Ľubomír Dlhý, ID: 102897

VYHĽADÁVANIE V DYNAMICKÝCH MNOŽINÁCH

Ľubomír Dlhý, ID: 102897

ÚVOD

Úlohou zadania bolo porovnať viacero implementácií dátových štruktúr z hľadiska efektivity operácií **insert** a **search** v rozličných situáciách.

Ja som sa rozhodol pre implementáciu **Splay stromu** (*súbor mySplay.c*) a **Hashovej tabuľky** (*súbor MyHashTable.c*) s riešením kolízií pomocou **zreťazovania**.

Na porovnanie som prevzal implementáciu **Red-Black stromu** (*súbor rbtree.c*) a **Hashovej tabuľky** (*súbor hashmap.c*) s riešením kolízií pomocou **otvoreného adresovania**, presnejšie **linear probing**.

Všetky implementácie ukladajú celočíselné dátove premenné a majú ošetrené duplicitné vkladanie hodnôt.

Ľubomír Dlhý, ID: 102897

SPLAY STROM (VLASTNÁ IMPLEMENTÁCIA)

Splay stromy majú funkciu ***splay***, ktorá sa zavolá po vložení a po vyhľadaní prvku v strome. Táto funkcia pomocou vhodných rotácií nastaví vkladaný/hľadaný vrchol na koreň stromu, čo v praxi znamená, že nedávno hľadané alebo vkladané vrcholy sú bližšie ku koreňu. Splay stromy sa vyvažujú iba pomocou funkcie splay, čo znamená, že v niektorých krajných prípadoch (napr. vkladanie hodnôt v zostupnom/vzostupnom poradí) je strom nevyvážený.

Štruktúra:
int **data**
*node** **left**
*node** **right**
*node** **parent**

```
typedef struct SplayNode {
    int data;
    struct SplayNode* left, *right, *parent;
} SPLAYNODE;
```

Algoritmus vkladania: binárnym vyhľadávaním nájdeme vhodnú pozíciu na vloženie, vrchol vložíme a zavoláme funkciu splay na vkladaný vrchol

Algoritmus hľadania: binárnym vyhľadávaním sa pokúsime nájsť vrchol, po nájdení na vrchol zavoláme funkciu splay

Funkcia Splay: Funkcia Splay vykoná vhodné rotácie, aby sa vrchol v parametri funkcie stal novým koreňom stromu. Môžu nastať nasledovné situácie:

1. Vrchol v parametri je root → koniec funkcie
2. Vrchol v parametri je dieťa rootu → Zig alebo Zag rotácia rootu
3. Vrchol v parametri nie je root ani dieťa rootu:
    - 3.a) LeftLeft prípad → Zig-Zig rotácia
    - 3.b) LeftRight prípad → Zag-Zig rotácia
    - 3.c) RightRight prípad → Zag-Zag rotácia
    - 3.d) RightLeft prípad → Zig-Zag rotácia

Rotácie:
Zig rotácie je ekvivalent pravej rotácie
Zag rotácie je ekvivalent ľavej rotácie
Zig-Zig rotácia je Zig rotácia starého rodiča, potom Zig rotácia rodiča
Zag-Zag rotácia je Zag rotácia starého rodiča, potom Zag rotácia rodiča
Zig-Zag rotácia je Zig rotácia rodiča, potom Zag rotácia starého rodiča
Zag-Zig rotácia je Zag rotácia rodiča, potom Zig rotácia starého rodiča

Výhody Splay stromov: Ak hľadáme/vkladáme nedávno pristupované vrcholy, splay strom je efektívnejší. (niekedy dokonca v konštantnom čase).

Nevýhody Splay stromov: Ak vkladáme/hľadáme v zostupnom/vzostupnom poradí, strom bude nevyvážený.

Ľubomír Dlhý, ID: 102897

RED-BLACK STROM (PREVZATÁ IMPLEMENTÁCIA)

Zdroj: https://github.com/supernum/rbtree

Red-Black stromy sú samovyvažovacie stromy, ktoré v svojej štruktúre využívajú ďalšiu vlastnosť – farbu vrcholov. Farba môže byť buď červená alebo čierna. Red-Black stromy majú striktné pravidlá, vďaka ktorým je strom vyvážený.

Štruktúra:

*node** **left**
*node** **right**
*node** **parent**
unsigned char **color**
long **key**

```
typedef struct rbt_tree_node {
    struct rbt_tree_node* parent;
    struct rbt_tree_node* left;
    struct rbt_tree_node* right;
    unsigned char color;
    long key;
} rbt_tree_node_t;
```

Algoritmus vkladania: binárnym vyhľadávaním nájdeme vhodnú pozíciu na vloženie, vrchol vložíme a postupujeme podľa nasledovného algoritmu:

1. Ak je strom prázdny, vložíme čierny uzol a ukončíme
2. Ak strom nie je prázdny, vložíme červený uzol
3. Ak je rodič sledovaného uzla čierny, ukončíme.
4. Ak je rodič sledovaného uzla červený, sledujeme farbu strýka (súrodenca rodiča):
    4a. strýko má červenú farbu – zmeníme farbu rodičovi, strýkovi a starému rodičovi (ak nie je root), potom spustíme krok č. 4 so starým rodičom
    4b. strýko má čiernu farbu – spravíme vhodnú rotáciu a vhodne zmeníme farbu

Algoritmus hľadania: binárnym vyhľadávaním vyhľadáme vrchol

Pravidlá Red-Black stromov:

1. Každý vrchol je buď čierny alebo červený
2. Koreň je vždy čiernej farby
3. NULL pointre (neexistujúce deti) majú čiernu farbu
4. Každý červený vrchol má dve čierne deti (rodič a dieťa nesmú mať červenú farbu)
5. Všetky možné cesty z ľubovoľného vrcholu do NULL pointra majú rovnaký počet čiernych vrcholov

Výhody Red-Black stromov: Sú časovo efektívne pri vkladaní prvkov, keďže vyžadujú menej rotácií ako AVL / Splay stromy.

Nevýhody Red-Black stromov: Keďže Red-Black stromy nie sú perfektne vyvážené, vyhľadávanie je pomalšie v porovnaní s AVL stromami.

Ľubomír Dlhý, ID: 102897

HASH TABUĽKA S REŤAZOVANÍM (VLASTNÁ IMPLEMENTÁCIA)

Hashové tabuľky sú dynamické množiny, v ktorých sú prvky uložené v poli v nezoradenom poradí (na rozdiel od stromov). Index prvku v poli je daný hashovou funkciou. Ak dva prvky majú rovnaký index, vzniká kolízia. Na riešenie kolízií som použil zreťazovanie - spájané zoznamy. To znamená, že v každom políčku poľa je spájaný zoznam a novovkladaný prvok sa pridá na začiatok daného spájaného zoznamu. Pri naplnení poľa 80% prvkov sa tabuľka dvojnásobne zväčší.

Štruktúra: *Tabuľka*

```
typedef struct hashtablee {
    unsigned int size;
    unsigned int filledSize;
    HASHTABLE_ENTRY** array;
}HASHTABLE;
```

Spájaný Zoznam

```
typedef struct hashtable_entry {
    int key;
    struct hashtable_entry* next;
}HASHTABLE_ENTRY;
```

Algoritmus vkladania: V prípade potreby zmeny veľkosti tabuľky (pri naplnení tabuľky na 80%), tabuľku dvojnásobne zväčším

Hashovou funkciou vypočítame index prvku v poli

Overím, či daný prvok v poli už existuje (prejdením spájaného zoznamu na vypočítanom indexe). Ak existuje, ukončím vkladanie.

Prvok vložím na začiatok spájaného zoznamu na daný index v poli

Algoritmus hľadania: Hashovou funkciou vypočítame index prvku v poli, prejdením spájaného zoznamu na danom indexe overíme existenciu prvku

Hashová funkcia: key % size

```
static unsigned int _hash(int key, unsigned int size) {
    return key > 0 ? key % size : (-key) % size;
}
```

Základná veľkosť tabuľky: 8
Hranica zväčšenia tabuľky: 80%
Koeficient zväčšenia tabuľky: 2

Výhody Hash Tabuľky s reťazovaním: Hľadanie je rýchlejšie oproti otvorenému adresovaniu kvôli menšiemu počtu výpočtov.

Nevýhody Hash Tabuľky s reťazovaním: Vyžaduje ďalšiu štruktúru (spájaný zoznam), prvky sú mimo tabuľky.

Ľubomír Dlhý, ID: 102897

HASH TABUĽKA S OTVORENÝM ADRESOVANÍM – LINEAR PROBING (PREVZATÁ IMPLEMENTÁCIA)

Zdroj: https://github.com/Jmc18134/hashmap

Na riešenie kolízií je použité otvorené adresovanie – lineárne sa hľadá prvé voľné okienko a do toho sa vloží prvok. Táto implementácia vkladá pár kľúč -> hodnota. Nastavujem kľúč a hodnotu na tú istú hodnotu.

Štruktúra: *Tabuľka*

```
struct HashMap {
    struct DictEntry** table;
    size_t size;
    size_t capacity;
};
```

Vkladaný prvok

```
struct DictEntry {
    int key;
    int value;
    unsigned long hash;
};
```

Algoritmus vkladania: V prípade potreby zmeny veľkosti tabuľky (pri naplnení tabuľky na 2/3), tabuľka zväčší svoju veľkosť na najbližšie prvočíslo k dvojnásobku svojej veľkosti
Overí, či daný prvok v poli už existuje. Ak existuje, aktualizuje hodnotu a skončí.
Prvok vloží na vhodné miesto (vhodné miesto nájdeme lineárnym hľadaním prvého voľného okienka od vypočítaného indexu)

Algoritmus hľadania: Hashovou funkciou vypočítame index prvku v poli, lineárne prechádzame okienka a porovnávame hodnoty kľúčov až po prvé voľné miesto

Hashová funkcia:

```
static inline unsigned long inthash(int num)
{
    return (unsigned long)num * 2654435761;
}
```

Základná veľkosť tabuľky: 7
Hranica zväčšenia tabuľky: 2/3 –> 66,67%
Veľkosť zväčšenej tabuľky: prvé prvočíslo po 2-násobku pôvodnej veľkosti

Ľubomír Dlhý, ID: 102897

TESTOVANIE

Na testovanie efektívnosti vkladania a hľadania prvkov v rôznych dynamických množinách som vytvoril program, ktorý skúša rôzne scenáre (rôzne usporiadanie vkladaných prvkov) na rôznych veľkostiach a tieto dátové štruktúry uvoľňuje z haldy pamäte, takže program vie bežať aj na veľkom počte testov.

TESTOVANIE VKLADANIA PRVKOV

```
void testInsertAll(int orderCase) {
    testerInsertions(100000,10, orderCase);         // 100 000 testov, v kazdom 10 insertov
    testerInsertions(10000, 100, orderCase);        // 10 000 testov, v kazdom 100 insertov
    testerInsertions(1000, 500, orderCase);         // 1 000 testov, v kazdom 500 insertov
    testerInsertions(1000, 1000, orderCase);        // 1 000 testov, v kazdom 1000 insertov
    testerInsertions(100, 5000, orderCase);         // 100 testov, v kazdom 5000 insertov
    testerInsertions(100, 10000, orderCase);        // 100 testov, v kazdom 10 000 insertov
    testerInsertions(20, 100000, orderCase);        // 20 testov, v kazdom 100 000 insertov
    testerInsertions(5, 1000000, orderCase);        // 5 testov, v kazdom 1 000 000 insertov
    testerInsertions(1, 10000000, orderCase);       // 1 test, v kazdom 10 000 000 insertov
}
```

Hlavná funkcia na vykonanie všetkých testov na vkladanie - všetky štruktúry používajú rovnaké vkladané hodnoty

```
/* orderCase:
    1 - asc insert
    2 - desc insert
    3 - random insert 0-1000M with duplicit values
    4 - random insert without duplicit values
*/
```

Poradie vkladaných hodnôt určované cez argument

```
void testSplayInsert(int iterations,int insertions, int* pole) {
    int failed = 0;
    SPLAYNODE** rootArray = calloc(iterations, sizeof(SPLAYNODE*));


    clock_t t1 = clock();
    for (int j = 0; j < iterations;j++) {
        for (int i = 0; i < insertions; i++) {
            if (!insertSplay(pole[i], &rootArray[j]))
                failed++;
        }
    }
    clock_t t2 = clock();

    for (int j = 0; j < iterations; j++) {
        removesplaytree(rootArray[j]);
    }
    free(rootArray);
```

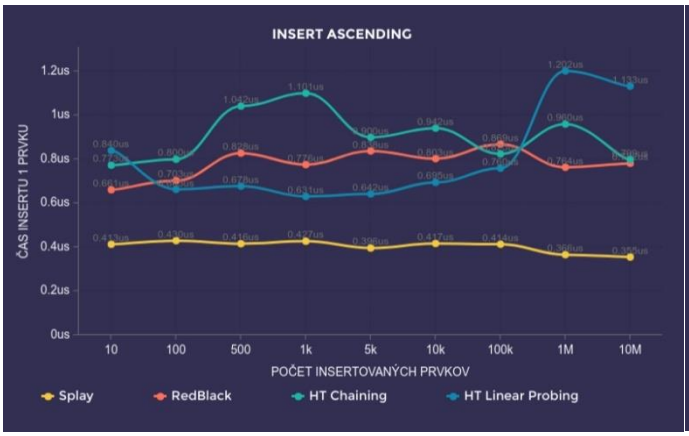
Priebeh testov v jednotlivých štruktúrach – vytvorím toľko štruktúr, koľko je počet testov a potom každú štruktúru naplním určitým počtom prvkov – napríklad pri 100 000 testoch a 10 insertoch sa vytvorí 100 000 štruktúr a do každej sa vloží 10 prvkov. Meriam čas vykonania všetkých (v tomto prípade 100 000) testov a vyjadrím čas insertu jedného prvku v mikrosekundách. Naplnené štruktúry potom uvoľním, čím dokážem spustiť viacero testov a scenárov v jednom behu programu.

```
int* pole = (int*)malloc(insertions * sizeof(int));

for (unsigned int j = 0; j < insertions; j++){ ... }

if (orderCase == 4)
    shuffle(pole,insertions);

testSplayInsert(iterations,insertions,pole);

testRbInsert(iterations,insertions,pole);

testMyHashInsert(iterations, insertions, pole);

testProbingHashInsert(iterations, insertions, pole);
printf("\n");
```

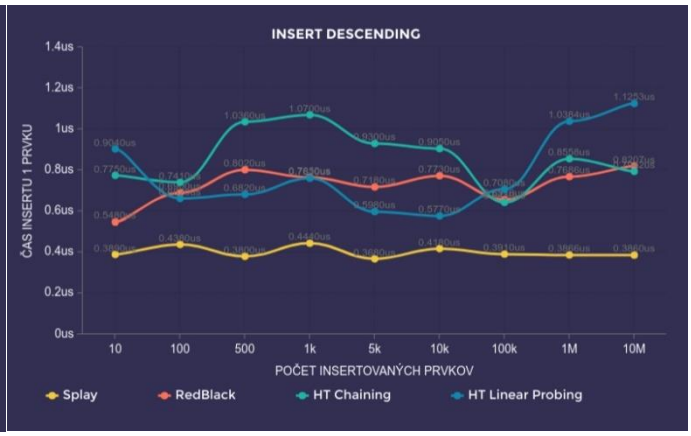
Všetky štruktúry používajú rovnakú množinu vkladaných prvkov – vkladané hodnoty si vygenerejum do poľa podľa daného scenáru (napríklad usporiadanie vzostupne, zostupne, náhodné usporiadanie) a do jednotlivých testovacích funkcií dávam toto pole cez argument, čím zaručím rovnaké podmienky.

Ľubomír Dlhý, ID: 102897

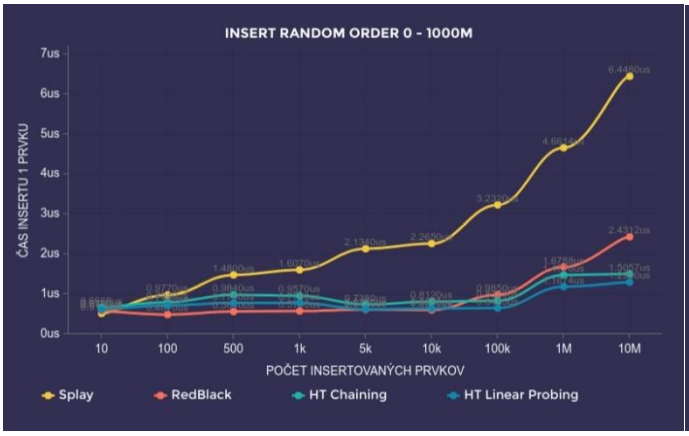
VÝSLEDKY TESTOVANIA VKLADANIA PRVKOV



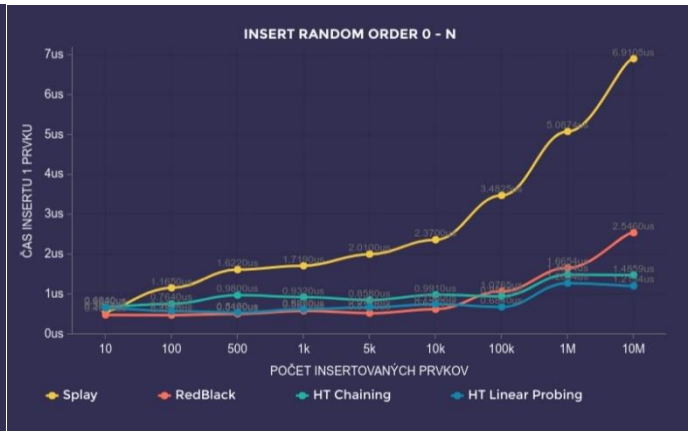
1a vkladanie vo vzostupnom poradí



1b vkladanie v zostupnom poradí



1c vkladanie v náhodnom poradí I. (rand)



1d vkladanie v náhodnom poradí II. (shuffle)

Scenáre vkladania vo **vzostupnom** (*1a*) a **zostupnom** (*1b*) poradí sú takmer totožné:
Splay strom **sa nevyvažuje**, pri každom vložení spraví iba jednu rotáciu (Zig alebo Zag) a prvky sú v jednej dlhej vetve, preto sa čas vkladania nemení.
RedBlack strom sa vyvažuje a vďaka jeho vlastnostiam je v niektorých momentoch dokonca rýchlejší ako hashové tabuľky.
Hashová tabuľka s reťazením dosť fluktuuje, čo je spôsobené častou zmenou veľkosti tabuľky. Tabuľka nemá žiadne kolízie kvôli vzostupnému/zostupnému poradiu hodnôt.
Hashová tabuľka s otvoreným adresovaním je spočiatku efektívna, no pri veľkom počte prvkov (1M a viac) sa stáva najhoršou množinou, čo je spôsobené kolíziami – príliš dlho hľadá voľné okienko pre vkladaný prvok.

Scenáre vkladania hodnôt v **náhodnom poradí**: čísla z intervalu 0 – 1000M s možným duplicitným výskytom (*1c*) a čísla z intervalu 1-N zamiešané v poli (*1d*):
Splay strom vykonáva veľké množstvo rotácií, keďže každý vkladaný prvok musí dostať na koreň stromu. Preto je neefektívny a čas sa zhoršuje s rastúcim počtom prvkov.
RedBlack strom sa vyvažuje a pri malom počte čísel je najefektívnejšou štruktúrou, s veľkým počtom prvkov (1M a viac) sa jeho efektivita zhoršuje.
Hashová tabuľka s reťazením je porovnateľná s **tabuľkou s otvorením adresovaním**, ale vo vzájomnom porovnaní je o čosi horšia, čo môže byť spôsobené zlou hashovacou funkciou a nedokonalou zmenou veľkosti.

Ľubomír Dlhý, ID: 102897

TESTOVANIE HĽADANIA PRVKOV

```
void testSearchAll(int caseType, double searchLastPercentage) {
    //printf("%d %lf\n",caseType,searchLastPercentage);
    testerSearch(500000, 10, caseType, searchLastPercentage);   // 500 000 testov, v kazdom         10 searchov,
    testerSearch(10000, 100, caseType, searchLastPercentage);   //  10 000 testov, v kazdom        100 searchov,
    testerSearch(5000, 500, caseType, searchLastPercentage);    //   5 000 testov, v kazdom        500 searchov,
    testerSearch(1000, 1000, caseType, searchLastPercentage);   //    1000 testov, v kazdom      1 000 searchov,
    testerSearch(1000, 5000, caseType, searchLastPercentage);   //    1000 testov, v kazdom      5 000 searchov,
    testerSearch(100, 10000, caseType, searchLastPercentage);   //     100 testov, v kazdom     10 000 searchov,
    testerSearch(10, 100000, caseType, searchLastPercentage);   //      10 testov, v kazdom    100 000 searchov,
    testerSearch(5, 1000000, caseType, searchLastPercentage);   //       5 testov, v kazdom  1 000 000 searchov,
    testerSearch(1, 10000000, caseType, searchLastPercentage);  //       1 testov, v kazdom 10 000 000 searchov,
}
```

Hlavná funkcia na vykonanie všetkých testov hľadania. Všetky štruktúry sú najprv naplnené rovnakými prvkami podľa scenáru a potom sú vykonávané hľadania podľa scenáru.

```
/* orderCase:
    1 - asc insert - asc search
    2 - asc insert - desc search
    3 - desc insert - asc search
    4 - desc insert - desc search
    5 - asc insert - ZigZag search
    6 - random insert with duplicit values - random search
    7 - random insert without duplicit values - random search
    8 - random insert without duplicit values - search [searchLast]% last elements
```

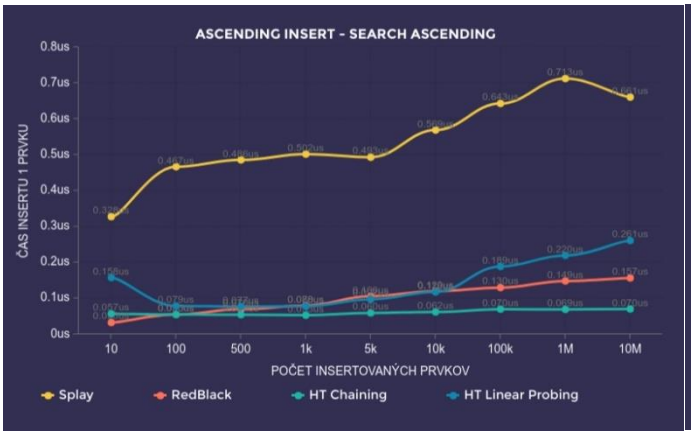
Scenáre vkladania prvkov a následného hľadania.

```
void testRbSearch(int iterations, int searches, rbt_tree_t* rb, int* pole) {

    int failed = 0;
    clock_t t1 = clock();
    for (int j = 0; j < iterations; j++) {
        for (int i = 0; i < searches; i++) {

            if (rbt_search(rb, pole[i]) == NULL)
                failed++;
        }
    }
    clock_t t2 = clock();
```

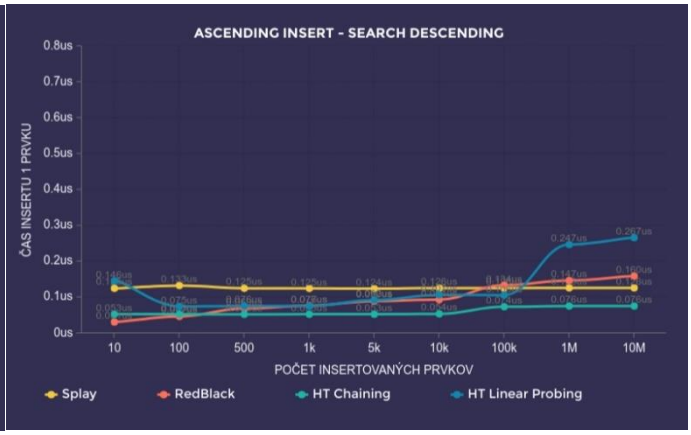
Priebeh testov v jednotlivých štruktúrach – cez argument prichádza naplnená štruktúra, počet testov, počet hľadaní a pole hľadaných prvkov. Odmeria sa čas, za ktorý zbehnú všetky testy, v každom teste sa hľadá daný počet prvkov z poľa. Nakoniec vyjadríme čas v mikrosekundách, za ktorý sa v priemere vyhľadá jeden prvok.

Ľubomír Dlhý, ID: 102897

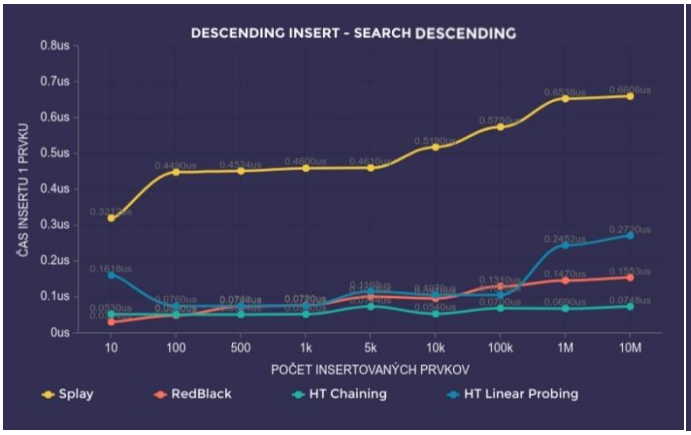
VÝSLEDKY TESTOVANIA HĽADANIA PRVKOV



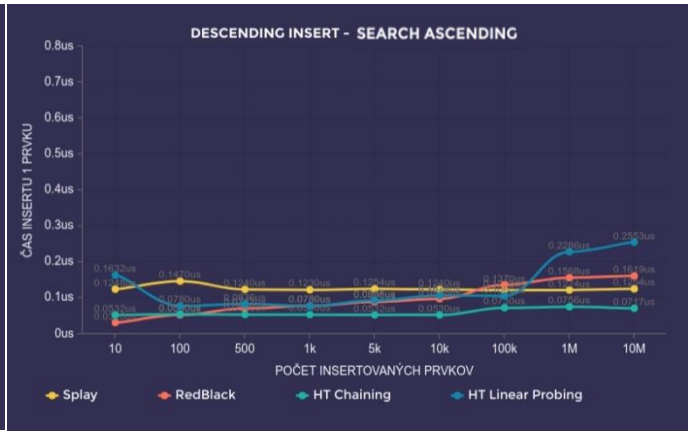
2a vzostupné vkladanie, vzostupné hľadanie



2b vzostupné vkladanie, zostupné hľadanie



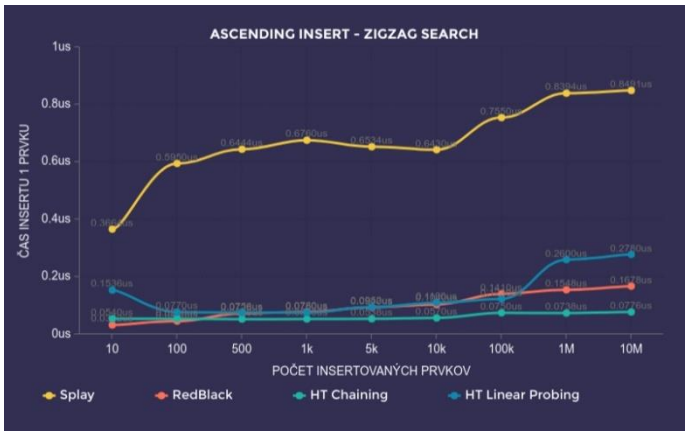
2c zostupné vkladanie, zostupné hľadanie



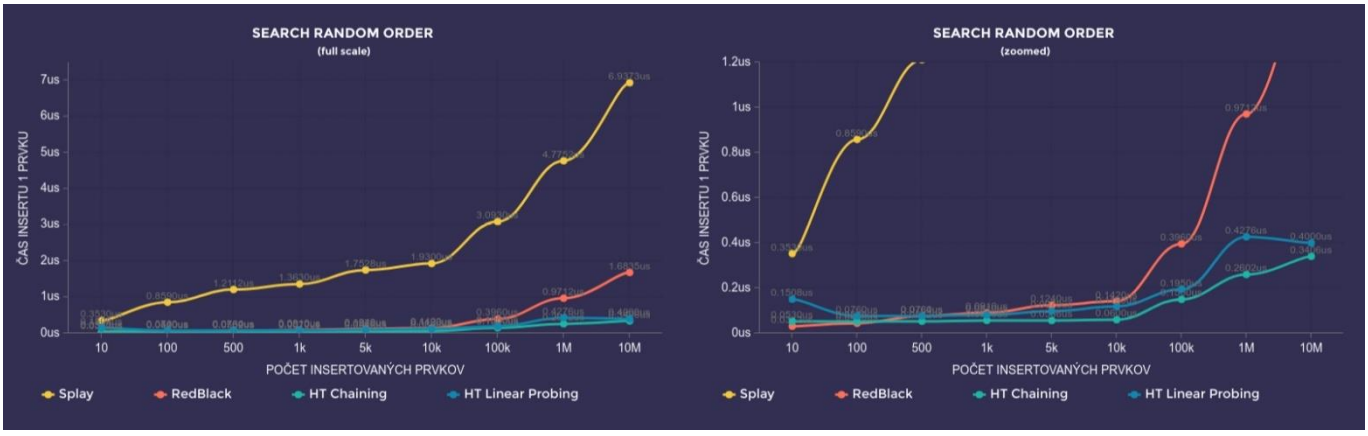
2d zostupné vkladanie, vzostupné hľadanie

Pri hashových tabuľkách a RedBlack strome sú vo všetkých scenároch podobné výsledky:
Tabuľka s reťazením je najefektívnejšia (vďaka riešeniu kolízií cez spájané zoznamy).
RedBlack a **tabuľka s otvoreným adresovaním** majú obdobné časy, ale pri veľkom počte prvkov sa začne prejavovať nevýhoda lineárneho probingu – nutnosť prehľadávať všetky indexy v poli až po prvé prázdne - a RedBlack strom začne byť efektívnejší.
Splay strom je pri opačnom poradí vkladania a vyhľadávania konštantný – prvok je hneď pod koreňom stromu a je nutné urobiť vždy iba jednu rotáciu. Pri rovnakom poradí vkladania a prehľadávania sú časy oveľa horšie a stane neefektívnou štruktúrou v dôsledku veľkého počtu rotácií.

Ľubomír Dlhý, ID: 102897



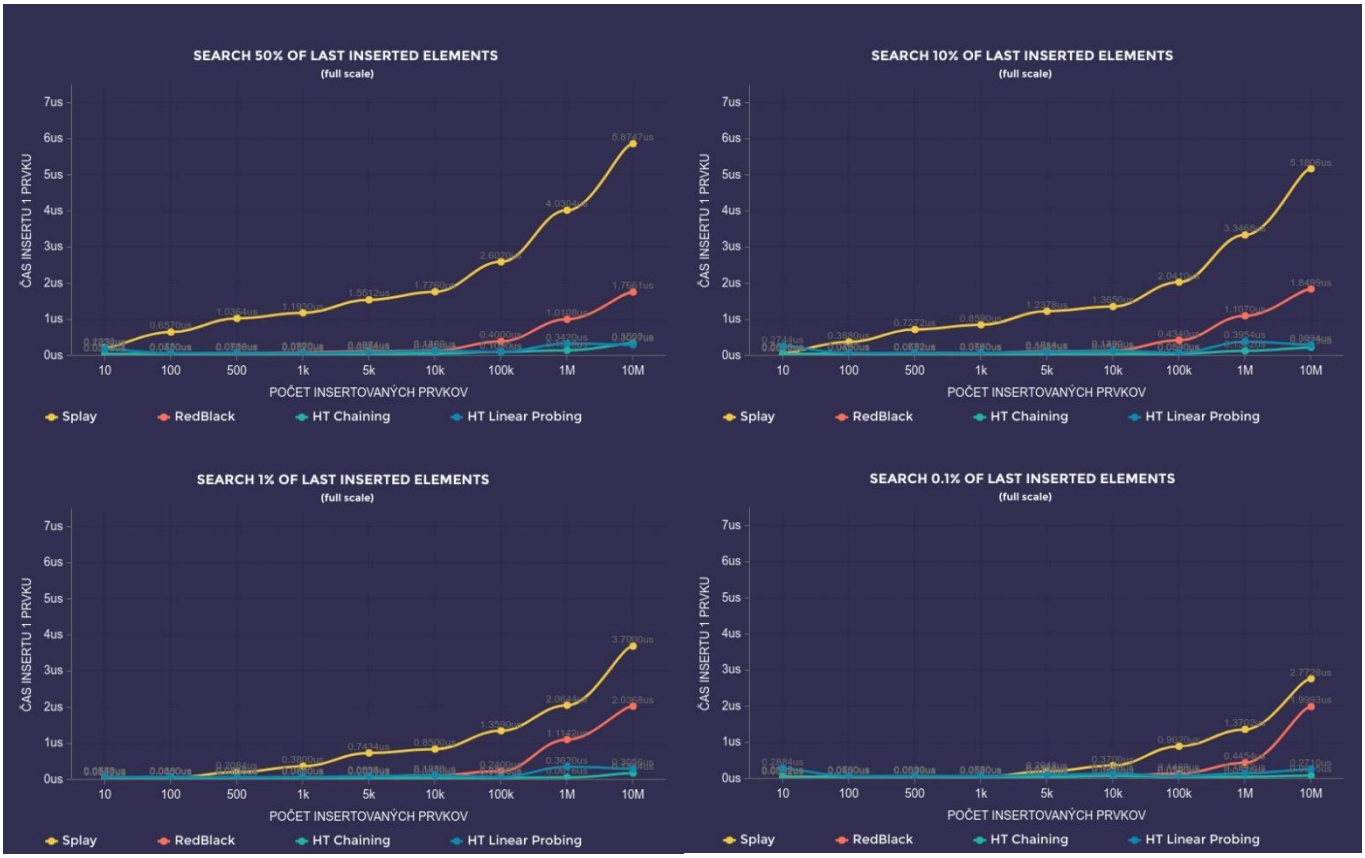
2e Vzostupné vkladanie + Zig Zag vyhľadávanie – vyhľadám najmenší prvok, potom najväčší, druhý najmenší, druhý najväčší..
Vidíme, že rozdiel oproti vzostupnému vyhľadávaniu je hlavne v časoch **Splay** stromu, ktorý musí v tomto type vyhľadávania spraviť viac rotácií.



2f náhodné vkladanie, náhodné hľadanie

Pri náhodnom vkladaní a náhodnom hľadaní prvkov sa významne zmení efektivita **Splay** stromu v dôsledku veľkého počtu rotácií pri nastavovaní hľadaného prvku na koreň stromu.
RedBlack strom je časovo porovnateľný s hashovými tabuľkami pri menšom počte prvkov, pri veľkom počte prvkov (100k+) začne byť rozdiel medzi tabuľkami a stromom značnejší.
Obidve tabuľky prejavujú ich konštantnú časovú efektivitu. V dôsledku kolízií a nutnosti prehľadania všetkých políčok až po prvé voľné je linear probing menej efektívny.

Ľubomír Dlhý, ID: 102897



2g náhodné vkladanie, náhodné hľadanie x% naposledy vkladaných hodnôt

Ak začneme vyhľadávať nedávno vkladané prvky, efektivita Splay stromu sa mimoriadne zvýši. Ostatné štruktúry majú približne rovnaké časy ako pri hľadaní všetkých hodnôt (*2f*).

Ľubomír Dlhý, ID: 102897

ZÁVER

Úlohou zadania bolo porovnať viacero implementácií dátových štruktúr z hľadiska efektivity operácií insert a search v rozličných situáciách. Ja som implementoval ***Splay strom*** a ***Hashovú tabuľku*** s riešením kolízií pomocou ***zreťazenia***. Prevzal som implementáciu ***Červeno-čierneho stromu*** a ***Hashovej tabuľky*** s ***otvoreným adresovaním*** (linear probing).
Všetky implementácie používajú ukladajú celočíselné dátove premenné a majú ošetrené duplicitné vkladanie.

Testovaním a porovnaním funkčných kódov som overil správnosť implementácií a zistil, v ktorých situáciách sú jednotlivé štruktúry vhodnejšie:

Splay strom – nevhodný pri vkladaní prvkov v zostupnom/vzostupnom poradí, pretože strom je nevyvážený. Efektívny pri hľadaní nedávno hľadaných/vkladaných prvkov - efektivita sa prejaví pri veľkom počte prvkov a malom počte nedávno hľadaných prvkov. V ostatných prípadoch je veľmi nevhodný kvôli veľkému počtu rotácií, keďže každý vkladaný/hľadaný prvok sa rotáciami nastaví na koreň stromu.

Red-Black strom – vhodný pri hľadaní aj vkladaní, pri menšom počte prvkov (menej ako milión) je časovo porovnateľný s hashovými tabuľkami. Pri väčšom počte prvkov (10M+) sa začne prejavovať logaritmická zložitosť (v porovnaní s tabuľkami). Je efektívnejší ako Splay strom vo väčšine prípadov.

Hashová tabuľka s reťazením – pri vkladaní je v niektorých prípadoch menej efektívna ako tabuľka s otvoreným adresovaním, čo môže byť spôsobené zlou hashovou funkciou a nedokonalým zväčšovaním tabuľky. Pri hľadaní je efektívnejšia ako tabuľka s otvoreným adresovaním vďaka rýchlejšiemu prechádzaniu kolízií v spájaných zoznamoch. Pri veľkom počte prvkov vychádza ako najefektívnejšia štruktúra.

Hashová tabuľka s otvoreným adresovaním (linear probing) – pri vkladaní je vo väčšine prípadov najefektívnejšia, čo je spôsobené dobrou hashovou funkciou a dobrým zväčšovaním tabuľky. Problém zaznamenávala pri vkladaní vo vzostupnom poradí pri veľkom počte prvkov, čo môže byť spôsobené veľkým počtom kolízií. V hľadaní je horšia ako tabuľka s reťazením v dôsledku kolízií a nutnosti lineárneho prehľadávania políčok tabuľky. Stále má však konštantnú časovú zložitosť.