STRUKTURE PODATAKA I ALGORITMI

Skripta - 2022 / 2023

CLASS

Klasa (eng. class) je korisnički – definiran tip podataka koji koristimo u našem programu. Klasa služi kao svojevrsni "blueprint" za kreiranje objekata. Članovi koji se nalaze u klasi su po default-u privatni, dok su u slučaju strukture javni.

Koja je razlika između klase i strukture ako su oboje "ista" stvar? **Nema je**… Razlika je u tome što točno planiramo raditi.

- Ako imamo potrebu za tipom čija je glavna namjena čuvanje podataka bez puno operacija na njima ---> **Struktura**
- Za sve ostalo ---> **Klasa**

Kako definiramo klasu? Klasa se definira na sljedeći način:

```
#pragma once
class Rectangle
{
private:
	double a, b;
public:
	// konstruktori
	Rectangle(double a, double b);

	// seteri
	void set_a(double a);
	void set_b(double b);

	// geteri
	double get_a();
	double get_b();

	// funkcije s kojima radimo
	double opseg();
	double povrsina();
};
```

Na slici se može vidjeti da se klasa radi po istom principu kao i struktura, jedina je razlike u tome što sada postoje **private** i **public** koji definiraju hoće li se varijable moći koristiti samo unutar klase ili van nje.

- Varijable koje se nalaze u private dijelu mogu se koristiti samo unutar klase, ne i u mainu.
- Varijable koje se nalaze u public dijelu mogu se koristiti van klase, tj. u mainu.

Što sve spremamo u public dio klase, a što u private dio?

- **PRIVATE** – private sadržava sve vaše varijable koje ćete koristiti, bilo to int, double, float. Uz to, tu se stavljaju sve varijable koje se ne koriste u mainu.
- **PUBLIC** – u public dio se stavljaju sve metode (funkcije), konstruktori i destruktori koje definiramo i koje želimo koristiti u mainu

Kako kreiramo klasu?

Desni klik na projekt ---> Add ---> Class ---> Klasi damo neki naziv ---> OK ---> Dobili smo Header file (.h) i cpp file. (CPP file će kasnije biti objašnjen)

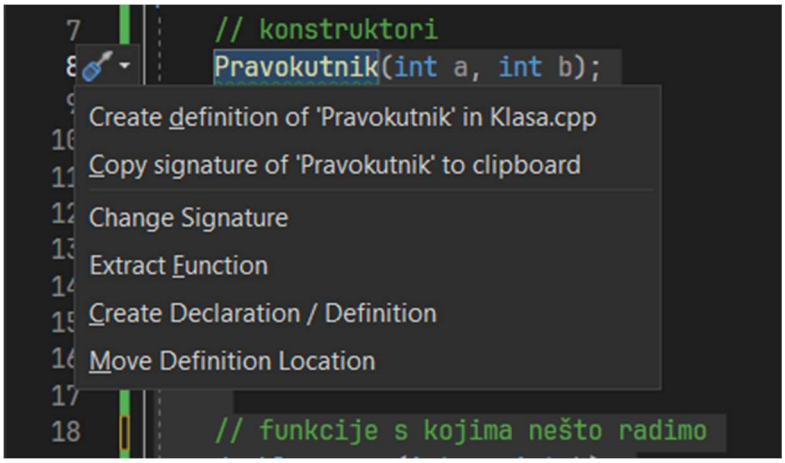
#pragma once dobijate po default-u u vašem header file-u, on služi kao include zaštita i njega ostavljate takvim kakav jest (detaljnije od ovoga nije potrebno znati).

Kako jednostavno deklariramo sve funkcije i inicijaliziramo varijable i metode?

1. KORAK – selectamo public dio

```
6   public:
7       // konstruktori
8       Pravokutnik(double a, double b);
9
10      // seteri
11      void set_a(double a);
12      void set_b(double b);
13
14      // geteri
15      double get_a();
16      double get_b();
17
18      // funkcije s kojima nešto radimo
19      double opseg(double a, double b);
20      double povrsina(double a, double b);
21  };
```

2. KORAK – kliknemo ALT + ENTER



3. KORAK – Kliknemo na Create Declaration / Definition

Kada smo to kliknuli, napokon ulazimo u nas .cpp file koji smo dobili uz .h file.

.cpp file bi vam inicijalno trebao izgledati ovako:

```cpp
#include "Rectangle.h"

Rectangle::Rectangle(double a, double b)
{
}

void Rectangle::set_a(double a)
{
}

void Rectangle::set_b(double b)
{
}

double Rectangle::get_a()
{
    return 0.0;
}

double Rectangle::get_b()
{
    return 0.0;
}

double Rectangle::opseg()
{
    return 0.0;
}

double Rectangle::povrsina()
{
    return 0.0;
}
```

Na slici se može vidjeti na nema onih varijabli koje su definirane u .h file-u. Iako ih ne vidimo, one su tu i aktivno ih možemo koristiti u daljnjem radu.

Za inicijalizaciju varijabli postoji nekoliko metoda, dvije najvažnije su:

- Inicijalizacija konstruktorom
- Inicijalizacija seterima

Inicijalizacija seterima izgleda ovako:

```
void Rectangle::set_a(double a)
{
    this->a = a;
}

void Rectangle::set_b(double b)
{
    this->b = b;
}
```

Kao što se može primijetiti, za inicijalizaciju varijable a i b koristili smo pointer **this. This** je pointer na samoga sebe (u prijevodu pokazuje na našu varijablu). This pointere koristimo umjesto drugih načina jer stvaramo dodatne varijable (ili kopije varijabli) kako bi inicijalizirali varijablu.

Inicijalizacija konstruktorom izgleda ovako:

```
Rectangle::Rectangle(double a, double b)
{
    this->a = a;
    this->b = b;
}
```

Kasnije ćemo detaljnije objasniti kako konstruktori funkcioniraju, te kako zapravo ovo radi. Za sada se držimo inicijalizacije seterima.

Za dohvaćanje potrebnih varijabli koristimo getere koje smo definirali ranije. Getere nije preporučljivo previše koristiti jer je cilj napraviti program koji se neće u potpunosti “otvoriti” korisniku (u prijevodu, getane varijable postaju javne, a to nije nužno dobro).

Dohvaćanje geterima izgleda ovako:

```
double Rectangle::get_a()
{
    return a;
}

double Rectangle::get_b()
{
    return b;
}
```

Za što nam služe geteri? Suština getera je da vrati traženu varijablu. U ovom slučaju, ako u mainu pozovemo funkciju iz klase naziva **get_b();** , ona će ga dohvatiti i vratiti varijablu nama (kasnije ćemo demonstrirati kako to izgleda).

Defaultni konstruktor vs. Korisnički definiran konstruktor:

Konstruktor je po definiciji metoda koja stvara objekt na klasi. Konstruktor može biti definiran na više načina, a jedan od njih je onaj defaultni.

Defaultni konstruktor je automatski generiran konstruktor koji se poziva onog trenutka kada u main stavimo **Pravokutnik p1;** Defaultni konstruktor je izuzetno opasan. Zašto je opasan? Opasan je zbog toga što one varijable a i b koje smo prije definirali neće biti inicijalizirane, nego će se u njima naći smeće.

Korisnički definiran konstruktor je onaj konstruktor koji je jasno definiran u klasi i na kraju pozvan u mainu. To je u suštini konstruktor kojemu najčešće šaljemo vrijednosti pri pozivu. Korisnički definiranom konstruktoru kojemu nije poslana niti jedna varijabla možemo namjestiti da iste defaultno inicijalizira. Objasnimo pomoću slika…

```
Rectangle::Rectangle(double a, double b)
{
    this->a = a;
    this->b = b;
}
```

Ovo je korisnički definiran konstruktor. Kao što se može vidjeti, konstruktor prima nekakve dvije varijable, te ih inicijalizira na zadanu vrijednost. U ovom slučaju, kakvu god vrijednost pošaljemo konstruktoru za varijable a i b, on će ih inicijalizirati, te će njihova vrijednost biti ona koju smo definirali (ovo je ujedno i razlog zašto je ovo bolje od defaultnog).

Kako pozivamo korisnički definiran konstruktor?

Korisnički definiran konstruktor pozivamo kroz main na sljedeći način:

```
#include <iostream>
#include "Rectangle.h"

using namespace std;

int main()
{

	Rectangle r(1, 2);

	return 0;
}
```

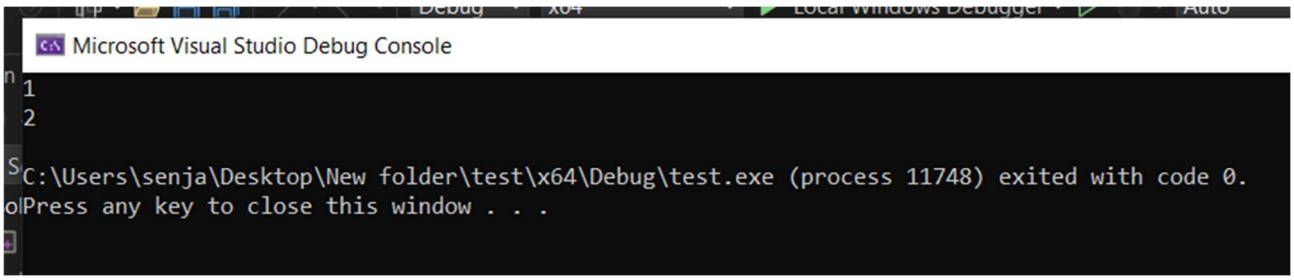
Vidi li se sada povezanost između konstruktora u mainu i metode u .cpp file-u? Odgovor je naravno! Pozivom konstruktora **Pravokutnik p(1, 2);** mi smo zapravo poslali vrijednosti varijable a i varijable b.

Kako to dokazati?

Jednostavno! Princip je isti kao i kod struktura iz prvog semestra:

```
#include <iostream>
#include "Rectangle.h"

using namespace std;

int main()
{

	Rectangle p(1, 2);

	cout << p.get_a() << endl;
	cout << p.get_b() << endl;

	return 0;
}
```

Dokaz da konstruktor funkcionira:



NAPOMENA:
Primijetite da u mainu stoji nova vrsta include-a. **#include "Pravokutnik.h"** nam omogućuje korištenje našeg header file-a i onoga što smo definirali u .cpp u mainu (zato i možemo pozvati **Pravokutnik p(1, 2);**). Header se uvijek uključuje u main ako ga želimo koristiti.

STRINGSTREAM

Uz sve što smo do sada prošli, postoji još jedan način kojim možemo ispisati ono što nas zanima, a to je stringstream. Stringstream je objekt, što u suštini znači da mu možemo dati neki naš naziv. Stringstream kao takav nije vezan za konzolu jer vraća string, pa je tako zapravo može pozvati bilo koji klijent, a ne samo konzola.

Stringstream još možemo opisati kao "bačvu" u koju stavljamo sve što mi želimo, te on na kraju sve to izbacuje kao jedan string

Da bi uopće koristili stringstream, moramo prvo napraviti metodu:

```
string to_string();
```

Da bi ova metoda radila moramo u naš header includeati **<string>** i ispod napisati **using namespace std;**

Zašto? Stringstream je stream stringova…

Kada smo to odradili, u našem .cpp file-u moramo uvesti novu vrstu includeati, a to je **<sstream>**. **<sstream>** omogučava korištenje stringstream-a, a princip rada sa stringstream-om je apsolutno isti kao i kod **cout-a.**

```
string Rectangle::to_string()
{
    stringstream ss;

    ss << get_a() << ", " << get_b();

    return ss.str();
}
```

MJERENJE VREMENA TRAJANJA ALGORITAMA

Kako bismo mjerili vrijeme trajanja izvršenja nekog programa/algoritma koji smo napisali, koristimo za to ugrađene funkcije koje dolaze s novom vrstom headera **#include <chrono>.**

Chrono je library za mjerenje / korištenje vremena. Mi ćemo ga koristiti najviše za mjerenje trajanja nekog algoritma.

Kako se to radi?

```
#include <iostream>
#include <chrono>
#include "Rectangle.h"

using namespace std;
using namespace std::chrono;

int main()
{
	auto begin = high_resolution_clock::now();
	Rectangle p(1, 2);

	cout << p.get_a() << endl;
	cout << p.get_b() << endl;

	auto end = high_resolution_clock::now();

	cout << duration_cast <nanoseconds> (end - begin).count() << endl;

	return 0;
}
```

Na slici možemo vidjeti da smo prvo includali **<chrono>** koji nam omogučava korištenje objekata **high_resolution_clock::now()**, te funkcije **duration_cast <nanoseconds>**. Kada smo njega includali, možemo napisati dio koda koji će mjeriti vrijeme. Na slici su to **auto begin** i **auto end** koji naznačavaju početak i kraj mjerenja vremena u našem programu.

Sve što se nalazi između begina i enda upada u mjerenje.

Preko funkcije **duration_cast <nanoseconds> (end – begin).count()** ispisujemo izmjereno vrijeme.

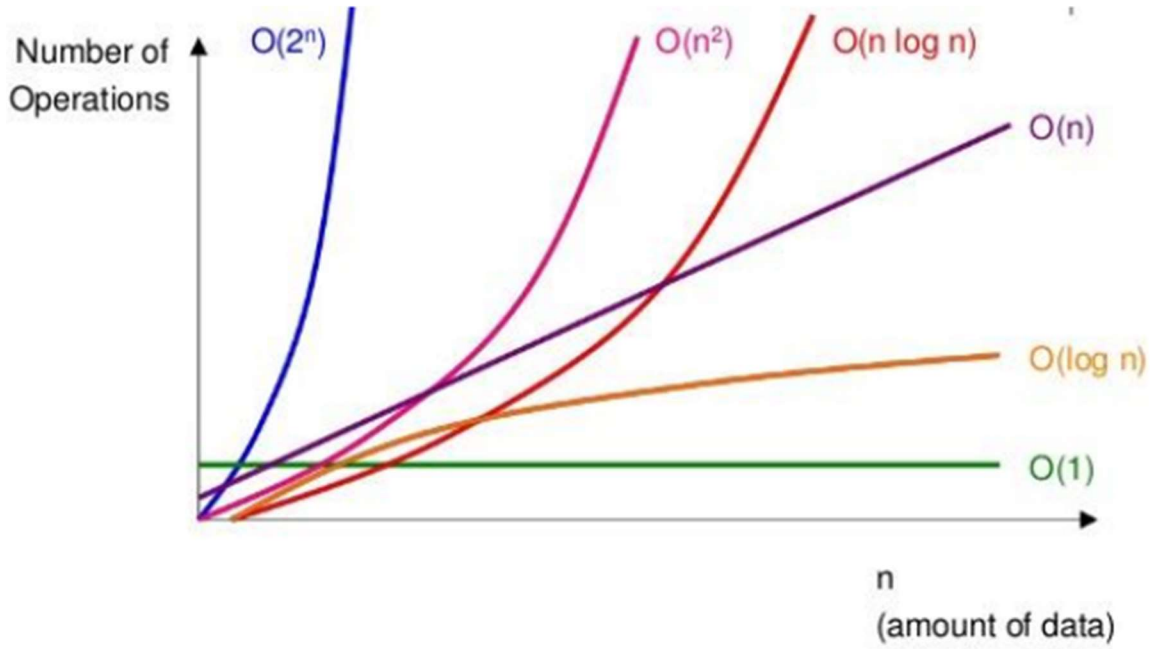
Valja napomenuti da se ovaj kod iznad može znatno skratiti eliminacijom chrono:: dijelova u kodu, a to se radi tako da u naš kod upišemo **using namespace std::chrono**.

KOMPLEKSNOST ALGORITAMA

Kompleksnost algoritama s kojima radimo mjeri se funkcijama koje moramo znati napamet:

- $f(n) = 1$
- $f(n) = \log n$
- $f(n) = n$
- $f(n) = n \log n$
- $f(n) = n^2$ (ili n^3, n^4, ...)
- $f(n) = 2^n$
- $f(n) = n!$

Odnos funkcija s gornje slike izgledaju ovako:



Na ispitu ćemo morati znati raspoznati koje funkcije su najbolje, za određeni uvjet. Tablica iznad prikazuje koja je od navedenih funkcija najbrža, a koja najsporija.

ARRAY

S array-em kao takvim smo se upoznali još u prvom semestru na programiranju. Tada smo ga deklarirali tako što smo samo napisali vrstu podatka, te naveli količinu, i to je bilo to.

U ovom slučaju tok priče ide u potpunosti drugačije:

```
#include <iostream>
#include <array>
#include <algorithm>

using namespace std;

int main()
{

	array <int, 100> arr;

	return 0;
}
```

Koja je razlika? Razlika je u tome što je array zapravo omotač oko našeg regularnog polja, što zapravo naš array čini klasom koja dolazi s novim funkcionalnostima koje regularno polje nema.

Osnovna razlika između array-a i vectora je u tome da je array “size immutable”, što znači da njemu moramo zadati neku veličinu koju nikad nećemo mijenjati.

Kako bi zapravo dobili mogućnost funkcionalnosti array-a, moramo prvo includati **<array>**, te zatim ako želimo i one funkcionalnosti koje ćemo analizirati, moramo includati i **<algorithm>**.

Uvođenjem array-a dobili smo na korištenje nekoliko novih funkcija koje će nam kasnije biti jako korisne. Sve te funkcije bit će obrađene u vektorima i jedan dio u Ishodu 2. Sintaksu za korištenje array-a nećemo pokazivati jer je sitaksa apsolutno ista kao i za obično polje, samo što se array ispod haube ponaša drugačije od regularnog polja.

VECTOR

Vector smo radili u prošlom semestru, u ovome ga radimo još detaljnije. Sada uvodimo par novih stvari što se tiče vectora.

Prije svega, vector je omotač oko polja koje može mijenjati veličinu. Elementi vectora su poslagani jedan iza drugog u memoriji (to nam govori i **.push_back()** kao optimalan način ubacivanja elemenata u vector). Vektor ima jednu manu, rezerviramo li kapacitet od 50 elemenata i probamo li ubaciti 51. element u vector alocirat će se novo, veće dinamičko polje. To je jako loše, jer se polje ne povećava uvijek za 1 element, nego čak za 1.000.000 elemenata ako se zadese uvjeti za to. **U suštini, rast vectora se dešava eksponencijalno!**

Postoje šest načina izrade vectora:

- **vector <int> jedan** – kreiranje praznog vectora
- **vector <int> dva (n)** – kreira vector od n elemenata inicijaliziranih na defaultnu vrijednost
- **vector<int> tri (n, val)** - Kreira vektor od n elemenata, svaki je kopija od val (fill)
- **vector<int> cetiri(tri.begin(), tri.end())** - Kreira vector kopiranjem elemenata iz zadanog raspona (range)
    - Prva vrijednost je početna adresa (uzima se i element na toj adresi)
    - Druga vrijednost je zadnja adresa (element na toj adresi se ne uzima)
- **vector <int> pet (tri)** - Kreira vector na način da kopira sve elemente iz zadanog vectora (copy)
- **vector<int> sest({ 11, 22, 33 })** – direktna inicijalizacija pomoću liste

Vektorima možemo provjeriti veličinu i kapacitet preko funkcija **.size()** & **.capacity()** respektivno.

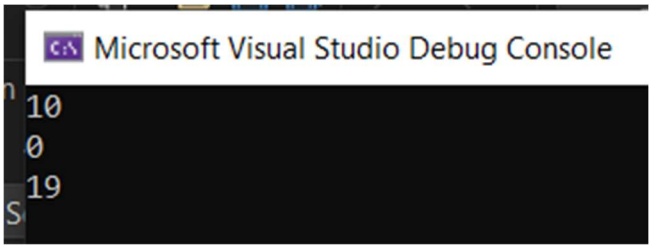
Veličinu i kapacitet vektora možemo promjeniti i ručno pozivom funkcija **.resize(n)** & **.reserve(n)**.

Vektori nude nekoliko načina pristupa elementima

- **v[i]** vraća referencu na element na mjestu i
- **v.at(i)** vraća referencu na element na mjestu i
- **v.front()** vraća referencu na prvi element
- **v.back()** vraća referencu na zadnji element

Primjer rada navedenih funkcija:

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

using namespace std;

int main()
{

	vector <int> v;

	for (int i = 0; i < 20; i++)
	{
		v.push_back(i);
	}

	cout << v.at(10) << endl;
	cout << v.front() << endl;
	cout << v.back() << endl;

	return 0;
}
```



Vektori imaju i funkcije za brisanje koje glase:

- **v.clear()** – čisti cijeli vektor
- **v.erase(v.begin(), v.end())** – radi istu stvar kao i clear, samo što se koristi pointerima kako bi se očistio izabrani range. Ovom funkcijom možemo obrisati i prvih 10 elemenata tako što modificiramo funkciju da bude **v.erase(v.begin(), v.end() + 10).**

Vektori uz funkcije za brisanje imaju i funkcije za umetanje koje će nam biti jako važne kasnije_

- **v.insert(v.begin(), i)** – insert je funkcija koja nam omogućuje da vektoru damo poziciju na koju želimo insertirati željeni element na određenu poziciju. Uz to ova funkcija dozvoljava umetanje na početak pomoću neke funkcije ili konkretno umetanje nečega preko for petlje. Insert **NE RADI** s indexima, nego isključivo s pointerima. To je ujedno i razlog zašto insertu šaljemo pointer na početak vektora.

- **v.assign()** – assign je funkcija koja nam omogućuje dodjelu / prijenos elemenata iz jednog vektora u drugi. Recimo da imamo vektore v1 i v2. Vektor v1 je ispunjen brojevima, a mi želimo prenijeti elemente iz vektora v1 u vektor v2. Jedan od načina je da pišemo for petlju i zakompliciramo stvari, a drugi je da funkciji assign koju ćemo imenovati v2.assign() u zagrade upišemo pointere na prvi i zadnji element vektora v1. Krajnji izraz izgleda ovako: **v2.assign(v.begin(), v.end())**

Kasnije ćemo obraditi i objasniti funkcije poput emplace_back(), pop_front(), pop_back(), push_front(), itd…

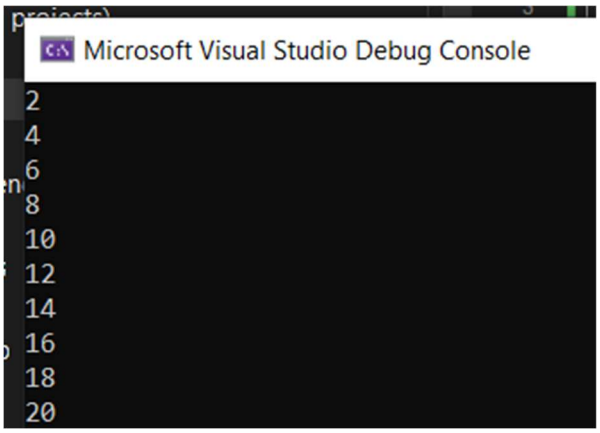
FOR_EACH

for_each() funkcija je jedna jako zanimljiva funkcija koja dolazi includana u <algorithm>. Ova funkcija nam služi kao svojevrsna zamjena za for petlju, jer njoj možemo poslati pointere na elemente u nekom range-u i uz to možemo poslati i naziv funkcije koju želimo da se izvrši za svaki taj element u range-u.

Demonstrirajmo ovo kroz jedan vrlo jednostavan primjer:

Recimo da želimo iz vektora ispisati svaki parni broj između 1 i 20. To bi odradili ovako….

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

using namespace std;

void parni(int& n)
{

	if (n % 2 == 0)
	{
		cout << n << endl;
	}

}

int main()
{

	vector <int> v;

	for (int i = 0; i < 20; i++)
	{
		v.push_back(i);
	}

	for_each(v.begin(), v.end(), parni);

	return 0;
}
```



Da sumiramo izvedbu ovog koda: kada smo pozvali for_each dodijelili smo mu pointer na prvi element i na zadnji element (exlusive (znaci da gleda na zadnji + 1)), te smo mu nakon toga predali pointer na funkciju koja prima referencu na int.

Uz for_each funkciju, valja napomenuti jednu funkciju koja do sada nije spomenuta. Recimo da imamo neko polje koje želimo ispisati u nazad. Logično, posežemo odmah za for petljom što predstavlja komplikaciju. DA bismo se riješili komplikacija <algorithm> nam nudi **reverse( )** funkciju kojoj predajemo pointere na prvi i na zadnji element.

RANDOM GENERATOR

Kako radimo RNG?

Na žalost, u C++ ne postoji funkcija koju možemo samu po sebi pozvati koja će nam generirati neke random brojeve iz nekog raspona. Valja napomenuti da zapravo i postoji, ali uvijek dobijate isti raspon random brojeva. Npr. prvi put dobijete brojeve 1, 4, 5, 7 i onda opet pokrenete program, dobit ćete opet iste brojeve.

Kako to rješavamo? Cijeli proces rješavamo tako da sami napravimo svoju funkciju koja daje random brojeve od nekog raspona. Kako ćemo to izvesti? Zapravo je vrlo jednostavno:

1. KORAK

Da bi započeli cijeli proces moramo includati zaglavlje **<ctime>** koje nam omogućuje korištenje funkcije **srand**.

2. KORAK

U mainu postavimo tzv. seed **srand(time(nullptr)).**

```
#include <iostream>
#include <ctime>
#include <algorithm>

using namespace std;

int main()
{

	srand(time(nullptr));

	return 0;
}
```

Valja napomenuti da ne moramo staviti **nullptr** unutar time(), compiler prihvaća i **NULL**.

3. KORAK

Nakon toga radimo funkciju za generiranje random brojeva.

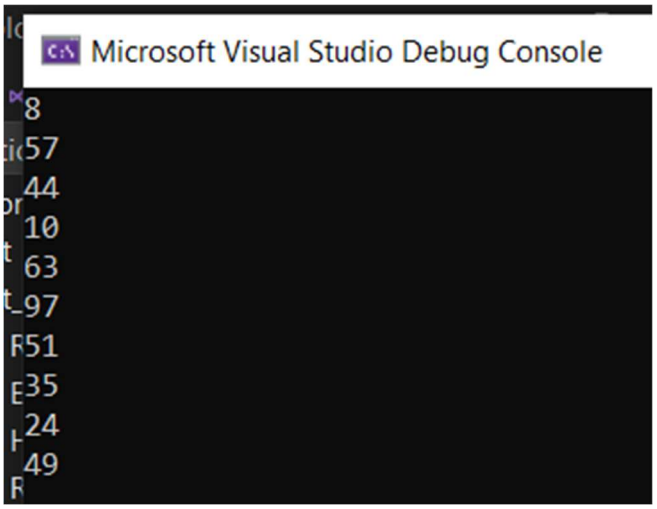
```
int gen_rnd(int min, int max)
{
	return rand() % (max - min + 1) + min;
}
```

4. KORAK

Popunimo (u ovom slučaju) vektor s 10 random brojeva od 0 – 100, te ga isprintamo.

```
void print(int& n)
{

	cout << n << endl;

}

int main()
{
	srand(time(nullptr));

	vector <int> v;

	for (int i = 0; i < 10; i++)
	{
		v.push_back(gen_rnd(0, 100));
	}

	for_each(v.begin(), v.end(), print);

	return 0;
}
```

Rezultat programa:



ISHOD 01 – Part 3:

- File including
- Parsiranje & Template

FILE INCLUDING

U narednom periodu jako često ćemo se susretati s .txt & .csv datotekama ili vanjskim .cpp & .h datotekama. Da bi mogli raditi s tim fajlovima, iste moramo ubaciti u naš projekt.

File se includa:

Desni klik na projekt ---> Open in file explorer ---> Nađemo našu datoteku i prevućemo je u folder našeg projekta ---> Kliknemo na Show All Files ikonu koja se nalazi u alatnoj traci solution explorera u Visual Studiu ---> Desni klik na datoteku koju želite includati ---> Include in Project

Valja napomenuti da .csv i .txt datoteke ne morate includati, možete ih samo ubaciti u projekt i prekopirati naziv u ifstream / ofstream. Datoteke .cpp & .h morate includati u projekt.

PARSIRANJE & TEMPLATE*

Odmah na samom početku, template koji će biti objašnjen naknadno nije obavezan na ispitu, ali vam olakšava život kod parsiranja za barem x100.

Parsiranje je jedan od zadataka koji Vas čeka na ispitu. Za one koji se nisu susretali s parsiranjem datoteka, morat će dosta vježbati jer se kod mora znati napamet. Imajte na umu da je parsiranje datoteka samo primjena dosadašnjeg znanja koje ste stekli na SPA + razumijevanje funkcija koje smo do sada radili (stringstream i getline). Cijeli proces parsiranja je pun vrlo interesantnih detalja koje je potrebno dobro proučiti i provježbati. Detaljnije primjere parsiranja datoteka imate na Infoeduci, a mi ćemo proći samo jedan.

Parsiranje se sastoji od nekoliko koraka:

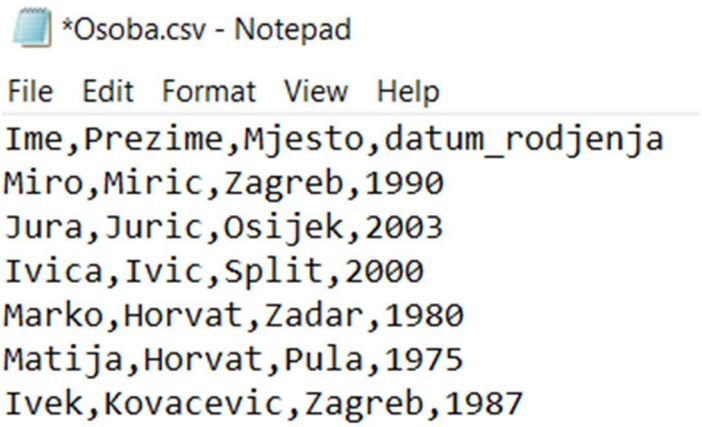
- Analiziranje datoteke (provjera s čime baratamo, tipovi datoteka, itd.)
- Ubacivanje datoteke u projekt koju želimo parsirati
- Stvaranje klase i definiranje varijabli, metoda,…
- Definiranje funkcija, stvaranje potrebnih konstruktora
- Pozivanje datoteke i provjera datoteke
- Parsiranje datoteke
- Rad s parsiranom datotekom

1. KORAK – Analiziranje datoteke

Kao primjer koristit ćemo datoteku Osoba.csv preko koje ćemo objasniti parsiranje.

Kod parsiranja s datotekama treba biti jako oprezan kako se barata istima. Jednu stvar koju ćete najčešće čuti je to da ne vjerujete izgledu fajla. Prva stvar koju morate napraviti je to da u windows exploreru uključite prikaz file extensiona. Zašto nam je to bitno? Probamo li otvoriti Osoba file sam po sebi, otvorit će nam se excel tablica. To, naravno, nije dobro po nas. Kao što se vidi iznad baratamo s .csv fajlom koji zapravo znači "Comma Separated Values". Kao što se može vidjeti, to nije excel, nego obična tekstualna datoteka čiji su elementi međusobno odvojeni zarezom (…razmakom ili bilo kojim drugim znakom).

Kada smo riješili jedan problem koji se najčešće događa, pređimo na analizu datoteke:



Kao što se vidi na slici baratamo s podatcima koji su redom: Ime, Prezime, Mjesto i datum_rodjenja. Sada kada malo bolje pogledamo, da bi ove podatke mogli parsirati, moramo odrediti tip podataka za svaki od stupaca. Ako bi sada išli upisati ime, prezime i mjesto direktno u visual studiu, prvo što bi napravili je to da se pozovemo na string varijable (u prijevodu, tri navedene će biti stringovi). Pratimo li logiku, datum rođenja će biti int varijabla. Super! Sada znamo da u nađemo kodu moramo imati tri stringa i jedan int.

2. KORAK - Ubacivanje datoteke u projekt koju želimo parsirati

Proces ubacivanja je objašnjen na početku ovog poglavlja, te je zapravo isti. Valja napomenuti da ne moramo datoteku includati u naš projekt, nego ju samo možemo copy + paste u folder u kojem se nalazi naš projekt.

3. KORAK - Stvaranje klase i definiranje varijabli, metoda,…

Osnova source.cpp fajla je ista kao i što smo do sada pisali, te ga nećemo dodatno prolaziti.

Da bi mogli uopće mogli parsirati datoteku, trebamo imati nekakvu klasu. Klasu radimo na isti način na koji je prije objašnjeno, te definiramo private i public dijelove klase. Nakon toga se moramo prisjetiti da smo tijekom analize spomenuli da imamo četiri varijable od kojih su tri stringa i jedan int.

```h
#pragma once
#include <string>

using namespace std;
class Osoba
{
private:
	string ime, prezime, mjesto;
	int godina_rodjenja;
public:
	Osoba(string ime, string prezime, string mjesto, int godina_rodjenja);
};
```

Da bi osigurali da naš kod radi kako treba, moramo definirati i naš konstruktor. Kao što znamo, rekli smo da je defaultni konstruktor jako loš za nas jer ne inicijalizira varijable, nego u njih stavlja smeće. Korisnički definiran konstruktor će nam biti potreban kada budemo morali baratati s klasom.

Super! Naš header je sada spreman, te možemo preći na definiranje.

4. KORAK - Definiranje funkcija, stvaranje potrebnih konstruktora

```cpp
#include "Osoba.h"

Osoba::Osoba(string ime, string prezime, string mjesto, int godina_rodjenja)
{
	this->ime = ime;
	this->prezime = prezime;
	this->mjesto = mjesto;
	this->godina_rodjenja = godina_rodjenja;
}
```

Ovo nam je za sada jedina metoda/konstruktor koji se nalazi u Osoba.cpp fajlu. Detaljniji rad nad datotekama (zadatci) bit će objašnjeni na vježbama, te ćete tamo proći puno detaljnije primjere. Radi jednostavnosti ove skripte, ovdje će biti objašnjen samo osnovni rad nad fajlom i algoritam parsiranja.

Kada smo definirali naš konstruktor i sve dodatne metode koje su nam potrebne (ako postoje), možemo krenuti na naš source.cpp file.

5. KORAK - Pozivanje datoteke i provjera datoteke

Datoteku koju smo maloprije ubacili u naš projekt moramo nekako i pozvati i provjeriti da li ga program može pronaći. To radimo u mainu na način kojim smo radili u prvom semestru na programiranju. Ovaj dio koda je uvijek isti:

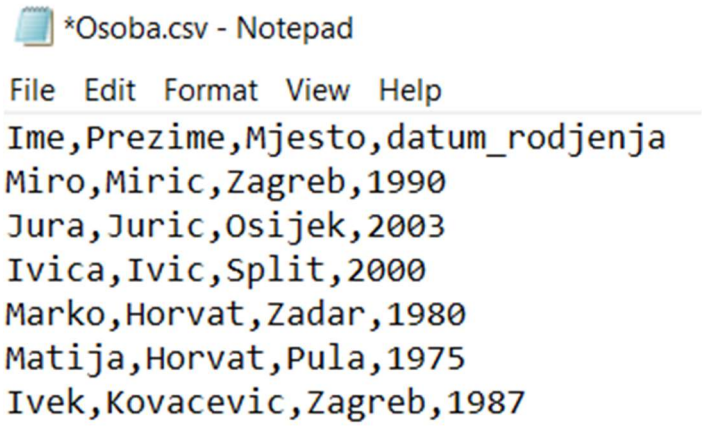
```
ifstream in("Osoba.csv");

if (!in)
{
	cout << "Nope!" << endl;
	return 1;


	in.close();
}
```

Naravno, uz ovaj komad koda moramo i includati **<fstream>**.

6. KORAK - Parsiranje datoteke

Jedan od problema koji nastaje tijekom rada nad datotekama je sam rad na datoteci. Cilj parsiranja je riješiti se datoteke jer datoteka je sama po sebi nešto što mi čitamo i nema neku konkretnu svrhu kasnije (tekstualni file nije kod koji možemo iskoristiti).

Nadalje, vratimo se nazad na našu datoteku:



Naš glavni cilj sada je ovu datoteku nekako isjeći na linije, te svaku liniju još jednom isjeći na manje komade koje moramo kasnije spremiti u objekt. Manji komadi u ovom slučaju su podatci odvojeni zarezom.

Prvo pitanje koje se postavlja je to kako da mi ove linije spremimo u objekt? To ćemo napraviti tako da sve podatke spremimo (za sada) u vektor objekata. U ovom slučaju naš kod će izgledati ovako:

```
vector <Osoba> v;
```

Što ovo znači za nas? To znači da će svaka linija biti svoj objekt (svaka linija je jedna osoba), te ćemo tijekom parsiranja svaku osobu gurnuti u vektor.

Kada smo definirali vektor osoba možemo krenuti na drugi dio zadatka, a to je presjecanje datoteke na manje komade koje možemo kasnije koristiti.

Prvo pitanje koje se postavlja je to kako da povučemo string kao liniju? To radimo tako da definiramo sljedeće:

```
string line;
getline(in, line);
```

Što smo postigli s ovime? Naime, naš cilj je pročitati jednu liniju iz ulazne datoteke, a to radimo tako da definiramo getline koji čita podatke iz cin-a i ubacuje u line datoteku (Zapamtimo: getline čita isključivo do prvog entera). Super! Na ovaj način pročitali smo prvu liniju, tj. liniju s nazivima stupaca i spremili ju u našu line varijablu. A što ćemo sada s ostalim linijama? Vrlo jednostavno! Treba se prisjetiti da smo u prošlom semestru već čitali iz datoteka dok nismo došli do kraja preko while petlje. Ovaj puta proces izgleda malo drugačije nego prije?

Prisjetimo se getline-a koji smo koristili. Nevjerojatno je da getline ispod haube radi na totalno drugi način, a to je da **getline radi kao bool varijabla koja vraća true ili false**. Ovo nam može poslužiti za čitanje iz datoteke dok se ona ne isprazni. Naš cilj je da getline-om, dokle god on vrača true, čitamo iz datoteke dok ju ne isprazni i vrati false. To je nama zapravo jedna sretna okolnost jer getline možemo koristiti kao uvjet u while petlji. Za sada maknimo getline iz gornjeg koda i stavimo ga u while petlju:

```
while (getline(in, line)){
}
```

Sada možemo i provjeriti da li naša datoteka radi, a to ćemo napraviti tako da ispišemo sve u konzolu:

```
while (getline(in, line))
{

      cout << line << endl;

}
```

Rezultat je onakav kako je i rečeno:

Sada se postavlja pitanje, treba li nam prva linija? Ne treba. Te linije se možemo riješiti tako da jednu liniju pročitamo van while petlje, te će naš kod izgledati ovako:

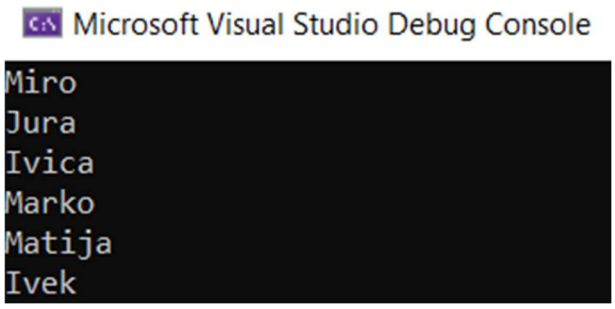
```
string line;
getline(in, line);

while (getline(in, line))
{
	// Miro,Miric,Zagreb,1990
	cout << line << endl;

}
```

Prisjetimo se da smo na početku rekli da liniju moramo isjeći na manje komande. Sada se postavlja pitanje kako napraviti to? To možemo napraviti preko getline-a, ali na žalost compiler ne dozvoljava to. Valja zapamtiti da getline ne čita stringove, on čita **stream-ove**. Prisjetimo se streamova koje smo do sada radili: konzola (iostream), file stream (fstream) i imamo stringstream (sstream). Prisjetimo se da smo na vježbama spajali stringove pomoću stringstreama. Istu stvar možemo napraviti i sada:

```
while (getline(in, line))
{
	string temp;
	stringstream ss(line);

	// Miro,Miric,Zagreb,1990

	// Miro
	getline(ss, temp, ',');

}
```

U gornjem kodu smo definirali temp varijablu tipa string koja će nam koristiti za privremeno spremanje isječene informacije. Uz njega, pozvali smo i stringstream kao konstruktor kojemu smo poslali liniju. To je upravo ono što nam omogučava spajanje stringova iz datoteke. Kako bi pročitali datoteku do prvog zareza, u getline-u smo definirali da se **string čita iz stringstream-a** u **temp varijablu** do **prvog zareza**. Vrlo je važno napomenuti da se treći uvjet definira jednostrukim navodnicima jer u suprotnom kod neće raditi.

Rezultat našeg koda možemo provjeriti ispisivanjem tempa u while petlji, a rezultat izgleda ovako:



Kao što se može vidjeti, uspješno smo isjekli ime naših osoba, a proces čitanja ostalih podataka je apsolutno isti i izgleda ovako:

```
while (getline(in, line))
{
	string temp;
	stringstream ss(line);

	// Miro,Miric,Zagreb,1990

	// Miro
	getline(ss, temp, ',');
	string ime = temp;

	// Miric
	getline(ss, temp, ',');
	string prezime = temp;

	// Zagreb
	getline(ss, temp, ',');
	string mjesto = temp;

}
```

U kodu možemo vidjeti da smo definirali nove varijable tipa string koje nam služe za spremanje iščitane temp varijable. Nakon toga nailazimo na jedan problem, a to je godina_rodjenja koju smo spomenuli prije.

Prisjetimo se da stringstream čita stringove, ne int varijable. Da bi mogli prikazati i spremiti godinu rođenja, naš string moramo nekako konvertirati u int. To možemo napraviti preko posebne funkcije:

```
getline(ss, temp, ',');
int datum_rodjenja = convert(temp);
```

Sada napravimo naš konverter na tradicionalan način:

```
int convert(string& s)
{
	int n;
	stringstream c(s);
	c >> n;
	return n;
}
```

U ovom kodu definirali smo funkciju convert kojoj šaljemo referencu na varijablu temp, naziva string. Unutar funkcije definirali smo n varijablu tipa int te pozvali stringstream naziva c koji prima temp varijablu (tj. naziv je “s” prema referenci). Taj stream smo na kraju samo poslali iz stringstream-a u n varijablu i vratili ga preko return-a nazad. Na ovaj način smo dobili godinu rodjenja u obliku int-a.

Sada dolazimo do teme koja je opcionalna ali vam izuzetno može olakšati posao i rješavanje ispita, a to je **template**. Template je jedna od opcija koja smanjuje broj funkcija za konvertiranje koji moramo napraviti (zapitajte se koliko funkcija (i koliko posla imamo) moramo imati ako u našoj datoteci imamo uz int i double tip podatka…). Da bi mogli napraviti template, u našem kodu iznad moramo definirati sljedeće:

```
template <typename T>
T convert(string& s)
{
	T n;
	stringstream c(s);
	c >> n;
	return n;
}
```

U kodu možemo vidjeti da smo pozvali template kojemu smo dali ime “T”. Umjesto da u funkciji vratimo int ili double, mi smo ovdje izmijenili kod i stavili “T” i uz to smo n varijabli dali tip podatka “T”. Što nam ovo omogučava? Ovo nam omogučava sljedeće:

```
while (getline(in, line))
{
	string temp;
	stringstream ss(line);

	// Miro,Miric,Zagreb,1990

	// Miro
	getline(ss, temp, ',');
	string ime = temp;

	// Miric
	getline(ss, temp, ',');
	string prezime = temp;

	// Zagreb
	getline(ss, temp, ',');
	string mjesto = temp;

	// 1990
	getline(ss, temp, ',');
	int datum_rodjenja = convert <int> (temp);

}
```

U kodu možemo vidjeti da smo pozivu funkcije convert dodali i tip podataka u koji želimo konvertirati naš stream. To smo napravili definiranjem <int>. Template možda izgleda malo teže nego tradicionalni način rada, ali vam isti može jako olakšati život jer mi sada u ovoj funkciji možemo konvertirati i tip double. Proces je isti, samo se u kodu umjesto <int> stavi <double>.

U suštini template nam dozvoljava da napravimo imaginarnu varijablu tipa “T” kojoj samo kasnije definiramo s kojim tipom podatka radimo.

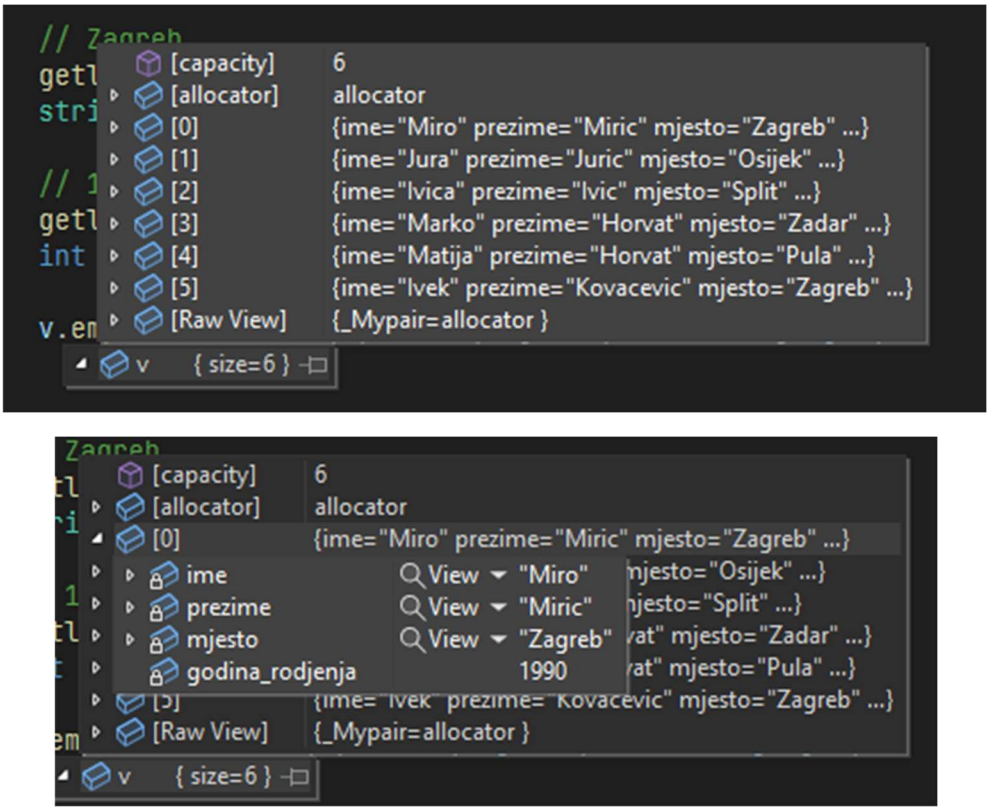
**Ponovna napomena da je template neobavezan i da se na ispitu prihvaća bilo koji način rada.

Kada smo sve ovo napravili ostala nam je još jedna stvar koju moramo napraviti, a to je da sve informacije koje smo parsirali spremimo u nešto. U ovom slučaju koristiti ćemo vektor koji smo definirali ranije, te ćemo upisati sljedeće u while petlju:

```
v.emplace_back(ime, prezime, mjesto, datum_rodjenja);
```

emplace_back() je jedna od funkcija koja je sastavni dio STL-a u kojemu radimo. On nam omogučava ubacivanje više elemenata u konstruktor koji smo ranije definirali. Valja napomenuti da compiler prepoznaje konstruktor koji smo definirali, te ga intellisense prikazuje. Ovo je još jedan od razloga zašto nam treba dobro definiran konstruktor za ovakve stvari. Rezultat ubacivanja preko emplace_back funkcije možemo provjeriti stavljanjem break point-a na in.close() funkciju, te nakon toga uđemo u debug.

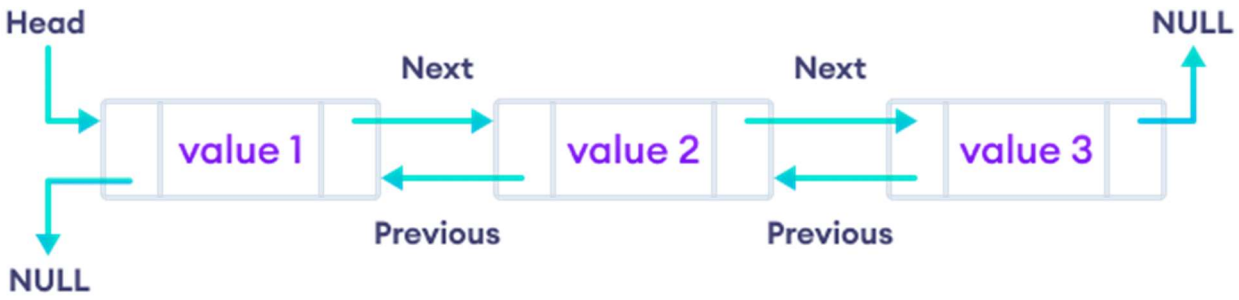
Rezultat izgleda ovako:



Ovime smo završili parsiranje naše datoteke, tako i naš prvi parser.

Dalje što nam ostaje je rad nad podatcima koji se nalaze u vektoru, a rad nad istima objašnjen je (ili će biti) na vježbama.

LIST & FORWARD LIST



C++ liste dolaze s dvije različite implementacije: List & Forward List. Razlike između dva navedena nisu vidljive na prvi pogled, ali postoje:

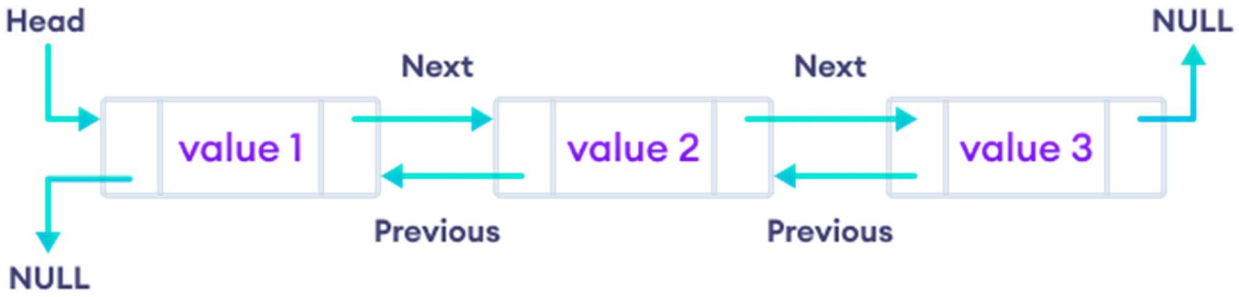
- **forward_list <T>** troši manje memorije po svakom čvoru jer sadrži samo pokazivač na sljedeći čvor
  - Važna stavka za okruženja s manjom količinom resursa (Arduino, …)
  - Omogućuje samo iteriranje (kretanje po listi) od početka prema kraju
- **list <T>** sadrži i pokazivač na prethodni element
  - Omogućuje i iteriranje (kretanje po listi) od kraja prema početku

Postoji nekoliko načina za kreiranje liste:

- **list <int> jedan;**
  - Kreira sasvim normalnu listu čiji elementi na početku nisu inicijalizirani.
- **list <int> dva(n);**
  - Kreira listu od n elemenata inicijaliziranih na defaultnu vrijednost (0).
- **list <int> tri(n, val);**
  - Kreira listu od n elemenata inicijaliziranih na val vrijednost
- **list <int> cetiri(iterator1, iterator2)**
  - Kreira listu u range-u zadanom pointerima.
- **lista <int> pet(tri);**
  - Kreira listu na temelju liste tri (kopira sve elemente iz liste tri u listu pet).
- **list <int> sest({1, 2, 3, 4, 5, 6});**
  - Direktna inicijalizacija liste nekim vrijednostima.

Mi smo do sada govorili o vektorima. Vektori su dobri samo za push_back i pop_back funkcije (iako nije niti to teoretski dobra stvar), te access i-tom elementu. Međutim, zamislimo da netko push-a i pop-a s početka? To je jako loše…

Riješenje ovog problema su liste i imamo ih dvije: List & Forward List.



Obije liste se, za razliku od vektora, sastoje od pointera prema sljedećem / prošlom elementu liste. Prema slici iznad bit će objašnjena oba primjera liste.

- **Forward lista (linked list)** je lista u kojoj možemo iterirati (kretati) samo unaprijed. Svaki element liste pokazuje na sljedeći element putem pointera. U FL ne postoje pointeri na prošli element liste.
- **Lista (double linked – list)** je nešto "skuplja" verzija FL-e. Razlika je isključivo u tome što lista ima pointere u oba smjera.

Liste su za nas dobre, jer se u slučaju brisanja neke vrijednosti iz liste samo ažurira pointer na prvi ili prvi i zadnji element liste, međutim liste su katastrofalne za pristup elementima. Zašto? Recimo da želimo pristupiti 900 elementu (referirajmo se na sliku) od njih 1000. Lista mora do tog elementa proći 899 ostalih elemenata prije nego dođe do traženog elementa. Iz ovoga možemo zaključiti da su liste katastrofalne za access.

Nadalje, da se riješimo teorijskog dijela, idemo prikazati kroz nekoliko primjera kako raditi na listi:

PRIMJER #1 – stvoriti listu s 10 elemenata i ispisati je

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

int main()
{

	list <int> l(10);

	for (auto it = l.begin(); it != l.end(); ++it)
	{
		cout << *it << " ";
	}

	return 0;
}
```

U ovom primjeru možemo vidjeti da za rad s listama moramo staviti #include <list> i #include <algorithm>. <algorithm> nam treba iz istog razloga zbog kojeg nam je trebao i za vektor/array.

Za kretanje po listi ne možemo koristiti tradicionalnu for petlju kao prije, te se moramo prisjetiti da liste rade s pointerima. To nas dovodi do for petlje u kojoj umjesto hodanja po index-u hodamo iteratorom od početka do kraja liste.

Iteriranje u listi radi na vrlo sličan način kao i kod regularne for petlje. Naime, for petlja započinje auto it dijelom s kojim smo definirali interator, te smo dodali da ćemo se kretati od početka liste (l.begin()), sve dok iterator ne bude jednak kraju liste (l.end()), te smo taj iterator povećali. Ova for petlja radi sasvim normalno:

Sada se postavlja pitanje, zašto koristimo ++it umjesto it++ za povećavanje vrijednosti? Odgovor je u tome što je ++it optimiziran. To znači da pri izvršavanju koda for petlji se ne šalje kopija vrijednosti već original vrijednosti + iza haube se mora izvršiti više operacija, što može usporiti cijeli kod.

PRIMJER #2 – listu ispisati unazad iteratorima, a zatim ispisati prvi i zadnji element liste

```
list <int> l2({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
for (auto it = l2.rbegin(); it != l2.rend(); ++it)
{
	cout << *it << " ";
}
```

U ovom primjeru možemo vidjeti dvije nove funkcije koje možemo koristiti i na vektorima, a to su rbegin i rend. Obije funkcije rade asolutno isto kao i one iz prošlog primjera, jedina je razlika što one iteriraju od nazad prema naprijed.

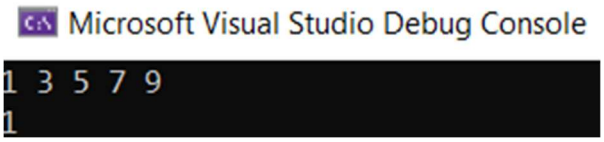
Nadalje, želimo li ispisati prvi i zadnji element liste, koristit ćemo sljedeće:

```
cout << l2.front() << endl;
cout << l2.back() << endl;
```

Ispis izgleda ovako:

PRIMJER #3 – ispis neparnih vrijednosti iz liste

```
list <int> l2({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
for (auto it = l2.begin(); it != l2.end(); advance(it, 2))
{
	cout << *it << " ";
}
```



U primjeru vidimo da se hodanje po listi može izvesti i bez zbrajanja iteratora. U ovom slučaju koristimo funkciju advance kojoj predajemo iterator i broj svakog elementa koji želimo ispisati.

PRIMJER #4 – ubacivanje 10 elemenata u forward listu + ispis iz forward liste

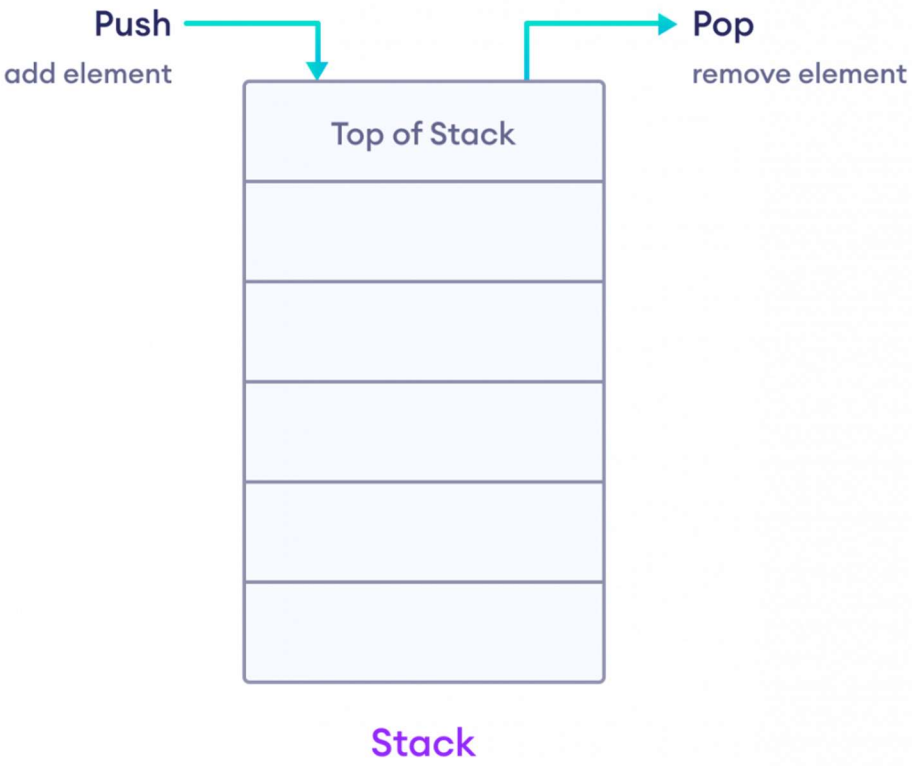
```
forward_list <int> fl;

for (int i = 0; i <= 10; i++)
{
	fl.push_front(i);
}

for (auto it = fl.begin(); it != fl.end(); ++it)
{
	cout << *it << " ";
}
```

Princip rada s forward listom je isti kao i s normalnom. Razlike se kreču događati onog trenutka kada pogledamo način guranja elemenata u listu. Primijetimo da fl nema push_back, nego push_front. S time potvrđujemo ideju o tome da fl radi u jednom smjeru s pointerima jer gura elemente isključivo na početak.

Pogledajmo sada razliku u ispisu:

Elementi nisu namjerno ispisani u nazad… Forward lista gura brojeve na početak, umjesto na kraj, što je i vidljivo na slici ispisa. Važno je napomenuti da smo prije govorili da je push_front skup. To vrijedi za vektore, ali ne i za liste.

STACK



ADT (Abstract Data Type) Stack je lista ili povezana lista uz neka ograničenja:

- Svako umetanje ili vađenje elemenata se vrši isključivo na vrhu (eng. top).
- Ispis elemenata se ne izvodi korištenjem klasične for petlje kao do sada.
- Podatci se ispisuju na sljedeće načine: **LIFO** (Last In First Out) ili **FILO** (First In Last Out)
    - To znači da lista ispisuje elemente na jedan potpuno interesantan način koji ćemo proći uskoro.

Stack je u programskom jeziku C++ implementiran kao klasa naziva stack <T>, gdje je T tip podatka. Stack ima mogućnost korištenja adaptive containera, a ti containeri moraju imati barem sljedeće metode:

- empty
- size
- back
- push_back
- pop_back

Klase koje mogu biti adaptive containeri su vector, list, deque (ne radimo ga u ovom ishodu).

Stack se izrađuje na iste načine kojima smo stvarali vektore i liste:

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

using namespace std;

int main()
{

	stack <int> s1;

	return 0;
}
```

Kao što smo spomenuli, stack može imati i adaptive container koji mu daje neke dodatne mogućnosti. U sljedećem primjeru bit će prikazano kako to izgleda tako što ćemo stacku dati vektor kao adaptive conatiner:

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

using namespace std;

int main()
{

	stack <int, vector<int>> s2;

	return 0;
}
```

U ovom kodu možemo vidjeti kako izgleda stack kojemu smo kao adaptive container dodjelili vektor. Važno je napomenuti da se includa **<vector>** na početku programa kako bi mogli pozvati vektor.

Osnovne funkcije koje moramo znati za rad nad stackom su sljedeće:

- s.push(val); ---> stavlja kopiju od val na vrh
- s.emplace(val1, val2, …); ---> kreira novi objekt na vrhu (treba nam #include <algorithm>)
- s.top(); ---> pozivom ove funkcije prikazujemo element koji se nalazi na vrhu
- s.pop(); ---> pozivom ove funkcije briše se **jedan element** s vrha stacka
- s.empty(); ---> vraća informaciju je li stack prazan
- s.size(); ---> vraća broj elemenata na stacku

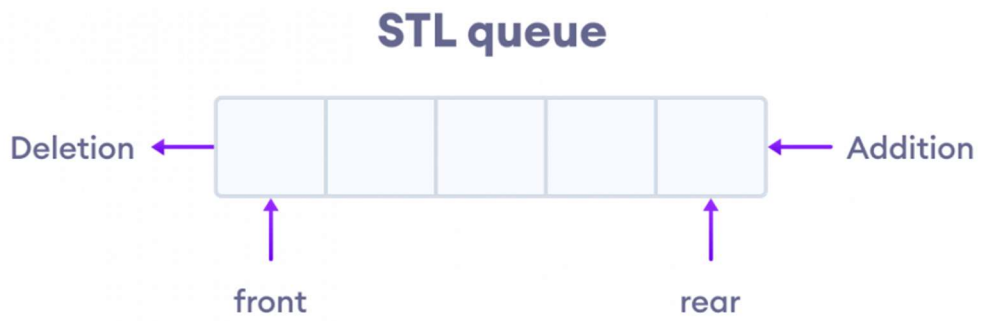
Nadalje, sada ćemo kroz jedan jednostavan primjer pokazati kako izgleda rad nad stackom:

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

using namespace std;

int main()
{

	stack <int> s;

	for (int i = 0; i <= 10; i++)
	{
		s.push(i);
	}

	while (!s.empty())
	{
		cout << s.top() << " ";
		s.pop();
	}

	return 0;
}
```

U sljedećem kodu vidimo da se stack puni na vrlo sličan način vektorima i listama. Razlika je u tome što stack ima funkciju **push()** koja je ujedno i jedina funkcija za ubacivanje elemenata u stack. Nadalje, da bi ispisali brojeve od 1 – 10 sa stacka ne možemo koristiti standardni način kao kod vektora jer stack ne radi na isti način. Prisjetimo se da smo u teorijskom djelu rekli da stack funkcionira na način **LIFO (last in first out)**, što nam govori da brojevi od 1 – 10 neće biti ispisani po redu. Provjerimo tu tezu ispisom brojeva od 1 - 10:

Ovaj primjer nam dokazuje da se ispis elemenata ne može vršiti klasičnim prolaženjem for petljom po indeksima jer stack ne koristi indekse. Jedini način ispisa svih elemenata je taj da preko funkcije while pozovemo funkciju koja provjerava je li stack prazan. Ako stack nije prazan, ispisuje element s vrha, te ga zatim briše.

QUEUE



Queue (red) je lista ili povezana lista koja ima dva ograničenja:

- Sva umetanja elemenata se rade na jednom kraju liste koji se onda naziva ulaz (engl. rear)
- Sva vađenja elemenata se rade na drugom kraju liste koji se onda naziva izlaz (engl. front)

Za razliku od stacka, queue funkcionira na sljedeće načine:

- **LILO** (Last In Last Out)
- **FIFO** (First In First Out)

Queue isto ima mogućnosti korištenja ostalih klasa kao adaptive containera.

Queue se u našem kodu prvo mora pozvati preko **#include <queue>**, a zatim isti u mainu zovemo s **queue <T>**, gdje je T tip podatka koji nam treba.

Funkcije za rad nad redom su vrlo slične onima od stacka:

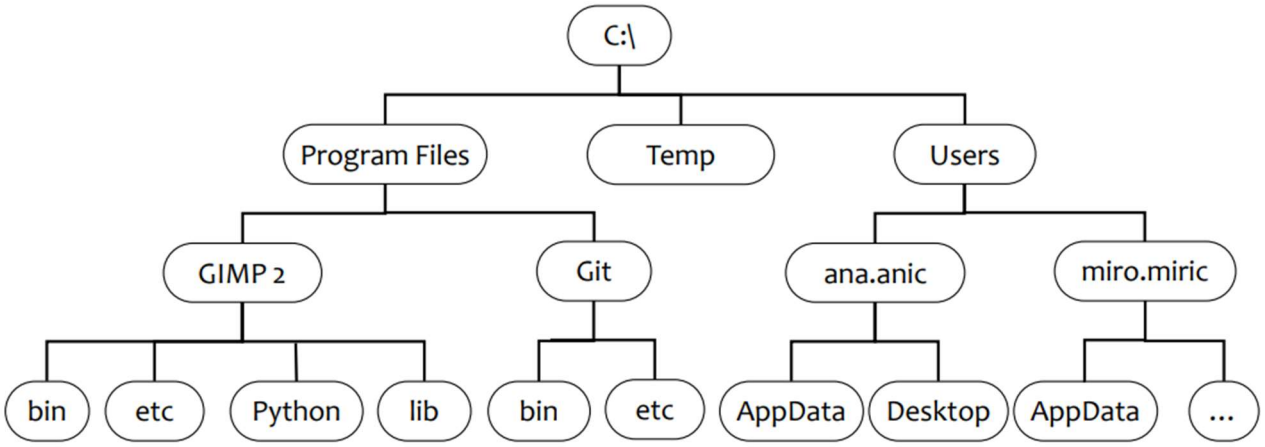
- q.empty(); ---> provjera je li queue prazan
- q.size(); ---> provjerava veličinu queue-a
- q.front(); ---> prikaz prvog elementa
- q.back(); ---> prikaz zadnjeg elementa
- q.push(val); ---> ubacivanje novog elementa u queue
- q.pop(); ---> brisanje elementa

Za primjer rada uzmimo prethodni primjer koji smo koristili za stack i promjenimo ga:

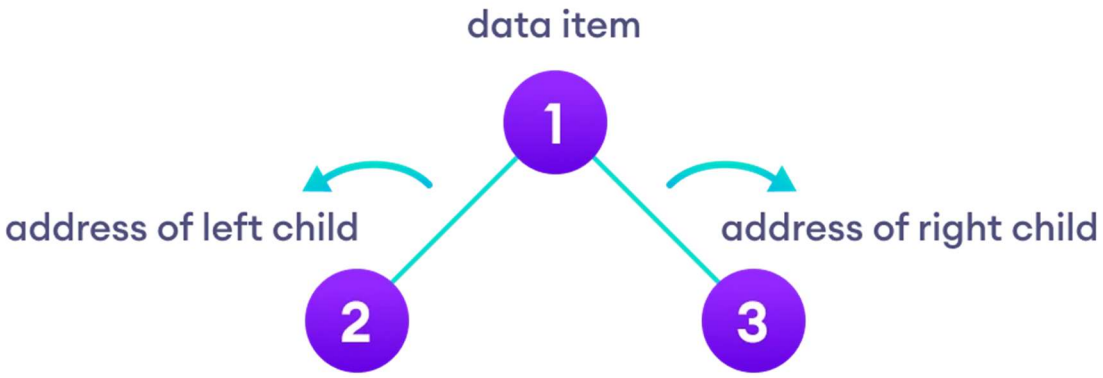
```cpp
#include <iostream>
#include <queue>

using namespace std;

int main()
{

	queue <int> q;

	for (int i = 0; i <= 10; i++)
	{
		q.push(i);
	}

	while (!q.empty())
	{
		cout << q.front() << " ";
		q.pop();
	}

	return 0;
}
```

Kao što se može vidjeti, queue funkcionira vrlo slično stacku. Razlika je u tome što pri ispisu elemenata ne koristimo **.top()** funkciju, nego **.front()** funkciju. Daljnji ispis elemenata se vrši na isti način kao i kod stacka, pozivom funkcije **.pop()** koja briše prvi element reda.

Kao što znamo, do sada smo koristili različite vrste podataka koje su bile linearno uređene poput vektora, liste, stoga, reda, itd. Sve je to super i sve zapravo radi, ali ako pokušamo u iste spremiti sljedeće podatke…



… zapravo ih ne možemo spremiti jer su ovi podatci sa slike hijerarhijske prirode. U svrhu toga, kako bi nam pomoglo, nastalo je binarno stablo (binary tree) čiji primjer možemo vidjeti ispod, a kasnije ćemo bolje objasniti isti.

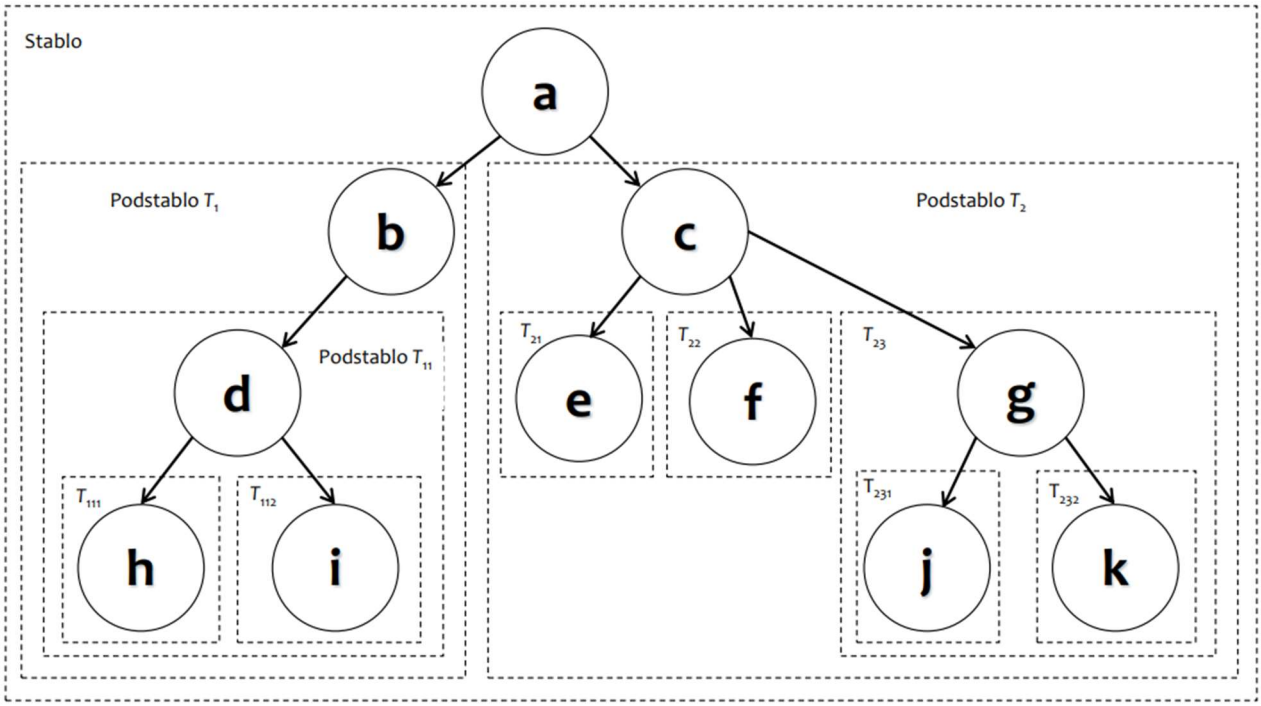


Stabla su nam općenito prikladna za sljedeće:

- Čuvanje hijerarhijskih podataka:
  - Obiteljsko stablo
  - Sportska natjecanja
  - Datotečni sustav
  - Organizacijska shema firme
  - Organizacijska shema vojske…
- Čuvanje podataka u obliku pogodnom za pretragu
  - Indexi u bazama podataka

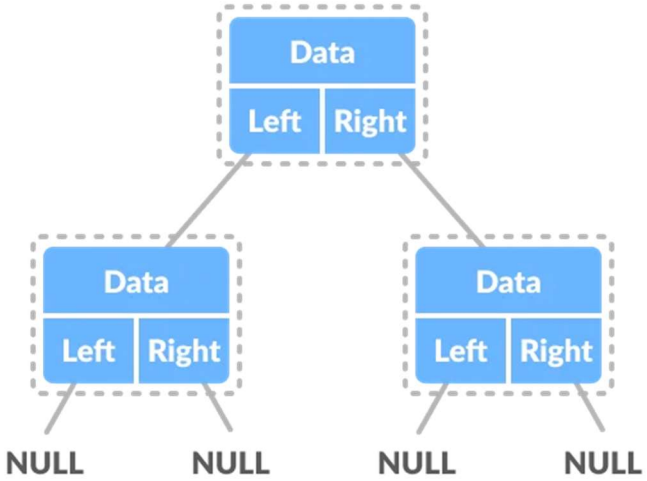
U suštini, binarno stablo je je skupina povezanih čvorova sa svojstvima:

- Svaki čvor (engl. node) sadrži jednu ili više vrijednosti.
- Čvorovi su hijerarhijski organizirani (roditelj – djeca).
- Postoji točno jedan čvor koji nema roditelja i koji se naziva korijen ili ishodište stabla (engl. tree root).
- Svaki čvor je ujedno i korijen podstabla (engl. subtree root), a to podstablo može biti složeno (sastavljeno od više čvorova) ili trivijalno (sastavljeno samo od 1 čvora).



Neki **najosnovniji pojmovi koje trebamo znati** o binarnim stablima:

- Čvorove koji se nalaze direktno ispod nekog čvora nazivamo njegovom **djecom** (engl. children)
  - Primjerice, čvorovi e, f i g su djeca čvora c
- Osim korijena stabla, **svaki čvor ima točno jednog roditelja** (engl. parent), **a to je čvor direktno iznad njega**
  - Primjerice, roditelj čvora h je čvor d
  - **Svaki čvor može imati više djece, ali najviše jednog roditelja**
- Čvorove s istim roditeljem nazivamo **braćom** (engl. siblings)
  - Primjerice, čvorovi e, f i g su braća



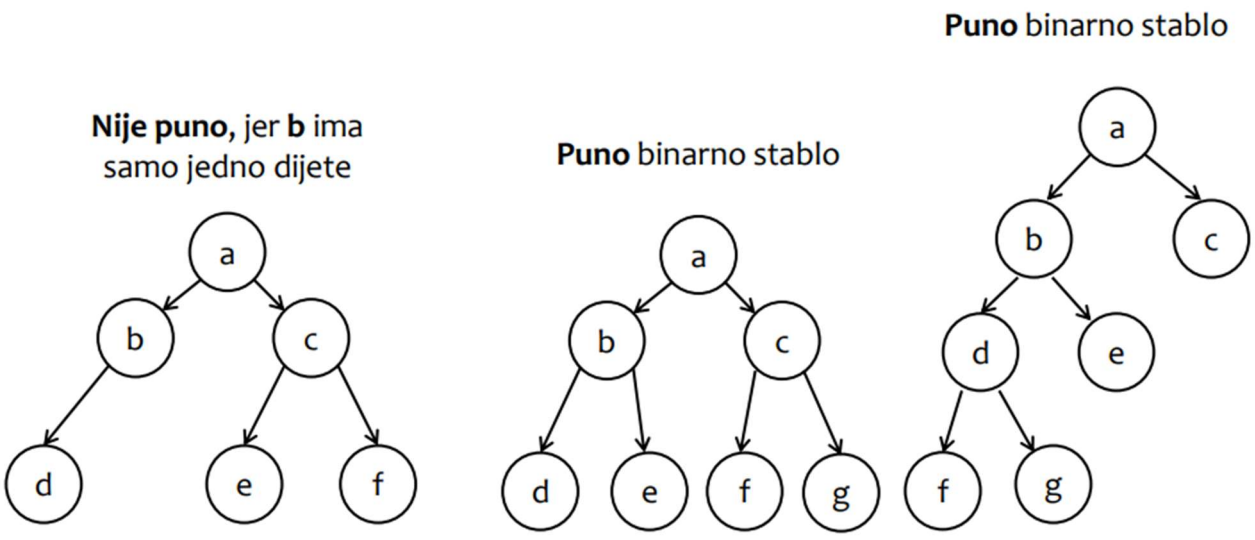
Kod binarnih stabala razlikujemo lijevo i desno dijete svakog čvora (sliak iznad). Bitno je napomenuti da ako neki node ima samo jedno dijete, nije svejedno da li je ono lijevo ili desno dijete.

Binarna stabla imaju i svoje tipove:

- **Puno (engl. full)** binarno stablo je ono u kojemu svaki čvor koji nije list ima točno 2 djeteta
- **Savršeno (engl. perfect)** binarno stablo je ono koje je puno i u kojem su svi listovi u istoj razini
- **Potpuno (engl. complete)** binarno stablo je ono u kojemu su sve razine (osim možda zadnje) popunjene, a zadnja razina ima sve čvorove popunjene s lijeva
    - To znači da se čvorovi dodaju u stablo na sljedeći način:
        - Krenemo od korijena i svaku razinu punimo s lijeva na desno dok ima mjesta
        - Kad više nema mjesta, prijeđemo na sljedeću razinu

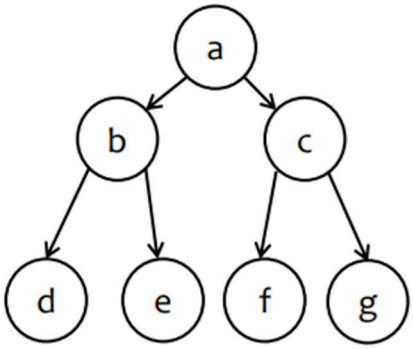
Ispod možemo vidjeti nekoliko primjera za svaku vrstu:

PRIMJER PUNOG BINARNOG STABLA

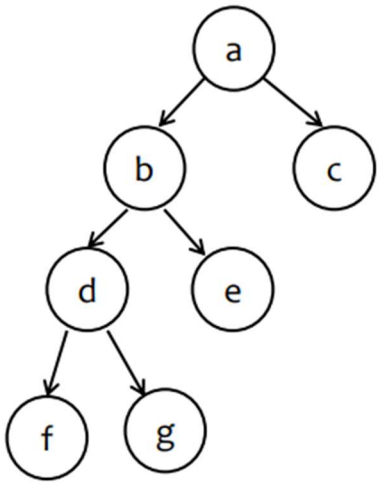


PRIMJER SAVRŠENOG BINARNOG STABLA

Savršeno binarno stablo

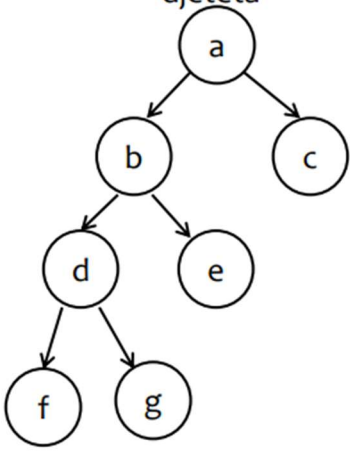


Puno, ali **nije savršeno** jer je razina listova **f** i **g** jednaka 3, lista **e** jednaka 2, a lista **c** jednaka 1

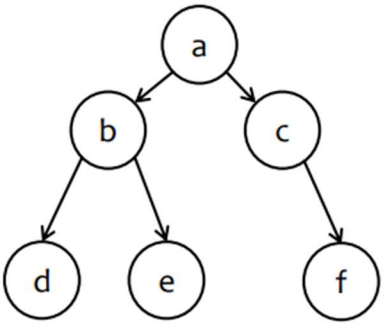


PRIMJER POTPUNOG BINARNOG STABLA

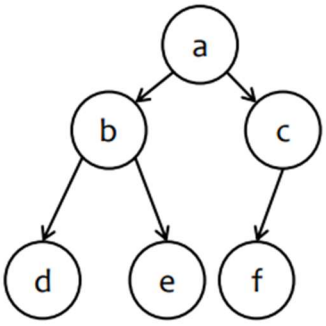
Puno, ali **nepotpuno**, jer **c** i **e** nisu u zadnjem nivou i nemaju oba djeteta



Nepotpuno, jer u zadnjoj razini nisu svi čvorovi popunjeni s lijeva



Potpuno binarno stablo



OBILAŽENJE BINARNOG STABLA

Obilazak stabla je postupak posjećivanja svih čvorova stabla uz uvjete:

- Posjetit ćemo svaki element u stablu
- Niti jedan element nećemo posjetiti dva ili više puta

Obilazak binarnih stabala je nešto složeniji nego klasično hodanje po vektoru, te postoji više načina obilaska po istom.

Najpoznatiji algoritmi obilaska binarnog stabla su:

- **DFS** algoritmi (eng. **Depth – First Search**):
  - **INORDER**
    - Princip obilaska: Left, Parent, Right
    - Obilazak kreće od lijevog podstabla pa ide na roditelja pa zatim na desno podstablo. Pri tome se svako podstablo obilazi na jednak način (rekurzivnost).
    - INORDER najčešće koristimo u binarnim stablima traženja (BST – Binary Search Tree)
  - **PREORDER**
    - Princip obilaska: Parent, Left, Right
    - Obilazak kreće od roditelja pa ide na lijevo i zatim na desno podstablo. Pri tome se svako podstablo obilazi na jednak način (rekurzivnost).
    - PREORDER najčešće koristimo za dupliciranje stabala jer pvo obradi roditelja, a tek onda djecu.
  - **POSTORDER**
    - Princip obilaska: Left, Right, Parent
    - Obilazak kreće od lijevog podstabla pa ide na desno podstablo i na kraju na roditelja. Pri tome se svako podstablo obilazi na jednak način (rekurzivnost).
    - POSTORDER se najčešće koristi za brisanje čvorova i uništenju stabla jer prvo obradi djecu, a tek onda roditelja.
- **BFS** algoritam (eng. **Breadth - First Search**)
  - BFS algoritam posjećuje čvorove razinu po razinu, s lijeva na desno:
    1. **Dodaj korijen u red**
    2. **Uzmi sljedeći element A iz reda i ispiši vrijednost***
    3. **Dodaj djecu čvora A u red**
    4. **Ako red nije prazan, idi na korak 2**
  - Primjerice, ako stablo prikazuje hijerarhiju, onda ovaj način obilaska prvo ispisuje one pri vrhu hijerarhije
  - Primjena ovog algoritma nije tako česta u praksi

IMPLEMENTACIJA STABLA

STL, na žalost, ne sadrži konkretnu implementaciju binarnog stabla, a za naše potrebe biti će nam dovoljni kodovi na sljedećim stranicama.

*****

Napominjem da će kasnije biti objašnjen sljedeći kod, a na ispitu ćemo imati gotov kod za kreiranje binarnog stabla, te nećemo previše ulaziti u to što se događa iza koda.

*****

HEADER

```
#ifndef _BINARNO_STABLO_H_
#define _BINARNO_STABLO_H_

#include <string>
using namespace std;

struct node
{
	string element;
	node* left_child;
	node* right_child;
};

class btree
{
private:
	node* root_node;
	node* create_new_node(string element);

public:
	btree(string element);
	void insert_left(node* parent, string element);
	void insert_right(node* parent, string element);
	node* root();
	node* get_left_child(node* parent);
	node* get_right_child(node* parent);
};

#endif
```

.CPP FILE

```
#include "btree.h"
#include <iostream>
using namespace std;

/*************** PRIVATNE METODE ***************/

node* btree::create_new_node(string element)
{
	node* novi = new node;
	novi->element = element;
	novi->left_child = nullptr;
	novi->right_child = nullptr;

	return novi;
}

/*************** JAVNE METODE ***************/

btree::btree(string element)
{
	root_node = create_new_node(element);
}

void btree::insert_left(node* parent, string element)
{
	parent->left_child = create_new_node(element);
}

void btree::insert_right(node* parent, string element)
{
	parent->right_child = create_new_node(element);
}

node* btree::root()
{
	return root_node;
}

node* btree::get_left_child(node* parent)
{
	return parent->left_child;
}

node* btree::get_right_child(node* parent)
{
	return parent->right_child;
}
```