Data Structures

Assignment 1

Linked Lists, Stacks, Queues

Due: 11pm on Tuesday, Feb 19, 2019

Late Submission Policy

This assignment can be submitted till 11pm on Friday, February 22nd, 2019 with a 10% penalty per day.

In this assignment, you are required to implement a linked list, stack, queue, and a simple version control system.

You may test your implemented data structures using the test cases provided to you. **Please note that your code must compile at the mars server under the Linux environment.**

Your Linux accounts have been created on the Mars server. The server is accessible from outside LUMS, therefore, you can log into Mars from home too.

Here is how you can access the Mars server:

Mars hostname: 203.135.62.10

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 assignment, 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 (T item):
  - 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 attempted Part 1 before you attempt this part.

STACK:

Stack class contains LinkedList type object. You are allowed to access all members of LinkedList 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:

VERSION CONTROL SYSTEM

Version Control Systems (VCSs), such as Git and Apache Subversion, are tools that help in the management of project files by keeping track of changes made to the source code over time. You have been coding for quite a while now and you will understand how irritating it becomes when you make changes to your stable code to add some new modification but end up destroying functionality of the stable methods as well. A VCS keeps track of these changes and allows you to turn the clock around and return to a version of your choice. While VCSs like Git are designed to manage huge repositories of files, however, for simplification we will design a VCS which only keeps track of changes in a *single* file. In this part, we will use our implementation of a list, stack and queue to design a simple VCS that keeps record of changes you make in a file.

In the VCS.h file provided to you, VCS_node is the individual node in your version control system. Message string is for any message you want to associate with the change such as '*Changed Dijkstra algorithm with Bellman-Ford algorithm*', whereas the time_stamp contains the information of when certain change was committed, in your case treat time_stamp as version of your file i.e., start with 0 and increase it discretely with every commit (0,1,2…).

We are leaving it up to you to decide between stack or queue as containers for your VCS. 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 need to specify what version the user is currently on. To specify the current version of the file, you can either use top/front of whichever data structure you are using or create a separate variable.

Member functions:

Write implementation for following methods as described here.

- VCS(string fname):
    - Simple base constructor, fname contains the name of original file you will be keeping track of.
- ~VCS():
    - Deallocate memory.

- void commit(string message):
    - This function is the core of VCS. When commit is called, you will read contents of original file and store them in the container (stack/queue). Also, make a separate version of this file and write contents of the commit to this version. For example, for a file with file named "random", whenever commit is called you will make a newer version of this file with file name convention of filename+time_stamp+.txt and write contents of original file at that point to this newer version [e.g. random0.txt, random1.txt, random2.txt …].
- void status():
    - Prints the information (Associated message and version number) about all nodes on the VCS. For checking purposes, you may print information of last 5 commits only.
- string undo():
    - Undo method takes user to one version back without losing information of the current version. For example, if you had 3 versions v1, v2 and v3 in order and you were on v3 currently, the undo method will return you to v2 but it will not lose v3. Retrieve the version number from node, read the file content associated to that version number, write them to original file and return contents of that file in a string.
- string redo():
    - Redo method takes user one step forward. Building on the example from previous paragraph, if you were on v2, calling redo will return v3 back without losing information of v2. Rest of the implementation will be like undo method.
- string revert(int t_stamp):
    - Revert method finds the version with time_stamp having value t_stamp and returns data stored in that version to user. While doing so it does not alter order in which nodes were already placed in VCS. For Example: if you had 5 versions v0, v1, v2, v3 and v4 and you were currently on v4, calling revert(2) will return you content of v2 and if you call undo from here you will be returned contents of v1. Similar to undo and redo, you will have to update original file according to version number as well.

Note: You need to figure out how you will be implementing the above functionalities. You are only allowed to use just two containers declared in the VCS.h file. Take this as a hint.

Happy Coding □

SUBMISSION CONVENTION:

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