Tehnici de programare - TP

Cursul 8 – Liste ordonate.
Liste liniare dublu înlănțuite. Liste circulare

Ș.l. dr. ing. Cătălin Iapă

catalin.iapa@cs.upt.ro

De data trecută – Liste simplu înlănțuite

Liste simplu înlănțuite - continuare

Liste ordonate

Liste dublu înlănțuite

Liste circulare

Liste liniare simplu înlănțuite

Listele liniare simplu înlănțuite sunt structuri de date care permit stocarea și gestionarea unui set de elemente într-o ordine specifică.

Fiecare element este legat printr-un pointer către următorul element din listă, astfel încât elementele pot fi accesate în ordine secvențială.

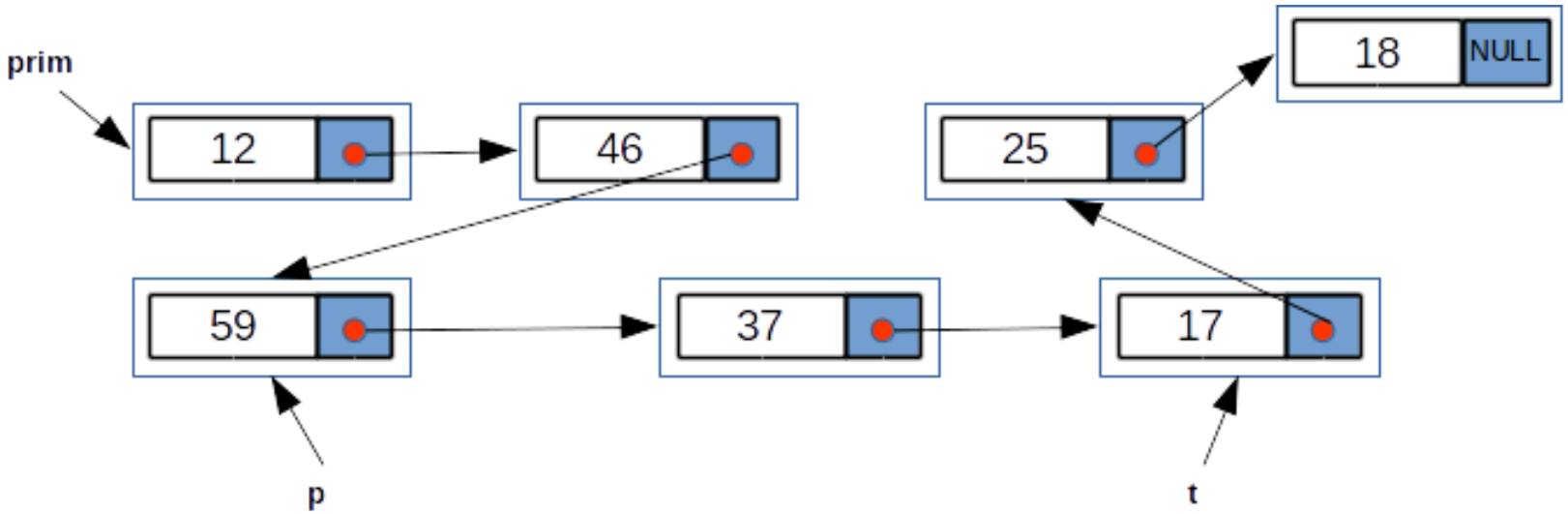
Lista simplu înlănțuită este diferită de alte structuri de date, cum ar fi vectorii sau matricele, deoarece lista poate fi extinsă sau micșorată în funcție de nevoile aplicației.

Operații de bază cu liste

Operaţiile care se pot efectua asupra listelor sunt:

- crearea listei;
- parcurgerea listei;
- inserarea unui nod într-o anumită poziţie (nu neapărat prima);
- căutarea unui nod cu o anumită proprietate;
- furnizarea conţinutului unui nod dintr-o anumită poziţie sau cu o anumită proprietate;
- ștergerea unui nod dintr-o anumită poziţie sau cu o anumită proprietate.

Relații între nodurile unei liste



prim este adresa elementului cu valoarea 12;

prim->info==12

prim->urm->info==46

prim->urm este adresa elemenului cu valoarea 46

prim->urm->urm==p

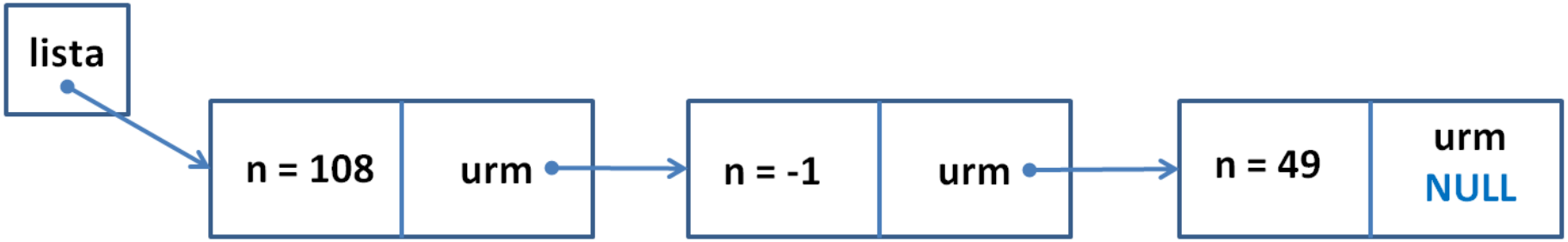
p->info==59

https://www.pbinfo.ro/articole/19576/liste-liniare-simplu-inlantuite-alocate-dinamic

Liste simplu înlănțuite - nodul

```
// un element al listei – un nod
typedef struct elem{
    int info;                   // informatia utila
    struct elem *urm;           // urmatorul nod
    }elem;

elem *lista = NULL;
```



De data trecută – Liste simplu înlănțuite

Liste simplu înlănțuite - continuare

Liste ordonate

Liste dublu înlănțuite

Liste circulare

Liste ordonate

Listele ordonate se construiesc de la bun început ordonate, plecând de la faptul că lista vidă este în mod intrinsec ordonată, iar nodurile ulterioare se inserează astfel încât să nu se strice ordinea deja existentă.

Într-o listă ordonată creşte performanţa operaţiilor de căutare.

Membrul care serveşte la identificarea nodurilor şi asupra căruia operează criteriul de ordonare este numit membrul sau câmpul cheie.

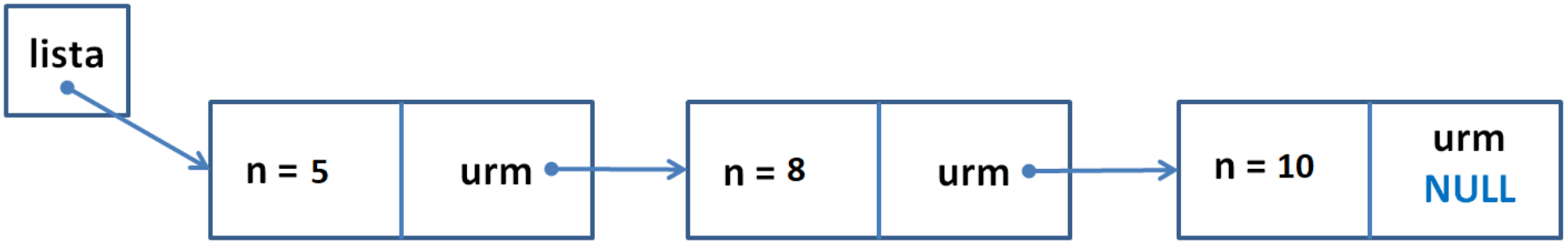
Căutarea unui element într-o listă ordonată

Căutarea unui element cu o cheie dată într-o listă ordonată se face traversând lista atâta timp cât nu s-a ajuns la sfârşitul listei *(p!=NULL)* şi cheile din nodurile curente sunt mai mici decât cheia căutată (*p->data < info*).

În cazul în care nu există în listă un nod cu cheia căutată, în cele mai multe cazuri nu se mai ajunge cu parcurgerea listei până la sfârşitul ei, ci căutarea se opreşte în momentul întâlnirii primului nod care are cheia mai mare decât cheia căutată.

Căutarea unui element într-o listă ordonată

```
elem* cauta_ordonat(nod *lista, int info) {
        elem *p;
        for(p=lista;p!=NULL && p->data < info;p=p->urm);
        if (p!=NULL && p->data == info)
                return p;
        else
                return NULL;
        }
```



Adăugarea unui element într-o listă ordonată

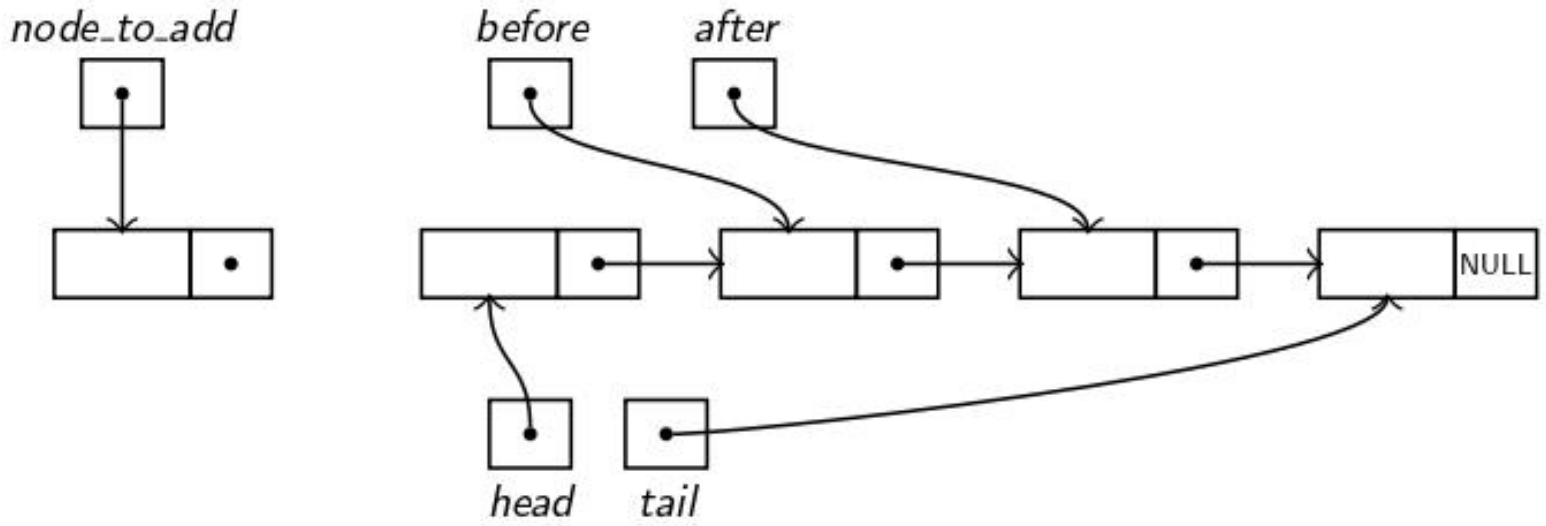
Adăugarea unui nou element la lista aflată în stare ordonată se face astfel încât lista să rămană ordonată şi după adăugarea noului element.

Pentru aceasta se face o căutare a poziţiei de inserare în listă: noul nod va fi inserat înaintea primului nod care are cheia mai mare decât cheia lui.

Se utilizează un pointer curent *p* care avansează în listă, atâta timp cât nodurile curente au chei mai mici decât noua cheie de inserat şi nu s-a ajuns la sfârşitul listei. Dacă *p* s-a oprit pe primul nod cu cheie mai mare decât cheia de inserat, noul nod trebuie inserat înaintea nodului *p*. Este necesar deci accesul şi la nodul anterior lui *p*, pentru a putea reface legăturile.

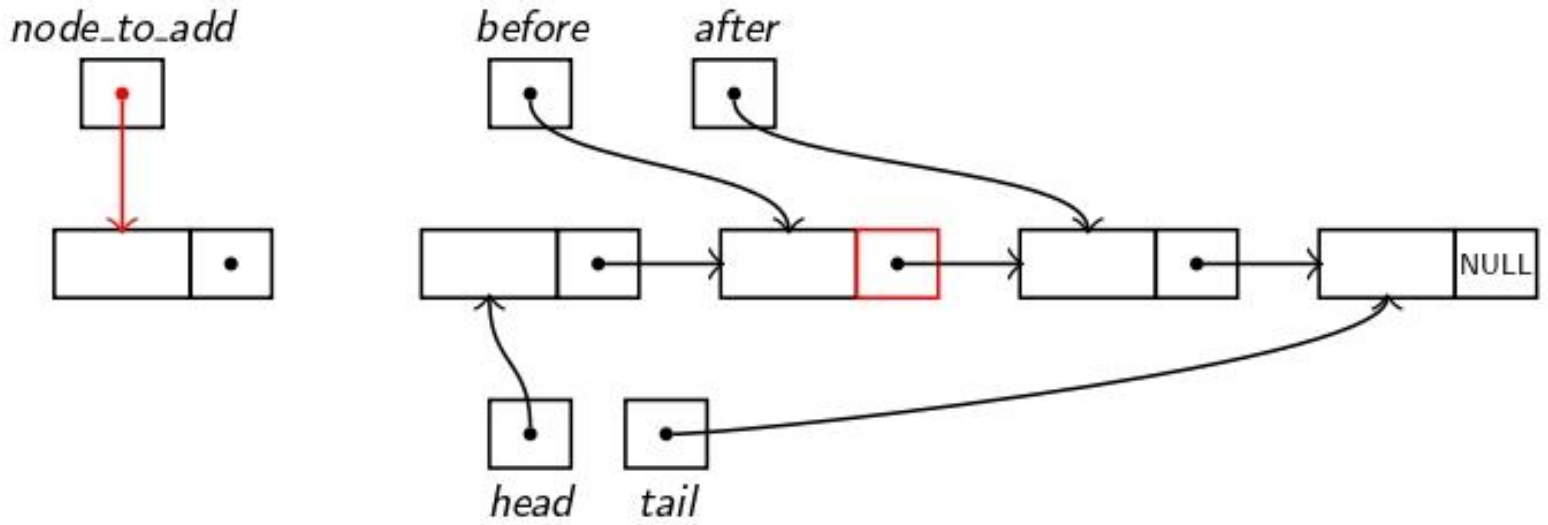
Ordered Lists

Adding a Node to an Ordered List



Ordered Lists

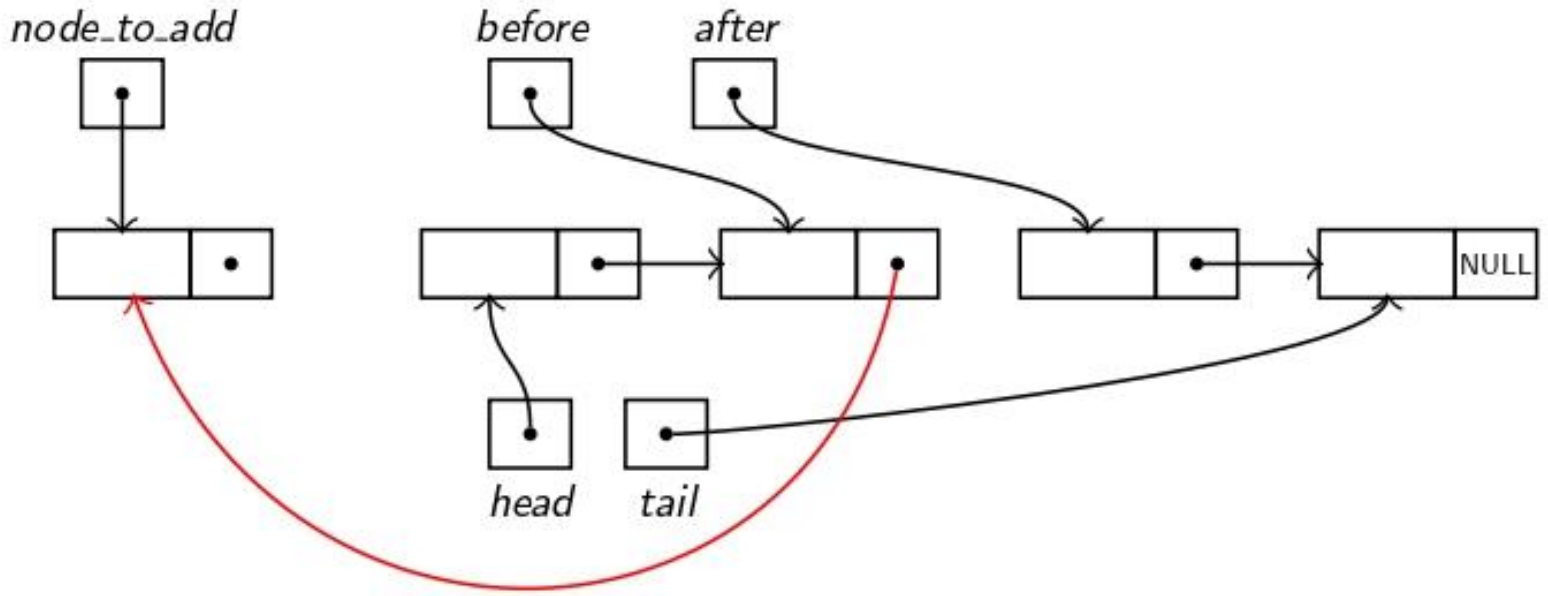
Adding a Node to an Ordered List




```
before->next = node_to_add;
```

Ordered Lists

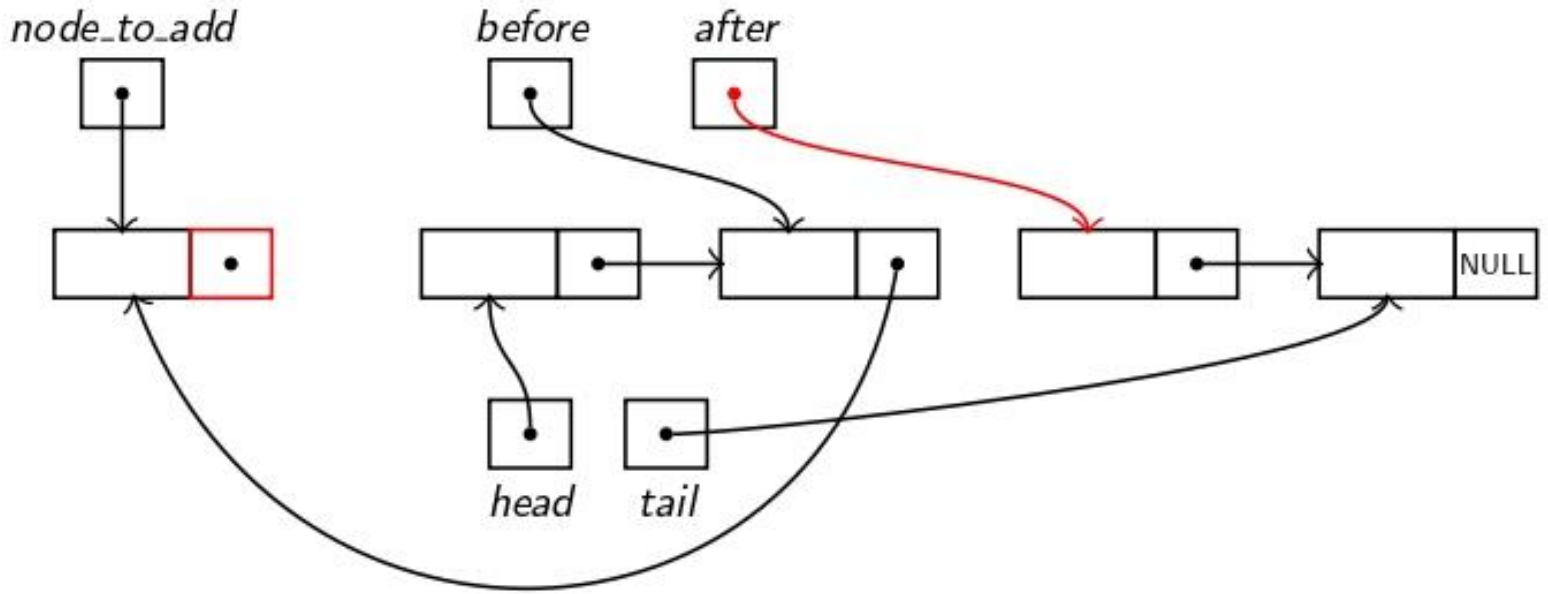
Adding a Node to an Ordered List




```
before->next = node_to_add;
```

Ordered Lists

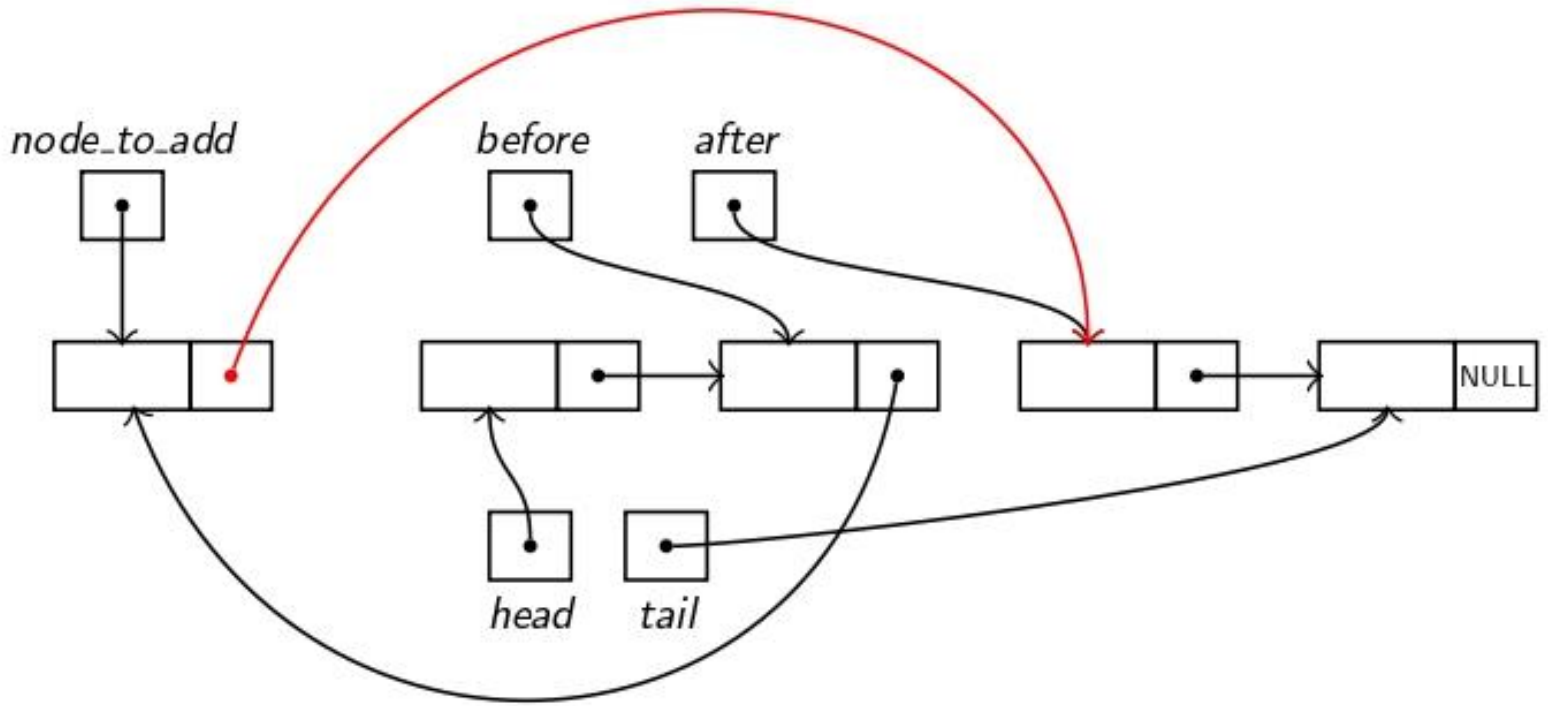
Adding a Node to an Ordered List




```
node_to_add->next = after;
```

Ordered Lists

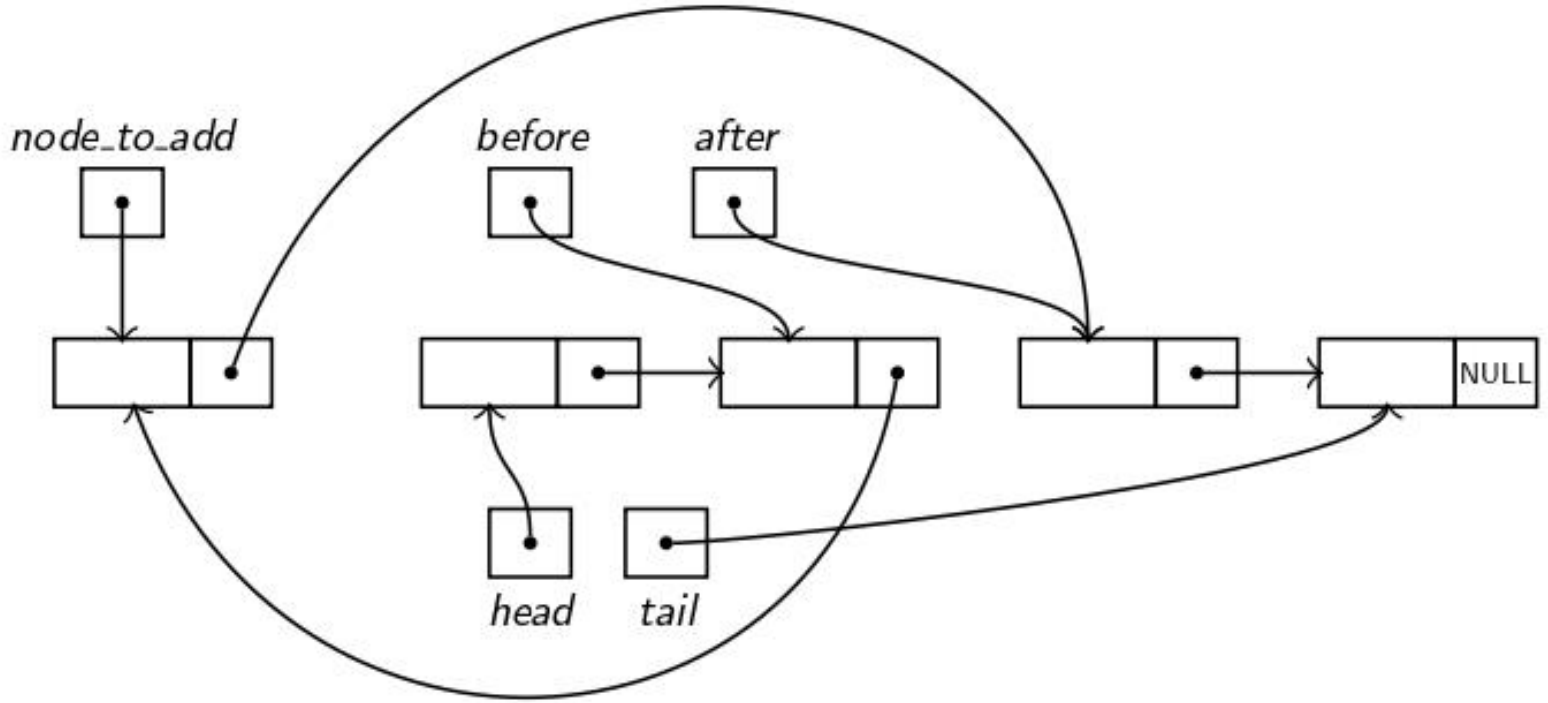
Adding a Node to an Ordered List




```
node_to_add->next = after;
```

Ordered Lists

Adding a Node to an Ordered List



Adaugarea unui element într-o listă ordonată

```
elem * adauga_ordonat(elem *lista, int n) {
elem *nou;
nou=nod_nou(n,NULL);
for(q1=q2=lista; q1!=NULL && q1->info<n ; q2=q1, q1=q1->urm);
if (q1!=NULL && q1->info == n) {
        printf("Eroare: %d apare deja in lista\n", n);
        return; }
else
        if (q1!=q2) {
                /* inserarea nu se face la inceput */
                q2->urm=nou;
                nou->urm=q1; }
        else {
                /* inserarea se face la inceputul listei */
                nou->urm=lista;
                lista=nou; }
return lista;
}
```

De data trecută – Liste simplu înlănțuite

Liste simplu înlănțuite - continuare

Liste ordonate

Liste dublu înlănțuite

Liste circulare

Liste liniare dublu înlănțuite

Pe lângă listele simplu înlănțuite, în C putem folosi și liste liniare dublu înlănțuite. Acestea sunt similare cu listele simplu înlănțuite, dar fiecare nod are și un pointer către nodul anterior, permițând parcurgerea listei în ambele direcții.

Adăugarea unui nod nou

- Adăugarea unui nod într-o listă liniară dublu înlănțuită poate fi realizată în trei cazuri: la începutul listei, la sfârșitul listei și în interiorul listei.
- În funcție de locul în care se adaugă trebuie actualizați mai mulți pointeri: prim, ultim, prev al nodului următor, urm al nodului anterior, prev al nodului nou, urm al nodului nou

Adăugarea unui nod nou

Folosirea listelor liniare simplu înlănțuite sau a listelor liniare dublu înlănțuite depinde de situația specifică în care sunt utilizate.

În general, lista simplu înlănțuită este mai ușor de implementat și consumă mai puțină memorie decât lista liniară dublu înlănțuită.

Pe de altă parte, lista liniară dublu înlănțuită oferă acces la nodurile anterioare, permițând parcurgerea listei în ambele direcții și făcând mai ușor inserarea sau ștergerea unui nod în interiorul listei.

Avantaje – liste dublu înlănțuite

Avantajul major al listelor dublu înlănțuite față de cele simplu înlănțuite este faptul că, dacă avem un pointer la un element din listă, toate operațiile relative la acel element (ștergere, inserare înainte de el, inserare după el) au complexitatea O(ct).

La o listă simplu înlănțuită, ștergerea unui element către care avem un pointer sau inserarea înainte de el au complexitate O(n), deoarece prima oară trebuie parcursă lista pentru a afla predecesorul acelui element.

Dezavantaje – liste dublu înlănțuite

Dezavantajele listelor dublu înlănțuite sunt consumul mai mare de memorie, deoarece fiecare element stochează doi pointeri și complexitatea mai mare a operațiilor, deoarece trebuie menținută sincronizarea mai multor pointeri.

Din aceste motive, de obicei listele dublu înlănțuite se folosesc în cazul aplicațiilor care au colecții cu un număr mediu sau mare de elemente (unde complexitatea O(ct) este un avantaj semnificativ față de O(n)) și în care sunt necesare multe operații de inserare sau ștergere.

Avantaje - Dezavantaje

Putem lua în considerare cazul unui editor de text în care este deschis un document mare.

- Dacă documentul este implementat cu vectori, pentru fiecare ștergere sau inserare trebuie să mutăm zone de memorie de ordinul MB, ceea ce este foarte lent.
- Dacă documentul este implementat cu liste simplu înlănțuite și ștergerea sau inserarea au loc înainte de elementul curent, trebuie prima oară să parcurgem lista pana la elementul precedent celui în care se face modificarea, operație de asemenea lentă.
- La implementarea cu liste dublu înlănțuite, ștergerea sau inserarea se poate face direct în orice punct dorim, fără a avea nevoie de niciun fel de operații suplimentare și fără a fi necesară reorganizarea elementelor deja existente.

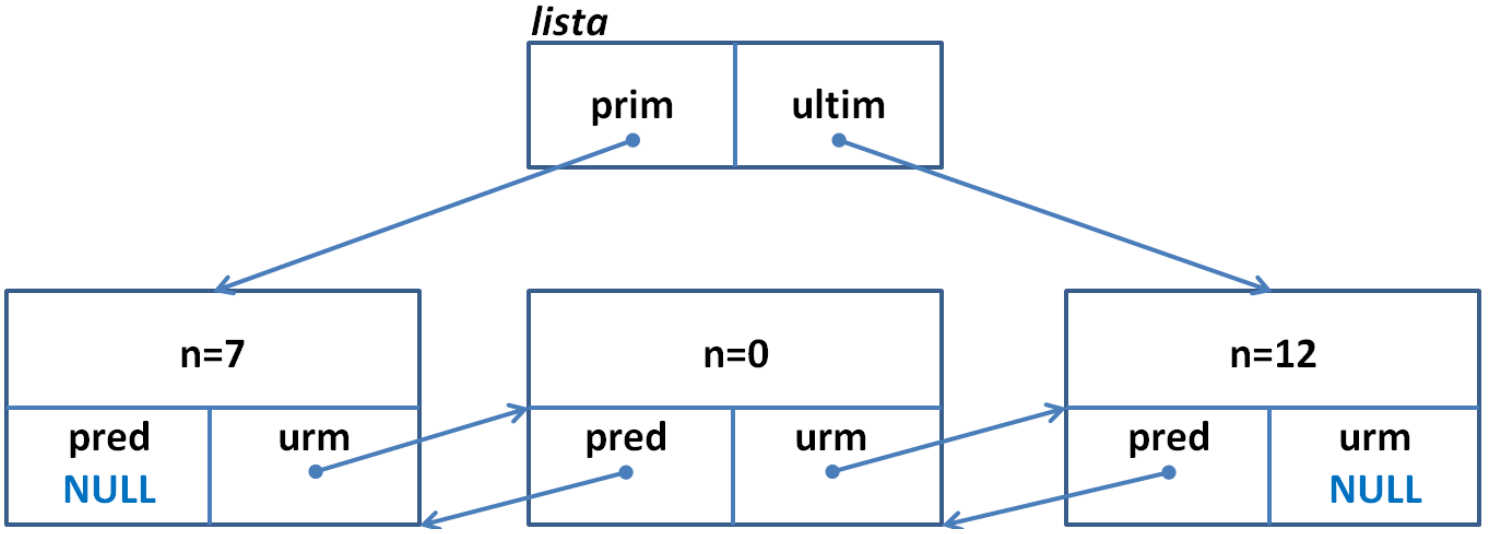
Avantaje - Dezavantaje

Pentru liste scurte, de doar câteva elemente, timpul necesar pentru iterarea unei liste simplu înlănțuite este compensat de timpul mai mic necesar pentru alte operații, deoarece acestea sunt mai simple decât la listele dublu înlănțuite.

Dacă în schimb lista trece de un anumit număr de elemente și operațiile de ștergere sau inserare înainte de elementul curent sunt frecvente, atunci deja diferența între cele două tipuri de date devine semnificativă.

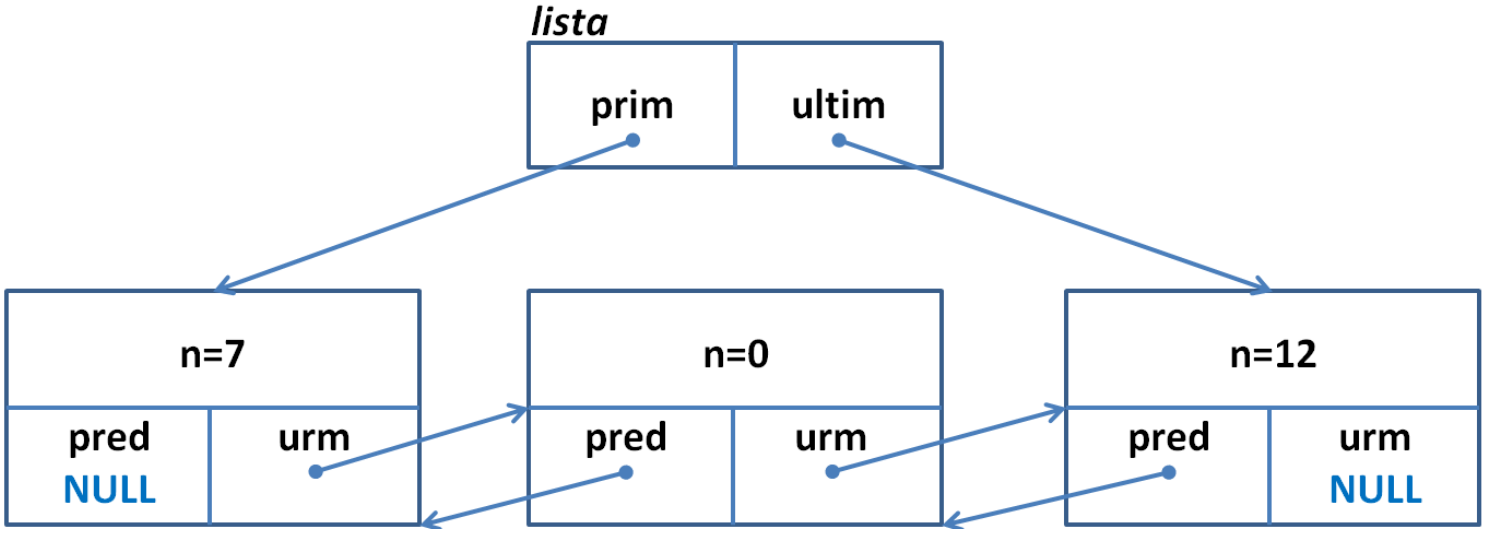
Structura

O listă dublu înlănțuită păstrează pentru fiecare element atât un pointer către următorul element din listă (***urm*** - următor), cât și un pointer către elementul anterior (***pred*** - predecesor).



Structura

```
typedef struct nod{
    int data;
    struct nod *pred;
    struct nod *urm;
    }nod;
```

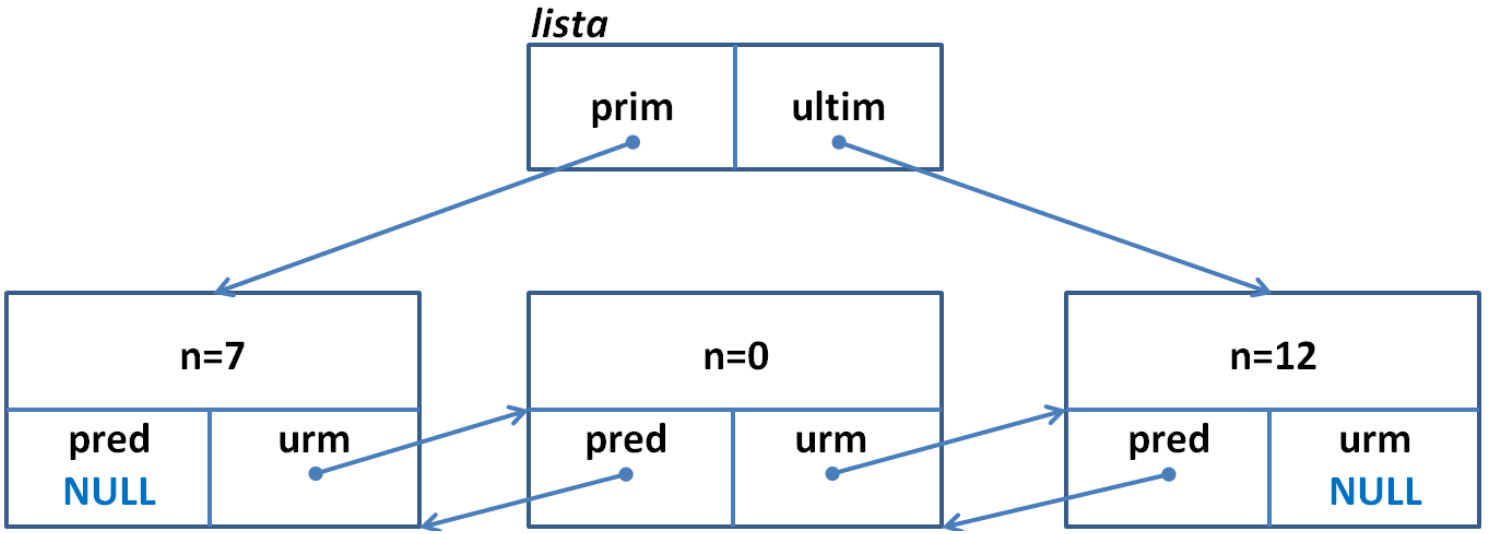


Crearea unui nod nou

```
nod *nod_nou(int info)
{
   nod *nou=(nod*)malloc(sizeof(nod));
   if(!nou){
      printf("memorie insuficienta");
      exit(EXIT_FAILURE);
      }
   nou->data=info;
   return nou;
}
```

Nodurile prim și ultim

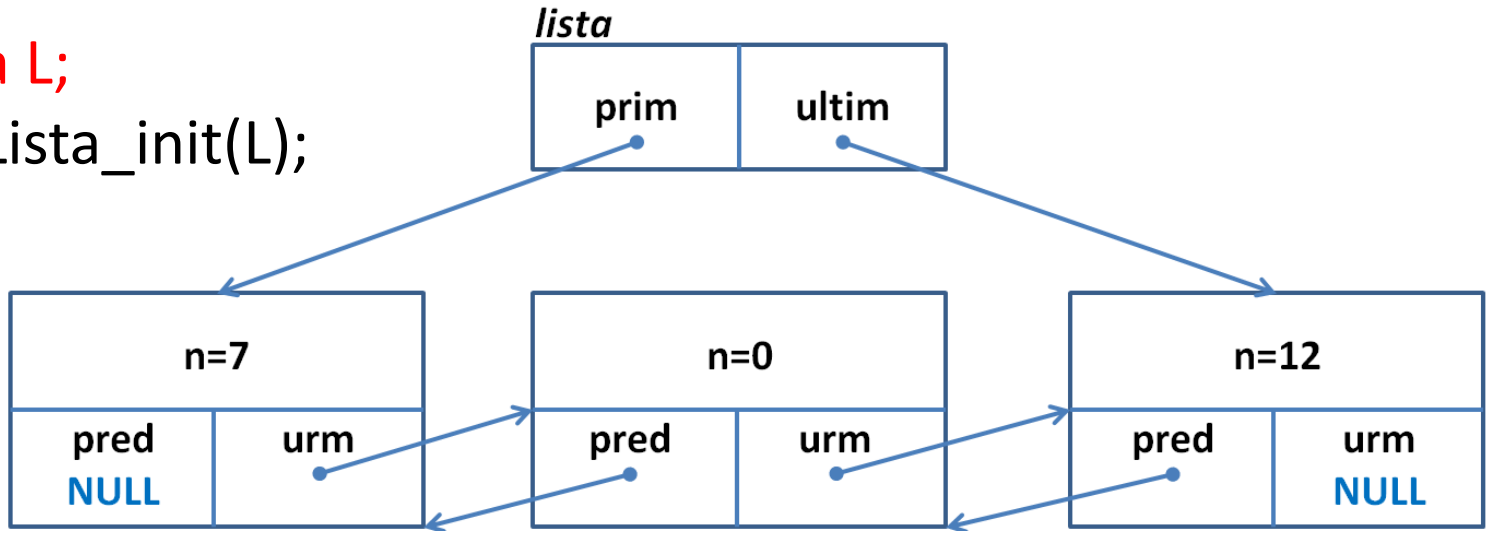
```
typedef struct{
    nod *prim;      // primul nod din lista
    nod *ultim;     // ultimul nod din lista
    }Lista;
```



Nodurile prim și ultim

```
// initializare Lista noua
Lista Lista_init(Lista L)
{
   L.prim=L.ultim=NULL;
   return L;
}
```

```
Lista L;
L = Lista_init(L);
```



Adaugarea unui nod la finalul listei

```
Lista Lista_adauga(Lista L,nod *c)
{
   c->pred=L.ultim;        // predecesorul nodului este ultimul nod din lista
   if(L.ultim!=NULL){      // daca mai sunt si alte cuvinte in lista
      L.ultim->urm=c;      // ultimul nod din lista va pointa catre noul nod
      }
   else{                   // altfel, daca c este primul nod din lista
      L.prim=c;                              // seteaza si inceputul listei la el
      }
   L.ultim=c;                          // seteaza sfarsitul listei pe noul nod
   c->urm=NULL;           // dupa nodul introdus nu mai urmeaza niciun nod

   return L;
}
```

Căutarea unui nod în listă

```
nod *Lista_cauta(Lista L,int info)
{
    nod *c;
    for(c=L.prim;c!=NULL;c=c->urm){
        if(c->data == info)
            return c;
        }
    return NULL;
}
```

Ștergerea unui nod din listă

```
Lista Lista_sterge(Lista L,nod *c)
{
   if(c->pred != NULL){                    // nodul nu este primul in Lista
      c->pred->urm=c->urm;                 // urm al pred lui c va pointa la nodul de dupa c
      }else{                               // nodul este primul in Lista
      L.prim=c->urm;          // seteaza inceputul listei pe urmatorul nod de dupa c
      }
   if(c->urm != NULL){                     // nodul nu este ultimul din Lista
      c->urm->pred=c->pred;                // campul pred al nodului de dupa c va
pointa la nodul de dinainte de c
      }else{                               // nodul este ultimul din Lista
      L.ultim=c->pred;                     // seteaza sfarsitul listei pe predecesorul lui c
      }
   free(c);
   return L;
}
```

Eliberarea memoriei folosite de listă

```
Lista Lista_elibereaza(Lista L)
{
   nod *c,*urm;
   for(c=L.prim;c!=NULL;c=urm){
      urm=c->urm;
      free(c);
      }
   L = Lista_init(L);
   return L;
}
```

Apelarea funcțiilor parcurse

```
int main()
{
   Lista p;
   int n, info, op;
   nod *c;
   p = Lista_init(p);
   do{
      printf("1 - Lista noua\n");
      printf("2 - afisare\n");
      printf("3 - stergere nod\n");
      printf("4 - iesire\n");
      printf("optiune: ");scanf("%d",&op);
      switch(op){
case 1:
            p = Lista_elibereaza(p);
            printf("Cate noduri are lista?\n");
            scanf("%d",&n);
            for(int i=0;i<n;i++){
               printf("el(%d) = ",i);
               scanf("%d",&info);
               nod *c=nod_nou(info);
               p = Lista_adauga(p,c);
               }
            break;
case 2:
            for(c=p.prim;c!=NULL;c=c->urm)
                printf("%d ",c->data);
            printf("\n");
            break;
case 3:
            printf("nod de sters:");
            scanf("%d",&info);
            c=Lista_cauta(p,info);
            if(c){
               p = Lista_sterge(p,c);
               }else{
               printf("nodul \"%d\" nu e in Lista\n",info);
               }
            break;
case 4:break;
default:printf("optiune invalida");
          }
       }while(op!=4);
   return 0;
}
```

De data trecută – Liste simplu înlănțuite

Liste simplu înlănțuite - continuare

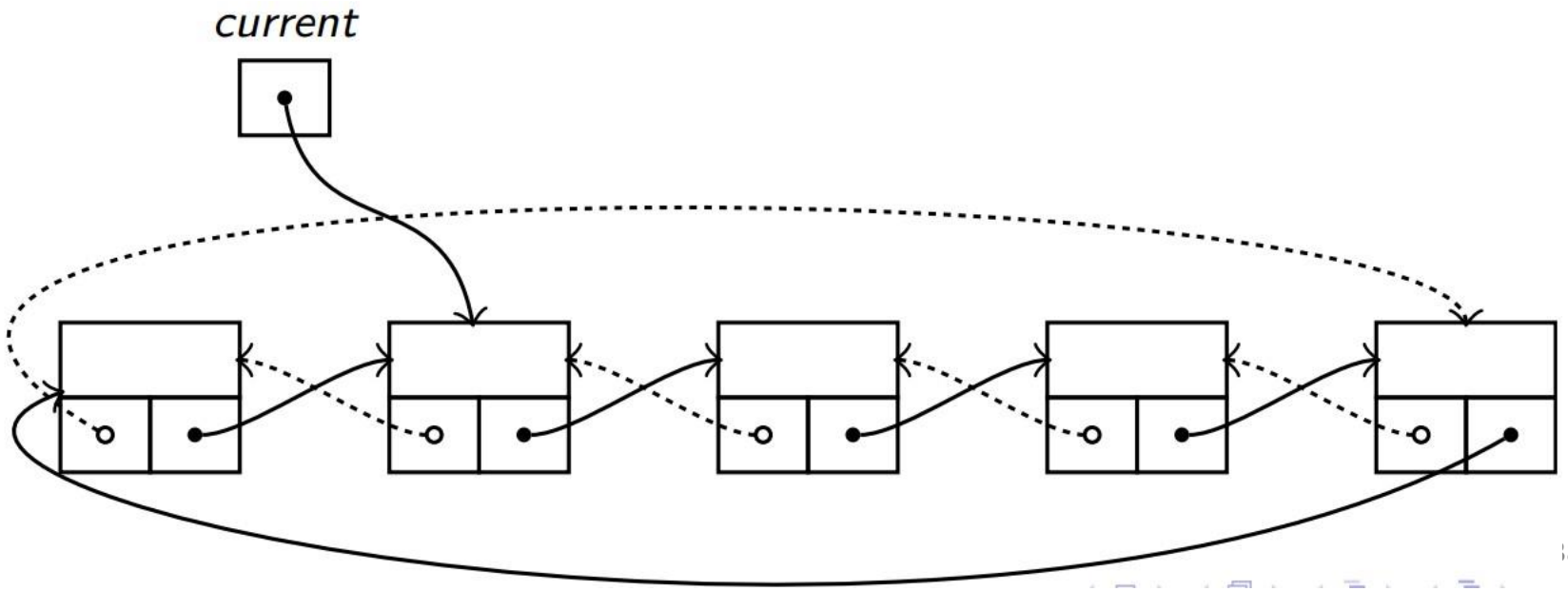
Liste ordonate

Liste dublu înlănțuite

Liste circulare

Liste circulare

Listele dublu înlănțuite se pot organiza uşor ca liste circulare (în care nu există nici un element prim sau ultim ci doar un pointer curent spre orice nod al listei și în care nu avem NULL la nici un pointer pred sau urm al nodurilor).



Circular Doubly Linked Lists

```
typedef struct _node {
  int info;
  struct _node *prev;
  struct _node *next;
} node;
```

- If we put a dummy sentinel node into the list, so that it never gets empty, then all operations upon the list are simplified (there will be no special cases to handle).

```
void init_list(node ** start)
{
  *start = create_node(-1);      /* Create the sentinel
                                  * node. */
  /*
   * Link both pointers of the sentinel node back to
   * itself (circular doubly linked list with one node).
   */
  (*start)->prev = (*start)->next = *start;
}
```

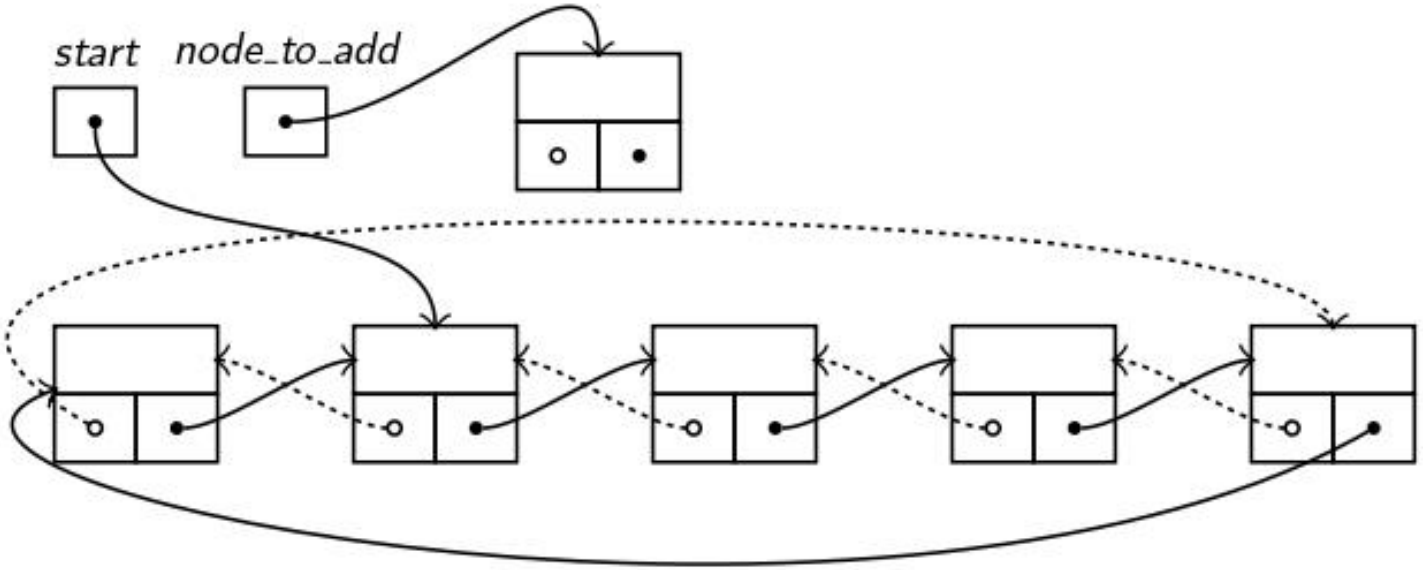
Circular Doubly Linked Lists

Visiting the Nodes in a Circular Doubly Linked List

```
void print_list(node * start)
{
  node *p = start->next;          /* Start from right after
                                   * the sentinel. */
  /*
   * If the next node after the sentinel is the sentinel,
   * then the list is actually empty.
   */
  if (p == start)
    printf("The list is empty.\n");
  else {
    /*
     * Loop while the sentinel is not reached again.
     */
    while (p != start) {
      printf("%d ", p->info);
      p = p->next;
    }
    printf("\n");
  }
}
```

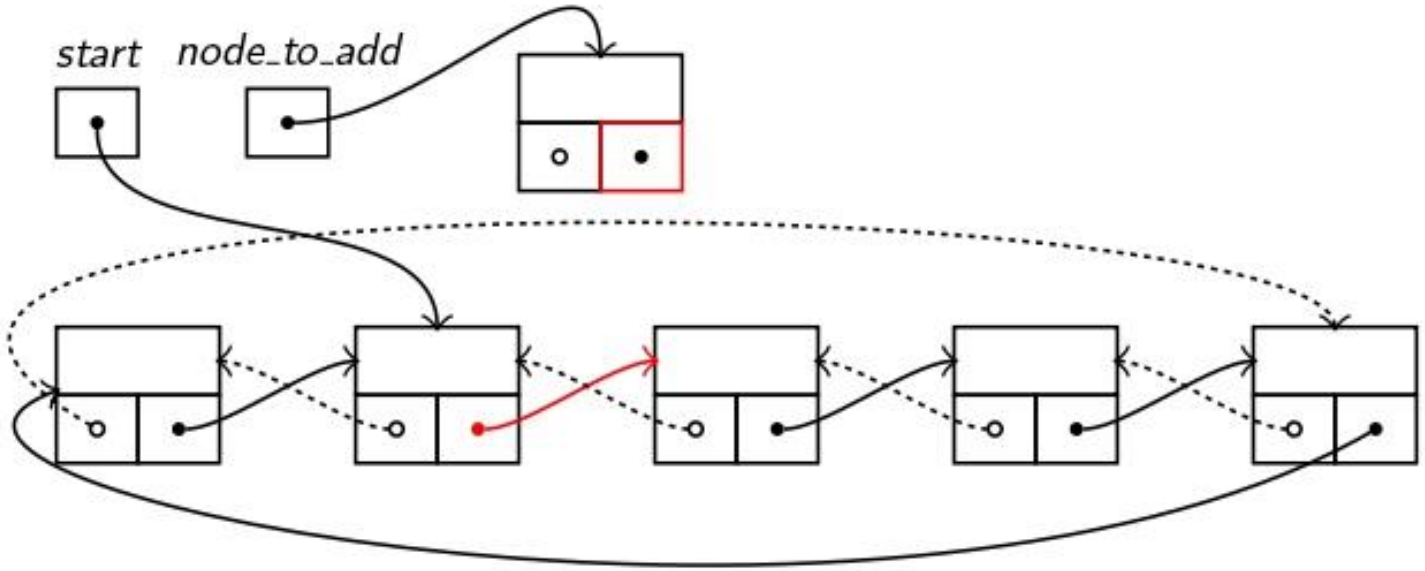
Doubly Linked Lists

- Adding a node to a doubly linked circular list



Doubly Linked Lists

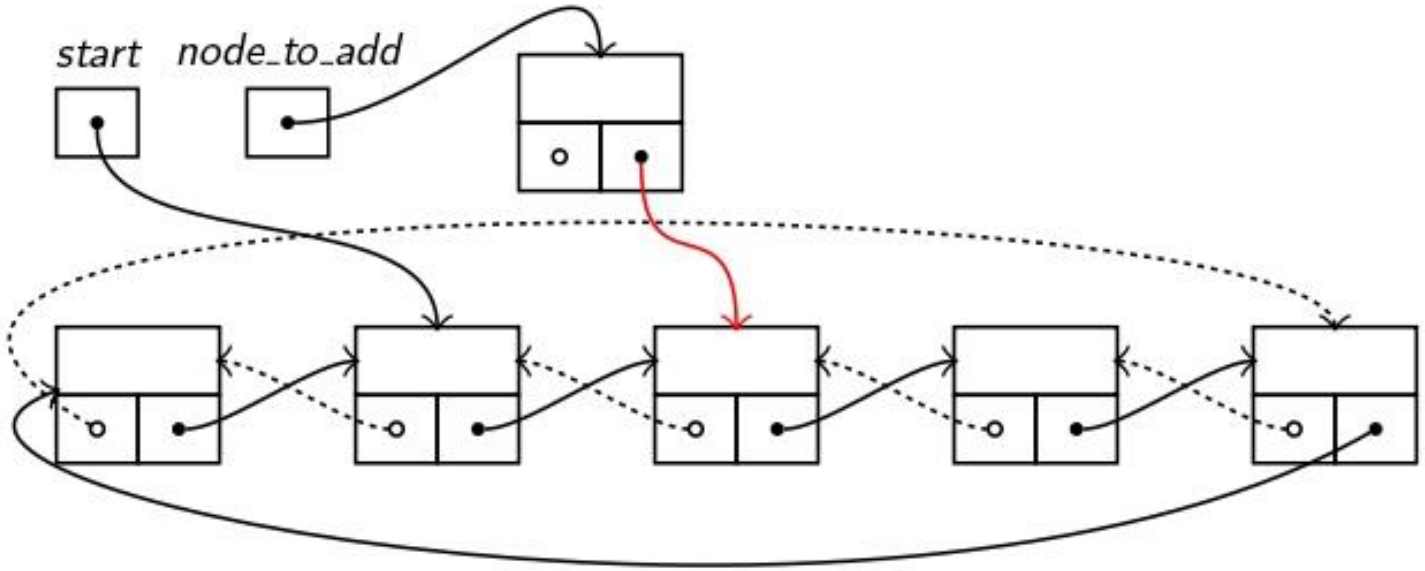
- Adding a node to a doubly linked circular list




```
node_to_add->next = start->next;
```

Doubly Linked Lists

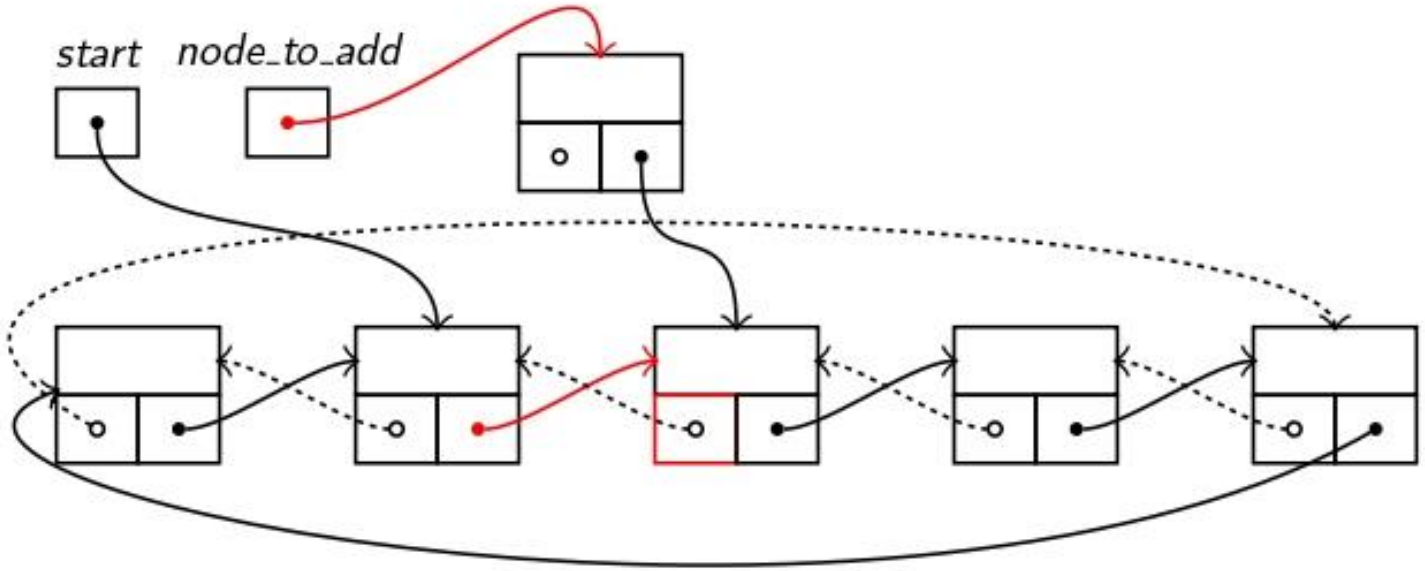
- Adding a node to a doubly linked circular list




```
node_to_add->next = start->next;
```

Doubly Linked Lists

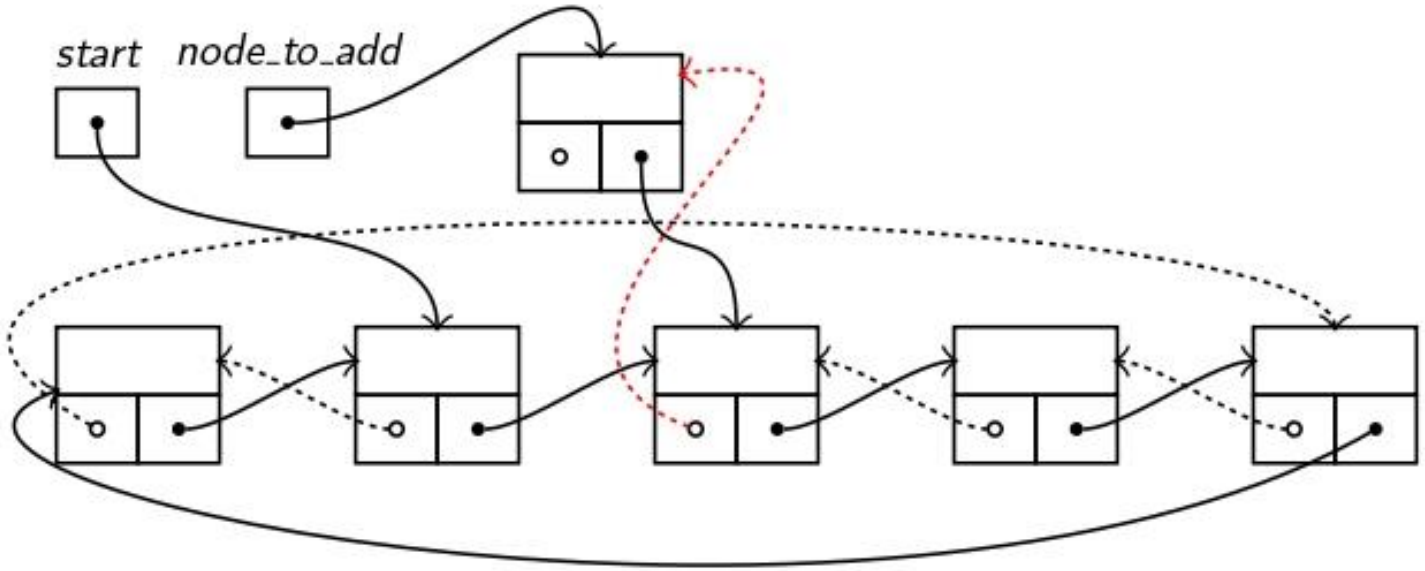
- Adding a node to a doubly linked circular list




```
start->next->prev = node_to_add;
```

Doubly Linked Lists

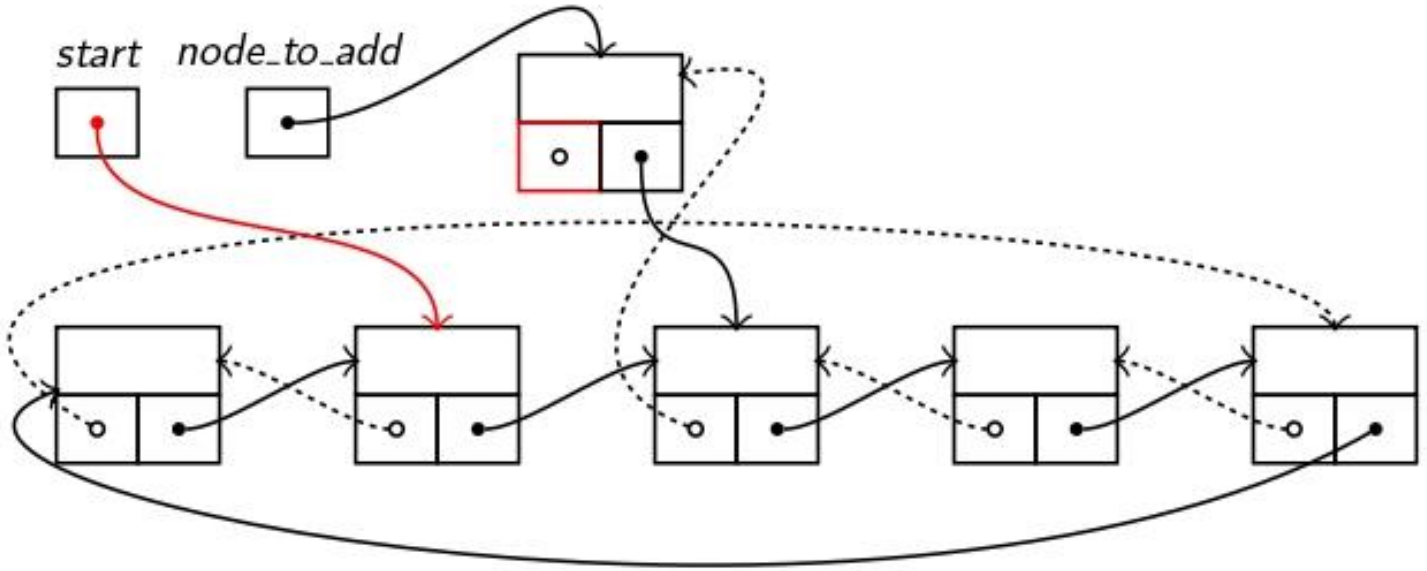
- Adding a node to a doubly linked circular list




```
start->next->prev = node_to_add;
```

Doubly Linked Lists

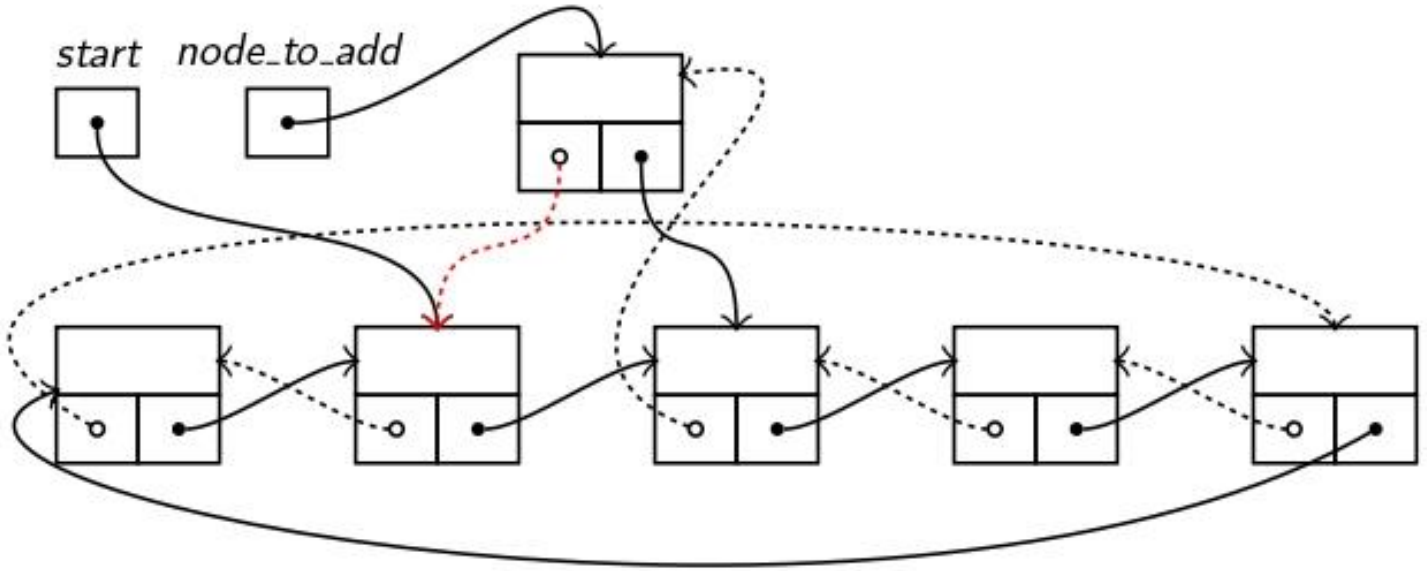
- Adding a node to a doubly linked circular list




```
node_to_add->prev = start;
```

Doubly Linked Lists

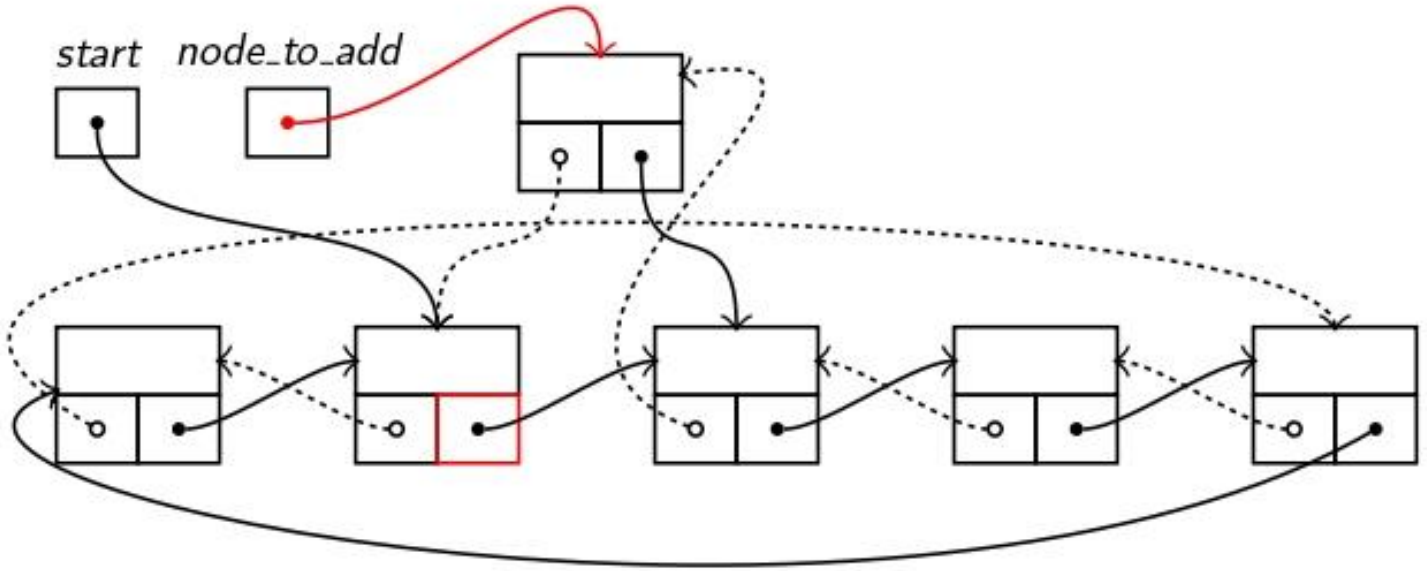
- Adding a node to a doubly linked circular list




```
node_to_add->prev = start;
```

Doubly Linked Lists

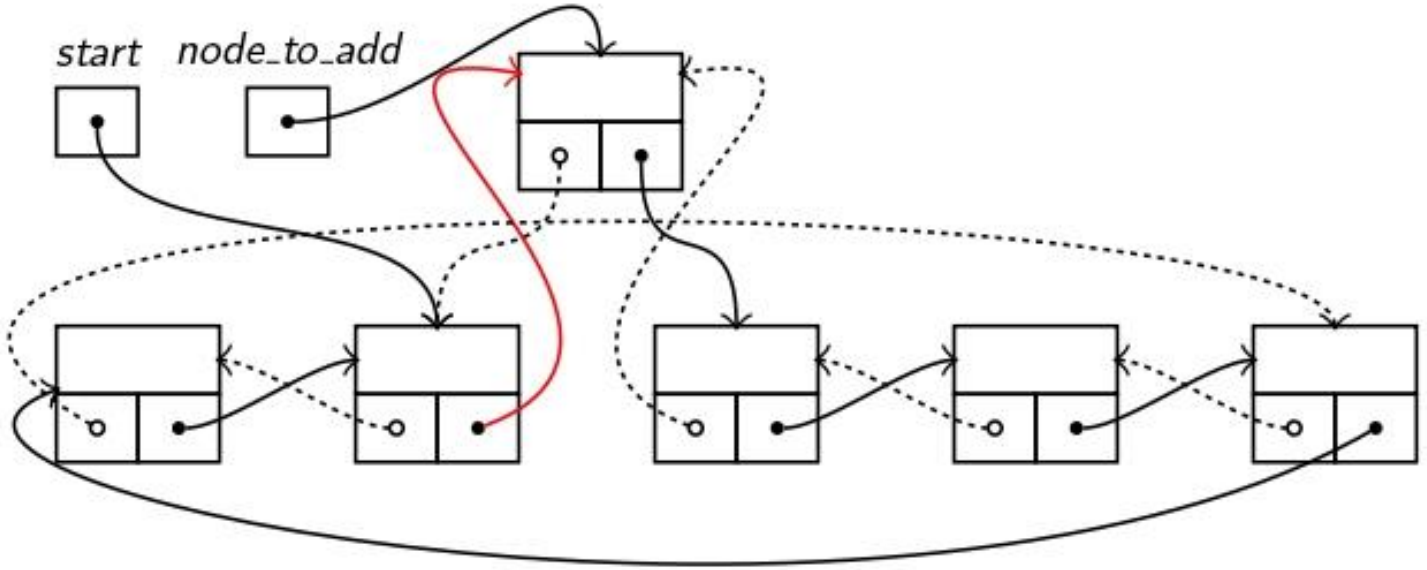
- Adding a node to a doubly linked circular list




```
start->next = node_to_add;
```

Doubly Linked Lists

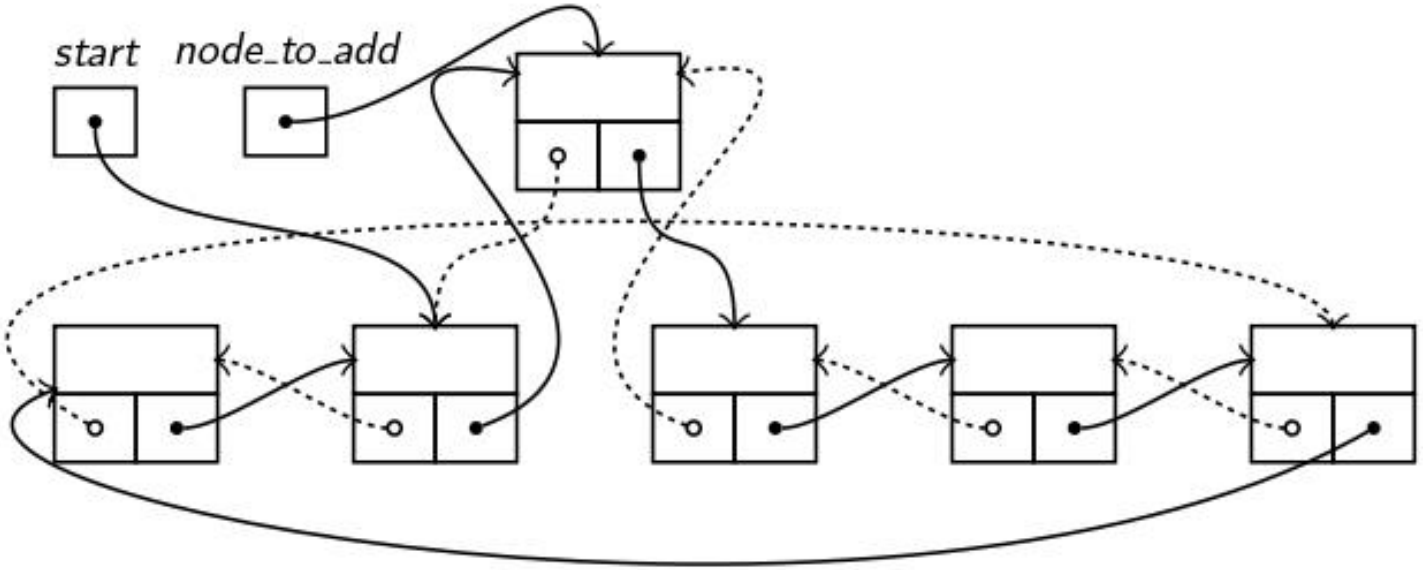
- Adding a node to a doubly linked circular list




```
start->next = node_to_add;
```

Doubly Linked Lists

- Adding a node to a doubly linked circular list



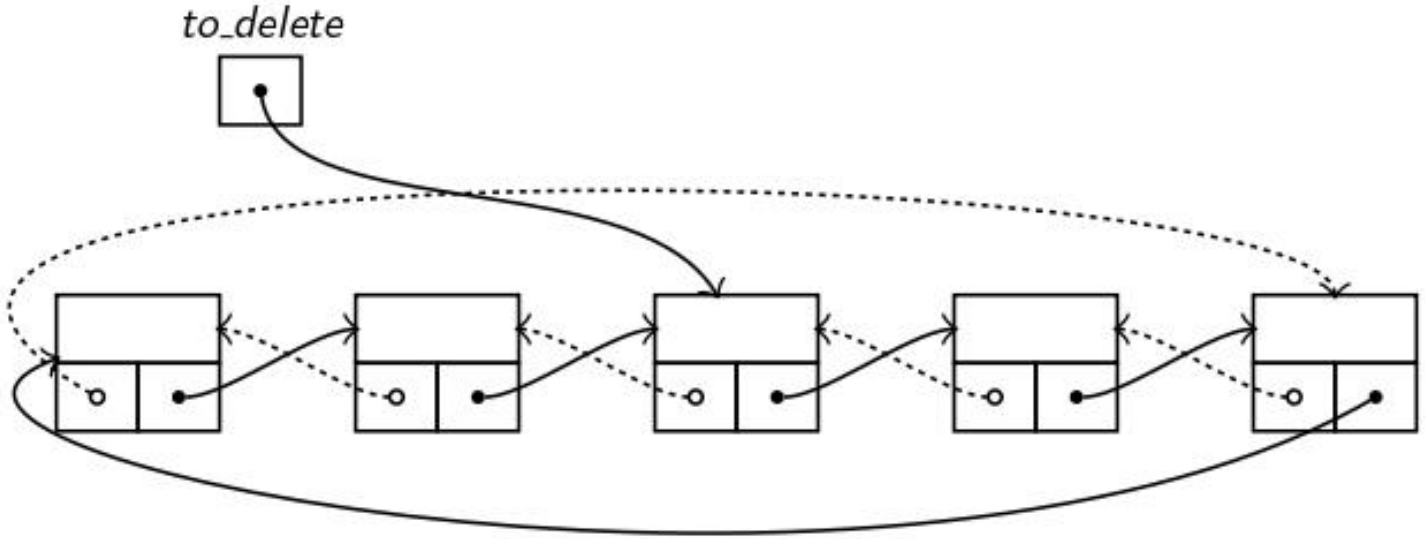
Circular Doubly Linked Lists

Adding a Node to a Circular Doubly Linked List

```
void add_node(node * node_to_add, node * start)
{
  /*
   * Make the double link between the new node and the
   * node after the sentinel
   */
  node_to_add->next = start->next;
  start->next->prev = node_to_add;
  /*
   * Make the double link between the new node and the
   * sentinel.
   */
  node_to_add->prev = start;
  start->next = node_to_add;
}
```

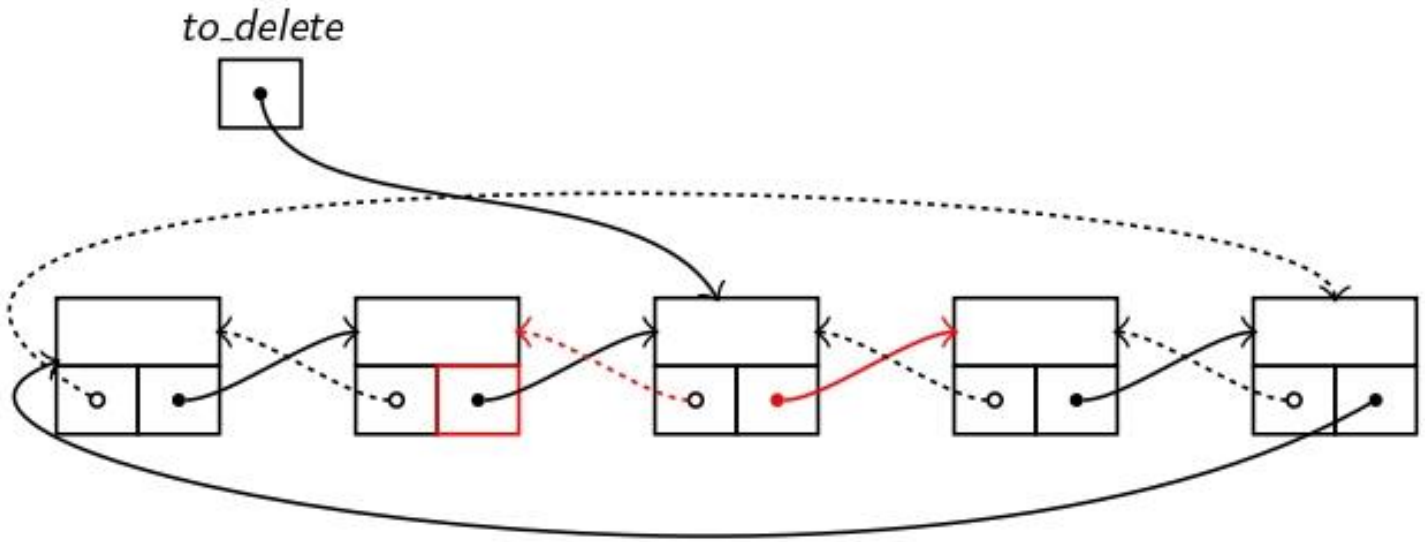
Doubly Linked Lists

- Deleting a node from a doubly linked circular list



Doubly Linked Lists

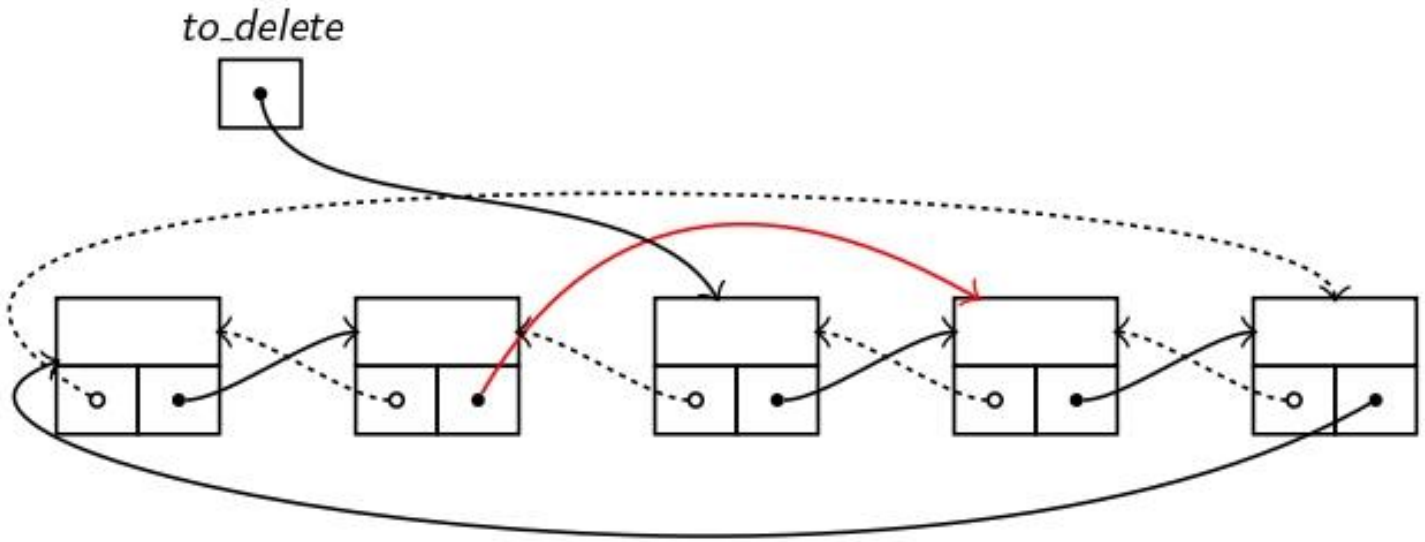
- Deleting a node from a doubly linked circular list




```
to_delete->prev->next = to_delete->next;
```

Doubly Linked Lists

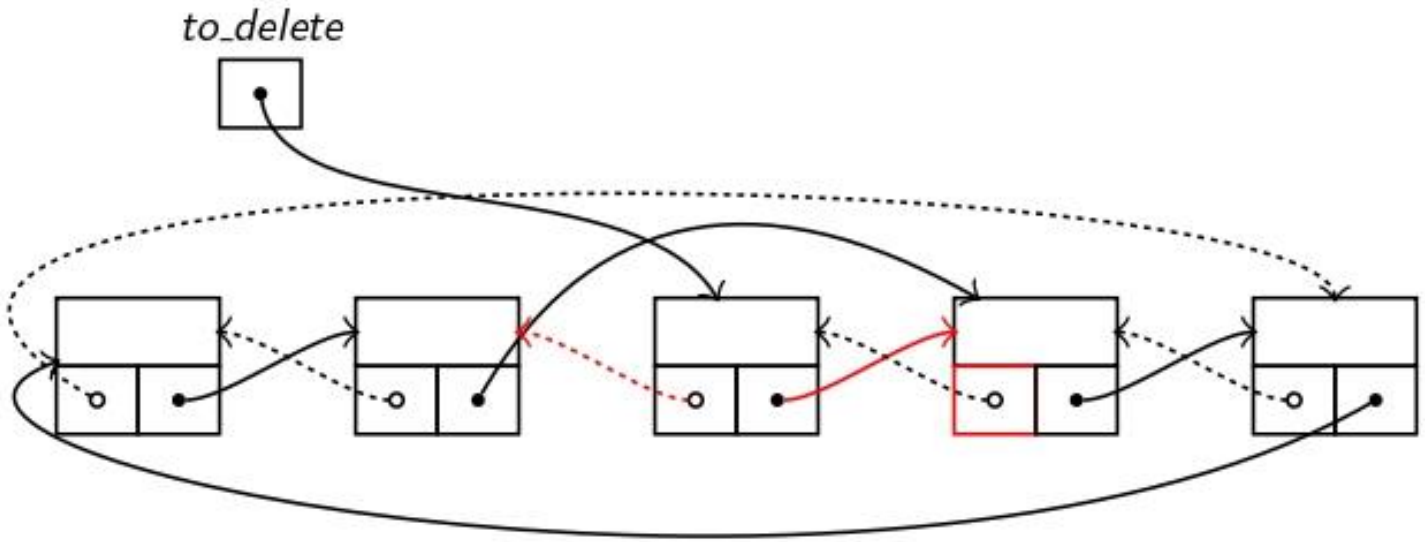
- Deleting a node from a doubly linked circular list




```
to_delete->prev->next = to_delete->next;
```

Doubly Linked Lists

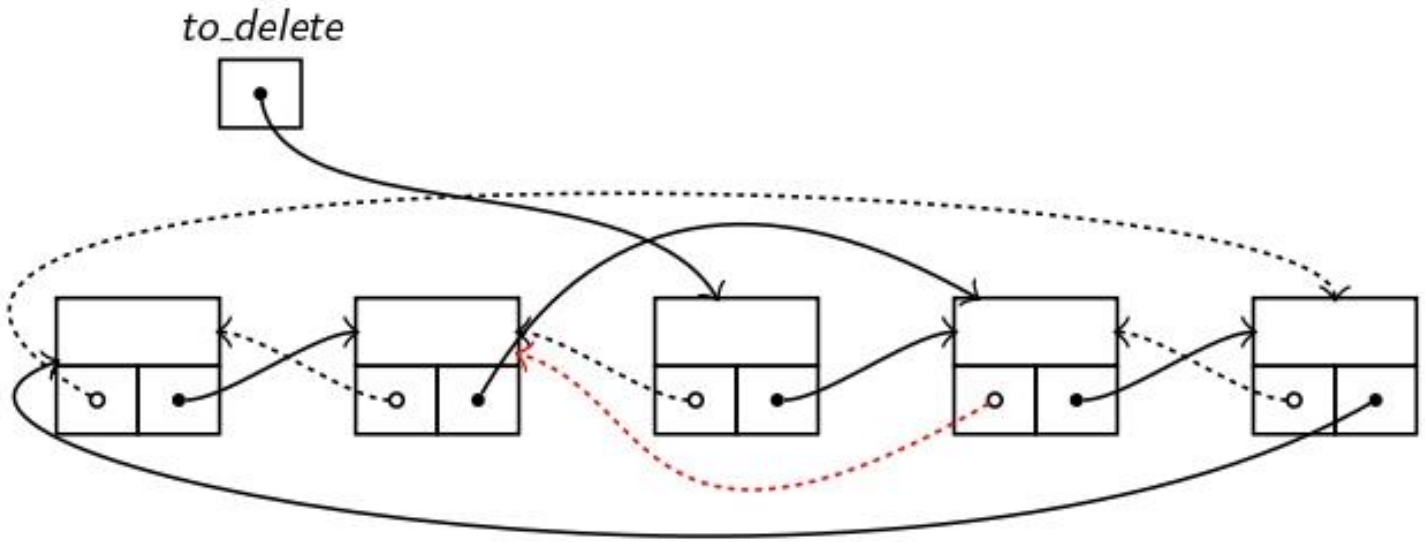
- Deleting a node from a doubly linked circular list




```
to_delete->next->prev = to_delete->prev;
```

Doubly Linked Lists

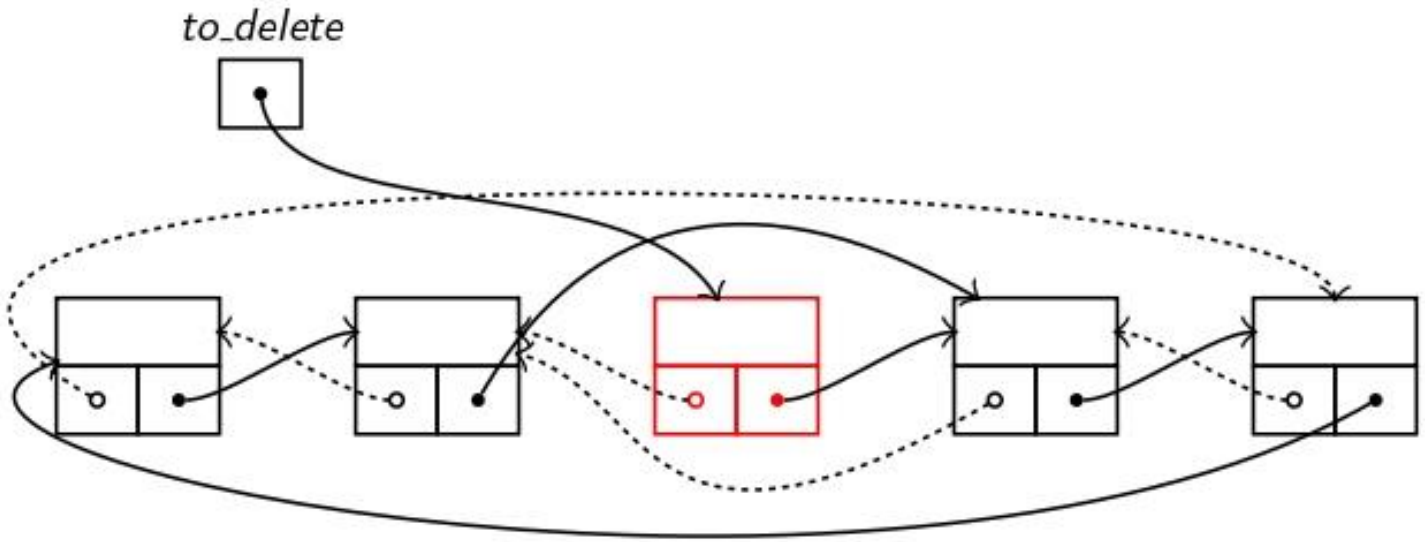
- Deleting a node from a doubly linked circular list




```
to_delete->next->prev = to_delete->prev;
```

Doubly Linked Lists

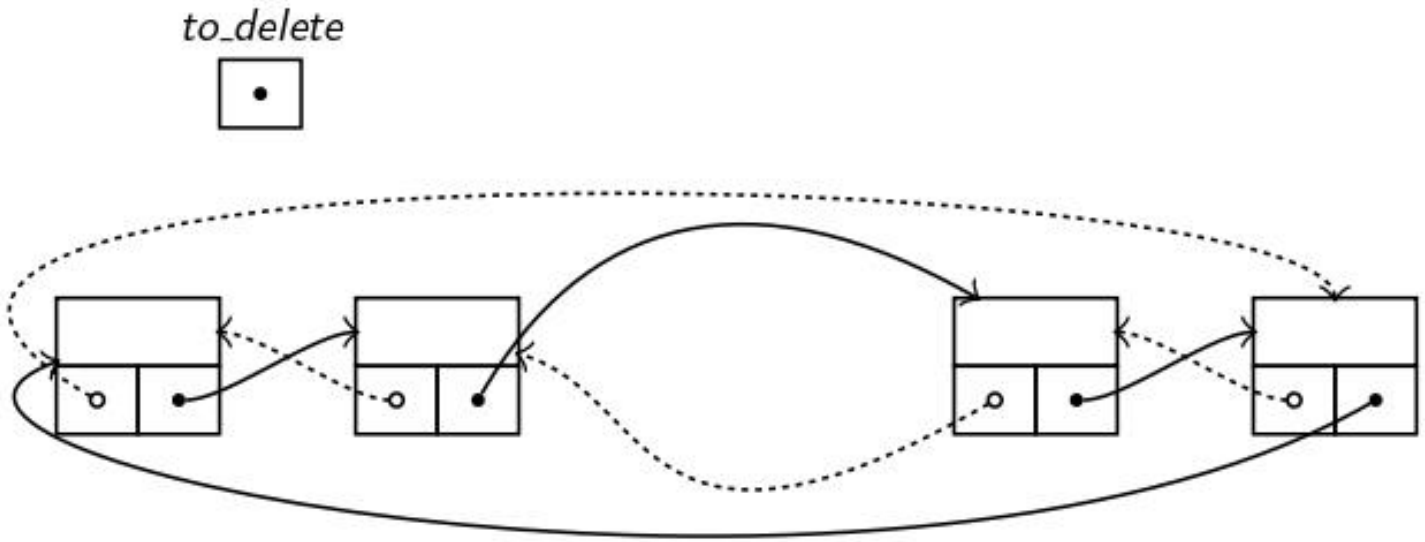
- Deleting a node from a doubly linked circular list




```
free(to_delete);
```

Doubly Linked Lists

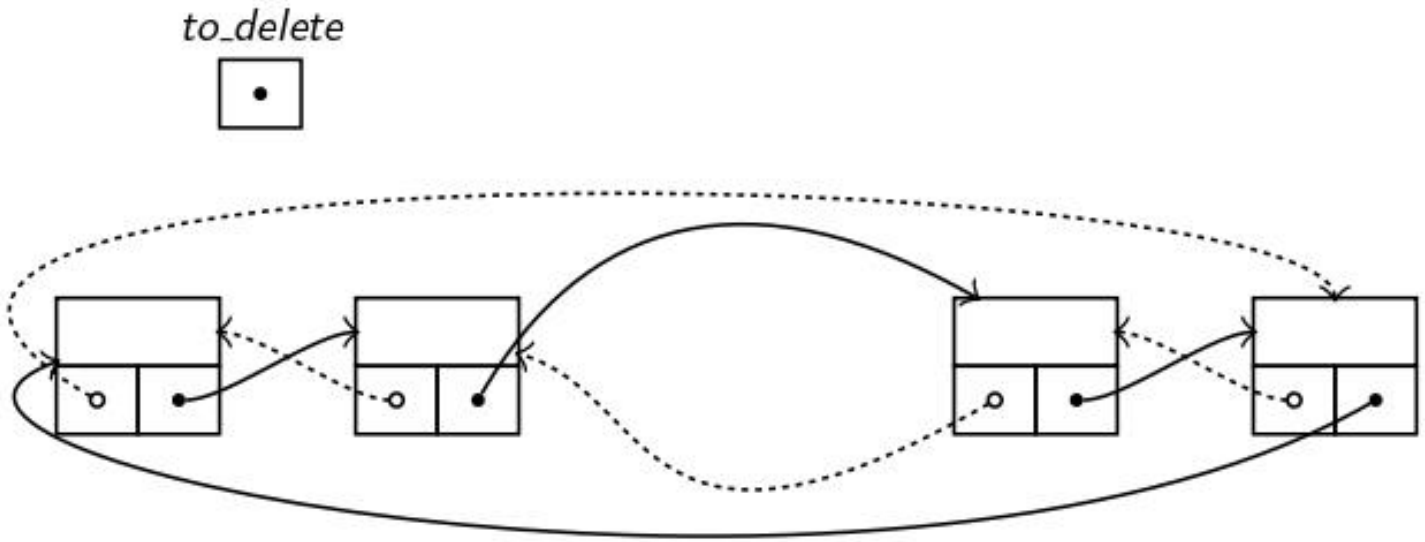
- Deleting a node from a doubly linked circular list




```
free(to_delete);
```

Doubly Linked Lists

- Deleting a node from a doubly linked circular list



Circular Doubly Linked Lists

Deleting a Node from a Circular Doubly Linked List

```
void delete(int info, node * start)
{
  /*
   * Search for the info in the list.
   */
  node *to_delete = search(info, start);
  if (to_delete) {
    /*
     * If a node was found, then just link its neighbors
     * one to another.
     */
    to_delete->prev->next = to_delete->next;
    to_delete->next->prev = to_delete->prev;
    /*
     * And, of course, free the memory used by the node.
     */
    free(to_delete);
  }
}
```

Vă mulțumesc!