Binary Search Tree and Tree Traversal - Inorder, Preorder, Postorder implemented in Java

November 18, 2013 by Mohamed Sanaulla 8 Comments

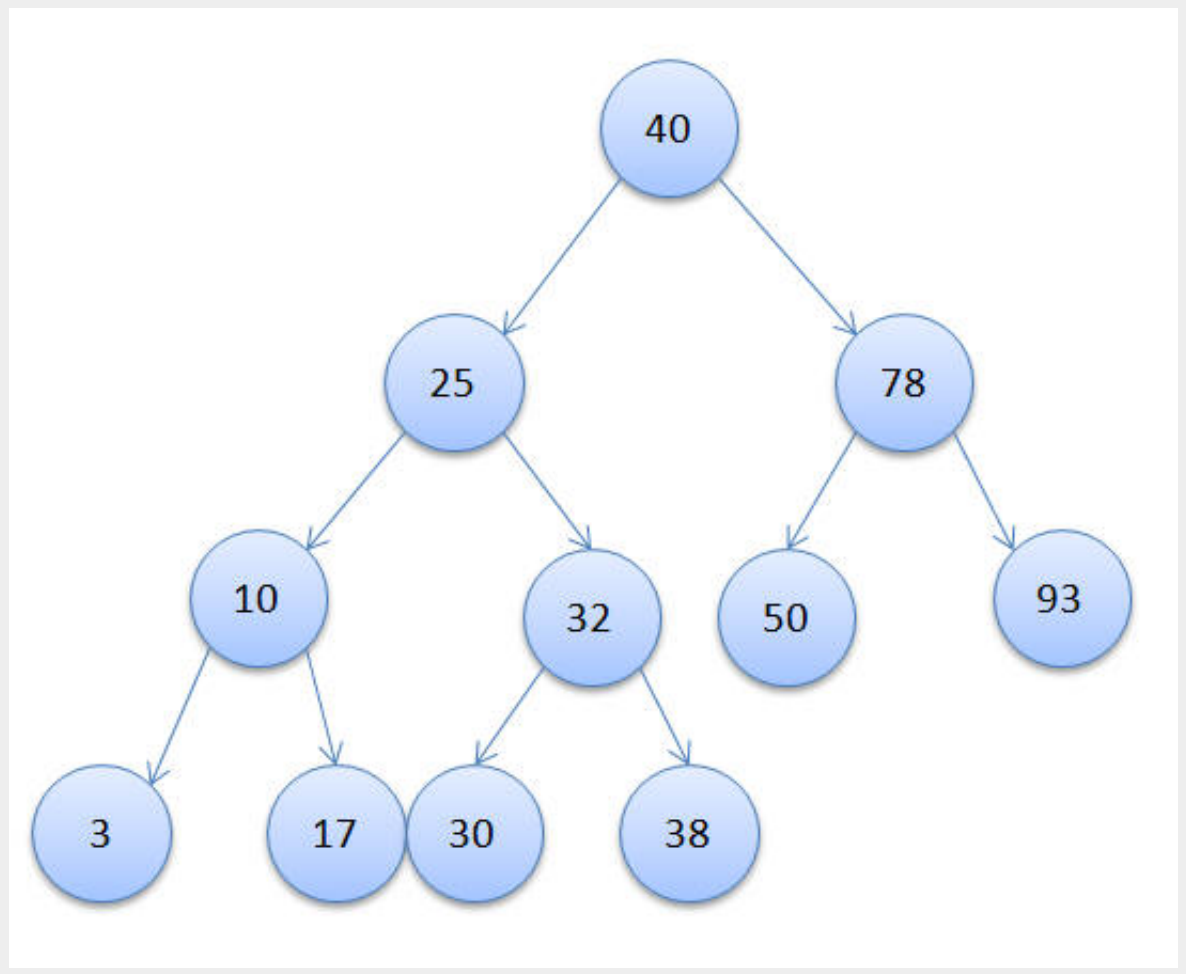
Most of the students fresh out of their e studies or those who are still studying w concept of Binary Search Trees fresh in t But with most of the people who have b college for many years now will kind of be having a not so clear idea of Binary Search trees unless they have been related concepts at their work. **In this tutorial I would show how to implement a Binary Search Tree (BST) in also show the following operations:**

1. Inserting/Building a BST
2. Finding maximum value node in BST
3. Finding minimum value node in BST
4. Inorder Traversal of BST
5. Preorder Traversal of BST
6. Postorder Traversal of BST

What is a Binary Search Tree (BST)?

Binary Search Tree (BST) is a binary tree data structure with a special feature where in the value store at each noc than or equal to the value stored at its left sub child and lesser than the value stored at its right sub child. Lets loc

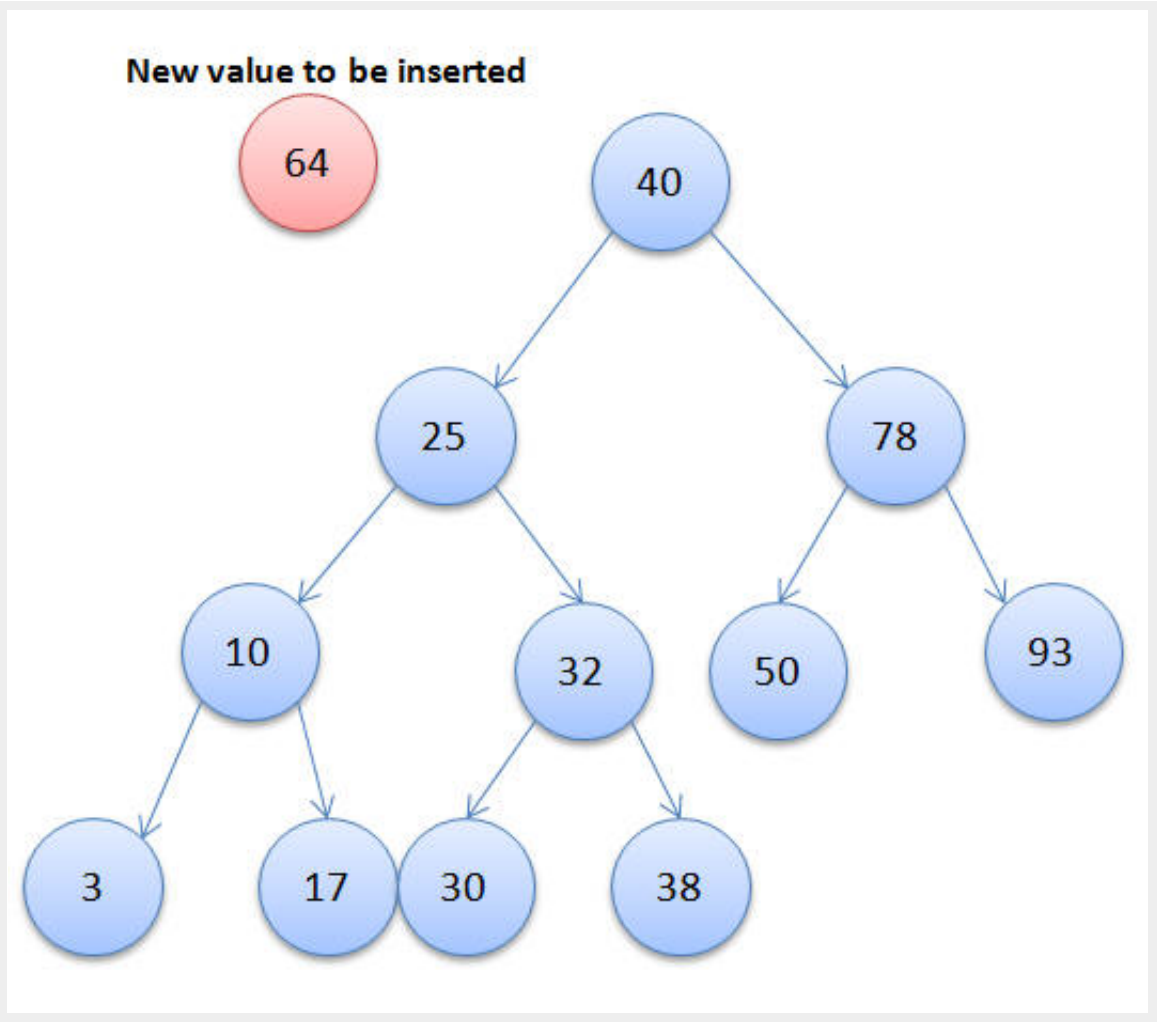
example of a BST:

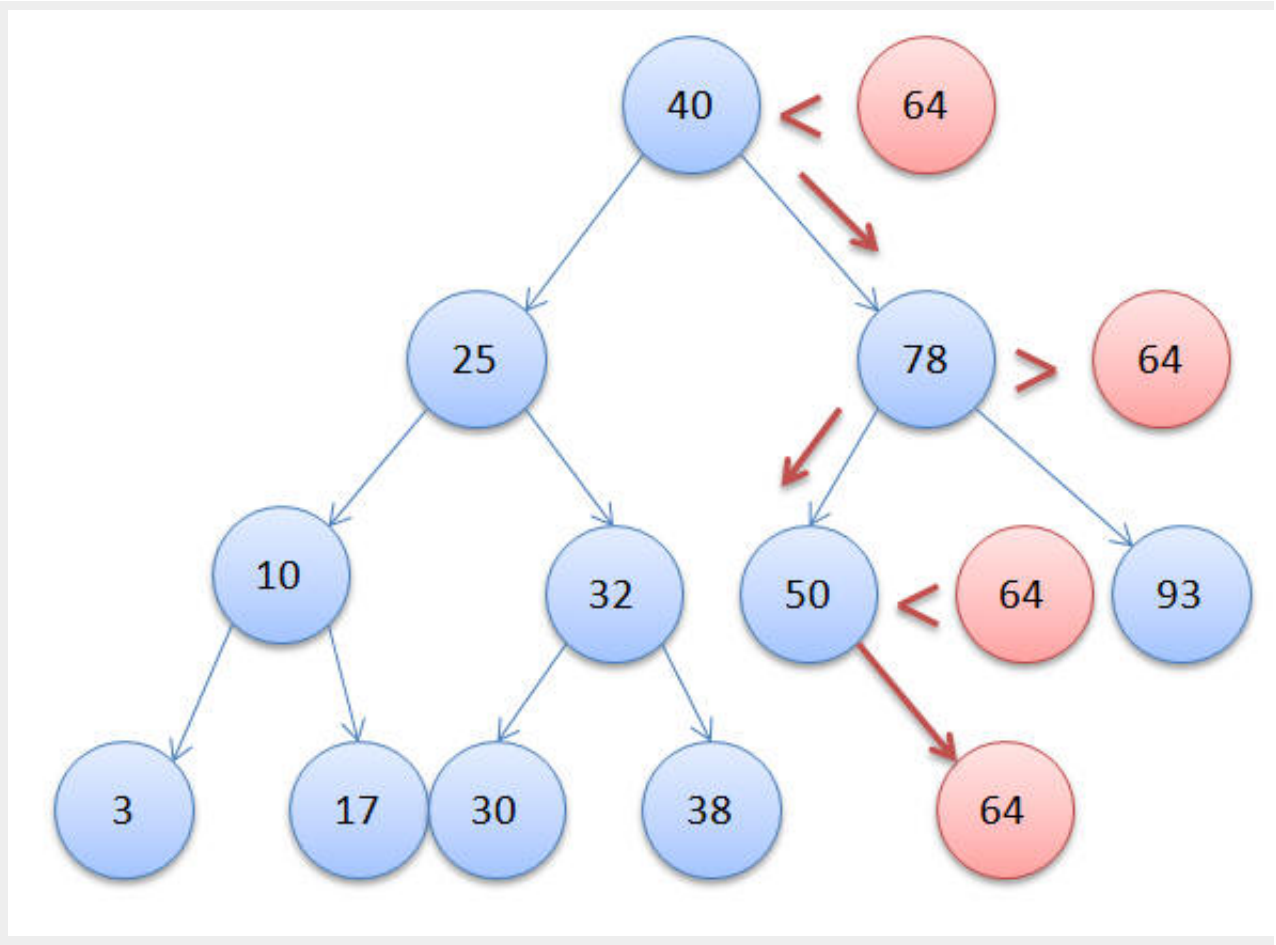


In the above example you can see that at each node the value in the left child is lesser than or equal to the value i and the value in the right child is greater than the value in the node.

Building a Binary Search Tree (BST)

Now that we have seen how a BST looks, let me show you how one can build a BST and insert nodes into the tree implementing the algorithm in Java. **The basic idea is that at each node we compare with the value being ins the value is lesser then we traverse through the left sub tree and if the value is greater we traverse throu right subtree.** Suppose we have to insert the value **64** in the above BST, lets look at the nodes traversed before it at the right place:


New value to be inserted
64
40
25
78
10
32
50
93
3
17
30
38



Each node in the BST is represented by the below java class:

```
1  public class Node<T> {
2    public int value;
3    public Node left;
4    public Node right;
5
6    public Node(int value) {
7      this.value = value;
8    }
9
10 }
```

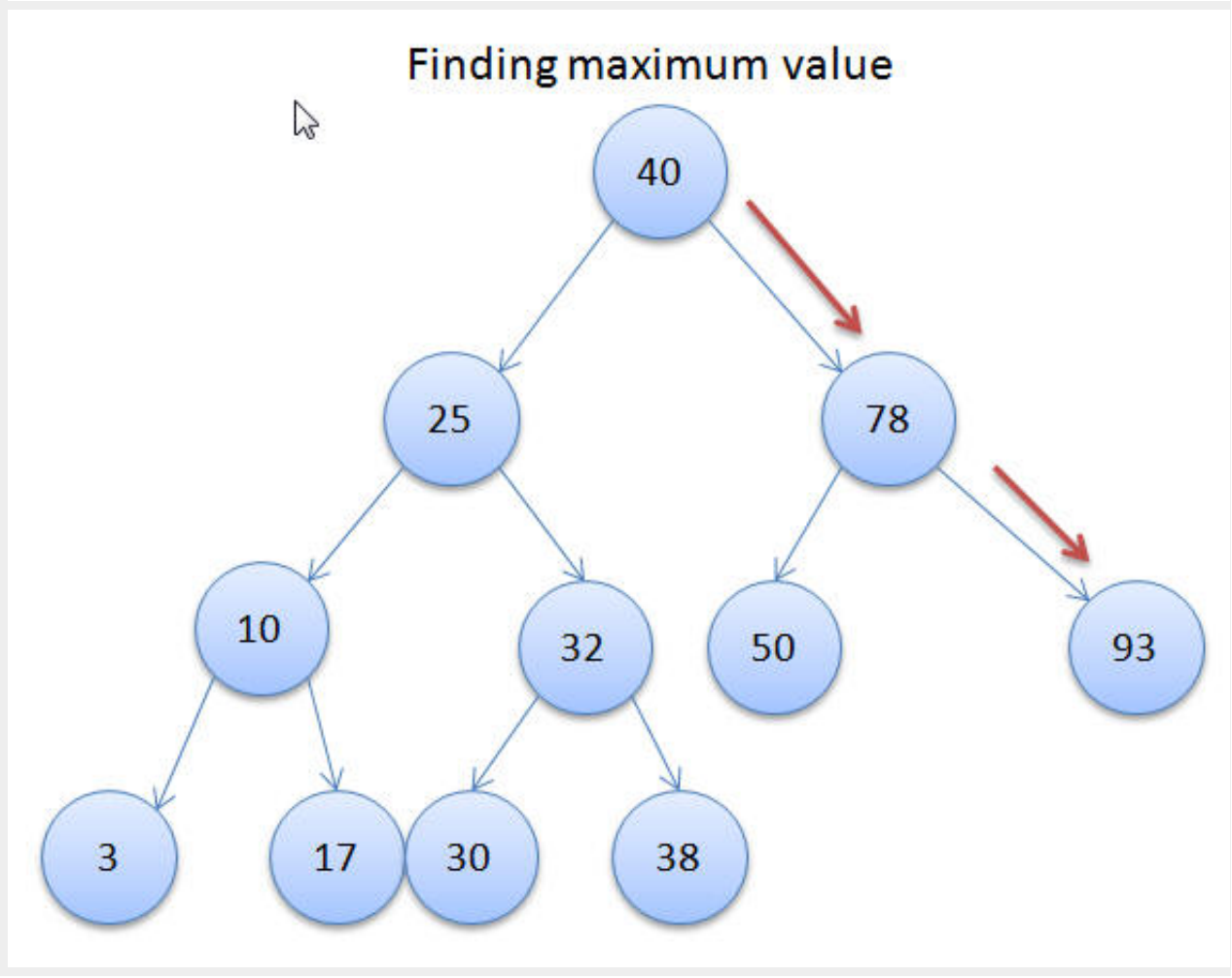
Lets look at the code in Java for achieving the above logic:

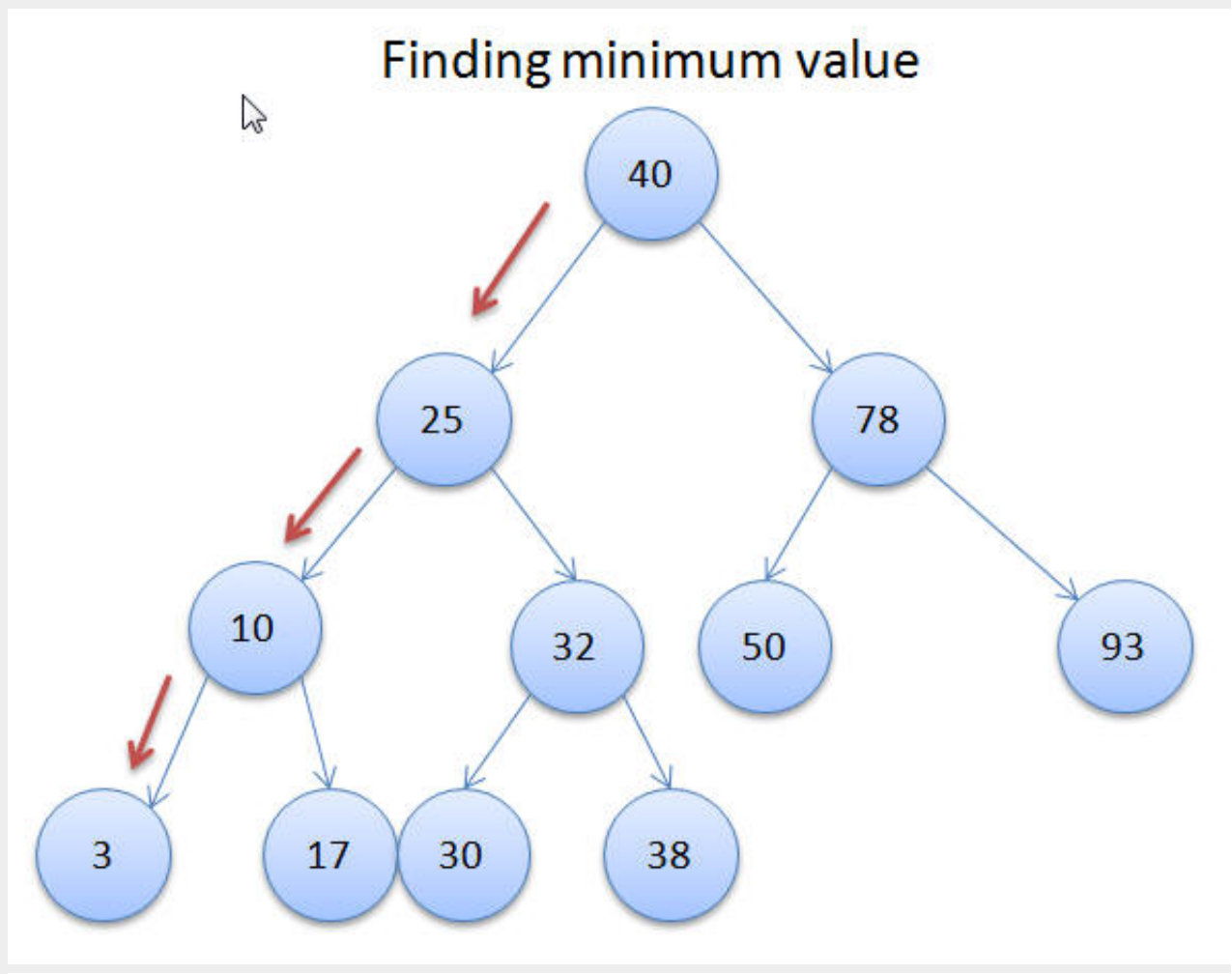
```
1  public class BinarySearchTree {
2    public Node root;
3
4    public void insert(int value){
5      Node node = new Node<>(value);
6
7      if ( root == null ) {
8        root = node;
9        return;
10     }
11
12     insertRec(root, node);
13
14   }
```

```
15
16    private void insertRec(Node latestRoot, Node node){
17
18      if ( latestRoot.value > node.value){
19
20        if ( latestRoot.left == null ){
21          latestRoot.left = node;
22          return;
23        }
24        else{
25          insertRec(latestRoot.left, node);
26        }
27      }
28      else{
29        if (latestRoot.right == null){
30          latestRoot.right = node;
31          return;
32        }
33        else{
34          insertRec(latestRoot.right, node);
35        }
36      }
37    }
38  }
```

Finding Maximum and Minimum Value in BST

If you have noticed in the above example that the leftmost node has the lowest value and the rightmost node has value. This is due to the sorted nature of the tree.





Using this principle the below methods return us the lowest and highest value in the Binary Search Tree:


```
1  /**
2   * Returns the minimum value in the Binary Search Tree.
3   */
4  public int findMinimum(){
```

```
5    if ( root == null ){
6      return 0;
7    }
8    Node currNode = root;
9    while(currNode.left != null){
10     currNode = currNode.left;
11   }
12   return currNode.value;
13 }
14
15 /**
16  * Returns the maximum value in the Binary Search Tree
17  */
18 public int findMaximum(){
19   if ( root == null){
20     return 0;
21   }
22
23   Node currNode = root;
24   while(currNode.right != null){
25     currNode = currNode.right;
26   }
27   return currNode.value;
28 }
```

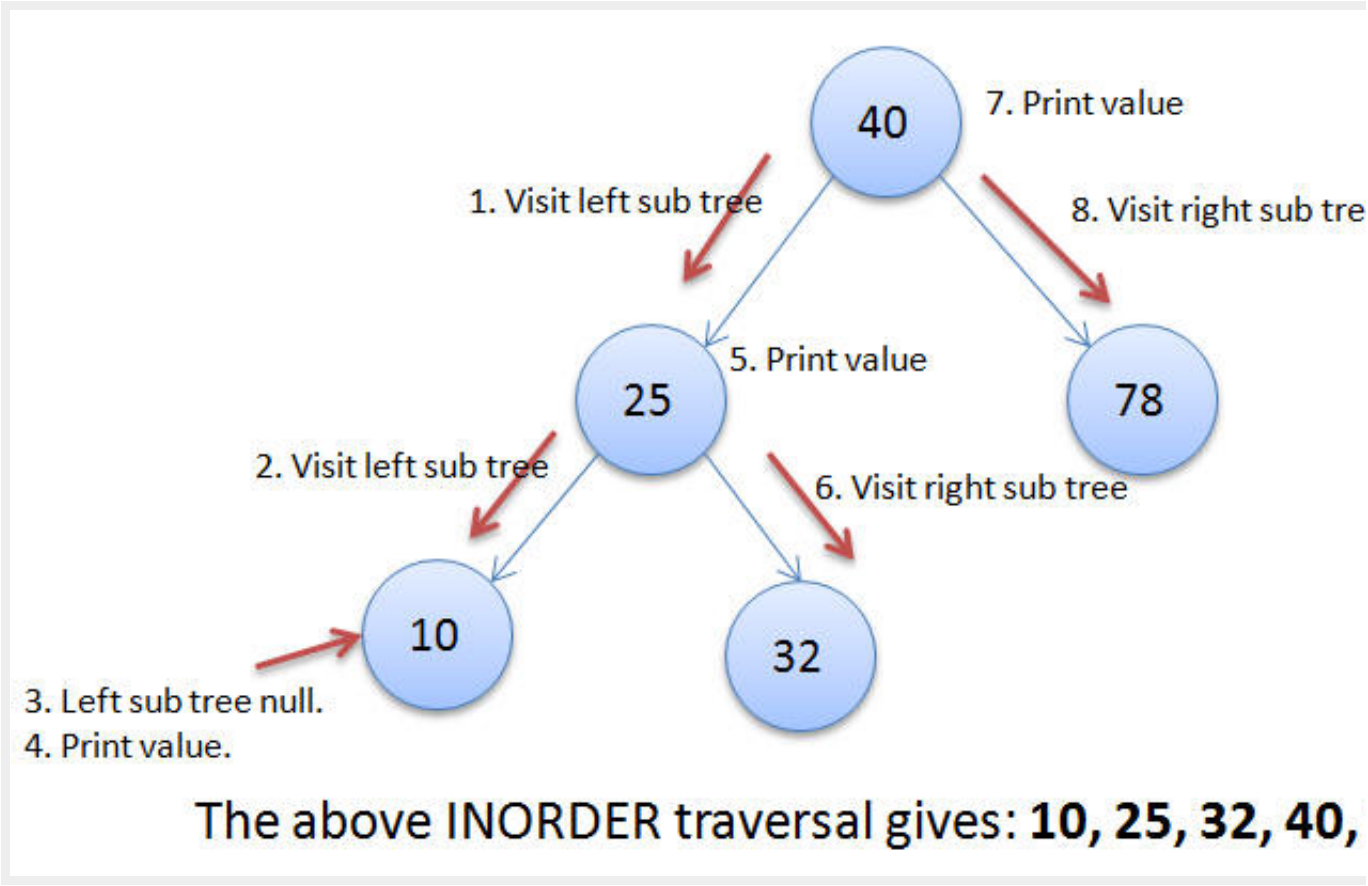
Traversing the Binary Search Tree (BST)

Traversing the tree or BST in this case is visiting each of the nodes present in the tree and performing some opera the value present in the node which in this case will be printing the value present in the node. **When we traverse we have to visit the value present in the node, then node's right sub tree and the left sub tree**. Visiting the left sub tree will be a recursive operation. The order in which we perform the three operations i.e visiting the valu tree and left sub tree gives rise to three traversal techniques:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

Inorder Traversal

In this traversal the left sub tree of the given node is visited first, then the value at the given node is printed and t right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either th tree is empty or the right sub tree is empty.

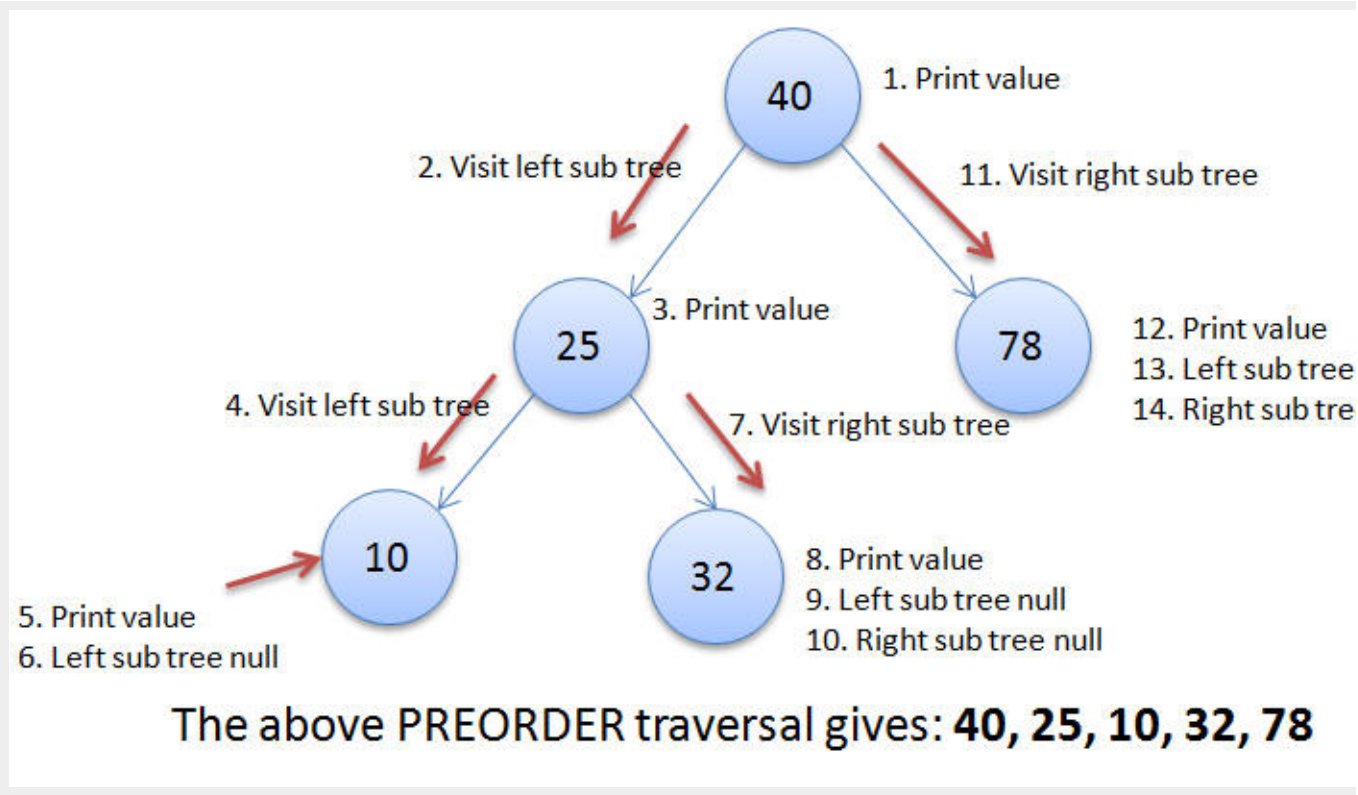


Applying the Inorder traversal for the give example we get: 3, 10, 17, 25, 30, 32, 38, 40, 50, 78, 78, 93.

```
1  /**
2   * Printing the contents of the tree in an inorder way.
3   */
4  public void printInorder(){
5    printInOrderRec(root);
6    System.out.println("");
7  }
8
9  /**
10  * Helper method to recursively print the contents in an inorder way
11  */
12 private void printInOrderRec(Node currRoot){
13   if ( currRoot == null ){
14     return;
15   }
16   printInOrderRec(currRoot.left);
17   System.out.print(currRoot.value+", ");
18   printInOrderRec(currRoot.right);
19 }
```

Preorder traversal

In this traversal the value at the given node is printed first and then the left sub tree of the given node is visited a right sub tree of the given node is visited. This process is applied recursively all the node in the tree until either th tree is empty or the right sub tree is empty.

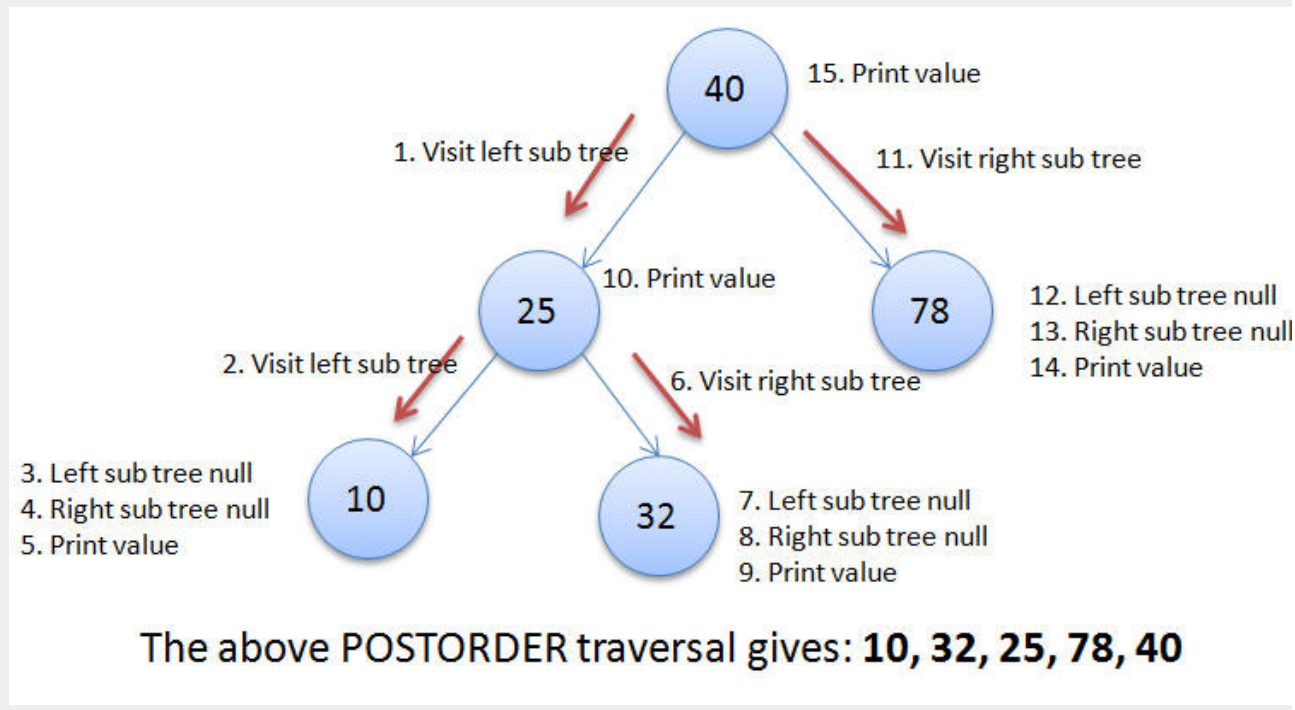


Applying the Preorder traversal for the give example we get: 40, 25, 10, 3, 17, 32, 30, 38, 78, 50, 78, 93.

```
1  /**
2   * Printing the contents of the tree in a Preorder way.
3   */
4  public void printPreorder() {
5    printPreOrderRec(root);
6    System.out.println("");
7  }
8
9  /**
10  * Helper method to recursively print the contents in a Preorder way
11  */
12 private void printPreOrderRec(Node currRoot) {
13   if (currRoot == null) {
14     return;
15   }
16   System.out.print(currRoot.value + ", ");
17   printPreOrderRec(currRoot.left);
18   printPreOrderRec(currRoot.right);
19 }
```

Postorder Traversal

In this traversal the left sub tree of the given node is traversed first, then the right sub tree of the given node is tra then the value at the given node is printed. This process is applied recursively all the node in the tree until either t tree is empty or the right sub tree is empty.



Applying the Postorder traversal for the give example we get: 3, 17, 10, 30, 38, 32, 25, 50, 93, 78, 78, 40.

```
1  /**
2   * Printing the contents of the tree in a Postorder way.
3   */
4  public void printPostorder() {
5    printPostOrderRec(root);
6    System.out.println("");
7  }
8
9  /**
10  * Helper method to recursively print the contents in a Post     r way
11  */
12 private void printPostOrderRec(Node currRoot) {
13   if (currRoot == null) {
14     return;
15   }
16   printPostOrderRec(currRoot.left);
17   printPostOrderRec(currRoot.right);
18   System.out.print(currRoot.value + ", ");
19
20 }
```

The complete code which builds the tree for the example explained in this code and prints the maximum, minimu inorder traversal, preorder traversal and post order traversal can be found below:

```
1  /**
2   * Represents a node in the Binary Search Tree.
3   */
4  public class Node<T> {
5    //The value present in the node.
6    public int value;
7
8    //The reference to the left subtree.
9    public Node left;
10
```

```
  //The reference to the right subtree.
  public Node right;

  public Node(int value) {
    this.value = value;
  }

}

/**
 * Represents the Binary Search Tree.
 */
public class BinarySearchTree {

  //Refrence for the root of the tree.
  public Node root;

  public BinarySearchTree insert(int value) {
    Node node = new Node<>(value);

    if (root == null) {
      root = node;
      return this;
    }

    insertRec(root, node);
    return this;
  }

  private void insertRec(Node latestRoot, Node node) {

    if (latestRoot.value > node.value) {

      if (latestRoot.left == null) {
        latestRoot.left = node;
        return;
      } else {
        insertRec(latestRoot.left, node);
      }
    } else {
      if (latestRoot.right == null) {
        latestRoot.right = node;
        return;
      } else {
        insertRec(latestRoot.right, node);
      }
    }
  }

  /**
   * Returns the minimum value in the Binary Search Tree.
   */
  public int findMinimum() {
    if (root == null) {
      return 0;
    }
    Node currNode = root;
    while (currNode.left != null) {
      currNode = currNode.left;
    }
    return currNode.value;
  }

  /**
   * Returns the maximum value in the Binary Search Tree
```

```
76     */
77    public int findMaximum() {
78      if (root == null) {
79        return 0;
80      }
81
82      Node currNode = root;
83      while (currNode.right != null) {
84        currNode = currNode.right;
85      }
86      return currNode.value;
87    }
88
89    /**
90     * Printing the contents of the tree in an inorder way.
91     */
92    public void printInorder() {
93      printInOrderRec(root);
94      System.out.println("");
95    }
96
97    /**
98     * Helper method to recursively print the contents in an inorder way
99     */
100   private void printInOrderRec(Node currRoot) {
101     if (currRoot == null) {
102       return;
103     }
104     printInOrderRec(currRoot.left);
105     System.out.print(currRoot.value + ", ");
106     printInOrderRec(currRoot.right);
107   }
108
109   /**
110    * Printing the contents of the tree in a Preorder way.
111    */
112   public void printPreorder() {
113     printPreOrderRec(root);
114     System.out.println("");
115   }
116
117   /**
118    * Helper method to recursively print the contents in a Preorder way
119    */
120   private void printPreOrderRec(Node currRoot) {
121     if (currRoot == null) {
122       return;
123     }
124     System.out.print(currRoot.value + ", ");
125     printPreOrderRec(currRoot.left);
126     printPreOrderRec(currRoot.right);
127   }
128
129   /**
130    * Printing the contents of the tree in a Postorder way.
131    */
132   public void printPostorder() {
133     printPostOrderRec(root);
134     System.out.println("");
135   }
136
137   /**
138    * Helper method to recursively print the contents in a Postorder way
139    */
140   private void printPostOrderRec(Node currRoot) {
```

```
141     if (currRoot == null) {
142       return;
143     }
144     printPostOrderRec(currRoot.left);
145     printPostOrderRec(currRoot.right);
146     System.out.print(currRoot.value + ", ");
147 
148   }
149 }
150 
151 public class BinarySearchTreeDemo {
152 
153   public static void main(String[] args) {
154     BinarySearchTree bst = new BinarySearchTree();
155     bst .insert(40)
156         .insert(25)
157         .insert(78)
158         .insert(10)
159         .insert(3)
160         .insert(17)
161         .insert(32)
162         .insert(30)
163         .insert(38)
164         .insert(78)
165         .insert(50)
166         .insert(93);
167     System.out.println("Inorder traversal");
168     bst.printInorder();
169 
170     System.out.println("Preorder Traversal");
171     bst.printPreorder();
172 
173     System.out.println("Postorder Traversal");
174     bst.printPostorder();
175 
176     System.out.println("The minimum value in the BST: " + bst.findMinimum());
177     System.out.println("The maximum value in the BST: " + bst.findMaximum());
178 
179   }
180 }
```

Comments

10 comments