Datoteke

Osnovno o datotekama

Koncept **datoteke** (engl. *file*) **razdvaja upotrebu sadržaja** (podataka) **od njihove organizacije**

Služe za dugotrajno čuvanje podataka, čine ih strukture podataka smeštene u masovnoj memoriji

Prema načinu pristupa podacima, datoteke se dele na:

- **sekvencijalne**
- **direktne** (sa slučajnim pristupom)

Prema organizaciji podataka, datoteke se dele na:

- **sekvencijalne**
- **direktne**
- **indeks-sekvencijalne** (indeksi (heševi) + podaci (zapisi u tabelama))

Prema prirodi podataka koje sadrže, datoteke se dele na:

- **tekstualne**
- **binarne**

Sistemi sa rad sa datotekama

Sistem za rad sa datotekama (engl. *filesystem*) kontroliše kako se čuvaju podaci i kako im se pristupa

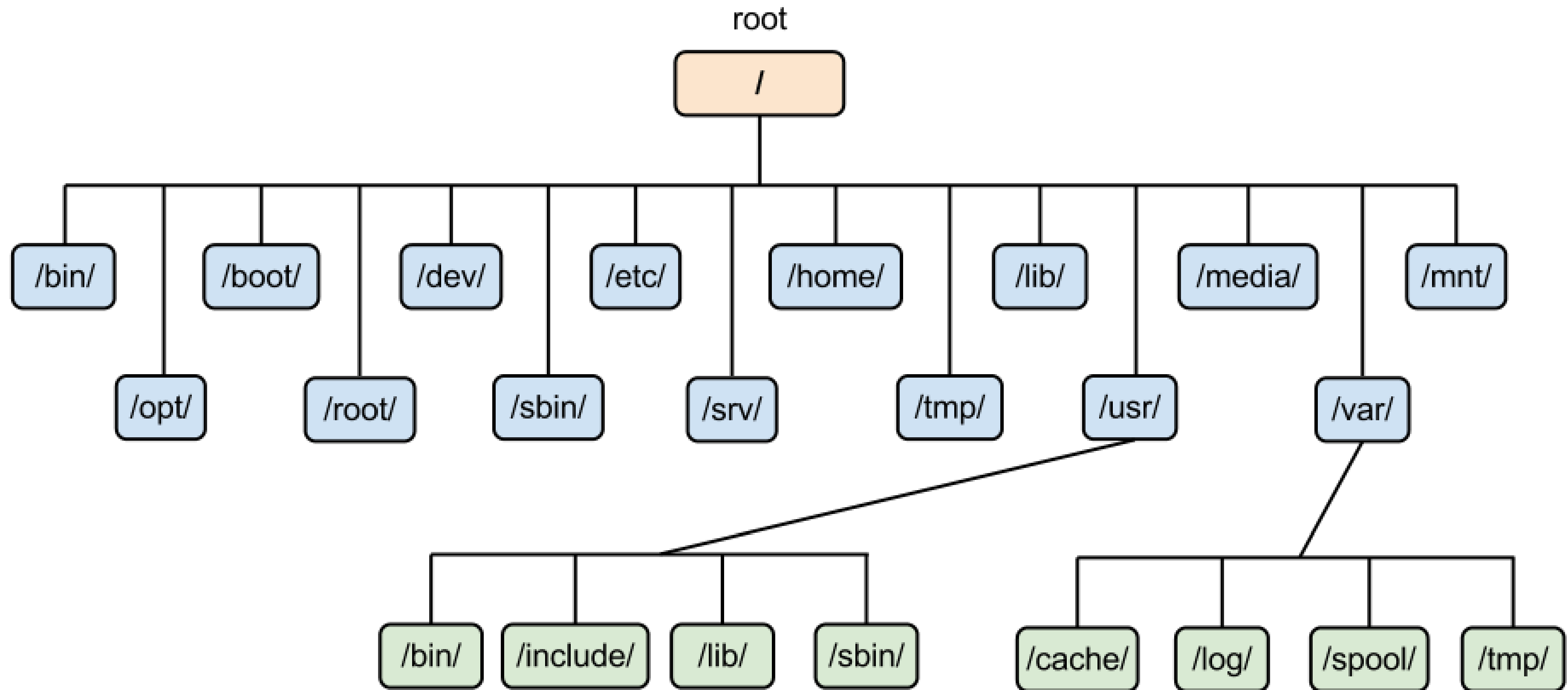
Globalno definiše način na koji računar organizuje, imenuje, čuva i manipuliše datotekama

Bez sistema za rad sa datotekama, podaci zapisani na memorijskom medijumu predstavljali bi jedinstvenu celinu, tj. ne bi mogli da kažemo gde se neka informacija završava, a sledeća počinje

Primeri sistema za rad sa datotekama:

- FAT (FAT16, FAT32), NTFS, ext (2, 3, 4), UDF, …

Primer – Linux filesystem (ext)



Izvor:https://freedompenguin.com/articles/how-to/learning-the-linux-file-system/

Tok rada sa datotekama u jeziku C

1. **Deklarisanje datotečne promenljive**

   FILE *datProm;

2. **Otvaranje datoteke**

   FILE *fopen(const char *nazivDatoteke, const char *rezim);

Režimi rada	
"r" / **"rt"** **"rb"**	Otvori za čitanje **tekstualnu** ili **binarnu** datoteku, pozicioniraj se na početak datoteke. Ako datoteka ne postoji, vraća NULL.
"w"/ **"wt"** **"wb"**	Otvori za pisanje **tekstualnu** ili **binarnu** datoteku, pozicioniraj se na početak datoteke. Gubi se stari zapis. Ako datoteka ne postoji, kreira se nova.
"a"/ **"at"** **"ab"**	Otvori za dodavanje **tekstualnu** ili **binarnu** datoteku, pozicioniraj se na kraj datoteke i omogući dodavanje novih zapisa. Čuva se stari zapis. Ako datoteka ne postoji, kreira se nova.
"r+" / **"rt+"** ili **"rb+"** **"w+"**/ **"wt+"** ili **"wb+"** **"a+"**/ **"at+"** ili **"ab+"**	Čitanje i pisanje od početka. Gubi se stari zapis. Čitanje i pisanje od početka. Gubi se stari zapis. Čitanje i pisanje od kraja. Čuva stari zapis.

Tok rada sa datotekama u jeziku C

3. **Čitanje ili pisanje podataka u datoteku**
   - koriste se različite funkcije u zavisnosti od vrste datoteke
   - čitanje do kraja pomeranjem internog pokazivača datoteke:

     ```
     int *feof(FILE *datProm);
     ```

4. **Zatvaranje datoteke**

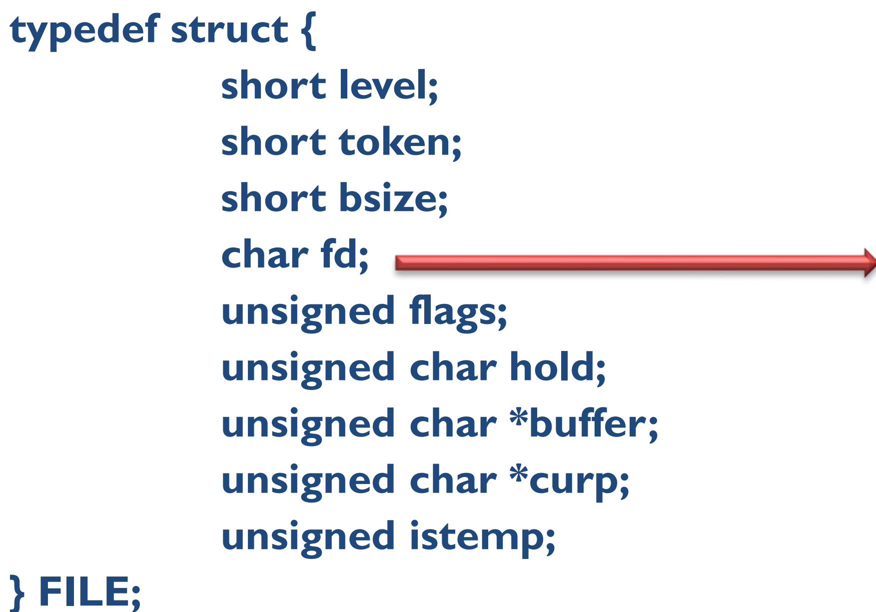
   ```
   int fclose(FILE *datProm)
   ```

Struktura FILE

Definisana u zaglavlju *stdio.h*

Direktorijum (folder) je samo specijalna vrsta fajla – fajl fajlova

Pruža neophodne informacije o datoteci ili toku koji obavlja ulazne i/ili izlazne operacije, primer iz K&R:



```
typedef struct {
        short level;
        short token;
        short bsize;
        char fd;
        unsigned flags;
        unsigned char hold;
        unsigned char *buffer;
        unsigned char *curp;
        unsigned istemp;
} FILE;
```

Deskriptor datoteke sadrži atribute datoteke: naziv, veličina, redni brojevi blokova, vreme nastanka, izmene, pristupa, ...

Tekstualne datoteke

Sadržaj se interpretira kao ASCII, čak i kontrolni karakteri

Funkcije za prenos znakova sa konverzijom:

int fscanf(FILE *datProm, const char *format [,adresa, ...])

int fprintf(FILE *datProm, const char *format [,prom, ...])

- isto kao i odgovarajuće funkcije za rad sa **stdin** i **stdout**
- navodi se datotečna promenljiva kako bi se znalo odakle se čita, tj. gde se piše

Tekstualne datoteke

Funkcije za prenos karaktera (bez konverzije):

Funkcije	
int **fgetc**(FILE *datProm)	Funkcija koja čita iz tekstualne datoteke jedan karakter koji vraća svojim identifikatorom
int **getc**(FILE *fajlProm)	Makro koji čita iz tekstualne datoteke jedan karakter koji vraća svojim identifikatorom
int **fputc**(int znak, FILE *fajlProm)	Funkcija koja upisuje znak u tekstualnu datoteku, dok svojim identifikatorom vraća taj isti znak ili kod greške
int **putc**(int znak, FILE *fajlProm)	Makro koji upisuje znak u tekstualnu datoteku, dok svojim identifikatorom vraća taj isti znak ili kod greške
char ***fgets**(char *str, int n, FILE *fajlProm)	Funkcija koja čita iz tekstualne datoteke niz od n-1 znakova ili dok ne naiđe na znak '\0'.
int **fputs**(const char *str, FILE *fajlProm)	Funkcija koja upisuje u tekstualnu datoteku niz znakova sa '\0' terminatorom

Tekstualne datoteke – primeri

Zadatak 1:
Napisati program koji učitava tekst iz tekstualnog fajla i isti ispisuje na ekran. Pretpostaviti da u jednom redu tekstualnog fajla može biti maksimalno 255 karaktera.

Vežba 1:
Proširiti program tako da omogući korisniku unos naziva tekstualnog fajla.

Vežba 2:
Napisati program koji omogućuje korisniku da unosi tekst sa tastature, koji će se potom čuvati u tekstualnom fajlu sa imenom po izboru korisnika.

Vežba 3:
Napisati program koji kopira sadržaj jednog tekstualnog fajla u drugi sa imenom po izboru korisnika.

Tekstualne datoteke – Zadatak 1

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

int main()
{
    FILE *ulaz;
    char nazivDat[31] = "poruka.txt";
    char red[256];

    if ((ulaz = fopen(nazivDat,"r")) == NULL)// Otvaranje datoteke sa proverom prava na citanje (r)
    {
        printf("\nGreska prilikom otvaranja datoteke \'%s\' za citanje.\n", nazivDat);
        exit(EXIT_FAILURE);                          // Prevremeni izlaz iz programa
    }

    while (fgets(red, 255, ulaz) != NULL)        // Citanje stringova max duzine 255 iz ulazne datoteke
        printf("%s", red);

    printf("\n");

    fclose(ulaz);                                // Zatvaranje datoteke
    return 0;
}
```

Binarne datoteke

Sadržaj se interpretira kao *n*-torka bitova (najčešće celobrojni umnožak bajta)

Može se tumačiti da je organizovana kao **struct**

Sadržaj binarne datoteke čita se pomoću funkcije:

```
int fread(void *lokacija, int velBlok, int brBlok,
          FILE *datProm)
```

Čita od trenutne pozicije internog pokazivača datoteke označenog sa **datProm**

Čita se **brBlok** blokova, gde je svaki blok veličine **velBlok**

Pročitane vrednosti smeštaju se u memoriju počev od adrese **lokacija**

Binarne datoteke

U binarnu datoteku se piše pomoću funkcije:

```
int fwrite(void *lokacija, int velBlok, int brBlok,
           FILE *datProm)
```

Piše se od trenutne pozicije internog pokazivača datoteke označenog sa **datProm**

Upisuje se **brBlok** blokova, gde je svaki blok veličine **velBlok**

Vrednosti koje treba upisati u datoteku, čitaju se iz memorije počev od adrese **lokacija**

Pozicioniranje unutar datoteke

Moguće je manipulisati internim pokazivačem datoteke

Pozicija internog pokazivača saznaje se upotrebom funkcije:

```
long ftell(FILE *datProm)
```

Interni pokazivač se na početak datoteke pomera pomoću funkcije:

```
void rewind(FILE *datProm)
```

Funkcija za pomeranje internog pokazivača datoteke:

```
int fseek(FILE *datProm, long offset, int reper)
```

Moguće vrednosti parametra *reper*	
SEEK_SET	***offset*** se računa u odnosu na početak fajla
SEEK_CUR	***offset*** se računa u odnosu na trenutnu poziciju internog pokazivača
SEEK_END	***offset*** se računa u odnosu na kraj fajla

Binarne datoteke – primeri

Zadatak 2:
Napisati program koji upisuje elemente niza sa 10 prirodnih brojeva u binarnu datoteku.

Zadatak 3:
Napisati program koji čita elemente niza sa 10 prirodnih brojeva iz binarne datoteke (sačuvane u prethodnom primeru).

Vežba 4:
Napisati program koji vodi evidenciju o polaznicima kursa. Maksimalno ima 40 polaznika. Svaki polaznik opisan je JMBG-om (koji ga jedinstveno identifikuje), imenom, prezimenom i nizom u kojem se čuva informacija o kursevima koje polaže. Prilikom izlaska iz programa, podaci se memorišu u binarnu datoteku. Prilikom pokretanja programa, podaci se učitavaju iz binarne datoteke u niz. Omogućiti korisniku da unosi, briše i modifikuje podatke o polaznicima kursa, kao i da prikaže podatke o svim polaznicima.

Binarne datoteke – Zadatak 2

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

#define VELICINA 10

int main()
{
  int i, niz[VELICINA];
  FILE *izlaz;

  for (i = 0; i < VELICINA; i++)                          // Inicijalizacija niza niz[]
     niz[i] = i + 1;

  if ((izlaz = fopen("podaci.dat", "wb")) == NULL)        // Otvaranje datoteke u binarnom modu
  {
     fprintf(stderr, "Greska pri otvaranju datoteke.");
     exit(1);
  }

  if (fwrite(niz, sizeof(int), VELICINA, izlaz) != VELICINA)   // Ispis niza u datoteku
  {
     fprintf(stderr, "Greska pri ispisu u datoteku.");
     exit(1);
  }

  fclose(izlaz);

  return 0;
}
```

Binarne datoteke – Zadatak 3

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

#define VELICINA 10

int main()
{
  int i, niz[VELICINA];

  FILE *ulaz;

  if ((ulaz = fopen("podaci.dat", "rb")) == NULL)         // Otvaranje datoteke za citanje u binarnom modu
  {
    fprintf(stderr, "Greska pri otvaranju datoteke.");
    exit(1);
  }

  if (fread(niz, sizeof(int), VELICINA, ulaz) != VELICINA) // Unos podataka u niz niz[]
  {
    fprintf(stderr, "Greska pri citanju datoteke.");
    exit(1);
  }

  fclose(ulaz);

  puts("Brojevi u datoteci PODACI.DAT su:");

  for (i = 0; i < VELICINA; i++)                           // Ispis podataka na ekran
    printf("\t%d\n", niz[i]);

  return 0;
}
```

Strukture podataka

Tipovi podataka prema C99

Izvor: D. Ivetić, *"Strukturirani pristup programiranju/Inženjering, algoritmi i programski jezici"*

Tipovi podataka

Osnovni (skalarni) tipovi

- celi brojevi (označeni, neoznačeni, znakovni)
- realni brojevi (jednostruka/dvostruka preciznost)
- nabrojivi tip
- pokazivači

Složeni (izvedeni) tipovi

- nizovi (engl. *array*)
- strukture (engl. *record*), polja bitova
- unije (engl. *union*)

Apstraktni tipovi podataka (engl. *abstract data types*)

Apstraktni tip podataka

Apstraktni tip podataka – ATP (engl. *abstract data type* – ADT) je matematički model u kome se tip podataka definiše na osnovu njegovog **ponašanja** (semantike) **iz tačke gledišta korisnika podataka**, u smislu **skupa mogućih vrednosti**, **izvodljivih operacija** nad podacima datog tipa i **ponašanja ovih operacija,** uključujući njihovu složenost izračunavanja (engl. **computational complexity**)

Nasuprot njima, **strukture podataka** (engl. *data structures*) predstavljaju **konkretne reprezentacije podataka iz tačke gledišta njihovog realizatora (implementatora)**, a ne korisnika

Apstraktni tip podataka

ATP se mogu definisati kao **klase objekata** čija je **logika ponašanja** definisana **skupom vrednosti** i **skupom operacija** – analogno algebarskim strukturama u matematici

ATP su teoretski koncept u računarstvu i koriste se u projektovanju i analizi algoritama i strukturama podataka

Ne odgovaraju direktno konkretnim konceptima u programskim jezicima

Pojam ATP su predložili Barbara Liskov (MIT) i Stephen Zilles (IBM, Adobe) 1974.

Vrste ATP

- **lista** (engl. *list*)
- **stek** (engl. *stack*)
- **red** (engl. *queue*)
- **red sa prioritetom** (engl. *priority queue*)
- **dek** (engl. *double-ended queue – deque*)
- **dek sa prioritetom** (engl. *priority deque*)
- **stablo** (engl. *tree*)
- **graf** (engl. *graph*)
- **mapa**, multimapa (engl. *map*, *multimap*)
- skup, multiskup (engl. *set*, *multiset*)

Primer ATP – lista

Lista sadrži **prebrojivo mnogo uređenih vrednosti**, pri čemu se svaka vrednost može pojaviti više puta

Može se implementirati putem **polja** (niza) (engl. array) ili **spregnute (povezane) liste** (engl. linked list)

Lista je primer jednostavnog **kontejnera**

Neke od tipičnih operacija sa listama:

- provera da li je lista prazna
- dodavanje elementa u listu
- brisanje elementa iz liste
- ...

Strukture podataka

Struktura podataka je određeni **način organizacije i čuvanja podataka u računaru** koji omogućava **efikasan pristup podacima i izmene podataka**

Struktura podataka je skup vrednosti podataka, odnosa između njih, i operacija koja se mogu primeniti nad podacima

Strukture podataka mogu da **implementiraju jedan ili više ATP**, kojima se specificiraju moguće operacije nad strukturom i njihova složenost

Struktura podataka je **konkretna implementacija** specifikacije određene putem **ATP**

Strukture podataka

Linearne strukture podataka

- polje (niz)
- spregnuta (povezana) lista
- stek
- red
- dek

Nelinearne strukture podataka

- stablo
- graf

Linearne strukture podataka

Sve **linearne strukture** imaju **jednodimenzionalno uređenje podataka**

Broj elemenata linearne strukture se naziva njenom **dužinom** i uobičajeno se obeležava sa ***n***

Kada je $n = 0$, linearna struktura je prazna

Kada je $n > 0$, linearna struktura ima elemente

Prvi element **nema prethodnika, poslednji element nema sledbenika, svi ostali elementi imaju i prethodnika i sledbenika**

Operacije nad linearnim strukturama

- Pristup elementima strukture (0, ..., n-1)
- Pretraživanje strukture i vraćanje pozicije elementa
- Pristupanje vrednosti pojedinog elementa radi pisanja ili čitanja
- Umetanje novog elementa na proizvoljnu poziciju
- Umetanje novog elementa pre prve pozicije
- Umetanje novog elementa iza poslednje pozicije
- Brisanje elementa
- Brisanje cele strukture
- Pronalaženje prethodnika ili sledbenika posmatranog elementa
- Određivanje dužine strukture
- Spajanje dve ili više struktura u jednu
- Razdvajanje strukture na dve ili više

Fizička realizacija struktura

1. **Sekvencijalna fizička realizacija**
2. **Spregnuta fizička realizacija**

Strukture u **obe fizičke realizacije mogu da podrže prethodno navedene operacije**, ali **sa različitim performansama**

- Primer pristupa elementu – polje $O(1)$, spregnuta lista $O(n)$
- Primer dodavanja elementa – polje $O(n)$, spregnuta lista $O(1)$

Za **specifičnu implementaciju** linearne strukture podataka, treba odrediti namenu i podskup operacija koje treba realizovati

Pogotovo treba obratiti pažnju na **izbor fizičke realizacije kod struktura sa specifičnim pravilima pristupa** (stek, red, dek)

Polje

Polje (niz)

Polje ili niz (engl. *array*, *vector*) je **linearna struktura podataka** koja se sastoji od **konačno mnogo elemenata istog skalarnog ili strukturnog tipa**

Svaki element niza je **jednako dostupan**, i svakom se elementu može **pristupati u proizvoljnom redosledu**

Elementima polja pristupa se pomoću **indeksa**

Polja, u zavisnosti od načina alokacije memorije za njihovo čuvanje, mogu biti **statička** ili **dinamička**

Vrste polja

Jednodimenzionalno polje ili **vektor** (engl. *array*, *vector*)- za identifikaciju pojedine pozicije u vektoru koristi se indeks (vrednost indeksa u programskom jeziku C uzima vrednosti iz skupa 0,..., *n*-1 pri čemu je *n* dužina vektora)

Višedimenzionalno polje – **tenzor** (engl. *multidimensional array* – *tensor*) predstavlja generalizaciju jednodimenzionalnog polja tj. vektora, **dvodimenzionalno** polje se naziva **matrica** (engl. *matrix*)

Operacije nad poljima

Proste operacije:

- memorisanje elementa polja (engl. *store*), a[i]=m;
- referenciranje/izdvajanje elementa polja (engl. *extract*), m=a[i];

Složene (kompozitne) operacije:

- dodavanje (engl. *insert*)
- brisanje (engl. *delete*)
- obilazak (engl. *traverse*)
- sortiranje (engl. *sort*)
  - bubble sort, selection sort, quick sort, merge sort…
- traženje (engl. *search*)
  - linearno
  - binarno

Polje u memoriji

Za smeštanje polja u memoriji se koristi **sekvencijalna reprezentacija**

Logička struktura i **fizička reprezentacija polja se poklapaju**

Direktno pristupanje elementima polja putem indeksa **veoma efikasno -** $O(1)$

Brisanje i dodavanje elemenata polja na proizvoljnoj lokaciji **neefikasno -** $O(n)$

Polje u memoriji

Polje se implementira kao **kontinualni skup susednih memorijskih lokacija**

U **statičkoj memoriji** uvek se **zauzima maksimalna potrebna količina memorije** pa se ona koristi za **smeštanje malih nizova**

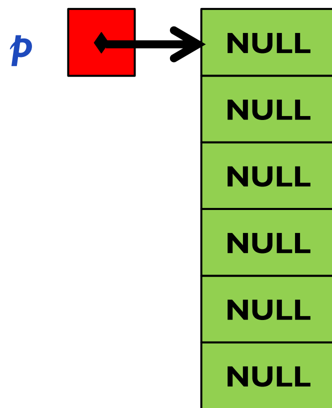
Za **dinamičko kreiranje nizova** tokom **izvršavanja programa** koristimo **gomilu** (engl. *heap*)

Dinamička matrica u jeziku C

Primer dinamičke matrice:

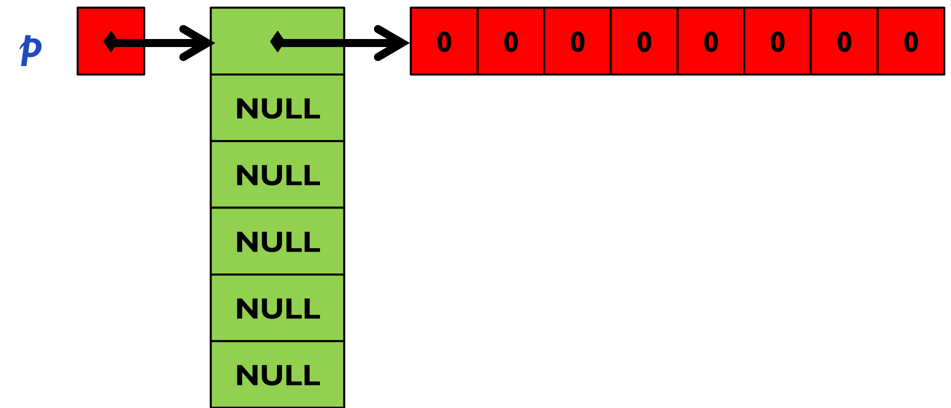
(a) Kreiranje osnove za m=6

```
p = (int**) calloc(m, sizeof (int*));
```



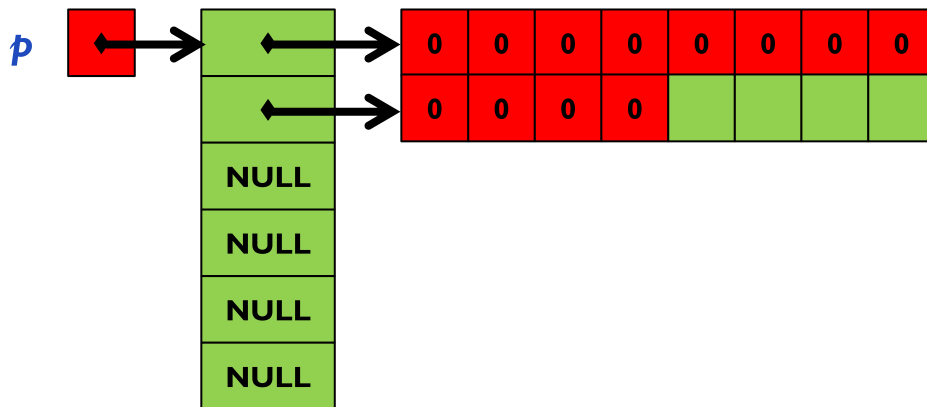
(b) Prva vrsta od k1=8 elemenata

```
p[0] = (int*) calloc(k1, sizeof (int));
```



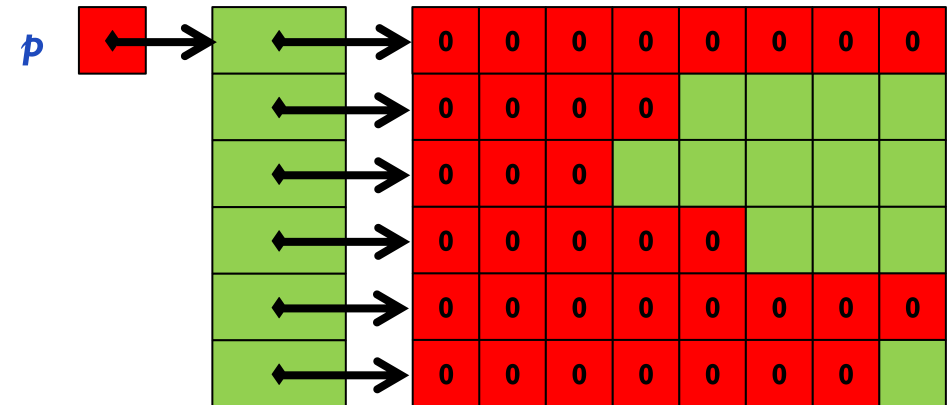
(c) Druga vrsta od k2=4 elementa

```
p[1] = (int*) calloc(k2, sizeof (int));
```



(d) m-ta vrsta od km=7 elemenata

```
p[m–1] = (int*) calloc(km, sizeof (int));
```



Dinamička matrica u jeziku C

Moguće definisanje pokazivača na neki drugi pokazivač, do proizvoljne "dubine" - pokazivač na pokazivač na celobrojni podatak:

```
int **p;
```

Prvi korak u stvaranju dinamičke matrice je alokacija memorije za niz pokazivača:

```
p = (int**)calloc(m, sizeof(int*));
```

- **p** je statički alociran dvostruki pokazivač u koji će biti smeštena adresa niza pokazivača
- **m je broj vrsta (redova) matrice**
- svaki pokazivač u alociranom nizu pokazivača će ukazivati na jedan niz elemenata koji će predstavljati vrste (redove) matrice
- **p** mora biti dvostruki pokazivač - pristup elementu matrice (p[i][j]) - mora se prvo odrediti adresa *i*-te vrste i pristupiti *i*-tom pokazivaču u nizu pokazivača, a zatim odrediti adresa *j*-tog elementa u nizu elemenata i pristupiti tom elementu:

```
p[i][j] isto što i: *(*(p+i)+j)
```

Dinamička matrica u jeziku C

Drugi korak u stvaranju dinamičke matrice je alokacija vrsta matrice:

```
for (i=0; i<m; i++)        // *(p+i) == p[i]
        *(p+i)= (int*)calloc(n, sizeof(int));
```

- u svakoj iteraciji petlje se alocira prostor za jednu vrstu matrice
- alocirani nizovi, koji predstavljaju vrste matrice, ne moraju biti fizički susedni u memoriji, već mogu biti razbacani bilo gde

Uništavanje (dealociranje) matrice se radi u obrnutom redosledu od redosleda kreiranja

- prvo se dealociraju nizovi koji predstavljaju vrste, a zatim niz pokazivača na vrste

```
for (i=0; i<m; i++)
        free (p[i]);        // isto što i free(*(p+i))
free (p);
```

Dinamička matrica – primer

Dinamička kvadratna matrica dimenzija *n*×*n* popunjena pseudoslučajnim brojevima u opsegu [0, 9]:

```
printf("N=");
scanf("%d", &n) ;
// alocira se memorija za n pokazivaca na vrste
a = (int**)calloc(n, sizeof(int*));
for (i = 0; i < n; i++) {
        // alocira se memorija za n elem. vrste koji su tipa int
        *(a+i)= (int*)calloc(n, sizeof(int));   // *(a+i) == a[i]
        for (j = 0; j < n; j++){
                *(*(a+i)+j) = rand()/((double)RAND_MAX+1)*10;
        }       // *(*(a+i)+j) == a[i][j]
}               // rand() i RAND_MAX su iz stdlib.h
```

Dinamička matrica - primer

Primer ispisivanja sadržaja matrice, vrstu po vrstu

Nakon svake iteracije spoljne petlje se u post-uslovu odštampa znak za novi red:

```
for (i=0; i<n; printf("\n"), i++)
    for (j=0; j<n; j++)
        printf("%d ", *(*(a+i)+j));
```

Šta radi sledeći kod?

```
for (i=n-1; i>=0; i--)
    printf("%d ", *(*(a+i)+n-1-i));
```

Dinamička matrica

Uzmimo primer kvadratne matrice za *n* = 4:

Izraz ***(*(a+i)+n-1-i))** za i = 3 postaje:

***(a + 3)** isto kao **a[3]**

***(*(a + 3) + 4 - 1 - 3)** isto kao **a[3][0]**

dok za i = 1postaje:

***(a + 1)** isto kao **a[1]**

***(*(a + 1) + 4 - 1 -1)** isto kao **a[1][2]**

Odgovor:

Prikazuje elemente na sporednoj dijagonali matrice!

Polje – rezime

Prednost polja kao linearne strukture podataka je **podudaranje fizičke i logičke strukture**, kao i **linearna adresna funkcija – pristup slučajnim elementima u konstantnom vremenu putem indeksiranja**

Nedostatak polja su **kompleksne operacije umetanja i brisanja elementa na proizvoljnom mestu**, operacija **promene veličine polja je skupa**

Navedeni nedostaci se mogu prevazići **spregnutom realizacijom** uz odgovarajuću cenu – nema besplatnog ručka!

Spregnuta lista

Spregnute strukture podataka

Kod **sekvencijalne fizičke realizacije linearne strukture** podataka **uzastopni elementi** su **susedni u memoriji**

Kod **spregnute fizičke realizacije linearne strukture podataka uzastopni elementi** mogu biti **bilo gde u memoriji**

Logički linearni poredak se održava tako što **elementi liste sadrže eksplicitnu adresu narednog elementa**

Samoupućujuća struktura – čvor

Često je pogodnije da se elementi urede na proizvoljan način pa da se povežu, nego da se fizički poređaju u sekvencu

Struktura koja **sadrži pokazivač na objekat istog tipa** kao što je ona sama naziva se **samoupućujuća ili samoreferencirajuća** (engl. self-referencing) **struktura podataka**

Samoupućujuća struktura podataka pod nazivom **čvor** predstavlja **osnovu za realizaciju spregnute liste**

Upotrebom samoupućujuće strukture mogu se realizovati različite linearne i nelinearne strukture (red, stek, dek, stablo, graf)

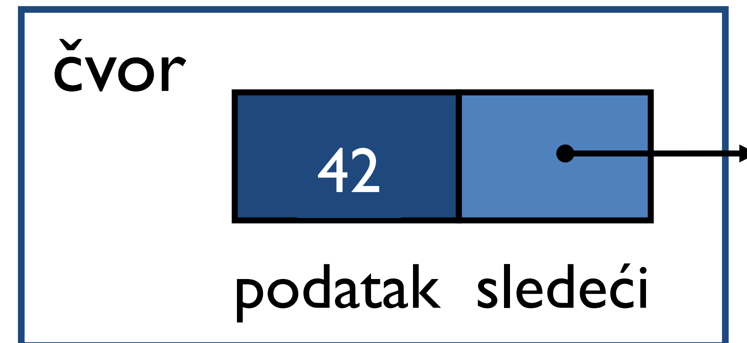
Samoupućujuća struktura – čvor

Svaki **čvor** sadrži dva polja:

- **podatak** – pamti **element liste** (skalarnog ili strukturnog tipa)
- **sledeći** – pamti **adresu sledećeg čvora**

Struktura čvor sa celobrojnim info poljem u jeziku C:

```
typedef struct cvor {
    int podatak;
    struct cvor *sledeci;
} tCvor;
```



Listi se pristupa preko eksternog pokazivača *start* koji ukazuje na prvi element liste koji se naziva i **glava** (engl. *head*) liste

Poslednji element liste kao link sadrži specijalnu vrednost **NULL** koja nije validna adresa

Spregnuta lista

Zajedničko za spregnute liste i **polja** je da čuvaju **linearne kolekcije podataka**

Osobine polja proizilaze iz njegove **strategije alokacije memorije** za elemente u vidu **jednog bloka** memorije

Spregnute liste koriste sasvim drugačiji pristup – **memorija se alocira za svaki element posebno** i **samo kada je** to **neophodno**

Spregnute liste su **pogodne** za upotrebu u situacijama kada **količina podataka** koja se treba čuvati u strukturi **nije predvidljiva**

Spregnuta lista

Spregnute liste su **dinamičke strukture** tako da se njihova **dužina** može **povećavati i smanjivati po potrebi**

Svaki od čvorova **ne mora nužno da se fizički u memoriji nalazi pored svojih suseda**

Spregnute liste mogu se držati u **uređenju** tako što se **čvorovi dodaju ili brišu** na **odgovarajućim mestima u listi**

Vrste spregnutih listi

- jednostruko spregnute liste ili jednosmerne (engl. *singly linked lists*)
- dvostruko spregnute liste ili dvosmerne (engl. *doubly linked lists*)
- necirkularne (engl. *noncircular*)
- cirkularne ili kružne (engl. *circular*)
- sa zaglavljem (engl. *with header*)
- bez zaglavlja (engl. *without header*)
- uređene (sortirane) (engl. *sorted*)
- neuređene (nesortirane) (engl. *unsorted*)

Operacije sa spregnutim listama

- traženje elementa u listi
- dodavanje elementa u listu
- brisanje elementa iz iste
- obilazak liste
- kopiranje liste
- spajanje (konkatenacija) listi
- cepanje liste
- ...

Jednostruko spregnuta lista

Jednostruko spregnuta lista se definiše kao uređeni par:

$$P = (S(P), r(P))$$

sa sledećim osobinama:

- struktura je linearna
- dozvoljen je pristup svakom elementu
- moguće je ukloniti bilo koji element
- dozvoljeno je dodati element na bilo kojoj poziciji

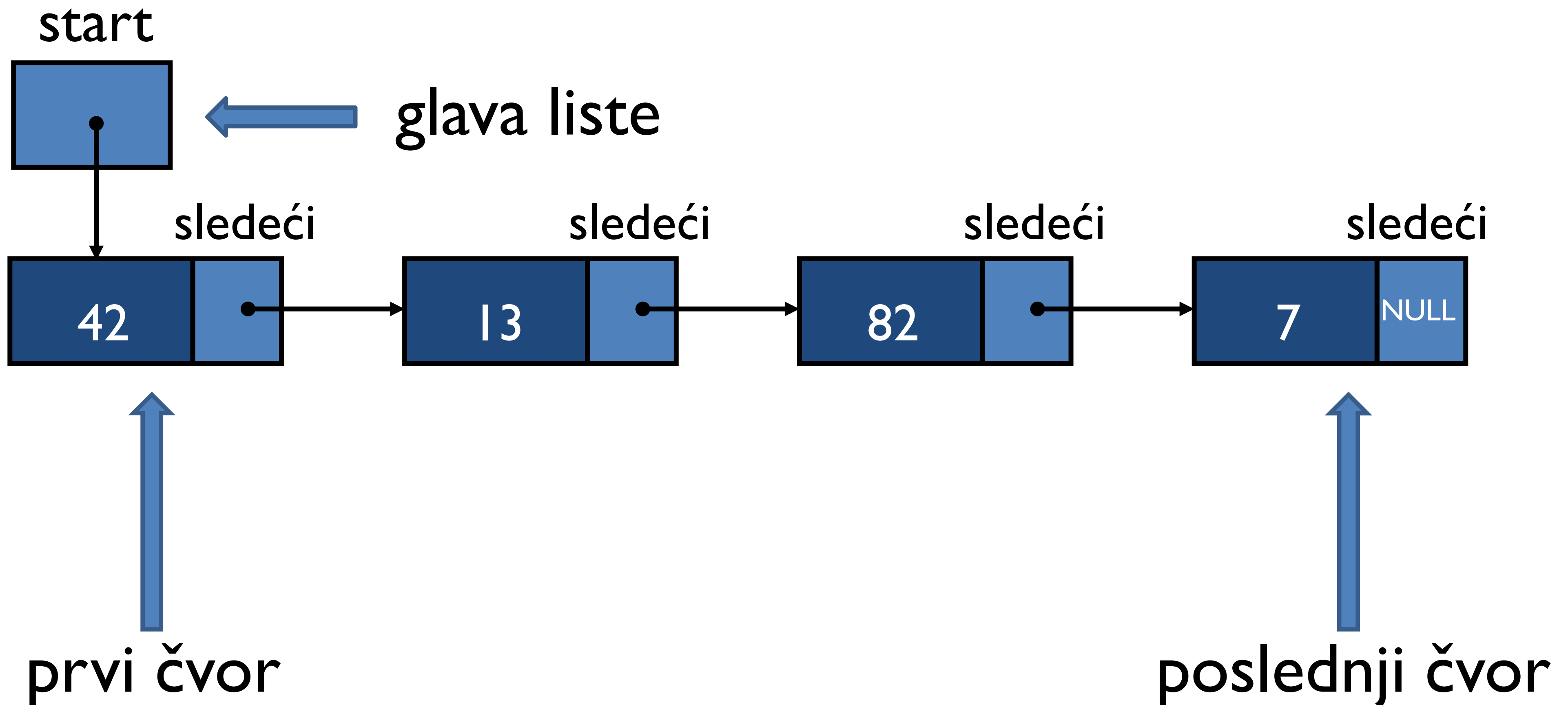
Jednostruko spregnuta lista

Jednostruko spregnuta lista je **skup čvorova povezanih pokazivačima u jednom smeru**

Svaki čvor je **strukturna promenljiva** koja sadrži član koji pokazuje unapred

Spregnuta lista može da sadrži **promenljivi broj čvorova**, što je jedna od **osnovnih prednosti** spregnutih struktura

Primer jednostruko spregnute liste



Tipične operacije

Tipične operacije sa jednostruko spregnutom listom su:

1) **inicijalizacija liste**

2) **dodavanje novog elementa**

3) **obilazak liste** (prikaz svih elemenata iz liste)

4) **brisanje elementa iz liste**

5) **brisanje liste**

Inicijalizacija liste

Inicijalizacija liste predstavlja postavljanje glave liste na NULL

Kada glava liste ima vrednost NULL znači da je lista prazna

***glava = NULL;**

start

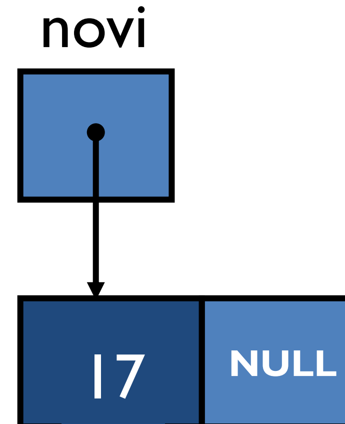
glava liste

Dodavanje novog elementa

Primer dodavanja novog elementa na kraj liste:

1. Formira se struktura tipa **tCvor** i pokazivač **novi**:

```
novi = (tCvor*)malloc(sizeof(tCvor));
```

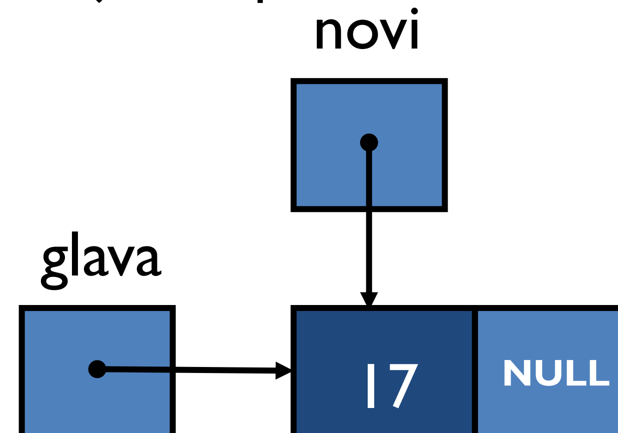


2. Popunjavaju se polja novog čvora:

```
novi->podatak = 17;
novi->sledeci = NULL;
```

3. Ako je lista prazna, tada se novi element ubacuje kao prvi element liste:

```
if(glava == NULL)
{
        glava = novi;
        return;
}
```



Dodavanje novog elementa

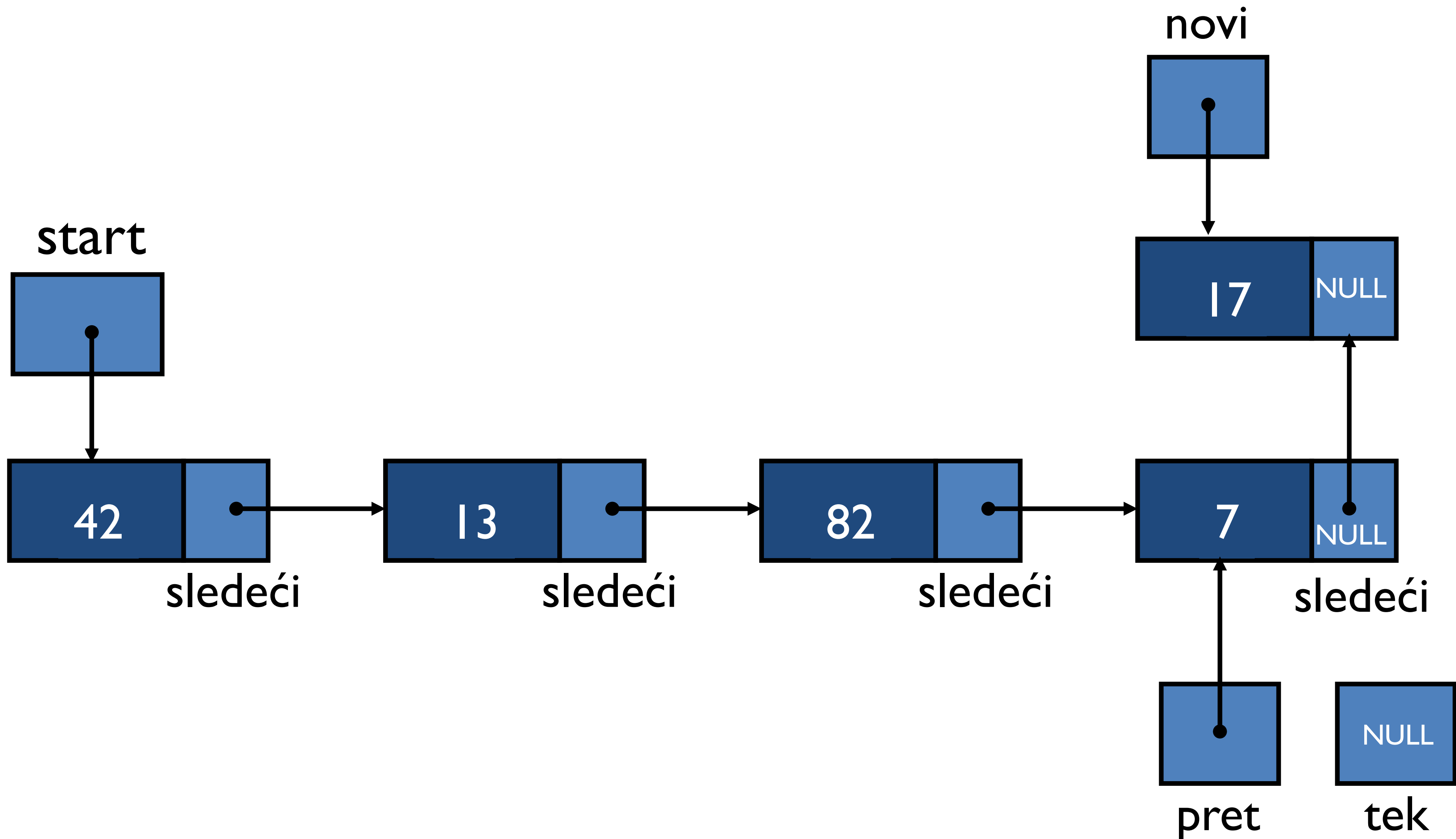
4. Ako lista već postoji, preskočiće se if iz tačke 3 i dodavanje čvora će se obaviti na sledeći način:

```
tek = glava;
pret = glava;
while(tek != NULL)
{
        pret = tek;
        tek = tek->sledeci;
}
pret->sledeci = novi;
```

Nakon prolaska kroz while ciklus **tek** **ne pokazuje ni na šta, a** **pret** **koji ga je "pratio" pokazivaće na poslednji čvor u listi**

5. Nakon **pret->sledeci = novi;** novi čvor će biti povezan kao poslednji element u spregnutoj listi

Dodavanje novog elementa



Obilazak spregnute liste

Obilazak liste predstavlja prikazivanje svih elemenata iz liste

Ako je glava==NULL znači da je lista prazna i nemamo šta prikazati

Ako lista nije prazna, novim pokazivačem polazi se od prvog sloga liste (tek=glava) i dok se ne dođe do kraja liste (tek==NULL) prikazuju se slogovi liste.

```
tek = glava;
while (tek != NULL) {
        printf(" %d ", tek->podatak);
        tek = tek->sledeci;   //prelazak na sledeci
}
```

Brisanje elementa iz spregnute liste

Brisanje elementa iz liste realizuje se tako što se **element prvo mora pronaći** (vrši se traženje elementa u listi)

Traženje elementa liste može se realizovati sledećim kodom:

```
tek = glava;
pret = glava;
while(tek != NULL && (tek->podatak != a))
{
        pret = tek;
        tek = tek->sledeci;
}
```

Razlikujemo dva slučaja brisanja:

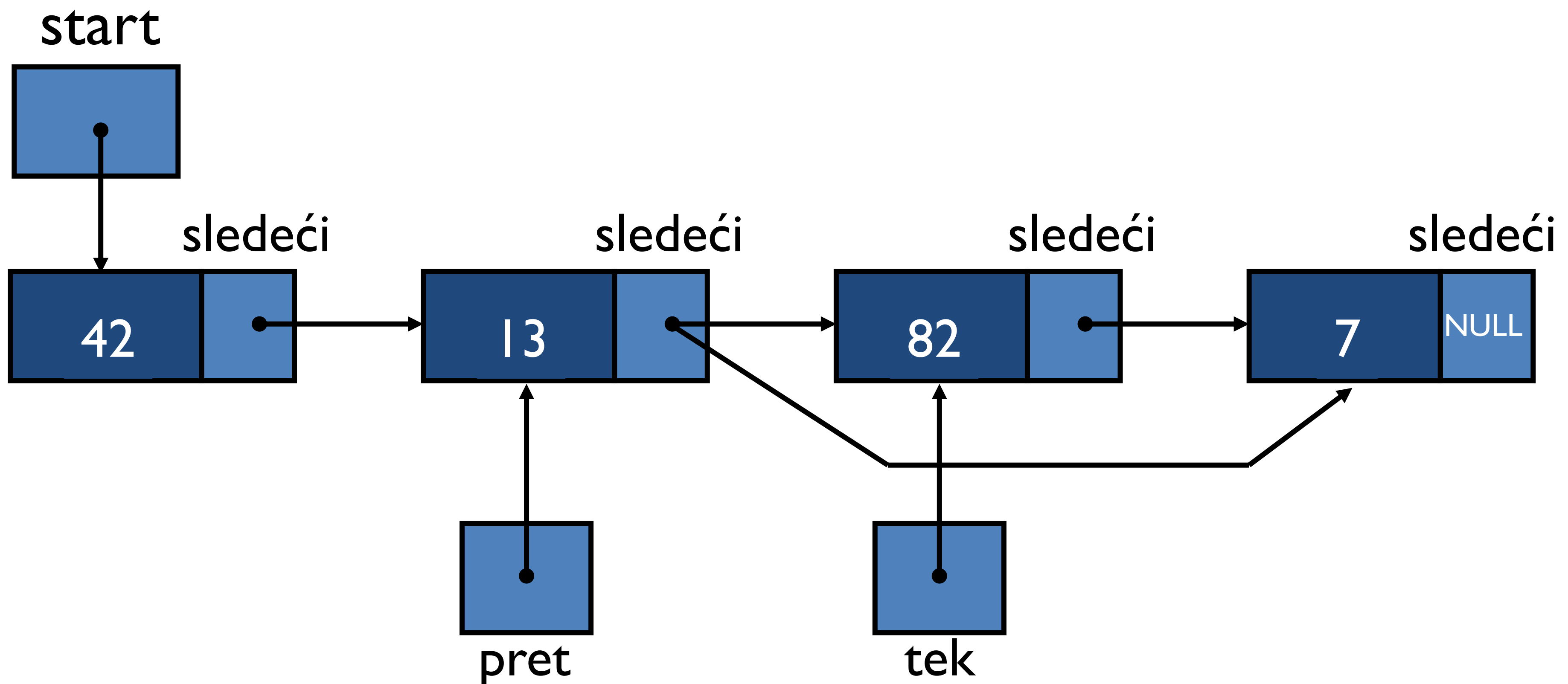
a) čvor koji se briše nije prvi element liste

b) čvor koji se briše jeste prvi element liste

Brisanje elementa iz spregnute liste

a) čvor koji se briše nije prvi element liste

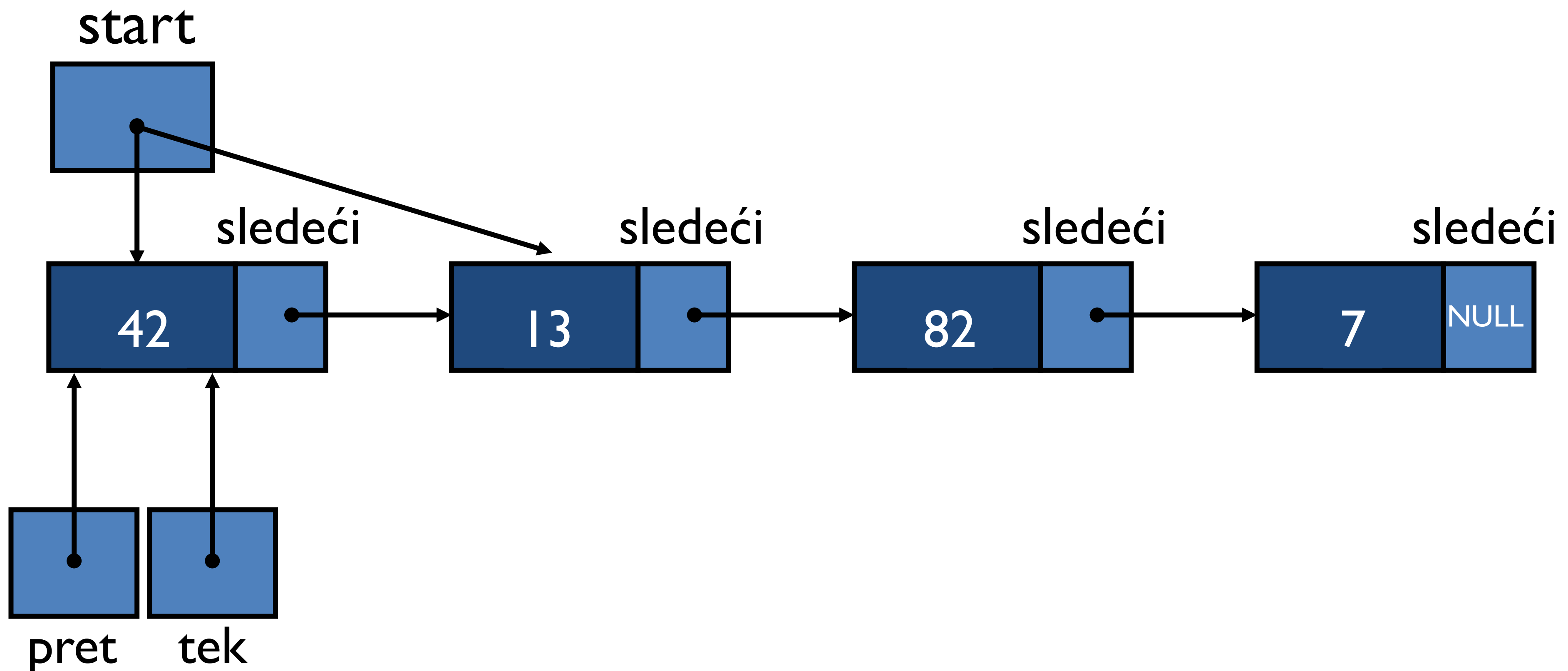
```
pret->sledeci=tek->sledeci;
tek->sledeci=NULL;
```



Brisanje elementa iz spregnute liste

b) čvor koji se briše jeste prvi element liste

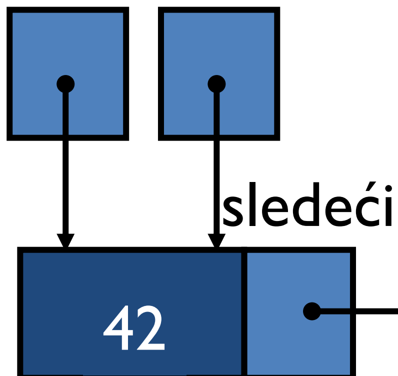
```
glava = tek->sledeci;
tek->sledeci=NULL;
```



Brisanje liste

```
while(glava!=NULL)
{
        tek = glava;
        glava = tek->sledeci;
        free(tek);
}
```

1. start tek



2. start tek