Programming Notes 18: Basic Linked Lists

As usual, more examples of the notes are in the example program.

1. Linked Lists

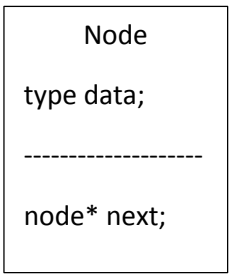
Linked lists are another fundamental concept in computer science, as they are used as the underlying implementation of many data structures, and can be used in a variety of situations. As of yet, we've only used arrays (dynamically allocated or otherwise) to store collections of data. A linked list is another way to store collections of data, but with a different set of advantages and disadvantages.

The main advantage of linked lists verses arrays is that they have no set limit on capacity (other than the maximum memory or your system), and the size can grow and shrink dynamically when data is added and removed. In an array, to change the size one must completely delete and reallocate the memory. Additionally, a linked list does not need to find a contiguous block of memory like an array does, making it faster and easier to allocate.

However, there is a significant disadvantage to linked lists: to access the data within them, you must traverse a number of spaces in memory to find the data you're looking for. This will make more sense in a minute. On the other hand, arrays, because they are a contiguous block of memory, can always retrieve any of their data in a constant, short amount of time.

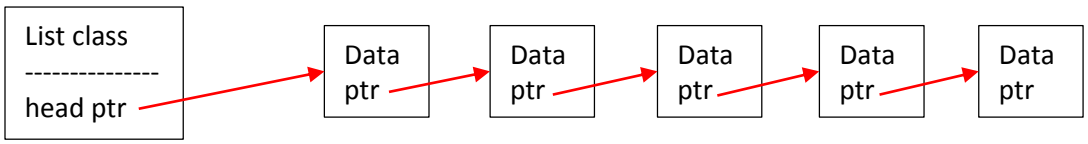
2. Nodes

The basic component of a linked list is what's called a "node." A node is what it sounds like: it represents one piece of the list and includes one piece of data. A node has two parts: data, to hold what is actually contained in the list, and one or more pointers to another node.



3. Links

The pointer portion of the node is what constitutes the "links" of the linked lists. You can probably guess where this is going: each node is "linked" to the next one in the list by storing its address in the node's pointer portion. With many nodes, this creates a chain. Of course, if you don't have a way to access the first node in the chain, you can't access any of them, so when you're using a linked list, you should always store a pointer to the "head," or first node. Note that here, if you ever lose your pointer to the first node, you have leaked memory, as you will never be able to get back to the first node.

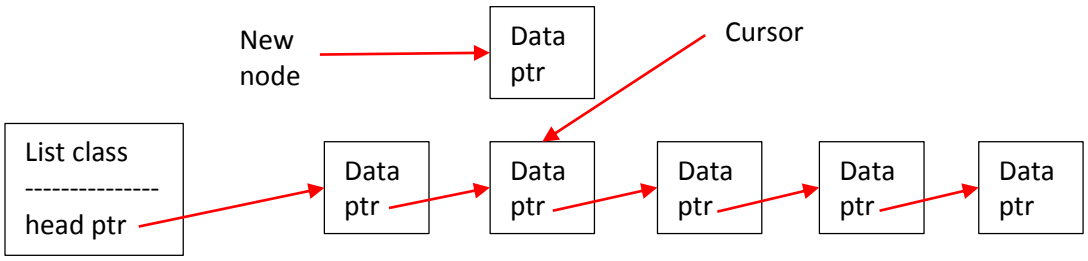


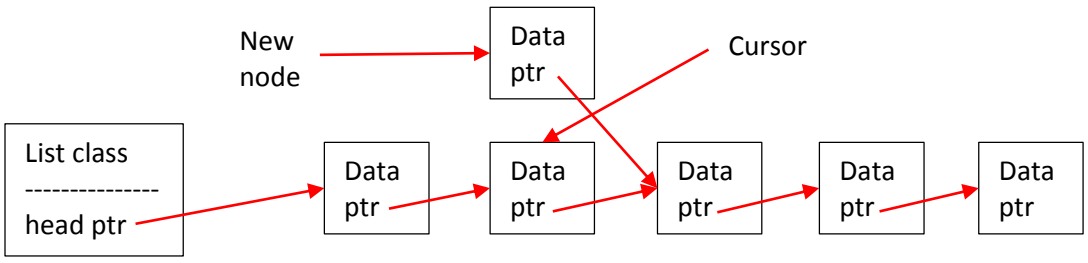
4. Adding and Removing Nodes

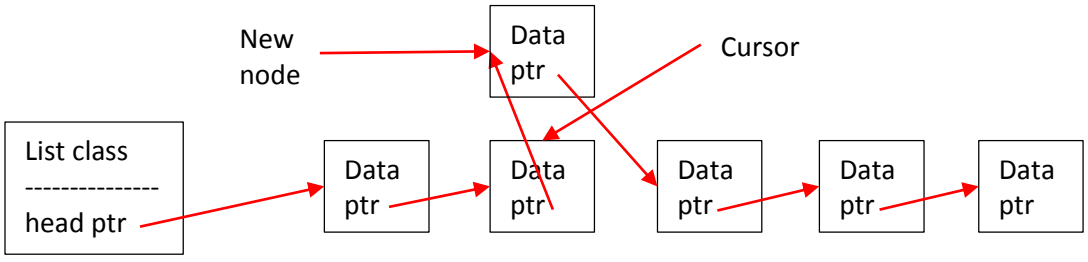
That's pretty much the entire basic concept of linked lists, although there are many other types of linked lists that add more complications. So, let's move on to working with linked lists. The basic operations are, of course, adding and removing nodes. Imagine we have a "cursor," which is a pointer to a node in the list. Adding a node after the cursor is extremely simple: you just have to have the new node point to the next node after where you're inserting it (new node->next = cursor->next), then have the node before it point to your new node (cursor->next = new node). Removing a node is also simple, although it is much easier if you are removing the node after your cursor.

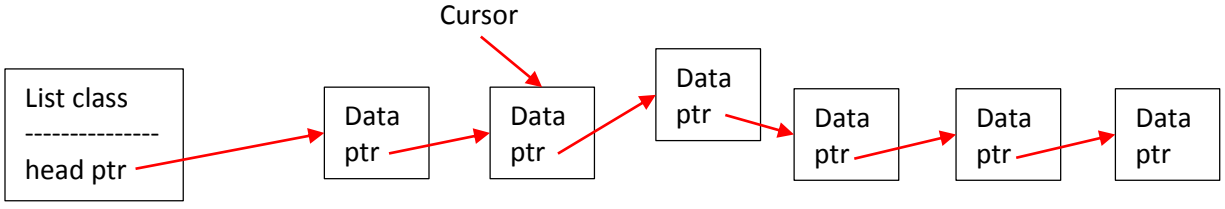
Remember that the examples here are only one case of what could happen. For example, image what you would need to do if you want to remove a node, but your cursor is pointing to the last node in your list. Further, these example processes are assuming you're using a singly linked list.

Inserting a node:

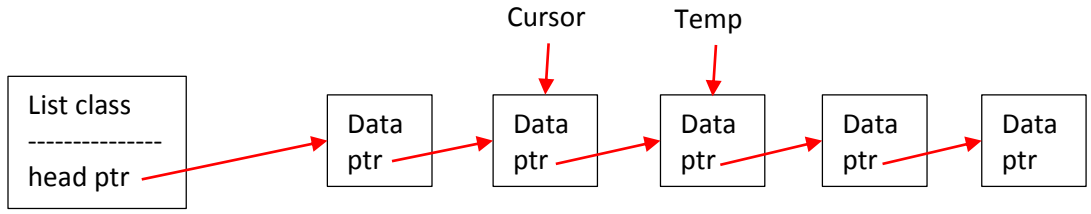


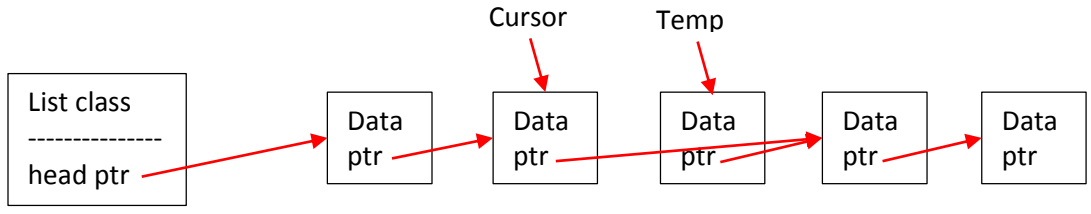


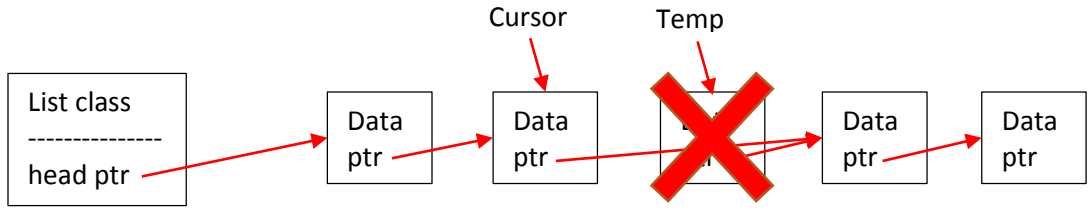


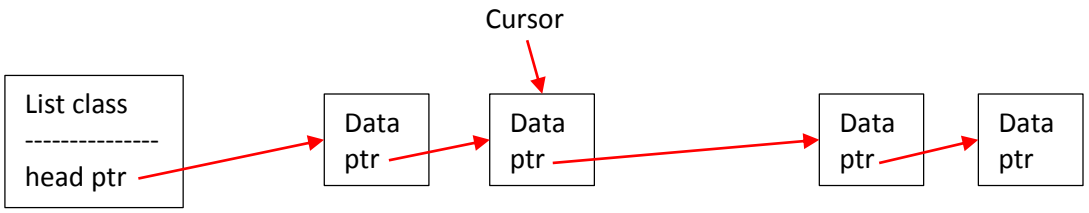


Removing a Node after the Cursor:







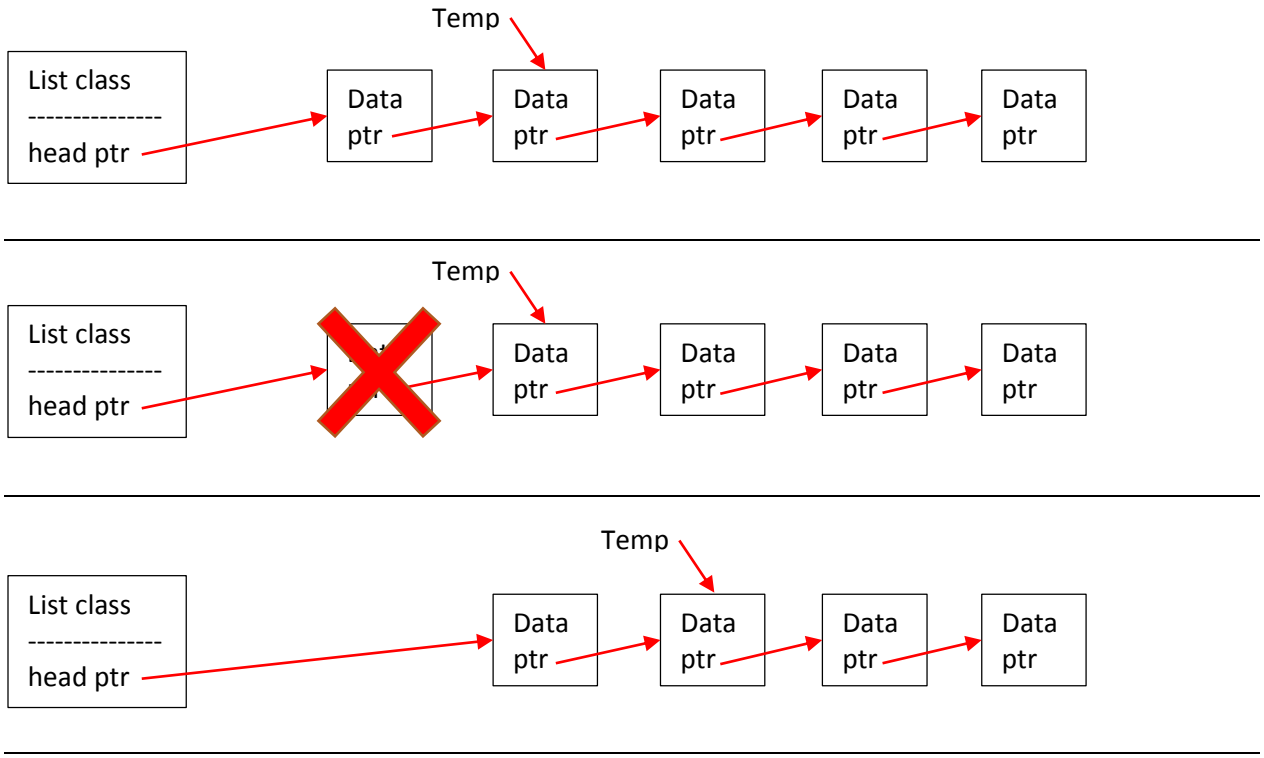


Again, there are many more situations than those shown here. For example, what if you wanted to remove the node that the cursor is pointing to? What if you want to insert a node before the cursor? There are too many possibilities to go into great detail here, but hopefully you can figure out what to do on your own.

Drawing pictures is extremely helpful! Diagramming simple processes like the ones above can make linked lists easy to deal with.

5. Clearing a Linked List

We'll go over one more process related to linked lists here: clearing them. At some point, you're always going to want to clear your linked lists, and you don't want to leak memory when doing so. As with the other processes, diagramming what needs to happen is a great idea.



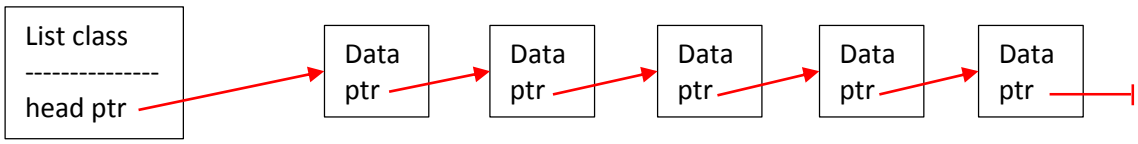
REPEAT

6. Some types of linked lists

So, these examples so far have been diagrammed assuming a singly linked list. However, there are several other types you should know of:

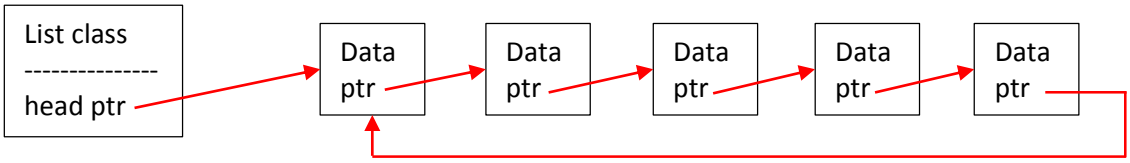
a. Singly Linked Lists

A list is singly linked if each node has one pointer to the next node in the chain. This is the simplest type of list. The end node should have "NULL" as the next node.



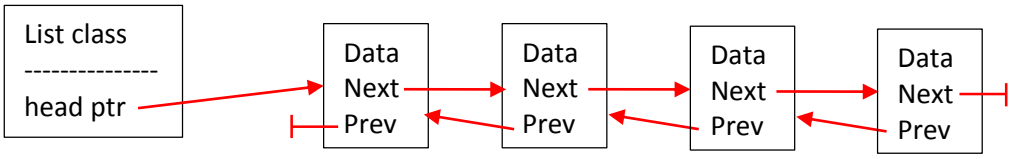
b. Circularly Linked Lists

A circularly linked list is exactly the same as a singly linked list, except that the final node's "next" node points back to the head of the list.



c. Doubly Linked Lists

A doubly linked list is what it sounds like: each node has two pointers, one two the next node, and one to the previous node. This makes traversing the list backwards way faster than a singly linked list (where you would have to traverse the list from the beginning until you found the node behind where you are), but comes at the cost of storing an extra pointer in each node.



Of course, you can also make a doubly linked list circular by having the first node point back to the last node and the last node point up to the first node.

d. Linked Trees

Finally, there are linked trees, which are a bit more complicated to deal with. They're also what they sound like: each node can have multiple "child" nodes, and point back towards their "parent" node.