Richard Šléher
ID: 120890

Dátové štruktúry a algoritmy

Zadanie č. 1

Richard Šléher
ID: 120890

Obsah

Richard Šléher
ID: 120890

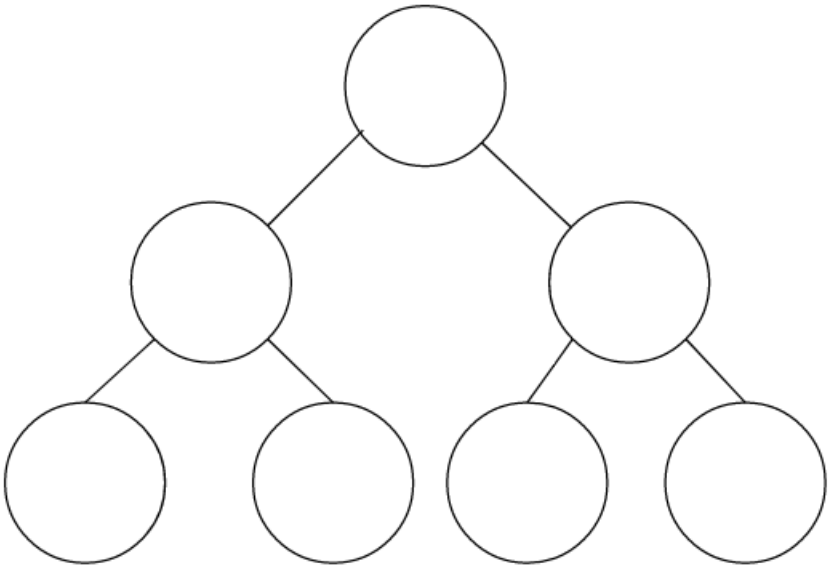
1. Binárne vyhľadávacie stromy

1.1 Binárny vyhľadávací strom

Binárny vyhľadávací strom je dátová štruktúra, ktorá sa riadi pravidlami na uchovávanie dát. Ak je vstupná hodnota menšia ako rodičovského uzla tak sa hodnota pridá do ľavého uzla, ak je väčšia tak do pravého. Na uzol môžu byť maximálne napojené dva detské uzly. Uzol ktorý nemá ani jeden nazývame list.

1.2 AVL strom

AVL strom využíva samovyvažovanie na základe vyvažovacieho faktoru, ktorý sa získava rozdielom ľavého a pravého podstromu. To znamená, že uzly musia obsahovať informáciu navyše a tou je výška. Na vyrovnanie vyvažovacieho faktoru sa používajú rotácie. Môžu nastať 4 situácie otáčania podľa stavu uzlov.



1.3 Vlastná implementácia AVL stromu

Insert:

Insert pre nový uzol sa vyhľadáva rekurzívne. Ak je hodnota menšia presunieme sa vľavo ak je väčšia postupujeme vpravo až kým nenarazíme miesto s hodnotou nullptr. Ak strom ešte nemá root tak sa vytvorí volaním metódy new_node(). Po vytvorení sa strom vyrovnáva ak pridanie nového uzla pokazilo vyvažovací faktor.

Richard Šléher
ID: 120890

```
//rekurzivny alg na pridavanie prvkov do stromu
node *insert_node(node *root, int value, string word) {

    if (root == nullptr) {
        return new_node(value, word);
    } else if (value < root->value) {

        root->left = insert_node(root->left, value, word);
        root->height = max(height_node(root->left), height_node(root-
>right)) + 1;

        if (balance(root) > 1) {
            //left left
            if (value < root->left->value) {
                return right_rotate(root);
            }
                //left right
            else {
                root->left = left_rotate(root->left);
                return right_rotate(root);
            }
        }
    } else if (value > root->value) {

        root->right = insert_node(root->right, value, word);
        root->height = max(height_node(root->left),
        height_node(root>right)) + 1;

        if (balance(root) < -1) {
            //right right
            if (value > root->right->value) {
                return left_rotate(root);
            }
                //right left
            else {
                root->right = right_rotate(root->right);
                return left_rotate(root);
            }
        }
    } else if (value == root->value) {
        return root;
    }
    return root;
}
```

Search:

Strom sa prechádza iteratívne a to tak, že sa porovnávajú hodnoty uzlov so zadanou hodnotou až kým sa nenájde uzol.

Richard Šléher
ID: 120890

```
//iterativny sposob na najdenie prvku v strome
node *search_node(node *root, int value) {

    while (root != nullptr) {

        if (root->value == value) {
            //cout << word << endl;
            return root;
        } else if (root->value < value) {
            root = root->right;
        } else if (root->value > value) {
            root = root->left;
        }
    }
    return root;
}
```

Delete:

Rekurzívne sme našli uzol, ktorý chceme zmazať a potom sme zisťovali koľko child uzly má. Ak mal nula tak sme ho iba vymazali. Ak mal iba jeden tak sme zistili či ľavý alebo pravý a napojili sme ho na rodičovský mazaného. Ak má dva child uzly tak nájdeme najmenší väčší uzol a nahradíme ho.

```
//rekurzivne mazanie
node *delete_node(node *root, int value) {

    node *tmp;

    if (root == nullptr) {
        return nullptr;
    } else if (value < root->value) {
        root->left = delete_node(root->left, value);
    } else if (value > root->value) {
        root->right = delete_node(root->right, value);
    } else {
        //ak ma 2 child nody
        if ((root->left != nullptr) && (root->right != nullptr)) {
            tmp = successor_node(root->right);
            root->value = tmp->value;
           root->word = tmp->word;
            root->right = delete_node(root->right, root->value);

        }
            //ak ma aspon 1 child node
        else if ((root->left == nullptr) || (root->right == nullptr)) {

            tmp = root;
            //ak je Lchild null tak Rchild sa prida do root a naopak, ak su
obe null tak sa iba vymaze
            if (root->left == nullptr) {
                root = root->right;
            } else if (root->right == nullptr) {
                root = root->left;
            }
            delete tmp;
        }
    }
    //ak je prazdny
    if (root == nullptr) {
        return root;
    }

    root->height = max(height_node(root->left), height_node(root->right)) +
1;

    //vybalancovanie
    if (balance(root) > 1) {
        if (balance(root->left) > 0) {
            return right_rotate(root);
        } else {
            root->left = left_rotate(root->left);
            return right_rotate(root);
        }
    }
    if (balance(root) < -1) {
        if (balance(root->right) < 0) {
            return left_rotate(root);
        } else {
            root->right = right_rotate(root->right);
            return left_rotate(root);
        }
    }
```

Richard Šléher
ID: 120890

1.4 Splay vyhľadávací strom

Splay strom je tiež samovyvažovací strom. Je unikátny funkciou splay, ktorá umiestňuje uzol a vrch stromu čiže na miesto rootu. Splayovanie nastáva vždy keď pridaný, hľadaný alebo mazaný. Tento algoritmus je výhodný hlavne pri stromoch v ktorých potrebujeme prístup k rovnakým uzlom.

1.5 Implementácia Splay vyhľadávacieho stromu

Splaying:

Funkcia hľadá hodnotu rekurzívne pričom volá ľavý alebo pravý child node. Keď je hľadana hodnota nájdená vykonajú sa rotácie, ktoré posúvajú node na miesto rootu.

```
node *splaying(node *root, int value) {

    //ak je strom prazdny alebo je hodnota uz v roote
    if (!root|| root->value == value)
        return root;

    //lava vetva
    if (root->value > value) {
        //ak nie je v strome
        if (!root->left)
            return root;

        //left-left
        if (root->left->value > value) {
            root->left->left = splaying(root->left->left, value);
            root = right_rotate(root);

        }
        //left-right
        else if (root->left->value < value) {
            root->left->right = splaying(root->left->right, value);
            if (root->left->right)
                root->left = left_rotate(root->left);
        }
        //rotacia pre root
        if (!root->left) {
            return root;
        }
        return right_rotate(root);
    }

    //prava vetva
    else {
        //ak nie je v strome
        if (!root->right)
            return root;

        //right-left
        if (root->right->value > value) {
            root->right->left = splaying(root->right->left, value);

            //rotacia pre root
            if (root->right->left)
                root->right = right_rotate(root->right);

        }
        //right-right
```

```
        else if (root->right->value < value) {
            root->right->right = splaying(root->right->right, value);
            root = left_rotate(root);
        }

        //rotacia pre root
        if (!root->right) {
            return root;
        }
        return left_rotate(root);

    }
}
```

Insert:

Zavoláme splayovanie a porovnáme hodnotu rootu a inputu. Následne pridáme nový node na miesto rootu a nastavíme mu left a right node

```
node *insert_node(node *root, int value, string word) {

    if (root == nullptr) {
        return new_node(value, word);
    }

    //splayovanie
    root = splaying(root, value);

    if (value == root->value) {
        return root;
    }

    node *new_root = new_node(value, word);

    //pripojenie rootu pod novy root
    if (root->value > value) {
        new_root->right = root;
        new_root->left = root->left;
        root->left = nullptr;
    }

    else {
        new_root->left = root;
        new_root->right = root->right;
        root->right = nullptr;
    }

    return new_root;
}
```

Search:

Iteratívne hľadáme a keď nájdeme hľadanú hodnotu.

```
node *search_node(node *root, int value) {

    while (root != nullptr) {

        if (root->value == value) {
            //cout << word << endl;
```

```
            return root;
        } else if (root->value < value) {
            root = root->right;
        } else if (root->value > value) {
            root = root->left;
        }
    }

    return root;
}
```

Delete:

Skontroluje či root nie je prázdny ak nie zavolá funkciu na splayovanie. Na mieste root sa uloží hodnota, ktorú chceme odstrániť. Potom sa skontroluje koľko child nodov root má. Ak má jeden tak jeho opačný uloží do tmp a nájdený odstráni. Ak má dva tak oba napojí a potom odstráni nájdený.

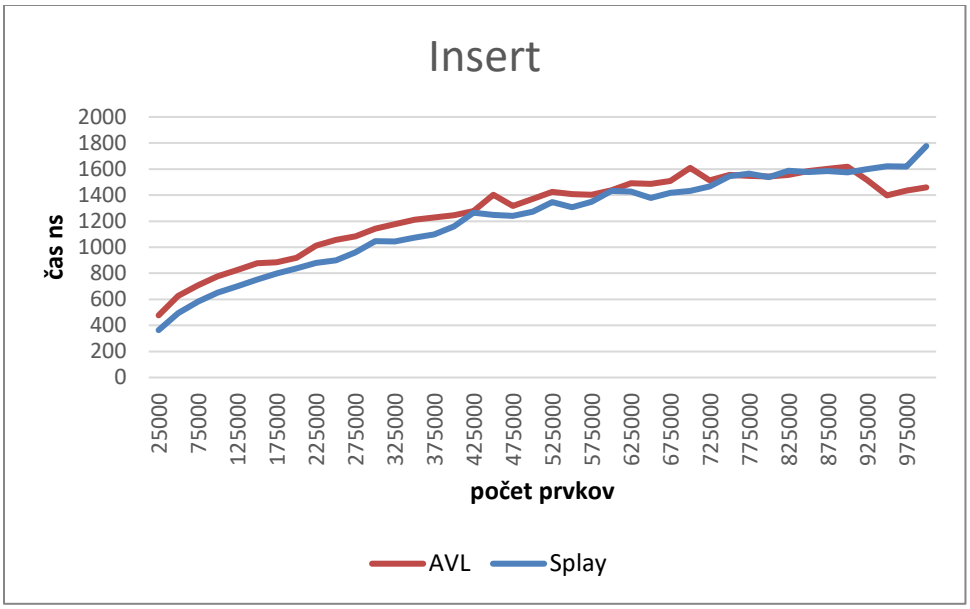
```
node *delete_node(node *root, int value) {
    node *tmp;

    if (root == nullptr) {
        return nullptr;
    }

    root = splaying(root, value);

    if (root->value != value) {
        return root;
    }

    //no right child
    if (root->right == nullptr) {
        tmp = root->left;
        delete root;
        return tmp;
    }
    //no left child
    else if (root->left == nullptr) {
        tmp = root->right;
        delete root;
        return tmp;
    }
    //both children
    else {
        tmp = root;
        root = splaying(root->left, value);
        root->right = tmp->right;
        delete tmp;
        return root;
    }
}
```

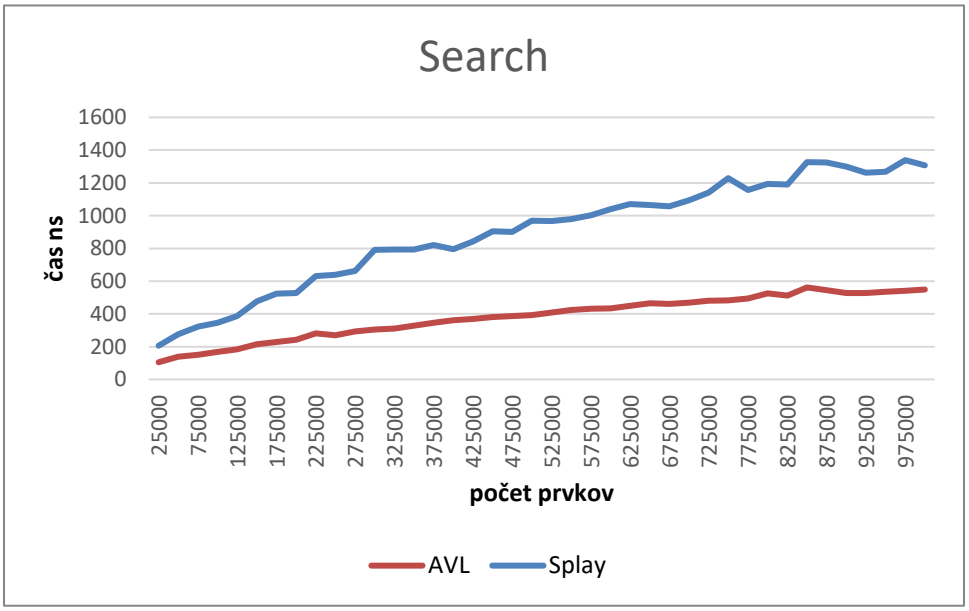
1.6 Porovnanie binárnych vyhľadávacích stromov

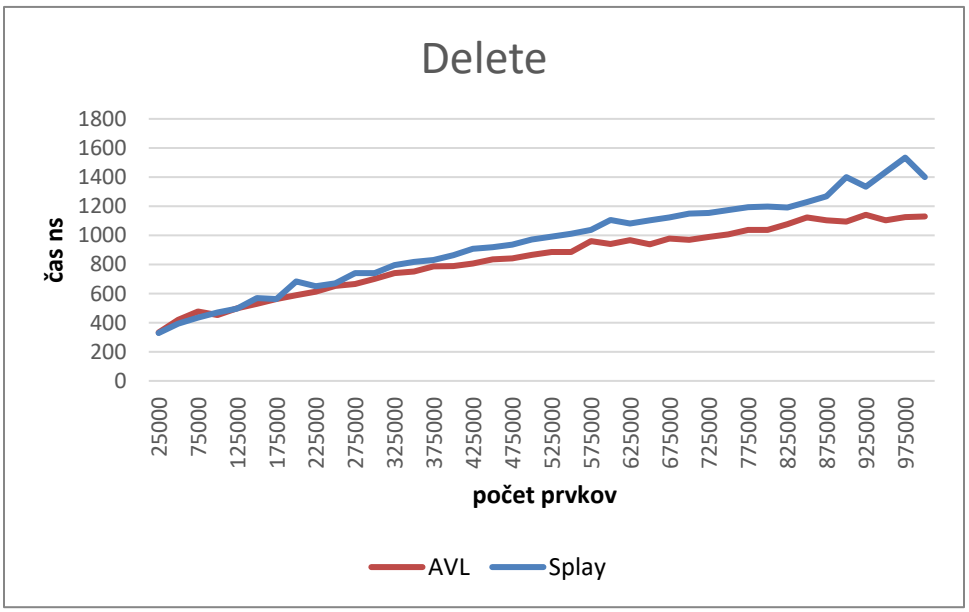
Porovnávať budeme rýchlosť vkladania, hľadania a mazania prvkov v binárnych vyhľadávacích stromoch.

Hodnoty sú merané v nanosekundách. Grafy znázorňujú priemernú rýchlosť jedného prvku vo zvyšujúcom sa počte prvkov.

Richard Šléher
ID: 120890







Richard Šléher
ID: 120890

Z nasledovných grafov si môžeme všimnúť, že AVL implementácia je rýchlejšia pri hľadaní aj mazaní a pri vkladaní sa implementácie vyrovnajú. Rozdiel pri hľadaní je spôsobený tým, že v Splay implementácii musíme nájsť prvok a potom ho ešte funkciou splayovania dostať na miesto rootu. Pri vkladaní a mazaní to nie je také značne, lebo pri týchto funkciách sa v AVL strome vykonáva aj balancovanie čo spôsobuje stratu času.

2. Hash tabuľky

2.1 Fungovanie hash tabuliek

Hash tabuľky je štruktúra, ktorá ukladá dáta do poľa s pomocou hash funkcie. Hash funkcia tvorí z kľúčov indexy pomocou ktorých sa uložené dáta vyhľadávajú. Hashe sa môžu opakovať. Tento jav sa nazýva kolízia. Na predchádzanie kolízii sa používajú rôzne metódy. V hash tabuľke treba brať do úvahy tiež zväčšovanie a zmenšovanie aby sa dáta do tabuľky zmestili a aby zostala efektívna.

2.2 Riešenie kolízií pomocou Separate Chainingu

Separate chaining hash tabuľka je dátová štruktúra, ktorá ukladá hodnoty s rovnakým hashom na to isté miesto a tvorí tam spájaný zoznam. Každá nová hodnota s rovnakým hashom sa presunie na koniec spájaného zoznamu.

2.3 Implementácia Separate chainingu

Insert:

Vkladám vstup na miesto, ktoré mi hash funkcia vygeneruje. Využívam funkciu push_back(), ktorá slúži na vkladanie do posledného miesta v spájanom zozname. Po vložení kontrolujeme na koľko je tabuľka naplnená. Ak je viac ako na 70% tabuľku zväčšíme ak je menej ako 30% tabuľku zmenšíme vo funkcii resize(). Kde sa vytvorí väčší spájaný zoznam do ktorého sa nanovo nahashujú inputy.

```
void insert(string input) {
    hashTable[getHash(input)].push_back(input);
    currentSize++;

    if (table_load() > 0.7) {
        resize(sizeTable * 2);
        // po zvacseni 0.35
    }
    if (table_load() < 0.3) {
        resize(sizeTable / 2);
    }
}
```

Search:

Cyklom prejdeme cez spájaný zoznam na pozícii hashu. Ak sa input nájde program ho vypíše a funkcia sa skončí.

```
void search(string input) {

    int hash = getHash(input);
    for (auto i = hashTable[hash].begin(); i != hashTable[hash].end(); i++)
{
        if (*i == input) {
            //cout << input << " was found" << endl;
            return;
        }
    }
}
```

Delete:

Podobným cyklom ako v search prechádzame spájaný zoznam. Keď sa input nájde zavoláme funkciu erase(), ktorá vymaže pozíciu s hľadaným inputom. Tiež skontrolujeme či je tabuľke potrebné zmeniť veľkosť.

```
void remove(string input) {
    int hash = getHash(input);

    for (auto i = hashTable[hash].begin(); i != hashTable[hash].end(); i++)
{
        if (*i == input) {
            hashTable[hash].erase(i);
            currentSize--;
            return;
        }
    }

    if (table_load() > 0.7) {
        resize(sizeTable * 2);
        // po zvacseni 0.35
    }
    if (table_load() < 0.3) {
        resize(sizeTable / 2);
    }
}
```

2.4 Riešenie kolízií pomocou Linear probingu

Pri tejto technike algoritmus prechádza tabuľku od miesta, kde mal byť input vložený, až kým nenájde voľné miesto kam input uloží.

2.5 Implementácia Linear probingu

Insert:

Najprv zväčšíme veľkosť a skontrolujeme či treba volať resize(). Potom ak pozícia nie je prázdny string tak sa posúvame o jedno políčko. Ak nastane koniec tabuľky tak ideme od začiatku.

```
void insert(string input) {

    currentSize++;

    if (table_load() > 0.7) {
        resize(sizeTable * 2);
        // po zvacseni 0.35
    }
    if (table_load() < 0.3) {
        resize(sizeTable / 2);
    }

    unsigned long long hash = getHash(input);

    while (hashTable[hash] != string()) {
        hash = (hash + 1) % sizeTable;
    }

    hashTable[hash] = input;
}
```

Search:

Môžu nastať dve situácie. Prvá keď nájdeme input na prvej pozícii. Druhá ak ho nenájdeme tak musíme prejsť tabuľku až kým input nenájdeme.

```
void search(string input) {

    unsigned long long hash = getHash(input);

    if (hashTable[hash] == input){
        //cout << input << " was found" << endl;
        return;
    }

    for (; (hash + 1) != hash ; hash = (hash + 1) % sizeTable) {
        if (hashTable[hash] == input){
            //cout << " was found" << endl;
            return;
        }
    }

    //cout << input << " was not found" << endl;
}
```

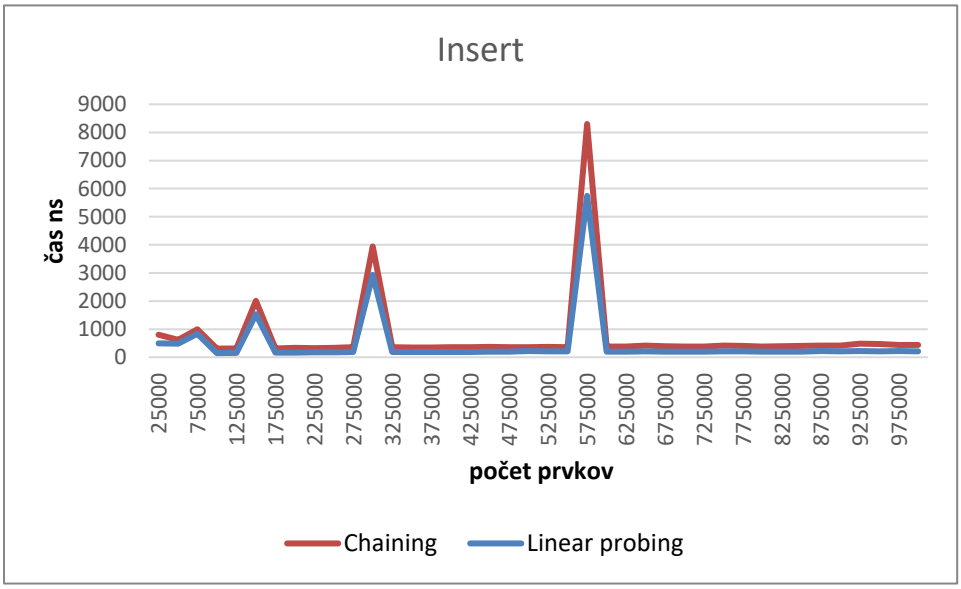
Delete:

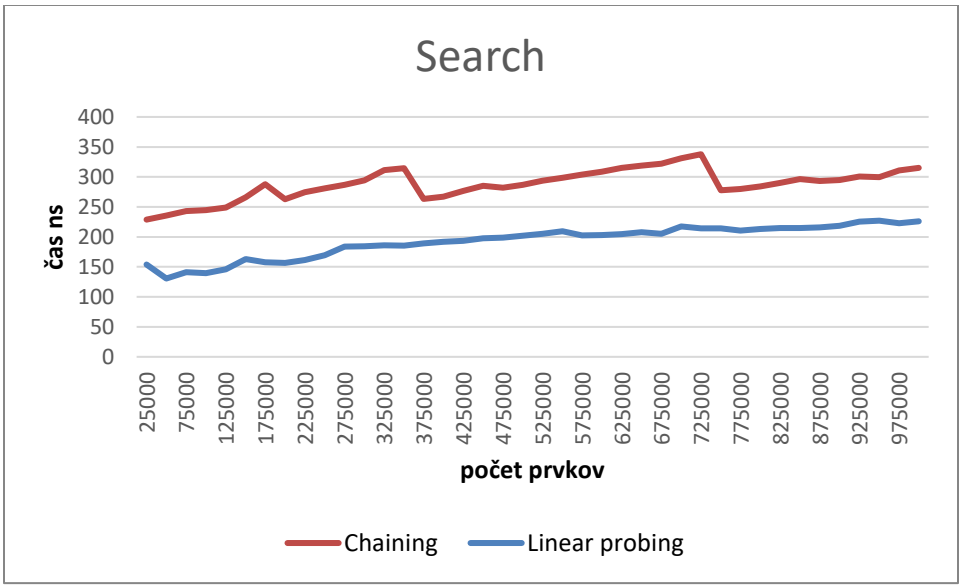
Podobne ako pri search môžu nastať dve situácie. Jediný rozdiel je, že nahrádzame mazané miesto za prázdny string a kontrolujeme či tabuľku nie je nutné zmenšiť.

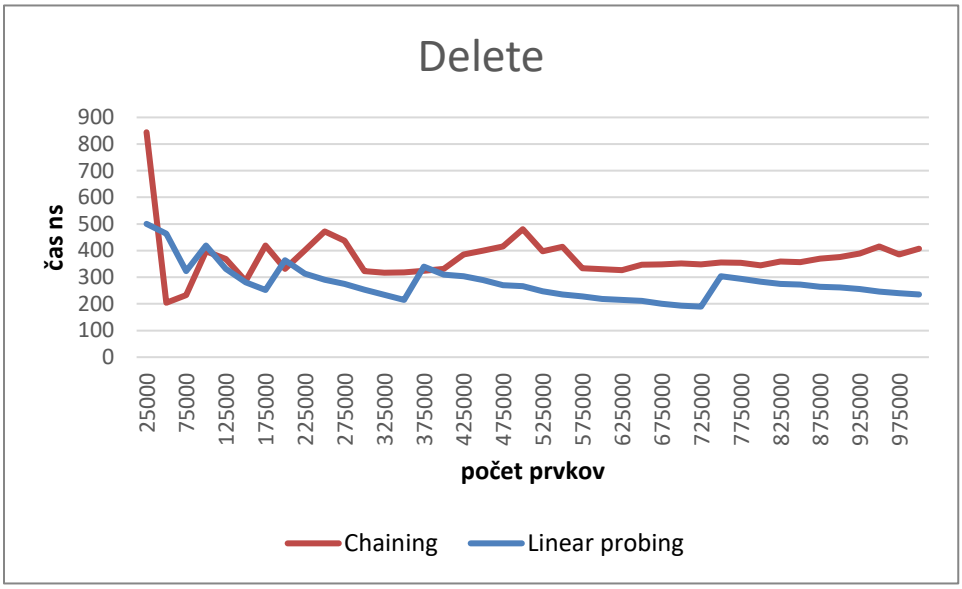
```
void remove(string input) {

    unsigned long long hash = getHash(input);

    if (hashTable[hash] == input){
        hashTable[hash] = string();
        currentSize--;

        if (table_load() < 0.3) {
            resize(sizeTable / 2);
        }
        return;
    }

    for (; (hash + 1) != hash ; hash = (hash + 1) % sizeTable) {
        if (hashTable[hash] == input){
            hashTable[hash] = string();
            currentSize--;

            if (table_load() < 0.3) {
                resize(sizeTable / 2);
            }
            return;
        }
    }
}
```

2.6 Porovnanie výsledkov hash tabuliek

Porovnávať budeme rýchlosť vkladania, hľadania a mazania prvkov v hash tabuľkách. Hodnoty sú merané v nanosekundách. Grafy znázorňujú priemernú rýchlosť jedného prvku vo zvyšujúcom sa počte prvkov.







Z predchádzajúcich grafov si môžeme všimnúť, že tabuľka s Linear Probing implementáciou je rýchlejšia pri všetkých funkciách. Je to spôsobené tým, že prvky sú bližšie pri sebe ako pri Chainingu, kde sú medzi spájanými zoznamy väčšie medzery.

3. Priebeh testovania

Jednotlivé implementácie som testoval volaním metód tried. Každá implementácia má vlastnú triedu. Predtým ako som začal volať metódy som si vytvoril vektory náhodných čísiel a náhodných stringov.

Insert som testoval tak, že som od 25 000 prvkov vkladal po 25 000 prvkov do 1 000 000. Vždy som odmeral rýchlosť 25 000 vložených prvkov a tú som potom delil 25 000 aby som získal priemerný rýchlosť jedného. Toto som opakoval 5-krát a týchto 5 hodnôt som spriemeroval.

Search som testoval vždy po vložení 25 000 prvkov. Hľadal som vždy všetky prvky, ktoré boli vložené. Čiže vložil som 25 000 prehľadal som tých 25 000, vložil som ďalších 25 000 prehľadal som 50 000 a tak ďalej.

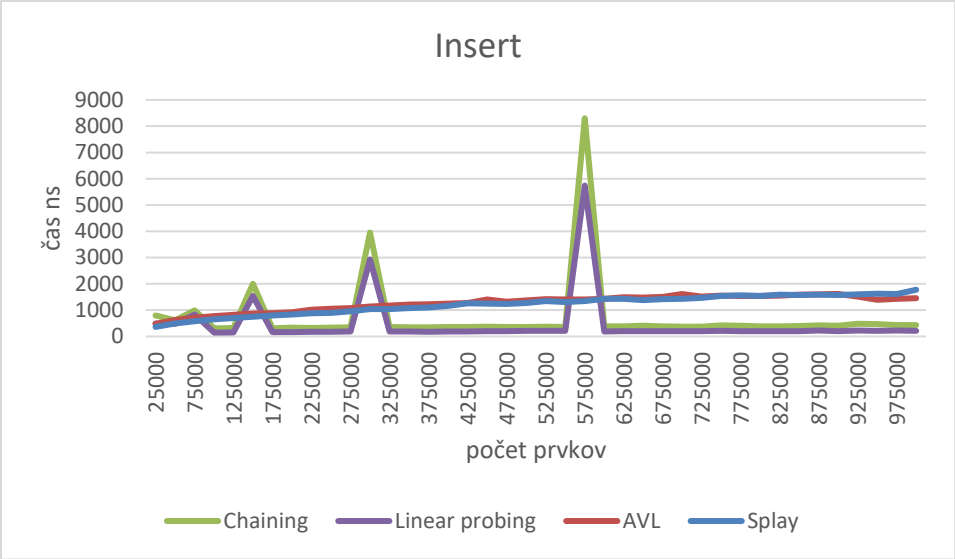
Delete bol testovaný tak, že som vložil prvky a tie boli následne všetky vymazané. Takže, keď som vložil 25 000 zmazal som všetky a následne som vložil 50 000 a zmazal 50 000, potom som vložil 75 000 a zmazal 75 000 a tak ďalej.

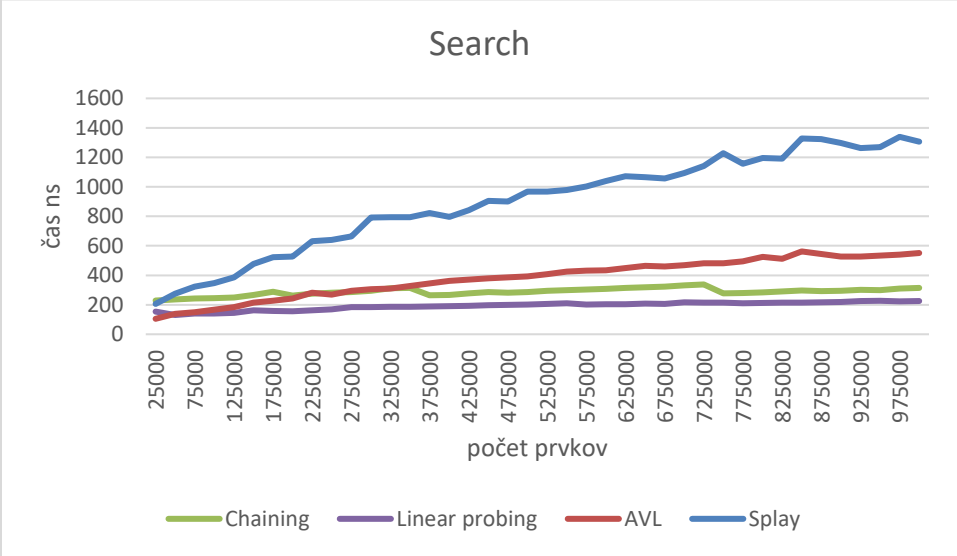
Prvky som bral vždy z predom vygenerovaných vektorov, ktoré mali 1 000 000 prvkov. S prvkami som robil podľa pozície. Čiže som pracoval s prvkami na pozíciách od 0 do 25 000 potom od 0 do 50 000 a tak ďalej. Tento spôsob

bol použitý pri hľadaní a mazaní. Pri vkladaní som postupoval od 0 do 24 999 od 25 000 do 49 999 a tak ďalej.

4. Výsledok

Ako môžeme vidieť na nasledovných grafoch, obe tabuľky sú rýchlejšie v porovnaní so stromami. Deje sa to kvôli tomu, že v tabuľkách nemusíme prehľadávať viac prvkov, keďže má každý prvok vlastný hash kľúč a priemerná zložitosť je O(1) zatiaľ čo pri stromoch to je O(log n). Jediné miesta, kde tabuľky spomalia sú tam, kde sa zväčšujú alebo zmenšujú. Najrýchlejšia štruktúra je Linear probing a najpomalšia Splay tree.





Richard Šléher
ID: 120890