

# Obsah

Úvod.....	- 2 -
1    Datové struktury.....	- 3 -
1.1    Abstraktní datové typy .....	- 3 -
1.2    Prvky datových struktur .....	- 3 -
1.3    Operace nad prvky datových struktur .....	- 3 -
2    Nelineární datové struktury - stromy.....	- 4 -
2.1    Binární stromy .....	- 4 -
2.2    Dokonale vyvážené stromy.....	- 5 -
2.3    AVL stromy .....	- 6 -
2.4    2-3-4 stromy .....	- 7 -
2.5    Red-Black stromy .....	- 7 -
2.6    B-stromy .....	- 9 -
2.6.1    B-strom .....	- 9 -
2.6.2    B+ strom .....	- 10 -
2.7    R-stromy .....	- 11 -
3    Implementace.....	- 14 -
3.1    Git .....	- 14 -
3.2    Rozhraní a prostředí .....	- 14 -
3.3    Implementace B+ stromu .....	- 15 -
3.4    Implementace R-stromu .....	- 17 -
4    Měření efektivity struktur.....	- 21 -
Použitá literatura.....	- 22 -

## Úvod

První kapitola práce má název Úvod. Slouží k zasazení řešené problematiky do širšího kontextu a v podobě stručného obsahu jednotlivých kapitol definuje strukturu písemné práce. Šablona obsahuje formátování, podle Závazných pokynů na stránkách FEI <http://www.fei.vsb.cz/cs/okruhy/studium-a-vyuka/informace-pokyny/pokyny-zpracovani-bp-dp>. Rozsah úvodu BP je doporučena cca půl strany A4, u DP cca 1 A4.

# 1 Datové struktury

Datová struktura je množina dat sloužící k jejich uchovávání a uspořádání. Pokud se velikost datové struktury může měnit, tak říkáme, že má dynamický charakter. Datové struktury mohou být lineární (pole, zásobník, fronta, seznam) či nelineární (stromy, grafy). Výběr datových struktur je pro vývoj programů často stěžejní. Výběr struktury ovlivňuje množství použité paměti, čas potřebný k operacím pro práci s daty, jakými jsou vkládání do datové struktury, vyhledávání dat či jejich odstranění. Různé datové struktury mají své specifické operace pro práci s daty. [1][3]

## 1.1 Abstraktní datové typy

Datové struktury, kterými se budeme zabývat, jsou abstraktní datové typy. Umožňují totiž sestavení programů s velkou mírou abstrakce. Jedná se o datové typy, ke kterým přistupujeme skrze jejich rozhraní, aniž bychom museli znát jejich konkrétní implementaci.[3]

## 1.2 Prvky datových struktur

Jednotlivé prvky datových struktur mohou být různé. Může se jednat o jednoduché typy (primitivní datové typy) či celé třídy s komplikovanou vnitřní strukturou.

Jednotlivé prvky lze od sebe na základě nějaké jejich vlastnosti rozlišovat porovnávat a tím pádem také uspořádat. [1]

## 1.3 Operace nad prvky datových struktur

Operace mohou být pro různé datové struktury různé. Obecně však lze operace rozdělit na 2 typy: dotazy a modifikující operace.

Dotazy vrací nějakou informaci o datové struktuře. Nejčastějšími dotazovacími operacemi jsou:

- DS.Search(k): vyhledávání prvku k v datové struktuře DS,
- DS.Min(): nalezení minimálního prvku v uspořádané datové struktuře DS,
- DS.Max(): nalezení maximálního prvku v uspořádané datové struktuře DS,

Modifikující operace mění datovou strukturu. Nejčastějšími modifikujícími operacemi jsou:

- DS.Insert(x): vložení prvku x do datové struktury DS,
- DS.Delete(x): odstranění prvku x z datové struktury DS.

## 2 Nelineární datové struktury - stromy

Strom je souvislý, acyklický, neorientovaný graf. Vrcholy takového grafu nazýváme uzly.

Kořenový strom je takový strom, který má jeden odlišný uzel. Tento uzel nazýváme kořen. Máme-li cestu mezi kořenem a libovolným jiným uzlem  $x$ , pak říkáme, že  $x$  je následovník kořene. Všechny uzly na této cestě od kořene k uzlu  $x$  jsou předchůdci uzlu  $x$ . Existuje vždy právě jedna cesta z jednoho uzlu ke každému jinému uzlu ve stromu.

Pokud mezi nějakým uzlem a kořenem není žádný jiný uzel, pak tento uzel nazýváme potomkem kořene. Každý uzel s výjimkou kořene má právě jeden uzel, který uzlu předchází. Takový uzel nazýváme rodičem.

Uzel, který nemá žádné potomky, nazýváme list, nebo listový uzel. Uzel s potomky je pak vnitřní uzel. Pro další uzly ve stromu používáme obdobné názvy jako v rodokmenu. Například rodiče, který je rodičem uzlu  $x$ , nazýváme prarodičem uzlu  $x$ . Ve stromu můžeme najít třeba i sourozence, což jsou uzly, které mají stejného rodiče a strýce. Strýc uzlu  $x$  je sourozenec rodiče uzlu  $x$ .

Pokud v každém uzlu musíme mít určitý počet potomků ve specifikovaném pořadí, pak tento strom nazýváme M-ární strom. Toto pravidlo však často neplatí pro kořen a vnější uzly - listy. [1][3]

### 2.1 Binární stromy

Binární strom je složen z uzlů majících dva potomky. Každý uzel s výjimkou kořene binárního stromu má právě jednoho rodiče. Pro každý uzel platí pravidla binárního stromu. Každý potomek je buď levým, nebo pravým podstromem binárního stromu. Kromě potomků pak každý uzel obsahuje nějaká data s výjimkou listů, které mohou být prázdné (pro dodržení podmínky, že každý uzel má právě 2 potomky).

#### 2.1.1 Binární vyhledávací strom

Binární vyhledávací strom je takový binární strom, který má potomky seříděny podle nějakého klíče. Toto seřídění je pak stejné pro všechny uzly v daném binárním vyhledávacím stromu. Levý potomek tedy bude vždy dle tohoto porovnání menší než pravý.

Většinou nevíme dopředu nic o klíčích binárního vyhledávacího stromu, dle kterých pak budou ve stromu přibývat nové uzly a které tedy budou strom formovat. Může se tedy stát, pokud budou přibývat uzly s klíči, které jsou seřazeny vzestupně (nový uzel se vždy zařadí jako nejpravější uzel stromu) či sestupně (nový uzel se vždy zařadí jako nejlevější uzel stromu). Takový strom pak degraduje na lineární seznam a k nalezení prvku je zde potřeba průměrně  $\frac{n}{2}$  porovnání. V nejhorším případě je potřeba až  $n$  porovnání.

Vyhledávání v Binárním vyhledávacím stromu probíhá zavoláním metody vyhledávání na kořen tohoto stromu. Pokud se klíč tohoto uzlu shoduje s hledaným, pak byl uzel úspěšně

nalezen. Pokud je vyhledávaný klíč větší nežli klíč v aktuálním uzlu a neexistuje pravý potomek, tak vyhledávání končí neúspěšně, což znamená, že uzel nalezen nebyl. Pokud však pravý potomek existuje, tak je rekurzivně volána metoda vyhledávání na tomto pravém potomkovi. Analogicky pokud hledaný klíč je menší jak klíč v aktuálním uzlu a neexistuje levý potomek, pak se hledaný klíč ve stromu nenachází. Pokud levý potomek existuje, tak je na něj rekurzivně volána metoda vyhledávání.

Vkládání do Binárního vyhledávacího stromu je obdobné jako vyhledávání. Vkládaný prvek je nejprve vyhledán, a pokud hledání končí úspěšně, tak je do stromu vkládán klíč, který se zde již nachází. Takový klíč nazýváme duplicitní klíč nebo duplicita. Pokud by vyhledávání skončilo neúspěšně (tedy potomek, na který by se volala metoda vyhledávání, neexistuje), tak se právě zde vytvoří nový uzel s klíčem, který je do stromu vkládán a který byl rovněž použit pro vyhledávání, jenž skončilo neúspěšně.

Jestliže je ve stromu povoleno vkládání duplicitních prvků, tak při vkládání je potřeba upravit podmínku pro rozhodování na kterou stranu se prvek umístí. Pokud je klíč vkládaného prvku shodný s klíčem aktuálního uzlu, tak bude vložen vždy jako pravý (případně vždy jako levý) potomek. Pokud již jeden neexistuje, tak bude metoda pro vkládání volána vždy na pravého (případně vždy levého) potomka. Při vyhledávání, pokud je dosaženo uzlu s klíčem stejným, jako je klíč vyhledávaného prvku, tak se bude pokračovat ve vyhledávání na pravém (případně levém, v závislosti na implementaci vkládání) potomkovi.

Rušení uzlu opět nejdříve zahrnuje jeho vyhledávání. Pokud prvek není nalezen, pak tato procedura končí. Pokud je prvek nalezen, pak další postup záleží na počtu potomků. Pokud rušený uzel nemá žádného potomka, pak jej lze odstranit bez jakékoli další akce. Pokud má uzel jednoho potomka, pak se rodič rušeného uzlu stane rodičem toho uzlu, který byl potomkem uzlu rušeného. Potomek rušeného uzlu se tak stane potomkem uzlu, který je rodičem od rušeného uzlu. Po vytvoření této vazby je možné rušený uzel odstranit. Jestliže má rušený uzel 2 potomky, pak máme dvě možnosti. Buď nahradíme rušený uzel nejpravějším uzlem levého podstromu, nebo nejlevějším uzlem pravého podstromu. Nejpravější (respektive nejlevější) uzel nalezneme rekurzivně tak, že projdeme levý (respektive pravý) podstrom rušeného uzlu a pokud existuje pravý (levý) potomek, tak je navštíven a rekurzivně se opět následuje pravým (levého) potomkem až do té doby, dokud takový potomek existuje. Poslední uzel, tedy ten, který již nemá pravého (levého) potomka je nejpravější (nejlevější) potomek.

## 2.2 Dokonale vyvážené stromy

Nejhorší případy pro sestavení binárního vyhledávacího stromu jsou takové, kdy vkládáme položky, které jsou již seřazené, mají velké množství duplicitních klíčů (pokud jsou samozřejmě ve stromu povoleny), jsou opačně seřazené, nebo alternují klíče s velkými a malými hodnotami. V takových případech strom degraduje a stává se i pro relativně malý počet položek vysokým. Vyhledávání v takovém stromu je tak pomalé v porovnání se stromem, který obsahuje stejný počet položek, ale má lepší strukturu - tedy není tak vysoký. [3]

Dokonale vyvážený strom je takový strom, který má počet uzlů v levém podstromu stejný jako v tom pravém, nebo se jejich počet liší maximálně o jeden. Toto pravidlo platí pro každý uzel takového stromu.

Dokonale vyvážené stromy jsou velmi výhodné pro vyhledávání, jelikož složitost vyhledávání se v nejhorším případě rovná délce nejdelší cesty ve stromu. Díky tomu, že dokonale vyvážený strom má pro všechny listy cestu stejně dlouhou, lišící se maximálně o 1, tak je tato cesta nejkratší v porovnání s jinými stromy, jejichž uzly mají stejný stupeň. Průměrný počet porovnání k nalezení uzlu v takovémto stromu je  $\log n$ .

Velkou nevýhodou dokonale vyvážených stromů je odebrání, nebo přidání nového uzlu. Tato akce je časově náročná, jelikož téměř vždy naruší dokonalou vyváženost stromu a vyžaduje tak jeho přestavění. Dokonale vyvážené stromy lze tedy použít v případě, že dopředu známe počet uzlu ve stromu a po jeho sestavení se pak jeho struktura již téměř nemění.

## 2.3 AVL stromy

Vyvážený strom je takový binární strom, u kterého se délka nejdelší cesty levého podstromu a délka nejdelší cesty pravého podstromu liší maximálně o 1. AVL stromy jsou vyvážené stromy a na rozdíl od dokonale vyvážených stromů není nutné při každém vkládání / rušení uzlu strom znovu konstruovat. Pokud však při vkládání strom přestane být vyvážený, pak je potřeba jej opět vyvážit.

Pro každý uzel si uchováváme informaci o jeho vyváženosti, a jestliže uzel není vyvážený, pak jej vyvážíme za pomoci rotace.

Rotace je operace, při níž dochází k výměně pozice rodiče a potomka takovým způsobem, aby byl strom opět vyvážený, a zároveň zachovává pravidlo binárního vyhledávacího stromu, že má rodič vlevo potomka s menším klíčem a vpravo potomka s klíčem větším než je klíč rodiče. Rotace rozlišujeme na jednoduché a dvojité. Jednoduchá rotace je buď pravá RR, nebo levá LL. Pravá rotace RR je operace, při níž se z rodiče stává levý potomek a současně z jeho původně pravého potomka se nově stává rodič. Levá rotace LL je pak operace, při níž se z levého potoka stává rodič a z rodiče pravý potomek. Dvojitě rotace rozlišujeme dvě a to LR a RL. Při LR rotaci je nejdříve provedena levá rotace s tím, že levý potomek původního pravého potomka (nynějšího rodiče) se stane pravým potomkem původně rodiče (nynějšího levého potomka) a následuje pravá rotace rodiče nového rodiče. RL rotace je opět nejdříve pravá rotace s přesunem pravého potomka od původního potomka (nynějšího rodiče) k původnímu rodiči (nynějšímu pravému potomku) jako jeho levý potomek. Následuje levá rotace rodiče od nového rodiče (původního levého potomka).

Při rušení uzlu, pokud je to potřeba a strom přestane být vyvážený, je opět potřeba strom vyvážit za pomoci zde popsaných rotací.

## 2.4 2-3-4 stromy

2-3-4 strom je takový strom, který obsahuje 3 typy uzlů. 2-uzel, 3-uzel a 4-uzel. Číslo uvedené u názvu jednotlivých uzlů říká, na kolik potomků daný uzel ukazuje. 2-uzel tedy ukazuje na 2 potomky, 3-uzel na 3 potomky a 4-uzel na 4. Každý z těchto uzlů má počet klíčů na kolik potomků ukazuje - 1, tedy 2-uzel je klasický uzel binárního stromu s 1 klíčem a dvěma potomky, kdežto 3-uzel obsahuje 2 klíče a 4-uzel 3.

Vkládání do takového stromu probíhá tak, že nalezneme pozici, kde by se měl uzel nacházet a pokud je zde 2-uzel, tak je do něj nový klíč přidán a stává se tak z něj 3-uzel. Obdobně je tomu u 3-uzlu, ze kterého se analogickým způsobem stane 4-uzel. V případě, že se na místě, kam chceme klíč vložit, nachází 4-uzel, pak je tento uzel rozdělen. Rozdělení uzlu je provedeno tak, že prostřední klíč je vložen do rodiče a z krajních klíčů se stávají potomci. Do jednoho z těchto potomků je pak nově vkládaný klíč vložen (v závislosti na porovnání s klíčem, který byl vložen do rodiče).

V případě, že by i rodič byl 4-uzlem, tak do něj opět nelze nový klíč vložit. Bylo by nutné provést štěpení - jak je popsáno v předchozím odstavci. Stejná situace by pak nastala i s jeho rodičem až, v nejhorším případě, s kořenem. Aby se tomuto zabránilo, tak vždy při vyhledávání místa pro vložení listu pokud se narazí na 4-uzel, tak je tento uzel rozdělen. Díky této operaci pak budou 4-uzly pouze v listech a máme jistotu, že při rozdělování 4-listu můžeme vložit prostřední hodnotu do rodiče, jelikož ten jistě 4-uzel není. Pokud by se kořen stal 4-uzlem, tak při jeho rozdělení se z prostředního klíče stane nový kořen, a ze 2 zbývajících pak jeho potomci.

Díky tomu, že strom "roste do výšky", tak se jeho výška zvětšuje pouze tehdy, když dělím kořen. 2-3-4 strom je tedy vždy dokonale vyvážený.

Vyhledávání, z důvodu vyskytujících se 3-uzlům a 4-uzlům, bude pomalejší než vyhledávání v binárním stromu.

## 2.5 Red-Black stromy

Algoritmus pro vkládání položek do 2-3-4 je snadný k pochopení, ale poněkud složitý na implementaci kvůli množství případů, které mohou nastat. Hlavní myšlenkou Red-Black stromů je tedy mít binární vyhledávací strom s výhodami 2-3-4 stromu.

Red-Black strom, někdy také červeno-černý strom, je částečně vyvážený binární strom s výškou  $2\log(n + 1)$ , kde  $n$  je počet uzlů daného stromu. Částečně vyvážený je proto, že každá cesta z libovolného uzlu do listu obsahuje vždy stejný počet černých uzlů. Počet takovýchto černých uzlů nazýváme černou výškou. Nejdelší cesta je tak vždy nejvýše tak dlouhá, jak dvojnásobek nejkratší cesty ze stejného uzlu.

Každý uzel Red-Black stromu je buď černý, nebo červený, což je zaznamenáno pomocí příznaku uvnitř každého uzlu. Každý list je černý a neobsahuje žádnou hodnotu. Je reprezentován hodnotou NULL. Jestliže je některý uzel červený, pak jsou jeho potomci vždy černí. Uzel je vždy černý.

Strom se díky tomuto příznaku stává velmi podobným 2-3-4 stromu. 2 červené potomky s černým rodičem si lze představit jako 4-uzel a 3-uzel je právě jeden červený potomek (levý nebo pravý) a černý rodič. [3]

Operace vkládání a rušení uzlů mají složitost  $O(\log n)$  a k vyváženosti jsou stejně jak u AVL stromů použity rotace. Počet těchto rotací je konstantní a tak se Red-Black stromy často využívají tam, kde se uzly rychle objevují a rychle mizí, nebo v aplikacích běžících v reálném čase.

V Red-Black stromech rozeznáváme 2 rotace - levou a pravou. Levou rotací rozumíme operaci, kdy se z rodiče stane potomek jeho původně pravého potomka rodič. Z levého potomka původního pravého potomka (nyní rodiče) se stane pravý potomek původního rodiče (nyní levého potomka). Pravou rotací pak rozumíme operaci inverzní k levé rotaci. Z rodiče se stane pravý potomek, z jeho levého potomka se stane nový rodič a z pravého potomka od tohoto původně levého potomka (nyní rodiče) se stane levý potomek původního rodiče (nynějšího pravého potomka).

Uzel, který vkládáme do tohoto stromu, je vždy červený. Vkládání probíhá na stejnou pozici jako bychom vkládali do binárního vyhledávacího stromu, tedy pokusíme se uzel vyhledat a v případě neúspěchu jej vložíme na místo, kde bychom jeho pozici předpokládali. Pokud je rodič černý uzel, pak jsou všechny podmínky pro Red-Black strom splněny. Pokud ne, pak může nastat několik dalších případů. Jestliže je rodič červený, tak dochází k porušení podmínky, že každý červený uzel musí mít 2 černé potomky. Důležitá je barva "strýce" vkládaného uzlu - tedy sourozence od červeného rodiče. Pokud je tento strýc červený, pak jen zaměníme barvu rodiče, strýce a prarodiče od vkládaného uzlu. Opět může být narušeno některé z pravidel Red-Black stromu, ale to o 2 úrovně výše, což je opět nutné opravit. Pokud však strýc není červený, ale černý, pak závisí, jestli vkládám nový prvek jako levého, nebo jako pravého potomka. Pokud je vkládán uzel jako levý potomek, pak je nutné provést pravou rotaci a obarvit původního rodiče a prarodiče (nynějšího rodiče a bratra) vkládaného uzlu. Jestliže však je strýc vkládaného uzlu černý a vkládá se nový uzel jako pravý potomek červeného uzlu, pak se provede levá rotace a původní rodič bude nyní jako nově vkládaný uzel. Tím pádem se nám problém mění na předchozí. Červený rodič, černý strýc a vlevo vložený nový prvek.

Pokud rušíme uzel, tak postupujeme nejdříve stejně jako u binárního vyhledávacího stromu. Pokud jsme smazali červený uzel, tak pravidla Red-Black stromu zůstala zachována. Pokud však rušíme uzel, který je černý a má černého potomka, tak jakmile potomek nahradí smazaného rodiče, tak jej označíme jako "dvojnásobně černý uzel". Snažíme se najít nejbližší červený uzel a dvojici červený - dvojnásobně černý uzel nahradit dvěma černými uzly. Máme 2 způsoby jak toho docílit: restrukturalizace a přebarvení. Restrukturalizace řeší problém lokálně, přebarvení šíří problém vzhůru. Pokud bratr dvojnásobně černého uzlu má červeného potomka, pak udělám restrukturalizaci. Pokud je tento potomek pravý, pak uděláme levou rotaci a přebarvíme dvojnásobně černý uzel na černý a červený uzel na černý. Pokud je onen potomek levý, tak provedeme pravou a levou rotaci a opět přebarvení oné dvojice uzlů. Pokud je bratr černý a má černého potomka, pak je třeba provést přebarvení bratra na červený uzel. Jestliže rodič



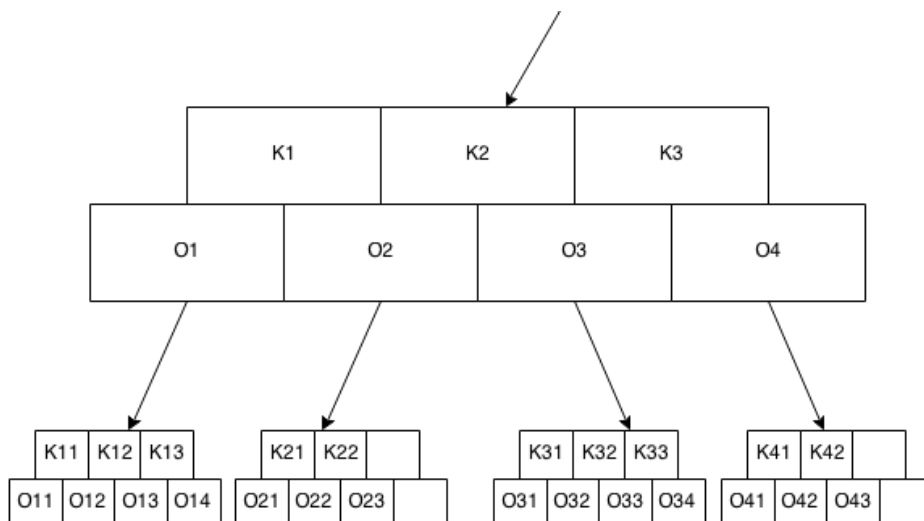
byl červený, tak je odebrání uzlu hotovo. Pokud však byl otec černý, tak se z něj stává dvojnásobně černý uzel a je potřeba na něj aplikovat některé pravidlo pro opravení tohoto jevu. Pokud je pravý bratr červený, pak je třeba rodiče obarvit na červenou, červeného bratra na černou a provést restrukturalizaci pomocí levé rotace. Problém je tak oddálen o 1 krok dále od kořene.

## 2.6 B-stromy

### 2.6.1 B-strom

Uvažujeme B-strom, který má řád  $M$ . Místo toho, abychom trvali na tom, že v každém uzlu musí být právě  $M$  klíčů, tak budeme trvat na tom, že každý uzel musí mít klíčů nejvýše  $M$  a nejméně pak  $M/2$ . Pokud se jedná o listovou stránku, pak stránka nemá žádné odkazy na potomky (podstromy, stránky). V opačném případě má stránka vždy  $k+1$  odkazů, kde  $k$  je počet klíčů dané stránky. Výjimkou v počtu klíčů je pochopitelně kořen, který musí mít alespoň jeden klíč. Pokud není zároveň listem, tak 2 odkazy na potomky a nejvýše pak také  $M$  klíčů s  $M+1$  odkazy. Takovéto uzly nazýváme stránky a při dodržení výše popsaných pravidel pro počet klíčů v každé stránce víme, že všechny nekořenové stránky v B-stromu jsou nejméně z poloviny zaplněny. Strom je buď prázdný, nebo obsahuje stránky - stromy, které reprezentují interval mezi klíči, mezi kterými je odkaz umístěn. Intervalem stromu zde rozumíme interval od nejmenšího klíče stromu (nejlevějšího klíče) po klíč stromu s nejvyšší hodnotou (nejpravější klíč). Pokud se jedná o první odkaz, pak celý interval je menší než první klíč v aktuální stránce, pokud se jedná o nejpravější odkaz, pak odkazuje na strom, jehož celý interval je větší než nejpravější klíč aktuální stránky. Pokud je odkaz mezi dvěma klíči, pak je nejmenší hodnota intervalu stromu, na nějž odkaz odkazuje, větší než levý klíč a zároveň nejvyšší hodnota intervalu odkazovaného stromu je nižší, než hodnota pravého klíče. [3]

Na obrázku 1.1 je znázorněna část B-stromu, kde  $M$  (maximální počet klíčů ve stromu) je 3. Každá stránka může mít tedy nejvýše 3 klíče se 4 odkazy na podstromy a nejméně 2 klíče se 3 odkazy. Ve vyobrazené části B-stromu je nejmenší klíč  $K_{11}$  a největší  $K_{42}$ . Klíče jsou uspořádány tak, že  $K_{11} < K_{12} < K_{13} < K_1 < K_{21} < K_{22} < K_2 < K_{31} < K_{32} < K_{33} < K_3 < K_{41} < K_{42}$ . Interval tohoto stromu je tedy  $K_{11} - K_{42}$ . Nejvyšší hodnota podstromu, na který odkazuje odkaz  $O_{11}$  by pak byla menší než  $K_{11}$  a nejmenší hodnota podstromu, který by byl odkazován odkazem  $O_{43}$  by byla větší než  $K_{42}$ .



Obrázek 1.1: Příklad B-stromu 3. řádu

Strom roste logaritmičticky a to vždy štěpením kořene. Všechny uzly tedy vždy leží ve stejné hloubce.

Vyhledávání ve stromu probíhá tak, že začínáme od kořenové stránky a procházíme její klíče. Pokud je hodnota stejná, pak jsme našli, co jsme hledali. Pokud je hodnota klíče větší, tak rekurzivně pokračujeme levým ukazatelem na další stránku. Pokud není žádný klíč větší než hledaný, tak stejným způsobem pokračujeme rekurzivním prohledáváním stránky pod nejpravějším odkazem. Jestliže chceme pokračovat v prohledávání stránky tam, kde žádný odkaz není, tak se ve stromu hledaný prvek nenachází.

Vkládání do stromu začíná opět, jako v každém jiném stromu, vyhledáváním. Jestliže prvek nalezneme, tak další akce bude spočívat v tom, jestli jsou ve stromu povoleny duplicity či nikoliv. Pokud narazím na odkaz, kterým nelze pokračovat v prohledávání stromu (žádný odkaz zde není) a nacházím se tedy v listové stránce, tak zde nový prvek přidám. Jestliže stránka není ještě zcela zaplněna, tak pouze přidám prvek na příslušnou pozici ve stránce tak, aby byly klíče ve stránce vzestupně seřazeny. Pokud je stránka již zcela zaplněna, pak je potřeba stránku rozštěpit - rozdělit na 2 stránky s polovičním zaplněním. Rozštěpení se provede tak, že z prvků stránky, do které chceme nový prvek vkládat a prvku vkládaného vybereme ten prostřední a vložíme jej do rodiče. Levý odkaz pak bude odkazovat na novou stránku s polovinou klíčů, které jsou menší než klíč, který byl vložen do rodiče a pravý odkaz pak bude odkazovat na stránku s polovinou klíčů, které jsou větší, než prvek vložený do rodiče. Pokud rodič nebyl zcela zaplněn, pak není třeba další akce. Pokud však již rodič obsahoval maximální množství klíčů, tak je potřeba rodiče opět rozštěpit. Tímto způsobem se v nejhorším případě rozštěpí všechny stránky na cestě ke kořeni a pak i samotný kořen. Pokud se rozštěpí kořen, tak se výška stromu zvětší o 1.

## 2.6.2 B+ strom

B+ strom vychází z B-stromu, ale liší se v umístění prvků ve struktuře. Zatímco B-strom má klíče s příslušnými daty rozmístěny ve všech stránkách stromu, B+ strom má všechny klíče s

daty uloženy v listech. Rozdíl v těchto dvou stromech tedy spočívá ve štěpení zaplněných uzlů. U B+ stromu místo toho, aby se prostřední prvek přesunul do rodiče, tak se do rodiče vloží pouze klíč a prvek s prostředním klíčem zůstane vždy v levém podstromu uložen jako nejpravější klíč. Vyhledávání se pak bude lišit tak, že pokud v prohledávané stránce bude aktuální klíč ne větší než hledaný, ale když bude větší nebo roven, tak se přemístím do uzlu pod levým odkazem. Až tehdy, když odkaz na další stránku neexistuje a klíč aktuální je roven klíči vyhledávanému, tak je nalezeno.

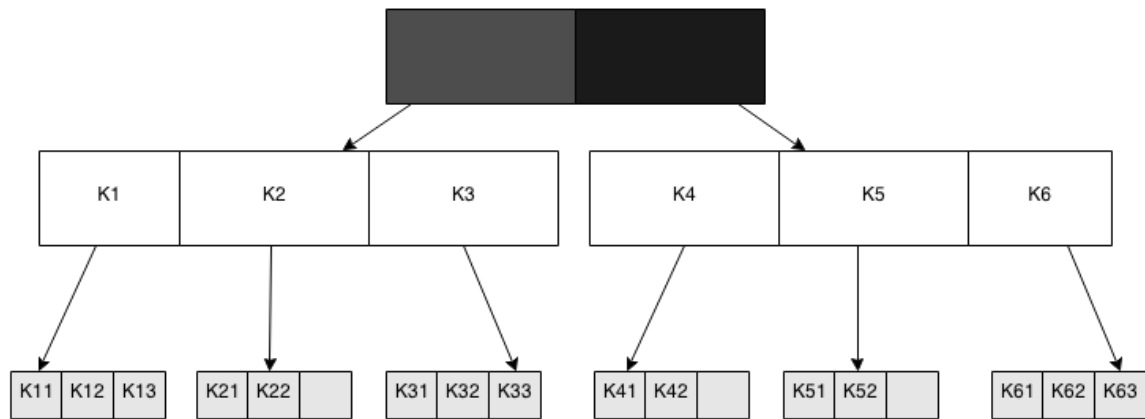
Výhodou B+ stromu je možnost rychlého procházení všech prvků, nebo libovolného počtu prvků sousedících. Tato výhoda je způsobena právě vlastností, že jsou všechny prvky uloženy v listových stránkách a vlastností, že všechny listové stránky leží ve stejné hloubce stromu. Nejpravější odkaz listové stránky je často implementován tak, že odkazuje na sousední listovou stránku. Ostatní odkazy stejně jak u B-stromu mohou být buď prázdné, nebo mohou odkazovat na data reprezentovaná klíči ve stromové struktuře.

## 2.7 R-stromy

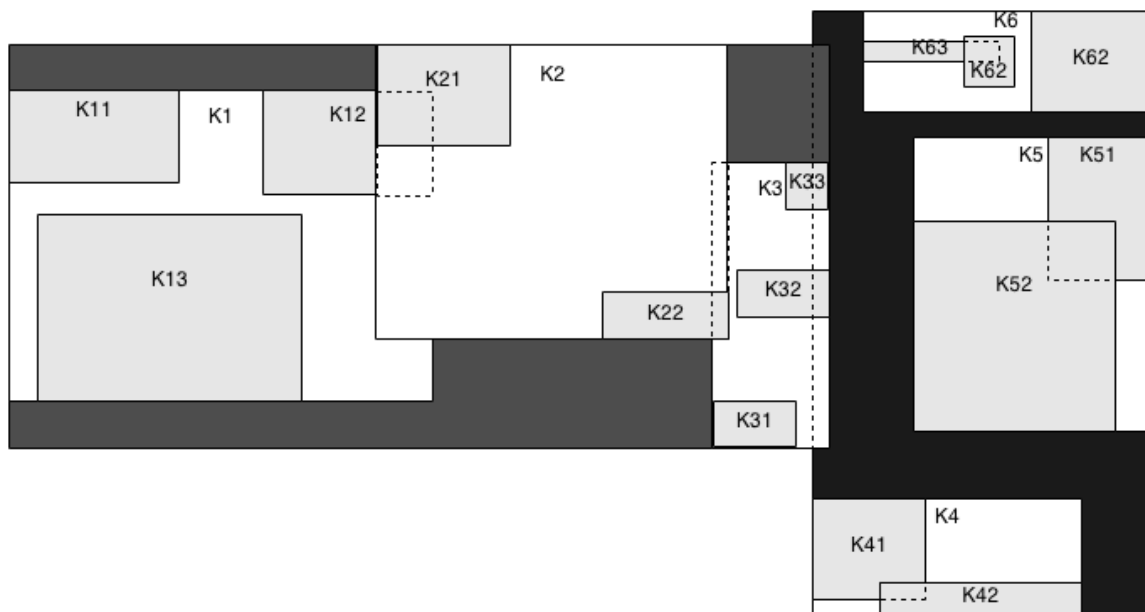
R-strom je prostorová datová struktura založená na B+ stromech. Tato struktura byla navržena především pro aplikace s geografickými daty a databázové systémy využívající indexovací mechanismus, jenž pomáhá získávat data rychleji, a to díky jejich prostorové lokaci. Při vyhledávání se totiž neprochází všechny objekty, ale pouze ty oblasti, ve kterých se může hledaný objekt nacházet. Tyto oblasti se zužují až do hledaného objektu. Vyhledávání bodu je stejně snadné jako vyhledávání úsečky či nějaké N-dimenzionální oblasti.

Jednotlivé uzly jsou množiny d-dimenzionálních objektů, kde d je menší nebo rovno N. Takovéto objekty nazýváme MBR (minimum bounding rectangles = minimální ohraničující obdélníky). Každý vnitřní uzel má pak MBR ohraničující MBR jeho potomků. Listové uzly pak, stejně jako v B+ stromech, ukazují na data. MBR v listových uzlech jsou pak pravoúhlé objekty ohraničující data. Jednotlivé MBR se mohou překrývat, mohou být obsaženy ve více jiných MBR, ale mají vždy jen jednoho rodiče. Pro 1-dimenzionální objekty to jsou úsečky, pro 2-dimenzionální obdélníky, 3-dimenzionální kvádry, ... Počet klíčů v jednotlivých uzlech je stejný jako počet klíčů ve stránkách B+ stromu. Rozdíl je však v kořeni, který musí obsahovat, pokud není listem, alespoň 2 záznamy.

Na obrázku 1.2 je ukázka stromové struktury R-stromu a na obrázku 1.3 jsou pak ukázána data z tohoto R-stromu a také to, jak se jednotlivé oblasti (MBR) mohou překrývat.



Obrázek 1.2: Ukázka stromové struktury R-stromu

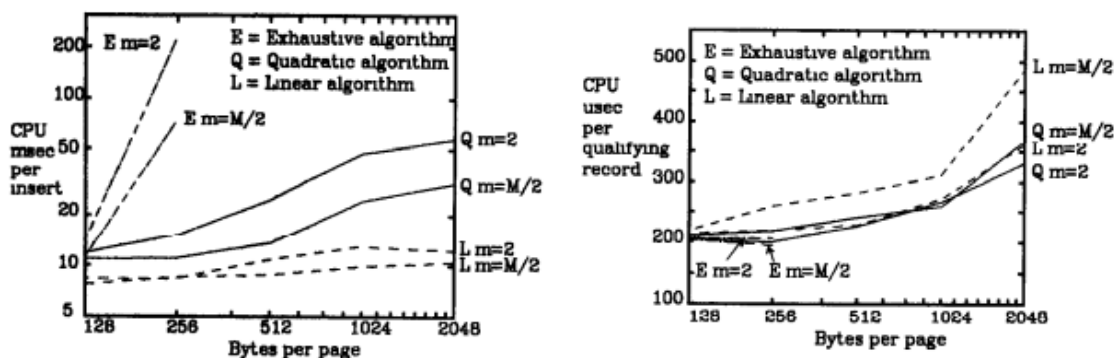


Obrázek 1.3: Ukázka dat R-stromu z obrázku 1.2

Vyhledávání v R-stromu je podobné vyhledávání v B+-stromu. I zde jsou všechna data uložena až v listech. Vyhledávání opět začíná v kořeni a prohledávají se rekurzivně potomci, kteří obsahují oblast, ve které se nachází to, co právě hledáme. Toto je jiné jako v B+ stromech, jelikož zde je možné pokračovat po více větvích, což vyplývá z vlastnosti, že se jednotlivé MBR mohou překrývat. Jestliže jsou takoví potomci nalezeni, tak se prohledává opět v jejich potomcích, až dokud se nedojde k listu. Pokud takoví potomci nalezeni nejsou, pak vyhledávání skončí neúspěšně. Pokud v listu existuje MBR s daty která jsou vyhledávána, tak pouze tehdy vyhledávání končí úspěšně.

Vkládání do R-stromu začíná vyhledáváním. Pokud vyhledávání končí úspěšně, pak buď vložíme duplicitní prvek, vložíme ke stávajícímu nějaká další data, nebo vkládání ukončíme. V případě, že vkládáme duplicitní prvek nebo prvek ve stromu nalezen nebyl, tak prvek můžeme do

listu vložit. Prvek vkládáme do stromu vždy tak, aby se v něm MBR co nejméně změnily. Při vkládání postupujeme podobně jako u B+ stromu - pokud je v listu místo, tak pouze prvek vložíme a následně, pokud je třeba, upravíme MBR listu, rodiče a všech uzlů na cestě ke kořeni. Pokud místo v listu není, pak dojde k jeho rozštěpení a vytvoření nového prvku MBR v rodiči. Opět pak dochází k úpravě všech MBR až ke kořeni, když je to třeba.



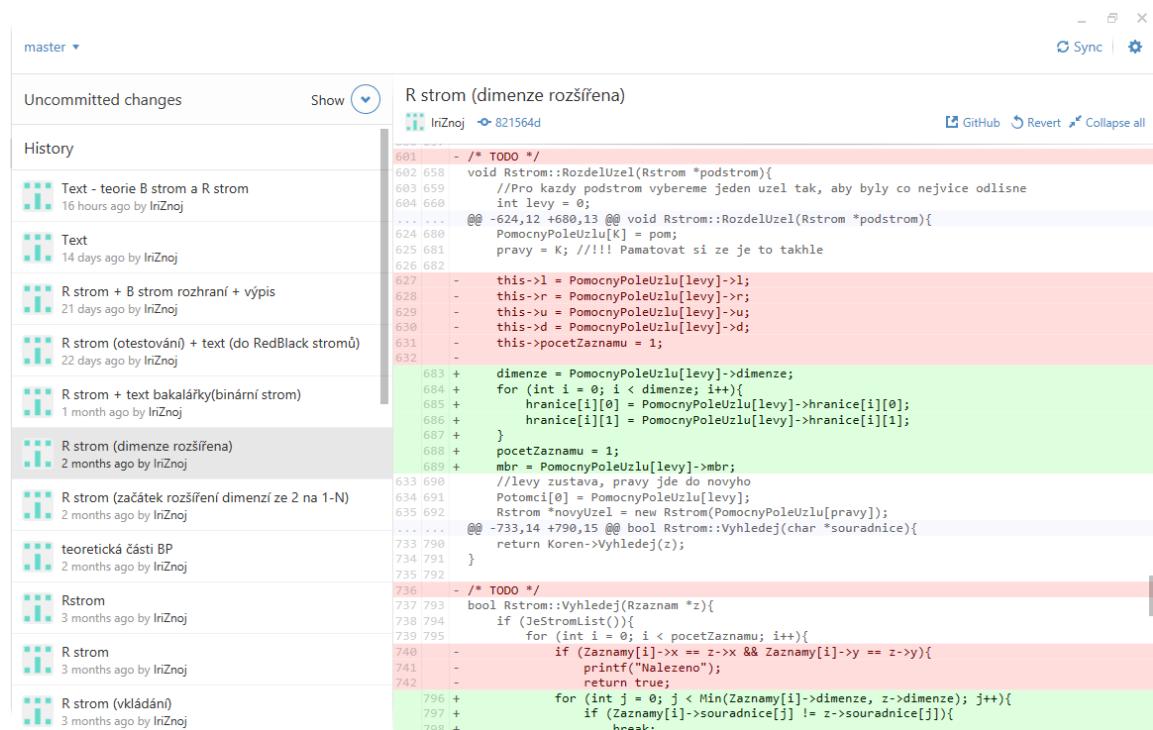
Obrázek 1.4: Porovnání algoritmů na rozdělávání uzlů: vlevo využití CPU při vkládání, vpravo při vyhledávání [4]

K rozštěpení uzlu existují 3 algoritmy: Exhaustive Algorithm generuje všechny možnosti, jak lze uzel rozštěpit a vybere tu nejlepší. Má však exponenciální složitost a je tak příliš pomalý pro velké uzly. Dalším algoritmem s kvadratickou složitostí je Quadratic Split. Funguje tak, že vybere 2 objekty, které by mezi sebou vytvořily nejvíce volného prostoru. Jinými slovy 2 objekty, které by vytvořily největší MBR. Zbývající objekty jsou přiřazovány podle toho, jak moc záleží na tom, ke kterému prvku budou přiřazeny. Jinými slovy nejdříve se přiřadí ty prvky, které by u jednoho prvku výrazně zvětšily MBR a u druhého podstatně méně. Prvky se samozřejmě musí rozdělit tak, aby byla splněna podmínka minimálního počtu prvků v každém z nově vytvořených objektů. Tento algoritmus je nejpoužívanější, protože má nejlepší poměr mezi složitostí algoritmu a vhodného rozdělení uzlu. Posledním zde zmíněným algoritmem je Linear Split. Tento algoritmus s lineární složitostí je stejný jako Quadratic Split s tím rozdílem, že najde 2 nejvzdálenější objekty a ke každému z nich pak vloží v náhodném pořadí do toho uzlu, jehož MBR by se zvětšila nejméně. Pokud se do některé skupiny musí vložit zbývající uzly tak aby byla dodržena podmínka minimálního počtu prvků v uzlu, pak je to provedeno a algoritmus končí. [4]

## 3 Implementace

### 3.1 Git

Při implementaci kódu jsem použil k jeho zálohování, správě verzí a obnovování systém Git s ukládáním nejen zdrojových kódů, ale i veškerých dat spojených s touto bakalářskou prací, jakými jsou například testovací data, text bakalářské práce, část použité literatury, obrázky, aj., na [https://github.com/IriZnoj/n\\_grams](https://github.com/IriZnoj/n_grams). Využil jsem repozitář webové služby GitHub proto, že jeho prostředí je mi blízké, je bezplatný a má širokou základnu uživatelů, což pro mne znamená jistou záruku spolehlivosti. Jako klienta jsem nejprve na svém počítači využíval služeb příkazové řádky, ale během implementace stromových struktur jsem přešel na program GitHub pro Windows, nabízený zdarma webovou službou GitHub.



Obrázek 1.5: Ukázka repozitáře programu GitHub pro Windows

### 3.2 Rozhraní a prostředí

V rámci této bakalářské práce jsem naimplementoval B+ strom popsáný v kapitole 2.6.2 a R-strom, který je popsán v kapitole 2.7. K implementaci jsem v souladu se zadáním bakalářské práce použil jazyk C s některými prvky jazyka C++ (např. třídy). Použití těchto struktur se v běžném použití, tedy ve volání veřejných metod použitých k vkládání nových záznamů do stromu (void VlozZaznam(char vstup[])), k vyhledávání ve stromové struktuře (bool Vyhledej(char retezec[])), k výpisu všech položek ve stromu uložených (void Vypis()) a k volání metody pro výpis stromové struktury (void UkazStrom()), nemění. Při vkládání do stromu jsou jednotlivé n-

gramy odděleny koncem řádku a jejich načítání probíhá ze standardního vstupu. Výpis n-gramů uložených v B+ stromu, výpis struktury jejich uložení či informace o existenci některého n-gramu ve struktuře jsou vypisovány na standardní výstup.

Jako vývojové prostředí k implementaci stromových struktur jsem se rozhodl použít Microsoft Visual Studio 2013 které firma Microsoft poskytuje studentům katedry informatiky zdarma pro nekomerční účely v rámci MSDN Academic Alliance.

### 3.3 Implementace B+ stromu

V implementovaném B+ stromu lze nastavit hodnotu konstanty K, která uvádí maximální počet klíčů ve stránkách. Tedy například pro  $K = 4$  může být v každé stránce 2 - 4 záznamů a tedy až 5 odkazů na potomky, pokud se nejedná o listovou stránku. Lze také nastavit konstantu MAX\_SLOVO, která uvádí maximální počet znaků, který se z vkládaného n-gramu do B+ stromu uloží.

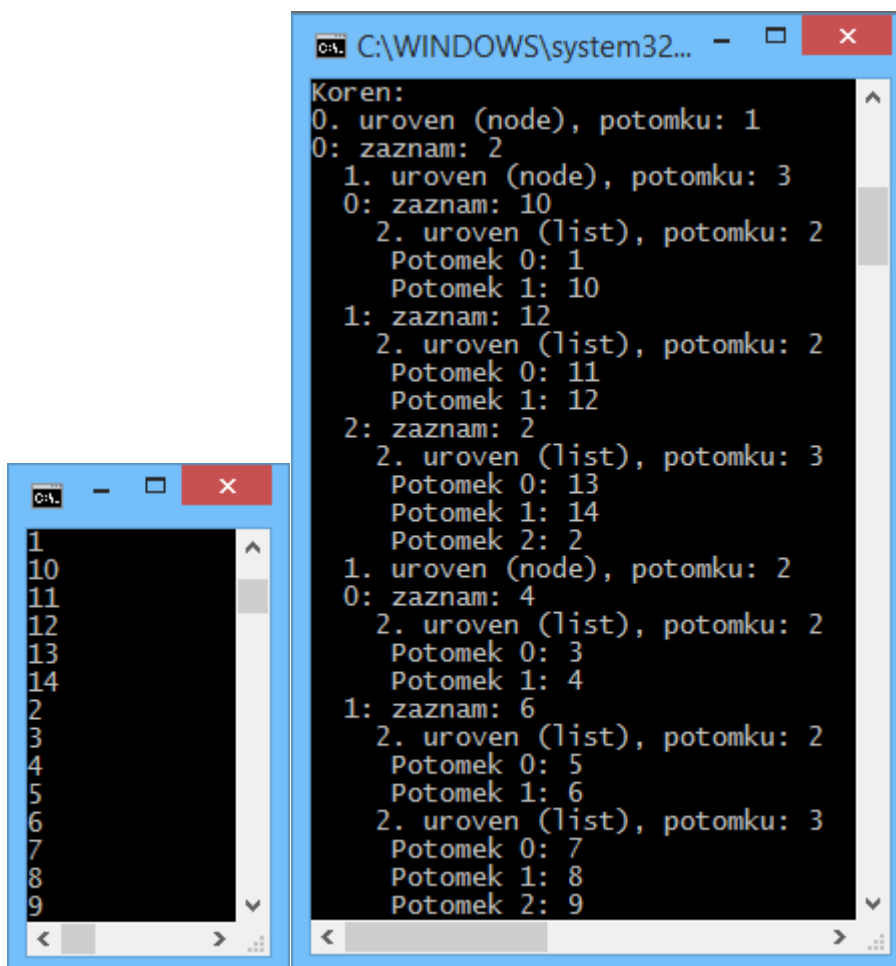
Vkládání do B+ stromu i vyhledávání v něm je implementováno tak, jak je popsáno v kapitole 2.6 pojednávající o B stromech. Když je vkládán nový záznam, tak je vždy vložen až do příslušné listové stránky (podle porovnání hodnot klíčů v jednotlivých úrovních) na pozici, která případně klíči nového záznamu v porovnání s ostatními záznamy v listové stránce. Pokud je však listová stránka zcela zaplněna (obsahuje K záznamů), tak proběhne její štěpení. Štěpení zde konkrétně probíhá tak, že v zaplněné stránce zůstane polovina záznamů (konkrétně  $\left\lfloor \frac{n}{2} \right\rfloor$ ) a zbylé záznamy se přesunou do stránky nové. Nový záznam je pak vložen do jednoho z těchto dvou stránek v závislosti na hodnotě klíče. Dle pravidel B+ stromu při štěpení stránek zůstávají všechny záznamy v listových stránkách. V této implementaci se do rodiče neukládá pouze klíč, ale interference na celý záznam, což je paměťově stejně náročné řešení, které zachovává stejnou strukturu jak pro vnitřní stránky, tak pro listové. Listové stránky se tak od těch vnitřních liší pouze tím, že neobsahují žádné odkazy na potomky. Pokud je vkládán záznam, který již ve struktuře existuje, pak je zcela ignorován. Toto chování lze ve zdrojovém kódu snadno změnit za pomoci podmínky, která je v kódu předpřipravena a zakomentována. Uvnitř této podmínky lze doimplementovat ukládání duplicit, doplnit záznam, který se již ve struktuře nachází, o další data, či provést nějakou jinou činnost s těmito daty.

K vyhledávání v implementovaném B+ stromu slouží metoda Vyhledej(char \*text), která je volána v kořenové stránce. Rekurzivně se pak v závislosti na hodnotě klíče hledaného záznamu a hodnotách klíčů ve stromové struktuře postupuje směrem k listové stránce. V listové stránce se porovná hodnota klíče hledaného záznamu s hodnotami klíčů všech záznamů v této stránce. Pokud se klíč shoduje, pak metoda vrací True (záznam se v B+ stromu nachází) a v opačném případě False (záznam nebyl nalezen).

K vypsání záznamů je možné použít metodu Vypis(). Metoda volá privátní metodu VypisPolozky(Bstrom \*strom), kde jako parametr vkládá kořen stromu. Pokud je parametrem stránka, která není listová, tak se volá tato metoda rekurzivně pro všechny její potomky, kteří slouží jako její parametr. Pokud je parametrem listová stránka, pak jsou všechna data jednotlivých

klíčů uložených v této listové stránce vypísána. V tomto případě jsou data stejná jako klíče používané pro správné zařazení záznamu do stromu. Tímto způsobem se data vypíší od nejlevějšího záznamu nejlevější listové stránky až po nejpravější záznam nejpravější listové stránky B+ stromu. Záznamy se tak vypíší vzestupně seřazeny. Na obrázku 1.6 vlevo je znázorněn výstup metody Vypis() pro vstupní data "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14".

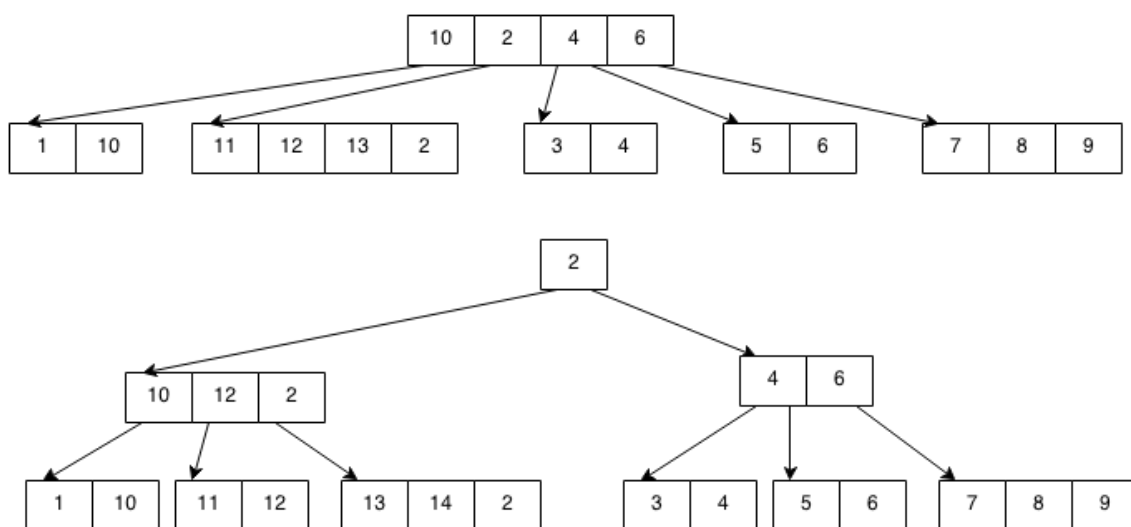
Dále lze vypsat data s dalšími informacemi, a to pomocí veřejné metody UkazStrom(), která volá na kořenové stránce metodu VypisZaznamySPotomky(int hloubka). Parametr hloubka udává, v jaké hloubce se nachází stránka, na které je metoda volána. Tato metoda pracuje obdobně jako metoda pro výpis dat popsaná výše s tím rozdílem, že vypisuje data strukturovaně tak, aby kopírovala strukturu B+ stromu. Kromě hloubky jsou vypisovány pro každou nelistovou stránku informace, že se jedná o vnitřní stránku, hloubka (vzdálenost od kořene), počet potomků a hodnoty jejich klíčů. Pro listové stránky je vypisována informace, že se jedná o listy, vypíše se počet záznamů a pro každý z nich data, která představují. Ukázka výpisu pro stejná vstupní data stejná jako v předchozím případě pro metodu Vypis() je na obrázku 1.6 vpravo.



Obrázek 1.6: Vlevo: ukázka výstupu metody Vypis(),  
vpravo: ukázka výstupu metody UkazStrom()



Na obrázku 1.7 je znázorněno vkládání do implementovaného B+ stromu, který je složen z klíčů "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12" a "13". Čísla jsou zde použita pro jejich krátký tvar a názornost. Klíče mohou tvořit libovolné sekvence omezená velikostí konstanty MAX\_SLOVO. Vkládán je n-gram "14", který způsobí nejdříve štěpení listové stránky s n\_gramy "11", "12", "13", "2" na 2 listové stránky s hodnoty "11", "12" v levé a "13", "2" v druhé, do které je vložen n-gram "14" tak jak je znázorněno na obrázku. Prostřední klíč původní listové stránky (tedy n-gram "12") je vložen do rodičové stránky, což způsobí další štěpení. Jelikož se jedná o kořen B+ stromu, tak toto štěpení způsobí růst výšky stromu.



Obrázek 1.7: Ukázka štěpení B+ stromu při vložení n-gramu s klíčem 14

### 3.4 Implementace R-stromu

Jelikož R-strom je struktura, která vychází z B+ stromu, tak jsem i k jeho implementaci B+ strom, popsany v kapitole 3.3, využil. Konstanty ovlivňující vlastnosti struktury jsou konstanta K udávající maximální počet klíčů v jednom uzlu, konstanta m udávající nejmenší počet záznamů v každém uzlu, konstanta MAX\_SLOVO udávající maximální velikost záznamu a konstanta D představující maximální dimenzi vkládaného záznamu.

Vkládání může být prováděno dvěma způsoby. Buď je parametrem řetězec složený z čísel a bílých znaků, který je pomocí metody RetezecNaCisla(char \*retezec) rozparsován na několik čísel, jejichž počet je omezen konstantou D. Takováto čísla pak představují souřadnice jednotlivých dimenzí vkládaného záznamu. Druhou možností je vložit záznam se 2 parametry, kterými jsou pole celočíselných hodnot a jeho velikost. Data jsou opět vždy vkládána tak, že se volá metoda pro vložení záznamu na kořenový uzel, z něhož se pak hledá na základě hranic jednotlivých mbr listový uzel. Záznamy se vždy vkládají do listových uzlů.

Pokud je záznam vložen do listu, pak je volána metoda ZkontrolujHranici(int hranice[], int dimenze), která pomocí metody PorovnejAZmen(int hranice[], int dimenze) zjistí, jestli je

třeba rozšířit hranice mbr. Pokud je třeba rozšířit hranice, tak jsou hranice v této metodě změněny a je vrácena hodnota True pro indikaci, že byla provedena změna hranic. V případě, že byla provedena změna, tak se v metodě ZkontrolujHranici(int hranice[], int dimenze) zavolá metoda PrepocitejObsah(), která pomocí metody VypocitejObsah(int hranice[][2], int dimenze) vypočítá novou hodnotu mbr tohoto listu. Pokud list není zároveň kořen, tak se stejným způsobem kontroluje, jestli je potřeba rozšířit hranice rodičovského uzlu, případně se rozšíření provede, přepočítá se mbr a opět se volá rodičovský uzel a to až do té doby, dokud nedojde ke zkontrolování uzlu, nebo pokud nebude třeba rodiče měnit. Výpočet mbr probíhá tak, že se násobí rozdíly nejnižších a nejvyšších hodnot jednotlivých dimenzí. Pokud některá dimenze neobsahuje žádnou, nebo obsahuje pouze jednu hodnotu, pak je násobena číslem 1. Nové záznamy jsou vkládány do pole zleva doprava. Záznamy v listových uzlech nejsou nijak seřazeny. Jejich pořadí závisí na tom kdy byly do listu vloženy a to jak při vkládání do stromu tak při vkládání při rozštěpení uzlu.

Pokud je volána metoda pro vkládání záznamu na nelistovém uzlu, tak se hledá potomek s takovými hranicemi, které potřebují nejmenší rozšíření. Jelikož se hranice jednotlivých uzlů mohou překrývat, tak může existovat více potomku, které nepotřebují, nebo potřebují stejné rozšíření, aby se do nich mohl vložit nový prvek. Pokud k tomu dojde, tak má přednost potomek s menší hodnotou mbr. Tento postup se rekurzivně opakuje až do té doby, dokud se nevkládá do uzlu, který je listový.

Jestliže je záznam vkládán do listového uzlu, ve kterém však již existuje K záznamů, tak je potřeba, stejně jako u B+ stromů, uzel rozštěpit. Rozhodl jsem se k tomu použít algoritmus Kvadratický Split. Ke všem K záznamům v tomto uzlu je přidán ještě i záznam vkládaný a za pomoci metody VybarDvaZaznamy(int &l, int &r) vyberu takové dva záznamy, které by vytvořily největší hodnotu mbr, kdyby byly oba v jednom uzlu. Tyto záznamy jsou vybrány tím způsobem, že pro všechny dvojice je vypočítána hodnota mbr. Jakmile mám 2 záznamy, tak jeden zůstane v levém listu a pro druhý je vytvořen nový, pravý list. Ostatní záznamy v levém listu nezůstávají, ale podle potřeby budou znovu vloženy, jinak by bylo nutné po přesunu některých záznamů do pravého uzlu vypočítat nové hranice, hodnotu mbr a prvky přesunout vlevo. Zbývající prvky jsou přiřazovány tak, že pomocí metody DalsiPrvek(Rstrom \*strom, bool prepocitatLevy, bool prepocitatPravy, long long zmenal[], long long zmenar[]) je vypočítáno číslo udávající, jak moc záleží, kam se daný záznam zařadí. Toto číslo je vypočítáno jako rozdíl změny mbr levého listu potřebné k uložení záznamu a změny mbr pravého listu, pokud by byl záznam uložen právě tam. Záznam s takovým to největším rozdílem bude přiřazen jako první. Pro každé přiřazení je potřeba opět přepočítat velikost změny pro záznamy, které ještě přiřazeny nebyly. Mnou implementovaný R strom je vyvinut tak, že jakmile je záznam vložen do pravého uzlu, tak pro všechny záznamy není třeba znovu přepočítávat změnu mbr pro levý uzel, ale pouze pro pravý a velikost celkového rozdílu. Stejně tak, pokud je záznam vložen do levého uzlu, tak není třeba znovu počítat změnu mbr po uzel pravý. Aby byla dodržena podmínka minimálního počtu záznamů v každém listu, tak jakmile hrozí její porušení, tak se zbývající záznamy vloží tak, aby k jejímu porušení nedošlo. Následně se do rodiče vloží reference na nový list, případně, pokud je dělen kořen, tak se vytvoří nový kořen se 2 potomky (původním listem a novým listem), vypočítá se jeho hranice a mbr. V případě, že se odkaz na nový listový uzel vkládá do rodiče, ve

kterém již na tento odkaz není místo, tak musí dojít ke štěpení vnitřního uzlu. Štěpení vnitřního uzlu probíhá obdobně jako štěpení listu.

Při jednoduchém výpisu záznamů ze stromu, tedy při volání metody Vypis() se volá na kořenovém uzlu metoda VypisPolozky(). Metoda je rekurzivní a volá sama sebe pro všechny potomky, pokud se jedná o vnitřní uzel. V případě listových uzlů metoda vypíše hodnotu každé dimenze všech záznamů uložených v listech a to stejně jako u B+ stromu od nejlevějšího záznamu nejlevějšího listového uzlu až po nejpravější záznam nejpravějšího listového uzlu R stromu.

Další možností je vypsat data metodou UkazStrom(), která na kořeni R stromu volá metodu VypisZaznamySPotomky(int hloubka). Parametr udává hloubku zanoření ve stromu a díky tomu lze z výpisu vypořadovat strukturu stromu. Výpis obsahuje hranice mbr každého uzlu, informaci jestli se jedná o listový či vnitřní uzel, hodnotu mbr a v listových uzlech pro každý záznam jeho pořadí v listu, dimenzi a pro každou dimenzi její hodnotu. Na obrázku 1.8 je uveden příklad takového výpisu pro vstupní data "1 2 3 4 5", "1 2 3", "1 3 2 4", "4 5 6 8", "7 4 1 0", "0 1 2 3 5", "4", "56 6", "8 7 77 8" a "9 9 9".

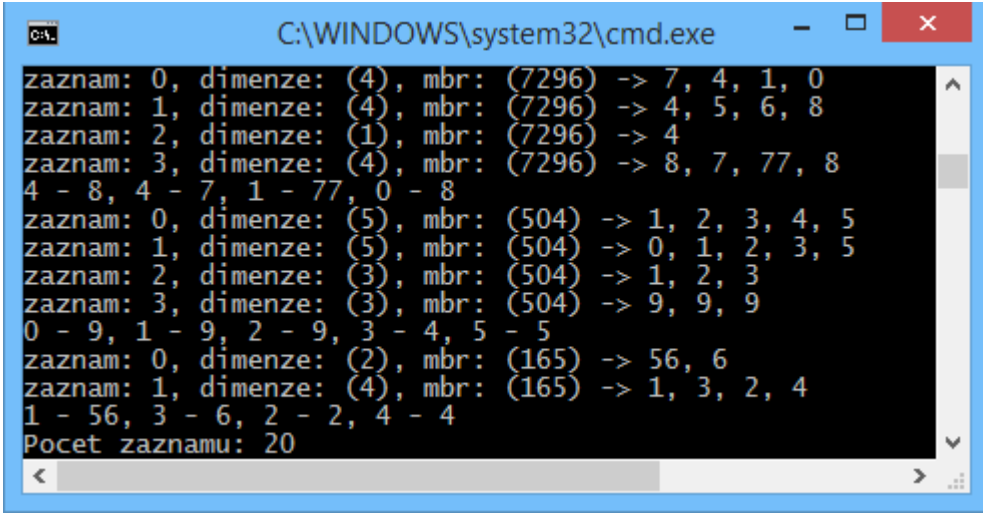
```

C:\WINDOWS\system32\cmd.exe
Koren:
0) (node), dimenze:
0: 0 - 56,
1: 1 - 9,
2: 1 - 77,
3: 0 - 8,
4: 5 - 5
1) (leaf) (mbr: 7296)
0: 4 - 8
1: 4 - 7
2: 1 - 77
3: 0 - 8
   zaznam: 0, dimenze: (4) -> 7, 4, 1, 0
   zaznam: 1, dimenze: (4) -> 4, 5, 6, 8
   zaznam: 2, dimenze: (1) -> 4
   zaznam: 3, dimenze: (4) -> 8, 7, 77, 8
1) (leaf) (mbr: 504)
0: 0 - 9
1: 1 - 9
2: 2 - 9
3: 3 - 4
4: 5 - 5
   zaznam: 0, dimenze: (5) -> 1, 2, 3, 4, 5
   zaznam: 1, dimenze: (5) -> 0, 1, 2, 3, 5
   zaznam: 2, dimenze: (3) -> 1, 2, 3
   zaznam: 3, dimenze: (3) -> 9, 9, 9
1) (leaf) (mbr: 165)
0: 1 - 56
1: 3 - 6
2: 2 - 2
3: 4 - 4
   zaznam: 0, dimenze: (2) -> 56, 6
   zaznam: 1, dimenze: (4) -> 1, 3, 2, 4

```

Obrázek 1.8: Ukázka výstupu metody UkazStrom()

Poslední možností je zde ještě použití metody `VypisPlus()`, která opět na kořenovém uzlu volá rekurzivní metodu `VypisPolozkyPlus()`, která volá sama sebe na potomky nelistových uzlů. U listových uzlů vypisuje kromě informací, které jsou vypsány při volání metody `Vypis()` ještě hranice listového uzlu a pro každý jeho záznam velikost dimenze a velikost mbr. Nakonec je pak vypsán celkový počet záznamů uložených ve stromu. Ukázka výstupu metody `VypisPlus()` je na obrázku 1.9



```
C:\WINDOWS\system32\cmd.exe
zaznam: 0, dimenze: (4), mbr: (7296) -> 7, 4, 1, 0
zaznam: 1, dimenze: (4), mbr: (7296) -> 4, 5, 6, 8
zaznam: 2, dimenze: (1), mbr: (7296) -> 4
zaznam: 3, dimenze: (4), mbr: (7296) -> 8, 7, 77, 8
4 - 8, 4 - 7, 1 - 77, 0 - 8
zaznam: 0, dimenze: (5), mbr: (504) -> 1, 2, 3, 4, 5
zaznam: 1, dimenze: (5), mbr: (504) -> 0, 1, 2, 3, 5
zaznam: 2, dimenze: (3), mbr: (504) -> 1, 2, 3
zaznam: 3, dimenze: (3), mbr: (504) -> 9, 9, 9
0 - 9, 1 - 9, 2 - 9, 3 - 4, 5 - 5
zaznam: 0, dimenze: (2), mbr: (165) -> 56, 6
zaznam: 1, dimenze: (4), mbr: (165) -> 1, 3, 2, 4
1 - 56, 3 - 6, 2 - 2, 4 - 4
Pocet zaznamu: 20
```

Obrázek 1.9: Ukázka výstupu metody `VypisPlus()`

## 4 Měření efektivity struktur

	Nejhorší případ			Průměrný případ		
	Insert	Search	Select	Insert	Search hit	Search miss
Indexované pole	1	1	M	1	1	1
Setříděné pole	N	N	1	N/2	N/2	N/2
Setříděný spojový seznam	N	N	N	N/2	N/2	N/2
Nesetříděné pole	1	N	$N \lg N$	1	N/2	N
Nesetříděný spojový seznam	1	N	$N \lg N$	1	N/2	N
Binární vyhledávání	N	$\lg N$	1	N/2	$\lg N$	$\lg N$
Binární vyhledávací strom	N	N	N	$\lg N$	$\lg N$	$\lg N$
Red-black tree	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$	$\lg N$
Randomized tree	$N^*$	$N^*$	$N^*$	$\lg N$	$\lg N$	$\lg N$
Hashing	1	$N^*$	$N \lg N$	1	1	1

[3] str 494

---

## Použitá literatura

- [1] ŽALUD, Václav. *Moderní radioelektronika*. 1. vyd. Praha: BEN, 2000, 656 s. ISBN 80-86056-47-3.
- [2] Výkony, limity ČTU a GL č. 12/R/2000. *KHnet.info* [online]. [cit. 2011-01-05]. Dostupné z: <http://forum.khnet.info/viewtopic.php?f=10&t=964#p7299>
- [3] Sedgewick, Robert. *Algorithms in C (Parts 1-4)*, 3. vyd. Addison-Wesley, 1998, 702 s. ISBN 0-201-31452-5.
- [4] A. Guttman, R-Trees: A dynamic index structure for spatial searching, in Proc. of ACM SIGMOD, June 1984, pp. 47–57.
- [5] <http://www.root.cz/knihy/pro-git/> - VYPSAT KNIHU