**Zadanie 2 – Vyhľadávanie v dynamických množinách**

Roman Páleník  
FIIT STU  
2020

Obsah

[1. Riešenie problematiky 3](#_Toc37952397)

[1.1 Úvod do problematiky 3](#_Toc37952398)

[1.2 Návrh riešenia 3](#_Toc37952399)

[1.3 Vlastné implementácie algoritmu 3](#_Toc37952400)

[1.3.1 AVL strom 3](#_Toc37952401)

[1.3.2 Chain metóda Hashovanie 10](#_Toc37952402)

[1.4 Prevzdaté implementácie 12](#_Toc37952403)

[1.4.1Červeno čierny strom 12](#_Toc37952404)

[1.4.2 Linear hashing 13](#_Toc37952405)

[2. Testovanie 14](#_Toc37952406)

[2.1 Všeobecná metodika testov 14](#_Toc37952407)

[2.2 Prvá sada testov – lineárne pridávané prvky 14](#_Toc37952408)

[2.3 Pridávanie alternujúcich prvkov 17](#_Toc37952409)

[2.4 Pridávanie pseudonáhodných prvkov 20](#_Toc37952410)

[2.5 Všeobecný záver testov 23](#_Toc37952411)

# Riešenie problematiky

## Úvod do problematiky

V zadaní máme porovnávať porovnať rýchlosť algoritmov na dátové štruktúry. Rôzne algoritmy majú rôznu časovú zložitosť. V tomto zadaní ich porovnávam a analyzujem v rôznych situáciach na rôznych vstupoch.

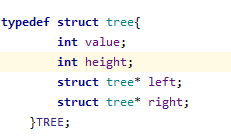
## Návrh riešenia

Riešenie porovnania realizujem pomocou 2 svojích vlastných implementácii AVL stromu a hash tabuľky a porovnávam to s 2 prebratými algoritmami z internetu. Čas meriam pomocou knižnice <rpc.h> ktorá mi sleduje počet frekvencii procesora a z neho vypočítava čas. Mám navrhnuté 4 sady testov – lineárne pridávanie od najväčšieho po najmenšie a opačne, alternujúco (1,n-1,2,n-2) a nakoniec aj pseudonáhodne s pevným seedom 1.

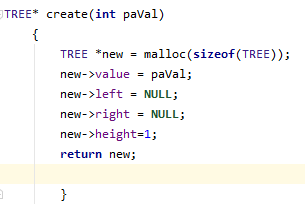
## Vlastné implementácie algoritmu

### AVL strom

Na vlastnú implementáciu vyváženého binárneho vyhľadávajúceho stromu som si vybral AVL vyhľadaváci strom. Strom mám realizovaný cez štruktúru. V nej si pamätám hodnotu daného uzla alebo listy, jeho výšku a smerník na pravý a ľavý pod-strom. Výška reprezentuje v mojom prípade, najdlhšiu cestu od daného uzlu k listu. Každý list má výšku jedna 0+1(reprezentácia toho daného listu) . Insert a search mám realizovaný rekurzívne.

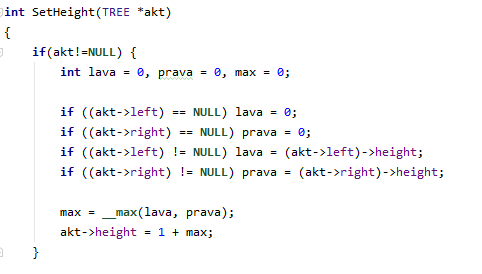


Insert – ako argumenty funkcie príde hodnota, ktorú chceme vložiť a koreň stromu. Funkcia skontroluje, či existuje strom a ak existuje porovná hodnotu koreňa s hodnotou, ktorú chceme zapísať. Ak hodnota, ktorú chceme zapísať je menšia ako hodnota v koreni, funkcia sa rekurzívne posunie na ľavý pod-strom inak na pravý. Nový koreň stromu je koreň ľavého alebo pravé pod-stromu. Funkcia sa rekurzívne vola dovtedy, dokým nenájdem už existujúci prvok alebo kým nenájdem voľné miesto, kam by som mohol uložiť požadovanú hodnotu. Vyhradím pamäť pre novú štruktúru(malloc) vložím do nej hodnotu, smerníky na pod-stromy nastavím na NULL a výšku nastavím na 1. Ak je hodnota už v strome, nezapíšem ju druhý krát.



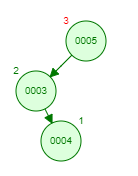
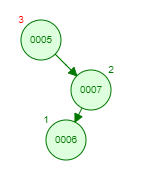
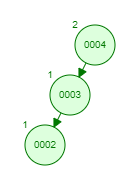
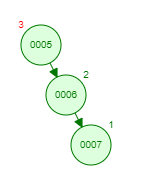
Rátanie výšok:

Keď vložím hodnotu do stromu, začnem sa rekurziou vracať po vetvách stromu. Keďže som pridal prvok, tak v každom vrátení sa musím nanovo prepočítať výšku daného uzla. Výšku rátam ako 1 + maximum z ľavého a pravé pod-stromu. Ak pod-strom neexistuje jeho výška je 0.



Rotácie:

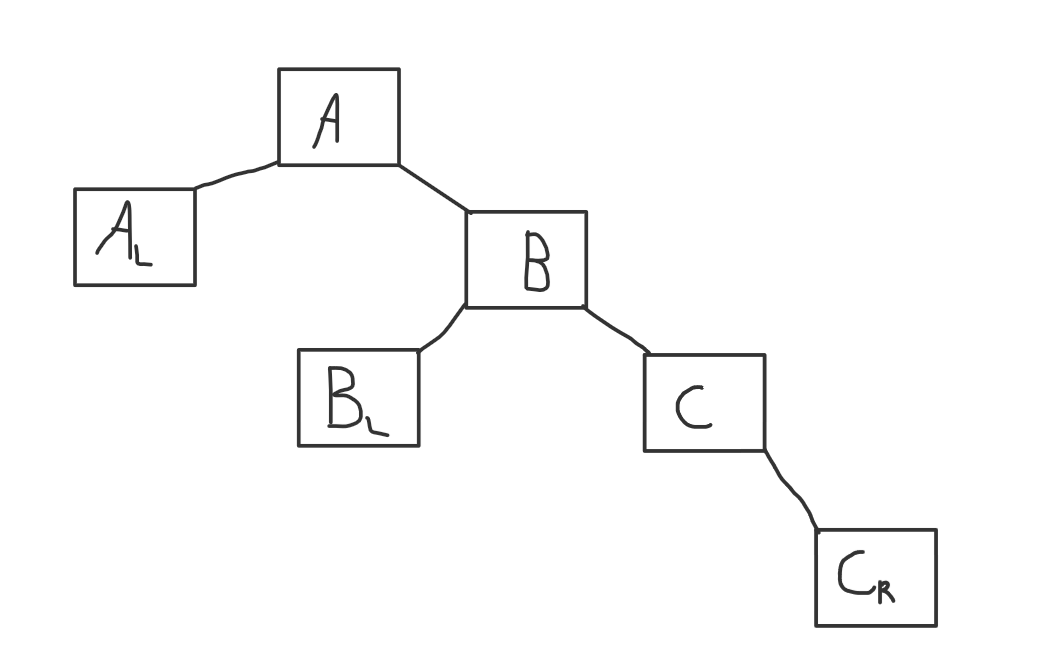
V každom vnorení funkcie sa tak isto skontroluje, či je strom vyvážený. Strom je vyvážený vtedy ak rozdiel najdlhšej cesty k listu z pravé pod-stromu a ľavého pod-stromu je v intervale od -1 až 1. Ak strom nie je vyvážený treba ha následne vyvážiť. Tu sú prístupné 4 scenáre: strom je prevážený doprava(obz.1), doľava(obr.2), doprava-doľava(obr.3) alebo doľava-doprava(obr.4)



Obr1 obr2 obr3 obr4

**Riešenie prvej situácie: preťaženie doprava**

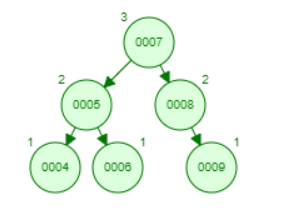
Ak je strom preťažený doprava(vpravo je viac prvkov ako vľavo a strom nie je vyvážený) vykoná sa funkcia PP\_Balance(). Táto funkcia vyrovná strom preťažený v tejto situácii nasledovne. Uvažujme strom nižšie na obrázku:

Po pridaní prvku väčšieho C sa tento strom preváži doprava:  


Strom sa teda vyrovná nasledovne: Nový koreň stromu bude z B. Jeho ľavý pod-strom bude tvoriť A s Al a pravý bude tvoriť C s Cr. Bl ako ľavý potomok B, musí ísť na ľavú stranu. Avšak v pôvodnom strome je väčšie ako A tak sa musí pridať ako A pravý potomok.

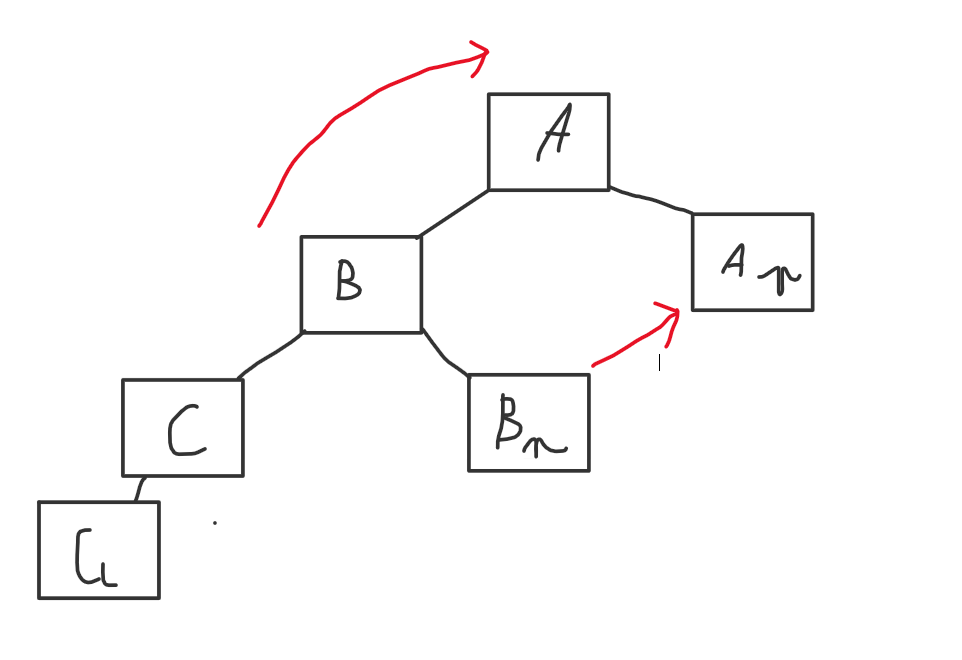
Inorder výpis pred vyvážením:  
  
4 = AL, 5=A,7=B,6=BL,8=C  
Po pridaní 9 a vyvážení, inorder:  


Na výpise vidno, že 6 ako BL sa teraz stala pravým pod-stromom 5-ky ako A a 7 sa stal novým koreňom.



**Riešenie druhej situácie: preťaženie doľava**

Proces zopakujeme analogicky aj s druhou stranou.

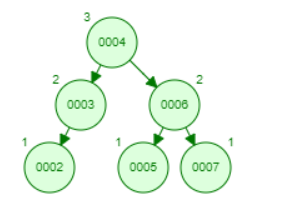


Strom preťažený do ľava. Zrotuje to doprava cez vrchol B a teda z B nový vrchol, ľavý pod-strom C, pravý pod-strom A. Bp sa priradí ako ľavý pod-strom A-čka.

In order pred vyváženým

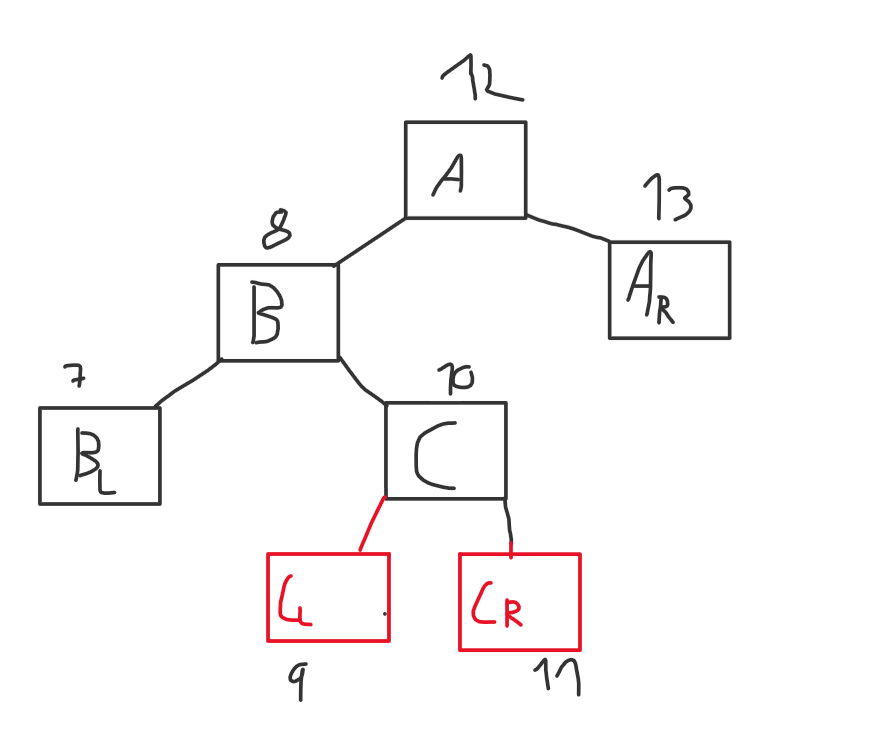


Po pridaní 2 vyvážení:  

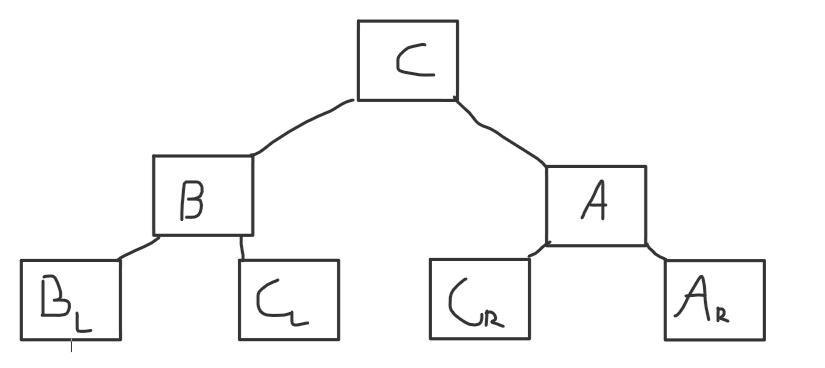



**Riešenie tretej situácie: preťaženie doprava doľava**

Túto situáciu som riešil v samostanej funkcii pomocou jedného rotovania. Zisťujem to podmienkou tak, že či v tom preťažení napravo. Ak som tak skontrolujem či sedí výška z ľavého jeho pod-stromu a pravého jeho pod-stromu. Ak je na ľavo viac prvkov nastáva táto situácia(Buď som pridal Cl alebo Cr ale rozlišuje to potom kam sa mi ten prvok napojí):



Nový koreň bude z C, ľavý pod-strom B. B ľavý pod-strom bude z Bl a B pravý pod-strom bude Cl(ak preťaženie vznikne pridaním prvku menšieho ako C). C pravý pod-strom je A, jeho ľavý pod-strom je Cr(ak nevyrovnanosť vznikla po pridaní prvku väčšieho ako C)



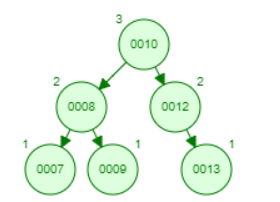
(Cr a Cl priradenie podľa vzniknutej situácie)

InOrder pred vyrovnaním:



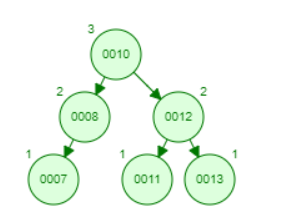
Pridanie 9(Cl) následné vyrovnanie:





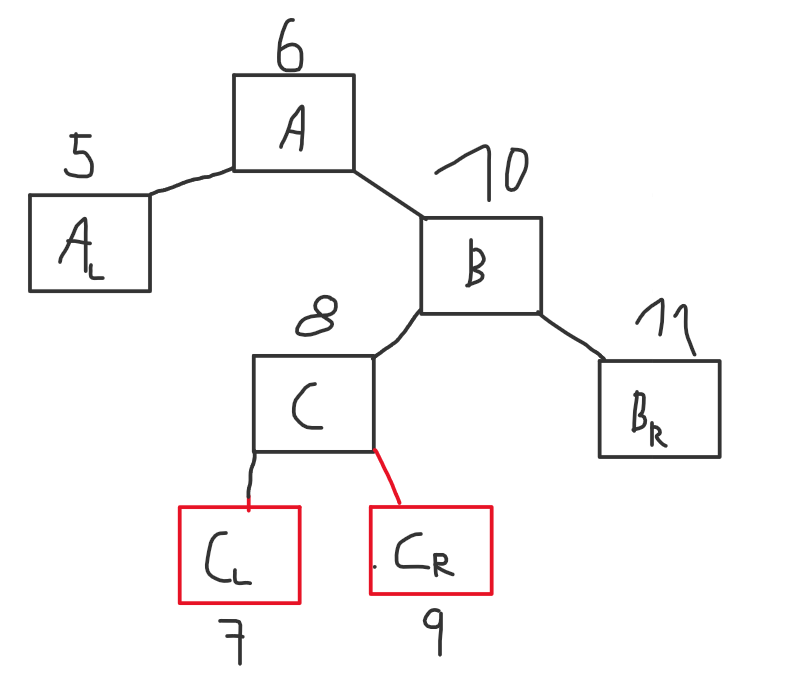
Po vložení 11(Cr prípad)





**Riešenie štvrtej situácie: preťaženie doľava doprava**

Program postupuje rovnako ako pri tretej situácii.



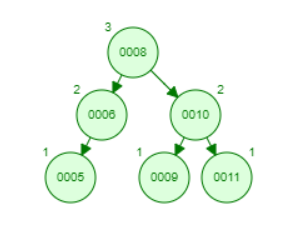
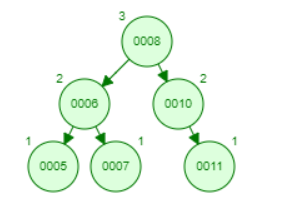
Z C sa stane nový koreň. Z A sa stane jeho ľavý pod-strom. Ľavý podstrom A zostane Al a pravý pod-stom bude Cl(ak preťaženie vzniklo po pridaní tohto prvku, inak bude prázdny). Pravý pod-strom koreňa bude B a jeho pravý pod-strom zostane Br a ľavý bude Cr(ak preťaženie vznklo po pridaní Cr prvku)

InOrder pred pridaním Cl alebo Cr prvku:



Po pridaní 7 (Cl prvku):





Po pridaní 9(Cl prvku):



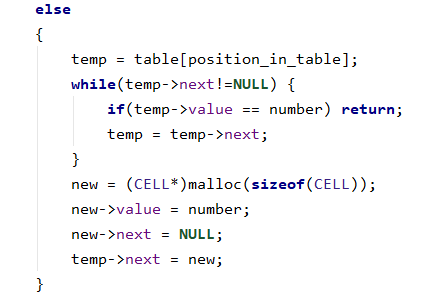
### Chain metóda Hashovanie

Na vlastnú implementáciu hašovania som zvolil hašovanie s riešením kolízii pomocou zreťazených zoznamov. Tabuľku mi zväčší keď je celkový počet zahešovaných prkov 1,5 krát väčší ako veľkosť tabuľky. Tabuľka je realizovaná ako pole štruktúr. Na začiatku si ju celú vyNULLujem a nastavím size na prvočíslo. Vždy bude veľkosť tabuľky prvočíslo. Je to z toho dôvodu, že po delení prvočíslom vznikne čo najviac rôznych zvyškov z dôvodu nesúdelideľnosti prvočísla a ľubovoľného iného čísla. Postupne opíšem ako mi funguje Insert, riešenie kolízii, Resize a Search.

**Insert**

Na začiatku sa vypočíta pozícia kam sa má daný prvok zahashovať. Vzťah na výpočet: Pozícia\_v\_tabuľke = veľkosť\_vkladaného\_prvku % veľkosť tabuľky. Ak je to miesto voľné číslo tam zapíšem. Ak to miesto nie je voľné tak hodnotu pridám na koniec zreťazeného listu na ten danej pozícii. Ak je hodnota už v tabuľke, druhý krát ju nepridám.



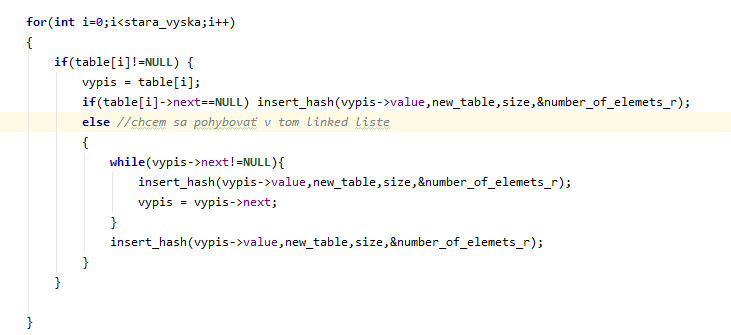
**Riešenie kolízii**  


Prehľadávanie linked listu a prerušenie ak nájdem už vloženú hodnotu

Vytvorenie novej štruktúry v hašovacej tabuľke.

**Resize**

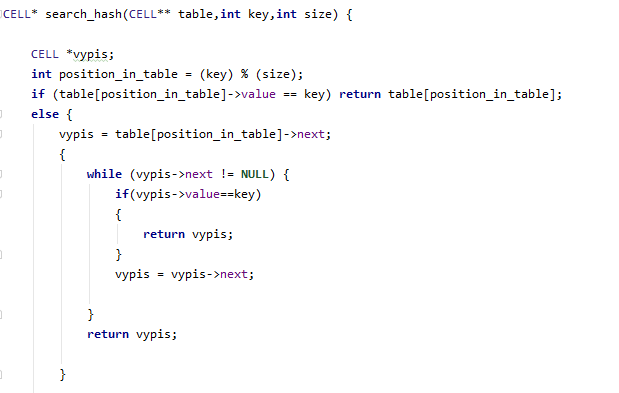
Zväčšenie tabuľky realizujem preto, aby som zefektívnil rýchlosť hash tabuľky. Namiesto toho aby sa mi vytvárali dlhé zreťazené listy na každej pozícii tabuľky, tabuľku zväčším a znova všetko prerozdelím. Tabuľku zväčšujem ak je počet prvok 1,5 krát väčší ako veľkosť tabuľky. Novú veľkosť tabuľky vytváram tak, že hľadám najbližšie prvočíslo k veľkosť\_tabuľky\*2. Po zväčšení tabuľky sa každý prvok znova hashuje a umiestni do novej tabuľky.



Znova zahešovanie prvku

**Search**

Hľadanej hodnote vypočítam pozíciu na ktorej sa nachádza a pristúpim na to miesto. Na danej pozícii skontrolujem či je tam daná hodnota, ak nie znamená to prítomnosť zreťazeného zoznamu a musím začať kontrolovať každý jeden prvok v zozname, dokým sa nedostanem na žiadaný prvok s hľadanou hodnotou. Ako návratovú hodnotu funkcia vracia smerník na pozíciu v hašovacej tabuľke hľadanej hodnoty.



Vrátenie hodnoty ak je na konci zreťazeného zoznamu

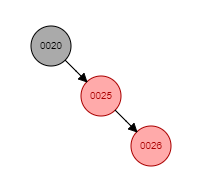
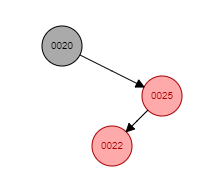
Kontrola či sa tam nachádza žiadaná hodnota

## Prevzdaté implementácie

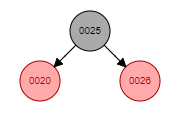
### 1.4.1Červeno čierny strom

Červeno čierny strom funguje na princípe, že každý vrchol je buď červený alebo čierny  
koreň je vždy čierny  
listy sú vždy čierne  
každý čierny vrchol má 2 čiernych potomkov  
Každá cesta z jedného vrcholu k listu obsahuje rovnaký počet čiernych vrcholov  
Vložený prvok zafarbím na červeno  
Strom nie je vyrovnaný ak sú pri sebe(rodič a potomok) červené

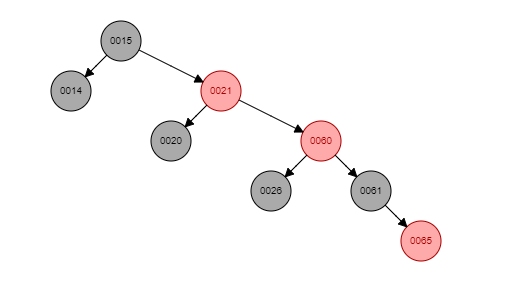
Insert –Ak je hodnota na vrchole menšia funkcia sa vnorím doprava, ak väčšia doľava. Touto analógiou sa algoritmus dostane až k listu, na chcenú pozíciu a zapíšem tam vrchol. Vložený vrchol zafarbím na červeno. Následne sa skontroluje či je strom vyrovnaný.

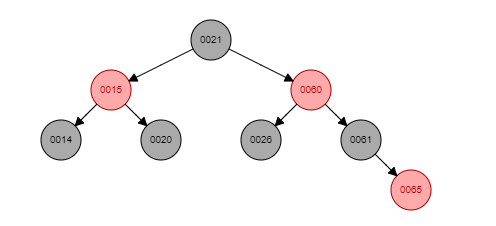
FixUp – funkcia skontroluje či je pridaný vrchol naľavo alebo napravo od rodiča a následne skontroluje jeho strýka(druhý potom rodiča pridaného listu). Potom funkcia skontroluje existenciu strýka a jeho farbu. Ak strýko je červenej farby a rodič čiernej tak sa strom len prefarbí. Strýko na čierno a starý otec na červeno. Ak je strýko čiernej farby tak neprebehne nič. Ak strýko neexistuje prebehne rotácia. Strom zrotuje tak, že stredný vrchol sa stane nový koreň pod-stromu  


Nový vrchol



Analógia rotácie platí aj na druhú stranu (preťaženie do ľava).   
Ak je rodič červenej farby musí prebehnúť rotácia doľava cez červeného rodiča, ktorý sa stane novým koreňom a preto sa nafarbí na čierno



Rotovanie cez 21, a následné prefarbenie 21 a 15  


### Linear hashing

Prevzatá implementácia používa metódu hash open adresing – linear probing. Princíp fungovania: hodnota sa v tabuľke zahashuje podľa hash funkcie. Kolízie sa rieši zahashošovaním na najbližšie voľné miesto lineárne. Pri hľadí funkcia pristúpi na pozíciu podľa hash funkcie. Ak sa tam nenachádza žiadaný prvok posunie sa na ďalšiu až kým nenájde žiadaný prvok. Z dôvodu malej veľkosti som musel implementovať zväčšenie tabuľky pri počte prvkov 0,75 krát veľkosť tabuľky.

# Testovanie

## Všeobecná metodika testov

Na testovanie rýchlostí a časovej efektivity daných algoritmov som navrhol 4 skupiny testov. 1 skupina pokrýva postupnosť od 0 do N, 2 skupina od N do 0, 3 skupina alternujúce prvky a 4 skupina pseudonáhodné prvky s pevným seedom. Na začiatku každého testu predložím teoretický predpoklad správania sa algoritmu. Potom zrealizujem merania a zhodnotím, či sa predpoklad naplnil alebo nenaplnil a prečo. Časovú efektivitu som meral pomocou funkcii z knižnice rpc.h, pomocou meraní frekvencii procesora. Výsledky som zdokumentoval do exelovskej tabuľky a výsledky vyhodnotil pomocou grafov. V každej skupine testov meriam rýchlosť funkcie insert, search, insert & search. Pri search teste vytvorenie tabuľky nerátam do času algoritmu.

**Vyriešenie duplicitného pridávania v algoritmoch** – aby som mal testy čo najviac objektívne musím docieliť aby mi algoritmy pristupovali rovnako k duplicitným vloženia. Tento problém nastane len pri náhodných číslach, preto som si vygeneroval dopredu pole s unikátnymi náhodnými číslami na elimináciu tohto problému. Pole je veľké 20M a pri vstupoch vkladám do funkcie najprv, prvých 10K prvkov, potom 100k....

## Prvá sada testov – lineárne pridávané prvky

V tejto sade testov som porovnával rýchlosť insert a search pri lineárnej postupnosti pridávaných prvkov.

**Predpoklad:**

Pri tejto postupnosti by malo pri stromoch nastať iba preťaženie doprava. Keďže AVL strom neovplvyní existencia kratšej cesty na pozícii n-1 v druhom potomkovi rodiča(strýka), bude rotácia prebehať častejšie ako pri červeno-čiernom strome, ktorá existencia strýka môže ovplyvniť. Čiže pre určité rotácie AVL stromu bude červeno čierny len prefarbovať vrcholy. Predpokladám teda, že z dôvodu menej častých rotácii bude insert rýchlejší pri červeno-čiernom strome. Avšak zo znalosti algoritmov vieme, že z tohto dôvodu je červeno-čierny strom menej vyvážený a preto insert bude pomalší.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10K | 100K | 1M | 5M | 10M |  |
| AVL strom | 0,00176 | 0,019673 | 0,235286 | 1,326001 | 2,909283 |  |
| Červeno-čierny strom | 0,001384 | 0,015255 | 0,197121 | 1,111957 | 2,384937 |  |
| Hashovanie chain | 0,001392 | 0,012679 | 0,177912 | 0,837625 | 1,762913 |  |
| Linear search | 0,001145 | 0,01484 | 0,134228 | 0,60682 | 1,244278 |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| AVL strom | 0,000602 | 0,007203 | 0,073721 | 0,396503 | 0,814833 |  |
| Červeno-čierny strom | 0,000617 | 0,007638 | 0,08444 | 0,461857 | 0,958058 |  |
| Hashovanie chain | 0,000318 | 0,003598 | 0,043167 | 0,219601 | 0,451249 |  |
| Linear search | 0,000072 | 0,001069 | 0,007448 | 0,037029 | 0,074789 |  |
|  |  |  |  |  |  |  |

**Výsledky merania:**

Merania mi potvrdili môj predpoklad o rýchlosti algoritmu. Červeno čierny je horšie vyvážený čo mu zvyšuje rýchlosť pri inserte ale uberá pri searchi. V linear\_hashi pri takto zadanej postupnosti nevzniknú žiadne kolízie, preto je rýchlosť vkladania a hľadania oveľa vyššia ako pri chain metóde, kde kolízie nastanú a teda oproti O(1) rýchlosti vkladania a hľadania linear\_hashu sa vytvorí linked list čo spomalí rýchlosť celého algoritmu.

## Pridávanie alternujúcich prvkov

V tejto sade testov som vkladal do štruktúr prvky veľkosti 10K,100K,200K,300K,500K,1M .

20,0,19,1,18,2,17,3,16,4,15,5,14,6,13,7

Počet prvkov som upravil oproti prvej sade testov, kvôli linear hash keďže tam môže nastať veľká časová zložitosť pri veľkých vstupoch. Alternujúco znamená, že som prvky pridával v poradí 0 , n, 1, n-1, 2, n-2....

**Predpoklad:**

Predpokladám, že situácia so stromami bude rovnaká ako v predchádzajúcej sade. Od určitej rotácie sa to bude preťažovať doľava, poprípade doľava-doprava. Z teórie vieme, že s týmito rotáciami si lepšie poradí červenočierny strom. Z dôvodu častejší rotácii doľava nám v červenočierno strome môžu znova vzniknúť situácie so strýkami, čo zrýchli vkladanie ale spomalí čas hľadania. Oproti prvému testu očakávam o niečo horšie časové výsledky z dôvodu, že doľava-doprava rotácia je zložitejšia ako len doľava rotácia. Hash očakávam rýchlejší od stromu ale pomalší od prvého testu. Problém nastáva preto, že v určitom momente sa postupnosť od 0 a postupnosť od n stretnú a ďalšie pridania určite vytvoria kolízie čo môže spomaliť pridávanie dokým nespravím resize. Situácia sa zopakuje keď ani resize tabuľka nepostačí a zase sa budú na určitý počet pridaní riešiť kolízie. V linear hashingu nastane ale väčší problém. Hodnoty sa k sebe budú stále približovať až sa v určitom momente spoja a ak príde ďalšia hodnota vznikne kolízia a bude hľadať najbližšie miesto čo bude m(počet pridaných)/2 ďalej od zahashovanej pozície a to hľadanie voľného sa stane na m/2 prvkoch lineárnym čím sa algoritmus značne spomalí. Z tohto dôvodu by malo aj pomalšie hľadať ako ostatné algoritmi.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 10K | 100K | 200K | 300K | 500K | 1M |
| AVL strom | 0,002314 | 0,026921 | 0,070646 | 0,083196 | 0,143459 | 0,29001 |
| Červeno-čierny strom | 0,004774 | 0,018435 | 0,053732 | 0,058877 | 0,101032 | 0,209375 |
| Hashovanie chain | 0,002782 | 0,017669 | 0,046905 | 0,044435 | 0,079465 | 0,164785 |
| Linear search | 0,076654 | 3,708384 | 15,038749 | 58,464678 | 2,378019 | 9,510198 |

V grafe nie je linear seach aby sa tam nestratili všetky hodnoty ostatné hodnoty.

Výsledky merania:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| AVL strom | 0,000626 | 0,007483 | 0,015457 | 0,022752 | 0,039288 | 0,033401 |
| Červeno-čierny strom | 0,000732 | 0,008109 | 0,016772 | 0,025886 | 0,04288 | 0,026597 |
| Hashovanie chain | 0,000354 | 0,003743 | 0,007767 | 0,012738 | 0,023018 | 0,016408 |
| Linear search | 0,000072 | 0,000777 | 0,001633 | 0,002329 | 0,004012 | 3,668929 |

**Výsledky merania:**

Predpoklad sa v stromoch naplnil, aj v inserte tabuliek. Hash je oproti predchádzajúcej sade pomalší v inserte ale hľadá rovnako rýchlo. V linear hashingu vznikli tie časté kolízie čo spôsobilo taký dlhý čas insertu, avšak hľadanie bolo rýchlejšie ako som predpokladal a dokonca rýchlejšie ako v predchádzajúcej sade.

**Doplnený predpoklad:**

Dlhý čas na insert v linear hashingu nemusí odrážať reálny stav tabuľky na konci pridávania, resp. ku koncu pridávania môže byť práve zväčená a preto search bude znova veľmi rýchly. Spravím preto test insert & search aby som zistil, či je ten čas iný ako keď najprv spravím insert a potom search.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| AVL strom | 0,002874 | 0,032648 | 0,065773 | 0,101798 | 0,252188 | 0,370915 |
| Červeno-čierny strom | 0,002296 | 0,029612 | 0,058592 | 0,091879 | 0,204593 | 0,336478 |
| Hashovanie chain | 0,001742 | 0,017052 | 0,033181 | 0,058588 | 0,147738 | 0,228011 |
| Linear search | 0,072909 | 7,125622 | 28,898225 | 98,025069 | 4,769951 | 18,900026 |

**Výsledky merania a záver:**

Search v linear hashingu trval 2-krát toľko ako insert. Je to z toho dôvodu, že každá kolízia vlastne zaberie 2-krát toľko času, raz na vloženie a následne navyhľadanie. Potvrdil sa teda predpoklad, že času bude iný ako pri search teste. Znamená to, že search je práve preto rýchlejší, že tabuľka ku koncu pridávania sa zväčší a nevzniknú žiadne kolízie, ktoré by spomaľovali hľadanie.

## Pridávanie pseudonáhodných prvkov

V tejto sade som vygeneroval pole veľké 20M prvkov, psedonáhodných unikátnych čísel. Postupne pridávam z tohto poľa 10K, 100K, 1M,5M a 10M prvkov. V tomto teste som aj pridal aj BV strom.

**Predpoklad:**

Pri náhodných číslach je veľmi ťažké odhadnúť ako sa budú štruktúry správať všetko ale bude závisieť od postupnosti čísel. Napriek tomu predpokladám, že pri stromoch bude insert trvať dlhšie ako pri predchádzajúcich testoch z dôvodu náhodnej postupnosti rôznych čísel pri čom vznikne veľa potencionálnych rotácii na rôznych miestach. Stále však predpokladám, že AVL bude pomalší ako červeno-čierny. Chain hash by náhodné čísla nemali veľmi ovplvyniť, kedže veľkosť tabuľky je prvočíslo a je teda pomerne obstojne zabezpečená čo najmenšia šanca kolízii + resize veľmi efektívne znova poprehadzuje prvky. Linear hash by tak isto nemal mať až taký veľký rýchlostný pokles, avšak predpokladám, že bude pomalší ako pri lineárnych testoch. Hash bude znova rýchlejší od stromov.

**Rozhodnutie pridať aj BVS :** v prechádzajúcich sadách testov by pri pridavaní prvkov pri BVS vznikala lineárne postupnosť a preto časy algoritmu boli veľmi vysoké, v tomto prípade rozdielne pridávanie prvkov do BVS strom pasívne „vyrovná“ čím nevznikne taká veľká hĺbka ako predtým a protim strom ušetrí čas na rotáciach, čím by sa vkladanie mohlo stať spomedzi stromov najrýchlejšie ale hľadanie najpomalšie.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10K | 100K | 1M | 5M | 10M |
| BVS strom | 0,001564 | 0,023291 | 0,603902 | 4,944864 | 12,22747 |
| AVL strom | 0,002462 | 0,03478 | 0,796335 | 6,406147 | 13,723278 |
| Červeno-čierny strom | 0,001779 | 0,025414 | 0,66319 | 5,349139 | 10,668219 |
| Hashovanie chain | 0,001637 | 0,024946 | 0,408697 | 2,096918 | 3,951735 |
| Linear search | 0,001752 | 0,024598 | 0,305428 | 1,572543 | 3,40777 |

**Výsledky merania:**

BVS strom má do 5M miliónov prvkov najrýchlejší insert lebo je pomerne dobre vývážený a nestráca čas na rotáciach. Avšak po 5 miliónoch prvkov jeho efektivita klesá, pretože jeho málo vývážený a niekde vzniká v jeho vetvách lineárnosť a už mu nechodia také pekné čísla aby sa pasívne a efektívne správne vyvažoval. Preto ho po 5M prvkov prekonáva pri mojej postupnosti červeno-čierny strom, ktorý je kontrolovane vyvážený a bude si udržiavať stály časový rast, narozdiel od BVS, ktorý bude stále a stále viac a viac pomalší. Hash spĺňa začiatočné predpoklady avšak od 5M prvok sa znižuje hranica medzi linear a chain metódou z dôvodu častejší kolízii v linear hashi.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BVS strom | 0,00105 | 0,019308 | 0,463077 | 4,312503 | 10,22148 |
| AVL strom | 0,000941 | 0,018138 | 0,436957 | 3,673138 | 8,662374 |
| Červeno-čierny strom | 0,001197 | 0,018106 | 0,514101 | 3,974293 | 9,571441 |
| Hashovanie chain | 0,00021 | 0,00296 | 0,064226 | 0,362291 | 0,679691 |
| Linear search | 0,000244 | 0,002415 | 0,060276 | 0,364991 | 1,059648 |

**Výsledky merania:**

Výsledky sú podobné s predpokladom. BVS strom má s pomedzi stromov najhoršie vyhľadávanie, aj keď nie je to taký drastický rozdiel čo svedčí o tom pasívnom vyrovnávaní BVS stromu. Pri AVL a červeno-čiernom platí to isté ako pri minulých testoch AVL > červeno-čierny. Pri hashi platí to isté čo pri vkladaní, častejšie kolízie v linear hashi spôsobujú aj pomalšie hľadanie oproti chain metóde. Hľadanie v chain metóde je v tomto prípade rýchlejšie ako linear, napriek tomu, že vkladanie bolo naopak. Je to z toho dôvodu, že vygenerovaná postupnosť vytvorila lineárnu postupnosť v linear hash tabuľke, ktorá sa musí často krát prechádzať ak sa prvok čo chcem nájsť nachádza v tom zhluku. V tomto má chain metóda výhodu. Buď vznikne jedna dlhá postupnosť na nejakom mieste tabuľky(čo pri prvočíselnej veĺkosti tabuľky nie je až tak pravdepodobné) alebo viacero menších a nehrozí riziko, že sa spoja a vytvoria jednu dlhú. Toto riziko avšak nastáva pri linear hashi a zvyšuje sa s každým pridaním prvku. Vo vygenerovaj postupnosti sa teda vytvoril lineárny zhluk čo malo za následok celkové spomalenie vyhľadávania.

## Všeobecný záver testov

Na testoch som porovnal rýchlosti rôznych algoritmov na dynamické dátové śtruktúry a zistil som ich potencionálne výhody a nevýhody. Čo sa týka stromov, na hľadanie prvkov strome je lepší AVL vyhľadávací strom. Jeho maximálne časové vyhľadávanie je 1,44logn, kde by červeno-čiernom strome to môže byť 2logn. BVS bez vyvažovania je veľmi rýchly na vkladaní náhodných prvkov avšak následne stráca v hľadaní. Všeobecne teda pri stromoch platí, že zvýšená rýchlosť pri vyhľadávaní ide na časový úkor vytvorenia danej štruktúry.

Čo sa týka hashu a stromov, hash je oveľa rýchlejšia dátová štruktúra lebo rýchlosť pridávania a hľadania mimo kolízie je O(1) a vrámci kolízii najhorší scenár pri chain hashu to je O(1+logm) kde m je dĺžka reťaze na ľubovoľnej pozícii hash tabuľky. Chain metóda je ďalej všestranejšia ako linear hashing práve z dôvodu kolízii(pozri 2.4 a 2.5). Linear hashing môže pri nevhodnej kombinácii vstupných kľúčov stratiť výhodu svojej rýchlosti a stať sa veľmi pomalým algoritmom čo sa týka pridávania prvkov. Avšak pri menších postupnostiach a pri hľadaní poskytuje vyššiu časovú efektivitu efektivitu. Hash tabuľky sú ďalej ovplyvnené podmienkou kedy budem robiť zväčšenie. Častejšie zväčšenia zvýšia rýchlosť vyhľadávania ale popritom zvýšia pamäťovú a časovú náročnosť pri vkladaní. Preto treba nájsť správne vyváženie medzi rýchlosťami vkladania, hľadania a pamäťovej zložitosti. Záleží preto od toho, aké údaje vkladáme s akým kľúčom a na akom aspekte nám záleží, podľa týchto kritérii je potom vhodné zobrať dátovú štruktúru a podľa potreby ju upraviť aby vyhovovala naším požiadávkam.