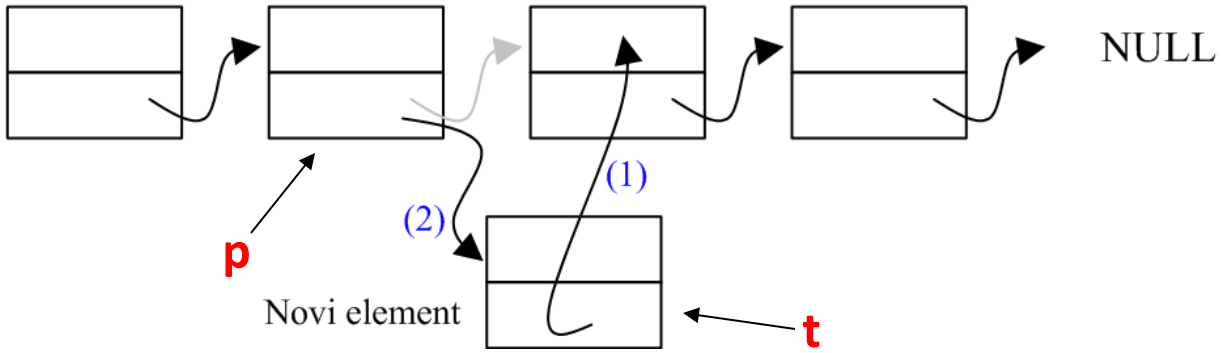
Programiranje I

Liste – Nastavak
Grafovi

Dodavanje elementa u sredinu liste

- Prvo grafički ilustrujmo šta se dešava u ovom slučaju:



- Treba odrediti poziciju (element liste na koji pokazuje pokazivač p) iza koje treba ubaciti novi element (na koji pokazuje pokazivač t).
- Zatim, pokazivač next iz elementa na koji pokazuje t treba usmjeriti na element koji se nalazi nakon elementa na koji pokazuje p (korak (1)).
- Konačno, pokazivač next iz elementa na koji pokazuje p treba preusmjeriti da pokaže na novododati element t (korak (2)).

Dodavanje elementa u sredinu liste

- Dio koda:

```
t->next = p->next;
p->next = t;
```

- **Vježba**: Pokušajte da napišete funkciju (bez gledanja u zbirku ☺) za koju važi sljedeće:

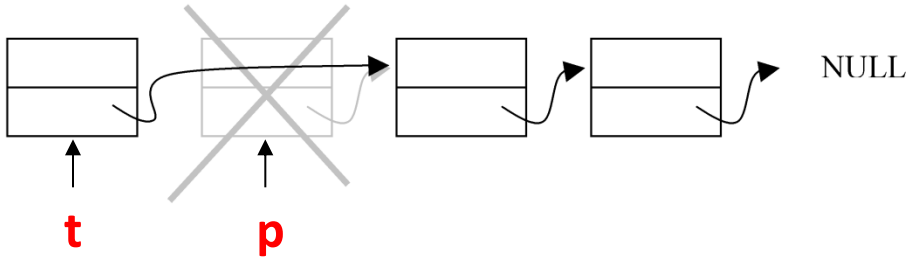
  Elementi (cijeli brojevi) upisani u čvorovima liste su sortirani u rastući redosljed. Napisati funkciju kojoj se prosljeđuju pokazivač na listu i cijeli broj, kojeg treba smjestiti u novi član liste postavljen tako da lista ostane sortirana. Funkcija vraća pokazivač na glavu liste.

Komentari problema umetanja

- Prva varijanta je da je predati element manji od elementa upisanog u glavi liste. U tom slučaju treba alocirati novi element, upisati u njega cijeli broj – argument funkcije, pa njegov pokazivač **next** usmjeriti na glavu liste. Tada novi element postaje glava liste i funkcija treba da vrati pokazivač na njega.
- Ako prva varijanta nije zadovoljena, krećemo u proceduru traženja prave pozicije za novi element liste. Sa srećom!

Brisanje elementa iz sredine liste

- Grafički se ovaj problem može prikazati na sljedeći način:



Procedura je relativno jednostavna. Treba pronaći element liste koji se briše. To se obavlja sa pozicije prethodnog elementa, na koji pokazuje pokazivač **t**. Zatim se zapamti pokazivač **t->next** (naredba **p=t->next**), a **t** se preusmjeri da pokaže na element nakon **t->next**, tj. **t->next = t->next->next**. Na kraju, vršimo dealociranje elementa na koji pokazuje **p** sa **free(p)**. Na ovaj način je "premošten" element koji se briše, a da bi se on obrisao moramo uvesti pomoćni pokazivač na njega (u našem slučaju **p**).

Zadatak za vježbu

- Kreirati funkciju kojoj se proslijeđuje glava liste i cijeli broj upisan u elementu liste kojeg treba obrisati. Lista je sortirana u neopadajući redosljed. Funkcija treba da pored brisanja elementa vrati glavu liste. U slučaju da traženi element ne postoji u listi ne treba vršiti brisanje. Treba predvidjeti i situaciju da je glava liste traženi element, kao i situaciju da je glava liste ujedno i rep liste i da je to traženi element (kad je lista prazna i nakon operacije brisanja treba vratiti NULL).

Brojanje elemenata liste – rekurzivno

- Da bismo ilustrovali rekurzivni rad sa listama, napišimo funkciju koja broji elemente liste. Radi olakšice, pretpostavimo da lista ima barem jedan element. Ako je taj element rep liste, funkcija treba da vrati 1. Ako element nije rep liste, funkcija treba da vrati 1 + broj_elemenata_u_ostatku_liste, a ostatak liste je opet lista koja počinje od narednog elementa.

```
int brojElemente(struct lista *glava) {
   if(glava->next == NULL)
        return 1;
   else
        return 1 + brojElemente(glava->next); }
```

- Napomena: Rekurzivna realizacija funkcije nije efikasna zbog velikog broj poziva funkcije. Iterativna realizacija je efikasnija.

Nadovezivanje listi – iterativno

- Imamo dvije liste i na kraj prve želimo da nadovežemo drugu. To znači da rep prve liste treba da pokaže na glavu druge liste. Nakon što odredimo rep prve liste, njegov pokazivač **next** treba preusmjeriti na glavu druge liste:

```
struct lista *nadovezi(struct lista *gl1, struct lista *gl2) {
    struct lista *pom = gl1;
    while(pom->next != NULL)
        pom = pom->next;
    pom->next = gl2;
    return gl1;
}
```

Uvodimo pomoćni pokazivač za kretanje kroz prvu listu.

Idemo na kraj prve liste.

Rep prve liste ukazuje na glavu druge liste.

Vraćamo glavu novonastale liste.

Specijalni tipovi listi

- Prvi specijalni tip liste je **stek** (eng. stack). U pitanju je lista kod koje se dodavanje novih elemenata i brisanje postojećih izvodi samo sa pozicije repa.
- Stek je FILO (First-In-Last-Out) memorija, odnosno član liste koji prvi uđe u listu posljednji iz nje izlazi, i obrnuto – posljednji uđe, prvi izađe.
- Operacije kod steka su:
  - **push** (dodavanje elementa),
  - **pop** (uklanjanje posljednjeg dodatog elementa) i
  - **peek** (čitanje posljednjeg dodatog elementa, bez modifikacije steka).
- Implementiraju se i operacije provjere da li je stek pun ili prazan.
- Osim preko listi, stekovi mogu realizovati i pomoću nizova.

Specijalni tipovi listi

- Drugi specijalni tip liste je **red** (eng. queue). Kod reda se podaci dodaju na početak liste, a brišu sa kraja liste.
- Kao takav, red je FIFO (First-In-First-Out) memorija, odnosno elementi koji prvi ulaze u listu iz nje se kao prvi i brišu, i obrnuto – oni koji posljednji uđu, posljednji izađu.
- Tri osnovne operacije kod reda su:
  - **enqueue** (dodavanje elementa na početak reda),
  - **dequeue** (uklanjanje elemenata sa kraja reda) i
  - **peek** (čitanje elementa sa kraja reda, bez njegovog uklanjanja).
- Ovdje, također, možemo implementirati operacije provjere da li je red pun ili prazan.
- Rad sa stekom i redom je jednostavniji nego rad sa jednostruko povezanom listom koja dozvoljava da se upis i brisanje vrše na svim pozicijama.

Dvostruko-povezana lista

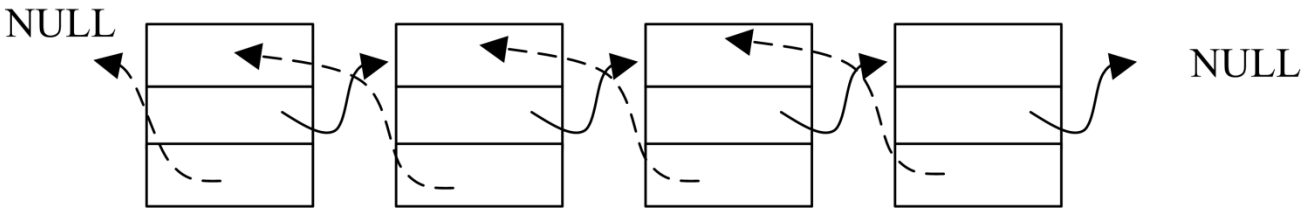
- Kvalitativno drugačiji tip liste je dvostruko-povezana lista. Kod ove liste čvorovi pamte, pored narednog, i prethodni čvor.
- Struktura pomoću koje se realizuje dvostruko-povezana lista se može deklarisati kao:

```
struct lista2pov {
    ... // podaci elementa liste
    struct lista2pov *next;
    struct lista2pov *prev;
};
```

Pokazivači na naredni i prethodni čvor liste

Vizuelizacija dvostruko-povezane liste

- Uobičajena grafička predstava dvostruko-povezane liste je:

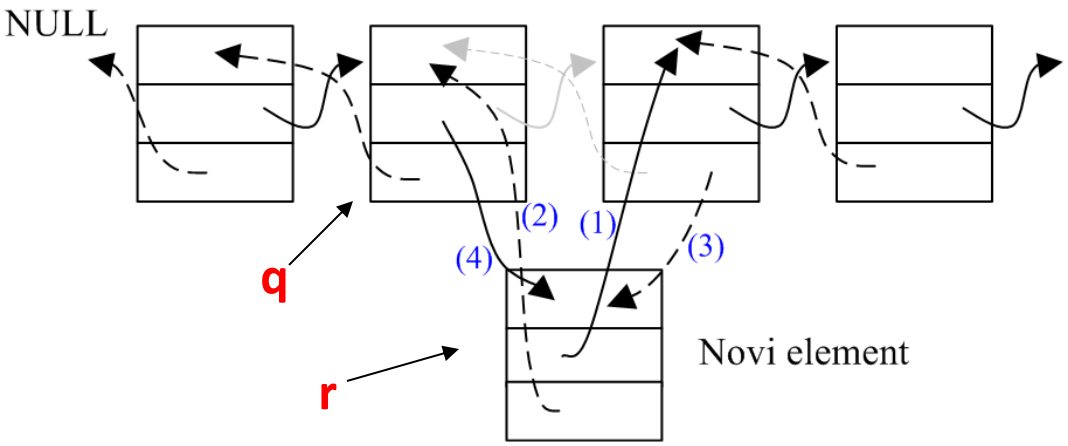


Punom linijom su označeni pokazivači na naredne elemente u listi, dok su isprekidanom označeni pokazivači na prethodne elemente. Ovaj tip liste ima dodatnu funkcionalnost, odnosno dozvoljava kretanje i unazad i unaprijed po listi.

Međutim, rad sa ovakvim tipom liste zahtjeva više pažnje, jer se mora voditi računa o dva pokazivača.

Dodavanje elementa u DP listu

- Vizuelizacija dodavanja elementa u dvostruko-povezanu (DP) listu:



Neka je r pokazivač na element liste koji se dodaje, dok je q pokazivač na element liste iza kojeg se novi element dodaje.

```
r->next=q->next;     (korak (1))
r->prev=q;           (korak (2))
q->next->prev=r;     (korak (3))
q->next=r;           (korak (4))
```

Jednostavno pravilo kojega se treba držati je da prvo novi element uspostavi veze sa ostalim čvorovima u listi, pa da se tek onda prekidaju i preusmjeravaju stare veze.

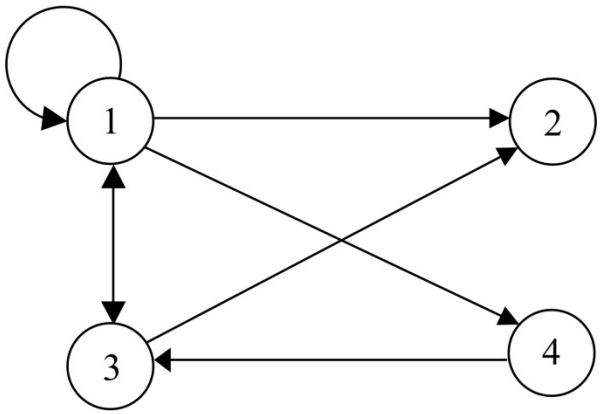
Grafovi

- Liste su napredni podaci. Pored pomenutih, postoje i drugi tipovi listi koji, iz razloga složenosti i vremena kojeg u kursu imamo, neće biti razrađivani.
- **Graf** je još jedan napredan tip podataka.
- Graf je skup čvorova u kojem svaki čvor može da pokaže na proizvoljan broj drugih čvorova.
- Precizna matematička definicija grafa je:

  Graf **G=(V,E)** je uređeni par koji se sastoji od konačnog nepraznog skupa čvorova **V** i skupa ivica **E**. Ako je ivica uređeni par čvorova **(v,w)** iz skupa **V**, gdje je **v** početak, a **w** kraj ivice, tada govorimo o **usmjerenom grafu**. U suprotnom je riječ o **neusmjerenom grafu**.

Vizuelizacija grafa

- Standardna metodologija za vizuelizaciju grafa je data na slici.



Predmetni graf ima četiri čvora. Ovo je usmjeren graf. Kod usmjerenog grafa koriste se strelice da prikažu početni i krajnji čvor ivice (krajnji čvor je onaj na koji pokazuje strelica).

Kod usmjerenog grafa dozvoljena je ivica **(v,v)**, dok kod neusmjerenog nije dozvoljeno da isti čvor bude i početak i kraj ivice.

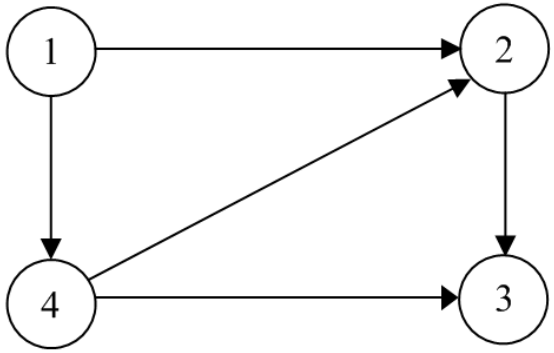
Graf – Definicije

- Ako postoji ivica **(v,w)** u grafu kaže se da je čvor **w** susjed čvora **v** (ovo ne implicira da je **v** susjed čvora **w**).
- Broj čvorova koji su susjedi čvora **v** naziva se izlazni stepen čvora.
- Skup ivica **{(v_1,v_2), (v_2,v_3), …, (v_{n-1},v_n)}** naziva se put od čvora **v_1** do čvora **v_n**.
- Dati put je dužine **n-1**.
- Jednostavan put je onaj kod kog se nijedan čvor osim prvog i posljednjeg ne ponavlja (početak i kraj jednostavnog puta može biti u istom čvoru, ali ne mora).
- Ciklus je jednostavan put, minimalne dužine **3**, koji počinje i završava se u istom čvoru.

Predstavljanje grafa

- Postavlja se pitanje kako se grafovi memorijski predstavljaju.
- Čvorovi grafa predstavljaju neke podatke strukturnog tipa.
- Postoje nekoliko načina da se memorišu veze između čvorova.
- Najčešće korišćeni način je preko **matrice susjedstva**. Ako je N_V broj čvorova u grafu, matrica susjedstva je matrica dimenzija $N_V \times N_V$, sa vrijednošću **1** na poziciji **[I][J]** ako je čvor **J** susjed čvoru **I**. Ako čvor **J** nije susjed čvoru **I**, onda se na odgovarajućem mjestu u matrici susjedstva nalazi **0**.

Matrica susjedstva



Graf

	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0

Matrica susjedstva

- Na osnovu matrice susjedstva se lako može provjeriti da li je u pitanju usmjereni ili neusmjereni graf. Kako?
- Takođe, veoma jednostavno se određuje izlazni stepen svakog čvora. Kako?
- Mnoštvo drugih informacija o grafu se može dobiti koristeći matricu susjedstva.

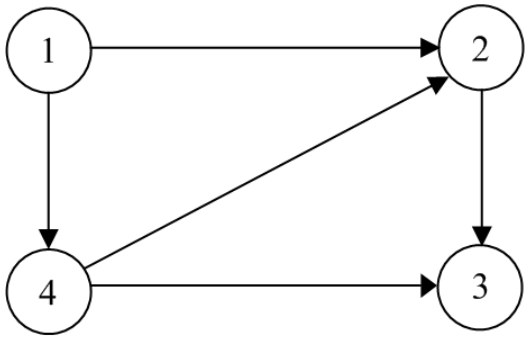
Mana matrice susjedstva

- Osnovni nedostatak matrice susjedstva je memorijska zahtjevnost. Naime, u grafu može postojati veoma veliki broj čvorova sa veoma malim brojem veza između njih.
- Pretpostavimo da je $\mathbf{N_V=1000}$. Tada matrica susjedstva ima milion elemenata, pri čemu se može dogoditi da je samo nekoliko hiljada nenultih (samo nekoliko hiljada ivica).
- Postoje strategije koje omogućavaju da se i dalje koristi matrica susjedstva:
  - prva strategija je preko polja bitova, kada se pamćenje jedne jedinice ili nule vrši preko 1 bita, a ne recimo 2 bajta kao za cijeli broj.
  - alternativa je da se umjesto matrice pamte uređeni parovi koji predstavljaju poziciju nenultih elemenata (za graf sa prethodnog slajda bi to bilo: (1,2), (1,4), (2,3), (4,2) i (4,3)). Ovakve matrice se nazivaju **rijetkim** (eng. sparse).

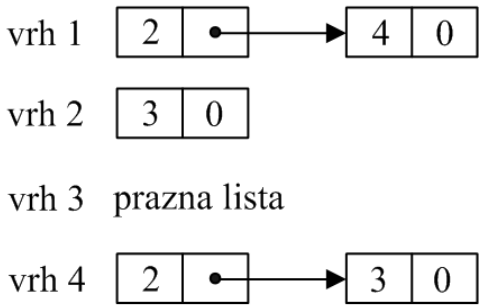
Alternativni načini predstavljanja grafa

- Ako se ne koristi matrica susjedstva u osnovnoj formi, gubi se osnovna prednost ovog načina predstavljanja grafa - jednostavnost.
- Stoga su uvedeni drugi načini za memorisanje veza između čvorova grafa.
- Jedan od načina je preko **listi susjedstva**.
- Za svaki čvor grafa formira se lista čiji su elementi njemu susjedni čvorovi.

Liste susjedstva – Primjer



Graf



Liste susjedstva

- Iz listi susjedstva lako zaključujemo da čvor (vrh) **1** ima susjedne čvorove **2** i **4**, dok čvor **3** nema nijedan susjedni čvor.
- Ukupan broj elemenata u listama susjedstva je jednak N_V+N_E, gdje je N_E broj ivica u grafu.
- Ovo je, očigledno, mnogo efikasnije za memorisanje kod grafova sa relativno velikim brojem čvorova i relativno malim brojem ivica.

Problem najkraćeg puta

- Grafovi su izuzetno upotrebljivi u brojnim primjenama:
  - algoritmi optimizacije;
  - planiranje saobraćaja;
  - planiranje poslovanja;
  - električna kola, itd.
- Obično treba dobro poznavati programiranje, problem koji treba riješiti i matematičku teoriju grafova. To znanje se brzo isplati izuzetno jednostavnim rješenjima.
- Da bismo ilustrovali snagu grafova, posmatraćemo poznati problem najkraćeg puta.

Problem najkraćeg puta

- Pretpostavimo da u grafu imamo N čvorova. Pretpostavimo da poznajemo cijene (težine) ivica između susjednih čvorova. Cijena je nenegativna veličina dodijeljena ivici. U slučaju da nema ivice između dva čvora cijena je beskonačna. Matrica sa cijenama ivica podsjeća na matricu susjedstva, ali umjesto binarnih vrijednosti upisane su vrijednosti u intervalu $[0,\infty)$.
- Problem je odrediti najjeftiniju putanju između svih čvorova I i J u grafu koja može prolaziti preko proizvoljno mnogo ivica.

Problem najkraćeg puta

- Problem najkraćeg puta se može riješiti kroz sljedeće algoritamske korake.
  - **Korak 1.** Zada se broj čvorova **N**.
  - **Korak 2.** Zada se cijena puta, tj. ivica, između svih čvorova: **l[i][j]** za i∈[0,N) i j∈[0,N) (uglasta zagrada označava uključenu granicu, a obična granicu koja nije uključena).
  - **Korak 3.** Postavi se početna iteracija: **cs[i][j]=l[i][j]** za **i≠j** i **cs[i][j]=0** za **i=j**, jer je najjeftinije ne kretati se iz jednog čvora u samog sebe.
  - **Korak 4.** Ciklus po **k** u granicama **[0,N)** u kojem se računa:

$$cn[i][j] = \min\{cs[i][j], cs[i][k]+cs[k][j]\}$$

Ključni korak algoritma!

Problem najkraćeg puta

- **Korak 5.** Ažuriranje stare vrijednosti matrice težina puta **cs[i][j]=cn[i][j]** čime se ciklus priprema za novu iteraciju. Povratak na **korak 4**.

- Što predstavlja ključni korak algoritma:

  cn[i][j] = min{cs[i][j], cs[i][k]+cs[k][j]} ?

- Ovaj korak bira jeftiniju od dvije moguće putanje od čvora i do čvora j:
  - > direktnu od i ka j, ili
  - > indirektnu od i ka j preko k (od i ka k, pa od k ka j).

- Nakon prve iteracije u petlji po k (k=0) matrica cn sadrži najjeftinije putanje između bilo koja dva čvora i i j koje su se mogle napraviti preko čvora 0, nakon toga za k=1 dobijamo matricu najjeftinijih putanja između bilo koja dva čvora koja se mogla napraviti preko čvorova 0 i 1.

Dijkstra algoritam

- Nakon posljednjeg koraka u algoritmu dobijamo najkraći put između svih čvorova u grafu i i j, a preko svih čvorova u grafu od 0 do N-1.
- Nestrpljivi studenti mogu da pokušaju da realizuju predmetni algoritam sami **(bez gledanja u materijale)**, dok oni strpljivi **(ovdje strpljivost nije vrlina)** mogu da sačekaju naredno predavanje.
- Predmetni algoritam se naziva **Dijkstra algoritam**.