**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# **Introduction**

A **Map** is a type of collection that associates a key with a value. The mapping of keys to values can be accomplished using different underlying data structures. In this three-part assignment, you will be working with two such data structures and then using them to implement a **Map**. One is called a **Linked List** and the other is called a **Tree**.

A **tree** data structure can be defined recursively as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with references to nodes (the "children"), with the constraints that no reference is duplicated, and none points to root. A **Binary Search Tree (BST*)*** is a specific type of tree where each node can only have up to two children, and all of the data values in the left subtree of any node is always less than the node value and all of the data values in the right subtree of any node is always to the right of the node value.

In part III, you will be implementing a BST, and then creating a Map that uses the BST to store and retrieve entries. As before, **your Map should only use the BST by calling its methods.** It should not touch inner variables like root or create node objects itself. That is the BST's job to do internally.

Since the key-value pairs stored in the map could be of any type, we will be using [Generics](https://www.geeksforgeeks.org/generics-in-java/) to implement all our classes. For example, we could have < “Andrew”, 1> as a key-value pair (which is of type <String, Integer>) and we could also have < 432, 34> as a key-value pair (which is of type <Integer, Integer>). Hence, it is important that we provide a way to reuse the same code with different inputs. **Generics** enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

**Note:** You are **NOT** allowed to use any built-in Java data structures for any part of this assignment.

# **Program and Starter Code**

You are being provided with starter code which is an interface representing a **Tree**. You can find it in the course public folder ([public/7P](https://cs.unh.edu/~cs416/public/7P)). It contains the signatures of the methods that you need to implement. Do not change **Tree.java**. You will also need **Map.java** from previous parts of this project.

## **BST.java inner class Node**

You will need to create **BST.java**. It will declare a class **BST**, (declared as **BST<T extends Comparable> implements Tree<T>** .) This ensures that all data stored in your BST will have the compareTo method. When you begin writing the BST class, you will first need to define its own inner class **Node** (declared as **public class Node** **implements Tree.Node<T>** .) You will need to add fields (such as the left child, right child, and parent) and possibly other methods to make it behave as a BST node. It will need to implement the following methods:

**public Node( T value )**

The constructor for Node. It should initialize any necessary fields.

**public void setValue( T value )**

Sets the value of this node to the given parameter.

**public T getValue( )**

Gets the value of this node. Required for testing.

**public Node getLeft( )**

Gets the left child of this node. Required for testing.

**public Node getRight( )**

Gets the right child of this node. Required for testing.

**public Node getParent( )**

Gets the parent of this node. Required for testing.

## **BST.java**

**Note:** Your code may generate warnings about "unchecked call to compareTo" in **BST.java**. You can put @SuppressWarnings("unchecked") before the signature of any methods that use compareTo to suppress these warnings. We are bending some Java rules, so the compiler will warn you of that unless you use the above annotation.

You also have to implement the following methods for the **BST** class:

**public BST( )**

The constructor for BST. It should initialize any necessary fields.

**public Node getRoot()**

Returns the root of the BST. Required for testing.

**public boolean add( T value )**

Appends the specified value to the correct location in this BST. You should use the compareTo method of the BST's nodes' data when comparing to o. (You should not call o.compareTo(node.value), but node.value.compareTo(o) for nodes' values in the tree.) Should not add duplicate items to the BST.

**public String toString()**

Returns a string representation of this BST using indentation to represent each "level" of the tree. The root should be the furthest left when the string is printed, its children should be indented by 6 spaces, their children by 12 spaces, etc. Each node other than the root should be labeled as L for left children and R for right children. For example:

| L even  the  L undo  R word  L zag  R zig  R zoo |
| --- |

**public void clear( )**

Removes all of the elements from this BST.

**public T get( Object o )**

Returns the data that matches the object given. You should use the compareTo method of the BST's nodes' data when comparing to o. (You should not call o.compareTo(node.value), but node.value.compareTo(o) for nodes' values in the tree.) Should return null if the object does not match any data in the BST.

**public boolean contains( Object o )**

Returns true if this BST contains the specified element, false otherwise. You should use the compareTo method of the BST's nodes' data when comparing to o. (You should not call o.compareTo(node.value), but node.value.compareTo(o) for nodes' values in the tree.)

**public boolean isEmpty( )**

Returns true if this BST contains no elements, false otherwise.

**public boolean remove( Object o )**

Removes the specified element from this BST, if it is present. If this BST does not contain the element, it is unchanged. Returns true if this BST contained the specified element, false otherwise. When a node which is a left child of its parent and has both children is removed, its own left child should take its place. When a node which is a right child of its parent and has both children is removed, its own right child should take its place. When the root is removed and has both children, its left child should take its place. You should use the compareTo method of the BST's nodes' data when comparing to o. (You should not call o.compareTo(node.value), but node.value.compareTo(o) for nodes' values in the tree.)

**public int size( )**

Returns the number of elements in this BST.

## **BSTMap.java inner class Entry**

**Note:** Your code may generate warnings about "unchecked call to compareTo" in **BSTMap.java**. You can put @SuppressWarnings("unchecked") before the signature of any methods that use compareTo to suppress these warnings. We are bending some Java rules, so the compiler will warn you of that unless you use the above annotation.

You will need to create **BSTMap.java**. It will declare a class **BSTMap**, (declared as **public class BSTMap<K extends Comparable, V> implements Map<K, V>** .) When you begin writing the BSTMap class, you will first need to define its own inner class **Entry** (declared as **public class Entry<K extends Comparable, V> implements Map.Entry, Comparable** .) It will need to implement the following methods:

**public Entry( K key, V value )**

The constructor for Entry. It should initialize any necessary fields.

**public K getKey()**

Returns this entry's key.

**public V getValue()**

Returns this entry's value.

**public String toString()**

Returns a string representation of this Entry. It should follow this format: (key, value)

**public boolean equals( Object o )**

Compares an object o to this entry. If the object is an Entry (you can detect this using o instanceof Entry), first cast o to a variable of type Entry, then return whether its key and value both match this entry's key and value. Otherwise, return whether o equals this entry's key. Required for testing.

**public int compareTo( Object o )**

Compares an object o to this entry. If the object is an Entry (you can detect this using o instanceof Entry), first cast o to a variable of type Entry<K, V> and then return the result of the compareTo method between this entry's key and o's. Otherwise, return the result of compareTo between this entry's key and o.

## **BSTMap.java**

Your BSTMap class should use an instance variable of type BST to store its entries. Note that BSTMap should only call methods from BST, **BSTMap should not edit the BST's nodes directly**. You have to implement the following methods for the **BSTMap** class:

**public BSTMap( )**

The constructor for BSTMap. It should initialize any necessary fields.

**public BST<Entry<K,V>> getTree()**

Returns the BST being used by this map. Mostly used for testing purposes.

**public V put( K key, V value )**

Associates the given key with the given value in the map. Should not add duplicate items to the map. If the map previously contained a mapping for the key, the old value is replaced by the specified value.

**public V putIfAbsent( K key, V value )**

If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.

**public String toString()**

Returns a string representation of this BSTMap. It should be in the same format as the BST toString described above.

**public void clear( )**

Removes all of the elements from this map.

**public V get( K key )**

Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

**public boolean containsKey( K key )**

Returns true if this map contains a mapping for the specified key.

**public boolean isEmpty( )**

Returns true if this map contains no elements, false otherwise.

**public V remove( K key )**

Removes the mapping for a key from this map if it is present. Returns the value to which this map previously associated the key, or null if the map contained no mapping for the key.

**public int size( )**

Returns the number of elements in this map.