Grafovi

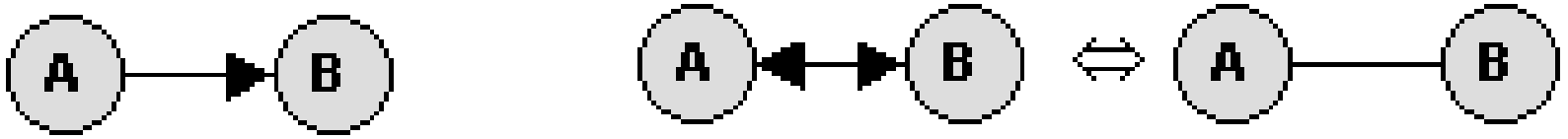
Definicija

- *Graf (engl. **graph**) je uređeni par $G = (V,E)$, gde je V konačan neprazan skup elemenata koji se nazivaju čvorovi ili temena (engl. **node**), a E konačan skup uređenih parova čvorova, tj. $E \subseteq V \times V$. Elementi skupa E nazivaju se grane ili potezi (engl. **edge**) grafa.*

Orijentisan i neorijentisani graf

Prethodna definicija odnosi se na takozvani **orijentisani graf** (**digraf**)

Ukoliko se skup grana E definiše kao skup parova čvorova, $E = \{ \{u,v\} | \, u,v \in V \wedge u \neq v \}$, graf je **neorijentisan**.



Terminologija

- Grana **e** orijentisanog grafa **izvire** iz čvora **v** ako se čvor **v** javlja kao prvi čvor u uređenom paru koji definiše dati poteg **e = (v,u)**, a ukoliko se nalazi kao drugi član uređenog para **e = (u,v)**, kaze se da **uvire** u dati čvor.
- **Izlazni stepen** nekog čvora je broj potega koji izvire iz njega, a **ulazni stepen**, broj potega koji uviru u njega. **Stepen čvora** je zbir ulaznog i izlaznog stepena.

Terminologija 2

- **Put P** u usmerenom grafu **G = (V,E)** je neprazan niz čvorova $P = (v_1, v_2, \ldots v_k)$ takvih da važi:
  $$(\forall v_i \in V) (1 \leq i \leq k \Rightarrow (v_i, v_{i+1}) \in E).$$
- **Dužina puta** P je **k-1**, odnosno, broj potega koje treba preći da bi se iz prvog stiglo u poslednji čvor tog puta.
- Čvor v_{i+1} naziva se **sledbenik** čvora v_i, a čvor v_{i-1} **prethodnik**.
- Put je **prost** ako i samo ako važi:
  $$(\forall (v_i,v_j) \in V) (1 \leq i < j \leq k \Rightarrow v_i \neq v_j).$$

Terminologija 3

- **Ciklus** je put kod koga se poklapaju prvi i poslednji čvor, tj. važi: $v_1 = v_k$.
- **Ciklus** je **prost** ako zadovoljava uslov jednostavnog puta.
- Ciklus dužine 1 naziva se **petlja**.
- Usmereni graf koji ne sadrži cikluse naziva se **usmereni aciklični graf**. (Primer takvog grafa je **stablo**)
- **Težinski grafovi** - grafovi kod kojih su dodeljeni težinski koeficijenati granama

Reprezentacije grafa

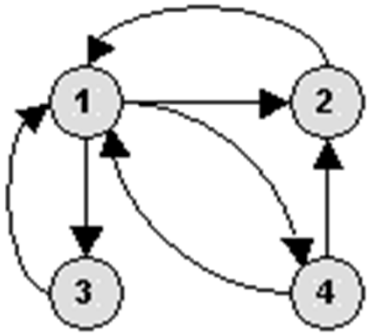
- Statička (matrca susedstva) i
- Dinamička (lista suseda)

Matrica susedstva

$$A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & (v_i, v_j) \notin E \end{cases}$$

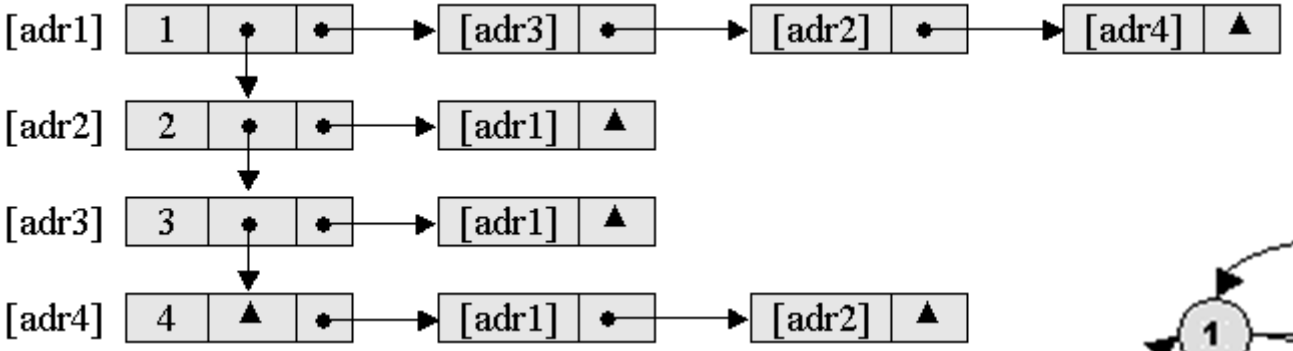
Matrica težina

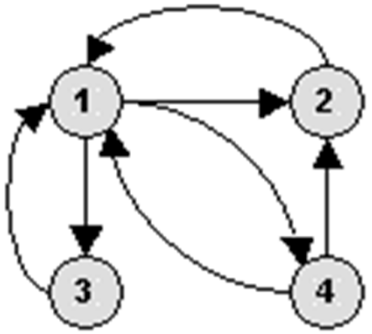
$$W_{i,j} = \begin{cases} t & (v_i, v_j) \in E \\ \infty & (v_i, v_j) \notin E \end{cases}$$



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Lista suseda





Osnovne operacije nad grafom

- **findNode** - pronalazi zadati čvor u grafu, ukoliko postoji,
- **insertNode** - dodaje novi čvor,
- **deleteNode** - briše zadati čvor, ukoliko postoji,
- **findEdge** - pronalazi zadati poteg u grafu, ukoliko postoji,
- **insertEdge** - dodaje poteg između dva zadata čvora i
- **deleteEdge** - briše poteg između dva zadata čvora, ukoliko postoji

Ostale operacije

- obilazak grafa (po širini - **breadthTrav** i dubini - **depthTrav**),
- topološko uređenje (**topologicalOrderTraversal**),
- provera postojanja ciklusa (**isCyclic**),
- povezanost grafa (**connectionLevel**) i
- nalaženje najkraćeg puta (**findShortestPath**)

Obilazak grafa

Obilazak grafa vrlo je sličan obilasku stabla, sa tom razlikom da u grafu mogu postojati ciklusi, pa je potrebno voditi računa o tome da li je neki čvor već bio posećen. Takođe, obzirom da graf nema neki poseban čvor koji ima ulogu korena stabla, kod svakog obilaska mora se zadati i čvor od koga se polazi. Graf se može obići:

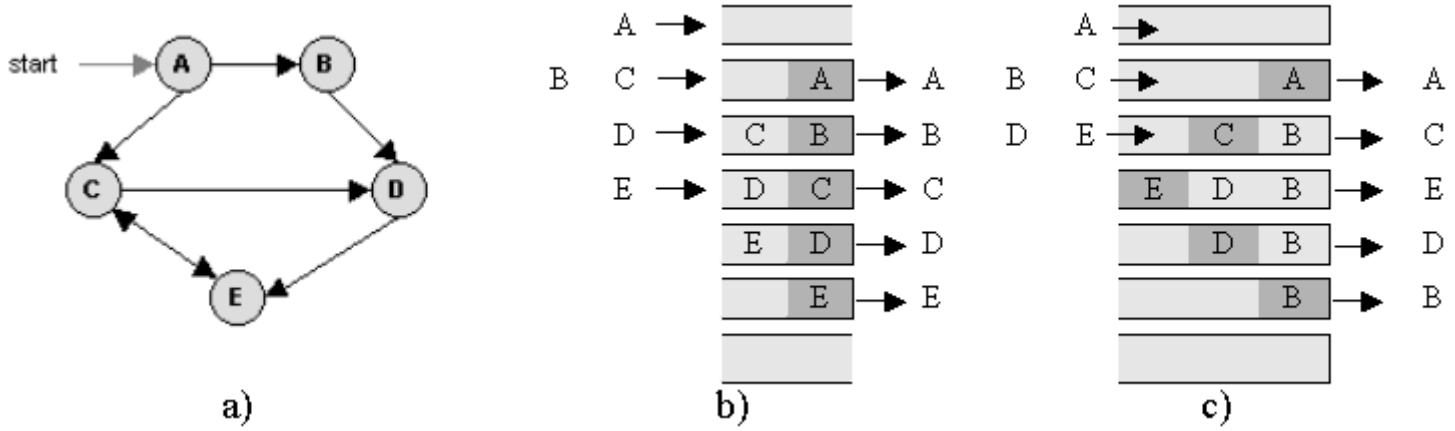
po širini - najpre se obilaze svi susedi jednog čvora pre nego što se pređe na sledeći i

po dubini - obilazi se najpre prvi sused datog čvora, pa zatim prvi sused prvog suseda, i tako redom.

Obilazak grafa

Algoritam obilaska ima sledeću strukturu:

1. svim čvorovima postaviti status na **neobrađen,**
2. uzeti startni čvor, smestiti ga u odgovarajuću strukturu i promenitimu status u **smešten,**
3. dok struktura nije prazna:
   a. uzeti čvor iz strukture, obraditi ga i promenoti mu status na **obrađen**,
   b. sve njegove susede koji imaju status **neobrađen** smestiti u strukturu i promeniti im status na **smešten**.



Obilazak grafa: a) primer grafa, b) obilazak po širini, c) obilazak po dubini

Topološko uređenje

Ukoliko grane grafa definišu zavisnost između čvorova koja zadovoljava relaciju parcijalne uređenosti, a ona je:

1. nerefleksivna,

2. antisimetrična i

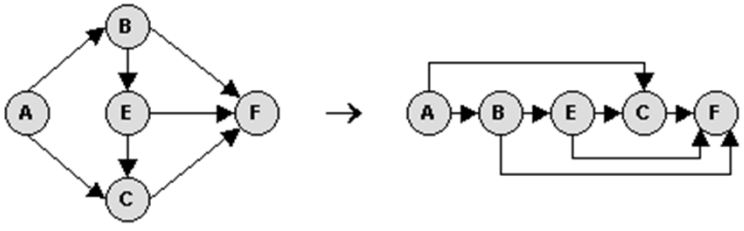
3. tranzitivna,

tada se graf može linearizovati, a proces linearizacije naziva se **topološko uređenje**. Topološkim uređenjem čvorova dobija se vektor čvorova koji zadržava uređenost definisanu granama grafa.

Topološko uređenje

Algoritam topološkog uređenja ima sledeću strukturu:

1. sve čvorove koji imaju ulazni stepen 0 smestiti u red,
2. dok red nije prazan:
   a. uzeti čvor sa početka reda i obraditi ga,
   b. svim njegovim susedima smanjiti ulazni stepen za 1 i ako je njihov ulazni stepen postao 0 smestiti ih u red.

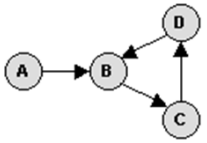


ulazni stepeni čvorova po koracima

čvor	korak 1	korak 2	korak 3	korak 4	korak 5
A	**0**				
B	1	**0**			
C	2	1	1	**0**	
E	1	1	**0**		
F	3	3	2	1	**0**
	↓	↓	↓	↓	↓
	A	**B**	**E**	**C**	**F**

Provera postojanja ciklusa i povezanost grafa

Topološko uređenje može da funkcioniše samo u slučaju orijentisanog grafa bez ciklusa. Ukoliko postoje cikluci neki čvorovi nikada neće biti posećeni, obzirom da nihov ulazni stepen prethodnom metodom nikada ne postaje 0.



brojČvorova != topologicalOrderTraversal()

Jako povezani graf je orijentisani graf kod koga postoji put između bilo koja dva čvora.

Slabo povezani graf nema to svojstvo, ali zahteva da odgovarajući neorijentisani graf (graf koji bi se dobio zamenom svih orijentisanih grana neorijentisanim) mora biti jako povezan.

Provera da li je orijentisani graf jako povezan vrši se proverom da li obilazak iniciran iz svakog čvora prolazi kroz sve čvorove grafa. Ukoliko je uslov ispunjen za sve čvorove grafa, graf je **jako povezan**. Ako uslov nije ispunjen za orijentisani graf, ali jeste za odgovarajući neorijentisani, graf je **slabo povezan**. Ukoliko uslov nije ispunjen ni za odgovarajući neorijentisani graf, graf je **nepovezan**.

Najkraći put

Postoji mnoštvo algoritama za nalaženje najkraćih puteva u grafu. Oni se generalno mogu podeliti u dve grupe:

- algoritmi koji nalaze najkraći put od jednog čvora do svih ostalih čvorova u grafu i
- algoritmi koji nalaze najkraće puteve od svakog do svakog čvora (tj. sve najkraće puteve).

Dijkstrin algoritam

1. svim čvorovima postaviti status *obrađen* na **false** i *rastojanje* na ∝,
2. izabrati startni čvor **s** i njemu postaviti status *obrađen* na **true** i *rastojanje* na **0**,
3. svim susedima $\mathbf{v_i}$ čvora **s** postaviti $\mathbf{v_i}$**.*rastojanje* = C(s,$\mathbf{v_i}$)**, gde je C(s,v_i) težina grane od čvora **s** do čvora $\mathbf{v_i}$,
4. dok nisu obrađeni svi čvorovi
   a. za radni čvor **r** izabrati (od svih čvorova u grafu) čvor sa najmanjom vrednošću *rastojanja* čiji je status *obrađen* = **false** i promeniti ga u **true**,
   b. za sve susede $\mathbf{v_i}$ čvora **r** čiji je status *obrađen* = **false** proveriti da li je vrednost $\mathbf{v_i}$**.*rastojanja*** > **r.*rastojanje* + C(r,$\mathbf{v_i}$)** i ako jeste postaviti $\mathbf{v_i}$**.*rastojanja*** = **r.*rastojanje* + C(r,$\mathbf{v_i}$)**,

Flojdov algoritam

Inicijalno, matrica najkraćih puteva postavlja se na vrednosti težina grana.

$$D_0(u,v) = \begin{cases} C(u,v) & (u,v) \in E \\ \infty & (u,v) \notin E \end{cases}$$

Flojdov algoritam zatim računa matrice D_1, D_2, ... D_V, korišćenjem sledeće formule:

$$D_{i+1}(v,w) = \min\{D_i(v,v_{i+1}) + D_i(v_{i+1},w), D_i(v,w)\}$$

kojom se za svaki par čvorova (v,w) proverava da li je rastojanje D(v,w), koje predstavlja najkraći put od v do w koji ne prolazi kroz v_{i+1} duže od zbira najkraćih rastojanja od v do v_{i+1} i od v_{i+1} do w. Ako jeste, novo najkraće rastojanje predstavlja sumu najkraćih puteva preko čvora v_{i+1}.

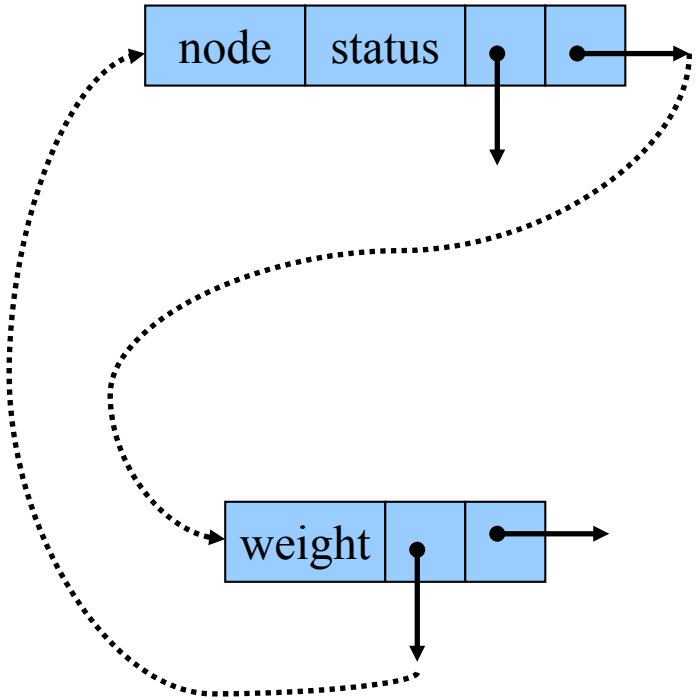
Flojdov algoritam

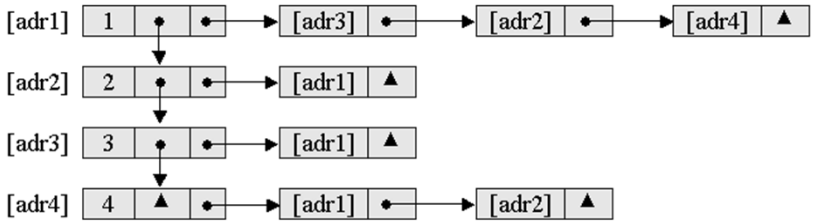
```
WFIalgorithm(matrix w) // w-težinska matrica
{
  for(int i=1, i<= vertexNo; i++)
     for(int j=1, j<= vertexNo; j++)
        for(int k=1, k<= vertexNo; k++)
           if( w[j][k] > (w[j][i] + w[i][k]) )
              w[j][k] =  w[j][i] + w[i][k];
}
```

Čvor i poteg lančane reprezentacije

```
template <class T, class W>
class LinkedNode
{
public:
        T node;
        Edge<T,W>* adj;
        LinkedNode<T,W>* next;
        int status;
};
```

```
template <class T, class W>
class Edge
{
public:
        LinkedNode<T,W>* dest;
        Edge<T,W>* link;
        W weight;
}
```





Čvor grafa

```
template <class T, class W>
class LinkedNode
{
public:
        T node;
        Edge<T,W>* adj;
        LinkedNode<T,W>* next;
        int status;
        inline LinkedNode(){ adj=NULL; next=NULL; status=0; }
        inline LinkedNode(T nodeN){ node=nodeN; adj=NULL; next=NULL; status=0; }
        inline LinkedNode(T nodeN, Edge<T,W>* adjN,
                                LinkedNode<T,W>* nextN, int statusN)
        {
                node = nodeN; adj = adjN; next = nextN; status = statusN;
        }
        inline void Visit() {cout << node << endl;}
};
```

Poteg grafa

```
template <class T, class W>
class Edge
{
public:
        LinkedNode<T,W>* dest;
        Edge<T,W>* link;
        W weight;
        inline Edge(){ dest=NULL; link=NULL; }
        inline Edge(LinkedNode<T,W>* destN,
                        Edge<T,W>* linkN) { dest=destN; link=linkN; }
};
```

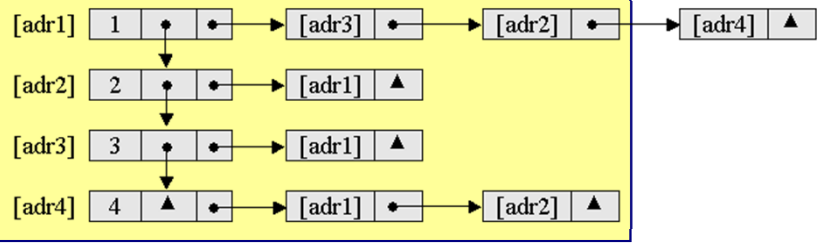
Lančana reprezentacija grafa

```
template <class T, class W>
class GraphAsLists
{
protected:
        LinkedNode<T,W>* start;
        long nodeNum;
public:
        inline GraphAsLists() { start = NULL; nodeNum=0; }
        ~GraphAsLists();
        LinkedNode<T,W>* findNode(T pod);
        Edge<T,W>* findEdge(T a, T b);
        bool insertNode(T pod);
        bool deleteNode(T pod);
        bool insertEdge(T a, T b);
        bool deleteEdge(T a, T b);
        long breadthTrav(T a);
        long depthTrav(T a);
        long topologicalOrderTrav();
        void print();
protected:
        void deleteEdgeToNode(LinkedNode<T,W>* ptr);
};
```

Štampanje grafa

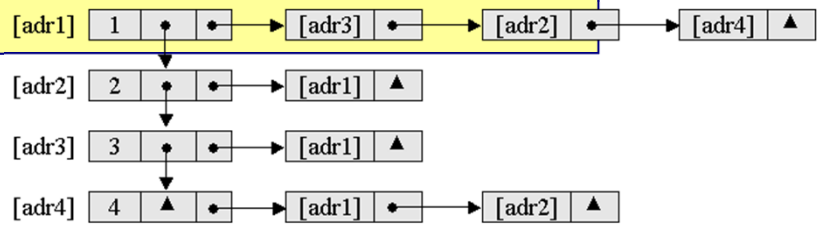
```
template <class T, class W>
void GraphAsLists<T,W>::print()
{
        LinkedNode<T,W>* ptr = start;
        while(ptr != NULL)
        {
                cout << ptr->node << "->";
                Edge<T,W>* pa = ptr->adj;
                while(pa != NULL){
                        cout << pa->dest->node << "|";
                        pa = pa->link;
                }
                cout << "\r\n";
                ptr = ptr->next;
        }

}
```



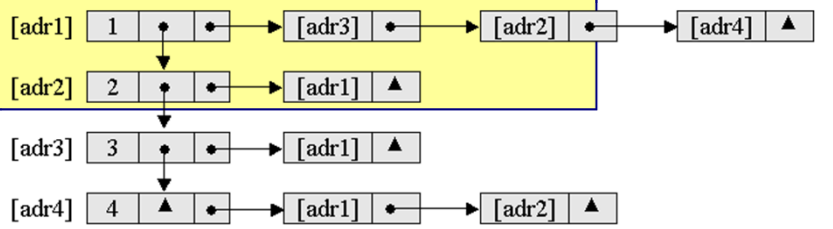
Traženje čvora

```
template <class T, class W>
LinkedNode<T,W>*
GraphAsLists<T,W>::findNode(T pod)
{
        LinkedNode<T,W>* ptr = start;
        while(ptr != NULL && ptr->node != pod)
                ptr = ptr->next;
        return ptr;
}
```



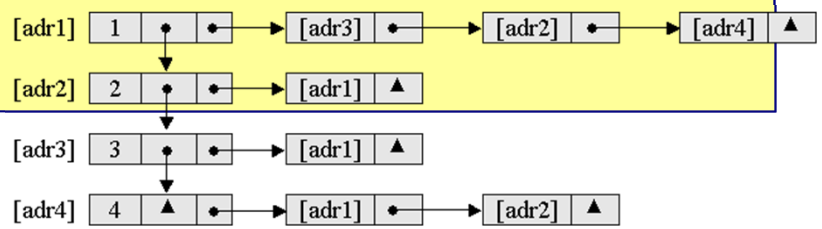
Traženje potega

```
template <class T, class W>
Edge<T,W>* GraphAsLists<T,W>::findEdge(T a, T b)
{
        LinkedNode<T,W>* pa = findNode(a);
        LinkedNode<T,W>* pb = findNode(b);
        if(pa == NULL || pb == NULL) return NULL;
        Edge<T,W>* ptr = pa->adj;
        while(ptr != NULL && ptr->dest != pb)
                ptr = ptr->link;
        return ptr;
}
```



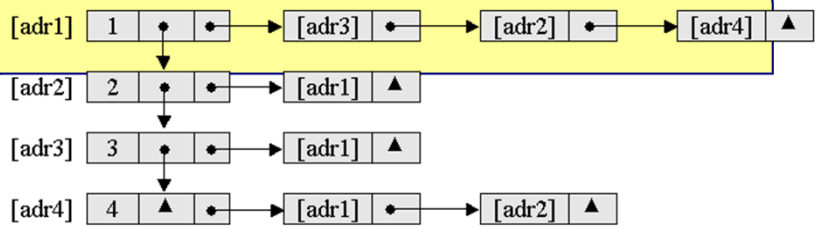
Umetanje čvora

```
template <class T, class W>
bool GraphAsLists<T,W>::insertNode(T pod)
{
        LinkedNode<T,W>* newNode =
                new LinkedNode<T,W>(pod, NULL, start, 0);
        if(newNode == NULL) return false;
        start = newNode;
        nodeNum++;
        return true;
}
```



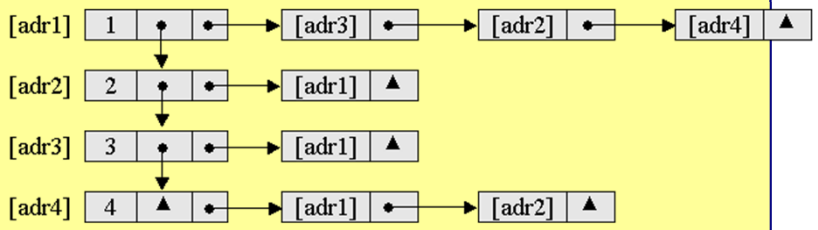
Umetanje potega

```
template <class T, class W>
bool GraphAsLists<T,W>::insertEdge(T a, T b)
{
        LinkedNode<T,W>* pa = findNode(a);
        LinkedNode<T,W>* pb = findNode(b);
        if(pa == NULL || pb == NULL) return false;
        Edge<T,W>* ptr = new Edge<T,W>(pb,pa->adj);
        if(ptr == NULL) return false;
        pa->adj = ptr;
        return true;
}
```



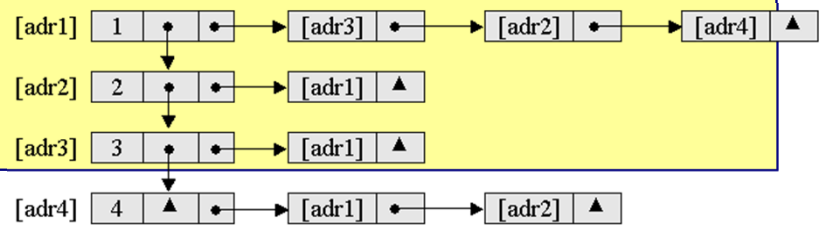
Brisanje potega

```
template <class T, class W>
bool GraphAsLists<T,W>::deleteEdge(T a, T b)
{
        Edge<T,W>* pot = findEdge(a,b);
        if(pot == NULL) return false;
        LinkedNode<T,W>* pa = findNode(a);
        if(pot == pa->adj){
                pa->adj = pot->link;
                delete pot;  return true;
        }
        Edge<T,W>* tpot = pa->adj;
        while(tpot->link != pot)
                tpot = tpot->link;
        tpot->link = pot->link;
        delete pot;   return true;
}
```



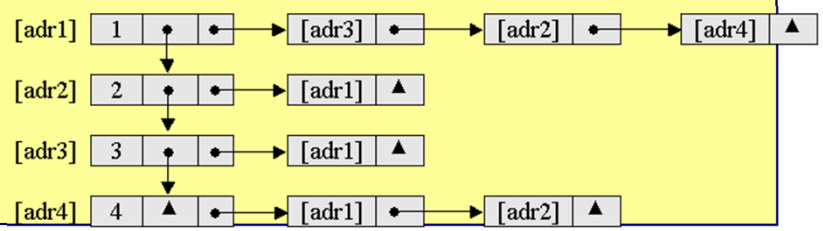
Brisanje čvora – 1. deo

```
template <class T, class W>
bool GraphAsLists<T,W>::deleteNode(T pod)
{
        LinkedNode<T,W>* ptr;
        if(start == NULL) return false;
        if(start->node == pod){
                Edge<T,W>* pot = start->adj;
                while(pot != NULL){
                        Edge<T,W>* tpot = pot->link;
                        delete pot;
                        pot = tpot;
                }
                deleteEdgeToNode(start);
                ptr = start;
                start = start->next;
                delete ptr;
                nodeNum--;
                return true;
        }
```



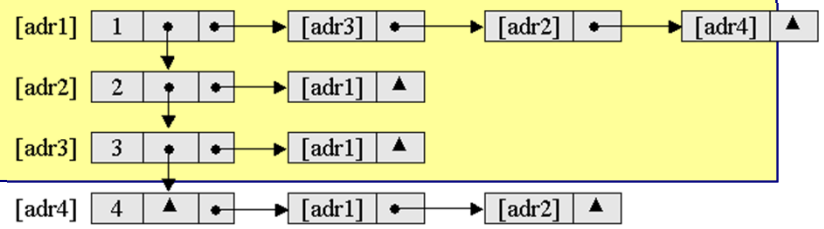
Brisanje čvora – 2. deo

```
        else{
                ptr = start->next;
                LinkedNode<T,W>* par = start;
                while(ptr != NULL && ptr->node != pod){
                        par = ptr;
                        ptr = ptr->next;
                }
                if(ptr == NULL) return false;
                par->next = ptr->next;
                Edge<T,W>* pot = ptr->adj;
                while(pot != NULL){
                        Edge<T,W>* tpot = pot->link;
                        delete pot;   pot = tpot;
                }
                deleteEdgeToNode(ptr);
                delete ptr;
                nodeNum--;
                return true;
        }
}
```



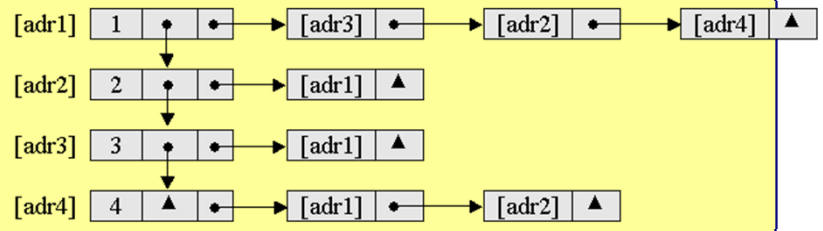
Obilazak po širini – 1. deo

```
template <class T, class W>
long GraphAsLists<T,W>::breadthTrav(T a)
{
        long retVal = 0;
        LinkedNode<T,W>* ptr = start;
        QueueAsArray<LinkedNode<T,W>*> que(nodeNum);
        while(ptr != NULL){
                ptr->status = 1;
                ptr = ptr->next;
        }
        ptr = findNode(a);
        if(ptr == NULL) return 0;
        que.enqueue(ptr);
        ptr->status = 2;
```



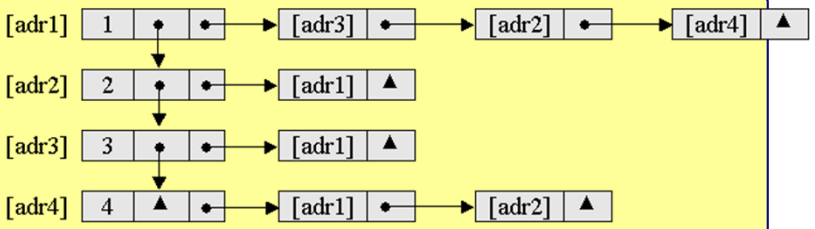
Obilazak po širini - 2. deo

```
        while(!que.isEmpty()){
                ptr = que.dequeue();
                ptr->status = 3;
                ptr->Visit();
                retVal += 1;
                Edge<T,W>* pot = ptr->adj;
                while(pot != NULL)          {
                        if(pot->dest->status == 1){
                                pot->dest->status = 2;
                                que.enqueue(pot->dest);
                        }
                        pot = pot->link;
                }
        }
        return retVal;
}
```

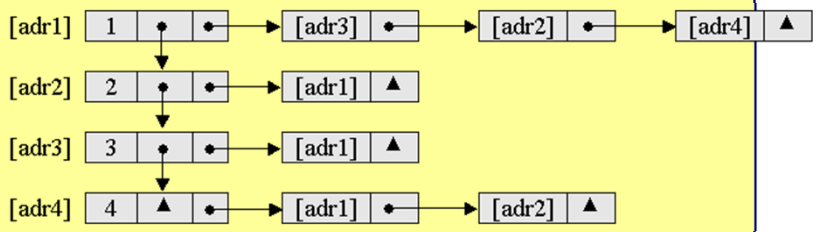


Topološki obilazak

```
template <class T, class W>
long GraphAsLists<T,W>::topologicalOrderTrav()
{
        int retVal = 0; //broj obidjenih cvorova
        // Odredjivanje ulaznog stepena za svaki cvor
        LinkedNode<T,W> *ptr;
        ptr = start;
        while(ptr != NULL){
                ptr->status = 0;
                ptr = ptr->next;
        }
        ptr = start;
        while(ptr != NULL){
                Edge<T,W>* pot = ptr->adj;
                while(pot != NULL){
                        pot->dest->status += 1;
                        pot = pot->link;
                }
                ptr = ptr->next;
        }
```



```
        // Sve izvorne potege smestiti u red
        QueueAsArray<LinkedNode<T,W>*> que(nodeNum);
        ptr = start;
        while(ptr != NULL){
                if(ptr->status == 0)  que.enqueue(ptr);
                ptr = ptr->next;
        }
        while(!que.isEmpty()){
                ptr = que.dequeue();
                ptr->Visit();   retVal += 1;
                Edge<T,W>* pot = ptr->adj;
                while(pot != NULL){
                        pot->dest->status -= 1;
                        if(pot->dest->status == 0)
                        que.enqueue(pot->dest);
                        pot = pot->link;
                }
        }
        return retVal;
}
```



Zadaci

- Projektovati klasu za rad da dinamičkom strukturom težinskog neusmerenog grafa i implemetirati funkciju za kreiranje minimalnog sprežnog stabla primenom Kraskalovog algoritma.

Minimalno sprežno stablo

Razmotrimo problem saobraćajnih komunikacija između gradova u otežanim uslovima. U slučaju obilnih snežnih padavina, na primer, nemoguće je održavati sve saobraćanice prohodnim.

Neophodno je opredeliti se za minimalan broj puteva, ali tako da oni ipak povezuju sve gradove. Logički posmatrano, mreža gradova i saobraćajnica predstavlja graf u kome gradovi predstavljaju čvorove, a putevi grane grafa. Da bi se ispunio uslov povezivanja svih gradova tako da sigurno postoji tačno jedan put između svaka dva, potrebno je *transformisati graf u stablo*.

Stablo koje sadrži sve čvorove grafa naziva se **sprežno stablo**.

Ukoliko su težine grana koje ostaju minimalne, dobija se **minimalno sprežno stablo**.

Najpoznatiji algoritmi za kreiranje minimalnog sprežnog stabla

- Kraskalov
- Jarnik-Primov
- Dijkstrin
- Boruvkin

Kraskalov algoritam (Kruskal)

Kreirati na početku prazno stablo i pomoćnu strukturu (Heap) u kojoj se nalaze sve grane grafa sortirane po težini. Dokle god stablo ne sadrži V-1 granu, gde je V broj čvorova u grafu, odnosno, dok svi čvorovi nisu povezani, dodaje se jedna po jedna grana iz pomoćne strukture, ali tako da ne formira ciklus sa granama koje su već dodate u stablo.

Obzirom da se dodaje grana po grana, pri čemu se ne vodi računa da li su one međusobno povezane ili ne, već samo da im je težina minimalna i da ne formiraju ciklus, stablo zapravo ne postoji sve dok se algoritam ne završi.

Složenost ovog algoritma direkno zavisi od brzine uređenja grana po težinama, tako da se kao pomoćna struktura obično koristi gomila. Za stablo koristiti klasu formiranu za graf.

Jarnik-Primov algoritam

Vrlo sličan prethodnom, osim što vodi računa da se dodaje samo grana koja je incidentna nekom od čvorova koji su već dodati u stablo.

Dijkstrin algoritam

Sličan prethodnim, ali se razlikuje po tome što ne zahteva prethodno uređenje grana po težinama. Grane se dodaju jedna po jedna, a ukoliko se formira ciklus, izbacuje se grana iz ciklusa koja ima najveću težinu.

Boruvkin algoritam
(*Boruvka*)

Formira od svakog čvora po jedno stablo (samo sa korenom).

Dokle god postoji više od jednog stabla, za svako od stabala traži se grana u grafu, minimalne težine, koja sadrži jedan čvor iz tog stabla i drugi koji ne pripada tom stablu.

Novo stablo kreira se spajanjem tekućeg i stabla kome pripada drugi čvor izabrane grane.