Algorithms and Data Structures

Drishti Maharjan

May 12, 2019

Assignment 7

Problem 7.1

a. *Code in 7.1a.cpp stack.h*
Time complexity analysis:
void push(T x) = $\theta(1)$
T pop() = $\theta(1)$
bool isEmpty() = $\theta(1)$
Since all the operations don't depend on the size of the stack and takes constant time, all operations have constant time complexity.

b. *Code in 7.1b.cpp Queue.h*

Problem 7.2

a. To reverse a linked list in linear time, we can't use our regular recursive function. Instead, we can reverse the direction of the pointer using pointers next, prev, current. In the pseudocode, head is of type List and it points to the first element of the list being passed. Then, pointers are reversed in line 4-8. The return type is a pointer to the reversed list.

Pseudocode for insitu algorithm that reverses a linked list of n elements in linear time:

Algorithm 1 reverseList(List head)

```
1: prev ← NULL
2: current ← head
3: next ← NULL
4: while (current != NULL) do
5:     next ← current.next
6:     current.next ← prev
7:     prev ← current
8:     current ← next
9: return prev
```

It is an insitu algorithm because it just uses 3 pointers and doesn't create a new List and the pointers occupy constant memory, and it doesn't depend on the size of the list.
Time complexity is linear as every operation inside the while loop has constant time complexity, and the while loop runs n times. Hence, the time complexity becomes linear.

b. *Code Files:*
Test code: 7.2b.cpp
Implementation code: DefImp.cpp
Headers: headers.h

Time Complexity analysis:
The inorder traversal for this tree occurs from right-root-left. This order is used to construct the linked list. As we need the time complexity to convert BST to LinkedList, I will ignore the time complexity required for initially creating a BST.So the transformation algorithm leads to inorder traversal and pushes the elements in the list (in front). It is trivial that insertion takes $\theta(1)$. For the recursive traversal, time complexity is $\theta(n/2) = \theta(n)$ as it divides the subproblems into 2 halves, each halve having to deal with n/2 elements. Hence, total time complexity = $\theta(n)$

c. *Code in 7.2c.cpp*

Time complexity analysis:

Every time we need to find the mid position, we have to iterate through the list. For worst case, this operation would have time complexity $\theta(n)$. Adding an element at binary search tree requires us to go through the height of the recursion tree which is lgn so time complexity of adding an element is $\theta(lgn)$. While transforming the list to BST, we divide the problems into 2 halves, and use recursion. Mathematically,

$T(n) = 2T(n/2) + n + lgn$
By recursion tree method, we obtain total time complexity = $\theta(n(n+nlgn)) = \theta(n^2)$ for my implementation.

Extra Note: We can imporve this time by using red black tree which has self balancing property and we don't have to iterate through mid position every time while adding element.