

# *Programiranje 2*

## *8. predavanje*

Saša Singer

PMF – Matematički odsjek, Zagreb

# Sadržaj predavanja

- Samoreferencirajuće strukture (uvod):
  - Primjeri: Vezane liste. Binarna stabla.
- Vezane liste (detaljno):
  - Deklaracija vezane liste.
  - Kreiranje i uništavanje elemenata.
  - Ubacivanje i izbacivanje na početku liste.
  - Prolaz kroz listu. Broj elemenata i ispis liste.
  - Traženje elemenata u listi.
  - Ubacivanje i izbacivanje bilo gdje u listi.
  - Ubacivanje na kraj liste s pamćenjem zadnjeg.

# Samoreferencirajuće strukture. Vezane liste.

# Sadržaj

- Samoreferencirajuće strukture (uvod):
  - Pokazivač na strukturu. Operator strelica (->).
  - Samoreferencirajuće strukture.
  - Primjeri samoreferencirajućih struktura:
    - Vezane liste.
    - Binarna stabla.

# Pokazivač na strukturu — ponavljanje

Pokazivač na strukturu definira se isto kao i pokazivač na druge tipove objekata.

Primjer.

---

```
struct tocka {
    int x;
    int y;
} p1, *pp1 = &p1;
```

---

Varijabla `pp1` je

- pokazivač na strukturu `struct tocka`,
- inicijaliziran adresom strukture `p1`.

## Operator strelica (->) — ponavljanje

Kad imamo pokazivač na neku strukturu, članovima te strukture može se izravno pristupiti korištenjem primarnog operatora strelica (->).

● Asocijativnost operatora -> je  $L \rightarrow D$ .

Ako je ptvar pokazivač na strukturu, a clan je neki član te strukture, onda je:

●  $ptvar \rightarrow clan \iff (*ptvar).clan$

Primjer.

---

```
struct tocka p1, *pp1 = &p1;
pp1->x = 18;    /* Isto sto i (*pp1).x = 18; */
pp1->y = 27;    /* Isto sto i (*pp1).y = 27; */
```

---

## Strukture koje sadrže pokazivače

Znamo da **pokazivač** na **objekt** nekog tipa

- smije biti **element** polja.

Polje **pokazivača** smo koristili za **sortiranje** rječnika.

Slično je i kod struktura. **Pokazivač** na **objekt** nekog tipa

- smije biti **član** strukture.

To omogućava “**povezivanje**” objekata **razno–raznih** tipova, ovisno o **tipu pokazivača**.

Posebno, **dozvoljeno** je da **pokazivač**, koji je član strukture,

- “**pokazuje**” na **istu** takvu strukturu,

tj. da **struktura** sadrži **pokazivač** na “**samu sebe**”.

**Napomena.** **Struktura ne sadrži samu sebe!** To “ne ide”.

# Samoreferencirajuće strukture

Struktura koja sadrži jedan ili više članova, koji su

- pokazivači na strukturu tog istog tipa,

zove se samoreferencirajuća struktura — jer sadrži “pointer na samu sebe”.

Upravo ovakve “rekurzivno vezane” strukture su glavna korist od pokazivača kao članova strukture. One omogućavaju

- “povezivanje” objekata istog tipa — na razne načine, ovisno o broju i svrsi pokazivača.

Služe za implementaciju “rekurzivno” definiranih složenih tipova podataka — kao što su

- vezane liste i binarna stabla.



# Samoreferencirajuće strukture — primjeri

Kako izgledaju takve “rekurzivno vezane” strukture i kako se deklariraju?

Svaki element je struktura koja ima dva bitna dijela:

- nekakav “koristan” sadržaj — jedan ili više članova nekih tipova, i
- jedan ili više pokazivača na isti takav element (strukturu).

Članovi strukture koji sadrže pokazivače na tu strukturu

- obično imaju standardna imena koja sugeriraju značenje pokazivača.

Pogledajmo kako izgledaju elementi

- vezane liste i binarnog stabla.

## Element vezane liste

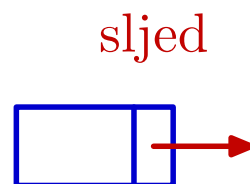
Element **vezane liste**, osim sadržaja, ima još

- **jedan** pokazivač na takav element — kojeg interpretiramo kao pokazivač na **sljedeći element** u listi.

“Standardna imena” za pripadni član strukture su

- **sljed**, **veza**, **next**, **link**.

Slika elementa:



Ako zamislimo cijelu **listu** istih ovakvih elemenata **iza** ovog, onda **pokazivač** možemo “**rekurzivno**” interpretirati i kao

- **listu sljedbenika** ovog elementa.

# Element binarnog stabla

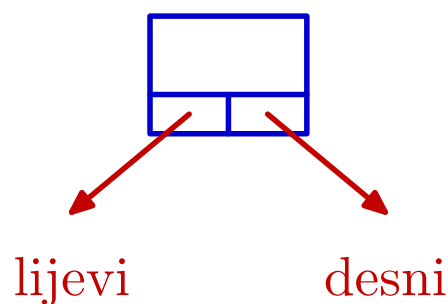
Element **binarnog stabla** obično se zove **čvor** stabla. Osim sadržaja, ima još

- **dva** pokazivača na takve elemente — koje interpretiramo kao pokazivače na **lijevo** i **desno dijete** tog čvora.

“Standardna imena” za pripadne članove strukture su

- **lijevi**, **desni**, ili **left**, **right**.

Slika elementa:



Ovdje **pokazivače** možemo “**rekurzivno**” interpretirati i kao

- **lijevo** i **desno podstablo** ovog elementa.

## Deklaracija elementa takve strukture

Kako se deklariraju elementi ovakvih struktura?

**Napomena.** Odmah koristimo `typedef` za deklaraciju

- tipa cijele strukture za `element`,

zato da **izbjegnemo** stalno pisanje riječi `struct`.

Uzmimo da se “**korisni**” sadržaj elementa sprema u **jednom** članu strukture.

- Neka se taj **član** strukture zove `info`, a

- njegov **tip** neka se zove `sadrzaj` (i ranije je deklariran).

Ovisno o vrsti strukture, u deklaraciju još treba **dodati** i

- jedan ili više **pokazivača** na **takav** element.

## Deklaracija elementa vezane liste

Deklaracija **tipa** za **element** **vezane liste** takvih podataka:

```
typedef struct _element {  
    sadrzaj info;           /* Sadrzaj. */  
    struct _element *sljed; /* Pokazivac. */  
} element;
```

Ova deklaracija izgleda “**rekurzivno**”, ali stvarno — **nije**.

- 🔴 Član **sljed** je **pokazivač** na **struct \_element**, a ne struktura.

Da nema znaka **\***, imali bismo pravu “**rekurziju**”, tj. pokušaj deklaracije strukture koja “sadrži **samu sebe**”, što (naravno) **nije dozvoljeno!**

# Deklaracija elementa vezane liste — napomene

Napomene uz deklaraciju:

---

```
typedef struct _element {
    sadrzaj info;           /* Sadrzaj. */
    struct _element *sljed; /* Pokazivac. */
} element;
```

---

U trenutku deklaracije pokazivača `sljed`,

• tip `struct _element` još nije potpuno određen.

Međutim, memorija potrebna za spremanje pokazivača na taj (ili neki drugi) tip — ne ovisi o tipu, pa je sve korektno.

Ovdje nam nužno treba ime strukture `_element`,

• za deklaraciju tipa pokazivača (taj tip moramo navesti).

## Primjer deklaracije elementa vezane liste

Primjer. Elementi **vezane liste** trebaju sadržavati **polje znakova** od **80** znakova. Pripadana deklaracija tipa je

```
typedef struct _element {
    char ime[80];           /* Sadržaj. */
    struct _element *next; /* Pokazivač. */
} element;
```

Ovdje koristimo ime **next** za **pokazivač** na **sljedeći element** liste.

Tipični primjer kombinacije hrvatskih i engleskih imena!

## Zadavanje vezane liste

Sad znamo kako izgledaju **elementi vezane liste**. Međutim, još nismo rekli kako izgleda **cijela** lista.

Po definiciji, **vezana lista** može biti **prazna**, tj. imati **nula** elemenata.

Zato se **vezana lista** **ne** zadaje **prvim** elementom, već

- 🔴 **pokazivačem** na **prvi** element, ako takav element **postoji**.
- 🔴 Lista je **prazna**, ako (i samo ako) je vrijednost tog **pokazivača = NULL**.

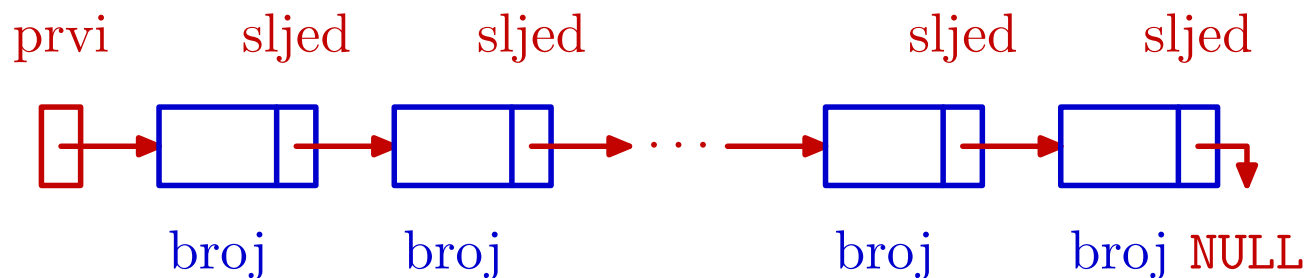
“Standardna imena” za taj **pokazivač** na **prvi** element liste su

- 🔴 **prvi, glava, first, head**.



## Izgled vezane liste

Slika nepravne vezane liste elementa:



Ključne stvari koje treba zapamtiti:

- lista se zadaje pokazivačem na prvi element — ovdje je to pokazivač s imenom prvi,
- svaki element ima pokazivač na sljedeći element u listi,
- taj pokazivač (član sljed) možemo interpretirati i kao listu sljedbenika tog elementa,
- zadnji element u listi ima praznu listu sljedbenika, tj. njegov sljed = NULL.

## Deklaracija pokazivača na element vezane liste

Pokazivač na prvi element liste možemo definirati ovako:

```
struct _element *prvi;    /* Pocetak liste. */
```

To može i prije deklaracije tipa element.

Nakon deklaracije tipa element smijemo pisati i

```
element *prvi;    /* Pocetak liste. */
```

Najbolje je odmah uvesti i deklaraciju tipa za pokazivače na elemente liste. Recimo, ovako:

```
typedef struct _element *lista;
```

## Deklaracija tipa za pokazivač na element liste

Taj oblik deklaracije **tipa** za **pokazivače** na **elemente** liste

```
typedef struct _element *lista;
```

**smijemo** napisati i **prije** deklaracije tipa **struct \_element**.

Međutim, “skraćenu” deklaraciju **tipa**

```
typedef element *lista;
```

**ne smijemo** napisati **prije** deklaracije tipa **element**, već samo **poslije**.

Zato, odmah na početku, pišemo **prvi** oblik (sa **struct**), a **svagdje** dalje koristimo **tip lista** — i u deklaraciji tipa **element**.

## Sve deklaracije tipova za vezanu listu

Sve potrebne deklaracije tipova za vezanu listu imaju sljedeći oblik:

```
typedef struct _element *lista;

typedef struct _element {
    sadrzaj info;           /* Sadržaj. */
    lista sljed;           /* Pokazivac. */
} element;

...
lista prvi = NULL;       /* Pocetak liste. */
```

Dodatno, u zadnjem redu definiramo pokazivač prvi (“početak liste”) i inicijaliziramo ga na NULL — prazna lista.

## Vezana lista kao dinamička struktura

Nakon svih ovih deklaracija, mogli bismo **definirati** i nekoliko **varijabli** za elemente liste i **povezati** ih u jednu listu.

```
element a, b, c;    /* Elementi liste. */
...
prvi = &a;
strcpy(a.ime, "prvi"); /* NE: a.ime = "prvi"; */
a.sljed = &b;
strcpy(b.ime, "drugi");
b.sljed = &c;
strcpy(c.ime, "treći");
c.sljed = NULL;
```

Međutim, to se **nikad** tako **ne radi** u praksi.

# Vezana lista kao dinamička struktura (nastavak)

Vezana lista i sve slične strukture su **idealne** za

- **dinamičku** promjenu **veza** među elementima,
- **dodavanje** novih i **izbacivanje** postojećih elemenata.

Zato se elementi takvih struktura uvijek kreiraju **dinamičkom alokacijom memorije**.

---

```
prvi = (lista) malloc(sizeof(element));
if (prvi == NULL) {
    printf("Alokacija nije uspjela.\n");
    exit(EXIT_FAILURE);    /* exit(1); */
}
strcpy(prvi->ime, "prvi");
prvi->sljed = NULL;
```

---

# Deklaracija elementa binarnog stabla

Deklaracija **tipa** za **element binarnog stabla** naših podataka:

---

```
typedef struct _cvor {
    sadrzaj info;           /* Sadrzaj. */
    struct _cvor *lijevi;   /* Pokazivac. */
    struct _cvor *desni;    /* Pokazivac. */
} cvor;
...
cvor *korijen;             /* Pokazivac na korijen. */
```

---

“Početni” element **stabla** obično se zove **korijen**. On **nije** “dijete” niti jednog elementa, tj. **nema** “roditelja” u stablu.

U **dinamičkoj** strukturi, to je **pokazivač** na “početni” element stabla.

## Deklaracije tipova za binarno stablo

Možemo odmah uvesti i **tip** za **pokazivače**, ovog puta na engleskom:

```
typedef struct _treenode *Treenode;

typedef struct _treenode {
    ... /* Sadržaj. */
    Treenode left; /* Pokazivac. */
    Treenode right; /* Pokazivac. */
} Treenode;

...
Treenode root; /* Pokazivac na korijen. */
```

Po definiciji, i **binarno stablo** može biti **prazno**, tj. na samom početku je **root = NULL**.



# Vežane liste

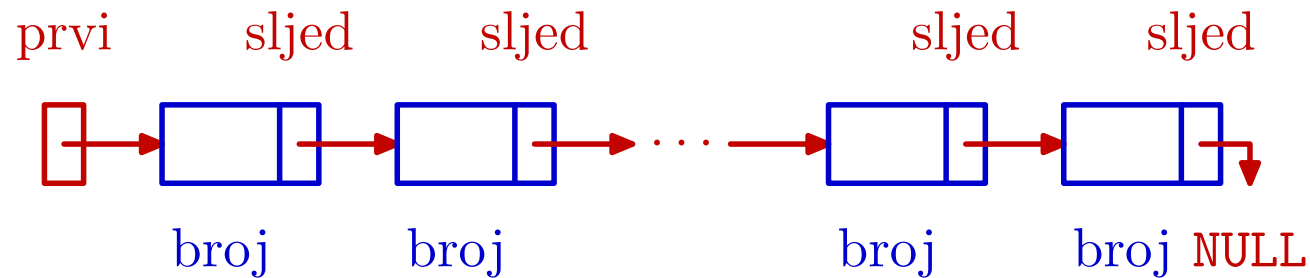
# Sadržaj

- **Vezane liste** (detaljno):
  - Deklaracija vezane liste.
  - Kreiranje i uništavanje elemenata.
  - Ubacivanje na početak liste.
  - Izbacivanje s početka liste.
  - Prolaz kroz listu. Broj elemenata i ispis liste.
  - Traženje elemenata u listi.
  - Ubacivanje i izbacivanje bilo gdje u listi.
  - Ubacivanje na kraj liste s pamćenjem zadnjeg.

## Vezana lista — uvod

Vezana lista je način spremanja **uređenog niza** podataka (kao i polje). Za razliku od polja,

- uređaj nije fizički, već je zadan pokazivačima.



Vezana lista je “**fleksibilna**” struktura (za razliku od polja). Zgodna je za spremanje **uređenog niza**, kod kojeg se **broj** i **uređaj** podataka može **dinamički** mijenjati:

- dodavanje** novih, **brisanje** postojećih elemenata,
- promjena veza** (uređaja).

Probajte to napraviti na polju!

# Operacije nad vezanim listama

Osnovne operacije nad vezanom listom su:

- kreiranje i uništavanje elemenata,
- dodavanje novog elementa u listu,
- brojanje elemenata liste,
- ispis liste,
- pretraživanje liste (prolaz kroz listu),
- izbacivanje ili brisanje elementa iz liste,
- spajanje (konkatenacija) dvije liste,
- sortiranje liste.

Sve ove operacije implementirat ćemo u sljedećim primjerima. Neke od njih i na više načina.

## Modelni primjer liste

Kroz sve primjere koristimo *vezanu listu cijelih brojeva*:

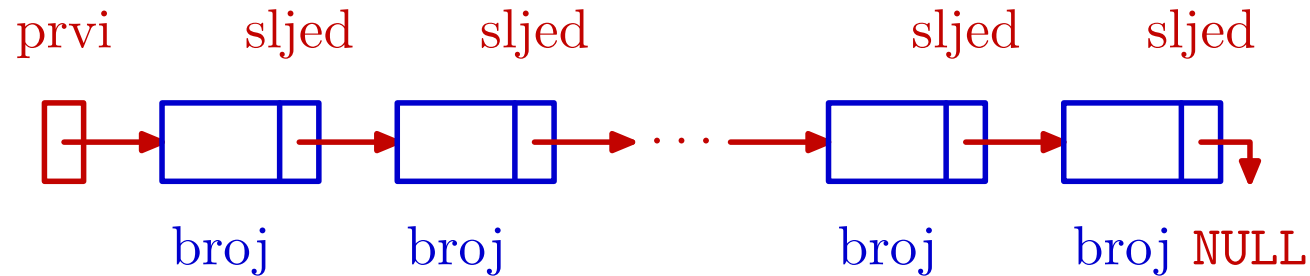
---

```
    /* Tip za pokazivac na element liste. */  
  
typedef struct _element *lista;  
  
    /* Tip za element liste. */  
  
typedef struct _element {  
    int broj;           /* Sadrzaj je broj. */  
    lista sljed;       /* Pokazivac na sljedeci */  
} element;           /* element u listi. */  
  
lista prvi;          /* Pokazivac na pocetak liste. */
```

---

# Modelni primjer liste (nastavak)

Slika takve vezane liste:



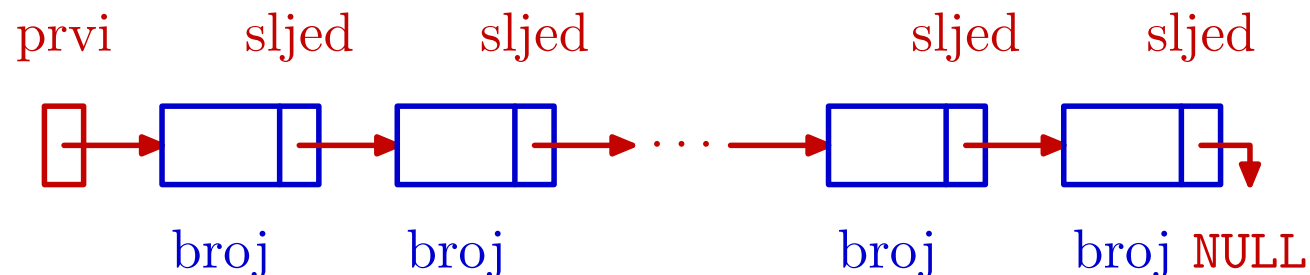
Operacije **dodavanja** i **izbacivanja** elemenata

☛ **najlakše** se rade na **početku** liste,  
zbog **sekvencijalnog** pristupa elementima.

**Pristup** bilo kojem **elementu** liste (ako lista nije prazna),

- ☛ moguć je **samo** preko **pokazivača prvi**,
- ☛ a onda se treba "**prošetati**" do **elementa**.

# Pristup elementima — općenito



Elementi liste su “anonimni” — nemaju imena kao obične varijable. To znači da je pristup bilo kojem elementu moguć

- samo indirektno — preko nekog pokazivača na njega.

Ime takvog elementa je oblika `nešto->`, tj.

- “element na koji pokazuje pokazivač taj-i-taj (`nešto`)”.

Zato sve funkcije koje nešto rade s elementima vraćaju

- pokazivač na element, tj. objekt tipa `lista`.

## Pristup elementima u vezanoj listi

Slično vrijedi i u **vezanoj listi**, kao **sekvencijalnoj** strukturi elemenata. Ako lista **nije** prazna, tj. ako je **prvi != NULL**,

- preko **pokazivača prvi** možemo **pristupiti** samo **prvom** elementu (**prvi->**).

Za **pristup** svim **ostalim** elementima u listi — **iza** prvog,

- moramo **krenuti** od **prvi** i “**doći**” do njegovog **prethodnika** u listi, a onda iskoristiti njegov **sljed**.

Ta “**šetnja**” se radi **pomoćnim** pokazivačem **pom**, jer

- pokazivač prvi** **uvijek** pokazuje na **početak** liste!

Zato sve **funkcije** koje nešto rade s **listom** vraćaju

- pokazivač** na **početak** liste, čak i kad se on **ne mijenja**.



# Osnovne operacije u vezanoj listi

# Kreiraj element

novi

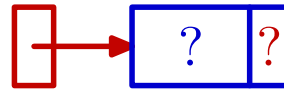


NULL

```
lista novi = NULL;
```

# Kreiraj element

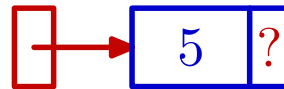
novi



```
lista novi = NULL;  
novi = (lista) malloc(sizeof(element));
```

# Kreiraj element

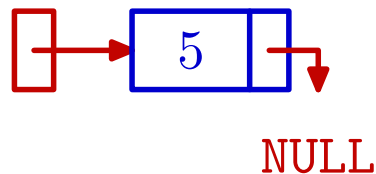
novi



```
lista novi = NULL;  
novi = (lista) malloc(sizeof(element));  
novi->broj = broj;
```

# Kreiraj element

novi



```
lista novi = NULL;  
novi = (lista) malloc(sizeof(element));  
novi->broj = broj;  
novi->sljed = NULL;
```

## *Funkcija* kreiraj\_novi

```
lista kreiraj_novi(int broj)
{
    lista novi = NULL;
    novi = (lista) malloc(sizeof(element));
    if (novi == NULL) {
        printf("Alokacija nije uspjela.\n");
        exit(EXIT_FAILURE);    /* exit(1); */
    }
    novi->broj = broj;
    novi->sljed = NULL;

    return novi;
}
```

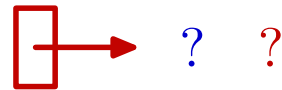
# Obriši element

stari



# Obriši element

stari



```
free(stari);
```



# Obriši element

stari



NULL

```
free(stari);  
stari = NULL;
```

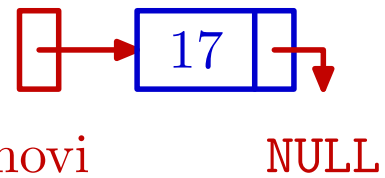
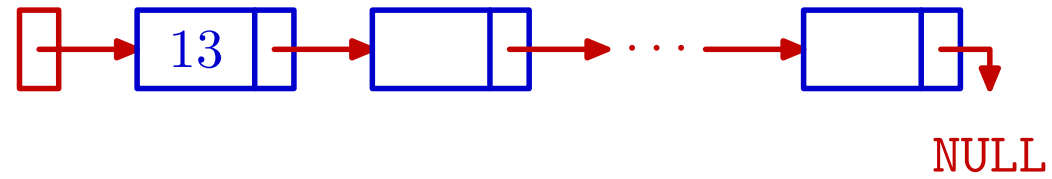
## *Funkcija* obrisi\_element

```
lista obrisi_element(lista stari)
{
    free(stari);

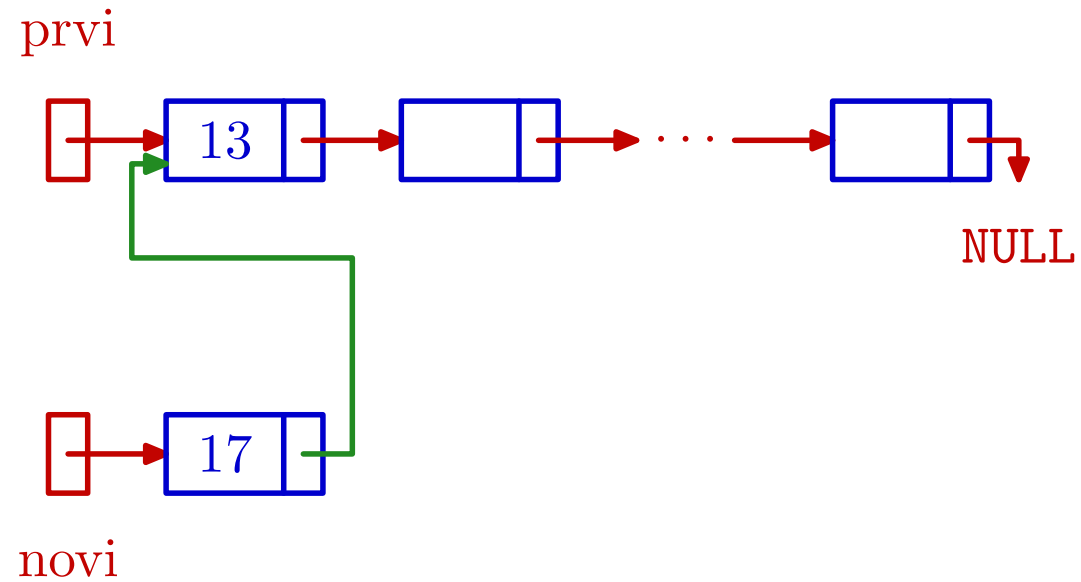
    return NULL;    /* Umjesto stari = NULL; */
}
```

# Ubaci na početak

prvi

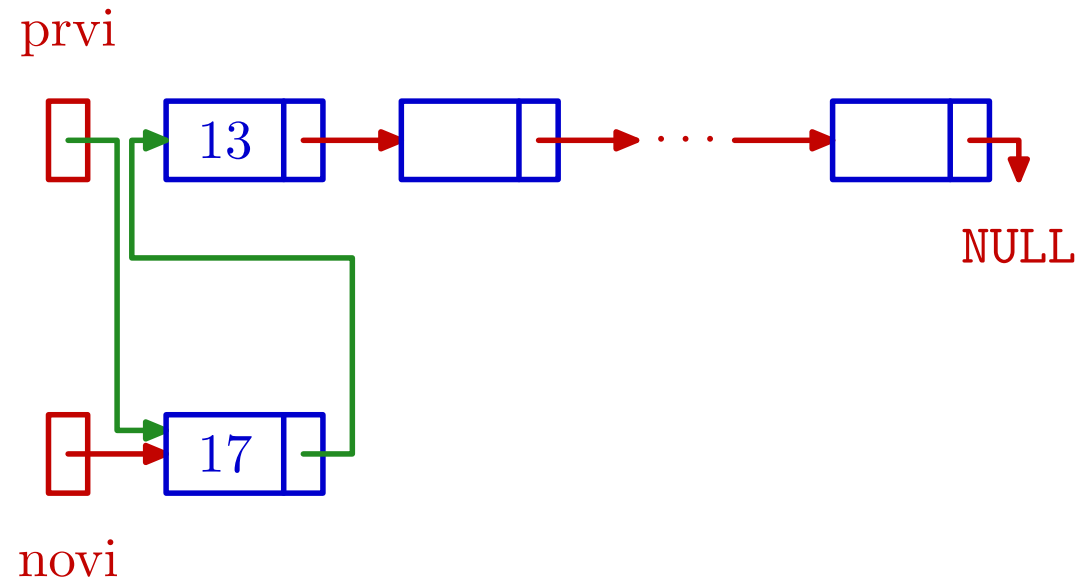


# Ubaci na početak



```
novi->sljed = prvi;
```

# Ubaci na početak



```
novi->sljed = prvi;  
prvi = novi;
```

## *Funkcija* ubaci\_na\_pocetak

```
lista ubaci_na_pocetak(lista prvi, lista novi)
{
    /* Ne provjerava novi != NULL. */
    novi->sljed = prvi;
    prvi = novi;

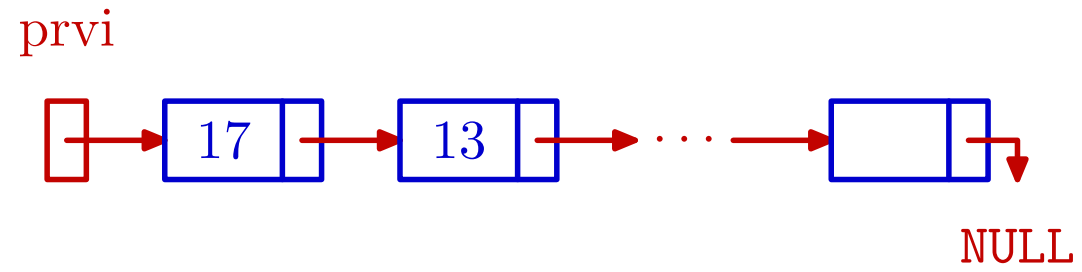
    return prvi;
}
```

Ideja **poziva** za **sve** funkcije za rad s listom je

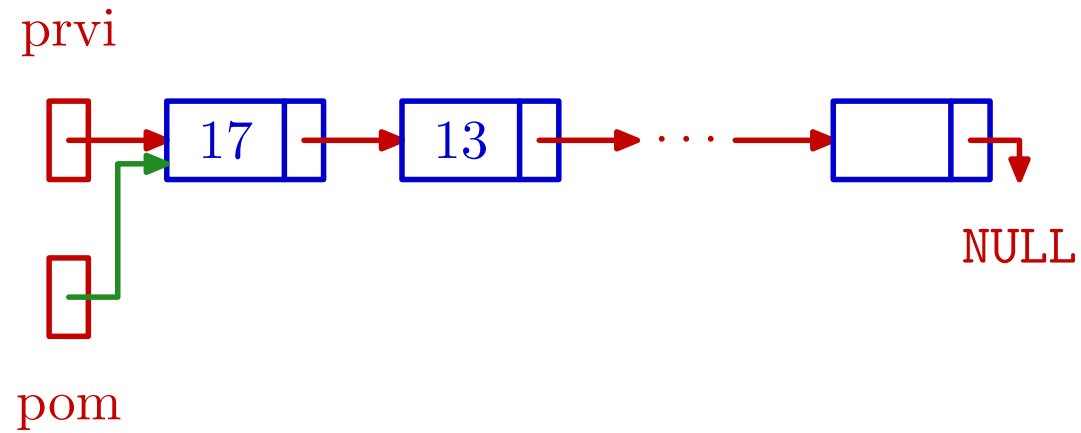
```
prvi = funkcija_na_listi(prvi, ...);
```

s tim da je **dozvoljeno** da je lista **prazna** (na ulazu i izlazu).

# Obriši prvog



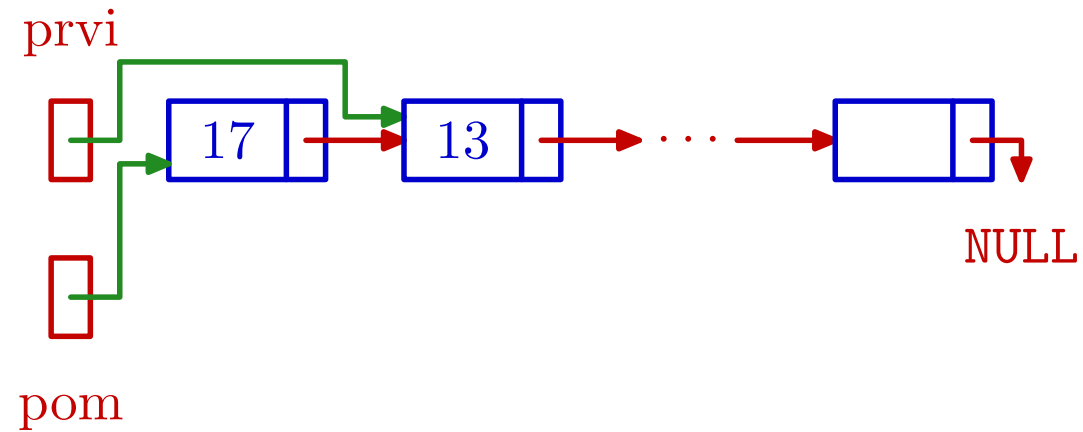
# Obriši prvog



```
pom = prvi;
```

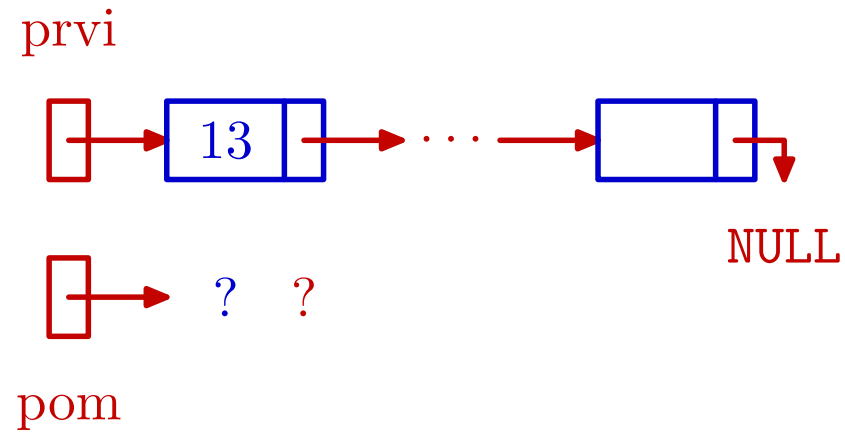


# Obriši prvog



```
prvi = pom;  
prvi = prvi->sljed;
```

# Obriši prvog



```
prvi = prvi;  
prvi = prvi->sljed;  
free(prvi);
```

## *Funkcija* obrisi\_prvog

```
lista obrisi_prvog(lista prvi)
{
    lista pom;

    if (prvi != NULL) {
        pom = prvi;
        prvi = prvi->sljed;
        free(pom);
        /* Ne treba pom = NULL; */
    }

    return prvi;
}
```

# Kreiraj novi element i ubaci ga na početak

## Operacije

- `kreiraj_novi` element sa zadanim sadržajem (brojem) i
- ubaci ga odmah na **početak** zadane liste — poziv funkcije `ubaci_na_pocetak`,

ima smisla **spojiti** u jednu operaciju — funkciju

- `kreiraj_sprijeda`.

**Prednost:** “pomoćni” pokazivač `novi` tad postaje

- **lokalni** objekt u toj funkciji — ne treba nam izvana.

## *Funkcija* kreiraj\_sprijeda

```
lista kreiraj_sprijeda(lista prvi, int broj)
{
    lista novi = NULL;
    novi = (lista) malloc(sizeof(element));
    if (novi == NULL) {
        printf("Alokacija nije uspjela.\n");
        exit(EXIT_FAILURE);    /* exit(1); */
    }
    novi->broj = broj;
    novi->sljed = prvi;

    /* Ne treba prvi = novi, vec samo ovo: */
    return novi;
}
```

## *Funkcija* obrisi\_listu

Brisanje **cijele** liste  $\iff$  u petlji `while (prvi != NULL)` ponavljamo `prvi = obrisi_prvog(prvi);` pa dobijemo:

---

```
lista obrisi_listu(lista prvi)
{
    lista pom;

    while (prvi != NULL) {
        pom = prvi;
        prvi = prvi->sljed;
        free(pom);
    }
    return NULL;    /*  $\iff$  return prvi; */
}
```

---

## *Funkcija* broj\_elemenata

Broj elemenata u listi treba *izračunati*, vrlo slično kao kod stringova — “šetnjom” do kraja liste.

---

```
int broj_elemenata(lista prvi)
{
    lista pom;
    int brojac = 0;

    for (pom = prvi; pom != NULL; pom = pom->sljed)
        ++brojac;

    return brojac;
}
```

---

## *Funkcija* ispisi\_listu

```
void ispisi_listu(lista prvi)
{
    lista pom;
    int brojac = 0;

    for (pom = prvi; pom != NULL; pom = pom->sljed)
    {
        printf(" Element %2d, broj = %2d\n",
              ++brojac, pom->broj);
    }

    return;
}
```



## *Funkcija* trazi\_broj

Funkcija za **traženje** zadanog **broja** — vraća

- **pokazivač** na **prvi** element koji sadrži zadani **broj**, ili **NULL**, ako takvog elementa **nema** u listi.

Uočite “**skraćeno**” računanje uvjeta u **while**.

```
lista trazi_broj(lista prvi, int broj)
{
    lista pom = prvi;

    while (pom != NULL && pom->broj != broj)
        pom = pom->sljed;

    return pom;
}
```

## *Funkcija* trazi\_zadnji

Funkcija vraća **pokazivač** na **zadnji** element u listi ( $\Leftrightarrow$  njegov **sljed** je **NULL**), ili **NULL**, ako takvog elementa **nema**.

---

```
lista trazi_zadnji(lista prvi)
{
    lista pom;

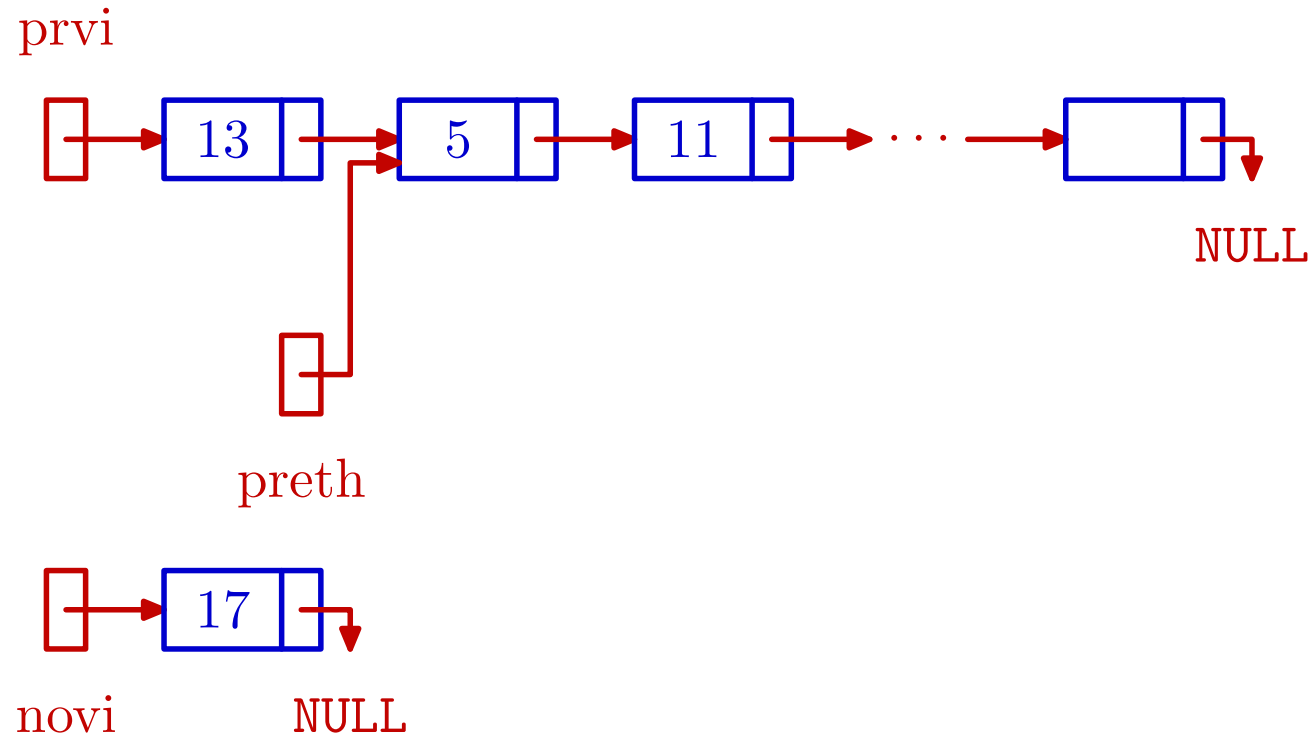
    if (prvi == NULL) return NULL;

    for (pom = prvi; pom->sljed != NULL;
         pom = pom->sljed);

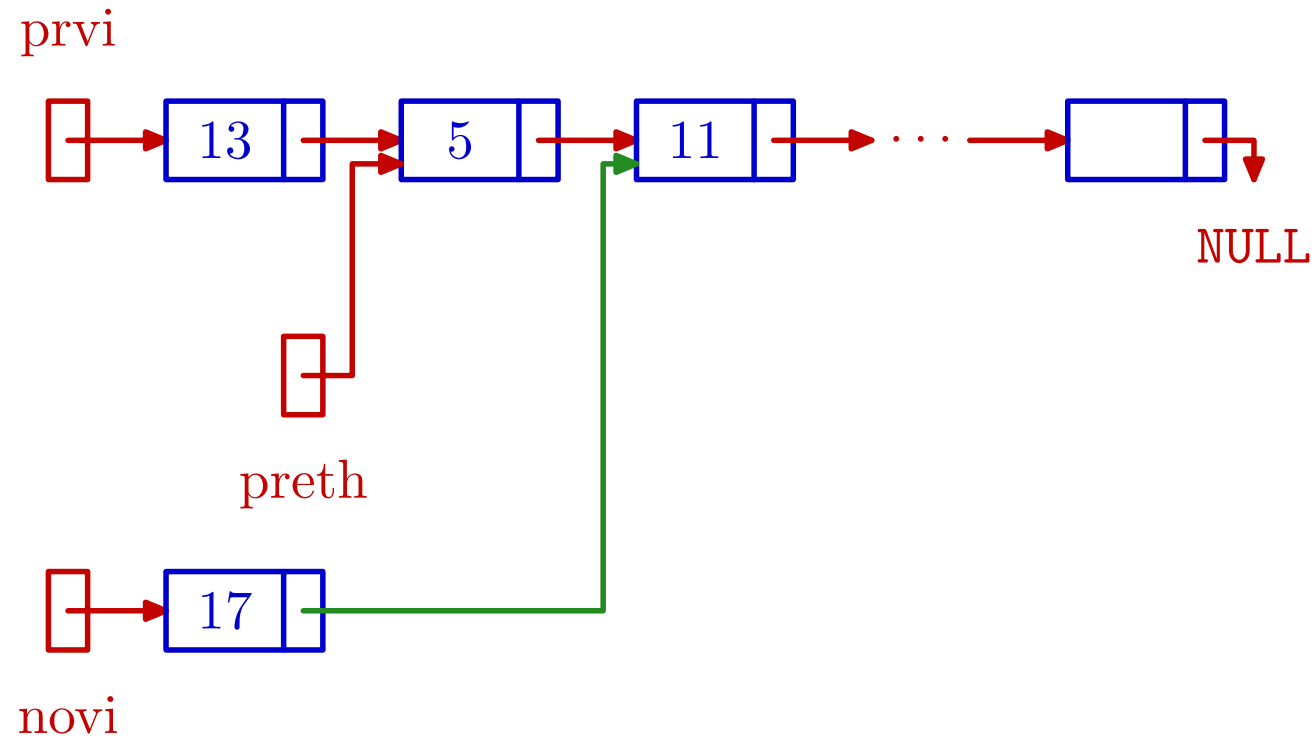
    return pom;
}
```

---

# Ubaci bilo gdje iza prvog

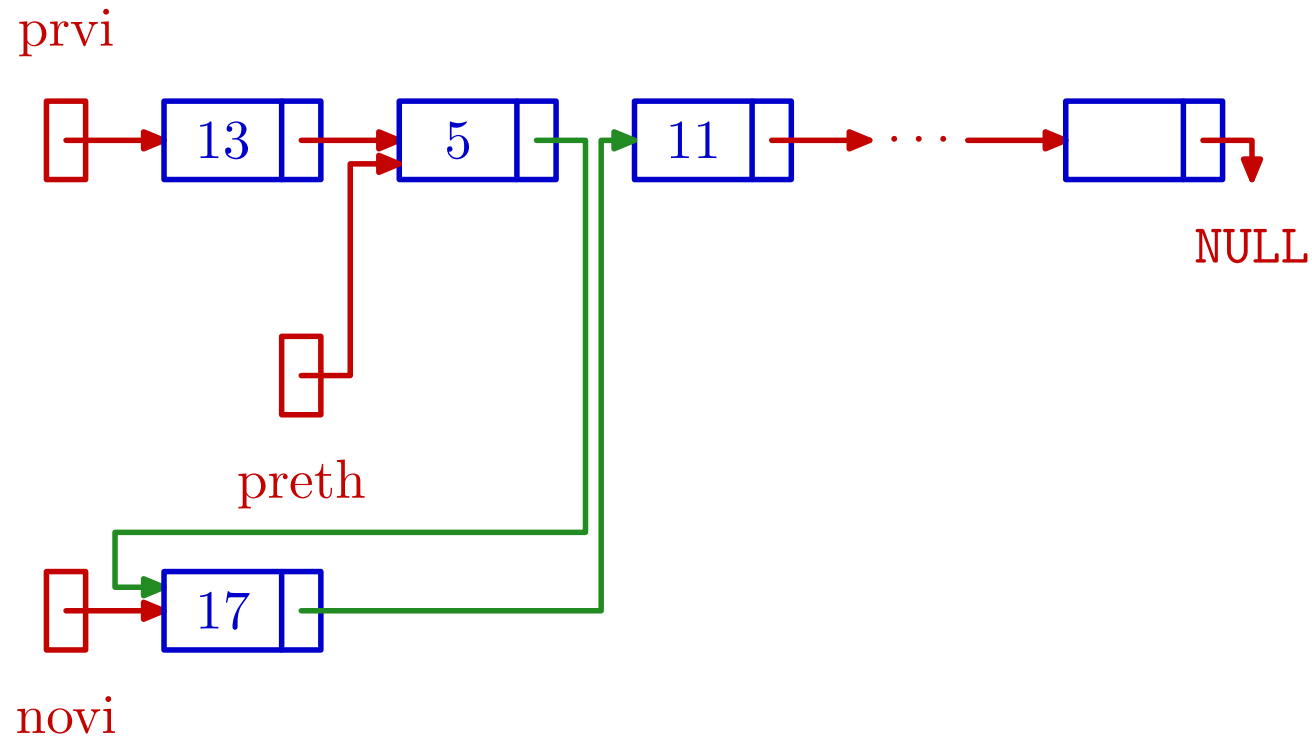


# Ubaci bilo gdje iza prvog



```
novi->sljed = preth->sljed;
```

# Ubaci bilo gdje iza prvog



```
novi->sljed = preth->sljed;  
preth->sljed = novi;
```

## *Funkcija* ubaci\_iza

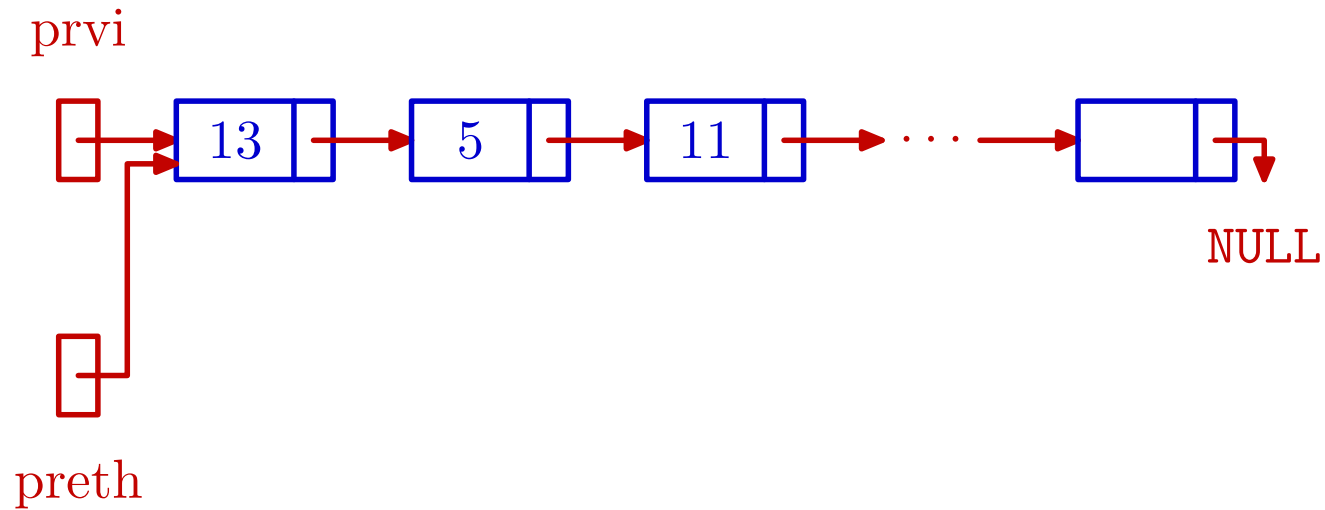
```
lista ubaci_iza(lista prvi, lista preth, lista novi)
{
    /* Ne provjerava novi != NULL. */
```

## *Funkcija* ubaci\_iza — *nastavak*

```
/* Ako je preth == NULL, ubacujemo na pocetak. */  
  
if (preth == NULL) {  
    novi->sljed = prvi;  
    prvi = novi;  
}  
else {  
    novi->sljed = preth->sljed;  
    preth->sljed = novi;  
}  
  
return prvi;  
}
```

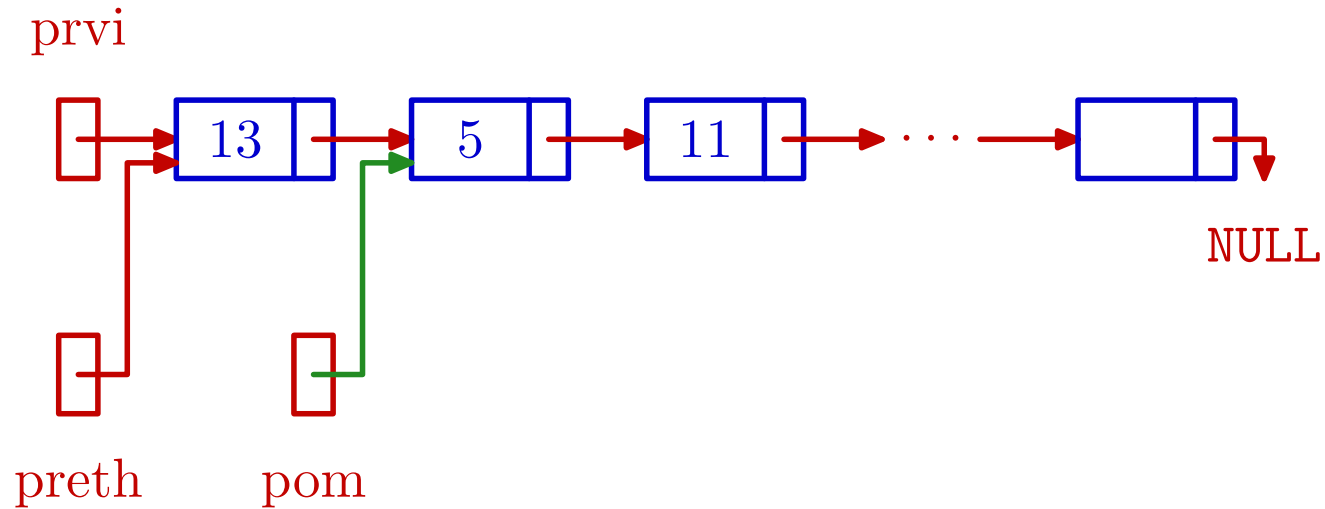
---

# Obriši bilo gdje iza prvog



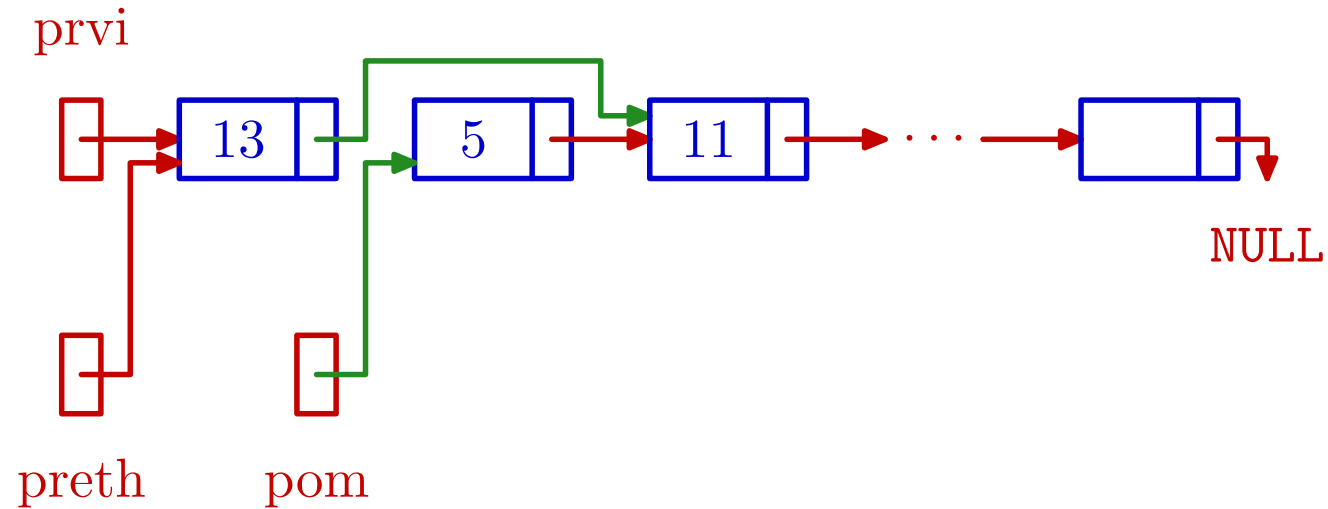


# Obriši bilo gdje iza prvog



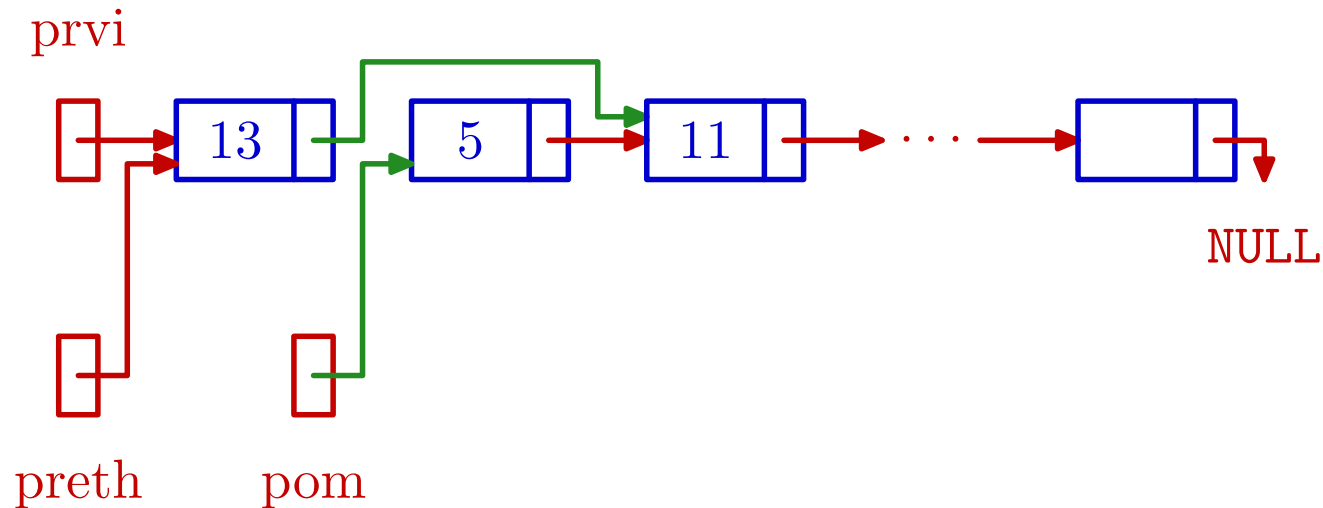
```
preth->sljed = pom;
```

# Obriši bilo gdje iza prvog



```
preth->sljed = pom->sljed;
```

## Obriši bilo gdje iza prvog



```
    pom = preth->sljed;  
    preth->sljed = pom->sljed;
```

Ako **izbačeni** element (na **pom**) zaista želimo “**obrisati**”, onda

```
    pom = obrisi_element(pom);   ili samo   free(pom);
```

No, s tim elementom možemo raditi i druge operacije.

## *Funkcija* obrisi\_iza

```
lista obrisi_iza(lista prvi, lista preth)
{
    lista pom;
```

## *Funkcija* obrisi\_iza — *nastavak*

```
/* Ako je preth == NULL, brisemo prvi element. */  
  
    if (preth == NULL) {  
        pom = prvi;  
        prvi = prvi->sljed;  
    }  
    else {  
        pom = preth->sljed;  
        preth->sljed = pom->sljed;  
    }  
    free(pom);  
  
    return prvi;  
}
```

## Funkcija `obrisi_iza` — komentari

Ovaj oblik funkcije `obrisi_iza` ima dva nedostatka:

- 🔴 `ne` provjerava je li ulazna lista `prazna`, tj. `ne` testira da li na početku vrijedi `prvi == NULL`,
- 🔴 `ne` pazi na `kraj` liste, ako je `preth == zadnji`, tj. `preth->sljed == NULL`.

Uvjerite se da u `oba` slučaja funkcija `ne radi` dobro!

Što sve treba `popraviti`?

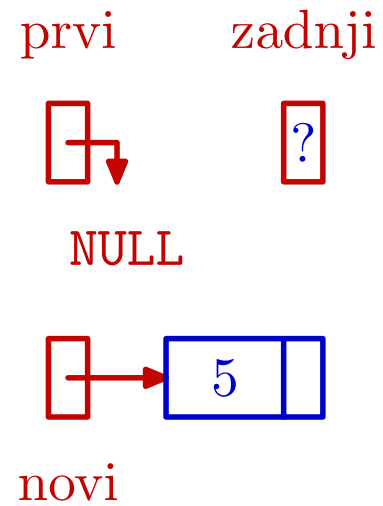
- 🔴 Pogledati `obrisi_iza_pazljivo` u `list_lib.c`.

Još “`pedantnija`” varijanta:

- 🔴 `provjerava` da li pokazivač `preth` zaista pokazuje na `neki` element liste zadane pokazivačem `prvi`.

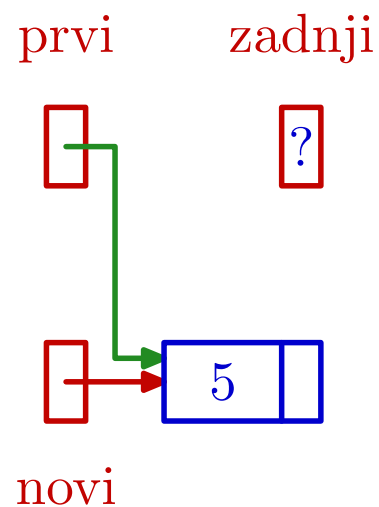
# Ubaci na kraj s pamćenjem zadnjeg

Dodaj u praznu listu:



## Ubaci na kraj s pamćenjem zadnjeg

Dodaj u praznu listu:

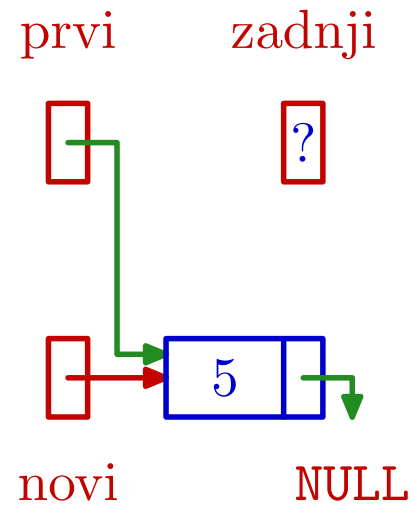


```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
```



# Ubaci na kraj s pamćenjem zadnjeg

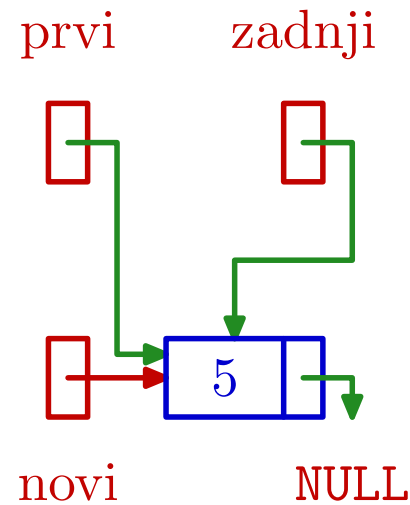
Dodaj u praznu listu:



```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
novi->sljed = NULL;
```

# Ubaci na kraj s pamćenjem zadnjeg

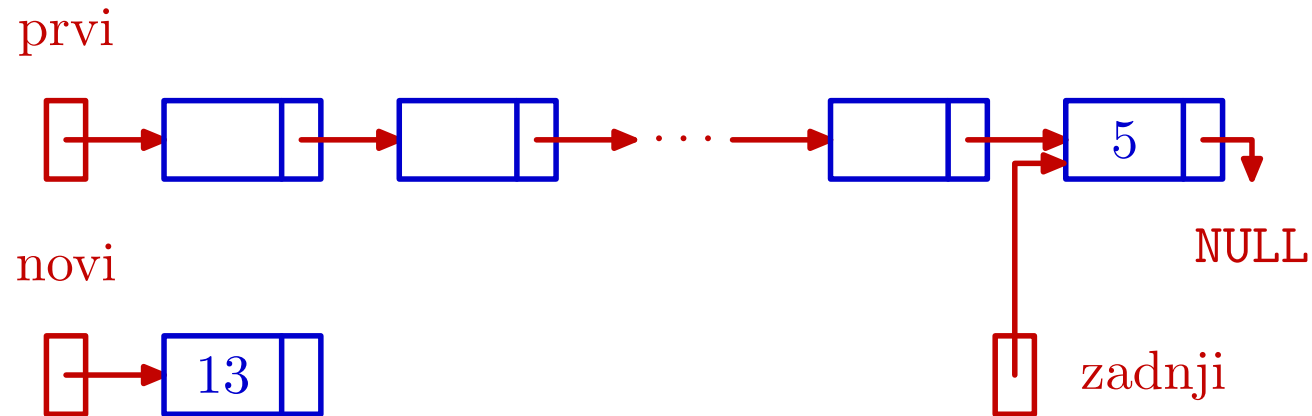
Dodaj u praznu listu:



```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
novi->sljed = NULL;
zadnji = novi;
```

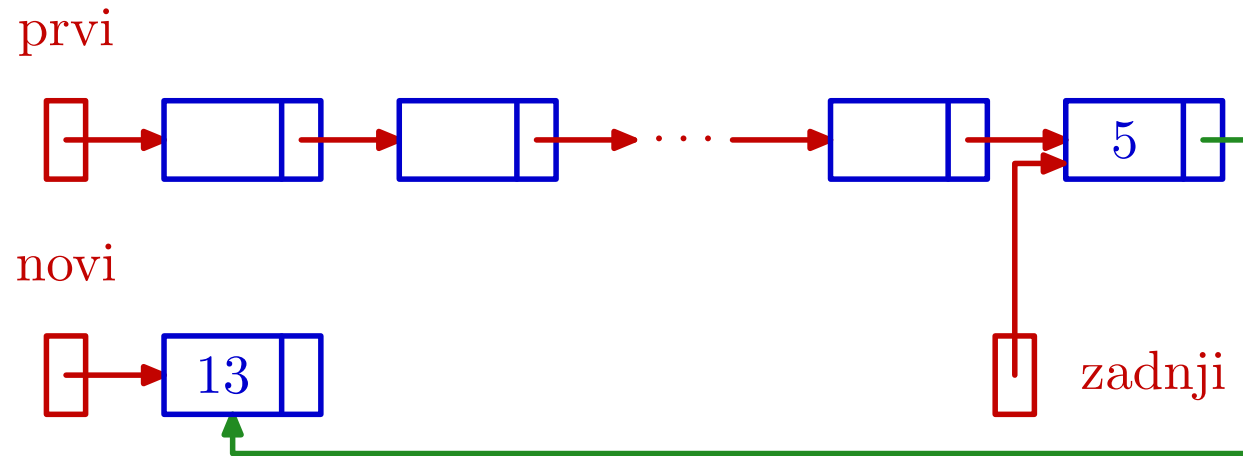
# Ubaci na kraj s pamćenjem zadnjeg

Situacija nakon barem jednog ubacivanja na kraj:



# Ubaci na kraj s pamćenjem zadnjeg

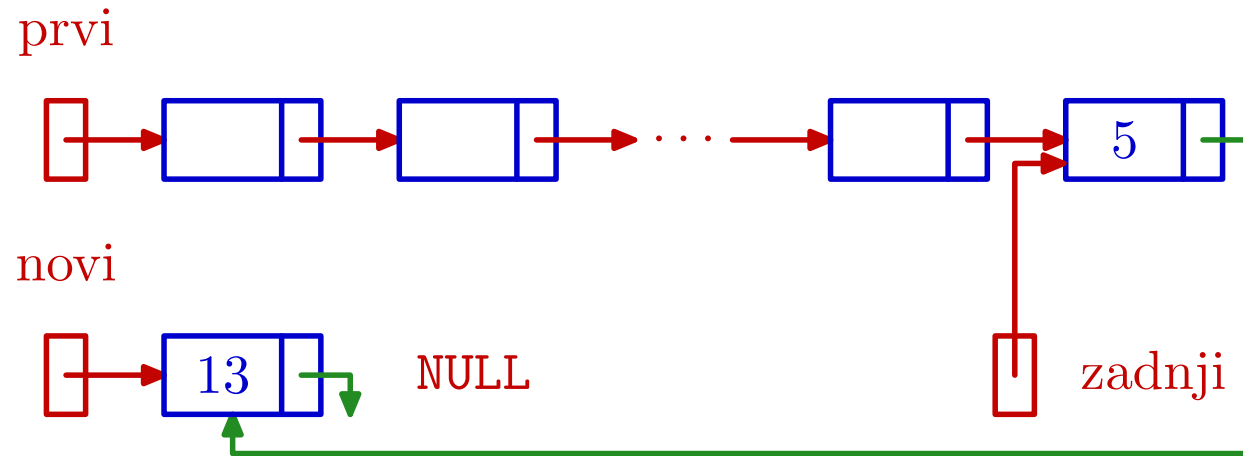
Situacija nakon barem jednog ubacivanja na kraj:



```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
```

# Ubaci na kraj s pamćenjem zadnjeg

Situacija nakon barem jednog ubacivanja na kraj:



```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
novi->sljed = NULL;
```

# Ubaci na kraj s pamćenjem zadnjeg

Situacija nakon barem jednog ubacivanja na kraj:

prvi



novi



```
if (prvi == NULL)
    prvi = novi;
else /* Očekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
novi->sljed = NULL;
zadnji = novi;
```

## *Funkcija* ubaci\_na\_kraj

```
lista ubaci_na_kraj(lista prvi, lista *p_zadnji,  
                    lista novi)  
{  
    /* Ne provjerava novi != NULL. */  
    /* Vraca zadnji kroz varijabilni  
       argument - pokazivac p_zadnji. */  
  
    lista zadnji = *p_zadnji;
```

## *Funkcija* ubaci\_na\_kraj — *nastavak*

```
        /* Moze: prvi == NULL || zadnji == NULL. */
if (prvi == NULL)
    prvi = novi;
else    /* Ocekujemo zadnji->sljed == NULL. */
    zadnji->sljed = novi;
novi->sljed = NULL;

/* Ne treba:
    zadnji = novi; *p_zadnji = zadnji;
*/
*p_zadnji = novi;    /* Vrati novi zadnji! */

return prvi;
}
```



## Primjeri programa i funkcija kreiraj\_straga

Programi `1_1.c`, `1_2.c` i `1_3.c` (dostupno na webu)

☘ koriste sve dosadašnje funkcije za rad s listama.

Posebno, `1_3.c` sadrži i primjer loše realizacije ubacivanja na kraj liste (stalno traži zadnji, umjesto da se pamti zadnji).

Zadatak. Napišite funkciju `kreiraj_straga`, po ugledu na funkciju `kreiraj_sprijeda`, sa zaglavljem

---

```
lista kreiraj_straga(lista prvi, lista *p_zadnji,  
                    int broj)
```

---

koja kreira novi element za zadani broj, “ubacuje” ga na kraj liste i korektno vraća pokazivače na prvi i zadnji element.

Pogledajte program `1_4.c`.