# ALGORITMI I STRUKTURE PODATAKA

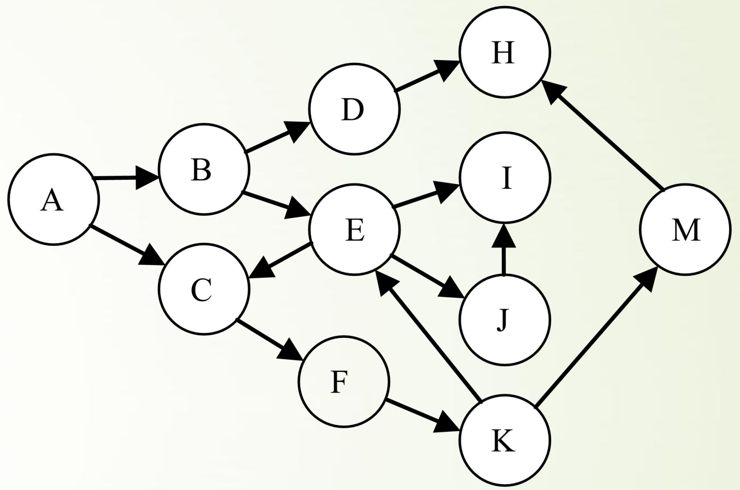
RAČUNSKE VEŽBE – TERMIN BR. 10 – GRAFOVI

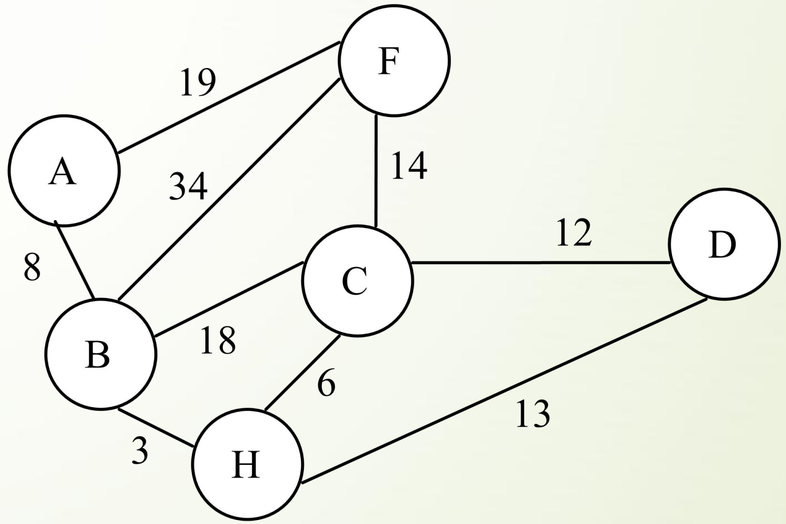
ALDINA AVDIĆ, DIPL. INŽ. – apljaskovic@np.ac.rs

RAČUNARSKA TEHNIKA, SOFTVERSKO INŽENJERSTVO, INFORMATIKA I MATEMATIKA

# Vrste grafova

- Težinski
- Neusmereni
- Usmereni



# 1. Obilazak grafa po širini (breath first)

- Na datom grafu ilustrovati:
  - obilazak grafa po širini
  - obilazak grafa po dubini
- OBILAZAK GRAFA:
  - svaki čvor grafa poseti na sistematičan način
  - samo po jednom izvrši neka obrada nad njim
- Graf nema neki poseban čvor od kojeg prirodno započinje obilazak
  - eksplicitno se zadaje kao argument operacije
  - slučajno se bira
  - Isti algoritam obilaska daje različite poretke čvorovazavisno od izbora početnog čvora.

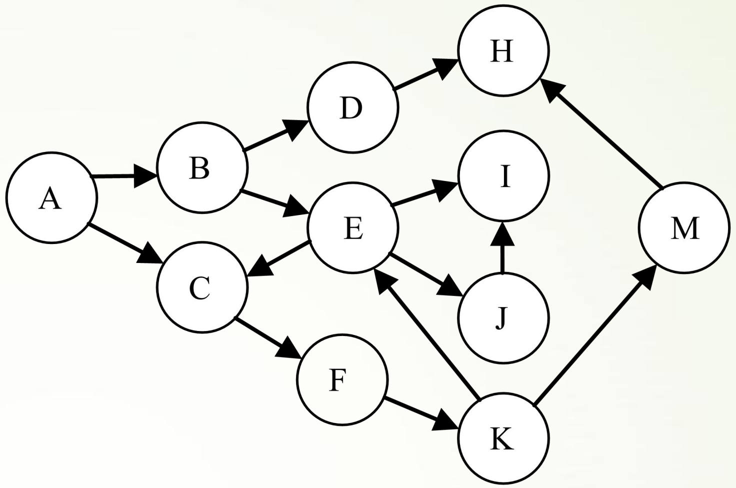
# 1. Obilazak po širini (breath first)

```
BFS(G, v)
for i= 1 tondo
    visit[i] = false
end_for
visit[v] = true
POSETA(v)
INSERT(Q, v)
while(not QUEUE-EMPTY(Q)) do
    v= DELETE(Q)
    for {u: (v, u)    E} do
        if (not visit[u]) then
            visit[u] = true
            POSETA(u)
            INSERT(Q, u)
        end_if
    end_for
end_while
```

- Od pomoćnih struktura, koriste se:
  neprioritetan red (FIFO)
  vektor *visit*
- Uloga vektora *visit* je da spreči višestruke obilaske nekog čvora
- Posećivanje čvora *u*se implementira procedurom POSETA(*u*)

# 1. Obilazak po širini (breath first)






| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | | | | | | | | | | |
| A | **B** | **C** | | | | | | | | |
| A | B | C | **D** | **E** | | | | | | |
| A | B | C | D | E | **F** | | | | | |
| A | B | C | D | E | F | **H** | | | | |
| A | B | C | D | E | F | H | **I** | **J** | | |
| A | B | C | D | E | F | H | I | J | **K** | |
| A | B | C | D | E | F | H | I | J | K | |
| A | B | C | D | E | F | H | I | J | K | |
| A | B | C | D | E | F | H | I | J | K | |
| A | B | C | D | E | F | H | I | J | K | **M** |

- Koristi se pomoćna struktura Red
- Pretpostavlja se smer (redosled)
obilaska suseda u alfabetskom poretku
njihovih oznaka.
- To znači, posetimo A, stavimo A u red
- Pošto red nije prazan, u red upisujemo čvorove
sa kojima ima zajedničku granu (B i C)
- A brišemo iz reda i štampamo na izlaz
- Ponavljamo ovo sada za B jer se red tako obrađuje,
ko je prvi ušao, prvi izađe i sve tako dok ne posetimo sve čvorove
- Rešenje: **ABCDEFHIJKM**

# 1. Obilazak grafa po dubini (depth first)



Obilazak po dubini: generalizacija *preorder* obilaska stabla

**Rekurzivna** realizacija obilaska po dubini

```
DFS-VISIT_R(v)
visit[v] = true
POSETA(v)
for { u, (v, u) ∈ E } do
  if (not visit[u]) then DFS-
   VISIT_R(u)
  end_if
end_for
```

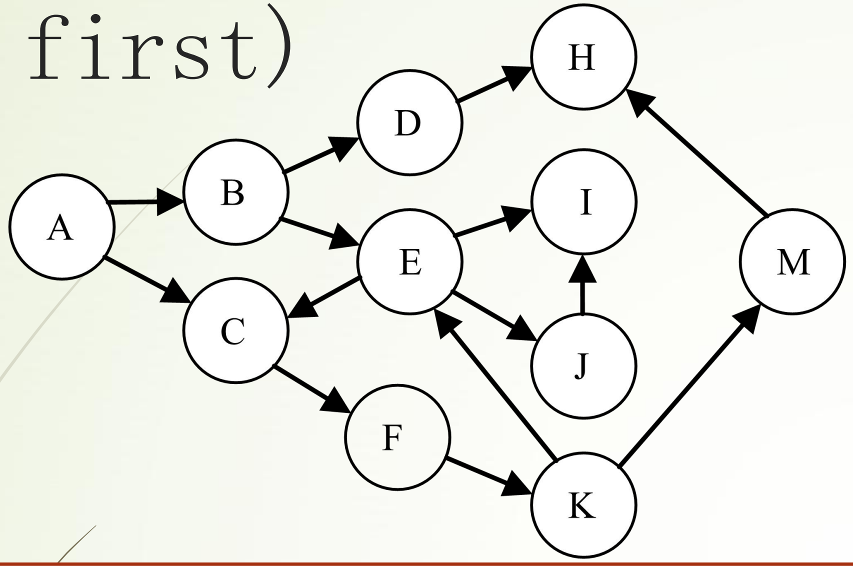
Potprogram DFS inicijalizuje vektor *visit*

```
DFS(G, v)
for i = 1 to n do
   visit[i] = false
end_for
DFS-VISIT_R(v) ili DFS-VISIT_I(v)
```

**Iterativna** realizacija obilaska po dubini

```
DFS-VISIT_I(v)
PUSH(S, v)
while not EMPTY(S) do
   v = POP(S)
   if (not visit[v]) then
      visit[v] = true
      POSETA(v)
      for { u, (v, u) ∈ E }
   do
    if (not visit[u]) then
       PUSH(S, u)
    end_if
      end_for
    end_if
end_while
```

# 1. Obilazak grafa po dubini (depth first)




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | | | | | | | | | | |
| **B** | **C** | | | | | | | | | |
| B | **F** | | | | | | | | | |
| B | **K** | | | | | | | | | |
| B | **E** | **M** | | | | | | | | |
| B | E | **H** | | | | | | | | |
| B | E | | | | | | | | | |
| B | **I** | **J** | | | | | | | | |
| B | I | | | | | | | | | |
| B | | | | | | | | | | |
| D | | | | | | | | | | |

- Pretpostavlja se smer (redosled) obilaska suseda u alfabetskom poretku njihovih oznaka.
- Koristi se **iterativni** algoritam.
- Pomoćna struktura stek
- To znači, stavimo A u stek, posetimo,

Izbacimo A iz steka, a dodamo one čvorove sa kojima ima zajedničku granu redom, znači B, pa C.

- **Pošto se sad radi o steku, sada to isto ponavljamo za element na vrhu steka a to je C**





Rešenje: **ACFKMHEJIBD**

# MST (Minimalno obuhvatno stablo)

- Za dati graf naći **minimalno** obuhvatno stablo:
  - a) upotrebom Prim-ovog algoritma
  - b) upotrebom Kruskal-ovog algoritma
- Obuhvatno stablo (*spanning tree*) grafa je stablo koje sadrži sve čvorove grafa
- sadrži neke grane grafa tako da se povežu svi čvoroviali da se ne formiraju ciklusi
- Posledica: između svaka dva čvora postoji tačno jedan put
- Obuhvatna stabla se generišu algoritmima za obilazak grafapo širini ili dubini
- Za isti graf može postojati više obuhvatnih stabala

# MST (Minimalno obuhvatno stablo)

- Minimalno obuhvatno stablo (MST – *minimum cost spanning tree*)
  - svaka grana ima cenu (težinu)
  - cena obuhvatnog stabla je suma cena grana
  - minimalno obuhvatno stablo ima najmanju cenu
  - može biti više takvih stabala (ista cena)
- Najpoznatiji algoritmi
  - Prim-ov
  - Kruskal-ov

# Primov algoritam

Algoritam radi inkrementalno, počevši od proizvoljnog čvora (s) koji postaje koren stabla.

PRIM($G$, $s$)

$U$ = {$s$}

$E'$ = $\emptyset$

**while** ($U \neq V$) **do**

find ($u$, $v$) $\Rightarrow$ min {$w(u, v)$ : ($u \in U$) and ($v \in (V - U)$) }

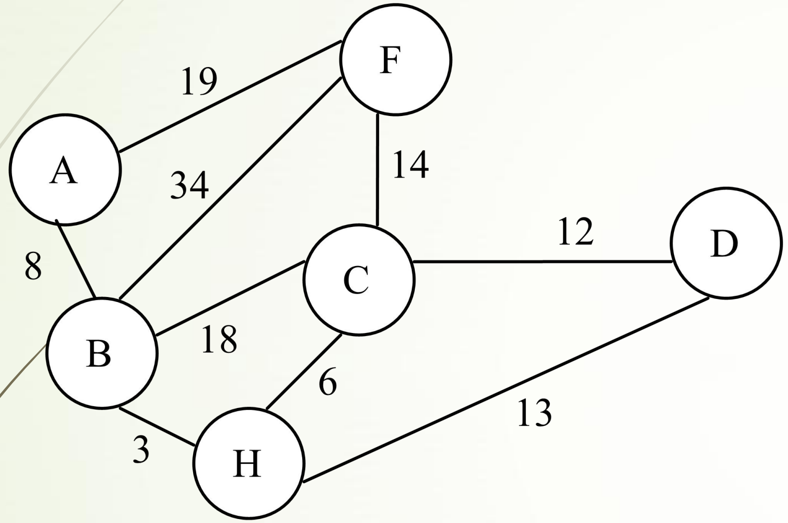
$U = U + \{v\}$

$E' = E' + \{(u, v)\}$

**end_while**

$MST = (U, E')$

# Primov algoritam




A-B

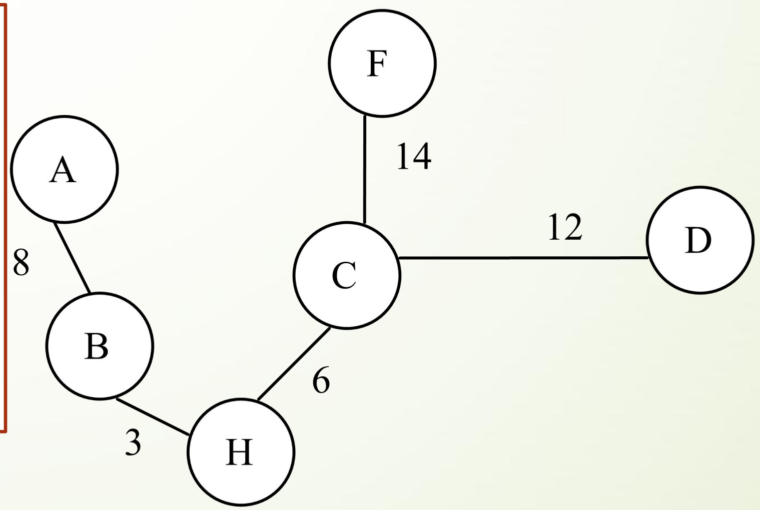
A-B, B-H

A-B, B-H, H-C

A-B, B-H, H-C, C-D

A-B, B-H, H-C, C-D, C-F

- Čvor A je odabran za koren MST stabla
- Zatim u MST se unosi grana sa najmanjom težinom, i sve tako dok ne budemo uneli grane tako da u MST postoji svaki čvor iz grafa a da pri tom ne pravimo petlje



# 2. Kruskalov algoritam

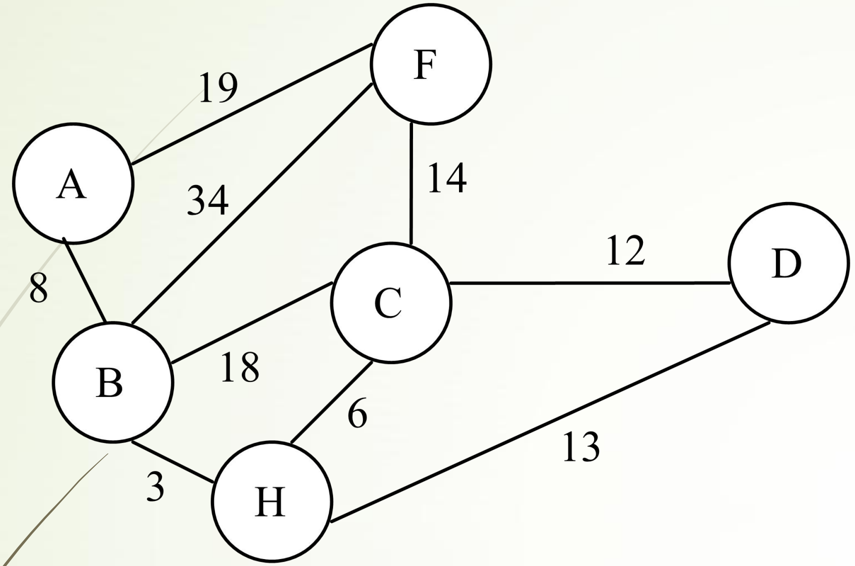


KRUSKAL($G$)
$E' = \varnothing$
**for** each $(u, v) \in E$ **do**
    PQ-INSERT($PQ$, $w(u, v)$)
**end_for**
$num = 0$
**while** ($num < n - 1$) **do**
    $w(u,v)$ = PQ-MIN-DELETE($PQ$)
    **if** (($u \in T_i$) and ($v \in T_j$) and ($i \neq j$)) **then**
        $E' = E' + \{(u, v)\}$
        $T_k = T_i + T_j$
        $num = num + 1$
    **end_if**
**end_while**
$MST = (V, E')$

- Inicijalno, graf se posmatra kao potpuno nepovezan (nepovezane komponente)
- Skup grana E se uređuje po neopadajućoj težini (prioritetni red, sortiran niz...)
- Nova grana se dodaje samo ako spaja dve odvojene komponente (T)

# 2. Kruskalov algoritam





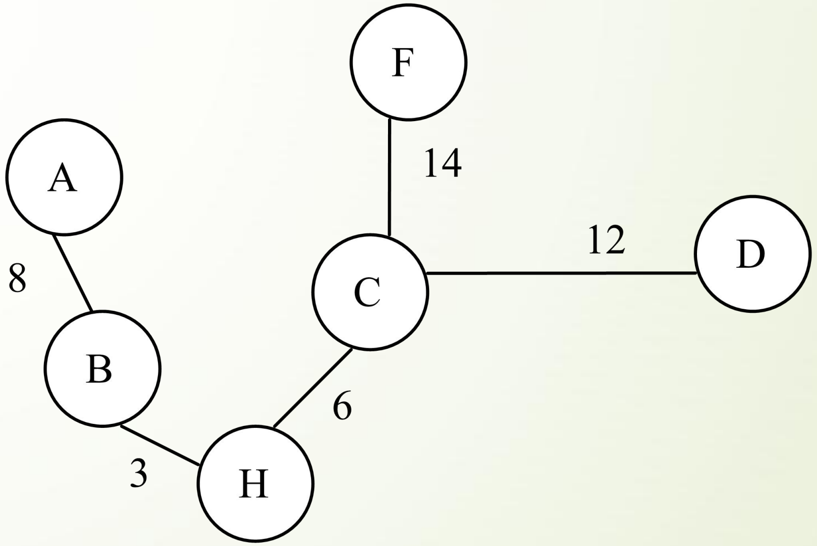

H-B

H-B, H-C

H-B, H-C, B-A

H-B, H-C, B-A, C-D

H-B, H-C, B-A, C-D, C-F

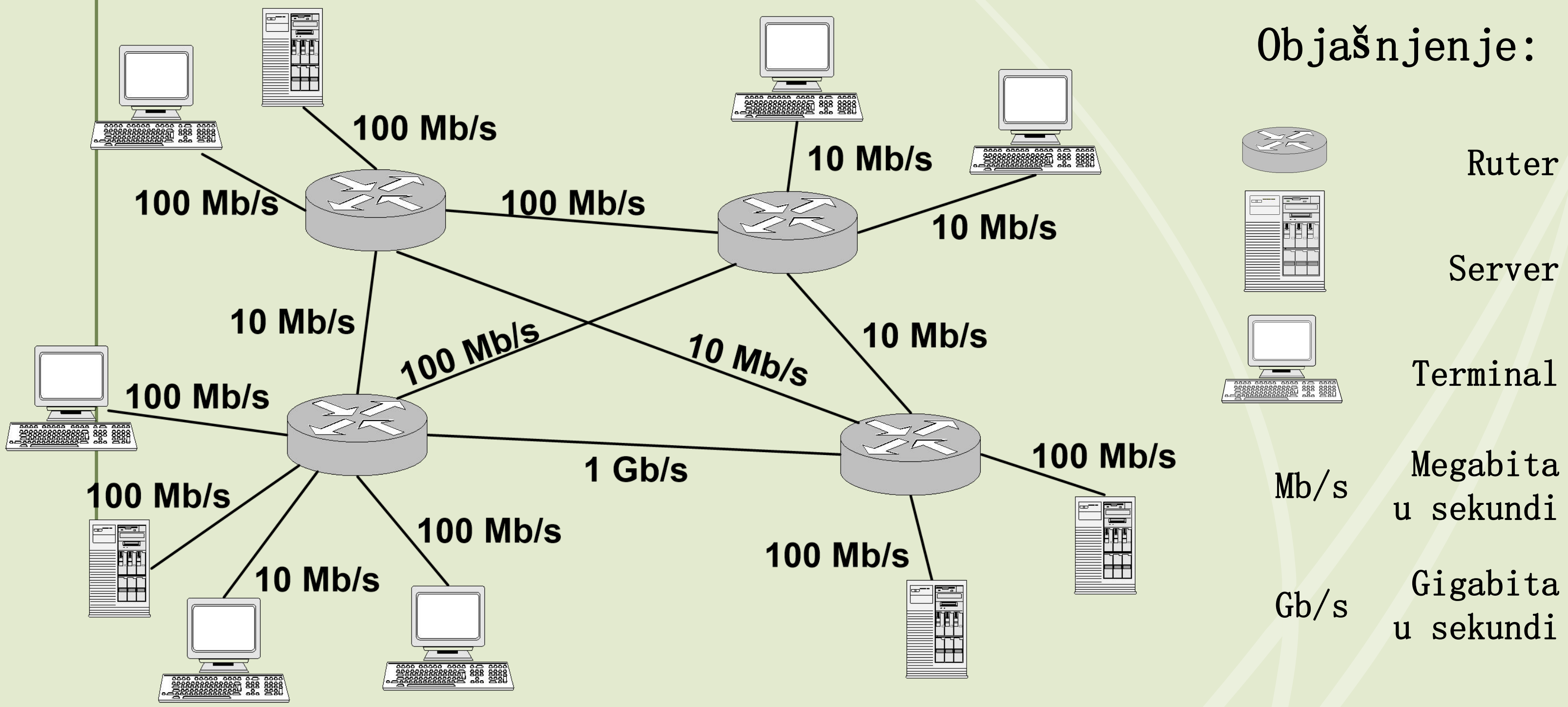



Od korena stabla mora da polazi
grana najmanje težine. Ovde se
bira čvor H. Proces se nastavlja unošenjem
Grana najmanje težine sve dok ne obuhvatimo
Sve čvorove, a da nema ciklusa

14

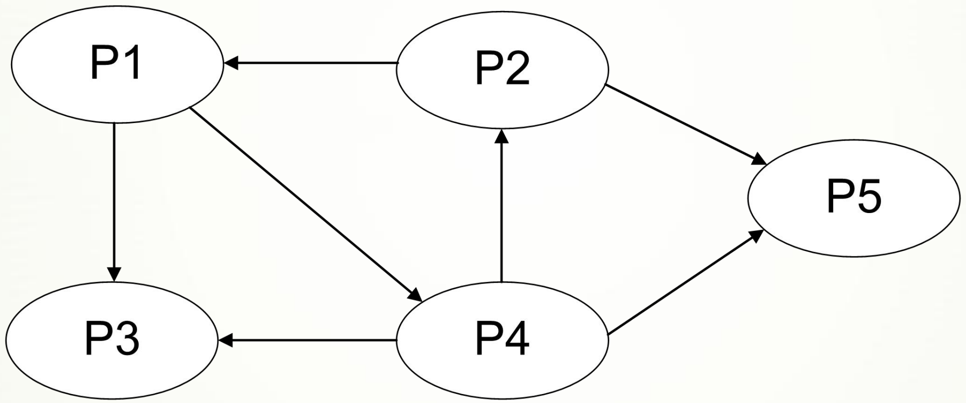
# 3. Primena MST

Na slici je prikazana šema jedne računarske mreže, koja se sastoji od **servera**, **terminala** i **rutera**. Uloga **rutera** je da prosleđuju komunikaciju između **servera** i **terminala** preko **najbržih** veza. Pri tome, oni obezbeđuju da između **bilo koja dva** uređaja (rutera, servera ili terminala) postoji **tačno jedan put**.



# 4. Određivanje matrice puta (Warshall-ov algoritam)

Programski sistem se sastoji od programskih modula P1, P2, P3, P4, P5. Ovaj sistem je predstavljen datim usmerenim grafom u kome su čvorovi moduli, a grane pozivi između njih, tako da grana (i, j) odgovara pozivu modula $P_j$, od strane modula $P_i$. Odrediti koji su moduli rekurzivni.

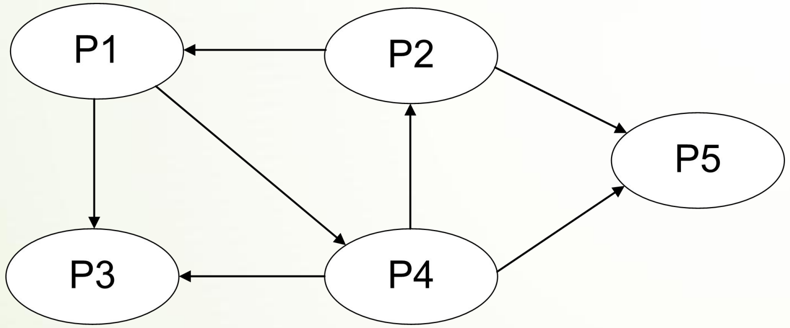



- Rekurzija može biti direktna ili indirektna:
  - direktna: modul poziva sam sebe
  - indirektna: modul A poziva modul B koji poziva modul A
- Pojava ciklusa u grafu ukazuje na postojanje rekurzije

# 4. Određivanje matrice puta (Warshall-ov algoritam)



- *Matrica puta* pokazuje međusobnu povezanost čvorova
  - p[i, j] = 1   ako postoji put od čvora *i* do *j*
  - p[i, j] = 0   u suprotnom
- Ponekad se naziva *matrica dostižnosti* (reachability)




$$p = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

# Zadatak 4 – određivanje matrice puta (Warshall)

- Matrica puta se dobija primenom *Warshall*-ovog algoritma
  - polazi se od matrice eksplicitno zadatih puteva (putevi dužine 1)
  - u svakom koraku *k* se u tekućoj matrici p[i,j] dodaju putevi koji se mogu ostvariti preko čvora *k*

```
WARSHALL
P=A
for k=1 to n do
    for i=1 to n do
        for j=1 to n do
            p[i,j]=p[i,j] or (p[i,k]
        and p[k,j])
        end_for
    end_for
end_for
```

Optimizacija: u najugnježdenijoj petlji je

$p[i,k]$=const, a $p[i,j]$ se neće menjati ako

je p$[i,k]$=0. Zato ćemo najugnježdeniju

petlju izvršavati samo ako je $p[i,k]$=1

```
if (p[i,k]=1) then
    for j=1 to n do
        p[i,j]=p[i,j] or p[k,j]
    end_for
end_if
```

# Zadatak 4 – određivanje matrice puta (Warshall)

$$p = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad p^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Polazna matrica susednosti

Matrica dostižnosti preko čvora 1

# Zadatak 4 – određivanje matrice puta (Warshall)

$$p^{(2)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrica dostižnosti preko čvorova 1 i 2

$$p^{(3)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrica dostižnosti preko čvorova 1, 2 i 3

Primetiti da je $p^{(2)}=p^{(3)}$.

# Zadatak 4 - određivanje matrice puta (Warshall)

$$p^{(4)} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Matrica dostižnosti preko čvorova 1, 2, 3 i 4

Primetiti da je $p^{(4)}=p^{(5)}$.

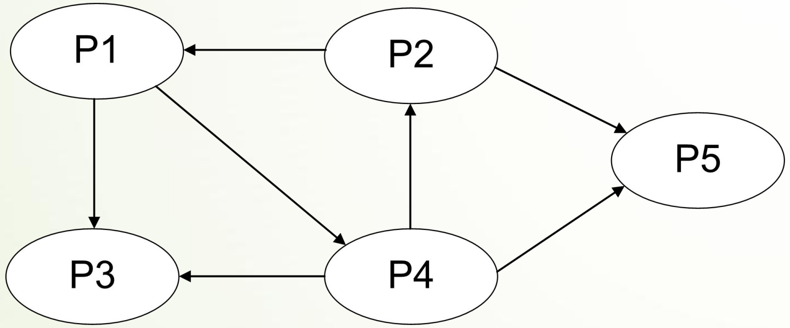
$$p^{(5)} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Konačna matrica dostižnosti (preko svih čvorova)

Rekurzivni su moduli 1, 2 i 4

# Zadatak 4 – određivanje matrice puta

- Drugi način:
  - Polazimo od reprezentacije grafa u vidu matrice susednosti
  - $p[i, j] = 1 \rightarrow$ postoji put dužine 1 između $i$ i $j$ (direktan put)
  - Obeležimo matricu sa




$$p^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

# Zadatak 4 – određivanje matrice puta

- Određivanje puta dužine **tačno** 2
  - za (*i*, *j*) pronalazi se čvor *k*, tako da postoje putevi (*i*, *k*) i (*k*, *j*)
  - put postoji ako p[i,k] * p[k,j] = 1
  - *i* i *j* su fiksni → p[i,k] * p[k,j] je **logički proizvod** vrste i kolone matrice
  - → matrica se množi sama sa sobom
- Put dužine **tačno** 3 : matrica se diže na treći stepen
- Od interesa je isključivo da li put postoji, ne koliko putanja ima

$$p^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad p^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

# Zadatak 4 – određivanje matrice puta

$$p^{(4)} = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$p^{(5)} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Nema potrebe da se računa za stepen veći od n, jer bi se napravio ciklus.
Matrica povezanosti dobija se kao logička suma svih izračunatih matrica.

$$p = p^{(1)} + p^{(2)} + p^{(3)} + p^{(4)} + p^{(5)}$$

Složenost celog postupka je $O(n^4)$

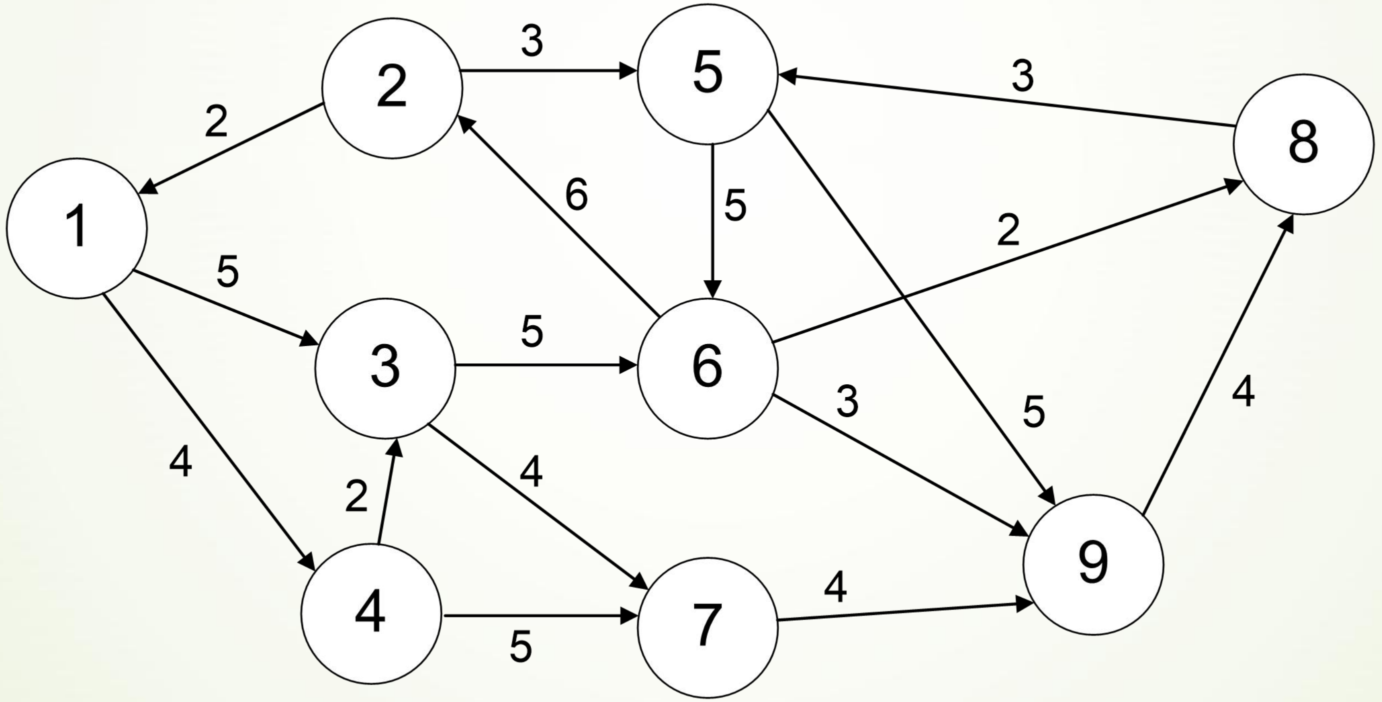
$$p = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Rekurzivni su moduli 1, 2 i 4

# 5. Dijkstra-in algoritam



Kakva je namena *Dijkstra* algoritma?
Za graf sa slike naći najkraće puteve od čvora 6 do ostalih čvorova primenom *Dijkstra*-inog algoritma.
Dati izgled vektora najkraćih rastojanja i vektora prethodnika posle svake iteracije.



# 5. Dijkstra-in algoritam

Dijkstra: Određivanje najkraćih puteva od jednog čvora grafa do svih ostalih

```
DIJKSTRA(W)
S = {x}
for i = 1 to n, i≠x do
    d[i] = w[x, i]
    if (w[x, i] ≠ ∞) then t[i] = x
    else   t[i] = 0
    end_if
end_for
for k = 1 to n - 1 do
    find min {d[i] : i ∈ (V – S) }
    if (d[i] = ∞) break
    S = S + {i}
    for each j ∈ (V – S) do
        if (d[i] + w[i, j] < d[j]) then
                d[j] = d[i] + w[i, j]
                t[j] = i
        end_if
    end_for
end_for
```

Ulaz predstavlja matrica težina *W*.

Izlaz čine:

- vektor d (dužina n-1)
- vektor *t* (dužina n-1)

gde je:

- *d*[*i*] najkraće rastojanje od polaznog čvora x do čvora *i*,
- *t*[*i*] je čvor-prethodnik čvora *i* na najkraćem putu od čvora x

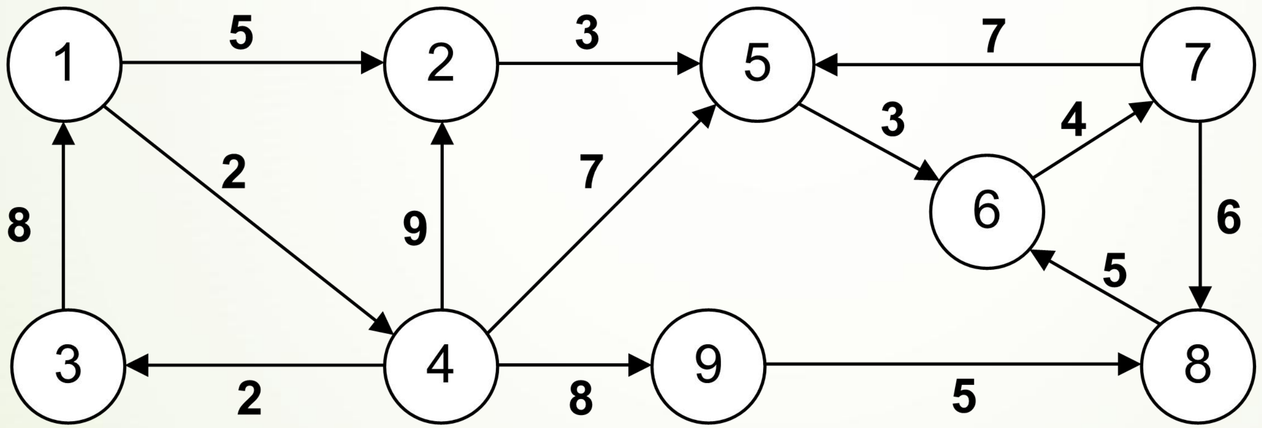
# 5. Dijkstra-in algoritam

| S | | D | | | | | | | | T | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 | 1 | 2 | 3 | 4 | 5 | 7 | 8 | 9 |
| 6 | - | ∞ | 6 | ∞ | ∞ | ∞ | ∞ | **2** | 3 | 0 | 6 | 0 | 0 | 0 | 0 | 6 | 6 |
| 6,8 | 8 | ∞ | 6 | ∞ | ∞ | 5 | ∞ | **2** | **3** | 0 | 6 | 0 | 0 | 8 | 0 | 6 | 6 |
| 6,8,9 | 9 | ∞ | 6 | ∞ | ∞ | **5** | ∞ | **2** | **3** | 0 | 6 | 0 | 0 | 8 | 0 | 6 | 6 |
| 6,8,9,5 | 5 | ∞ | **6** | ∞ | ∞ | **5** | ∞ | **2** | **3** | 0 | 6 | 0 | 0 | 8 | 0 | 6 | 6 |
| 6,8,9,5,2 | 2 | **8** | **6** | ∞ | ∞ | **5** | ∞ | **2** | **3** | 2 | 6 | 0 | 0 | 8 | 0 | 6 | 6 |
| 6,8,9,5,2,1 | 1 | **8** | **6** | 13 | **12** | **5** | ∞ | **2** | **3** | 2 | 6 | 1 | 1 | 8 | 0 | 6 | 6 |
| 6,8,9,5,2,1,4 | 4 | **8** | **6** | **13** | **12** | **5** | 17 | **2** | **3** | 2 | 6 | 1 | 1 | 8 | 4 | 6 | 6 |
| 6,8,9,5,2,1,4,3 | 3 | **8** | **6** | **13** | **12** | **5** | **17** | **2** | **3** | 2 | 6 | 1 | 1 | 8 | 4 | 6 | 6 |
| 6,8,9,5,2,1,4,3,7 | 7 | **8** | **6** | **13** | **12** | **5** | **17** | **2** | **3** | 2 | 6 | 1 | 1 | 8 | 4 | 6 | 6 |

- Krećemo od čvora 6
- Matricu D apdejtujemo gde ima put od čvora 6 do drugog čvora (ovde do 2, 8 i 9)
- U T tabeli piše čvorprethodnik preko koga smo došli
- Zatim biramo najkraći put, to je grana dužine 2 do čvora 8. Zato je sledeći u tabeli 8.
- Ponavljamo postupak...

# 6. Zadatak

Za graf sa slike naći najkraće puteve od čvora 4 do ostalih čvorova primenom *Dijkstra*-inog algoritma. Dati izgled vektora najkraćih rastojanja i vektora prethodnika posle svake iteracije.



# Zadatak 6 – rešenje

| S | | D | | | | | | | | T | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9 | 1 | 2 | 3 | 5 | 6 | 7 | 8 | 9 |
| 4 | - | ∞ | 9 | 2 | 7 | ∞ | ∞ | ∞ | 8 | 0 | 4 | 4 | 4 | 0 | 0 | 0 | 4 |
| 4,3 | 3 | 10 | 9 | **2** | 7 | ∞ | ∞ | ∞ | 8 | 3 | 4 | 4 | 4 | 0 | 0 | 0 | 4 |
| 4,3,5 | 5 | 10 | 9 | **2** | **7** | 10 | ∞ | ∞ | 8 | 3 | 4 | 4 | 4 | 5 | 0 | 0 | 4 |
| 4,3,5,9 | 9 | 10 | 9 | **2** | **7** | 10 | ∞ | 13 | **8** | 3 | 4 | 4 | 4 | 5 | 0 | 9 | 4 |
| 4,3,5,9,2 | 2 | 10 | **9** | **2** | **7** | 10 | ∞ | 13 | **8** | 3 | 4 | 4 | 4 | 5 | 0 | 9 | 4 |
| 4,3,5,9,2,1 | 1 | **10** | **9** | **2** | **7** | 10 | ∞ | 13 | **8** | 3 | 4 | 4 | 4 | 5 | 0 | 9 | 4 |
| 4,3,5,9,2,1,6 | 6 | **10** | **9** | **2** | **7** | **10** | 14 | 13 | **8** | 3 | 4 | 4 | 4 | 5 | 6 | 9 | 4 |
| 4,3,5,9,2,1,6,8 | 8 | **10** | **9** | **2** | **7** | **10** | 14 | **13** | **8** | 3 | 4 | 4 | 4 | 5 | 6 | 9 | 4 |
| 4,3,5,9,2,1,6,8,7 | 7 | **10** | **9** | **2** | **7** | **10** | **14** | **13** | **8** | 3 | 4 | 4 | 4 | 5 | 6 | 9 | 4 |

# Predstavljanje grafa u C-u

- Preko matrice susedstva
- Preko liste susedstva
  - Težinski
  - Usmereni
  - Neusmereni

# Neusmereni graf

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

// Define maximum number of vertices in the graph
#define N 6

// Data structure to store graph
struct Graph {
    // An array of pointers to Node to represent adjacency list
    struct Node* head[N];
};

// A data structure to store adjacency list nodes of the graph
struct Node {
    int dest;
    struct Node* next;
};

// data structure to store graph edges
struct Edge {
    int src, dest;
};
```

# Neusmereni graf – Inicijalizacija

```
struct Graph* createGraph(struct Edge edges[], int n)
{
    unsigned i;

    // allocate memory for graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

    // initialize head pointer for all vertices
    for (i = 0; i < N; i++)
        graph->head[i] = NULL;

    // add edges to the directed graph one by one
    for (i = 0; i < n; i++)
    {
        // get source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;

        // 1. allocate new node of Adjacency List from src to dest

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = dest;

        // point new node to current head
        newNode->next = graph->head[src];

        // point head pointer to new node
        graph->head[src] = newNode;

        // 2. allocate new node of Adjacency List from dest to src

        newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = src;
```

```
        // point new node to current head
        newNode->next = graph->head[dest];

        // change head pointer to point to the new node
        graph->head[dest] = newNode;

    }

    return graph;
}
```

# Neusmereni graf – Štampanje

```
// Function to print adjacency list representation of graph
void printGraph(struct Graph* graph)
{
    int i;
    for (i = 0; i < N; i++)
    {
        // print current vertex and all ts neighbors
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("(%d -> %d)\t", i, ptr->dest);
            ptr = ptr->next;
        }

        printf("\n");
    }
}
```

# Neusmereni graf - Main funkcija

```
// Undirected Graph Implementation in C
int main(void)
{
    // input array containing edges of the graph (as per above diagram)
    // (x, y) pair in the array represents an edge from x to y
    struct Edge edges[] =
    {
        { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },
        { 3, 2 }, { 4, 5 }, { 5, 4 }
    };

    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct graph from given edges
    struct Graph *graph = createGraph(edges, n);

    // print adjacency list representation of graph
    printGraph(graph);

    return 0;
}
```

# Usmereni graf - Inicijalizacija

```
struct Graph* createGraph(struct Edge edges[], int n)
{
    unsigned i;

    // allocate memory for graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

    // initialize head pointer for all vertices
    for (i = 0; i < N; i++)
        graph->head[i] = NULL;

    // add edges to the directed graph one by one
    for (i = 0; i < n; i++)
    {
        // get source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;

        // allocate new node of Adjacency List from src to dest
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = dest;

        // point new node to current head
        newNode->next = graph->head[src];

        // point head pointer to new node
        graph->head[src] = newNode;
    }

    return graph;
}
```

# Usmereni graf - Štampanje

```c
// Function to print adjacency list representation of graph
void printGraph(struct Graph* graph)
{
    int i;
    for (i = 0; i < N; i++)
    {
        // print current vertex and all ts neighbors
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("(%d -> %d)\t", i, ptr->dest);
            ptr = ptr->next;
        }

        printf("\n");
    }
}
```

# Usmereni graf - Main

```
// Directed Graph Implementation in C
int main(void)
{
    // input array containing edges of the graph (as per above diagram)
    // (x, y) pair in the array represents an edge from x to y
    struct Edge edges[] =
    {
        { 0, 1 }, { 1, 2 }, { 2, 0 }, { 2, 1 },
        { 3, 2 }, { 4, 5 }, { 5, 4 }
    };

    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct graph from given edges
    struct Graph *graph = createGraph(edges, n);

    // print adjacency list representation of graph
    printGraph(graph);

    return 0;
}
```

# Težinski graf - Incijalizacija

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

// Define maximum number of vertices in the graph
#define N 6

// Data structure to store graph
struct Graph {
    // An array of pointers to Node to represent adjacency list
    struct Node* head[N];
};

// A data structure to store adjacency list nodes of the graph
struct Node {
    int dest, weight;
    struct Node* next;
};

// data structure to store graph edges
struct Edge {
    int src, dest, weight;

};
```

```c
// Function to create an adjacency list from specified edges
struct Graph* createGraph(struct Edge edges[], int n)
{
    unsigned i;

    // allocate memory for graph data structure
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));

    // initialize head pointer for all vertices
    for (i = 0; i < N; i++)
        graph->head[i] = NULL;

    // add edges to the directed graph one by one
    for (i = 0; i < n; i++)
    {
        // get source and destination vertex
        int src = edges[i].src;
        int dest = edges[i].dest;
        int weight = edges[i].weight;

        // allocate new node of Adjacency List from src to dest
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->dest = dest;
        newNode->weight = weight;

        // point new node to current head
        newNode->next = graph->head[src];

        // point head pointer to new node
        graph->head[src] = newNode;
    }

    return graph;
}
```

# Težinski graf – Štampanje i Main funkcija

```c
// Function to print adjacency list representation of graph
void printGraph(struct Graph* graph)
{
    int i;
    for (i = 0; i < N; i++)
    {
        // print current vertex and all ts neighbors
        struct Node* ptr = graph->head[i];
        while (ptr != NULL)
        {
            printf("%d -> %d (%d)\t", i, ptr->dest, ptr->weight);
            ptr = ptr->next;
        }

        printf("\n");
    }
}
```

```c
// Weighted Directed Graph Implementation in C
int main(void)
{
    // input array containing edges of the graph (as per above diagram)
    // (x, y, w) tuple in the array represents an edge from x to y having weight w
    struct Edge edges[] =
    {
        { 0, 1, 6 }, { 1, 2, 7 }, { 2, 0, 5 }, { 2, 1, 4 },
        { 3, 2, 10 }, { 4, 5, 1 }, { 5, 4, 3 }
    };

    // calculate number of edges
    int n = sizeof(edges)/sizeof(edges[0]);

    // construct graph from given edges
    struct Graph *graph = createGraph(edges, n);

    // print adjacency list representation of graph
    printGraph(graph);

    return 0;
}
```

# Test

1. Koje su tri vrste grafova i navedite razlike?
2. Koja su dva obilaska grafa?
3. Koje se pomoćne strukture korste kod obilaska grafova?
4. Šta je matrica puta?
5. Šta je MST? Preko koja dva algoritma se nalazi?
6. Koja je praktična primena MST?
7. Čemu služi Warshalow algoritam?
8. Čemu služi Dijkstijin algoritam?
9. Kako se može implementirati graf u C-u?
10. U kojim funkcijama je razlika kod implementacije grafova?

# Test

- Test poslati do 11.05.2020. u 14h na mejl apljaskovic@np.ac.rs prema uputstvima sa sajta univerziteta

# Hvala na pažnji!