Algoritmi i strukture podataka
5. Čas — Bibliotečke strukture podataka

Sana Stojanović Đurđević

November 5, 2018

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

1 Neki primeri iz jezika C++

Od ovog časa pa nadalje koristićemo redovno klase nabrojane u nastavku teksta. Prve tri klase (*pair*, *tuple*, *vector*) smo videli na prvom času, tako da ih ovde nećemo eksplicitno ponavljati.

1. `pair`
2. `tuple`
3. `vector`
4. `list`
5. `forward_list`
6. `stack`
7. `queue`
8. `deque`
9. `priority_queue`

2 Dinamički nizovi — najmanje odstupanje od proseka

Problem: Učitavaju se brojevi do kraja standardnog ulaza. Koji element najmanje odstupa od proseka?

Prilikom rešavanja ovog problema cele brojeve ćemo učitavati dok ima elemenata na standardnom ulazu. Prikazaćemo rešenje koje koristi klasu `string`,

preusmeravanje standardnog ulaza `cin` i funkciju `stoi`. Unos elemenata, pod Linux-om, se može prekinuti sa `CTRL+D`.

Ovaj program se prevodi uz opciju `-std=c++11`. U okviru ovog programa koriste se naredne funkcije:

stoi `http://www.cplusplus.com/reference/string/stoi/`

Funkcija koja konvertuje broj iz proizvoljne osnove koji je zapisan u promenljivoj tipa *string*, u njegovu celobrojnu vrednost. Ako se osnova u kojoj je broj zapisan ne navede, podrazumeva se da je osnova 10.

This feature is introduced by the latest revesion of the C++ standard (2011). Older compilers may not support it.

abs `http://www.cplusplus.com/reference/cmath/abs/`

Vraća apsolutnu vrednost broja x: `|x|`.

There are convenience `abs` overloads that are exclusive of C++. In C, `abs` is only declared in `<stdlib.h>` (and operates on `int` values). U C++-u moguće je da x bude `double`, `float`, `long double`, ili neki od integralnih tipova (`bool`, `char`, `int` — koji mogu biti `signed`, `unsigned`, `short`, `long` ili `long long`). `http://www.cplusplus.com/reference/type_traits/is_integral/`

Since C++11, additional overloads are provided in this header (`<cmath>`) for the integral types: These overloads effectively cast x to a double before calculations (defined for any integral type).

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

int main() {
  // ucitavamo sve elemente u vektor
  vector<int> a;
  string linija;

  cout << "Unesite niz celih brojeva: " << endl;
  cout << "(unos mozete prekinuti sa CTRL+D)" << endl;
  while (cin >> linija)
    a.push_back(stoi(linija));

  // izraunavamo prosek
  int n = a.size();
  int zbir = 0;
  for (int i = 0; i < n; i++)
    zbir += a[i];

  double prosek = (double) zbir / (double) n;
  cout << "Prosek je: " << prosek << endl;

  // odredjujemo i ispisujemo element koji najmanje
  // odstupa od proseka
  int min = 0;

  for (int i = 1; i < n; i++)
    if (abs(a[i] - prosek) < abs(a[min] - prosek))
      min = i;

  cout << "Element koji najmanje odstupa od proseka je: ";
  cout << a[min] << endl;

  return 0;
}
```

3 Stekovi

Stacks are a type of container adaptor, specifically designed to operate in a LIFO context (last-in first-out), where elements are inserted and extracted only from one end of the container.
`http://www.cplusplus.com/reference/stack/stack/`

3.1 Uparenost zagrada

Problem: Napisati program koji proverava da li su zagrade u infiksno zapisanom izrazu korektno uparene. Moguće je pojavljivanje malih, srednjih i velikih zagrada (tzv. običnih, uglastih i vitičastih).

Kada bismo radili samo sa jednom vrstom zagrada, dovoljno bi bilo samo održavati broj trenutno otvorenih zagrada. Pošto imamo više vrsta zagrada održavamo stek na kome pamtimo sve do sada otvorene zagrade (koje do tog trenutka nemaju svoj par-zatvorenu zagradu). Kada naiđemo na otvorenu zagradu, stavljamo je na stek. Kada naiđemo na zatvorenu zagradu, proveravamo da li se na vrhu steka nalazi njoj odgovarajuća otvorena zagrada i nju skidamo sa steka (ako je stek prazan ili ako se na vrhu nalazi neodgovarajuća otvorena zagrada, konstatujemo grešku). Na kraju proveravamo da li se stek ispraznio.

Zbog korišćenja *range-based* for petlje, ovaj program se prevodi uz opciju `-std=c++11`.

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

bool uparena(char oz, char zz) {
  return oz == '(' && zz == ')' ||
    oz == '[' && zz == ']' ||
    oz == '{' && zz == '}';
}

bool otvorena(char c) {
  return c == '(' || c == '[' || c == '{';
}

bool zatvorena(char c) {
  return c == ')' || c == ']' || c == '}';
}

int main() {
  string izraz;

  cout << "Unesite izraz u kome zelite da proverite ";
  cout << "da li su zagrade korektno uparene: " << endl;
  cin >> izraz;

  stack<char> zagrade;

  bool OK = true;
  for (char c : izraz) {
    if (otvorena(c))
      zagrade.push(c);
    else if (zatvorena(c)) {
      if (zagrade.empty() || !uparena(zagrade.top(), c)) {
        OK = false;
        break;
      }
      zagrade.pop();
    }

  }

  if (!zagrade.empty())
    OK = false;

  cout << (OK ? "Uparene su" : "Nisu uparene") << endl;
  return 0;
}
```

3.2 Vrednost postfiksnog izraza

Problem: Napisati program koji izračunava vrednost ispravnog postfiksno zadatog izraza koji sadrži samo jednocifrene brojeve i operatore + i * (bez razmaka između njih).
Na primer, za izraz 34+5* program treba da izračuna vrednost 35.

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}

int primeniOperator(char op, int op1, int op2) {
  int v;

  switch(op) {
  case '+': v = op1 + op2; break;
  case '*': v = op1 * op2; break;
  }

  return v;
}

int main() {
  string izraz;
  cin >> izraz;
  stack<int> st;

  for(char c : izraz){
    if (isdigit(c))
      st.push(c-'0');
    else if (jeOperator(c)) {
      int op2 = st.top();
      st.pop();
      int op1 = st.top();
      st.pop();
      st.push(primeniOperator(c,op1,op2));
    }
  }

  cout << st.top() << endl;
  return 0;
}
```

3.3 Prevođenje potpuno zagrađenog infiksnog izraza u postfiksni oblik

Problem: Dat je ispravan infiksni aritmetički izraz koji ima zagrade oko svake primene binarnog operatora. Napisati program koji ga prevodi u postfiksni oblik.
Na primer, `(((3*5)+(7+(2*1)))*4)` se prevodi u `35*721*++4*`. Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi i da se javljaju samo operacije sabiranja i množenja.

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}

int main() {
  string izraz;
  cin >> izraz;
  stack<char> operatori;

  //deo pod komentarima vraca rezultat u obliku niske
  //odkomentarisati kod ako zelite izlaz dobijen na taj nacin
  //string postfiksni;

  for(char c : izraz){
    if (isdigit(c)){
      cout << c;
      //postfiksni += c;
    }
    else if (jeOperator(c))
      operatori.push(c);
    else if (c == ')'){
      cout << operatori.top();
      //postfiksni += operatori.top();
      operatori.pop();
    }
  }

  cout << endl;
  //cout << postfiksni << endl;

  return 0;
}
```

3.4 Vrednost potpuno zagrađenog infiksnog izraza

Problem: Dat je ispravan infiksni aritmetiki izraz koji ima zagrade oko svake primene binarnog operatora. Na primer, `(((3*5)+(7+(2*1)))*4)`. Napisati program koji izračunava njegovu vrednost (za izraz u zagradama to je vrednost 96). Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi i da se javljaju samo operacije množenja i sabiranja.

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}

int primeniOperator(char op, int op1, int op2) {
  int v;

  switch(op) {
  case '+': v = op1 + op2; break;
  case '*': v = op1 * op2; break;
  }

  return v;
}


int main() {
  string izraz;
  cin >> izraz;

  stack<char> operatori;
  stack<int> vrednosti;

  for(char c : izraz){
    if (isdigit(c)){
      vrednosti.push(c-'0');
    }
    else if (jeOperator(c))
      operatori.push(c);
    else if (c == ')'){
      char op = operatori.top();
      operatori.pop();

      int op2 = vrednosti.top();
      vrednosti.pop();
      int op1 = vrednosti.top();
      vrednosti.pop();

      int v = primeniOperator(op, op1, op2);
      vrednosti.push(v);
    }
  }

  cout << vrednosti.top() << endl;

  return 0;
}
```

3.5 Prevođenje nezagrađenog infiksnog u postfiksni izraz

Problem: Napisati program koji vrši prevođenje ispravnog infiksno zadatog izraza u njemu ekvivalentan postfiksno zadati izraz. Pretpostaviti da su svi brojevi jednocifreni. Na primer, za izraz `2*3+4*(5+6)` program treba da ispiše `23*456+*+`.

```
#include <iostream>
#include <string>
#include <stack>
using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}
int prioritet(char c) {
  if (c == '+' || c == '-') return 1;
  if (c == '*' || c == '/') return 2;
}
void prevedi(const string& izraz) {
  stack<char> operatori;
  for (char c : izraz) {

    if (isdigit(c))
      cout << c;

    if (c == '(')
      operatori.push(c);

    if (c == ')') {
      // prebacujemo na izlaz sve operatore unutar zagrade
      while (operatori.top() != '(') {
        cout << operatori.top();
        operatori.pop();
      }
      // uklanjamo otvorenu zagradu
      operatori.pop();
    }

    if (jeOperator(c)) {
      // prebacujemo na izlaz sve prethodne operatore vieg prioriteta
      while (!operatori.empty() && jeOperator(operatori.top()) &&
             prioritet(operatori.top()) >= prioritet(c)) {
        cout << operatori.top();
        operatori.pop();
      }
      // stavljamo operator na stek
      operatori.push(c);
    }
  }

  // prebacujemo na izlaz sve preostale operatore
  while (!operatori.empty()) {
    cout << operatori.top();
    operatori.pop();
  }
  cout << endl;
}

int main() {
  string izraz;
  cin >> izraz;
  prevedi(izraz);
  return 0;
}
```

3.6 Vrednost infiksnog izraza

Problem: Napisati program koji izračunava vrednost ispravno zadatog infiksno zapisanog izraza koji sadrži operatore `+` i `*`. Jednostavnosti radi pretpostaviti da su svi operandi jednocifreni brojevi. Na primer, za izraz `(3+4)*5+6` program treba da ispiše 41.

```
#include <iostream>
#include <string>
#include <stack>

using namespace std;

bool jeOperator(char c) {
  return c == '+' || c == '*';
}

int prioritet(char c) {
  if (c == '+')
    return 1;
  if (c == '*')
    return 2;
}

// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujuci ih sa rezultatom primene te operacije
void primeni(stack<char>& operatori, stack<int>& vrednosti) {

  // operator se nalazi na vrhu steka operatora
  char op = operatori.top(); operatori.pop();

  // operandi se nalaze na vrhu steka operatora
  int op2 = vrednosti.top(); vrednosti.pop();
  int op1 = vrednosti.top(); vrednosti.pop();

  // izracunavamo vrednost izraza
  int v;
  if (op == '+') v = op1 + op2;
  if (op == '*') v = op1 * op2;

  // postavljamo ga na stek operatora
  vrednosti.push(v);
}
```

```
int vrednost(const string& izraz) {

  stack<int> vrednosti;
  stack<char> operatori;

  for (char c : izraz) {
    if (isdigit(c))
      vrednosti.push(c - '0');

    else if (c == '(')
      operatori.push('(');

    else if (c == ')') {
      // izraunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        primeni(operatori, vrednosti);

      //uklanjamo otvorenu zagradu
      operatori.pop();
    }
    else if (jeOperator(c)) {
      // obradjujemo sve prethodne operatore viseg prioriteta
      while (!operatori.empty() && jeOperator(operatori.top()) &&
             prioritet(operatori.top()) >= prioritet(c))
        primeni(operatori, vrednosti);

      operatori.push(c);
    }
  }

  // izraunavamo sve preostale operacije
  while (!operatori.empty())
    primeni(operatori, vrednosti);

  return vrednosti.top();
}

int main() {
  string izraz;
  cin >> izraz;

  cout << vrednost(izraz) << endl;

  return 0;
}
```

3.7 DFS nerekurzivno

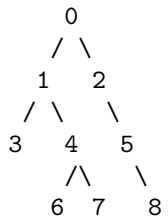
Problem: Implementirati nerekurzivnu funkciju koja vrši DFS obilazak drveta ili grafa (zadatog pomoću lista suseda).

Obilazak započinje iz proizvoljnog zadatog čvora r, korena pretrage u dubinu. Koren se označava kao posećen. Zatim se bira proizvoljni neoznačeni čvor r_1, susedan sa r, pa se iz čvora r_1 rekurzivno startuje pretraga u dubinu. Iz nekog nivoa rekurzije izlazi se kad se naiđe na čvor u kome su svi susedi (ako ih ima) već označeni. Ako su u trenutku završetka pretrage iz r_1, svi susedi čvora r označeni, onda se pretraga za čvor r završava. U protivnom, bira se sledeći proizvoljni neoznačeni sused r_2 čvora r, izvršava se pretraga polazeći od r_2, itd.

Poetak algoritma je u korenu stabla (kod grafa se neki čvor odredi za koren), a zatim se pretražuje duž svih grana koliko god je to moguće pre povratka u koren.

Uobičajena su dva načina predstavljanja grafa: matricom povezanosti (susedstva) i listom povezanosti (susedstva). Ovde ćemo koristiti liste povezanosti.

Stablo:



će biti predstavljeno na naredni način:

```
vector<vector<int>> susedi
{{1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};
```

Na poziciji 0 će se nalaziti susedi (naslednici) čvora označenog sa 0 (to su čvorovi 1 i 2); na poziciji 1 će se nalaziti susedi čvora 1 (to su 3 i 4); ...; čvorovi 6, 7 i 8 nemaju nove susede.
Struktura koja će biti korišćena je stek.

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

vector<vector<int>> susedi
{{1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};

void dfs(int cvor) {
  int brojCvorova = susedi.size();
  vector<bool> posecen(brojCvorova, false);

  stack<int> s;
  s.push(cvor);

  while (!s.empty()) {
    cvor = s.top();
    s.pop();

    if (!posecen[cvor]) {
      posecen[cvor] = true;
      cout << cvor << endl;

      for (int sused : susedi[cvor]) {
        if (!posecen[sused])
            s.push(sused);
      }
    }
  }
}

int main() {
  dfs(0);
  return 0;
}
```

3.8 Najbliži veći prethodnik

Problem: Za svaku poziciju i u nizu celih brojeva a pronaći poziciju $j < i$, takvu da je $a_j > a_i$ i da je j nabliža poziciji i (tj. od svih pozicija levo od i na kojima se nalaze elementi koji su strogo veći od a_i, j je najveća). Za svaku pronađenu poziciju j ispisati element a_j, a ako takva pozicija j ne postoji (ako su levo od i svi elementi manji ili jednaki a_i, tada ispisati karakter $-$). Takav element zvaćemo najbliži veći prethodnik.

Na primer, za niz `3 7 4 2 6 5`, ispisati `- - 7 4 7 6`. Voditi računa o prostornoj i vremenskoj efikasnosti.

Problem ćemo rešiti čuvanjem na steku kandidata za najbliže veće prethod-

nike. Naime, kada se nakon nekog elementa x pojavi neki element y koji je veći ili jednak njemu, tada element x prestaje da bude kandidat za najbližeg većeg prethodnika elemenata koji se javljaju u nizu iza y, pa ga možemo obrisati sa steka. Otuda, u svakom trenutku elementi koji su kandidati za najbliže manje predhodnike čine opadajui niz.
Prvi element u nizu potencijalnih kandidata koji je strogo veći od tekućeg elementa je ujedno njegov najbliži veći prethodnik. Ako takav element ne postoji, onda element nema većeg prethodnika.

Na steku možemo čuvati bilo pozicije ili vrednosti elemenata niza. U narednom programu čuvaćemo samo vrednosti.

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

int main() {
  int n;
  cin >> n;
  // stek na kojem cuvamo kandidate za najblize vece prethodnike
  stack<int> s;

  // obradujemo sve elemente
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;

    // skidamo sa steka elemente koji prestaju da budu kandidati
    // jer ih je tekuci element zaklonio
    while (!s.empty() && s.top() <= x)
      s.pop();

    if (s.empty())
      // ako na steku nema preostalih kandidata, tada
      // tekui element nema veih prethodnika
      cout << "-" << " ";

    else
      // element na vrhu steka je najblizi vei prethodnik
      // tekuem
      cout << s.top() << " ";

    // tekuci element postaje kandidat za najblizeg veceg
    // prethodnika narednim elementima
    s.push(x);
  }

  cout << endl;
  return 0;
}
```

3.9 Raspon akcija

Problem: Poznata je vrednost nekog proizvoda tokom n dana. Definišimo da je rok važenja akcija dana k (vrednosti a_k) najduži period prethodnih uzastopnih dana u kojima je vrednost akcija manja ili jednaka vrednosti u tom danu. Odredi rok važenja akcija za svaki dan.

Na primer, za niz `3, 7, 4, 2, 6, 5,` treba ispisati `1 2 1 1 3 1`. Konkretno, za vrednost 6 koja je na poziciji $k = 4$, prva prethodna vrednost koja je veća

od 6 je na poziciji 1 ($a_1 = 7$), to znači da tekuća akcija traje 3 dana.

Ovaj zadatak je veoma sličan prethodnom, jedino što za svaki element ne treba da znamo vrednost, već poziciju njemu najbližeg prethodika koji je strogo veći od njega (tj. 1 ako takav prethodnik ne postoji). Sve elemente ćemo na početku učitati u niz, a na steku ćemo pamtiti pozicije (a ne vrednosti) preostalih kandidata.

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

int main() {
  int n, i;
  cout << "Unesite broj dana: " << endl;
  cin >> n;
  vector<int> a(n);

  // stek na kojem cuvamo pozicije kandidata za najblize
  // vece prethodnike
  stack<int> s;

  cout << "Unesite vrednosti akcija: " << endl;
  // ucitavamo vrednosti akcija tokom n dana
  for (i = 0; i < n; i++)
    cin >> a[i];

  // obradjujemo sve elemente
  for (i = 0; i < n; i++) {

    // skidamo sa steka elemente, tj.pozicije, koji prestaju
    // da budu kandidati jer ih je tekuci element zaklonio
    while (!s.empty() && a[s.top()] <= a[i])
      s.pop();

    if (s.empty())
      // ako na steku nema preostalih kandidata, tada
      // tekui element nema veih prethodnika
      // pa je vreme trajanja akcije i+1
      cout << i+1 << " ";

    else
      // element na vrhu steka cuva poziciju najblizeg veceg
      // prethodnika da bi dobili broj dana trajanja akcije
      // oduzimamo ga od i
      cout << i - s.top() << " ";

    // pozicija tekuceg elementa postaje kandidat za najblizeg
    // veceg prethodnika narednim elementima
    s.push(i);
  }

  cout << endl;
  return 0;
}
```

3.10 Najbliži veći sledbenik

Problem: Za svaku poziciju i u nizu celih brojeva odrediti i ispisati poziciju njemu najbližeg sledbenika koji je strogo veći od njega, tj. najmanju od svih pozicija $j > i$ za koje važi $a_i < a_j$. Ako takva pozicija ne postoji (ako su svi elementi desno od pozicije i manji ili jednaki a_i), ispisati broj članova niza n. Pozicije se broje od nule.
Na primer, za niz `1 3 2 5 3 4 7 5` treba ispisati `1 3 3 6 5 6 8 8`.

Na steku (u rastućem redosledu) nalaziće se pozicije svih elemenata čiji najbliži veći sledbenik još nije određen tj. oni elementi iza kojih još nije pronađen neki veći element. Za svaki element koji obrađujemo, sa vrha steka skidamo pozicije onih elemenata koji su strogo manji od njega i beležimo da je traženi najbliži veći sledbenik za elemente na tim pozicijama upravo element koji trenutno obrađujemo. Na vrh steka se postavlja pozicija tekućeg elementa, jer ni njemu još nismo pronašli većeg sledbenika.

Pošto pozicije sledbenika ne saznajemo u redosledu u kojem ih je potrebno ispisati, moramo ih pamtiti u pomoćnom nizu koji ć e na kraju biti ispisan. Iz tog istog razloga, ovaj problem mora biti rešen čuvanjem pozicija elemenata na steku (a ne njihovih vrednosti).

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

int main() {
  // dimenzija niza
  int n;
  // stek na kome se nalaze elementi ciji najblizi veci sledbenik
  // jos nije pronadjen
  stack<int> s;

  cout << "Unesite broj elemenata niza: ";
  cin >> n;
  vector<int> a(n);
  // pozicije najblizih vecih sledbenika
  vector<int> p(n);

  cout << "Unesite elemente niza: ";
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // obilazimo sve elemente niza sleva nadesno
  for (int i = 0; i < n; i++) {

    // skidamo sa steka sve element kojima je najblizi
    // veci sledbenik element na poziciji i
    while (!s.empty() && a[s.top()] < a[i]) {
      // pamtimo da je element na poziciji i njihov najblizi veci
      // sledbenik
      p[s.top()] = i;
      s.pop();
    }
    // elementu na poziciji i jos uvek nismo odredili
    // poziciju najblizeg veceg sledbenika
    s.push(i);
  }

  // preostali elementi na steku nemaju vecih sledbenika
  while (!s.empty()) {
    p[s.top()] = n;
    s.pop();
  }

  // ispisujemo sve pozicije
  for (int i = 0; i < n; i++)
    cout << p[i] << " ";
  cout << endl;
  return 0;
}
```

3.11 Segmenti oivičeni maksimumima

Problem: Odrediti koliko u nizu koji sadrži sve različite elemente postoji segmenata (bar dvočlanih podnizova uzastopnih elemenata niza) u kojima su svi elementi unutar segmenta strogo manji od elemenata na njihovim krajevima.

Efikasno rešenje možemo dobiti tako što prilagodimo tehniku određivanja pozicija najbližih prethodnika strogo većih od datog elementa.

Za svaku poziciju j emo odrediti broj segmenata koji zadovoljavaju dati uslov, a kojima je desni kraj na poziciji j.

```
// Krecemo od resenja problema "Najblizi veci prethodnik"

#include <iostream>
#include <stack>
using namespace std;

int main() {
  int n;
  cin >> n;

  // ukupan broj segmenata oivicenih maksimumima
  int brojSegmenata = 0;

  // stek na kojem cuvamo kandidate za najblize vece prethodnike
  stack<int> s;

  // obradujemo sve elemente
  for (int j = 0; j < n; j++) {
    // ucitavamo j-ti element niza
    int x;
    cin >> x;

    // skidamo sa steka elemente koji prestaju da budu kandidati
    // jer ih je tekuci element zaklonio
    while (!s.empty() && s.top() < x) {
      // kako su elementi na steku u opadajucem poretku,
      // poslednji element na steku i tekuci element x odredjuju
      // granice segmenta
      brojSegmenata++;
      s.pop();
    }

    // ako ima jos elemenata na steku, poslednji element na
    // steku i tekuci element x odredjuju granice segmenta
    if (!s.empty())
      brojSegmenata++;

    // tekuci element postaje kandidat za najblizeg veceg
    // prethodnika narednim elementima, i granica za novi
    // skup segmenata
    // dodajemo ga na stek
    s.push(x);
  }

  cout << brojSegmenata << endl;
  return 0;
}
```

3.12 Najveći pravougaonik u histogramu

Problem: Niz brojeva predstavlja visine stubića u histogramu (svaki stubić je jedinične širine). Odredi površinu najvećeg pravougaonika u tom histogramu.

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

int main() {
  int n;
  cout << "Unesite broj stubica: ";
  cin >> n;
  vector<int> a(n);

  cout << "Unesite visine stubica: ";
  for (int i = 0; i < n; i++)
    cin >> a[i];

  int max = 0;
  stack<int> s;

  for (int d = 0; d < n || !s.empty(); d++) {
    // skidamo sa steka sve stubice kojima je najblizi manji
    // sledbenik na poziciji d
    while (!s.empty() && (d == n || a[s.top()] > a[d])) {
      int h = a[s.top()];
      s.pop();

      // skidamo sve stubice cija je visina jednaka h,
      // da bi nasli najblizi strogo manji prethodnik stubicu h
      while (!s.empty() && a[s.top()] == h)
        s.pop();

      // ako takvog stubica nema, stubic h nema manjih prethodnika i
      // pravougaonik koji mu odgovara moze se siriti do levog kraja
      int l = s.empty() ? -1 : s.top();

      // izracunavamo povrsinu tekuceg pravougaonika
      int P = (d - l - 1) * h;
      // i uporedjujemo je sa tekucim maksimumom
      if (P > max)
        max = P;
    }
    if (d < n)
      s.push(d);
  }
  cout << max << endl;
  return 0;
}
```

3.13 Red pomoću dva steka

Problem: Implementiraj funkcionalnost reda korišćenjem dva steka.

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

stack<int> ulazni, izlazni;

void prebaci() {
  while (!ulazni.empty()) {
    izlazni.push(ulazni.top());
    ulazni.pop();
  }
}

void push(int x) {
  ulazni.push(x);
}

void pop() {
  if (izlazni.empty())
    prebaci();
  izlazni.pop();
}

int top() {
  if (izlazni.empty())
    prebaci();
  return izlazni.top();
}

int main() {
  push(1);
  push(2);
  cout << top() << endl;
  pop();
  push(3);
  cout << top() << endl;
  pop();
  cout << top() << endl;
  pop();
  return 0;
}
```

4 Redovi

Queues are a type of container adaptor, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the container and extracted from the other.
`http://www.cplusplus.com/reference/queue/queue/`

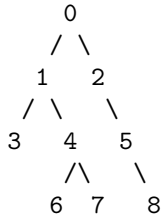
Podržane su sledeće metode:

- `push` - postavlja dati element na kraj reda
- `pop` - skida element sa početka reda
- `front` - očitava element na početku reda (pod pretpostavkom da red nije prazan)
- `back` - očitava element na kraju reda (pod pretpostavkom da red nije prazan)
- `empty` - proverava da li je red prazan
- `size` - vraća broj elemenata u redu

4.1 Nerekurzivni BFS

Problem: Implementiraj nerekurzivnu funkciju koja vrši BFS obilazak drveta ili grafa.

Stablo:



je predstavljeno pomoću liste suseda:

```
vector<vector<int>> susedi
{{1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};
```

Obilazak stabla DFS se koristi za agresivan pristup gde se ide što je moguće dalje u dubinu i radi se *backtracking* samo kad mora (primer je obilazak lavirinta). BFS se koristi kada je potrebno čvorove obići u nivoima. Dok je stek struktura podataka koja odgovara DFS algoritmu, struktura podataka koja odgovara BFS algoritmu je red.

Dovoljno je samo da u rešenju za DFS obilazak stabla zamenimo stek sa redom i dobićemo BFS obilazak stabla.

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

vector<vector<int>> susedi
{{1, 2}, {3, 4}, {5}, {}, {6, 7}, {8}, {}, {}, {}};

void bfs(int cvor) {
  int brojCvorova = susedi.size();
  vector<bool> posecen(brojCvorova, false);

  queue<int> s;
  s.push(cvor);

  while (!s.empty()) {
    cvor = s.front();
    s.pop();

    if (!posecen[cvor]) {
      posecen[cvor] = true;
      cout << cvor << endl;

      for (int sused : susedi[cvor]) {
        if (!posecen[sused])
          s.push(sused);
      }
    }
  }
}

int main() {
  bfs(0);
  return 0;
}
```

5 Red sa dva kraja

`Deque` (usually pronounced like "deck") is an irregular acronym of **d**ouble-**e**nded **que**ue. Double-ended queues are sequence containers with dynamic sizes that can be expanded or contracted on both ends (either its front or its back).
`http://www.cplusplus.com/reference/deque/deque/`

Specific libraries may implement deques in different ways, generally as some form of dynamic array. But in any case, they allow for the individual elements to be accessed directly through random access iterators, with storage handled automatically by expanding and contracting the container as needed.

Therefore, they provide a functionality similar to vectors, but with efficient insertion and deletion of elements also at the beginning of the sequence, and not only at its end. But, unlike vectors, deques are not guaranteed to store all its elements in contiguous storage locations: accessing elements in a deque by offsetting a pointer to another element causes undefined behavior.

Both vectors and deques provide a very similar interface and can be used for similar purposes, but internally both work in quite different ways: While vectors use a single array that needs to be occasionally reallocated for growth, the elements of a deque can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time and with a uniform sequential interface (through iterators). Therefore, deques are a little more complex internally than vectors, but this allows them to grow more efficiently under certain circumstances, especially with very long sequences, where reallocations become more expensive.

Podržava naredne operacije:

- `push_front` - dodavanje elemenata na početak reda,
- `push_back` - dodavanje elemenata na kraj reda,
- `pop_front` - uklanjanje elemenata sa početka reda,
- `pop_back` - uklanjanje elemenata sa kraja reda,
- `front` - očitava element na poetku reda (pod pretpostavkom da red nije prazan),
- `back` - oĭtava element na kraju reda (pod pretpostavkom da red nije prazan),
- `empty` - provera da li je red prazan,
- `size` - broj elemenata u redu.

For operations that involve frequent insertion or removals of elements at positions other than the beginning or the end, deques perform worse and have less consistent iterators and references than `lists` and `forward_lists`.
`http://www.cplusplus.com/reference/list/list/`
`http://www.cplusplus.com/reference/forward_list/forward_list/`
The main design difference between a `forward_list` container and a `list` container is that `forward_list` keeps internally only a link to the next element, while `list` keeps two links per element: one pointing to the next element and one to the preceding one, allowing efficient iteration in both directions, but consuming additional storage per element and with a slight higher time overhead inserting and removing elements. `forward_list` objects are thus more efficient than `list` objects, although they can only be iterated forwards.

6 Redovi sa prioritetom

Red sa prioritetom je vrsta reda u kome elementi imaju na neki način pridružen prioritet, dodaju se u red jedan po jedan, a uvek se iz reda uklanja onaj element koji ima najveći prioritet od svih elemenata u redu.

Priority queues are a type of container adaptors, specifically designed such that its first element is always the greatest of the elements it contains, according to some strict weak ordering criterion.
`http://www.cplusplus.com/reference/queue/priority_queue/`

Red sa prioritetom podržava sledeće metode:

- `push` - dodaje dati element u red
- `pop` - uklanja element sa najvećim prioritetom iz reda
- `top` - očitava element sa najvećim prioritetom (pod pretpostavkom da red nije prazan)
- `empty` - proverava da li je red prazan
- `size` - vraća broj elemenata u redu

6.1 Nekoliko načina kreiranja reda sa prioritetom

`http://www.cplusplus.com/reference/queue/priority_queue/priority_queue/`

6.2 Sortiranje pomoću reda sa prioritetom (HeapSort)

Problem: Napisati program koji sortira niz pomoću reda sa prioritetom.

U pitanju je tzv. algoritam sortiranja uz pomoć *hipa* tj. hip sort (engl. heap sort). Naziv dolazi od strukture podataka hip (engl. heap) koja se koristi za implementaciju reda sa prioritetom. U pitanju je varijacija algoritma sortiranja selekcijom (engl. selection sort) u kojem se, podsetimo se, u svakom koraku najmanji element dovodi na početak niza. Određivanje minimuma preostalih elemenata vrši se klasičnim algoritmom određivanja minimuma niza (tj. njegovog odgovarajućeg sufiksa) koji je linerne složenosti, što daje ukupnu složenost sortiranja $O(n^2)$. Algoritam hip sort koristi činjenicu da je određivanje i uklanjanje najmanjeg elementa iz reda sa prioritetom prilično efikasna operacija (obično je složenosti $O(\log k)$, gde je k broj elemenata u redu sa prioritetom). Stoga se sortiranje može realizovati tako što se svi elementi umetnu u red sa prioritetom, iz koga se zatim pronalazi i uklanja jedan po jedan najmanji element.

```
#include <iostream>
#include <queue>
#include <functional>
using namespace std;

int main() {
  // ovo je nacin da se u C++-u definise red sa prioritetom u kome
  // su elementi poredani u opadajucem redosledu prioriteta
  // (ovde, vrednosti)
  priority_queue<int, vector<int>, greater<int> > Q;

  // ucitavamo sve elemente niza i ubacujemo ih u red
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int ai;
    cin >> ai;
    Q.push(ai);
  }

  // vadimo jedan po jedan element iz reda i ispisujemo ga

  while (!Q.empty()) {
    cout << Q.top() << " ";
    Q.pop();
  }
  cout << endl;

  return 0;
}
```

6.3 k najmanjih učitanih brojeva

Problem: Napisati program koji omogućava određivanje k najmanjih od n učitanih brojeva unetih sa ulaza. Voditi računa o prostornoj i vremenskoj efikasnosti.

Potrebno je u svakom trenutku da u nekoj strukturi podataka održavamo k najmanjih do tada viđenih elemenata. U prvoj fazi, dok se ne učita prvih k elemenata svaki novi element samo ubacujemo u strukturu. Nakon toga svaki novi učitani element poredimo sa najvećim elementom u strukturi podataka i ako je manji od njega, taj najveći element izbacujemo, a novi element ubacujemo umesto njega. Dakle, potrebno je da imamo strukturu podataka u kojoj efikasno možemo da odredimo najveći element, da taj najveći element izbacimo i da u strukturu ubacimo proizvoljni element. Te operacije nam omogućava red sa prioritetom.

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

int main() {
  int k;
  cout << "Unesite broj k: ";
  cin >> k;
  priority_queue<int> Q;
  int n;
  cout << "Unesite broj n: ";
  cin >> n;

  cout << endl << "Unesite " << n << " brojeva: " << endl;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;

    if (Q.size() < k)
      Q.push(x);

    else if (Q.size() == k && x < Q.top()) {
      Q.pop();
      Q.push(x);
    }
  }

  while (!Q.empty()) {
    cout << Q.top() << " ";
    Q.pop();
  }
  cout << endl;
}
```

6.4 Medijane

Problem: Napisati program koji omogućava operaciju unošenja novog elementa u niz i određivanja medijane do tog trenutka unetih elemenata.

Veoma dobar način da se ovaj problem reši je da u svakom trenutku u jednoj (reći ćemo levoj) kolekciji čuvamo sve elemente koji su manji ili jednaki središnjem, a u drugoj (reći ćemo desnoj) sve one koji su veći ili jednaki središnjem, pri uslovu da ako postoji paran broj elemenata, te dve kolekcije treba da sadrže isti broj elemenata, a ako postoji neparan broj elemenata, desna kolekcija može da sadrži jedan element više. Ako ima neparan broj elemenata, tada je medijana jednaka najmanjem elementu desne kolekcije, a u suprotnom je jednaka aritmetičkoj sredini između najvećeg elementa leve i najmanjeg elementa desne kolekcije. Svaki novi element se poredi sa najmanjim elementom

desne kolekcije i ako je manji ili jednak njemu ubacuje se u levu kolekciju, a ako je veći od njega, ubacuje se u desnu kolekciju. Tada se proverava da li se sredina promenila. Ako se desilo da leva kolekcija ima više elemenata od desne (što ne dopuštamo), najveći element leve kolekcije treba da prebacimo u desnu. Ako se desilo da u desnoj kolekciji ima dva elementa više nego u levoj, tada najmanji element desne kolekcije prebacujemo u levu. Dakle, leva kolekcija treba da bude takva da lako možemo da pronađemo i izbacimo njen najveći element, a desna da bude takva da lako možemo da pronađemo i izbacimo njen najmanji element, pri čemu obe kolekcije moraju da podrže efikasno ubacivanje proizvoljnih elemenata. Te kolekcije mogu da budu redovi sa prioritetom u kojima se najmanja tj. najveća vrednost može očitati u konstantnom vremenu, ukloniti u logaritamskom, isto koliko je potrebno i da se umetne novi element.

```
#include <iostream>
#include <queue>
#include <string>
#include <vector>
#include <functional>

using namespace std;

priority_queue<int, vector<int>, greater<int> > veci_od_sredine;
priority_queue<int, vector<int>, less<int> > manji_od_sredine;

double medijana() {
  if (manji_od_sredine.size() == veci_od_sredine.size())
    return (manji_od_sredine.top() + veci_od_sredine.top()) / 2.0;
  else
    return veci_od_sredine.top();
}

void dodaj(int x) {
  if (veci_od_sredine.empty())
    veci_od_sredine.push(x);
  else {
    if (x <= veci_od_sredine.top())
      manji_od_sredine.push(x);
    else
      veci_od_sredine.push(x);
    if (manji_od_sredine.size() > veci_od_sredine.size()) {
      veci_od_sredine.push(manji_od_sredine.top());
      manji_od_sredine.pop();
    } else if (veci_od_sredine.size() > manji_od_sredine.size() + 1) {
      manji_od_sredine.push(veci_od_sredine.top());
      veci_od_sredine.pop();
    }
  }
}

int main() {

  while (true) {
    string s;
    if (!(cin >> s))
      break;
    if (s == "m")
      cout << medijana() << endl;
    else if (s == "d") {
      int x;
      cin >> x;
      dodaj(x);
    }
  }
  return 0;
}
```