CSCI 3320-001 Assignment 2 Alan Ramirez

Lists, Stacks, and Queues

1) (Hint: binary search) For a given sorted list (ascending order) and a target number, find the first index of this number in O(log n) time complexity. If the target number does not exist in the list, return -1.

Example 2:

Input: [1, 2, 3, 3, 4, 5, 10], 3
Output: 2

Explanation:
the first index of 3 is 2.

a) Conceptual Description – get the minimum (first), middle (median), and maximum (last) value of a list, then, compare the target to the middle value. If the target is greater than the middle value, set the minimum value to the middle value, and compute a new middle value. If the target is less than the middle value, set the maximum value to the middle value, and compute a new middle value. Repeat this until the low value is less than or equal to the high value.

b) Code

i)
```
def lowest_index(inputList, target):
    currentLowest = -1
    low = 0
    high = len(inputList) - 1

    while low <= high:
        mid = (low + high) // 2
        if inputList[mid] == target:
            currentLowest = mid
        elif inputList[mid] < target:
            low = mid + 1
            continue
        high = mid - 1

    return currentLowest

userList = [1, 2, 3, 3, 4, 5, 10]
targetNumber = 3
print(f"The first index of {targetNumber} is {lowest_index(userList,
targetNumber)}.\n")
```

ii) “The first index of 3 is 2.” *## Given the input list in the example.*

c) Complexity Analysis – let n be the length of the input list. Using the binary search method, we’re accessing the list $O(\log_2(n))$ times. Thus, the overall time complexity is $O(\log(n))$.

Lists, Stacks, and Queues

2) (Hint: stack) Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid in O(n) time. The brackets must close in the correct order, "()" and "()[]{}" are all valid but "(]" and "([)]" are not.

Example:

Input: "([)]" Output: False

Input: "()[]{}" Output: True

a) Conceptual Description – get the input string of brackets. If it's an open bracket [{(, append it to the stack. If it's a closed bracket)}], check if the top value in the stack is the bracket's pair. If a closed bracket does not match the top value in the stack, then return False. If the stack is empty after iterating through all values in the input string, then return True. If the input string is empty or the stack is not empty, return False.

b) Code

i)
```
def verify_string(inputString):
    stack = [] if len(inputString) > 0 else True
    bracketPairs = {'(': ')', '{': '}', '[': ']'}

    for char in inputString:
        if char in bracketPairs.keys():
            stack.append(char)
        elif stack and bracketPairs[stack[-1]] == char:
            stack.pop()
        else:
            return False

    return not stack

userString = "()[]{}"
print(f"The given string is {'valid' if verify_string(userString) else 'not valid'}.\n")
```

ii) "The given string is valid." *## Given the string "()[]{}".*

iii) "The given string is not valid." *## Given the string "([)]".*

c) Complexity Analysis – let *n* be the number of characters in the input string. Each access to a bracket from the input string requires O(1) time. Thus, the overall time complexity is O(*n*).

Lists, Stacks, and Queues

3) Implement a Queue **by linked list with only the given node (no other usable built-in data structures)**. Support the following basic methods:

enqueue(item). Put a new item in the queue.

dequeue(). Move the first item out of the queue, return it.

Example:

Input:
enqueue(1)
enqueue(2)
enqueue(3)
dequeue() // return 1
enqueue(4)
dequeue() // return 2

```
private static class Node<AnyType>
{
   private AnyType data;
   private Node<AnyType> next;

   public Node(AnyType data, Node<AnyType> next)
   {
      this.data = data;
      this.next = next;
   }
}
```

a) Conceptual Description – given an enqueue request, create a node with parameters "data" and "nextNode". If the queue is empty, set this node to the head of the queue and the tail of the queue. If the queue is not empty, set the tail node's nextNode parameter to the new node, then make the new node the tail. Given a dequeue request, print the data of the queue's head, then set the queue's head to the current head's nextNode.

b) Code

i)
```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.nextNode = None

class Queue:
    def __init__(self):
        self.header = None
        self.tail = None

    def enqueue(self, data = None):
        newNode = Node(data)
        if self.header == None:
            self.header = newNode
        else:
            self.tail.nextNode = newNode
        self.tail = newNode

    def dequeue(self):
        val = None
        if self.header != None:
            val = self.header.data
            self.header = self.header.nextNode
        print(val)

queue = Queue()

queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.dequeue()
queue.enqueue(4)
queue.dequeue()
```

ii) 1 *## Given the same queue in the example.*

iii) 2 *## Given the same queue in the example.*

c) Complexity Analysis – each enqueue and dequeue is in O(1) time. Thus, the time complexity is O(n) where n is the number of queue and dequeue requests.

Trees

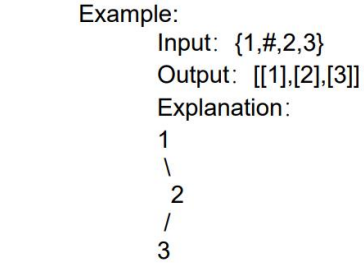
4) Given a binary tree, return the **level order traversal** of its nodes' values. (i.e., from left to right, level by level), Write the description and code of the algorithm. (The input can be **either** Tree-Node implementation, **or** array implementation of a binary tree)

Example:

Input: {1,#,2,3}

Output: [[1],[2],[3]]

Explanation:




```
Definition of TreeNode:
public class TreeNode {
    public int val;
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}
```

a) Conceptual Description – create a tree with nodes containing parameters data, left, and right. Set the left and right of the root to relevant nodes. To print in level order, the tree must be printed from left to right, top to bottom. Find the height of the tree; then, iterate through each level, printing from left to right.

b) Code

i)
```
class TreeNode:
    def __init__(self, data = None):
        self.data = data
        self.left = None
        self.right = None

def printLevel(node, level):
    if node == None:
        return
    if level == 1:
        print([node.data], end=", ")
    else:
        printLevel(node.left, level-1)
        printLevel(node.right, level-1)

def findHeight(node):
    if node == None:
        return 0
    leftHeight = findHeight(node.left)
    rightHeight = findHeight(node.right)
    if leftHeight > rightHeight:
        return leftHeight + 1
    else:
        return rightHeight + 1

def printLevelOrder(root):
    print("Level order: ", end="")
    height = findHeight(root)
    for level in range(1, height + 1):
        printLevel(root, level)

root = TreeNode(1)
root.right = TreeNode(2)
root.right.left = TreeNode(3)

printLevelOrder(root)
```

ii) Level order: [1], [2], [3], *## Given the tree shown in the example.*

c) Complexity Analysis – given that for each node in the tree, to print the current level, the code has to iterate through the node's left and right subtrees. In a tree with n nodes, the time complexity to print the level order is $O(n^2)$.

Trees

5) Given a binary tree, return all root-to-leaf paths (**depth-first**). Write the description and code of the algorithm.

Example
Input: {1,2,3,#,5}
Output: ["1->2->5","1->3"]
Explanation:


```
public class TreeNode {
    public int val;
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}
```

a) Conceptual Description – create a tree with nodes containing parameters data, left, and right. Set the left and right of the root to relevant nodes. Beginning at the root node, trace and append all paths to each of the leaves to a list. Then, print the list of all paths.

b) Code

i)
```
class TreeNode:
    def __init__(self, data = None):
        self.data = data
        self.left = None
        self.right = None

def getPaths(node, allPaths = []):
    def depthSearch(node, path = ''):
        if node == None:
            return path

        path += str(node.data)
        if node.left == None and node.right == None and path != 'None':
            allPaths.append('->'.join(path))

        depthSearch(node.left, path)
        depthSearch(node.right, path)

    depthSearch(node)
    return allPaths

root = TreeNode(1)
root.left = TreeNode(2)
root.left.right = TreeNode(5)
root.right = TreeNode(3)

print(getPaths(root))
```

ii) ['1->2->5', '1->3'] *## Given the tree shown in the example.*

c) Complexity Analysis – given a tree with *n* nodes, the code will iterate through all nodes at least once, requiring O(1) time complexity. Thus, the overall time complexity is O(*n*).