Strukture podataka

Adrian Satja Kurdija
askurdija@gmail.com

Osnovne strukture

- **Niz**
  - fiksne veličine
  - lako dohvaćamo i mijenjamo element *(random access)*
- **C++ vector**
  - varijabilne veličine
  - lako dohvaćamo i mijenjamo element *(random access)*
  - lako je dodavati na kraj i izbacivati s kraja

Problem: mnoge druge operacije su spore.

Ubrzavanje operacija

Ako gradimo strukturu pogodnu za brzo izvođenje neke operacije, neke druge operacije postat će sporije.

Svaka od struktura koju ćemo navesti u nastavku prikladna je samo za određenu vrstu operacija.

Red

Podržava ubacivanje elementa na kraj i dohvaćanje / izbacivanje s početka.

Ilustracija: red ljudi u banci.

Implementacija: niz s kazaljkama na početak i kraj -- izbacivanje se svodi na pomicanje lijeve kazaljke.

U C++u najbolje je koristiti *queue*.

Najčešća primjena: BFS (traženje najkraćeg puta u grafu ili tablici).

Stog

Podržava stavljanje na vrh i dohvaćanje / skidanje s vrha.

Ilustracija: hrpa tanjura u restoranu.

U C++ možemo koristiti *stack* ili *vector*.

Eksplicitno se koristi npr. u Tarjanovom algoritmu za traženje SCC-a u usmjerenom grafu.

Implicitno je prisutan u pozivanju potprograma, rekurziji, DFS-u...

Monotoni stog

Običan stog u koji elemente dodajemo i izbacujemo tako da budu sortirani uzlazno ili silazno.

Koristi se u zadacima gdje treba pronalaziti npr. prvi manji element lijevo od trenutnog.

Primjeri:

- traženje najvećeg pravokutnika u histogramu
- http://www.spoj.com/problems/DIFERENC/
- zadatak *Patrik* (HIO 2007.)

C++ deque: obostrani red

```
#include <deque>
deque<int> D;
```

- `.size()` *// veličina*
- `.back()` *// dohvati zadnji*
- `.front()` *// dohvati prednji*
- `.push_back(x)` *// ubaci na kraj*
- `.push_front(x)` *// ubaci na početak*
- `.pop_back()` *// makni s kraja*
- `.pop_front()` *// makni s početka*

Vezana lista

Podržava brzo ubacivanje i izbacivanje elementa bilo gdje u sredinu liste, ali samo ako se na tome mjestu već nalazimo.

Treba nam petlja da bismo došli do nekog mjesta u nizu.

Primjer: zadatak *Cenzura* (http://hsin.hr/dmih09/zadaci/pascal_c_cpp/dan1/druga/zadaci.pdf)

Vezana lista: ideja

Svaki element ima pokazivač na sljedeći element (ako ima i na prethodni, riječ je o dvostruko vezanoj listi).

Ubacivanje i izbacivanje svodi se na preusmjeravanje pokazivača.

Implementirana u STL-u:

http://www.cplusplus.com/reference/list/list/list/

STL klase u C++u

- *priority_queue*
  - dohvaćanje najvećeg elementa *O(1)*
  - izbacivanje najvećeg elementa *O(log N)*
  - ubacivanje elementa *O(log N)*
  - koristi se npr. u Dijkstrinom algoritmu
  - implementiran je binarnim stablom zvanim *heap*

STL klase u C++u

- *set*
  - moćna struktura: može sve što i priority queue, a i više od toga
  - zato je malo sporiji -- usprkos dobroj složenosti, ima veliku konstantu
  - briše duplikate (ako to ne želimo, koristimo *multiset*)
  - implementiran je balansiranim binarnim stablom traženja (*red-black tree*)

STL klase u C++u

- *set*
  - ubacivanje, izbacivanje bilo kojeg elementa: *O(log N)*
  - najmanji, najveći element (*begin*, *rbegin*): *O(1)*
  - pronalazak određenog elementa (*find* ili *count*): *O (log N)*
  - binarno pretraživanje (*lower_bound* za prvi veći ili jednak, *upper_bound* za prvi veći): *O(log N)*
  - prolazak po svim elementima: *O(N log N)*

STL klase u C++u

- *map*
  - omogućuje da pišemo stvari poput

    ```
    ocjena["Marko"] = 5;
    ```
  - zapravo u nju ubacujemo parove (ključ, vrijednost) i očitavamo vrijednost za neki ključ
  - funkcije su složenosti *O(log N)*, također je velika konstanta

Nedostatak seta

C++ set bio bi ultimativna struktura kad bio *indeksiran*, tj. kad bi mogao odgovarati i na sljedeće upite:

- koji je element k-ti po veličini?
- za zadani element, koji je on po veličini?

Budući da ne može, natjecatelji ponekad sami implementiraju binarno stablo traženja koje će to podržati.

(I set je implementiran jednim takvim stablom.)

Binarno stablo traženja, ukratko

Osnovna ideja: lijevo podstablo svakog elementa sadrži manje elemente, a desno podstablo veće elemente.

Na taj način možemo, krenuvši od korijena, pronaći traženi element u stablu u složenosti *O (dubina elementa)*.

Ubacivanje u stablo vršimo na sličan način.

Binarno stablo traženja, ukratko

Problem: ako stablo ispadne “duguljasto”, složenost raste na *O(N)*, a mi želimo *O(log N)*.

Jedno je rješenje brinuti o balansiranosti stabla. Tako dobivamo *AVL tree*, *splay tree* ili *red-black tree*.

Drugo je rješenje je randomizirati način ubacivanja u stablo. Tako dobivamo *randomized binary search tree* ili *treap*.

Treće je rješenje periodički (ili kad maksimalna dubina postane prevelika) iznova izgraditi cijelo stablo tako da bude balansirano.

Binarno stablo traženja: indeksiranje

Traženo indeksiranje elemenata (tko je k-ti po veličini, koji je ovaj po veličini) ostvarujemo na sljedeći način:

- U svakom čvoru dodatno pamtimo broj elemenata njegovog podstabla.
- Te brojeve koristimo prilikom traženja elementa spuštajući se stablom.