

# Računske vježbe 9

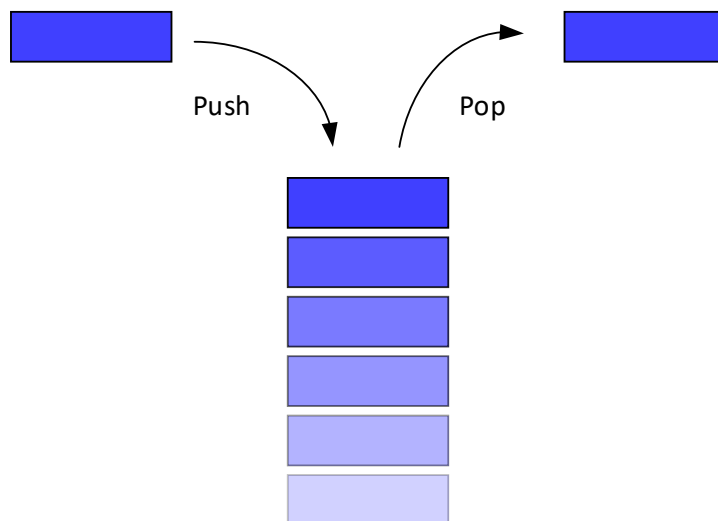
## Programiranje II

1. Napisati program koji provjerava da li je zadati izraz ispravan. Izraz se sastoji od otvorenih i zatvorenih obliha `()`, vitičastih `{}` i uglastih zagrada `[]`.

```
1 #include <iostream>
2 #include <string>
3 #include <stack>
4
5 using namespace std;
6
7 bool brackets(string);
8
9 int main()
10 {
11     string exp = "{{{(([]))}}}";
12     if (brackets(exp))
13         cout << "Izraz je validan!" << endl;
14     else
15         cout << "Izraz nije validan!" << endl;
16 }
17
18 bool brackets(string exp)
19 {
20     stack<char> s;
21     for (int i = 0; i < exp.length(); i++)
22     {
23         if (exp[i] == '(' || exp[i] == '[' || exp[i] == '{')
24             s.push(exp[i]); // vrsimo upis sve dok su zagrade otvorene
25         else
26         {
27             if (s.empty()) // provjeravamo da li je stek prazan
28                 return false;
29             if (exp[i] == ')' && s.top() != '(')
30                 return false;
31             else if (exp[i] == ']' && s.top() != '[')
32                 return false;
33             else if (exp[i] == '}' && s.top() != '{')
34                 return false;
35             s.pop(); // izbacujemo element s vrha
36         }
37     }
38     return s.empty();
39 }
40 }
```

U našem zadatku ćemo se upoznati sa stekom (engl. *stack*) koji predstavlja apstraktni tip podatka odnosno strukturu podataka koja se zasniva na LIFO principu (engl *last in, first out*). Stek se može zamisliti kao gomila tanjira naslaganih jedan na drugi gdje je očigledno da ćemo tanjiru na vrhu lako pristupiti dok nam je pristup tanjirima ispod njega otežan. Elementima steka se ne može pristupiti preko indeksa, moguće

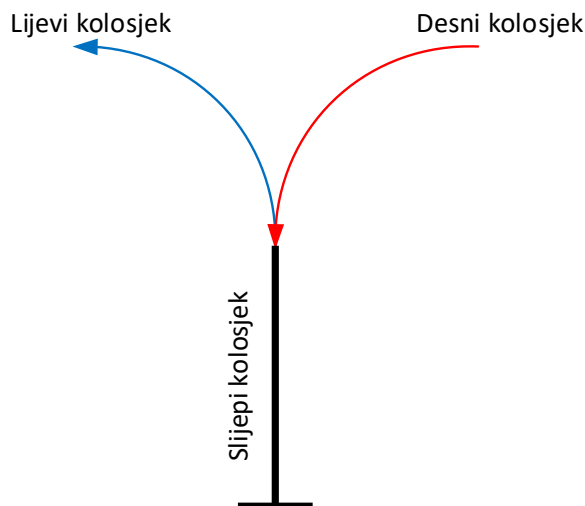
je pristupiti samo elementu na vrhu. Grafička reprezentacija steka se može vidjeti na slici 1. U našem



Slika 1: Grafička reprezentacija steka. Operacija **push** smješta neki element na vrh steka dok operacija **pop** miče element sa steka. Često se nad stekovima implementira još jedna operacija **top** odnosno **peek**. Pomoću nje možemo vidjeti koji je trenutni element na vrhu steka.

zadatku smo koristili stek kako bismo provjerili izraz koji se sastoji od otvorenih i zatvorenih zagrada. Na stek slažemo otvorene zagrade sve dok ne dođemo do prve zatvorene. Kako je na vrhu steka posljednja otvorena zagrada, provjeravamo da li se ta otvorena zagrada poklapa sa posmatranom zatvorenom. Ukoliko se ne poklapaju onda izraz sigurno nije ispravan, a ukoliko se poklapaju otvorena zagrada se miče sa steka i prelazi se na sljedeću zatvorenu.

2. Ka slijepom kolosjeku dolazi voz sa strane označene kao desni kolosjek:



Dozvoljeno je od voza otkočiti jedan ili više vagona (moguće i cio voz) sa njegove prednje strane i uvesti ih na slijepi kolosjek. Vagone je moguće izvesti sa slijepog kolosjeka na lijevi kolosjek, ali ne i vratiti ih na desni kolosjek. Nakon toga je moguće ponovo uvesti vagone sa desnog kolosjeka. Vagone sa lijevog kolosjeka nije moguće vratiti na slijepi kolosjek. Poznato je kojim redom dolaze vagoni. Potrebno je provjeriti da li je moguće reorganizovati voz tako da su vagoni na lijevom kolosjeku poredani kao 1, 2, ... , N.

```

1 #include <iostream>
2 #include <stack>
3 #include <list>
4
5 using namespace std;
6
7 bool reorganize(list<int>);
8
9 int main()
10 {
11     int N;
12     int temp;
13     list<int> track;
14     cout << "Unesite broj vagona na kolosjeku 1:" << endl;
15     cin >> N;
16     cout << "Unesite vagone na kolosjeku 1:" << endl;
17     for (int i = 0; i < N; i++)
18     {
19         cin >> temp;
20         track.push_back(temp);
21     }
22     if (reorganize(track))
23         cout << "Voz se moze reorganizovati!" << endl;
24     else
25         cout << "Voz nije moguće reorganizovati!" << endl;
26 }
27
28 bool reorganize(list <int> rightTrack)
29 {
30     list<int> leftTrack;
31     stack<int> spurTrack;
32     int length = rightTrack.size();
33     int current = 1;
34     list<int>::iterator it;
35     while (true)
36     {
37         if (spurTrack.empty() || spurTrack.top() != current)
38         {
39             it = rightTrack.begin();
40             while (it != rightTrack.end())
41             {
42                 spurTrack.push(*it);
43                 it = rightTrack.erase(it);
44                 if (spurTrack.top() == current) break;
45             }
46         }
47         if (!spurTrack.empty() && spurTrack.top() == current)
48         {
49             leftTrack.push_back(spurTrack.top());
50             spurTrack.pop();
51             current++;
52         }
53         else
54         {
55             break;
56         }
57     }
58     cout << "Na desnom kolosjeku se sada nalaze vagoni: ";
59     for (it = leftTrack.begin(); it != leftTrack.end(); ++it)

```

```

60 {
61     cout << *it << " ";
62 }
63 cout << endl;
64 return length == leftTrack.size();
65 }

```

U ovom zadatku smo slijepi kolosjek predstavili kao stek, a lijeve i desne kolosjeke kao listu. Sa listama smo se upoznali još na predmetu Programiranje I. One omogućavaju efikasno umetanje i izbacivanje elemenata i u unutrašnjosti liste, a ne samo na krajevima. Pristupanje elementima po proizvoljnom redosljedu je vrlo neefikasno. Liste mogu biti jednostruko ili dvostruko povezane odnosno dati čvor liste može pokazivati samo na sljedeći element ili i na onaj iza njega. Elementima liste može da se pristupa samo od početka ili sa kraja! Generička klasa `list` ostvaruje dvostruko povezane liste. Liste se mogu sortirati, obrnuti, spojiti, filtrirati... Brisanje elementa iz liste dok vršimo iteraciju kroz nju može da dovede do problema ukoliko ne ažuriramo iterator na ispravan način. Metoda `erase` briše element na poziciji na koju ukazuje iterator, a vraća iterator koji ukazuje na element koji se nalazio nakon onog koji je obrisao.

Pa kako smo mi to riješili naš zadatak? Kako se lijevi kolosjek može puniti samo iz slijepog, prvo provjeravamo da li je slijepi kolosjek prazan (ukoliko jeste onda sigurno moramo preuzeti vagone iz desnog) ili ukoliko nije da li je vagon na vrhu onaj vagon koji očekujemo. Ukoliko je slijepi kolosjek prazan ili ukoliko nam vagon sa vrha ne odgovara, slijepi kolosjek punimo sve dok ne naiđe onaj vagon koji je po redu. Ukoliko ne uspijemo da dođemo do tog vagona, izvršenje se prekida. Ukoliko do njega dođemo on se prebacuje u lijevi kolosjek. Objasnimo reorganizaciju voza na sljedećem primjeru. Neka je  $N$  jednako 4 i neka vagoni sa desnog kolosjeka dolaze po sljedećem redu [4, 3, 1, 2]. Prvi vagon koji želimo da se nađe na lijevom kolosjeku je 1. Gledamo da li se on nalazi u slijepom kolosjeku koji je u prvoj iteraciji prazan. Kako vagona broj 1 nema, uvozimo vagone sa desnog kolosjeka sve dok ne dođemo do vagona broj 1. Dakle, stanje u steku je sada [1, 3, 4] zato što je na vrhu slijepog kolosjeka posljednji dospjeli vagon. Uvozimo vagon broj 1 u lijevi kolosjek. Sada se slijepi kolosjek sastoji od vagona [3, 4], a desni kolosjek od vagona broj 2. Sljedeći po redu za lijevi kolosjek je vagon broj 2. Posmatramo slijepi kolosjek, ali on se ne nalazi na njegovom vrhu. Uvozimo vagone iz desnog kolosjeka sve dok ne naiđemo na vagon broj 2 i naravno sve dok desni kolosjek nije prazan. Uvezli smo vagon broj 2 pa je slijepi kolosjek sada [2, 3, 4]. Vagon broj 2 prebacujemo u lijevi kolosjek koji sada očekuje vagon broj 3...