Arbori

5 aprilie 2004

Noţiunea de arbore. Terminologie

Arborii ne permit să structurăm ierarhic o mulţime de elemente
– structura de directoare şi fişiere într-un calculator
– arborele genealogic (o persoană, părinţii, bunicii, străbunicii, etc.)
– structura ierarhică pt. organizaţie (director, şefi departamente, etc.)
– circuite logice, expresii aritmetice/logice, baze de date strucurate

Un arbore e format din *noduri*, din care unul e *rădăcina*.
Fiecare nod în afară de rădăcină are un nod *părinte* $\Rightarrow$ ierarhizare

Definiţie recursivă:
– un arbore e fie un singur nod n (care reprezintă şi rădăcina arborelui)
– sau un nod n, împreună cu arborii $T_1, \ldots, T_k$ ai căror rădăcini $n_1, \ldots, n_k$ îl au pe n ca părinte
Nodurile n_i sunt *fiii* lui n, iar arborii T_i sunt *subarborii* lui n.

Un nod fără fii se mai numeşte nod *frunză*.
Uneori se include în definiţie şi *arborele vid*, fără nici un nod.

Tipul de date abstract arbore

Operaţii pe tipul abstract arbore

- init(arbore) /* iniţializează arborele ca fiind vid (NULL) */
- părinte(arbore, nod): nod /* părintele nodului în arbore sau NULL */
- fiu_stâng(arbore, nod): nod /* returnează primul fiu sau NULL */
- frate_drept(arbore, nod): nod /* return. următorul frate sau NULL */
- rădăcina(arbore): nod /* returnează rădăcina arborelui sau NULL */
- creează(nod, arbore_1, ..., arbore_k): arbore
  /* creează un arbore cu rădăcina şi subarborii specificaţi */
- inserează(arbore, nodparinte, nodnou) /* inserează la părinte */
- şterge(arbore, nod) /* şterge un nod dintr-un arbore */

Practic, cel mai des ne referim la arbore prin nodul său rădăcină
$\Rightarrow$ *arbore* şi *nod* vor fi acelaşi tip

Traversarea arborilor

În general, se ordinea în care sunt daţi fii unui nod *are* importanţă.

Nodurile unui arbore pot fi *traversate* (enumerate) în diverse moduri:

- Traversarea în *preordine*
  - se vizitează întâi rădăcina
  - apoi se traversează pe rând în *preordine* toţi subarborii
- Traversarea în *postordine*
  - se traversează pe rând în *postordine* toţi subarborii
  - apoi se vizitează rădăcina
- Traversarea în *inordine*
  - se traversează întâi în *inordine* primul subarbore (stâng)
  - se vizitează rădăcina
  - se traversează pe rând în *inordine* toţi ceilalţi subarborii

Obs. Definiţiile de mai sus sunt *recursive*.

Cazul de bază: pentru traversarea arborelui *vid* nu se face nimic

Traversarea arborilor (cont.)

```
typedef ??? node_t; /* vom discuta posibile structuri de date */
#define EMPTY ???           /* o valoare pentru arborele vid */
void preorder(tree_t n) { /   * arborele e dat prin rădăcină */
  tree_t c;
  if (n == EMPTY) return;
  visit(n); /* conţine ce trebuie făcut pt. fiecare nod */
  for (c = fiu_stang(n); c != EMPTY; c = frate_drept(n, c))
    preorder(c);
}

void postorder(tree_t n) {
  tree_t c;
  if (n == EMPTY) return;
  for (c = fiu_stang(n); c != EMPTY; c = frate_drept(n, c))
    postorder(c);
  visit(n);
}
```

Traversarea arborilor (cont.)

```
void inorder(tree_t n) {
  tree_t c;
  if (n == EMPTY) return;
  if ((c = fiu_stang(n)) != EMPTY) inorder(c);
  visit(n);
  for (; c != EMPTY; c = frate_drept(n, c)) inorder(c);
}
```

Observaţii:
– procedurile de traversare sunt scrise *independent* de reprezentarea arborelui; folosesc doar operaţiile (funcţiile) `fiu_stang` şi `frate_drept` şi valoarea `EMPTY` $\Rightarrow$ s-a definit într-adevăr un *tip de date abstract*
– preordine: dacă trebuie transmisă informaţie din părinte la fii
– postordine: dacă trebuie transmisă informaţie de la fii la părinte (ex. evaluarea unei expresii; numărarea nodurilor; adâncimea arborelui)
– inordine: ex. pentru sortarea cu arbori binari ordonaţi

Reprezentarea arborilor

Arborii pot fi reprezentaţi, ca şi listele, static, cu tablouri, folosind indici pentru referirea la nodurile fiu.
Cea mai frecventă reprezentare este însă dinamică, cu pointeri.

```
typedef struct n {
  /* aici se pune informaţia utilă din nod */
  struct n *fiu_stang, *frate_drept;
} node_t; /* node_t e un tip structură sinonim cu struct n */
typedef node_t *tree_t;   /* un arbore e un pointer la nod */
```

O altă variantă ar fi să folosim o *listă* separată pentru fii:

```
typedef struct n {
  /* informaţia utilă */
  struct l *fii;
} node_t;
```

```
typedef struct l {
  struct n *nod; /* pointer la nod */
  struct l *next; /* urm. în listă */
} list_t;
```

Uneori, se adaugă şi un pointer (redundant) părinte, dacă pt. problema dată e necesar accesul rapid şi eficient la părintele unui nod dat.

Arbori binari

Caz particular în care orice nod are doi fii: fiul stâng și cel drept
– în general: oricare (sau amândoi) pot lipsi (= arbore vid)
– uneori: arbore binar propriu-zis: fiecare nod are 0 sau 2 fii.

```
typedef struct n {
  /* informaţia utilă din nod */
  struct n *left, *right;
} node_t;
```

Exemple:
– reprezentarea unei expresii: nodurile intermediare conţin operatori, nodurile frunză conţin valori; calcul prin parcurgere în postordine (operatorii unari vor avea subarborele drept vid)
– arbori de decizie binari, pentru reprezentarea funcţiilor boolene
noduri intermediare: etichetate cu variabile; nodurile terminale: 0 și 1
arborele stâng: valoarea funcţiei când variabila respectivă e 0
arborele drept: valoarea funcţiei când variabila respectivă e 1

Arbori binari ordonați (de căutare)

Fiecare nod are o cheie (valoare) a unui tip ordonat (întreg, real, șir)
Pentru fiecare nod `c` din subarborele `n.left` avem `c.key <= n.key`
Pentru fiecare nod `c` din subarborele `n.right` avem `c.key >= n.key`

Folosiți pentru a păstra o mulțime de elemente, ordonată după chei, într-o structură flexibilă (nu tablou fix), cu căutare/modificare rapidă.

```
typedef int key_t; /* sau alt tip ordonat */
typedef struct n {
  key_t key; /* sau un alt tip ordonat */
  struct n *left, *right;
} node_t;
```

Căutarea și inserarea într-un arbore binar ordonat

```
node_t *search(node_t *n, key_t key) {
  if (!n) return NULL;        /* arbore nul, cheia nu s-a găsit */
  else if (key == n->key) return n; /* găsit, returnează nodul */
  else if (key < n->key) return search(n->left, key);
  else return search(n->right, key); /* caută într-un subarbore */
}
void insert(node_t **n, key_t key) { /* poate modifica *n */
  while (*n) /* caută un loc gol potrivit */
    /* variantă care acceptă duplicate, inserate la stânga */
    /* fără duplicate: se iese la test de egalitate */
    if (key <= (*n)->key) n = &(*n)->left;   /* caută la stânga */
    else n = &(*n)->right;                   /* caută la dreapta */
  if (!(*n = malloc(sizeof(node_t)))) return; /* aloca nodul */
  (*n)->left = (*n)->right = NULL;        /* noul nod e terminal */
  (*n)->key = key;
}
```

Ştergerea dintr-un arbore binar ordonat

```
void delete (node_t **n, key_t key) { /* poate modifica *n */
  while (*n) {
    if (key == (*n)->key) { /* şterge. caz simplu: 1 fiu  */
      node_t *p = *n;
      if (!(*n)->left) *n = (*n)->right;
      else if (!(*n)->right) *n = (*n)->left;
      else { /* 2 fii. coboară spre dreapta în cel stăng */
        do n = &(*n)->right while (*n);
        *n = p->right; /* inserează subarborele drept la *n */
      }
      free(p); return;  /* eliberează memoria pentru nodul şters */
    } else if (key < (*n)->key) n = &(*n)->left;
    else n = &(*n)->right;
  }
}
```

Sortarea cu arbori binari ordonaţi

– se creează un arbore binar ordonat (vid)
– se inserează pe rând elementele de sortat
– se parcurge arborele în *inordine* $\Rightarrow$ se obţin elementele în ordine

Complexitate:
– toate operaţiile (căutare, inserţie, sortare) au complexitate liniară în adâncimea h a arborelui
– în cazul ideal (și mediu), $h \simeq \log n$ (nr. de noduri)
– în cazul defavorabil: $h = n$ (deja sortat $\Rightarrow$ arborele devine listă)
– sortarea e $O(n \log n)$ în medie, dar poate fi $O(n^2)$

Soluţie: diverse tipuri de arbori binari echilibraţi