ASP TUTORIALI

by bojzi

1.

UVOD U OBJEKTNO ORIJENTIRANO PROGRAMIRANJE PREKO RAZREDA KOMPLEKSNI

Objektno orijentirano programiranje najlakše je promotriti kroz primjer programa s puno komentara, pa ce to tako i biti obavljeno i to kroz razred Kompleksni koji ce raditi s kompleksnim brojevima neke uobicajene svakodnevne radnje. :)

```
// Tipican uvod u C++ program. iostream samo zamijenjuje stdio.h, a posto se
// koristi iostream mora se i napisati using namespace std; (ta linija rezervira
// mjesto u memoriji za sve funkcije koje iostream sadrzi). Ako se ne definira
// biti ce javljena greska da te funkcije (npr. cin i cout) ne postoje (posto
// nisu dobili svoje mjesto u memoriji, pa tako ni nisu stvorene).
#include <iostream>
using namespace std;

// Ovime se otvara nova klasa (razred) iz kojeg ce se "stvarati" novi objekti
// koji ce svi imati svojstva ovog razreda (npr. svoj Re i Im dio).
class Kompleksni{

// U private-u se nalaze dvije varijable koje su najbitnije za svaki stvoreni
// objekt razreda Kompleksni. One su u private-u kako se do njih ne bi mogli
// korisnici koji ce samo htjeti njima pristupiti kroz glavni program.
// Koristenjem private-a one su zasticene tako da se njima mora pristupati kroz,
// za to predvidjene, funkcije GetX(), GetY(), SetX() i SetY().
private:
     float x, y;

// Public dio dostupan je korisnicima koji ce se koristiti funkcijama iz ovog
// dijela.
public:


// Ovo je osnovni konstruktor. Konstruktor sluzi za stvaranje novih objekata iz
// ovog razreda (Kompleksni). Ako se ne zadaju nikakve vrijednosti,
// predodredjene vrijednosti (0,0) biti ce pridruzene varijablama.

     Kompleksni(float Re=0, float Im=0);


// Sljedece cetiri funkcije sluze za rad s varijablama. Posto korisnik ne smije
// jednostavno iz programa pozvati varijable svakog objekta dobio je na
koristenje
// set funkcija s kojima ipak moze uzimati i mijenjati varijable svakog objekta.
// Funkcije se nalaze u istoj klasi kao i varijable tako da mogu "vidjeti" i
mijenjati
// iste, makar su private. Iz tog razloga ove funkcije rade.

     void SetX (float Re);
     void SetY (float Im);
     float GetX ();
     float GetY ();


// Zbrajanje sluzi za zbrajanje dva kompleksna broja. Kao argumente prima upravo
// ta dva kompleksna broja s kojima ce nesto raditi. Naravno, kod upisivanje
// tipa podatka moramo napisati da su dio klase Kompleksni.

     void Zbrajanje(Kompleksni broj1, Kompleksni broj2);
```

```
// Oduzimanje je potpuno analogno zbrajanju
     void Oduzimanje(Kompleksni broj1, Kompleksni broj2);

};

// U nastavku se nalaze definicije konstruktora i funkcija navedenih u razredu
// Kompleksni. Treba primjetiti kako se svaka od tih funkcija mogla direktno
// napisati nakon njegon navodjenja u samom razredu, no ajmo reci da je ovako
// preglednije, a usput se vjezba kako ce to izgledati kad objekti dobe svoje
// zasebne datoteke. Scope operator (::) sluzi kako bi se prikazalo kojem
// razredu pripada koja funkcija.

// Ovo je funkcija koju obavlja konstruktor (koji je deklariran u samom
// razredu). Kao sto se vidi iz njega samoga, on jednostavno uzme vrijednosti
// koje su mu poslane kao argumenti i upisuje ih u varijable (posto je on dio
// razreda i ima pristup istima).
Kompleksni::Kompleksni(float Re, float Im) {
     x = Re;
     y = Im;

}


// Funkcije SetX i SetY, deklarirane u samom razredu, sluze za pridruzivanje
// novih vrijednosti varijablama za svaki novi objekt koji se koristi.
void Kompleksni::SetX(float Re) {
     x = Re;

}


void Kompleksni::SetY(float Im) {
     y = Im;

}


// Funkcije GetX i GetY, takodjer deklarirane u razredu, sluze za dohvacanje
// vrijednosti varijabla svakog objekta. Kad se ne bi koristile ove funkcije vec
// bi se direktno pristupalo tim varijablama, program bi prijavio gresku posto
// on te varijable jednostavno "ne vidi".
float Kompleksni::GetX() {
     return x;

}


float Kompleksni::GetY() {
     return y;

}


// Funkcija koja obavlja zbrajanje. Kao argumente je primila dva kompleksna
// broja koja ce zbrojiti kako bi dobila vrijednost 3. Ovdje je zgodno vidjeti
// kako se, posto je to funkcija razreda Kompleksni, moze direktno pristupati
// podacima koji su private makar bili iz tudjih objekata; bitno je samo da su
// iz istog razreda.
void Kompleksni::Zbrajanje(Kompleksni broj1, Kompleksni broj2) {
     x = broj1.x + broj2.x;
     y = broj1.y + broj2.y;
```

```
    }

// Funkcija oduzimanja analogna je funkciji zbrajanja
void Kompleksni::Oduzimanje(Kompleksni broj1, Kompleksni broj2) {

    x = broj1.x - broj2.x;
    y = broj1.y - broj2.y;

}


// Napokon GLAVNI PROGRAM :)
int main() {

// Stvaranje novih objekata. U ovom primjeru stvaram dva objekta: broj1 i broj2.
// Obojica ce koristiti razred Kompleksni tako da ce obojica imati po dvije
// varijable (x i y, tj. Re i Im dio broja) i cijeli set funkcija koje su u
// samom razredu Kompleksni. broj1 biti ce stvoren kao novi objekt s
// vrijednostima varijabli (0,0), a broj2 kao objekt s vrijednostima varijabli
// (1,2)

    Kompleksni broj1, broj2(1,2), broj3;

// pomx i pomy cu koristiti kako bih mogao sam stavljati nove vrijednosti
// brojevima

    float pomx, pomy;

// Ove dvije naredbe koje slijede (cout) su naredbe za ispis koje se smiju
// koristiti posto smo umjesto stdio.h koristili iostream. cout je C++ inacica
// printf-a, a koristi se tako da se znakovima << samoj funkciji salje string
// koji treba ispisati. Taj nacin je vrlo fleksibilan jer mozemo povezivati puno
// stvari jednu za drugom bez potrebe za puno printf-ova. endl znaci skok u
// sljedecu liniju.

    cout << "Ispis nakon stvaranja novih objekata: " << endl;
    cout << "1. kompleksni broj je: " << broj1.GetX() << " + " <<
    broj1.GetY() << "i, a 2. kompleksni broj je: " << broj2.GetX() << " +
    " <<
    broj2.GetY() << "i." << endl << endl;


// Sljedece cetiri naredbe postaviti ce nove vrijednosti u broj1 i broj2
// koristeci za to predvidjene funkcije koje se nalaze u razredu.

    broj1.SetX(3);
    broj1.SetY(4);
    broj2.SetX(4);
    broj2.SetY(5);

// Takodjer, ispis ce se vrsiti opet pomocu cout-a.

    cout << "Ispis nakon dodijeljivanja novih vrijednosti: " << endl;
    cout << "1. kompleksni broj je: " << broj1.GetX() << " + " <<
    broj1.GetY() << "i, a 2. kompleksni broj je: " << broj2.GetX() << " +
    " <<
    broj2.GetY() << "i." << endl << endl;

// Sljedece linije koda sluze kako bih sam mogao upisati nove vrijednosti u
// varijable svojih brojeva. cout ce samo ispisati string koji mu je zadan.

    cout << "Upisite zeljenu vrijednost x za broj1: ";
```

```
// cin je C++ alternativa scanf-u. On ce u pomx upisati vrijednost koju je
// ucitao s tastature. Primjetite kako sada strelice (>>) idu u suprotnom
// smjeru. To je zato sto te strelica oznacavaju kamo ce se poslati argumenti
// (ajmo tako reci). cout je funkcija koja prima neke argumente pa s njima radi
// (ispisuje) i zato strelicu oznacavaju "utakanje" u cout (<<), a cin upisuje
// argumente u neku varijablu pa su strelice okrenute prema varijabli (>>).
    cin >> pomx;

// Ponovno koristenje SetX funkcije samo ovaj puta saljemo varijablu kao
// argument, a ne goli broj.
    broj1.SetX(pomx);

// Ostatak naredbi analogan je s ove prve tri.
    cout << "Upisite zeljenu vrijednost y za broj1: ";
    cin >> pomy;
    broj1.SetY(pomy);
    cout << "Upisite zeljenu vrijednost x za broj2: ";
    cin >> pomx;
    broj2.SetX(pomx);
    cout << "Upisite zeljenu vrijednost y za broj2: ";
    cin >> pomy;
    broj2.SetY(pomy);

// Ispis kompleksnih brojeva preko cout-a.
    cout << "Ispis nakon vlastitog upisivanja vrijednosti: " << endl;
    cout << "1. kompleksni broj je: " << broj1.GetX() << " + " <<
    broj1.GetY() << "i, a 2. kompleksni broj je: " << broj2.GetX() << " +
    " <<
    broj2.GetY() << "i." << endl << endl;

// Posto je broj3 iz razreda Kompleksni, tako je i on naslijedio svoje dvije
// varijable i funkcije, a izmedju ostalog i Zbrajanje(). Tako ce, pozivajuci
// Zbrajanje s argumentima dva kompleksna broja dobiti njihov zbroj.
    broj3.Zbrajanje(broj1,broj2);

// Standardni ispis pomocu cout-a.
    cout << "Ispis zbroja: " << endl;
    cout << "3. kompleksni broj je: " << broj3.GetX() << " + " <<
    broj3.GetY() << "i." << endl;

// Oduzimanje je analogno zbrajanju.
    broj3.Oduzimanje(broj1,broj2);

// Standardni ispis pomocu cout-a.
    cout << "Ispis razlike: " << endl;
    cout << "3. kompleksni broj je: " << broj3.GetX() << " + " <<
    broj3.GetY() << "i." << endl << endl;

}
```

2.

STOG U SVOJOJ NAJJEDNOSTAVNIJOJ VARIJANTI

Stoga cu se privremeno dotaknuti ovdje samo kako bi se dobio neki mali osjecaj za najjednostavniju implementaciju stoga, a to je preko polja. Necu ulaziti ni u jedan drugi nacin u ovom trenutku, vec ce to biti obradjeno u jednoj od kasnijih tutoriala.

```
// Stog kao takav radi po principu FILO (First In Last Out) i to ste culi
// vjerojatno vec jedno 500 puta. :) Uglavnom, tu se nema sto puno ni nadodati,
// bitno je samo da ne skacete po polju koje ce ostvarivati stog jer se to
// ne smije posto se sa stoga smije dodavati ili uzimati samo s vrha.

// Tipican pocetak programa, ovdje jos nema nikakve magije :)
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

// MAXSTOG ce nam dati podatak o tome koliko cemo maksimalno podataka
// primiti na stog.
#define MAXSTOG 5

// Potprogram Dodaj dodaje zadani broj na vrh stoga. Kao argumente
// mora primati broj koji dodaje na vrh, pokazivac na vrh stoga i
// sam stog. Broj koji se dodaje je logicno da se prosljedjuje, ali
// pitanje je zasto dajemo vrh i sam stog.
// U biti ne dajemo tocno vrh i stog vec samo njihove adrese (pokazivace).
// To radimo zato da bi znali koji je po redu zadnji dodani element (a
// upravo to nam pokazuje pokazivac na vrh), a pokazivac na stog dajemo
// da bi se upisani broj vidio i u glavnom programu, a ne nestao sa
// zavrsetkom potprograma.
int Dodaj(int broj, int *vrh, int *stog) {

// Prvo treba provjeriti da li smo prekoracili vrh stoga i u tom
// slucaju se vracamo u glavni program s 0
     if ((*vrh) >= MAXSTOG) return 0;

// Ako je sve u redu (nema prekoracenja) dodaje se broj na vrh stoga
// i nakon toga se vrh povecava za jedan (zbog ++ koji je stavljen
// iza (*vrh)).
     stog[(*vrh)++] = broj;

// Sve je proslo kako treba, vrati 1.
     return 1;

}
```

```
// Skini skida najvisi element s vrha stoga. Ovdje se kao argumenti salju
// samo vrh i stog iz analogno slicnih razloga kao i u Dodaj. Vrh nam treba
// da bi vidjeli index elementa kojeg brisemo (najviseg), a stog kao takav
// da bi imali uopce sto brisati :).
// E sad, ovdje postoji jedan mali trik. Naime, stog kao takav, tj. kakvog
// mi radimo, ne brise doslovce elemente koje skida, vec samo pomice vrh na
// element "nize" sto bi u praksi znacilo da ce kod sljedeceg citanja, dodavanja
// ili brisanja zanemariti to sto postoje elementi "iznad" vrha.
int Skini (int *vrh, int *stog) {

// Prvo provjeravamo je li vrh stoga 0, tj. postoji li uopce elemenata
// na stogu. Ako ne postoji, ne treba nista ni skidati :)

     if ((*vrh) == 0) return 0;

// Nakon toga koristimo vec spomenuti trik i samo smanjujemo pokazivac
// na vrh stoga.

     (*vrh)--;

// Sve je proslo kako treba, vrati 1.

     return 1;

}


// GLAVNI PROGRAM
void main() {

// jos ce se koristiti za while petlju u kojoj cemo ucitavati
// zeli li korisnik jos raditi s programom ili mu je dosta. broj
// ce sadrzati broj koji dodajemo na vrh, a odabir ce sluziti za
// odabir dodavanja ili skidanja sa stoga.

     int jos=1, broj=0, vrh=0, i, odabir;

// I naravno, sam stog.

     int stog[MAXSTOG];

// Ulazak u petlju koja ispituje zeli li korisnik jos dodavati ili
// skidati sa stoga.

     while (jos) {

// Ovisno o upisu 1 ili 2 program ce skociti na dodavanje ili skidanje
// sa stoga.

          printf ("_____________________________________________________
          ___\n");
          printf ("Upisite zelite li dodati ili skinuti sa stoga(1/2):
          ");
          scanf ("%d", &odabir);

// switch provjerava je li upisano 1 ili 2 i odabire sto ce se raditi

          switch (odabir) {

               case 1: {
```

```
// U slucaju dodavanja na stog treba i preuzeti broj koji
// korisnik zeli dodati na vrh stoga.
                    printf ("\nUnesite broj koji zelite dodati
                    na stog: ");
                    scanf ("%d", &broj);

// Ovdje je razlog zasto sam stavljao return 0 ili 1 na kraju
// potprograma. Kad se vrati 0 ide na else dio, a kada 1 ide na
// ovaj dio odmah ispod if-a.
                    if (Dodaj(broj, &vrh, stog)) {
                    printf ("Broj je dodan na stog!");
                    } else {
                    printf ("Broj nije dodan na stog, nema
                    mjesta!");
                    }

// Mora biti break da se ne bi izvodio i case 2: {
                    break;

               }

               case 2: {

// Opet isti razlog returna kao i kod Dodaj.
                    if (Skini(&vrh, stog)) {
                    printf ("Broj je skinut s vrha stoga!");
                    } else {
                    printf ("Prazan stog!");
                    }

// I opet break.
                    break;

               }

// default kaze sto ce se dogoditi ako je uneseno bilo sto osim 0 ili 1.
               default: printf ("Unesi pravu stvar. :)");
               }

// Ispis stoga. Ovdje se napokon tocno vidi kako stog izgleda.
// Naime, kada se stog ostvaruje preko polja jednostavno se
// svakom elementu dodaje index po kojem se zna koji je on po
// redu. vrh kao takav cuva index zadnje dodanog elementa.
// Ako znamo tu cinjenicu, nije problem putovati po stogu jedan po
// jedan element i igrati se s dodavanjem i skidanjem.
          printf ("\n\nStog izgleda ovako:\n");
          for (i=0; i<vrh; i++) {

               printf ("[%d]: %d\n",i+1, stog[i]);
```

```
// Ova linije je dodana samo kako bi ispisao da je to zadnji
// element stoga. Na taj nacin se moze vidjeti i kako putuje
// vrh stoga koristenjem dodavanja i skidanja s istog.
                if ((i+1) == vrh) printf ("Ovo je bio zadnji
                element stoga.\n");

            }

// Provjera zeli li korisnik jos raditi sa stogom.
            printf ("\nZelite li jos raditi sa stogom? 1 za jos, 0 za
            prekid: ");
            scanf ("%d", &jos);

        }

}
```

3.a

BUBBLE SORT

Bubble Sort je dobio ime po tome sto se u svakom prolazu kroz neki niz nalazi njegov najveci clan i on "ispliva" na pocetak niza. Autori su htjeli reci kako cemo kod prolaza kroz niz najveci clan na koji naidjemo "smjestiti u mjehuric", on ce isplivati na vrh (tamo gdje treba i doci), a nakon toga ce biti izoliran od ostatka, post je vec postao najvecim elementom.

Pitanje je kako ce taj prolaz kroz niz izgledati... Kroz niz se prolazi uvijek usporedjujuci dva susjedna clana. Ako je lijevi veci dolazi do zamjene. Nakon toga se provjera pomice samo za 1 element (tako da se element koji smo mozda upravo zamijenili opet nadje u provjeri) i radi se jednaka stvar. I to je super stvar kod ovog sorta. Dogodi li se da clan koji smo upravo mijenjali nije najveci u nizu, kod nekog od provjera on ce biti ostavljen na mjestu na kojem je bio, tj. sort ce sam izolirati najveci element. Ovo mozda zvuci malo zbunjujuce, pa bi trebalo krenuti na pravu stvar.

Kako to izgleda u stvarnom nizu?

Ovako:

12 55 18 67 88 12 (ovo je nas pocetni niz)

1. Provjera prva dva clana
   12 55 18 67 88 12 <- desni clan je veci, nije potrebna zamjena

2. Provjera sljedeca dva clana

   12 **55 18** 67 88 12 <- desni clan je manji, potrebno je zamijeniti clanove
   -> 2.a: 12 **18 55** 67 88 12

3. Provjera sljedeca dva clana

   12 18 **55 67** 88 12 <- desni clan je veci, ne treba zamjena

4. Provjera sljedeca dva clana

   12 1855 **67 88** 12 <- desni clan je veci, ne treba zamjena

5. Provjera zadnja dva clana

   12 1855 67 **88 12** <- desni clan je veci, treba zamijeniti
   -> 5.a: 12 1855 67 **12 88**

Sada treba uzeti mali predah jer je prvi korak gotov. Najveci element (88) "isplivao" je na vrh niza. Posto je on vec tako postavljen, nestaje potreba za prolaskom kroz cijelo polje, vec se provjeravaju svi clanovi osim ovog postavljenog.

12 18 55 67 12 88(ovo je nas niz nakon prvog koraka)

1. Usporedjujemo prva dva clana

   12 18 55 67 12 88 <- desni je veci, ne treba zamjena

2. Usporedjujemo sljedeca dva clana

   12 **18 55** 67 12 88 <- desni je opet veci, ne treba zamjena

3. Usporedjujemo sljedeca dva clana

12 18 **55 67** 12 88 <- desni je opet veci, ne treba zamjena

4. Usporedjujemo sljedeca dva clana

12 18 55 **67 12** 88 <- desni je manji, potrebna je zamjena
-> 4.a: 12 18 55 **12 67** 88

Nakon ovog koraka isplivao je sljedeci clan po velicini (67), tako da ce i on biti iskljucen iz daljnjeg sortiranja.

12 18 55 12 67 88 (pocetni niz u 3. koraku)

1. Usporedjujemo prva dva clana

12 18 55 12 67 88 <- desni je veci, ne treba zamjena

2. Usporedjujemo sljedeca dva clana

12 **18 55** 12 67 88 <- desni je veci, ne treba zamjena

3. Usporedjujemo sljedeca dva clana

12 18 **55 12** 67 88 <- desni je manji, treba zamijeniti
-> 3.a: 12 18 **12 55** 67 88

I opet imamo slucaju "isplivavanja" i odrezivanja slozenog elementa od daljnjeg ispitivanja.

12 18 12 55 67 88 (ovo je nas niz u 4. koraku)

1. Usporedjujemo prva dva clana

12 18 12 5567 88 <- desni clan je veci, ne treba zamjena

2. Usporedjujemo druga dva clana

12 **18 12** 5567 88 <- desni clan je manji, potrebno je zamijeniti
-> 2.a: 12 **12 18** 5567 88

I jos jedan korak je gotov. 18 je isplivao na vrh dozvoljenog dijela niza i ostaju jos samo dva clana. Njih usporedjujemo na isti nacin kao i do sada, no to vise necu raditi jer je ocito da su jednaki.

Eto, to bi bio bubble sort. Nije uopce problematican kad se ovako skuzi ;).

Algoritam (za polje a[] koje sadrzi n clanova):

Kod:

```
// Ovaj for odredjuje kroz koliko ce se polje prolaziti
// u sljedecem prolazu (tj. smanjuje polje za 1 clan
// u svakom novom koraku).
for (i = 0; i < n-1; i++) {
// Ova for petlja prolazi kroz to polje i usporedjuje
// dva po dva clana, te ih zamjenjuje ako za to postoji
// potreba.
    for (j = 0; j < n-i-1; j++) {
        if (a[j] > a[j+1]) {
            tmp = a[j];      // Ovo je zamjena
            a[j] = a[j+1];   // dvoje susjednih
            a[j+1] = tmp;    // clanova.
        }
    }
}
```

3.b

SELECTION SORT

Selection sort, kao sto mu samo kaze, selektira, tj. odabire nesto. I to je istina, naime, on odabire najmanji clan u polju. Nakon sto je odabrao taj najmanji clan u polju salje ga na pocetak niza, zamijenjujuci ga s elementom koji je bio na tom mjestu. Nakon toga, polje se skracuje za taj, poslozeni, clan i krece iz pocetka. Razlika izmedju Selection i Bubble Sorta je ta sto u Selection Sortu nema "isplivavanja" clanova (stalnim usporedjivanjem), vec se kroz polje prolazi ne radeci nista i na kraju se najmanji clan posalje na prvo mjesto (mijenjajuci mjesta s tadasnjim prvim clanom).

Sam algoritam u pocetku uzme prvi clan (tog koraka) za najmanji i onda prolazi kroz polje trazeci najmanji clan.

Kako to izgleda u stvarnom nizu?

12 55 18 67 88 12 (ovo je nas pocetni niz)

1. 12 se uzima kao najmanji clan (**MIN = 12**)
2. Usporedjuje se kroz cijeli niz
   -> 2.a: 12 **55** 18 67 88 12
   -> 2.b: 12 55 **18** 67 88 12
   -> 2.c: 12 55 18 **67** 88 12
   -> 2.d: 12 55 18 67 **88** 12
   -> 2.e: 12 55 18 6788 **12**

U nasem prvom koraku nije doslo ni do kakve promjene posto je najmanji clan vec bio na prvom mjestu (ova 12-ica na kraju nije zamijenjena jer se gleda da clan bude strogo manji od MIN clana).

12 **55 18 67 88 12** (niz u 2. koraku)

1. 55 se uzima kao najmanji clan (**MIN = 55**)
2. Usporedjuje se kroz niz

   -> 2a: 12 55 **18** 67 88 12 <- 18 je manji od 55 i postaje najmanjim clanom (**MIN = 18**)
   -> 2b: 12 55 18 **67** 88 12
   -> 2c: 12 55 18 67 **88** 12
   -> 2d: 12 55 18 67 88 **12** <- 12 je manji od 18 i **MIN = 12**

3. Posto je najmanji clan 12 na zadnjem mjestu, zamjenjuje se s prvim mogucim mjestom (a to je drugo mjesto u nasem nizu, posto je prvo mjesto poslozeno i izolirano u 1. koraku).

   12 **55** 18 67 88 **12** -> 12 **12** 18 67 88 **55**

U 2. koraku poslozili smo drugi element i idemo dalje po istom principu kao i do sada.

12 12 **18 67 88 55** (niz u3. koraku)

1. Biramo 18 kao najmanji clan (**MIN = 18**)

2. Prolazimo kroz ostatak niza

-> 2a: 12 12 18 **67** 88 55
-> 2b: 12 12 18 67 **88** 55
-> 2c: 12 12 18 67 88 **55**

Najmanji clan je opet na prvom mjestu i nije bilo nikakve potrebe da se clanovi zamijenjuju.

12 12 18 **67 88 55** (niz u 4. koraku)

1. Biramo 67 kao najmanji clan (**MIN = 67**)
2. Prolazimo kroz ostatak niza

-> 2a: 12 12 18 67 **88** 55
-> 2b: 12 1218 67 88 **55** <- 55 je manji od 67 i stoga **MIN = 55**

3. Zamijenjujemo najmanji clan s prvim mogucim

12 12 18 **67** 88 **55** ->12 12 18 **55** 88**67**

Ostao je jos samo jedan korak (posto su jos samo dva clana) i nas Sort je gotov!

12 12 18 67 **88 55** (niz u 5. koraku)

1. Biramo 88 kao najmanji clan (**MIN = 88**)
2. Prolazimo kroz ostatak niza (ogroman li je :))

-> 2a: 12 12 18 67 88 **55** <- 55 je manji od 88 i stoga **MIN = 55**

3. Zamjena najmanjeg clana s prvim mogucim

12 12 18 67 **88 55** -> 12 12 18 67 **55 88**

I to je Selection Sort. Nista pametno, jel tako? ;)

Algoritam (opet za polje a[] i broj clanova n):

Kod:

```
// Petlja "i" prolazit ce kroz polje i svaki korak ga smanjivati
// za 1, vec slozeni, clan.
for (i=0; i < n-1; i++) {
// Takodjer, potrebno je odmah postaviti min clan i index min
// clana (zbog eventualnog nenalazenja jos manjeg clana)
    min = a[i];
    minind = i;
// "j" petlja ce prolaziti kroz ostale clanove polja i traziti
// najmanji clan.
        for (j = i+1; j < n; j++) {
            if (a[j] < min) {
                min = a[j];
                minind = j;
           }
// Nakon svega vrsi se zamjena clanova.
        pom = a[i];
        a[i] = a[minind];
        a[minind] = pom;
   }
}
```

3.c

INSERTION SORT

Insertion Sort, kao takav, dobio je takodjer ime po svom principu rada. Taj princip rada kaze otprilike sljedece: usporedjuj mi susjedne elemente; ako naidjes na to da je desni element manji, vracaj ga skroz dok ne nadjes mjesto gdje ga mozes umetnuti.

Takav Sort dosta podsjeca na bubble (zbog ovog usporedjivanja susjednih elemenata). Princip je ovakav: krene se od pocetka niza i usporedjuju se susjedni elementi. Cim se naleti na to da desni element bude manji od lijevog potrebni ih je zamijeniti, ali to nije sve! :) Osim sto ih treba zamijeniti, potrebno je s tim elementom (bivsim desnim) putovati kroz niz prije sve dok ne naidje na element koji je manji od njega, te ga staviti ispred tog elementa. Nakon toga se nastavi tamo gdje je usporedjivanje stalo. Zbunjujuce? Evo primjer, pa ce postati jasno i, nadam se, lagano. ;)

Samo mala napomenica - ovdje cu racunati jedan korak tek kad se dogodi neka promjena medju clanovima.

Kako to izgleda u stvarnom nizu?

12 55 18 67 88 12 (ovo je nas pocetni niz)

1. Usporedjujemo prva dva clana

   12 55 18 67 88 12

2. Usporedjujemo sljedeca dva clana

   12 **55 18** 67 88 12 <- 18 je manje od 55, treba "umetnuti" u elemente ispred 55
   -> 2a: 12 **55 55** 67 88 12 <- prvo 55 dolazi na mjesto prijasnjeg clana
   -> 3a: 12 **18 55** 67 88 12<- 18 je veci od 12 i ne treba vise "putovati"

Mozemo reci kako je prvi korak gotov, tj. obavljeno je umetanje elementa 18 na pravo mjesto. Krenimo dalje.

12 18 **55 67 88 12** (ovo je nas pocetni niz)

1. Usporedjujemo clanove s mjesta na kojem smo stali

   12 18 **55 67** 88 12

2. Usporedjujemo sljedece clanove

   12 1855 **67 88** 12

3. Usporedjujemo opet dalje clanove

   12 1855 67 **88 12** <- 12 je manje od 88, potrebno je "umetnuti" 12 na pravo mjesto
   -> 3a: 12 1855 67 **88 88** <- 88 dolazi na mjesto prijasnjeg clana, **pamtimo 12**
   -> 3b: 1218 55 **67 67** 88 <- 67 dolazi na sljedece mjesto
   -> 3c: 12 18 **55 55** 67 88 <- 55 dolazi na sljedece mjesto
   -> 3d: 12 **18 18** 55 67 88 <- 18 dolazi na sljedece mjesto
   -> 3e: 12 **12** 18 55 67 88 <- Napokon, 12 dolazi na svoje mjesto (koje je ispred prvog elementa posto se uvjet postavlja kao strogo manje)

Posto smo onim prolaskom kroz niz, trazeci manji clan s kojim se moze "putovati" prosli cijeli niz, Sort je gotov.

Algoritam (polje a[], broj clanova n):

Kod:

```
// Imat cemo samo jednu petlju (no to ne znaci da slozenost
// nije n^2). Ona ce prolaziti kroz clanove polja, a na pocetku
// ce generirati i (za jedan manji od j) i pomocni element.
// To ce koristiti u while petlji koja mijenja clanove (kao
// u 2. koraku naseg primjera) i simulira putovanje kroz niz
// clanova. Kada se dodje do elementa koji je manji od onog
// koji "umecemo" prekida se while petlja i element se "umece"
for (j = 1; j < n; j++) {
    i = j - 1;
    pom = a[j];
    while ((i >= 0) && (pom < a[i])) {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = pom;
}
```

3.d

SHELL SORT

Shell Sort je genijalan, ali je muka Isusova za objasnit. :)

Osobno mislim da nam na MI nece dati da slozimo cijeli Shell Sort vec samo neki korak Shell Sort-a, ali pokusat cu ovdje proci kroz cijeli...

Osnovna ideja Shell Sort-a je da se ide po koracima. Korak je oznaka razmaka izmedju elemenata koji ce se usporedjivati. Konkretno bi to znacilo da se za sortiranje 3. koraka gledaju elementi udaljeni za 3 mjesta (npr. 1. i 4., 2. i 5. itd). Osim sto se gledaju na taj nacin, mora se gledati i retroaktivno. To bi znacilo da, ako postoji dovoljno prostora ispred elemenata koje smo gledali, treba gledati i u raniji niz s istim korakom (ovo ce postati jasnije kroz primjer).
Nakon zavrsenog koraka prelazi se na sljedeci korak koji se moze odredjivati na bilo koji princip, ali najcesce se uzima korak/2. Prvotni korak se najcesce odredjuje sa n/2.

Znam da je ovo malo sve zbunjujuce (pogotovo ovo s korakom), ali evo pokusat cu prvo napraviti mali primjer koraka 3 na malo vecem nizu cisto da vidite kako to pici, a kasnije cemo preci na nas mali primjerni niz.

13 22 10 65 11 16 35 89 77 5 (nas pocetni niz za ovaj primjer)

1. Posto je korak 3 krecemo s provjerom
   13 22 10 **65** 11 16 35 89 77 5 <- sve u redu ne treba mijenjat

2. Sljedeci element

   13 **22** 10 65 **11** 16 35 89 77 5 <- treba zamijeniti, 11 je manji od 22
   13 **11** 10 65 **22** 16 35 89 77 5

3. Sljedeci element

   13 11 **10** 65 22 **16** 35 89 77 5 <- sve u redu ne treba mijenjati

4. Sljedeci element

   -> 4a: 13 11 10 **65** 22 16 **35** 89 77 5 <- potrebno je zamijeniti
   -> 4b: 13 11 10 **35** 22 16 **65** 89 77 5 <- ovdje cemo vidjeti ono o cemu sam pricao
   ->4c: **13** 11 10 **35** 22 16 65 89 77 5 <- treba provjeriti i za korak 3 unazad; kod nas je to u redu

5. Sljedeci element

   13 11 10 35 **22** 16 65 **89** 77 5 <- u redu

6. Sljedeci element

   13 11 10 35 22 **16** 65 89 **77** 5 <- u redu

7. Sljedeci element

-> 7a: 13 11 10 35 22 16 **65** 89 77 **5** <- 5 je manji od 65, treba zamijeniti
-> 7b: 13 11 10 35 22 16 **5** 89 77 **65** <- i opet treba putovati unazad po koraku 3
-> 7c: 13 11 10 **35** 22 16 **5** 89 77 65 <- potrebna zamjena!
-> 7d: 13 11 10 **5** 22 16 **35** 89 77 65 <- i opet putovanje!
-> 7e: **13** 11 10 **5** 22 16 35 89 77 65 <- i opet zamjena!
-> 7f: **5** 11 10 **13** 22 16 35 89 77 65 <- sad se vise nema kamo putovati

---

Za ovakav niz se kaze da je 3-sortiran! Nadam se da ste shvatili bit oko ovog putovanja. Najbitnije je da pamtite kako razlika izmedju clanova mora biti 3 kod 3-koraka (4-1 = 3), 5 kod 5-koraka itd. Takodjer, cisto primjera radi, sljedeci korak kod ovog sortiranja bi bio 1, tj. usporedjivao bi se svaki clan sa svakim i niz bi postao poslozen (s time da se morate sjetiti da je ovaj sort retroaktivan tako da ce gledati i unazad da li je sve poslozeno).

Ajmo mi sad na nas primjeric pa da vidimo koliko ce to biti efikasno...

Kako to izgleda u stvarnom nizu?

12 55 18 67 88 12 (ovo je nas pocetni niz)

1. Ima 6 clanova, pa krecemo s korakom 6/2 = **3**
2. Krecemo s prvim elementom

12 55 18 **67** 88 12 <- u redu

3. Dalje

12 **55** 18 67 **88** 12 <- u redu

4. Dalje

-> 4a: 12 55 **18** 67 88 **12** <- potrebna zamjena
-> 4b: 12 55 **12** 67 88 **18** <- ne mozemo gledati unazad jer nema dosta clanova

---

Nas je niz sada 3-sortiran!

1. Krenimo dalje na korak 3/2 = **1**!
2. Pocetak

12 55 12 67 88 18 <- u redu

3. Dalje

-> 3a: 12 **55 12** 67 88 18 <- zamjena
-> 3b: 12 **12 55** 67 88 18 <- posto se dogodila zamjena, moramo gledati unazad
-> 3c: **12 12** 55 67 88 18 <- u redu

4. Dalje

12 12 **55 67** 88 18 <- u redu

5. Dalje

12 12 55 **67 88** 18 <- u redu

6. Dalje

-> 6a: 12 12 55 67 **88 18** <- potrebna je zamjena
-> 6b: 12 12 55 67 **18 88** <- posto se dogodila promjena idemo unazad
-> 6c: 12 12 55 **67 18** 88 <- potrebna je zamjena
-> 6d: 12 12 55 **18 67** 88 <- dogodila se zamjena pa gledamo jos unazad
-> 6e: 12 12 **55 18** 67 88 <- potebna je zamjena
-> 6f: 12 12 **18 55** 67 88 <- dogodila se zamjena gledamo jos unazad
-> 6g: 12 **12 18** 55 67 88 <- u redu, ne moramo vise gledati unazad

Posto smo u startu usporedjivali zadnja dva elementa, nas Sort je gotov i nas je niz sortiran!

Nadam se da ste uspjeli uloviti bit Shell Sort-a jer je stvarno tezak za objasnit, ali je krajnje jednostavan kad se uhvati bit. ;)

3.e

MERGE SORT

Merge Sort je u biti krajnje jednostavan. Ideja iza cijelog tog sorta jest da se niz brojeva dijeli po pola sve dok ne ostane samo po jedan element. Kad se to dogodi, usporede se elementi zadnjeg dijeljenja (posto ostanu samo 1+1 element) i onda se smjeste u novo polje. Tako se od najjednostavnijih djelica spoji jedan veliki poslozeni.

Najlakse ga je shvatiti konkretno kroz primjer (kao ostale nije bilo :)), pa krecemo...

Kako to izgleda u stvarnom nizu?

12 55 18 67 88 12 (ovo je nas pocetni niz)

1. Razbijamo pocetni niz na dva dijela, A1 i B1

    A1 -> 12 55 18
    B1 -> 67 88 12

    => 2A: Razbijamo A dio na jos manje dijelove A2 i A3

        A2 -> 12 55
        A3 -> 18

    => 3A: Razbijamo A2 dio na jos manje dijelove A4 i A5

        -> 3A.a: A4 -> 12, A5 -> 55
        -> 3A.b: Sada gledamo koji je od ta dva manji i njega stavljamo prvog,
        a veceg stavljamo drugog u nas A2 niz
        -> 3A.c: A2 -> 12 55

    => 4A: A3 je imao samo jedan clan pa se smatra sortiranim

    => 5A: Posto sada imamo dva niza (A2 i A3) koji su sortirani, mozemo i njih opet spojiti u A1 ali tako da budu sortirani. To se radi uzimanjem jednog po jednog clana iz oba i usporedjivanjem, te uvrstavanjem u A1.

        A1 -> 12 18 55

    => 2B: Razbijamo B dio na jos manje dijelove B2 i B3

        B2 -> 67 88
        B3 -> 12

    => 3B: Postupak je potpuno analogan dijelu A, tj. sad razbijamo B2 na jos manje dijelove B4 i B5

        -> 3B.a: B4 -> 67, B5 -> 88
        -> 3B.b: Posto su i B4 i B5 nizovi s jednim elelementom, uvrstavanje
        nazad u B2 prvo manji, pa veci dobit cemo sortirani niz od dva elementa
        -> 3B.c: B2 -> 67 88

    => 4B: B5 ne treba rastavljati na jednostavnije dijelove posto ima samo jedan element.

=> 5B: Ponovno, uzimanjem manjeg clana od dva koja se uzimaju usporedno iz, jednostavnijih i sortiranih, nizova dobit ce se veci sortirani niz.

B1 -> 12 67 88

Mala pauza: Dobili smo dva polja koja su sortirana. Iz njih ce se istim principom vaditi elementi i tako ce se dobiti na kraju niz koji je sortiran. Ako vam do sada nije jasan princip evo ga jos jednom, ali detaljnije (jer prije nije bilo moguce).

A -> 12 18 55
B -> 12 67 88

1. Uzimamo po prvi clan iz oba niza

   A -> **12** 18 55
   B -> **12** 67 88

2. Svejedno je koji uzmemo jer su jednaki pa cemo uzeti iz A i dodati u konacni niz:

   N -> 12

3. Uzimamo opet po prvi clan iz oba niza

   A -> **18** 55
   B -> **12** 67 88

4. Prvi clan niza B je manji, pa on ide u nas konacni niz:

   N -> 12 12

5. Uzimamo opet po prvi clan iz oba niza

   A -> **18** 55
   B -> **67** 88

6. Clan niza A je manji, ide u nas krajnji niz

   N -> 12 12 18

7. Prvi clan iz oba niza

   A -> **55**
   B -> **67** 88

8. Clan niza A je manji ide u nas krajnji niz

   N -> 12 12 18 55

9. Niz A je prazan, znaci samo dodajemo niz B

   N -> 12 12 18 55 67 88

GOTOVO!

4.

STOG PREKO POLJA I LISTA, PROCEDURALNO I OBJEKTNO

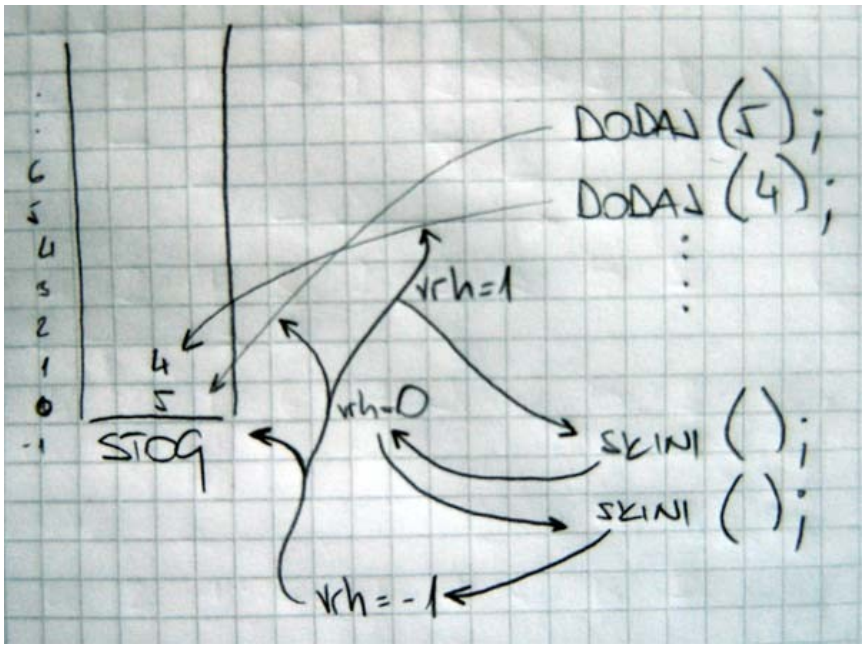
STOG PREKO POLJA

Sto se stogova tice, obradit cu ih preko vec napisanog koda zato sto ce vam to i trebati u MI i to vjerujem kako ce doci samo funkcije Dodaj() i Skini(). Sve ostalo su varijante na te funkcije i ne bi trebali biti problem.
Samo da nabrzaka opet spomenem ono sto vec i lusteri u ucionama znaju :) - stog radi po principu LIFO (Last In First Out) sto znaci da ono sto smo zadnje dodali prvo skidamo bla bla. :)

Sad na mesoooo... :)

Stog preko polja ostvaren je tako da se uzme neko polje i jednostavnim stavljanjem po indexa se u njega pucaju clanovi stoga koje zelimo imati. Uvijek treba imati na umu da se elementima u tom polju ne smije pristupati direktno vec samo preko funkcija Skini() i Dodaj() (i to zato da ne bi poremetili LIFO sustav). Sto bi to konkretno znacilo?
To bi konkretno znacilo da ako zelite naci najveci element na stogu i izbaciti ga nije dovoljno zavrtiti petlju i proci sve clanove nego treba cijelo polje "skinuti" na pomocno polje i kod vracanja vratiti sve clanove osim najveceg. Veselo, jel tako? ;)

Stog kao takav se sastoji od elemenata koje sadrzi i vrh-a koji kaze koji je element trenutno na vrhu. Doljnja slika (nacrckana nabrzaka) pokazuje kako se dodavanjem i skidanjem elemenata pomice vrh. Treba primjetiti kako se kod skidanja elemenata samo pomice vrh, a da se elementi ne brisu (stednja vremena). :)



DODAJ ()

Ajmo sada na kod koji nije tako tezak...

```
// Funkcija kao takva ima sljedeci prototip
     int dodaj (int stavka, int stog[], int n, int *vrh) {

// Iz njega vidimo kako se u funkciju prenosi stavka koju dodajemo
// u stog (ova petica i cetvorka na gornjem crtezu npr.), sam stog
// (zato sto s necimo moramo raditi), n kao maksimalan broj clanova
// koji smije biti na stogu, te vrh kao pointer jer se njegova promjena
// mora ocitovati i u glavnom programu. Kad bi se vrh predavao kao
// "nepointer" njegove promjena ne bi uzrokovala nista posto bi se
// lokalna varijabla izbrisala cim se zavrsi funkcija dodaj().

// Prvo treba provjeriti jesmo li dostigli maksimalan broj clanova
// koje stog moze primiti. Ako jesmo vratit cemo 0 u glavni program
          if (*vrh >= n-1) return 0;

// Nakon toga se dogadja povecanje vrha (koje mozete vidjeti i na
// crtezu). Stavka (element) se dodaje na stog tek nakon povecanja
// vrha. Zasto je to tako? Zato sto su kolege sa ZPR-a zamislile stog
// na nacin da im je vrh zapravo index zadnjeg dodanog clana. To znaci
// da clan koji je dodan kao 5. po redu ce imati index 4 (posto se
// krece od 0) i vrh stoga ce biti 4.
// Stog se takodjer mogao ostvariti tako da vrh bude index
// (nepostojeceg) clana vise od zadnjeg dodanog. Takvu implementaciju
// mozete vidjeti na prvoj stranici ove teme.
          (*vrh)++;
          stog[*vrh] = stavka;

// Sve je uspjesno obavljeno, vracamo 1
          return 1;

     }
```

To bi bilo sve o proceduralnoj funkciji dodaj, ajmo sada istu tu funkciju pogledati objektno...

```
// Razred s kojim se radi zove se "StogP" i ima privatne varijable
// _MAXSTOG, _vrh i *_stog. _MAXSTOG sadrzi brojcanu vrijednost
// koja kaze koliko se elemenata smije smjestiti na stog. _vrh je vrh
// stoga kao i prije, a *_stog je upravo nase polje koje predstavlja
// stog.
// Metoda (posto je izvan razreda) izgleda ovako:
     void StogP::Dodaj (int stavka) {

// i prima samo stavku s kojom mora raditi. Ovo je prednost OOP-a!
// Mogucnost da se radi sa stogom i vrhom bez ikakvih dodatnih slanja
// istog u funkciju, zato sto je ovo funkciji (bolje reci metodi)
// dostupan i stog i vrh i _MAXSTOG preko privatnih varijabli.

// Prvo provjerimo jesmo li presli granicu stoga. Ako jesmo, bacaj
// van!
          if (_vrh >= _MAXSTOG - 1) throw "Stog je pun!";
```

```
// Nakon toga se po identicnom principu kao prije povecava vrh, a
// nakon toga stavlja stavka na to mjesto. Jos jednom primjetite
// kako vrh vise nije pointer posto je smjesten u privatnu varijablu

          _vrh++;
          _stog [_vrh] = stavka;

     }
```

SKINI ()

Skidanje sa stoga ostvarenog poljem zapravo je lakse od dodavanja zato sto sve sto se dogadja je samo pomicanje vrha "nadolje". Nista vise, ali ni manje...

```
// Prototip funkcije glasi:

     int skini (int *stavka, int Stog[], int *vrh) {

// i prima stavku (kao pointer), stog s kojim mora raditi i vrh
// opet kao pointer. Vrh se prima kao pointer iz istog razloga
// iz kojeg se primao u funkciji Dodaj(). Ali, zasto stavka
// dolazi kao pointer?
// Stavka dolazi kao pointer zato da bi mogli zapamtiti element
// koji skidamo u neku varijablu u glavnom programu zato sto
// ovakva implementacija stoga vraca 0 ili 1 u ovisnosti o uspjesnosti
// skidanja sa stoga. Lagano, ne? :)

// Prvo provjerimo ima li uopce elemenata na stogu. Ako je vrh manji
// 0 (tj. ako iznosi -1) ne mozemo nista skidati.

          if (*vrh < 0) return 0;

// Nakon toga u stavku spremamo vrijednost koju skidamo i smanjujemo
// vrh za jedan. Opet primjetite kako se stavka s vrha nije brisala vec
// se samo vrh pomaknuo za jedan dolje.

          *stavka = Stog[*vrh];
          (*vrh)--;
          return 1; }
```

OOP varijanta skidanja sa stoga potpuno je analogna proceduralnoj i vrijede sva pravila koju su vrijedila i za objektni dodaj().

```
// Prototip funkcije:

     void StogP::Skini (int& stavka) {

// koji prima referencu na stavku je takav zato sto u glavnom programu
// zelimo dobiti stavku koja bi inace bila izgubljena.

//Klasicna provjera je li vrh manji od 0

          if (_vrh < 0) throw "Stog je prazan!";

// Spremanje upravo skinute stavke i pomicanje vrha za 1 nadolje.
// I opet vrh nije pointer zato sto je privatna varijabla.

          stavka = _stog [_vrh];
          _vrh--; }
```

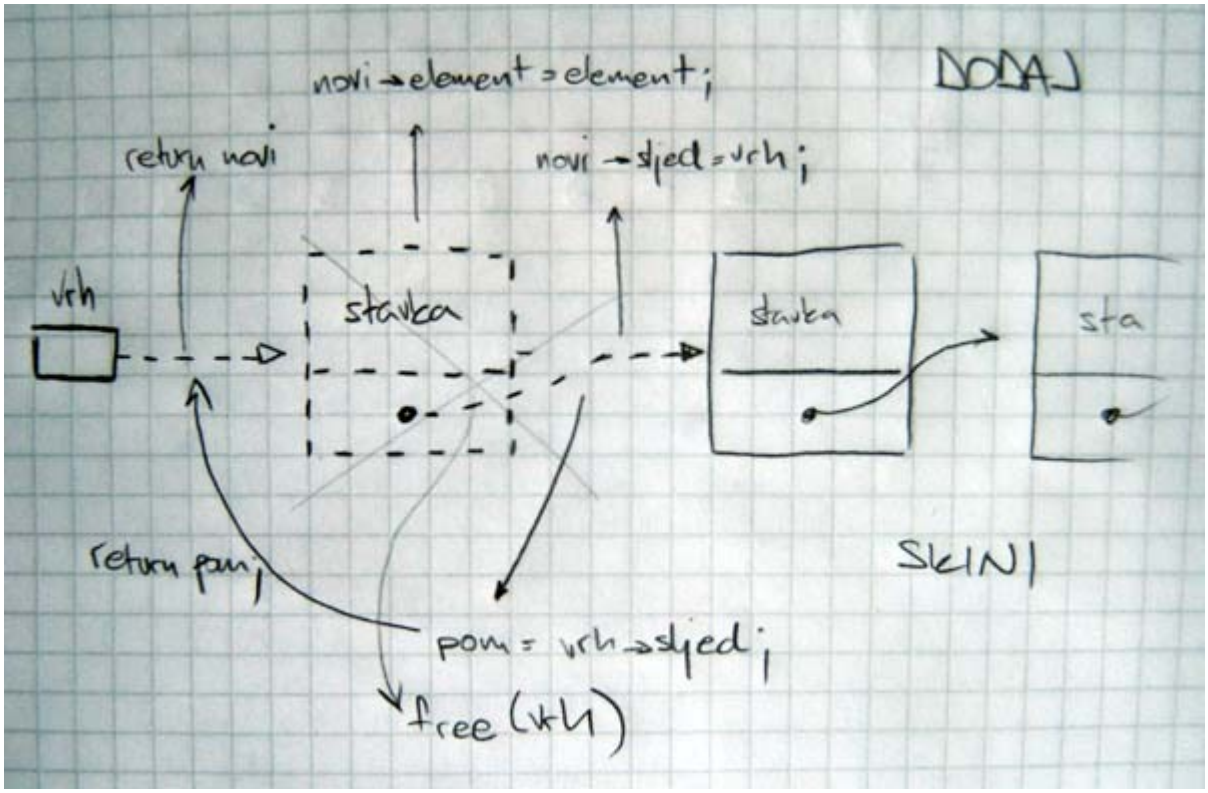
To je sve sto se tice stoga ostvarenim preko polja. Vise od ovoga ne bi smjelo doc (osim neke funkcije tipa trazenja najveceg elementa, ali kao sto rekoh, to su sve varijacije na temu).

STOG PREKO LISTE

Problem kod stoga preko liste je taj sto mi listu jos dotaknuli nismo. Opcenito izgleda kako se ove godine (bolonjske gospodnje :)) sve radi jako jako zbrda zdola, pa bi ovo bilo puno lakse preko duplih pointera, ali sto je tu je. ;)

Uglavnom, prica je ovakva. Liste kao takve su zapravo niz elemenata koji se sastoje od dvije stvari - neka vrijednost + pokazivac na sljedeci element.
Otprilike kao vlak. Svaki vagon ima svoj sadrzaj i pristekan je (pokazivacem) na sljedeci. Ista je prica sa listom, samo sto mi ne radimo listu, pa da mozemo nasu kompoziciju vlakova mijenjati kako nam odogovara. Mi radimo stog, sto znaci da opet moramo raditi po principu LIFO.

ZPR-ovski nacin ostvarivanja (mozda ne najlaksi, ali sigurni bodovi) je taj da se svaki novi element dodaje na pocetak (a ne na kraj) liste. Svaka lista mora imati svoju glavu i rep. Glava se sastoji samo od poveznice na neki element. Kao kad bi lokomotiva bila samo spojnica koja visi u zraku, ali ipak odrzava vlak. E, pa mi nas novonastali element u stogu upravo vezemo na tu glavu, te spojnica naseg novo elementa vezemo na stari element. Evo i slike koja to ilustrira (+ tocno pokazuje koja naredba sto radi):



DODAJ ()

Realno se vise sto o stogu preko liste nema reci, pa prelazim na kod...

```
// Kao sto sam vec rekao, dvostrukim pointerom bi se izbjegle
// neke neugodnosti ovdje, ali nema veze, moze se i s njima.
// Nas prototip izgleda:
     atom *dodaj (atom *vrh, tip element) {

// atom kaze da ce nasa funkcija vracati nesto tipa atom.
// Sto je atom? atom je upravo jedan vagon u vlaku. Tip
// podatka koji ima element + poveznicu. I to je sve! :)
// Takodjer, funkciji prosljedjujemo nas vrh, tj. onu glavu
// o kojoj sam pricao sad malo prije gore, te naravno element
// koji treba dodati u taj nas vagoncic.

// U nasoj funkciji trebat cemo izgraditi prvo vagoncic
// sljedecom naredbom:
          atom *novi;

// Potom se vrsi provjera ima li uopce mjesta u memoriji
// za nas novonastali element (vagoncic). Ako nema, funkcija
// preskace dodavanje vrijednosti elementu *novi i on se vraca
// u funkciju kao prazan (NULL)
          if ((novi = (atom *) malloc(sizeof(atom))) != NULL) {

// Nakon toga vrse se dvije stvari - *novi dobiva svoju
// vrijednost (novi->element = element;) i povezujemo
// sljedeci element na nas novi. vrh je do sada pokazivao
// na prvi element u stogu. Sada ce tu ulogu preuzeti ovaj
// novi element (pogledajte si i sliku). Jedino sto jos ostaje
// je povezati vrh (tj. glavu) tako da pokazuje na nas
// novododani element, no to se ovdje vrsi u glavnom programu.
               novi->element = element;
               novi->sljed = vrh;

          }

// Ovdje se vraca taj novi vagoncic koji ce u glavnom programu
// povezati na glavu.
          return novi;

     }
```

To je to sto se tice proceduralnog stoga, ajmo sad na objektni. Razlika kod ova dva ce biti to sto opet mozemo svemu pristupiti, pa ce neke stvari biti jednostavnije, a osim toga vidjet cemo tocno ovaj zadnji korak (povezivanje glave s novonastalim elementom) koji se u proceduralnom nije vidio.

```
// Ime razreda je StogL i metoda kao argument prima samo
// stavku:
     void StogL::Dodaj (int stavka) {

// Opet na pocetku stvarano novi vagoncic
          atom *novi;

// I u OOP-u ga moramo napraviti kao novi objekt razreda
```

```
// atom
          novi = new atom;

// Provjera ima li dosta memorije
          if (novi == NULL)throw "Nema memorije!";

// Ovo je mjesto gdje se cuda dogadjaju. :) Novonastalom
// elementu pridruzujemo stavku koju smo doveli preko
// argumenata. Nakon toga pokazivac (poveznica) od novonastalog
// pocinje pokazivati na prijasnji prvi element u listi, a
// vrh pocinje pokazivati na novonastali element.
// Opet, crtezom je mozda cak i lakse za shvatiti.
          novi->element = stavka;
          novi->sljed = _vrh;
          _vrh = novi;

     }
```

To je sve sto se ima za reci o dodavanju novog clana u listu.

SKINI ()

Skidanje sa stoga ostvarenog listom opet se vrsi (kao i prije) pomicanjem vrha stoga. Ovaj puta, postupak je vrlo slican dodavanju, samo sto cemo poveznice medju vagoncicima (elementima) mijenjati u suprotnom smjeru, tj. prvo ce se poveznica sa elementom koji je prvi nakon brisanja spremiti, pa ce se unistiti poveznica glave sa elementom koji se brise i onda ce se povezati vrh i prvi element koji se ne brise (opet, ako nije jasno, slika dosta jasnije pokazuje).

Ajmo na kod... (WOOOOOOOHOOOOOOOOO, sori ljudi lud sam vec malo :))

```
// Prototip funkcije:
     atom *skini (atom *vrh, tip *element) {

// opet ima isti princip kao i kod dodavanja.

// Dodajemo pomocni vagoncic koji ce samo privremeno odrzavati
// vezu izmedju prvog sljedeceg elementa i glave u trenutku
// puknuca veze izmedju glave i elementa koji se brise. To si
// zamislite kao supermana koji odrzava taj vlak i onda ih
// fino pospoji (a naravno, brisuci vagoncic je nestao u
// velikoj eksploziji :)).
          atom *pom;

// Provjera je li stog prazan.
          if (vrh == NULL) return NULL;

// Preuzimanje vrijednosti elementa kojeg brisem (jer se to
// obicno trazi u programima).
          *element = vrh->element;
```

```
// E sad Superman dolazi u igru. :)
// Pom uzme poveznicu koja povezuje element koji se brise i
// glavu. Free unisti tu poveznicu, a u glavnom programu
// nakon returna, ta poveznica se poveze s prvim sljedecim
// elementom. Kod OOP nacina ce se to vidjeti jos bolje.

          pom = vrh->sljed;
          free (vrh);
          return pom;

     }
```

Realizacija preko je i opet zapravo laksa od proceduralne jer nema toliko bas bakcanja s prototipom funkcije i upravo slijedi:

```
// Razred je opet StogL,a metoda Skini. Referenca na stavku
// sluzi glavnom programu da sacuva vrijednost stavke elementa
// koji skida sa stoga.

     void StogL::Skini (int& stavka) {

// Opet nam treba pomocni koji ce odrzavati kompoziciju vlaka.

          atom *pom;

// Naravno, provjera je li stog prazan...

          if (!_vrh) throw "Stog je prazan!";

// ... i upisivanje stavke u referencu.

          stavka = _vrh->element;

// A evo i cuda koje se dogadja :). Identicno kao kod
// proceduralnog, pom prima poveznicu glave na sljedeci
// element. Nakon toga se poveznica s tim elementom brise
// (to je kao da brisemo taj element), a glava dobiva novu
// poveznicu na prvi element iza onog obrisanog. Naravno,
// konzultirajte sliku za detalje.

          pom = _vrh->sljed;
          free (_vrh);
          _vrh = pom;

     }
```

5.

VEZANE LISTE

UVODIC

Vezane liste same po sebi nisu neka velika pamet, ali u vezanim listama dolazi do jednog potpuno novog pojma koji jako zna zbuniti ljude u pocetku, no s kojim je puno lakse raditi, a to je dupli pointer.
Ne bih zelio ulaziti previse u bit tog duplog pointera zato sto bih mozda dotakao nesto sto ne znam (i to vrlo lako) i krivo napisao.
Bit duplog pointera je ovakva: Kad udjemo u neku funkciju, varijable dane preko argumenata funkcije stvaraju lokalne verzije tih varijabli sebi u petlji. Da bismo izbjegli takav problem kod obicnih varijabli uveli smo pointere koji su to jako lijepo rjesavali, no pojavim vezanih lista javlja se problem, a to je problem da ti pointeri vise nisu dovoljni i javljaju se lokalne verzije nasih elemnata u vezanoj listi.
Da bismo to izbjegli predajemo referencu na taj element i gledamo ga kao pointer na pointer ilitiga pokazivac na to sto smo predali. Zapravo smo opet izbjegli da nas stvaranje lokalnih kopija varijabli sprijeci od rada s tim varijablama globalno. Nista vise i nista manje.

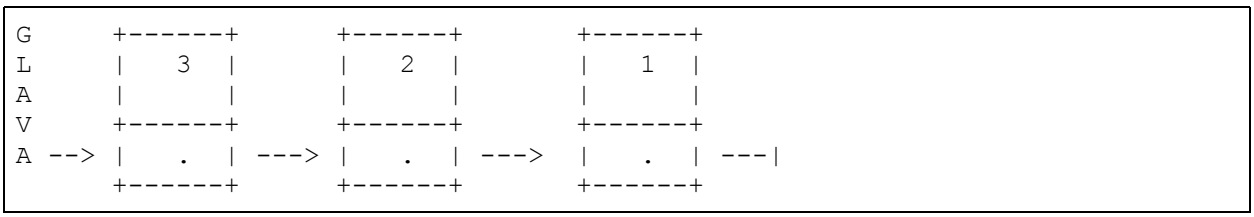
Ovo sam samo htio na pocetku rascistiti jer to zapravo i nije neka strasna mudrolija, samo mali trik da se i sa ovakvim tipom podataka moze fino funkcionirati. ;)

OPCENITO

Sto opcenito reci o listama?
Liste mogu biti vise vrsta, no ona lista koju cemo mi obradjivati (i koja ce se 90% naci na zavrsnom ispitu) je jednostruko povezana lista.
Listu mozemo vrlo lako usporediti sa poljem. Polje kao takvo rezervira komad memorije koji slijedi jedan iza drugog i tako moze imati vise podataka. Lista je vrlo slicna tome s jednom osnovnom razlikom. Podaci su u memoriji rasuti.
Normalno, postavlja se pitanje: "Pa kako onda podatak u listi zna koji je po redu itd?" Vrlo lako - tu dolaze u igru pokazivaci.
Lista se (kao i polje) sastoji od vise podataka koji su povezani. Svaki clan te liste mora imati najmanje dva dijela: neku svoju vrijednost (ili vise njih), te pokazivac na sljedeci clan. E, u tom pokazivacu je tajna.
Posto su podaci rasuti po memoriji, svaki clan te liste pokazivat ce tocno u memoriji na podatak koji ga nasljedjuje i to je tajna funkcioniranja lista.

Primjer:
Imamo 3 podatke (3, 2 i 1) od kojih je 3 pocetni, a 1 zavrsni podatak. Organizirani su u listu i pokazuju jedan na drugoga. To bi izgledalo otprilike ovako:

Kod:



Svaki clan ove liste prikazan je sa jednim podatkom (brojkama 3, 2 i 1), te pokazivacem na sljedeci clan. Taj pokazivac kaze tocno gdje u memoriji se nalazi sljedeci clan (clan s trojkom kaze gdje se nalazi clan s dvojkom, clan s dvojkom kaze gdje se nalazi clan s jedinicom itd.)

Ostaje jos samo misterij glave. Glava kao takva alocira se u glavnom porgramu i ona zapravo ne sadrzi nikakvu vrijednost, vec samo pokazuje dalje na svoje clanove. Ona sluzi za organizaciju podataka, da bi se znalo od kuda se krece.

O listama se nema vise sto previse reci. Krenimo na prakticni dio.

IZGLED CLANA LISTE

Kao sto sam rekao, clan nase liste mora sadrzavati bar 2 dijela, pa cemo ga tako i definirati. Clanove liste zvat cemo elementima i sadrzavati ce brojcanu vrijednost + pokazivac na sljedeci element.
To onda izgleda ovako:

Kod:

```
struct element {

    int br;
    struct element *sljed;

};
```

Primjetite kako pokazivac na sljedeci element mora biti istog tipa kao i sam element (clan) te liste. To je zapravo vrlo logicno, pa on upravo pokazuje na drugi clan liste koji je jednakog tipa kao i trenutni clan, zar ne? ;)

DODAVANJE NOVOG ELEMENTA U LISTU

U okviru ovog malog tutoriala dodavati cemo clanove uvijek na prvo mjesto. Za neke druge varijante (dodavanje na kraj npr.) konzultirajte primjere sa fer.hr-a jer ce biti jasni ako ovo shvatite.

Dodavanje clanova u listu uvijek se sastoji od 3 koraka (koja mozete zapamtiti kao i ime ovog kolegija - ASP :)):

1. **A**lociraj i spremi
2. Povezi sa **s**ljedecim
3. Povezi sa **p**rethodnim

Ovo zapravo i zvuci vrlo logicno. Zamislimo da imamo listu koja se sastoji od clanova 2 i 1, te zelimo dodati 3. Ta trojka ce biti ispred dvojke.
Dodavanje ce se dogadjati po nasem **ASP** principu.

1. Alociraj i spremi
   Kod:
   ```
   struct element *novi = NULL;
       novi = (struct element *) malloc (sizeof(struct element));
       novi->br = dodbr;
   ```
   Ovdje stvaramo mjesto za novi element tako da ga stvorimo te mu priredimo mjesto u memoriji. Nakon toga samo jos pridruzimo njegovoj vrijednosti neki "dodbr", tj. broj koji mu zelimo pridruziti.

2. Povezi sa sljedeci
   Kod:
   ```
   novi->sljed = glava;
   ```
   Glava ce u nasem slucaju biti onaj vrh liste koji sam spomenuo. On pokazuje na brojku 2 (posto je 2 prvi element nakon glave). Najpametnije ce biti da mi s pokazivacem naseg novog elementa takodjer pokazemo na tu dvojku kako bi pokazali na sljedeceg kojeg zelimo.

3. Povezi sa prethodnim
Kod:

```
glava = novi;
```

Posto smo pokazali novom clanu sto mu slijedi, treba mu pokazati i sto mu prethodi, a prethodi mu upravo glava. Pa recimo glavi neka pokazuje na novi i sve smo sredili.

Doslovce gledano, ugurali smo trojku izmedju glave i dvojke. Pokusat cu to pokazati i slikicama:

1. Alociranje
Kod:

```
+---+
| 3 |
+---+
```

2. Povezivanje sa sljedecim

Kod:

```
G   2   1           G  3->2   1
  ^         ====>
  3
```

3. Povezivanje s prethodnim

Kod:

```
G  3->2   1  ====>   G->3->2   1
```

I tako se nasa trojka ugurala medju ovo dvoje.

I ne, jos nismo gotovi. :)
Ovo sto sam napisao (programski kod) nikad ne bi funkcionirali jer se ne koristi duplim pointerima (cija je potreba objasnjena na pocetku). Evo sada cijele funkcije kako bi trebala izgledati sa 3 poziva na istu koja ce stvoriti listu 3 2 1.

Kod:

```
void Dodaj (struct element **glava, int dodbr) {

    struct element *novi;

    novi = (struct element *) malloc (sizeof(struct element));
    novi->br = dodbr;
    novi->sljed = *glava;
    *glava = novi;

}
```

A poziva se sa (iz glavnog programa):

Kod:

```
    struct element *glava;

    Dodaj (&glava, 1);
    Dodaj (&glava, 2);
    Dodaj (&glava, 3);
```

Ne smije se zaboraviti da se prvo u glavnom programu stvara glava, a tek onda ide dalje. ;)

DULJINA NASE LISTE

Posto smo uspjeli stvoriti listu, bilo bi lijepo kad bi mogli saznati koliko imamo clanova. To cemo uciniti tako da cemo putovati po listi, te brojati clanove sve dok ne stignemo na prazan clan (sto oznacuje kraj).

Kod:

```
int Duljina (struct element *glava) {

    int duljina = 0;

    while (glava != NULL) {
        duljina++;
        glava = glava->sljed;
    }

    return duljina;
}
```

Sto se ovdje dogadja?
Nasa while petlja izvodi se sve dok trenutni clan ne postane NULL, tj. oznaci kraj. Dok god se to ne dogodi povecavat ce se brojac, te cemo se pomicati clan po clan tako da stalno skacemo na sljedeci (glava->sljed).
Ona prica s duplim pointerima ima i svojih dobrih stvari. Da smo inace ovako pridruzivali (glava = nesto) pointere unistili bismo vrijednost liste u glavnom programu, ali posto ovdje trebamo duple pointere da bismo mogli mijenjati listu, kod ispisa to mozemo ispustiti i u tom slucaju nema potrebe za stvaranjem pomocnog pokazivaca koji ce putovati po listi (i toboze sacuvati njen sadrzaj i glavu).

ISPIS LISTE

Ispis liste vrlo je slican prebrojavanju njenih clanova, samo sto cemo njega izvesti s pomocnim pointerom, cisto radi potpunosti.

Kod:

```
void Ispisi (struct element *glava) {

    struct element *pom;
    pom = glava;

    while (pom != NULL) {

        printf ("%d ", pom->br);
        pom = pom->sljed;

    }

}
```

While petlja ista je kao prije. Novost je samo prebacivanje uloge putnika po listi na varijablu pom, umjesto same glave.

VADJENJE CLANA

Vadjenje clana (po principu dodavanja, tj. odmah iza glave) necu opisivati jer je to vrlo pojednostavljena verzija naseg ASP pristupa, tj. potrebno je samo sacuvati vrijednost koja se nalazi u tom prvom elementu i povezati glavu sa sljedecim elementom iza onog kojeg vadimo (doslovce - *glava = pom->sljed, s time da je u pom spremljeno ovako: pom = glava->sljed).

Treba primjetiti sljedece - stavljanje ovako clanova na vrh s funkcijom Dodaj i vadjenje clanova na goreopisani nacin zapravo nas dovodi do neceg sto smo vec susreli - STOG! Tko kaze da je stog preko lista tezak? ;)

ZAKLJUCAK

Toliko o listama. Za svo sire znanje pogledajte gotove primjere sa fer.hr-a, sada ih vise ne bi trebalo biti toliko tesko pratiti. :)

Naravno, za kraj i cijeli program po kojem mozete mijenjati sto hocete:

Kod:

```
#include <stdio.h>
#include <stdlib.h>

struct element {

    int br;
    struct element *sljed;

};

void Dodaj (struct element **glava, int dodbr) {

    struct element *novi;

    novi = (struct element *) malloc (sizeof(struct element));
    novi->br = dodbr;
    novi->sljed = *glava;
```

```
    *glava = novi;

}

int Duljina (struct element *glava) {

    int duljina = 0;

    while (glava != NULL) {
        duljina++;
        glava = glava->sljed;
    }

    return duljina;
}

void Ispisi (struct element *glava) {

    struct element *pom;
    pom = glava;

    while (pom != NULL) {

        printf ("%d ", pom->br);
        pom = pom->sljed;

    }

}


int main()
{

    struct element *glava = NULL;

    Dodaj (&glava, 1);
    Dodaj (&glava, 2);
    Dodaj (&glava, 3);

    printf ("Duljina nase liste je: %d.\n", Duljina (glava));
    printf ("Duljina nase liste je: %d.\n", Duljina (glava));

    printf ("Nasa lista izgleda ovako: ");
    Ispisi (glava);

    return 0;

}
```

6.

BINARNA STABLA

(Mala napomenica: Mozete slobodno raditi po Cupicevom objasnjenju stabala, ali odmah da kazem kako se ja tamo nisam snasao ni malo jer se on u potpunosti oslanja na duple pointere, a s njima sam malo na Vi, tako da ja to izbjegavam i opcenito mislim da je ovaj tutorial vise user-friendly od Cupicevog. No, izaberite sami sto vam se vise svidja, na kraju krajeva bitno je da vi skupite bodove i prodjete/razvalite predmet. :))

OPCENITO

Opcenito, koja je fora sa stablima?
Fora sa stablima je takva da ona u biti vise lice na korijen nego na stablo. Korijen zato sto pocinjemo od jednog elementa koji se grana na vise elemenata, od kojih se svaki grana na jos vise elemenata.
Primjer:

Kod:

```
                              ELEM
                             /    \
                         ELEM      ELEM
                        /    \
                    ELEM     ELEM
                  (.....)  (.....)
itd.
```

Znaci, ovo je neko kratko stablo koje se grana sve vise i vise. Sada vjerojatno shvacate zasto sam rekao da to vise lici na korijen nego na stablo.

BINARNO STABLO

Stablo koje je nama zanimljivo i koje ce se koristiti je binarno stablo. Binarno stablo je takvo stablo koje ima uvijek samo dvije grane koje se nastavljaju iz njega (kao i nas primjer na slici).

Takvo binarno stablo definirano je ovako:

Kod:

```
struct cvor {

    int broj;
    struct cvor *lijevi;
    struct cvor *desni;

};
```

Analizirajmo taj "struct cvor". Taj struct cvor ce biti upravo struktura svakog clana naseg stabla (u gornjoj slici - ELEM). Svaki ELEM je tipa struct cvor i sadrzi barem 3 stvari -> pokazivac na lijevu granu (koja slijedi), pokazivac na desnu granu (koja slijedi), te neki podatak. Naravno, podataka unutra moze biti koliko god korisnik hoce ili treba, ali struktura (2 + 1) mora ostati (s time da ovih 2 prikazuje grane na koje se nastavlja nase stablo).

BINARNO SORTIRANO STABLO

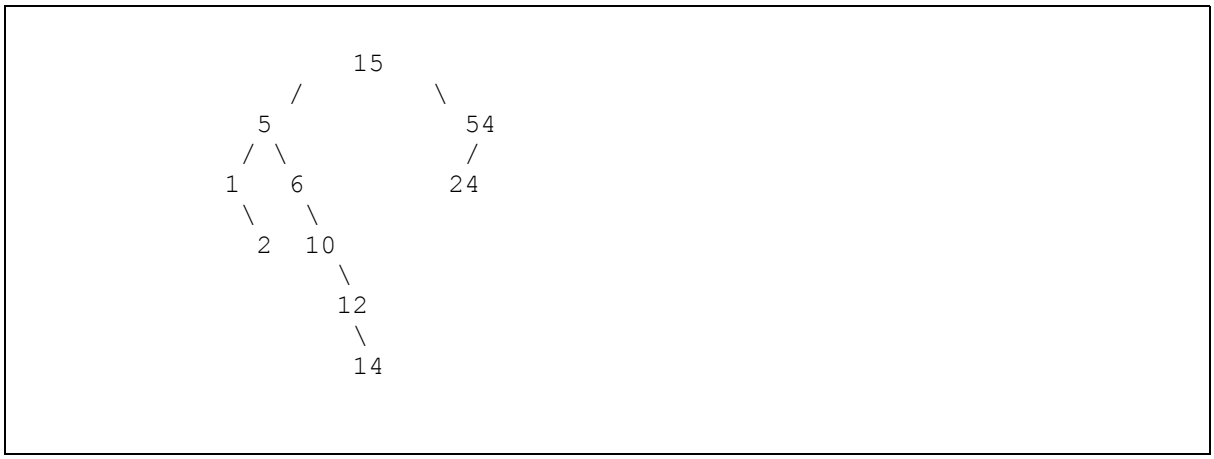
Za nas primjer uzeti cu binarno sortirano stablo jer su ona najzanimljivija i najveca je mogucnost da ce doci na zavrsnom.

Samo kratka digresija prije nego nastavim... Spomenuo sam binarno **sortirano** stablo. Sto to znaci? To znaci da je to stablo koje je uvijek sortirano po nekom elementu za razliku od obicnog binarnog stabla koje prima elemente bilo kako.

NASE BINARNO SORTIRANO STABLO

Binarno sortirano stablo radi po principu da uvijek element koji treba doci na svoje mjesto mora biti s lijeve strane elemenata vecih od sebe, a desno od elemenata manjih od sebe. Zbunjeni? Evo skice, pa ce biti jasnije:

Kod:



Elementi dodavani u ovo stablo isli su redom ovako: 15, 5, 6, 10, 54, 24, 12, 1, 2, 14.

Sto nam kaze gornja skica?

Krecemo sa 15, 15 je nas Alpha (ne maco muskarac iz 6. grupe, vec pocetak :)). Nakon dodavanja tog prvog clana, ide clan 5. Njega dodajemo na lijevu granu clana 15 jer je manji od njega. Nakon toga ide clan 6. On ide na desnu granu clana 5 jer je veci od njega. Nakon toga ide 10, on ide na desnu granu clana 6 jer je veci od njega. Nakon toga ide 54 i on ide na desnu granu clana 15 jer je veci od njega. Nakon toga ide 24, on ide na lijevu stranu od 54 jer je manji od njega.

Do sada ste stvarno mogli shvatiti bit price. Ali evo nesto zanimljivo. 12 ce se nasaditi na desnu granu 10, a ne na lijevu granu od 24. Zasto je to tako? Jednostavno, zato sto je 12 manji od 15, tako da nikako ne moze doci u desnu grane clana 15.

Ostaju jos 1 i 2 koji su jasni, te 14 koji opet zbog istog razloga kao i 12 ide s desne strane 12, ali lijeve strane od 15.

Eto, to je nase binarno stablo koje cemo koristiti u nasem programceku. Krenimo sa funkcijama!

FUNKCIJE ZA NASE BINARNO STABLO

a) NoviCvor

NoviCvor je funkcija koja jednostavno stvara novi element u nasem stablu tocno na vrijeme kada je to potrebno, odnosno kada je pozvana. U njoj se zapravo dogadja malo stvari, a to je: alociranje memorije za novi clan, upisivanje vrijednosti koju zelimo da novi element ima, te pridruzivanje NULL-a objema granama.
NULL pridruzujemo objema granama zato sto nakon novog elementa (koji je tek upravo umetnut) ocito nema nasljednih elemenata (grana) tako da one moraju biti NULL.

Kod:

```
struct cvor* NoviCvor (int novibroj) {

    struct cvor *cvor;

    cvor = (struct cvor*) malloc (sizeof(struct cvor));
    cvor->broj = novibroj;
    cvor->lijevi = NULL;
    cvor->desni = NULL;

    return cvor;

}
```

b) Umetni

Funkcija Umetni radi rekurzivno i to ovako (uz citanje ovoga pametno bi bilo gledati usputno kod koji se nalazi odmah tu ispod ovog pasusa)...

Samo da prvo rascistim nesto, rekurzivno se ne radi zato da bi zakomplicirao, vec zato sto je tako puno lakse putovati kroz stablo. Rekurzivno stalno ulazimo u tu istu funkciju, ali joj dajemo da obradjuje neki drugi clan (tj. ili lijevu ili desnu granu trenutnog clana).

Funkcija prvo provjerava je li u trenutnom koraku dosla do NULL clana (koji oznacava prazno mjesto bas tamo gdje hocemo). Kad se to dogodi, mi brzo ubacimo ovdje novi clan koristeci funkciju NoviCvor (1).

Ako jos uvijek nismo dosli do zadovoljavajuceg mjesta (po principu binarnog sortiranog stabla) bitno je da nastavimo traziti usporedjujuci broj koji zelimo dodati s trenutnim brojem u cvoru u kojem jesmo. Ako je broj koji zelimo umetnuti manji od trenutnog, treba krenuti u lijevu granu (2), a u suprotnom treba krenuti u desnu granu (3).

Kod:

```
struct cvor *Umetni (struct cvor *cvor, int broj) {

    if (cvor == NULL) {
        return (NoviCvor(broj));        // 1

    } else {
        if (broj <= cvor->broj) {
            cvor->lijevi = Umetni(cvor->lijevi, broj);  // 2

        } else {
            cvor->desni = Umetni(cvor->desni, broj);    // 3

        }
        return cvor;

    }
}
```

Radi potpunosti dodat cu i komad iz glavnog programa s kojim se upisuje (koristeci ovu gornju funkciju) tocno nase stablo iz skice gore.

Kod:

```
int i[10] = {15,5,6,10,54,24,12,1,2,14};
    int j;
    struct cvor *cvor=NULL;

    for (j = 0; j < 10; j++) {

        cvor = Umetni (cvor, i[j]);

    }
```

c) Trazi

Trazi je jedna vrlo cesta funkcija koja bi ekstremno lako mogla doci u zavrsni, a zapravo je vrlo lagana. Vraca 1 ako je nasla vrijednost, te 0 ako nije.

Bit je ovakva - posto je nase stablo binarno sortirano ne moramo prolaziti kroz cijelo stablo, vec samo slijediti onaj put koji je obiljezen samim obiljezjem tog stabla. Konkretno, nase sortirano stablo obiljezeno (sortirano) je koristeci vrijednosti koje se nalaze u cvorovima (elementima).

Kako ce nam to pomoci? Lako!

Sve sto je potrebno jest da usporedjujemo vrijednost koju trazimo sa vrijednosti u trenutnom clanu. Ako smo dosli do clana koji je NULL (tj. usli u praznu granu, koja vise nema clana) (1) znaci da nismo uspjeli naci trazeni clan i treba vratiti nulu. Ako se, naprotiv, trazeni element podudario s trenutnim elementom, SUPER, nasli smo clan i vracamo 1 (2).
Medjutim, ono sto se najcesce dogadja je to da treba putovati dublje u stablo. To se radi tako da se rekurzivno opet udje u stablo i to u granu koja nam odgovara. Sjetite se kako je to islo. Ako je trazeni broj manji od onog koji je u cvoru u kojem smo trenutno, potrebno je otici u lijevu granu (gdje su manji brojevi) (3) i obratno (4).

Kod:

```
int Trazi (struct cvor *cvor, int trazbr) {

    if (cvor == NULL) {
        return 0;                // 1

    } else {
        if (cvor->broj == trazbr) {
            return 1;            // 2

        } else {
            if (trazbr < cvor->broj) {
                return (Trazi(cvor->lijevi, trazbr)); // 3

            } else {
                return (Trazi(cvor->desni, trazbr));  // 4

            }
        }
    }
}
```

I opet radi potpunosti, evo dijela programa koji poziva ovaj potprogram, te za nekoliko vrijednosti ispise postoje li ili ne (u zagradama stoji ocekivana vrijednost).

Kod:

```
printf ("10 -> %d (1), 13 -> %d (0), 54 -> %d (1), 72 -> %d (0).\n",
        Trazi(cvor,10), Trazi(cvor,13), Trazi(cvor,54), Trazi(cvor,72));
```

d) MinClan

MinClan je funkcija koja vraca najmanji clan iz naseg binarnog sortiranog stabla.
Do sada ste mogli shvatiti kako je najmanji clan upravo onaj koji je najdoljni i najlijeviji. Tu cinjenicu iskoristit cemo ovdje kako bi uspjeli dobiti najmanji clan u nasem stablu.

Najlakse ostvarenje toga je da putujemo po nasem stablu samo po lijevoj grani sve dok ne dodjemo do NULL clana (jer ovdje grana prestaje) i svaki puta biljezimo sadrzaj trenutnog clana. Vracanjem vrijednosti pom dobit cemo upravo najmanji clan.

Kod:

```
int MinClan (struct cvor *cvor) {

    int pom;

    pom = cvor->broj;

    while (cvor->lijevi != NULL) {
        cvor = cvor->lijevi;
        pom = cvor->broj;

    }

    return pom;

}
```

Zbog potpunosti, dio iz glavnog programa.

Kod:

```
printf ("Najmanja vrijednost u nasem stablu je: %d (2).\n",
MinClan(cvor));
```

ZAKLJUCAK

Prica o stablima ovdje nije zavrsena. Ima se tu jos dosta dodati, ali je bila bit da shvatite sto se tocno krije iza tog misterioznog stabla i da shvatite kako to stvarno nije neki bauk, vec samo mali vise vizualan tip podatka (koji se najlakse prati crtezom).
Ostaje npr. objektno orijentirano rjesenje, pa stablo u datoteci, pa jos razne funkcije itd., ali za sve to postoji Google, mresetar.tk itd. itd.

I normalno, evo cijelog programa u jednom velikom kodu (naravno, detaljno komentiranom):

Kod:

```
#include <stdio.h>

// Osnovni izgled cvora u binarnom stablu.
// Sastoji se od podatka te od dva pokazivaca. Ta dva pokazivaca pokazuju
na
// lijevi i desni ogranak trenutnog cvora.

struct cvor {

    int broj;
    struct cvor *lijevi;
    struct cvor *desni;

};

// Funkcija NoviCvor ima zadatak stvoriti novi clan kod upisivanja u
binarno
// stablo. Taj novi cvor poprimiti ce vrijednost koja mu se zada, te
svoje
// grane proglasiti praznima (NULL), sto uistinu jesu, posto je to novi
cvor.

struct cvor* NoviCvor (int novibroj) {

    struct cvor *cvor;

    cvor = (struct cvor*) malloc (sizeof(struct cvor));
    cvor->broj = novibroj;
    cvor->lijevi = NULL;
    cvor->desni = NULL;

    return cvor;

}

// Napomena: Ova funkcija umece nove cvorove (s odgovarajucom
vrijednosti) u
// postojece binarno sortirano stablo.
// Funkcija Umetni realizirana je rekurzivno.
// Funkcija Umetni radi vrlo slicno kao i funkcija Trazi. Naime, funkcija
prvo
// provjerava da li na zadano mjesto (kod kojeg vec rekurzivnog poziva)
moze
```

```
// umetnuti novi clan (a to ce dobiti tako da ce clan koji je bio
sljedeci
// u biti biti NULL clan) (1). Ako jos nismo dostigli NULL clan treba
// provjeriti je li broj manji ili veci od sadasnjeg tako da znamo u koju
granu
// krenuti (lijevo ako je manji, tj. desno ako je veci) (2) i (3).

struct cvor *Umetni (struct cvor *cvor, int broj) {

    if (cvor == NULL) {
        return (NoviCvor(broj));        // 1

    } else {
        if (broj <= cvor->broj) {
            cvor->lijevi = Umetni(cvor->lijevi, broj);  // 2

        } else {
            cvor->desni = Umetni(cvor->desni, broj);    // 3

        }
        return cvor;

    }
}


// Napomena: Ovo je funkcija trazenja koja vrijedi za binarno sortirano
stablo
// sortirano po vrijednosti u svojim cvorovima.
// Funkcija trazenja ostvarena rekurzivno.
// Funkcija u svakom svojem pozivanju prvo provjeri je li trenutni cvor
kojim
// slucajem prazan (1). Ako nije, ide se na provjeru jesmo li mozda nasli
// trazeni clan (2). Ako ni to nije slucaj, ocito postoji jos clanova i
moramo
// otici u odgovarajucu granu sljedece razine (lijevo ili desno). Posto
imamo
// sortirano binarno stablo provjeravamo u koju cemo granu na nacin da
// usporedimo trazeni broj sa trenutnim. Ako je trazeni broj manji,idemo
lijevo,
// (3) odnosno desno, ako je veci (4).

int Trazi (struct cvor *cvor, int trazbr) {

    if (cvor == NULL) {
        return 0;               // 1

    } else {
        if (cvor->broj == trazbr) {
            return 1;           // 2

        } else {
            if (trazbr < cvor->broj) {
                return (Trazi(cvor->lijevi, trazbr)); // 3

            } else {
                return (Trazi(cvor->desni, trazbr));  // 4

            }
        }
    }
}
```

```
// Odredjivanje najmanjeg clana u binarnom sortiranom stablu svodi se na
to
// da nadjemo najdonji i najlijeviji clan u stablu. To cemo lako dobiti
ako cemo
// putovati samo po lijevo strani stabla dok ne dodjemo do kraja. Kad
dodjemo
// do kraja ostat ce pridruzena najmanja vrijednost u pom i to cemo
vratiti iz
// funkcije.

int MinClan (struct cvor *cvor) {

    int pom;

    pom = cvor->broj;

    while (cvor->lijevi != NULL) {
        cvor = cvor->lijevi;
        pom = cvor->broj;
    }

    return pom;

}




int main()
{
    int i[10] = {15,5,6,10,54,24,12,1,2,14};
    int j;
    struct cvor *cvor=NULL;

    // Upisivanje vrijednosti u nase stablo.

    for (j = 0; j < 10; j++) {

        cvor = Umetni (cvor, i[j]);

    }

    // Ovaj mali printf samo je provjera radi li nase pretrazivanje
binarnog
    // sortiranog stabla.

    printf ("10 -> %d (1), 13 -> %d (0), 54 -> %d (1), 72 -> %d (0).\n",
        Trazi(cvor,10), Trazi(cvor,13), Trazi(cvor,54), Trazi(cvor,72));

    // Najmanja vrijednost u binarnom sortiranom stablu.

    printf ("Najmanja vrijednost u nasem stablu je: %d (2).\n",
MinClan(cvor));

    return 0;
}
```

7.

GOMILA (ILUSTRACIJA)
+ POTPUNO BINARNO STABLO

UVOD

Ovaj tutorial ce se sastojati od crtanja gomila, zadatka koji bi se vrlo vjerojatno mogao pojaviti na zavrsnom ispitu, a koji je u biti zicer (sigurni bodovi).

Jedino pitanje kojeg cemo se ovdje dotaknuti je - kako se oblikuju gomile? Kada se odgovori na ovo pitanje puno je lakse shvatiti i programsku bit.

U proslom tutorialu pricao sam o sortirani binarnim stablima, kod njih je osnovno pravilo bilo da su manji clanovi uvijek u lijevoj grani, a veci uvijek u desnoj. Kod gomile se slijede neka druga pravila i to tocno 2.

Ta dva pravila su:

1. Najveci clan mora biti prvi clan gomile
2. Clanovi koji su djeca trenutnog clana moraju biti manji od samog clana.

Treba primjetiti kako ne mora najveci clan biti prvi, moze biti npr. i najmanji. To ovisi o tome kako zelite graditi svoju gomilu, no mi cemo graditi tako da je najveci clan prvi clan.

POTPUNO BINARNO STABLO

Postoji jos jedan uvjet kako bi nasa gomila bila pravovaljana, a to je da nase stablo (koje gradimo/crtamo) bude potpuno. Potpuno binarno stablo je takvo stablo koje ima sve razine (osim zadnje) u potpunosti popunjene, a zadnja razina mora biti popunjavanja s lijeva na desno.

Za lakse shvacanje - evo mali primjer (koji cu koristiti kroz cijeli tutorial):

Ulazni niz -> (5, 15, 2, 35, 12, 38, 17, 21, 10)

Nase binarno stablo ce biti gradjeno na sljedeci nacin:

- Kao prvi element ulazi 5 i to je najvisi clan naseg stabla

Kod:

```
5
```

- Nakon toga dodajemo lijevu granu, tj. 15.

Kod:

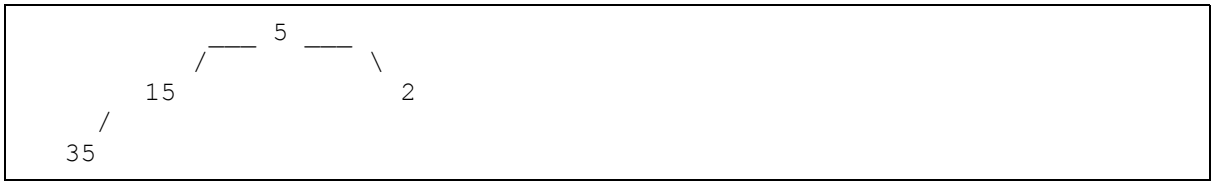
```
          ___ 5 ___
         /
       15
```

- Sada necemo ici opet spustati sljedeci clan na clan 15 vec popunjavamo nase stablo i stavljamo sljedeci clan na desnu granu pocetnog elementa.

Kod:

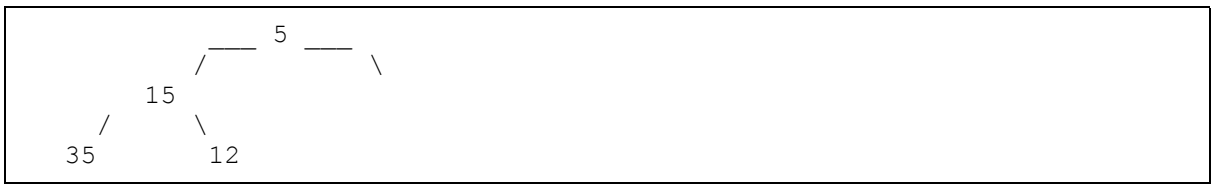
- Sada su gotove dvije razine naseg potpunog stabla. Sljedeci korak je popunjavati ga dalje, naravno kao i do sada, s lijeva na desno. To znaci da dodajemo sljedeci clan (35) na lijevu granu clana 15.

Kod:



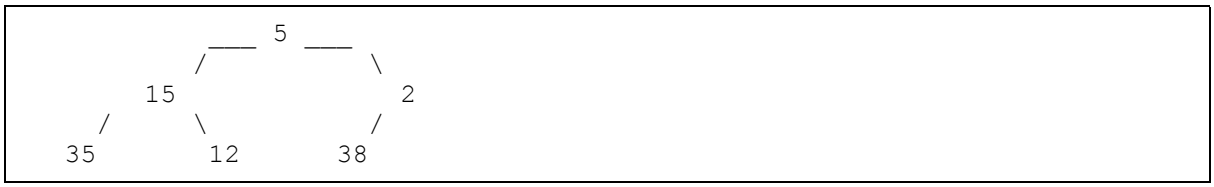
- Sljedeci clan (12) je, logicno, desna grana clana 15 jer je ona, gledano s lijeva na desno na 3. razina, clan koji slijedi.

Kod:



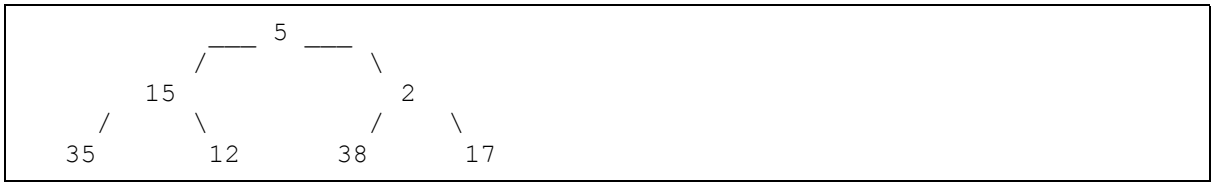
- I idemo dalje sa sljedecim clanom (38) koji ide kao lijeva grana clana 2. Opet ne prelazimo na sljedecu razinu nego ostajemo na ovoj dok nije popunjena, pa tako novi clan dodajemo na clan 2.

Kod:



- Sljedeci clan (17) upotpuniti ce nasu 3. razinu kao posljednji clan iste.

Kod:



- Sada bi sljedeca razina primala 8 clanova, no nama je preostalo samo dva. Taj problem se rijesava kao i do sada, popunjavamo s lijeve strane dok ne potrosimo sve elemente. To je sve. ;)

Kod:

- Ostaje jos samo clan 10 koji dolazi kao posljednji na desnu granu clana 35.

Kod:

```
             ___ 5 ___
           /           \
        15               2
      /    \           /    \
    35      12       38      17
   /  \
 21   10
```

Ovo je konacni oblik naseg stabla. Analizirajmo ga malo. Stablo sadrzi 4 razine, 9 elemenata i potpuno je. Potpuno je jer je oblika kakvog zelimo, naime niti jedna razina koja prethodi posljednjoj nema rupa (nedostajucih elemenata), a zadnja razina popunjena je s lijeva na desno svim preostalim elementima.

I tako izgleda nase potpuno stablo. Sada mozemo krenuti na dva nacina oblikovanja nasih gomila.

OBLIKOVANJE GOMILE

A) SA SLOZENOSTI O(n*logn)

Princip slaganja stabla ovakvom slozenosti je takava da kod dodavanja svakog clana provjerimo njegovog prethodnika je li manji od njega. Ako jest, potrebno ga je zamijeniti clanom koji umecemo. Takodjer provjeravamo i njegovog prethodnika i tako sve dok se broj ne smjesti na svoje mjesto. Ako smo mijenjali neku granu stabla, potrebno je i pregledati cijelu tu granu koja je mijenjana je li ona sredjena.

Ovako to malo djeluje zbunjujuce, no na primjeru cete vidjeti da tu nema nekog problema...

Opet imamo nas ulazni niz koji izgleda ovako - (5, 15, 2, 35, 12, 38, 17, 21, 10). Zelimo od njega stvoriti gomilu... Krenimo!

- Opet krecemo s pocetnim elementom, a to je 5

Kod:

```
             ___ 5 ___
```

- Sljedece dodajemo sljedeci clan na lijevu grana clana 5 i provjeravamo je li 15 veci od 5. Posto to jest, potrebno ih je zamijeniti. To se vidi na sljedeca dva crteza.

Kod:

```
             ___ 5 ___
           /
        15
```

Kod:

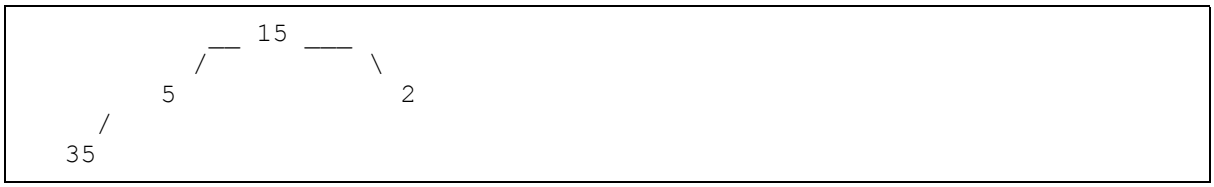
```
             __ 15 ___
           /
         5
```

- Vrijeme je za sljedeci element (2). Njega dodajemo po principu potpunog stabla i ne treba nista mijenjati jer je 2 manje od 15.

Kod:

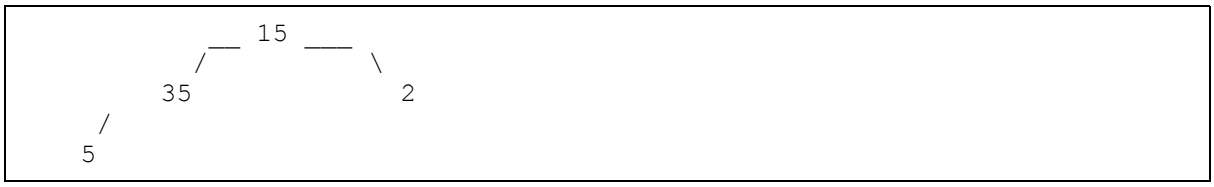
- Krenimo dalje, clan 35 dodaje se na lijevu granu clana 5, opet po principu potpunog stabla.

Kod:



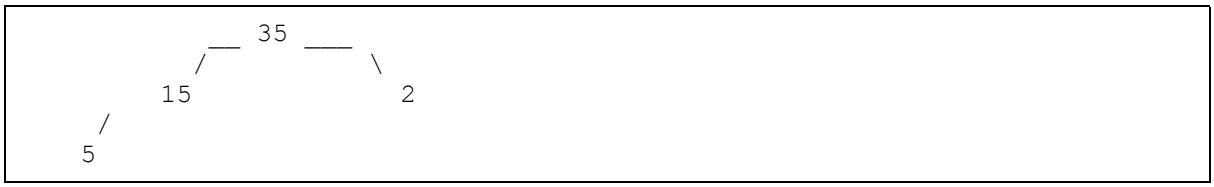
- Dodali smo taj clan, no sada treba provjeriti je li on veci ili manji od svojih prethodnika (roditelja). Provjeravamo za 5 i uvidjamo da jest. Potrebno ih je zamijeniti.

Kod:



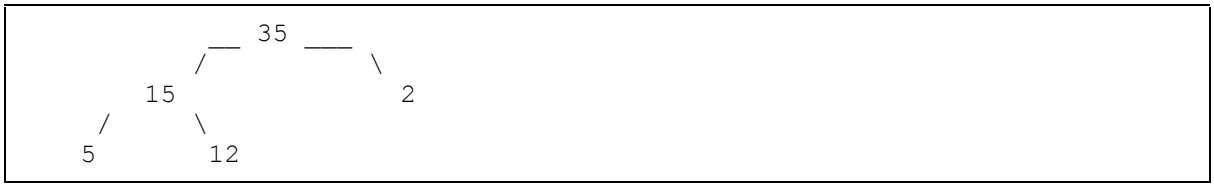
- Sada je potrebno provjeriti da li 35 mozda moze putovati jos "uz granu". Moze, zato sto je 35 veci od 15! Zamijenimo i to!

Kod:



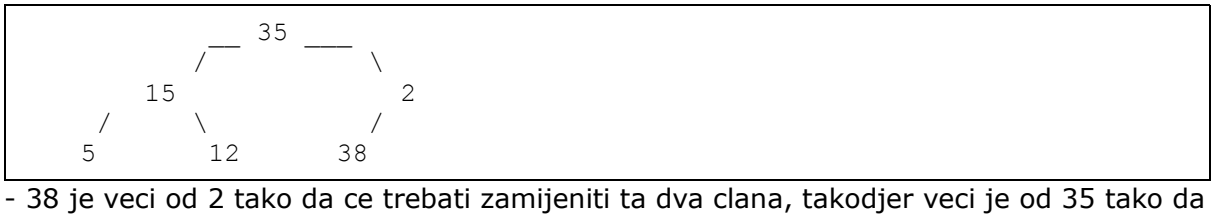
- 35 je dosao na svoje mjesto. Sad dolazimo do onog dijela kada treba provjeriti je li ostatak grane u redu. Grana koja ja mijenjana mijenjana je kod broja 5 i provjeravamo treba li zamijeniti 5 i 15. Ne treba jer je 5 manje od 15. Krenimo na dodavanje sljedeceg elementa.

Kod:



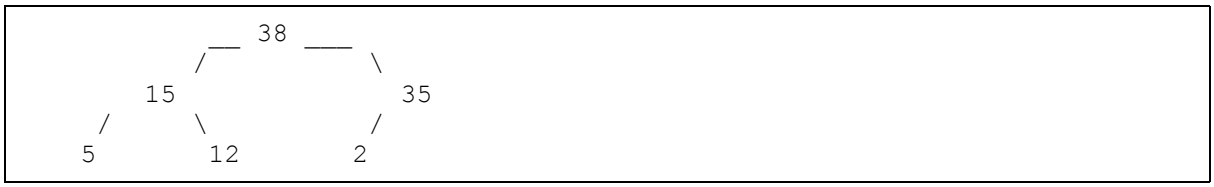
- 12 moze ostati gdje jest zato sto je taj clan manji od svog roditelja. Dodajmo sljedeci element.

Kod:



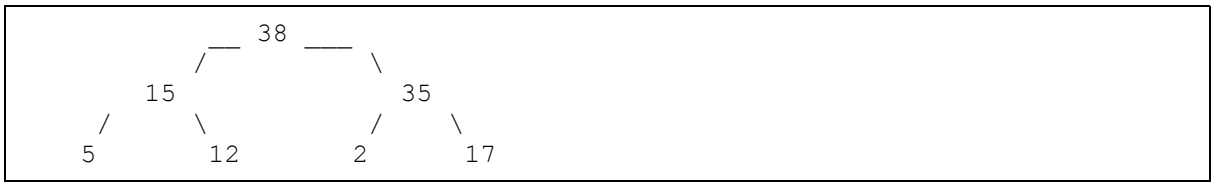
- 38 je veci od 2 tako da ce trebati zamijeniti ta dva clana, takodjer veci je od 35 tako da moze i jos putovati "uz granu". Ta dva koraka prikazat cu kao jedan, a nakon toga slijedi analiza je li potrebno jos sto mijenjati.

Kod:



- Ostatak grane izgleda kako je u redu. Dvojka je manja od 35 i mozemo dalje na dodavanje sljedeceg elementa...

Kod:



- 17 ostaje na mjestu gdje jest jer zadovoljava uvjet (tj. manji je od 35). Dodajemo sljedeci element...

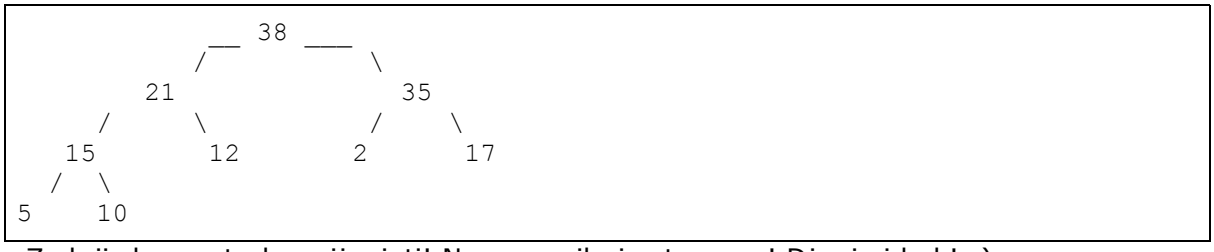
Kod:

- 21 mora putovati "uz granu" i to preko 5 i preko 15. Rezultat tog putovanja ce izgledati ovako (nakon dvije zamjene):

Kod:

- Zamijenili smo prvo sa 5, pa sa 15 i dobili gornju sliku. Provjerimo je li potrebno jos zamijeniti 5 sa 15, no to ocito nije slucaj. Dodajmo i zadnji clan!

Kod:



- Zadnji clan ne treba mijenjati! Nasa gomila je stvorena! Djenja i bok! :)

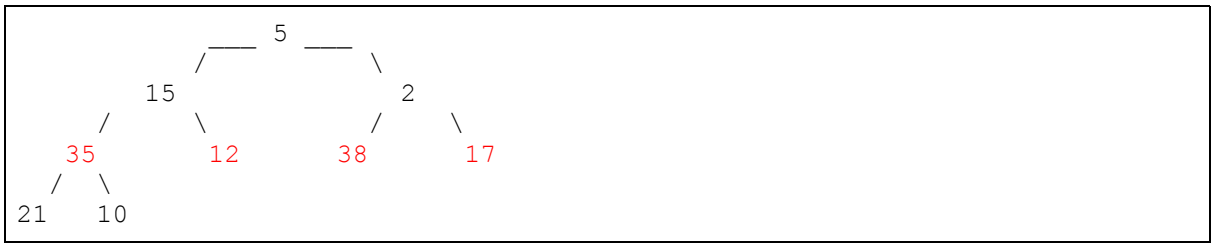
B) SA SLOZENOSTI O(n)

Stvaranje gomile algoritmom slozenosti O(n) ostvaruje se tako da se prvo slozi potpuno stablo (na nacin koji je vec opisan u ovom tutoriali), a nakon toga se krece od predzadnje razine, te se, clan po clan, s desna, gledaju djeca tih clanova. Ako je koje dijete (ili oba) vece od clana kojeg gledamo, zamijenjujemo ih, te takodjer gledamo dublje u granu treba li jos sto zamijeniti. Nakon sto smo prosli sve clanove neke razine, krecemo na visu razinu.

Opet to zvuci malo kompliciran, no zapravo nije, pa krecemo...

- Krenut cemo sa vec stvorenim potpunim stablom od prije i oznaciti razinu koju gledamo

Kod:

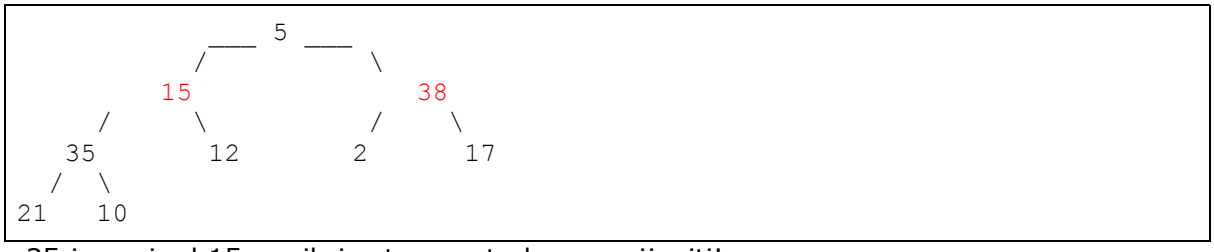


- Provjeravanje djece clanova vrsi se s desne (sto znaci da smo imali djece kod 12 i 38, prvo bi provjeravali njih). No, ovdje imamo samo clan 35 s djecom, pacemo samo njega provjeriti. I 21 i 10 su manji pa tu ne treba nista mijenjati. Krenimo na sljedecu razinu:

Kod:

- Posto krecemo s desna, prvo provjeravamo djecu clana 2. To su 38 i 17 koji su oboje veci od 2, no mi biramo clana koji je veci od ta dva, a to je 38. Mijenjamo ih i krecemo s gledanjem sljedeceg clana (15).

Kod:



- 35 je veci od 15, pa ih je stoga potrebno zamijeniti!

Kod:

- Sada dolazimo do onoga sto vec spominjem neko vrijeme. Posto smo mijenjali 15 i 35 potrebno je provjeriti je li 15 stvarno na pravom mjestu (posto smo tu granu dirali). Ocito nije, posto je 21 veci od 15, pa je potrebno i to zamijeniti.

Kod:

- 2. Razina je takodjer gotova i prelazimo na posljednju, 1. razinu.

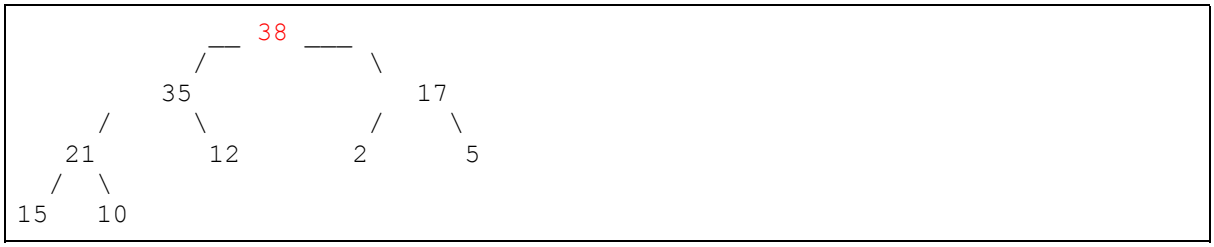
Kod:

- Pogledajmo sada djecu clana 5. I 35 i 38 su veci, no 38 je veci od 35 tako da je to potrebno zamijeniti.

Kod:

- I opet treba provjeriti niz granu i tu se uvidja kako je 17 veci od 5, tako da cemo i to promijeniti.

Kod:



- Gomila je gotova!

Slobodno provjerite, no svi uvjeti gomila su zadovoljeni.

ZAKLJUCAK

Crtanje gomila nije toliko tesko, samo je potrebno malo prakse i naravno, literatura iz koje se to moze nauciti. Sad je imate, pa na posao. ;)

8.

HEAP SORT (ILUSTRACIJA)

I, vrijeme je za zadnji tutorial u ovome semestru! :)

Heap sort necu objasnjavati nista vise do onoga sto vam je potrebno u MI, a to je nacrtati kako on djeluje. Krenimo...

Koristiti cu nas niz iz ranijih primjera, tocnije -> (5, 15, 2, 35, 12, 38, 17, 21, 10).
Takodjer, kod ovog sortiranja bitno je da uz stablo prikazujete i polje clanova, pa ce to takodjer biti slika za sebe, sa svojim zamjenama.

> Kao prvo, treba oblikovati potpuno stablo kao sto je prikazano u tutorialu prije ovog.

Kod:

```
              ___ 5 ___
             /         \
         15              2
       /    \          /    \
    35       12      38      17
   /  \
21   10
```

Kod:

```
| 5 | 15 | 2 | 35 | 12 | 38 | 17 | 21 | 10 |
```

Nakon toga cemo od ovog potpunog stabla naciniti gomilu, a oblikovanje te gomile takodjer mozete vidjeti na gornjem linku (gledajte pod varijantu b, tj. slozenost O(n)).

Kod:

```
              __ 38 ___
             /         \
         35              17
       /    \          /    \
    21       12      2       5
   /  \
15   10
```

Kod:

```
| 38 | 35 | 17 | 21 | 12 | 2 | 5 | 15 | 10 |
```

Samo mala digresija prije nego krenem dalje. U ranijem oblikovanju gomile nisam pokazivao kako se oblikovanje iste odrazava na postojeci niz. Ali, tu zapravo ni nema neke velike magije. Tocno one clanove koje zamijenite na stablu zamijenite u polju, sto bi znacilo da, ako u stablu mijenjate npr. roditelja (38) i dijete (17), u nizu koji izgleda | 1 | 38 | 2 | 3 | 17 | 4 |, sljedeci korak izgledao bi | 1 | 17 | 2 | 3 | 38 | 4 |.

To je sve. ;)

Dobro, sada imamo odlicno postavljene temelje za heap sort - imamo nase polje prebaceno u stablo, te nakon toga u gomilu, prava zabava moze poceti. :)

Heap sort kao takav, ekstremno je jednostavan nakon sto postavite gomilu. To je zato sto se sada svi daljnji dogadjaji odvijaju u 2 koraka, sve dok ne dobijete sortirani niz. Ta dva koraka su:

-> Uzmi korijen (glavu) stabla i spremi ga u novo polje.
-> Premjesti ZADNJI (najdoljni i najdesniji :)) clan stabla na njegovo mjesto i ponovno stvori gomilu.

Nista vise, ali ni nista manje. Da bi to mogli lakse shvatiti idemo s nasim primjerom, pa ce te vidjeti koliko je to pljugica. ;)

NAPOMENA: Ponovno oblikovanje gomile radimo odozgo prema dolje, a ne kao na prijasnjem oblikovanju odozdo prema gore.

Koliko se sjecam, ostali smo na ovome:

Kod:

```
            __ 38 ___
           /         \
         35            17
       /    \        /    \
     21      12     2      5
   /   \
 15    10
```

Kod:

```
| 38 | 35 | 17 | 21 | 12 | 2 | 5 | 15 | 10 |
```

Super, sada, kao sto sam rekao, uzimamo najveci clan i spremamo ga u novo polje, te izbacujemo iz stabla i starog polja taj isti clan.

Kod:

```
            __ 38 ___
           /         \
         35            17
       /    \        /    \
     21      12     2      5
   /   \
 15    10
```

Kod:

```
Staro polje
| 38 | 35 | 17 | 21 | 12 | 2 | 5 | 15 | 10 |
```

Kod:

```
Novo, sortirano polje
| 38 |
```

Nakon ovoga idemo na drugi korak, te premjestamo zadnji clan stabla na mjesto upravo uzetog.

Kod:

```
             __ 10 ___
           /           \
        35               17
     /     \           /     \
   21       12       2       5
  /
15
```

Kod:

```
Staro polje
| 10 | 35 | 17 | 21 | 12 | 2 | 5 | 15 |
```

Kod:

```
Novo, sortirano polje
| 38 |
```

Stvorimo opet gomilu postupkom odozgo prema dolje.

Kod:

```
             __ 10 ___
           /           \
        35               17
     /     \           /     \
   21       12       2       5
  /
15
```

Kod:

```
Staro polje
|10 | 35 | 17 | 21 | 12 | 2 | 5 | 15 |
```

Kod:

```
Novo, sortirano polje
| 38 |
```

I 35 i 17 su veci od 10, no 35 je veci od 17, pa on ima prednost. Potrebno ih je zamijeniti.
Nakon zamijene, opet pregledavamo clanova, ali, kao sto je vec i u tutorialu o oblikovanju gomile receno, gledamo SAMO U GRANU KOJA JE MIJENJANA.
21 je opet veci i od 10 i od 12, pa ima prednost.

Kod:

```
             __ 35 ___
           /           \
        10               17
     /     \           /     \
   21       12       2       5
  /
15
```

Kod:

```
Staro polje
| 35 | 10 | 17 | 21 | 12 | 2 | 5 | 15 |
```

Kod:

```
Novo, sortirano polje
| 38 |
```

Zamijenjujemo ih i krecemo dalje. Ostalo je jos samo provjeriti 15 naspram 10 i tu ce se opet dogoditi promjena.

Kod:

```
             __ 35 ___
           /          \
        21              17
      /    \          /    \
    10       12      2      5
   /
  15
```

Kod:

```
Staro polje
| 35 | 21| 17 | 10 | 12 | 2 | 5 | 15 |
```

Kod:

```
Novo, sortirano polje
| 38 |
```

Nakon te zamjene gotovi smo s oblikovanjem nase gomile.
Pogodite sto ide dalje!
Tak je ;), opet uzimamo korijen stabla, vadimo ga van i stavljamo u sortirano polje, a na ovom nasem starom opet treba oblikovati gomilu.

Kod:

```
             __ 35 ___
           /          \
        21              17
      /    \          /    \
    15       12      2      5
   /
10
```

Kod:

```
Staro polje
| 35 | 21| 17 | 15 | 12 | 2 | 5 | 10 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 |
```

Eto, vec dva clana sortirana! Ajmo jos jednom sloziti gomilu...

Prvo se korijen nadomjescuje zadnjim elementom (vidi, opet 10 :)).

Kod:

```
             __ 10 ___
           /          \
        21              17
```

```
      /     \           /     \
    15       12       2       5
```

Kod:

```
Staro polje
| 10 | 21| 17 | 15 | 12 | 2 | 5 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 |
```

I opet slazemo gomilu korak po korak, s time da cu sada odmah oznaciti sto se sve zamijenjuje, da bas ne bude previse koraka. :)
Znaci, sto se sve zamijenjuje u sljedecem oblikovanju gomile?
Mijenjaju se prvo 10 i 21 jer je 21 veci i od 10 i od 17, pa ima prednost.
Takodjer, kad 10 dodje na mjesto 21, potrebno je zamijeniti i 10 i 15 posto je 15 vece od 10 (a i od 12, pa ima prednost).

Kod:

```
           __ 10 ___
         /          \
       21             17
      /     \        /     \
    15       12     2       5
```

Kod:

```
Staro polje
| 10 | 21 | 17 | 15 | 12 | 2 | 5 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 |
```

Nakon mijenjanja opet imamo gomilu, pa opet uzmima korijen.

Kod:

```
           __ 21 ___
         /          \
       15             17
      /     \        /     \
    10       12     2       5
```

Kod:

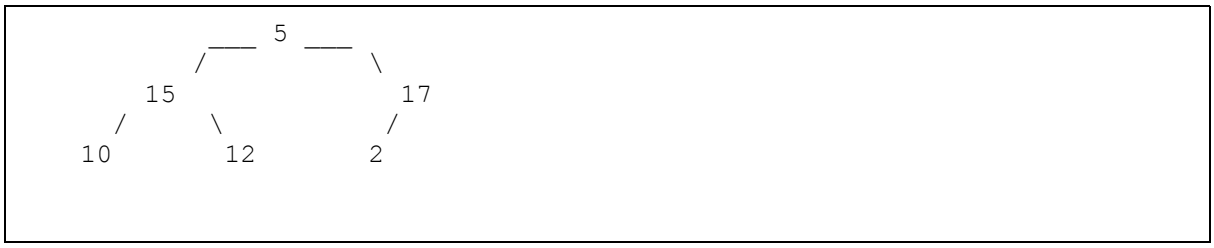
```
Staro polje
| 21 | 15 | 17 | 10 | 12 | 2 | 5 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 | 21 |
```

3 slozena clana! Vrijeme za *cagicu*! :)
Nakon *cagice* :) opet uzimamo zadnji clan (to je sada 5) i stavljamo na prvo mjesto.

Kod:



Kod:

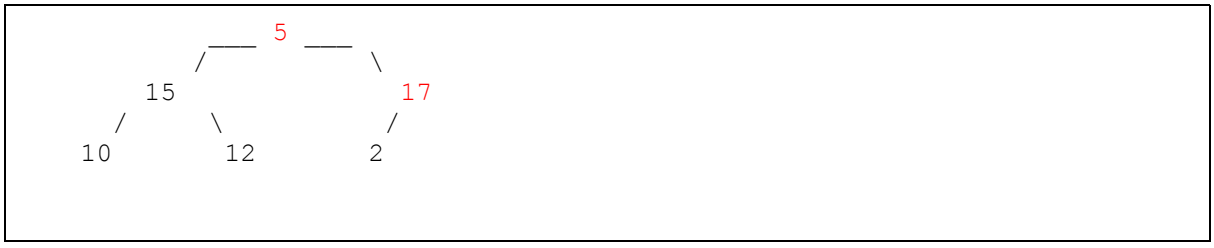
```
Staro polje
| 5 | 15 | 17 | 10 | 12 | 2 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 | 21 |
```

Nase stablo je vec poprilicno maleno. :)
Opet slazemo gomilu i to mijenjamo 17 i 5, a dvojka (tj. nastavak te grane) ostaje gdje je jer je manja od 5.

Kod:



Kod:

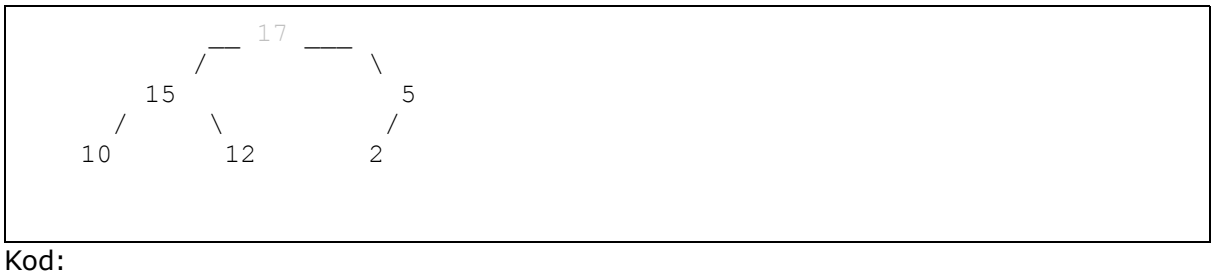
```
Staro polje
| 5 | 15 | 17 | 10 | 12 | 2 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 | 21 |
```

Nakon zamjene opet uzimamo korijen kao sljedeci clan sortiranog polja.

Kod:



Kod:

```
Staro polje
| 17 | 15 | 5 | 10 | 12 | 2 |
```

Kod:

```
Novo, sortirano polje
| 38 | 35 | 21 | 17 |
```

I sada vec imamo 4 clana.
Ostatak sorta vise necu raditi jer ste stvarno mogli shvatiti bit. Sljedeci clan koji dolazi na vrh je dvojka koja ce se onda spustiti niz granu clana 15 i 12, te ce nakon toga 15 biti izbacen van, itd. itd.

I jos nesto - ako slucajno dodje da sortirate silazno, morate samo raditi tako da gomila uvijek bude stvarana tako da je najmanji element na vrhu. Znaci, vrijedi sve isto kao ovdje, samo je bitno da uvijek vrijedi da je roditelj manji od oboje djece.

Pozdrav i puno srece! :)