C212 Semester Project

cmtidmar

December 2019

Abstract

This project is a comparison between two Binary Search Tree algorithms and their efficiency.

1 Introduction

This semester project is for Fall 2019 C212. The project is divided into three parts: Stage 01, Stage 02, and Stage 03. Each Stage 01 and Stage 02 implements an algorithm for a Binary Search Tree, or BST. Stage 03 is a personal analysis of how the previous stages were executed and what I have learned, during this process.

2 Stage 01

Question (R17.10): Design an algorithm for finding the kth element (in sort order) of a binary search tree. How efficient is your algorithm?

In class, we have discussed generic methods. Here, we will implement a generic method into our binary search tree as the question did not specify what type is to be used, therefore a generic method (where T refers to any type and value refers to any variable of T) can be used to guide you into Stage 02.

```
public class GenericBinarySearchTree<T extends Comparable<T>> {
  private Node<T> root;
  GenericBinarySearchTree<T> left, right;
  //int left, right;


  public GenericBinarySearchTree() {
    this.root = null;
  }
  public GenericBinarySearchTree(Node<T> node) {
    this.root = node;
  }
  public GenericBinarySearchTree(T value) {
```

```
    this.root = new Node<T>(value);
  }
  public void insert(T value) {
    if (this.root == null)
      this.root = new Node<T>(value);
    else if (this.root.value().compareTo(value) > 0)
      this.root.left.insert(value);
    else if (this.root.value().compareTo(value) < 0)
      this.root.right.insert(value);
  }
  public String print() {
    if (this.root == null) return "()";
    else return "(" + this.root.print() + ")";
  }
  public String toString() {
    return this.root == null ? "" : this.root + "";
  }

  public int size(T value) {
      if (value == null) return 0;
      else return 1 + GenericBinarySearchTree.size(value.left) + GenericBinarySearchTree
  }

  public void find(T value) {
      if (value == 1 + GenericBinarySearchTree.size(this.left)) return this.GenericBinar
      else if (value <= GenericBinarySearchTree.size(this.left)) {
          return this.left.find(value);
      } else {
          return this.right.find(value - 1 - GenericBinarySearchTree.size(this.left));
      }
  }
```

Here is the Node class as a generic method:

```
class Node<T extends Comparable<T>> {
  T value;
  GenericBinarySearchTree<T> left, right;
  public Node(T value) {
    this.value = value;
    this.left = new GenericBinarySearchTree<T>();
    this.right = new GenericBinarySearchTree<T>();
  }
  public String toString() {
    return this.left + " " + this.value + " " + this.right;
  }
  public T value() {
```

```
    return this.value;
  }
  public String print() {
    return this.value + " " + this.left.print() + " " + this.right.print();
  }
}
```

3 Stage 02

Question (R17.11): Design an O(log(n)) algorithm for finding the kth element in a binary search tree, provided that each node has an instance variable containing the size of the subtree. Also describe how these instance variables can be maintained by the insertion and removal operations without affecting their big-Oh efficiency.

```
public class BinarySearchTree {
    int value;
    BinarySearchTree left, right;
```

Here is a tester that we will use for our Binary Search Tree - find kth:

```
    public static void main(String[] args) {
        BinarySearchTree BinarySearchTree = new BinarySearchTree(10, null, null);
        BinarySearchTree.add(5);   // 1
        BinarySearchTree.add(1);   // 2
        BinarySearchTree.add(64);  // 3
        BinarySearchTree.add(32);  // 4
        BinarySearchTree.add(12);  // 5
        BinarySearchTree.add(87);  // 6
        BinarySearchTree.add(43);  // 7
        BinarySearchTree.add(98);  // 8
        BinarySearchTree.add(49);  // 9
        BinarySearchTree.add(100); // 10
        System.out.println(BinarySearchTree.find(7));
    }
```

After creating a generic method of a Binary Search Tree, creating a more efficient an O(log(n)) version will not be as difficult as the the O(log(n)) algorithm can be modeled off the generic method.

```
    public BinarySearchTree(int node, BinarySearchTree left, BinarySearchTree right) {
        this.value = node;
        this.left = left;
        this.right = right;
    }
```

```
public String print() {
    return "(" + this.value + " " +
            ((this.left == null) ? "." : this.left.print()) + " " +
            ((this.right == null) ? "." : this.right.print()) + ")";
}

public String toString() {
    return this.left + " " + this.value + " " + this.right;
}

public static int size(BinarySearchTree node) {
    if (node == null) return 0;
    else return 1 + BinarySearchTree.size(node.left) + BinarySearchTree.size(node.ri
}

public int find(int k) {
    if (k == 1 + BinarySearchTree.size(this.left)) return this.value;
    else if (k <= BinarySearchTree.size(this.left)) {
        return this.left.find(k);
    } else {
        return this.right.find(k - 1 - BinarySearchTree.size(this.left));
    }
}


public boolean add(int node) {
    if (this.value == node) {
        return false;
    } else {
        if (node < this.value) {
            if (this.left == null) {
                this.left = new BinarySearchTree(node, null, null);
                return true;
            } else {
                return this.left.add(node);
            }
        } else {
            if (this.right == null) {
                this.right = new BinarySearchTree(node, null, null);
                return true;
            } else {
                return this.right.add(node);
            }
        }
    }
}
```

```
}
```

4 Stage 03: Conclusion and Reflection

One could safely assume that Stage 02's algorithm is more efficient and will run faster than Stage 01. This is because Stage 01 is written using a generic method which then can be translated into any other method. However, for Binary Search Tree's, an O(log(n)) method is the most efficient method as it has the fastest time complexity, whereas, an O(n) method has the worst time complexity.

I know my generic method (Stage 01) not completely correct due to a few errors I get when I run it, but I have decided to keep it in this way to show my understanding of generic methods and to hopefully receive feedback on how to better improve my implementation of generic methods. I understand a majority of the concept of generic methods, and can hand-trace the code until I get to the errors. That's when my understanding gets a little side-tracked, but the errors for me are learning curves and force me to think differently in the sense of my initial way didn't work out, what can I do differently to solve the problem. For me, the syntax and implementation of generic methods were a little harder to grasp onto, and I really had to reread the chapter several times in order to get an understanding. That's why I wanted to use a generic method in my project to demonstrate an understanding on something that I struggled with, even though it might not be fully correct. At the end of this project, I have gained a better understanding of the implementation and why they are useful. Though, my understanding might not be fully correct and I still have some difficulties, I think this project has reduced some of what I struggle with and has helped me develop myself in Java a little bit more.