University of Calgary
Faculty of Science
Department of Computer Science

CPSC 331- Data Structures, Algorithms, and Their Analysis
Assignment 2
Manipulations of Data Structures
Spring 2022

1 Objectives

In this assignment you will gain experience implementing data structures as well as applying data structures for problem solving. You will implement a linked list. As part of the implementation of your linked list you will also implement several algorithms to facilitate the common operations on your linked list, such as element insertion, search, and deletion, as well as some problem-specific algorithms. Similar to assignment 1, you will need to provide some analysis of the runtime of these algorithms, however in assignment 1 you empirically tested the runtime of your algorithms, for this assignment you will need to analytically demonstrate your algorithm's time complexity.

2 Background

A local music festival has had some issues with their performer lineup. The music festival is made up of several venues, and each venue has a planned performance order which we will refer to as the "schedule", however the schedules are represented unconventionally for such an event. There is no central schedule of the performance order, instead each performer has simply been told who will follow them on stage. Some performers were told incorrectly who follows them onstage which is creating schedules that require performers to play at more than one venue, however performers playing at multiple venues is logistically not possible, so these schedules need to be fixed. In other cases, performers have been told that a performer who is scheduled earlier in the day is to follow them, in this case the schedule has a loop so that the show will never end!

3 The Problem

In some cases the same performer shows up twice so that the performance will start normally, but eventually cycle between a few performers endlessly. This can be fixed by finding the performer that is going to start the loop (in other words: the last unique performer in the schedule) and asking them to end the show instead. The other problem is that performers at different venues sometimes have been told that the same performer should follow them. Since it is not possible for performers to switch venues in a single day, this cannot happen. These problems can be represented using linked lists. In fact, the "each performer tells the next" strategy of scheduling is remarkably similar to a linked list. If the order of the bands is translated into a linked list, then some linked list algorithms might help us solve the conflicts. The two main problems that you need to solve are:

1. Remove shared nodes from two linked lists
2. Remove cycles from linked lists

Solving problems 1 and 2 will require a working implementation of several common linked list operations: length, traversal, insertion, deletion.

4 Implementation Details

The following sections are helpful in your implementation[1]

4.1 Generic Programming

For assignment 1 you most likely wrote a couple lines of code that looked something like this:

```
HashMap <Integer, Long> myMap = new HashMap <Integer, Long> ();
myMap.put(4, (long)5);
```

The class of the variable myMap and the corresponding constructor method are both generics. The part of the type declaration and constructor call in the angle brackets ⟨⟩ are called type parameters. The type parameters here determine what data type is stored in the hash map, and in general type parameters indicate what data types a method or class operates on. Another useful tool for generic programming is the abstract classes. Abstract classes are classes that provide some method signatures for which they have no implementation, such methods are known as abstract methods. An interface is essentially a class where all of the methods are abstract. Interfaces are intended to describe behavior which is later implemented in a child class that implements the interface.

[1] Please follow the directions of your TA in the tutorials sessions, and as posted on D2L shell of the course

4.2 Linked List

A linked list is a data structure that is used to store a collection of elements. Elements in a linked list are stored as a sequence of nodes where each node contains one element of the list, and a reference to the subsequent element of the list. The linked list data structure is made up of at least a node class that stores a value and the subsequent node (or null for the end) and a list class that records the head node and provides some methods on the list. A visualization of this arrangement is provided below describing a linked list that stores int values.

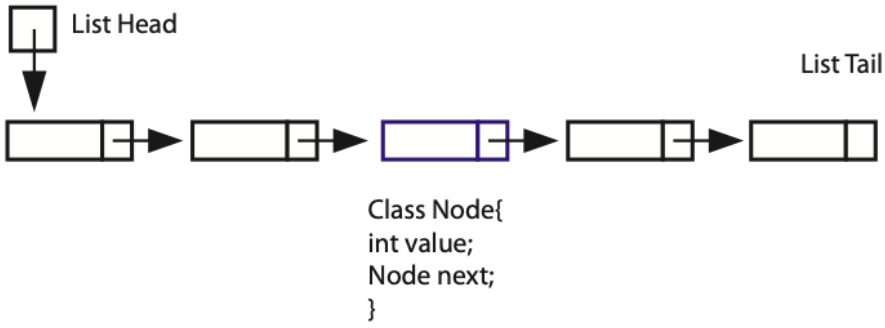

Figure 1: An example of a linked list referenced by its head. Each node of the list consists of two main items: Value and Node.

In some ways, a linked list resembles a data structure you are already familiar with: an array. An array is a data structure that holds a fixed number of values, and these values can be referred to using an index. Arrays are advantageous in that it is easy to access any element, but they are limited because resizing an array is costly, usually requiring all the elements of the array be copied into a new array of the desired size. Linked lists present a trade off on this property. Accessing elements becomes more difficult, as you must traverse the list starting at the head node until the desired element is found. Conversely, resizing the list is easy, there is no fixed size, to add an element you need only to create a new node and modify some node in the list to refer to it, the node that should be modified depends on the desired placement of the new node. A linked list class usually only needs to store the head node of the list, and provide methods that traverse the list starting at the head node.

For the linked list you will need to produce a generic java implementation of a linked list that supports the operations below. Recall that Java does not permit non-member functions meaning that every function is actually a method. The *this* variable, used below, refers to the object that the method belongs to. The following are a list of the operations,

with their respective pre-conditions and post-conditions, that need to be implemented:

1. getHeadNode

   Precondition: *this* is a linked list

   Postcondition: The head node is returned, if the list is empty null is returned instead

2. Append to tail by *value*

   Precondition: *this* is a linked list, and *value* is a variable with appropriate type for this list, which is not already present in the list

   Postcondition: The tail node of this list is a new node with node.value = *value*, the length of the list is the previous length + 1, no other nodes have been changed. If a node with the given value already exists, an exception is generated.

3. Append to tail by *node*

   Precondition: *this* is a linked list, a *node* with appropriate type for this list such that *node.value* is not already present in the list.

   Postcondition: The tail node of this list is *node*, the length of the list is the previous length + 1, no other nodes have been changed.

4. Length

   Precondition: *this* is a linked list

   Postcondition: The number of nodes in this list is returned, 0 is returned if the list is empty

5. Delete by *value*

   Precondition: *this* is a linked list and a *value* is a variable of appropriate type for the list.

   Postcondition: If a node with *node.value* = *value* is present in the list it is removed, the order of elements in the list is otherwise unchanged. If no node with node.value = value is present in the list, the list is unchanged. If the list is empty an exception is generated.

6. Search by *value*

   Precondition: *this* is a linked list and a *value* is a variable of appropriate type for the list.

Postcondition: If a node with node.value = value is present in the list, it is returned. If no node with *node.value = value* is present in the list, null is returned. If the list is empty an exception is generated.

7. toString

   Precondition: *this* is a linked list.

   Precondition: A string representing this list is returned, the string should contain all of values present in the list in the same order that they are present in the list.

The linked list should implement the **MyLinkedListInterface** interface provided in

MyLinkedListInterface.java

in the skeleton code on d2l. The interface provides an internal node class. Also make sure that your linked list class generates appropriate exceptions when illegal operations are performed, such as deleting from an empty list. The toString method should generate a string that represents all of the elements in the linked list. A nice format of such a string is:

```
value 1 -> value 2 -> value 3
```

4.3 Related Algorithms

The following are a couple of algorithms that you will be using in this implementation:

4.3.1 Common nodes in linked lists

For this problem you need to identify if two lists have common nodes, and if so, remove the common nodes from the shorter list. To identify the common nodes in linked lists start by calculating the length of the two lists. The length method of the MyLinkedList class should make this trivial (although you will have to implement that method first). Let list0 be the longer of the two lists and list1 be the shorter. As part of the precondition of this algorithm you can assume that there is at least one unique node in both lists, in other words, both lists are not identical, and the shorter list is not completely within the longer list.

Create two index nodes, one at the head of each list. Take the difference in lengths of the two lists and advance the list0 node by that many steps. Next, increment both nodes, checking to see if they are equivalent as you go (also, keep count of how many nodes you visit, it will be useful later). If the lists share some nodes the index nodes will eventually meet at the first shared node, if they do not share nodes then both index nodes will eventually reach the end of the list. To separate the lists, use the counter from the

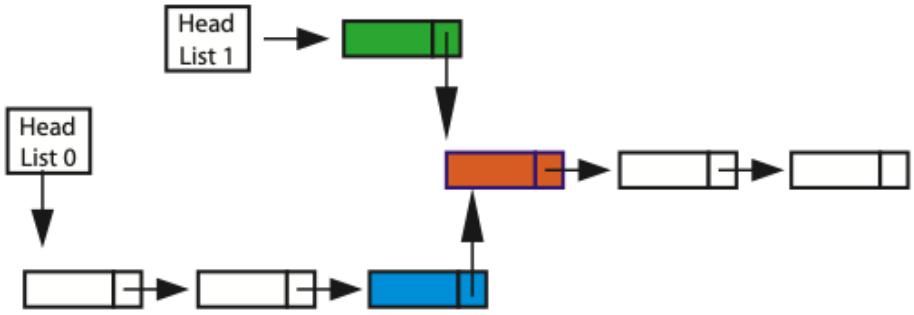

Figure 2: Two lists are indicated by Head List 0, and Head List 1, of length 6, and 4 respectively. Node 0 start indicated by the blue node, and Node 1 start is indicated by the green node. Both Node 0, and Node 1 meets at the red node.

previous steps. The counter has recorded how many steps it took to get from the top of the shorter list to the first shared node. Simply start at the top of the shorter list again and step down the list for the counter value-1 steps. This is the last non-common node in the short list, so setting its next node to null will remove the subsequent nodes from the list.
The algorithm **RemoveSharedLinkedListNodes(list0, list1)** has the following steps:

Precondition: *list0* and *list1* are linked lists, there is at least one non-shared node in each list.

Postcondition: *list0* and *list1* do not share any nodes. If they did have shared nodes to begin with, those nodes have been removed from whichever list was shorter to begin with.

1. Ensure list0 is the longer list, if it is not swap them
2. Compute the length difference but avoid calling length() more than necessary, it is costly, *i*.e. O(n).
3. Initialize two nodes and set them to the heads of each list.
4. Advance the node from the longer list (list0) by a number of nodes equal to the difference in lengths of the two lists.
5. At this point both nodes are the same number of steps from the end of both lists.
6. Advance both nodes along the lists checking equality as you go. The nodes will be equal at the first shared node between the lists.

7. If there are no shared nodes, then the nodes will both eventually be null.
8. Keep track of the number of steps it takes to reach the shared node.
9. If a shared node is found then reset the list1 iterating node back to the head of list1
10. Iterate through the list for one less steps than the steps it took to reach the shared node. This is the last node in list1 that is not shared.
11. Set its next value to null to remove the subsequent nodes from this list.

4.3.2 Floyd's Tortoise and Hare Algorithm

This algorithm can be used to identify the index of the first node in a cycle in a linked list. In principle it works by traversing the list using two pointers, a quick pointer (the hare) and a slow pointer (the tortoise). The hare moves twice as fast as the tortoise, so eventually the two will overlap, when this happens it confirms that the list does have a loop as pictured here.

Conversely, in the case that the list does not have a loop the pointers simply reach the end of the list. Next, the "period" of the loop must be identified, this is the number of nodes in the loop. This is simple, since we already have hare=tortoise from the last step you can simply let the hare continue visiting nodes[2], incrementing a counter as you go. When the hare is once again equal to the tortoise node then you have counted all the nodes in the cycle, as seen in the following diagram.

Note that the fact that the tortoise and hare met for the second time at the same node is coincidental, if you try this with a different cycle they may meet elsewhere. Next, the first node in the cycle needs to be identified, and there is a bit of a trick to it. We reset the positions of the tortoise and hare to the beginning of the list. We once again allow the hare to move ahead, but a little differently. The hare should get a "head start" equal to the period that we identified.

Put more plainly: you must advance the hare by period steps, then advance the hare and tortoise at the same rate until they are once again equal. Due to the hare's head start, the tortoise and hare will meet at the first node in the cycle. Finally, to find the last node in the cycle you can advance one of the index nodes by period-1 steps to reach the last node in the cycle. Setting this node's next property to null will remove the cycle from the list. For clarity, divide the algorithm into two methods. One to detect the cycle and identify the period, and one to remove the cycle.

Below is the pseudocode for both algorithms of detecting a cycle, and for removing a cycle:
The algorithm **detectCycleAndPeriod(list0)** could have the following steps:

Precondition: *list0* is a linked list.

[2]Note that the hare is only going one node at a time now.

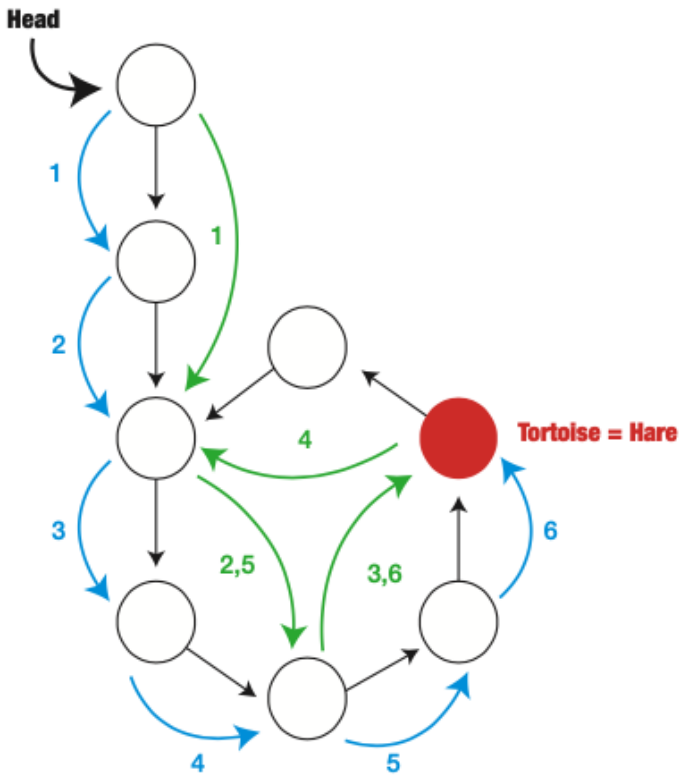

Figure 3: A snapshot of the Floyd's Tortoise and Hare algorithm.

Postcondition: If *list0* contains a cycle, the period is returned. If *list0* does not contain a cycle, 0 is returned.

1. Initialize the tortoise and hare to the head node
2. Advance the tortoise 1 node, and the hare two nodes and check if they are equal (occupying the same node).
3. Repeat this process until the end of the list is reached (no loop) or the two nodes are equal.
4. If the hare and tortoise meet then that means there is a cycle, and we should identify the period.
5. If there is no cycle then return –1 for the period.
6. To identify the period,
   (a) Let the hare continue traversing the list (one node at a time this time) until it returns to the tortoise's position.

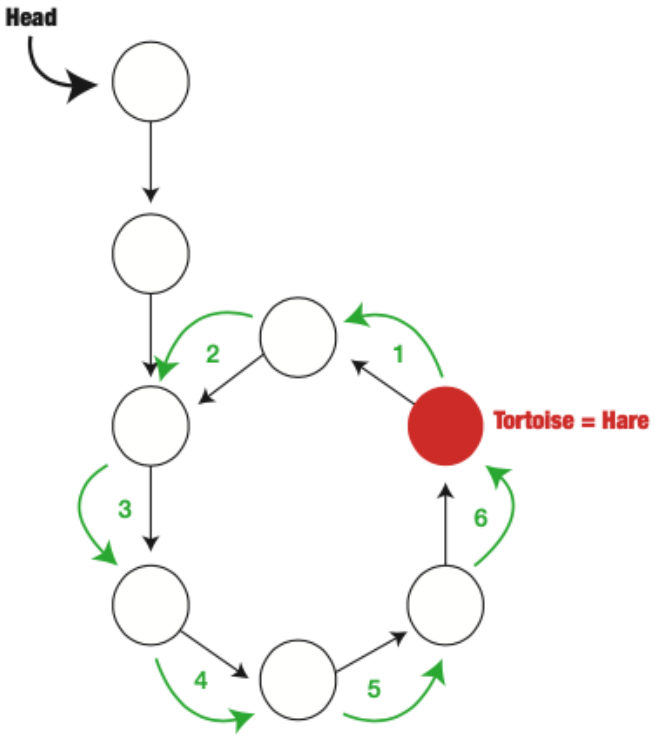

Figure 4: A snapshot of the above example showing a period 6.

(b) Count the number of steps that this takes.
(c) The count is the period
(d) Return it.

The algorithm **removeCycle(list0, period)** could have the following steps:

Precondition: *list0* is a linked list with a cycle, period is the number of nodes in the cycle

Postcondition: *list0* does not have a cycle, the order of the nodes in the list is unchanged (except for the removal of repeated nodes in the order), the number of the nodes in the list is unchanged

1. Initialize the tortoise and hare to the head node
2. Advance the hare "period" nodes into the list.

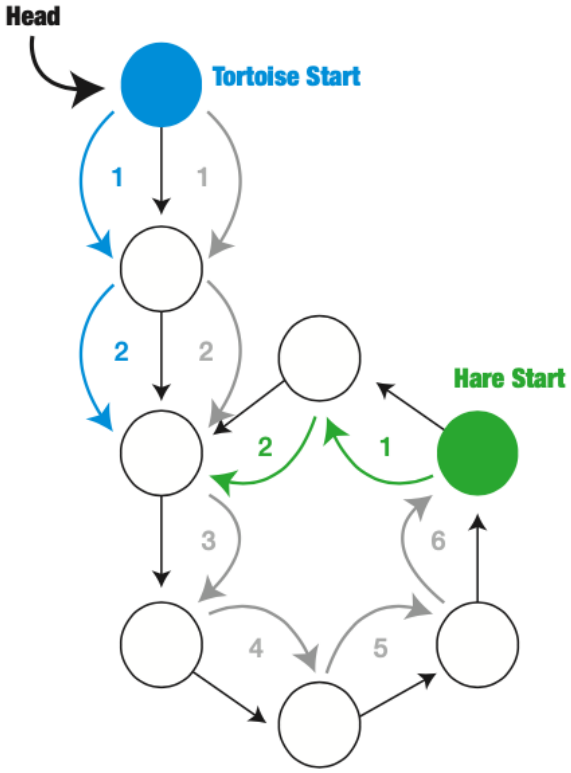

Figure 5: Another snapshot of the above example showing the Tortoise start and the Hare start in Blue, and green, respectively.

3. Advance the hare and the tortoise at the same rate, checking for equality as you go.
4. The tortoise and hare meet at the first node in the cycle.
5. Advance the hare period-1 nodes, this is the last node in the cycle.
6. Set the last node in the cycle's next value to be null.

4.4 Running Time Analysis

In addition of the above you need to provide a brief report to prove the following:

1. The runtime of Floyd's Tortoise and Hare algorithm is in O(n).
2. The runtime of the removeSharedLinkedListNodes algorithm is in O(n)

5 Grading

Grading will be assigned for all aspects, including efficient programming, documentation style, and the identification of runtime/storage complexity. The total grade of this assignment will initially be set to 51 points but will be factored in the final grade as per the outline of the course. The following guidelines will be used[3]:

Written Part This part will carry 10 points (5 points for each proof).

Linked List This part will carry 15 points, 3 points for implementing each of the following operations:

1. getHeadNode and length
2. appendToTail by value and by node
3. deletion and search by value
4. generate appropriate exeptions
5. toString

Algorithms This part will carry 24 points distributed as follows:

1. 6 points for implementing detectCycleAndPeriod
2. 6 points for implementing removeCycle
3. 12 points for implementing removeSharedLinkedListNodes

Code documentation 2 points.

6 Grading policy

Students could be randomly selected to demonstrate their programs and to show that they understand their codes.

The skeleton code provided demonstrates how the code will be tested, however the skeleton code does not test every aspect of the functionality, and additional tests will be part of the grading process. Work should be done individually. Any source of help MUST be indicated and cited

Grading will be done as follows:

1. You will receive less than 20% of the total grade if your program compiles, but it doesn't work at all.

2. You will receive 0 points if your program will not compile.

[3]Code efficiency will be considered in grading each of the implementation questions.

3. For identical code, in part or full, the UofC rules and regulations for plagiarism will apply.

4. After the due time, a %20 deduction will apply on every late day or part of the day.

It's better to have something working, even with little functionality, than a big program that crashes (or doesn't compile). We expect to see your own program, otherwise the above will apply.

7 To Submit

Submit all java files required to compile your project, including all the provided files, even if you have not modified them, and your written component files as a .zip archive. Also be sure to include any additional files you may have written (if any). Name your compressed file **Assign2-ID.zip**, where ID is your UCID.

For the written component please submit a legible document with your two proofs. You may typeset your document using word processing software such as Microsoft Word, LibreOffice Writer, or Latex, or you can hand-write your proofs. If you are writing them by hand submit a clear scan or photograph of your proofs. Please ensure that scans/photos of handwritten proofs are clearly readable. The written document should be named **Assign2Proof-ID**.

You are URGED to check your submission after loading it on D2L, unzipping it and ensuring the contents are as expected. Errors in submission after the deadline will be subject to the indicated penalties above.

8 Deadline

Sunday June 5, 2022 before 11:55 P.M. For each late day (in full or in part), a deduction of %20 will be applied.

9 Reference

`https://docs.oracle.com/javase/tutorial/java/generics/index.html`