Algoritmi e Strutture Dati & Laboratorio di Algoritmi e Programmazione

Appello dell' 8 Febbraio 2005

Esercizio 1 (ASD)

1. Dire quale delle seguenti affermazioni è vera giustificando la risposta.
   (a) $n \lg n = O(\lg n)$
   (b) $\lg n = \Omega(n)$
   (c) $n \lg n = \Omega(n)$
   (d) Nessuna delle precedenti è vera
2. Un algoritmo di tipo divide et impera per risolvere un problema di dimensione n lo decompone in 4 sottoproblemi di dimensione $(n/2)$ ciascuno, e ricombina le loro soluzioni con un procedimento la cui complessità asintotica è caratterizzata dalla funzione $f(n) = 2n^3 + 3n$. Qual è la complessità asintotica dell' algoritmo? Giustificare la risposta.

Soluzione

1. Risposta esatta: (c) - Si può applicare la definizione della classe $\Omega(n)$: $n \leq n \lg n$ per ogni $n \geq 2$.
   Si può anche calcolare il limite: $\lim_{n\to\infty} \frac{n \lg n}{n} = \infty$ per concludere che $n \lg n = \omega(n)$ e quindi $n \lg n = \Omega(n)$.
2. Si tratta di risolvere la ricorrenza:
   $T(n) = 4T(\frac{n}{2}) + 2n^3 + 3n$. E' facile vedere che la ricorrenza può essere trattata con il Master Theorem. Si ha: $a = 4$, $b = 2$ e $\log_b a = \log_2 4 = 2$ E' quindi facile verificare che $f(n) = \Omega(n^{\log_b a+1}) = \Omega(n^3)$, ad esempio mostrando che $n^3 \leq 3n^3 + n$ per ogni $n \geq 1$. Poichè vale anche (condizione di regolarità): $af(\frac{n}{b}) = 4[2(\frac{n}{2})^3 + 3(\frac{n}{2})] = n^3 + 6n \leq \frac{2}{3}(2n^3 + 3n) = \frac{4}{3}n^3 + 2n$, per ogni $n \geq 4$, si ricade nel caso 3 del Master Theorem. La soluzione è quindi: $T(n) = \Theta(n^3)$.

Esercizio 2 (ASD)

Dire quale delle seguenti affermazioni è esatta relativamente a ciascuna delle seguenti domande.
1) Quanti scambi vengono effettuati dall'algoritmo HeapSort?
2) Quanti confronti vengono effettuati dall'algoritmo QuickSort?

(a) $O(\lg n)$ nel caso medio

(b) $O(n^3)$ nel caso peggiore

(c) $O(n^2)$ nel caso peggiore

(d) $O(n \lg n)$ nel caso peggiore

Soluzione

1) Risposta esatta: (d)
2) Risposta esatta: (c)

Esercizio 3 (ASD)

Dire quali delle seguenti affermazioni sono applicabili al seguente albero, giustificando la risposta.

```
             17
           /     \
          /       \
         14        27
        / \        /
       5  16     20
      /  /       / \
     2  15      18 21
```

(a) L'albero è un max-heap
(b) L'albero è un min-heap
(c) L'albero è un albero binario di ricerca che puo' essere colorato in modo da divenire R/N
(d) L'albero è un albero binario di ricerca non bilanciato

Soluzione

La sola affermazione applicabile è la (d) in quanto è facile vedere che è soddisfatta la condizione BST ma che l'abero non può essere colorato poichè sul cammino 17,27,20,18,NIL non può esserci lo stesso numero di nodi neri del cammino 17,27,NIL senza violare altri punti della proprietà RB. Inoltre è facile verificare che l'albero non può rappresentare né un max-heap (es. $17 < 27$) né un min-heap (es. $17 > 14$).

Esercizio 4 (ASD e Laboratorio)

Si consideri la classe *BTNode* per memorizzare i nodi di un albero binario (riportata alla fine del testo d'esame). Si consideri inoltre la seguente classe *QuasiCompleto* che costruisce alberi quasi completi (ovvero alberi completi in cui mancano zero o più foglie "più a destra" dell'ultimo livello). Gli alberi vengono costruiti per livelli e in modo che siano sempre quasi completi. Quindi ogni nuovo elemento viene inserito dopo la foglia più a destra dell'ultimo livello.

```
public class QuasiCompleto  {
    BTNode root;      // radice dell'albero
    int count;           // numero di elementi dell'albero

    // post: crea un albero vuoto
    public QuasiCompleto() { root = null; count = 0; }
    ...
    ...
    // pre: albero non vuoto e quasi completo
    // post: ritorna il riferimento all'ultima foglia inserita nell'albero
    private BTNode ultimaFoglia() {...}
}
```

Si richiede di implementare il metodo *ultimaFoglia* che ritorna il riferimento all'ultima foglia inserita nell'albero. In particolare, si richiede di:

1. (**ASD e Laboratorio**) scrivere lo pseudocodice di un algoritmo iterativo che risolva il problema
2. (**ASD e Laboratorio**)provare la correttezza dell'algoritmo
3. (**Laboratorio**) scrivere l'implementazione Java del metodo
4. (**EXTRA**) Scrivere lo pseudocodice di un algoritmo che risolva il problema in O(log(n)) (dove n è il numero di nodi dell'albero), sfruttando l'informazione relativa al numero di elementi presenti nell'albero quasi completo e le relazioni padre-figlio tra i nodi.

Soluzione

1. Una soluzione consiste nell'effettuare la visita in ampiezza dell'albero salvando il riferimento di ciascun nodo. L'ultimo nodo visitato sarà proprio la foglia cercata. Un'altra soluzione è la seguente:

ultimaFoglia()
pos ← root
while left[pos] ≠ nil do
if height(right[pos]) = height(left[pos])
then pos ← right[pos]
else pos ← left[pos]
return pos

2. Dobbiamo dimostrare che l'algoritmo ritorna correttamente il riferimento all'ultima foglia inserita nell'albero. L'invariante del ciclo è il seguente:

   INV = l'ultima foglia inserita nel sottoalbero radicato in pos è l'ultima foglia dell'albero radicato in root.

   Infatti:

   Inizializzazione; all'inizio pos punta all'albero radicato in root e quindi l'invariante è vero.

   Mantenimento; supponiamo che l'invariante sia vero per pos fissato ovvero che l'ultima foglia del sottoalbero radicato in pos sia anche l'ultima foglia dell'albero radicato in root. Se il sottoalbero sinistro e destro di pos hanno la stessa altezza allora la foglia cercata è l'ultima foglia del sottoalbero destro di pos; altrimenti è l'ultima foglia del sottoalbero sinistro di pos. L'assegnamento a pos ristabilisce l'invariante per il ciclo successivo.

   Terminazione: il ciclo termina quando pos punta ad una foglia. L'invariante assicura che essa è l'ultima foglia dell'albero radicato in root.

3.

```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode ultimaFoglia() {
    BTNode pos = root;
    while (pos.left != null) {
        if (BNode.height(pos.right) ==
            BNode.height(pos.left))
            pos = pos.right;
        else
            pos = pos.left;
    }
    return pos;
}
```

4. **ultimaFoglia()**
pos ← root
k ← count
i ← 1
while k >1 do
A[i] ← k mod 2
k ← k div 2
i ← i+1
for j ← i downto 1
if A[j] = 0
pos ← pos.left
else
pos ← pos.right
return pos

   Soluzione Java:

```
// pre: albero non vuoto e quasi completo
// post: ritorna il riferimento all'ultima foglia inserita nell'albero
private BTNode ultimaFoglia() {
    BTNode pos = root;

    // la codifica binaria del numero di elementi nell'albero quasi completo
    // serve da guida per trovare il percorso che arriva all'ultima foglia inserita
    String s = Integer.toBinaryString(count);
    int k = s.indexOf("1") + 1;
```

```
        while (k < s.length()) {

            if (s.charAt(k) == '0')
                pos = pos.left;
            else
                pos = pos.right;
            k++;
        }
        return pos;
    }
```

Esercizio 5 (ASD e Laboratorio)

1. (**ASD**) Relativamente alle visite di alberi generali rappresentati come un insieme di nodi a cui sono associati gli attributi `child` e `sibling`, scegliere tra le seguenti una affermazione corretta e giustificare la risposta riportando lo pseudocodice dell'algoritmo corrispondente.
   (a) La visita in ampiezza utilizza una struttura dati coda
   (b) La visita in profondità utilizza una struttura dati pila
   (c) La visita in profondità utilizza una struttura dati coda
   (d) La visita in ampiezza utilizza una struttura dati pila

2. (**Laboratorio**) Scrivere l'implementazione Java della struttura dati scelta (solo le operazioni di inserimento e rimozione).

Soluzione

1. Ci sono diverse soluzioni possibili dato che la visita in profondità iterativa utilizza una struttura dati pila mentre la visita in ampiezza utilizza una struttura dati coda. Supponendo di avere inizializzato le strutture S (pila) e Q (coda) e di effettuare una chiamata esterna con x uguale alla radice dell'albero, i due algoritmi sono:

```
visita-iter-DFS(x)
    push(x,S)
    while (not empty-stack(S))
       do x <- top(S)
          pop(S)
          if not(empty-tree(x))
             then "visita x"
                  push(sibling[x],S)
                  push(child[x],S)

visita-BFS(x)
    enqueue(x,Q)
    while (not empty-queue(Q))
       do x <- head(Q)
          dequeue(Q)
          while not(empty-tree(x))
             do "visita x"
                 if not(empty-tree(child[x]))
                     then enqueue(child[x],Q)
                 x <- sibling[x]
```

2. Si vedano ad esempio le classi StackArray.java e QueueArray.java realizzate durante il corso.

Esercizio 6 (Laboratorio)

Sia s una stringa contenente solamente parentesi tonde aperte e chiuse. Diciamo che s è ben formata se rispetta le regole sulle parentesi, ovvero che 1) ogni parentesi aperta deve essere seguita (anche non immediatamente) da una

parentesi chiusa e che 2) ogni parentesi chiusa deve corrispondere ad una parentesi precedentemente aperta. Diciamo invece che s è mal formata se non rispetta queste regole.
Ad esempio le stringhe (()(()())) e ()(()) sono ben formate, mentre le stringhe)(e (()())) sono mal formate.
Data una lista semplice L di tipo *SLList* in cui ciascun record contiene un carattere (precisamente un oggetto di tipo *Character*) con valore '(' oppure ')', si richiede di implementare il metodo *MalFormata* della seguente classe:

```
import BasicLists.*;
import Utility.*;
public class Esercizio6 {

   // pre: L diverso da null
   // post: ritorna true se la stringa di parentesi memorizzate in L e' mal formata;
   //       ritorna false altrimenti
   public boolean MalFormata(SLList L) {...}
}
```

Soluzione

```
import BasicLists.*;
import Utility.*;
public class prova {

    public boolean MalFormata(SLList L) {
        Iterator iter = L.iterator();
        int k = 0;
        Character parentesi = new Character('(');
        while (iter.hasNext() && k >= 0) {
            if (((Character)iter.next()).equals(parentesi))
                k++;
            else
                k--;
        }
        return (k!=0);
    }
}
```

```
*********************** CLASSE BTNode ***********************************************************
class BTNode {
    Object key;       // valore associato al nodo
    BTNode parent;    // padre del nodo
    BTNode left;      // figlio sinistro del nodo
    BTNode right;     // figlio destro del nodo

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

     // post: ritorna l'altezza del nodo n rispetto all'albero
    //       in cui si trova
    static int height(BTNode n) {...}

    ...
}
*********************** CLASSE SLRecord *********************************************************
package BasicLists;
class SLRecord {
    Object key;             // valore memorizzato nell'elemento
    SLRecord next;         // riferimento al prossimo elemento

    // post: costruisce un nuovo elemento con valore v, e prossimo elemento nextel
    SLRecord(Object ob, SLRecord nextel) { key = ob; next= nextel; }

    // post: costruisce un nuovo elemento con valore v, e niente next
    SLRecord(Object ob) { this(ob,null); }
}
*********************** CLASSE SLList ***********************************************************
package BasicLists;
import Utility.Iterator;
public class SLList {
    SLRecord head;         // primo elemento
    int count;                  // num. elementi nella lista

    // post: crea una lista vuota
    public SLList() { head = null; count = 0; }

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

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty() {...}

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

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista
    //       Ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean insert(Object ob) {...}

    // pre: l'oggetto passato non e' nullo
    // post: ritorna true sse nella lista c'e' un elemento uguale a value
    public boolean contains(Object value) {...}

    // pre: l'oggetto passato non e' nullo
    // post: rimuove l'elemento uguale a value
    //       ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean remove(Object value) {...}

    // post: ritorna un oggetto che scorre gli elementi della lista
    public Iterator iterator() { return new SLListIterator(head);}
 }
*********************** CLASSE SLListIterator *****************************************************v**
package BasicLists;
import Utility.Iterator;
public class SLListIterator implements Iterator {
    SLRecord first;              // riferimento al primo elemento
    SLRecord current;            // riferimento all'elemento corrente

    // post: inizializza first e current secondo il parametro passato
    public SLListIterator(SLRecord el) {first = current = el;}

    // post: reset dell'iteratore per ricominciare la visita
    public void reset() {...}

    // post: ritorna true se la lista non e' terminata
    public boolean hasNext() {...}

    // pre: lista non terminata
    // post: ritorna il valore dell'elemento corrente e sposta l'iteratore all'elemento successivo
    public Object next() {...}

    // post: ritorna il valore dell'elemento corrente
    public Object val() {...}
}
```