CSCI 2270– Data Structures

Recitation 7

Binary Search Trees

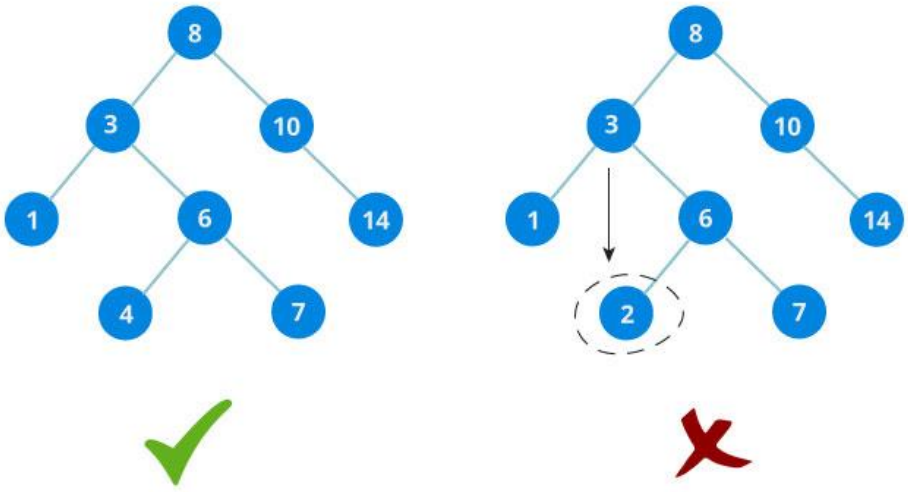
1.Why Binary Search Trees

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

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.

CSCI 2270– Data Structures

Recitation 7

Binary Search Trees

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. 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,

$$h = O(\log n)$$

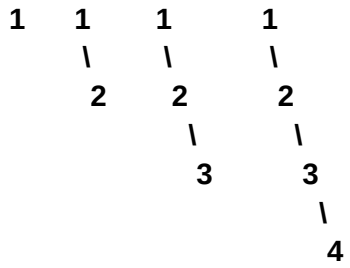
CSCI 2270– Data Structures

Recitation 7

Binary Search Trees

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.



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. Accept the assignment on Github and clone it, it has header and implementation files on BST.
2. Implement the function findKthSmallest and the corresponding helper function.This function will find the k^{th} smallest element in the tree
3. Implement the destructor

B. Gold Badge Problem (Optional)

1. Complete the rangePrint function and the corresponding helper function. This function will accept two arguments: lowerbound and upper bound. It will print all the the values from the BST satisfying the condition lower bound <= node->key <=upperbound