

Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello del 14 Giugno 2005

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
 - (a) $\lg n = \Omega(n^2)$
 - (b) $n^2 = \Omega(\lg n)$
 - (c) $n^2 = \Omega(n \lg n)$
 - (d) Nessuna delle precedenti risposte è esatta.
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 2 sottoproblemi di dimensione $(n/2)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è quadratica. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Esercizio 2 (ASD)

Qual è la complessità dell'algoritmo di inserimento di una chiave in una coda con priorità di n elementi realizzata con un max-heap binario? Dire quale delle seguenti risposte è esatta. Giustificare la risposta.

- (a) $O(n)$ nel caso peggiore
- (b) $O(\log n)$ nel caso peggiore
- (c) $O(n \log n)$ nel caso peggiore
- (d) $O(\log n)$ nel caso medio ed $O(n)$ nel caso peggiore

Esercizio 3 (ASD)

Disegnare un albero R/B che contiene le chiavi: 6,23,13,11,16,2,4,17,35,3,7

Esercizio 4 (ASD)

Si consideri il seguente algoritmo scritto nello pseudo codice:

test

```
i ← 1
while (i < n) and (T[i+1] ≤ T[parent(i+1)])
    do i ← i+1
return (i=n)
```

e si dimostri, utilizzando la tecnica dell'invariante, che l'algoritmo restituisce **true** se e solo se l'array T contiene un max-heap.

Esercizio 5 (ASD e Laboratorio)

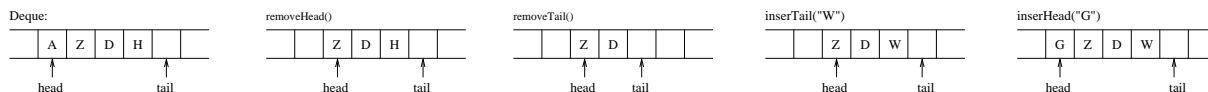
1. **(ASD)** Scrivere lo pseudo codice di un algoritmo che calcola il numero di figli unici in un albero generale rappresentato utilizzando gli attributi: key, child, sibling.
2. **(LABORATORIO)** Si consideri il package *Trees* visto durante il corso e relativo agli alberi generali. Si vuole aggiungere alla classe *GenTree* il seguente metodo, che ritorna il numero di nodi dell'albero che hanno più di un figlio.

```
// post: ritorna il numero di nodi dell'albero che hanno piu' di un figlio
public int contaNodiConPiuDiUnFiglio() {...}
```

Si richiede di completare l'implementazione del metodo usando la ricorsione.

Esercizio 6 (Laboratorio)

La struttura dati *Deque* è simile ad una coda in cui però è possibile inserire e rimuovere dati sia dalla testa (*head*) che dalla coda (*tail*). Nella figura che segue sono riportati alcuni esempi di rimozione e inserimento in una *Deque*.



Si vuole realizzare la struttura dati *Deque* utilizzando un array circolare. Si richiede quindi di contribuire all'implementazione della seguente classe scrivendo i metodi *Deque*, *isEmpty*, *Head*, *insertTail* e *removeHead*, gestendo l'array *D* in modo circolare.

```
public class Deque {
    private static final int MAX=100;    // dimensione massima della deque
    private Object[] D;                  // la deque
    private int head;                     // puntatore alla testa
    private int tail;                     // puntatore alla coda
    private int numel;                    // totale elementi nella deque

    // post: costruisce una deque vuota
    public Deque() {...}

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

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

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da head
    public Object Head() {...}

    // pre: deque non vuota
    // post: ritorna il valore dell'elemento puntato da tail
    public Object Tail() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte tail)
    public void insertTail(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da head
    public Object removeHead() {...}

    // pre: value non nullo
    // post: inserisce in deque (parte head)
    public void insertHead(Object ob) {...}

    // pre: deque non vuota
    // post: ritorna e rimuove l'elemento puntato da tail
    public Object removeTail() {...}
}
```

```

***** CLASSE TreeNode *****
package Trees;
class TreeNode {
    Object key;           // valore associato al nodo
    TreeNode parent;      // padre del nodo
    TreeNode child;       // figlio sinistro del nodo
    TreeNode sibling;      // fratello destro del nodo

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

    // post: ritorna un albero contenente value e i sottoalberi specificati
    TreeNode(Object ob, TreeNode parent, TreeNode child, TreeNode sibling) {
        key = ob;
        this.parent = parent;
        this.child = child;
        this.sibling = sibling;
    }
}

***** CLASSE GenTree *****
package Trees;
import Utility.*;
public class GenTree implements Tree{
    private TreeNode root;    // radice dell'albero
    private int count;        // numero di nodi dell'albero
    private TreeNode cursor;  // riferimento al nodo corrente

    // post: costruisce un albero vuoto
    public GenTree() {
        root = cursor = null;
        count = 0;
    }
    ...
    ...
}

```