Algoritmi e Strutture Dati
&
Laboratorio di Algoritmi e Programmazione

Appello del 24 Gennaio 2007

Esercizio 1 (ASD)

1. Sia $T(n) = T(n/6) + T(n/3) + \Theta(n)$. Considerare ciascuna delle seguenti affermazioni e dire se è corretta o no. Giustificare la risposta.

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

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

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

Soluzione

(a) $T(n) = \Omega(n)$ è vera. Usando il metodo di sostituzione, possiamo dimostrare che esistono due costanti positive n_0, d tali che $T(n) \geq dn$, per ogni $n > n_0$.

$$\begin{aligned} T(n) &= T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + \Theta(n) \\ &\geq T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + cn && \text{per definizione di } \Theta(n), \text{ con } c > 0 \\ &\geq \tfrac{1}{6}dn + \tfrac{1}{3}dn + cn && \text{per ipotesi induttiva} \\ &\geq \tfrac{1}{2}dn + cn \\ &\geq dn && \text{se } d \leq 2c \end{aligned}$$

(b) $T(n) = O(\lg n)$ è falsa. Poiché $n = \omega(\lg n)$, per il punto precedente e le proprieta' sulle classi si ottiene $T(n) = \omega(\lg n)$. Poichè per ogni $g(n)$, $\omega(g(n)) \cap O(g(n)) = \emptyset$, si deduce $T(n) \neq O(\lg n)$.

(c) $T(n) = O(n \lg n)$. E' vera. Usando il metodo di sostituzione, è facile dimostrare che esistono due costanti positive n_0, d tali che $T(n) \leq dn \lg n$, per ogni $n > n_0$.

$$\begin{aligned} T(n) &= T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + \Theta(n) \\ &\leq T(\tfrac{1}{6}n) + T(\tfrac{1}{3}n) + cn && \text{per definizione di } \Theta(n), \text{ con } c > 0 \\ &\leq d(\tfrac{1}{6}n \lg \tfrac{1}{6}n) + d(\tfrac{1}{3}n \lg \tfrac{1}{3}n) + cn && \text{per ipotesi induttiva} \\ &\leq \tfrac{1}{2}dn \lg n - \lg 6n - \lg 3n + cn && \text{per ipotesi induttiva} \\ &\leq \tfrac{1}{2}dn \lg n + cn \lg n \\ &\leq dn \lg n && \text{se } d \geq 2c \end{aligned}$$

Esercizio 2 (ASD)

Si assuma di disporre dei seguenti costruttori di alberi binari:
- tree_NIL() che restituisce una foglia_NIL,
- tree(x, T1,T2) che data una chiave x e due alberi binari T1 e T2 restituisce un nuovo albero binario con radice x, sottoalbero sinistro T1 e sottoalbero destro T2.

Dato un max-heap memorizzato in un array A[1..n], si sviluppi un algoritmo efficiente per costruire un BST T che contiene tutte le chiavi di A. Si dica quale è la complessità dell'algoritmo e si dimostri la sua correttezza.

Nota: Si osservi che non si richiede che T sia bilanciato.

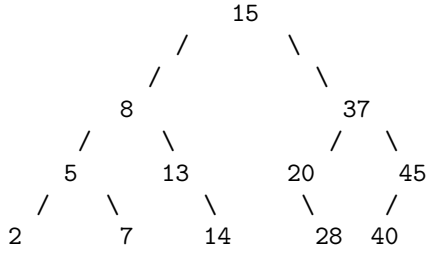
Soluzione

```
trasforma(A,n)
  if n>1
    then
        root <- A[1]
        A[1] <- A[n]
        heapify(A,1,n-1)
        T1 <- trasforma(A,n-1)
        T2 <- tree_NIL()
        return tree(root,T1,T2)
    else return tree_NIL()
```

Si tratta di una diversa formulazione dell'algoritmo heapsort; la complessità si valuta in modo analogo ed è $\Theta(n \lg n)$. Per la correttezza possiamo dimostrare che per ogni $m \geq 0$ se A[1..m] è un max-heap allora `trasforma`(A,m) costruisce un albero BST che è in realta' una catena a sinistra ordinata in ordine decrescente dalla radice alla foglia più a sinistra. La correttezza è ovvia se m=0. Per $m = n > 0$, osserviamo che prima della chiamata ricorsiva A[1..(n-1)] contiene un max-heap e pertanto, per ipotesi induttiva, `trasforma`(A,n-1) costruisce un BST T1 che è una catena a sinistra e contiene tutte le chiavi di A[1..n-1]. Osservando poi che root è l'elemento più grande di A[1..n] otteniamo il risultato.

Esercizio 3 (ASD)

1. In quanto tempo è possibile trovare una chiave in un albero R/B di n elementi? Giustificare la risposta.
2. Dato il seguente albero BST

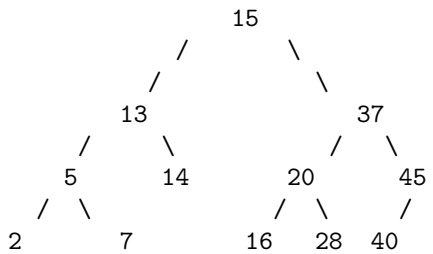


si consideri l'inserimento della chiave 16 seguito dalla cancellazione della chiave 8 e si disegni l'albero risultante.

Soluzione

1. La risposta corretta è $O(\lg n)$, dato che un albero R/B è un BST bilanciato.

2. L'albero finale è:



Esercizio 4 (ASD)

Sia T un albero generale i cui nodi hanno chiavi intere e gli attributi: chiave, figlio, fratello. Scrivere un algoritmo che trasforma T raddoppiando i valori di tutte le chiavi sui livelli dispari dell'albero.

Soluzione

Si tratta di realizzare una visita dell'albero che tiene conto del livello del nodo considerato. La chiamata esterna sarà `trasforma(root[T],false)`, assumendo che la radice si trovi sul livello 0 (pari).

```
trasforma(x,raddoppia)
    if x =/= NIL
       then
        if raddoppia then chiave[x] <- 2*chiave[x]
        trasforma(fratello[x],raddoppia)
        trasforma(figlio[x], not(raddoppia))
```

Per gli esercizi seguenti siano date le seguenti interfacce.

```
public interface BinTree  {

      Node root();               //  la radice del BinTree

      Node parent(Node p);       // padre di p nel BinTree

      Node left(Node p);         // figlio sinistro di p nel BinTree

      Node right(Node p);        // figlio destro di p nel BinTree

      boolean internal(Node p);  // true sse p e' un nodo interno nel BinTree

      boolean leaf(Node p);      // true sse p e' una foglia nel BinTree

}

public interface Node {

      Comparable key()           // la chiave memorizzata nel nodo

}
```

Esercizio 5 (LAB)

Definite, in Java, l'implementazione del metodo `enumerate()` descritto dalla seguente specifica. L'implementazione deve garantire una complessità asintotica pari a $\Theta(m + h)$, dove m è il numero di chiavi enumerate e h è l'altezza dell'albero.

```
/**
 * POST: restituisce una enumerazione di tutte le chiavi k tali che a <= k <= b nel
 *       sottoalbero di T radicato nel nodo p, dove <= e' la relazione di ordine tale
 *       che a <= b sse a.compareTo(b) <= 0
 *
 * PRE:  T != null e' un BinTree con nodi di tipo Node, in cui ogni nodo ha zero o
 *       due figli. Le chiavi sono memorizzate nei soli nodi interni di T, e sono
 *       organizzate in modo da soddisfare la BST property.
 *       Il nodo p e' un nodo di T, e a <= b.
 */
public static Iterator enumerate(BinTree T, Node p, Comparable a, Comparable b)
```

Soluzione

```
public static Iterator enumerate(BinTree T, Node p, Comparable a, Comparable b) {
if (T.leaf(p))  return new Vector().iterator();

if (((Comparable)p.key()).compareTo(a) < 0)
    return enumerate(T, T.right(p),a,b);
else if (((Comparable)p.key()).compareTo(b) <= 0) {
        Vector els = new Vector();
    Iterator itleft = enumerate(T, T.left(p), a, b);
    Iterator itright = enumerate(T, T.right(p), a, b);
    els.add(p.element());
    while (itleft.hasNext()) els.add(itleft.next());
    while (itright.hasNext()) els.add(itright.next());
    return els.iterator();
}
else
    return enumerate(T, T.left(p),a,b);
}
```

Esercizio 6 (LAB)

Definite, in Java, l'implementazione del metodo `find()` descritto dalla seguente specifica. L'implementazione deve garantire una complessità asintotica pari a $O(\mathtt{n})$.

```
/**
 * POST: restituisce un riferimento al nodo raggiunto all'n-esimo passo di una
 *       visita per livelli di T, in cui i livelli sono attraversati da sinistra
 *       a destra. La visita attraversa i soli nodi interni di T e causa l'eccezione
 *       se T ha meno di n nodi interni.
 *
 *
 * PRE: T != null e' un BinTree con nodi di tipo Node, in cui ogni nodo ha zero o
 *      due figli; n >= 0
 */
 public static Node find(BinTree T, int n) throws NoSuchElementException
```

Soluzione

```
public static Node find (BinTree T, int n) {
Vector Q = new Vector();
Node p = null; int m = 0;

if (T.internal(T.root())) {
      Q.add(T.root());  m = n;
}
while (!Q.isEmpty() && m > 0) {
    p = (Node)Q.elementAt(0);
    Q.removeElementAt(0); m--;

    if (T.internal(T.left(p))) Q.add(T.left(p));
    if (T.internal(T.right(p))) Q.add(T.right(p));
}
if (m > 0) throw new NoSuchElementException();
else return p;
}
```