Corso di Programmazione

II Accertamento del 17 Giugno 2009

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Astrazione sui dati in Scheme

Una realizzazione alternativa della struttura *coda con priorità* si basa sugli alberi binari di ricerca, per i quali vale la seguente proprietà: qualunque sia il nodo considerato, il valore associato ad esso è maggiore o uguale ai valori associati ai nodi del rispettivo sottoalbero sinistro e minore o uguale a quelli del sottoalbero destro. In particolare, nel codice riportato sotto, la coda con priorità è rappresentata da una lista di tre componenti: la funzione peso (che misura la priorità di ciascun elemento), il numero complessivo di elementi contenuti nella struttura, l'albero binario di ricerca (dove sono rappresentati gli elementi). Un albero vuoto è rappresentato dalla lista vuota; un albero non vuoto è rappresentato da una lista di tre componenti: l'elemento radice, i corrispondenti sottoalberi sinistro e destro, rappresentati allo stesso modo. Completa la realizzazione codificando le procedure `first-item` e `but-first`.

```
;; Costruttore di una coda con priorità vuota

(define priority-queue    ; valore: coda con priorità
  (lambda (weight)        ; weight: procedura [elementi -> pesi]
    (list weight 0 null)  ; rispettivamente: funzione priorità, numero di elementi, albero binario
    ))

;; Inserimento di un nuovo elemento itm nella coda con priorità queue

(define add-item          ; valore: coda con priorità (risultante dall'introduzione di itm in queue)
  (lambda (itm queue)     ; itm: elemento, queue: coda con priorità
    (list (car queue) (+ (cadr queue) 1) (tree-add (car queue) itm (caddr queue)))
    ))

(define tree-add          ; inserimento di itm nell'albero binario di ricerca tree
  (lambda (wgt itm tree)
    (cond ((null? tree)                    ; tree vuoto: itm è il solo elemento dell'albero risultante
           (list itm null null))
          ((< (wgt itm) (wgt (car tree)))  ; itm < radice di tree: itm viene inserito nel sottoalbero sinistro
           (list (car tree) (tree-add wgt itm (cadr tree)) (caddr tree)))
          (else                            ; itm ≥ radice di tree: itm viene inserito nel sottoalbero destro
           (list (car tree) (cadr tree) (tree-add wgt itm (caddr tree))))
          )))

;; Numero di elementi della coda con priorità

(define size cadr)

;; Elemento di priorità più elevata nella struttura queue

(define first-item        ; valore: elemento (di priorità massima in queue)
  (lambda (queue) ...))   ; queue: coda con priorità

;; Rimozione dell'elemento di priorità più elevata dalla struttura queue

(define but-first         ; valore: coda con priorità (risultante dalla rimozione dell'elemento di priorità massima)
  (lambda (queue) ...))   ; queue: coda con priorità
```

```
(define first-item       ; valore: elemento
  (lambda (queue)        ; queue: coda con priorita'
    (rightmost (caddr queue))
    ))

(define rightmost
  (lambda (tree)
    (if (null? (caddr tree))
        (car tree)
        (rightmost (caddr tree))
        )))
```

(Astrazione sui dati in Scheme)

```
(define but-first        ; valore: coda con priorita'
  (lambda (queue)        ; queue: coda con priorita'
    (list (car queue) (- (cadr queue) 1) (but-rightmost (caddr queue)))
    ))

(define but-rightmost
  (lambda (tree)
    (if (null? (caddr tree))
        (cadr tree)
        (list (car tree) (cadr tree) (but-rightmost (caddr tree)))
        )))
```

2. Memoization

Considera il seguente metodo statico formalizzato nel linguaggio *Java*:

```
public static int bc( int n, int k ) {  // 0 < k < n
  if ( k == 1 ) {
    return n;
  } else if ( 2*k > n ) {
    return bc( n, n-k );
  } else {
    return bc( n-1, k-1 ) + bc( n-1, k );
  }
}
```

Trasforma il programma ricorsivo applicando opportunamente la tecnica di *memoization*.

```
  public static int bcMem( int n, int k ) {  // 0 < k < n

    int[][] h = new int[ n+1 ][];

    for ( int i=2; i<=n; i++ ) {
      h[i] = new int[ i ];
      for ( int j=1; i<i; i++ ) {
        h[i][j] = UNDEFINED;
    }}
    return bcRec( n, k, h );
  }


  public static int bcRec( int n, int k, int[][] h ) {  // 0 < k < n

    if ( h[n][k] == UNDEFINED ) {
      if ( k == 1 ) {
        h[n][k] = n;
      } else if ( 2*k > n ) {
        h[n][k] = bcRec( n, n-k, h );
      } else {
        h[n][k] = bcRec( n-1, k-1, h ) + bcRec( n-1, k, h );
    }}
    return h[n][k];
  }


  public static final int UNDEFINED = 0;
```

3. Classi in Java

Qui sotto è riportato il codice Java della classe `BinaryString`, proposta come esercizio a lezione.

```
public class BinaryString {

  private String bin;

  public BinaryString( int n ) {
    bin = "" + ( n % 2 );  n = n / 2;
    while ( n > 0 ) {
      bin = ( n % 2 ) + bin;  n = n / 2;
    }
  }

  public int value() {
    int n = 0;
    for ( int k=0; k<bin.length(); k=k+1 ) {
      n = 2*n + ((bin.charAt(k)=='0') ? 0 : 1);
    }
    return n;
  }

  public void rotateLeft() {
    bin = bin.substring(1) + bin.charAt(0);
  }

}    // BinaryString
```

Immagina di aver eseguito il seguente codice ($n \geq 100$):

```
BinaryString[] bs = new BinaryString[ n+1 ];
for ( int k=1; k<=n; k=k+1 ) {
  bs[k] = new BinaryString( k );  bs[k].rotateLeft();
}
```

Con riferimento allo stato così ottenuto, riporta i risultati restituiti da ciascuna delle invocazioni del metodo `value()`:

`bs[1].value()`	→	1	`bs[7].value()`	→	7
`bs[2].value()`	→	1	`bs[8].value()`	→	1
`bs[3].value()`	→	3	`bs[24].value()`	→	17

4. Oggetti in Java

Supponi che sia disponibile una classe *Stack*, simile a quella discussa a lezione, per rappresentare stack i cui elementi sono caratteri (di tipo `char`). Il protocollo di questa classe comprende: un costruttore che crea una struttura vuota; il metodo `empty()` che consente di determinare se lo stack è vuoto; il metodo `push(`*c*`)` che aggiunge il carattere *c* in cima allo stack; il metodo `pop()` che rimuove dalla struttura e restituisce il carattere in cima allo stack.
Una sequenza di parentesi rappresenta una parentitizzazione corretta se ogni parentesi chiusa corrisponde a una precedente parentesi aperta dello stesso tipo. Per esempio, le stringhe `"[]"`, `"([]())()()"`, `"{[()(()())]()}[()()]"` rappresentano parentetizzazioni corrette, mentre `"())("`, `"(()())()("`, `"{[()(()())}()}[()()]"` risultano scorrette. È facile verificare questa proprietà introducendo le parentesi aperte in uno stack e rimuovendole all'occorrenza di una parentesi chiusa, quando la corrispondente parentesi aperta deve trovarsi in cima allo stack.
Completa il codice del metodo statico *correct* per verificare se la stringa di parentesi *pars* passata come argomento rappresenta una parentetizzazione corretta — nel qual caso restituisce *true*, altrimenti restituisce *false*. A tal fine, assumi che si possano utilizzare tre tipi di parentesi aperte e chiuse: parentesi tonde, quadre e graffe.

```
public static boolean correct( String pars ) {  // pars: stringa di '(', ')', '[', ']', '{', '}'

  Stack stk = new Stack();
  for ( int i=0; i<pars.length(); i=i+1 ) {
    if ( (pars.charAt(i) == '(') || (pars.charAt(i) == '[') || (pars.charAt(i) == '{') ) {
      stk.push( pars.charAt(i) );
    } else {

      if ( stk.empty() ) {
        return false;
      } else {
        char o = stk.pop(), c = pars.charAt(i);
        if ( !(((o=='(')&&(c==')'))||((o=='[')&&(c==']'))||((o=='{')&&(c=='}'))) ) {
          return false;
      }}

  }}
  return   stk.empty();                                                              ;
}
```

5. Asserzioni e invarianti

Il seguente metodo in Java *fonde* due array ordinati di interi in un array ordinato, che viene restituito al termine dell'esecuzione. Si assume pertanto che gli array *u* e *v* passati come argomenti siano già ordinati; eventuali elementi che occorrono ripetutamente, saranno ripetuti anche nel risultato. Commenta il programma introducento opportune asserzioni: precondizioni, postcondizioni e invarianti del primo comando iterativo; proponi inoltre una funzione di terminazione, sempre relativamente al primo ciclo. (In questo esercizio non è richiesta alcuna dimostrazione.)

```
public static int[] merge( int[] u, int[] v ) {

  /*  Pre:  ∀ x, y ∈ [0, u.length–1] . ( x < y  ⇒  u[x] ≤ u[y] )

        ∀ x, y ∈ [0, v.length–1] . ( x < y  ⇒  v[x] ≤ v[y] )
  */

  int[] m = new int[ u.length + v.length ];
  int i = 0, j = 0, k = 0;
  while ( (i < u.length) && (j < v.length) ) {

    /*  Inv:  i ≤ u.length  ∧  j ≤ v.length  ∧  ∀ x, y ∈ [0, k–1] . ( x < y  ⇒  m[x] ≤ m[y] )

          ∀x∈[0,k–1], y∈[i,u.length–1] . m[x]≤u[y]  ∧  ∀x∈[0,k–1], y∈[j,v.length–1] . m[x]≤v[y]

          ∀x∈[0,i–1] . ∃y∈[0,k–1] . m[y]=u[x]  ∧  ∀x∈[0,j–1] . ∃y∈[0,k–1] . m[y]=v[x]
    */

    /*  Term:  m.length – k                                                    */

    if ( u[i] < v[j] ) {
      m[k] = u[i];  i = i + 1;
    } else {
      m[k] = v[j];  j = j + 1;
    }
    k = k + 1;
  }
  while ( i < u.length ) {
    m[k] = u[i];  i = i + 1;  k = k + 1;
  }
  while ( j < v.length ) {
    m[k] = v[j];  j = j + 1;  k = k + 1;
  }

  /*  Post:  ∀ x, y ∈ [0, m.length] . ( x < y  ⇒  m[x] ≤ m[y] )

        ∀x∈[0,u.length–1].∃y∈[0,k–1] . m[y]=u[x] ∧ ∀x∈[0,v.length–1].∃y∈[0,k–1] . m[y]=v[x]
  */

  return m;
}
```