Slovenská Technická Univerzita
Fakulta Informatiky a Informačných Technológií

Dátové Štruktúry a Algoritmy

Zadanie 2 - Vyhľadávanie v dynamických množinách

Marek Adamovič

Cvičenie: Pondelok 16:00 – 17:40
Ing. Peter Pištek, PhD.

Rok: 2019/2020

Použité algoritmy a riešenia

1. **Vlastná hash tabuľka s riešením kolízií pomocou reťazenia**

Pre správny chod tejto hash tabuľky potrebujeme najskôr zavolať funkciu vlastna_hash_init, ktorá nám vytvorí pole o danej veľkosti s vždy prvým prvkom spájaného zoznamu na indexe poľa.

```
6  NODE_HASH *vlastna_hash_init(int size){
7      NODE_HASH *hash_array = (NODE_HASH *)malloc(size * sizeof(NODE_HASH));
8      for(int i = 0; i < size; i++){
9          (&hash_array[i])->data = 0;
10         (&hash_array[i])->next = NULL;
11     }
12     return hash_array;
13 }
```

Inicializačná funkcia hash tabuľky

Funkcia nám vráti začiatok takejto tabuľky, ktorý následne posielame do ostatných funkcií. Jednou z ďalších funkcií je vlastna_hash_insert, ktorá slúži na pridanie prvku do hash tabuľky. Najskôr skontroluje naplnenie tabuľky a v prípade, že dosiahne určité naplnenie (0.55), tak sa rozšíri pomocou funkcie vlastna_hash_expand na dvojnásobnú veľkosť (+1;+3) ešte pred vložením prvku. Následne pošle kľúč (teda dáta, ktoré chcem do tabuľky uložiť) do funkcie vlastna_hash, ktorá jednoduchým modulom vráti pozíciu, na ktorú sa snažíme uložiť kľúč. Ak sa na pozícií nejaký prvok nachádza, tak vytvoríme nový uzol, ktorý následne napojíme na koniec spájaného zoznamu daného indexu.

```
1  typedef struct node_hash{
2      int data;
3      struct node_hash *next;
4  }NODE_HASH;
```

Štruktúra uzlu v tabuľke

Ak chceme zistiť, či sa ľubovoľný kladný prvok z množiny N+ nachádza v našej tabuľke, tak použijeme funkciu vlastna_hash_search, ktorá si vypočíta index, na ktorom by sa mal daný prvok nachádzať a následne prehľadáva spájaný zoznam daného indexu až dokým nenájde prvok, respektíve nepríde na koniec spájaného zoznamu.

Výhody takejto implementácie spočívajú v tom, že všetky prvky s rovnakým hashom máme pokope a nemiešajú sa nám s ostatnými prvkami, čo znamená aj vyššiu rýchlosť pri prehľadávaní prvkov s podobnými hashmi. Ako menšiu nevýhodu vnímam vyššie pamäťové nároky.

2. Prevzatá hash tabuľka s riešením kolízií pomocou otvoreného adresovania [1]

Pre správny chod tejto tabuľky musí užívateľ definovať SIZE, čo bude predstavovať maximálnu veľkosť tabuľky počas chodu programu (tabuľka nepodporuje zväčšovanie jej veľkosti počas behu programu). Ďalej pracuje s dvomi globálnymi premennými, z ktorých jedna je samotná hashová tabuľka (pole veľkosti SIZE) a druhá je „počítadlo“ prvkov v tabuľke, ktoré slúži len na ošetrenie prípadov prázdnej/plnej tabuľky.

Tabuľka má 2 hlavné funckie, Insert a Search, pomocou ktorých vieme do tabuľky pridávať a následne vyhľadať prvky. Ak pri vkladaní do tabuľky dôjde ku kolízii, posunieme sa s indexom o +1 a skúsime vložiť prvok tam. Ak aj tam nastane kolízia, tak postupne sa posúvame s indexom ďalej, až kým nenájdeme voľné miesto. Search prehľadáva od indexu, kde by sa mal nachádzať kľúč a pokračuje dokým prvok nenájde alebo dokým nenájde prázdny prvok (z čoho vychádza, že prvok sa v tabuľke nenachádza).

```
4   int arr[SIZE]={};
5   int count;
6
7   int CalculateHash(int key)
8   {
9       return key%SIZE;
10  }
```

Globálne premenné a hash funkcia

```
12  void Insert(int element)
13  {
14      if(count==SIZE){
15          printf("Error.\nTable if FULL\n");
16          exit(EXIT_FAILURE);
17      }
18      int probe=CalculateHash(element);
19      while(arr[probe]!=0 && arr[probe]!=-1)
20      {
21          probe=(probe+1)%SIZE;
22      }
23      arr[probe]=element;
24      count++;
25  }
```

Jednoduchá funkcia Insert, slúžiaca na vkladanie prvkov do tabuľky

3. Vlastný binárny vyhľadávací strom typu splay

Binárne vyhľadávacie stromy typu splay sa vyvažujú pomocou funkcie splay, ktorá využíva rotácie rodičov pridaného, čím tento prvok postupne posúva až na koreň stromu. Z tohto dôvodu si musíme do štruktúry uzlu pripraviť okrem ľavého a pravého potomka aj ukazovateľ na rodiča.

```
3  typedef struct node_strom{
4      int data;
5      struct node_strom*  parent;
6      struct node_strom*  left;
7      struct node_strom*  right;
8  } NODE_STROM;
9
```

Štruktúra uzlu v splay strome

Pridávanie prvkov do stromu riešime pomocou funkcie vlastny_insert, ktorá najskôr vytvorí uzol s hodnotou, ktorú sme do funkcie poslali, následne ju priradí do stromu podľa pravidiel klasického binárneho vyhľadávacieho stromu. Nakoniec tento uzol pošle do funkcie vlastny_splay, ktorá ho postupne „vyplaví na povrch“ až na miesto koreňa. Takto pridaný uzol potom funkcia vráti ako koreň.

```
53     if(future_root->parent->parent == NULL){       //zig, otec je koren
54         if(future_root->parent->left == future_root){      //future je lave dieta
55             rotation_right(future_root->parent);
56             return vlastny_splay(future_root);
57         }
58         if(future_root->parent->right == future_root){     //future je prave dieta
59             rotation_left(future_root->parent);
60             return vlastny_splay(future_root);
61         }
62     }
63
64     else if(future_root->parent->left == future_root && future_root->parent->parent->left == future_root->parent){
65         rotation_right(future_root->parent);
66         rotation_right(future_root->parent);
67         return vlastny_splay(future_root);
68     }
```

Funkcia vlastny_splay zisťuje, v akom vzťahu je pridaný prvok s jeho rodičom a starým rodičom

Vyhľadávanie prvkov sa nelíši od vyhľadávania v klasickom binárnom vyhľadávacom strome. Začneme v koreni a po porovnaní hodnoty postupne prechádzame na nižšie úrovne až dokým nenájdeme hľadaný prvok alebo neprídeme na koniec stromu. Tento typ stromu predpokladá, že užívateľ bude chcieť vyhľadávať prvky, ktoré vložil nedávno. Vďaka tomuto vynikajú splay stromy práve v takýchto scenároch.

4. Prevzatý binárny vyhľadávací strom typu AVL [2]

AVL stromy pracujú s tzv. balance factorom, ktorý vypočítavajú z výšky uzlov potomkov. Kvôli tomuto potrebujeme v štruktúre uzla zadefinovať výšku.

```
5   typedef struct node
6   {
7       int data;
8       struct node*  left;
9       struct node*  right;
10      int       height;
11  } node;
12
```

Štruktúra uzlu

Pri pridávaní prvkov do takéhoto typu stromu najskôr pridáme prvok klasicky podľa pravidiel binárnych vyhľadávacích stromov. Následne, počítame balance faktor podľa výšky potomkov a nakoniec pomocou balance faktoru jednotlivých uzlov určíme, ktorý uzol máme otáčať do ktorej strany (s jediným pravidlom, že na konci nemôže byť balance faktor v žiadnom uzle väčší ako 1).

```
189     else if( e < t->data )
190     {
191         t->left = insert( e, t->left );
192         if( height( t->left ) - height( t->right ) == 2 )
193             if( e < t->left->data )
194                 t = single_rotate_with_left( t );
195             else
196                 t = double_rotate_with_left( t );
197     }
```

Kontrolovanie balance faktoru vo funkcii insert a následná rotácia uzlu

Vyhľadávanie prvkov sa nelíši od vyhľadávania v klasickom binárnom vyhľadávacom strome. Začneme v koreni a po porovnaní hodnoty postupne prechádzame na nižšie úrovne až dokým nenájdeme hľadaný prvok alebo neprídeme na koniec stromu.

Testovanie

Snažíme sa zvoliť také testovacie scenáre, aby sme na každom vedeli opísať výhody/nevýhody jednotlivých metód, prečo sú v niektorých scenároch rýchlejšie, naopak, ktoré scenáre sú pre ne nevýhodné a na konci zhodnotiť, ktorá metóda sa v priemere držala najlepšie a ktorá zas najhoršie. V každom scenári robíme 10 meraní pre každú metódu (s tými istými číslami) a následne tieto merania priemerujeme, aby sme porovnávali priemerné hodnoty. Správnosť riešení kontrolujeme výpisom, keby sa niektorý z prvkov „stratil" a nenašiel alebo ak nájdem prvok, ktorý sa v ostatných nevyskytuje (výpisy treba zapnúť/vypnúť v jednotlivých implementáciach). Pred každým testom musíme skontrolovať statickú veľkosť SIZE, ktorú používa prevzatá hash tabuľka. Moja hash tabuľka začína v každom teste na veľkosti 10 007 a v prípade potreby sa zväčší. V žiadnom zo scenárov sa nevyskytujú duplikáty ani záporné čísla. Časy rozdeľujeme na čas insertu a čas searchu, aby sme vedeli lepšie porovnať výhody a nevýhody jednotlivých metód. V každom test case máme 2 sady testovaní, kde v prvom vyhľadávame všetky prvky, ktoré sme vložili a v druhom len jeden prvok X, väčší ako všetky pridané prvky vo väčšine testov, ktorý sa v žiadnej štruktúre nenachádza. X bolo vo všetkýc testoch určené ako číslo 1 500 001.

1. **Testovací scenár – 50 000 náhodných čísel z intervalu 1 – 1 000 000**

Tento scenár sa snaží simulovať reálnu situáciu, všetky čísla sú náhodné a zároveň všetky sa ich snažíme nájsť v danej štruktúre. Statickú veľkosť SIZE nastavíme na 60 000, takže na konci bude prevzatá hash tabuľka naplnená na približne 83%.

```
**********TEST_CASE_1**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.038800 sekundy
Insert: 0.028200 sekundy
Search: 0.010600 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.030900 sekundy
Insert: 0.022200 sekundy
Search: 0.008700 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.005100 sekundy
Insert: 0.004300 sekundy
Search: 0.000800 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.003700 sekundy
Insert: 0.001900 sekundy
Search: 0.001800 sekundy

**************************************
```

Vyhľadávanie existujúcich prvkov

```
**********TEST_CASE_1**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.031100 sekundy
Insert: 0.028900 sekundy
Search: 0.002200 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.026900 sekundy
Insert: 0.022700 sekundy
Search: 0.004200 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.004800 sekundy
Insert: 0.004600 sekundy
Search: 0.000200 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.004000 sekundy
Insert: 0.002000 sekundy
Search: 0.002000 sekundy

**************************************
```

Vyhľadávanie neexistujúcich prvkov

Z výsledku vidíme, že hash tabuľky mali oveľa lepšie časy, ako stromy. To z toho dôvodu, že pri náhodných číslach z takého veľkého intervalu sú hodnoty hashov väčšinov jedinečné a tým pádom majú hash tabuľky priamy prístup k väčšine prvkov. Vyhľadávali sme prvky v takom poradí, v akom sme ich pridávali, čo je v neprospech splay stromov, ktoré profitujú z vyhľadávania relatívne nových prvkov v tabuľke.

2. Testovací scenár – 50 000 čísel idúcich za sebou z intervalu 1 – 50 000

Tento scenár je zameraný hlavne na ukázanie nevýhody splay stromu pri nenáhodných prvkoch. Keďže každé ďalšie pridávané číslo je väčšie, ako predchádzajúce, tak namiesto splay stromu nám vzniká akýsi „splay konár“. Následne pri vyhľadávaní prvých pridaných prvkov musíme prejsť celým stromom, aby sme sa dostali na koniec. Statickú veľkosť SIZE nastavíme na veľkosť 50 001, takže na konci bude prevzatá hash tabuľka skoro úplne plná.

```
**********TEST_CASE_2**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 6.587700 sekundy
Insert: 0.003600 sekundy
Search: 6.584100 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.021100 sekundy
Insert: 0.016600 sekundy
Search: 0.004500 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.002500 sekundy
Insert: 0.002100 sekundy
Search: 0.000400 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.000800 sekundy
Insert: 0.000400 sekundy
Search: 0.000400 sekundy

**************************************
```

Vyhľadávanie existujúcich prvkov

```
**********TEST_CASE_2**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.003500 sekundy
Insert: 0.003000 sekundy
Search: 0.000500 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.018300 sekundy
Insert: 0.013100 sekundy
Search: 0.005200 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.002200 sekundy
Insert: 0.001900 sekundy
Search: 0.000300 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.011400 sekundy
Insert: 0.000200 sekundy
Search: 0.011200 sekundy

**************************************
```

Vyhľadávanie neexistujúcich prvkov

Z výsledku vidíme, že pri takomto scenári sú splay stromy pri vyhľadávaní úplne nevýhodné, keďže dopadli skoro okolo 350x horšie ako AVL stromy. Najlepšie si viedla moja hash tabuľka, keďže sme scenár pripravili tak, aby ani jedenkrát nemusela riešiť kolíziu. Cudzia hash tabuľka stroskotala na vyhľadávaní neexistujúcich prvkov, keďže sme zvolili takú distribúciu čísel, pri ktorej neboli v hash tabuľke skoro žiadne voľné miesta. Pri vyhľadávaní neexistujúceho prvku splay strom ihneď na začiatku zistil, že nič nie je napravo od koreňa, tým pádom veľmi rýchlo zistil, že neexistujúci prvok sa v strome nenachádza.

3. Testovací scenár – 10 000 násobkov čísla 10 000 idúcich za sebou

Tretí testovací scenár sme postavili v neprospech hash tabuliek, keďže ako čísla sme zvolili násobky. Statickú veľkosť SIZE nastavíme na 20 000 (na konci bude plná na 50%). S touto veľkosťou a distribúciou čísel bude riešiť kolíziu pri každom druhom pridanom prvku. Keďže vlastná hash tabuľka má implementované zväčšenie a jej počiatočná veľkosť je prvočíslo 10 007, mala by skončiť s oveľa lepším časom.

```
**********TEST_CASE_3**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.250000 sekundy
Insert: 0.002700 sekundy
Search: 0.247300 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.003600 sekundy
Insert: 0.002900 sekundy
Search: 0.000700 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.000600 sekundy
Insert: 0.000600 sekundy
Search: 0.000000 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.360900 sekundy
Insert: 0.180600 sekundy
Search: 0.180300 sekundy

**************************************
```

Vyhľadávanie existujúcich prvkov

```
**********TEST_CASE_3**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.496900 sekundy
Insert: 0.002700 sekundy
Search: 0.494200 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.003300 sekundy
Insert: 0.002600 sekundy
Search: 0.000700 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.000600 sekundy
Insert: 0.000400 sekundy
Search: 0.000200 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.547600 sekundy
Insert: 0.184500 sekundy
Search: 0.363100 sekundy

****************************************
```

Vyhľadávanie neexistujúcich prvkov

Splay stromy dopadli podobne zle, ako v scenári 2, keďže aj v tomto scenári bol každý ďalší prvok väčší, ako ten predchádzajúci. Napriek tomu, že sme znovu mali „splay konár“ namiesto splay stromu, cudzia hash tabuľka dopadla horšie, keďže pri mnohých pridaniach riešila kolíziu. Vďaka kombinácií riešenia kolízií pomocou reťazenia, počiatočnou veľkosťou prvočísla a možnosti zväčšiť tabuľku, dopadla moja hash mnohokrát lepšie ako prevzatá. Všimnime si, že hash tabuľke s otvoreným adresovaním trvalo približne rovnako vkladanie a hľadanie (pri hľadaní existujúcich prvkov). To z dôvodu, že tak isto, ako sa posúvala pri riešení kolízií pri vkladaní sa musela posúvať aj pri hľadaní prvkov. Taktiež stojí za povšimnutie zdvojnásobenie vyhľadávacieho času pri splay strom počas vyhľadávania neexistujúceho prvku. To nastalo z dôvodu, že keď sme vyhľadávali všetky vložené prvky, tak čím „novší“ prvok sme hľadali, tým rýchlejšie sme ho našli. Avšak ak sme sa pokúšali nájsť neexistujúci prvok, ktorý by sa mal nachádzať v druhej polovici stromu, tak je to podobné, ako keby sme hľadali vždy jeden zo starších prvkov. Teraz nám nenastáva prípad ako v predchádzajúcom teste, že splay stromy majú rýchle vyhľadávanie neexistujúceho, keďže koreň je tentokrát väčší ako neexistujúci prvok, čiže sa musím vnárať hlbšie do stromu, kde mnoho prvkov je taktiež väčších ako neexistujúci.

4. Testovací scenár – 1 000 000 náhodných čísel z intervalu 1 – 1 500 000

V tomto testovacom scenári nebudeme vyhľadávať všetky vložené prvky, ale len druhú pridanú polovicu. Taktiež keď uvážime, že všetky čísla budú náhodné, môžeme predpokladať, že splay strom dopadne s lepším výsledkom ako AVL strom. Statickú veľkosť SIZE nastavíme na 1 400 000, čiže na konci programu bude hash zaplnená približne na 71%.

```
**********TEST_CASE_4**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 1.138800 sekundy
Insert: 0.946000 sekundy
Search: 0.192800 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 1.141000 sekundy
Insert: 0.877200 sekundy
Search: 0.263800 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.135400 sekundy
Insert: 0.120300 sekundy
Search: 0.015100 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 49.762300 sekundy
Insert: 24.753700 sekundy
Search: 25.008600 sekundy

**************************************
```

Vyhľadávanie existujúcich prvkov

```
**********TEST_CASE_4**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.952300 sekundy
Insert: 0.889400 sekundy
Search: 0.062900 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.939700 sekundy
Insert: 0.830400 sekundy
Search: 0.109300 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.125000 sekundy
Insert: 0.117800 sekundy
Search: 0.007200 sekundy
```

Vyhľadávanie neexistujúcich prvkov

Vidíme, že v tomto prípade dopadol AVL o trochu horšie ako splay strom. Pri vyhľadávaní je tento rozdiel už výraznejší. Prekvapivo, hash tabuľka s otvoreným adresovaním dopadla katastrofálne (v teste s neexistujúcim prvkom som po 60 minútach prerušil test), a to napriek tomu, že jej maximálne naplnenie bolo 71%. Tu vidno masívny rozdiel medzi prechádzaní poľom pri otvorenom adresovaní a pri reťazení. Je pravdepodobné, že tabuľka, ktorá podporuje seba-zväčšenie taktiež riešila menej kolízií, keďže začínala na veľkosti 10 007 a postupne sa zväčšovala.

5. Testovací scenár – 10 000 náhodných čísel z intervalu 1 – 1 000 000

V poslednom testovacom scenári budeme taktiež hľadať len druhú polovicu vložených vstupov. SIZE sme nastavili na 10 001, čo znamená skoro úplné naplnenie prevzatej hash tabuľky. Cieľom tohto scenáru je presvedčiť sa, či metódy fungujú podobne aj pri menších vstupoch.

```
**********TEST_CASE_5**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.004500 sekundy
Insert: 0.004300 sekundy
Search: 0.000200 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.003500 sekundy
Insert: 0.002900 sekundy
Search: 0.000600 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.000600 sekundy
Insert: 0.000500 sekundy
Search: 0.000100 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.011900 sekundy
Insert: 0.007000 sekundy
Search: 0.004900 sekundy

****************************************
```

Vyhľadávanie existujúcich prvkov

```
**********TEST_CASE_5**********

Priemerny cas potrebny pre moj_strom (splay):
Celkovy: 0.005200 sekundy
Insert: 0.004900 sekundy
Search: 0.000300 sekundy

Priemerny cas potrebny pre cudzi_strom (AVL):
Celkovy: 0.004800 sekundy
Insert: 0.004400 sekundy
Search: 0.000400 sekundy

Priemerny cas potrebny pre moja_hash (retazenie):
Celkovy: 0.001000 sekundy
Insert: 0.000900 sekundy
Search: 0.000100 sekundy

Priemerny cas potrebny pre cudzia_hash (otvorene adresovanie):
Celkovy: 0.535300 sekundy
Insert: 0.003800 sekundy
Search: 0.531500 sekundy

****************************************
```

Vyhľadávanie neexistujúcich prvkov

Prevzatá hash tabuľka dopadla najhoršie v oboch prípadoch kvôli počtu kolízií, čo musela riešiť. O poznanie lepšie dopadli stromy. Vyhľadávanie v splay strome prebehlo rýchlejšie ako v AVL strome a to z dôvodu prehľadávania len druhej polovice prvkov. Hash tabuľka používajúca zreťazenie a seba-zväčšenie mala najlepší čas.

Zhodnotenie

Každá z metód sa preukázala byť v niektorom kole testovania ako najlepšia. Z toho vyplýva, že ak poznáme vstupnú množinu prvkov, s ktorou plánujeme pracovať a taktiež operácie, ktoré budeme využívať viac, je lepšie tomu prispôsobiť aj dátovú štruktúru. Napríklad keď vieme, že budeme pracovať hlavne s prvkami, ktoré sme vkladali len nedávno, zvolíme splay strom. Ak budeme pracovať s násobkami čísel, nebolo by múdre vyberať hash tabuľku, tak vyberieme AVL strom. Alebo ak máme postupne sa zväčšujúcu množinu, určite si nevyberieme splay strom. Rýchlosť teda značne ovplyvňuje množina prvkov, s ktorou pracujeme. Ak však nevieme, aké prvky dostaneme, resp. čo s nimi budeme robiť, mali by sme zvoliť najuniverzálnejšiu z metód, teda tú, ktorá mala dobrý celkový priemer časov a v žiadnom prípade nebola úplne pozadu za ostatnými. Na základe našich testovacích scenárov zisťujeme, že za takúto univerzálnejšiu metódu môžeme brať hash tabuľku s riešením kolízií pomocou reťazenia a implementovaným automatickým zväčšovaním podľa naplnenie a hneď za ňou AVL stromy, ktoré mali tiež veľmi dobré priemerné časy.

Zoznam bibliografických odkazov

1. Prabakaran.A, 2014, Hashing - Linear Probing (Open addressing), https://github.com/prabaprakash/Data-Structures-and-Algorithms-Programs/blob/master/Hashing%20-%20Linear%20Probing%20(Open%20addressing).c
2. C AVL Tree, https://www.zentut.com/c-tutorial/c-avl-tree/