# Zadanie č.1*

Hlib Kokin ID: 117991
Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
xkokin@stuba.sk

20.03.2022

*Prve zadanie v predmete Dátové štruktúry a algoritmy, ak. rok 2021/22, vedenie: Peter Lehoczký

**Abstrakt**

V svojom zadanie som sa rozhodol implementovať binárne vyhľadávacie stromy AVL(2) a 2-3(3), ako aj hashovacie tabuľky s rozlíšením kolízií pomocou otvoreného adresovania(4) a reťazenia(5). Najprv v dokumentácii zvážime vlastnosti každej štruktúry, ako aj ich implementáciu v jazyku C, potom prejdeme k testovacím údajom(6), z výsledkov ktorých bude možné vyvodiť definitívny záver.

# Obsah



Dokumentaciá zadania č.1

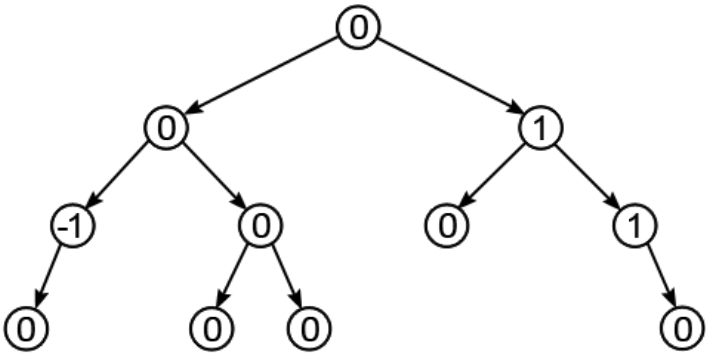
# 1 Úvod

V binárnych stromoch sú dáta uložené v odkazoch, ktoré sú vzájomne prepojené do stromovej štruktúry, takže každý odkaz (v závislosti od stromu) bude obsahovať: kľúč, alebo kľúče, dáta, ukazovatele na deti a tiež smerník na rodiča. Ak odkaz nemá žiadne potomky ani rodiča, potom bude ukazovateľ ukazovať na NULL.

# 2 AVL strom

Tento typ binárneho vyhľadávacieho stromu sa považuje za vyvážený.

AVL-strom je vyvážený binárny vyhľadávací strom s k=1. Pre jeho uzly je definovaný faktor rovnováhy. Faktor rovnováhy je rozdiel medzi výškami pravého a ľavého podstromu, ktorý nadobúda jednu hodnotu z množiny {-1, 0, 1}.




Obr. 1: príklad stromu AVL, ktorého každý uzol má priradený koeficient vyvažovania.

## 2.1 Vkladanie

Operáciu vloženia nového uzla do stromu AVL som implementoval rekurzívne.

1. Pomocou kľúča tohto uzla sa hľadá bod vloženia: algoritmus prechádzajúci stromom porovná kľúč pridaného uzla s kľúčmi, ktoré sa našli;
2. Potom sa vloží nový prvok;
3. Po návrate z rekurzie sa uje vyváženie uzlov a v prípade potreby sa vykoná vyvažovanie.

## 2.2 Vyvažovanie

Ak sa po vykonaní operácie pridania alebo vymazania faktor rovnováhy ktorémhokoľvek uzla stromu AVL rovná 2, potom musíme vykonať operáciu vyváženia.

```
btree* insertAVL(btree * Head, int index) {
    if (index < Head->index) {
        Head->nextL = insertAVL(Head->nextL, index);
        Head->nextL->parent = Head;
    }
    else {
        Head->nextR = insertAVL(Head->nextR, index);
        Head->nextR->parent = Head;
    }
    return Balance(Head);
}
```

Obr. 2: Zdrojový kód funkcie vloženia nového uzla

Vykonáva sa otáčaním (otočením) uzlov - zmenou väzieb v podstrome. Rotácie nemenia vlastnosti binárneho vyhľadávacieho stromu a sú vykonávané v konštantnom čase. Celkovo existujú 4 typy:

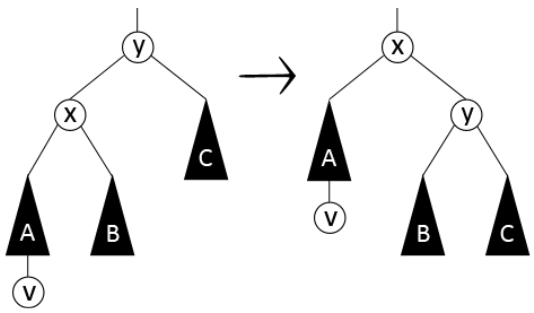
```
void OverHeight(btree* p){

    int hleft = GetHeight(p->nextL);
    int hright = GetHeight(p->nextR);

    p->height = (hleft > hright ? hleft : hright) + 1;
}
```

Obr. 3: Zdrojový kód funkcie nastavenia vyšky pre odkaz

1. Malá rotácia vpravo;
2. Veľká rotácia vpravo;
3. Malá rotácia doľava;
4. Veľká rotácia doľava.

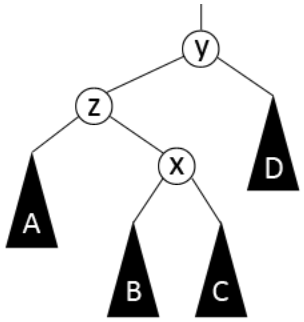
Oba typy veľkých rotácií sú kombináciou malých rotácií (pravo-ľavé alebo ľavo-pravé otáčanie).

Existujú dva prípady nerovnováhy. Prvý z nich je opravený typmi 1 a 3 a druhý typmi 2 a 4. Zvážme prvý prípad. Na obrázku č. 3 môžete vidieť, ako sa vykonáva vyváženie vykonaním malého pravého otočenia.




Obr. 4: Malá rotácia vpravo

Malá rotácia doľava sa vykonáva symetricky k malej rotácii doprava. Druhý prípad nerovnováhy sa koriguje veľkou rotáciou doprava alebo doľava.(5) Najprv vykonáme malú odbočku doprava a potom sa malá odbočka doľava a spojka X stane koreňom a Z a Y sa stanú jej ľavými a pravými synmi.




Obr. 5: Malá rotácia vpravo

### 2.3 Vyhľadávanie prvku

Funkcia vyhľadávania odkazov sa implementuje presunom z koreňového na nasledujúci odkaz, kým sa nenájde odkaz s kľúčom, ktorý hľadáme. Vpravo, ak je kľúč väčší ako aktuálny, a vľavo, ak je kľúč menší.

### 2.4 Vymazanie prvku

Funkcia odstránenia odkazu je implementovaná rekurzívne. Najprv sa pohybujeme po uzloch nadol, kým nenájdeme uzol, ktorý potrebujeme, potom hľadáme minimálny prvok v pravom podstrome tohto uzla, aby sme ho po odstránení presunuli na svoje miesto. A úplne na záver zavoláme vyvažovaciu funkciu, ktorá strom v prípade potreby zarovná.

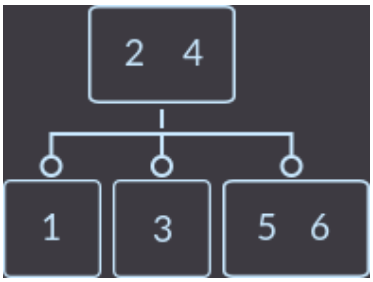
## 3 2-3 strom

2-3 strom je dátová štruktúra, ktorej prvky môžu obsahovať iba 2-vrcholy (vrcholy s jedným poľom a 2 potomkami) a 3-vrcholy (vrcholy s 2 poľami a 3 potomkami). Výnimkou sú vrcholy listov – nemajú potomkov (ale môžu mať jedno alebo dve polia). 2-3 stromy sú vyvážené, to znamená, že každý ľavý, pravý a stredný podstrom má rovnakú výšku, a teda obsahuje rovnaké (alebo takmer rovnaké) množstvo údajov.

Každý objekt triedy stroma 2-3 má 4 ukazovatele pre deti a pole 3 premenných celočíselného typu na ukladanie kľúčov.

### 3.1 Vkladanie

Ak chceme do stromu vložiť prvok s kľúčom, musíme postupovať podľa algoritmu:




Obr. 6: príklad 2-3 stromu

1. Ak je strom prázdny, vytvorme nový vrchol, vložme kľúč a vráťme tento vrchol ako koreň, inak
2. Ak je vrcholom list, vložíme kľúč do tohto vrcholu a ak dostaneme 3 kľúče na vrchole, tak ho rozdelíme, inak
3. Porovnajme kľúč s prvým kľúčom uzla a ak je kľúč menší ako tento kľúč, prejdime na prvý podstrom a prejdime na krok 2, inak
4. Pozrieme sa, ak vrchol obsahuje iba 1 kľúč (je to 2-vrchol), potom prejdeme do pravého podstromu a prejdeme k bodu 2, inak
5. Porovnajme kľúčový kľúč s druhým kľúčom v uzle a ak je kľúč menší ako druhý kľúč, prejdime do stredného podstromu a prejdime na krok 2, inak
6. Choďme do pravého podstromu a prejdime k bodu 2.

```
bt23* insert23(bt23* head, int x) {
    if (isLeaf(head)) {
        insertToNode(head, x);
    }

    else if (x <= head->keys[0]) head->sons[0] = insert23(head->sons[0], x);

    else if ((head->size == 1) || ((head->size == 2) && (x <= head->keys[1]))) head->sons[1] = insert23(head->sons[1], x);

    else head->sons[2] = insert23(head->sons[2], x);

    return split(head);
}
```

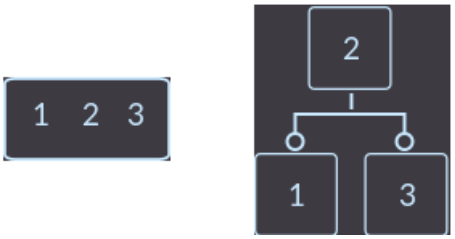
Obr. 7: Zdrojový kód funkcie vkladania pre 2-3 strom

## 3.2 Vyvažovanie

Na rozdiel od vyváženia stromu AVL s rotáciami sa tu veci zdajú jednoduchšie vďaka implementácii pomocou vrcholov s viacerými kľúčmi. K deleniu dochádza tiež podľa nasledujúceho princípu: ak má vrchol 3 kľúče, potom stred týchto kľúčov prenesieme na rodiča a zvyšok kľúčov roztriedime a spravíme z nich potomkov rodiča alebo nového koreňa, ak predtým žiadny rodič neexistoval.

## 3.3 Vyhľadávanie prvku

Poďme popísať algoritmus vyhľadávania:




Obr. 8: príklad vyvažovania 2-3 stromu

1. Hľadajme požadovaný kľúčový kľúč v aktuálnom vrchole, ak ho nájdeme, potom vráťme vrchol, inak
2. Ak je kľúč menší ako prvý kľúč vrcholu, prejdime do ľavého podstromu a prejdime na krok 1, inak
3. Ak je v strome 1 kľúč, potom prejdeme do pravého podstromu, a prejdeme na krok 1, inak
4. Ak je kľúč menší ako druhý kľúč vrcholu, prejdime do stredného podstromu a prejdime na bod 1, inak
5. Choďme do pravého podstromu a prejdime k bodu 1.

### 3.4 Vymazanie prvku

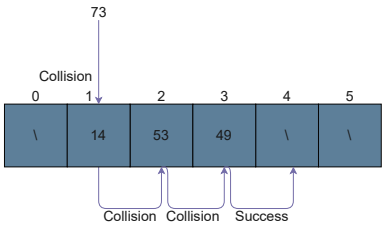
Vymazanie v strome 2-3, podobne ako v strome AVL, sa vyskytuje iba z listu. Preto, keď sme našli kľúč na odstránenie, najprv musíme skontrolovať, či sa tento kľúč nachádza v listovom alebo nelistovom vrchole. Ak je kľúč vo vrchole bez listu, musíte nájsť ekvivalentný kľúč pre odstránený kľúč z vrcholu listu a vymeniť ich.

## 4 Hašovacia tabuľka metodou otvoreneho adresovania

Objekt triedy bunky hašovacej tabuľky obsahuje premennú na ukladanie údajov, ako aj premennú pre kľúč. Hlavný rozdiel medzi hašovacou tabuľkou a binárnym vyhľadávacím stromom je v tom, že bunky v tabuľke, v ktorých sú uložené kľúče a údaje, sa nachádzajú na určitom indexe, ktorý je možné získať zašifrovaním samotného kľúča pomocou funkcie hash. Na prístup k prvku s požadovaným kľúčom bude stačiť nájsť prvok podľa indexu. Hlavnou nevýhodou takejto dátovej štruktúry je možnosť, že dva rôzne kľúče môžu byť zašifrované do rovnakého indexu. V tejto časti sa budeme zaoberať riešením kolízie metódou otvoreného adresovania.

### 4.1 Vkladanie

Na vloženie údajov do tabuľky musíme najskôr získať index bunky, kam ich chceme vložiť. V prvej aj druhej hašovacej tabuľke používam funkciu Key mod




Obr. 9: Schematické znázornenie riešenia kolízie metódou otvorenej adresy

Capacity. Ak je bunka voľná, tak tam môžeme zapisovať dáta a procedúra je ukončená, ale ak je bunka už obsadená, tak sa použije metóda otvoreného adresovania, vezmeme ďalší index, kým nenájdeme voľný. Aj vo funkcii vkladania údajov skontrolujeme plnosť tabuľky. Keď je už 70 percent tabuľky zaplnených, rozšíri sa 2-krát, čím sa vytvorí dostatočný počet voľných buniek.

```
int hashFunction1(int key, int capacity) {
    return fmod(key, capacity);
}
```

Obr. 10: Zdrojový kód Hash-funkcie pre Hašovacie tabuľky

### 4.2 Vyhľadávanie prvku

Ak chceme nájsť prvok v tabuľky, najprv sa na prvok tabuľky odkážeme podľa indexu a porovnáme kľúče, ak sa kľúč nezhoduje, potom to nie je prvok, ktorý potrebujeme, podobne ako pri postupe vkladania, budeme hľadať bunku s kľúčom, ktorý nám vyhovuje. V najhoršom prípade sa pozrieme na každú bunku tabuľky, ale hašovacia funkcia Key mod Capacity nemá vysokú mieru kolízií, čo nám umožňuje netráviť na to príliš veľa času.

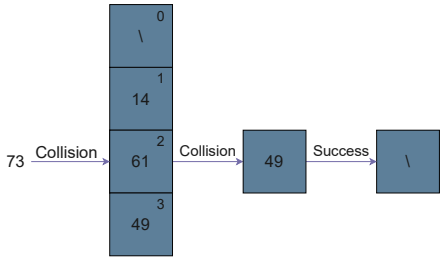
### 4.3 Vymazanie prvku

Pomocou vyhľadávacieho algoritmu nájdeme prvok, ktorý potrebujeme a zresetujeme v ňom údaje, aby sa dal znova použiť. V tomto môžete vidieť aj plusy hash tabuľky, pretože čas navyše sa nepoužije na vytvorenie prvku a jeho odstránenie, stačí resetovať údaje a tabuľka hash bude naďalej vykonávať svoje funkcie

## 5 Hašovacia tabuľka metodou reťazenia

Štruktúra tohto typu tabuľky je takmer identická s predchádzajúcou, až na to, že každý objekt tabuľky bude mať ukazovateľ na rovnaký typ prvku. Na začiatku je každý zavádzací ukazovateľ nastavený na hodnotu NULL, ale použije sa pri kolíziách.

## 5.1 Vkladanie




Obr. 11: Schematické znázornenie riešenia kolízie metódou reťazenia

V tomto type štruktúry pri kolízii skontrolujeme smerník na nasledujúci prvok a ak neexistuje, tak vytvoríme nový prvok a tam zapíšeme svoje údaje. Ak ukazovateľ nie je prázdny a pri prístupe k ukazovateľu obsahuje nejaké údaje, potom musíme skontrolovať ukazovateľ tohto ukazovateľa a pohybovať sa takto, kým nenájdeme nulový ukazovateľ alebo ukazovateľ bez zapísaných údajov. Tento spôsob riešenia kolízií si vyžaduje viac času, pretože musíte zakaždým vytvoriť nový prvok, zatiaľ čo pri metóde otvoreného adresovania jednoducho rozšírime tabuľku dvakrát.

## 5.2 Vyhľadávanie prvku

Aby sme našli prvok, najskôr pristúpime k indexu, ktorý získame pomocou hašovacej funkcie, ak neobsahuje kľúč, ktorý potrebujeme, potom prejdeme na ďalší prvok ukazovateľom a tak ďalej, kým nenájdeme kľúč, ktorý potrebujeme potreba alebo kým nebude ukazovateľ NULL . Ak je ukazovateľ NULL, môžeme dospieť k záveru, že takýto prvok v tabuľke nie je.

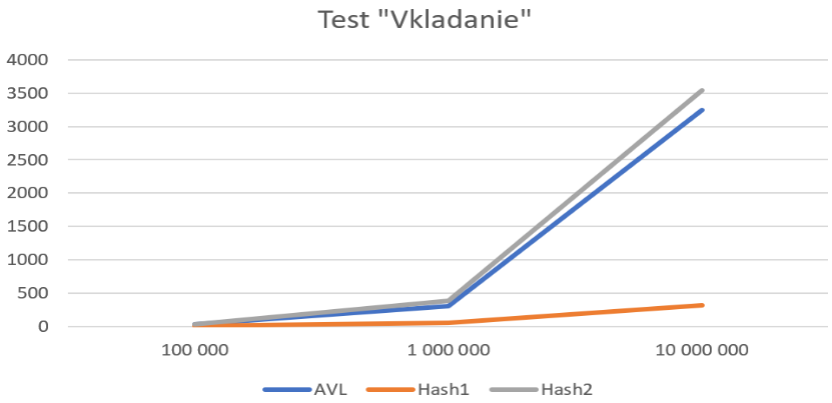
## 5.3 Vymazanie prvku

Podobne pri funkcii vyhľadávania prvkov nájdeme odkaz na vymazanie, ale namiesto vyčistenia pamäte a presunu ukazovateľov, ako sme to robili v binárnych stromoch, odtiaľ údaje jednoducho vymažeme, a ak už ide o prvok jednoducho prepojený zoznam, potom sa doň môže niečo zapísať znova bez prerozdelenia pamäte.

# 6 Testovanie

Testovanie prebiehalo porovnaním času, za ktorý AVL strom, prvý typ hašovacej tabuľky a hašovacia tabuľka druhého typu vytvárali, nachádzali a odstraňovali prvky. Pre porovnanie bolo vykonaných niekoľko testov s rôznym počtom prvkov, 100 000, 1 000 000 a 10 000 000 . Nižšie budú priložené grafy znázorňujúce závislosť rýchlosti operácií od počtu prvkov, čas je uvedený v milisekundách.

## 6.1 Test: Vkladanie




| | 100 000 | 1 000 000 | 10 000 000 |
|---|---|---|---|
| AVL | 33 | 301 | 3246 |
| Hash1 | 2 | 48 | 312 |
| Hash2 | 33 | 381 | 3543 |

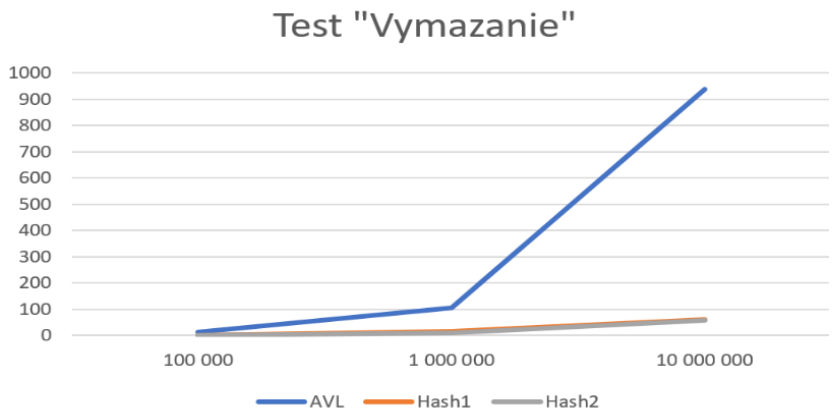
Obr. 12: Test: Vkladanie

Graf ukazuje, že všetky štruktúry si rýchlo poradia so 100 000 prvkami, no akonáhle počet prvkov narastie na 1 000 000, vidíme, že hash tabuľka s metódou riešenia kolízií otvoreným adresovaním sa s testom vyrovná rýchlejšie ako ostatné, takéto výsledky sú spôsobené tým, že pri kolízii sa v tejto hašovacej tabuľke nevytvoria nové prvky, ktoré sa vytvoria pri každom vložení uzla do AVL stromu alebo pri kolíziách v hašovacej tabuľke druhého typu[1] sa okamžite zapíšu do už na začiatku pridelená pamäť a keď sa tabuľka rozšíri, okamžite sa zväčší 2-krát, čo nezaberie veľa času, ako vytvorenie nového odkazu pre každý prvok zvlášť v stromoch AVL. Podľa výsledkov testu možno zdôrazniť, že strom AVL vkladá prvky do tabuľky rýchlejšie ako hash tabuľka druhého typu

## 6.2 Test: Vymazanie

V teste funkcie odstránenia môžeme vidieť, že hašovacie tabuľky majú približne rovnaký čas vymazania, pretože vyhľadávanie ukazovateľov trvá o niečo dlhšie ako vyhľadávanie indexov. Strom AVL za nimi výrazne zaostáva, pretože pri každom odstránení sa ešte musí vyrovnávať, to vysvetľuje taký časový rozdiel. Podľa údajov v tabuľke je vidieť, že pri odstraňovaní prvkov hašovacia tabuľka druhého typu pracuje rýchlejšie o 5 milisekúnd, ale to nestačí na záver, že táto hašovacia tabuľka je efektívnejšia. Boli aj testy na vyhľadávanie prvkov v štruktúrach, ale tam všetky štruktúry vykazovali príliš rýchle výsledky, bez ohľadu na počet prvkov v nich. Preto som ich výsledky neporovnával, aj keď intuitívne sa zdá, že AVL stromy by v tomto mali byť o niečo pomalšie ako hashovacie tabuľky

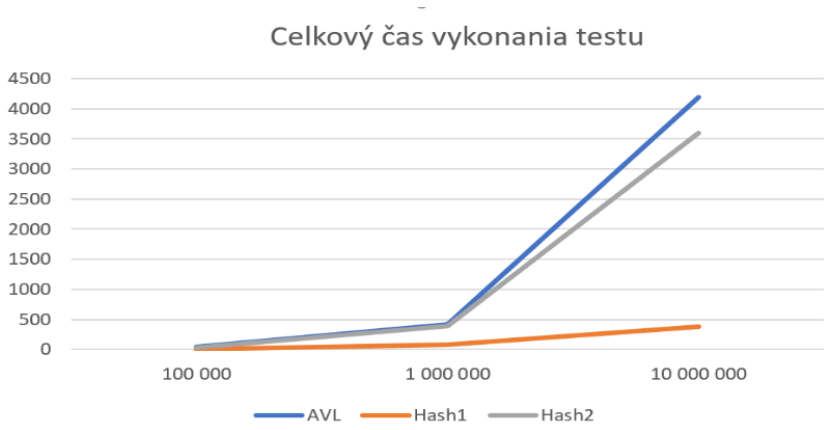
[1] Hašovacia tabuľka z metodou riešenia kolizií reťazením




| | 100 000 | 1 000 000 | 10 000 000 |
|---|---|---|---|
| AVL | 12 | 106 | 939 |
| Hash1 | <0.1 | 15 | 62 |
| Hash2 | <0.1 | 10 | 57 |

Obr. 13: Test: Vymazanie

## 6.3 Záver




| | 100 000 | 1 000 000 | 10 000 000 |
|---|---|---|---|
| AVL | 46 | 410 | 4187 |
| Hash1 | 3 | 86 | 377 |
| Hash2 | 35 | 393 | 3601 |

Obr. 14: Celkový čas vykonania testu

Z celkového času vykonania testu môžeme usúdiť, že najefektívnejšie sú hašovacie tabuľky prvého typu, nasledované účinnosťou hašovacích tabuliek druhej

úrovne a strom AVL sa ukázal ako najpomalšia štruktúra.