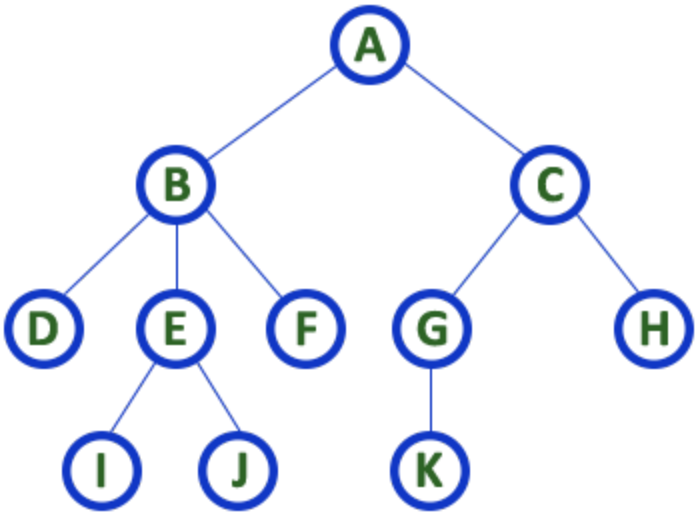
CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

Objectives

1. Tree Data Structure
2. Binary Trees
3. Binary Tree Traversals
    a. Preorder traversal
    b. Inorder traversal
    c. Postorder traversal
4. Why Binary Search Trees
5. Operations on Binary Search Trees
6. Finding an element in Binary search tree
7. Inserting an element in Binary search tree
8. Deleting an element from Binary search tree
9. Exercise



CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

1. Tree Data Structure

Technical Definition: A tree is a collection of entities called nodes. Nodes are connected by edges. Each node contains a value or data, and it may or may not have a child node.
Tree is an example of non-linear data structures. A structure is a way of representing the hierarchical nature of a structure in graphical form.

In simple terms, a tree is a data structure that is similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.

In trees ADT(Abstract data type), order of elements is not important. If we need ordering information linear data structures like linked lists, stacks, queues, etc can be used.

- The root of a tree is the node with no parents. There can be at most one root node in a tree.
- An edge refers to the link from parent to child.
- A node with no children is called leaf node.
- Children of the same parent are called siblings.
- A node p is an ancestor of node q if there exists a path from root to q and p appears on the path.
- Set of all nodes at a given depth is called level of the tree.The root node is at level 0.

2. Binary Trees

A tree is called a binary tree in which each node has at the most two children, which are referred to as the left child and the right child i.e each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualise a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

Types of binary trees

1. Full Binary Tree - A full binary tree is a binary tree in which all nodes except leaves have two children.
2. Complete Binary Tree - A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

3. Perfect Binary Tree - A perfect binary tree is a binary tree in which all internal nodes have two children and all leaves are at same level.

Example code to create a Binary Tree:

```
struct Node
{
  int data;
  Node *left;
  Node *right;
};
```

Applications of trees data structure

1. Expression trees are used in compilers.
2. Manipulate hierarchical data.
3. Used to implement search algorithms.

3. Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them. The process of visiting all nodes of a tree is called tree traversal. Each node is processed only once but it may be visited more than once. As we have seen already that in linear data structures like linked lists, stacks and queues the elements are visited in sequential order. But, in tree structures, there are many different ways.

Traversal possibilities

Starting at the root of a binary tree, there are 3 main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node, traversing to the left child node, and traversing to the right child node. This process can be easily defined through recursion.

1. LDR: Process left subtree, process the current node data and then process the right subtree
2. LRD: Process left subtree, process right subtree, process current node data.
3. DLR: Process current node data, process left subtree, process right subtree.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

4. DRL: Process current node data, process right subtree, process left subtree.
5. RDL: Process right subtree, process current node data,, process left subtree.

6. RLD: Process right subtree, process current node data, process left subtree.

Classifying the traversals

The sequence in which these nodes are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node(D) and if D comes in the middle it does not matter whether L is on the left side of D or R is on the left side of D. Similarly, it doesn't matter whether L is on the right side of D or R is on the right side of D. Due to this, the total possibilities are reduced to 3 and these are:

1. Preorder Traversal (DLR)
2. Inorder Traversal (LDR)
3. PostOrder Traversal (LRD)

There is another traversal method which does not depend on above orders and it is :
Level Order Traversal. (We will cover this later.)

Note - The below traversal techniques can be done using both using recursion and iterative techniques. However for this class we will only focus on recursive techniques, since it is much simpler and easy to understand.

PreOrder Traversal

In preorder traversal, each node is processed before(pre) either of its subtrees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree. Processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in reverse order.

Pre order Traversal is defined as follows.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

1. Visit the root.
2. Traverse the left subtree in Preorder.
3. Traverse the right subtree in Preorder.

```
void preorder(Node *root)
{
	if(root != NULL)
	{
		print(root->data);
		preorder(root->left);
		preorder(root->right);
	}
}
```

InOrder Traversal

In inorder traversal the root is visited between the subtrees, inorder traversal is defined as follows.

1. Traverse the left subtree in Inorder.
2. Visit the root.
3. Traverse the right subtree in Inorder.

```
void inorder(int *root)
{
	if(root != NULL)
	{
		inorder(root->left);
		print(root->data);
		inorder(root->right);
	}
}
```

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

PostOrder Traversal

In post order traversal, the root is visited after both subtrees. Postorder traversal is defined as follows.

1. Traverse the left subtree in PostOrder.
2. Traverse the right subtree in PostOrder.
3. Visit the root.

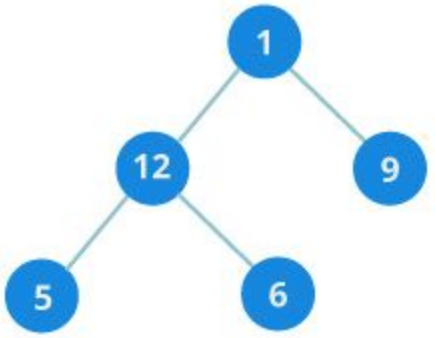
```
void postorder(int *root)
{
      if(root == NULL){
         return;
      }

      postorder(root->left);
      postorder(root->right);
      print(root->data);
}
```

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

Example tree with PreOrder, InOrder and PostOrder Traversals:



Inorder (Left, Root, Right) : 5, 12, 6, 1, 9
Preorder (Root, Left, Right) : 1, 12, 5, 6, 9
Postorder (Left, Right, Root) : 5, 6, 12, 9, 1

Complexity Analysis of Binary Tree Traversals

For all of these traversals - whether done recursively or iteratively we visit every node in the binary tree. That means that we'll get a runtime complexity of O(n) - where n is the number of nodes in the binary tree.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

4. Why Binary Search Trees

In the previous recitation we discussed different tree representations and in all of them we did not impose any restriction on the node data. As a result, to search for an element we need to check both in left subtree and in right subtree. Due to this, the worst case complexity of search operation is O(n).

In this class, we will discuss another version of binary trees. Binary Search Trees(BSTs). As the name suggests, the main use of this representation is for searching. In this representation, we impose restrictions on the kind of data a node can contain. As a result, it reduces the worst-case runtime for search from O(n) to O(logn) (in case of balanced BSTs).

Binary search tree property

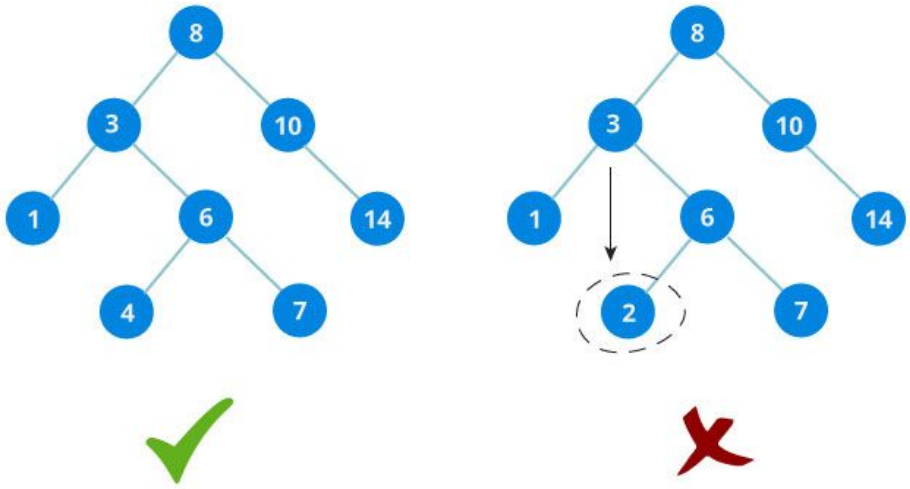
In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that this property should be satisfied at every node in the tree.

- 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 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs



Binary Search Tree Declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure.

5. 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

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

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 inorder traversal on binary search tree produces a sorted list.

6. Finding an element in Binary Search Trees

Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is the same as nodes data then we return the current node. If the data we are searching is less than nodes data, then search the left subtree of the current node; otherwise, search the right subtree of the current node. If the data is not present, we end up in a ***null*** link.

```
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);
}
```

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

7. Inserting an element into Binary Search Tree

For inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further.

We then insert the node as a left or right child of the leaf node based on whether the node is lesser or greater than the leaf node. We note that **a new node is always inserted as a leaf node**. If a tree is empty, this new node becomes the root.

A recursive algorithm for inserting a node into a BST is summarized as follows:
1. Start from root.
2. Compare the element that is getting inserted with the root, if it is less than root, then recurse in the left, else recurse in the right side.
3. After reaching the end, just insert that node at left, if less than current, else right.

8. Deleting an element from Binary Search Tree

Deleting a node could affect all subtrees of that node. So we need to be careful about deleting nodes from a tree. We should delete an element without violating the Binary Search Tree property! The best way to deal with deletion seems to be considering special cases. We

Case 1: Deleting a leaf node:

If the node to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointed NULL.

Case 2: Deleting a node with one child:

If the node to be deleted has one child: In this case we need to send the current nodes child to its parent.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

If the node to be deleted has a left child - Send that left child to the parent of the deleted node. The deleted node's parent will adopt its left child.

If the node to be deleted has a right child - Send that right child to the parent of the deleted node. The deleted node's parent will adopt its right child.

Case 3: Deleting a node with two children:

This is a tricky case as we need to deal with two subtrees!
First we find a replacement node for the node to be deleted. We need to do this while maintaining the BST order property. Then we copy the data of the replacement node to the node to be deleted and delete the replacement node. (Now the deletion may recursively reduce to either Case 1, or Case 2)

How do we find the replacement node?
Search the data from the subtree of node to be deleted and find the node whose data when placed at the place of node to be deleted will keep the Binary Search Tree property(key in each node must be greater than all keys stored in the left subtree, and smaller than all keys in right subtree) intact.

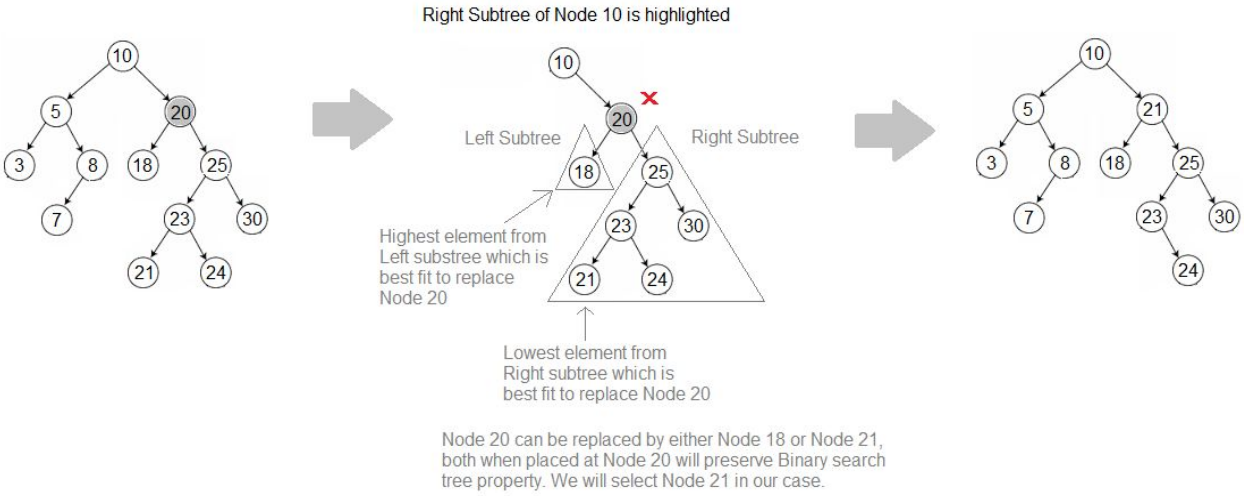
If the node to be deleted is N, find the largest node in the left sub tree of N or the smallest node in the right subtree of N. These are two candidates that can replace the node to be deleted without losing the order property.

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

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs



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 built in a "balanced" fashion, it maintains h = O(log n) ⇒ all operations run in O(log n) time. Let's 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 that 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,

CSCI 2270 – Data Structures

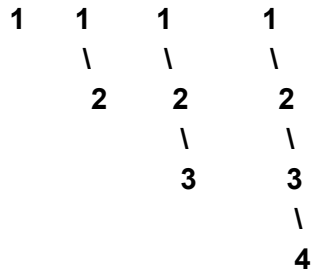
Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

h = O(log n)

where the big-O notation hides some superfluous details.

If the data is randomly distributed, then we can expect that a tree can be “almost” balanced, or there is a good probability that it would be. 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 of the left subtrees are empty. Because of this behavior, *in the worst case,* each of the operations takes time O(*n*). From the perspective of the worst case, we might as well be using a linked list and linear search. Therefore, great care needs to be taken in order to keep the tree as balanced as possible.

CSCI 2270 – Data Structures

Recitation 4, CSCI Summer 2020

Trees, Traversal techniques and BSTs

9. 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 Moodle, 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**.