

Assignment 8

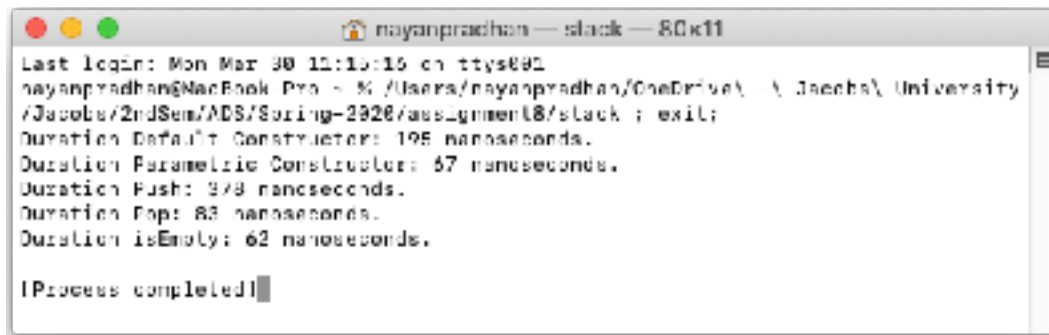
Done By: Nayan Man Singh Pradhan

Submitted: 30th March 2020

****NOTE: The resolution of the picture was bad when converting to pdf so please refer to the attached files**

Problem 8.1 (a)

Implemented in “stack.cpp” (make stack). I have two main() functions (one of them commented out). The first main function measures the runtime of each specific operation (Image 1) and the second main function tests the typical stack operations (push, pop, isEmpty) and the constructors (Image 2) along with the try...catch block. I have commented out certain print messages (like popping x, pushing x, etc) in their respective functions in order to get a more accurate runtime analysis (for Image 1). But, I have them while running the implementation (Image 2) in order to know what’s going on.



```
nayanpradhan — stack — 80x11
Last login: Mon Mar 30 12:15:15 on ttys001
nayanpradhan@MacBook Pro ~ % ./Users/nayanpradhan/OneDrive\ \ Jacobs\ University
/Jacobs/2ndSem/ADS/Spring-2020/assignment8/stack ; exit;
Duration Default Constructor: 195 nanoseconds.
Duration Parametric Constructor: 67 nanoseconds.
Duration Push: 378 nanoseconds.
Duration Pop: 83 nanoseconds.
Duration isEmpty: 62 nanoseconds.

[Process completed]
```

Image 1. Runtime Analysis of “stack.cpp” (file: stackTime)

Image 1 represents one instance of the time taken to compute the operations. The default constructor is making a default Stack, the parametric constructor is making a Stack with 5 elements, the Push is pushing the first element to the Stack, the Pop is popping the first (and only) element of the Stack, and the isEmpty is returning whether the Stack is empty (1) or not (0). We notice that the Default Constructor takes a longer time than the parametric constructor. Out of Push, Pop, and isEmpty method, the Push method takes the most time, the pop, then isEmpty. Push takes the most time as it has to add an element to the Stack by allocating the memory, changing the top (and bottom when the stack is empty), changing the pointers to point to the next element, and incrementing the currentSize. Pop removes the top element, changes the top, and changes the pointers. isEmpty simply checks whether top is null or not and returns true or false respectively (taking the least time).

```
nayanpradhan — stack — 85x28
Last login: Mon Mar 29 11:18:27 on ttys001
nayanpradhan@MacBook-Pro ~ % cd /Users/nayanpradhan/OneDrive\ -\ Jacobs\ University\ Jacobs\ 2ndSem\ ADS\ Spring-2020\ assignment\5\stack ; exit;
Pushing 1
myStack: 1
Pushing 2
myStack: 1 2
Pushing 3
myStack: 1 2 3
Pushing 4
myStack: 1 2 3 4
Pushing 5
myStack: 1 2 3 4 5
Stack is Full!
is empty: 0
Popping 5
myStack: 1 2 3 4
Popping 4
myStack: 1 2 3
Popping 3
myStack: 1 2
Popping 2
myStack: 1
Popping 1
Stack is Empty!
is empty: 1

[Process completed]
```

Image 2. Implementation of “stack.cpp” (file stack)

Problem 8.1 (b)

Implemented in “queue.cpp” (make queue). The behavior of a queue is simulated using two stacks (the stack implementation is a slightly modified version of **Problem 8.1 (a)**).

```
nayanpradhan — queue — 86x28
Last login: Mon Mar 29 11:22:12 on ttys001
nayanpradhan@MacBook-Pro ~ % cd /Users/nayanpradhan/OneDrive\ -\ Jacobs\ University\ Jacobs\ 2ndSem\ ADS\ Spring-2020\ assignment\6\queue ; exit;
Pushing 1
My Queue: 1
Pushing 2
My Queue: 2 1
Pushing 3
My Queue: 3 2 1
Pushing 4
My Queue: 4 3 2 1
Pushing 5
My Queue: 5 4 3 2 1
Queue Full!
My Queue: 5 4 3 2 1
is empty: 0
Popping 1
My Queue: 5 4 3 2
Popping 2
My Queue: 5 4 3
Popping 3
My Queue: 5 4
Popping 4
My Queue: 5
Popping 5
My Queue: Queue Empty!

[Process completed]
```

Image 3. Implementation of “twoStacks.cpp” (file queue)

Problem 8.2 (a)

Pseudocode for an in-situ algorithm that reverses a linked list (i used singly linked list as double linked list is not mentioned in the question) of n elements in $\theta(n)$.

	<u>Cost</u>
head is the first element of then linkedList	
<i>nextElem</i> = <i>NULL</i>	$\theta(1)$
<i>prevElem</i> = <i>NULL</i>	$\theta(1)$
<i>currElem</i> = <i>head</i>	$\theta(1)$
<i>while</i> (<i>currElem</i> != <i>NULL</i>)	$\theta(n + 1)$
<i>nextElem</i> = <i>currElem</i> -> <i>next</i>	$\theta(1)$
<i>currElem</i> -> <i>next</i> = <i>prevElem</i>	$\theta(1)$
<i>prevElem</i> = <i>currElem</i>	$\theta(1)$
<i>currElem</i> = <i>nextElem</i>	$\theta(1)$
* <i>head</i> = <i>prevElem</i>	$\theta(1)$

$$\begin{aligned}\text{Total Cost} &= \theta(1) + \theta(1) + \theta(1) + \theta(n + 1) + n \times (\theta(1) + \theta(1) + \theta(1) + \theta(1)) + \theta(1) \\ &= (c \times \theta(1)) + \theta(n + 1), \text{ where } c \text{ is a constant} \\ &= \theta(n)\end{aligned}$$

This is in-situ because our algorithm does not require extra space/create another auxiliary array in order to reverse the linked list.

Problem 8.2 (b)

BST is our Binary Search Tree , LL is Linked List and convToLL is our function
convToLL (BST, root, LL) //pass root first

```
convToLL (Tree BST, Node *node, linkedList& LL)
    if (node -> left != NULL)
        convToLL (BST, node -> left, LL)
    LL.insert(node->data)
    if (node -> right != NULL)
        convToLL (BST, node -> right, LL)
```

Here, we can see that our algorithm needs to visit each and every node. We keep going left (until we reach the left corner), insert the node data into our linked list, then visit the right node, and recursively continue

the process until we reach the right most element (right corner) of our binary search tree. Since we are visiting each node, the complexity of our algorithm is $\theta(n)$.

Bonus Problem 8.2 (c)

BST is our Binary Search Tree , LL is Linked List and convToBST is our function
convToBST (LL, 0, size) //pass whole linked list at first

```
convToBST (linkedList& LL, int left, int right)
    mid = getMiddleTerm[LL]
    root = mid
    while (left->data != NULL)
        root.leftChild = convToBST (LL, left, mid)
    while (right->data != NULL)
        root.rightChild = convToBST (LL, mid+1, right)
```

The asymptotic time complexity of this algorithm is $\log_2(n)$.

The search time complexity of the Linked List is $O(n)$ as for the worst case, if the targeted element is at the end of the linked list, we have to iterate through all the elements of the list. In the case of our binary search tree, the time complexity of the Binary Search Tree is $\log_2(n)$, which is better than searching in a linkedList. To search in a BST made from a sorted linked list, one can check the root, go to the left of the root if the target number is smaller than the root, or go to the right of the root if the target number is greater than the root, and continue the process recursively until the target element is found.