CSCI 2270 – Data Structures

Recitation 7, Fall 2021

Binary Search Trees

1. Why Binary Search Trees

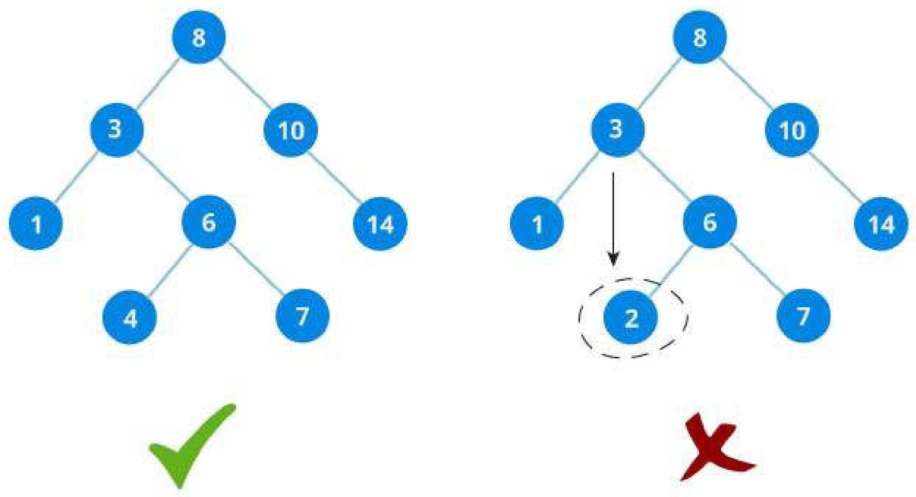
In the previous section, we discussed different tree representations. In each of them, no restrictions were imposed on the data of every node relative to its location in the Tree. This results in a worst-case time-complexity of O(n) for search operations.

In this section, we will discuss another version of binary trees called Binary Search Trees (BST). As the name suggests, BSTs are used for searching efficiently. BST imposes restrictions on the on the node's data relative to its location. As a result, it reduces the worst-case runtime for search from O(n) to O(log n) (in case of balanced BSTs).

Property of a Binary Search Tree (BST)

In BSTs, the data stored in a left sub-tree of a node is less than or equal to the node's data. And the data stored in the right sub-tree of a node is greater than node's data. This is the property of a Binary Search Tree.

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.



CSCI 2270 – Data Structures

Recitation 7, Fall 2021

Binary Search Trees

Declaration of a Binary Search Tree

The declaration of a Binary Search Tree is the same as a regular binary tree. The restriction imposed is on the data relative to its location in the tree and not the structure of the tree.

2. Operations on Binary Search Trees

Following are the main operations that are supported by binary search trees:

Main Operations:

- Find / Find Minimum / Find Maximum in binary search trees.
- Inserting an element in binary search trees.
- Deleting an element from binary search trees

Auxiliary Operations:

- Checking whether the given tree is a binary search tree or not.
- Finding kth smallest element in tree.
- Sorting the elements of binary search tree and many more.

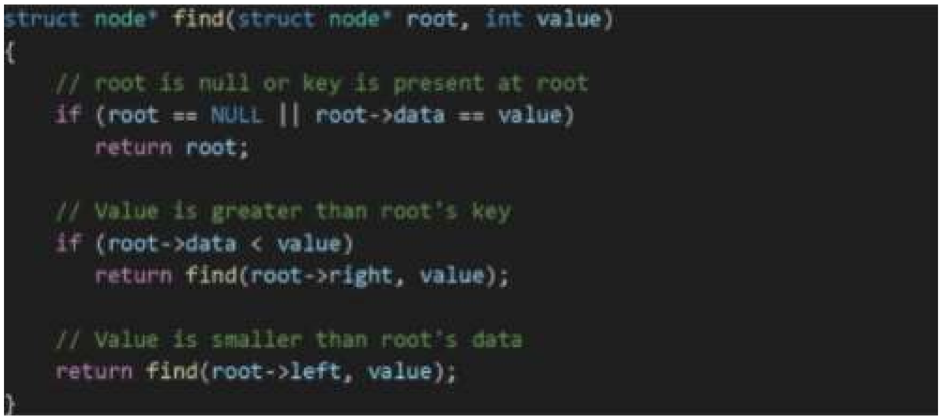
Tip: Since root data is always between left subtree and right subtree data, performing in-order traversal on binary search tree produces a sorted list.

3. Finding an element in Binary Search Trees

We take advantage of the BST property to find if an element is present in the BST. The algorithm is as follows.

- Start at the root, and recursively traverse if the element to be found is less than / greater than the current node's element.
- If its less than the current node's element, traverse to its left sub-tree.
- Otherwise, traverse to its right sub-tree.
- Terminate if the traversal leads to a nullptr, or if the element is found.

The code is given below.



```
struct node* find(struct node* root, int value)
{
    // root is null or key is present at root
    if (root == NULL || root->data == value)
       return root;

    // Value is greater than root's key
    if (root->data < value)
       return find(root->right, value);

    // Value is smaller than root's data
    return find(root->left, value);
}
```

4. Inserting an element into Binary Search Tree

Insert operation on a BST is similar to find operation. The algorithm is as follows.

- Create a new node. Now, traverse the BST to find a suitable location such that post insertion, BST property holds.
- Start at the root node and traverse its left sub-tree if the new node's element is less than the element stored in root and traverse its right sub-tree otherwise.
- Recursively traverse the tree till the left sub-tree or the right sub-tree is empty.
- **The new node is inserted here as a leaf node.**

5. Deleting an element from Binary Search Tree

On deletion, we must ensure the BST property is not violated. Below, we list out all the possible cases of deletions on a BST and how the BST property is preserved post deletion.

Case 1: Deleting a leaf node.

If the node to be deleted is a leaf node: Delete the required node and return NULL to its parent. Thus, the deleted node's parent now points to NULL.

CSCI 2270 – Data Structures

Recitation 7, Fall 2021

Binary Search Trees

Case 2: Deleting a node with one child.

If the node to be deleted has one child: the deleted node's child becomes the child of the deleted node's parent (the grandchild becomes the child of the deleted node's parent)

If the node to be deleted has a left child: the deleted node's left child will be the left-child of the parent.

If the node to be deleted has a right child: the deleted node's right child will now be the right-child of the parent.

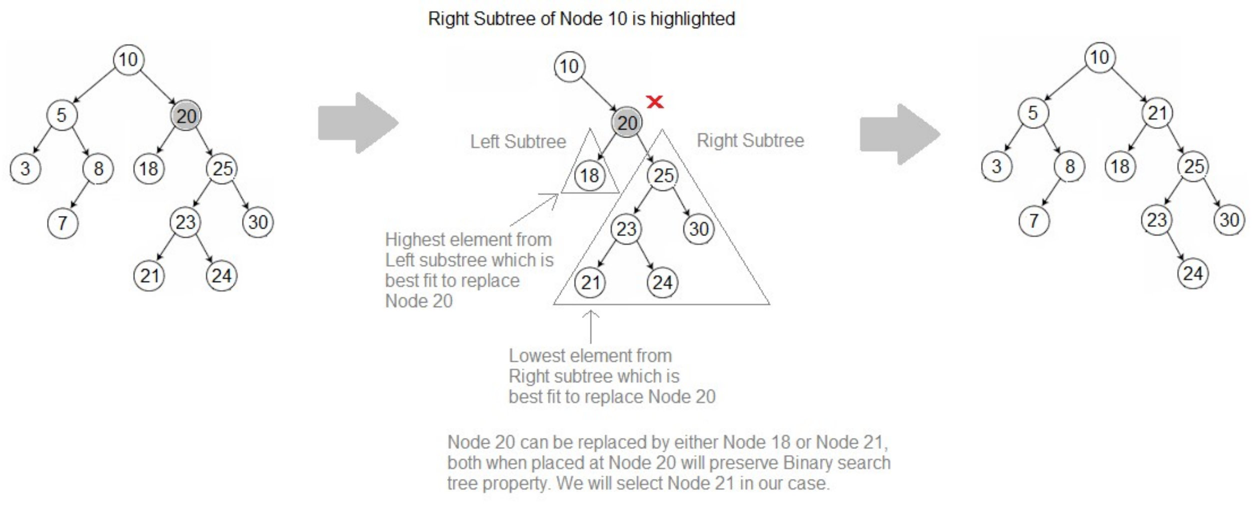
Case 3: Deleting a node with two children.

This is a tricky case as the deleted node has two children, and any of these two children can be sub-trees. We must find a replacement node such that post deletion, the BST property is preserved.

How do we find the replacement node?

Recursively search for the largest of the left sub-tree or the smallest of the right sub-tree. I.e., If the node to be deleted is N, find the largest node in the left sub tree of N or the smallest node in the right subtree of N. These are the two candidates that can replace the deleted node.

For example, consider the following tree and suppose we need to delete the node with data 20.


Right Subtree of Node 10 is highlighted
Left Subtree
Right Subtree
Highest element from Left substree which is best fit to replace Node 20
Lowest element from Right subtree which is best fit to replace Node 20

Node 20 can be replaced by either Node 18 or Node 21, both when placed at Node 20 will preserve Binary search tree property. We will select Node 21 in our case.

6. Time Complexity of BST Operations

What is a Balanced BST and why do we need them?

Each of the BST operation we have seen so far - doing lookup, insertion and deletion, the cost of our algorithms is proportional to the height of the tree. Height of a tree is same as height of the root. Height of a node is the longest path from the node to any leaf.

If the BST is "balanced", its height (h) is utmost $\log_2 n \Rightarrow$ all operations run in O(log n) time. Let us see how a "balanced" BST with *n* nodes has a maximum order of log(n) levels!

Consider an arbitrary BST of the height h. We then count nodes on each level, starting with the root, assuming each level has the maximum number of nodes. The total possible number of nodes is given by:

$n = 1 + 2 + 4 + ... + 2^{h-1} + 2^h = 2^{h+1} - 1$

Solving this with respect to h, we obtain,

$h = O(\log n)$

where the big-O notation hides some superfluous details.

If the data is randomly distributed, it is highly likely that the tree is "almost" balanced. However, if the data already has a pattern, then just naïve insertion into a BST will result in unbalanced trees. Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order.

```
1    1     1       1
     \     \       \
      2     2       2
             \       \
              3       3
                       \
                        4
```

We do not get a branching tree, but a linear tree. All the left subtrees are empty. This tree structure results in every operation to take O(n).

Exercise

You can always use helper functions to perform recursion. We are using them in this exercise too.

A. Silver Badge Problem

1. Download the **zip** files from Canvas, it has header and implementation files on BST.
2. Given a range, **removeRange()** function deletes all keys in the BST which are in the given range. Complete the TODOs in the **deleteNode()** function in **BST.cpp**.

B. Gold Badge Problem

1. Complete the **isValidBST()** function in the **BST.cpp** file**.**
2. Given a root of a binary tree, **isValidBST()** should return **true** if the tree is a valid BST, else return **false**.