Structuri de Date

Anul universitar 2019-2020

Prof. Adina Magda Florea

Curs Nr. 2

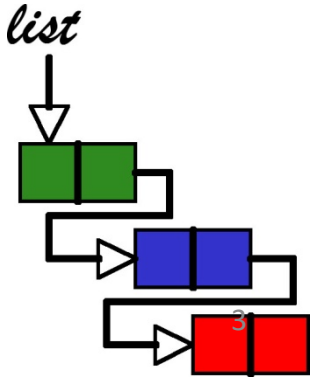
Liste înlănţuite

- Definitie
- Avantaje
- Reprezentari
- Operatii de baza cu liste
- Reprezentari extinse
- Variante de creare liste

1. Liste înlănţuite - caracteristici

Structură de date de bază

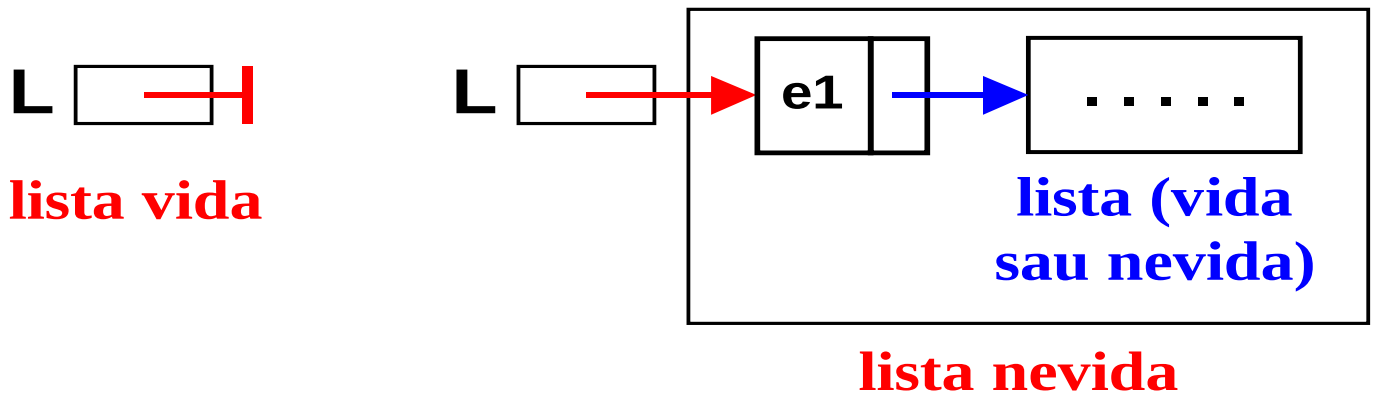
- O serie de obiecte ordonate
- O mulțime de elemente în care fiecare element face parte dintr-un nod care conține și un link la un alt nod sau vid
- Fiecare obiect are un obiect urmator - **next** (cu excepţia ultimului) şi un obiect precedent – **prev** (cu excepţia primului)



Liste înlănţuite

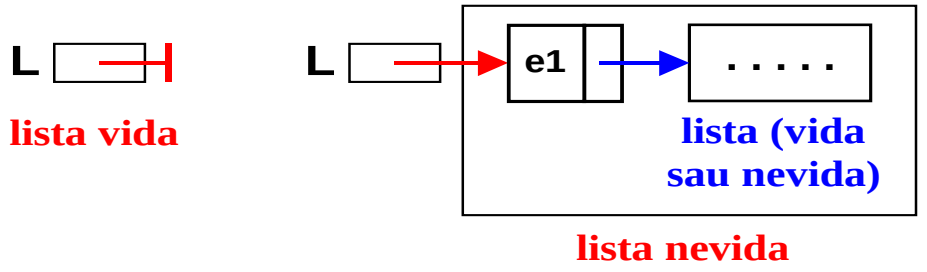
Definiție recursivă. O listă este:

- o listă vidă
- o legatură (link) la un nod care conține o valoare și o legatură (link) la o listă înlanțuită



Liste înlănţuite

- Elementele liste: celule, noduri
- **Alocarea memoriei** pentru liste se face la nivelul fiecarui nod, numai cand avem nevoie de un nou nod (un nou element)
- Alocarea este dinamica (se realizeaza in heap)
- O **variabila de tip lista** are ca valoare
  - NULL daca lista este vida
  - adresa primei celule daca lista este nevida



2. Liste vs. vectori

Vectori

- Necesita estimarea nr elem
- Cautarea unui element se face direct, prin index; elem k costa O(1)
- Inserarea si eliminarea unui element gasit se face in O(N)
- Compacte in memorie; se aloca static sau dinamic cu dimensiune fixa

Liste inlantuite

- Nu necesita sa stim nr elem
- Trebuie sa parcurgem toate elementele pana la elementul cautat; elem k costa O(N)
- Inserarea si eliminarea unui element gasit se face in O(1)
- Nu sunt compacte in memorie, au nevoi de spatiu suplim pt legaturi

3. Reprezentare (R1)

```
typedef int Item;
typedef struct cel {
        Item    elem;
        struct  cel  *next;} ListCel, *TList;
```

cu declaratia

```
TList  head = NULL;
```

si alocarea unei celule

```
head = (TList) malloc(sizeof (ListCel));
```

NB: Se recomanda ca variabilele de tip lista (TList) sa fie initializate cu NULL

Reprezentare (R2)

sau

R1 si R2 sunt echivalente

```
typedef struct node *Link;
typedef struct node {
                Item  elem;
                Link  next; } ListNode;
```

cu declaratia

```
Link head = NULL;
```

si alocarea unei celule

```
head = (Link) malloc(sizeof (ListNode));
/*  sau   t1 = malloc(sizeof  *t1);  */
```

4. Adresari de baza in liste

- **TList p = NULL;**
- **test lista vida: (p == NULL)**
- **Pentru lista nevida (p != NULL)**
  - **valoarea primului element: p->elem**
  - **adresa primului element: &(p->elem)**
  - **adresa urmatoarei liste (celule): p->next**
  - **test continuare lista: (p->next != NULL)**
  - **valoarea urmatorului element: (p->next->elem)**
  - **avans la urmatoarea lista (celula): (p = p->next)**

5. Operatii de baza cu liste

- Testare lista vida
- Parcurgere lista (afisare ordine directa, ordine inversa)
- Creare lista
- Lungime lista (numar de elemente)
- Cautare un element in lista
- Inserare un element in lista (la inceput, la sfarsit, dupa un anumit element, in lista ordonata)
- Eliminare element (de la inceput, de la sfarsit, un element gasit)
- Distrugere lista (eliminarea tututor elementelor)

5.1 Parcurgere lista

Algoritm parcurgere lista si afisare elemente

/* Primeste lista referita de head
Nu modifica lista */

t ← head

cat timp t != NULL **repeta** /* nu am ajuns la sfarsit */

- afiseaza informatia din nodul referit de t
- mergi la urmatorul nod

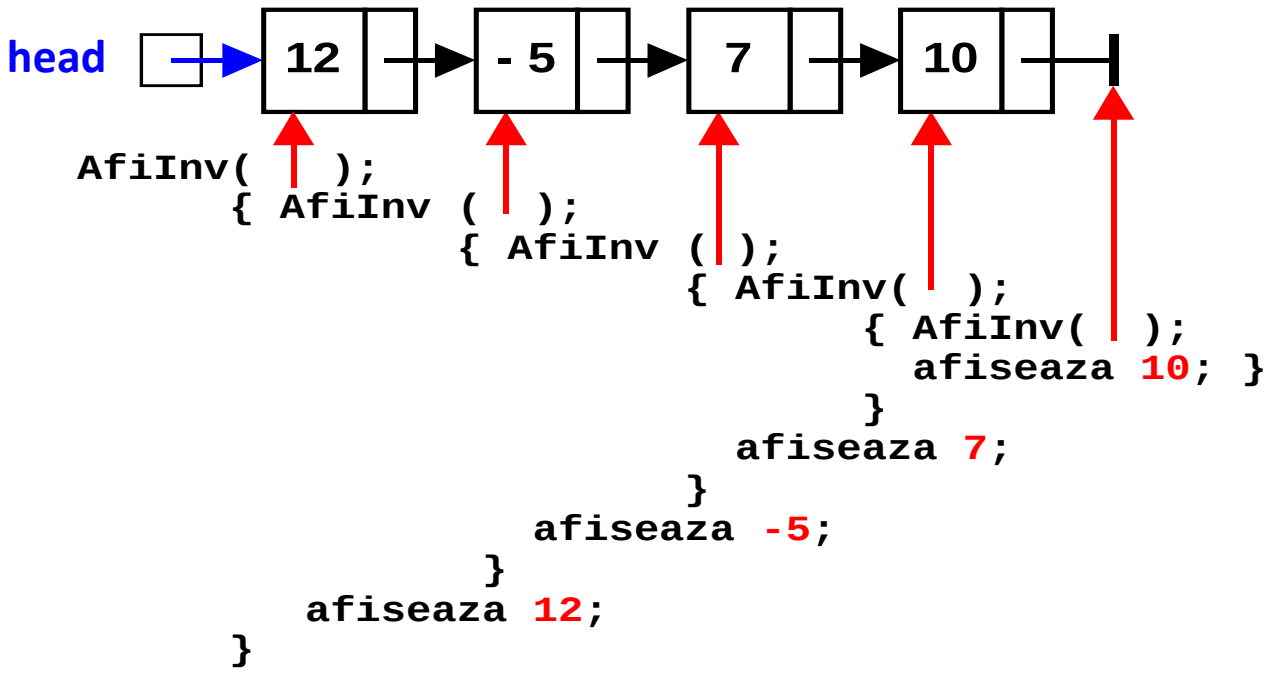
sfarsit

```
TList  t, head;
```

```
t = head;
while (t !=NULL)
            { printf("%d ", t->elem);
              t = t->next;}
```

Afisare elemente in ordine inversa

```
void AfiInv(TList head)
{    if (head!=NULL) {
            AfiInv(head->next);
            printf("%d ", head->elem);}
}
```



5.2 Creare lista

Algoritm creare lista

/* Creaza lista si intoarce un pointer catre primul element */

head = NULL /* initializez la lista vida */
cat timp mai sunt date *d* **repeta**
- aloca in *t* memorie pentru un nou nod
  /**t* pointer la nod */
- depune *d* in nodul referit de *t* (t -> elem=d)
- actualizeaza legatura din nodul *t* la *head*
  (t -> next = head)
- *head* = *t*

intoarce *head*
sfarsit

Creare lista (R1)

```
typedef int Item;
typedef struct cel {
          Item    elem;
          struct  cel  *next;} ListCel, *TList;
```

```
TList head = NULL,t;
```

```
for(i=1; i<=10; i++) {
      t = (TList) malloc(sizeof(ListCel));
      t -> elem = i;
      t -> next  = head;
      head = t;
   }
```

Creare lista (R2)

```
typedef int Item;
typedef struct node *Link;
typedef struct node {
            Item  elem;
            Link  next;} ListNode;

Link create()
{ Link t, head=NULL; int i;
    for(i=1;i<=10;i++) {
        t = (Link)malloc(sizeof(ListNode));
        t -> elem = i;
        t -> next = head;
        head = t;
    }
    return head;
}
```

Acelasi mecanism

5.3 Cautare element *x* in lista

Cautare element *x* in lista *head*

/ intoarce pointer la elementul gasit sau NULL */*

p = head

cat timp *p* != NULL si p -> elem != x **repeta**

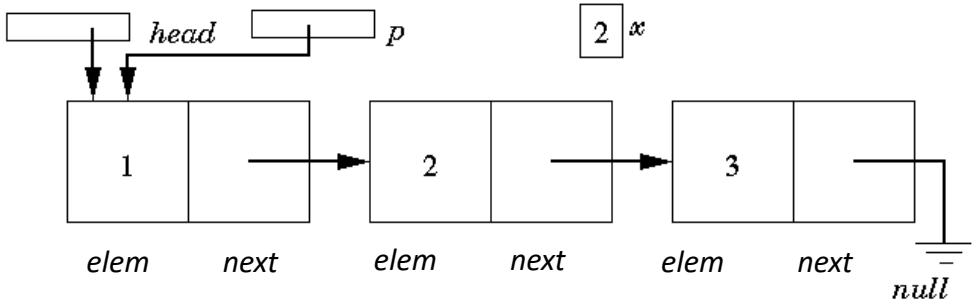
- p = p->next

daca p == NULL **atunci intoarece** NULL /* nu s-a gasit */

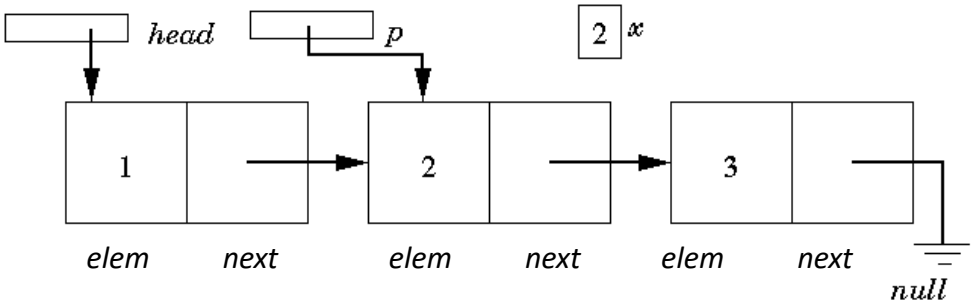
altfel intoarce *p*

sfarsit

Inainte de intrarea in bucla



Dupa prima iteratie



5.4 Inserare element *d* in lista

(1) Inserare la inceputul listei *head*

- aloca in *t* memorie pentru un nou nod

/**t* pointer la nod */

- depune *d* in nodul referit de *t* (**`t -> elem = d`**)
- actualizeaza legatura din nodul *t* la *head*

(**`t -> next = head`**)

- *head* = *t*;
- **intoarce** *head*

sfarsit

Inserare element *d* in lista

(2) Inserare la sfarsitul listei *head*

- aloca in *t* memorie pentru un nou nod

/**t* pointer la nod */

- t -> next = NULL /* *t* va fi ultimul element */
- depune *d* in nodul referit de *t* (**`t -> elem=d`**)

daca head=NULL **atunci intoarce** *t*

altfel

- localizeaza adresa ultimului element *prev* al listei
- actualizeaza legatura din *prev* la *t*

(**`prev->next = t`**)

- **intoarce** *head*

sfarsit

Inserare element in *d* lista

(3) Inserare dupa un anumit element *x* al listei *head*

- aloca in *t* memorie pentru un nou nod

/**t* pointer la nod */

- depune *d* in nodul referit de *t* (**`t -> elem=d`**)
- localizeaza adresa elementului *x*

daca elementul x exista in lista **atunci**

- actualizeaza legatura din *t* la nodul ce urmeaza lui *x*

(**`t -> next = x -> next`**)

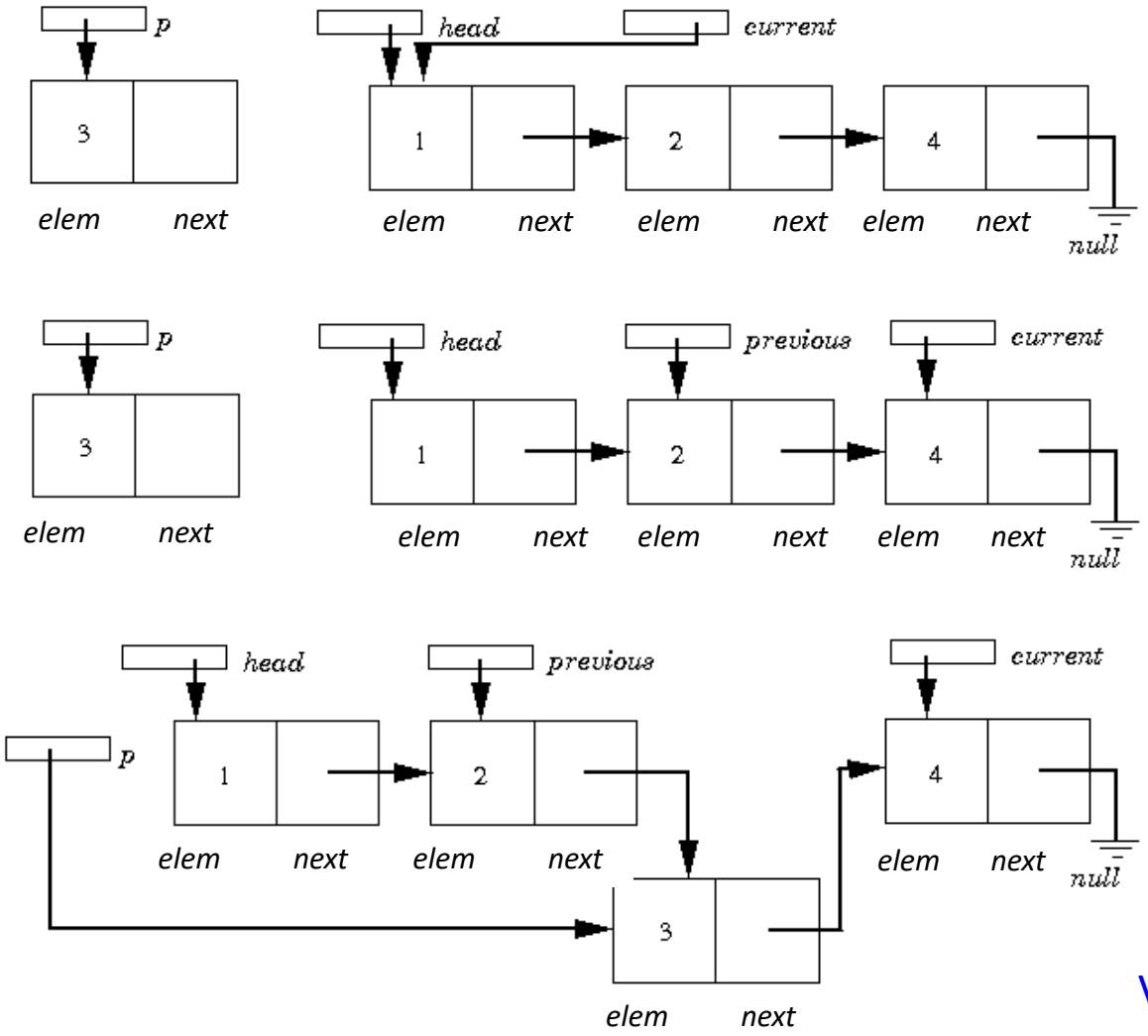
- actualizeaza legatura din *x* la *t* (**`x -> next = t`**)

intoarce *head*

sfarsit

Inserare element in lista

(4) Inserare element intr-o lista ordonata *head*



Vezi laborator

5.5 Eliminare element din lista

(1) Eliminare prim element din lista *head*

daca head == NULL **atunci intoarce** esec;

t = salveaza adresa celulei de eliminat

modifica adresa inceput lista (**`head = head->next`**)

elibereaza spatiul ocupat de celula eliminata (**`free(t)`**)

intoarce succes;

sfarsit

Eliminare element din lista

(2) Eliminare unui element *d* din lista *head*

daca head == NULL **atunci intoarce** esec;

localizeaza adresa *x* a elementului predecesor lui *d*

daca *d* nu exista in lista **atunci intoarce** esec

t = salveaza adresa celulei de eliminat

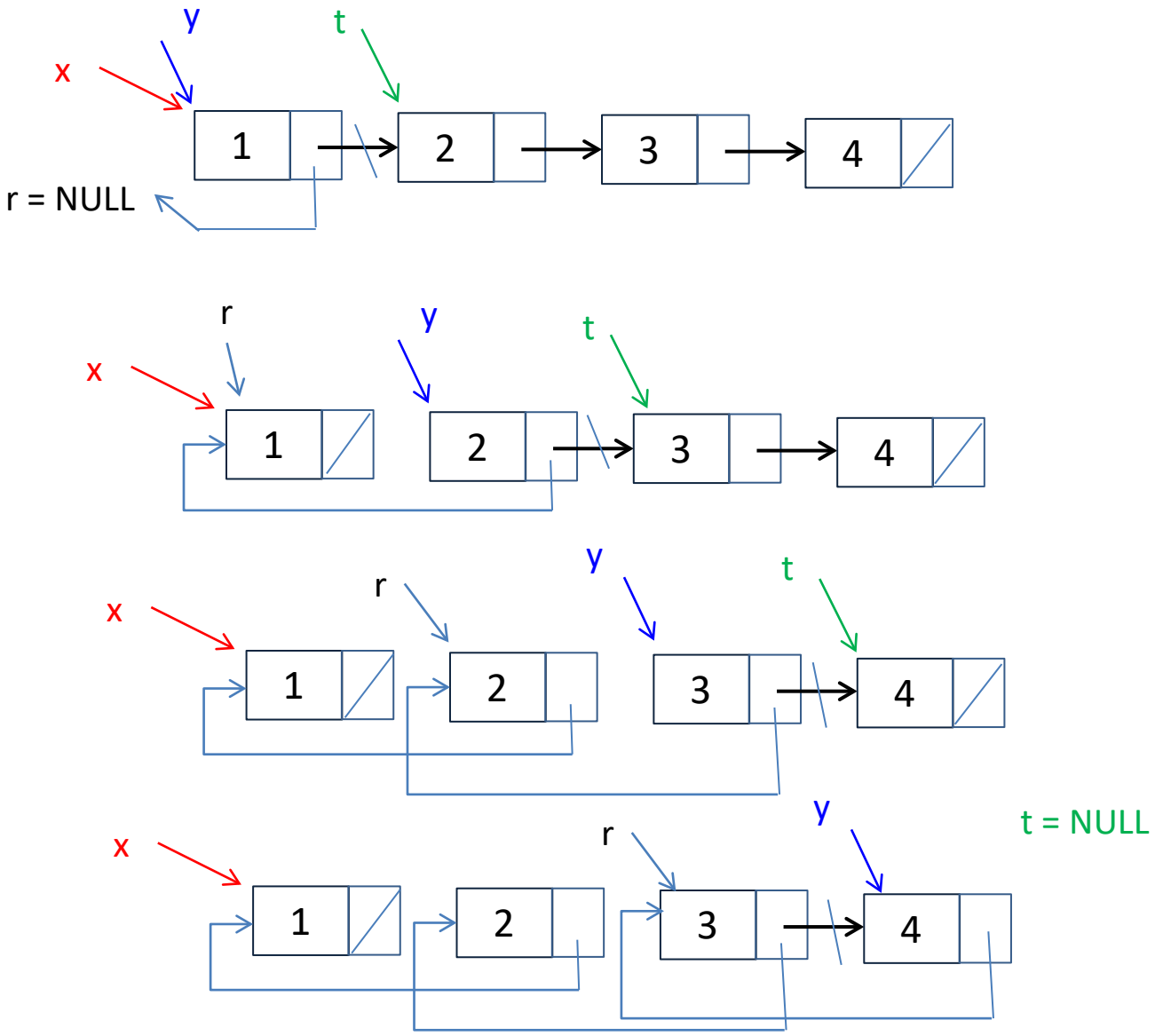
elimina celula (**`x->next = x->next->next`**)

elibereaza spatiul ocupat de celula eliminata (**`free(t)`**)

intoarce succes;

sfarsit

5.6 Inversarea unei liste



Inversarea unei liste - recursiv

Link inverse(Link head)

/* intoarce pointer la lista inversata */

daca head == NULL **atunci intoarce** NULL

t = inverse(head->next)

t1 = ins_la_sfarsit(t, head->elem)

elibereaza prima celula (**`free(head)`**)

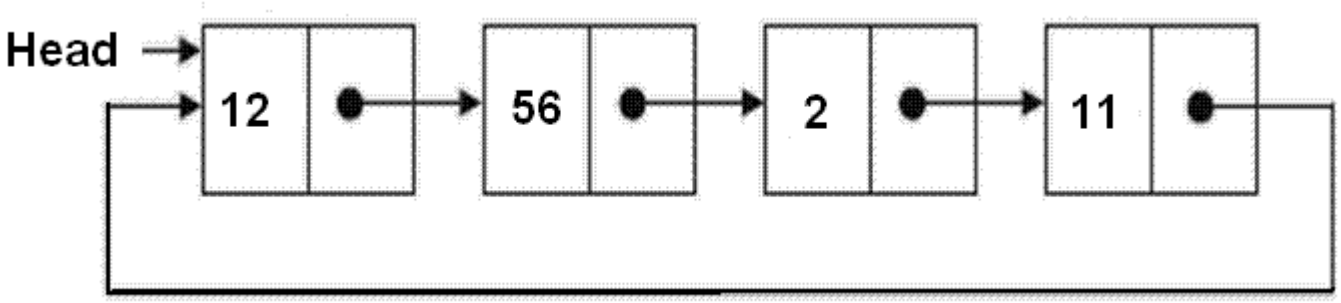
intoarce t1

sfarsit

6. Lista circulara

- De multe ori este convenabil sa utilizam reprezentari speciale pentru liste, de ex liste circulare sau liste cu celule suplimentare care sa faciliteze operatiile cu aceste structuri

Lista circulara – nu va fi vida niciodata



6.1 Lista circulara - implementare

Inserare prim element

```
head->next = head;
```

Inserare *t* dupa *x*

```
t -> next = x->next; x -> next = t;
```

Eliminare element dupa x

```
x -> next = x -> next -> next;
```

Parcurgere

```
t = head;
do { … t = t -> next;} while (t != head);
```

Testare daca lista are 1 element

```
if (head -> next = head)
```

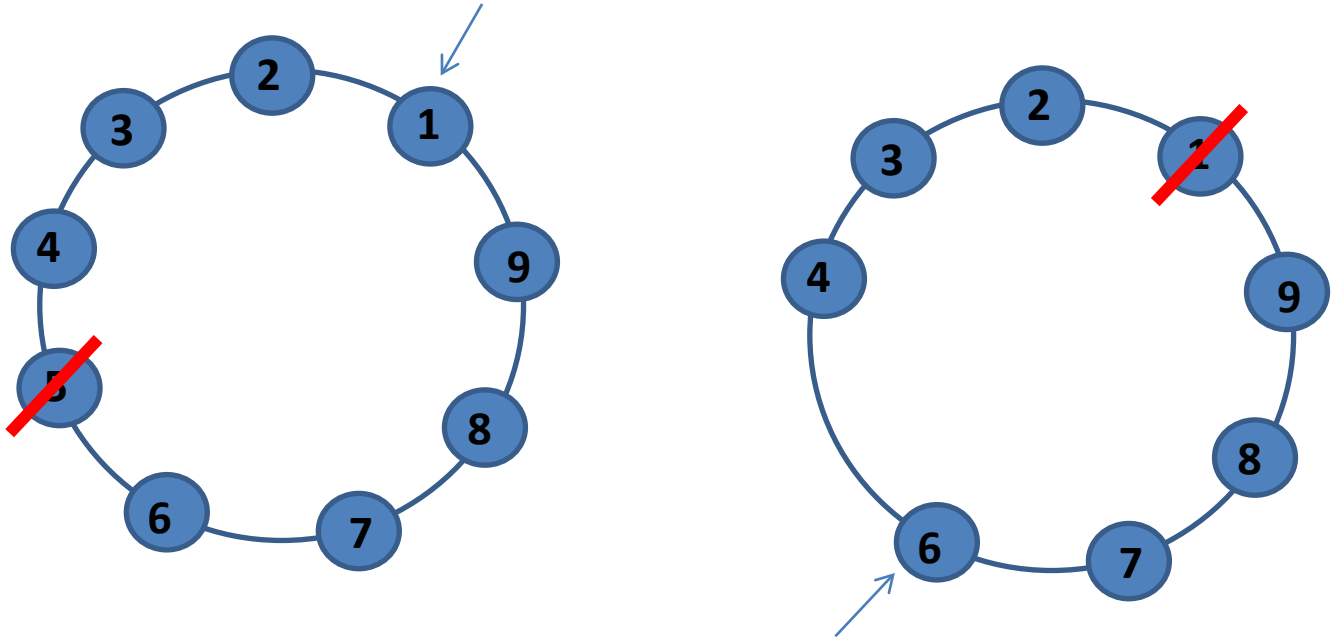
6.2 Problema Iosefiana (Josephus problem)

- Un numar de N persoane se aseaza intr-un cerc. Se elimina repetitiv persoana de pe pozitia M pana ramane o singura persoana. Acea persoana este aleasa leader.

Iosephus Flavius

Problema Iosefiana (Josephus problem)

- Vom implementa aceasta problema folosind o lista circulara



Problema Iosefiana (Josephus problem)

```
typedef int Item;
typedef struct node *link;
typedef struct node {
            Item  elem;
            link  next;} ListNode;
link jcreate(int N)
{   link x; int i;
    link head = malloc(sizeof *t);  x = head;
    head->elem = 1; head->next = head;  /* lista circulara */
    for(i=2; i<=N; i++)
    {
        x = (x->next = malloc(sizeof *x));
        x->elem = i; x->next=head;
    }
    return t;
}
```

Problema Iosefiana (Josephus problem)

```
link jelim(link l,int M)
{
    link x=l, t; int i;
    for(i=1; i<M; i++) {t=x;x=x->next;}
    t->next=x->next;
    printf("am eliminat pe %d\n", x->elem);
    free(x);
    return t->next;
}
void listprint(link l)
{
    link t=l;
    while(t->next!=l)
            {printf("%d ", t->elem); t=t->next;}
    printf("%d \n", t->elem);
}
```

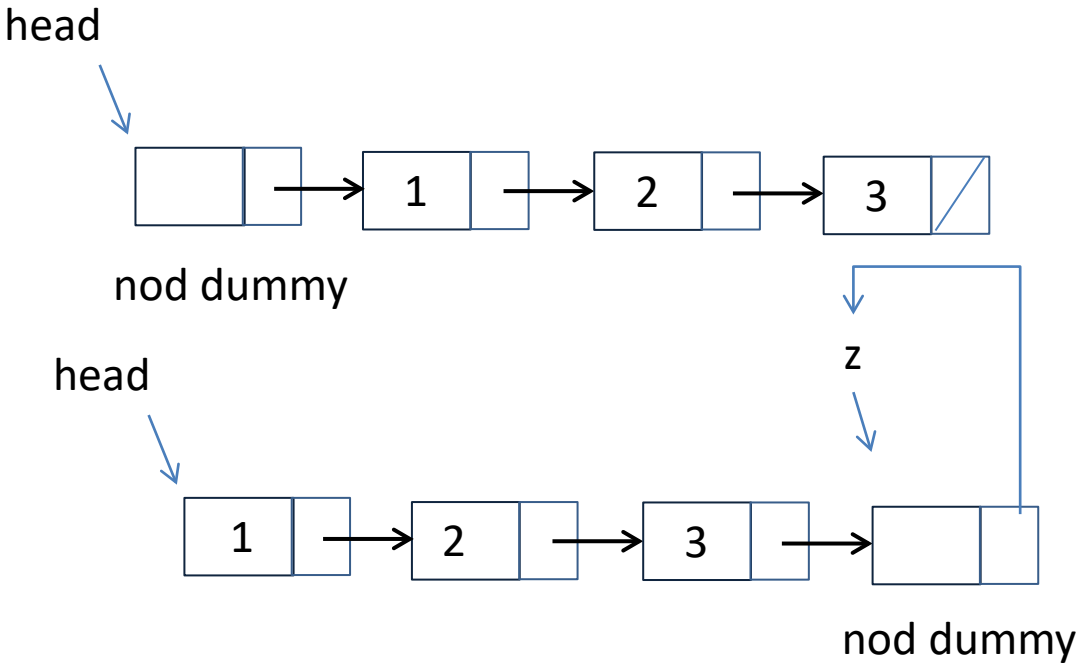
Problema Iosefiana (Josephus problem)

```
main()
{
int i, n=9, m = 5;
link mylist,t;
mylist = jcreate(n);
printf("\n Lista initiala \n");
listprint(mylist);
printf("\n Eliminare \n");
while(mylist!=mylist->next)
    { mylist=jelim(mylist,m);
      listprint(mylist);
    }

}
```

```
 Lista initiala
1 2 3 4 5 6 7 8 9

 Eliminare
am eliminat pe 5
6 7 8 9 1 2 3 4
am eliminat pe 1
2 3 4 6 7 8 9
am eliminat pe 7
8 9 2 3 4 6
am eliminat pe 4
6 8 9 2 3
am eliminat pe 3
6 8 9 2
am eliminat pe 6
8 9 2
am eliminat pe 9
2 8
am eliminat pe 2
8
```

7. Lista cu santinela (dummy)

- Folosim un nod santinela (dummy) pe post de prim element de lista
- Poate fi folosit si ca ultim element in lista
- Acestea nu contin o informatie utila dar pot ajuta la operatiile executate pe liste



7.1 Lista cu celula santinela la inceput

Initializare

```
head = malloc(sizeof *head);
head -> next = NULL;
```

Inserare *t* dupa *x*

```
t -> next = x->next; x -> next = t;
```

Parcurgere

```
For (t = head->next; t!=NULL; t = t->next)
```

Testare daca lista este vida

```
if (head -> next = NULL)
```

7.2 Lista cu celula santinela la sfarsit

Initializare
```
z = malloc(sizeof *z);
head = z; z->next = z;
```

Inserare *t* dupa *x*
```
t -> next = x->next; x -> next = t;
```

Eliminare t dupa x
```
x->next = x->next->next;
```

Parcurgere
```
For (t = head; t!=z; t = t->next)
```

Testare daca lista este vida
```
if (head == z)
```

8. Crearea / Inserarea in lista

- Crearea sau inserarea in lista implica alocare de spatiu. Deoarece aceasta poate esua se recomanda testarea rezultatului operatiei de alocare.

```
typedef struct node *Link;
typedef struct node {
                    Item  elem;
                    Link  next; } ListNode;
Link head = NULL;
.........
head = (Link) malloc(sizeof (ListNode));
if (head==NULL) { printf("ERROR"); exit(1);}
else {….prelucrari …
```

Inserarea in lista

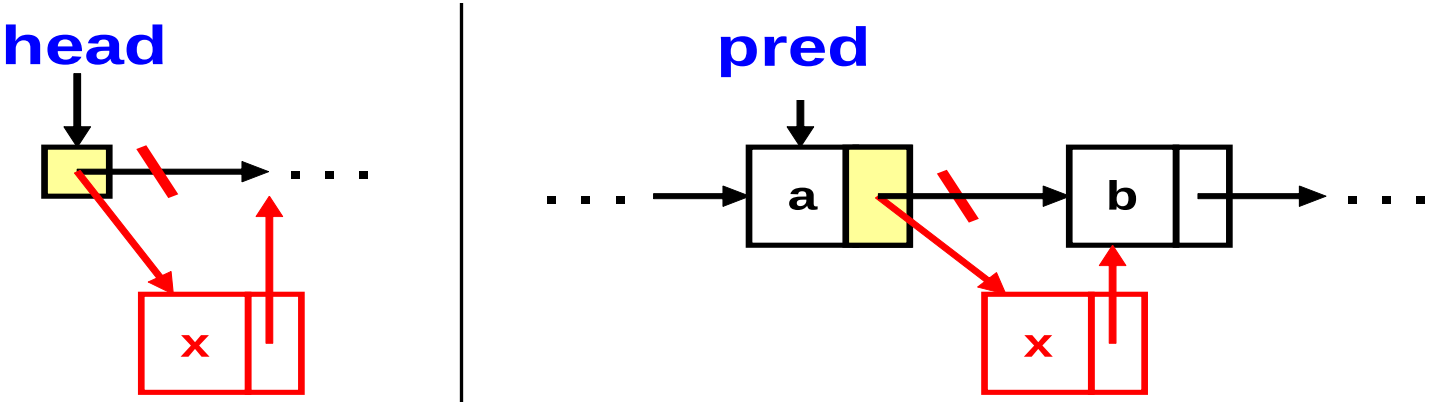
- In cazul listelor ce reprezinta ***multimi*** inserarea se efectueaza numai daca lista nu contine deja elementul de inserat → este necesara o cautare prealabila si eventuala semnalare a incercarii de duplicare.

Inserarea in lista

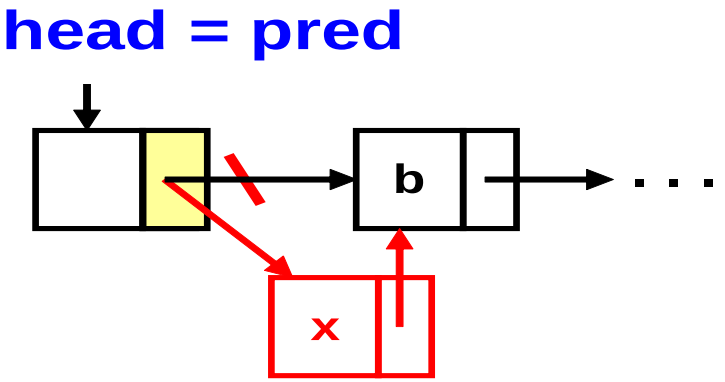
- Locul de inserare se stabileste in functie de organizarea listei, determinata de specificul aplicatiei
  - La inceput in cazul colectiilor nesortate → O(1)
  - La sfarsit in cazul colectiilor in care trebuie respectata ordinea de inserare → O(N)
  - In locul stabilit de o conditie / relatie de ordine, de exemplu in cazul unei liste de angajati, sortata dupa CNP sau alta informatie cu rol de **cheie** → O(N)

Particularitati inserare celula

In cazul listei fara santinela (dummy) inserarea la inceput difera de cea in interiorul listei.



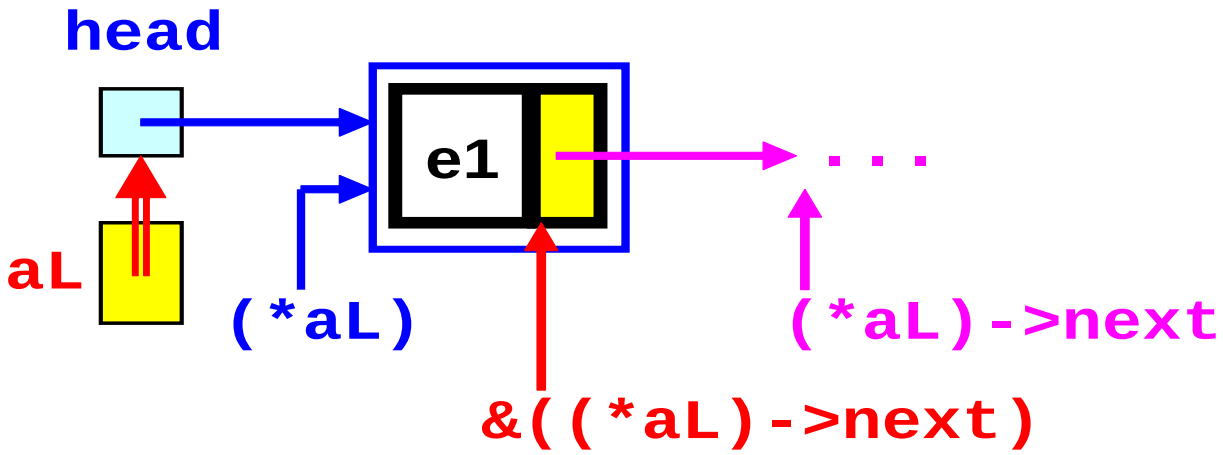
Utilizarea santinelei unifica cele 2 tratari.



Inserare folosind adresa legaturii

Cele 2 tratari pot fi unificate si prin utilizarea unui pointer la legatura afectata.

```
Link *aL = &head;
```



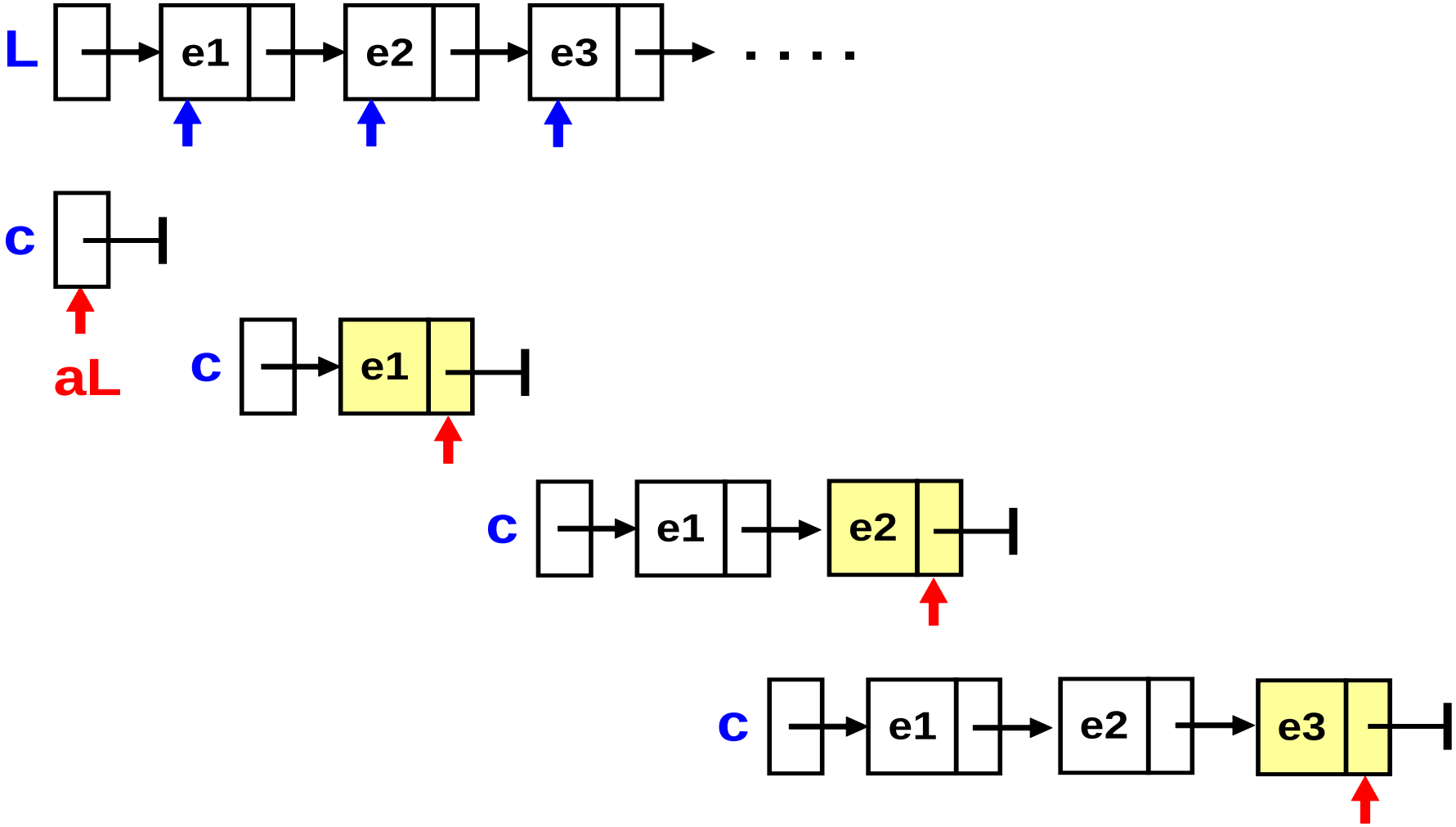
In acest caz avansul in lista se codifica astfel:

```
aL = &((*aL)->next)
```

Variante de creare a unei liste

- **Lista cu elemente citite** – trebuie precizata o conditie de oprire (de exemplu introducerea unui caracter cu rol de terminator). Elementele se memoreaza in ordinea citirii, ceea ce implica inserari la sfarsitul listei create.
- **Elementele sunt generate aleator** - in acest caz se pot face sistematic inserari la inceputul listei.
- **Lista cu elemente copiate** - se construieste **o copie**, integrala sau partiala, a unei liste existente.

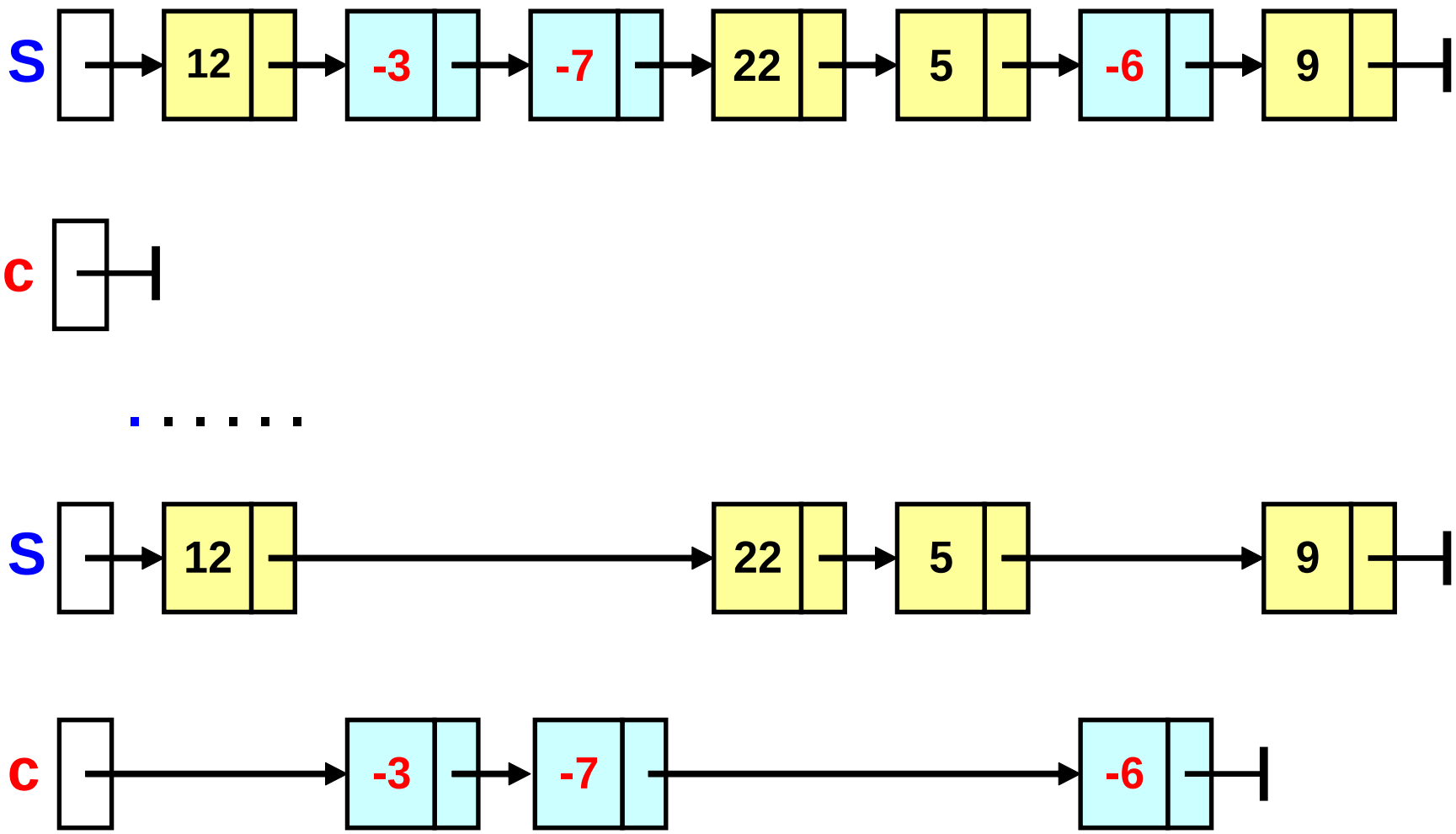
Exemplu copiere lista



Algoritm copiere integrala lista

```
{ initializari;
  cat timp exista celula de copiat
  {  aloca celula copie
     daca nu reuseste alocare celula copie
      atunci
      { distruge copia partiala;
        intoarce NULL;
      }
      altfel
      { completează celula copie;
        avans in copie partiala;
        avans in lista sursa;
      }
  }
  intoarce adresa copie;
}
```

Exemplu transfer intr-o noua lista



Algoritm transfer intr-o noua lista

```
{ initializari;
  cat timp exista celula in lista sursa
  { daca indeplineste conditia de transfer
      atunci
      { deconecteaza din sursa;
        conecteaza in copie;
        avans in lista copie;
      }
      altfel
        avans in lista sursa;
  }
  intoarce adresa copie;
}
```

Sortare prin insertie cu lista

- Vom genera N valori intregi aleatoare
- Vom crea o lista cu aceste numere – lista ***a***
- Aranjam numerele in lista ***b*** a.i. sa fie ordonate crescator.

Sortare prin insertie cu lista

Algoritm sortare lista prin insertie

- construieste lista ***a*** cu N numere aleatoare
- initializeaza lista ***b***
- **pentru** fiecare nod ***t*** din lista ***a*** **repeta**
    - **pentru** fiecare nod ***x*** din lista ***b*** **repeta**
        daca valoarea din next ***x*** > valoarea din ***t***
        atunci break
    - introduce nodul **t** in lista **b**

sfarsit

Sortare prin insertie cu lista

- Folosim o celula santinela ca prim element atat a listei *a* cat si a listei *b*
- Putem astfel sa realizam omogen inserarea cu instructiunile

  `t -> next = x->next; x -> next = t;`

  a elementului *t* din *a* dupa *x* din *b*, *x* fiind ultimul element mai mic decat *t* din *b*

NB. Listele a si b nu ocupa memorie separata. Numai cele 2 celule dummy sunt 2 variabile distincte

Listele a si b au aceleasi celule numai ca ele sunt altfel aranjate

Sortare prin insertie cu lista

```
typedef struct node *Link;
typedef struct node {
            Item  elem;
            Link  next;} ListNode;
/*foloseste celula dummy la inceput */

ListNode heada, headb;
/*dummy first cells; nu sunt alocate dinamic */
Link t,u,x, a = &heada, b; int i;

for(i=0,t=a; i < 10; i++)
{
    t->next = malloc(sizeof *t);
    t = t->next; t->next = NULL;
    t->elem = rand()%100;
}
```

Sortare prin insertie cu lista

```
/* afisare lista nesortata */
t = a->next;
while(t!=NULL)
{  printf("%d ", t->elem);
   t=t->next;}
```

Sortare prin insertie cu lista

```
b = &headb; b->next = NULL;
for(t = a->next; t!=NULL; t=u)
{  u=t->next;
   for(x=b; x->next!=NULL; x=x->next)
      if(x->next->elem > t->elem)
                              break;
    t->next = x->next;
    x->next = t;
}
```

Sortare prin insertie cu lista

```
/* afisare lista sortata */
t = b->next;
while(t!=NULL)
{ printf("%d ", t->elem);
  t=t->next;}
```