

## 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 (worst-case running time complexity of \_\_\_\_\_). But, it has issues with insertion and deletion.

Whenever we need to insert a new item into an ordered array, we first need to find where it should go and then shift all of the items that are greater than the new item up by one space in the array to make a room for the new item. 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 head 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 the 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 in reading (from left to right) each row 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 duplicate keys 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:

- Searching
- Insertion
- Deletion
- Traversal

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

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

    /**
     * Searches for an element with key in the tree.
     * @param key key value to search
     * @return boolean value indicating success or failure
     */
    boolean find(int key);

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

    /**
     * Deletes an element from the tree.
     * @param key key of the element to be deleted
     */
    void delete(int key);

    /**
     * Traverses and prints key and value pairs in the tree.
     */
    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;

    Node(int k, double v) {
        key = k;
        value = v;
        left = null;
        right = null;
    }
}
```

**Searching for a given key. (Iterative approach)**

```

public boolean find(int key) {
    if (root == null) { // tree is empty
        return _____;
    }
    Node current = _____;
    // while no match
    while (current.key != key) {
        if (current.key < key) {
            current = current._____;
        } else {
            current = current._____;
        }
        if (current == _____) { // not found
            return _____;
        }
    }
    return _____; // 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 item

To insert a new item, 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 item.

```

public void insert(int key, double value) {
    Node newNode = new Node(key, value);
    if(root == null) { // tree is empty
        root = _____;
        return;
    }

    Node parent = _____;
    Node current = _____;
    while (true) {
        if (current.key == key) {
            _____;
        }
        parent = _____;
        if (current.key < key) {
            current = current._____;
            if (current == _____) {
                parent._____ = _____;
                return;
            }
        } else {
            current = current._____;
            if (current == _____) {
                parent._____ = _____;
                return;
            }
        }
    } // end of go right or left
} // end of while loop
} // 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 an item**

Deletion is more complicated than other operations.

To delete, we start by finding the node (using key) 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 (using key ) to delete and **case 1: not found**

```

Node parent = _____; // keep track of parent
Node current = _____; // keep track of current
_____ ;

while (current.key != key) {
    parent = current;
    if (current.key < key) {
        _____ ;
        current = current._____ ;
    } else {
        _____ ;
        current = current._____ ;
    }
    // case 1: not found
    if (current == _____) {
        return;
    }
}

```

**Case 2: a leaf**

Simply set it to null!

```

// case 2: a leaf (no children)
if (current._____ == null && current._____ == null) {
    if (current == root) {
        _____ = null;
    } else if (_____ ) {
        parent._____ = null;
    } else {
        parent._____ = 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 in a linked list.*

```

} else if (current._____ == null) {
    // case 3: one child
    if (current == root) {
        root = current._____;
    } else if (_____)
        parent._____ = current._____;
    } else {
        parent._____ = current._____;
    }
} else if (current._____ == null) {
    // case 3: one child
    if (current == root) {
        root = current._____;
    } else if (_____) {
        parent._____ = current._____;
    } else {
        parent._____ = current._____;
    }
}
}

```

### 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 a child has its own children too. The trick is to find its successor and replace the node with the successor. (*Note: using the predecessor is another option.*)

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 (key) in the right sub-tree of the node to be deleted.***

#### Finding the successor

```
// method to find successor, next-highest key after toDelete
private Node getSuccessor(Node toDelete) {
    _____;
    Node successor = toDelete;
    Node current = _____;

    while (current != null) {
        _____;
        successor = _____;
        current = current._____;
    }

    // if the successor is not the right child of toDelete
    if (successor != _____) {
        _____ = successor._____;
        successor._____ = toDelete._____;
    }
    return successor;
}
```

## Deletion continued

```

} else {
    // case 4: two children
    Node successor = getSuccessor(current);
    if (current == _____) {
        root = _____;
    } else if (_____ ) {
        parent._____ = _____;
    } else {
        parent._____ = _____;
    }

    // Are we done?

}

```

**Efficiency of deletion**

This also first requires searching for the node (using key) 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.

Most commonly used approach is inorder traversal.

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:

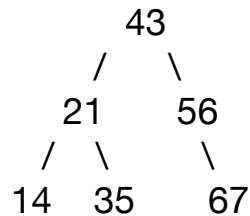
- Call itself to traverse the node's left subtree.
- Visit (print) the node (key and value)
- Call itself to traverse the node's right subtree.

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

private void inOrderHelper(Node toVisit) {
    if (_____) {
        inOrderHelper(toVisit._____);
        System.out.print(toVisit);
        inOrderHelper(toVisit._____);
    }
}
```

**Exercise 3**

Given a BST below, trace the `traverse()` in three different ways (preorder, inorder, and postorder).

**Efficiency of traversing**

Naturally, traversing is not as fast as other operations. It is  $O(\text{_____})$ .

**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.***