Peter Brenkus, ID: 120762

FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
SLOVENSKÁ TECHNICKÁ UNIVERZITA

Ilkovičova 2, 842 16 Bratislava 4

2022/2023
Dátové štruktúry a algoritmy
Zadanie č.1

Úvod a zadanie

Účelom tohto článku je zdokumentovať moju prácu na 1. zadaní v predmete Dátové štruktúry a algoritmy. Zdrojový kód som písal v jazyku C++. V prvom semestri som sa naučil jazyk C, no C++ má napríklad knižnicu <list>, typ ‘auto‘ a niekoľko ďalších vecí, ktoré mi pri vypracovávaní tohto zadania zjednodušili život, a navyše som sa naučil nový programovací jazyk.

Kód som písal a testoval v Microsoft Visual Studio Code 1.76.2 na zariadení MacBook Pro 2020 13“ s operačným systémom macOS Big Sur 11.2.3, štvorjadrovým procesorom Intel Core i7 (2,3GHz) a 16GB RAM.

Zadaním bolo implementovať dva binárne stromy s rôznymi vyvažovacími algoritmami podľa výberu a dve hashovacie tabuľky s roznym riešením kolízii a schopnosťou dynamického prispôsobovania veľkosti tabuliek.

Obsah

Zdrojové kódy

AVL Strom

Ako prvú dátovú štruktúru som si vybral AVL strom na základe odporúčania kamarátov z vyšších ročníkov. Definoval som si štruktúru Node (uzol stromu) v ktorej sa uchovávajú vstupné dáta. Aby som sa naučil pracovať s objektovou paradigmou, ktorá bola pre mňa nová, implementoval som ho v triede – AVLTree, kde som jednotlivé funkcie zadefinoval ako metódy tejto triedy. AVL stromy sa balansujú tak, že každý uzol má v sebe navyše informáciu o svojej výške v strome a rozdiel výšok jeho detí nemôže byť väčší ako 1 alebo menší ako -1.

Balansovanie

Na účely vyvažovania stromu som si zadefinoval pomocné funkcie **getHeight(*node)**, **getBalance(*node)** a **updateHeight(*node)**, tieto sú ďalej využité vo funkcii **balanceNode(*node)**.

Funkcia **getHeight(*node)** je veľmi jednoduchá, no veľa krát sa používa a preto je jednoduchšie si ju zadefinovať zvlášť. Ak uzol neexistuje (*node == NULL), vráti 0, inak vráti jeho výšku. Týmto spôsobom sa vyhýbam možnému segmentation faultu.

Funkcia **getBalance(*node)** vráti 0, ak uzol neexistuje (*node == NULL), inak vypočíta a vráti balans daného uzla: *getHeight(ľavé dieťa) mínus getHeight(pravé dieťa).*

Funkcia **updateHeight(*node)** sa volá po vložení alebo vymazaní uzla a slúži na aktualizáciu výšky daného uzla. Funkcia **max(...)** vráti maximálnu hodnotu z výšok detí a k tomu sa pripočíta +1.

```
int getHeight(Node* node){
    if (node==NULL){
        return 0;
    }
    else return (node->height);
}
int getBalance(Node* node){
    if (node==NULL) {
        return 0;
    }
    else return (getHeight(node->lower) - getHeight(node->higher));
}
void updateHeight(Node* node){
    if (node!=NULL){
        node->height = (max(getHeight(node->lower), getHeight(node->higher)) + 1);
    }
}
```

Vyššie spomenuté funkcie majú ako som už spomenul uplatnenie vo funkcii **balanceNode(*node).** V tejto funkcii sa na základe hodnoty balansu *(int bf)* volajú jednotlivé rotácie:

- Ak je balans uzla > 1 a balans jeho menšieho dieťaťa je < 0,
  zavolá sa najprv **leftrotate(*node)** a potom **rightRotate(*node)**
- Ak je balans uzla > 1 a balans jeho menšieho diťaťa je >= 0,
  zavolá sa iba **rightRotate(*node)**
- Ak je balans uzla < -1 a balans jeho väčšieho dieťaťa je > 0,
  zavolá sa najprv **rightRotate(*node)** a potom **leftRotate(*node)**

- Ak je balans uzla < -1 a balans jeho väčšieho dieťaťa je <= 0, zavolá sa iba **leftRotate(*node)**

```
Node *balanceNode(Node *node){
    if (node == NULL){
        return node;
    }
    int bf = getBalance(node);
    if (bf > 1){
        if (getBalance(node->lower) < 0){
            node->lower = leftRotate(node->lower);
        }
        return rightRotate(node);
    }
    else if (bf < -1){
        if (getBalance(node->higher) > 0){
            node->higher = rightRotate(node->higher);
        }
        return leftRotate(node);
    }
    return node;
}
```

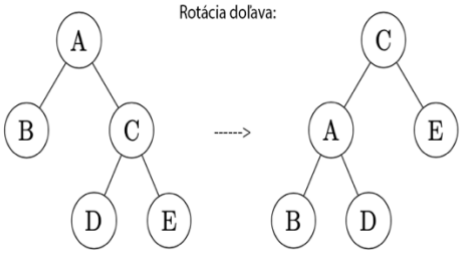
```
Node* leftRotate(Node* node){
    //cout <<"rotL\n";
    Node* temp = node->higher;
    Node* temp2 = temp->lower;
    temp->lower = node;
    node->higher = temp2;
    updateHeight(node);
    updateHeight(temp);
    return temp;
}
Node* rightRotate(Node* node){
    //cout <<"rotR\n";
    Node* temp = node->lower;
    Node* temp2 = temp->higher;
    temp->higher = node;
    node->lower = temp2;
    updateHeight(node);
    updateHeight(temp);
    return temp;
}
```

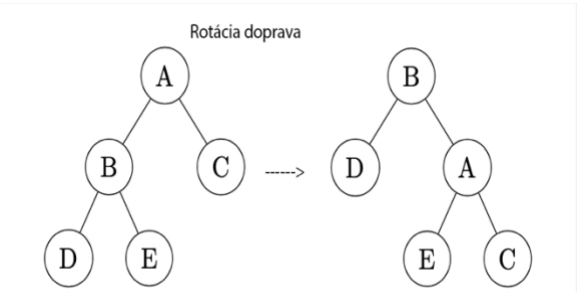
Tieto rotácie sú jadrom balansovania pomocou AVL algoritmu, a zabezpečujú fakt, že v AVL strome nikdy nie je niektorá vetva dlhšia (vyššia) o viac ako jeden uzol od druhej. Vďaka tomu sú AVL stromy kompletne vybalansované, narozdiel od mnohých iných stromov, ktoré sú iba približne vybalansované.

Funkcia **leftRotate(*node)** dostáva ukazovateľ na uzol *(node),* podľa ktorého chceme strom otočiť doľava. Do pomocnej premennej *(temp)* uloží väčšie dieťa tohto uzla, a do ďalšej pomocnej premennej *(temp2)* uloží menšie dieťa štruktúry uloženej v temp. Tieto kroky sú nevyhnutné na to, aby sa zabránilo strácaniu detí alebo porušeniu vlastností stromu. Ďalej sa zamenia temp za pôvodný uzol, aktualizuje sa výška uzlov, ktoré sa vymieňali a vracia sa ukazovateľ temp.

Funkcia **rightRotate(*node)** robí analogický opak funkcie ľavého otáčania. Praktický príklad týchto rotácii je uvedený na obrázkoch nižšie.

Insert, Search, Delete





Trieda AVLTree má ďalej metódy:

- **addNode()** – na vkladanie uzlov
- **findNode()** – na vyhľadávanie uzlov
- **deleteNode()** – na mazanie uzlov

Funkcia **addNode(*node, ...)** funguje rekurzívne. Postupne hľadá kam má vložiť zadané dáta podľa hodnoty value *(key)* a akonáhle sa posunie na miesto stromu, kde nie je uzol, tak ho vyrobí a vloží na to miesto. Ak zadaná hodnota v strome už je, funkcia vráti pôvodný strom (neurobí teda nič). Ak sa nejaký prvok do stromu pridal, zavolajú sa vyššie spomenuté funkcie **updateHeight(*node)** a **balanceNode(*node).**

```
Node* addNode(Node* node, int value, string meno){
if (!(node)){
    node =  new Node;
    node->lower = NULL;
    node->higher = NULL;
    node->name = meno;
    node->key = value;
}
else if (value>((node)->key)){
    (node)->higher = addNode(((node)->higher), value, meno);
}
else if (value<((node)->key)){
    (node)->lower = addNode(((node)->lower), value, meno);
}
else{
    return (node);
}
updateHeight(node);
return balanceNode(node);
}
```

Funkcia **findNode(*node, value)** funguje tiež rekurzívne. Postupne prechádza strom podľa zadanej hodnoty, ktorú má nájsť a ak nájde uzol s danou hodnotou, vráti dáta uchované v tomto uzle, ináč vráti správu o tom že uzol nebol nájdený.

```
string findNode(Node* node, int value){
    if (!node){
        return "No such node";
    }
    if ((node->key)==value){
        return node->name;
    }
    if ((value < node->key)&&((node->lower)!=NULL)){
        return findNode(node->lower, value);
    }
    else if ((value > node->key)&&((node->higher)!=NULL)){
        return findNode(node->higher, value);
    }
    else{
        return "No such node";
    }
}
```

Funkcia **deleteNode(*node)** najprv rekurzívne nájde uzol, ktorý sa má vymazať podľa zadanej hodnoty, podobne ako pri hľadaní vo funkcii **findNode(*node, value)**,s tým rozdielom, že ak sa uzol danej hodnoty v strome nenáchádza, nič sa nestane.

```
Node* deleteNode(Node* node, int value){
if (!node){
    return (node);
}
if (value < (node->key)){
    (node->lower) = deleteNode((node->lower), value);
}
else if (value > (((node)->key))){
    (node->higher) = deleteNode((node->higher), value);
}
```

Následne, môžu nastať štyri situácie. Uzol, ktorý sa má vymazať má:

- žiadne dieťa
- iba menšie alebo iba väčšie dieťa
- obidve deti

Funkcia rozpozná, o akú situáciu ide a vyrieši ich nasledovne:

- ak nemá žiadne dieťa, jednoducho ho vymaže
- ak má iba jedno dieťa, ukazovateľ na uzol sa posunie na svoje dieťa a pôvodný uzol sa vymaže
- ak má obe deti, funkcia nájde najmenší uzol z pravého podstromu pôvodného uzla *(temp)*, nahradí hodnotu pôvodného uzla touto hodnotou *(temp->key)* a následne rekurzívne vymaže duplikátnu hodnotu.

```
    else{
        if (((node->lower) == NULL)&&((node->higher) == NULL)){
            delete (node);
            node=NULL;
        }
        else if (((node->lower) == NULL)&&((node->higher) != NULL)){
            Node* temp = node;
            node = node->higher;
            delete temp;
            temp = NULL;
            return node;
        }
        else if (((node->higher) == NULL)&&((node->lower) != NULL)){
            Node* temp = node;
            node = (node->lower);
            delete temp;
            temp = NULL;
            return node;
        }
        else{
            Node* temp = (node->higher);
            while ((temp->lower)!=NULL){
                temp=(temp->lower);
            }
            (node)->key = (temp->key);
            (node->higher) = deleteNode((node->higher), (temp->key));
        }
    }
    updateHeight(node);
    return balanceNode(node);
}
```

Po vymazaní sa samozrejme opäť zavolajú funckie **updateHeight(*node)** a **balanceNode(*node),** aby sa zaistilo, že strom je aj naďalej vyvážený.

Výpis a main funkcia

Poslednou metódou v triede AVLTree je **printTree(*node, depth),** ktorá jednoducho rekurzívne vypíše uzly stromu od koreňa do terminálu v smere zprava doľava. Ak je strom prázdny, funkcia neurobí nič.

```
void printTree(Node* node, int depth = 0){
    if (node == NULL){
        return;
    }
    printTree(node->higher, depth + 1);
    for (int i = 0; i < depth; i++){
        cout << "    ";
    }
    cout << node->key << " (" << node->name << ")" << endl;
    printTree(node->lower, depth + 1);
}
```

Na začiatku mainu mám deklarovaných zopár premenných, pomocou ktorých sa vykonávajú zmeny v strome, ukazovateľ na objekt triedy AVLTree a ukazovatele na súbory rôznych veľkostí s náhodne vygenerovanými číslami a reťazcami pre účely testovania.

Ďalej mám v maine už len veľký switch(), ktorý na základe užívateľom zadaného vstupu rozhoduje o tom, či sa bude so stromom pracovať manuálne alebo prebehne automatický test.

```
while (true){
    cout << "m->MANUAL/a->AUTO/q->QUIT\n";
    cin >> vstup;
    switch(vstup){
        case 'm' :{…
        case 'a' :{…
        case 'q' :{
            return 0;
        }
        default :{
            cout << "m->MANUAL/A->AUTO/q->QUIT\n";
            break;
        }

    }
}
return 0;
}
```

Samozrejme, že užívateľ má možnosť program aj vypnúť, ak stlačí ‘q’.

V prípade že stlačí čokoľvek iné ako zadané hodnoty, program len vypíše správu s inštrukciami na používanie.

Ak užívateľ stlačí ‘m’, spustí sa ďalší switch(), v pomocou ktorého môže používateľ po jednom buď pridávať uzly do stromu, vyhľadávať ich, mazať ich alebo strom vypísať do terminálu. Samozrejme, že aj z tohto miesta má užívateľ možnosť sa vrátiť naspäť na výber medzi manuálnym používaním a automatickým testom. Na začiatku, ako aj po zadaní hociakého neakceptovaného vstupu program vypíše správu s inštrukciami.

Ak užívateľ stlačí ‘a‘, spustí sa switch(), v ktorom si môže užívateľ vybrať, na akej veľkej sade uzlov sa strom automaticky otestuje. Test prebehne tak, že sa najprv zadaný počet uzlov vloží do stromu, potom sa po jednom vyhľadajú a nakoniec sa po jednom vymažú. Program následne vypíše správu o čase trvania jednotlivých operácii ako aj celkovom čase, za ktorý to celé prebehlo. Výpis správy s inštrukciami ako aj možnosť vrátenia sa tu fungujú rovnako ako pri manuálnom testovaní.

Tieto switche sú na príliš veľa riadkov na to, aby som ich mohol nejako rozumne celé ukázať tu, no vždy vypíšu užívateľovi na začiatku možnosti ovládania a sú pomerne jednoduché na používanie. Pre celistvosť tejto dokumentácie je ešte nižšie uvedená prvá možnosť z automatického testovania, ktorá je vzorom pre všetky ostatné.

```
case 'a' :{
    auto startTime = high_resolution_clock::now();
    for (int i = 0; i< 10; i++){
        numero10 >> number;
        gringo10 >> meno;
        root = myTree->addNode(root, number, meno);
    }
    auto insertTime = high_resolution_clock::now();
    numero10.seekg(0);
    for (int i = 0; i< 10; i++){
        numero10 >> number;
        myTree->findNode(root, number);
    }
    auto searchTime = high_resolution_clock::now();
    numero10.seekg(0);
    for (int i = 0; i< 10; i++){
        numero10 >> number;
        root = myTree->deleteNode(root, number);
    }
    auto endTime = high_resolution_clock::now();
    auto insertion = duration_cast<microseconds>(insertTime - startTime).count();
    cout << "Insert took: " << insertion << " microseconds." << endl;
    auto search = duration_cast<microseconds>(searchTime - insertTime).count();
    cout << "Search took: " << search << " microseconds." << endl;
    auto deletion = duration_cast<microseconds>(endTime - searchTime).count();
    cout << "Delete took: " << deletion << " microseconds." << endl;
    auto duration = duration_cast<microseconds>(endTime - startTime).count();
    cout << "Total time taken: " << duration << " microseconds." << endl;
    numero10.seekg(0);
    break;
}
```

Záverečné poznámky:

- Časy meraní sú uvedené v mikrosekundách, pri sadách 1M a 10M sú však uvedené v milisekundách, nakoľko desatinné miesta sú pri týchto možnostiach zanedbateľné.
- Podľa nastavenia compileru môže po spustení programu užívateľa privítať až krásnych 56 warningov, všetky sa však sťahujú na zmienený typ ‘auto’, ktorý je použitý pri deklaráciach časových premenných pri automatickom testovaní. Tento typ je totiž súčasťou jazyka C++ až od verzie 11 a vyššie, dnes sa už však všeobecne používa verzia 17, takže tieto warningy sú viacmenej bezpredmetné – hovoria len o tom, že program je nekompatibilný so zastaranou verziou jazyka.
- Ak sa použije možnosť q vo výbere m, tj. je možnosť, že používateľ manuálne pridal do stromu nejaké uzly a zabudol ich odtiaľ pred odchodom korektne vymazať, zavolá sa funkcia destructTree(), ktorá to urobí za neho. Preistotu sa volá aj po stlačení q vo výbere a, aj keď kód automatických testov všetky uzly vymazáva.

Splay strom

Ako druhý algoritmus na balansovanie binárneho vyhľadávacieho stromu som si vybral Splay, kvôli jeho zaujímavej charaktristike, ktorá ho v mojich očiach robí dobrým kandidátom na použitie v mnohých prípadoch, kde sa niektoré dáta používajú opakovane. Splay stromy sa vyvažujú pomocou splay funkcie popísanej nižšie.

Balansovanie

Na balansovanie Splay stromu slúži funkcia **splayNode(*myNode, *rootNode),** ktorá na vstupe dostáva ukazovateľ na uzol, ktorý má byť „vynesený“ *(mynode)* a ukazovateľ na uzol, ktorého dieťaťom sa má stať *(rootNode),* tj. pozíciu kam sa má *myNode* „vyniesť“. Pomocou rotácii potom „vynesie“ zadaný uzol na zadanú pozíciu v strome, najčastejšie na koreň – ak má byť uzol „vynesený“ na koreň, destinácia *rootNode* je nastavená na nullptr, keďže koreň ako jediný nemá rodiča.

Funkcia prebieha, pokým rodič daného uzla nie je ten ktorý bol zadaný. Najprv sa vždy overí či náhodou nie je rodič rodiča uzla jeho finálnou destináciou, v tom prípade totiž stačí jediná posledná rotácia:

- Ak je uzol napravo od svojej destinácie tak sa zrotuje doľava
- Ak je uzol naľavo od svojej destinácie tak sa zrotuje doprava

Ak nie, môžu nastať štyri situácie:

- Uzol je menší od svojho rodiča a jeho rodič je menší od svojho rodiča, treba vykonať dve pravé rotácie
- Uzol je väčší od svojho rodiča a jeho rodič je väčší od svojho rodiča, treba vykonať dve ľavé rotácie
- Uzol je menší od svojho rodiča a jeho rodič je väčší od svojho rodiča, treba vykonať najprv pravú rotáciu a potom ľavú rotáciu
- Uzol je väčší od svojho rodiča a jeho rodič je menší od svojho rodiča, treba vykonať najprv ľavú rotáciu a potom pravú rotáciu

Tieto rotácie sa inak nazývajú aj zig, zag, zig-zig, zig-zag,...

Rotácie, ktoré sú v tejto funkcii volané, sú podobné tým z AVL stromu, no tieto zahŕňajú aj zmeny ukazovateľov na rodičov prvkov, ktoré sa rotujú, keďže v tomto prípade je aj táto premenná v hre. Kód popísaných funkcii je pohromade na ďalšej strane.

```
Node* splayNode(Node* myNode, Node* rootNode){
    if (myNode->parent == rootNode){
        return myNode;
    }
    while(myNode->parent != rootNode){
        Node* grandParent = myNode->parent->parent;
        Node* Parent = myNode->parent;
        if (grandParent == rootNode){
            if (myNode == Parent->lower){
                rightRotate(Parent);
            }
            else if (myNode == Parent->higher){
                leftRotate(Parent);
            }
        }
        else if((myNode == Parent->lower)&&(Parent == grandParent->lower)){
            grandParent = rightRotate(grandParent);
            Parent = rightRotate(Parent);
        }
        else if((myNode == Parent->higher)&&(Parent == grandParent->higher)){
            grandParent = leftRotate(grandParent);
            Parent = leftRotate(Parent);
        }
        else if((myNode == Parent->lower)&&(Parent == grandParent->higher)){
            Parent = rightRotate(Parent);
           grandParent = leftRotate(grandParent);
        }
        else if((myNode == Parent->higher)&&(Parent == grandParent->lower)){
            Parent = leftRotate(Parent);
            grandParent = rightRotate(grandParent);
        }
    }
    return myNode;
}
```

```
Node* leftRotate(Node* myNode){
    Node* temp = myNode->higher;
    myNode->higher = temp->lower;
    if (temp->lower != nullptr){
        temp->lower->parent = myNode;
    }
    temp->parent = myNode->parent;
    if (myNode->parent == nullptr){
        root = temp;
    }
    else if(myNode == myNode->parent->lower){
        myNode->parent->lower = temp;
    }
    else if(myNode == myNode->parent->higher){
        myNode->parent->higher = temp;
    }
    if (temp != nullptr){
        temp->lower = myNode;
        myNode->parent = temp;
    }
    return temp;
}
```

```
Node* rightRotate(Node* myNode){
    Node* temp = myNode->lower;
    myNode->lower = temp->higher;
    if(temp->higher != nullptr){
        temp->higher->parent = myNode;
    }
    temp->parent = myNode->parent;
    if (myNode->parent == nullptr){
        root = temp;
    }
    else if(myNode == myNode->parent->lower){
        myNode->parent->lower = temp;
    }
    else if (myNode == myNode->parent->higher){
        myNode->parent->higher = temp;
    }
    if (temp != nullptr){
        temp->higher = myNode;
        myNode->parent = temp;
    }
    return temp;
}
```

Insert, Search, Delete

Podobne ako predtým, aj trieda SplayTree má tieto metódy:

- **addNode()**
- **findNode()**
- **deleteNode()**

Funkcia **addNode()** už však nie je rekurzívna ale využíva while cyklus a pomocnú funkciu **newNode(),** ktorá jednoducho vytvorí nový uzol stromu, uloží do neho zadané hodnoty a ukazovatele nastaví na nullptr.

AddNode() najprv pozrie, či koreň stromu je prázdny. Ak áno, pomocou newNode() vytvorí uzol a vráti ho ako koreň. Ak nie, presunie sa do while cyklu, v ktorom sa presúva po strome, kým nenájde vhodné miesto kam nový uzol vložiť. Pri tom nastávajú tri situácie:

- uzol s danou hodnotou už v strome existuje
- momentálne objavený uzol má väčšiu hodnotu ako je zadávaná
- momentálne objavený uzol má menšiu hodnotu ako je zadávaná

```
Node* addNode(Node* myNode, int key, string name){
    if (myNode == nullptr){
        root = newNode(key, name);
        return root;
    }
    Node* temp = myNode;
    while(true){
        if(temp->key == key){
            splayNode(temp, nullptr);
            return root;
        }
        else if(temp->key > key){
            if(temp->lower == nullptr){
                //cout << "add2\n";
                temp->lower = newNode(key, name);
                temp->lower->parent = temp;
                root = splayNode(temp->lower, nullptr);
                return root;
            }
            else{
                temp = temp->lower;
            }
        }
        else if(temp->key < key){
            if (temp->higher == nullptr){
                //cout << "add3\n";
                temp->higher = newNode(key, name);
                temp->higher->parent = temp;
                root = splayNode(temp->higher, nullptr);
                return root;
            }
            else {
                temp = temp->higher;
            }
        }
```

Ak je uzol s danou hodnotou už v strome, zavolá sa splayNode() a tam to končí.

Ak má momentálny uzol väčšiu alebo menšiu hodnotu ako je zadávaná, môžu nastať dve totožné situácie:

- príslušné dieťa uzla je voľné
- príslušné dieťa nie je voľné

Ak je ukazovateľ na väčšie alebo menšie dieťa voľný, najprv sa pomocou **newNode()** na príslušné miesto pridá nový uzol a potom sa zavolá **splayNode().**

Ak ukzovateľ na príslušné dieťa nie je voľný, pomocný ukazovateľ na prehľadávanie sa jednoducho posunie na príslušnú pozíciu a cyklus sa opakuje, kým sa buď neobjaví uzol sa danou hodnotou alebo sa nepridá nový uzol.

Funkcia **findNode()** je totožná s tou z AVL stromu uvedenou vyššie. Jediným rozdielom je, že po úspešnom nájdení uzla sa volá funkcia **splayNode(),** ktorá hľadaný uzol „vynesie“ na koreň stromu.

```
string findNode(Node* myNode, int key){
    if (!myNode){
        return "No such node\n";
    }
    if (myNode->key == key){
        splayNode(myNode, nullptr);
        return myNode->name;
    }
    else if (myNode->key > key){
        return findNode(myNode->lower, key);
    }
    else if (myNode->key < key){
        return findNode(myNode->higher, key);
    }
}
```

Funkcia **deleteNode()** najprv „vynesie“ zadný uzol na koreň stromu pomocu findNode(). Ak sa v strome nenachádza uzol so zadanou hodnotou, nič sa nestane.

Ak sa však nachádza, tj. bol vynesený na koreň stromu, môžu nastať tieto situácie:

- koreň stromu nemá žiadne deti, tj. je jediným uzlom stromu
- koreň stromu má buď iba ľavé alebo iba pravé dieťa
- koreň stromu má aj pravé aj ľavé dieťa

Ak nemá žiadne deti, stačí ho jednoducho vymazať, ak má iba jedno dieťa, riešenie je podobne jednoduché – do pomocnej premennej sa uloží uzol na ktorý ukazuje koreň, ukazovateľ na koreň sa posunie na svoje dieťa a uzol, ktorý je uložený v dočasnej premennej sa vymaže.

Ak má koreň obidve deti, postup je podobný ako pri mazaní uzla, ktorý má obe deti z AVL stromu. Tam sa najpv hľadal najmenší uzol z pravého podstromu uzla, v tomto prípade sa deje analogický opak – hľadá sa najväčší uzol z ľavého podstromu. Tieto prístupy majú v princípe rovnaký výsledok, v oboch prípadoch ide o nájdenie uzla, ktorého hodnota v strome vie plniť rovnakú úlohu ako hodnota toho, ktorý sa má vymazať. V každom strome som použil iný prístup na podčiarknutie tejto skutočnosti.

Po nájdení najväčšieho uzla z ľavého podstromu a uložení tohto uzla do pomocnej premennej *temp* môžu nastať dve situácie:

- najväčší uzol ľavého podstromu je dieťa uzla, ktorý sa má vymazať
- najväčší uzol ľavého podstromu nie je dieťa uzla, ktorý sa má vymazať

Pri prvej možnosti sa pravý podstrom uzla, ktorý sa má vymazať presunie na pravú stranu uzla temp. Uzol ktorý sa má vymazať sa potom uloží do ďalšej pomocnej premennej *temp2,* ukazovateľ na koreň sa posunie na svoje menšie dieťa, uzol uložený v premennej temp2 sa vymaže a funkcia vracia nový koreň.

Pri druhej možnosti sa najprv musí ešte zavolať funkcia splayNode(), ktorá vynesie uzol temp na pozíciu ľavého dieťaťa koreňa stromu, zvyšok vyzerá rovnako.

```
Node* deleteNode(Node* myNode, int key){
    if (myNode == nullptr){
        return nullptr;
    }
    else{
        findNode(root, key);
    }
    if (root->key != key){
        return nullptr;
    }
```

```
if((root->lower == nullptr)&&(root->higher == nullptr)){
    delete root;
    return nullptr;
}
else if((root->lower == nullptr)&&(root->higher != nullptr)){
    Node* temp = root;
    root = root->higher;
    delete temp;
    root->parent = nullptr;
    return root;
}
else if((root->lower != nullptr)&&(root->higher == nullptr)){
    Node* temp = root;
    root = root->lower;
    delete temp;
    root->parent = nullptr;
    return root;
}
else if ((root->lower != nullptr)&&(root->higher != nullptr)){
    Node* temp = root->lower;
    while (temp->higher != nullptr){
        temp = temp->higher;
    }
    if (root->lower != temp){
        root->lower = splayNode(temp,root);
    }
    temp->higher = root->higher;
    root->higher->parent = temp;
    Node* temp2 = root;
    root = root->lower;
    delete temp2;
    root->parent = nullptr;
    return root;
}
```

Separate Chaining Hash tabuľka

Ako prvú hash tabuľku som si zvolil separate chaining metódu. Kolízie sa v nej riešia tak, že na každom indexe tabuľky sa nachádza štruktúra linked list. Vďaka knižnici <list> som mal prácu na tomto algoritme pomerne rýchlo hotovú, aj kód je pomerne krátky.

Hash funkcia

Pre optimalizáciu rýchlosti tabuľky som si vybral FNV-1a hash funkciu. Jej hlavnou výhodou je, že hodnoty, ktoré vracia (indexy v tabuľke), sú rozmiestnené rovnomerne po celej tabuľke, a nedochádza tak ku „clusterovaniu“.

```
int hashit(string name, int maxPocet){
    unsigned int offset=1083019949, prime = 8388817;
    unsigned int hash = offset;
    for (int i=0; i < name.length() ;i++){
        unsigned int letter = name[i];
        hash = (hash ^ letter);
        hash = (hash * prime);
    }
    return (hash%maxPocet);
}
```

Funkcia zoberie počiatočné prvočíslo, a pre každé písmeno stringu, ktorý sa hashuje, xoruje číselnú hodnotu tohto písmena s tým prvočíslom. Po každom xore ešte vynásobí výsledok s ďalším prvočíslom. Na konci vracia výslednú hodnotu modulo veľkosť tabuľky.

Insert, Search, Delete, Resize

Funkcia **addNode()** pridáva prvky do tabuľky nasledovným spôsobom:

1. zistí index prvku pomocou hashovacej funkcie
2. do pomocnej premennej nahodí dáta prvku a pridá ju do listu na danom indexe
3. Zvýši premennú, ktorá uchováva počet prvkov tabuľky
4. Ak počet prvkov presiahne stanovenú hodnotu, zavolá funkciu **resize()**

```
void addNode(string name, int value){
    int tablePlace = hashit(name, maxPocet);
    node temp;
    temp.name = name;
    temp.value = value;
    table[tablePlace].push_back(temp);
    pocet++;
    double ratio = pocet ;
    ratio = (ratio/ maxPocet);
    if (ratio >= maxLoad){
        Switch = false;
        resizeTable();
    }
}
```

Funkcia **findNode()** funguje takto:

1. Nájde index zadaného prvku v tabulke
2. Prejde list na danom indexe, ak nájde prvok vráti true
3. Ak sa prvok v danom liste nenašiel, vráti false

```
bool findNode(string name){
    int tablePlace = hashit(name, maxPocet);
    for(auto temp = table[tablePlace].begin(); (temp != table[tablePlace].end()); temp++){
        if (name == ((temp)->name)){
            return true;
        }
    }
    return false;
}
```

Funkcia **deleteNode()** funguje takýmto spôsobom:

1. zavolá funkciu **findNode(),** ak jej hodnota je false, skončí
2. Ak nie, nájde index prvku v tabuľke pomocou hashovacej funkcie
3. Ďalej prejde list na danom indexe a vymaže z neho daný prvok
4. Potom ešte zníži počet prvkov v tabuľke, ak je ich menej ako je stanovená hodnota, zavolá funkciu **resize()**

```
void deleteNode(string name) {
    if (findNode(name)==false){
        cout << "No such entry: " << name <<endl;
        return;
    }
    int tablePlace = hashit(name, maxPocet);
    for (auto it = table[tablePlace].begin(); it != table[tablePlace].end(); it++) {
        if (it->name == name) {
            table[tablePlace].erase(it);
            break;
        }
    }
    pocet--;
    double ratio = pocet ;
    ratio = (ratio/maxPocet);
    if (ratio <= minLoad){
        Switch = true;
        resizeTable();
    }
}
```

Funkcia **resize()** podľa premennej Switch buď zväčší veľkosť tabuľky na dvojnásobok alebo ju zmenší na polovicu. V oboch prípadom to prebieha následovne:

1. Vytvorí novú tabuľku o danej veľkosti
2. Prehashuje všetky prvky zo starej do novej
3. Vymaže starú tabuľku a za jej novú hodnotu nastaví novú tabuľku

```
void resizeTable(){
    if (Switch==false){
        maxPocet = (maxPocet<<1);
        list<node>* temp = new list<node>[maxPocet];
        for (int i = 0; i < (maxPocet>>1); i++) {
            for (auto temp2 = (table[i].begin()); (temp2 != (table[i].end())); temp2++) {
                int tablePlace = hashit((temp2)->name, maxPocet);
                temp[tablePlace].push_back(*temp2);
            }
        }
        delete[] table;
        table = temp;
    }
    else{
        maxPocet = (maxPocet>>1);
        list<node>* temp = new list<node>[maxPocet];
        for (int i = 0; i < (maxPocet<<1); i++) {
            for (auto temp2 = (table[i].begin()); (temp2 != (table[i].end())); temp2++) {
                int tablePlace = hashit((temp2)->name, maxPocet);
                temp[tablePlace].push_back(*temp2);
            }
        }
        delete[] table;
        table = temp;
    }
}
```

Ďalej už mám vo svojej implementácii tejto tabuľky len jednoduchú funkciu na výpis tabuľky, funkciu na deštrukciu tabuľky a main funkciu skoro totožnú s tou v mojich stromoch.

Open addressing hash tabuľka

V tejto tabuľke som sa pre optimalizáciu výkonu rozhodol kolízie riešiť zmiešaním kvadratického probovania a lineárneho probovania.

Lineárny prístup samotný by stačil, no sám o sebe je neefektívny, pretože pri ňom vzniká clustering – veľa indexov vedľa seba je zaplnených a čas nájdenia prázdneho indexu preto stúpa, ak hashovacia funkcia vráti index na začiatku alebo uprostred „clusteru“. Kvadratický probing sa tomuto problému vyhýba, lebo sa posúva vždy o druhú mocninu faktoru (tj. najprv o 1 index, potom o 4, potom o 9, atď), no pri ňom zasa na väčších datasetoch vzniká iný problém. Kvadratický probing totižto nikdy nevráti všetky indexy tabuľky kým jeho faktor nepresiahne veľkosť tabuľky, tj. kým nie je menej efektívny ako lineárny probing.

V tabuľke nižšie sa jedná o hashovaciu tabuľku o veľkosti 20 prvkov. Je tam vidno štyri stĺpce, v ktorých je úplne na vrchu náhodný index v tabuľke vrátený hashovaciou funkciou, zvýraznený zelenou farbou. Pod ním sú naľavo indexy vrátené kvadratickým probovaním a napravo faktor kvadratického probovania. Modrou farbou sú zvýraznené tie indexy, ktoré sú ***unikátne.***

0			9			10			19	
index	faktor		index	faktor		index	faktor		index	faktor
1	1		10	1		11	1		0	1
5	2		14	2		15	2		4	2
14	3		3	3		4	3		13	3
10	4		19	4		0	4		9	4
15	5		4	5		5	5		14	5
11	6		0	6		1	6		10	6
0	7		9	7		10	7		19	7
4	8		13	8		14	8		3	8
5	9		14	9		15	9		4	9
5	10		14	10		15	10		4	10
6	11		15	11		16	11		5	11
10	12		19	12		0	12		9	12
19	13		8	13		9	13		18	13
15	14		4	14		5	14		14	14
0	15		9	15		10	15		19	15
16	16		5	16		6	16		15	16
5	17		14	17		15	17		4	17
9	18		18	18		19	18		8	18
10	19		19	19		0	19		9	19

Z tohoto je vidno, že v prvých šiestich iteráciach sa vždy vráti unikátny index, potom už sa indexy začínajú opakovať. Neskôr sa ešte niekoľko unikátnych indexov vráti, ale menej často a aj tak sa nikdy nevrátia všetky.

Na veľa dátach sa preto môže stať (aj sa mi to stalo), že všetky indexy vrátené kvadratickým probingom sú už obsadené a faktor sa preto dokola zvyšuje až na čísla v miliónoch, ktorých druhá mocnina je tak veľká, že typ integer „pretečie“ na zápornú hodnotu, čo má za následok segmentation fault. Tento problém nevyriešilo ani pretypovanie indexovej premennej na *unsigned long long*, čo v praxi potvrdzuje fakt, že kvadratický probing jednoducho niektoré indexy nikdy nevráti a ak všetky, ktoré vracia sú už obsadené, algoritmus nevie, čo má ďalej robiť a opakuje sa do nekonečna.

0	0	1	-	-	4	5	6	-	-	9	10	11	-	-	14	15	16	-	-	19
9	0	-	-	3	4	5	-	-	8	9	10	-	-	13	14	15	-	-	18	19
10	0	1	-	-	4	5	6	-	-	9	10	11	-	-	14	15	16	-	-	19
19	0	-	-	3	4	5	-	-	8	9	10	-	-	13	14	15	-	-	18	19

V tejto tabuľke sú pre lepšie pochopenie farebne vyznačené poznatky z tabuľky vyššie.

Zeleným sú označené východiskové indexy z obrázka vyššie, červené pomlčky označujú indexy, ktoré kvadratický probing nevráti.

Keďže kvadratický probing má najväčšiu efektivitu na prvých šiestich faktoroch (do 30% veľkosti tabuľky), kde vždy vracia nový index, rozhodol som sa môj probing zložiť nasledovne:

1. Najprv sa indexy hľadajú kvadratickým probingom
2. Ak faktor kvad. probingu presiahne 30% veľkosti tabuľky, prejdem na lineárny probing

Aj napriek tomu, že hodnota 30% je odvodená z tabuľky o veľkosti iba 20 prvkov, preukázala sa optimálna aj pritestovaní na tabuľke o veľkosti 10 miliónov prvkov – toto som testoval jednoducho tak že som skúšal rôzne hodnoty premennej chokePoint a meral čas, za ktorý sa vložilo 10M prvkov:

```
const float maxLoad = 0.75;
const float minLoad = 0.2;
const float chokePoint = 0.3;
```

Takouto optimalizáciou som dosiahol dokonca lepšie časy ako pri mojej Separate chaining hash tabuľke, a to je minimálne spomedzi mojich spolužiakov výnimkou.

Tento algoritmus by sa dal ešte vylepšiť. V druhej tabuľke si môžme všimnúť, že indexy, ktoré kvadratický probing nikdy nevráti, sú zoskupené do akýchsi „medzier“, ktoré majú konštantnú veľkosť a konštantnú vzdialenosť jedna od druhej. Optimálnym riešením namiesto tvrdého presedlania na lineárny probing by preto bolo vymyslieť taký algoritmus, ktorý by:

1. Našiel najbližšiu takú „medzeru“ a pozrel či v nej nie je voľný index
2. ak nie, vypočítal by, aký veľký „skok“ musí urobiť, aby sa presunul na ďalšiu „medzeru“
3. Takto skákal po „medzerách“, až kým by v nich nenašiel voľný index

Takéto riešenie by bolo najefektívnejšie, lebo by predišlo tomu, aby sa v druhom kroku probovania lineárne pozerali tie isté hodnoty, ktoré už kvadratický probing kontroloval, no vzhľadom

na časové obmedzenie tohto zadania som sa k tomu už nedostal, keďže vymyslieť taký algoritmus pre všeobecne veľkú tabuľku rozhodne nie je triviálny problém. Pre účely tohto zadania som sa preto rozhodol pokračovať s jednoduchým presedlaním na lineárny probing, aj keď v budúcnosti rozhodne plánujem skúsiť prísť na ešte optimálnejšie riešenie, nakoľko ma táto oblasť veľmi zaujíma.

Hash funkcia, Class Node

Hashovaciu funkciu som použil tú istú ako pri prvej hashovacej tabuľke, no prvok Node som si pretypoval zo štruktúry na triedu, aby som mohol využiť možnosti konštruktorov triedy, ktoré sa mi pri tejto implementácii zišli.

```
class node{
    public:
        string name;
        int value;
    node(string inputName, int inputValue){
        this->name = inputName;
        this->value = inputValue;
    }
    node (){
        name = "deleted";
    }
};
```

Insert, Search, Delete, Upscale, Downscale

Funkcia **addNode()** funguje nasledovne:

1. Hashovaciou funkciou zistí index pre zadané dáta
2. Ak je index v tabuľke voľný, pridá tam zadané dáta
3. Ak nie, spomenutým zmiešaným probingom nájde najbližší voľný index a vloží dáta
4. Zvýši počet prvkov v tabuľke a prípadne zavolá funkciu **upScaleTable(),** ktorá tabuľku zväčšuje (– v tomto programe som **resize()** funkciu rozdelil na dve)

```
void addNode(string name, int value){
    unsigned int index = hashit(name);
    if (table[(index%tableSize)]==nullptr){
        table[(index%tableSize)] = new node(name, value);
    }
    else{
        unsigned int j = 1;
        while (table[(index%tableSize)]!=nullptr){
            index = (index + (j*j));
            j++;
            if ( j >  chokePoint){
                unsigned int k = 1;
                while (table[(index%tableSize)]!=nullptr){
                    index  = (index + k);
                    k++;
                }
            }
        }
        table[(index%tableSize)]= new node(name, value);
    }
    elementCount++;
    double ratio = elementCount ;
    ratio = (ratio / tableSize);
    if (ratio >= maxLoad){
        //cout << "scaling up\n";
        upScaleTable();
    }
    return;
}
```

Funkcia **findNode()** funguje podobne ako pri prvej tabuľke, až na to, že ak hľadaný prvok nie je na indexe, ktorý vracia hashovacia funkcia, prvok sa ďalej hľadá pomocou rovnakého probingu, akým sa prvky vkladajú.

```
bool findNode(string name){
    unsigned int index = hashit(name);
    unsigned int j = 1;
    if (table[(index%tableSize)]->name==name){
        return true;
    }
    else{
        while ((table[(index%tableSize)]!=nullptr)&&(table[(index%tableSize)]->name!=name)){
            index = (index + (j*j));
            j++;
            if ( j >  chokePoint){
                unsigned int k = 1;
                while ((table[(index%tableSize)]!=nullptr)&&(table[(index%tableSize)]->name!=name)){
                    index  = (index + k);
                    k++;
                }
            }
        }
    }
    if (table[(index%tableSize)]->name == name){
        return true;
    }
    else{
        return false;
    }
}
```

Funkcia **deleteNode()** funguje podobne ako **findNode(),** s tým rozdielom, že keď nájde zadaný prvok, tak ho vymaže. Ak ho nenájde, tiež vracia hodnotu false.

```
bool deleteNode(string name){
    unsigned int index = hashit(name);
    unsigned int j = 1;
    if (table[(index%tableSize)]->name==name){
        delete table[(index%tableSize)];
        table[(index%tableSize)] = new node();
        elementCount--;
        double ratio = elementCount;
        ratio = (ratio / tableSize);
        if (ratio <= minLoad){
            //cout << "scaledown\n";
            downScaleTable();
        }
        return true;
    }
    else{
        while ((table[(index%tableSize)]!=nullptr)&&(table[(index%tableSize)]->name!=name)){
            index = (index + (j*j));
            j++;
            if ( j >  chokePoint){
                unsigned int k = 1;
                while ((table[(index%tableSize)]!=nullptr)&&(table[(index%tableSize)]->name!=name)){
                    //cout << "zapajam linear\n";
                    index  = (index + k);
                    k++;
                }
            }
        }
    }
```

Pokračovanie funkcie deleteNode()

```
if ((table[(index%tableSize)] != nullptr)&&(table[(index%tableSize)]->name == name)){
    delete table[(index%tableSize)];
    table[(index%tableSize)] = new node();
    elementCount--;
    double ratio = elementCount;
    ratio = (ratio / tableSize);
    if (ratio <= minLoad){
        //cout << "scaledown\n";
        downScaleTable();
    }
    return true;
}
else{
    return false;
}
```

Funkcia **upScaleTable()** tabuľku zväčšuje a funguje veľmi podobne ako pri prvej hashovacej tabuľke, ale na prehashovanie hodnôt zo starej tabuľky do novej pochopiteľne používa spomenutý zmiešaný probovací algoritmus. Funkcia **downScaleTable()** robí to isté, len tabuľku zmenšuje.

```
void upScaleTable(){
    int newTableSize = (tableSize*2);
    node** tempTable = new node*[newTableSize];
    for (int i = 0; i < newTableSize; i++){
        tempTable[i] = nullptr;
    }
    for (int i = 0; i < tableSize; i++){
        if ((table[i]!=nullptr)&&((table[i]->name) != "deleted")){
            unsigned int index = hashit(table[i]->name);
            if (tempTable[(index%newTableSize)]==nullptr){
                tempTable[(index%newTableSize)] = new node(table[i]->name, table[i]->value);
            }
            else{
                unsigned int j = 1;
                while (tempTable[(index%newTableSize)]!=nullptr){
                    index = (index + (j*j));
                    j++;
                    if ( j >  chokePoint){
                        unsigned int k = 1;
                        while (tempTable[(index%newTableSize)]!=nullptr){
                            index  = (index + k);
                            k++;
                        }
                    }
                }
                tempTable[(index%newTableSize)]= new node(table[i]->name, table[i]->value);
            }
        }
    }
    delete[] table;
    table = tempTable;
    tableSize = newTableSize;
    return;
}
```

Main funkcia a výpis sú opäť totožné.

Testovanie

Testoval som tri scenáre:

1. Insert
2. Insert – Search
3. Insert – Search – Delete

Každý scenár sa odohral na datasetoch o veľkostiach 10 až 10^7 prvkov, ktoré som náhodne vygeneroval a uložil do textových súborov. Textové súbory s datasetmi som nahral na google drive, link je na konci dokumentácie pri zdrojoch. Každý test som vykonal 5 krát a z toho som vyrobil priemerné časy. Z týchto priemerných časov som potom ťahal informácie do grafov pre ilustráciu výsledkov.

Insert

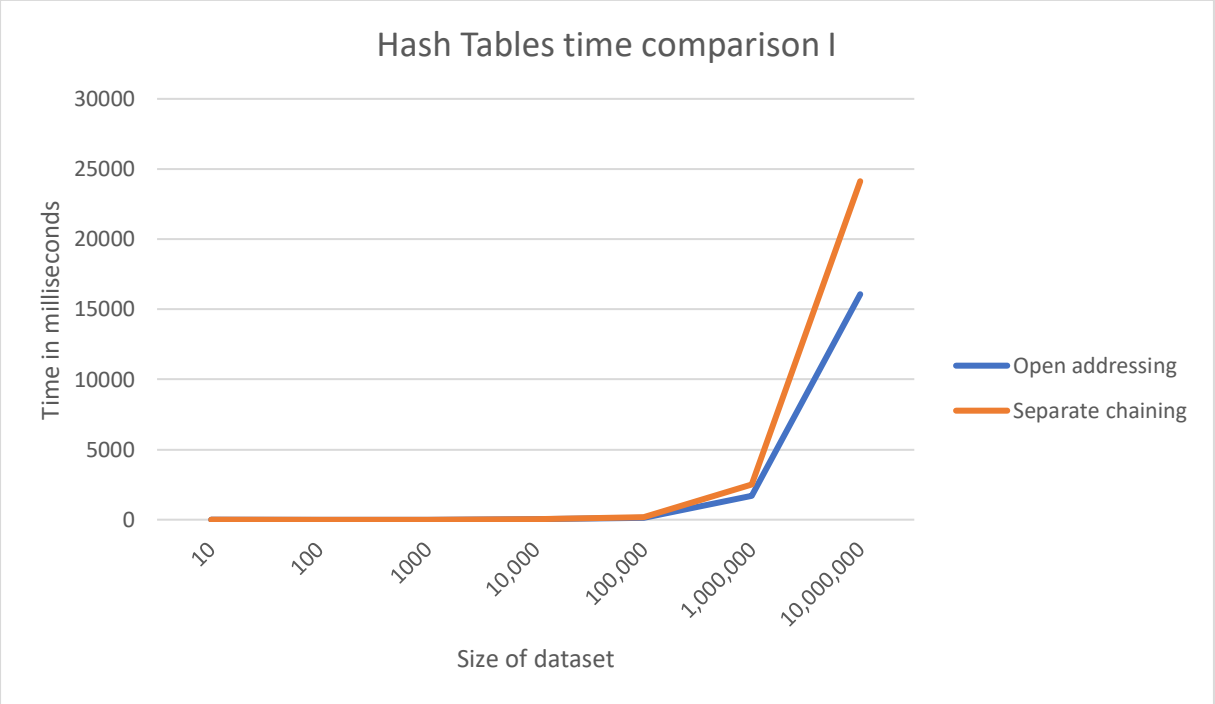
Tento scenár sa odohráva tak, že sa jednoducho z daného datasetu vložia všetky dáta do jednotlivých dátových štruktúr.

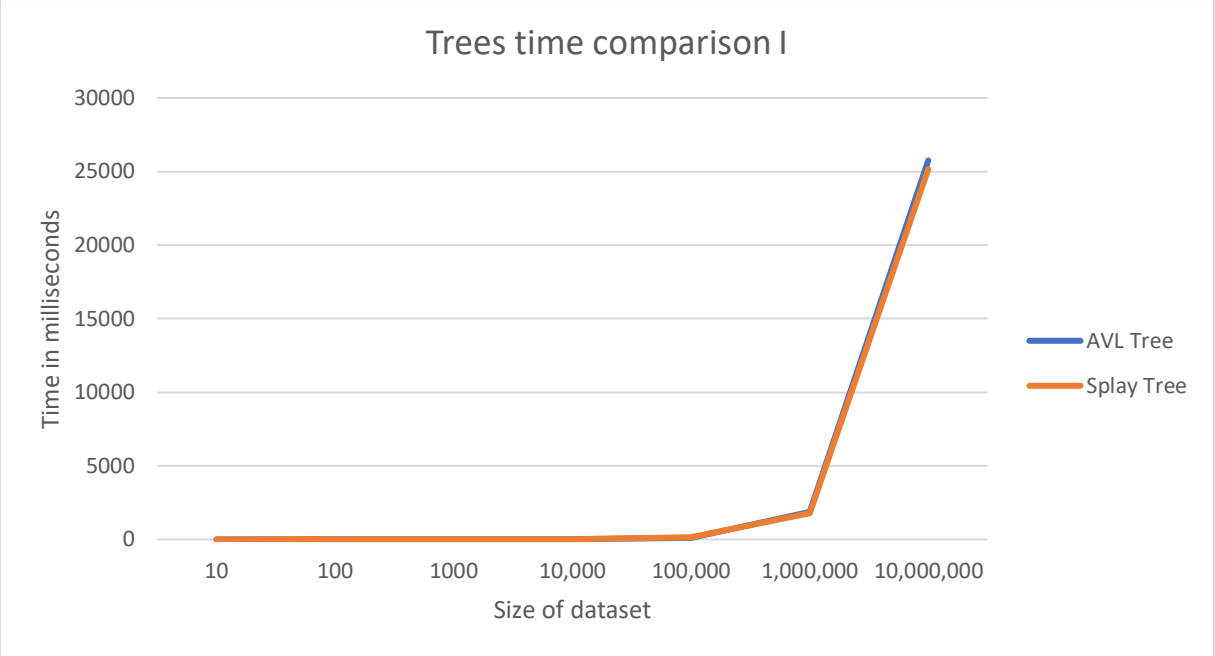
Open adressing hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,0836	0,109	0,086	0,083	0,072	0,068
100	0,1934	0,19	0,18	0,195	0,204	0,198
1000	1,4696	1,48	1,481	1,446	1,467	1,474
10,000	13,2086	13,16	13,01	13,197	12,985	13,691
100,000	138,4266	137,342	145,655	135,319	132,57	141,247
1,000,000	1719,8	1698	1663	1790	1661	1787
10,000,000	16072,4	16201	16365	15826	15829	16141

Separate Chaining hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,074	0,080	0,089	0,064	0,073	0,065
100	0,218	0,238	0,220	0,209	0,211	0,214
1000	2,018	2,017	2,019	1,999	2,024	2,031
10,000	18,637	18,494	19,119	18,574	18,487	18,509
100,000	187,835	182,655	185,347	189,275	185,037	196,859
1,000,000	2508,4	2329	2743	2449	2569	2452
10,000,000	24127,2	23405	24666	24174	24027	24364

AVL Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,068	0,068	0,064	0,08	0,063	0,065
100	0,1456	0,142	0,143	0,148	0,142	0,153
1000	0,9562	0,958	0,961	0,94	0,979	0,943
10,000	10,5038	10,507	10,649	10,473	10,523	10,367
100,000	130,3326	130,932	137,343	129,685	129,479	124,224
1,000,000	1856,6	1783	1825	1769	2079	1827
10,000,000	25754,2	26122	26743	25716	25076	25114

Splay Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,077	0,079	0,06	0,073	0,094	0,077
100	0,145	0,129	0,126	0,167	0,143	0,161
1000	0,972	1,021	0,962	1,004	0,94	0,933
10,000	10,790	11,544	10,445	10,276	11,24	10,446
100,000	131,366	134,957	137,36	128,406	132,596	123,511
1,000,000	1776	1802	1728	1753	1725	1872
10,000,000	25168	25307	26575	23997	25588	24373





Insert - Delete

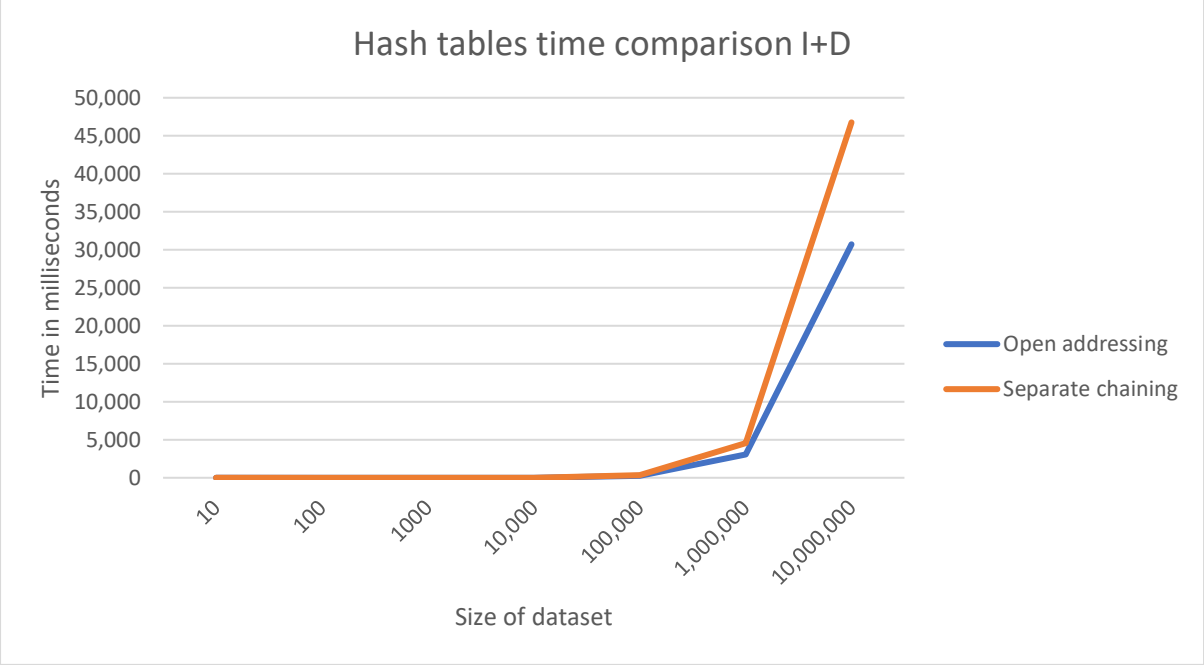
V tomto scenári sa najprv vložia všetky dáta z datasetu a potom sa všetky vymažú.

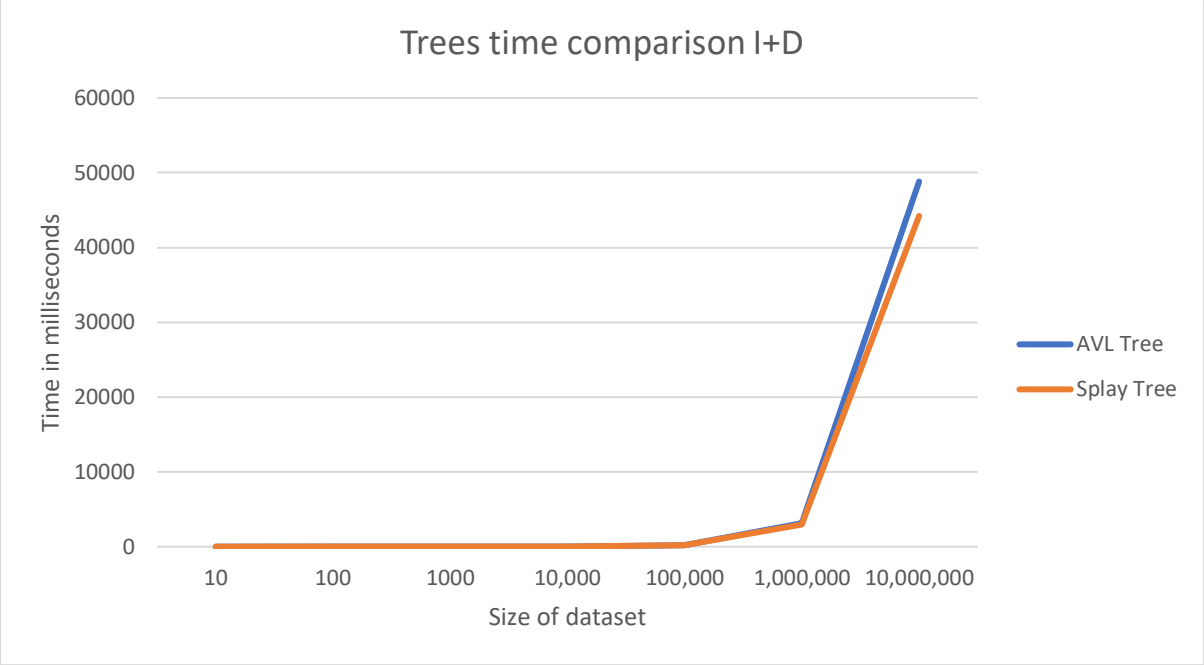
Open adressing hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,106	0,106	0,105	0,105	0,109	0,106
100	0,353	0,480	0,291	0,387	0,297	0,311
1000	2,577	2,639	2,608	2,549	2,593	2,498
10,000	24,857	24,377	23,875	23,294	27,346	25,395
100,000	261,883	247,327	282,542	263,184	264,103	252,259
1,000,000	3 077	3043	3159	3058	3049	3076
10,000,000	30 717,6	32735	31723	29919	29648	29563

Separate Chaining hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,115	0,112	0,103	0,124	0,109	0,127
100	0,373	0,358	0,392	0,369	0,374	0,372
1000	3,6968	3,649	3,721	3,643	3,622	3,849
10,000	34,31	32,167	34,933	34,248	36,318	33,884
100,000	347,8976	341,28	369,937	335,015	351,812	341,444
1,000,000	4557,4	4458	4574	4705	4465	4585
10,000,000	46761,8	46471	46976	46435	47109	46818

AVL Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,087	0,096	0,095	0,081	0,084	0,080
100	0,194	0,207	0,180	0,191	0,177	0,213
1000	1,406	1,414	1,409	1,399	1,397	1,411
10,000	16,578	16,008	16,773	16,271	15,982	17,855
100,000	215,565	218,127	212,213	218,445	212,709	216,333
1,000,000	3109,6	3085	3150	3108	3136	3069
10,000,000	48826	47733	48280	49386	50070	48661

Splay Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,109	0,119	0,111	0,154	0,081	0,081
100	0,203	0,187	0,199	0,222	0,195	0,212
1000	1,452	1,435	1,431	1,46	1,481	1,453
10,000	18,649	19,865	19,347	17,454	17,78	18,799
100,000	218,389	210,342	216,714	224,792	217,256	222,84
1,000,000	2977,8	2949	2986	2938	3019	2997
10,000,000	44227,6	43737	43920	43491	43808	46182





Insert - Search- Delete

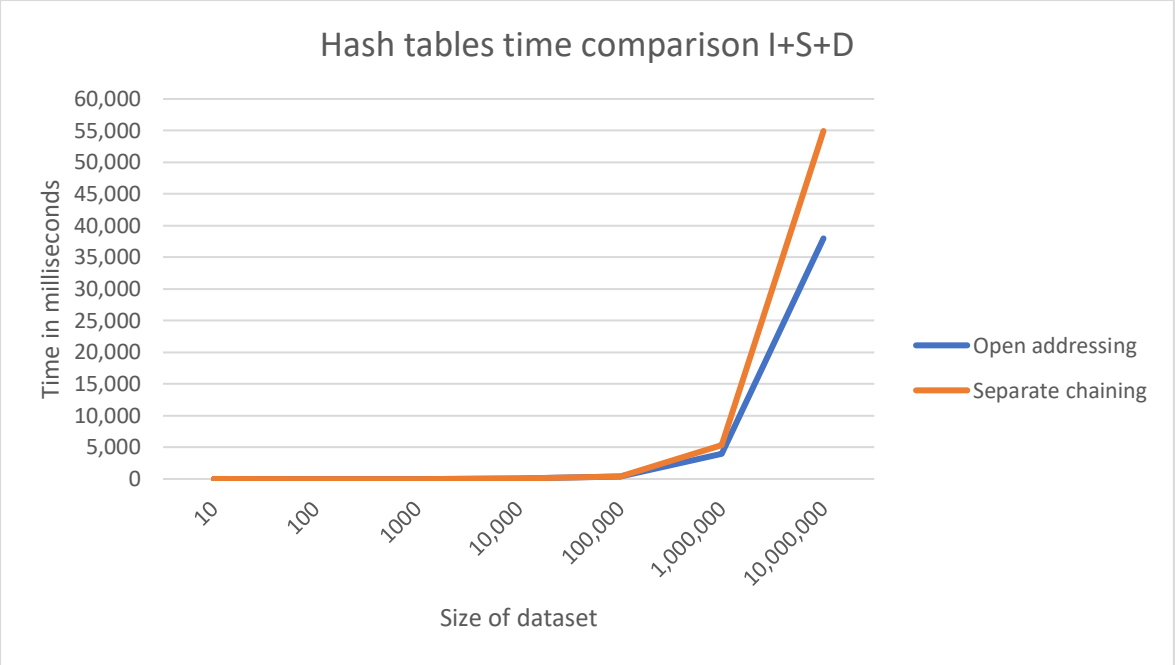
V tomto scenári sa medzi vložením a vymazaním prvkov ešte navyše všetky po jednom vyhľadajú.

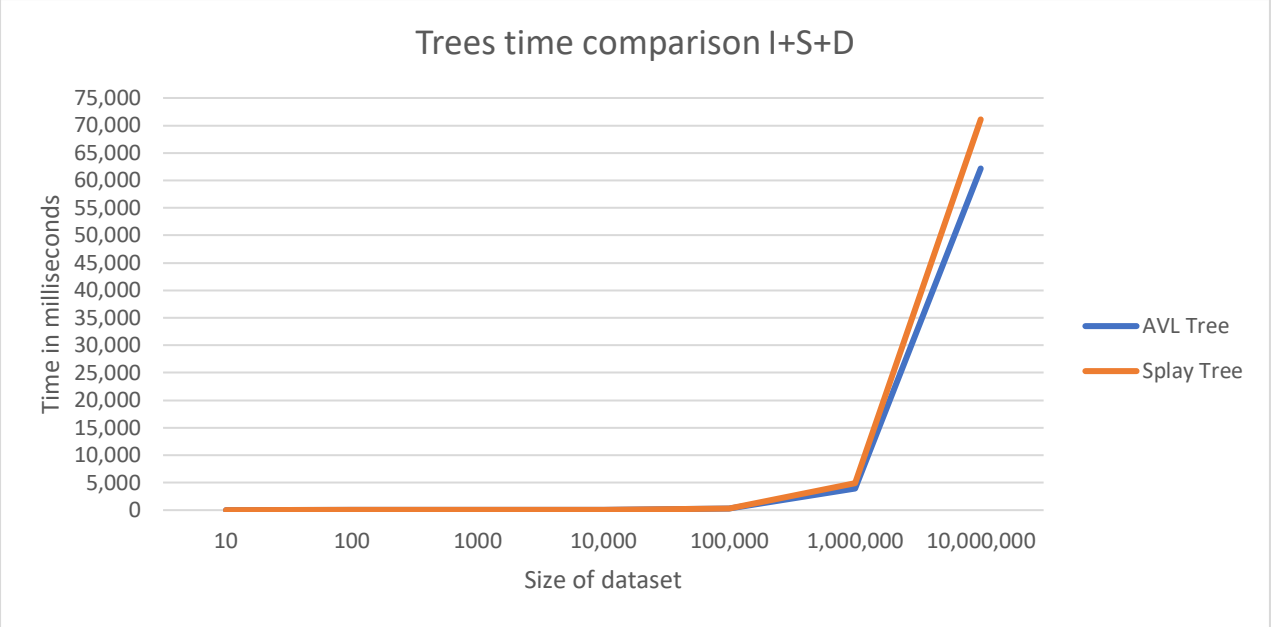
Open adressing hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,129	0,120	0,107	0,172	0,130	0,118
100	0,369	0,375	0,354	0,355	0,387	0,374
1000	3,160	3,148	3,150	3,183	3,243	3,075
10,000	34,196	36,959	34,608	31,662	30,192	37,560
100,000	338,099	314,378	351,963	316,625	379,978	327,551
1,000,000	3927,2	3967	3918	4018	3837	3896
10,000,000	37979	37773	38963	37178	38754	37227

Separate Chaining hashtable						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,114	0,110	0,096	0,129	0,125	0,109
100	0,435	0,416	0,443	0,429	0,441	0,448
1000	4,457	4,545	4,284	4,559	4,541	4,354
10,000	41,359	38,773	40,840	44,638	40,418	42,124
100,000	417,327	374,761	429,299	435,310	447,457	399,806
1,000,000	5389,4	5205	5343	5453	5255	5691
10,000,000	54942	54934	55371	54670	54755	54980

AVL Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,104	0,098	0,108	0,107	0,098	0,109
100	0,2284	0,235	0,23	0,223	0,226	0,228
1000	1,7204	1,745	1,76	1,71	1,682	1,705
10,000	20,3462	20,593	20,647	20,033	19,378	21,08
100,000	260,1048	255,067	257,666	260,367	262,299	265,125
1,000,000	3921,8	3971	3901	3931	3896	3910
10,000,000	62171	63201	62143	61884	62803	60824

Splay Tree						
	Time in miliseconds					
Dataset	Average	Run 1	Run 2	Run 3	Run 4	Run 5
10	0,101	0,101	0,114	0,087	0,096	0,107
100	0,2306	0,234	0,223	0,225	0,239	0,232
1000	2,0236	2,002	2,053	2,006	1,998	2,059
10,000	25,6348	25,506	25,493	25,107	25,867	26,201
100,000	338,9634	326,519	335,503	334,302	330,227	368,266
1,000,000	4851,6	4645	4670	5083	4767	5093
10,000,000	71100,6	70966	72956	69975	71359	70247





Záver

V tejto časti zhrniem poznatky z testov, ktoé som vykonal na mojich algoritmoch. Vzájomné porovnávanie implementácii som rozdelil na stromy a hashovacie tabuľky.

Stromy

V prvom scenári dosahovali obidva stromy v podstate rovnaké časy na všetkých veľkostiach datasetov. V druhom scenári bol Splay na 10 milióch dát v priemere o čosi rýchlejší, no hodnoty boli celkovo stále veľmi podobné. Najväčší rozdiel bol v treťom scenári, kde bol v priemere AVL strom na 10 miliónoch dát rýchlejší o skoro 10 sekúnd. Tento rozdiel je následkom toho, že pri vyhľadávaní prvkov v stromoch sa Splay algoritmus zdržal tým, že všetky postupne splayoval na koreň stromu, zatiaľ čo AVL pri hľadaní žiadne ďalšie funkcie nevolal.

Vzhľadom na charakteristku oboch algoritmov a spôsob testovania nie sú tieto výsledky veľmi prekvapivé. Keďže som testoval pomocou náhodne generovaných dát, a nevyhľadával som niektoré prvky viackrát ale všetky iba raz, silná stránka Splay algoritmu nemohla tak dobre vyniknúť.

Treba však poznamenať, že jediné rozdiely, ktoré sa medzi nimi prejavili, sa ukázali až na datasete o veľkosti 10 miliónov dát, dovtedy boli ich časy prakticky identické naprieč všetkými scenármi.

Hash tabuľky

V prvom scenári scenári boli obidve tabuľky rovnako rýchle až po 100-tisícový dataset, pozorovateľnejší rozdiel nastal až na miliónovom datasete a potvrdil sa na 10 miliónovom.
V obidvoch týchto prípadoch bola moja Open Addressing tauľka priemerne rýchlejšia o zhruba 33%, čo na najväčšom datasete predstavovalo v priemere až 8 sekúnd.

V druhom a treťom scenári boli výsledky prakticky rovnaké ako pri prvom, testovanie hashovacích tabuliek preto zjavne vyhrala moja Open Addressing tabuľka. Tento výsledok taktiež nie je veľmi prekvapivý, nakoľko s jej optimalizáciou som si dal naozaj záležať a vyťažil som to najlepšie zo známych alogrotimov probovania. Možným dôvodom horšieho výkonu Separate chaining tabuľky na väčších datasetoch je zvýšený počet kolízii, ktoré potom vedú k dlhším listom. Linked listy sú neefektívne z hľadiska prístupu ku konkrétnemu prvku a preto sa jej čas pri testoch predlžoval.

Zdroje

Zdroje z ktorých som čerpal informácie na pochopenie dátových štruktúr, ktoré som si vybral:

1. https://www.geeksforgeeks.org/list-cpp-stl/
2. https://docs.google.com/presentation/d/11uQIvDZZdD8kY-pdTTmgsfwI9LZtwurd0kl-Wx18fhI/edit#slide=id.g1de0d66c01a_7_58
3. https://www.geeksforgeeks.org/quadratic-probing-in-hashing/
4. https://www.geeksforgeeks.org/bidirectional-iterators-in-cpp/
5. https://www.geeksforgeeks.org/this-pointer-in-c/
6. https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial/?ref=footer

Odkaz na google drive s testovacími súbormi, prístup je povolený pre STU účty.