Algorithms and Data Structures

Spring 2019

Assignment 7

Date: April 4, 2019

Taiyr Begeyev

Problem 7.1 Stacks and Queues

(a) Implement using C++, Python or Java the data structure of a stack backed up by a linked list, that can store data of any type, and analyze the running time of each specific operation. Implement the stack such that you have the possibility of setting a fixed size but not necessarily have to (size should be 1 if unset). Your functions should print suggestive messages in cases of underflow or overflow. You can assume that if the size is passed, it will have a valid value.

Checkout the **Stack.h**, **Stack.cpp**, **testStack.cpp** for the implementation of the task.
To execute run: **make -f Stack.mk**

Analysis of the running time of each specific operation:
1. Push
Time complexity is $T(n) = O(1)$. As we can observe, there are no loops and any recursive calls, therefore, it is completed in constant time.
2. Pop
Time complexity is $T(n) = O(1)$. As we can observe, there are no loops and any recursive calls, therefore, it is completed in constant time.
3. isEmpty
Time complexity is $T(n) = O(1)$. As we can observe, there are no loops and any recursive calls, therefore, it is completed in constant time.
4. print
Even though it is not a part of the task, I implemented this function in order to see all elements in a Stack. Time complexity is $T(n) = O(n)$. As we can observe, there is one *for* loop, thanks to which we iterate until the end of the Stack, while the current cursor is not *NULL*, to print all elements. Therefore, it is completed in Linear Time.
5. Constructors
They are obviously run in constant time. $T(n) = O(1)$.

(b) Implement a queue which uses two stacks to simulate the queue behavior.

Checkout the **Queue.h**, **Queue.cpp**, **testQueue.cpp** for the implementation of the task.
To execute run: **make -f Queue.mk**

Problem 7.2 Linked List and Rooted Trees

(a) Write down the pseudocode for an in-situ algorithm that reverses a linked lis of n elements in $\Theta(n)$. Explain why it is an in-situ algorithm.

```
struct LinkedList {
    int data;
    struct LinkedList *next;
};
typedef struct LinkedList List;

List* reverseLinkedList(List* myList) {
    List *prev, *current, *next;
    current = myList;
    prev = NULL;
    next = NULL;

    while(current != NULL) {
        //store next
        next = current->next;
        current->next = prev;
        //move pointers
        prev = current;
        current = next;
    }
```

```
    current = prev;
    return current;
}
```

Algorithm 1 Reverse a Linked List

```
procedure ReverseLinkedList(List myLinkedList)
    // Declare three pointers of struct List
    current = myList
    prev = NULL
    next = NULL
    while current is not NULL
        next = current.next
        current.next = prev
        prev = current
        current = next
    current = prev
    return current
```

To make this **ReverseLinkedList** function work, we need to initialize three pointers: previous, current, next. Then we iterate through the loop. We set the current's next elements to **next**. Then change the next of current. That's the place where actual reversing takes place. Also we need to adjust pointers by moving them. It is an in-situ algorithm, because we don't use any auxuliary space (no additional data structures are created). Since we have just one loop, the time complexity is $T(n) = \Theta(n)$.
Code is taken from **"Programming in C II"** Lab, Assigment 3, problem 2.

(b) Implement an algorithm to convert a binary search tree to a sorted linked list and derive its asymptotic time complexity.
Checkout the **BSTtoLinkedList.cpp** for the implementation of the task.
To execute run: **make -f BSTtoLinkedList.mk**
Since the program makes recursive in-order traversal calls and pushes elements at the front of the list. Every insertion takes takes constant time and each travesal function call takes $\Theta(n)$, therefore the time complexity is $\Theta(n)$

(c) Implement an algorithm to convert a sorted linked list to a binary search tree and derive its asymptotic time complexity.
Checkout the **testLinkedListtoBST.cpp** for the implementation of the task.
To execute run: **make -f testLinkedListtoBST.mk**
Time Complexity: Time complexity of the above solution is $\Theta(n)$ where n is the number of nodes.

Reference:

1. codercareer.blogspot.com. "Binary Search Tree and Double-linked List."
`http://codercareer.blogspot.com/2011/09/interview-question-no-1-binary-search.html`
2. GeeksForGeeks. "Construct Complete Binary Tree from its Linked List Representation."
`https://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-li`