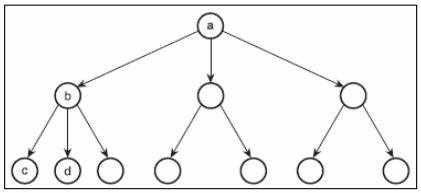
SOMMARIO

- ALBERI: introduzione
- ALBERI BINARI: introduzione
- VISITE RICORSIVE DI ALBERI
  - Dimensione e Altezza
- ALBERI BINARI DI RICERCA (BST)
  - Introduzione
  - Ricerca, inserimento e cancellazione
  - Implementazione

ALBERI : introduzione

- Gli alberi rappresentano una generalizzazione delle liste. Mentre ogni elemento delle liste ha al più un successore, ogni elemento degli alberi può avere più di un successore.
- Gli alberi sono solitamente utilizzati per rappresentare partizioni ricorsive di insiemi e strutture gerarchiche.
- Un esempio di struttura gerarchica è rappresentato dagli alberi genealogici.

ALBERI



TERMINOLOGIA:

`a` è il nodo *root* (radice)
`b`, `c` ,`d` sono nodi *figli*
`b` è nodo *padre* (di `c` e `d`)
`c` e `d` sono *foglie* (nodi che non hanno figli)
`c` e `d` sono *fratelli* (nodi sullo stesso livello)

livello: altezza di un nodo nell'albero (`a` è a livello 0, `b` a livello 1, `c` e `d` a livello 2)

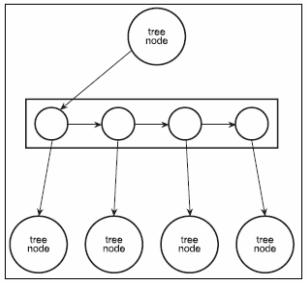
altezza: è la massima profondità a cui può trovarsi una foglia.

ALBERI

- In un albero esiste un unico percorso dal nodo root a un figlio qualsiasi, quindi ogni nodo (eccetto il nodo *root*) ha un solo genitore.
- L'implementazione degli alberi può essere basata sia su *array,* sia su *nodi.*
- La rappresentazione basata su array può risultare particolarmente inefficiente se gli alberi non sono pieni e densi.
- Inoltre la rappresentazione basata su nodi è più intuitiva e permette di supportare modifiche alla struttura in modo più efficace.

ALBERI

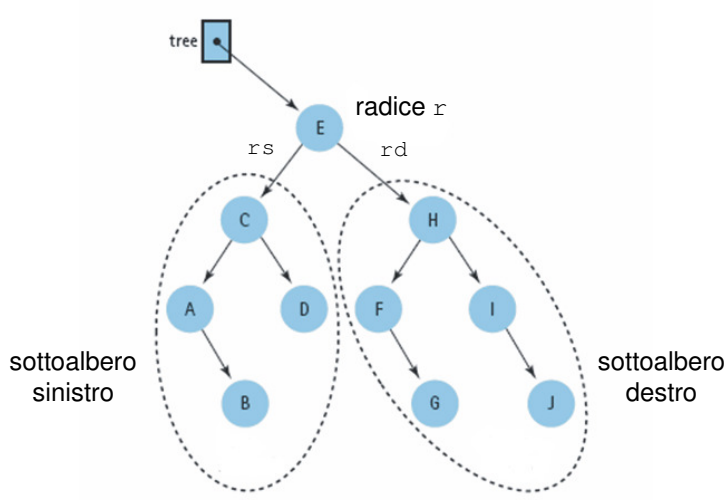
- A differenza delle linked-list in un albero generico ogni nodo può avere il riferimento a più nodi. Nell'implementazione più generica ogni nodo avrà il riferimento ad una lista linkata di nodi (i suoi figli).



ALBERI BINARI

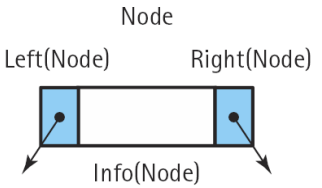
- Ogni nodo può avere al massimo due figli (figlio sinistro e figlio destro).
- Un albero vuoto è un albero binario che non contiene nessuna chiave e che viene indicato con `null`.
- Un albero binario contenente `n` elementi è costituito dalla radice `r`, i rimanenti `n-1` elementi sono divisi in due gruppi disgiunti, *ricorsivamente* organizzati in due sottoalberi disgiunti, con a capo rispettivamente i figli sinistro (`rs`) e destro (`rd`) della radice `r`.

ALBERI BINARI



ALBERI BINARI: Nodi

- Nella rappresentazione basata sui nodi ogni nodo di un albero binario avrà due riferimenti a ciascuno dei figli (Left e Right). In alcune implementazioni si può avere anche il riferimento al nodo padre.



ALBERI BINARI : Nodi

```
class BinaryTreeNode{

        private BinaryTreeNode left;
        private BinaryTreeNode right;
        private Object data;

        public BinaryTreeNode(){
            left = null;
            right = null;
        }

        public BinaryTreeNode(object d){
            left = null;
            right = null;
            data = d;
        }
    ...
```


ALBERI BINARI: Nodi

```
...

        public BinaryTreeNode Left{
             get { return left; }
             set { left = value; }
         }

         public BinaryTreeNode Right{
             get { return right; }
             set { right = value; }
         }


         public Object Data{
             get { return data; }
             set { data = value; }
         }
    }
```


ALBERI BINARI: implementazione

```
class BinaryTree{

        private BinaryTreeNode root;

        public BinaryTree()
        {
            root = new BinaryTreeNode(null);
        }

        public BinaryTree(object d)
        {
            root = new BinaryTreeNode(d);
        }
       ...
}
```


Visite di alberi

- Molti algoritmi basati sugli alberi richiedono di attraversare un albero visitandone tutti i nodi.
- Diversamente dalle collezioni di oggetti rappresentati in modo lineare (es. liste) è necessario seguire tutti i possibili rami a partire dalla radice.
- I procedimenti iterativi per le visite degli alberi possono essere complicati, in quanto è ncessario mantenere istante per istante l'elenco dei nodi che rappresentano i punti di ramificazione rimasti in sospeso e da cui la visita deve proseguire.

Visite di alberi: algoritmi ricorsivi

- Gli alberi binari permettono di progettare algoritmi ricorsivi seguendo una metodologia generale di risoluzione.
- L'idea alla base di questi algoritmi è che ogni nodo può essere considerato come il nodo root del corrispondente sottoalbero.
- Il problema della visita di un albero può quindi essere formulato in termini ricorsivi.

Visite di alberi: algoritmi ricorsivi

CASO BASE: albero vuoto.

PASSO INDUTTIVO: l'albero non è vuoto, la ricorsione avviene sui due figli (sottoalberi).

RICOMBINAZIONE: le grandezze calcolate dal passo induttivo vengono ricombinate.

- Ogni nodo viene attraversato un numero costante di volte, per cui se il caso base e la ricombinazione richiedono un tempo costante, la complessità di questi algoritmi è O(n).

Visite di alberi: dimensione

- Consideriamo ad esempio la funzione `Dimension()` che restituisce il numero di elementi in un albero, ovvero la sua **dimensione** n.
- La dimensione di un albero può essere definita ricorsivamente in quanto:

  - un albero vuoto ha dimensione 0

  - la dimensione di un albero non vuoto è pari alla dimensione dei suoi sottoalberi incrementata di 1 per includere la radice

Visite di alberi: dimensione

```
public int Dimension(){
       return Dimension(root);
}

private int Dimension(BinaryTreeNode node){
       if (node == null)
          return 0;
       int c = 1;
       if (node.Left != null)
          c += Dimension(node.Left);
       if (node.Right != null)
          c += Dimension(node.Right);
       return c;
}
```

Visite di alberi: algoritmi ricorsivi

- La classe `BinaryTree` contiene un riferimento al nodo `root` dell'albero.
- Le implementazioni ricorsive devono effettuare la ricorsione sui nodi.
- Il metodo pubblico `Dimension()` viene chiamato sull'oggetto di tipo `BinaryTree`.
- Esso richiama un metodo privato `Dimension(BinaryTreeNode node)` che ha un parametro in ingresso di tipo `BinaryTreeNode` e permette di effettuare la ricorsione sulla struttura interna dell'albero.

Visite di alberi: altezza

- Un altro parametro importante negli alberi è l'**altezza**: la massima profondità raggiunta dalle sue foglie.
- L'altezza di una foglia è 0, l'albero vuoto ha altezza -1.
- Calcolare esplicitamente la profondità di ogni foglia, prendendone poi la massima per trovare l'altezza è inefficiente. Anche per risolvere questo problema si può utilizzare un algoritmo ricorsivo.

Visite di alberi: altezza

```
public int Altezza(){
      return Altezza(root);
 }

private int Altezza(BinaryTreeNode node){
      int altezzaSX, altezzaDX;
      if(node == null)
         return -1;
      else{
         altezzaSX= Altezza(node.Left);
         altezzaDX=Altezza(node.Right);
         return System.Math.Max(altezzaSX,altezzaDX) +1;
      }
}
```


ALBERI BINARI DI RICERCA

- Le operazioni tipiche che possono essere svolte sulle collezioni di dati sono `search`, `insert` e `delete`.
- Come si è visto precedentemente in un insieme statico e ordinato si può effettuare l'algoritmo di ricerca binaria che ha una complessità O(log n), ma inserimento e cancellazione hanno complessità O(n) se vogliamo che l'insieme rimanga ordinato.

ALBERI BINARI DI RICERCA

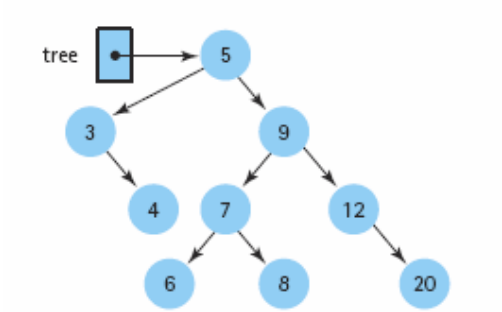
- Usando un array non ordinato o una lista gli inserimenti hanno complessità O(1) mentre ricerche e cancellazioni hanno complessità O(n).
- Gli Alberi di Ricerca Binari (BST) permettono di ottenere una collezione di oggetti in cui, sotto opportune ipotesi, tutte e tre le operazioni hanno complessità logaritmica.

ALBERI BINARI DI RICERCA

Un BST soddisfa le seguenti proprietà:

- ogni nodo *v* contiene un elemento cui è associata una chiave presa da un dominio totalmente ordinato.
- le chiavi nel sottoalbero sinistro di *v* sono minori della chiave di *v*
- le chiavi nel sottoalbero destro di *v* sono maggiori della chiave di *v*

ALBERI BINARI DI RICERCA



ALBERI BINARI DI RICERCA: interfaccia

Un BST è un ADT, pertanto possiamo descriverlo mediante la sua interfaccia

```
interface IBinarySearchTree{

        bool IsEmpty();
        int Dimension();
        bool Find(object item);
        void Insert(object item);
        void Delete(object item);
        void Print();
        void MakeEmpty();
}
```

ALBERI BINARI DI RICERCA: implementazione

- Analogamente al caso dell'albero binario `BinaryTree` l'albero di ricerca binario `BinarySearchTree` avrà come dato membro il riferimento alla radice dell'albero.
- Inoltre implementerà l'interfaccia che abbiamo definito.

```
class BinarySearchTree : IBinarySearchTree
{
        private BinaryTreeNode root;
        ...
}
```


ALBERI BINARI DI RICERCA: ricerca

- La ricerca di un elemento all'interno di un BST avviene in maniera analoga a quanto già visto per l'algoritmo di ricerca binaria.
- Si parte dalla radice dell'albero e si confronta il suo contenuto con la chiave da ricercare (`key`).
- Se la il contenuto è diverso e `key` è minore proseguiamo la ricerca nel sottoalbero di sinistra, altrimenti proseguiamo la ricerca nel sottoalbero di destra.

ALBERI BINARI DI RICERCA: ricerca

- Poiché dopo ogni confronto, se non abbiamo trovato l'elemento, scendiamo di un livello nell'albero, il tempo richiesto dalla ricerca è O(h), dove h è l'altezza dell'albero.
- Se l'albero è molto profondo e sbilanciato tenderà ad essere simile ad una linked list, pertanto h=n e si ha O(n).
- Se l'albero è stato costruito in maniera bilanciata si ha h=log(n) e complessità O(log n).

ALBERI BINARI DI RICERCA: ricerca, implementazione

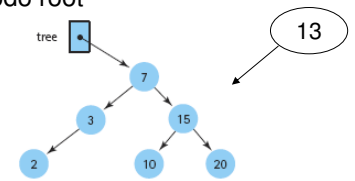
```
public bool Find(object item){
            return Find(root, item);
}

private bool Find(BinaryTreeNode node, object item){
       if(node==null)
            return false;
       else if(((((IComparable)item).CompareTo(node.Data))<0)){
            return Find(node.Left,item);
       }
       else if(((((IComparable)item).CompareTo(node.Data))>0)){
            return Find(node.Right,item);
       }
       else{
            return true;
       }
}
```

ALBERI BINARI DI RICERCA: inserimento

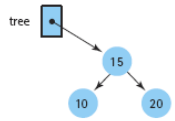
- E' necessario inserire nella maniera corretta un oggetto all'interno di un BST.
- Ogni oggetto viene inserito come una nuova foglia all'interno del BST.
- Per sviluppare il metodo `Insert` si utilizza un approccio ricorsivo: ogni nodo è a sua volta il nodo root di un sottoalbero binario.

ALBERI BINARI DI RICERCA: inserimento

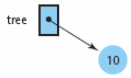
(1) si confronta il contenuto del nodo da inserire con il contenuto del nodo root



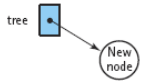
(2) il procedimento è ricorsivo

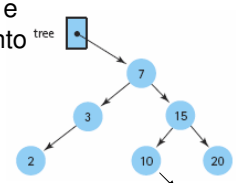


(3) quando si incontra un nodo senza figli si ferma la ricorsione



(4) si crea il nuovo nodo e si assegna il riferimento





ALBERI BINARI DI RICERCA: inserimento, implementazione

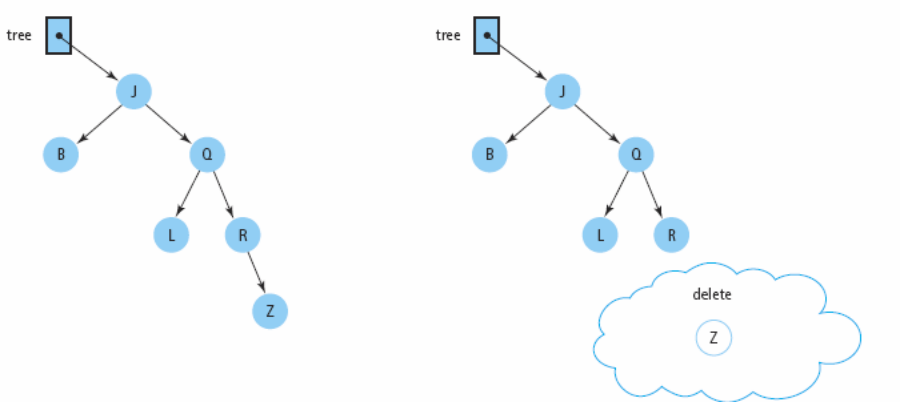
```
public void Insert(object data){
        root = Insert(root, data);
}

private BinaryTreeNode Insert(BinaryTreeNode node, object data){
        if (node == null){
          node = new BinaryTreeNode(data);
        }
        else{
          if ((((IComparable)data).CompareTo(node.Data)) < 0){
              node.Left = Insert(node.Left, data);
          }
          else{
              node.Right = Insert(node.Right, data);
          }
        }
        return node;
}
```


ALBERI BINARI DI RICERCA: rimozione

- Innanzitutto bisogna ricercare nell'albero il nodo da rimuovere (analogamente a quanto visto con il metodo `Find`)
- Il metodo di rimozione deve tenere conto di che tipo di nodo vogliamo cancellare.
- Inoltre l'albero deve mantenere le sue proprietà di BST.
- La cancellazione di un nodo *foglia* è semplice: bisogna settare a `null` il corrispondente riferimento (left o right) del nodo padre.

ALBERI BINARI DI RICERCA: rimozione

RIMOZIONE DI UN NODO FOGLIA

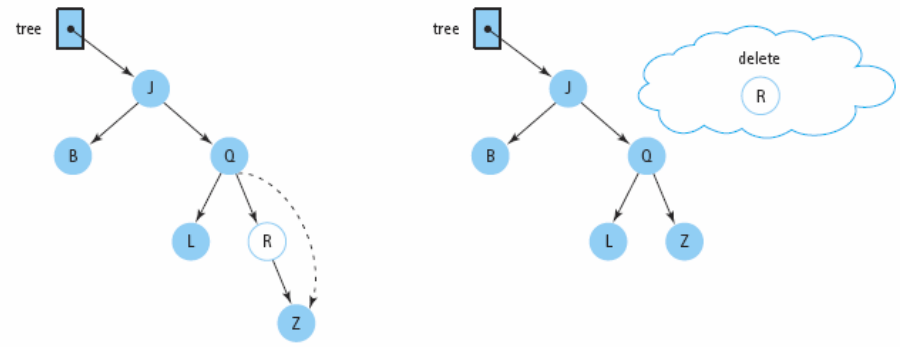


ALBERI BINARI DI RICERCA: rimozione

- Se vogliamo cancellare un nodo con dei figli bisogna prestare attenzione a non perdere il riferimento al sottoalbero relativo.
- Nel caso di un nodo con un solo figlio il riferimento del padre deve essere cambiato in modo tale da non puntare più al nodo che vogliamo cancellare, ma al suo nodo figlio.

ALBERI BINARI DI RICERCA: rimozione

RIMOZIONE DI UN NODO CON UN SOLO FIGLIO

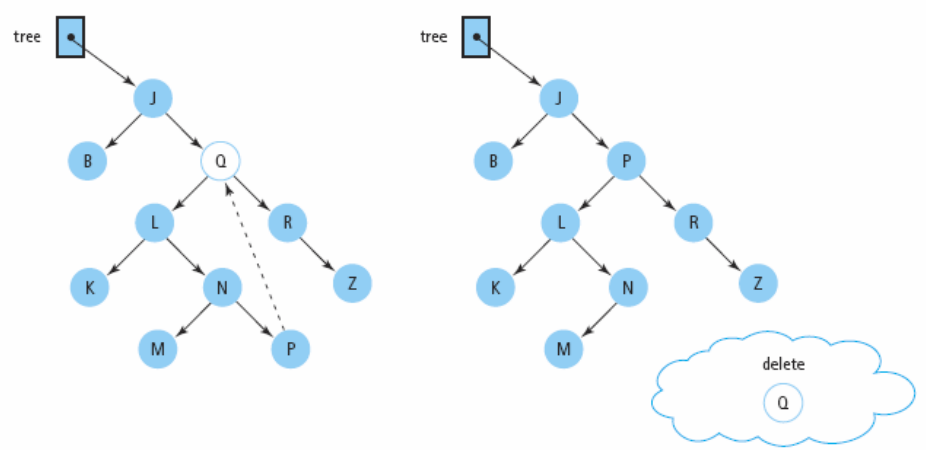


ALBERI BINARI DI RICERCA: rimozione

- La rimozione di un nodo con due figli è il caso più complicato: il riferimento del nodo padre non può puntare a entrambi i figli del nodo da cancellare.
- Esistono varie strategie, una è quella di non cancellare il nodo ma sostituire la sua informazione con quella di un altro nodo dell'albero, per poi cancellare quest'ultimo nodo.
- Per sostituire l'informazione del nodo, si usa il suo predecessore o successore, ovvero il nodo che *logicamente* precede o segue quello da cancellare.

ALBERI BINARI DI RICERCA: rimozione

RIMOZIONE DI UN NODO CON DUE FIGLI



ALBERI BINARI DI RICERCA: rimozione, implementazione

```
public void Delete(object item){
       Delete(ref root, item);
}
private void Delete(ref BinaryTreeNode node, object item){
     if (node == null)
          Console.WriteLine("item non presente");
     else if ((((IComparable)item).CompareTo(node.Data)) < 0){
          BinaryTreeNode tmp = node.Left;
          Delete(ref tmp, item);
          node.Left = tmp;
     }
     else if ((((IComparable)item).CompareTo(node.Data)) > 0){
          BinaryTreeNode tmp = node.Right;
          Delete(ref tmp, item);
          node.Right = tmp;
     }
     else{
          DeleteNode(ref node);
     }
}
```


ALBERI BINARI DI RICERCA: rimozione, implementazione

```
void DeleteNode(ref BinaryTreeNode node){
     object data;
     if (node.Left == null){
          node = node.Right;
     }
     else if (node.Right == null){
          node = node.Left;
     }
     else{
          GetPredecessor(node.Left, out data);
          node.Data = data;
          BinaryTreeNode tmp = node.Left;
          Delete(ref tmp, data);
          node.Left = tmp;
     }
}
private void GetPredecessor(BinaryTreeNode node, out object data){
      while (node.Right != null)
            node = node.Right;
      data = node.Data;
}
```


ALBERI BINARI DI RICERCA: inserimento e rimozione

- Sia l'inserimento sia la cancellazione hanno costo O(h) in quanto in entrambi i casi bisogna effettuare una ricerca all'interno dell'albero.
- Si è visto che h=log n se l'albero è bilanciato.
- Pur partendo da un albero bilanciato a seguito di inserimenti e cancellazioni tale proprietà potrebbe perdersi.
- Esistono diversi approcci per mantenere il bilanciamento degli alberi di ricerca binari.

ALBERI BINARI DI RICERCA: stampa e attraversamento dell'albero.

- L'implementazione del metodo `Print()` richiede l'attraversamento dell'albero, in maniera analoga a quanto visto per il calcolo di dimensione e altezza degli alberi.
- Anche in questo caso il metodo è ricorsivo.
- Esistono varie strategie per attraversare un albero, a seconda che vengano analizzati prima il ramo sinistro e poi quello destro, o viceversa, prima i rami e poi la radice, o viceversa. I tre metodi sono chiamati: `inorder`, `postorder` e `preorder`.

ALBERI BINARI DI RICERCA: stampa e attraversamento dell'albero.

- Nell'attraversamento *inorder* il valore di un nodo è stampato tra la stampa dei valori del suo sottoalbero sinistro e la stampa dei valori del suo sottoalbero destro. L'attraversamento inorder in un BST produce una stampa dei valori in ordine crescente.
- E' il tipo di attraversamento implementato nel metodo `Print()` dell'interfaccia.

ALBERI BINARI DI RICERCA: stampa *inorder*

```
public void Print(){
      Print(root);
}

private void Print(BinaryTreeNode node){
      if (node != null){
           Print(node.Left);
           Console.Write((node.Data).ToString() + " ");
           Print(node.Right);
      }
}
```


ALBERI BINARI DI RICERCA: stampa *postorder*

- Nell'attraversamento *postorder* un nodo viene visitato dopo i suoi sottoalberi sinistro e destro.

```
public void PrintPostOrder(){
      PrintPostOrder(root);
}

private void PrintPostOrder(BinaryTreeNode node){
      if (node != null){
          PrintPostOrder(node.Left);
          PrintPostOrder(node.Right);
          Console.Write((node.Data).ToString() + " ");
      }
}
```

ALBERI BINARI DI RICERCA: stampa *preorder*

- Nell'attraversamento *preorder* un nodo viene visitato prima dei suoi sottoalberi sinistro e destro.

```
public void PrintPreOrder(){
       PrintPreOrder(root);
}

private void PrintPreOrder(BinaryTreeNode node){
       if (node != null){
          Console.Write((node.Data).ToString() + " ");
          PrintPreOrder(node.Left);
          PrintPreOrder(node.Right);
       }
}
```


ALBERI BINARI DI RICERCA: esempio

```
static void Test1(){
    BinarySearchTree myTree = new BinarySearchTree();
    myTree.Insert(17);
    myTree.Insert(15);
    myTree.Insert(21);
    myTree.Insert(13);
    myTree.Insert(16);
    myTree.Insert(5);
    myTree.Insert(10);
    myTree.Insert(20);
    myTree.Insert(18);
    myTree.Insert(19);
    myTree.Insert(27);
    myTree.Insert(25);
    myTree.Insert(30);
```


ALBERI BINARI DI RICERCA: esempio

```
Console.WriteLine("Elementi presenti:");
myTree.Print();
Console.WriteLine();
Console.WriteLine("-------------");
Console.WriteLine("Ricerca key 13:" + myTree.Find(13));
Console.WriteLine("Ricerca key 14:" + myTree.Find(14));
Console.WriteLine("-------------");
Console.WriteLine("Eliminazione 9");
myTree.Delete(9);
Console.WriteLine("Eliminazione 30");
myTree.Delete(30);
```


ALBERI BINARI DI RICERCA: esempio

```
  Console.WriteLine("-------------");
  Console.WriteLine("Stampa inorder");
  myTree.Print();
  Console.WriteLine();
  Console.WriteLine("Stampa postorder");
  myTree.PrintPostOrder();
  Console.WriteLine();
  Console.WriteLine("Stampa preorder");
  myTree.PrintPreOrder();
  Console.WriteLine();
  Console.WriteLine("-------------");
  Console.WriteLine("dimensione: " + myTree.Dimension());
  Console.WriteLine("altezza: " + myTree.Altezza());
  Console.WriteLine("-------------");
  myTree.MakeEmpty();
  Console.Write("Albero vuoto:");
  Console.WriteLine(myTree.IsEmpty());
}
```

ALBERI BINARI DI RICERCA: esempio

```
Elementi presenti:
5 10 13 15 16 17 18 19 20 21 25 27 30
-------------
Ricerca key 13:True
Ricerca key 14:False
-------------
Eliminazione 9
item non presente
Eliminazione 30
```

```
-------------
Stampa inorder
5 10 13 15 16 17 18 19 20 21 25 27
Stampa postorder
10 5 13 16 15 19 18 20 25 27 21 17
Stampa preorder
17 15 13 5 10 16 21 20 18 19 27 25
-------------
dimensione: 12
altezza: 4
-------------
Albero vuoto:True
```