<http://ostermiller.org/find_loop_singly_linked_list.html>

**Finding a Loop in a Singly Linked List**

**Motivation**

A singly linked list is a common data structure familiar to all computer scientists. A singly linked list is made of nodes where each node has a pointer to the next node (or null to end the list). A singly linked list is often used as a stack (or last in first out queue (LIFO)) because adding a new first element, removing the existing first element, and examining the first element are very fast O(1) operations.

When working with singly linked list, you are typically given a link to the first node. Common operations on a singly linked list are iterating through all the nodes, adding to the list, or deleting from the list. Algorithms for these operations generally require a well formed linked list. That is a linked list without loops or cycles in it.

If a linked list has a cycle:

* The malformed linked list has no end (no node ever has a null next\_node pointer)
* The malformed linked list contains two links to some node
* Iterating through the malformed linked list will yield all nodes in the loop multiple times

A malformed linked list with a loop causes iteration over the list to fail because the iteration will never reach the end of the list. Therefore, it is desirable to be able to detect that a linked list is malformed before trying an iteration. This article is a discussion of various algorithms to detect a loop in a singly linked list.

**Incorrect "Solutions"**

**Traverse the List Until the End**

Just look at the entire list to see if it has and end. When it ends, return.

// Incorrect 'solution'

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

while (currentNode = currentNode.next());

return false;

}

The problem with this solution is that if the linked list does have a loop, the program will never terminate. There is no way for this algorithm to return true when the linked list does have a loop.

**Mark Each Node**

Traverse the list and mark each node as having been seen. If you come to a node that has already been marked, then you know that the list has a loop.

// Incorrect 'solution'

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

do {

if (currentNode.seen) return true;

currentNode.seen = true;

} while (currentNode = currentNode.next());

return false;

}

The problem with this solution is ensuring that "seen" is marked as false for all the nodes before you start. If the linked list has a loop, it isn't possible to iterate over each node to set "seen" to "false" as an initial value for each node. It might be possible to overcome some of this by using a big integer rather than a boolean and using a random integer as your marker. In that case there is is a good chance that no node will have your inital value, but a small chance that one would and your algorithm would fail.

Even if you are able to solve the initial value problem, each node in a linked list may not have a field to use for this purpose. Requiring such a field in each node would mean that this is not a generic solution. As we will see later, this field is not needed for a perfectly correct and efficient solution anyway.

**Detect Only Full Loops**

When asked to come up with a solution, a common pitfall is not detecting all loops, but just a loop where the last node links to the first. A loop could still occur (and not be detected) if the last element linked to (for example) the second element.

// Incorrect 'solution'

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

while (currentNode = currentNode.next()){

if (currentNode == startNode) return true;

}

return false;

}

**Inefficient Solutions**

**Keep a hash set of all nodes seen so far**

O(n) time complexity, O(n) space complexity

Keeping a set of all the nodes have seen so far and testing to see if the next node is in that set would be a perfectly correct solution. It would run fast as well. However it would use enough extra space to make a copy of the linked list. Allocating that much memory is prohibitively expensive for large lists.

// Inefficient solution

function boolean hasLoop(Node startNode){

**HashSet nodesSeen = new HashSet();**

Node currentNode = startNode;

do {

if (nodesSeen.contains(currentNode)) return true;

nodesSeen.add(currentNode);

} while (currentNode = currentNode.next());

return false;

}

**Use a doubly linked list**

O(n) time complexity

Doubly linked lists make it easy to tell if there is a loop. If you encounter any node that doesn't link to the last node you visited, you know that there are two nodes linking to that node. Because the back links could be initially messed up in some other way, this algorithm is only correct if you can trust the back links. Otherwise it is just a malformed doubly linked list finder. The singly linked list can even be converted into a doubly linked list with little additional work. Again this will require that we change the structure of the Node to accomodate a second link. Something that may not be possible in all cases. Usually a singly linked list is used because the amount of space to allocate for each node is at a premium.

// Inefficient solution

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

Node previousNode = null;

do {

if (previousNode && currentNode.prev() && previousNode != currentNode.prev()) return true;

if (!currentNode.prev()) currentNode.setPrev(previousNode);

previousNode = currentNode;

} while (currentNode = currentNode.next());

return false;

}

**Check the Entire List So Far**

O(n^2) time complexity

For each node, assume that the portion of the list examined so for has no loops and check to see if the next node creates a loop by iterating again over the entire list up to that point.

// Inefficient solution

function boolean hasLoop(Node startNode){

Node currentNode = startNode.next();

int i=0;

do {

Node checkNode = startNode;

int j=0;

do {

if (checkNode == currentNode) return true;

j++;

} while (j<i && checkNode = checkNode.next());

i++;

} while (currentNode = currentNode.next());

return false;

}

**Reverse the list**

O(n) time complexity

If you reverse the list, and remember the inital node, you will know that there is a cycle if you get back to the first node. While efficient, this solution changes the list. Reversing the list twice would put the list back in its initial state, however this solution is not appropriate for multi-threaded applications. In some cases there may not be a way to modify nodes. Since changing the nodes is not needed to get the answer, this solution is not recommended.

// Solution modifies the list

function boolean hasLoop(Node startNode){

Node previousNode = null;

Node currentNode = startNode;

Node nextNode;

if (!currentNode.next()) return false;

while(currentNode){

nextNode = currentNode.next();

currentNode.setNext(previousNode);

previousNode = currentNode;

currentNode = nextNode;

}

return (previousNode == startNode);

}

Credit for this solution goes to Piyush Srivastava.

**Use Memory Allocation Information**

O(n) time complexity in the amount of memory on the computer

Some programming languages allow you to see meta information about each node -- the memory address at which it is allocated. Because each node has a unique numeric address, it is possible to use this information to detect cycles. For this algorithm, keep track of the minimum memory address seen, the maximum memory address seen, and the number of nodes seen. If more nodes have been seen than can fit in the address space then some node must have been seen twice and there is a cycle.

// Depends on size of available computer memory rather than size of list

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

int minAddress, int maxAddress = ¤tNode;

int nodesSeen = 0;

while(currentNode = currentNode.next()){

nodesSeen++;

if (¤tNode < minAddress) minAddress = ¤tNode;

if (¤tNode > maxAddress) maxAddress = ¤tNode;

if (maxAddress - minAddress < nodesSeen) return true;

}

return false;

}

This algorithm relies on being able to see memomory address information. This is not possible to implement in some programming languages such as Java that do not make this information available. It is likely that the entire list will be allocated close together in memory. In such a case the implementation will run close to the running time of the length of the list. However, if the nodes in the list are allocated over a large memory space, the runtime of this algorithm could be much greater than some of the best solutions.

**Best Solutions**

**Catch Larger and Larger Loops**

O(n) time complexity

Always store some node to check. Occasionally reset this node to avoid the "Detect Only Full Loops" problem. When resetting it, double the amount of time before resetting it again.

// Good solution

function boolean hasLoop(Node startNode){

Node currentNode = startNode;

Node checkNode = null;

int since = 0;

int sinceScale = 2;

do {

if (checkNode == currentNode) return true;

if (since >= sinceScale){

checkNode = currentNode;

since = 0;

sinceScale = 2\*sinceScale;

}

since++;

} while (currentNode = currentNode.next());

return false;

}

This solution is O(n) because sinceScale grows linearly with the number of calls to next(). Once sinceScale is greater than the size of the loop, another n calls to next() may be required to detect the loop. This solution requires up to 3 traversals of the list.

This solution was devised by Stephen Ostermiller and proven O(n) by Daniel Martin.

**Catch Loops in Two Passes**

O(n) time complexity

Simultaneously go through the list by ones (slow iterator) and by twos (fast iterator). If there is a loop the fast iterator will go around that loop twice as fast as the slow iterator. The fast iterator will lap the slow iterator within a single pass through the cycle. Detecting a loop is then just detecting that the slow iterator has been lapped by the fast iterator.

// Best solution

function boolean hasLoop(Node startNode){

Node slowNode = Node fastNode1 = Node fastNode2 = startNode;

while (slowNode && fastNode1 = fastNode2.next() && fastNode2 = fastNode1.next()){

if (slowNode == fastNode1 || slowNode == fastNode2) return true;

slowNode = slowNode.next();

}

return false;

}

This solution is "Floyd's Cycle-Finding Algorithm" as published in "Non-deterministic Algorithms" by Robert W. Floyd in 1967. It is also called "The Tortoise and the Hare Algorithm".