IMPLEMENTAREA ÎNLĂNŢUITĂ A LISTELOR LINIARE

Remember (reamintirea unor noţiuni din programarea C/C++)

Pointeri

Pentru o variabilă simplă, de exemplu,

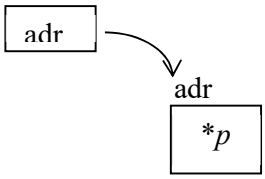
```
int x;
```

`x` = reprezintă valoarea variabilei, iar
`&x` = adresa din memorie a variabilei `x`

Un *pointer de date*/obiecte (nu pointer către functii sau către tipul void) conţine adresa unei variabile din memorie. De exemplu pentru: *p*

```
int * p;
```

//p este pointer către o variabilă de tip întreg
`p` = adresa variabilei la care se referă pointerul
`*p` = conţinutul variabilei la care se referă pointerul



Este important de reţinut că orice variabilă pointer trebuie iniţializată cu o valoare validă, 0 sau NULL (constantă din C/C++, declarată în stdio.h, stdlib.h, etc) sau adresa unui obiect (care este de exemplu FFF4, oricum diferită de zero), înainte de a fi utilizată. În caz contrar, efectele pot fi grave, deoarece la compilare sau în timpul execiţiei nu se fac verificări ale validităţii valorilor pointerilor. Pentru vizualizarea adreselor, în C, se poate folosi funcţia `printf`, cu specificatorul de format `"%p"`; adresele sunt afişate in hexazecimal (de exemplu, FFF4).

Structuri

O structură este o colecţie de obiecte de orice tip, referite cu un nume comun. De exemplu structura "student" poate conţine date despre numele şi medii sale. O declaraţie de structură precizează identificatorii şi tipurile elementelor componente şi constituie o definiţie a unui tip de date nou. De exemplu,

```
struct fractie
{
   float numitor, numarator; //campurile structurii
} f1, f2;  //var f1 si f2 sunt declarate de tip fractie
```

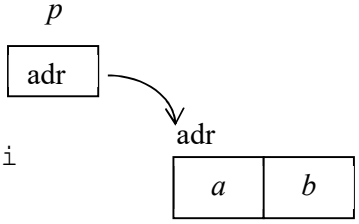
Am şi declarat structura `fractie` şi variabilele `f1`, respectiv `f2` de tip `fractie`.
Referirea la un câmp se face cu punct:

```
f1.numarator = 1;    f1.numitor = 2;
```

Pointeri către structuri

Exemplu:

```
struct fractie
{
  int a,b; //campurile structurii
} *p;
```

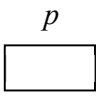


Referirea la un câmp se face astfel:

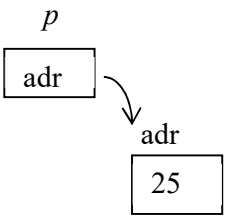
```
(*p).a = 1;    p->a = 1;
(*p).b = 2;    p->b = 2;
```

Alocare dinamică în C++

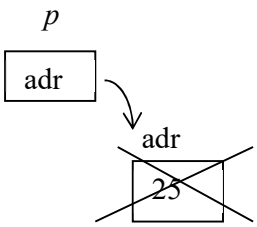
Exemplu:




```
int *p;
```
//se declara un pointer care deocamdata nu se refera la nimic

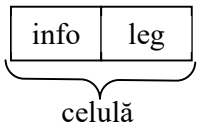



```
p = new int;  //sau p = new(int);
```
//se aloca spatiu in memori heap pentru un întreg,
// iar adresa acestuia se retine în pointerul p
```
*p = 25;
```
//valoarea variabilei la care se referă p va fi 25
//cele 2 instrucţiuni sunt echivalente cu `p = new int (25);`




```
delete p;
```
//la final se eliberează spaţiul ocupat de variabila la care se referă `p`

Revenim acum la implementarea înlănţuită (dinamică) a listelor liniare.



În implementarea înlănţuită, fiecare element al listei este format din două părţi: *partea de informaţie utilă* şi *partea/părţile de legătură*. Partea de informaţie este cea care se prelucrează, iar partea/părţile de legătură indică adresa/adresele de memorie unde se află elementul / elementele care au o relaţie logică cu acest element. Un element format din partea de dată şi partea de legătură o vom numi mai departe *celulă*. Dacă un element este ultimul în ordine logică, legătura lui este o valoare care nu se poate referi la o locaţie *validă*. Această valoare o vom numi *legătura(adresa) vidă* şi o în C/C++ este constanta numerică NULL.

Pentru parcurgerea unei liste este de ajuns să ştim adresa celulei în care se găseşte primul element al său. De aceea dăm ca valoare pentru variabila cu numele unei liste adresa primului element al său. Dacă lista este vidă (nu conţine nici un element) atunci variabilei corespunzătoare ei i se dă valoarea legătura vidă (cazul listelor reprezentate *fără header*).

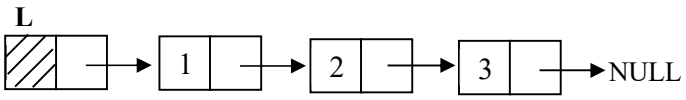
Implementarea dinamică a listelor are dezavantajul că accesul la o anumită dată se face încet. Cel de-al *i*-lea element al unei liste se găseşte parcurgând *i* legături şi timpul de acces creşte cu *i*, pe când într-un tablou timpul de acces este independent de *i*. Un alt dezavantaj este faptul că este necesar spaţiul de memorie şi pentru legături, spaţiu ce în mod obişnuit este folosit pentru memorarea datelor. Marele avantaj este că permite o alocare dinamică a memoriei.

Să se implementeze în C/C++ operaţiile elementare efectuate asupra unei liste liniare reprezentată cu legături.

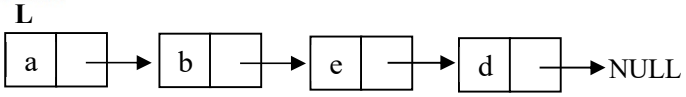
Soluţie:

Listele simplu înlănţuite pot fi implementate folosind două abordări: cu header şi fără header. Declaraţia celulei nu depinde de abordare. Header-ul este o celulă suplimentară în listă care face legătura către prima celulă efectivă din listă şi care oferă avantajul ca inserările/ştergerile în/din listă să se facă pe acelaşi tipar oriunde în listă. La unele aplicaţii se pot folosi ambele abordări cu acelaşi succes, însă există aplicaţii în care este de preferat utilizarea uneia dintre ele.

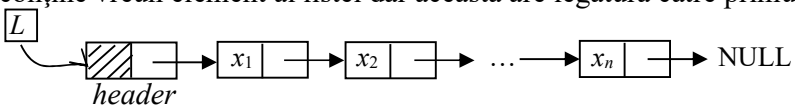
Exemplu de listă cu header:



Exemplu de listă fără header:



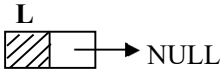
Aici, se consideră cazul **implementării cu header (santinelă)**, în care prima celulă (capul listei) nu conţine vreun element al listei dar aceasta are legătură către primul element al listei.



În acest caz **declararea listei** liniare se poate face astfel:

```
struct celula
{ int info;          // am considerat elementele listei numere întregi
  celula *leg;       // pointer către următoarea celulă (element al listei)
} *L;                // L este pointer către header
```

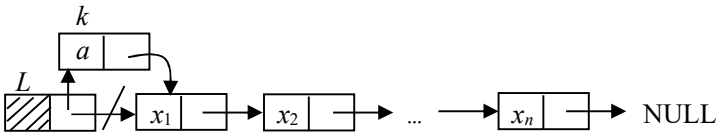
Iniţializarea listei presupune crearea header-ului (santinela) şi initializarea legăturii sale cu NULL (nu are legătură la o celulă) care în C++ poate fi:




```
void Initializare (celula *&L)
{L = new (celula);
  //se creeaza o noua celula spatiu in mem cat pt o celula, adresa acesteia
 L->leg = NULL;  // se retine în L, apoi celula "se leagă" la NULL
}
```

Inserarea la începutul listei a unui element *a* presupune crearea unei noi celule *k* în care se pune elementul *a*, apoi legarea celulei *k* de primul element al listei dat de header şi legarea heder-ului de noua celulă introdusă.

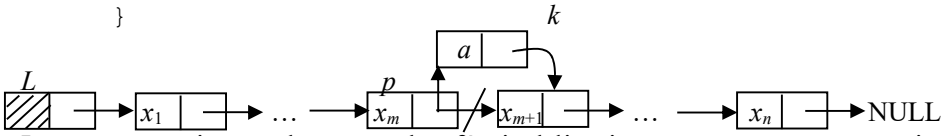
```
void InserareInceput (celula *&L, int a) {
celula *k=new celula;  //se declara si se creeaza o noua celula, adr fiind reţinută în k
k->info = a;  //in celula referita de k, în al câmpul "info" se pune valoarea a
k->leg = L->leg;  // celula k "se leaga" la celula de după L
L->leg = k;  //celula L "se leagă" la celula k
}
```



Inserarea unui element *a* după celula de pe locul *m* în listă (pentru *m* = 0 se pune la începutul listei) presupune crearea unei noi celule în care se pune noul element, determinarea adresei *p* a celulei în care se află elementul de pe locul *m* în listă prin avansarea în listă de *m* ori pornind de la header (am considerat că dacă lista are mai puţin de *m* elemente se adaugă elementul la sfârşitul listei), apoi schimbarea legăturilor celulei nou introduse şi a lui *p*.

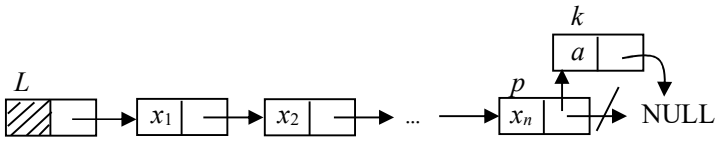
```
void InserareInterior (celula *&L, int a, int m){
celula *k = new(celula), *p = L;  //p este un pointer referit la header
k->info = a;
for(int i = 1; i <= m && p->leg != NULL; i++)
   p = p->leg;  //avansez în listă cu p de m ori si daca am unde avansa
```

```
    //în final p se referă la celula a m-a sau la ultima
    k->leg = p->leg;  //noua celula se leaga la celula a (m+1)-a
    p->leg = k;  //celula a m-a se leagă la noua celula
    }
```



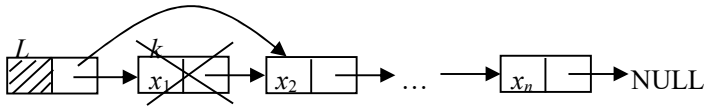
Inserarea unui nou element *a* la sfârşitul listei presupune crearea unei noi celule în care se pune elementul şi legarea ei de ultimul element al listei, găsit prin parcurgerea listei până când se determină o celulă cu legătura `NULL`.

```
void InserareSfarsit (celula *&L, int a)
{  celula *k = new(celula), *p = L;  //k=pointer la o noua celula
   while(p->leg != NULL) p = p->leg;
   // cat timp p are legatura, p avans pe legatura sa  =>  p se va referi la ultima celula
   k->info = a; // in noua celula se pune informatia a
   k->leg = NULL;  //noua celula se leaga la NULL (nu mai este nimic dupa ea)
   p->leg = k;  // anterior ultima celula se leaga la noua celula
}
```



Eliminarea primului element din listă presupune: legarea header-ului la cel de-al doilea element al listei şi ştergerea celulei ce conţinea primul element.

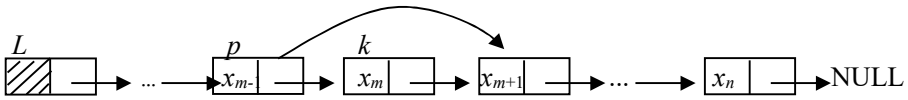
```
void StergereInceput (celula *&L, int &a){
if(L->leg != NULL)  // lista nu este vidă
   {
    celula *k = L->leg;  //se reţine adr primei celule efective (ce dupa L) in k
    a = k->info;          //se reţine informaţia acestei celule
    L->leg = k->leg;      //se leagă headerul la al doilea element din listă
    delete(k); //se eliberează zona de mem ocupată de celula la care se referă k
   }
   else cout << " Nu sunt elemente de sters ";
}
```



Eliminarea elementului de pe locul *m* din listă presupune determinarea adresei *k* a acestei celulei şi a celei precedente *p*, găsite prin avansarea în listă de *m*-1 ori pornind de la header, apoi legarea lui *p* de următoarea celulă de după *k* şi eliberarea celulei *k*.

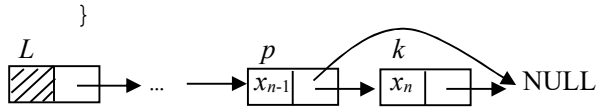
```
void StergereInterior (celula *&L, int &a, int m)
{  celula *p = L, *k;
   for(int i = 1; i < m && p->leg != NULL; i++)
       p = p->leg;          //avansez în listă de m-1 ori
   k = p->leg;              // k se referă la elementul al m-lea
   if (k == NULL) cout << "Nu exista elementul" << m;
   p->leg = k->leg;    // se leagă celula a (m-1)-a la celula a (m+1)-a
   a = k->info;        // se reţine valoarea celulei ce se va elimina
```

```
        delete(k);              //se sterge celula referita de k (celula a m-a)
    }
```



Eliminarea ultimului element al listei presupune, determinarea adresei *k* a ultimei celule şi a celei precedente *p*, care se leagă la NULL şi eliberarea celulei *k*.

```
void StergereSfarsit (celula *&L, int &a){
 celula *p = L, *k;
 if(L->leg != NULL) {
    while(p->leg->leg != NULL)
        p = p->leg;  // p se va referi la penultima celula
    k = p->leg;    //k se va referi la ultima celula
    a = k->info;   //retin informatia din ultima celula
    p->leg = NULL; //leg penultima celula la NULL
    delete(k);    //sterg din memorie ultima celula
 }
 else cout << " Nu sunt elemente de sters ";
}
```



CREARE UNEI LISTE CU HEADER

```
void CreareLista(celula *&L)
{
 int n;
 cout<<"Cate elemente veti introduceti in lista? ";
 cin>>n;
 //in lista va fi initial header-ul "legat" la NULL
 L = new(celula);
 L->leg = NULL;
 //citim elementele efective ale listei (dacă nu se introduc elemente ramane doar header-ul
 cout<<"Dati elementele listei: ";
 celula *ultim = L, *temp;  //ultim este pointer catre ultimul element inserat in lista

for (int i=1;i<=n;i++)
 {
  temp = new(celula);
  //inserez valoarea citita in campul info din temp si fac legaturile coresp.
  cin>>temp->info;
  ultim->leg = temp;
  //intai se leaga ultima celula deja inserata la noua celula
  temp->leg = NULL;
  //apoi se leaga noua celula la temp
  //desi acest lucru se poate face dupa for pt ultima cel.temp
  ultim = temp; //acum ultima celula creata este aceasta
  }
 }
```

```
}
```

AFIŞAREA ELEMENTELOR UNEI LISTE CU HEADER

```
void Listare(celula *L) {
 if (L->leg == NULL) cout<<" lista vida"; //exista doar header-ul
 else {
  cout<<"Elementele listei sunt: ";
  celula *p=L; //pornim de la header
  //"plimbaretul" prin lista care porneste de pe header
  while (p!=NULL) {
   p = p->leg; //intai avansez
   cout<<p->info<<"  "; //apoi afisez info din celula curenta
  }
}
```

sau, echivalent

```
void Listare(celula *L) {
 if (L->leg == NULL) cout<<" lista vida"; //exista doar header-ul
 else  {
  cout<<"Elementele listei sunt: ";
  celula *p=L->leg; //pornim de pe prima celula efectiva
  //se porneste de pe prima celula efectiva-sigur exista aici
  while (p!=NULL) {//p se refera la o celula efectiva
   cout<<p->info<<"  "; //intai afisez info din celula curenta
   p = p->leg; //apoi avansez
  }
}
```

sau, făcând parcurgerea listei cu for:

```
void Listare(celula *L) {
 if (L->leg == NULL)
   cout<<" lista vida"; //exista doar header-ul
 else
 {
  cout<<"Elementele listei sunt: ";
  for (celula *p=L->leg; p; p = p->leg)   cout<<p->info<<"  ";
 }
}
```

CREARE UNEI LISTE FĂRĂ HEADER

Diferența va consta în faptul că valoarea primului element din listă se va pune în celulă referită de *L*, iar celelalte elemente se leagă în şir la *L*.

```
void CreareLista(celula  *&L) {
 L = NULL; //initializarea unei liste fara header
 int n;
 cout<<"Cate elemente introduceti in lista? ";cin>>n;
 if (n>0) {
  cout<<"Dati elementele listei: ";
  //citesc primul element si il pun in celula referita de L
  L = new(celula);   cin>> L->info;   L->leg = NULL;
```

```
  celula *ultim=L, *temp;
  for (int i=2;i<=n;i++)
  //se merge cu "tiparul" de la liste cu header
  {
   temp = new(celula);
   cin>>temp->info;
   ultim->leg = temp;
   temp->leg = NULL;
   ultim = temp;
  }
 }
}
```

AFIŞAREA ELEMENTELOR UNEI LISTE FARA HEADER

```
void Listare(celula *L) {
 if (L == NULL)
   cout<<" lista vida"; //L nu se refera la nicio celula
 else
 {
  cout<<"Elementele listei sunt: ";
  celula *p=L;
  //se porneste de pe prima celula efectiva data de header
  while (p!=NULL) //p se refera la o celula efectiva
  {
   cout<<p->info<<"  "; //intai afisez info din celula curenta
   p = p->leg; //apoi avansez
  }
}
```

sau, făcând parcurgerea listei cu "for":

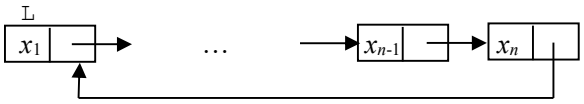
```
void Listare(celula *L)
{
 if (!L) //sau L == NULL
   cout<<" lista vida";
 else
 {
  cout<<"Elementele listei sunt: ";
  for (celula *p=L; p; p = p->leg)
    cout<<p->info<<"  ";
 }
}
```

LISTELE CIRCULARE

Sunt liste cu legături în care legătura ultimului element este primul element al listei, în loc de NULL. Se pot parcurge elementele listei plecând din orice punct al listei.

(Problema lui Josephas) Se dă o listă cu elementele 1, 2, …, *n*. Începând de la primul element, cu un pas dat să se afişeze şi să se elimine toate elementele listei.

Soluţie: Se poate folosi o implementare cu listă circulară simplu înlănţuită *fără header*, primul element fiind la adresa dată de pointerul *L* şi ultimul element al listei fiind legat la adresa dată de pointerul *L* (la primul element). Funcţia care rezolvă problema este dată în continuare:




```
void Josephas(celula *&L, int pas)
{//elimina din lista L elemente din pas in pas
 int i;
 if (L == NULL) { cout<<"Lista este vida"; return; }
 celula *p = L, *temp;
 while (p->leg != L) p=p->leg;
 //p se va referi la ultima celula din lista circulara
 cout<<"Elementele eliminate sunt urmatoarele: ";
 while (L->leg != L)
  //cat timp exista cel putin 2 elemente in lista
  {for (i=1; i<pas; i++) p=p->leg; //p avanseaza de pas-1 ori
   temp=p->leg; //temp se va referi la celula de sters
   cout<<temp->info<<" ";//se afiseaza informatia din aceasta celula
   p->leg = L = temp->leg; //se refac legaturile sarind peste cea de sters
   delete temp; //se sterge din memorie celula referita de pointerul temp
  }
  cout<<L->info<<endl;
  delete L; //la final se sterge ultima celula ramasa in lista
}
```

Observaţie: Crearea unei liste circulare cu elementele 1, 2, …, *n*, se poate face astfel:

```
void CreareLista(celula *&L) {  //lista circulara cu elem 1,…,n
 int n;
 cout<<"n="; cin>>n;
 if (n<=0) {cout<<"Eroare"; getch(); exit(1);}
 L = new(celula); L->info = 1; L->leg = L;
 //s-a creat prima celula (cu informatia 1)
 celula *ultim = L,*temp;
 for(int i=2;i<=n;i++) {
   temp = new(celula);
   temp->info = i;
   ultim->leg = temp; //ultima celula creata pana acum se leaga
   temp->leg = L;     //la noua celula, care se leaga apoi la L
   ultim = temp; //aceasta este acum ultima celula adaugata
 } }
```

ALTE APLICAȚII ALE LISTELOR SIMPLU ÎNLĂNȚUITE

1. Să se realizeze o funcţie ce numără elementele unei liste.
Soluţie: Vom considera întâi cazul implementării fără header.

```
int NumarElemente(celula *L) {
  celula *p = L; //pointer catre prima celula din lista
  //care contine efectiv un element (lista fara header)
  int contor = 0;
  while (p) //p!=NULL, se refera efectiv la o celula
  {
   contor++;      //numaram celula curenta
   p = p->leg;  //trecem la urmatoarea celula
  }
 return contor;
}
```

În cazul implementării cu header, pornim cu *p* de pe prima celula efectivă, adică:

```
celula *p = L->leg;
```

Este singura modificare necesară.

2. Să se realizeze o funcţie ce numără elementele unei liste egale cu o anumită valoare.
Soluţie: În cazul anterior număram toate celulele, acum va trebui să le contorizăm doar pe cele cu o anumită informaţie.

```
int NumarElementeI (celula *L, int i) {
  celula *p=L; //consideram cazul listei fara header
  //pointer catre prima celula din lista
  int contor = 0;
  while (p!=NULL) { //p se refera efectiv la o celula
   if (p->info == i)   contor++;
   //numaram celulele cu informatia i
   p = p->leg;  //apoi oricum trecem la celula urmatoare
  }
 return contor;
}
```

3. Să se realizeze reuniunea a două mulţimi reprezentate ca liste liniare cu legături simple.
Soluţie: Dându-se două liste simplu înlănţuite, pentru a determina reuniunea lor se consideră toate elementele din a doua listă care nu se găsesc în prima listă şi se inserează în lista finală care este iniţializată cu elementele din prima listă. Aici listele se consideră implementate fără header. Vom da în continuare funcţia care realizează lista de capăt *L* ce reprezintă reuniunea elementelor listelor de capete *L*1 şi *L*2, ştergându-se celulele ce nu intră în reuniune.

```
void Reuniune (celula *L1, celula *&L2, celula *&L)
{celula *p, *p2;
 L = L1; //initial elementele primei liste "le trecem" si in reuniune
 if (L == NULL) { L=L2; return; }
    //prima lista vida => reuniunea contine doar elemente din a doua lista
 while (L2) //se parcurg elementele din a doua lista
 {
  p2 = L2; //primul element din a doua lista
  L2 = L2->leg;  //L2 se va referi la urmatorul element din lista a doua
  for (p=L; p->info!=p2->info&&p->leg!=NULL; p=p->leg);
```

```
  //parcurg L pana gasesc o celula cu info ca p2->info sau pana la ultima celula
  if (p->info != p2->info)  //am gasit informatia din p2
    {
     p->leg = p2;  //leg celula p la p2
     p2->leg = NULL; //=> in fata celulei referita de p2
    }
  else delete p2;  //element din a doua lista care-i si in prima
 }
}
```

4. Să se împartă elementele unei liste cu legături în două liste, prima conţinând elementele de ordin impar şi cea de-a doua pe cele de ordin par.

Soluţie: Rezolvarea problemei presupune schimbarea legăturilor astfel: primul element din lista iniţială va fi legat de al treilea, al doilea de al patrulea etc. Se consideră cazul listelor reprezentate fără header.

```
void PozParImpar(celula *L, celula *&L1, celula *&L2)
{ celula *p1, *p2;
  p1 = L1 = L; //p1 porneste de pe primul element
  if (L == NULL) { L2 = NULL; return; }
  p2 = L2 = L->leg; //p2 porneste de pe al 2-lea elem
  while (p2) //p2!= NULL => p2 se refera la o celula
   {
    p1->leg = p2->leg; //leg p1 la ce este dupa p2
    p1 = p2;  //p1 avansaza pe urmatoarea (referita de p2)
    p2 = p2->leg; //p2 avanseaza pe urmatoarea celula
   }
}
```

5. Fiind dată o listă simplu înlănţuită de numere întregi, să se creeze cu elementele ei două liste ce conţin elementele pare şi respectiv impare ale ei.

Soluţie: Se parcurge lista iniţială şi în cazul în care elementul este par se leagă la prima listă, iar dacă este impar se leagă la a doua listă. O funcţie ce poate realiza aceasta este următoarea:

```
void ValParImpar(celula *L, celula *&L1, celula *&L2)
{celula *p, *p1, *p2; //liste fara header
 p1 = p2 = L1 = L2 = NULL; //se initializeaza noile liste
 while (L) //exista elemente in lista initiala
 {
  p = L; //primul element din lista initiala
  L = L->leg; //L va fi capul restului de lista
  if (p->info%2)  //daca elementul este impar se pune in prima lista
    if (!L1) { //ca prim element
       L1 = p1 = p; //capul de lista L1 este la celula p
       p->leg = NULL;
      }
    else {
    //nu ca prim element => se leaga de ultima inserata
       p1->leg = p;
       p->leg = NULL;
       p1=p;
      }
  else //altfel se insereaza la sfarsitul celei de-a doua lista
    if (!L2) { //ca prim element
       L2 = p2 = p;
       p->leg = NULL;
      }
```

```
        else {
           p2->leg = p;
           p->leg = NULL;
           p2=p;
          }
    }
}
```

6. Să se inverseze legăturile unei liste simplu înlănţuite în aşa fel încât primul element să devină ultimul şi reciproc.

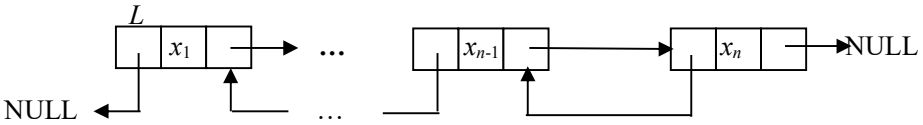
Soluţie: Presupunem cazul listei reprezentate fără header. Rezolvarea problemei presupune folosirea a trei pointeri: *p*1, *L*, *p*3 care fac referire la trei elemente consecutive din listă; *p*1 şi *L* se folosesc pentru inversarea legăturii, iar *p*3 va reţine adresa capătului părţii din lista iniţială, nemodificată încă.

```
void Inversare(celula *&L)
{ celula *p1 = NULL, *p3;
  //p1 = pointer catre celula de dinainte de L
  //p3 = pointer catre celula de dupa L
  if (L == NULL) return; // L nu se mai refera la o celula efectiv
  while((p3 = L->leg) != NULL)
    //daca p3 care este pointer catre celula de dupa L se refera la o celula
    {
     L->leg = p1; //leg L la celula anterioara
     p1 = L; //p1 avanseaza in restul listei
     L = p3; //p3 avanseaza in restul listei
    }
  L->leg = p1; //a mai ramas aceasta legatura
}
```

LISTE CU LEGĂTURI DUBLE

Listele cu legături duble au celule cu câte două legături, una către celula predecesoare, pe care o vom numi *legătură stângă* şi o vom nota *legs*, şi una către celula succesoare, pe care o vom numi *legătură dreaptă* şi o vom nota *legd*.

Aplicaţie (suplimentară). Să se implementeze în C/C++ operaţiile cu liste dublu înlănţuite.
Soluţie: Se foloseşte o reprezentare a listei *fără header* şi spre deosebire de listele simplu înlănţuite aici există două legături ale unei celule: una spre celula din stânga (anterioară) şi una spre celula din dreapta (următoare).



În acest caz, structura unei celule este:

```
struct celula
{ int info;
  celula *legs, *legd;
} *L;
```

Codul C++ pentru iniţializarea listei poate fi:

```
void Initializare() { L = NULL; }
```

Inserarea unui nou element *a* la începutul listei se poate face astfel:

```
void InserareInceput(int a){
  celula *k = new(celula);
  k->info = a;
  k->legd = L;  k->legs = NULL;  L = k;
  if(k->legd != NULL) k->legd->legs = k;
}
```

Inserarea unui nou element *a* după al *m*-lea elemnt din listă sau la sfârşitul listei, dacă *m* este mai mare decât numărul de elemente din listă se poate face astfel:

```
void InserareInterior(celula *&L, int a, int m){
  celula *k = new(celula), *s = NULL, *d = L;
  k->info = a;
  for( ; m > 0 && d != NULL; m--) { s = d; d = d->legd;}
  k->legs = s;  k->legd = d;
  if(s != NULL) s->legd = k;
  else L = k;
  if(d != NULL) d->legs = k;
}
```

Inserarea unui nou element *a* la sfarşitul listei se poate face astfel:

```
void InserareSfarsit(celula *&L, int a){
  celula *k = new(celula), *s = L;
  k->info = a;
  k->legd = NULL;
  if(L == NULL) { k->legs = NULL; L = k; return; }
  while(s->legd != NULL) s = s->legd;
  s->legd = k; k->legs = s;
}
```

Eliminarea primului element din listă se poate face astfel:

```
void StergereInceput(celula *&L, int &a)
{
  celula *k = L;
  if(k == NULL) { cout << "Lista vida"; return; }
  a = k->info;
  L = L->legd;
  if(L != NULL) L->legd = NULL;
  delete(k);
}
```

Eliminarea elementului de ordin *m* din listă se poate face astfel:

```
void StergereInterior(celula *&L, int &a, int m)
{
  celula *s = NULL, *k = L, *d;
  while(--m > 0 && k != NULL) { s = k; k = k->legd; }
  if(m) cout << "Nu exista element de ordinul dat";
  else
  { a = k->info;    d = k->legd;
    if(s != NULL) s->legd = d;  else L = d;
    if(d != NULL) d->legs = s;
    delete(k);  }
}
```

Eliminarea ultimului element din listă se poate face astfel:

```
void StergereSfarsit(celula *&L, int &a)
{
  celula *s = NULL, *k = L;
  if(L == NULL) cout << "Lista vida";
  while(k->legd != NULL) { s = k; k = k->legd; }
  a = k->info;
  if(s != NULL) s->legd = NULL;  else L = NULL;
  delete(k);
}
```