LISTE

I. ASPECTE TEORETICE

1. Introducere

Lista este o mulțime dinamică, înțelegând prin aceasta faptul că ea are un număr variabil de elemente.

La început lista este o mulțime vidă. În procesul execuției programului se pot adăuga elemente noi listei şi totodată pot fi eliminate diferite elemente din listă.

* Elementele unei liste sunt date de acelaşi tip. Tipul comun elementelor unei liste este un tip utilizator. => **Elementele unei liste sunt structuri de acelaşi tip.**

* De multe ori listele sunt organizate sub formă de tablou. Acest lucru nu este eficient deoarece listele sunt dinamice, iar tablourile din limbajul C nu sunt dinamice.

* Există situații în care este dificil de a evalua numărul maxim al elementelor unei liste, precum si cazuri cand numarul lor difera de la executie la executie. => Apare problema de a organi-za o astfel de multime de tip lista.

Ordonarea elementelor unei liste se face cu ajutorul pointerilor care intra in compunerea elementelor listei. Datorita acestor pointeri, elementele listei devin structuri recursive. Listele organizate in acest fel se numesc liste inlantuite.

Prin definitie, o multime dinamica de structuri recursive de acelasi tip, pentru care sunt definite una sau mai multe relatii de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numeste lista inlantuita.

Elementele unei liste se numesc **noduri**.

Daca intre nodurile unei liste exista o singura relatie de ordine, atunci lista se numeste **simplu inlantuita**. In mod analog, lista este **dublu inlantuita** daca intre nodurile ei sunt definite doua relatii de ordine.

O lista este **n-inlantuita** daca intre nodurile ei sunt defi-nite n relatii de ordine.

Operatii ce tin de o lista inlantuita:
a) crearea listei inlantuite;
b) accesul la un nod oarecare al listei;
c) inserarea unui nod intr-o lista inlantuita;
d) stergerea unui nod dintr-o lista inlantuita;
e) stergerea unei liste inlantuite.

2. Lista simplu inlantuita

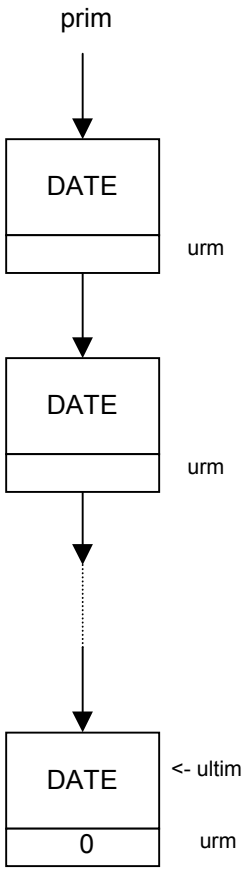
Intre nodurile listei simplu inlantuite avem o singura relatie de ordonare. Exista un singur nod care nu mai are succe-sor si un singur nod care nu mai are predecesor. Aceste noduri formeaza capetele listei.

Vom utiliza doi pointeri spre cele doua capete pe care ii notam cu:

prim - pointerul spre nodul care nu are predecesor

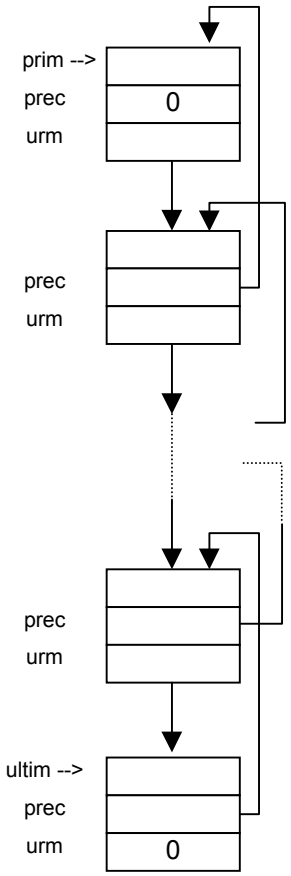
ultim - pointerul spre nodul care nu are succesor.

Pointerul **urm** defineste relatia de succe-sor pentru nodurile listei. Pentru fiecare nod, el are ca valoare adresa nodului urmator din lista. Exceptie face nodul spre care pointeaza varia-bila ultim (pentru care **urm** ia valoarea zero).



3. Lista dublu inlantuita

Intr-o astfel de lista fiecare nod contine doi pointeri: unul spre nodul urmator si unul spre nodul precedent. Astfel fiecare nod al listei are un nod precedent definit prin pointerul **prec** si un nod urmator definit prin pointerul **urm.**



4. Crearea unei liste simplu inlantuite

La crearea unei liste simplu inlantuite se realizeaza urma-toarele:

a) Se initializeaza pointerii prim si ultim cu valoarea zero, deoarece la inceput lista este vida.

b) Se rezerva zona de memorie in memoria heap pentru nodul cu-rent.

c) Se incarca nodul curent cu datele curente, daca exista si apoi se trece la pasul d). Altfel lista este creata si se revine din functie.

d) Se atribuie adresa din memoria heap a nodului curent pointeru-lui:
ultim -> urm, daca lista nu este vida;
prim -> urm, daca lista este vida.

e) Se atribuie lui **ultim** adresa nodului curent.

f) **ultim -> urm = 0**

g) Procesul se reia de la punctul b) de mai sus pentru a adauga un nod nou la lista.

5) Accesul la un nod intr-o lista simplu inlantuita

Putem avea acces la nodurile unei liste simplu inlantuite incepand cu nodul spre care pointeaza variabila globala **prim** si trecand apoi pe rand de la un nod la altul, folosind pointerul **urm**.

O alta metoda, mai buna, este aceea de a avea o data compo-nenta a nodurilor, care sa aiba valori diferite, pentru noduri diferite. In acest caz se poate defini accesul la nodul din lista pentru care data respectiva are o valoare data. O astfel de data, care este componenta a nodurilor unei liste si are valori dis-tincte pentru noduri diferite ale unei liste se numeste **cheie** si este o data de un tip oarecare (long, char, double, etc.). Functia returneaza pointerul spre nodul cautat sau zero in cazul in care lista nu contine un nod a carui cheie sa aiba valoarea indicata de parametrul ei.

6) Inserarea unui nod intr-o lista simplu inlantuita

Intr-o lista simplu inlantuita se pot face inserari de noduri in diferite pozitii:

a) inserare inaintea primului nod;

b) inserarea inaintea unui nod precizat printr-o cheie;

c) inserarea dupa un nod precizat printr-o cheie;

d) inserarea dupa ultimul nod al listei.

7) Stergerea unui nod dintr-o lista simplu inlantuita

Se au in vedere urmatoarele cazuri:

a) stergerea primului nod al listei simplu inlantuite;

b) stergerea unui nod precizat printr-o cheie;

c) stergerea ultimului nod al listei simplu inlantuite.

Functiile de stergere utilizeaza functia **elibnod**. Aceasta functie elibereaza zona de memorie alocata nodului care se sterge, precum si eventualele zone de memorie alocate suplimentar prin intermediul functiei **incnod** pentru a pastra diferite compo-nente ale unui nod (de exemplu componente de tip sir de carac-tere).

II. DESFASURAREA LUCRARII

NOTA: Functiile sau programele de la fiecare punct din desfasura-rea lucrarii se vor scrie in fisiere separate cu extensia **.cpp** .

1. Sa se scrie o functie care citeste un cuvant si-l pastreaza in memoria heap (prin cuvant intelegandu-se o succesiune de litere mici sau mari). Functia returneaza adresa de inceput a zonei din memoria heap in care se pastreaza cuvantul citit sau zero la intalnirea sfarsitului de fisier (EOF sau Ctrl Z).

O posibila solutie:

```
char *citcuv()
/* - citeste un cuvant si-l pastreaza in memoria heap;
   - returneaza pointerul spre cuvantul respectiv sau
   zero la sfarsit de fisier. */
{
 int c, i;
 char t[255];
 char *p;

 /*salt peste caracterele care nu sunt litere */
 while((c=getchar()) < 'A' || (c > 'Z' &&
              c < 'a') || c > 'z')
            if(c == EOF)
                     return 0; /* s-a tastat EOF */

 /* se citeste cuvantul si se pastreaza in t */
 i=0;
 do
   {
    t[i++] = c;
   } while(((c=getchar()) >= 'A' && c <= 'Z' ||
                 c >= 'a') && c <= 'z');
 if(c == EOF)
    return 0;
 t[i++] = '\0';

 /* se pastreaza cuvantul in memoria heap */
 if((p = (char *)malloc(i)) == 0)
             {
              printf("memorie insuficienta\n");
              exit(1);
             }
 strcpy(p,t);
 return p;
}
```

Observatie: Modul de lucru al functiei **strcpy** va fi inteles prin studierea help-ului din C (CTRL + F1).

2. Se considera tipul utilizator:

```
typedef struct tnod
                                        {char *cuvant;
                                         int frecventa;
                                         struct tnod *urm;
                                        }TNOD
```

care va fi utilizat in toate punctele urmatoare de la desfasura-rea lucrarii.

Se cere sa se scrie o functie **incnod** care incarca datele curente intr-un nod de tip TNOD. Functia de fata apeleaza functia **citcuv** si atribuie adresa returnata de ea pointerului cuvant din nodul curent. De asemenea, se atribuie valoarea 1 variabilei frecventa. Functia incnod returneaza valoarea -1 daca citcuv returneaza valoarea zero si 1 altfel.

O posibila solutie:

```
int incnod(TNOD *p)
/* incarca datele curente in nodul spre care pointeaza p */
{
 if((p -> cuvant = citcuv()) == 0)  return -1;
 p -> frecventa = 1;
 return 1;
```

```
}
```

3) Sa se scrie o functie **elibnod** care elibereaza zonele din memoria heap ocupate de nodul de tip TNOD definit in exercitiul precedent.

O posibila solutie:

```
void elibnod(TNOD *p)
/* elibereaza zonele din memoria heap ocupate de nodul
  spre care pointeaza p */
{
 free(p -> cuvant);
 free(p);
}
```

4) Sa se scrie o functie **adauga**, care permite adaugarea unui nod de tip TNOD la o lista simplu inlantuita, dupa ultimul nod al listei (spre care pointeaza variabila ultim). Aceasta inseamna ca dupa inserarea nodului, variabila **ultim** pointeaza spre nodul respectiv, acesta devenind nodul din lista care nu are succesor.

Functia de fata apeleaza functiile incnod si elibnod defi-nite in exercitiile precedente.

O posibila solutie:

```
TNOD *adauga()
/* - adauga un nod la o lista simplu inlantuita;
   - returneaza pointerul spre nodul adaugat sau
   zero daca nu s-a realizat adaugarea. */
{
 extern TNOD *prim, *ultim;
 TNOD *p;
 int n;

/* se rezerva zona de memorie pentru un nod si se incarca datele din zona respectiva */
 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 ultim devine nodul
                                                     spre care pointeaza p */

              ultim = p; /* acesta devine nodul spre
                                                   care pointeaza p */
             }
     p -> urm=0;
     return p;
    }
 if(p == 0)  /* nu s-a reusit inserarea nodului in lista */
    {
     printf("memorie insuficienta\n");
     exit(1);
    }
 elibnod(p);
 return 0;
}
```

5) Considerand o lista simplu inlantuita ale carei noduri au tipul TNOD (definit la punctul 2), sa se scrie o functie care cauta, in lista respectiva, nodul pentru care pointerul **cuvant** are ca valoare adresa unui cuvant dat.

Cu alte cuvinte, pointerul **cuvant** joaca rol de cheie si se cere sa se gaseasca nodul a carui cheie pointeaza spre un cuvant dat.

O posibila solutie:

```
TNOD *cncs(char *c)
/* - cauta un nod al listei pentru care cuvantul spre care
   pointeaza cuvant este identic cu cel spre care pointeaza c;
   - returneaza pointerul spre nodul determinat sau zero
```

```
   daca nu exista un astfel de nod. */
{
 extern TNOD *prim;
 TNOD *p;

 for(p = prim; p; p = p->urm)  /* cautarea incepe cu nodul spre
                                                        care pointeaza variabila prim
                                                        si se baleiaza toata lista */
    if(strcmp(p->cuvant,c) == 0)
            return p; /* s-a gasit un nod cu cheia c */

 return 0; /* nu exista nici un nod in lista cu cheia c */
}
```

Observatie: Modul de lucru al functiei **strcmp** va fi inteles prin studierea help-ului din C (CTRL + F1).

6) Sa se scrie o functie care sterge ultimul nod al unei liste simplu inlantuite ale carui noduri au tipul TNOD (definit la punctul 2). Aceasta functie este dependenta numai de pointerul **urm** din tipul nodurilor. Ea va apela functia elibnod.

O posibila solutie:

```
void sun() /* sterge ultimul nod din lista */
{
 extern TNOD *prim, *ultim;
 TNOD *q, *q1;
 q1 = 0;
 q = prim;
 if(q == 0)
   return; /* lista vida */
 while(q != ultim) /* se parcurge lista pana
                           se ajunge la ultimul nod al ei */
   {
    q1 = q;
    q = q->urm;
   }
 if(q == prim) /* - lista contine un singur nod care se sterge
                        - lista devine vida */
    prim = ultim = 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
            ultim = q1 */
    {
     q1 -> urm=0;
     ultim = q1;
    }
 elibnod(q);
}
```

7) Sa se scrie un program care citeste cuvintele dintr-un text si afiseaza numarul de aparitii al fiecarui cuvant din textul respectiv. Cuvantul se defineste ca o succesiune de litere mici si/sau mari. Textul se termina prin sfarsitul de fisier (CTRL+Z).

Programul se realizeaza utilizand o lista simplu inlantuita ale carei noduri au tipul TNOD deja definit si se executa astfel:

a) La intalnirea unui cuvant se construieste un nod pentru cuvan-tul respectiv. Acesta contine pointerul spre cuvant, care este pastrat in memoria heap. Frecventa de aparitie a cuvantului se face egala cu 1.

b) Nodul alcatuit la punctul 1 se adauga la lista simplu inlantuita care se construieste.

c) Se cauta in lista un nod care sa corespunda cuvantului curent. Daca exista un astfel de nod si acesta nu este ultimul nod al listei, atunci se mareste frecventa din nodul respectiv si apoi se sterge ultimul nod al listei (cel adaugat la punctul b) deoarece cuvantul exista deja in lista.

Dupa aceea se revine la pasul a) si ciclul continua pana cand nu mai sunt cuvinte de citit (s-a ajuns la sfarsitul de fisier). In acest moment se listeaza cuvintele si frecventa lor de aparitie, corespunzatoare nodurilor listei.

Programul va include toate functiile definite la punctele anterioare.