# 2. Liste ÎnlĂnŢuite

Liste întâlnim şi utilizăm în fiecare zi: la şcoală profesorul are câte o listă cu elevii fiecărei clase – catalogul; la restaurant găsim o listă cu produsele oferite şi preţurile lor – meniul; la o firmă de calculatoare vom primi o listă de componente şi preţuri – oferta; ş.a.m.d. Prin urmare, utilizăm liste ori de câte ori este necesar să organizăm într-o formă secvenţială un ansamblu de informaţii.

**Lista** este o structură de date abstractă constituită dintr-o succesiune de elemente. Fiecare element din listă are un succesor (cu excepţia ultimului element al listei) şi un predecesor (cu excepţia primului element din listă).

Cel mai simplu mod (dar nu întotdeauna şi cel mai eficient) este de a memora elementele listei într-un vector. Succesiunea elementelor este în acest caz implicită (ordinea indicilor vectorului).

În cazul în care structura de date este dinamică (suportă frecvent operaţii de inserare, respectiv ştergere a unor elemente din structură) această implementare a listelor nu este eficientă. O altă implementare posibilă, eficientă pentru structuri de date liniare dinamice, sunt listele înlănţuite. În funcţie de numărul de legături existente între elemente consecutive ale listei, lista este simplu sau dublu înlănţuită.

## 2.1. Liste simplu înlănţuite

O **listă simplu înlănţuită** este o structură de date constituită dintr-o succesiune de elemente denumite noduri. Fiecare nod din listă conţine două părţi: o parte de informaţie (în care sunt memorate informaţiile corespunzătoare nodului, specifice problemei) şi o parte de legătură (în care este memorată adresa următorului element din listă).

Pentru simplitate, vom considera, fără a restrânge generalitatea, că informaţiile memorate în nodurile listei sunt numere întregi. Declaraţia structurii care reprezintă un nod din listă va fi:

**struct** Nod

{**int** inf;

**struct** Nod \* urm;};

Observăm că adresa fiecărui nod din listă este conţinută într-un alt nod, cu o singură excepţie: primul nod al listei. Prin urmare, pentru a identifica o listă simplu înlănţuită este suficient să reţinem adresa primului element al listei.

Vom defini tipul de date Lista, care este adresa unui nod din listă:

typedef **struct** Nod \* Lista;

Exceptând primul nod (a cărui adresă nu este conţinută de nici un nod al listei), într-o listă simplu înlănţuită mai există un nod special: ultimul nod. După acest nod nu mai urmează nici un alt nod al listei, deci ultimul nod va conţine în partea de legătură pointerul NULL.

Putem figura o listă simplu înlănţuită cu elemente întregi astfel:

3

9

2

◼

Prim

Am figurat prin săgeţi legăturile dintre noduri. Prim este un pointer care reţine adresa primului element din listă. Simbolul ◼ semnifică marcajul de sfârşit de listă (pointerul NULL).

Operaţiile fundamentale pe care trebuie să le implementăm sunt:

* 1. Inserarea unui nod în listă
  2. Ştergerea unui nod din listă
  3. Consultarea unei liste.

### Inserarea unui nod într-o listă simplu înlănţuită

Pentru inserare vom implementa o funcţie denumită Insert cu trei parametri:

1. Prim – pointer care conţine adresa primului nod din lista în care se face inserarea; acest parametru va fi transmis prin referinţă, deoarece în urma inserării începutul listei se poate modifica;
2. p – pointer care conţine adresa nodului din listă după care se face inserarea; dacă p este NULL deducem că inserarea noului nod se va face la începutul listei;
3. x – informaţia nodului care urmează să fie inserat în listă.

Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reţine în variabila pointer q. În zona de informaţii vom memora valoarea x (q->inf=x).

La inserare apar două cazuri distincte. Dacă p==NULL noul nod va fi inserat la începutul listei; dacă p!=NULL inserarea se face în interiorul listei.

Exemplu

Să ilustrăm pas cu pas inserarea la începutul listei din exemplul precedent a unui nou nod, cu informaţia 7, a cărui adresă va fi memorată în pointerul q:

1. Creăm o legătură de la noul nod către primul nod din listă, memorând în câmpul urm al noului nod adresa primului nod din listă (q->urm=Prim).

3

9

2

◼

Prim

7

1. În acest moment primul nod din listă este noul nod (Prim=q).

3

9

2

◼

Prim

7

În cel de al doilea caz trebuie să inserăm noul nod după nodul a cărui adresă este memorată în parametrul p.

Exemplu

Să ilustrăm pas cu pas inserarea în lista din exemplul precedent a unui nou nod, cu informaţia 7, după nodul cu informaţia 9.

1. Creăm o legătură de la noul nod către nodul care urmează după nodul indicat de p (q->urm=p->urm):

3

9

2

◼

Prim

7

p

q

1. Modificăm legătura de la nodul indicat de p, după acest nod va urma noul nod (p->urm=q).

3

9

2

◼

Prim

7

p

q

**void** Insert(Lista & Prim, Lista p, **int** x)

{Lista q=new Nod;

q->inf=x;

**if** (!p) {q->urm=Prim; Prim=q; }

**else** {q->urm=p->urm; p->urm=q; }

}

Observaţii

1. Crearea unei liste se realizează prin inserări succesive de elemente.
2. Funcţia de inserare creată este generală, funcţionează pentru toate cazurile posibile (inserare la începutul listei, inserare în interiorul listei şi inserare la sfârşitul listei).

### Ştergerea unui nod dintr-o listă simplu înlănţuită

Pentru ştergere vom implementa o funcţie denumită Delete cu doi parametri:

1. Prim – pointer care conţine adresa primului nod al listei din care se face ştergerea; acest parametru va fi transmis prin referinţă, deoarece în urma ştergerii începutul listei se poate modifica;
2. p – pointer care conţine adresa nodului din listă care precedă nodul ce urmează a fi şters (se transmite adresa nodului precedent şi nu adresa nodului de şters pentru a putea restaura corect legăturile în listă); dacă p este NULL deducem că va fi şters primul nod din listă.

Şi la ştergere apar două cazuri distincte. Dacă p==NULL va fi şters primul nod din listă; dacă p!=NULL va fi şters un nod din interiorul listei.

Exemplu

Să ilustrăm pas cu pas ştergerea primului nod al listei din exemplul precedent:

1. Memorăm în variabila q adresa nodului ce urmează a fi şters (q=Prim), apoi adresa primului element din listă se schimbă (Prim=Prim->urm).

3

9

2

◼

Prim

q

1. La sfârşit eliberăm zona de memorie alocată nodului ce a fost eliminat din listă (delete q).

În cel de al doilea caz trebuie să ştergem nodul care urmează după nodul a cărui adresă este memorată în parametrul p.

Exemplu

Să ilustrăm pas cu pas ştergerea din lista din exemplul precedent a nodului cu informaţia 9.

1. Memorăm în variabila q adresa nodului ce urmează să fie şters (q=p->urm), apoi modificăm legătura care exista de la p la q şi o transformăm într-o legătură la nodul care urmează după cel ce va fi şters (p->urm=q->urm).

3

9

2

◼

Prim

q

1. La sfârşit eliberăm zona de memorie alocată nodului ce a fost eliminat din listă (delete q).

**void** Delete(Lista & Prim, Lista p)

{Lista q;

**if** (p)

{q=p->urm;

**if** (q) {p->urm=q->urm; delete q; }

}

**else**

{q=Prim;

**if** (q) {Prim=Prim->urm; delete q; }

}

}

### Parcurgerea unei liste

A parcurge o listă presupune a vizita fiecare nod din listă, în scopul prelucrării informaţiilor asociate nodurilor.

În acest scop vom utiliza un pointer p căruia îi vom atribui iniţial adresa primului element din listă. Cât timp lista nu s-a terminat (p!=NULL), se vizitează nodul curent, apoi se trece la nodul următor (p=p->urm). Funcţia următoare reali­zează parcurgerea unei liste simplu înlănţuite cu afişarea informaţiilor din noduri:

**void** Afisare(Lista Prim)

{Lista p;

**for** (p=Prim; p; p=p->urm)

cout<<p->inf<<' ';

cout<<endl;

}

## 2.2. Liste simplu înlănţuite circulare

O listă simplu înlănţuită este circulară dacă după ultimul element din listă urmează primul element al listei.

Exemplu

3

9

2

Prim

4

Observaţii

1. Nici unul dintre nodurile unei liste circulare nu conţine valoarea NULL în partea de legătură. Operaţiile elementare asupra listelor simplu înlănţuite circulare vor fi implementate ţinând cont de această observaţie.
2. Pentru a identifica o listă simplu înlănţuită circulară putem reţine adresa oricărui element din listă.

### Inserarea unui nod într-o listă simplu înlănţuită circulară

Pentru inserare vom implementa o funcţie denumită Insert cu trei parametri:

* 1. Prim – pointer care conţine adresa primului nod din lista în care se face inserarea; acest parametru va fi transmis prin referinţă, deoarece în urma inserării începutul listei se poate modifica;
  2. p – pointer care conţine adresa nodului din listă după care se face inserarea;
  3. x – informaţia nodului care urmează să fie inserat în listă.

Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reţine în variabila pointer q. În zona de informaţii vom memora valoarea x (q->inf=x).

La inserare apar două cazuri distincte. Dacă Prim==NULL (lista este vidă) noul nod inserat va fi singurul element din listă (deci în partea de legătură vom reţine adresa sa); dacă Prim!=NULL inserarea se face în interiorul listei.

**void** Insert(Lista & Prim, Lista p, **int** x)

{Lista q=new Nod;

q->inf=x;

**if** (!Prim) {q->urm=q; Prim=q; }

**else** {q->urm=p->urm; p->urm=q; }

}

### Ştergerea unui nod dintr-o listă simplu înlănţuită circulară

Pentru ştergere vom implementa o funcţie denumită Delete cu doi parametri:

1. Prim – pointer care conţine adresa primului nod al listei din care se face ştergerea; acest parametru va fi transmis prin referinţă, deoarece în urma ştergerii începutul listei se poate modifica;
2. p – pointer care conţine adresa nodului din listă care precedă nodul ce urmează a fi şters (se transmite adresa nodului precedent şi nu adresa nodului de şters pentru a putea restaura corect legăturile în listă).

Şi la ştergere apar două cazuri distincte. Dacă p->urm==Prim va fi şters primul nod din listă, deci valoarea parametrului Prim va fi actualizată (va primi adresa următorului nod din listă sau NULL dacă lista era formată dintr-un singur nod); dacă p->urm!=Prim va fi şters un nod oarecare din interiorul listei.

**void** Delete(Lista & Prim, Lista p)

{Lista q=p->urm;

p->urm=q->urm;

**if** (q==Prim)

{Prim=Prim->urm;

**if** (Prim->urm==Prim)

Prim=NULL; }

delete q; }

}

### Parcurgerea unei liste simplu înlănţuite circulare

Pentru a parcurge o listă simplu înlănţuită circulară, vom utiliza un pointer p căruia îi vom atribui iniţial adresa primului element din listă. La fiecare pas, se vizitează nodul curent, apoi se trece la nodul următor (p=p->urm), până când se revine la nodul de plecare. Funcţia următoare reali­zează parcurgerea unei liste simplu înlănţuite circulare cu afişarea informaţiilor din noduri:

**void** Afisare(Lista Prim)

{Lista p=Prim;

**if** (!Prim) **return**;

**do**

{cout<<p->inf<<' '; p=p->urm; }

**while** (p!=Prim);

cout<<endl; }

## 2.3. Liste dublu înlănţuite

O **listă dublu înlănţuită** este o structură de date constituită dintr-o succesiune de elemente denumite noduri. Fiecare nod din listă conţine trei părţi: o parte de informaţie (în care sunt memorate informaţiile corespunzătoare nodului, specifice problemei) şi două părţi de legătură (în care este memorată adresa următorului element din listă şi respectiv adresa precedentului element din listă).

Pentru simplitate, vom considera, fără a restrânge generalitatea, că informaţiile memorate în nodurile listei sunt numere întregi. Declaraţia structurii care reprezintă un nod dintr-o listă dublu înlănţuită cu informaţii întregi va fi:

**struct** Nod

{**int** inf;

**struct** Nod \* urm, \* pre; };

typedef **struct** Nod \* Lista;

Exemplu

Putem figura o listă dublu înlănţuită cu elemente întregi astfel:

Prim

7

2

◼

1

◼

### Inserarea unui nod într-o listă dublu înlănţuită

Pentru inserare vom implementa o funcţie denumită Insert cu trei parametri:

* 1. Prim – pointer care conţine adresa primului nod din lista în care se face inserarea; acest parametru va fi transmis prin referinţă, deoarece în urma inserării începutul listei se poate modifica;
  2. p – pointer care conţine adresa nodului din listă după care se face inserarea; dacă p este NULL deducem că inserarea noului nod se va face la începutul listei;
  3. x – informaţia nodului care urmează să fie inserat în listă.

Vom aloca dinamic memorie pentru un nou nod, a cărui adresă o vom reţine în variabila pointer q. În zona de informaţii vom memora valoarea x (q->inf=x).

La inserare apar două cazuri distincte. Dacă p==NULL noul nod va fi inserat la începutul listei; dacă p!=NULL inserarea se face în interiorul listei.

Exemplu

Să ilustrăm pas cu pas inserarea la începutul listei din exemplul precedent a unui nou nod, cu informaţia 3, nod a cărui adresă va fi memorată în pointerul q:

1. Creăm o legătură de la noul nod către primul nod din listă, memorând în câmpul urm al noului nod adresa primului nod din listă (q->urm=Prim). Noul nod va fi primul nod din listă, deci nu are precedent (q->pre=NULL).

Prim

7

2

◼

1

◼

3

◼

q

1. Precedentul primului nod din lista va fi q (Prim->pre=q), iar primul nod din lista este de acum noul nod (Prim=q).

Prim

7

2

1

◼

3

◼

q

În cel de al doilea caz trebuie să inserăm noul nod după nodul a cărui adresă este memorată în parametrul p.

Exemplu

Să ilustrăm pas cu pas inserarea în lista din exemplul precedent a unui nou nod, cu informaţia 3, după nodul cu informaţia 7.

1. Creăm legăturile dintre noul nod şi precedentul său în listă (q->pre=p), respectiv succesorul său în listă – nodul indicat de p->urm (q->urm=p->urm):

Prim

7

2

◼

1

◼

3

q

1. Modificăm legăturile dintre nodurile din listă: după nodul indicat de p va urma noul nod (p->urm=q), iar înaintea nodului care urma după cel indicat de p (dacă acesta există) va fi noul nod (q->urm->pre=q).

q

Prim

7

2

◼

1

◼

3

p

**void** Insert(Lista & Prim, Lista p, **int** x)

{Lista q=new Nod;

q->inf=x; q->pre=p;

**if** (!p) //inserare la inceputul listei

{q->urm=Prim;

**if** (Prim) Prim->pre=q;

Prim=q; }

**else**

{q->urm=p->urm; p->urm=q;

**if** (q->urm) q->urm->pre=q; }

}

### Ştergerea unui nod dintr-o listă dublu înlănţuită

Pentru ştergere vom implementa o funcţie denumită Delete cu doi parametri:

1. Prim – pointer care conţine adresa primului nod al listei din care se face ştergerea; acest parametru va fi transmis prin referinţă, deoarece în urma ştergerii începutul listei se poate modifica;
2. q – pointer care conţine adresa nodului din listă care urmează a fi şters.

Pentru simplitate, vom utiliza doi pointeri suplimentari – p care reţine adresa nodului care precedă nodul de şters şi respectiv r, care reţine adresa nodului care succedă nodul de şters. Dacă p==NULL va fi şters primul nod din listă şi va trebuie să actualizăm valoarea parametrului Prim.

**void** Delete(Lista & Prim, Lista q)

{Lista p=q->pre, r=q->urm;

**if** (p) p->urm=r;

**else** Prim=r;

**if** (r) r->pre=p;

delete q; }

### Consultarea unei liste dublu înlănţuite

Consultarea unei liste dublu înlănţuite se realizează în mod similar cu cea a unei liste simplu înlănţuite, numai că o listă dublu înlănţuită poate fi consultată atât în sens direct (utilizând legăturile urm), cât şi în sens invers (utilizând legăturile pre).