Čas 4.1, 4.2, 4.3 - Bibliotečke strukture podataka

Da bi algoritmi mogli efikasno funkcionisati potrebno je da podaci koji se obrađuju budu organizovani na način koji omogućava da im se efikasno pristupa i da se efikasno modifikuju i ažuriraju. Za to se koriste *strukture podataka* (ovaj pojam ne treba mešati sa strukturama u programskom jeziku C).

Strukture podataka su obično kolekcije podataka koje omogućavaju neke karakteristične operacije. U zavisnosti od implementacije samih struktura izvođenje tih operacija može biti manje ili više efikasno. Prilikom dizajna algoritma često se fokusiramo upravo na operacije koje nam određena struktura nudi, podrazumevajući da će sama struktura podataka biti implementirana na što efikasniji način. Sa stanovišta dizajna algoritma nam je potrebno samo da znamo koje operacije podrazumeva određena struktura podataka i da imamo neku procenu (obično asimptotsku ili amortizovanu) izvođenja te operacije. U tom slučaju kažemo da algoritme konstruišemo u odnosu na *apstraktne strukture podataka*, čime se prilikom dizajna i implementacije algoritama oslobađamo potrebe da brinemo o detaljima implementacije same strukture podataka. Najznačajnije strukture podataka su implementirane u bibliotekama savremenih programskih jezika (C++, Java, C#, Python i mnogi drugi imaju veoma bogate biblioteke struktura podataka). U ovom poglavlju ćemo se baviti upotrebom i primenom gotovih struktura podataka u implementaciji nekih interesantnih algoritama. Sa druge strane, poznavanje implementacije samih struktura podataka može biti značajno za njihovo duboko razumevanje, ali i za mogućnost njihovog prilagođavanja upotrebi u nekim specifičnim algoritmima, kao i za mogućnost definisanja novih struktura podataka, koje nisu podržane bibliotekom programskog jezika koji se koristi. Stoga ćemo se u narednom poglavlju baviti načinima implementacije struktura podataka.

Skalarni tipovi

Pojedinačni podaci se čuvaju u promenljivama osnovnih, skalarnih tipova podataka (to su obično celobrojni i realni tipovi podataka, karakterski tip, logički tip, pa čak i niske, ako se gledaju kao atomički podaci, bez analize njihovih delova).

Parovi, torke, slogovi

Najjednostavnije kolekcije podataka sadrže samo mali broj (2, 3, 4, ...) podataka, koji mogu biti i različitih tipova. U savremenim programskim jezicima postoje dva načina da se podaci upakuju na taj način. Jedno su uređeni parovi (engl.

pair) tj. uređene torke (engl. tuple) gde se svakom pojedinačnom podatku pristupa na osnovu pozicije. Drugo su slogovi tj. strukture (engl. record, struct) u gde se svakom pojedinačnom podatku pristupa na osnovu naziva. Prilikom korišćenja slogova potrebno je eksplicitno definisati novi tip podataka, pa njihovo korišćenje zahteva malo više programiranja, nego korišćenje parova i torki koje obično teče ad hoc.

Parovi

U jeziku C++ se tip para navodi kao `pair<T1, T2>` gde su `T1` i `T2` tipovi prve i druge komponente. Pristup prvom elementu para vrši se poljem `first`, a drugom poljem `second`. Par se od vrednosti dobija funkcijom `make_pair`, a moguća je inicijalizacija korišćenjem vitičastih zagrada (npr. `pair<int, int> p{0, 0}`). Funkcijom `tie` moguće je par razložiti na komponente (npr. `tie(x, y) = p` uzrokuje da promenljiva `x` sadrži prvu, a `y` drugu komponentu para). Prikažimo sada upotrebu parova u jeziku C++ za modelovanje tačaka u ravni.

Problem: Date su koordinate tri tačke trougla. Odredi koordinate njegovog središta.

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

int main() {
   pair<double, double> A, B, C;
   cin >> A.first >> A.second;
   cin >> B.first >> B.second;
   cin >> C.first >> C.second;
   pair<double, double> T;
   T.first = (A.first + B.first + C.first) / 3.0;
   T.second = (A.first + B.first + C.first) / 3.0;
   cout << T.first << " " << t.second << endl;
   return 0;
}
```

U jeziku Python, upotreba parova je još jednostavnija.

```
(Ax, Ay) = (int(input()), int(input()))
(Bx, By) = (int(input()), int(input()))
(Cx, Cy) = (int(input()), int(input()))
(Tx, Ty) = ((Ax + Bx + Cx) / 3.0, (Ay + By + Cy) / 3.0)
print(Tx, Ty)
```

Torke

U jeziku C++ se tip torke navodi kao `tuple<T0, T1, ...>` gde su `Ti` redom tipovi komponenata torke. Pristup i-tom elementu torke vrši se funkcijom `get<i>`. Torka se od pojedinačnih vrednosti gradi funkcijom `make_tuple`. Funkcijom `tie` moguće je torku razložiti na komponente. Pod pretpostavkom da se pojedinačne komponente mogu porediti i torke se mogu porediti relacijskim operatorima (i tada se koristi leksikografski poredak). Prikažimo sada upotrebu torki u jeziku C++ za modelovanje poređenja datuma (koje je suštinski leksikografsko).

Problem: Sa ulaza se učitava niz datuma. Odredi najkasniji od njih.

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

tuple<int, int, int> ucitaj_datum() {
   int d, m, g;
   cin >> d >> m >> g;
   return make_tuple(g, m, d);
}

void ispisi_datum(const tuple<int, int, int>& datum) {
   cout << get<2>(datum) << " " <<
           get<1>(datum) << " " <<
           get<0>(datum) << endl;
}

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

   tuple<int, int, int> max_datum = ucitaj_datum();
   for (int i = 1; i < n; i++) {
      tuple<int, int, int> datum = ucitaj_datum();
      if (datum > max_datum)
         max_datum = datum;
   }
   ispisi_datum(max_datum);
   return 0;
}
```

Korišćenje torki u jeziku Python je još jednostavnije.

```
def ucitaj_datum():
    (d, m, g) = (int(input()), int(input()), int(input()))
    return (g, m, d)

def ispisi_datum(datum):
    (g, m, d) = datum
```

```
    print(d, m, g)

n = int(input())
max_datum = ucitaj_datum()
for i in range(1, n):
    datum = ucitaj_datum()
    if datum > max_datum:
       max_datum = datum
ispisi_datum(max_datum)
```

Slogovi (strukture)

Nema velike razlike između definisanja i korišćenja struktura u jeziku C i jeziku C++. Ilustrujmo to na primeru razlomaka.

Problem: Definisati strukturu za predstavljanje razlomaka, funkciju za sabiranje razlomaka i glavni program koji ih testira.

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

struct razlomak {
   int brojilac, imenilac;
};

int nzd(int a, int b) {
    while (b != 0) {
       int tmp = a % b;
       a = b;
       b = tmp;
    }
    return a;
}

void skrati(razlomak& r) {
   int n = nzd(r.brojilac, r.imenilac);
   r.brojilac /= n;
   r.imenilac /= n;
}

razlomak saberi(const razlomak& r1, const razlomak& r2) {
   razlomak zbir;
   zbir.brojilac = r1.brojilac * r2.imenilac + r2.brojilac * r1.imenilac;
   zbir.imenilac = r1.imenilac * r2.imenilac;
   skrati(zbir);
   return zbir;
}

int main() {
```

```
    razlomak r1{3, 5}, r2{2, 3};
    razlomak zbir = saberi(r1, r2);
    cout << zbir.brojilac << "/" << zbir.imenilac << endl;
    return 0;
}
```

Nizovi (statički, dinamički)

Nizovi (u raznim oblicima) predstavljaju praktično osnovne kolekcije podataka. Osnovna karakteristika nizova je to da omogućavaju efikasan pristup (po pravilu u složenosti $O(1)$ ili bar u amortizovanoj složenosti $O(1)$) elementu niza na osnovu njegove pozicije (kažemo i indeksa). Pristup van granica niza obično prouzrokuje grešku u programu (u nekim jezicima poput Java, C# ili Python se podiže izuzetak, a u nekim poput C ili C++ ponašanje nije definisano). U zavisnosti od toga da li je broj elemenata niza poznat (i ograničen) u trenutku pisanja i prevođenja programa ili se određuje i menja tokom izvršavanja programa nizove delimo na statičke i dinamičke. Osnovna operacija u radu sa nizovima je pristup i-tom elementu. U velikom broju savremenih programskih jezika (npr. C++, Java, C#, Python) brojanje elemenata počinje od nule.

Statički nizovi

Rad sa statičkim nizovima u jeziku C++ je veoma sličan radu sa statičkim nizovima u C-u. Ilustrujmo to jednim jednostavnim primerom.

Problem: Učitava se najviše 100 brojeva. Izračunaj element koji najmanje odstupa od proseka.

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

int main() {
    // učitavamo broj elemenata i zatim sve elemente u niz
    int a[100];
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
        cin >> a[i];

    // izračunavamo prosek
    int zbir = 0;
    for (int i = 0; i < n; i++)
        zbir += a[i];
    double prosek = (double) zbir / (double) n;

    // određujemo i ispisujemo element koji najmanje
```

```
    // odstupa pod proseka
    int min = 0;
    for (int i = 1; i < n; i++)
        if (abs(a[i] - prosek) < abs(a[min] - prosek))
            min = i;
    cout << a[i] << endl;

    return 0;
}
```

Naglasimo da je u ovom zadatku bilo neophodno elemente prilikom učitavanja memorisati, jer su nam za rešenje zadatka potrebna dva prolaska kroz podatke (prvi u kom se određuje prosek i drugi u kom se određuje minimalno odstupanje od proseka). Da se tražilo samo određivanje proseka ili prosečno odstupanje od minimuma, niz nije bilo neophodno koristiti (razmisli kako bi se to uradilo u memorijskoj složenosti $O(1)$).

Dinamički nizovi

Kada nije unapred poznata dimenzija niza ili kada se očekuje da će se ona prilično menjati pri raznim pokretanjima programa poželjno je korišćenje dinamičkih nizova. Dinamički nizovi obično podržavaju rezervaciju prostora za određeni broj elemenata (a taj broj kasnije može biti menjan). Takođe, dopuštena operacija je dodavanje na kraj niza. U slučajevima kada u nizu nema dovoljno prostora za smeštanje elemenata vrši se automatska realokacija.

U jeziku C++ dinamički nizovi su podržani klasom `vector<T>` gde je `T` tip podataka koji se smeštaju u vektor. Prilikom konstrukcije moguće je navesti inicijalni broj elemenata. Metod `size` se može koristiti za određivanje veličine niza (tj. broja njegovih elemenata). Metodom `resize` vrši se efektivna promena veličine niza. Ako je veličina manja, krajnji elementi će biti obrisani. Ako je veličina veća (i ako ima dovoljno memorije) novi elementi niza će biti inicijalizovani na svoju podrazumevanu vrednost (za `int` to je vrednost 0). Moguće pored veličine niza i eksplicitno navesti vrednost na koju želimo da elementi niza budu inicijalizovani. Rezervaciju prostora (ali bez promene veličine niza) moguće je postići pozivom metode `reserve`. Metodom `push_back` vrši se dodavanje elementa na kraj niza (time se veličina niza povećava za jedan).

Prikažimo upotrebu vektora na sledećoj varijaciji prethodnog zadatka.

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

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

int main() {
```

```
    // učitavamo sve elemente u vektor
    vector<int> a;
    string linija;
    while (cin >> linija)
        a.push_back(stoi(linija));

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

    // određujemo 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 << a[min] << endl;

    return 0;
}
```

Iako se osnovne kolekcije u jeziku Python nazivaju liste, one zapravo predstavljaju dinamičke nizove (pre svega zbog efikasnog indeksnog pristupa i zbog mogućnosti dodavanja elemenata metodom `append`).

```
import sys

# učitavamo elemente u listu
a = []
for linija in sys.stdin:
    a.append(int(linija))

prosek = sum(a) / len(a)

min = 0;
for i in range(1, n):
    if abs(a[i] - prosek) < abs(a[min] - prosek):
        min = i

print(a[min])
```

U jeziku Java dinamički nizovi su podržani putem klase `ArrayList`, a u jeziku C# putem klase `List` (pri čemu u ovim jezicima ne postoje klasični statički nizovi, već se obični nizovi takođe alociraju na hipu, tokom izvršavanja programa, jedino što im nije moguće dinamički menjati dimenziju).

Višedimenzionalni nizovi, matrice (statički, dinamički)

U nekim slučajevima su nam potrebni višedimenzionalni nizovi. U nekim jezicima oni su podržani direktno, a nekada se realizuju kao nizovi nizova. U dvodimenzionom slučaju jedno od osnovnih pitanja je da li su u pitanju matrice kod kojih sve vrste imaju isti broj elemenata ili su u pitanju nizovi vrsta kod kojih svaka vrsta može imati različit broj elemenata. I višedimenzionalni nizovi mogu biti statički i dinamički.

U jeziku C++ statički višedimenzionalni nizovi se koriste na veoma sličan način kao jednodimenzionalni (i veoma slično kao u C-u). Ilustrujmo to jednim jednostavnim primerom.

Problem: Napisati program koji učitava i pamti matricu dimenzije 5×5 i zatim izračunava njen trag (zbir dijagonalnih elemenata).

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

int main() {
  // učitavamo matricu
  int A[5][5];
  for (int v = 0; v < 5; v++)
    for (int k = 0; k < 5; k++)
      cin >> A[v][k];

  // izračunavamo i ispisujemo trag
  int trag = 0;
  for (int i = 0; i < 5; i++)
    trag += A[i][i];
  cout << trag << endl;

  return 0;
}
```

Ukoliko dimenzija nije unapred poznata, najlakše nam je da za smeštanje matrice upotrebimo vektor vektora.

Problem: Napisati program koji učitava i pamti matricu dimenzije $n \times n$ i zatim izračunava njen trag.

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

int main() {
  // učitavamo elemente matrice
  int n;
  cin >> n;
  // alociramo prostor za n vrsta matrice
```

```
  vector<vector<int>> A(n);
  for (int v = 0; v < n; v++) {
     // u vrsti v alociramo prostor za n elemenata
     A[v].resize(n);
     // učitavamo elemente vrste v
     for (int k = 0; k < n; k++)
       cin >> A[v][k];
   }

   // izračunavamo i ispisujemo trag
   int trag = 0;
   for (int i = 0; i < n; i++)
     trag += A[i][i];
   cout << trag << endl;

   return 0;
}
```

U slučaju vektora vektora, vrste mogu imati i različit broj kolona.

Problem: Napisati program koji učitava i pamti ocene nekoliko studenata i za svakog izračunava prosečnu ocenu. Sa ulaza se prvo učitava broj studenata, a zatim za svakog studenta broj ocena i nakon toga pojedinačne ocene (broj ocena različitih studenata može biti različit).

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

int main() {
  // učitavamo tabelu ocena svih učenika
  int brojUcenika;
  cin >> brojUcenika;
  vector<vector<int>> ocene(brojUcenika);
  for (int u = 0; u < brojUcenika; u++) {
    int brojOcena;
    cin >> brojOcena;
    ocene[u].resize(brojOcena);
    for (int o = 0; o < n; o++)
      cin >> ocene[u][o];
  }

  // izračunavamo i ispisujemo sve proseke ocena
  for (int u = 0; u < brojUcenika; u++) {
    int zbir = 0;
    int brojOcena = ocene[u].size();
    for (int o = 0; o < brojOcena; o++)
      zbir += ocene[u][o];
    cout << (double) zbir / (double) brojOcena << endl;
  }
```

```
  return 0;
}
```

Stekovi

Stek predstavlja kolekciju podataka u koju se podaci dodaju po LIFO principu - element se može dodati i skinuti samo na vrha steka.

U jeziku C++ stek se realizuje klasom `stack<T>` gde `T` predstavlja tip elemenata na steku. Podržane su sledeće metode:

- `push` - postavlja dati element na vrh steka
- `pop` - skida element sa vrha steka
- `top` - očitava element na vrhu steka (pod pretpostavkom da stek nije prazan)
- `empty` - proverava da li je stek prazan
- `size` - vraća broj elemenata na steku

Stek u jeziku C++ je zapravo samo adapter oko neke kolekcije podataka (podrazumeano vektora) koji korisnika tera da poštuje pravila pristupa steku i sprečava da napravi operaciju koja nad stekom nije dopuštena (poput pristupa nekom elementu ispod vrha).

U jeziku Python stek se simulira pomoću običnih lista. Metoda `append` dodaje element na kraj liste (koji se tumači kao vrh steka), dok metoda `pop` uklanja element sa steka.

Prikažimo upotrebu steka kroz nekoliko interesantnih primera.

Istorija veb-pregledača

Problem: Pregledač veba pamti istoriju posećenih sajtova i korisnik ima mogućnost da se vraća unatrag na sajtove koje je ranije posetio. Napisati program koji simulira istoriju pregledača tako što se učitavaju adrese posećenih sajtova (svaka u posebnom redu), a kada se učita red u kome piše `back` pregledač se vraća na poslednju posećenu stranicu.

Rešenje je krajnje jednostavno. Spisak posećenih sajtova čuvamo na steku. Naredbom `back` uklanjamo i ispisujemo sajt sa vrha steka (uz proveru da stek nije prazan).

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

int main() {
   stack<string> istorija;
```

```
   string linija;
   while (getline(cin, linija)) {
      if (linija == "back")
         if (!istorija.empty()) {
            cout << istorija.top() << endl;
            istorija.pop();
         } else {
            cout << "-" << endl;
         }
      else
         istorija.push(linija);
   }
   return 0;
}
```

U jeziku Python stek možemo simulirati pomoću najobičnije liste. Metodom `append` dodajemo element na kraj liste, a metodom `pop` skidamo element sa kraja.

```
import sys

istorija = []
for linija in sys.stdin:
    linija = linija.strip()
    if linija == "back":
        if istorija:
            print(istorija.pop())
        else:
            print("-")
    else:
        istorija.append(linija)
```

Problem: Napisati program koji sve učitane linije sa standardnog ulaza ispisuje u obratnom redosledu.

Jedno od mogućih rešenja je da se sve učitane linije smeste na stek, a da se zatim ispišu uzimajući jednu po jednu sa steka. Pošto stek funkcioniše po LIFO principu, redosled će biti obrnut (najkasnije dodata linija biće prva skinuta i ispisana, dok će prva postavljena linija biti skinuta i ispisana poslednja).

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

int main() {
   stack<string> s;
   string linija;
   while (cin >> linija)
      s.push(linija);
   while (!s.empty()) {
```

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

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, a nezatvorene zagrade. 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 skidamo je 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.

```
#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;
   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 ? "tacno" : "netacno") << endl;
      return 0;
}
```

Vrednost postfiksnog izraza

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

Postfiksna notacija se ponekad naziva i poljska notacija, a postfiksna notacija se ponekad naziva i obratna poljska notacija (engl. reverse polish notation, RPN) u čast logičara Jana Lukašijeviča koji ju je izumeo. Velika prednost postfiksno zapisanih izraza je to što im se vrednost veoma jednostavno izračunava uz pomoć steka. Kada naiđemo na broj postavljamo ga na vrh steka. Kada naiđemo na operator, skidamo dve vrednosti sa vrha steka, primenjujemo na njih odgovarajuću operaciju i rezultat postavljamo na vrh steka. Vrednost celog izraza se na kraju nalazi na vrhu steka. Npr. za izraz `34+5*` na stek postavljamo 3, zatim 4, nakon toga te dve vrednosti uklanjamo i postavljamo 7, zatim postavljamo 5 i na kraju uklanjamo 7 i 5 i menjamo ih sa 35.

```
#include <iostream>
#include <string>
#include <stack>
#include <cctype>
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;
}
```

Prevođenje potpuno zagradjenog 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.

Činjenica da je izraz potpuno zagrađen olakšava izračunavanje, jer nema potrebe da vodimo računa o prioritetu i asocijativnosti operatora.

Jedan način da se pristupi rešavanju problema je da se primeni induktivno-rekurzivni pristup. Obrada strukturiranog ulaza rekurzivnim funkcijama se naziva *rekurzivni spust* i detaljno se izučava u kursevima prevođenja programskih jezika. Definišemo rekurzivnu funkciju čiji je zadatak da prevede deo niske koji predstavlja ispravan infiksni izraz. On može biti ili broj, kada je prevođenje trivijalno jer se on samo prepiše na izlaz ili izraz u zagradama. U ovom drugom slučaju čitamo otvorenu zagradu, zatim rekurzivnim pozivom prevodimo prvi operand, nakon toga čitamo operator, zatim rekurzivnim pozivom prevodimo drugi operand, nakon toga čitamo zatvorenu zagradu i ispisujemo operator koji smo pročitali (on biva ispisan neposredno nakon prevoda svojih operanada).

Promenljiva `i` menja svoju vrednost kroz rekurzivne pozive. Primetimo da se ona prenosi po referenci tako da predstavlja i ulaznu i izlaznu veličinu funkcije. Zadatak funkcije je da pročita izraz koji počinje na poziciji i, da ga prevede u postfiksni oblik i da promenljivu `i` promeni tako da njena nova vrednost i' ukazuje na poziciju niske neposredno nakon izraza koji je preveden.

```
void prevedi(const string& izraz, int& i) {
```

```
  if (isdigit(izraz[i]))
     cout << izraz[i++];
  else {
    // preskačemo otvorenu zagradu
    i++;
    // prevodimo prvi operand
    prevedi(izraz, i);
    // pamtimo operator
    char op = izraz[i++];
    // prevodimo drugi operand
    prevedi(izraz, i);
    // preskačemo zatvorenu zagradu
    i++;
    // ispisujemo upamćeni operator
    cout << op;
  }
}

void prevedi(const string& izraz) {
  int i = 0;
  prevedi(izraz, i);
  cout << endl;
}
```

Da bismo se oslobodili rekurzije, potrebno je da upotrebimo stek. Ključna opaska je da se u stek okviru funkcije, pre rekurzivnog poziva za prevođenje drugog operanda pamti operator. Ovo nam sugeriše da nam je za nerekurzivnu implementaciju neophodno da održavamo stek na koji ćemo smeštati operatore. Kada naiđemo na broj prepisujemo ga na izlaz, kada naiđemo na operator stavljamo ga na stek, a kada naiđemo na zatvorenu zagradu skidamo i ispisujemo operator sa vrha steka.

```
void prevedi(const string& izraz) {
  stack<char> operatori;
  for (char c : izraz) {
    if (isdigit(c))
      cout << c;
    else if (c == ')') {
      cout << operatori.top();
      operatori.pop();
    } else if (jeOperator(c))
      operatori.push(c);
  }
  cout << endl;
}
```

Za vežbu vam ostavljamo da modifikujete ove funkcije tako da rezultat vrate u obliku niske umesto da ga ispisuju na standardni izlaz.

Vrednost potpuno zagradjenog infiksnog izraza

Problem: Dat je ispravan infiksni aritmetički 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.

Jedno rešenje je rekurzivno i u njemu bi se funkcija koja prevodi deo niske koji predstavlja ispravan izraz modifikovala tako da umesto prevođenja vraća njegovu vrednost.

```
int vrednost(string& izraz, int& i) {
  if (isdigit(izraz[i]))
     return izraz[i++] - '0';
  else {
    // preskačemo otvorenu zagradu
    i++;
    // čitamo i izračunavamo vrednost prvog operanda
    int op1 = vrednost(izraz, i);
    // pamtimo operator
    char op = izraz[i++];
    // čitamo i izračunavamo vrednost drugog operanda
    int op2 = vrednost(izraz, i);
    // preskačemo zatvorenu zagradu
    i++;
    // izračunavamo i vraćamo vrednost izraza
    return primeniOperator(op, op1, op2);
  }
}

int vrednost(string& izraz, int& i) {
  int i = 0;
  return vrednost(izraz, i);
}
```

Do rešenja možemo doći tako što iskombinujemo algoritam za prevođenje izraza u postfiksni oblik (pomoću steka na kom pamtimo operatore) i algoritma koji izračunava vrednost postfiksnog izraza (pomoću steka na kom pamtimo vrednosti). Kada bi se tokom prevođenja u postfiksni oblik na izlazu pojavio broj, taj broj ćemo postavljati na stek vrednosti. Kada bi se pojavio operator, sa vrha steka vrednosti ćemo skidati dva elementa, primenjivaćemo na njih odgovarajuću operaciju i rezultat vraćati na stek vrednosti.

Svaki podizraz je ili broj ili izraz u zagradama. Program ćemo organizovati tako da koristi dva steka (stek vrednosti i stek operatora). Nametnućemo uslov (invarijantu) da se neposredno nakon čitanja svakog broja i nakon svakog izraza u zagradama njihova vrednost nađe na vrhu steka vrednosti, pri čemu je stanje steka vrednosti ispod te vrednosti identično kao pre čitanja tog broja tj. izraza u

zagradama, a da je stanje na steku operatora identično kao pre početka čitanja tog broja tj. izraza. Takođe, neposredno nakon čitanja operatora on treba da se nađe na vrhu steka operatora, pri čemu se ne menja stek vrednosti niti prethodni sadržaj steka operatora.

Krećemo od praznih stekova. Da bi se ova invarijanta održala, kada naiđemo na broj postavljaćemo ga na stek vrednosti. Slično, čim pročitamo operator, postavljaćemo ga na stek operatora. Najinteresantniji slučaj je kada naiđemo na zatvorenu zagradu. Neposredno pre te zagrade pročitan je drugi operand koji je ili broj ili izraz u zagradama i znamo da se njegova vrednost nalazi na vrhu steka vrednosti. Na osnovu invarijante znamo da se operator koji predstavlja operaciju u zagradi koja je upravo zatvorena nalazi na vrhu steka operatora (jer je nakon čitanja drugog operanda stek operatora u identičnom stanju kao pre njegovog čitanja, a to je stanje u kojem je upravo pročitan operator, pa je postavljen na stek operatora). Neposredno pre tog operatora pročitan je prvi operand (koji je opet ili broj izraz u zagradama), tako da na osnovu invarijante znamo da se njegova vrednost nalazi kao druga sa vrha steka (ispod vrednosti drugog operanda). Zato skidamo dve vrednosti sa vrha steka vrednosti, skidamo operator sa vrha steka operatora, na osnovu tih elemenata izračunavamo novu vrednost i postavljamo je na stek vrednosti. Ovim smo postigli da se vrednost izraza u zagradama nalazi na vrhu steka vrednosti i da je stanje na stekovima ispod te vrednosti identično kao pre čitanja te otvorene zagrade. Pošto program čita jedan (isprvan) izraz u zagradama, kada se on pročita, znamo da se njegova vrednost nalazi na vrhu steka vrednosti.

```
int vrednost(const string& izraz) {
  stack<char> operatori;
  stack<int> vrednosti;
  for (char c : izraz) {
    if (isdigit(c)) {
      // stavljamo broj na vrh steka vrednosti
      vrednosti.push(c - '0');
    } else  if (c == ')') {
      // operator se nalazi na vrhu steka operatora
      char op = operatori.top(); operatori.pop();
      // operandi se nalaze na vrhu steka vrednosti
      int op2 = vrednosti.top(); vrednosti.pop();
      int op1 = vrednosti.top(); vrednosti.pop();
      // izračunavamo vrednost izraza u zagradi
      int v = primeniOperator(op, op1, op2);
      // postavljamo ga na stek vrednosti
      vrednosti.push(v);
    } else if (jeOperator(c)){
      // stavljamo operator na vrh steka operatora
      operatori.push(c);
    }
  }
  return vrednosti.top();
```

```
}
```

Prikažimo rad ovog algoritma na primeru izraza `(((3*5)+(7+(2*1)))*4)`.

```
stek operatora        stek vrednosti         ulaz
                                             (((3*5)+(7+(2*1)))*4)
                                              ((3*5)+(7+(2*1)))*4)
                                               (3*5)+(7+(2*1)))*4)
                                                3*5)+(7+(2*1)))*4)
                      3                          *5)+(7+(2*1)))*4)
    *                 3                           5)+(7+(2*1)))*4)
    *                 3 5                          )+(7+(2*1)))*4)
                      15                            +(7+(2*1)))*4)
    +                 15                             (7+(2*1)))*4)
    +                 15                              7+(2*1)))*4)
    +                 15 7                             +(2*1)))*4)
    + +               15 7                              (2*1)))*4)
    + +               15 7                               2*1)))*4)
    + +               15 7 2                              *1)))*4)
    + + *             15 7 2                               1)))*4)
    + + *             15 7 2 1                              )))*4)
    + +               15 7 2                                ))*4)
    +                 15 9                                   )*4)
                      24                                      *4)
    *                 24                                       4)
    *                 24 4                                      )
                      96
```

Prevođenje infiksnog u postfiksni izraz

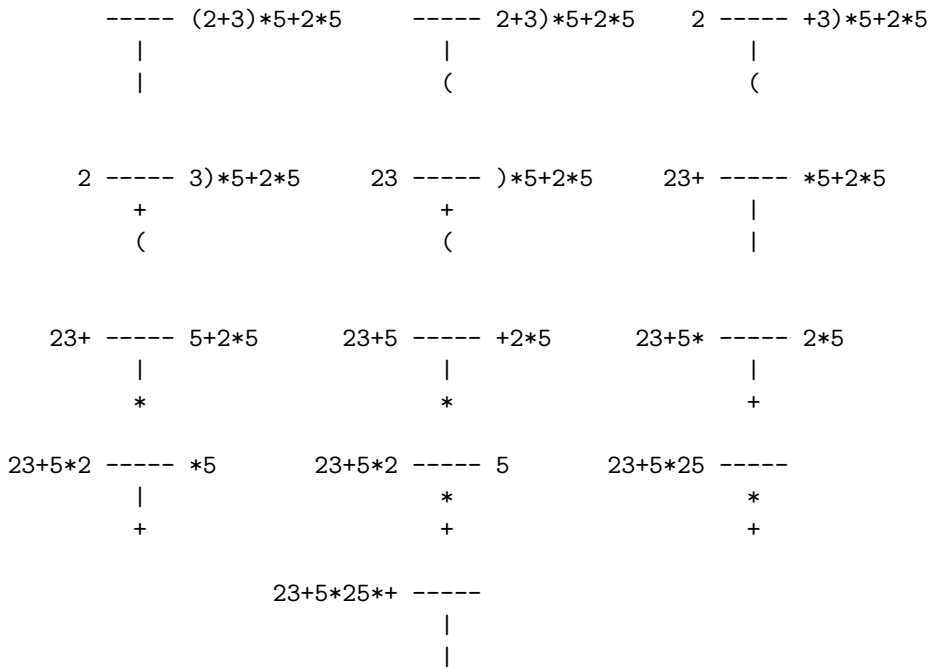
Problem: Napisati program koji vrši prevođenje ispravnog infiksno zadatog izraza u njemu ekvivalentan postfiksno zadati izraz. Pretpostaviti da su svi brojevi jednocifreni. Na primer, za izraz `2*3+4*(5+6)` program treba da ispiše `23*456+*+`.

Problem se rešava slično kao kod potpuno zagrađenih izraza, ali ovaj put se mora obraćati pažnja na prioritet i asocijativnost operatora. Rešenje se može napraviti rekurzivnim spustom, ali se time nećemo baviti u ovom kursu. Ključna dilema je šta raditi u situaciji kada se pročita `op2` u izrazu oblika `i1 op1 i2 op2 i3` gde su `i1`, `i2` i `i3` tri izraza (bilo broja bilo izraza u zagradama), a `op1` i `op2` dva operatora. U tom trenutku na izlazu će se nalaziti izraz `i1` preveden u postfiksni oblik i iza njega izraz `i2` preveden u postfiksni oblik, dok će se operator `op1` nalaziti na vrhu steka operatora. Ukoliko `op1` ima veći prioritet od operatora `op2` ili ukoliko im je prioritet isti, ali je asocijativnost leva, tada je potrebno prvo izračunavati izraz `i1 op1 i2` time što se operator `op1` sa vrha steka prebaci na izlaz. U suprotnom (ako `op2` ima veći prioritet ili ako je prioritet isti, a

asocijativnost desna) operator `op1` ostaje na steku i iznad njega se postavlja operator `op2`.

Ovo je jedan od mnogih algoritama koje je izveo Edsger Dejkstra i naziva se na engleskom jeziku *Shunting yard algorithm*, što bi se moglo slobodno prevesti kao algoritam sortiranja železničkih vagona. Zamislimo da izraz treba da pređe sa jednog na drugi kraj pruge. Na pruzi se nalazi sporedni kolosek (pruga je u obliku slova T i sporedni kolosek je uspravna crta). Delovi izraza prelaze sa desnog na levi kraj (zamislimo da idu po gornjoj ivici slova T). Brojevi uvek prelaze direktno. Operatori se uvek zadržavaju na sporednom koloseku, ali tako da se pre nego što operator uđe na sporedni kolosek sa njega na izlaz prebacuju svi operatori koji su višeg prioriteta u odnosu na tekući ili imaju isti prioritet kao tekući a levo su asocijativni. I otvorene zagrade se postavljaju na sporedni kolosek, a kada naiđe zatvorena zagrada sa sporednog koloseka se uklanjaju svi operatori do otvorene zagrade. Kada se iscrpi ceo izraz na desnoj strani, svi operatori sa sporednog koloseka se prebacuju na levu stranu. Jasno je da sporedni kolosek ima ponašanje steka, tako da implementaciju možemo napraviti korišćenjem steka na koji ćemo stavljati operatore.

Ilustrujmo ovu železničku analogiju jednim primerom.



Kada se algoritam razume, implementaciju je prilično jednostavno napraviti.

```
// prioritet operatora
int prioritet(char c) {
  if (c == '+' || c == '-')
```

```
    return 1;
  if (c == '*' || c == '/')
    return 2;
  throw "Nije operator";
}

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 višeg 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;
   return 0;
}
```

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`.

Na kraju, izračunavanje vrednosti izraza je opet kombinacija prevođenja u postfiksni oblik i izračunavanja vrednosti postfiksnog izraza.

```
// primenjuje datu operaciju na dve vrednosti na vrhu steka,
// zamenjujući 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 == ')') {
      // izračunavamo vrednost izraza u zagradi
      while (operatori.top() != '(')
        primeni(operatori, vrednosti);
    } else if (jeOperator(c)) {
       // obrađujemo sve prethodne operatore višeg prioriteta
       while (!operatori.empty() && jeOperator(operatori.top()) &&
              prioritet(operatori.top()) >= prioritet(c))
          primeni(operatori, vrednosti);
       operatori.push(c);
    }
  }

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

  return vrednosti.top();
}
```

Nerekurzivno brzo sortiranje

Problem: Implementiraj brzo sortiranje nerekurzivno.

Jedna veoma važna upotreba steka je za realizaciju rekurzije. Kao što znamo, tokom izvršavanja rekurzivnih funkcija, na stek se smeštaju vrednosti lokalnih promenljivih i argumenata svakog aktivnog poziva funkcije. Rekurziju uvek možemo ukloniti i umesto sistemskog steka možemo ručno održavati stek sa tim podacima.

Prikažimo ovu tehniku uklanjanja rekurzije na primeru nerekurzivne implementacije algoritma brzog sortiranja. Recimo i da je ova tehnika opšta i da se rekurzija uvek može eliminisati na ovaj način. Za razliku od toga, eliminisanje specifičnih oblika rekurzije (poput, na primer, repne) nije uvek primenljivo, ali kada jeste, dovodi do bolje memorijske (pa i vremenske) efikasnosti jer se ne koristi stek.

Na steku ćemo čuvati argumente rekurzivnih poziva funkcije sortiranja. Na početku je to par indeksa $(0, n-1)$. Glavna petlja se izvršava sve dok se stek ne isprazni i u njoj se obrađuje par indeksa koji se skida sa vrha steka. Umesto rekukrzivnih poziva njihove ćemo argumente postavljati na vrh steka i čekati da oni budu obrađeni u nekoj od narednih iteracija petlje. Primetimo da se argumenti drugog rekurzivnog poziva obrađuju tek kada se u potpunosti reši potproblem koji odgovara prvom rekurzivnom pozivu, što odgovara ponašanju funkcije kada je zaista implementirana rekurzivno.

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

int main() {
  // niz koji se sortira
  vector<int> a{3, 5, 4, 2, 6, 1, 9, 8, 7};
  // stek na kome čuvamo argumente rekurzivnih poziva
  stack<pair<int, int>> sortirati;
  // sortiranje kreće od obrade celog niza tj. pozicija (0, n-1)
  sortirati.push(make_pair(0, a.size() - 1));
  while (!sortirati.empty()) {
    // skidamo par (l, d) sa vrha steka
    auto p = sortirati.top();
    int l = p.first, d = p.second;
    sortirati.pop();
    // obrađujemo par (l, d) na isti način kao u rekurzivnoj implementaciji
    if (d - l < 1)
      continue;
    int k = l;
    for (int i = l+1; i <= d; i++)
```

```
      if (a[i] < a[l])
        swap(a[++k], a[i]);
    swap(a[k], a[l]);
    // umesto rekurzivnih poziva njihove argumente
    // postavljamo na stek
    sortirati.push(make_pair(l, k-1));
    sortirati.push(make_pair(k+1, d));
  }
  // ispisujemo sortirani niz
  for (int x : a)
    cout << x << endl;
  return 0;
}
```

DFS nerekurzivno

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

Oslobađanje od rekurzije i ovaj put teče na isti način kao u prethodnom slučaju (na stek argument glavnog poziva i dok se stek ne isprazni na umesto rekurzivnih poziva na stek stavljamo njihove argumente).

```
#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;
}
```

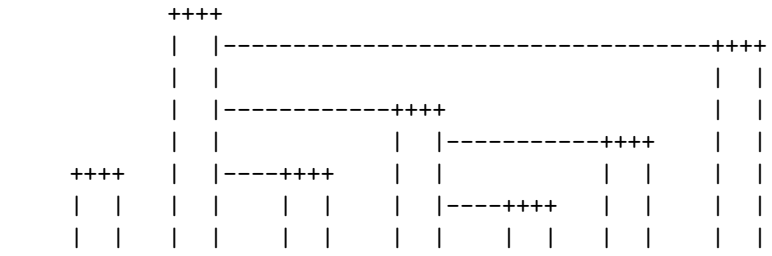
U nastavku ćemo razmotriti još jednu familiju algoritama koji se implementiraju uz pomoć steka.

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 najbliža 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.

Naglasimo da nema nikakve značajne razlike da li se traži najbliži veći prethodnik, najbliži manji prethodnik, najbliži prethodnik koji je manji ili jednak ili najbliži prethodnik koji je veći ili jednak. Isti algoritam se može primeniti na bilo koju relaciju poretka.

Jedna lepa interpretacija prethodnog zadatka je sledeća. Neka niz a sadrži visine zgrada poređanih duž jedne ulice. Betmen želi da postavi horizontalno uže na svaku zgradu koje će ga dovesti do neke prethodne zgrade. Za svaku zgradu je potrebno ispisati visinu njoj prethodne zgrade do koje će voditi horizontalno uže sa tekuće zgrade. Ako su visine zgrada redom $2, 7, 2, 4, 1, 3, 6$, tada užad vode do zgrada čije su visine $-, -, 7, 7, 4, 4, 7$.



Naivno rešenje zasnovano na linearnoj pretrazi u kom bi se za svaki element redom unazad tražio njemu najbliži veći prethodnik bi bilo veoma neefikasno (složenost bi mu bila $O(n^2)$ i taj najgori slučaj bi se javljao kod neopadajućih nizova).

```
for (int i = 0; i < n; i++) {
  bool nadjen = false;
  for (int j = i-1; j >= 0; j--)
```

```
    if (a[j] > a[i]) {
       cout << a[j] << endl;
       nadjen = true;
       break;
    }
  if (!nadjen)
    cout << "-" << endl;
}
```

Pokušajmo da efikasniji algoritam konstruišemo induktivno-rekurzivnom konstrukcijom.

- Bazu čini jednočlan niz i sigurni smo da početni element nema prethodnika većeg od sebe (jer uopšte nema prethodnika). Sa prve zgrade nije moguće razvući kanap.
- Pretpostavimo da za svaki element niza dužine k umemo da odredimo najbližeg većeg prethodnika i razmotrimo kako bismo odredili najbližeg većeg prethodnika poslednjeg elementa u nizu (elementa na poziciji k). Analizu krećemo od direktnog prethodnika tekućeg elementa (za poziciju k, analizu krećemo od pozicije $k-1$). Ako je taj element strogo veći od tekućeg, on mu je najbliži veći prethodnik (i do njega razvlačimo kanap). U suprotnom rekurzivno određujemo njegovog najbližeg većeg prethodnika i tako dobijeni element upoređujemo sa tekućim elementom. Taj postupak ponavljamo sve dok ne dođemo do elementa koji je veći od tekućeg elementa ili do situacije u kojoj neki element manji ili jednak od tekućeg nema većih prethodnika.

Ovaj algoritam možemo rekurzivno izraziti na sledeći način.

```
// funkcija vraća poziciju najbližeg većeg prethodnika elementa
// a[k], tj. -1 ako a[k] nema većih prethodnika
int najblizi_veci_prethodnik(int a[], int k) {
  // početni element nema prethodnika
  if (k == 0)
    return -1;
  // pretragu počinjemo od neposrednog prethodnika tekućeg elementa
  int p = k-1;
  // dok god je prethodnik unutar niza, ali nije strogo veći od tekućeg
  while (p != -1 && a[p] <= a[k])
    // prelazimo na analizu njegovog prethodnika
    p = najblizi_veci_prethodnik(a, p);
  // petlja se završila ili kada je p=-1 pa element nema prethodnika
  // ili kada je a[p] > a[k], pa je a[p] najbliži veći prethodnik elementu a[k]
  return p;
}

int main() {
  // učitavamo elemente niza
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // za sve elemente
  for (int k = 0; k < n; k++) {
    // nalazimo poziciju najbližeg većeg prethodnika
    int p = najblizi_veci_prethodnik(a, k);
    // ispisujemo njegovu vrednost ili - ako ne postoji
    if (p != -1)
       cout << a[p] << endl;
    else
       cout << "-" << endl;
  }
  return 0;
}
```

Ova implementacija je neefikasna jer dolazi do višestrukog pozivanja funkcije za iste argumente. Stvar se može popraviti tako što primenimo tehniku dinamičkog programiranja (o njoj je bilo reči na kursu P2, a mnogo više reči o njoj će biti u nastavku ovog kursa) i pamtimo vrednosti rekurzivnih poziva u nizu.

```
// na poziciji k se nalazi vrednost najbližeg
// većeg prethodnika elementa a[k]
vector<int> najblizi_veci_prethodnik(n);

// početni element nema prethodnika
najblizi_veci_prethodnik[0] = -1;
// analiziramo redom naredne elemente
for (int k = 1; k < n; k++) {
  // pretragu počinjemo od neposrednog prethodnika tekućeg elementa
  int p = k-1;
  // dok god je prethodnik unutar niza, ali nije strogo veći od tekućeg
  while (p != -1 && a[p] <= a[k])
    // prelazimo na analizu njegovog prethodnika
    p = najblizi_veci_prethodnik[p];

  // za buduće potrebe pamtimo u nizu poziciju najbližeg većeg
  // prethodnika tekućeg elementa na poziciji k (tj. -1 ako ne postoji)
  najblizi_veci_prethodnik[k] = p;

  // ispisujemo vrednost najbližeg većeg prethodnika
  // ili - ako tekući element nema većih prethodnika
  if (p != -1)
    cout << a[p] << endl;
  else
    cout << "-" << endl;
}
```

Možda iznenađujuće, jer implementacija sadrži ugnežđene petlje, složenost najgoreg slučaja prethodne implementacije je linearna tj. $O(n)$, no to nije uopšte jednostavno dokazati. Ključni argument koji će nam pomoći i da uprostimo implementaciju i da lakše analiziramo složenost je to da tokom izvršavanja mnogi elementi niza prestaju da budu kandidati za najbliže veće prethodnike, pa se kroz njih nikada ne iterira tokom unutrašnje petlje `while`. Naime, kada se nakon nekog elementa x pojavi neki element y koji je veći od njega ili mu je jednak, tada element x prestaje da bude kandidat za najbližeg većeg prethodnika svih elemenata koji se javljaju u nizu iza y (jer će y uvek biti bliži prethodnik elementima iza sebe od x, koji će ako su strogo manji od x biti sigurno strogo manji i od y, jer je $x \leq y$). Zaista, ako se iza neke zgrade pojavi zgrada iste visine ili viša od nje ona će zakloniti prethodnu zgradu i nijedan kanap nadalje neće moći biti razvučen do te niže zgrade. Stoga se tokom petlje `while` iterira kroz relativno mali broj kandidata. U svakom trenutku elementi koji su kandidati za najbliže manje predhodnike čine opadajući niz. Svaki naredni element eliminiše one tekuće kandidate koji su manji od njega ili su mu jednaki (svaka nova zgrada sakriva zgrade koje su niže od nje ili su joj iste visine). 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.

Pošto je niz kandidata opadajući, kandidati koji bivaju eliminisani (tj. zaklonjeni) se mogu nalaziti samo na kraju niza potencijalnih kandidata. Ovo ukazuje na to da se tekući kandidati za najbližeg većeg prethodnika mogu čuvati na steku i da nije potrebno istovremeno čuvati čitav niz `najblizi_veci_prethodnik`, već samo one kandidate koji nisu eliminisani jer su zaklonjeni nekim većim ili jednakim elementom (na steku možemo čuvati bilo pozicije tih kandidata, bilo njihove vrednosti). U trenutku kada se obrađuje element na poziciji j na steku se nalaze vrednosti ili pozicije i takve da je a_i maksimum elemenata niza a na pozicijama iz intervala $[i, j)$. Ovim dobijamo implementaciju veoma sličnu prethodnoj, i sa istom asimptotskom memorijskom složenošću najgoreg slučaja. Ipak, za mnoge ulaze, zauzeće memorije će biti manje (ako na steku čuvamo vrednosti, ne moramo čak ni da pamtimo ceo originalni niz).

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

using namespace std;

int main() {
  int n;
  cin >> n;
  // stek na kojem čuvamo kandidate za najbliže veće prethodnike
  stack<int> s;
  // obrađujemo 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 tekući element zaklonio
    while (!s.empty() && s.top() <= x)
        s.pop();
    if (s.empty())
       // ako na steku nema preostalih kandidata, tada
       // tekući element nema većih prethodnika
       cout << "-" << endl;
    else
       // element na vrhu steka je najbliži veći prethodnik
       // tekućem
       cout << s.top() << endl;

    // tekući element postaje kandidat za najbližeg većeg
    // prethodnika narednim elementima
    s.push(x);
  }
  return 0;
}
```

Prikažimo rad ovog algoritma na primeru niza `3 1 2 5 3 1 4`.

Na početku je stek prazan.

`Stek:`

Pošto je stek prazan, `3` nema manjih prethodnika. Nakon toga se on dodaje na stek.

`Stek: 3`

Pošto je element `1` manji od elementa `3` koji je na vrhu steka, element `3` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `1`.

`Stek: 3 1`

Kada na red dođe element `2` sa steka se skida element `1` koji je manji od njega i pronalazi se da je najbliži veći prethodnik broja `2` broj `3`. Element `1` zaista prestaje da bude kandidat svim kasnijim elementima (zgrada visine `2` zaklanja zgradu visine `1`) i element `2` koji se postavlja na stek preuzima njegovu ulogu.

`Stek: 3 2`

Kada na red dođe element `5` sa steka se skidaju elementi `3` i `2`. Elementi `3` i `2` zaista prestaju da budu kandidati svim kasnijim elementima (zgrada visine `5` zaklanja zgrade visine `3` i `2`). Pošto je stek prazan, element `5` nema većih prethodnika. Element `5` se postavlja na stek.

`Stek: 5`

Pošto je element `3` manji od elementa `5` koji je na vrhu steka, element `5` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `3`.

`Stek: 5 3`

Pošto je element `1` manji od elementa `3` koji je na vrhu steka, element `3` je njegov najbliži veći prethodnik. Nakon toga se na stek dodaje `1`.

`Stek: 5 3 1`

Kada na red dođe element `4` sa steka se skidaju elementi `1` i `3` koji su manji od njega i pronalazi se da je najbliži veći prethodnik broja `4` broj `5`. Elementi `3` i `1` zaista prestaju da budu kandidati svim kasnijim elementima (zgrada visine `4` zaklanja zgrade visine `1` i `3`) i element `4` koji se postavlja na stek preuzima njihovu ulogu.

Raspon akcija

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

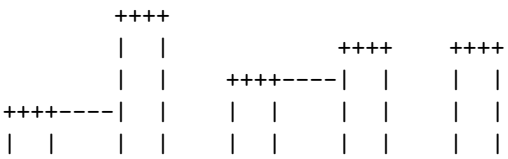
Ovaj zadatak je veoma sličan prethodnom, jedino što za svaki element ne treba da znamo vrednost, već poziciju njemu najbližeg prethodika koji je strogo veći od njega (tj. -1 ako takav prethodnik ne postoji). Sve elemente ćemo na početku učitati u niz, a na steku ćemo pamtiti pozicije (a ne vrednosti)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

```
stack<int> s;
for (int i = 0; i < n; i++) {
  while (!s.empty() && a[s.top()] <= a[i])
    s.pop();
  cout << (s.empty() ? i + 1 : i - s.top()) << endl;
  s.push(i);
}
```

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`.

Ovaj zadatak ima veoma sličnu interpretaciju kao onaj sa nalaženjem najbližih većih prethodnika, jedino što se kanapi sada šire nadesno. Na primer, ako je niz `1 4 2 3 3`, kanapi su raspoređeni kao na narednoj slici.



Primetimo da se ovde traže pozicije, a ne vrednosti sledbenika, ali to ništa značajno ne menja u postupku rešavanja, kao ni to da li se radi o većim ili manjim sledbenicima (ponovo je algoritam isti za bilo koju relaciju poretka).

Jedan način da se zadatak reši je da se koristi praktično isti postupak kao u prethodnom slučaju, ali da se niz obilazi zdesna na levo (u tom redosledu obilaska prethodnik je isto što i sledbenik u obilasku sleva nadesno).

Ipak, pokazaćemo i direktno rešenje nastalo modifikacijom tehnike koju smo videli prilikom traženja najbližih većih prethodnika, koje nam može biti korisno i u nekim drugim kontekstima. Invarijanta petlje će biti to da se na steku (u rastućem redosledu) nalaze pozicije svih elemenata čiji najbliži veći sledbenik još nije određen tj. oni elementi iza kojih još nije pronađen neki veći element. Elementi čije su pozicije na steku će biti uvek u nerastućem redosledu. 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 osnovu invarijante znamo da se iza tih elemenata nije ranije pojavio element veći od njih, pa pošto je tekući element veći od njih upravo njima najbliži veći sledbenik). Za elemente sa steka koji su veći ili jednaki od tekućeg znamo da nisu u delu niza pre tekućeg elementa imali veće sledbenike, a pošto su oni veći ili jednaki od tekućeg elementa njima ni on nije veći sledbenik, tako da oni ostaju na steku, a na vrh steka se postavlja pozicija tekućeg elementa, jer ni njemu još nismo pronašli većeg sledbenika.

Nakon obrade celog niza na steku su ostali elementi koji nemaju većeg sledbenika.

Pošto pozicije sledbenika ne saznajemo u redosledu u kojem je potrebno ispisati ih, moramo ih pamtiti u pomoćni niz koji ćemo na kraju ispisati.

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

int main() {
  // učitavamo elemente niza
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];
```

```
  // pozicije najbližih većih sledbenika
  vector<int> p(n);

  // stek na kome se nalaze elementi čiji najbliži veći sledbenik
  // još nije obrađen
  stack<int> s;
  // obilazimo sve elemente niza sleva nadesno
  for (int i = 0; i < n; i++) {
    // skidamo sa steka sve element kojima je najbliži
    // veći sledbenik element na poziciji i
    while (!s.empty() && a[s.top()] < a[i]) {
      p[s.top()] = i;
      s.pop();
    }
    // elementu i još ne znamo poziciju najbližeg većeg sledbenika
    s.push(i);
  }

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

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

  return 0;
}
```

Razmotrimo rad algoritma na primeru niza `1 4 2 3 3`.

Stek je na početku prazan.

`Stek: Niz p: ? ? ? ? ?`

Posle toga obrađujemo element `1` i on nije najbliži veći sledbenik ni jednog elementa (jer je stek prazan). Pošto ni za element `1` još ne znamo najbližeg većeg sledbenika, postavljamo njegovu poziciju na stek.

`Stek: 0 Niz p: ? ? ? ? ?`

Posle toga obrađujemo element `4`. Pošto je on veći od elementa `1` čija se pozicija nalazi na vrhu steka, on je najbliži veći sledbenik elementu na poziciji `0`. Sa steka skidamo poziciju `0` i upisujemo poziciju `1` elementa `4`, jer za nju još ne znamo poziciju najbližeg većeg sledbenika.

`Stek: 1 Niz p: 1 ? ? ? ?`

Posle toga obrađujemo element `2`. Pošto je on manji od elementa `4` čija se

pozicija nalazi na vrhu steka, još ne znamo rešenje za element 4, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju 2 elementa 2, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

```
Stek: 1 2 Niz p: 1 ? ? ? ?
```

Posle toga obrađujemo element 3. Pošto je on veći od elementa 2 čija se pozicija nalazi na vrhu steka, on je najbliži veći sledbenik elementu na poziciji 2. Tada sa steka skidamo poziciju 2. Pošto je on manji od elementa 4 čija se pozicija nalazi na vrhu steka, još ne znamo rešenje za element 4, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju 3 elementa 3, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

```
Stek: 1 3 Niz p: 1 ? 3 ? ?
```

Posle toga obrađujemo drugi element 3. Pošto je on jednak elementu 3 čija se pozicija nalazi na vrhu steka, još ne znamo rešenje za element 3, pa njegova pozicija ostaje na steku. Na vrh steka dodajemo poziciju 4 elementa 3, jer ni za nju ne znamo poziciju najbližeg većeg sledbenika.

```
Stek: 1 3 4 Niz p: 1 ? 3 ? ?
```

Pošto smo stigli do kraja niza, elementi koji su ostali na steku nemaju većih sledbenik i na te pozicije u nizu p upisujemo dužinu niza 5.

```
Stek: Niz p: 1 5 3 5 5
```

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 untar segmenta strogo manji od elemenata na njihovim krajevima.

I ovom slučaju je, naravno, direktno rešenje prilično neefikasno. Bolje rešenje možemo dobiti tako što prilagodimo tehniku određivanja pozicija najbližih prethodnika strogo većih od datog elementa.

Za svaku poziciju j ćemo odrediti broj segmenata koji zadovoljavaju dati uslov, a kojima je desni kraj na poziciji j. Ukupan broj takvih segmenata će biti zbir brojeva segmenata za svako j.

Da bi segment oivičen maksimumima mogao da počne na poziciji i neophodno je da važi da je a_i maksimum svih elemenata na pozicijama u intervalu $[i, j)$. Pretpostavimo da znamo sve pozicije $i_1 < i_2 < \ldots < i_k$ koje zadovoljavaju takav uslov (znamo da su to jedini preostali kanidati za leve krajeve segmenata koji su maksimalni). Elementi na tim pozicijama su opadajući tj. važi $a_{i_1} > a_{i_2} > \ldots > a_{i_k}$. Pozicija $j-1$ sigurno je poslednja u nizu (jer je a_{j-1} maksimum segmenta određenog jednočlanim intervalom pozicija $[j-1, j)$). Dodatno, svi elementi između dve susedne pozicije su manji i od prve i od druge (jer bi u suprotnom poslednji od njih koji je veći od druge pozicije bio veći od svih elemenata iza sebe

i morao bi da se nađe u nizu). Na primer, ako je prefiks niza ispred pozicije j niz $8, 4, 6, 3, 7, 2, 5$, karakterističi maksimumi su elementi $8, 7, 5$ (8 je maksimum celog niza, 7 je maksimum dela niza koji započinje, i isto važi i za 5, oni čine opadajući niz i između bilo koja dva od njih nalaze se elementi strogo manji od njih).

Ako za neku takvu poziciju i važi i da je $a_i < a_j$, onda interval $[i, j]$ sigurno zadovoljava uslov zadatka (svi elementi u unutrašnjosti su manji od elemenata na krajevima). Ako je $a_i > a_j$, to ne mora biti slučaj, međutim, analizom našeg niza karakterističnih maksimuma, možemo identifikovati tražene segmente. Analizu započinjemo zdesna i za svaki element a_i koji je manji od a_j znamo da smo pronašli segment koji zadovoljava uslove zadatka. Prvi maksimum zdesna koji je veći od a_j takođe određuje segment koji zadovoljava uslove zadatka. Ako je to a_{j-1}, uslovi su trivijalno ispunjeni, jer između a_{j-1} i a_j nema drugih elemenata niza. Ako je to neki $a_{i_m} > a_j$ takav da je $a_{i_{m+1}} < a_j$, uslovi su opet ispunjeni, jer su svi elementi na pozicijama od i_{m+1} do $j-1$ manji od a_j (jer su manji ili jednaki $a_{i_{m+1}}$), što takođe važi i za elemente na pozicijama od $i_m + 1$ do $i_{m+1} - 1$, jer smo već ranije konstatovali da su svi elementi između dva susedna maksimuma strogo manja od oba od njih, pa su manja i od $a_{i_{m+1}}$ koji je strogo manji od a_j. Ako je $a_{i_m} > a_j$ i $a_{i_{m+1}} > a_j$, jasno je da je element $a_{i_{m+1}}$, između pozicija i_m i j, a da je veći od minimuma elemenata na tim pozicijama.

Dakle, pod pretpostavkom da je poznat niz karakterističnih maksimuma za prefiks niza koji se završava na poziciji j, možemo jednostavno da odredimo broj segmenata koji se završavaju na toj poziciji i koji zadovoljavaju uslove zadatka. Pokažimo i da je niz karakterističnih maksimuma moguće veoma lako održavati i inkrementalno ga ažurirati kada se prefiks produžava jednim po jednim elementom zdesna. Naime, ako prefiks proširujemo elementom a_j, tada ni jedan element a_i koji je ranije bio maksimum i manji je od a_j više nije maksimum (zato što prefiks sada na svom desnom kraju uključuje i element a_j koji ranije nije bio uključen). Svi raniji maksimumi a_i koji su veći od a_j ostaju da budu maksimumi. Na kraju, a_j je takođe maksimum.

Na osnovu svega rečenog, možemo formulisati jednostavan algoritam za rešavanje ovog zadatka. Primetimo da se niz karakterističnih maksimuma ponaša kao stek (elemente dodajemo na desni kraj i obrađujemo ih i uklanjamo sa tog desnog kraja). Prvi element niza postavljamo na stek. Nakon toga, obrađujemo jedan po jedan element niza a_j, od drugog pa do poslednjeg. Skidamo jedan po jedan element sa vrha steka koji su manji od a_j i za svaki od njih uvećavamo broj segmenata i to dok se stek ne isprazni ili dok se na vrhu steka ne nađe element veći od a_j. U ovom drugom slučaju broj segmenata uvećavamo za još jedan. Nakon toga element a_j postavljamo na vrh steka.

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

using namespace std;
```

```cpp
int main() {
  int n;
  cin >> n;
  // ukupan broj segmenata oivičenih maksimumima
  int brojSegmenata = 0;
  // na steku se čuvaju svi elementi ai takvi da je ai maksimum
  // elemenata na pozicijama [i, j), gde je i pozicija elementa ai
  stack<int> s;
  for (int j = 0; j < n; j++) {
    int aj;
    cin >> aj;
    while (!s.empty() && s.top() < aj) {
      // ako je ai = s.top() i ako je i njegova pozicija,
      // tada je segment na pozicijama [i, j] oivičen maksimumima
      brojSegmenata++;
      s.pop();
    }
    if (!s.empty())
      // ako je ai = s.top() i ako je i njegova pozicija,
      // tada je segment na pozicijama [i, j] maksimumima
      brojSegmenata++;

    // svi preostali elementi na steku su i dalje maksimumi
    // segmenta [ai, j+1), pa treba da ostanu na steku
    // važi i da je aj maksimum segmenta [j, j+1),
    // pa ga treba dodati na stek
    s.push(aj);
  }
  cout << brojSegmenata << endl;
  return 0;
}
```

Prikažimo rad algoritma na narednom primeru niza `8 4 6 3 7 2 5`.

```
Stek:
Stek: 8
Stek: 8 4     - pronađen segment [8, 4]
Stek: 8       - pronađen segment [4, 6]
Stek: 8 6     - pronađen segment [8, 4, 6]
Stek: 8 6 3   - pronađen segment [6, 3]
Stek: 8 6     - pronađen segment [3, 7]
Stek: 8       - pronađen segment [6, 3, 7]
Stek: 8 7     - pronađen segment [8, 4, 6, 3, 7]
Stek: 8 7 2   - pronađen segment [7, 2]
Stek: 8 7     - pronađen segment [2, 5]
Stek: 8 7 5   - pronađen segment [7, 2, 5]
```

Najveći pravougaonik u histogramu

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

Za svaki stubić u histogramu potrebno je da odredimo poziciju prvog stubića levo od njega koji je strogo manji od njega (ili poziciju -1 neposredno ispred početka, ako takav stubić ne postoji) i poziciju prvog stubića desno od njega koji je strogo manji od njega (ili poziciju n, neposredno iza kraja niza, ako takav stubić ne postoji). Ako te pozicije obeležimo sa l i d, onda možemo zaključiti da je najveća površina pravougaonika koji sadrži tekući stubić $(d - l - 1) \cdot h[i]$ (taj pravougaonik je visine $h[i]$, a pod pretpostavkom da je svaki stubić jedinične širine, ukupna širina mu je $d - l - 1$). Pošto su svi stubići od pozicije $l + 1$ do $d - 1$ visine veće ili jednake od $h[i]$, takav pravougaonik je moguće upisati histogram. Jasno je i da ne može da postoji pravougaonik koji bi bio viši od ovoga (jer je i-ti stubić visine $h[i]$ i viši pravougaonik bi njega prevazišao) niti da postoji pravougaonik koji bi bio širi od ovoga (jer čak i da postoje stubići na pozicijama l i d, oni su niži od $h[i]$ i pravougaonik visine $h[i]$ ne bi mogao da se proširi i upiše u njih).

Naivan način da se za svako i odrede l i d je da za svaki stubić i iznova puštamo petlje koje nalaze odgovarajuće ivične stubiće.

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

int main() {
  int n;
  cin >> n;
  vector<int> h(n);
  for (int i = 0; i < n; i++)
    cin >> h[i];

  int max = 0;
  for (int i = 0; i < n; i++) {
    int l = i;
    while (l >= 0 && h[l] >= h[i])
      l--;
    int d = i;
    while (d < n && h[d] >= h[i])
      d++;
    int P = (d - l - 1) * h[i];
    if (P > max)
      max = P;
  }
  cout << max << endl;

  return 0;
```

```
}
```

Ovaj algoritam je prilično neefikasan (složenost najgoreg slučaja mu je očigledno $O(n^2)$ i ona nastupa, na primer, kada su svi stubići jednake visine).

Već smo razmatrali probleme određivanja najbližeg manjeg prethodnika i najbližeg manjeg sledbenika za svaki element niza i videli smo da oba problema možemo rešiti u vremenu $O(n)$ (zapravo, tražili smo veće prethodnike i sledbenike, ali to ništa suštinski ne menja). Jedno moguće rešenje je da u jednom prolazu odredimo pozicije najbližih manjih prethodnika svakog elementa, u drugom pozicije najbližih većih sledbenika za svaki element niza i u trećem da na osnovu tih pozicija izračunamo maksimalne površine pravougaonika za svaki stubić.

Međutim, pokazaćemo da se zahvaljujući sličnosti algoritama za određivanje najbližeg manjeg prethodnika i najbližeg manjeg sledbenika sve može uraditi samo u jednom prolazu. Na stek ćemo postavljati one stubiće za koje još ne znamo poziciju najbližeg manjeg sledbenika (i za koje još nismo odredili površinu maksimalnog pravougaonika koji određuju). Stubiće obilazimo s leva na desno i za svaki stubić h_d na koji naiđemo sa vrha steka redom skidamo i obrađujemo jedan po jedan stubić h koji je strogo veći od njega. Svakom od tih stubića h stubić h_d je najbliži manji sledbenik. Ako ispod h na steku postoji neki stubić, on je najbliži manji ili jednak prethodnik stubiću h. Da bismo našli najbližeg strogo manjeg prethodnika potrebno je da nastavljamo da sa steka skidamo sve stubiće čija je visina jednaka h. Stubić h_l koji se nalazi na vrhu steka nakon toga je najbliži strogo manji prethodnik stubiću h. Ako takvog stubića nema, stubić h nema manjih prethodnika i histogram koji mu odgovara se može širiti skroz do levog kraja histograma (tada je $l = -1$). Površina maksimalnog pravougaonika koji uključuje i stubić h je $h \cdot (d - l - 1)$. Primetimo da se za stubiće jedanke visine stubiću h koji su skinuti sa steka ne računa površina pravougaonika - za tim nema potrebe jer je ona jednaka površini pravougaonika određenog stubićem h. Kada se stubići sa vrha steka obrade (kada se stek isprazni ili kada se na njegovom vrhu nađe stubić h_l koji je manji ili jednak od h_d), na vrh steka se stavlja h_d.

Naglasimo i da je po dolasku do kraja niza, potrebno još obraditi sve stubiće koji su ostali na vrhu steka (to će biti stubići za koje ne postoji strogo manji sledbenik). Njihova obrada teče po istom prinicipu kao i ranije, osim što se za vrednost pozicije d uzima n. Zato je u implementaciji poželjno objediniti tu završnu obradu sa prvom fazom algoritma.

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

int main() {
  int n;
```

```
  cin >> n;
  vector<int> a(n);
  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++) {
    while (!s.empty() && (d == n || a[s.top()] > a[d])) {
      int h = a[s.top()];
      s.pop();
      while (!s.empty() && a[s.top()] == h)
          s.pop();
      int l = s.empty() ? -1 : s.top();
      int P = (d - l - 1) * h;
      if (P > max)
         max = P;
    }
    if (d < n)
      s.push(d);
  }
  cout << max << endl;
  return 0;
}
```

Prikažimo rad algoritma na jednom primeru.

```
3 5 5 8 6 4 9 2 4

Stek: 0                    3
Stek: 0 1                  3 5
Stek: 0 1 2                3 5 5
Stek: 0 1 2 3              3 5 5 8
6 na poziciji 4 je najbliži strogo manji sledbenik za 8
Stek: 0 1 2                3 5 5
5 na poziciji 2 je najbliži strogo manji prethodnik za 8
P = 8
Stek: 0 1 2 4              3 5 5 6
4 na poziciji 5 je najbliži strogo manji sledbenik za 6
Stek: 0 1 2                3 5 5
5 na poziciji 2 je najbliži strogo manji prethodnik za 6
P = 12
4 na poziciji 5 je najbliži strogo manji sledbenik za 5
Stek: 0 1                  3 5
Stek: 0                    3
3 na poziciji 0 je najbliži strogo manji pretodnik za 5
P = 20
```

```
Stek: 0 5                   3 4
Stek: 0 5 6                 3 4 9
2 na poziciji 7 je najbliži strogo manji sledbenik za 9
Stek: 0 5                   3 4
4 na poziciji 5 je najbliži strogo manji prethodnik za 9
P = 9
2 na poziciji 7 je najbliži strogo manji prethodnik za 4
Stek: 0                     3
3 na poziciji 0 je najbliži strogo manji prethodnik za 4
P = 24
2 na poziciji 7 je najbliži strogo manji sledbenik za 3
Stek:
3 nema strogo manjeg prethodnika
P = 21
Stek: 7                     2
Stek: 7 8                   2 4
4 nema najbližeg strogo manjeg sledbenika
Stek: 7                     2
2 na poziciji 7 je najbliži strogo manji prethodnik za 4
P = 4
2 nema najbližeg strogo manjeg sledbenika
Stek:
2 nema najbližeg strogo manjeg prethodnika
P = 18


maxP = 24
```

Red pomoću dva steka

Problem: Implementiraj funkcionalnost reda korišćenjem dva steka.

```cpp
#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;
}
```

Redovi

Red predstavlja kolekciju podataka u koju se podaci dodaju po FIFO principu - element se dodaje na kraj i skida samo na početka reda.

U jeziku C++ red se realizuje klasom `queue<T>` gde `T` predstavlja tip elemenata u redu. 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

Prikažimo upotrebu reda kroz nekoliko interesantnih primera.

Nerekurzivni BFS

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

```
#include <iostream>
#include <vector>
#include <queue>
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);
  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;
}
```

Maksimalni zbir segmenta dužine k

Problem: Učitava se n brojeva. Napiši program koji određuje segment uzastopnih k elemenata sa najvećim zbirom. Voditi računa o zauzeću memorije.

Najjednostavniji način bi bio da se svi elementi učitaju u niz i da se zatim određuju zbirovi segmenata dužine k. Zbirove, naravno, možemo računati inkrementalno. Dva susedna segmenta dužine k imaju zajedničke sve elemente osim prvog elementa levog i poslednjeg elementa desnog segmenta. Dakle, da bismo izračunali zbir narednog segmenta od zbira prethodnog segmenta treba da oduzmemo njegov početni element i da na zbir dodamo završni element

novog segmenta. Da bismo ovo mogli realizovati nije nam neophodno da čuvamo istovremeno sve elemente već samo elemente tekućeg segmenta dužine k. Pošto se uklanjaju početni elementi segmenta, a segment se proširuje završnim elementima najbolje je elemente čuvati u redu.

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

using namespace std;

int main() {
  // broj elemenata niza i duzina segmenta
  int n, k;
  cin >> n >> k;

  // red u kojem u svakom trenutku cuvamo tekuci segment
  queue<double> q;

  // ucitavamo prvi segment duzine k, smestamo elemente u red i
  // racunamo mu zbir
  double zbir = 0.0;
  for (int i = 0; i < k; i++) {
    double x; cin >> x;
    q.push(x);
    zbir += x;
  }

  // trenutna maksimalna suma segmenta i indeks njenog pocetka
  int maxPocetak = 0;
  double maxZbir = zbir;

  for (int i = 1; i <= n-k; i++) {
    // ucitavamo naredni element
    double x; cin >> x;
    // azuriramo zbir
    zbir = zbir - q.front() + x;
    // menjamo "najstariji" element u redu
    q.pop(); q.push(x);
    // ako je potrebno, azuriramo maksimum
    if (zbir >= maxZbir) {
      maxZbir = zbir;
      maxPocetak = i;
    }
  }

  // ispisujemo pocetak poslednjeg segmenta sa maksimalnom sumom
  cout << maxPocetak << endl;

  return 0;
```

```
}
```

Maksimalna bijekcija

Problem: Dat je konačni skup A i funkcija $f : A \to A$. Pronaći maksimalnu kardinalnost skupa $S \subseteq A$, takva da je restrikcija f na S bijekcija.

Ovo je zapravo pravi algoritamski problem u kome su redovi samo pomoćno sredstvo u implementaciji. Da bismo utvrdili da je funkcija bijekcija na nekom konačnom skupu ona mora biti surjekcija na tom skupu i za svaki element skupa mora postojati bar neki element skupa koji se slika u njega. Dakle, ako postoji neki element skupa u koji se ni jedan element ne slika, on ne može biti deo skupa u kom je f bijekcija. Njega možemo ukloniti iz skupa, time redukovati dimenziju problema i do rešenja doći induktivno-rekurzivnom konstrukcijom. Bazu čini slučaj kada se u svaki element skupa slika bar neki element skupa (do ovog će se slučaja uvek doći tokom rekurzivnog postupka - zašto?). Ako je u tom slučaju dimenzija skupa n, pošto se u n elemenata skupa neki element slika, a ukupno imamo n originala, na osnovu Dirihleovog principa možemo lako dokazati da svaki original može da se slika u najviše jednu sliku (kada bi se dva originala slikala u istu sliku, za neku sliku ne bi postojao original koji se u nju slika). Utvrdili smo, dakle, da važi teorema koja tvrdi da je svaka surjekcija na konačnom skupu ujedno injekcija, pa je i bijekcija.

Što se tiče implementacije, jasno je da je bitno da za svaki element tekućeg skupa odredimo koliko se originala slika u njega (reći ćemo da je to ulazni stepen slike). Ulazne stepene možemo čuvati u jednom nizu (na poziciji i čuvamo koliko se elemenata slika u element i). Niz lako možemo popuniti tako što ga inicijalizujemo na nulu, prođemo kroz sve originale i za njihove slike uvećamo broj originala koji se u te slike slikaju. Prolaskom kroz taj niz možemo identifikovati sve elemente u koje se ni jedan element ne slika (to su oni koji imaju ulazni stepen nula). Njih je potrebno da uklonimo iz skupa. Ulazni stepen slike svakog elementa koji uklanjamo moramo umanjiti za jedan. To ne možemo uraditi istovremeno za sve elemente ulaznog stepena nula, već ih možemo ubaciti u red i zatim obrađivati jedan po jedan. Ako nakon smanjenja ulazni stepen nekog elementa postane nula i njega je potrebno ukloniti iz skupa. Da bi on u nekom trenutku bio uklonjen i obrađen, postavićemo ga u red. Dakle, invarijanta našeg programa biće da se u redu nalaze svi elementi čiji je ulazni stepen nula. Kada se red isprazni znaćemo da nema više elemenata čiji je ulazni stepen nula i funkcija će biti bijekcija na skupu elemenata koji nisu uklonjeni iz polaznog skupa. Pošto nas zanima samo njihov broj, održavaćemo samo promenljivu koja čuva kardinalnost tekućeg skupa i prilikom uklanjanja elemenata iz skupa smanjivaćemo je za jedan.

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

```cpp
using namespace std;

int main() {
  int n;
  cin >> n;
  vector<int> f(n);
  for (int i = 0; i < n; i++)
    cin >> f[i];

  // računamo ulazne stepene svih elemenata
  vector<int> ulazniStepen(n, 0);
  for (int i = 0; i < n; i++)
    ulazniStepen[f[i]]++;

  // u redu čuvamo one elemente čiji je ulazni stepen 0
  queue<int> q;
  for (int i = 0; i < n; i++)
    if (ulazniStepen[i] == 0)
      q.push(i);
  // broj preostalih elemenata
  int brojElemenata = n;
  while (!q.empty()) {
    // izbacujemo element koji je na redu (on ima ulazni stepen 0)
    int i = q.front(); q.pop();
    brojElemenata--;
    // ako nakon izbacivanja elementa i slika f[i] ima
    // ulazni stepen 0 ubacujemo je u red
    if (--ulazniStepen[f[i]] == 0)
      q.push(f[i]);
  }
  cout << brojElemenata << endl;
  return 0;
}
```

Način upotrebe reda u ovom rešenju je često koristan i pri rešavanju drugih problema. Kada god je potrebno obrađivati nekoliko elemenata koji ne mogu da budu obrađeni istovremeno, oni se mogu smestiti u red i zatim obrađivati jedan po jedan. Ako se tokom obrade pojavljuju novi elementi koje je potrebno obraditi, oni se dodaju na kraj reda. Obrada traje sve dok se red ne isprazni. Invarijanta je obično da red sadrži sve one elemente za koje je utvrđeno da moraju da budu obrađeni.

Red sa dva kraja

Red sa dva kraja je struktura podataka koja kombinuje funkcionalnost steka i reda, jer se elementi mogu i dodavati i skidati sa oba kraja.

U jeziku C++ red sa dva kraja se realizuje klasom `deque<T>`, gde je `T` tip elementa u redu. Red 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 početku reda (pod pretpostavkom da red nije prazan),
- `back` - očitava element na kraju reda (pod pretpostavkom da red nije prazan),
- `empty` - provera da li je red prazan,
- `size` - broj elemenata u redu.

Sve ove operacije su konstantne složenosti $O(1)$.

Još jedna klasa koja pruža isti interfejs je `List<T>`, koja je implementirana pomoću dvostruko povezane liste. Osnovna razlika je to što `deque<T>` ima pristup elementu na osnovu indeksa u konstantnom vremenu. Više reči o implementaciji svih ovih struktura biće dato kasnije.

Prikažimo korišćenje reda sa dva kraja kroz nekoliko primera.

Ograničena istorija pregledača veba

Problem: U istoriji se čuva najviše n prethodno posećenih veb-sajtova. Naredbom `back` vraćamo se na prethodno posećeni sajt. Napiši program koji simulira rad pregledača. Učitava se jedna po jedna adresa ili naredba `back`. Kada se učita `back` vraćamo se na prethodni sajt i ispisujemo njegovu adresu. Kada nema prethodnog sajta, ispisuje se `-`.

```
#include <iostream>
#include <string>
#include <deque>

using namespace std;

int main() {
  int n;
  cin >> n;
  string linija;
  deque<string> istorija;
  while (getline(cin, linija)) {
    if (linija == "back") {
      if (!istorija.empty()) {
        cout << istorija.back() << endl;
        istorija.pop_back();
      } else {
        cout << "-" << endl;
```

```
      }
    } else {
      if (istorija.size() == n)
        istorija.pop_front();
      istorija.push_back(linija);
    }
  }
  return 0;
}
```

Maksimumi segmenata dužine k

Problem: Napisati program koji za dati niz određuje maksimume svih njegovih segmenata dužine k. Na primer, ako je $k = 4$ i ako je dan niz $3, 8, 6, 5, 6, 3$, segmenti dužine k su $3, 8, 6, 5$, zatim $8, 6, 5, 6$ i $6, 5, 6, 3$ i njihovi maksimumi su redom 8, 8 i 6.

Naivno rešenje u kom bi se svi elementi smestili u niz i u kom bi se iznova tražio maksimum za svaki segment dužine k bilo bi složenosti $O(k \cdot (n - k))$ što je za $k \approx n/2$ jednako $O(n^2)$.

Zadatak možemo rešiti i u složenosti $O(n)$, ako upotrebimo sličnu tehniku čuvanja karakterističnih maksimuma (ili minimuma), kao u slučaju kada smo određivali najbližeg manjeg prethodnika.

Slično kada smo određivali maksimalni zbir segmenta dužine k, elemente tekućeg segmenta možemo čuvati u redu dužine k. Prilikom prelaska na svaki naredni segment, red ažuriramo tako što uklanjamo element sa početka reda, a novi element dodamo na kraj. Maksimum ažuriranog reda želimo da računamo inkrementalno, na osnovu maksimuma reda pre ažuriranja. Međutim, lako možemo utvrditi da to neće teći tako jednostavno. Naime, ako je maksimum segmenta pre ažuriranja njegov prvi element, nakon njegovog uklanjanja gubimo potpuno informaciju o maksimumu preostalih elemenata. Zato moramo ojačati invarijantu i čuvati više informacija. Kada početni element koji je ujedno maksimum segmenta ispadne iz segmenta, moramo znati maksimum preostalih elemenata. Dakle, moramo čuvati maksimum celog segmenta, zatim maksimum dela segmenta iza tog maksimuma, zatim maksimum dela segmenta iza tog maksimuma itd. Na primer, ako je segment $3, 8, 6, 5, 6, 3$ potrebno je da pamtimo vrednosti $8, 6, 6, 3$. Nazovimo to nizom karakterističnih maksimuma. Uklanjanje početnog elementa segmenta ne menja taj niz, osim u slučaju kada je on jednak maksimumu u tom slučaju se taj element uklanja sa početka niza. Razmotrimo sada kako se menja niz karakterističnih maksimuma kada se segment proširuje novim završnim elementom. Svi elementi niza karakterističnih maksimuma na desnom kraju koji su strogo manji od novog elementa segmenta se uklanjaju, jer oni više nisu maksimumi dela segmenta iza sebe. Kada se takvi elementi uklone (moguće je i da ih nema), novi element se dodaje na kraj niza karakterističnih maksimuma, jer je on maksimum jednočlanog segmenta koji sam čini.

Struktura podataka kojom modelujemo niz karakterističnih maksimuma treba da omogući ispitivanje vrednosti elementa koji se nalazi na početku, uklanjanje tog elementa, ispitivanje vrednosti elemenata koji se nalaze na kraju, uklanjanje tih elemenata i dodavanje novog elementa na kraj. Sve te operacije su omogućene u konstantnom vremenu kada se koristi red sa dva kraja.

```
#include <iostream>
#include <vector>
#include <deque>
#include <queue>

using namespace std;

int main() {
  int k;
  cin >> k;
  // ucitavamo ulazne podatke
  int n;
  cin >> n;

  // red u kome cuvamo trenutni segment
  queue<int> segment;

  // red karakteristicnih maksimuma tekuceg segmenta
  deque<int> maksimumi;
  for (int i = 0; i < n; i++) {
    if (i >= k) {
      // Posle prvih k koraka pocinjemo da skracujemo
      // tekuci segment
      int prvi = segment.front();
      segment.pop();
      // Ako je element koji se izbacuje jednak prvom elementu
      // u redu maksimuma, on se izbacuje iz reda.
      if (maksimumi.front() == prvi)
         maksimumi.pop_front();
    }

    // dodajemo novi element
    int ai;
    cin >> ai;
    segment.push(ai);

    // Azurira se red maksimuma
    while(!maksimumi.empty() && ai > maksimumi.back())
      maksimumi.pop_back();
    maksimumi.push_back(ai);

    // nakon sto je prvih k elemenata ubaceno u segment,
    // pocinjemo da u svakom koraku ispisujemo maksimume
    // segmenata
```

```
    if (i >= k - 1)
      cout << maksimumi.front() << endl;
  }

  return 0;
}
```

Prikažimo rad algoritma na primeru niza `3 8 6 3 1 5 9 4 2 7 6 5` i vrednosti $k = 4$.

i	segment	karakteristični maksimumi	izlaz
0	3	3	
1	3 8	8	
2	3 8 6	8 6	
3	3 8 6 3	8 6 3	8
4	8 6 3 1	8 6 3 1	8
5	6 3 1 5	6 5	6
6	3 1 5 9	9	9
7	1 5 9 4	9 4	9
8	5 9 4 2	9 4 2	9
9	9 4 2 7	9 7	9
10	4 2 7 6	7 6	7
11	2 7 6 5	7 6 5	7

Iako se unutar petlje `for` nalazi petlja `while` složenost algoritma jeste linearna tj. $O(n)$. Naime, unutrašnja petlja `while` se ne može izvršiti veliki broj puta. Ako se u nekom koraku njeno telo izvrši veliki broj puta, tada će se puno elemenata izbaciti iz reda karakterističnih maksimuma i već u naredom će taj red biti slabo popunjen, što znači da će se tada telo moći izvršiti znatno manji broj puta. Ukupan broj izvršavanja tela petlje `while` je zapravo ograničen sa n jer se svaki element niza samo jednom može dodati i samo jednom može ukloniti iz reda. Stoga je ukupno vreme izvršavanja algoritma linearno tj. iznosi $O(n)$.

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.

U jeziku C++ red sa prioritetom se realizuje klasom `priority_queue<T>`, gde je `T` tip elemenata u redu. 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

Operacije `push` i `pop` su obično složenosti $O(\log k)$, gde je k broj elemenata u redu, dok su ostale operacije složenosti $O(1)$.

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 način da se u C++-u definiše red sa prioritetom u kome su
   // elementi poređani u opadajućem redosledu prioriteta (ovde, vrednosti)
   priority_queue<int, vector<int>, greater<int>> Q;

   // učitavamo 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();
   }
   return 0;
}
```

Memorijska složenost ove implementacije je $O(n)$, jer se u redu čuva svih n elemenata, a vremenska složenost je $O(n \log n)$ jer se n puta izvode operacije složenosti $O(\log n)$ - podsetimo se, to što se broj elemenata smanjuje u svakoj narednoj operaciji izbacivanja elemenata ne utiče na asimptotsku složenost. Jednačina koja opisuje fazu umetanja, a i fazu izbacivanja je $T(n) = T(n-1) + O(\log n)$, a njeno rešenje je $O(n \log n)$.

U narednom poglavlju ćemo videti i kako se ovaj algoritam može direktno implementirati uz pomoć niza, bez korišćenja naprednih bibliotečkih struktura podataka.

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.

Jedan način bi bio da se učita niz i da se sortira, ali bi se tim pristupom nepotrebno trošilo i vreme i prostor (zamislite da je potrebno odrediti hiljadu najvećih od milion učitanih elemenata - potpuno nepotrebno bi se smeštalo 999000 elemenata i potpuno nepotrebno bi se tokom sortiranja određivao njihov međusobni redosled). 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. Memorijska složenost ove implementacije je $O(k)$ jer se u redu čuva najviše k elemenata, a vremenska je $O(n \log k)$ jer se $O(n)$ puta vrše operacije složenosti $O(\log k)$ (jednačina je $T(n) = T(n-1) + O(\log k)$).

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

int main() {
   int k;
   cin >> k;
   priority_queue<int> Q;
   int n;
   cin >> n;
   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() << endl;
        Q.pop();
    }

    return 0;
}
```

Medijane

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

Naivno rešenje bi podrazumevalo da se svi učitani elementi čuvaju u nizu i da se medijana svaki put računa iznova. Najefikasniji način da se medijana izračuna zahteva $O(k)$ operacija gde je k dužina niza, pa bi se ovim dobio algoritam složenosti $O(n^2)$. Slično bi bilo i da se elementi održavaju u stalno sortiranom nizu. Tada bi medijana mogla biti izračunata u vremenu $O(1)$, ali bi umetanje elementa na njegovo mesto zahtevalo $O(k)$ operacija, pa bi složenost opet bila $O(n^2)$.

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;
}
```

Ukupno se vrši n dodavanja elemenata, a u svakom koraku se vrše operacije složenosti $O(\log k)$, gde je k trenutni broj elemenata umetnutnih u redove (u svakom redu će biti zapravo oko $k/2$ elemenata, pa će broj operacija biti $O(\log k/2)$, no to je isto što i $O(\log k)$). Otud je ukupna složenost $O(n \log n)$.

Silueta zgrada - skyline

Problem: Sa broda se vide zgrade na obali velegrada. Duž obale je postavljena koordinatna osa i za svaku zgradu se zna pozicija levog kraja, visina i pozicija desnog kraja. Napisati program koji izračunava siluetu grada.

Svaku zgradu predstavljaćemo strukturom koja sadrži levi kraj zgrade a, zatim desni kraj zgrade b i njenu visinu h.

```
struct zgrada {
  int a, b, h;
  zgrada(int a = 0, int b = 0, int h = 0)
    : a(a), b(b), h(h) {
  }
};
```

Silueta je deo-po-deo konstantna funkcija i određena je intervalima konstantnosti $(-\infty, x_0)$, $[x_0, x_1)$, $[x_1, x_2)$, $\ldots$, $[x_{n-1}, +\infty)$, određenim tačkama podele $x_0 < x_1 < \ldots < x_{n-1}$ i vrednostima 0, h_0, $\ldots$, h_{n-2} i 0 funkcije na svakom od intervala.

$$\begin{array}{ccccccccccccc} & 0 & & h_0 & & h_1 & & \ldots & & h_{n-2} & & 0 & \\ -\infty & & x_0 & & x_1 & & x_2 & \ldots & x_{n-2} & & x_{n-1} & & +\infty \end{array}$$

Podrazumevamo da su krajnje tačke $-\infty$ i $+\infty$ i da su vrednosti na tim intervalima jednake nuli. Dakle, deo-po-deo konstantna funkcija se može predstaviti pomoću n tačaka $x_0, \ldots x_{n-1}$ i $n-1$ vrednosti $h_0, \ldots, h_{n-2}$. Jednostavnosti radi mi ćemo ovakve funkcije predstavljati pomoću n uređenih parova (x_0, h_0), (x_1, h_1), $\ldots$, (x_{n-2}, h_{n-2}) i $(x_{n-1}, 0)$. Dakle, naš algoritam prima niz uređenih trojki koji opisuje pojedinačne zgrade, a vraća niz uređenih parova koji opisuje siluetu.

Svaki uređeni par (x_i, h_i) predstavljaćemo strukturom promena, a siluetu ćemo predstavljati vektorom promena.

```
struct promena {
  int x, h;
  promena(int x = 0, int h = 0)
    : x(x), h(h) {
```

```
  }
};

vector<promena> silueta;
```

Postoji nekoliko načina da se ovaj zadatak reši. Osnovna induktivna konstrukcija podrazumeva da umemo da nađemo siluetu sve osim jedne zgrade i da na kraju umemo da tu zgradu uklopimo u siluetu ostalih zgrada. Da bismo zgradu uklopili u postojeću siluetu potrebno nam je linearno vreme (iako deo siluete u koji se zgrada integriše možemo možda uraditi binarnom pretragom i iako bismo umetanje novih promena u siluetu mogli vršiti u konstantnom vremenu ako bismo umesto vektora upotrebili listu, potrebno je da prođemo i proanaliziramo sve promene u postojećoj silueti u delu gde se umeće nova zgrada, a njih može da bude puno ako je zgrada široka). To znači da će rešenje biti složenosti $T(n) = T(n-1) + O(n)$, $T(1) = O(1)$, što daje složenost $O(n^2)$.

Jedno efikasnije rešenje zasnovano na dekompoziciji ćemo prikazati kada se budemo detaljnije bavili tom tehnikom konstrukcije algoritama.

Drugo efikasnije rešenje možemo postići ako zgrade obrađujemo sleva nadesno. Silueta se menja samo u tačkama u kojma počinje ili se završava neka zgrada. Stoga možemo napraviti niz karakterističnih tačaka koji sadrži sve početke i krajeve zgrada i informaciju o tome da li je tekuća tačka početak ili kraj. Kod svake karakteristične tačke potrebno je da odredimo visinu siluete nakon te tačke. Tu visinu ćemo odrediti kao najveću visinu svih zgrada koje počinju levo od te karakteristične tačke (uključujući eventualno i tu tačku) a čiji se desni kraj ne nalazi levo od te tačke (uključujući eventualno i te tačke). Tu na scenu stupaju strukture podataka.

Potrebno je da održavamo strukturu podataka u koju ćemo ubacivati zgradu po zgradu kako nailaze (kada se naiđe na njihov levi kraj), izbacivati zgrade kako prolaze (kada se naiđe na njihov desni kraj) i u svakom trenutku moći efikasno da pronađemo maksimum trenutno ubačenih zgrada. Problem sa ovim zahtevima je to što je teško efikasno ih ostvariti istovremeno. Naime, ako bismo čuvali podatke o zgradama nekako uređene na osnovu njihovih visina, efikasno bismo pronalazili najvišu, ali bi izbacivanje zgrada na osnovu pozicija krajeva bilo komplikovano (jer zgrade ne bi bile uređene po tom kriterijumu). Sa druge strane, ako bismo zgrade čuvali nekako uređene po koordinatama krajeva, pronalaženje one sa maksimalnom visinom bi bilo neefikasno. Prvi pristup je ipak bolji, jer ne možemo nikako da se odreknemo mogućnosti efikasnog nalaženja maksimuma. Struktura koja nam to omogućava je red sa prioritetom koji je uređen na osnovu visina zgrada. Problem sa takvim redom je to što je izbacivanje zgrade kada se naiđe na njen desni kraj problematično (red sa prioritetom nam daje veoma efikasno izbacivanje zgrade koja je najviša, ali ne i ostalih zgrada). Ključni trik, koji se veoma često može upotrebiti kod korišćenja redova sa prioritetom je da izbacivanje elemenata iz reda odložimo i da u strukturi podataka dopustimo čuvanje podataka koji su po nekom kriterijumu zastareli i koji ne bi više trebalo da se upotrebljavaju. Naime, pretpostavimo da se u redu sa prioritetom nalaze

sve zgrade na čiji smo početak do sada naišli. Kada želimo da pronađemo najvišu zgradu od njih koja se još nije završila, možemo da razmotrimo zgradu na vrhu reda. Moguće je da se ona još nije završila i u tom slučaju ona predstavlja rešenje. U suprotnom, ako se ta zgrada završila, njoj nije više mesto u redu i možemo da je izbacimo iz reda. Međutim, za razliku od trenutka kada smo naišli na njen desni kraj, kada je njeno izbacivanje bilo komplikovano, u ovom trenutku se ona nalazi na vrhu reda i izbacivanje se može izvršiti veoma efikasno. Dakle, analiziraćemo i izbacivaćemo jednu po jednu najvišu zgradu sa vrha reda sve dok ne dođemo do zgrade koja se još nije završila.

```
vector<promena> napraviSiluetu(vector<zgrada>& zgrade) {
  vector<promena> silueta;

  // sortiramo sve zgrade na osnovu pocetka
  struct PorediPocetak {
    bool operator() (const zgrada& z1, const zgrada& z2) {
      return z1.a < z2.a;
    }
  };
  sort(begin(zgrade), end(zgrade), PorediPocetak());

  // pravimo vektor svih pocetnih i krajnjih tacaka zgrada i za svaku
  // tacku belezimo da li je pocetna
  vector<pair<int, bool>> tacke(2 * zgrade.size());
  int i = 0;
  for (auto z : zgrade) {
    tacke[i++] = make_pair(z.a, true);
    tacke[i++] = make_pair(z.b, false);
  }
  // sortiramo sve znacajne tacke na osnovu koordinate
  sort(begin(tacke), end(tacke), [](auto p1, auto p2){
      return p1.first < p2.first;
    });


  // cuvamo visine do sada obradjenih zgrada tako da veoma brzo mozemo
  // da odredimo zgradu najvece visine - koristimo red sa prioritetom
  struct PorediVisinu {
    bool operator() (const zgrada& z1, const zgrada& z2) {
      return z1.h < z2.h;
    }
  };
  priority_queue<zgrada, vector<zgrada>, PorediVisinu> pq;

  // obilazimo sve znacajne tacke sleva nadesno
  int z = 0;
  for (auto p : tacke) {
    int x = p.first;
    int je_pocetak = p.second;
```

```
    // dodajemo u red sve zgrade koje pocinju na trenutnoj koodrinati
    if (je_pocetak)
      while (z < zgrade.size() && zgrade[z].a == x)
        pq.push(zgrade[z++]);

    // trazimo najvisu visinu do sada zapocete zgrade
    // eliminisemo zgrade koje su do sada zavrsene
    while (!pq.empty() && pq.top().b <= x)
      pq.pop();
    int h = !pq.empty() ? pq.top().h : 0;

    // integrisemo visinu najvise zgrade na trenutnoj koordinati u
    // tekucu siluetu
    dodajPromenu(silueta, x, h);
  }

  // vracamo rezultat
  return silueta;
}
```

Ostaje još pitanje kako novu promenu integrisati u postojeću siluetu. To nije teško, ali je potrebno obratiti pažnju na nekoliko specijalnih slučajeva (koji uglavnom nastupaju usled toga što više zgrada mogu imati isti početak ili kraj). Invarijanta koju želimo da nametnemo na siluetu je da su x koordinate svih uzastopnih promena različite i da ne postoje dve uzastopne promene sa istom visinom (druga promena je tada višak). Ako poslednja promena u silueti ima istu x koordinatu kao i promena koja se ubacuje, onda se umesto dodavanja nove promene ažurira visina te poslednje promene, ako je to potrebno (ako je nova visina veća o postojeće). Time se može desiti da nakon ažuriranja poslednja i pretposlednja promena imaju istu visinu, pa je u tom slučaju potrebno ukloniti poslednju promenu. Na kraju, ako nova promena ima istu visinu kao i poslednja promena u silueti, nema potrebe da se dodaje. Ovim se invarijanta održava.

```
void dodajPromenu(vector<promena>& silueta, int x, int h) {
  int n = silueta.size();
  if (n > 0) {
    int xb = silueta[n-1].x;
    int hb = silueta[n-1].h;

    if (xb == x) {
      if (h > hb) {
        silueta[n-1].h = h;
        if (n > 1 && silueta[n - 2].h == h)
          silueta.pop_back();
      }
    } else if (hb != h)
      silueta.push_back(promena(x, h));
  } else
```

```
    silueta.push_back(promena(x, h));
}
```

Skupovi

Skup je osnovni matematički pojam, a ponekad u programiranju imamo potrebu da održavamo skup elemenata. Savremeni programski jezici obično pružaju bibliotečku podršku za to.

U jeziku C++ skup je podržan kroz dve klase: `set<T>` i `unordered_set<T>`, gde je `T` tip elemenata skupa. Implementacija je različita (prva je zadata na balansiranim binarnim drvetima, a druga na heš tablicama, o čemu će više biti reči u narednom poglavlju), pa su im vremenske i prostorne karakteristike donekle različite.

Skupovi podržavaju sledeće osnovne operacije (za pregled svih operacija upućujemo čitaoca na dokumentaciju):

- `insert` - umeće novi element u skup (ako već postoji, operacija nema efekta). Kada se koristi `set` složenost umetanja je $O(\log k)$, gde je k broj elemenata u skupu, a kada se koristi `unordered_set`, složenost najgoreg slučaja je $O(k)$, dok je prosečna složenost $O(1)$, pri čemu je amortizovana složenost uzastopnog dodavanja većeg broja elemenata takođe $O(1)$. Naglasimo i da konstante kod složenosti $O(1)$ mogu biti relativno velike.
- `find` - proverava da li skup sadrži dati element i vraća iterator na njega ili `end` ako je odgovor negativan. Tako se provera pripadnosti elementa `e` skupu `s` može izvršiti sa `if (s.find(e) != s.end()) ...` Složenost najgoreg slučaja ove operacije ako se koristi `set` je $O(\log k)$, a ako se koristi `unordered_set` složenost najgoreg slučaja je $O(k)$, ali je prosečna složenost $O(1)$.
- `erase` - uklanja dati element iz skupa. Složenost je ista kao u prethodnom slučaju.

Prikažimo upotrebu skupova na nekoliko primera.

Problem: Napiši program koji određuje da li među učitanih n brojeva ima duplikata.

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

int main() {
  int n;
  cin >> n;
  bool duplikati = false;
  set<int> vidjeni;
  for (int i = 0; i < n; i++) {
```

```
    int x;
    cin >> x;
    if (vidjeni.find(x) != vidjeni.end()) {
       duplikati = true;
       break;
    }
    duplikati.insert(x);
  }
  cout << (duplikati ? "da" : "ne") << endl;
  return 0;
}
```

Složenost najgoreg slučaja ovog pristupa (a to je slučaj kada nema duplikata) je $O(n \log n)$, jer se n puta izvršava pretraga i umetanje u skup, a ta operacija ima složenost $O(\log k)$, gde je k trenutni broj elemenata u skupu. Ovim dobijamo sumu logaritama od 1 do n, a za nju smo već dokazali da je $\Theta(n \log n)$.

Naravno, zadatak je moguće rešiti i na druge načine. Jedan smo već videli - učitamo sve elemente u niz, sortiramo ga i onda proveravamo uzastopne elemente. Asimptotska složenost dolazi od sortiranja i takođe je $O(n \log n)$, ali ta varijanta može biti memorijski i vremenski malo efikasnija (naravno, samo za konstantni faktor), jer treba imati na umu da naprednije strukture podataka kakav je `set` često nose određenu cenu u ondnosu na klasične nizove.

Problem: Učitavaju se šifre proizvoda koji se nalaze u dva magacina. Napisati program koji određuje one koji se nalaze u oba.

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

int main() {
  int n1;
  cin >> n1;
  set<int> proizvodi1;
  for (int i = 0; i < n1; i++) {
     int x;
     cin >> x;
     proizvodi1.insert(x);
  }

  int n2;
  cin >> n2;
  set<int> proizvodi2;
  for (int i = 0; i < n2; i++) {
     int x;
     cin >> x;
     proizvodi2.insert(x);
  }
```

```
  for (int x : priozvodi1)
     if (proizvodi2.find(x) != proizvodi2.end())
        cout << x << endl;

  return 0;
}
```

Jezik C++ daje i funkcije koje izračunavaju presek, uniju i razliku dva skupa (`set_intersection`, `set_union`, `set_difference`).

```
   set<int> presek;
   set_intersection(begin(proizvodi1), end(proizvodi1)
                    begin(proizvodi2), end(proizvodi2),
                    inserter(presek, begin(presek)));
   for (int x : presek)
      cout << x << endl;
```

Mape

Mape (rečnici, konačna preslikavanja, asocijativni nizovi) su strukture podataka koje skup ključeva iz nekog konačnog domena preslikavaju u neki skup vrednosti. Osnovne operacije su pridruživanje vrednosti datom ključu, očitavanje vrednosti pridružene nekom datom ključu, ispitivanje da li je nekom ključu pridružena vrednost i brisanje elementa sa datim ključem.

U jeziku C++ mape su podržane klasama `map<K, V>` i `unordered_map<K, V>`, gde je `K` tip ključeva, a `V` tip vrednosti. Slično kao u slučaju skupa, implementacija prve klase je zasnovana na balansiranim binarnim drvetima, a druga na heš tablicama. Na raspolaganju su nam sledeće operacije.

- Osnovni operator u radu sa mapama je operator indeksnog pristupa. Na primer, ako mapa `ocene` preslikava imena učenika u njihove prosečne ocene, tada se ocena učenika `pera` može i pročitati i postaviti pomoću `ocene["pera"]`. Kada se koristi `map`, garantovana složenost ovog operatora je $O(\log k)$ gde je k broj trenutno pridruženih ključeva u mapi. Kada se koristi `unordered_map` prosečna složenost je $O(1)$, ali konstantni faktor može biti veliki, dok je složenost najgoreg slučaja $O(k)$.
- Metoda `find` vraća iterator koji ukazuje na slog sa datim ključem u mapi. Ako taj ključ ne postoji u mapi, vraća se iterator na kraj mape. Tako se provera pripadnosti ključa `k` mapi `m` može izvršiti sa `if (m.find(k) != m.end()) ...`. Složenost je identična kao u slučaju umetanja.
- Metoda `erase` briše element iz mape. Argument može biti bilo vrednost ključa, bilo iterator koji pokazuje na element koji se uklanja. Složenost brisanja date vrednosti ključa je identična kao u prethodnim slučajevima.

Jezik dopušta i iteriranje kroz elemente mape. Na primer, kroz sve ocene se može proći sa sledećom petljom.

```
for (auto it : ocene)
  cout << it.first << " " << it.second << endl;
```

Primetimo da promenljiva `it` tokom iteracije uzima uređene parove (ključ, vrednost), pa se ključu pristupa preko polja `first`, a vrednosti preko polja `second`.

Problem: Organizuj strukturu podataka koja studentima pridružuje broj poena.

```
#include <iostream>
#include <map>
using namespace std;
int main() {
   map<string, int> poeni =
      {{"pera", 84}, {"ana", 92}, {"joca", 67}};
   cout << poeni["ana"] << endl;
   poeni["joca"] = 72;
   cout << poeni["joca"] << endl;
   poeni["ivana"] = 48;
   cout << poeni["ivana"] << endl;
   return 0;
}
```

Problem: Napisati program koji izračunava frekvenciju (broj pojavljivanja) svake od reči u tekstu.

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

int main() {
   map<string, int> frekvencije;
   string rec;
   while (cin >> rec)
      frekvencije[rec]++;
   for (auto it : frekvencije)
      cout << it.first << ": " << it.second << endl;
   return 0;
}
```

Naglasimo da se u slučaju kada je domen mali rečnici mogu realizovati i preko klasičnih nizova. Preuredimo prethodni program tako se broje slova unutra jedne reči koja se sastoji samo od malih slova. Tada rečnik možemo realizovati nizom od 26 elemenata i ključeve (mala slova) lako prevesti u ključeve (oduzimajući ASCII kôd karaktera `'a'`).

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

int main() {
  int frekvencije[26];
  string rec;
  cin >> rec;
  for (char c : rec)
    frekvencije[c - 'a']++;
  for (int i = 0; i < frekvencije.size(); i++)
    cout << (char)('a' + i) << " " << frekvencije[i] << endl;
  return 0;
}
```

Broj segmenata čiji su svi elementi različiti

Prikažimo kako se mape mogu upotrebiti i u rešenjima složenijih algoritamskih zadataka.

Problem: Dat je niz celih brojeva. Odrediti ukupan broj segmenata čiji su svi elementi različiti.

Jedno prilično elegantno rešenje se zasniva na tome da za svaku poziciju analiziramo sve segmente kojima je kraj upravo na toj poziciji, a kojima su svi elementi različiti. Ako su svi elementi nekog segmenta različiti, onda su i svim njegovim sufiksima svi elementi različiti. Ako neki segment sadrži duplikate, onda duplikate sadrže i svi segmenti kojima je on sufiks. Zato je dovoljno da pronađemo najduži mogući segment koji se završava na poziciji $kraj$ i kojem su svi elementi različiti i znaćemo da su svi segmenti sa različitim elementima koji se završavaju na poziciji $kraj$ tačno svi njegovi sufiksi. Ako je to segment određen pozicijama iz intervala $[pocetak, kraj]$ onda će takvi biti i segmenti određeni intervalima pozicija $[pocetak + 1, kraj]$, $\ldots$, $[kraj - 1, kraj]$. Njih je ukupno $kraj - pocetak$.

Ostaje pitanje kako za datu poziciju $kraj$ odrediti poziciju $pocetak$. Pokušajmo da primenimo induktivno-rekurzivnu konstrukciju. Za $kraj = 0$ važi da je $pocetak = 0$, jer su svi elementi jednočlanog segmenta koji čini samo element na poziciji 0 različiti i on je najduži takav segment koji se zavšava na poziciji 0. Pretpostavimo da je $kraj > 0$ i da već znamo rešenje za prethodnu poziciju kraja, tj. pretpostavimo da znamo da interval pozicija $[pocetak, kraj - 1]$ određuje najduži segment kome su svi elementi različiti i koji se završava na poziciji $kraj - 1$.

Ako se element na poziciji $kraj$ ne sadrži u tom segmentu, onda je segment $[pocetak, kraj]$ naš traženi. Ako se sadrži, onda je on sigurno jedini duplikat u segmentu $[pocetak, kraj]$. Ako pretpostavimo da se element na poziciji $kraj$ u tom segmentu javlja i na poziciji p, tada je segment koji tražimo $[p + 1, kraj]$, zato što svi segmenti koji počinju od pozicije $pocetak$, pa sve do pozicije p sadrže

isti taj duplikat. Segment $[p+1, kraj]$ ne sadrži duplikate i najduži je takav segment, tako da u toj situaciji *pocetak* treba postaviti na vrednost $p+1$.

Na kraju, ostaje pitanje kako utvrditi da li se a_{kraj} javlja u segmentu $[pocetak, kraj-1]$ i ako se javlja, kako odrediti poziciju p na kojoj se javlja. Direktno rešenje podrazumeva linearnu pretragu segmenta prilikom svakog proširenja niza, što bi znatno degradiralo složenost celog algoritma. Bolje rešenje je da se čuva asocijativni niz (mapa, rečnik) u kojem se elementi niza iz segmenta pozicija $[pocetak, kraj-1]$ preslikavaju u njihove pozicije. Tada se jednostavnom pretragom mape tj. rečnika (čija je složenost konstantna ili najviše logaritamska) utvrđuje da li se novi krajnji element javlja u prethodnom segmentu i na isti način se određuje i njegova pozicija. Recimo i da se prilikom pomeranja početka na poziciju $p+1$ segment skraćuje, što treba da se oslika i u mapi - zato je tada potrebno iz mape ukloniti sve elemente koji se javljaju u nizu, na pozicijama od *pocetak*, pa zaključno sa p.

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

int main() {
  // ucitavamo dati niz brojeva
  int n;
  cin >> n;
  vector<int> a(n);
  for (int i = 0; i < n; i++)
    cin >> a[i];

  // ukupan broj segmenata niza ciji su svi elementi razliciti
  int broj = 0;

  // za svaku poziciju kraj zelimo da pronadjemo najduzi segment
  // oblika [pocetak, kraj] koji ima sve razlicite elemente

  // za svaki element u tekucem segmentu [pocetak, kraj] pamtimo
  // poziciju na kojoj se pojavljuje
  unordered_map<int, int> prethodno_pojavljivanje;

  int pocetak = 0;
  for (int kraj = 0; kraj < n; kraj++) {
    if (prethodno_pojavljivanje.find(a[kraj]) != prethodno_pojavljivanje.end()) {
      // nijedan segment koji se zavrsava na poziciji kraj, a pocinje
      // pre ranijeg pojavljivanja elementa a[kraj] ne moze da ima sve
      // razlicite elemente, pa zato razmatramo samo segmente koji se
      // zavrsavaju na poziciji kraj i pocinju iza pozicije tog
      // prethodnog pojavljivanja - najduzi takav pocinje na prvoj
      // poziciji iza te pozicije
```

```
      int novi_pocetak = prethodno_pojavljivanje[a[kraj]] + 1;
      // brisemo iz segmenta sve elemente od starog do ispred novog pocetka
      // i mapu uskladjujemo sa time
      for (int i = pocetak; i < novi_pocetak; i++)
        prethodno_pojavljivanje.erase(a[i]);
      // pomeramo pocetak
      pocetak = novi_pocetak;
    }
    // prosirujemo segment elementom a[kraj], pa pamtimo poziciju
    // njegovog pojavljivanja
    prethodno_pojavljivanje[a[kraj]] = kraj;

    // [pocetak, kraj] sadrzi sve razlicite elemente i on je najduzi
    // takav koji se zavrsava na poziciji kraj
    // sigurno su takvi i [pocetak+1, kraj], ..., [kraj-1, kraj]
    // njih ima (kraj - pocetak) i taj broj dodajemo na ukupan broj
    // trazenih segmenata
    broj += kraj - pocetak;
  }

  // ispisujemo ukupan broj pronadjenih segmenata
  cout << broj << endl;

  return 0;
}
```

Broj segmenata datog zbira

Problem: Napiši program koji za dati niz celih brojeva određuje sve neprazne segmente uzastopnih elemenata niza čiji je zbir jednak datom broju.

Zbirovima segmenata niza celih brojeva smo se već bavili kada smo tražili onaj najvećeg zbira, ali je sada zadatak malo drugačiji.

Direktno rešenje grubom silom podrazumevalo bi da se provere svi segmenti uzastopnih elemenata, da se za svaki izračuna zbir i da se proveri da li je taj zbir jednak traženom. Svi segmenti se mogu nabrojati ugnežđenim petljama. Složenost ovog pristupa je kubna tj. $O(n^3)$ nu odnosu na dimenziju n (postoji kvadratni broj segmenata i za sabiranje elemenata svakog od njih potrebno je linearno vreme).

Prethodni algoritam se može unaprediti ako se zbirovi računaju inkrementalno čime se složenost celog algoritma redukuje na kvadratnu tj. $O(n^2)$.

Elegantan i često primenjivan način da se dobiji zbirovi svih segmenata uzastopnih brojeva je da se izračunaju zbirovi prefiksa i da se zbir elemenata segmenta $[i, j]$ izrazi kao razlika zbira elemenata segmenta $[0, j]$ i zbira elemenata segmenta $[0, i-1]$. Dakle, u pomoćni niz b na svaku poziciju k možemo smestiti zbir prvih k elemenata niza (ovo opet možemo uraditi inkrementaln). Prvi element niza b

je nula i on ima jedan element više od niza a. Tako se zbir elemenata niza a iz segmenta $[i, j]$ uvek određuje kao $b_{j+1} - b_i$. Zbirovi prefiksa (parcijalne sume niza) se mogu u jeziku C++ odrediti i bibliotečkom funkcijom `partial_sum` koja prima dva iteratora koji ograničavaju deo niza (ili vektora) čije se parcijalne sume izračunavaju i iterator koji ukazuje na početak dela niza (ili vektora) u koji se parcijalne sume upisuju. Primetimo da bismo zbirove prefiksa mogli smestiti i u sam niz a, ako je memorija kritičan resurs. Ako proveravamo svaki par elemenata $i < j$, ponovo dobijamo algoritam kvadratne složenosti.

Prethodna ideja nam daje mogućnost da stignemo do efikasnijeg rešenja. Problem možemo formulisati i ovako. Za svaki zbir b_{j+1} prefiksa $[0, j]$ potrebno je da pronađemo da li postoji zbir b_i prefiksa $[0, i)$ za $i < j$ takva da je $b_{j+1} - b_i = z$, gde je z traženi zbir, tj. da se proveri da li se među zbirovima prethodnih prefiksa nalazi vrednost $b_i = b_{j+1} - z$. Ako se ta pretraga vrši linearno, dolazimo do implementacije veoma slične prethodnoj, koja ispituje svaki par elemenata $i < j$. Pošto među elementima niza može biti i negativnih, zbirovi prefiksa nisu sortirani i ne možemo primeniti ni binarnu pretragu. Ostaje nam, međutim, mogućnost da u nekoj strukturi podataka koja omogućava efikasno pretraživanje čuvamo sve zbirove prefiksa za indekse $i < j$. Ako algoritam organizujemo tako da j uvećavamo od 0 do $n - 1$, tada se na kraju svakog koraka u tu strukturu može dodati i zbir tekućeg segmenta (b_{j+1}). Struktura treba da realizuje pretragu po ključu, tako da je najbolje upotrebiti asocijativno preslikavanje, odnosno mapu tj. rečnik. Pošto se u zadatku traži samo određivanje broja segmenata sa datim zbirom, ključevi mogu biti zbirovi prefiksa, a vrednost pridružena svakom ključu može biti broj pojavljivanja prefiksa sa tim zbirom. Da se tražila samo provera da li postoji segment sa datim zbirom, mogli smo umesto preslikavanja čuvati samo skup ranije viđenih vrednosti zbirova prefiksa, a da su se eksplicitno tražili svi prefiksi, onda bismo svaki ključ preslikavali u niz vrednosti i takvih da je b_i jednako tom ključu (mogla bi se upotrebiti i multimapa o čemu će više reči biti kasnije). Ako računamo da će pretraga biti realizovana u $O(\log n)$ (što je najčešće slučaj ako se koriste strukture podataka zasnovane na binarnim stablima, kao što je u slučaju `map`), tada će ukupna složenost ove implementacije biti $O(n \log n)$, što je znatno efikasnije nego prethodne implementacije. Napomenimo da smo dobitak na efikasnosti platili dodatnom memorijom koju smo angažovali, međutim, ako je memorija kritičan resurs u ovom scenariju nije neophodno pamtiti polazni niz, tako da memorijska složenost neće biti značajno povećana.

```
int brojSegmenataDatogZbira(const vector<int>& a, int trazeniZbir) {
  // ucitavamo trazeni zbir
  int trazeniZbir;
  cin >> trazeniZbir;

  // zbir prefiksa
  int zbirPrefiksa = 0;

  // broj segmenata sa trazenim zbirom
  int broj = 0;
```

```
  // broj pojavljivanja svakog vidjenog zbira prefiksa
  map<int, int> zbiroviPrefiksa;
  // zbir pocetnog praznog prefiksa je 0 i on se za sada pojavio
  // jednom
  zbiroviPrefiksa[0] = 1;

  // ucitavamo elemente niza niz
  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    int x;
    cin >> x;
    // prosirujemo prefiks tekucim elementom
    zbirPrefiksa += x;

    // trazimo broj pojavljivanja vrednosti zbirPrefiksa - trazeniZbir
    // i azuriramo broj pronadjenih segmenata
    auto it = zbiroviPrefiksa.find(zbirPrefiksa - trazeniZbir);
    if (it != zbiroviPrefiksa.end())
      broj += it->second;

    // povecavamo broj pojavljivanja trenutnog zbira
    zbiroviPrefiksa[zbirPrefiksa]++;
  }

  return broj;
}
```

Još jedan način da se efikasno pretražuju zbirovi prefiksa je da se umesto mape svi oni smeste u niz, da se onda taj niz sortira i da se primeni binarna pretraga. Međutim, ovde treba biti obazriv, jer se nakon sortiranja na osnovu zbira izgubi originalni poredak prefiksa. Zbir segmenta se može dobiti samo ako se od zbira dužeg prefiksa oduzme zbir nekog kraćeg (nikako obratno). Stoga je potrebno da uz svaki zbir prefiksa pamtimo i dužinu tog prefiksa. Nakon sortiranja, obrađujemo jedan po jedan prefiks polaznog niza i ako je zbir tekućeg prefiksa z_1 binarnom pretragom pronalazimo sve one prefikse čiji je zbir z_2 takve da je $z_1 - z_2 = z$, gde je z traženi zbir, a koji su kraći od segmenta čiji je zbir z_1 (više reči o primenama binarne pretrage biće dato kasnije).

Multiskupovi i multimape

U skupu se svaki element pojavljuje najviše jednom. Multiskup je struktura podataka u kojoj se elementi mogu pojavljivati i više puta. Multiskup, dakle, odgovara mapi koja slika ključeve u njihov broj pojavljivanja (i tako se interno može implementirati).

U jeziku C++ multiskupovi se reprezentuju objektima klase `multiset<T>`, gde je `T` tip elemenata multiskupa. Na raspolaganju su nam sledeće operacije.

- `insert` umeće dati element u multiskup.
- `erase` briše element iz multiskupa. Ako je argument iterator, briše se element na koji taj iterator pokazuje, a ako je argument vrednost, brišu se sva pojavljivanja te vrednosti.
- `count` izračunava broj pojavljivanja datog elementa u multiskupu.

Ilustrujmo upotrebu multiskupova na par primera.

Problem: Sortirati sve reči koje se čitaju sa ulaza. Ako se neka reč pojavila više puta, prikazati je više puta.

Jedan od načina je da se sve reči umetnu u multiskup i da se zatim ispišu redom svi elementi multiskupa. Iako je ovaj pristup malo sporiji od korišćenja običnog niza niski i bibliotečkog sortiranja, asimptotska složenost mu je i dalje $O(n \log n)$.

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

int main() {
   multiset<string> reci;

   string rec;
   while (cin >> rec)
     reci.insert(rec);

   for (auto rec : reci)
     cout << rec << endl;

   return 0;
}
```

Problem: Sa ulaza se unosi broj n i zatim n reči. Nakon toga se unosi broj k i k reči. Napisati program za svaku od tih k učitanih reči određuje koliko se puta pojavila među prvih n reči.

Pored elementarnog rešenja koje bi smestilo sve reči u niz i sortiralo taj niz, jedan način koji se često koristi je da se upotrebi mapa kojom se svaka reč preslikava u njen broj pojavljivanja. U narednom rešenju umesto mape, koristimo multiskup.

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

int main() {
```

```
    // učitavamo reči i smeštamo ih u multiskup
    multiset<string> reci;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        string s;
        cin >> s;
        reci.insert(s);
    }

    // za k reči očitavamo i prijavljujemo broj pojavljivanja
    int k;
    cin >> k;
    for (int i = 0; i < k; i++) {
        string s;
        cin >> s;
        cout << s << ": " << reci.count(s) << endl;
    }

    return 0;
}
```

Multimape dopuštaju da se isti ključ preslikava u više od jedne vrednosti. Jedan način da se realizuju je preslikavanje ključeva u kolekciju (niz, vektor) vrednosti.

U jeziku C++ se multimape direktno predstavljaju objektima klase `multimap<K, V>`, gde je `K` tip ključeva, a `V` tip vrednosti elemenata. Na raspolaganju imamo sledeće operacije.

- `insert` pridružuje novu vrednost ključu. Ključ i vrednost se zadaju kao argument i to kao par (vrednost tipa `pair<K, V>`). Ova operacija ima garantovanu složenost najgoreg slučaja $O(\log k)$, gde je k broj trenutnih elemenata u multimapi.
- `find` vraća iterator koji ukazuje na jednu vrednost pridruženu datom ključu ili `end`, ako ključu nije pridružena ni jedna vrednost. I ova operacija ima garantovanu složenost $O(\log k)$.
- `equal_range` vraća par iteratora koji ograničavaju deo multimape koji sadrži sve vrednosti pridružene datom ključu. I ova operacija ima garantovanu složenost $O(\log k)$.

Problem: Napisati program koji učitava imena gradova i zemalja u kojima se nalaze. Napiši program koji ispisuje te podatke sortirane po zemljama, a zatim posebno ispisuje sve gradove iz Srbije.

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

```
int main() {
  multimap<string, string> gradovi;

  int n;
  cin >> n;
  for (int i = 0; i < n; i++) {
    string grad, zemlja;
    cin >> grad >> zemlja;
    gradovi.insert(make_pair(zemlja, grad));
  }

  for (auto it : gradovi)
    cout << it.first << " " << it.second << endl;

  auto gradovi_srbije = gradovi.equal_range("Srbija");
  for (auto it = gradovi_srbije.first; it != gradovi_srbije.second; it++)
    cout << it->second << endl;

  return 0;
}
```

Osmišljavanje apstraktnih struktura podataka

Problem: Opisati realizaciju apstraktne strukture podataka koja podržava sledeće operacije:

- `umetni(x)`: umetni ključ `x` u strukturu podataka, ako ga tamo već nema;
- `obriši(x)`: obriši ključ `x` iz strukture podataka (ako ga ima);
- `naredni(x)`: pronađi najmanji ključ u strukturi podataka koji je strogo veći od `x`. Izvršavanje svake od ovih operacija treba da ima vremensku složenost $O(\log k)$ u najgorem slučaju, gde je k broj elemenata u strukturi podataka.

Pogodna struktura podataka za realizaciju ovog tipa je skup i implementacija skupa u jeziku C++ podržava sve tri operacije. Umetanje se vrši metodom `insert`, brisanje metodom `erase`, dok se nalaženje prvog strogo većeg elementa može realizovati metodom `upper_bound`. Metodom `lower_bound` može se pronaći prvi element koji je veći ili jednak datom. Obe metode vraćaju iterator koji ukazuje iza kraja skupa, ako traženi element ne postoji.

Implementacija može biti sledeća.

```
set<int> s;

void umetni(int x) {
  s.insert(x);
}

void obrisi(int x) {
```

```
  s.erase(x);
}

bool naredni(int x, int& y) {
  auto it = s.upper_bound(x);
  if (it == s.end()) return false;
  y = *it;
  return true;
}
```

Problem: Konstruisati apstraktnu strukturu podataka za čuvanje preslikavanja ključeva u vrednosti. Struktura podataka treba da podržava sledeće operacije:

- `vrednost(x)`: odrediti vrednost pridruženu elementu sa ključem `x` (nula, ako ključ `x` nema pridruženu vrednost);
- `umetni(x, y)`: ključu `x` pridruži vrednost `y`;
- `obriši(x)`: obriši element sa ključem `x`;
- `uvećaj(x, y)`: vrednost pridruženu ključu `x` uvećaj za `y`;
- `uvećajSve(y)`: vrednosti pridružene svim ključevima uvećaj za `y`.

Vreme izvršavanja svake od ovih operacija u najgorem slučaju treba da bude $O(\log k)$, gde je k broj ključeva kojima su pridružene vrednosti.

Sve operacije osim poslednje direktno su podržane mapama. Da bi se realizovala operacija `uvećajSve(y)` možemo da održavamo promenljivu `uvećanje` koja će sadržati zbir svih vrednosti uvećanja, tj. zbir svih argumenata funkcije `uvećajSve`. Prema tome, `uvećajSve(y)` prosto dodaje vrednost `y` promenljivoj `uvećanje`, što se realizuje u složenosti $O(1)$. Da bi struktura funkcionisala ispravno, funkcija `vrednost(x)` treba da vrati zbir vrednosti pridruženoj ključu `x` i promenljive `uvećanje`. Funkcija `umetni(x, y)` ključu `x` treba da dodeli razliku vrednosti `y` i promenljive `uvećanje`.

Implementacija može biti sledeća.

```
map<int, int> m;
int uvecanje = 0;

int vrednost(int x) {
   auto it = m.find(x);
   if (it == m.end()) return 0;
   return *it + uvecanje;
}

void umetni(int x, int y) {
   m[x] = y - uvecanje;
}

void obrisi(int x) {
   m.erase(x);
}
```

```
void uvecaj(int x, int y) {
   m[x] += y;
}

int uvecajSve(int y) {
   uvecanje += y;
}
```

Problem: Dizajniraj apstraktnu strukturu podataka koja podržava naredne operacije.

- `umetni(x)`: umetanje se izvršava i ako je broj `x` već jednom prethodno umetnut u strukturu podataka; drugim rečima, struktura podataka treba da pamti duplikate;
- `y = ukloni()`: ukloni proizvoljan elemenat iz strukture podataka i dodeli ga promenljivoj y. Ako postoji više kopija istog elementa, uklanja se samo jedna od njih.

Ovaj apstraktni tip podataka može se nazvati *skladište* (engl. pool). On se može iskoristiti za smeštanje poslova. Novi poslovi se generišu i umeću u skladište, a kad neki radnik postane raspoloživ, uklanja se neki posao. Sve operacije treba da se izvršavaju za vreme $O(1)$.

S obzirom na to da redosled izbacivanja elemenata nije preciziran postoji zaista mnogo načina da se ovakva struktura realizuje. Na primer, moguće je upotrebiti red ili stek.

```
queue<int> q;

void umetni(int x) {
   q.push(x);
}

int ukloni() {
   int x = q.top();
   q.pop();
   return x;
}
```

Problem: Kako se prethodna apstraktna struktura podataka može prilagoditi tako da se proizvoljan elemenat može pojaviti najviše jednom u strukturi podataka? Umetanje mora da bude praćeno proverom postojanja duplikata. Realizovati iste operacije kao u prethodnom slučaju, ali sa proverom postojanja duplikata. Koja je složenost izvođenja operacija?

Ako ne znamo raspon iz kojeg dolaze elementi, uz red možemo čuvati i skup elemenata koji se trenutno nalaze u strukturi podataka.

```
queue<int> red;
set<int> elementi;

void umetni(int x) {
  if (elementi.find(x) == elementi.end()) {
    red.push(x);
    elementi.insert(x);
  }
}

int ukloni() {
  int x = red.top();
  elementi.erase(x);
  red.pop();
  return x;
}
```

Složenost operacija je $O(\log n)$. Umesto `set` možemo koristiti i `unordered_set` uz prosečnu složenost $O(1)$ (ali složenost najgoreg slučaja $O(n)$).

Ako se unapred zna da će svi elementi biti iz nekog intervala $[0, n]$, onda umesto pomoću bibliotečkih klasa skup možemo realizovati pomoću niza logičkih vrednosti.

Kada se ne bi zahtevalo da se elementi uklanjaju u redosledu njihovog umetanja, tada bi se red mogao izbaciti i uvek izbacivati prvi (najmanji) element iz skupa.

```
int ukloni() {
   int x = *elementi.begin();
   elementi.erase(elementi.begin());
   return x;
}
```

Problem: Kako se prethodna apstraktna struktura podataka može prilagoditi tako da se svakom umetnutom elementu može pridružiti neka vrednost? Potrebno je realizovati sledeće operacije.

- `umetni(x, y)` - ključu `x` pridružuje se vrednost `y`. Ako je tom ključu vrednost dodeljena i ranije, ona se zanemaruje.
- `y = ukloni()` - uklanja i vraća proizvoljnu vrednost pridruženu nekom ključu.
- `nađi(x)` - vraća vrednost pridruženu nekom ključu.

Rešenje je slično kao prethodno, ali umesto skupa moramo upotrebiti mapu. Ako želimo da redosled uklanjanja odgovara redosledu umetanja, održavaćemo i red elemenata.

```
queue<int> red;
map<int, int> elementi;

void umetni(int x, int y) {
```

```
  if (elementi.find(x) == elementi.end()) {
    red.push(x);
    elementi[x] = y;
  }
}

int ukloni() {
  int x = red.front();
  int y = elementi[x];
  elementi.erase(x);
  red.pop();
  return y;
}

bool nadji(int x, int& y) {
  auto it = elementi.find(x);
  if (it == elementi.end())
    return false;
  y = it->second;
  return true;
}
```

Složenost svih operacija je $O(\log n)$. I u ovom slučaju je moguće upotrebiti `unordered_map`. Ako je raspon ključeva iz nekog intervala $[0, n]$, tada se mapa može realizovati pomoću običnog niza.

Ako redosled uklanjanja nije bitan, možemo izbaciti red i uvek brisati prvi element iz mape (onaj sa najmanjom vrednošću ključa).

```
int ukloni() {
   int x = elementi.begin()->first;
   int y = elementi.begin()->second;
   elementi.erase(elementi.begin());
   return y;
}
```

Kada je mapa realizovana pomoću niza, tada moramo održavati i neku listu ubačenih ključeva (red, stek, ...).

Problem: Neka je $S = \{s_1, s_2, \ldots, s_m\}$ vrlo veliki skup, izdeljen u k disjunktnih blokova. Kreirati apstraktnu stukturu podataka koja omogućava rad sa malim podskupovima T skupa S, i to sledeće operacije nad T:

- `umetni(si)` - umeće s_i u T
- `obrisi(si)` - briše s_i iz T
- `obrisiSveIzBloka(j)` - briše iz T sve elemente koji pripadaju bloku j.

Složenost svake od ovih operacija treba da bude $O(\log n)$, gde je n tekući broj elemenata u T. Pretpostaviti da su m i k veoma veliki brojevi, a da je n manji od njih.

Jednostavnosti radi pretpostavimo da je S skup celih brojeva. Možemo definisati mapu kojom se svakom elementu pridružuje broj bloka kom pripada.

```
map<int, int> blok;
```

Podskup T onda možemo predstaviti mapom kojom se svaki redni broj bloka slika u skup elemenata koji pripadaju tom bloku i podskupu T (umesto mape možemo koristiti i niz dimenzije k, ali pošto je n dosta manje od k mapom se štedi memorija, jer neće svi blokovi imati elemente u T).

```
map<int, set<int>> podskup;
```

Operacije je onda relativno jednostavno implementirati.

```
void umetni(int x) {
  int b = blok[x];
  podskup[b].insert(x);
}

void obrisi(int x) {
  int b = blok[x];
  podskup[b].erase(x);
}

void obrisiSveIzBloka(int b) {
  podskup.erase(b);
}
```