Oliver Leontiev, ID: 103027
DSA, Zadanie 2
2020

Vyhľadávanie v dynamických množinách

Dokumentácia k zadaniu 2

Úvod

Program sa skladá z piatich súborov: štyri hlavičkové s implementáciami a jeden s testami a funkciou main. Bola využitá x86 architektúra, čo znamená že smerníky sú 4 byty. Implementáciám zodpovedajú nasledovné súbory:

Vlastná implementácia červeno-čierneho stromu – *RB_BST_vlastny.h*

Prebratá implementácia AVL stromu – *AVL_BST_prebraty.h*

Vlastná hashovacia tabuľka so zreťazením – *HASHTABLE_vlastna.h*

Prebratá hashovacia tabuľka s otvoreným adresovaním – *HASHTABLE_prebrata.h*

Funkcia main a testy – *main.c*

Do množín sme vkladali iba celé kladné čísla typu integer (nepoužili sme unsigned int lebo nebol použitý v prebratých implementáciách).

V nasledujúcej dokumentácii bude poskytnutý krátky komentár ku každej implementácii, ktorý bude zahŕňať stručný opis priestorovej a časovej efektivity. Následne priebeh a vyhodnotenie testovania.

**Komentár sa vzťahuje na konkrétne prevedenie v programe, nie na všeobecnú množinu.*

Implementácie dynamických množín

Červeno-čierny strom

Jedna bunka (prvok) červeno-čierneho stromu obsahuje **kľúč** (dáta, ktoré uchováva) informáciu o **zafarbení** – boolovska hodnota (je alebo nie je červená – true/false) a tri smerníky: **rodič**, **ľavé dieťa** a **pravé dieťa**.

```
typedef struct RBNode {
	int key;
	struct RBNode* parent;
	struct RBNode* left_child;
	struct RBNode* right_child;
	bool red;
}RBNODE;
```

Práve smerník na rodiča je niečo, čo sa pri iných vyhľadávacích stromoch dá ušetriť (najmä pri rekurzívnom prevedení). Na označenie zafarbenia nám stačí jeden bit, ktorý sa dá schovať napríklad do kľúča. Toto potenciálne zefektívnenie sme neimplementovali. Využívame teda jeden byte, to je ale stále menej ako integer na zapamätanie si výšky, ako pri AVL strome.

Č-č strom teda využíva teoreticky (bez zarovnávania) **n*17 bytov**, kde n je počet buniek. V našom prípade je 17 zarovnaná kompilátorom na 20 bytov: takže prakticky n*20. **Priestorová efektivita je O(n).**

Vkladanie aj vyhľadávanie prebieha iteratívne a podľa pravidiel č-č stromu. Prevedenie potenciálne spomaľuje využitie pomocných funkcií (whichChildRB, getSiblingRB), ktoré ale zvyšujú čitateľnosť.

Časová efektivita je O(log n) pre vkladanie aj vyhľadávanie, kde n je počet buniek stromu. Treba ale poznamenať, že vyvažovanie je menej striktné ako pri AVL stromoch a maximálna výška stromu je tiež vyššia. To znamená rýchlejšie (menej striktné) vkladanie do stromu, ale pomalšie vyhľadávanie, keďže je potom strom menej vyvážený.

AVL strom

AVL strom bol prebratý z internetového portálu GeeksForGeeks:
https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Jedna bunka AVL stromu obsahuje **kľúč**, smerníky na **ľavé a pravé dieťa** a **výšku** v strome.

```
typedef struct AVLNode
{
	int key;
	struct AVLNode* left;
	struct AVLNode* right;
	int height;
}AVLNODE;
```

Rodič je vynechaný najmä preto, že aj vkladanie aj vyhľadávanie je implementované **rekurzívne**, a teda ho nepotrebujeme. Musíme si ale pamätať výšku a to je integer, čiže 4 byty. Z toho vyplýva, že aj keď je ušetrený smerník na rodiča, teoretická priestorová efektivita je iba o 1 byte lepšia ako č-č strom. Ukáže sa, že kvôli rekurzívnemu vkladaniu a vyhľadávaniu bude AVL strom konzistentne časovo veľmi neefektívny.

AVL strom teda zaberá **n*16 bytov**, kde n je počet buniek (aj so zarovnávaním). **Priestorová efektivita je O(n).**

Časová efektivita je O(log n) pre vkladanie aj vyhľadávanie, kde n je počet buniek v strome. AVL strom ma nižšiu maximálnu výšku ako č-č strom, vďaka striktnejšiemu vyvažovaniu, ktoré je ale tým pádom pomalšie. Z toho vyplýva, že vkladanie do AVL stromu je pomalšie a vyhľadávanie rýchlejšie v porovnaní s č-č stromom. Toto sa v našom strome neprejaví kvôli rekurzii, ktorá AVL strom značne spomalí.

**Poznámka: zdá sa pravdepodobné, že pri bežných implementáciách oboch stromov bude č-č strom zaberať menej pamäte, keďže smerník na rodiča budú mať obidva a ak schováme flag pre zafarbenie pri č-č strome, tak AVL strom má o jeden integer väčšiu bunku.*

Oliver Leontiev, ID: 103027
DSA, Zadanie 2
2020

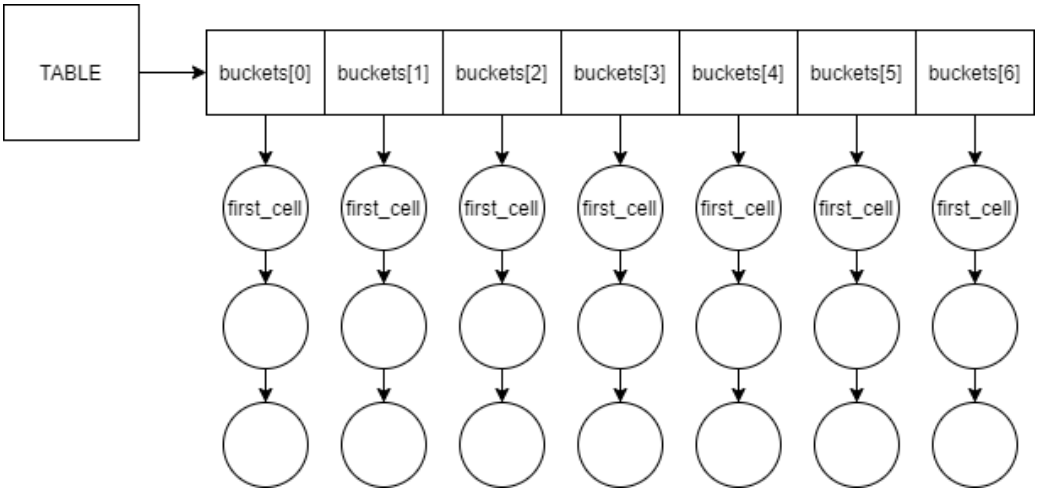
Hashovacia tabuľka so zreťazením

Táto hashovacia tabuľka využíva tri štruktúry, ktorých vzťah je zobrazený na obrázku. **TABLE** obsahuje informáciu o veľkosti tabuľky (**size**), počte vložených prvkov (**cells**) a poli smerníkov na spájané zoznamy (**buckets**). Tieto smerníky sú zabalené v štruktúre **BUCKET** pre lepšiu zrozumiteľnosť. Bunka tabuľky (**CELL**) drží kľúč (**key**) a smerník na ďalšiu bunku v zozname (**next**).

```
typedef struct cell {
	int key;
	struct cell* next;
}CELL;

typedef struct bucket {
	CELL* first_cell;
}BUCKET;

typedef struct table {
	int size;
	int cells;
	BUCKET* buckets;
}TABLE;
```



Takúto sme zvolili hashovaciu funkciu:

```
unsigned long long xorshift(unsigned long long key) {
	key = key ^ (key >> 17);
	key = key ^ (key << 15);
	return key % DIV_CONST;
}

int hashOwn(int key,int tableSize) {
	unsigned long long hashkey = (unsigned long long)key * MUL_CONST;
	hashkey = xorshift(hashkey);
	hashkey = hashkey*RN_ONE + RN_TWO;
	return hashkey%tableSize;
}
```

**Poznámka: Vzhľadom na to, že veľkosť tabuľky je rovnakého typu, ako vkladané dáta (integer) a nesnažíme sa dáta utajiť, tak ideálna hashovacia funkcia by bola iba key % tablesize, lebo ak*

je veľkosť tabuľky prvočíslo, rozptyl bude perfektný. Naša hashovacia funkcia teda výrazne znižuje časovú efektivitu.

Faktor preťaženia držíme pod 0.5 a pri prekročení sa tabuľka zväčší na prvočíselnú veľkosť, najbližšiu dvojnásobku. Toto zväčšenie prebieha prekopírovaním kľúčov do novej tabuľky a vymazaním starej. Hľadanie prvočísla zaberie nejaký čas.

Priestorová efektivita je O(n), kde n je počet buniek.

Časová efektivita je O(1) pre vyhľadávanie aj vkladanie, pokiaľ neboli žiadne kolízie. V praxi je to o niečo viac, keď sa prehľadávajú aj spájané zoznamy.

Hashovacia tabuľka s otvoreným adresovaním

Hashovacia tabuľka bola prebratá z githubu užívateľa anholt:

https://github.com/anholt/hash_table (int_set.h a int_set.c)

V tejto tabuľke má každá bunka tag, či je obsadená alebo vymazaná. Vymazávanie nepoužívame.

Tabuľka je štruktúra, ktorá si drží nasledovné informácie: pole buniek (**table**), veľkosť tabuľky (**size**), posun pri kolízii (**rehash**), maximálny počet buniek pred zväčšením (**max_entries**), index veľkosti tabuľky v predpripravenom poli (**size_index**), počet buniek (**entries**) a počet vymazaných buniek (**deleted entries**)

```
struct int_set_entry {
	uint32_t value;
	unsigned int occupied : 1;
	unsigned int deleted : 1;
};

struct int_set {
	struct int_set_entry* table;
	uint32_t size;
	uint32_t rehash;
	uint32_t max_entries;
	uint32_t size_index;
	uint32_t entries;
	uint32_t deleted_entries;
};
```

Hashovacia funkcia je iba kompresná.

```
hash_address = value % set->size;
```

A kolízie sú riešené hľadaním voľného miesta pomocou hodnoty rehash.

```
double_hash = 1 + value % set->rehash;

hash_address = (hash_address + double_hash) % set->size;
```

Zväčšenie tabuľky je riešené vybratím novej veľkosti z predpripraveného statického poľa a prekopírovaním prvkov do novej tabuľky. Stará sa potom vymaže.

Priestorová efektivita je O(n), kde n je počet buniek.

Časová efektivita je O(1) pre vyhľadávanie aj vkladanie, pokiaľ neboli žiadne kolízie. V praxi je to o niečo viac, keď sa hľadá nové miesto po kolízii.

Testovanie

Používame 2 druhy testov: **IntervalTest** a **randomTest**.

IntervalTest testuje vkladanie a vyhľadávanie čísel z intervalu, pričom si môžeme zvoliť, či chceme čísla premiešať pred vkladaním a pred vyhľadávaním.

randomTest testuje vkladanie určeného počtu náhodných čísel, bez opakovania čísel. Vygenerovať čísla mu môže trvať dlho, keďže generuje rôzne čísla taktikou pokus-omyl, preto zadávame menší počet čísel.

Obe tabuľky začínajú na veľkosti 5 a zväčšujú sa svojimi internými algoritmami.

Testy, v ktorých nie je vidno zmenu či zaujímavú informáciu neuvádzame, ale v programe sú.

Vkladanie a vyhľadávanie za sebou idúcich čísel

Pri vkladaní a vyhľadávaní čísel od 0 do 1000000 vzostupne bola najrýchlejšia hashovacia tabuľka s otvorením adresovaním a najpomalší bol AVL strom.

```
Insert numbers from 0 to 1000000 in ascending order and search for them 10 times

My red-black tree insert time: 541.000000 ms
My red-black tree search time: 1089.000000 ms
Taken AVL tree insert time: 2110.000000 ms *huge because of recursive insert
Taken AVL tree search time : 3143.000000 ms

My hash table with chaining insert time: 1382.000000 ms
My hash table with chaining search time: 2550.000000 ms
Taken hash table with open addressing insert time: 153.000000 ms
Taken hash table with open addressing search time: 598.000000 ms
```

Pri vkladaní a vyhľadávaní čísel od 0 do 1000000 v náhodnom poradí bola najrýchlejšia hashovacia tabuľka s otvorením adresovaním a najpomalší bol AVL strom. Oba stromy výrazne spomalili, a to najmä pri vyhľadávaní, zatiaľ čo hashovacie tabuľky si zachovali rýchlosť. Tabuľka so zreťazením výrazne predbehla č-č strom pri vyhľadávaní. Z toho vyplýva, že stromy sú rýchlejšie pri vkladaní postupností.

```
Insert numbers from 0 to 1000000 in random order and search for them 10 times

My red-black tree insert time: 867.000000 ms
My red-black tree search time: 5201.000000 ms
Taken AVL tree insert time: 3079.000000 ms *huge because of recursive insert
Taken AVL tree search time : 7827.000000 ms

My hash table with chaining insert time: 1356.000000 ms
My hash table with chaining search time: 2563.000000 ms
Taken hash table with open addressing insert time: 259.000000 ms
Taken hash table with open addressing search time: 1032.000000 ms
```

Pokiaľ čísla premiešame pred vyhľadávaním, nič sa nezmení, preto to ďalej robiť nebudeme. Stojí za zmienku že č-č strom je stále rýchlejší vo vyhľadávaní ako hashovacia tabuľka so zreťazením, a teda záleží na poradí, v ktorom sú prvky vložené a nie vyhľadávané.

```
Insert numbers from 0 to 1000000 in ascending order and search for them 10 times
 Numbers are shuffled before searching

My red-black tree insert time: 465.000000 ms
My red-black tree search time: 1020.000000 ms
Taken AVL tree insert time: 2112.000000 ms *huge because of recursive insert
Taken AVL tree search time : 3103.000000 ms

My hash table with chaining insert time: 1337.000000 ms
My hash table with chaining search time: 2505.000000 ms
Taken hash table with open addressing insert time: 146.000000 ms
Taken hash table with open addressing search time: 587.000000 ms
```

Pri vkladaní a vyhľadávaní čísel od 0 do 300 000 vzostupne bola najrýchlejšia hashovacia tabuľka s otvorením adresovaním a najpomalší bol AVL strom. Stromy majú menšiu hĺbku, ale tabuľky sa musia menej krát zväčšiť.

```
Insert numbers from 0 to 300000 in ascending order and search for them 10 times

My red-black tree insert time: 140.000000 ms
My red-black tree search time: 301.000000 ms
Taken AVL tree insert time: 584.000000 ms *huge because of recursive insert
Taken AVL tree search time : 863.000000 ms

My hash table with chaining insert time: 333.000000 ms
My hash table with chaining search time: 626.000000 ms
Taken hash table with open addressing insert time: 61.000000 ms
Taken hash table with open addressing search time: 179.000000 ms
```

Pri vkladaní v náhodnom poradí si môžeme všimnúť, že pri menšom počte prvkov stromy zrýchlili ostrejšie ako tabuľky. A teda pri menšom počte prvkov stromy zvládajú náhodné poradie lepšie ako pri väčšom počte prvkov.

```
Insert numbers from 0 to 300000 in random order and search for them 10 times

My red-black tree insert time: 194.000000 ms
My red-black tree search time: 1032.000000 ms
Taken AVL tree insert time: 762.000000 ms *huge because of recursive insert
Taken AVL tree search time : 1716.000000 ms

My hash table with chaining insert time: 323.000000 ms
My hash table with chaining search time: 634.000000 ms
Taken hash table with open addressing insert time: 97.000000 ms
Taken hash table with open addressing search time: 213.000000 ms
```

Vkladanie náhodných čísel

Pri vkladaní a vyhľadaní 200 000 náhodných čísel bola najrýchlejšia hashovacia tabuľka s otvoreným adresovaním a najpomalší bol AVL strom. Č-č strom predbehol hashovaciu tabuľku so zreťazením vo vkladaní, tá bola ale rýchlejšia pri vyhľadávaní.

Oliver Leontiev, ID: 103027
DSA, Zadanie 2
2020

```
Insert 200000 random numbers and search for them 10 times

My red-black tree insert time: 119.000000 ms
My red-black tree search time: 601.000000 ms
Taken AVL tree insert time: 508.000000 ms *huge because of recursive insert
Taken AVL tree search time : 1015.000000 ms

My hash table with chaining insert time: 292.000000 ms
My hash table with chaining search time: 449.000000 ms
Taken hash table with open addressing insert time: 60.000000 ms
Taken hash table with open addressing search time: 229.000000 ms
```

Vyhodnotenie

Vo všetkých prípadoch premiešanie čísel pred vyhľadávaním nič nezmenilo na výsledku, preto tieto prípady neuvádzame. Taktiež sme skúšali vkladať väčšie čísla, čo tiež výsledok nijako neovplyvnilo.

AVL strom bol konzistentne najpomalší, čo je spôsobené najmä rekurzívnym prevedením. Naopak hashovacia tabuľka s otvoreným adresovaním bola konzistentne najrýchlejšia, čo prisudzujeme najmä najjednoduchšej hashovacej funkcii spolu s perfektným rozptylom. Taktiež predpripravené veľkosti tabuľky ju urýchlili, lebo nemusela robiť zbytočné výpočty.

Červeno-čierny strom vkladá prvky rýchlejšie, ako hashovacia tabuľka so zreťazením. Toto by sa dalo vysvetliť časom spotrebovaným na zväčšenie tabuľky v kombínacií s hashovacou funkciou. Hashovacia tabuľka prvky rýchlejšie vyhľadáva, vďaka tomu že nemusí prechádzať tak veľa prvkami. Výnimkou je ak vkladáme za sebou idúcu postupnosť prvkov, vtedy je č-č strom rýchlejší aj pri vyhľadávaní.

Nepodarilo sa nám nájsť efektívne využitie rekurzívneho AVL stromu. Z výsledkov vyplýva, že hashovacie tabuľky sú okrem výnimočných prípradov, vždy rýchlejšie pri vyhľadávaní ako binárne stromy. Rýchlosť tabuľky záleží od hashovacej funkcie a rýchlosti zväčšovania, čomu pripisujeme veľké rozdiely medzi tabuľkami v našom programe.