Liste

12 aprilie 2005

Ce e o listă ? Cum o implementăm în C ?

Lista = o înșiruire de elemente
– pe care o putem parcurge secvențial (de la început la sfârșit)
– putem introduce și șterge elemente în/dintr-o anumită poziție

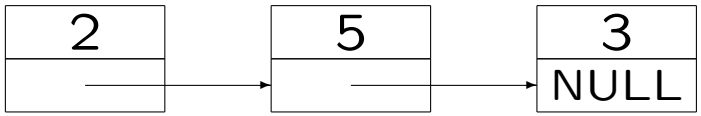
Fie lista de întregi `5, 2, 3, 6`.
O putem implementa cu un tablou: `int a[4] = { 5, 2, 3, 6};` Dar:
– are 4 elemente, fără loc pentru altele (ex. să inserăm pe 7 după 2)
(putem declara tabloul mai mare: `int a[10];` dar tot se va umple)
– pentru a șterge `a[1]=2` din listă trebuie mutate elementele de după

⇒ o implementare simplă cu tablou nu e eficientă și flexibilă

⇒ trebuie să reprezentăm *înlănțuirea*, care e la baza noțiunii de listă
(cum ajungem de la un element la următorul ? în tablou: indice++)
– folosim *pointeri* și păstrăm în fiecare element adresa următorului

Implementarea cu pointeri a listelor

```
typedef int elem_t;      /* tipul elementelor din listă */
typedef struct n {
  elem_t e;              /* câmp cu informatia utilă -- întregul */
  struct n *next;        /* câmp cu adresa elementului următor */
} node_t;                /* node_t e nume echivalent cu struct n */
typedef node_t *list_t; /* lista e adresa unui element (primului) */
```



Lista vidă e reprezentată ca pointer `NULL` $\Rightarrow$ inițializăm lista:

```
list_t l = NULL;
```

Inserarea după un element în listă

```
node_t *insertafter(node_t *n, elem_t e) {
  // presupune că n e un nod valid, diferit de NULL
  node_t *p; // adresa pentru noul nod, trebuie alocat!
  if (!(p = malloc(sizeof(node_t)))) return NULL; // eroare
  p->e = e; // completăm elementul în noul nod
  p->next = n->next; // legăm noul nod la succesorul celui vechi
  n->next = p; // legăm vechiul nod la noul nod creat
  return p; // returnăm noul nod creat
} // noul nod p a fost inserat după vechiul nod n
```

Inserarea ca prim element în listă

```
node_t *insertfirst(node_t *n, elem_t e) {
  node_t *p; /* adresa pentru noul nod, trebuie alocat! */
  if (!(p = malloc(sizeof(node_t)))) return 0; /* eroare */
  p->e = e; /* completăm elementul în noul nod */
  p->next = n; /* legăm noul nod la succesorul celui vechi */
  return p; /* noul nod, devenit capul listei */
}
```

Obs: nu putem insera la fel înaintea unui element *arbitrar* `n` din listă:
nu știm care e elementul anterior care are legătură spre `n`
(legătură care trebuie modificată să arate spre noul nod `p`)
– e nevoie să știm elementul *dinaintea* lui `n`
– la fel și pentru ștergere

Ştergerea unui element din listă

```
void deleteafter(node_t *n)  // sterge nodul de după n
{
  // presupune că n e nod valid, diferit de NULL
  node_t *p = n->next; // nodul care trebuie şters
  if (p == NULL) return; // nu e nimic de şters
  n->next = p->next; // scoate pe p din listă (inainte n->next==p)
  // si leagă n la succesorul nodului p de sters
  free(p); // eliberează memoria pentru nodul şters
}
```

Căutarea unui element într-o listă

```
node_t *lookup(list_t l, elem_t e)
{
  for ( ; l != NULL; l = l->next)      // caută până la sfârşit
    if (e == l->e) return l;   // găsit: returnează nodul (poziţia)
  return NULL;                 // nu s-a găsit: returnează NULL
}
```

Putem implementa ușor căutarea, privind lista ca obiect recursiv:
Def.: o listă este: fie lista vidă, fie un element urmat de o listă

```
node_t *lookup(list_t l, elem_t e)
{
  if (l == NULL) return NULL; // negăsit: lista vidă
  else if (e == l->e) return l; // găsit la pozitia curenta
  else return lookup(l->next, e); // caută mai departe
}
```

O bibliotecă pentru lucru cu liste

Cu funcţiile de mai sus putem să scriem programe cu liste.
Dar e util să nu trebuiască să rescriem funcţiile de fiecare dată
⇒ putem crea o *bibliotecă* de lucru cu liste.
Declaraţiile de tipuri şi funcţii necesare le punem într-un fişier `lista.h`:

```
typedef int elem_t;       // tipul elementelor din listă
typedef struct n node_t; // declaraţie incompletă
// ne spune că node_t e un tip structură, fără a preciza conţinutul
typedef node_t *list_t; // tipul lista e adresa unui nod
```

Declaraţia completă a tipului structură şi *definiţiile* le punem într-un fişier `lista.c` care poate fi compilat separat, şi linkeditat apoi cu programul care utilizează funcţiile.

Inversarea unei liste

```
list_t reverselist(list_t head) {          /* varianta iterativă */
  node_t *nxt, *rev = NULL; /* nxt=urm. nod, rev=lista inversată */
  while (head) {                    /* leagă următorul elem. la rev */
    nxt = head->next; head->next = rev;
    rev = head; head = nxt;  /* avansează nxt în listă */
  }
  return rev;
}
list_t reverselist(list_t rev, list_t rest)
{
  if (!rest) return rev;
  else {
    node_t *nxt = rest->next;
    rest->next = rev;
    return reverselist(rest, nxt);
  }
} /* la început apelăm reverselist(NULL, head); */
```