Homework 9

(P1) a) .ee Vs code implementation

Time Complexity Analysis:

1. push function for the push operation

Time Complexity: O(1) → constant time because the push function just creates a new node, sets its next pointer to the current top and modifies to update the top pointer.

Every operation mentioned previously is constant in time. In addition to that, the check for overflow is constant as well.

2) Pop function for the top operation

Time Complexity: O(1) → constant time, as it involves removing the top node by updating the top pointer to top->next and deleting the previous node that was at the top. ⇒ All constant-time operations

Also, the underflow check is also constant

3) The empty check: Time complexity is O(1) (constant time): checking whether the stack is empty is a simple comparison (top == null) which is a constant-time operation.

4) Print operation (print()): Time complexity: O(n) (linear) = it iterates through the whole stack (from top to bottom) and proceeds to print each node's data. As each node is visited once => O(n) time complexity where n is the number of nodes.

Problem 9.3

```
Algorithm ReverseLinkedList(L)
1.      if L.head = NULL then
// L.head = NULL (=) the list is empty
2.              error "empty list"

3.      prev = NULL
// 'prev' will hold the node coming before the current node as we
// reverse the links
4.      current = L.head
// pointing current to the head

5.      while current ≠ NULL do
// traverse the list until we reach the end
6.              nextNode = current.next
// temporarily storing the next node to not lose its reference.
7.              current.next = prev
// Reversing the direction of the linked list.
```

8 prev = current
// Move 'prev' forward to 'current', and 'current' is now processed.
9 current = nextNode
// Move to the next node in the original list using 'nextNode'.
10 end while

11 L head = prev
// set the head of the list to the last node processed, which is the new head of the reversed list.

Pseudocode breakdown: imposing an if condition that would output a message in case the list is empty. The message will be something explanatory. We initialise prev with null and current to the current head of the List. As long as we don't reach the end of the list we execute the while loop. We introduce a new variable and initialise it with the value of the next node to avoid loosing its reference. We will reverse the direction of the link by writing: current.next = prev (make the current node point to "prev" instead of "nextNode". Update the value of prev to the current node. We will also update the current node with the value of the next node. All of these commands will be repeatedly executed until there are no more nodes left to traverse, thus reversing

the direction of the link for all elements. After breaking out of the loop prev will still point to the last node we processed in the original list. This node is now the head of the new list that is reversed so we update head to prev.

The code provided above is an in-situ algorithm because it modifies the linked list directly, without requiring any additional data structures that store copies of the list or a major rearrangement of the list;

b) Code implementation in VS code

To derive the Time complexity of the given code, we need to analyze the functions and operations being performed. We will focus on the ones dependent on the size of the input

1) inOrderTraversal function: Time complexity: O(n): the function traverses each node exactly once and adds the values of the nodes to a singly linked list.

2) BST to LinkedList: O(n), since inOrderTraversal has a Time complexity of O(n) and the current function simply calls it.

3) printList: O(n), since this function traverses all n nodes of the linked list.

4) deleteTree: O(n): this function recursively deletes the nodes of the tree and each node is visited once and deleted.

5. deleteList: traverses and deletes each node of the list, n nodes ⇒ Time complexity is O(n)

Overall time complexity: Since all of the significant functions are of time complexity O(n) ⇒ O(n)

(c) Time complexity of converting the Linked List to BST:

The function LinkedListTOBST recursively does the conversion from the linked list to the binary search tree by finding the middle node, which we will proceed to make the root, and recursively applying the same process to the left and right sublists. Because a recursive split is performed at each step ⇒ the recursion depth is proportional to the number of nodes in the binary search tree: Thus, the recursion depth will be approximately O(log n), (n → number of elements in the linked list). Each recursive call processes the list by finding the middle node. This would require going through all n elements of the list, which takes O(n) time (since we need to move both the slow and fast pointer through the entire list). Thus, the overall time complexity will be: O(n log n).

2) Time Complexity for Searching in the resulting BST: Assuming the tree is balanced, the time complexity would be O(log n). This is because, in a balanced BST, we compare at each step the search value with the root node and move either left or right to the subtree. Also, the height of BST is log n, so the time search is logarithmic.

3) Time Complexity for Searching in a linked list O(n): all n elements of the linked list are traversed,

4) Comparison of Search Time Complexity:
In a sorted linked list: O(n) → we traverse through the list element by element
In a balanced BST: The search time complexity is O(log n), because we can efficiently eliminate half of the search space at each step by moving left or right based on comparisons.
Thus, the search time complexity in BST is lower than in the sorted linked list, thus BST is more efficient for searching, as O(log n) is much better than O(n) for large n.