**Lesson 17 – Trees I**

**Reading Assignment:**

* Chapter 11, Sections 1 and 2 of the text.

**Learning Objectives:**

* List the basic operations of the ADT Binary Tree
* Describe the algorithm for traversing a Binary Tree
* Program a reference-based implementation of a Binary Tree

**Terminology:**

* **Trees** are used to represent relationships.
  + For example, a tree could be used to represent an organization, a family tree, or a military unit, such as an Infantry Company.
  + Each name or position within the organization would be represented by a **node**, or **vertex**, in the tree. At the top of tree is the Company Commander. At the next level, are the Platoon Leaders.
  + The lines between the nodes are called **edges**. The edge represents some form of relationship between the two nodes. In this example, the relationship is that of superior/subordinate.
* All trees are **hierarchical** in nature.
  + This means that a “parent-child” relationship exists between the nodes in the tree.
  + If an edge is between node *n* and node *m*, and node *n* is above node *m* in the tree, then *n* is the **parent** of *m*, and *m* is a **child** of *n*.
  + Children of the same parent are called **siblings**. Each node in a tree has, at most, one parent, and exactly one node—called the **root** of the tree—has no parent.
  + A node that has no children is called a **leaf** of the tree.
  + The parent-child relationship between the nodes is generalized to the relationships **ancestor** and **descendant**. For a node *n*, any node along the path between *n* and the root is an ancestor of *n* (including the root). Any child of *n* (or a child of a child, etc.) is a descendent of node *n*.
  + A **subtree** in a tree is any node in the tree together will all of its descendants. A **subtree of a node *n*** is a subtree rooted at a child of *n*.

Timeline

Description automatically generated with medium confidence

A picture containing text, envelope

Description automatically generated

* Formally, a binary tree is a set T of nodes such that either:
  + *T* is empty, or
  + *T* is partitioned into three disjoint subsets:
    - A single node *r*, the root;
    - Two possibly empty sets that are binary trees, called **left** and **right** subtrees of r.
* T is a binary tree if either
  + *T* has no nodes, or
  + *T* is of the form

*r*

/ \

*TL* *TR*

Where r is a node and *TL* and *TR* are both binary trees.

Diagram

Description automatically generated

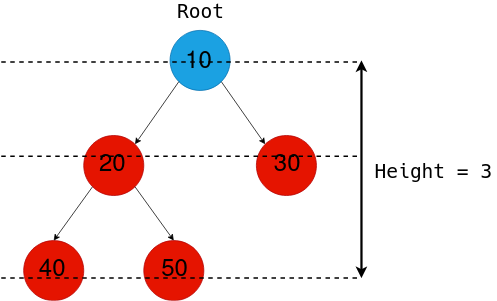
* The **height of a tree** is the number of nodes on the longest path from the root to a leaf. There are other equivalent ways to define the height of a tree *T*. One way uses the following definition of the level of a node *n*:
  + If n is the root of *T*, it is at level 1.
  + If n is not the root of *T*, its level is 1 greater than the level of its parent.
* For binary trees, it is often convenient to use a recursive definition of height:
  + If *T* is empty, its height is 0.
  + If *T* is a nonempty binary tree, then because *T* is of the form

*r*

/ \

*TL* *TR*

the height of *T* is 1 greater than the height of its root’s taller subtree; that is



**Binary Search Trees:**

* Notice two things about the binary trees we can build with the TreeNode class:
  + There is not ordering to the values of item in each node.
  + There is no requirement to minimize the height of the tree.
* By controlling both of these factors, we can maintain an organized tree that provides higher efficiency of operation.
* A **binary search tree** is a binary tree that is in a sense sorted according to the values in its nodes. For each node *n*, a binary search tree satisfies the following three properties:
  + *n*’s value is greater than all values in its left subtree *TL*.
  + *n*’s value is less than all values in its right subtree *TR*.
  + Both *TL* and *TR* are binary search trees.
* Discuss the construction of a BST.

**Chart, line chart

Description automatically generated**

**Recursive Implementation of a BST:**

* We can implement the basic functionality of a binary tree with a modification to the Node class:

public class BST<T extends Comparable<T>> {

    private T root;

    private BST<T> left;

    private BST<T> right;

    public BST() {

        this.root = null;

    }

    public BST(T root) {

        this.root = root;

        this.left = new BST<T>();

        this.right = new BST<T>();

    }

    public boolean isEmpty() {

        return (this.root == null);

    }

    public void removeAll() {

        this.root = null;

        this.left = null;

        this.right = null;

    }

    public int numElements() {

        if (this.isEmpty())

            return 0;

        else

            return 1 + this.left.numElements() + this.right.numElements();

    }

    public int height() {

        if (this.isEmpty())

            return 0;

        else

            return 1 + Math.max(this.left.height(), this.right.height());

    }

    public void add(T item) {

        if (this.isEmpty()) {

            this.root = item;

            this.left = new BST<T>();

            this.right = new BST<T>();

        } else {

            int compare = item.compareTo(this.root);

            if (compare < 0)

                this.left.add(item);

            else

                this.right.add(item);

        }

    }

    public T max() {

        if (this.isEmpty())

            throw new TreeException("Tree is empty!");

        if (this.right.isEmpty())

            return this.root;

        else

            return this.right.max();

    }

    public T min() {

        if (this.isEmpty())

            throw new TreeException("Tree is empty!");

        if (this.left.isEmpty())

            return this.root;

        else

            return this.left.max();

    }

*Use buildTreeString as starter code and only do remove if time permits*

 public void remove(T item) {

        if (this.isEmpty())

            throw new TreeException("Tree is empty.");

        int compare = item.compareTo(this.root);

        if (compare < 0)

            this.left.remove(item);

        else if (compare > 0)

            this.right.remove(item);

        else {

            if (this.left.isEmpty() && this.right.isEmpty()) {

                this.root = null;

                this.left = null;

                this.right = null;

            } else if (this.left.isEmpty()) {

                this.root = this.right.root;

                this.left = this.right.left;

                this.right = this.right.right;

            } else if (this.right.isEmpty()) {

                this.root = this.left.root;

                this.right = this.left.right;

                this.left = this.left.left;

            } else {

                T newRoot = this.right.min();

                this.root = newRoot;

                this.right.remove(newRoot);

            }

        }

    }

    public String toString() {

        StringBuilder buffer = new StringBuilder(50);

        if (!this.isEmpty())

            buildTreeString(buffer, "", "");

        return buffer.toString();

    }

    private void buildTreeString(StringBuilder buffer, String prefix, String childrenPrefix) {

        buffer.append(prefix);

        buffer.append(this.root.toString());

        buffer.append('\n');

        if (!this.left.isEmpty())

            if (!this.right.isEmpty())

                left.buildTreeString(buffer, childrenPrefix + "├── ", childrenPrefix + "│   ");

            else

                left.buildTreeString(buffer, childrenPrefix + "└── ", childrenPrefix + "    ");

        if (!this.right.isEmpty())

            right.buildTreeString(buffer, childrenPrefix + "└── ", childrenPrefix + "    ");

    }

}

* We can use this class to build different binary trees:

import java.util.Scanner;

public class TreeTest {

    public static void main(String[] args) {

        BST<Integer> tree = new BST<>();

        for (int i = 0; i < 12; i++) {

            tree.add((int) (Math.random() \* 25));

        }

        System.out.println(tree.toString());

        System.out.println();

        System.out.println("Num Elements: " + tree.numElements());

        System.out.println("Height: " + tree.height());

        System.out.println("Max: " + tree.max());

        System.out.println("Min: " + tree.min());

        //System.out.println("In Order: " + tree.inOrder());

        //System.out.println("Pre Order: " + tree.preOrder());

        //System.out.println("Post Order: " + tree.postOrder());

        System.out.println("Enter a number to remove: ");

        Scanner console = new Scanner(System.in);

        int num = console.nextInt();

        tree.remove(num);

        console.close();

        System.out.println(tree.inOrder());

        System.out.println(tree.toString());

    }

}