lezione 7

Alberi binari

Correzione prima esercitazione: metodo distinct

```
// post: ritorna il numero di elementi distinti del multi insieme
public int distinct( ) {
   int d = 0;
   int i = 0;
   int j;

   // INV1: d e' il numero di elementi distinti in M[0..i-1]
   while (i < count) {
      // INV2: gli elementi di M[0...j-1] sono diversi da M[i]
      for (j = 0; j<i && !M[i].equals(M[j]); j++);
      if (j == i) // non trovato!
         d++;
      i++;
   }

   return d;
}
```

Correttezza metodo distinct

INV2: Gli elementi di M[0..j-1] sono diversi da M[i].
Inizializzazione: j=0. La porzione di array è vuota. Vero.
Mantenimento: sia vero per j fissato. Allora gli elementi di M[0..j-1] sono diversi da M[i]. Se M[j] ≠ M[i] allora j viene incrementato, e possiamo concludere che tutti gli elementi di M[0…j] sono diversi da M[i], ovvero l'invariante viene mantenuto per il ciclo successivo.
Se M[j] = M[i] allora il ciclo termina senza incremento di j. Vale comunque che gli elementi di M[0…j-1] sono diversi da M[i].
Terminazione: il ciclo termina per due motivi:
- i=j e quindi l'invariante garantisce che tutti gli elementi di M[0…i-1] sono diversi da M[i]
- M[j] = M[i] e l'invariante garantisce comunque che gli elementi M[0…j-1] sono diversi da M[i]

Correttezza metodo distinct

INV1: d è il numero di elementi distinti in M[0…i-1]
Inizializzazione: i=0. La porzione di array è vuota. Vero.
Mantenimento: sia vero per i fissato. Allora d è il numero di elementi distinti di M[0..i-1]. Il corpo del ciclo considera M[i] e lo confronta con tutti gli elementi precedenti. L'invariante del for assicura che:
-se i=j tutti gli elementi di M[0…i-1] sono diversi da M[i]. In tal caso è corretto incrementare il numero d di elementi distinti
-se i≠j l'elemento M[i] è presente in M[0…i-1] e il metodo non fa nulla.
In entrambi i casi sfruttando l'ipotesi induttiva, posso dire che d è il numero di elementi distinti di M[0…i], ovvero l'invariante viene mantenuto all'incrementare di i.
Terminazione: il ciclo termina quando i=count. Allora l'invariante garantisce che d è il numero di elementi distinti in M[0…count-1], cioè di tutto il multi-insieme.

Correzione seconda esercitazione: toString

```
// pre: Q non nulla
// post: ritorna una stringa che contiene tutti gli elementi in Q
//       separati da uno spazio. L'ordine degli elementi nella stringa
//       deve essere corrispondere all'ordinamento FIFO della coda.
//       Il metodo non deve alterare lo stato originale di Q
public static String toString(QueueArray Q) {
   Object ob;
   String s = "";
   int i = Q.size();
   while (i > 0) {
      ob = Q.dequeue();
      s = s + ob.toString() + " ";
      Q.enqueue(ob);
      i--;
   }
   return s;
}
```

Correttezza metodo toString

Sia N il numero di elementi in coda.
INV: Per ogni elemento x in coda sia k la sua posizione originaria ($0 \le k \le N-1$) in Q. Allora la posizione corrente di x è

$$(k+i) \bmod N$$

Inoltre s contiene i primi N-i elementi della coda originaria.

Inizializzazione: i=N. Allora la nuova posizione di x è $(k+N) \bmod N = k$ e s contiene i primi 0 elementi di Q. Vero.
Mantenimento: sia vero per i fissato. Allora x è in posizione $h = (k+i) \bmod N$ e s contiene i primi N-i elementi di Q.
Il corpo del ciclo elimina il primo elemento di Q, lo concatena ad s e poi lo riaccoda. Quindi s contiene i primi $N-i+1=N-(i-1)$ elementi di Q e tutti gli elementi in coda sono traslati di una posizione modulo N. Per l'elemento generico x la nuova posizione è $(h-1) \bmod N = (k + (i - 1)) \bmod N$
Quindi l'invariante viene mantenuto al decrementare di i.
Terminazione: i= 0. Allora k' = $(k+0) \bmod N = k$ e s contiene i primi N-0 elementi di Q.

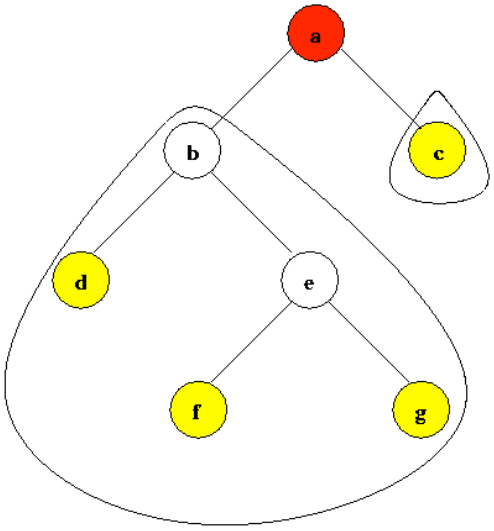
Alberi Binari: definizione ricorsiva

Gli alberi binari possono essere definiti ricorsivamente come segue:
Un *albero binario* è una struttura definita su un insieme di *nodi* che:
- non contiene nessun nodo (*albero vuoto*), oppure
- contiene un nodo *radice*, un albero binario detto *sottoalbero sinistro* ed un albero binario detto *sottoalbero destro*.

Molti algoritmi su alberi binari possono essere descritti in modo naturale sfruttando questa definizione ricorsiva, comprendendo un caso base (per l'albero vuoto) e una clausola ricorsiva (con due chiamate ricorsive, per i due sottoalberi).

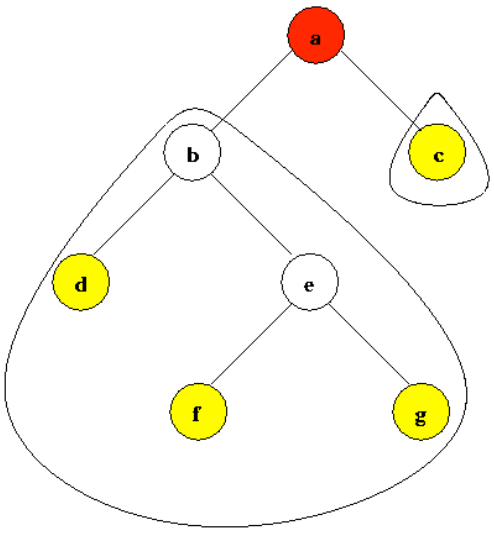
Alberi Binari: terminologia

Alcuni termini che già conoscete:
nodo radice
nodo foglia
nodo interno
nodo padre
nodo figlio (sinistro, destro)
sottoalbero (sinistro, destro)



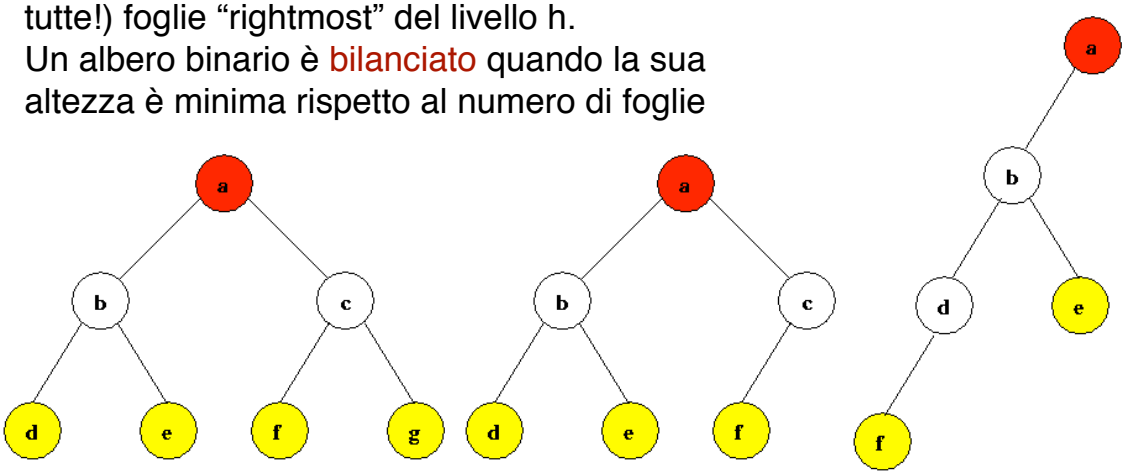
Alberi Binari: terminologia

Cammino: sequenza di nodi n1, n2, ... tali che, per ogni i, ni è padre di ni+1
Lunghezza di un cammino: numero dei nodi che formano il cammino - 1
Altezza di un nodo: lunghezza del cammino più lungo da quel nodo ad una foglia
Altezza di un albero: altezza della sua radice
Profondità di un nodo: lunghezza del cammino dalla radice a quel nodo
Livello di un nodo: definito ricorsivamente:
- il livello della radice è 0
- il livello di un qualsiasi altro nodo è il livello di suo padre +1



Alberi Binari: terminologia

Un albero binario completo di altezza h ha tutte le foglie del livello h.
Un albero binario quasi completo di altezza h si ottiene da un albero completo di altezza h rimuovendo 0 o più (ma non tutte!) foglie "rightmost" del livello h.
Un albero binario è bilanciato quando la sua altezza è minima rispetto al numero di foglie

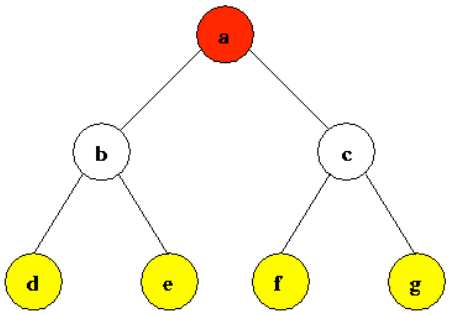


Alberi Binari: algoritmi di visita

Sia n un nodo di un albero binario A. Vediamo l'algoritmo di visita in pre-ordine del sottoalbero radicato in n:

```
preorder(n)
  if n ≠ NIL
    then "visita key[n]"
         preorder(left[n])
         preorder(right[n])
```

Allora preorder(root) esegue la seguente visita:
a b d e c f g

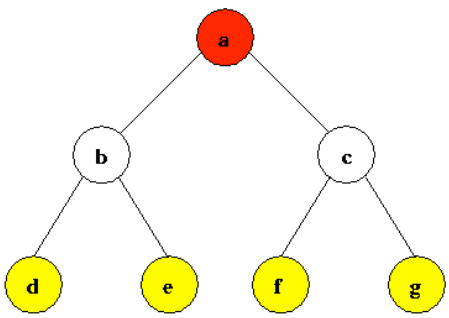


Alberi Binari: algoritmi di visita

Sia n un nodo di un albero binario A. Vediamo l'algoritmo di visita in post-ordine del sottoalbero radicato in n:

```
postorder(n)
  if n ≠ NIL
    then postorder(left[n])
         postorder(right[n])
         "visita key[n]"
```

Allora postorder(root) esegue la seguente visita:
d e b f g c a

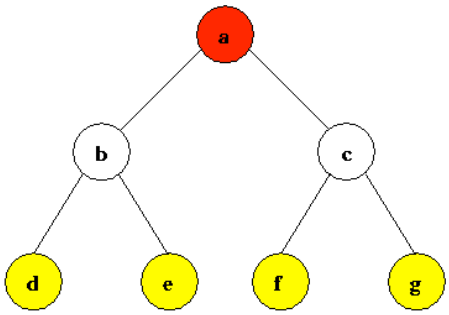


Alberi Binari: algoritmi di visita

Sia n un nodo di un albero binario A. Vediamo l'algoritmo di visita in-ordine del sottoalbero radicato in n:

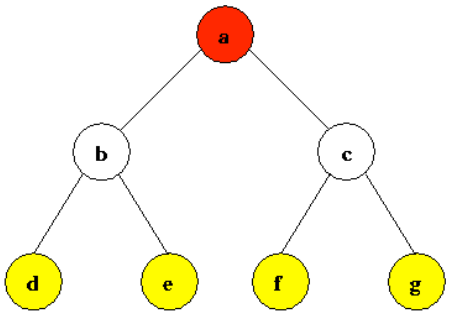
```
inorder(n)
  if n ≠ NIL
    then inorder(left[n])
         "visita key[n]"
         inorder(right[n])
```

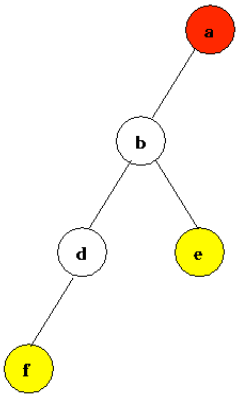
Allora inorder(root) esegue la seguente visita:
d b e a f c g



Alberi binari: rappresentazione lineare

È possibile rappresentare un albero in modo lineare, ad esempio con una stringa?





(a (b (d) (e)) (c (f) (g))) (a (b (d (f) ()) (e)) ())

Le parentesi indicano quindi la struttura dell'albero.
Questa notazione può essere utilizzata per fare input/output di un albero da/su una stringa

Tipo di dato Albero Binario

Data la definizione di albero binario, le operazioni definite per il tipo di dato sono:

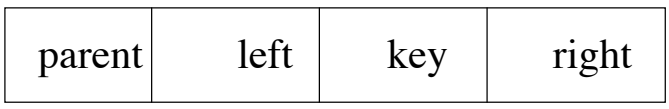
-insert(T,k) inserisce un nuovo nodo in T con chiave k
-remove(T,k) rimuove la prima occorrenza di k in T
-contains(T,k) ritorna true sse k è presente in T

Sono inoltre definite le operazioni di visita dell'albero:

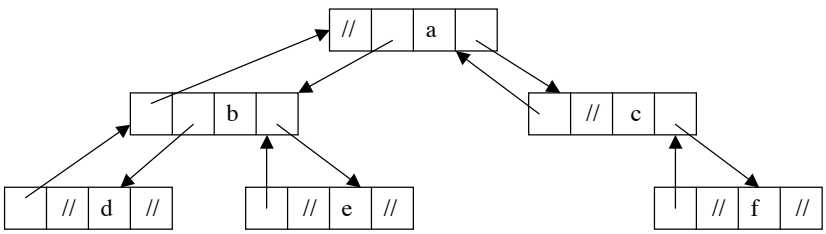
-preorder(T) esegue una visita in pre-ordine di T
-inorder(T) esegue una visita in-ordine di T
-postorder(T) esegue una visita in post-ordine di T

Alberi Binari: rappresentazione

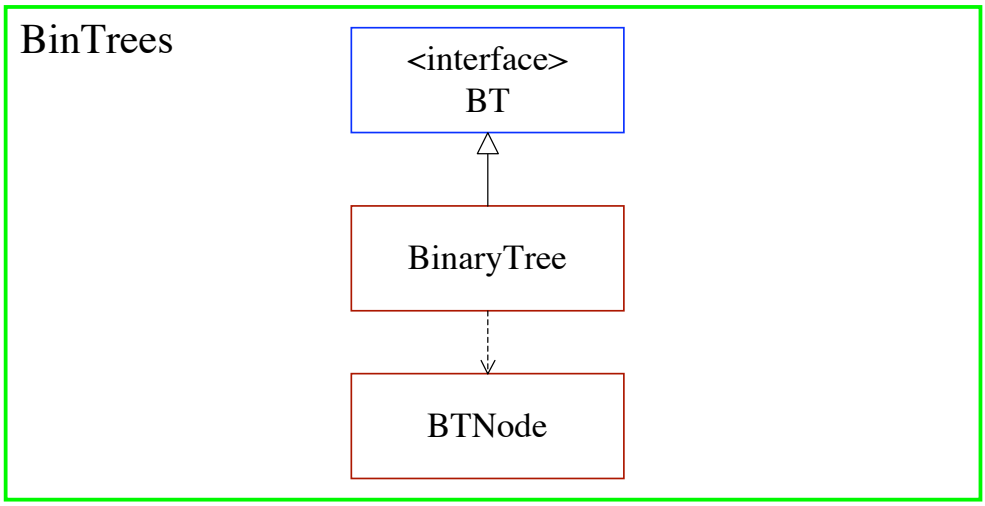
Utilizziamo una rappresentazione degli alberi binari come struttura collegata. Ciascun record ha la seguente struttura:



Un albero viene quindi rappresentato così:



Struttura del package BinTrees



Alberi binari: assunzioni

Dichiariamo un cursore che possa muoversi all'interno dell'albero. Servono quindi operazioni di spostamento del cursore:
-reset (sposta il cursore sulla radice)
-moveUp (sposta il cursore sul padre del nodo corrente)
-moveLeft (sposta il cursore sul suo figlio sinistro)
-moveRight (sposta il cursore sul suo figlio destro)
Lo spostamento non viene effettuato se il cursore esce dai limiti dell'albero (es: moveUp con cursore posizionato sulla radice).

L'inserimento di un nuovo elemento avviene sempre a livello foglia. Se il cursore non si trova già posizionato su una foglia allora l'inserimento non viene eseguito.
La cancellazione avviene sempre dalle foglie: se il cursore non è posizionato su una foglia allora la rimozione non viene eseguita.

Alberi binari: interfaccia (1)

```
package BinTrees;
public interface BT  {

    // post: ritorna il numero di elementi dell'albero
    public int size();

    // post: ritorna true sse l'albero e' vuoto
    public boolean isEmpty();

    // post: svuota l'albero
    public void clear();

    // post: l'elemento corrente diventa la root
    public void reset();

    // pre:  albero non vuoto
    // post: se possibile, sosta l'elemento corrente sul figlio sinistro e
    //       ritorna true. Ritorna false altrimenti.
    public boolean moveLeft();

    // pre:  albero non vuoto
    // post: se possibile, sosta l'elemento corrente sul figlio destro e
    //       ritorna true. Ritorna false altrimenti.
    public boolean moveRight();

    // pre:  albero non vuoto
    // post: se possibile, sosta l'elemento corrente sul padre e ritorna true.
    //       Ritorna false altrimenti.
    public boolean moveUp();
```

Alberi binari: interfaccia (2)

```
    // pre: albero non vuoto
    // post: ritorna il valore del nodo puntato da cursor
    public Object getValue();

    // pre:  ob non nullo
    // post: se l'albero e' vuoto, inserisce ob come root.
    //       se cursor non ha figlio sinistro, inserisce ob come figlio
    //       sinistro di cursor e ritorna true. Altrimenti, non
    //       inserisce ob e ritorna false.
    public boolean insertLeft(Object ob);

    // pre:  ob non nullo
    // post: se l'albero e' vuoto, inserisce ob come root.
    //       se cursor non ha figlo destro, inserisce ob come figlio destro di
    //       cursor e ritorna true. Altrimenti, non inserisce ob e ritorna
    //       false.
    public boolean insertRight(Object ob);

    // pre:  albero non vuoto
    // post: se l'elemento corrente e' una foglia, la rimuove e ne ritorna
    //       il valore. Ritorna null altrimenti
    public Object remove();
```

Alberi binari: interfaccia (3)

```
    // post: ritorna una stringa che contiene gli elementi dell'albero,
    //       visitati in pre-ordine
    public String preorderVisit();

    // post: ritorna una stringa che contiene gli elementi dell'albero,
    //       visitati in post-ordine
    public String postorderVisit();

    // post: ritorna una stringa che contiene gli elementi dell'albero,
    //       visitati in-ordine
    public String inorderVisit();

}
```

Alberi binari: record (1)

```
package BinTrees;
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 un albero contenente value e i sottoalberi
    // specificati
    BTNode(Object ob, BTNode left, BTNode right, BTNode parent) {
        key = ob;
        this.parent = parent;
        setLeft(left);
        setRight(right);
    }
```

Alberi binari: record (2)

```
    // post: imposta il sottoalbero sinistro a newLeft
    //       e sistema il valore di parent
     void setLeft(BTNode newLeft) {
        if (left != null && (left.parent == this))
            left.parent = null;
        left = newLeft;
        if (left != null)
            left.parent = this;
    }

    // post: imposta il sottoalbero destro a newRight
    //       e sistema il valore di parent
     void setRight(BTNode newRight) {
        if (right != null && (right.parent == this))
            right.parent = null;
        right = newRight;
        if (right != null)
            right.parent = this;
    }
```

Alberi binari: record (3)

```
// post: ritorna true se "this" e' figlio sinistro di parent.
    boolean isLeftChild() {
        if (parent == null)
            return false;
        else
           return this == parent.left;
    }
    // post: ritorna true se "this" e' figlio destro di parent.
    boolean isRightChild() {
        if (parent == null)
           return false;
        else
           return this == parent.right;
    }
}
```

BinaryTree (1)

```
package BinTrees;
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() {clear();}

    // post: ritorna il numero di nodi nell'albero
    public int size() {return count;}

    // post: ritorna true sse l'albero e' vuoto
    public boolean isEmpty() {return count == 0;}

    // post: rimuove tutti i nodi dall'albero
    public void clear() {
        root = null;
        cursor = null;
        count = 0;
    }
```

BinaryTree (2)

```
    // post: sposta il cursore sulla root
    public void reset() { cursor = root; }

    // pre: albero non vuoto
    // post: se possibile, cursor si sposta sul figlio sinistro e
    //       ritorna true. Ritorna false altrimenti.
    public boolean moveLeft() {
        if (cursor.left != null) {
           cursor = cursor.left;
           return true;
        }
        else
           return false;
    }
```

BinaryTree (3)

```
// pre: albero non vuoto
// post: se possibile, cursor si sposta sul figlio sinistro e
//       ritorna true. Ritorna false altrimenti.
public boolean moveRight() {
    if (cursor.right != null) {
       cursor = cursor.right;
       return true;
    }
    else
       return false;
}
// pre:  albero non vuoto
// post: se possibile, cursor si sposta al nodo padre e
// ritorna true. Ritorna false altrimenti.
public boolean moveUp() {
    if (cursor != root) {
        cursor = cursor.parent;
        return true;
    }
    else
        return false;
}
```

BinaryTree (4)

```
// pre:  albero non vuoto
// post: ritorna il valore del nodo puntato da cursor
public Object getValue() { return cursor.key; }

// pre:  ob non nullo
// post: se l'albero e' vuoto, inserisce ob come root.
//       se cursor e' una foglia, inserisce ob come figlio sinistro di
//      cursor e ritorna true. Altrimenti, non inserisce ob e ritorna false.
public boolean insertLeft(Object ob) {

    // inserimento non valido
    if (cursor !=null && cursor.left != null)
       return false;

    // primo caso: albero vuoto
    if (cursor == null)
        cursor = root = new BTNode(ob);
    else
        // secondo caso: cursor e' posizionato su una foglia
        cursor.setLeft(new BTNode(ob, null, null, cursor));
    count++;
    return true;
}
```

BinaryTree (5)

```
// pre:  ob non nullo
// post: se l'albero e' vuoto, inserisce ob come root.
//       se cursor e' una foglia, inserisce ob come figlio destro di
//      cursor e ritorna true. Altrimenti, non inserisce ob e ritorna false.
public boolean insertRight(Object ob) {

    // inserimento non valido
    if (cursor !=null && cursor.right != null))
       return false;

    // primo caso: albero vuoto
    if (cursor == null)
        cursor = root = new BTNode(ob);
    else
        // secondo caso: cursor e' posizionato su una foglia
        cursor.setRight(new BTNode(ob, null, null, cursor));
    count++;
    return true;
}
```

BinaryTree (6)

```
// pre:  albero non vuoto
// post: se cursor e' posizionato su una foglia, rimuove il nodo puntato
//       da cursor, sposta cursor sul nodo padre (se esiste) e ritorna il
//       valore dell'elemento rimosso. Altrimenti ritorna null.
public Object remove() {
    // se cursor non e' posizionato su una foglia ritorno null
    if (cursor.left != null || cursor.right != null)
       return null;

    Object value = cursor.key;
    if (cursor.isLeftChild()) {
        moveUp();
        cursor.setLeft(null);
    }
    else if (cursor.isRightChild()) {
        moveUp();
        cursor.setRight(null);
    } else {
        root = cursor = null;
    }
    count--;
    return value;    }
```

BinaryTree (7)

```
// post: ritorna una stringa che contiene gli elementi dell'albero, visitati
//       in pre-ordine
public String preorderVisit() {
    StringBuffer pre = new StringBuffer("Visita pre-order: ");
    if (root != null)
       formatPreorder(root, pre);
    return pre.toString();
}
// pre:  parametri diversi da null
// post: appende a sb una stringa che rappresenta il sottoalbero con
//       radice n visitato in pre-order
private void formatPreorder(BTNode n, StringBuffer sb) {
    if (n.left == null && n.right == null)
         sb.append(" ("+n.key+")");
    else {
        sb.append(" ("+n.key);
        if (n.left != null)
            formatPreorder(n.left,sb);
        else
            sb.append(" ()");
        if (n.right != null)
            formatPreorder(n.right,sb);
        else sb.append(" ()");
        sb.append(")");
    }
}
```

BinaryTree (8)

```
// post: ritorna una stringa che contiene gli elementi dell'albero, visitati
//       in post-ordine
public String postorderVisit() {
    StringBuffer post = new StringBuffer("Visita post-order: ");
    if (root != null)
       formatPostorder(root, post);
    return post.toString();
}
// pre:  parametri diversi da null
// post: appende a sb una stringa che rappresenta il sottoalbero con
//       radice n visitato in post-order
private void formatPostorder(BTNode n, StringBuffer sb) {
    if (n.left == null && n.right == null)
         sb.append(" ("+n.key+")");
    else {
        sb.append("( ");
        if (n.left != null)
            formatPostorder(n.left,sb);
        else sb.append(” ()");
        if (n.right != null)
            formatPostorder(n.right,sb);
        else sb.append(" ()");
        sb.append(n.key + ")");
    }
}
```

BinaryTree (9)

```
    // post: ritorna una stringa che contiene gli elementi dell'albero, visitati
    //       in ordine simmetrico
    public String inorderVisit() {
        StringBuffer in = new StringBuffer("Visita in-order: ");
        if (root != null)
           formatInorder(root, in);
        return in.toString();
    }
    // pre:  parametri diversi da null
    // post: appende a sb una stringa che rappresenta il sottoalbero con
    //       radice n visitato in ordine simmetrico
    private void formatIorder(BTNode n, StringBuffer sb) {
        if (n.left == null && n.right == null)
             sb.append(" ("+n.key+")");
        else {
            sb.append("( ");
            if (n.left != null)
                formatInorder(n.left,sb);
            else sb.append(" ()");
            sb.append(n.key);
            if (n.right != null)
                formatInorder(n.right,sb);
            else sb.append(" ()");
            sb.append(")");
        }
    }
}   /*fine classe BinaryTree.java*/
```