

Algoritmi e Strutture Dati  
&  
Laboratorio di Algoritmi e Programmazione

— Appello del 30 Giugno 2009 —

**Esercizio 1 - ASD**

1. Sia  $T(n) = 4T(n/2) + n^2 + 3n$ . Considerare ciascuna delle seguenti affermazioni e dire se è corretta o no. Giustificare la risposta.

(a)  $T(n) = \Omega(n)$

(b)  $T(n) = O(n^3)$

(c)  $T(n) = O(n \lg n)$

2. Sia  $T(n) = T(n/6) + T(n/3) + n$ . Verificare usando il metodo di sostituzione la correttezza della seguente affermazione.  $T(n) = O(n \lg n)$

**Esercizio 2 - ASD**

Si consideri il seguente array  $A = [14, 5, 12, 9, 7, 8, 24, 6, 10, 5]$  e lo si trasformi applicando l'algoritmo BuildMaxheap.

**Esercizio 3 - ASD**

1. Si sviluppi un algoritmo che, dato un albero generale  $T$  rappresentato tramite gli attributi: **fratello**, **figlio** e **padre** e un intero  $k > 0$ , calcola il numero dei nodi di grado  $k$  presenti in  $T$ . (Ricordiamo che il grado di un nodo di un albero è pari al numero dei suoi figli.)

2. Si dimostri la correttezza dell'algoritmo proposto.

**Esercizio 4 - ASD**

Sia  $T$  un albero R/B. Per ciascuna delle seguenti affermazioni dire se essa è vera o falsa. Giustificare la risposta.

- L'operazione di inserimento di una nuova chiave in  $T$  ha complessità logaritmica nel numero dei nodi di  $T$ .
- L'operazione di ricerca di una chiave in  $T$  ha complessità logaritmica nel numero dei nodi di  $T$ .
- Per ogni nodo  $x$  di  $T$ , su ogni cammino da  $x$  alla radice esiste almeno un nodo rosso.
- Per ogni nodo  $x$  di  $T$ , su ogni cammino da  $x$  ad una foglia esiste lo stesso numero di nodi neri.

## Esercizio 1 (Laboratorio)

Date le classi *StackArray* (che realizza uno stack con un array) e *QueueArray* (che realizza una coda con un array) sviluppate durante il corso, si richiede di completare l'implementazione dei due metodi della seguente classe:

```
import Queues.QueueArray;
import Stacks.StackArray;
public class Esercizio1 {

    // pre: S non nullo
    // post: scambia l'ordine del primo e dell'ultimo elemento
    //        presenti nello stack S, lasciando inalterati tutti
    //        gli altri elementi. Se S contiene meno di due elementi
    //        allora rimane inalterato
    public static void scambiaS(StackArray S) {...}

    // pre: Q non nulla
    // post: scambia l'ordine del primo e dell'ultimo elemento
    //        presenti nella coda Q, lasciando inalterati tutti
    //        gli altri elementi. Se Q contiene meno di due elementi
    //        allora rimane inalterata
    public static void scambiaQ(QueueArray Q) {...}
}
```

È possibile utilizzare strutture dati di appoggio solo di tipo *StackArray* e solo se strettamente necessarie.

## Esercizio 2 (Laboratorio)

Data la definizione:

Un nodo  $n$  di un albero binario si dice *nodo sinistro* se è figlio sinistro di suo padre. Si dice invece *nodo destro* se è figlio destro di suo padre. La radice è un nodo a parte, cioè non è né sinistro né destro.

Si consideri il package *BinTrees* sviluppato durante il corso e relativo agli alberi binari. Si richiede di aggiungere alla classe *BinaryTree* un'implementazione ricorsiva del seguente metodo:

```
// post: ritorna la differenza tra il numero di nodi sinistri e il numero
//        di nodi destri dell'albero
public int nodiSxmenoDx() {...}
```

Il metodo deve essere lineare rispetto al numero di nodi dell'albero. Se necessario, è possibile utilizzare metodi privati di supporto.

```

***** classe StackArray *****
package Stacks;
public class StackArray implements Stack {
    private static final int MAX=100; // dimensione massima dello stack
    private Object[] S; // lo stack
    private int head; // puntatore al top dello stack

    // post: costruisce uno stack vuoto
    public StackArray() {...}

    // post: ritorna il numero di elementi nello stack
    public int size() {...}

    // post: ritorna lo stack vuoto
    public void clear() {...}

    // post: ritorna true sse lo stack e' vuoto
    public boolean isEmpty() {...}

    // post: ritorna true sse la coda e' piena
    public boolean isFull() {...}

    // pre: stack non vuoto!
    // post: ritorna l'oggetto in cima allo stack
    public Object top() {...}

    // post: inserisce ob in cima allo stack
    public void push(Object ob) {...}

    // pre: stack non vuoto!
    // post: ritorna e rimuove l'elemento in cima allo stack
    public Object pop() {...}
}

***** classe QueueArray *****
package Queues;
public class QueueArray implements Queue {
    private static final int MAX=100; // dimensione massima della coda
    private Object[] Q; // la coda
    private int head; // puntatore al primo elemento in coda
    private int tail; // puntatore all'ultimo elemento della coda

    // post: costruisce una coda vuota
    public QueueArray() {...}

    // post: ritorna il numero di elementi nella coda
    public int size() {...}

    // post: ritorna true sse la coda e' vuota
    public boolean isEmpty() {...}

    // post: ritorna true sse la coda e' piena
    public boolean isFull() {...}

    // post: svuota la coda
    public void clear() {...}

    // pre: coda non vuota
    // post: ritorna il valore del primo elemento della coda
    public Object front() {...}

    // pre: value non nullo
    // post: inserisce value in coda
    public void enqueue(Object ob) {...}

    // pre: coda non vuota
    // post: ritorna e rimuove l'elemento il primo elemento in coda
    public Object dequeue() {...}
}

***** classe BTreeNode *****
package BinTrees;
class BTreeNode {
    Object key; // valore associato al nodo
    BTreeNode parent; // padre del nodo
    BTreeNode left; // figlio sinistro del nodo
    BTreeNode right; // figlio destro del nodo

    // post: ritorna un albero di un solo nodo, con valore value e sottoalberi
    // sinistro e destro vuoti
    BTreeNode(Object ob) {
        key = ob;
        parent = left = right = null;
    }

    // post: ritorna un albero contenente value e i sottoalberi specificati
    BTreeNode(Object ob,
        BTreeNode left,
        BTreeNode right,
        BTreeNode parent) {

```

```
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
    ...
}
***** classe BinaryTree *****
package BinTrees;
import Queues.*;
public class BinaryTree implements BT {
    private BTNode root;        // la radice dell'albero
    private BTNode cursor;     // puntatore al nodo corrente
    private int count;         // numero nodi dell'albero

    // post: crea un albero binario vuoto
    public BinaryTree() {
        root = null;
        cursor = null;
        count = 0;
    }
    ...
}
```