



Kth Largest Element in a BST

binary-tree

binary-search-tree

coding-interview-questions

google-interview-questions

amazon-interview-questions

Difficulty: Medium, **Asked-in:** Amazon, Google

Key takeaway: an excellent problem to learn problem-solving using **inorder** traversal and BST **augmentation** (storing extra information inside BST nodes for solving a problem).

Let's understand the problem!

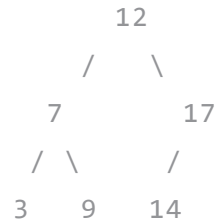
Given the root of a binary search tree and an integer k , write a program to return the k th largest value among all the node values in the given BST.

If the tree is empty, return `INT_MIN`.

Assume that all values in the BST nodes are unique.

Example

Input:



If $k = 2$, then 14 is the 2nd largest element.

If $k = 4$, then 9 is the 4th largest element.

If $k = 5$, then 7 is the 5th largest element.

Important note: before moving on to the solutions, we recommend trying this problem on paper for at least 15 or 30 minutes. Enjoy problem-solving!

Discussed solution approaches

A brute force approach using inorder traversal and extra space

Recursive in-place solution: reverse in-order traversal

Iterative in-place solution: reverse in-order traversal using stack

An efficient solution using BST augmentation

Solution approach 1: A brute force idea using inorder traversal and extra space

Solution idea

As we know, a binary search tree is a sorted version of the binary tree where each node value is greater than all the node values in the left subtree and greater than all the node values in the right sub-tree. So if we perform an in-order traversal, it will produce elements in increasing or sorted order. Think!

So a basic idea would be:

Traverse BST using inorder traversal and store elements in an extra array.

Now n array elements are arranged in increasing order, and we can easily find the kth largest element by accessing the element at **(n - k)th** index from the start. Note: as we know that all elements are unique. So the largest element will be present at (n - 1)th index of the array.

BST node structure

```
class BSTNode
{
    int data
    BSTNode left
    BSTNode right
}
```

Solution pseudocode

```
int kthLargestBST(BSTNode root, int k)
{
    if(root == NULL)
```

```

        return INT_MIN
    }
    vector<int> SortedArray
    inorder(root, SortedArray)
    int n = SortedArray.length
    return SortedArray[n - k]
}

void inorder(BSTNode root, vector<int>& SortedArray)
{
    if(root == NULL)
        return
    inorder(root->left, SortedArray)
    SortedArray.append(root->data)
    inorder(root->right, SortedArray)
}

```

Solution analysis

Time complexity = Time complexity of inorder traversal to store elements in a sorted order + Accessing (n - k)th element from the sorted list = $O(n) + O(1) = O(n)$. Space complexity = **$O(n)$** , we are using an extra array list of size n.

Solution approach 2: A recursive approach using reverse in order traversal

Solution idea

The critical question is: can we solve the problem without using extra space?
can we track the kth largest element using the in-order traversal? Think!

In-order traversal: left subtree -> root -> right subtree. We first access the leftmost node in in-order traversal which stores the smallest value of BST. In other words, it explores all the nodes in increasing order.

Reverse in-order traversal: right subtree -> root -> left subtree. If we perform reverse inorder traversal, it will first explore the rightmost node that stores the largest element of BST. In other words, it explores all the nodes in decreasing order. Think!

So one idea would be: while doing the reverse inorder traversal, we keep decreasing the value of k by one after accessing each node. When the k becomes equal to 0, we stop the traversal because the current node is our kth largest. So we return the value stored in the current node. This way, one could speed up the solution because there is no need to do complete inorder traversal, and one could stop after finding the kth largest element.

Solution pseudocode

```
int kthLargestBST (BSTNode root, int k)
{
    if (root == NULL)
        return INT_MIN
    kthLargestBST(root->right, k)
    k = k - 1
    if (k == 0)
        return root->data
    kthLargestBST(root->left, k)
}
```

Solution analysis

At each step of traversal, we are accessing nodes in decreasing order and decrementing the value of k by 1. So in general, our traversal will return value after the k number of steps. So time complexity = $k * \text{time complexity of each step} = k * O(1) = \mathbf{O(k)}$.

In the worst case, $k = n$, so the worst-case time complexity = $O(n)$

In the best case, $k = 1$, so the best-case time complexity = $O(1)$

Space complexity = $\mathbf{O(h)}$ for recursion call stack, where h is the height of the tree.

Solution approach 3: An iterative approach using reverse in order traversal

Solution idea

This idea is similar to the above approach, but instead of using recursive inorder traversal in reverse order, we are using iterative reverse traversal with the help of a stack. The idea here is: when we pop node from the stack, we decrement the value of k by 1 and check k is equal to 0 or not. If yes, we stop and return the value stored in the current node.

you can explore this blog: [iterative tree traversal using stack](https://www.enjoyalgorithms.com/blog/iterative-tree-traversal-using-stack).

Solution pseudocode

```
int kthLargestBST(BSTNode root, int k)
{
    if(root == NULL)
        return INT_MIN
    Stack<BSTNode> BSTStack
    BSTNode curr = root
    while (BSTStack.empty() == false || curr != NULL)
    {
        if (curr != NULL)
        {
            BSTStack.push(curr)
            curr = curr->right
        }
        else
        {
            curr = BSTStack.pop()
            k = k - 1
            if (k == 0)
                return curr->data
            curr = curr->left
        }
    }
}
```

Solution analysis

As mentioned in the analysis of the previous approach, the iterative traversal will also access nodes in decreasing order by decrementing the value of k. So iteration will stop after k number of steps. Time complexity = $k * O(1) = O(k)$.

The worst-case time complexity = $O(n)$

The best-case time complexity = $O(1)$

Space complexity = $O(h)$ for recursion call stack, where h is the height of the tree.

Solution approach 4: An efficient solution using augmented BST

Solution idea

The critical question is: what if the BST is often modified using insert or delete operations, and we need to find the k th largest element frequently? Can we optimize the time complexity? Let's think!

We know from the BST property that the value stored in all the nodes in the right sub-tree of any node is larger than the value stored in that node. So for the given root in the BST, if we know the count of elements in its right sub-tree, we can easily define the rank or order of the root node. Suppose there are 5 elements in the right subtree, then the root element would be the 6th largest element in the tree.

Solution steps

Step 1: we define a new structure of the BST node where we use an extra parameter **rightCount** to store the node count in the right sub-tree.


```
class BSTNode
{
    int data
    BSTNode left
    BSTNode right
    int rightCount
    BSTNode(int value)
    {
        data = value
        left = right = NULL
        rightCount = 0
    }
}
```

Step 2: now we build a BST of the given input by inserting each element one by one. During the insertion, we can keep track of a number of elements in the right subtree and update the rightCount for each node.

Pseudocode Implementation: inserting an element by updating the rightCount

```
BSTNode insertBST(BSTNode root, int value)
{
    if (root == NULL)
        return BSTNode(value)

    if (value > root->data)
    {
```

```
        root->right = insertBST(root->right, value)
        root->rightCount = root->rightCount + 1
    }

    else if (value < root->data)
        root->left = insertBST(root->left, value)

    return root
}
```

Step 3: now we define a function **kthLargestBST(root, k)** to return the kth largest element.

We start from the root node by comparing k with the order or rank of the root node. What would be the rank of the root node? The answer is simple: If there are root-> rightCount nodes in the right subtree, then the root is (root-> rightCount + 1)th largest element in the tree. Think!

If (k = root-> rightCount + 1): the root->data is the required kth maximum element and we return this value as an output.

If (k < root-> rightCount + 1): the kth largest would be present in the right sub-tree because the rank of the root is greater than the kth largest element. So kth largest element of the overall tree would also be the kth largest element of the right subtree. So we search the kth largest element recursively in the right subtree i.e. kthLargestBST(root->right, k).

If (k > root-> rightCount + 1): the kth largest would be present in the left sub-tree because the rank of the root is smaller than the kth largest

element. So we search the kth largest element recursively in the right subtree i.e. `kthLargestBST(root->left, k - rightCount - 1)`. Note: Why are we passing **k - rightCount - 1** as the input parameter? The answer is simple: we are searching the kth largest element in the left subtree, and $(\text{rightCount} + 1)$ number of elements are already greater than all the elements in the left subtree. So the rank of the kth largest element in the left subtree would be $k - \text{rightCount} - 1$. Think!

Solution pseudocode

Recursive implementation of `kthLargestBST(root, k)`

```
int kthLargestBST(BSTNode root, int k)
{
    if (root == NULL)
        return INT_MIN
    if (k == root->rightCount + 1)
        return root->data
    if (k < root->rightCount + 1)
        return kthLargestBST(root->right, k)
    else
        return kthLargestBST(root->left, k - root->rightCount - 1)
}
```

Iterative implementation of `kthLargestBST(root, k)`

```
int kthLargestBST(BSTNode root, int k)
{
```

```
if (root == NULL)
    return INT_MIN
BSTNode curr = root
while(curr != NULL)
{
    if (k == curr->rightCount + 1)
        return curr->data

    else if (k < curr->rightCount + 1)
        curr = curr->right

    else
    {
        k = k - curr->rightCount - 1
        curr = curr->left
    }
}
```

Solution analysis

The time complexity of insert and delete operations into a BST is $O(h)$, where h is the height of the binary tree. h is $O(\log n)$ for the balanced tree and $O(n)$ for a skewed tree.

Once our BST is ready with the rightCount of each node, we are searching the kth largest element by going left or right at each step based on the comparison. In other, we reduce the input size by the size of the left or right

subtree at each step. So in the worst case, we will be traversing the tree's height and doing an $O(1)$ operation at each step. So time complexity of finding kth largest = $O(h)$. Think!

Space complexity = storing rightCount inside each node = $O(n)$. For recursive implementation, we also need to also consider the size of the recursion call stack which is $O(h)$. But overall space complexity would be $O(n)$ in both implementations. Think!

Important note: we recommend transforming the above pseudo-codes into a favorite programming language (C, C++, Java, Python, etc.) and verifying all the test cases. Enjoy programming!

Critical ideas to think!

How can we modify the above approaches to find Kth smallest in a BST?

Design an algorithm to find the kth largest in a binary tree.

What is the worst and best-case scenario of space complexity in the above approaches?

Instead of reverse inorder traversal, can we solve it using in-order traversal?

How can we solve this problem using Morris traversal?

Can we think of solving this problem using max or minheap?

What would be the time complexity of building BST of n nodes?

If BST is balanced, the time complexity would be $O(\log n)$. So in the scenario of frequent insert or delete operations, BST may get imbalanced. How can we modify the above insert operation to ensure a balance BST? We should explore the idea of AVL or a red-black tree.

Comparison of time and space complexities

Inorder traversal and extra space: Time = $O(n)$, Space = $O(n)$

Reverse in-order traversal (recursive): Time = $O(k)$, Space = $O(h)$

Reverse in-order traversal (iterative): Time = $O(k)$, Space = $O(h)$

Using BST augmentation: Time = $O(h)$, Space = $O(n)$

Suggested Problems to Solve


Kth smallest element in a BST

Kth largest in an unsorted array

Merge two BSTs with constant extra space

Convert Sorted Array to Binary Search Tree

Please write in the message below if you find anything incorrect, or you want to share more insight, or you know some different approaches to solve this problem. Enjoy learning, Enjoy algorithms!

Author
Shubham Gautam 

Reviewer
EnjoyAlgorithms Team 

Share Feedback

More from EnjoyAlgorithms

Reverse a linked list

Write a program to reverse a linked list. A head pointer of a linked list is given and our task to reverse the entire list

Triplet with Zero Sum

Given an array $X[]$ of distinct elements, write a program to find all the unique triplets in the array whose sum is equal to zero. For example, suppose such triplets in the...

s... resulted list is traversed it looks like w...

[Read More](#)[Read More](#)

Find the next greater element for every element in an array

Given an array, find the next greater element for every element in the array. The next greater element for an element is the first greater element on the right side of...

[Read More](#)

Segment Tree Implementation Part 1: Build and Range Query Operations

A Segment Tree is a data structure used to answer multiple range queries on an array efficiently. Also, it allows us to modify the array by replacing an element or...

[Read More](#)

Minimum number of Jumps to reach End

An array of non-negative integers is given and the aim is to reach the last index in the minimum number of jumps. You are initially positioned at the first index of the array...

[Read More](#)

Find Product of Array Except Self

Given an array $X[]$ of n integers, write a program to find an array $product[]$ such that $product[i]$ is equal to the product of all the elements of $X[]$ except $X[i]$. We need t...

[Read More](#)

Our weekly newsletter

Subscribe to get free weekly content on data structure and algorithms, machine learning, system design, oops design and mathematics.

Subscribe

Home

Coding Interview

OOPS Design

Machine Learning

System Design

Latest Content

Follow us on:



© 2020 EnjoyAlgorithms Inc.

All rights reserved.