Tipuri de date abstracte. Stive. Cozi

8 ianuarie 2004

Tipuri de date abstracte

- *tip de date*: mulţimea valorilor pe care le poate lua o variabilă.
  – fiecare tip de date are definiţi anumiţi operatori.
- funcţiile / procedurile pot fi văzute ca o extindere a operatorilor
  Ex.: concatenarea a două şiruri; înmulţirea a două matrici
  (există chiar ca operatori în limbaje mai bogate în tipuri)
- *tip de date abstract*: un model matematic + operaţii pe acel model
  Ex.: tipul *mulţime* (cu test de membru, reuniune, intersecţie)
- *structură de date*: colecţie de variabile (posibil de tipuri diferite), pentru implementarea tipurilor de date abstracte într-un program

Tipul de date abstract *stivă*

– o listă (un șir) în care elementele sunt adăugate și extrase la același capăt, în ordinea inversă introducerii (LIFO - last in, first out)
– denumire inspirată din realitate (ex. o stivă de cărți)

Operații pt. tipul abstract *stivă*

- init(stiva) /* inițializează stiva */
- empty(stiva) /* testează dacă stiva e goală */
- push(stiva, element) /* pune pe stivă */
  /* pop și top necesită ca precondiție o stivă nevidă */
- pop(stiva) : element /* extrage și returnează vârful stivei */
- top(stiva) : element /* returnează vârful stivei */
- full(stiva) /* testează dacă stiva e plină */

Implementarea stivei cu un tablou

```
#define MAX 100 /* dimensiunea maximă a stivei */
typedef int elem_t /* sau orice alt tip dorit */
typedef struct s {
  elem_t t[MAX];
  int sp; /* indicele stivei */
} stack;
void init(stack *s) { s->sp = 0; }
int empty(stack *s) { return s->sp == 0; }
void push(stack *s, elem_t e) { if (sp < MAX) s->t[s->sp++] = e; }
elem_t pop(stack *s) { return s->sp ? s->t[--s->sp] : 0; }
elem_t top(stack *s) { return s->sp ? s->t[s->sp-1] : 0; }
int full(stack *s) { return s->sp == MAX; }
```

Implementarea stivei cu memorie dinamică

```
typedef int elem_t;
typedef struct { elem_t *base, *sp, *lim; } stack;
void init(stack *s) { s->base = s->sp = s->lim = NULL; }
int empty(stack *s) { return s->sp == s->base; }
void push(stack *s, elem_t e) {
  if (s->sp == s->lim) {
    elem_t *p=realloc(s->base,(s->sp-s->base+64)*sizeof(elem_t));
    if (!p) return; /* eroare, memorie insuficientă */
    s->sp += p - s->base; s->lim += (p - s->base) + 64; s->base = p;
  }
  *s->sp++ = e;
}
elem_t pop(stack *s) { return (s->sp!=s->base) ? *--s->sp : 0; }
elem_t top(stack *s) { return (s->sp!=s->base) ? *(s->sp-1) : 0; }
```

Implementarea stivei cu memorie dinamică (cont.)

Variantă cu dealocarea memoriei in pop() (simetric cu push):

```
elem_t pop(stack *s) {
  elem_t e = (s->sp!=s->base) ? *--s->sp : 0;
  if (s->lim - s->sp == 64) { /* limită pt. dealocare */
    p = realloc(s->base, (s->sp-s->base)*sizeof(elem_t));
    s->lim = s->sp += p - s->base; s->base = p;
  }
  return e;
}
```

Încapsularea

== faptul că detaliile de implementare sunt ascunse de utilizator
Pentru folosirea stivei, ar fi suficient un fișier `<stiva.h>` cu

```
typedef int elem_t; /* trebuie specificat tipul elementului */
typedef struct s stack; /* tip incomplet */
void init(stack *s);
int empty(stack *s);
void push(stack *s, elem_t e);
elem_t pop(stack *s);
int full(stack *s);
```

Implementarea: într-un fișier compilat separat, invizibil utilizatorului
– trebuie recompilat însă dacă schimbăm definiția elementului
– o soluție: stivă de pointeri `void *`, nu obiecte propriu-zise

Stiva și apelurile de funcţii

Din apelurile încuibate de funcţii se revine în ordine inversă fată de apel
⇒ stiva este foarte naturală pentru implementare
– arhitectura procesorului: registru pentru vârful stivei
– pe stivă se pun în ordine: parametrii, apoi adresa de revenire, apoi în funcţie se creează variabilele locale
⇒ variabilele locale dispar la revenirea din funcţie
⇒ nu e corectă returnarea *adresei* unei variabile locale

Tipul de date abstract *coadă*

– o listă (un șir) în care inserarea se face la un capăt, și extragerea la celălalt, în ordinea introducerii elementelor (FIFO = first in, first out)

Operații pt. tipul abstract *coadă*

- init(coada) /* inițializează coada */
- empty(coada) /* testează dacă coada e goală */
- enqueue(coada, element) /* adaugă la coadă, dacă nu e plină */
- dequeue(coada) : element /* extrage din coadă nevidă */
- full(coada) /* testează dacă coada e plină */

Implementarea cozii cu un tablou circular

```
#define MAX 100 /* dimensiunea maximă a cozii */
typedef int elem_t /* sau orice alt tip dorit */
typedef struct {
  elem_t t[MAX];
  int head, tail; /* inserare la tail, extragere de la head */
} queue;
void init(queue *q) { q->tail = q->head = 0; }
int empty(queue *q) { return q->head==q->tail; }
void enqueue(queue *q, elem_t e) {
  if (((q->tail+1)%MAX) == q->head) return; /* coadă plină */
  q->t[q->tail++] = e; q->tail %= MAX;
}
elem_t dequeue(queue *q) {
  if (q->head==q->tail) return 0; /* coadă vidă */
  { elem_t e = q->t[q->head++]; q->head %= MAX; return e; }
}
int full(queue *q) { return ((q->tail+1)%MAX) == q->head; }
```