C212/A592 Lab 12

Intro to Software Systems

Instructions:

- Review the requirements given below and Complete your work. Please submit all files through Canvas (including all input and output files).
- The grading scheme is provided on Canvas

Lab 13: Binary Search Tree

- A BST is a ubiquitous data structure that provides fast look up times of data
- Object Oriented Programming is the typical programming paradigm for software development
- Data is typically stored in Objects.
- This is outside the scope of this class to discuss how to determine keys of objects, but we can find some way so that each object has a unique key
    - We use this key to store and look up the object, then query the object for the data we want
    - This data is typically called *satellite data*
- Thus we will be creating a binary search tree that has <Key, Value> pairs
- We will then be adding appropriate **Junit tests** to test the program.

Definition: A *Binary Search Tree* (BST) is a binary tree where each node has a `Comparable` key (and associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in the node's right subtree. (*Algorithms 4th edition, Sedgwick and Wayne)*

Definition: A *leaf node* is a node at that lowest level of the tree and has no children.

- That is, starting with the *Root Node* of the tree, any node's key to the left of the root is smaller than the root's key, and any node's key to the right of the root is larger than the root's key.
- This property holds for each node in the tree
- See figure 1 for an image of a BST

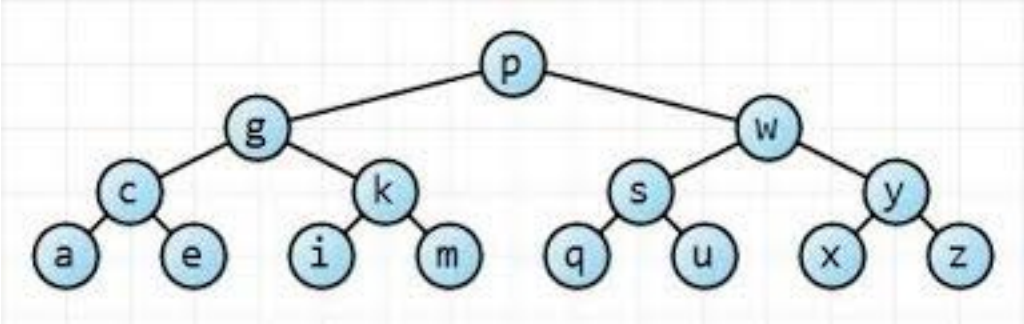

Figure 1

- We will be using Characters for values and their ASCII value as keys
- We will not be implementing the full functionality of a typical Binary Search Tree
- See the following BST API on next page:
    - Notice in the class definition we are defining our first class with generics
    - For the Key, we are putting the restriction on the Key that it must implement the *Comparable interface*
    - To be able to compare objects with each other in their natural ordering, we implement the Comparable interface
        - See Java documentation on comparable interface https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
        - We can use the *compareTo* method to determine which the direction the object should move down the tree
    - An object can be typed as any interface it implements, thus we will have a compile time error if we try to instantiate a BST with a key that does not implement the Comparable interface
    - See the following tree traversal Wikipedia page for the walk method https://en.wikipedia.org/wiki/Tree_traversal
        - To implement a walk, you recure on the left subtree until there are no children, then recure on the right subtree until no children
        - Were to handle the current node in the two recursion branches determines the order of the traversal/walk
- **Note:** Trees are almost always implemented with recursion because it is very natural to do

```
public class BST<Key, extends Comparable<Key>, Value> {

        private Node root;

        // inner Node class
        // note: an outer class has direct access to values in an inner class
        private class Node {
                private Key key;
                private Value value;
                private Node lChild;   // left child
                private Node rChild;   // right child

                 // number of nodes at this subtree
                // the value of N for the root will be # of nodes in entire tree
                // the value of N for a leaf node would be 1
                private int N;

                public Node (Key key, Value val, int N) { }
        }

        public int size() { } // returns # of nodes in the tree
```

```
        // returns the value associated with they key
        // returns null if the key is not in the tree
        public Value get(Key key) { }

        public void put(Value val, Key key) { } // inserts the key value pair into the tree

        // performs an in order walk of the tree printing the values
        public void walk() { }

        // Returns a pre-order, post-order, and in-order walk of the tree printing the values
        public String toString() { }

        // returns true is this tree (using root node) is exactly same as another BST, return false otherwise.
        public boolean isEqual(BST another) { }

        // here are some of the test cases I performed
        pubic static void main(String[] args) {
                Random rand = new Random();
                BST<Integer, Character> tree = new BST<>();

                for (int i = 0; i < 25; i++) {
                        int key = rand.nextInt(26) + 'a';
                        char val = (char)key;
                        tree.put(key, val);
                }

                // note:  not all of these chars will end up being generated from the for loop
                // some of these return values will be null
                System.out.println(tree.get((int)'a'));
                System.out.println(tree.get((int)'b'));
                System.out.println(tree.get((int)'c'));
                System.out.println(tree.get((int)'f'));
                System.out.println(tree.get((int)'k'));
                System.out.println(tree.get((int)'x'));
                tree.walk();
                //write code to test isEqual and toString methods!
        }
}
```