Tipuri de date abstracte. Liste

23 martie 2004

Recapitulare: stive și cozi

Stiva: o secvență cu inserare și extragere la un singur capăt
⇒ elementul extras (pentru prelucrare) e întotdeauna ultimul introdus
– principiul stivei: folosit la implementarea apelului de funcții
(pentru salvarea adresei de revenire în program și a variabilelor locale)
⇒ poate fi folosită pentru simularea recursivității
– Exemplu: calculator de buzunar pentru expresii *postfix*
(operatorul urmează după operanzi; nu sunt necesare paranteze)

Coada: o secvență cu inserare la un capăt și extragere la altul
⇒ elementul extras e întotdeauna cel mai vechi introdus
⇒ folosită pentru prelucrarea secvențială (în care pasul de prelucrare poate produce la rândul lui noi elemente de prelucrat)
– Exemplu: regiunea accesibilă dintr-un punct într-un plan cu obstacole

Tipul de date abstract *listă*

Lista = o înșiruire de elemente care se poate parcurge secvențial,
și în care se pot insera elemente în poziția dorită.
Def. recursivă: o listă este fie vidă, fie un element urmat de o listă

Operații pentru tipul de date abstract listă

element: tipul de date stocat în listă (informația utilă)
poziție: tip care identifică locația unui element în listă (posibil: pointer)

- init(lista) /* crează listă vidă */
- empty(lista): boolean /* lista este vidă ? */
- first(lista) : pozitie /* returnează prima poziție din listă */
- next(pozitie) : pozitie /* următoarea poziție; lista e implicită */
- lookup(lista, element) : pozitie /* caută elementul în lista */
- insertfirst(lista, element) /* inserează la început */
- insertafter(pozitie, element) /* inserează după poziție */
- delete(lista, pozitie) /* șterge poziția din listă */

Tablouri, liste, mulțimi: tipul și implementarea potrivită

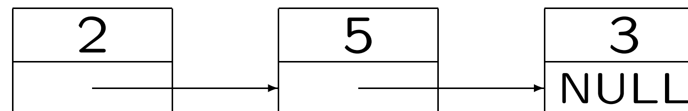
Tabloul: o structură de date pentru un șir de elemente identice
– oferă nu numai secvențiere, dar și *acces direct*
– implementează o *funcție* de la mulțimea indicilor la cea de elemente
⇒ fiecare element poate fi identificat printr-un întreg (poziția în tablou)

O listă s-ar putea implementa cu un tablou. Dar apar și dezavantaje:
– dacă lista crește, tabloul poate fi prea mic (⇒ realocat dinamic)
– ștergerea unui element din tablou implică
   – fie mutarea celorlalte elemente (schimbă corespondența între indici și element; e costisitoare dacă trebuie păstrată ordinea)
   – fie marcarea elementelor șterse (cu un câmp fanion suplimentar) (parcurgerea devine ineficientă dacă sunt multe elemente șterse)

Mulțimea: tip similar cu lista, dar fără ordonare și fără duplicate
⇒ implementabilă ca listă cu `insertfirst`, `delete` și test de membru

Implementarea cu pointeri a listelor

```
typedef int elem_t;      /* sau alt tip */
typedef struct n {
  elem_t e;              /* informatia utilă */
  struct n *next;        /* pointer la elementul următor */
} node_t;                /* nume echivalent cu struct n */
typedef node_t *list_t; /* tipurile listă şi poziţie sunt acelaşi,
  adică un pointer la node_t */
```



Implementarea cu pointeri a listelor

```
void init(list_t *pl) { *pl = NULL; } /* modifică valoarea listei,
  deci trebuie pointer la listă ca parametru */
int empty(list_t l) { return l != NULL; }
node_t *first(list_t l) { return l; }
node_t *next(node_t *n) { return n->next; }

int insertfirst(list_t *pl, elem_t e) {
  node_t *p;
  if (!(p = malloc(sizeof(struct n)))) return 0; /* eroare */
  p->e = e; p->next = *pl; *pl = p; return 1; /* succes */
} /* modifică capul listei, deci are parametru pointer la listă */

int insertafter(node_t *n, elem_t e) {
  node_t *p;
  if (!(p = malloc(sizeof(node_t)))) return 0; /* eroare */
  p->e = e; p->next = n->next; n->next = p; return 1; /* succes */
} /* noul nod p e inserat după vechiul nod n */
```

Lookup: Implementare iterativă și recursivă

```
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;   /* s-a găsit, returnează poziţia */
  return NULL;               /* returnează NULL dacă nu s-a găsit */
}

node_t *lookup(list_t l, elem_t e)
{
  if (!l || e == l->e) return l; /* găsit sau sfârşit */
  return lookup(l->next, e); /* caută începând cu următorul */
}
```

Delete: implementare iterativă şi recursivă

```
/* parametrul 2 e nodul de şters, eventual găsit cu lookup */
int delete(list_t *pl, node_t *n) {/* poate schimba capul listei */
  node_t p = *pl;             /* deci ia parametru pointer la listă */
  if (!p || !n) return 0;     /* listă sau nod vid, eroare */
  if (p != n) {               /* n nu e primul nod din listă */
   while (p && p->next != n) p = p->next; /* caută predecesorul */
   if (!p) return 0;          /* nu s-a găsit n, eroare */
   p->next = n->next;         /* 'sare' peste nodul n */
  } else *pl = n->next;       /* n e primul, schimbă capul listei */
  free(n); return 1;          /* eliberează n, returneaza succes */
}
int delete (list_t *pl, node_t *n) /* presupune n nenul */
{
  if (!*pl) return 0;         /* listă vidă, deci nu s-a găsit */
  if (*pl == n) { *pl = n->next; free(n); return 1; } /* şterge */
  else return delete (&(*pl)->next, n); /* încearcă mai departe */
} /* apelată din nou cu *adresa* pointerului la următorul nod */
```

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); */
```

Utilizarea listelor: reprezentarea grafurilor

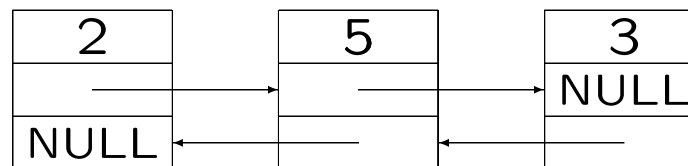
Graf: o colecţie de *noduri* şi *muchii* care leagă două noduri.
Exemplu: o mulţime de localităţi cu drumurile între ele
Un nod poate fi legat cu oricât de multe alte noduri ⇒ folosim listă

```
typedef struct n {
  int id; /* un număr pt. identificare */
  /* alte informaţii despre nod */
  struct e *edges; /* lista de muchii */
} node_t;
typedef struct e {
  struct n *dest;  /* celălalt capăt al muchiei */
  struct e *next;  /* pointer spre următoarea muchie */
} edge_t;
```

Liste dublu înlănţuite

În listele prezentate până acum:
– parcurgerea se poate face intr-un singur sens (nu şi înapoi)
– ştergerea necesită parcurgerea listei (chiar dat fiind nodul de şters) pentru a găsi nodul precedent care conţine legătura spre nodul dat
Soluţie: se reţin legături (pointeri) spre vecinii în ambele sensuri:

```
typedef struct n {
  elem_t info;           /* informaţia utilă din nod */
  struct n *prev, *next; /* pointeri spre cei vecini */
} node_t;
```



Capul listei are predecesor nul, coada listei are succesor nul.
Variantă: *listă circulară*, legând capul şi coada listei.

Liste dublu înlănțuite (cont.)

```
int insertafter(node_t *p, elem_t e) {
  node_t *n = malloc(sizeof(node_t));
  if (!n) return 0;          /* eroare, memorie insuficientă */
  n->info = e;
  n->prev = p; n->next = p->next; /* leagă nodul n la vecini */
  p->next->prev = n; p->next = n; /* leagă vecinii la nodul n */
  return 1;                  /* succes */
}
void delete(list_t *pl, node_t *p)
{ /* trebuie modificate cel mult două legături */
  if (p->next) p->next->prev=p->prev; /* dacă p nu e ultimul */
  if (p->prev) p->prev->next=p->next; /* dacă p nu e primul */
  else *pl = p->next; /* p era primul, schimbă capul listei */
  free(p);          /* eliberează memoria pentru nodul şters */
}
```