COMP202 HOMEWORK 5

FALL 2020

I have completed this assignment individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used during this assignment:

(i) Course textbook,

(ii) All material that is made available to me by the professor (e.g., via Blackboard for this course, course website, email from professor / TA),

(iii) Notes taken by me during lectures.

I have not used, accessed or taken any unpermitted information from any other source. Hence,

all effort belongs to me.

I signed as Sinan Cem Erdoğan (68912).

```
void add(data)
    root = recursiveAdd(root, data)

TreeNode recursiveAdd(root,data)
    if(root = null)
      root = new TreeNode(data)
      return root

    root.childs = root.childs + 1
    if(root.data > data) {
      root.left = recursiveAdd(root.left,data))

    else if (root.getData() < data)
      root.right = recursiveAdd(root.right,data))

    return root
```

Space Complexity for add method:
To add an element to the three, we first search fort he location of the new node.

We are using recursiveAdd function to search the location.

In average, its time complexity is O(h) where h is the height of the tree.

In the worst case, we need to go deepest node so, its time complexity is O(n), where n is the number of nodes in the tree.

Since the time compleixty of the add function is O(1).

We have the time complexities for adding a new node to the tree:

Average case : O(h) + O(1) = O(h)

Worst case : O(n) + O(1) = O(n)

Space Compleixty for add method:

Whenever we use add method we allocate memory for a single node.

To construct a binary search tree , we need to allocate memory for number of nodes in the tree.

So, the space compleixty for constructing a tree consisting of n nodes is O(n).

```
double kthElement(k)
    kthElementInOrder(root,k)
    countNode = 0
    return kth


void kthElementInOrder(root,k)
    if (root == null)
        return

    if (countNode <= k)
        kthElementInOrder(root.left, k)
        countNode++

        if (countNode == k)
          kth = root.data


        kthElementInOrder(root.right, k)
```

Time Compleixty for kthElement method:
The visit every node using inorder traversal until the kth node.
It takes O(k) times which is O(n).
Space Compleixty for kthElement method: O(1)

```
 int size()
    return sizeRecursive(root)

int sizeRecursive(root)
  if (root = null)
    return 0

  return sizeRecursive(root.left) + 1 + sizeRecursive(root.right)
```

Time Compleixty for kthElement method:
The visit every node using inorder traversal.

It takes O(n) times (n is the number of nodes in the tree).
Space Compleixty for size method: O(1)

```
int[] toArray()
    int[] array = new int[root.getNChilds() + 1]
    toArrayRecursive(root,array)
    index = 0
    return array

void  toArrayRecursive(root ,array)
     if (root == null)
          return
      toArrayRecursive(root.left array)
      array[index] = root.data
      index++
      toArrayRecursive(root.right array)
```

Time Compleixty for kthElement method:
We take elements of the tree in to an integer array one by one. For an array size of n element, it take O(n) times. So our time complexity is linear.

Space Compleixty for kthElement method:
We create an array size of n elemnts. We use O(n) extra space.

```
double perfectMedian(bst)
    return bst.root.data
```

Time Compleixty for perfectMedian exercise:
In perfect Binary Search Tress root is the median. Just returning the root and it takes constant time O(1).
Space Compleixty for perfectMedian exercise: O(1)

```
  public static double anyMedian(BinarySearchTree bst)
    int size = bst.size

    if(size % 2 == 0)
      return (bst.kthElement(size/2) + bst.kthElement((size + 2) / 2) ) / 2
     else
      return bst.kthElement((size + 1)  / 2)
```

Time Compleixty for anyMedian exercise:
We use the kthElement method to find the median. First we find the size to locate the median node of the tree then we return its value. It takes O(n)+O(k) times which is O(n).
Space Compleixty for anyMedian exercise: O(1).

```
double nChildrenMedian(bst)
    TreeNode root = bst.root
    numberOfNodes = root.childs + 1
    target = (numberOfNodes + 1) / 2
    x = medianNChild(root,target)

    if( numberOfNodes % 2 == 0)
      return x + medianNChild(root,target + 1) ) / 2
    else
      return x

double medianNChild(root,target)
    int k
    if(root.left = null)
      k = -1

    else
       k = root.left.childs


    if (k+2 = target) return root.data
    else if (k+2 > target)
      return medianNChild(root.left,target)
    else
      return medianNChild(root.right,target - k - 2);
```

Time Compleixty for nChildrenMedian exercise:
We use the helper function called medianChild to get to target node. At each step, we decide whether we should go right or left using the information number of childs. It takes O(h) time where h is the height of the tree. In the worst case it takes O(n) times where n is the number of nodes in the tree. nChildrenMedian function does basic operation to return the value so, it takes constant time.

Average case : O(h) + O(1) = O(h)

Worst case : O(n) + O(1) = O(n)

Space Compleixty for nChildrenMedian exercise: O(1).

```
double twoTreesMedian(tree1, tree2)

    int[] array1 = tree1.toArray
    int[] array2 = tree2.toArray
    int[] array3 = combineArrays(array1,array2)

    newTree = arrayToBst(array3)

    return nChildrenMedian(newTree)
```

```
int[] combineArrays(array1 ,array2)
    int i = 0, j = 0, k = 0
    int length1 = array1.length
    int length2 = array2.length
    int[] array3 = new int[length1 + length2]

        while (i<length1 && j <length2)
            if (array1[i] < array2[j])
                array3[k++] = array1[i++]
            else
                array3[k++] = array2[j++]

        while (i < length1)

            array3[k++] = array1[i++]

        while (j < length2)
            array3[k++] = array2[j++]

    return array

BinarySearchTree arrayToBst(array)
    BinarySearchTree newTree = new BinarySearchTree()
    for(int i = 0; i < array.length; i++)
      newTree.add((array[i]))
    return newTree
```

Time Compleixty for twoTreesMedian exercise:

To perform this exercise, we use 1 helper method called toArray and 2 helper function called combineArrays and arrayToBst.

First we take the elements of the trees into two different array using inorder traversel. Let's say number of the elements in tree1 and tree2 are k and j respectively. It takes O(k) + O(j) times.

Second we combine two sorted elements using merge sort this takes O(k + j) times.

Finally we return the median value of the new tree using nChildrenMedian which takes O(h) times.

Over all, it takes O(k) + O(j) + O(k +j) + O(h). In the worst case (k + j = h) we have O(2(k+j)) which is linear complexity O(n).

Space Compleixty for twoTreesMedian exercise:

We are creating tree array sizes of k, j, and k + j and a tree size of k+j. So space complexity is linear O(k + j) or we can say O(n).