STU v Bratislave

DSA – Zadanie 1

FIIT

Norbert Matuška, xmatuskan@stuba.sk
2021/2022

Norbert Matuška, xmatuskan@stuba.sk

Vlastná implementácia AVL binárneho stromu

Pri implementácii som využil struct, ktorá si zapisuje do pamäte pre každý uzol číslo, pravého, ľavého potomka a výšku.

```
//struct na vytvorenie stromu
typedef struct avltree {
    int num;
    struct avltree* right;
    struct avltree* left;
    int height;
}AVLTREE, * PAVLTREE;
```

Fungovanie binárnych stromov

Binárne stromy fungujú na princípe usporiadania podľa veľkosti (na ľavej strane sú uzly s menším číslom a na druhej strane zase naopak)

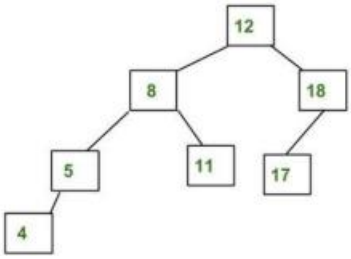
Funkcia pridania uzlu do stromu

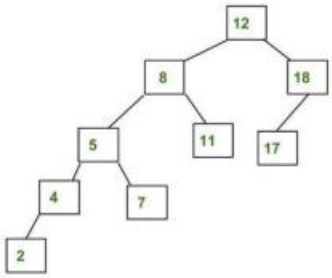
Funkciu pridania do stromu som riešil rekurzívne, aby som sa mohol posúvať po strome na správne miesto. Taktiež popri inserte riešim aj balans stromu a to nasledovne:

```
PAVLTREE insert(PAVLTREE node, int num) {
    //treba nam najst spravnu poziciu na inser a tam to vlozit
    if (node == NULL) {
        return (newTree(num));
    }
    if (num < node->num) {
        node->left = insert(node->left, num);
    }
    else if (num > node->num) {
        node->right = insert(node->right, num);
    }
    else {
        return node;
    }
    //treba "aktualizovat" vyvazovaci faktor pre kazdu node a vyvazovat strom
    node->height = 1 + max(calcHeight(node->left), calcHeight(node->right));

    int balance = findBalance(node);

    if (balance > 1 && num < node->left->num) {
        return rightRotation(node);
    }
    if (balance < -1 && num > node->right->num) {
        return leftRotation(node);
    }
    if (balance > 1 && num > node->left->num) {
        node->left = leftRotation(node->left);
        return rightRotation(node);
    }
    if (balance < -1 && num < node->right->num) {
        node->right = rightRotation(node->right);
        return leftRotation(node);
```

```
    }

    return node;
}
```

Ukážka vyváženého a nevyváženého stromu





Ľavý potomok má výšku 4,

Pravý zase výšku 2, rozdiel nemôže byť > 1 alebo naopak < ako -1

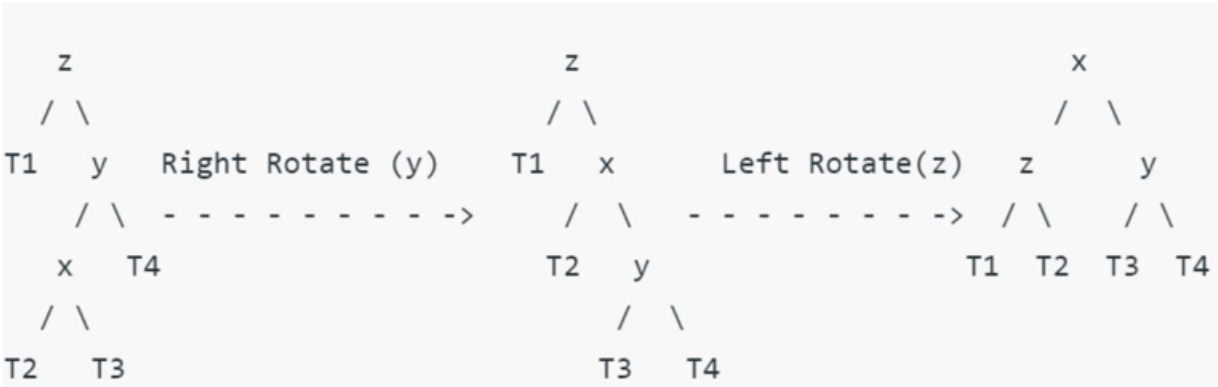
Možnosti vyvažovania

Môžu nastať 4 rôzne stavy a to sú nasledovné:

Left Right Case

V princípe si najprv upravíme ľavého potomka, aby sme mohli urobiť rotáciu left-left

Right Left Case



V tejto situácii si upravíme ľavého potomka, aby sme mohli urobiť rotáciu right right.

Funkcia search

Takmer rovnaký princíp ako pri insert, porovnávame veľkosti, ak sa zhoduje s našou hodnotou, našli sme správny uzol.

```
PAVLTREE searchAVL(PAVLTREE node, int num) {
    if (node == NULL) {
        return NULL;
    }
    else {
        if (num < node->num) {
            return searchAVL(node->left, num);
        } else if (num > node->num) {
            return searchAVL(node->right, num);
        } else {
            return node;
        }
    }
}
```

Funkcia delete

Podobne ako predtým, hľadáme prvok v strome, následne vymažeme a taktiež musíme riešiť balans ako pri insert, keďže narábame s uzlami.

```
//funkcia na vymazanie splayTr
PAVLTREE delete(PAVLTREE root, int num) {
    //musime najst a vymazat splayTr
    if (root == NULL) {
        return root;
    }
    if (num < root->num) {
        root->left = delete(root->left, num);
    }
    else if (num > root->num) {
        root->right = delete(root->right, num);
    }
    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            PAVLTREE temp = root->left ? root->left : root->right; //ternarny
operator

            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
            else {
                *root = *temp;
            }
            free(temp);
        }
        else {
            PAVLTREE temp = minimalValNode(root->right);
```

```
            root->num = temp->num;
            root->right = delete(root->right, temp->num);
        }
    }
    if (root == NULL) {
        return root;
    }

    //Musime vyvazovat strom
    root->height = 1 + max(calcHeight(root->left), calcHeight(root->right));

    int balance = findBalance(root);
    if (balance > 1 && findBalance(root->left) >= 0) {
        return rightRotation(root);
    }
    if (balance > 1 && findBalance(root->left) < 0) {
        root->left = leftRotation(root->left);
        return rightRotation(root);
    }
    if (balance < -1 && findBalance(root->right) <= 0) {
        return leftRotation(root);
    }
    if (balance < -1 && findBalance(root->right) > 0) {
        root->right = rightRotation(root->right);
        return leftRotation(root);
    }
    return root;
}
```

Vlastná implementácia Splay binárneho stromu

Tento algoritmus sa veľmi podobá na AVL, máme tu rotácie a podľa nich balansujeme strom. V mojej implementácii som využil struct, ktorý si zapisuje do pamäte pre každý uzol svoje číslo, pravého a ľavého potomka ale tentokrát bez výšky.

```
typedef struct splayTree {
    int num;
    struct splayTree *right;
    struct splayTree *left;
} SPLAY, *PSPLAY;
```

Fungovanie Splay

Podobne ako AVL alebo RB stromy, Splay je samo balansovaný algoritmus. Hlavná myšlienka tohto algoritmu je priniesť posledne pristúpenú hodnotu do korenu, takže časová komplexita pre naposledy navštívenú hodnotu je O(1) ak k nemu pristúpime znova. Toto je výhodné pre veľmi veľké dátové štruktúry, kde sa využíva iba približne 20% obsahu.

Funkcia search

K tejto funkcii som nepristupoval rekurzívne, pretože som to nevedel rozchodiť, tak som si vybral jednoduchý while loop cez ktorý porovnávam hodnoty v štruktúre s hodnotou ktorú hľadám. Ak sa nájde

zhoda, použijeme funkciu splay() s ktorou prinesieme túto hodnotu ku koreňu, aby bolo jednoduchšie ju v budúcnosti hľadať.

```
PSPLAY searchSplay(PSPLAY root, int num) {

PSPLAY tmp = root;
    if (root == NULL) {
        return root;
    }
    else if(root->num == num) {
        return root;
    }
    else {
        while(1) {
            if (tmp->num > num) {
                if (tmp->left == NULL) {
                    return root;
                }
                else {
                    tmp = tmp->left;
                }
            }
            else if (tmp->num < num) {
                if (tmp->right == NULL) {
                    return root;
                }
                else {
                    tmp = tmp->right;
                }
            }
            else if (tmp->num == num) {

                return splay(tmp, num);

            }
        }
    }
}
```

Funkcia insert

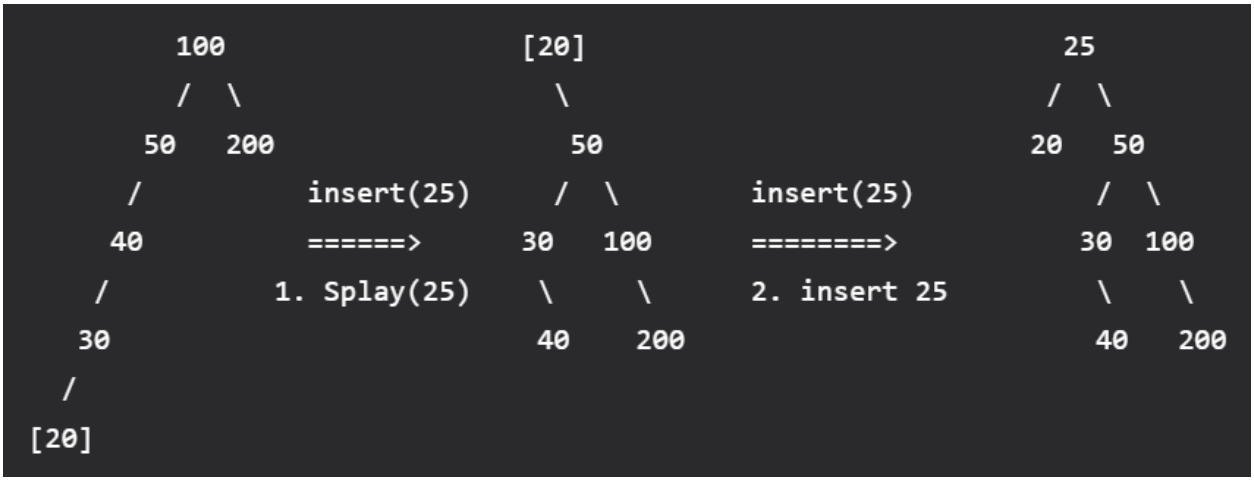
Veľmi jednoduchá funkcia vďaka funkcii splay(), na začiatku si prinesieme najbližšieho potomka ku koreňu, alokujeme pamäť a porovnávame, či máme ist do prava alebo do ľava.

```
//vkladacia funkcia
PSPLAY insertSplay(PSPLAY root, int num) {
    //ak je strom prazdny
    if (root == NULL) {
        return newSplay(num);
    }
    //prinesie najblizsi "list" ku korenu
    root = splay(root, num);
    //ak je uz num v strome, return
```

```
    if (root->num == num) {
        return root;
    }
    else {
        //treba alokovat pamat pre novy num
        PSPLAY newNum = newSplay(num);

        //ak je num korena vacsi, spravime koren ako pravy child newNum
        //a skopirujeme lavy child korena do newNum
        if (root->num > num) {
            newNum->right = root;
            newNum->left = root->left;
            root->left = NULL;
        }
        else {
            newNum->left = root;
            newNum->right = root->right;
            root->right = NULL;
        }

        return newNum; //newNum sa stane novym korenom
    }
}
```

Rôzne možnosti insertovania

1) Koreň je NULL: Alokujeme nový uzol a vrátime ho ako koreň
2) „Splayneme“ danú hodnotu. Ak už je prítomná v strome, jednoducho ju prinesieme ku koreňu
3) Ak je nová hodnota koreňa rovnaká ako hodnota num, nerobíme nič
4) Alokujeme pamäť pre nový uzol a porovnávame hodnoty



Funkcia delete

Vyhľadáme hodnotu, ktorá nám bola daná, prinesieme ju do koreňa, vymažeme a spravíme jedného z potomkov starého koreňa novým koreňom.

```
//funkcia na vymazanie num zo Splay Tree
//vracia novy koren stromu po vymazani num
```

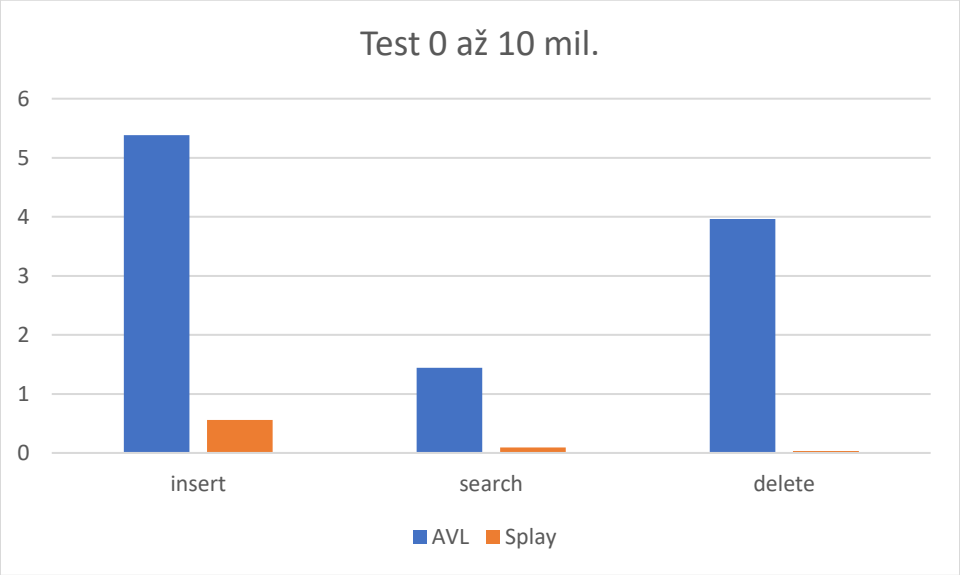
```
PSPLAY deleteSplay(PSPLAY root, int num) {
    PSPLAY temp;

    if(root == NULL) {
        return NULL;
    }
    //"splayneme" num ktore nam bolo dane

    root = searchSplay(root, num);

    //ak sa tam nenachadza spravne num, return root
    if (num != root->num) {
        return root;
    }

    if(root->left == NULL) {
        temp = root;
        root = root->right;
    }

    else {

        temp = root;
        //kedze num == root->num, tak po splay(root->left, num) strom kt.
dostaneme
        //nebude mat pravych child/potomkov a max splayTr v lavom podstrome
bude "splaynuta"
        //novy root
        root = splay(root->left, num);

        //spravime praveho potomka stareho korena ako praveho potomka noveho
korena
        root->right = temp->right;
    }

    free(temp);

    //novy koren
    return root;

}
```

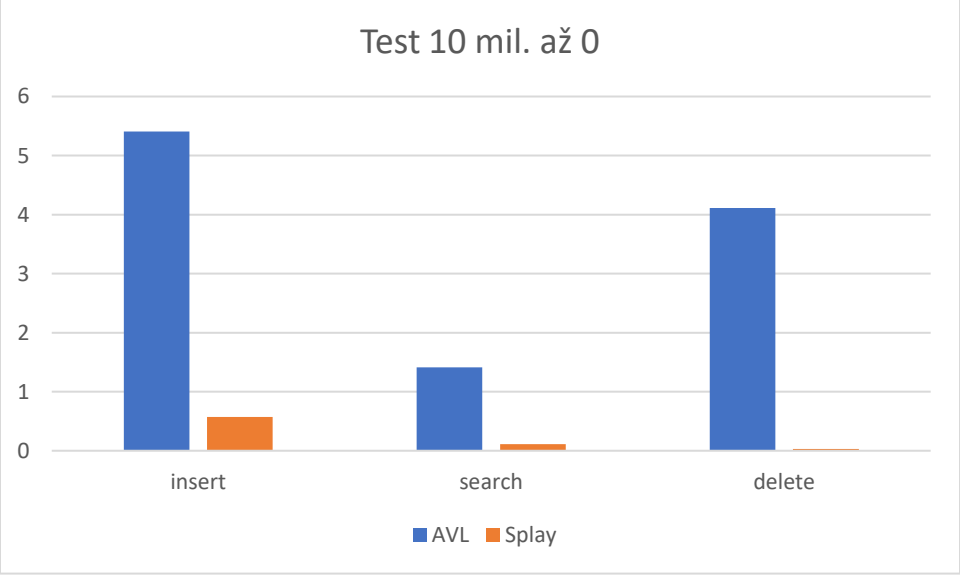
Testovanie

Testoval som pri 10 mil. hodnotách s insert, search a delete.

```
Test 0 az 10 milionov
cas insertu do AVL je:    5.382000
cas prehladavania AVL je:    1.444000
cas deletu AVL je:    3.961000

cas insertu do Splay je:    0.557000
cas prehladavania Splay je:    0.094000
cas deletu Splay je:    0.031000
```

```
Test 10 milionov az 0
cas insertu do AVL je:    5.405000
cas prehladavania AVL je:    1.414000
cas deletu AVL je:    4.112000

cas insertu do Splay je:    0.571000
cas prehladavania Splay je:    0.110000
cas deletu Splay je:    0.031000
```





Vlastná implementácia Chain Hash Table

Hashovacia tabulka je dátová štruktúra, ktorá obsahuje kľúče s hodnotami. Ja som implementoval najprv tabuľku s riešením kolízií pomocou zreťazenia. Znamená to, že každé miesto v poli odkazuje na zreťazený zoznam vložených záznamov.

V mojej implementácii mám dva structy. Jeden reťazový, kde sa ukladajú hodnoty a ukazovateľ na ďalší prvok a v druhom sa zase ukladá tabulka, veľkost a počet elementov. Vyzerá nasledovne:

```
typedef struct chain {
    int num;
    struct chain *next;
} CHAIN, *PCHAIN;

typedef struct tableChain {
    PCHAIN *table;
    int size;
    int elementsNum;
} TABLECHAIN, *PTABLECHAIN;
```

Funkcia insert

Potrebujeme nájsť správne miesto v tabuľke a pridať ho na jeden koniec.

```
void insertChain(PTABLECHAIN info, int num) {
    int funcHash = num % info->size;
    PCHAIN *chainTable = info->table;
    info->elementsNum++;
    PCHAIN temp;
    PCHAIN last = chainTable[funcHash];

    if (chainTable[funcHash] == NULL) {
        chainTable[funcHash] = malloc(sizeof(CHAIN));
        chainTable[funcHash]->num = num;
        chainTable[funcHash]->next = NULL;
    }
    else {
        temp = malloc(sizeof(CHAIN));
        temp->num = num;
        temp->next = NULL;
        while (last->next != NULL) {
            if(last->num == num) {
                return;
            }
            last = last->next;
        }
        if(last->num == num) {
            return;
        }
        last->next = temp;
    }
    info->table = chainTable;
}
```

Funkcia search

Prechádzame tabuľkou tak, že sa posúvame while loopom pomocou pointerov shift->next na ďalšie miesto dokým nenájdeme zhodu.

```
//searchChain pre hash table
PCHAIN searchChain(PTABLECHAIN info, int num) {
    PCHAIN *temp = info->table;
    int funcHash = num % info->size;
    PCHAIN shift = temp[funcHash];

    while(1) {
        if (shift->num == num) {
            return shift;
        }
        else {
            if(shift->next == NULL) {
                break;
            }
            shift = shift->next;
        }
    }
}
```

Funkcia delete

Veľmi jednoduchá funkcia, pomocou funkcie search prehľadávame tabuľku do kým sa nenájde zhoda, to následné nastavíme na nulovú hodnotu a tým pádom vymažeme hodnotu z tabuľky.

```
void deleteChain(PTABLECHAIN info, int num) {
    int funcHash = num % info->size;
    PCHAIN *temp = info->table;

    if (temp[funcHash] == NULL) {
        return;
    }
    else {
        PCHAIN find = searchChain(info, num);
        if (find->num == num) {
            find->num = 0;
            //info->size--;

        }
    }
}
```

Vlastná implementácia Linear Hash Table

Pri tomto algoritme pridávame do tabulky na základe hash() funkcie hodnoty a kolízie riešime nie zreťazením ale zisťovaním, či v pravo alebo v ľavo je voľno, čiže hľadáme najbližšie voľné miesto až pokiaľ nedôjdeme na koniec tabuľky, v tom prípade začneme znova od začiatku.

Funkcia insert

Jednoducho sa posúvame po tabuľke kým si nenájdeme miesto, následne insertneme.

```
void insertLinear(int key, int num) {
    item = (PLINEAR) malloc(sizeof(LINEAR));
    item->key = key;
    item->num = num;

    //zobereme nasu hash
    int linearIndex = linearCode(key);

    //posuvame sa po poli kym nenajdeme prazdnu alebo vymazanu cast
    while(linArr[linearIndex] != NULL) {
        if(linArr[linearIndex]->key == key){
            return;
        }
        //posuvame sa dalej
        linearIndex++;

        linearIndex %= sizeLin;
    }
    linArr[linearIndex] = item;
    numOfLin++;

    if(linearLoad(numOfLin, sizeLin)) {
        rehash();
    }
}
```

Funkcia search

Posúvame sa po tabuľke kým nenastane zhoda.

```
PLINEAR searchLinear(int key) {

    int linearIndex = linearCode(key);

    while (linArr[linearIndex] != NULL ) {
        if (linArr[linearIndex]->key == key) {
            return linArr[linearIndex];
        }
        //posuvame sa dalej
        linearIndex++;

        linearIndex %= sizeLin;
    }
    return NULL;
}
```

Funkcia delete

Bohužial, funkciu delete som nestihol implementovať.

Testovanie

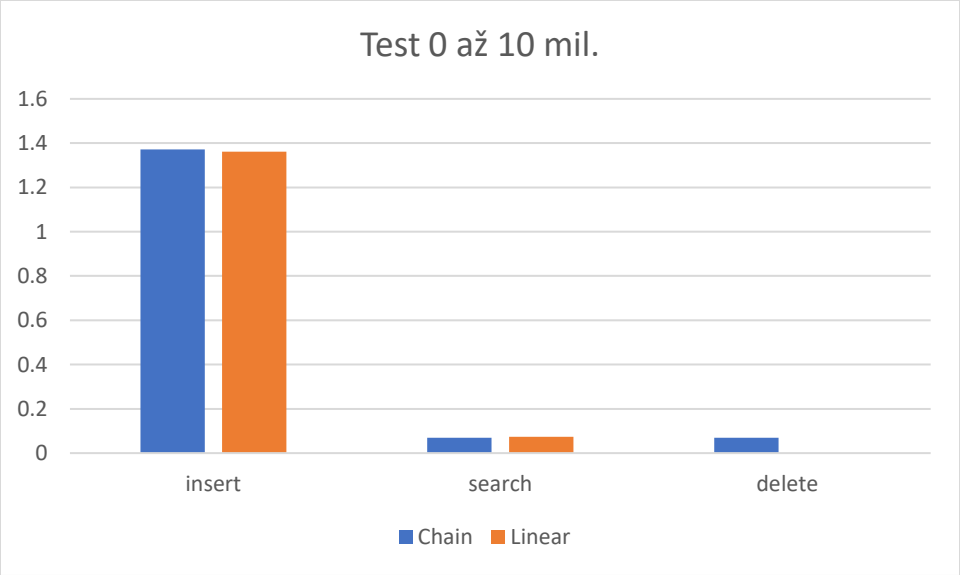
Testoval som s použitím rovnakých techník ako pri BVS.

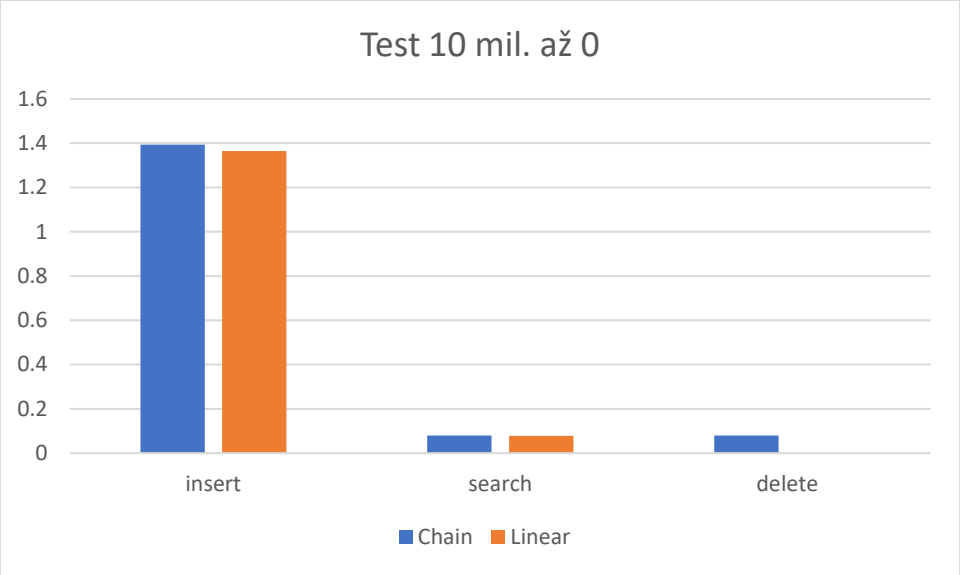
Test 0 až 10 miliónov

```
cas insertu do Chain Hash je:   1.371000
cas prehladavania Chain Hash je:   0.069000
cas deletu Chain Hash je:   0.069000

cas insertu do Linear Hash je:   1.362000
cas prehladania Linear Hash je:   0.073000
```

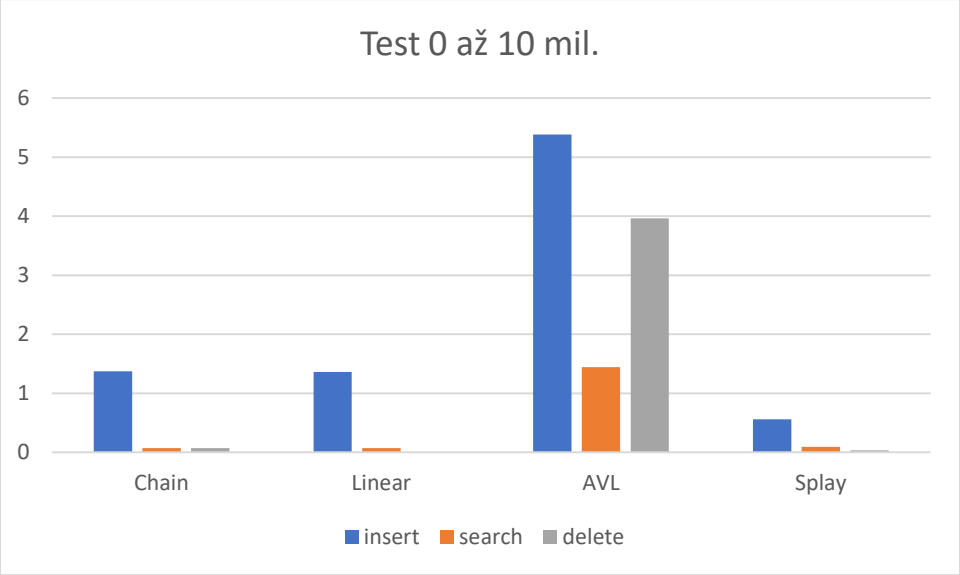
Test 10 miliónov až 0

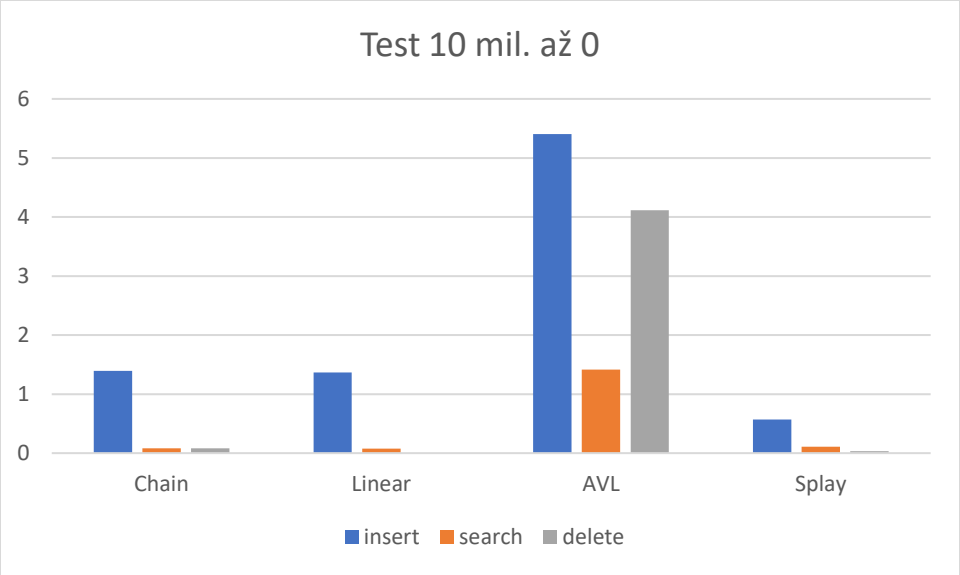
```
cas insertu do Chain Hash je:   1.394000
cas prehladavania Chain Hash je:   0.079000
cas deletu Chain Hash je:   0.079000

cas insertu do Linear Hash je:   1.364000
cas prehladania Linear Hash je:   0.078000
```





Porovnanie všetkých





Ako môžete vidieť, AVL strom má najhoršie časy zo všetkých implementácii, pravdepodobne kvôli tomu, že som ho nie úplne efektívne implementoval a taktiež aj balans celého stromu je procesovo náročná akcia.

Zdroje

https://www.sanfoundry.com/c-program-implement-hash-tables-chaining-with-singly-linked-lists/

https://www.programiz.com/dsa/hash-table

https://www.geeksforgeeks.org/avl-tree-set-1-insertion/?ref=lbp