# Esercitazione 6
# Alberi Binari di Ricerca

Corso di Fondamenti di Informatica II

BIAR2 (Ing. Informatica e Automatica) e BSIR2 (Ing. dei Sistemi)

A.A. 2013/2014

22 Novembre 2013

**Sommario**

Scopo di questa esercitazione è realizzare una struttura dati per gestire un albero binario di ricerca.

## 1 Rappresentazione di un Binary Search Tree

Un albero binario di ricerca (Binary Search Tree o BST) è un TDA che contiene un insieme di coppie $\langle chiave, valore \rangle$, che supporta sia inserimento di un elemento arbitrario sia ricerca di elementi per chiave, ed eventuale rimozione. L'insieme delle chiavi è totalmente ordinato. In questa esercitazione assumeremo per semplicità che le chiavi siano di tipo intero, mentre i valori saranno di un tipo generico `V`.

Un BST soddisfa la proprietà seguente: dati tre nodi $u$, $v$ e $w$ tali che $u$ è nel sottoalbero sinistro di $v$ e $w$ è nel sottoalbero destro di $v$, allora $\texttt{key}(u) \leq \texttt{key}(v) \leq \texttt{key}(w)$. L'attraversamento *in ordine* (un nodo è visitato dopo il suo sottoalbero sinistro e prima del suo sottoalbero destro), di conseguenza, visita le chiavi in ordine non decrescente. Utilizzeremo il BST come una mappa, ovvero non si hanno mai due valori distinti con la stessa chiave.

Per quanto riguarda l'albero binario, utilizzeremo una (variante della) rappresentazione basata su riferimenti (vedi Esercitazione 2, alberi $n$-ari). In questa rappresentazione, il generico nodo dell'albero, di tipo `Node<V>`, memorizza:

- la chiave (di tipo `int`) associata al nodo
- il valore (di tipo `V`) associato al nodo
- il riferimento (di tipo `Node<V>`) al figlio sinistro
- il riferimento (di tipo `Node<V>`) al figlio destro.

Scrivere una classe Java parametrica `BSTree<V>` con la seguente interfaccia.

## 1.1 Costruttori

```
public BSTree(int k, V value)
```

Costruisce un BST avente un unico nodo, costituito dalla chiave `k` e dal valore `value`.

```
private BSTree(Node<V> x)
```

Costruisce un BST avente un unico nodo, costituito da `x`.

## 1.2 Operazioni di ricerca e inserimento

```
public V find(int k)
```

Restituisce l'unico valore associato associato a `k`, se esiste, o `null` altrimenti. La ricerca non deve visitare tutto l'albero ma seguire lo schema della ricerca binaria. Il metodo deve avere costo $O(h)$, dove $h$ è l'altezza corrente del BST.

```
private V find(Node<V> x, int k)
```

Metodo ausiliario ricorsivo di `find`. Restituisce il valore associato a `k` nel sottoalbero con radice `x` (o `null` se non esiste).

```
public void insert(int k, V value)
```

Inserisce nel punto appropriato del BST la coppia $\langle$`k`, `value`$\rangle$; se esiste già un valore associato alla chiave `k`, il metodo sovrascrive il vecchio valore. Il metodo deve avere costo $O(h)$, dove $h$ è l'altezza corrente del BST.

```
private Node<V> insert(Node<V> x, int k, V value)
```

Metodo ausiliario ricorsivo di `insert`. Inserisce la coppia $\langle$`k`, `value`$\rangle$ nel punto appropriato del sottoalbero la cui radice è `x`. Se esiste già un valore associato alla chiave `k`, il metodo sovrascrive il vecchio valore. Restituisce il riferimento al "nuovo" `x`.

## 1.3 Operazioni di cancellazione

```
private Node<V> findMin(Node<V> x)
```

Metodo privato ricorsivo (propedeutico a `remove`). Restituisce il nodo a chiave minima nel sottoalbero la cui radice è `x`.

```
private Node<V> removeMin(Node<V> x)
```

Metodo privato ricorsivo (propedeutico a `remove`). Elimina il nodo a chiave minima dal sottoalbero la cui radice è `x`. Restituisce il nodo radice del sottoalbero modificato (o `null`, se `x` non ha figli).

```
public void remove(int k)
```

Elimina dal BST il nodo con chiave `k` (se esiste). Il metodo deve avere costo $O(h)$, dove $h$ è l'altezza corrente del BST.

```
private Node<V> remove(Node<V> x, int k)
```

Metodo ausiliario ricorsivo di `remove`. Elimina il nodo con chiave `k` (se esiste) dal sottoalbero la cui radice è `x`. Il valore di ritorno è usato per mantenere ricorsivamente la radice del sottoalbero che deve rimpiazzare il nodo da eliminare.

## 1.4 Test

Si proceda a testare `BSTree<V>` attraverso il driver `Test.java` fornito nella cartella dell'esercitazione.

## 1.5 La funzione `floor`

Un'operazione ausiliaria utile nel tipo di dati BST consiste nel trovare, data una chiave, il nodo che più si "avvicina" a tale chiave, senza superarla. Più precisamente, si vuole considerare la funzione `floor`, definita in questo modo: `floor(k)` è il nodo del BST avente la chiave *maggiore possibile* tra tutte quelle *minori o uguali* a `k`. La Figura 1 mostra due esempi (con chiavi di tipo carattere).

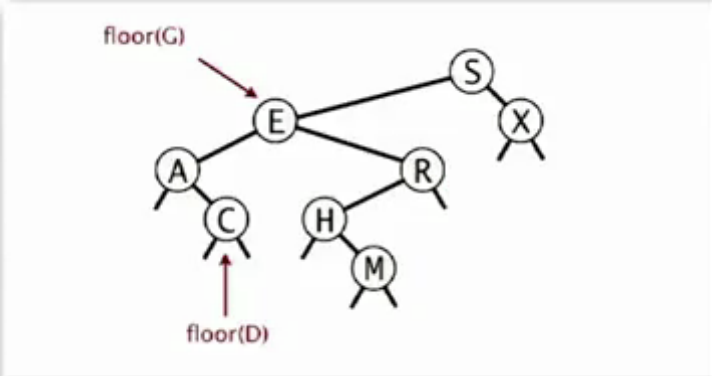


Figura 1: La funzione `floor`

Il calcolo di `floor(k)` in un BST può essere effettuato sfruttando le seguenti proprietà:

- Se `k` è uguale alla chiave nella radice, `floor(k)` è la radice.
- Se `k` è minore della chiave nella radice, `floor(k)` è nel sottoalbero sinistro.
- Se `k` è maggiore della chiave nella radice, `floor(k)` è nel sottoalbero destro *se* il sottoalbero destro contiene qualche chiave $\leq$ `k`; altrimenti `floor(k)` è la radice.

Si arricchisca la classe `BSTree<V>` dei seguenti metodi.

```
public Integer floor(int k)
```

Restituisce la chiave del nodo `floor(k)` (o `null` se il BST non contiene nessun nodo con chiave $\leq$ `k`). Il metodo deve avere costo $O(h)$, dove $h$ è l'altezza corrente del BST.

```
private Node<V> floor(Node<V> x, int k)
```

Metodo ausiliario ricorsivo di `floor`. Restituisce il nodo `floor(k)` calcolato nel sottoalbero la cui radice è `x` (o `null` se tale sottoalbero non contiene nessun nodo con chiave $\leq$ `k`).