**FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ**

**SLOVENSKÁ TECHNICKÁ UNIVERZITA**

Ilkovičova 2, 842 16 Bratislava 4

**2020/2021**

Dátové štruktúry a algoritmy

**Zadanie č.1**

**Cvičiaci: Mgr. Peter Lehoczký Vypracoval: Šimon Valíček**

**Čas cvičení: Štvrtok 13:30-14:40 AIS ID: 116320**

Šimon Valíček AIS ID: 105345

Obsah

**Binárny vyhľadávací strom....................................................................................................................3**

1. **Úvod........................................................................................................................................3**
2. **Implementácia.........................................................................................................................3**
   1. *Zadefinovanie uzlu............................................................................................................3*
   2. *Vkladanie..........................................................................................................................3*
   3. *Vyvažovanie......................................................................................................................4*
   4. *Vyhľadávanie....................................................................................................................5*
   5. *Odstránenie......................................................................................................................6*

**Hashovanie............................................................................................................................................7**

1. **Úvod........................................................................................................................................7**
2. **Implementácia – tabuľka 1.......................................................................................................7**
   1. *Hashovacia funckia...........................................................................................................7*
   2. *Zadefinovanie...................................................................................................................7*
   3. *Vkladanie..........................................................................................................................8*
   4. *Vyhľadávanie....................................................................................................................8*
   5. *Odstraňovanie..................................................................................................................9*
3. **Implementácia – tabuľka 2......................................................................................................10**
   1. *Hashovacia funkcia...........................................................................................................10*
   2. *Zadefinovanie...................................................................................................................10*
   3. *Vkladanie........................................................................................................................ .10*
   4. *Vyhľadávanie.................................................................................................................. .11*
   5. *Odstraňovanie................................................................................................................. 11*

**Testovanie........................................................................................................................................... 12**

**Binárny vyhľadávací strom**

1. **Úvod**

V binárnom strome sú údaje ukladané v uzloch. Moja implementácia je riešená s použitím

štruktúr, ktorá obsahuje údaje o kľúči, výške a smerníky na ďalší ľavý uzol, ktorého kľúč je

menší ako údaj uložený v aktuálnom a na ďalší pravý, v ktorom je uložený údaj väčší ako

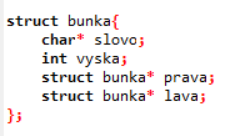
v aktuálnom uzle. V prípade, že takéto uzly neexistujú alebo ešte neboli pridané, smerník je

nastavený na NULL.

Prehľadávanie je následne v tom jednoduchšie, že nie je potrebné prehľadávať všetky uzly.

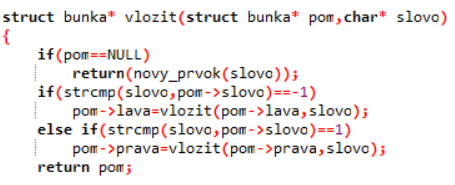
1. **Implementácia**
   1. **Zadefinovanie uzlu**

V úplnom úvode písania môjho projektu som si musel v prvom rade zadefinovať, čo je to uzol. Spravil som tak pomocou globálne premennej bunka, ktorá vyzerá nasledovne.



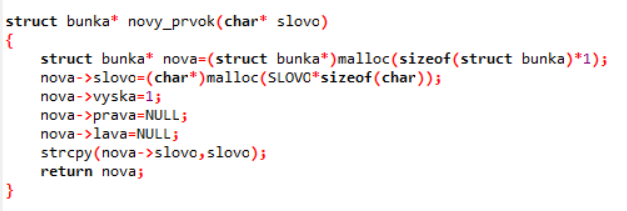
Jej obsahom je odkaz na reťazec slovo, ktorý bude slúžiť na uloženie reťazca do uzlu. Ďalej je tam premenná výška, ktorá bude predstavovať výšku daného uzla, ktorá slúži na vybalancovanie celého binárneho stromu uzlov. Táto výška sa bude meniť pridávaním a odstraňovaním prvkov. Keďže ide o binárny strom t.j. strom, ktorý sa vetví do dvoch smerov, je potrebné zadefinovať si odkaz na pravú a ľavú vetvu daného uzlu. Pri inšpirovaní sa pomocou internetu som v niektorých prípadoch našiel zaimplementovanie takzvaného rodiča, respektíve odkaz na uzol, ktorý je o 1 vyššie, ako náš daný uzol. Túto implementáciu som však nepoužil, nakoľko uzly pridávam pomocou rekurzie (viď nižšie), a tak nie je nutné si pamätať odkaz na predošlí uzol.

* 1. **Vkladanie**



Vkladanie prvkov do binárneho stromu je spravené pomocou rekurzie. Na začiatku

skontroluje, či uzol, kde sa má vložiť nový údaj je prázdny. Ak áno, znamená to, že uzol, ktorý sa chystáme vložiťdo funckie, treba najprv vytvoriť. Zavoláme funkciu novy\_prvok, ktorá nám vytvorí uzol a uloží doň zadaný reťazec.



Funkcia vlozit sa vtedy končí. Ak však uzol nie je prázdny tzn., že uzol je obsadený, porovná sa zadaný reťazec s reťazcom obsadeného uzla a z výsledku porovnávania vieme, či bude putovať nový uzol doľava alebo doprava. Na základe toho zavoláme rekurzívnu funkciu a priradíme pomocou nej tento uzol na príslušnú vetvu. Funkcia sa vykonáva rekurzívne, až kým nenájdeme danému uzlu príslušné miesto v strome t.j. prehľadávaním sa dostaneme na uzol s hodnotou NULL, kde opäť voláme funkciu na vytvorenie nového prvku (viď vyššie). V prípade, že uzol s príslušnou hodnotou už v strome existuje, nedôjde k zmene. Pri každom vnáraní hlbšie sa zároveň aj nastavuje nová výška stromu.

* 1. **Vyvažovanie**

Vyvažovanie je dôležitá súčasť binárneho stromu, ktorá zabezpečí rýchlosť chodu programu. V neoptimálnom prípade, by veľkosti jednotlivých vetiev stromu boli rozdielne dlhé, čo by malo za následok spomalenie prehľadávania. Ak zájdeme až do extrému, znamenalo by to, že v krajnom prípade by boli všetky prvky pridané na jednu a tú istú vetvu a dosiahnutie posledného prvku vetvy by malo za následok prejdenie všetkých predchádzajúcich. Aby sme sa tomu vyhli, snažíme sa držať všetky vetvy rovnako dlhé, aby sme tak optimalizovali a urýchlili vyhľadávací proces. Vyvažovanie prebieha pomocou koeficientu vyvažovania. Prehľadáva sa postupne strom od aktuálne pridaného smerom ku koreňu a hľadá sa prvý nevyvážený uzol. Keďže sa do stromu nové uzly pridávajú rekurzívne, program má priamo prístup k predchádzajúcim uzlom a nie je sú potrebné smerníky späť (ako som už spomínal vyššie).

Koeficient vyvažovania sa vypočíta ako odčítanie výšky pravého uzla podstromu od ľavého podstromu, pričom výšky sú uložené v každom podstrome. Ak je koeficient v intervale od 1 do -1 je strom vyvážený. V opačnom prípade je nutné strom vyvážiť rotáciami podľa štyroch scenárov.

Predstavme si, že máme tri uzly *a,b,c.*

Ak:

1. *b* je ľavým uzlom *a* a *c* je ľavým *b*, vtedy sa strom vyváži dvakrát rotáciou doľava, ktoré

sa dá vykonať raz rotáciou doprava.

2. *c* je pravým uzlom *b* a *b* je ľavým uzlom *c*, vtedy sa strom vyváži rotáciou doľava

a doprava.

3. *b* je pravým uzlom *a* a *c* je pravým uzlom *b*, vtedy sa strom vyváži dvakrát rotáciou

doprava, ktoré sa dá vykonať raz rotáciou doľava.

4. *b* je pravým uzlom *a* a *c* je ľavým uzlom *b,* vtedy sa strom vyváži rotáciou doprava

a potom doľava.

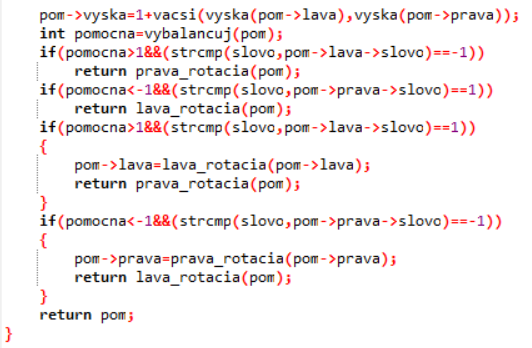
Všetky štyri rotácie sa dajú spraviť kombináciou doprava alebo doľava, preto je postačujúce

implementovať iba tieto dve funkcie a volať ich podľa potreby. Tiež je potrebné aktualizovať

výšku podľa toho, ktoré uzly a ako sa menia. Ak sa rotácie aplikujú aj na koreň stromu, je

potrebné zmeniť koreň, aby bolo možné pristupovať ku zmenenému stromu.

V mojom programe to vyzerá nasledovne:



Mohli sme si všimnúť, že moje „balancovanie“ zahŕňa volanie pomocných funkcii na vyýpočet výšky stromu, pomocou funkcie *vacsi*, ktorá vracia väčší prvok z dvoch a prída k nemu hodnotu 1. Dvomi prvkami myslíme hodnoty výšok uzlov podstromu (viď vyššie). Ďalej tu používam funkcie prava\_rotacia a lava\_rotacia, ktorých názov hovorí sám za seba. Nebudem ich tu podrobnejšie rozpisovať, nakoľko sa jedná čisto o teóriu z prednášok a sú dostupné v mojom kóde.

Poslednou pomocnou funkciou, ktorú voláme je funkcia vybalancuj, ktorá v prípade nenulového uzla vráti rozdiel hodnôt výšok pravého a ľavého podstromu.

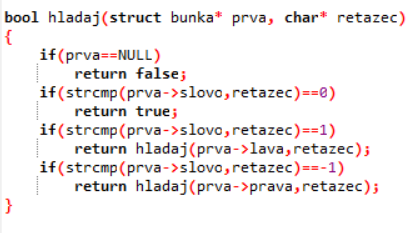
* 1. **Vyhľadávanie**

Prvky sa prehľadávajú ako v nevyváženom strome. Efektívnejší spôsob je iteratívne, keďže nie je potrebné mať prístup k predchádzajúcim uzlom. Strom program prehľadáva od koreňa

a posúva sa doprava alebo doľava podľa toho, či je kľúč väčší alebo menší ako kľúč práve

prehľadávaného uzla.

V praxi to vyzerá nasledovne:



Máme zadanú funkciu, ktorej rekurzívny chod zabezpečuje volanie samej seba podľa potreby tzn. porovná dva reťazce. Táto funkcia zabezpečuje postupné prehľadávanie po vetvách. Podľa výsledku porovnávania zadaného reťazcu sa posúva buď po ľavej vetve, alebo pravej vetve nášho stromu. V prípade, že narazí na prázdny uzol, vracia boolovskú hodnotu false, to znamená, že prehľadávaním sme sa dostali až na koniec daného uzlu a nenašli sme popritom zhodný reťazec v strome s tým, ktorý sme zadali. V opačnom prípade, ak sa porovnané reťazce zhodujú, vracia boolovskú hodnotu true.

Na výpis výsledku vyhľadávania nám slúži funkcia *vysledok\_hladania* so vstupným parametrom boolovskou hodnotou z funkcie *hladaj.* Ďalej ju nebudem rozpisovať, v prípade potreby sa dá pozrieť v kóde.

* 1. **Odstraňovanie**

Funkciu na odstraňovanie som vytvoril pomocou kombinácie častí funkcii na vyhľadávanie a na vkladanie prvok do stromu. Na začiatku rekurzívnym volaním funkcia prehľadáva vetvy stromu, ako to bolo pri vyhladávacej funkcii. Ak sa nájde prvok, ktorý chceme odstrániť, vetvíme to pomocou cyklu if do dvoch katégórií. Prvá je v prípade, že prvok je posledný na danej vetve, druhá v prípade, že na prvok nadväzujú zdola ďalšie prvky.

Následne prebieha balancovanie presne tak, ako to bolo vo funkcii na vkladanie tzn. pomocou pravých a ľavých rotácii a porovnávaním výšky uzlov.

**Hashovanie**

1. **Úvod**

Na ukladanie a vyhľadávanie údajov pomocou hashovania slúži hashovacia tabuľka. Dáta sú

v nej uložené pomocou poľa. Umiestnenie v poli určíme v hashovacej funkcii, ktorá pre každý

reťazec vypočíta algoritmom jedinečnú hodnotu a následným upravením získame index.

Časová zložitosť vyhľadávania pomocou hashovacej tabuľky v ideálnom prípade, kde nie sú

kolízie, by bola O(1), keďže ku pozícii sa dostaneme pomocou hashovacej funkcie. V prípade,

že nám však funkcia vráti rovnaký index a je potrebné riešiť kolízie, časová zložitosť je O(n),

kde n je veľkosť poľa. To by však nastalo iba vtedy, ak by pre všetky prvky vznikol rovnaký

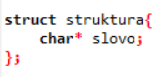
index, inak to bude zvyčajne menej.

V mojom projekte pracujem s dvoma rozdielnymi druhmi tabuliek:

1. Tabuľka je vytvorená staticky.
2. Tabuľka je vytvorená dynamicky ako pole prvkov.
3. **Implementácia – tabuľka 1**
   1. **Hashovacia funkcia**

Táto funkcia je pre obe tabuľky totožná. Zo vstupného reťazca postupným prehľadávaním získame súčet ASCII hôdnôt jednotlivých prvok. Tie následne vydelíme pomocou %MAX, aby tak nedošlo k prípadu, že hodnota reťazca je väčšia ako veľkosť tabuľky. Moja hashovacia funkcia je dostupná v kóde.

* 1. **Zadefinovanie**

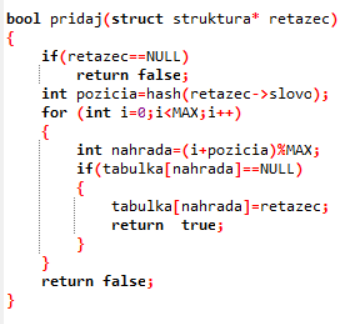


Jednotlivé prvky v tabuľke sú definované takto. Čiže obsahom prvku je iba reťazec znakov.



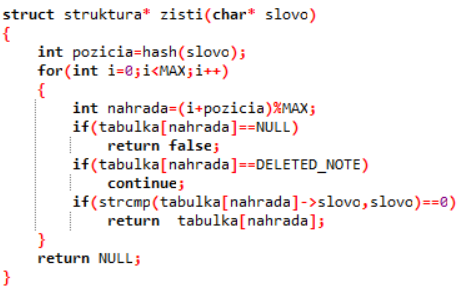
Tabuľka je definovaná ako statické pole prvkov s konštantou MAX, ktorú mením v hlavičke programu podľa potreby.

* 1. **Vkladanie**



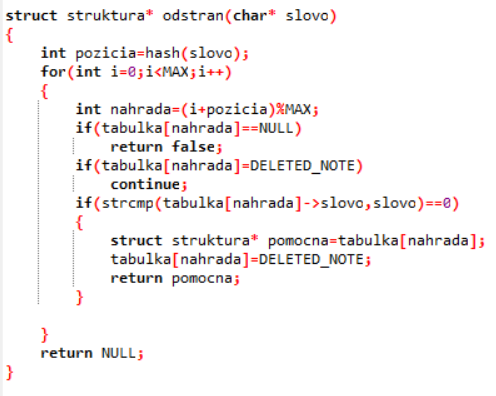
Vkladanie prvok prebieha nasledovne. Najprv zistíme, či reťazec, ktorý sme pridali nemá hodnotu NULL. V takom prípade vraciame funkciu false. V opačnom prípade voláme hashovaciu funkciu, ktorá nám zabezpečí vytvorenie tzv. kľúču, pomocou ktorého vieme v tabuľke určiť miesto, na ktoré sa reťazec vloží. Aby nám nedochádzalo ku kolízii, vytvoríme vo for cykle náhradnú pozíciu v tabuľke pre prípad, že by mali dva reťazce rovnakú hodnotu ASCII. V prípade úspešného zaradenia prvku sa nám vráti boolovská hodnota true.

* 1. **Vyhľadávanie**



Hľadanie prvku mi zabezpečuje funckia *zisti.* Jej vstupom je reťazec, ktorému je volaním hashovacej funkcie priradená pozícia v tabuľke. Nakoľko som použil pri vkladaní prvkov náhradnú pozíciu, aby nedošlo ku kolízii, treba ju použiť aj pri vyhľadávaní. Vo for cykle si ošetríme prípady, ktoré môžu nasťať. V prvom prípade, že na danom mieste v tabuľke nie je žiaden prvok, funkcia vráti hodnotu false. V prípade, že tam prvok bol, ale už nie je, vráti nám hodnotu DELETED\_NOTE, ktorú vysvetlím nižšie vo funkcii odstraňovania. V prípade, že sa prvom našiel, funkcia vracia ukazovateľ na daný prvok. V prípade, že sa prvok nenašiel vôbec, vracia funkcia nulový ukazovateľ.

* 1. **Odstraňovanie**



Vstupom je opäť reťazec, ktorý sa snažíme tentokrát odstrániť. Pomocou volania funkcie hash, zistíme jeho ASCII hodnotu. Následne vo for cykle použijeme už vyššie spomínanú „náhradu“ a začneme prehľadávať. Ak sa daná pozícia v tabuľke rovná NULL, vrátime hodnotu false. Ak narazíme na hodnotu DELETED\_NOTE, znamená to, že prvok na danom mieste už bol odstránený a tak pokračujeme ďalej. Ak sa však reťazec na danom mieste tabuľky zhoduje s reťazcom, ktorý sme zadali, nahradíme ho hodnotou DELETED\_NOTE, ktorá mi slúži ako pomocná hodnota pri výpise tabuľky, aby som si vedel overiť správnosť všetkých mojich funkcii. V prípade potreby však vieme v tomto kroku zadefinovať iné správanie, prerobením tejto vetvy if, napríklad môžeme prvok odstrániť a nahradiť ho prázdnym, aby sme tak uvoľnili miesto v tabuľke. Následne nám táto vetva if vracia ukazovateľ na dané miesto v tabuľke. V prípade neúspechu vracia funkcia nulový ukazovateľ.

1. **Implementácia – tabuľka 2**
   1. **Hashovacia funkcia**

Ako som už vyššie spomínal, táto funkcia je pre obe tabuľky totožná. Zo vstupného reťazca postupným prehľadávaním získame súčet ASCII hôdnôt jednotlivých prvok. Tie následne vydelíme pomocou %MAX, aby tak nedošlo k prípadu, že hodnota reťazca je väčšia ako veľkosť tabuľky. Moja hashovacia funkcia je dostupná v kóde.

* 1. **Zadefinovanie**

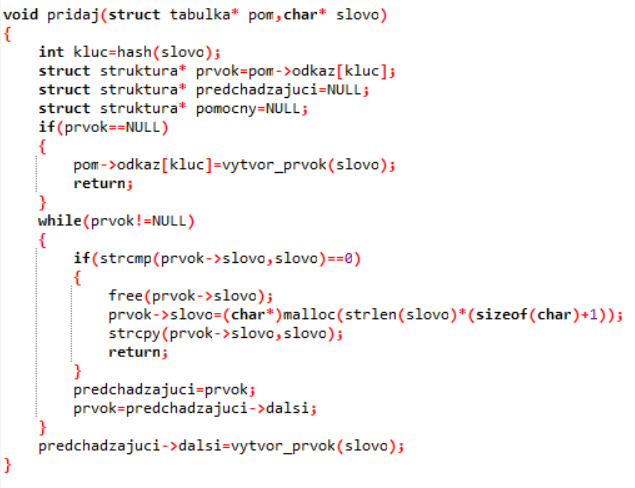
Obrázok, na ktorom je text

Automaticky generovaný popis

Jednotlivé prvky som si zadefinoval nasledovne. Prvou hodnotou prvku je reťazec znakov, druhá hodnota predstavuje odkaz na ďalľí prvok tabuľky. Z toho vyplýva, že aj tabuľka bude dynamické pole prvkov. Jej definícia vyzerá takto. Vzhľadom na to, že sa jedná o dynamické polia, vytvoril som si zvlášť funkcie na alkovoanie pamäte pre prvok a na alokovanie pamäte pre tabuľku. Ďalej to tu nebudem rozpisovať, je to dostupné v mojom kóde.

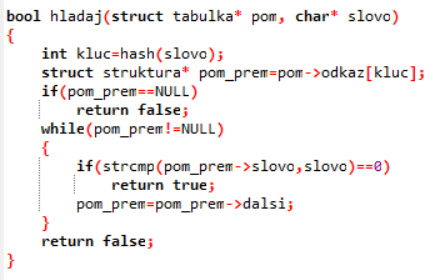


* 1. **Vkladanie**



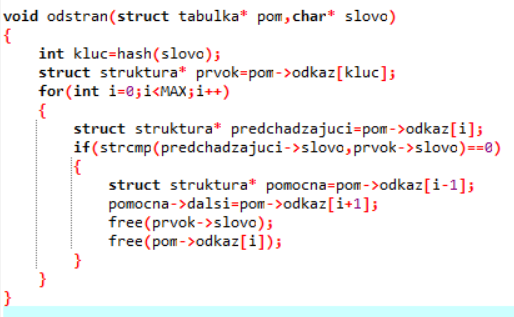
Funkcia na vkladanie vyzerá nasledovne. Najprv si pomocou funkcie hash zistíme pozíciu prvku v tabuľke. Následne vytvoríme pomocnú premennú, do ktorej uložíme hodnotu na danej pozícii. Ak je nulová, vytvoríme na danom mieste prvok. V opačnom prípade prehľadávame tabuľku a hľadáme zhodu. Ak sme zhodu našli, nahradíme prvok, v podstate tým istým prvkom a pokračujeme v prehľadávaní.

* 1. **Vyhľadávanie**

****

Vyhľadávanie zabezpečuje boolovská funkcia hladaj. Vstupným parametrom je tabuľka a reťazec, ktorý sa snažíme vyhľadať. Použijeme funkciu hash, aby sme zistili pozícia slova v tabuľke. Následne začneme prehľadávať pomocou pomocnej premennej. Ak sa rovná NULL, vraciame sa s neúspechom. Inak v cykle while prehľadávame jednotlivé prvky, až kým nenájdeme zhodu. Ak zhodu nakoniec nenájdeme, funkcia vracia hodnotu false. Na následný výpis výsledku hľadania je v programe funkcia, ktorá to špeciálne zabezpečuje. V dokumentácii som sa rozhodol nevenovať jej pozornosť.

* 1. **Odstraňovanie**



Vymazávanie prvkov mi zabezpečuje naledujúca funkcia. Vstupom je tabuľka a hľadaný reťazec v nej. Vytvorí sa pomocná premenná prvok, ktorá pomocou hashovacej funkcie zistí ak sa niektorý zhoduje s tým, ktorý som zadal. Uvoľní sa dané pole a napojí sa predchádzajúci odkaz na ten, ktorý nasleduje po odstránenom. Následne sa ukončí funkcia.

**Testovanie**

Všetky testy prebehli pre otpimálnosť na tej istej skupine náhodných prvkov. Jediný faktor, ktorý sa pri testovaní menil, bol ich počet. Výsledkom bolo zistenie, že tabuľky sú efektívnejšie resp. minimálne rýchlejšie, ako strom, aj keď nie o veľa. Výsledky testovania je možné vidieť nižšie. Boli porovnávaných 6 hodnôt a to: 10 000 slov, 50 000 slov, 100 000 slov, 500 000 slov, 1 000 000 slov a 5 000 000 slov.

Tento graf porovnáva rýchlosť jednotlivých funkcii naprieč AVL stromom.

Tento graf porovnáva rýchlosť jednotlivých funkcii naprieč prvou tabuľkou.

Tento graf porovnáva rýchlosť jednotlivých funkcii naprieč druhou tabuľkou.

Na záver je tu stĺpcový graf, ktorý porovnáva všetky 3 hodnoty vyššie

spomenutých grafov.