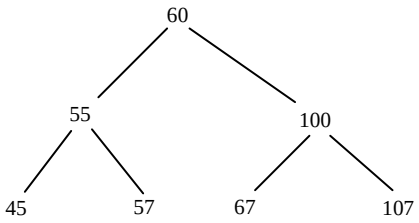
Binary Search Tree, Lecture 2

- For every node, X, in the tree, the values of all the items in its left subtree are smaller than the item in X, and the values of all the items in its right subtree are greater than the item in X.

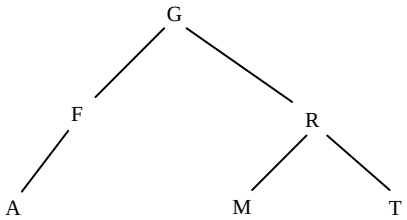
- A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure.

- Every node has at most two children

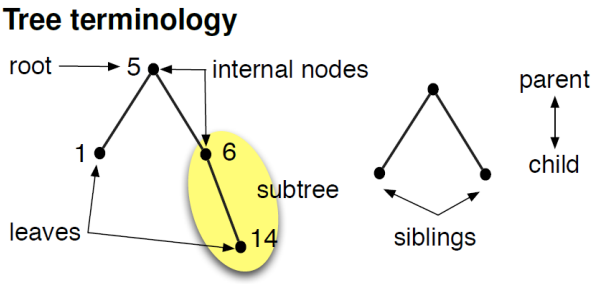
- It is either empty or consists of an element, called the *root*, and two distinct binary trees, called the *left subtree* and *right subtree*. Examples:



(A)



(B)

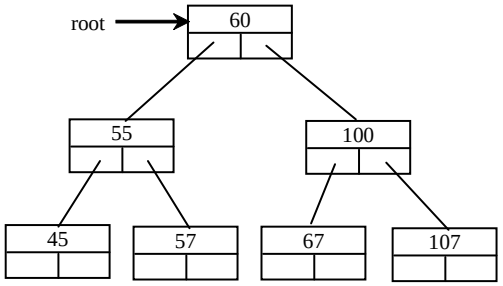


- A Binary tree can be implemented using either a Linked list or an array (An array is more appropriate for complete trees (See Heaps).)

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown below:

```
class TreeNode <E> {
      E element;
     TreeNode <E> left;
     TreeNode<E> right;

     public TreeNode(E e) {
       element = e;
     }
}
```



The variable *root* refers to the root node of the tree. If the tree is empty, *root* is *null*.

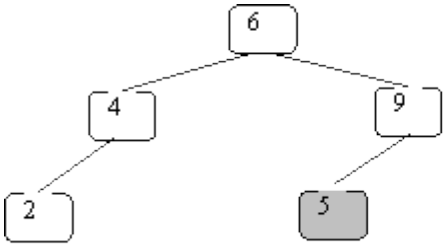
The coding steps to create the first three nodes in BST above are as follows.

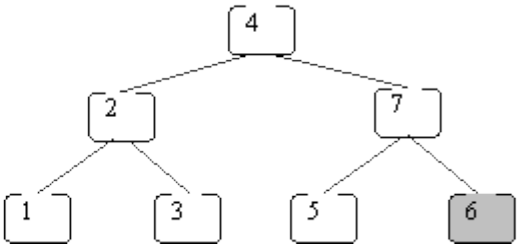
```
// Create the root node
TreeNode<Integer> root = new TreeNode<Integer>(new Integer(60) );

// Create the left child node
root.left = new TreeNode<Integer>(new Integer(55) );

// Create the right child node
root.right = new TreeNode<Integer>(new Integer(100) );
```

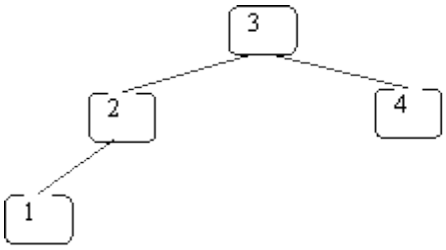
The trees below are not BSTs:

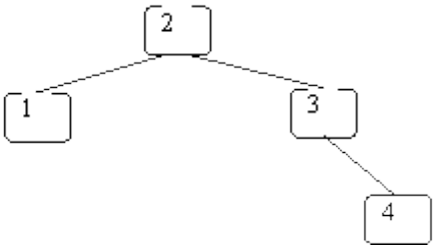




In the left one 5 is not greater than 6. In the right one 6 is not greater than 7.

Note that more than one BST can be used to store the same set of key values. For example, both of the following are BSTs that store the same set of integer keys:





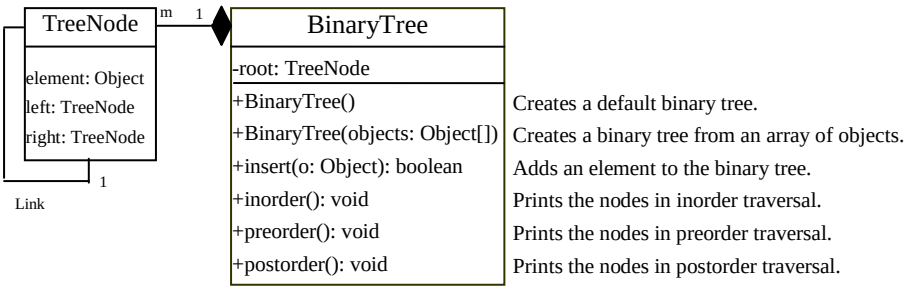
The reason binary-search trees are important is that the following operations can be implemented efficiently using a BST:

- insert a key value
- determine whether a key value is in the tree
- remove a key value from the tree
- print all of the key values in sorted order

Java Programs for BST:

(1)

DisplayBinaryTree.java – GUI application to demonstrate insertion and deletion of nodes.
(This program should be updated for Generic class representation)



(2)
Binary Tree animation applet to demonstrate search, insert and delete in a binary search tree.

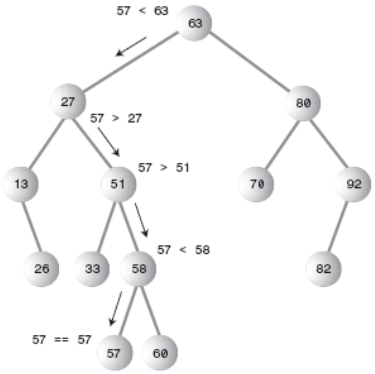
http://www.cs.armstrong.edu/liang/intro7e/exercise/Exercise25_15.html

For both insertion and searching, the process is as follows:

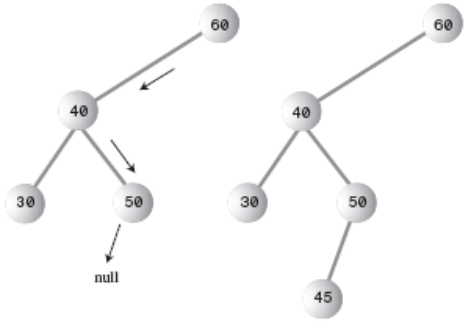
If value < parent value, go left
If value > parent value, go right

Deletion is more complicated than insertion (See Liang book for details).

Example 1:
For the BST below, show the search path to find the node 57



Example 2:
Inserting node with value 45 in BST below:



a) Before insertion b) After insertion

Exercise 1:
Show the resulting BST that results from inserting the values 5 3 7 6 2 1 4 in that order.

Show the resulting BST that results from inserting the values 1 2 3 4 5 6 7 in that order.

Show the resulting BST that results from inserting the values 4 3 5 2 6 1 7 in that order.

Tree Traversal
Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree. These methods include the *inorder*, *preorder* and *postorder* traversals.

The **inorder** traversal is to visit the left subtree of the current node first, then the current node itself, and finally the right subtree of the current node.
The inorder traversal of a binary search tree prints the node values in *ascending order*.
The process of creating a binary search tree actually sorts the data; thus, it's called the *binary tree sort*.
Inorder traversal in short: Left child, Root, Right child

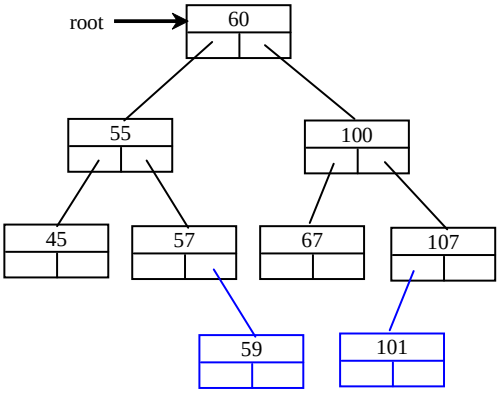
The **preorder** traversal is to visit the current node first, then the left subtree of the current node and finally the right subtree of the current node.
Preorder traversal in short: Root, then children

The **postorder** traversal is to visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself.
Postorder traversal in short: Children, then Root

Java Program : *TestBinaryTree.java*

Example 1:

Determine the inorder (Sorted order), preorder and postorder traversals of the binary search tree below.



Inorder : 45 55 57 59 60 67 100 101 107.

Preorder : 60 55 45 57 59 100 67 107 101.

Postorder : 45 59 57 55 67 101 107 100 60.

Exercise 2:

22.5 Provide the inorder, preorder and postorder traversals of the binary search tree of Fig. 22.20.

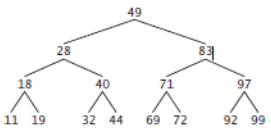


Fig. 22.20 | Binary search tree with 15 nodes.

Binary Search Tree efficiency

The advantage of using a binary Search tree (instead of, say, a linked list) is that, if the tree is reasonably balanced (shaped more like a "full" tree than like a "linear" tree) the insert, lookup, and delete operations can all be implemented to run in O(log N) time, where N is the number of stored items. For a linked list, although insert can be implemented to run in O(1) time, lookup and delete take O(N) time.

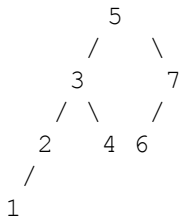
Logarithmic time is generally **much** faster than linear time. For example, for N = 1,000,000: $\log_2 N = 20$.

Of course, it is important to remember that for a **"linear"** tree (one in which every node has one child), the worst-case times for insert, lookup, and delete will be O(N).

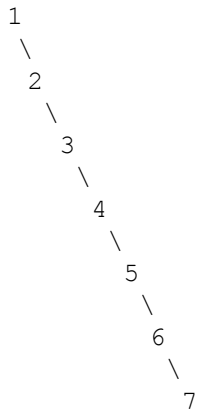
Answers to exercises:

Exercise 1:

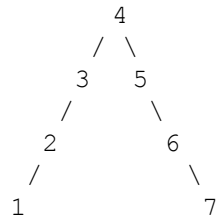
The BST that results from inserting the values 5 3 7 6 2 1 4 in that order is:



The BST that results from inserting the values 1 2 3 4 5 6 7 in that order is:



The BST that results from inserting the values 4 3 5 2 6 1 7 in that order is:



Exercise 2:

The inorder traversal is
11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

The preorder traversal is
49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

The postorder traversal is
11 19 18 32 44 40 28 69 72 71 92 99 97 83 49