1. Apsolutni pobednik na glasanju

vreme	memorija	ulaz	izlaz
0,1 s	64 Mb	standardni ulaz	standardni izla

Apsolutni pobednik izbora je onaj ko osvoji bar jedan glas više od polovine izašlih birača. Ako su poznati svi glasački listići, odredi da li postoji apsolutni pobednik izbora.

Ulaz

Sa standardnog ulaza se unosi broj glasača nn, a zatim i glasovi (svaki glas predstavlja šifru nekog kandidata - ceo broj iz intervala $[0,10^9]$).

Izlaz

Na standardni izlaz ispisati broj pobednika ako postoji apsolutni pobednik, tj. nema u suprotnom.

Primer 1

Ulaz
```
10
342
123
342
756
123
756
123
756
756
756
```
Izlaz
```
nema
```

Primer 2

Ulaz
```
13
342
123
342
756
123
756
123
756
756
```

756
342
756
756
Izlaz
756

REFORMULACIJA:
Neka je *E* zadati niz brojeva x[0], x[1],..., x[n-1]. Višestrukost broja x u E je broj pojavljivanja broja x u E. Odrediti element E višestrukosti veće od n/2 ("preovlađujući element") ili ustanoviti da takav ne postoji.

na primer: 2 3 2 5 5 5 5 2 5 1 5 4 5 = > 5 je preovlađujući

Ideja 1 : izvrši se sortiranje, te ako postoji preovlađujući broj, on je u sredini i izvrši se za njega provera pozicije u sortiranom nizu. Može li efikasnije?

Ideja 2: Odrediti medijanu (videti u knjizi poglavlje o rangovskim statistikama - medijana je n/2-ti najmanji element), te je nađen kandidat za proveru. Može li efikasnije od O(*n* log *n*)?

PODSEĆANJE: *Formulacija problema rangovskih statistika: Za zadati niz elemenata S sa n članova i broj k, 0 <= k < n, odrediti k-ti najmanji elemenat u S.* Kao i kod sortiranja razdvajanjem, loš izbor pivota vodi kvadratnom algoritmu.

Ideja 3: Ako je element x[i] različit od elemnta x[j] i ako se izbace oba ova elementa iz niza, onda ako postoji *y* koji je preovlađujući element starog niza, onda je on preovlađujući element i novog niza (obratno ne važi).

U algoritmu se u jednom prolazu koriste promenljive *C*, *M*, gde *C* je jedini kandidat za preovlađujući element u nizu x[0], x[1],..,x[i-1].
M je broj pojavljivanja elementa *C* u nizu x[0], x[1],...,x[i-1], bez onih elemenata C koji su izbačeni.

Ako je x[i]==C, onda se M uvećava za 1, a inače smanjuje za 1. (tada se smatra da x[i] može biti izbačen zajedno sa nekim članom jednakim C). Ako M postane 0, onda C dobije novu vrednost x[i] i time i M postane 1.

Algoritam Preovladjuje (x,n):
ulaz: x, n /* niz pozitivnih brojeva x, dimenzije n*/
izlaz: preovlada /* preovladjujuci alaement (ako postoji) ili -1 */

```
C=x[0];
M=1;
for(i=1; i < n; i++)
```

```
    if (M==0)
    {C=x[i]; M=1;}
    else
        if (x[i]==C) M++; else M--;

if (M==0) preovlada=-1;   /*nema preovladjujuceg elementa*/
else brojac=0;

for (i=0; i < n; i++)
    if (x[i]==C) brojac++;

if (brojac > n/2) preovlada=C; else preovlada=-1;
```

REŠENJE apsolutnog pobednika na glasanju
Ovaj problem se u literaturi naziva i **majority voting.**

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

using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  // ucitavamo glasove
  int n;
  cin >> n;
  vector<int> glasovi(n);
  for (int i = 0; i < n; i++)
    cin >> glasovi[i];

  // odredjujemo kandidata za pobednika glasove delimo u dve grupe: u
  // prvoj grupi se svi glasovi mogu ponistiti tako sto se spajaju dva
  // po dva razlicita glasa, a u drugoj grupi su svi glasovi za
  // kandidata za pobednika

  // broj glasova u drugoj grupi
  int brojac = 0;
  // kandidat za pobednika
  int kandidat_za_pobednika;
  for (int i = 0; i < n; i++) {
    // druga grupa je prazna
    if (brojac == 0) {
      // trenutni glas je kandidat za pobednika i ubacujemo ga u drugu
      // grupu
      kandidat_za_pobednika = glasovi[i];
```

```
      brojac = 1;
    } else if (glasovi[i] == kandidat_za_pobednika)
      // trenutni glas je za kandidata i ubacujemo ga u drugu grupu
      brojac++;
    else
      // trenutni glas moze da se ponisti sa jednim glasom za
      // kandidata i oba ih prebacujemo u prvu grupu
      brojac--;
  }

  // proveravamo da li postoji kandidat za pobednika i da li je
  // ostvario vise od n/2 glasova
  if (brojac > 0 &&
      count(begin(glasovi), end(glasovi), kandidat_za_pobednika) > n / 2)
    cout << kandidat_za_pobednika << endl;
  else
    cout << "nema" << endl;
/*
    // rucna implementacija
    // brojimo glasove za kandidata
    int broj_glasova_za_kandidata = 0;
    for (int i = 0; i < n; i++)
      if (glasovi[i] == kandidat_za_pobednika)
        broj_glasova_za_kandidata++;
    // proveravamo da li je osvojio vecinu
    if (broj_glasova_za_kandidata > n / 2)
      cout << kandidat_za_pobednika << endl;
    else
      cout << "nema" << endl;
  */
    return 0;


}
```

2. Dat je niz od ***n*** prirodnih brojeva sa više ponavljanja elemenata, tako da je broj različitih elemenata u nizu O(log n).

a) Konstruisati algoritam za sortiranje ovakvih nizova, u kome se izvršava najviše O(n log log n) upoređivanja brojeva.
b) Zašto je složenost ovog algoritma manja od donje granice Ω(n log n) za sortiranje?

REŠENJE:

a) Svaki element niza umetnuće se kao jedno polje čvora balansiranog binarnog stabla pretrage. Drugo polje čvora će čuvati broj pojava tog elementa u nizu. Ako je broj različitih elemenata u nizu O(log n), onda je

broj čvorova u stablu O(log n),
te je visina stabla O(log log n) => broj upoređivanja je najviše O(n log log n).

Zatim se stablo obilazi inorder obilaskom i njegovi elemeenti se kopiraju u neopadajućem redosledu u izlazni niz dužine *n* tako što se svaki element kopira onoliko puta kolika je vrednost odgovarajućeg brojačkog polja.

b) Opisani algoritam pod a) nije obuhvaćen modelom stabla odlučivanja, jer je pri konstrukciji algortma bilo od značaj da postoji relativno mali broj različitih elemenata u nizu, tj. vrednosti elemenata niza su bile značajne.

3. Segment maksimalnog zbira – jedan problem više rešenja

Problem: Definisati efikasnu funkciju koja pronalazi najveći mogući zbir segmenta (podniza uzastopnih elemenata) datog niza brojeva. Proceni joj složenost.

Kadanov algoritam (dinamičko programiranje)

Prikažimo sada još nekoliko algoritama linearne složenosti za rešavanje ovog problema. Možda najpoznatiji algoritam je Kadanov algoritam, koji se ubraja u algoritme dinamičkog programiranja (o kojima će više biti reči kasnije).

Pokušavamo da algoritam zasnujemo na induktivnoj konstrukciji.

Za prazan niz, jedini segment je prazan i njegov je zbir nula (to je ujedno najveći zbir koji se može dobiti). Smatramo da umemo da problem rešimo za proizvoljan niz dužine *n* i na osnovu toga pokušavamo da rešimo zadatak za niz dužine *n*+1 (polazni niz proširen jednim dodatnim elementom).

Segment najvećeg zbira u proširenom nizu se ili ceo sadrži u polaznom nizu dužine *n* ili čini sufiks proširenog niza, tj. završava se na poslednjoj poziciji (uključujući i mogućnost da je tu i prazan segment).

Na osnovu induktivne hipoteze znamo da izračunamo najveći zbir segmenta. Jedan način je da prilikom svakog proširenja niza iznova analiziramo sve segmente koji se završavaju na tekućoj poziciji, ali čak iako to radimo inkrementalno (krenuvši od praznog sufiksa, pa dodajući unazad jedan po jedan element) najviše

što možemo dobiti je algoritam kvadratne složenosti (probajte da se uverite da je to zaista tako). Ključni uvid je to da najveći zbir sufiksa koji se završava na tekućoj poziciji možemo inkrementalno izračunati znajući najveći zbir segmenta koji se završava na prethodnoj poziciji (tj. najvećeg sufiksa niza pre proširenja). Naime, ako je zbir najvećeg segmenta koji se završava na prethodnoj poziciji i tekućeg elementa pozitivan, onda je to upravo najveći zbir sufiksa proširenog niza (prazan segment ima zbir nula, pa je njegov zbir manji od pronađenog pozitivnog zbira segmenta, a neprazni sufiksi moraju da uključe tekući element i neki sufiks niza pre proširenja, pa nam je jasno da zbir tog sufiksa mora biti određen optimalno). Ako je zbir najvećeg sufiksa pre proširenja niza i tekućeg elementa negativan, onda je optimalan zbir sufiksa nakon proširenja niza 0 (uzimamo prazan segment).

Dakle, proširićemo induktivnu hipotezu i pretpostavićemo da za niz umemo da izračunamo najveći zbir segmenta, ali i najveći zbir sufiksa.

Ako bismo formirali rekurzivnu funkciju koja vrši takvo izračunavanje, dobili bismo neefikasan algoritam jer bi se isti pozivi ponavljali više puta. Umesto toga možemo napraviti iterativan algoritam kome je invarijanta da u svakom koraku petlje znamo ove dve vrednosti (maksimum segmenta i maksimum sufiksa).

```
int maxSufiks = 0, maxSegment = maxSufiks;

for (int i = 0; i < n; i++) {

  maxSufiks += a[i];

  if (maxSufiks < 0)

    maxSufiks = 0;

  if (maxSegment < maxSufiks)

    maxSegment = maxSufiks;

}



cout << maxSegment << endl;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno n puta, pa je složenost ovog algoritma takođe $O(n)$.

4. Zbirovi prefiksa (parcijalne sume niza)

Još jedan algoritam kojim možemo efikasno rešiti ovaj zadatak se zasniva na
korišćenju *zbirova prefiksa*. Naime, ako znamo zbir svakog prefiksa niza, onda zbir svakog segmenta možemo dobiti kao razliku zbirova dva prefiksa. Naime, važi da je

$$\sum_{i=l}^{d} a_i = \sum_{i=0}^{d} a_i - \sum_{i=0}^{l-1} a_i$$

Računamo da je zbir praznog segmenta $\sum_{i=0}^{-1} a_i$ po definiciji jednak nuli.

Obratite pažnju na to da je ova tehnika zapravo analogon
Njutn-Lajbnicove formule koju ste sretali tokom izučavanja matematičke analize.

Ponovo koristimo induktivnu konstrukciju iz prethodnog zadatka i niz proširujemo za jedan po jedan element. Da bismo umeli da izračunamo maksimum segmenta proširenog niza potrebno je da znamo maksimum segmenta polaznog niza i da izračunamo maksimalni sufiks proširenog niza. Na osnovu razlaganja na zbirove prefiksa, maksimalni zbir sufiksa proširenog niza se dobija kao razlika zbira celog proširenog niza (tj. zbira prefiksa do tekuće pozicije) i zbira nekog prefiksa neproširenog niza (prazan sufiks ne moramo analizirati, jer je prazan niz već uzet u obzir u sklopu baze indukcije). Pošto je umanjenik konstantan, da bismo maksimizovali razliku potrebno da znamo najmanji mogući umanjilac, tj. da znamo najmanji zbir prefiksa koji se završava na nekoj poziciji ispred tekuće. I tekući i minimalni zbir prefiksa možemo održavati inkrementalno. Kada niz proširimo jednim elementom, zbir prefiksa uvećavamo za taj element, poredimo ga sa dotadašnjim minimalnim zbirom prefiksa i ako je manji, ažuriramo minimalni zbir prefiksa. Naravno, održavamo i globalni maksimalni zbir segmenta koji ažuriramo svaki put kada naiđemo na segment (sufiks) čiji je zbir veći od dotadašnjeg maksimuma.

Dakle, i u ovom rešenju je induktivna hipoteza pojačana i pretpostavljamo da pored segmenta najvećeg zbira u obrađenom delu niza
umemo da odredimo i maksimalni zbir sufiksa obrađenog dela niza.

```
int zbirPrefiksa = 0;

int minZbirPrefiksa = zbirPrefiksa;

int maxZbir = 0;

for (int i = 0; i < n; i++) {

  zbirPrefiksa += a[i];

  int zbirSufiksa = zbirPrefiksa - minZbirPrefiksa;

  if (zbirSufiksa > maxZbir)

    maxZbir = zbirSufiksa;

  if (zbirPrefiksa < minZbirPrefiksa)

    minZbirPrefiksa = zbirPrefiksa;

}

cout << maxZbir << endl;
```

Analiza složenosti je veoma jednostavna. Telo petlje koje je konstantne složenosti se izvršava tačno *n* puta, pa je složenost ovog algoritma takođe $O(n)$.

5. Maksimalna suma nesusednih elemenata pozitivnog niza

Dat je niz celih brojeva **a** sa **n** elemenata. Odrediti podniz niza **a** čija je suma elemenata maksimalna, a u kome nema susednih elemenata niza **a**. Smatrati da je suma elemenata praznog niza jednaka 0.

TEST PRIMERI

ULAZ(a)	IZLAZ(podniz)
5 2 -1 8 10 -2 -5	5 10
-1 -2 -4 3 1 -2 1	3 1
-3 4 -5 -4 1 18 12	4 18

REŠENJE: Neka je a niz od n celih brojeva.
Neka je R=R(A) jedan podniz iz formulacije zadatka koji odgovara nizu a.
Neka je A_k niz koji se sastoji od prvih k elemenata niza a.

Podniz niza A_0 je prazan niz, dok je podniz niza A_1 element a[1] ako je a[1]>0.

1. Ako element a[n] ne pripada podnizu R=R(A), onda je R podniz iz formulacije zadatka koji odgovara nizu A_{n-1}.
2. Ako element a[n] pripada podnizu R=R(A), onda je R \ a[n] podniz iz formulacije zadatka koji odgovara nizu A_{n-2}.

Dakle, za i=2..n, važi da $R(A_i)$ je ili $R(A_{i-1})$ ili $R(A_{i-2})$* {a[i]}, tj. uzima se onaj od podnizova koji ima veci zbir.

Neka je niz s takav da:
s(i) je suma podniza $R(A_i)$.
Iz gore izlozenog, jasno je da:
s(0)=0,
s(1)=max(0, a[1]),
s(i)=max{ s(i-1), s(i-2) + a[i] }, i=2..n

Iz niza s će se ispisati rešenje R.

```
Algoritam MSN (a,n)
ulaz: a, n   /* niz a duzine n, BSO clanovi su a[1]...a[n] */
izlaz: podniz niza a cija suma je maksimalana,
gde podniz ne sadrzi uzastopne elemente iz a
{
  s[0]=0;       /* s[j] = suma elemenata podniza b koji je resenja zadatka
za podniz A(j) */
                /* s[0]=0, jer suma praznog niza je 0, po dogovoru */
if (a[1] > 0)  s[1]=a[1];
            else s[1]=0;


for (i=2;i <= n; i++)  /*uzimamo da indeks prvog clana niza je 1, drugog
je 2,...*/

    if  (s[i-2] + a[i]  > s[i-1])
          s[i]= s[i-2] + a[i];
     else s[i]= s[i-1];

MSN_ispis (n);
}


procedure  MSN_ispis (n)
ulaz: n
izlaz: ispis clanova podniza koji sadrzi nesusedne clanove niza,
       ali tako da je suma podniza maksimalna
{
```

```
if (n > 0)
  if (s[n] = = s[n-1)  MSN_ispis(n-1);
  else
    {  MSN_ispis(n-2);
       ispisati  a[n];   /*jer je clan niza b */
    }

}
```

Reformulacija problema: Napiši program koji određuje najveći zbir podniza datog
niza nenegativnih brojeva koji ne sadrži dva uzastopna člana niza. Na primer, za niz 7, 3, 2, 4, 1, 57,3,2,4,1,5 najveći takav podniz je 7, 4, 57,4,5,
čiji je zbir 1616.

Pokušamo da zadatak rešimo induktivno-rekurzivnom konstrukcijom. Niz elemenata možemo razložiti na poslednji element i prefiks bez njega. Maksimalni zbir je veći od dva zbira: prvog koji se dobija tako što se poslednji element (jer je uvek nenegativan) doda se na maksimalni zbir elemenata prefiksa koji ne uključuje pretposlednji element i drugog koji se dobija kao zbir elemenata prefiksa koji uključuje pretposlednji element. Dakle, pretpostavljamo da za prefiks znamo maksimalni zbir bez njegovog poslednjeg i sa njegovim poslednjim elementom. Zato ojačavamo induktivnu hipotezu i pretpostavljamo da za svaki prefiks niza umemo da odredimo upravo te dve vrednosti. Bazu indukcije čini jednočlani prefiks niza. Maksimalni zbir sa uključenim njegovim jedinim elementom jednak je tom elementu, dok je maksimalni zbir bez njega jednak nuli. Induktivnu hipotezu proširujemo tako što prilikom dodavanja novog elementa maksimalni zbir tekućeg prefiksa sa tim novim elementom određujemo kao zbir tekućeg elementa i zbira prethodnog prefiksa bez tog elementa, dok maksimalni zbir tekućeg prefiksa bez tog novog elementa određujemo kao veći od maksimalnog zbira prethodnog prefiksa sa njegovim poslednjim i maksimalnog zbira prethodnog prefiksa bez njegovog poslednjeg elementa. Kada se petlja završi, veći od dva maksimalna zbira predstavlja traženi globalni maksimum.

```
#include <iostream>

#include <algorithm>

using namespace std;
```

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

 int x;
 cin >> x;
 int maks_zbir_bez = 0;
 int maks_zbir_sa = x;
 for (int i = 1; i < n; i++) {
  int x;
  cin >> x;
  int novi_maks_zbir_bez =
   max(maks_zbir_sa, maks_zbir_bez);
  maks_zbir_sa = maks_zbir_bez + x;
  maks_zbir_bez = novi_maks_zbir_bez;
 }

 cout << max(maks_zbir_bez, maks_zbir_sa) << endl;

 return 0;
}
```

Složenost ovog algoritma je prilično očigledno $O(n)$.

6. Neka **n** ljudi čeka u redu da kupi karte za predstavu, pri cemu $\mathbf{t_i}$ je vreme koje je **i**-tom kupcu potrebno da kupi kartu. Ako se po dvoje

suseda u redu udruzi da kupi karte - na primer **k**-ti i **k+1**-vi kupac-onda vreme potrebno da oni kupe karte je $\mathbf{p_k}$, k=1..n-1. Udruzivanjem kupaca moze da se ubrza kupovina karata, a i ne mora. Ulazni podaci su broj kupaca **n** i nizovi **t** i **p**. Konstruisati algoritam koji odredjuje takav nacin udruzivanja da ukupno vreme potrebno da svih **n** kupaca kupi kartu bude minimalno.

Resenje: Neka je niz **a** takav da **a[k]** je usteda u vremenu kupovine karata koja nastaje udruzivanjem **k**-tog i **k+1**-vog kupca, tj:
a[k]=t[k]+t[k+1]-p[k]
Jasno da clanovi niza **a** u opstem slucaju su i pozitivni i negativni celi brojevi.
Dalje, **k**-ti covek se ne moze istovremeno udruziti sa **k+1**-vim i **k-1**-vim covekom,
tj. odatle sledi da u nizu ne mogu istovremeno i a[k-1] i a[k] biti ustede, tj.potrebno je pronaci podniz niza **a** koji ima najvecu sumu u kojoj ne ucestvuju susedni clanovi, a to je upravo problem koji je tema prethodnog zadatka.

7. STEK

Дата је ниска коју чини N малих слова енглеске абецеде. Над ниском дефинишемо операцију Sazimanje на следећи начин: уклањање серије од K узастопних једнаких слова (1≤K≤N). Конструисати алгоритам сложености O(N) који ће над датом улазном ниском примените операцију Sazimanje докле год је могуће и исписати резултујућу ниску. (Резултујућа ниска је јединствена.) У првом реду стандардног улаза налазе се два броја N, K ($1 \le K \le N \le 10^5$). У другом реду стандардног улаза дата је полазна ниска која се трансформише на описани начин. На стандардни излаз исписати тражену ниску.

Пример улаза

~~~

9 4

abbbccccb

~~~

Пример излаза

~~~

a
~~~

~~~

Решење: Идеја решења је да се користи стек који на врху памти текући карактер из улазне ниске и број узастопних појава. Када наиђе текући карактер у нисци, пореди се са врхом стека.

Ако је врх стека једнак карактеру, увећа се број појава.

(и евентуално скине врх стека ако је број појава једнак K).

Иначе, се поставља нови карактер на стек.

На крају се испише садржај са стека.

```
#include <iostream>
#include <deque>
using namespace std;
int main()
{
   int n, k;
   string str;
   cin >> n >> k >> str;
// Pravimo stack stk kao deque, da na kraju kad stampamo rezultujucu jedinstvenu nisku
//ne moramo da obrcemo nisku
   deque< pair<char, int> > stk;

// jednim prolazom kroz ulaznu nisku vrsimo obradu
   for (int i = 0; i < n; ++i)
   {
      if (!stk.empty() && stk.back().first == str[i])

// ako se tekuci element ulazne niske poklapa sa vrhom steka, uvecajmo broj pojava tog karaktera
         stk.back().second += 1;
```
~~~

```
        else
            // u suprotnom, tekuci karakter ide na vrh steka
            stk.push_back({str[i], 1});

        if (stk.back().second == k)  // Ako poslednje malo slovo na steku ima broj pojava jednak sa k, uklonimo vrh steka
            stk.pop_back();
    }

    while (!stk.empty())    // stampamo rezultat
    {
        cout << string(stk.front().second, stk.front().first);

        stk.pop_front();
    }
    cout << endl;
    return 0;
}
```

8. Najveći pravougaonik u histogramu

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

Hint: Prikažimo rad algoritma koji koristi stek 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
```

9. DFS nerekurzivno

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

```
#include <iostream>

#include <vector>
```

```cpp
#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;
}
```

10. Red pomocu dva steka

Problem: Implementiraj funkcionalnost reda korišćenjem dva steka.

Решење: Имплементација реда помоћу два стека није јединствена. Једна од операција enQueue (додавање) или deQueue (уклањање елемента из реда), у зависности од имплементације, неће бити ефикасна.

Идеја 1 (Операција enQueue има већу временску сложеност)

У овом решењу, елемент који је први ушао у ред је увек на врху у стеку 1, тако да операција deQueue врши уклањање са стека 1 (POP).

Кад желимо да ставимо елемент на врху у стеку 1, онда користимо стек 2.

enQueue(q, x) - ДОДАТИ вредност x на крај реда q

1) Док стек 1 није празан, додају се на стек 2 сви елементи са стека1.

2) Додати вредност x на стек 1 (под претпоставком да радимо са унапред неограниченим стековима).

3) Пребацити све елементе (операцијом PUSH и POP) са стека 2 назад на стек 1.

deQueue(q) - БРИСАЊЕ елемента са почетка реда q

1) Ако стек 1 није празан, онда исписати поруку о грешци

2) Скини елемент са врха стека1 и врати га као резултат операције deQueue(q)

Идеја 2 (Операција deQueue има већу временску сложеност)

У овом решењу, у операцији enQueue, нови елемент се додаје на врх стека 1.

У операцији deQueue, ако стек2 је празан, онда се сви елементи померају на стек2 и као резултат deQueue враћа врх стека 2.

enQueue(q, x) - ДОДАТИ вредност x на крај реда q

1) Додати вредност x на крај реда. Додавање се врши на стек 2 (осим ако је стек 1 празан, јер се тада врши додавање на стек 1).

deQueue(q) - БРИСАЊЕ елемента са почетка реда q

1) Функција уклања елемент са почетка реда

2) Уклањање се врши са стека 1 и врх стека 1 се враћа као резултат

Ако приликом уклањања први стек остане празан а у други стек су додати нови елементи

Елементи се пребацују са стека 2 на стек 1

Прилажемо имплементацију идеје 2:

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

using namespace std;

struct Red {

    // Red se cuva pomocu dve strukture stek
    stack<int> s1; // Prvi stek se koristi za skidanje elemenata
    stack<int> s2; // dok se drugi stek koristi za dodavanje

    // Funkcija dodaje novi element na kraj reda, dodavanje se vrsi na
dugi stek,
    // jedini izuzetak je dodavanje na prvi stek ako je prvi stek prazan
```

```
   void enqueue(int n)
   {
      if(s1.empty())
         s1.push(n);
      else
         s2.push(n);
   }

   // Funkcija uklanja element sa pocetka reda, uklanjanje se vrsi sa
prvog steka
   int dequeue()
   {
      if(s1.empty())
      {
         cerr << "Prazan red!" << endl;
         exit(EXIT_FAILURE);
      }

      int front_item = s1.top();
      s1.pop();


      // Ako prilikom uklanjanja prvi stek ostane prazan a u drugi stek su
dodati novi elementi,
      // elementi se prebacuju iz drugog steka u prvi stek
      if(s1.empty())
      {
         while(!s2.empty())
         {
            s1.push(s2.top());
            s2.pop();
         }
      }

      return front_item;
   }


   // Funkcija proverava da li je red prazan, za proveru je dovoljno
proveriti da li je
   // prvi stek prazan, jer se nakon uklanjanja poslednjeg elementa sa
prvog steka svi
   // elementi sa drugog steka prebacuju u prvi stek i time drugi red
ostaje prazan.
   bool empty()
   {
      return s1.empty();
```

```
    }

};

int main()
{
    int n;

    Red red;

    for(int i = 0; i < 10; i++)
    {
        cin >> n;
        red.enqueue(n);
    }

    while(!red.empty())
    {
        cout << red.dequeue() << " ";
    }

    return 0;
}
```

11. 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 bfs(int cvor) {
  int brojCvorova = susedi.size();
```

```
  vector<bool> posecen(brojCvorova, false);
  queue<int> s;
  s.push(cvor);
  while (!s.empty()) {
    cvor = s.front();
    s.pop();
    if (!posecen[cvor]) {
      posecen[cvor] = true;
      cout << cvor << endl;
      for (int sused : susedi[cvor])
        if (!posecen[sused])
          s.push(sused);
    }
  }
}


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