Sociálna sieť

Autor: Jozef Varga

Hlavná myšlienka

Zadanie je vypracované v jazyku C. V mojom riešení využijem hashovaciu tabuľku, v ktorej budem ukladať stránky, ktoré užívatelia like-li. Vďaka hashu je vyhľadávanie veľmi rýchle. Kolízie pri hashovaní riešim zreťazením. Každá stránka bude obsahovať binárny vyvažovací strom (AVL), ktorý bude ukladať užívateľov, ktorí like-ovali danú stránku. Zaradzovanie mien bude abecedné. Pri vyberaní k-teho užívateľa nebudem využívať inorder prehľadávanie, ale budem vo vrcholoch ukladať počet ľavých detí daného vrcholu. Tým sa to zrýchli, nakoľko inorder je pomalý (nemusím prehľadávať celý strom). Tým, že ukladám iba ľavé deti, v stránke si uložím celkový počet like-ov a tým efektívne zistím či k-ty užívateľ existuje alebo nie.

Riešenie

Použité knižnice:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Definície symbolických konštánt:

```
#define VELKOST_MENA 100 – určuje počet písmen v mene užívateľa
#define VELKOST_STRANKY 100 – počet písmen v mene stránky
#define VELKOST_HASH_TABULKY 100000 – veľkosť hash tabuľky
```

ďalšie nám len definujú možné stavy/možnosti ktoré zasielame do funkcií a ovplyvňujú čo dané funkcie vykonajú (využívam to pre lepšiu čitateľnosť kódu a prehľad):

```
#define DVOJITA_PRAVA_ROTACIA 3
#define DVOJITA_LAVA_ROTACIA 2
#define PRAVA_ROTACIA 1
#define LAVA_ROTACIA 0
#define TAZKA_LAVA_STRANA 2
#define TAZKA_PRAVA_STRANA -2
#define ROVNOST 0
#define UNLIKE 0
#define GET_USER 1
#define NEEXISTUJE NULL
```

Vlastné štruktúry:

```
typedef struct stranka{
```

```
    int pocet_like;
    char meno[VELKOST_STRANKY];
    struct stranka *dalsia;
    struct strom *vrchol;
}STRANKA;
```

Táto štruktúra uchováva mená stránok a vrchol AVL stromu. Obsahuje:
-pocet_like čo je vlastne počet like-ov od rôznych užívateľov. V programe to využívame pri vyhľadávaní K-teho užívateľa. Ak je táto hodnota menšia ako hodnota k, vieme že taký like v poradí neexistuje (zrýchľuje nám to program nakoľko nemusíme prehľadávať strom).
-meno (meno stránky)
-pointer na ďalšiu stránku. To využívame, ak máme viac stránok, ktorým sa vypočíta rovnaký hash (aby sa neprepísali, tak ich zreťazíme – spájaný zoznam).
-vrchol je smerník ukazujúci na štruktúru stromu (AVL - stromu) v ktorom si uchovávame užívateľov.

```
typedef struct strom{
    char meno[VELKOST_MENA];
    int vyska;
    int pocet_lave_deti;
    struct strom *prave_dieta;
    struct strom *lave_dieta;
}STROM;
```

Túto štruktúru využívame pri vytváraní AVL stromu, t.j. ak pracujeme s užívateľom. Jej obsah je nasledovný:
-vyska ktorá nám hovorí počet krokov k najnižšiemu listu.
-pocet_lavych_deti nám hovorí o počte, koľko prvkov obsahuje ľavý podstrom (rátame aj jeho). Táto informácia sa využíva pri vyhľadávaní k-teho užívateľa.
-meno obsahuje meno užívateľa
-smerník na ľavý (lave_dieta) a pravý (prave_dieta) podstrom (dieťa)

Globálne premenné:

```
struct stranka *hash_tabulka[VELKOST_HASH_TABULKY];
```

Hash_tabulka je naša jediná globálna premenná obsahujúca celú našu hash tabuľku so zoznamom stránok.

Funkcie

STROM *vytvor_vrchol(char *meno){} – vracia nám vytvorenú a nastavenú štruktúru AVL-stromu. Nastavuje meno vrcholu (aké zašleme), výšku (1), počet ľavých detí (0), a nastaví svoje „deti“ na NULL

int vrat_vysku(STROM *vrchol){}- funkcia vracia výšku stromu ak existuje daný vrchol, ak neexistuje vracia 0

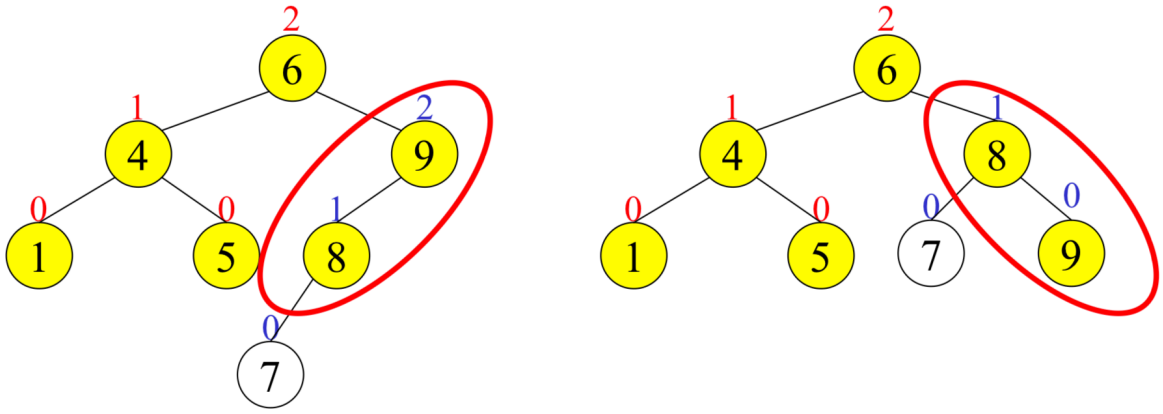
int balanc_vrcholu(STROM *vrchol){} – funkcia vracia balanc vrcholu, čo je vlastne veľkosť výšky ľavého dieťaťa – veľkosť výšky pravého dieťaťa

int vyska_vrcholu(STROM *lavy, STROM *pravy){}- funkcia vracia hodnotu vyššieho vrcholu (porovnáva ich)

int porovnanie_mien(char *meno, STROM *vrchol){}- funkcia vracia záporné číslo, ak meno je (lexikálne/ abecedne) menšie ako meno vo vrchole, kladné ak je väčšie a rovné 0 ak je rovné 0.

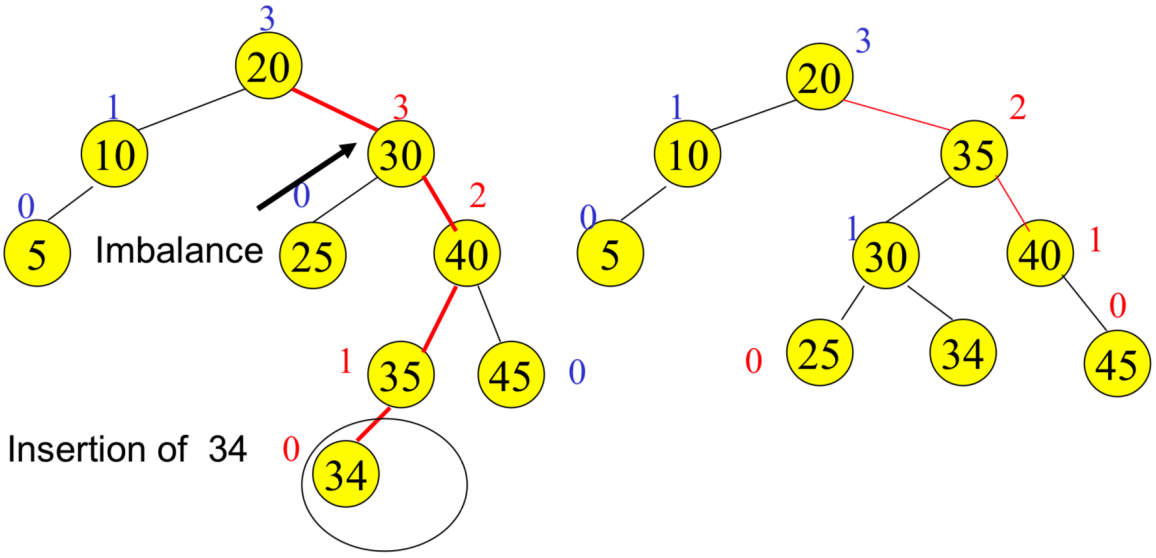
STROM *rotovanie(int smer, STROM *vrchol){}- táto funkcia slúži na rotovanie AVL stromu (tým ho vyvažujeme), posielame jej parametre, ktoré určujú druh rotácie vrchola. Máme 4 možnosti rotácie a to :
-Ľavá rotácia
-Pravá rotácia
-Dvojitá ľavá rotácia
-Dvojitá pravá rotácia

Rotácie sa využívajú pri rotovaní stromu. Druh rotovania určujeme podľa balancu (vyváženia), teda či strom je „ťažší“ na ľavej alebo pravej strane. Jednoduché (nie dvojité) rotácie si môžeme predstaviť ako ťahanie šnúrky. Napr. rotácia doprava sa vykonáva ak je ľavý podstrom (ľavé dieťa) ťažší (má väčšiu výšku).



Obr.1. Pravá rotácia (zdroj: https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf)
Na obrázku je vidno akoby sme 9 chytili a potiahli doprava. Ľavá rotácia je podobná len opačne.

Dvojité rotácie sú vlastne spojením jednoduchých rotácií. Napr. dvojitá ľavá rotácia je najskôr rotácia dieťaťa doprava a následne rotácia vrcholu doľava.



Obr.1. Dvojitá ľavá rotácia (zdroj: https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf)

Na obrázku je vidno ako najskôr rotujeme 40 doprava (dieťa 30ky) a následne rotujem 30 doľava. Dvojitá pravá rotácia je rovnaká len v opačnom poradí.

int update_vysky(STROM *vrchol){}-vracia novú výšku vrchola. Ak sa napríklad vloží prvok do stromu, musíme upraviť výšky. (zistíme výšku väčšieho dieťaťa vyska_vrcholu() a pripočítame k tomu 1)

STROM *vloz_vrchol(STROM *vrchol, char *meno){}- Funkcia vkladá prvok do AVL stromu. Najskôr nájde požadované miesto (presúva sa po deťoch –ak je meno lexikálne menšie ako meno vo vrchole, idem do ľavého dieťaťa a naopak) tým sa dostane na vhodné miesto kde vytvorí vrchol – vytvor_vrchol() . Následne zistí, či je strom vyvážený a ak nie je, tak vykonáva rotovanie (zistovanie prebieha viac krad nakoľko sa rekurzívne vraciame na vrchol stromu). Druh rotovania závisí od balancu a lexikálneho/abecedného porovnania mien. Nakoniec sa vykonáva update výšky vrcholu –update_vysky().

STROM *predchodca_vrcholu(STROM *vrchol){}-funkcia prechádza pravé deti (pokiaľ nejaké sú) a ak príde na posledné vráti ho. Funkcia slúži z časti ako „presuccessor“.

STROM *vymazanie_vrcholu(STROM *vrchol,char *meno){} – slúži na vymazanie vrcholu (ak vymazávame like nejakého užívateľa). Najskôr ho musíme nájsť (podobne ako pri vkladaní), následne zisťujeme, či má iba ľavé dieťa, pravé dieťa, obe deti, alebo ani jedno. Ak má iba ľavé/pravé dieťa tak na miesto vymazávaného prvku dáme jeho dieťa. Ak nemá ani jedno dieťa len ho vymažeme. A ak má obe deti, nájdeme (lexikálne) najbližší menší prvok x a jeho meno nastavíme nášmu prvku s tým, že zavoláme mazanie na prvok x (pričom vieme že prvok x sa nachádza v ľavom dieťati) táto funkcia je takťiež rekurzívna a vďaka tomu sa skontroluje balanc a vykonajú rotácie ako pri vloz_vrchol() taktiež sa vymazávajú ľavé deti.

STROM *k_ty_uzivatel(STROM *vrchol, int poradie){}- funkcia hľadá k-teho užívateľa. Na vyhľadávanie využíva počet ľavých detí vo vrcholoch. Ak hľadáme 30-ty like, najskôr nájdeme stránku, prejdeme na jej AVL strom (na vrchol) a kontrolujeme. Ak vrchol má 25 ľavých detí, vieme, že náš hľadaný like bude v pravom dieťati. Vojdeme do pravého dieťaťa a

zase sa pýtame na počet ľavých detí (pri prechodoch treba dbať na hodnotu, ktorá hovorí o poradí, ktorý v poradí by mal byť hľadaný prvok. Teda, ak idem do ľavého dieťaťa, tak hodnota ostáva, ale ak idem do pravého, musím od danej hodnoty odčítať počet detí, ktoré som skontroloval, t.j počet ľavých detí vrchola z ktorého idem + 1) atď. pokiaľ nenájdeme hľadaný like. Pre väčšiu efektivitu máme v štruktúre stránky vytvorenú premennú, ktorá nám hovorí o celkovom počte like na stránke a vďaka nej vieme zistiť či vôbec v našom prípade 30-ty like existuje bez toho, aby sme museli prejsť nejaký vrchol. Funkcia na konci vráti štruktúru vrcholu.

char *meno_k_ty_uzivatel(STROM *vrchol, int poradie){}- funkcia zavolá funkciu vyhľadania k-teho užívateľa –k_ty_uzivatel() a pošle mu poradie ktoré dekrementuje o 1. Po navrátení vrcholu z funkcie vyberie zo štruktúry meno a vráti ho.

int hash(char *meno){}- funkcia vytvára hash (jedinečnú hodnotu, ktorá určuje miesto vloženia stránky v tabuľke). Hash sa vytvára nasledovne:
VYSLEDOK = (VYSLEDOK * PRVOCISLO) + PISMENO
s tým, že sa to opakuje v cykle pokiaľ neprejdem všetky písmená. Tým, že mi pri tomto spôsobe nastal problém s veľmi veľkým výsledkom, rozhodol som sa posúvať hodnotu o 1 bajt doprava. Môj výsledný hash nakoniec vyzerá takto:
VYSLEDOK = ((VYSLEDOK*17)>>1) + PISMENO

STRANKA *vytvor_stranku(char *meno_stranky, char *meno_uzivatela){}- vracia nám vytvorenú a nastavenú štruktúru stránky. Nastavuje meno stránky (aké zašleme), pocet_like (1) , hodnotu dalsia (nasledujúca štruktúra stránky) na NULL a vytvorí vrchol (štruktúra AVL stromu) –vloz_vrchol()

int porovnanie_stranok(char *meno, STRANKA * hladana){} - funkcia vracia záporné číslo, ak je meno(lexikálne) menšie ako meno stránky, kladné ak je väčšie a rovné 0 ak je rovné 0.

void vyhladanie_stranky_like(char *meno_stranky,char *meno_uzivatela){}- funkcia pomocou mena nechá vypočítať hash – hash() a vyhľadá danú bunku v tabuľke. Ak v tabuľke neexistuje, vytvorí stránku –vytvor_stranku(). Ak v tabuľke existuje, prehľadá daný spájaný zoznam, kde buď nájde danú stránku a vloží do nej uživatela –vloz_vrchol(), alebo nenájde a na konci zoznamu vytvorí stránku – vytvor_stranku().

char *vyhladanie_stranky(int rozhodnutie ,char *meno_stranky, char *meno_uzivatela,int k){}- daná funkcia podľa parametrov plní úlohu buď UNLIKE alebo GET_USER. Obe spočívajú v podobnom princípe ako vyhladanie_stranky_like(), no po nájdení v spájanom zozname podľa parametra buď spustí -vymazavanie_vrcholu() (UNLIKE) alebo meno_k_ty_uzivatel() (GET_USER).

void init(){}-funkcia nám nastaví všetky bunky hash tabuľky (hash_tabulka) na NULL

void like(char *page, char *user){} zavolá funkciu vyhladanie_stranky_like()

void unlike(char *page, char *user){} -zavolá funkciu vyhladanie_stranky s parametrom UNLIKE

char *getuser(char *page, int k){} –zavolá funkciu vyhladanie_stranky s parametrom GET_USER

Testovanie

Svoj program som testoval priebežne. Ako prvé som vytvoril binárny vyhľadávací strom pri ktorom som skúšal vkladať menám a vypisovať si ich postupne inorder výpisom. Následne som doimplementoval vyvažovanie a mazanie a kontroloval som takto vytvorený AVL strom vkladaním mien (funkciu xxx() som ponechal v kóde):

```
root = vloz_vrchol(root, "Jozef");
root = vloz_vrchol(root, "Barbad");
root = vloz_vrchol(root, "Euklid");
root = vloz_vrchol(root, "Xena");
printf("---------------------------------\n");
xxx(root); //inorder vypis
printf("---------------------------------\n");
```

vo výpise som si nechal vypisovať aj ľavé deti, výšku stromu a balanc.
Keď som si bol istý, že AVL strom funguje, vytvoril som hash tabuľku a pridávanie stránok. To som kontroloval pridávaním stránok a následnou kontrolou v Xcode pomocou breakpointov som skúmal rozptyl. Keď som to už celé spojil, vypísal som si krajné hodnoty (možnosti kde by mohla nastať chyba) a následne ich skúšal (striedavé like(), unlike(), getuser()). Tým pádom moje testovanie bolo priebežné.

Zložitosť

Časová zložitosť:

Čo sa týka časovej zložitosti, využili sme hashovaciu funkciu ktorá má zložitosť O(1) (vypočíta sa hash a zvolí sa konkrétna bunka) no ako riešenie jej kolízií sme využili zreťazenie, čo má časovú zložitosť v najhoršom prípade O(n). Pri vyhľadávaní užívateľa sme využili AVL strom ktorý má časovú zložitosť O(log(n)) (pri prehľadávaní sa rozhodujeme vždy medzi dvoma deťmi a prejdenie od vrcholu ku ľubovoľnému vrcholu pri vyváženom strome je počet krokov log(n)). Čiže zlozitosť getuser(), like(), unlike() by tým pádom mala byť O(n) no reálne vďaka hash tabuľke by mala byť priemerná zložitosť O(n.log(n)). Init() má časovú zložitosť O(1).

Priestorová zložitosť:

Priestorová zložitosť závisí od viacerých vecí a to veľkosť hash tabuľky (VELKOST_HASH_TABULKY * veľkosť štruktúry stránky) + počet stranok (počet zreťazených stránok * veľkosť štruktúry stránky) + počet like-ov (počet like-ov užívateľov * veľkosť štruktúry stromu).

Použitá literatúra

Z týchto materiálov som čerpal informácie a naučil som sa ako presne funguje AVL strom.

Pomocne materialy k AVL stromu.
Rotacie, vkladanie:
https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture12.pdf
(Gunnar Gothsalks) - odstranovanie z AVL stromu:
http://www.eecs.yorku.ca/course_archive/2008-09/F/2011/slides/22-AVLtrees.pdf

Zhodnotenie

Tento program ma veľa naučil čo sa týka binárnych vyvažovacích stromov (AVL), hashovacích funkcií a spracovania väčšieho množstva dát vo väčšej rýchlosti. Čo sa týka efektívnosti riešenia si myslím, že AVL strom bol správnou voľbou pre jeho rýchle prehľadávanie. Spojenie s hash tabuľkou (kolízie zreťazením) mi dovolilo rýchly prístup k potrebným užívateľom. Toto riešenie je rýchle a výhodne. Napríklad, ak by som využil lineárne hashovanie, potreboval by som veľmi veľkú tabuľku a nastal by problém, že by som musel vedieť maximálny počet stránok, čo mi nevieme. Ak by som použil obyčajný a nie vyvažovací strom, prehľadávanie by bolo veľmi pomalé. Tento príklad mi ukázal nové možnosti spracovania, ukladania a následného upravovania dát.