11. LISTE DINAMICE

11.1. Liste simplu înlănțiute

Lista este o mulțime dinamică, adică are un număr variabil de elemente. La început lista este o mulțime vidă. În procesul execuției progamului se pot adăuga elemente noi listei şi totodată pot fi eliminate diferite elemente din listă de care nu mai este nevoie. Elementele unei liste sunt structuri de acelaşi tip. Se obişnuieşte să se ordoneze elementele unei liste folosind pointeri care intră în compunerea elementelor listei. Datorită acestor pointeri, elementele listei devin structuri recursive. Listele organizate în acest fel se numesc **liste înlănțuite**. Aşadar, o mulțime dinamică de structuri recursive de acelaşi tip, pentru care sunt definite una sau mai multe relații de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numeşte **listă înlănțuită**. În legătură cu listele înlănțuite se au în vedere următoarele aspecte:

- crearea unei liste înlănțuite;
- accesul la un nod oarecare al unei liste înlănțuite;
- inserarea unui nod într-o listă înlănțuită;
- ştergerea unui nod dintr-o listă înlănțuită;
- ştergerea unei liste înlănțuite.

Elementele unei liste se numesc **noduri**. Dacă între nodurile unei liste există o singură relație de ordine, atunci lista se numeşte **simplu înlănțuită**, iar dacă între nodurile listei sunt definite 2 relații de ordine, atunci lista se numeşte **dublu înlănțuită**.

Într-o listă simplu înlănțuită de obicei relația de ordonare este cea de *succesor*, adică fiecare nod conține un pointer a cărui valoare reprezintă adresa nodului *următor* din listă. În mod analog se poate defini relația de *precedent*. Într-o listă simplu înlănțuită pentru care nodurile satisfac relația de succesor există totdeauna un nod şi numai unul care nu mai are succesor, precum şi un nod care nu este succesorul nici unui alt nod. Aceste noduri formează **capetele** listei simplu înlănțuite.

Pentru a gestiona nodurile unei liste simplu înlănțuite, vom utiliza 2 pointeri spre cele 2 capete. Numim **prim** pointerul spre nodul care nu este succesorul nici unui nod al listei şi cu **ultim** pointerul spre nodul care nu are succesor. Aceşti pointeri pot fi definiți fie ca variabile globale în cazul în care nu se gestionează mai multe liste simplu înlănțuite în program, fie ca parametri pentru funcțiile de prelucrare a listei în cazul în care programul gestionează mai multe liste simplu înlănțuite. În cele ce urmează vom considera cazul în care pointerii *prim* şi *ultim* sunt variabile globale.

Tipul unui nod într-o listă simplu înlănțuită se poate defini folosind o declarație de forma:

```
typedef struct tnod {
        declarații
    struct tnod *urm;
} TNOD;
```

Pointerii *prim* şi *ultim* se declară în afara oricărei funcții prin: *TNOD *prim, *ultim;* De obicei ei vor fi declarați înaintea definirii funcției *main.*

Pointerul *urm* defineşte relația de succesor pentru nodurile listei. Pentru fiecare nod el are ca valoare adresa nodului următor din listă cu excepția nodului spre care pointează variabila *ultim* (în acest caz *urm* are valoarea zero (pointerul nul)).

11.1.1. Crearea unei liste simplu înlănțiute

La crearea unei liste simplu înlănțuite se realizează următoarele:

1. Se inițializează pointerii *prim* şi *ultim* cu valoarea zero, deoarece lista la început este vidă.
2. Se rezervă zona de memorie în memoria heap pentru nodul curent.
3. Se încarcă nodul curent cu datele curente,dacă există şi apoi se trece la pasul 4. Altfel lista este creată şi se revine din funcție.
4. Se atribuie pointerului *ultim->urm* adresa din heap a nodului curent, dacă lista nu este vidă. Altfel se atribuie lui *prim* această adresă.
5. Se atribuie lui *ultim* adresa nodului curent.
6. *ultim->urm = 0*
7. Procesul se reia de la punctul 2 de mai sus pentru a adăuga un nod nou în listă.

Pentru a încărca datele curente în nod (punctul 3) vom aplea o funcție care este specifică aplicației. Această funcție poate să aibă un nume dinainte precizat sau să fie transferată printr-un parametru la funcția de creare a listei, dacă programul prelucrează mai multe liste. Acest parametru va fi un pointer spre funcția resprectivă. În cazul de față s-a presupus că programul prelucrează o singură listă şi de aceea vom considera că datele se încarcă prin funcția cu numele *incnod*. Ea returnează: 0 - la eroare; 1 - la încărcare normală a datelor; -1 - cîn nu mai sînt date de încărcat în nod. De exemplu, dacă funcția *incnod* citeşte datele pe care le încarcă în nod dintr-un fişier, atunci ea va returna valoarea -1 la întîlnirea sfîrşitului de fişier (nu mai sînt date de încărcat în nod).

Funcția *incnod* are un singur parametru adresa de memorie heap rezervată pentru nodul curent, deci acest parametru este un pointer spre tipul comun nodurilor structurii, adică spre TNOD. Rezultă că funcția *incnod* are prototipul:

```
int incnod(TNOD *p);
```

0 altă funcție specifica aplicației concrete şi care se apelează la crearea listei este cea care eliberează zona de memorie rezervată nodului curent. Vom numi *elibnod* această funcție. Ea are prototipul:

```
void elibnod(TNGD *p);
```

Această funcție se apelează după un apel al funcției *incnod* în care aceasta nu a încărcat date în nodul curent (s-a revenit din ea cu valoarea 0 sau -1). În acest caz există zona de memorie rezervată în memoria heap pentru nodul curent, dar la apelul funcției *incnod* nu s-au încărcat date ori s-au încărcat date parțial. De aceea, în acest caz vom elibera zona de memorie rezervată apelînd *elibnod.*

Funcția pentru crearea listei simplu înlănțuite o numim *crelist.* Ea returnează una din valorile: 0 - la eroare; -1 - la crearea normală a listei. Ținînd seama de cele de mai sus, definim funcția *crelist* astfel:

```
int crelist ( )
```

```
{   /*- creaza o lista simplu inlantuita; - returneaza: 0 - la eroare; -1 - creare normala. */
extern TNOD *prim,*ultim;
int i,n;
TNOD *p;
n = sizeof(TNOD);
prim=uItim=0;          // initial lista este vida
    // se rezerva n octeti pentru nod in memoria heap
for(;;) {
    if ((p=(TNOD *)malloc(p))==0){
        printf("memorie insuficienta la crearea listei\n"); exit(1);
    }   // se incarca date in nod
    if((i=incnod(p))!=1){
        elibnod(p);    // se elibereaza zona rezervata pentru nod deoarece nu s-au incarcat date in nod
        return i;
    }                  // se inlanţsie nodul in lista
    if (ultim!=0)      // lista nu este vida
        ultim -> urm=p;
    else               // lista vida
        prim=p;
ultim=p;   ultim -> urm = 0;
}   // sfirsit for
}       // sfirsit crelist
```

Aşa cum s-a arătat înainte, o listă poate fi organizată păstrind elementele ei în memoria heap, iar adresele lor într-o tabelă de pointeri. Dăm mai jos o variantă a funcţiei *crelist* care să creeze o listă după acest principiu. În acest caz, tabloul de pointeri va fi global şi are un număr maxim de elemente. Numim *tpnod* acest tablou. De asemenea, vom nota cu *itpnod* variabila globală care are ca valoare indicele primului element al tabloului *tpnod* care este liber. Cu alte cuvinte, *tpnod[itpnod-1]* este pointerul spre ultimul nod adăugat la listă. Iniţial *itpnod* are valoarea zero. Tipul nodurilor nu mai este necesar să fie recursiv, deoarece în acest caz ordonarea se realizează prin indici: primul nod are indicele zero; al doilea nod are indicele 1; ...; ultimul nod are indicele *itpnod-1*. De aceea, în locul tipului TNOD vom folosi tipul:

```
typedef struct {
    declaratii
} TTNOD;
```

Tabloul *tpnod* se declară astfel:

```
TTNOD *tpnod[MAX];
```

unde: MAX - este o constantă simbolică în prealabil definită printr-o construcţie *#define* şi valoarea ei se va a1ege aşa încît să nu fie depăşită de numărul elementelor listei.

Variabila *itpnod* este de tip *int*:

```
int itpnod;
```

Indicăm mai jos funcţia pentru crearea listei cu utilizarea tabloului *tpnod*. Ea utilizează funcţiile *incnod* şi *elibnod* ca în cazul funcţiei *crelist*, fiind analogă cu aceasta.

```
int tpcrelist( ) // creaza o lista folosind un tablou de pointeri care contine adresele nodurilor listei;
{           // functia returneaza: 0 - la eroare; -1 - la creare normala.
extern TTNOD *tpnod[ ];
extern int itpnod; int i,n;
TTNOD *p;
n=sizeof(TTNOD);
for(itpnod=0; itpnod<MAX; itpnod++) {
    if((p=(TTNOD *)malloc(n))==0) {
        printf("memorie insuficienta\n");   exit(1);
}   // se incarca date in nod
if ((i=incnod(p))!=l) {
        elibnod(p);    return i;
}
    tpnod[ itpnod] =p;    // pastreaza adresa nodului in tabloul de pointeri
}   // sfirsit for
}
```

11.1.2. Accesul la un nod al unei liste simpiu înlănţuite

Putem avea acces la nodurile unei liste simplu înlănţuite începînd cu nodul spre care pointează variabila globala *prim* şi trecînd apoi pe rînd de la un nod la altul, folosind pointerul *urm*. Există cazuri în care dorim acces la un nod anumit al listei. În acest caz, este necesar să definim modul de indentificare al nodului la care dorim să avem acces. Un mod simplu este acela de a indica numărul de ordine al nodului la care se doreşte acces, de exemplu, se doreşte acces la al n-lea nod al listei. Cum lista este o mulţime dinamică, acest mod de a defini accesul la un nod al ei nu este totdeauna cel mai nimerit. O altă metodă este aceea, de a avea o dată componentă a nodurilor, care să aibă valori diferite, pentru noduri diferite. În acest caz se poate defini accesul la nodul din lista pentru care data respectivă are o valoare dată. O astfel de dată, care este componentă a nodurilor unei liste şi are valori distincte pentru noduri diferite ale unei liste se numeşte **cheie**. Cheia poate fi o dată de un tip oarecare.

În cazul de faţă vom considera cheia de tip *int*. Funcţia returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conţine un nod a cărui cheie sa aibă valoarea indicată de parametrul ei. Nodurile listei au tipul definit astfel:

```
typedef struct tnod {
        declaraii
    int cheie;
        declaratii
    struct tnod *urm;
}TNOD;
```

Putem acum defini funcția de căutare ca mai jos:

```
TNOD *cnci(int c )  // - cauta un nod al listei pentru care cheie=c; - returneaza pointerul spre nodul determinat in acest fel
    {                   // sau zero daca nu exista nic un nod asa incit cheia sa aiba valoarca c.
extern TNOD *prim;
    TNOD *p;
    p=prim;             // cautarea incepe cu nodul spre care pointeaza variabila prim
    while(p!=0) {
        if (p->cheie==c)  return p;   // s-a gasit un nod pentru care cheie=c
        p=p ->urm;    // se trece la nodul urmator din lista
    }
    // in acest punct se ajunge cind nu exista un nod in lista cu proprietatea ca cheie=c
    return 0;
}
```

O funcție similară se poate construi şi pentru cazul în care la implementarea listei se utilizează tabloul de pointeri *tpnod*. În acest caz tipul nodurilor listei se declară astfel:

```
typedef struct {
        declaratii
    int cheie;
        declaratii
} TTNOD;
```

Funcția caută nodul din listă parcurgînd-o cu ajutorul indicelui care variază de la zero pînă la *itpnod-1* inclusiv.

```
TTNOD *tpcnci (int c)  //- cauta un nod al listei pentru care cheie=c; - returneaza pointerul spre nodul respectiv
    {                       // sau zero daca nu exista in lista un astfel de nod
    extern TTNOD *tpnod[ ];
    extern int itpnod;
    int i;
    for(i=0; i<itpnod; i++)
        if(tpnod[i] -> cheie==c) return tpnod[i];
    return 0;
 }
```

Ambele funcții parcurg elementele listei, nod cu nod, fie pînă la întîlnirea nodului căutat, fie pînă la sfîrşitul listei dacă nu există în listă un nod a cărui cheie sa aibă valoarea cerută. Aceasta metodă de căutare se numeşte **căutare liniară**. Ea este o metodă foarte simplă de căutare dar nu este eficientă şi de aceea se aplică numai la liste cu un număr relativ mic de noduri.

0 metodă mult mai eficientă este metoda **căutării binare**. Această metodă poate fi aplicată simplu la listele implementate cu ajutorul tabloului de pointeri *tpnod*. În acest scop lista trebuie întîi sortată, de exemplu crescător, în raport cu valorile cheii. Pentru sortare se procedează astfel: cuvintele citite de la intrarea standard se pastrează în memoria heap şi în paralel cu aceasta se defineşte un tablou cu adresele la care sînt păstrate aceste cuvinte. Apoi se realizează sortarea lor folosind metodele clasice (metoda bulelor, sortare shell, sortare rapida, sortare cu ajutorul arborilor etc.).

11.1.3. Inserarea unui nod într-o listă simpiu înlănțuită

Într-o listă simplu înlănțuită se pot face inserări de noduri în diferite poziții: inserare înaintea primului nod; inserare înaintea unui nod precizat printr-o cheie; inserare după un nod precizat printr-o cheie; inserare după ultimul nod al listei (adăugarea unui nod nou la listă).

În cele ce urmeaza se definesc funcții atît pentru liste simplu înlănțuite, cît şi variantele lor cînd lista se implementează cu ajutorul tabloului de pointeri *tpnod*. Tipurile *TNOD* şi *TTNOD* se consideră declarate.

11.1.3.1. Inserarea unui nod într-o listă simplu înlănțuită înaintea primului ei nod

Primul nod al unei liste simplu înlănțuite este nodul spre care nu pointează pointerul *urm* al nici unui nod al listei, adică este nodul care nu este succesorul nici unui nod din listă. Amintim ca spre acest nod pointează variabila *prim* dacă lista nu este vidă.

În cazul utilizării tabloului *tpnod*, primul nod al listei este nodul spre care pointeaza elementul: *tpnod[0]*. Funcția de inserare returnează pointerul spre nodul inserat în listă sau zero în cazul în care nu se poate realiza inserarea. De altfel, toate funcțiile de inserare returnează aceste valori.

```
TNOD *iniprim( ) {  // insereaza nodul curent inaintea primului nod al listei
    extern TNOD *prim,*ultim;
    TNOD *p;
    int n;
    n=sizeof(TNOD);
        // rezerva memorie heap si incarca datele in zona respectiva
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        if (prim==0) { // lista vida
            // lista se compune numai din nodul curent
            prim=ultim=p;     p -> urm=0;   // nu exista succesor
    }
```

```
        else {
            p -> urm=prim;   // primul nod al listei devine succesorul nodului care se insereaza
            prim=p;          // nodul inserat nu este succesorul nici unui alt nod si de aceea prim pointeaza spre el
    }
        return p;
    }
    if(p==0) {
        printf("maeorie insuficienta\n");     exit(1);
}
        //s-a rezervat memorie pentru nod dar nu s-au incarcat date in nod
    elibnod(p); return 0;
}
```

Inserarea în cazul utilizării tabloului *tpnod* implică eliberarea elementului *tpnod[0]*. În acest scop se fac atribuirile:

tpnod[i] = tpnod[i-1] pentru i = itpnod, itpnod-1, ..., 2, 1

Se observă că funcția de inserare înaintea primului nod pentru liste, care foloseşte tabloul de pointeri *tpnod* nu mai este aşa de eficientă ca funcția *iniprim* definita mai sus.

```
TTNOD *tpiniprim( ) {       // insereaza un nod inaintea primului nod al listei
    extern TTNOD *tpnod[ ];
    extern int itpnod;
    int n, i;
    if(itpnod >= MAX) {
            printf("lista are prea multe elemente\n");        return 0;
    }
    n=sizeof(TTNOD);
    if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1)) {       // deplaseaza valorile elementelor tabloului tpnod
        for(i=itpnod++;i>0;i--)
            tpnod[i]=tpnod[i-1];
        tpnod[0]=p;    return p;
}
    if(p==0) {
        printf("memorie insuficienta\n");    exit(1);
    }
    elibnod(p);     return 0;
}
```

11.1.3.2. Inserarea unui nod într-o listă simplu înlănțuită înaintea unui nod precizat printr-o cheie

Vom presupune că cheia este de tip *int* ca şi în cazul funcției *cnci* definită în § 13.

```
TNOD *inici(int c)   // insereaza un nod inaintea unui nod precizat printr-o cheie numerica; - returneaza pointerul
                     // spre nodul inserat sau zero daca inserarea nu are loc.
{
    extern TNOD *prim;
    TNOD *q,*ql,*p;int n;
    n=sizeof(TNOD);
        // - cauta nodul de cheie=c; - la terminarea ciclului: q - pointeaza spre nodul respectiv;
        // q1 - pointeaza spre nodul precedent celui spre care pointeaza q.
    q1=0;  q=prim;
    while(q) {
    if (q->cheie==c)
        break; //s-a gasit nodul cautat
    q1 = q;      q = q -> urm;
    }
    if (q==0) {     //nu exista in lista un nod de cheie=c
    printf("nu exista in lista un nod de cheie= %d\n",c);        return 0;
    }
    // se rezerva zona pentru nod si se incarca datele nodului in zona respectiva
    if (((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
    if (q==prim) {     // cheie=c pentru primul nod si inserarea se face inaintea primului nod
        p->urm - prm;     prim = p;
    }
        else {       // nodul spre care pointeaza p se insereaza intre nodul spre care pointeaza q1 si nodul spre care pointeaza q
        q1->urm = p; // succesorul nodului spre care pointeaza q1 este nodul spre care pointeaza p
        p->urm = q;   // succesorul nodului spre care pointeaza p este nodul spre care pointeaza q
        }
        return p;
    }
    // nu s-a facut inserarea ceruta
    if (p==0) {
        printf("memorie insuficienta\n");    exit(1);
}
```

```
    elibnod(p);    return 0;
}
```
În continuare definim funcția care utilizează tabloul de pointeri *tpnod*.
```
TTNOD *tpinici(int c)        // - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;
{                            // - returneaza pointerul spre nodul inserat sau zero daca nu are loc inserarea.
    extern TTNOD *tpnod;
    extern int itpnod;
    int i,k;
    /* daca itpnod>=MAX nu exista loc liber in tabloul de pointeri */
    if(itpnod >= MAX) {
        printf("lista are prea multe elemente\n");        return 0;
    }
    /* cauta nodul de cheie=c */
    for(i=0;i<itpnod;i++)
        if(tpnod[i]->cheie==c)   break;      /* s-a gasit nodul cautat */
    if (i==itpnod) {        /* nu exista in lista un nod de cheie=c */
        printf("nu exista in lista un nod de cheie=%d\n",c);        return 0;
    }
    n=sizeof(TTNOD);
/* rezcrva zona pentru nod si se incarca datele nodului in zona respectiva */
    if(((p=(TTNOD *)malloc(n))!=0)&&(incnod(p)==1))    // deplaseaza valorile elementelor tabloului tpnod:
    {                                                  // tpnod[k] = tpnod[k-1], pentru k= itpnod,itpnod-l,...,i+1.
        for(k=itpnod++;k>i;k--)
            tpnod[k]=tpnod[k-1];    /* tpnod[i] devine egal cu adresa nodului care se insereaza */
            tpnod[i]=p;    return p;
    }
    /* nu s-a facut inserarea */
    if(p==0) {
        printf("memorie insuficienta\n");        exit(1)
}
    elibnod(p);    return 0;
}
```
11.1.3.3. Inserarea unui nod într-o listă simplu înlănțuită după un nod precizat printr-o cheie

Vom presupune că, cheia este de tip *int*.
```
TNOD *indci(int c)  // - insereaza un nod dupa un nod precizat psintr-o cheie numerica;
{                   // - returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc.
    extern TNOD *prim,*ultim;
    TNOD *p,*q;     int n;
        /* cauta nodul de cheie=c */
    for(q=prim;q;q=q->urm)
        if(q->cheie==c)   break;
    if(q==0) {
        printf("nu exista in lista un nod de cheie=%d\n",c);        return 0;
}
    /* se rczerva zona pentru nod si se incarca datele in nod */
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
    /* se insereaza nodul spre care pointeaza p dupa nodul spre care pointeaza q */
        p->urm=q->urm;      //nodul succesor nodului spre care pointeaza q devine succesor nodului spre care pointeaza p
        q->urm=p;           // succesorul nodului spre care pointeaza q devine nodul spre care pointeaza p
        if (ultim==q)       // - nodul spre care pointeaza q nu a avut succesor;
                ultim=p;    // - in acest moment nodul spre care pointeaza p nu are succesor.
        return p;
    }
    /* nu s-a reusit inserarea nodului */
    if(p==0) {
        printf("memorie insuficienta\n");        exit(1);
    }
    elibnod(p);         return 0;
}
```
Funcția de inserare pentru liste implementate cu ajutorul tabloului de pointeri *tpnod* care realizează inserarea după un nod de cheie dată este asemănătoare cu funcția *tpinici* definită în paragraful precedent. În acesi caz, se modifică instrucțiunea *for* pentru deplasarea elementelor tabloului *tpnod* şi instrucțiunea de atribuire următoare ei:
```
for(k=itpnod++;k>i+1;k--)
    tpnod[k]=tpnod[k-1];
tpnod[i+1]=p;
```

11.1.3.4. Adăugarea unui nod la o listă simplu înlănțuită

Adăugarea unui nod la o listă simplu înlanțuită înseamnă inserarea lui după nodul spre care pointează variabila *ultim*. Aceasta înseamnă că după inserarea nodului, variabila *ultim* pointează spre nodul respectiv, acesta devenind nodul din listă care nu are succesor.

```
TNOD *adauga( ) {  // adauga un nod la o lista simplu inlantuita;
                   // returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc.
extern TNOD *prim, *ultim;
    TNOD *p;     int n;
    n=sizeof(TNOD);
    if (((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        if (prim==0) /* lista este vida */
            prim=ultim p;
        else {
            ultim->urm=p;    // succesorul nodului spre care pointeaza u1tim devine nodul spre care pointeaza p
            ultim=p;         // acesta devine nodul spre care pointeaza ultim
        }
        p->urm = 0;          // nodul spre care pointeaza p nu are succesor
    return p;
}
    // nu s-a reusit inserarea nodului in lista
    if (p==0) {
        printf("memorie insuficienta\n");    exit(1);
}
    elibnod(p);     return 0;
}
```

Definim mai jos funcția de adăugare a unui nod la o listă implementată cu ajutorul tabloului *tpnod*.

```
TTNOD *tpadauga( )     // - adauga un nod la o lista;
    {                  // - returneaza pointerul la nodul inserat sau zero daca inserarea nu se realizeaza.
    extern TTNOD *tpnod[ ];
    extern int itpnod; int n;
    if (itpnod >= MAX) {
        printf("lista are prea multe elemente\n");        return 0;
    }
    n=sizeof(TTNOD);
    if(((p=TTNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        tpnod[itpnod++]=p;       return p;
    }
    if(p==0) {
    printf("memorie insuficienta\n");        exit(1);
    }
    elibnod (p);    return 0;
}
```

11.1.4. Ştergerea unui nod dintr-o listă simplu înlănțuită

Dintr-o listă simplu înlănțuită se pot şterge noduri. Aceasta se poate realiza în mai multe moduri. În cele ce urmează avem în vedere următoarele cazuri: a). ştergerea primului nod al listei simplu înlănțuite; b). ştergerea unui nod precizat printr-o cheie; c). ştergerea ultimului nod al listei simplu înlănțuite.

Funcțiile de ştergere utilizează funcția *elibnod*. Această funcție eliberează zona de memorie alocată nodului care se şterge, precum şi eventualele zone de memorie alocate suplimentar prin intermediul funcției *incnod* pentru a păstra diferite componente ale unui nod (de exemplu componeate de tip şir de caractere). Alături de funcțiile obişnuite de ştergere, se definesc astfel de funcții şi pentru listele implementate cu ajutorul tabloului *tpnod*.

11.1.4.1. Ştergerea prlmulul nod al unel liste simplu înlănțulte

```
void spn ( ) {       // sterge primul nod din lista
    extern TNOD *prim,*ultim;
    TNOD *p;
    if (prim==0) /* lista vida */
        return;
    p=prim;    prim=prim -> urm;    elibnod(p);
    if (prim==0) // lista a devenit vida
        ultim=0;
}
```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod*, se elimină nodul spre care pointează elementul *tpnod[0]*, apoi se deplasează elementele tabloului *tpnod* folosind atribuirile: tpnod[i] = tpnod[i+1], pentru i=0, 1, ..., itpnod-2.

```
void tpspn ( ) {     // sterge primul nod din lista
extern TTNOD *tpnod[ ];
    extern int itpnod; int i;
    elibnod(tpnod[8]);
    for(i=0;i<itpnod-1,i++)
```

```
        tpnod[i]=tpnod[i+1];
    itpnod--;
}
```

11.1.4.2. Şterge un nod precizat printr-o cheie, dintr-o listă simplu înlănțuită

Vom considera că, cheia este de tip *int*. În acest caz tipul *TNOD* se defineşte ca în § 13.

```
void snci(int c) {      // sterge nodul pentru care cheie=c
    extern TNOD *prim,*ultim;
    TNOD *q, *ql;
q1=0;       q=prim;
// - se cauta nodul de cheie=c;   - la terminarea ciclului q pointeaza spre nodul respectiv sau q=0 daca nu exista un astfel de nod;
// - q1 pointeaza spre nodul a carui succesor este nodul spre care pointeaza q sau q1=0 daca q=prim.
    whiIe(q) {
        if (q->cheie==c)  break;       // s-a gasit nodul cautat
        ql=q;   q=q->urm;
    }
    if (q=0) {       // nu exista in lista un nod pentru care cheie=c
        printf("lista nu contine un nod de cheie=%d\n",c);     return;
}
    // se sterge nodul spre care pointeaza q
    if (q==prim) {     // se sterge primul nod din lista
        prim=prim->urm;     elibnod(q);
        if(prim==0)   ultim=0;   // lista a devenit vida
}
    else {
    ql->urm=q->urm; //succesorul nodului spre care pointeaza q1 devine succesorul nodului spre care pointeaza q
        if (q==ultim)       // se sterge ultimul nod, deci nodul spre care pointeaza q1 devine ultimul nod al listei
            ultim=ql;
        elibnod(q);
    }
}
```

În cazul în care lista este implementată cu ajutorul tabloului *tpnod,* se caută nodul pentru care cheie=c. Fie acesta nodul a cărui adresă este dată de elementul tpnod[i].

După eliminarea nodului respectiv, se realizează atribuirile: tpnod[k]=tpnod[k+1], pentru k=i, i+1, ..., itpnod-2.

```
void tpsnci (int c) {   // sterge nodul de cheie=c
    extern TTNOD *tpnod[ ];
    extern int itpnod; int i;
    for(i=0;i<itpnod;i++)
        if(tpnod[i]->cheie==c)   break;
    if(i==itpnod) {
        printf("nu exista un nod de cheie=%d\n",c);    return;
    }
    elibnod(tpnod[i]);
    for( ;i<itpnod-1;i++)
        tpnod[i]=tpnod[i+1];
    itpnod--;
```

11.1.4.3. Ştergerea ultimului nod dintr-o listă simplu înlănțuită

```
void sun( ) {  // sterge ultimul nod din lista
    extern TNOD *prim,*ultim;
    TNOD *q, *ql;
    q1=0;       q=prim;
    if(q==0)   return; /* lista vida */
    while (q!=ultim) {     // se parcurge lista pina se ajunge la ultimul nod al ei
        ql=q;       q=q->urm;
    }
    if (q==prim)       // - lista contine un singur nod care se sterge; - lista devine vida.
        prim=u1tim=0;
    else {  // - nodul spre care pointeaza q1 are ca succesor nodul spre care pointeaza q si acesta este ultimul nod al listei;
            //- cum nodul spre care pointeaza q se sterge, nodul spre care pointeaza q1 devine ultimul, deci q1-> urm=0 si ultim=q1.
        ql->urm=0;    ultim=ql;
    }
    elibnod(q);
}
```

În cazul în care lista se implementează folosind tabloul *tpnod*, operația de ştergere a ultimului nod este mult mai eficientă, deoarece nu necesită parcurgerea nodurilor listei.

```
void tpsum( ) {    /* sterge uliimul nod al listei */
    extern TTNOD *tpnod[ ];
```

```
    extern int itpnod;
    if(itpnod==0)     return;     /* lista vida */
    elibnod(--itpnod);
```

Observație: Funcțiile definite în paragrafele precedente realizează operațiile cele mai frecvent utilizate asupra listelor. Unele dintre aceste funcții necesită parcurgerea nodurilor listei, parțial sau în totalitate. Alte funcții nu necesită astfel de parcurgeri. Funcțiile care parcurg nodurile unei liste au o eficiență scăzută în comparație cu cele care nu fac astfel de parcurgeri. De aceea, se recomandă pe cît posibil utilizarea funcțiilor care nu necesită parcurgerea nodurilor ei. Astfel de funcții sint:

iniprim - Inserează nodul curent înaintea primului nod al listei.
Adauga - Adaugă un nod la o listă.
Spn - Ştergerea primului nod al listei.

În cazul listelor implementate cu ajutorul tabloului *tpnod*, cele mai eficiente funcții sînt cele care nu necesită instrucțiuni ciclice care să se execute asupra elementelor tabloului *tpnod*. Astfel de funcții sînt:

tpadauga - Adaugă un nod la o listă.
tpsun - Şterge ultimul nod al listei.

11.1.5. Ştergerea unei liste simplu ÎnlĂnȚuite

Se utilizează funcția *elibnod* pentru fiecare nod al listei.

```
void sterglist ( )  {   /* sterge o lista sirnplu inlantuita */
    extern TNOD *prim,*ultim;
    TH0D *p;
    while(prim) {
        p=prim;   prim=prim->urm;     elibnod(p);
    }
    ultim=0;
}
```

Mai jos definim funcția de ştergere a listei implementate cu ajutorul tabloului *tpnod*.

```
void tpsterglist ( )     {   /* sterge lista implementata cu tablou de pointeri */
    extern TTNOD *tpnod[ ];
    extern int itpnod; int i;
    for(i=0;i<itpnod;i++)     elibnod(tpnod[i]);
    itpnod = 0;
}
```

11.2. Stive şi cozi

În general, o stivă se implementează printr-o listă simplu înlănțuită, cu toate că poate fi implementată şi printr-un tablou. O **stivă** este o listă simplu înlănțuită gestionată conform principiului LIFO (Last in First Out). Conform acestui principiu, ultimul nod pus în stivă este primul care este scos din stivă. Stiva, ca şi lista are 2 capete, **baza stivei** şi **vîrful stivei**.

Asupra unei stive se definesc cîteva operații, dintre care cele mai importante sînt: pune un element pe stivă (push); scoate un element din stivă (pop); şterge (videaza) stiva (clear). Primele 2 operații se realizează în vîrful stivei. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vîrful stivei şi în continuare, cel pus anterior lui pe stivă ajunge în vîrful stivei. Daca un element se pune pe stivă, atunci acesta se pune în vîrful stivei.

Pentru a implementa o stivă printr-o listă simplu înlănțuită va trebui să identificăm baza şi vîrful stivei cu capetele listei simplu înlănțuite. Există 2 posibilitați: a) nodul spre care pointează variabila *prim* este baza stivei, iar nodul spre care pointează variabila *ultim* este vîrful stivei; b) nodul spre care pointează variabila *prirn* este vîrful stivei, iar nodul spre care pointează variabila *ultim* este baza stivei.

În cazul a), funcțiile *push* şi *pop* se indentifică prin funcțiile *adauga* şi respectiv *sun*. Dacă funcția *adauga* este eficientă, în schimb funcția *sun* nu este eficientă.

În cazul b), funcțiile *push* şi *pop* se indentifică prin funcțiile *iniprim* şi respectiv *spn*. În acest caz ambele funcții sînt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlănțuită conform cazului b) indicat mai sus.

Vidarea stivei se realizează cu ajutorul funcției *sterglist*.

În concluzie, o stivă se poate implementa printr-o listă simplu înlănțuită, pentru care se pot utiliza funcțiile: *iniprim* - realizează operația *push*; *spn* - realizează operația *pop*; *sterglist* - realizează operația *clear*.

Stivele pot fi implementate folosind liste care la rîndul lor se implementează cu ajutorul tabloului de pointeri *tpnod*. În acest caz, la baza stivei se află nodul spre care pointează elementul *tpnod[0]*, iar în vîrful stivei se află nodul spre care pointează elementul *tpnod[itpnod-1]*. În acest caz se vor utiliza funcțiile: *tpadauga* - realizează operația *push*; *tpsun* - realizează operația *pop*; *tpsterglist* - realizează operația *clear*.

De obicei, în acest caz se mai definesc 2 operații asupra stivei: *isempty* şi *isfull*. Prima se defineşte printr-o funcție care returnează valoarea 1 dacă stiva este vidăa şi zero în caz contrar. Cea de a doua returnează valoarea 1 dacă stiva este plină şi zero în caz contrar. Păstrînd denumirile de mai sus, aceste funcții se definesc simplu astfel:

```
typedef enum (false,true) Boolean;
Boolean isempty ( )  {   /* returneaza true daca stiva este vida si false altfel */
    extern int itpnod;
    return itpnod==0;
}
 Boolean isfull( ) {        /* returneaza true daca stiva este plina si false altfel */
    extern int itpnod;
    return itpnod >= MAX;
}
```

Un alt principiu de gestiune a listelor simplu înlănțuite este principiul FIFO (First In-First Out). Conform acestui principiu, primul element introdus în listă este şi primul care este scos din listă. Despre o listă gestionată în acest fel se spune că formează o **coadă**. Cele 2 capete ale listei simplu înlănțuite care implementează o coadă sînt şi capetele cozii. Asupra cozilor se definesc 3 operații, ca şi asupra stivelor: pune un element în coadă; scoate un element din coadă; ştergerea (vidarea) unei cozi.

Pentru a respecta principiul FIFO, vom pune un element în coadă folosind funcția *adauga* şi vom scoate un element din coadă folosind funcția *spn*. Deci, la un capăt al cozii se pun elemente în coadă, iar din celălalt capăt se scot elementele din coadă. Ambele funcții, *adauga* şi *spn* sînt funcții eficiente.

Ştergerea unei liste se realizează cu ajutorul funcției *sterglist*.

În concluzie, coada este o listă simplu înlănțuită pentru care se pot utiliza funcțiile: *adauga* - realizează operația de adăugare a unui nod la coadă; spn - realizează operația de scoatere a unui nod din coadă; *tpsterglist* - realizează ştergerea nodurilor existente în coadă. Cozile definite ca mai sus corespund celor din viața de toate zilele şi din această cauză ele se folosesc frecvent în probleme de simulare a fenomenelor reale.

Stivele se utilizează la descrierea proceselor recursive, inversarea ordinii elementelor unei mulțimi ordonate etc.

11.3. Listă circulară simplu înlănțuită

Lista simplu înlănțuită s-a definit ca o mulțime ordonată de noduri, fiecare nod conținînd un pointer spre un alt nod al listei, numit următorul nodului respectiv, exceptînd un singur nod, care nu mai are următor. Acest nod, care nu mai are urmator onstituie un capăt al listei simplu înlănțuite. Spre acest nod pointează variabila *ultim*. De asemenea, lista simplu înlănțuită conține un nod care nu este următorul nici unui alt nod al ei. Acest nod constituie celălalt capăt al listei şi spre el pointează variabila *prim*. Pointerul prezent în fiecare nod al listei care defineşte ordinea nodurilor a fost numit *urm*. Conform celor spuse mai sus, ultim->urm=0. Dacă într-o listă simplu înlănțuită schimbăm valoarea expresiei ultim->urm făcînd: *ultim->urm=prim;* atunci lista simplu înlănțuită devine o **listă simplu înlănțuită circulară**. În continuare prin listă circulară vom înțelege o listă circulară simplu înlănțuită.

Într-o listă circulară toate nodurile sînt echivalente: fiecare nod are un următor şi fiecare nod este următorul unui nod. Într-o astfel de listă nu mai sînt capete şi de aceea nu mai sînt necesare variabilele *prim* şi *ultim*. Gestiunea nodurilor listei circulare se realizează folosind o variabilă globală care pointează spre un nod oarecare al listei. Numim *ptrnod* această variabilă,declarată astfel:

TNOD *ptrnod;

unde TNOD - este tipul comun nodurilor listei.

Asupra listelor circulare se definesc aceleaşi operații ca şi asupra listelor simplu înlănțuite. Listele circulare au o serie de aplicații dintre care amintim: operații cu numere întregi care au un număr mare de cifre; operații asupra polinoamelor de una sau mai multe variabile; alocarea dinamică a memoriei.

Menționăm că funcțiile *malloc*, *free*, precum şi celelalte utilizate la gestiunea dinamică a memoriei utilizează o zonă de memorie organizată ca o listă circulară. Memoria liberă este o listă circulară de noduri, fiecare cu o zonă de memorie liberă. Un nod conține dimensiunea lui, un pointer spre nodul următor din listă şi spațiul liber care se pune la dispoziția utilizatorului.

Alocarea se face astfel: la un apel a lui *malloc* se parcurg nodurile listei pîna cînd se găseşte o zonă de memorie de dimensiune cel puțin egală cu cea cerută la apel. Dacă zona este mai mare, atunci ea se divide şi partea neutilizată se înlănțuie cu celelalte blocuri ale listei. Cînd nu se găseşte o zonă de memorie corespunzătoare, atunci se încearcă obținerea ei printr-un apel la sistemul de operare.

Eliberarea unei zone de memorie prin intermediul funcției *free* va înlănțui zona respectivă la lista circulară. Dacă 2 noduri cu zone de memorie liberă ocupă zone contigue, atunci ele se concatenează formînd o singură zonă liberă de dimensiune egală cu suma dimensiunilor lor. De aceea, se recomandă utilizarea funcției *free* pentru a elibera zone de memorie de îndată ce nu mai este nevoie de ele. În felul acesta se poate preîntîmpina divizarea excesiva a zonei gestionată dinamic prin funcțiile de felul lui *malloc* şi *free*.

11.3.1. Crearea unei liste circulare

La crearea unei liste circulare, ca şi la crearea unei liste simplu înlănțuite, se utilizează funcțiile *incnod* şi *elibnod*. Prima se apelează pentru a încărca datele curente într-un nod al listei, iar cea de a doua pentru a elibera zonele de memorie alocate pentru un nod. Amintim ca funcția *incnod* returneaza: 0 - la eroare; 1 - la încărcarea normală; -1 - nu mai sînt date de încărcat în nod. Funcția de creare a listei circulare returnează: 0 - la eroare; -1 - la crearea fără erori a listei.

```
int ccrelist ( ) {        /* - creaza o lista circulara; - returneaza: 0 - la eroare; -1 - la creare normala. */
    extern TNOD *ptrnod;
    int i,n;      TNOD *p;
n=sizeof(TNOD);
    ptrnod=0;          /* lista este vida la inceput */
    while(((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==l))
    /* s-a rezervat zona pentru nod si s-au incarcat date in zona respectiva */
        if (ptrnod==0 ) {       /* lista vida */
            ptrnod=p;      ptrnod ->urm=p;
        }
        else {            /* nodul curent se insereaza dupa cel spre care pointeaza ptrnod */
            p->urm=ptrnod->urm;    ptrnod -> urm=p; ptrnod=p; /* ptrnod pointeaza spre ultimul nod inserat in lista */
        }        /* sfirsit else */
     / * sfirsit while: p=0 sau incnod nu a returnat valoarea 1 */
    if(p==0) {      /* nu s-a rezervat zona pentru nod */
    printf("memorie insuficienta\n");         exit(1);
    }
    // - s-a rezervat zona pentru nod dar incnod nu a reusit sa incarce date in zona respectiva returnind
    // o valoare diferita de unu, deci zero sau -1; - valoarea returnata de incnod va fi returnata si de functia ccrelist.
    elibnod(p);     /*elibereaza zona de memorie rezervata pentru nod si care n-a mai fost incarcata cu date */
    return i;         /* i are ca valoare valoarea returnata de incnod */
}
```

Observație: Funcția *ccrelist* a fost scrisă mai compact decît funcția *crelist* utilizînd expresia:
((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==1)
care se evaluează astfel: se apelează funcția *malloc* pentru a rezerva n O în memoria heap. În cazul în care se poate rezerva o zonă de n O, lui *p* i se atribuie o valoare diferita de zero şi deci primul operand al operatorului && are valoarea adevărat. În acest caz se evaluează operandul al doilea al operatorului &&, care apelează funcția *incnod* pentru a încărca datele curente în zona a cărei adresă de început este valoarea lui *p*. În cazul în care nu se pot rezerva n O în memoria heap, lui *p* i se atribuie valoarea zero şi deci primul operand al operatorului && este fals. În acest caz întreaga expresie este falsă şi deci nu se mai evaluează cel de al doilea operand al operatorului &&. De asemenea, expresia este falsă şi în cazul în care lui *p* i s-a atribuit o valoare diferită de zero, dar *incnod* a returnat o valoare diferită de unu.

11.3.2. Accesul la un nod al unei liste circulare

Ca în cazul listelor simplu înlănțuite şi în cazul listelor circulare putem căuta un nod după o cheie sau mai multe chei. În cazul listelor circulare, căutarea va începe cu nodul spre care pointează variabila globală *ptrnod*. Mai jos, se defineşte funcția *ccnci* care este analogă funcției *cnci* definită în cazul listelor simplu înlănțuite. Ea caută un nod după o cheie numerică şi returnează una din valorile: - pointerul spre nodul cautat; - 0 daca nu există un astfel de nod.

```
TNOD *ccnci(int c)      // - cauta nodul de cheie=c;
{                       // - returneaza pointerul spre nodul respectiv sau 0 daca nu exista un astfel de nod.
    extern TNOD *ptrnod;
    TNOD *p;
p=ptrnod;
    if (p==0)  return 0;   /* lista vida */
    do {
        if(p->cheie==c)   return p;
        p=p->urm;
    } while(p!=ptrnod);
    return 0;
}
```

11.3.3. Inserarea unui nod într-o listă circulară

Considerăm 2 operații de inserare a unui nod într-o listă circulară: 1) inserarea unui nod înaintea unuia precizat printr-o cheie numerica de tip *int*; 2) inserarea unui nod după unul precizat printr-o cheie numerică de tip *int*.
Pentru inserarea unul nod înaintea unuia precizat printr-o cheie de tip *int* se poate utiliza următoarea funcție:

```
TNOD *cinici(int c)     // - inserarea unui nod intr-o lista circulara inaintea unui nod precizat printr-o cheie de tip int;
{                       //'- returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc.
    extern TNOD *ptrnod;
    TNOD *p,*q,*q1;    int n;
    if(ptrnod==0)     return 0;   /* lista vida */
    q=ptrnod;
    do {
        ql=g;   q=q->urm;
        if(q->cheie==c)   break; /* s-a gasit nodul inaintea caruia se va face inserarea */
    } while(q!=ptrnod);
    if (q->cheie!=c) {
        printf("nu exista un nod de cheie=%d\n",c);    return 0;   /* nu s-a facut nici o inserare */
/* rezerva zona pentru nod si incarca datele in nodul respectiv */
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        ql->urm=p;    p->urm=q;     return p;
    }
    if(p==0) {
    printf("memorie insuficienta\n");         exit(1);
}
    elibnod(p);    return 0;
}
```

Pentru inserarea unui nod după un nod precizat printr-o cheie de tip int se poate utiliza următoarea funcție:

```
TNOD *cindci(int c) {   // insereaza un nod intr-o lista circulara dupa un nod precizat printr-o cheie de tip int
    extern TNOD *ptrnod;
    TNOD *p,*q;     int n;
    if(ptrnod==0) return 0;        /* lista vida */
    q=ptrnad;
    do {
        if (q->cheie==c)  break;
        q=q->urm;
    } while(qt=ptrnod);
    if(q->cheie!=c) {
        printf("nu exista un nod de cheie =%d\n",c);        return 0;
    }
    /* se face inserarea dupa nodul spre care pointeaza q */
```

```
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n)))=0)&&(incnod(p)==1)) {
        p->urm=q->urm;      q->urm=p;     return p;
    }
    if (p==0) {
        printf("memorie insuficienta\n");    exit(1);
    }
    elibnod(p);     return 0;
}
```

11.3.4. Ştergerea unui nod dintr-o listă circulară

Dăm o funcție care permite ştergerea dintr-o lista circulara a unui nod precizat printr-o cheie de tip *int*. În cazul în care variabila *ptrnod* pointează chiar spre nodul care se şterge, convenim ca *ptrnod* să pointeze spre nodul precedent celui şters, dacă lista n-a devenit vidă. În acest ultim caz, lui *ptrnod* i se atribuie valoarea zero.

```
void csnci (int c ) {        /* sterge nodul pentru care cheie=c */
    extern TNOD *ptrnod;
    TNOD *p,*pl;
    if(ptrnod==0)      return;      /* lista vida */
    p=ptrnod;
    do {
        pl=p;   p=p->urm;
        if(p->cheie==c)   break;
    } while(pl=ptrnod);
    if(p->cheie!=c) {
        printf("lista nu contine un nod de cheie=%d\n",c);          return;
}
    if(p==p->urm) {       // lista are un singur nod
        ptrnod=0;
}
    else {
        pl->urm=p ->urm;
        if (p==ptrnod) ptrnod=pl;          // se sterge nodul spre care pointeaza ptrnod
    }
    elibnod(p);
}
```

11.3.5. Ştergerea unei liste circulare

```
void csterglist( )  {    /* sterge o lista circulara */
    extern TNOD *ptrnod;
    TNOD *p,*pl;
    if ((p=ptrnod)==0)    return;      /* lista vida */
    do {
        pl=p;        p=p->urm;     elibnod(pl);
    } while(pt=ptrnod);
    ptrnod=0;
}
```

11.4. Listă dublu înlănțuită

Atît listele simplu înlănțuite, cît şi listele circulare, conduc adesea la parcurgeri neefective ale lor. De exemplu, ştergerea ultimului nod al unei liste simplu înlănțuite implică parcurgerea listei respective (vezi funcția *sun*). Astfel de parcurgeri pot fi uneori evitate folosind liste dublu înlănțuite. Într-o astfel de listă fiecare nod conține 2 pointeri: unul spre nodul următor, altul spre nodul precedent. În § de față, vom presupune că nodurile listelor dublu înlănțuite au tipul definit ca mai jos:

```
typedef struct tnod {
        declaragii
    struct tnod *prec; struct tnod *urm;
}TNOD;
```

Pentru a gestiona o listă dublu înlănțuită vom utiliza variabilele globale *prim* şi *ultim*, ca în cazul listelor simplu înlăanțuite. Variabila *prim* pointează spre nodul pentru care *prim->prec=0*. Pentru restul nodurilor, pointerul *prec* al unui nod pointează spre nodul precedent al listei. Variabila *ultim* pointează spre un nod pentru care *ultim->nrm=0*. Pentru restul nodurilor, pointerul *urm* al unui nod pointează spre nodul următor al listei. În concluzie, fiecare nod al listei are un nod precedent definit prin pointerul *prec* şi un următor definit prin pointerul *urm*. 0 excepție de la această regulă o constituie nodurile spre care pointează variabilele *prim* şi *ultim*. Nodul spre care pointează *prim* nu are precedent, iar nodul spre care pointează *ultim* nu are următor. Aceste noduri constituie capetele listei dublu înlănțuite.

În legătură cu listele dublu înlănțuite se pot defini aceleaşi operații ca şi în cazul listelor simplu înlănțuite. Operațiile: accesul la un element al unei liste dublu înlănțuite; inserarea unui nod într-o listă dublu înlănțuită; ştergerea unui nod dintr-o listă dublu înlănțuită; ştergerea unei liste dublu înlănțuite; - se realizează ca în cazul listelor simplu înlănțuite.

11.4.1. Crearea unei liste dublu înlănțuite

Definim mai jos funcția *dcrelist* utilizată pentru a crea o listă dublu înlănțuită. Ea este analogă cu funcția *crelist*. Funcțiile *incnod* și *elibnod* au aceeași semnificație și utilizare ca în cazul funcției *crelist*.

```
int dcrelist( ) {          // - creaza o lista dublu inlantuita; - returneaza: 0 - la eroare; -1 - creare normala.
    extern TNOD *prim,*ultim;
    TNOD *p;     int i,n;
    n=sizeof(TNOD);     prim=ultim=0;
    while(((p=(TNOD *)malloc(n))!=0)&&((i=incnod(p))==1))
        if(prim==0) {
            prim=ultim=p;     p->prec=p->urm=0;
} else {
            ultim->urm=p;     p->prec=ultim;     p->urm=0;     ultim=p;
        }
    if(p==O) {
        printf("memorie insuficienta\n");          exit(1);
    }
    elibnod(p);     return i;
}
```

11.4.2. Inserarea unui nod într-o listă dublu înlănțuită

Într-o listă dublu înlănțuită se pot face inserări în diferite poziții. Pentru inserarea unul nod într-o listă dublu înlănțuită înaintea primulul nod al ei se utilizează următoarea funcție:

```
TNOD *diniprim( ) {     // - insereaza nodul curent inaintea primului nod al listei; - returneaza pointerul spre nodul inserat.
    extern TNOD *prim,*ultim;
    TNOD *p;     int n;
    n=sizeof(THOB);
    if(((p=(TNOD *p)malloc(n))!=0)&&(incnod(p)==1)) {
        if(prim==0) {
            prim=ultim=p;     p->prec = p->urm = 0;
        } else {
            p->urm = prim;    p->prec=0;     prim->prec = p;    prim=p;
        }
        return p;
    }
    if(p==0) {      printf("memorie insuficienta\n");     exit(1);
            }    elibnod(p);     return 0;
}
```

Pentru inserarea unul nod într-o listă dublu înlănțuită înaintea unui nod precizat printr-o cheie de tip int se utilizează o funcție de inserare, care apelează funcția *dcnci* care caută într-o listă dublu înlănțuită un nod de cheie dată. Tipul nodurilor se declara astfel:

```
typedef struct tnod {
        declaratii
    int cheie;
        declaratii
    struct tnod *prec; struct tnod şurm;
} TNOD;
TNOD *dcnci(int c)       // - cauta un nod al listei pentru care cheie=c; - returneaza pointerul spre nodul determinat
{                        // in acest fel sau zero daca nu exista nici un nod pentru care cheie=c.
    extern TNOD *prim; TNOD *p;
    for(p=prim;p;p=p -> urm)
        if(p->cheie==c)   return p;
    return 0;
}
TNOD *dinici(int c)      // - insereaza un nod inaintea unui nod precizat printr-o cheie numerica;
{                        // - returneaza pointerul spre nodul inserat sau zero daca nu are loc inserarea.
    extern TNOD *prim;
    TNOD *p,*q;      int n;
    if((q=dcnci(c))==0) {     /* cauta nodul pentru care cheie=c */
        printf("nu exista in lista un nod de cheie=%d\n",c);    return 0;
    }
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        p->prec=q->prec;p->urm=q;
        if(q->prec!=0)     // precedentul lui q are ca urmator nodul inserat
            q->prec->urm=p;
        q->prec=p;
        if(prim==q)        // s-a inserat inaintea primului nod
            prim=p;
```

```
        return p;
    }
    if(p==0) {
        printf("memorie insuficienta\n);     exit(1);
    }
    elibnod(p);    return 0;
}
```

Pentru inserarea unul nod într-o listă dublu înlănțuită dupa unul după unul precizat printr-o cheie de tip int se utilizează o funcție de inserare, care apelează funcția *dcnci*. Tipul nodurilor se declara la fel ca şi la inserarea unul nod într-o listă dublu înlănțuită înaintea unui nod precizat printr-o cheie.

```
TNOD *dindci(int c)// - insereaza un nod dupa unul precizat printr-o cheie numerica;
{                           // - returneaza pointerul spre nodul inserat sau zero daca inserarea nu are loc.
    extern TNOD *ultim;
    TNOD *p,*q;     int n;
    if((q=dcnci(c))==0) {
        printf("nu exista nodul de cheie=%d\n",c);return 0;
}
    n=sizeof(TNOD);
    if(((p=(TNOD *p)malloc(n))!=0)&&(incnod(p)==1)) {
        p->prec=q;    p->urm=q->urm;
        if(q->urm!=0)/* urmatorul lui q are ca si precedent nodul inserat */
            q->urm->prec=p;
        q->urm=p;
        if(ultim==q)       /* s-a inserat dupa ultimul nod */
            ultim=p;
        return p;
    }
    if(p==0) {     printf("memorie insuficienta\n");    exit(1);
    }       elibnod(p);     return 0;
}
```

Pentru inserarea unul nod într-o listă dublu înlănțuită dupa unul după ultimul nod (adăugarea unui nod) tipul nodurilor se declara astfel:

```
typedef struct tnod {
        declaratii
    struct tnod *prec; struct tnod *urm;
} TNOD;
```

nefiind necesară prezența unei chei.

```
TNOD *dadauga( )       // - adauga un nod la o lista dublu inlantuita;
{                           // - returneaza pointerul spre nodul inserat sau zero daca nu se realizeaza inserarea.
    extern TNOD *prim,*ultim;
    TNOD *p;    int n;
    n=sizeof(TNOD);
    if(((p=(TNOD *)malloc(n))!=0)&&(incnod(p)==1)) {
        if(prim==0) {
            prim=ultim=p;    p->prec=p->urm=0;
    } else {
            ultim->urm=p;    p->prec = ultim;  p->urm=0;    ultim=p;
    }
        return p;
    }
    if(p==0) {     printf("memorie insuficienta\n");    exit(1);
}   elibnod(p);    return 0;
}
```

11.4.3. Ştergerea unui nod dintr-o listă dublu înlănțuită

Dintr-o listă dublu înlănțuită se pot şterge noduri. Pentru ştergerea primului nod al unei liste dublu înlănțuite :

```
void dspn ( ) {    /* sterge primul nod din lista */
extern TNOD *prim,*ultim;TNOD *p;
if(prim= 0)    return;
    p=prim;    prim=prim->urm;     elibnod(p);
    if(prim==0)   ultim=0;       // lista a devenit vida
    else              prirn->prec=0;
}
```

Pentru ştergerea unui nod dintr-o listă dublu înlănțuită precizat printr-o cheie de tip int.

```
void dsnci (int c) {    /* sterge nodul de cheie=c */
    extern TNOD *prim,*ultim;TNOD *p;
    if (prim==0)  return;      /* lista vida */
    if((p=dcnci(c))==0) {
```

```
        printf("lista nu contine nodul de cheie = %d\n",c);     return;
}
    if(prim==p)&&(ultim==p) {     /* lista are un singur nod; devine vida */
        prim=ultim=0;     elibnod(p);
    return;
    if (prim==p) {     /* se sterge primul nod din lista */
        prim=prim->urm;     prim->prec=0;     elibnod(p);     return;
    }
    if (ultim==p) {     /* se sterge ultimul nod */
        ultim=ultim->prec;    ultim->urm=0;     elibnod(p);     return;
}
// se sterge un nod diferit de capete. urmatorul nodului care se sterge are ca precedent, precedentul nodului care se sterge
    p->urm->prec=p->prec;
/* precedentul nodului care se sterge are ca urmator, urmatorul nodului care se sterge */
    p->prec->urm=p->urm;       elibnod(p);
}
```

Pentru ştergerea ultimului nod al unei liste dublu înlănțuite tipul TNOD nu necesită prezența unei chei.

```
void dsun ( ) {     /* sterge ultimul nod din lista */
    extern TNOD *prim,*ultim;
    TNOD *p;
    if(prim==0)    return;
    p=ultim;    ultim=ultim->prec;
    if(ultim==0)   prim=0;    /* lista devine vida */
    else     ultim->urm=0;
    elibnod(p);
}
```

Observații:

1. Funcția *dsun*, spre deosebire de funcția *sun* relativă la listele simplu înlănțuite, devine eficientă deoarece ea nu mai necesită parcurgerea nodurilor listei.

2. Funcțiile de inserare înainte şi după un nod precizat printr-o cheie numerică întreagă sînt mai simple în cazul listelor dublu înlănțuite în comparație cu analogele lor pentru liste simplu înlănțuite deoarece ele folosesc funcția *dcnci* pentru localizarea nodului în raport cu care se face inserarea.

3. Funcția de ştergere a unui nod precizat printr-o cheie numerică întreagă este mai simplă în cazul listelor dublu înlănțuite în comparație cu funcția corespunzătoare pentru liste simplu înlănțuite deoarece ea foloseşte funcția *dcnci* pentru localizarea nodului care se şterge.

4. O listă dublu înlănțuită devine o listă circulară dublu înlănțuită dacă se fac atribuirile: *ultim->urm=prim; prim->prec=ultim*. Pentru a gestiona o astfel de listă nu mai sînt necesare variabilele *prim* şi *ultim*, lista ne mai avînd capete. În locul lor se utilizează un pointer spre un nod arbitrar al listei, ca în cazul listelor circulare simplu înlănțuite.

În legătură cu listele circulare dublu înlănțuite se ridică aceleaşi probleme ca şi în cazul listelor circulare simplu înlănțuite: creare; acces la un nod de cheie dată; inserări de noduri; ştergeri de noduri; ştergerea listei.