Tipuri de date abstracte. Stive. Cozi

17 mai 2005

Recapitulare: Liste

1. Definim un tip structura pentru un element din lista

```
struct nod {
  int num; char sir[20]; char *ps; // etc.: informatia utila
  struct nod *next; // pointer la nodul urmator
};
```

Putem sa declaram acum noduri de lista sau pointeri la ei:

```
struct nod nod1; struct nod *head;
```

1a. Pentru a nu scrie tot timpul `struct nod` definim un sinonim:

```
typedef struct nod node_t; // node_t e sinonim cu struct nod
```

sau direct in declaratia dinainte:

```
typedef struct nod {
  // aici vine informatia utila din nod
  struct nod *next; // pointer la nodul urmator
} node_t;
```

Putem declara acum: `node_t nod1; node_t *head;`

Recapitulare: Liste

2. Pentru a crea un nou element din lista, il alocam dinamic:
`node_t *n; n = malloc(sizeof(node_t));`

3. Completam apoi informatia utila din nod:
`int x; scanf("%d", &x); n->num = x;` Pentru un camp tablou (sir) de caractere, acesta trebuie copiat:
`char s[20]; scanf("%19s", s); strcpy(n->sir, s);` Pentru un camp pointer (de ex. la sir), acesta trebuie intai alocat!
`char s[20]; scanf("%19s", s);`
`n->ps = malloc(strlen(s)+1); strcpy(n->ps, s);`

4. Legam apoi nodul la locul dorit in lista. Initial, `node_t *head = NULL;`
a) pentru inserare in capul listei, legam `n->next = head; head = n;`
b) pentru inserare in coada listei, memoram si `node_t *tail;` si legam:
`if (head==NULL) head = n; else tail->next = n;` si apoi `tail = n;`
(daca lista e goala, noul element devine capul listei, altfel e inserat dupa vechea coada; in orice caz, el devine noua coada)

Tipuri de date abstracte. Stive. Cozi

- *tip de date*: mulțimea valorilor pe care le poate lua o variabilă.
  – fiecare tip de date are definiți anumiți operatori.
- funcțiile / procedurile pot fi văzute ca o extindere a operatorilor
  Ex.: concatenarea a două șiruri; înmulțirea a două matrici
  (există chiar ca operatori în limbaje mai bogate în tipuri)
- *tip de date abstract*: un model matematic + operații pe acel model
  Ex.: tipul *mulțime* (cu test de membru, reuniune, intersecție)
- *structură de date*: colecție 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*

- stiva create() /* crează o stivă nouă */
- 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 maxima a stivei
typedef int elem_t; // sau alt tip de element dorit
typedef struct _stk {
  elem_t t[MAX];
  int sp; // indicele stivei
} stack; // tipul stack e o structura
stack *create(void) { stack s = malloc(sizeof(struct _stk));
  if (s) s->sp = 0; return s; }
// empty, full si top nu modifica stiva, push si pop da
int empty(const stack *s) { return s->sp == 0; }
int full(const stack *s) { return s->sp == MAX; }
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(const stack *s) { return s->sp ? s->t[s->sp-1] : 0; }
```

Implementarea stivei cu memorie dinamică

```
typedef int elem_t;
typedef struct _stk { elem_t *base, *sp, *lim; } stack;
stack *create(void) { stack s = malloc(sizeof(struct _stk));
  if (s) { s->base = s->sp = s->lim = NULL; } return s; }
int empty(stack *s) { return s->sp == s->base; }
int full(stack *s) { return 0; }
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; }
```

Obs: toate functiile in afara de `empty()` si `top()` modifica stiva, deci e necesara transmiterea unui *pointer* la stiva !

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 _stk *stack; // declaratie incompleta de tip
// stack e tip pointer la o structura neprecizata inca
stack create(void);
int empty(stack s);
int full(stack s);
void push(stack *s, elem_t e); // aici, push si pop au param. pointer
elem_t pop(stack *s); // pentru ca modifica stiva
elem_t top(stack s);
```

Implementarea: într-un fișier `stiva.c` invizibil utilizatorului

– acesta poate fi compilat separat

– si apoi linkeditat cu programul principal care utilizeaza stiva

Stiva și apelurile de funcții

Pentru apelurile de functii, calculatorul foloseste o *stiva*.
Procesorul are un registru special in care tine minte varful stivei.
Cand apelam o functie, pe stiva se pun urmatoarele:
– argumentele (parametrii) functiei (de regula primul cel mai sus)
– adresa instruct. de dupa apel (unde se revine la terminarea functiei)
– apoi pe stiva se creeaza variabilele locale (ele dispar la revenirea din funcție ⇒ nu e corectă returnarea *adresei* unei variabile locale

Din functii se revine in ordine inversa in care au fost apelate
(daca f apeleaza pe g si aceasta pe h, se revine din h, apoi din g, si f)
⇒ stiva este foarte naturală pentru implementare

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; }
```