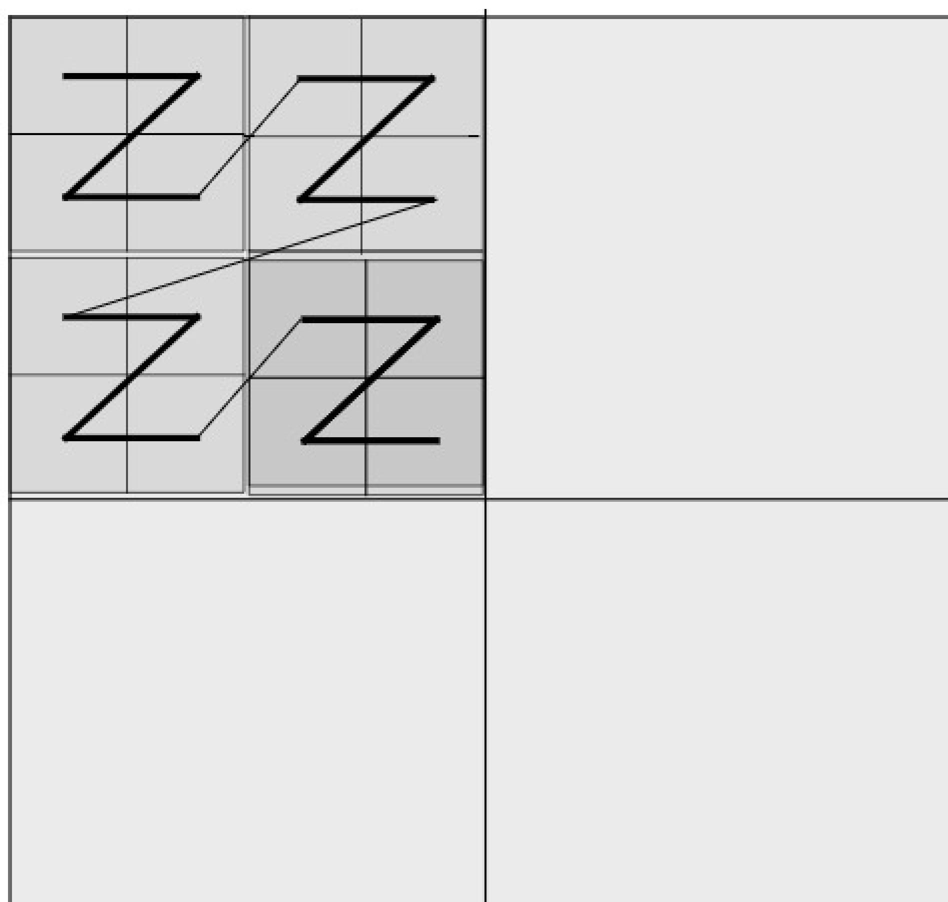
1.7.1 Mortonův rozklad (Z-order curve)

Z-order křivka převádí multidimenzionální data na jednodimensionální. Z-hodnota Mortonova rozkladu je vypočtena prokládáním binárních reprezentací čísel jeho souřadnicových hodnot. Vstupem pro převod na Z číslo je dvourozměrný bod, který se přeloží do binární soustavy, a poté sestavíme hodnotu čísla Z složením binární hodnoty souřadnice x a y. Vstupem je také velikost rozdělení do bloků, počet bitů, do kterých se má prostor rozdělit. [13]



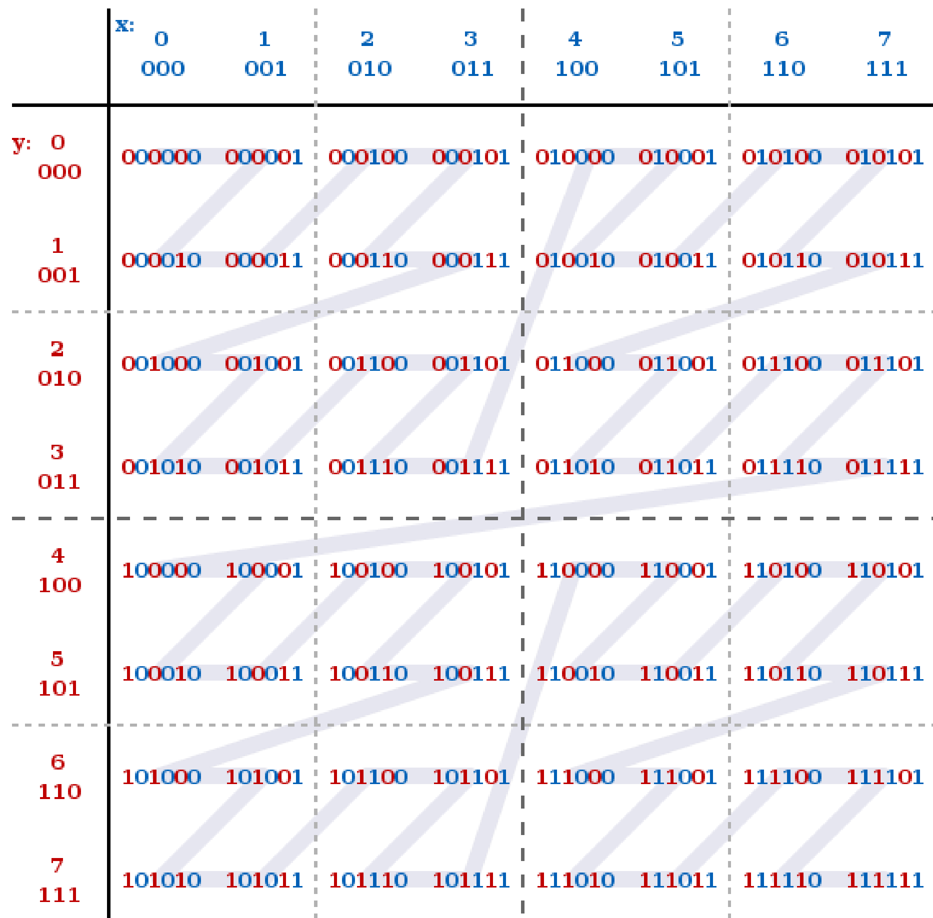
Obrázek 22 – Rozložení bodů Z-order křivky

Na tomto obrázku vidíme transformaci Z-order křivky do 1D prostoru.



Obrázek 23 – Rozložení Z-order křivky ve větším měřítku

Na obrázku 23 máme zobrazeno postupné procházení prostoru Z-Order křivkou, kde je vidět, jak na sebe body jednotlivě navazují. Z obrázku je i patrné i to, že prostor je rozložen na 8 bloků.

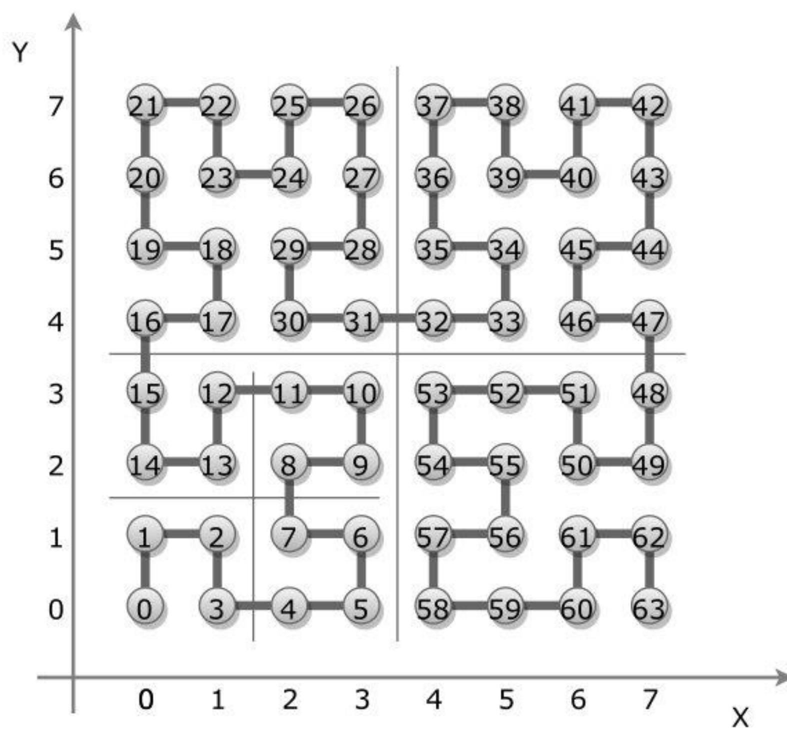


Obrázek 24 – Rozdělení prostoru Z-order křivkou [17]

Jak je vidět na obrázku, prostor je rozdělen na 7 bloků. Máme bod [0,0] a reprezentaci pro $x_0 = 000$ a $y = 000$, kde $z_0 = 000000$.

1.7.2 Hilbertova křivka

Podobně jako u Z-order křivky, tak i u Hilbertovy křivky převádíme multidimenzionální data na jednodimensionální data. V našem případě transformace z 2D do 1D. Jak můžeme vidět na obrázku 18, máme rozloženou plochu na 8 bloků, kde Hilbertova křivka začíná v levém dolním rohu a projde všechny body pouze jednou bez opakování. [7] [18]



Obrázek 25 – Rozložení bodů pomocí Hilbertovy křivky [7]

Algoritmus bez použití rekurzivního přístupu si rozkládá prostor z decimálního na binární.

2 VZOROVÉ IMPLEMENTACE R-STROMU

V této kapitole se zaměříme na implementaci datových struktur, kde bude názorně ukázaná implementace některých metod, kde bylo potřeba navrhnout i vlastní řešení algoritmů.

2.1 Implementace R-stromu

Při implementaci základní datové struktury R-strom bylo nutné si představit, jak strukturu rozdělit do jednotlivých tříd. Hlavním základním kamenem je třída MBR, která nám zapouzdřuje metody pro práci s MBR, jejich porovnávání, vypočet obvodu, protínání dvou obdélníků, obsažení jednoho obdélníku v druhém a podobně. Dále se budeme zabývat implementací některých z algoritmů R-stromu. Z důvodu čerpání z anglické literatury a vlastní implementace jednotlivých metod jsem ponechal anglické názvy. [3] [5] [12]

2.1.1 Implementace algoritmu Insert

V algoritmu Insert bylo potřeba si uvědomit, kolik různých stavů uzlů může nastat. Jelikož chceme nový záznam vkládat do uzlových listů, musíme se k němu nejprve dostat. K tomu nám pomůže metoda chooseLeaf, která nám vrátí listový uzel, do kterého budeme nový záznam vkládat. Pokud však je daný uzel plný, vyvolá se algoritmus splitNode, který přeplněný uzel rozdělí na dva nové uzle.

```
1.    public void insert(INode root, IRecord addRecord) {
2.         INode rTreeChooseLeaf = chooseLeaf(root, addRecord);
3.         try {
4.          if (rTreeChooseLeaf.getNodeKind().equals(NodeKind.LEAF)) {
5.           if (rTreeChooseLeaf.getDescendant().size() < maxEntries) {
6.            if (addRecord instanceof INode) {
7.             INode tempNode = (INode) addRecord;
8.             tempNode.setParent(rTreeChooseLeaf);
9.            }
10.           rTreeChooseLeaf.addRecord(addRecord);
11.           if (rTreeChooseLeaf.getCounter() == null ||
rTreeChooseLeaf.getCounter().equals(0)) {
12.            rTreeChooseLeaf.setCounter(uid.incrementAndGet());
13.           }
14.          } else {
15.           INode[] splitNodes = rTreeSplitNode(rTreeChooseLeaf,
addRecord);
16.           rTreeAdjustTree(splitNodes[0], splitNodes[1]);
17.          }
18.         }
19.         if (rTreeChooseLeaf.getNodeKind().equals(NodeKind.ROOT)) {
20.          if (rTreeChooseLeaf.getDescendant().size() < maxEntries) {
21.           rTreeChooseLeaf.addRecord(addRecord);
22.           if (rTreeChooseLeaf.getCounter() == null ||
rTreeChooseLeaf.getCounter().equals(0)) {
23.            rTreeChooseLeaf.setCounter(uid.incrementAndGet());
24.           }
25.           if (addRecord instanceof INode) {
26.            INode tempNode = (INode) addRecord;
27.            tempNode.setParent(rTreeChooseLeaf);
28.           }
29.          } else {
30.           INode[] splitNodes = rTreeSplitNode(rTreeChooseLeaf,
addRecord);
```

```
31.            rTreeAdjustTree(splitNodes[0], splitNodes[1]);
32.           }
33.          }
34.         } catch (Exception e) {
35.          e.printStackTrace();
36.         }
37.        }
```

Na řádku 4 a 19 se ptáme, jestli uzel je listový. Musíme si dát pozor na fakt, že listový uzel může být i kořen. Při vkládání prvních záznamů je kořen současně i listový uzel. Pokud jsme našli listový uzel, pokusíme se do něj vložit záznam, avšak nesmí dojít k překročení limitu. Když dojde k překročení povoleného limitu, vyvolá se metoda splitNode, která vytvoří dva nové uzle a po rozdělení uzlu dochází k vyvolání metody adjustTree, díky které dojde k reorganizaci stromu.

2.1.2 Implementace algoritmu AdjustTree

Algoritmus AdjustTree nám zajišťuje reorganizaci stromu od listového uzlu nebo jen od uzlu ke kořeni. Jelikož implementace tohoto algoritmu je dlouhá, bude rozdělena na 2 části. Toto je první část, kde začínáme krokem 2, kde na řádku jedna se ptáme, dokud uzel ukazatel rodiče n na potomka není null, což znamená, že kořen nemá ukazatel na žádného předka, když je uzel n kořen, tak skonči. Na řádku 4 si nastavíme předka uzlu n a procházíme všechny potomky uzlu předka p. Je to z toho důvodu, že mohlo dojít ke změně MBR a my ho tak musíme přepočítat, a tím zapouzdřujeme všechny MBR na dané úrovni od listu až po kořen.

```
1. // STEP 2
2.    while (n.getParent() != null) {
3.     //STEP 3
4.     INode parentP = n.getParent();
5.     for (IRecord iRecord: parentP.getDescendant()) {
6.      INode tempNode = (INode) iRecord;
7.      if (tempNode.equals(n)) {
8.       Mbr tempMbr = new Mbr();
9.       for (IRecord iRecord1: n.getDescendant()) {
10.       if (iRecord1 instanceof IndexRecord) {
11.        IndexRecord record = (IndexRecord) iRecord1;
12.        tempMbr.add(record.getMbr());
13.       } else if (iRecord1 instanceof INode) {
14.        INode nonLeafRecord = (INode) iRecord1;
15.        tempMbr.add(nonLeafRecord.getMbr());
16.       }
17.      }
18.      tempNode.setMbr(tempMbr);
19.     }
20.    }
21.    //STEP 4
22.    if (nn != null) {
23.     if (parentP.getDescendant().size() < maxEntries) {
24.      Mbr tempMbr = new Mbr();
25.      for (IRecord iRecord1: nn.getDescendant()) {
26.       if (iRecord1 instanceof IndexRecord) {
27.        IndexRecord record = (IndexRecord) iRecord1;
28.        tempMbr.add(record.getMbr());
29.       } else if (iRecord1 instanceof INode) {
```

```
30.         INode nonLeafRecord = (INode) iRecord1;
31.         for (IRecord iRecord2: nonLeafRecord.getDescendant()) {
32.          tempMbr.add(iRecord2.getMbr());
33.         }
34.        }
35.       }
36.
37.       NodeKind kind = NodeKind.NODE;
38.
39.       if (nn.getDescendant().iterator().next() instanceof IndexRecord) {
40.        kind = NodeKind.LEAF;
41.       }
42.
43.       INode nonRecord = new Node(minEntries, maxEntries, tempMbr, kind,
parentP);
44.       nonRecord.setDescendant(nn.getDescendant());
45.
46.       if (nonRecord.getCounter() == null ||
nonRecord.getCounter().equals(0)) {
47.        nonRecord.setCounter(uid.incrementAndGet());
48.       }
49.
50.       for (IRecord iRecord: nonRecord.getDescendant()) {
51.        if (iRecord instanceof INode) {
52.         INode tempNode = (INode) iRecord;
53.         tempNode.setParent(nonRecord);
54.        }
55.       }
56.
57.       parentP.getDescendant().add(nonRecord);
58.       nn = null;
59.      } else {
60.       INode[] rTreeSplitNode = rTreeSplitNode(parentP, nn);
61.       n = rTreeSplitNode[0];
62.       nn = rTreeSplitNode[1];
63.       continue;
64.      }
65.     }
66.     //STEP 5
67.     n = parentP;
68.
69.    }
```

Pokud však došlo k rozlomení uzlu, musíme podobnou akci udělat i pro nový uzel nn. Od řádku 22 až po řádek 65 pracujeme s novým uzlem nn, kde se nejprve ptáme, jestli rodičovský uzel, ve kterém se nachází nn, není plný. Pokud není, projdeme všechny potomky uzlu nn a přepočítáme celkové MBR. Zde se ještě rozhodujeme, jestli se jedná o záznamy nebo uzle, pro obě dvě entity je nutné přepočítat MBR. Dále od řádku 50 až 55 musíme nastavit potomkům rodiče nn. Tohle je velmi důležité, aby každý uzel měl ukazatele na potomky a na svého předka a indexové záznamy na potomky. Když nebudete opatrní, může to vést k deformaci stromu. Například se může stát, že vkládání bude fungovat, ale začnou se nad stromem provádět operace Delete. Může dojít k tomu, že se podaří odstranit uzel, který však má potomky, které se přiřadí do jiného uzle, a tady dojde k indikované chybě, sice nový uzel má pointer na své potomky, ale potomci mají pointer na starý uzel, a tím s nimi dále již nelze pracovat. Proto je potřeba kontrolovat, že každý uzel má ukazatel jak na rodiče, tak i na potomky vyjma kořene, ten má jen ukazatel na potomky. Pokud nastavíme potomky dále, pak přidáme uzel nn do množiny předka p. Pokud je uzel přeplněný, vyvoláme metodu split a uzel rozdělíme na uzel n a nn, kde

starý uzel nn je vložený v jednom z nich. Poslední krok je nastavení před p do uzle n, abychom mohli pokračovat ve stoupání od listu po kořen. V druhé části jsme se dostali do bodu, kdy jsme na úrovni kořene a máme dva uzly, uzel n a uzel nn, jak říká podmínka na řádku 1. Pokud nastane tato situace, musíme vytvořit nový kořenový uzel r, do kterého vložíme uzel n a nn. Uzlům n a nn nastavíme nového předka r a přepočítáme MBR nového uzlu r.

```
1. if (null == n.getParent() && nn != null && null == nn.getParent()) {
2.  INode root = new Node(minEntries, maxEntries, NodeKind.ROOT);
3.  root.addRecord(n);
4.  root.addRecord(nn);
5.  n.setParent(root);
6.  nn.setParent(root);
7.  this.root = root;
8.  Mbr tempMbr = new Mbr();
9.  for (IRecord iRecord2: root.getDescendant()) {
10.     INode nonLeafDescendant = (INode) iRecord2;
11.     tempMbr.add(nonLeafDescendant.getMbr());
12.    }
13.    NodeKind kind = NodeKind.NODE;
14.    if (n.getDescendant().iterator().next() instanceof IndexRecord) {
15.     kind = NodeKind.LEAF;
16.    }
17.    for (IRecord iRecord: n.getDescendant()) {
18.     if (iRecord instanceof INode) {
19.      INode tempNode = (INode) iRecord;
20.      tempNode.setParent(n);
21.     }
22.    }
23.    for (IRecord iRecord: nn.getDescendant()) {
24.     if (iRecord instanceof INode) {
25.      INode tempNode = (INode) iRecord;
26.      tempNode.setParent(nn);
27.     }
28.    }
29.    if (n.getCounter() == null || n.getCounter().equals(0)) {
30.     n.setCounter(uid.incrementAndGet());
31.    }
32.    if (nn.getCounter() == null || nn.getCounter().equals(0)) {
33.     nn.setCounter(uid.incrementAndGet());
34.    }
35.    n.setNodeKind(kind);
36.    nn.setNodeKind(kind);
37.    root.setMbr(tempMbr);
38.   }
39.
40.   } catch (Exception e) {
41.    e.printStackTrace();
42.   }
```

Od řádku 17 až po řádek 27 přenastavíme ukazatele na rodiče potomků uzlů n a nn na ně, abychom měli jistotu, že nám zde nevisí staré ukazatele na neexistující uzly.

2.1.3 Implementace algoritmu ConsendeseTree

ConsendeseTree se trochu podobá AdjustTree, ale na rozdíl od AdjustTree, který se volá při vkládání, ConsendeseTree se vyvolává při mazání záznamu ze stromu. Tento algoritmus zabraňuje, aby nastal stav nevyváženého stromu a také reorganizuje strom. Vysvětlení implementace algoritmu rozdělím opět do dvou částí. Tento algoritmus je navržen stejně jako AdjustTree od zdola ke kořeni. Dokud uzel n nemá prázdného rodiče, pokračuj. Na řádku 3 vytvoříme rodičovský uzel P z uzlu nodeL a na řádku 7 přiřadíme do pomocného uzle Ex uzel n, uzel Ex zde máme jako pomocnou proměnnou. Na řádku 11 se ptáme, zda nodeL má méně než m záznamů. Pokud je tato podmínka pravdivá, odstraníme pomocný uzel nodeEx z předka P a dále na řádku 13. Pokud projde podmínka, že nodeL má více záznamů než 0 a jeho potomci jsou záznamy (IndexRecordy), vezmeme tyto záznamy a vložíme do pomocného množiny setQ. Pokud potomci uzlu nodeL nejsou záznamy, ale jsou to uzly, nastává situace, kdy procházíme uzle, dokud nenarazíme na potomky, které jsou indexové záznamy, jak vidíme od řádku 18 až po řádek 27. Pokud takové najdeme, vložíme je do pomocné množiny setQ.

```
1.   Set < IRecord > setQ = new HashSet < > ();
2.   while (nodeL.getParent() != null) {
3.    INode parentP = nodeL.getParent();
4.    INode nodeEx = null;
5.    for (IRecord iRecord: parentP.getDescendant()) {
6.     if (iRecord.equals(nodeL)) {
7.      nodeEx = (INode) iRecord;
8.      break;
9.     }
10.      }
11.      if (nodeL.getSize() < minEntries) {
12.       parentP.getDescendant().remove(nodeEx);
13.       if (nodeL.getDescendant().size() > 0 &&
  nodeL.getDescendant().iterator().next() instanceof IndexRecord) {
14.        for (IRecord iRecord: nodeL.getDescendant()) {
15.         setQ.add(iRecord);
16.        }
17.       } else {
18.        Queue < IRecord > queue = new LinkedList < > ();
19.        queue.addAll(nodeL.getDescendant());
20.        while (!queue.isEmpty()) {
21.         IRecord poll = queue.poll();
22.         if (poll instanceof INode) {
23.          INode tempNode = (INode) poll;
24.          if (tempNode.getNodeKind().equals(NodeKind.LEAF)) {
25.           for (IRecord iRecord: tempNode.getDescendant()) {
26.            setQ.add(iRecord);
27.           }
28.          } else {
29.           queue.addAll(tempNode.getDescendant());
30.          }
31.
32.         }
33.        }
34.       }
35.       if (parentP.getDescendant().contains(nodeL)) {
36.        Mbr tempMbr = new Mbr();
37.        for (IRecord iRecord: nodeL.getDescendant()) {
38.         tempMbr.add(iRecord.getMbr());
39.        }
```

```
40.         nodeL.setMbr(tempMbr);
41.        }
42.       }
43.       nodeL = nodeL.getParent();
44.      }
```

Na řádku 35 se ptáme, zda se pořád nachází uzel nodeL v rodičovském uzle P. Pokud ano, přepočítáváme MBR uzlu P. Abychom mohli dále pokračovat v traversování stromu směrem nahoru, na řádku 43 nastavíme do nodeL jeho předka. V druhé části se dostáváme do situace, kdy kořenový uzel R má méně než m záznamů, jak vidíme na řádku 1. Pokud aktuální uzel nodeL (tedy kořenový uzel) má méně než m záznamů a zároveň jeho potomci jsou indexové záznamy, vložíme tyto záznamy do množiny setQ. Pokud nejsou indexovými záznamy, ale jedná se o uzle, vybíráme tyto uzle a procházíme je, dokud nenajdeme všechny záznamy, které pak vkládáme do pomocné množiny setQ. Nakonec tuto množinu vyprázdníme vkládáním jednotlivých záznamů do stromu; řádek 27.

```
1. if (nodeL.getNodeKind().equals(NodeKind.ROOT) &&
   nodeL.getDescendant().size() < minEntries) {
2.       if (nodeL.getDescendant().size() > 0 &&
nodeL.getDescendant().iterator().next() instanceof IndexRecord) {
3.        for (IRecord iRecord: nodeL.getDescendant()) {
4.         setQ.add(iRecord);
5.        }
6.       } else {
7.        Queue < IRecord > queue = new LinkedList < > ();
8.        queue.addAll(nodeL.getDescendant());
9.        while (!queue.isEmpty()) {
10.        IRecord poll = queue.poll();
11.        if (poll instanceof INode) {
12.         INode tempNode = (INode) poll;
13.         if (tempNode.getNodeKind().equals(NodeKind.LEAF)) {
14.          for (IRecord iRecord: tempNode.getDescendant()) {
15.           setQ.add(iRecord);
16.          }
17.         } else {
18.          queue.addAll(tempNode.getDescendant());
19.         }
20.
21.        }
22.       }
23.      }
24.      nodeL.getDescendant().clear();
25.     }
26.     for (IRecord iRecord: setQ) {
27.      insert(root, iRecord);
28.     }
```

Jak zde vidíme, algoritmus ConsendenseTree vybírá indexové záznamy a znovu je vkládá do stromu. Tímto je zaručené, že struktura stromu se bude pořád měnit.

2.2 Implementace R*-stromu

V této kapitole se zaměříme na implementaci R*-stromu a rozdílu mezi R*-stromem a R-stromem. [1] [10] [11] [12]

2.2.1 Implementace algoritmu Insert

Algoritmus Insert se skládá ze dvou částí. Z metody Insert a metody InternalInsert, kde metoda Insert slouží k získání flagu pro první vkládání. Tento flag využijeme dále v algoritmu overflowTreatment.

```
1. public void insert(INode node, IRecord rec) {
2.         insertInternal(node, rec);
3.         flag = true;
4.     }
```

Jak vidíme na ř. 3, máme zde flag = true po prvním vložení. Tento flag využíváme pro identifikaci, jestli máme v metodě overflowTreatment volat metodu re-insert, nebo metodu split.

```
1. private void insertInternal(INode node, IRecord rec) {
2.     List < IRecord > chooseSubtrees = new ArrayList < > ();
3.     chooseSubtree2(node, rec, chooseSubtrees);
4.     INode chooseSubtree = (INode) chooseSubtrees.get(0);
5.     if (chooseSubtree.getDescendant().size() < maxEntries) {
6.      chooseSubtree.addRecord(rec);
7.      Mbr tempMbr = new Mbr();
8.      for (IRecord iRecord: chooseSubtree.getDescendant()) {
9.       tempMbr.add(iRecord.getMbr());
10.     }
11.     chooseSubtree.setMbr(tempMbr);
12.     if (rec instanceof INode) {
13.      INode tempNode = (INode) rec;
14.      tempNode.setParent(chooseSubtree);
15.     }
16.    } else {
17.     overflowTreatment(chooseSubtree, rec);
18.    }
19.   }
```

Jak vidíme na řádku 17, pokud daný uzel, kam chceme vkládat nový záznam, má více prvků, než je M, vyvoláme metodu overflowTreatment. Jinak vložíme nový záznam do uzlu, který jsme našli podle metody chooseSubtree, a upravíme MBR uzlu.

2.2.2 Implementace algoritmu overflowTreatment

Implementace této metody je založena právě na rozhodováním flagu z předešlé metody Insert. Z ř. 3 vidíme podmínku, pokud uzel není kořen a flag = true, vyvolej metodu reInsert, pokud již došlo k vyvolání metody reInsert, vyvolej naopak metodu split a potom metodu AdjustTree, která nám reorganizuje celý strom.

```
1. private void overflowTreatment(INode node, IRecord rec) {
2.      boolean callOverFlow = false;
3.      if (!node.getNodeKind().equals(NodeKind.ROOT) && flag) {
4.       flag = false;
5.       reInsert(node, rec);
6.      } else {
```

```
7.         SeparateGroups split = split(node, rec);
8.         rTreeAdjustTree(split.getGroup1(), split.getGroup2());
9.        }
10.     }
```

2.2.3 Implementace metody reInsert

Metoda reInsert slouží k znovuvložení prvků z daného uzlu. V literatuře jsem se dočetl, že nejlepší poměr pro výběr počtu uzlů, které mají být znovu vloženy, je 30 % z celkového počtu M. Na ř. 2 vidíme výpočet čísla, kolik uzlů má být znovu vloženo do stromu. Od ř. 4 až po ř. 16 vypočítáme hodnou centroidu každého z potomků uzlu n a seřadíme je od nejmenšího po největší a poté vybereme prvních p záznamů, které z množiny odstraníme, to je od ř. 17 až po ř. 30. Dále upravíme MBR uzlu n a vložíme po jednom všechny záznamy zpět do stromu.

```
1. private void reInsert(INode node, IRecord rec) {
2.     int p = roundUp((maxEntries * 0.3));
3.     IRecord[] removeRecs = new IRecord[p];
4.     List < IRecord > recordList = new ArrayList<>(node.getDescendant());
5.     recordList.add(rec);
6.     for (int i = 0; i < recordList.size(); i++) {
7.      for (int j = i; j < recordList.size(); j++) {
8.       int centroid1 =
node.getMbr().calculateDistance(recordList.get(i).getMbr());
9.       int centroid2 =
node.getMbr().calculateDistance(recordList.get(j).getMbr());
10.      if (centroid1 < centroid2) {
11.       IRecord temp = recordList.get(i);
12.       recordList.set(i, recordList.get(j));
13.       recordList.set(j, temp);
14.      }
15.     }
16.    }
17.    for (int i = 0; i < p; i++) {
18.     removeRecs[i] = recordList.get(i);
19.
20.    for (IRecord iRecord: recordList) {
21.     if (!node.getDescendant().contains(iRecord)) {
22.      node.getDescendant().add(iRecord);
23.     }
24.
25.    for (int i = 0; i < removeRecs.length; i++) {
26.     IRecord removeRec = removeRecs[i];
27.     if (node.getDescendant().contains(removeRec)) {
28.      node.getDescendant().remove(removeRec);
29.     }
30.    }
31.    Mbr tempMbr = new Mbr();
32.    for (IRecord descendant: node.getDescendant()) {
33.     tempMbr.add(descendant.getMbr());
34.    }
35.    node.setMbr(tempMbr);
36.    for (int i = 0; i < removeRecs.length; i++) {
37.     IRecord removeRec = removeRecs[i];
38.     insertInternal(root, removeRec);
39.    }
40.   }
```

Jak jsem již zmínil, v tomto kroku, kdy znovu vkládáme záznamy do struktury, dojde k vyvolání metody insertInternal, ten poprvé na této úrovni má nastavený flag = true, ale při

prvním vyvolání metody reInsert se nastaví flag = false a dále pak pokračuje vyvoláním metody split.

2.2.4 Implementace metody split

Hlavním rozdílem metody split u R*-stromu je výběr splitIndexu. Využil jsem jako pomocnou třídu třídu SeparateGroups, do které se ukládají mezi výsledky jednotlivých splitů. Jak je vidět, ř. 8 nejdříve vybereme podle toho, jaké osy chceme split vyvolat. Poté na ř. 9 díky metodě chooseSplitIndex dostaneme dvojici nových uzlů, které vzniknou z původního uzlu node.

```
1. private SeparateGroups split(INode node, IRecord rec) {
2.     List < IRecord > tempList = new ArrayList < > ();
3.     tempList.addAll(node.getDescendant());
4.     tempList.add(rec);
5.
6.     INode parent = node.getParent();
7.
8.     Axis chooseSplitAxis = chooseSplitAxis(tempList);
9.     SeparateGroups chooseSplitIndex = chooseSplitIndex(chooseSplitAxis,
tempList);
10.
11.    chooseSplitIndex.getGroup1().setParent(parent);
12.    node.getDescendant().clear();
13.
node.getDescendant().addAll(chooseSplitIndex.getGroup1().getDescendant());
14.    Mbr tempMbr = new Mbr();
15.
16.    for (IRecord iRecord: node.getDescendant()) {
17.     tempMbr.add(iRecord.getMbr());
18.    }
19.
20.    node.setMbr(tempMbr);
21.
22.    chooseSplitIndex.setGroup1(node);
23.
24.    chooseSplitIndex.getGroup2().setParent(parent);
25.    chooseSplitIndex.getGroup2().setNodeKind(node.getNodeKind());
26.
27.    for (IRecord iRecord: chooseSplitIndex.getGroup1().getDescendant())
{
28.     if (iRecord instanceof INode) {
29.      INode tempNode = (INode) iRecord;
30.      if (!tempNode.getParent().equals(chooseSplitIndex.getGroup1())) {
31.       tempNode.setParent(chooseSplitIndex.getGroup1());
32.      }
33.     }
34.    }
35.    for (IRecord iRecord: chooseSplitIndex.getGroup2().getDescendant())
{
36.     if (iRecord instanceof INode) {
37.      INode tempNode = (INode) iRecord;
38.      if (!tempNode.getParent().equals(chooseSplitIndex.getGroup2())) {
39.       tempNode.setParent(chooseSplitIndex.getGroup2());
40.      }
41.     }
42.    }
43.
44.    if (chooseSplitIndex.getGroup1().getCounter() == null ||
chooseSplitIndex.getGroup1().getCounter().equals(0)) {
```

```
45.      chooseSplitIndex.getGroup1().setCounter(uid.incrementAndGet());
46.     }
47.     if (chooseSplitIndex.getGroup2().getCounter() == null ||
chooseSplitIndex.getGroup2().getCounter().equals(0)) {
48.      chooseSplitIndex.getGroup2().setCounter(uid.incrementAndGet());
49.     }
50.
51.     return chooseSplitIndex;
52.
53.   }
```

Proměnná chooseSplitIndex je objekt třídy SeparateGroups, která obsahuje dva uzle. Tyto dva uzle jsou naše finální nové uzle, které se vkládají do stromu. Od ř. 27 až po řádek 49 vybíráme jednotlivou groupu (nový uzel), přepočítáme mu MBR a potomkům nastavíme ukazatel na nového předka. Rozdíl od klasického R-stromu je v přístupu nacházení nejlepších dvou nových uzlů, které se mají rozdělit při splitu. Vypočítáváme zde, jak celkovou pokrytou plochu nově vytvořených uzlů, ale i překrytí těch dvou uzlů, které se snažíme minimalizovat, tak i minimalizaci velikosti MBR obou uzlů.

2.2.5 Pomocná metoda pro výpočet okrajové plochy

Metoda pro výpočet okrajové plochy nám bere jako parametr metody list záznamů. Pro počet iterací, pro které se mají záznamy rozdělit, je:

$$M - (2 * m) + 2$$

Toto číslo nám udává počet iterací, pro které se má výpočet plochy provést. Například máme m = 2 a M = 4, tedy počet iterací, pro které se skupiny mají rozdělit, je 2. Dále tu máme pomocnou $sorting = m - 1 + i$, kde m je min. počet záznamů a i je iterace kroku od 1 do 2.

Jak vidíme v tabulce, první skupina bude obsahovat 2 záznamy, druhá 3 a v druhé iteraci to bude naopak. Tímto vypočítáme pokrytí plochy a vybereme dvojici s nejmenším pokrytou plochou.

Iterace	Skupina 1 počet prvků	Skupina 2 počet záznamů
1.	2	3
2.	3	2

Tabulka 2 – Rozdělení záznamů do uzlů

```
1. private SeparateGroups calculateMarginAreaOfGroups(List < IRecord >
   tempList) {
2.      SeparateGroups minGroups = null;
3.      int maxK = maxEntries - (2 * minEntries) + 2;
4.      int minK = 1;
5.      int m = minEntries;
6.
7.      Mbr tempMbr1 = new Mbr();
8.      Mbr tempMbr2 = new Mbr();
9.
10.     int tempMinArea = Integer.MAX_VALUE;
11.     for (int i = minK; i <= maxK; i++) {
12.      int sorting = m - 1 + i;
```

```
13.     int counter = 0;
14.     INode rec1 = new Node(minEntries, maxEntries, tempMbr1,
NodeKind.LEAF, null);
15.     INode rec2 = new Node(minEntries, maxEntries, tempMbr2,
NodeKind.LEAF, null);
16.     for (IRecord iRecord: tempList) {
17.      if (counter < sorting) {
18.       tempMbr1.add(iRecord.getMbr());
19.       rec1.getDescendant().add(iRecord);
20.      } else {
21.       tempMbr2.add(iRecord.getMbr());
22.       rec2.getDescendant().add(iRecord);
23.      }
24.      counter++;
25.     }
26.     if (tempMbr1.calculateMarginArea(tempMbr2) < tempMinArea) {
27.      minGroups = new SeparateGroups(rec1, rec2);
28.      tempMinArea = tempMbr1.calculateMarginArea(tempMbr2);
29.     }
30.    }
31.    return minGroups;
32.   }
```

Jak je vidět, výpočet počtu iterací je na ř. 3. Výpočet pomocné sorting na ř. 12 a dále pak rozřazování do skupin a výběr skupiny s nejmenším. Obdobně funguje metoda pro výpočet Overlap mezi dvěma uzly.

2.3 Implementace R+-stromu

R+-strom je jiný než předešlé dvě datové struktury, liší se v tom, že žádný z uzlů se nesmí překrývat. Jedině indexové záznamy se mohou překrývat. Také jeden uzel může ležet ve více rodičovských uzlech, jelikož daný uzel byl buď přeplněn, nebo se nacházel v uzlu, který byl též přeplněn a byla potřeba provést metodu split. Tím se uzel dostal do více rodičovských uzlů. U metody insert nastává stav, kdy dochází ke špatnému vkládání do stromové struktury a dochází tak k deformaci stromu. Jedná se o vkládání záznamu mezi více uzlů, kdy je třeba záznam rozřadit do více jak jednoho uzlu a v tomto okamžiku dochází ke špatnému rozdělení. Dále je zde ještě jeden stav kdy záznam, který se vkládá do více jak 2 uzlů, dochází k jeho špatnému rozlomení a tím také k deformaci stromu. Když nastane situace, kdy dojde k deformaci stavu může to ovlivnit jak vyhledávání záznamu, tak i jejich odmazávání.

[7] [9] [12]

2.3.1 Metoda Insert

Metoda Insert se od předchozích metod liší propagací velikosti MBR jak rodičů, tak i potomků. Může nastat případ, kdy záznam je vložen do dvou či více uzlů, kde může dojít ke splitu, ale také nemusí. Je potřeba zařídit přepočítání ohraničujících MBR rodičovských uzlů, aby se navzájem nepřekrývaly. V prvním kroku nejdříve zjistíme, jestli vkládáme záznam do

kořenového uzlu, nebo již do listových uzlů, či je potřeba najít správný listový uzel průchodem všech uzlů. Na řádku 2 až 7 vidíme vkládání a split kořenového uzlu.

```
1. public void newInsert(INodePlusNew node, IRecordPlusNew record) throws
   Exception {
2.  if (node.getNodeKind().equals(NodeKind.ROOT) &&
   (node.getDescendant().isEmpty() ||
   node.getDescendant().iterator().next() instanceof IndexRecordPlus)) {
3.   node.addRecord(record);
4.   if (node.getDescendant().size() > maxEntries) {
5.    NodePlusNew[] array = splitNode(node);
6.    rTreeAdjustTree(array[0], array[1], null);
7.   } else {
8.    propagationBoundDownWithoutSplit(node, record);
9.   }
10.    } else {
11.  if ((node.getNodeKind().equals(NodeKind.ROOT) ||
    node.getNodeKind().equals(NodeKind.NODE)) &&
    node.getDescendant().iterator().next().getNodeKind().equals(NodeKind.LE
    AF)) {
12.
13.      boolean addNode = false;
14.      List < IRecordPlusNew > stack = new LinkedList < > ();
15.      for (IRecordPlusNew iRecord: node.getDescendant()) {
16.       INodePlusNew tempNode = (INodePlusNew) iRecord;
17.       if (tempNode.getMbrPlus().overlapWithMbr(record.getMbrPlus())) {
18.        stack.add(tempNode);
19.       }
20.      }
21.      if (stack.isEmpty()) {
22.       List < IRecordPlusNew > findNode = new ArrayList < > ();
23.       chooseSubtree2(node, record, findNode);
24.       for (IRecordPlusNew iRecord: findNode) {
25.        newInsert((INodePlusNew) iRecord, record);
26.       }
27.      } else {
28.       for (IRecordPlusNew iRecord: stack) {
29.
30.        INodePlusNew tempNode = (INodePlusNew) iRecord;
31.        if (tempNode.getDescendant().size() + 1 > maxEntries) {
32.         tempNode.addRecord(record);
33.         NodePlusNew[] array = splitNode(tempNode);
34.         rTreeAdjustTree(array[0], array[1], null);
35.        } else {
36.         propagationBoundDownWithoutSplit(tempNode, record);
37.        }
38.       }
39.      }
40.     }
```

Od řádku 11 až po řádek 34 řešíme, jestli daný uzel je kořenový, nebo již střední uzel, kde se snažíme najít listový uzel, do kterého je optimální nový listový uzel vložit. Po vložení do struktury přepočítáme rodičovské MBR.

```
41.    if (node.getNodeKind().equals(NodeKind.LEAF)) {
42.     System.out.println("3");
43.     node.addRecord(record);
44.     if (node.getDescendant().size() > maxEntries) {
45.      splitNode(node);
```

```
46.      } else {
47.       propagationBoundDownWithoutSplit(node, record);
48.      }
49.     }
50.
51.    if ((node.getNodeKind().equals(NodeKind.NODE) ||
node.getNodeKind().equals(NodeKind.ROOT)) &&
node.getDescendant().iterator().next().getNodeKind().equals(NodeKind.NODE))
{
52.     System.out.println("4");
53.     Set < IRecordPlusNew > clone = new HashSet < >
(node.getDescendant());
54.     for (IRecordPlusNew iRecord: clone) {
55.      INodePlusNew tempNode = (INodePlusNew) iRecord;
56.      if (node.getMbrPlus().overlapWithMbr(tempNode.getMbrPlus())) {
57.       newInsert(tempNode, record);
58.      }
59.     }
60.    }
61.   }
```

2.3.2 Metoda Sweep

Metoda Sweep slouží pro vybrání nejlepší dvojice uzlů a rozdělení jejich potomků pro provedení splitu přeplněného uzlu. Na řádku 3 a 4 si vytvoříme pomocné proměnné group1 a group2, které nám budou sloužit pro rozdělení záznamů do dvou skupin. Pomocné mbr1 a mbr2 zapouzdřují dočasnou velikost MBR pokrývající prostor. Pomocí fill factoru a nastavení osy, podle které chceme provádět split, seřadíme záznamy podle nejmenších souřadnic dané osy a vytvoříme pomocné group1, kam naplníme podle fillFactoru počet záznamů a rozdělíme záznamy podle největší hodnoty, pro jakou osu chceme řezat, a tu nastavíme jako hranici, jak je vidět od řádku 12 po řádek 40.

```
1. private SeparateGroupsPlusNew sweep(Axis axis, int fillFactor, List <
   IRecordPlusNew > recordList) {
2.      SeparateGroupsPlusNew group = new SeparateGroupsPlusNew();
3.      INodePlusNew group1 = new NodePlusNew();
4.      INodePlusNew group2 = new NodePlusNew();
5.
6.      MbrPlusNew mbr1 = new MbrPlusNew();
7.      MbrPlusNew mbr2 = new MbrPlusNew();
8.      int bound = Integer.MIN_VALUE;
9.      int count = 0;
10.     for (IRecordPlusNew iRecord: recordList) {
11.      logger.debug("[SWEEP] FF : " + fillFactor + ", count : " + count);
12.      if (count < fillFactor) {
13.       if (iRecord instanceof INodePlusNew) {
14.        INodePlusNew temp = (INodePlusNew) iRecord;
15.
16.        mbr1.add(temp.getMbrPlus());
17.
18.        if ((count + 1) == fillFactor) {
19.         if (Axis.AXISX.equals(axis)) {
20.          bound = temp.getMbrPlus().getMaxX();
21.         } else {
22.          bound = temp.getMbrPlus().getMaxY();
23.         }
24.        }
25.       } else {
26.        IndexRecordPlus indexRec = (IndexRecordPlus) iRecord;
```

```
27.        mbr1.add(indexRec.getMbrPlus());
28.
29.        if ((count + 1) == fillFactor) {
30.         if (Axis.AXISX.equals(axis)) {
31.          bound = indexRec.getMbrPlus().getMaxX();
32.         } else {
33.          bound = indexRec.getMbrPlus().getMaxY();
34.         }
35.        }
36.       }
37.       count++;
38.      } else {
39.       if (iRecord instanceof INodePlusNew) {
40.        NodePlusNew temp = (NodePlusNew) iRecord;
41.
42.        mbr2.add(temp.getMbrPlus());
43.       } else {
44.        IndexRecordPlus indexRec = (IndexRecordPlus) iRecord;
45.        mbr2.add(indexRec.getMbrPlus());
46.       }
47.      }
48.     }
```

V další části algoritmu Sweep nastavujeme hranice, jestli se jedná o hranici pro osu x, anebo pro osu y, jak vidíme od řádku 49 až po řádek 64. Poté vezmeme pomocné MBR a vložíme do nich záznamy podle rozdělení do jednotlivých skupin.

```
49.   if (axis.equals(Axis.AXISX)) {
50.     mbr1.setBoundX(bound);
51.     mbr1.setDirectionX(MbrPlusNew.Direction.LEFT);
52.     logger.debug("[SWEEP] [X] mbr1[" + axis + "] : minX,minY,maxX,maxY
:[" + mbr1.getMinX() + "," + mbr1.getMinY() + "," + mbr1.getMaxX() + "," +
mbr1.getMaxY());
53.     mbr2.setDirectionX(MbrPlusNew.Direction.RIGHT);
54.     mbr2.setBoundX(bound);
55.
56.     logger.debug("[SWEEP] [X] mbr2[" + axis + "] : minX,minY,maxX,maxY
:[" + mbr2.getMinX() + "," + mbr2.getMinY() + "," + mbr2.getMaxX() + "," +
mbr2.getMaxY());
57.    } else {
58.     mbr1.setDirectionY(MbrPlusNew.Direction.UP);
59.     mbr1.setBoundY(bound);
60.     mbr2.setDirectionY(MbrPlusNew.Direction.DOWN);
61.     mbr2.setBoundY(bound);
62.    }
63.    for (IRecordPlusNew iRecord: recordList) {
64.     if (mbr1.containsMBR2(iRecord.getMbrPlus())) {
65.      group1.addRecord(iRecord);
66.     }
67.     if (mbr2.containsMBR2(iRecord.getMbrPlus())) {
68.      group2.addRecord(iRecord);
69.     }
70.     if (mbr2.intersectionMBR2(iRecord.getMbrPlus())) {
71.      if (group2.getDescendant().stream().anyMatch(r -> {
72.        return r.equals(iRecord);
73.       })) {
74.       group2.addRecord(iRecord);
75.      }
76.     }
77.     if (mbr1.intersectionMBR2(iRecord.getMbrPlus())) {
78.      if (group1.getDescendant().stream().anyMatch(r -> {
```

```
79.         return r.equals(iRecord);
80.        })) {
81.        group1.addRecord(iRecord);
82.       }
83.      }
84.     }
85.     int tempCount = 0;
86.     for (IRecordPlusNew iRecord: recordList) {
87.      if (!mbr1.containsMBR2(iRecord.getMbrPlus()) &&
mbr1.intersectionMBR2(iRecord.getMbrPlus())) {
88.       tempCount++;
89.      }
90.     }
91.     String pom = "";
92.     for (IRecordPlusNew iRecord: group1.getDescendant()) {
93.      pom += iRecord.getMbrPlus().idMBR;
94.     }
95.     pom = "";
96.     for (IRecordPlusNew iRecord: group2.getDescendant()) {
97.      pom += iRecord.getMbrPlus().idMBR;
98.     }
99.     group.setCount(tempCount);
100.    group.setSubsetS1(mbr1);
101.    group.setSubsetS2(mbr2);
102.    group.setMaxValue(bound);
103.    group.setGroup1(group1);
104.    group.setGroup2(group2);
105.    saveService.SaveObject("sweepG1G2", group);
106.    return group;
107.   }
```

Na závěr do pomocných uzlů nastavíme jednotlivé pomocná hodnota ohraničující souřadnice, jako je například MBR a ohraničující hodnotu a vrátíme toto rozdělení jako skupinu dvou uzlů.

2.3.3 Metoda Partition

Tato metoda nám slouží pro vybrání osy splitu daného uzlu, který se má rozdělit. Seřadíme záznamy uzlu podle nejmenších hodnot pro danou osu. Na řádku 18 vyvoláme metodu Sweep pro osu x a na řádku 29 vyvoláme metodu Sweep pro osu y. Dále pak porovnáme jednotlivé heuristiky MBR a vybereme nejlepší dvojici.

```
1. private SeparateGroupsPlusNew partition(INodePlusNew node, int
   fillFactor) {
2.      List < INodePlusNew > listNode = new ArrayList < > ();
3.      SeparateGroupsPlusNew splitGroup = null;
4.      if (node.getDescendant().size() < 2 || node.getDescendant().size()
<= fillFactor) {
5.       return null;
6.      }
7.      List < IRecordPlusNew > recordList = new ArrayList < >
(node.getDescendant());
8.      Collections.sort(recordList, new Comparator < IRecordPlusNew > () {
9.       @Override
10.      public int compare(IRecordPlusNew o1, IRecordPlusNew o2) {
11.       if (o1.getMbrPlus().getMinX() > o2.getMbrPlus().getMinX()) {
12.        return 1;
13.       } else {
14.        return -1;
15.       }
16.      }
```

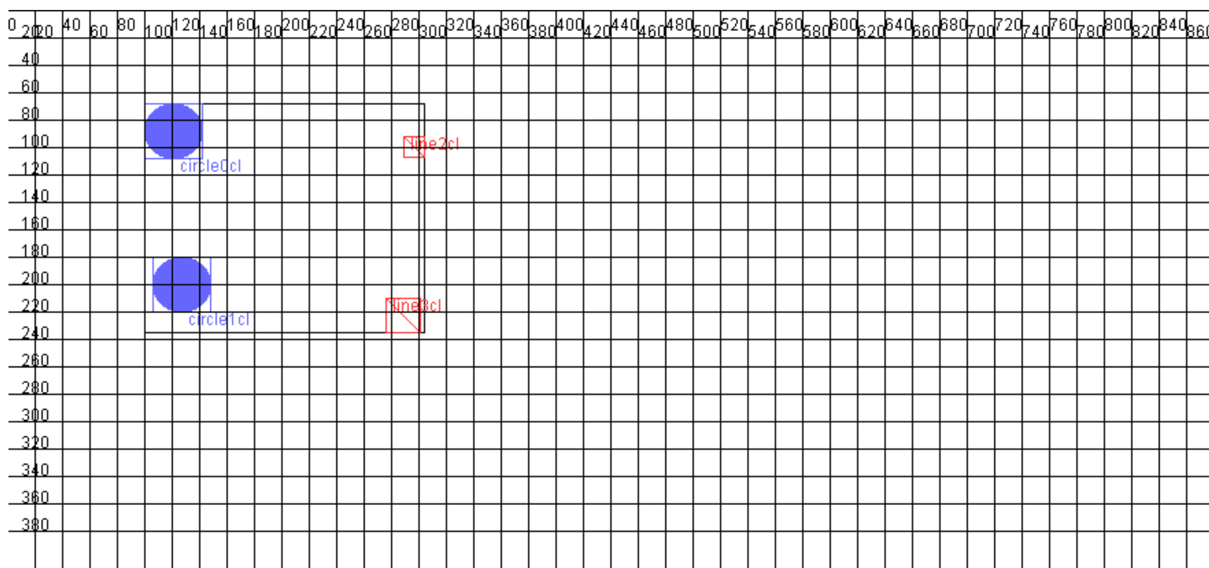
```
17.     });
18.     SeparateGroupsPlusNew groupX = sweep(Axis.AXISX, fillFactor,
recordList);
19.     Collections.sort(recordList, new Comparator < IRecordPlusNew > () {
20.      @Override
21.      public int compare(IRecordPlusNew o1, IRecordPlusNew o2) {
22.       if (o1.getMbrPlus().getMinY() > o2.getMbrPlus().getMinY()) {
23.        return 1;
24.       } else {
25.        return -1;
26.       }
27.      }
28.     });
29.     SeparateGroupsPlusNew groupY = sweep(Axis.AXISY, fillFactor,
recordList);
30.     // this is cost of sweep
31.     Axis temp = null;
32.     if (groupX.getCount() < groupY.getCount()) {
33.      splitGroup = groupX;
34.      temp = Axis.AXISX;
35.     } else if (groupX.getCount() == groupY.getCount()) {
36.      if (groupX.getArea() < groupY.getArea()) {
37.       splitGroup = groupX;
38.       temp = Axis.AXISX;
39.      } else {
40.       splitGroup = groupY;
41.       temp = Axis.AXISY;
42.      }
43.     } else {
44.      temp = Axis.AXISY;
45.      splitGroup = groupY;
46.     }
47.     saveService.SaveObject("partition" + temp.toString(), splitGroup);
48.     return splitGroup;
49.    }
```

Od řádku 32 až pořádek 45 porovnáváme skupiny podle osy x a y, vypočítáme počet rozdělení uzlů, které by muselo nastat, chtěli-li bychom použít jednotlivou dvojici. Při stejném počtu splitu vybíráme podle velikosti pokryté plochy.

3 DEMONSTRACE R-STROMŮ

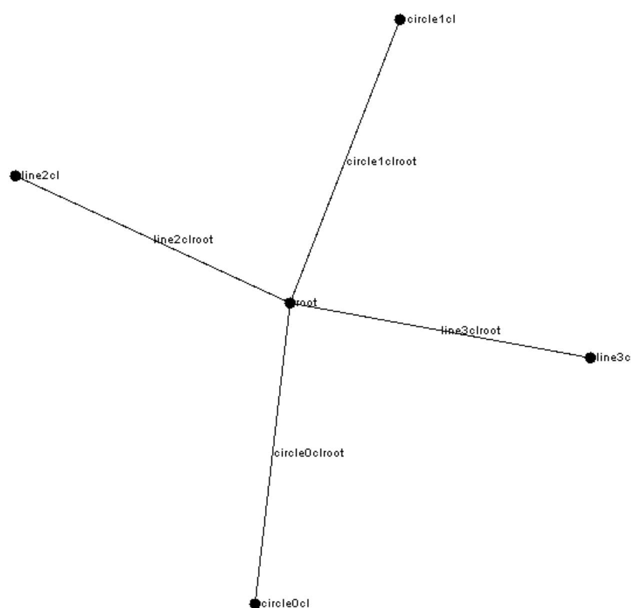
3.1 R-strom

Zde si ukážeme demonstraci vkládání prvků do datové struktury R-strom, která je založena na 2D datech. Parametry pro sestavení struktury jsou minimální počet záznamů m = 2, maximální počet záznamů M = 4. Na obrázku č. 21 máme první listový a zároveň kořenový uzel, který je nyní plný a kde v dalším kroku dojde k jeho rozlomení.



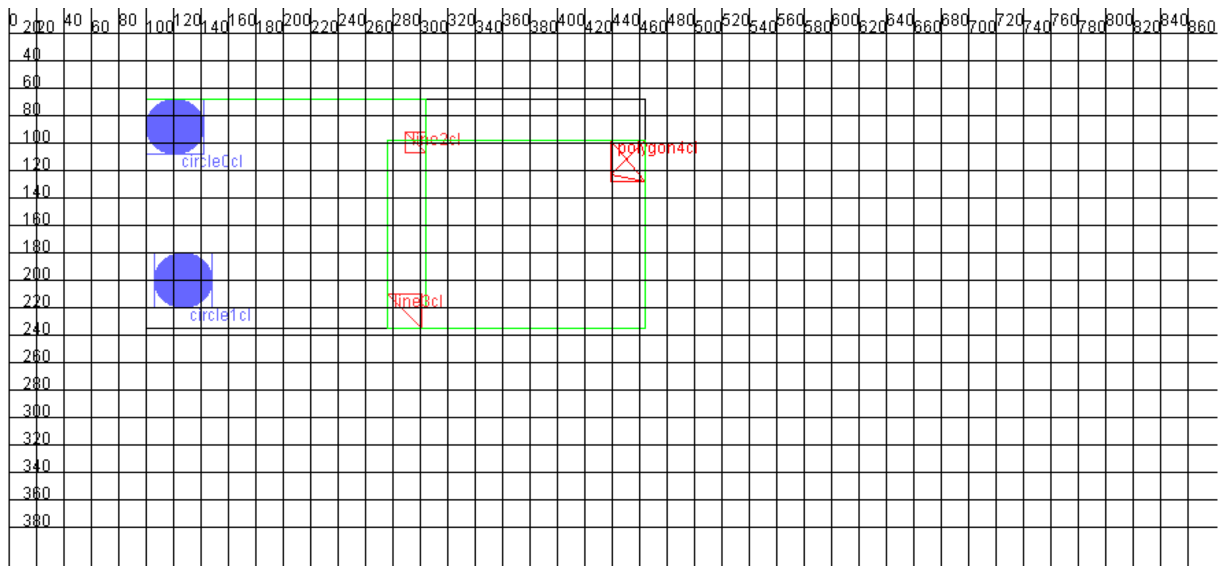
Obrázek 26 – Vkládání do prvního uzlu

Na obrázku č. 22 vidíme hierarchii kořenového uzlu root, který má ukazatele na své potomky.

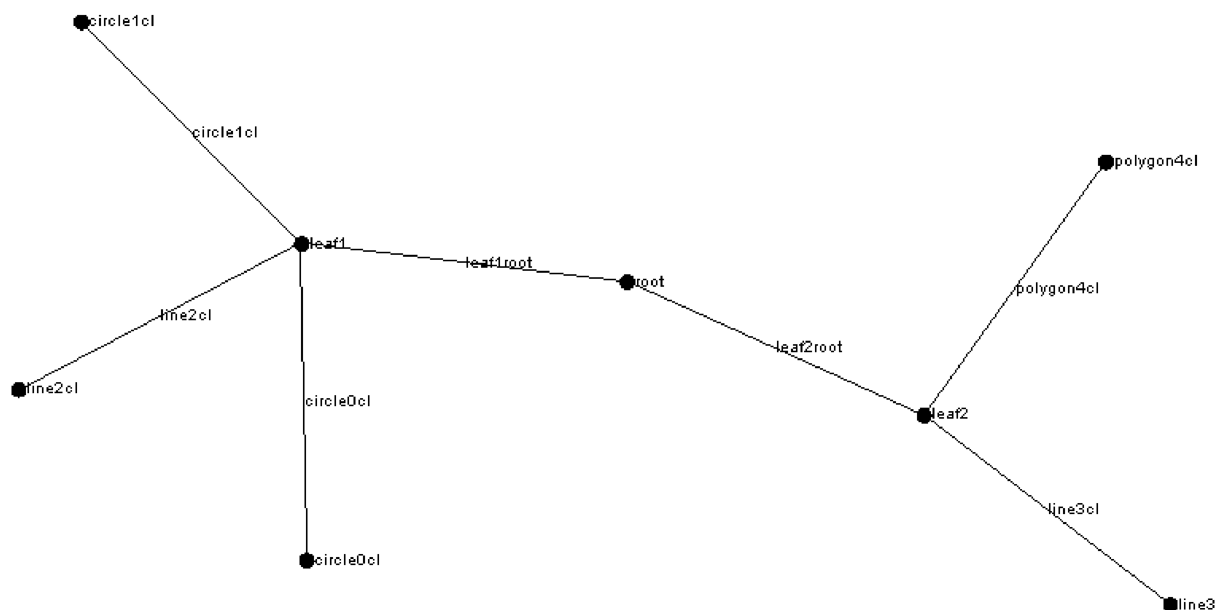


Obrázek 27 – Vkládání do prvního uzlu hierarchie záznamů

Na tomto obrázku vidíme, jak došlo k rozdělení root kořene na dva nové uzly obsahující všechny záznamy, kde se vytvořil listový uzel leaf1, který má 3 potomky, a leaf2 obsahující 2 potomky. Zde byl vkládaný objekt polygon4cl přidán do struktury a podle splitovacího algoritmu rozdělil dva nody, kde vybral dva nejvzdálenější body, tedy polygon4cl a circle0cl, a k nim metodou pickSeeds přiřadil zbylé záznamy.

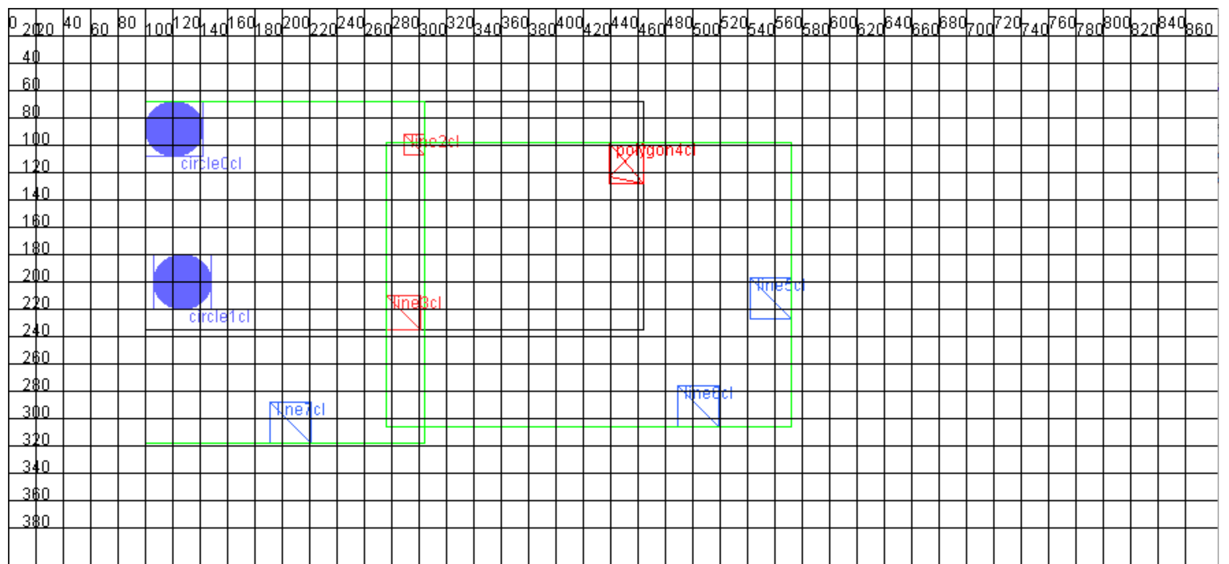


Obrázek 28 – Vkládání rozkladu kořenového uzlu

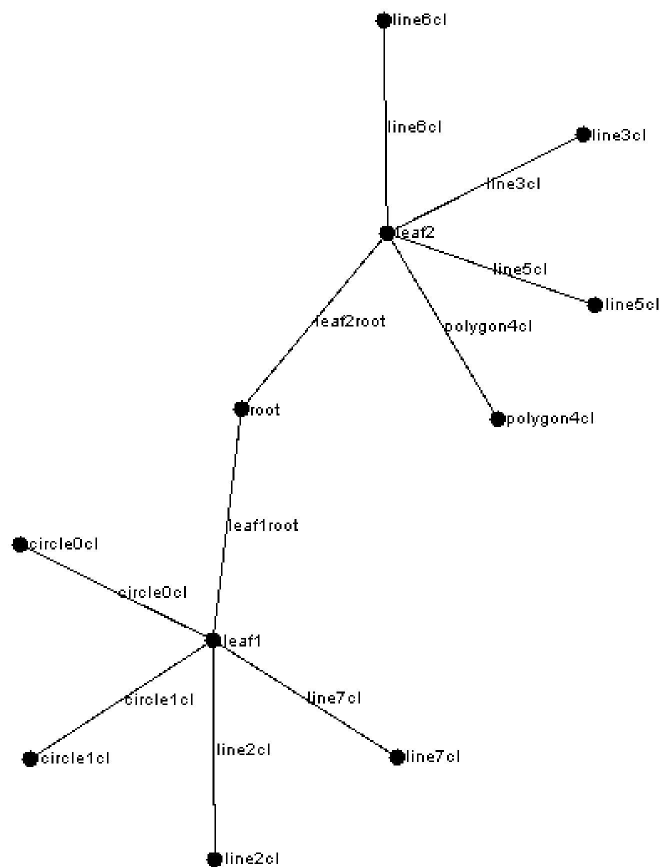


Obrázek 29 – Vkládání rozkladu kořenového uzlu – hierarchie

Na obrázku č. 25 vidíme naplnění 2 uzlů, kde při vložení dalšího záznamu dojde k rozdělení, a uvidíme rozdělení do 3 nových uzlů.



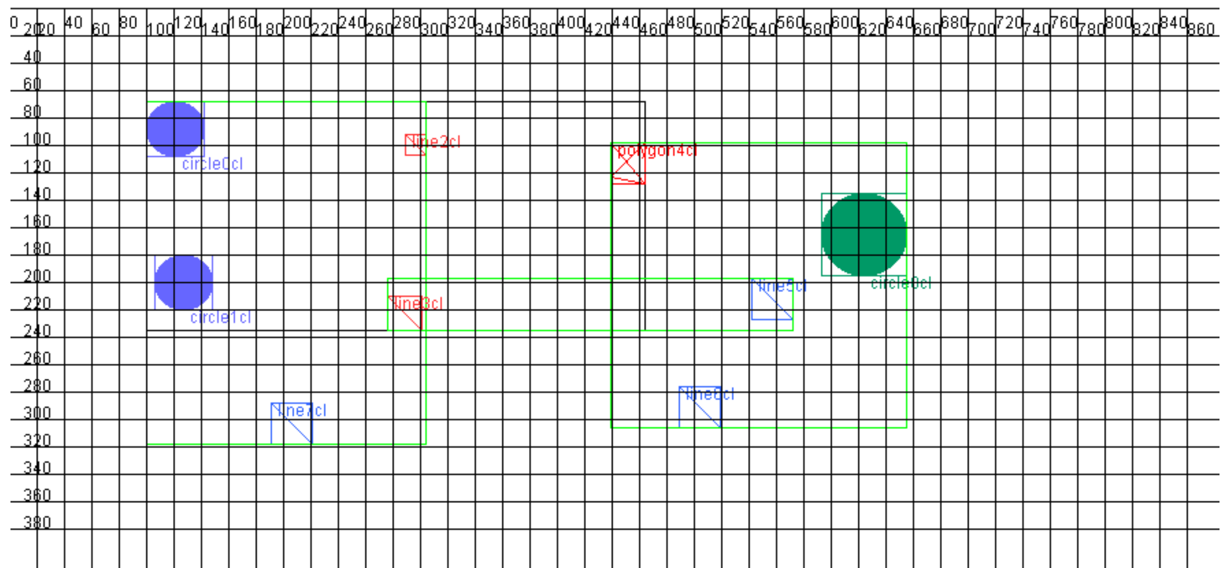
Obrázek 30 – Naplnění 2 uzlů



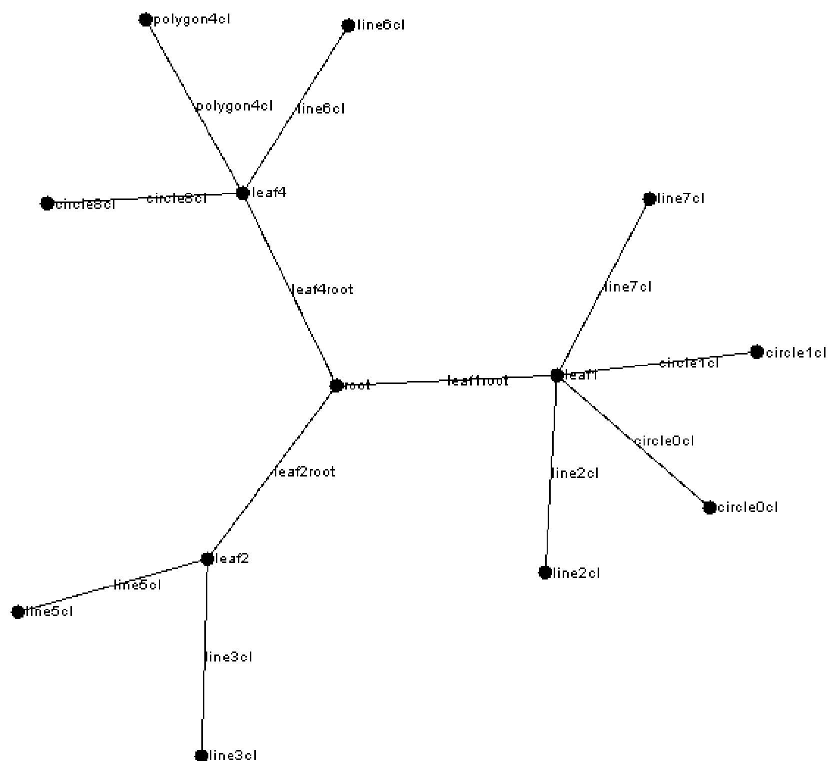
Obrázek 31 – Naplnění 2 uzlů – hierarchie

Po vložení dalšího záznamu do struktury došlo k přetečení uzlu, kde algoritmus vybral pro vložení nového záznamu leaf2, kam se pokusil nový záznam vložit. Zde došlo k přetečení maximální kapacity uzlu a byla vyvolána metoda split, kde podle algoritmu byla vybrána

nejhorší dvojice záznamů, což byl záznam line3cl a nově přidaný záznam circle8cl a poté k nim byly přiřazeny zbylé záznamy.



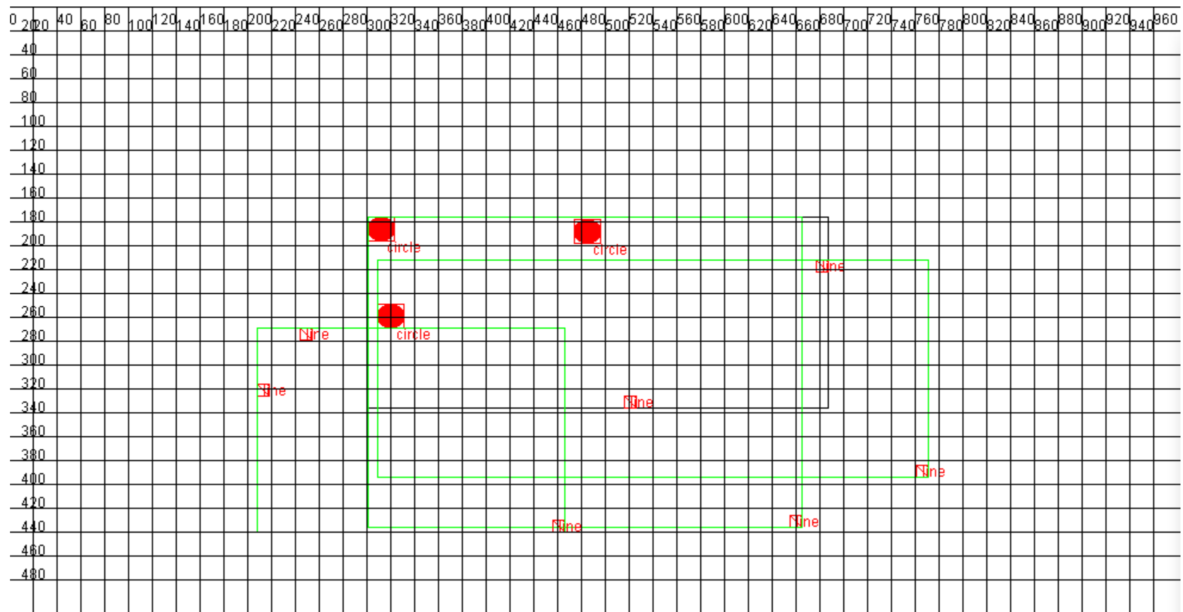
Obrázek 32 – Rozdělení do 3 nových uzlů



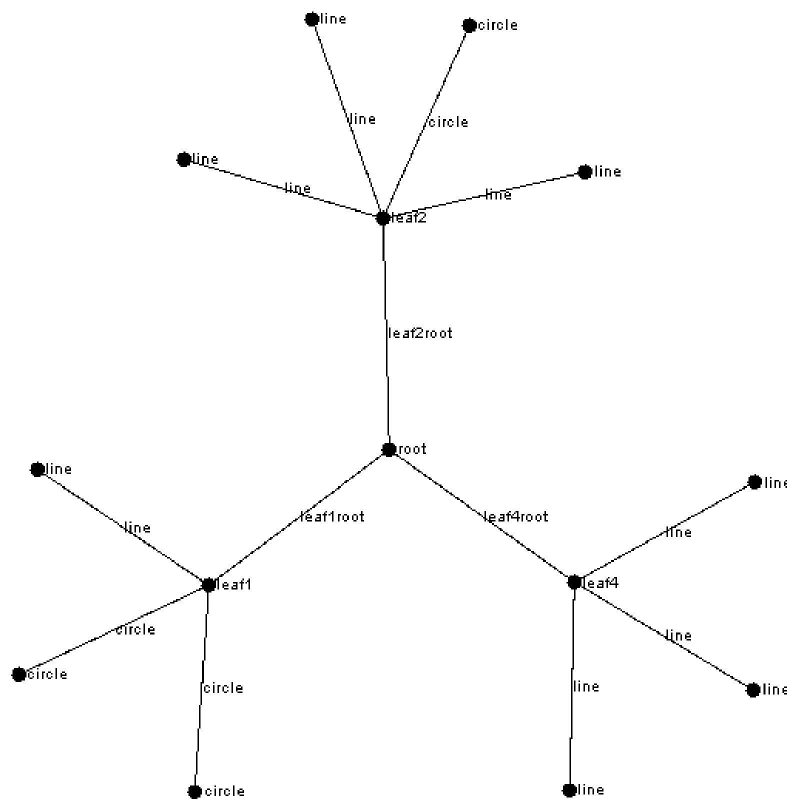
Obrázek 33 – Rozdělení do 3 nových uzlů – hierarchie

3.2 R*-strom

Zde si ukážeme demonstraci R*-stromu která je založena 2D datech. Parametry pro sestavení struktury jsou minimální počet záznamů m = 2, maximální počet záznamů M = 4. Vizualizace ukazuje rozdělení uzlů a jejich indexových záznamů.



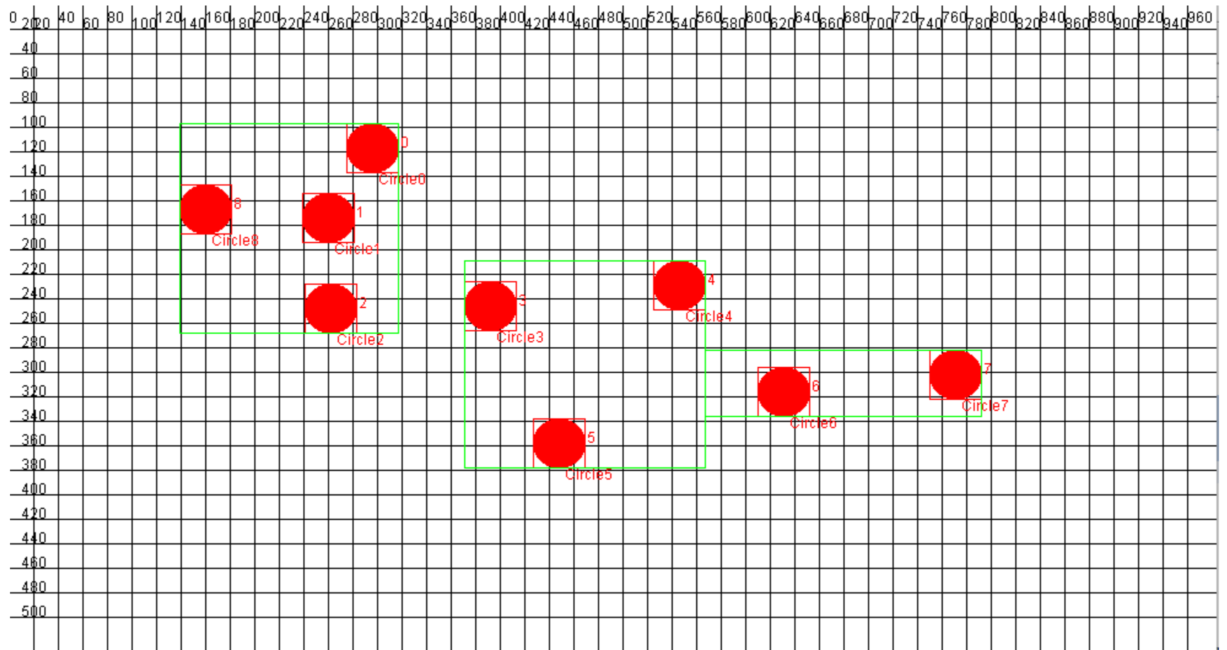
Obrázek 34 - Vizualizace R*-stromu



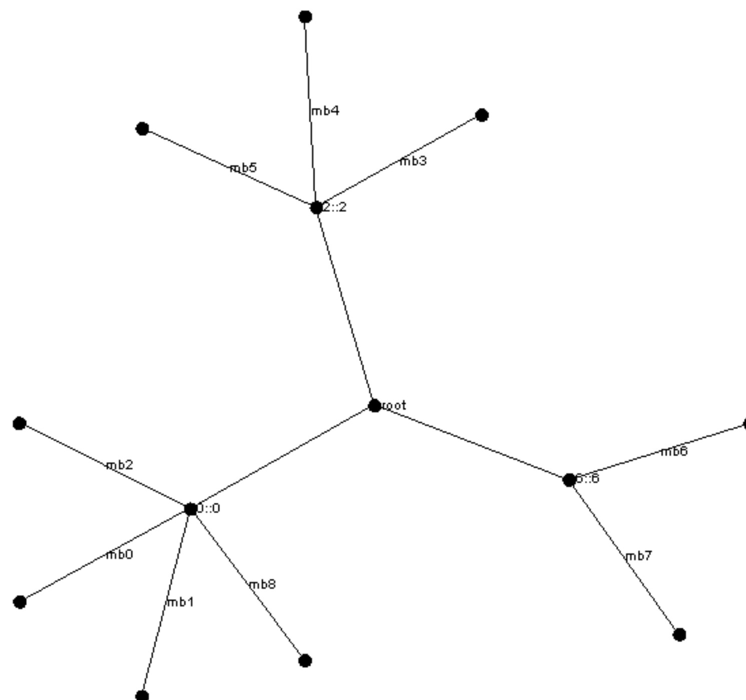
Obrázek 35 - Rozložení uzlů R*-stromu

3.3 R+-strom

Zde si ukážeme demonstraci R+-stromu která je založena 2D datech. Parametry pro sestavení struktury jsou minimální počet záznamů m = 2, maximální počet záznamů M = 4. Vizualizace ukazuje rozdělení uzlů a jejich indexových záznamů.



Obrázek 36 - Vizualizace R+-stromu



Obrázek 37 - Hiearchie uzlů v R+-stromu

4 ZÁVĚR

Cílem diplomové práce bylo provést seznámení s vybranými datovými strukturami typu R-strom a následnou implementací, kde primárním cílem je vytvoření softwarové aplikace pro výuku a demonstraci R-stromů.

Pro splněn cílů bylo potřeba implementovat vybrané datové struktury, kde byla popsána jejich základní charakteristika, popis vybraných implementačních metod a možnost vybudování struktur ze statistických dat, kde se použily prostor vyplňující křivky pro seřazení statistických dat a následné vložení do struktury. Implementace vybraných datových struktur je implementována v jazyce Java s pomocí build nástroje Maven.

Pro přiblížení datových struktur byly vytvořeny vzorové aplikace, které demonstrují chování vybraných datových struktur a možnost sestavení statistických dat pomocí vybraných prostor vyplňujících křivek.

5 POUŽITÁ LITERATURA

[1] Arge L, Mark Berg M, Haverkort H, and Ke Yi. 2008. *The priority R-tree: A practically efficient and worst-case optimal R-tree*. ACM Trans. Algorithms 4, 1, Article 9 (March 2008), 30 pages. DOI=http://dx.doi.org/10.1145/1328911.1328920

[2] Beckmann N, Kriegel Hans-Peter, Schneider R, and Seeger B. 1990. *The R*-tree: an efficient and robust access method for points and rectangles*. In Proceedings of the 1990 ACM SIGMOD international conference on Management of data (SIGMOD '90). ACM, New York, NY, USA, 322-331. DOI=http://dx.doi.org/10.1145/93597.98741

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

[4] Elashry, A & Shehab, Abdulaziz & Riad, Alaa el-din & Aboul-Fotouh, Ahmed. (2018). *2DPR-Tree: Two-Dimensional Priority R-Tree Algorithm for Spatial Partitioning in SpatialHadoop.* ISPRS International Journal of Geo-Information. 7. 179. 10.3390/ijgi7050179.

[5] GoodRich, M. T and Tamassia R. *Algorithm design: foundations, analysis, and Internet examples*. 2002. vyd. New York: Wiley, c2002, xii, 708 p. ISBN 04-713-8365-1.

[6] Guttman A, *R-Trees: A dynamic index structure for spatial searching.* in Proc. of ACM SIGMOD, June 1984, pp. 47–57.

[7] Chwedczuk M. Iterative algorithm for *drawing Hilbert curve. Programming is Magic*[online]. Polsko, Varšava: marcin-chwedczuk, 2016. Dostupné z: https://marcin-chwedczuk.github.io/iterative-algorithm-for-drawing-hilbert-curve

[8] Kamel I. and Faloutsos Ch. 1994. *Hilbert R-tree: An Improved R-tree using Fractals*. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94), Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo (Eds.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 500-509.

[9] Konečný, J. *Algoritmy a složitost 1: ALS1 – Přednáška 8 R-tree* [přednáška]. Olomouc: Katedra informatiky, Univerzita Palackého 2018. Dostupné z: http://phoenix.inf.upol.cz/~konecnyj

[10] Konečný, J. *Algoritmy a složitost 1: ALS1 – Přednáška 9 R+-tree a R*-strom* [přednáška]. Olomouc: Katedra informatiky, Univerzita Palackého 2018. Dostupné z: http://phoenix.inf.upol.cz/~konecnyj

[11] Lewis, H. R and Denenberg L. *Data structures*. 1997. vyd. New York, NY: HarperCollins Publishers, c1991, xv, 509 p. ISBN 06-733-9736-X.

[12] Manolopoulos, Y, A Nanopoulos, A. A. Papadopoulos a Y Theodoridis. *R-trees: Theory and Applications. Series in Advanced Information and Knowledge Processing.* 2005. Switzerland: Springer, 2005. ISBN 978-1-84628-293-5.

[13] Oracle. *Simple Example: Inserting, Indexing, and Querying Spatial Data. Spatial and Graph Developer's Guide* [online]. USA: Oracle, 2018. Dostupné z: https://docs.oracle.com/database/121/SPATL/simple-example-inserting-indexing-and-querying-spatial-data.htm#SPATL486

[14] Samet H. *Foundations of multidimensional and metric data structures*. San Francisco: Morgan Kaufmann, 2006, xxvii, 993 s. ISBN 978-012-3694-461.

[15] Sellis K. T, Roussopoulos N, and Faloutsos Ch. 1987. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87),* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507-518.

[16] SQLite. *The SQLite R*Tree Module*. [online]. North Carolina: SQLite, 1995. Dostupné z: https://sqlite.org/rtree.html

[17] Welch A. *Understanding Interleaved Sort Keys in Amazon Redshift, Part 1. Chartio home logo* [online]. San Francisco: Chorito, 2015. Dostupné z: https://blog.chartio.com/posts/understanding-interleaved-sort-keys-in-amazon-redshift-part-1

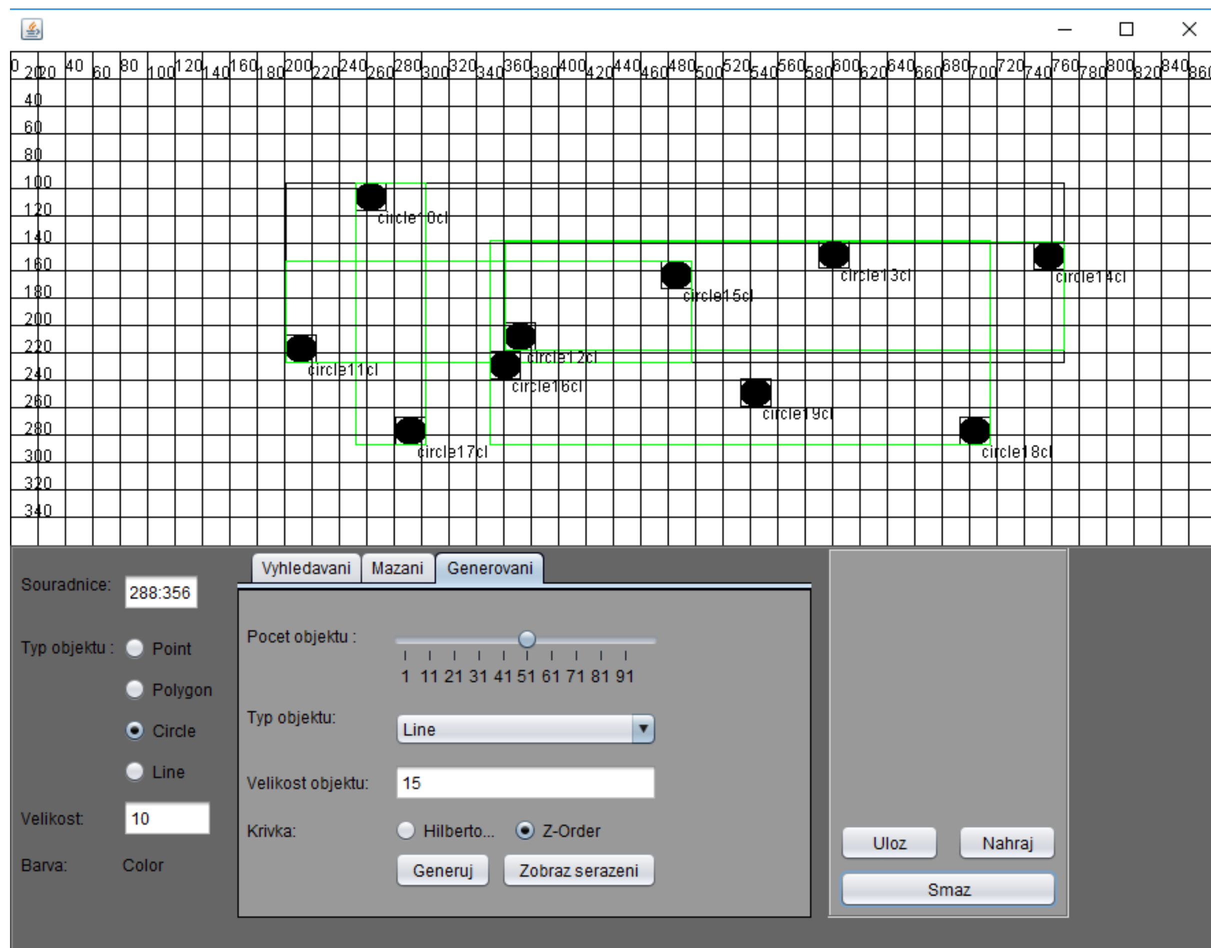
[18] Xu, Pan, Cuong Nguyen and Srikanta Tirthapura. *Onion Curve: A Space Filling Curve with Near-Optimal Clustering*. CoRR abs/1801.07399 (2018): n. pag.

PŘÍLOHA A – CD S VYTVOŘENOU APLIKACÍ

V příloze na CD se nachází funkční vzorové aplikace i jejich zdrojové kódy, které jsou napsané v IDE Netbeans verze 8.2 s build nástrojem maven.

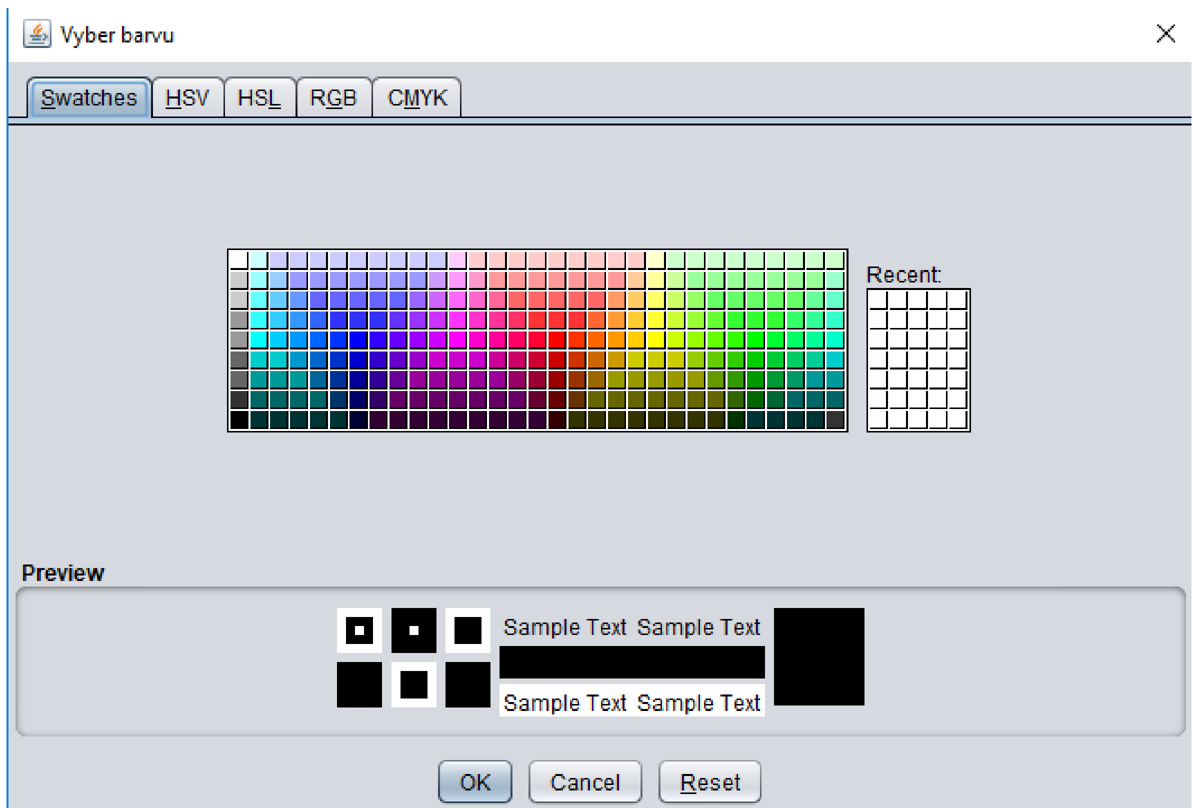
PŘÍLOHA B – UŽIVATELSKÁ DOKUMENTACE

Tato příloha popisuje ovládání aplikace, která slouží pro demonstraci a vizualizaci datových struktur.



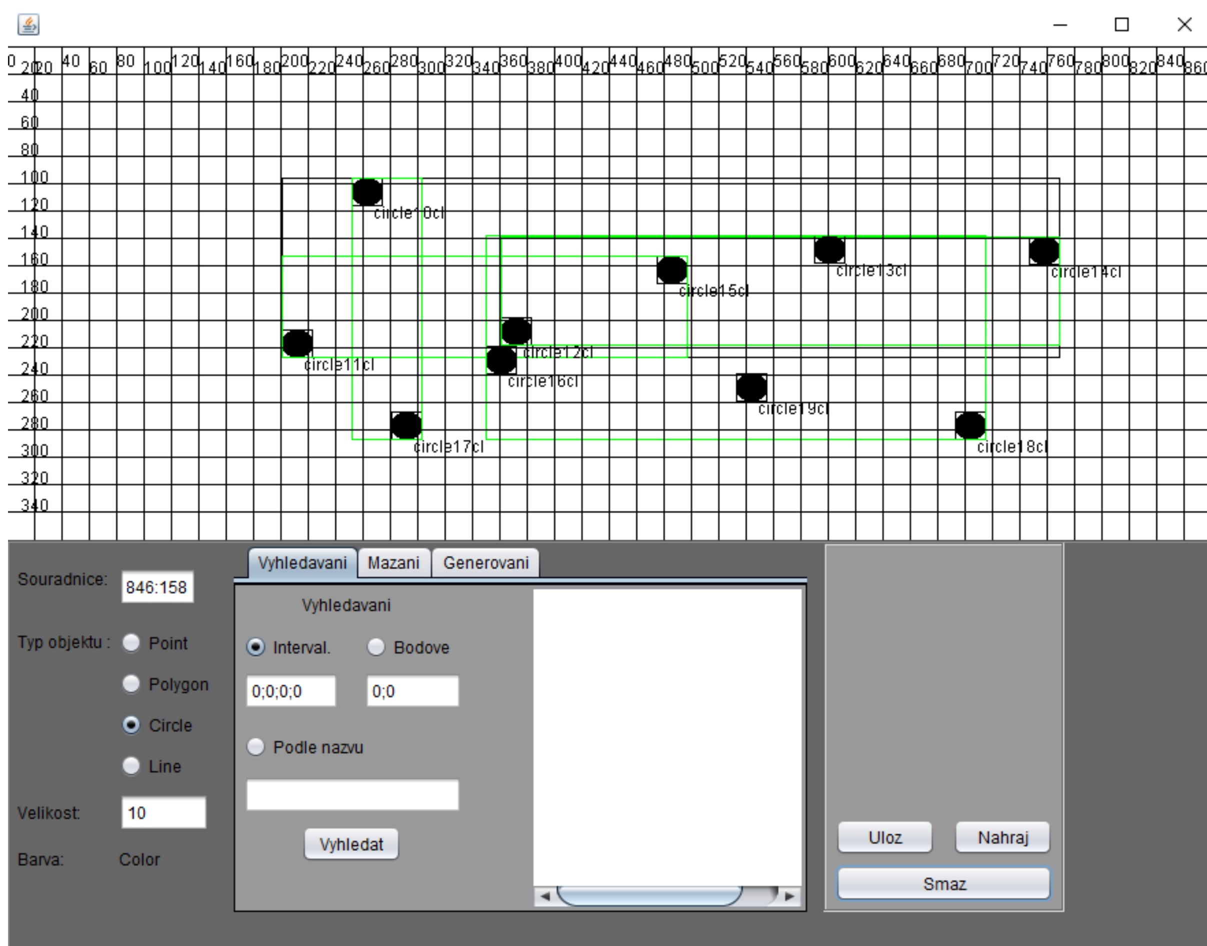
Obrázek 38 – Design aplikace generovaní

V levé části panelu se nachází souřadnice, které se mění při projíždění nad bílým plátnem. Dále je zde typ objektu, který slouží k vybrání toho, jaký chceme vkládat objekt na bílé plátno pomocí stisku levého tlačítka myši. Dále je zde velikost, jak daný objekt má být velký, a posledním případem je zde i barva. Při kliknutí na label color se nám zobrazí componenta pro vybrání barvy.



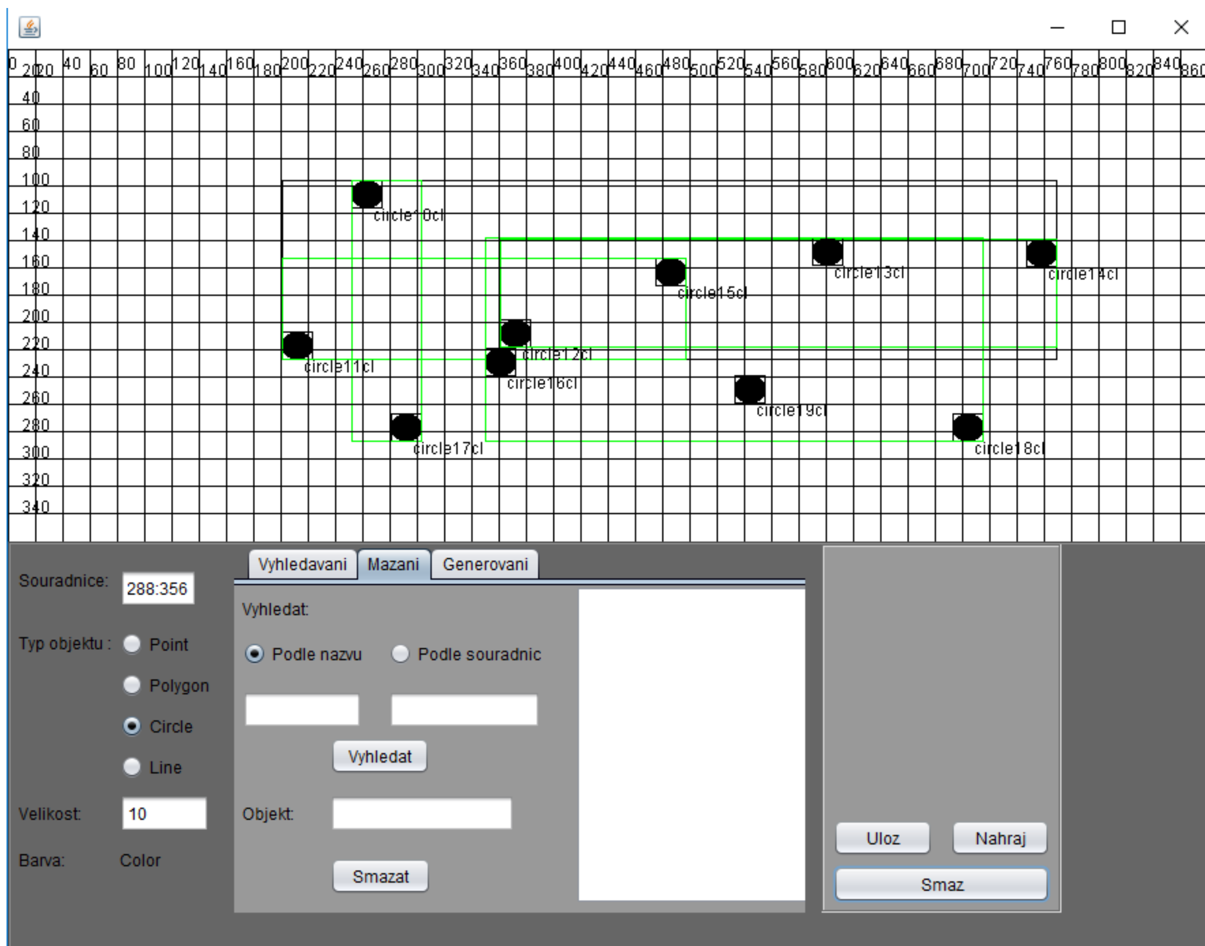
Obrázek 39 – Komponenta pro výběr barvy

Tlačítko Smaž slouží ke smazání stromu a jeho záznamů. Záložka generování slouží pro vygenerování statistických dat na bílém plátně. Nastavíme si, kolik chceme vygenerovat objektů (rozmezí 1–101), vybereme, jaký druh objektu chceme (např.: Line), nastavíme velikost, jak daný objekt má být velký a vybereme křivku, pomocí které seřadíme statistická data a vložíme je do datové struktury. Záložka vyhledávání objektů umožnuje vyhledat objekty podle intervalu, kde se vkládá bod ve formátu [minX;minY;maxX,maxY], při nedodržení formátu a správných hodnot nastane výjimka. Obdobně je to u bodového vyhledávání, kde se zadávají jenom souřadnice bodu [x,y] a v poslední řadě zde máme možnost vyhledávání podle id objektu.



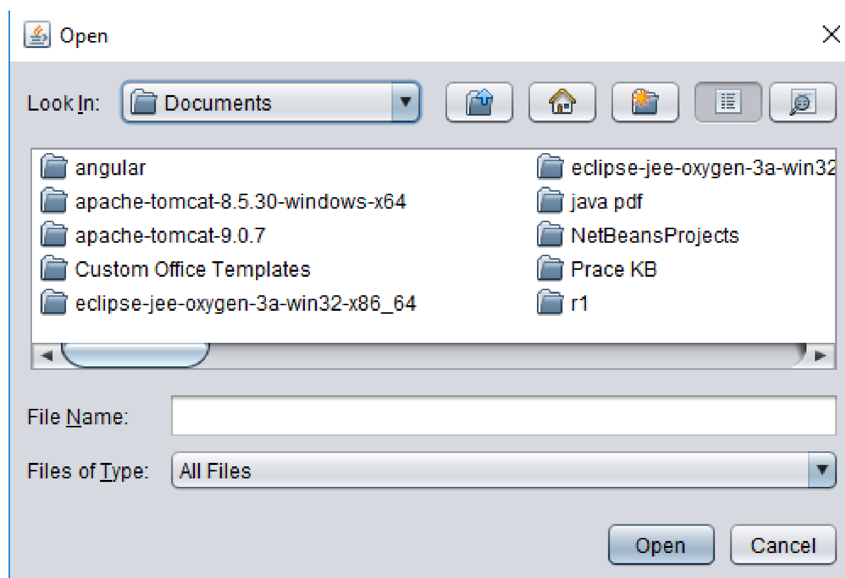
Obrázek 40 – Vyhledávání objektů

Při vyhledání objektů se dané objekty vypíší do textArea, která se nachází po pravé straně. Na posledním obrázku je ukázána záložka mazání kde si nejprve zvolíme jakým způsobem chceme daný objekt pro smazání nalézt buď si můžeme vybrat způsob zadání id záznamu (název záznamu), nebo podle souřadnice. Po zadání jedné z těchto dvou variant potvrdíme tlačítko vyhledat a v levé části v textArea se nám daný záznam zobrazí a ukáže se nám i v textFieldu. Pokud budeme opravdu tento záznam chtít smazat zmáčkneme tlačítko smazat.



Obrázek 41 - Mazání objektů

V pravé části se nachází panel s možností uložení souboru a jeho opětovné nahrání. Po jejich rozkliknutí se nám zobrazí panel, který se dotazuje jestli chceme daný objekt uložit nebo nahrát.



Obrázek 42 - Nahrání dokumentu

Kde nahráváme soubor s koncovkou .ser. Poslední tlačítko je tlačítko smaž, který smaže celou datovou strukturu.