Algoritmi i strukture podataka
4. Čas — Primeri induktivno-rekurzivne konstrukcije

Sana Stojanović Đurđević

October 29, 2018

Pred vama je prateći materijal koji se može koristiti uz skriptu. Sastoji se od rešenja nekoliko zadataka koji su pređeni na vežbama.

1 Apsolutni pobednik na glasanju

Problem: Održano je glasanje i glasalo se za više kandidata. Osoba je apsolutni pobednik ako je dobila strogo više glasova nego svi ostali kandidati zajedno. Definisati algoritam koji na osnovu niza svih glasačkih listića sa glasanja određuje da li postoji apsolutni pobednik i koji je.

Rešenje se zasniva na tome da ako *imamo* apsolutnog pobednika u većem skupu glasova, i izbacimo bilo koja dva *razliita glasa*, onda će manji skup imati apsolutnog pobednika samo ako ga je imao i veći i u pitanju je isti apsolutni pobednik.

```
#include<iostream>
#include<vector>    // zbog koriscenja vektora
#include<algorithm> // zbog funkcije count

using namespace std;

void apsolutni_pobednik(vector<int> glasovi){
  // odredjujemo kandidata za pobednika (ako postoji) i broj
  // glasova koji preostane kada se poniste parovi razlicitih glasova

  int kandidatZaPobednika;
  int brojGlasovaZaKandidata = 0;

  for (int glas : glasovi)
    if (brojGlasovaZaKandidata == 0 ||
        glas == kandidatZaPobednika) {
      kandidatZaPobednika = glas;
      brojGlasovaZaKandidata++;
    } else
      brojGlasovaZaKandidata--;

  int n = glasovi.size();

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario vise od n/2 glasova
  if (brojGlasovaZaKandidata > 0 &&
      count(begin(glasovi), end(glasovi), kandidatZaPobednika) > n/2)
    cout << kandidatZaPobednika << endl;
  else
    cout << "nema" << endl;
}

int main() {

  vector<int> glasovi;
  int listic;

  cout << "Unesite glasove (pozitivne cele brojeve)";
  cout << "ili nulu kao oznaku za kraj: " << endl;

  do {
    cin >> listic;
    glasovi.push_back(listic);
  } while (listic);

  // izbacujemo nulu sa kraja
  glasovi.pop_back();

  apsolutni_pobednik(glasovi);

  return 0;
}
```

2 Dijametar binarnog drveta

Problem: Rastojanje između dva čvora binarnog drveta je broj grana na jedinstvenom putu koji ih povezuje. Dijametar drveta je najveće moguće rastojanje dva njegova čvora. Konstruisati efikasan algoritam za određivanje dijametra datog drveta i odrediti njegovu vremensku složenost.

Problem se jednostavno rešava korišćenjem induktivno-rekurzivne konstrukcije. Najduži put između dva čora ili prolazi ili ne prolazi kroz koren. Ako ne prolazi, onda se oba čvora nalaze ili u levom ili u desnom poddrvetu, pa se rastojanje rastojanje između njih može odrediti na osnovu induktivne hipoteze. Da bismo odredili najduži put koji prolazi kroz koreni čvor, potrebno je da znamo visinu levog i desnog poddrveta (visina je jednaka broju grana na najdaljem putu između korena i lista).

Prikazaćemo dva rešenja različite složenosti. U oba rešenja ćemo koristiti već poznate funkcije za rad sa stablima (koje su korišćenje u okviru predmeta Programiranje 2). Te funkcije su izdvojene u narednom delu koda.

```
#include<iostream>
#include<algorithm>

using namespace std;

typedef struct Cvor {
  Cvor(int n):vrednost(n){};

  int vrednost;
  Cvor *levo = nullptr;
  Cvor *desno = nullptr;
}cvor;

cvor* dodaj_u_stablo(cvor *koren, int n)
{
  if (koren == nullptr)
    return new cvor(n);

  if (n <= koren->vrednost)
    koren->levo = dodaj_u_stablo(koren->levo, n);
  else
    koren->desno = dodaj_u_stablo(koren->desno, n);

  return koren;
}

void oslobodi(cvor *koren)
{
  if(koren != nullptr) {
    oslobodi(koren->levo);
    oslobodi(koren->desno);
    delete koren;
  }
}


void LKD(Cvor *koren)
{
    if(koren != nullptr)
    {
        LKD(koren->levo);
        cout << koren->vrednost << " ";
        LKD(koren->desno);
    }
}
```

U prvom rešenju nezavisno implementiramo funkcije za računanje visine i za računanje dijametra. Kako je složenost funkcije *visina* $O(n)$, složenost funkcije *dijametar* je $O(n \log n)$ u slučaju balansiranog stabla, tj. $O(n^2)$ u opštem

slučaju.

```
int visina(cvor* koren) {
  if (koren == nullptr)
    return 0;
  return max(visina(koren->levo), visina(koren->desno)) + 1;
}

int dijametar(cvor* koren) {
  if (koren == nullptr)
    return 0;

  int dijametar_l = dijametar(koren->levo);
  int dijametar_d = dijametar(koren->desno);

  int visina_l = visina(koren->levo);
  int visina_d = visina(koren->desno);

  int dijametar_c = visina_l + visina_d;

  return max({dijametar_l, dijametar_c, dijametar_d});
}

int main() {

  cvor *koren = nullptr;

  int n;
  cout << "Unesite elemente drveta (0 za kraj): ";

  cin >> n;
  while(n!=0){
    koren = dodaj_u_stablo(koren,n);
    cin >> n;
  }

  LKD(koren);

  cout << "Dijametar je: " << dijametar(koren) << endl;

  return 0;

}
```

Kako je struktura, i redosled obilaska čvorova, ovih dveju funkcija ista, možemo ih spojiti u jednu funkciju koja će istovremeno računati i visinu i dijametar. Za prenos argumenata kroz rekurzivne pozive funkcije koristimo reference.

```
int visina_i_dijametar(cvor* koren, int& visina, int& dijametar) {

  if (koren == nullptr) {
    visina = 0;
    dijametar = 0;
    return 0;
  }

  int dijametar_l, dijametar_d;
  int visina_l, visina_d;

  visina_i_dijametar(koren->levo, visina_l, dijametar_l);
  visina_i_dijametar(koren->desno, visina_d, dijametar_d);

  int dijametar_c = visina_l + visina_d;

  visina = max(visina_l, visina_d) + 1;
  dijametar = max({dijametar_l, dijametar_c, dijametar_d});
}


int main() {

  cvor *koren = nullptr;
  int visina, dijametar;

  int n;
  cout << "Unesite elemente drveta (0 za kraj): ";

  cin >> n;
  while(n!=0){
    koren = dodaj_u_stablo(koren,n);
    cin >> n;
  }

  LKD(koren);

  visina_i_dijametar(koren, visina, dijametar);

  cout << "Dijametar je: " << dijametar << endl;

  return 0;

}
```

3 Broj rastućih segmenata

Problem: Dat je niz a celih brojeva. Definisati efikasan algoritam kojim se određuje koliko u tom nizu postoji rastućih segmenata (rastući segment čine uzastopni elementi niza $a_i < a_{i+1} < \ldots < a_j$, $0 \leq i \leq j \leq n$) i proceni mu složenost.

Problem se može rešiti najefikasnije ako posmatramo desni kraj segmenta i u svakom koraku pronalazimo broj rastućih segmenata koji se završavaju na poziciji 1, zatim koji se završavaju na poziciji 2, zatim na poziciji 3, itd., sve do pozicije $n - 1$ (na poziciji 0 se ne može završiti rastući segment, jer zahtevamo da su segmenti bar dvočlani).

Rešenje se zasniva na tome da broj segmenata koji se završavaju na poziciji j možemo odrediti veoma jednostavno inkrementalno, znajući broj segmenata koji se završavaju na poziciji $j - 1$. Naime, ako je $a_j \leq a_{j-1}$, segment na tom mestu počinje da opada, pa se na poziciji j ne završava ni jedan rastući segment. U suprotnom, svaki rastući segment koji se završavao na poziciji a_{j-1} može biti produžen elementom a_j, a rastući je i dvočlani segment $[a_{j-1}, a_j]$.

```
#include<iostream>
#include<vector>
using namespace std;

int prebroj(vector<int> a){

  int n = a.size();

  // ukupan broj rastucih serija
  int ukupanBrojRastucih = 0;
  // broj rastucih koji se zavrsavaju na tekucoj poziciji
  int brojRastucih = 0;
  for (int i = 1; i < n; i++) {
    if (a[i] > a[i-1]) {
      // tekuci element produzava sve rastuce segmente koji su
      // se zavrsili na prethodnoj poziciji i dodaje jos jedan
      // nov dvoclan rastuci segment
      brojRastucih++;
      // dodajemo broj rastucih koji se zavrsavaju na poziciji i na
      // ukupan broj rastucih segmenata
      ukupanBrojRastucih += brojRastucih;
    } else {
      // na tekucoj poziciji se ne zavrsava ni jedan rastuci segment
      brojRastucih = 0;
    }
  }

  return ukupanBrojRastucih;
}

int main() {

  vector<int> a;
  int x;

  cout << "Unesite elemente niza (prirodne brojeve)";
  cout << "ili nulu kao oznaku za kraj: " << endl;

  do {
    cin >> x;
    a.push_back(x);
  } while (x);

  // izbacujemo nulu sa kraja
  a.pop_back();

  cout << "Broj rastucih segmenata je: " << prebroj(a) << endl;

  return 0;

}
```

4 Maksimalna suma nesusednih elemenata pozitivnog niza

Problem: Napiši program koji određuje najveći zbir podniza datog niza nenegativnih brojeva koji ne sadrži dva uzastopna člana niza. Na primer, za niz $7, 3, 2, 4, 1, 5$ najveći takav podniz je $7, 4, 5$, čiji je zbir 16.

Niz elemenata ($[0, j]$) koji se završava na poziciji j možemo razložiti na poslednji element i prefiks niza bez tog poslednjeg elementa ($[0, j-1]$). Traženi maksimalni zbir (podniza koji ne sadrži dva uzastopna člana niza) može ili sadržati taj poslednji element, ili ne. Prema tome, uvešćemo dve promenljive, $max\_zbir\_sa(j)$ i $max\_zbir\_bez(j)$. Traženi maksimum će biti jednak većoj od te dve vrednosti.

Ako sadrži poslednji element, onda će maksimalni zbir biti jednak zbiru vrednosti poslednjeg elementa i maksimalnog zbira podniza do pozicije $j-2$ (pošto uključivanje elementa na poziciji j znači da ne sme biti uključen element na poziciji $j-1$), što je jednako $a[j] + max\_zbir\_bez(j-1)$.

Ako ne sadrži poslednji element, onda se maksimalni zbir nije promenio u odnosu na poziciju $j-1$ pa je jednak većem od vrednosti $max\_zbir\_bez(j-1)$ i $max\_zbir\_sa(j-1)$.

Inicijalizacija ovih promenljivih se vrši nad jednočlanim nizom. Maksimalni zbir sa uključenim njegovim poslednjim (jedinim) elementom jednak je tom elementu, dok je maksimalni zbir bez njega jednak nuli. Kada se petlja završi, veći od dva maksimalna zbira predstavlja traženi globalni maksimum.

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
  int n, x;

  cout << "Unesite broj elemenata niza: ";
  cin >> n;

  cout << "Unesite elemente niza: ";

  // prvi element ucitavamo van petlje, zbog inicijalizacije
  cin >> x;
  int maks_zbir_bez = 0;
  int maks_zbir_sa = x;

  for (int i = 1; i < n; i++) {
    cin >> x;

    int novi_maks_zbir_bez = max(maks_zbir_sa, maks_zbir_bez);
    maks_zbir_sa = maks_zbir_bez + x;
    maks_zbir_bez = novi_maks_zbir_bez;
  }
  cout << max(maks_zbir_bez, maks_zbir_sa) << endl;

  return 0;
}
```