Dátové štruktúry a algoritmy

Zadanie 2: binárne vyhľadávacie stromy a hashovacie tabuľky

Peter Plevko

FIIT STU

2020

Peter Plevko

AVL strom uvedenie do tematiky

AVL strom v informatike je údajová štruktúra, prvý vynájdený samovyvažovaci binárny vyhľadávací strom.

V avl strome sa pre každý uzol rozdiel výšky dvoch podstromov líši najviac o jednotku, preto je aj známy ako výškovo vyvážený. Pridávanie a mazanie môže vyžadovať vyváženie stromu jednou alebo viacerými rotáciami stromu.

Koeficient vyváženia uzla je výška jeho pravého podstromu mínus výška jeho ľavého podstromu. Uzol s koeficientom vyváženie 1,0 alebo -1 sa považuje za vyvážený. Uzol s koeficientom vyváženie -2 alebo 2 sa považuje za nevyvážený a vyžaduje vyváženie stromu.

operácia	priemer	Worst case
vyhľadanie	O(log n)	O(log n)
insert	O(log n)	O(log n)

Takto vyzerá struct môjho avl stromu

```
typedef struct vrchol
{
    int hodnota;
    struct vrchol *lavy;
    struct vrchol *pravy;
    int vyska;
}VRCHOL;
```

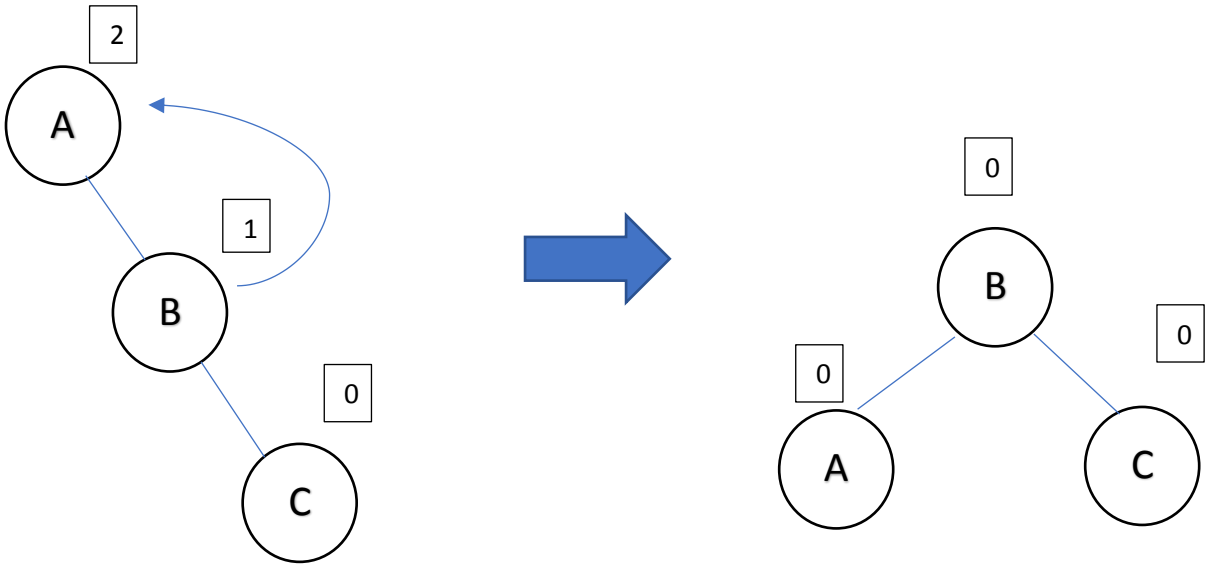
Takýto struct zaberá 32 bajtov

Peter Plevko

Toto vyvažovanie spočíva vo štyroch rôznych rotáciách týmito rotáciami sú:

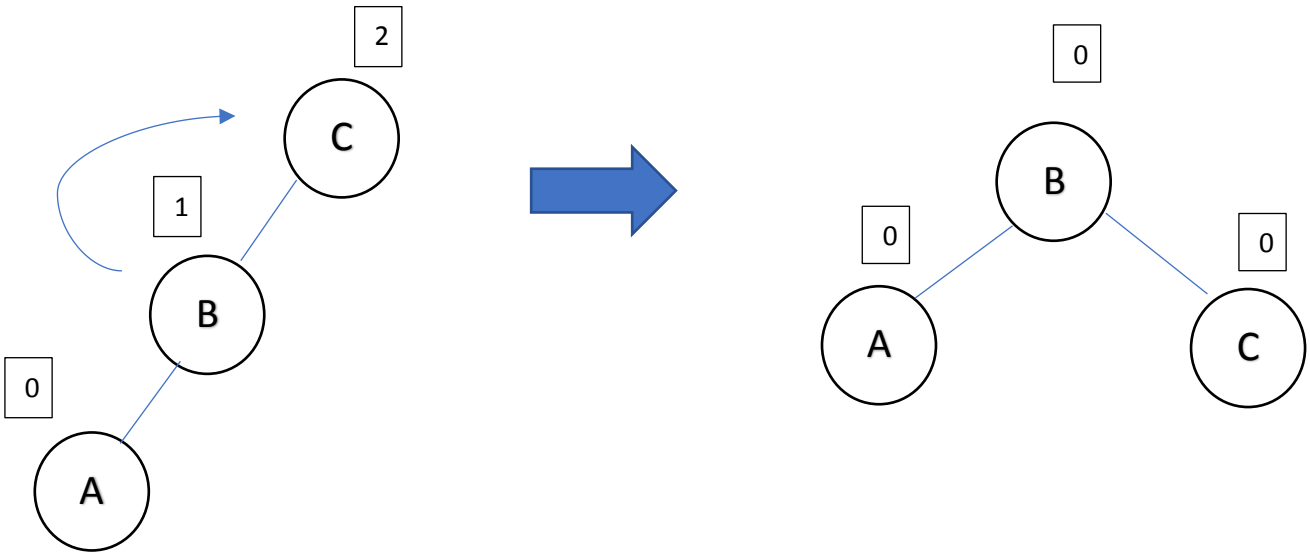
Ak sa strom stane nevybalansovaný koreň pravypodstrom pravypodstrom prevedieme rotáciu doľava

Left rotation



Ak sa strom stane nevybalansovaný koreň lavypodstrom lavypodstrom prevedieme rotáciu doprava

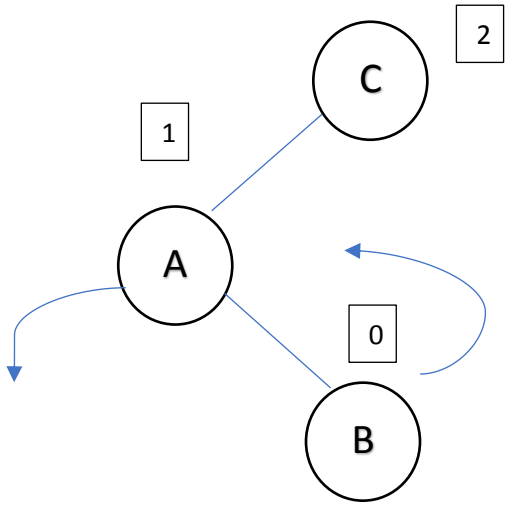
Right rotation

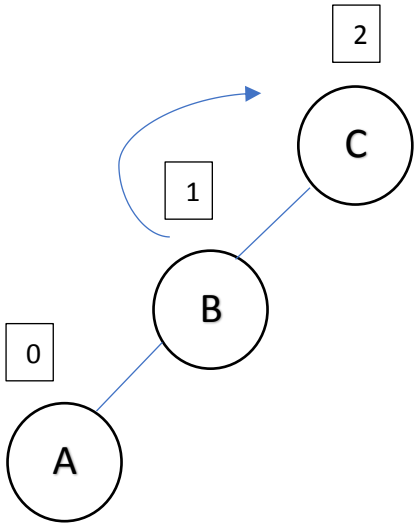


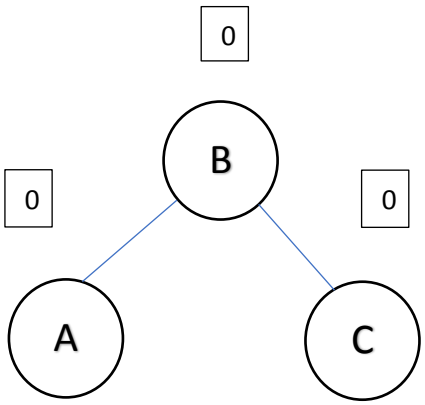
Peter Plevko

Left-right rotation

Dvojité rotácie sú trosku zložitejšie verzie dvoch predošlých rotácii. Left right rotation je kombinácia ľavej rotácie ktorá je nasledovaná pravou rotáciou.



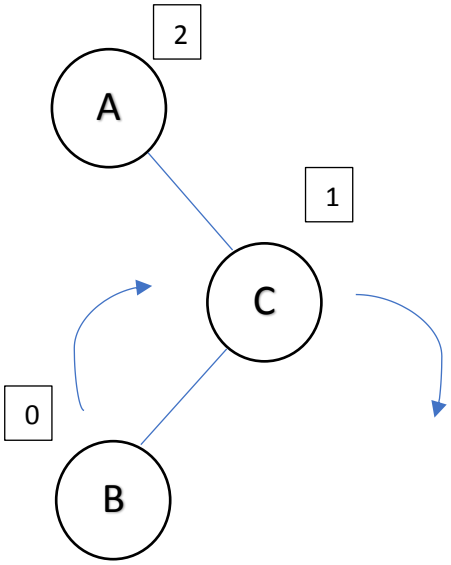


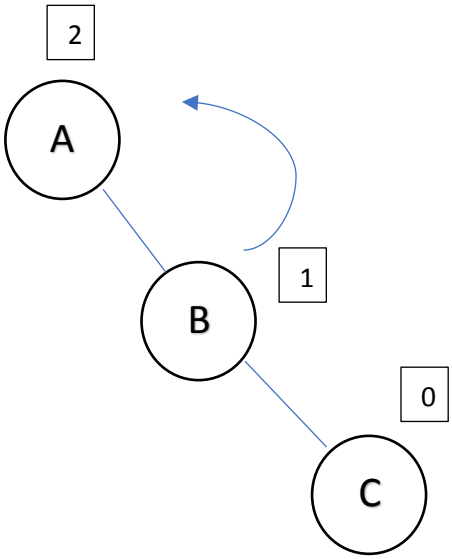


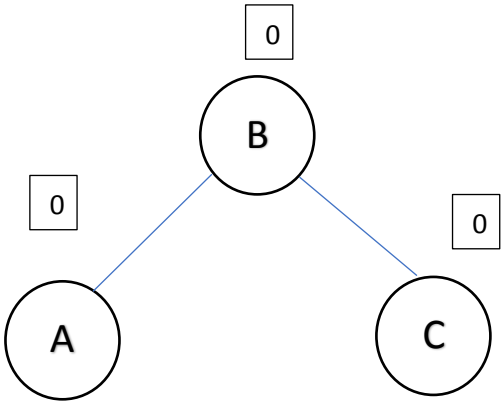
Peter Plevko

Right-left rotation

Right-left rotation je kombinácia pravej rotácie ktorá je nasledovaná ľavou.







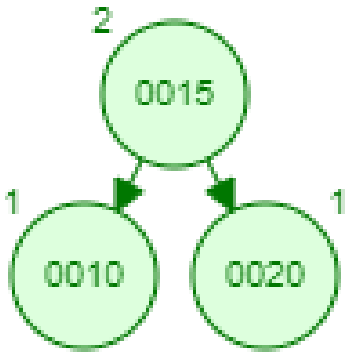
Peter Plevko

Otestovanie správnosti môjho riešenia:

Left rotation

vložím prvky 10 15 20

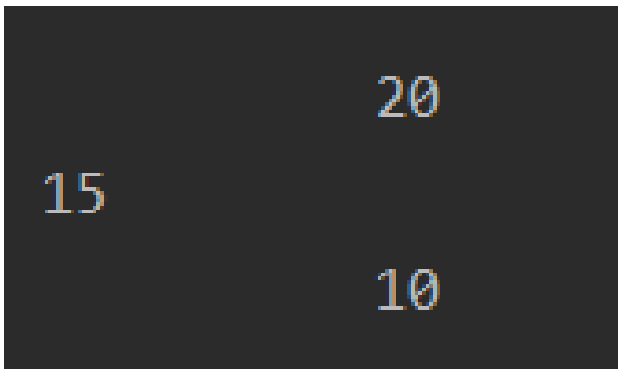
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto:



Strom pred vybalansovaním

V tejto chvíli sa strom stal nevybalansovaní pretože výška koreňa je 2, musí nastáť rotácia

Môj strom po vykonaní rotácie, výsledok left rotáciu mi vykonalo správne

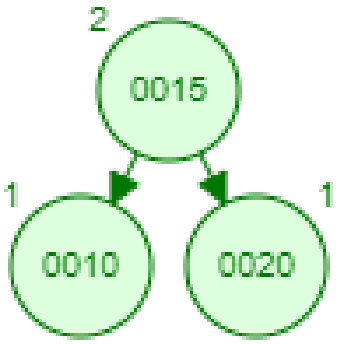


Peter Plevko

Right rotation

vložím prvky 20 15 10

Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Strom pred vybalansovaním

V tejto chvíli sa strom stal nevybalansovaní pretože výška koreňa je 2, musí nastáť rotácia

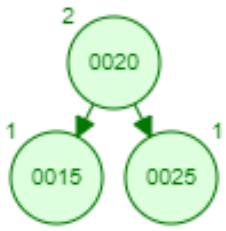
Môj strom po vykonaní rotácie vyzerá takto, right rotation mi vykonalo správne

Peter Plevko

Right left rotation

Vložím prvky 15 25 20

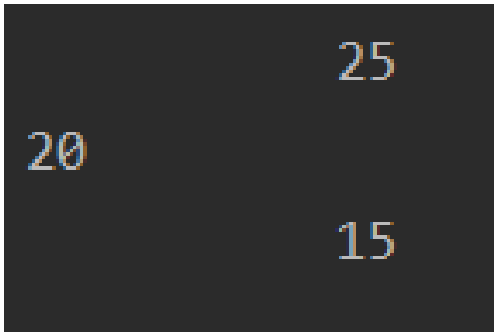
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Strom pred vybalansovaním

Stav po prvej rotácii

Finálny strom



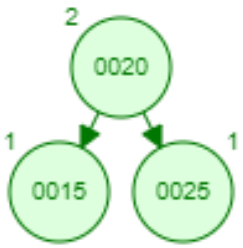
Right left mi vykonalo správne

Peter Plevko

Left right rotation

Vložím prvky 25 15 20

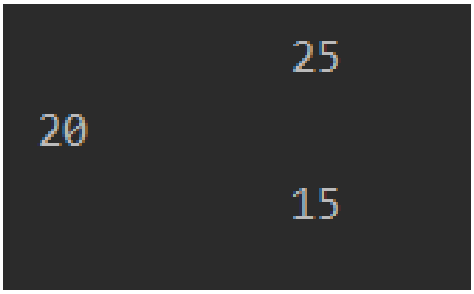
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Strom pred vybalansovaním

Stav stromu po prvej rotácii

Stav finálneho stromu

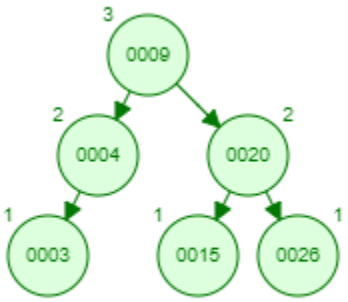


Left right mi vykonalo správne

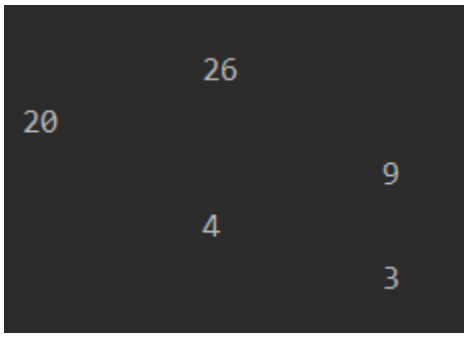
Peter Plevko

Zložitejšie testy:

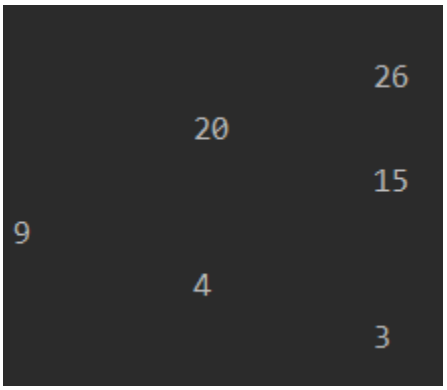
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Stav stromu pred vložením



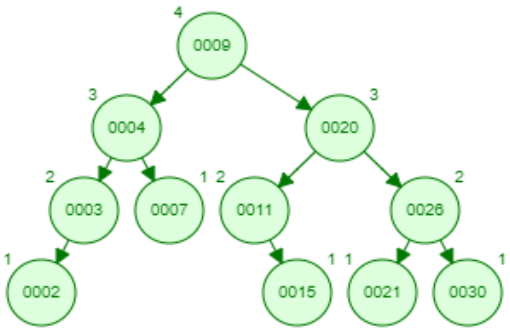
Vložím číslo 15



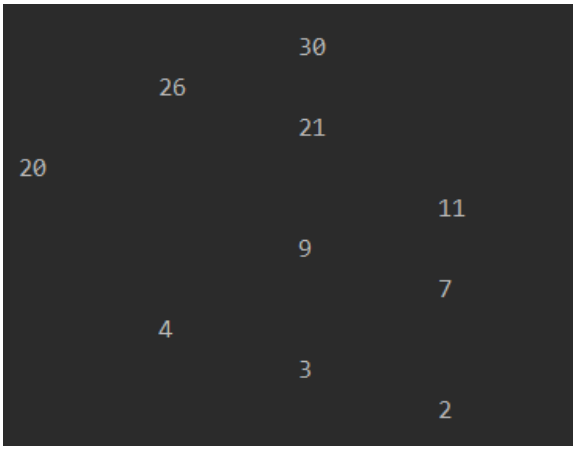
Vykonal sa správne

Peter Plevko

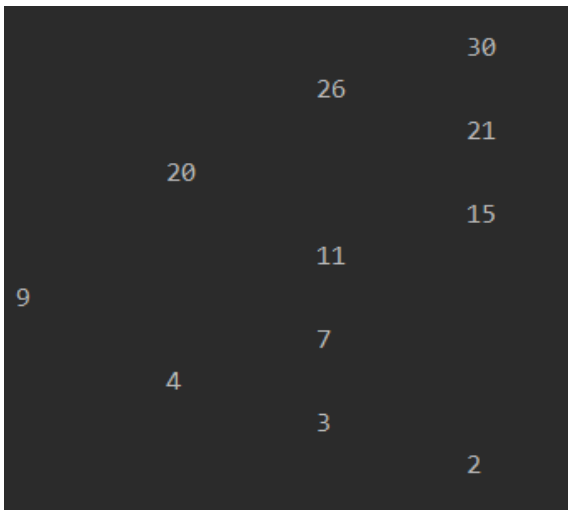
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Stav stromu pred vložením



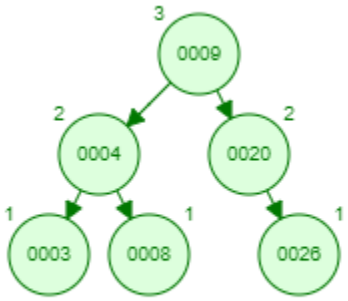
Vložím 15



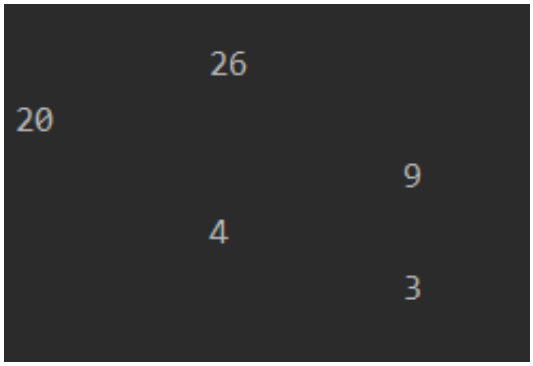
Vykonal sa správne

Peter Plevko

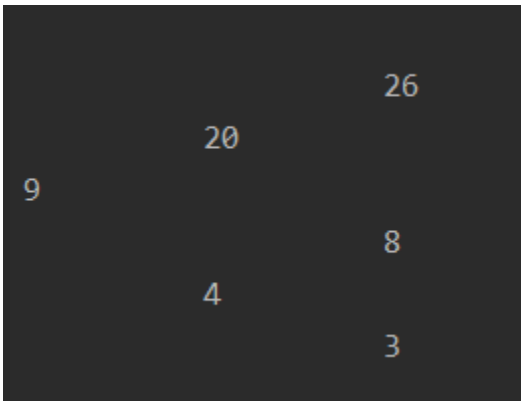
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Stav stromu pred vložením



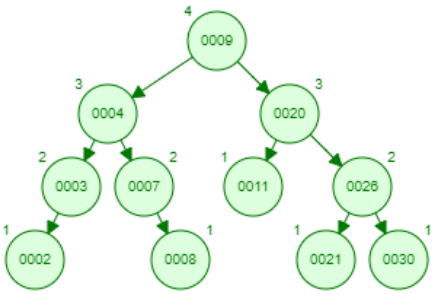
Vložím 8



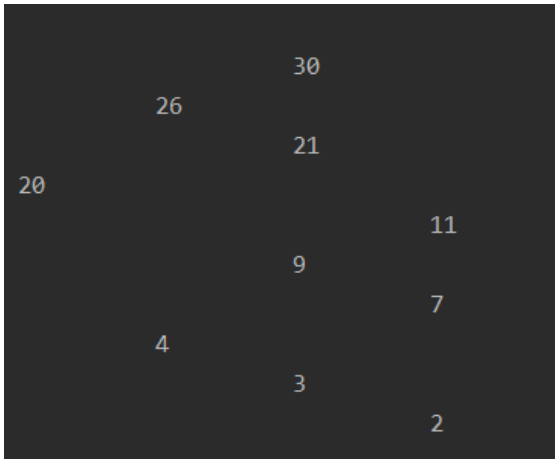
Vykonal sa správne

Peter Plevko

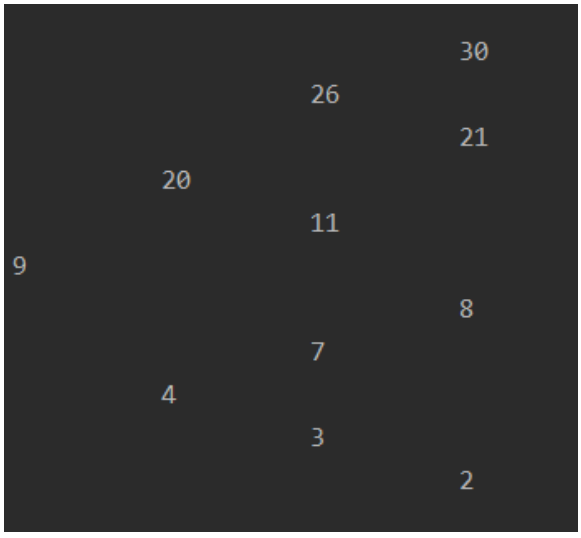
Avl strom po vložení prvku a jeho vybalansovaní by mal vyzerať takto



Strom pred vložením



Vložím 8



Vykonal sa správne

Peter Plevko

Uvedenie do témy RB strom:

Je to druh samovyvažovaci binárneho stromu. Každý uzol ma v sebe atribút ktorým je farba. Tieto farby sa používajú aby strom po vložení prvku ostal vybalansovaný. Balans stromu je zachovaný nafarbením stromu pomocou dvoch farieb v takom štýle že si strom zachováva určité vlastnosti.

Vlastnosti:

1: každý uzol je buď červený alebo čierny

2: Koreň je vždy čierny

3:kazdy list je čierny

4: ak je uzol červený obe deti sú čierne

5: Každá cesta z daného uzla k jeho listovým potomkom prechádza rovnakým počtom čiernych uzlov.

operácia	priemer	Worst case
vyhľadanie	O(log n)	O(log n)
insert	O(log n)	O(log n)

Takto vyzerá struct red black stromu

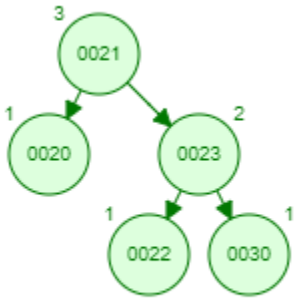
```
enum nodeColor
{
    RED,
    BLACK
};

struct rbNode
{
    int data, color;
    struct rbNode *link[2];
};
```

Takýto struct zaberá 24 bajtov takže rb je pamäťovo efektívnejší

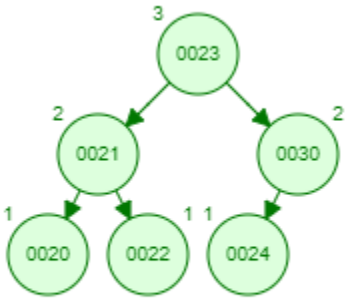
Peter Plevko

Predpoklady ktoré chcem zistiť

Po prečítaní si problematiky a zoznámení sa z fungovaním týchto algoritmov som vyvodil nasledovne závery AVL strom by mal byt rýchlejší vo vyhľadávaní a RB v inserte a to s toho dôvodu že AVL strom rieši preťaženie vždy rotáciami čo je dosť náročná operácia a RB strom to niekedy rieši iba jednoduchým prefarbením uzlov napríklad v tomto prípade

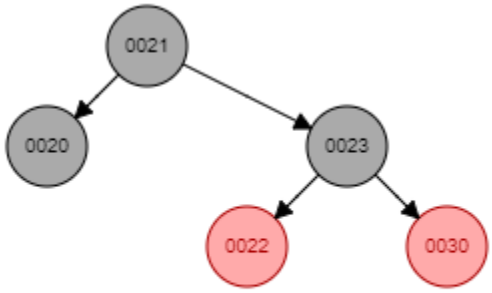


Po vložení prvku 24 nastane rotácia

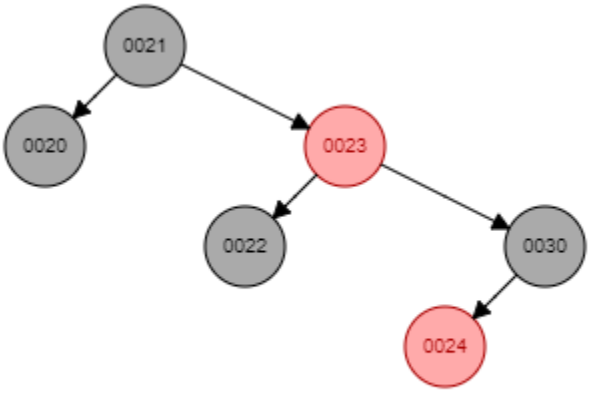


Pričom red black sa sprava rozdielne

Peter Plevko



Teraz po vložení čísla 24 nastane iba prefarbenie nie rotácia



V tejto chvíli je vidno že insert do avl stromu bol náročnejší ako do rb stromu ale je aj vidno že rb strom je menej vyvážený ako ten avl čo znamená že search bude v rb strome náročnejší ako v avl

S tejto logiky znamená že avl strom by sa mal používať najme keď s prvkami často nehýbem iba ich prezerám. Rb zase naopak v takých kde s údajmi často pohybujem. Rb strom je následne aj pamäťovo lepší pretože zaberá menej pamäte.

Peter Plevko

Moje testy

V zadaní mame porovnať rýchlosti rôznych dátových štruktúr, ja konkrétne som si vybral AVL strom a RB strom stiahnutý z internetu a následne som si vybral chaining metódu hashovacej tabuľky a tu porovnávam s lineár hashingom z internetu.

Zložitosť a rýchlosť týchto algoritmov budem porovnávať na rôznych vstupoch ktoré som si pripravil týmito vstupmi sú

1: od 0 po N

2: od N po 0

3: Small N big N respektíve 0 N 1 N-1

4: štvrtý test je random N

Tento random test však potrebujem mat pre každý algoritmus rovnaký to znamená že mam vytvorené pole random čísel podľa srand(42) to znamená že mam náhodne pole ale som vždy schopný si tuto náhodnú postupnosť zopakovať a random číslo ďalej generujem pomocou rand()*rand() aby som dosiahol väčší rozptyl čísel s toho dôvodu že duplikáty v žiadnej z vyššie uvedených dátových štruktúr netolerujem.

Všetky tieto vstupy mam v dynamických poliach ktoré následne vkladám do funkcii ktoré akceptujú toto pole.

```
void vlozDoAvl(int *pole)
```

```
void vlozDoBinarneho(int *pole)
```

```
void vlozDoRB(int *pole)
```

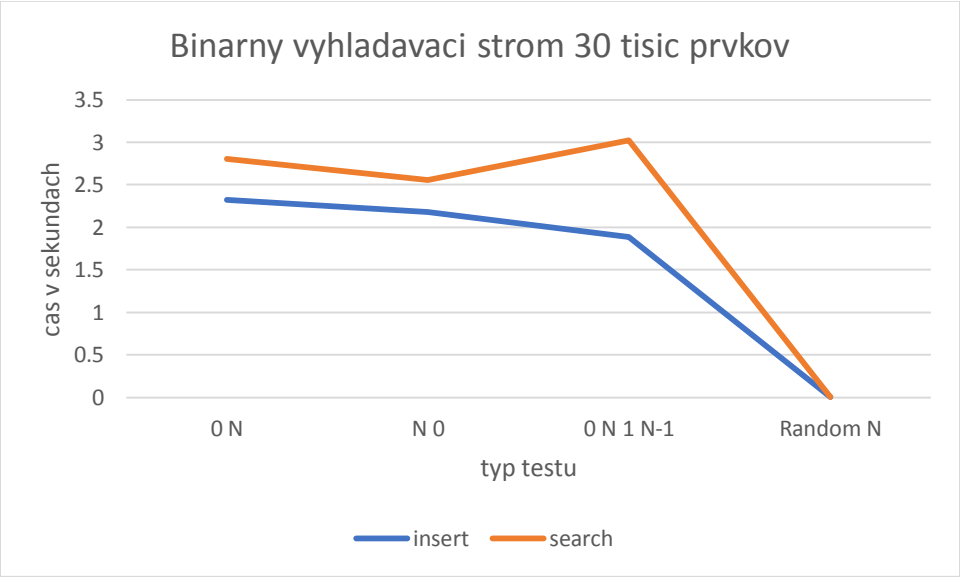
```
void vlozChainHash(int *pole)
```

```
void vlozLinearHash(int *pole)
```

V maine zavolám funkcie na vytvorenie týchto poli a potom už iba zavolám vyššie spomínané funkcie vlož takto to mam rozdelene kvôli tomu aby môj kód bol prehľadnejší a ak si niekto bude chcieť otestovať aj iné vstupy ma to jednoduché stačí iba vytvoriť pole a poslať ho do funkcie

Peter Plevko

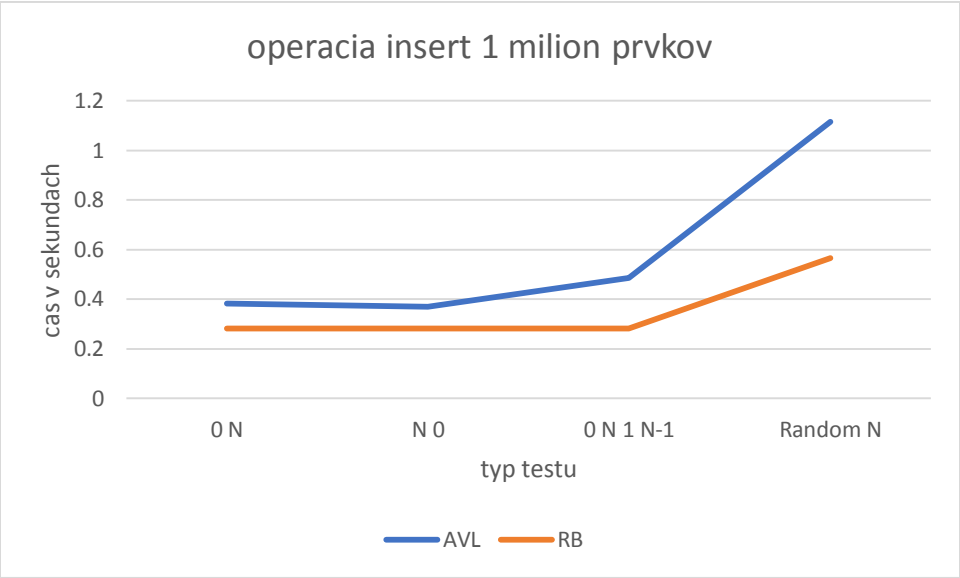
Grafy

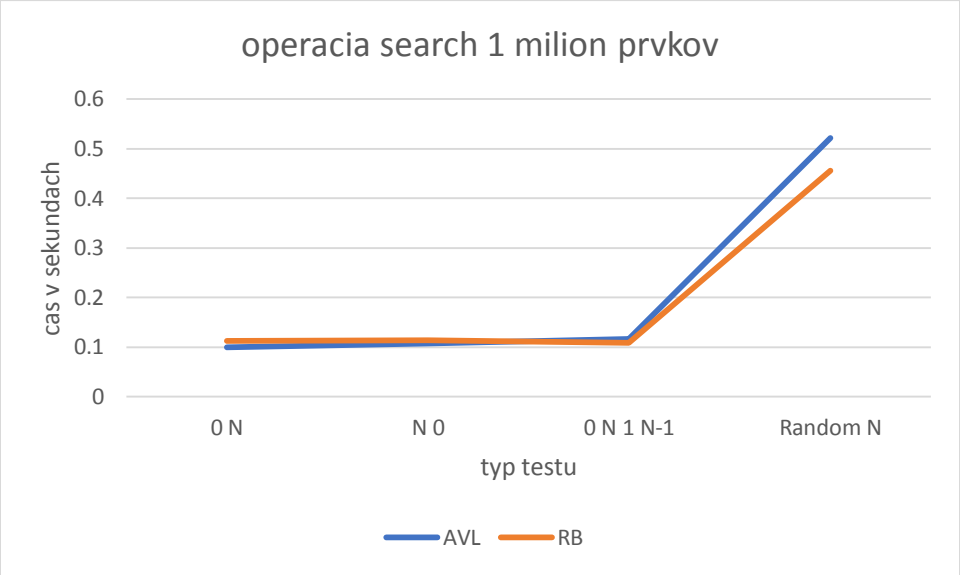


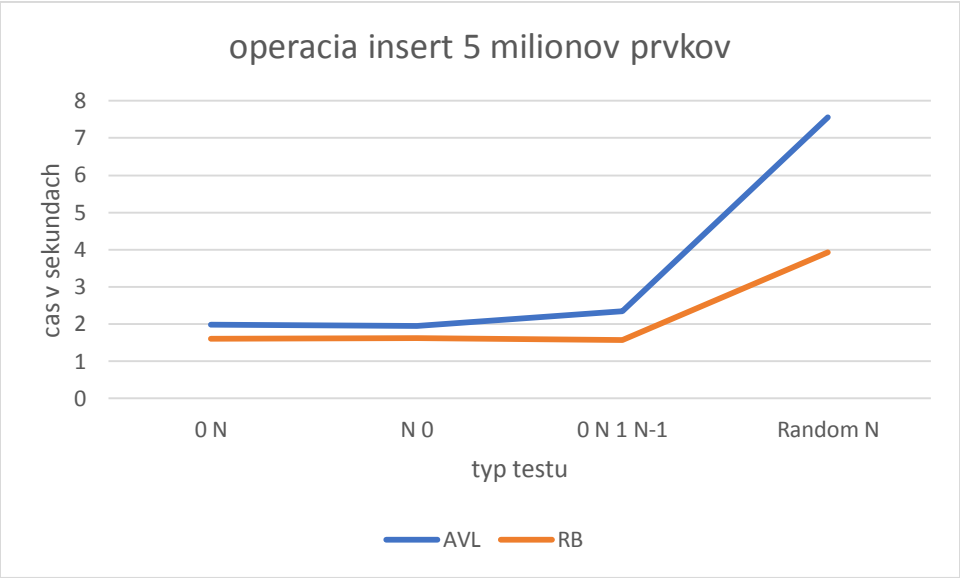
Tento graf ukazuje rýchlosť BVS.

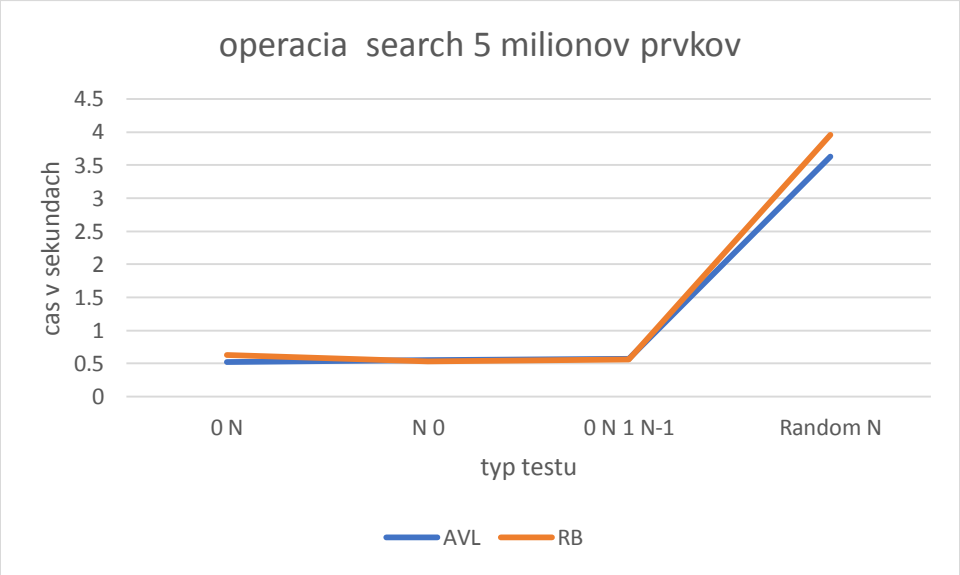
Zistil som že najrýchlejší je pri random N, je to logické pretože teoreticky v ideálnom prípade môže nastáť aj taká situácia že tento môj strom bude dokonale vyvážený.

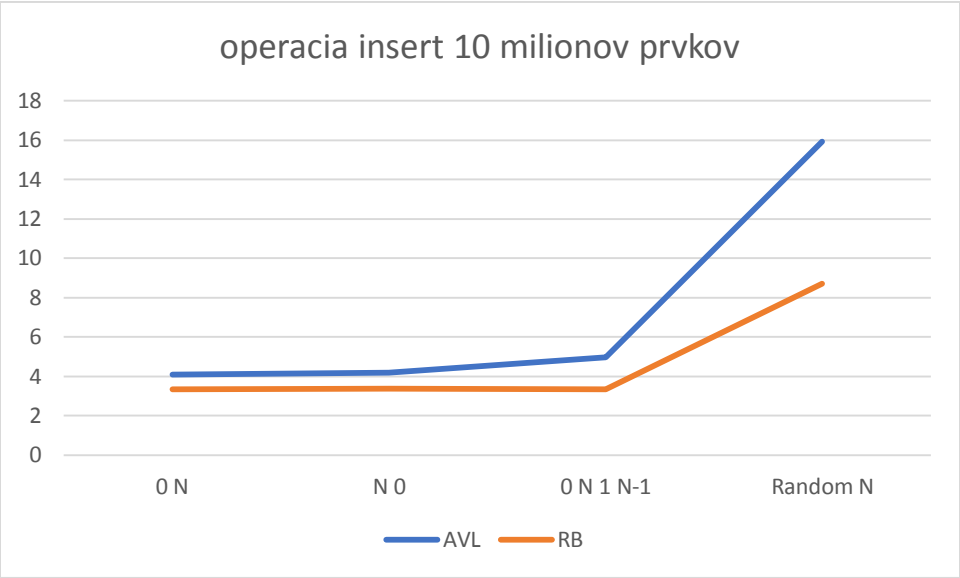
Peter Plevko

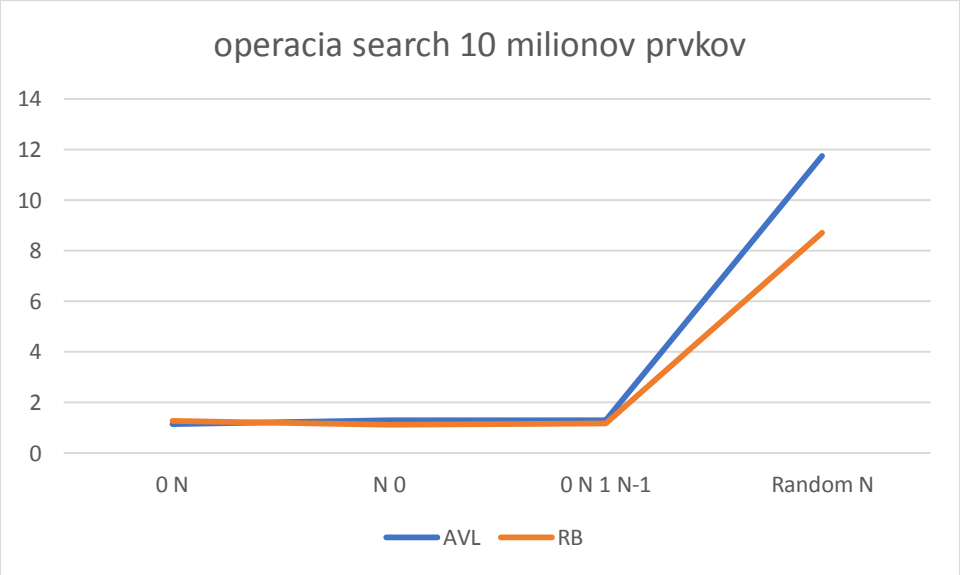

operacia insert 1 milion prvkov
cas v sekundach
1.2
1
0.8
0.6
0.4
0.2
0
0 N
N 0
0 N 1 N-1
Random N
typ testu
AVL
RB


operacia search 1 milion prvkov
cas v sekundach
0.6
0.5
0.4
0.3
0.2
0.1
0
0 N
N 0
0 N 1 N-1
Random N
typ testu
AVL
RB

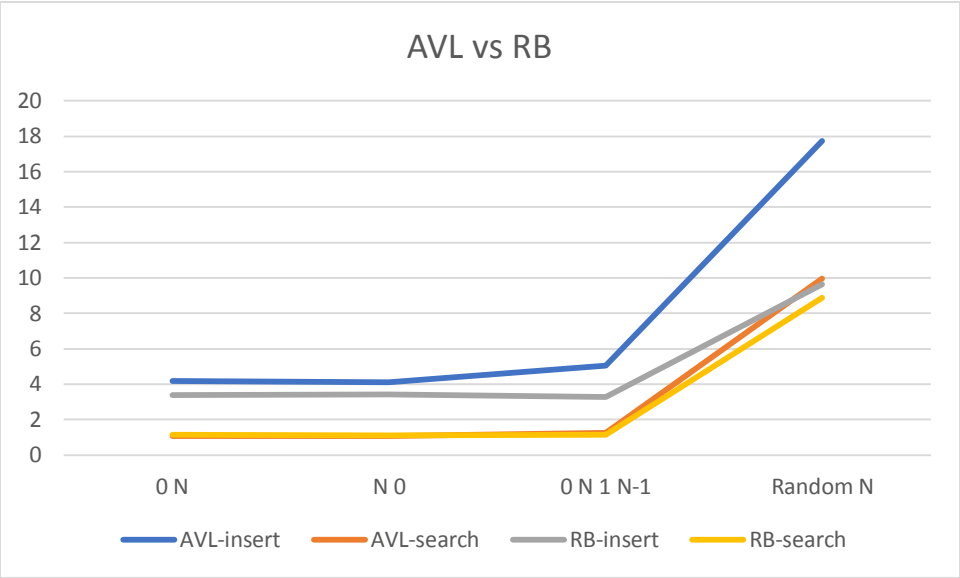

operacia insert 5 milionov prvkov
cas v sekundach
8
7
6
5
4
3
2
1
0
0 N
N 0
0 N 1 N-1
Random N
typ testu
AVL
RB


operacia search 5 milionov prvkov
cas v sekundach
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
0 N
N 0
0 N 1 N-1
Random N
typ testu
AVL
RB





V tomto grafe by malo byt logicky vidieť najväčší rozdiel medzi týmito dvomi implementácii avšak tu nie je ten rozdiel vidno ba naopak AVL je pomalsi ako, dôvod tohto bude že moja AVL implementácia nie je úplne správnou/efektívnou.



Takže na začiatku som napísal predpoklady že ako by sa mal AVL a RB správať na tomto grafe môžeme vidieť že AVL insert trvá najdlhšie zo všetkých takže môj predpoklad sa potvrdil, trvá najdlhšie pretože rotácie sú zložitejšia operácia ako prefarbenie.

Mojim ďalším predpokladom bolo že search v AVL strome by mal byt rýchlejší ako RB z grafu môžeme vidieť že to sa s časti potvrdilo, ale pri Random N je search v AVL pomalší ako RB, to znamená že moja implementácia AVL stromu nie je dokonala na jednej strane to môže byt neefektívnym algoritmom alebo nie úplne správnym vyväzovaním stromu. Tento rozdiel v rýchlostiach je však minimálny.

Peter Plevko

Hashovanie

Hashovacie tabuľka je v informatike údajová štruktúra ktorá asociuje kľúče s hodnotami. Tabuľka podporuje operácie search a insert. Hashovacie tabuľky poskytujú vyhľadávanie s konštantným časom O(1) v priemernom prípade, nezávisle od poctu položiek v tabuľke. Ale zriedkavý najhorší prípad môže byt až O(n). V porovnaní s inými údajovými štruktúrami asociatívnych poli sú hashovacie tabuľky najužitočnejšie pre uloženie veľkého množstva údajov. Hashovacie tabuľky sú oveľa rýchlejšie ako samovyvažovacie binárne vyhľadávacie stromy.

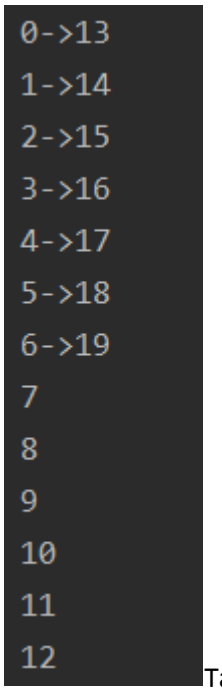
Voľba dobrej hashovacej funkcie: Dobra hashovacie funkcia ma zásadný vplyv na výkon hashovacej tabuľky. To znamená že ak hashovacia funkcia bude vracať podobne hodnoty výsledkom bude pomalé vyhľadávanie. Veľkosť poľa hashovacej tabuľky sa často volia prvočísla. Robí sa to preto aby sme sa vyhli výskytu spoločných deliteľov hashu a veľkosti tabuľky. Ktoré by boli príčinou kolízii po operácii modulo.

Riešenie kolízii ak funkcia vráti pre dva rôzne kľúče rovnaký index zodpovedajúce záznamy nemožno uložiť na to iste miesto keďže je už obsadene musíme nájsť iné miesto pre uloženie záznamu a zabezpečiť aby sme ho našli keď ho neskôr vyhľadáme.

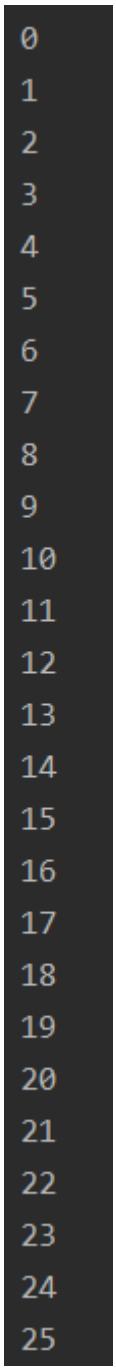
V mojej implementácii som použil techniku zreťazenia čo vlastne znamená že každé miesto v poli odkazuje na zreťazený zoznam vložených záznamov. Pre vloženie je potrebne nájsť správne miesto a pridať ho na jeden koniec.

V cudzej implementácii som použil techniku lineár hashingu pridávam do tabuľky na základe hash funkcie a kolízie riešim nie zreťazením ale pozriem sa doprava a zisťujem ci tam je voľno, čiže hľadám najbližšie voľne miesto a keď dôjdem na koniec tabuľky tak začnem skúšať znova od nuly.

Peter Plevko



Tabuľka pred resizom

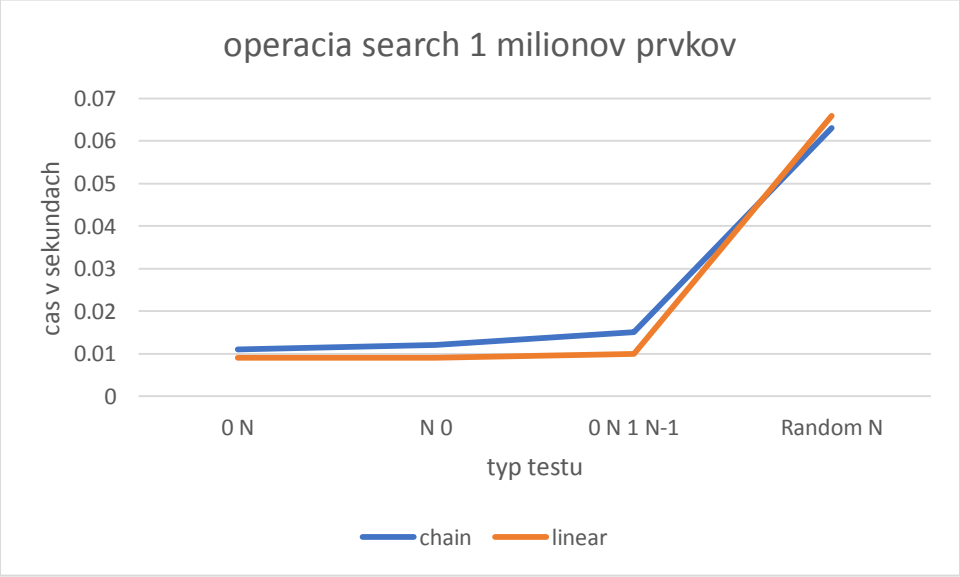
tabuľka po resize 

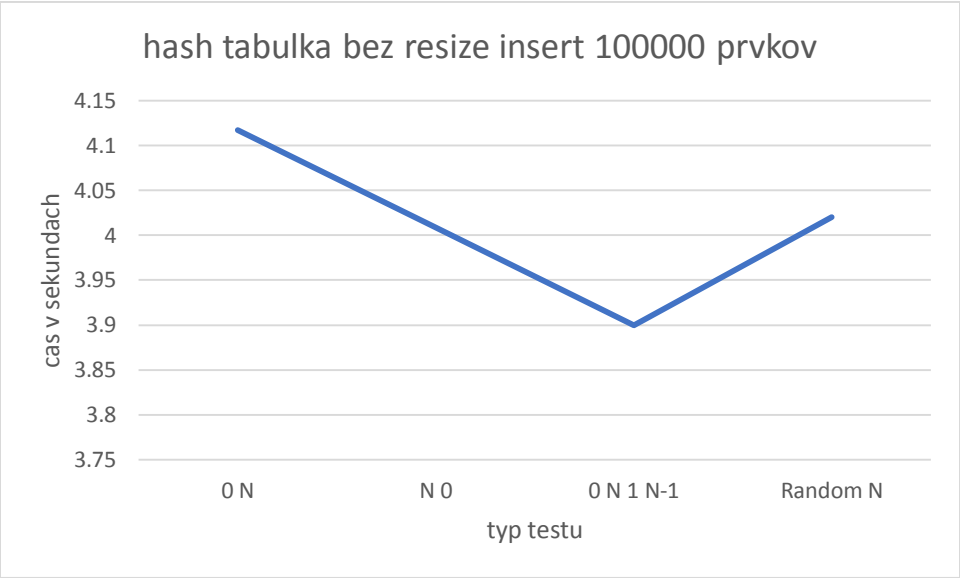
Resize tabuľky mi vykonalo správne

Peter Plevko

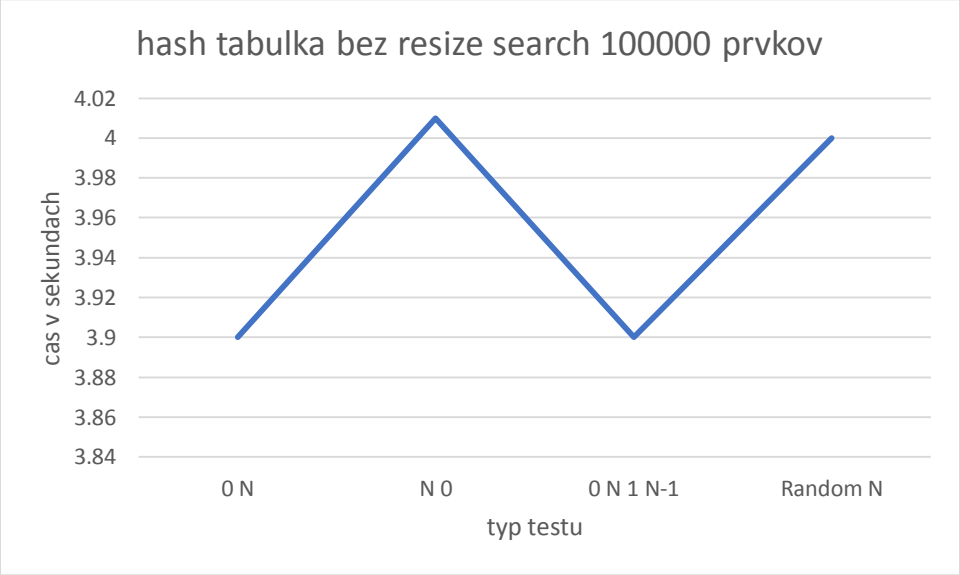
Testy lineárneho hashu pri väčších vstupoch ako 1 mil padajú a to z toho dôvodu že mi dôjde heap zásobník.

Pri alternujúcich keď vznikne kolízia v blízkosti stredu poľa a nenastane resize tak vyriešenie kolízie je že musí nájsť najbližšiu pozíciu ktorá je za nim voľná to znamená že on musí prejsť všetky veľké čísla ktoré pridal a všetky malé čísla ktoré pridal a nájsť ideálnu pozíciu. Čiže keď vznikne kolízia musí prejsť všetky prvky ktoré pridal a stáva sa veľmi neefektívnym a počítaču dôjde pamäť potrebná k výpočtu.



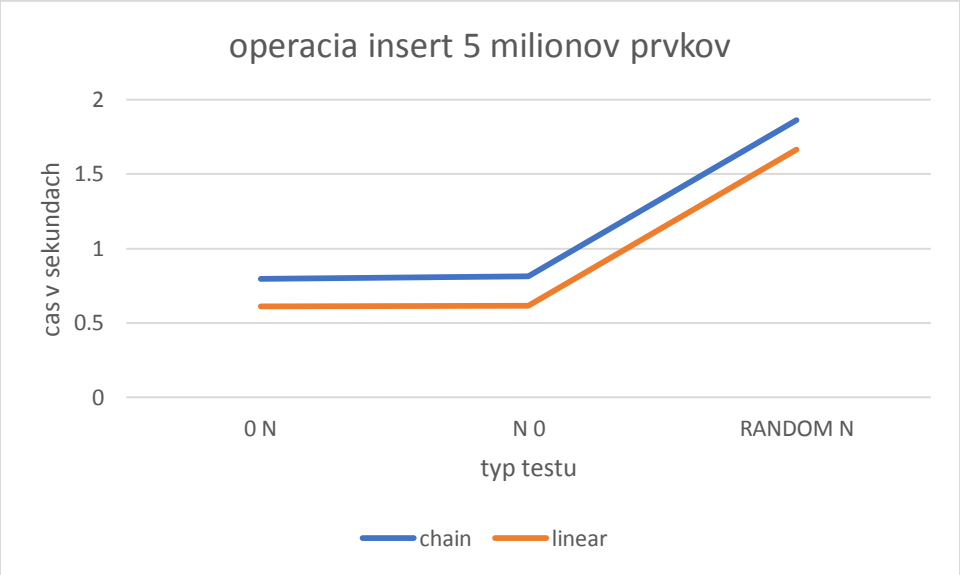


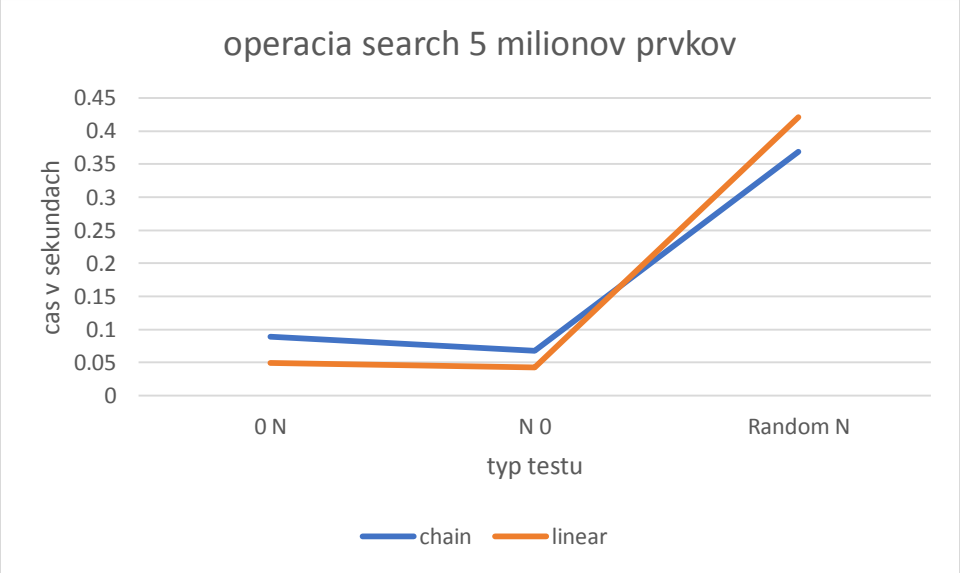
Na tomto grafe môžeme vidieť môj chain hash v ktorom som odstránil resize aby som mohol poukázať na to aký veľký rozdiel je medzi tabuľkou s funkciou resize a bez funkcie resize už pri 100000 prvkoch to trvá 4 sekundy čo je obrovské číslo oproti sekunde pri milióne prvkov s resizom dôvod tohto dlhého insertu je že mam nazačiatku daný size 13 a tento size ďalej nerozširujem to znamená že 100000 prvkov sa musí zmestiť do týchto 13 s toho vyplýva že sa mi to začne strašne moc chainovať čim sa stážuje insert napríklad na pozícii pole[0] môžem mat kľudne aj tisíc prvkov v chaine.

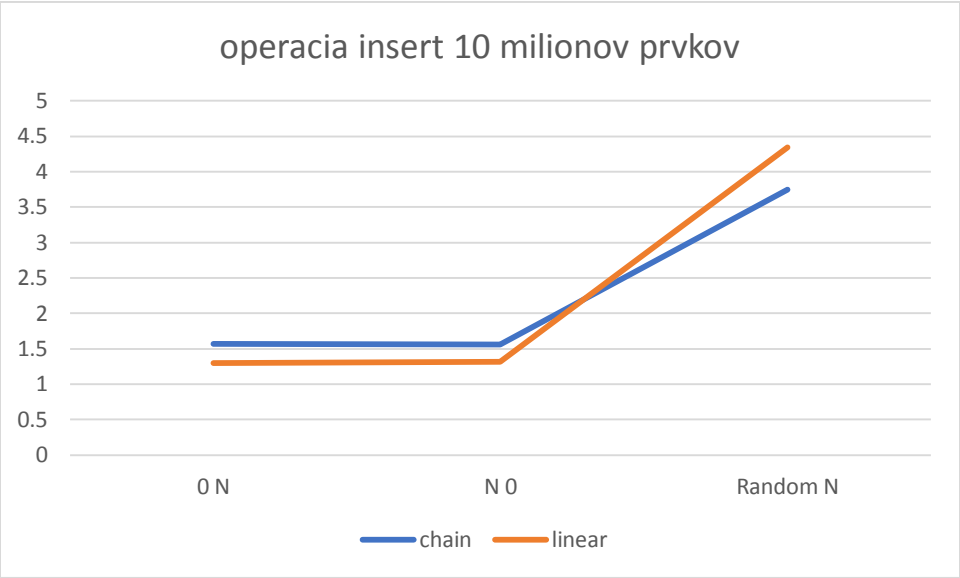


Ako na insert zrušenie resizu malo vliv aj na search a to s toho dôvodu že musím prejsť obrovské množstvo prvkov predtým než nájdem ten ktorí potrebujem. Keď si porovnáme search, insert 100000 prvkov v tabuľke bez resizu s 1000000 prvkami s resizom všimneme si aký veľký rozdiel je medzi nimi s tohto vieme odvodiť že resize je veľmi dôležitý.

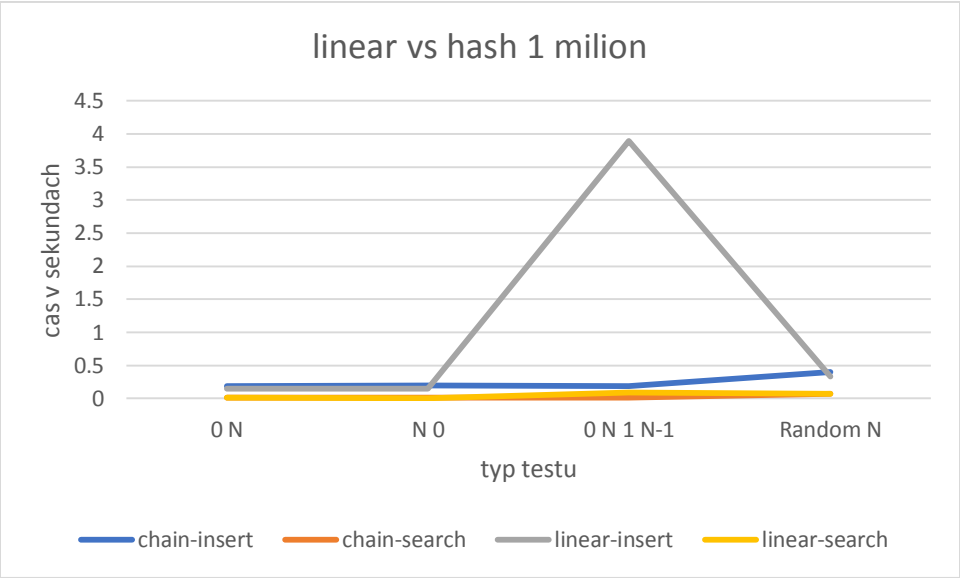
Peter Plevko

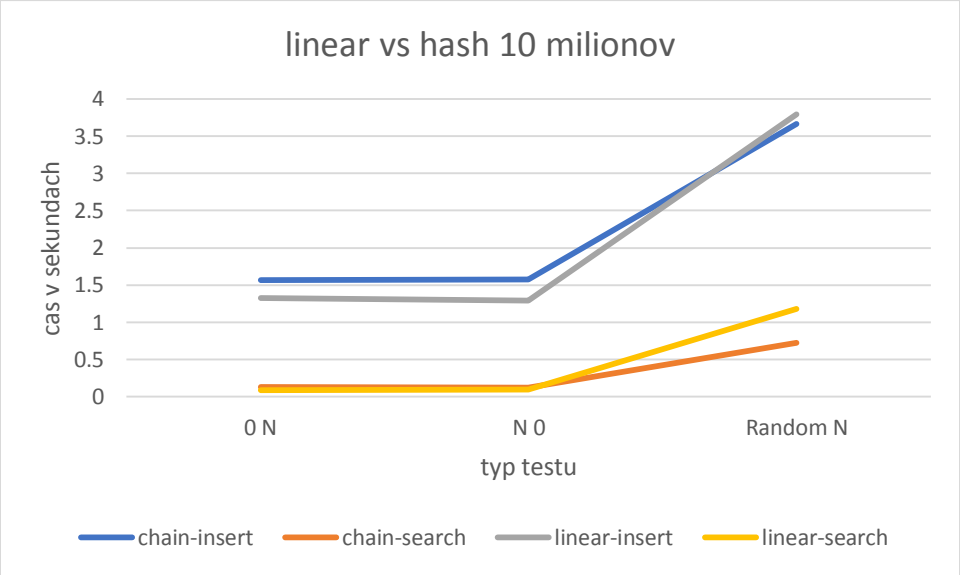

operacia insert 5 milionov prvkov
cas v sekundach
2
1.5
1
0.5
0
0 N
N 0
RANDOM N
typ testu
chain
linear


operacia search 5 milionov prvkov
cas v sekundach
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
0 N
N 0
Random N
typ testu
chain
linear



Peter Plevko





Insert by mal byt v lineár hashingu rýchlejší ale pretože resize nie je prvočíslom ale 2 krát zväčším tabuľku tak nastáva veľa kolízii a to robí tento algoritmus neefektívnym. Vyššie už som spomínal ako sa riešia kolízie v takomto prípade

V chain metóde by mal byt insert najdlhšie trvajúci a graf to potvrdil.

Lineár search by mal byt rýchlejší ale nie je, je to s dôvodu neefektívnosti prevzatého riešenia

Peter Plevko

Vyhodnotenie:

Mojou úlohou bolo porovnať rôzne dátové štruktúry na ukladanie údajov. Najprv som vyskúšal ci mnou a nasledovne aj stiahnuté implementácie s internetu fungujú. Vyskúšal som to pomocou rôznych testov z toho niektoré som aj spomenul v mojej dokumentácii. Za pomoci grafu som sa pokúsil ukázať ktoré algoritmy sú najefektívnejšie. Efektívnosť každého algoritmu sa líši od toho na akom poli som to testoval ale celkovo najzdĺhavejšou operáciou je AVL insert a to sedí. Ďalej najkratšou operáciou je lineár search toto tiež graf potvrdil.

Čo sa týka stromov na hľadanie prvkov v binárnom strome je veľmi rýchle vkladanie náhodných prvkov ale stráca rýchlosť pri vyhľadávaní. Hash je oveľa rýchlejšia dátová štruktúra lebo rýchlosť hľadania a pridávania je rýchlosť o(1). Chain metóda je všestrannejšia kvôli kolíziám. Lineár hashing môže stratiť výhodu svojej rýchlosti avšak pri hľadaní poskytuje väčšiu časovú aktivitu. Tabuľky sú ďalej ovplyvnene veľkosťou tabuľky čim častejšie zväčšujem tabuľku tým je search rýchlejší ale na úkor toho je insert dlhší preto treba nájsť správnu rovnováhu kedy robiť resize. Strom je vždy log(n) kde kto tabuľka sa môže pri niektorých situáciách úplne pokaziť a stáť sa neefektívnou.

Zistil som:

1: Hashovanie je rýchlejšie oproti stromom.

2: Stromy sú stabilnejšie dátové štruktúry vždy si budú držať to log(n) pričom pri hash tabuľkách nemám tu istotu, pri hash zbytočne používam miesto ktoré nepotrebujem a môžem stratiť rýchlosť keď mi prídu nevhodne údaje, pri stromoch nevhodne údaje neexistujú preto sú stromy bezpečnejšie a všestrannejšie ako hash tabuľky.

3: Čim rýchlejší insert tým pomalší search toto platí samozrejme aj naopak čim pomalší insert tým rýchlejší search, pri hash tabuľke som zistil že je veľmi doležíte v ktorej fáze sa nachádza hash tabuľka, či sa nachádza pred alebo po resize, pretože po resize je oveľa efektívnejšia a oveľa rýchlejšia ako pred resizom a to mi vie ovplyvniť porovnávania.

4: Nezáleží len na objeme údajov respektíve kľúčov ale aj na tom v akom poradí prišli a aké majú samostatne vlastnosti väčšie, menšie, alternujúce lineárne, to že mam niekde väčší objem údajov neznamená že to teda pôjde aj pomalšie. Nie hlavný faktor je objem údajov ale to ako ich mam uložene.

Peter Plevko

Zdroje:

RB strom: https://www.programiz.com/dsa/red-black-tree

Hash table: https://www.tutorialspoint.com/data_structures_algorithms/hash_table_program_in_c.htm

Tieto stiahnuté implementácie dátových štruktúr som kvôli svojim potrebám mierne pozmenil.

Peter Plevko