Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológii
4. apríl 2021

Dátové štruktúry a algoritmy

Dokumentácia k zadaniu č. 2

Roland Vdovják
xvdovjak@stuba.sk
ID: 110912

Obsah

Roland Vdovják
ID: 110912

1 Zadanie

Existuje veľké množstvo algoritmov, určených na efektívne vyhľadávanie prvkov v dynamických množinách: binárne vyhľadávacie stromy, viaceré prístupy k ich vyvažovaniu, hašovanie 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.

Vašou úlohou v rámci tohto zadania je porovnať viacero implementácií dátových štruktúr z hľadiska efektivity operácií **insert** a **search** v rozličných situáciách (operáciu **delete** nemusíte implementovať):

• (2 body) Vlastnú implementáciu binárneho vyhľadávacieho stromu (BVS) s ľubovoľným algoritmom na vyvažovanie, napr. AVL, Červeno-Čierne stromy, (2,3) stromy, (2,3,4) stromy, Splay stromy, ...

• (1 bod) Prevzatú (nie vlastnú!) implementáciu BVS s iným algoritmom na vyvažovanie ako v predchádzajúcom bode. Zdroj musí byť uvedený.

• (2 bod) Vlastnú implementáciu hašovania s riešením kolízií podľa vlastného výberu. Treba implementovať aj operáciu zväčšenia hašovacej tabuľky.

• (1 bod) Prevzatú (nie vlastnú!) implementáciu hašovania s riešením kolízií iným spôsobom ako v predchádzajúcom bode. Zdroj musí byť uvedený.

Za implementácie podľa vyššie uvedených bodov môžete získať 6 bodov. Každú implementáciu odovzdáte v jednom samostatnom zdrojovom súbore (v prípade, že chcete odovzdať všetky štyri, tak odovzdáte ich v štyroch súboroch). Vo vlastných implementáciách nie je možné prevziať cudzí zdrojový kód. **Pre úspešné odovzdanie musíte zrealizovať aspoň dve z vyššie uvedených implementácií** – môžete teda napr. len prevziať existujúce (spolu 2 body), alebo môžete aj prevziať existujúce aj implementovať vlastné (spolu 6 bodov). Správnosť overte testovaním-porovnaním s inými implementáciami.

V technickej dokumentácii je vašou úlohou zdokumentovať vlastné aj prevzaté implementácie a uviesť podrobné scenáre testovania, na základe ktorých ste zistili, v akých situáciách sú ktoré z týchto implementácií efektívnejšie. **Vyžaduje to tiež odovzdanie programu**, ktorý slúži na testovanie a odmeranie efektívnosti týchto implementácií ako jedného samostatného zdrojového súboru (obsahuje funkciu **main**). **Bez testovacieho programu, a teda bez úspešného porovnania aspoň dvoch implementácií bude riešenie hodnotené 0 bodmi**. Za dokumentáciu, testovanie a dosiahnuté výsledky (identifikované vhodné situácie) môžete získať najviac 4 body. Hodnotí sa kvalita spracovania.

Môžete celkovo získať 10 bodov, **minimálna požiadavka sú 4 body**.

Riešenie zadania sa odovzdáva do miesta odovzdania v AIS do stanoveného termínu (oneskorené odovzdanie je prípustné len vo vážnych prípadoch, ako napr. choroba, o možnosti odovzdať zadanie oneskorene rozhodne cvičiaci, príp. aj o bodovej penalizácii). Odovzdáva sa jeden **zip** archív, ktorý obsahuje jednotlivé zdrojové súbory s implementáciami a jeden súbor s dokumentáciou vo formáte **pdf**.

2 Teoretické východiská[1]

2.1 Binárne vyhľadávacie stromy

Pre správnu implementáciu algoritmov použitých pre binárne vyhľadávacie stromy musíme najskôr vysvetliť isté pojmy.

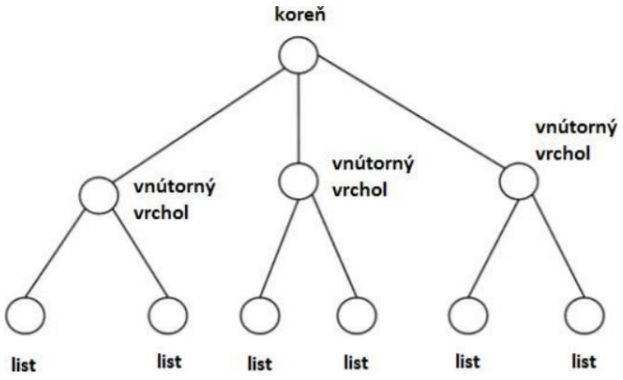
2.1.1 Binárne stromy

Strom je súvislý neorientovaný graf bez cyklov, množina vrcholov sa označuje písmenom *V* a množina hrán písmenom *G*. Tento graf je neorientovaný a súvislý.

Zakorenený strom je taký strom, ktorý má koreň (root).

List je koncový vrchol, ktorý nemá nasledovníkov.

Príklad stromu:



Hĺbka vrcholu predstavuje číslo, ktoré určuje počet hrán od koreňa stromu k danému vrcholu.

Výška vrcholu predstavuje číslo, ktoré určuje dĺžku cesty z daného vrcholu ku koncovému vrcholu, listu.

Binárny strom je strom, ktorý má najviac dvoch priamych nasledovníkov (potomkov). Potomkovia sa označujú *ľavý* a *pravý*.

(Kohútka, Abstraktné dátové typy, 2021, s. 39-46)

2.1.3 Binárny vyhľadávací strom

Binárny vyhľadávací strom (ďalej len BVS) je binárny strom, môže byť prázdny. Ak je neprázdny, tak platí, že každý prvok má kľúč, všetky kľúče sú rôzne. Všetky kľúče v ľavom podstrome sú menšie ako kľúč v koreni. Všetky kľúče v pravom podstrome sú väčšie ako kľúč v koreni. Ľavý aj pravý podstrom sú tiež BVS.

Pri práci s BVS využívame tieto operácie:

- Insert- pridať prvok do stromu
- Search- vyhľadať prvok v strome

(Kohútka, Binárne (vyhľadávacie) stromy, 2021, s. 21-22)

[1] Obsah tejto kapitoly je prispôsobený nášmu riešeniu, opisujeme len teoretické východiská, s ktorými sme pracovali

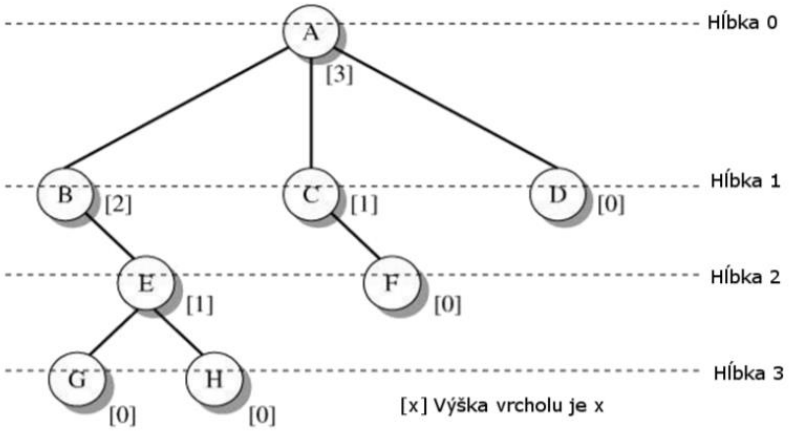
2.1.4 AVL stromy

BVS je AVL (Adeľson-Veľskij, Landis) strom práve vtedy, keď pre každý vrchol *x* v strome platí:

|(výška ľavého podstromu vrchola x) - (výška pravého podstromu vrchola x)| ≤ 1

Výška AVL stromu s n vrcholmi je O(log n).

Ukážka AVL stromu:



(Kohútka, Pokročilé algoritmy vyhľadávania, 2021, s. 11)

2.2 Hašovanie

Hašovanie je operácia, ktorá do hašovacej tabuľky priradí údaje na určitý index tabuľky podľa kľúča, ktorý je spracovaný hašovacou funkciou.

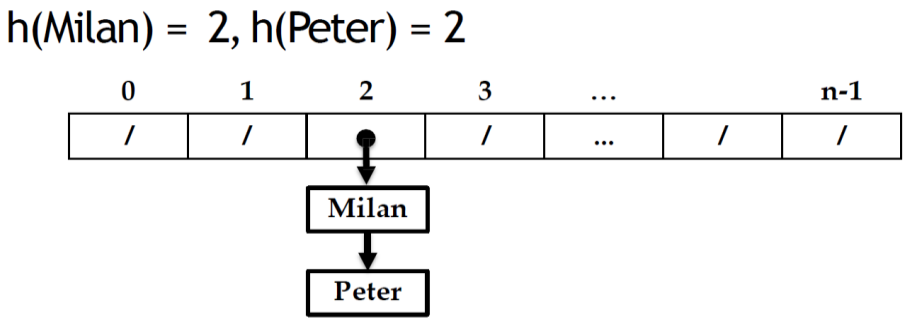
2.2.1 Riešenie kolízií

Kolízia je situácia, ku ktorej prichádza ak hašovacia funkcia priradí dvom kľúčom rovnaký index v tabuľke.

Reťazenie (chaining)

Reťazenie je spôsob riešenia kolízií. Funguje na spôsobe vedierok(bucketov), ktoré sú v indexe tabuľky. Do vedierok je možné uložiť viac kľúčov na základe dynamickej sekundárnej dátovej štruktúry.

Ukážka reťazenia:



(Kohútka, Hašovanie, 2021, s. 12)

3 Riešenie

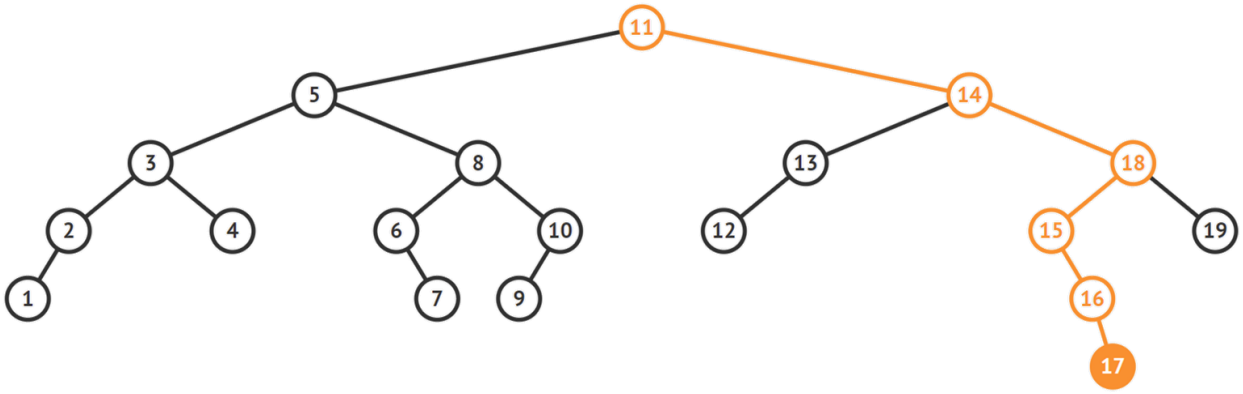
3.1 Binárne vyhľadávacie stromy

V našom riešení sme použili binárny vyhľadávací strom AVL.

3.1.1 Vkladanie (*insert*)

Funkcia *insert* vkladá nový vrchol do binárneho stromu. Funkcia funguje na rekurzívnom princípe, pomocou ktorého hľadá vhodné miesto na vloženie vrcholu. Dané miesto nájde porovnávaním kľúčov vkladaného vrcholu a vrcholov v binárnom strome.

Majme napríklad binárny strom uvedený nižšie. Chceme pridať vrchol s kľúčom *17*. Program prechádza od koreňa po oranžovo-zvýraznenej ceste. Vyberá si nasledujúci vrchol podľa toho, či je *17* väčší alebo menší od vrcholu, v ktorom sa program nachádza. Keď dorazí ku vrcholu, z ktorého nemôže pokračovať ďalej, vloží ho na správnu stranu. Ďalej sa rekurzívnym princípom „vracia“ po ceste, ktorou bol vrchol pridaný. Cestou kontroluje podmienky, aby bolo celý strom vyvážený. Pre prácu s vyvažovaním uchovávame aj informáciu o výške.



Ukážka časti kódu funkcie *insert*:

```
} else if (k < g->key){              // Vklada vlavo
            g->left = insert(k, g->left);
            if (height(g->left) - height(g->right) == 2)        // Vyvažovanie
                if (k < g->left->key) g = R(g);                 // ..
                else g = LR(g);                                 // ..

} else if (k > g->key){              // Vklada vpravo
            g->right = insert(k, g->right);
            if (height(g->right) - height(g->left) == 2)        // Vyvažovanie
                if (k > g->right->key) g = L(g);                // ..
                else g = RL(g);                                 // ..
}

g->height = bigger(height(g->left), height(g->right)) + 1;      // Vyska vrchola
```

3.1.2 Vyvažovanie

Tým, že typ stromu, ktorý sme si vybrali je výškovo vyvážený. Pri pridávaní jednotlivých vrcholov je nutné ošetriť situácie, v ktorých dochádza ku porušení podmienky na to, aby strom bol AVL. Ak sa strom stane nevyváženým, je nutné ho rotovať, aby sme docielili vyváženosť.

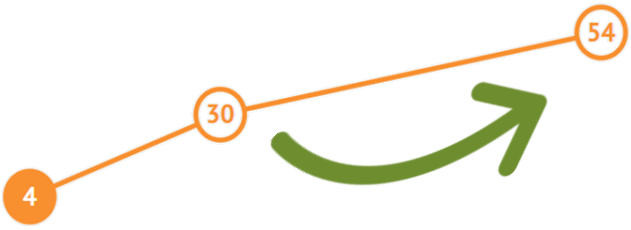
Nasledujúce situácie môžu nastať v ktoromkoľvek podstrome nášho stromu, preto sú tieto situácie zjednodušené, modelové.

Situácie, ktoré môžu nastať:

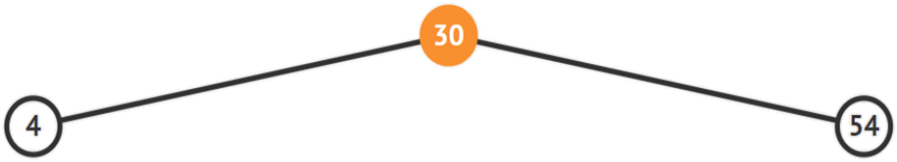
Potrebná rotácia vpravo

Strom je nevyvážený vľavo, výška ľavého podstromu kľúča *54* je rovná 2, výška pravého podstromu je rovná 0. Z týchto výsledkov vyplýva, že potrebujeme rotovať, aby sme strom vyvážili.
V nasledujúcom príklade vkladáme kľúč *4*, po rotácii vpravo sa kľúč *30* stane novým koreňom.

Rotácia vpravo:



Výsledný stav stromu:



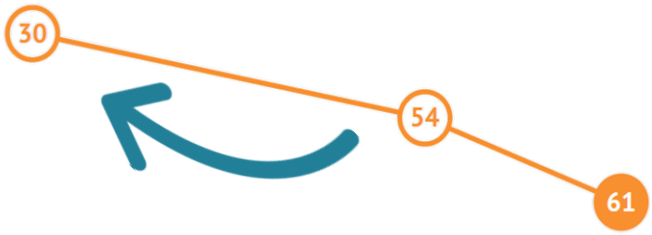
Ukážka kódu:

```
node* lchild = NULL;
lchild = par->left;     // lchild = lavy potomok; par = predchadzajuci vrchol
par->left = lchild->right;
lchild->right = par;
```

Potrebná rotácia vľavo

Strom je nevyvážený vpravo, výška ľavého podstromu kľúča *30* je rovná 0, výška pravého podstromu je rovná 2. Z týchto výsledkov vyplýva, že potrebujeme rotovať, aby sme strom vyvážili.
V nasledujúcom príklade vkladáme kľúč *61*, po rotácii vľavo sa kľúč *54* stane novým koreňom.

Rotácia vľavo:



Výsledný stav stromu:

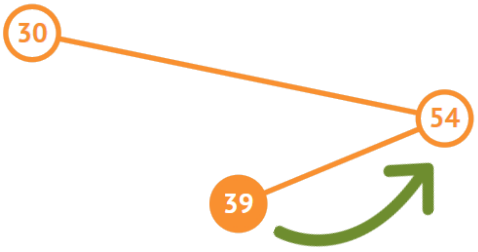
Ukážka kódu:

```
node* rchild = NULL;
rchild = par->right;                    // rchild = pravy potomok
par->right = rchild->left;
rchild->left = par;
```

Potrebná rotácia vpravo-vľavo

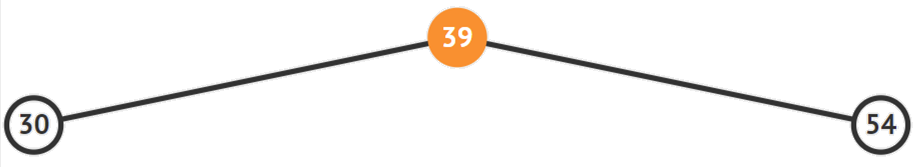
Strom je nevyvážený, výška ľavého podstromu kľúča *30* je rovná 0, výška pravého podstromu je rovná 2. Z týchto výsledkov vyplýva, že potrebujeme rotovať, aby sme strom vyvážili. V nasledujúcom príklade vkladáme kľúč *39*, po rotácii vľavo sa strom stane nevyvážený tak, ako v prípade rotácie vľavo, takže ho rotujeme vľavo.

Rotácia vpravo:



Rotácia vľavo:

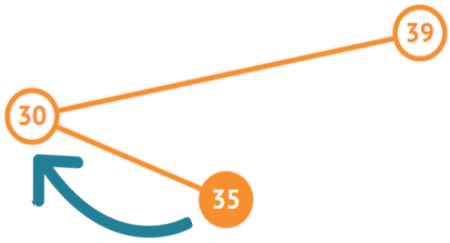
Výsledný stav stromu:



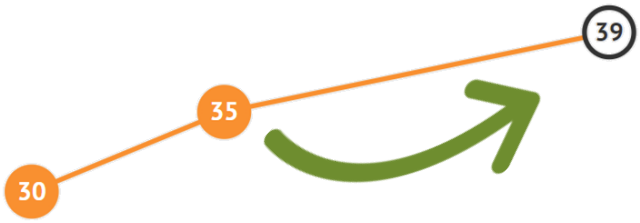
Potrebná rotácia vľavo-vpravo

Strom je nevyvážený, výška ľavého podstromu kľúča *30* je rovná 0, výška pravého podstromu je rovná 2. Z týchto výsledkov vyplýva, že potrebujeme rotovať, aby sme strom vyvážili. V nasledujúcom príklade vkladáme kľúč *39*, po rotácii vľavo sa strom stane nevyvážený tak, ako v prípade rotácie vľavo, takže ho rotujeme vľavo.

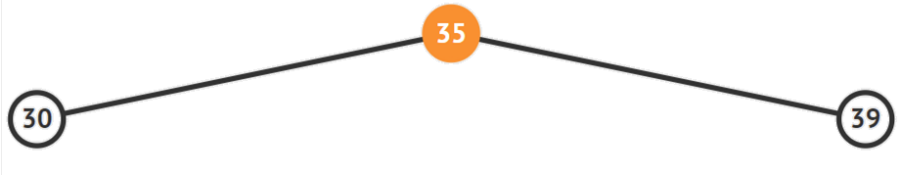
Rotácia vľavo:



Rotácia vpravo:



Výsledný stav stromu:



Tieto situácie sa zisťujú vo funkcií *insert* pomocou porovnávania hodnoty *node->key*, *(node->RightChild)->key* alebo *(node->LeftChild)->key*.

Ak je teda rozdiel v hodnote výšky podstromov, je nutné rotovať. To, ako sa rotuje sa určuje podľa toho, z akého vrcholu sa rekurzívne vracia. Ak ide z ľavej strany, je možné rotovať doprava alebo doľava doprava. Ak ide z pravej strany, rotuje sa doľava alebo doprava. Nie je potrebné rotovať na rovnakú stranu, z akej sa program rekurzívne vracia, resp. ako pokračuje v chode.

Ukážka kódu:

```
} else if (k < g->key){                          // Chod vlavo
            g->left = insert(k, g->left);
            if (height(g->left) - height(g->right) == 2)
                if (k < g->left->key) g = R(g);
                else g = LR(g);
} else if (k > g->key){                          // Chod vpravo
            g->right = insert(k, g->right);
            if (height(g->right) - height(g->left) == 2)
```

```
                if (k > g->right->key) g = L(g);
                else g = RL(g);
    }
```

3.1.3 Prehľadávanie (*Search*)

Funkcia *search* má nájsť hľadaný vrchol podľa kľúča, toto hľadanie sme realizovali rekurzívne. Program sa rozhoduje, či má nasledovať do ľavého alebo pravého nasledujúceho vrchola podľa hodnoty zadaného kľúča a kľúča vo vrchole, v ktorom sa nachádza.

Ukážka kódu:

```
if (g == NULL) return NULL;
if (k < g->key) return search(k, g->left);              // k = hladany kluc
else    if (k > g->key) return search(k, g->right);
        else return g;
```

3.1.4 Pomocné funkcie

Aby sme si boli istí, že funkcia *insert* funguje správne, naprogramovali sme funkciu *inorder*. Po spustení tejto funkcie by mali vložené kľúče len stúpať, v prípade, že by to bolo inak, funkcia *insert* nefunguje správne.

Ukážka kódu:

```
if (g != NULL) {
       inorder(g->left);
       printf("%5d", g->key);
       inorder(g->right);
}return;
```

3.2 Hašovanie

V našom riešení sme ako metódu riešenia kolízií použili reťazenie.

3.2.1 Vkladanie

Funkcia *Hinsert* má za úlohu vložiť kľúč a hodnotu na miesto v hašovacej tabuľke. Tabuľku, kľúč aj hodnotu má funkcia ako parametre.

Naša hašovacia tabuľka je pole spájaných zoznam jednotlivých *item*-ov, zároveň ich môžeme označiť za vedierka.

Ukážka kódu:

```
typedef struct ITEM {
       long int key;
       int val;
       struct ITEM* next;
} item;
```

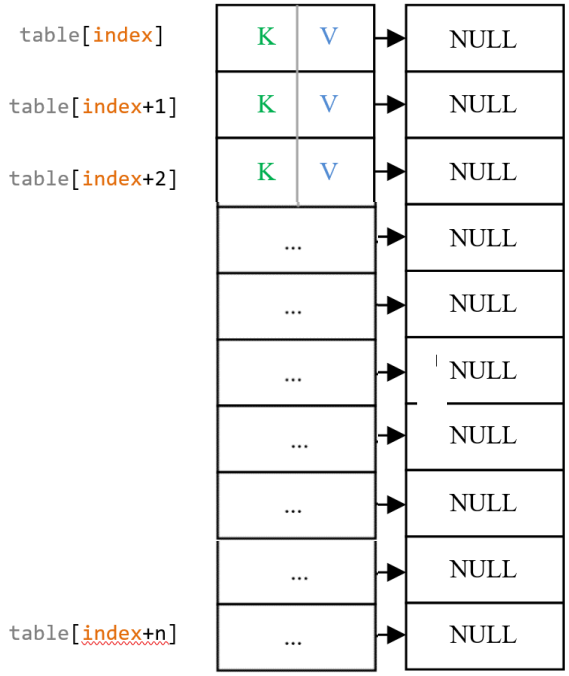
Pole smerníkov je vytvorené nasledovne:

```
item * table = (item*) malloc(size * sizeof(item));
item * tableitem = (item*) malloc(sizeof(item));

tableitem->val = -1;
tableitem->next = NULL;
tableitem->key = NULL;
```

```
for (int i = 0; i < size; i++) {
	*(table + i) = *tableitem;
}
```

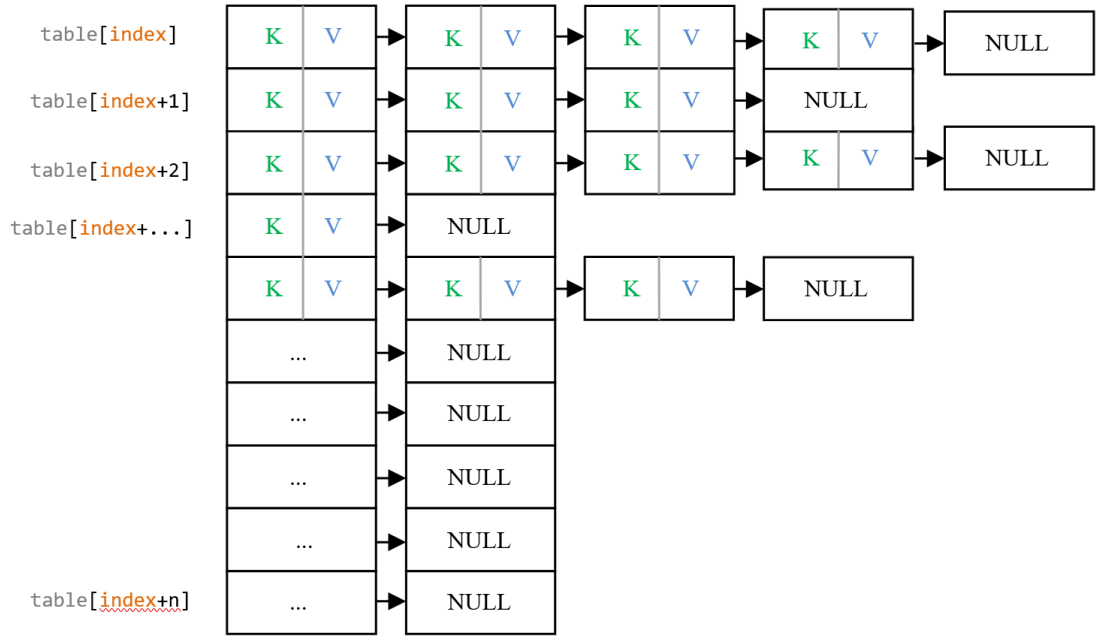
Vizuálne táto štruktúra vyzerá nasledovne:



3.2.2 Reťazenie

Reťazenie program využíva ak je dvom kľúčom priradený rovnaký index v tabuľke, tak v danom mieste v tabuľke, kde by sa mal vložiť kľúč a hodnota, sa vloží do štruktúry *item->next*. Tento proces sa opakuje vždy, keď sa nachádza už zaplnené miesto v tabuľke na mieste, do ktorého chceme vložiť nový *item*.

Po zaplnení štruktúra vyzerá nasledovne:



Ukážka kódu:

```
bool Hinsert(item* table, long val, long key) {
	if (table == NULL) return false;

	item* it = malloc(sizeof(item));
	it->key = key;
	it->val = val;
	it->next = NULL;

	long index = Hfunction(it->key);
	item* temp;
	temp = table + index;

	if (table[index].val != -1) {
		while (temp != NULL && temp->next != NULL) {
			temp = temp->next;
		} temp->next = it;
	}
	else {
		table[index] = *it;
		used++;
	}

	return true;
}
```

Funkcia je typu bool, aby sme vedeli, či vloží do tabuľky korektne. Po počiatočnej kontrole, či existuje tabuľka si program vytvorí nový *item* „*it*“, tomu vloží kľúč „*key*“ a hodnotu „*val*“, smerník na nasledujúci item pre istotu nastaví na *NULL*. Číslu *index* hašovacej funkcie priradí správnu hodnotu.

Hašovacia funkcia funguje na princípe: *KĽÚČ % VEĽKOSŤ_TABUĽKY*

Modulo nám vráti vždy hodnotu v rozpätí veľkosti tabuľky. Je pre nás dôležité mať vhodne veľkú tabuľku, t.j. prvočíslo. Nech nám hašovacia funkcia hašuje rovnomerne. Toto prvočíslo docielime vzorcom *(size / 6 + 1) * 6 + 1*, kde *size* je celé číslo veľkosti, ktorú chceme inicializovať. Použitím tohto vzorca nedocielime prvočíslo vždy, no často to bude prvočíslo trochu väčšie ako veľkosť, ktorú sme zadali, čo nám vyhovuje pri hľadaní, pretože kľúče budú rozmiestnené rovnomernejšie, akoby mala byť veľkosť ľubovoľné číslo.

Ukážka kódu:

```
	long int Hfunction(long int key) {
		return key % s;
	}
```

Ďalej vkladanie pokračuje podmienkou, či je v danom poličku poľa už nejaký item, ak áno, prechádza štruktúrou na koniec, kde priradí do smerníka na nasledujúci item vkladaný item.

3.2.3 Vyhľadávanie

Vyhľadávacia funkcia dostane ako parameter kľúč, podľa ktorého vráti hodnotu. Funguje tak, že daný kľúč upraví pomocou hašovacej funkcie, ktorá vráti index v poli. Keď funkcia nájde na danom mieste v poli jeden prvok, tak vráti jeho hodnotu. Akonáhle tam nie je len jeden, postupne prehľadáva item po item-e dokým sa kľúč item-u nezhoduje s kľúčom, ktorý hľadáme.

Ukážka kódu:

```
if (table[index].key == key) return table[index].val;
else while (table[index].key != key) {
                    bucket = bucket->next;
        } return table[index].val;
```

Vizuálne to vyzerá nasledovne (h = hodnota, k = kľúč):

Nech Hľadáme podľa kľúča 2513. Index sa vypočíta: *2513 % 13*, čo je *„4“*. Program ide postupne po šípkach, až sa kľúče rovnajú.

Table[index]

0	h: 14 (k: 1131)		
1	h: 11 (k: 9309)	- h: 18 (k: 8113	
2	h: 10 (k: 1822)		
3	h: 2 (k: 809)	- h: 4 (k: 8050)	- h: 16 (k: 1771)
4	h: 0 (k: 2539)	- h: 3 (k: 1876)	- h: 8 (k: 2513)
5	h: 5 (k: 6492)		
6	h: 15 (k: 1098)		
7	---		
8	h: 9 (k: 3115)		
9	---		
10	h: 13 (k: 4352)		
11	h: 7 (k: 661)	- h: 12 (k: 869)	- h: 17 (k: 1896)
12	h: 1 (k: 9814)	- h: 6 (k: 6460)	- h: 19 (k: 2105)

4 Testovanie

4.1 Binárne vyhľadávacie stromy

Testovali sme na množstve vrcholov sto až 25 miliónov rovnako pri vkladaní aj vyhľadávaní.

Cykly vyzerali nasledovne:

```
for (i = 0; i < max; i++) {             // Naplnenie
      j = ((j + 2) % max) + 1;
      g = insert(j, g);
}
for (i = 0; i < max; i++) {             // Prehľadanie
      j = ((j + 2) % max) + 1;
      t = search(j, g);
}
```

Operácia *„((j + 2) % max) + 1“* programu hovorí, aby išiel po násobkoch istého čísla, v tomto prípade násobky čísla 3.

4.2 Hašovanie

Testovali sme na množstve vrcholov sto až milión rovnako pri vkladaní aj vyhľadávaní. Aby sem docielili čo najobjektívnejšie testovanie, testovali sme rovnaké kľúče, tie sa vytvorili nasledovne.

```
unsigned long items = pow(10, 5);
unsigned long *keys = malloc(items * sizeof(long));
for (int i = 0; i < items; i++) {
        th = rand() % 10000;
        ht = rand() % 1500;
        num = th + ht * 1000 ;
        keys[i] = num;
}
```

Podfarbenou *power* funkciou ovplyvňujeme počet itemov. V cykle *for* sa vytorí náhodné číslo od 0 po 1 500 000.

Cykly vyzerali nasledovne:

```
for (int i = 0; i < items; i++) {
      k = keys[i];
      if (Hinsert(t, i, k));
}
for (int i = 0; i < items; i++){
      j = Hsearch(t, keys[i]);
}
```

5 Výsledky

5.1 Binárne vyhľadávacie stromy

Pre čo najstabilnejšie výsledky sme každú situáciu testovali rovnako päťkrát a potom urobili aritmetický priemer výsledkov.

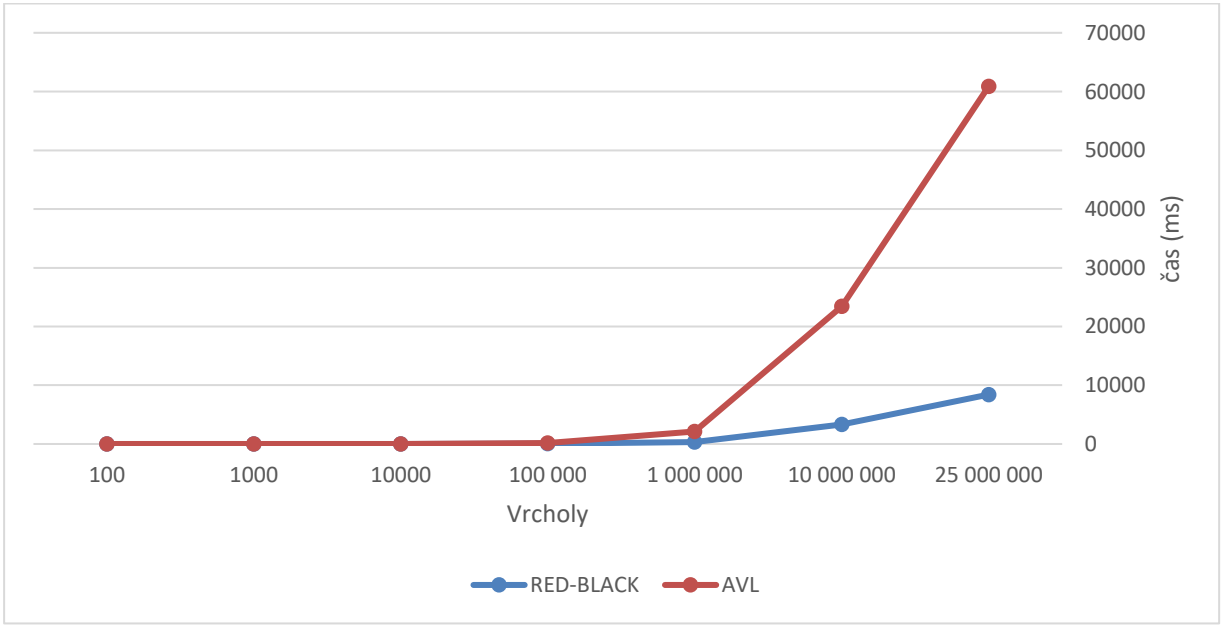
5.1.1 Vkladanie

Najväčšiu hodnotu sme stanovili 25 miliónov z dôvodu toho, že Visual Studio má obmedzenie na využívanie 2GB ram, pri väčšej hodnote ako 26 miliónov sme prekročili túto hranicu a program crashol. Oranžovo podfarbené sú bunky, pri ktorých je čas tak malý, že ho nevieme zistiť.

		25*10^6	10^7	10^6	10^5	10^4	10^3	10^2
RED-BLACK	1	8804	3256	346	32	3		
	2	8295	3291	313	30	2		
	3	8217	3289	312	31	3		
	4	8477	3271	315	55	3		
	5	8259	3407	316	30	3		
	AVG	**8410,4**	**3302,8**	**320,4**	**35,6**	**2,8**	**0**	**0**
AVL	1	52358	20136	1827	149	13	1	
	2	52528	20020	1822	150	12	1	
	3	52790	19998	1813	153	12	1	
	4	52185	20330	1832	147	12	2	
	5	52463	20163	1824	146	13	1	
	AVG	**52464,8**	**20129,4**	**1823,6**	**149**	**12,4**	**1,2**	**0**

Tabuľka 1

Výsledky sú reprezentované nasledovne:



Graf 1

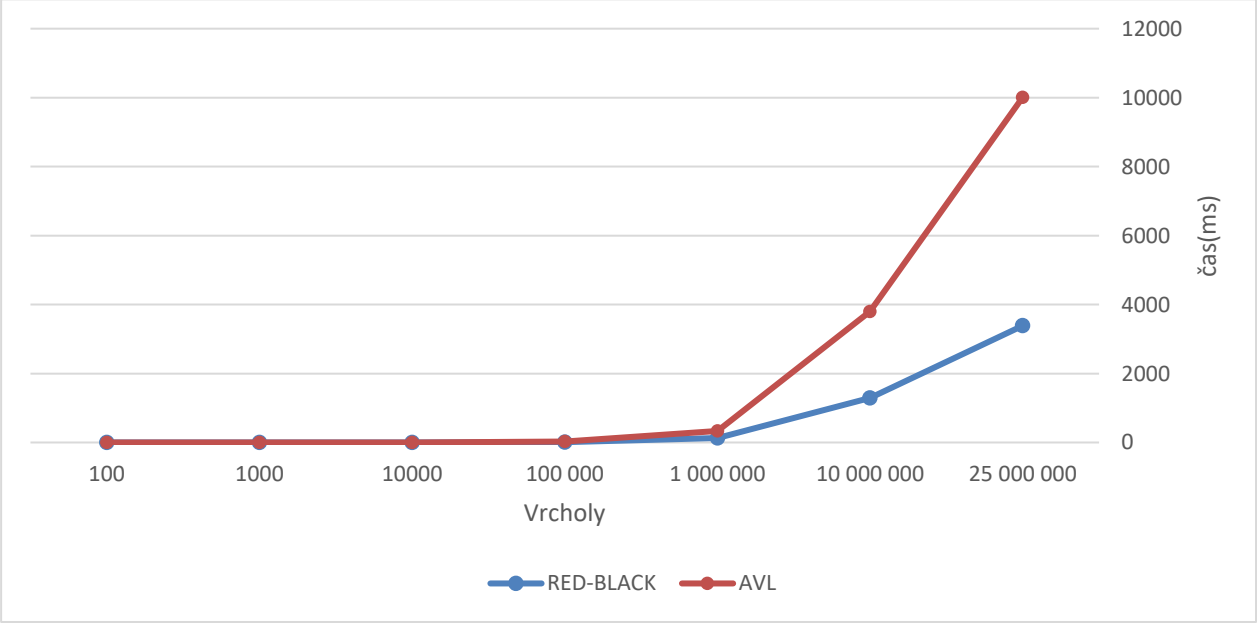
Výsledky sú očakávané, Red-Black stromy sú rýchlejšie vo vkladaní ako AVL. Predpokladáme, že AVL strom by mal mať zložitosť vkladania O(logN[2]), Red-Black tiež O(logN). Rozdiel nastáva v tom, ako často treba vyvažovať strom AVL, počet operácií rotácií pri vkladaní robí hlavný časový rozdiel. Je to spôsobené aj tým, že vkladané vrcholy majú hodnoty násobkov čísla tri. Preto program musí často vyvažovať. Pri inej vzorke to nemusí byť tak zjavné, na ukážku nevýhody resp. aj výhody jedného stromu oproti druhému sme to testovali práve takto.

5.1.2 Vyhľadávanie

		25*10^6	10^7	10^6	10^5	10^4	10^3	10^2
RED-BLACK	1	3479	1284	126	12	1		
	2	3361	1294	123	11	1		
	3	3370	1284	124	10	1		
	4	3361	1283	128	11	1		
	5	3359	1287	126	11	1		
	AVG	**3386**	**1286,4**	**125,4**	**11**	**1**	**0**	**0**
AVL	1	9948	3781	325	28	2	1	
	2	10277	3803	328	28	2	1	
	3	10054	3809	328	27	2	2	
	4	9924	3802	335	30	2	1	
	5	9839	3770	354	28	3	1	
	AVG	**10008,4**	**3793**	**334**	**28,2**	**2,2**	**1,2**	**0**

Tabuľka 2

Výsledky sú reprezentované nasledovne:



Graf 2

Priemerná časová zložitosť pei vyhľadávania pri AVL strome je O(logN), pre Red-Black je to O(logN). Vyhľadávanie v prevzatom strome bolo rýchlejšie, rozdiely neboli až také zásadné, ako pri vkladaní. Mohli byť spôsobené tým, že naše prehľadávanie funguje na rekurzívnom princípe

[2] N predstavuje počet vrcholov v binárnom strome

a prevzaté nie. Taktiež rýchlosť prevzatého stromu mohla byť ovplyvnená spôsobom vkladania, ktorý sme zvolili.

5.2 Hašovanie

5.2.1 Vkladanie

Pri hašovacej tabuľke sme testovali rýchlosť vkladania na vzorke stotisíc až sedemstotisíc kľúčov. Neskôr sme tieto údaje spracovali v tabuľke číslo 4, tá predstavuje hodnotu časovej náročnosti pre vloženie jedného kľúča, tejto tabuľke odpovedá graf číslo 3.

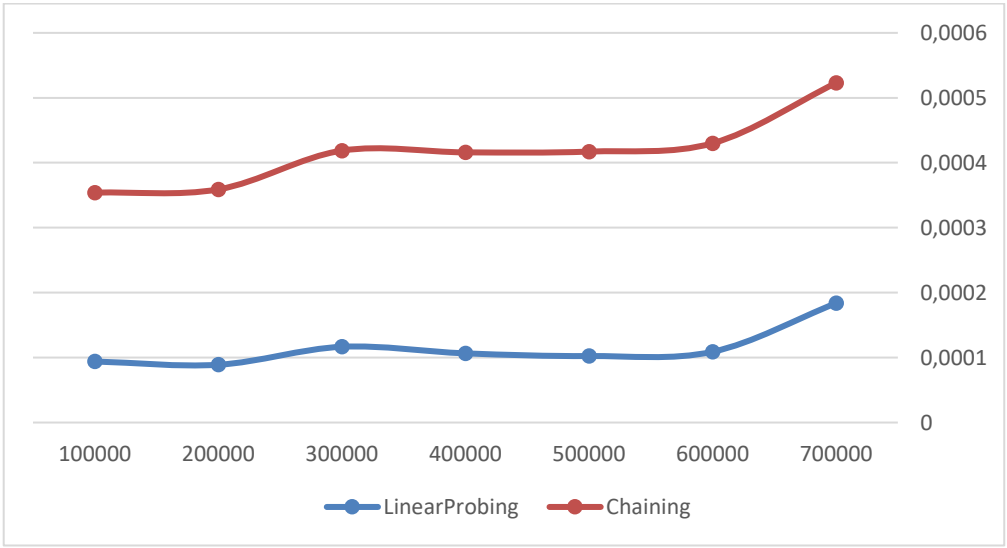
		700000	600000	500000	400000	300000	200000	100000
LINEAR PROBING	1	131	41	49	40	29	21	10
	2	144	70	58	43	34	15	7
	3	125	68	44	45	33	19	10
	4	134	73	53	42	41	14	8
	5	109	75	52	43	38	20	12
	AVG	**128,6**	**65,4**	**51,2**	**42,6**	**35**	**17,8**	**9,4**
CHAINING	1	190	159	138	106	78	53	26
	2	229	186	144	113	87	53	26
	3	266	211	166	134	91	52	26
	4	254	209	185	137	97	57	25
	5	248	198	154	129	100	55	27
	AVG	**237,4**	**192,6**	**157,4**	**123,8**	**90,6**	**54**	**26**

Tabuľka 3

	700000	600000	500000	400000	300000	200000	100000
Linear Probing	0,0001837	0,000109	0,0001024	0,0001065	0,0001167	0,000089	0,000094
Chaining	0,0003391	0,000321	0,0003148	0,0003095	0,000302	0,00027	0,00026

Tabuľka 4

Výsledky sú reprezentované nasledovne:



Graf 3

Vkladanie do hašovacej tabuľky má rozdielnu rýchlosť hlavne z dôvodu toho, že *Linear probing* prehľadáva tabuľku kým nenájde voľné miesto, toto môže trvať kratšie ako reťazenie, ak je vo vedierku viac kľúčov. Najhorší prípad časovej zložitosti pre obidva druhy riešenia kolízií je O(n),

5.2.2 Vyhľadávanie

Pri hašovacej tabuľke sme testovali rýchlosť vkladania na vzorke stotisíc až sedemstotisíc kľúčov. Neskôr sme tieto údaje spracovali v tabuľke číslo 6, tá predstavuje hodnotu časovej náročnosti pre vloženie jedného kľúča, tejto tabuľke odpovedá graf číslo 4.

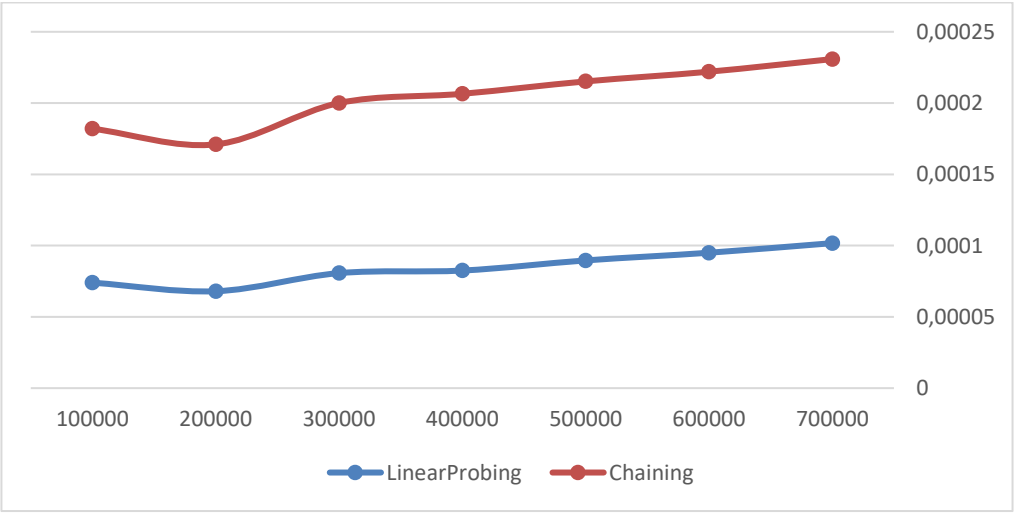
		700000	600000	500000	400000	300000	200000	100000
LINEAR PROBING	1	58	49	44	33	26	11	6
	2	73	64	52	35	24	13	9
	3	61	52	34	32	26	14	6
	4	85	57	45	33	22	15	8
	5	79	63	49	32	23	15	8
	AVG	**71,2**	**57**	**44,8**	**33**	**24,2**	**13,6**	**7,4**
CHAINING	1	81	58	66	42	28	17	10
	2	96	77	68	43	35	21	11
	3	94	89	66	58	39	22	11
	4	78	82	55	53	37	22	11
	5	103	75	59	52	40	21	11
	AVG	**90,4**	**76,2**	**62,8**	**49,6**	**35,8**	**20,6**	**10,8**

Tabuľka 5

	700000	600000	500000	400000	300000	200000	100000
Linear Probing	0,0001017	0,000095	0,0000896	0,0000825	0,0000807	0,000068	0,000074
Chaining	0,0001291	0,000127	0,0001256	0,000124	0,0001193	0,000103	0,000108

Tabuľka 6

Výsledky sú reprezentované nasledovne:



Graf 4

Rovnako, ako pri vkladaní, vyhľadávanie má priemernú časovú náročnosť O(1), výsledky pre prevzaté aj nami implementované riešenie boli veľmi podobné ale neboli ideálne. Väčšie vzorky ako 100 000 sme netestovali, pretože prevzatý program nedokázal naplniť tabuľku väčšiu ako 100 000.

Bibliografia

Kohútka, L. (3. Marec 2021). Abstraktné dátové typy. Dostupné na Internete: dokumentový server, is.stuba.sk, DSA_03

Kohútka, L. (10. Marec 2021). Binárne (vyhľadávacie) stromy. Dostupné na Internete: dokumentový server, is.stuba.sk, DSA_04

Kohútka, L. (24. marec 2021). Hašovanie. Dostupné na Internete: dokumentový server, is.stuba.sk, DSA_06

Kohútka, L. (20. marec 2021). Pokročilé algoritmy vyhľadávania. Dostupné na Internete: dokumentový server is.stuba.sk, DSA_05