CS-202 Data Structures

Assignment 1

Linked Lists, Stacks, Queues

Due: 11pm on Sunday, Feb 23, 2020

Late Submission Policy

10% per day deduction for up to 3 days after the due date

In this assignment, you are required to implement a linked list, stack, queue and a simple PDC checkout system. You may test your implemented data structures using the test cases provided to you. **Please note that your code <u>must</u> compile at the mars server under the Linux enviroment.** Your Linux accounts have been created on the Mars server. The server is accessible from outside LUMS, therefor, you can log into Mars from home too.

Here is how you can access the Mars server:

Mars hostname: mars.lums.edu.pk

Mars port number: 46002

Username: your-roll-number (e.g., 19100132)

Password: your-roll-number

You can log into Mars using ssh (ssh -p 46002 mars.lums.edu.pk) or telnet. Start

early as the assignment would take time.

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee

PART 1:

DOUBLY LINKED LIST

In this part, you will be applying what you learnt in class to implement different functionalities of a linked list efficiently. The basic layout of a linked list is given to you in the LinkedList.h file.

The template ListItem in LinkedList.h represents a node in a linked list. The class LinkedList implements the linked list which contains pointers to head and tail and other function declarations. You are also given a file, test1.cpp for checking your solution. However, you are only allowed to make changes in LinkedList.cpp file.

NOTE: When implementing functions, pay special attention to corner cases such as deleting from an empty list.

Member functions:

Write implementation for the following methods as described here.

- LinkedList():
  - Simple default constructor.
- LinkedList(const LinkedList<T>& otherList):
  - Simple copy constructor, given pointer to otherList this constructor copies all elements from otherList to new list.
- ~LinkedList():
  - Never forget to deallocate the memory!
- void InsertAtHead(T item):
  - Inserts item at start of the linked list.
- void InsertAtTail(T item):
  - Inserts item at the end of the linked list.
- void InsertAfter(T toInsert, T afterWhat):
  - Traverse the list to find afterWhat and insert the toInsert after it.
- ListItem<T> *getHead():
  - Returns pointer to the head of list.
- ListItem<T> *getTail():
  - Returns pointer to tail.
- ListItem<T> *searchFor (Titem):
  - Returns pointer to item if it is in list, returns null otherwise.
- void deleteElement(T item):
  - Find the element item and delete it from the list.
- void deleteHead():
  - Delete head of the list.
- Void deleteTail():
  - Delete tail of the list.
- int length():
  - Returns number of nodes in the list.

PART 2:

STACKS AND QUEUES

In this part, you will use your implementation of a linked list to write code for different methods of stacks and queues. As Part 1 is a pre-requisite for this part so you must have completed Part 1 before you attempt this part.

STACK:

Stack class contains LinkedList type object. You can only access member functions of LinkedList class for your implementation of stack (and queue).

Member functions:

Write implementation for following methods as described here.

- Stack():
  - Simple base constructor.
- Stack(const Stack<T>& otherStack):
  - Copies all elements from otherStack to the new stack.
- ~Stack:
  - Deallocate any memory your data structure is holding.
- void push(T item):
  - Pushes item at top of the stack.
- T top():
  - Returns top of stack without deleting it.
- T pop():
  - Return and delete top of the stack.
- int length():
  - Returns count of number of elements in the stack.
- bool isEmpty():
  - Return true if there is no element in stack, false otherwise.

QUEUE:

Member functions:

Write implementation for following methods as described here.

- Queue():
  - Simple base constructor.
- Queue(const Queue<T>& otherQueue):
  - Copy all elements of otherQueue into the new queue.
- ~Queue():
  - Deallocate any memory your data structure is holding.
- void enqueue(T item):
  - Add item to the end of queue.

- T front():
    - Returns element at the front of queue without deleting it.
- T dequeue():
    - Returns and deletes element at front of queue.
- int length():
    - Returns count of number of elements in the queue.
- bool isEmpty():
    - Return true if there is no element in queue, false otherwise.

PART 3:

PDC Checkout Lines

In this part, you will use your implementation of a list, stack and queue to design a fair system to serve the students standing at the checkout counter at PDC.

Assume that the PDC counter allows two simultaneous checkout lines at a single counter (served by one cashier). Students are served from the front of the line and join at the back.

Wait System: For fairness purposes, a new person entering the line should have a longer wait than everyone else already in either of the two lines. When the cashier has to decide between serving the students at the front of both lines, he/she picks the one that has been waiting for longer. So, if one of the checkout lines was empty and the other had 20 people waiting in it, joining the empty checkout line would still mean waiting for all the other 20 people to be served first since they had all been waiting longer than this person who has just joined.

Functionality wise, how do you think your two checkout lines differ from a single checkout line. Are people being served faster or slower on average this way, or is there no difference?

Alternate Wait System: The checkout system is making the students unhappy, so the Student Council steps in and decides to introduce a different method. Now the cashier alternates between serving the two lines, ignoring wait time (you'll still have to implement the *wait_stamp* variable for the later parts).

Implement joinCheckoutLine function which ensures minimum wait time.

History: In the CheckoutLine.h file provided to you, *person_node* represents a person in the checkout line and the *wait_stamp* contains the information of how long they have been waiting, and *time_stamp* is an incremental value which indicates when they were served. The first person served has *time_stamp 1*, the second one served *2* and so on. Implement a *history* function which takes in an input n and returns the last n people served ordered by last served. Think about which data structure would best suit your needs.

The serving history should not be altered after returning from this function i.e. if history(n) is called twice, same result should be returned.

If 4 people have been served so far e.g. (person A, time_stamp: 1) (person B, time_stamp: 2) (person C, time_stamp: 3) (person D, time_stamp: 4)

history(3) => returns: [D, C, B]

Compare waits: Compare the waiting times of the two serving systems. Add 10 people to one checkout line, and 10 to the other and write a function to compare their wait times. The history function and time_stamp may prove useful. Does one always perform better than the other? Which serving system is better? (Answer in the form of comments in your file).

We are leaving it up to you to decide between stack or queue as containers for your system. You have studied in class, characteristics of both containers and the advantages and disadvantages they have over each other. Use this knowledge to make your choice. You need to come up with at least one argument in favor of whichever data structure you choose.

NOTE: You cannot traverse the stacks/queues you use in this part, only use the functions provided in their classes (pop, dequeue etc.). It is suggested keeping track of a global time variable to implement the wait system. **Don't edit the header files**.

Member functions:

Write implementation for following methods as described here.

- CheckoutLine(int size):
  - Simple base constructor, where size is the maximum size of both checkout lines.
- ~CheckoutLine ():
  - Deallocate memory.
- bool joinCheckoutLine (string name, int line_number, int type):
  - Given a person name and line_number (1 or 2), a person can get in one of the two lines if they are not full. Return True if successfully joined and False otherwise.
  - Type can be 1 or 2 where 1 implements the first serving system and 2 implements the alternate waiting system.
- void serveCheckoutLine (int type):
  - This function is the core of the Checkout system. When a person is served, you will store their information in the relevant data-structure to maintain a serving history (next part).
  - Type can be 1 or 2 where 1 implements the first serving system and 2 implements the alternate waiting system.
- std::vector<string> history(int n):
  - returns an array of the last n people served.
  - e.g (person A, time_stamp: 1) (person B, time_stamp: 2) (person C, time_stamp: 3) (person D, time_stamp: 4)
    history(3) => returns: [C, B, A]
  - History should be unchanged after returning from this function.

You need to figure out how you will be implementing the above functionalities. You are only allowed to use **three and only three containers** which should be declared in the CheckoutLine.h file. Don't make any other change in the header file(s).

Happy Coding :)

SUBMISSION CONVENTION:

ZIP all the files and name the file as PA1_rollnumber.zip.