Java OOP—Binary Search Tree

This tutorial is about creating a simple binary search tree in Java programming language by implementing Java Object-Oriented Programming (Java OOP). This Binary Search Tree is to store the integer values. The program provides a menu of choices to operate the Binary Search Tree data structure. See the sample menu below:

==========

Binary Search Tree Operations Menu

1. Add a node

2. Delete a node

3. Show min node

5. Show max node

6. Find a node

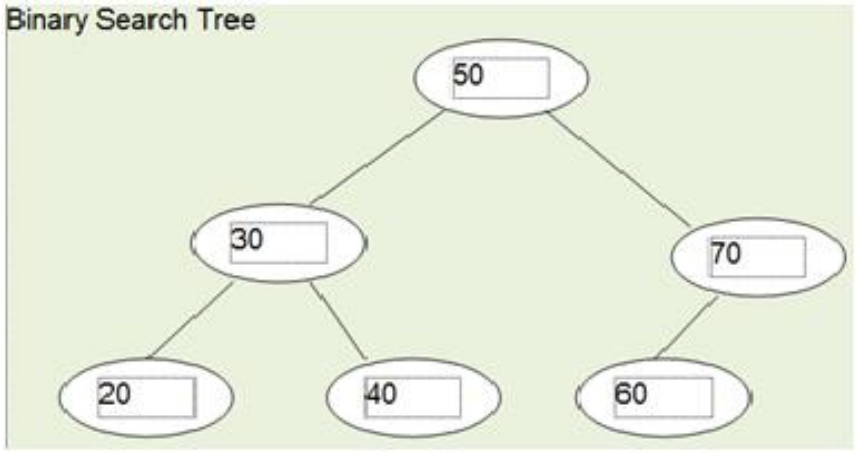
7.Print all nodes

7. Exit

Enter your choice:1

Understand Binary Search Tree

In a Binary Search Tree data structure that stores the integer values, there is root node. The nodes that are less than the root node will be put in the left side and the nodes that are greater than the root node will be put in the right side. The nodes that are in the higher positions are called the parent nodes and the nodes that are in the lower positions are called child nodes. The parent nodes may have more than one child. It also may not have a child. The node that doesn't have a child is called leaf.



Step 1: Defining the binary search tree node

To implement the binary search tree in Java, Each node of the binary search tree should contain 4 components--data, left child link, right child link and parent link. The left child link points to the left child and the right child link points to the right child. The parent link will be used to store the parent node.

```
class TreeNode{
        TreeNode(int elem) { data = elem;left=null;right=null;parent=null; }
        public int data; //node data
        public TreeNode left;//left child link
        public TreeNode right;//right child link
        public TreeNode parent;//parent link

}
```

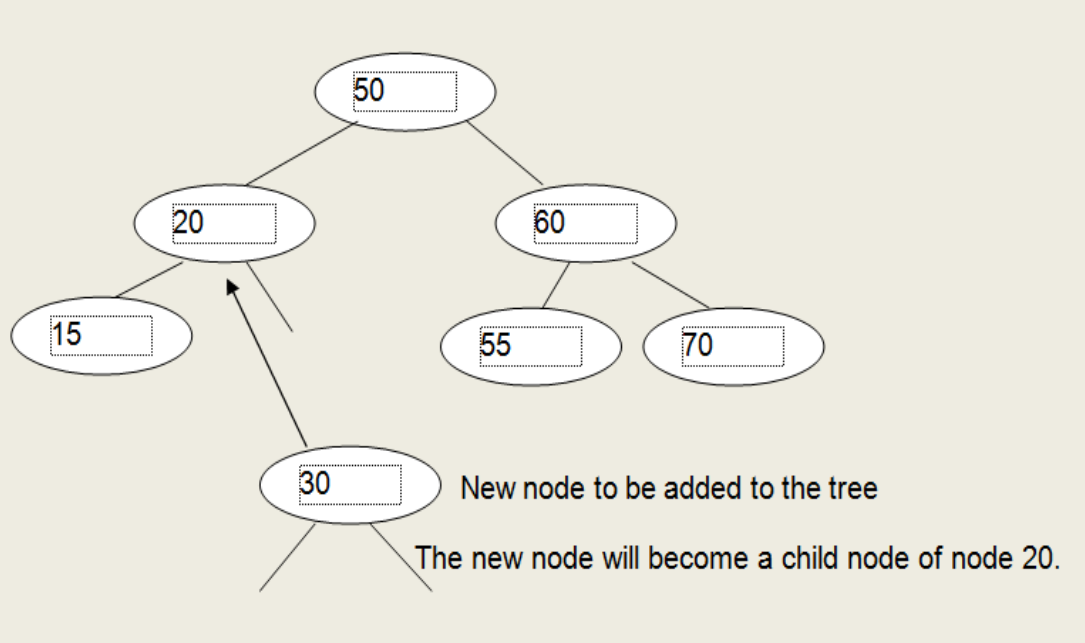
To operate the binary search tree, we also provide another abstract class called Cls that will be inherited by the our main class: BTree.

```
abstract class Cls{

public TreeNode insert(TreeNode Tree, int Tar){

return null;

}

public void delete(TreeNode Tree){

}

public TreeNode findmin(TreeNode Tree){
return null;
}

public TreeNode find(TreeNode Tree,int Tar){
return null;
}

public void printall(TreeNode Tree){

}


}
```

Step 2: Add node to the tree

To add a new node to the tree, we need a recursive process to locate the appropriate location of the tree. The process starts from the root node. If the tree is empty, the new node will become the root node. If the tree is not empty, the new node will be checked against the parent node. If it is less than

the parent node, the process will look for the proper location in the left side, otherwise, it looks in the right side. If the parent node is equal to the new node, it will do nothing. This process continues recursively until it finds the proper location.




```
///java code to insert a node to the tree
public TreeNode insert(TreeNode Tree, int Tar)
{

//The place to insert the node
if(Tree==null){
TreeNode item= new TreeNode(Tar);
Tree=item;

}

else{

//insert to the left
if(Tar<Tree.data) {Tree.left=insert(Tree.left,Tar);Tree.left.parent=Tree;}

//insert to the right
else if(Tar>Tree.data){Tree.right=insert(Tree.right,Tar);Tree.right.parent=Tree;}

}

return Tree;

}
```

Step 3: Finding the min and max nodes

In binary search tree, the smallest node is in the left side and the largest node is in the right side.

To find the smallest node, the process will check the parent node. In case that the parent node is not empty, if it doesn't a left child node, the smallest node is the parent node, otherwise the smallest node is its left child node.

```
//Find the min node

public TreeNode findmin(TreeNode Tree){
        if(Tree==null) return null;
        else if(Tree.left==null) return Tree;
        else return findmin(Tree.left);
}
```

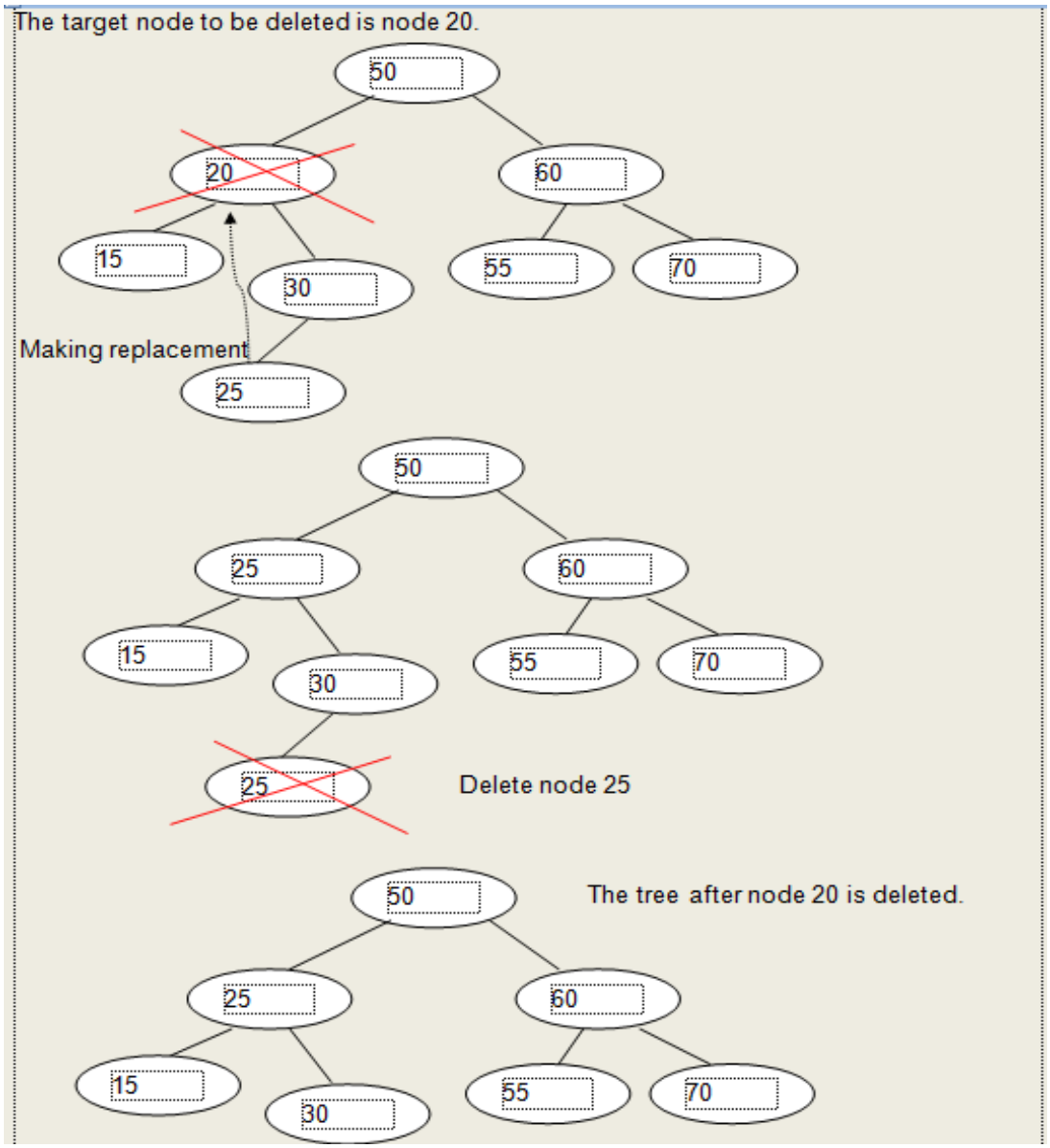
To find the largest node, the process will check the parent node. In case that the parent node is not empty, if it doesn't a right child node, the largest node is the parent node, otherwise the largest node is its right child node.

```
//Find the max node

public TreeNode findmax(TreeNode Tree){
        if(Tree==null) return null;
        else if(Tree.right==null) return Tree;
        else return findmax(Tree.right);
}
```

Step 4: Delete a node of the tree

Deleting a node from the tree is a more complicate process. To delete a node there are two important things to condition:

1. The target node has two children nodes: To delete a node that has two children node, you need to replace the data value of the node with the data value of the smallest node of its right child node. Then the smallest node used in replacement will be delete.



2. The target node has only one or zero child node. If the node to be deleted has only right child node(Its left child is empty), we let the node takes its right child node's address, otherwise we let the node takes its left child node. This process also can be applied to the node that has zero child.

```
///delete a node from the tree

public void delete(TreeNode Tree,int Tar)
{

TreeNode Min,Temp;
if(Tree==null) {return;}

else if(Tar<Tree.data) delete(Tree.left,Tar);//look in the left
else if(Tar>Tree.data) delete(Tree.right, Tar);//look in the right
else{ //found node to delete

   if(Tree.left!=null && Tree.right!=null) //two children
   {
    Min=findmin(Tree.right);
```

```
    Tree.data=Min.data;
    delete(Tree.right,Tree.data);

}

   else{ //one or zero child

     if(Tree.left==null)
     {
     if(Tree.parent==null) Root=Tree.right; //The root node is to be deleted.
     else{
     if(Tree.right!=null){
     Tree.right.parent=Tree.parent;
     }

  if(Tree==Tree.parent.left)
     Tree.parent.left=Tree.right;
  else Tree.parent.right=Tree.right;
  }

}
   else if(Tree.right==null)
    {
     if(Tree.parent==null) Root=Tree.left;

      else{

       Tree.left.parent=Tree.parent;
       if(Tree==Tree.parent.left)
          Tree.parent.left=Tree.left;
       else Tree.parent.right=Tree.left;
             }
        }
```

Step 5: Searching for a node in the tree

To search for a node in the tree, we need a recursive function find(). This function firstly tests whether the tree is empty. If the tree is not empty, the target value is compared firstly with the data of the root node. If the target value is less than the data of the root node, the search process will continue to left side. It continues to the right side if the target value is greater than the data. The process stops when the data of a node is equal to the target value or when it reaches the end of the tree.

```
public TreeNode find(TreeNode Tree,int Tar){
if(Tree==null) return null;
if(Tar<Tree.data) //look in the left side
  return find(Tree.left,Tar);
else if(Tar>Tree.data)//look in the right side
  return find(Tree.right,Tar);
else return Tree; //found node
}
```

Step 6: Print all data of the tree

To print the data of every node of the tree, you need to traverse through the tree. The process will start from the root node and go to the left side and right side of the tree.

```
//Print all nodes
public void printall(TreeNode Tree)
{
if(Tree!=null){

 System.out.print(Tree.data+"\t");
 printall(Tree.left);
 printall(Tree.right);
}

}
```

Step 7: Display BS Tree operations menu

We define a showmenu() function to display a list of choices. We also need another function called select() that allows the user to choose each of the options and decides whether he/he wants to exit the program.

```
public static void showmenu(){

System.out.println("================================");
System.out.println("Binary Tree Operations Menu");
System.out.println("================================");
System.out.println("1.Add a new item");
System.out.println("2.Delete an item");
System.out.println("3.Show min node");
System.out.println("4.Show max node");
System.out.println("5.Find a node");
System.out.println("6.Print all nodes");

System.out.println("7.Exit");

}

public static void select(){
Scanner sc=new Scanner(System.in);
int val, ch,yes=0;
BTree mylist=new BTree();
TreeNode temp=null;
showmenu();

try{
   while(yes==0){
   System.out.print("Enter your choice:");
```

```
   ch=sc.nextInt();
   switch(ch){

   case 1:
    System.out.print("Value:");
    val=sc.nextInt();
    mylist.Root=mylist.insert(mylist.Root,val);
    break;

   case 2:
     System.out.print("Value to be deleted:");
     val=sc.nextInt();
     mylist.delete(mylist.Root,val);
    break;


   case 3:
     temp=mylist.findmin(mylist.Root);
     System.out.println("The min node is:"+temp.data);
     break;

   case 4:
     temp=mylist.findmax(mylist.Root);
     System.out.println("The max node is:"+temp.data);
     break;

   case 5:
     System.out.print("Value:");
     val=sc.nextInt();
     temp=mylist.find(mylist.Root,val);
    if(temp!=null)
      System.out.println("The found node is:"+temp.data);
    else System.out.println("Not found");
    break;
   case 6:
     System.out.println("All items:");
     mylist.printall(mylist.Root);
     break;

   case 7: System.exit(0);

   default: System.out.println("Invalid choice!");

}


  System.out.print("\nContinue? Press 0 to continue:");
  yes=sc.nextInt();

}

}catch(Exception e){}
```

}

By: http://www.worldbestlearningcenter.com