TIPOVI PODATAKA

Dragan dr Dinu

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 APSTRAKTNIH TIPOVA PODATAKA

- 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

Dragan dr Dinu

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 LINEARNIH 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)

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

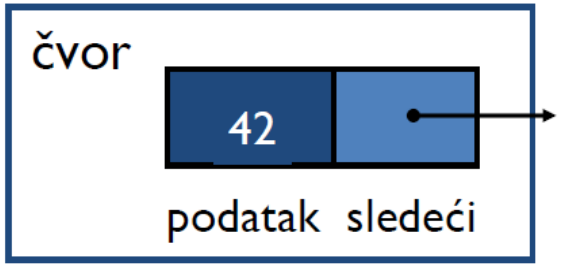
SAMOUPUĆUJUĆA STRUKTURA – ČVOR …

- 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
- 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

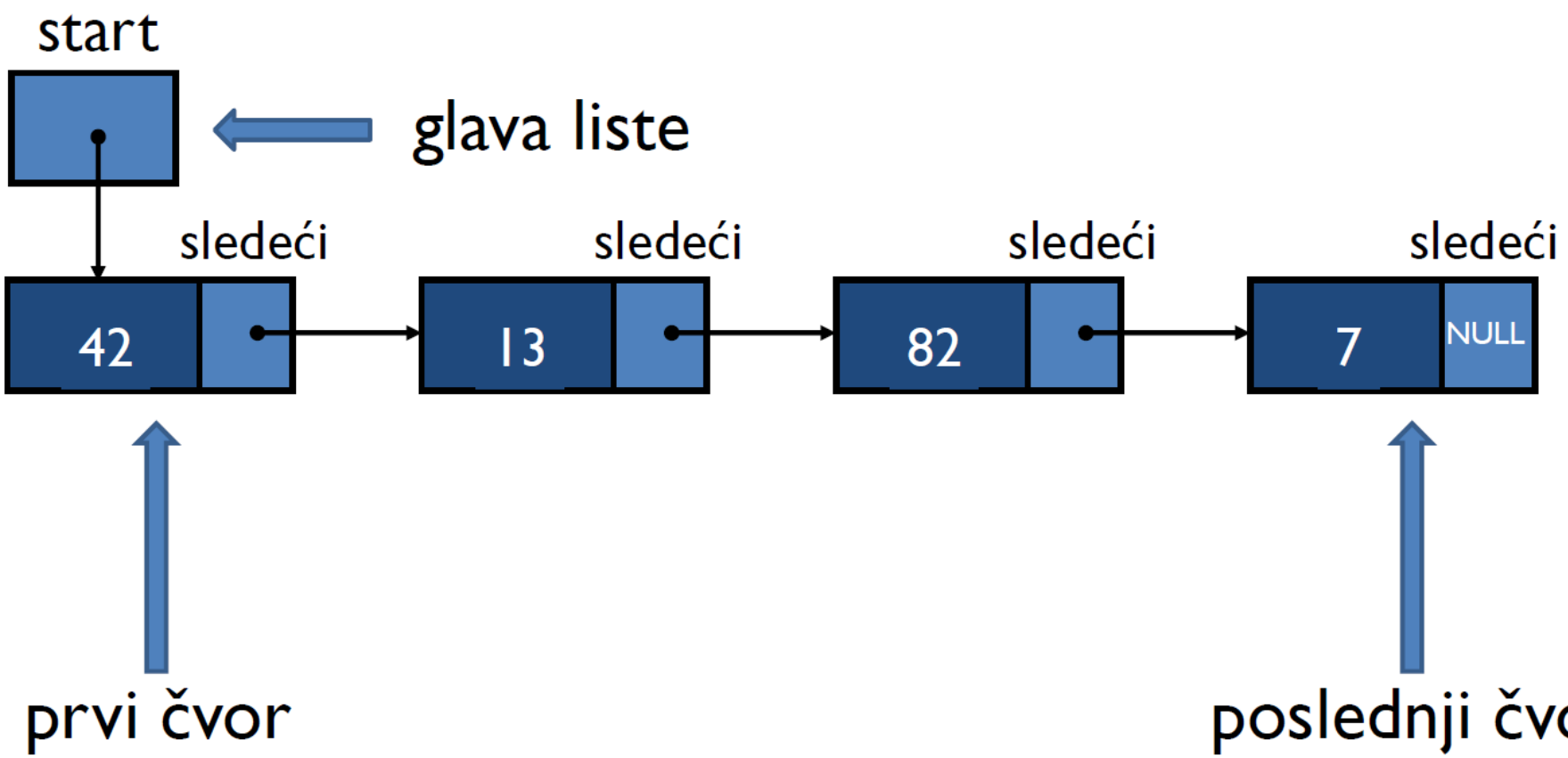
JEDNOSTRUKO SPREGNUTA LISTA

- 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 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

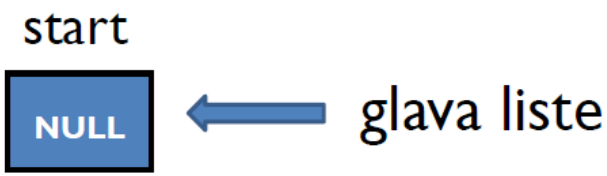
- Tipične operacije sa jednostruko spregnutom listom su:

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;**

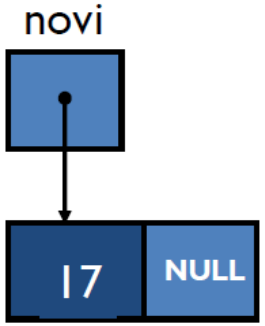


DODAVANJE NOVOG ELEMENTA …

Primer dodavanja novog elementa na kraj liste:

1. Formira se struktura tipa tcvor i pokazivač novi:

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

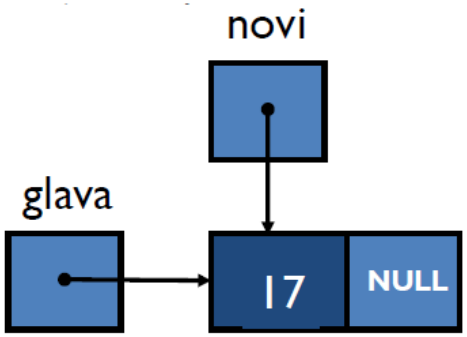


2. Popunjavaju se polja novog čvora:

```
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 …

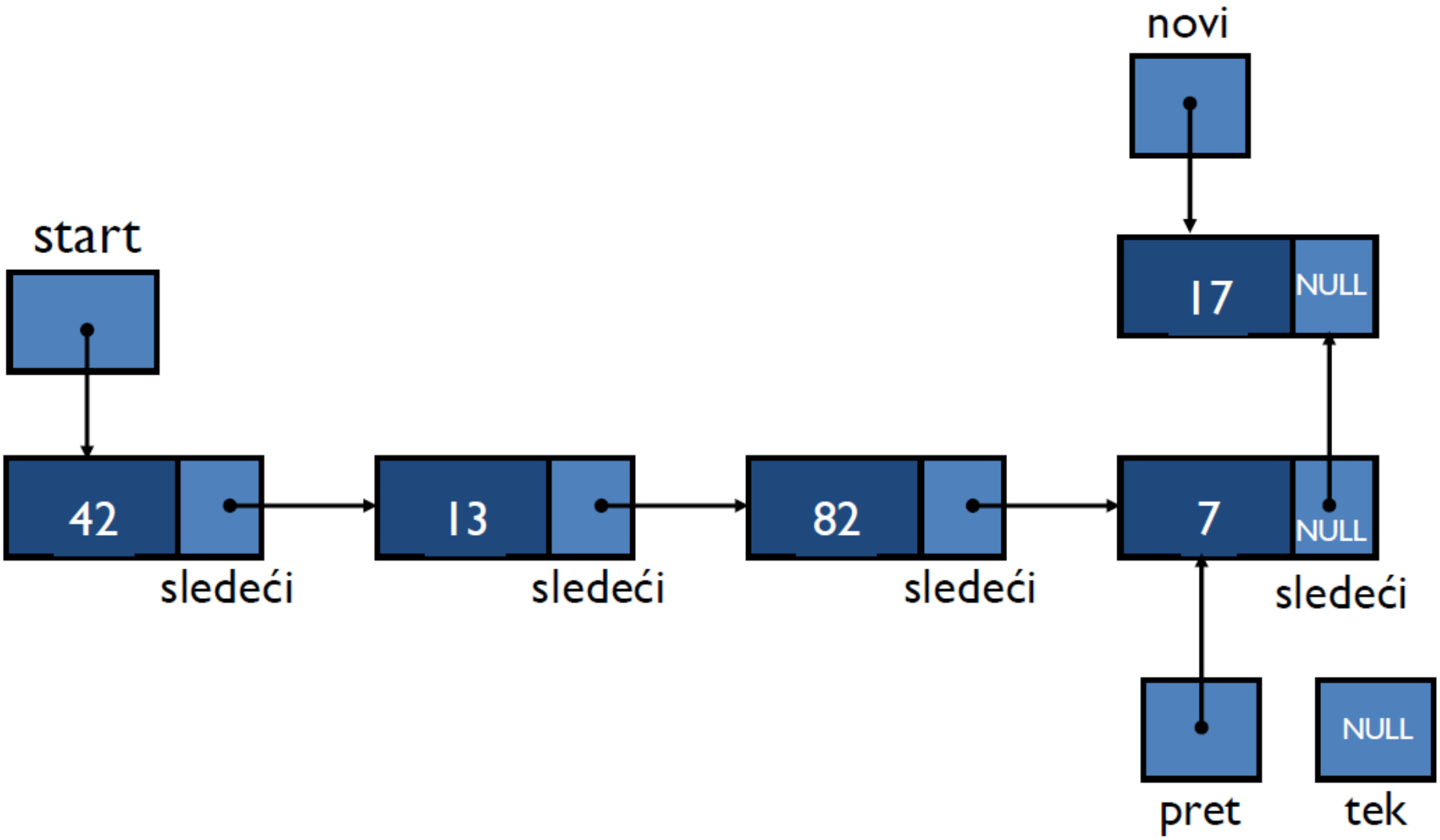
4. Ako lista već postoji, preskočiće se **if** iz tačke 3 i dodavanje čvora će se obaviti na sledeći način:

```
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

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.Kada glava liste ima vrednost **NULL** znači da je lista prazna

```
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:

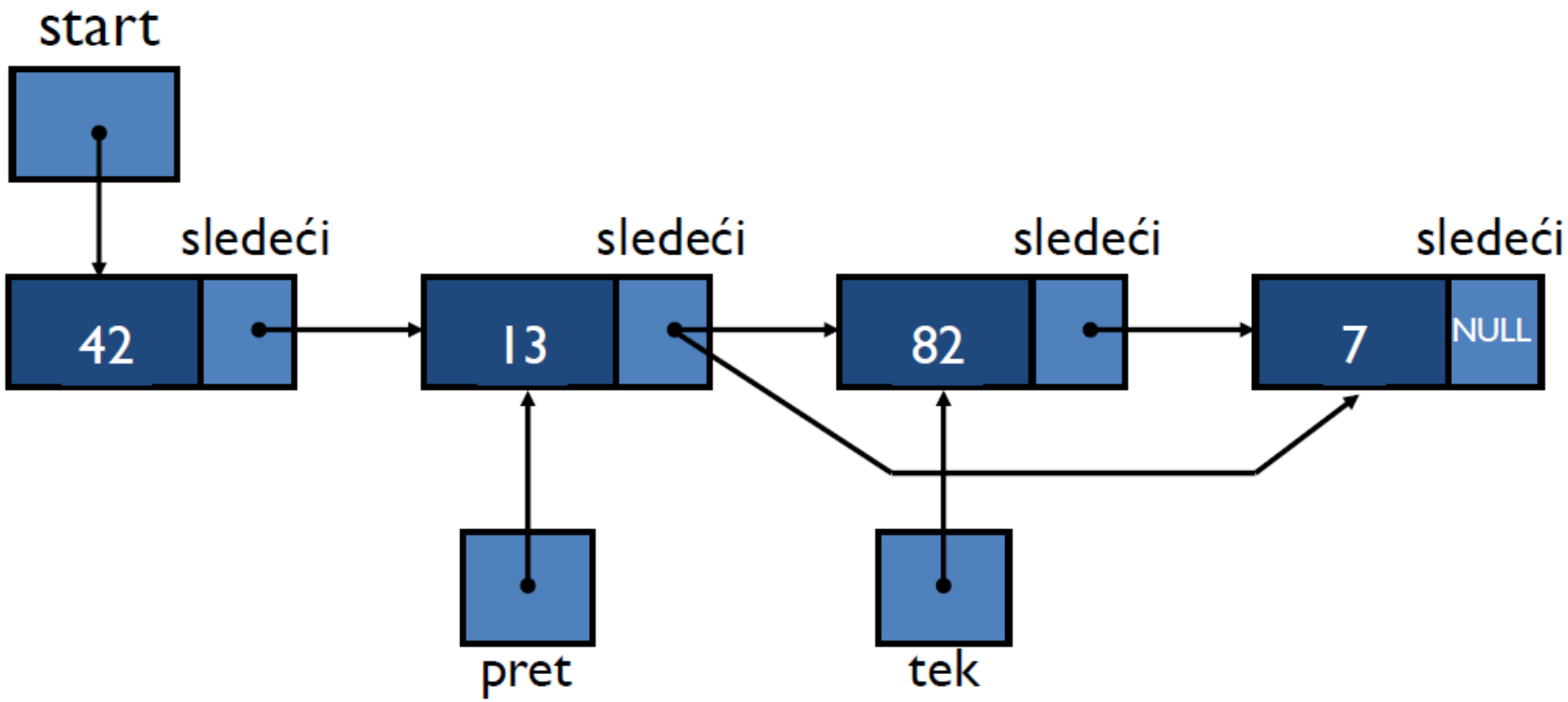
```
tek = glava;
pret = glava;
while(tek != NULL && (tek->podatak != a))
{
        pret = tek;
        tek = tek->sledeci;
}
```

- 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:

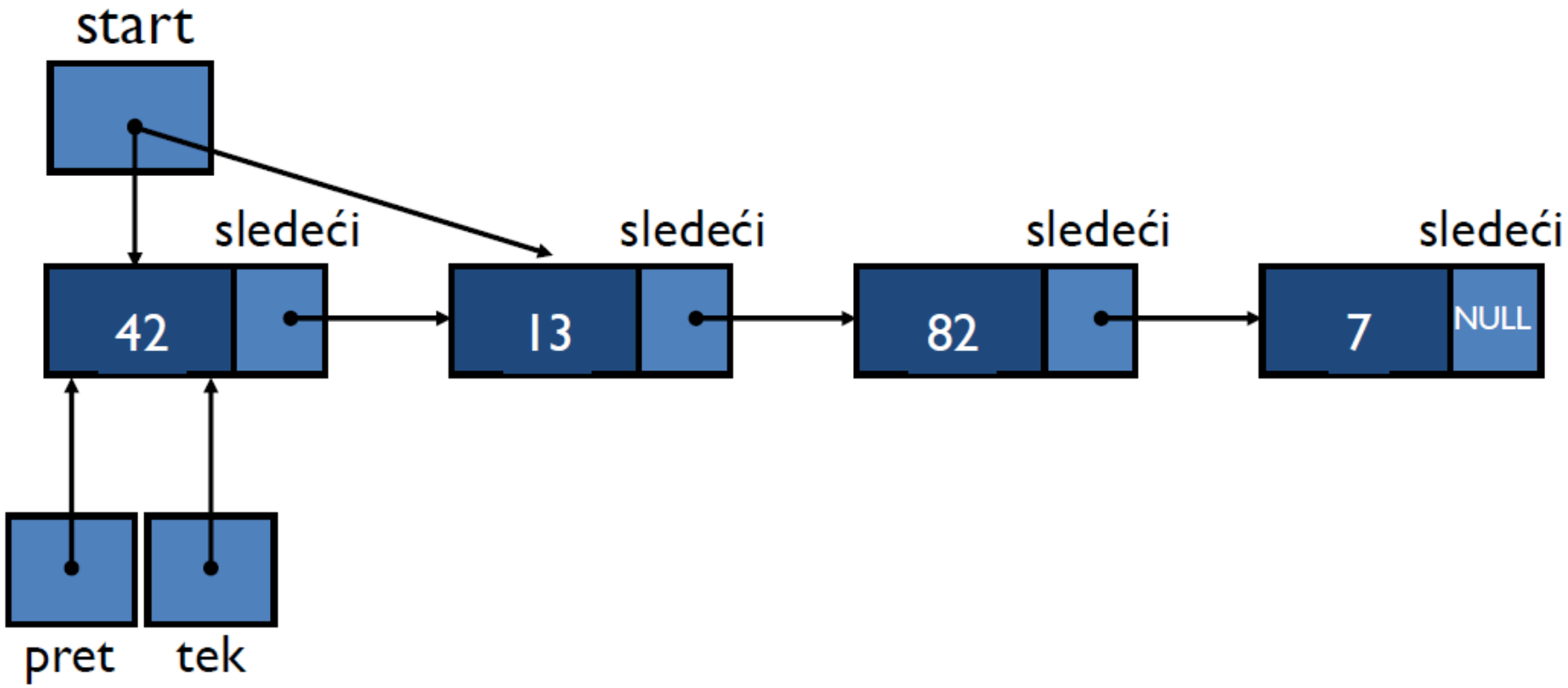
```
pret->sledeci=tek->sledeci;
tek->sledeci=NULL;
```



… BRISANJE ELEMENTA IZ SPREGNUTE LISTE

b) čvor koji se briše jeste prvi element liste:

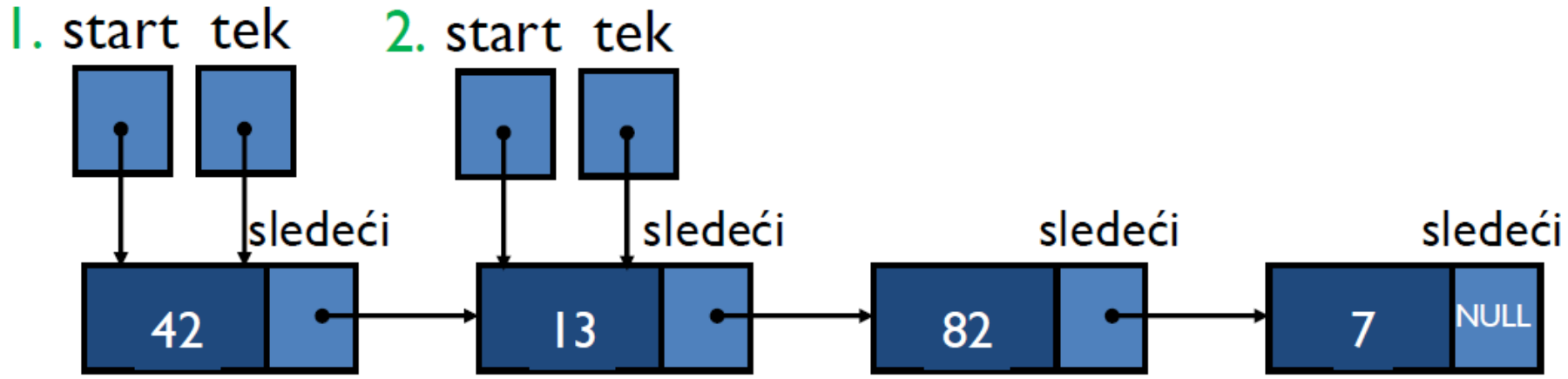
```
glava = tek->sledeci;
tek->sledeci=NULL;
```



BRISANJE LISTE

- Mora se obrisati svaki pojedinačni član liste:

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



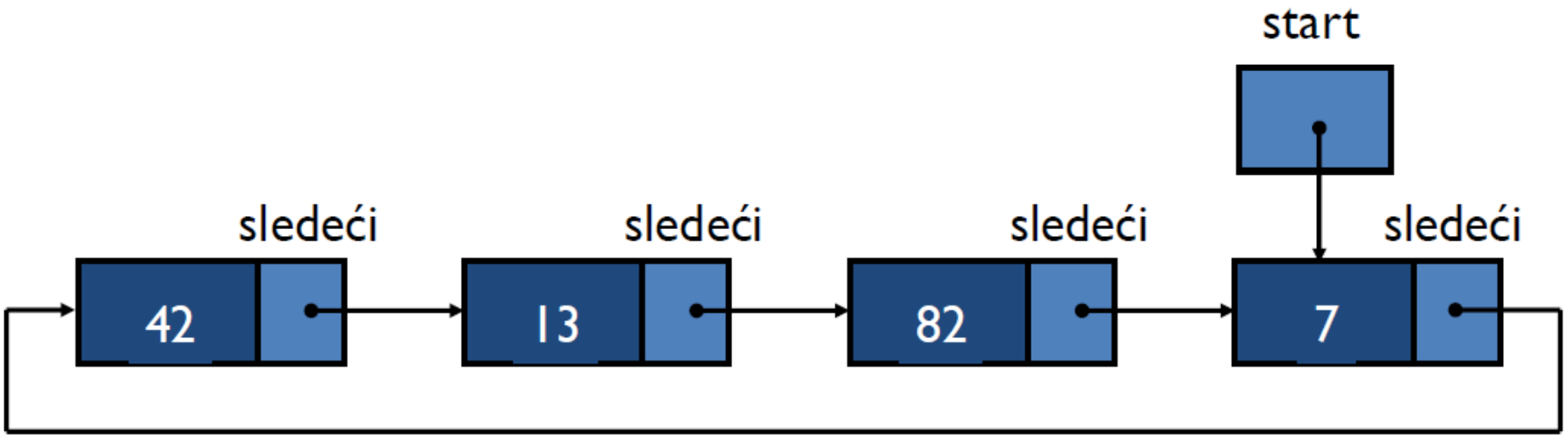
CIRKULARNA SPREGNUTA LISTA

CIRKULARNA SPREGNUTA LISTA

- Vrednost pokazivača na sledeći čvor u čvoru na kraju jednostruko spregnute liste je NULL, pošto on ne pokazuje ni na jedan čvor
- U cirkularnoj ili kružnoj spregnutoj listi svaki čvor pokazuje na sledeći u listi. Poslednji čvor pokazuje natrag na početak liste formirajući na taj način krug
- U slučaju kružne liste jednostavnije se izvode operacije kao što su pretraživanje od nekog unutrašnjeg čvora ka bilo kom delu liste
- Stek (magacin) i red često se implementiraju kao cirkularne spregnute liste

GLAVA CIRKULARNE SPREGNUTE LISTE

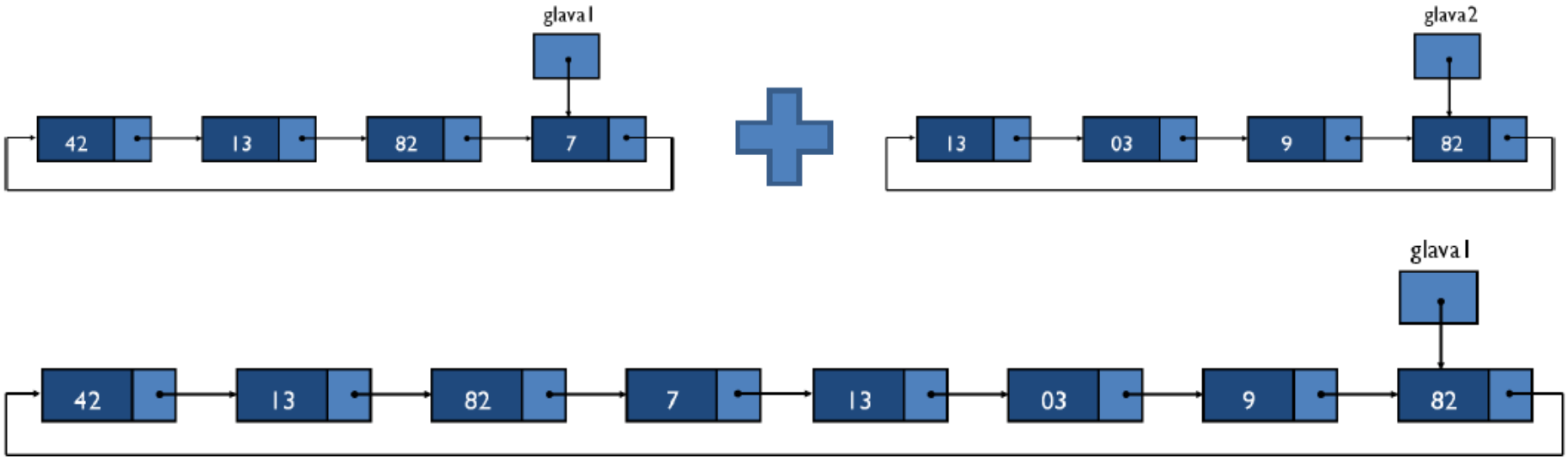
- Zbog osobine simetričnosti kružne liste termini “prvi” i “poslednji” čvor su upitni
- Korisnije je da glava liste pokazuje na poslednji čvor liste, jer to čini efikasnijim operacije umetanja čvora na početak i kraj liste, kao i brisanja čvora sa početka i kraja



SPAJANJE DVE CIRK. SPREGNUTE LISTE

- Operacija spajanja dve liste spoji (**glava1**, **glava2**) ne zahteva prolazak do kraja prve liste pošto pokazivač **glava1** ukazuje na poslednji čvor

- Na kraju operacije pokazivač **glava1** ukazuje na poslednji čvor objedinjene liste

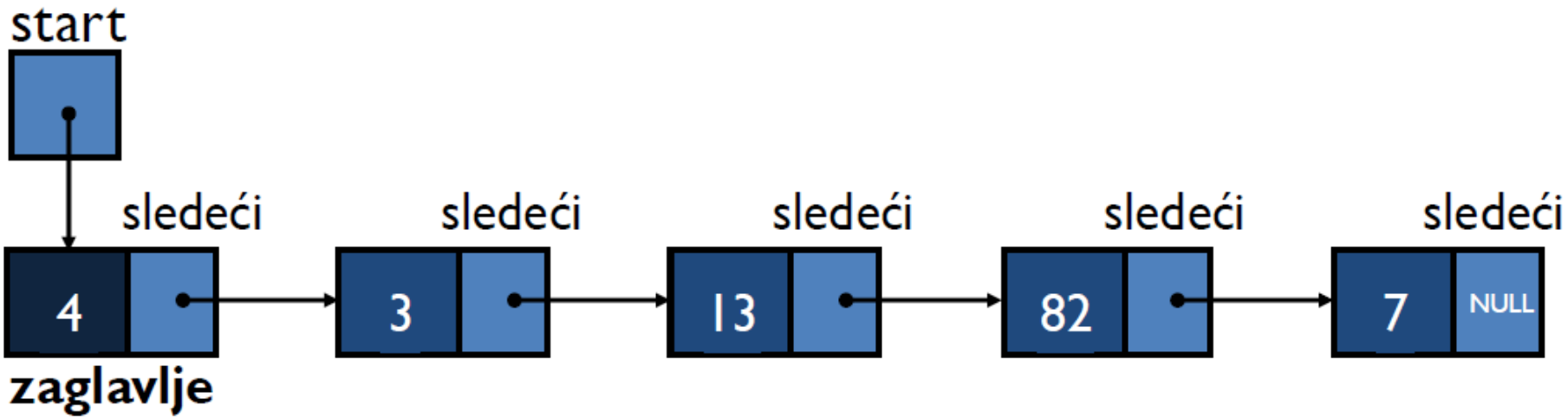


DVOSTRUKO SPREGNUTA LISTA

SPREGNUTA LISTA SA ZAGLAVLJEM

- Zaglavlje (engl. header) je poseban čvor koji se nalazi na početku liste i nije element liste

- Polje podatak u okviru čvora zaglavlja može biti prazno, ali češće sadrži globalne informacije o listi (broj čvorova, datum kreiranja liste, ...) i/ili pokazivač na kraj liste i/ili pokazivač na tekući čvor prilikom obilaska liste



DVOSTRUKO SPREGNUTA LISTA …

- Dvostruko spregnuta lista predstavlja uopštenje jednostruko spregnutih lista
- Za svaku vezu (uređeni par) oblika (a, b) uvodi se veza u obrnutom smeru (b, a)
- Drugi naziv za dvostruko spregnute liste je simetrične liste
- Dvostruko spregnute liste mogu biti u cirkularne ili necirkularne
- Dvostruko spregnute liste mogu biti sa zaglavljem ili bez zaglavlja

… DVOSTRUKO SPREGNUTA LISTA …

- Pojedinačne operacije sa dvostruko spregnutom listom su sporije od jednostruke zbog ažuriranja dva pokazivača
- Razlika u osnovnim operacijama u odnosu na jednostruko spregnutu listu je mala
- Dvostruko spregnuta lista organizuje podatke po jednoj relaciji gde pokazivački deo čvora liste ima dve informacije: koji je sledeći i koji je prethodni čvor po datoj relaciji
- Cilj je da se obezbedi brži pristup čvorovima liste a time i brže operacije dodavanja i uklanjanja

… DVOSTRUKO SPREGNUTA LISTA …

- **Nedostaci:**
  - zahteva više memorijskog prostora
  - manipulacija listom je sporija (jer se više linkova mora izmeniti)
  - više mogućnosti za pravljenje grešaka (zato što se mora manipulisati sa više linkova)
- **Prednosti:**
  - može se obilaziti u oba smera
  - neke operacije, kao što su brisanje ili dodavanje ispred čvora, postaju jednostavnije

… DVOSTRUKO SPREGNUTA LISTA

- Dvostruko spregnuta lista se definiše kao uređeni par:

$$DP=(S(DP), r(DP))$$

- pri čemu se relacija r može razbiti na dve komponente za koje treba da je zadovoljeno:

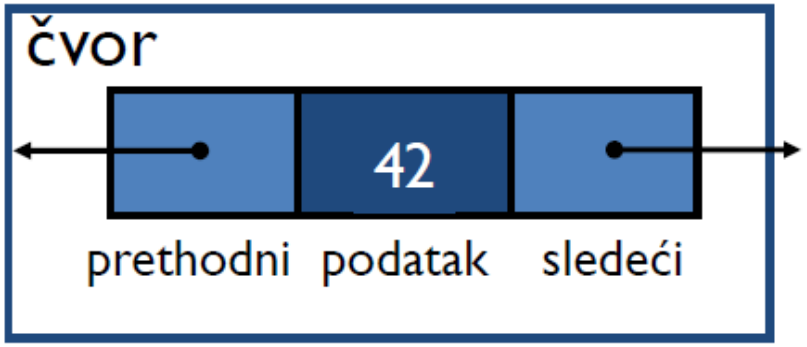
$$r = r_1 \cup r_2 \qquad r_1 \cap r_2 = \emptyset$$

- i da structure $(S(DP), r_1(DP))$ i $(S(DP), r_2(DP))$ budu jednostruko spregnute liste

ČVOR U DVOSTRUKO SPREGNUTOJ LISTI

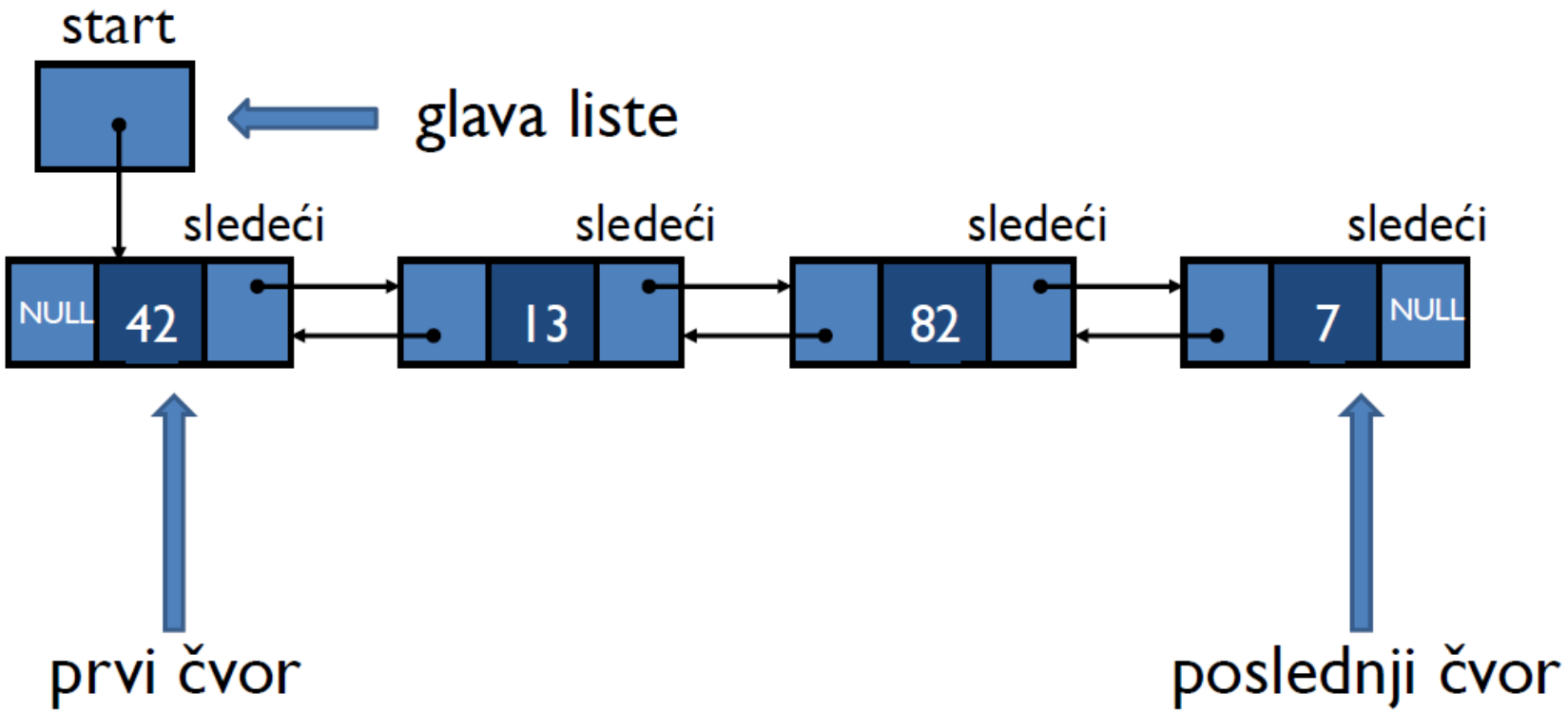
- Svaki čvor sadrži tri polja:
  - podatak – pamti element liste (skalarnog ili strukturnog tipa)
  - sledeći – pamti adresu sledećeg čvora
  - prethodni – pamti adresu prethodnog čvora
- Struktura čvor sa celobrojnim info poljem u jeziku C:

```
typedef struct dCvor {
    int podatak;
    struct dCvor *sledeci;
    struct dCvor *prethodni;
} tDCvor;
```

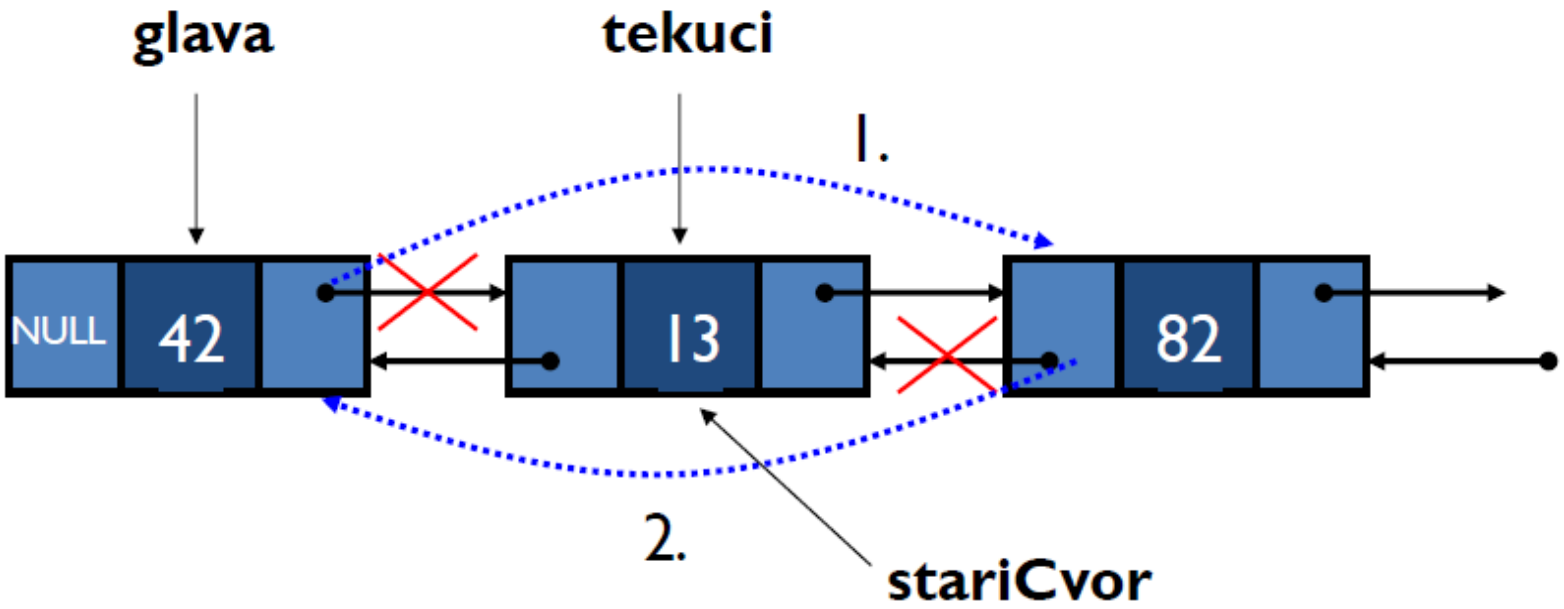


PRIMER DVOSTRUKO SPREGNUTE LISTE

- Listi se pristupa preko eksternog pokazivača start koji ukazuje na prvi element liste i koji se naziva i glava (engl. head) liste
- Prvi čvor dvostruko povezane liste u polju prethodni i poslednji čvor u polju sledeći imaju vrednost NULL koja nije validna adresa

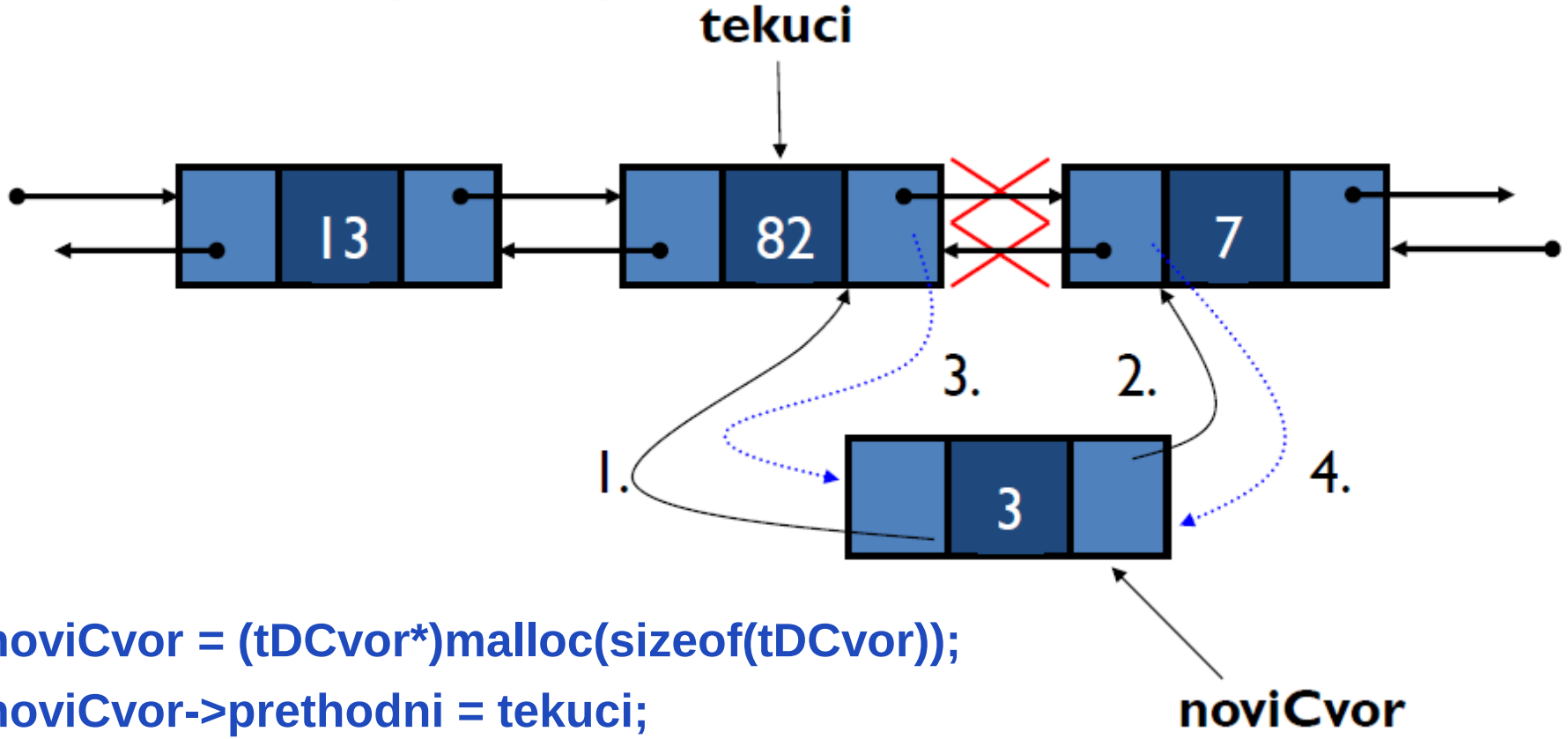


BRISANJE ELEMENTA




```
stariCvor = tekuci;
stariCvor->prethodni->sledeci = stariCvor->sledeci;
stariCvor->sledeci->prethodni = stariCvor->prethodni;
free(stariCvor);
tekuci = glava;
```

DODAVANJE ELEMENTA




```
noviCvor = (tDCvor*)malloc(sizeof(tDCvor));
noviCvor->prethodni = tekuci;
noviCvor->sledeci = tekuci->sledeci;
noviCvor->prethodni->sledeci = noviCvor;
noviCvor->sledeci->prethodni = noviCvor;
tekuci = noviCvor;
```

SPREGNUTA LISTA – PRIMER

Zadatak 1:

Napisati u jeziku C program za rad sa dvostruko spregnutom listom. Implementirati operacije za dodavanje elemenata na početak i kraj liste, kao i obilazak liste unapred i unazad.

Vežba 1:

Dopuniti prethodni program da uključuje operacije brisanja elementa sa početka i kraja liste, kao i operacije pretraživanje i sortiranja liste u rastućem redosledu.

Vežba 2:

Izmeniti prethodni program tako da korisnik može da bira željenu operaciju za rad sa dvostruko spregnutom listom. Omogućiti ponavljanje izabranih operacija sve dok korisnik ne odluči da izađe iz programa.

SPREGNUTA LISTA – ZADATAK 1 …


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

typedef struct dCvor {
        int podatak;
        struct dCvor* sledeci;
        struct dCvor* prethodni;
} tDCvor;

tDCvor* glava; // Globalna promenljiva - pokazivac na pocetak liste

// Prototipovi implementiranih funkcija
tDCvor* kreirajCvor(int);
void dodajNaPocetak(int);
void dodajNaKraj(int);
void obilazakUnapred();
void obilazakUnazad();
```

… SPREGNUTA LISTA – ZADATAK 1 …


```
// Kod za testiranje implementacije dvostruko spregnute liste
int main() {
    glava = NULL; // Prazna lista, glava se postavlja NULL.
    // Dodavanje cvorova i potom obilazak liste unapred i unazad
    dodajNaKraj(2); obilazakUnapred(); obilazakUnazad();
    dodajNaKraj(4); obilazakUnapred(); obilazakUnazad();
    dodajNaPocetak(1); obilazakUnapred(); obilazakUnazad();
    dodajNaKraj(8); obilazakUnapred(); obilazakUnazad();
    dodajNaPocetak(3); obilazakUnapred(); obilazakUnazad();
}

tDCvor* kreirajCvor(int x) { // Kreira novi cvor i vraca pokazivac na njega
    tDCvor* noviCvor = (tDCvor*)malloc(sizeof(tDCvor));
    noviCvor->podatak = x;
    noviCvor->sledeci = NULL;
    noviCvor->prethodni = NULL;
    return noviCvor;
}
```

… SPREGNUTA LISTA – ZADATAK 1 …


```
void dodajNaPocetak(int x) {        // Dodaje novi cvor na pocetak dvostruko spregnute liste
    tDCvor* noviCvor = kreirajCvor(x);
    if (glava == NULL) {            // Ako je lista prazna, novi cvor dodaje se odmah na pocetku
        glava = noviCvor; return;
    }
    glava->prethodni = noviCvor;
    noviCvor->sledeci = glava;
    glava = noviCvor;
}
void dodajNaKraj(int x) {           // Dodaje novi cvor na kraj dvostruko spregnute liste
    tDCvor* tekuci = glava;
    tDCvor* noviCvor = kreirajCvor(x);
    if (glava == NULL) {            // Ako je lista prazna, novi cvor dodaje se odmah na pocetku
        glava = noviCvor; return;
    }
    while (tekuci->sledeci != NULL)
        tekuci = tekuci->sledeci;   // Idi do poslednjeg cvora
    tekuci->sledeci = noviCvor;
    noviCvor->prethodni = tekuci;
}
```

… SPREGNUTA LISTA – ZADATAK 1 …


```
// Obilazi dvostruko spregnutu listu unapred
void obilazakUnapred() {
    tDCvor* tekuci = glava;
    printf("Obilazak dvostruko spregnute liste unapred: ");
    while (tekuci != NULL) {
        printf("%d ", tekuci->podatak);
        tekuci = tekuci->sledeci;
    }
    printf("\n");
}
```

… SPREGNUTA LISTA – ZADATAK 1


```
// Obilazi dvostruko spregnutu listu unazad
void obilazakUnazad() {
    tDCvor* tekuci = glava;
    if (tekuci == NULL) return;            // Prazna lista
    while (tekuci->sledeci != NULL) {  // Idi do poslednjeg cvora
        tekuci = tekuci->sledeci;
    }
    // Obilazak unazad koriscenjem linka prethodni
    printf("Obilazak dvostruko spregnute liste unazad: ");
    while (tekuci != NULL) {
        printf("%d ", tekuci->podatak);
        tekuci = tekuci->prethodni;
    }
    printf("\n");
}
```

SPREGNUTE LISTE – REZIME

- Vrste spregnutih listi – jednostruke, dvostruke, cirkularne, necirkularne, sa i bez zaglavlja, uređene i neuređene
- Prednost spregnutih listi kao linearne strukture podataka su efikasne operacije dodavanja i brisanja čvorova u listi, tj. efikasna promena veličine spregnute liste
- Nedostatak spregnutih listi su spora operacija pristupa slučajnom elementu u listi, kao i povećani memorijski zahtevi u odnosu na polja, pogotovu kod dvostruko spregnutih listi

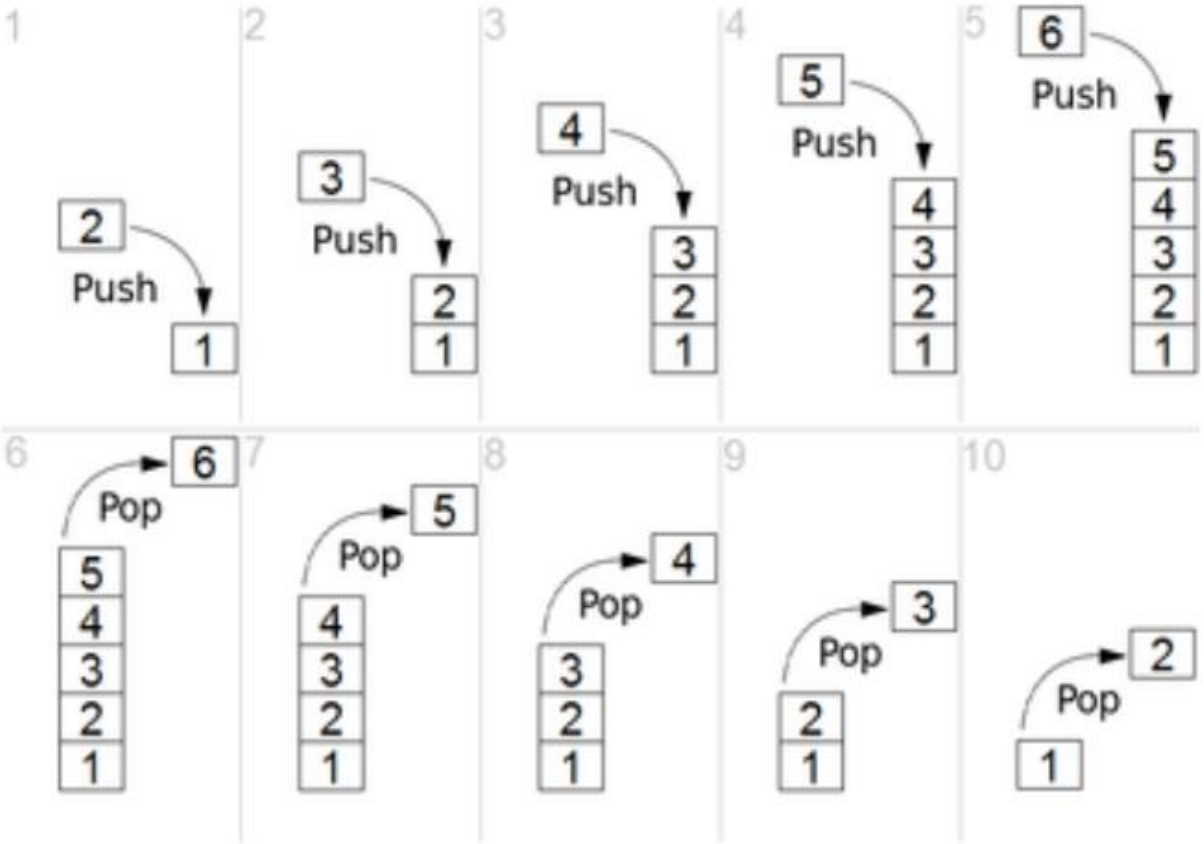
STEK

STEK

- Stek (engl. stack) ili magacin je apstraktni tip podataka koji predstavlja kontejner u koji se elementi dodaju ili iz koga se elementi brišu po principu “poslednji stigao, prvi izašao” (engl. last in, first out - LIFO)
- Stek ima linearnu strukturu i veoma široku primenu
- Posebne operacije za rad sa stekom – smesti (engl. push) i skini (engl. pop)
- Realizacija steka može biti:
  - sekvencijalna (polje)
  - spregnuta (spregnuta lista)

PRIMER RADA SA STEKOM

- Operacije **push** i **pop**



Izvor: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)#/media/File:Lifo_stack.png

OPERACIJE SA STEKOM …

- **push (smesti)** – dodavanje elementa na vrh steka
- **pop (skini)** – preuzimanje elementa koji je poslednji smeštan na stek
- Ostale operacije:
  - **peek (proveri)** – provera vrednosti na vrhu steka bez modifikacije stanja steka
  - **isEmpty (daLiJePrazan)** – provera da li je stek prazan
  - **isFull (daLiJePun)** – provera da li je stek pun
  - **size (vratiVelicinu)** – vraća veličinu steka

… OPERACIJE SA STEKOM

- Kada se stek posmatra kao linearna struktura podataka (sekvencijalna kolekcija), sve operacije sa stekom se realizuju isključivo na jednom kraju strukture koji se naziva vrh steka
- Usled prethodnog, stek se može implementirati kao jednostruko spregnuta lista sa pokazivačem na element koji se nalazi na vrhu steka
- Realizacija steka ima ograničen kapacitet. Ako je stek pun, onda nema prostora za prihvatanje elementa koji se prosleđuje operacijom push i tada kažemo da je stek u stanju prepunjenosti (engl. stack overflow)

IMPLEMENTACIJA STEKA – PRIMERI

Zadatak 1:

Napisati u jeziku C program kojim se stek realizuje sekvencijalno (u vidu polja). Implementirati operacije za inicijalizaciju i uništavanje steka, dodavanje i brisanje elemenata i proveru da li je stek pun i prazan.

Zadatak 2:

Napisati u jeziku C program kojim se stek realizuje spregnuto (u vidu jednostruko spregnute liste). Implementirati operacije za inicijalizaciju steka, dodavanje i brisanje elemenata, proveru da li je stek prazan, vraćanje informacije o veličini steka i prikaz sadržaja steka.

Vežba 1:

Izmeniti prethodne programe tako da korisnik može da bira željenu operaciju za rad sa stekom. Omogućiti ponavljanje izabranih operacija sve dok korisnik ne odluči da izađe iz programa.

SEKVENCIJALNI STEK – ZADATAK 1 …


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

typedef struct {
    char *podaci;
    int maxVelicina;
    int vrh;
} tStek;

void inicijalizacijaSteka(tStek *, int);
void unistavanjeSteka(tStek * );
void push(tStek *, char);
char pop(tStek *);
int daLiJePrazan(tStek *);
int daLiJePun(tStek *);
```

… SEKVENCIJALNI STEK – ZADATAK 1 …


```
// Primer rada sa sekvencijalno implementiranim stekom
int main() {
	int pom;
	tStek noviStek;
	inicijalizacijaSteka(&noviStek, 3);
	push(&noviStek, 'A');
	push(&noviStek, 'B');
	push(&noviStek, 'C');
	pom = daLiJePun(&noviStek); printf("%d ", pom);
	pop(&noviStek);
	pop(&noviStek);
	pop(&noviStek);
	pom = daLiJePun(&noviStek); printf("%d ", pom);
	pom = daLiJePrazan(&noviStek); printf("%d ", pom);
	unistavanjeSteka(&noviStek);
	return 0;
}
```

… SEKVENCIJALNI STEK – ZADATAK 1 …


```
void inicijalizacijaSteka(tStek * pStek, int maxVelicina){
        char *noviPodaci; // Alokacija novog polja za smestanje sadrzaja
        noviPodaci = (char*)calloc(maxVelicina, sizeof(char));
        if (noviPodaci == NULL) {
            fprintf(stderr, "Nedovoljno memorije za inicijalizaciju steka!\n");
            exit(1);
        }
        pStek ->podaci = noviPodaci;
        pStek ->maxVelicina = maxVelicina;
        pStek ->vrh = -1; // Stek je prazan
}

void unistavanjeSteka(tStek *pStek){
        free(pStek ->podaci);
        pStek ->podaci = NULL;
        pStek ->maxVelicina = 0;
        pStek ->vrh = -1;
}
```

… SEKVENCIJALNI STEK – ZADATAK 1 …


```
void push(tStek *pStek, char element) {
	if (daLiJePun(pStek)) {
		fprintf(stderr, "Element se ne moze smestiti: stek je pun!\n");
		exit(1);
	}
	// Smestanje elementa na vrh steka i azuriranje vrha steka
	pStek ->podaci[++pStek ->vrh] = element;
}

char pop(tStek *pStek) {
	if (daLiJePrazan(pStek)) {
		fprintf(stderr, "Element se ne moze preuzeti: stek je prazan!\n");
		exit(1);
	}
	return pStek ->podaci[pStek ->vrh--]; //Preuzimanje elementa sa vrha
}
```

… SEKVENCIJALNI STEK – ZADATAK 1


```
// Funkcija za proveru da li je stek prazan – vraca 1 ako je prazan, 0 ako nije
int daLiJePrazan(tStek *pStek) {
    return pStek->vrh < 0;
}

// Funkcija za proveru da li je stek pun – vraca 1 ako je pun, 0 ako nije
int daLiJePun(tStek *pStek)
{
    return pStek->vrh >= pStek->maxVelicina - 1;
}
```

SPREGNUTI STEK – ZADATAK 2 …


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

struct cvor {
int podatak;
struct cvor *sledeci;
}*vrh;

void inicijalizacijaSteka();
int daLiJePrazan();
int peek();
void push();
void pop();
int vratiVelicinuSteka(struct cvor*);
void prikaziStanjeSteka(struct cvor*);
```

… SPREGNUTI STEK – ZADATAK 2 …


```
// Primer rada sa spregnuto implementiranim stekom
int main() {
    // Inicijalizacija steka
    inicijalizacijaSteka();
    // Smestanje elemenata na stek
    push(1); prikaziStanjeSteka(vrh);
    push(2); prikaziStanjeSteka(vrh);
    push(3); prikaziStanjeSteka(vrh);
    push(4); prikaziStanjeSteka(vrh);
    printf("Velicina steka: %d\n", vratiVelicinuSteka(vrh));
    printf("\nElement na vrhu steka: %d\n", peek());
    printf("Stek kao jednostruko spregnuta lista:\n");
    prikaziStanjeSteka(vrh);
    // Uklanjanje elemenata sa steka
    pop(); prikaziStanjeSteka(vrh);
    pop(); prikaziStanjeSteka(vrh);
    pop(); prikaziStanjeSteka(vrh);
    pop(); prikaziStanjeSteka(vrh);
    pop(); prikaziStanjeSteka(vrh);
    return 0;
}
```

… SPREGNUTI STEK – ZADATAK 2 …


```
inicijalizacijaSteka() {
    vrh = NULL;
}
int daLiJePrazan() {
    if (vrh == NULL)
        return 1;
    else
        return 0;
}
int peek() {
    return vrh->podatak;
}
```

… SPREGNUTI STEK – ZADATAK 2 …


```
void push(int element) {
    struct cvor *tmp;
    tmp = (struct cvor *)malloc(1 * sizeof(struct cvor));
    tmp->podatak = element;
    if (vrh == NULL) {
        vrh = tmp;
        vrh->sledeci = NULL;
    }
    else {
        tmp->sledeci = vrh;
        vrh = tmp;
    }
}
```

… SPREGNUTI STEK – ZADATAK 2 …


```
void pop() {
    struct cvor *tmp;
    if (daLiJePrazan()) {
        printf("\nStek je prazan!\n"); return;
    }
    else {
        tmp = vrh;
        vrh = vrh->sledeci;
        printf("Sa steka je uklonjen element: %d\n", tmp->podatak);
        free(tmp);
    }
}
```

… SPREGNUTI STEK – ZADATAK 2 …


```
int vratiVelicinuSteka(struct cvor *glava){
    if (glava == NULL) {
        printf("Greska: pokazivac na stek nije validan!\n");
        return -1;
    }
    struct cvor* tek = glava;
    int duzina = 0;
    while (tek != NULL){
        tek = tek->sledeci;
        duzina++;
    }
    return duzina;
}
```

… SPREGNUTI STEK – ZADATAK 2


```
void prikaziStanjeSteka(struct cvor *tek) {
    while (tek != NULL) {
        printf("%d", tek->podatak);
        tek = tek->sledeci;
        if (tek != NULL)
            printf("-->");
    }
    printf("\n");
}
```