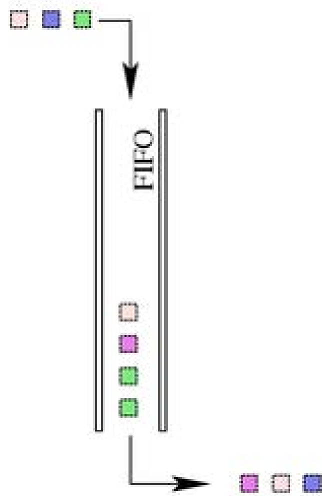
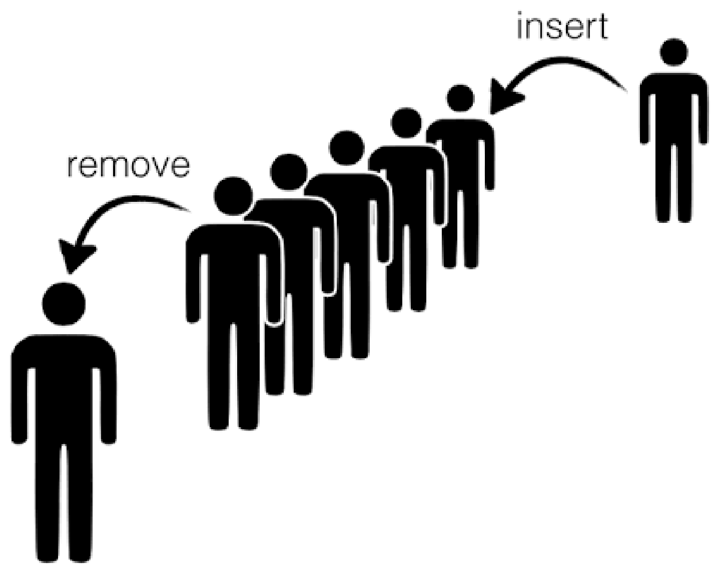
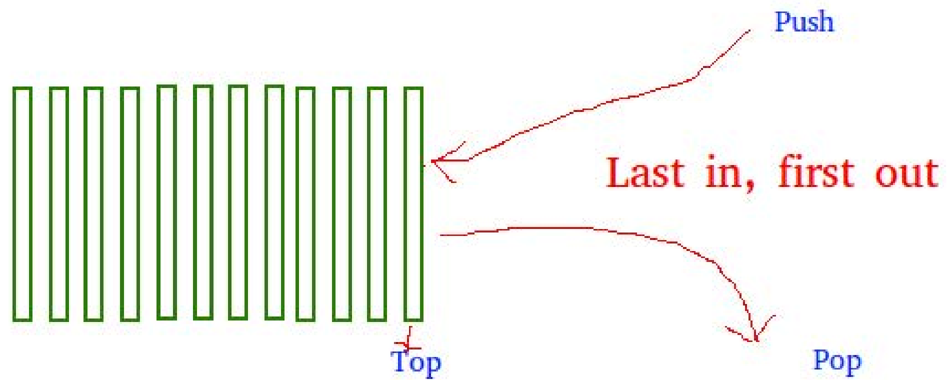
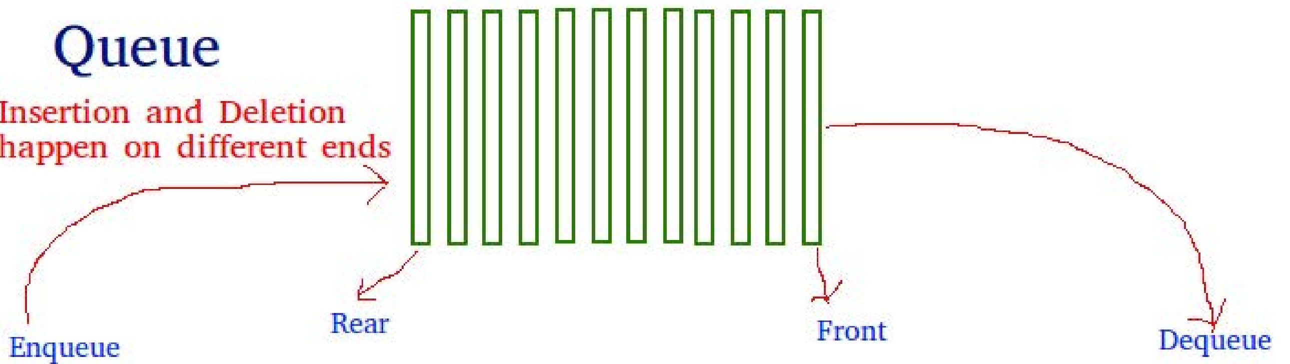
Uvod u C++ STL :: Standard Template Library

Dodatni resursi
http://www.sgi.com/tech/stl/
http://www.cplusplus.com/reference/stl/









Red

Da bi se koristila struktura reda mora se uključiti direktiva queue iz STL-a:

```
#include <queue>
```

Ako želimo da deklarišemo red za elemente tipa T, onda deklaracija glasi:

```
queue<T> q;
```

Dozvoljeni metodi

- Za proveru da li je red prazan, koristi se metoda empty

```
q.empty() //->bool
```

- Da bi se doznalo koliko ukupno elemenata je uneseno u red koristi se metod size

```
q.size() //-> int
```

- Da bi se postavio element t tipa T na kraj reda q, koristi se

```
q.push(t) //-> void
```

- Za čitanje i uklanjanje elementa sa početka reda q koristi se metod pop, kao u navedenom slučaju

```
q.pop() //->void
```

- Za čitanje, ali ne i uklanjanje poslednjeg elementa u redu koristi se metod back

```
q.back() //-> T
```

- Za čitanje, ali ne i uklanjanje prvog elementa (prednji deo reda) koristi se metoda front

```
q.front() //-> T
```

Primer 01:

```
//direktive za ukljucivanje potrebnih klasa
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
//deklarisanje reda ciji elementi su tipa string
queue<string> q;
//Unos tri stringa u red

  q.push("PROBA 1");
  q.push("PROBA 2");
  q.push("PROBA 3");


string w;
cin>>w;
q.push(w);
//Stampanje broja clanova reda
cout <<"Broj clanova reda = "<< q.size()<< endl;
//Dokle god red nije prazan
while (!q.empty()) {
//Citanje stringa sa vrha reda i stampanje, FIFO
cout << q.front() << endl;
//Citanje i uklanjanje stringa sa vrha reda
q.pop();
}
return 0;
}
```

Napomena

Važno je naglasiti da funkcija top (u strukturi stack), funkcije front i back (u strukturi `queue`) vraćaju referencu na odgovarajući element u strukturi podataka. Dakle, ne vraćaju pokazivač, niti kopiju elementa, već referencu. To

znači da preko reference koja se vraća, možete da pozovete bilo koji metod ili da izvršite bilo koju od dozvoljenih operacija. Na ovoj način, direktno ćete izmeniti element, unutar strukture podataka, samo zato što ovi metodi vraćaju konstantnu referencu.

Primer:

```
s.top()=t1;//izmena elementa na vrhu steka
q.back()=t2;//izmena elementa na kraju reda
q.front()=t3;//izmena elementa na pocetku reda
```

U biblioteci queue definisana je jos jedna struktura podataka, prioritetni red tj. priority_queue.

Primer 02

```
//prioritetni red
#include <iostream>
#include <queue>
using namespace std;

int main()
{ priority_queue <int> q;
  q.push(42); q.push(2); q.push(24);
  q.push(22); q.push(44);
    cout << "Broj clanova je: " << q.size() <<endl;
  for(;!q.empty();q.pop())
  cout << q.top() << ' ';
    return 0;
}
```

Klasa priority_queue ima metode kao i klasa queue, uz to da se metoda front() zove top().

U *priority_queue* se moze cuvati struktura podataka *max-heap* tako da uzimanjem elementa sa pocetka uvek dobijamo najveci element. Vremenska složenost metode top() je i dalje O(1), dok O(log n) je vremenska složenost metode push(x) i pop(). Ako smestimo niz elemenata u priority_queue, i uzimamo elemente redom, onda smo uspeli sortirati niz u vremenu O(n log n).

Želimo li raditi sa min hipom, možemo napraviti korisnički metod za poređenje (jednostavna zamena operatorom >).

Pri definisanju minHipa koristimo konstruktor koji zahteva 2 dodatna argumenta:

priority_queue<int,vector<int>, poredi > minHip;

Dakle, u tom konstruktoru, osim tipa podataka koji cemo cuvati u minHipu, moramo navesti i nad kojim kontejnerom se izgradjuje minHip(najbolje staviti vector<int>), kao i ime strukture ili klase koja ima preoptrecen operator() radi poredjenja.

Primer 03

```
//min heap
#include <iostream>
#include <queue>
 using namespace std;

 struct poredi
 {
   bool operator()(const int& l, const int& r)
   {
       return l > r;
   }
 };

 int main()
 {
     priority_queue<int,vector<int>, poredi > minHip;

     minHip.push(3);
```

```
    minHip.push(5);
    minHip.push(1);
    minHip.push(8);
    while ( !minHip.empty() )
    {
        cout << minHip.top() << endl;
        minHip.pop();
    }
   return 0;
}
```

IZLAZ

1
3
5
8

REŠENJE BROJ 2

```
#include <bits/stdc++.h>
using namespace std;
int main ()
{  // Kreiranje  min heap
   priority_queue <int, vector<int>, greater<int> > pq;
   pq.push(3);
   pq.push(5);
   pq.push(1);
   pq.push(8);
   // uzimanje vrha min heap-a
   while (pq.empty() == false)
   {   cout << pq.top() << " ";
      pq.pop();
   }
   return 0;
}
```

Liste

Da bi se koristile liste u STL-u kao strukture podataka, mora da se doda sledeća direktiva na početku programa:

```
#include <list>
```

Da bi se deklarisala lista određenog tipa T koristimo sledeću liniju programa:

```
list<T> q;
```

Dozvoljeni metodi

- Da bi se proverilo da li je lista prazna, koristi se metoda empty, kao na primer

```
ls.empty() //-> bool
```

- Da bi se saznao tekući broj elementi u listi, koristi se metod size

```
ls.size() //-> int
```

- Za dodavanje elementa t tipa T na kraj liste ls, koristi se naredba

```
ls.push_back(t) //-> void
```

- Za čitanje i uklanjanje elementa sa kraja liste ls, koristi se metod pop_back

```
ls.pop_back() //-> void
```

- Analogno, postoji sličan metod za dodavanje i uklanjanje elementa na početak liste ls.

```
ls.push_front(t) //-> void
ls.pop_front() //-> void
```

- Za sortiranje liste se koristi metod sort, kao u sledećem primeru:

```
ls.sort() //-> void
```

- Za brisanje svih elemenata liste se koristi metod clear

```
ls.clear() //-> void
```

- Za “obrtanje“ liste se koristi metod reverse

```
ls.reverse() //-> void
```

- Za dodeljivanje kopije liste ls1 drugoj listi ls2, se koristi naredba

```
ls2=ls1;
```

Klasa list iz STL-a definisana je u biblioteci (zaglavlju) list i predstavlja impementaciju dvostruko povezane liste. Svaki element u takvoj listi pokazuje na element ispred i iza sebe i na taj nacin omogucuje ubacivanje i izbacivanje elemenata u vremenskoj slozenosti O(1).
Dohvatanje n-tog elementa u takvoj listi ima vremensku slozenost O(n).
POREDJENJE SA kontejnerom vector: Vremenska slozenost izbacivanja u vector je O(n), ali dohvatanje n-tog elementa slozenosti O(1).

Primer 01: uvod u rad sa listom

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

int main()
```

```
{ list<int> l;
  l.push_back(3);
  l.push_front(1);
  l.insert(++l.begin(),2);
  list<int>::iterator it;
  for(it=l.begin();it!=l.end();it++)
   cout << * it << ' ';
    return 0;
}
```

IZLAZ
1 2 3

U 6. liniji
```
  list<int> l;
```
kreiramo praznu listu celih brojeva koju nazovemo l.

Mogli smo kreirati i listu sa 10 praznih elemenata na sledeci nacin (slicno kao kod vektora)
```
list<int> l(10);
```

Mogli smo kreirati i listu sa 5 elemenata jednakih 7 na sledeci nacin (slicno kao kod vektora)
```
list<int> l(5,7);
```

U 7. liniji
```
 l.push_back(3);
```
dodaje se 3 na kraj liste. Vremenska slozenost ove operacije je O(1).

U 8. liniji
```
 l.push_front(1);
```
dodaje se 1 na pocetak liste. Vremenska slozenost ove operacije je O(1).

Takodje, kao i za vektor postoje metodi pop_front(), pop_back(). Postoje i metodi front() i back() koje u vremenskoj slozenosti O(1) vracaju prvi i poslednji element liste.
Pomocu navedenih metoda se lista moze koristiti kao queue ili stack sa neogranicenim kapacitetom. O kapacitetu se brine lista.

U 9. liniji
```
  l.insert(++l.begin(),2);
```
je ubacen broj 2 ispred 2. elementa.

Naredbu l.insert(++l.begin(),2);
nismo smeli napisati kao
```
  l.insert(l.begin()+1,2);
```
jer iteratori liste nisu isti kao i iteratori vektora (slabiji su iteratori liste) i nemaju preopterecen operator sabiranja i oduzimanja, nego samo operatore ++ i --. To je i logicno, jer bi se operatori + i – morali izvoditi u slozenosti O(n).

Metod ***insert(iterator, vrednost)*** se izvrsava u vremenskoj slozenosti O(1), za razliku od vektora koji tu operaciju obavlja u vremenskoj slozenosti O(n). Lista takodje ima metod ***insert*** i ***erase*** kao i vektor, ali su kod liste brži.

Liste se mogu, kao i vektori, uporedjivati u slozenosti O(n) pomocu operatora == != <= < > >=, te se takodje mogu pridruzivati u O(n).

Liste, za razliku od vektora, nemaju preopterecen operator [] i nijma **se ne moze pristupati kao** da se radi o nizu.

Zato 10. liniju programskog koda NISMO mogli ispisati u obliku
```
for (int i=0;i<l.size();i++)   cout << l[i] << ' ';
```

Ispis liste smo obavili u 10. i 11. liniji programskog koda u formi

```
  for(it=l.begin();it!=l.end();it++)
   cout << * it << ' ';
```

Koristili smo iteratore koji su inicijalizovani kao it=l.begin().
U 11. liniji pristupamo pojedinacnom elementu sa *it.

Liste (kao i vektori) mogu sadrzavati bilo sta, tako da je moguce definisati:

```
 list<vector<list<string> > > listaVektoraSaListomStringova;
```

Metod erase moze izazvati problem ako se ne koristi pazljivo. Ako napisemo naredbu

```
  l.erase(it);
```

nakon toga ne smemo dalje koristiti iterator **it**, jer postaje nevazeci (tj. mora se iznova izracunati). To, takodje, vazi i za vektore. Razlog je što brišući element, gubimo element, te ne možemo iz njega pročitati koji element je sledeći.
Želimo li nakon brisanja elementa imati i dalje važeći iterator, onda to možemo učiniti ovako:

```
  it=l.erase(it);
```

Metod *erase* vraca iterator na element nakon obrisanog.
Takodje, mozemo napisati

```
   l.erase(it++);
```

To je moguce zbog nacina na koji je operator ++ preopterecen.

Ako napisemo

```
  *it=2;
  l.insert(it,1);
```

onda se element 1 ubacuje pre elementa 2, a iterator *it* ce i dalje pokazivati na element s vrednoscu 2.

Lista takodje ima metodu *swap* kao i vektor, koja je O(1), kao i metodi *clear* i *empty*.

Lista ima dodatni metod *reverse* koji okrene listu u O(n) i metod *sort* koji sortira listu u slozenosti O(n log n).

Primer 02: metodi za rad sa listom

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

void ispis (list<int> &li)
{ for(list<int>::iterator x=li.begin();x!=li.end();x++)
    cout << *x << ' ';
    cout << endl;
}

int main()
{  list<int> l;
   l.push_back(3); l.push_back(2);
   l.push_back(4); l.push_back(1);
   ispis(l);
   l.reverse();
   ispis(l);
   l.sort();
   ispis(l);
   l.reverse();
   ispis(l);
    return 0;
}
```

U 5. liniji ne mozemo pisati

```
const list<int> &li
```

umesto
list<int> &li

Vektori (prvo-klasni niz)

Vektor je osnovni kontejner iz STL-a i zamenjuje običan niz. Kad mu je potreban dodatni prostor u memoriji, vektor se sam povećava, tako sto se ceo vektor prekopira na dvostruko veci prostor. Buduci da se vektor svaki put povecava za dvostruko, ukupno se povecava log n puta. Time je svako novo kopiranje veće od svih ranije kreiranih memorijskih polja vektora, tako da je ukupno vreme potrebno za sva prethodna kopiranja jednako veličini vektora. **Dakle, ne moramo se uopšte brinuti o veličini vektora, jer se STL sam o tome brine.**

Možemo reći da se ubacivanje na kraj vektora odvija konstantno (O(1))

Pristupanje svakom elementu vektora je konstantno, kao i izmena nekog elementa vektora.

Vremenska složenost funkcije size() je O(1), jer ***vector*** pamti svoju trenutnu veličinu.
http://www.cplusplus.com/reference/vector/vector/size/

Ipak, ubacivanje na početak ili sredinu vektora je linearno tj. O(n).

Vektor je vrlo jednostavan kontejner (klasa koja služi za čuvanje podataka). Nalazi se u biblioteci (zaglavlju) **vector**.

Vektori se razlikuju od liste zato što postoji mogućnost za direktan pristup elementu vektora preko indeksa, t.j. operatora [].
Da bi mogla da se koristi ova struktura potrebno je u programu uključiti direktivu:
```
#include <vector>
```
Da bi deklarisali vektor određenog tipa T , naredba je:
```
vector<T> v;
```

Dozvoljeni metod

- Za proveru da li je vektor v prazan, koristi se metod empty
```
v.empty() //-> bool
```
- Da biste saznali trenutni broj elemenata unutar vektora v koristi se metod size
```
v.size() //-> int
```
- Da biste dodali novi element t tipa T na kraj vektora v, koristi se metod push_back
```
v.push_back(t) //-> void
```
- Ako želite da pročitate i uklonite element sa kraja vektora v, koristi se metod pop_back
```
v.pop_back() //-> T
```
-Za pristup (konstantna referenca) prvom elementu na početku vektora, koristi se metod front
```
v.front() //-> T
```
- Za pristup (konstantna referenca) poslednjem elementu na kraju vektora v koristi se metod back
```
v.back() //-> T
```
- Za direktan pristup i-tom elementu vektora v (0<= i<v.size()) bez provere da li element odnosno indeks postoji, koristi se baš isti način kao i kod običnog niza:
```
v[i]
```
- Za siguran pristup i-tom elementu koristi se metod at, kao što je pokazano u primeru
```
v.at(i)// -> referenca
```
- Za dodeljivanje kopije vektora v1 vektoru v2 koristi se naredba dodele
```
v2=v1;
```

Primer 01:
```
#include <iostream>
#include <vector>
using namespace std;
int main(){
//Inicijalizujmo vektor sa 4-ri celobrojne vrednosti
```

```
vector<int> v(4);
//Inicijalizujmo vrednosti
v[0]=100;
v[1]=200;
v[2]=300;
v[3]=400;
//metod push_back pravi proširivanje vektora za element
//koji je argument metoda
v.push_back(10);
v.push_back(20);
v.push_back(30);
v.push_back(40);
v.push_back(50);
for(int i=0;i<v.size();i++)
cout<<i<<" - ti element: "<<v[i]<<endl;
return 0;
}
```

Primer 02

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

int main()
{   vector <int> v;
   v.push_back(3); v.push_back(5);
   v.push_back(7);
   for(int i=10;i<14;i++)
     v.push_back(i);
   v[0]--;
   v[5]=-200;
   for(unsigned i=0;i<v.size();i++)
    cout << v[i] << ' ';
   cout << endl;
   return 0;
}
```

IZLAZ

2 5 7 10 11 -200 13

U ovom zadatku smo koristili metod push_back() koji dodaje argument metoda na kraj vektora.
U 11. liniji programskog koda, pristupamo vektoru preko preopterećenog (overload) operatora [], kao da radimo sa nizovima (v[0]--;).
U 15. liniji programskog koda, koristimo metod size() koji vraća veličinu vektora.

Primer 03: vektor može čuvati bilo koji tip podataka

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

int main()
{   vector <string> vs;
   vs.push_back("ja"); vs.push_back("volim");
   vs.push_back("programiranje");
```

```
    for(unsigned i=0;i<vs.size();i++)
     cout << vs[i] << ' ';
    cout << endl;

    vector < vector<int> > matrica;
    for(int i=0;i<5;i++)
    {
       vector <int> v;
       matrica.push_back(v);
       for (int j=0;j<=i;j++)
       matrica[i].push_back(j);
    }

    for(unsigned i=0;i<matrica.size();i++)
    {
       for(unsigned j=0;j<matrica[i].size();j++)
        cout << matrica[i][j] << ' ';
       cout << endl;
    }


    return 0;
}
```

U 7.liniji programskog koda

```
   vector <string> vs;
```

kreiran je prazan vektor vs koji u sebi čuva stringove.

U 8. i 9. liniji smo u vektor ubacili 3 stringa koristeći metod push_back. Stringove smo ubacili na kraj vektora i time povećali veličinu vektora.

U 15. liniji definisali smo vektor koji u sebi sadrži vektor celih brojeva.

```
    vector < vector<int> > matrica;
```

Na taj način smo kreirali strukturu sličnu dvodimenzionom nizu celih brojeva.
Pri deklaraciji vektora matrica, morali smo voditi računa da ne izostavimo blanko karakter između karaktera > >, jer bi deklaracija

```
 vector < vector<int>> matrica;
```

izazvala sintaksnu grešku.

U 27. liniji ispisujemo vrednost vektora pristupajući im pomoću operatora []. Moramo voditi računa da operator [] ne koristimo za mesta u vektoru koja nisu zauzeta (zauzeta su ona koja smo ubacili pomoću metoda push_back!!!).

Ako bismo iz našeg programskog koda izbacili 18. i 19. liniju programskog koda

```
 vector <int> v;
 matrica.push_back(v);
```

programski kod bi se uspešno kompajlirao, ali bi se verovatno srušio prilikom izvršavanja. Na takve greške moramo sami pazitii često ih je teško uočiti.

IZLAZ

```
ja volim programiranje
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

Primer 04: vektor i iteratori

U STL-u postoje iteratori kako bi se pomoću njih moglo iterirati kroz neke strukture podataka u kojima se ne može definisati koji je element po redu, nego samo koji je element pre ili nakon nekog elementa. U vektorima se može direktno pristupiti nekom elementu što nije slučaj sa svim strukturama podataka koje ćemo kasnije obraditi.

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

int main()
{   vector <int> v;
    for(unsigned i=0;i<15;i++)
     v.push_back(i);
    vector<int>::iterator it;
    for(it=v.begin(); it!=v.end();it++)
     cout << *it << ' ';
    cout << endl;
    return 0;
}
```

IZLAZ
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

U 10. liniji programskog koda kroz vektor v umesto brojačem, iteriramo iteratorom it. Deklaracija iteratora it je
vector<int>::iterator it;
gde iterartor je naziv tipa, a koristimo notaciju :: da odredimo kojoj klasi pripada.
U 10. liniji inicijalizujemo iterator na početak vektora, tj. dodeljujuemo it=v.begin();

Metod begin() vraća iterator na 1. element vektora, a metod end() vraća iterator na element iza poslednjeg.
Pomoću operatora ++ menjamo iterator tako da pokazuje na sledeći element.
Pomoću operatora != gledamo da li je it različit od nekog iteratora.

U 11. liniji programskog koda ispisujemo elemente vektora pomoću iteratora koristeći preopterećen operator * kako bismo pristupili elementu na koji iterator pokazuje.

Primer 05: još neki metodi klase vector

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

int main()
{   vector <int> v(10); //v.size() je 10
    v[0]=-1; v[9]=99;
    v[4]=v[5]=100;
    v.push_back(2345);
    for(unsigned i=0;i<v.size();i++)
    cout << v[i] << ' ';
    v.pop_back();
    cout << v.front() << ' ' << v.back() << endl;
    v.clear(); //sve brise i size postaju 0
   if (v.empty()) cout << "v je prazan\n";
    return 0;
}
```

U 6. liniji programskog koda stvaramo vector<int> koji ima 10 elemenata postavljenih na 0. Time v.size() vraca 10, a v.push_back(nesto); ubacuje na lokaciji v[10].
U 12. liniji metoda pop_back izbacuje poslednji element (tj. v[v.size()-1]) iz vektora u vremenskoj slozenosti O(1) i smanjuje v.size() za 1.
U 13. liniji programskog koda v.front() vraca prvi element vektora, tj. v[0], dok v.back() vraca poslednje element vektora tj. v[v.size()-1].
U 14.liniji pozivamo metod v.clear(); da bi se obrisali svi elementi vektora v i onda ce v.size() postati 0.
U 15. liniji pozivamo metod v.empty() koji vrati true ako je v.size()==0

Primer 06: još neki metodi klase vector

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

inline void ispisi(const vector<int>& x)
{  for(int i=0;i<x.size();i++) cout << x[i] << ' ';
   cout << endl;
}

int main()
{   vector <int> v,q;
    v.insert(v.end(),7,1);
    v.insert(v.begin()+2,3);
    ispisi(v);
    q.insert(q.begin(),10,8);
    q.insert(q.end(),9);
    v.insert(v.begin()+4, q.end()-3, q.end());
    ispisi(v);ispisi(q);
    v.erase(v.begin()+2);
    v.erase(v.begin()+3,v.begin()+6);
    ispisi(v);
    vector <int> ve(3,4);
    ispisi(ve);
    v.swap(ve);
    ispisi(v);ispisi(ve);
    return 0;
}
```

IZLAZ

```
1 1 3 1 1 1 1 1
1 1 3 1 8 8 9 1 1 1 1
8 8 8 8 8 8 8 8 8 8 9
1 1 1 1 1 1 1
4 4 4
4 4 4
1 1 1 1 1 1 1
```

Napomene:

U 5. liniji programskog koda implementirana je funkcija koja ispisuje sadrzaj vektora. Kako bismo ubrzali rad funkcije, opredelili smo se da kao argument funkcije primami referencu na vektor i zbog toga ga ne kopiramo. Kako bismo se zastitili od slucajne izmene vektora, argument je konstantna referenca.

U 12. liniji programskog koda

```
v.insert(v.end(),7,1);
```
dodajemo sedam 1ca ispred iteratora v.end(), gde v.end() pokazuje ne element iza poslednjeg.

U 13.liniji programskog koda dodajemo broj 3
```
v.insert(v.begin()+2,3);
```
dodajemo broj 3 ispred iteratora v.begin()+2. To je iterator na 3. element vektora. Metod insert i erase imaju slozenost O(n).

Iteratori nisu jednaki za sve kontejnere iz STL-a, tako da iterator na vektor moze imati vecu funkcionalnost nego iterator na neki drugi kontejner.
Iterator na vektor moze se opteretiti + ili – ili pomeriti nekoliko elemenata unapred ili unazad (jer vektor interno podatke cuva u obliku niza), dok neki drugi iteratori to nece moci.

U 15. liniji
```
q.insert(q.begin(),10,8);
```
dodajemo 10 puta broj 8 na pocetak vektora q.

16. linija
```
q.insert(q.end(),9);
```
moze da se napise i kao
```
q.push_back(9)
```

17. linija
```
v.insert(v.begin()+4, q.end()-3, q.end());
```
sadrzi metod insert koji prima tri iteratora.
insert(it, begin, end) dodaje sve elemente između iteratora begin, end i to ispred iterarora it.

U 19. liniji
```
v.erase(v.begin()+2);
```
brišemo 3. element vektora v.

U 20. liniji
```
v.erase(v.begin()+3,v.begin()+6);
```
brišemo 4., 5., 6. element vektora v.

U 22. liniji
```
vector <int> ve(3,4);
```
kreiramo vektor ve koji sadrži tri četvorke.

Vektori se u lineranoj složenosti mogu pridruživati jedni drugima.
Na primer.
```
vector <int> s;
 s=ve;
 ispisi(s);
```

Vektori se u lineranoj složenosti mogu upoređivati operatorima == != <= < >= >.

U 24. liniji
```
v.swap(ve);
```
zamenjuju se vektori v i ve u vremenskoj slozenosti O(1).

Nakon trampe vektora metodom swap, svi iteratori koji pokazu na bilo koji od dva zamenjena vektora postaju nevažeći (moraju se ponovo izracunati).

Kontejneri i iteratori

Kontejner je svaka struktura podataka koja sadrži elemente istog tipa. Liste, vektori, stek i red su kontejneri.

Elementima unutar kontejnera se pristupa preko iteratora. Tako, na primer, za deklarisanje iteratora za vektor sa celobrojnim vrednostima se koristi izraz

```
vecor<int>::iterator
```

Iteratori su naročito važni za liste budući da ne postoji način za direktan pristup elementima liste. Važno je da se napomene da su iteratori zapravo pokazivači na elemente kontejnera.
Dakle, neka je deklarisana lista ls za elemente tipa int.

```
list<int> ls;
```

Radi obilaska liste, može se elementima liste pristupiti preko iteratora, odnosno preko pokazivača.

```
list<int>::iterator p;
for(p=ls.begin();p!=ls.end();p++)
cout<<*p<<endl;
```

- Deklaracija iteratora za kontejnera tipa C

```
C::iterator p
```

- Pomicanje iteratora na sledeći element kontejnera se vrši u formi

```
p++
```

- Sam iterator p je pokazivač. Vrednost na koju pokazuje p je

```
*p
```

- Za postavljanje iteratora da pokazuje na prvi element kontejnera C koristi se metod

```
c.begin()
```

- Za postavljanje iteratora da pokazuje na poslednji element kontejnera C koristi se metod

```
c.end()
```

Liste i iteratori

Kada je reč o listi (u smislu kontejnera i iteratora), postoje metode koje dozvoljavaju umetanje elemenata u listu i uklanjanje elemenata iz liste.
Dodavanje element x u listu ls ispred iteratora p se vrši metodom insert:

```
ls.insert(p,x);
```

Brisanje element iz liste ls, na koju pokazuje iterator p se vrši metodom erase:

```
ls.erase(p);
```

Primer:

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

void print(list<char> ls){
for(list<char>::iterator p=ls.begin();p!=ls.end();p++)
cout<<*p<<" ";
cout<<endl;
}

int main(){
list<char> ls;
list <char>::iterator p;
ls.push_back('o');
ls.push_back('a');
ls.push_back('t');
p=ls.begin();
// p pokazuje na 'o' unutar liste ('o', 'a', 't')
cout <<" "<< *p<<endl;
print(ls);
ls.insert(p, 'c');
// ls sada izgleda ('c', 'o', 'a', 't'), p još pokazuje na 'o'
cout <<" "<< *p<<endl;
print(ls);
ls.erase(p);
// p pokazuje na 'o' koje više nije u listi ls
cout <<" "<< *p<<endl;
```

```
print(ls);
//za uklanjanje prvog elementa iz liste ls
ls.erase(ls.begin());
print(ls);
return 0;
}
```

Napomena

Uključivanjem direktive
`#include<algorithm>`
i upotrebom iteratora i vektora može se efikasno sortirati niz upotrebom metode sort.

Primer:

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

//Funkcija za stampanje elemenata vektora a
void print( vector<int> a){
for(vector<int>::iterator ai=a.begin(); ai!=a.end(); ++ai)
cout << *ai << " ";
cout << endl;
cout << "----------------"<<endl;
}

int main()
{
//Deklarisanje vektora a
vector<int> a;
// Unos brojeva redom u vektor 9,8,7,6,5,4,3,2,1
for(int i=0; i<9;++i) a.push_back(9-i);
//Elementi vektora su poredjani u redosledu 9,8,7,6,5,4,3,2,1
print(a);
//Sortiranje elemenata vektora upotrebom metode sort iz STL biblioteke <algorithm>
sort( a.begin(), a.end() );
//Elementu su sortirani
print(a);
return 0;
}
```

Multimap

Multimap kontejner može da se koristi umesto hash strukture. Svaki element je sastavljen od dva dela: ključ i vrednost. Struktura dozvoljava ponavljanje ključeva po kom se vrši pretraga .
Klučevi po kojima se pretražuju moraju biti uporedljivi, tj. mora da može da se nad njima definiše relacija <. Za standardne tipove podataka, uključujući i biblioteku string, već imaju definisne ove operatore za poređenje.

Da bi mogla da se koristi ovaa struktura, potrebno je na početku programa da se uključi sledeća direktiva
`#include <map>`

Primer

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
// Definisanje multimap-a
multimap<string,int> mymm;
// Definisanje iteratora za ovakav tip multimape
```

```
multimap<string,int>::iterator it;
// Unos elemenata u multimap-u
mymm.insert(pair<string,int>("marko",10));
mymm.insert(pair<string,int>("ana",20));
mymm.insert(pair<string,int>("ana",30));
mymm.insert(pair<string,int>("mica",40));
mymm.insert(pair<string,int>("maca",50));
mymm.insert(pair<string,int>("kuca",60));
mymm.insert(pair<string,int>("ana",60));

// Stampanje elemenata multimap-e
for (it = mymm.begin();it != mymm.end();++it)
cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;

// Stampanje svih vrednosti pri svim pojavljivanjima kljuca ana
pair<multimap<string,int>::iterator,multimap<string,int>::iterator> ret;
ret = mymm.equal_range("bliksa");
for (it=ret.first; it!=ret.second; ++it)
cout << " " << (*it).second;
cout << endl;
// Broj elemenata sa kljucem ana
cout<<endl<<"Broj elemenata sa kljucem 'ana' : "<<mymm.count("ana");
// Dodavanje elementa na pocetak multiset-a
it=mymm.begin();
mymm.insert(it,pair<string,int>("bla",123));
// Stampamo elemente multimap-e
for (it = mymm.begin();it != mymm.end();++it)
cout << " [" << (*it).first << ", " << (*it).second << "]" << endl;
return 0;
}
```

PAIR

Klasa *pair* (tj. par) nalazi se u biblioteci *utility,* ali je često ne moramo uključiti (include-ovati), jer je uključe razne druge biblioteke. Klasa *pair* je veoma jednostavna i koristi se kada želimo da čuvamo dva različita tipa podataka pod istim nazivom.
Par ne podržava iteriranje, te nije kontejner. Često se dešava da funkcije koje moraju vratiti dve vrednosti, vrate par vrednosti.

Primer 01
```
#include <iostream>
#include <utility>
using namespace std;

int main()
{   pair<int,double> a;
   pair<int,int> b(3,4);
   pair<int,int> c(6,2);
a=make_pair(2,3.14);
cout << a.first << ' ' << a.second << endl;
cout << (b<c) << ' ' << (b==c) << endl;
b=c;
cout << (b==c)  << endl;

pair<pair<int,int>,int> trojka;
trojka=make_pair(make_pair(1,2),3);
cout << trojka.first.first << ' ';
cout << trojka.first.second << ' ';
cout << trojka.second << endl;
```

```
    return 0;
}
```

U liniji

```
  pair<int,double> a;
```

kreiran je par celog i realnog broja

U naredne dve linije kreirani su parovi <int,int> kojima su pridružene vrednosti.

Naredbom a=make_pair(2,3.14); pridružene su vrednosti paru a.

Paru se može pristupiti pomoću članova *first* i *second*.
Parovi se mogu i porediti i to tako što se najpre porede prvi elementi iz para, a ao su oni jednaki porede se i drugi elementi. Zato je par b(3,4) < c(6,2)

Par se cesto koristi za kreiranje ovakvih struktura:

```
vector  <pair<int,int> > v1;
vector  <pair< vector<int>, list<int> > > v2;
```

Primer 02

1. Data je pravougaona mrežu ulica predgrađa poznatog kao Bisergrad. Postoji 50000 vertikalnih ulica u smeru sever-jug (označenih x-koordinatama od 1 do 50000) i 50000 horizontalnih ulica u smeru istok-zapad (označenih y-koordinatama od 1 do 50000). Sve ulice su dvosmerne. Presek horiznotalne i vertikalne ulice naziva se biserraskršće. Stanovnici Bisergrada mogi biti vrlo neodgovorni i nesaavesni vozači, te je iz zbog bezbednosnih potreba, gradonačelnik Bisergrada postavio semafore na **N** biserraskršća. **Put** između dva biserraskršće je **opasan** ako na njemu negde postoji **skretanje bez semafora**, a inače je bezopasan. Nije moguće osigurati da svi putevi budu bezopasni, ali gradonačelniku je dovoljno da **između svaka dva semafora** bar **jedan od najkraćih puteva** bude **bezopasan**. Na žalost, trenutni raspored semafora to ne osigurava. Vaš je zadak postaviti **još neke semafore** (manje od 700 000) tako da skup semafora (koji sadrži i stare i nove semafore!) zadovoljava traženi zahtev. Razmislite i pomozite Bisergradu!

ULAZ

U prvoj liniji standardnog ulaza nalazi se prirodan broj **N** ($2 \leq N \leq 50000$) koji predstavlja broj postavljenih semafora. U svakoj od sledećih **N** linija nalazi se lokacija jednog semafora, predstavljena prirodnim brojevima **X** i **Y** ($1 \leq X, Y \leq 50000$), koordinatama vertikalne i horizontalne ulice koje se seku u tom biserraskršću. Svi semafori biće jedinstveni.

IZLAZ

Ispišite lokacije novih semafora, svaku u posebnoj liniji. Dozvoljeno je postavljanje više semafora na istu lokaciju. Broj novih semafora mora biti **manji od 700 000**.

TEST PRIMERI

ULAZ	**IZLAZ**
2 1 1 3 3	1 3
3 2 5 5 2 3 3	3 5 5 3
5 1 3 2 5 3 4 4 1 5 2	1 5 3 3 3 5 4 2 4 3

Resenje:

Date semafore sortirajmo po x-koordinati. Neka je **x'** srednja (median) x koordinata u tom nizu. Neka je **A** skup datih semafora levo od **x'**, a **B** skup datih semafora desno od **x'**.

Povežimo bezopasnim putem svaki par semafora **a**, **b** takvih da je **a** iz skupa **A**, te **b** iz skupa **B**. Kako? Tako da dodamo nove semafore na lokacije (**x'**, **y**) za sve y-koordinate **y** iz skupova **A** i **B**. Sada za semafore **a** i **b** imamo bezopasan put (x_a, y_a) → (**x'**, y_a) → (**x'**, y_b) → (x_b, y_b).

Još je preostalo povezati međusobno semafore unutar skupa **A**, kao i semafore unutar skupa **B**. To činimo tako da rekurzivno ponovimo opisani postupak posebno za skup **A** i posebno za skup **B**. Ovaj način razmišljanja naziva se *podeli pa vladaj* (divide and conquer).

Koliki je broj dodatih semafora? Budući da delimo skup na dva dela, najveća je dubina rekurzije O(log **N**). Posmatrajmo neki početni semafor na lokaciji (**x**, **y**). U najgorem slučaju, na svakoj dubini rekurzije on će biti uključen u jedan skup i tamo će generisati jedan novi semafor (**x'**, **y**). Dakle, jedan početni semafor generise O(log **N**) novih semafora, što daje ukupno O(**N** log **N**) novih semafora.
Tačan broj uz pažljivu implementaciju ispada manji od 700000.

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

const int MaxN = 100005;
pair<int, int> semafor[MaxN];

void bPretraga(int donja, int gornja) {
  if (donja + 2 == gornja) {
    printf("%d %d\n", semafor[donja].first, semafor[donja + 1].second);
    return;
  }
  if (donja + 2 > gornja) return;
  int sredina = (donja + gornja) / 2;
  int x = semafor[sredina].first;
  for (int i = donja; i < gornja; ++i)
    if (i != sredina)
      printf("%d %d\n", x, semafor[i].second);
  bPretraga(donja, sredina);
  bPretraga(sredina + 1, gornja);
}

int main () {
  int n;
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) {
    scanf("%d%d", &semafor[i].first, &semafor[i].second);
  }
  sort(semafor, semafor + n);
  bPretraga(0, n);
}
```

SKUP (klasa set)

Klasa set definisana je u biblioteci set.

Literatura za dalji rad

1. **http://www.csci.csusb.edu/dick/samples/**
2. **http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html**
3. **http://www.cplusplus.com/reference/stl/**
4. http://web.math.pmf.unizg.hr/~vpetrice/OP/Vjezbe%2002%20-%20STL.pdf

1.

```
#include <iostream>
#include <string>
#include <vector>
#include <deque>
#include <queue>
#include <stack>
#include <list>


using namespace std;

int main()
{

  // Klasa string (zaglavlje <string>) ------------------------------
  // Konstruktori:
  // string s = "Hello";
  // string s1; // Prazan string
  // string s2 = s; // Copy konstrukcija

  // Operator dodele:
  // s = "Hello C++";
  // s = s1;

  // Aritmeticki i relacioni operatori:
  // s + s1 (konkatenacija)
  // s += s1 (dopisivanje na s)
  // s + " world" (konkatenacija sa char *)
  // s == s1, s != s1 (poredjenje na jednakost i razlicitost)
  // s < s1, s <= s1, s > s1, s >= s1 (leksikografsko poredjenje)
  // s.size() (trenutna duzina stringa).
  // s.c_str() (vraca char * -- adresu niza karaktera koji sadrzi
  //                             string)
  // cin >> s; cout << s; (operatori ulaza i izlaza)
  // s[i] (indeksni pristup karakterima)
  // podrzava iteratore, resize(), capacity() i  sl. kao i u vektoru.

  string str;

  // vector<T> (zaglavlje <vector>) ------------------------------------
  // Konstruktori:
  // vector<int> v; (prazan vektor, size() == 0)
  // vector<int> v1 = v; (copy konstrukcija)
  // vector<int> v2(n); (vektor duzine n, svi elementi su podrazumevane
  //                      vrednosti, u slucaju int-a -- nule)
  // vector<int> v3(n, x) (vektor duzine n, svi elementi su inicijalizovani
  //                        vrednoscu x)
  // v2 = v1; -- operator dodele, kopira sadrzaj vektora v1 u v2 (nakon
  //             dodele vazi v1 == v2)
  // v1 == v2, v1 != v2 (poredjenje na jednakost i razlicitost)
  // v1 < v2, v1 > v2, v1 <= v2, v1 >= v2 (leksikografsko poredjenje)
  // v.size()  --  velicina

  // v.capacity() -- kapacitet ( >= size())
  // v.resize(n) -- menja velicinu (uz odsecanje elemenata na
  //                kraju ako je n < v.size(), ili dopisivanje podrazumevanih
  //                vrednosti za dati tip u slucaju da je n > v.size())
  // v.reserve(n) -- menja kapacitet (najcesce nema potrebe za tim)
  // v.push_back(x) -- dodaje na kraj (uz eventualnu realokaciju)
  // v.pop_back() -- skida poslednji element
  // v.back() -- referenca na poslednji element
```

```
  // v.front() -- referenca na prvi element
  // v[i]      -- indeksni pristup (indeksi idu od 0 do size() - 1)
  // v.clear()  -- brise sadrzaj vektora (prazni ga i vraca size() na nulu)
  // indeksni pristup ne vrsi automatsku realokaciju!! Ukoliko je indeks
  // i >= v.size(), tada je v[i] neispravna upotreba vektora. Potrebno je
  // najpre resize() funkcijom uvecati velicinu niza (npr. v.resize(i + 1),
  // kako  bi u vektoru v postojao element sa indeksom i. Sa druge strane,
  // push_back() automatski uvecava size() za jedan, vrseci realokaciju
  // ako je potrebno.


  vector<int> v;

  cout << "size(): " << v.size() << endl;
  cout << "capacity(): " << v.capacity() << endl;

  for(int i = 0; i < 50; i++)
    {
      v.push_back(i);
      cout << "size(): " << v.size() << endl;
      cout << "capacity(): " << v.capacity() << endl;
      cout << "back(): " << v.back() << endl;
    }

  // Iteracija pomocu indeksa
  for(int i = 0; i < v.size(); i++)
    {
      cout << v[i] << endl;
    }

    // Iteracija pomocu iteratora
for(vector<int>::iterator it = v.begin() ; it != v.end(); it++)
    {
      cout  << *it << endl;
    }

  v.clear();
  cout << "size(): " << v.size() << endl;
  cout << "capacity(): " << v.capacity() << endl;


  // deque<T> -- dek (vektor koji se moze efikasno prosirivati sa oba kraja,
  //              zaglavlje <deque>)


  // Osim push_back() i pop_back(), dek sadrzi i push_front() i pop_front(),
  // koji omogucavaju efikasno umetanje elementa na pocetak deka. U svemu
  // ostalom se dek ponasa kao i vektor, ima efikasni indeksni pristup
  // elementima. Dodavanje u sredinu je, kao i kod vektora, neefikasno!!


    // Adapterske klase queue i stack: ove dve klase interno koriste neki od
   // sekvencijalnih kontejnera (deque ili list, u slucaju steka moze i vector)
 // ali korisniku nude jednostavniji i prirodniji interfejs.


  // zaglavlje <queue> ----------------------------------------------
 // queue<T> -- red (podrazumevano implementiran kao deque<T>)
  // queue<T, list<T> >  -- red implementiran pomocu liste


  // Konstruktor:
  // queue<int> q; // prazan red
  // q.push(x) // dodaje  x na kraj reda
  // q.pop() // skida sa pocetka reda
  // q.front() // vraca element sa pocetka reda
  // q.back() // vraca element sa kraja reda

   queue<int> q;
```

```
// zaglavlje <stack> ------------------------------------------
// stack<T> -- stek (podrazumevano implementiran kao deque<T>)
// stack<T, vector<T> > -- stek implementiran pomocu vektora

 // Konstruktor:
 // stack<int> s;
 // s.push(x); // postavlja x na stek
 // s.pop();  // skida element sa vrha steka
 // s.top(); // vraca element trenutno na vrhu steka bez skidanja
 //

 stack<int> s;

  // list<T> (zaglavlje <list>, dvostruko povezana lista)
// Omogucava efikasno umetanje i brisanje bilo gde u kontejneru
 // Ne omugucava efikasno pristupanje proizvoljnom elementu, ne
 // poseduje indeksni operator []. Elementima se mora pristupati
 // redom.
 // Konstruktori:
 // list<int> l;  (prazna lista)
 // list<int> l(n); (lista od n int-ova inicijalizovanih podrazumevanim
 //                  vrednostima -- nulama)
 // list<int> l(n, x); (lista od n int-ova inicijalizovanih vrednoscu x)
 // l.size(); // velicina liste
 // l.push_back(), l.pop_back(), l.push_front(), l.pop_front(), l.back(),
 // l.front() -- isto znacenje kao i kod vektora i deka
 // NAPOMENA: Jedini nacin prolaska kroz listu je pomocu iteratora, buduci
 // da ne postoji indeksni pristup!!


 list<int> lista;
 lista.push_back(1);
 lista.push_back(2);
 lista.push_back(8);

 list<int>::iterator it = lista.begin();
 it++;
 it++;

lista.insert(it, 5);

  for(list<int>::iterator iter = lista.begin(); iter != lista.end();
     iter++)
    {
     cout << *iter << endl;
    }

 // Iteratori:
// Svaka kontejnerska klasa ima ugnjezdeni tip iterator koji predstavlja
 // klasni tip koji simulira "pametni pokazivac". Svaki iterator je interno
 // implementiran na nacin koji odgovara kontejneru u kome je definisan,
 // a spolja se svi iteratori ponasaju isto, tj. imaju isti interfejs, nalik
 // pokazivackoj sintaksi:

 // list<int>::iterator i1 (iterator koji pokazuje na element liste int-ova)
 // vector<double>::iterator i2  (iterator koji pokazuje na element vektora
 //                               double-ova).

 // deque< vector<int> >::iterator i3 (iterator koji pokazuje na element deka
 //                                    ciji su elementi vektori intova)

 // Iteratori podrzavaju sledece operacije:
 // *it -- vraca referencu na element na koji trenutno pokazuje
 // it++ (++it) -- pomera se na sledeci element u kontejneru
 // it-- (--it) -- pomera se na prethodni element u kontejneru
 // it1 == it2 -- da li iteratori pokazuju na isti element?
 // it1 != it2 -- da li iteratori pokazuju na razlicite elemente?
```

```
  // Iteratori u kontejnerima sa indeksnim pristupom (vector, deque)
  // podrzavaju i sledece dodatne operacije:
  // it + n (iterator koji pokazuje na element n pozicija udesno u odnosu
  //         na element na koji pokazuje it)
  // it - n (slicno, samo ulevo)
  // it += n, it -= n (kombinacija prethodna dva sa dodelom)
  // it1 - it2 (ako it1 i it2 pokazuju na elemente u istom kontejneru,
  //            tada je rezultat rastojanje izmedju elemenata na koje pokazuju)
  // it1 < it2, it1 > it2, it1 <= it2, it1 >= it2 (prvi slucaj znaci da it1
  // pokazuje na element u kontejneru pre elementa na koji pokazuje it2, ostalo
  // analogno).
  // it[n]  (ekvivalentno sa *(it + n))


  // NAPOMENA: Svi kontejneri sadrze funkcije clanice begin() i end() koje
  // vracaju iterator na prvi element u kolekciji i iterator na poziciju

  // NEPOSREDNO IZA POSLEDNJEG ELEMENTA u kolekciji (prva invalidna pozicija)
  // VAZNO: end() NE VRACA ITERATOR NA POSLEDNJI ELEMENT, u tu svrhu se morate
  // vratiti nazad za jedno mesto (it--).

    // Iteratori nude uniforman nacin za prolazak kroz kontejner, bez obzira
// na tip kontejnera!!

  // Umetanje i brisanje pomocu iteratora:
  // Iteratori omogucavaju da se u svim tipovima kontejnera vrsi umetanje
  // i brisanje elemenata na proizvoljnoj poziciji. Zato svi kontejneri
  // podrzavaju funkcije insert() i erase():
  // c.insert(it, x) // umece u kontejner c element x neposredno ispred
  //                    elementa na koji pokazuje it.

  // Primetimo da je c.push_back(x) <=> c.insert(c.end(), x), kao i
  //                 c.push_front(x) <=> c.insert(c.begin(), x).
  // c.insert(it, n, x) // umece n kopija vrednosti x neposredno ispred
  //                       elementa na koji pokazuje it
  // c.insert(it, it_beg, it_end) // umece elemente nekog drugog kontejnera
  //                                 u opsegu [it_beg, it_end) u kontejner
  //                                 c neposredno pre elementa na koji pokazuje
  //                                 it.
  // c.erase(it)  // brise element kontejnera c na koji pokazuje it
  // c.erase(it1, it2) // brise sve elemente kontejnera c u opsegu [it1, it2)


  // VAZNA NAPOMENA: Iako svi kontejneri podrzavaju navedene insert() i erase()
  // operacije, ove operacije se efikasno izvrsavaju na proizvoljnoj poziciji
  // samo za liste. Kod vektora su operacije insert() i erase() efikasne samo
  // u slucaju iteratora koji vraca end(), tj. brisanje i dodavanje na kraj.

  // Kod deka su operacije insert() i erase() efikasne samo u slucaju
  // iteratora koji vraca begin() ili end(), tj. brisanje i dodavanje na pocetak
  // ili kraj. U ostalim situacijama se vrsi fizicko pomeranje elemenata koji
  // slede nakon pozicije umetanja, sto je neefikasno u opstem slucaju.

  // Postavljamo iterator it da pokazuje na treci element u listi.

  it = lista.begin();
  it++;
  it++;

  // Neposredno pre treceg elementa ubacujemo u listu prva tri elementa

  // vektora v

  lista.insert(it, v.begin(), v.begin() + 3);

  for(list<int>::iterator iter = lista.begin(); iter != lista.end();
      iter++)
    {
      cout << *iter << endl;
```

```
    }


  // JOS O ITERATORIMA:

  // Pored klase iterator, svaki kontejner definise i klasu const_iterator.
  // konstantni iteratori funkcionisu na identican nacin kao i obicni
  // iteratori, jedino sto prilikom primene operatora dereferenciranja
  // (operator*) vracaju referencu na konstantan element kontejnera. Time
  // se sprecava modifikacija elementa na koji iterator pokazuje.

  // Funkcije begin() i end() u kontejnerima vracaju iterator akko je kontejner
  // nekonstantan, u suprotnom vracaju const_iterator. Dozvoljeno je
  // konvertovati iterator u const_iterator, ali ne i obrnuto. To je u skladu
  // sa politikom jezika o kvalifikacionim konverzijama.

  for(list<int>::const_iterator iter = lista.begin(); iter != lista.end();
      iter++)
    {
      // iter je const_iterator. lista je nekonstantan objekat klase list<int>
      // zbog cega lista.begin() vraca iterator. Konverzija iteratora u
      // const_iterator je dozvoljena, tako da prevodilac ne prijavljuje
      // gresku. U obrnutoj situaciji bi se greska javila!!


      // *iter = 5; // POGRESNO!! iter je const_iterator, pa je *iter
      //               tipa const int, zato nije moguce dodeliti mu vrednost.


      cout << *iter  << endl; // Ovo je u redu!


    }

  // Konstrukicja pomocu iteratora:
  // Dozvoljeno je inicijalizovati jedan kontejner sekvencom elemenata
  // drugog kontejnera ogranicenom parom iteratora:

  vector<int> vec(lista.begin(), lista.end()); // Konstruise vektor
  // koji se sastoji iz istih elemenata kao i lista.


  for(vector<int>::const_iterator iter = vec.begin(); iter != vec.end();
      ++iter)
    {
      cout << *iter << endl;
    }

  return 0;
}

// Sledi spisak funkcija koje opisuju koriscenje kontejnera kao argumenata
// funkcija. O tome ce se kasnije vise pricati


// Prenosenje vektora funkciji

void f1(const vector<int> & v)
{
  // Ako necemo da menjamo vektor

}

void f2(vector<int> & v)
{
  // Ako zelimo da menjamo vektor

}
```

```
void f3(vector<int> v)
{
  // Kreira se lokalna kopija (copy
  // konstrukcijom). Retko potrebno,
  // veoma neefikasno.

}

// Vracanje vektora iz funkcije
vector<int>  g1()
{
  vector<int> v;

  // Kod koji pridruzuje vrednosti u vektor ...

  return v; // Vraca vektor copy konstrukcijom
}

vector <int> & g2()
{
  vector<int> v;

  // ...

   return v; // Pogresno!! Vraca referencu na
           // objekat koji ce biti unisten
            // nakon zavrsetka funkcije

}



class X {

private:
  vector<double> _v;

public:
    const vector<double> & getV() const
  {
    return _v; // U redu, vraca referencu na
               // objekat koji je clan klase
               // i samim tim nastavlja da
               // zivi i nakon zavrsetka
               // funkcije.

  }

};
```

2. Proceniti vremensku složenost sledećeg koda, imajući na umu da poziv metoda `empty` ima konstantnu vremensku složenost u odnosu na postavljeni kontejner.

```
#include <iostream>       // std::cout
#include <stack>          // std::stack

using namespace std;
int main ()
{
  stack<int> stek1;
  int sum=0;

  for (int i=1;i<=10;i++) stek1.push(i);

  while (!stek1.empty())
  {
     sum += stek1.top();
     stek1.pop();
```

```
  }

  cout << "Ukupna suma: " << sum << '\n';

  return 0;
}
```

3. Proceniti vremensku složenost sledećeg koda imajući na umu da je vremenska složenost metoda count je logaritam od veličine mape.

```
// map::count
#include <iostream>
#include <map>
using namespace std;

int main ()
{
  map<char,int> mapa_niz;
  char c;

  mapa_niz ['a']=1;
  mapa_niz ['c']=2;
  mapa_niz ['f']=3;

  for (c='a'; c<'h'; c++)
  {
    cout << c;
    if (mapa_niz.count(c)>0)
      std::cout << " jeste element niza mapa_niz.\n";
    else
      std::cout << " nije element niza mapa_niz.\n";
  }

  return 0;
}
```

4. Šta je rezultat rada sledećeg programa koji koristi multimape?

```
// multimap::count
#include <iostream>
#include <map>
using namespace std;

int main ()
{
  multimap<char,int> niz;

  niz.insert(make_pair('x',50));
  niz.insert(make_pair('y',100));
  niz.insert(make_pair('y',150));
  niz.insert(make_pair('y',200));
  niz.insert(make_pair('z',250));
  niz.insert(make_pair('z',300));

  for (char c='x'; c<='z'; c++)
  {
    cout << "Postoji " << niz.count(c) << " elemenata ciji kljuc je " << c << ":";
    multimap<char,int>::iterator it;
    for (it=niz.equal_range(c).first; it!=niz.equal_range(c).second; ++it)
      std::cout << ' ' << (*it).second;
    std::cout << '\n';
  }

  return 0;
}
```

5. Šta je rezultat rada sledećeg programa koji koristi metod count_if?

```
// count_if example
```

```
#include <iostream>     // std::cout
#include <algorithm>    // std::count_if
#include <vector>       // std::vector

bool IsOdd (int i) { return ((i%2)==1); }

int main () {
  std::vector<int> myvector;
  for (int i=1; i<10; i++) myvector.push_back(i); // myvector: 1 2 3 4 5 6 7 8 9

  int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
  std::cout << "myvector sadrzi " << mycount  << " neparnih vrednosti.\n";

  return 0;
}
```

6. Šta je rezultat rada sledećeg programa koji koristi metod make_heap?

```
// range heap example
#include <iostream>     // std::cout
#include <algorithm>    // std::make_heap, std::pop_heap, std::push_heap, std::sort_heap
#include <vector>       // std::vector

int main () {
  int myints[] = {10,20,30,5,15};
  std::vector<int> v(myints,myints+5);

  std::make_heap (v.begin(),v.end());
  std::cout << "polazni max heap   : " << v.front() << '\n';

  std::pop_heap (v.begin(),v.end()); v.pop_back();
  std::cout << "max heap nakon operacije pop : " << v.front() << '\n';

  v.push_back(99); std::push_heap (v.begin(),v.end());
  std::cout << "max heap nakon operacije push: " << v.front() << '\n';

  std::sort_heap (v.begin(),v.end());

  std::cout << "nakon sortiranja:";
  for (unsigned i=0; i<v.size(); i++)
    std::cout << ' ' << v[i];

  std::cout << '\n';

  return 0;
}
```

7.

```
// priority_queue::push/pop
#include <iostream>       // std::cout
#include <queue>          // std::priority_queue

int main ()
{
  std::priority_queue<int> mypq;

  mypq.push(30);
  mypq.push(100);
```

```
  mypq.push(25);
  mypq.push(40);

  std::cout << "Nakon operacije pop...";
  while (!mypq.empty())
  {
     std::cout << ' ' << mypq.top();
     mypq.pop();
  }
  std::cout << '\n';

  return 0;
}
```