Chapter 4: Link-Based Implementations

Concepts

Arrays

When you declare an array – either statically or dynamically – all the memory for the array is allocated at the same time. The amount of memory allocated is the *size-of-data-type x amount-of-elements*.

Example: If we're declaring an array of chars with a length of 10, it would be declared like this:

```
char arr[10];
```

The size of a char is 1 byte, so the amount of memory allocated is **1 byte x 10 positions = 10 bytes.** If we tried to output the address of the array like this: `cout << arr;` it would give us the address of the 0th item in the list. To get the item at index 1, we could take the `arr` address, plus 1 byte over in memory. To get the item at index 2, we could take the `arr` address, plus 2 bytes over in memory.

Pros and cons of the array

This is why accessing elements of an array is essentially *instantaneous*; we know where the next element is stored in memory. However, the down side of arrays is that, if we want to "resize" an array, we need to do the following steps:

1. Create a new dynamic array of a larger size
2. Copy the values from the old array over to the new array
3. Free the memory for the old array
4. Update the array pointer to point to the new array address

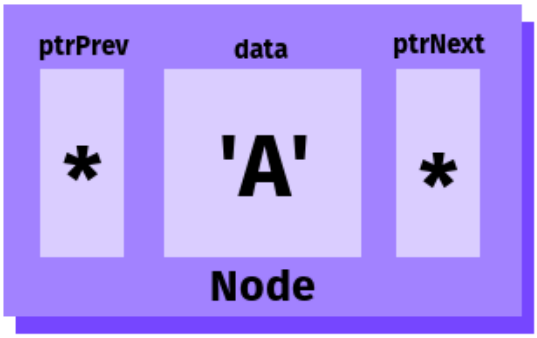
If we had to resize the array often, this would actually be a costly procedure. Copying data around takes a while, and as the array grows, it will just get slower and slower.

Linked implementations

Another way we can create a linear data structure is to only allocate elements for the list as-needed. However, each time we allocate memory for a new element, it will be in a seemingly-random space in memory; the elements will no longer be *contiguous* in memory like with an array.

Because of this, we have to *wrap* our elements in their own data type – usually, a **Node**.

A Node contains the single element's data, as well as one or more pointers: A **ptrNext**, and sometimes a **ptrPrevious**.




```
struct Node
{
    public:
    Node();

    Node* m_ptrNext;
    Node* m_ptrPrev;

    DATATYPE m_data;
};
```

This is just one aspect of a linked data structure. With a data structure like a Linked List, we require two structures: a Node, and a List class.

Creating Nodes

With a link-based structure, we only allocate new memory as-needed. Instead of allocating memory for *n* elements all at once, we start with an empty structure and dynamically create a single Node every time an "Insert" or "Add" function is called.

When a new Node is created, we need to set its data and its ptrPrev/ptrNext pointers, if there are any other items in the list.

Creating a new Node

```
Node* newNode = new Node;
newNode->m_data = newData;
```

Setting its pointers

```
newNode->m_ptrPrev = someNodePtr;
newNode->m_ptrNext = otherNodePtr;
```

If there are no other Nodes to point to as the previous or next (such as, if the Node is the first or last item in the list), then the ptrPrev/ptrNext pointers should be set to **nullptr**.

List class

The over-arching class that contains the functions needed to work with the list – Add, Remove, Find, etc. - as well as store pointers to the **first item** and (sometimes) the **last item** is usually called a **Linked List**.

If a Linked List's Nodes only store pointers to the next item, it is a singly linked list.

If a Linked List's Nodes store pointers to both the previous and the next item, it is a doubly linked list.

A Linked List declaration usually looks like this:

```
class LinkedList
{
    private:
    Node* m_ptrFirst;
    Node* m_ptrLast;
    int m_itemCount;

    public:
    // Functions go here
};
```

Pros and cons of a linked structure

Notice that the Linked List only stores pointers to the first item and the last item – it doesn't store pointers to any items in-between. In practice, how would we? We would end up resorting to using arrays to store all the data, and that would defeat the purpose of a Linked List.

So, if we want to get to a certain item in the list, we have to **traverse** the list instead, starting at the beginning, and stepping forward ***i*** times.

So what are the trade-offs here? Why use a Linked List vs. an Array?

- With an array, you have to allocate all the elements' memory all at once – can be a waste of space until all the elements are filled.
- With a linked list, you only allocate new memory when a new element is added.
- With an array, if you need to resize the array, you have to allocate more space and copy all the elements over – this is costly.
- There is no big "resize" function of a linked list, because items are only added as-needed. Each insert essentially is a mini-resize.
- With an array, you can instantaneously access an element at position ***i*** by taking the array's address, plus *i x data-size* bytes.
- With a linked list, you cannot instantaneously access an element at position ***i***; instead, you must begin at the first node, and traverse forward (via the **ptrNext pointer**) *i* times. This adds to the access time.

It is all about design and what you're implementing: An Array may be a better structure if you're accessing elements frequently, but not adding new items very often.

On the flip side, a Linked List may be better if you're adding items frequently, but accessing them less often.

Core Linked List methods, Page 139

With our data structures, we generally want to be able to **add new items**, **remove items**, and **search for an item**. We usually also want helper functions such as “Is This Empty?”, “Clear All Items”, or even “Get Frequency Of Some Value”.

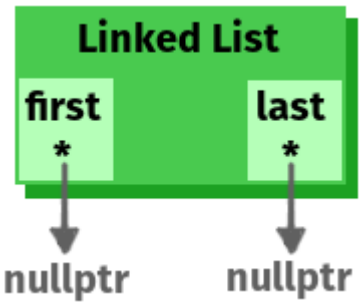
Let’s look at the functionality of these from a visual point of view.

PushBack

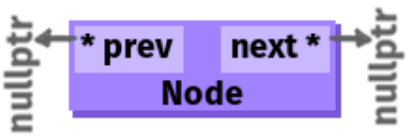
The “PushBack” function will add an item to the **end** of the list. We need to keep in mind that how we insert something will depend on whether the list is empty or not.

Linked List is empty

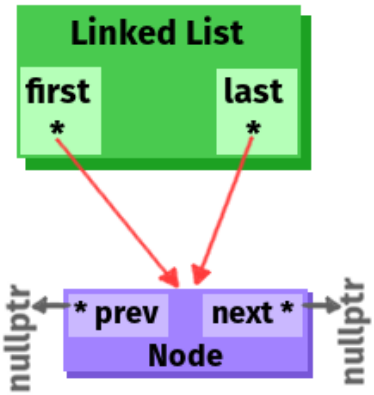
If the Linked List is empty, then there are no neighbor pointers to set up – we simply create a new node, and set it as the Linked List’s **first** and **last** item.



1. List is empty; the **first** and **last** pointers are both pointing to nullptr, and the **itemCount** variable is set to 0.



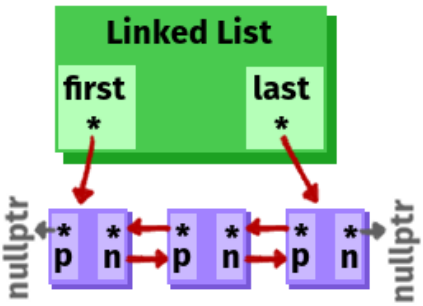
2. Allocate data for a new Node. Set up its data, and make sure its **prev** and **next** pointers are pointing to nullptr.



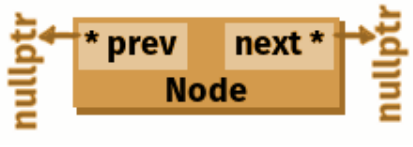
3. Set the List’s **first** and **last** pointers both to the new Node. Increment **itemCount** by 1.

Linked List is not empty

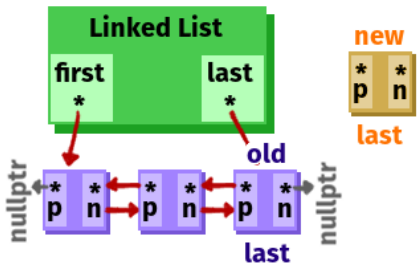
If the Linked List already has some items, then we need to add a new last item to the list. This also requires updating the neighbor pointer of the current-last-item.



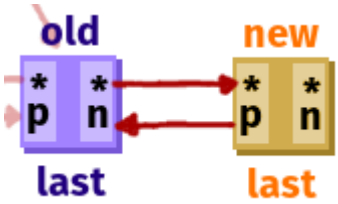
1. The List already has **first** and **last** pointers pointing to something.



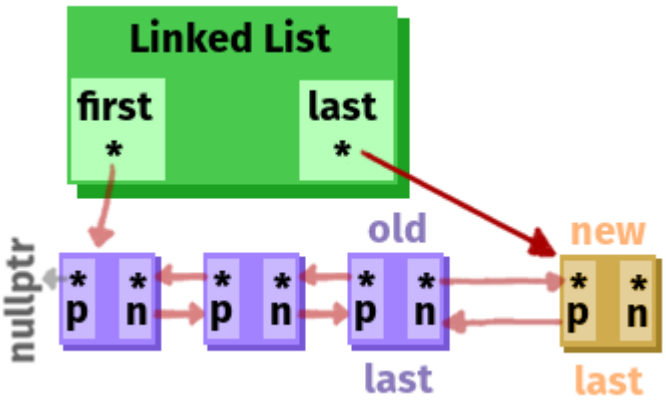
2. Allocate data for a new Node. Set up its data, and make sure its **prev** and **next** pointers are pointing to nullptr.



3. The new Node will be the **new-last**, and the List's current last node is **old-last…**



4. Set the **old-last**'s **next** pointer to the **new-last**, and set the **new-last**'s **previous** pointer to the **old-last**.



5. Then update the List's **last** pointer to point to the **new-last**. Make sure to add 1 to the **itemCount**.

PushFront

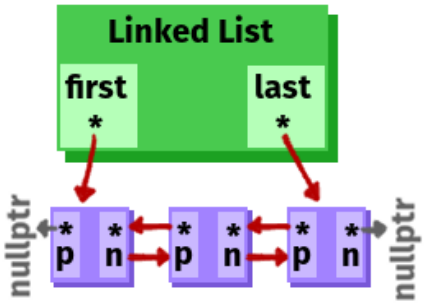
The "PushFront" function will add an item to the **beginning** of the list. We need to keep in mind that how we insert something will depend on whether the list is empty or not.

If the list is empty, then the functionality will be the same as for **PushBack** when the list is empty

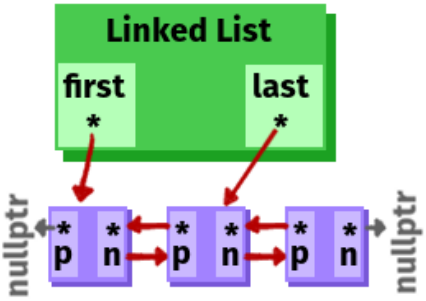
If the list is not empty, then the functionality will be *similar* as for PushBack, except you're working with the **first** Node, and placing the new node *before* it in the list to create a new-first Node.

PopBack

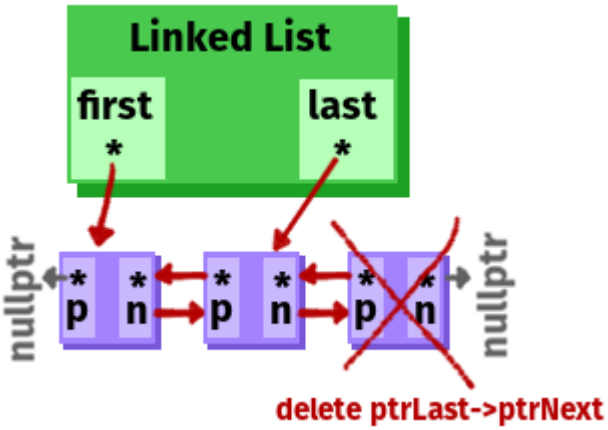
The "PopBack" function will remove the Node at the **end** of the list. This requires updating what Node is **last**, as well as updating that last Node's **next** pointer, and deleting the **old-last** Node.



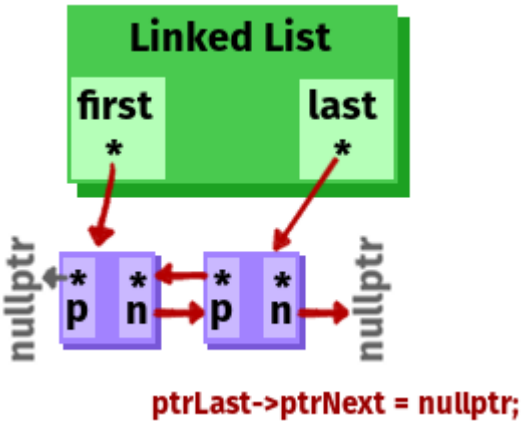
1. The List already has **first** and **last** pointers pointing to something.



2. Update the **last** pointer to point to the second-to-last Node.



3. Delete the Node at the end of the list; you will have to access it as **ptrLast→ptrNext** now.



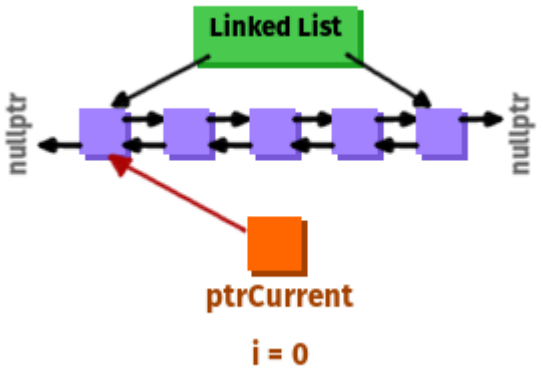
4. Update the new **last** Node's **ptrNext** to nullptr.

PopFront

The "PopFront" function will remove the Node at the **beginning** of the list. This requires updating what Node is **first**, as well as updating that last Node's **prev** pointer, and deleting the **old-first** Node.

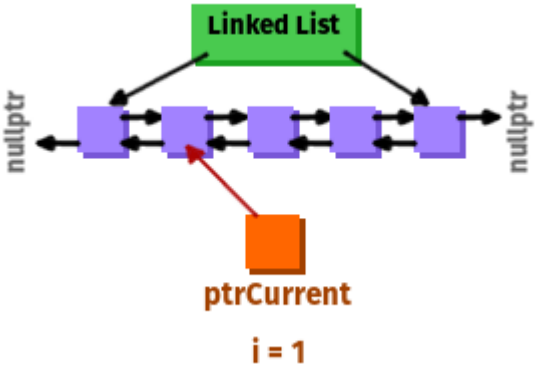
GetAtIndex

The "GetAtIndex" function allows us to access the ***i***-th Node in the list. To do this, we have to begin at the first Node and traverse forward ***i*** amount of times.



1. First, we create a Node pointer that will point at one of the existing nodes. The **ptrCurrent** traversal Node begins at the first item in the List, and we have an integer counter that starts at 0.

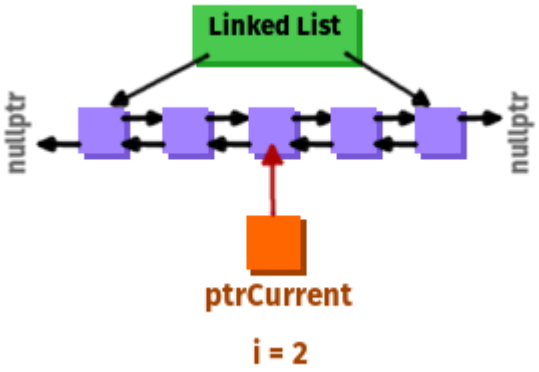
NOTE: We're not using the traversal node to allocate new memory! We're only pointing at *existing* memory!



2. We're going to keep traversing forward in the list, until our counter variable gets to the index that we want to arrive at.

To move the **ptrCurrent** traversal node forward, we use

```
ptrCurrent = ptrCurrent->ptrNext;
```



3. Once we get to the proper index, we stop looping.

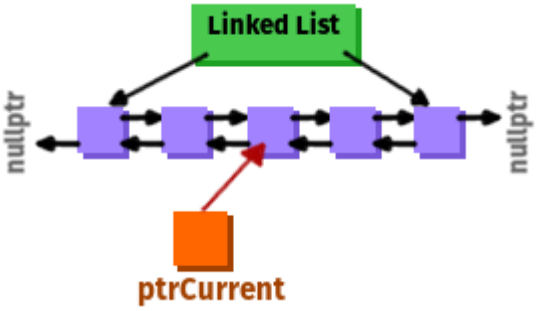
```
return ptrCurrent;

return ptrCurrent->data;
```

4. We return that Node, or the Node's data (depending on what the return type is.)

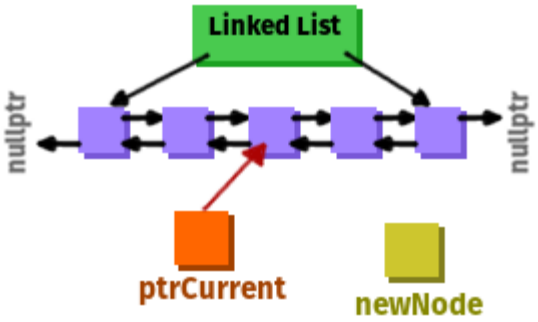
InsertAt

The “Insert” function will allow us to add an item in the middle of a list, in-between two nodes. This requires updating prev/next pointers for three different nodes, and it also requires that we traverse the list first to find the appropriate position.

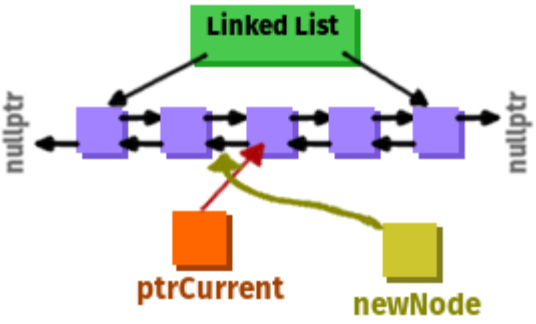
If you’ve implemented the “GetAtIndex” function to return a Node*, you can skip the traversal step simply by calling it. Once you have the appropriate index, it’s a matter of creating a new node and updating the neighbors.



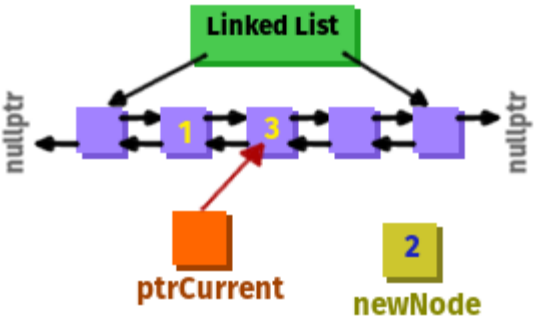
1. Find the Node at the index where we’re going to insert the new node.



2. Create a new Node

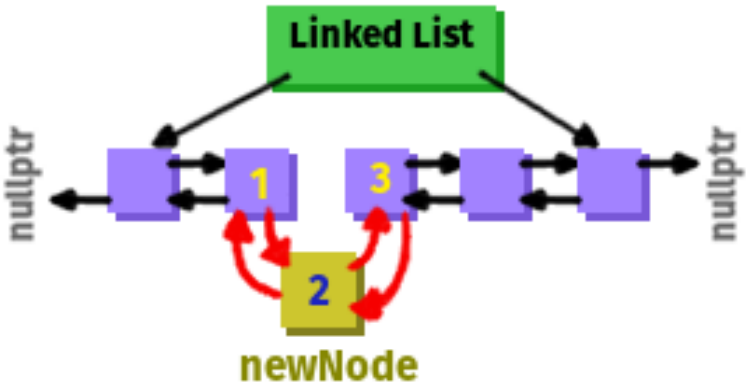


3. We want to insert the **newNode** at the current position of **ptrCurrent**, so the **ptrCurrent** will be moved forward 1 position, and the item before **newNode** will be the Node currently behind **ptrCurrent**.




```
Node* ptrOne = ptrCurrent->ptrPrev;
Node* ptrTwo = newNode;
Node* ptrThree = ptrCurrent;
```

4. It might help to create some extra “helper pointers” to keep things straight.



5. Update the following:
   - ptrOne's ptrNext is ptrTwo
   - ptrTwo's ptrPrev is ptrOne
   - ptrTwo's ptrNext is ptrThree
   - ptrThree's ptrPrev is ptrTwo
6. Don't forget to add 1 to the Linked List's **itemCount**.

GetBack

If the **first** Node isn't nullptr, then return its data.

GetFront

If the **last** Node isn't nullptr, then return its data.

Other types of linked lists

For these examples, we are using a "Doubly Linked List". This means that the Nodes of the list keep track of the **previous** and **next** items.

In a "Singly Linked List", the Nodes only keep track of the **next** item.

In a “Circularly Linked List”, the **last** Node of the list has its **next** pointer set to the **first** Node of the list, and vice versa if the Nodes store the **previous** items.

Linked List code

Paste in your Linked List code here for reference later on.

PushBack

paste code here

PushFront

paste code here

InsertAtIndex

paste code here

PopBack

paste code here

PopFront

paste code here

RemoveAtIndex

paste code here

GetAtIndex

paste code here

GetBack

paste code here

GetFront

paste code here