Liste

Općenita o listama

- Lista se sastoji od podataka istog tipa (a_1, a_2, … , a_n)
  - Primjer: riječ je lista znakova
  - Ti podatci se nazivaju elementi
  - Lista nije isto što i polje
    - Nove elemente lako se ubaci u listu (a postojeće izbaci s nje)
    - Broj elemenata nije fiksiran na početku
  - n je (trenutna) duljina liste
  - Ako je n=0, imamo praznu listu
  - Identitet elementa određen je pozicijom u listi, a ne njegovom vrijednošću
  - Važno je svojstvo liste da postoji linearni uređaj - elementi su poredani tako da se element a_i nalazi ispred a_{i+1}, a iza a_{i-1}

- Primjeri za liste:
  - redak teksta je lista riječi (ili znakova)
  - tekst je lista redaka
  - Polinom $P(x) = a_1x^e_1 + a_2x^e_2 + \ldots + a_nx^e_n$ je lista oblika $((a_1,e_1), (a_2,e_2), \ldots, (a_n,e_n))$

- Za potpuno definiranje ATP (apstraktnog tipa podataka) lista potrebno je definirati i operacije na listama

Apstraktni tip podataka LIST

- elementtype … bilo koji tip (int, float, double, char, …)
- LIST … podatak tipa LIST je konačan niz podataka tipa elementtype
- position … podatak ovog tipa identificira element u listi; za listu od n elemenata definirano je n+1 pozicija (pozicija kraja liste neposredno iza n-tog elementa)
- END(L) … funkcija koja vraća poziciju na kraju liste L
- MAKE_NULL(&L) … funkcija isprazni listu L i vraća poziciju END(L)
- INSERT(x,p,&L) … funkcija ubacuje podatak x na poziciju p u listi L
  - elementi od p-tog do n-tog se pomiču za jedno mjesto
  - ako ne postoji pozicija p, rezultat je nedefiniran
- DELETE(p,&L) … funkcija izbacuje element na poziciji p iz liste L
  - rezultat nedefiniran ako u L nema pozicije p, ili je p jednak END(L)
- FIRST(L) …funkcija vraća prvu poziciju u listi L
  - za praznu vraća END(L)
- NEXT(p,L), PREVIOUS(p,L) … funkcije koje vraćaju poziciju iza/ispred p u L; nedefinirane ako p ne postoji u L, NEXT() nedefinirana za p==END(L), PREVIOUS() nedefinirana za p==FIRST(L)
- RETRIEVE(p,L) … funkcija vraća element na poziciji p u L; nedefinirana ako p ne postoji ili za p==END(L)

- Postoje dva osnovna pristupa realizaciji liste:

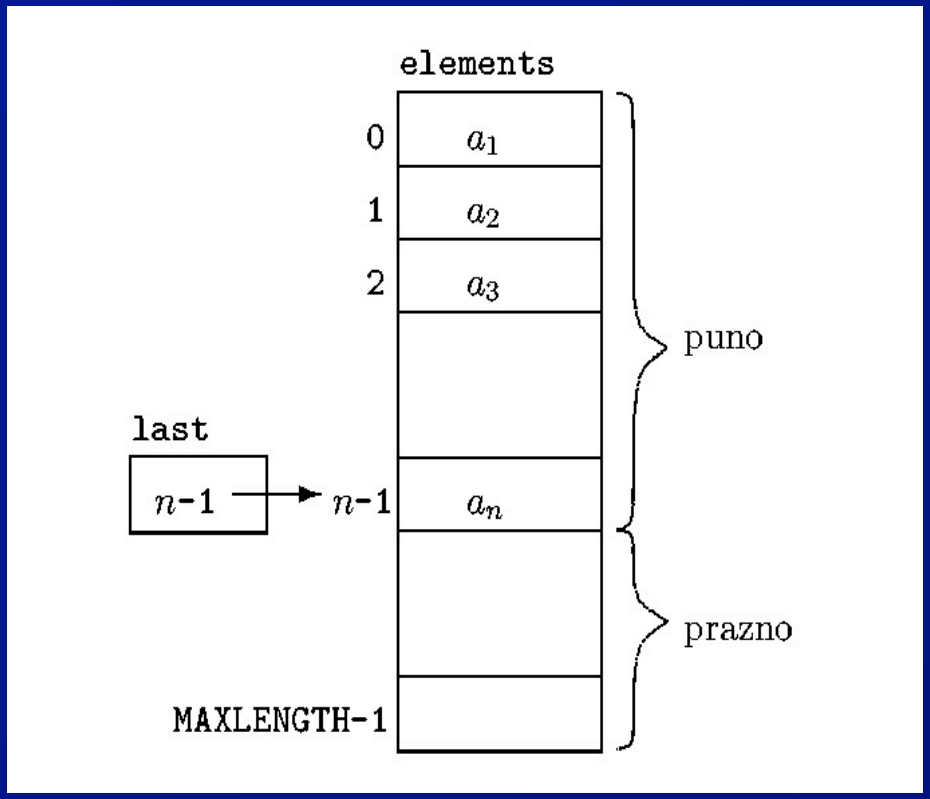
1) kada se logički redoslijed elemenata u listi poklapa s fizičkim redoslijedom u memoriji (Statička struktura podataka)

2) kada se logički i fizički redoslijed ne poklapaju, pa se mora eksplicitno zapisati veza među elementima (Dinamička struktura podataka)

- Oba pristupa dozvoljavaju razne varijante

Implementacija liste pomoću polja

- Elementi liste spremljeni u uzastopnim ćelijama jednog polja
- Potreban je kursor koji pokazuje gdje se zadnji element liste nalazi u polju (n-1)
- Prednosti: lako čitanje i-tog elementa, lako dodavanje i uklanjanje elementa s kraja liste, jednostavan programski kod
- Mane: ubacivanje i izbacivanje u unutrašnjosti liste zahtjeva pomicanje dijela podataka; duljina liste je ograničena (MAXLENGTH)



C kod za implementaciju liste pomoću polja

```
#define MAXLENGTH …

 typedef struct {
       int last;
       elementtype elements[MAXLENGTH];
 } LIST;

 typedef int position;                    // Što radimo ovdje?

 position END(LIST L) {                   // A ovdje?
       return (L.last + 1);
 }

 position MAKE_NULL(LIST *L_ptr) {        // Zašto "*"?
       L_ptr->last = -1;                  // Zašto "->"?
       return 0;
 }
```

```
void INSERT(elementtype x, position p, LIST *L_ptr) {
     position q;
     if (L_ptr->last >= MAXLENGTH -1)
          error("Lista je puna");
     else
          if (p > L_ptr->last+1) || (p < 0))
               error("Pozicija ne postoji");
          else {
               for (q = L_ptr->last ; q >= p ; q--)
               L_ptr->elements[q+1] = L_ptr->elements[q];
               L_ptr->last++;
               L_ptr->elements[p] = x;
          }
}

position FIRST(LIST L){
     return 0;
}
```

```
void DELETE(position p, LIST *L_ptr) {
     position q;
     if ( (p > L_ptr->last) || (p < 0))
         error("Pozicija ne postoji");
     else {
         L_ptr->last--;
         for (q = p ; q <= L_ptr->last; q++)
              L_ptr->elements[q] = L_ptr->elements[q+1];
     }
}

position NEXT(position p, LIST *L_ptr) {
     return ++p;              // Može li tu biti p++? (Zašto ne?)
}

position PREVIOUS(position p, LIST *L_ptr) {
     return --p;
}
```

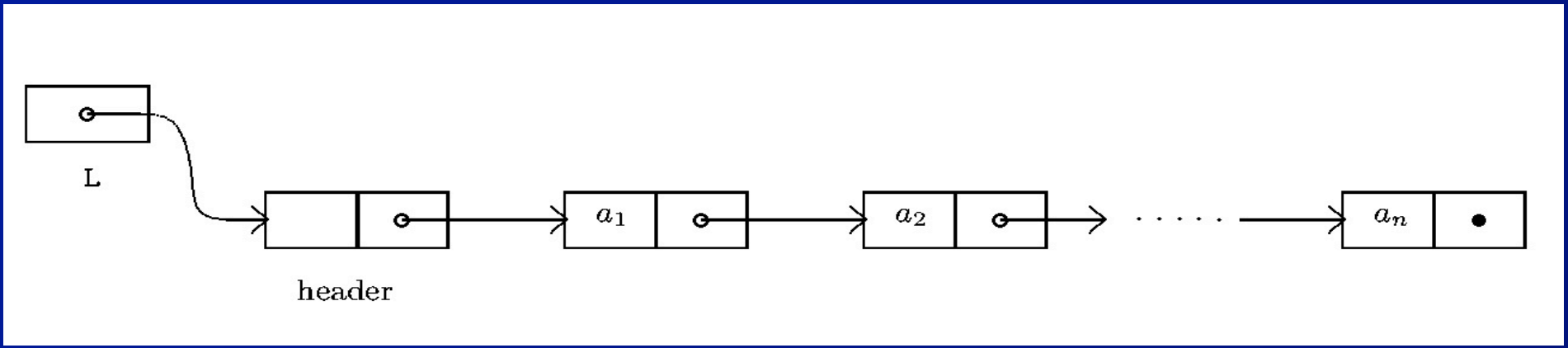
```
elementtype RETRIEVE(position p, LIST *L_ptr) {
      if (p >= 0 && p <= L_ptr->last)
            return L_ptr->elements[p];
      else
            error("Nepostojeca pozicija");
      return 0;
}
```

Broj potrebnih operacija za funkcije INSERT() i DELETE() je u najgorem slučaju jednak broju elemenata liste O(n), a za ostale funkcije je uvijek jedan korak O(1).

Ovakva implementacija se koristi kada je moguće unaprijed zadati gornju ogradu za duljinu liste i kada nema mnogo ubacivanja/izbacivanja u unutrašnjost liste.

Implementacija liste pomoću pokazivača

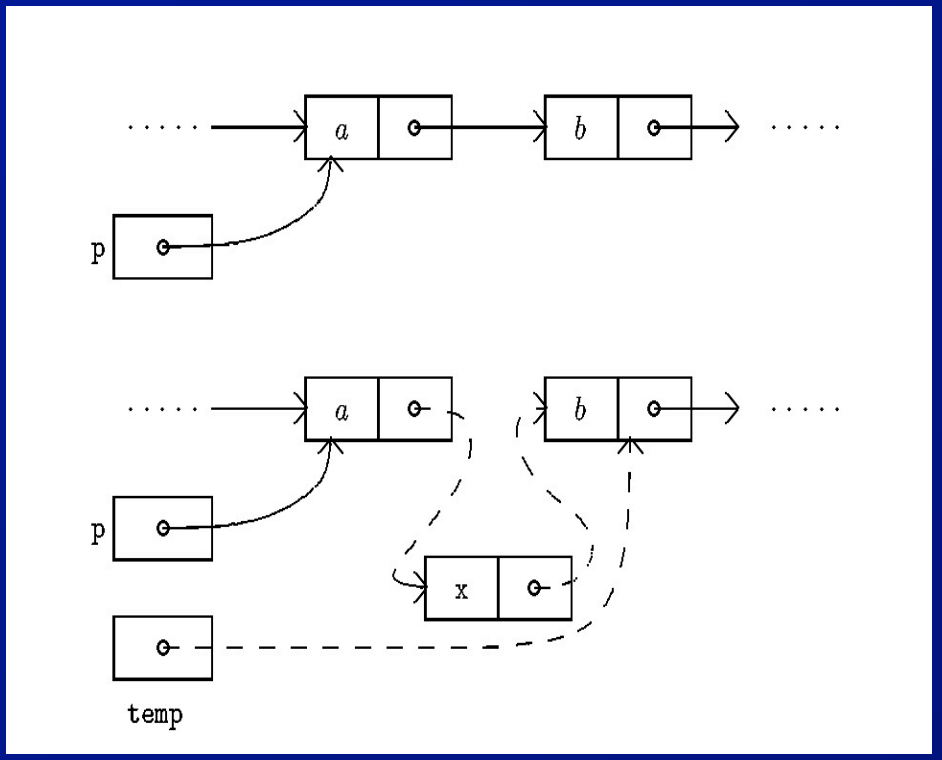
- Lista se prikazuje nizom ćelija, svaka ćelija sadrži jedan element liste i pokazivač na istu takvu ćeliju koja sadrži idući element liste
- Polazna ćelija - glava (header) označava početak liste i ne sadrži element
- Ovakva struktura se obično zove **vezana lista**
- Prednosti: lagano ubacivanje i izbacivanje elemenata po cijeloj duljini liste
- Mane: da bi se pročitao i-ti element treba pročitati sve elemente prije njega, teže je odrediti kraj liste i prethodni element (jer pokazivač pokazuje na sljedeći, a ne na prethodni)
- Lista se poistovjećuje s pokazivačem na glavu; pozicija elementa a_i je pokazivač na ćeliju koja sadrži pokazivač na a_i (pozicija od a_1 je pokazivač na glavu), pozicija END(L) je pokazivač na zadnju ćeliju



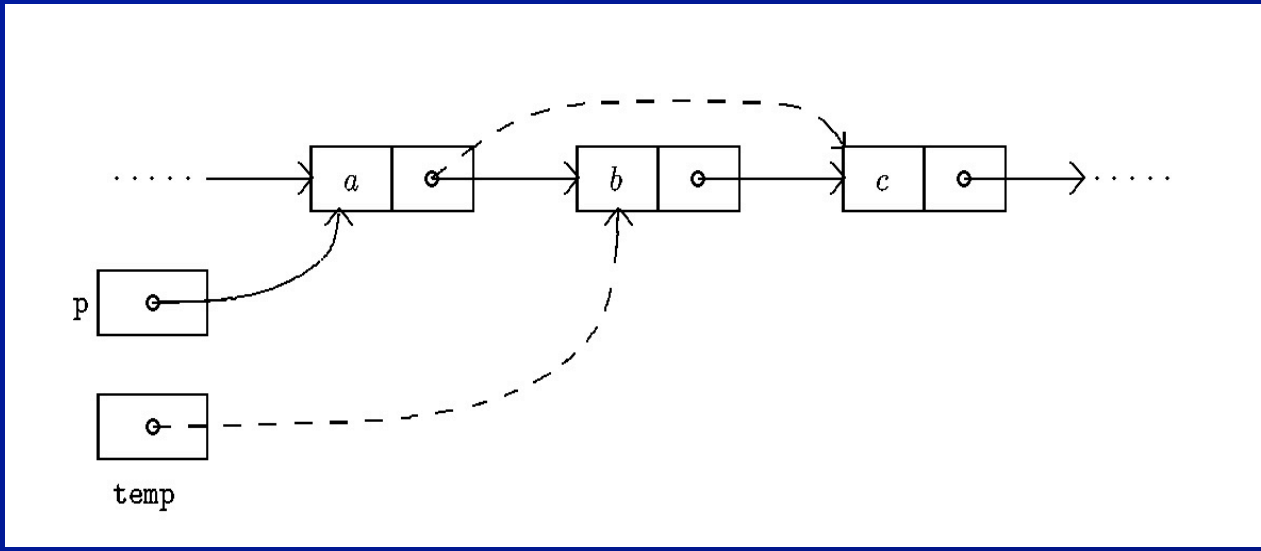
C kod za implementaciju liste pomoću pokazivača

```
typedef struct cell_tag {
     elementtype element;
     struct cell_tag *next;
}  celltype;

typedef celltype *LIST;

typedef celltype *position;

position END(LIST L) {
   position q;
   q = L;
   while (q->next != NULL)
      q = q->next;
   return q;
}
```

```
position MAKE_NULL(LIST *Lptr) {
    *Lptr = (celltype*) malloc(sizeof(celltype));
    (*Lptr)->next = NULL ;
    return (*Lptr);
}

void INSERT(elementtype x, position p){
   position temp;
   temp = p->next;
   p->next = (celltype*) malloc(sizeof(celltype));
   p->next->element = x;
   p->next->next = temp;
}

position FIRST(LIST *Lptr) {
   return *Lptr;
}
```



```
void DELETE(position p) {
    position temp;
    temp = p->next;
    p->next = p->next->next;
    free(temp);
}

position NEXT(position p) {
   return p->next;
}

elementtype RETRIEVE(position p) {
    return p->next->element;
}

position PREVIOUS(position p, LIST L) {
    position q = L;
    while(q->next != p)
       q = q->next;
    return q;
}
```

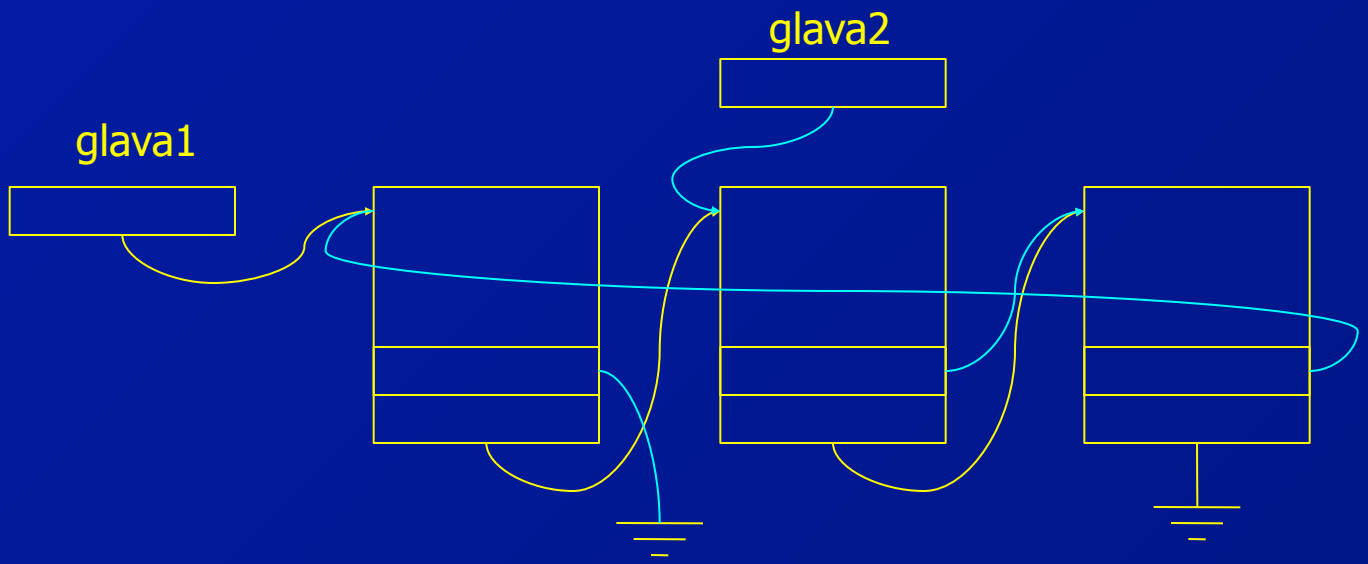


Broj koraka za izvršavanje END() i PREVIOUS() je jednak O(n), a za sve ostale funkcije je jedan korak O(1).

Ova implementacija se koristi kada ima mnogo ubacivanja/izbacivanja u listu i kada se duljina liste može jako mijenjati.

Liste s više ključeva

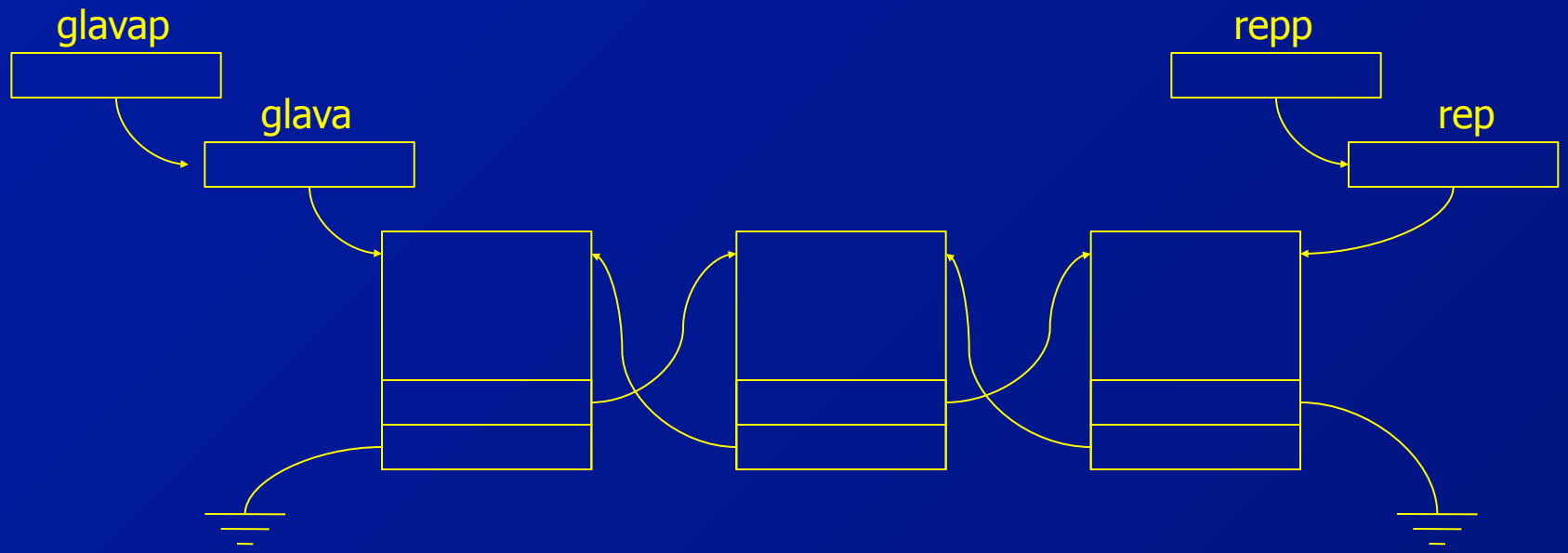
- Složeniji slučaj liste: moguća je struktura podataka koja uz element sastavljen od više ćelija istog ili različitog tipa podataka sadrži i više pokazivača na idući element, pa se elementi mogu sortirati po više ključeva



- Primjer: element liste se sastoji od polja znakova i cijelog broja (recimo ime i prezime osobe i njen matični broj) i uz svaki element idu 2 pokazivača, čime se omogućuje sortiranje liste po matičnom broju i prezimenu, a da su podaci zapisani samo jednom

Dvostruko povezana lista

- Radi bržeg traženja u oba smjera kretanja po listi, ona može biti dvostruko povezana. Svaki čvor osim elementa s podacima, sadrži pokazivač na sljedeći čvor i pokazivač na prethodni čvor.
- Lista ima glavu i rep.



Stog

- Stog je specijalna vrsta liste u kojoj se sva ubacivanja i izbacivanja elemenata obavljaju na jednom kraju koji se zove vrh (kao hrpa tanjura - uzimamo ili dodajemo po jedan tanjur na vrh)
- Struktura podataka kod koje se posljednji pohranjeni podatak prvi uzima u obradu (zadnji-unutra-prvi-van ili FIFO)
- Primjeri stoga:
  - hrpa tanjura, hrpa knjiga
  - glavni program poziva potprograme koji pozivaju druge potprograme: potrebno je da potprogrami pamte adresu povratka u nadređenu proceduru (sjećate se rekurzije)
  - računanje na kalkulatoru: postfix notacija: (a+b)*c-d = ab+c*d- , računanje se provodi čitanjem izraza s lijeva, operandi se stavljaju na stog; kada se pročita operator, sa stoga se skida toliko operanada koliko taj operator traži, obavi se operacija i rezultat se vraća na stog

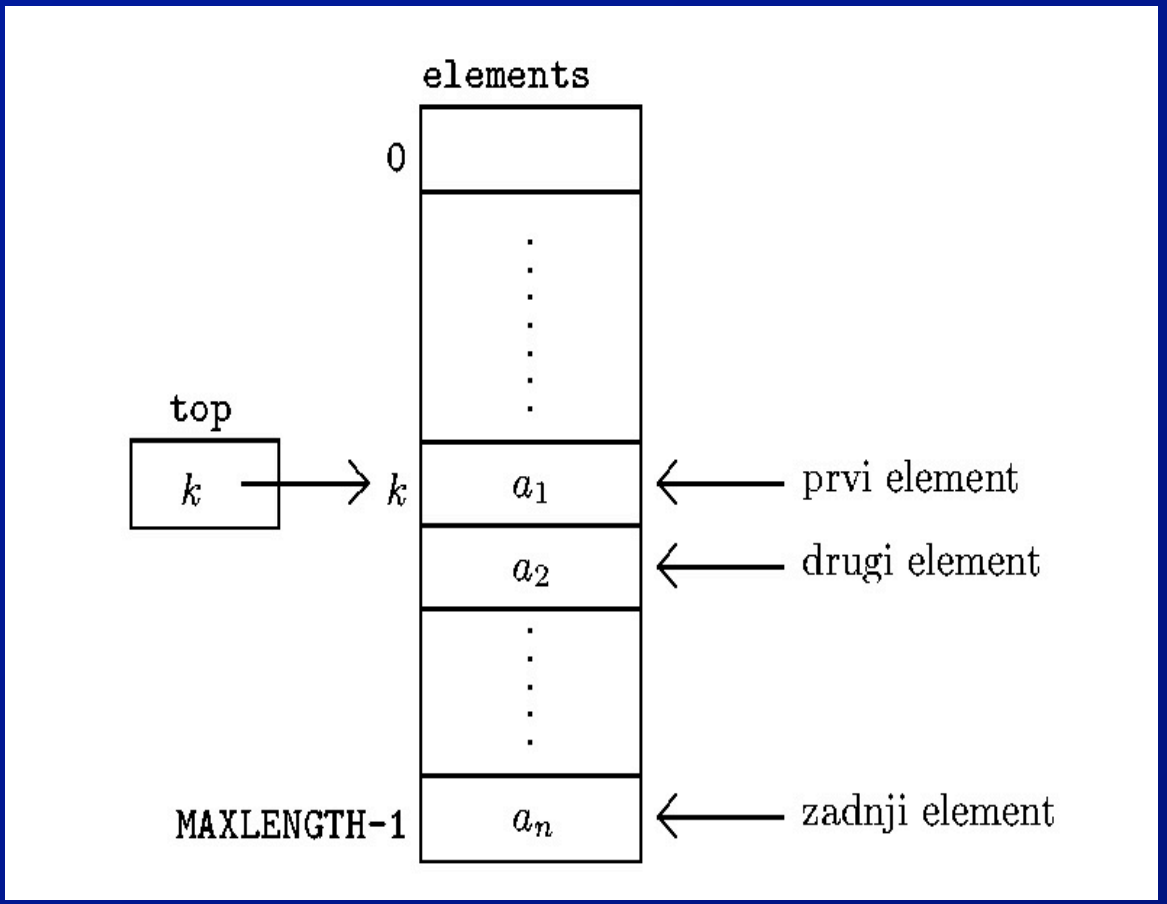
Apstraktni tip podataka STACK (STOG)

- elementtype … bilo koji tip
- STACK … podatak tipa STACK je konačni niz podataka tipa elementtype
- MAKE_NULL(&S) … funkcija prazni stog S
- EMPTY(S) … funkcija koja provjerava je li stog S prazan (vraća true ako je)
- PUSH(x, &S) … funkcija ubacuje element x na vrh stoga S; u terminologiji lista, to odgovara funkciji INSERT(x, FIRST(S), &S)
- POP(&S) … funkcija uklanja element s vrha stoga S; ekvivalentno funkciji za liste DELETE(FIRST(S), &S)
- TOP(S) … funkcija vraća vrijednost elementa koji je na vrhu stoga S (stog ostaje nepromijenjen); ekvivalentno RETRIEVE(FIRST(S),S)

- svaka implementacija liste može se iskoristiti za implementaciju stoga
- operacije na stogu su jednostavnije nego operacije s nekom općenitom listom
- postoje i implementacije stoga pomoću polja i pomoću pokazivača

Implementacija stoga pomoću polja

- Ova implementacija se zasniva na strukturi podataka opisanu za općenitu listu s jednom promjenom da stog smještamo u donji dio polja, a ne u gornji kao općenitu listu
- Tako prilikom ubacivanja/izbacivanja ne treba prepisivati ostale elemente
- Dakle, stog raste prema gore, tj. manjim indeksima polja



C kod za implementaciju stoga pomoću polja

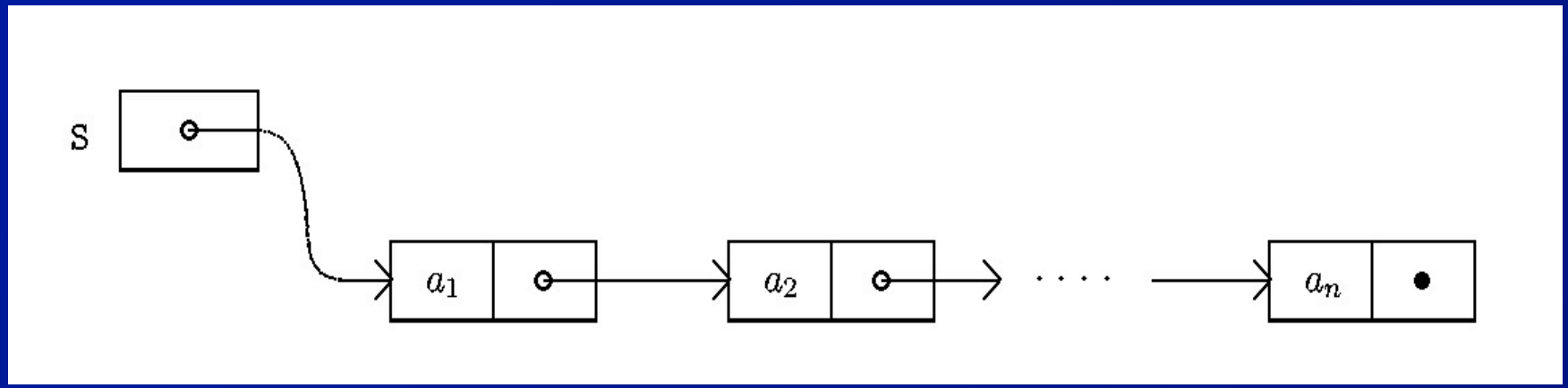
```
#define MAXLENGTH ...

  typedef struct {
      int top;
      elementtype elements[MAXLENGTH];
  } STACK;

  void MAKE_NULL (STACK *S_ptr)  {
       S_ptr->top = MAXLENGTH;
  }

 int EMPTY(STACK S)  {
       if (S.top >= MAXLENGTH)
           return 1;
       else
           return 0;
 }
```

```
void PUSH(elementtype x, STACK *S_ptr)  {
     if (S_ptr->top == 0)
         error("Stog je pun");
     else {
         S_ptr->top--;
         S_ptr->elements[S_ptr->top] = x;
     }
}

void POP(STACK *S_ptr) {
    if (EMPTY(*S_ptr))
        error("Stog je prazan");
    else
        S_ptr->top++;
}

Elementtype TOP(STACK S) {
    if (EMPTY(S))
        error("Stog je prazan");
    else
        return (S.elements[S.top]);
}
```

Broj koraka u izvršavanju svake funkcije je 1 ⇒ vrlo efikasna i jednostavna implementacija

Implementacija stoga pomoću pokazivača

- Zasniva se na vezanoj listi
- Kod stoga ne postoji pojam pozicije pa nije potrebna polazna ćelija glava (header), već je dovoljan pokazivač na prvu ćeliju, što pojednostavljuje strukturu
- Ćelija je građena jednako kao u slučaju vezane liste
- Vrh stoga je na početku vezane liste
- Stog se poistovjećuje s pokazivačem na početak vezane liste



Implementacija stoga pomoću pokazivača

- MAKE_NULL(&S) pridružuje S=NULL
- EMPTY(S) provjerava da li je S==NULL
- TOP(S) vraća S->element (ako je S neprazan)
- Vrijeme izvršavanja bilo koje operacije je O(1)
- Funkcije PUSH() i POP() liče na INSERT() i DELETE() iz implementacije liste pomoću pokazivača, ali su jednostavnije, jer rade samo na početku liste

- Prikaz stoga pomoću liste zahtijeva više memorije po podatku (jer postoji i pokazivač), međutim daje veću fleksibilnost
- Više stogova može paralelno koristiti isti memorijski prostor.
- Korištenje memorije je proporcionalno broju podataka na stogu, a nije određeno maksimalnim kapacitetima stogova
- kapacitet pojedinog stoga ograničen je samo raspoloživom memorijom.

Red (QUEUE)

- Red je specijalna vrsta liste (kao red ljudi u banci ili dućanu)
- Elementi se ubacuju na jednom kraju liste (začelje), a izbacuju na suprotnom kraju (čelo)
- Prvi-unutra-prvi-van lista (FIFO)
- Primjeri za red:
  - ljudi koji čekaju na blagajni u dućanu
  - štampač na lokalnoj mreži računala
  - izvođenje programa u batch modu
- Također se može definirati kao posebni apstraktni tip podatka

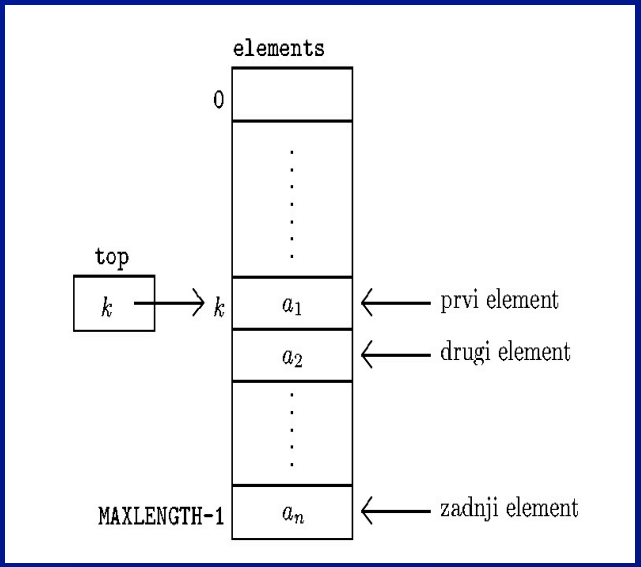
Apstraktni tip podatka QUEUE

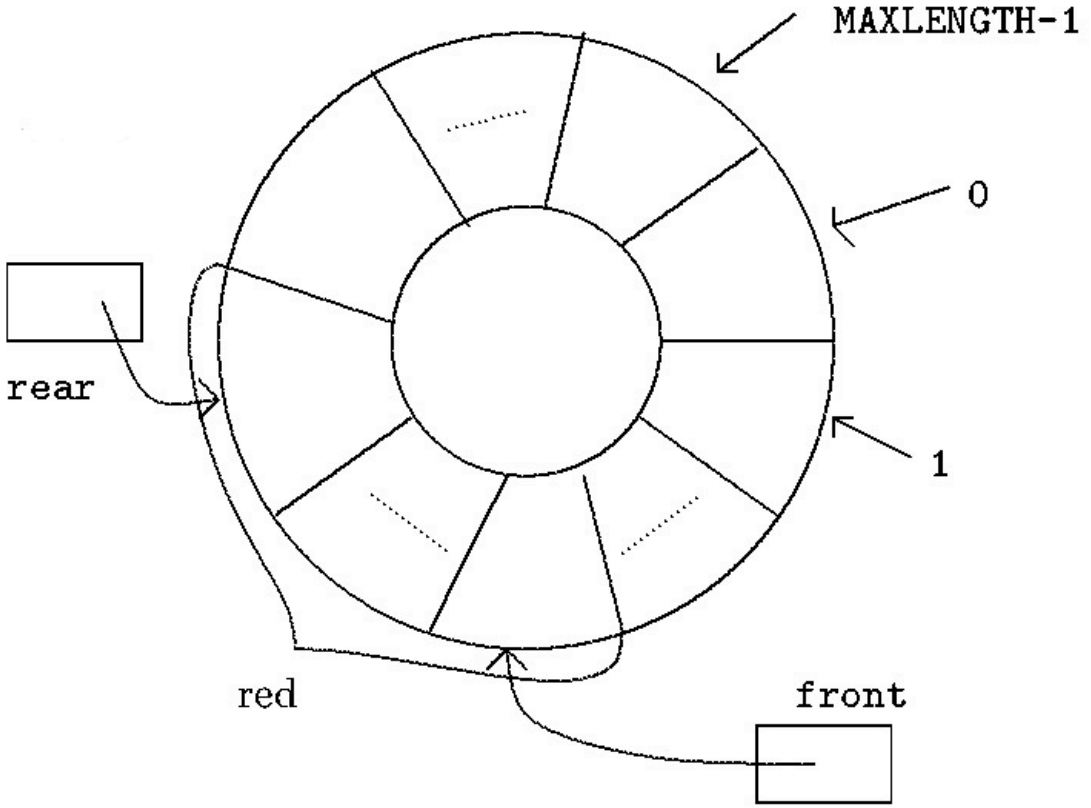
- elementtype … bilo koji tip
- QUEUE … podatak tipa QUEUE je konačni niz podataka tipa elementtype
- MAKE_NULL(&Q) … funkcija prazni red Q
- EMPTY(Q) … provjerava je li red Q prazan
- ENQUEUE(x, &Q) … funkcija dodaje element x na začelje reda Q; u terminologiji operacija na listama, to je INSERT(x, END(Q), &Q)
- DEQUEUE(&Q) … funkcija briše element na čelu reda Q; odgovara operaciji na listama DELETE(FIRST(Q),&Q)
- FRONT(Q) funkcija vraća vrijednost elementa na čelu reda Q, dok red ostaje nepromijenjen; ekvivalent operaciji na listama RETRIEVE(FIRST(Q),Q)

- Implementacija reda tako se može dobiti iz implementacije općenite liste uz odgovarajuća pojednostavljenja

Implementacija reda pomoću cirkularnog polja

- Može se doslovno preuzeti implementacija liste pomoću polja i uzeti a_1 za čelo
- Funkcija ENQUEUE() se tada obavlja u jednom koraku jer ne zahtjeva pomicanje ostalih elemenata liste
- Ali zato funkcija DEQUEUE() zahtjeva da se cijeli ostatak reda prepiše za jedno mjesto prema gore
- Trik: uvede se još jedan kursor (pokazivač) na početak reda, ne treba se više prepisivati, ali ubacivanjem/izbacivanjem red putuje prema donjem kraju polja
- Bolje rješenje je cirkularno polje: nakon zadnjeg indeksa ponovo slijedi početni indeks
  - Red zauzima niz uzastopnih ćelija polja i postoje kursor na čelo i začelje
  - Cirkularnost se postiže tako da s indeksima računamo modulo MAXLENGTH
  - Prazan red: čelo == začelje
  - Puni red: (začelje+1) % MAXLENGTH == čelo




MAXLENGTH-1
0
1
rear
red
front

C kod za implementaciju reda cirkularnim poljem

```
#define MAXLENGTH …

 typedef struct {
     elementtype elements[MAXLENGTH];
     int front, rear;
 } QUEUE;

 int addone(int i) {
     return ((i+1) % MAXLENGTH);
 }

 void MAKE_NULL(QUEUE *Q_ptr)  {
     Q_ptr->front = 0;
     Q_ptr->rear = 0;
}
```

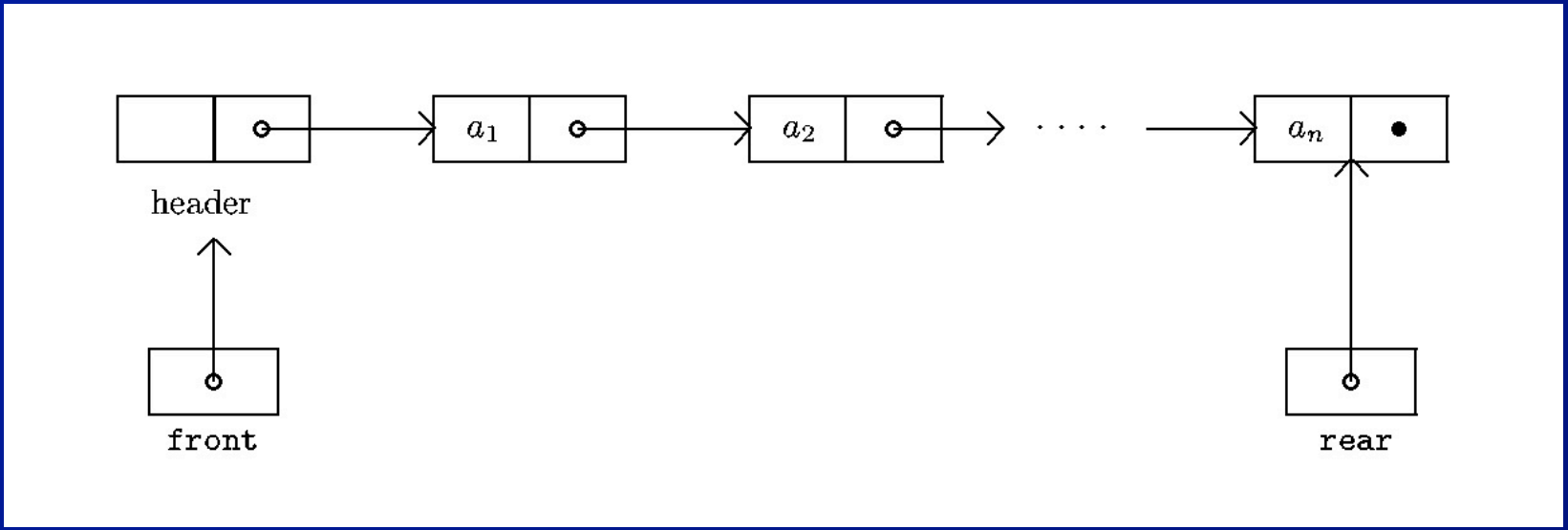
```
int EMPTY(QUEUE Q)  {
     if (Q.rear== Q.front) return 1;
     else return 0;
}

void ENQUEUE(elementtype x, QUEUE *Q_ptr)  {
    if (addone(Q_ptr->rear) == (Q_ptr->front))
        error("Red je pun");
    else {
        Q_ptr->rear = addone(Q_ptr->rear);
        Q_ptr->elements[Q_ptr->rear] = x;
    }
}

void DEQUEUE(QUEUE *Q_ptr)  {
    if (EMPTY(*Q_ptr)) error("Red je prazan");
    else Q_ptr->front = addone(Q_ptr->front);
}
```

```
elementtype FRONT(QUEUE Q)  {
     if (EMPTY(Q)
         error(“Red je prazan”);
     else
         return (Q.elements[Q.front]);
}
```

Broj koraka u izvršavanju bilo koje funkcije je jedan, tj. vrijeme izvršavanja je konstantno i ne ovisi o broju elemenata u redu, O(1).

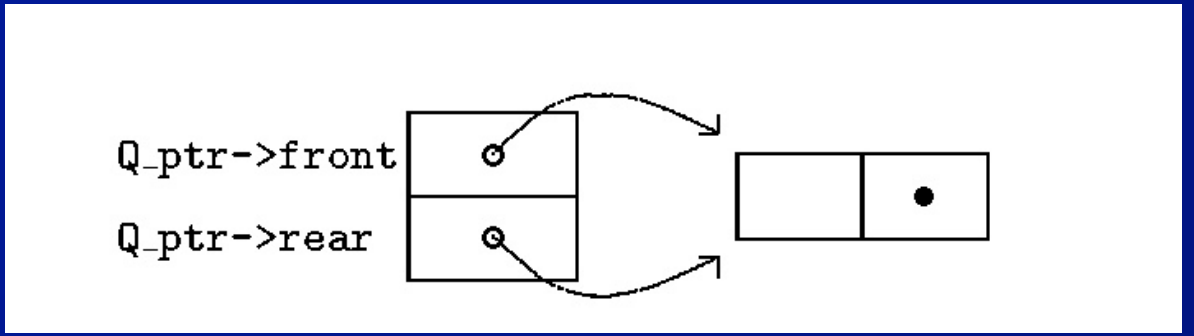
Implementacija reda pomoću pokazivača

- Radi se slično kao u slučaju vezane liste
- Početak vezane liste je čelo reda
- Dodaje se još pokazivač na kraj vezane liste (začelje reda)
- Glava (header) olakšava prikaz praznog reda
- Vrijeme izvršavanja svake funkcije je također konstantno (jedan korak) O(1)

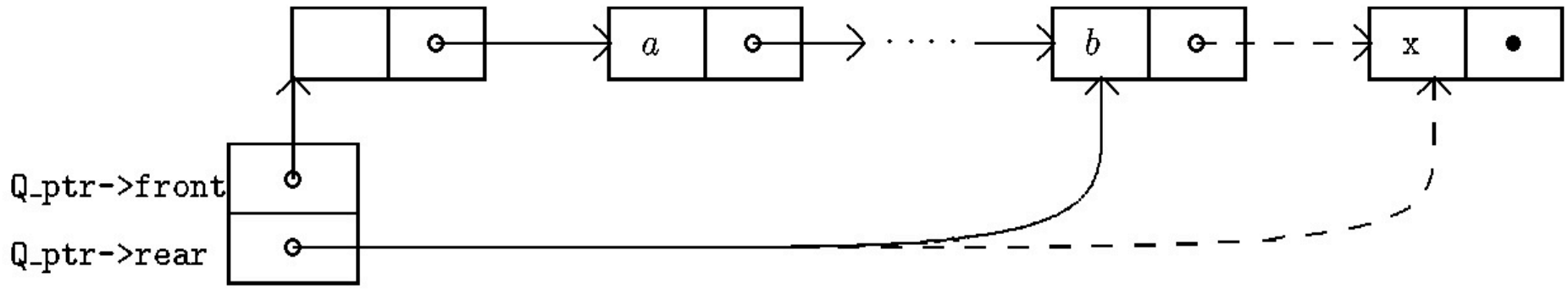


C kod za implementaciju reda pomoću pokazivača

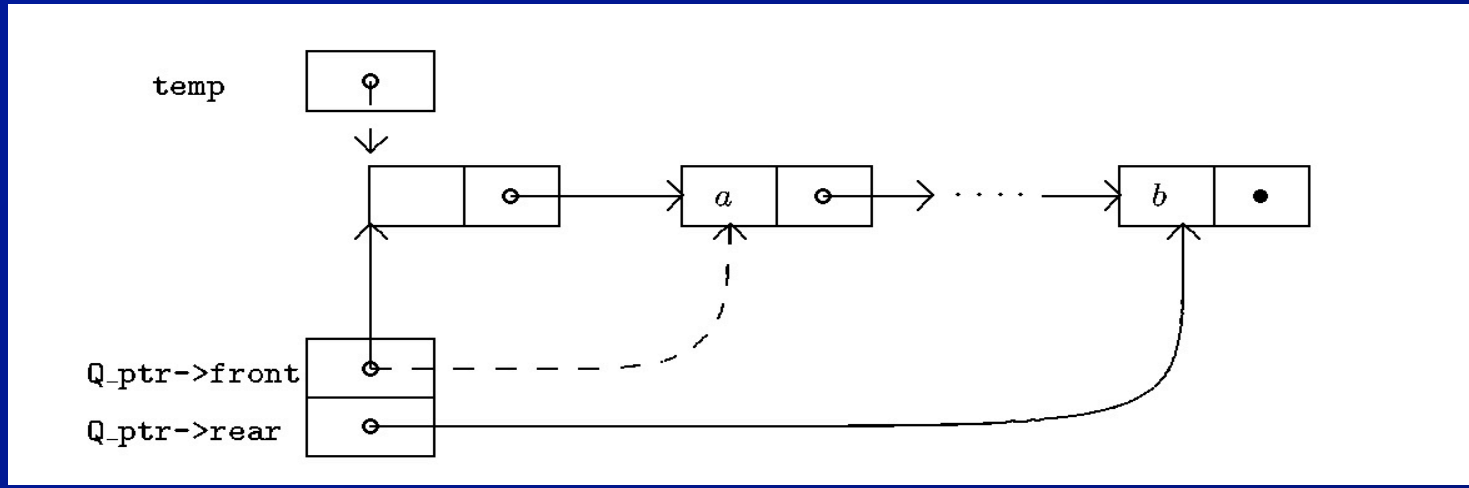
```
typedef struct cell_tag {
    elementtype element;
    struct cell_tag *next;
}  celltype;

typedef struct {
    celltype *front, *rear;
} QUEUE;

void MAKE_NULL(QUEUE *Q_ptr) {
    Q_ptr->front = (celltype*) malloc(sizeof(celltype));
    Q_ptr->front->next = NULL;
    Q_ptr->rear = Q_ptr->front;
}
```



```
int EMPTY(QUEUE Q) {
     if (Q.front == Q.rear) return 1;
     else return 0;
}

void ENQUEUE(elementtype x, QUEUE *Q_ptr)  {
   Q_ptr->rear->next = (celltype*) malloc(sizeof(celltype));
   Q_ptr->rear = Q_ptr->rear->next;
   Q_ptr->rear->element = x;
   Q_ptr->rear->next = NULL;
}
```



```
void DEQUEUE(QUEUE *Q_ptr)  {
     celltype *temp;
     if (EMPTY(*Q_ptr)) error("Red je prazan");
     else {
          temp = Q_ptr->front;
          Q_ptr->front = Q_ptr->front->next;
          free(temp);
     }
}
```




```
 elementtype FRONT(QUEUE Q) {
     if (EMPTY(Q)) error ("Red je prazan");
     else return (Q.front->next->element);
 }
```

Domaća zadaća

- 1. Naći bolji algoritam za onaj problem s dva jajeta i zgradom
- 2. Napravite HP kalkulator (funkcije koje treba imati su zbrajanje i oduzimanje) koristeći ATP stog preko implementiran preko pokazivača