10. LISTELE – STRUCTURI DINAMICE NECONTIGUE

10.1 Consideraţii privind structurile de date de tip listă

O *listă liniară* (numită şi listă înlănţuită -"Linked List") este o colecţie de n>=0 elemente x[1], … x[n] toate de un tip oarecare, numite *noduri* între care există o relaţie de ordine determinată de poziţia lor relativă. Ea este deci o mulţime eşalonată de elemente de acelaşi tip având un număr arbitrar de elemente. Numărul n al nodurilor se numeşte *lungimea listei*. Dacă n=0, lista este vidă. Dacă n>=1, x[1] este primul nod iar x[n] este ultimul nod. Pentru 1<k<n, x[k] este precedat de x[k-1] şi urmat de x[k+1].

Acest tip de structură de date se aseamănă cu o structură standard: tipul tablou cu o singură dimensiune (vector), ambele structuri conţinând elemente de acelaşi tip iar între elemente se poate stabili o relaţie de ordine. Una dintre deosebiri constă în numărul variabil de elemente care constituie lista liniară, dimensiunea acesteia nu trebuie declarată şi deci cunoscută anticipat (în timpul compilării) ci se poate modifica dinamic în timpul execuţiei programului, în funcţie de necesităţi. Astfel utilizatorul nu trebuie să fie preocupat de posibilitatea depăşirii unei dimensiuni estimate iniţial, singura limită fiind mărimea zonei heap din care se solicită memorie pentru noile elemente ale listei liniare. Un vector ocupă în memorie un spaţiu continuu de memorie, pe când elementele unei liste simplu înlănţuite se pot găsi la adrese nu neapărat consecutive de memorie.

O altă deosebire avantajează vectorii, deoarece referirea unui element se face prin specificarea numărului de ordine al respectivului element, pe când accesul la elementele unei liste liniare se face secvenţial, pornind de la capul listei (adresa primului nod al listei) până la ultimul element al ei, ceea ce măreşte uneori considerabil timpul de acces la un anumit element. Pentru o listă liniară este obligatoriu să existe o *variabilă*, declarată în timpul compilării, denumită *cap de listă* care să păstreze adresa primului element al listei. Pierderea acestei valori va duce la imposibilitatea accesării elementelor listei liniare.

Pentru implementarea dinamică a unei liste liniare, folosind pointeri, nodurile listei vor fi structuri ce conţin două tipuri de informaţie:

- câmpurile ce conţin *informaţia structurală a nodului*
- câmpurile ce conţin *informaţia de legătură*, ce vor conţine pointeri la nodurile listei. Înlănţuirea secvenţială a elementelor unei liste se face utilizând variabile de tip pointer, care specifică adresa de memorie a elementelor adiacente. Fiecare nod are un predecesor şi un succesor, mai puţin elementele prim şi ultim dacă lista nu este circulară.

Listele înlănţuite cu un singur câmp de legătură se numesc *liste simplu înlănţuite* (legătura indică următorul element din listă). Fiecare nod conţine un pointer ce conţine adresa nodului următor din listă.

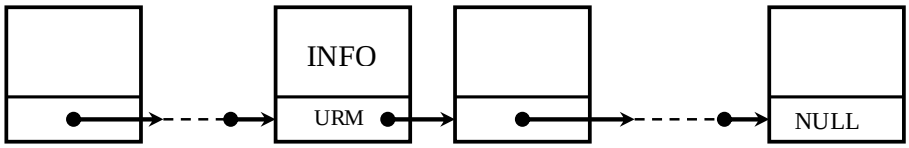


Figura 10.1 *Lista simplu înlănţuită*

Ultimul element poate conţine ca adresă de legătură fie constanta NULL fie constanta 0 (indicând astfel că ultimul nod nu are nici un succesor).

Tipul unui nod într-o listă simplu înlănţuită se poate defini folosind o declaraţie de forma:

```
class Lista;
class ElementLista
{ TINFO info;
  ElementLista *urm;
public:
  ElementLista(int val=0);
  friend class Lista;
};
```

Listele înlănţuite cu 2 câmpuri de legătură se numesc *liste dublu înlănţuite* (o legătură indică nodul precedent iar cealaltă nodul succesor).

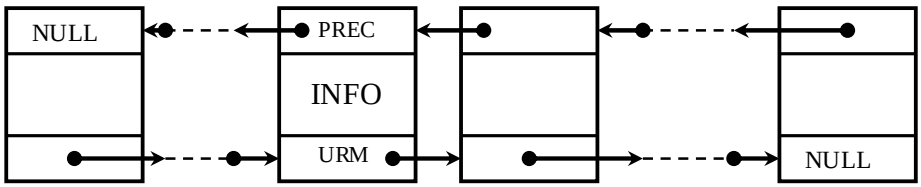


Figura 10.2 *Lista dublu înlănţuită*

Nodurile unei liste dublu înlănţuite au tipul definit după cum urmează, pointerul *urm* definind relaţia de succesor pentru nodurile listei, iar *prec* pe cea de predecesor:

```
class Lista;
class ElementLista
{ ElementLista *prec;
  TINFO info;
  ElementLista *urm;
public:
  ElementLista(int val=0);
  friend class Lista;
};
```

În situaţia în care se realizează o închidere a înlănţuirilor, pierzându-se astfel noţiunile de început şi de sfârşit ale unei liste liniare, se obţine o *listă circulară* simplu sau dublu înlănţuită.

Într-o listă circulară simplu înlănţuită toate nodurile sunt echivalente, fiecare nod având un următor şi fiecare la rândul său constituind următorul unui alt nod. Acest considerent este valabil şi pentru listele circulare dublu înlănţuite extinzându-se şi asupra relaţiei de precedenţă care există între nodurile listei.

Relativ la o poziţie *K* din cadrul unei liste, putem defini următoarele tipuri de operaţii:

- accesul la cel de-al *K*-lea nod al listei pentru examinare sau modificare;
- ştergerea nodului aflat pe poziţia *K* din cadrul listei;
- inserarea unui nou nod înainte de, respectiv după nodul *K*.

Limitând efectuarea acestor trei operaţii la primul sau la ultimul nod al unei liste liniare simplu înlănţuite se obţin structurile de date derivate de tip *stivă* şi *coadă*. În ceea ce priveşte stiva, aceasta constituie o listă simplu înlănţuită în care toate inserările şi suprimările de noduri se efectuează la un singur capăt al acesteia, numit *vârful stivei*. Stivele se mai numesc structuri listă de tip LIFO, *Last-In-First-Out*, adică *ultimul-introdus-primul-suprimat* sau liste de tip *pushdown*.

Relativ la structura de date de tip coadă, trebuie specificat că elementele noi vor fi inserate la un capăt al acesteia, numit *spate*, iar suprimările de noduri vor avea loc la celălalt capăt, numit *faţă*. Din acest motiv, cozile se mai numesc şi liste de tip FIFO, *first-in-first-aut*, adică liste de tip *primul-venit-primul-servit*.

Prezenţa mai multor înlănţuiri într-un acelaşi nod conferă mai multă flexibilitate structurilor de date de tip listă, acestea devenind aşa-numitele *multiliste*, nodurile acestora aparţinând în acelaşi timp la mai multe liste înlănţuite simple.

10.2. Lista simplu înlănţuită

10.2.1 Crearea listelor. Inserţia nodurilor

Într-o astfel de listă există întotdeauna un nod care nu mai are succesor, precum şi un nod care nu este succesorul nici unui alt nod, aceste noduri constituind capetele listei simplu înlănţuite. Într-o primă variantă de gestionare a acestui tip de listă vom apela la doi pointeri *prim* şi *ultim* pentru a referiri nodurile terminale ale listei.

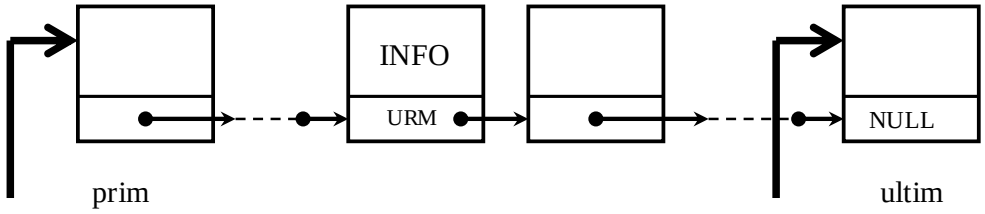


Figura 10.3 *Lista simplu înlănţuită marcată de pointerii prim şi ultim*

Dacă specificarea începutului listei prin pointerul *prim* este obligatorie, utilizarea pointerului de sfârşit *ultim* este opţională, însă eficientă atunci când se doreşte crearea listei în ordine naturală printr-o

inserţie a noilor noduri la sfârşitul acesteia. În practică, parcurgerea întregii liste pentru a determina ultima poziţie a acesteia este o soluţie ineficientă, convenabilă fiind cunoaşterea, cu ajutorul pointerului *ultim*, a nodului terminal al acesteia. În prezenţa lui *ultim*, secvenţa de program care inserează un nod la sfârşitul unei liste liniare şi concomitent îl actualizează pe *ultim* este următoarea:

```
void Lista::inserare_sfarsit(TINFO val)
{
  ElementLista* ptr=new ElementLista(val);
  if (ptr==NULL)
  { cout<<"Eroare alocare spatiu la inserare";
    return;
  }
  Ultim->urm=ptr;
  Ultim=ptr;
  if (Prim==NULL) Prim=Ultim;
  cout<<"Inserare la sfarsitul listei cu succes!\n";
}
```

Se observă că după adăugarea unui nou element la sfârşitul listei, se actualizează pointerul Ultim, care va păstra adresa noului element ultim al listei.

Dacă elementul adăugat la sfârşitul listei este primul element al listei (caz în care pointerul Prim conţine valoarea NULL), vom actualiza şi valoarea pointerului Prim, elementul adăugat fiind în acelaşi timp primul şi ultimul element al listei.

Dacă se doreşte crearea unei liste prin inserţia noilor noduri la începutul acesteia, se va apela metoda *inserare_inceput*, aceasta fiind valabilă şi în cazul unei liste vide.

```
void Lista::inserare_inceput(TINFO val)
{
  ElementLista* ptr=new ElementLista(val);
  if (ptr==NULL)
  { cout<<"Eroare alocare spatiu la inserare";
    return;
  }
  if(Prim==NULL) Ultim=ptr;
  ptr->urm=Prim;
  Prim=ptr;
  cout<<"Inserare la inceputul listei cu succes!\n";
}
```

Dacă elementul adăugat la începutul listei este primul element al listei (caz în care pointerul Prim conţine valoarea NULL), vom actualiza şi valoarea pointerului Ultim, elementul adăugat fiind în acelaşi timp ultimul şi primul element al listei.

Operaţia de inserare a unui nod într-o listă nu se reduce numai la cele două situaţii prezentate, ci implică de asemenea posibilitatea inserării

acestuia într-o poziţie oarecare a listei, şi anume, după sau înaintea unui anumit nod specificat prin referinţa sa *spec*. Schematic cele două situaţii sunt prezentate în figurile următoare, observându-se o oarecare dificultate la operaţia de inserţie a noului nod înaintea celui specificat.

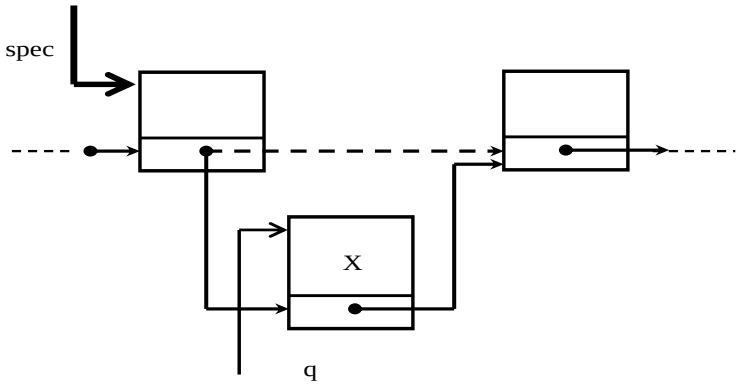


Figura 10.4 *Inserţia unui nod după un nod specificat*

```
void Lista::inserare_dupa_un_element(TINFO X, TINFO cheie)
{
  ElementLista* q, *spec;
  for(spec=Prim;spec&&spec->info!=cheie;spec=spec->urm);
  if(spec)
  {
    q=new ElementLista(X);
    if (q==NULL)
    {
      cout<<"Eroare alocare spatiu la inserare";
      return;
    }
    if(spec==Ultim) Ultim=q;
    q->urm=spec->urm;
    spec->urm=q;
    cout<<"Inserare dupa element cu succes!\n";
  }
  else cout<<"Nu exista cheia specificata!\n";
}
```

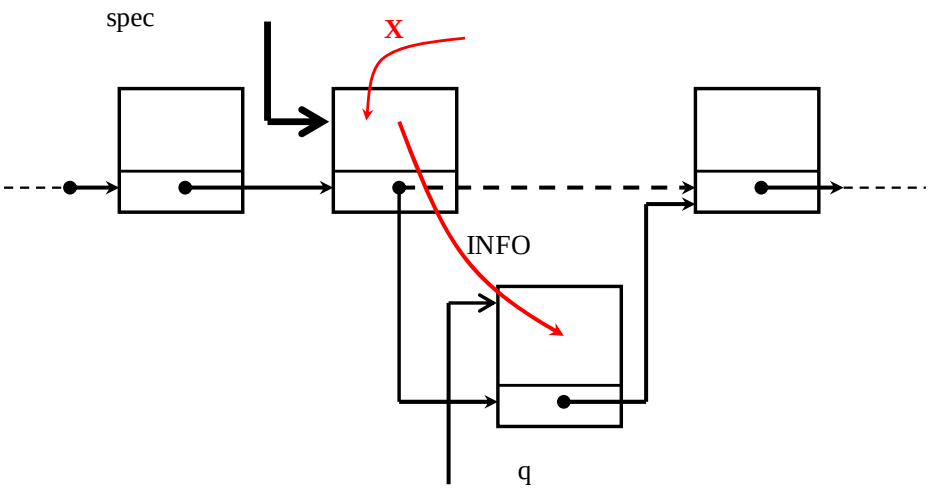


Figura 10.5 *Inserarea unui nod înaintea unui nod specificat*

Inserarea noului nod înaintea celui referit prin *spec* se implementează ca o inserare după acesta, noul nod preluând conţinutul câmpului *info* al acestuia, în care apoi se va memora informaţia *X*.

```
void Lista::inserare_inaintea_unui_elem(TINFO X,TINFO cheie)
{
  ElementLista* q, *spec;
  for(spec=Prim;spec&&spec->info!=cheie;spec=spec->urm);
  if(spec)
  {
    q=new ElementLista(X);
    if (q==NULL)
    {
      cout<<"Eroare alocare spatiu la inserare";
      return;
    }
    if(spec==Ultim) Ultim=q;
    q->info=spec->info;
    q->urm=spec->urm;
    spec->urm=q;
    spec->info=X;
    cout<<"Inserare inaintea unui element cu succes!\n";
  }
  else cout<<"Nu exista cheia specificata!\n";
}
```

Se pune problema creării unei liste înlănţuite ordonate după câmpul *info*, caz în care inserarea unui nod nou nu trebuie să afecteze relaţia de ordonare existentă. Vom traversa lista utilizând doi pointeri consecutivi şi vom identifică astfel poziţia de inserare a noului nod. Inserarea noului nod se va face dupa nodul referit de pointerul q1.

Cei doi pointeri q_1 şi q_2 avansează simultan de-a lungul listei până când valoarea q_1 -> *info* devine mai mare sau egală cu valoarea de inserat *X*. Inserarea noului nod va avea loc între nodurile referite de q_1 şi q_2.

```
void Lista::inserare_ordonata(TINFO X)
{
  ElementLista *q1=NULL,*q2;
  ElementLista* ptr=new ElementLista(X);
  if (ptr==NULL)
  {
    cout<<"Eroare la alocare spatiu pentru inserare";
    return;
  }
  for(q2=Prim;q2&&q2->info<X;q1=q2,q2=q2->urm);
  if(q2==Prim)
  {
    if(Prim==NULL) Ultim=ptr;
    ptr->urm=Prim;
    Prim=ptr;
}
  else
  {
    if(q2==NULL) Ultim=ptr;
    q1->urm=ptr;
    ptr->urm=q2;
  }
  cout<<"Inserare ordonata cu succes!\n";
}
```

10.2.2 Localizarea unui nod

Identificarea unui nod în cadrul unei liste se poate efectua aplicând *metoda de căutare liniară*, aceasta presupunând parcurgerea elementelor listei, nod cu nod, fie până se localizează nodul dorit, fie până la sfârşitul listei dacă elementul căutat este inexistent.

```
/* caută nodul X în cadrul listei şi returnează
pointerul spre nodul identificat şi 0 în caz
contrar */

ElementLista *Lista::cautare(TINFO X)
{
  ElementLista *q;
  for(q=Prim;q&&q->info!=X;q=q->urm);
  if(q)
```

```
  return(q);
  else
    return(0);
}
```

Să se ofere o implementare a *problemei concordanţei,* inserarea unui nou nod realizându-se la începutul listei. În final, lista va conţine toate "cuvintele" *X* distincte şi numărul de apariţii ale acestora.

```
#include<iostream.h>
class Lista;
class ElementLista
{ int info,contor;
  ElementLista *urm;
public:
  ElementLista(int val=0);
  friend class Lista;
};
class Lista
{
protected:
      ElementLista *Ultim, *Prim;
public:
  Lista()
  { Prim = NULL;
    Ultim = new ElementLista();
  }
  ~Lista();
  void traversare();
  void inserare(int);
};

ElementLista::ElementLista(int val)
{
  info=val;
  contor=1;
  urm=NULL;
}

Lista::~Lista()
{
  ElementLista* ptr=Prim;
  while (Prim)
  {
    Prim=Prim->urm;
    delete ptr;
    ptr=Prim;
  }
}

void Lista::traversare()
{
  ElementLista* p=Prim;
  if (p==NULL)
  cout << "\n Lista este vida! \n";
  else
    while (p!=NULL)
      {
       cout<<p->info<<" numar de aparitii: "<<p->contor<<"\n";
       p=p->urm;
```

```
        }
   cout<<"\n";
}

void Lista::inserare(int X)
{
       ElementLista *p;
       for(p=Prim;p&&p->info!=X;p=p->urm);
       if(p)
              p->contor++;
       Else
       {
              p=new ElementLista(X);
              p->urm=Prim;
              Prim=p;
       }
}

void main()
{
       Lista Listamea;
       int cheie;
       cout<<"Introduceti cheia de inserat:";
       cin>>cheie;
       while(cheie)
       { Listamea.inserare(cheie);
         cout<<"Introduceti cheia de inserat:";
         cin>>cheie;
       }
       cout<<"Inserare incheiata!\n";
       cout<<"\nTraversare lista:\n";
       Listamea.traversare();
}
```

O îmbunătăţire substanţială a procesului de căutare poate avea loc prin aplicarea aşa-numitei *metode de căutare cu reordonare*. Ori de câte ori un *cuvânt* se caută şi se localizează în listă, el va fi mutat la începutul acesteia, astfel încât la proxima apariţie să fie găsit imediat. Cu alte cuvinte, lista se reordonează după fiecare căutare finalizată cu succes. Dacă un nod nu este găsit în listă, atunci el va fi inserat la începutul acesteia.

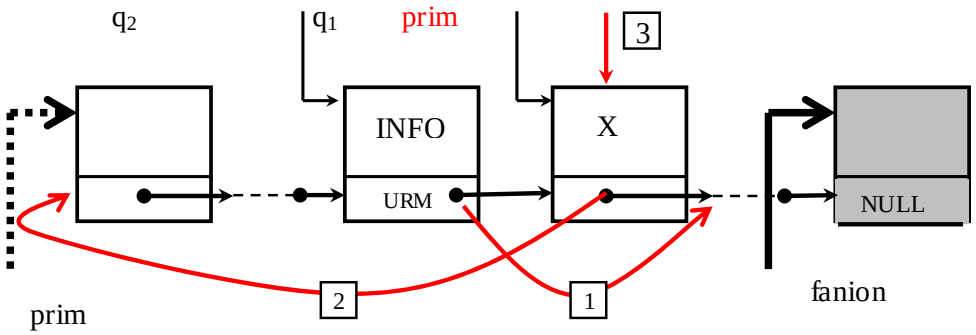


Figura 10.6 *Mutarea nodul X la începutul listei*

```
void Lista::inserare(int X)
{
       ElementLista *q1, *q2=NULL, *p;
       for(q1=Prim;q1&&q1->info!=X;q2=q1,q1=q1->urm);
```

```
      if(q1)
      {
            q1->contor++;
            if(q2)
            {
                  q2->urm=q1->urm;
                  q1->urm=Prim;
                  Prim=q1;
            }
      }
      else
      {
              p=new ElementLista(X);
              p->urm=Prim;
              Prim=p;
      }
}
```

10.2.3 Suprimarea nodurilor

Se propune eliminarea unui anumit nod, referit prin *q1*, din cadrul unei liste liniare înlănţuite. Pentru aceasta vom folosi un pointer auxiliar, q2, care indică predecesorul elementului ce urmează să fie suprimat .

```
void  suprima (TNOD *q);
{
  ElementLista *q1=NULL,*q2;
  for(q2=Prim;q2&&q2->info!=X;q1=q2,q2=q2->urm);
  if(q2)
  {
    if(q2==Prim) Prim=q2->urm;
    if(q2==Ultim) Ultim=q1;
    if(q1) q1->urm=q2->urm;
    delete(q2);
    cout<<Suprimare cu succes!\n";
  }
  Else
    cout<<"Nodul nu exista!\n";
}
```

10.3 Lista circulară simplu înlănţuită

Listele circulare sunt liste înlănţuite ale căror înlănţuiri se închid, în aceste condiţii dispărând noţiunea de început şi de sfârşit al listei. Pentru gestiunea unei astfel de lista, vom păstra un pointer spre „ultimul” element al listei.

```
# include <iostream.h>
# define TINFO int
class Lista;
class ElementLista
{ TINFO info;
  ElementLista *urm;
```

```
public:
  ElementLista(int val=0);
  friend class Lista;
};

class Lista
{
protected:
      ElementLista *Ultim;
public:
  Lista()
  { Ultim = NULL;}
  ~Lista();
  void traversare();
  void inserare_sfarsit(TINFO);
  void inserare_inaintea_unui_elem(TINFO,TINFO);
  ElementLista *cautare(TINFO);
  void suprimare(TINFO);
 };

ElementLista::ElementLista(TINFO val)
{
  info=val;
  urm=NULL;
}

Lista::~Lista()
{
  ElementLista* ptr;
  if(Ultim!=NULL)
  { while (Ultim->urm!=Ultim)
      {
        ptr=Ultim->urm;
        Ultim->urm=Ultim->urm->urm;
      delete ptr;
      }
    delete Ultim;
  }
}

void Lista::traversare()
{
  ElementLista* ptr=Ultim;
  if (Ultim==NULL)
      cout << "\n Lista este vida! \n";
  else
  { while (ptr->urm!=Ultim)
      {
       ptr=ptr->urm;
          cout << ptr->info<<" ";
      }
      cout << ptr->urm->info<<" ";
      cout<<"\n";
  }
}

void Lista::inserare_sfarsit(TINFO val)
{
      ElementLista* ptr=new ElementLista(val);
      if (ptr==NULL)
      {
```

```
            cout<<"Eroare la alocare spatiu pentru inserare";
            return;
      }
      if(Ultim)
      {
            ptr->urm=Ultim->urm;
            Ultim->urm=ptr;
            Ultim=ptr;
      }
      else
      {
            Ultim=ptr;
            ptr->urm=ptr;
      }
      cout<<"Inserare sfarsit cu succes!\n";
}

void Lista::inserare_inaintea_unui_elem(TINFO X,TINFO cheie)
{
  ElementLista* ptr, *q1=Ultim, *q2;
  if(Ultim==NULL)
  {
        cout<<"Lista vida! Inserare fara succes!\n";
        return;
  }
  for(q2=Ultim->urm;q2!=Ultim&&q2->info!=cheie;q1=q2,q2=q2->urm);
  if(q2->info==cheie)
  {
       ptr=new ElementLista(X);
       if (ptr==NULL)
     {
        cout<<"Eroare alocare spatiu la inserare";
        return;
       }
       q1->urm=ptr;
       ptr->urm=q2;
       cout<<"Inserare cu succes!\n";
  }
  else cout<<"Nu exista cheia specificata!\n";
}

ElementLista *Lista::cautare(TINFO X)
{
      ElementLista *ptr;
      if(Ultim==NULL) return(0);
      for(ptr=Ultim->urm;ptr!=Ultim&&ptr->info!=X;ptr=ptr->urm);
      if(ptr->info==X)
            return(ptr);
      else
            return(0);
}

void Lista::suprimare(TINFO X)
{
      ElementLista *q1=Ultim, *q2;
      if(Ultim==NULL)
      {
            cout<<"Lista vida! Suprimare fara succes!\n";
          return;
      }
      for(q2=Ultim->urm;q2!=Ultim&&q2->info!=X;q1=q2,q2=q2->urm);
```

```
      if(q2->info==X)
      {
            if(q2->urm==q2)
            {
                  Ultim=NULL;
                  delete(q2);
            }
            else
            {
                  if(q2==Ultim) Ultim=q1;
                  q1->urm=q2->urm;
                  delete(q2);
            }
            cout<<"Suprimare cu succes!\n";
      }
      else
            cout<<"Nodul nu exista!\n";
}

void main()
{
    Lista Listamea;
      int opt;
      TINFO valoare,cheie;
      do
      {
        cout<<"\n Optiuni de lucru cu lista:";
        cout<<"\n 1 - Afisare lista";
        cout<<"\n 2 - Inserare element la sfarsitul listei";
        cout<<"\n 3 - Inserare element inaintea unui elem specificat";
        cout<<"\n 4 - Cautarea unui element specificat";
        cout<<"\n 5 - Suprimarea unui element";
        cout<<"\n 9 - Terminare lucru \n\n";
        cout<<"Introduceti optiunea dorita:";
        cin>>opt;
        switch(opt)
            {
            case 1:
                  {
                  cout<<"Traversare lista:";
                  Listamea.traversare();
                  break;
                  }
            case 2:
                  {
                  cout<<"Introduceti elementul de inserat:";
                  cin>>valoare;
                  Listamea.inserare_sfarsit(valoare);
                  break;
                  }
            case 3:
                  {
                  cout<<"Introduceti elementul de inserat:";
                  cin>>valoare;
                  cout<<"Introduceti elem inaintea caruia inseram:";
                  cin>>cheie;
                  Listamea.inserare_inaintea_unui_elem(valoare,cheie);
                  break;
                  }
            case 4:
                  {
```

```
                    cout<<"Introduceti elementul cautat:";
                    cin>>valoare;
                    if(Listamea.cautare(valoare))
                          cout<<"Elementul a fost gasit!\n";
                    else
                          cout<<"Elementul nu exista in lista!\n";
                    break;
                    }
              case 5:
                    {
                    cout<<"Introd elem pe care doriti sa-l suprimati:";
                    cin>>valoare;
                    Listamea.suprimare(valoare);
                    break;
                    }
              case 9:     break;
              default:
                    cout<<"\n Nu exista optiunea! \n ";
              }
        }while (opt!=9);
}
```

10.4 Structuri de date dinamice necontigue

Structurile de date statice sunt cele care se definesc în program, iar în faza de compilare se calculează deplasări pentru fiecare câmp elementar sau grup de câmpuri, astfel încât după editarea de legături orice evaluare de adresă să permită referirea corectă a structurilor sau a componentelor lor.

Caracterul local sau global al variabilelor conduce la o abordare diferenţială a variabilelor statice. Chiar variabilele care se definesc în cadrul unui bloc şi care au caracter local, a căror alocare de memorie şi iniţializare este realizată prin mecanisme proprii mediului de programare, sunt tratate tot ca structuri sau funcţii apelate la cerere de către programatori.

În continuare, se grupează sub denumirea de structuri dinamice acele structuri pentru care alocarea şi dezalocarea memoriei este gestionată de programator.

Funcţiile care alocă memorie au ca parametri acele informaţii care conduc la răspunsuri neambigue la întrebările:

- care este lungimea zonei de memorie care se alocă?
- unde este stocată adresa zonei de memorie alocată, pentru a fi la dispoziţia programatorului?
- se iniţializează zona de memorie alocată înainte de a fi utilizată de programator?

Mecanismul de alocare dinamică a memoriei, are la bază faptul că în memoria internă a unui sistem de calcul, apare o zonă disponibilă delimitată prin $[D_i, D_f] \cap N$, unde:

- D_i – adresa de început a zonei;
- D_f – adresa de sfârşit a zonei.

La intrarea în execuţie a programului, un registru RO, sau o zonă de memorie fixată, se iniţializează cu valoarea D_i.

Dacă într-un program este activată funcţia de alocare definită prin:

$$y = alocare\ (lg\ (T_i))\qquad (10.1)$$

şi

$$\text{alocare}: \mathrm{N} \rightarrow [\,\mathrm{D_i}, \mathrm{D_f}\,] \cap \mathrm{N} \qquad (10.2)$$

$$\mathrm{y} = \text{cont}\,(\mathrm{RO}), \quad \text{unde tip}\,(\mathrm{y}) = (\text{pointer}, \mathrm{T_i}) \qquad (10.3)$$

$$RO = RO + lg\,(T_i) \qquad (10.4)$$

Dacă:

$$cont\,(RO) > D_f \Rightarrow y = 0 \qquad (10.5)$$

Dacă într-un program este activată funcţia de dezalocare a memoriei:

$$z = dezalocare\,(y) \qquad (10.6)$$

$$RO = RO - lg\,(T_i) \qquad (10.7)$$

Şi în cazul alocării, dacă:

$$tip\,(y) = (pointer, T_j) \qquad (10.8)$$

şi variabila *x* din

$$alocare\,(lg\,(x)) \qquad (10.9)$$

are tipul T_i, se impune efectuarea unei conversii de tip.

Astfel,

$$\mathrm{y} = \text{convtp}\,(\text{tip}(\mathrm{y}), \text{tip}\,(\text{alocare}\,(\mathrm{x}))) \qquad (10.10)$$

unde, *convtp( )* este o funcţie de conversie a tipului de dată T_p.

Algoritmii de structurare a memoriei în pagini, modul în care se face încărcarea programelor, precum şi momentele uneori aleatoare din program la care se apelează funcţiile de alocare/dezalocare, conduc în cele mai multe cazuri ca, pentru două apelări consecutive ale funcţiei de alocare, zonele alocate să fie necontigue.

Necesitatea definirii în programe a datelor de tip dinamic, este dată de utilizarea mai bună a memoriei, lungimea unui program variind în funcţie de volumul datelor cu care se lucrează.

În programe C/C++, pentru alocarea dinamică de memorie, se apelează proceduri din biblioteca *malloc.h* sau operatorul *new*, iar pentru dezalocare, procedura *free( )* sau operatorul *delete*. Ca argumente, operatorul *new* are tipul de dată şi numărul de elemente pentru care se face alocarea zonei de memorie, iar *free( )* şi *delete* au o variabilă pointer, care indică adresa zonei ce se eliberează.

Se consideră un program *P* pentru calculul inversei unei matrice. Pentru a-i oferi o mai largă aplicabilitate, se consideră că acest program este definit aşa fel încât să inverseze o matrice cu cel mult 50 linii şi 50 coloane.

Definirea statică a matricei:

```
int a[50][50];
```

determină ocuparea unei zone de *2500 * lg (int)* baiţi.

Deci:

$$\lg(P) = 2500 * \lg(\text{baiti}) + \lg(\text{text executabil}) + \alpha \qquad (10.11)$$

unde α reprezintă lungimea totală a zonei de memorie rezervată altor variabile de control sau de lucru.

Indiferent de problema de rezolvat, *lg(P) = constant*.

Se ştie că în lucru multiuser, un factor care influenţează aşteptarea într-un fir, înainte de intrarea în prelucrare, este şi lungimea programului.

În contextul definirii matricei ca operand alocat dinamic, la execuţia de fiecare dată a programului, se introduce dimensiunea matricei de inversat, folosind variabila *n* de tip întreg.

Lungimea programului pentru rezolvarea problemei P_i, se determină dinamic:

$$\lg(P_i) = n * \lg(\text{integer}) + \lg(\text{text executabil}) + \alpha \qquad (10.12)$$

şi se are în vedere că:

$$lg(P_i) \Leftarrow lg(P) \qquad (10.13)$$

ştiut fiind faptul că zona la dispoziţia alocării dinamice este $[D_i, D_f] \cap N$, finită.

Restricţiile impuse asupra domeniul pe care este definită funcţia *alocare( )*, determină filozofia întregului proces de construire a variabilelor dinamice.

În cazul în care:

$$\text{alocare}: Z \rightarrow [D_i, D_f] \cap N \qquad (10.14)$$

este creată posibilitatea efectuării de reacoperiri de zone, deci se generează mecanisme de realizare a uniunii de structuri dinamice de date.

De exemplu, se consideră structurile:

```
int a[10];
int b[10];
```

şi variabilele *pa* şi *pb* de tip T_p = *(pointer, int)*, prin:

```
pa = alocare (10 * lg (int))
pb = alocare (7 * lg (int))
pb = alocare (-3 * lg (int))
```

se obţinute reuniunea cu modelul grafic:

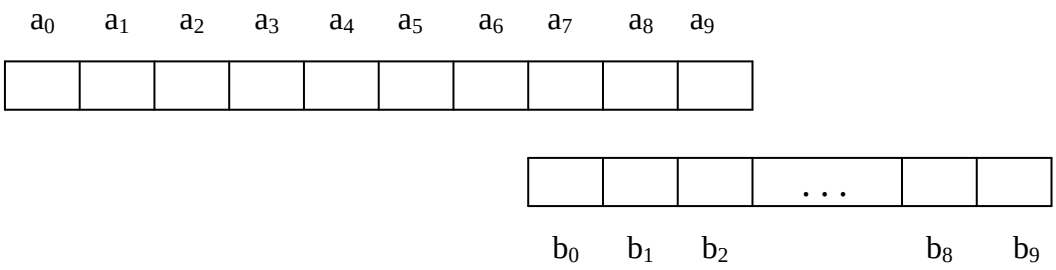


Figura 10.7 *Modelul grafic al reuniunii*

În cele mai multe cazuri, modul de definire al funcţiilor de alocare este limitativ, dar reuniunea este totuşi posibilă prin aritmetica variabilelor pointer, care se iniţializează cu aceste funcţii.

Structurile de date formate din elementele omogene $E_1, E_2, ..., E_m$, de un tip derivat sau fundamental *TE*, se numesc necontigue, dacă există cel puţin o pereche (E_i, E_{i+1}) astfel încât:

$$adr(E_{i+1}) > adr(E_i) + lg(TE) \quad (10.15)$$

şi dacă:

$$adr(E_{i+1}) = adr(E_i) + lg(TE) + \varepsilon_i \quad (10.16)$$

unde ε e o variabilă aleatoare, de regulă uniform distribuită, pentru a avea acces la elementele $E_1, E_2, ..., E_m$, sub o formă care să permită adresarea corectă a elementelor necontigue.

Există posibilitatea prin definirea unor pointeri spre pointeri şi iniţializarea corespunzătoare a acestora, să se procedeze la reaşezarea operanzilor alocaţi dinamic, în aşa fel încât să dispară *golurile* dintre operanzi, rezultate în procesul de alocare/dezalocare.

Dacă se consideră constantele $C_1, C_2, ..., C_n$ având tipul T_i, care ocupă zonele de memorie $Z_1, Z_2, ..., Z_n$, şi numerele aleatoare $\theta_1, \theta_{2, ..., } \theta_n$ a căror lege de distribuţie este, de regulă, uniform distribuită, se spune că mulţimile de perechi (Z_j, θ_j) determină o structură de date necontiguă de tip listă, dacă:

$$succ(Z_j) = Z_j + 1 \quad (10.17)$$

$$pred(Z_j) = Z_j - 1 \quad (10.18)$$

şi

$$adr(Z_{j+1}) = adr(Z_j) + lg(T_i) + \varepsilon_j = \theta_j \quad (10.19)$$

Dacă:

$$\varepsilon_1 = \varepsilon_2 = ... = \varepsilon_n \quad (10.20)$$

atunci se obţine cazul particular de dată unidimensională cu necontiguitate nulă, ce corespunde masivului unidimensional contiguu, respectiv vectorul,

şi zona de memorie pentru conservarea variabilei θ_i nu se mai justifică, întrucât ea este calculabilă ca:

$$\theta_j = adr(Z_j) + lg(T_i) = adr(Z_1) + (j-1)*lg(T_i) \qquad (10.21)$$

Dacă se ia în considerare mecanismul alocării dinamice a memoriei, atunci:

$$\theta_j = cont(RO) \qquad (10.22)$$

la momentul t_j, ce corespunde alocării zonei de memorie pentru perechea (Z_{j+1}, θ_{j+1}).

Privită din punct de vedere al tipului de dată, Z_j reprezintă informaţia utilă, iar θ_j reprezintă informaţia de identificare a succesorului.

$$adr(succ(Z_i)) = \theta \qquad (10.23)$$

Perechea (Z_j, θ_j) defineşte o structură de date de tip listă, numită *TL*, unde θ_j este dată de tip T_p = (pointer, TL).

10.5 Modelul grafic al listei ca structură necontiguă

Se consideră necesar ca o matrice rară să fie memorată fără a se cunoaşte în prealabil numărul elementelor nenule ale sale.

Rând pe rând, se introduce de la terminal linia, coloana şi valoarea elementului nenul. Se alocă dinamic o zonă de memorie pentru stocarea acestor elemente, precum şi un câmp pentru stocarea adresei zonei în care se stochează elementul următor.

```
class zona
{
      int i;
      int j;
      float val;
      zona *poz;
};
zona * pa;
```

Folosind succesiv operatorul *new*:

```
pg = new  zona;
```

unde *pg* este o variabilă de tip *pa*, iar *pa* la rândul ei este un pointer spre structura *zona*, se alocă o zonă de memorie de adresa *cont(RO)* şi lungime $lg(T_{zona})$, ce este referită folosind variabila pointer *pg*, care este de tipul T_p = (pointer, T_{zona}).

În program, membrii structurii se referă prin:

```
pg->i
pg->j
```

```
pg->val
```

Dacă *pn* este variabilă de tip *pa* şi se efectuează atribuirea *pn = new zona*, variabila *pn* conţine adresa zonei în care este stocat următorul element al matricei rare.

Atribuirea:

```
pg->poz = pn
```

este echivalentă cu:

$$\theta_j = adr(succ(Z_j)) \qquad (10.24)$$

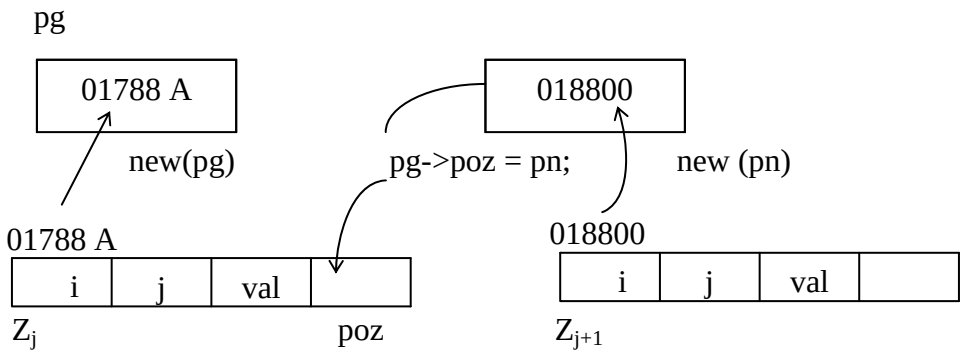


Figura 10.8 *Alocarea zonelor de memorie*

Dacă:

$$succ(Z_n) = \theta \qquad (10.25)$$

atunci:

$$adr(succ(Z_n)) = NULL \qquad (10.26)$$

$$pg->poz->poz = NULL; \qquad (10.27)$$

Acest algoritm de construire a şirului θ_1, $\theta_{2, \ldots,}$ θ_n conduce la modelul grafic:

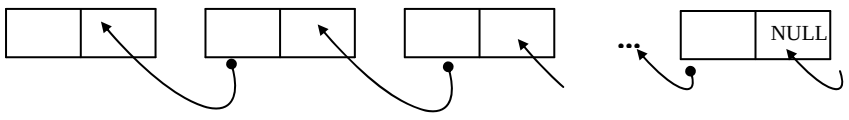


Figura 10.9 *Modelul grafic al listei simplu înlănţuită*

O astfel de listă se numeşte listă liniară simplu înlănţuită.

Încărcarea matricei rare se face exact cu atâtea elemente câte valori nenule se află pe linii şi coloane. Spre deosebire de cazul în care se foloseau 3 vectori pentru stocarea informaţiilor, acum nu mai există restricţii legate de rezervările predefinite ale celor 3 vectori.

Totul depinde de modul în care programul solicită spaţii din zonă $[D_i, D_f] \cap N$ şi mai ales de dimensiunile proiectate ale acestei zone la generarea sistemului de operare.

Dacă în loc de:

$$adr(succ(Z_n)) = NULL \quad (10.28)$$

se construieşte:

$$adr(succ(Z_n)) = adr(Z_1) \quad (10.29)$$

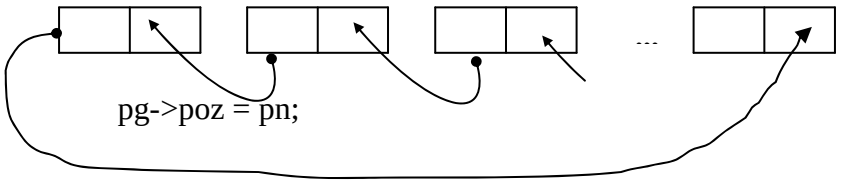
lista se numeşte circulară şi θ_n <> *NULL*.

Pentru transformarea listei care stochează matricea rară în listă circulară, la terminarea prelucrării:

```
pn->poz = pg;
```

unde *pg* stochează adresa zonei de memorie alocată pentru primul element nenul al matricei rare.

Modelul grafic al listei circulare este:



pg->poz = pg;

Figura 10.10 *Modelul grafic al listei circulare*

Lungimea listei liniare:

$$L = \sum_{i=1}^{\infty} lg[(Z_i, \theta_i)] \quad (10.30)$$

$$\theta_i <> NULL \quad (10.31)$$

unde $lg[Z_i, NULL] = 0$.

10.6 Operaţii cu liste liniare simplu înlănţuite

Parcurgerea listei liniare simplu înlănţuite corespunde funcţiei de extragere a adresei elementului succesor şi de referire a membrilor acestuia.

Pentru tipărirea conţinutului elementelor matricei rare, cu număr necunoscut de elemente nenule, funcţia de parcurgere, construită recursiv este:

```
parcurgere (pg)
{
      while (pg->poz != NULL)
      {
            tipareste (pg->i, pg->j, pg->val);
            parcurgere (pg->poz);
      }
}
```

Considerând lista ca mulţimea de perechi de forma *(Z, θ)* de tip *TL*,

$$w = adr(Z_1) \qquad (10.32)$$

conţine adresele de regăsire a elementelor listei.

```
parcurgere (adr (w))
{
      while (w !=  NULL)
      {
            tipareste (w, z);
            parcurgere (ref (w), θ);
       }
}
```

Ştergerea unui element (Z_k, θ_k) al listei

Înainte de ştergere lista este:

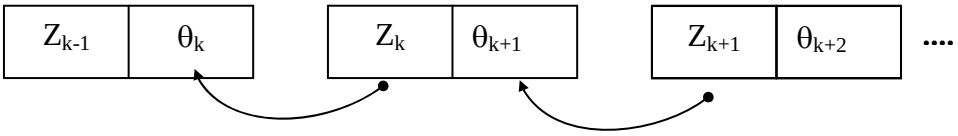


Figura 10.11 *Lista înainte de ştergere*

După efectuarea ştergerii:

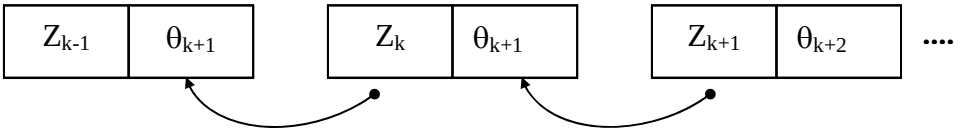


Figura 10.12 *Lista după de ştergerea elementului*

```
stergere ( w, w1, Zk )
{
      while (ref (w).θ != NULL || ref (w).Z != Zk)
            {
                  w1 = ref (w, θ);
                  stergere (w1, w, Zk)
            }
```

```
        if( ref (w1).Z = =Zk)
              ref (w).θ = ref (w1).θ;
}
```

Concatenarea a două liste

Concatenarea a două liste *(Z, θ)* şi *(U, θ)*, reprezintă ca ultimul element al primei liste să-şi schimbe valoarea din NULL a lui θ_n cu adresa elementului (U_1, θ_1).

Deci:

$$ref(Z_n).\ \theta_n = adr[(U_1, \theta_1)] \qquad (10.33)$$

Funcţia care efectuează concatenarea listelor este:

```
concatenare((Z, θ), (U, θ))
{
      while (ref(Z).θ != NULL)
      {
            concatenare ((ref(Z).θ,θ),(U,Z));
      }
      ref(Z).θ = adr((U,θ))
}
```

Modelul grafic al concatenării:

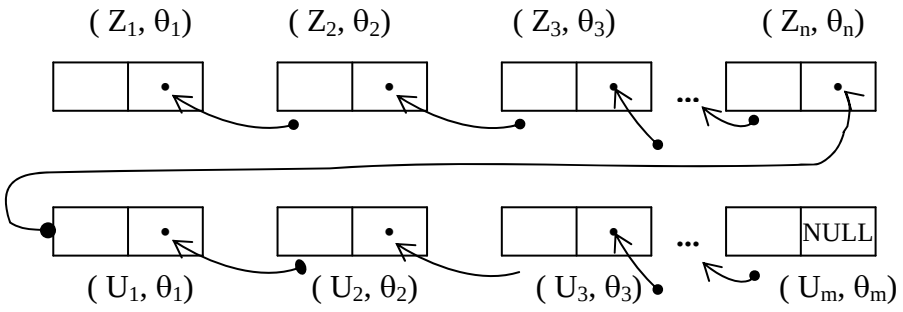


Figura 10.13 *Modelul grafic al concatenării listelor*

Lista concatenată are ca prin element (Z_1, θ_1), iar ca ultim element (U_m, θ_m).

Fizic, lista *(U, θ)* nu s-a modificat. Conservând (Z, θ_1) şi (U, θ_1) se lucrează cu cele două liste, ca şi cum concatenarea nu s-a executat. Totuşi lista concatenată, pentru operaţia de parcurgere se comportă ca o listă cu $m + n$ componente.

Modificarea unui element al listei

Fie lista (Z_i, θ_i), $i = 1, 2, ..., n$. Pentru înlocuirea unei valori *a* cu valoarea *b*, trebuie mai întâi găsit elementul θ_k pentru care:

$$cont(Z_k) = a \qquad (10.34)$$

după care se realizează atribuirea:

$$Z_k = b; \qquad (10.35)$$

În toate cazurile, parcurgere, ştergere, concatenare, modificare, disciplina de parcurgere este de la primul element către ultimul, *First In – First Out*.

```
modificare (w)
{
      if (ref (w). z != a)
            modificare (ref (w).θ);
      else
            ref (w). z = b;
}
```

Copierea unei liste

Fie lista (Z_j, θ_j), $j = 1, 2, ..., n$. Se pune problema obţinerii unei liste:

$$(Z'_j, \theta'_j),\ j = 1, 2, \ldots, n \qquad (10.36)$$

astfel încât:

$$cont(Z'_j) = cont(Z_j) \qquad (10.37)$$

pentru $j = 1, 2, ..., n$.

```
copiere_lista (w, u)
{
      while (ref (w).θ != NULL)
      {
            ref (u).Z = ref (w).Z;
            alocare (v);
            ref (u).θ = v;
            copiere_lista (ref (w).θ), v);
      }
      ref (u). θ = NULL;
}
```

Inserarea unui element în listă

Se spune că o listă este ordonată crescător dacă:

$$\text{cont}(Z_j) \Rightarrow \text{cont}(Z_j + 1) \qquad (10.38)$$

pentru orice $j = 1, 2, \ldots, n - 1$.

A insera un element într-o listă, înseamnă mai întâi a găsi o valoare k $\in$ {1, 2, …, n} astfel încât:

$$\text{cont}(Z_k) \Leftarrow a \Leftarrow \text{cont}(Z_k + 1) \qquad (10.39)$$

sau

$$\text{cont}(Z_k) \Rightarrow a \Rightarrow \text{cont}(Z_k + 1) \qquad (10.40)$$

după cum lista este ordonată crescător sau descrescător. În aceste condiţii, inserarea elementului *a*, înseamnă conform modelului grafic, a trece de la configuraţia:

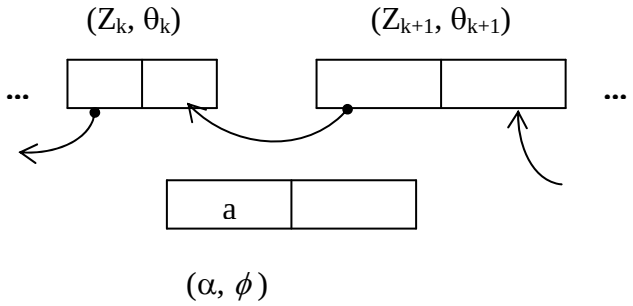


Figura 10.14 *Configuraţia înainte de inserare nodului în interiorul listei*

la configuraţia:

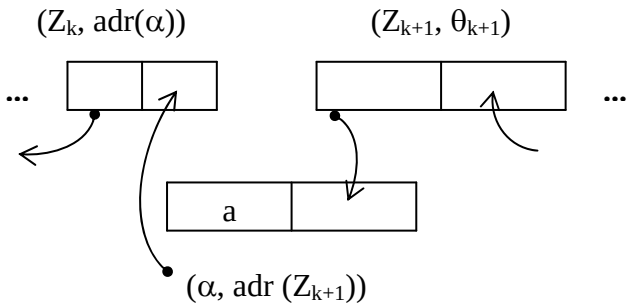


Figura 10.15 *Configuraţia după inserarea nodului în interiorul listei*

Există cazuri particulare de inserare în care elementul este poziţionat fie la începutul listei, fie la sfârşitul acesteia, operaţia numindu-se adăugare.

Dacă elementele *a* şi *b* vor fi inserate la începutul, respectiv, la sfârşitul listei, se trece de la configuraţia:

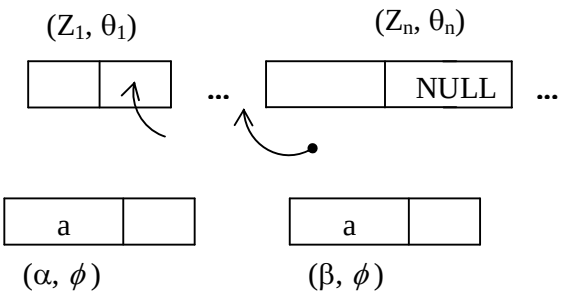


Figura 10.16 *Configuraţia înainte de inserarea nodului la un capăt al listei*

la configuraţia:

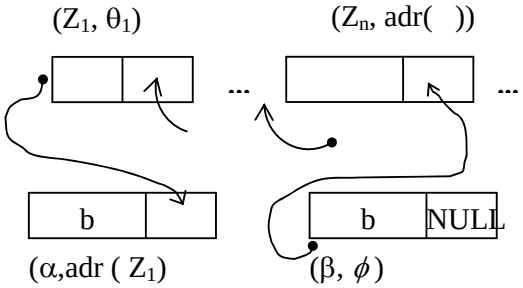


Figura 10.17 *Configuraţia după inserarea nodului la un capăt al listei*

Interschimbul între două elemente ale listei

Interschimbul nu se realizează fizic, zonele ce corespund celor două elemente modificându-şi doar adresele de referire a elementelor.

Modelul grafic al listei înainte de interschimbul elementelor (Z_k, θ_k) şi (Z_j, θ_j) este:

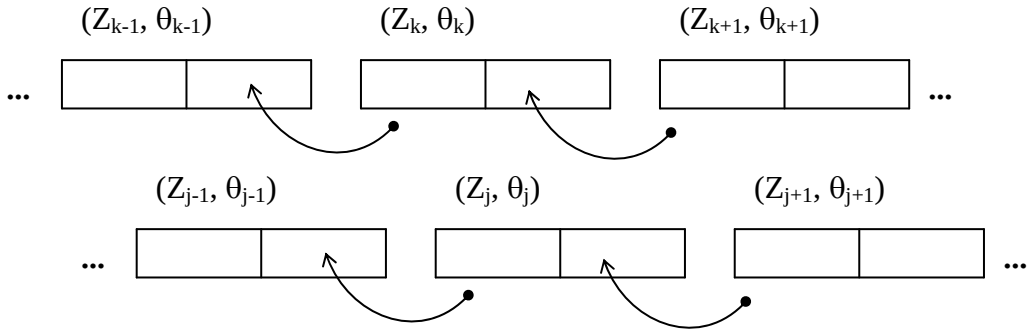


Figura 10.18 *Modelul grafic al listei înainte de interschimbul nodurilor*

După efectuarea interschimbului, legăturile dintre componente sunt:

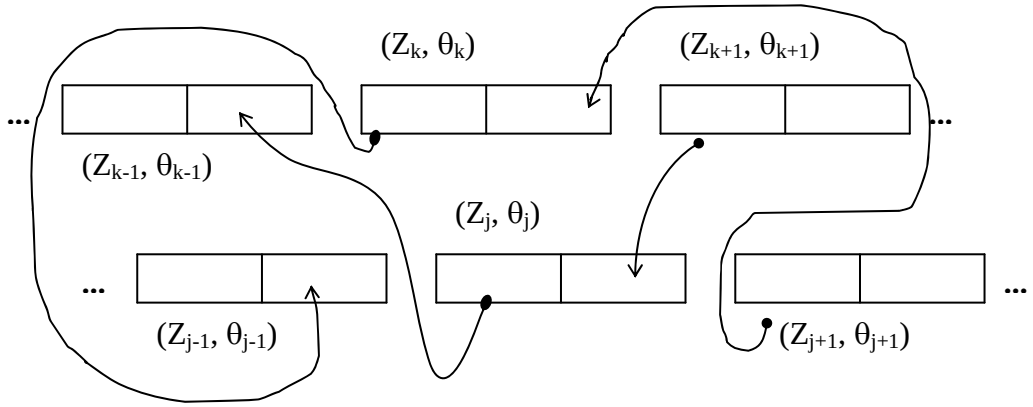


Figura 10.19 *Modelul grafic al listei după interschimbul nodurilor*

Funcţia pentru efectuarea interschimbului, realizează atribuirile:

$$\theta_{k-1} = adr(\ Z_j\ ) \quad (10.41)$$

$$\theta_{j-1} = adr(\ Z_k\ ) \quad (10.42)$$

$$\theta_j = adr(\ Z_{k+1}\ ) \quad (10.43)$$

$$\theta_k = adr(\ Z_{j+1}\ ) \quad (10.44)$$

ceea ce înseamnă că la un moment dat sunt gestionate şase adrese de variabile de tip *TL*, ale elementelor ce se interschimbă, precum şi a elementelor adiacente.

Sortarea elementelor unei liste

Fiind dată o structură de date de tip listă (Z_j, θ_j), $j = 1, 2, ..., n$, funcţia de sortare transformă această structură de date într-o nouă listă (Z'_k, θ'_k), $k = 1, 2, ..., n$ astfel încât oricărui $k \in [1, n] \subset N$ îi corespunde un $j \in [1, n] \subset N$ şi numai unul aşa încât:

$$cont(\ Z'_k\ ) = cont(\ Z'_j\ ) \quad (10.45)$$

şi

$$\text{cont}\,(\,Z_{k'}\,) \Leftarrow \text{cont}\,(\,Z_k + 1'\,) \quad (10.46)$$

pentru orice k = 1, 2, …, n – 1.

Funcţia de sortare apelează la rândul ei funcţia de interschimb a două elemente adiacente, până când în final se obţine ordinea crescătoare sau descrescătoare a termenilor Z_i din lista iniţială.

Un exemplu simplu de sortare, fără a lua în considerare multitudinea de tehnici este:

```
sortare (w)
{
      k = 1;
```

```
        while (k != 0)
        {
                k = 0;
                while (ref (w).θ != NULL)
                {
                        if (ref (w).Z > ref (w).ref(θ).Z)
                        {
                                k = 1;
                                interschimb (w, w.θ);
                        }
                }
        }
}
```

10.7 Liste dublu înlănţuite

Spre deosebire de listele simplu înlănţuite care permit parcurgerea de la primul element spre ultimul alocat dinamic, listele dublu înlănţuite realizează şi drumul invers, permiţând şi parcurgerea de la ultimul element către primul element.

Modelul grafic al listei dublu înlănţuite este:

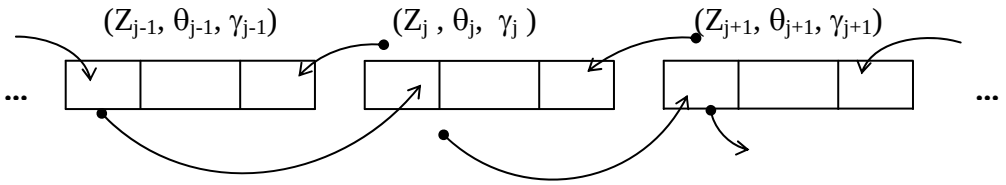


Figura 10.20 *Model grafic al listei dublu înlănţuite*

sau:

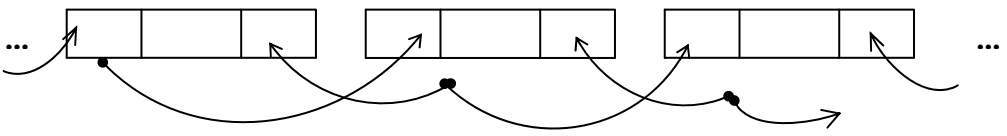


Figura 10.21 *Model grafic al listei dublu înlănţuite*

Lista dublu înlănţuită este de fapt formată din două liste (Z_j, θ_j) şi (U_j, γ_j) cu proprietăţile:

$$\begin{aligned}
&\text{adr}(Z_j) = \text{adr}(U_j) \Rightarrow \text{cont}(Z_j) = \text{cont}(U_j)\\
&\text{adr}(\theta_j) = \text{adr}(\gamma_j) + \lg(\gamma_j)\\
&\text{cont}(Z_j, \theta_j) = \text{adr}(Z_j + 1)\\
&\text{cont}(Z_j, \gamma_j) = \text{adr}(Z_j - 1)\\
&\text{cont}(\theta_{j-1}) = \text{cont}(\gamma_{j+1})
\end{aligned} \qquad (10.47)$$

Şi cu listele dublu înlănţuite se efectuează operaţii precum:

- inserarea unui element;
- adăugarea unui element;
- ştergerea unui element;
- inversarea a două elemente;
- ştergerea listei;
- parcurgerea într-un sens şi în sensul opus;
- transformarea listei în listă circulară dublu înlănţuită.

Un exemplu de creare, inserare, căutare, parcurgere şi ştergere a unei liste dublu înlănţuite, este următorul program:

```
# include <iostream.h>
# define TINFO int
class ListaDubluInlan;

class ElementLista
{ TINFO info;
  ElementLista *pred, *suc;
public:
  ElementLista(int val=0);
  friend class ListaDubluInlan;
};

class ListaDubluInlan
{
protected:
      ElementLista *Prim;
public:
  ListaDubluInlan()
  { Prim = NULL;}
  ~ListaDubluInlan();
  void traversare_inainte();
  void inserare_inceput(TINFO);
  void inserare_inaintea_unui_elem(TINFO,TINFO);
  void inserare_dupa_elem(TINFO,TINFO);
  ElementLista *cautare(TINFO);
  void suprimare(TINFO);
 };

ElementLista::ElementLista(TINFO val)
{
  info=val;
  pred=suc=NULL;
}

ListaDubluInlan::~ListaDubluInlan()
{
  ElementLista* ptr;
  while (Prim)
      {
        ptr=Prim;
        Prim=Prim->suc;
        delete ptr;
      }
}

void ListaDubluInlan::traversare_inainte()
{
  ElementLista* ptr=Prim;
```

```
  if (Prim==NULL)
      cout << "\n Lista este vida!";
  else
    while (ptr)
      {
      cout << ptr->info<<" ";
      ptr=ptr->suc;
      }
  cout<<"\n";
}

void ListaDubluInlan::inserare_inceput(TINFO val)
{
      ElementLista* ptr=new ElementLista(val);
      if (ptr==NULL)
      {
            cout<<"Eroare la alocare spatiu pentru inserare";
            return;
      }
      ptr->pred=NULL;
      ptr->suc=Prim;
      if(Prim)
            Prim->pred=ptr;
      else
            Prim=ptr;
      cout<<"Inserare la inceput cu succes!\n";
}

void ListaDubluInlan::inserare_inaintea_unui_elem(TINFO X,TINFO cheie)
{
  ElementLista* ptr, *p;
  if(Prim==NULL)
  {
       cout<<"Lista vida! Inserare fara succes!\n";
       return;
  }
  for(p=Prim;p&&p->info!=cheie;p=p->suc);
  if(p)
  {
     ptr=new ElementLista(X);
     if (ptr==NULL)
     {
       cout<<"Eroare alocare spatiu la inserare";
       return;
     }
     if(p==Prim)
     {
       ptr->pred=NULL;
       ptr->suc=p;
       p->pred=ptr;
       Prim=ptr;
     }
     else
     {
       ptr->pred=p->pred;
       ptr->suc=p;
       p->pred->suc=ptr;
       p->pred=ptr;
     }
     cout<<"Inserare inainte element cu succes!\n";
  }
```

```
  else cout<<"Nu exista cheia specificata!\n";
}

void ListaDubluInlan::inserare_dupa_elem(TINFO X,TINFO cheie)
{
  ElementLista* ptr, *p;
  if(Prim==NULL)
  {
        cout<<"Lista vida! Inserare fara succes!\n";
        return;
  }
  for(p=Prim;p&&p->info!=cheie;p=p->suc);
  if(p)
  {
       ptr=new ElementLista(X);
       if (ptr==NULL)
       {
          cout<<"Eroare alocare spatiu la inserare";
          return;
       }
       ptr->suc=p->suc;
       ptr->pred=p;
       if(p->suc)
            p->suc->pred=ptr;
       p->suc=ptr;
       cout<<"Inserare dupa element cu succes!\n";
  }
  else cout<<"Nu exista cheia specificata!\n";
}

ElementLista *ListaDubluInlan::cautare(TINFO X)
{
      ElementLista *ptr;
      for(ptr=Prim;ptr&&ptr->info!=X;ptr=ptr->suc);
      if(ptr)
            return(ptr);
      else
            return(0);
}

void ListaDubluInlan::suprimare(TINFO X)
{
      ElementLista *ptr;
      if(Prim==NULL)
      {
            cout<<"Lista vida! Suprimare fara succes!\n";
            return;
      }
      for(ptr=Prim;ptr&&ptr->info!=X;ptr=ptr->suc);
      if(ptr)
      {
            if(ptr==Prim)
            {
                  Prim=Prim->suc;
                  if (Prim) Prim->pred=NULL;
            }
            else
            {
                  ptr->pred->suc=ptr->suc;
                  if(ptr->suc)
                        ptr->suc->pred=ptr->pred;
```

```
            }
            delete(ptr);
            cout<<"Suprimare cu succes!\n";
      }
      else
            cout<<"Nodul nu exista!\n";
}

void main()
{
      ListaDubluInlan Listamea;
      int opt;
      TINFO valoare,cheie;
      do
      {
        cout<<"\n Optiuni de lucru cu lista:";
        cout<<"\n 1 - Afisare lista";
        cout<<"\n 2 - Inserare element la inceputul listei";
        cout<<"\n 3 - Inserare element inaintea unui elem specificat";
        cout<<"\n 4 - Inserare dupa un element specificat";
        cout<<"\n 5 - Cautarea unui element specificat";
        cout<<"\n 6 - Suprimarea unui element";
        cout<<"\n 9 - Terminare lucru \n\n";
        cout<<"Introduceti optiunea dorita:";
        cin>>opt;
            switch(opt)
            {
            case 1:
                  }
                  cout<<"Traversare lista:";
                  Listamea.traversare_inainte();
                  break;
                  }
            case 2:
                  {
                  cout<<"Introduceti elementul de inserat:";
                  cin>>valoare;
                  Listamea.inserare_inceput(valoare);
                  break;
                  }
            case 3:
                  {
                  cout<<"Introduceti elementul de inserat:";
                  cin>>valoare;
                  cout<<"Introduceti elem inaintea caruia inseram:";
                  cin>>cheie;
                  Listamea.inserare_inaintea_unui_elem(valoare,cheie);
                  break;
                  }
            case 4:
                  {
                  cout<<"Introduceti elementul de inserat:";
                  cin>>valoare;
                  cout<<"Introduceti elementul dupa care inseram:";
                  cin>>cheie;
                  Listamea.inserare_dupa_elem(valoare,cheie);
                  break;
                  }
            case 5:
                  {
                  cout<<"Introduceti elementul cautat:";
```

```
                  cin>>valoare;
                  if(Listamea.cautare(valoare))
                        cout<<"Elementul a fost gasit!\n";
                  else
                        cout<<"Elementul nu exista in lista!\n";
                  break;
                  }
            case 6:
                  {
                  cout<<"Introd elem pe care doriti sa-l suprimati:";
                  cin>>valoare;
                  Listamea.suprimare(valoare);
                  break;
                  }
            case 9:     break;
            default:
                  cout<<"\n Nu exista optiunea! \n ";
            }
      }while (opt!=9);
}
```