CS 3310 – Data and File Structures

Instructor: Dr. Ajay Gupta, Western Michigan University

Lab TA: Yu Guo

Ken Rivard

**SOFTWARE LIFE CYCLE REPORT – FOR ASSIGNMENT #2**

Design and Justifications

The problem specification asks to solve a series of problems.

1. The first problem given is to design a Java class stack using linked lists instead of using the stack implementation from the Java library; within this class stack should be at least two functions push() and pop(). So from here I took my Node class from assignment 1 and copied it over to this assignment since I needed to establish a linked list and the previous assignment did just that. All I had to do was implement Java Generics and it was within the problem specifications. Then within my stack class, I created a series of methods to help with the overall stack solution.

a. First, I implemented Java Generics within the stack class per to problem specifications

b. Stack() – to initialize the stack

c. isEmpty() – to check if the stack is empty or not and returning null if it is

d. size() – to keep track of the size of the stack

e. push() – adding elements to the stack in a LIFO standard

f. pop() – deleting elements from the stack in a LIFO standard

g. displayStack() – display the stack to the console to test the functionality

h. minValue() – //TODO – NOT WORKING RIGHT NOW

2. The second problem given is the same as the first except instead of stacks we are using queues. Two functions were required (add() and delete()) except I called these enqueue () and dequeue() in my program. Similar to the stack class, I created a series of methods within the queue class to help with my implementation of queues.

a. First, I implemented Java Generics within the queue class per the problem specifications

b. queue() – to initialize the queue

c. isEmpty() – to check if the queue is empty or not and returning null if it is

d. size() – to keep track of the size of the queue

e. enqueue() – adding elements to the queue in a FIFO standard

f. dequeue() – deleting elements from the queue in a FIFO standard

g. displayQueue() – display the queue to the console to test the functionality

3. Next, the third problem given is to implement a strategy of using to stacks to implement the functionality of a queue. I accomplished this by simply creating another stack within the main method, so now I have stack1 and stack2. From here, stack1 is created with a set of elements (not null/empty), then stack2 calls for elements from stack1 to be popped. As demonstrated in my main method, the first stack created is LIFO, and when stack2 pops elements from stack1, stack2 is created in a FIFO fashion therefore implementing the functionality of a queue using two stacks.

4. The fourth problem given is to implement a strategy of using queues to implement the functionality of a stack. I used the same idea as above. I created 2 queues (queue1 and queue2), then after adding elements to the first queue in a FIFO fashion, queue2 calls for elements of queue2 to be dequeued, therefore creating a stack of LIFO fashion.

5. Next, instead of writing a method in the Stack class I wrote code in the main method to handle finding the minimum value of the stack using 2 queues. I have also included the necessary file I/O for the minimum value test file as well as the stacks and queues test file. As per Dr Gupta’s specification’s within the email, the test for the minimum value works on integers (as requested in the email) and the stacks and queues test works on strings (as requested in the email). Now for the minimum value problem, my implementation of popping the minimum values to a second stack requires more space but requires much less time, therefore why I decided to use two stacks to achieve time of O(1). This implementation of two stacks will always return the lowest number is the original stack.

6. Lastly, the stacks and queues work on int, float, double, and String type values through the use of Java Generics

**Comparison of Theoretical v. Empirical Complexities**

Stack Class

**My push() method add elements to the stack in O(1) since it utilizes a linked list to insert elements to the stack, which is O(1).**

**The empirical analysis of push() equates to c1 + 5 since there is a constant object being pushed and there are 5 spaces in which memory needs to be allocated leading to O(1).**

**push Method**

*//add element to stack*

**public void** push(T obj){

Node oldTop = **top**;

**top** = **new** Node(obj);

**top**.**data** = obj;

**top**.**next** = oldTop;

**stackSize**++;

}*//end push()*

**My pop() method removes elements from the stack in O(1) since, like push() utilizes a linked list to delete elements from the stack, which is O(1).**

**The empirical analysis of pop() equates to c1 + 4 since there is a constant object being popped and there are 4 spaces in which memory needs to be allocated leading to a time of O(1).**

**pop Method**

*//remove element from stack*

**public** T pop() {

**if**(isEmpty()) **throw new** NoSuchElementException(**"Stack underflow"**);

T obj = **top**.**data**;

**top** = **top**.**next**;

**stackSize**--;

**return** obj;

}*//end pop()*

**My displayStack() method displays elements to the console in O(n) because it has to access every element from the stack in order to print each and every element to the console.**

**If the stack is empty, the empirical analysis equates to c1 + 1 since null is returned and only one comparison is being made.**

**If the stack has one element, the empirical analysis equates to c1 + 2 since the top element is returned and it looks for the next element and compares it to null.**

**If the stack has more than one element, the empirical analysis equates to 4n + 5 since within the while loop there are 4 spaces of memory that need to be allocated an N amount of times and outside the while loop there are 5 instances of memory that need to be allocated.**

**displayStack Method**

*//display the stack to the console*

**public void** displayStack() {

System.***out***.println(**"Stack Data Structure: "**);

**if**(**stackSize** == 0) {

System.***out***.println(**"empty!\n"**);

**return**;

}**if**(**top**.getLinkNext() == **null**) {

System.***out***.println(**top**.getData());

**return**;

}

Node ptr = **top**;

System.***out***.println(**top**.getData());

ptr = **top**.getLinkNext();

**while**(ptr.getLinkNext() != **null**) {

System.***out***.println(ptr.getData());

ptr = ptr.getLinkNext();

}

System.***out***.println(ptr.getData()+**"\n"**);

System.***out***.println(**"Stack Size = "** + size() + **"\n"**);

}*//end displayStack()*

Queue Class

**My enqueue() method add elements to the queue in O(1) since, like stack, utilizes a linked list to insert elements to the queue, which is O(1).**

**The empirical analysis of the enqueue() method is c1 + 7 since one constant object is being added to the queue and 7 memory slots are needed to be allocated to complete the enqueue process.**

**enqueue Method**

***//method to add elements to the queue***

**public void** enqueue(T obj) {

Node<T> oldLast = **rear**;

**rear** = **new** Node<T>(obj);

**rear**.**data** = obj;

**rear**.**next** = oldLast;

**if**(isEmpty()) **front** = **rear**;

**else** oldLast.**next** = **rear**;

**queueSize**++;

}*//end enqueue()*

**My dequeue() method removes elements to the queue in O(1) since it utilizes a linked list to remove elements from the queue, which is O(1)**

**The empirical analysis of the dequeue() method is c1 + 3 since one constant object is being removed from the queue and 3 slots of memory need to be allocated for this dequeue to happen.**

**dequeue Method**

*//method to delete elements from the queue*

**public** T dequeue() {

**if**(isEmpty()) **throw new** NoSuchElementException(**"Queue Underflow"**);

T obj = **rear**.**data**;

**front** = **rear**.**next**;

**queueSize**--;

**if**(isEmpty()) **rear** = **null**;

**return** obj;

}*//end dequeue()*

**My displayQueue() method prints out all queue elements to the console in O(n) time since it has to access every element within the queue in order for the queue to be printed.**

**If the queue is empty, the empirical analysis equates to c1 + 1 since null is returned and only one comparison is being made.**

**If the queue has one element, the empirical analysis equates to c1 + 2 since the top element is returned and it looks for the next element and compares it to null.**

**If the queue has more than one element, the empirical analysis equates to 5n + 3 since the for loop depends on how big the queue is and within the loop there are 5 comparisons being made which require 5 slots of memory to be allocated. Then outside the for loop there are 3 necessary memory slots that need to be allocated in order for the for loop to iterate through the queue correctly**

**displayQueue Method**

*//method to display the queue to the console*

**public void** displayQueue() {

System.***out***.println(**"Queue Data Structure: "**);

**if**(**queueSize** == 0) {

System.***out***.println(**"empty!\n"**);

**return**;

}**if**(**front**.getLinkNext() == **null**) {

System.***out***.println(**front**.getData());

**return**;

}

Node temp;

System.***out***.println(**front**.getData());

temp = **front**.getLinkNext();

**for**(**int** i = 0; i < size()-1; i++) {

System.***out***.println(temp.getData());

temp = temp.getLinkNext();

}

System.***out***.println();

System.***out***.println(**"Queue Size = "** + size());

}*//end displayQueue()*