Adam Žúrek, ID: 79829

Dátové štruktúry a algoritmy

(Zadanie č. 2)

Obsah zadana: Sociálne siete teraz letia. Ľudia vám zadarmo vypĺňajú databázu osobnými údajmi a vy ich len ďalej predávate inzerentom. Chcete, aby vaša sociálna sieť bola lepšia ako ostatné, tak ste sa rozhodli, že na fanúšikovskej stránke si budú návštevníci môcť naživo prehliadať aktuálny zoznam fanúšikov. Napr. keď si tak prehliadate zoznam fanúšikov na vašej obľúbenej stránke, očami ste na 4774. mieste v abecednom poradí, a v tom sa posunie fanúšik o jedno miesto vyššie, pretože niekto v abecede skôr stránku odlajkoval. Návštevník má teda presný prehľad o aktuálnom stave fanúšikov podľa abecedy.

Túto funkcionalitu budete realizovať nasledovnými funkciami:

```
void init();
void like(char *page, char *user);
void unlike(char *page, char *user);
char *getuser(char *page, int k);
```

Vo vašom riešení implementujte tieto funkcie. Testovač najskôr zavolá funkciu `init()`, v ktorej si môžete pripraviť vaše dátové štruktúry v globálnych premenných (napr. inicializácia stromu). Následne bude volať funkcie `like()`, `unlike()` a `getuser()`, s nasledovným významom:

`void like(char *page, char *user)` - Používateľ s menom user lajkoval stránku s názvom page

`void unlike(char *page, char *user)` - Používateľ s menom user odlajkoval stránku s názvom page.

`char *getuser(char *page, int k)` - Zaujíma nás meno k-teho používateľa v abecednom poradí, ktorý lajkuje stránku page. Ak je týchto používateľov menej ako k, tak funkcia vráti NULL.

Vďaka tejto funkcii sa vašej sociálnej sieti začalo veľmi dariť, a prichádzajú na ňu desiatky tisíc požiadaviek za sekundu. Implementujte vyššie uvedenú funkcionalitu čo možno najefektívnejšie.

Pred odovzdaním si vlastnú implementáciu dôkladne otestujte. Do testovača by ste mali odovzdávať už funkčnú verziu. Pri spustení vám testovač vypíše protokol v rozličných scenároch.

Adam Žúrek, ID: 79829

Vypracovanie: V mojom riešení stránku predstavuje jeden AVL strom, ktorého listy sú používatelia sociálnej siete, ktorí lajkli danú stránku. V sociálnej sieti je veľa stránok, ktoré môžu používatelia lajknúť, či dislajknúť. Preto som vytvoril hashovaciu tabuľku spájaného zoznamu, ktorého prvky sú smerníky na korene AVL stromov, čiže stránok. K jednotlivým prvkom poľa sa program dostane cez kľúč, ktorý je generovaný z názvu stránky, čiže reťazca charov za pomoci Hornerovej schémy.

Rozbor jednotlivých funkcií môjho programu som rozdelil na 3 časti:

1. Funkcie AVL stromu:

AVL strom je samovyvažovací binárny strom, ktorý sa používa na stabilne-rýchle vyhľadávanie prvkov. Na vyhľadanie používateľov v stánke je výborným otimalizovaným riešením. AVL strom vyžaduje veľa rozličných funkcií, ktoré sú ale veľmi potrebné.

- `NODE* newNode(char* value)` slúži na vytvorenie nového listu v strome. Funkcia obsahuje parameter v tvare smerníka v pamäti na začiatok reťazca charov, predstavujúci názov nového používateľa, ktorého do stromu pridáme.

- `int getHeight(NODE* n)` – funkcia vracajúca výšku aktuálneho listu, ktorý dostala v parametri. Ak dostala v parametri list s nepriradenou pamäťou, vráti nulu.

- `int getBalance(NODE* node)` – vracia rozdiel výšok ľavého a pravého childa lista v parametri. Ak dostala v parametri list s nepriradenou pamäťou, vráti nulu.

- `int max(int a, int b)` – vracia väčšie číslo z čísel v parametri.

- `NODE* minValue(NODE* node)` – vracia najlavejší list z aktuálne zadaného listu v parametri.

- `NODE* rightRotate(NODE* z)` – funkcia vyvažujúca strom pravou rotáciou jeho listov, ktorý je nevyvážený.

- `NODE* leftRotate(NODE* z)` – funkcia vyvažujúca strom ľavou rotáciou jeho listov, ktorý je nevyvážený.

- `NODE* addNode(NODE* node, char* value)` – funkcia pridávajúca list do stromu. Miesto, kde má funkcia nový list uložiť je hľadaný za pomoci rekurzie. Ak sa také miesto nájde, funkcia musí zmeniť výšky všetkých zasiahnutých listov a skontrolovať ich vyváženosť. Ak je strom po vložení nevyvážený, musí sa uskutočniť práve jedna z možných rotácií: (RR,LL,RL,LR), pričom rotácia RR je pravá rotácia, LL je ľavá rotácia a RL a LR kombináciou pravej a ľavej rotácie.

- `NODE* deleteNode(NODE* node, char* value)` – funkcia mazajúca list zo stromu podľa zadaného mena listu v parametri. Rekurzívne prvok nájde, skontroluje, koľko má childov, vymaže prvok a kontroluje vyváženosť stromu. Dokiaľ nebude vyvážený rotuje.

- `char* inOrder(NODE* node, int* k)` – funkcia hľadajúca k-ty prvok v strome podľa in order alogitmu. Nájde k-te meno v abecede používateľov, ktorí sa nachádzajú v strome.

2. Funkcie hashovacej tabuľky:

Hashovacia tabuľka je ďalším veľmi dobrým riešením problému vyhľadávania. K prvkom sa vie program veľmi rýchlo dostať cez tzv. kľúče, ktoré za pomoci algoritmu rýchlo priradia prvku miesto v poli a následne ho aj pomocou neho na tom mieste aj vyhľadá. Kvôli možným kolíziam som implementoval reťazenie.

- `unsigned int charToInt(char* page)` – funkcia vytvárajúca kľúč k danému menu vytvorenej stránky podľa Hornerovej schémy.

- `LIST* findPage(char* page)` – funkcia hľadajúca zadanú stránku v hashovacom poli. Vráti smerník na štruktúru obsahujúcu smerník na koreň stromu.

3. Funkcie zadané v obashu zadania:

- `void init()` – funkcia inicializujúca pole smerníkov predstavujúcich stránky na NULL
- `void like(char *page, char *user)` – funkcia pridávajúca zadaného používateľa do zadanej stránky. Teda funkcia zavolá funkciu findPage, ktorá nájde v hashovacej tabuľke potrebnú stránku so smerníkom na koreň stromu používateľov. Na vyhľadaný koreň stromu sa zavolá funkcia addNode, ktorá pridá používateľa do daného stromu.
- `void unlike(char *page, char *user)` – funkcia obdobne ako funkcia like zavolá najskôr funkciu findPage a následne deleteNode, do ktorej parametra zadá nájdený koreň stromu.
- `char *getuser(char *page, int k)` – funkcia taktiež vyhľadá stránku za pomoci findPage a na nájdený koreň zavolá funkciu inOrder, ktorá vráti smerník na začiatok reťazca znakov predstavujúce meno používateľa.

Záver: Zadanie je funkčné a dostatočné k realizácii v praxi. Efektivita vyhľadávania je výborná a celý program je veľmi dobre optimalizovaný. Riešenie má zopár menších chýb, kvôli ktorým nezbehol správne scenár číslo 7. Taktiež by sa dala rýchlosť ešte zlepšiť pomocou implementácie červeno-čierneho stromu, pri ktorom je vkladanie prvkov do stromu rýchlejšie. Ak by sa veľkosť hashovacej tabuľky zvýšila, predošlo by sa tak viacerým kolíziam a program by bol síce pamäťovo náročnejší, ale výrazne rýchlejší.

Adam Žúrek, ID: 79829

Časová zložitosť:

- operácie AVL stromu (like, unlike) – O(log n)
- getUser – O(n)
- hashovacia tabuľka – najlepší prípad O(1), najhorší O(n)

Pamäťová zložitosť:

- operácie AVL stromu (like, unlike) – P(n)
- getUser – P(1)
- hashovacia tabuľka – P(n)