Spregnuta lista – proširenja

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

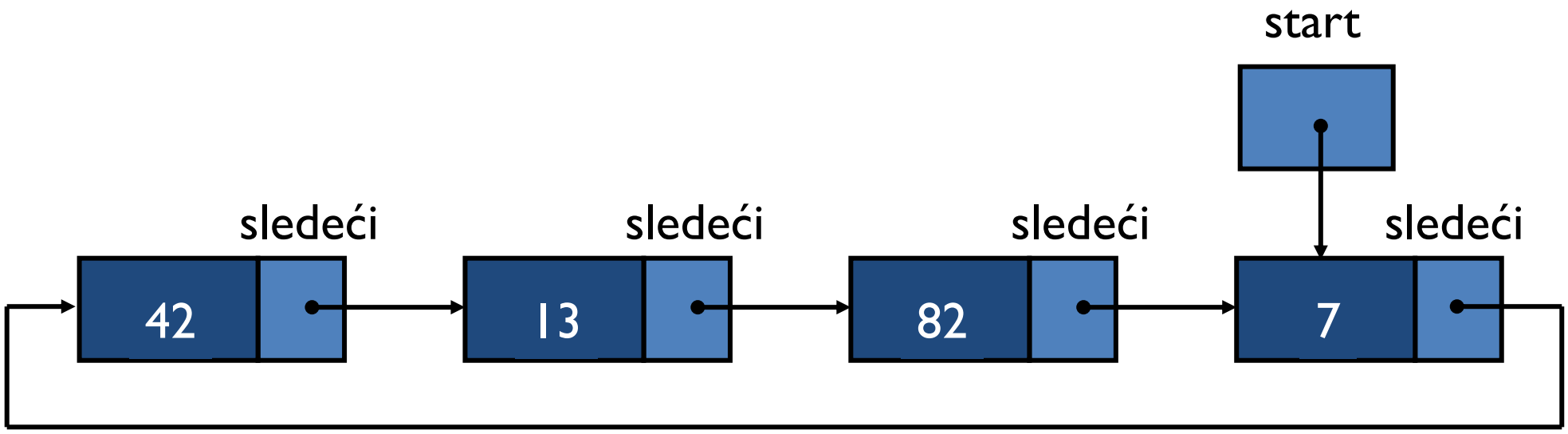
U slučaju kružne liste jednostavnije se izvode operacije kao što su pretraživanje od nekog unutrašnjeg čvora ka bilo kom delu liste

Stek (magacin) i **red** često se implementiraju kao **cirkularne spregnute liste**

Cirkularna spregnuta lista

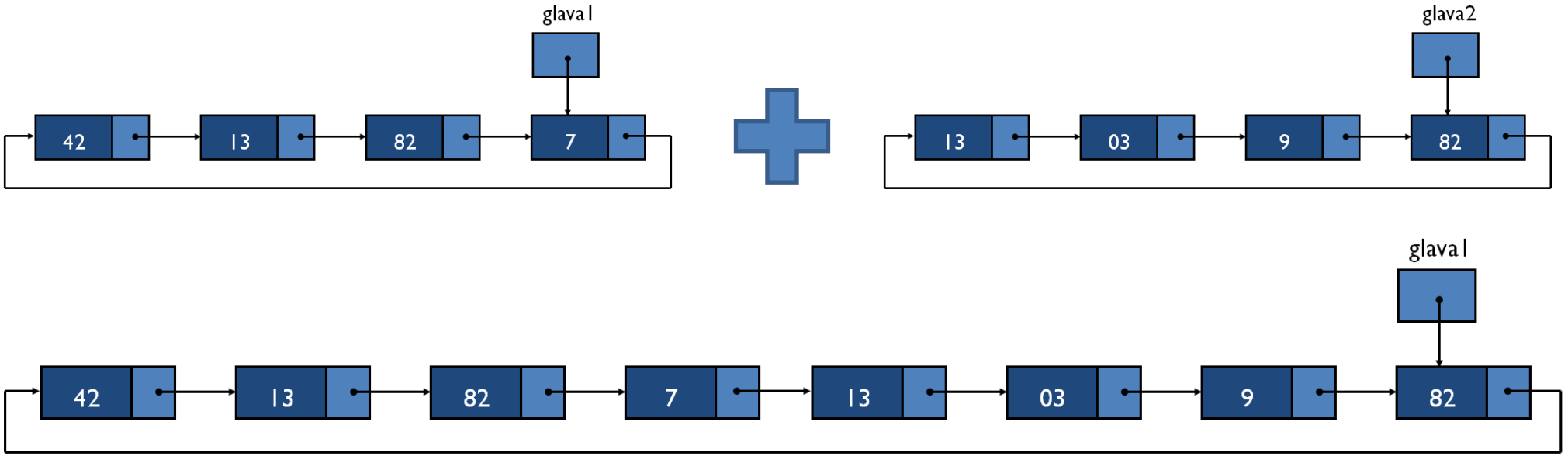
Zbog osobine simetričnosti kružne liste termini "prvi" i "poslednji" čvor su upitni

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



Cirkularna spregnuta lista

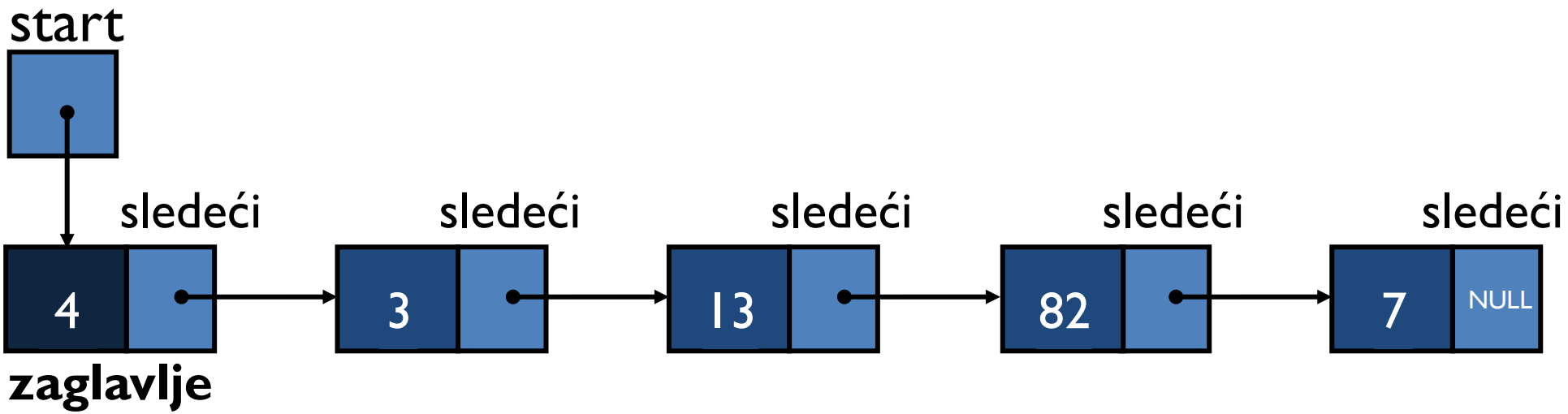
Operacija **spajanja dve liste** *spoji*(*glava1*, *glava2*) ne zahteva prolazak do kraja prve liste pošto pokazivač *glava1* ukazuje na poslednji čvor. Na kraju operacije pokazivač *glava1* ukazuje na poslednji čvor objedinjene liste



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 strukture $(S(DP), r_1(DP))$ i $(S(DP), r_2(DP))$ budu jednostruko spregnute liste

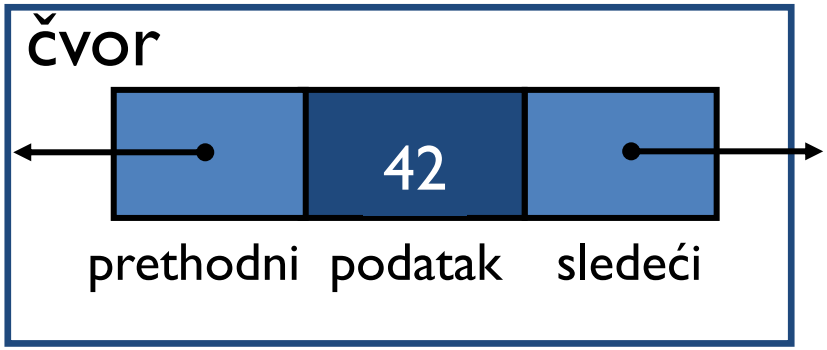
Čvor u dvostruko spregnutoj listi

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:

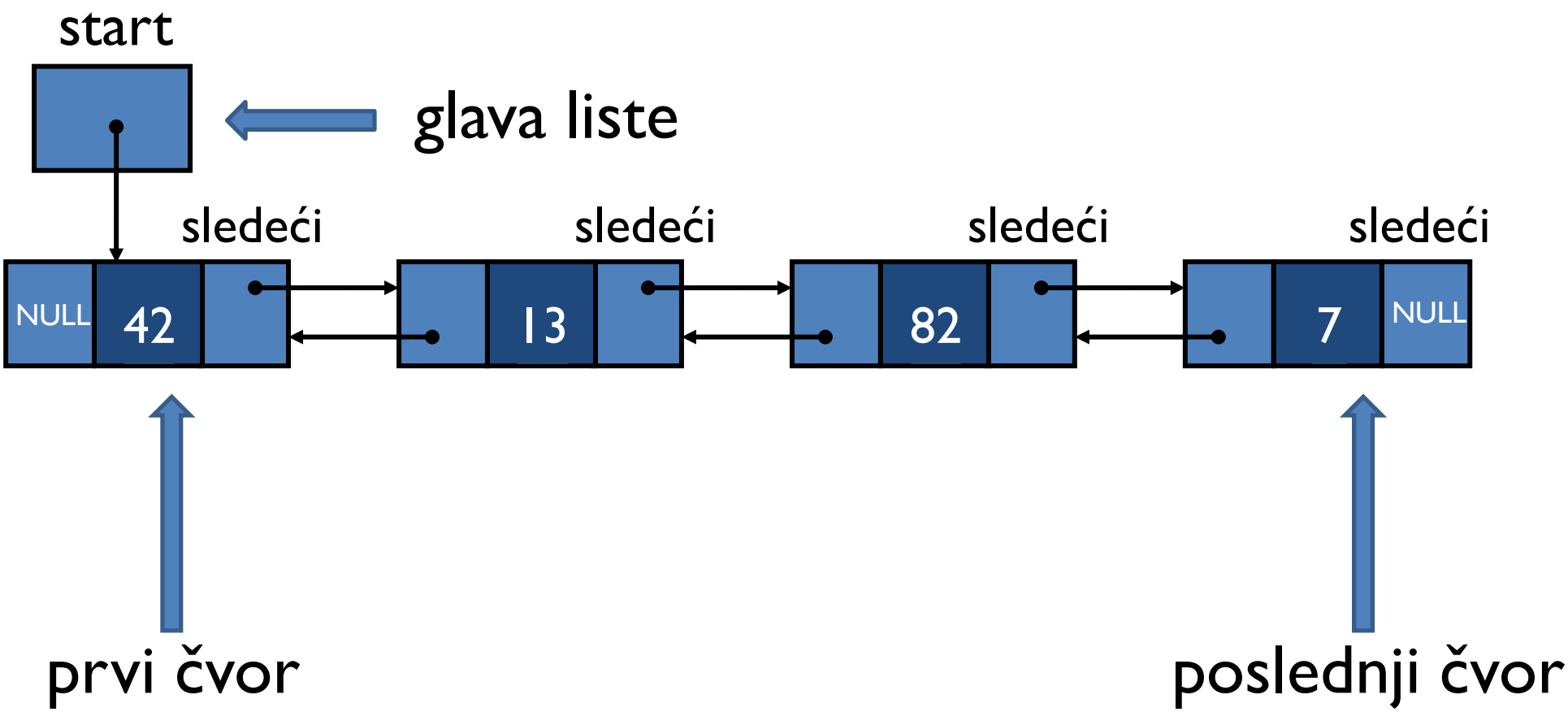
```
typedef struct dCvor {
        int podatak;
        struct dCvor *sledeci;
        struct dCvor *prethodni;
} tDCvor;
```



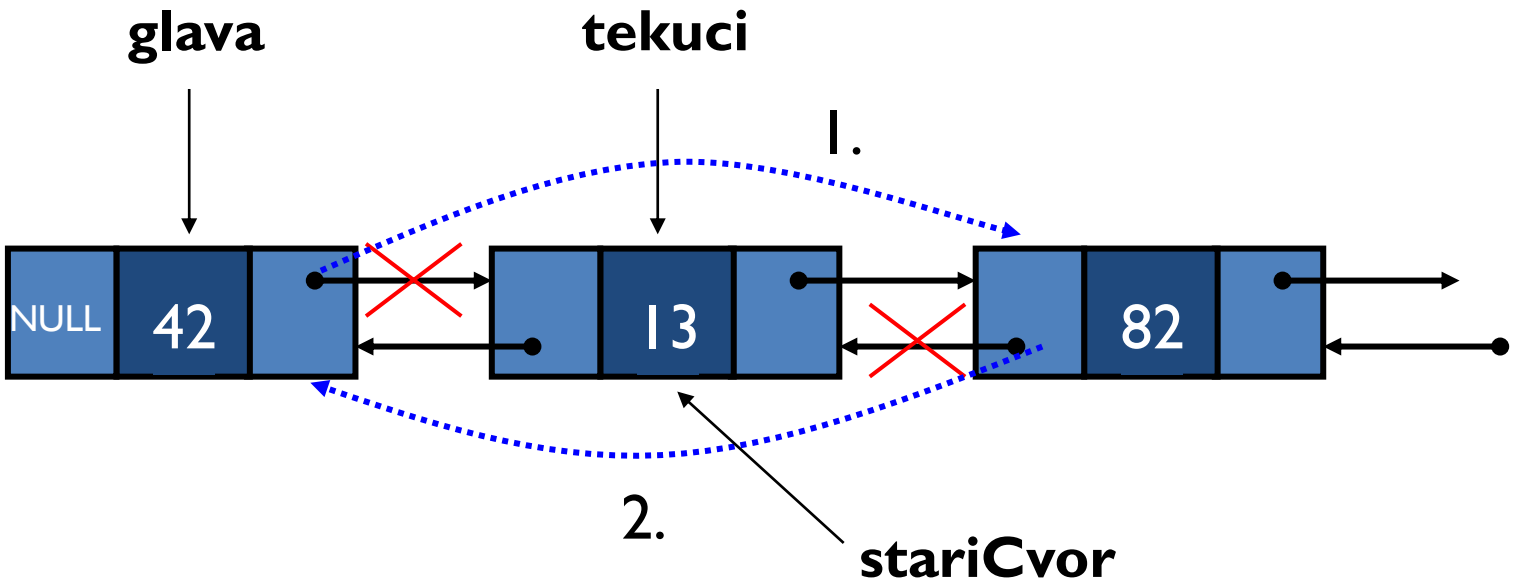
Listi se pristupa preko eksternog pokazivača *start* koji ukazuje na prvi element liste i koji se naziva i **glava** (engl. *head*) liste

Prvi čvor dvostruko povezane liste u polju *prethodni* i poslednji čvor u polju *sledeći* imaju vrednost **NULL** koja nije validna adresa

Primer dvostruko spregnute liste

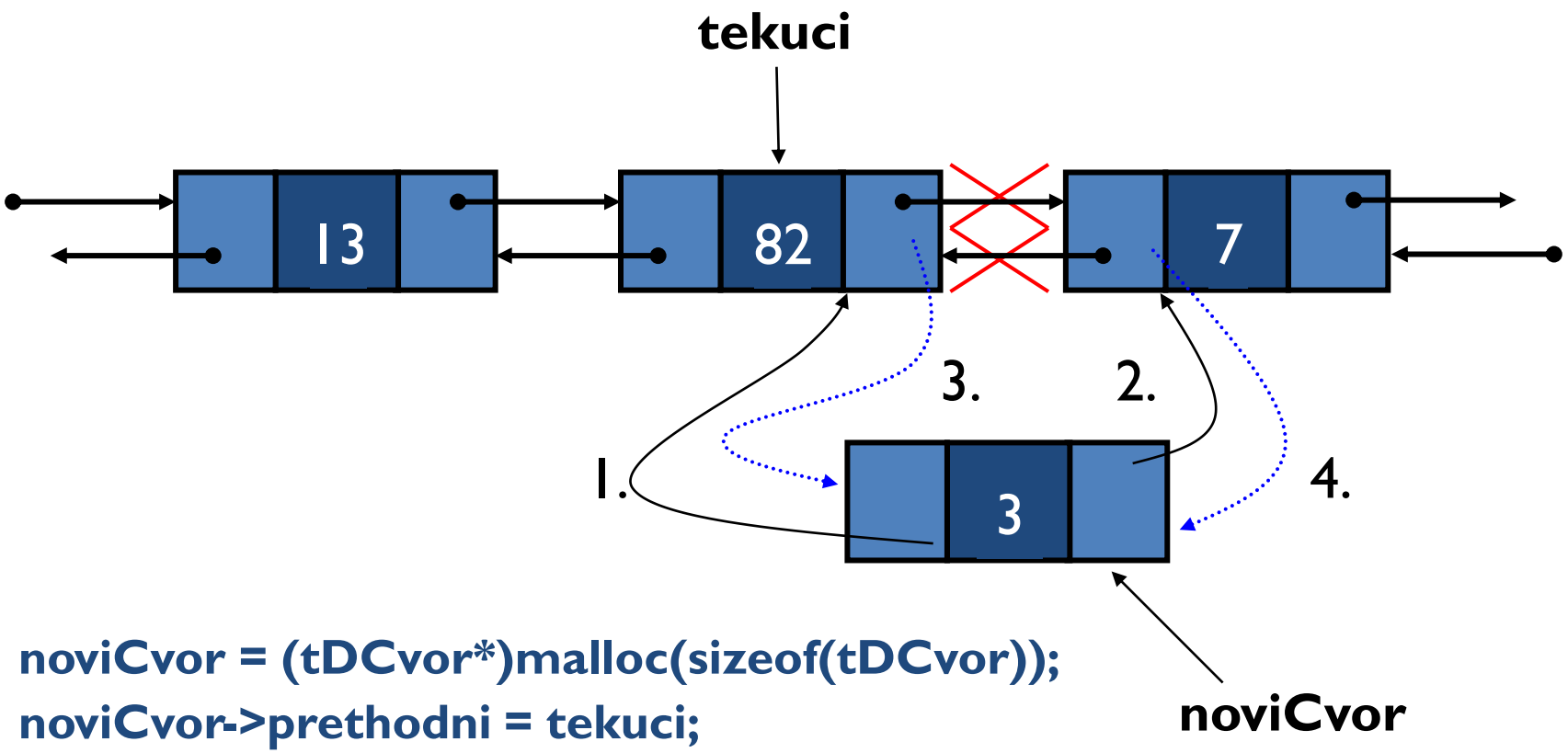


Brisanje elementa




```
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 I

```
#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**

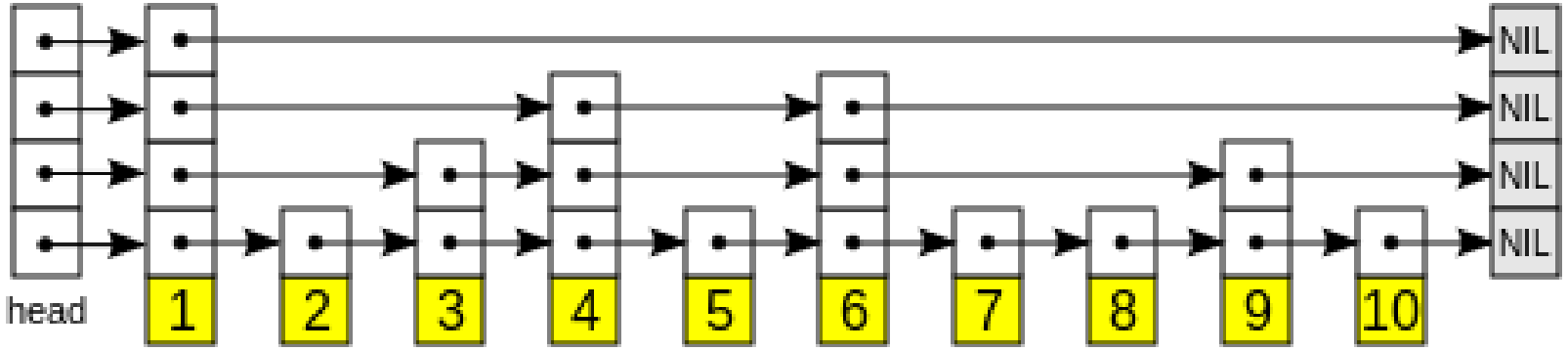
```
// Obilazi dvostruko spregnutu listu unazad

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");
}
```

Lista sa preskokom

Lista sa preskokom (engl. *skip list*) je probabilistička struktura podataka zasnovana na spregnutim listama koja omogućava **brzo pretraživanje uređene sekvence elemenata**

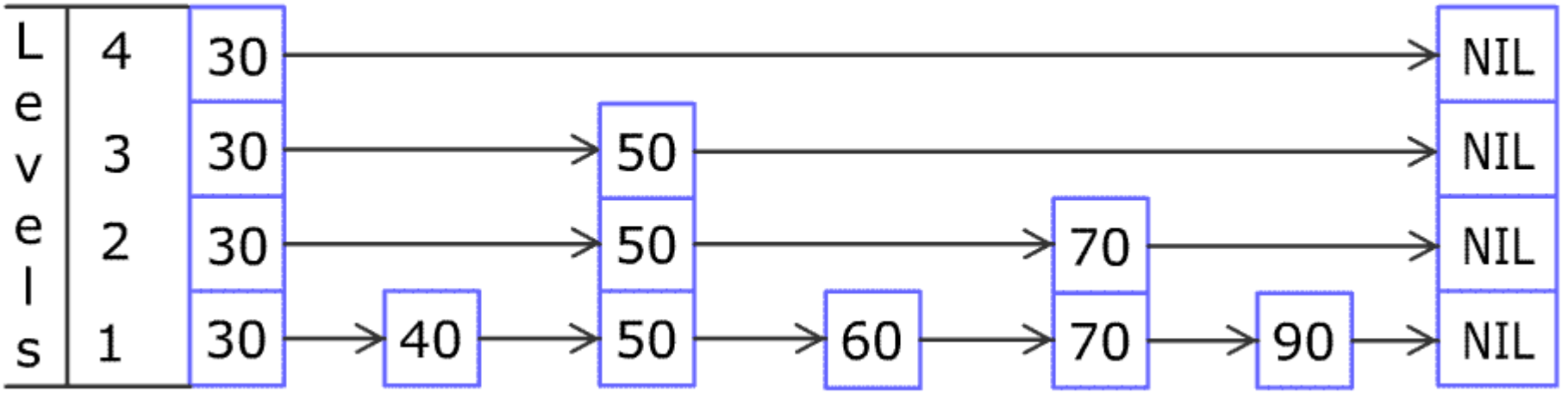
Brzo pretraživanje ($O(\log N)$) je omogućeno **spregnutom hijerarhijom podsekvenci**, kod koje svaka sledeća podsekvenca preskače manje elemenata od prethodne



Izvor: https://en.wikipedia.org/wiki/Skip_list#/media/File:Skip_list.svg

Lista sa preskokom

Operacija ubacivanja novog elementa u skip listu



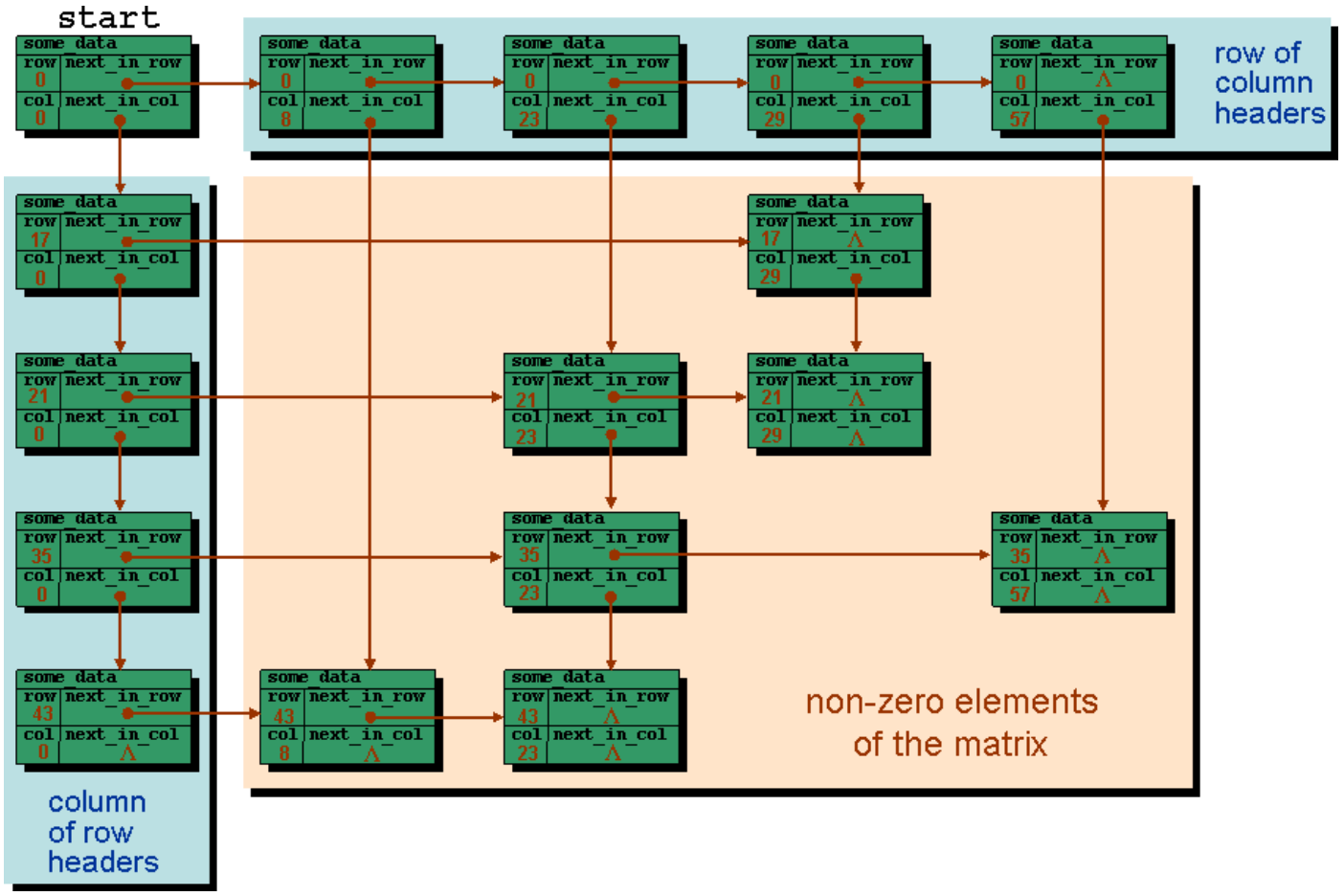
Izvor: https://en.wikipedia.org/wiki/Skip_list#/media/File:Skip_list_add_element-en.gif

Primer implementacije liste sa preskokom u jeziku C:

https://gist.github.com/zhpengg/2873424

Primene spregnutih listi

1. **Linearna algebra** – rešavanje sistema jednačina - predstavljanje retko-posednutih matrica (engl. *sparse matrices*)

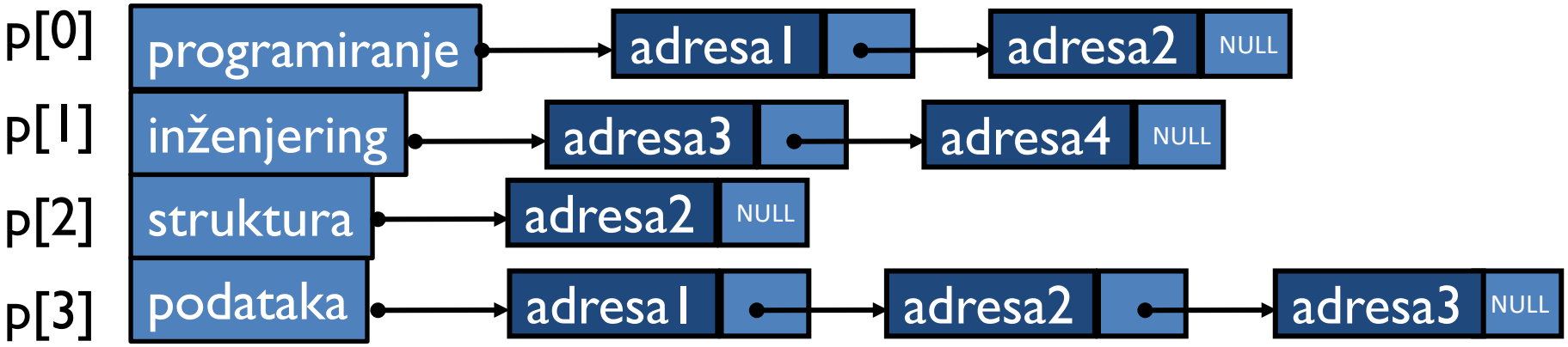


Izvor: http://ultra.pr.erau.edu/~jaffem/classes/cs315/cs315_homework/homework/sparse.htm

Primene spregnutih listi

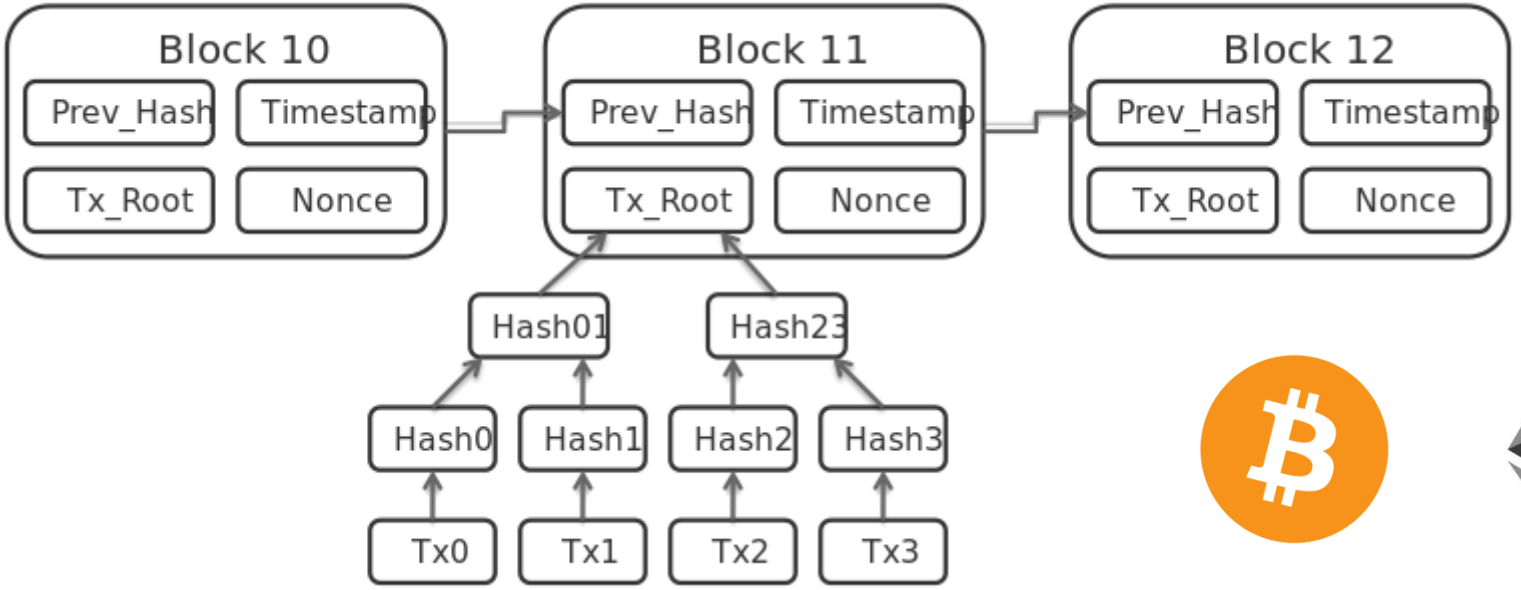
2. **Indeksiranje na Internetu (World Wide Web-u)** - niz spregnutih listi, elementi niza reči, svakom elementu niza odgovara spregnuta lista čiji čvorovi sadrže web adrese stranica (engl. *URL - Unified Resource Locator*) na kojima se reč pojavljuje

Niz pojmova *p*:



Primene spregnutih listi

3. **Distribuirani sistemi – blokčejn i kriptovalute** – struktura podataka u vidu **lanca blokova** (engl. *blockchain*) koji su povezani linkovima i čiji je sadržaj kriptografski obezbeđen



Izvor: https://i.stack.imgur.com/HrKX0.png

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

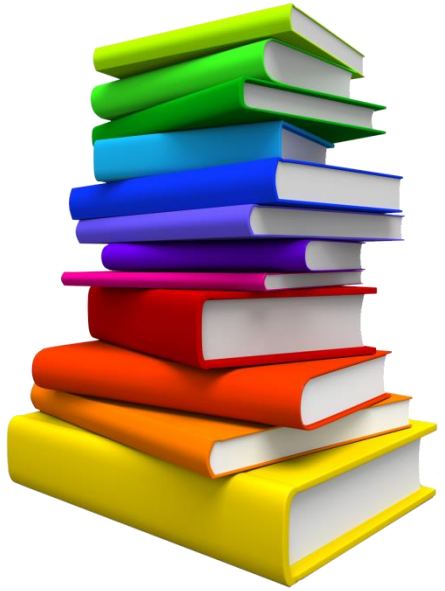
Stek (engl. *stack*) ili **magacin** je **apstraktni tip podataka** koji predstavlja kontejner u koji se **elementi dodaju ili iz koga se elementi brišu** po principu "**poslednji stigao, prvi izašao**" (engl. *last in, first out* - ***LIFO***)

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:

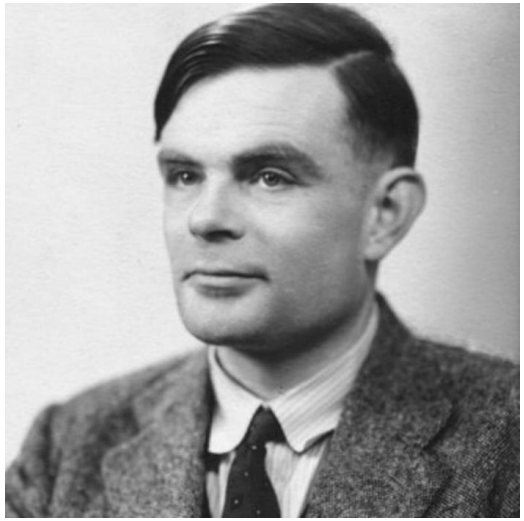
- **sekvencijalna (polje)**
- **spregnuta (spregnuta lista)**



Stek

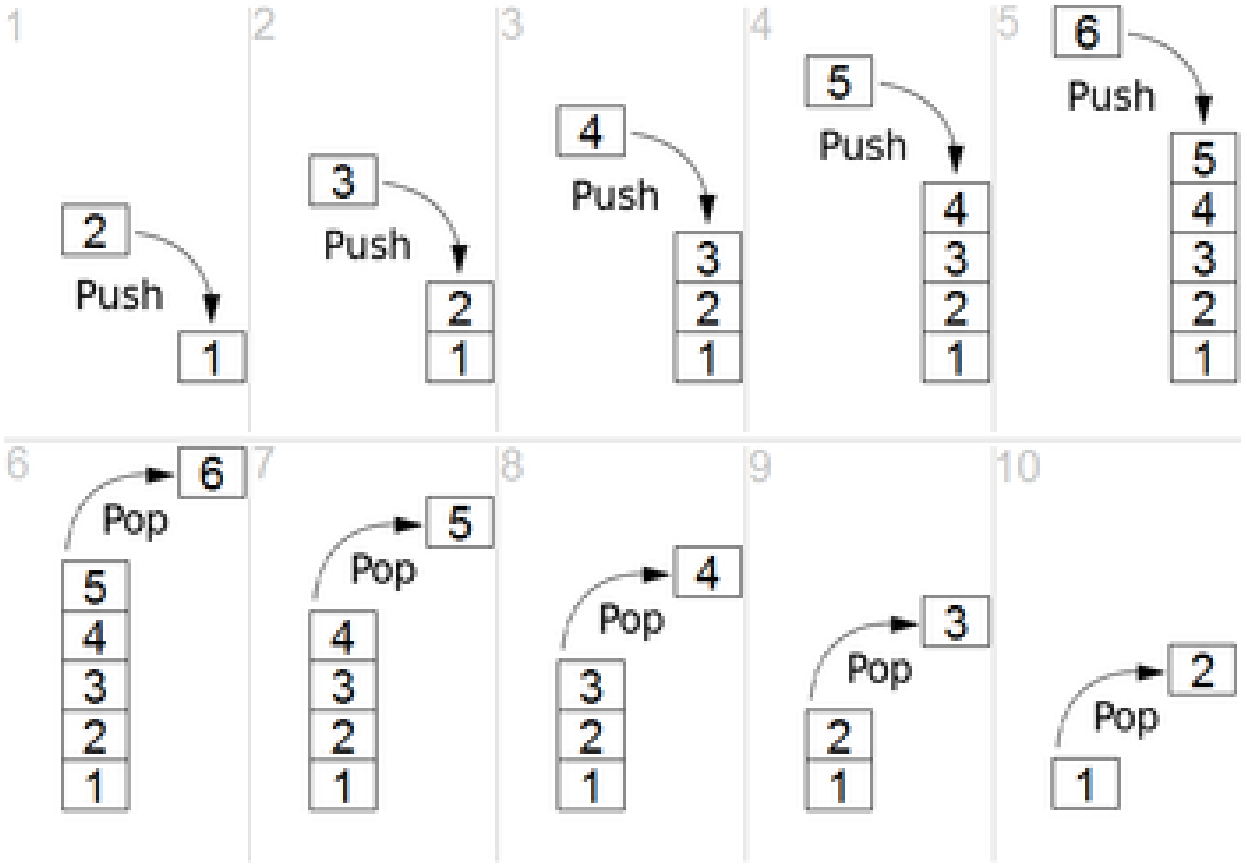
Pojam steka prvi put se pojavljuje u računarstvu 1946. kada je Alan Turing koristio termine "pokopati" (engl. *bury*) i "otkopati" (engl. *unbury*) kao sredstva prilikom poziva potprograma i povratka iz potprograma

Rad sa potprogramima pomoću steka prvi je implementirao Konrad Zuse u svom računaru Z4 1945.



Stek

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

```
void 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;
        }
}

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);
        }
}
```

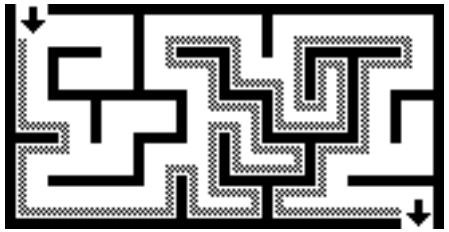
Spregnuti stek – Zadatak 2

```
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;
}

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

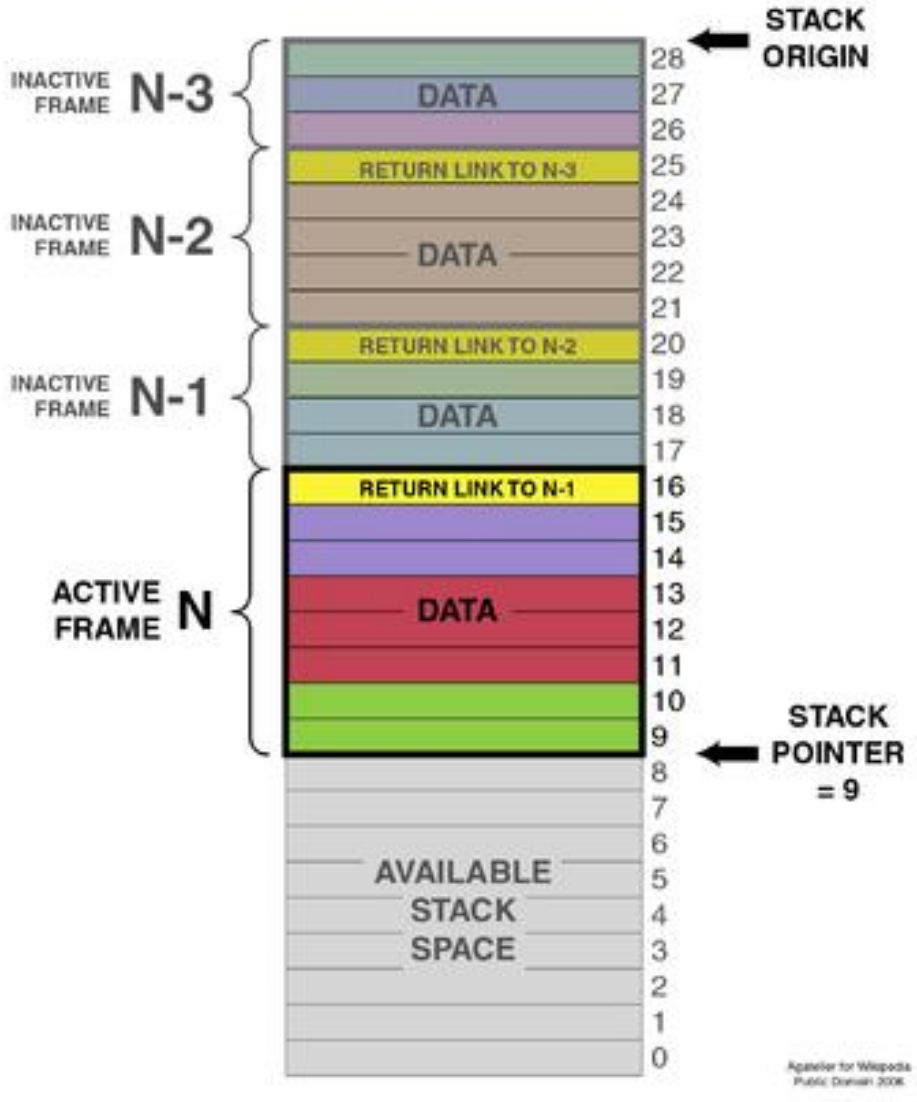
Primene steka

1. **Evaluacija izraza** – inverzna poljska notacija kod kalkulatora – konverzija između prefiksne, infiksne i sufiksne notacija primenom steka 3 – 4+5 ⇨ 3 4 – 5 +
2. **Parsiranje sintakse** u programskim prevodiocima – programski jezici zasnovani na bezkontekstnim gramatikama – razdvajanje koda na osnovne delove moguće pomoću steka
3. **Praćenje unazad** (engl. *backtracking*) – traženje puta u lavirintu – algoritam traženja po dubini (engl. *depth-first search*)



Primene steka

4. **Upravljanje memorijom u vreme prevođenja** – poziv funkcija u jeziku C, prilikom svakog poziva formira se odgovarajući stek (magacinski) okvir (engl. *stack frame*), izvršavanje operacija na Java virtuelnoj mašini zasnovano na steku



Izvor: https://en.wikipedia.org/wiki/Stack_(abstract_data_type)#/media/File:ProgramCallStack2_en.png

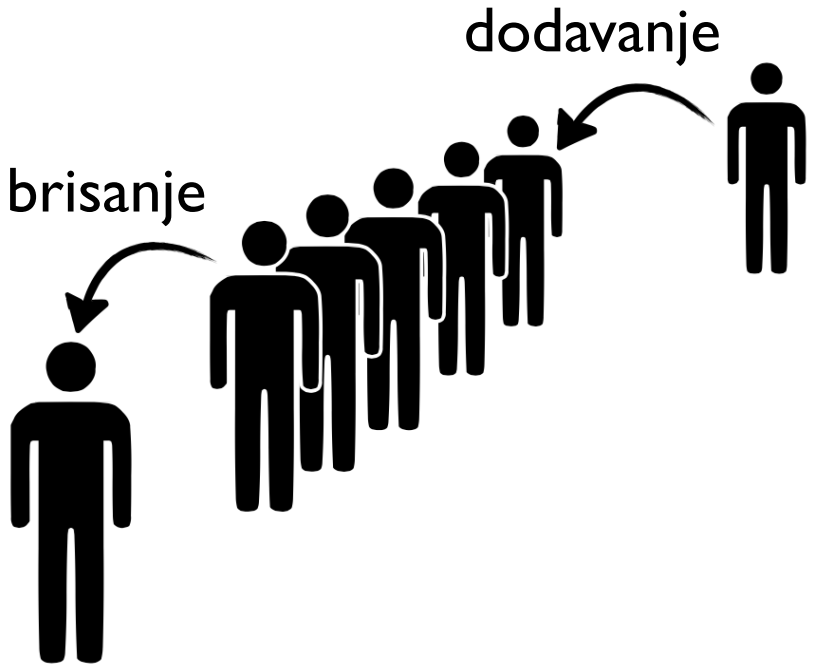
Red

Red

Red (engl. *queue*) je apstraktni tip podataka koji predstavlja kontejner u koji se **elementi dodaju ili iz koga se elementi brišu** po principu "**prvi stigao, prvi izašao**" (engl. *first in, first out* - ***FIFO***)

Red realizuje situacije u kojima **više korisnika čeka servis sa jednog izvora**

Dodavanje novog elementa u red vrši se na **kraju reda,** dok se **operacije pristupa i uklanjanja elementa** obavljaju na **početku reda**



Red

Red ima **linearnu strukturu** i veoma **široku praktičnu primenu** (npr. funkcija **bafera**)

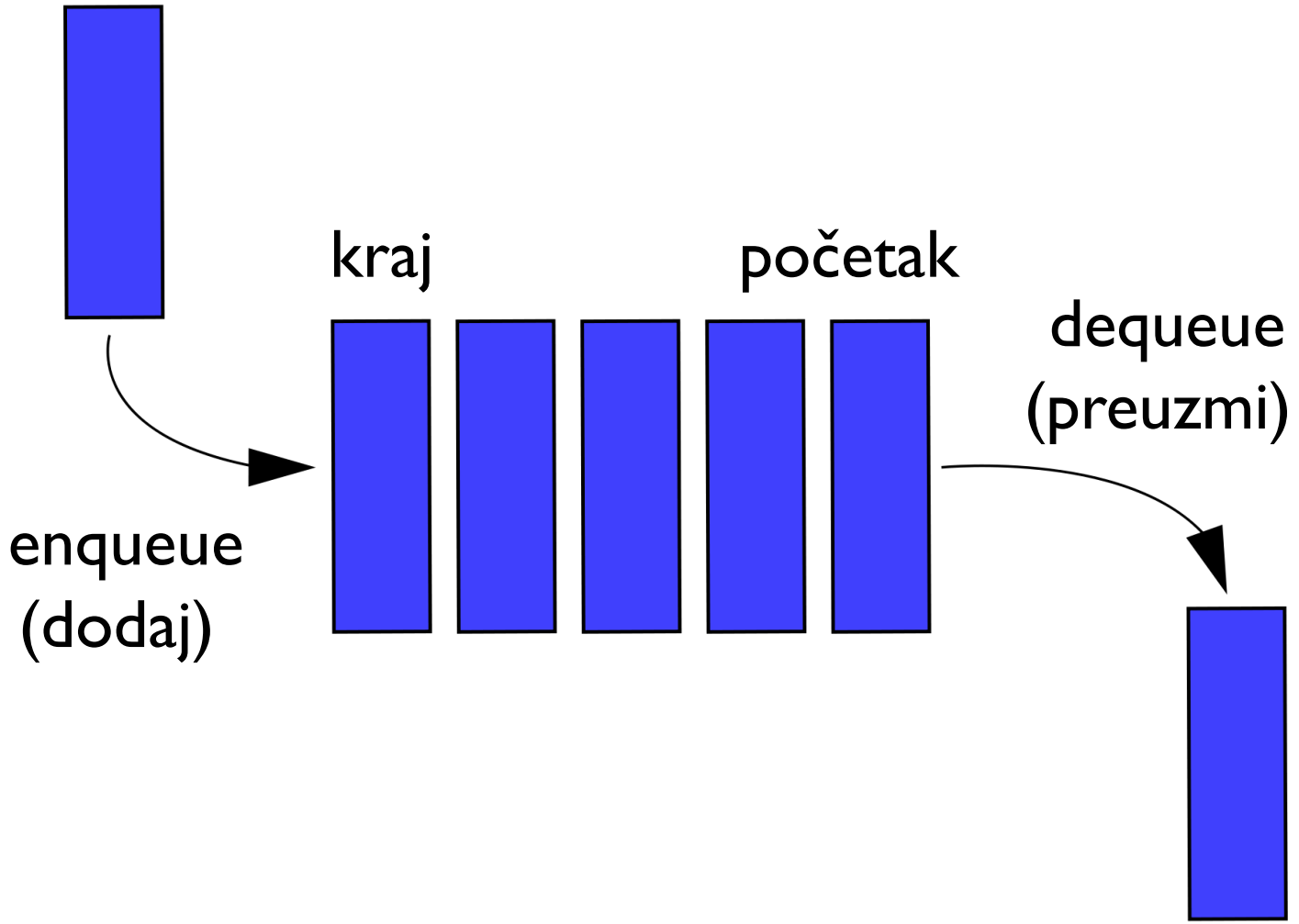
Posebne operacije za rad sa redom – **enqueue** (dodaj u red) i **dequeue** (preuzmi iz reda)

Realizacija reda može biti:

- **sekvencijalna (polje)**
  - Ako je *n* veličina polja, onda izračunavanjem indeksa po modulu *n* pretvaramo polje u krug
- **spregnuta (spregnuta lista)**
  - Prirodan izbor dvostruko spregnute liste, ali je moguća i realizacija sa jednostruko spregnutom uz dodavanje pokazivača na kraj

Red

Primer rada sa redom - operacije **enqueue** i **dequeue**:



Operacije sa redom

enqueue (dodaj) – dodavanje elementa na kraj reda

dequeue (preuzmi) – preuzimanje elementa sa početka reda

Ostale operacije:

peek ili **front (vratiPocetak)** – provera vrednosti na početku reda bez modifikacije stanja reda

rear (vratiKraj) – provera vrednosti na kraju reda bez modifikacije stanja reda

isEmpty (daLiJePrazan) – provera da li je red prazan

isFull (daLiJePun) – provera da li je red pun

size (vratiVelicinu) – vraća veličinu reda

Ciklični (kružni) red

Kod **cikličnog (kružnog, cirkularnog) reda** prilikom pristupa vraća se nazad na početak kada se dosegne kraj reda

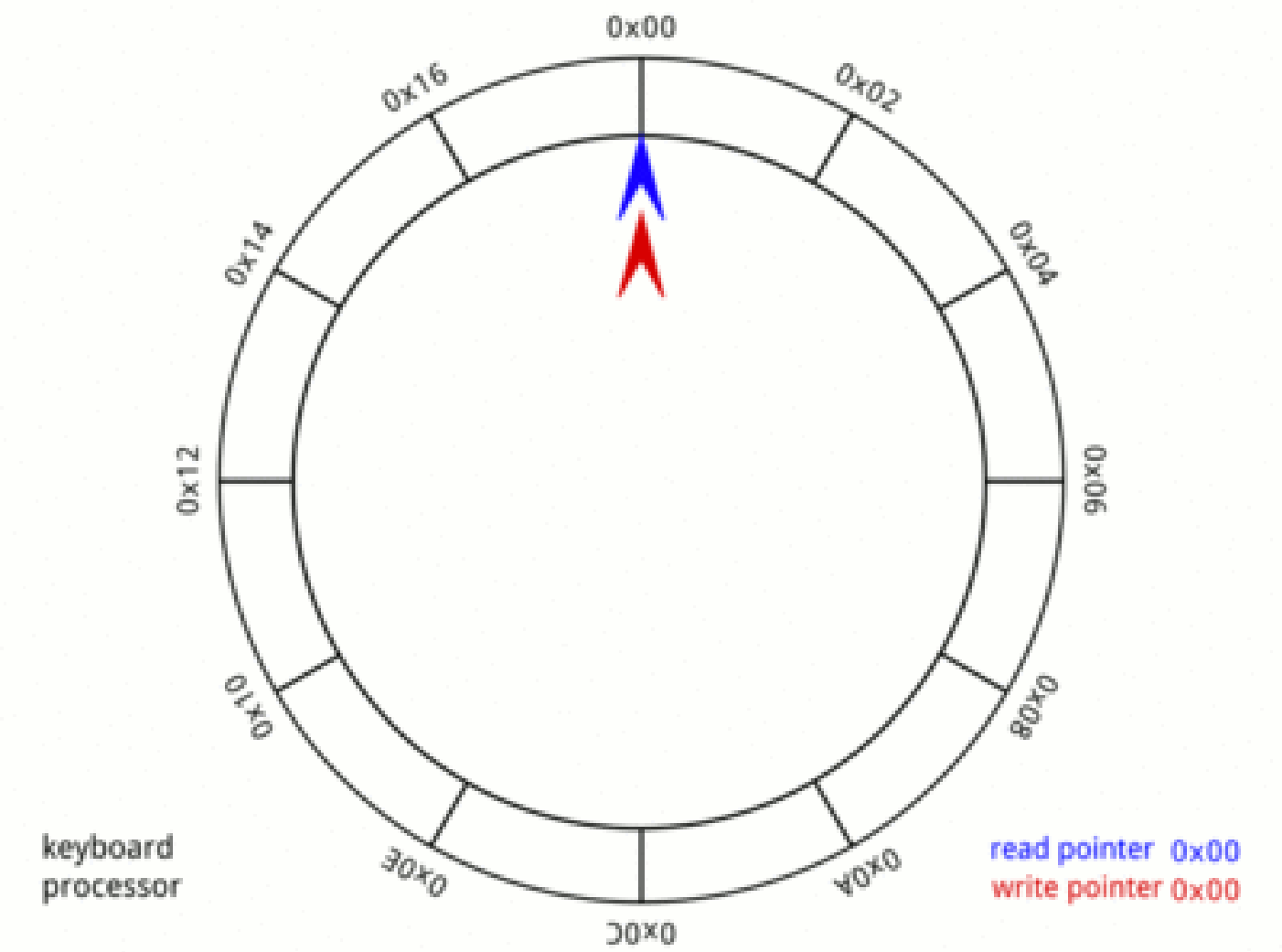
Primer kružnog reda je **cirkularni bafer** (engl. *circular or ring buffer*) **za tastaturu**, koji prima znakove brzinom kojom se kucaju, a procesu ih predaje na zahtev

Ako je proces zauzet nekim drugim poslom, ovi baferi omogućuju da se nastavi sa kucanjem, a da se ne izgubi sadržaj

Ako su **indeksi početka i kraja jednaki**, da li je **red pun ili prazan**?

Za razlikovanje ove dve situacije može se "žrtvovati" jedno mesto u redu - **indeks kraja** se **inicijalizuje na jedno mesto iza indeksa početka** i potom mu nije dozvoljeno da ga sustigne

Cirkularni bafer za tastaturu



Izvor: https://en.wikipedia.org/wiki/Circular_buffer#/media/File:Circular_Buffer_Animation.gif

Implementacija reda – primeri

Zadatak 1:

Napisati u jeziku C program kojim se red realizuje **sekvencijalno (u vidu polja)**. Implementirati operacije za inicijalizaciju reda, dodavanje i brisanje elemenata, pružanje informacija o prvom, odnosno poslednjem elementu u redu i proveru da li je red pun i prazan.

Zadatak 2:

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

Vežba 1:

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

Sekvencijalni red – Zadatak 1

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

typedef struct          // Struktura koja predstavlja red
{
        int* podaci;
        int pocetak, kraj, trenutnaVelicina;
        unsigned maxVelicina;
} tRed;

tRed* inicijalizacijaReda(unsigned);
int daLiJePun(tRed*);
int daLiJePrazan(tRed*);
void dodaj(tRed*, int);
int preuzmi(tRed*);
int vratiPocetak(tRed*);
int vratiKraj(tRed*);
```

Sekvencijalni red – Zadatak 1

```
// Primer rada sa sekvencijalno implementiranim redom
int main()
{
	tRed* red = inicijalizacijaReda(50);

	dodaj(red, 1);
	dodaj(red, 2);
	dodaj(red, 3);
	dodaj(red, 4);
	printf("%d preuzeto iz reda!\n", preuzmi(red));
	printf("Element na pocetku reda je %d.\n", vratiPocetak(red));
	printf("Element na kraju reda je %d.\n", vratiKraj(red));

	return 0;
}
```

Sekvencijalni red – Zadatak 1

```
tRed* inicijalizacijaReda(unsigned kapacitet)   // Funkcija za kreiranje reda zeljenog kapaciteta
{
        tRed* red = (tRed*) malloc(sizeof(tRed));
        red->maxVelicina = kapacitet;
        red->pocetak = red->trenutnaVelicina = 0;
        red->kraj = kapacitet - 1;
        red->podaci = (int*)malloc(red->maxVelicina * sizeof(int));
        return red;
}

int daLiJePun(tRed* red)
{
                return (red->trenutnaVelicina == red->maxVelicina);
}

int daLiJePrazan(tRed* queue)
{
                return (red->trenutnaVelicina == 0);
}
```

Sekvencijalni red – Zadatak 1

```
void dodaj(tRed* red, int element)
{
        if (daLiJePun(red))
                return;
        red->kraj = (red->kraj + 1) % red->maxVelicina;
        red->podaci[red->kraj] = element;
        red->trenutnaVelicina = red->trenutnaVelicina + 1;
        printf("%d dodat u red!\n", element);
}

int preuzmi(tRed* red)
{
        if (daLiJePrazan(red))
                return INT_MIN;
        int element = red->podaci[red->pocetak];
        red->pocetak = (red->pocetak + 1) % red->maxVelicina;
        red->trenutnaVelicina = red->trenutnaVelicina - 1;
        return element;
}
```

Sekvencijalni red – Zadatak 1

```
int vratiPocetak(tRed* red)
{
        if (daLiJePrazan(red))
                return INT_MIN;
        return red->podaci[red->pocetak];
}

int vratiKraj(tRed* red)
{
        if (daLiJePrazan(red))
                return INT_MIN;
        return red->podaci[red->kraj];
}
```

Spregnuti red – Zadatak 2

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

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

typedef struct
{
        tCvor *pocetak, *kraj;
} tRed;

tCvor* kreirajCvor(int);
tRed* kreirajRed();
int daLiJePrazan(tRed*);
void dodaj(tRed*, int);
tCvor* preuzmi(tRed*);
```

Spregnuti red – Zadatak 2

```
int main()
{
        tRed *red = kreirajRed();

        dodaj(red, 1);
        dodaj(red, 2);
        dodaj(red, 3);
        dodaj(red, 4);
        dodaj(red, 5);

        tCvor *n = preuzmi(red);
        if (n != NULL)
                printf("%d preuzeto iz reda!\n", n->podatak);
        return 0;
}
```

Spregnuti red – Zadatak 2

```
tCvor* kreirajCvor(int x)
{
        tCvor *noviCvor = (tCvor*)malloc(sizeof(tCvor));
        noviCvor->podatak = x;
        noviCvor->sledeci = NULL;
        return noviCvor;
}

tRed* kreirajRed()
{
        tRed *red = (tRed*)malloc(sizeof(tRed));
        red->pocetak = red->kraj = NULL;
        return red;
}

int daLiJePrazan(tRed* red)
{
        return (red->pocetak == NULL);
}
```

Spregnuti red – Zadatak 2

```
void dodaj(tRed* red, int x)
{
        // Kreiraj novi cvor
        tCvor *tekuci = kreirajCvor(x);
        // Ako je red prazan, onda je novi cvor i pocetak i kraj
        if (daLiJePrazan(red))
        {
                red->pocetak = red->kraj = tekuci;
                printf("%d dodat u red!\n", x);
                return;
        }
        // Dodaj novi cvor na kraj i azuriraj kraj
        red->kraj->sledeci = tekuci;
        red->kraj = tekuci;
        printf("%d dodat u red!\n", x);
        return;
}
```

Spregnuti red – Zadatak 2

```
tCvor* preuzmi(tRed* red)
{
        if (daLiJePrazan(red))
                return NULL;
        // Sacuvaj trenutni pocetak i pomeri se na novi pocetak
        tCvor *tekuci = red->pocetak;
        red->pocetak = red->pocetak->sledeci;

        // Ako pocetak postane NULL, onda i kraj treba da bude NULL
        if (red->pocetak == NULL)
                red->kraj = NULL;
        return tekuci;
}
```

Primene reda

- Programska simulacija bilo koje situacije iz realnog života u kojoj postoji koncept reda
    - kase u prodavnicama, bioskopima ili pozorištima, kontola leta, izvršavanje finansijskih transakcija...
- Obrada zahteva klijentskih računara od strane servera na Web-u
- Red čekanja procesa za procesorsko vreme ili za pristup printeru

Vrste redova

Običan red (FIFO) (engl. *queue*)

- necikličan
- cikličan

Dek (engl. *double-ended queue*)

Red sa prioritetom (engl. *priority queue*)

Dek sa prioritetom (engl. *priority dequeue)*

Dek

Dek (engl. *deqeue – double-ended queue*) je apstraktni tip podataka koji predstavlja kontejner u koji se **elementi mogu dodavati ili iz koga se elementi mogu brisati i na početku i na kraju reda – red sa dva početka/kraja**

Dek predstavlja **uopštenje steka i reda** u smislu načina pristupa, tj. dodavanja i uklanjanja elemenata

Najčešće se **implementira** pomoću **dvostruko spregnute liste** – (engl. *head/tail linked list*)

Operacije sa dekom

insertFirst (dodajPrvi) – dodavanje elementa na početak deka

removeFirst (preuzmiPrvi) – preuzimanje elementa sa početka deka

insertLast (dodajPoslednji) – dodavanje elementa na kraj deka

removeLast (preuzmiPoslednji) – preuzimanje elementa sa kraja deka

Ostale operacije:

first (vratiPrvi) – provera vrednosti na početku deka bez modifikacije deka

last (vratiPoslednji) – provera vrednosti na kraju deka bez modifikacije deka

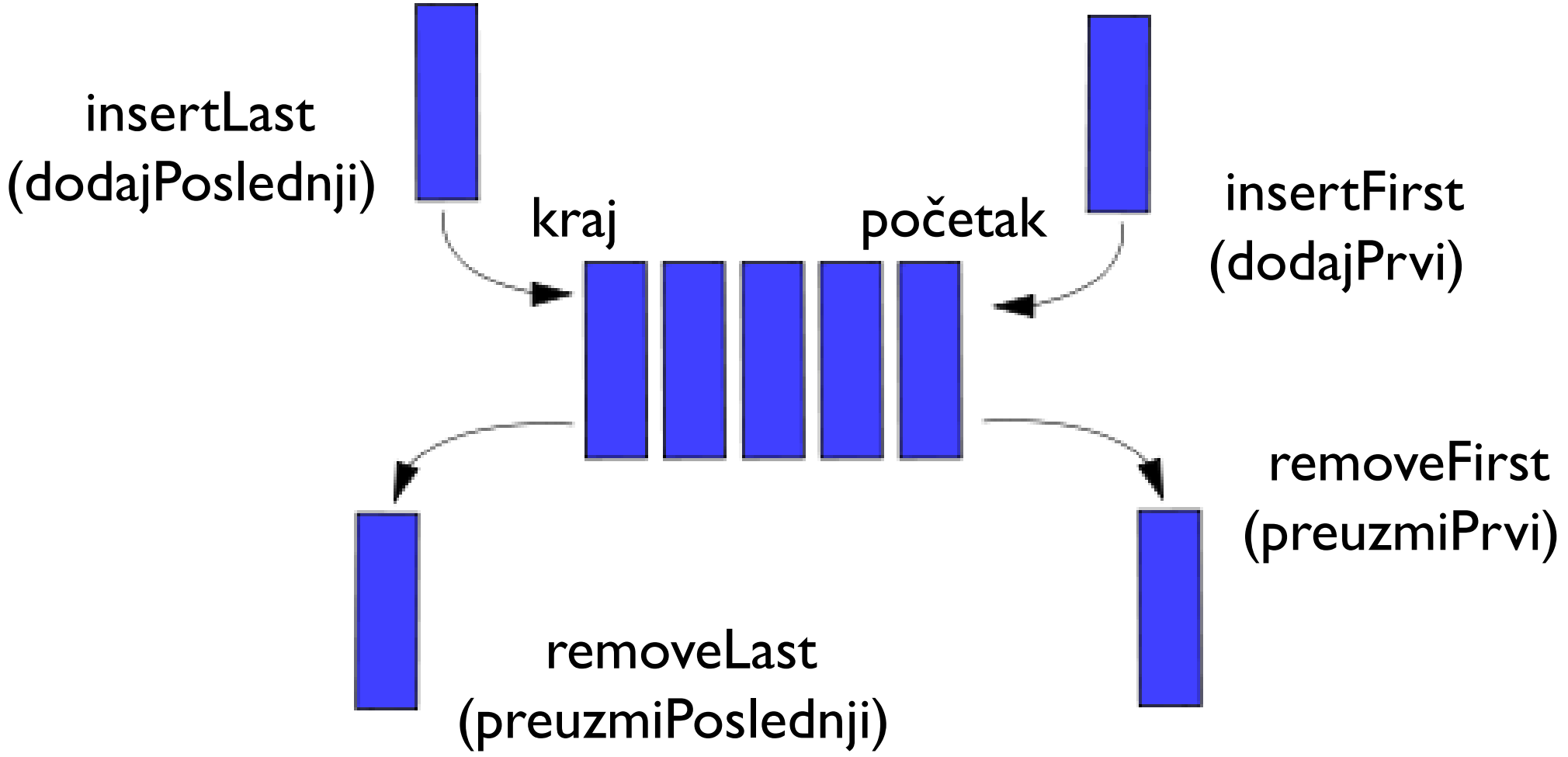
isEmpty (daLiJePrazan) – provera da li je dek prazan

isFull (daLiJePun) – provera da li je dek pun

size (vratiVeličinu) – vraća veličinu deka

Dek

Primer rada sa dekom:



Red sa prioritetom

Red sa prioritetom (engl. *priority queue*) je **proširenje apstraktnog tipa podataka red** kod koga su **elementima pridruženi** odgovarajući **prioriteti**.

Elementi se uklanjaju sa početka reda

Elementi su **uređeni** na osnovu vrednosti polja **prioriteta** tako da se oni **sa najvišim prioritetom** nalaze **na početku reda**

Ako dva elementa imaju **isti prioritet**, onda se "opslužuju" u skladu sa njihovim **redosledom u običnom redu**

Obično se implementiraju pomoću **gomila** (engl. heap)

Operacije kod reda sa prioritetom

insertWithPriority (dodajSaPrioritetom) – dodaj element sa pridruženim prioritetom u red

pullHighestPriorityElement (preuzmiElementSaNajvisimPrioritetom) – preuzmi element sa najvišim prioritetom iz reda

Ostale operacije:

peek (proveri) – provera vrednosti na početku reda bez modifikacije, u ovom konteksu obično se zove **findMax** ili **findMin**

isEmpty (daLiJePrazan) – provera da li je red sa prioriteom prazan

Primene reda sa prioritetom

Upravljanje ograničenim resursima – dodela procesorskog vremena procesima u višeprocesnim operativnim sistemima, propusni opseg kod rutera

Sortiranje pomoću gomile (engl. *heapsort*) – algoritam za sortiranje zasnovan na dodavanju elemenata u red sa prioritetom

Dijkstrin algoritam za pronalaženje najkraćeg puta između para čvorova u grafu

Huffman-ovo kodiranje – metod za kompresiju podataka