

FIIT STU

Dátové štruktúry a algoritmy

Zadanie 2 – Vyhľadávanie v dynamických množinách

Adam Miškove

ID: 103056

2020/2021

AVL strom

Klasické binárne stromy fungujú na princípe zoradenia prvkov podľa daných hodnôt, pričom nižšie hodnoty patria vľavo a vyššie vpravo.

Týmto spôsobom je možné spraviť veľmi dlhé a vysoké stromy, pričom ich výška/hĺbka drasticky ovplyvňuje rýchlosť vyhľadávania prvkov. Ak ich vybalancujeme algoritmom pre AVL stromy, vieme pozitívne ovplyvniť rýchlosť narábania so stromom. Týmto dosiahneme maximálnu výšku $O(\log n)$.

Náš strom bude vytvorený zo štruktúry obsahujúcej dané dáta, podľa ktorých sa bude zoradovať a pointerov na svoju ľavú a pravú podvetvu. Pri vyvažovaní stromu pomocou AVL budeme potrebovať zisťovať hĺbku jednotlivých prvkov v strome, na čo bude slúžiť ďalšia súčasť štruktúry.

```
typedef struct person{
    char name[MAXNAME];
    int age;
}PERSON;

|

typedef struct node{
    PERSON *pers;
    int depth;
    struct node *left;
    struct node *right;
}NODE;
```

Základ stromu vytvárame rovnako ako pri normálnom strome bez jeho vyváženia, alokujeme každému prvku požadované miesto a inicializujeme ho.

```
NODE *createNode(char *inputName, int inputAge){ //funkcia vytvorí noveho cloveka
    NODE *newNode = (NODE*)malloc(sizeof(NODE)); //alokujem miesto
    PERSON *newPerson = (PERSON*)malloc(sizeof(PERSON)); //alokujem miesto
    newNode->pers = newPerson;
    strcpy(newNode->pers->name, inputName);
    newNode->pers->age = inputAge;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->depth = 1;
    return (newNode);
}
```

Rozdiel v tomto prípade je **int depth**, ktorý inicializujeme ako 1 (pri inicializovaní má vždy prvok hĺbku 1).

Pomocná funkcia **max()** vráti maximum zo zadaných hodnôt.

Pomocná funkcia **determinePlace()** berie ako argumenty prvok, do ktorého chceme vkladať, a prvok, ktorý chceme vkladať. Vráti 0, ak vkladany prvok patrí na ľavú stranu (je menší), vráti 1 ak je to naopak, a vráti 2 ak je rovnaký.

Pomocná funkcia **printInOrder()** vypíše prvky v našom strome v správnom poradí.

Prejde ho celý rekurziou, pričom prvé sa vypíšu prvky čo najviac vľavo.

Pomocná funkcia **randomString()** vygeneruje náhodný string z malých písmen.

Pomocná funkcia **createRandomPerson()** vytvorí prvok s náhodným človekom (náhodné meno a vek).

Pomocná funkcia **findDepth()** vracia hĺbku prvku, pričom vracia nulu, ak je prvok, na ktorý siahame, NULL. Toto predchádza segmentation faultom.

Pomocná funkcia **findBalance()** vracia hodnotu rovnováhy prvku, toto je potrebné pri balancovaní AVL stromu.

Pomocná funkcia **updateDepth()** vracia novú hodnotu hĺbky prvku. Funkciu používam pri vkladaní prvku do stromu, pričom je šanca, že sa bude hodnota hĺbky meniť.

V jednej z hlavných funkcií programu, **avlInsertNode()**, používame podstatnú funkciu **rotateLeft()** a jej opak **rotateRight()**.

RotateRight() má za úlohu vykonať rotáciu prvkov stromu vľavo.

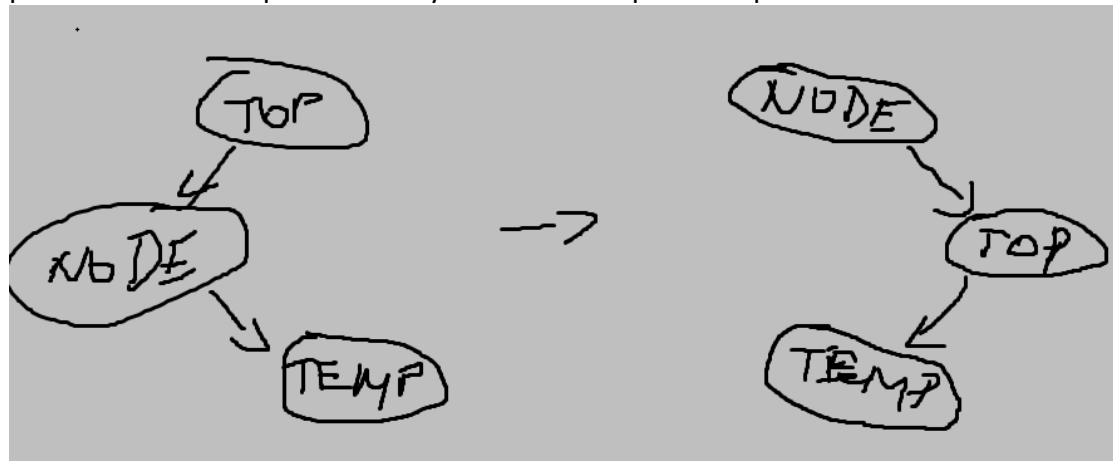
```
NODE *rotateRight(NODE *top){
    NODE *node = top->left;
    NODE *temp = node->right;

    node->right = top;
    top->left = temp;

    top->depth = updateDepth(top);
    node->depth = updateDepth(node);

    return node;
}
```

Rotácia sa vykoná okolo prvku "top", po rotácii vpravo bude prvok "top" pravou vetvou prvku "node", ktorý bude nad ním. Prvok "temp", ktorý bol na pravej vetve prvku "node" bude premiestnený na ľavú vetvu prvku "top".



Hĺbka prvku v AVL strome znamená maximum z hĺbky jeho oboch vetiev.

```
int updateDepth(NODE *tree){
    int newDepth;
    newDepth = max(findDepth(tree->left), findDepth(tree->right)) + 1;
    return newDepth;
}
```

Táto hĺbka je potrebná pre dôležitú súčasť vyvažovania stromu, a to práve zisťovanie nevyrovnanosti stromu. Vyrovnanosť stromu znamená, že každý prvok má "balance" v intervale $<-1;1>$.

```
int findBalance(NODE *tree){  
    if(tree==NULL){  
        return 0;  
    }  
    return findDepth(tree->left)-findDepth(tree->right);  
}
```

Balance prvku vypočítame = hĺbka ľavej vetvy - hĺbka pravej vetvy.

```
NODE *avlInsertNode(NODE *tree, NODE *myNode){  
    int balance;  
    if(tree==NULL){  
        //klasicky insert do stromu  
        return myNode;  
    }  
    if (determinePlace(tree, myNode)==1){  
        tree->right = avlInsertNode(tree->right, myNode);  
    }  
    else if (determinePlace(tree, myNode)==0){  
        tree->left = avlInsertNode(tree->left, myNode);  
    }  
    else return tree;  
    tree->depth = updateDepth(tree);  
    balance = findBalance(tree);  
    //zistim ci je strom vyvazeny  
    printf("balance tree name: %s    balance: %d    \n", tree->pers->name, balance);  
  
    if(balance > 1 && (strcmp(myNode->pers->name, tree->left->pers->name)<0)){  
        //left left rotacia  
        printf("LL rotate\n");  
        return rotateRight(tree);  
    }  
    if(balance > 1 && (strcmp(myNode->pers->name, tree->left->pers->name)>0)){  
        //left right rotacia  
        printf("LR rotate!!!!!!!!!!!!\n");  
        tree->left = rotateLeft(tree->left);  
        return rotateRight(tree);  
    }  
    if(balance < -1 && (strcmp(myNode->pers->name, tree->right->pers->name)>0)){  
        //right right rotacia  
        printf("RR rotate\n");  
        return rotateLeft(tree);  
    }  
    if(balance < -1 && (strcmp(myNode->pers->name, tree->right->pers->name)<0)){  
        //right left rotacia  
        printf("RL rotate!!!!!!!!!!!!\n");  
        tree->right = rotateRight(tree->right);  
        return rotateLeft(tree);  
    }  
    return tree;  
}
```

Vloženie prvku do AVL stromu:

1) Prvý krok je podobný/rovnaký ako vloženie prvku do normálneho binárneho stromu, rekurzívne sa prejde strom, pričom sa prechádza smerom vpravo, ak je prvok väčší (funkcia **determinePlace()** vráti 1), a prechádza sa smerom vľavo ak je prvok menší, pričom funkcia **determinePlace()** vráti 0. Do nášho stromu duplikáty nedávame.

2) Po prvom kroku (klasickom vložení) musíme aktualizovať hodnoty hĺbky našich ovplyvnených prvkov, aby sme vedeli nájsť správnu hodnotu vyrovnanosti cez **findBalance()**, ktorá bude diktovať, čo sa bude odohrávať v treťom kroku.

3) Tretí krok sa dá pochopiť ako riešenie novej nevyrovnanosti stromu (balance nepatrí do $<-1;1>$). Ak je balance > 1 , znamená to, že od daného prvku je strom nevyrovnaný, s tým, že je na ľavej vetve hlbší. Ak je balance < -1 , znamená to, že je pravá vetva hlbšia.

Existujú štyri typy nevyrovnanosti stromu, po pridaní prvku do už vyrovnaného stromu.

Zadanie 2 – Vyhľadávanie v dynamických množinách

A) LeftLeft rotácia nastáva, ak je prvok nevyrovnaný a jeho ľavá vetva je hlbšia. Zároveň musí byť vkladací prvok vložený na ľavú vetvu. Vykonáva sa **rotateRight()**.

B) RightRight rotácia nastáva v opačnom prípade, teda je pravá vetva hlbšia a prvok je vkladací napravo. Vykonáva sa **rotateLeft()**.

C) LeftRight rotácia nastáva, ak je prvok nevyrovnaný a jeho ľavá vetva je hlbšia, a zároveň je prvok vložený na pravú vetvu. Vykonáva sa **rotateLeft()** na ľavej podvetve prvk. Potom sa vykonáva **rotateRight()**.

D) RightLeft rotácia nastáva, ak je prvok nevyrovnaný a jeho pravá vetva je hlbšia, a zároveň je prvok vložený na ľavú vetvu. Vykonáva sa **rotateRight()** na pravej podvetve prvk. Potom sa vykonáva **rotateLeft()**.

Vyhľadávanie prvku v AVL strome:

Vo vyrovnanom AVL strome je vyhľadávanie omnoho efektívnejšie ako v normálnom nevyrovnanom binárnom strome.

```
NODE* avlSearch(NODE *tree, char *name, int age){
    if(tree==NULL){
        return NULL;
    }
    else if(strcmp(name,tree->pers->name)<0){
        avlSearch(tree->left,name,age);
    }
    else if(strcmp(name,tree->pers->name)>0){
        avlSearch(tree->right,name,age);
    }
    else if(age<tree->pers->age){
        avlSearch(tree->left,name,age);
    }
    else if(age>tree->pers->age){
        avlSearch(tree->right,name,age);
    }
    else if(age == tree->pers->age && strcmp(name,tree->pers->name) == 0){
        //printf("\nnasiel sa %s, %d\n",tree->pers->name, tree->pers->age);
        return tree;
    }
    //printf("\nnebolo najdene\n");
    return NULL;
}
```

Funguje to rovnako ako pri normálnom vyhľadávaní v binárnom strome, strom sa prechádza rekurzívne podľa pozície vyhľadávaného prvku voči aktuálne prechádzanému prvk. Podľa toho, či je prvok väčší alebo menší, vieme, do ktorej podvetvy sa ďalej vydať. Ak sa prvky zhodujú, náš prvok sme našli a algoritmus končí.

Testovanie pre túto časť zadania som rozdelil na **testInsertStrom()** a **testSearchStrom()**. Tieto dve funkcie fungujú na rovnakom princípe, vykonajú daný počet iterácií pre náš strom, pričom sú spojené do jednej funkcie, **fullTestStrom()**, ktorá funguje ako ich sprostredkovateľ.

Všetko testovaniearába s prvkami, kde je relevantné info 5-miestny náhodne vygenerovaný string malých písmen, a náhodné číslo v intervale <0;49>. Tieto dáta reprezentujú meno a vek človeka.

Prevzatý vyrovnaný strom

Základ kódu prevzatý z <https://www.codesdope.com/course/data-structures-red-black-trees-insertion/>

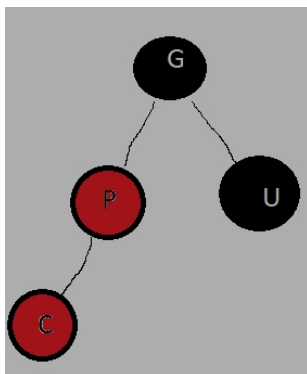
Metóda riešenia: **Red Black Tree**

Red Black stromy majú hĺbku/výšku najviac $O(\log n)$.

Každý prvok tohto stromu je označený ako červený alebo čierny. Koreň stromu a jeho listy sú vždy čierne. Listy označujeme takisto ako NIL prvky. Prvky, ktoré su podvetvami červeného prvku musia byť čierne. Toto znamená, že neexistujú dva spojené červené prvky. Všetky cesty z prvku ku svojim NIL prvkom obsahujú rovnaký počet čiernych prvkov. Čierne prvky majú svoju špeciálnu čiernu výšku, ktorá je určená počtom čiernych prvkov na ceste daného prvku k listom.

Pri **vkľadaní** prvku do red black stromu začneme podobne ako pri normálnych binárnych stromoch, a teda jednoduchým vložením prvku. Tento bude zafarbený na červeno, čo ale nevyhovuje existujúcim pravidlám. Môže porušiť pravidlo, ktoré hovorí o tom, že dva spojené prvky nemôžu byť červené. Takisto môže porušiť pravidlo, že koreň stromu je vždy čierny, čo sa ale dá jednoducho opraviť.

Funkcia **insertion_fixup()** opravuje porušenie pravidiel, ktoré naše vloženie mohlo spôsobiť.



Pre zjednodušenie:

G:Grandparent, P:Parent, C:Child, U:Uncle

Sú tri základné prípady, ktoré nastanú:

- 1) C je nami vložený prvok(červený). G je čierny, P je červený a U je červený. Posunieme farby smerom nahor, čo môže ale porušiť pravidlá vyššie v strome. V tom prípade jednoducho znova posunieme farby, až dokým sa posunú bez problémov.
- 2) C je nami vložený prvok(červený), G je čierny, P je červený a U je čierny. V tomto prípade je nami vložený prvok C vložený vpravo od P. Toto zmeníme na prípad 3 vykonaním rotácie vľavo funkcie **left_rotate()** okolo prvku P.
- 3) Tretí a posledný prípad je identický druhému, lenže nami vložený prvok C je vľavo od P. V tomto prípade je postup taký, že prefarbíme P na čiernu a G na červenú, a následne vykonáme pravú rotáciu **right_rotate()** okolo prvku G. Toto napraviť pravidlá v strome a vráti ho do rovnováhy.

Tieto tri prípady nastávajú aj pri opačnej pozícii prvku P voči prvku G, teda keď sú prvky P a U vymenené. Princíp ich riešenia je symetrický, pričom vymeníme ľavú a pravú stranu.

Adam Miškove
ID: 103056

Vyhľadávanie v tomto binárnom strome funguje rovnako ako pri klasických binárnych stromoch a obnáša rovnaké výhody ako pri AVL stromoch, čo je opísané vyššie.

Testovanie som sa snažil pre lepšie porovnanie spraviť rovnako ako pri AVL strome.

Hash

Hashovanie je metóda ukladania dát, ktorá takisto kladie dôkaz na efektivitu vyhľadávania konkrétnych prvkov.

Toto dosahuje použitím nejakej hash funkcie, ktorá vygeneruje z kľúča(našich vkladanych dát) číslo. Toto číslo viacmenej diktuje, kde sa v poli alebo nejakej dátovej štruktúre bude náš prvok nachádzať.

Pri tomto prípade som použil rovnakú štruktúru ako v prvom(AVL strom), aby som zachoval férovosť pri porovnávaní.

V tomto riešení kolízií pri hashovaní som si vybral spájané zoznamy.

Prvky do spájaného zoznamu vytvárajú pomocné funkcie **createNewNode()**, **createPerson()**, **randomString()** a **randomNumber()**.

createNewNode() inicializuje nový prvok v kontexte spájaného zoznamu.

createPerson() inicializuje a vytvorí nového človeka, pri čom jej pomôžu funkcie **randomString()** a **randomNumber()**.

```
NODE *createNewNode(PERSON *person){
    NODE *newNode = (NODE*) malloc(sizeof(NODE));
    newNode->pers = person;
    newNode->next = NULL;
    return newNode;
}

char *randomString(int size){           //vygeneruje random string malych pismen
    char pismena[] = "abcdefghijklmnopqrstuvwxyz";
    char *str = NULL;
    str = (char*)malloc(sizeof(char)*(size+1));
    for(int i=0; i<size; i++){
        str[i]=pismena[rand()%(26)];
        if(i==size-1) str[i+1]='\0';
    }
    return str;
}

int randomNumber(int max){             //vygeneruje random cislo s maximom
    int num;
    num = rand() % max;
    return num;
}

PERSON *createPerson(char *inputName, int inputAge){ //vytvori/inicializuje struct person
    PERSON *newPerson = (PERSON*) malloc(sizeof(PERSON));
    strcpy(newPerson->name, inputName);
    newPerson->age = inputAge;
    return newPerson;
}
```

Funkcia **appendToList()** pre prvok nájde korektné miesto v tabuľke spájaných zoznamov(ktoré závisí od hashu) a pripojí ho na koniec daného zoznamu.

```
void appendToList(NODE *newNode, NODE **table, unsigned long int hash){
    NODE *pomocnik = table[hash];
    if(pomocnik == NULL){
        table[hash] = newNode;
    }
    else{
        while(pomocnik->next!=NULL){
            pomocnik = pomocnik->next;
        }
        if(pomocnik->next == NULL){
            pomocnik->next = newNode;
            overlap++;
        }
    }
}
```

Spájaný zoznam jednoducho prechádza až dokým nenarazí na jeho koniec, na ktorý vkladany prvok patrí.

Pomocná funkcia **findInList()** hľadá daný prvok v zozname. Zoznam prechádza až do jeho konca, pričom sa zastaví, ak hľadaný prvok nájde.

```
NODE* findInList(PERSON *subject, NODE* list){
    if(list==NULL){
        return NULL;
    }
    else{
        NODE *pomocnik = list;
        while(pomocnik!=NULL){
            if(strcmp(subject->name,pomocnik->pers->name)==0){
                if(subject->age == pomocnik->pers->age){
                    return pomocnik;
                }
            }
            pomocnik = pomocnik->next;
        }
    }
    return NULL;
}
```

Adam Miškove
ID: 103056

Pomocná funkcia **search()** funguje ako sprostredkovateľ medzi testovacou funkciou a funkciou **findInList()**.

Funkcia **createHash()** je naša hashovacia funkcia. Vracia hash(číslo) pre náš konkrétny prvok. Táto hashovacia funkcia je inšpirovaná hashovacou funkciou **djb2** od Dana Bernsteina.

```
unsigned long int createHash(PERSON *pers, long int tableSize){    //inspi
    unsigned long int hash = 5381;
    int i = 0;
    while(pers->name[i] != '\0'){
        hash = (hash * (unsigned long)pow(2,5)) + hash + pers->name[i];
        i++;
    }
    hash += pers->age;
    return (hash%tableSize);
}
```

Začíname s hodnotou hashu **5381**, čo je prvočíslo. Pre každý charakter v mene subjektu sa k hashu **pripočíta $hash * 2^5 + \text{ASCII hodnota daného charakteru}$** . Na konci jednoducho **pripočíta k hashu vek subjektu**.

Operáciu zväčšenia tabuľky mám implementovanú vo funkcii **rehash()**. Túto funkciu volám iba ak počet prvkov v tabuľke presiahne veľkosť tabuľky. Nová, zväčšená tabuľka je 5x väčšia ako pôvodná.

```
1 NODE **rehash(NODE **table, long int tableSize){
2     NODE *pomoc = table[0], *rakety;
3     long int newSize = tableSize * 5;
4     NODE **newTable = (NODE**)malloc(sizeof(NODE*)*(newSize));
5     for(long int i=0;i<newSize;i++){
6         newTable[i] = NULL;
7     }
8
9     for(long int i=0; i<tableSize;i++){
10        pomoc = table[i];
11        while(pomoc!=NULL){
12            unsigned long int hash = createHash(pomoc->pers,newSize);
13            rakety = createNewNode(createPerson(pomoc->pers->name,pomoc->pers->age));
14            appendToList(rakety,newTable,hash);
15            pomoc = pomoc->next;
16        }
17    }
18    for(long int i = 0;i<tableSize;i++){
19        free(table[i]);
20    }
21    free(table);
22
23    return newTable;
24 }
```



1) Najprv alokujem miesto pre novú, 5x väčšiu tabuľku a inicializujem ju s hodnotami NULL.

2) V druhom kroku prejdem celú starú(pôvodnú) tabuľku, pričom každý spájaný zoznam prejdem a každý vložený prvok zoznamov preložím s novým hashom do novej tabuľky.

Pre **testovanie** som vytvoril tri pomocné funkcie, **testInsertHash()**, **testSearchHash()** a **initializeTestHash()**. Insert a search iterujú toľko krát, koľko im nakážeme.

initializeTestHash() vytvorí, alokuje miesto pre tabuľku a inicializuje ju. Túto tabuľku vráti a podá funkcii **testInsertHash()**, ktorá vykoná nami zadané iterácie pre vkladanie náhodných prvkov do hashovej tabuľky. Táto funkcia volá aj funkciu **rehash()**. Konečnú verziu zväčšenej tabuľky vracia a podá funkcii **testSearchHash()**, ktorá prevedie požadované množstvo iterácií pre vyhľadávanie náhodných prvkov v tabuľke. Všetko testovanie narába s prvkami, kde je relevantné info 5-miestny náhodne vygenerovaný string malých písmen, a náhodné číslo v intervale <0;49>. Tieto dáta reprezentujú meno a vek človeka.

Prevzatý hash

Pre prevzatý hash som si vybral metódu riešenia **linear probing**.

Dôležitý rozdiel medzi mojim a prevzatým je, že ja som používal na riešenie kolízií spájané zoznamy, pričom tento prevzatý hash používa linear probing.

Linear probing v prípade kolízie vkladaneho prvku v hash tabuľke sa jednoducho pokúsi nájsť najbližšie voľné nasledujúce miesto v tabuľke. Toto je veľmi výhodné a jednoduché v prípadoch, kde je tabuľka niekoľkokrát väčšia ako počet vkladanych prvkov. Pri vyhľadávaní funguje na rovnakom princípe, podľa vygenerovaného hashu nájde index v tabuľke, a ak sa prvok nezhoduje, posunie sa ďalej, až dokým prvok nenájde. Toto môže byť problematické, ak hľadáme prvok, ktorý sa v tabuľke nenachádza.

```
long int searchElementHashCudzi(char *inputName, int inputAge)
{
    if(countHash==0){
        printf("Error.\nTable is EMPTY\n");
        exit(EXIT_FAILURE);
    }
    long int probe=createHashHashCudzi(createPerson(inputName,inputAge), SIZE);
    while(arr[probe]!=NULL)
    {
        if(probe == SIZE - 1) return 0;
        if(arr[probe]->pers->age==inputAge && strcmp(arr[probe]->pers->name,inputName)==0){
            //printf("person found! name: %s, age: %d\n",inputName,inputAge);
            return probe;
        }
        probe=(probe+1)%SIZE;
    }
    return -1;
}
```

Vyhľadávanie prvku sa správa tak, že si funkcia **searchElement()** vygeneruje hash pre hľadaný prvok pomocou funkcie **createHash()**. Pomocou daného hashu potom skúša, či sa na danom indexe v tabuľke nachádza hľadaný prvok. Ak nie, tak sa postupne posúva na ďalšie indexy, s tým, že sa prvok snaží nájsť.

```
void InsertHashCudzi(NODE *element)
{
    if(countHash==SIZE){
        printf("Error.\nTable is FULL\n");
        return;
    }
    long int probe=createHashHashCudzi(element->pers, SIZE);
    while(arr[probe]!=NULL)
    {
        probe=(probe+1)%SIZE;
    }
    arr[probe]=element;
    countHash++;
}
```

Vkladanie prvku sa správa tak, že si funkcia **Insert()** vygeneruje hash vkladaneho prvku, a potom sa ho snaží vložiť na daný index do tabuľky. Ak je tento index zabratý, pokúsi sa o rovnakú vec o jeden index ďalej.

Pri **testovaní** som použil podobný princíp ako pri mojom hashi.

Testovanie: AVL stromy / Red Black stromy.

Testovací scenár: 1 000 iterácií, 10 000 iterácií, 100 000 iterácií po 10 krát.

AVL Stromy:

```
the insertion of 1000 nodes took 0.001000 seconds  
the search of 1000 nodes took 0.000000 seconds
```

1 000:

Medzi jednotlivými testmi neboli takmer žiadne rozdiely, každý bol približne priemerný a rovnaký. Časový priemer pre vloženie aj vyhľadanie: takmer nula.

```
the insertion of 10000 nodes took 0.011000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes took 0.012000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes took 0.012000 seconds  
the search of 10000 nodes took 0.006000 seconds  
  
the insertion of 10000 nodes took 0.010000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes took 0.012000 seconds  
the search of 10000 nodes took 0.008000 seconds  
  
the insertion of 10000 nodes took 0.012000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes took 0.010000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes took 0.010000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes took 0.010000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes took 0.010000 seconds  
the search of 10000 nodes took 0.005000 seconds
```

10 000:

(Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.011 sekundy.

Časový priemer pre vyhľadanie: približne 0.0052 sekundy.

```
the insertion of 100000 nodes took 0.149000 seconds  
the search of 100000 nodes took 0.080000 seconds  
  
the insertion of 100000 nodes took 0.125000 seconds  
the search of 100000 nodes took 0.079000 seconds  
  
the insertion of 100000 nodes took 0.134000 seconds  
the search of 100000 nodes took 0.085000 seconds  
  
the insertion of 100000 nodes took 0.128000 seconds  
the search of 100000 nodes took 0.076000 seconds  
  
the insertion of 100000 nodes took 0.125000 seconds  
the search of 100000 nodes took 0.075000 seconds  
  
the insertion of 100000 nodes took 0.123000 seconds  
the search of 100000 nodes took 0.073000 seconds  
  
the insertion of 100000 nodes took 0.121000 seconds  
the search of 100000 nodes took 0.075000 seconds  
  
the insertion of 100000 nodes took 0.124000 seconds  
the search of 100000 nodes took 0.080000 seconds  
  
the insertion of 100000 nodes took 0.124000 seconds  
the search of 100000 nodes took 0.073000 seconds  
  
the insertion of 100000 nodes took 0.123000 seconds  
the search of 100000 nodes took 0.077000 seconds
```

100 000:

(Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely. Vloženie pre prvý test trvalo približne o 50% dlhšie ako priemer ostatných.

Časový priemer pre vloženie: približne 0.132 sekundy.

Časový priemer pre vyhľadanie: približne 0.0773 sekundy.

Red Black stromy:

```
the insertion of 1000 nodes took 0.001000 seconds
the search of 1000 nodes took 0.000000 seconds

the insertion of 1000 nodes took 0.002000 seconds
the search of 1000 nodes took 0.000000 seconds
```

1 000:

Medzi jednotlivými testmi neboli žiadne výrazné rozdiely, každý bol približne priemerný a rovnaký. Časový priemer pre vloženie aj vyhľadanie: takmer nula.

```
the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.003000 seconds

the insertion of 10000 nodes took 0.006000 seconds
the search of 10000 nodes took 0.004000 seconds

the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.004000 seconds

the insertion of 10000 nodes took 0.004000 seconds
the search of 10000 nodes took 0.004000 seconds

the insertion of 10000 nodes took 0.004000 seconds
the search of 10000 nodes took 0.005000 seconds

the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.004000 seconds

the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.005000 seconds

the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.004000 seconds

the insertion of 10000 nodes took 0.005000 seconds
the search of 10000 nodes took 0.004000 seconds
```

10 000: (Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.005 sekundy.

Časový priemer pre vyhľadanie: približne 0.0045 sekundy.

```
the insertion of 100000 nodes took 0.095000 seconds
the search of 100000 nodes took 0.066000 seconds

the insertion of 100000 nodes took 0.067000 seconds
the search of 100000 nodes took 0.059000 seconds

the insertion of 100000 nodes took 0.063000 seconds
the search of 100000 nodes took 0.060000 seconds

the insertion of 100000 nodes took 0.064000 seconds
the search of 100000 nodes took 0.061000 seconds

the insertion of 100000 nodes took 0.063000 seconds
the search of 100000 nodes took 0.060000 seconds

the insertion of 100000 nodes took 0.064000 seconds
the search of 100000 nodes took 0.065000 seconds

the insertion of 100000 nodes took 0.064000 seconds
the search of 100000 nodes took 0.066000 seconds

the insertion of 100000 nodes took 0.062000 seconds
the search of 100000 nodes took 0.060000 seconds

the insertion of 100000 nodes took 0.063000 seconds
the search of 100000 nodes took 0.056000 seconds

the insertion of 100000 nodes took 0.061000 seconds
the search of 100000 nodes took 0.059000 seconds
```

100 000: (Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely. Vloženie pre prvý test trvalo približne o 50% dlhšie ako priemer ostatných.

Časový priemer pre vloženie: približne 0.066 sekundy.

Zadanie 2 – Vyhľadávanie v dynamických množinách

Časový priemer pre vyhľadanie: približne 0.061 sekundy.

Zhrnutie porovnania týchto implementácií AVL a Red Black stromov:

Pri 1 000 iteráciách nebol zaznamenaný žiadny dôležitý rozdiel.

Pri 10 000 iteráciách sme zaznamenali približne dvojnásobne pomalšie vloženie prvkov do stromu pre AVL strom. Taktiež bol zaznamenaný malý rozdiel v rýchlosti vyhľadávania prvkov pre Red Black strom. V tejto kategórii jednoznačne vyhráva Red Black strom.

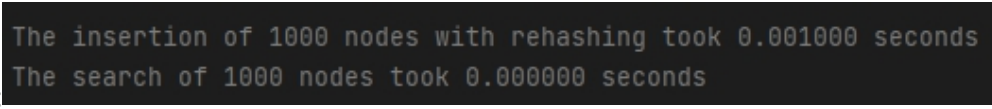
Pri 100 000 iteráciách sme zaznamenali pokračovanie trendu, ktorý sa začal vyskytovať pri predošlom teste.

Je možné vyvodiť záver, že z týchto implementácií je efektívnejší Red Black strom.

Testovanie: Hash tabuľka s riešením kolízií spájaným zoznamom / linear probingom.

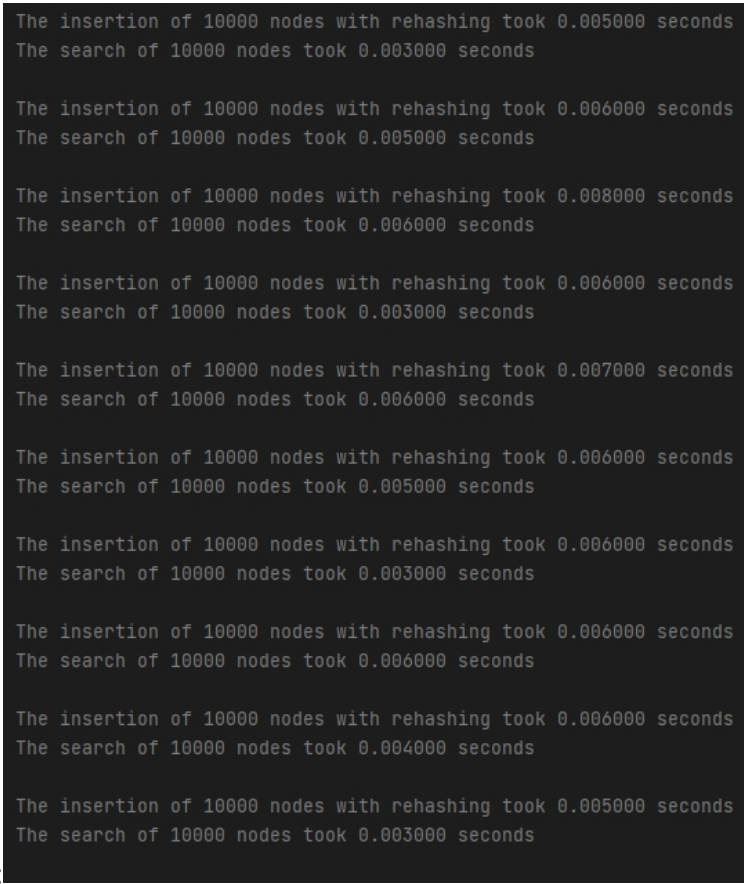
Testovací scenár: 1 000 iterácií, 10 000 iterácií, 100 000 iterácií po 10 krát.

Riešenie kolízií spájaným zoznamom:

1 000: 
The insertion of 1000 nodes with rehashing took 0.001000 seconds
The search of 1000 nodes took 0.000000 seconds

Rozdiely medzi jednotlivými testmi boli zanedbateľné.

Časový priemer pre vloženie aj vyhľadanie: takmer nula.


The insertion of 10000 nodes with rehashing took 0.005000 seconds
The search of 10000 nodes took 0.003000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.005000 seconds

The insertion of 10000 nodes with rehashing took 0.008000 seconds
The search of 10000 nodes took 0.006000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.003000 seconds

The insertion of 10000 nodes with rehashing took 0.007000 seconds
The search of 10000 nodes took 0.006000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.005000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.003000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.006000 seconds

The insertion of 10000 nodes with rehashing took 0.006000 seconds
The search of 10000 nodes took 0.004000 seconds

The insertion of 10000 nodes with rehashing took 0.005000 seconds
The search of 10000 nodes took 0.003000 seconds

10 000: (Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.0065 sekundy.

Časový priemer pre vyhľadanie: približne 0.005 sekundy.

```
The insertion of 100000 nodes with rehashing took 0.076000 seconds
The search of 100000 nodes took 0.042000 seconds

The insertion of 100000 nodes with rehashing took 0.060000 seconds
The search of 100000 nodes took 0.035000 seconds

The insertion of 100000 nodes with rehashing took 0.055000 seconds
The search of 100000 nodes took 0.034000 seconds

The insertion of 100000 nodes with rehashing took 0.055000 seconds
The search of 100000 nodes took 0.034000 seconds

The insertion of 100000 nodes with rehashing took 0.054000 seconds
The search of 100000 nodes took 0.035000 seconds

The insertion of 100000 nodes with rehashing took 0.054000 seconds
The search of 100000 nodes took 0.035000 seconds

The insertion of 100000 nodes with rehashing took 0.055000 seconds
The search of 100000 nodes took 0.036000 seconds

The insertion of 100000 nodes with rehashing took 0.054000 seconds
The search of 100000 nodes took 0.034000 seconds

The insertion of 100000 nodes with rehashing took 0.056000 seconds
The search of 100000 nodes took 0.034000 seconds

The insertion of 100000 nodes with rehashing took 0.055000 seconds
The search of 100000 nodes took 0.034000 seconds
```

100 000:

(Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.06 sekundy.

Časový priemer pre vyhľadanie: približne 0.037 sekundy.

Riešenie kolízií linear probingom:

Keďže v tejto implementácii sa nezväčšuje tabuľka, veľkosť tabuľky je vždy trojnásobok počtu požadovaných iterácií-

1 000:

```
the insertion of 1000 nodes(without rehashing) took 0.000000 seconds  
the search of 1000 nodes took 0.000000 seconds
```

Rozdiely medzi jednotlivými testmi boli zanedbateľné.

Časový priemer pre vloženie aj vyhľadanie: takmer nula.

```
the insertion of 10000 nodes(without rehashing) took 0.004000 seconds  
the search of 10000 nodes took 0.003000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.004000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.004000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.004000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.003000 seconds  
the search of 10000 nodes took 0.003000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.003000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.003000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.004000 seconds  
the search of 10000 nodes took 0.005000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.003000 seconds  
the search of 10000 nodes took 0.004000 seconds  
  
the insertion of 10000 nodes(without rehashing) took 0.003000 seconds  
the search of 10000 nodes took 0.003000 seconds
```

10 000:

(Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.0035 sekundy.

Časový priemer pre vyhľadanie: približne 0.0035 sekundy.

```
the insertion of 100000 nodes(without rehashing) took 0.049000 seconds
the search of 100000 nodes took 0.043000 seconds

the insertion of 100000 nodes(without rehashing) took 0.053000 seconds
the search of 100000 nodes took 0.037000 seconds

the insertion of 100000 nodes(without rehashing) took 0.035000 seconds
the search of 100000 nodes took 0.036000 seconds

the insertion of 100000 nodes(without rehashing) took 0.035000 seconds
the search of 100000 nodes took 0.035000 seconds

the insertion of 100000 nodes(without rehashing) took 0.034000 seconds
the search of 100000 nodes took 0.036000 seconds

the insertion of 100000 nodes(without rehashing) took 0.037000 seconds
the search of 100000 nodes took 0.036000 seconds
|
the insertion of 100000 nodes(without rehashing) took 0.036000 seconds
the search of 100000 nodes took 0.035000 seconds

the insertion of 100000 nodes(without rehashing) took 0.036000 seconds
the search of 100000 nodes took 0.036000 seconds

the insertion of 100000 nodes(without rehashing) took 0.035000 seconds
the search of 100000 nodes took 0.034000 seconds

the insertion of 100000 nodes(without rehashing) took 0.036000 seconds
the search of 100000 nodes took 0.036000 seconds
```

100 000:

(Pre zachovanie estetiky súboru zmenšené)

Medzi jednotlivými testmi boli minimálne rozdiely.

Časový priemer pre vloženie: približne 0.04 sekundy.

Časový priemer pre vyhľadanie: približne 0.035 sekundy.

Zhrnutie porovnania týchto implementácií hashovacej tabuľky s riešením kolízií spájaným zoznamom / linear probingom:

Implementácia s linear probingom neobsahuje zväčšovanie tabuľky, čo ovplyvňuje výsledkové časy.

Pri 1 000 iteráciách nebol zaznamenaný žiadny dôležitý rozdiel.

Pri 10 000 iteráciách sme zaznamenali takmer dvojnásobne rýchlejšie vloženie a aj vyhľadávanie prvkov s linear probingom.

Pri 100 000 iteráciách sme zaznamenali oveľa menší rozdiel medzi vložením v našich algoritmoch, pričom vyhľadávanie bolo v zásade rovnaké.

Poznámka: Ak pri linear probingu vytvoríme tabuľku s rovnakou veľkosťou, ako je počet iterácií, efektivita algoritmu výrazne klesá, a spájaný zoznam sa v porovnaní správa efektívnejšie.

Je možné vyvodiť záver, že oboje implementácie majú svoje svetlé stránky, a sú lepšie prispôsobené na rôzne počty iterácií. Spájaný zoznam lepšie zvláda prechod na vyššie množstvá iterácií oproti svojej efektivite na nižších, v porovnaní s linear probingom.

Linear probing je jednoduchší a efektívnejší pri nižších množstvách iterácií.

Poznámka: Pre jednoduchšiu implementáciu všetkých zdrojových kódov do jedného testovacieho som niektorým funkciám/štruktúram pridal na koniec názvu meno projektu:

AVL strom: Strom

Red Black strom: StromCudzi

Hash so spájaným zoznamom: Hash

Hash s linear probingom: HashCudzi