Università Roma La Sapienza
Corsi di Laurea
Informatica/Tecnologie Informatiche

Pile

Prof. Stefano Guerrini
guerrini@di.uniroma1.it

Programmazione II (can. P-Z)
A.A. 2005-06

Pile: implementazioni

- **Liste concatenate**
  - Ogni elemento della lista contiene un elemento della pila,
  - la testa della lista è anche la testa della pila.
- **Vettore** (di dimensione N)
  - Gli elementi della pila vengono inseriti nelle prime posizioni del vettore *V*,
  - se la pila contiene *n* elementi
    - l'elemento *V[n-1]* è la testa della pila;
    - la posizione *V[n]* è quella in cui andrà inserito il prossimo elemento.

Pile: liste /1

- Supponiamo di voler mantenere anche il numero di elementi presenti nella pila.
- Rappresento una pila con una struct di due campi:
  - il puntatore alla lista degli elementi
  - il numero di elementi nella pila (lungh. della lista)
- La pila vuota corrisponde alla lista vuota (NULL).
- La push alloca un nuovo nodo e lo inserisce in testa alla lista (costo $\Theta(1)$)
- La pop elimina dalla lista e dealloca il nodo in testa alla lista (costo $\Theta(1)$).

Pile: liste /2

```
#define   EMPTY      0

typedef   ...    data;

struct elem {        /* un elemento nella pila */
   data          d;
   struct elem   *next;
};

typedef   struct elem   elem;

struct pila {
   int    size;      /* dimensione pila     */
   int    cnt;       /* conta gli elementi  */
   elem   *top;      /* lista degli elementi */
};

typedef   struct pila    pila;
```

Pile: liste /3

```
void      empty(stack *stk);

boolean   isEmpty(const stack *stk);

boolean   isFull(const stack *stk);

void      push(data d, stack *stk);

data      pop(stack *stk);

data      top(stack *stk);
```

Pile: liste /4

```
void initialize(stack *stk) {
   stk -> cnt = 0;
   stk -> top = NULL;
}

boolean isEmpty(const stack *stk) {
   return ((boolean) (stk -> cnt == EMPTY));
}

boolean isFull(const stack *stk) {
   return ((boolean) (stk -> cnt == FULL));
}
```

Pile: liste /5

```
void initialize(stack *stk) {
   stk -> cnt = 0;
   stk -> top = NULL;
}

void push(data d, stack *stk) {
   elem   *p;

   p = malloc(sizeof(elem));
   p -> d = d;
   p -> next = stk -> top;
   stk -> top = p;
   stk -> cnt++;
}
```

Pile: liste /6

```
data pop(stack *stk) {
   data   d;
   elem   *p;

   d = stk -> top -> d;
   p = stk -> top;
   stk -> top = stk -> top -> next;
   stk -> cnt--;
   free(p);
   return d;
}

data top(stack *stk) {
   return (stk -> top -> d);
}
```

Pile: vettore /1

- Supponiamo di voler mantenere anche in numero di elementi presenti nella pila.
- Rappresento una pila con una struct di due campi:
  - un vettore (di dimensione *N*)
  - il numero di elementi nella pila (lungh. della lista)
- La pila vuota corrisponde alla lista vuota (NULL).
- La push alloca un nuovo nodo e lo inserisce in testa alla lista (costo $\Theta(1)$)
- La pop elimina dalla lista e dealloca il nodo in testa alla lista (costo $\Theta(1)$).

Pile: vettore /2

```
#define   MAX_LEN    1000
#define   EMPTY      -1
#define   FULL       (MAX_LEN - 1)

typedef struct stack {
   data   s[MAX_LEN];
   int    top;
} stack;

void      reset(stack *stk);
void      push(data c, stack *stk);
char      pop(stack *stk);
char      top(const stack *stk);
boolean   empty(const stack *stk);
boolean   full(const stack *stk);
```

Pile: vettore /3

```
void reset(stack *stk) {
   stk -> top = EMPTY;
}

boolean empty(const stack *stk){
   return ((boolean) (stk -> top == EMPTY));
}

boolean full(const stack *stk) {
   return ((boolean) (stk -> top == FULL));
}
```

Pile: vettore /4

```
void push(char c, stack *stk) {
   stk -> top++;
   stk -> s[stk -> top] = c;
}

char pop(stack *stk){
   return (stk -> s[stk -> top--]);
}

char top(const stack *stk){
   return (stk -> s[stk -> top]);
}
```

Confronto implementazioni

- Nella implementazione con vettori, le operazioni sono molto semplici, ma la dimensione del vettore, limita il numero massimo di elementi nella pila.

Parentesi.c /1

```
/*
 * Verifica se una sequenza di parentesi chiusa da cend
 * e' bilanciata.
 * Ignora tutti i caratteri diversi da parentesi, cend o EOF.
 */
int isBalanced(char cend) {
  stack stk = emptyStack(MAX_NESTING);
  int t, c = next(cend);
  char *p;

  while (c != cend) {
    if (index(open, c) != NULL) {
      pushStack(stk, c);
      c = next(cend);
    } else if ((p = index(close, c)) != NULL) {
      if (isEmptyStack(stk)) {
        printf("Parentesi chiusa inattesa: %c\n", c);
        return ERR_UNB_CL;
      }
```

Parentesi.c /2

```
      popStack(stk, &t);
      if (open[p-close] != t) {
        printf("Parentesi chiusa inattesa: %c\n", c);
        /* stampa il contenuto dello stack */
        printf("Parentesi non bilanciate: %c", t);
        do {
          popStack(stk, &t);
          putchar(t);
        } while (!isEmptyStack(stk));
        return ERR_UNB_CL;
      }
      c = next(cend);
    } else if (c == EOF) {
      printf("EOF inatteso!\n");
      return ERR_EOF;
    } else {
      printf("Carattere inatteso!\n");
      return ERR_INV_CH;
    }
  }  /* while (c != cend) */
```

Parentesi.c /3

```
    if (isEmptyStack(stk))
      return OK;
    else {
      /* stampa il contenuto dello stack */
      printf("Parentesi non bilanciate: ");
      do {
        popStack(stk, &t);
        putchar(t);
      } while (!isEmptyStack(stk));
      return ERR_UNB_OP;
    }
  }
```

Parentesi.c /4

```c
/*
 * parentesi aperte e chiuse
 * open[i] e close[i] forma una coppia di
 *   parentesi aperta/chiusa
 */
char open[] = {'(','[','{','<','\0'};
char close[] = {')',']','}','>','\0'};

/*
 * Legge il successivo carattere che e' uguale a cend o EOF
 * o che e' una parentesi
 */
int next(int cend) {
  int c = getchar();

  while (index(open, c) == NULL
      && index(close, c) == NULL
      && c != cend && c != EOF)
    c = getchar();
  return c;
}
```

Parentesi.h /1

```c
#define    MAX_NESTING   256

/*
 * parentesi aperte e chiuse
 * open[i] e close[i] forma una coppia di
 *   parentesi aperta/chiusa
 */
extern char open[];
extern char close[];

/* valori di ritorno della funzione isBalanced */
typedef    enum error_t {
  OK = 0,
  ERR_UNB_CL,
  ERR_UNB_OP,
  ERR_EOF,
  ERR_INV_CH
} error_t;
```

Parentesi.h /2

```c
/*
 * Verifica se una sequenza di parentesi chiusa da cend
 * e' bilanciata.
 * Ignora tutti i caratteri diversi da parentesi, cend o EOF.
 */
int isBalanced(char cend);
```

ParentesiRic.c /1

```c
/*
 * Cerca la successiva parentesi chiusa non bilanciata
 * Ritorna la parentesi chiusa trovata, oppure
 *   la parentesi chiusa nn bilanciata trovata
 *   OK se trova cend dopo una sequenza di parentesi bilanciate
 *   -ERR_UNB_OP se trova con delle parentesi aperte non bilanciate
 *   -ERR_UNB_CL se trova una parentesi chiusa che cerca di chiudere una
parentesi di tipo diverso
 *   -ERR_EOF se trova EOF prima di cend
 * Ignora tutti i caratteri diversi da parentesi, cend o EOF
 */
int nextClosed(char cend) {
  int c = next(cend);
  char *p;

  while (index(close, c) == NULL) {
    if (c == cend)
      return OK;
    if (c == EOF) {
      printf("EOF inatteso!\n");
      return -ERR_EOF;
    }
```

ParentesiRic.c /2

```
if ((p = index(open, c)) != NULL) {
   int nc = nextClosed(cend);
   if (nc > 0) {
      /* verifica se parentesi chiusa trovata chiude
         correttamente la parentesi in sospeso */
      if (nc == close[p-open])
         /* OK. Aperta in sopseso e' stata chiusa.
            Leggi un'altra parentesi */
         c = next(cend);
      else {
         printf("Parentesi chiusa inattesa: %c\n");
         printf("Parentesi non bilanciate: %c", c);
         return -ERR_UNB_CL;
      }
}
```

ParentesiRic.c /3

```
         else if (nc == OK) {
            printf("Parentesi non bilanciate: %c", c);
            return ERR_UNB_OP;
         } else if (nc == -ERR_UNB_OP || nc == -ERR_UNB_CL) {
            printf("%c", c);
            return nc;
         } else /* altra situazione di errore */
            return nc;
      }
   }
   return c;
}
```