



Tutorial #11

(March 7th, 2013)

Objectives:

- Introduction to Trees & Binary Trees.
- Binary Search Trees (BST) definition.
- Insertion Operation in the BST and its implementation.

Introduction to Trees & Binary Trees:

All data structures examined so far are linear data structures:

- Each element in a linear data structure has a clear predecessor and a clear successor.
- Predecessors and successors may be defined by arrival time or by relative size.

Trees are used to represent hierarchies of data:

- Each node contains some references (children), and a data element.
- The topmost node in the tree is called **the root**.
- Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called **a parent**.
- Each node can be connected to arbitrary number of children.
- Nodes with no children are called **leaves**, or **external nodes**.
- Nodes which are not leaves are called **internal nodes**.
- Nodes with the same parent are called **siblings**.
- **The depth** of a node is the number of edges from the root to the node.
- **The height** of a node is the number of edges from the node to the deepest leaf.
- **The height of a tree** is a height of the root.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move sub-trees around with minimum effort.

Binary Trees:

A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right".

- A **full binary tree** is a binary tree in which each node has exactly zero or two children, see Figure (1).
- A **complete binary tree** is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right, see Figure (1). A complete binary tree is very special tree; it provides the best possible ratio between the number of nodes and the height. The height h of a complete binary tree with n nodes is at most $O(\log n)$.

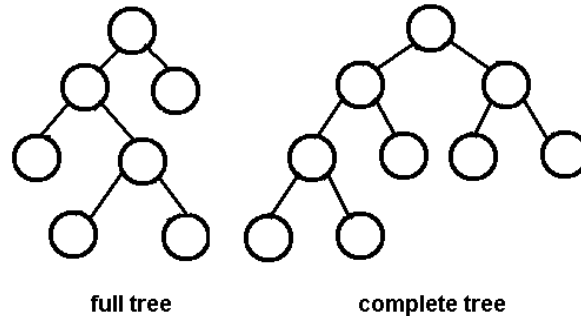


Figure (1)

Binary Search Trees (BST):

We consider a particular kind of a binary tree called a Binary Search Tree (BST). The basic idea behind this data structure is to have such a storing repository that provides the efficient way of data sorting, searching and retrieving.

A **BST** is a binary tree where nodes are ordered in the following way:

- Each node contains one key (also known as data)
- The keys in the left sub-tree are less than the key in its parent node, in short $L < P$;
- The keys in the right sub-tree are greater than the key in its parent node, in short $P < R$;
- Duplicate keys are not allowed.

In Figure (2), all nodes in the left sub-tree of 10 have keys < 10 while all nodes in the right sub-tree > 10 . Because both the left and right sub-trees of a BST are again search trees; the above definition is recursively applied to all internal nodes:

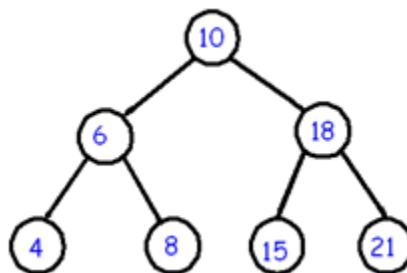


Figure (2)

Insertion Operation in the BST:

The insertion procedure is quite simple. It can be divided into two stages:

- Search for a place to put a new element;
- Insert the new element to this place.

Let us see these stages in more detail.

Search for a place

At this stage an algorithm should follow binary search tree property. If a new value is less than the current node's value, go to the left sub-tree; else go to the right sub-tree. Following this simple rule, the algorithm reaches a node which has no left or right sub-tree.

By the moment a place for insertion is found, we can say for sure that a new value has no duplicate in the tree. Initially, a new node has no children, so it is a leaf. Let us see it at the picture. Gray circles on Figure (3) indicate possible places for a new node.

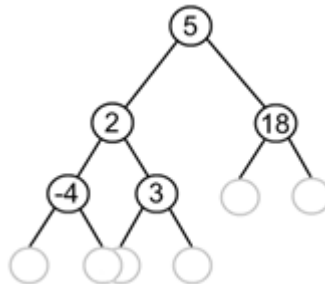


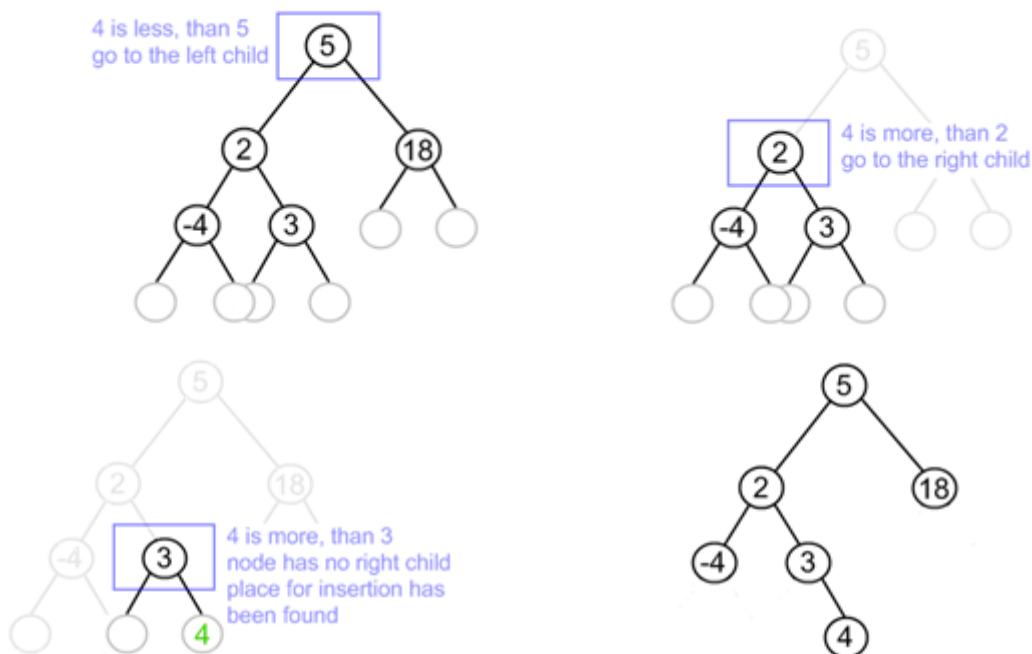
Figure (3)

Now, let's go down to algorithm itself. Here and in almost every operation on BST recursion is utilized. Starting from the root,

1. Check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,
2. If a new value is less than the node's value:
 - If a current node has no left child, place for insertion has been found;
 - Otherwise, handle the left child with the same algorithm.
3. If a new value is greater than the node's value:
 - If a current node has no right child, place for insertion has been found;
 - Otherwise, handle the right child with the same algorithm.

Example

Insert 4 to the tree, shown on Figure (3).



Exercise:

Given a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31

Draw a binary search tree by inserting the above numbers from left to right.

Code snippets:

The only difference between the algorithm above and the real routine is that first we should check, if a root exists. If not, just create it and don't run a common algorithm for this special case. This can be done in the *BinarySearchTree* class. Principal algorithm is implemented in the *BSTNode* class.

Code
<pre>public class BinarySearchTree { ... public boolean insert(int value) { if (root == null) { root = new BSTNode(value); return true; } else return root.insert(value); } } public class BSTNode { ... public boolean insert(int value) { if (value == this.value) return false; else if (value < this.value) { if (left == null) { left = new BSTNode(value); return true; } else return left.insert(value); } else if (value > this.value) { if (right == null) { right = new BSTNode(value); return true; } else return right.insert(value); } return false; } }</pre>