Lab Assignment #6

Binary Search Tree

Full Name: Shreyas Srinivasa
BlazerId: SSRINIVA

CS 303 Algorithms and Data Structures

Spring 2023

1. Problem Specification

The purpose of this project was to implement an algorithm for binary search tree insertion, in-order traversal and search based on a key and evaluate its performance (machine specific execution time) with a large dataset. We use the binary search tree program to create a real world application to search for a particular record (key->description) from a very large dataset using just the keys provided in another dataset. We then compare and contrast the real world performance of the algorithm with its theoretical analysis

2. Program Design

```
class Node {
    Long key;
    String data;
    Node left = null;
    Node right = null;

    Node(Long key, String data) {
        this.key = key;
        this.data = data;
    }

    @Override
    public String toString() {
        return String.format("%d %s", key, data);
    }
}
```

The class representing a single element in the linked list data structure we use to represent out BST

```
 public static Node treeInsert(Node root, Node z) {
     Node y = null;
     Node x = root;

     while (x != null) {
         y = x;
         if (z.key < x.key)
             x = x.left;
         else
             x = x.right;
     }
     if (y == null)
         return z;
     else if (z.key < y.key)
         y.left = z;
     else
         y.right = z;
     return root;
 }
```

The iterative algorithm to insert a node into the tree. It takes the root node and the new node to be inserted as arguments and returns the root node

```java
public static void inorderTreeWalk(Node root) {
    if (root != null) {
        inorderTreeWalk(root.left);
        System.out.println(root);
        inorderTreeWalk(root.right);
    }
}
```

```java
public static void inorderTreeWalkIterative(Node root) {
    if (root == null)
        return;

    Stack<Node> s = new Stack<Node>();
    Node x = root;

    while (x != null || s.size() > 0) {
        while (x != null) {
            s.push(x);
            x = x.left;
        }
        x = s.pop();
        System.out.println(x);
        x = x.right;
    }
}
```

The in-order traversal algorithm implemented in its iterative and recursive forms

```java
public static Node treeSearch(Node root, Long key) {
    if (root == null || root.key.equals(key)) {
        return root;
    }
    if (key < root.key) {
        return treeSearch(root.left, key);
    } else
        return treeSearch(root.right, key);
}
```

```java
public static Node iterativeTreeSearch(Node root, Long key) {
    while(root != null && !root.key.equals(key)) {
        if(key < root.key) root = root.left;
        else root = root.right;
    }
    return root;
}
```

The BST search algorithm implemented in its iterative and recursive forms. It only requires the key to search for the respective node

```java
public static Node upcTree() {
    Node root = null;
    Integer lineCount = 0;
    final Integer totalLines = 177650;
    try (Scanner fileScanner = new Scanner(Paths.get("UPC.csv"))) {
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(",");
            Long key = Long.valueOf(elements[0]);
            String data = elements[1] + ", " + elements[2];
            root = treeInsert(root, new Node(key, data));
            lineCount++;
            System.out.format("%f\n", (lineCount * 1.0 / totalLines) * 100f);
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
    }
    return root;
}
```

The function that uses the BST insertion algorithm to form a tree using the records read from UPC.csv

```java
public static List<Long> getSearchKeys() {
    List<Long> keys = new ArrayList<>();
    try (Scanner fileScanner = new Scanner(Paths.get("input.dat"))) {
        while (fileScanner.hasNextLine()) {
            String line = fileScanner.nextLine();
            String elements[] = line.split(",");
            Long key = Long.valueOf(elements[0]);
            keys.add(key);
        }
        fileScanner.close();
    } catch (Exception e) {
        System.out.println("Error: " + e.toString());
        System.exit(1);
    }
    return keys;
}
```

The function used to read the search keys from input.dat

```java
public static void test1() {
    Node root = treeInsert(null, new Node(10L, ""));
    treeInsert(root, new Node(14234L, ""));
    treeInsert(root, new Node(5L, ""));
    treeInsert(root, new Node(1000L, "Data"));
    treeInsert(root, new Node(10L, ""));
    treeInsert(root, new Node(10000000L, ""));
    inorderTreeWalkIterative(root);
    Node x = iterativeTreeSearch(root, 1000L);
    System.out.println(x);
}

public static void test2() {
    Node upc = upcTree();
    System.out.println(iterativeTreeSearch(upc, 28800143178L));
    System.out.println(iterativeTreeSearch(upc, 18949005918L));
    System.out.println(iterativeTreeSearch(upc, 13286452029L));
}
```

The test functions to test the algorithms

3. **Output**

```
Time taken for UPC tree formation:= 378932 ms
79 , INDIANA LOTTO
Time taken to search for key 79:= 0 ms
93 , treo 700w
Time taken to search for key 93:= 0 ms
123 , Wrsi Riversound cafe cd
Time taken to search for key 123:= 0 ms
161 , Dillons/Kroger Employee Coupon ($1.25 credit)
Time taken to search for key 161:= 0 ms
2140000070 , Rhinestone Watch
Time taken to search for key 2140000070:= 2 ms
2140118461 , """V"": Breakout/The Deception  VHS Tape"
Time taken to search for key 2140118461:= 1 ms
2144209103 VHS, Tintorera - Tiger Shark
Time taken to search for key 2144209103:= 1 ms
2144622711 , Taxi : The Collector's Edition VHS
Time taken to search for key 2144622711:= 1 ms
2147483647 , Toshiba 2805 DVD player
Time taken to search for key 2147483647:= 1 ms
2158242769 288/1.12Z, GREEN SUGAR COOKIES4276
Time taken to search for key 2158242769:= 0 ms
2158561631 , HOT COCOA W/BKMK
Time taken to search for key 2158561631:= 0 ms
2158769549 njhjhn, gjfhjbgkj
Time taken to search for key 2158769549:= 0 ms
2160500567 2.25 oz (64)g, Dollar Bar Rich Raspberry
Time taken to search for key 2160500567:= 0 ms
2172307284 , Mixed seasonal flower bouquet
Time taken to search for key 2172307284:= 0 ms
2177000074 , 4 way 13 AMP Extension Lead (Wilkinson UK)
Time taken to search for key 2177000074:= 0 ms
2184000098 21 oz, Christopher's Assorted Fruit Jellies
Time taken to search for key 2184000098:= 0 ms
2187682888 , fairway
Time taken to search for key 2187682888:= 0 ms
```

Output 1. BST Search Results

4. Analysis and Conclusions

Time and Space Complexity Analysis by **Shwet Shukla** from **OpenGenus**

1. Time complexity:

i. Best case: When the tree is balanced we have to traverse through a node after making h comparisons for searching a node which takes time which is directly proportional to the height of the tree (log N) and then copying the contents and deleting it requires constant time so the overall time complexity is O(log N) which is the best case time complexity.

ii. Average case: Average case time complexity is same as best case so the time complexity in deleting an element in binary search tree is O(log N).

iii. Worst case: When we are given a left skewed or a right skewed tree(a tree with either no right subtree or no left subtree), then we have to traverse from root to last leaf node and the perform deletion process so it takes O(n) time as height of the tree becomes 'n' in this case. So overall time complexity in worst case is O(n).

2. Space complexity:

The space complexity of this algorithm would be O(n) with 'n' being the depth of the tree since at any point of time maximum number of stack frames that could be present in memory is 'n'.

OPERATION	WORST CASE	AVERAGE CASE	BEST CASE	SPACE
Search	O(N)	O(log N)	O(1)	O(N)
Insert	O(N)	O(log N)	O(1)	O(N)
Delete	O(N)	O(log N)	O(N)	O(N)

Complexity Analysis from GeeksFromGeeks

Searching: For searching element 1, we have to traverse all elements (in order 3, 2, 1). Therefore, searching in binary search tree has worst case complexity of O(n). In general, the time complexity is O(h) where h is the height of BST.

Insertion: For inserting element 0, it must be inserted as the left child of 1. Therefore, we need to traverse all elements (in order 3, 2, 1) to insert 0 which has the worst-case complexity of O(n). In general, the time complexity is O(h).

Deletion: For deletion of element 1, we have to traverse all elements to find 1 (in order 3, 2, 1). Therefore, deletion in binary tree has worst case complexity of O(n). In general, the time complexity is O(h).

As we already know from previous labs:-
The Time Complexity of an algorithm/code is **not** equal to the actual time required to execute a particular code, but the number of times a statement executes.

The **actual time required to execute code is machine-dependent**.

Instead of measuring actual time required in executing each statement in the code,Time Complexity considers how many times each statement executes.

Observations based on the lab experiment

- We perform multiple insertions to form the UPC BST (177650 to be exact). Therefore our total insertion time should have the average case complexity of **O(nlogn)** and worst case complexity of **$O(n^2)$**
- The search operations performed in the experiment execute incredibly fast, and take virtually no time when measured in the order of milliseconds. This is a testament to the performance of BST.

5. References

- https://algs4.cs.princeton.edu/32bst/#:~:text=A%20binary%20search%20tree%20(BST,in%20that%20node's%20right%20subtree.
- https://opendsa-server.cs.vt.edu/ODSA/Books/lc/cs383/fall-2018/OpenDSA_Content/html/BST.html
- https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/

- https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/
- https://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/