Homework 3 – Singly Linked List

Topics: This set of problems is based on concepts from **Chapter 3.2** of your Data Structures & Algorithms textbook, **Chapter 17** of your C++ Early Objects book, and **Chapter 3.1** of Open Data Structures book. Make sure to read through the book chapters and review your lecture notes before you attempt this exercise. Please be sincere in your attempt. You should expect to see similar problems on your exam.

Points: Counts towards your participation grade. **You must make an honest attempt at all problems to get a full participation grade.**

Team-based? Individual

Due: **Thursday, 2015-Feb-26 @ 1pm in class.** Late work will not be accepted.

Hand in details: You must turn in your work as a printed or handwritten paper copy. Emailed copies will not be accepted. Please write legibly. Provide your name in your hand in.

Caution: **Cases of plagiarism will be dealt in a strict manner and according to the guidelines described on the course syllabus and university policies on academic dishonesty.**

Getting Started: Problems 1-5 are based on the code in hw3.cpp available for you to download from Titanium.

Problem 1:

Activity 1 (a):

Modify the SinglyLinkedList class downloaded by you to add a print member function that will print the contents of the info field for each node. Prototype: `void printList();`

From your main create a linked list with three nodes and info fields as 5, 20, and 15 respectively. Next create another linked list with three nodes and info fields as 50, 200, and 150. Then call your printList function to display each list as below:

```
First list: 5 20 15

Second list: 50 200 150
```

Activity 1 (b):

From the previous activity you are asked to absorb the second list into the first list such that the first list contains all of its nodes followed by all the nodes of second list. Your algorithm should not create any new nodes. Once the second list is absorbed it's head pointer should be set to null. Modify the SinglyLinkedList class to add a member function that takes a singly linked list named second as an

argument and absorbs it into the first list. Your first list is the one that calls the member function absorb, see prototype below.

Prototype: `void absorb(SinglyLinkedList & second);`

Print the first list from your main function:

```
First list: 5 20 15 50 200 150
```

Include C++ code for your printList, absorb and updated main functions here along with the sample runs of your program as shown in the activities.

Problem 2:

Modify the SinglyLinkedList class to add a member function for inserting a new node at a specified position:

Prototype: `void insert(int value, int position);`

A position of 0 means that node with info set to value will become the first node on the list, a position of 1 means that node will become the second item on the list, and so on. A positon equal to, or greater than, the length of the list means that the node is placed at the end of the list.

First create a list using add from problem 1 with nodes having values 5, 20, 15, 50, 200, 150. Next print your list by calling the printList function from problem 1. Next insert node with value 77 at position 4. Call your printList function again. Below is one sample run.

```
List: 5 20 15 50 200 150

After inserting new node with value 77:

List: 5 20 15 50 77 200 150
```

First give figures for your list (hand drawn) as it looks before and after the insertion of 77 and the intermediate steps needed to insert 77, show all pointer manipulations. Next include C++ code for your insert and updated main functions here. Give three sample runs: inserting a node at the 0 position, in between and at the end.

Problem 3:

Modify the SinglyLinkedList class to add a member function for deleting a node at a specified position:

Prototype: `void remove(int position);`

A position of 0 means that the first node on the list (node pointed at by the head pointer) is deleted. The function does nothing if the value passed for positon is greater or equal to the length of the list.

First create a list using add from problem 1 with nodes having values 5, 20, 15, 50, 200, 150. Next print your list by calling the printList function from problem 1. Next delete node at position 4. Call your printList function again. Below is a sample run.

```
List: 5 20 15 50 200 150

After deleting node at position 4:

List: 5 20 15 50 150
```

First give figures for your list (hand drawn) as it looks before and after the deletion and the intermediate steps needed to delete node at position 4, show all pointer manipulations. Next include C++ code for your remove and updated main functions here. Give three sample runs: deleting a node at the 0 position, a position in between, last position, and a position that is higher than that of the last node.

Problem 4:

Modify the SinglyLinkedList class to add a member function for reversing the list in-place:

Prototype: `void reverse();`

The member function rearranges the nodes in the list so that their order is reversed. You should do this without using any secondary data structures, and without creating or destroying any nodes. This is known as in-place reverse. Note: Your Data structures and Algorithms textbook in section 3.4.2 describes a **non** in-place solution, and that is **not** what we are looking for here.

First create a list using add from problem 1 with nodes having values 10, 20, 30. Next print your list by calling the printList function from problem 1. Next reverse your list. Call your printList function again. Below is a sample run.

```
List: 10 20 30

After reversing:

List: 30 20 10
```

Firstly, give figures for your list (hand drawn) as it looks before and after the reversal and the intermediate steps needed to reverse, show all pointer manipulations. Secondly, give **pseudo-code** (refer section 1.7.2 of your Data Structures and Algorithms text book) of your reverse algorithm. Lastly include C++ code for your reverse and updated main functions here along with a sample run of your program as shown above.

Problem 5:

The search time for an item stored in a linear list necessarily varies with its positon in the list. In many applications it is advantageous to place frequently accessed items near the start of the list. However, in many situations it cannot be determined in advance which items will be accessed most often. In cases like these a dynamic reorganization of the list, based on current access activity, can reduce expected search time.

Modify the SinglyLinkedList class to add a member function search:

Prototype: `bool search(int searchInfo);`

The member function simultaneously searches and reorganizes the list in the following fashion. A node with info as searchInfo is sought. The nodes of the list are examined in turn. If the node is found, it is removed from its current position and moved to the start of the list. The return value of the function search is true if the requested element is found and false if it is not. Note you should do this without destroying or creating any nodes and without using any secondary data structure.

First create a list using add from problem 1 with nodes having values 40, 10, 30, 60, 20. Next print your list by calling the printList function from problem 1. Next call the search function to search for 60. Call your printList function again. Below is a sample run.

```
List: 40 10 30 60 20

Search for 60: found

List: 60 40 10 30 20
```

Firstly, give figures for your list (hand drawn) as it looks before and after the search and the intermediate steps needed to reorganize the list, show pointer manipulations for reorganization. Secondly, give **pseudo-code** (refer section 1.7.2 of your Data Structures and Algorithms text book) of your search algorithm. Lastly include C++ code for your search and updated main functions here along with a sample run of your program as shown above.