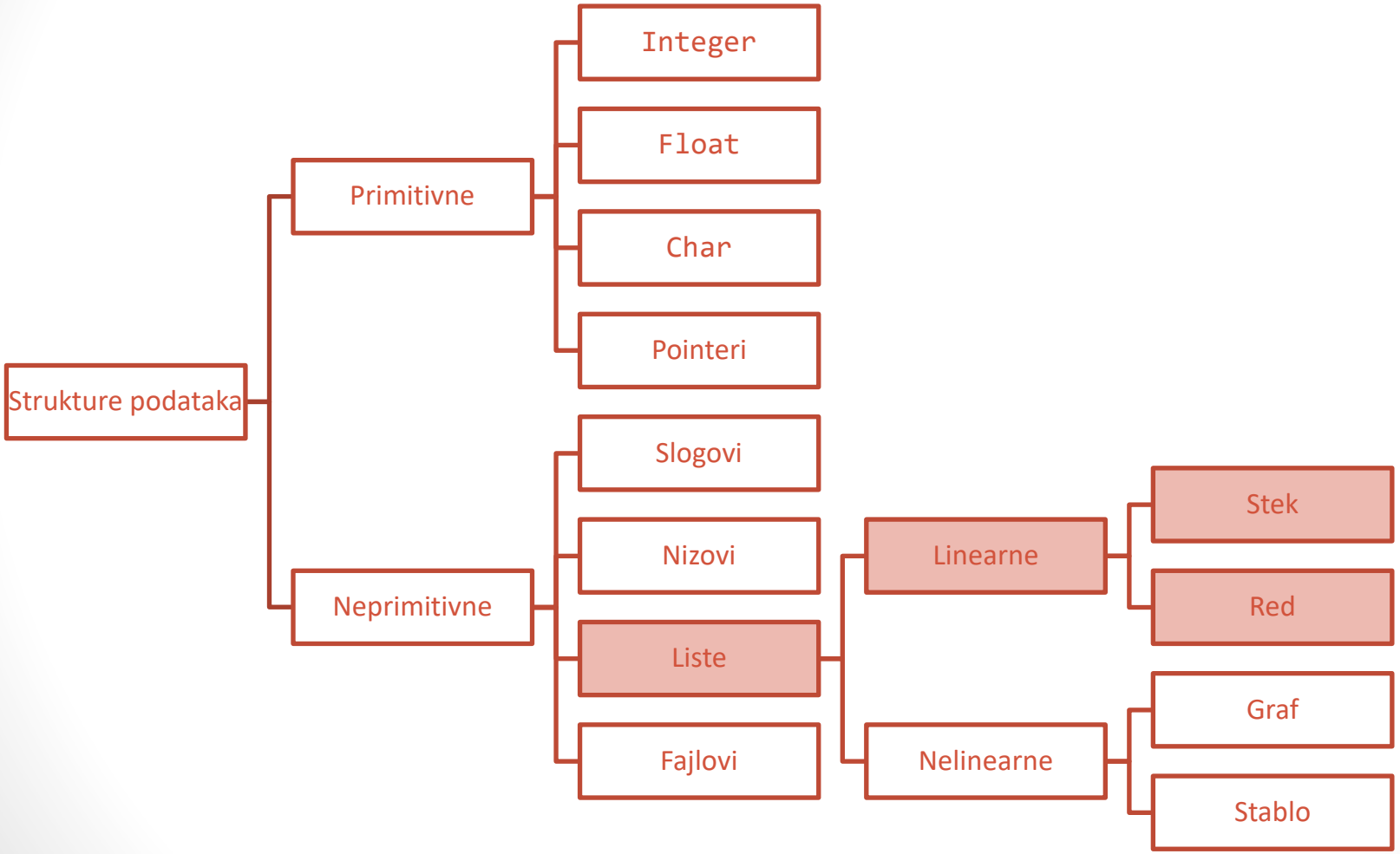
# LISTE

# Tipovi struktura podataka



# Povezane (pointerske) liste podataka

- Povezane liste su
  - Poput nizova linarna struktura podataka
  - Za razliku od nizova elementi nisu smešteni u kontinuitetu, već su povezani pointerima

- Ograničenja nizova
  - Maksimalan broj elemenata mora biti poznat unapred
  - Umetanje i brisanje elementa zahteva pomeranje ostalih elemenata niza što podrazumeva dodatno trošenje vremena i memorije

- Prednosti povezane liste
  - Promenljiva dužina liste
  - Jednostavno dodavanje i brisanje elemenata

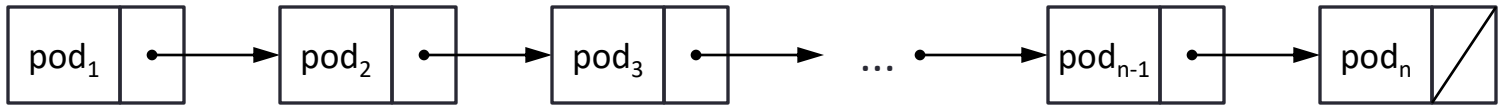
# Povezane (pointerske) liste podataka

- Nedostaci povezane liste
  - Nema direktnog pristupa elementima
    - Za pristup ma kom elementu mora se proći kroz sve elemete pre toga
    - Nije moguće efikasno primaniti binarnu pretragu i slično
  - Svaki element zauzima dodatni prostor koji sadrži pokazivač ka narednom elementu
  - Nije „*cache friendly*“ pošto elementi u memoriji nisu u kontinuitetu

# Povezane (pointerske) liste podataka

- Svaki element liste sadrži dva dela
  - Podatak, *data field*
  - Adresu narednog elementa, vezu, *link*

```
struct element
{
    int podatak;
    struct element *sledeci;
};
```



# Kreiranje elementa liste

- Za svaki novi element povezane liste je neophodno
  - Zauzeti memorijsku lokaciju
  - Dodeliti vrednost članicama stukture koje sadrže informacije o podacima
  - Veza ka narednom elemntu liste se može inicijalno postaviti na NULL

```
/* kreiranje novog elementa */
    p = (struct element *)malloc(sizeof(struct element));

    if(p == NULL)
    {
        printf("Greska.\n");
        exit(0);
    }

    p->podatak = n;
    p->sledeci = NULL;
```

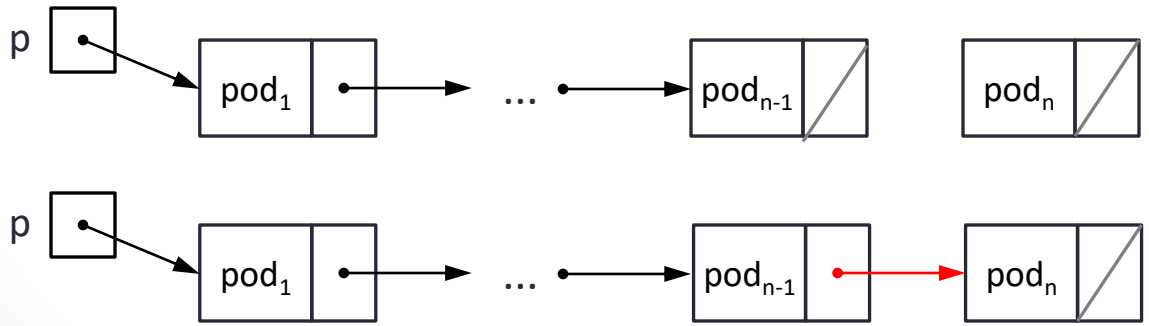
# Dodavanje elementa u listu

- Svim elementima liste se pristupa polazeći od prvog elementa
- Da bi rad sa listom bio moguć, ukoliko postoji, uvek je neophodno znati poziciju prvog elementa liste
- Pokazivač na prvi element liste se često naziva glava, *head*
- Kada je lista prazna (ne postoji) tada je vrednost glave NULL

# Dodavanje elementa u listu

- Pri dodavanju novog elementa liste na poslednje mesto u listi razlikuju se dva slučaja
  - Lista je prazna
    - U kom slučaju se nakon kreiranja novog elementa glava liste postavlja da pokazuje na taj element


  - Lista nije prazna
    - Prolaskom kroz listu se stiže do poslednjeg elementa i njegova veza dobija adresu dodatog elementa



# Prolazak kroz listu

- Pri radu sa listama često se javlja potreba za prolaženjem kroz elemente liste
  - Pri štampanju, traženju određenog elementa, traženju kraja liste i sl.
- Postupak za prolazak kroz listu je uvek isti
  - Postavljanje pomoćnog pokazivača na početak liste
  - Sve dok postoje elementi liste
    - Obraditi element liste
    - Pomeriti pokazivač na naredni

```
pom = p;
while (pom != NULL)
{
    OBRADA ELEMENTA LISTE
    pom = pom->sledeci;
}
```

# Primer.

- Napisati program koji kreira i ispisuje listu od N celih brojeva

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

struct element
{
    int podatak;
    struct element *sledeci;
};
```

# Primer.

```
/* funkcija za dodavanje novog elementa na kraj liste */
struct element *dodaj(struct element *p, int n)
{
    struct element *pom;

/* ako je postojeca lista prazna novi element se dodaje kao pocetni */
    if(p == NULL)
    {

/* kreiranje novog elementa */
        p = (struct element *)malloc(sizeof(struct element));
        if(p == NULL)
        {
            printf("Greska.\n"); exit(0);
        }

/* novom elementu se dodeljuje prosledjeni podatak */
        p->podatak = n;
        p->sledeci = NULL;
    }
...
```

# Primer.

```
...
    else
    {
        pom = p;

/* prolazak kroz postojecu listu da bi se dobio pokazivac na poslednji element */
        while (pom->sledeci != NULL)
            pom = pom->sledeci;

/* kreiranje novog elementa */
        pom->sledeci = (struct element *)malloc(sizeof(struct element));
        if(pom->sledeci == NULL)
        {
            printf("Greska.\n"); exit(0);
        }

/* novom elementu se dodeljuje prosledjeni podatak,
   a njegova adresa se upisuje kao link prethodnog elementa */
        pom = pom->sledeci;
        pom->podatak = n;
        pom->sledeci = NULL;
    }

    return (p);
}
```

# Primer.

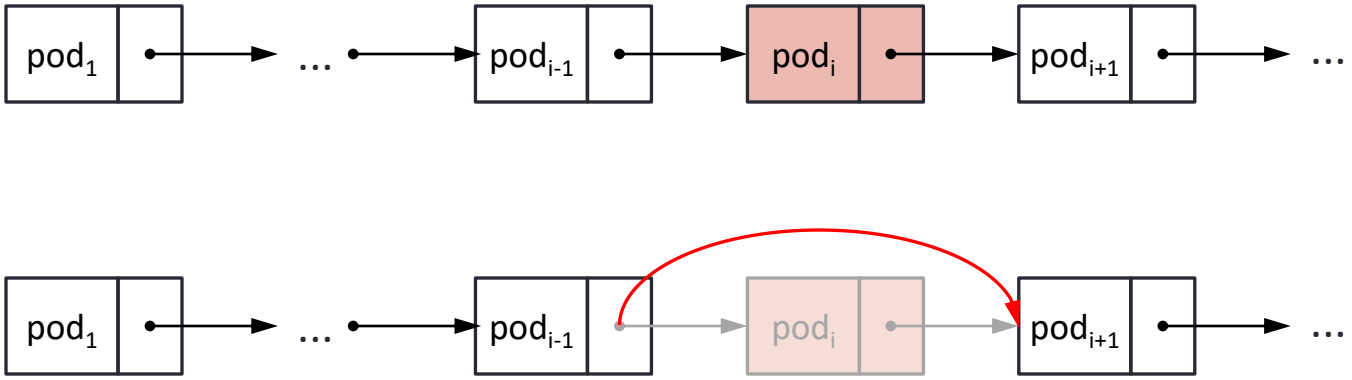
```
/* funkcija za stampanje liste */
void stampaj_listu( struct element *p )
{
    struct element *pom;
    pom = p;
    if(p != NULL)
    {
        do
        {
            printf("%d\t", pom->podatak);
            pom = pom->sledeci;
        } while (pom != NULL);
        printf("\n");
    }
    else
        printf("Lista je prazna.");
        printf("\n");
}
```

# Primer.

```
void main()
{
    int n,i;
    int x;
    struct element *glava = NULL;

    printf("Unesi broj elemenata u listi:\n");
    scanf("%d",&n);

    for(i = 0; i < n; i++)
    {
        printf("Unesi %d. element:\n", i+1);
        scanf("%d",&x);
        glava = dodaj(glava, x);
    }

    printf("Kreirana lista je:\n");
    stampaj_listu( glava );
}
```

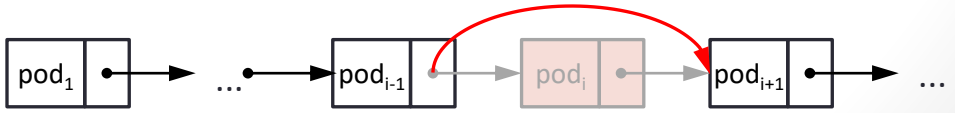
# Brisanje elementa iz liste

- Da bi element bio obrisan iz liste, potrebno je pronaći ga u listi i pri tome zapamtiti pokazivač na prethodni element
- Pre oslobađanja memorije koju zauzima element koji se briše, neophodno je da se prethodni element u listi „poveže“ na narednim u odnosu na posmatrani element



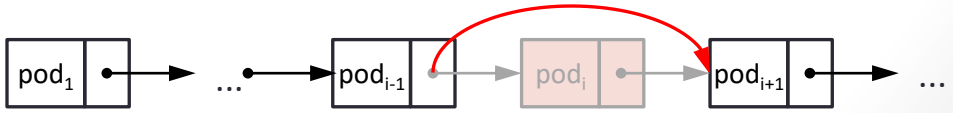
# Brisanje elementa iz liste

```
/* funkcija za brisanje zadatog elementa iz liste */
struct element *obrisi(struct element *p, int vrednost)
{
    struct element *prethodni, *trenutni ;

    if (p == NULL )
    {
        printf("Lista je prazna. \n");
    }
    else
    {
        prethodni = NULL;
        trenutni = p;
        while (trenutni != NULL &&  trenutni->podatak != vrednost)
        {
            prethodni = trenutni;
            trenutni = trenutni->sledeci;
        }
...
```



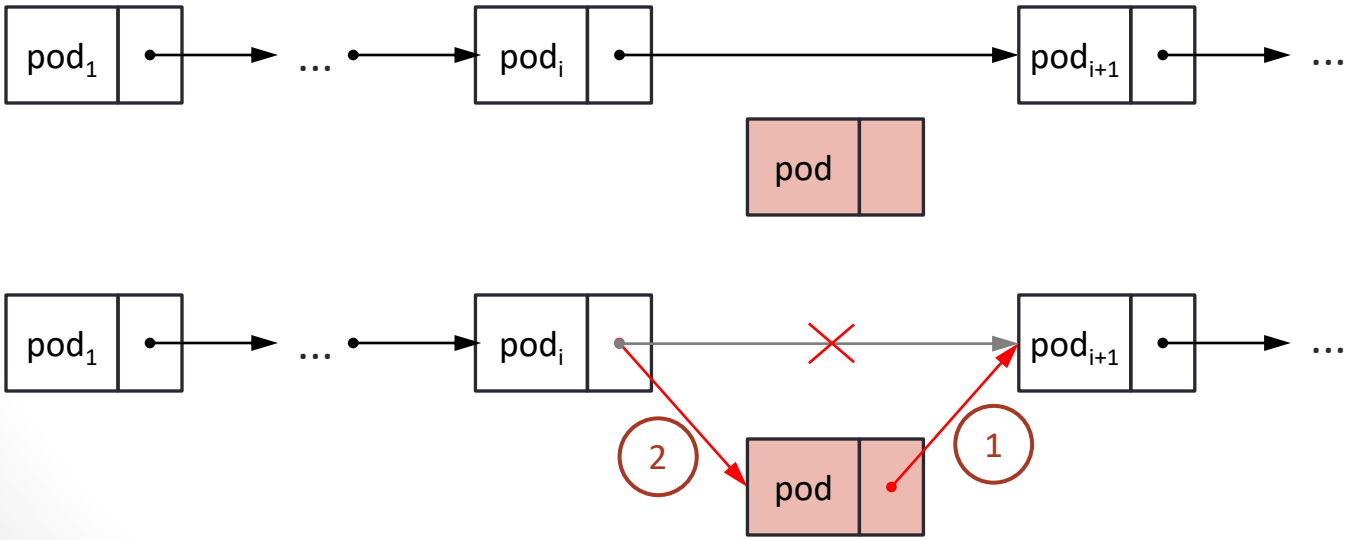
# Brisanje elementa iz liste

```
...
        if (trenutni == NULL)
        {
            printf("Element nije pronadjen\n");
            return (p);
        }

        if (prethodni == NULL)
        {
            p = trenutni->sledeci;
            free(trenutni);
        }
        else
        {
            prethodni->sledeci = trenutni->sledeci;
            free(trenutni);
        }
    }

return (p);
}
```



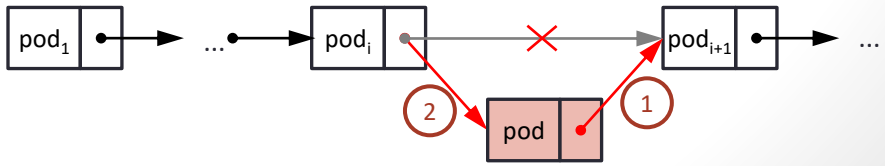
# Umetanje elementa u listu

- Pri umetanju elementa u listu neophodno je odrediti iza kog elementa će biti dodat (po poziciji, po vrednosti,...)
- Umetanje elemnta podrazumeva
  1. Kreiranje novog elementa
  2. Postavljanje veza novog elementa ka narednom
  3. Postavljanje veze prethodnika ka novom elementu



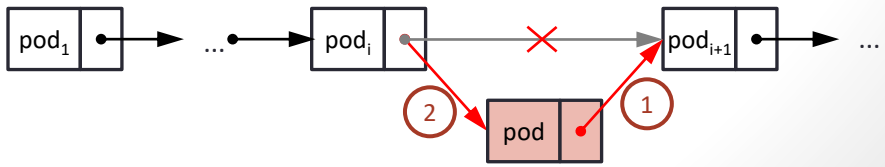
# Umetanje elementa u listu

```
/* funkcija za umetanje novog elementa na zadatu poziciju u listi */
struct element * umetni(struct element *p, int redni_broj, int vrednost)
{
    struct element *novi, *pom;
    int i;

    if (redni_broj < 0 || redni_broj > duzina(p))
    {
        printf("Greska! Zadati element ne postoji.\n");
        exit(0);
    }

    if (redni_broj == 0)
    {
        novi = (struct element *)malloc(sizeof(struct element));
        if (novi == NULL)
        {
            printf("Greska  pri  alociranju  memorije.  \n");
            exit(0);
        }
        novi->podatak = vrednost;
        novi->sledeci = p;
        p = novi ;
    }
...
```



# Umetanje elementa u listu

```
...
    else
    {
        pom = p;
        i = 1;
        while (i < redni_broj)
        {
            i = i+1;
            pom = pom->sledeci;
        }

        novi = (struct element *)malloc(sizeof(struct element));
        if (novi == NULL)
        {
            printf("Greska  pri  alociranju  memorije.\n");
            exit(0);
        }
        novi->podatak = vrednost;
        novi->sledeci = pom->sledeci;
        pom->sledeci = novi;
    }
    return (p);
}
```

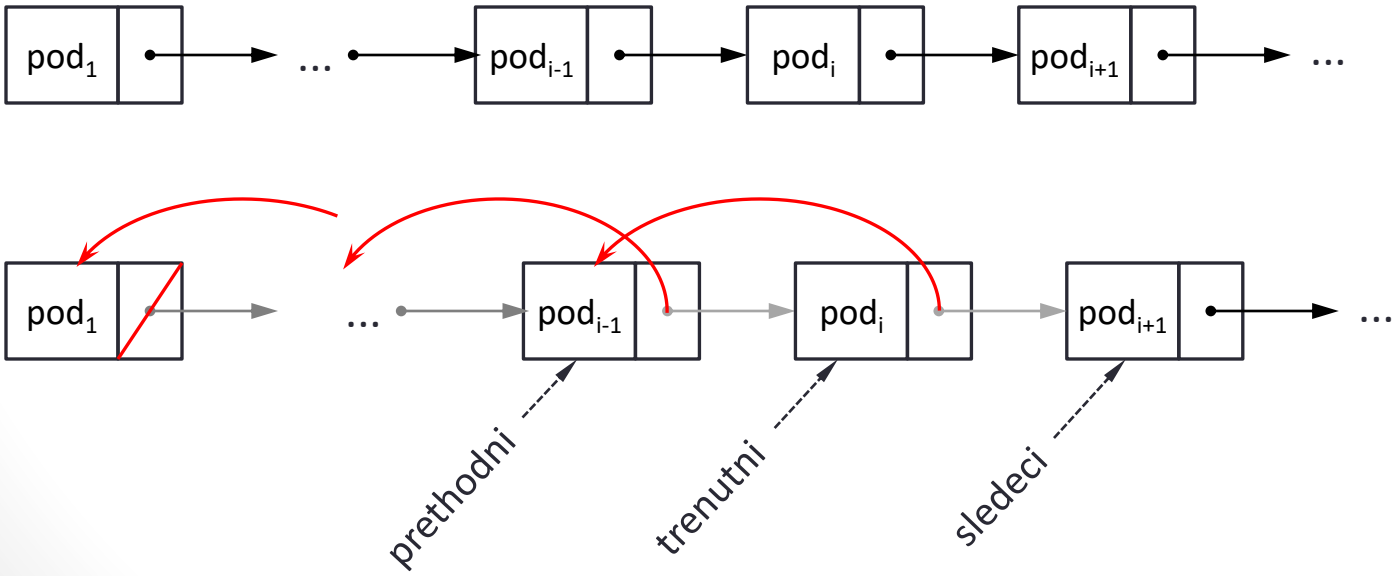


# Umetanje elementa u listu

```
/* funkcija za odredjivanje duzine liste */
int duzina(struct element *p)
{
    int broj = 0;
    while (p != NULL)
    {
        broj++;
        p = p->sledeci;
    }
    return (broj);
}
```

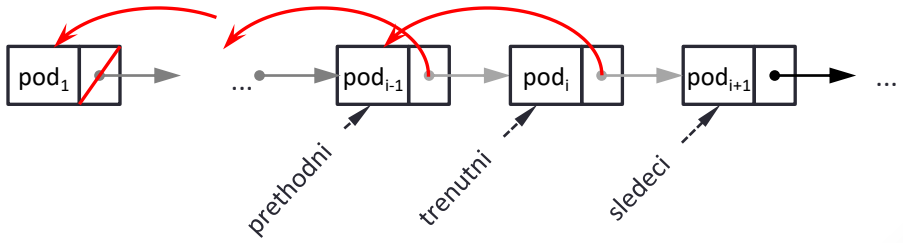
# Okretanje redosleda u listi

- Da bismo izvršili okretanje redosleda u listi neophodno je da za svaki element znamo pokazivače na njegovog prethodnika i sledbenika.
  - link trenutnog elementa da postavimo tako da pokazuje na prethodnika
  - trenutni element proglašavamo za prethodnika, a
  - njegovog sledbenika za trenutni.



# Okretanje redosleda u listi

```
prethodni = NULL;
while (trenutni != NULL)
{
    sledeci = trenutni->sledeci;
    trenutni->sledeci = prethodni;
    prethodni = trenutni;
    trenutni = sledeci;
}
```



# Pronalaženje elementa po poziciji

```
struct element *pronadji_kti(struct element *p, int k)
{
    struct element *tmp;
    int i = 0;
    tmp = p;
    while(tmp != NULL && ++i < k)
        tmp = tmp->sledeci;
    return (tmp);
}

main()
{
    ...
    struct element *tmp;
    int x;
    scanf("%d",&x);
    tmp = pronadji_kti(glava, x);
    if (tmp)
        printf("Pronadjen %d\n", tmp->podatak);
    else
        printf("Nema\n");
}
```

# Pronalaženje elementa po vrednosti

- Iterativno

```
struct element *pronadji_it(struct element *p, int vrednost)
{
    struct element *tmp;
    tmp = p;
    while(tmp != NULL && tmp->podatak != vrednost)
        tmp = tmp->sledeci;
    return (tmp);
}
```

- Rekurzivno

```
struct element *pronadji_rek(struct element *p, int vrednost)
{
    if (p == NULL || p->podatak == vrednost)
        return (p);
    return pronadji_rek(p->sledeci, vrednost);
}
```