COMP2119 Introduction to Data Structures and Algorithms
Assignment 4 - Trees and Sorting Algorithms

Cheng Ho Ming, Eric (3036216734) [Section 1C, 2024]

Monday 25^{th} November, 2024 18:09

Question 1

(a)

```
class BinarySearchTree:
    TreeNode rootNode = None
    # The insert method to support adding tree nodes into the tree
    Function Insert(TreeNode newTreeNode, TreeNode root=rootNode):
        if root is None:
            root = newTreeNode
        else:
            if root.value > newTreeNode.value:
                # Insert new TreeNode into left subtree
                if root.left is None:
                    root.left = newTreeNode
                else:
                    Insert(newTreeNode, root.left)
            else if root.value < newTreeNode.value:
                # Insert new TreeNode into right subtree
                if root.right is None:
                    root.right = newTreeNode
                else:
                    Insert(newTreeNode, root.right)
            else:
                # Should not happen, as we assume the values of all tree nodes are distinct.
```

(b)

(i)

Given that a node can be a descendant of itself (a.k.a. not required a proper descendant), the lowest common ancestor (denote as c) of node a and node b must satisfy the following conditions:

$((\text{a.value} < \text{b.value}) \rightarrow (\text{a.value} < \text{c.value} \leq \text{b.value})) \cup ((\text{a.value} > \text{b.value}) \rightarrow (\text{a.value} \geq \text{c.value} > \text{b.value})) \cup (\text{a.value} = \text{b.value} = \text{c.value})$

The algorithm in pseudocode is:

```
class BinarySearchTree:
    # The method to find the lowest common ancestor of two nodes
    Function FindLowestCommonAncestor(TreeNode a, TreeNode b, TreeNode c=rootNode):
        # If the two given nodes are the same, return the node itself as the lowest common ancestor
        if a.value == b.value:
            return a
        # Ensure that a.value < b.value
        else if a.value > b.value:
            return FindLowestCommonAncestor(b, a, c)
        # c is too large (not the lowest common ancestor)
        # "reduce" the value of c by moving to the left subtree
        else if c.value > b.value:
            return FindLowestCommonAncestor(a, b, c.left)
        # c is too small (not the lowest common ancestor)
        # "increase" the value of c by moving to the right subtree
        else if c.value < a.value:
            return FindLowestCommonAncestor(a, b, c.right)
        else:
            return c
```

(ii)

```
class BinarySearchTree:
    Function GetHeight(TreeNode node):
        if node is None:
            return 0
        else:
            return 1 + max(GetHeight(node.left), GetHeight(node.right))

    Function IsAVLtree(TreeNode root=rootNode):
        # If the tree is empty, it is an AVL tree
        if root is None:
            return True
        # Check if the tree is balanced
        if abs(GetHeight(root.left) - GetHeight(root.right)) > 1:
            return False
        # Check if the left subtree is an AVL tree
        if not IsAVLtree(root.left):
            return False
        # Check if the right subtree is an AVL tree
        if not IsAVLtree(root.right):
            return False
        return True
```

Question 2

(1) Insert the values: 30, 20, 40, 10, 25

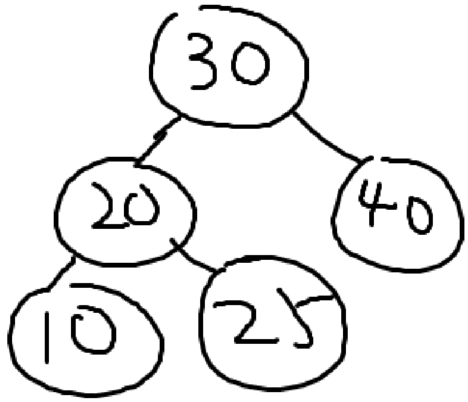

Figure 1: AVL Tree after inserting 30, 20, 40, 10, 25

(2) Insert the value: 5

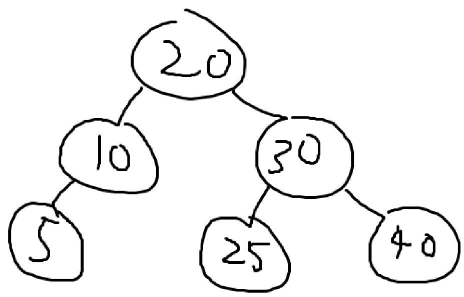

Figure 2: AVL Tree after inserting 5

(3) Delete the value: 30

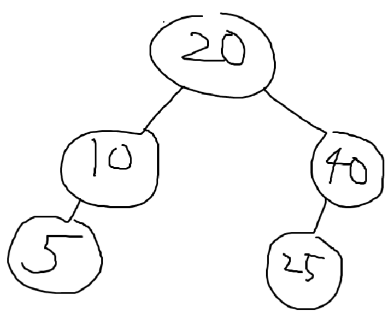

Figure 3: AVL Tree after deleting 30

Question 3

(a)

(i)

$O(n)$

(ii)

$O(n \log n)$

(iii)

$O(n^2)$

(iv)

$O(n^2)$

(v)

$O(n \log n)$

(b)

(i)

Algorithm 1 pseudocode that sorts the array in ascending order in O(n) time and O(1) extra space.

```
function REORDER(S)
    nextNegOne ← 0
    current ← 0
    lastPosOne ← length(S) − 1
    while current ≤ lastPosOne do
        if S[current] = −1 then
            Swap(S[nextNegOne], S[current])
            nextNegOne ← nextNegOne + 1
            current ← current + 1
        else if S[current] = 0 then
            current ← current + 1
        else                                        ▷ S[current] = 1
            Swap(S[current], S[lastPosOne])
            lastPosOne ← lastPosOne − 1
        end if
    end while
end function
```

(ii)

The *nextNegOne* variable is used to keep track of the next position to place the next -1. The *current* variable is used to iterate through the array. The *lastPosOne* variable is used to keep track of the last position to place the next 1. The algorithm iterates through the entire array, and if the current element is -1, it swaps the current element with the element at the *nextNegOne* position, increases *nextNegOne* and *current* by 1. If the current element is 0, it increases *current* by 1. If the current element is 1, it swaps the current element with the element at the *lastPosOne* position, and reduces *lastPosOne*.

Since the algorithm will iterate the entire array only once no matter what (for any value of $S[current]$, either *current* or *lastPosOne* will be increased or decreased by 1). Therefore, the time complexity is $O(n)$. Three variables are used to store the positions, and recursion is not used, so the space complexity is $O(1)$.

Question 4

```cpp
class MedianFinder {
    private:
        // Max heap to store the smaller half of the numbers
        std::priority_queue<int, std::vector<int>, std::less<int>> maxHeap;
        // Min heap to store the larger half of the numbers
        std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
    public:
        MedianFinder() {}

        void addNum(int num) {
            maxHeap.push(num);
            // elements in minHeap should be greater or equal to elements in maxHeap
            minHeap.push(maxHeap.top());
            maxHeap.pop();

            if (minHeap.size() > maxHeap.size()) {
                maxHeap.push(minHeap.top());
                minHeap.pop();
            }
        }

        double findMedian(void) {
            // There are odd number of elements
            if (maxHeap.size() > minHeap.size()) {
                return maxHeap.top();
            }
            // There are even number of elements
            return (maxHeap.top() + minHeap.top()) / 2.0;
        }
};
```

For the $addNum$ method, since the priority queue is implemented as a binary heap, the time complexity of adding an element (.push()) is $O(\log n)$, where n is the number of elements in the heap. The time complexity of removing the top element (.pop()) is also $O(\log n)$. Therefore, the time complexity of the $addNum$ method is $O(\log n)$.

For the $findMedian$ method, since the top element for each of the heap is located at the root of the heap, the time complexity of getting the top element (.top()) is $O(1)$. Therefore, the time complexity of the $findMedian$ method is $O(1)$.