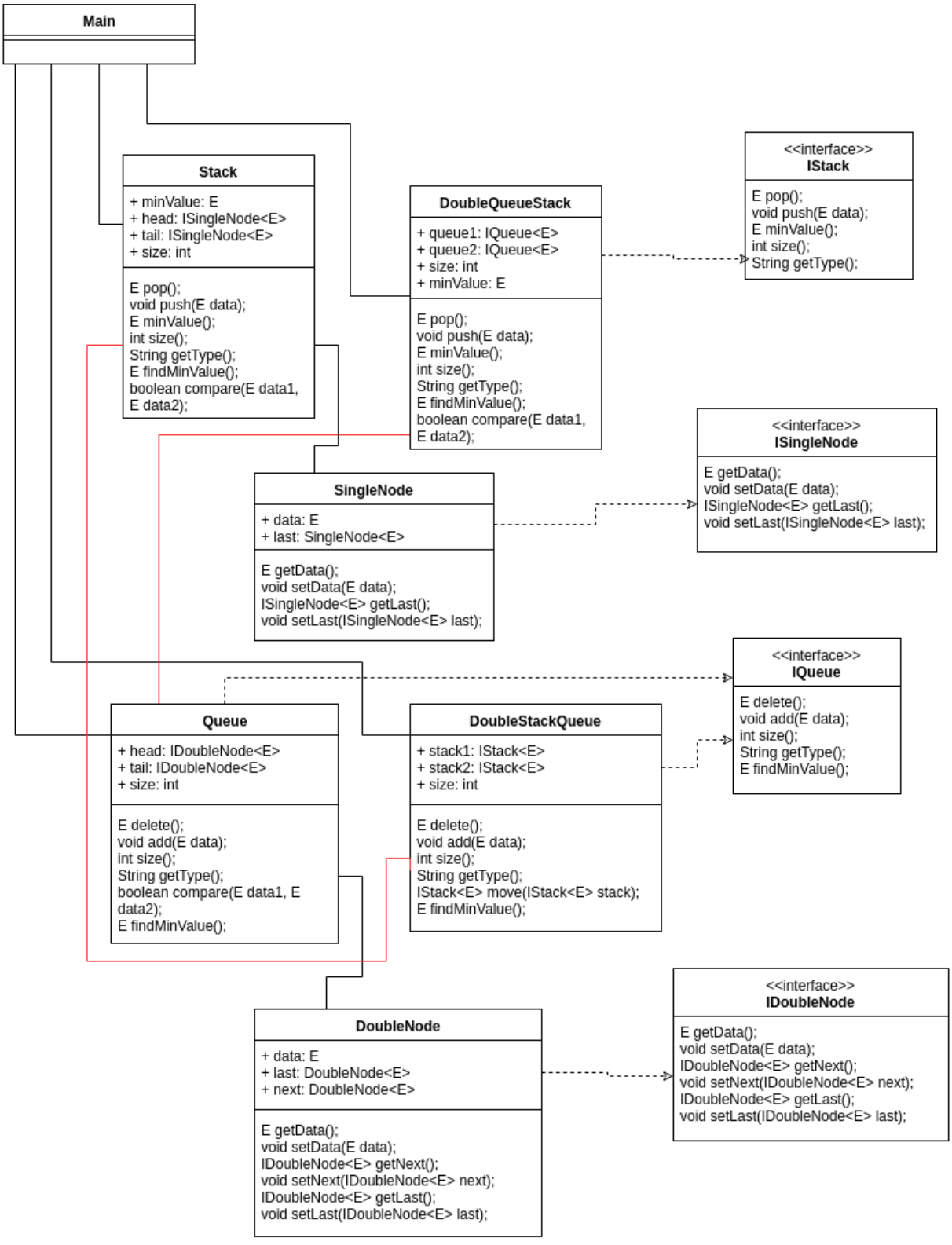
CS 3310 - Data and File Structures
Instructor: Ajay Gupta, Western Michigan University
Lab TA: Yu Guo

Tyler Thompson

abus for Fall 2016

SOFTWARE LIFE CYCLE REPORT - FOR HOMEWORK ASSIGNMENT 2

DESIGN



DESIGN JUSTIFICATION

The design of my program is laid out in the UML diagram above. The red lines should be treated the same as the black and are only red to help distinguish the difference between them and the line they cross. The main method does not show an methods because those methods are irrelevant to understanding the structure of the program. In reality the main method runs various tests through the different data structures. I have also chosen to write two types of node classes, one to be used with a doubly linked list and another to be used with a singly linked list. This was unnecessary, but increased my stack implementation efficiency on space very slightly. The function "findMinValue()" had to be added to the queue implementation in order for the DoubleQueueStack class to fully support the "minValue()" function.

THEORETICAL COMPLEXITY ANALYSIS

- **Stack**

  The stack I have implemented in my program uses a singly linked list design and has a space complexity of O(n). The space it consumes grows linearly with how many elements are pushed onto it.

  - *Pop / push*

    Time complexity of both the pop and push functions are O(1). Since I have implemented it with a singly linked list design, the computer just has to update pointers.

  - *MinValue*

    The minValue function has two time complexities depending on the situation. As the computer pushes elements into the stack, it keeps track of the minimum value with time complexity O(1). As soon as the computer pops the minimum value though, the computer will the have to search through the stack to find the new minimum value the next time it is called. This time complexity is O(n).

- **Queue**

  The queue I have implemented in my program uses a doubly linked list design and has a space complexity of O(n). As you add elements to the queue, the space it consumes grows linearly.

  - *Add / delete*

    Time complexity of both the add and delete functions for the queue are O(1). Just like the stack, the queue is implemented with a linked list design, except it utilizes a doubly linked list. To add or delete elements the computer just has to update pointers.

- **Stack implemented using two queues (DoubleQueueStack)**

  Using a stack the has been implemented using two queues is a bad idea, but shares the same space complexity as a normal stack. The only difference is that the two queues are implemented using a doubly linked list design, which would have twice as many pointers compared to a the normal singly linked list stack design. Though you have to use two queues to perform the necessary actions, the space in use stays the same as a normal stack. Since the queues are implemented using a linked list design they can grow and shrink dynamically based on how many elements are in the list.

  - *Pop*

    Like a normal stack, the pop function of the stack implemented using two queues has a time complexity of O(1). The data is stored in one of the queues in the correct order for a stack and just has to have to pointers updated in order to have an element popped.

  - *Push*

    Unlike the pop function, the push function has a time complexity of O(n). In order to push an element into the stack in the right order, all elements in the queue holding the data have to be move into the second queue temporarily. Then the new element is added to the first queue and finally all the elements are moved back into the first queue. The function for this time complexity would be T(n) = 2n+1. Which in big-O is O(n).

  - *minValue*

    Like in a normal stack, the minValue function has two time complexities depending on the situation. As the computer pushes elements into the stack, it keeps track of the minimum value with time complexity O(1). As soon as the computer pops the minimum value though, the computer will the have to search through the stack to find the new minimum value the next time it is called. This time complexity is O(n).

- **Queue implemented using two stacks (DoubleStackQueue)**

  Using two stacks to implement a queue is also a bad idea, but still shares the same space complexity as that of a normal queue. Deleting an element from a queue like this has a time complexity of O(n) compared to a normal queue which is O(1). One advantage though is a slit increase in space efficiency. Since the two stacks are implemented using a singly linked list design, they only need one pointer per node as opposed to a normal queue which requires two. Just like moving all the data from one queue to another, move data between two stacks does not require any extra space since they are implemented using a linked list design. The computer just has to update the pointers and the lists can grow and shrink dynamically.

- *Add*

  Like a normal queue, adding an element to a queue implemented using two stacks has a time complexity of O(1). Only one of the stacks actually holds the data and since it is implemented using a linked list design, only pointers have to be updated by the computer.
- *Delete*

  Unlike a normal queue, the time complexity of the delete function is O(n). In order to remove the right element from the queue, the computer first has to pop all the data in the stack holding the data into a second temporary stack. The the element to be removed is popped off the temporary stack and finally all the elements are popped off the temporary stack back into the first stack. The function for this time complexity would be 2n+1 or in big-O, O(n).

EMPIRICAL COMPLEXITY ANALYSIS

The following chart shows an empirical analysis of the runtime on each data structure. The runtimes are calculated with 114380 elements and is an average of 5 runs calculated in milliseconds. The best case runtime is when the minimum value is in the list somewhere and not popped before calling the minValue function. The worst case runtime is when the minimum value is popped and the next minimum value is at the beginning of the list, so the program has to search the entire list for the new minimum value. Since the queue data structures do not need to include a minimum value function, these were not calculated for them. In these runtime tests all the elements in the list were added or pushed to the data structure and then all of them were popped or deleted from the data structure. This table clearly shows which data structures are the fastest and which are the slowest.

Data Structure	**Runtime Best Case**	**Runtime Worst Case**
Stack	101 ms	115.4 ms
Queue	77.8 ms	n/a
DoubleQueueStack	165802 ms	171852.6 ms
DoubleStackQueue	510977.6 ms	n/a