Data Structures

Assignment 1

Linked Lists, Stacks, Queues

Due: 11pm on Friday, Feb 23, 2018

This assignment has two parts. In the first part, you will implement: (i) a **doubly** linked list and (ii) the **`Set`** ADT using a linked list. In the second part, you will implement: (i) the **`stack`** and **`queue`** ADTs using your linked list implementation and (ii) solve some problems using these implementations.

You may test your implemented data structures with the test cases that we have provided you (test1.cpp, test2.cpp, test3.cpp, test_isPalindrome.cpp and test_towers.cpp. **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 (mars.lums.edu.pk). 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: 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 using a free client like **putty** (on Windows machines), which you can download from https://www.putty.org/ or using the Terminal application in Linux and Mac.

Start early as the assignment would take time.

The course policy about plagiarism is as follows:

1. Students must not share their program code with other students.
2. Students must be prepared to explain any program code they submit.
3. Students must indicate any assistance they received.
4. Students cannot copy code from the Internet
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

Note: *In the following write-up, anything in green either refers to something in the actual code or a file name.*

PART 1

TASK 1 (Linked List):

In this part of the assignment, you are required to implement a **doubly** linked list that contains various member functions. The basic layout of this linked list class is provided to you in *LinkedList.h.*

The Template ListItem in the LinkedList.h file represents a node in the linked list. The class LinkedList (see LinkedList.h) implements the linked list, which contains a pointer to the head of the linked list and function declarations. Test cases for the LinkedList class are provided in test1.cpp, make sure you pass all of them before submission.

NOTE: *When implementing the member functions, make sure you handle all error conditions such as deletion from an empty linked list*.

Member functions:

This section defines the purpose of the member functions in the List class for which you must write the code

- List()

This is simply the default constructor of the List class

- List(const List<T>& otherList)

This is the copy constructor of the List class which when provided with a pointer to another list otherList constructs a linked list with the same elements as otherList.

- ~List()

Destructor for the List class. Deletes all the nodes in the list and frees the memory allocated to the linked list

- void insertAtHead(T item)

Function which inserts an item of type T (remember the basic unit is a template) at the head of the list.

- void insertAtTail(T item)

This function inserts an item of type T at the tail of the list.

- void insertAfter(T toInsert, T afterWhat)

This function first goes through the linked list to find the item afterWhat. When this is found it then inserts toInsert after the afterWhat linked list node.

- void insertSorted(T item)

This function adds item in its correct position in a linked list which is already sorted in ascending order (i.e., the head would have smallest value and tail would have the largest value). Note that after the insertion, the resulting list should remain sorted.

- ListItem<T> *getHead()

Returns the head pointer of the linked list, returns NULL if empty.

- ListItem<T> *getTail()

Returns the tail pointer of the linked list, returns NULL if empty.

- ListItem<T> *searchFor(T item)

Returns the pointer to node contain the element item. The function returns NULL if the list is empty or the item is not found.

- void deleteElement(T item)

This function deletes the first node containing the element item.

- void deleteHead()

Deletes the first node (i.e., the head) of the linked list.

- void deleteTail()

Deletes the tail node of the linked list.

- int length()

This function returns the length of the linked list i.e., the number of nodes currently in the linked list.

- void reverse()

This function reverses the order of the linked list (Note: Nodes must be swapped, NOT Values):
e.g., 1, 2, 3, 4, 5 ------> 5, 4, 3, 2, 1

- void parityArrangement()

This function changes the order of nodes in the linked list in the following way: e.g., 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ------> 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

i.e., all the odd indexed nodes will be together at the front of the list, followed by all the even indexed nodes.

- bool isPalindrome()

Given a doubly linked list, you are required to figure out if it is a palindrome or not. This function should return 1 if a linked list is a **palindrome**, otherwise return 0.

A palindrome is a word or phrase (in our case a list of numbers) that reads the same, backwards and forwards. For example:
5->8->2->7->2->8->5

For this function, ***your solution must be in O(n) time.***

TASK 2 (Set):

In this part, you are required to implement the **`MySortedSet`** `class`. The **`MySortedSet`** is a container that stores **unique** elements in a specific (sorted) order. The value of the elements in a `set` cannot be modified once in the container but they can be inserted or removed from the container. To test your implementation, test2.cpp has been provided.

You will be required to implement the following member functions:

- MySortedSet()

Default constructor of the set class.

- ~MySortedSet()

Destructor of the set class.

- ListItem<T>* getListHead()

Returns a pointer to the **head** of the list in the set

- int insert (T ele)

Insert ele into the set. The function should return 1 if the insertion is successful and 0 if it is a duplicate.

- int deleteEle (T ele)

Remove ele from the set. The function should return 1 if the deletion is successful and 0 if the set did not have ele in it.

- int isEmpty ()

Returns 1 if there are no elements in the set and 0 if there are elements in the set.

- int isMember (T ele)

Returns 1 if ele is a member of the set and 0 otherwise.

- int isEqual (MySortedSet<T>* anotherSet)

Returns 1 if the set contains the same elements as anotherSet (passed as a parameter to this function).

- int isSubset (MySortedSet<T>* anotherSet)

Returns 1 if anotherSet is a subset of this set and 0 otherwise.

- int count()

Returns the number of elements in the set. (**Think:** *Can you do it in O(1)?*)

- MySortedSet<T>*set Union(MySortedSet<T>* anotherSet)

Returns a **third** set that contains union of this and anotherSet. Union of two sets contain all distinct elements contained in **either** of the two sets.

- MySortedSet<T>* setIntersection(MySortedSet<T>* anotherSet)

Returns a **third** set that contains the intersection of this and anotherSet. The intersection of two sets consists of all distinct elements that are contained in **both** sets.

- MySortedSet<T>* setDifference(MySortedSet<T>* anotherSet)

Returns a **third** set that contains the set difference of this and anotherSet. The set difference of two sets will contain all distinct elements that are in this but not in anotherSet.

PART 2

TASK 3 (Stacks and Queues):

For this task, you will use your linked list class to implement the stack and queue data structures.

Stack:

The Stack class contains a List type object. This means that all the member functions defined in the Linked List class would be accessible. (The same is the case for the Queue class)

Member functions:

This section defines member functions of the Stack class for which you have to write the code given in the header file stack.h.

Stack()

This is the default constructor of the Stack class.

Stack(const Stack<T>& otherStack)

This is the copy constructor for the Stack class which when provided with a pointer to another Stack, constructs a new Stack with the same elements as the existing otherStack object.

~Stack()

Destructor for the Stack class. Be sure to free any allocated memory.

void push(T item)

Function which inserts elements in the Stack object (which follows the Last-In First-Out (LIFO) policy for insertions and deletions).

T top()

Function which returns the value on top of the Stack i.e., the element that was last inserted. This function only returns the element without removing it from the Stack.

T pop()

Function which returns the value on top of the Stack (the element last inserted into the Stack) and in addition, removes the element from the Stack.

int length()

This function returns the number of elements in the Stack.

bool isEmpty()

This function returns true if the stack is empty and returns false otherwise.

For a few bonus marks, copy the stack cpp file and re-implement it such that the stack class maintains a ***set of stacks*** in its inner working. The user is presented the interface in such a way that the user still thinks there is a ***single*** stack present but on the inside, there is a set of stacks. Once a given number of elements are inserted in the given stack, a new stack is created and the further insertions would be done on the new stack.

Queue:

The Queue also contains a List type object, just like the Stack.

Member functions:

Queue()

This is the default constructor of the Queue class.

Queue(const Queue<T>& otherQueue)

Copy Constructor for the Queue class which when provided with a pointer to another Queue contructs a new Queue object with the same elements as the existing otherQueue object.

~Queue()

Destructor for the Queue class. Be sure to free any allocated memory.

void enqueue(T item)

Function which is used to add elements to the Queue object (which follows the First-In First-Out (FIFO) policy for insertions and deletions).

T front()

Function which returns the element at the front of the Queue i,e., the element in the queue which was inserted first. This function only returns the element without removing it from the Queue.

T dequeue()

Function which returns and removes the element at the front of the Queue.

int length()

This function returns the number of elements in the Queue.

bool isEmpty()

This function returns true if the queue is empty and false otherwise.

TASK 4 (Towers of Hanoi using stacks):

The **`Tower of Hanoi`** is a mathematical game or puzzle. It consists of three rods, and several disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.
The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

I. Only one disk may be moved at a time.
II. Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
III. No disk may be placed on top of a smaller disk.

Your task is to solve this puzzle using the stack class you implemented in the previous part. See towers.cpp for further details.