**Linked List**

When we create an array, the entire array is given some memory space. Thus, every member lies physically next to their neighbours in the memory. Say we have an array of 4 integers, which takes up 16 bytes of memory. The way the compiler detects an integer at a certain index is that it goes to the address of the first element, and then adds the address to find the address of the required element.

This is a poor way to handle memory however. If we do not know how many elements we will have, then we will need to allocated a large amount of space just to be safe. If we wanted to increase the size of the array, we would need to create an entirely new array. Both these actions waste a huge amount of space.

A much better method would be if we could dynamically allocate memory as we need it. This is essentially the principle behind a linked list. Elements are added in whatever position is available, then is linked to the rest of the list using pointers. If we remember the position of the first element, we can get to the second element from there since the first element has a pointer to the second element. The last element has a null pointer. The location of the element and the pointer together create a node.

typedef struct node Node;  
struct Node  
{  
 int value;  
 Node\* next;  
}  
Node\* createNode (int item, Node\* next)  
{  
 Node\* newNode = new Node();  
 if (newNode == **NULL**) *//memory full* else  
 {  
 newNode -> value = item;  
 newNode -> next = next;  
 return newNode;  
 }  
}

C++

This function simply creates a new node. It does not link it to the rest of the list. We will be looking at that later.

Note that the method used here to create a new node by allocating memory space to it is only usable in C++. In the C language we would need to use this:

((Node \*) malloc (sizeof (Node));

C

We can call the createNode method from the main function like this:

int main()  
{  
 Node \*n;  
 n = createNode (15, **NULL**);  
}

C++

There are three ways to add a node, at the beginning, at the end, and somewhere in the middle.

To add a node to the head, we can use the prepend function. Here, we create a node, point the next pointer of our newly created node to the head, and set the head to this node.

Node\* prepend (int item, Node\* head)  
{  
 Node\* newNode = createNode (item, head);  
 return newNode;  
}

C++

From the main function, we call it like this:

head = prepend (15, head);

C++

We are using the createNode function to create the new node, and we return the node to the main function, where it is being assigned to head. This function has a time complexity of O(1).

Note that when we first create the head in the main function, we must declare it **NULL** like this:

Node\* head = **NULL**;

C++

To add a node to the end of the list, we can use the append function. We create a new node and then we starting looking for the last node in the list. Since we only know what the head is, we must start there and keep going to the next nodes until we find one that points to **NULL**. This is the last node, so we make it point to the new node, which then points to **NULL**.

Node\* append (int item, Node\* head)  
{  
 Node\* newNode = createNode(item, **NULL**);  
 if (head == **NULL**) return newNode;  
 else  
 {  
 Node\* temp = new Node();  
 temp = head;  
 while (temp->next != **NULL**) temp = temp->next;  
 temp->next = newNode;  
 return head;  
 }  
}

C++

From the main function, we call it like this:

head = append (9, head);

C++

We had to check whether we had a head at all towards the start of the algorithm. If we did not, the node we just created would be set to the head (look at the call from the main function). If we did, then we make our changes and then still return the same head. There are two reasons why we return the head. Firstly, it is to accommodate the case where we did not have a head. The second reason is to maintain similarity. All the functions we will use to insert or remove nodes will be called in a very similar manner.

The append function has a time complexity of O(n), since it needs to traverse the entire list to find the tail.

Now let us look at how to insert an item in the middle of the list. In order to do this, we will need to know which item to insert it after (labelled the key here). It is also possible to design an algorithm that looks at the item before which we want to insert our new node, but that is very similar to this one and I am certain you will be able to figure that out on your own.

Node\* insertNode (int item, Node\* head, int key)  
{  
 Node\* newNode = createNode(item, **NULL**);  
 Node\* temp = new Node();  
 temp = head;  
 while (temp != **NULL** && temp->value != key) temp = temp->next;  
 if (temp != **NULL**)  
 {  
 newNode->next = temp->next;  
 temp->next = newNode;  
 }  
 return head;  
}

C++

In the function, we have a while loop that keeps taking our temporary node to the next node, until it has reached the end (is **NULL**), or is sitting on the key. If it does reach the end, the block of code inside the if statement does not run. If that block does run, then it changes the pointers so that the current node at temp, which is the one after which we want to insert the new node, points to the next node, and the new node points to where temp was previously pointing. The order in which we change the pointers is important. Remember that setting newNode->next to temp->next after we have changed where temp->next points (unlike what we have done above) will give us incorrect results.

Since, in the worst case, the key does not exist and we just traversed through the entire list for absolutely no reason, the time complexity of this function is O(n).

Notice that we are returning the head here as well. This means from the main function we call it like this:

head = insertNode(10, head, 5);

C++

You may raise the question of why we had to do this. This function would work just fine with a void return type, since it never even attempts to edit the head. The reason we kept it like this is again, to maintain similarity between the functions.

Now let’s take a look at removing the node. Removing the head is simple enough:

Node\* removeHead (Node\* head)  
{  
 head = head->next;  
 return head;  
}

C++

And in the main function:

head = removeHead(head);

C++

This function of course has a time complexity of O(1).

Removing some other node is a little more difficult since we need to find the node first.

Node\* removeNode (Node\* head, int item)  
{  
 Node\* cur = new Node();  
 Node\* prev = new Node();  
 cur = head;  
 while (cur->value != item && cur != **NULL**)  
 {  
 prev = cur;  
 cur = cur->next;  
 }  
 if (cur != **NULL**) prev->next = cur->next;  
 return head;  
}

C++

We needed two nodes to keep track of the current node and the previous node. When we found the node we wanted to remove, we set the previous node’s next pointer to the next pointer of the current node. If we did not find the node we wanted to remove, we would end up with a **NULL** value. We return head here as well for no better reason than we had before. This function has a time complexity of O(n) in the worst case, since we may have to go through the entire list pointlessly (pun intended).

Now let’s look at how to display the entire list (yay, our first void function):

void display (Node\* node)  
{  
 if (node != **NULL**)  
 {  
 cout<<node->value<<" ";  
 display(node->next);  
 }  
}

C++

This function is simple enough. We keep going to the next element and printing it until we reach the **NULL** value. This uses a recursive algorithm, but of course there are other ways to do this.

If we were to flip the cout and recursive display commands inside the if statement, we would end up with each function having to call the next function before being able to display its own results. This means the last node will be displayed first, meaning our list will be printed in reverse.

We could do one last thing. Actually, flipping our list so that every node points to the one before it instead of the one after it.

Node\* flipList (Node\* head)  
{  
 Node\* cur = new Node();  
 Node\* prev = new Node();  
 Node\* after = new Node();  
 cur = head;  
 while (cur != **NULL**)  
 {  
 after = cur->next;  
 cur->next = prev;  
 prev = cur;  
 cur = after;  
 }  
 return prev;  
}

C++

The order in which we changed things is very important here. We set the after node to the node next to the current one, then set the next pointer of the current node to the previous one (flipping it), then ‘incremented’ prev and cur. Finally, we return prev, since it is the new head.

An easier way to handle this would be if we had another pointer for each of our nodes, pointing to the previous node. Such a list is called a doubly linked list (as opposed to our current singly linked list).