Linked Lists

Along with arrays, linked lists form the basis for pretty much every other data stucture out there. This makes learning and understand linked lists very important. They are also usually the next step in memory management that people encounter after learning pointers.

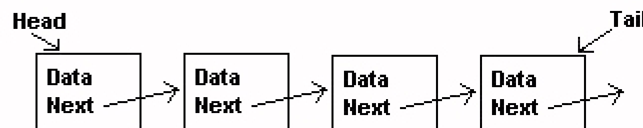
I've written a linked list structure specifically for this tutorial which can be found in the downloadable source code. I commented it thoroughly so it should be able to stand alone as its own tutorial. To make things even easier on you, I will go through the list structure function by function and discuss what the function does as well as show pictures that demonstrate what's going on.

I recommend that you open the downloadable source code now and follow along. Reading both at the same time will allow you to see the theory as well as the implementation.

If you plan on getting more advanced at game programming, spend some time learning linked lists! If you need a refresher on pointers, see my Pointers Tutorial. And if you don't know what a template is, see my Template Tutorial.

What is a Linked List?

A linked list is a node-based data structure. What does that mean? It means that a linked list is a structure composed of smaller structures called nodes, which store the actual data in the list. We link these data storing nodes together into a list to create a linked list. Here is a picture of a linked list:



The squares in this picture represent the nodes of the list. Notice that each square (node) contains a data member and a pointer called Next which points to the next node in the list. The last node in a linked list always points to NULL. Here is the code for a node structure:

```
template <class DataType>
class Node
{
    DataType Data;
    Node };
```

So we know that a linked list contains a group of structures called nodes which are linked together. Each node stores some data and a pointer to the next node in the list. The last node in the list points to NULL. This node is called the *tail*. The front node is called the *head*.

Now that we know what a linked list is, let's discuss why we would use one and then go on to discussing the implementation of a linked list.

Why Use a Linked Lists?

When we want to store data, we usually first think of an array. Arrays are definately the most popular way to store data. After we decide on the size of our array, we can add data to it simply by specifying where we want it to be within our array; we access data the same way.

This means that arrays gives us instant access to our data. Even better, aside from pointers that we might have pointing to our arrays, we never really need to worry about memory management. So why use a data structure that is constructed entirely from pointers?

The reason is that linked lists allow us to store varying amounts of data. With arrays, we have to decide on the amount of data to store from the beginning, and we can never easily change this size. With linked lists however, all we need to do is create a new node and tag it on the end of the list.

The choice between using an array or using a linked list is a matter of memory vs. speed. If you're not worried about wasting memory and you want a lot of speed, just use a really big array. Some memory will be wasted if you don't use up the entire array, but you get a lot of speed because you can access the data in an array instantly by specifying the index you want to access.

Linked lists on the other hand are good when you need the memory. You never waste space with a linked list because you never need to have more nodes than you have data. Unfortunately, you don't get instant access time with linked lists. You can't just say you want node 3 of a list and get it instantly. The reason is that you only have pointers to the front and back of a linked list.

Take a look at that picture I just showed you. Head points to the front of the list and Tail points to the back. If you want to get node 3, you have to start at Head and work your up through the list to get to node 3.

For this reason, you probably won't find yourself using linked lists as much as you use arrays. However, linked lists form the basis for more advanced structures like Stacks,

Queues, Trees, and Graphs. So pay attention!

The Structure of a Linked List

I'm now going to go through the creation of a linked list. I will only be discussing the concepts here. See the downloadable source code for a heavily commented implementation of a linked list. I strongly suggest that you read this along with the code. Make sure you understand how everything works because once you do, you'll have a much better understanding of programming in general.

Once we have a node structure defined, we create a class to handle the linking of these nodes. This class will be our main linked list structure. The only data members of the class will be a pointer to the first node (the head), a pointer to the last node (the tail), and a variable that keeps track of the number of nodes in our structure.

When we initialize a linked list, we set its head and tail pointers to NULL so that they're not pointing to anything weird. This is what our list looks like after intialization:

Adding Nodes to a List

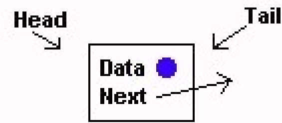
So we have both our head and tail pointers pointing to NULL. Now let's try adding some nodes. There are three ways to add a node to a linked list. We can add it to the front of the list, the back, or we can add it somewhere in the middle.

Adding a node at the head of a list. When adding a node at the front of the list, we set the new node's Next pointer to point to the head and the Head pointer to point to the new node. There are two scenarios we have to deal with when doing this.

The first scenario occurs when the list is empty. When this happens, we assign the new node's Next pointer to the head, which results in it pointing to NULL because the head was NULL. This way we get the Head pointer pointing to the new node, and the new node's Next pointer pointing to NULL.

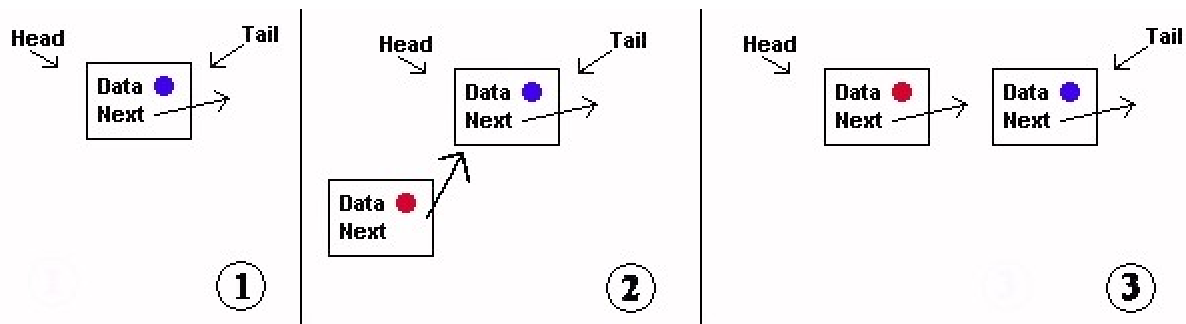
But what about the tail? If the tail points to the last node in the list, and there's only one node in the list, then it should point to the new node too. In the end, we have a new node which points to NULL, and our Head and Tail pointers point to the new node.

Here's what our linked list looks now.



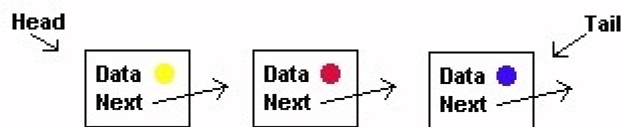
The second scenario occurs when we already have nodes in the list. Because we are adding a new node at the head of the list, the node that is currently the head of the list becomes the second node in the list, and the new node takes its place at the front. So, to add a node to the front of the list we first assign the new node's Next pointer to the current head, and then we set the Head pointer to the new node.

Here's a picture to illustrate the process:



In **(1)**, we just have our original list with one element in it. In **(2)**, we have created a new node and set its Next pointer to the current head of the list. In **(3)**, we have set our new node as the head. Notice how the node that was the head (the blue node) is still the tail.

Here's a picture of our list after another node is inserted:



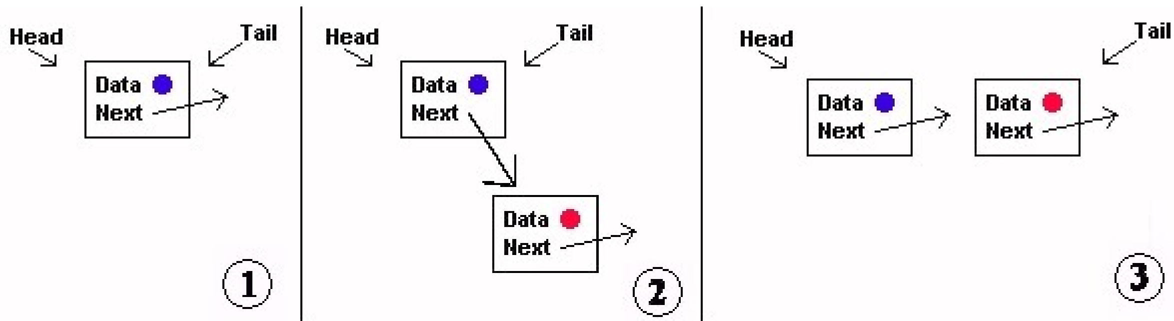
So the new node **(yellow)** is set to point to the old head **(red)**. Then the new node **(yellow)** is made the head of the list. After this process, we can see that the new node **(yellow)** is the head, the old head **(red)** is the second node in the list, and the old tail **(blue)** is still the tail.

Adding a node at the back of a list. When we add a node at the end of our list, we set its Next pointer to NULL because the last node in a linked list has nothing to point to.

The same two senarios occur when adding to the back of a list as when adding to the front. There will either be something in the list already, or there won't. The process of adding a node at the end of an empty list is the same as adding a node at the front of an empty list. In both cases we set the new node's Next pointer to NULL, and then set the link list's Head and Tail pointers to the new node.

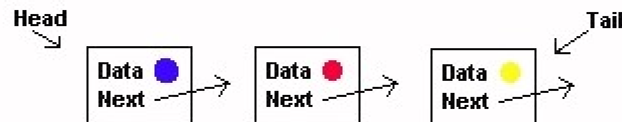
The process of adding the node when there is already something in the list, however, is slightly different. To add a node at the end of a list, we first have to set the current tail node's Next pointer to the new node. We then set the linked list's Tail pointer to the new node.

Here's a picture illustrating the process:



In **(1)**, we just have our original list with one element in it. In **(2)**, we have created a new node and set the old tail node's Next pointer to the new node. In **(3)**, we have set the linked list's Tail pointer to our new tail node.

Here's a picture of our list after another node is inserted:



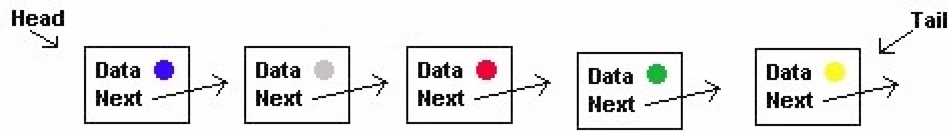
The new node **(yellow)** becomes the tail node and has it's Next pointer set the NULL. The old tail **(red)** becomes the second last node in the list.

Adding a node in the middle of a list. The process of adding a node somewhere in the middle of a linked list is slightly more complicated than adding a node at the front or the back.

The first problem that can occur is when we try to add the node to a location in the list that doesn't exist. For example, say we have a list of 5 nodes and we try to add a node between the 10th and 11th node. Wouldn't work now would it? There are two things we can do in this case. We can do nothing, or we can just add the node to the end of our list. I like to add the node at the end of the list so that's what we'll do. To accomplish this, we just call our AddTail function.

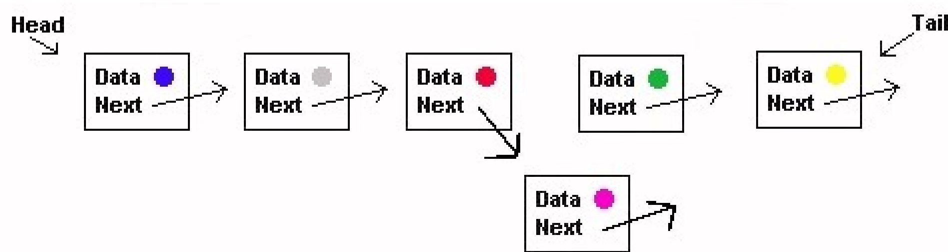
Another problem is trying to add a node in the middle of an empty list. We've already got this problem covered though! If there are no nodes in our list, then any location we try to add a node at won't exist, so our AddTail function will be called. AddTail already handles the case of an empty list!

If the above problems don't occur, we'll have a valid location at which we need to add our new node. Let's first take a look at a list with five nodes.

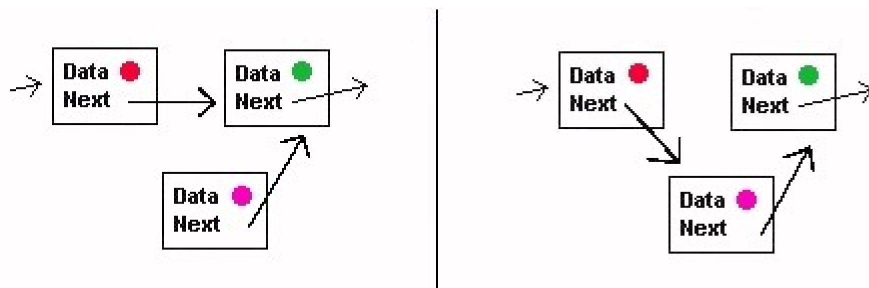


Now let's assume we want to add a node between node 2 (red, counting starts at 0!!!) and node 3 (green). We've got a few problems here. First, we don't have a pointer to either of these nodes. We only have pointers to the start and end of the list. To get around this, we'll have to start at the front of the list and travel from node to node until we get to the red node.

Another problem is how to insert the new node once we get there. If we immediately set the red node's pointer to point to our new node, we'll lose the rest of the list. Here's a picture:



See, if we just set the red node's Next pointer to the new node, we no longer have anything pointing to the green node. The green node will become a memory leak and our list will be ruined. What we need to do is first set our new node's Next pointer to the green node, and then we can set the red node's Next pointer to the new node.

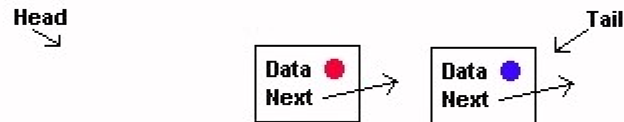


At the end of the process, our linked list looks like this:

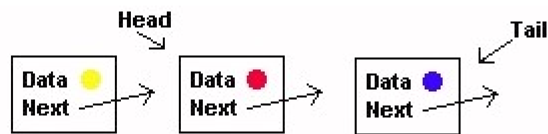
Removing Nodes From a List

Removing nodes from a list is a fairly easy process. We just have to make sure to avoid losing references to nodes (and thus causing memory leaks). Note that in each case we first check to make sure the list is not empty. If it is empty, we don't have to do anything.

Removing a node from the front of a list. We have a dilemna when removing a node from the front of a list. If we start by deleting the node at the front of the list, we get the following:



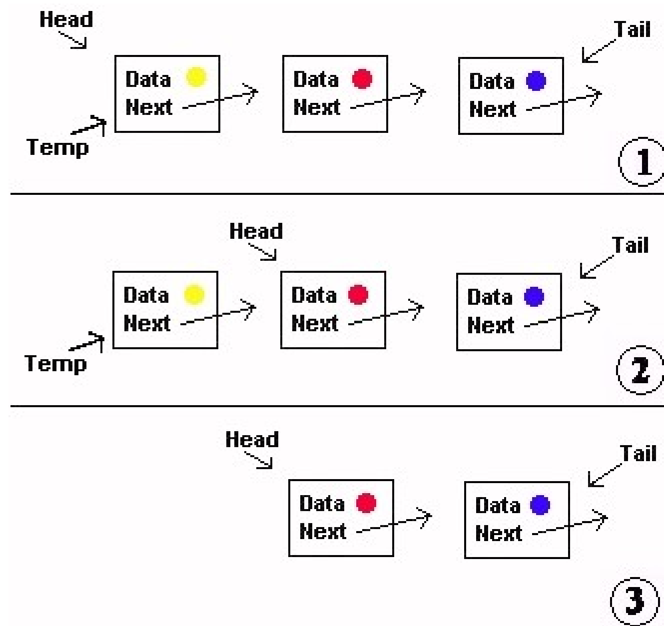
We have indeed gotten rid of the node at the front of the list, but we've also lost all references to the red node! So let's try first moving the linked list's Head pointer to the next node in the list before we delete anything.



Now the problem is how to actually delete the node at the front of the list. Since we've moved our Head pointer forward, we no longer have any reference to the front of the list.

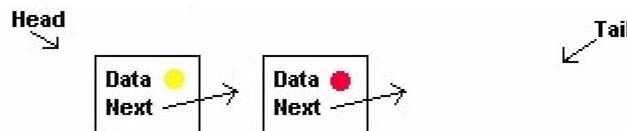
The solution to our problem is to first create a temporary node pointer that points to the head of the list. We then change the linked list's Head pointer. Finally, we delete the front of the list by deleting the node that the temporary pointer points to (it points to the front node).

Here's the process in photo-realistic detail:



So in **(1)**, we create our temporary pointer. In **(2)**, we move our Head pointer up one node. In **(3)**, we delete the node that the temporary pointer points to.

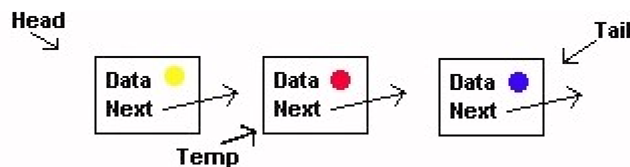
Removing a node from the end of a list. As with removing a node from the front of a list, we are faced with problems when we try to delete a node from the end of a list. Our first idea might be to just delete the node that our linked list's Tail pointer points to. Let's see what happens:



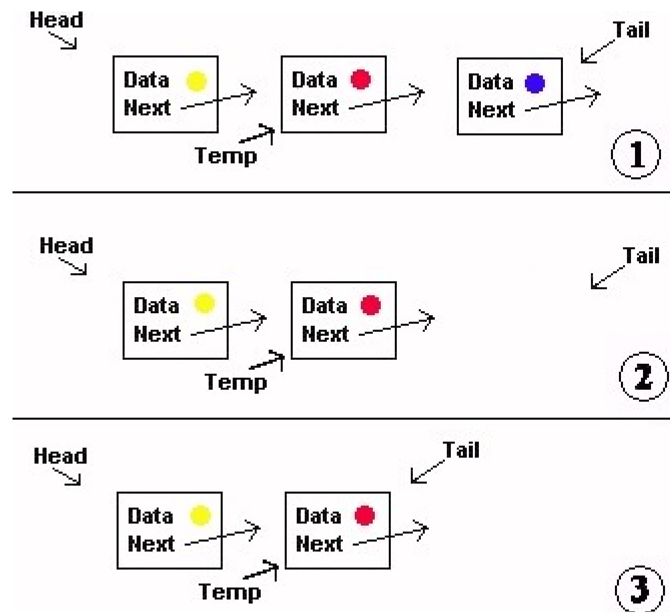
We no longer have our Tail pointer pointing to anything. This isn't as big of a problem as with removing the front node because we can still reach the last node in the list. We just start from Head and keep moving up until the node we're at points to NULL. Remember, the last node of a list always points to NULL. We can then set our Tail pointer to the end node.

Advanced readers might be bothered by what I just said. When we delete the object that a pointer points to, the pointer still points somewhere. We have to explicilty set our pointer to NULL before it actually points to NULL. We could very well crash our program by searching our linked list for a NULL reference that isn't there (because we just deleted the tail, we didn't set the new tail's pointer to NULL).

The solution is to first start at the front of the list and move to the node that is before the last node. Here's a picture:



Now our temporary node pointer points to the node that will become the tail of our list. This node's Next pointer also just so happens to point to the current tail. All we have to do now is delete the node that our current node's (the one that our temporary pointer points to) Next pointer points to (which is the current tail), set the Next pointer to NULL, and make our linked list's Tail pointer point to the node our temporary pointer points to. Here's the picture:

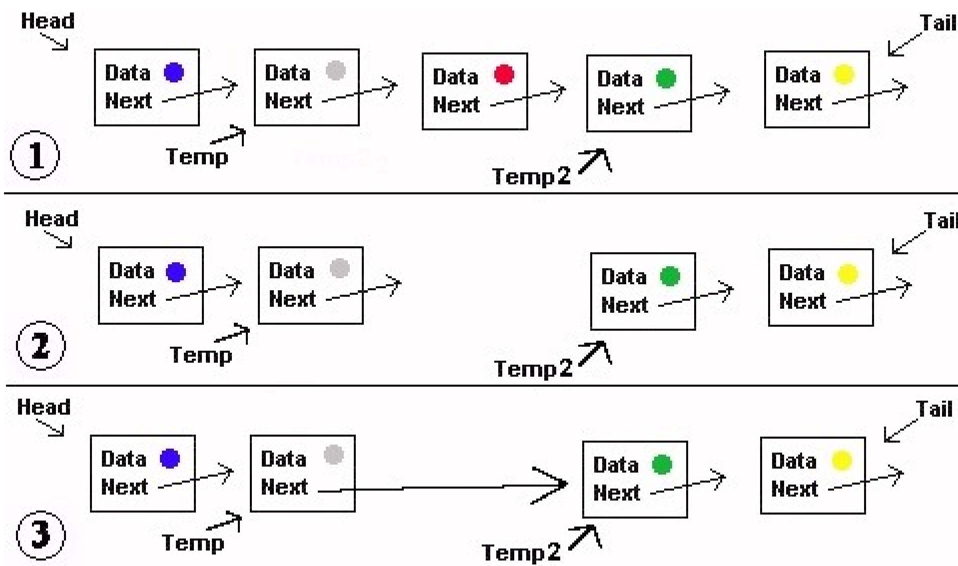


In **(1)**, we have a pointer to the node that will become the new tail. In **(2)**, we delete the current tail. In **(3)**, we change the linked list's Tail pointer to point to the new tail.

Removing a node from the middle of a list. We're almost done! Removing a node from the middle of a list is a lot like removing a node from the end. We create a temporary node pointer and move it to the node before the one we want to delete. Now we have to be careful though. Let's see what happens when we just delete the node like we did with the tail node.

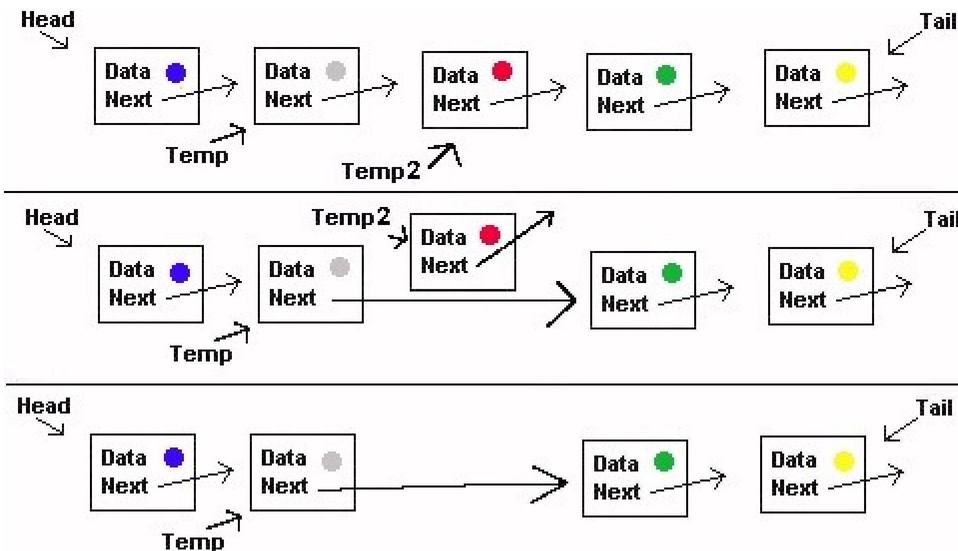
Here we've deleted the third node in our list. In doing so, we've lost all reference to the fourth node in the list. What we need is another temporary node pointer to point to the

fourth node while we delete the third node. That way, we can reconnect our list. Picture time!



In **(1)**, we have two temporary variables pointing to the nodes before and after the node we'll be deleting. In **(2)**, we delete the node. In **(3)**, we set the node before the deleted node to the node after the deleted node. After this process, we'll have a properly connected list, minus the deleted node.

Note that another way to do this would be to create a temporary pointer and set it to the node to be deleted. Then connect the nodes that come before and after the node to be deleted. Then, delete the node.



Retrieving data from a linked list. Getting data out of a linked list is really easy. Since each node contains data, all have to do is get to the node that has the data we want. For the cases where the data is in the first or last node, we just use our Head and Tail

pointers. If the data is somewhere in the middle of the list, we create a temporary node pointer and move it up the list until we get to the node we want. That's it!

By now I hope you fully understand how a linked list works. If you haven't been going through the downloadable source code already, I suggest you do it now. Being able to write your own linked list is the key to being able to write more advanced data structures like graphs and trees.

Click here to download the source code for this tutorial.

© Copyright Aaron Cox 2004-2005, All Rights Reserved.