Konstrukcija i analiza algoritama

AISP obnavljanje

Rekurentne jednačine za analizu složenosti, master teorema

Složenost rekurzivnih funkcija se može opisati **rekurentnim jednačinama**. Rekurentne jednačine gde se problem svodi na problem dimenzije za jedan manje od početne:

- T(n) = T(n - 1) + O(1), T(0) = O(1) - rešenje: O(n), npr. traženje minimuma niza
- T(n) = T(n - 1) + O($\log n$), T(0) = O(1) - rešenje: O(n$\log n$), npr. formiranje balansiranog binarnog drveta
- T(n) = T(n - 1) + O(n), T(0) = O(1) - rešenje: O(n^2), npr. sortiranje selekcijom

Rekurentne jednačine gde se problem svodi na dva ili više problema dimenzije za jedan manje od početne:

- T(n) = 2T(n - 1) + O(1), T(0) = O(1) - rešenje O(2^n), npr. Hanojske kule
- T(n) = T(n - 1) + T(n - 2) + O(1), T(0) = O(1) - rešenje: O(2^n), npr. Fibonačijevi brojevi

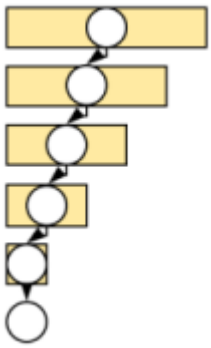
Rekurentne jednačine gde se problem svodi na jedan ili više problema značajno (bar duplo) manje dimenzije od početne:

- T(n) = T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: O($\log n$), npr. binarna pretraga sortiranog niza
- T(n) = T($\frac{n}{2}$) +O(n), T(0) = O(1) - rešenje: O(n), npr. pronalaženje medijane
- T(n) = 2T($\frac{n}{2}$) + O(1), T(0) = O(1) - rešenje: O(n), npr. obilazak potpunog binarnog drveta
- T(n) = 2T($\frac{n}{2}$) + O(n), T(0) = O(1) - rešenje: O(n$\log n$), npr. sortiranje objedinjavanjem

Da bismo stekli intuiciju kako se dobijaju ova asimptotska rešenja, možemo primeniti **"odmotavanje rekurzije"**.

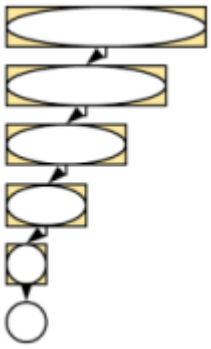
Primer: T(n) = T(n - 1) + O(1), T(0) = O(1)

- T(n) = T(n - 1) + O(1) = T(n - 2) + O(1) + O(1) = ... = T(0) + n · O(1) = O(1) + n · O(1) = O(n)



Primer: T(n) = T(n - 1) + O(n), T(0) = O(1)

- T(n) = T(n - 1) + cn = T(n - 2) + c(n - 1) + cn = ... = T(0) + c · (1 + ... + n) = O(1) + c$\frac{n(n+1)}{2}$ = O(n^2)



Jednačine zasnovane na dekompoziciji problema na manje potprobleme koje su oblika T(n) = aT($\frac{n}{b}$) + O(n^k), T(0) = O(1) se rešavaju na osnovu **master teoreme**:

- Rešenje rekurentne jednačine T(n) = aT($\frac{n}{b}$) + cn^k, T(0) = O(1), gde su a i b celobrojne konstante $\geq$ 1 i c i k pozitivne realne konstante je:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \log_b a > k \\ \Theta(n^k \log n), & \log_b a = k \\ \Theta(n^k), & \log_b a < k \end{cases}$$

Primeri:

- T(n) = 2T($\frac{n}{2}$) + O(1), T(0) = O(1) $\Rightarrow$ a = 2, b = 2, k = 0 $\Rightarrow$ $\log_2 2$ = 1 > k $\Rightarrow$ Θ(n)
- T(n) = 2T($\frac{n}{2}$) + O(n), T(0) = O(1) $\Rightarrow$ a = 2, b = 2, k = 1 $\Rightarrow$ $\log_2 2$ = 1 = k $\Rightarrow$ Θ(n$\log n$)
- T(n) = T($\frac{n}{2}$) + O(n), T(0) = O(1) $\Rightarrow$ a = 1, b = 2, k = 1 $\Rightarrow$ $\log_2 1$ = 0 < k $\Rightarrow$ Θ(n)

Strukture podataka - klasifikacija, primeri upotrebe

Efikasna oragnizacija podataka u memoriji predstavlja važan preduslov pisanja efikasnih programa. Pored primitivnih tipova podataka savremeni programski jezici nude veliki broj bibliotečkih struktura podataka. **Strukture podataka** možemo posmatrati preko njihovih operacija, tj. funkcionalnosti koje pružaju pa je bitno poznavati njihov apstraktni interfejs, ali imati i osećaj o složenosti operacija. Po načinu implementacije strukture podataka se mogu grupisati na:

- **sekvencijalne kontejnere** $\rightarrow$ **array, string, vector, list, forward_list, deque**
- **adaptore kontejnera** $\rightarrow$ **stack, queue, priority_queue**
- **asocijativne kontejnere** $\rightarrow$ **set, mulitset, map, multimap**
- **neuređene asocijativne kontejnere** $\rightarrow$ **unordered_set, unordered_multiset, unordered_map, unordered_mulitmap**

Sekvencijalni kontejneri se koriste za skladištenje serija elemenata. Oni predstavljaju određena uopštenja primitivnih statički alociranih nizova. Karakteriše ih mogućnost obilaska svih elemenata redom (nekad i u oba pravca) kao i mogućnost indeksnog pristupa elementima. Sve osim array (koji je samo omotač primitivnog statičkog niza) su dinamički alocirani i dopuštaju umetanje i brisanje elemenata na početak, sredinu ili kraj pri čemu složenost zavisi od konkretnog kontejnera.

Adaptori kontejnera predstavljaju sloj iznad nekog od postojećih sekvencijalnih kontejnera. Podrazumevani sekvencijalni kontejner im se može promeniti prilikom deklarisanja promenljivih.

- **stack** implementira funkcije steka (**LIFO**) gde se elementi mogu dodavati i uklanjati sa jednog kraja.
- **queue** implementira funkcije reda (**FIFO**) gde se elementi mogu dodavati na jedan, a uklanjati sa drugog kraja.
- **priority_queue** implementira funkcije reda sa prioritetom gde se elementi dodaju u proizvoljnom redosledu, a uklanjaju u nerastućem poretku vrednosti.

Asocijativni kontejneri podrazumevaju da se umetanje, pretraga i brisanje elemenata vrši na osnovu vrednosti, a ne na osnovu pozicije na kojoj se nalaze, tj. elementima se pristupa preko ključa, a ne preko indeksa. Osnovni asocijativni kontejneri su **skupovi** i **mape**, tj. **rečnici**. Skupovi odgovaraju matematičkim skupovima i pružaju efikasno dodavanje i izbacivanje elemenata, kao i proveru da li element pripada skupu. Rečnici čuvaju konačna preslikavanja u kojima se ključevima pridružuju vrednosti. Asocijativni kontejneri se dele na **uređene** i **neuređene**. Uređeni kontejneri nude dodatne funkcionalnosti kao što je ispis elemenata u traženom poretku, ali su ponekad sporiji od neuređenih. Svi oni pretpostavljaju da se njihovi elementi (skupovi) odnosno ključevi (mape) mogu porediti relacijskim operatorom. Obično su implementirani pomoću samobalansirajućih uređenih binarnih drveta i osnovne operacije su im logaritamske složenosti. Takođe postoje i **multiskup**, gde je moguće postojanje više istih elemenata u skupu, i **multimapa**, gde jedan ključ može imati više pridruženiih vrednosti.

Neuređeni asocijativni kontejneri su obično implementirani pomoću heš-tabela i karakteriše ih amortizovana konstantna složenost osnovnih operacija. Oni pretpostavljaju da postoji heš-funkcija koja elemente skupa/ključeve mapa slika u celobrojne vrednosti koje određuju njihovu poziciju. Prilikom heširanja moguće su i **kolizije**, tj. da više elemenata ima iste heš-kodove, tj. da se slikaju na istu poziciju. Jedan od načina rešavanja kolizija jeste da se na pozicijama čuva lista svih elemenata koji imaju tu vrednost heš-funkcije.

Strukture podataka sa sekvencijalnim pristupom

Niz - array predstavlja samo tanak omotač oko klasičnog statičkog niza čija je dimenzija poznata u trenutku deklaracije. On omogućava da se klasični statički nizovi mogu koristiti na isti način kao i drugi kontejneri, npr. dodela niza nizu, poređenje nizova sa == i slično.

Vektor - vector je implementiran preko dinamičkog niza koji se realocira po potrebi. Ova implementacija omogućava efikasan pristup, iteraciju u oba smera i dodavanje i skidanje elemenata sa kraja. Može biti neefiksan kada čuva velike objekte zbog česte realokacije.

- **push_back** - dodavanje na kraj
- **pop_back** - uklanjanje sa kraja

Niska - string je implementiran slično kao vektor uz dodatne optimizacije i operacije specifične za podatke tekstualnog tipa.

Jednostruko povezana lista - forward_list je implementirana preko jednostruko povezane liste. Omogućava dodavanje na kraj i početak, kao i brisanje sa početka u vremenu O(1), dok brisanje sa kraja zahteva O(n) jer se mora izmeniti pretposlednji čvor. Umetanje i brisanje ostalih elemenata, kao i indeksni pristup zahtevaju vreme O(n).

Dvostruko povezana lista - list je implementirana preko dvostruko povezane liste. Omogućava i dodavanje i brisanje sa kraja i početka u vremenu O(1). Umetanje i brisanje ostalih elemenata je takođe u vremenu O(1). Mana u odnosu na jednostruke liste je to što je potrebno više memorije, kao i to što pojedinačne operacije mogu biti malo sporije jer se ažuriraju dva pokazivača. Prednost je efikasnije brisanje sa kraja i iteracija unazad. Liste se sve manje koriste jer je alokacija čvorova skupa mada prednost u odnosu na dek i nizove je to što mogu da čuvaju i velike podatke jer ne dolazi do realokacije.

Red sa dva kraja - deque dopušta da se elementi dodaju i uzimaju sa bilo kog kraja reda odnosno kombinuje funkcionalnosti i steka i reda. Implementiran je kao specifična struktura podataka koju možemo zamisliti kao vektor u kome se nalaze pokazivači na blokove elemenata fiksne veličine. Zbog toga on nudi efikasno dodavanje i uklanjanje i sa početka i sa kraja ali i iteraciju u oba smera i indeksni pristup. Sve operacije su složenosti O(1).

- **push_front**/**push_back** - dodavanje na početak/kraj
- **front**/**back** - čitanje sa početka/kraja pod pretpostavkom da red nije prazan
- **pop_front**/**pop_back** - uklanjanje sa početka/kraja
- **empty** - provera da li je red prazan
- **size** - broj elemenata u redu

Strukture podataka sa asocijativnim pristupom

Skupovi pružaju efikasno dodavanje i izbacivanje elemenata kao i proveru da li se element nalazi u skupu. Podržan je kroz klase **set** i **unordered_set**. U njima nije dozvoljeno ponavljanje elemenata, pa postoje **multiset** i **unordered_multiset** u kojima jeste.

- **set** je implementiran pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi skupa gde nije moguće postojanje više čvorova sa istim vrednostima. Drvo je uređeno binarno drvo ako je prazno ili ako je njegovo levo i desno poddrvo uređeno i ako je čvor u korenu veći od svih čvorova u levom poddrvetu i manji od svih čvorova u desnom poddrvetu.
- **multiset** je implementiran pomoću uređenog binarnog drveta u čijim se čvorovima nalaze elementi multiskupa sa dozvoljenim ponavljanjem elemenata ili u čijim čvorovima se nalaze elementi i broj njihovih ponavljanja.
- **unordered_set** i **unordered_multiset** su implementirani preko heš-tabela.

Operacije (kod neuređenog skupa su malo efikasnije):

- **insert** - ubacuje element u skup ako već nije u skupu. Složenost kod set je O($\log n$). Kod unordered_set je najgora složenost O(n), a prosečna O(1). Složenost dodavanja m elemenata je O(m$\log m$).
- **erase** - uklanja element iz skupa ako je u skupu. Složenost ista kao kod insert.
- **find** - vraća iterator na element traženog elementa ili end ako se on ne nalazi u skupu. Složenost ista kao kod insert.
- **size** - vraća broj elemenata skupa.
- **lower_bound** i **upper_bound** - ove operacije imaju uređeni skupovi.

Mape predstavljaju kolekcije podataka u kojima se ključevima pridružuju neke vrednosti. Podržane su kroz klase **map** i **unordered_map**. Ključevi sortirane mape mogu biti samo vrednosti koje se mogu porediti relacijskim operatorom, a ključevi nesortirane mape mogu biti samo vrednosti koje se lako mogu pretvoriti u broj. U njima nije dozvoljeno pridružiti više vrednosti istom ključu, pa postoje **multimap** i **unordered_multimap** u kojima jeste.

- **map** je implementirana pomoću uređenog binarnog drveta u čijim se čvorovima nalaze ključevi i pridružene vrednosti gde nije moguće postojanje više vrednosti za isti ključ.
- **multimap** je implementirana pomoću uređenog binarnog drveta u čijim se čvorovima nalaze ključevi i pridružene vrednosti gde je moguće postojanje više vrednosti za isti ključ.
- **unordered_map** i **unordered_multimap** su implementirane preko heš-tabela.

Operacije (kod neuređene mape su malo efikasnije):

- **find** - pretraga preko ključa, vraća iterator na element ili end ako element nije u mapi. Složenost je O($\log n$).

- **insert** - ubacuje par ključ-vrednost u mapu. Složenost je O($\log n$).
- **erase** - uklanja par ključ-vrednost iz mape. Složenost je O($\log n$).

Stek - operacije, implementacija, ilustrovanje kroz primere

Stek predstavlja kolekciju podataka gde se oni dodaju po **LIFO** (**last-in-first-out**) principu, tj. element se može dodati i skinuti samo sa vrha steka. Operacije su složenosti O(1), podržane kroz klasu **stack**:

- **push** - postavlja element na vrh steka
- **pop** - skida element sa vrha steka ako nije prazan
- **top** - čita element sa vrha steka ako nije prazan
- **empty** - proverava da li je stek prazan
- **size** - vraća broj elemenata na steku
- **emplace** - umesto push ako se na steku čuvaju parovi ili n-torke

Stek obično podrazumeva da se koristi neki oblik niza ili jednostruko povezane liste. Ukoliko je broj elemenata koji se može naći na steku ograničen može se koristiti statički niz, a ako nije onda se koristi dinamički niz ili lista. U C++ stek za implementaciju koristi vector. Na primer, stack<int> podrazumeva stack<int, vector<int>>.

Primer: Učitavati linije do kraja ulaza, a potom ih ispisati u obrnutom redosledu.

```
void obrniLinije() {
        stack<string> linije;
        string linija;
        while(getline(cin, linija))
                linije.push(linija);
        while(!linije.empty()) {
                cout << linije.top() << endl;
                linije.pop();
        }
}
```

Red - operacije, implementacija, ilustrovanje kroz primere

Red predstavlja kolekciju podataka gde se oni dodaju po **FIFO** (**first-in-first-out**) principu, tj. element se dodaje na kraj, a skida sa početka reda. Operacije su složenosti O(1), podržane kroz klasu **queue**:

- **push** - postavlja element na kraj reda
- **pop** - skida element sa početka reda ako nije prazan
- **top** - čita element sa početka reda ako nije prazan
- **empty** - proverava da li je red prazan
- **size** - vraća broj elemenata u redu
- **emplace** - umesto push ako se u redu čuvaju parovi ili n-torke

Red obično podrazumeva da se koristi neki oblik niza ili liste. Ukoliko je broj elemenata koji se može naći na steku ograničen može se koristiti statički niz gde se elementi smeštaju u krug, a ako nije onda se koristi dinamički niz ili lista. U C++ stek za implementaciju koristi deque. Na primer, queue<int> podrazumeva queue<int, deque<int>>.

Primer: Ispisati poslednjih k linija standardnog ulaza.

```
void ispisiLinije(int k) {
        queue<string> linije;
        string linija;
        while(getline(cin, linija)) {
                linije.push(linija);
                if(linije.size() > k)
                        linije.pop();
        }
        while(!linije.empty()) {
                cout << linije.front() << endl;
                linije.pop();
        }
}
```

Red sa prioritetom - operacije, implementacija, ilustrovanje kroz primere

Red sa prioritetom je vrsta reda u kome elementi imaju pridružen prioritet, dodaju se u red jedan po jedan, a uvek se uklanja element sa najvećim prioritetom. Operacije dodavanja i skidanja elementa su složenosti O($\log n$), a ostale su O(1) i podržane su kroz klasu **priority_queue**:

- **push** - dodaje element u red
- **pop** - uklanja element sa najvećim prioritetom iz reda ako nije prazan
- **top** - čita element sa najvećim prioritetom ako red nije prazan
- **empty** - proverava da li je red prazan
- **size** - vraća broj elemenata u redu

Red sa prioritetom za skladištenje podataka koristi vector. Na primer, priority_queue<int> podrazumeva priority_queue<int, vector<int>, less<int>> gde je less<int> funkcija za poređenje elemenata i daje opadajuću uređenost elemenata. Moguće je imati red sa rastućim poretkom prosleđivanjem funkcije greater<int>, ali nije moguće imati oba istovremeno. U vektoru su smešteni elementi specijalnog drveta koje se naziva **hip**.

Primer: Ispisati najmanjih k brojeva učitanih sa standardnog ulaza.

```
void kNajmanjih(int n, int k) {
    priority_queue<int, vector<int>, less<int>> red;
    int x;
    while(n--) {
        cin >> x;
        red.push(x);
        if(red.size() > k)
            red.pop();
    }
    while(!red.empty()) {
        cout << red.top() << endl;
        red.pop();
    }
}
```

Hip - primena i implementacija

Hip (**heap**) je struktura podataka koja se koristi za implementaciju redova sa prioritetom. On podržava sledeće operacije:

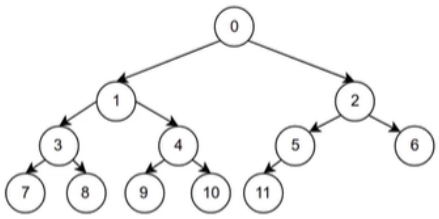
- dodavanje elementa u složenosti O($\log n$)
- uklanjanje najvećeg elementa u složenosti O($\log n$)
- čitanje najvećeg elementa u složenosti O(1)

Umesto hipa koji podržava operacije sa najvećim elementom (**maks-hip**) moguće je razmatrati i hip koji podržava operacije sa najmanjim elementom (**min-hip**), ali nikako oba istovremeno.

Elementi hipa se čuvaju u nizu ali zamišljamo da je struktura podataka binarno drvo koje treba da zadovoljava dva uslova:

- **Strukturni uslov** - drvo se popunjava redom, po nivoima i svaki nivo se popunjava redom.
- **Uslov rasporeda vrednosti** - u svakom čvoru drveta nalazi se vrednost veća ili jednaka od vrednosti koje se nalaze u njegovim sinovima.

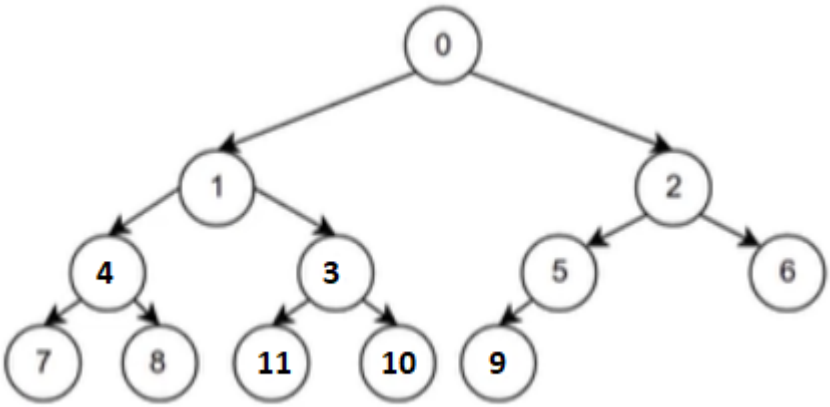
Ukoliko se čvor nalazi na poziciji k, njegovi sinovi (ako postoje) nalaze se na pozicijama 2k+1 i 2k+2. Roditelj čvora na poziciji m $>$ 0 nalazi se na poziciji $\lfloor \frac{m-1}{2} \rfloor$.



Implementacija operacija:

- **Čitanje najvećeg elementa** - najveći element će se uvek nalaziti u korenu drveta, tj. na početnoj poziciji u nizu. Složenost je O(1).
- **Dodavanje elementa** - element se prvo doda na kraj niza, da bi bio zadovoljen strukturni uslov hipa. Nakon toga vrednost se "penje uz hip", tj. razmenjuje se sa svojim roditeljem sve dok se ne zadovolji uslov rasporeda vrednosti hipa. Složenost je O($\log n$) jer je broj penjanja elementa ograničen visinom drveta, a ona je ograničena logaritmom broja čvorova (jer u potpuno drvo od k nivoa staje 2^k - 1 čvorova).
- **Uklanjanje najvećeg elementa** - iz drveta brišemo poslednji list, a njegovu vrednost upisujemo u koren drveta. Nakon toga vrednost se "spušta niz hip", tj. razmenjuje se sa većim od svojih naslednika sve dok se ne zadovolji uslov rasporeda vrednosti hipa. Složenost je O($\log n$) iz istog razloga kao i za dodavanje elementa.

Min-hip, kao i maks-hip, ne mora biti sortiran s leva na desno na konkretnim nivoima. Na sledećoj slici se takođe nalazi min-hip jer su svi čvorovi popunjeni redom i za svaki čvor važi da je manji od svoje dece. Elementi na konkretnim čvorovima nisu sortirani kao u hipu sa prethodne slike.



Binarno stablo

Stablo ili **drvo** je struktura koja prirodno opisuje određene vrste hijerarhijskih objekata. Sastoji se od **čvorova** i **usmerenih grana** između njih. Svaka grana povezuje **roditelja** sa njegovim **detetom**. Čvor koji se zove **koren** stabla nema roditelja, a svaki drugi čvor ima tačno jednog roditelja. **List** je čvor koji nema potomaka. Stablo u kojem svaki čvor ima najviše dva deteta naziva se **binarno stablo**. **Uređeno binarno stablo** (**binarno pretraživačko stablo** ili **stablo binarne pretrage**) je stablo u kome su podaci organizovani u odnosu na neku izabranu relaciju poretka. Na primer, za svaki čvor važi da su svi elementi levog podstabla manji od njega, a svi elementi desnog podstabla veći ili jednaki njemu. Uređeno binarno stablo omogućava efikasniju obradu podataka, ali se ta efikasnost gubi ako je stablo degenerisano. Na primer, ako svaki čvor ima samo desnog potomka ili ako koren ima 10 levih i 2 desna potomka. **Balansirano binarno stablo** je stablo u kome je broj čvorova u levom i desnom podstablu svakog čvora balansiran, odnosno visina levog i desnog podstabla svakog čvora se razlikuje najviše za jedan.