ADS Homework 9

Name - Tewodros Adane
Email - tadane@jacobs-university.de

Problem 9.1 – Stacks & Queues

a) The template implementation of a Stack backed up by a linked list is in the file "*Stack.h"*, and the usage is in "*problem_1.cpp"*.

b) The template implementation of a Queue using two stacks is in the file "*Queue.h"*, and the usage is in "*problem_1.cpp"*.

Problem 9.2 – Linked Lists & Rooted Trees

a) The pseudocode for an in-situ algorithm to reverse a linked list is given below:

```
REVERSE-LINKED-LIST(List)
     current_node = List->head // start at the head
     next = current_node->next // the node next to the head
     prev = NULL // the previous node to the current node
     WHILE (next != NULL)  // while the next node is not NULL
          // the current_node's next points to the previous
          current_node->next = prev
          // the new previous becomes the current node
          prev = current_node
          // the new current_node becomes the next node
          current_node = next
          // the next node becomes the current_node's next
          next = current_node->next
     // the current_node's next (which is now the tail node)
     points to the previous
     current_node->next = prev
     // the current_node becomes the head of the reversed
     linked list
     List->head = current_node
```

- The algorithm above will run in $\Theta(n)$ because it will make constant time operations **n** times as it moves through the linked list from the head to the tail.

- The algorithm is in-situ because it does not use additional memory to move around the elements of the list. It only changes the direction where the pointers point to, so that the linked list reverses in place.

b) To convert a Binary Search Tree (BST) into a sorted linked list we need to find a way to traverse through the tree and insert elements into a linked list in O(1), i.e., insert at the front of the linked list. And we need to insert starting from the largest element to the smallest, so that the linked list will be sorted in increasing order. Therefore, by using a reverse inorder traversal and inserting the elements into the linked list, we can convert a BST to a sorted linked list.

To derive the Asymptotic time complexity of this algorithm we just need to count number of nodes it has visited and multiply it by the time complexity of the operation done on the nodes. Since the algorithm visits n nodes and the operation at each node is O(1), the Asymptotic time complexity of the whole algorithm is O(n).

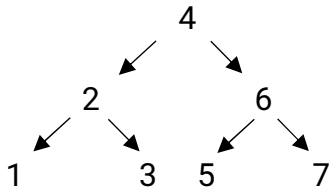
The template implementation of a BST including the method to convert to Linked List is in the file "*BinarySearchTree.h*", and the usage is in "*problem_2.cpp*".

c) The question requires the resulting BST should have a searching time complexity better than searching through a linked list (or better than O(n) time), which means the BST created should be balanced.

To convert a sorted linked list into a BST, we need to find the middle element of the linked list and divide them into left and right parts based on where they are relative to the middle element. Then recursively find and insert the middle element of the left and right parts until we have none left. This will result in a balanced BST.

For example:

Given the linked list, `A = 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7`, if we recursively find the middle element and insert into the BST, we will get the following:



The Asymptotic time complexity of this algorithm is dependent on the time it takes to insert into a BST which is O(h), but since this is a balanced BST it takes O(logn). Because there are n elements in the sorted linked list, thus n insertions into the BST, the Asymptotic time complexity of the algorithm is O(nlogn).

The template implementation of a Linked List including the method to convert to BST is in the file "*LinkedList.h"*, and the usage is in "*problem_2.cpp"*.