# ECE 264: Advanced C Programming

Programming Assignment 7 - Linked List, Due 7/22/19

## 1 Learning Goals

You will learn:

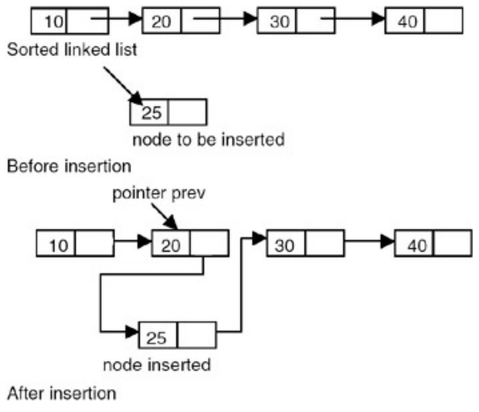
1. Using a linked list data structure
2. Creating and removing nodes
3. Manipulating pointer addresses of nodes
4. Implement a loop detection algorithm for a linked list

## 2 Background

### 2.1 Linked Lists

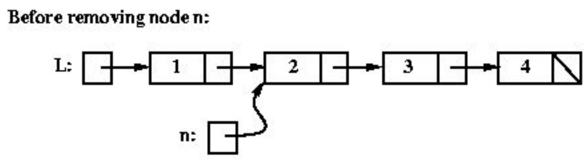
A linked list is a collection of data elements called nodes which are used to store data. Each node is linked to the next node via a pointer. You may wonder why would I use this if I already know how to use arrays? There are several advantages as well as disadvantage for linked lists.

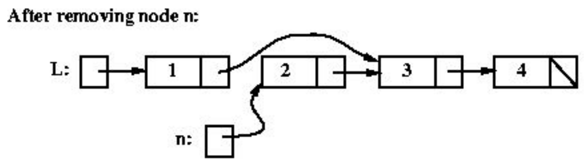
First, linked lists allow for very quick insertions and deletions. For example, lets say there is a list of numbers that we wanted to keep in order at all times. When we want to add a new number to the collection, we have to see where to add it and then perform the insertion operation. With a linked list, this is very quick, because once we know where to add the new number, we can just link our new node to the rest of the list by splicing the new node in. See the following diagram:



Likewise, removing a node is quick as well. Once we have a pointer to the node we want to remove, we can simply take the previous node, and link it to the next node of the node we are removing. See the following diagram:

You might wonder why that matters, because with an array we can also perform insertions and deletions. However, arrays require all the elements to be adjacent to one another. If there are 1000 integers in an array, and we wanted to add an integer at index 500, we would have to shift half of the array in order to make room for the new element! Similarly, when removing an element we would have to shift all the numbers back. With a linked list we don't have to perform these expensive operations in order to add and delete elements.

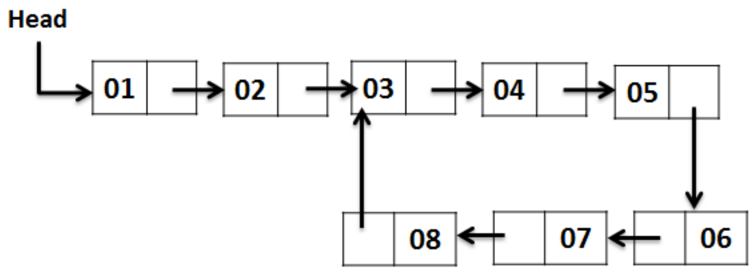




However, there are some drawbacks to linked lists. Linked lists are slow comparatively to arrays to access a *particular* element. For example, with an array to access index 500, we can calculate where that would be in memory from the start of the array and immediately get the data that is there. With a linked list we would have to start from the head of the list, and traverse it one node at a time, until we reach the $500_{th}$ node. This can take some time.

## 2.2 Circular loops in linked lists

Notice that each node in the linked list is connected to the next node. For a linked list to be valid we require that eventually, the last node in the list is connected to NULL. This means that there is no more next node in the linked list, and for all purposes we can treat that last node as the end of the linked list. Notice if this is **not** the case, that is, if this never happens because there is a loop in the linked list where one node is connected to a previous node in the list (see the following diagram):



then we would not be able to ever traverse this list to completion. There would always be a further next node to go to and we would never see the end.

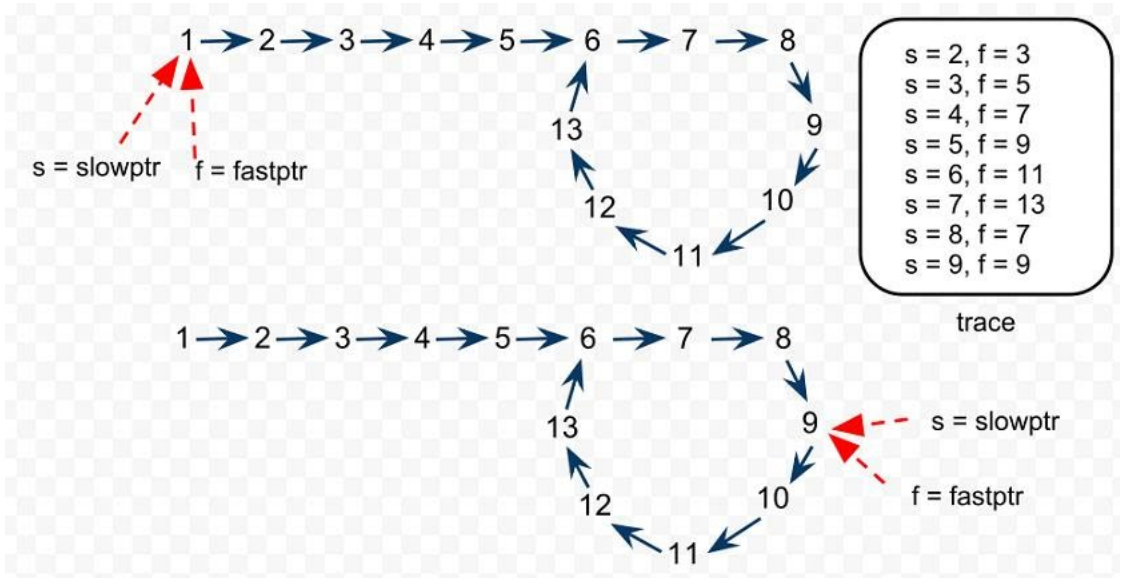
Your job for this assignment is to write a function that determines whether or not a linked list is malformed in this manner. That is, if the linked list is has a circular loop your method should return true, and if it does not have a circular loop your method should return false.

## 2.3 Algorithms

There are several algorithms that you may come up with to perform this task. One idea, is to save each address you visit in an array. As you traverse the linked list you can search your array for that address, and if you spot that you've already visited that node's address before you can stop and return `true`. However, searching an array take time, especially if the linked list is large. There are other data structures that allow you to instantly lookup if an item is in a list or not (look up Hashsets on Google for more background information), but the problem with those is that it would still require you to save all the addresses.

There is a very clever solution that works no matter how large the linked list is, requires no saving of previous addresses, and only needs 2 pointers to work.

Consider two pointers, a slow pointer, and a fast pointer. Initially both will point to the beginning of the linked list. The fast pointer is incremented twice, while the slow pointer is only incremented once. If the fast pointer reaches the end of the linked list then we know there isn't a loop because the linked list has been found to terminate. If the fast pointer ever loops back around and catches up to the slow pointer, then we know that there must be a loop. See the following diagram:



# 3 What you need to do

Modify `answer07.c` to detect if there is a circular loop in a given linked list using the two pointer algorithm mentioned above. Your method should return `true` if and only if there is a circular loop and `false` otherwise.

We have provided a rudimentary implementation of a linked list that allows insertions and deletions at a specified index with the node holding an integer for data. **You need to understand how this data structure operates before you start writing code**. The `insert` method will add a node at an index and return a pointer to the node it just added. A `NULL` pointer means it was not able to add a node. The `rem` method removes the node at a specified index from the linked list, and returns `true` or `false` on success and failure.

## 3.1 Testing your code

You will need to develop your own test cases in pa07.c to verify the correct functionality of your isCircular method inside of answer07.c. The following code snippets show how you can use the provided linked list data structure to add and remove nodes.

```
node_t * head = NULL; //initialize head of the linked list to be NULL
for(int i = 0; i < 10; i++)
{
  insert(&head, i, 50 + i); //a reference to head is passed so it can initialize it
      to a correct value
}
printlist(head);

//head: 0x19d9e80
//0x19d9e80: data = 50, next = 0x19d9ea0
//0x19d9ea0: data = 51, next = 0x19d9ec0
//0x19d9ec0: data = 52, next = 0x19d9ee0
//0x19d9ee0: data = 53, next = 0x19d9f00
//0x19d9f00: data = 54, next = 0x19d9f20
//0x19d9f20: data = 55, next = 0x19d9f40
//0x19d9f40: data = 56, next = 0x19d9f60
//0x19d9f60: data = 57, next = 0x19d9f80
//0x19d9f80: data = 58, next = 0x19d9fa0
//0x19d9fa0: data = 59, next = (nil)
```

We can make the above code faster by keeping track of where the end of the linked list is so that the insert method does not have to iterate through the linked list to find the $i_{th}$ node. Remember iteration through a large linked list can take time.

```
node_t * head = NULL; //initialize head of linked list to be NULL
node_t * tail = NULL; //same thing for the tail

tail = insert(&head, 0, 50); //set tail to be wherever this first node was added.
    head also gets initialized inside of insert
for (int i = 1; i < 10; i++)
{
  tail = insert(&tail, 1, 50 + i); //insert at "index 1" with respect to the tail
      the number 50 + i and update tail
}

printlist(head);

//0x1457ec0: data = 50, next = 0x1457ee0
//0x1457ee0: data = 51, next = 0x1457f00
//0x1457f00: data = 52, next = 0x1457f20
//0x1457f20: data = 53, next = 0x1457f40
//0x1457f40: data = 54, next = 0x1457f60
//0x1457f60: data = 55, next = 0x1457f80
//0x1457f80: data = 56, next = 0x1457fa0
//0x1457fa0: data = 57, next = 0x1457fc0
//0x1457fc0: data = 58, next = 0x1457fe0
//0x1457fe0: data = 59, next = (nil)
```

Prepending to the beginning of the linked list

```
node_t * head = NULL;
for (int i = 0; i < 10; i++)
{
  insert(&head, 0, 50 + i); //insert at "index 0" with respect to the head the
      number 50 + i
}
printlist(head);

//head: 0x798fa0
//0x798fa0: data = 59, next = 0x798f80
//0x798f80: data = 58, next = 0x798f60
//0x798f60: data = 57, next = 0x798f40
//0x798f40: data = 56, next = 0x798f20
//0x798f20: data = 55, next = 0x798f00
//0x798f00: data = 54, next = 0x798ee0
//0x798ee0: data = 53, next = 0x798ec0
//0x798ec0: data = 52, next = 0x798ea0
//0x798ea0: data = 51, next = 0x798e80
//0x798e80: data = 50, next = (nil)
```

Remove a node of a linked list

```
node_t * head; // assume initialized to some value
rem(&head, 3); // remove node at index 3. use 0 to remove the first node and update
    the head
```

**Before starting to work on the isCircular method you are encouraged to play around with the linked list data structure to understand how insertion and deletion work.**

**Note:** You do not need to use Valgrind to test this assignment's code (and we will not use it while grading)

## 4 What you need to submit

Submit your answer07.c file. You do not need to submit any test code you wrote.

*We will deduct 20 points for all Git submission related errors.*