DOPE (/blog/)

Binary Search Tree in Java

Sept. 27, 2018 TREE (/blog/tag/tree/?tag=tree) BINARY TREE (/blog/tag/binary-tree/?tag=binary-tree) BINARY SEARCH TREE (/blog/tag/binary-search-tree/?tag=binary-search-tree) DATA STRUCUTRE (/blog/tag/data-strucutre/?tag=data-strucutre) 1216

(https://play.google.com/store/apps/details?id=com.blogsdope&pcampaignid=MKT-Other-global-all-co-prtnr-py-PartBadge-Mar2515-1)

Previous

1. Trees in Computer Science (https://www.codesdope.com/blog/article/trees-in-computer-science)
2. Binary Tree (https://www.codesdope.com/blog/article/binary-trees)
3. Binary Tree in Java: Traversals, Finding Height of Node (https://www.codesdope.com/blog/article/binary-tree-in-java-traversals-finding-height-of-n)
4. Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree)

This post is about the coding implementation of ***BST*** in Java and its **explanation**. To learn about the concepts behind a binary search tree, the post Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree) would be helpful.

```
class Node{
    private int data;
    private Node left;
    private Node right;

    public Node(int element){
        data = element;
        left = null;
        right = null;
    }

    public void setRightChild(Node n)
    {
        right = n;
    }

    public void setLeftChild(Node n){
        left = n;
    }

    public Node getRightChild(){
        return right;
    }

    public Node getLeftChild(){
        return left;
    }

    public int getData(){
        return data;
    }

    public void setData(int x){
        data = x;
    }
}

class Tree{

    // mathod to search for an element in a tree
    public static Node search(int x, Node n){
        if (n==null || n.getData()==x) //if data of root is x then the el
            return n;
        else if(n.getData()>x) // x is greater, so we will search the rig
            return search(x, n.getLeftChild());
        else //x is smaller than the data, so we will search the left sub
            return search(x, n.getRightChild());
    }

    //method to find the minimum value in a tree
    public static Node findMinimum(Node root){
        if(root==null)
            return null;
        else if(root.getLeftChild() != null) // node with minimum value w
            return findMinimum(root.getLeftChild()); // left most element
        return root;
    }

    // method to insert a new node
    public static Node insert(Node root, int x){
        if (root == null)
            return new Node(x);
        else if(x>root.getData()) // x is greater. Should be inserted to
            root.setRightChild(insert(root.getRightChild(),x));
        else // x is smaller should be inserted to left
            root.setLeftChild(insert(root.getLeftChild(),x));
        return root;
    }

    // method to delete a node
    public static Node delete(Node root, int x){
        //searching for the item to be deleted
        if(root==null)
            return null;
        if (x>root.getData())
            root.setRightChild(delete(root.getRightChild(), x));
        else if(x<root.getData())
            root.setLeftChild(delete(root.getLeftChild(), x));
        else
        {
            //No Children
            if(root.getLeftChild()==null && root.getRightChild()==null)
            {
                root = null;
                return null;
            }

            //One Child
            else if(root.getLeftChild()==null || root.getRightChild()==nu
            {
                Node temp;
                if(root.getLeftChild()==null)
                    temp = root.getRightChild();
                else
                    temp = root.getLeftChild();
                root = null;
                return temp;
            }

            //Two Child
            else
            {
                Node temp = findMinimum(root.getRightChild());
                root.setData(temp.getData());
                root.setRightChild(delete(root.getRightChild(), temp.getD
            }
        }
        return root;
    }

    //method for inorder
    public static void inorder(Node root){
        if(root!=null){ // checking if the root is not null
            inorder(root.getLeftChild()); // visiting left child
            System.out.print(" "+root.getData()+" "); // printing data at
            inorder(root.getRightChild()); // visiting right child
        }
    }

    /*
                  20
```

```
                   /    \
                  /      \
                 5       30
              /   \      /\
             /      \   /  \
            1      15 25   40
                  /           \
                 /             \
                9             45
             /   \           /
            /     \         /
           7      12       42
    */

    public static void main(String[] args){
        Node root;
        root = new Node(20);
        insert(root,5);
        insert(root,1);
        insert(root,15);
        insert(root,9);
        insert(root,7);
        insert(root,12);
        insert(root,30);
        insert(root,25);
        insert(root,40);
        insert(root, 45);
        insert(root, 42);

        inorder(root);
        System.out.println("");

        root = delete(root, 1);
        /*
                         20
                       /     \
                      /       \
                     5       30
                       \     /\
                        \    /  \
                        15 25   40
                      /            \
                     /              \
                    9               45
                  /   \            /
                 /     \          /
                7      12       42
        */

        root = delete(root, 40);
        /*
                         20
                       /     \
                      /       \
                     5       30
                       \     /\
                        \    /  \
                        15 25  45
                       /        /
                      /        /
                     9       42
                    /    \
                  /        \
                7           12
        */

        root = delete(root, 45);
        /*
                             20
                           /     \
                         /         \
                        5           30
                          \         /\
                            \     /    \
                            15  25    42
                          /
                        /
                        9
                    /    \
                  /        \
                7           12
        */
        root = delete(root, 9);
        inorder(root);
        /*
                             20
                           /     \
                         /         \
                        5           30
                          \         /\
                            \     /    \
                            15  25    42
                          /
                        /
                        12
                    /
                  /
                7
        */
        System.out.println("");
    }
}
```

Explanation

The making of a node and traversals are explained in the post Binary Tree in Java: Traversals, Finding Height of Node (https://www.codesdope.com/blog/article/binary-tree-in-java-traversals-finding-height-of-n/). Here, we will focus on the parts related to the binary search tree like inserting a node, deleting a node, searching, etc. Also, the concepts behind a binary search tree are explained in the post Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree).

Search

```
public static Node search(int x, Node n){
    if (n==null || n.getData()==x)
        return n;
    else if(n.getData()>x)
        return search(x, n.getLeftChild());
    else
        return search(x, n.getRightChild());
}
```

`search` is a function to find any element in the tree. To search an element we first visit the root and if the element is not found there, then we compare the element with the data of the root and if the element is greater, then it must lie on the right subtree (property of a BST – All elements greater than the data at the node are on the right subtree), otherwise on the left subtree. We repeat this process until the element is found or a null value is reached (the element is not in the tree).

`if(n==null || n.getData()==x)` → If the null value is reached then the element is not in the tree and if the data at the root is equal to ‘x’ then the element is found.

`else if(x>n.getData())` → The element is greater than the data at the root, so we will search in the right subtree – `search(n.getRightChild(), x)`. Otherwise, on the left subtree – `search(n.getLeftChild(),x)`.

Inserting a new node

```
public static Node insert(Node root, int x){
    if (root == null)
        return new Node(x);
    else if(x>root.getData())
        root.setRightChild(insert(root.getRightChild(),x));
    else
        root.setLeftChild(insert(root.getLeftChild(),x));
    return root;
}
```

Inserting a new node is similar to searching for an element in a tree. We first search for the element and if it is not found at the required place (where it should be) then we just insert a new node at that position. If the element to be inserted is greater than the data at the node, then we insert it in the right subtree – `root.setRightChild(insert(root.getRightChild(), x))`. Suppose, we have to insert a new node to the right child of a node ‘X’. So, we will first create a new node which is returned by `insert(root.getRightChild, x)` and then make the right child of 'X' equal to that node – `root.setRightChild(insert(root.getRightChild, x))`. So after searching, if we reach to a null node, then we just insert a new node there – `if(root==null) → return new Node(x)`.

Thus, the entire `insert` function can be summed up as – If the current node is null then just return a new node. If the data at the current node is smaller than the data to be inserted, then we will change the right child of the current node with the right subtree obtained with the `insert` function. The insert function will either return a new node (in case of a null node) or the modified subtree itself (`return root`).

Delete

As discussed in Binary Search Tree (https://www.codesdope.com/blog/article/binary-search-tree), the code for the deletion is:

```
public static Node delete(Node root, int x){
    if(root==null)
        return null;
    if (x>root.getData())
        root.setRightChild(delete(root.getRightChild(), x));
    else if(x<root.getData())
        root.setLeftChild(delete(root.getLeftChild(), x));
    else
    {
        //No Children
        if(root.getLeftChild()==null && root.getRightChild()==null)
        {
            root = null;
            return null;
        }

        //One Child
        else if(root.getLeftChild()==null || root.getRightChild()==null)
        {
            Node temp;
            if(root.getLeftChild()==null)
                temp = root.getRightChild();
            else
                temp = root.getLeftChild();
            root = null;
            return temp;
        }

        //Two Child
        else
        {
            Node temp = findMinimum(root.getRightChild());
            root.setData(temp.getData());
            root.setRightChild(delete(root.getRightChild(), temp.getData(
        }
    }
    return root;
}
```

If the tree has no children (`if(root.getLeftChild()==null && root.getRightChild()==null)`) – Just delete the node – `root=null`.

If only one child (`(root.getLeftChild()==null || root.getRightChild()==null)`)– Make the parent of the node to be deleted point to the child. – `if(root.getLeftChild()==null)` – Only right child exists. We have to delete this node but we also have to point its parent to its child, so we are storing its child into a temporary variable – `temp = root.getRightChild()` and then deleting the node – `root=null`.

If two children – Find the minimum element of the right subtree – `findMinimum(root.getRightChild())`. Replace the data of the node to be deleted with the data of this node – `root.setData(temp.getData())`. Delete node found by the minimum function – `delete(root.getRightChild(), temp.getData())`.

So, this post was all about the coding implementation of the binary search tree (https://www.codesdope.com/blog/article/binary-search-tree) in Java. You can see the implementation of a BST in C (https://www.codesdope.com/c-introduction) in the post – Binary Search Tree in C (https://www.codesdope.com/blog/article/binary-search-tree-in-c).

Amit Kumar (/blog/author/54322/?author=54322)

Developer and founder of CodesDope.

(https://www.facebook.com/codesdope) (https://www.twitter.com/codesdope) (https://www.linkedin.com/in/amit-kumar-66903395)