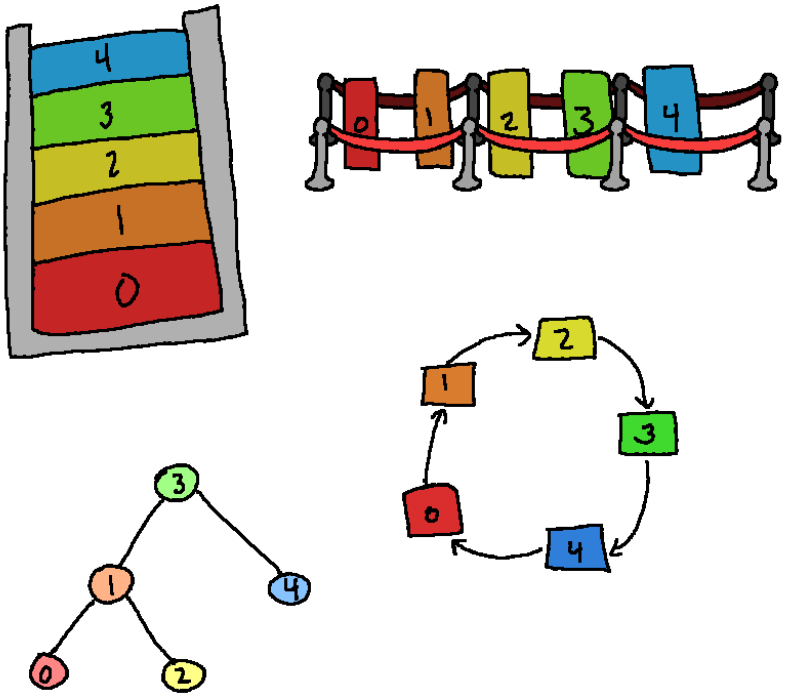
# Rachel's
# Data Structures Notes
**(Work in progress)**




**A more concise overview of topics by Rachel Singh**

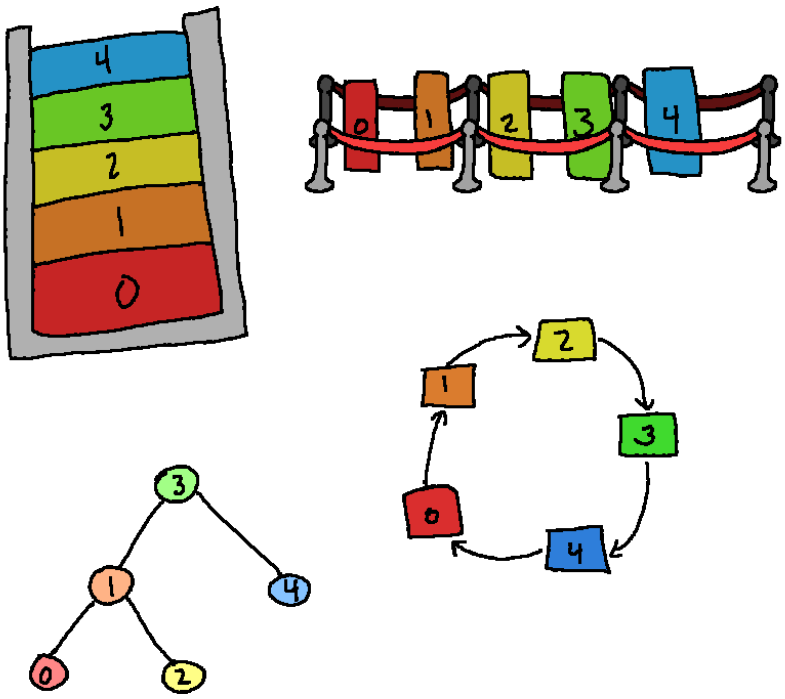
This work is licensed under a
Creative Commons Attribution 4.0 International License.

Last updated November 8, 2020

# Contents

# Topic 1

# Introduction to Data Structures and Algorithm Analysis



## 1.1 What are data structures?

A **data structure** is an object (a class, a struct - some type of **structured** thing) that holds **data**. In particular, the entire job of a data structure object is to store data and provide an interface for **adding, removing,** and **accessing** that data.

**"Don't all the classes we write hold data? How is this different from other objects I've defined?"**

In the past, you may have written classes to represent objects, like perhaps a book. A book could have member variables like its title, isbn, and year published, and some methods that help us interface with a book.

| **Book** |
|---|
| -title : string<br>-isbn : string<br>-year : int |
| +Setup() : void<br>+Display() : void |

However - this is not a data structure.
We *could*, however, use a data structure to *store* a list of books.

A data structure will store a series of some sort of data. Often, it will use an **array** or a **linked list** as a base structure and functionality will be built on top.

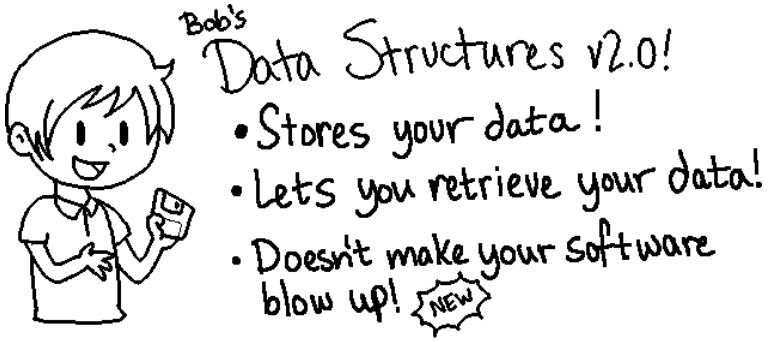
Ideally, the user doesn't care about *how* the data structure works, they just care that they can **add, remove,** and **access** data they want to store.

| **SmartBookArray** |
|---|
| -bookArray : Book[ ]<br>-arraySize : int<br>-bookCount : int |
| +AddBook( Book newBook ) : void<br>+RemoveBook( int index ) : void<br>+RemoveBook( string title ) : void<br>+GetBook( int index ) : Book<br>+FindBookIndex( string title ) : int<br>+DisplayAll() : void<br>+Size() : int<br>+IsEmpty() : bool |

Ideally, when we are writing a data structure, it should be:

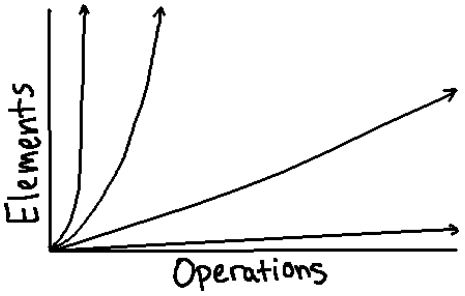
- Generic, so you could store *any data type* in it.
- Reusable, so that the data structure can be used in many different programs.
- Robust, offering exception handling to prevent the program from crashing when something goes wrong with it.
- Modular, handling the inner-workings of dealing with the data, without the user (or other programmers working outside the data structure) having to write special code to perform certain operations.

The way I try to conceptualize the work I'm doing on a data structure is to pretend that I'm a developer that is going to create and sell a C++ library of data structures that *other developers at other companies* can use in their own, completely separate, software projects. If I'm selling my data structures package to other businesses, my code should be dependable, stable, efficient, and relatively easy to use.



## 1.2 What is algorithm analysis?

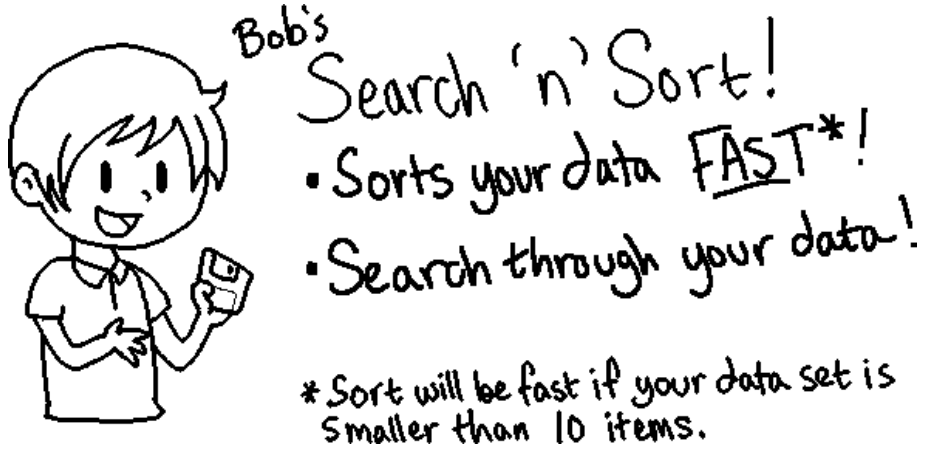
Algorithm Analysis is the process of figuring out how **efficient** a function is and how it scales over time, given more and more data to operate on.




This is another important part of dealing with data structures, as different structures will offer different trade offs when it comes to the efficiency of **data access** functions, **data searching** functions, **data adding** functions, and **data removal** functions. Every operation takes a little bit of processing time, and as you iterate over data, that processing time is multiplied by the amount of times you go through a loop.

**Example: Scalability**
Let's say we have a sorting algorithm that, for every $n$ records, it takes $n^2$ program units to find an object. If we had 100 records, then it would take $100^2 = 10,000$ time-units to sort the set of data.

What the time-units are could vary - an older machine might take longer to execute one instruction, and a newer computer might process an instruction much more quickly, but we think of algorithm complexity in this sort of generic form.



**Example: Efficiency of an Array-based structure**

| kansas | missouri | arkansas | ohio | oklahoma |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

In an array-based structure, we have a series of **elements** in a row, and each element is accessible via its **index** (its position in the array). Arrays allow for random-access, so **accessing** element #2 is instant:

```
1 cout << arr[2];
```

No matter how many elements there are ($n$), we can access the element at index 2 without doing any looping. We state that this is $O(1)$ ("Big-O of 1") time complexity for an **access** operation on an **array**.

However, if we were **searching** through the unsorted array for an item, we would have to start at the beginning and look at each item, one at a time, until we either found what we're looking for, or hit the end of the array:

```
1 for ( int i = 0; i < ARR_SIZE; i++ )
2 {
3         if ( arr[i] == searchTerm )
4         {
5                 return i;       // found at this
   position
6         }
7 }
8 return -1;       // not found
```

For **search**, the *worst-case scenario* is having to look at *all elements of the array* to ensure what we're looking for isn't there. Given $n$ items in the array, we have to iterate through the loop $n$ times. This would end up being $O(n)$ ("Big-O of $n$") time complexity for a **search** operation on an **array**.

We can build our data structures on top of an array, but there is also a type of structure called a **linked** structure, which offers its own pros and cons to go with it. We will learn more about algorithm analysis and types of structures later on.

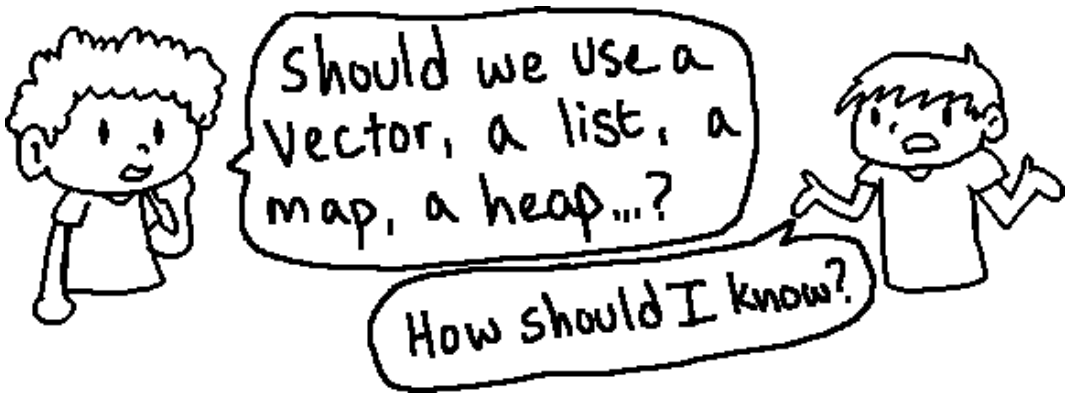
## 1.3 The point of a Data Structures class

Of course, plenty of data structures have already been written and are available out there for you to use. C++ even has the **Standard Template Library** full of structures already built and optimized!

| | |
|---|---|
| `vector` | http://www.cplusplus.com/reference/vector/vector/ |
| `list` | http://www.cplusplus.com/reference/list/list/ |
| `map` | http://www.cplusplus.com/reference/map/map/ |

**"So why are we learning to write data structures if they've already been written for us?"**

While you generally *won't* be rolling your own linked list for projects or on the job, it *is* important to know how the inner-workings of these structures operate. Knowing how each structure works, its tradeoffs in efficiency, and how it structures its data will help you **choose** what structures to use when faced with a **design decision** for your software.



# Topic 2

# Linked Lists

## 2.1 Coming from arrays...

Before taking a Data Structures course, you've probably only managed data in your program via **arrays**.

When we declare an array in C++, we must give it a size (whehter we're making a dynamic array or a vanilla array). This ensures that memory can be allocated so that each element of the array is side-by-side with other elements.

### Investigating arrays and memory addresses:

Let's write a simple program to look at how arrays are stored in memory, in case you don't remember.

First, how big is one element? We can use the `sizeof` function to find out how big, in bytes, a given data type is:

```
1 cout << "The size of a int is: "
2      << sizeof( int ) << " bytes." << endl;
```

```
The size of a int is: 4 bytes.
```

One integer is 4 bytes. Next, we can declare an array of integers and look at how many bytes that takes up:

```
1 const int SIZE = 10;
2 int intArr[SIZE];
3 cout << "Size of a int array with "
4      << SIZE << " elements: "
5      << sizeof( intArr ) << " bytes." << endl;
```

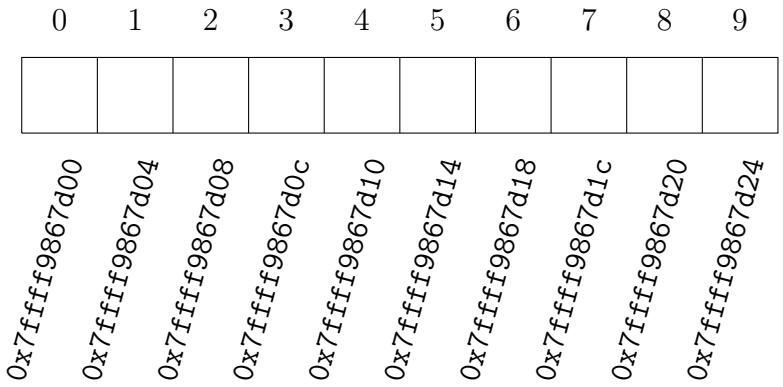
```
Size of a int array with 10 elements: 40 bytes.
```

Now, let's look at the memory addresses of each element:

```
1 cout << "Memory addresses for the int array:" << endl;
2 for ( int i = 0; i < SIZE; i++ )
3 {
4     cout << "Element " << i << ": " << &(intArr[i]) <<
    endl;
5 }
```

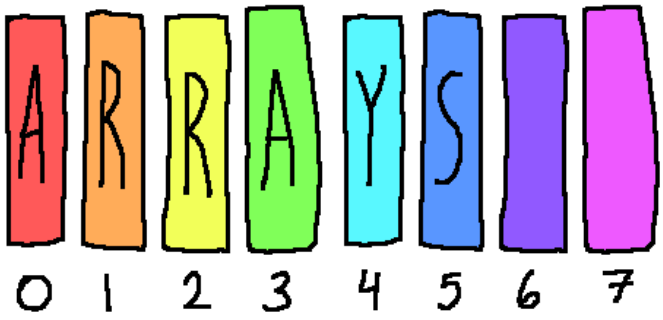
```
Memory addresses for the int array:
Element 0: 0x7ffff9867d00
Element 1: 0x7ffff9867d04
Element 2: 0x7ffff9867d08
Element 3: 0x7ffff9867d0c
Element 4: 0x7ffff9867d10
Element 5: 0x7ffff9867d14
Element 6: 0x7ffff9867d18
Element 7: 0x7ffff9867d1c
Element 8: 0x7ffff9867d20
Element 9: 0x7ffff9867d24
```

Each time this program runs, we will have different memory addresses for the `intArr`, but those addresses **will always be contiguous in memory, 4 bytes apart.**




**Hexadecimal**
When we write in hexadecimal, we want each number to only take up one character space. So, we have 0-9 to be 0 through 9, but then 10 is A, 11 is B, 12 is C, 13 is D, 14 is E, and 15 is F.

**Address of the beginning of the array**
The address of `intArr` is the same as the address of `intArr[0]`.



## Pros and Cons of arrays for storing data:

**Pros:**

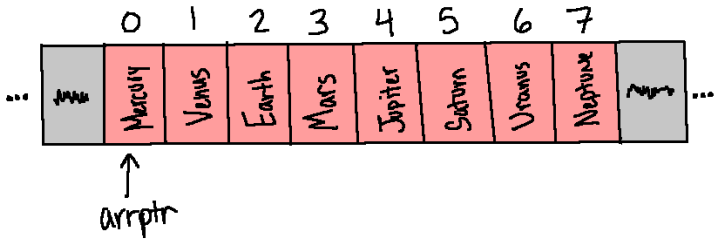
- **Random access is instant:** This means, we can access an element at *any arbitrary index* instantaneously. How? Well, we have the address of the 0th element, and to get an item at position $i$, we just have to add `i * sizeof(int)`. A simple math operation is virtually instant.

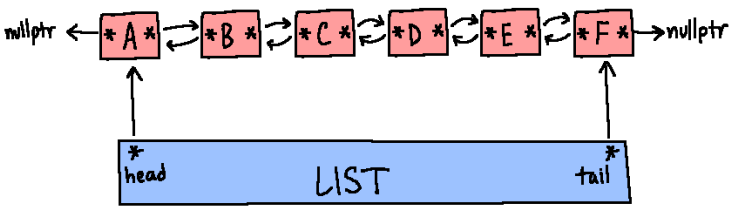
```
AddressOf( i ) = AddressOf( 0 ) + i * sizeof( dataType )
```

**Cons:**

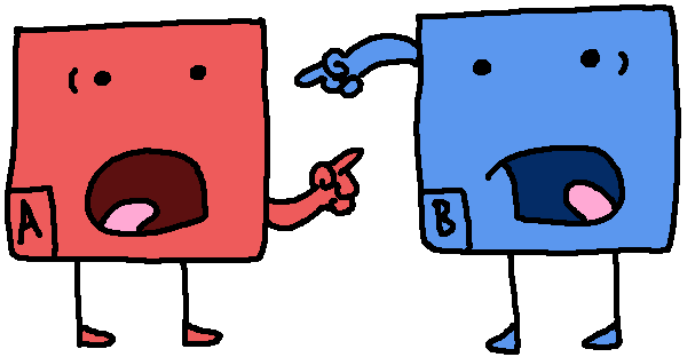
- **Resizing a dynamic array is costly:** Any time our array is full, to resize it we have to allocate memory for a *new array* of a bigger size. Then, we have to copy each element from the old array to the new array. This takes time any time resize occurs.
- **We have to over-estimate the array size:** We don't want to have to resize the entire array every time a new item is inserted, so we generally will resize it by some quantity each time (not just adding 1 to the size of the array). This means we're allocating more space than we're *actually using* if the array isn't full.

The study of Data Structures is all about the trade-offs between different data types: Some are faster to access data but slower to insert new data and vice versa. Instead of storing data in an array-based structure, we can make a tradeoff of **less memory used, instant insertion of new data** in exchange for **slower access** using a linked structure.



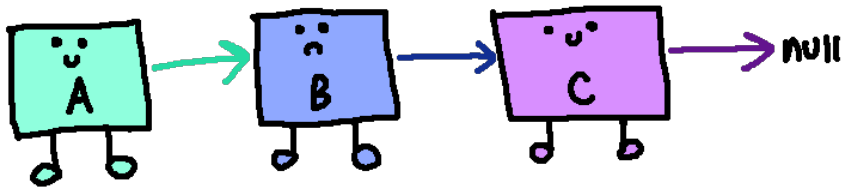



## 2.2 Anatomy of a Node




The **Node** is one of the two structures we build when implementing a linked list or other linked type structure. The node is responsible for storing the data itself, and storing one or more pointers.

### Singly-linked List's Nodes

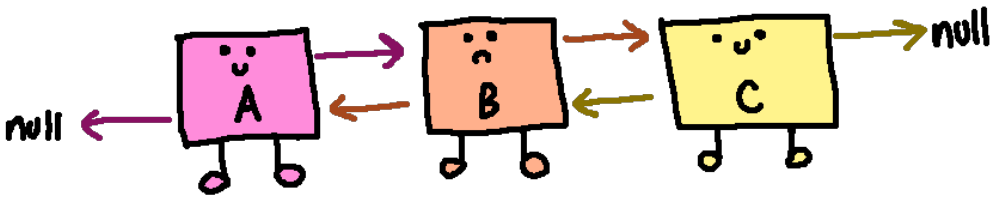



With a **singly-linked list**, the nodes only point to the **next Node** after them. These nodes don't know what comes before each of themselves.

| **SinglyLinkedNode<TYPE>** | | |
|---|---|---|
| + SinglyLinkedNode<TYPE>() | | |
| + data | : | TYPE |
| + ptrNext | : | Node<TYPE>* |

---

### Doubly-linked List's Nodes




| **DoublyLinkedNode<TYPE>** | | |
|---|---|---|
| + DoublyLinkedNode<TYPE>() | | |
| + data | : | TYPE |
| + ptrNext | : | Node<TYPE>* |
| + ptrPrev | : | Node<TYPE>* |

With a **doubly-linked list**, the nodes point to both the **next Node** and the **previous node**.

**Warning!**
Remember that any class with pointers in it should be setting those pointers to `nullptr` in its constructor!

A Node is simple enough that you could implement it as a **struct** instead of a **class**, but it's up to you. A Node is only used by the List (or linked structure) itself, the user wouldn't be accessing the Node or know about it.

**A doubly-linked Node:**

```
1  template <typename T>
2  struct Node
3  {
4  public:
5      Node();
6
7      Node<T>* ptrNext;
8      Node<T>* ptrPrev;
9
10     T data;
11 };
```

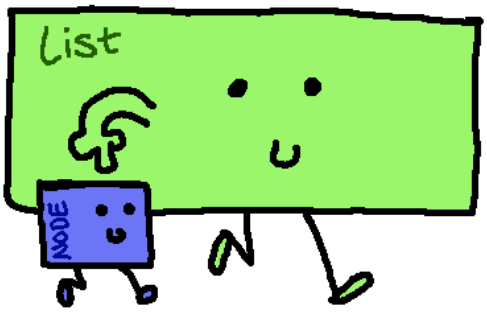
**A singly-linked Node:**

```
1  template <typename T>
2  struct Node
3  {
4  public:
5      Node();
6
7      Node<T>* ptrNext;
8
9      T data;
10 };
```

**Single vs. Double?**
I generally prefer to teach how to implement a doubly-linked list first, because the functionality is easier to implement when each Node is aware of what comes before itself.

Once you understand how a doubly-linked list works, you can figure out a singly-linked list; it just means having to iterate through the list more often.

## 2.3 Anatomy of a List




The List class itself is what handles interfacing with the "user" (or other programmers using your List). All[1] Data Structures will have **Access**, **Add**, **Remove**, and **Search** functionality, and those functions (and helper functions) are stored in our List.

The List will also keep track of the **first Node** in the list, and often also the **last Node**[2] by storing `Node*` pointers.

The list is also responsible for the **memory management** - it will allocate memory for a new `Node` when it's needed, and free the memory of a `Node` when it's to be removed.

[1] I think? Unless there are some weird ones I don't know about??

[2] You could get by without the last Node pointer, but it's just easier, man.

Generally, a List will have functionality like:

| **LinkedList<TYPE>** | | |
|---|---|---|
| + PushFront( newData : T ) | : | void |
| + PushBack( newData : T ) | : | void |
| + PushAtIndex( index : int, newData : T ) | : | void |
| | | |
| + PopFront() | : | void |
| + PopBack() | : | void |
| + PopAtIndex( index : int ) | : | void |
| | | |
| + GetFront() | : | T& |
| + GetBack() | : | T& |
| + GetAtIndex( index : int ) | : | T& |
| | | |
| + Clear() | : | void |
| + IsEmpty() | : | bool |
| + Size() | : | int |
| - ptrFirst | : | Node<TYPE>* |
| - ptrLast | : | Node<TYPE>* |
| - itemCount | : | int |

The **Push** functions are to add new items. There are three varieties: Add to the *start* of the list (PushFront), Add to the `end` of the list (PushBack), and Add to **somewhere in the middle** of the list (PushAtIndex).

The **Pop** functions are to remove items.

And the **Get** functions are to access data stored in Nodes.

Depending on the design of the List, sometimes the "AtIndex" functions won't be implemented.

**Linked List declaration:**

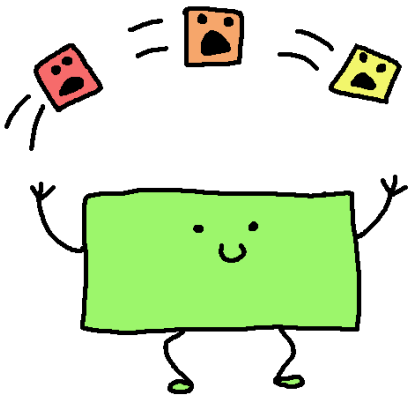
```
template <typename T>
class LinkedList
{
private:
    Node<T>* ptrFirst;
    Node<T>* ptrLast;
    int itemCount;

public:
    LinkedList();
    ~LinkedList();

    void PushFront( T newData );
    void PushBack( T newData );
    void PushAtIndex( int index, T newData );

    void PopFront() noexcept;
    void PopBack() noexcept;
    void PopAtIndex( int index );

    T& GetFront();
    T& GetBack();
    T& GetAtIndex( int index );

    void Clear();
    bool IsEmpty();
    int Size();
};
```

## 2.4 Functionality of a Doubly-Linked List

### The Destructor

The destructor should make sure that all memory is freed when the LinkedList is destroyed. I've moved this functionality into the `Clear()` function, so just make sure to call `Clear()` within the destructor.

We're going to look at the way functions work, assuming we're using doubly-linked Nodes. This is because it's more straightforward, and once you understand how doubly-linked lists work, you can figure out how a singly-linked list works (it just means more traversing).



## The Constructor

First and foremost - remember to set your pointers to `nullptr`! You will also want to set your `m_itemCount` to 0.

## Adding new data

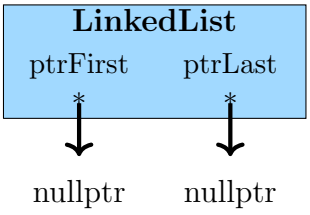
### PushFront

```
1 void PushFront( T newData );
```

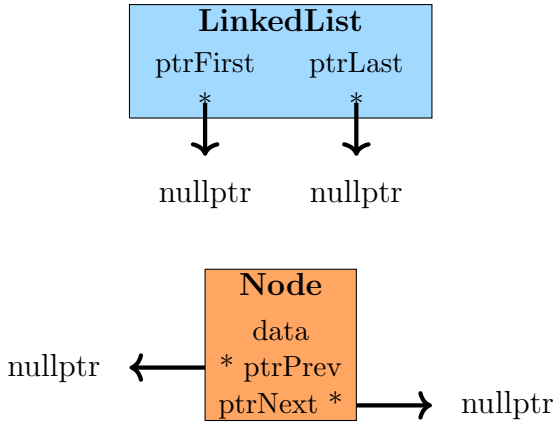
When adding a new item to our list, we need to cover two different possible scenarios: Adding the item to an *empty* List, and adding the item to a *non-empty* list.

**Push Front - Scenario 1 - Adding to an empty list:** We can check for this criteria with a simple if statement; if `itemCount` is 0.

**Initial state:** Starting off, we have just our LinkedList and no Nodes in memory. The List's `ptrFirst` and `ptrLast` will be initialized to `nullptr`.



**Step 1: Create Node** First, we allocate memory for one new `Node`. Remember that the Node contains a place to store the element's data, so we set up this new node with the `newData` from the function's parameter.

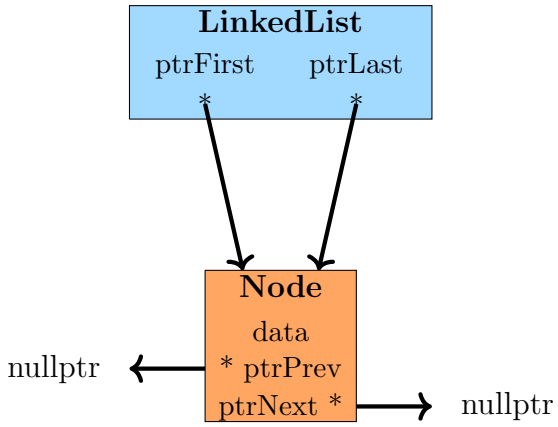



```
1 Node<T>* newNode = new Node<T>;
2 newNode->data = newData;
```

---

**Step 2: Update Pointers** Next, we need to update the LinkedList's `ptrFirst` and `ptrLast`. Since this new Node is the *only* item in the List, it is both the first *and* the last item.

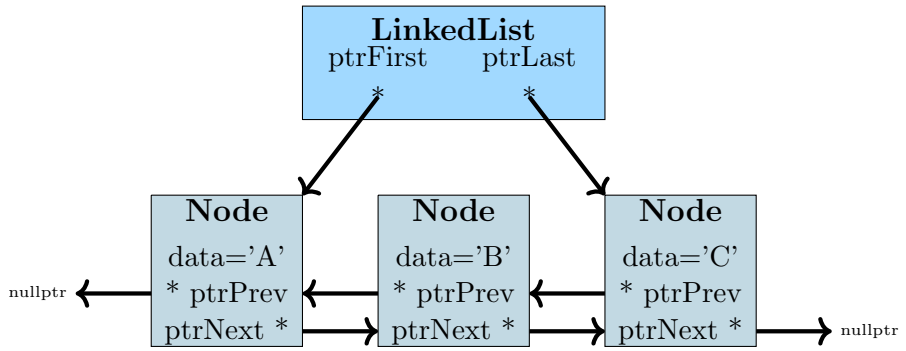
You will also need to increment the `itemCount` by 1.




```
1 ptrFirst = newNode;
2 ptrLast = newNode;
3 itemCount++;
```

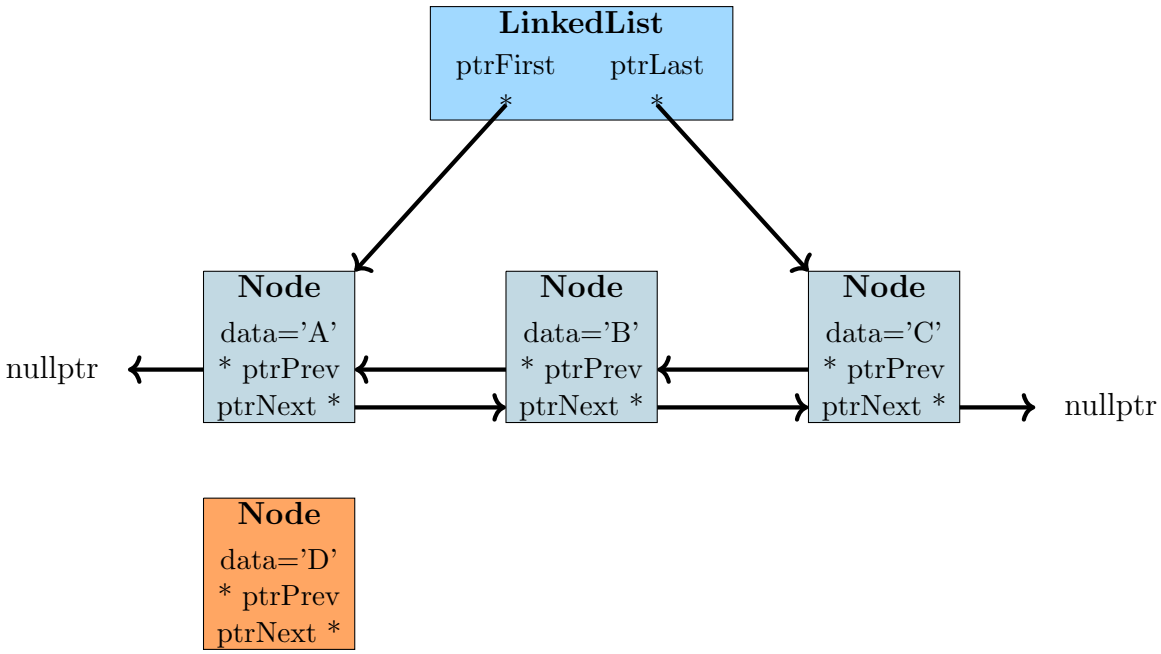
### Push Front - Scenario 2 - Adding to a non-empty list:

**Initial state:** The initial state here will be that there are one or more Nodes already in the list.




Since we are doing **PushFront** this means that we will be adding our Node to the *beginning* of the list, replacing the current first item.

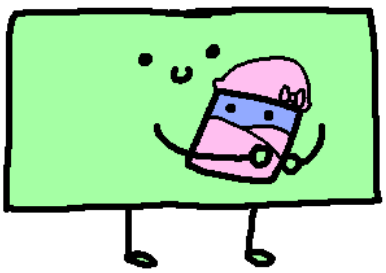
**Step 1: Create Node**



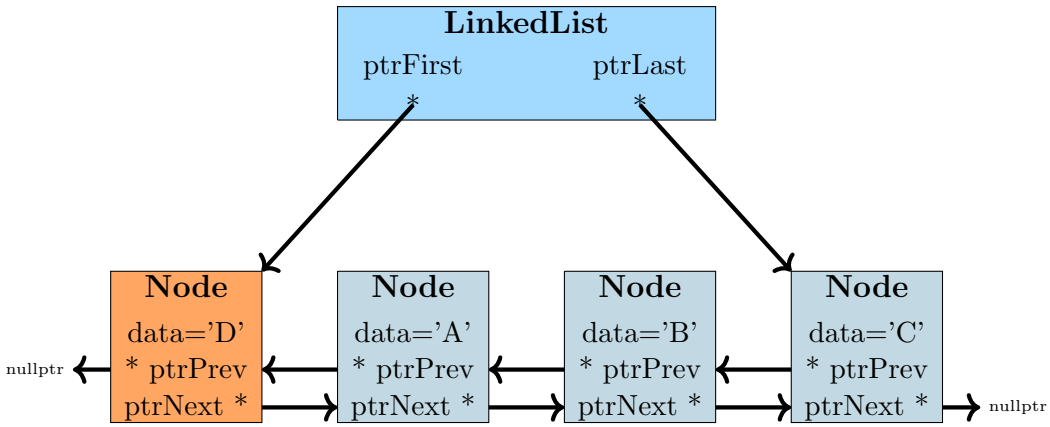

**New node created**

First, we allocate memory for one new `Node`. Remember that the Node contains a place to store the element's data, so we set up this new node with the `newData` from the function's parameter.

```
1 Node<T>* newNode = new Node<T>;
2 newNode->data = newData;
```



### Step 2: Update Pointers




Now we need to update the pointers. For **PushFront**, the updates are:

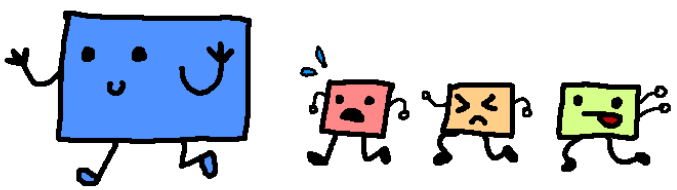
1. Set the newNode's `ptrNext` to the current `ptrFirst` of the List.
2. Set the current `ptrFirst` of the list's `ptrPrev` to the newNode.
3. Set the List's `ptrFirst` to point to the newNode.

And then don't forget to increment the item count.

```
1 newNode->ptrNext = ptrFirst;
2 ptrFirst->ptrPrev = newNode;
3 ptrFirst = newNode;
4 itemCount++;
```

---

Remember that these two scenarios are both part of the **PushFront** function - use an if statement to decide which scenario gets executed.



### PushBack

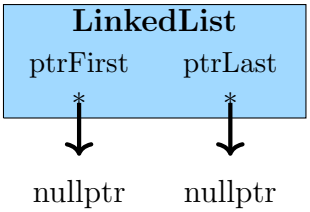
```
1 void PushBack( T newData );
```

Push back's functionality is similar to PushFront's, except we're working at the opposite end of the list - adding a new end item. We again have two scenarios: it's an empty list, or it's not.

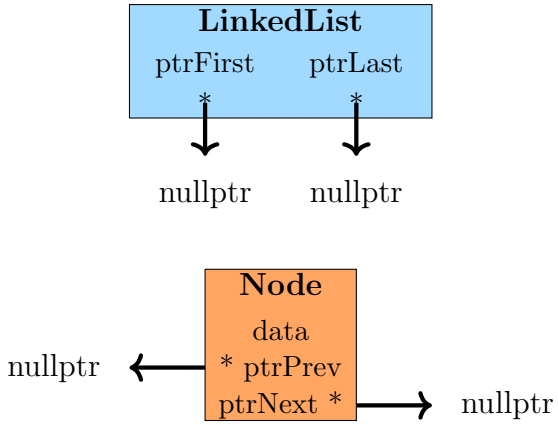
**Push Back - Scenario 1 - Adding to an empty list:** We can check for this criteria with a simple if statement; if `itemCount` is 0.

(This is the same process as with PushFront.)

**Initial state:** Starting off, we have just our LinkedList and no Nodes in memory. The List's `ptrFirst` and `ptrLast` will be initialized to `nullptr`.



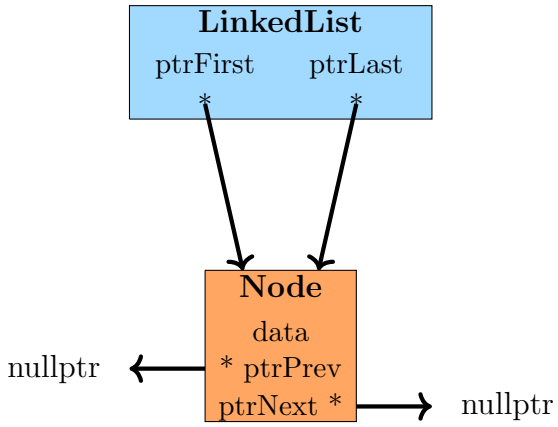
**Step 1: Create Node** First, we allocate memory for one new `Node`. Remember that the Node contains a place to store the element's data, so we set up this new node with the `newData` from the function's parameter.




```
1 Node<T>* newNode = new Node<T>;
2 newNode->data = newData;
```

---

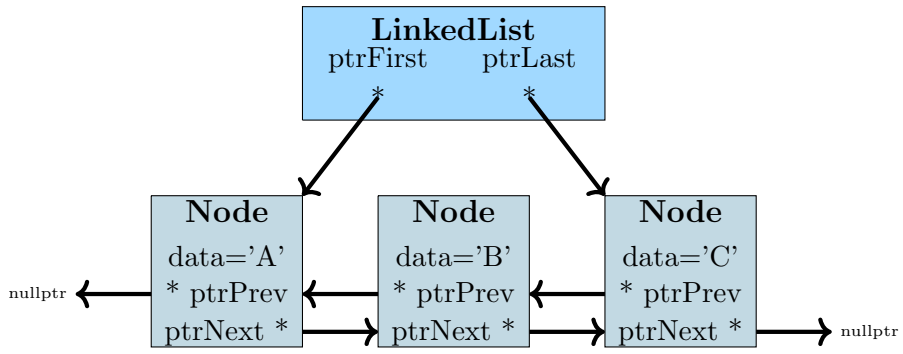
**Step 2: Update pointers** Next, we need to update the LinkedList's `ptrFirst` and `ptrLast`. Since this new Node is the *only* item in the List, it is both the first *and* the last item. You will also need to increment the `itemCount` by 1.




```
1 ptrFirst = newNode;
2 ptrLast = newNode;
3 itemCount++;
```

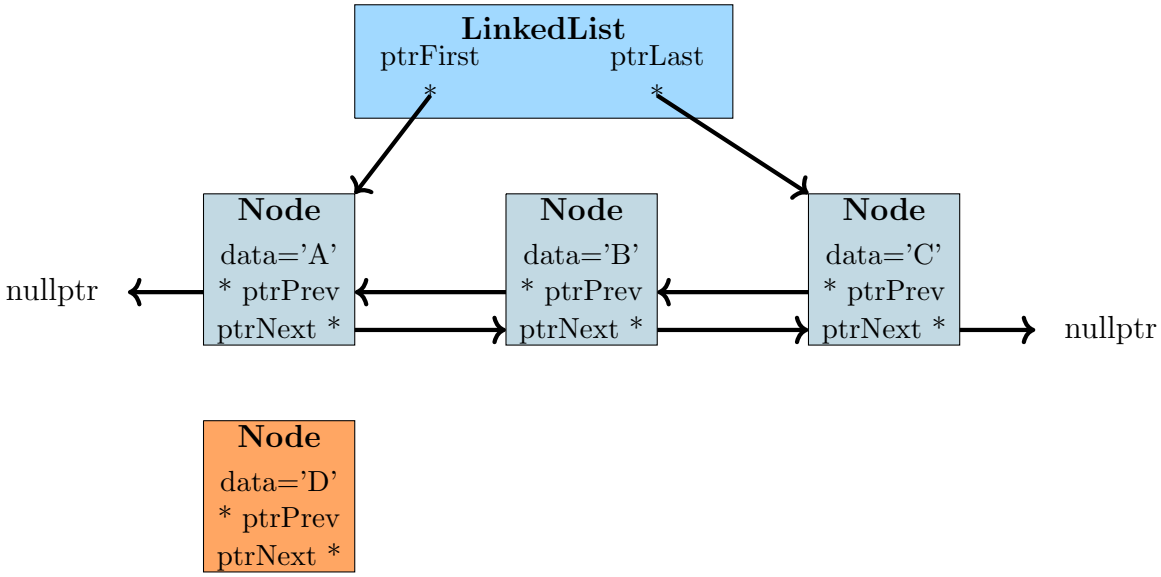
### Push Back - Scenario 2 - Adding to a non-empty list:

**Initial state:** The initial state here will be that there are one or more Nodes already in the list.




Since we are doing **PushBack** this means that we will be adding our Node to the *end* of the list, replacing the current last item.

**Step 1: Create Node**

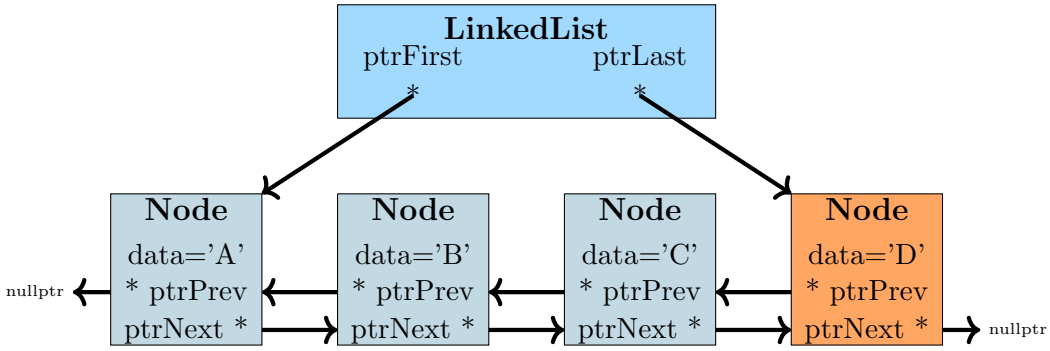



**New node created**

First, we allocate memory for one new `Node`. Remember that the Node contains a place to store the element's data, so we set up this new node with the `newData` from the function's parameter.

```
1 Node<T>* newNode = new Node<T>;
2 newNode->data = newData;
```

### Step 2: Update pointers




Now we need to update the pointers. For **PushBack**, the updates are:

1. Set the newNode's `ptrPrev` to the current `ptrLast` of the List.
2. Set the current `ptrLast` of the list's `ptrNext` to the newNode.
3. Set the List's `ptrLast` to point to the newNode.

And then don't forget to increment the item count.

```
1 newNode->ptrPrev = ptrLast;
2 ptrLast->ptrNext = newNode;
3 ptrLast = newNode;
4 itemCount++;
```

# Removing data

## PopFront

```
1 void PopFront() noexcept;
```

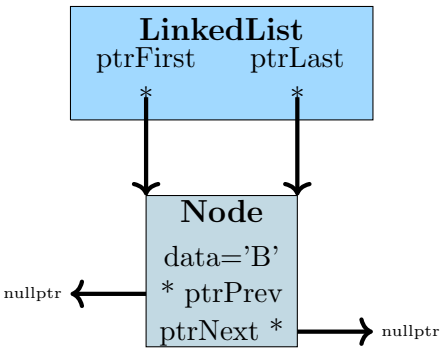
The **PopFront** function is responsible for removing the first item of the List and setting up a new first item (the old second item becomes the new first item).

The scenarios we have to consider here are whether we're **removing the last item**, or **there is more than one item in the List**.

---

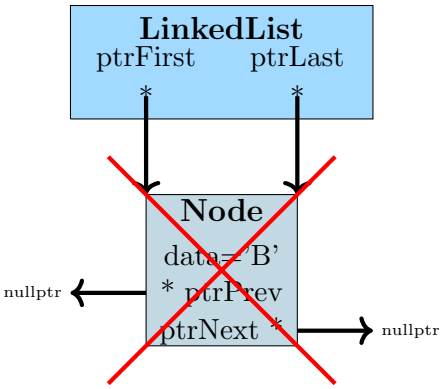
### Pop Front - Scenario 1 - Removing the last item:

**Initial state:** There is only one item in the List.



**Step 1: Free the memory** Use the `delete` command to free memory via either the `ptrFirst` or `ptrLast` pointer (doesn't matter which).

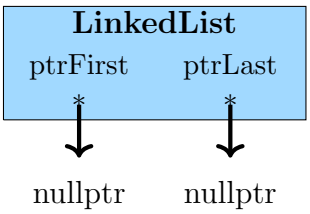
```
1 delete ptrFirst;
```




---

**Step 2: Update pointers** After that memory is freed, you will set the List's `ptrFirst` and `ptrLast` to both point to nullptr.

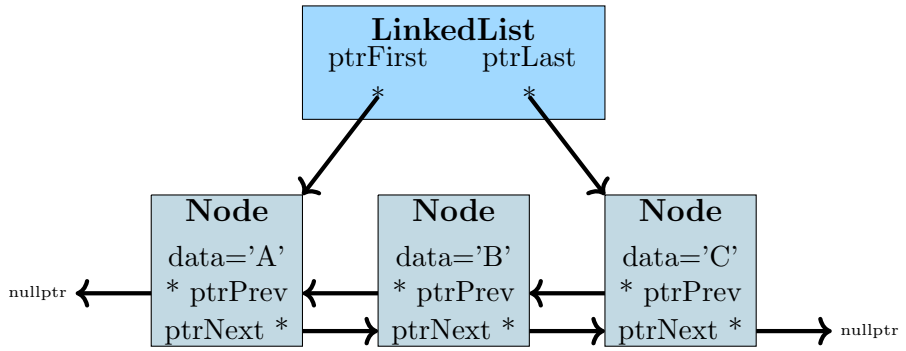
Don't forget to decrement the `itemCount` by 1.




```
1 ptrFirst = nullptr;
2 ptrLast = nullptr;
3 itemCount--;
```

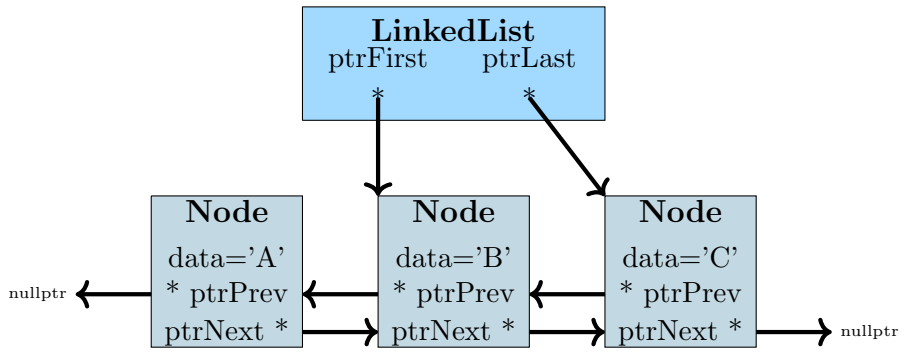
### Pop Front - Scenario 2 - There are 2 or more items in the list:

**Initial state:** There are two or more items in the list. We are going to remove the first item and set a new first in the list.



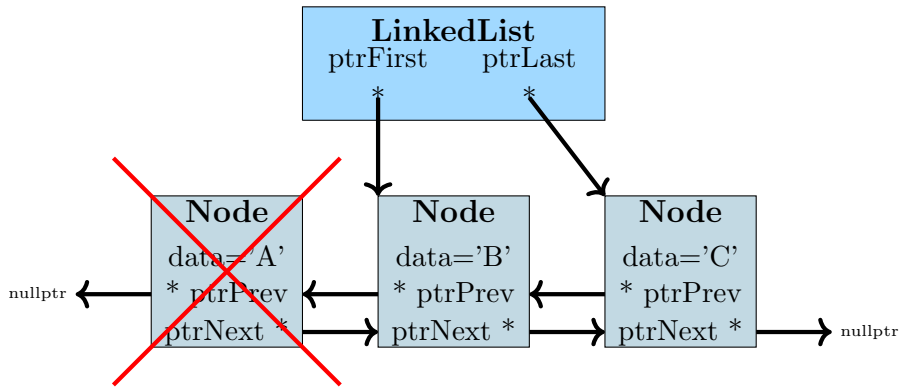

---

**Step 1: Set the new first** First, we update the list and tell it that the new first item is the `ptrNext` of the current `ptrFirst`...




```
1 ptrFirst = ptrFirst->ptrNext;
```

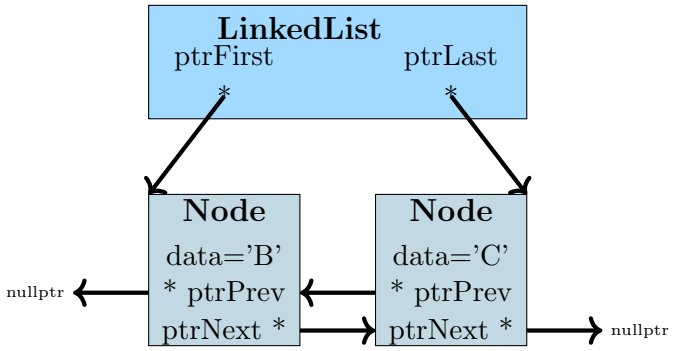
**Step 2: Delete the old first Node** You can do this by accessing the old-first with `ptrFirst->ptrPrev`.




```
1 delete ptrFirst->ptrPrev;
```

---

**Step 3: Update `ptrFirst`'s `ptrPrev` to point to nullptr.** And make sure to decrement the `itemCount`!




```
1 ptrFirst->ptrPrev = nullptr;
2 itemCount--;
```

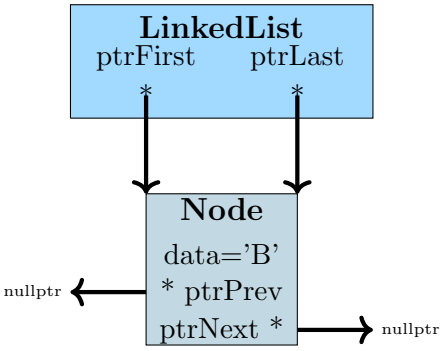
### PopBack

```
1 void PopBack() noexcept;
```

The **PopBack** function is responsible for removing the last item of the List and setting up a new last item (the old second-to-last item becomes the new last item).

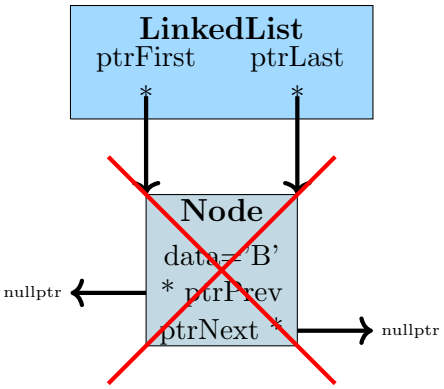
### Pop Back - Scenario 1 - Removing the last item:

**Initial state:** There is only one item in the List.



**Step 1: Free the memory** Use the `delete` command to free memory via either the `ptrFirst` or `ptrLast` pointer (doesn't matter which).

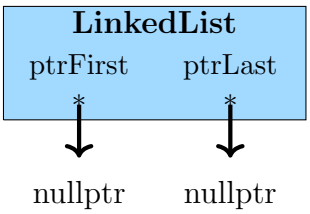
```
1 delete ptrFirst;
```




---

**Step 2: Update pointers** After that memory is freed, you will set the List's `ptrFirst` and `ptrLast` to both point to nullptr.

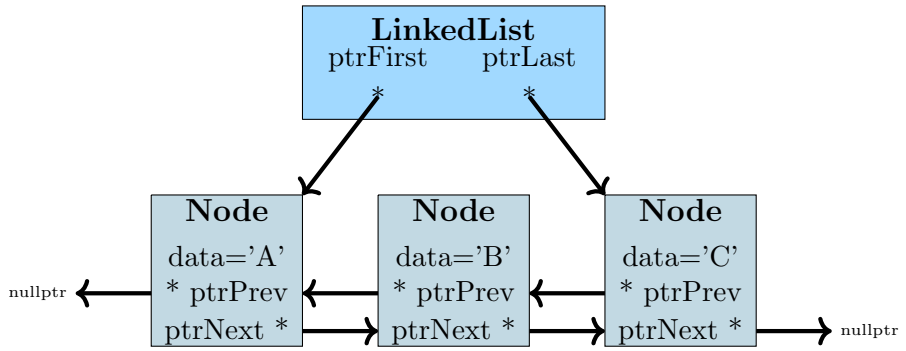
Don't forget to decrement the `itemCount` by 1.




```
1 ptrFirst = nullptr;
2 ptrLast = nullptr;
3 itemCount--;
```

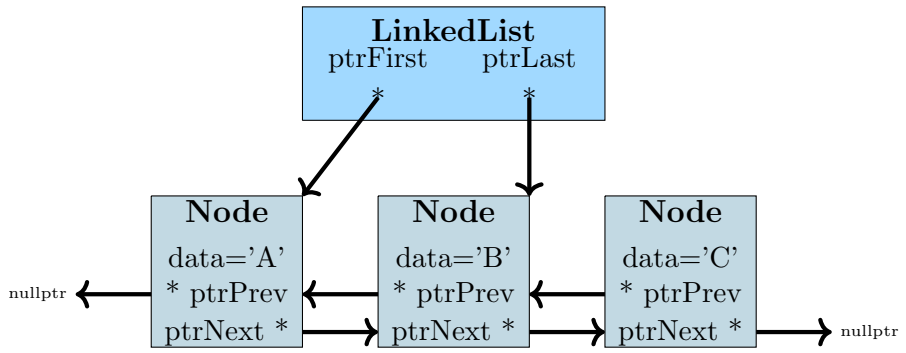
### Pop Back - Scenario 2 - There are 2 or more items in the list:

**Initial state:** There are two or more items in the list. We are going to remove the first item and set a new first in the list.



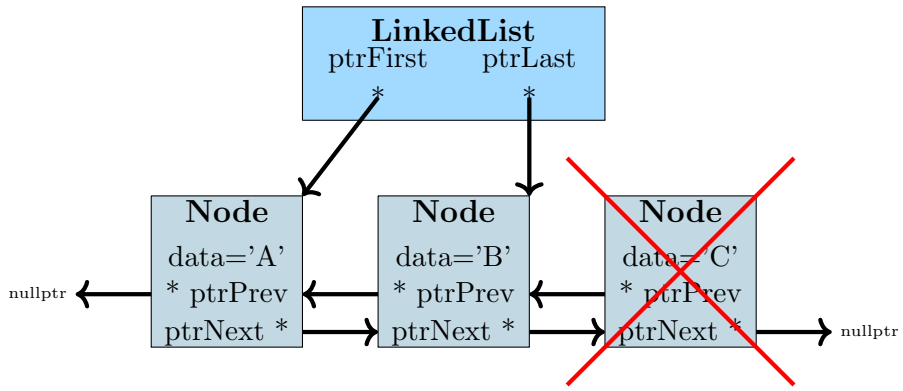

---

**Step 1: Set the new last** First, we update the list and tell it that the new last item is the `ptrPrev` of the current `ptrLast`...




```
1 ptrLast = ptrLast->ptrPrev;
```

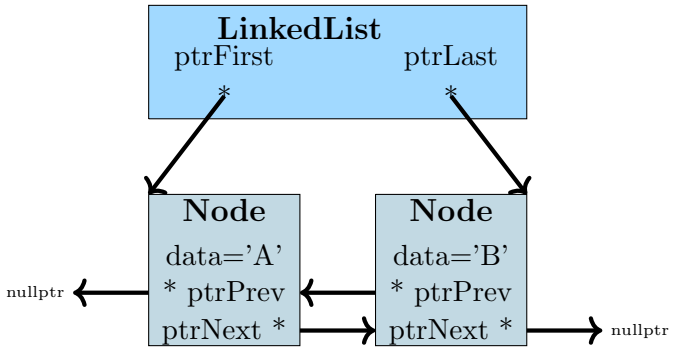
**Step 2: Delete the old last Node** You can do this by accessing the old-last with `ptrLast->ptrNext`.




```
1 delete ptrLast->ptrNext;
```

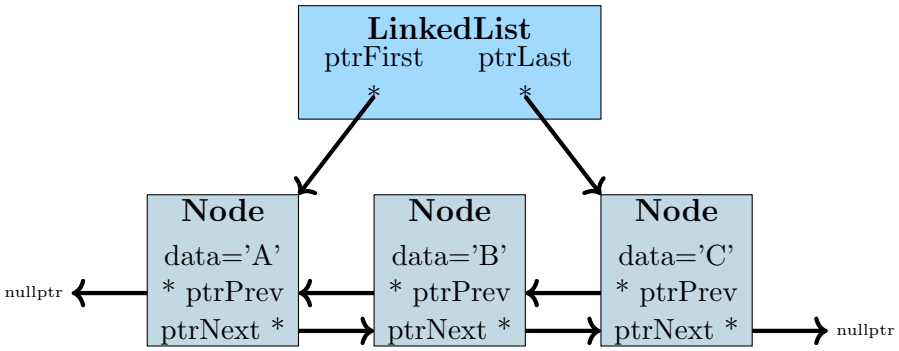
---

**Step 3: Update `ptrLast`'s `ptrNext` to point to nullptr.** And make sure to decrement the `itemCount`!




```
1 ptrLast->ptrNext = nullptr;
2 itemCount--;
```

## Accessing data




### GetFront

```
1 T& GetFront()
2 {
3     // Add error checking
4     return ptrFirst->data;
5 }
```

This Get function is pretty straightforward; just get the `data` stored within the first Node via `ptrFirst`...

---

### GetBack

```
1 T& GetBack()
2 {
3     // Add error checking
4     return ptrLast->data;
5 }
```

Similar to GetFront, except we're getting the data at the end.

> **Empty lists!**
> If the user tries to Get an item from an empty list, the function should throw an exception.

### GetAtIndex

```
1 T& GetAtIndex( const int index );
```

...or maybe...

```
1 T& operator[]( const int index );
```

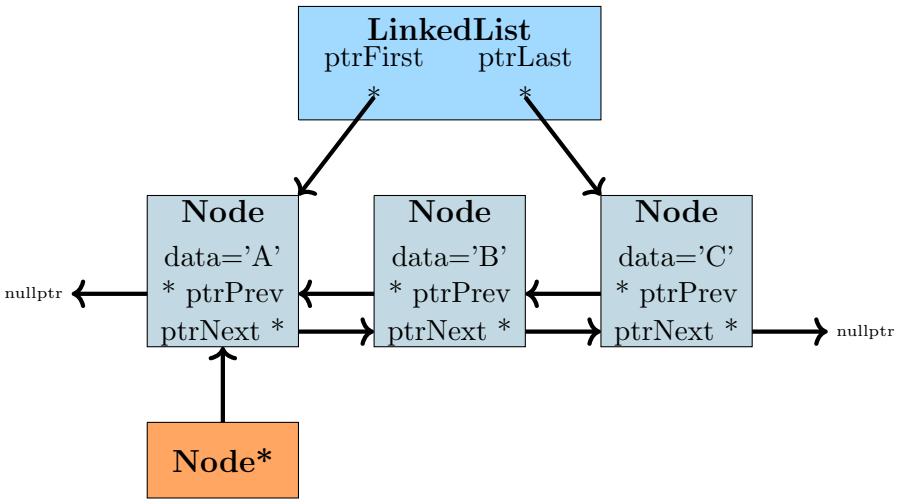
When you want to access an element at a specific position in the List, and that position isn't the `ptrFirst` or `ptrLast`, then you have to **walk the linked list**.

Linked lists don't have the ability to randomly access, say, "item at index 2". Instead, we work like this:

1. Set index counter to 0.
2. Is this Node pointer pointing to `nullptr`?
   Return that we can't find the item
3. Are we at the index the user wants?
   Return this Node's data
4. Else...
   Go to the next Node and increment the counter.

The steps to implement it are illustrated next.

**Step 1: Creating a pointer to walk the list**




First, we will create a Node pointer. This Node pointer is used to **traverse** the list, and we are **not using this Node to create a new Node**!

We're going to start off this Node pointer pointing to the first Node in the list.

```
1 Node* ptrCurrent = ptrFirst;
```

---

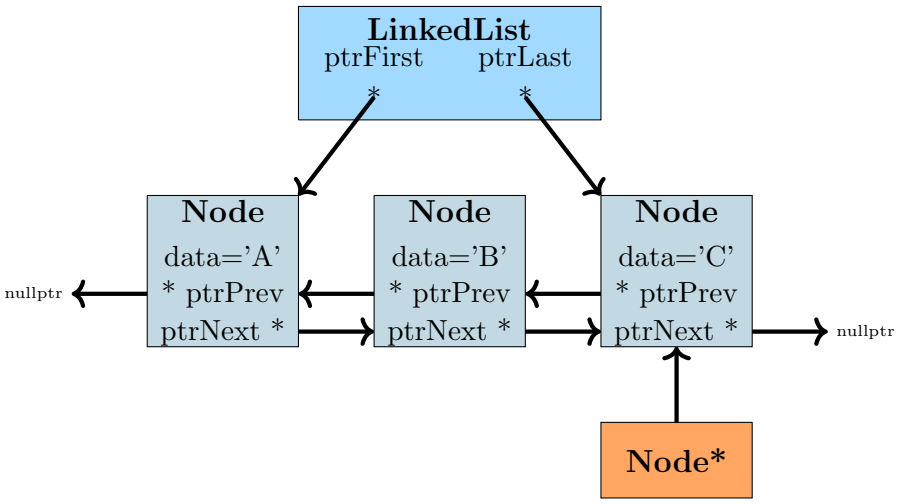
**Step 2: Writing a loop** Using a for loop or a while loop, we can write code to move $n$ positions until we get to the `index` passed in as a parameter.

While we aren't at the destination Node, we step forward by using `ptrCurrent = ptrCurrent->ptrNext;`

```
1 for ( int i = 0; i < index; i++ )
2 {
3     ptrCurrent = ptrCurrent->ptrNext;
4 }
```

**Warning!**
It's possible that the user may have entered an invalid index. You will need to do error checking to either make sure that the index is valid, or to make sure that `ptrCurrent` is never `nullptr` when `ptrCurrent->ptrNext` is called.

### Step 3: Return the data




By the time our loop is finished, we should be at the correct index. At this point, `ptrCurrent` is pointing to the correct Node and you can return that Node's `data` directly through `ptrCurrent`.

```
1 return ptrCurrent->data;
```

### Helper functions

#### Clear

```
1 void Clear();
```

The Clear function will be responsible to freeing the memory of all the nodes. You can implement this function by writing a while loop that continues looping while there are still items in the list, calling a Pop function until it is finally empty.

```
1 while ( !IsEmpty() )
2 {
3     PopFront();
4 }
```

---

#### IsEmpty

```
1 bool IsEmpty();
```

This should simply return `true` if `itemCount` is 0. Otherwise, return `false`.

---

#### Size

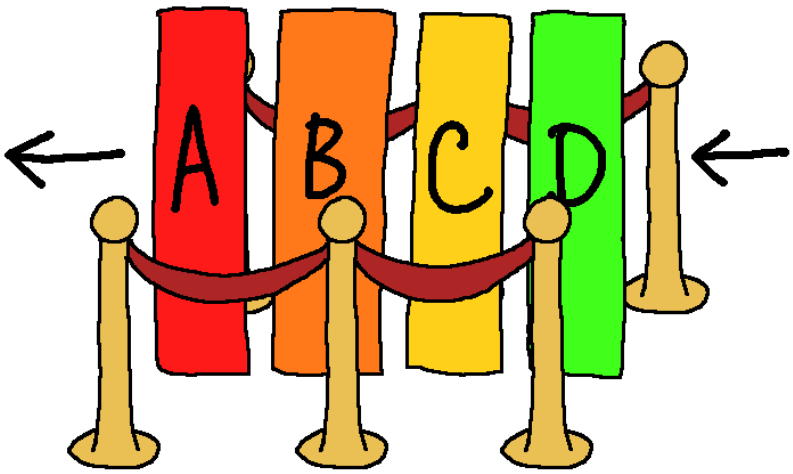
```
1 int Size();
```

Return the value of `itemCount`.

## 2.5 Types of Linked Lists

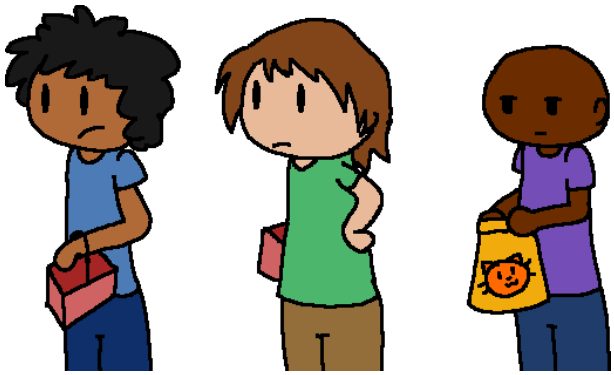
# Topic 3

# Queues




## 3.1 What are Queues?

Queues are a type of structure that only allows new items to be added to the *end* of the queue, and only allows items to be removed at the *beginning* of the queue. It is commonly called **FIFO: First In First Out**. A queue is basically any line you have to stand in at a store.



Remember that a **data structure** will have functionality to **add**, **remove**, and **access** (and sometimes **search**) the structure, but some data structures are for special types of applications. A queue has add, remove, and access, but these are all restricted - you can't *insert* in the middle or beginning of the queue, or *remove* from the middle or end of the queue.

## Building a Queue

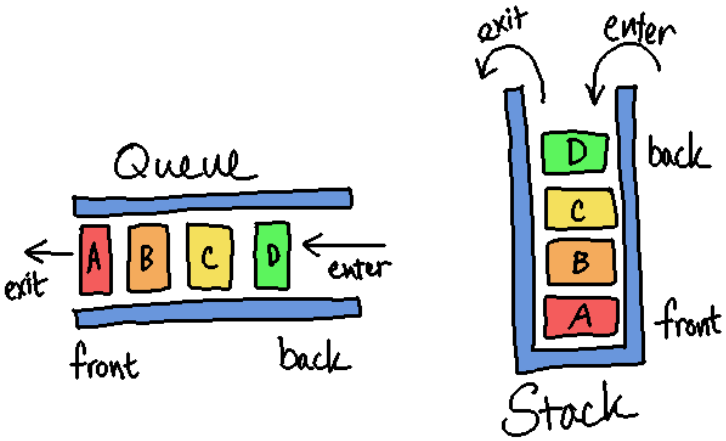
A Queue, in and of itself, is just an idea. We can use our **dynamic array** structure or our **linked list** structure to build a Queue, and only allowing certain functions to be called and others to be unavailable. We could simply do this by writing a Queue class that inherits from or contains a `LinkedList` or `Vector`, and only providing access to certain methods.

Since a Queue only has one place items can be added, we would only have a simple `Push` function instead of having a `PushFront` and `PushBack`, and same for the rest of the functions.

| Vector | List | Queue | Stack |
|---|---|---|---|
| PushFront | PushFront | - | - |
| PushBack | PushBack | Push | Push |
| InsertAt | -* | - | - |
| | | | |
| PopFront | PopFront | Pop | - |
| PopBack | PopBack | - | Pop |
| RemoveAt | -* | - | - |
| | | | |
| GetFront | GetFront | Peek | - |
| GetBack | GetBack | - | Top |
| GetAt | -* | - | - |

* You could implement an insert/remove/get to a list, but in the STL List object it is not available.

We will cover Stacks in another chapter, but it is a cousin of Queues - they also have restricted access, but they're **LIFO** (Last In First Out).

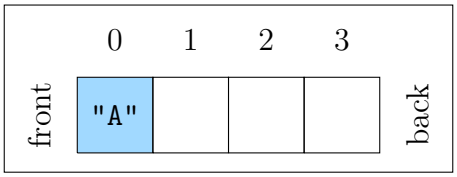


## Adding to a Queue (Push/Enqueue)

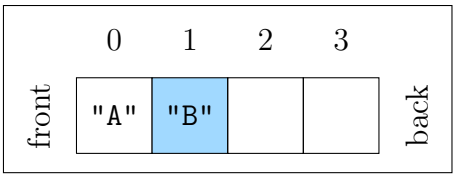
The Queue will add items using `PushBack` functionality, adding each item to the *end* of the list. Let's say we're running the following commands in order:

```
Push( "A" );    Push( "B" );    Push( "C" );    Push( "D" );
```

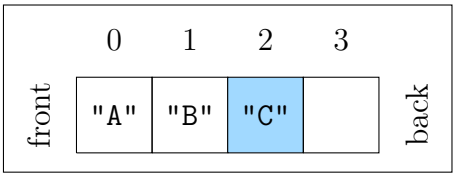
1. `Push("A")`:



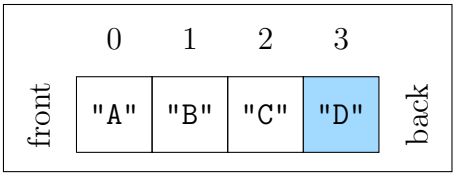

2. `Push("B")`:




3. `Push("C")`:




4. `Push("D")`:

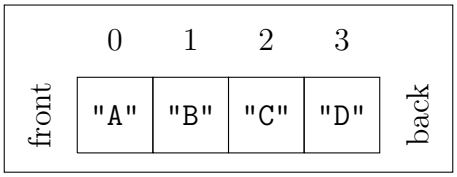


## Removing from a Queue (Pop/Dequeue)

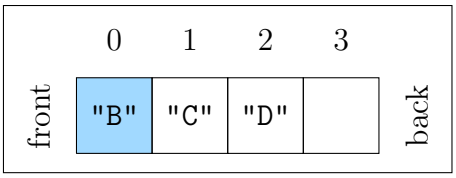
When we remove items from the Queue, the front item will be removed, and each item will be shifted one space toward the front.

`Pop();` `Pop();` `Pop();`

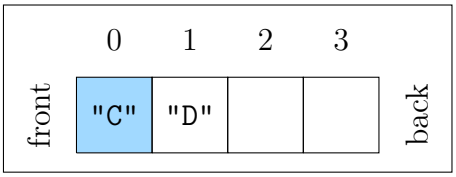
1. `Initial state:`



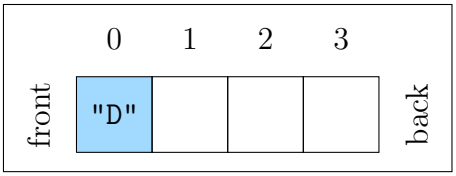

2. `Pop():`




3. `Pop():`



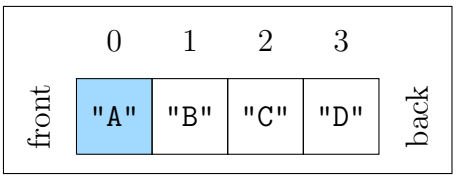

4. `Pop:`



## Access with a Queue (Get/Peek)

When we call the function to access from a Queue, we can only access one item: the front-most item.

```
Get();
```



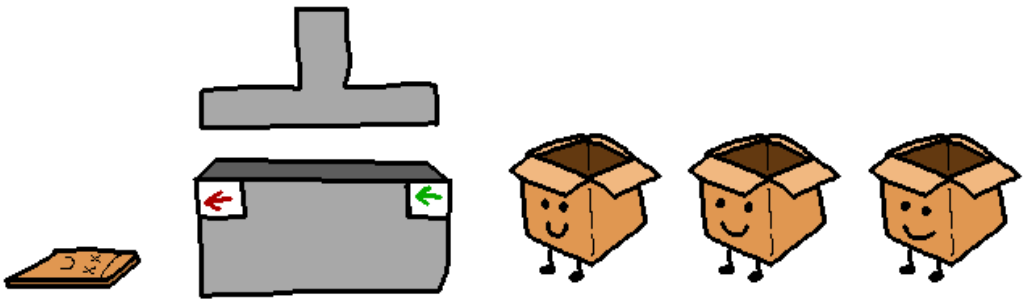

---

## Use of a Queue in software

With this single-minded, "access the front item of the Queue, then discard that and access the next front item", what is a Queue structure used for?

Similarly to in real life, it's a way to make items wait their turn for some form of processing. At the grocery store, there are limited resources (cash registers). At each register, people queue up, waiting for their time until they can go through the process of having their items rung up and paying for them. And, we wait in a "first come, first served" order (people will generally get angry if you try to enter at the *front* of the line).

The same is true of a Queue. Perhaps we have many different things we want to process, but a limited amount of resources. So, as these items come in, if we're busy processing something, everything else *queues up* and waits its turn.

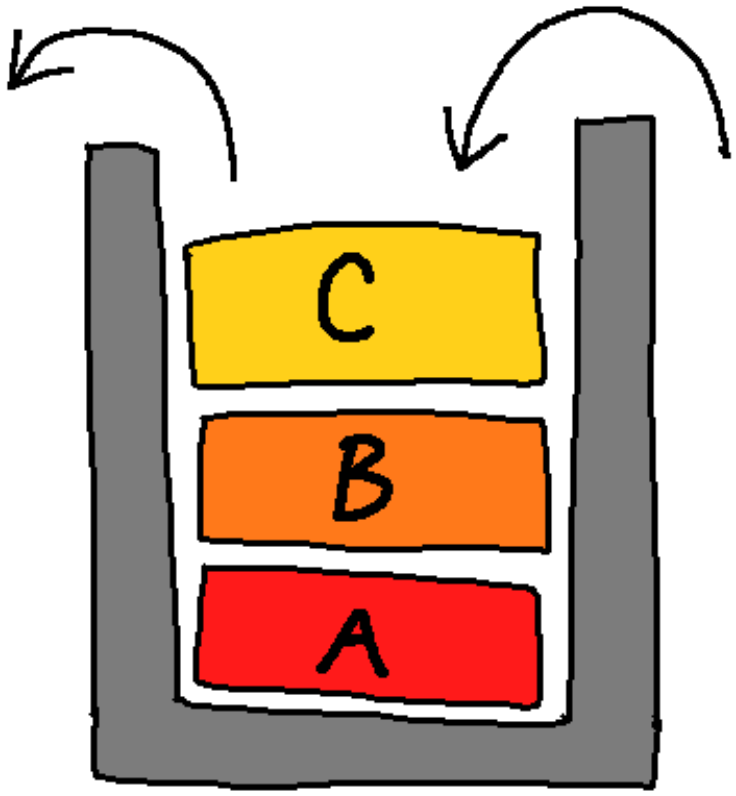


## 3.2 Implementing a Queue

A Queue will be simplest to implement if you build the class on top of an existing structure.

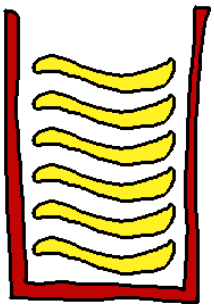
# Topic 4

# Stacks



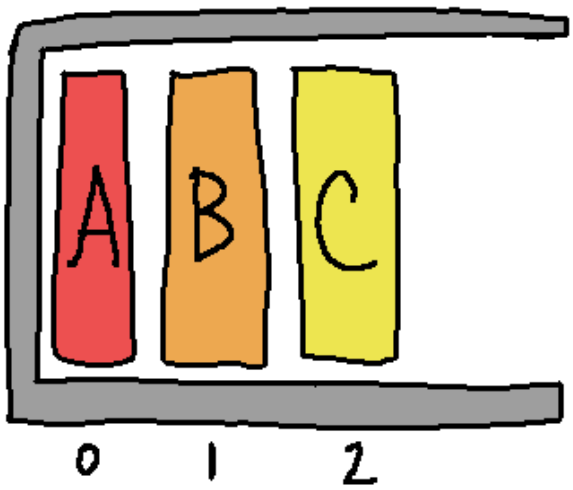

## 4.1 What are Stacks?

Stacks are another type of restricted-access data structure. In a way, it's a cousin to a Queue, since they both only allow access to certain items stored within and are used for specialized operations.



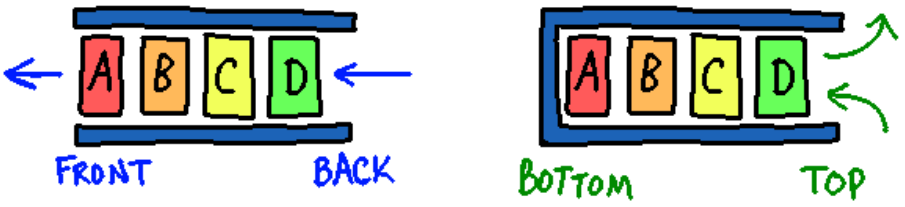
The Stack, however, is **LIFO: Last In First Out**. The stack can be visualized as a can of Pringles chips, where you only have access to whatever is on the top of the internal stack of chips.

Of course, we don't usually think of **arrays** or **linked lists** in a vertical manner, so it might also help to see the stack structure like this:



While similar to a Queue, which allows **add** data to the back and **accessing/removing** data from the front, a Stack only allows **adding, accessing, and removing** data all from the "top".

**Queue** **Stack**




Just like with a Queue, we can implement a Stack on top of an array or linked structure, taking advantage of code we've already previously written to create our new data structure, the Stack:

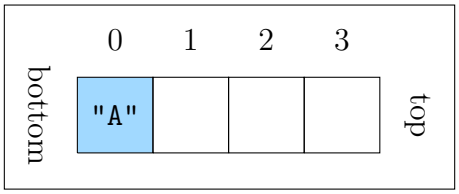
| Vector | List | Queue | Stack |
|---|---|---|---|
| PushFront | PushFront | - | - |
| PushBack | PushBack | Push | Push |
| InsertAt | -* | - | - |
| | | | |
| PopFront | PopFront | Pop | - |
| PopBack | PopBack | - | Pop |
| RemoveAt | -* | - | - |
| | | | |
| GetFront | GetFront | Peek | - |
| GetBack | GetBack | - | Top |
| GetAt | -* | - | - |

## Adding to a Stack (Push)

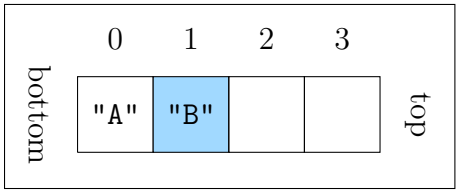
The Stack will add items using `PushBack` functionality, adding each item to the *top* of the list. Let's say we're running the following commands in order:

```
Push( "A" );    Push( "B" );    Push( "C" );    Push( "D" );
```

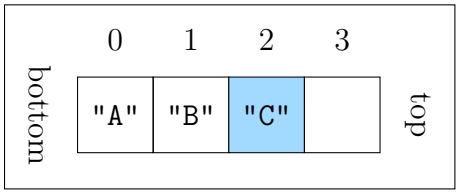
1. `Push("A")`:



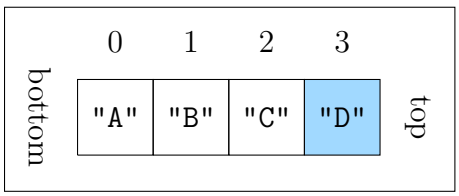

2. `Push("B")`:




3. `Push("C")`:




4. `Push("D")`:

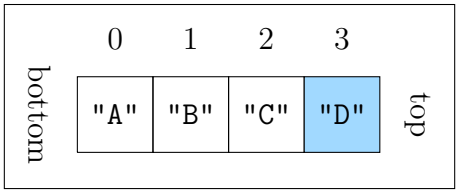


## Removing from a Stack (Pop)

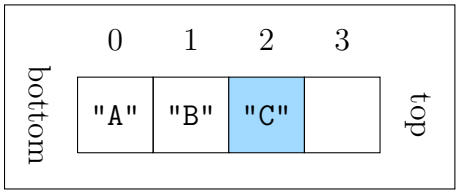
When we remove items from the Stack, the top item will be removed.

```
Pop();    Pop();    Pop();
```

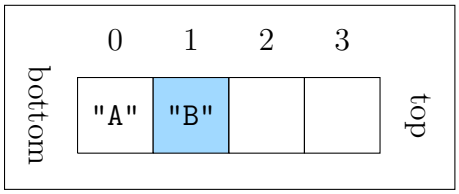
1. `Initial state:`



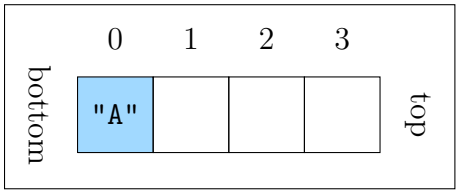

2. `Pop():`




3. `Pop():`

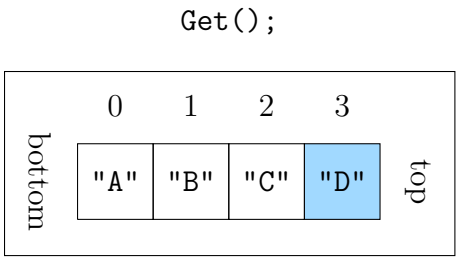



4. `Pop:`



## Access with a Stack (Get/Peek)

When we call the function to access from a Stack, we can only access one item: the top-most item.



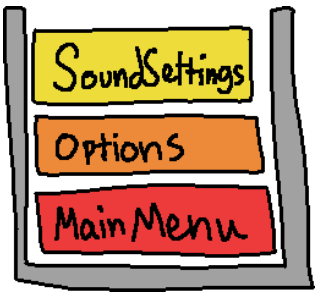

---

## Use of a Stack in software

Stacks are a handy instruction in computer science, allowing us to essentially "pause" where we're at with something when we **push** a new task to the top of the stack, and simply **pop** that task off and return directly to the previous item. But how, in particular, are they applied?

## Screen navigation

In a program that generally only displays one screen at a time - such as a video game, or a small computer device like a GPS - we can **Push** new view states onto a "view stack" any time we navigate to a new screen.

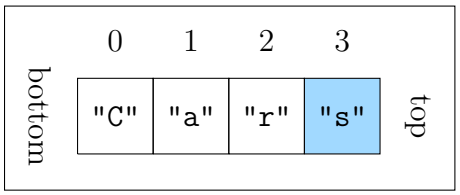



Maybe the first screen of the program is the **main menu** (`Push(mainMenu);`). When the user clicks the "options" button, they go to the **options menu** (`Push(options);`). Then, they may click the "sound settings" tab, taking them to the **sound settings menu** (`Push(soundSettings);`).

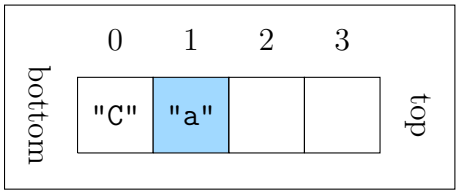
Once they're done adjusting the sound, they click the back button. Instead of having to write logic like "what comes before sound settings?" we simply **Pop()** the current view off the stack, which returns us to the **options menu**.

## Undo functionality

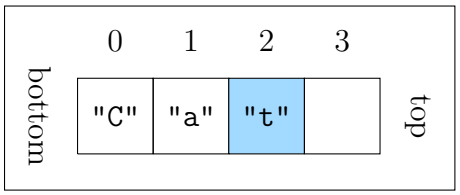
Let's say you're typing some text, and behind-the-scenes, the computer is storing all the text you've typed in a buffer like this...

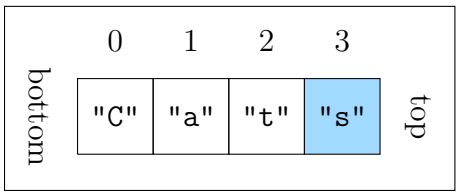



But oops, you meant to type "Cats", not "Cars". The **backspace** button is essentially a **Pop()**, and each time you hit backspace, the "top-most" (i.e., most recent) keystroke is removed from the buffer. We hit backspace twice (`Pop(); Pop();`) and we have gone backwards twice:




Now as we type in our correction, each new keystroke is a **Push()** onto the buffer stack. We fix our error (`Push("t"); Push("s");`)...

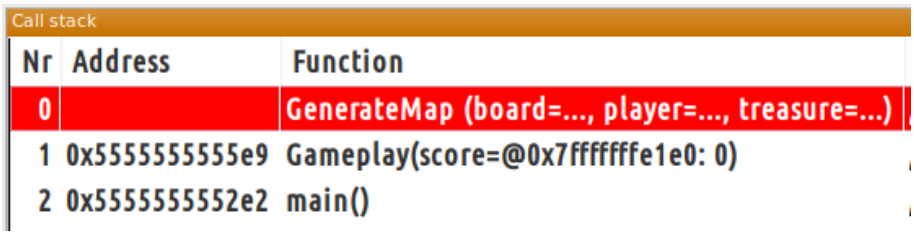


### Call stack

A stack is utilized any time we make a **function call**.

When debugging, you should notice that there is a **Call Stack** pane, which lists all the functions that have been called, leading up to where the program is currently paused (when using breakpoints).



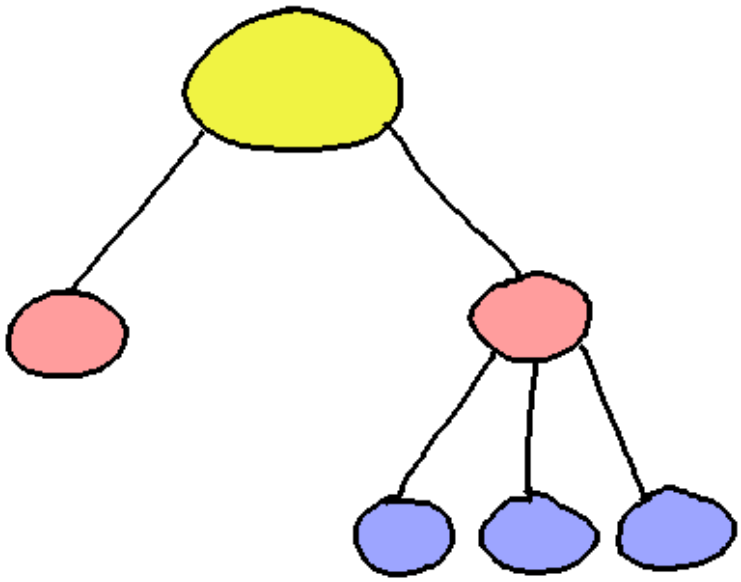

The top-most item in the call stack is the most recently called function. The bottom-most is the first function called (such as `main()`.)

When a function is called, it is **Pushed** onto the call stack. When the function ends, it is **Popped** off of the call stack, returning us to whatever previous function was active before it.

## 4.2 Implementing a Stack

# Topic 5

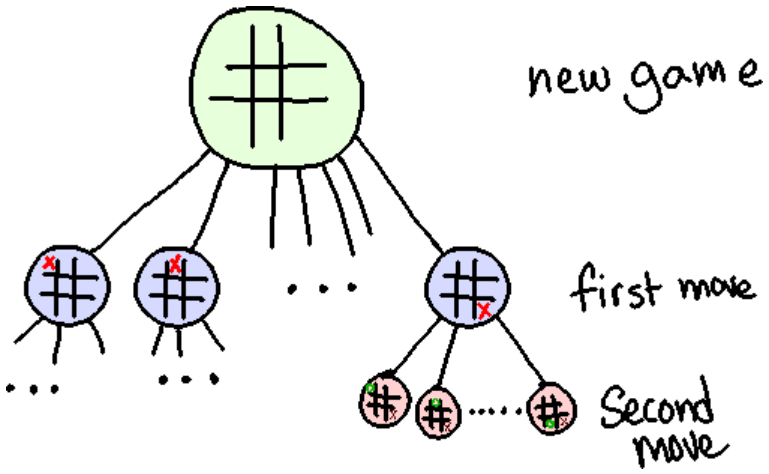
# Trees



## 5.1 Linear structures vs. tree structure

Lists, Queues, and Stacks are all examples of linear structures. Now we

will work with trees, which are hierarchical.



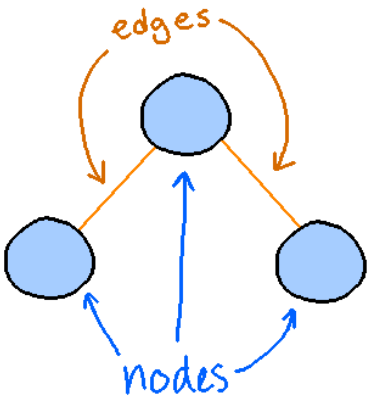

A tree structure could be used to model a filesystem, moves or progress in a game, a family tree. And with a binary search tree, you could store any kind of data and keep it organized as each new item is inserted.

## 5.2 Basic terminology

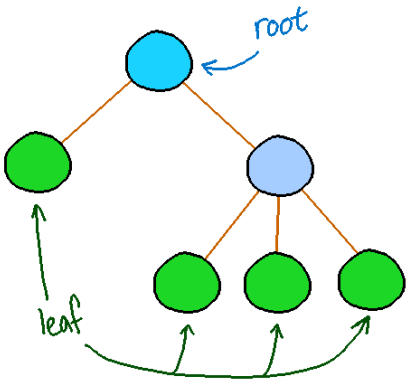
A tree is a type of graph that is completely connected and has no cycles. A tree is made up of **nodes/vertices** and **edges**.

**Node:** A **node** is a point in a tree; in the code, it will be a structure that contains data.

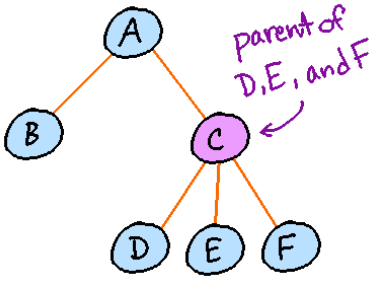
**Edge:** An **edge** is a line that connects two nodes; in the code, it will be the pointer from one node to another.



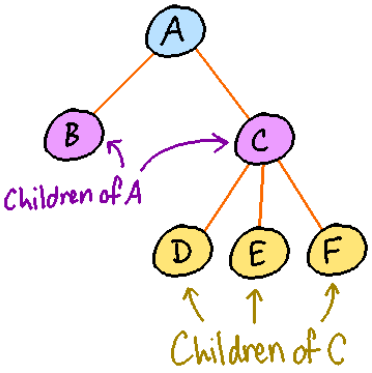
**Root:** Each tree will have exactly **one root**, which all other nodes branch out from. When drawing a tree, we will generally make it the topmost node.
The root is the only node that has **no parent**, and every other node has **exactly one parent**.



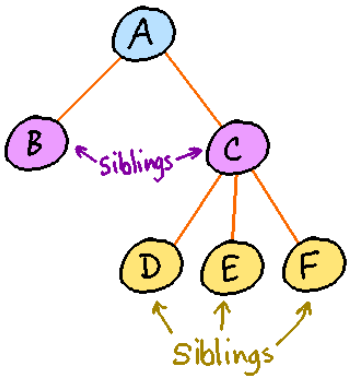

**Leaf:** A leaf is a node with no children.



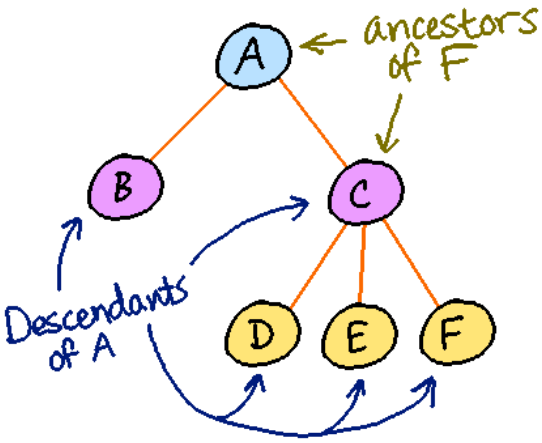

**Parent:** In a tree, each node has **exactly one parent**, except for the root node, which has no parent. The parent of node $n$ is the node directly "above" $n$, on the path back towards the root.




**Child:** A node in a tree can have any amount of children. A child of node $n$ are the nodes directly under $n$, branching off from it.

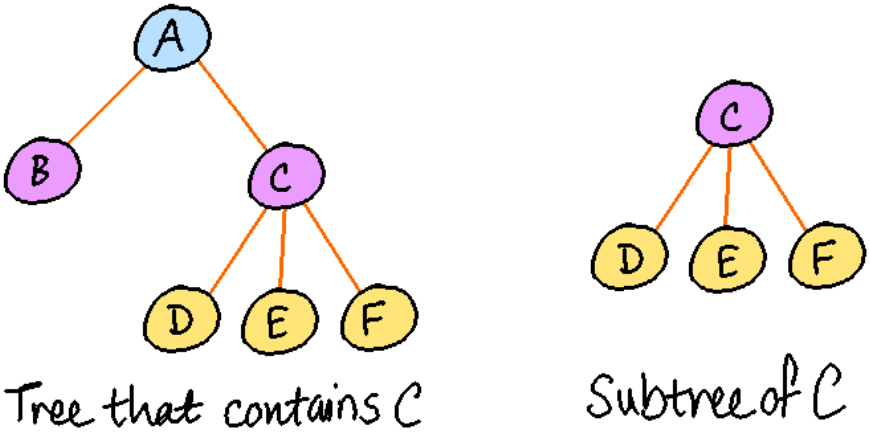



**Sibling:** Nodes that are children of the same parent node are siblings of each other.

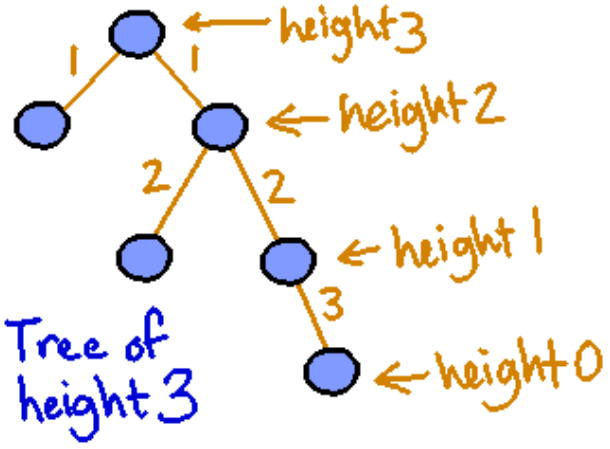



**Ancestors:** The **ancestor of node** $n$ is any node between $n$ and the root (including the root).

**Descendants:** The **descendants of node** $n$ is any node derived from node $n$, going all the way down to the leaves.

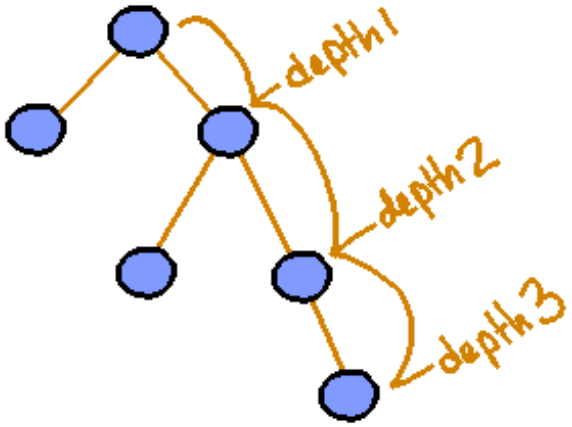



**Subtree:** The **subtree of node** $n$ is a tree with node $n$ as the root, derived from an original tree that contains $n$. Each node in a tree is essentially the root of its own subtree.



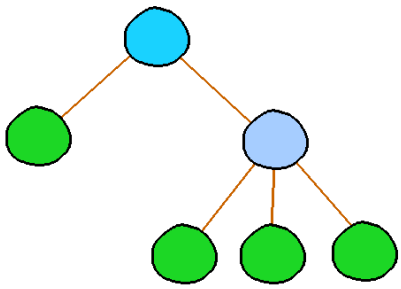

**Height of a node $n$:** The height of some node $n$ is the amount of edges on the **longest path** from that node $n$ to any descendant leaf.

**Height of a tree:** The height of a tree is the number of edges on the **longest path** from the root to a leaf.




**Depth of a node $n$:** The length of the path (amount of edges) from that node $n$ to the root of the tree.

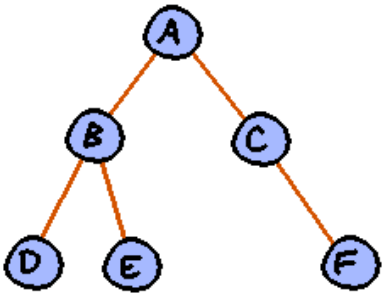
## 5.3 Types of trees

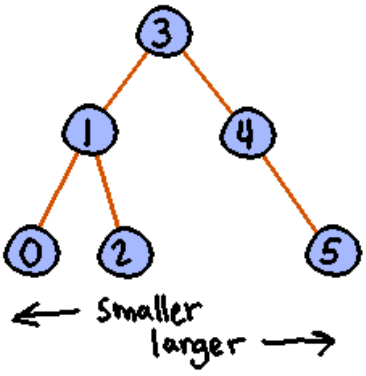


**General tree:** A general tree contains a root and subtrees that branch off from that root. There are no restrictions on the way it must be filled, or how many children each node can have.

**$n$-ary tree:** An $n$-ary tree is a type of tree where each node can have no more than $n$ children.

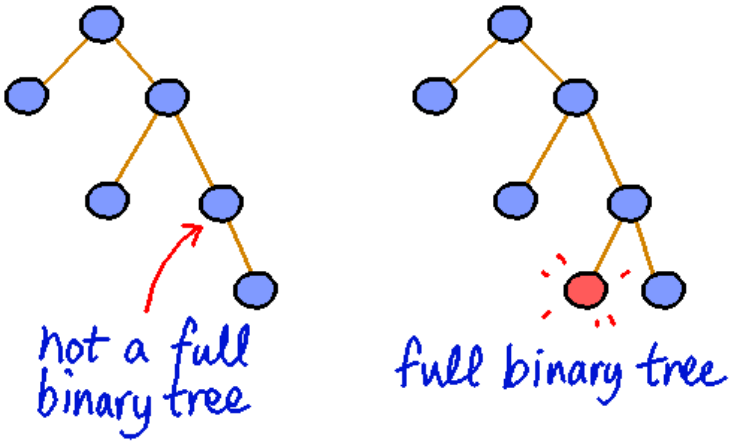
**Binary tree:** A tree where each node can have no more than 2 children. This means a node can have 0, 1, or 2 children.



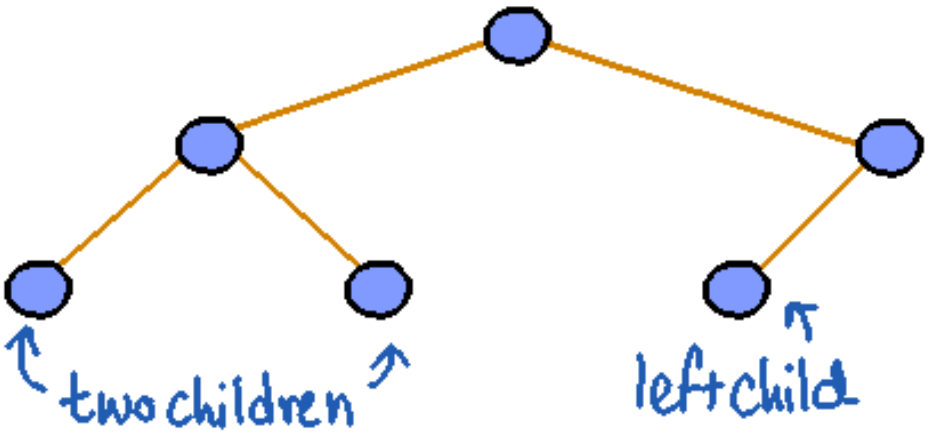




**Binary search tree:** A binary seach tree is a binary tree that is sorted. For any given node, its **left child** has a lesser value, and its **right child** has a greater value.

## Tree fullness, completeness, and balance

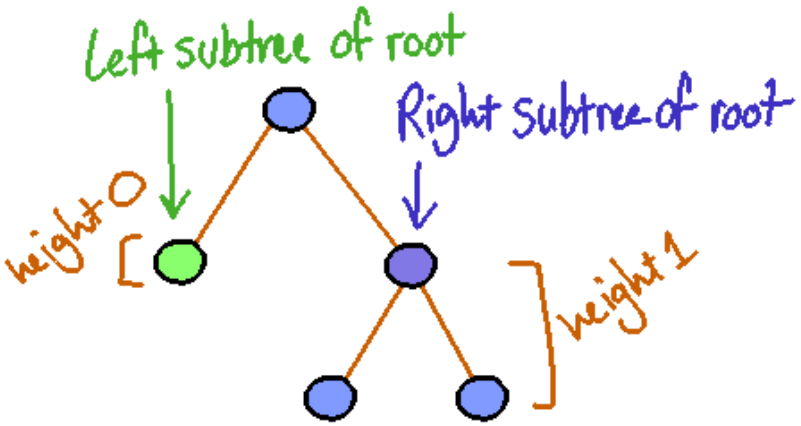



**Full binary tree:** A full binary tree is one where *every node* either has 0 children or 2 children.




**Complete binary tree:** A complete binary tree is one where all nodes have 2 children, except possibly in the lowest level. Additionally:

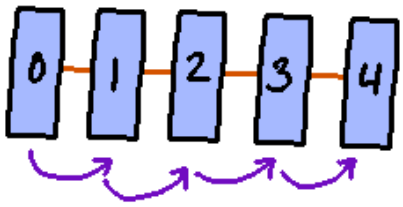
- In the 2nd-to-lowest level, if a node has children, then all its siblings to its left must have 2 children each.
- If a node in the 2nd-to-lowest level has only one node, it must be a left child.
- This mean the tree fills from left-to-right.

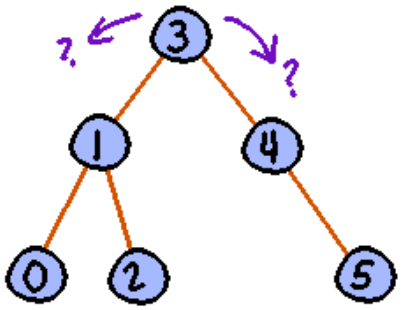



**Balanced binary tree:** A binary tree is height balanced when, for each node of the tree, each node's **left subtree** and **right subtree** differ in height by no more than 1 level.

---

## 5.4 Binary tree traversals



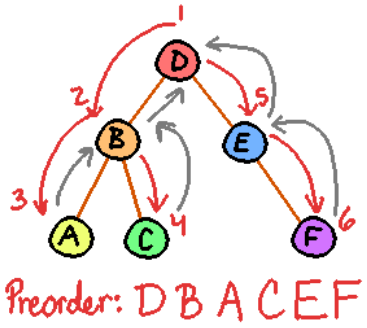




When working with a list, it's easy to just output all the contents from beginning-to-end and call it a day - simple! But how do you output the contents of a binary tree?

## Preorder traversal

We begin at the **root node**. The algorithm here is:

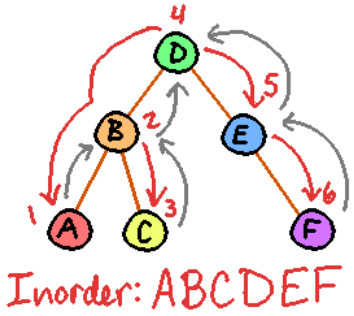
1. Display current node's value
2. Traverse to the left subtree
3. Traverse to the right subtree




## Inorder traversal

We begin at the **root node**. The algorithm here is:

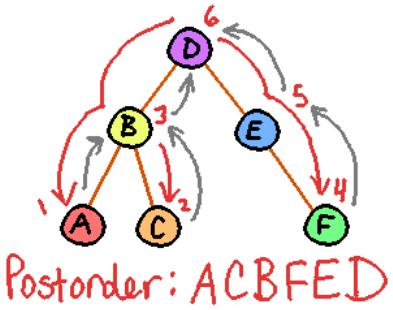
1. Traverse to the left subtree
2. Display current node's value
3. Traverse to the right subtree




## Postorder traversal

We begin at the **root node**. The algorithm here is:

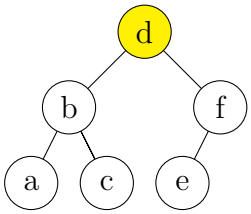
1. Traverse to the left subtree
2. Traverse to the right subtree
3. Display current node's value



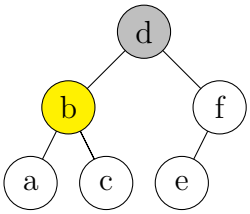
## Step-by-step example: Preorder traversal

**Algorithm:** recurse left, display self, recurse right

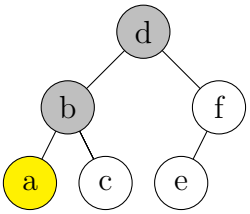
1. Begin at root
Display self
Output: "d"



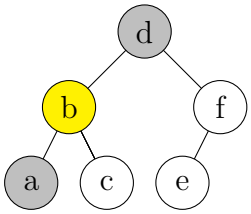

2. Recurse left to b
Display self
Output: "db"



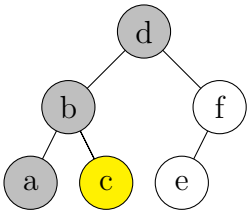

3. Recurse left to a
Display self
Output: "dba"



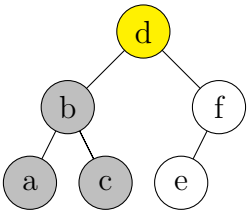

4. Can't recurse left
Return to b



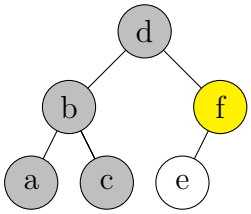

5. Recurse right to c
Display self
Output: "dbac"



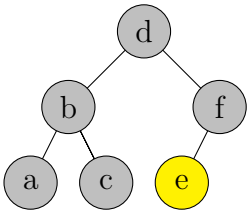

6. Can't recurse
Return to b
Done recursing
Return to d



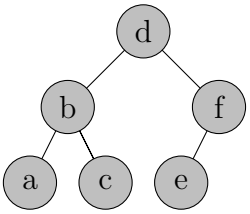

7. Recurse right to f
Display self
Output: "dbacf"




8. Recurse left to e
Display self
Output: "dbacfe"




9. Can't recurse
Return to f
Done recursing
Return to d
Done recursing
Algorithm done

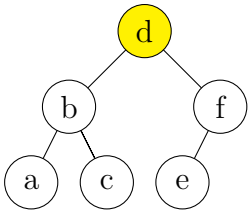



**Output:**
**"dbacfe"**

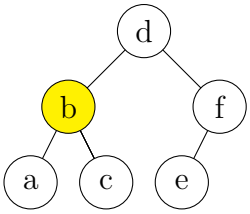
## Step-by-step example: Inorder traversal

**Algorithm:** display self, recurse left, recurse right

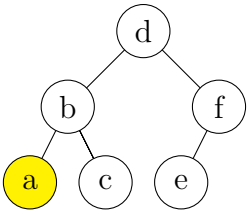
1. Begin at root
Recurse left to b



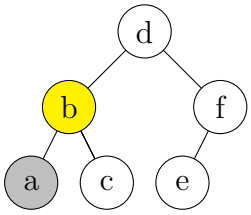

2. Recurse left to a



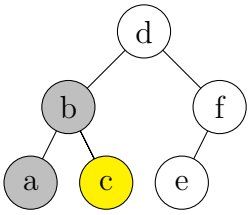

3. Can't recurse left
Display self
Output: "a"
Can't recurse right
Return to b



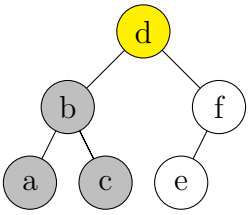

4. Display self
Output: "ab"
Recurse right to c



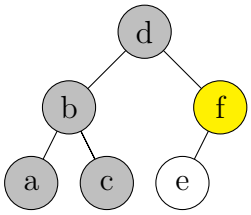

5. Can't recurse left
Display self
Output: "abc"
Return to b



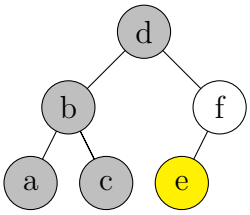

6. Done recursing
Return to d
Display self
Output: "abcd"
Recurse right to f



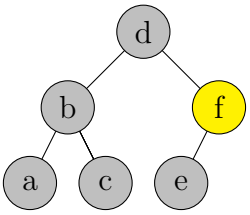

7. Recurse left to e




8. Can't recurse left
Display self
Output: "abcde"
Return to f




9. Can't recurse right
Display self
Ouptut: "abcdef"
Return to d
Done with algorithm

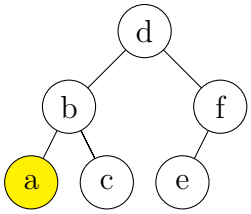



**Output:**
**"abcdef"**

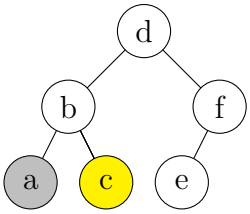
## Step-by-step example: Postorder traversal

**Algorithm:** recurse left, recurse right, display self

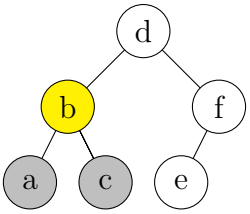
1. Begin at root
Recurse left to b
Recurse left to a



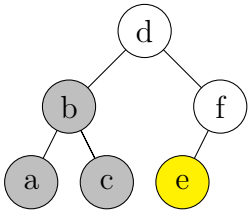

2. Can't recurse
Display self
Output: "a"
Return to b
Recurse right to c



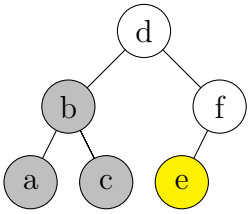

3. Can't recurse
Display self
Output: "ac"
Return to b



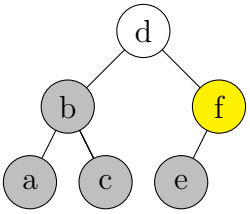

4. Done recursing
Display self
Output: "acb"
Return to d
Recurse right to f
Recurse left to e



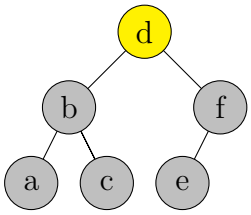

5. Can't recurse
Display self
Output: "acbe"
Return to f




6. Done recursing
Display self
Output: "acbef"
Return to d




7. Done recursing
Display self
Output: "acbefd"
Done with algorithm




**Output:**
**"acbefd"**

# Topic 6

# Binary Search Trees

asdf