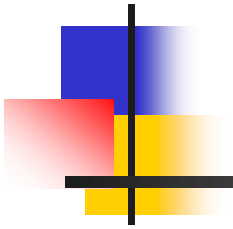
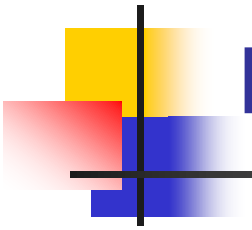


PROGRAMIRANJE II

P-07: Linearne strukture podataka (2. dio)

prof. dr **Dražen Brđanin**
2023/24



P-06: Linearne strukture podataka

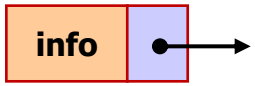
- **Sadržaj predavanja**
  - Dvostruko ulančana lista
  - Stek
  - Red / kružni bafer

Povezane (ulančane) liste

- **Povezana (ulančana) lista** (eng. *linked list*)
  = ulančana implementacija linearne strukture
- **dinamička struktura**
  - dinamička alokacija
  - pristup elementima je indirektan (pokazivači)
- osnovni element: **ČVOR** (eng. *node*)
  - informacioni sadržaj
  - pokazivač(i)

- **Prema načinu povezanosti**
  - jednostruko povezane liste
  - dvostruko povezane liste

Čvor u jednostruko povezanoj listi



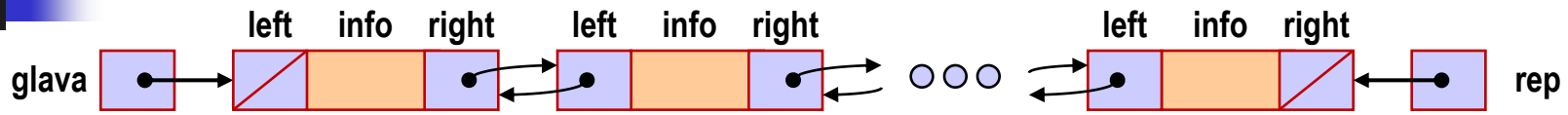
Čvor u dvostruko povezanoj listi


```
typedef struct node {
                        <tip> info;
                        struct node *next;
                      } NODE;
```

```
typedef struct node {
                           <tip> info;
                           struct node *left;
                           struct node *right;
                         } NODE;
```

Samoreferišuća struktura = struktura koja posjeduje pokazivač na istu strukturu

Dvostruko povezana lista



`right(left(p)) = p = left(right(p))`

početak i kraj liste:

- uobičajeno postoje dva spoljašnja pokazivača na listu:
  - **glava** / head (na početak liste)
  - **rep** / tail (na kraj liste)

čvorovi:

- čvorovi se dinamički alociraju/dealociraju
- svaki čvor ima dva pokazivača:
  - **left** (na prethodni čvor)
  - **right** (na sljedeći čvor)
- dvostruki pokazivači omogućavaju **lakše kretanje kroz listu** (**od početka prema kraju i od kraja prema početku**)

formiranje liste:

- na početku je lista prazna

```
NODE *glava = NULL, *rep = NULL;
```

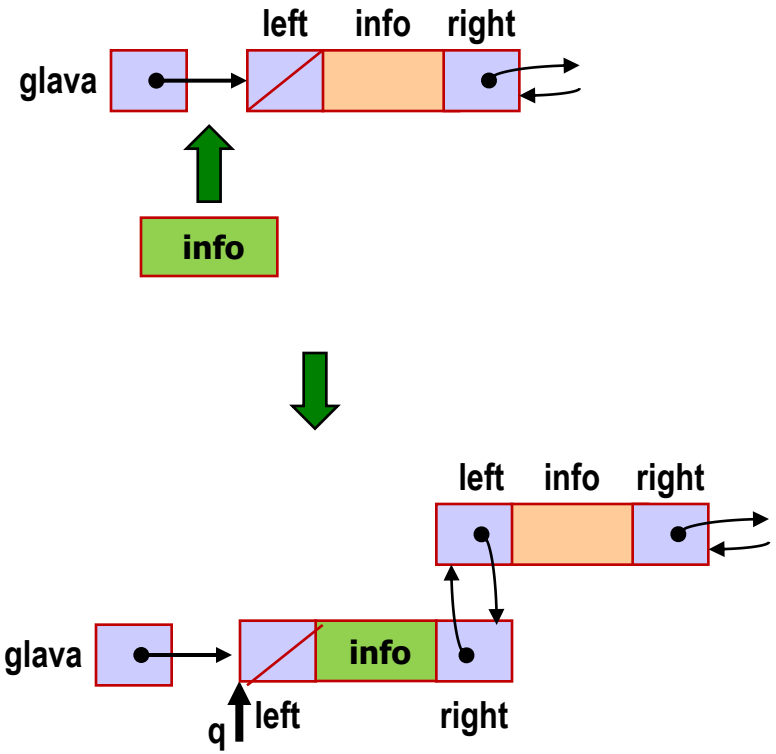
- dodavanje prvog čvora u listu

```
glava = rep = (NODE *) malloc(sizeof(NODE));
glava->info = info;
glava->left = glava->right = NULL;
```

- dodavanje novih čvorova
  - `add_front(&glava, info)`
  - `add_back(&rep, info)`
  - `insert_after(&cvor, info)`
  - `insert_before(&cvor, info)`
- brisanje čvorova
  - `delete(cvor)`
  - `delete_front(&glava)`
  - `delete_back(&rep)`

Dvostruko povezana lista

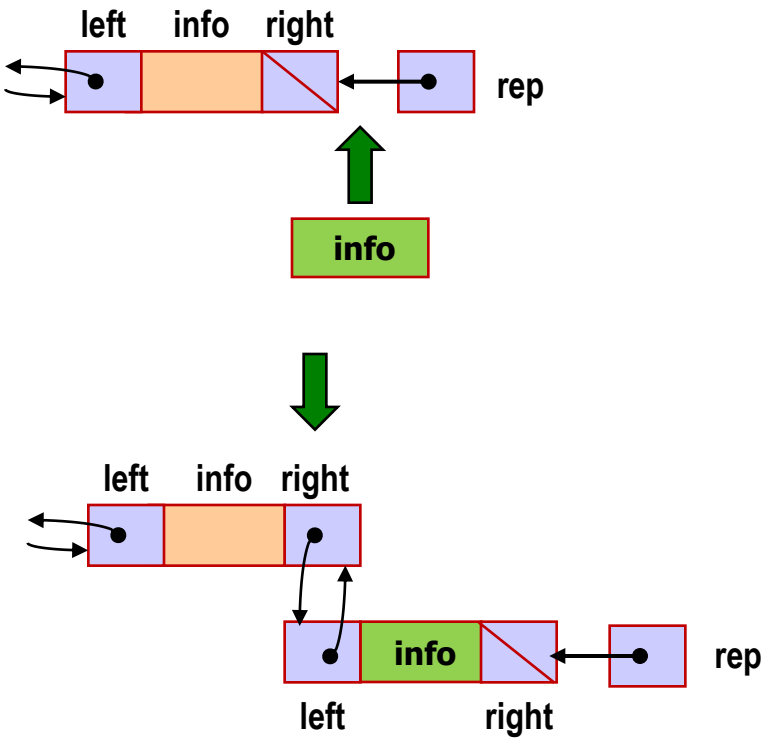
Dodavanje čvora na početak




```
int add_front(NODE **glava, int info)
{
   NODE *q = (NODE *) malloc(sizeof(NODE));
   if (q==NULL) return 0;
   q->info = info;

   if (*glava == NULL)
   {
      q->left = q->right = NULL;
      *glava = q;
   }
   else
   {
      q->right = *glava;
      q->left = NULL;
      (*glava)->left = q;
      *glava = q;
   }
   return 1;
}
```

Dvostruko povezana lista

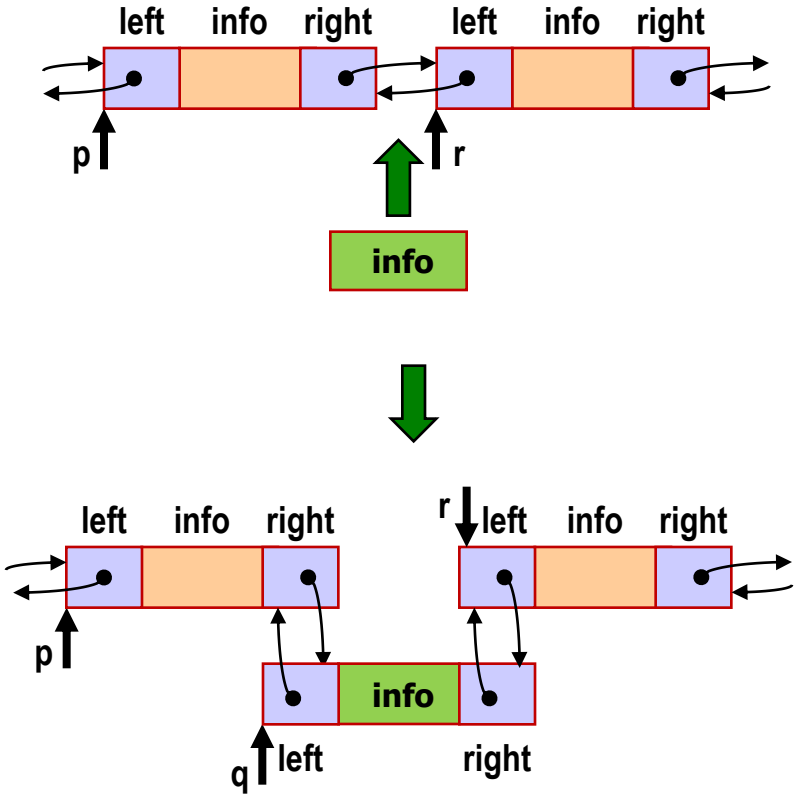
Dodavanje čvora na kraj




```
int add_back(NODE **rep, int info)
{
   NODE *q = (NODE *) malloc(sizeof(NODE));
   if (q==NULL) return 0;
   q->info = info;

   if (*rep == NULL)
   {
      q->left = q->right = NULL;
      *rep = q;
   }
   else
   {
      q->left = *rep;
      q->right = NULL;
      (*rep)->right = q;
      *rep = q;
   }
   return 1;
}
```

Dvostruko povezana lista

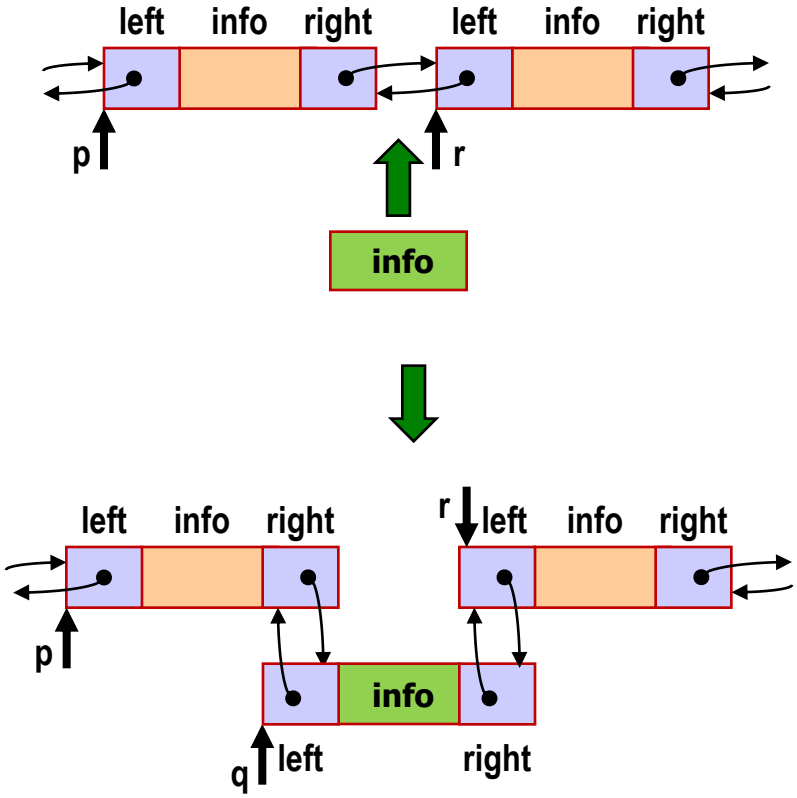
Ubacivanje čvora iza zadatog čvora




```
int insert_after(NODE *p, int info)
{
   NODE *r = p->right;

   if (r == NULL) return 0;

   NODE *q = (NODE *) malloc(sizeof(NODE));
   if (q==NULL) return 0;
   q->info = info;

   q->right = r;  r->left = q;
   q->left = p;   p->right = q;

   return 1;
}
```

Dvostruko povezana lista

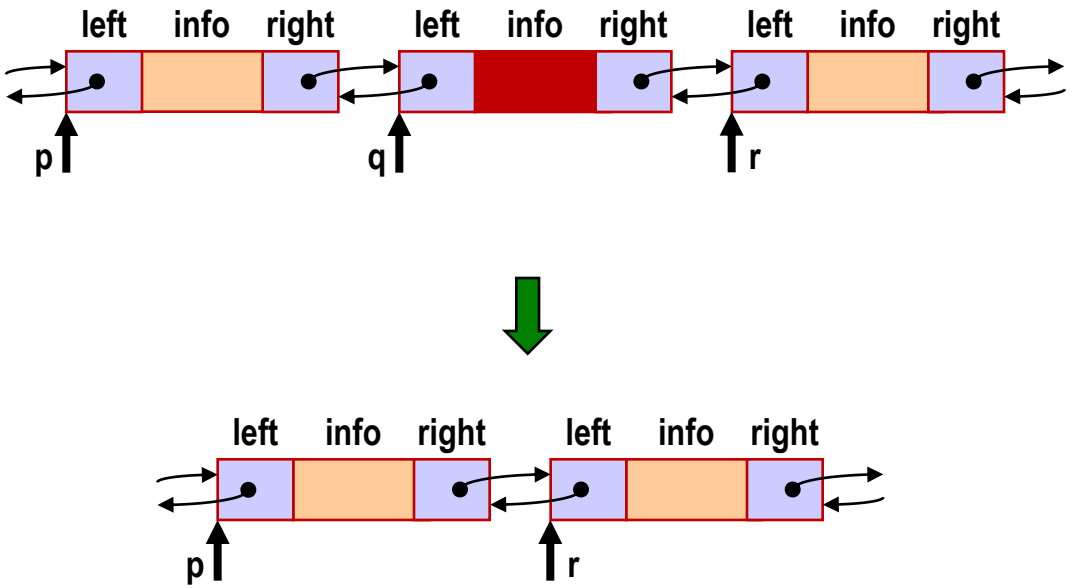
Ubacivanje čvora ispred zadatog čvora




```
int insert_before(NODE *r, int info)
{
   NODE *p = r->left;

   if (p == NULL) return 0;

   NODE *q = (NODE *) malloc(sizeof(NODE));
   if (q==NULL) return 0;
   q->info = info;

   q->right = r;  r->left = q;
   q->left = p;   p->right = q;

   return 1;
}
```

Dvostruko povezana lista

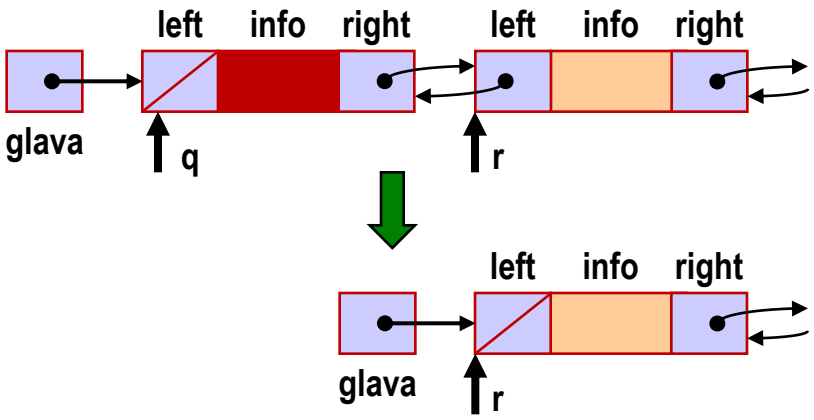
Brisanje zadatog čvora




```
int delete_node(NODE *q)
{
   if (q->left && q->right)
   {
     NODE *p = q->left;
     NODE *r = q->right;
     p->right = r;
     r->left = p;
     free(q);
     return 1;
   }
   return 0;
}
```

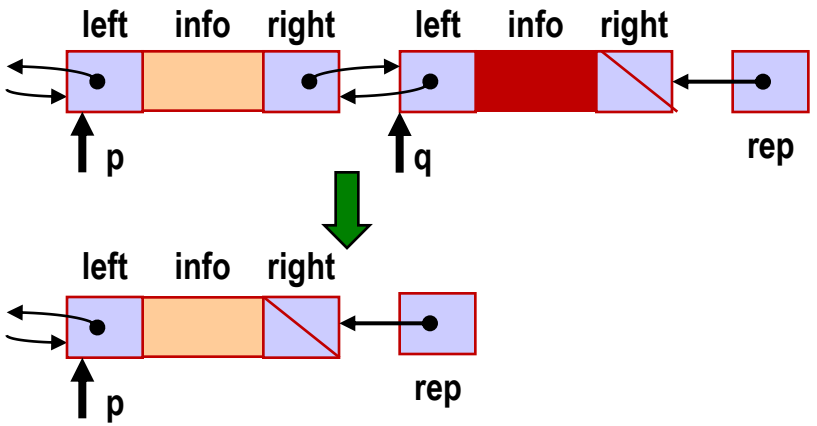
Dvostruko povezana lista

Brisanje početog čvora




```
int delete_front(NODE **glava)
{
   if (*glava==NULL) return 0;

   NODE *q = *glava;
   NODE *r = q->right;
   if (r != NULL) r->left = NULL;
   *glava = r;
   free(q);
   return 1;
}
```

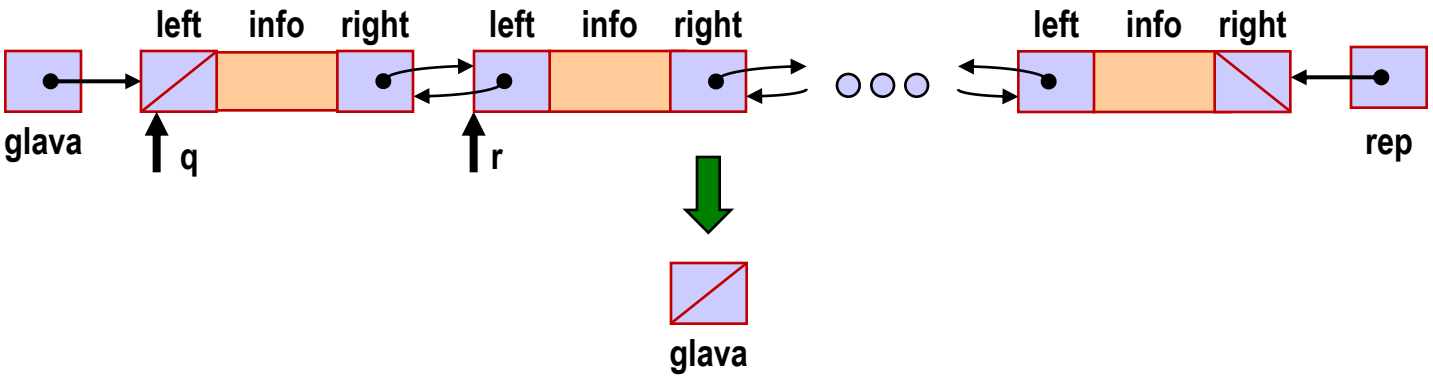
Brisanje posljednjeg čvora




```
int delete_back(NODE **rep)
{
   if (*rep==NULL) return 0;

   NODE *q = *rep;
   NODE *p = q->left;
   if (p != NULL) p->right = NULL;
   *rep = p;
   free(q);
   return 1;
}
```

Dvostruko povezana lista

Brisanje liste

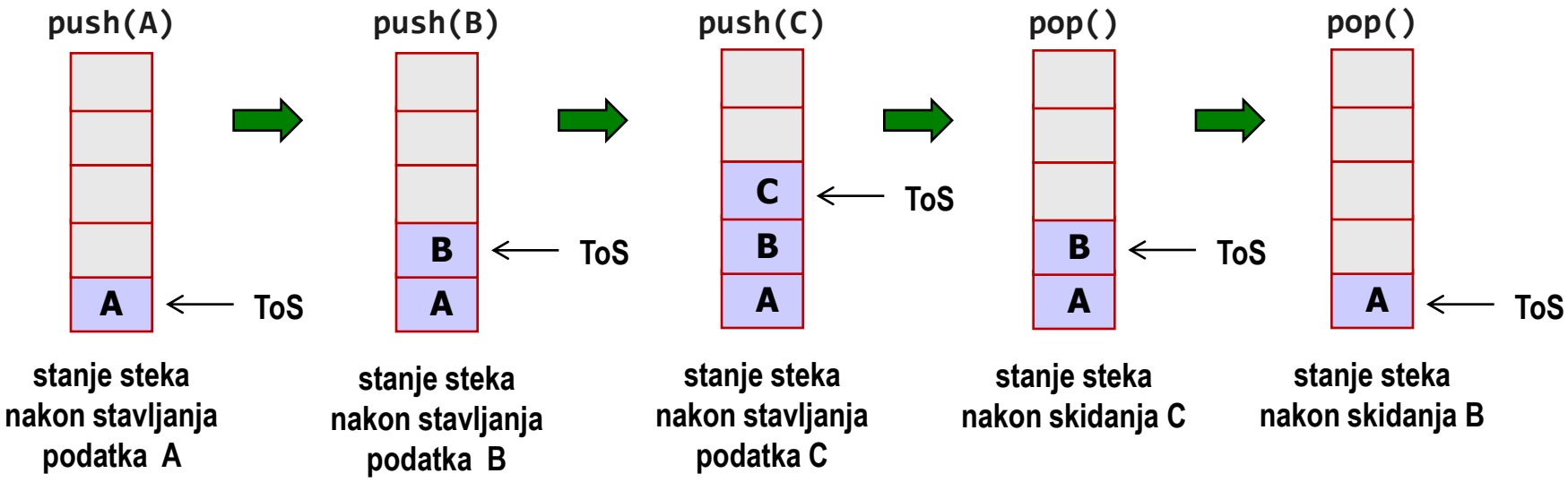



```
int delete_list(NODE **glava)
{
   while (*glava)
   {
      NODE *q = *glava;
      NODE *r = q->right;
      *glava = r;
      free(q);
   }
   return 1;
}
```

```
int delete_list(NODE **glava)
{
   while (delete_front(glava));
   return 1;
}
```

Stek

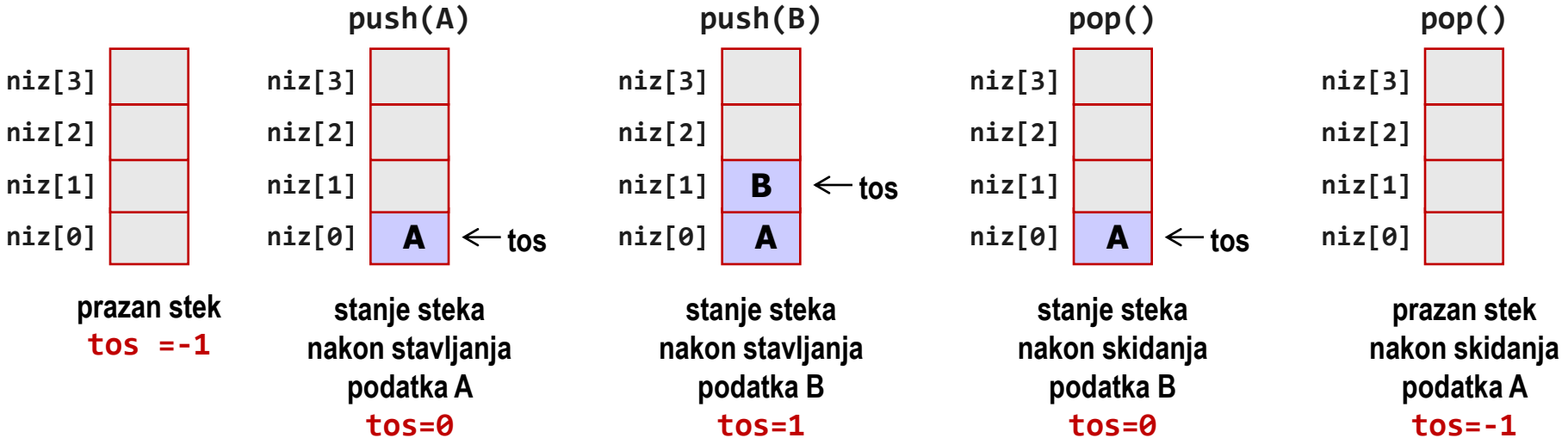
- **STEK** = linearna struktura sa **LIFO** disciplinom pristupa
  - **LIFO** = ***Last In – First Out*** **(Posljednji unutra – prvi napolje)**
  - jedan pristupni kraj = **VRH STEKA** (*Top of Stack* – **ToS**)
  - operacije sa stekom:
    - **push** – stavljanje na stek
    - **pop** – skidanje sa steka



Sekvencijalni stek

- **Sekvencijalna reprezentacija steka**
  - implementacija pomoću niza
  - **kapacitet steka** je ograničen: niz[0] – niz[n-1]
  - **sadržaj steka**: niz[0] – niz[tos]
  - **kontrola pristupa:**
    - na stek ne može da se doda novi podatak ako je **stek pun** (`tos=n-1`)
    - sa steka ne može da se skine podatak ako je **stek prazan** (`tos=-1`)

```
typedef struct stek
        {
          <tip> niz[MAX]; int tos;
        } STEK;
```



Sekvencijalni stek

Operacije na sekvencijalnom steku

```
typedef struct stek
        {
           <tip> niz[MAX]; int tos;
        } STEK;
```

```
/* provjera da li je stek pun */
int isFull(STEK *s)
{
   return s->tos == MAX-1;
}
```

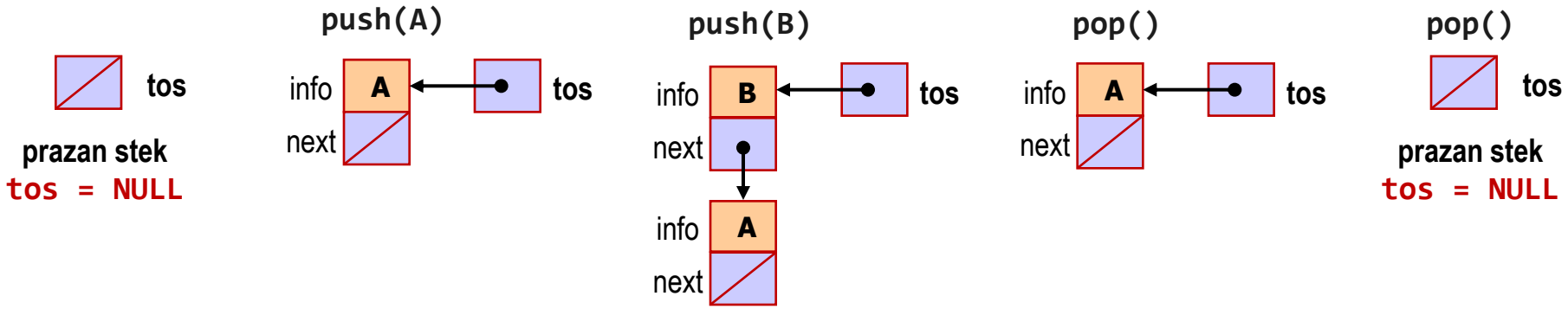
```
/* provjera da li je stek prazan */
int isEmpty(STEK *s)
{
   return s->tos == -1;
}
```

```
/* stavljanje na stek */
int push(STEK *s, <tip> info)
{
   if (isFull(s)) return 0;
   s->niz[++s->tos] = info;
   return 1;
}
```

```
/* skidanje sa steka */
int pop(STEK *s, <tip> *info)
{
   if (isEmpty(s)) return 0;
   *info = s->niz[s->tos--];
   return 1;
}
```

Ulančani stek

- **Ulančana reprezentacija steka**
  - implementacija pomoću (jednostruko) povezane liste
  - **kontrola pristupa:**
    - sa steka ne može da se skine podatak ako je **stek prazan** (`tos=NULL`)




```
/* inicijalizacija */
NODE *tos = NULL;
```

```
/* stavljanje na stek */
push(&tos, info);
```

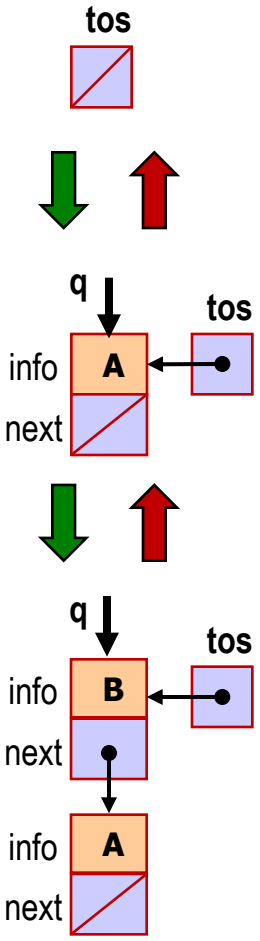
```
/* skidanje sa steka */
pop(&tos, &info);
```

```
typedef struct node { <tip> info; struct node *next; } NODE;
```

Ulančani stek

Operacije na ulančanom steku

```
/* stavljanje na stek */
int push(NODE **tos, <tip> info)
{
  NODE *q=(NODE*) malloc(sizeof(NODE));
  if (q==NULL) return 0;
  q->info = info;
  q->next = *tos;
  *tos = q;
  return 1;
}
```

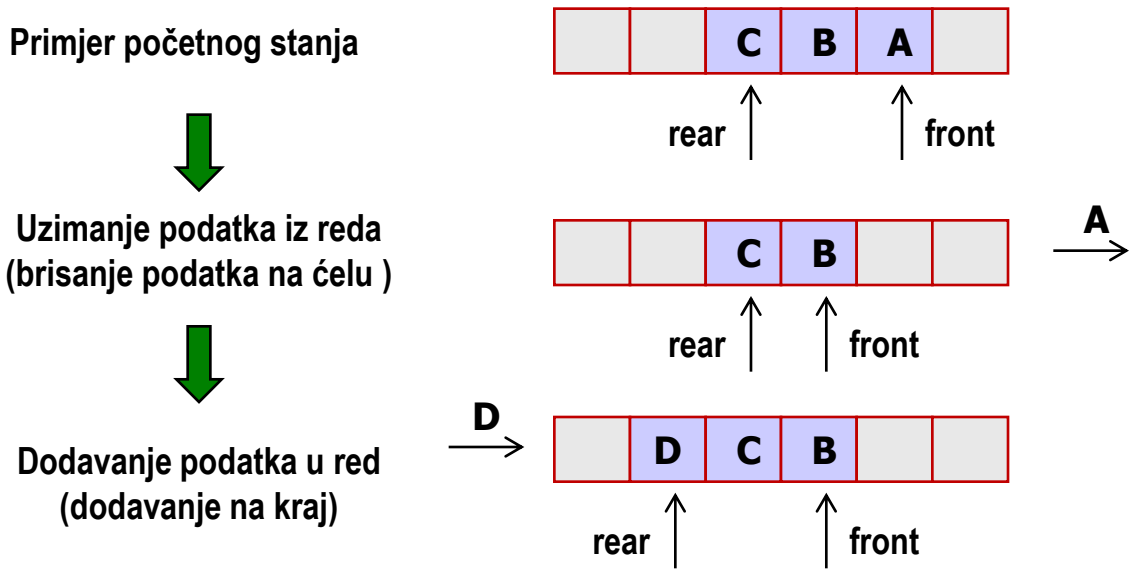



```
/* skidanje sa steka */
int pop(NODE **tos, <tip> *info)
{
  if (*tos == NULL) return 0;
  NODE *q = *tos;
  *info = q->info;
  *tos = q->next;
  free(q);
  return 1;
}
```

```
typedef struct node { <tip> info; struct node *next; } NODE;
```

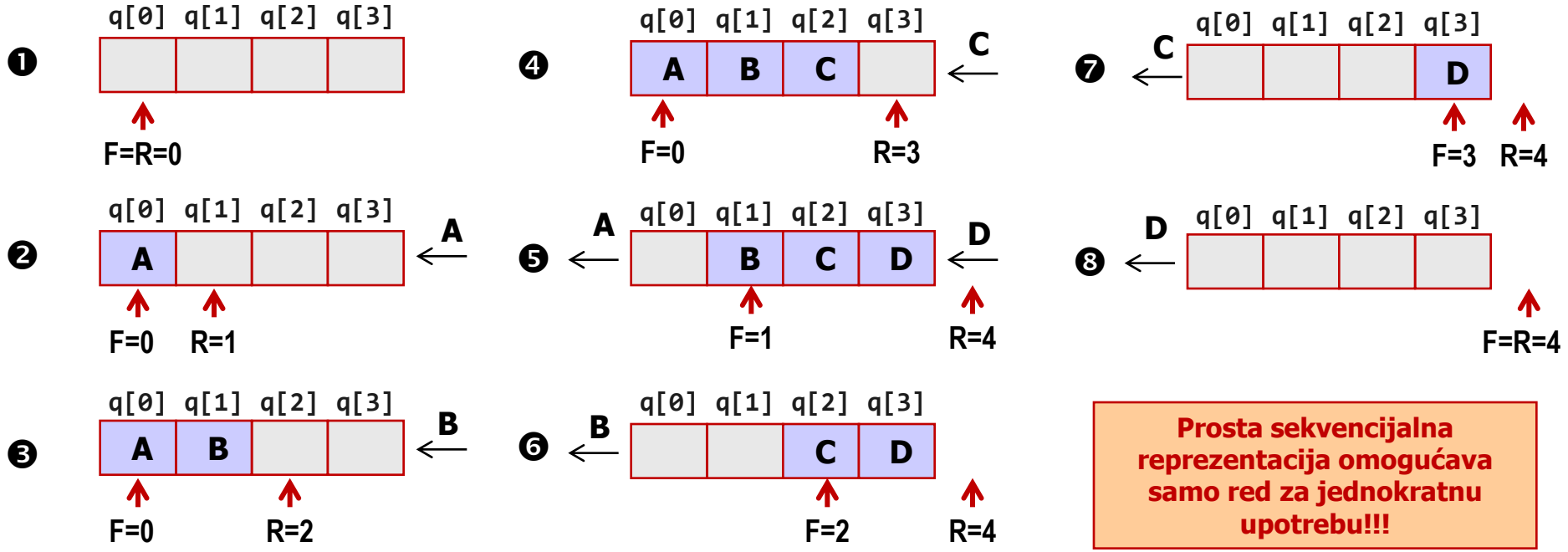
Red

- **RED** (*Queue*) = linearna struktura sa **FIFO** disciplinom pristupa
  - **FIFO = *First In – First Out*** **(Prvi unutra – prvi napolje)**
  - dva pristupna kraja: **čelo** (*front*) i **začelje** (*rear/back*)
  - operacije sa redom:
    - **insert/put** – dodavanje na začelje (na kraj reda)
    - **delete/get** – brisanje/uzimanje iz reda (brisanje na čelu)



Sekvencijalni red

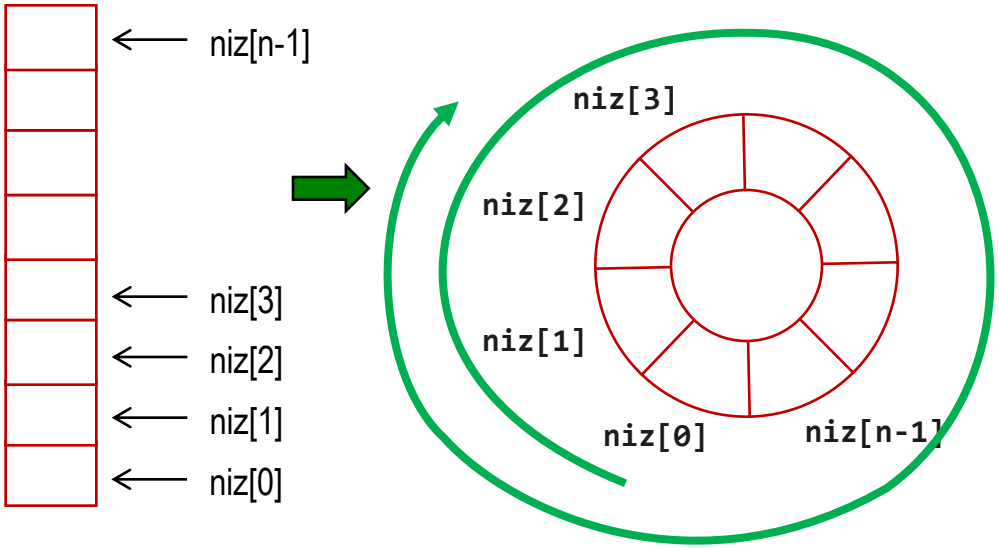
- **Sekvencijalna reprezentacija reda**
  - implementacija pomoću niza
  - **kapacitet reda** je veoma ograničen: niz[0] – niz[n-1]
  - **kontrola pristupa:**
    - novi podatak ne može da se doda na kraj reda, ako začelje pokazuje izvan niza (rear=n)
    - iz reda podatak može da se uzme ako je front<rear



Prosta sekvencijalna reprezentacija omogućava samo red za jednokratnu upotrebu!!!

Sekvencijalni red/Kružni bafer

- **Kružni (cirkularni) bafer** = kružna sekvencijalna reprezentacija reda
  - elementi niz[0] i niz[n-1] su logički susjedne lokacije



- **kapacitet kružnog bafera**

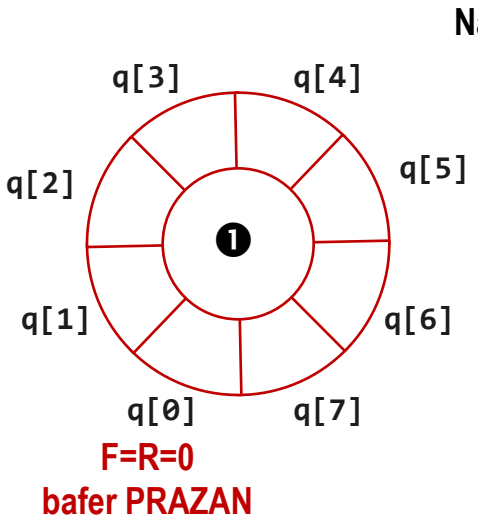
  niz[0] – niz[n-1]

- **kontrola pristupa:**
  - početak reda – front (f)
    - pozicija sa koje se uzima podatak
  - kraj reda – rear (r)
    - pozicija na koju se dodaje podatak

```
typedef struct kruzniBafer
        {
           <tip> niz[MAX];
           int f, r;
        } KB;
```

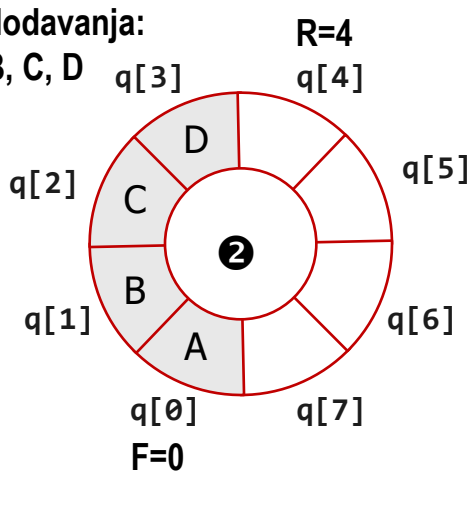
Sekvencijalni red/Kružni bafer

Primjer dodavanja u KB:



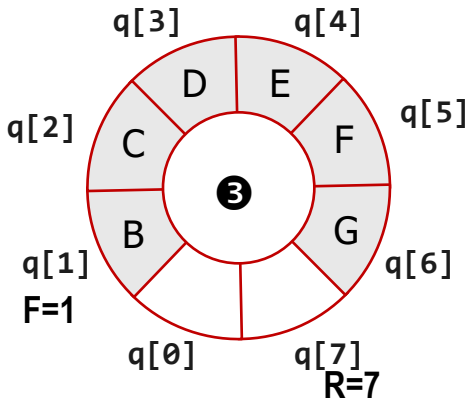
F=R=0
bafer PRAZAN

Nakon dodavanja:
A, B, C, D

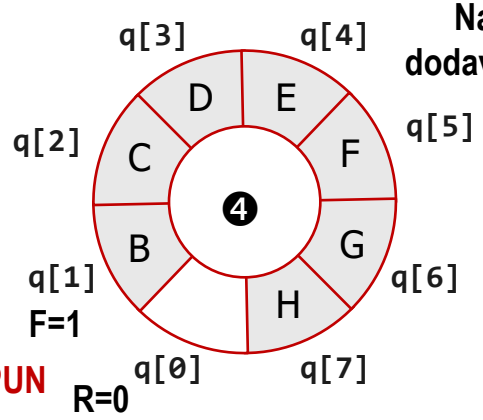


F=0

Nakon dodavanja: E, F, G
Nakon brisanja: A



Nakon dodavanja: H



bafer PUN

Bafer je PUN kad se na kraj ne može dodati novi element
Ako bi se dodao novi element, dobili bismo
F==R => BAFER PRAZAN

Bafer je pun kad F i R pokazuju na susjedne elemente
(R+1)%MAX == F => BAFER PUN

Sekvencijalni red/Kružni bafer

Dva scenarija ako je bafer pun

nije dozvoljeno prepisivanje (prenos bez gubitka informacija)

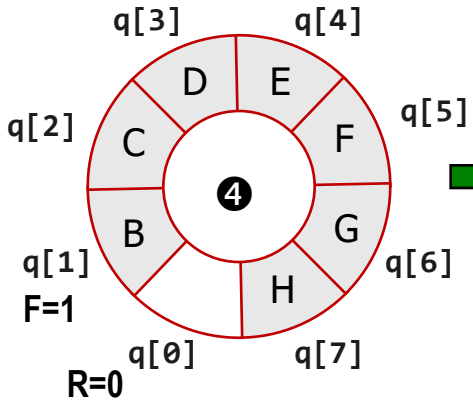
npr. štampanje dokumenta

•novi podatak može da se upiše u bafer tek kad se oslobodi mjesto u baferu (nakon što se F pomjeri)

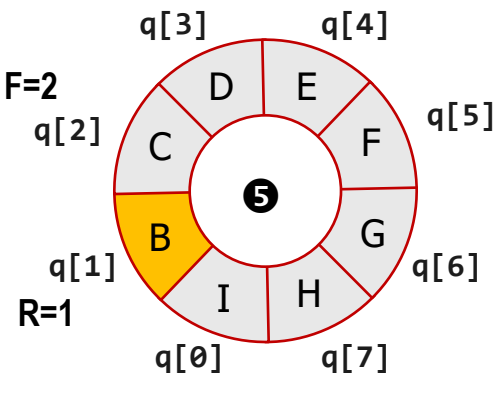
dozvoljeno prepisivanje (prenos sa gubitkom informacija)

npr. *streaming*

•novi podatak upisuje se na kraj reda, a početak se pomjera za jedno mjesto (gubi se najmanje značajan podatak u baferu)



bafer PUN



Nakon dodavanja: I

bafer PUN

podatak q[1] je izgubljen

Bafer PUN, ali na začelje dodat novi podatak (I), a čelo pomjereno za jedno mjesto (česta realizacija u realnom vremenu da se ne gube posljednji značajni podaci).

Sekvencijalni red/Kružni bafer

Operacije na cirkularnom sekvencijalnom redu

```
typedef struct kruzniBafer
        {
           <tip> niz[MAX];
           int f, r;
        } RED;
```

```
/* provjera da li je red pun */
int isFull(RED *kb)
{
   return (kb->r+1) % MAX == kb->f;
}
```

```
/* provjera da li je red prazan */
int isEmpty(RED *kb)
{
   return kb->f == kb->r;
}
```

```
/* brisanje iz reda */
int delete(RED *kb, <tip> *info)
{
   if (isEmpty(kb)) return 0;
   *info = kb->niz[kb->f];
   kb->f = (kb->f + 1) % MAX;
   return 1;
}
```

Sekvencijalni red/Kružni bafer

Dodavanje u kružni bafer

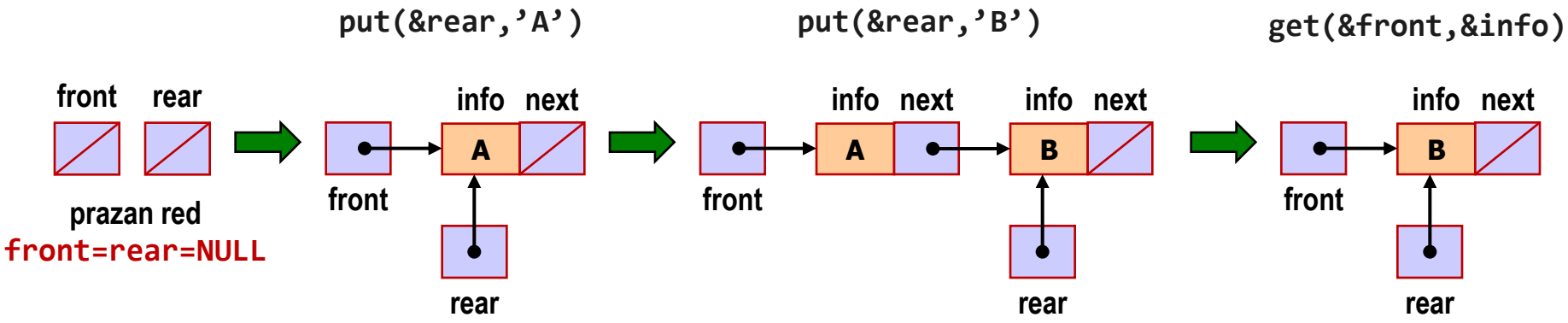
```
/* dodavanje u red bez prepisivanja */
int insertNoRewrite(RED *kb, <tip> info)
{
   if (isFull(kb)) return 0;
   kb->niz[kb->r] = info;
   kb->r = (kb->r + 1) % MAX;
   return 1;
}
```

```
/* dodavanje u red sa prepisivanjem */
int insertRewrite(RED *kb, <tip> info)
{
   if (isFull(kb))
      kb->f = (kb->f + 1) % MAX;
   kb->niz[kb->r] = info;
   kb->r = (kb->r + 1) % MAX;
   return 1;
}
```

```
/* dodavanje u red */
int insert(RED *kb, <tip> info, int prepis)
{
   if (!isFull(kb) || prepis)
   {
      kb->niz[kb->r] = info;
      kb->r = (kb->r + 1) % MAX;
      if (isEmpty(kb))
         kb->f = (kb->f + 1) % MAX;
      return 1;
   }
   return 0;
}
```

Ulančani red

- **Ulančana reprezentacija reda**
  - implementacija pomoću (jednostruko) povezane liste
  - **kontrola pristupa:**
    - podatak se dodaje na kraj liste/reda (rep/rear), a uzima s početka liste/reda (glava/front)




```
/* inicijalizacija */
NODE *front = NULL;
NODE *rear = NULL;
```

```
/* stavljanje na kraj reda */
put(&rear, info);
```

```
/* uzimanje s pocetka reda */
get(&front, &info);
```

```
typedef struct node { <tip> info; struct node *next; } NODE;
```

Ulančani red

Operacije na ulančanom redu

```
/* stavljanje na kraj reda */
int put(NODE **front, NODE **rear, <tip> d)
{
  NODE *q = (NODE*)calloc(1,sizeof(NODE));
  if (q == NULL) return 0;
  q->info = d;
  if (*rear == NULL)
     *front = *rear = q;
  else
  {
     (*rear)->next = q;
     *rear = q;
  }
  return 1;
}
```

```
/* uzimanje s pocetka reda */
int get(NODE **front, NODE **rear, <tip> *d)
{
  if (*front == NULL) return 0;
  NODE *q = *front;
  *d = q->info;
  *front = q->next;
  if (*front == NULL) *rear = NULL;
  free(q);
  return 1;
}
```