Linked Lists – Part 1
Pages 236 – 244 in your textbook

With the message board assignment, and the word counting assignment, you're using an array and likely seeing the limitations of that data structure for what you're doing. For the message board, when you remove an item from the list then you have to perform lots of shifting operations to fill the gap. With the word counting assignment, you need to keep doubling the array size and copying over the existing elements in the array. Very costly.

With arrays, there is a contiguous block in memory that is allocated. Something like this:

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	A[13]

Another approach for storing a collection of data is to allocate memory for individual elements, and then create pointers that link them together.

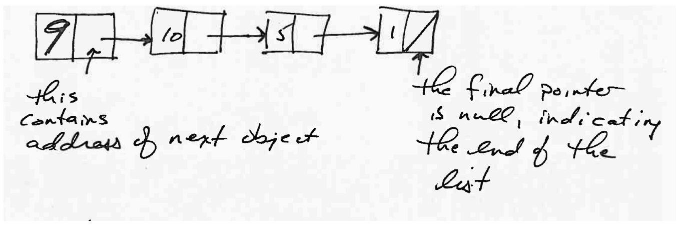
This data structure is called a linked list. The order of objects in the structure is determined by pointers between individual elements in the list, instead of by indices as we see with arrays. (Vectors probably use linked lists.)

There are two types of linked lists, singly linked and doubly linked lists.

Singly linked lists:
Objects point to the next object, but not to the previous object.

Doubly linked lists:
Objects point to the next and the previous object. More later.

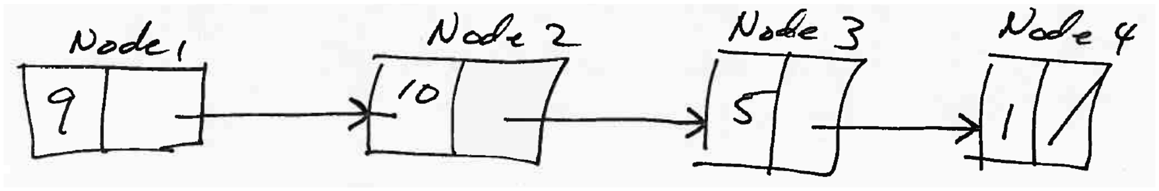
Example singly linked list:
We know that objects are stored in memory and all objects have a memory location. We can point to that memory location using a pointer variable. Create a simple linked list where each node has one property that is an integer key, and a pointer to the next node in the list. It looks like this:


9
10
5
1
this contains address of next object
the final pointer is null, indicating the end of the list

The number indicates the key for the node, and the empty box would hold a pointer to the address of the next node.

The final pointer on the last node in the list is NULL, indicating that we've reached the end of the list.
If the block diagram were implemented with a struct, it would look something like:

```
struct node{
  int x;
  node *next; //pointer to object of same type
}
```



Node 1 next is Node 2
Node 2 next is Node 3
Node 3 next is Node 4
Node 4 next is NULL

The object in this example is a struct with an integer and a pointer. But, the object could be anything. It could be a struct of your message board items. Or, it could be a class instance. You could create a linked list of Battleships, Cars, Elevators. The requirement here for it to be a linked list is that the order of the list is established with the next pointer, or in languages without pointers, some other representation of order independent of array index.

Things you might want to do to a linked list:
Insert
Delete
Search

The beginning of the list is the head of the list, and this is important because with a singly linked list, we can only go forward. We keep a separate pointer to store the head of the list.

Pseudo-code for Insert into singly linked list, at the head

```
List_insert(L, x)
  x.next = L.head
  L.head = x
```

Insert a node between two nodes

The order of operations really matters here. We can't lose the forward pointer, so we need to store that first before we do any updating of our list.

Complexity of insert into a list

Adding to the front of the list: O(1). This is a constant operation, independent of the size of the linked list. O(1) doesn't mean 1 operation, it just means a constant number of operations.

Adding to the back of the list is an O(n) operation, unless you store a pointer to the end of the list. Each time you want to add, you have to follow the pointers from the head of the list to the end, and then replace the last pointer to NULL with a pointer to the new node.

Searching a list for a given key value. In our example, we have one member that we could search for, which is called key. With a singly linked list, we start our search at the beginning of the list and follow the pointers until we find what we're looking for. We can return a pointer to the node.

//L is the linked list, k is the value to search for, in code we pass in a pointer to the head of the list. In pseudo-code, we pass in the list and get the head of the list as x. It's effectively the same thing if you consider that not all languages have pointers, or handle argument passing in the same way.

```
List_search(L, k)
  x = L.head
  while x.next != NIL and x.key != k
     x = x.next
  return x
```

The search function returns a pointer to the node where the key is found, or NIL if it isn't found.

Runtime is O(n) in worse case, occurs when you search the entire list. What is the best-case runtime and when do we see that case? It’s O(1) and occurs when the item we’re searching for is the first item in the list.