Homework 7

Dushan Terzikj

due 10 April, 2018

Problem 1:

NOTE: All the source files contain documentation inside the source files. Due to lack of *doxygen* knowledge, I could not generate proper documentation of the code. Everything you need to know is explained before every method implementation. Please read them carefully to ensure easier homework checking.

(a) Implement in Python or C++ the data structure of a stack backed up by a linked list, that can work for whatever type, and analyze the running time of each specific operation. In order to achieve this in C++, make use of template ¡ class T ¿, on which you can find more information at Templates. Implement the stack such that you have the possibility of setting a fixed size but not necessarily have to (size is -1 when unset). Check for the correctness of your functions and throw exceptions with suggestive messages in cases of underflow or overflow(C++,Python). You can assume that if the size is passed, it always has a valid value. Checking will be done by case-tests alone. The tests will cover overflow and underflow cases, as well as tests for different types of variables.

Solution:

Please check directory *p1a*. It contains the source file for the Stack container which the task requires. Read the documentation carefully for fully understanding the code. *textStack.cpp* is used for testing. Since the type of your test cases are not specified, if your testing design differs from the test design in *testStack.cpp*, then please write your own script.

(b) Show how a queue can be implemented with two stacks.

Solution:

Queue can be implemented with two stacks in the following way:

Main methods for a queue are *enqueue* and *dequeue*. One of the methods needs to be costly, i.e., one of the methods will be $\Theta(1)$ and the other one $\Theta(n)$. Let us make *enqueue* $\Theta(1)$ and *dequeue* $\Theta(n)$. Below you can see the psuedocode of both methods. Assume that exceptions for overflow and underflow are never thrown.

Algorithm 1 enqueue

```
1: procedure ENQUEUE(Q, x)    ▷ Q is the queue and x is the new element to be pushed. Q is a class/structure which cointains two stacks inside it, say stack1 and stack2
2:     push x into stack1
3: end procedure
```

Algorithm 2 dequeue

```
1: procedure DEQUEUE(Q)
2:     if stack1 and stack2 are empty then return
3:     end if
4:     if stack2 is empty then
5:         while stack1 is not empty do
6:             x := stack1.pop()
7:             stack2.push(x)
8:         end while
9:     end if
10:    y := stack2.pop() return y
11: end procedure
```

Problem 2:

(a) Program an in-situ algorithm that reverses a linked list of n elements in $\Theta(n)$. Add an explanation to why it is an in-situ algorithm in the code.

Solution:

Please check folder *p2a*. It contains a class called *LinkedList*. Header file *LinkedList.h* contains the declaration of the methods and member variables/objects and source code *LinkedList.cpp* contains their implementation. *main.cpp* is used for testing. Since the type of your test cases are not specified, if your testing design differs from the test design in *main.cpp*, then please write your own script. Since the problem did not specify what kind of types it should support, this *LinkedList* is designed to support only **integer** types. Read the documentation in the source files for code clarification.

(b) Program an algorithm to convert a binary search tree to a sorted linked list and derive its asymptotic time complexity.

Solution:

Please check folder *p2b*. It contains two header files which contain the declaration of the methods and member variables/objects *LinkedList.h* and *BinarySearchTree.h* and their corresponding source files for implementation *LinkedList.cpp* and *BinarySearchTree.cpp*. *main.cpp* is used for testing. Since the type of your test cases are not specified, if your testing design differs from the test design in *main.cpp*, then please write your own script. Since the problem did not specify what kind of types it should support, this *LinkedList* and *BST* is

designed to support only **integer** types. Read the documentation in the source files for code clarification. The method you are looking for is called *toLinkedList()* and can be found in *BinarySearchTree.cpp*

Asymptotic time complexity:

Since the algorithm recursively makes an inverse inorder traversal (instead of left-root-right, it goes right-root-left) and pushes the elements at the front of the list. Every insertion in the list takes constant time and the traversal takes $\Theta(n)$, therefore the time complexity of the whole algorithm is $\Theta(n)$.

(c) Program an algorithm to convert a sorted linked list to a binary search tree and derive its asymptotic time complexity.

Solution:

Please check folder *p2c*. It contains two header files which contain the declaration of the methods and member variables/objects *LinkedList.h* and *BinarySearchTree.h* and their corresponding source files for implementation *LinkedList.cpp* and *BinarySearchTree.cpp*. *main.cpp* is used for testing. Since the type of your test cases are not specified, if your testing design differs from the test design in *main.cpp*, then please write your own script. Since the problem did not specify what kind of types it should support, this *LinkedList* and *BST* is designed to support only **integer** types. Read the documentation in the source files for code clarification. The method you are looking for is called *toBST()* and can be found in *LinkedList.cpp*

Asymptotic time complexity:

Read the steps of the algorithm. You can find them on top of the methods implementation. Since we iterate through the list every time when we need to find an element on position *mid*, this costs us $\Theta(n)$ at most. The adding of the element into the BST, costs us at most $\Theta(logn)$. We recursively "divide" the list into two equal by length parts. Therefore we have the equation:

$$T(n) = 2T(\frac{n}{2}) + n + \log_2 n$$

Using recursion tree method, we get $\Theta(n(n + logn)) \Rightarrow \Theta(n^2) = T(n)$

We can improve this time by using red-black tree or AVL trees. Then, *mid* element iteration would not be necessary because of the self-balancing property, leaving us with build time of $\Theta(nlogn)$.