Curs SDA (PC2)
Curs 5
Liste (continuare)

Iulian Năstac

Recapitulare

Prin definiţie, o mulţime dinamică de structuri recursive de acelaşi tip, pentru care sunt definite una sau mai multe relaţii de ordine cu ajutorul unor pointeri din compunerea structurilor respective, se numeşte **listă înlănţuită.**

Recapitulare

- Elementele unei liste se numesc **noduri**.

- Dacă între nodurile unei liste există o singură relaţie de ordine, atunci lista se numeşte **simplu înlănţuită**. În mod analog, lista este **dublu înlănţuită** daca între nodurile ei sunt definite două relaţii de ordine.

- O listă este **n-înlănţuită** dacă între nodurile ei sunt definite n relaţii de ordine.

Recapitulare

Operaţii ce ţin de o listă înlănţuită:

a) crearea listei înlănţuite;

b) accesul la un nod oarecare al listei;

c) inserarea unui nod într-o listă înlănţuită;

d) ştergerea unui nod dintr-o listă înlănţuită;

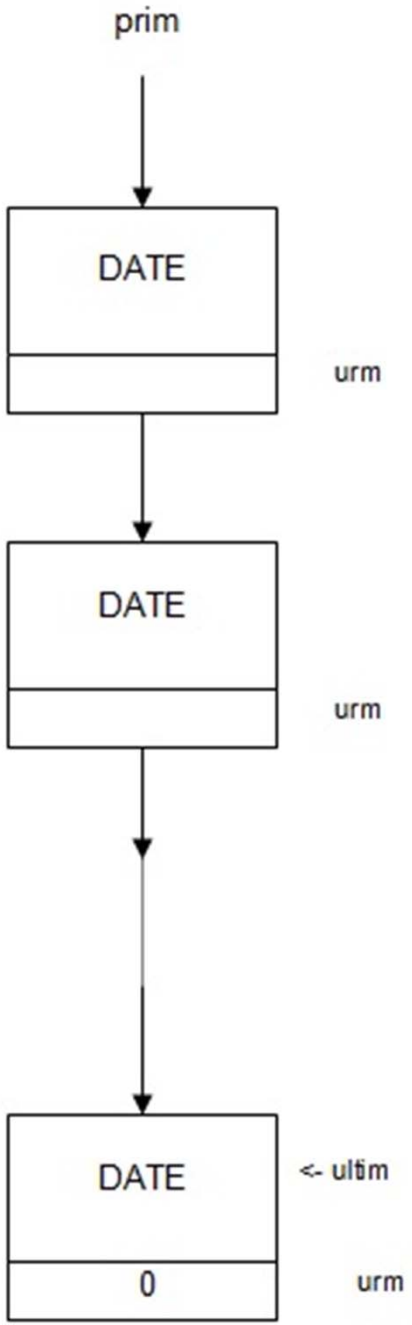
e) ştergerea unei liste înlănţuite.

2. Lista simplu înlănţuită
(Recapitulare)

Între nodurile listei simplu înlănţuite avem o singură relaţie de ordonare. Există un singur nod care nu mai are succesor şi un singur nod care nu mai are predecesor. Aceste noduri formează capetele listei.

Vom utiliza doi pointeri spre cele două capete pe care îi notăm cu:

- **prim** - pointerul spre nodul care nu are predecesor
- **ultim** - pointerul spre nodul care nu are succesor.



Pointerul **urm** defineşte relaţia de succesor pentru nodurile listei. Pentru fiecare nod, el are ca valoare adresa nodului următor din listă. Excepţie face nodul spre care pointează variabila **ultim** (pentru care **urm** ia valoarea zero).

2.1. Crearea unei liste simplu înlănţuite

(Recapitulare)

La crearea unei liste simplu înlănţuite se realizează următoarele operaţii:

1) Se iniţializează pointerii **prim** şi **ultim** cu valoarea zero, deoarece la început lista este vidă.

2) Se rezervă zona de memorie în memoria heap pentru nodul curent.

3) Se încarcă nodul curent **p** cu datele curente, dacă există şi apoi se trece la pasul ***4)***. Altfel, dacă nu există date, se șterge nodul curent **p**. Lista se consideră creată din noduri anterior introduse (daca există) şi se revine din funcţie.

4) Se atribuie adresa din memoria heap a nodului curent pointerului:

ultim -> urm = p, dacă lista nu este vidă;

prim = ultim = p, dacă lista este vidă.

5) Se atribuie lui **ultim** adresa nodului curent (**ultim=p**).

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

7) Procesul se reia de la punctul ***2)*** de mai sus pentru a adăuga un nod nou la listă.

Observaţii:

- Se consideră tipul utilizator:

```
typedef struct nod
        { declarații;
          struct nod *urm;
        } NOD;
```

- La punctul **3)** se încarcă datele printr-o funcție pe care o vom denumi **incnod** care încarcă datele curente într-un nod de tip NOD.

- Funcţia incnod returnează:
  - 1 la încărcarea corectă a datelor în nodul curent
  - -1 când nu mai sunt de încărcat date
  - 0 la apariția unei erori (ex: memorie insuficientă)

- Funcţia incnod are prototipul:

  int incnod(NOD *p)

- O altă funcție necesară este cea care eliberează zona de memorie rezervată unui nod:

  void elibnod(NOD *p)

2.2. Accesul la un nod într-o listă simplu înlănţuită

- Putem avea acces la nodurile unei liste simplu înlănţuite începând cu nodul spre care pointează variabila globală **prim** şi trecând apoi pe rând de la un nod la altul, folosind pointerul **urm**.
- O altă metodă, mai bună, 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 listă pentru care data respectivă are o valoare fixată. O astfel de dată se numeşte **cheie** şi este de un tip oarecare (uzual se foloseşte char şi int). Funcţia returnează pointerul spre nodul căutat sau zero în cazul în care lista nu conţine un nod a cărui cheie să aibă valoarea indicată de parametrul funcției.

- O cheie se inserează în tipul utilizator:

```
typedef struct nod
        { declarații;
          tip cheie;
          struct  nod *urm;
        } NOD;
```

- Cheia poate fi int, long, double, char, etc.

Exemplu:

- Se consideră tipul utilizator:

```
typedef struct nod
        { char *cuvant;  /*aceasta este cheia */
          int frecventa;
          struct nod *urm;
        } NOD;
```

- Considerând o listă simplu înlănţuită ale cărei noduri au tipul **NOD** se scrie o funcţie care caută, în lista respectivă, nodul pentru care pointerul **cuvant** are ca valoare adresa unui cuvânt dat.

- Pointerul **cuvant** joacă rol de cheie şi se cere să se găsească nodul a cărui cheie pointează spre un cuvânt dat.

```
NOD *cncs(char *c)      /*caută nod comun în șir*/
/* - 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 NOD *prim;
 NOD *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 */
}
```

2.3. Inserarea unui nod într-o listă simplu înlănţuită

Într-o listă simplu înlănţuită se pot face inserări de noduri în diferite poziţii:

a) inserare înaintea primului nod;

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

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

d) inserarea după ultimul nod al listei.

Exemplu:

- inserarea după ultimul nod al listei

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

/* se rezerva zona de memorie pentru un nod si se incarca
datele din zona respectiva */
 n = sizeof(NOD);
…
```

```
…
if(((p = (NOD *)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 alocarea de memorie */
    {
     printf("memorie insuficienta\n");
     exit(1);
    }
 elibnod(p);
 return 0;
}
```

2.4. Ştergerea unui nod dintr-o listă simplu înlănţuită

Sunt avute în vedere următoarele cazuri:

a) ştergerea primului nod al listei simplu înlănţuite;

b) ştergerea unui nod precizat printr-o cheie;

c) ştergerea ultimului nod al listei simplu înlănţuite.

Observaţii:

- Funcţiile de ştergere utilizează funcţia **elibnod**.
- Această funcţie eliberează zona de memorie alocată nodului care se şterge, precum şi eventualele zone de memorie alocate suplimentar prin intermediul funcţiei **incnod** pentru a păstra diferite componente ale unui nod (de exemplu componente de tip şir de caractere).
- Codurile funcţiilor **incnod** şi **elibnod** depind de aplicaţia concretă în care sunt utilizate.

Ștergerea primului nod

```
void spn()
{
extern NOD *prim, *ultim;
NOD *p;
if (prim == 0) return;
p = prim;
prim = prim -> urm;
elibnod(p);
if (prim == 0)  /* lista este vida */
        ultim = 0;
}
```

Observație:

- Există o mare diferență, în complexitate, dacă comparăm între ele o funcție care șterge primul nod cu o funcție care șterge ultimul nod al listei.

2.5. Ștergerea unei liste simplu înlănțuite

```
void sterge_l()
{
        extern NOD *prim, *ultim;
        NOD *p;
        while(prim)
                {
                        p = prim;
                        prim = prim -> urm;
                        elibnod (p);
                }
        ultim = 0;
}
```

Stiva

- **O *stivă* este o listă simplu înlănţuită gestionată conform principiului LIFO (Last In First Out).**

- Conform acestui principiu, ultimul nod pus în **stivă** este primul nod care este scos din stivă. **Stiva**, ca şi lista obişnuită, are două capete:
  - **baza stivei**
  - **vârful stivei**

Asupra unei stive se definesc câteva operaţii, dintre care cele mai importante sunt:

1. pune un element pe stivă (**PUSH**);

2. scoate un element din stivă (**POP**);

3. şterge (videază) stiva (**CLEAR**).

- Primele două operaţii se realizează în ***vârful stivei***. Astfel, dacă se scoate un element din stivă, atunci acesta este cel din vârful stivei şi în continuare, cel pus anterior lui pe stivă ajunge în vârful stivei.

- Dacă un element intră pe stivă, atunci acesta se pune în vârful stivei şi în continuare el desemnează vârful stivei.

Pentru a implementa o stivă printr-o listă simplu înlănţuită va trebui să identificăm baza şi vârful stivei cu capetele listei simplu înlănţuite. Există două posibilităţi:

a. **nodul spre care pointează variabila prim este baza stivei, iar nodul spre care pointează variabila ultim este vârful stivei;**

b. **nodul spre care pointează variabila prim este vârful stivei, iar nodul spre care pointează variabila ultim este baza stivei.**

Observaţii:

- În cazul **a**, funcţiile PUSH şi POP se identifică prin funcţiile **adauga** şi respectiv **sun**, definite în şedinţa anterioară de laborator. Dacă funcţia **adauga** este eficientă, în schimb funcţia **sun** nu este eficientă.

- În cazul **b**, funcţiile PUSH şi POP se identifică prin funcţiile **iniprim** şi respectiv **spn**, ce vor fi definite în această lucrare de laborator. În acest caz, ambele funcţii sunt eficiente. De aceea, se recomandă implementarea stivei printr-o listă simplu înlănţuită conform cazului **b** indicat mai sus.