Lukáš Michalík, ID:103052
Pondelok 09:00

Slovenská Technická Univerzita v Bratislave
Fakulta informatiky a informačných technológií

Dátové štruktúry a algoritmy

Zadanie 2 – Vyhľadávanie v dynamických množinách

Úvod

Existuje veľké množstvo algoritmov, určených na efektívne vyhľadávanie prvkov v dynamických množinách:

binárne vyhľadávacie stromy, viaceré prístupy k ich vyvažovaniu, hashovanie a viaceré prístupy k riešeniu kolízií. Rôzne algoritmy sú vhodné pre rôzne situácie podľa charakteru spracovaných údajov, distribúcii hodnôt, vykonávaným operáciám, a pod. V tomto zadaní máte za úlohu implementovať a porovnať tieto prístupy.

V mojom riešení som porovnal vlastnú implementáciu AVL vyvažovacieho stromu a vlastnú implementáciu hashovania, ktoré rieši kolízie pomocou zreťazenia.

Vlastná implementácia AVL stromu

AVL strom je samovyvažovací binárny vyhľadávací strom. V AVL strome sa pre každý uzol rozdiel výšky dvoch podstromov detských uzlov líšia najviac o jednotku, preto je známy aj ako výškovo vyvážený. Hľadanie, vkladanie, a mazanie majú zložitosť O(log n) v priemernom aj najhoršom prípade. Pridávanie a mazanie môže vyžadovať vyváženie stromu jednou alebo viacerými rotáciami stromu.

Vkladanie prvku

```
TREE *insertTree(TREE *tree, char *name, int age) {

    int balanceFactor;

    //empty leaf
    if (tree == NULL){
        tree = createNewLeaf(name, age);
        return tree;
    }

    //go left   tree->data.name > data->name
    if ( strcmp(tree->data.name, name) == 1 ) {
        tree->left = insertTree(tree->left, name, age);
    }

    //go right      tree->data.name < data->name
    else if ( strcmp(tree->data.name, name) == -1 ) {
        tree->right = insertTree(tree->right, name, age);
    }

    //same data
    else {
        return tree;
    }

    //update height
    tree->height = 1 + calculateHeigh(tree);
```

Rotácie

```
TREE *rotateLeft(TREE *leaf) {

    TREE* leafRight = leaf->right;
    TREE* leafRightLeft = leafRight->left;

    leaf->right = leafRightLeft;
    leafRight->left = leaf;

    leaf->height = 1 + calculateHeigh(leaf);
    leafRight->height = 1 + calculateHeigh(leafRight);

    return leafRight;
}

TREE *rotateRight(TREE *leaf) {

    TREE* leafLeft = leaf->left;
    TREE* leafLeftRight = leafLeft->right;

    leaf->left = leafLeftRight;
    leafLeft->right = leaf;

    leaf->height = 1 + calculateHeigh(leaf);
    leafLeft->height = 1 + calculateHeigh(leafLeft);

    return leafLeft;
}
```

Vlastná implementácia HASH tabuľky (Reťazenie)

Moja implementácia HASH tabuľky rieši kolízie zreťazením. Veľkosť tabuľky je vždy prvočíslo. Je to preto, lebo prvočísla majú menej súdeliteľných čísel a teda nastáva menej kolízií.

Moja implementácia má jednu HASH funkciu, ktorá vypočíta pozíciu v tabuľke, na ktorú sa majú zapísať dáta.

Ak na tejto pozícii sa nič nenachádza, vloží sem dáta, ale pokiaľ táto pozícia už obsahuje nejaké dáta, tak sa na ne nadpojí. To znamená, že bude prechádzať cez jednotlivé dáta na tejto pozícii, až kým nepríde na koniec. Popri tom sa vždy kontroluje, či sa náhodu niektoré dáta z tejto pozície nerovnajú vkladaným, a ak áno, tak sa vkladanie preruší a nič sa nevloží.

Môže nastať prípad, kedy sa prekročí tzv. faktor naplnenia. Ten mám nastavený na hodnotu 0.2. Pokiaľ sa táto hodnota prekročí tak je nutné HASH tabuľku zväčšiť.

```
HASHTABLE *insertHTable(HASHTABLE *hTable, DATA *newData) {

    int index = hashFunction(newData);
    HASHTABLE *pos = (hTable + index);
    DATA *dataPos;
    //newData->next = NULL;

    if (pos->data == NULL) {
        pos->data = newData;
        pos->size = 1;
        inserts++;
        return hTable;
    }
    else {
        dataPos = pos->data;
        while (dataPos->next != NULL) {
            if ( strcmp(dataPos->name, newData->name) == 0 ) {
                free(newData);
                return hTable;
            }
            dataPos = dataPos->next;
        }
        if (dataPos->next == NULL) {
            if ( strcmp(dataPos->name, newData->name) == 0 ) {
                free(newData);
                return hTable;
            }
            inserts++;
            dataPos->next = newData;
            pos->size++;
        }
        float loadFactor = (float)(1.0 * pos->size / hTableSize);
        if (loadFactor >= 0.1) {
            //printf("Resize\n");
            hTable = resizeHTable(hTable);
        }
        return hTable;
    }
}
```

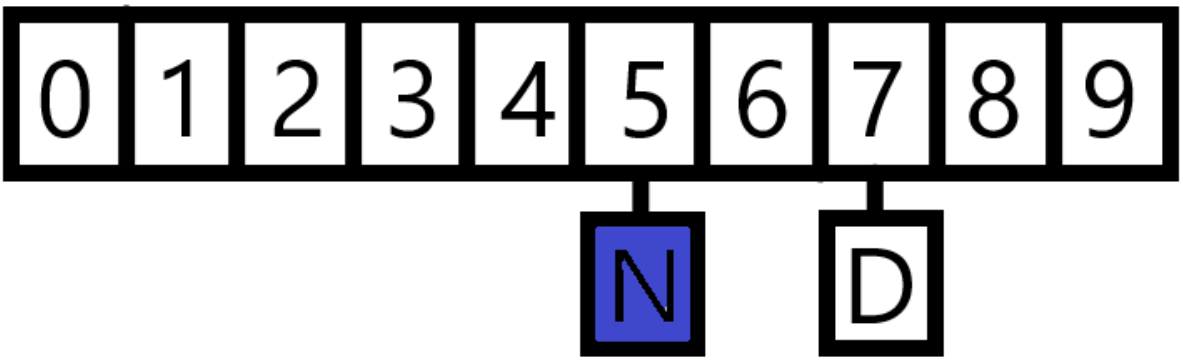
Zväčšovanie tabuľky

Zväčšovanie je riešené tak, že na začiatku sa odloží adresa začiatku starej tabuľky do dočasnej premennej. Ďalej sa vytvorí nová tabuľka s vyrátanou veľkosťou novej tabuľky pomocou prevzatých a upravených funkcií. Následne sa spustí proces kopírovania dát, kde všetky dáta zo starej tabuľky premiestnia do novej tabuľky a potom sa stará tabuľka uvoľní.

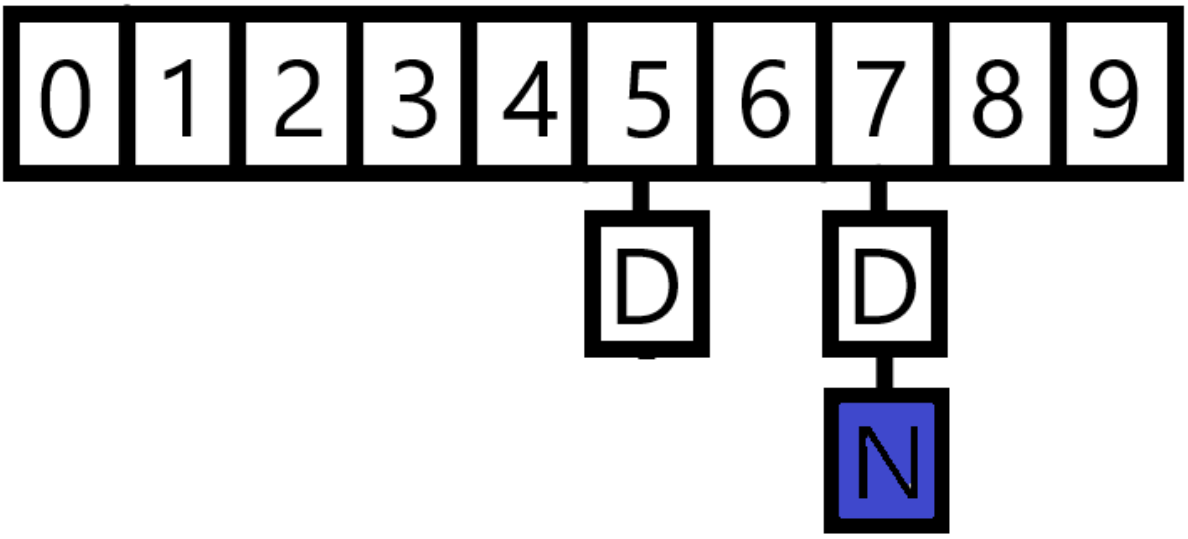
```
HASHTABLE *resizeHTable(HASHTABLE *hTable) {

    int oldHTableSize = hTableSize;
    int newHTableSize = nextPrime(2 * hTableSize);

    HASHTABLE *oldHTable = hTable;
    HASHTABLE *newHTable = createHTable(newHTableSize);
    DATA *oldData, *cpyData;

    for (int i = 0; i < oldHTableSize; i++) {
        oldData = (oldHTable + i)->data;
        if (oldData == NULL)
            continue;
        else {
            while (oldData != NULL) {

                newHTable = insertHTable(newHTable, oldData);
                inserts--;
                cpyData = oldData;
                oldData = oldData->next;
                cpyData->next = NULL;
            }
        }
    }
    free(oldHTable);
    return newHTable;
}
```

Reťazenie

Majme tabuľku o veľkosti 10, a chceme do nej vložiť nové DATA, pre ktoré HASH funkcia vypočítala pozíciu 5. Na pozícii 5 nie sú uložené žiadne dáta, a tak sa vložia nové dáta.



Ďalej chceme vložiť druhé nové DATA, pre ktoré bola vypočítaná pozícia 7. Avšak na pozícii 7 už sú uložené iné dáta. Tak sa tieto nové dáta umiestnia za ne, na koniec.



Testovanie

```
Insert AVL tree test pre 1000 prvkov trval 0.000000 sekund
Search AVL tree test pre 1000 prvkov trval 0.000000 sekund
Insert HASH table test pre 1000 prvkov trval 0.000000 sekund
Search HASH table test pre 1000 prvkov trval 0.001000 sekund
```

```
Insert AVL tree test pre 10000 prvkov trval 0.006000 sekund
Search AVL tree test pre 10000 prvkov trval 0.002000 sekund
Insert HASH table test pre 10000 prvkov trval 0.003000 sekund
Search HASH table test pre 10000 prvkov trval 0.004000 sekund
```

```
Insert AVL tree test pre 100000 prvkov trval 0.071000 sekund
Search AVL tree test pre 100000 prvkov trval 0.037000 sekund
Insert HASH table test pre 100000 prvkov trval 0.048000 sekund
Search HASH table test pre 100000 prvkov trval 0.044000 sekund
```

```
Insert AVL tree test pre 1000000 prvkov trval 1.437000 sekund
Search AVL tree test pre 1000000 prvkov trval 0.761000 sekund
Insert HASH table test pre 1000000 prvkov trval 0.406000 sekund
Search HASH table test pre 1000000 prvkov trval 0.460000 sekund
```