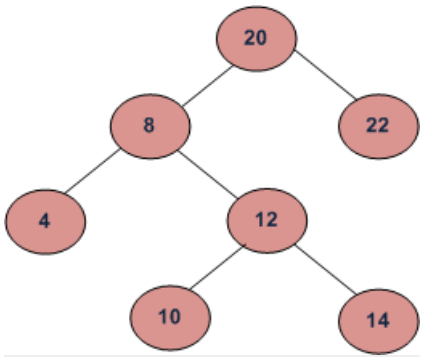
K'th Largest Element in BST when modification to BST is not allowed

Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree.

For example, in the following BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



We have discussed two methods in this post. The method 1 requires O(n) time. The method 2 takes O(h) time where h is height of BST, but requires augmenting the BST (storing count of nodes in left subtree with every node).

Can we find k'th largest element in better than O(n) time and no augmentation?

We strongly recommend to minimize your browser and try this yourself first.

In this post, a method is discussed that takes O(h + k) time. This method doesn't require any change to BST.

The idea is to do reverse inorder traversal of BST. The reverse inorder traversal traverses all nodes in decreasing order. While doing the traversal, we keep track of count of nodes visited so far. When the count becomes equal to k, we stop the traversal and print the key.

```
// C++ program to find k'th largest element in BST
#include<iostream>
using namespace std;

struct Node
{
    int key;
    Node *left, *right;
};

// A utility function to create a new BST node
Node *newNode(int item)
{
    Node *temp = new Node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

// A function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
```

```
    if (c == k)
    {
        cout << "K'th largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
    kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    kthLargestUtil(root, k, c);
}

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
              50
           /     \
          30      70
         /  \    /  \
       20   40  60   80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    int c = 0;
    for (int k=1; k<=7; k++)
        kthLargest(root, k);

    return 0;
}
```

```
K'th largest element is 80
K'th largest element is 70
K'th largest element is 60
K'th largest element is 50
K'th largest element is 40
K'th largest element is 30
K'th largest element is 20
```

Time complexity: The code first traverses down to the rightmost node which takes O(h) time, then traverses k elements in O(k) time.

Therefore overall time complexity is O(h + k).