Lecture 15

Binary Trees
Mainly Binary Search Tree

Last time:

We looked at

1. Ordered Array: Using binary search, we can search for an item pretty quickly (running time complexity of __________). But, it has issues with insertion and deletion. Whenever we need to insert a new object into an ordered array, we first need to find where it should go and then shift all of the objects that are greater than the object up by one space in the array to make a room for the new object. A similar operation of shifting elements is required for a deletion.

   If you are going to have a lot of insertions and deletions, an ordered array is not a good option.

2. Linked List: As a possible alternative to this issue of an ordered array, we looked at Linked List. Insertions and deletions are quick on a linked list once the place to insert or the node to delete is found. It requires us to change only a few references (running time complexity of __________). Unfortunately, searching for a specific item in a linked list is slow because we MUST start from the beginning of the list (Singly Linked List) and check each element until we find it (or not). Remember! Linked List is a sequential access data structure.

Here is today's question!

> ***What if we can have a data structure that has the advantages of both ordered array and linked list?***

Let's embark on finding an answer to this question.

Why would anyone want to use a tree?

The answer is that ***it combines the advantages of two other data structures:***

- Ordered Array: Search is quick using binary search
- Linked List: Insert and delete items quickly (in a sense that there is *no need to shift elements.*)

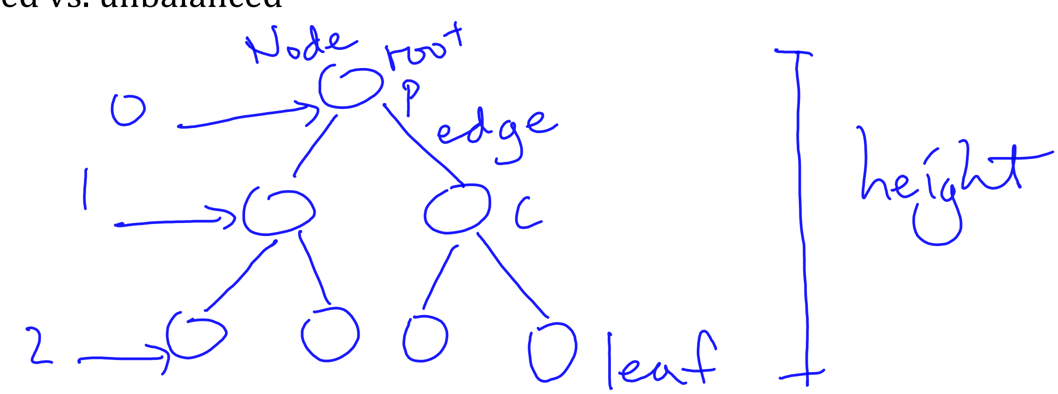
Step 1: Conceptual View

A tree is an extension of the concept of linked data structures.

It has nodes with more than one self-referenced field.

Tree Terminology:

- **Root**: The node at the top of the tree.
- **Parent**: When any node (except the root) has exactly one edge running upward to another node. The node above is called parent of the node.
- **Child**: Any node may have one or more lines running downward to other nodes. These nodes below the given node called its children.
- **Leaf**: A node that has no children is called a leaf. There can be only one root in a tree but there can be many leaves.
- **Level (Height)**: the level of a particular node refers to how many generations the node is from the root. The root is at level 0 and its children are at level 1 and so on.
- Balanced vs. unbalanced



A binary tree has nodes that contain a data element, a *left* reference, and a *right* reference. In other words, every node in a binary tree can have at most two children.

A full binary tree is a binary tree where each node has exactly zero or two children.

A complete binary tree is a binary tree that is completely filled (from left to right) with the possible exception of the bottom level.

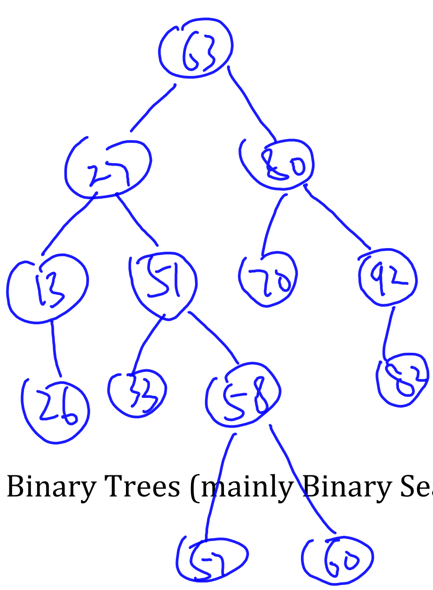
Binary Search Tree

The first usage of a binary tree we will be discussing is binary search tree.

> **The defining characteristic, or the ordering invariant**, of binary search tree: *At any node with a key value, k, in a binary search tree, all keys of the elements in the left sub-tree must be less than k, while all keys of the elements in the right sub-tree must be greater than k. (Meaning no duplicates are allowed.)*

Exercise 1
Let's build a binary search tree given the following values.

63, 27, 80, 51, 70, 92, 13, 33, 58, 26, 60, 57, 82



Step 2: Implementation View

Major operations we want to have in our binary search tree are:

- Insertion
- Searching
- Deletion
- Traversal

First, let's have an interface for **BST**.

```
// Binary Search Tree interface
public interface BSTInterface {

    /**
     * search a value in the tree
     * @param key key value to search
     * @return boolean value indicating success or failure
     */
     boolean find(int key);

    /**
     * insert a new element into the tree
     * @param key key of the element
     * @param value value of the element
     */
     void insert(int key, double value);

    /**
     * delete a value from the tree
     * @param key key of the element to be deleted
     */
     void delete(int key);

    /**
     * traverse the tree and print values
     */
     void traverse();

}
```

Time to sketch our **BST class**.

```
public class BST implements BSTInterface {
    private Node root;

    @Override
    public boolean find(int key) {
        //TODO implement this.
    }

    @Override
    public void insert(int key, double value) {
        //TODO implement this.
    }

    @Override
    public void delete(int key) {
        //TODO implement this.
    }

    @Override
    public void traverse() {
        //TODO implement this.
    }
}
```

Of course, we need **Node class** as a private static nested class within the BST class.

```
private static class Node {
    private int key;
    private double value;
    private Node left, right;

    public Node(int key, double value) {
        this.key = key;
        this.value = value;
        left = right = null;
    }
}
```

Searching for a Node with a given key. (Iterative approach)

```
public boolean find(int key) {
    if(root==null) return ___false____;  // tree is empty
    Node current = __root____;

    // while no match
    while(current.key != key) {
        if(current.key < key)
           current = current.___right_______;
        else
           current = current.___left_______;
        if(current == ___null_______) // not found
           return ___false_______;
    }

    return __true______;  // found
}
```

Efficiency of searching

Let's take a look again at the example we used before and search for 58.
63, 27, 80, 51, 70, 92, 13, 33, 58, 26, 60, 57, 82

Now, let's find 57.

What do we need to look at to figure out the running time complexity?
Height of the tree, that is ______________________. ***(BUT PROBABLY!)***

Inserting a new Node

To insert a node, the first thing we need to do is to find the place to insert it. This is basically the same process we did in search but we need a few more steps to successfully take care of inserting a new node.

```
public void insert(int key, double value) {
    Node newNode = new Node(key, value);
    if(root == null)  // tree is empty
        root = ___newNode___;
    else {
        Node current = _root_______;
        Node parent;

        while(true) {
            if(key == current.key) ___return________;
            parent = ___curr____________;
             if(key < current.____key______) {
                 current = current.____left____;
                 if(current == ___null______) {
                     parent.___left______ = __newNode___;
                     return;
                 }
            } else {
                 current = current.____right____;
                 if(current == ___null_______) {
                     parent.___right____ = __newNode___;
                     return;
                 }
            } // end of go right or left
        } // end of while loop
    } // end of empty or not
} // end of insert method
```

Exercise 2
Draw a binary search tree for 84, 41, 96, 24, 50, 13, 98 and insert 37 in the tree.

Efficiency of Inserting
Inserting a new item in BST requires searching, ________________, ***(once again probably!)*** and a small amount of time to connect the new node with its parent.

Deleting a Node
Deletion is more complicated than other operations.
To delete, we start by finding the node that needs to be deleted using the same process we used for searching. After the search, **there are four cases to consider:**

- CASE 1: The node is not in the tree.
- CASE 2: The node is a leaf.
- CASE 3: The node has one child.
- CASE 4: The node has two children.

Find the node to delete first and take care of **case 1: Not found**

```
Node current = __root______;  // keep track of current
Node parent = ___root______;  // keep track of parent
___isLeftChild = true_______;

while(current.key != key) {
     parent = current;
     if(key < current.__key___) {
          __parent = curr__________;
          current = current.___left____;   isLC = true;
     } else {
          __parent = curr__________;
          current = current.___right___;   isLC = false;
     }
     if(current == ___null______)
         return;
}
```

Case 2: a leaf

Simply set it to null!

```
// in case of leaf (no children)
if(current.__left______ == null && current.__right____ == null) {
     if(current == root)
          ___root____ = null;
     else if(___isLC_________)
          parent.____left_____ = null;
     else
          parent.___right_____ = null;
}
```

Case 3: a node with one child

In a sense, this process is pretty simple. We can simply push the subtree of the node to be deleted up closer to the root. *In fact, this process is identical to deleting a node from a linked list.*

```
else if(current.____right____ == null) { // no right child
    if(current == root)
        root = current.____left____;
    else if(____isLC____)
        parent.___left___ = current.___left___;
    else
        parent.___right___ = current.___left___;
} else if(current.___left___ == null) {  // no left child
    if(current == root)
        root = current.____right____;
    else if(___isLC___)
        parent.___left___ = current.___right___;
    else
        parent.___right___ = current.___right___;
}
```

Case 4: a node with two children

Now the fun begins! We cannot just replace it with one of its children because there are cases where the child has its own children too. The trick is to find its successor and replace the node with the successor. (*Note: we can use the predecessor too.)*

By successor, I mean the next value of the value to be deleted if all of the values in the tree are in ascending order.

How do we find the successor of a node? Basically, ***the successor is the smallest node in the right sub-tree of the node to be deleted.***

Finding the successor

```
// method to find successor, next-highest value after delNode
private Node getSuccessor(Node delNode) {
    _Node____successorParent_______;
    Node successor = delNode;
    Node current = delNode.__right_______;

    while(current != null) {
         ____SP = successor________________________;
         successor = ______________________________;
          current = current.________________________;
    }

    // if the successor is not the right child of delNode
    if(successor != delNode.___________________) {
         ___________________________ = successor.______________;
         successor.________________ = delNode.________________;
    }
    return successor;
}
```

Deletion continued

```
else {  // have two children
    Node successor = getSuccessor(current);
    if(current == __root_____)
        root = __successor___;
    else if(____isLeftChild____________)
        parent.__left_________ = __Successor_____;
    else
        parent.__right________ = __successor_____;

    // Are we done?

    Successor. left = curr. left
}
```

Efficiency of deletion
This also first requires searching the node to be deleted, ________________.
(Once again, probably!)

Additionally, it requires a few more comparisons to find its successor and a small constant time to disconnect the item and/or connect its successor.

Traversing a BST

This means visiting each node in a specific order. This process is not as commonly used in BST as other operations such as search, insertion and deletion.

There are three simple ways to traverse a tree: preorder, inorder and postorder.

inorder LPR (symmetric)
preorder PLR
postorder LRP

Most commonly used approach to traverse is inorder.

An inorder traversal will visit all the nodes in ascending order.

Revisiting recursion

The simplest way to carry out an inorder traversal in a BST is recursion:

- Call itself to traverse the node's left subtree.
- Visit the node
- Call itself to traverse the node's right subtree.

```
// inorder traversal of the tree
public void traverse() {
    inOrderHelper(root);
}

private void inOrderHelper(Node nodeToVisit) {
    if(nodeToVisit != null) {
        inOrderHelper(nodeToVisit.__left______________);
        System.out.print("["+nodeToVisit.key+",
"+nodeToVisit.value+"]");
        inOrderHelper(nodeToVisit.__right_____________);
    }
}
```

O(n)

Exercise 3

Given a BST below, trace the inOrderTraversal().

```
       43
      /  \
    21    56
   /  \     \
 14   35     67
```

inorder: 14, 21, 35, 43, 56, 67

preorder: 43, 21, 14, 35, 56, 67

post order: 14, 35, 21, 67, 56, 43

Efficiency of traversing

Naturally, traversing is not as fast as other operations. It is O(___n___).

Wait, we are not done yet.

Let's take a look at the following input data.

1, 2, 3, 4, 5, 6, 7

Build a binary search tree with the values.

Now, how long will it take to find 7?

Also, how long will it take to delete 7? And, how long will it take to add 8?

So, what is the worst-case running time complexity of a BST for search, insertion, and deletion?

To remedy this worst-case running time of BSTs, there have been many inventions of self-balanced tree structures such as Red-Black Tree, 2-3 tree, Splay Tree, AVL tree, etc.