COP 3530 – Data Structures and Algorithms I

Project 3

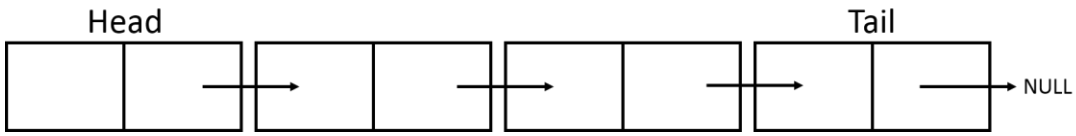
Due: Friday, March 4 11:59 P.M. CST

Objective:

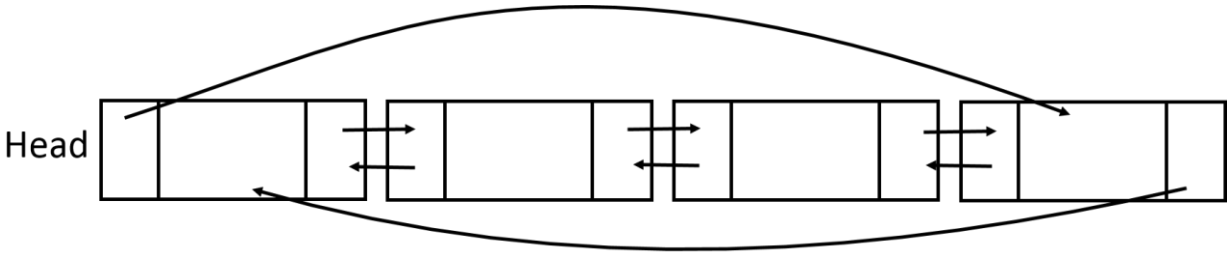
This project will build upon your knowledge of linked lists. In this project, you will need to create an abstract data type for representing a doubly linked list with the added requirement that the list is circular. This data structure is similar in some ways to the deque that was created in the previous project. However, while the deque was of a set size, the list will be able to grow.

Problem Description:

You have been given an implementation for a singly linked list on eLearning. A singly linked list can be represented in the following way:



Note that this is the same linked list that we have been going over in class. We keep track of a head and a tail node for easy access to the beginning and the end of the linked list. Each node in the linked list contains a single pointer to the next node in the list. This list is noncircular as can be seen with the final node which points to NULL indicating the end of the linked list. In contrast, a circular, doubly-linked list can be represented in the following way:



Note a few of the changes that have been made. Now each node not only contains a link to the next node in the linked list, but it also points to the previous node. We no longer keep track of the tail node. The reason for this is that with the circular list, the last node's next will be the head of the linked list, while the head's previous will point to the tail of the list. You can traverse this list in both the forward direction using next and in the backwards direction using previous. Your project will be to implement this type of linked list using the code for the singly linked list as your initial starting point.

Your linked list should have the same functions as the previous implementation with the following changes:

- You should update the list implementation to properly manage previous pointers for your nodes. This also includes correctly creating nodes that have previous pointers.
- A list with only a single node should have that node's next and previous pointers point to itself.
- You should remove code relating to the tail of the list as our new list only keeps track of the head. This means making the appropriate changes to each function. You will need to determine what these changes are.
- Your insert, removeAt, and get functions should now work with negative indices. Negative indices mean to traverse the list in the reverse direction. As an example, get(list, -1) would get the last element in the linked list. The call get(list, -2) would get the element before the last element in the linked list. **You may not use mod operations to do this. Your list must be circular.**
- Your insert, removeAt, and get functions should now be able to handle indices that are greater than or equal to the size of the list. As an example, get(list, 4) for a list with only 4 elements would get the first element in the list while get(list, 5) would get the second element in the same list. **You may not use mod operations to do this. Your list must be circular.**
- As with the current implementation, inserting a value with insert(list, size, value) in a list with size number of elements should insert a value at the end of the linked list.

You should also add the following function to your linked list implementation:

- void printListFromEnd(List) – Will print the list in reverse order.

In addition to these changes that will need to be made to the list implementation, you will also need to make changes to the node ADT. You will need to add a pointer for the previous node to the node struct. Your createNode function will need to change accordingly. In addition, you will need to add the following functions:

- Node getPrevious(Node) – Will return the Node pointed to by the previous pointer for the given Node.
- void setNode(Node, Node previous) –Will set the previous Node for the Node given as the first argument.

Creating Your Client

Write a program that displays a menu on the screen for the user to perform operations on your linked list. Each operation, except for createList and freeList, is a separate item in your menu. Your list should be initialized before the menu is shown. An Exit menu item will allow the user to exit from your program. At the time of the exit, the list must be deleted using the corresponding operation before the program terminates. Your menu should allow users to perform the following operations:

1. Get the current size of the linked list.
2. Prepend an item to the linked list.
3. Append an item to the linked list.
4. Insert an item into the linked list.
5. Remove an item from the linked list.
6. Get an item at a specific position in the linked list.
7. Print the list in the forwards direction.
8. Print the list in the reverse direction.
9. Exit from the program.

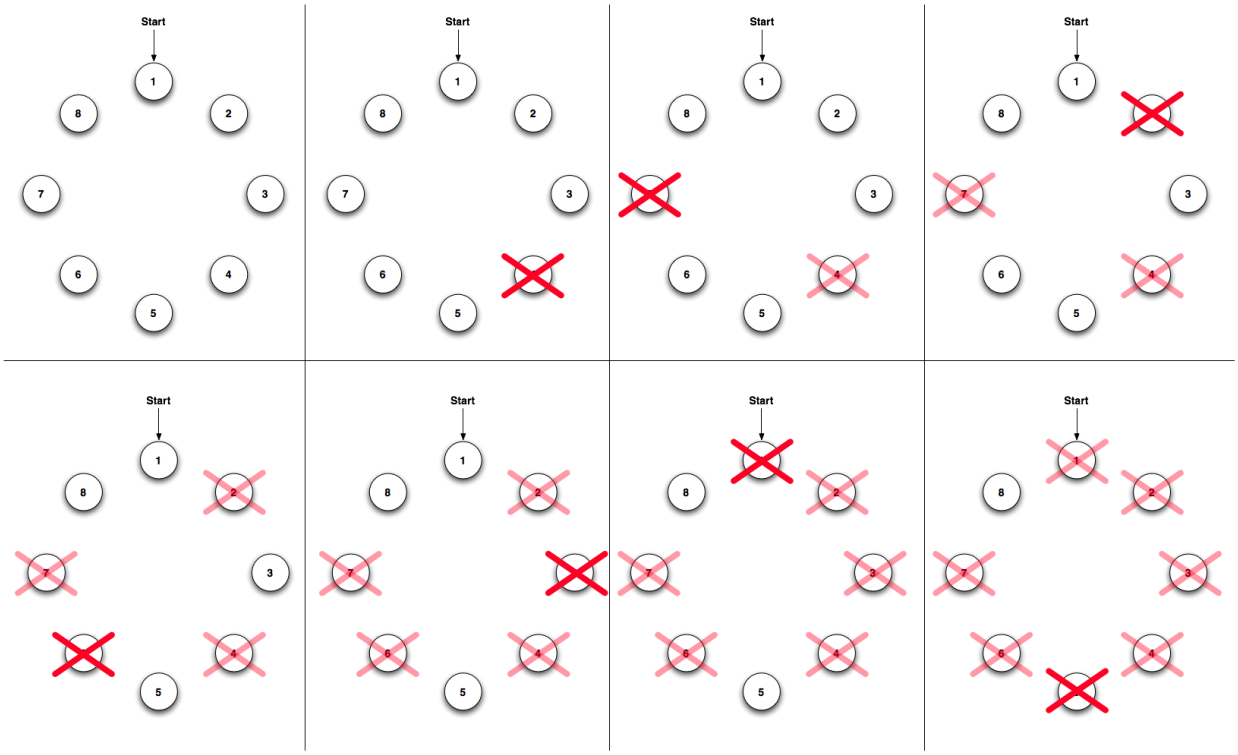
For each of the above operations, the program should prompt for any additional arguments. As an example, option 2 will prompt for an integer to insert while option 4 will prompt for an index and then for a value. The menu should be redisplayed after every operation is performed. If any operation fails to be performed, you must print an appropriate message.

Additional Tasks:

The following questions should be answered in addition to submitting your program. The first two questions are designed to help you understand the relationship between the linked list and the previous data structures discussed in this course. The second two questions are designed to give you example problems that can be solved using circular linked lists. You can see an example of the last question on the next page.

1. Explain how you would use your implementation of the linked list to implement a stack, queue, and deque.
2. Explain the benefits and consequences of using a linked list over an array for these data structures.
3. Suppose that I gave you a sequence of numbers to insert into your circular list. Then, I asked you to determine whether another sequence of numbers appeared in your circular list in either the forward or backwards direction. As an example, suppose your linked list contained 1 2 3 4 5 and I asked if 2 1 5 appeared in the list. In this case, the sequence would appear and it would appear in the reverse direction, **1 2** 3 4 **5**. How would you write a function that tested for this?
4. Describe how you would use your linked list implementation to solve the Josephus problem. The Josephus problem can be described as imagining that N people have decided to hold an election by arranging themselves in a circle and eliminating every Mth person around the circle, closing ranks as each person drops out and continuing until only one person remains. You want to be able to print the order in which each person drops out and the final winner.

An example of the Josephus problem where M = 3.



Steps for Completion:

1. Update the node ADT to properly handle previous pointers. Be sure to properly comment the header file. (15 points)
2. Update the linked list ADT from the singly linked implementation to the circular, doubly-linked implementation. Be sure to update the comments in your header. (50 points)
3. Create the client for your linked list. You should adequately test your program with multiple inputs. (15 points)
4. Complete the additional tasks. (20 points)

In addition, **please add your name as a comment to the top of each file.**

Submission Instructions:

Make sure that before submitting your program that you have tested it on the ssh server (*ssh.cs.uwf.edu*). There is information for how to do this on eLearning.

For this project, submit a zip file containing all files required to run your project these files will be named as the files for the previous list and node ADTs. The client should be named list_client.c. Submit them along with a document containing the results to the additional tasks (name the file answers, and submit it as a doc or docx file). Submit your project using the eLearning dropbox. Remember that all students who turn in their projects by the due date that receive a B or better will receive an added 0.5% on their final overall average.