Programming Assignment Report 5

Binary Search Tree

(CSCI B505 Fall 19)

Manisha Suresh Kumar

UID:2000529881

11.29s.2019

INTRODUCTION

Binary Search Tree is a data structure with a particular order to the elements stored in it. It is node-based and each node can have at most two children with the left child being less than itself and right child greater than it is. Operations such as insertion, deletion, search, etc., can be done efficiently with this structure. This assignment implements such basic operations and another operation to replace each node with sum of nodes greater than itself in the tree.

IMPLEMENTATION

A class called 'binarySearchTree' is created and it has the following variables and methods:

```
class binarySearchTree
  #Initializations
  self.key = None
  self.left = None
  self.right = None
  self.p = None   #Parent node
  self.root=None
  self.BSTkey=None    #sum of greater node of a particular key
  self.sum=None   #Used while calculating sum of greater nodes
  self.dict_DP ={}    #Dynamic Programming structure to hold the sums

  def insert(key)
  def contains(x,key)
  def inorder(x)
  def size()
  def smallest(x)
  def largest(x)
  def successor(key)
  def predecessor(key)

  def greaterSumTree()
```

Where 'x' is an object of the class and its key is that of the root. Note that other functions are also in the class that are called within these main classes. They are explicitly mentioned later.

ANALYSIS

PART-1

Note**:**

As suggested for the assignment, implementation of insert(), contains(), inorder(), smallest(), largest(), successor() and predecessor() was done referencing Chapter 12 of CLRS.

- **`insert(key)`**

  The insert function starts at the root and goes down the tree on either one of the side: left or right. In the worst case,

  Time complexity is *O(h)* where 'h' is the height of the tree.

  Space complexity is *O(n)* where 'n' is the number of nodes already in the tree.

- **`contains(x,key)`**

  The contains function searches for 'key' in the tree and returns 'True' if found otherwise, returns 'False'. The working is similar to the insert function. That is, starting from the root and recursively moving doing either subtrees until the key is found or not. In the worst case,

  Time complexity is *O(h)* where 'h' is the height of the tree.

  Space complexity is *O(n)* where 'n' is the number of nodes already in the tree.

- **`inorder(x)`**

  The inorder function is used to print the elements in the binary tree in a sorted manner from smallest to largest. This is done by recursively expanding left subtree first, print the current node, then expand right subtree. If we wish to print keys from largest to smallest, we need only swap the right subtree exploration with the left. Since, we need to visit every node in the tree, both Space and Time complexity will be:

  O(n) where 'n' is the number of nodes already in the tree.

- **`size()`**

  One way is to use the recursive approach similar to the one in inorder function were you replace the print statement with a count variable. Here, I attempted an iterative approach instead and used a stack which initially holds just the root of the tree. Then until the stack is empty, pop a node from the stack, increment the count, and push the children, if they exist, to the stack.

Time complexity is *O(n)* where 'n' is the number of nodes already in the tree.

Space complexity is *O(n)* where 'n' is the number of nodes already in the tree and in the stack.

- **`smallest() and largest()`**

  These functions find the smallest and largest key in the tree respectively. For this we need only pass down through the left subtree and right subtree respectively till we reach the leaf node which will be the required key.

  Time complexity is *O(h)* where 'h' is the height of the tree.

  Space complexity is *O(n)* where 'n' is the number of nodes already in the tree.

- **`successor(key) and predecessor(key)`**

  These functions find the element just larger and just smaller than the key respectively. Consider successor() function. With the key that's passed to the function, we first get the node for the key using get_item(x,key) function. Now, if the node is not the leaf, then just calling the smallest() on the right subtree will give the answer. Otherwise, we will need to traverse upwards till the answer is found. For predecessor() function we call largest() on the left subtree of the node that's isn't the leaf. In the worst case,

  Time complexity is *O(h)* where 'h' is the height of the tree.

  Space complexity is *O(n)* where 'n' is the number of nodes already in the tree.

PART-2

- **`greaterSumTree()`**

  To do this I created two other functions that are called within this function; greaterSumPerNode(x) and replaceWithGreaterSum(x) where x is the root node. The first function calculates the sum of greater nodes for all nodes and stores it in the 'BSTkey' variable of that node. The second function then replaces the actual 'key' with 'BSTkey'. The greaterSumPerNode(x) functions in turn calls rightTreeSum(x) on the right subtrees to find the sum of nodes in the right subtree of a particular node. Once the sum is found, this becomes the sum of greater elements of the node that called the function. Memorization is implemented by storing this sum so that if the afore mentioned node is greater than any other node, then we needn't calculate the sum all over again. So, rightTreeSum(x) need only run once for every node in the tree.

Time complexity is $O(n)$ where ‘n’ is the number of nodes in the tree.

Space complexity is $O(n)$ where ‘n’ is the number of nodes in the tree.

To explain inorder function on BST before and after calling greaterSumTree(), we’ll use the following example. Elements are inserted into the tree using insert() in the following order:

11, 2, 29, 1, 7, 15, 40, 35

Calling inorder() on this, we get:

1, 2, 7, 11, 15, 29, 35, 40

Calling inorder() after greaterSumTree(),

139, 137, 130, 119, 104, 75, 40, 0

Clearly calling inorder() on the tree before and after greaterSumTree(), changes the order form smallest-largest to largest-smallest. The logic behind this is simple. The sum of greater elements of the node in the tree that is already the largest will be the smallest and the sum of greater elements of the node in the tree that is the smallest will be the largest. Hence, the reversal in order.

REFERENCES

1. Introduction to Algorithms by Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen