Table of Contents

Kth Largest Element BST

Harsh Goyal
Last Updated: May 13, 2022

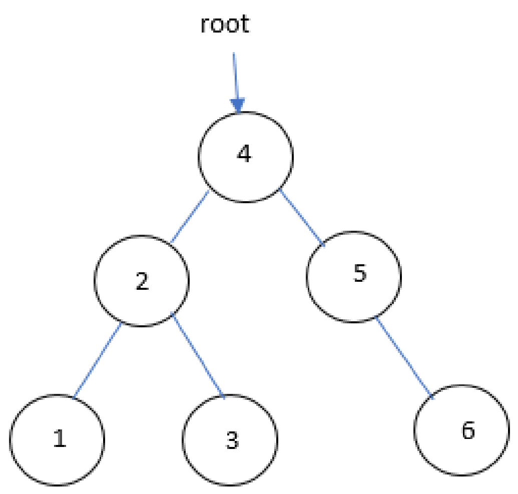
Share :

Introduction-

This blog will discuss the various approaches to find out Kth Largest Element in Binary Search Tree. Before jumping into the problem, let's first understand what exactly a Binary Search Tree is?

A Binary Search Tree is a type of tree data structures where for each and every node, all the nodes in the tree which are on the left of this node which means the left subtree of that node is lesser value than the current of that node has a value greater than the value of current node, along with the fact that both left subtree and right subtree are Binary Search Trees.

Let's say this BST has n nodes. We have to find out Kth Largest element in this BST, where K is always less equal to n.



Brute Force Approach -

Binary search trees have a property that If we do Inorder traversal of the binary search tree, then we get sorted data. We will use this property to solve this given problem. First, we will do an inorder traversal of BST, and we will keep on storing the traversed elements in

Algorithm -

Step 1. Create a recursive function that will accept the root of the tree and a list of integers.

Step 2. The recursive function makes a call to the left subtree

Step 3. Add the current node's value to the list.

Step 4. The recursive function makes a call to the right subtree

Step 5. Return from the recursive function.

Step 6. Go back to the caller function and return the Kth value from the list.

```
import java.util.ArrayList;
import java.util.List;


class TreeNode{
int val;
TreeNode left;
TreeNode right;

TreeNode(int val)
{
this.val = val;
}
}
```

```
public void KthLargestElement(TreeNode root, List<
Integer> list)
{
if(root == null)
        {
return;
}


        // Traverse left subtree
KthLargestElement(root.left, list);

        list.add(root.val);

        // Traverse right subtree
        KthLargestElement(root.right, list);

}

public static void main(String[] args) {


   TreeNode root = new TreeNode(5);
   root.left = new TreeNode(2);
   root.right = new TreeNode(9);
   root.left.left = new TreeNode(1);
   root.left.right = new TreeNode(3);
   root.right.left = new TreeNode(8);
   root.right.right = new TreeNode(12);

   int k = 2;

   BST obj = new BST();
   List<Integer> list = new ArrayList<Integer>();
```

```
        // As list is sorted, so index from beginning will si
    ze - 'K'
        int index = list.size() - k ;

        int KthLargestElement = list.get(index);

        System.out.println(KthLargestElement);

    }


    }
```

Algorithm Complexity of Kth Largest Element:

Time Complexity: O(N)

The complexity of inorder traversal of BST is O(N), as we will be traversing over each node for the inorder traversal.

Space Complexity: O(N)

As we are maintaining a sorted list of 'N' nodes, hence space complexity will be O(N).

Better Approach -

problem without using a list. From the above approach we can deduce that (N - K)th element would be our answer where 'N' is the total number of nodes and 'K' is the user input. Let's find out the size of the tree and then maintain a variable that keeps track of how many elements have been traversed. Once we reach the (N - K)th element, we will return that element, and our work is done.

Algorithm -

Step 1. Create a utility function to find out the size of the tree.

Step 2. Define an instance level variable 'currentCount' and initialize it to -1.

Step 3. Create a recursive function that will take the root node and size of the tree and 'K' index.

Step 4. The recursive function makes a call to the left subtree.

Step 5. Increase 'currentCount'.

Step 6. Check if currentCount == size - K, then return that node.

Step 7. Call the recursive function for the right subtree.

```
class TreeNode{
int val;
TreeNode left;
```

```
TreeNode(int val)
{
this.val = val;
}
}


public class BST {

// We are considering first element as 0th index ele
ment so we have initialized it to -1
int currentCount = -1;


// Utility function to get tree size
public int getTreeSize(TreeNode root)
{
if(root == null)
        {
return 0;
}


return getTreeSize(root.left) + getTreeSize(root.righ
t) + 1;
}

public TreeNode KthLargestElement(TreeNode root,
int n, int k)
{
if(root == null)
        {
return null;
}
```

```
TreeNode left = KthLargestElement(root.left, n, k);

if(left != null)
            {
return left;
            }

currentCount++;

if(currentCount == n - k)
            {
return root;
            }

// Traverse right tree
TreeNode right = KthLargestElement(root.right, n,
k);

if(right != null)
            {
return right;
            }

return null;

}

public static void main(String[] args) {


    TreeNode root = new TreeNode(5);
    root.left = new TreeNode(2);
    root.right = new TreeNode(9);
    root.left.left = new TreeNode(1);
    root.left.right = new TreeNode(3);
```

```
        int k = 2;

        BST obj = new BST();

        int n = obj.getTreeSize(root);

        TreeNode KthLargestNode = obj.KthLargestElem
    ent(root, n, k);

        System.out.println(KthLargestNode.val);

    }


    }
```

Algorithm Complexity of Kth Largest Element:

Time Complexity: O(N)

The complexity of inorder traversal of BST is O(N), as we will be traversing over each node for the inorder traversal.

Space Complexity: O(H)

We are not using any extra space, but recursion is internally using ‘H’ (Height) space, so complexity is O(H)

Optimal

A threaded Binary Tree needs to be used to solve this problem efficiently. Let's understand what Threaded Binary Trees are?

In a Threaded Binary Tree, the nodes will store the in-order predecessor/successor rather than NULL in the left/right child pointers.

So the basic idea of a threaded binary tree is that for the nodes whose right pointer is null, we store the node's in-order successor (if-exists), and for the nodes whose left pointer is null, we store the in-order predecessor of the node(if-exists).

Note that a tree's leftmost and the rightmost child pointer always points to null, as their in-order predecessor and successor do not exist.

You can visit this link for more understanding of Threaded Binary Trees.

Another concept needed to solve this problem is Reverse Morris Traversal, which is the reverse of Morris Traversal.

The idea of Morris's traversal is based on the Threaded Binary Tree. This traversal will create links to the predecessor back to the current node to trace it back to the top of a binary tree. Here we don't need to find a predecessor for every node, and we will be seeing a predecessor of nodes with only a valid left child.

You can visit this link for more understanding of Tree Traversals.

While doing Reverse Morris Traversal, we will keep track of the count, and once that count is equal to 'K', we will

Algorithm -

Step 1. Initialize a counter variable to 0.

Step 2. Run a while loop until the root is not null.

Step 3. If the root has no right child, then increase the counter and check if the counter is equal to K. If yes, we found our desired element and now move to the left subtree of that root.

Step 4. Else, there are 2 scenarios possible -

a) Find the inorder successor of the root. If inorder successor's left child == null

then set root as the left child of its inorder

Successor and move the root node to its right.

b) Else if root node, and it's inorder successor already has threaded link -

1) Set the left pointer of the inorder successor as NULL.

2) Increase counter and check if counter == K, then return that element.

3) Else go to the left child.

```
class TreeNode{
    TreeNode right;

    TreeNode(int val)
    {
    this.val = val;
    }
}


public class BST {

// Traverse from Right to the left in inorder fashion a
nd use Morris Traversal on it
public TreeNode KthLargestElement(TreeNode root,
int k)
{


int counter = 0;
TreeNode KthLargestElement = null;

  while (root != null)
  {
    if (root.right == null)
    {
     // Keep increasing counter with each visiting n
odes
      counter++;
      if (k == counter)
        KthLargestElement = root;

      root = root.left;
    }
    else
    {
```

```
        // This will help to find inorder successor of the
current node
            while(successor.left != null && successor.left !
= root)
            {
                successor = successor.left;
                }

            if (successor.left == null)
            {
             successor.left = root;
             root = root.right;
            }
            else
            {
                    // This block will help to restore the origin
al links of tree
                    successor.left = null;

                    // Keep increasing counter with each visit
ing nodes
                    counter++;
                  if (k == counter)
                  {
                     KthLargestElement = root;
                  }
                  root = root.left;
            }
        }
    }
    return KthLargestElement;

}

public static void main(String[] args) {
    TreeNode root = new TreeNode(5);
    root.left = new TreeNode(2);
    root.right = new TreeNode(9);
    root.left.left = new TreeNode(1);
    root.left.right = new TreeNode(3);
    root.right.left = new TreeNode(8);
    root.right.right = new TreeNode(12);

    int k = 3;

    BST obj = new BST();

    TreeNode KthLargestNode = obj.KthLargestElem
ent(root, k);

    System.out.println(KthLargestNode.val);

}


}
```

Algorithm Complexity of Kth Largest Element:

Time Complexity: O(N)

We are traversing the tree in inorder fashion and modifying the links between the leaf nodes at the same time, which will be done in constant time so total time complexity is O(N) + O(1) = O(N)

Space Complexity: O(1)

Frequently asked questions-

1) What is a Binary Search Tree?

A Binary Search Tree is a type of tree data structure where for every value of a node, all the nodes in the tree which are on the left of this node have a value lesser than the current node's value. All the nodes in the tree which are on the right of this node have a value greater than the current node's value, along with the fact that both the left subtree and right subtree are Binary Search Trees.

2) What is a Threaded Binary Tree?

In a Threaded Binary Tree, the value of nodes will store the in-order predecessor/successor instead of NULL in the left or right or both child pointers. So, in threaded binary tree nodes whose right pointer is null, we store the node's in-order successor (if-exists), and for the nodes whose left pointer is null, we store the in-order predecessor of the node(if-exists).

Note that a tree's leftmost and the rightmost child pointer always points to null, as their in-order predecessor and successor do not exist.

3) What is Morris Traversal?

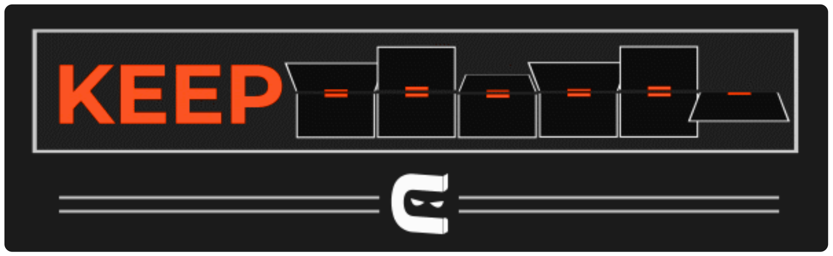
The idea of Morris's traversal is based on the Threaded Binary Tree. This traversal will create links to the predecessor for every node, and we will be finding a predecessor of nodes with only a valid left child.

Key takeaways-

This article discussed the various approaches to finding out Kth Largest Element in Binary Search Tree. We explored the concept of Morris's traversal and found out how we can use this algorithm to solve problems in constant space. Try this problem on Codestudio and you and learn more about this in C++ DSA course.

If you think that this blog helped you, then share it with your friends!.

Until then, All the best for your future endeavours, and Keep Coding.



Was this article helpful ?

0 upvotes

Share this article with friends and colleague :

Comments

Write your thoughts...

Post

Be the first to share what you think