Introduction

What is DSA (Data Structures and *Algorithms*)?

Any data representation and its associated operators.

It is a combination of **two** topics of *Computer Science* that deal with **How to organize data** and **manipulate data** efficiently.

For Example

A *sorted list of integers* and *arrays*.

There are also *Types of Data Structures*.

Also, knowing how *Memory Allocation* works is really important as well.

Algorithms

It's a **well defined computational procedure** that takes some value, or a set of values as *input* and *produces* some value or a set of value *in a unit of time*.

We have to make sure that the *Algorithms* we make are *Efficient*.
Their *Efficiency* matters a lot!

Speaking of Algorithms... Let's talk about few Algorithms for *Sorting Arrays*.

Efficiency

Many algorithms have the *same output* but the *way it processes the input differs leading to time difference / difference of efficiency*.

> **Note**
>
> This kind of difference can be much more significant that the differences due to hardware and software.

There are *two types of complexities*...

1) *Time Complexity*.
2) *Space Complexity*.

Time Complexity

The time complexity of an algorithm quantify *the amount of time taken* by an algorithm to run **as a function of the length of the input**.

> **Note**
>
> The time to run is a function of length of the input and not the actual execution time of the machine on which the algorithm is running.

Example:

Let's take the following code as an example:

```cpp
int getFirstelement(int arr[]) {
    return arr[0];
}
```

Looking at this example... this function takes just one line of code... lets take it as *O*...
So, we can say *O(1)*... *1* could be any unit like *ms* or something... but for now let's just say *O(1)*...

> **Note**
>
> *O* is called **Big-O** and it's a *constant*.

Now, if we take a function which includes a for that'll run *n-times*...

```C++
for(int i = 0 ; i < n ; i++) {
    // Some task...
    // .
    // .
    // .
}
```

For this case, we can say *O(n)*. Making sense?

Okay now... if a function has *2 for loops*... then what?

If a function has two for loops that are nested and each loop runs `n` times, then the time complexity of the function is *O(n^2)*.

Here's why: For each iteration of the outer loop, the inner loop runs `n` times. Since the outer loop also runs `n` times, the total number of iterations is `n * n`, which is *n^2*. Therefore, we say that the function has a quadratic time complexity, which is denoted as *O(n^2)*.

This is under the assumption that the work being done inside the inner loop is constant time, i.e., it does not depend on the size of the input. If the work inside the loop also depends on `n`, then the time complexity could be higher.

Here's a simple example in Python:

C++

```cpp
#include <iostream>

void function(int n) {
    for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            // constant time work here
        }
    }
}
```

In this code, `function` has a time complexity of *O(n^2)*.

Space Complexity

Problem-Solving using computer requires memory to hold *temporary data* or f*inal result* while the program is in execution...

The amount of space required by the algorithm to solve given problem is called space complexity.

Sorting Arrays

We can sort arrays in MANY ways but as we talked about *Algorithms* before that their *Time Complexity* matters a lot.

What we do is try out different ways of sorting an array and see which method suits the best for a specific task...

First one is *Binary Search*.

Binary Search

The binary search algorithm is **a divide and conquer algorithm that you can use to search for and find elements in a sorted array**.
The algorithm is fast in searching for elements because it removes half of the array every time the search iteration happens.

Following are the few steps to do a Binary search:

1) First, as mentioned above, *sort the array*.
2) Then find the *highest* and the *lowest* value.
3) Then find the *middle point* of the array.

Note </>

To Find the middle point of an array, we can use the formula:

$$lowest + \frac{highest - lowest}{2}$$

Let's take a look at an example...

Example:

if we have the following array:

And let's say, we have to find *20*.

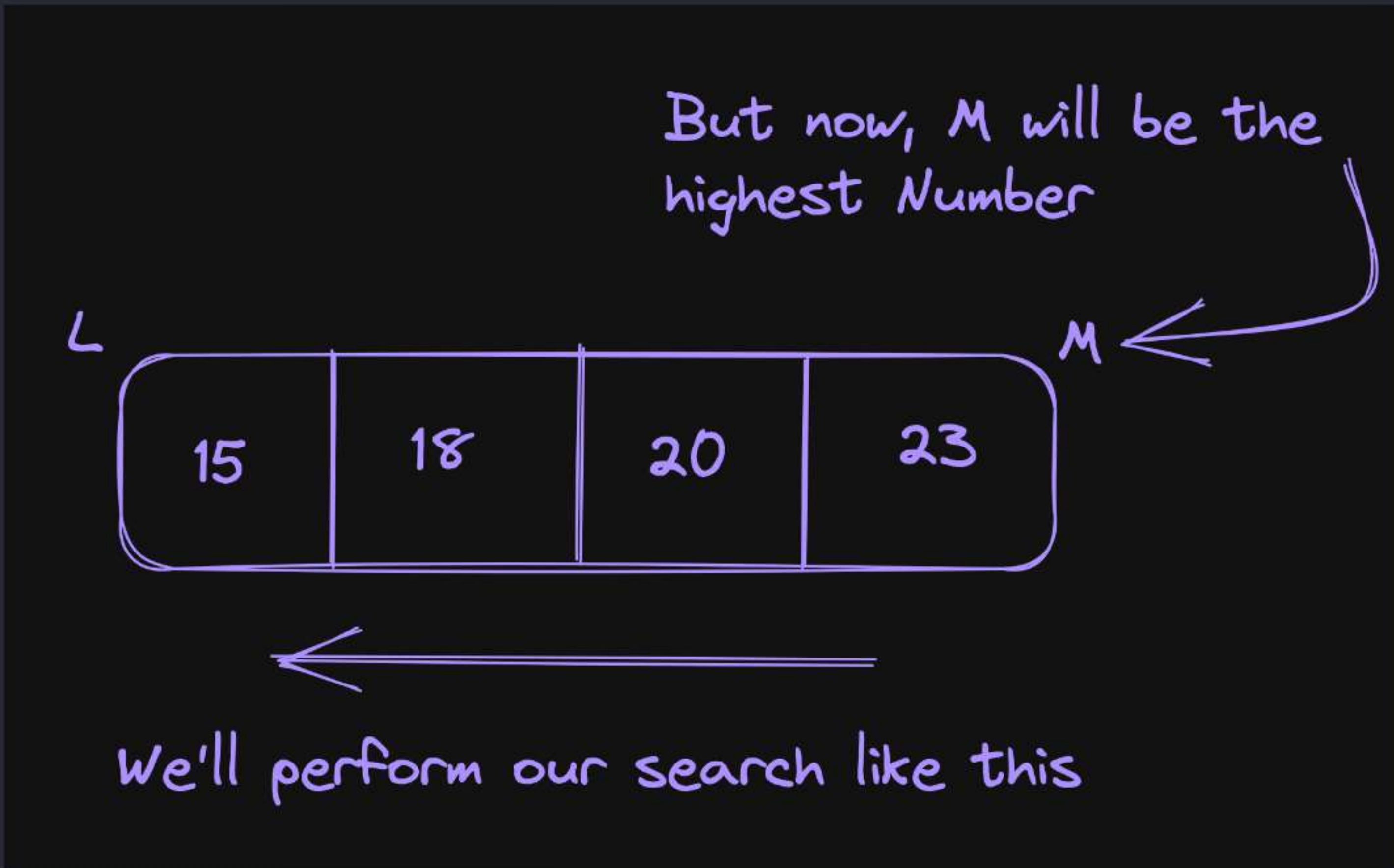
Now after finding the *mid point*, see if the number we are searching for **is the mid number**, if not, **is it less than the number**? If yes, then look at the left side. If not, then look at the right side.

Now, taking the above array as an example, we have to find the number from **L - M**.

Now after finding the *mid point*, see if the number we are searching for **is the mid number**, if not, **is it less than the number**? If yes, then look at the left side. If not, then look at the right side.

Now, taking the above array as an example, we have to find the number from **L - M**.

We're search like this:



Now again find the Mid point and follow the steps again till the number is found.

Types of Data Structures

There are *two* types of Data Structures.

- *Linear Data Structure*
- *Non-Linear Data Structure*

Here's a graphical representation for better understanding...

Linear Data Structure

In *Linear Data Structure*, the data is arranged in a *linear form*...

For Example

-	1
-	2
.	.
.	.
-	n

In the above figure, the data is arranged linearly from top to bottom.

Same goes for horizontal.

There are also some lists that points to another list... We'll look into that further later on.

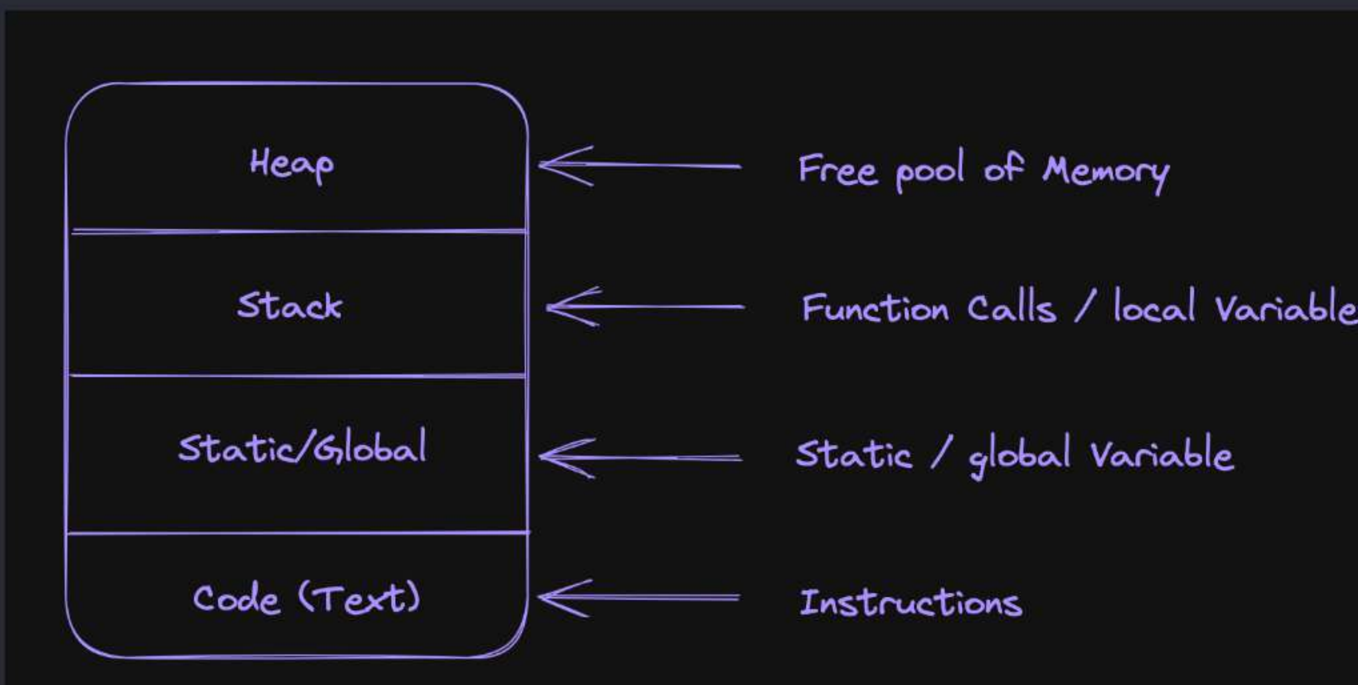
Non-Linear Data Structure

Non-linear Data Structures include structures which are arranged in a random order like **Trees**, **Graphs**, and **Tables**.

Memory Allocation

Memory Allocation refers to how *Memory* is assigned to a *Program / Software / Application*. It has **Four** parts.

1) *Code (Text)*
2) *Static OR Global*
3) *DSA/Lecture 2/Stack*
4) *Heap*



Code (Text)

In this part of the memory, simply put, the code you write on a file is stored.
It contains all the *instructions* for your program.

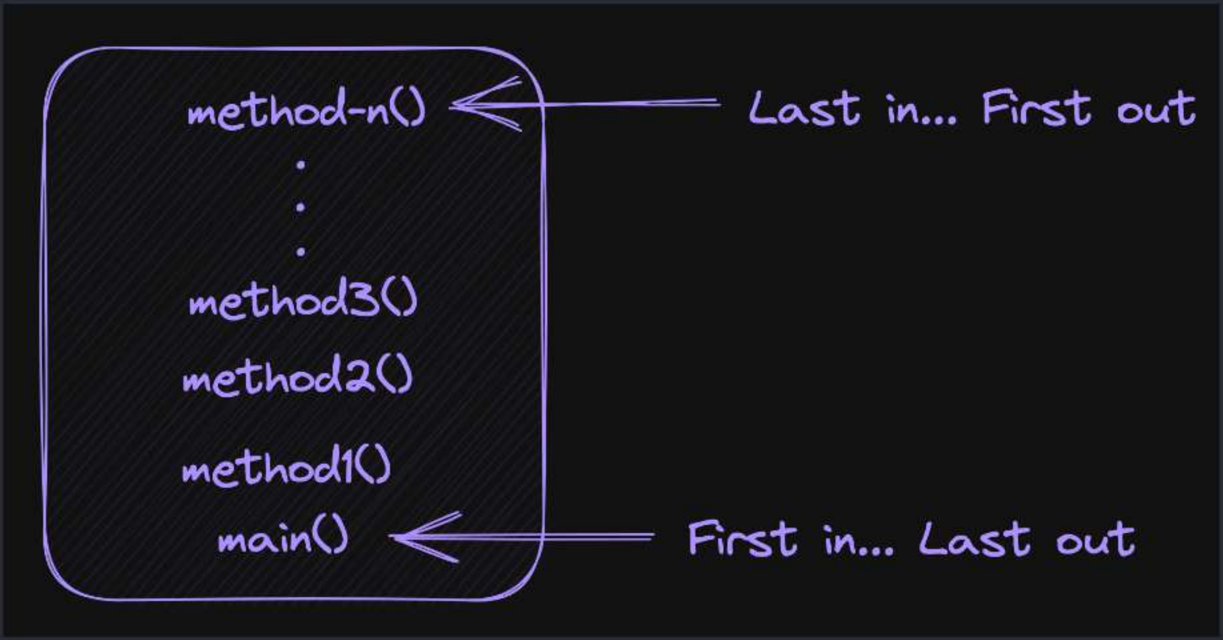
Static OR Global

In this part of the **Memory**, *static/global variables* are stored.
Like... their *address*, *name*, etc...

Stack

In this part of the *memory*, the **functions / methods** stack up on one another when they are called.

Like this...



As the figure says, the *latest method/function called* will be freed **first** after completing its task... and it'll go on to `method3()` then `method2()` and so on to `main()`... after that, the program will be terminated.

> **Note**
>
> Functions/methods are **pushed** into memory and **popped** out of memory.

Heap

In **heap**, the memories are *allocated* by us programmers which are used for other variables like *Pointers*... These are stored during *runtime*.

Heap helps us *allocate* and *deallocate* memory as we please.

For Example:
We can use `new` keyword to allocate memory in **Heap**.

> **Note**
>
> In heap, there isn't a specific allocation of memory... like, it grows

Pointers

Definition:

Pointers are the variables that *points to other variables* by storing *their address*...

There are **two types of Pointers**:

1) *Single Pointer*
2) *Double Pointer*

Let's talk about *Pointer Arithmetics*.

Oh! And there's also *Dangling Pointers* and *Void Pointers*.

Single Pointer

Single Pointer just points to a single variable...

For Example:

```C++
int a = 25;
int *ptr = &a;    //&a means address of 'a'

cout << a << endl; // 25
cout << ptr << endl; // 0xbbbaaa321
cout << *ptr << endl; // 25
```

So what's actually happening here is that `ptr` stored the address of `a` and now `ptr` is pointing to `a`... Simple right?

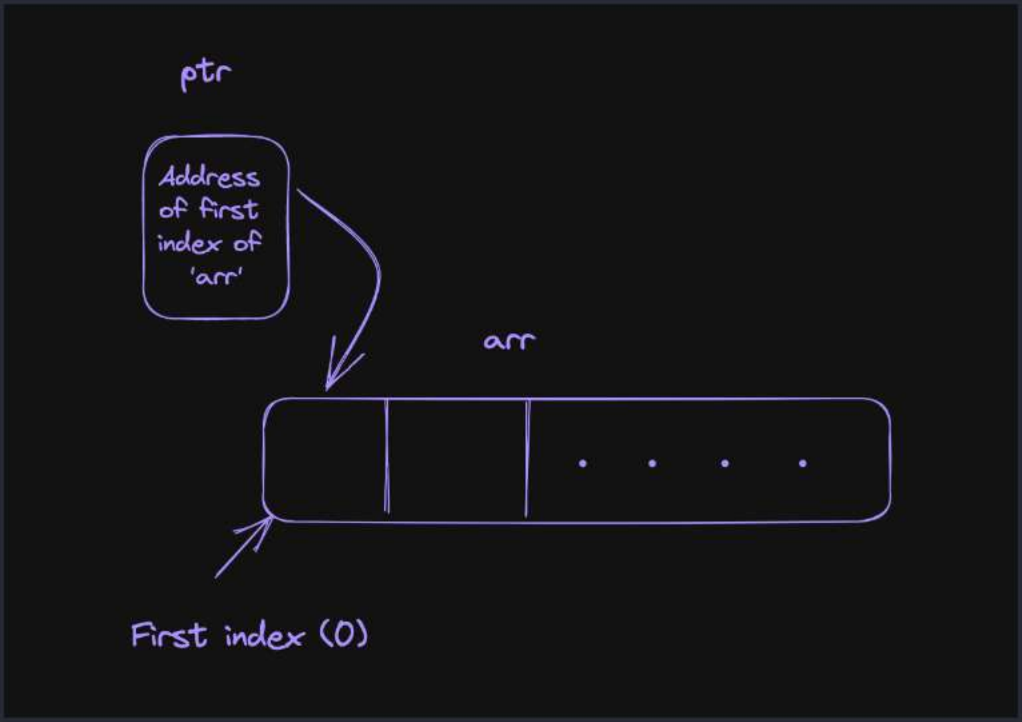
> **Note**
>
> `&` is a *reference* operator and `*` is a *Dereference Operator*.

Pointers and Arrays behave a bit differently tho...

Pointers and Arrays

When **Pointers** points to *Arrays*, the pointer is basically holding the *address of the first of that array*...

```C++
int arr[];
int *ptr = arr;
```

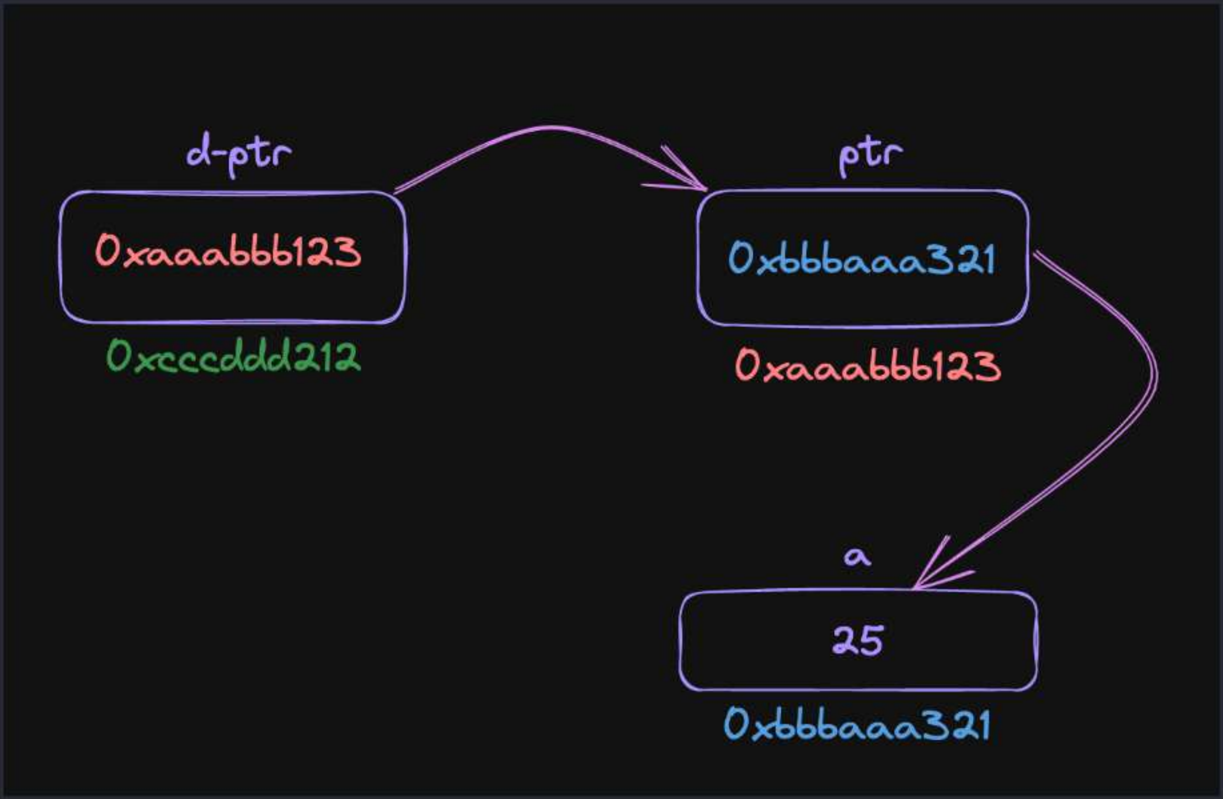


> ✎ **Note**
>
> We don't use `&` operator when we want to point a pointer to an array...

Double Pointer

Double Pointer is the same as *Single Pointer* but the only difference is that, **Double Pointer** can point to a *pointer*...

```C++
int a = 25;
int *ptr = &a;    //&a means address of 'a'
int **d-ptr = &ptr;
```



Pointer Arithmetics

There are some arithmetic operators that we can use on Pointers because Pointer *holds the address of a variable* and that address is a *numeric value*.

Dangling Pointers

Dangling pointers is **a situation where you have valid pointers in the stack, but it is pointing to invalid memory**.

You might end up in this situation when you *deallocate the heap memory before the pointers* in stack are deallocated. This is a *security issue*.

Void Pointers

We use this pointer when we *don't know where we want to a pointer to point to*... By that I mean to **what datatype** the pointer should point to.

We can create the void pointer like this:

```C++
void *voidPointer;
```

Linked List

Linked list has the following Properties which differentiates it from Array.

- Non-contiguous
- Dynamic
- Heterogeneous
- Abstract Data Type (ADT)

Linked List consists of **Nodes** which are linked together to create some sort of a *train* (Like the train has carts which are connected together) like Array.

To create a *Linked List*, we need some stuff xD

Let's make a Linked list for a student...

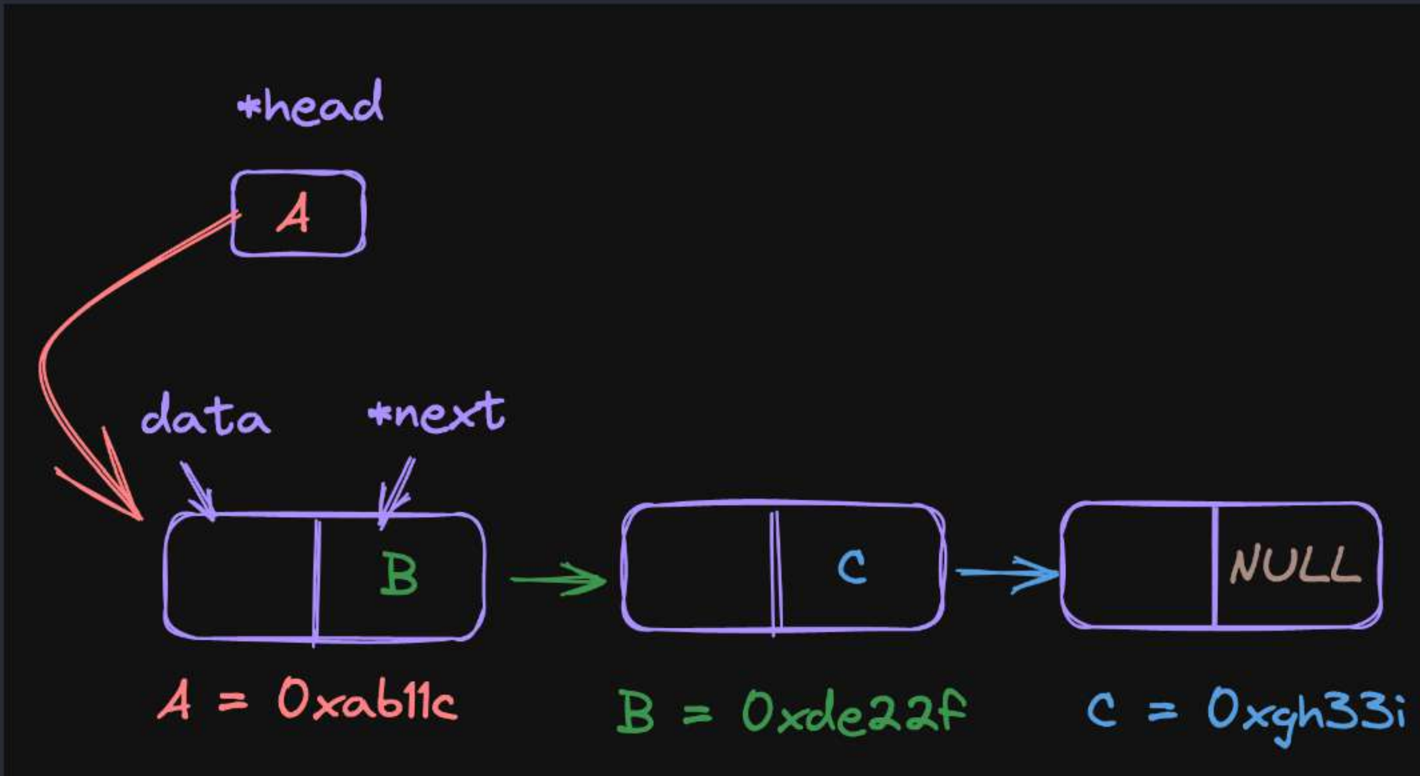
First create a **Structure or a Class** for a *Student*.

```
struct Student
{
    int data;
    Student *next;
};
```

We have a structure here ok? Notice `Student *next;`? Yeah... we'll look into that in a bit... But first, lets make an Object for `Student`.

```
int main()
{
    Student S1;
    Student *ptr;
}
```

Ok, so let's visualize what's happening here...



Operations on List:

`IsEmpty` : Determine whether or not the list is empty.

`InsertNode` : Insert a new node at a particular position.

`FindNode` : Find a node with a given value.

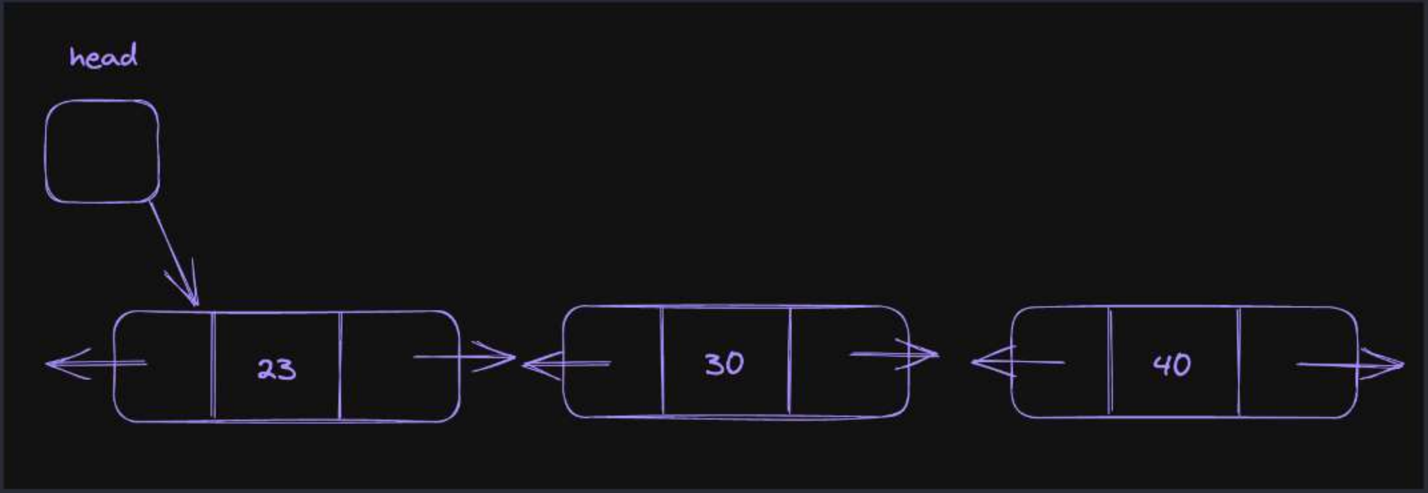
`DeleteNode` : Delete a node with a given value.

`DisplayList` : Print all the nodes.

There's also [Doubly Linked List](#).

Doubly Linked List

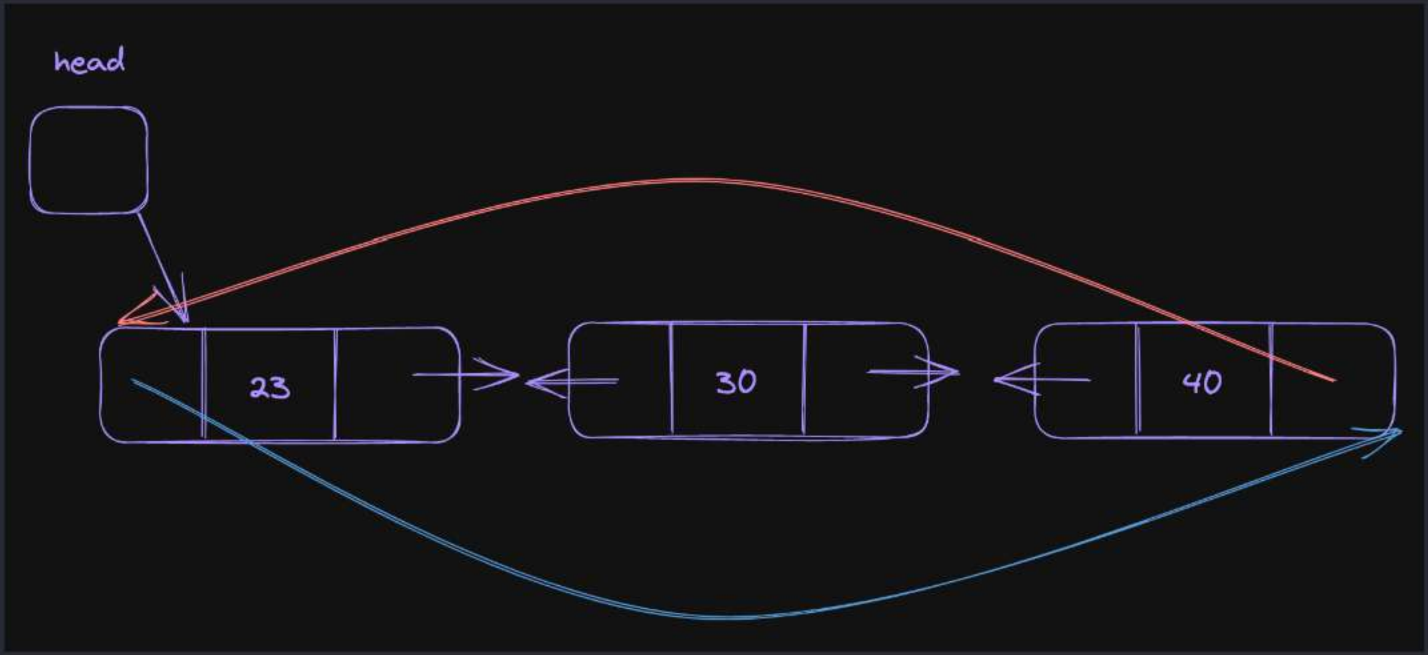
In this kind of list there's a little bit of change in the nodes...
We were making nodes with *2 boxes (Memory Locations)* but in this one, we'll make *3 boxes for a single node*.
The **center box will consist some kind of data**, the **right box will point to the next node**, **the left box will point to the previous node**. The header will also be pointing to it.



This can be implemented like this:

```
struct Node {
    int data;
    Node *next;
    Node *prev;
}
```

Now, to make a **Circular Linked list**, we'll make the first node point to the last node and the last node to the first Node.



But let's not talk about that for now... xD

Advantages of Simple DLL:

- We can go both back and forth with *DLL*.
- We can quickly insert a new node.
- We can also quickly delete a Node without making an extra pointer that points to the previous node because we can go back and forth... In other words, *Traversing made easy*.

> **Note**
>
> It isn't a good practice to make an extra pointer that points to the last Node because in Linked List, there can only be one entry and that is from the header.

Inserting a Node:

If we insert a Node in the beginning of the DLL, we have to make the `*next` of the new node to point to the former first node and the `*prev` of the former first node to the new node.

Disadvantages:

- Every node of DLL requires extra space for a previous pointer. It is possible to implement DLL with single pointer though.
- All operations require an extra pointer previous to be maintained.

Operations:

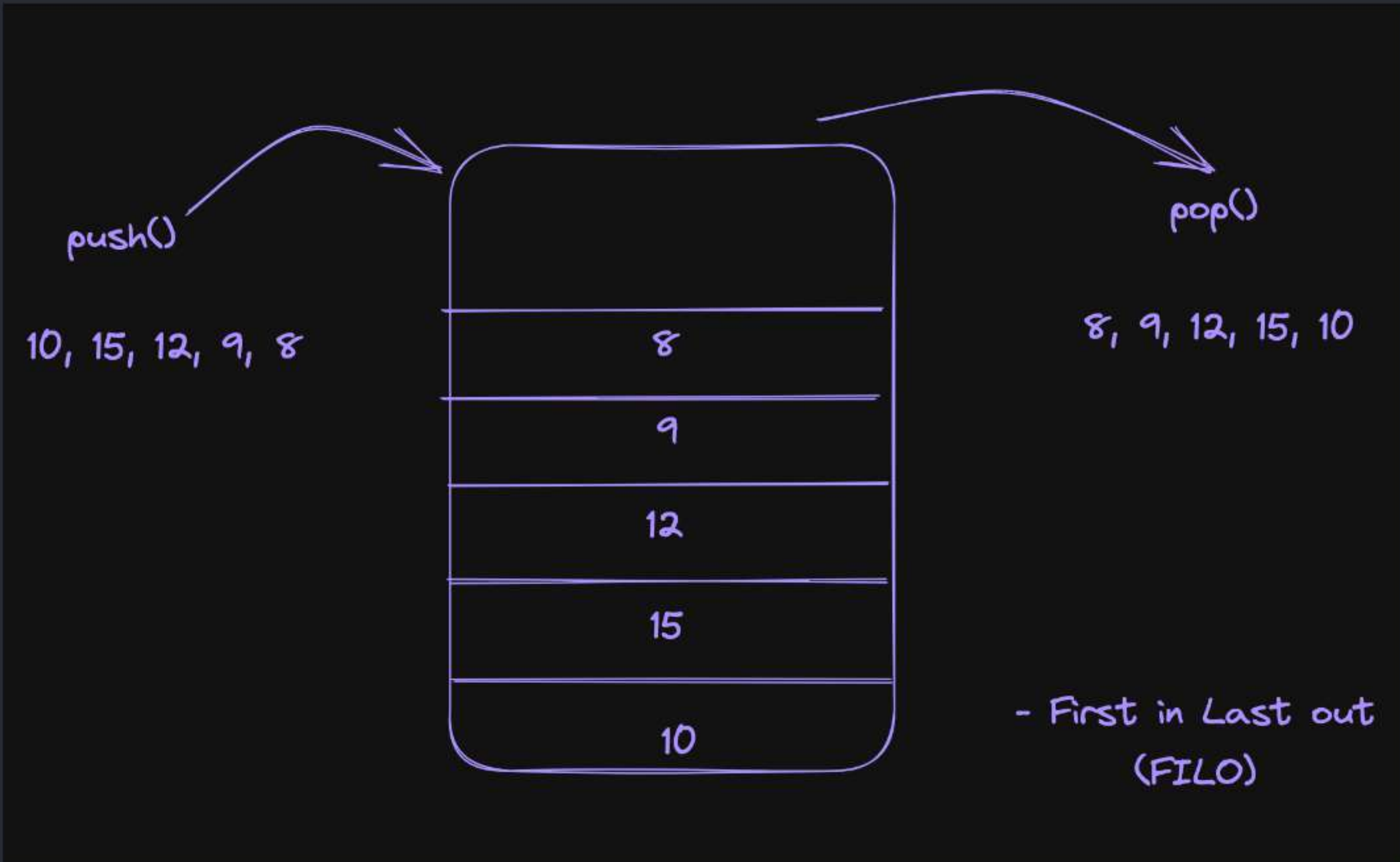
- Insertion in any part
- Deletion at any part
- Searching
- Traversing

Stack

Introduction:

- Stack is a Linear Data Structure
- Will be created with Linked List.

Here's a pictorial representation of how it works:



> Note
>
> When pushing values through *Stack*, we'll start incrementing the loop pointer from `-1` and then while popping, we'll decrement that same loop pointer till all the values are popped. We can make the Loop pointer *Global*.
>
> It will be called `TOP` indicating the *top of the stack*.

How to insert Elements in a Linked List like Stack?

As we know that the starting point of the linked list is from the `*head`... So we'll implement the method `insertionFromBegining`. This will help us easily implement the `push()` and `pop()` methods.

> Extra info
>
> The concept of Stack also applies to *Function calls*... think about it ;p

Operations on Stack:

- `Push()`
- `Pop()`
- `Status() or TOP()`

Now let's see How to Implement Stack.

We also have to know about Precedence of Operators...

How to Implement Stack

First step is to *Declare* `TOP`.

```
stack[100];
n = 100;
TOP = -1;
```

Now we saw *2 important functions...* Those are `PUSH()` and `POP()`.

> **Note**
>
> We can only call `POP()` IF `TOP != -1` that is, the Stack is *Empty*.

Ok, so now to add values into stack, we'll increment `TOP` by 1.

So:

```
PUSH(int val) {
    TOP++;
    s[TOP] = val;
}
```

Our first element has been added at `0` index (index of `TOP`). We can do this process over and over till the size of array, that is `100`...

So, for that, we'll add a condition...

```
if(TOP > n-1) {
    // Don't push more elements
    cout << "Stack is FULL!" << endl;
}
```

For `POP()`:

```
POP() {
    cout << "Popped " << s[TOP] << endl;
    TOP--;
}
```

Limitation:

- We have to check the size of array before *Pushing* and element into the Stack...

Precedence of Operators

Operators	Precedence
()	(3)
^	(2) R -> L
/ *	(1) L -> R
+ -	(0) L -> R

Types:

1. Infix
2. Prefix
3. Postfix

Example:

Let's look at the Working of Operators in Backend.

Working of Operators in Backend

When operations are performed, the operators are stored in Stack...

Let's look at it a bit more...

There are 3 columns:

if we have an equation:

$$A + (B * C - (D/E^F) * G) * H$$

Read	Output	Stack	Instruction
A	A		
+	A	+	Stored + in stack
(	A	+(	Stored (in Stack
B	AB	+(	
*	AB	+(*	Stored * after (
C	ABC	+(*	
-	ABC*	+(-	<--- Popped * and replaced it with -
(	ABC*	+(-(	Stored (in Stack
D	ABC*D	+(-(	
/	ABC*D	+(-(/	
E	ABC*D	+(-(/	
^	ABC*D	+(-(/^	
F	ABC*DEF^/	+(-	
)	ABC*DEF^/G*-H	+*	
	ABC*DEF^/G*-H*+		

Okay... we know the backend... but what about the Frontend... ouchie

Let's have a look at How to implement it???.

How to implement it???

We can create a Stack using the keyword `stack <int> S;`.

Basic Stack Methods:

- `S.top()` : Used to print the *TOP* Element in Stack.
- `S.push(val)` : Used to insert an Element in Stack.
- `S.pop()` : Used to delete the *TOP* element in Stack.

Queue

We can Implement *queue* with arrays and Linked Lists.

Drawback of Queue with Arrays:

- When we dequeue, the front of the queue remains empty and unaccessible and nothing can be added there so it remains empty and useless which's a waste.

We also have Circular Queue.

Circular Queue

To fill up the empty spaces at the front of the Queue which we discussed in Queue, we can use *Circular Queue*.

Implementation:

We need 2 variables like before:

1. `front = -1`
2. `rear = -1`

To add out first element (*Enqueue*), we have to check if the Queue is empty... it'll be empty only when `front = -1`. We can implement it as follows:

```
if(front == -1) {
    front = rear = 0;
    arr[r] = value;
}
```

But if it wasn't already empty... then:

```
else {
    r++;
    arr[r] = value;
}
```

But what about when *Queue is full*?
We can implement it as follows:

```
if(rear == size - 1) {
    cout << "Queue is Full!" << endl;
}
```

But... this condition is incomplete... what if we dequeued an element? The above condition will still be satisfied... so... we have to check the front aswell:

```
if(rear = size - 1 && front != 0) {
    r = 0;
    a[r] = value;
}
```

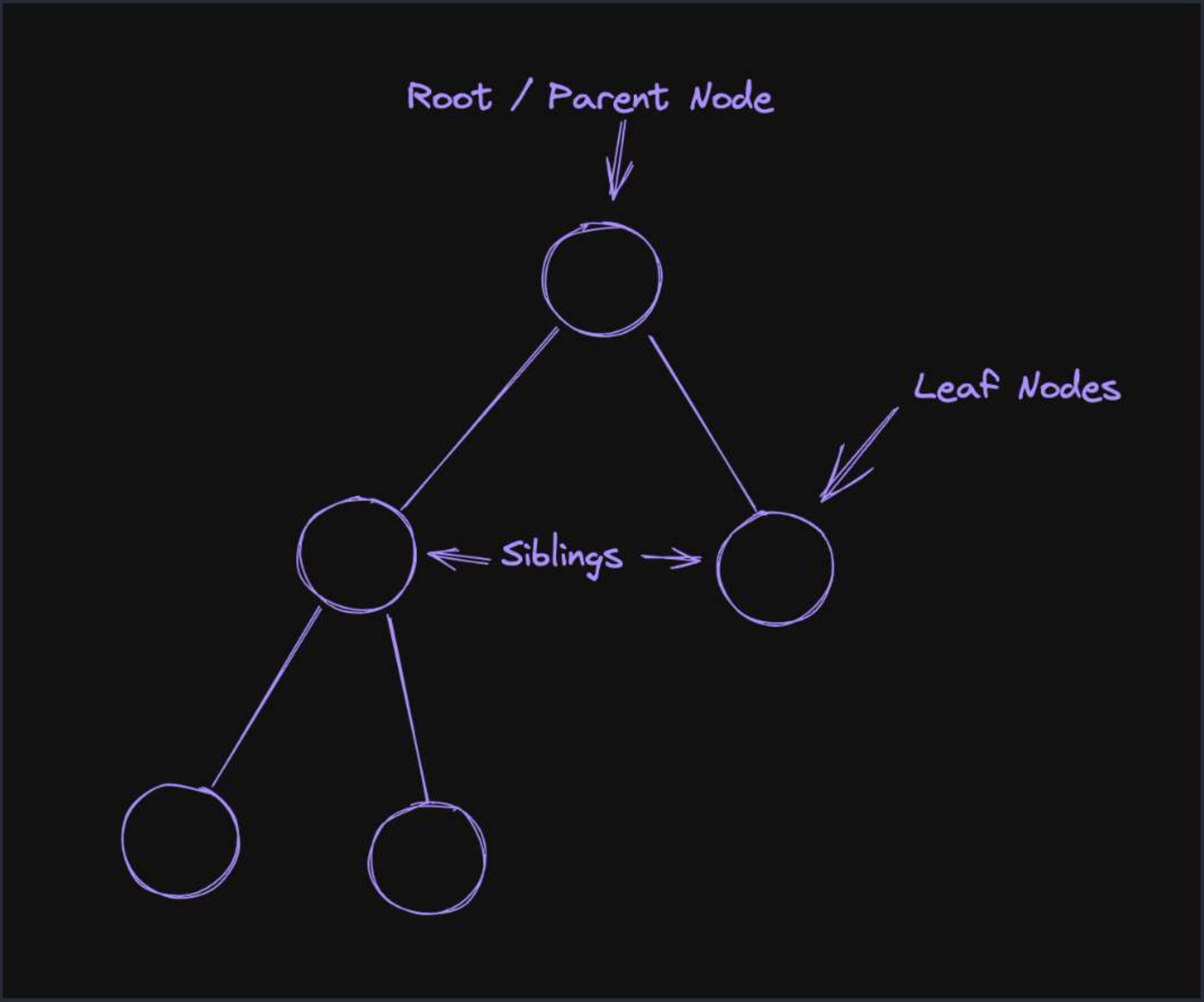
Suppose we dequeued till front is at the rear, we set rear to 0 that is at the beginning...
To then again check if the queue is full again (Circular), we'll use the following condition:

```
if((rear + 1) % length == front) {
    cout << "Queue is Full" << endl;
}
```

Trees

- A *Tree* is a Non-Linear Data Structure.
- It has *Vertices and Edges*.

Just like how a real tree has extreme points *i.e: Roots and leaves*... Tree data structure has a similar concept...



- A *Node* is the basic building unit of the tree structure.
- *Root Node* Doesn't have a Parent.

There a couple terminologies for Trees...

- Height
- Depth

Non-Linear Data Structure

Non-linear Data Structures include structures which are arranged in a random order like **Trees**, **Graphs**, and **Tables**.