DATA STRUCTURES & ALGORITHMS

Notes made by Soikot Shahriar
[t.me/soikot_shahriaar]
[github.com/soikot-shahriaar]

Data Structures & Algorithms

Data Structures - These are like the ingredients we need to build efficient algorithms. These are the ways to arrange data so that they (data items) can be used efficiently in the main memory. Examples: Array, Stack, Linked List, and many more.

Algorithms – Sequence of steps performed on the data using efficient data structures to solve a given problem, be it a basic or real-life-based one. Examples include: sorting an array.

Some other Important Terminologies:

Database – Collection of information in permanent storage for faster retrieval and updation. Examples are MySQL, MongoDB, etc.

Data Warehouse – Management of huge data of legacy data (the data we keep at a different place from our fresh data in the database to make the process of retrieval and updation fast) for better analysis.

Big data – Analysis of too large or complex data, which cannot be dealt with the traditional data processing applications.

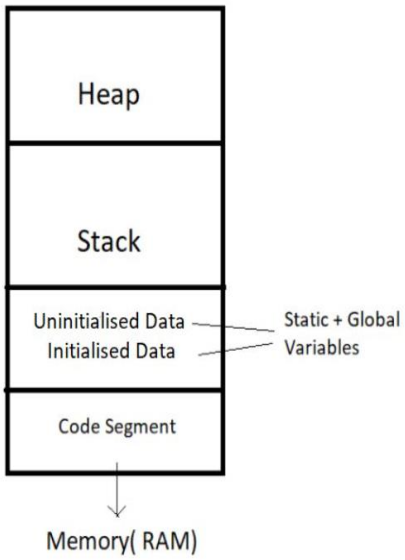
Memory Layout of C Programs:

When the program starts, its code gets copied to the main memory.

The **stack** holds the memory occupied by functions. It stores the activation records of the functions used in the program. And erases them as they get executed.

The **heap** contains the data which is requested by the program as dynamic memory using pointers.

Initialized and uninitialized data segments hold initialized and uninitialized global variables, respectively.



Time Complexity and Big O Notation:

Time Complexity is the study of the efficiency of algorithms. It tells us how much time is taken by an algorithm to process a given input.

Calculating Order in terms of Input Size:

In order to calculate the order (time complexity), the most impactful term containing n is taken into account (here n refers to Size of input). And the rest of the smaller terms are ignored.

Let us assume the following formula for the algorithms in terms of input size n:

Algo 1: $k_1n^2 + k_2n + 36 = O(n^2)$

Highest Order Term — Can ignore other lower order terms

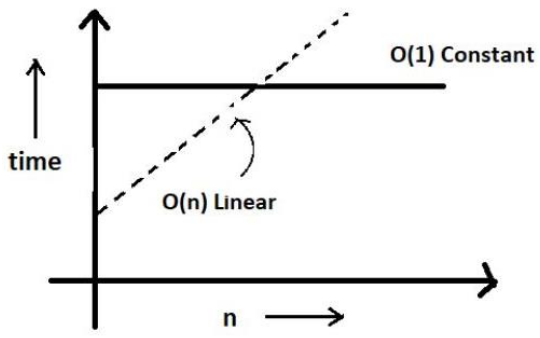
Algo 2: $k_1k_2^2 + k_3k_2 + 8 = O(n^0) = O(1)$

These are the formulas for the time taken by their program.

Here, we ignored the smaller terms in algo 1 and carried the most impactful term, which was the square of the input size. Hence the time complexity became n^2. The second algorithm followed just a constant time complexity.

Big O stands for 'order of' in our industry, but this is pretty different from the mathematical definition of the big O. Big O in mathematics stands for all those complexities our program runs in. But in industry, we are asked the minimum of them. So this was a subtle difference.

Visualizing Big O: If we were to plot O(1) and O(n) on a graph,



Asymptotic Notation:

Asymptotic Notation gives us an idea about how good a given algorithm is compared to some other algorithm.

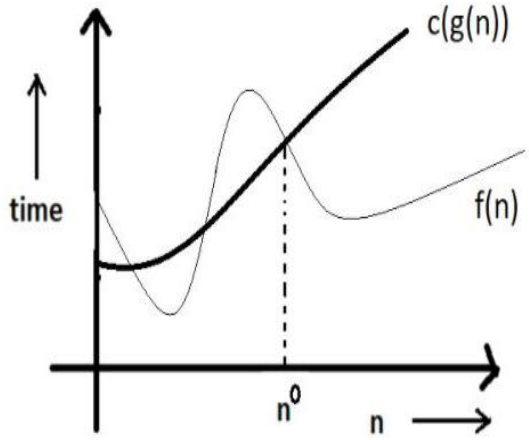
Big Oh Notation (O): used to describe an asymptotic upper bound.

Mathematically, if f(n) describes the running time of an algorithm; f(n) is O(g(n)) if and only if there exist positive constants c and n° such that:

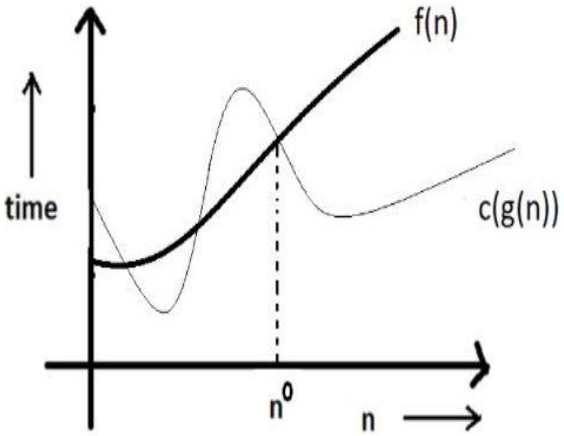
$$0 \leq f(n) \leq c\ g(n) \qquad \text{for all } n \geq n°$$

Here, n is the input size, and g(n) is any complexity function, for example: n, n2, etc. (It is used to give upper bound on a function)

If a function is O(n), it is automatically O(n2) as well! Because it satisfies the equation given above.



Big Oh Notation (O)



Big Omega Notation (Ω)

Big Omega Notation (Ω): used to give the lower bound on a function.

Just like O notation provides an asymptotic upper bound, Ω notation provides an asymptotic lower bound.

Let f(n) define the running time of an algorithm; f(n) is said to be Ω (g(n)) if and only if there exist positive constants c and n° such that:

$$0 \leq c\ g(n) \leq f(n) \qquad \text{for all } n \geq n°$$

If a function is Ω (n2) it is automatically Ω (n) as well since it satisfies the above equation.

Big Theta Notation (θ): provides a better picture of an algorithm's run time.

Let f(n) define the running time of an algorithm.

F(n) is said to be θ (g(n)) if f(n) is O (g(n)) and f(x) is Ω (g(n)) both.

Mathematically,

$0 \le f(n) \le c_1 g(n) \quad \forall \quad n \ge n_o$

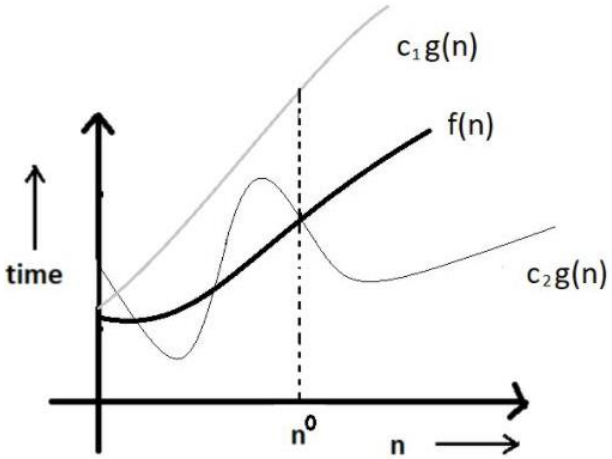
$0 \le c_2 g(n) \le f(n) \quad \forall \quad n \ge n_o$

for sufficiently large value of n

Merging both the equations, we get:

$0 \le c2\ g(n) \le f(n) \le c1\ g(n) \quad \forall \quad n \ge n_o$

The equation simply means that there exist positive constants c1 and c2 such that f(n) is sandwiched between c2 g(n) and c1 g(n).



Increasing order of common runtimes:

$1 < \log n < n < n\log n < n^2 < n^3 < 2^x < n^x$

Better

Worse

Common runtimes from better to worse

Best, Worst and Average Case Analysis:

A program finds it best when it is effortless for it to function and worse otherwise.

By considering a search algorithm used to perform a sorted array search, we will analyze this feature.

Analysis of a Search Algorithm:

Consider an array that is sorted in increasing order

1 7 18 28 50 180

We have to search a given number in this array and report whether it's present in the array or not. In this case, we have two algorithms, and we will be interested in analyzing their performance separately.

Algorithm 1 – Start from the first element until an element greater than or equal to the number to be searched is found.

Algorithm 2 – Check whether the first or the last element is equal to the number. If not, find the number between these two elements (center of the array); if the center element is greater than the number to be searched, repeat the process for the first half else, repeat for the second half until the number is found. And this way, keep dividing your search space, making it faster to search.

Analyzing Algorithm 1: (Linear Search)

We might get lucky enough to find our element to be the first element of the array. Therefore, we only made one comparison which is obviously constant for any size of the array.

- Best case complexity = O(1)

If we are not that fortunate, the element we are searching for might be the last one. Therefore, our program made 'n' comparisons.

- Worst-case complexity = O(n)

For calculating the average case time, we sum the list of all the possible case's runtime and divide it with the total number of cases. Here, we found it to be just O(n). (Sometimes, calculation of average -case time gets very complicated.)

Analyzing Algorithm 2: (Binary Search)

If we get really lucky, the first element will be the only element that gets compared. Hence, a constant time.

- Best case complexity = O(1)

If we get unlucky, we will have to keep dividing the array into halves until we get a single element. (that's the array gets finished)

Hence the time taken: n + n/2 +n/4 + ... +1 = log n with base 2

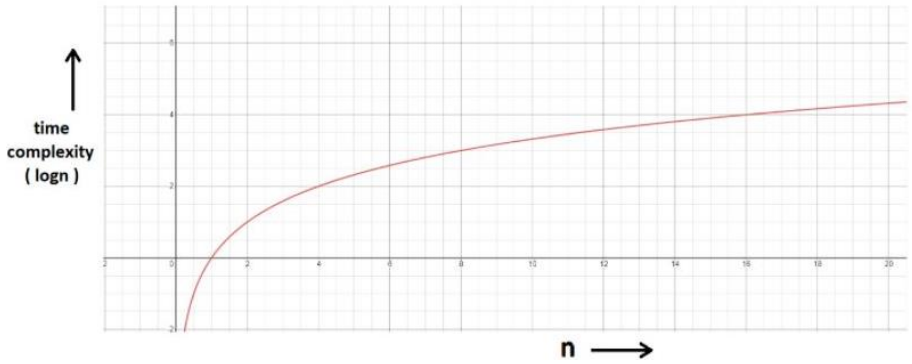
- Worst-case complexity = O(log n)

N.B: log n refers to how many times needed to divide n units until they can no longer be divided (into halves).

log 8 = 3 ⇒ 8/2 + 4/2 + 2/2 → Can't break anymore.

log 4 = 2 ⇒ 4/2 + 2/2 → Can't break anymore.

We can see in the graph, how slowly the time complexity (Y- axis) increases when we increase the input n (X - axis).



Why we can't calculate complexity in seconds when dealing with time complexities:

a. Not everyone's computer is equally powerful. So, we avoid handling absolute time taken.
b. We just measure how time (runtime) grows with input in the asymptotic analysis.

Techniques to calculate Time Complexity:

Once we are able to write the runtime in terms of the size of the input (n), we can find the time complexity. For example:

T(n) = n2 → O(n^2)

T(n) = log n → O(log n)

Here are some tricks to calculate Complexities:

- Drop the Constants:

Anything we might think is O(kn) is O(n) as well (where k is a constant). This is considered a better representation of the time complexity since the k term would not affect the complexity much for a higher value of n.

- Drop the non-dominant terms:

Anything we represent as O(n2+n) can be written as O(n2). Similar to when non-dominant terms are ignored for a higher value of n.

- Consider all variables which are provided as input:

O (mn) and O (mnq) might exist for some cases.

In most cases, we try to represent the runtime in terms of the inputs which can even be more than one in number. For example,

The time taken to paint a park of dimension m * n → O (kmn) → O (mn)

Notes Made by

SOIKOT SHAHRIAR

[t.me/soikot_shahriaar]

[github.com/soikot-shahriaar]

Arrays and Abstract Data Types

ADTs or Abstract Data Types are the ways of classifying data structures by providing a minimal expected interface and some set of methods.

It is very similar to when we make a blueprint before actually getting into doing some job, be it constructing a computer or a building. The blueprint comprises all the minimum required logistics and the roadmap to pursuing the job.

Array – ADT holds the collection of given elements (can be int, float, custom) accessible by their index. An abstract data type is just another data type as an int or float, with some user-defined methods and operations. It's a kind of customized data type.

1. Minimal required Functionality:

We have two basic functionalities of an array, a get function to retrieve the element at index i and a set function to assign an element to some index in the array.

a. get(i) – get element i.
b. set(i, num) – set element i to num.

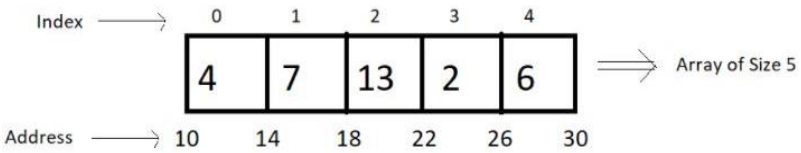
2. Operations:

We can have a whole lot of different operations on the array we created, but we'll limit ourselves to some basic ones.

a. Max ( )
b. Min ( )
c. Search (num)
d. Insert (i, num)
e. Append (x)

Static and Dynamic Arrays:
a. Static arrays – Size cannot be changed
b. Dynamic arrays – Size can be changed

Memory Representations of Array:



- Elements in an array are stored in contiguous memory locations.
- Elements in an array can be accessed using the base address in constant time → O (1).
- Although changing the size of an array is not possible, one can always reallocate it to some bigger memory location. Therefore, resizing in an array is a costly operation.

Implementing an array as an Abstract Data Type:

Suppose we want to build an array as an abstract data type with our customized set of values and customized set of operations in a heap. Let's name this customized array myArray. Let our set of values which will represent our customized array include these parameters:

- total_size
- used_size
- base_address

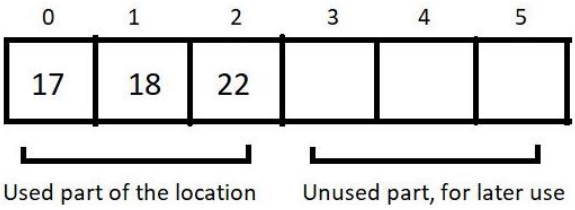
And the operations include operators namely,

- max( )
- get(i)
- set(i, num)
- add(another_array)

So, now when we are done creating a blueprint of the customized array. We can very easily code their implementation, but before that, let's first learn what these values and operations, we have defined.

Understanding the ADT above:

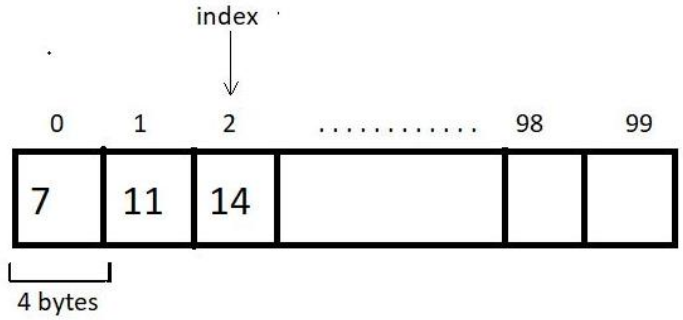
1. total_size: This stores the total reserved size of the array in the memory location.
2. used_size: This stores the size of the memory location used.
3. base_address: This is a pointer that stores the address of the first element of the array.



Here, the total_size returns 6, and the used_size returns 3.

Operations on Arrays in Data Structures:

Important note on Arrays:



If we create an array of length 100 using arr[100] in C language, we need not use all the elements. It is possible for a program to use just 60 elements out of these 100. (But we can't go beyond 100 elements).

Operations on an Array:

1. Traversal
2. Insertion
3. Deletion
4. Search

Other operations include sorting ascending, sorting descending etc.

Traversal:

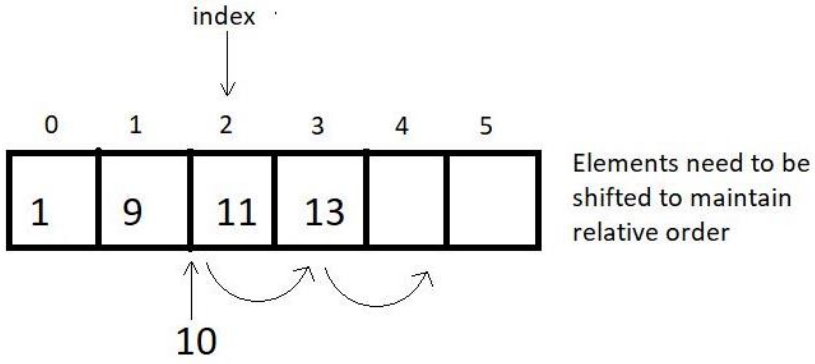
Visiting every element of an array once is known as traversing the array. An array can easily be traversed using a for loop in C language.

Traversal is used for these cases:

a. Storing all elements – Using scanf()
b. Printing all elements – Using printf()
c. Updating elements.

Insertion:

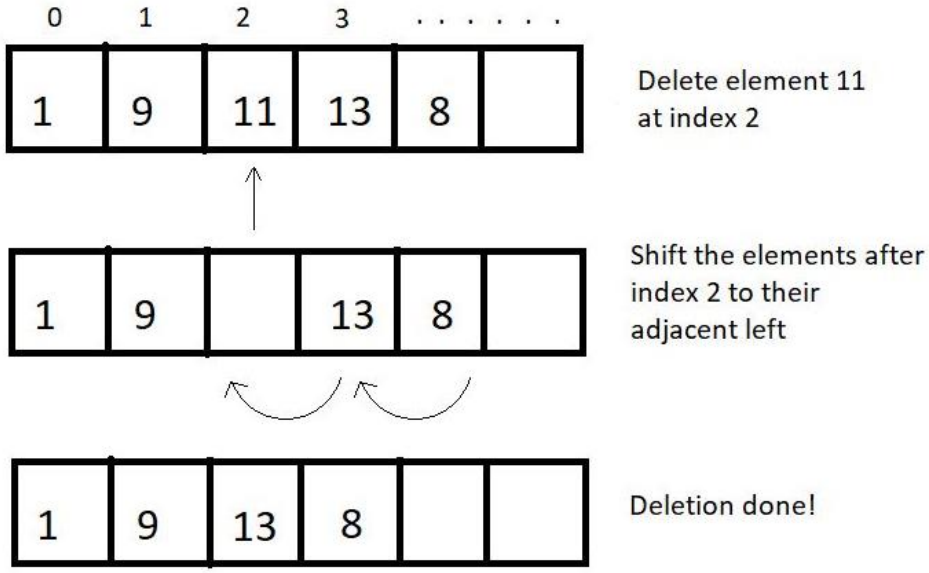
An element can be inserted in an array at a specific position. For this operation to succeed, the array must have enough capacity. Suppose we want to add an element 10 at index 2 in the below-illustrated array, then the elements after index 1 must get shifted to their adjacent right to make way for a new element.



When no position is specified, it's best to insert the element at the end to avoid shifting, and this is when we achieve the best runtime O(1).

Deletion:

An element at a specified position can be deleted, creating a void that needs to be fixed by shifting all the elements to their adjacent left, as illustrated in the figure below.

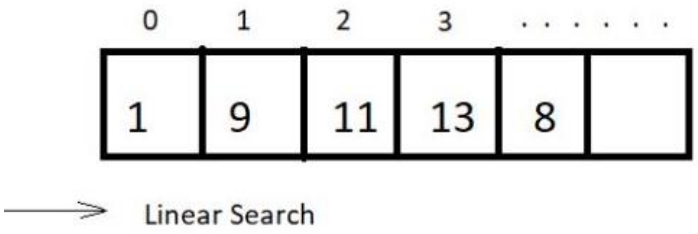


We can also bring the last element of the array to fill the void if the relative ordering is not important.

Searching:

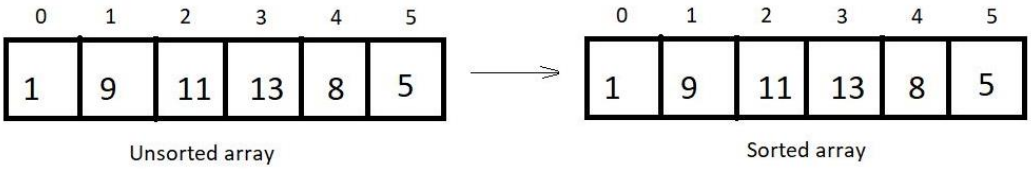
Searching can be done by traversing the array until the element to be searched is found. O(n)

Therefore, for sorted arrays, the time taken to search is much less than an unsorted array. O(log n)



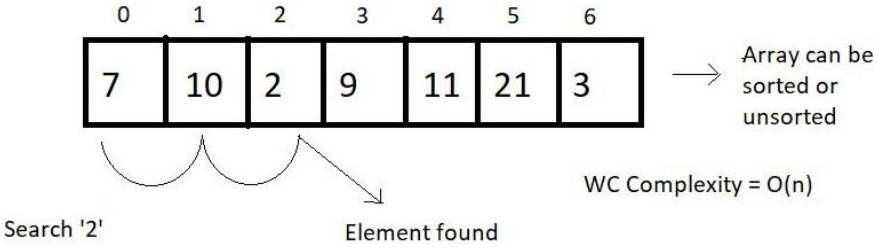
Sorting:

Sorting means arranging an array in an orderly fashion (ascending or descending).



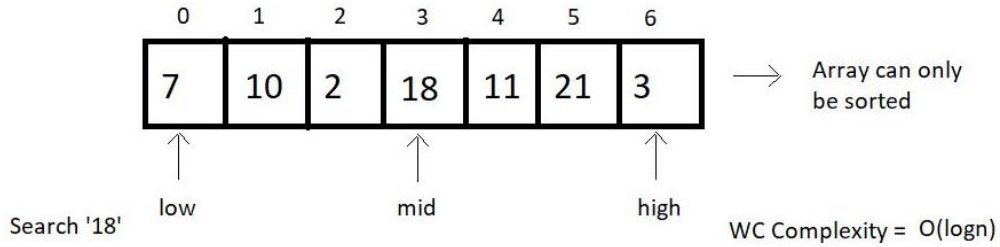
Linear Search and Binary Search:

Linear Search method searches for an element by visiting all the elements sequentially until the element is found or the array finishes. It follows the array traversal method.



Binary Search method searches for an element by breaking the search space into half each time it finds the wrong element.

This method is limited to a sorted array. The search continues towards either side of the mid, based on whether the element to be searched is lesser or greater than the mid element of the current search space.



Linked List

The fundamental differences between Linked lists and Arrays:

Arrays demand a contiguous memory location. Lengthening an array is not possible. We would have to copy the whole array to some bigger memory location to lengthen its size. Similarity inserting or deleting an element causes the elements to shift right and left, respectively.

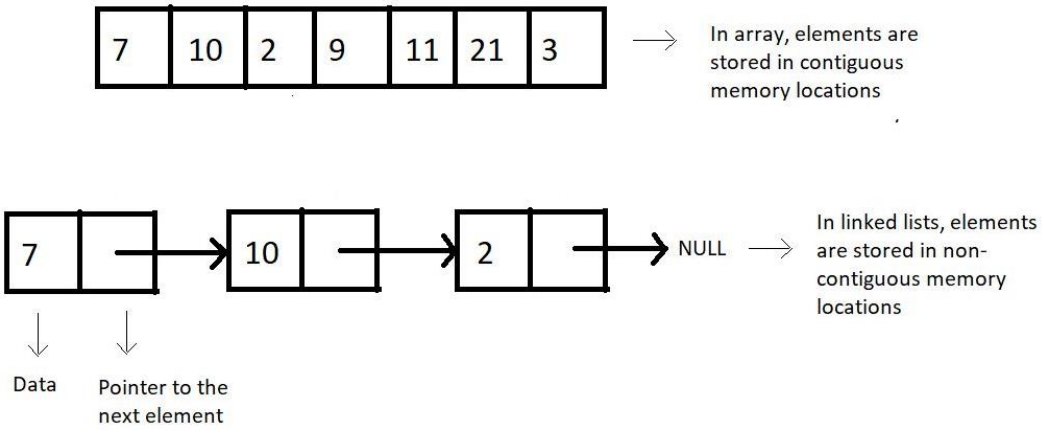
But linked lists are stored in a non-contiguous memory location. To add a new element, we just have to create a node somewhere in the memory and get it pointed by the previous element. And deleting an element is just as easy as that. We just have to skip pointing to that particular node. Lengthening a linked list is not a big deal.

Structure of a Linked List:

Every element in a linked list is called a node and consists of two parts, the data part, and the pointer part. The data part stores the value, while the pointer part stores the pointer pointing to the address of the next node.

Both of these structures (arrays and linked lists) are linear data structures.

Linked Lists VS Arrays:



Why to use Linked Lists: Memory and the capacity of an array remain fixed, while in linked lists, we can keep adding and removing elements without any capacity constraint.

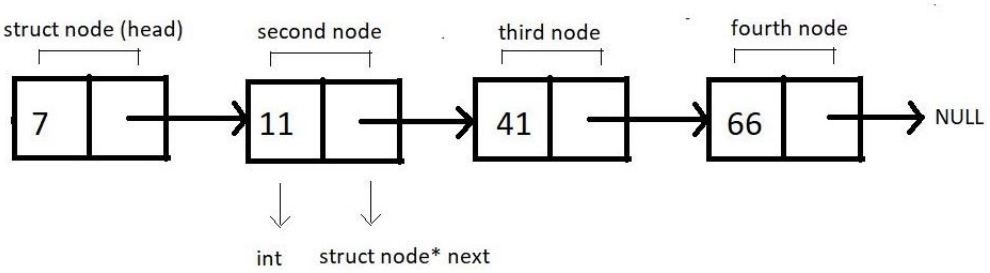
Drawbacks of Linked Lists: Extra memory space for pointers is required (for every node, extra space for a pointer is needed). Random access is not allowed as elements are not stored in contiguous memory locations.

Implementation:

Linked lists are implemented in C language using a structure. We construct a structure named Node.

Then define two of its members, an integer data, which holds the node's data, and a structure pointer, next, which points to the address of the next structure node.

```
struct Node
{
    int data;
    struct Node *next; // Self Referencing Structure
};
```

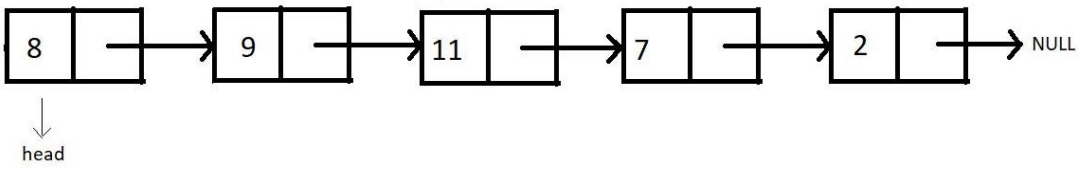


Syntax for allocate memory for nodes in Heap:

```
head = (struct Node *)malloc(sizeof(struct Node));
```

Insertion in a Linked List:

Consider the following Linked List,



Insertion in this list can be divided into the following categories:

- Case 1: Insert at the beginning.
- Case 2: Insert in between.
- Case 3: Insert at the end.
- Case 4: Insert after the node.

For insertion following any of the above-mentioned cases, we would first need to create that extra node. And then, we overwrite the current connection and make new connections. And that is how we insert a new node at our desired place.

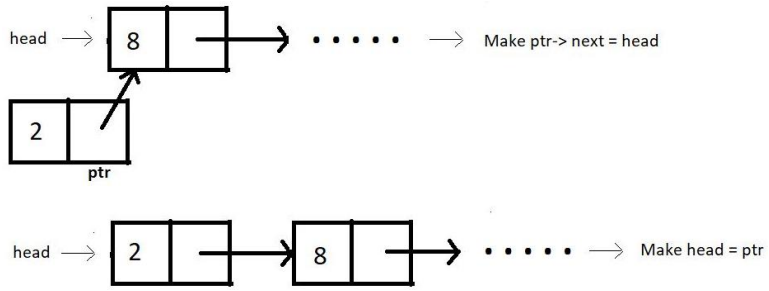
Syntax for creating a Node:

```
struct Node *ptr = (struct Node *)malloc(sizeof(struct Node))
```

The next thing one would need to do is set the data for this node.

Insert at the Beginning:

In order to insert the new node at the beginning, we would need to have the head pointer pointing to this new node and the new node's pointer to the current head.

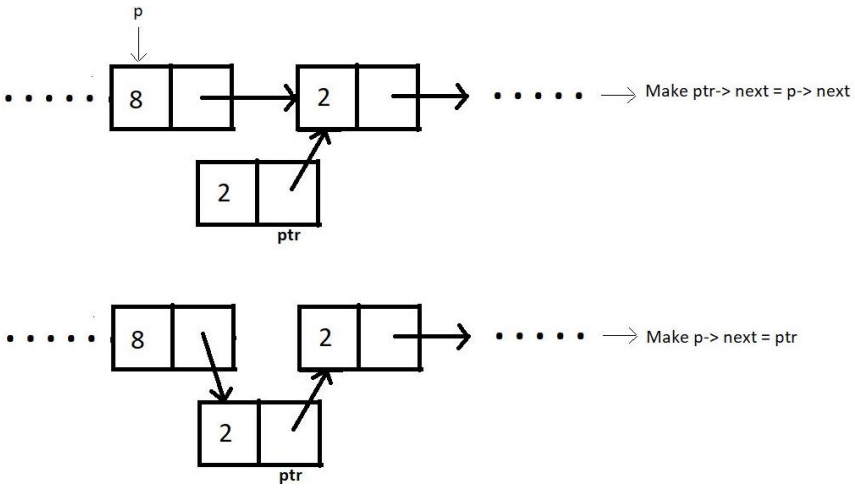


Insert in Between:

Assuming index starts from 0, we can insert an element at index i>0 as follows:

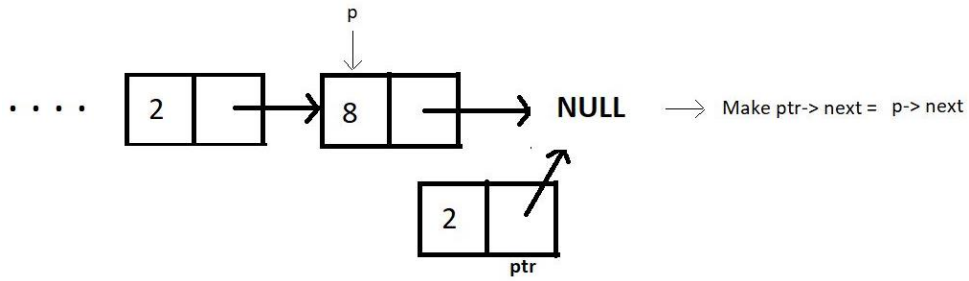
Bring a temporary pointer p pointing to the node before the element we want to insert in the linked list.

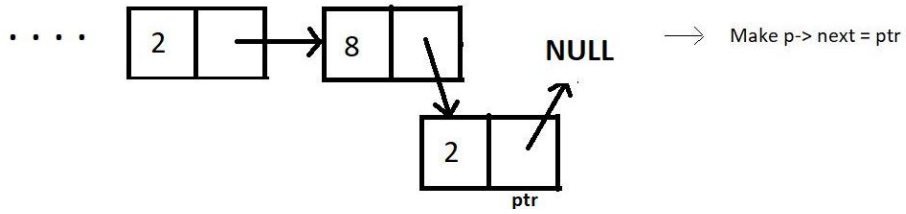
Since we want to insert between 8 and 2, we bring pointer p to 8.



Insert at the End:

In order to insert an element at the end of the linked list, we bring a temporary pointer to the last element.

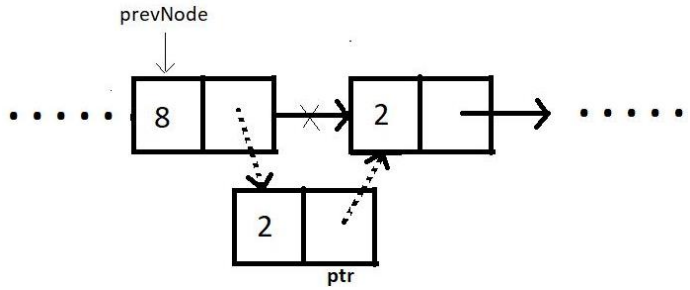




Insert after a Node:

Similar to the other cases, ptr can be inserted after a node as follows:

a. ptr->next = prevNode-> next;
b. prevNode-> next = ptr;

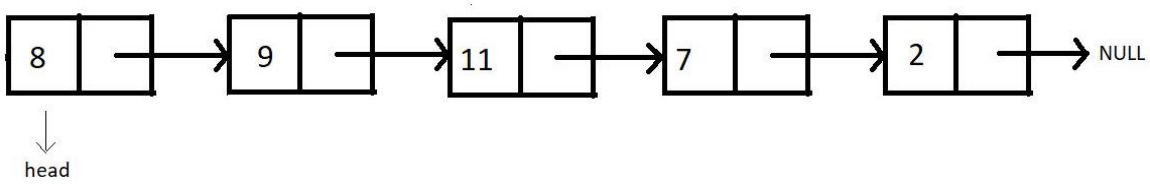


Learning about the **Time Complexity** while inserting these nodes, inserting at the beginning has the time complexity O(1), and inserting at some node in between puts the time complexity O(n) since we have to go through the list to reach that particular node.

Inserting at the end has the same time complexity O(n) as that of inserting in between. But if we are given the pointer to the previous node where we want to insert the new node, it would just take a constant time O(1).

Deletion in a Linked List:

Consider the following Linked List:



Deletion in this list can be divided into the following categories:

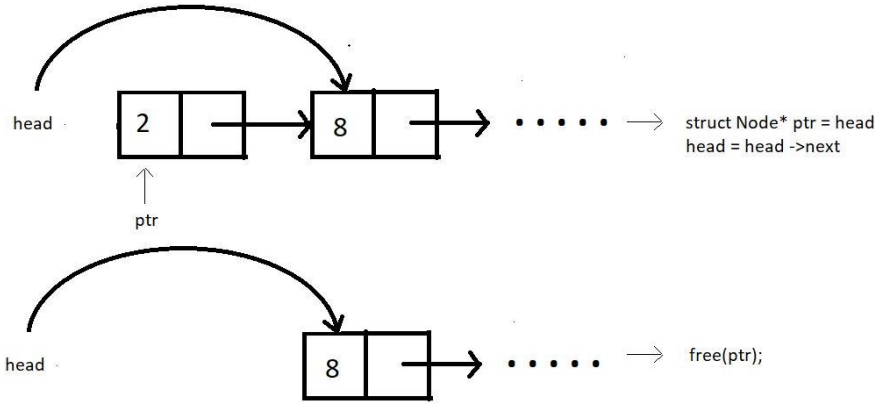
- Case 1: Deleting the first node.
- Case 2: Deleting the node at the index.
- Case 3: Deleting the last node.
- Case 4: Deleting the first node with a given value.

For deletion, following any of the above-mentioned cases, we would just need to free that extra node left after we disconnect it from the list. Before that, we overwrite the current connection and make new connections. And that is how we delete a node from our desired place.

Syntax for Freeing a Node: `free(ptr);`

Delete from the Beginning:

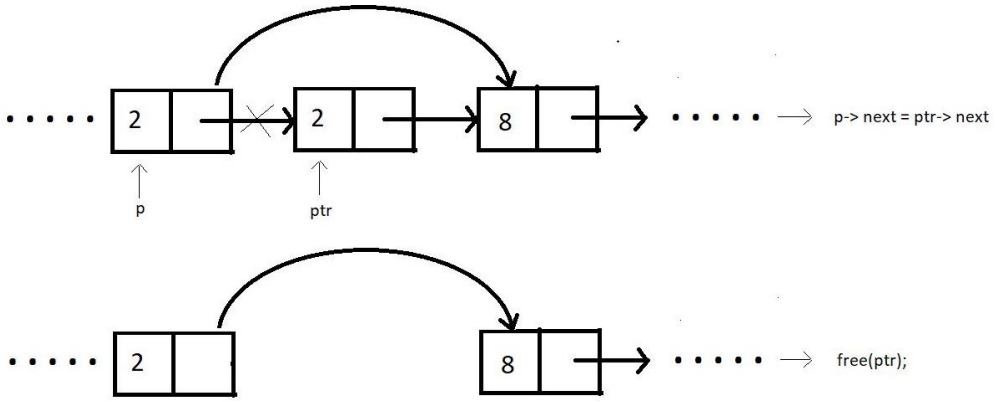
In order to delete the node at the beginning, we would need to have the head pointer pointing to the node second to the head node, that is, head-> next. And we would simply free the node that's left.



Deleting at some Index in Between:

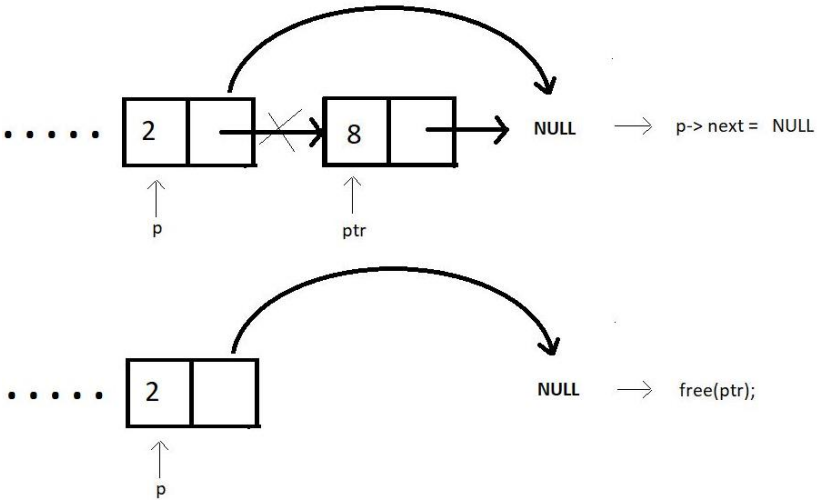
Assuming index starts from 0, we can delete an element from index i>0 as follows:

a. Bring a temporary pointer p pointing to the node before the element we want to delete in the linked list.
b. Since we want to delete between 2 and 8, we bring pointer p to 2.
c. Assuming ptr points at the element we want to delete.
d. We make pointer p point to the next node after pointer ptr skipping ptr.
e. We can now free the pointer skipped.



Delete from the End:

In order to delete an element at the end of the linked list, we bring a temporary pointer ptr to the last element. And a pointer p to the second last. We make the second last element to point at NULL. And we free the pointer ptr.

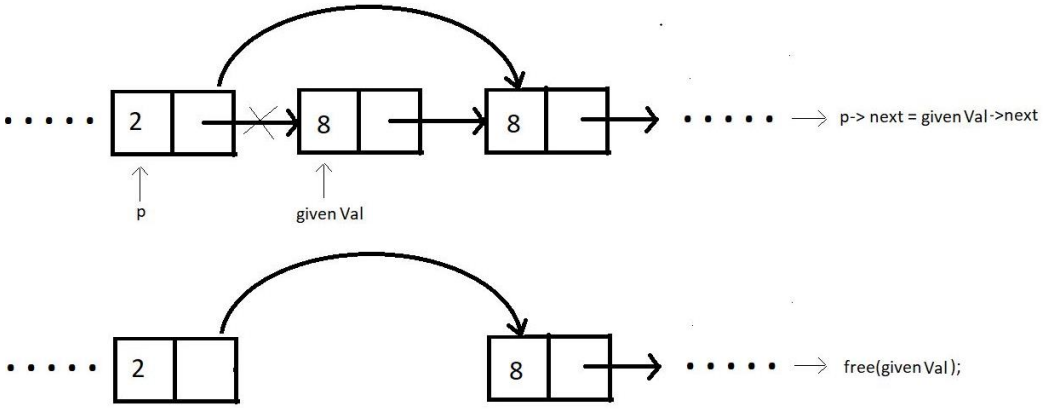


Deleting the First Node with a Given Value:

ptr can be deleted for a given value as well by following few steps:

a. p->next = givenVal-> next;
b. free(givenVal);

Since, the value 8 comes twice in the list, this function will be made to delete only the first occurrence.

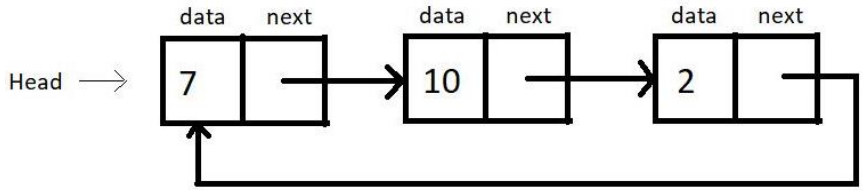


Learning about the **Time Complexity** while deleting these nodes, we found that deleting the element at the beginning completes in a constant time, O(1). Deleting at any index in between is no big deal either, it just needs the pointer ptr to reach the node to be deleted, causing it to follow O(n). And the same goes with case 3 and case 4. We have to traverse through the list to reach that desired position.

Circular Linked List:

A circular linked list is a linked list where the last element points to the first element (head) hence forming a circular chain. There is no node pointing to the NULL, indicating the absence of any end node.

In circular linked lists, we have a head pointer but no starting of this list. Refer to the illustration of a circular linked list below:



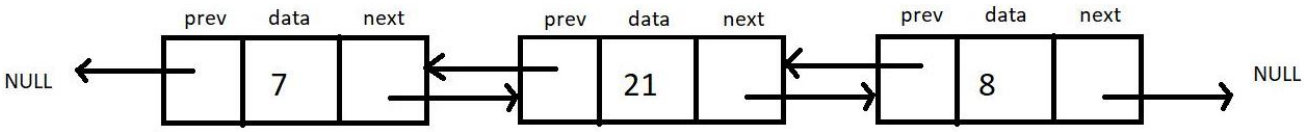
Operations in a Circular Linked List:

Operations on circular linked lists can be performed exactly like a singly linked list. It's just that we have to maintain an extra pointer to check if we have gone through the list once.

Doubly Linked List:

Each node contains a data part and two pointers in a doubly-linked list, one for the previous node and the other for the next node.

Below illustrated is a Doubly Linked List with three nodes. Both the end pointers point to the NULL.



How is it different from a singly linked list?

a. A doubly linked list allows traversal in both directions. We have the addresses of both the next node and the previous node. So, at any node, we'll have the freedom to choose between going right or left.
b. A node comprises three parts, the data, a pointer to the next node, and a pointer to the previous node.
c. Head node has the pointer to the previous node pointing to NULL.

Implementation:

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

Operations in a Doubly Linked List:

The Insertion and Deletion on a doubly linked list can be performed by recurring pointer connections, just like we saw in a singly linked list.

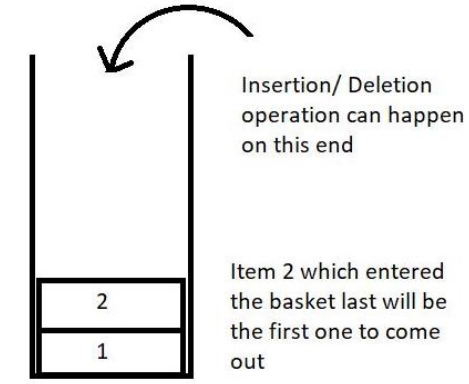
The difference here lies in the fact that we need to adjust two-pointers (prev and next) instead of one (next) in the case of a doubly linked list.

Stack

A stack is a linear data structure. Any operation on the stack is performed in LIFO (Last In First Out) order. This means the element to enter the container last would be the first one to leave the container. It is imperative that elements above an element in a stack must be removed first before fetching any element.

An element can be pushed in this basket-type container illustrated here.

Any basket has a limit, and so does our container too. Elements in a stack can only be pushed to a limit. And this extra pushing of elements in a stack leads to stack overflow.



LIFO (Last In First Out)

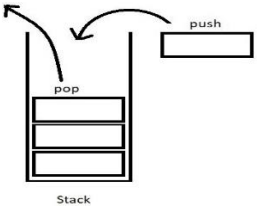
Applications of Stack:

We have talked about function calls before as well. A function until it returns reserves a space in the memory stack. Any function embedded in some function comes above the parent function in the stack. So, first, the embedded function ends, and then the parent one. Here, the function called last ends first (**LIFO**). Parenthesis matching, Infix to postfix conversion (and other similar conversions) will be dealt with in future.

Stack ADT:

In order to create a stack, we need a pointer to the topmost element to gain knowledge about the element which is on the top so that any operation can be carried about. Along with that, we need the space for the other elements to get in and their data.

Operations in Stack:



push( ) to push an element into the stack.

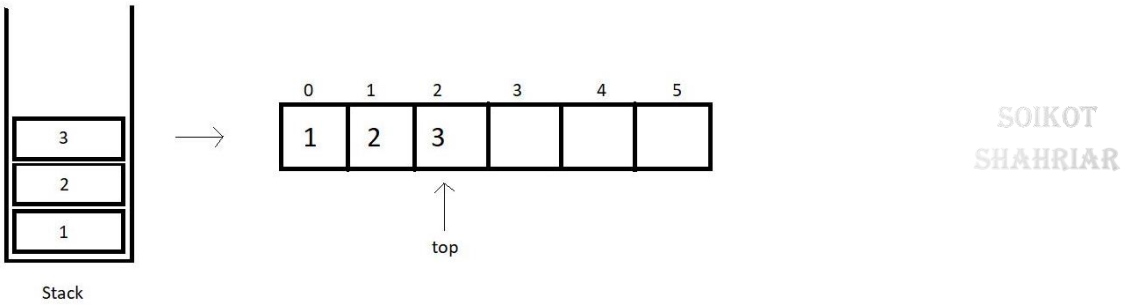
pop( ) to remove the topmost element from the stack.

peek(index) to return the value at a given index.

isEmpty( ) / isFull( ) to determine whether the stack is empty or full to carry efficient push and pull operations.

Implementation of Stack Using Array:

To implement a stack using an array, we'll maintain a variable that will store the index of the top element.



So, basically, we have few things to keep in check when we implement stacks using arrays.

a. A fixed-size array. This size can even be bigger than the size of the stack we are trying to implement, to stay on the safe side.
b. An integer variable to store the index of the top element, or the last element we entered in the array. This value is -1 when there is no element in the array.

We will try **constructing a structure** to embed all these functionalities. Let's see how:

```
struct stack{
    int size;
    int top;
    int* arr;
}
```

So, the struct above includes as its members, the size of the array, the index of the top element, and the pointer to the array we will make.

To use this struct,

- We will just have to declare a struct stack.
- Set its top element to -1.
- Furthermore, we will have to reserve memory in the heap using malloc.

Example for defining a Stack:


```
struct stack s;
s.size = 100;
s.top = -1;
s.arr = (int*)malloc(s.size*sizeof(int));

struct stack *s;
s->size = 100;
s->top = -1;
s->arr = (int *)malloc(s->size * sizeof(int));
```

We have used an integer array above, although it is just for the sake of simplicity. We have the freedom to customize our data types according to our needs.

Operations in Stack:

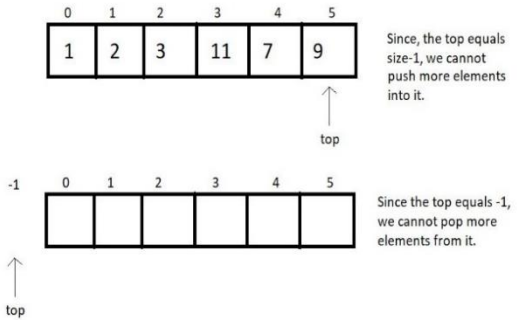
Two Important Points:

1. One cannot push more elements into a full stack.
2. One cannot pop any more elements from an empty stack.

push( ) Operation:

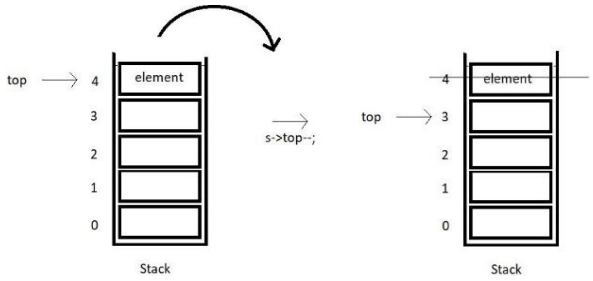
a. The first thing is to define a stack. Suppose we have the creating stack and declaring its fundamental part done, then pushing an element requires to first check if there is any space left in the stack.
b. Then call the isFull function. If it's full, then we cannot push anymore elements. This is the case of stack overflow. Otherwise, increase the variable top by 1 and insert the element at the index top of the stack.

By pushing, we mean inserting an element at the top of the stack. While using the arrays, we have the index to the top element of the array. So, we'll just insert the new element at the index (top+1) and increase the top by 1. This operation takes a constant time, O(1). It's intuitive to note that this operation is valid until (top+1) is a valid index and the array has an empty space.



pop( ) Operation:

Pop means to remove the last element entered in the stack, and that element has the index top.

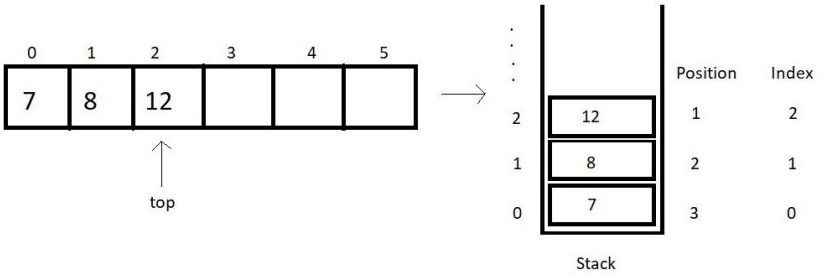


We'll just have to decrease the value of the top by 1, and we are done. The popped element can even be used in any way we like.

a. The first thing again is to define a stack. Then popping an element requires to first check if there is any element left in the stack to pop.
b. Call the isEmpty function. If it's empty, then we cannot pop any element, since there is none left. This is the case of stack underflow. Otherwise, store the topmost element in a temporary variable. Decrease the variable top by 1 and return the temporary variable which stored the popped element.

peek( ) Operation:

Peek operation requires the user to give a position to peek at as well. Here, position refers to the distance of the current index from the top element +1. Visualize this via illustrations:



The **Index**, mathematically, is **(top -position+1)**

So, before we return the element at the asked position, we'll check if the position asked is valid for the current stack. Index 0, 1 and 2 are valid for the stack illustrated above, but index 4 or 5 or any other negative index is invalid. peek(1) returns 12 here.

Time Complexities of Operations:

isEmpty( ): This operation just checks if the top member equals -1. This works in a constant time, hence, O(1).

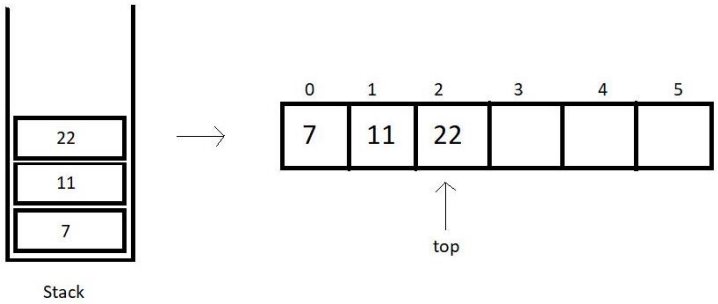
isFull( ): This operation just checks if the top member equals size -1. Even this works in a constant time, hence, O(1).

push( ): Pushing an element in a stack needs us to just increase the value of top by 1 and insert the element at the index. This is again a case of O(1).

pop( ): Popping an element in a stack needs us to just decrease the value of top by 1 and return the element we ignored. This is again a case of O(1).

peek( ): Peeking at a position just returns the element at the index, (top - position + 1), which happens to work in a constant time. So, even this is an example of O(1).

So, basically all the operations we discussed follow a constant time complexity.



stackTop( ):

This operation is responsible for returning the topmost element in a stack. Retrieving the topmost element was never a big deal. We can just use the stack member top to fetch the topmost index and its corresponding element.

Here, in the illustration above, we have the top member at index 2. So, the stackTop operation shall return the value 22.

stackBottom( ):

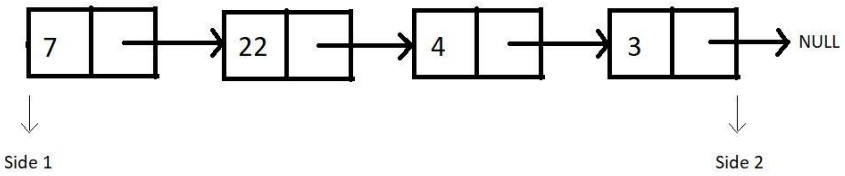
This operation is responsible for returning the bottommost element in a stack, which intuitively, is the element at index 0. We can just fetch the bottommost index, which is 0, and return the corresponding element.

Here, in the illustration above, we have the bottommost element at index 0. So, the stackBottom operation shall return the value 7.

One thing one must observe here is that both these operations happen to work in a constant runtime, that is O(1). Because we are just accessing an element at an index, and that works in a constant time in an array.

Implementation of Stack Using Linked List:

We can consider a singly linked list, following the illustration below.

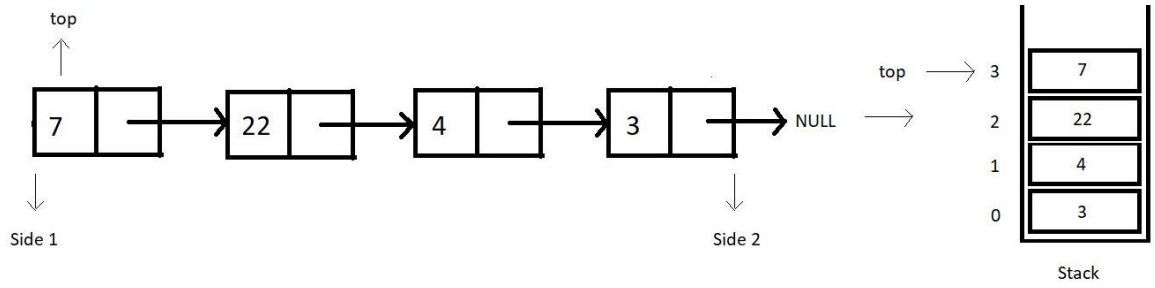


Consider this linked list functioning as a stack. And as we know, we have two sides of a linked list, one the head, and the other pointing to NULL.

Which side we consider as the top of the stack, where we push and pop from? Head Side. And why the head side, that is side 1?

Because that's the head node of the linked list, and insertion and deletion of a node at head happens to function in a constant time complexity, O(1). Whereas inserting or deleting a node at the last position takes a linear time complexity, O(n).

So that stack equivalent of the above illustrated linked list looks something like this:



Stack Empty or Full:

a. Stacks when implemented with linked lists never get full. We can always add a node to it. There is no limit on the number of nodes a linked list can contain until we have some space in heap memory.
b. Whereas stacks become empty when there is no node in the linked list, hence when the top equals to NULL.

The head node we had in linked lists, is the top for our stacks now. So, from now on, the head node will be referred to as the top node.

Even though a stack linked list has no upper limit to its size, we can always set a custom size for it.

Conditions:

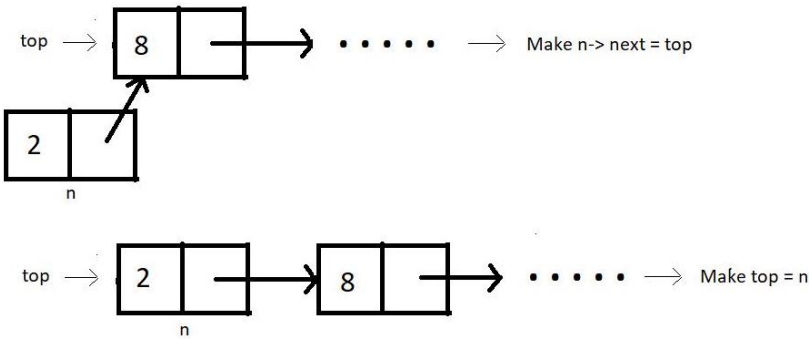
Stack Full: When heap memory is exhausted.

Stack Empty: top == NULL

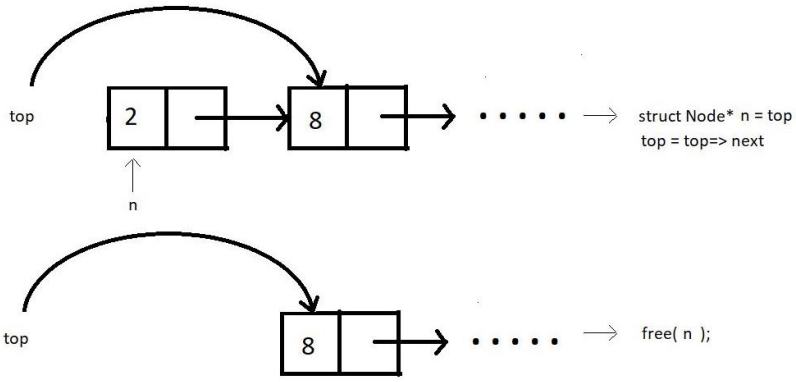
Implementing all the Stack Operations using Linked List:

Before writing the codes, we must discuss the algorithm we'll put into operations. Let's go through them one by one.

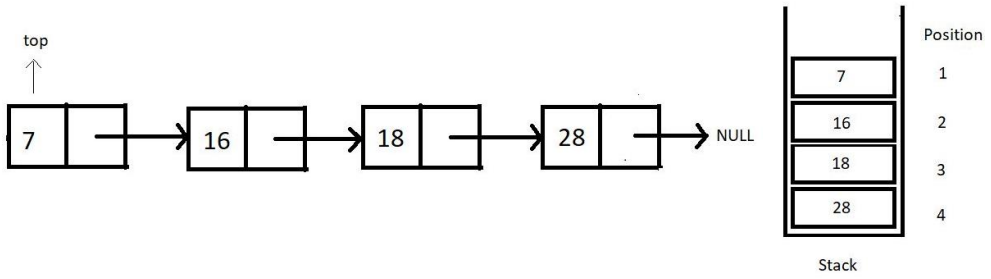
1. **isEmpty:** It just checks if our top element is NULL.
2. **isFull:** A stack is full, only if no more nodes are being created using malloc. This is the condition where heap memory gets exhausted.
3. **Push:** The first thing we need before pushing an element is to create a new node. Check if the stack is not already full. Now, we follow the same concept we learnt while inserting an element at the head or at the index 0 in a linked list. Just set the address of the current top in the next member of the new node, and update the top element with this new node.



4. **Pop:** First thing is to check if the stack is not already empty. Now, we follow the same concept we learnt while deleting an element at the head or at the index 0 in a linked list. Just update the top pointer with the next node, skipping the current top.



5. **Peek:** This operation is meant to return the element at a given position. Do mind that the position of an element is not the same as the index of an element. In fact, there is nothing as an index in a linked list. Refer to the illustration below.



Peeking in a stack linked list is not as efficient as when we worked with arrays. Peeking in a linked list takes O(n) because it first traverses to the position where we want to peek in. So, we'll just have to move to that node and return its data.

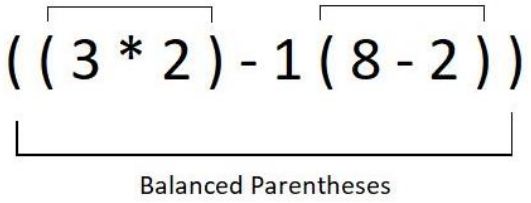
6. **stackTop:** This operation just returns the topmost value in the stack. That is, it just returns the data member of the top pointer.

7. **stackBottom:** This operation just returns the bottommost value in the stack.

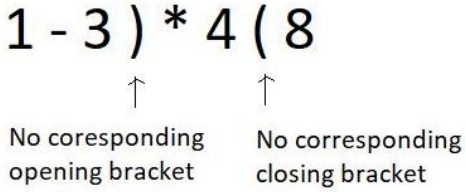
Parenthesis Matching using Stack:

Learning mathematics in school, we had BODMAS there, which required us to solve the expressions, first enclosed by brackets, and then the independent ones. That's the bracket we're referring to. We have to see if the given expression has balanced brackets which means every opening bracket must have a corresponding closing bracket and vice versa.

Given illustrations would surely make it clear.



Balanced Parentheses



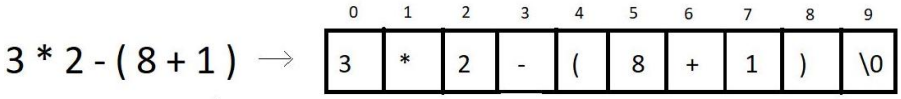
Unbalanced Parentheses

Checking if the parentheses are balanced or not must be a cakewalk for humans, since we have been dealing with this for the whole time. But even we would fail if the expression becomes too large with a great number of parentheses.

This is where automating the process helps. And for automation, we need a proper working algorithm. We will see how we accomplish that together.

We'll use **stacks to match these parentheses**. Let's see how:

1. Assume the expression given as a character array.



2. Iterate through the character array and ignore everything we find other than the opening and the closing parenthesis. Every time we find an opening parenthesis, push it inside a character stack. And every time we find a closing parenthesis, pop from the stack, in which we pushed the opening bracket.

Conditions for Unbalanced Parentheses:

- When we find a closing parenthesis and try achieving the pop operation on the stack, the stack must not become underflow. To match the existing closing parenthesis, at least one opening bracket should be available to pop. If there is no opening bracket inside the stack to pop, we say the expression has unbalanced parentheses.

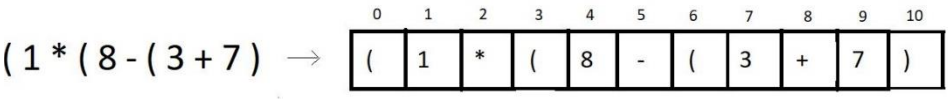
  For example: the expression (2+3)*6)1+5 has no opening bracket corresponding to the last closing bracket. Hence unbalanced.

- At EOE, that is, when we reach the end of the expression, and there is still one or more opening brackets left in the stack, and it is not empty, we call these parentheses unbalanced.

  For example: the expression (2+3)*6(1+5 has 1 opening bracket left in the stack even after reaching the EOE. Hence unbalanced.

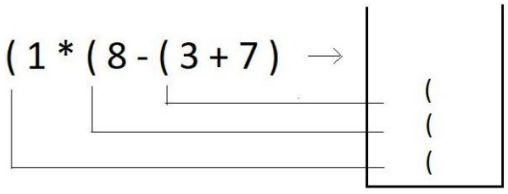
Note: Counting and matching the opening and closing brackets numbers is not enough to conclude if the parentheses are balanced. For eg: 1+3)*6(6+2.

Example:

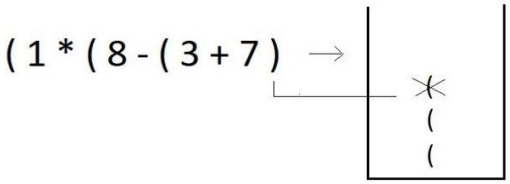


We'll try checking if the above expression has balanced parentheses or not.

Step 1: Iterate through the char array, and push the opening brackets at positions 0, 3, 6 inside the stack.



Step 2: Try popping an opening bracket from the stack when we encounter a closing bracket in the expression.



Step 3: Since we reached the EOE and there are still two parentheses left in the stack, we declare this expression of parentheses unbalanced.

Note: Parenthesis matching nowhere tells us if the given expression is mathematically valid or not. Because it is not supposed to, this algorithm has been meant just to return whether the parentheses in the expression are balanced or not.

For example, the expression ((8)(*9)) is mathematically invalid but has balanced parentheses.

Multiple Parenthesis Matching using Stack:

In mathematics, we have expressions consisting of three types of parentheses. Now, we will be interested in matching parentheses when all three types of parentheses are used in any expression. This is what we called multi-parenthesis matching. Our focus is mainly on the three types of an opening parenthesis, [{ (and their corresponding closing parentheses,) }].

Modifying what we did earlier to make it work for multi-matching needs very little attention. Just follow these steps:

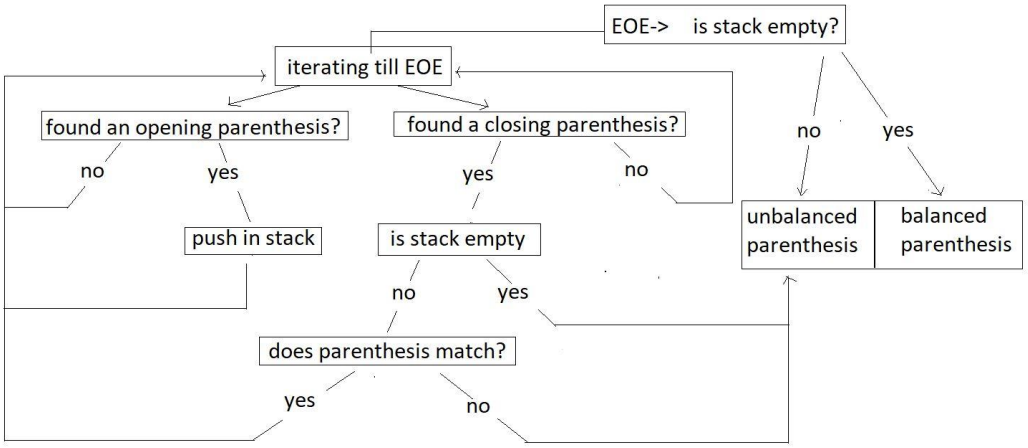
1. Whenever we encounter an opening parenthesis, we simply push it in the stack, similar to what we did earlier.

2. And when we encounter a closing parenthesis, the following conditions should be met to declare its balance:

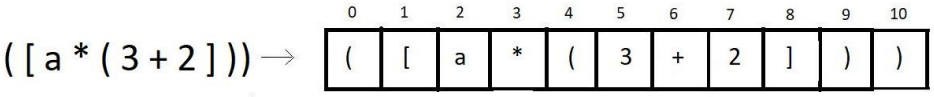
- Before we pop, this size of the stack must not be zero.
- The topmost parenthesis of the stack must match the type of closing parenthesis we encountered.

3. If we find a corresponding opening parenthesis with conditions in point 2 met for every closing parenthesis, and the stack size reduces to zero when we reach EOE, we declare these parentheses, matching or balanced. Otherwise not matching or unbalanced.

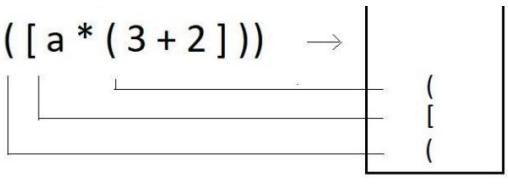
So, basically, we modified the pop operation. And that's all. Let's see what additions to the code we would like to make. But before that follow the illustration below to get a better understanding of the algorithm.



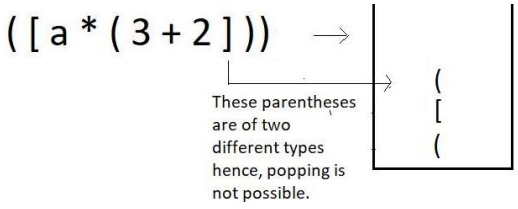
Example: We'll try checking if the given expression has balanced multi-parentheses or not.



Step 1: Iterate through the char array, and push the opening brackets of all types at positions 0, 1, 4 inside the stack.



Step 2: When we encounter a closing bracket of any type in the expression, try checking if the kind of closing bracket we have got matches with the topmost bracket in the stack.



Step 3: Since we couldn't pop an opening bracket corresponding to a closed bracket, we would just end the program here, declaring the parentheses unbalanced.

Infix, Prefix and Postfix Expressions:

The three terms, infix prefix, and postfix will be dealt with individually later. In general, these are the notations to write an expression. Mathematical expressions have been taught to us since childhood. Writing expressions to add two numbers for subtraction, multiplication, or division. They were all expressed through certain expressions. That's what we're learning today: different expressions.

Infix: < operand 1 >< operator >< operand 2 >

This is the method we have all been studying and applying for all our academic life. Here the operator comes in between two operands. And we say, two is added to three. For e.g.: 2 + 3, a * b, 6 / 3 etc.

Prefix: < operator >< operand 1 >< operand 2 >

Here the operator comes before the two operands. And we say, Add two and three. For e.g.: + 6 8, * x y, - 3 2 etc.

Postfix: < operand 1 >< operand2 >< operator >

Here the operator comes after the two operands. And we say, Two and three are added. For e.g.: 5 7 +, a b *, 12 6 / etc.

So far, we have been dealing with just two operands, but a mathematical expression can hold a lot more. We will now learn to change a general infix mathematical expression to its prefix and postfix relatives. But before that, it is better to understand why we even need these methods.

Why these Methods?

When we evaluate a mathematical expression, we have a rule in mind, named BODMAS, where we have operators' precedence in this order; brackets, of, division, multiplication, addition, subtraction.

But what would we do when we get to evaluate a 1000 character long-expression, or even longer one? We will try to automate the process. But there is one issue. Computers don't follow BODMAS; rather, they have their own operator precedence. And this is where we need these postfix and prefix notations. In programming, we use **postfix** notations more often, likewise, following the precedence order of machines.

Converting Infix to Prefix:

Consider the expression, x - y * z

1. Parentheses the expression. The infix expression must be parenthesized by following the operator precedence and associativity before converting it into a prefix expression. Our expression now becomes (x - (y * z))

2. Reach out to the innermost parentheses. And convert them into prefix first, i.e. (x - (y * z)) changes to (x - [* y z]

3. Similarly, keep converting one by one, from the innermost to the outer parentheses. (x - [* y z]) → [- x * y z]

Converting Infix to Postfix:

Consider the same expression, x - y * z

1. Parentheses the expression as we did previously. Our expression now becomes (x - (y * z))

2. Reach out to the innermost parentheses. And convert them into postfix first, i.e. (x - (y * z)) changes to (x - [y z *])

3. Similarly, keep converting one by one, from the innermost to the outer parentheses. (x - [y z *]) → [x y z * -]

Infix To Postfix using Stack:

Converting an infix expression to its postfix counterpart needs us to follow certain steps. The following are the steps:

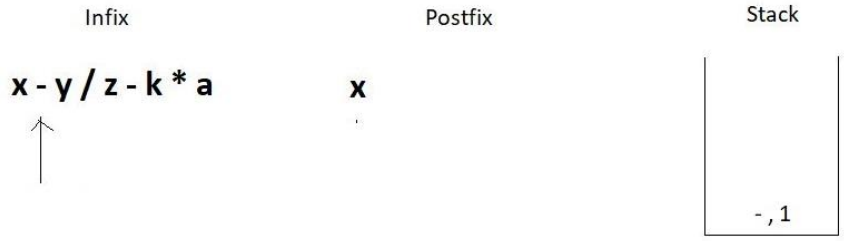
1. Start moving left to right from the beginning of the expression.
2. The moment we receive an operand, concatenate it to the postfix expression string.
3. And the moment we encounter an operator, move to the stack along with its relative precedence number and see if the topmost operator in the stack has higher or lower precedence. If it's lower, push this operator inside the stack. Else, keep popping operators from the stack and concatenate it to the postfix expression until the topmost operator becomes weaker in precedence relative to the current operator.

4. If we reach the EOE, pop every element from the stack, and concatenate them as well. And the expression we will receive after doing all the steps will be the postfix equivalent of the expression we were given.

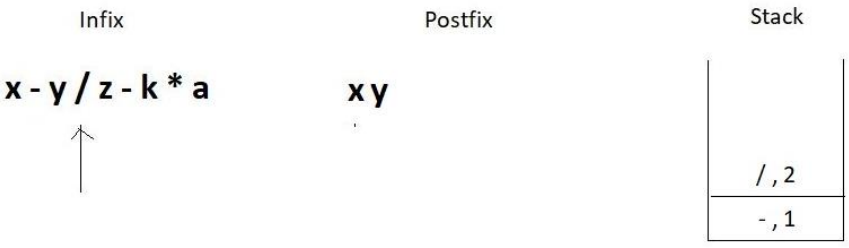
For our understanding today, let us consider the expression x - y / z - k * a. Step by step, we will turn this expression into its **postfix equivalent using stacks.**

1. We will start traversing from the left.

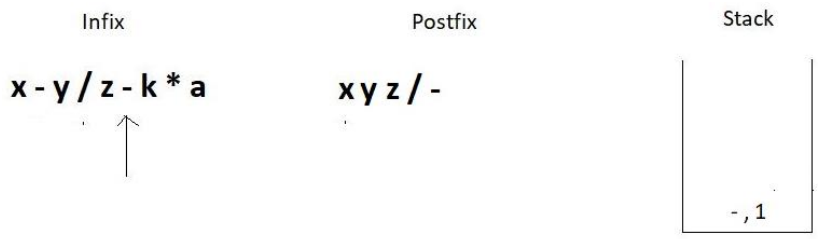
2. First, we got the letter ‘x’. We just pushed it into the postfix string. Then we got the subtraction symbol ‘-’, and we push it into the stack since the stack is empty.



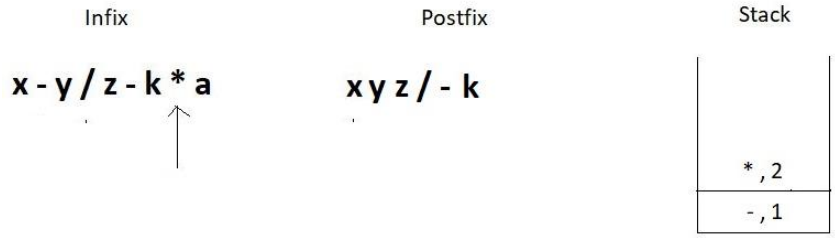
3. Similarly, we push the division operator in the stack since the topmost operator has a precedence number 1, and the division has 2.



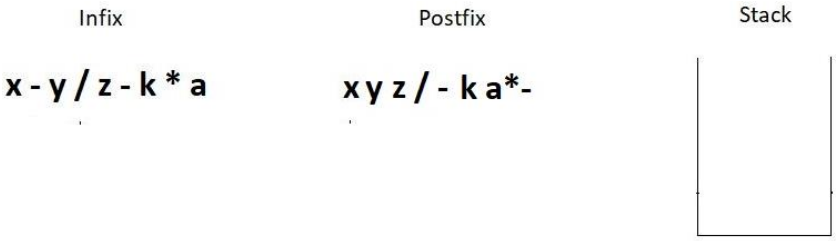
4. The next operator we encounter is again a subtraction. Since the topmost operator in the stack has an operator precedence number 2, we would pop elements out from the stack until we can push the current operator. This leads to removing both the present operators in the stack since they are both greater or equal in precedence. Don’t forget to concatenate the popped operators to the postfix expression.



5. Next, we have a multiplication operator whose precedence number is 2 relative to the topmost operator in the stack. Hence, we simply push it in the stack.



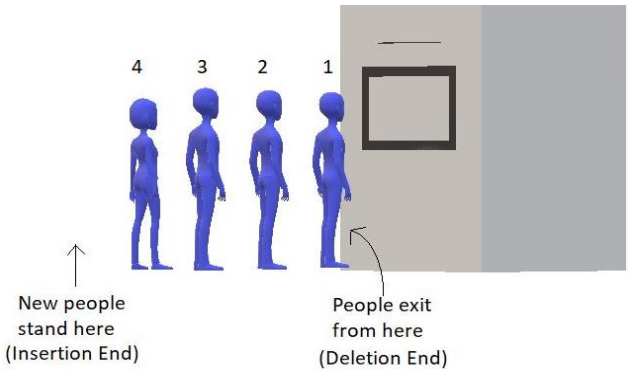
6. And then we get to the EOE and still have two elements inside the stack. So, just pop them one by one, and concatenate them to the postfix. And this is when we succeed in converting the infix to the postfix expression.



Queue

Unlike stacks, where we followed LIFO (Last In First Out) discipline, here in the queue, we have **FIFO (First In First Out).** Follow the illustration below to get a visual understanding of a queue.

In stacks, we had to maintain just one end, head, where both insertion and deletion used to take place, and the other end was closed. But here, in queues, we have to maintain both the ends because we have insertion at one end and deletion from the other

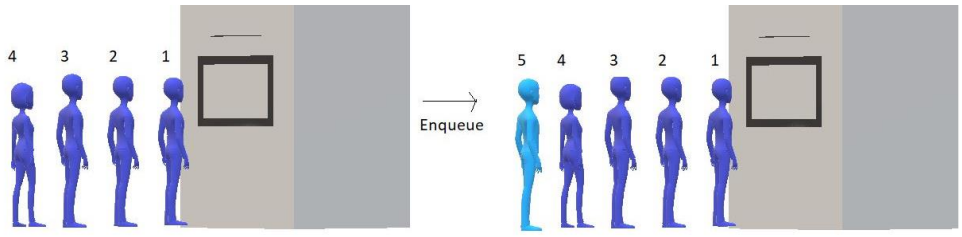


Queue ADT:

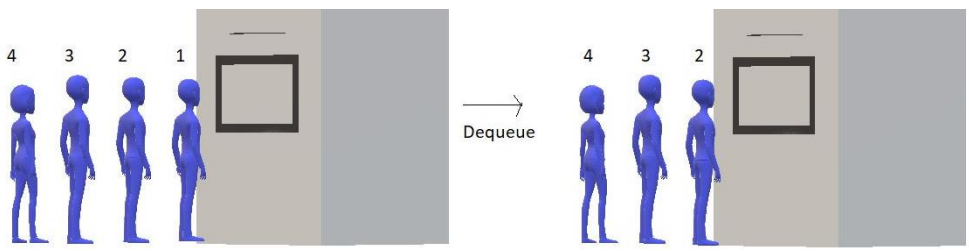
Data: In order to create a queue, we need two pointers, one pointing to the insertion end, to gain knowledge about the address where the new element will be inserted to. And the other pointer pointing to the deletion end, which holds the address of the element which will be deleted first. Along with that, we need the storage to hold the element itself.

Methods: Here are some of the basic methods we would want to have in queues:

1. enqueue() : to insert an element in a queue.



2. dequeue() : to remove an element from the queue



3. firstVal(): to return the value which is at the first position.

4. lastVal(): to return the value which is at the last position.

5. peek(position): to return the element at some specific position.

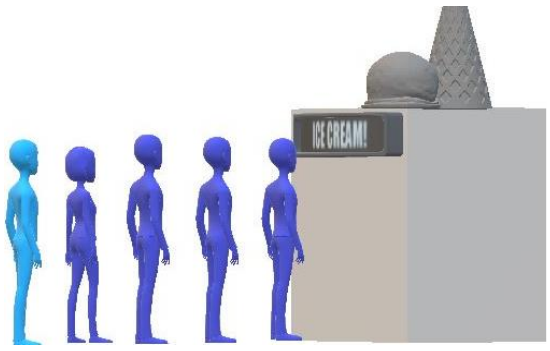
6. isempty() / isfull(): to determine whether the queue is empty or full, which helps us carry out efficient enqueue and dequeue operations.

This was our abstract data type, Queue.

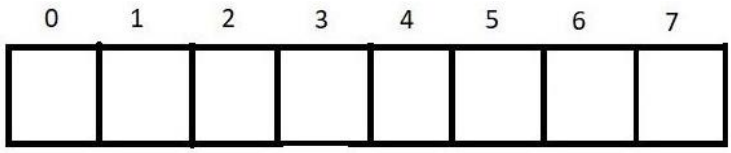
A queue can be implemented in a number of ways. We can use both an array and a linked list and even a stack, and not just that, but by any ADT. We'll see all these methods in the coming tutorials.

Implementation of Queue using Array:

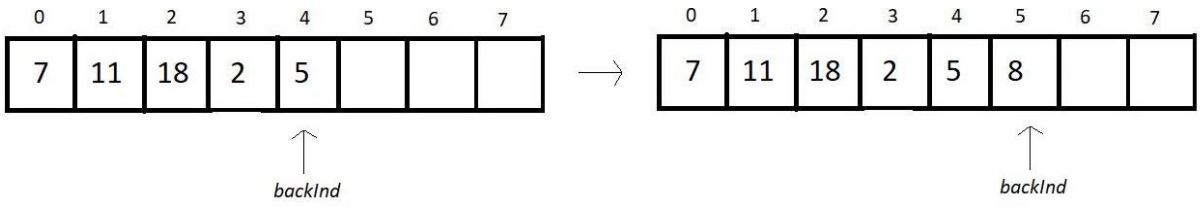
It is analogous to a queue in front of any ticket counter or an ice cream shop illustrated here. Here, we have shown a branded ice cream shop that is famous enough to have a queue of people waiting to get one of their choices.



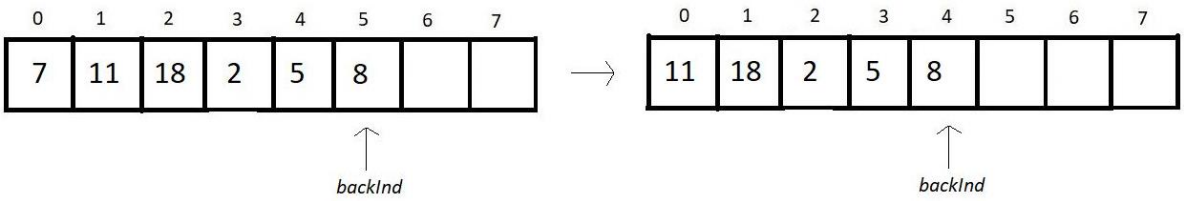
The shop owner wants to store the information of these people, so he uses an array to accomplish that. Assuming that we have 8 people and we want to store their information, we'll have an array as illustrated below:



Here, we'll maintain an index variable, *backInd*, to store the index of the rearmost element. So, when we insert an element, we just increment the value of the *backInd* and insert the element at the current *backInd* value. Follow the array below to know how inserting works:



Now suppose we want to remove an element from the queue. And since a queue follows the FIFO discipline, we can only remove the element at the zeroth index, as that is the element inserted first in the queue. So, now we will remove the element at the zeroth index and shift all the elements to its adjacent left. Follow the illustrations below:



But this removal of the zeroth element and shifting of other elements to their immediate left features O(n) time complexity.

Summing up this method of enqueue and dequeue, we can say:

1. **Insertion (enqueue):**

- Increment backInd by 1
- Insert the element
- Time complexity: O(1)

2. **Deletion (dequeue):**

- Remove the element at the zeroth index
- Shift all other elements to their immediate left
- Decrement backInd by 1

3. Here, our first element is at index 0, and the rearmost element is at index backInd.

4. Condition for queue empty: backInd = -1.

5. Condition for queue full: backInd = size-1.

❖ **Better** way to accomplish these tasks:

We can use another index variable called frontInd, which stores the index of the cell just before the first element. We'll maintain both these indices to bring about all our operations.

Let's now enlist the changes we'll see after we introduce this new variable:

1. **Insertion (enqueue):**

- Increment backInd by 1
- Insert the element
- Time complexity: O(1)

2. **Deletion (dequeue):**

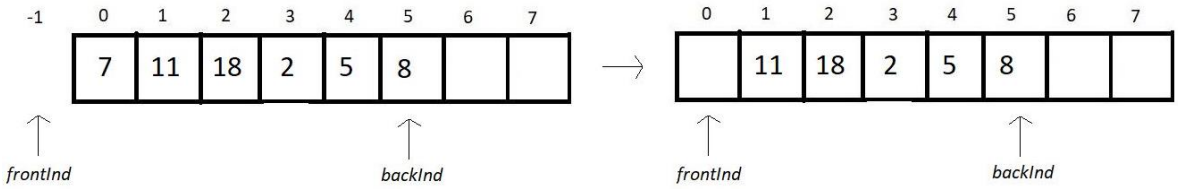
- Remove the element at the zeroth index (no need for that in an array)
- Increment frontInd by 1
- Time complexity: O(1)

3. Our first element is at index frontInd +1, and the rearmost element is at index backInd.

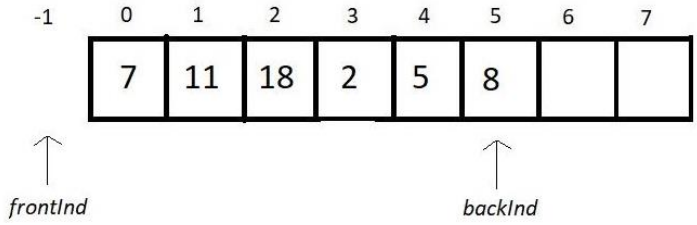
4. Condition for queue empty: frontInd = backInd.

5. Condition for queue full: backInd = size-1.

Now, we were able to achieve both operations in constant run time. And the new dequeue operation goes as follow:



Array Implementation of Queue and it's Operations:



To implement this Queue, we'll use a **structure** and have the following members inside it:

a. size: to store the size of the array
b. frontInd: to store the index prior to the first element.
c. backInd: to store the index of the rearmost element.
d. *arr: to store the address of the array dynamically allocated in heap.

Now to use this struct element as a queue, we need to **initialize** its instances as:

1. struct Queue q; (we are not dynamically allocating q here for now, as we did in stacks).
2. Use dot here, and not arrow operator to assign values to struct members, since q is not a pointer.
3. q.size = 10; (this gives size element the value 10)
4. q.frontInd = q.backInd = -1;(this gives both the indices element the value -1)
5. Use malloc to assign memory to the arr element of struct q.

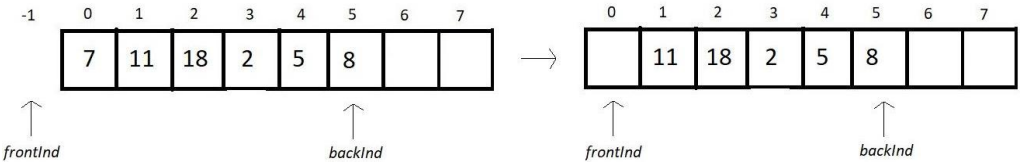
And this is how we initialize a queue. We will now devote our attention to two important operations in a queue: enqueue and dequeue.

Enqueue: Enqueuing is inserting a new element in a queue. Prior to inserting an element into a queue, we need to take note of a few points.

- First, check if the queue is already not full.
- If it is, it is the case of queue overflow, else just increment backInd by 1, insert the new element there. Follow the illustration below.

Dequeue: Dequeuing is deleting the element in a queue which is the first among all the elements to get inserted. Prior to deleting that element from a queue, we need to follow the below-mentioned points:

- First, check if the queue is already not empty.
- If it is, it is the case of queue underflow, else just increment frontInd by 1. In arrays, we don't delete elements; we just stop referring to the element. Follow the illustration below.



Condition for isEmpty:

If our frontInd equals backInd, then there is no element in our queue, and this is the case of an empty queue.

Condition for isFull:

If our backInd equals (the size of the array) -1, then there is no space left in our queue, and this is the case of a full queue.

Circular Queue:

When we discussed queues, we decided to have two index variables f and r, which would maintain the two ends of the queue. If we follow the illustration below, we would see that our queue gets full when element 8 is pushed in the queue. In other words, we can only enqueue in a queue until the queue isn't full.

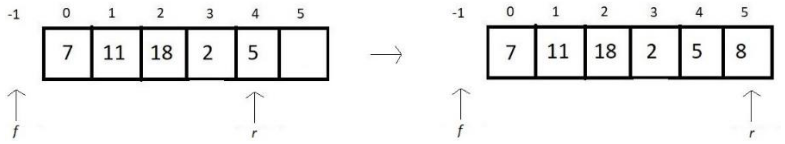


Figure 1: Using two integer variables to maintain the ends of a queue

Now, we start dequeuing some elements. Let's remove the first three elements. And now, if we carefully observe, our queue is still full since the rear end is at the array's threshold. But technically, it has space worth three elements left. And this is one characteristic cum drawback of a linear/normal queue when implemented using arrays. We don't get to efficiently utilize the space acquired by the array in the heap. Here the remaining three spaces remain unused for the whole time.

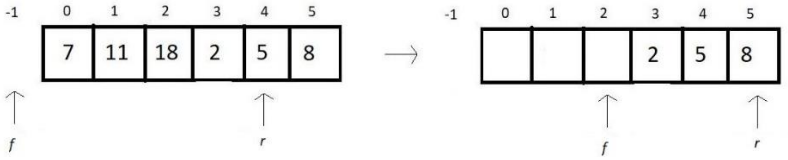


Figure 2: Dequeuing leaves vacant spaces behind

When we talk about utilizing these spaces rather than letting them go unused, we introduce circular queues.

Let's now see how we can eliminate this drawback and what modifications this situation calls for.

One optimizing call would be to reset f and r to -1 whenever the queue becomes empty, or in other words, they both become equal. This makes all the space in the array reusable. Here, the queue was full since r equals the size - 1 of the array. But resetting both the index variables to -1 empties the queue.

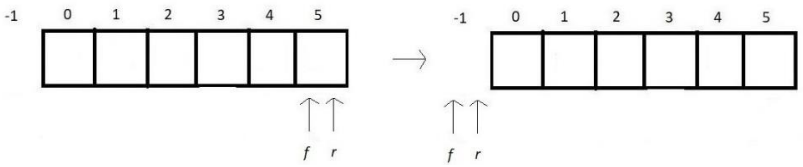


Figure 3: Resetting the index variables to -1

However, the efficiency of this method is limited by the requirement that the front and rear be the same. It is ineffective in the case of figure 2. Therefore, we need a more optimized solution. This is when circular queues come to the rescue.

Circular Queues:

In circular queues, we mainly focus on the point that we don't increment our indices linearly. Linearly increasing indices cause the case of overflow when our index reaches the limit, which is size-1.

In linear increment, i becomes i+1.

But in a circular increment; i becomes **(i+1)%size**. This gives an upper cap to the maximum value making the index repeat itself.

Linear Increment: 0 1 2 3 4 5 6 7 8 9 . . . .

Circular Increment: 0 1 2 3 4 0 1 2 3 4 . . . . .
(let the size be 5)

And this makes us start from the beginning once we reach the threshold of the array. Refer to the illustration below to visualize the movement of the cursor.

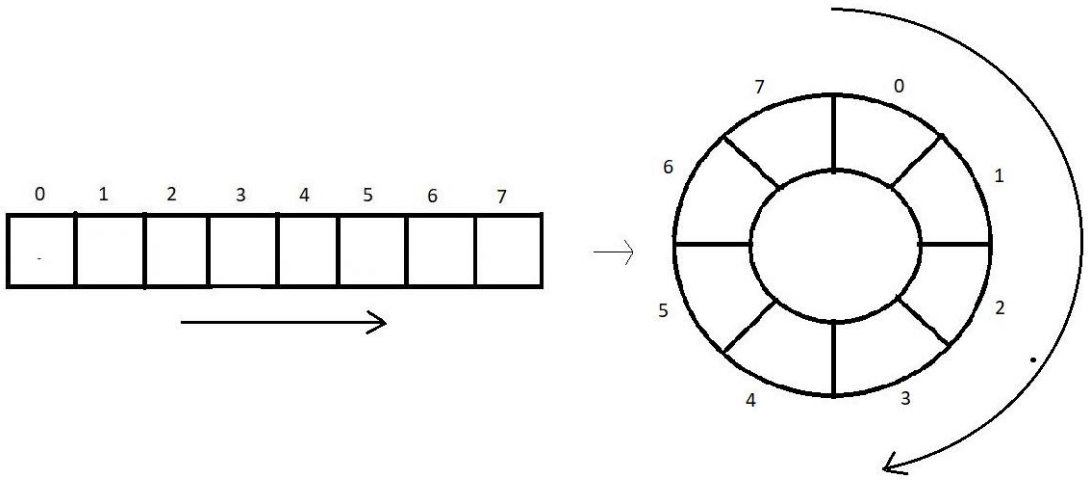


Figure 4: Conversion of a linear queue to a circular queue

And this is the circular implementation of the same array we used to implement linearly. This allows the leftover spaces to be used again. This wheel type array is called the circular queue.

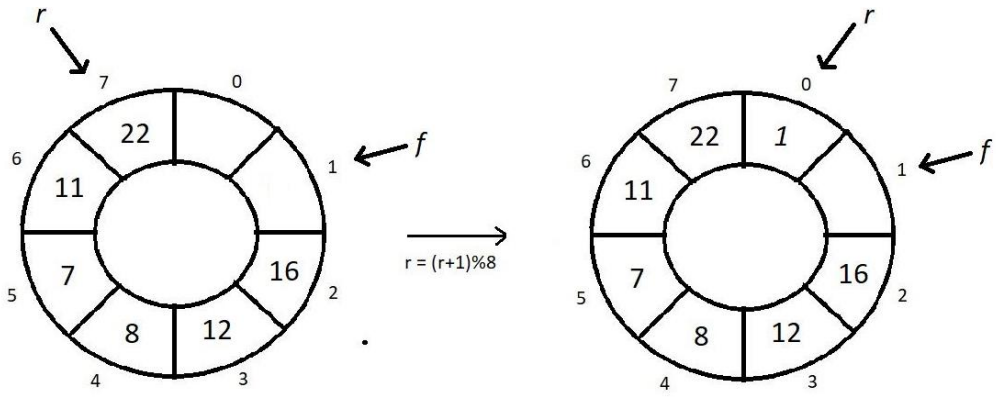
Operations on Circular Queue:

Condition for isEmpty: If our f equals r, then there is no element in our queue, and this is the case of an empty queue.

Condition for isFull: If our (r+1)%size equals f, then there is no space left in our queue, and this is the case of a full queue.

Enqueue: Inserting a new element in a queue requires the user to input a value that we would pass into the queue function. Before inserting,

a. First, check if the queue is already not full. Here, the usual method to check the full condition wouldn't work. We will now check if the next index to the rear is whether the front or not.
b. If it is, it means the queue is full. Because front f represents the starting of the queue, and rear r represents the end. And the front coming next to the rear indicates that the queue is full. Therefore, this is the case of queue overflow. Else just increment the rear by 1 and take its modulus by the queue's size. This is called the circular increment. Insert the new element there. Follow the illustration below.

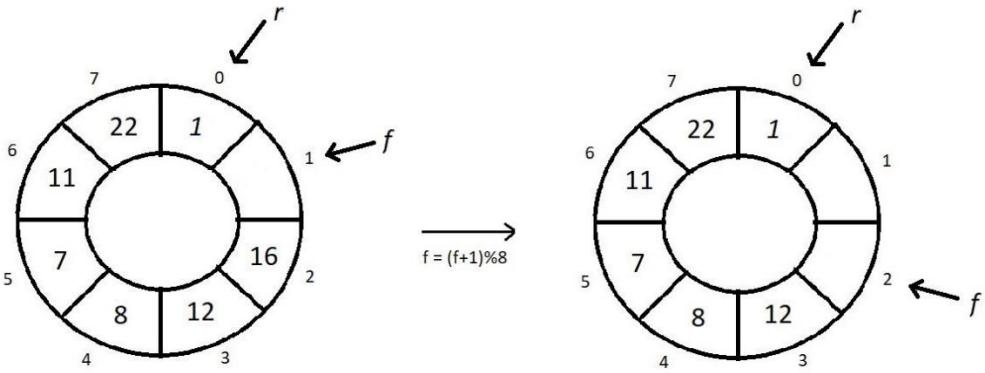


Now, since the f is just next to the r, the queue is full, and no more elements can get pushed.

Dequeue: Dequeuing is deleting the element in a queue which is the first among all the elements to get inserted. And since the front f holds the index of that element, we can just remove that. But before doing that,

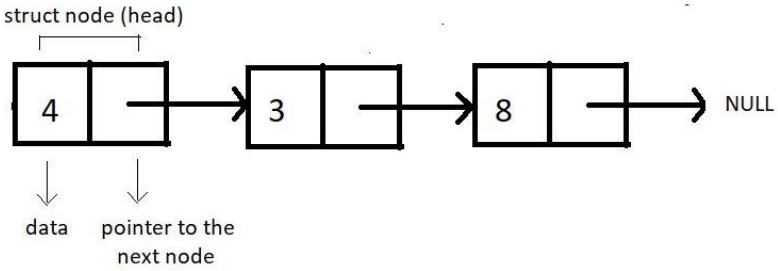
a. First, check if the queue is already not empty. Previously, we would just check if our front equals the rear, and if it did, we declared the queue empty. And we'll be amazed to know that it works here as well. There are zero modifications here.

b. So, if the front f equals the rear r, it is the case of queue underflow, else just increment f by 1 and take its modulus by the queue's size. While dequeuing, we store the element being removed and return it at the end. Follow the illustration below.

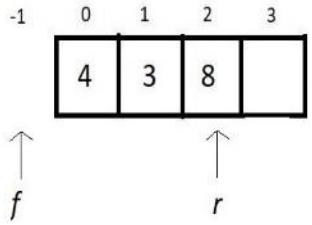


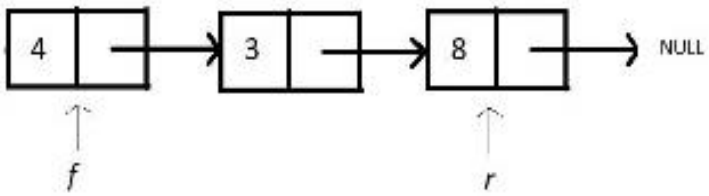
Queue using Linked Lists:

If we remember, linked lists are chain-like structures with nodes having two parts, an integer variable to hold data and another node pointer to hold the address of the next node. Below illustrated is a linked list with three nodes. The last node points to NULL. And the first node is called the head.



We maintained two index variables, f, and r, to mark the beginning and the end of the queue. Below illustrated is a queue with three elements.





Since we are implementing this queue using a linked list, the index variables are no longer integers. These become the pointers to the front and the rear nodes. And the queue somewhat starts looking like this.

Condition for isEmpty:

The only condition for the queue linked list to be empty is that the f node is NULL, which means there is no beginning, hence no element.

Condition for isFull:

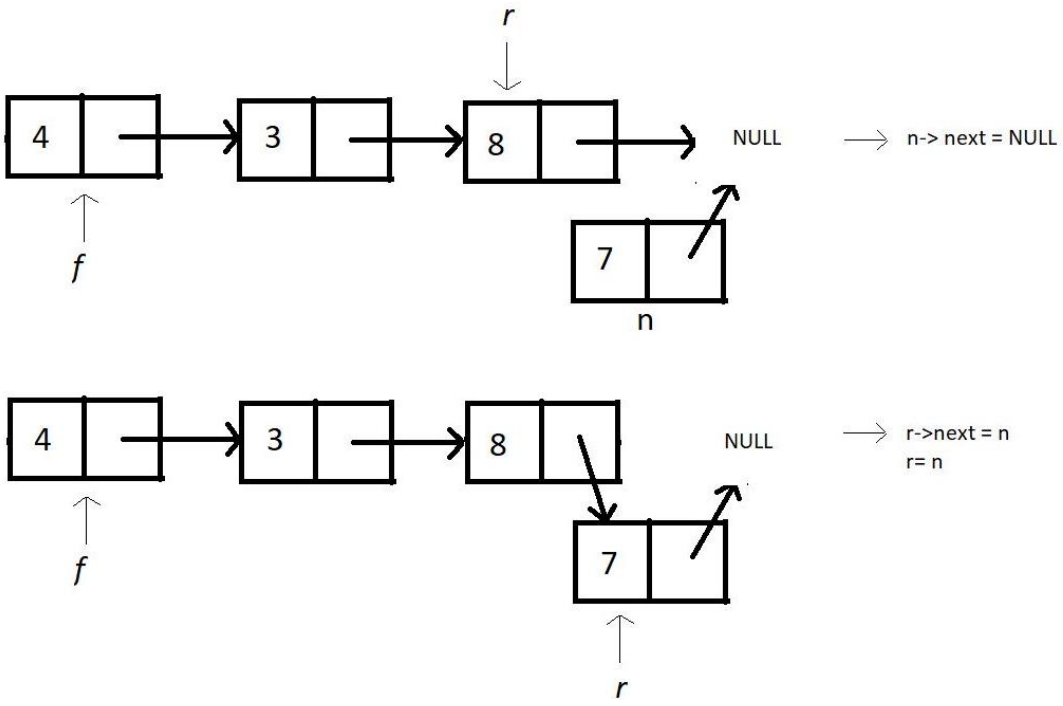
Queues implemented using linked lists never usually become full until the space in the heap memory is exhausted.

Therefore, the only condition for the queue linked list to be full is that the new node becomes NULL when created.

Enqueue in a Queue linked list:

Enqueuing in a queue linked list is very much similar to just inserting at the end in a linked list. Inserting a new node at the end requires us to follow few steps:

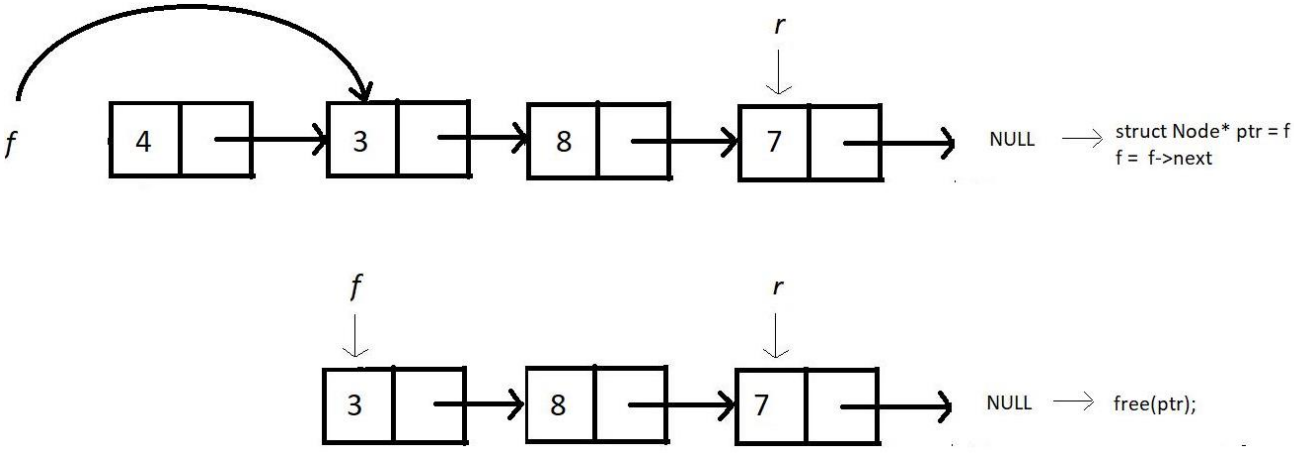
a. Check if there is a space left in the heap for a new node.
b. If there is, create a new node n, assign it memory in heap, and fill its data with the new value the user has given.
c. Point the next member of this new node n to NULL, and point the next member of the r to n. And make r equal to n. And we are done.
d. There is one exception here. When we insert the first element, both f and r are pointing to NULL. So, instead of just making r equal to n, we make f equal to n as well. This marks the beginning of the list.



Dequeue in a Queue linked list:

Dequeuing in a queue linked list is very much similar to deleting the head node in a linked list. Deleting the head node from the list requires us to follow few steps:

a. Check if the queue list is already not empty using the isEmpty function.
b. If it is, return -1. Else create a new node ptr and make it equal to the f node. And don't forget to store the data of the f node in some integer variable.
c. Make the f equal to the next member of f, and free the node ptr. Return the value we stored.



Double-Ended Queue (DE-Queue):

DE-Queue **don't follow FIFO**. As the name suggests, this variant of the queue is double-ended. This means that unlike normal queues where insertion could only happen at the rear end, and deletion at the front end, these double-ended queues have the freedom to insert and delete elements from the end of their choice.

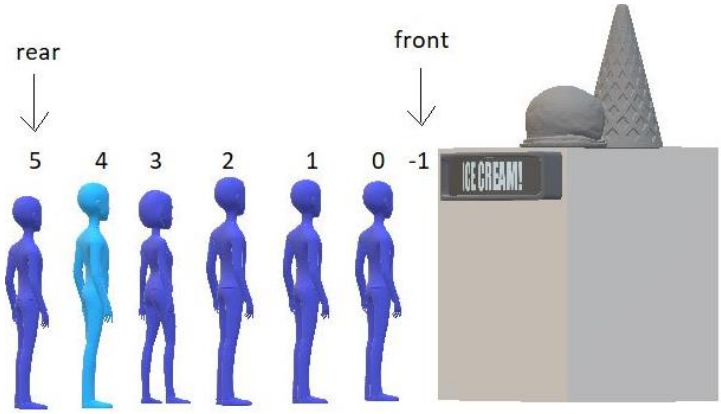
Double-ended queues, hence, have the following characteristics:

- They don't follow the FIFO discipline.
- Insertion can be done at both the ends of the queue.
- Deletion can also be done from both ends of the queue.

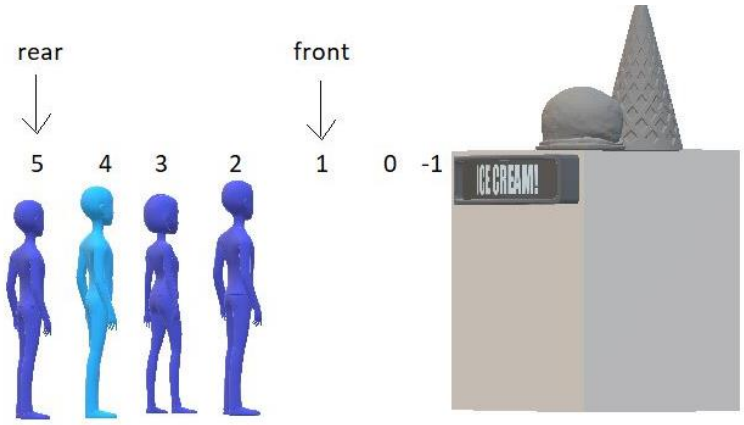
We would assume the implementation part of double-ended queues to be on the tough side, but it is straightforward to consume. We'll use illustrations to make us understand things better.

Insertion in a DE-Queue:

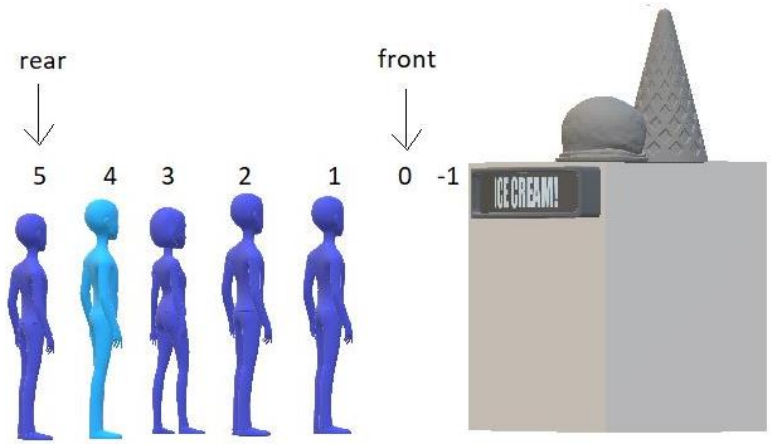
Insertion in a DE-Queue is very intuitive. Follow the illustration below:



Now since the front has no space to insert, we can only insert at the rear end. But if the front manages to have some space after some dequeuing, then our condition would be something like this:

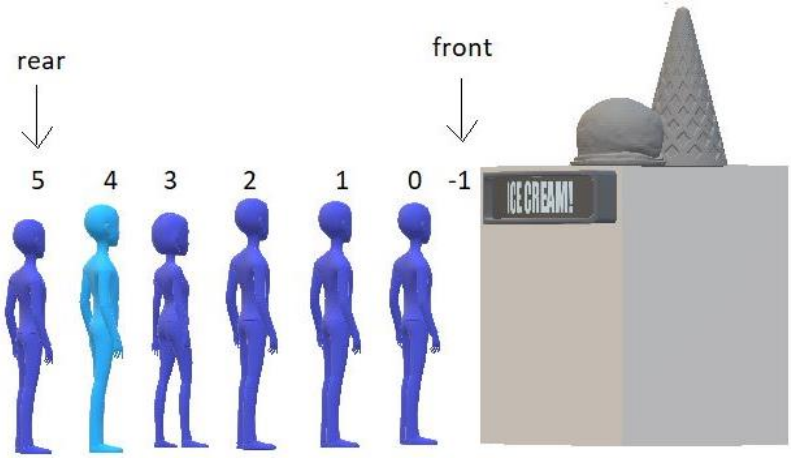


Now, we have 2 places to fill in front as well. And in DE-Queue, we have no restrictions. We would just fill our new element at the front and decrease its value by 1. And that would be it. See the results below:

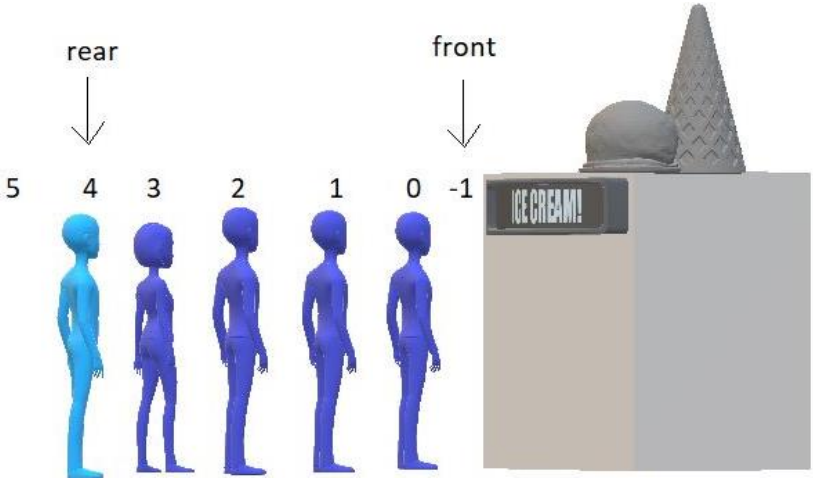


Deletion in a DE-Queue:

Deletion in a DE-Queue is very similar to what we did above. Follow the illustration below:



Now, for one moment, think of the rear as the front end. We would simply then increase the front value by 1 and delete the element at the new front. Similarly, here we first delete the element at rear and decrease the value of the rear by 1. See the results below.



And yeah, we are done deleting the element from the rear end. And inserting at the front end. Moving further, DE-Queues are of two types:

Restricted Input DE-Queue: Input restricted DE-Queues don't allow insertion on the front end. But we can delete from both ends.

Restricted Output DE-Queue: Output restricted DE-Queues don't allow deletion from the rear end. But we can perform the insertion on both the ends.

Sorting Algorithms:

Sorting is a method to arrange a set of elements in either increasing or decreasing order according to some basis or relationship among the elements. Two types of Sorting:

1. Sorting in **Ascending Order**:

Sorting any set of elements in ascending order refers to arranging the elements, let them be numbers, from the smallest to the largest. As example, the set (1, 9, 2, 8, 7), when sorted in ascending order, becomes (1, 2, 7, 8, 9).

2. Sorting in **Descending Order**:

Sorting any set of elements in descending order refers to arranging the elements, let them be numbers, from the largest to the smallest. As example, the same set (1, 9, 2, 8, 7), when sorted in descending order, becomes (9, 8, 7, 2, 1).

Why do we need sorting?

To make you understand the reason why we need sorting in the simplest of ways, we can see some real-life applications of sorting that we might encounter almost daily.

The most useful application is the dictionary. In a dictionary, the words are sorted lexicographically for you to find any word easily.

Criteria for Analysis of Sorting Algorithms:

Time Complexity: Lesser the time complexity, the better is the algorithm.

- We observe the time complexity of an algorithm to see which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets. What if one sorting algorithm sorts only 4 elements efficiently and fails to sort 1000 elements. What if it takes too much time to sort a large data set? These are the cases where we say the time complexity of an algorithm is very poor.
- In general, O(N log N) is considered a better algorithm time complexity than O(N 2), and most of our algorithms' time complexity revolves around these two.

Space Complexity:

- The space complexity criterion helps us compare the space the algorithm uses to sort any data set. If an algorithm consumes a lot of space for larger inputs, it is considered a poor algorithm for sorting large data sets. In some

cases, we might prefer a higher space complexity algorithm if it proposes exceptionally low time complexity, but not in general.

- And when we talk about space complexity, the term in-place sorting algorithm arises. The algorithm which results in constant space complexity is called an in-place sorting algorithm. In-place sorting algorithms mostly use swapping and rearranging techniques to sort a data set. One example is Bubble Sort.

Stability:

The stability of an algorithm is judged by the fact whether the order of the elements having equal status when sorted on some basis is preserved or not. It probably sounded technical, but let us explain.

Suppose, we have a set of numbers, 6, 1, 2, 7, 6 and we want to sort them in increasing order by using an algorithm. Then the result would be 1, 2, 6, 6, 7. But the key thing to look at is whether the 6s follow the same order as that given in the input or they have changed. That is, whether the first 6 still comes before the second 6 or not. If they do, then the algorithm we followed is called stable, otherwise unstable.

Adaptivity:

Algorithms that adapt to the fact that if the data are already sorted and it must take less time are called adaptive algorithms. And algorithms which do not adapt to this situation are not adaptive.

Recursiveness:

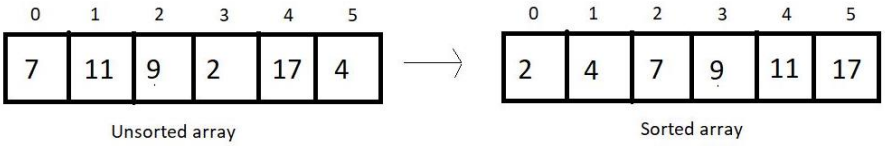
If the algorithm uses recursion to sort a data set, then it is called a recursive algorithm. Otherwise, non-recursive.

Internal & External Sorting Algorithms:

When the algorithm loads the data set into the memory (RAM), we say the algorithm follows internal sorting methods. In contrast, we say it follows the external sorting methods when the data doesn't get loaded into the memory.

Bubble Sort Algorithm

With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration.

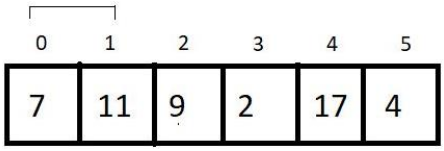


Bubble sort intends to sort an array using (n-1) passes where n is the array's length. And in one pass, the largest element of the current unsorted part reaches its final position, and our unsorted part of the array reduces by 1, and the sorted part increases by 1.

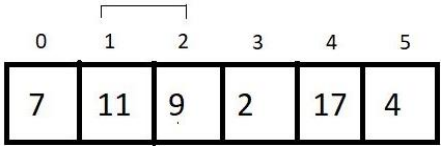
At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array reaches its final position at the end. Since our array is of length 6, we will make 5 passes. It wouldn't take long for us to understand why.

1st Pass:

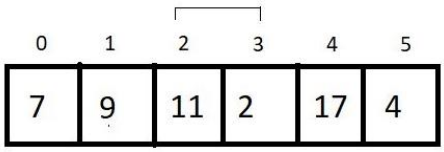
At first pass, our whole array comes under the unsorted part. We will start by comparing each adjacent pair. Since our array is of length 6, we have 5 pairs to compare. Let's start with the first one.



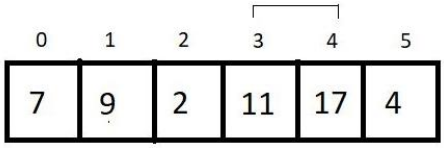
Since these two are already sorted, we move ahead without making any changes.



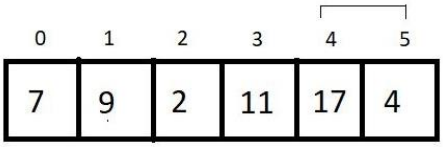
Now since 9 is less than 11, we swap their positions to make them sorted.



Again, we swap the positions of 11 and 2.

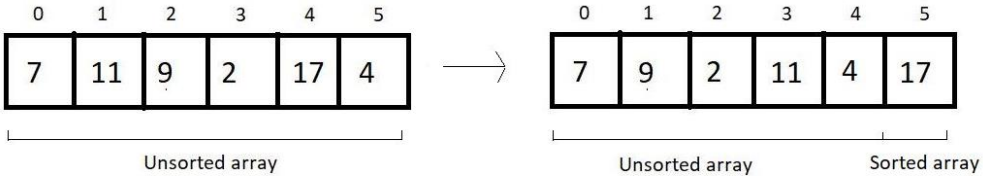


We move ahead without changing anything since they are already sorted.



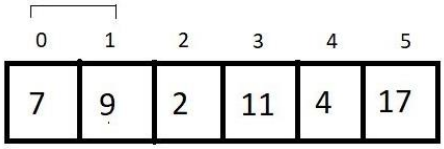
Here, we make a swap since 17 is greater than 4.

And this is where our first pass finishes. We should make an overview of what we received at the end of the first pass.

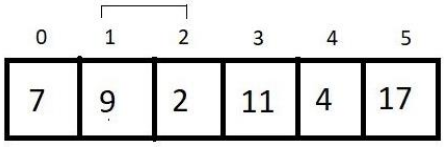


2nd Pass:

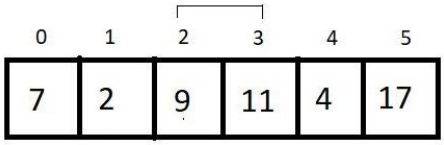
We again start from the beginning, with a reduced unsorted part of length 5. Hence the number of comparisons would be just 4.



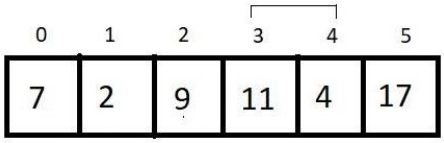
No changes to make.



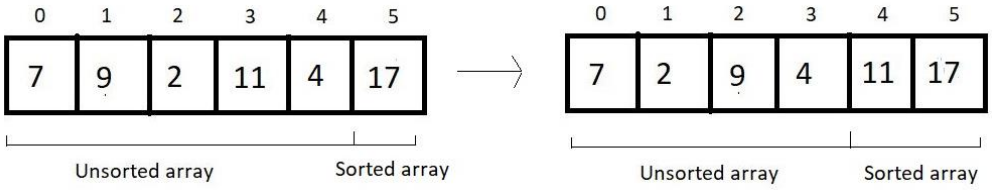
Yes, here we make a swap, since 9>2.



Since 9 < 11, we move further.

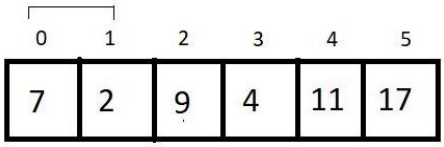


And since 11 is greater than 4, we make a swap again. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

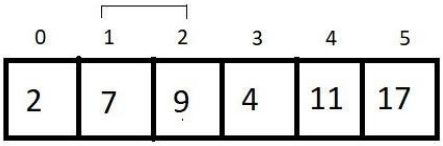


3rd Pass:

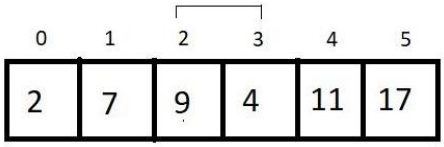
We'll again start from the beginning, and this time our unsorted part has a length of 4; hence no. of comparisons would be 3.



Since 7 is greater than 2, we make a swap here.



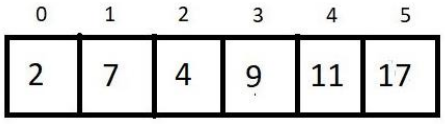
We move ahead without making any change.



In this final comparison, we make a swap, since 9 > 4.

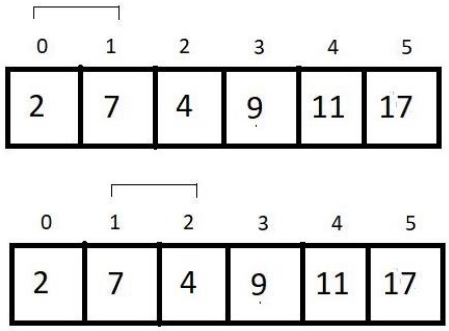
And that was our third pass.

And the result at the end was:

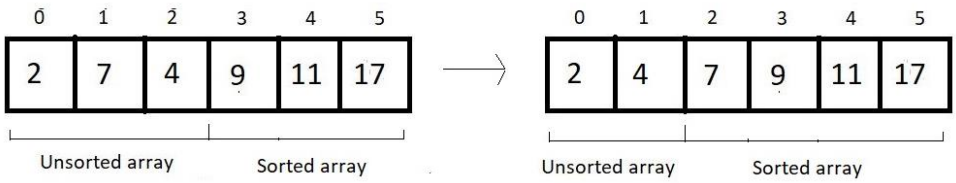


4th Pass:

We just have the unsorted part of length 3, and that would cause just 2 comparisons. So, let's see them.

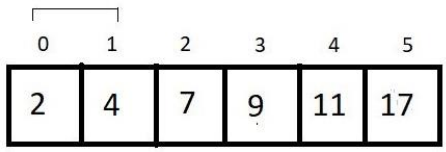


We swap their positions. And that is all in the 4th pass. The resultant array after the 4th pass is:

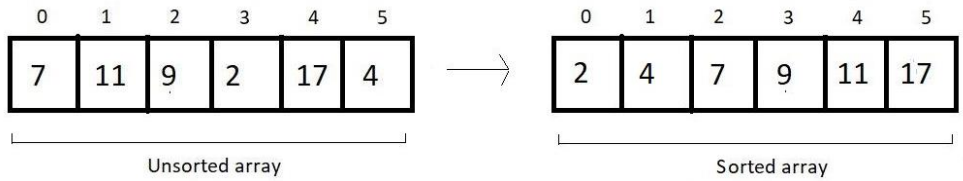


5th (last) Pass:

We have only one comparison to make here.



And since these are already sorted, we finish our procedure here. And see the final results:



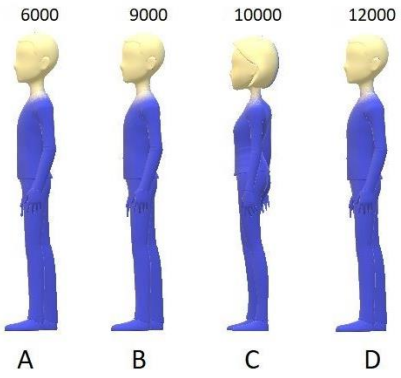
And this is what the Bubble Sort algorithm looks like. We have a few things to conclude and few calculations regarding the complexity of the algorithm to make.

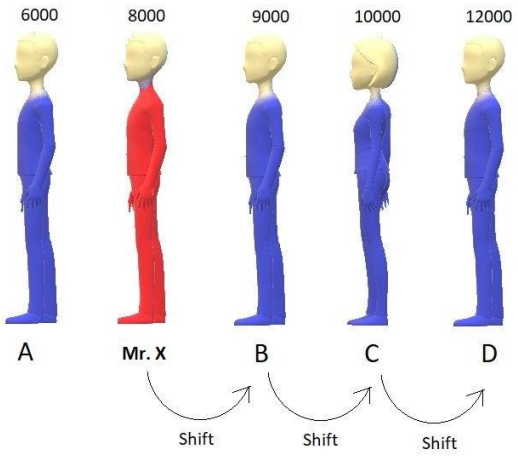
Time Complexity of Bubble Sort:

- If we count the number of comparisons we made, there were (5+4+3+2+1), that is, a total of 15 comparisons. And every time we compared, we had a fair probability of making a swap. So, 15 comparisons intend to make 15 possible swaps. Let us quickly generalize this sum. For length 6, we had 5+4+3+2+1 number of comparisons and possible swaps. Therefore, for an array of length n, we would have (n-1) + (n-2) + (n-3) + (n-4) + . . . + 1 comparison and possible swaps.
- This is a high school thing to find the sum from 1 to n-1, which is n(n-1)/2, and hence our complexity of runtime becomes O(n^2).
- And if we could observe, we never made a swap when two elements of a pair become equal. Hence the algorithm is a stable algorithm.
- It is not a recursive algorithm since we didn't use recursion here.
- This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness.

Insertion Sort Algorithm

Suppose Mr. X wants to stand in a queue where people are already sorted on the basis of the amount of money they have. Person with the least amount is standing in the front and the person with the largest sum in his pocket stands last. The below illustration describes the given situation.

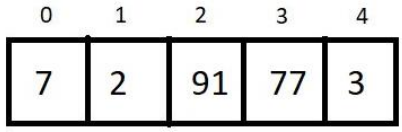




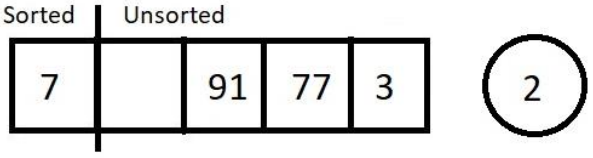
Problem arises when Mr. X suppose he has 8000tk in his pocket, and he wants to be a part of this queue. He doesn't know where to stand. So, now he starts from the last and keep asking the person standing there whether he has more money than him or less money than him. If he finds someone with more money, simply he ask him/her to shift backward. And the moment he finds a person having less money than him, he stands just behind him/her. So, after doing all this, Mr. X finds a position in the 2nd place in the queue.

Now, suppose these were not the people but the numbers in an array. It would have been as simple as it is right now. We would keep comparing two numbers, and if we find a number greater than the number we want to insert, we shift it backward. And the moment we find a number smaller, we insert the element at the vacant space just behind the smaller number.

Insertion Sort Algorithm: Let's just take an array, and use the insertion sort algorithm to sort its elements in increasing order.

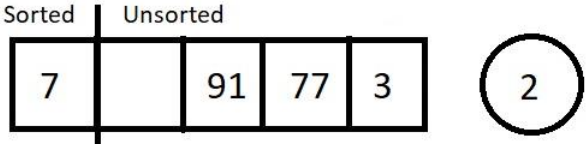


We have learned to put an arbitrary element inside a sorted array, using the insertion method we saw above. And an array of a single element is always sorted. So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

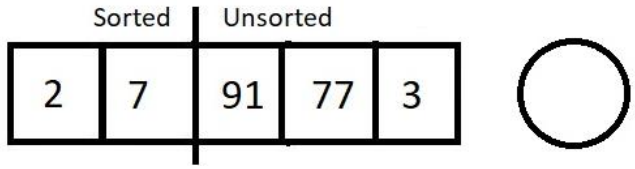


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let's call each of these insertions and the traversal of the sorted subarray to find the best position, a pass.

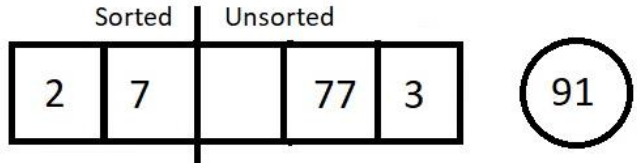
So, let's start with pass 1, which is to insert 2 in the sorted array of length 1.



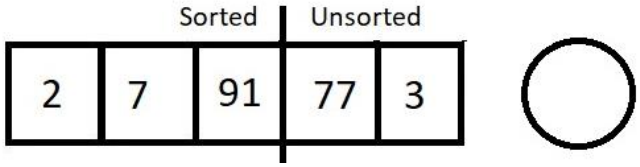
So, we plucked the first element from the unsorted part. Let's insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



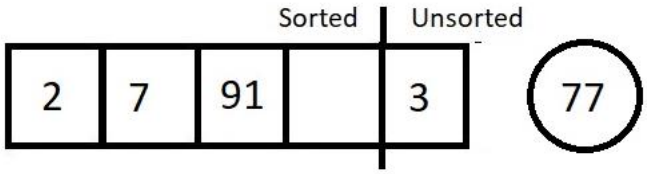
Let's proceed to the next pass.



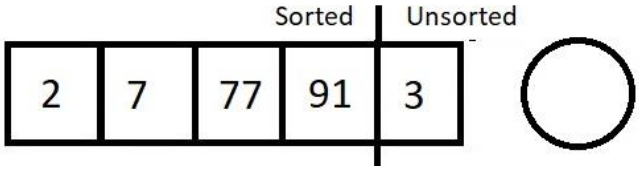
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



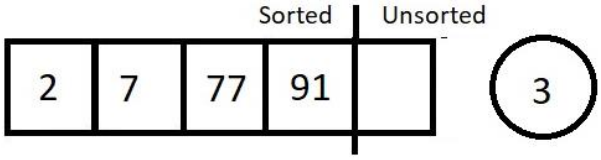
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let's proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



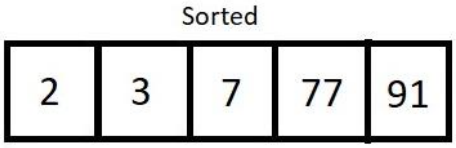
We started checking its best fit, and found the place next to element 7. So, this time it would cause just a single shift of element 91.



As a result, we are left with a single element in the unsorted subarray. Let's pull that out too in our last pass.



Since our new element to insert is the element 3, we started checking for its position from the back. The position is, no doubt, just next to element 2. So, we shifted elements 7, 77, and 91. Those were the only three shifts. And the final sorted we received is illustrated below.



So, this was the main procedure behind the insertion sort algorithm.

Analysis: Conclusively, we had to have 4 passes to sort an array of length 5. And in the first pass, we had to compare the to-be inserted element with just one single element 7. So, only one comparison, and one possible swap. Similarly, for i th pass, we would have i number of comparisons, and i possible swaps.

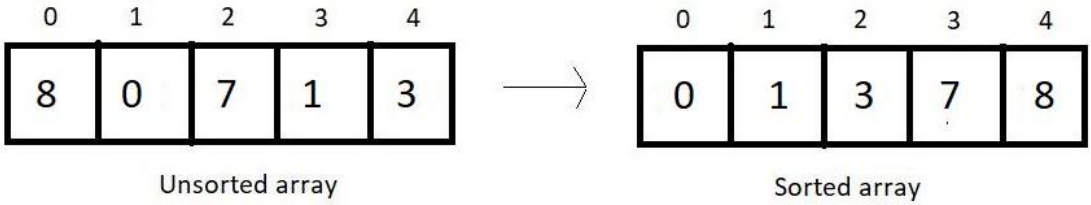
Time Complexity of Insertion Sort Algorithm:

- We made 4 passes for this array of length 5, and for I th pass, we made i number of comparisons. So, the total number of comparisons is 1+2+3+4. Similarly, for an array of length n, the total number of comparison/possible swaps would be 1+2+3+4+ . . . + (n-1) which is n(n-1)/2, which ultimately is O(n2).
- Insertion sort algorithm is a stable algorithm, since we start comparing from the back of the sorted subarray, and never cross an element equal to the to be inserted element.
- Insertion sort algorithm is an adaptive algorithm. When our array is already sorted, we just make (n-1) passes, and don't make any actual comparison between the elements. Hence, we accomplish the job in O(n).

Note: At each pass, we get a sorted subarray at the left, but this intermediate state of the array has no real significance, unlike the bubble sort algorithm where at each pass, we get the largest element having its position fixed at the end.

Selection Sort Algorithm

Suppose we are given an array of integers, and we are asked to sort them using the selection sort algorithm, then the array after being sorted would look something like this.



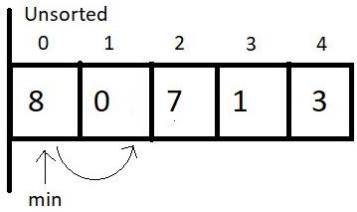
In selection sort, at each pass, we make sure that the smallest element of the current unsorted subarray reaches its final position. And this is pursued by finding the smallest element in the unsorted subarray and replacing it at the end with the element at the first index of the unsorted subarray. This algorithm reduces the size of the unsorted part by 1 and increases the size of the sorted part by 1 at each respective pass. Let's see how these work. Take a look at the unsorted array above, and we'll walk through each pass one by one and see how we reach the result.

At each pass, we create a variable min to store the index of the minimum element. We start by assuming that the first element of the unsorted subarray is the minimum. We will iterate through the unsorted part of the array, and compare every element to this element at min index. If the element is less than the element at min index, we replace min by the current index and move ahead. Else, we keep going. And when we reach the end of the array, we replace the first element of the unsorted subarray with the element at min index. And doing this at every pass ensures that the smallest element of the unsorted part of the array reaches its final position at the end.

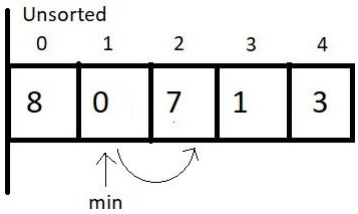
Since our array is of length 5, we will make 4 passes. We must have realized by now the reason why it would take just 4 passes.

1st Pass:

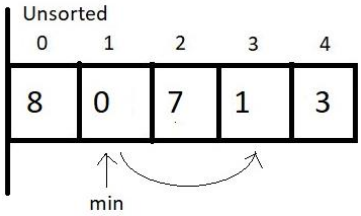
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the **min** index. Now, we'll have to check among the remaining 4 elements if there is still a lesser element than the first one.



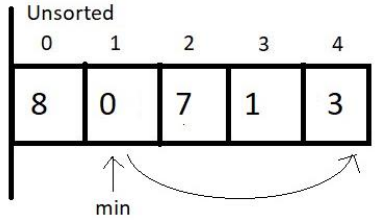
And when we compared the element at min index with the element at index 1, we found that 0 is less than 8 and hence we update our min index to 1.



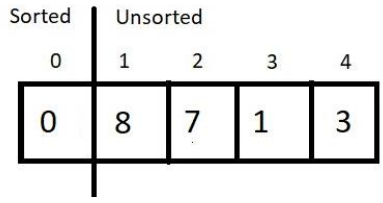
And now we keep checking with the updated min. Since 7 is not less than 0, we move ahead.



And now we compared the elements at index 1 and 3, and 0 is still lesser than 1, so we move ahead without making any changes.

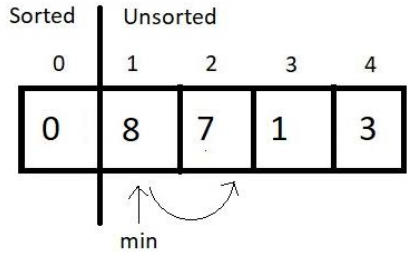


And now we compared the element at the min index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at 0th index with the element at the min index. And this gives us our first sorted subarray of size 1. And this is where our first pass finishes. We should make an overview of what we received at the end of the first pass.

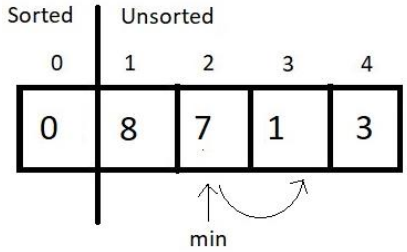


2nd Pass:

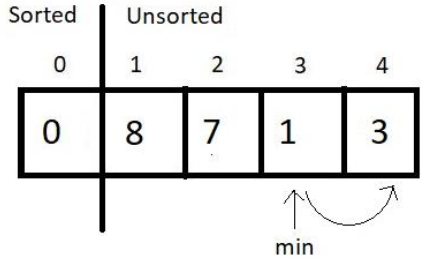
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the min index and start iterating to the right for finding the minimum element.



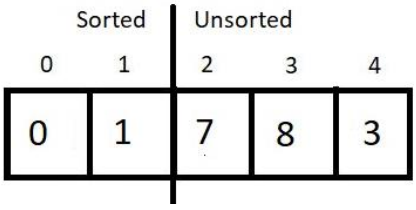
Since 7 is less than 8, we update our min index with 2. And move further.



Next, we compared the elements 7 and 1, and since 1 is still lesser than 7, we update the min index by 3. Then, we move ahead to the next comparison.

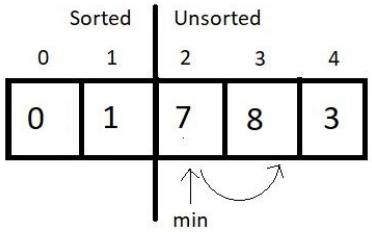


And since 3 is greater than 1, we don't make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at min index. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

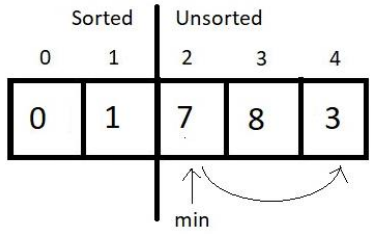


3rd Pass:

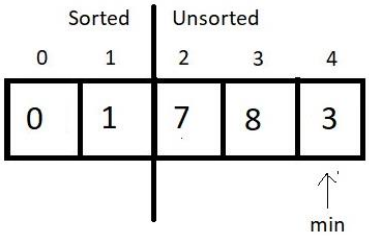
We'll again start from the beginning of the unsorted subarray which is from the index 2, and make the min index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.



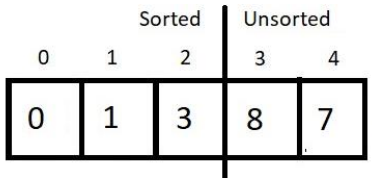
Since 8 is greater than 7, we would make no change, but move ahead.



Comparing the elements at index min and 4, we found 3 to be smaller than 7 and hence an update is needed here. So, we update min to 4.

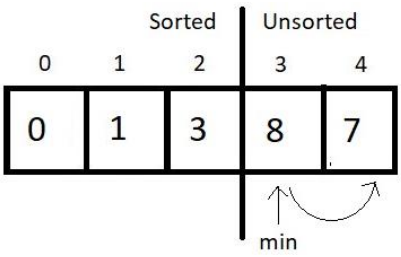


And since that was the last comparison of the third pass, we make a swap of the indices 2 and min. And the result at the end would be:

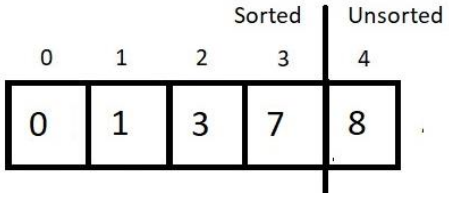


4th Pass:

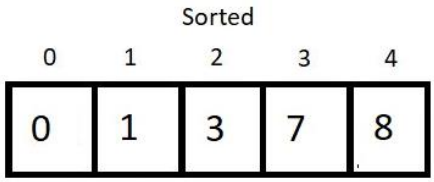
We now have the sorted subarray of length 3, hence the new min would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let's see that.



And since 7 is less than 8, we update our min to 4. And since that was the only comparison in this pass, we finish our procedure here by swapping the elements at the indices min and 3. And see at the final results:



And since a subarray with a single element is always sorted, we ignore the only unsorted part and make it sorted too.



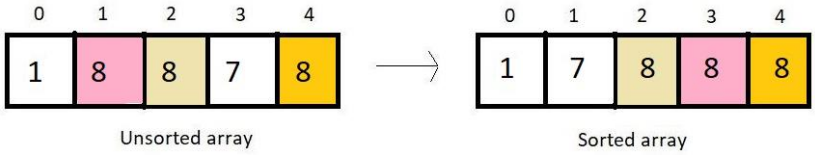
And this is why the Selection Sort algorithm got its name. We select the minimum element at each pass and give it its final position. Few conclusions before we proceed to the programming segment:

Time Complexity of Selection Sort:

We made 4 passes for an array of length 5. Therefore, for an array of length n we would have to make n-1 passes. And if you count the number of comparisons we made at each pass, there were (4+3+2+1), that is, a total of 10 comparisons. And every time we compared; we had a fair possibility of updating our min. So, 10 comparisons are equivalent to making 10 updates.

So, for length 5, we had 4+3+2+1 number of comparisons. Therefore, for an array of length n, we would have (n-1) + (n-2) + (n-3) + (n-4) + ... + 1 comparisons. Sum from 1 to n-1, we get, and hence the time complexity of the algorithm would be O(n2).

Selection sort algorithm is **not a stable algorithm.** Since the smallest element is replaced with the first element at each pass, it may jumble up positions of equal elements very easily. Hence, unstable. Refer to the example below:



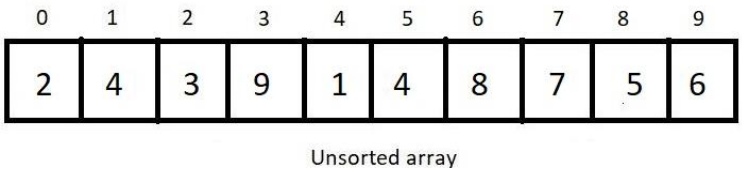
Selection sort would anyways compare every element with the min element, regardless of the fact if the array is sorted or not, hence selection sort is **not an adaptive algorithm** by default.

This algorithm offers the benefit of making the least number of swaps to sort an array. We don't make any redundant swaps here.

Quick Sort Algorithm

The Quick Sort algorithm is quite different from the ones we have studied so far. Here, we use the divide and conquer method to sort our array in pieces reducing our effort and space complexity of the algorithm. There are two new concepts we must know before you jump into the core. First is the **divide and conquer method**. As the name suggests, Divide and Conquer divides a problem into subproblems and solves them at their levels, giving the output as a result of all these subproblems. The second is the **partition method** in sorting. In the partition method, we choose an element as a pivot and try pushing all the elements smaller than the pivot element to its left and all the greater elements to its right. We thus finalize the position of the pivot element. Quick Sort is implemented using both these concepts

Suppose we are given an array of integers, and we are asked to sort them using the quicksort algorithm, then the very first task we would do is to choose a pivot. Pivots are chosen in various ways, but for now, we'll consider the first element of every unsorted subarray as the pivot. Remember this while we proceed.



In the quicksort algorithm, every time we get a fresh unsorted subarray, we do a partition on it. Partition asks you to first choose an element as a pivot. And as already decided, we would choose the first element of the unsorted subarray as the pivot. We would need two more index variables, i and j. Below enlisted is

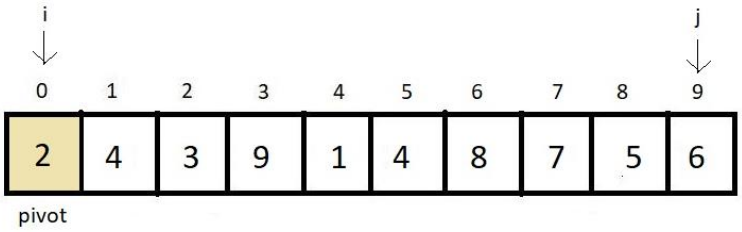
the flow of our partition algorithm we must adhere to. We always start from step 1 with each fresh partition call.

a. Define i as the low index, which is the index of the first element of the subarray, and j as the high index, which is the index of the last element of the subarray.
b. Set the pivot as the element at the low index i since that is the first index of the unsorted subarray.
c. Increase i by 1 until you reach an element greater than the pivot element.
d. Decrease j by 1 until you reach an element smaller than or equal to the pivot element.
e. Having fixed the values of i and j, interchange the elements at indices i and j.
f. Repeat steps 3, 4, and 5 until j becomes less than or equal to i.
g. Finally, swap the pivot element and the element at the index.

This was the **partitioning algorithm**. Every time we call a partition, the pivot element gets its final position. A partition never guarantees a sorted array, but it does guarantee that all the smaller elements are located to the pivot's left, and all the greater elements are located to the pivot's right.

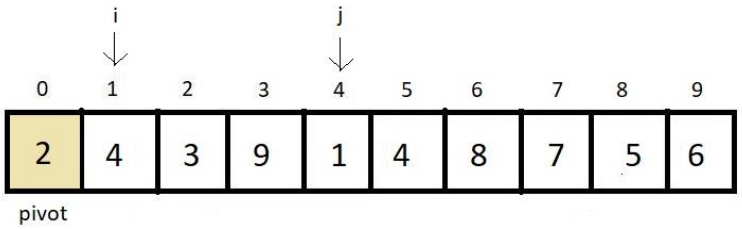
Now let's look at how the array we received at the beginning gets sorted using partitioning and divide and conquer recursively for smaller subarrays.

Firstly, the whole array is unsorted, and hence we apply quicksort on the whole array. Now, we apply a partition in this array. Applying partition asks us to follow all the above steps we discussed.

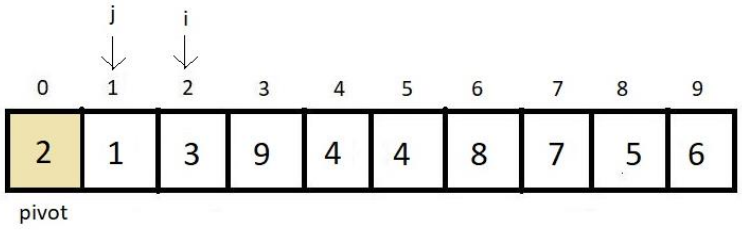


Current Unsorted Subarray

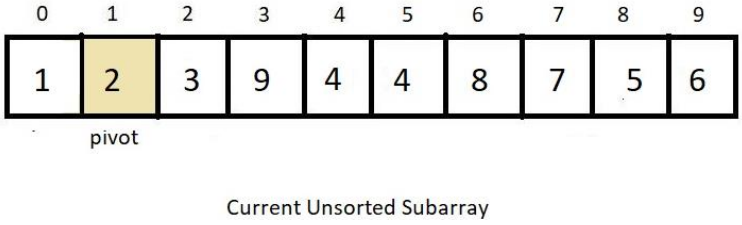
Keep increasing i until we reach an element greater than the pivot, and keep decreasing j until we reach an element smaller or equal to the pivot.



Swap the two elements and continue the search further until j crosses i or becomes equal to i.

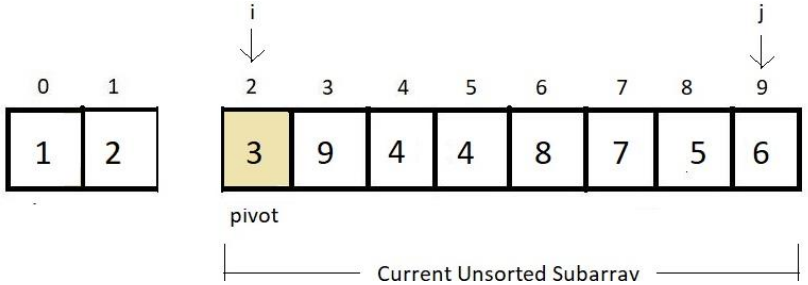


As j crossed i while searching, we followed the final step of swapping the pivot element and the element at j.

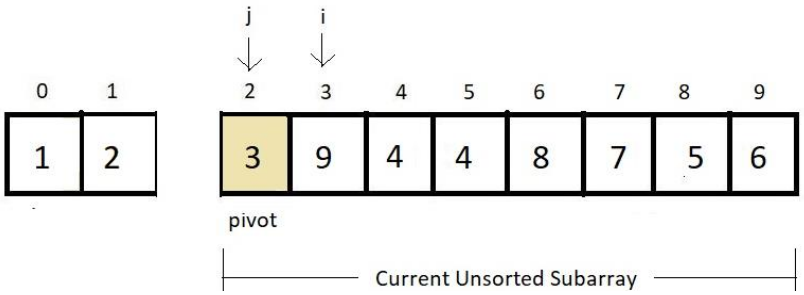


And this would be the final position of the current pivot even in our sorted array. As we can see, all the elements smaller than 2 are on the left, and the rest greater than 2 are on the right. Here comes the role of divide and conquer. We separate our focus from the whole array to just the subarrays, which are not sorted yet. Here, we have subarrays {1} and {3, 9, 4, 4, 8, 7, 5, 6} unsorted. So, we make a call to quicksort on these two subarrays.

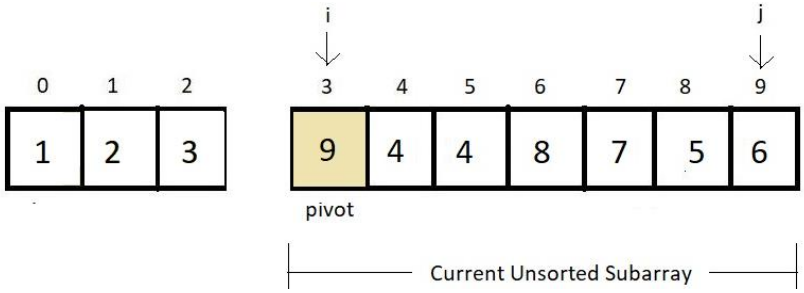
Now since the first subarray has just a single element, we consider it sorted. Let's now sort the second subarray. Follow all the partition steps from the beginning.



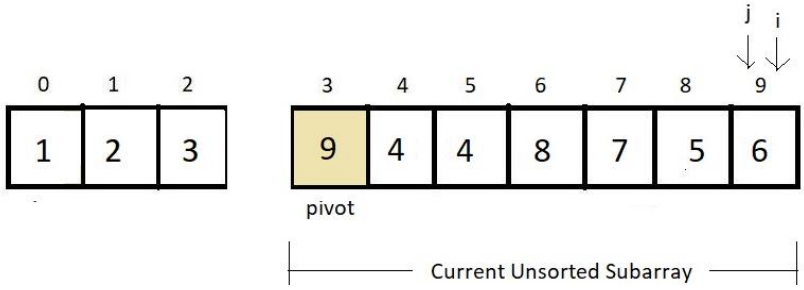
Now, our new pivot is the element at index 2. And i and j are 2 and 9, respectively, marking the start and the end of the subarray. Follow steps 3 and 4.



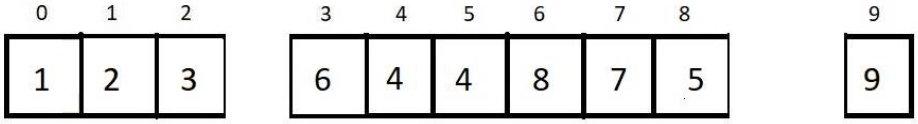
And since there were no elements smaller than 3, j crosses i in the very first iteration. This means 3 was already at its sorted index. And there are no elements to its left; the only unsorted subarray is {9, 4, 4, 8, 7, 5, 6}. And our new situation becomes:



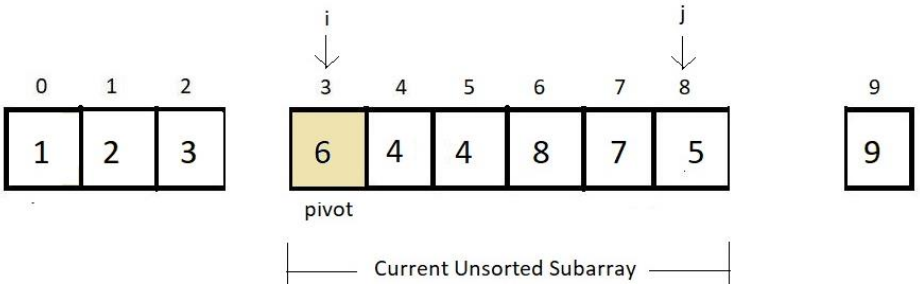
Repeating steps 3 and 4.



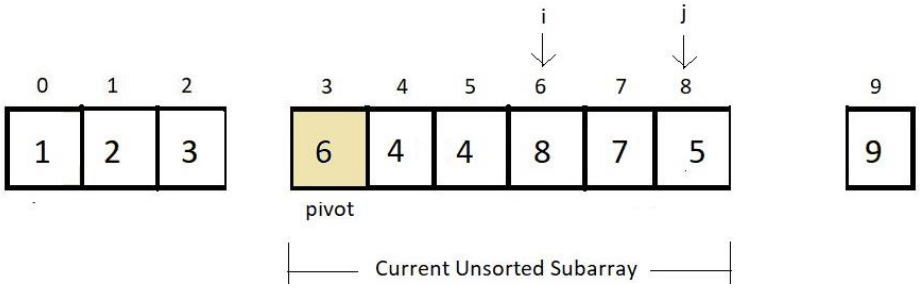
We found that there was no element greater than 9, and hence i reached the last. And 6 was the first element j found to be smaller than 9, and they collided. And this is where we do step 7. Swap the pivot element and the element at index j.



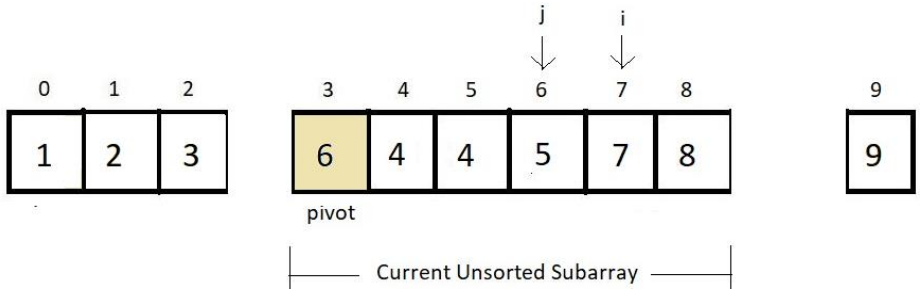
And since there are no elements to the right of element 9, we just have one sorted subarray {6, 4, 4, 8, 7, 5}. Let's call a partition on this as well.



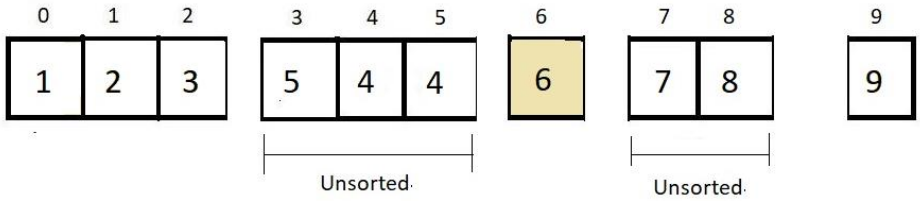
Repeat steps 3 and 4.



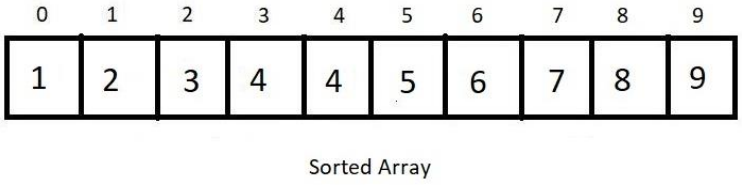
And since the condition j<=i has not been met yet, we just swap the elements at index i and j and continue our search.



And now i and j crossed each other, and now only we swap our pivot element and element at j.



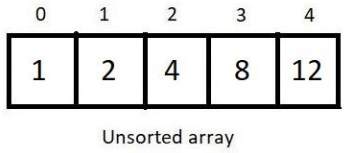
And now, we again divide and consider only the elements that remain unsorted to the left of the pivot and the right of the pivot. And moving things further would just waste our time. We can assume that things move as expected, and it will get sorted at the end and would look something like this.



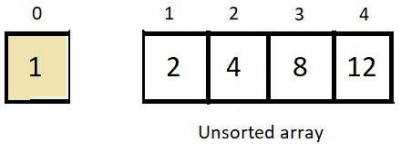
Analysis of Quick Sort Algorithm:

Let's start with the **runtime complexity** of the quick sort algorithm:

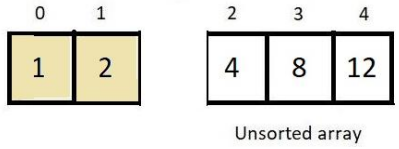
Worst Case: The worst-case in a quicksort algorithm happens when our array is already sorted. I'll take a sorted array of length 5 to demonstrate how it reaches the worst case. Take the one below as an example.



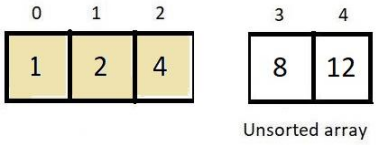
In the first step, we choose 1 as the pivot and apply a partition on the whole array. Since 1 is already at its correct position, we apply quicksort on the rest of the subarrays.



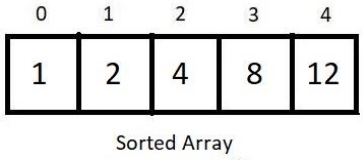
Next, the pivot is element 2, and when applied partition, we found that there is no element less than 2 in the subarray; hence, the pivot remains there itself. We further apply quicksort on the only subarray that is to the right.



Now, the pivot is 4, and since that is already at its correct position, applying partition did make no change. We move ahead.



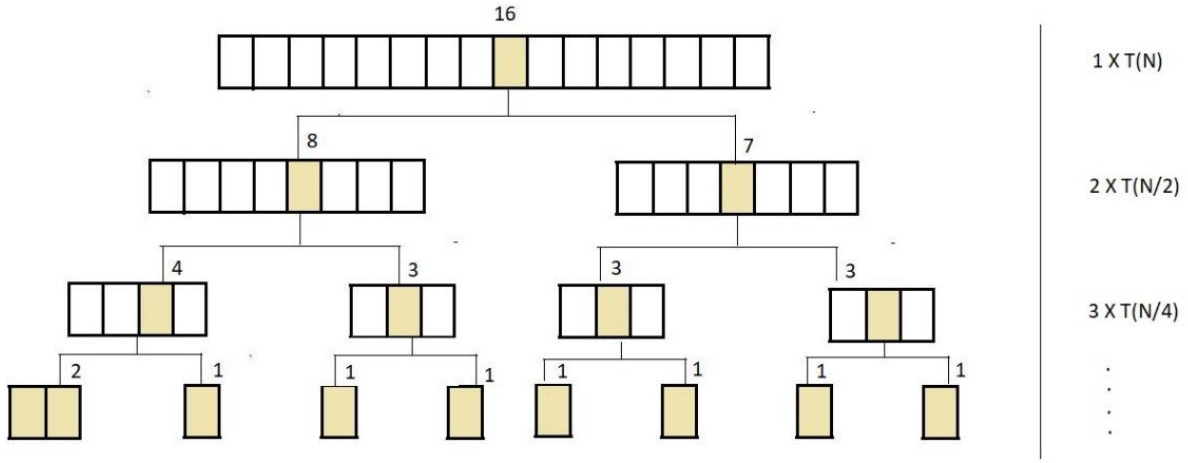
Now, the pivot is 8, and even that is at its correct position; hence things remain unchanged, and there is just one subarray with a single element left. But since any array with a single element is always sorted, we mark the element 12 sorted as well. And hence our final sorted array becomes,



So, if we calculate carefully, for an array of size 5, we had to partition the subarray 4 times. That is, for an array of size n, there would be (n-1) partitions. Now, during each partition, long story short, we made our two index variables, i and j run from either direction towards each other until they actually become equal or cross each other. And we do some swapping in between as well. These operations count to some linear function of n, contributing O(n) to the runtime complexity.

And since there are a total of (n-1) partitions, our total runtime complexity becomes n(n-1) which is simply **O(n2)**. This is our worst-case complexity.

Best Case: The condition when our algorithm performs in its best possible time complexity is when our array gets divided into two almost equal subarrays at each partition. Below mentioned tree defines the state of best-case when we apply quicksort on an array of 16 elements, of which each newly made subarray is almost half of its parent array.



No. partitions were different at each level of the tree. If we count starting from the top, the top-level had one partition on an array of length (n=16), the second level had 2 partitions on arrays of length n/2, then the third level had 4 partitions on arrays of length n/4… and so on.

For the above array of length 16, the calculation goes like the one below.

Here, T(x) is the time taken during the partition of the array with x elements. And as we know, the partition takes a linear function time, and we can assume T(x) to be equal to x; hence the total time complexity becomes,

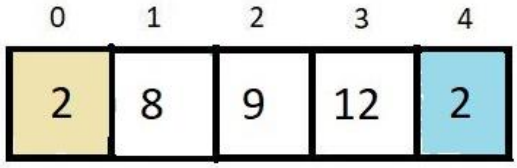
Total time = 1(n) + 2(n/2) +4(n/4) + ...+ until the height of the tree(h) Total time = n*h

H is the height of the tree, and the height of the tree is log2(n), where n is the size of the given array. In the above example, h = 4, since log2(16) equals 4. Hence the time complexity of the algorithm in its best case is **O(nlogn)**.

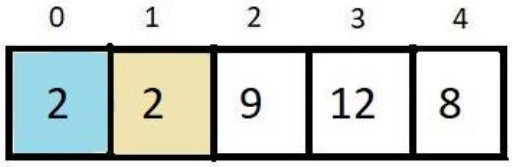
Note: The average time complexity remains **O(nlogn)**. Calculations have been avoided here due to their complexity.

Stability:

The quick sort algorithm is not stable. It does swaps of all kinds and hence loses its stability. An example is illustrated below.



When we apply the partition on the above array with the first element as the pivot, our array becomes



And the two 2s get their order reversed. Hence quick sort is not stable.

Note: Quicksort algorithm is an in-place algorithm. It doesn't consume any extra space in the memory. It does all kinds of operations in the same array itself.

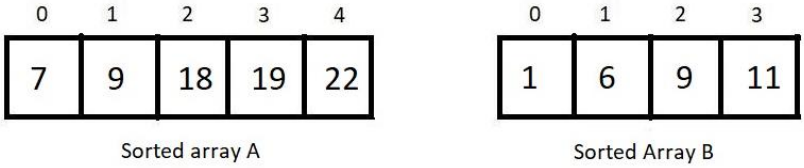
There is no hard and fast rule to choose only the first element as the pivot; rather, we can have any random element as its pivot using the rand() function and that we wouldn't believe actually reduces the algorithm's complexity.

Merge Sort Algorithm

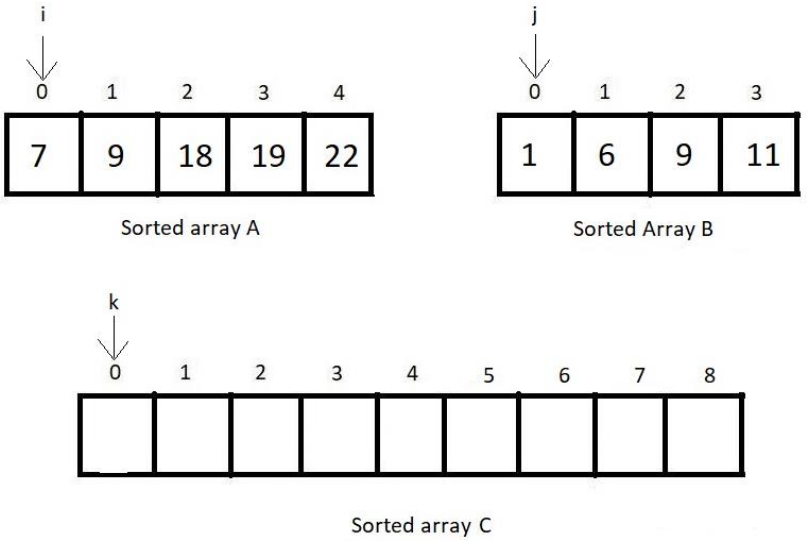
In this algorithm, we divide the arrays into subarrays and subarrays into more subarrays until the size of each subarray becomes 1. Since arrays with a single element are always considered sorted, this is where we merge. Merging two sorted subarrays creates another sorted subarray. We'll show first how merging two sorted subarrays works.

Merging Procedure:

Suppose we have two sorted arrays, A and B, of sizes 5 and 4, respectively.

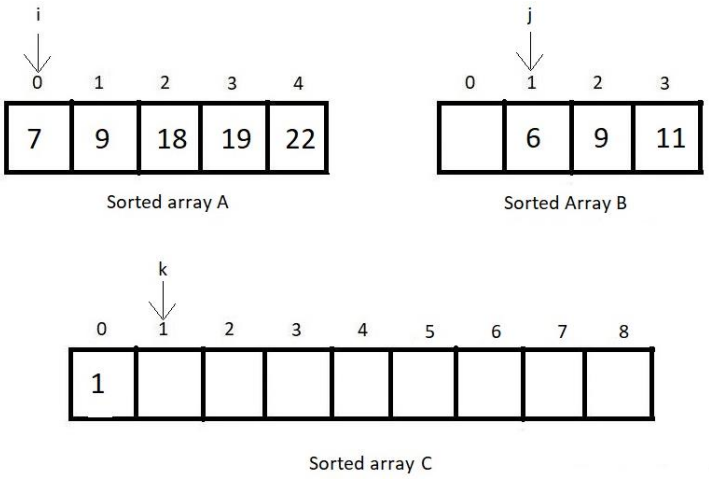


- And we apply merging on them. Then the first job would be to create another array C with size being the sum of both the raw arrays' sizes. Here the sizes of A and B are 5 and 4, respectively. So, the size of array C would be 9.
- Now, we maintain three index variables i, j, and k. i looks after the first element in array A, j looks after the first element in array B, and k looks after the position in array C to place the next element in.

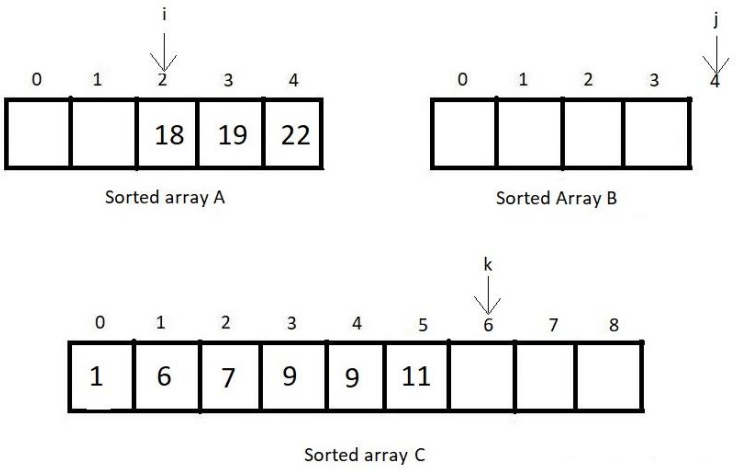


- Initially, all i, j, and k are equal to 0.
- Now, we compare the elements at index i of array A and index j of array B and see which one is smaller. Fill in the smaller element at index k of array C and increment k by 1. Also, increment the index variable of the array we fetched the smaller element from.

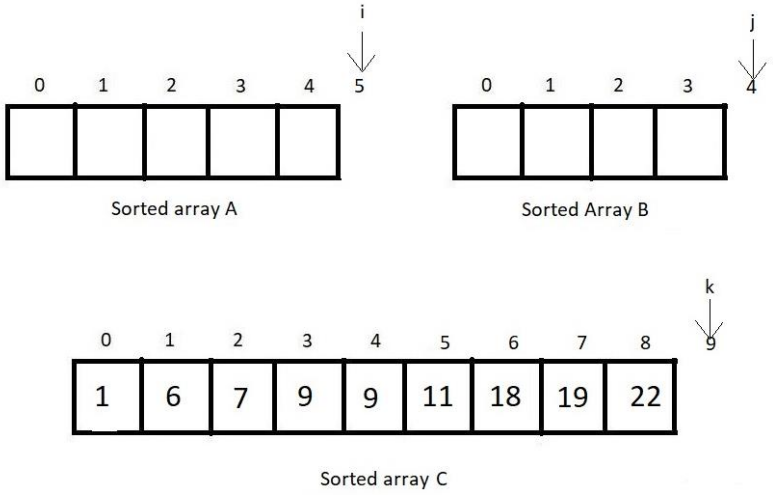
- Here, A[i] is greater than B[j]. Hence, we fill C[k] with B[j] and increase k and j by 1.



- We continue doing step 5 until one of the arrays, A or B, gets empty.



Here, array B inserted all its elements in the merged array C. Since we are only left with the elements of element A, we simply put them in the merged array as they are. This will result in our final merged array C.



The programming part of the merge procedure is not that tough. We just need to follow these steps:

a. Take both the arrays and their sizes to be merged as the parameters of the merge function. By summing the sizes of the two arrays, we can create one larger array.

b. Create three index variables i, j & k. And initialize all of them with 0.

c. And then run a while loop with the condition that both the index variables i and j don't exceed their respective array limits.

d. Now, at each run, see if A[i] is smaller than B[j], if yes, make C[k] = A[i] and increase both i and k by 1, else C[k] = B[j] and both j and k are incremented by 1.

e. And when the loop finishes, either array A or B or both get finished. And now we run two while loops for each array A and B, and insert all the remaining elements as they are in the array C. And we are done merging.

❖ The pseudocode for the above procedure has been attached below.

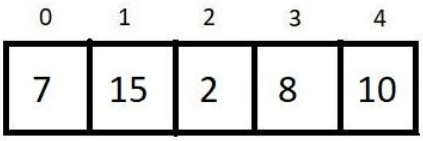
```
void Merge(int A[], int B[], int C[], int n, int m)
{
  int i=0, j=0, k=0;
  while (i <=n && j <= m){
    if (A[i] < B[j]){
      C[k] = A[i];
      i++;
      k++;
    }
    else{
      C[k] = B[j];
      j++;
      k++;
    }
  }
  while (i <=n){
    C[k] = A[i];
    k++;
    i++;
  }
  while (j <= m){
    C[k] = B[j];
    k++;
    j++;
  }
}
```

Copying all remaining elements from A to C

Copying all remaing elements from B to C

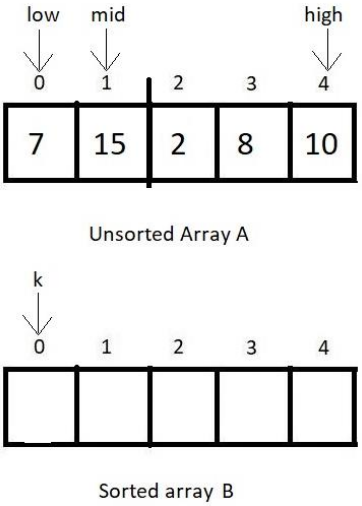
Now, this would quite not be our situation when sorting an array using the merge sort. We wouldn't have two different arrays A and B, rather a single array having two sorted subarrays. So, how to merge two sorted subarrays of a single array in the array itself.

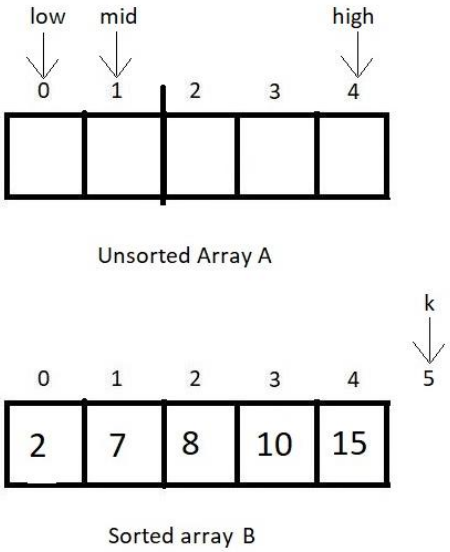
Suppose there is an array A of 5 elements and contains two sorted subarrays of length 2 and 3 in itself.



Unsorted Array A

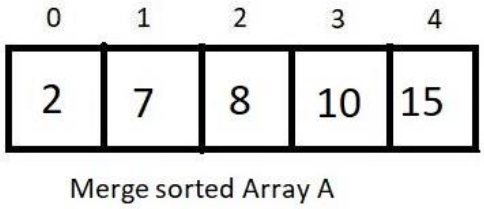
To merge both the sorted subarrays and produce a sorted array of length 5, we will create an auxiliary array B of size 5. Now the process would be more or less the same, and the only change we would need to make is to consider the first index of the first subarray as low and the last index of the second subarray as high. And mark the index prior to the first index of the second subarray as mid.





Previously we had index variables i, j, and k initialized with 0 of their respective arrays. But here, i gets initialized with low, j gets initialized with mid+1, and k gets initialized with low only. And similar to what we did earlier, i runs from low to mid, j runs from mid+1 to high, and until and unless they both get all their elements merged, we continue filling elements in array B.

After all the elements get filled in array C, we revert back to our original array A and fill the sorted elements again from low to high, making our array merge-sorted.

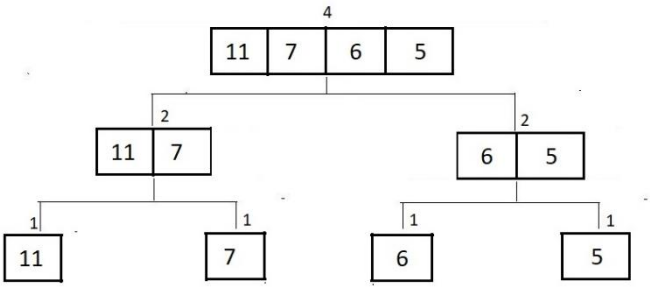


So that was our merging segment.

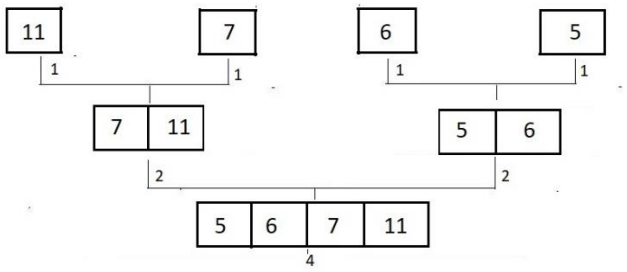
There were few changes we had to make in the pseudocode.

```
void merge(int A[], mid,  low, high)
{
   int i, j, k , B[high+1];
   i = low;
   j = mid + 1;
   k = low;
   while (i <= mid && j <= high){
      if (A[i] < A[j]){
         B[k] = A[i];
         i++;
         k++;
      }
      else{
         B[k] = A[j];
         j++;
         k++;
      }
   }
   while (i <= mid){                          Copying all remaining
      B[k] = A[i];                            elements from A to C
      k++;
      i++;
   }
   while (j <= high){                         Copying all remaing
      B[k] = A[j];                            elements from B to C
      k++;
      j++;
   }
   for (int i = low; i <= high; i++)          Copying elements back
      A[i] = B[i];                            to A from B
```

Whenever we receive an unsorted array, we break the array into fragments till the size of each subarray becomes 1. Let this be clearer via an illustration.



So, we divided the array until there are all subarrays of just length 1. Since any array/subarray of length 1 is always sorted, we just need to merge all these singly-sized subarrays into a single entity. Visit the merging procedure below.

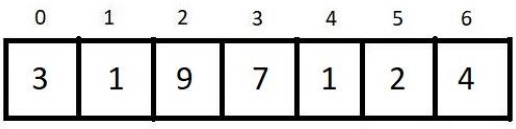


And this is how our array got merge sorted. To achieve this divided merging and sorting, we create a recursive function merge sort and pass our array and the low and high index into it. This function divides the array into two parts: one from low to mid and another from mid+1 to high. Then, it recursively calls itself passing these divided subarrays. And the resultant subarrays are sorted. In the next step, it just merges them. And that's it. Our array automatically gets sorted. Pseudocode for the merge sort function is illustrated below.

```
void MS(A[], low, high){
   int mid;
   if(low<high){
      mid = (low + high) /2;
      MS(A, low, mid);
      MS(A, mid+1, high);
      Merge(A, mid, low, high);
   }
}
```

Count Sort Algorithm

The most intuitive and **easiest** sorting algorithm, known as the count sort algorithm. Suppose we are given an array of integers and are asked to sort them using any sorting algorithm of our choice, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. Still, the method you choose to reach the result matters the most. Count Sort is one of the **fastest** methods of all. The below figure shows the array given.



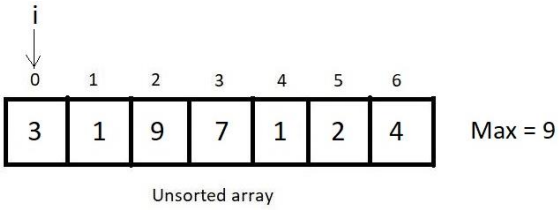
Unsorted array

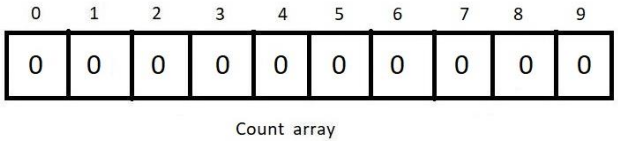
- The algorithm first asks us to find the largest element from all the elements in the array and store it in some integer variable max. Then create a count array of size max+1. This array would count the number of occurrences of some number in the given array. We will have to initialize all count array elements with 0 for that to work.
- After initializing the count array, traverse through the given array, and increment the value of that element in the count array by 1. By defining the size of the count array as the maximum element in the array, we ensure that

each element in the array has its own corresponding index in the count array. After we traverse through the whole array, we'll have the count of each element in the array.

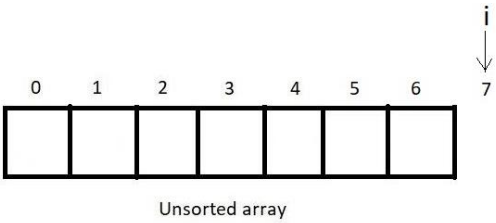
- Now traverse through the count array, and look for the nonzero elements. The moment we find an index with some value other than zero, fill in the sorted array the index of the non-zero element until it becomes zero by decrementing it by 1 every time we fill it in the resultant array. And then move further. This way, we create a sorted array. Let's take the one we have as an example above and use the count sort algorithm to sort it.

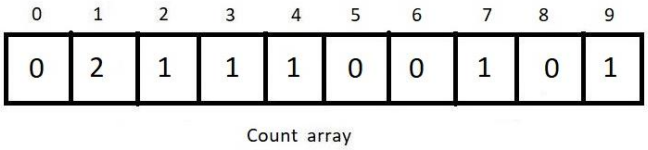
First of all, find the maximum element in the array. Here, it is 9. So, we'll create a count array of size 10 and initialize every element by 0.



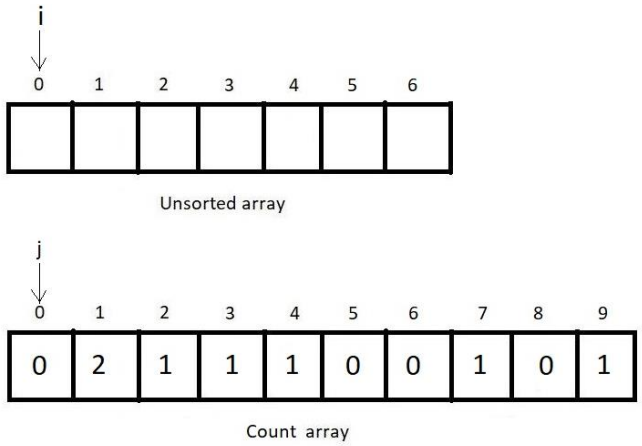


Now, let's iterate through the given array and count the no. of occurrences of each number less than equal to the maximum element.

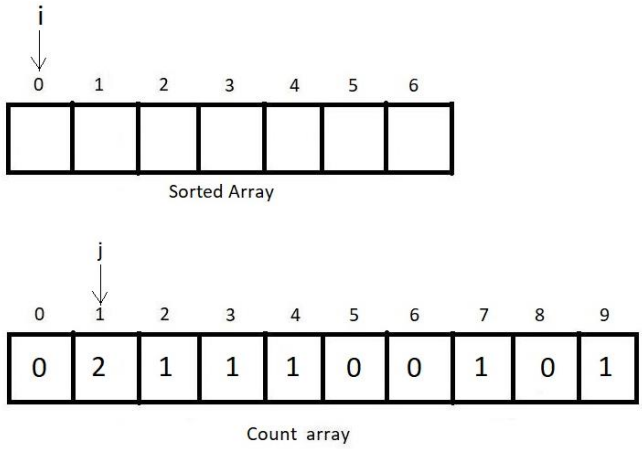




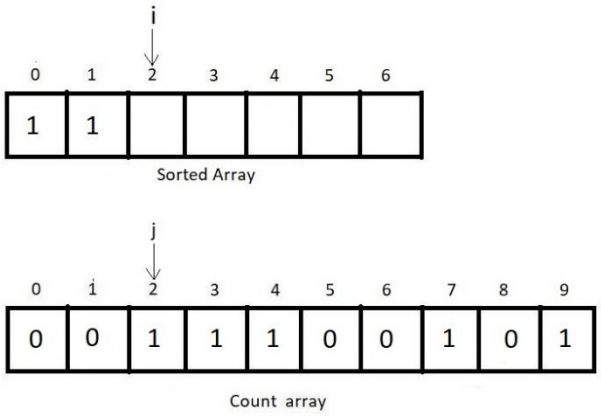
We would iterate through the count array and fill the unsorted array with the index we encounter having a non-zero number of occurrences.



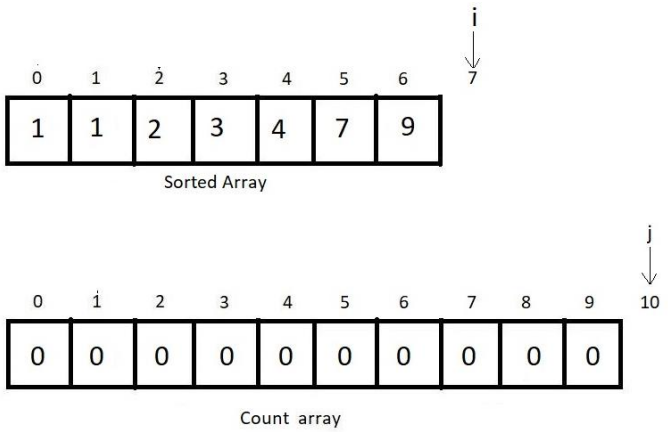
Now since there are zero numbers of zeros in the given array, we move further to index 1.



And since there were two ones in the array we were given, we push two ones in the sorted array and move our iterator to the next empty index.



And following a similar procedure for all the elements in the count array, we reach our sorted array in no time.



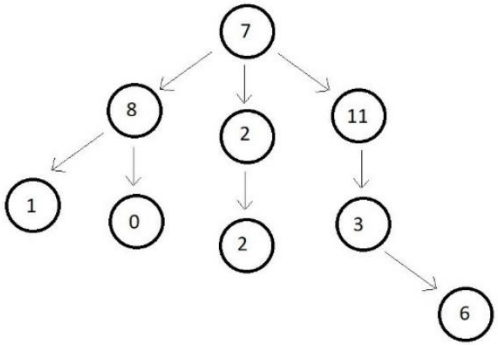
Time Complexity of Count Sort:

If we carefully observe the whole process, we only ran two different loops, one through the given array and one through the count array, which had the size equal to the maximum element in the given array. If we suppose the maximum element to be m, then the algorithm's time complexity becomes O(n+m), and for an array of some huge size, this reduces to just O(n), which is the most efficient by far algorithm.

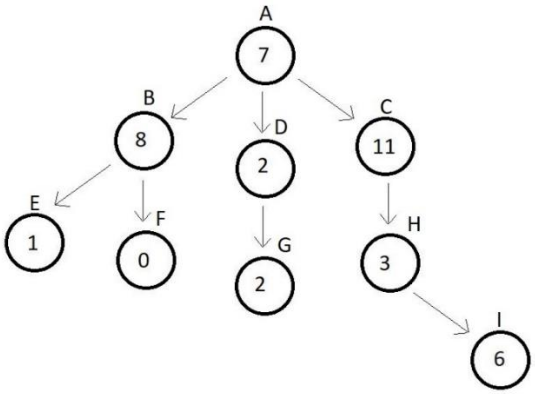
However, there is a negative point as well. The algorithm uses extra space for the count array. And this linear complexity is reachable only at the cost of the space the count array takes.

Tree

A tree usually represents the hierarchy of elements and depicts the relationships between the elements. Trees are considered as one of the largely used facets of data structures. To get a better idea of how a tree looks like, let's give an example:



Every circle represents a node, and every arrow represents the hierarchy. For us to be able to understand the terminology associated with trees, we will further name these nodes.

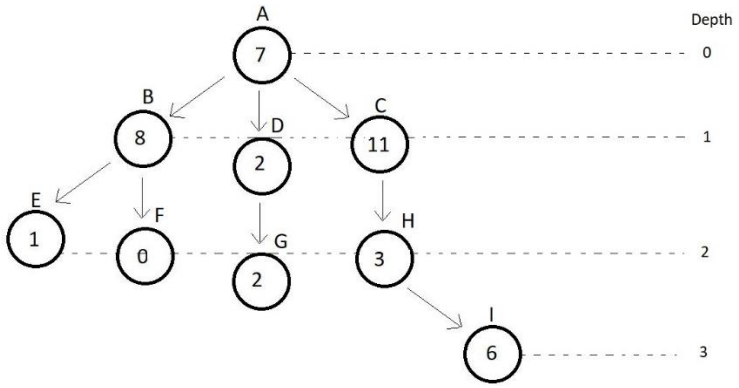


Now, we can very easily say that node C is the child of node A or node B is the parent of node E.

The terminologies used in Trees:

- **Root:** The topmost node of a tree is called the root. There is no edge pointing to it, but one or more than one edge originating from it. Here, A is the root node.
- **Parent:** Any node which connects to the child. Node which has an edge pointing to some other node. Here, C is the parent of H.

- **Child:** Any node which is connected to a parent node. Node which has an edge pointing to it from some other node. Here, H is the child of C.
- **Siblings:** Nodes belonging to the same parent are called siblings of each other. Nodes B, C and D are siblings of each other, since they have the same parent node A.
- **Ancestors:** Nodes accessible by following up the edges from a child node upwards are called the ancestors of that node. Ancestors are also the parents of the parents of … that node. Here, nodes A, C and H are the ancestors of node I.
- **Descendants:** Nodes accessible by following up the edges from a parent node downwards are called the descendants of that node. Descendants are also the child of the child of … that node. Here, nodes H and I are the descendants of node C.
- **Leaf/ External Node:** Nodes which have no edge originating from it, and have no child attached to it. These nodes cannot be a parent. Here, nodes E, F, G and I are leaf nodes.
- **Internal Node:** Nodes with at least one child. Here, nodes B, D and C are internal nodes.
- **Depth:** Depth of a node is the number of edges from root to that node. Here, the depth of nodes A, C, H and I are 0, 1, 2 and 3 respectively.



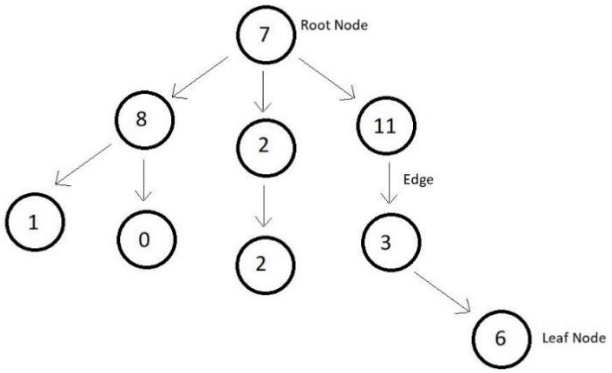
- **Height:** Height of a node is the number of edges from that node to the deepest leaf. Here, the height of node A is 3, since the deepest leaf from this node is node I. And similarly, height of node C is 2.

Apart from these, there are a few additional points that we would like to add.

- Why a tree with n nodes has n-1 edges?

Because in a tree, there is one and only edge corresponding to all the nodes except the root node. The root node has no parent, hence no edge pointing to it. Therefore, a total of n-1 edges.

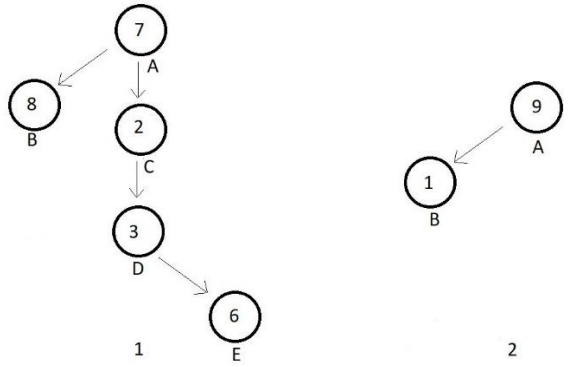
- The degree of a node in a tree is the number of children of a node.
- The degree of a tree is the highest degree of a node among all the nodes present in the tree.



In the above tree, the number of nodes is 9, and hence the number of edges are 8. We can even count and verify the fact that a tree with n nodes has n-1 edges. Moreover, the highest degree of a node is that of the root node, which has 3 children. Hence the degree of the tree is also 3.

Binary Tree:

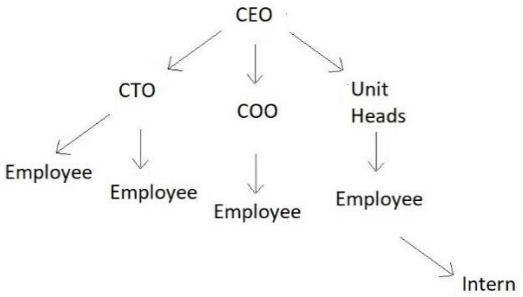
A binary tree is a special type of tree where each node has a degree equal to or less than two which means each node should have at most two children. Examples of a binary tree are mentioned below:



Example 1 has nodes A, B, C, D, E with degrees {2, 0, 1, 1, 0} respectively which satisfies the conditions for a binary tree. Similarly, example 2 has nodes A and B, having degrees 1 each, hence a binary tree.

Types of Binary Trees:

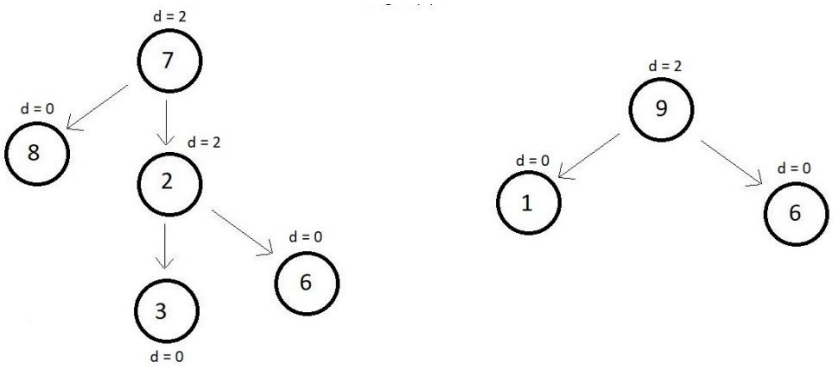
Trees are useful in representing an organizational hierarchy. Refer to the illustration below.



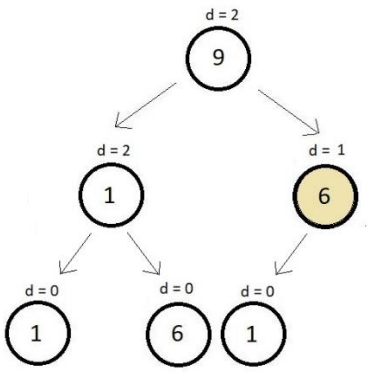
Another good example would be our file manager in desktops. There is a C Drive, it contains folders Windows, Program Files, etc. which further contain folders. So, they are hierarchically represented using a tree.

❖ Full or Strict Binary Tree:

Binary trees as we said earlier have a degree of 2 or less than 2. But a strict binary tree is a binary tree having all of its nodes with a degree of 2 or 0. That is each of its nodes either have 2 children or is a leaf node. Few simple examples follow:

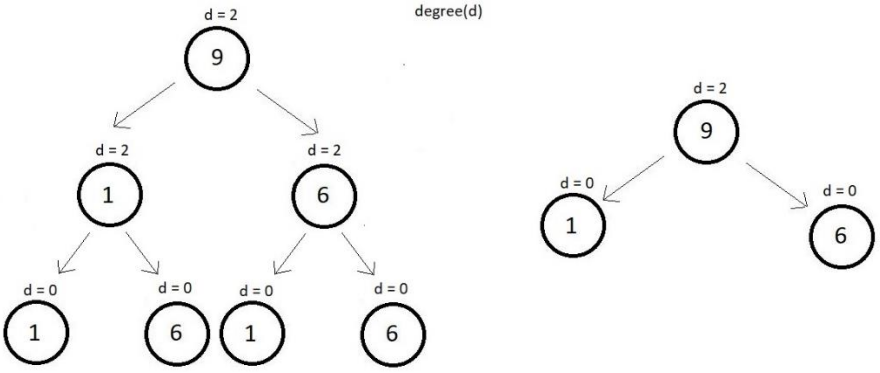


Below illustrated is a binary which is not a strict or full binary tree because the colored node has a degree of just 1.

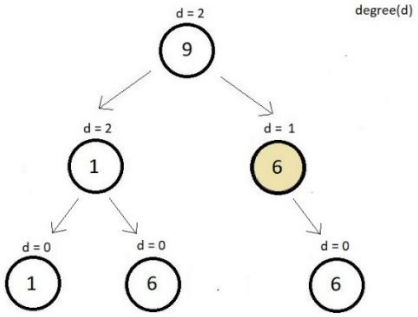


❖ Perfect Binary Tree:

A perfect binary tree has all its internal nodes with degree strictly 2 and has all its leaf nodes on the same level. A perfect binary tree as the name suggests appears exactly perfect. Few examples follow:

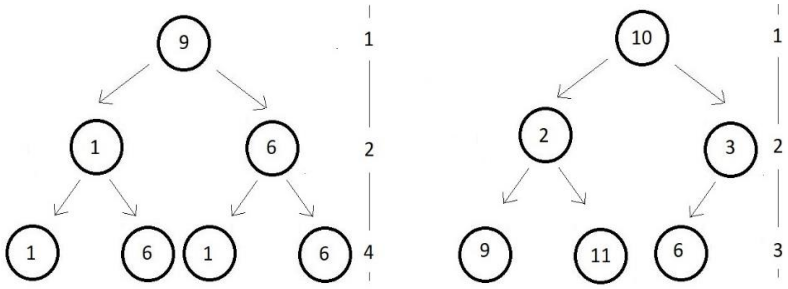


Every leaf node above in both the examples are on the same level and all the internal nodes have a degree 2. Below illustrated is a binary which is not a perfect binary tree because the colored node is an internal node and has a degree of just 1, although each leaf node is on the same level.

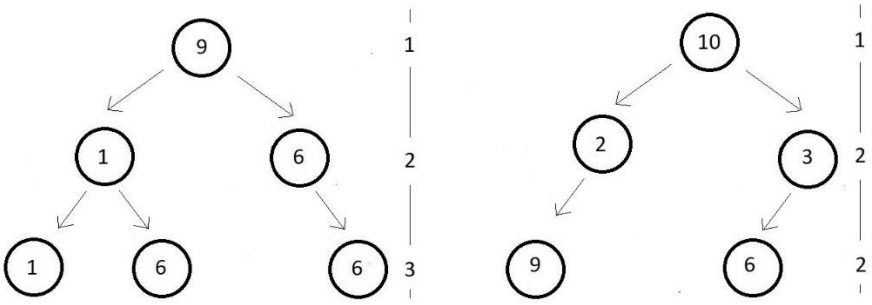


❖ Complete Binary Tree:

A complete binary tree has all its levels completely filled except possibly the last level. And if the last level is not completely filled then the last level's keys must be all left-aligned. It must have sounded tough to figure out. But the illustrations below would clear all our confusion.



As indicated in figure 1, all levels are completely filled, so nothing further needs to be done. It is a complete binary tree. In figure 2, all nodes are completely filled except the last level which has just 3 keys. It is nonetheless a complete binary tree because all keys are left-aligned.



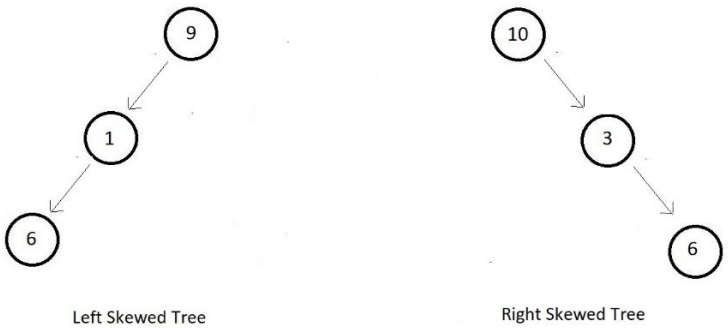
In both the figures, the last level is not complete. And hence we check if all the nodes are aligned to the left. But they aren't in both cases. And hence both of them are not complete.

❖ Degenerate Tree:

The easiest of all, degenerate trees are binary trees where every parent node has just one child and that can be either to its left or right. Few of its examples:

❖ Skewed Tree:

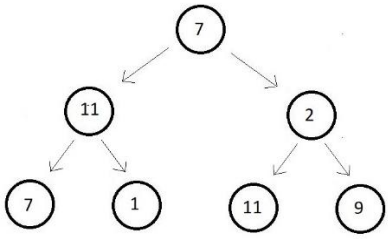
Skewed trees are binary trees where every parent node has just a single child and that child should be strict to the left or to the right for all the parents. Slewed trees having all the child nodes aligned to the left are called left-skewed trees, and skewed trees having all the child nodes aligned to the right are called right-skewed trees. Examples of both left and right-skewed trees are given below.

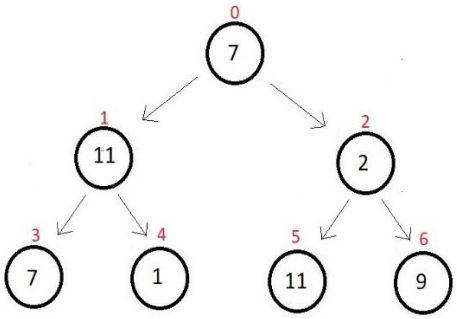


Array representation of Binary Trees:

Arrays are linear data structures and for arrays to function, their size must be specified before elements are inserted into them. And this counts as the biggest demerit of representing binary trees using arrays. Suppose we declare an array of size 100, and after storing 100 nodes in it, we cannot even insert a single element further, regardless of all the spaces left in the memory. Another way we'd say is to copy the whole thing again to a new array of bigger size but that is not considered a good practice.

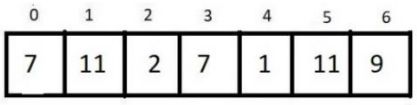
Anyways, we will use an array to represent a binary tree. Suppose we have a binary tree with 7 nodes.





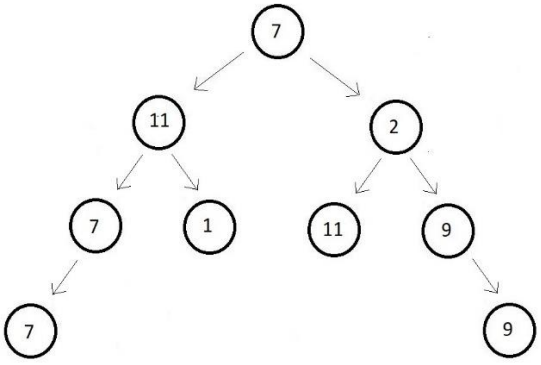
And there are actually a number of ways to represent these nodes via an array. I'll use the most convenient one where we traverse each level starting from the root node and from left to right and mark them with the indices these nodes would belong to.

And now we can simply make an array of length 7 and store these elements at their corresponding indices.

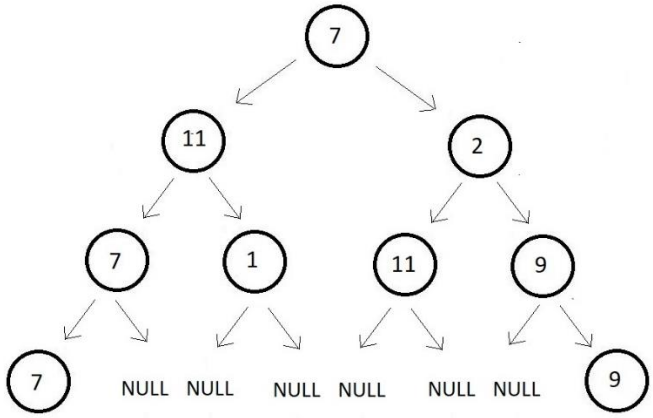


And we might be wondered about the cases where the binary is just not perfect. What if the last level has distributed leaves?

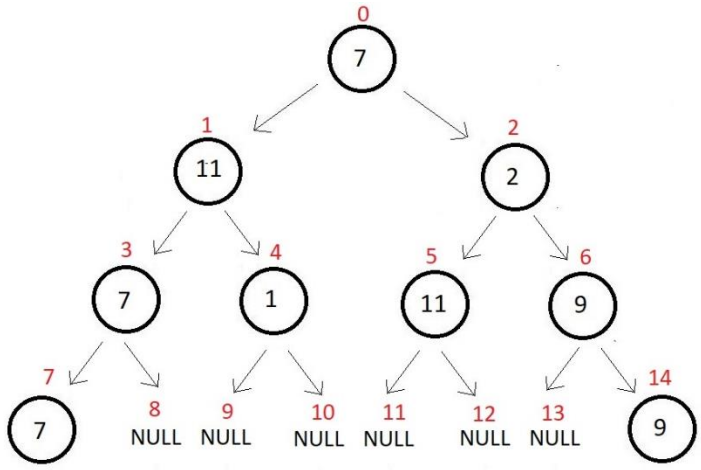
Then, there is a way out for that as well. Let's consider one case here. A binary tree with 9 nodes, and the last two nodes on the extremities of the last level.



Here, while traversing we get stuck at the 8th index. We don't know if declaring the last node as the 8th index element makes it a general representation of the tree or not. So, we simply make the tree perfect ourselves. We first assume the remaining vacant places to be NULL.



And now we can easily mark their indices from 0 to 14.



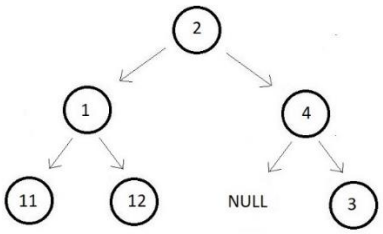
And the array representation of the tree looks something like this. It is an array of length 15.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	11	2	7	1	11	9	7							9

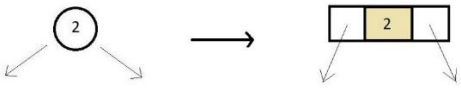
But was this even an efficient approach? Like Binary Trees are made only for efficient traversal and insertion and deletion and using an array for that really makes the process troublesome. Each of these operations becomes quite costly to accomplish. And that size constraint was already for making things worse. So overall, we would say that the array representation of a binary is not a very good choice.

Linked Representation of Binary Trees:

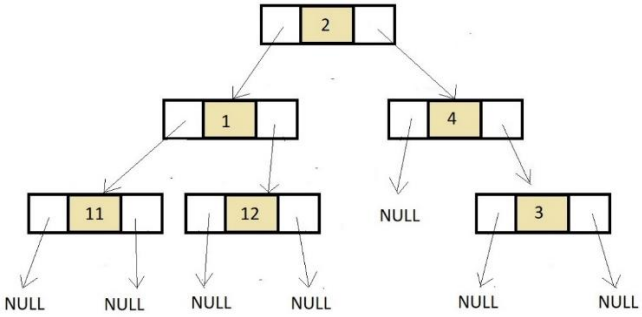
This method of representing binary trees using linked nodes is considered the most efficient method of representation. For this, we use doubly-linked lists. Suppose we have a binary tree of 3 levels.



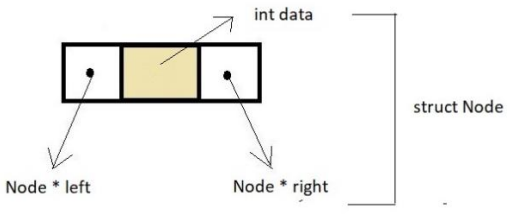
Now if we remember a doubly linked list helped us traversing both to the left and the right. And using that we would create a similar node here, pointing both to the left and the right child node. Follow the below representation of a node here in the linked representation of a binary tree.



We can see how closely this representation resembles a real tree node, unlike the array representation where all the nodes succumbed to a 2D structure. And now we can very easily transform the whole tree into its linked representation which is just how we imagined it would have looked in real life.



So, this was the representation of the binary tree we saw above using linked representation. And what are these nodes? These are structures having three structure members, first a data element to store the data of the node, and then two structure pointers to hold the address of the child nodes, one for the left, and the other for the right.

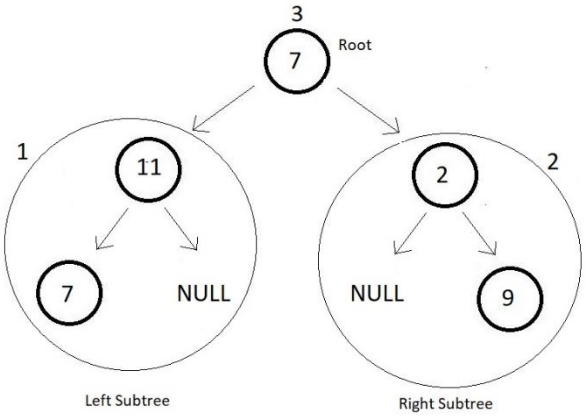


Traversal in Binary Tree:

Basically, we have three different ways to traverse in a binary tree. They are **PreOrder, PostOrder** and **InOrder** Traversals. Let's take a sample tree, and walk through it one by one, using each traversal technique for better understanding.

PreOrder Traversal:

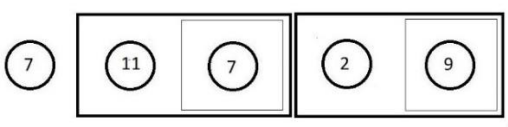
In this traversal technique, the first node we start with is the root node. And thereafter we visit the left subtree and then the right subtree. Taking the above example, we'll mark our order of traversal as below. We first visit section 1, then 2, and then 3.



This was to give us a general idea of the traverse in a binary tree using PreOrder Traversal. Each time we get a tree, first visit its root node, and then move to its left subtree, and then to the right.

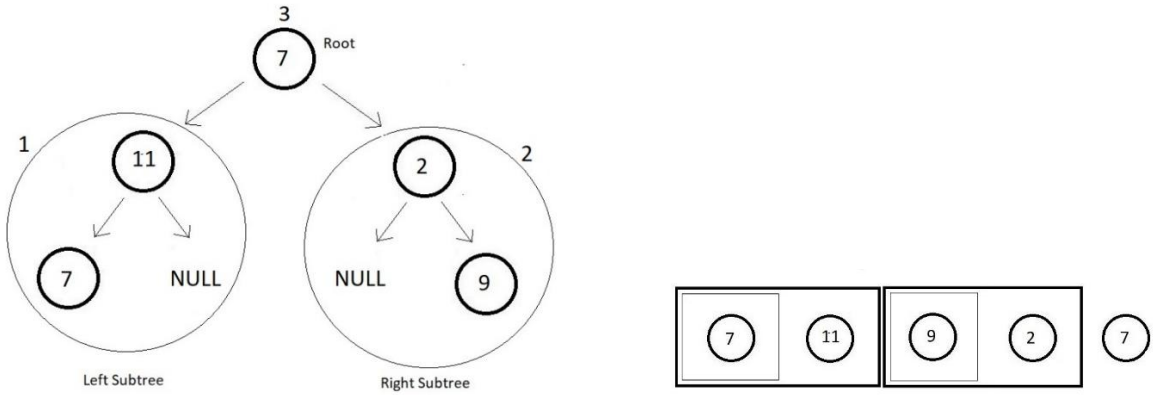
So, here we first visit the root node element 7 and then move to the left subtree. The left subtree in itself is a tree with root node 11. So, we visit that and move further to the left subtree of this subtree. There we see a single element 7, we visit that, and then move to the right subtree which is a NULL. So, we're finished with the left subtree of the main tree. Now, we move to the right subtree which has element 2 as its node. And then a NULL to its left and element 9 to its right.

So basically, we recursively visit each subtree in the same order. And our final traversal order is given below where each block represents a different subtree.



PostOrder Traversal:

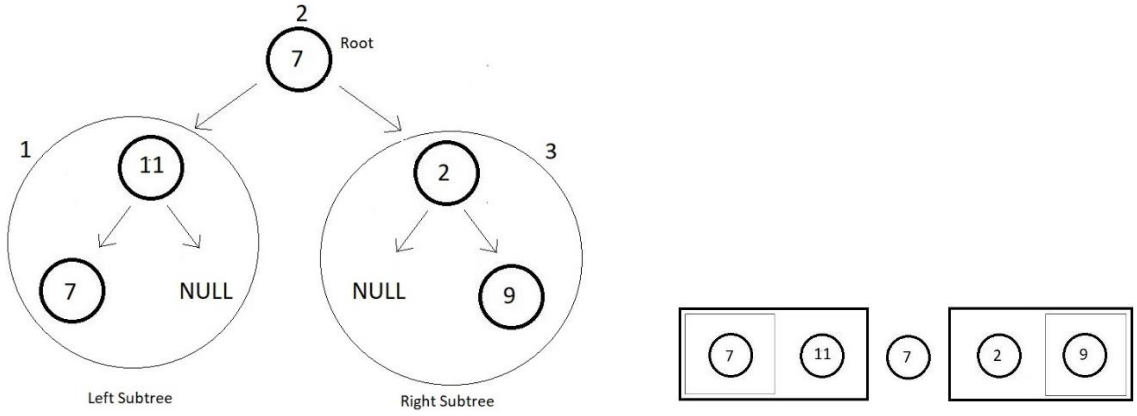
In this traversal technique, things are quite opposite to the PreOrder traversal. Here, we first visit the left subtree, and then the right subtree. So, the last node we'll visit is the root node. Taking the same above example, we'll mark our order of traversal as below. We first visit section 1, then 2, and then 3.



Our final traversal order should be like this. Here, each block represents a different subtree.

InOrder Traversal:

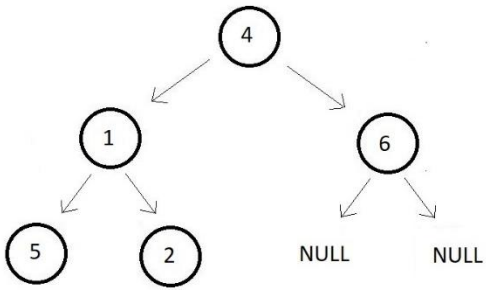
In this traversal technique, we simply start with the left subtree, that is we first visit the left subtree, and then go to the root node and then we'll visit the right subtree. Taking the same above example, we'll mark our order of traversal as below. We first visit section 1, then 2, and then 3.



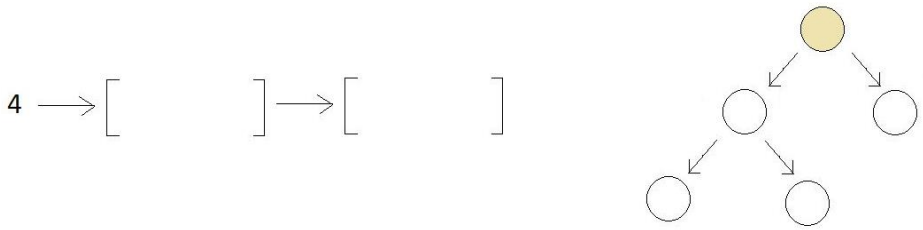
Our final traversal order should be like this. Here, each block represents a different subtree. And these were the traversal techniques we had.

PreOrder Traversal in a Binary Tree:

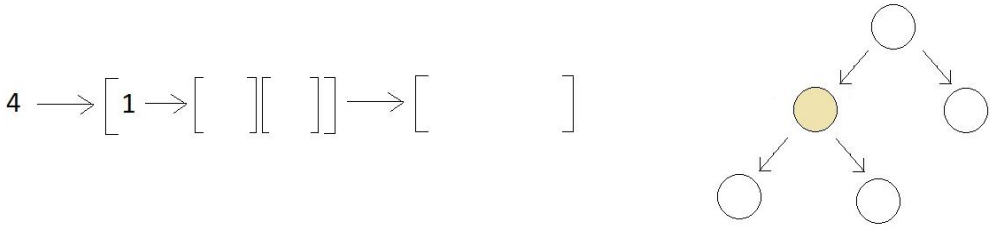
Let's first take an example binary tree, and apply PreOrder Traversal on the same.



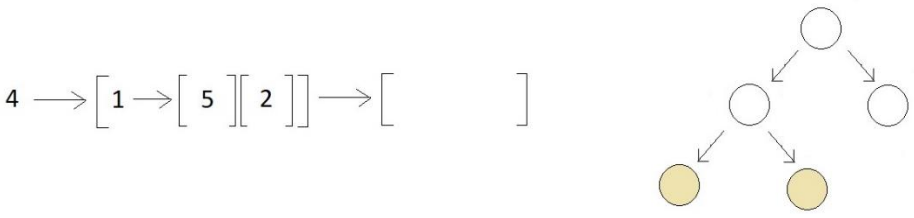
In the first step, we visit the root node and mark the presence of the left and the right subtree as separate individual trees.



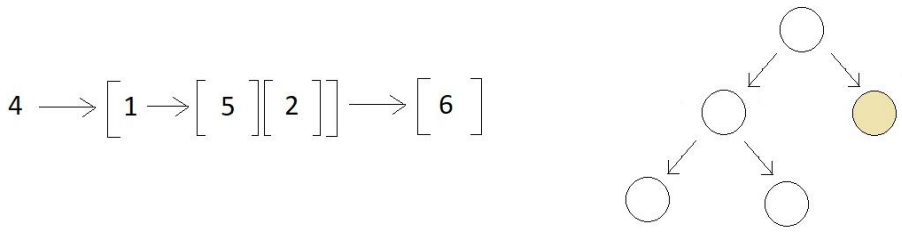
After we visit the root node, we move to the left subtree considering it as a different tree, and start with its root node.



And then we proceed further with the left and right subtrees of this new tree we considered. And since both the left and right subtrees of this tree have just a single element in them, we finish visiting them, and return back to our original tree.

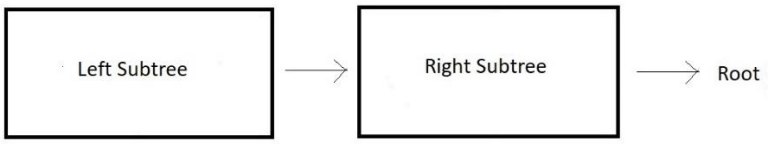


And finally, we visit the right subtree, and since it contains no left or right subtree further, we finish our preorder traversal here itself.

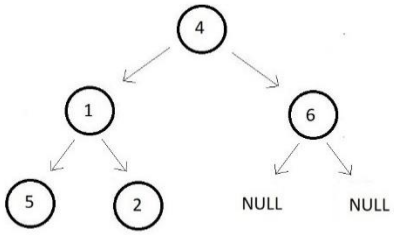


And our final order of preorder traversal is: 4 → 1 → 5 → 2 → 6.

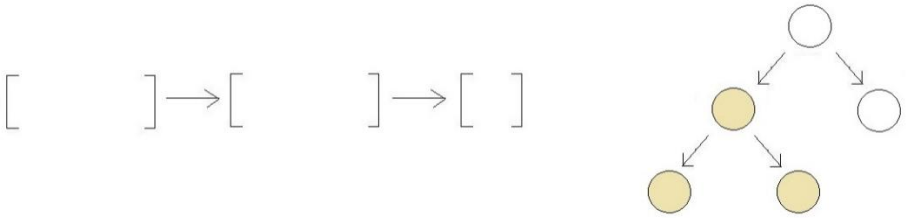
PostOrder Traversal in a Binary Tree:



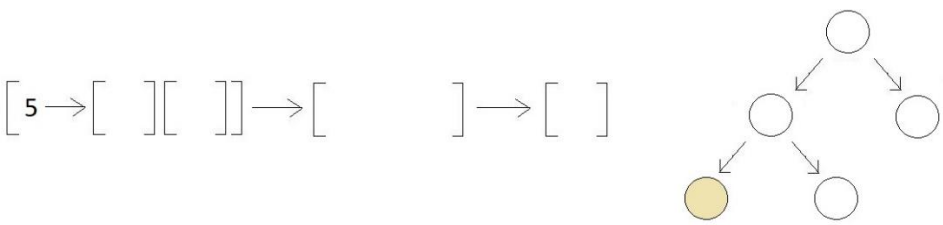
We will take an example binary tree, and apply PostOrder Traversal on the same.



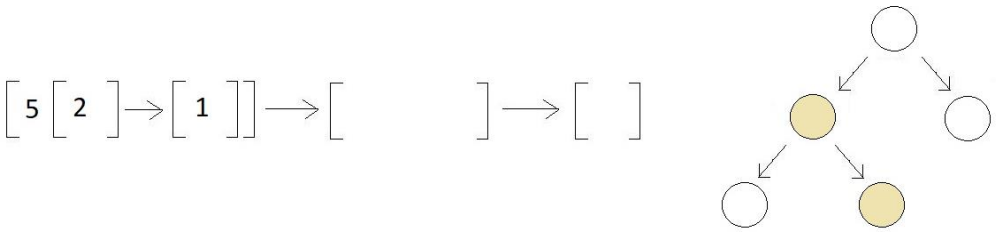
In the first step, we visit the left subtree considering it as a totally different tree. But before we start with the left subtree, we mark the presence of the right subtree and the root as following:



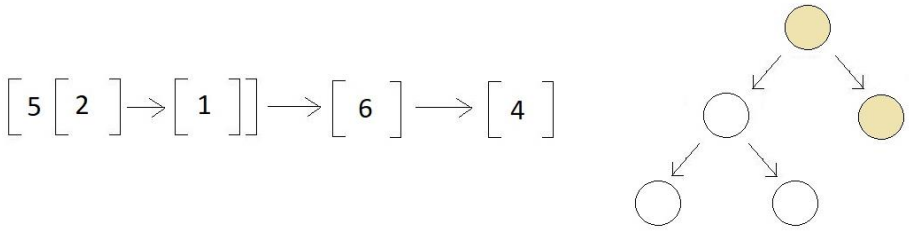
Now, we go through the left subtree, and visit its left node first.



And then we proceed further with the right subtree of this new tree we considered. And since the right subtree of this tree has just a single element, we finish visiting it and then the root of the node, and return back to our original tree.

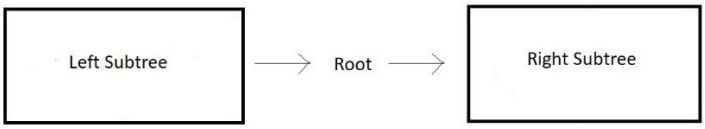


And then, we visit the right subtree, and since it contains no left or right subtree further, we finish visiting our right subtree and then move to the root. And there we mark our completion of PostOrder traversal.

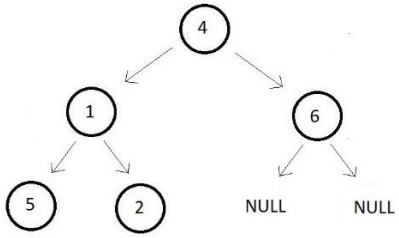


And our final order of postOrder traversal is: 5 → 2 → 1 → 6 → 4.

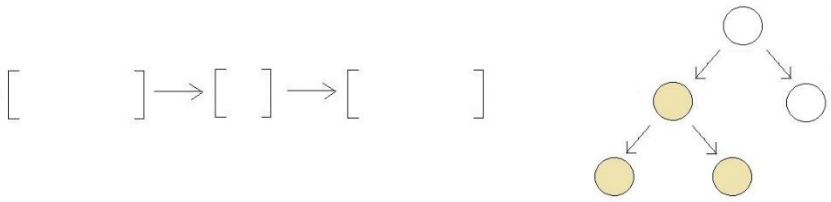
InOrder Traversal in a Binary Tree:



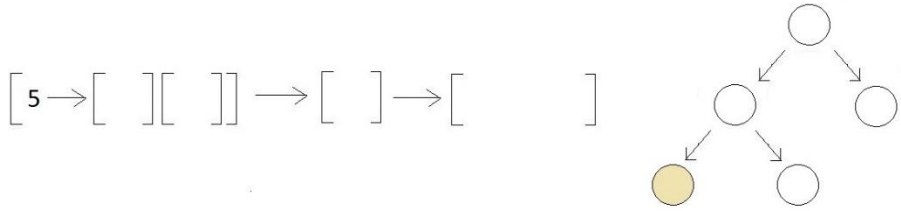
We will take an example binary tree, and apply InOrder Traversal on the same.



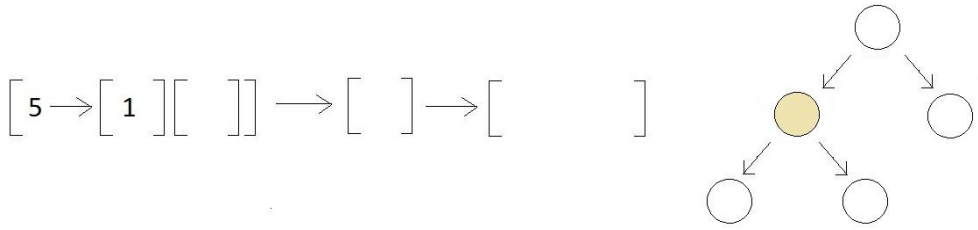
In the first step, we consider the left subtree as a completely different one and apply InOrder separately to it. But before we start with the left subtree, make sure to mark the presence of the root node and the right subtree as following:



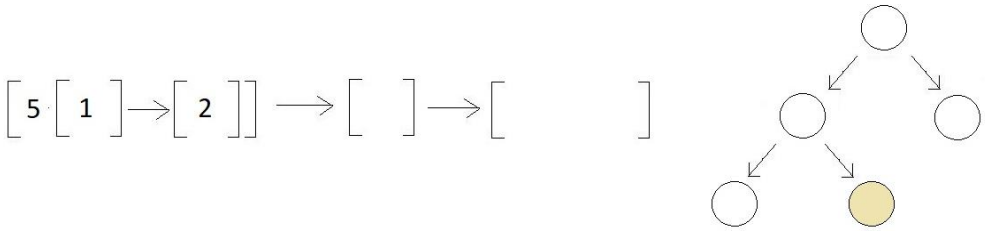
Now, we go through the left subtree, and further visit the left subtree of this new tree first.



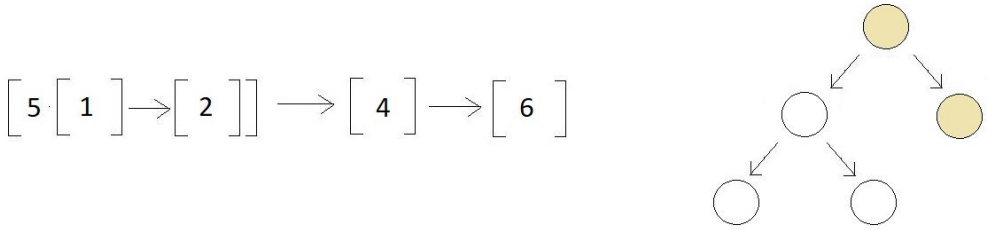
And then we proceed with the root of this new tree we considered.



And after that, we visit the right subtree of this tree which has just a single element.



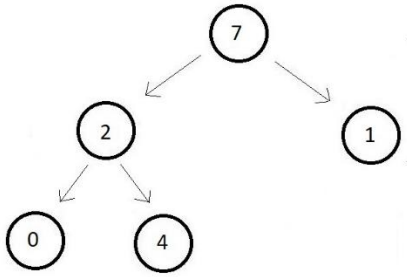
With that being visited we move back to the original tree. And here, we visit the root node first and then the right subtree, and since it contains no left or right subtree further, we finish visiting our right subtree. There we mark the end of our InOrder traversal.



And our final order of inorder traversal is: 5 → 1 → 2 → 4 → 6.

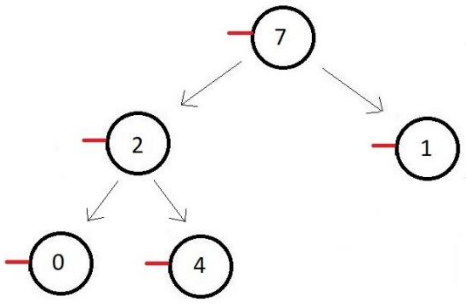
Best Trick to Find PreOrder, InOrder & PostOrder Traversal:

We would like to provide an example of a Binary Tree and then traverse through it using different techniques.

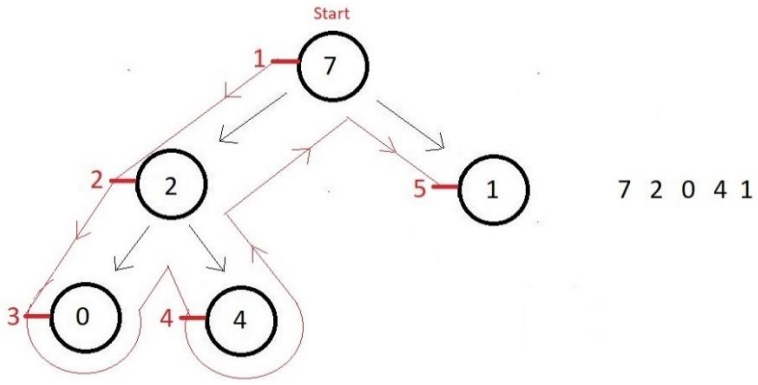


PreOrder Traversal:

In this method, we start with the node, and then move to the left subtree and then to the right subtree. But the trick says, extend an edge to the left of all the nodes. Follow the figure below to understand what is being said.



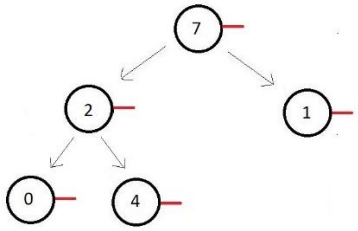
And now, start driving from the root to the left of the root node, and whenever we intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been made to direct us to the path we have followed.



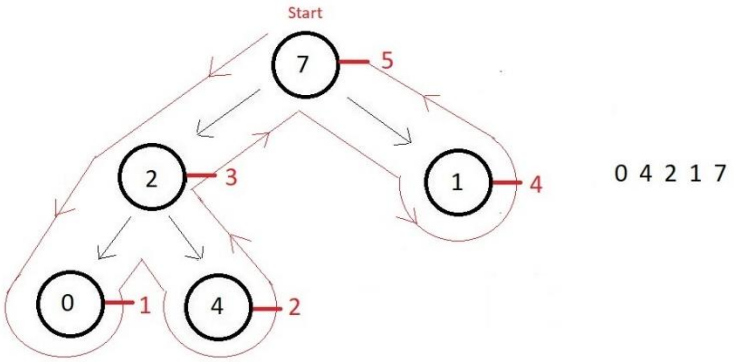
And this is how we were quickly able to write the order of nodes we visited first in the PreOrder Traversal. It was 7→2→0→4→1.

PostOrder Traversal:

In this method, we start with the left subtree first, and then move to the right subtree, and then finally to the root node. But the trick illustrates, extending an edge to the right of all the nodes. Earlier it was left, now it is right. Follow the figure below to understand.



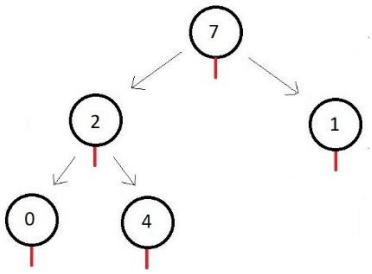
And now, drive the same way we did earlier. Start driving from the root to the left of the root node and whenever we intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been drawn to direct us to the path we have taken.



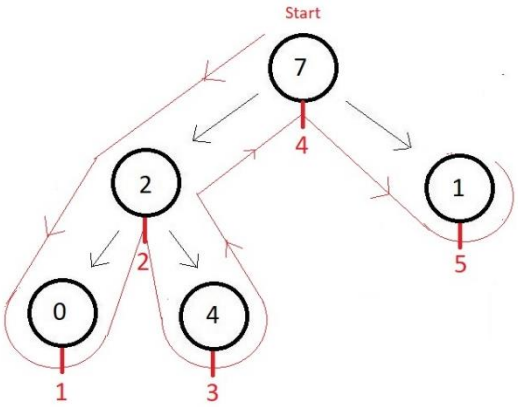
And this is how we were quickly able to write the order of nodes we visited first in the PostOrder Traversal. It was 0→4→2→1→7.

InOrder Traversal:

In this method, we start with the left subtree first, and then move to the root node and then finally to the right subtree. But the trick says, extend an edge to the bottom of all the nodes. We went to the left, to the right, and now to the bottom. Follow the figure below to understand.



Then, drive as we have done all along. Start driving from the root to the left of the root node and whenever we intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been drawn to direct us to the path we have taken.



0 2 4 7 1

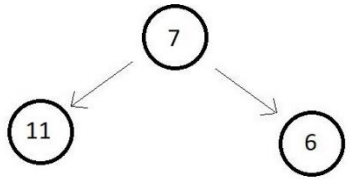
And this is how we were quickly able to write the order of nodes we visited first in the InOrder Traversal. It was 0→2→4→7→1.

Binary Search Trees:

Following are the properties of a binary search tree:

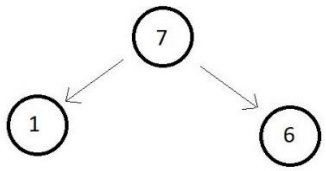
1. It is a type of binary tree.
2. All nodes of the left subtree are lesser than the node itself.
3. All nodes of the right subtree are greater than the node itself.
4. Left and Right subtrees are also binary trees.
5. There are no duplicate nodes.

Figure 1:



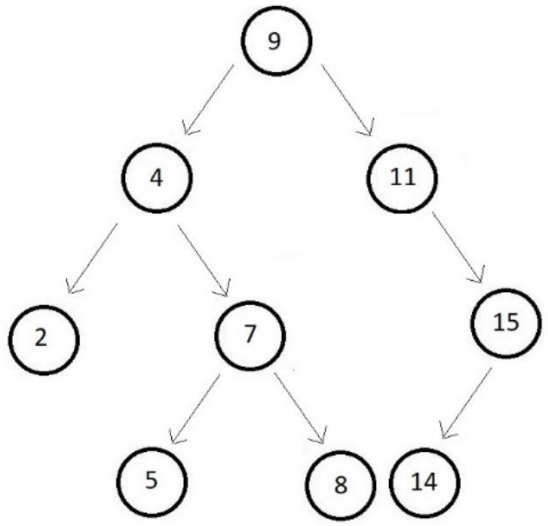
Since the left subtree of the root node has a single element that is greater than the root node violating the 2nd property, it is not a binary search tree.

Figure 2:



Tell if this is a binary search tree. Still NO. Because the left subtree is good but the right subtree of the root node is lesser than the root node itself violating the 3rd property.

Figure 3: Is this a binary search tree or not. YES. Why?

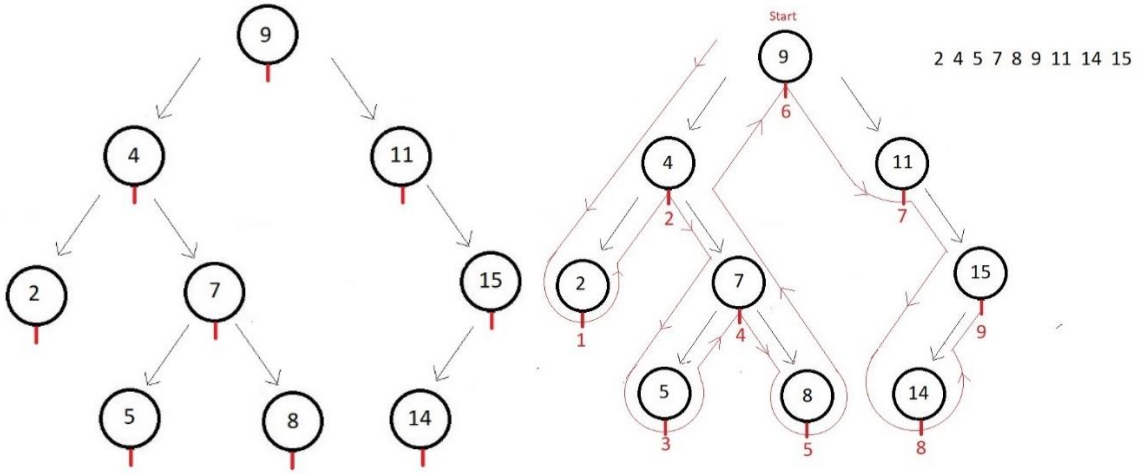


The very first thing to observe here is the properties of a Binary Search Tree. We would check if all the properties are satisfied for each of the nodes of the tree. So, we first start with the root node which is element 9 and see if all the nodes on the left subtree {4, 2, 5, 7} are smaller than 9 and all the nodes of the right subtree {11, 15, 14} are greater than 9. And since they are, we'll proceed with the next node. Doing this for all the nodes, we'll conclude that this is a Binary Search Tree.

Lastly, there is one more amazing property that says the **InOrder traversal of a binary search tree gives an ascending sorted array**. So, this is one of the easiest ways to check if a tree is a binary search tree.

Let's write the InOrder Traversal of the tree in figure 3.

- We'll use the same technique we learned in the last lecture to quickly find the InOrder traversal. We'll extend an edge to the bottom of each node.
- Then, we'll start from the root node, and drive towards the left subtree and follow the arrows I've drawn for our convenience.



So, the final InOrder traversal order of the above tree is:

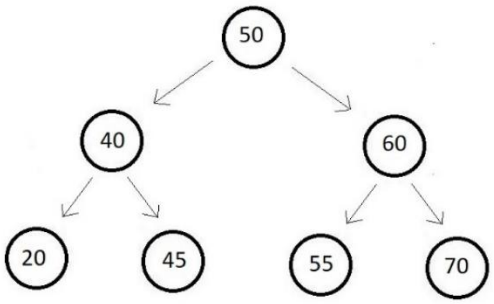
$$2 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 14 \rightarrow 15,$$

which is obviously in increasingly sorted order. Hence, it is a binary search tree.

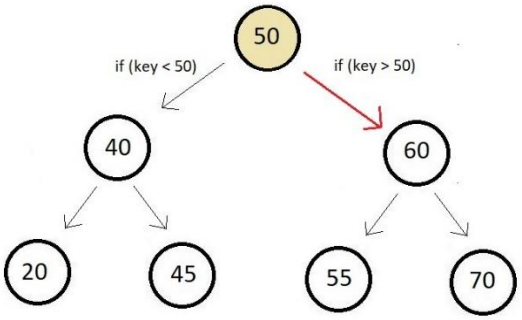
Searching in a Binary Search Trees (Search Operation):

One of the major applications of using a binary search tree, is to be able to search some key in the tree in **logn** time complexity in the best case where n is the number of nodes. Each time we compare a node with our key, we divide the search space to its half. But let's not proceed without an example.

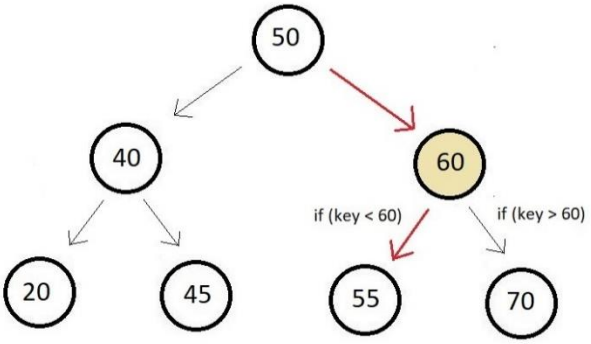
For our purpose of understanding, we will first examine a sample binary search tree. Suppose we have a Binary Search Tree illustrated below.



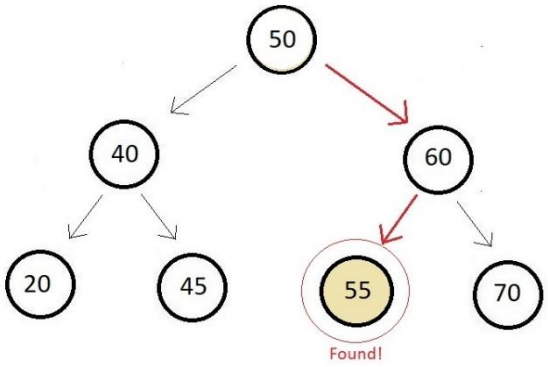
And let's say the key we want to search in this binary search tree is 55. Let's start our search. So, we'll first compare our key with the root node itself, which is 50.



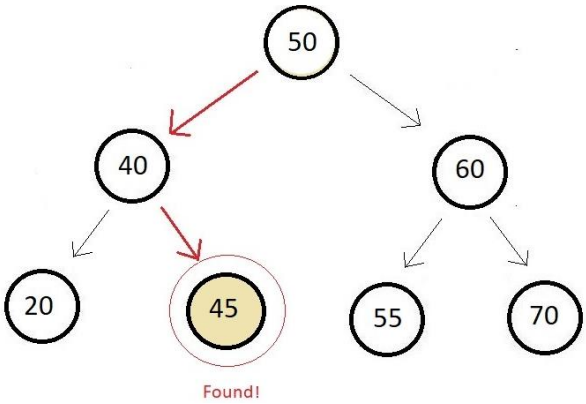
But since 50 is less than 55, which side should we proceed with? Left or Right? Of course, right. Since all the elements in the right subtree of a node are greater than that node, we'll move to the right. The first element we check our key with is 60.



Now, since our key is smaller than 60, we'll move to the left of the current node. The Left subtree of 60 contains only one element and since that is equal to our key, we revert the positive result that yes, the key was found. Had this leaf node been not equal to the key, and since there are no subtrees further, we would have stopped here itself with negative results, saying the key was not found.



Let's see for one more key which is 45. Now, we'll directly illustrate the path we followed and whether it was found or not.



And yes, key 45 was found too. The path we followed is colored red. We went to 50 and found it smaller so moved to its left. Then we found 40, and since our key was greater than that, we moved to its right, and the leaf node we found was equal to 45 only.

Time Complexity of the Search Operation in a Binary Search Tree:

Searching in a binary search tree holds **O(logn)** time complexity in the best case where n is the number of nodes making it incredibly easier to search an element in a binary search tree, and even operations like inserting get relatively easier.

Let's calculate exactly what happens. If we could see the above examples, the algorithm took the number of comparisons equal to the height of the binary search tree, because at each comparison we stepped down the depth by 1. So, the time complexity $\mathbf{T \propto h}$, that is, our time complexity is proportional to the height of the tree. Therefore, the time complexity becomes **O(h)**.

Now, if we remember, the height of a tree ranges from logn to n,

that is **(logn) ≤ h ≤ n**

So, the best-case time complexity is **O(logn)** and the worst-case time complexity is **O(n)**.

Pseudocode for searching in a Binary Search Tree:

- There will be a struct node pointer function search which will take the pointer to the root node and the key we want to search in the tree. And before we do anything, just check if the root node is not NULL. If it is, return NULL here itself. Otherwise, proceed further.
- Now, check if the node we are at is the one we were looking for. If it is, return that node. And that would be it. But if that is not the one, just see if that key is greater than or less than that node. If it is less, then return recursively to the left subtree, otherwise to the right subtree. And that is all.

Pseudocode for the search function:

```
Node * search(node* root, key){
   if(root==NULL){
      return NULL;
   }
   if(key==root->data){
      return root;
   }
   else if(key<root->data){
      return search(root->left, key);
   }
   else{
      return search(root->right, key);
   }
}
```

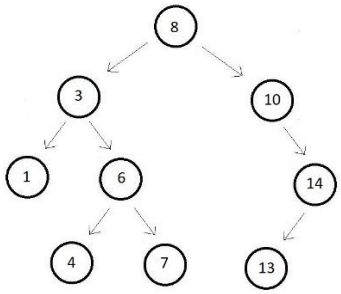

Insertion in a Binary Search Tree:

We know a few facts about binary search trees and we'll only mention the ones we will focus on, while we learn the insertion operation.

There are no duplicates in a binary search tree. So, if we could search the element we are being asked to insert, we would return that the number already exists.

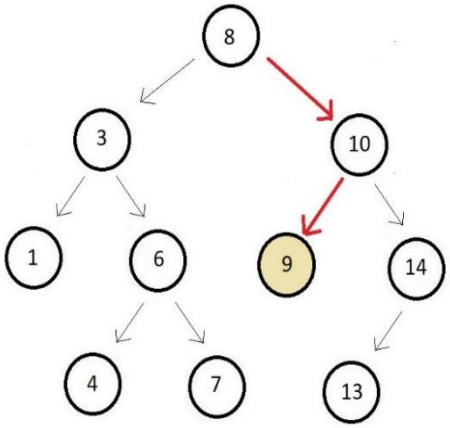
Now, we would follow what we did in the search operation.

Here is an example binary search tree, and the element we want to insert is 9.



Now, we would simply start from the root node, and see if the element we want to insert is greater than or less than.

And since 9 is greater than 8, we move to the right of the root. And then the root is the element 10, and since this time 9 is less than 10, we move to the left of it. And since there are no elements to its left, we simply insert element 9 there.

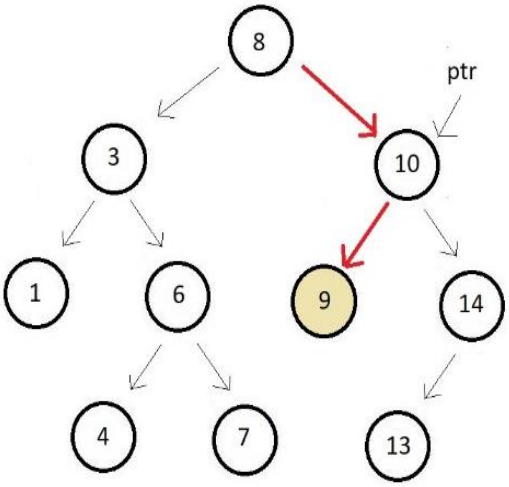


This was one simple case, but things become more complex when we have to insert our element at some internal position and not at the leaf.

Now, before we insert a node, the first thing we would do is to create that node and allocate memory to it in heap using malloc. Then we would initialize the node with the data given, and both the right and the left member of the node should be marked NULL.

And another important thing to see here is the pointer we would follow the correct position with. In the above example, to be able to insert at that position, the pointer must be at node 10.

And then we check whether going to the left side is good, or the right. Here we came to the left, but had it been right, we would have updated our pointer ptr further and maintained a second pointer to the previous root.



Deletion in a Binary Search Tree:

Whenever, we talk about deleting a node from binary search tree, we have the following three cases in mind:

a. The node is a leaf node.

b. The node is a non-leaf node.

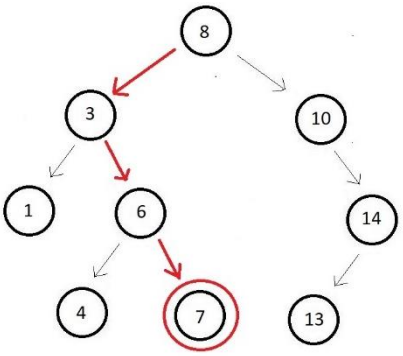
c. The node is the root node.

Let's deal with each of these in detail, starting with deleting the leaf node.

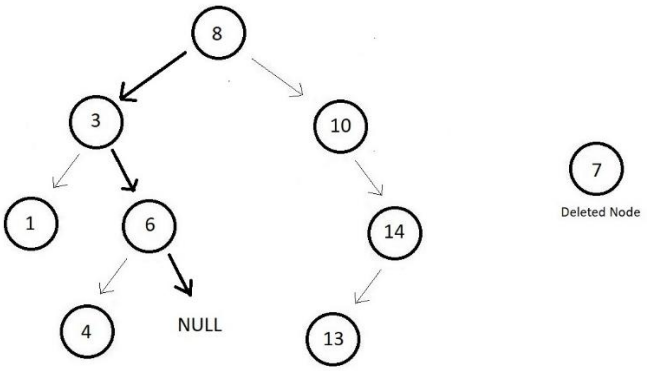
❖ Deleting a Leaf Node:

Deleting a leaf node is the simplest case in deletion in binary search trees where the only thing we have to do is to search the element in the tree, and remove it from the tree, and make its parent node point to NULL. To be more specific, follow the steps below to delete a leaf node along with the illustrations of how we delete a leaf node in the above tree:

1. Search the node.



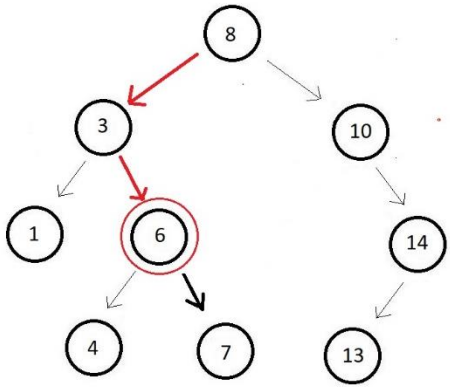
2. Delete the node.



❖ Deleting a Non-leaf Node:

Now suppose the node is not a leaf node, so we cannot just make its parent point to NULL, and get away with it. We have to even deal with the children of this node. Let's try deleting node 6 in the above binary search tree.

So, the first thing we would do is to search element 6.

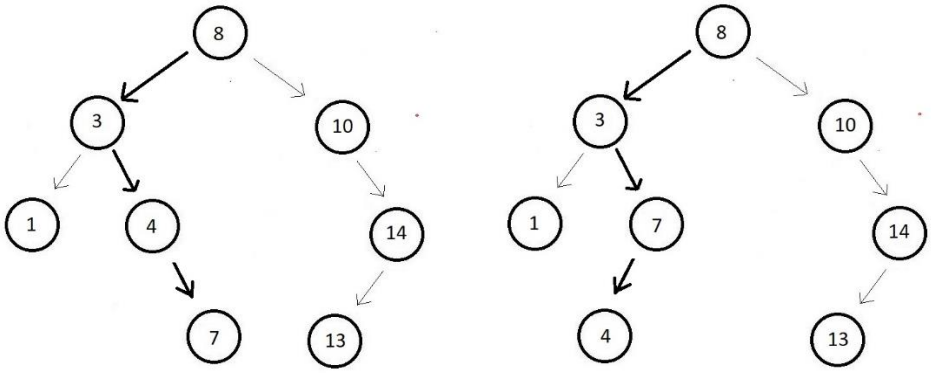


Now the dilemma is, which node will replace the position of node 6. Well, there is a simple answer to it. It says, when we delete a node that is not a leaf node, we replace its position with its InOrder predecessor or InOrder successor.

It means that if we write the InOrder traversal of the above tree, the nodes coming immediately before or after node 6, will be the one replacing it. So, if we write the InOrder traversal of the tree, we will get:

$$1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 13 \rightarrow 14$$

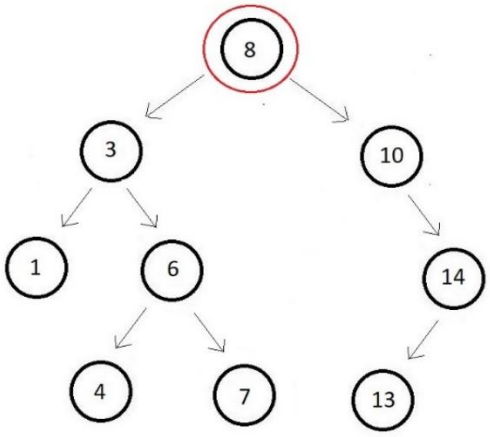
So, the InOrder predecessor and the InOrder successor of node 6 are 4 and 7 respectively. Hence, we can substitute node 6 with any of these nodes, and the tree will still be a valid binary search tree. Refer to how it looks below.



So, both are still binary search trees. In the first case, we replaced node 6 with node 4. And the right subtree of node 4 is 7, which is still bigger than it. And in the second case, we replaced node 6 with node 7. And the left subtree of node 7 is 4, which is still smaller than the node. Hence, a win-win for us.

❖ Deleting the Root Node:

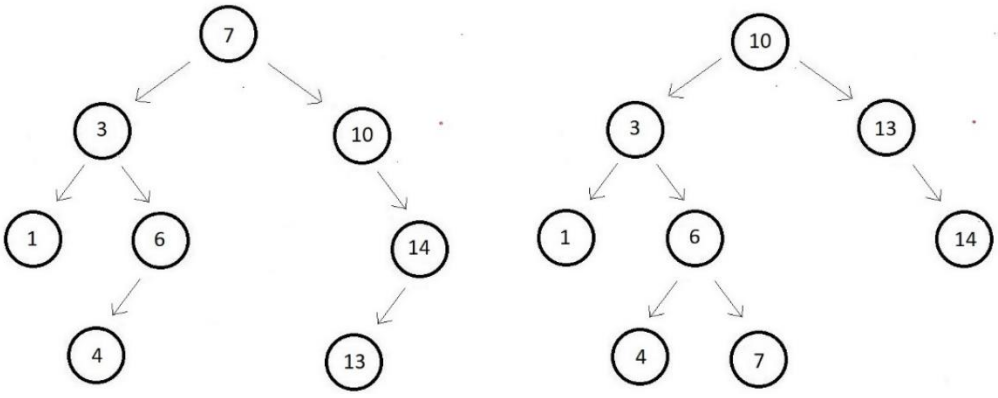
Now, if we carefully observe, the root node is still another non-leaf node. So, the basics to delete the root node remains the same as what we did for a general non-leaf node. But since the root node holds a big size of subtrees along with, we have put this as a separate case here.



So, the first thing we do is write the InOrder traversal of the whole tree. And then replace the position of the root node with its InOrder predecessor or Inorder successor. So, here the traversal order is,

$$1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 13 \rightarrow 14$$

So, the InOrder predecessor and the Inorder successor of the root node 8 are 7 and 10 respectively. Hence, we can substitute node 8 with any of these nodes, but there is a catch here. So, if we substitute the root node here, with its InOrder predecessor 7, the tree will still be a binary search tree, but when we substitute the root node here, with its InOrder successor 10, there still becomes an empty position where node 10 used to be. So, we still placed the InOrder successor of 10, which was 13 on the position where 10 used to be. And then there are no empty nodes in between. This finalizes our deletion.



So, there are a few steps:

1. First, search for the node to be deleted.
2. Search for the InOrder Predecessor and Successor of the node.
3. Keep doing that until the tree has no empty nodes.

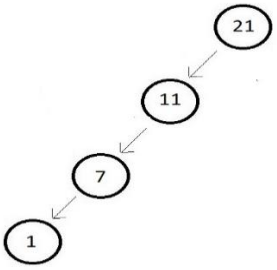
And this case is not limited to the root nodes, rather any nodes falling in between a tree. Well, there could be a case where the node was not found in the tree, so, for that, we would revert the statement that the node could not be found.

AVL Trees - Introduction:

Well, the operations we have discussed lately have been observed to work faster when the tree is highly distributed, or where the height is actually loose to logn and the complexity tends to O(logn), but they come close to O(n) when our tree becomes sparse, and looks unnecessarily lengthened. This is where AVL trees come to the rescue.

We'll take an example to understand the concept. Suppose we're told to create a Binary Search Tree out of some elements given to us. Suppose the numbers were {1, 11, 7, 21} and we simply sort the elements and write them as shown in the figure below.

We'll take an example to understand the concept. Suppose we're told to create a Binary Search Tree out of some elements given to us. Suppose the numbers were {1, 11, 7, 21} and we simply sort the elements and write them as shown in the figure.

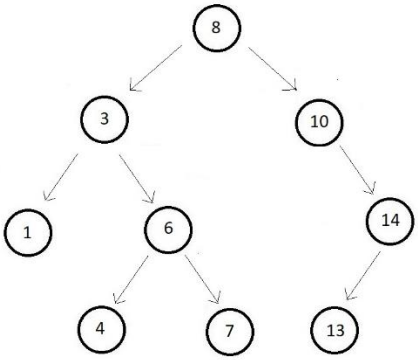


And to be honest, we cannot deny the fact that this is indeed a binary search tree, though skewed. There is no violation of any of the rules of a binary search tree. But any operation here has a complexity O(n).

AVL Tree is needed because:

a. Almost all the operations in a binary search tree are of order O(h) where h is the height of the tree.
b. If we don't plan our trees properly, this height can get as high as n where n is the number of nodes in the Binary Search Tree (Skewed tree).
c. So, to guarantee an upper bound of O(logn) for all these operations we use balanced trees.

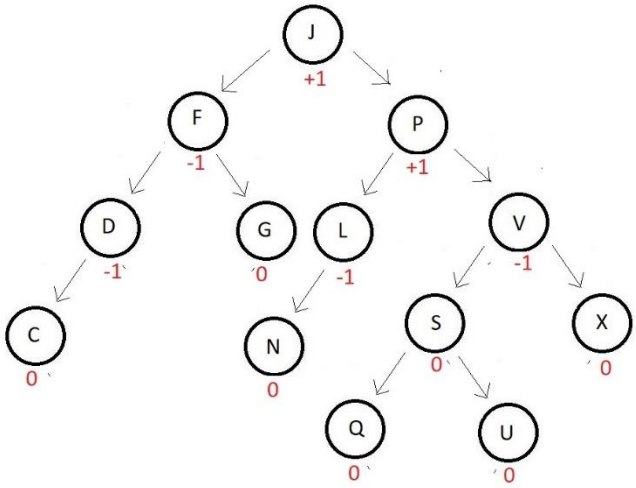
This is actually very practical. Because when a binary search tree takes the form of a list, our operations, say searching, starts taking more time. Consider the Binary search tree below.



To search 1, in this binary search tree, we would need only 3 operations, while to search in a list type binary search tree having the same elements, this would have taken 9 operations.

What are AVL Trees?

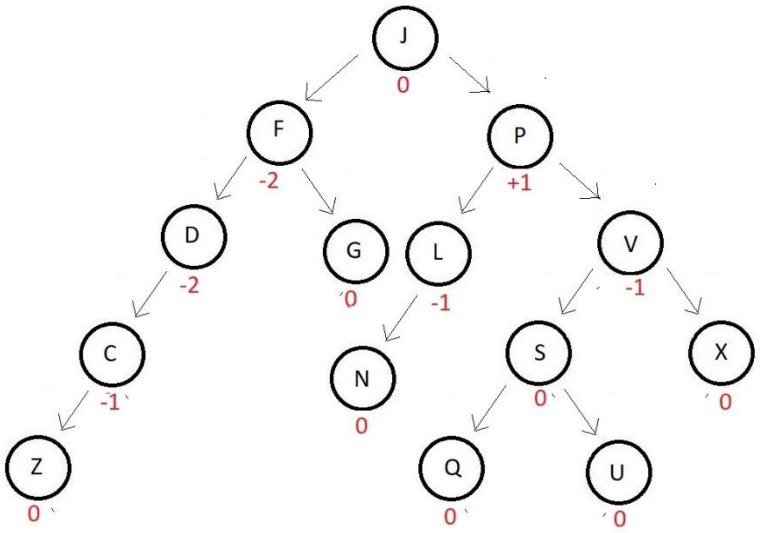
- AVL trees are height balanced binary search trees. Because most of the operations work on O(h), we would want the value of h to be minimum possible, which is log(n).
- Height difference between the left and the right subtrees is less than 1 or equal in an AVL tree.
- For AVL trees, there is a balance factor BF, which is equal to the height of the left subtree subtracted from the height of the right subtree. If we consider the below binary search tree, we can see the balance factor mentioned beside each node. Carefully observe each of those.



We can see, none of the nodes above have a balance factor more than 1 or less than -1. So, for a balance tree to be considered an AVL tree, the value of |BF| should be less than or equal to 1 for each of the nodes, that is, $|BF| \leq 1$.

- And even if some of the nodes in a binary search tree have a |BF| less than or equal to 1, those nodes are considered balanced. And if all the nodes are balanced, it becomes an AVL.

One thing before we finish. An AVL tree gets disturbed sometime when we try inserting a new element in it. For example, in the above AVL tree, if we try inserting an element Z at the end of the leftmost element, the balanced factor gets updated for each of the nodes following above. And the tree is no more an AVL tree. See the updated tree below.



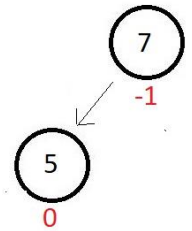
And to avoid this unbalancing, we have an operation called rotation in AVL trees. This helps maintain the balancing of nodes even after a new element gets inserted.

Insertion and Rotation in AVL Tree:

Before we proceed to see what different types of rotation in an AVL tree we have, we would first like to know why rotation is even done.

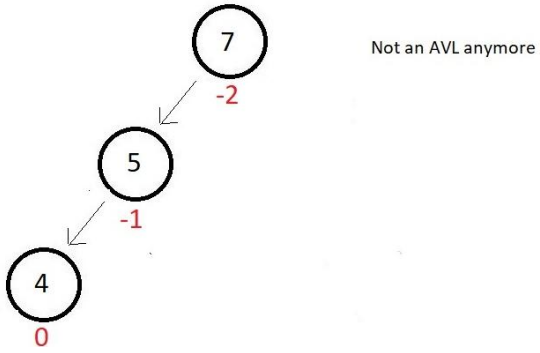
In an AVL tree, rotations are performed to avoid the unbalancing of a node caused by insertion.

Now, suppose we have a small AVL tree having just these two nodes.



An AVL tree

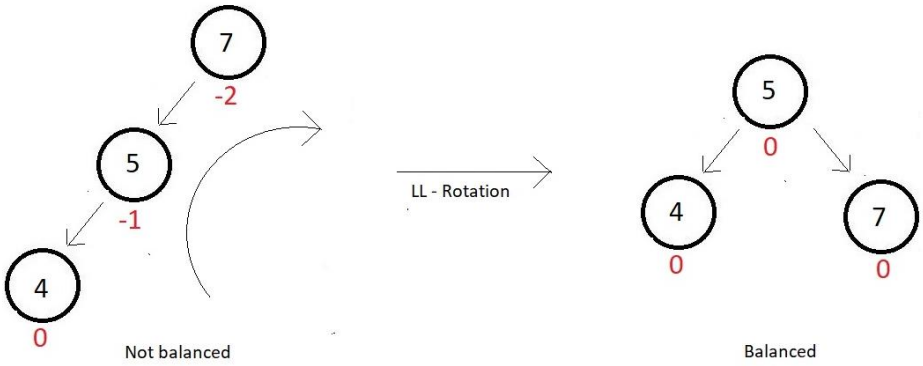
And we can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 4, our updated tree becomes unbalanced to the left. The absolute balance factor of node 7 becomes greater than 1.



So, the method we will follow to make this tree an AVL again is called rotation. Now, rotations can be of different types, one of them being the LL rotation.

LL Rotation:

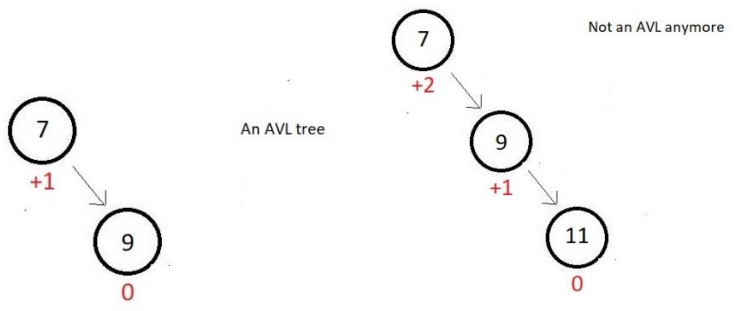
The name LL, just because we inserted the new element to the **left subtree of the root.** In this rotation technique, we just simply rotate our tree one time in the clockwise direction as shown below.



So, our tree got balanced again, with a perfect balance factor at each of its nodes. Next, we have the RR rotation.

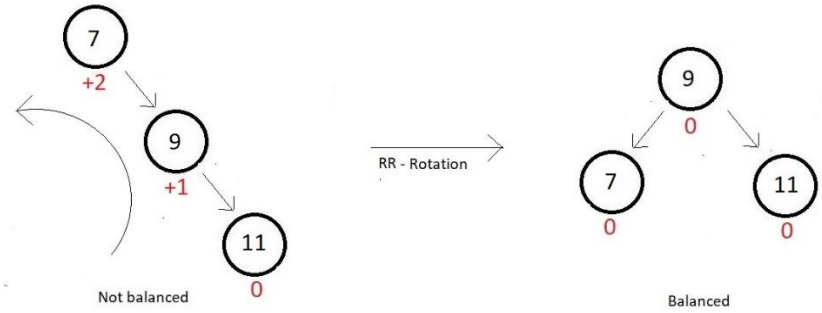
RR Rotation:

Now, suppose we have a small AVL tree having just these two nodes.



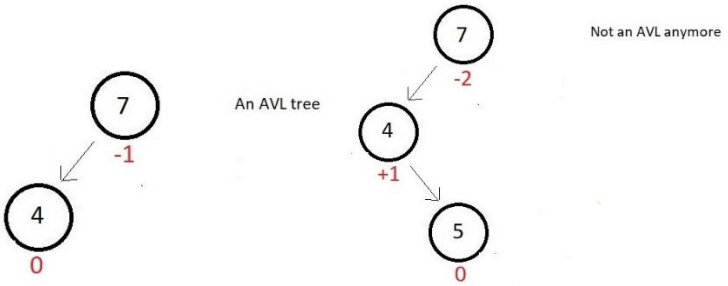
And we can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 11, our updated tree becomes unbalanced to the right. The absolute balance factor of node 7 becomes greater than 1.

So, the method we will follow now to make this tree an AVL again is called the RR rotation. The name RR, just because we inserted the new element to the **right subtree of the root**. In this rotation technique, we just simply rotate our tree one time in an anti-clockwise direction as shown below.

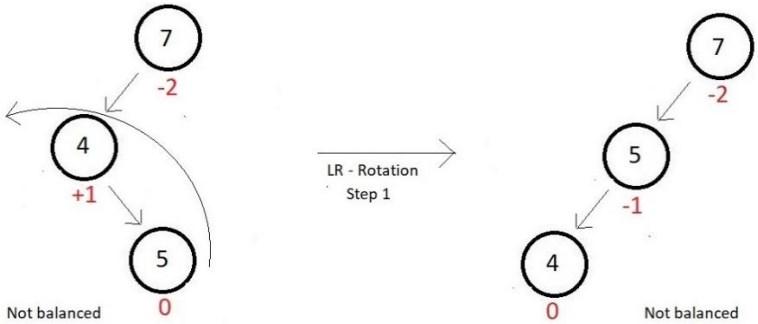


LR Rotation:

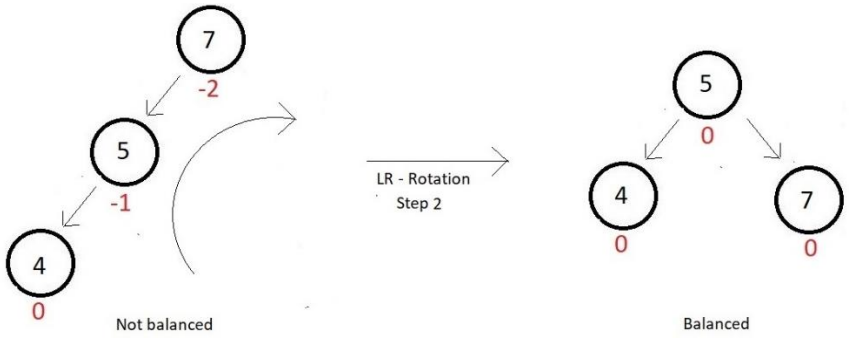
Now, suppose we have a small AVL tree having just these two nodes. And we can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 5, our updated tree becomes unbalanced to the left. The absolute balance factor of node 7 becomes greater than 1.



So, the method we will follow now to make this tree an AVL again is called the LR rotation. The name LR, just because we inserted the new element to the right to the left subtree of the root. In this rotation technique, there is a subtle complexity, which says, first rotate the left subtree in the anticlockwise direction, and then the whole tree in the clockwise direction. Step 1:



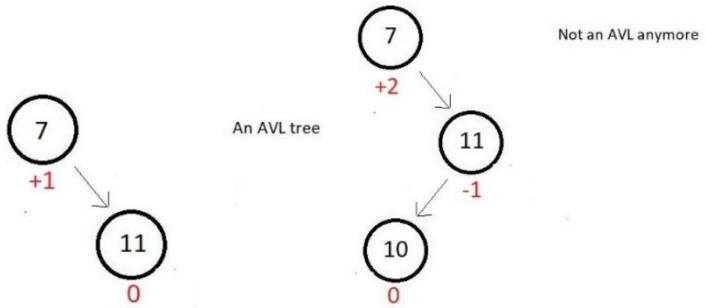
Step 2:



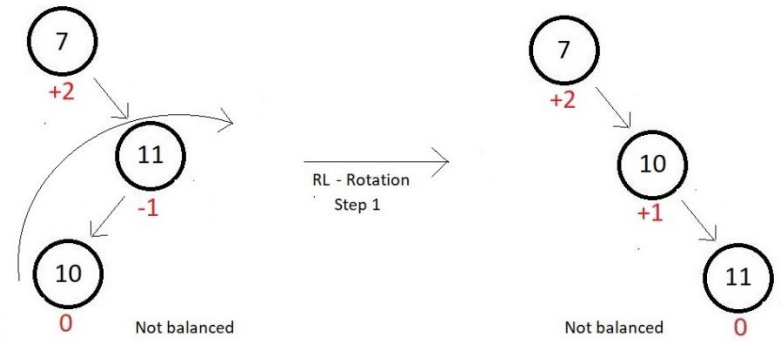
So, our tree got balanced again, with a perfect balance factor at each of its nodes. Although it was a bit clumsy, it was achievable.

RL Rotation:

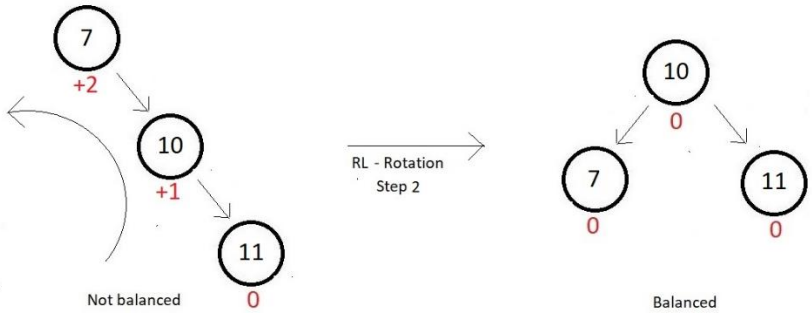
Now, suppose we have a small AVL tree having just these two nodes and we can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 10, our updated tree becomes unbalanced to the right. The absolute balance factor of node 7 becomes greater than 1.



So, the method we will follow now to make this tree an AVL again is called the RL rotation. The name RL, just because we inserted the new element to the left to the right subtree of the root. We follow the same technique we used above, which says, first rotate the right subtree in the clockwise direction, and then the whole tree in the anticlockwise direction. Step 1:



Step 2:



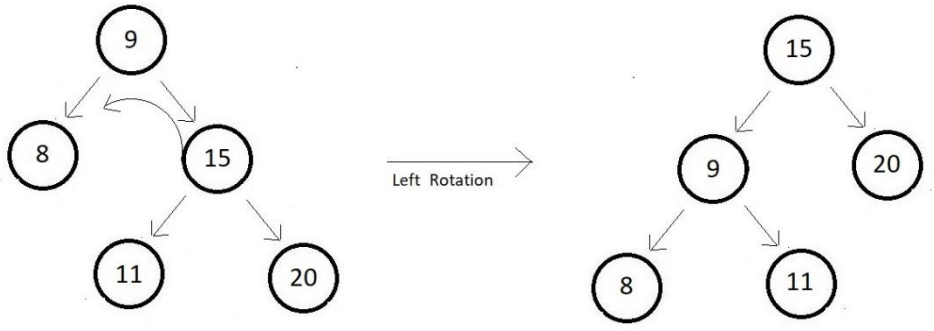
So, our tree got balanced again, with a perfect balance factor at each of its nodes. And those were our different types of rotation techniques that helped gain the balance of our AVL trees back after the insertion of new elements.

AVL Trees - LL LR RL and RR rotations:

We should know how a binary search tree that is unbalanced can be balanced with a few **Rotate Operations**. We can perform Rotate operations to balance a binary search tree such that the newly formed tree satisfies all the properties of a binary search tree. Following are the two basic Rotate operations:

Left Rotate Operations:

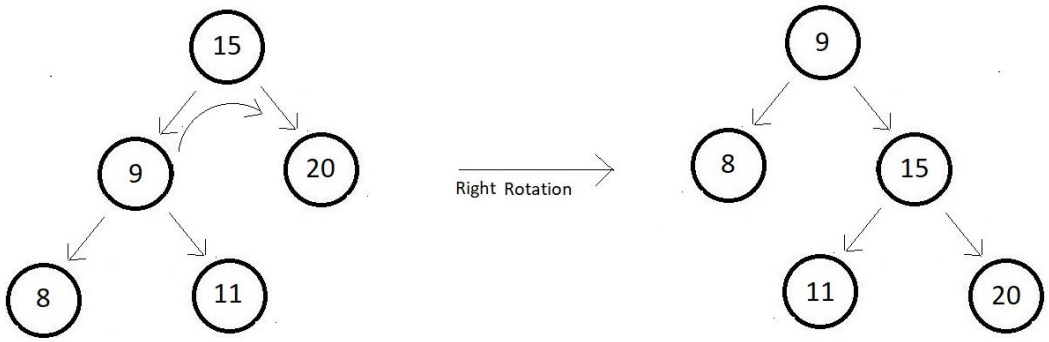
In this Rotate Operation, we move our unbalanced node to the left. Consider a binary search tree given below, and the newly formed tree after its left rotation with respect to the root.



One thing to observe here is that node 11 had to change its parent after the rotation to be able to maintain the balance of the tree.

Right Rotate Operations:

In this Rotate Operation, we move our unbalanced node to the right. Consider a binary search tree given below, and the newly formed tree after its right rotation with respect to the root.

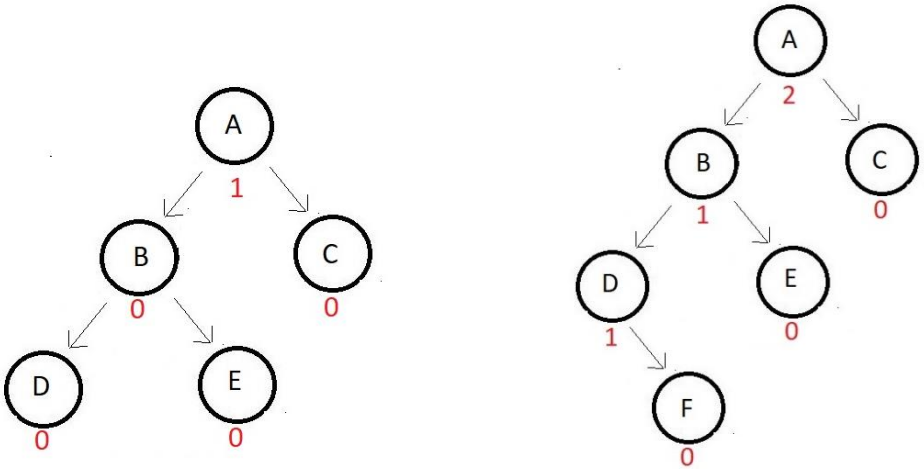


Again, as we can see that node 11 had to change its parent after the rotation to be able to maintain the balance of the tree. And rotating a tree to its left, and then again to the right, yields the same original tree as we can see from the above two examples.

Let's move back to the balancing of the AVL tree after insertion. So, when it comes to complex trees, in order to balance an AVL tree after insertion, we can follow the below-mentioned rules:

- For Left-Left insertion - Right rotate once with respect to the first imbalance node.
- For Right-Right insertion - Left rotate once with respect to the first imbalance node.
- For Left-Right insertion - Left rotate once and then Right rotate once.
- For Right-Left insertion - Right rotate once and then Left rotate once.

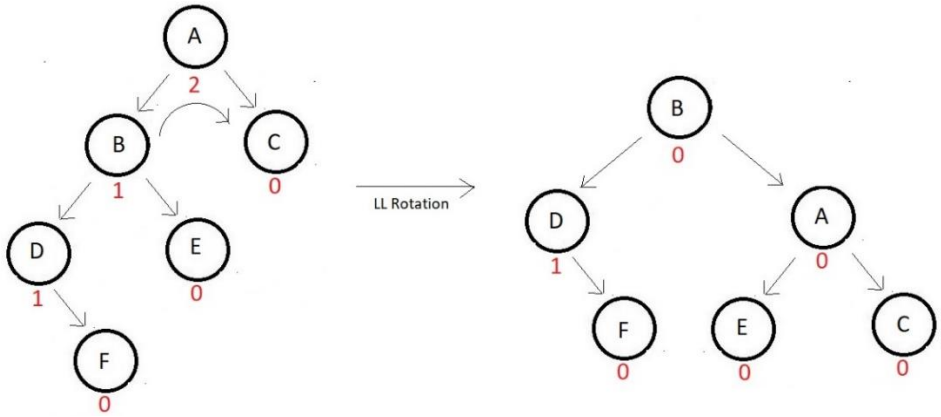
We'll now see how a complex tree gets balanced again after an insertion. Consider the binary search AVL tree below:



The absolute balance factor of each node is written beside and we can see how balanced the values are. Now suppose we need to insert an element that gets its position to the right of node D. Now the updated tree looks like the second one.

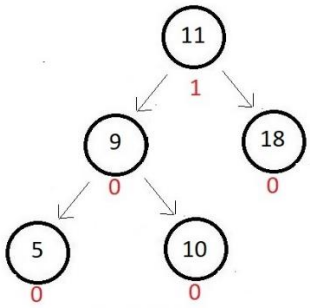
And the tree got imbalanced. Now, we follow these steps:

1. The first thing we would do is search for the node which got imbalanced first. We start iterating from the node we inserted at and move upwards looking for that first imbalance node. Here node A is the one we were searching for.
2. Second, we see what type of insertion was this with respect to the node we found. Here, the insertion happened to the left to the left of node A. So, this belongs to the first rule we saw above.
3. Do what the rule says. Here the rule says to right rotate once with respect to the first imbalance node. So, the tree after rotating to the right becomes:

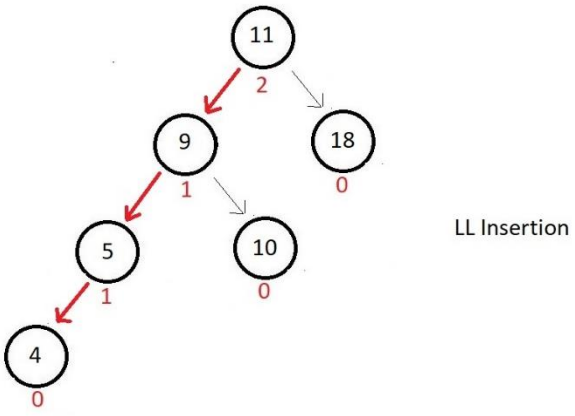


And similarly, had this been a type of right-right insertion, we would have first searched the first imbalanced node, and then would have rotated left with respect to it. But since this was just a demonstration, we would deal with two out of these four in detail now with actual numbers. Due to the similarity between the first and second rotations, we will take the LL rotation first and then one of the LR or RL rotations second.

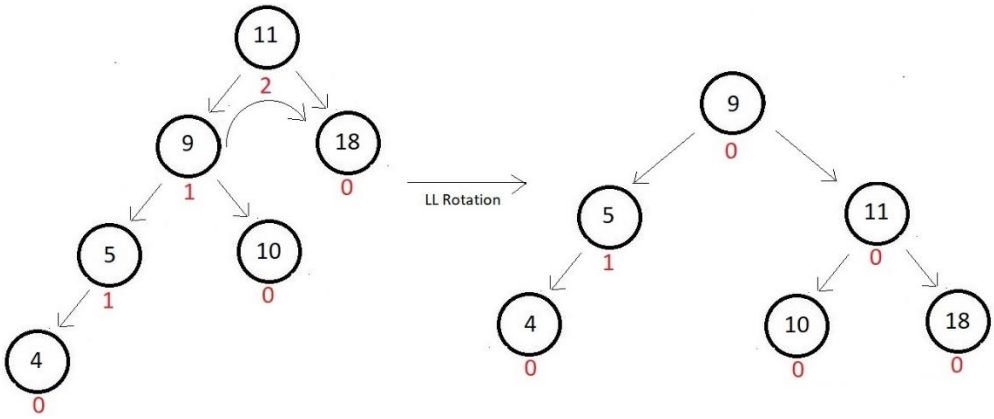
LL Rotation: We would just use LL Rotation to balance a relatively complex AVL tree.



Absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 4. So, that would get inserted to the left to node 5. The updated tree and their balance factors are:



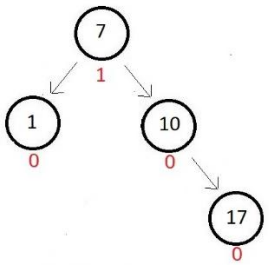
And since this is a case of left-left insertion, we would rotate right once with respect to the root node, since that's the first one to get imbalanced. And in that process, we might lose the position of node 10. So, we give it a new position to the left of node 11 to accommodate it again into the tree. And our tree gets balanced again.



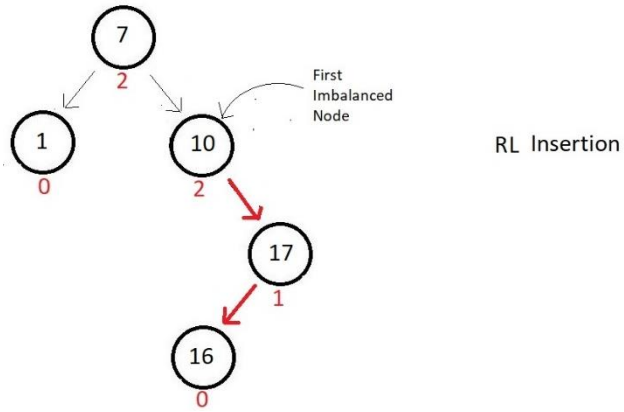
This is just a coincidence that our root node is the one we are rotating with respect to. We could come across examples where the first imbalanced ode is not the root. So, we would rotate with respect to the one we'll find first, not the root.

Let's now deal with one of the RL insertion cases. LR would be more or less the same, so, we'll just ignore that.

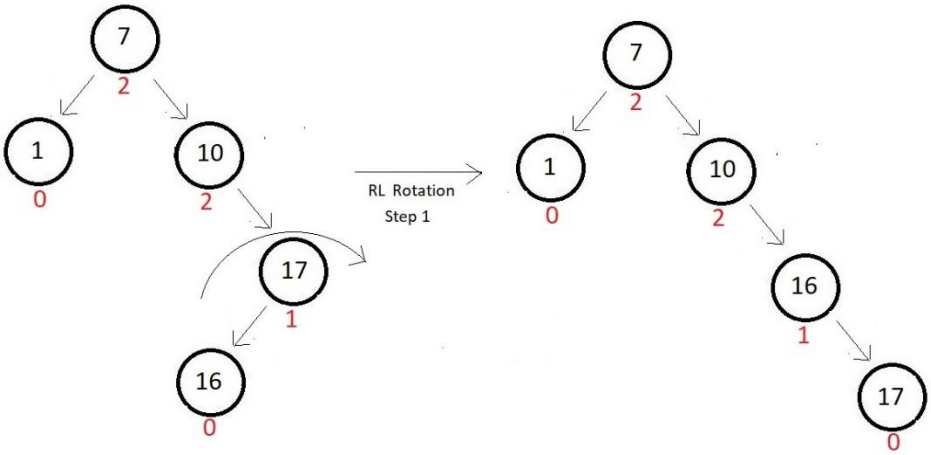
RL Rotation: We would just use RL Rotation to balance a relatively complex AVL tree.



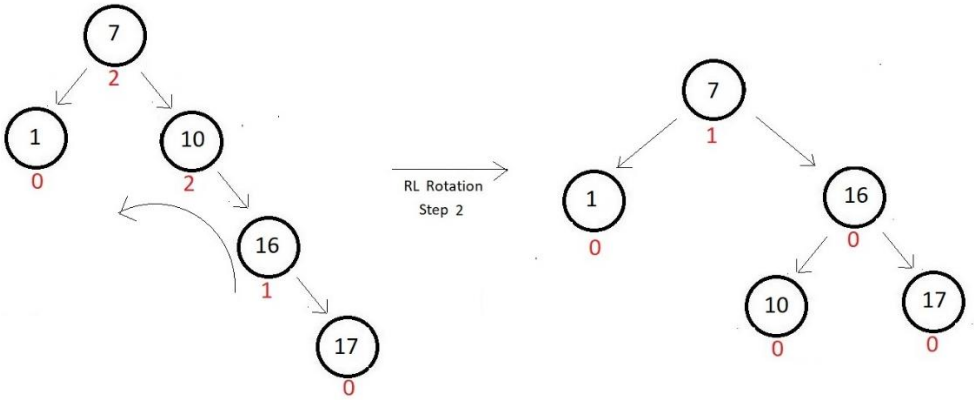
The absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 16. So, that would get inserted to the left of node 17. The updated tree and their balance factors are:



And since this is a case of right-left insertion with respect to the first imbalanced node which is node 10, we would first rotate right once with respect to the child of the first imbalanced node which comes into the path of the insertion node. Follow the figure below.

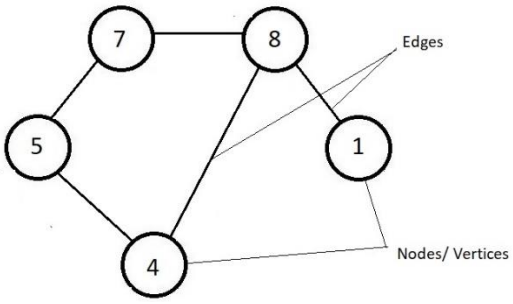


And now, we rotate left with respect to the node we found first imbalanced, here 10. And this would do our job. Our tree gets balanced again.

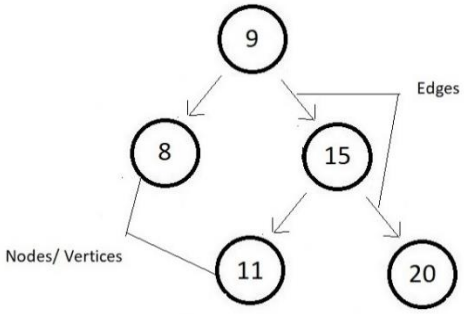


Graph

Graphs are an example of non-linear data structures. A graph is a collection of nodes connected through edges. Example of a simple graph:



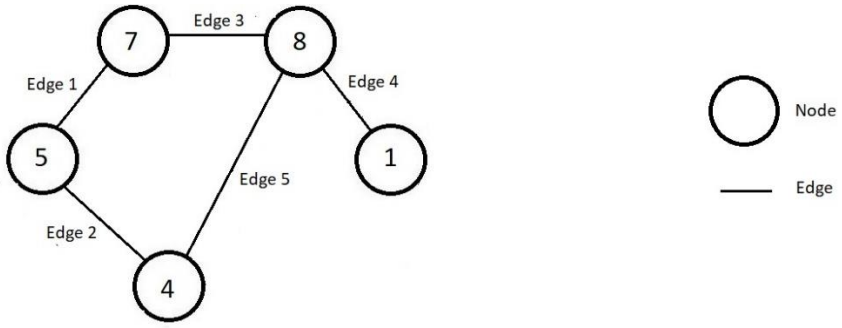
This was one of the general types of graphs we would learn. We might be surprised to learn that trees are also a type of graph. Follow the illustration of a tree below.



A tree is also a collection of nodes and edges, and hence is a graph only. Let's now quickly see all those elements of a graph.

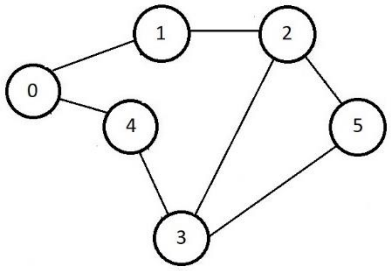
- **Nodes/ Vertices & Edges:**

A vertex or node is one fundamental unit/entity of which graphs are formed. Nodes are the data containing parts, connected with each other using edges. An edge is uniquely defined by its 2 endpoints or the two nodes it is connecting. We can take the analogy of people in a network, or a widespread chain, where people could be representing the nodes and their connections in the network could be represented using edges. It is a structure.



➢ **Formal Definition:**

A graph G = (V, E) is technically a collection of vertices and edges connecting these vertices. Follow the below-illustrated graph for reference:

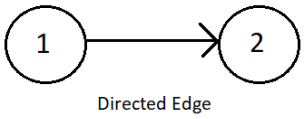


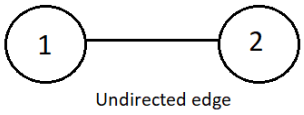
Here, V is the set of all the vertices. Therefore V = {0, 1, 2, 3, 4, 5}, and E is the set of all edges, therefore E = {(0, 1), (1, 2), (0, 4), (4, 3), (3, 5), (2, 5), (2, 3)}. Every edge connects the pair they are represented with. Edge (0, 1) connects node 0 to node 1. And hence, a graph is always represented using its set of vertices, V and its set of edges, E in the form G = (V, E).

Applications of Graphs:

Graphs have a wide range of applications in our world. Learning graphs also becomes a necessity for anyone pursuing software development due to its applications. Graphs are used to model paths in a city, as often seen in Google Maps. They are used to model social networks such as Facebook or LinkedIn. Graphs are also used to monitor website backlinks, internal employee networks, etc.

Types of Edges:



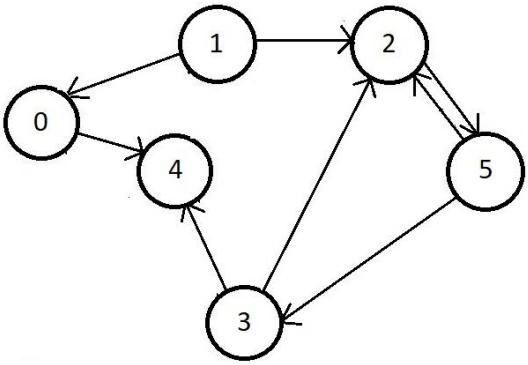


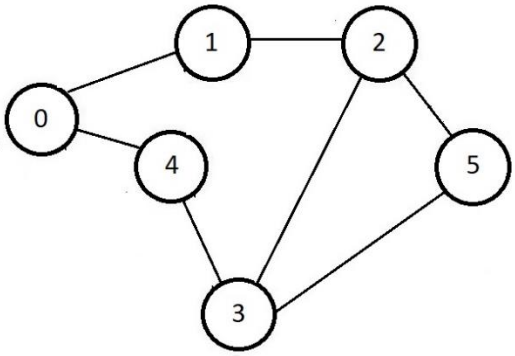
1. **Directed Edge** is an edge connecting two nodes but strictly defines the way it is connected from and to. Node 1 connecting to node 2, and not vice - versa. We can take the analogy of Facebook's follow feature, where if we follow someone, then it's not that the other person who followed us too. It's just one way. Another great example is that of a hyperlink, where one can even link google.com to their own websites on the internet, but then google.com would not necessarily link their website to their page.

2. **Undirected Edge** is an edge connecting two nodes from the way. Node 1 connecting to node 2, and at the same time node 2 connecting to node 1. We can take the analogy of Facebook's friend feature, where if we make someone a friend, then we too become their friend, so it's both ways.

According to the type of edges that a graph has, graphs can be further divided into two types:

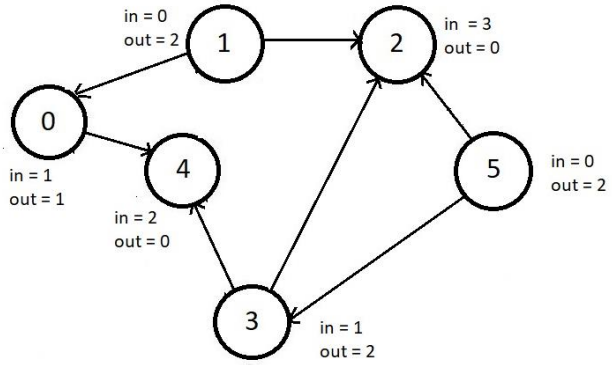
1. **Directed Graphs** are graphs having each of its edges directed
2. **Undirected Graphs** are graphs having each of its edges undirected.





Indegree and Outdegree of a Node:

Indegree of a node is the number of edges coming to the node, and outdegree of a node is the number of edges originating from that node. Consider the directed graph and we can write the indegree represented by in and outdegree represented by out, in the above graph for each of these nodes.



Let's talk about one real-life example of graphs - a graph of users - FACEBOOK!

- Although the users using Facebook need not understand the graph theory, once we create our profile there, Facebook uses graphs to model relationships between nodes.
- We can apply graph algorithms to suggest friends to people, calculate the number of mutual friends, etc.

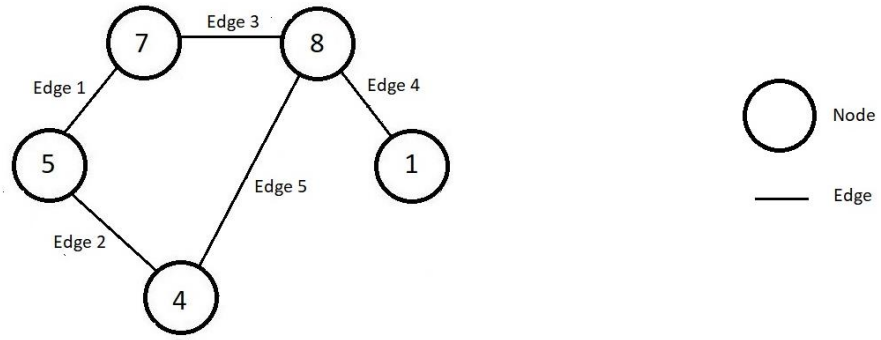
Suppose we have friends X, Y, and Z on Facebook. And we are one common friend to all of the three. Now, Facebook observes the connections and would suggest our friends to get connected with each other seeing your connection

with them. And we would be shown as one mutual friend to all three of them, and this is the concept behind the mutual friends and friends suggestions system.

- Other examples of graphs include the result of a web crawl for a website or for the entire world wide web, city routes as seen on Google Maps, etc. Furthermore, different search engines have different web network models.

Representation of Graphs:

A simple undirected graph looks like this:



Ways to Represent a Graph:

Any representation should basically be able to store the nodes of a graph and their connections between them. And this can be accomplished in so many ways but primarily the most used way to represent a graph is,

1. **Adjacency List** - Mark the nodes with their neighbours.
2. **Adjacency Matrix** - Aij = 1, if there is an edge between i and j, 0 otherwise.

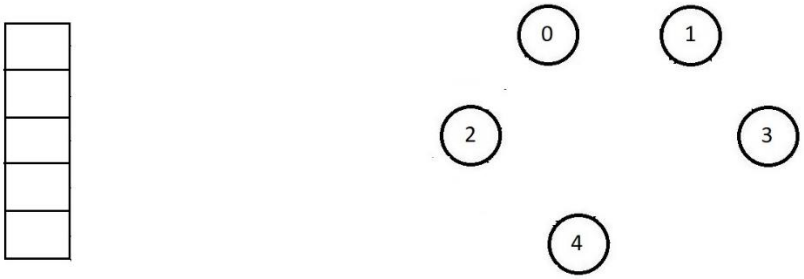
We'll see the above two in detail. Meanwhile, other representations include:

Edge Set - Store the pair of nodes/vertices connected with an edge. Example: {(0, 1), (0, 4), (1, 4)}.

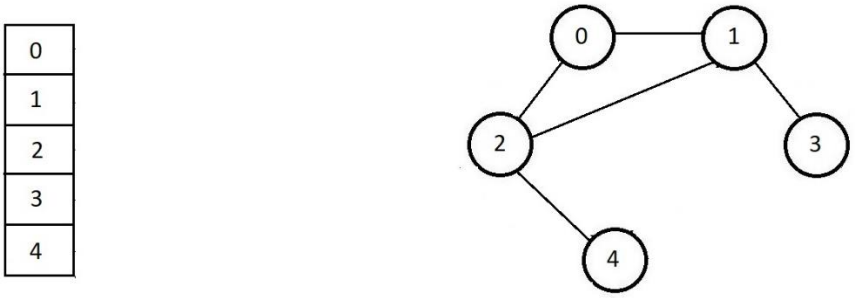
Other implementations to represent a graph also exist. For example, Compact list representation, cost adjacency list, cost adjacency matrix, etc.

Adjacency List:

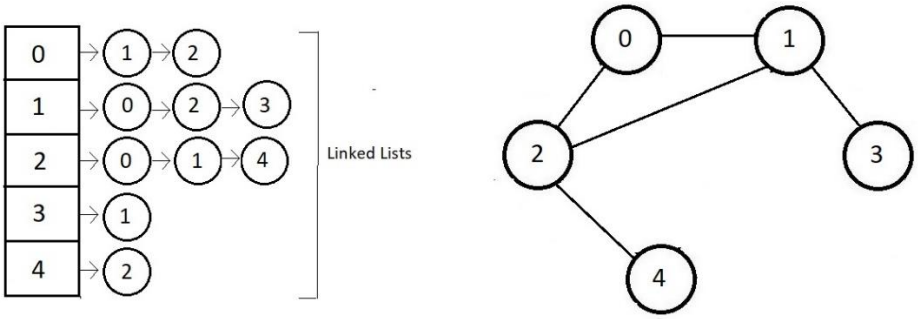
In this method of representation of graphs, we store the nodes, along with the list of their neighbors. Basically, we maintain a list of all the does, and along with it, we store the list of nodes a particular node is connected with. Consider the nodes illustrated below.



So first, we store the nodes we have in the graph.



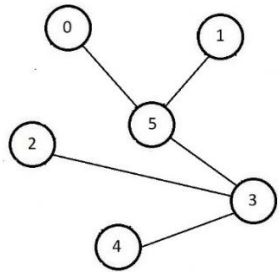
Next, we look for the connections of each of these nodes we stored. Starting with 0, we can see that 0 is connected with both 1 and 2. So, we will store that beside node 0. Next, 1 is connected with all 0, 2, and 3. 2 is connected with both 0 and 4, and 3 is connected with only 1, and 4 is connected with only 2. So, this information gets stored as shown below.



And the information about the connections of each of the nodes gets stored in separate linked lists as shown in the figure above. And each of the nodes itself acts as a pointer stored in an array pointing to the head of each of the linked lists. So, in this case, we would have an array of length 5 where the first index stores a pointer to the head of the adjacency linked list of the node 0.

Adjacency Matrix:

Adjacency matrix is another method of representation of graphs, where we represent our graph in the form of a matrix where cells are either filled with 0 or 1. Let's call the cell falling on the intersection of ith row and jth column be Aij, then the cell would be filled with 1 if there is an edge between node i and j, otherwise, the cell would be filled with 0. Consider the graph illustrated below:



Now, we'll make a 6x6 matrix as follows the we'll iterate through each of the cells, and see if there is an edge between the row number and the column number or not. If there is an edge, we'll place 1 in that cell, otherwise 0. Since there are no self-loops even, we'll mark 0 to the cell having i=j. The filled adjacency matrix for the above graph would be:

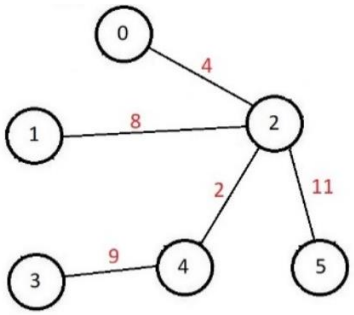
	0	1	2	3	4	5
0	0	0	0	0	0	1
1	0	0	0	0	0	1
2	0	0	0	1	1	0
3	0	0	1	0	1	1
4	0	0	0	1	0	0
5	1	1	0	1	0	0

Find easily whether there is an edge between any two nodes by simply looking for the cell representing the two nodes and checking if there is a 1 or a 0.

We can further extrapolate the application of the adjacency matrix by replacing these ones in the matrix with the weights for a weighted graph. Weighted graphs have a value/cost for each of the respective edges. These costs could represent anything, be it distance or time or cost literally. There is one traveling salesman problem where we store the shortest path from a city to some other city. Let's look at the cost adjacency matrix in detail.

Cost Adjacency Matrix:

The cost adjacency matrix is another method of representation of weighted graphs, where we represent our graph in the form of a matrix where cells are either filled with 0 or the cost of the edge. Let's call the cell falling on the intersection of ith row and jth column be Aij, then the cell would be filled with the cost of the edge between node i and j if there is an edge between node i and j, otherwise, the cell would be filled with a 0 and if the cost could also be 0, then we'll fill -1 in the cell where there is no edge.



Now, we'll make a 6x6 matrix as follows and iterate through each of the cells, and see if there is an edge between the row number and the column number or not. If there is an edge, we'll place 1 in that cell, otherwise a zero. Since there are no self-loops even, we'll mark 0 to the cell having i=j. The filled adjacency matrix for the above graph would be, assuming the cost could be zero as well:

Now, we'll make a 6x6 matrix as follows and iterate through each of the cells, and see if there is an edge between the row number and the column number or not. If there is an edge, we'll place 1 in that cell, otherwise a zero. Since there are no self-loops even, we'll mark 0 to the cell having i=j. The filled adjacency matrix for the above graph would be, assuming the cost could be zero as well:

	0	1	2	3	4	5
0	-1	-1	4	-1	-1	-1
1	-1	-1	8	-1	-1	-1
2	4	8	-1	-1	2	11
3	-1	-1	-1	-1	9	-1
4	-1	-1	2	9	-1	-1
5	-1	-1	11	-1	-1	-1

So, this was the cost adjacency matrix representation of graphs. Other implementations are not that frequently used, so let's go through them quickly and in brief.

Edge Set: Store the pair of nodes/vertices connected with an edge. Example: {(0, 1), (0, 2), (1, 2)} for a graph having nodes 0, 1 and 2 all connected with each other.

Cost Adjacency List: Similar to the adjacency list, but instead of just storing the node value, we'll also store the cost of the edge too in the linked list.

Compact List Representation: Here, the entire graph is compressed and stored in just one single 1D array.

Graph Traversal and Graph Traversal Algorithms:

Graph traversal refers to the process of visiting (checking and/or updating) each vertex (node) in a graph. A smaller graph seems relatively easy to traverse. But when it comes to traversing a graph with a huge number of nodes, the process needs to be automated. Doing things manually increases the chances of missing some vertices or so. And visiting nodes of a graph becomes important when we need to change something for some nodes, or just need to retrieve the value present there or something else. And this is where our graph traversing algorithms come to the rescue.

Sequences of steps that are used to traverse a graph are known as graph traversal algorithms. There are two algorithms that are used for traversing a graph. They are:

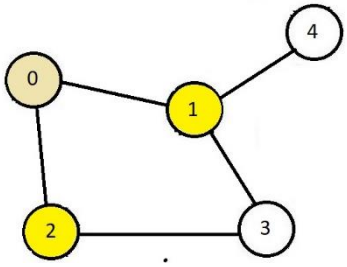
❖ Breadth-First Search (BFS)

There are a lot of ways to traverse a graph, but the breadth-first search says, start with a node, and first, visit all the nodes connected to this node. But before we actually delve deep into this, we need to know a few terminologies:

Exploring a Vertex (Node):

- In a typical graph traversal algorithm, we traverse through (or visit) all the nodes of the graph and add it to the collection of all visited nodes.
- Exploring a vertex in a graph means visiting all the connected vertices.

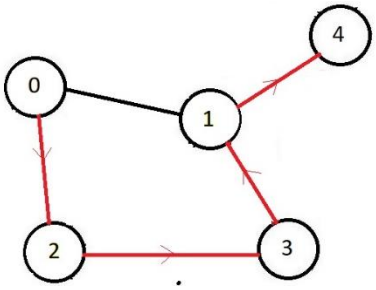
Follow the illustration of the graph mentioned below. We'll use this for understanding the breadth-first search.



In breadth-first search, we use the queue data structure, and we won't be implementing the queue data structure all over again, rather we'll presume that we have all its utility functions pre-defined. Here, suppose we start with node 0 and put it in the list of visited nodes, then we visit nodes 2 and 1 since they are directly connected to node 0. And then gets visited node 3 and 4. So, the order of exploring would be, 0, 2, 1, 3, 4.

In breadth-first search, we visit from left to right all the nodes which lie on the same level. Here, node 0 was on one level, followed by nodes 2 and 1, and then nodes 3 and 4.

❖ Depth First Search (DFS)



In depth-first search, we use the stack data structure, and we'll not implement a stack either. Suppose they are pre-defined and available for us to use. Here, suppose we start with node 0 and put it in the list of visited nodes, then we visit node 2 and then visit nodes further connected to node 2. So, we visit node 3 and since it is further connected to node 1 and node 1 is connected to 4, we'll follow them in the same order. So, the order of exploring would be, 0, 2, 3, 1, 4.

In depth-first search, we visit from top to down all the nodes which are connected with each other. Here, node 0 was the parent, which is connected to node 2, which is again connected to node 3, and then node 1 and then node 4.

Breadth First Search (BFS) Graph Traversal:

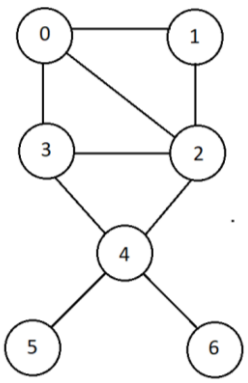
Graph traversal refers to the process of visiting (checking and/or updating) each vertex (node) in a graph. And since traversing a very large graph manually becomes impossible, some algorithms are used to accomplish this task. One such algorithm of graph traversal is Breadth-First Search.

In **Breadth-First Search**, we start with a node (not necessarily the smallest or the largest) and start exploring its connected nodes. The same process is repeated with all the connecting nodes until all the nodes are visited.

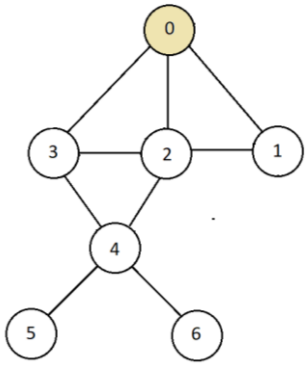
We should first learn the concept of the BFS spanning tree in order to understand the Breadth-First Search in a very intuitive way.

❖ Method 1: BFS Spanning Tree:

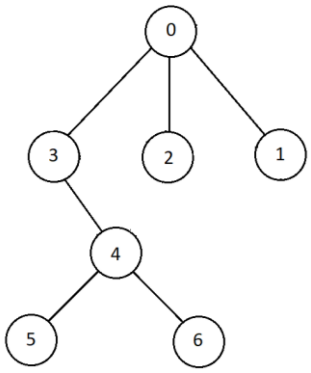
To understand what BFS Spanning Tree means, consider the graph illustrated below.



Now, choose any node, say 0, and try to construct a tree with this chosen node as its root. Or basically, hang this whole tree in a gravity-driven environment with this node and assume those edges are not rigid but flexible. So, the graph would now look something like this.



Now, mark dashed or simply remove all the edges which are either sideways or duplicate (above a node) to turn this graph into a valid tree, and as you know for a graph to be a tree, it shouldn't have any cycle. So, we can remove the edges between nodes 2 and 3, and then between nodes 1 and 2 being sideways. Then also between 2 and 4. We could have rather removed the one between node 3 and 4 instead of 2 and 4, but both ways work since these are duplicate to node 4. The tree we receive after we do these above-mentioned changes is,



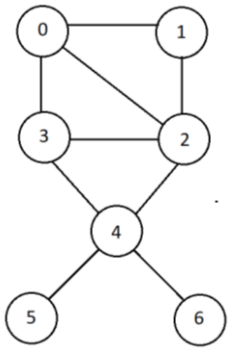
This constructed tree above is known as a BFS Spanning Tree. And surprisingly, the level order traversal of this BFS spanning tree gives us the Breadth-First Search traversal of the graph we started originally with. And if we remember what a level order traversal of a tree is, we simply write the nodes in the same level from left to right. So, the level order traversal of the above BFS Spanning Tree is **0, 3, 2, 1, 4, 5, 6.**

And a BFS Spanning Tree is not unique to a graph. We could have removed, as discussed above, the edge between nodes 3 and 4, instead of nodes 2 and 4. That would have yielded a different BFS Spanning Tree.

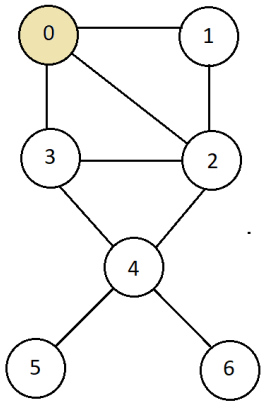
Therefore, given a graph, this is probably the easiest way, to write the Breadth-First Search traversal of the graph. Here, the chances of making a mistake are minimal.

❖ Method 2: Conventional Breadth-First Search Traversal Algorithm:

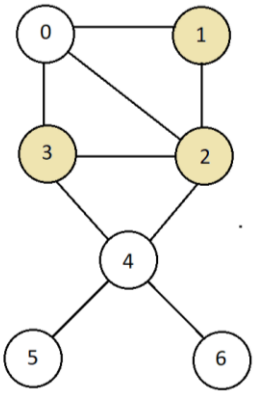
Consider the same graph we covered above. Illustrate that again.



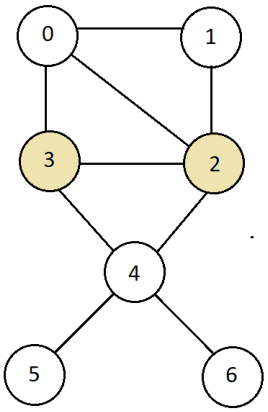
Considering we could begin with any source node; we'll start with 0 only. Let's define a queue named **exploration queue** which would hold the nodes we'll be exploring one by one. We would maintain another array holding the status of whether a node is already visited or not. Since we are starting with node 0, we would enqueue 0 into our exploration queue and mark it visited.



Now, we'll start visiting all the nodes connected to node 0, and remove node 0 from the exploration queue, enqueuing all the currently visited nodes which were nodes 1, 2, and 3. We are pushing them inside the exploration queue because these might further have some unvisited nodes connected to them. Mark these nodes visited as well.

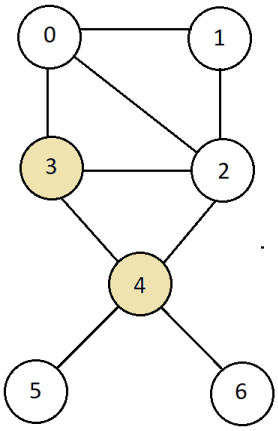


After this, we have node 1 at the top in the exploration queue, so we'll take it out and visit all unvisited nodes connected to it. Unfortunately, there aren't any. Therefore, we'll continue exploring further.



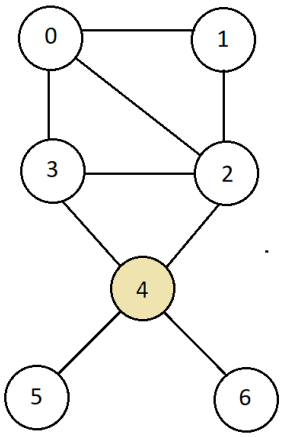
Visited: 0 1 2 3
Exploration Queue: 2 3

Next, we have node 2. And the only unvisited node connected to node 2 is node 4. So, we'll mark it visited and will also enqueue it in our exploration queue.



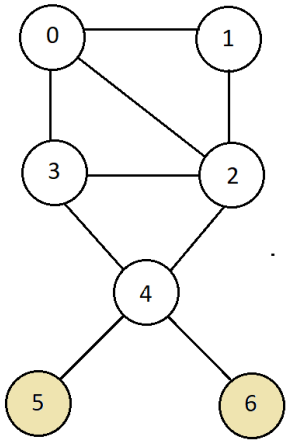
Visited: 0 1 2 3 4
Exploration Queue: 3 4

Node 3 is the next in line. Since, all nodes 1, 2, and 4 which are the nodes connected to it are already visited, we have nothing to do here while we are on node 3.



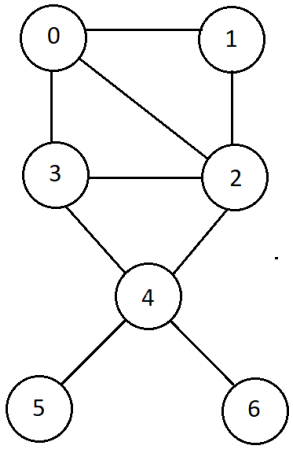
Visited: 0 1 2 3 4
Exploration Queue: 4

Next, we have node 4 on the top in the exploration queue. Let's get it out and see what nodes are connected and unvisited to it. So, we got nodes 5 and 6. Mark them visited and push them inside the exploration queue.



Visited: 0 1 2 3 4 5 6
Exploration Queue: 5 6

And now we can explore the other two nodes left in the queue, and since all nodes are already visited, we'll get nothing in them. And this got our queue emptied and every node traversed in Breadth-First Search manner.



Visited: 0 1 2 3 4 5 6
Exploration Queue:

And the order in which we marked our nodes visited is the Breadth-First Search traversal order. Here, it is **0, 1, 2, 3, 4, 5, 6**. So basically, the visited array maintains whether the node itself is visited or not, and the exploration queue maintains whether the nodes connected to a node are visited or not. This was the difference.

Let's now see how the process we did manually above can be automated in C although we will be using **pseudocode** for now.

a. We'll take a whole graph and the information about its nodes and edges as the input along with the source node s.

b. We'll mark the node s visited and then create a queue, and enqueue s in it.

c. We'll then initiate a while loop which will run until the queue is not empty. At each iteration, we will take out the first element out of the queue, and visit all the vertices already not visited and connected to it while enqueueing every new node we visit in the queue.

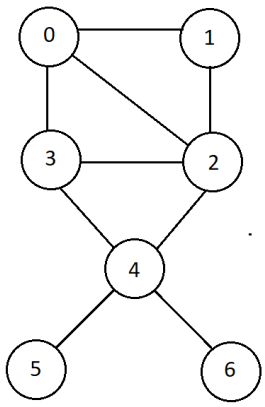
- **Input**: A graph G = (V,E) and source code s in V
- **Algorithm**:

```
- Mark all nodes v  in V as unvisited
- Mark source node s as visited
- enq(Q,s) //First-in first-out Queue
- while(Q is not empty)
      {
          u:=deq(Q);
          for each unvisited neighbour v of u {
              mark v as visited;
              enq(Q,v);
          }
      }
```

Few important points before we leave:

- We can start with any vertex.
- There can be multiple Breadth-First Search results for a given graph
- The order of visiting the vertices can be anything.

Adjacency Matrix corresponding to the graph illustrated below.



	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	1	0	0	0	0
2	1	1	0	1	1	0	0
3	1	0	1	0	1	0	0
4	0	0	1	1	0	1	1
5	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0

An adjacency matrix holds a value 1 for a cell that is at the intersection of the ith row and jth column if there is an edge between node i and node j.

Depth First Search (DFS) Graph Traversal:

In **Depth First Search**, we start with a node and start exploring its connected nodes, keeping on suspending the exploration of previous vertices. And those suspended nodes are explored once we finish exploring the node below. And this way we explore all the nodes of the graph.

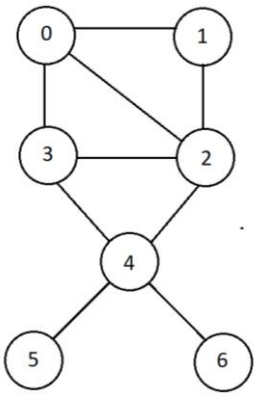
If we now **compare** what we are doing differently from BFS, we would find that there is one major thing that shouldn't go unnoticed.

In BFS, we explore all the nodes connected to a node, then explore the nodes we visited in a horizontal manner, while in DFS, we start with the first connected node, and similarly go deep, so this looks like visiting the nodes vertically.

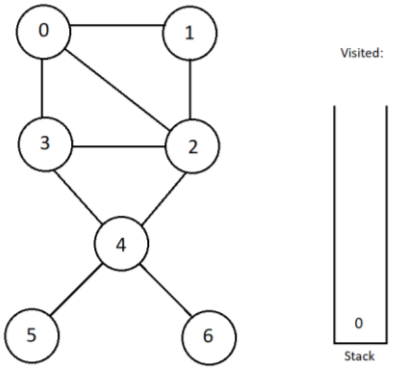
DFS Procedure:

1. Let's define a stack named which would hold the nodes we'll be suspending to explore one by one later. And stack, if we remember, is a data structure in which the element we push the last comes out the first. Choose any node as the source node and push it into the top of the stack we created. We would maintain another array holding the status of whether a node is already visited or not.
2. Take the top item of the stack and mark it visited or add it to the visited list.
3. Create a list of the nodes directly connected to the vertex we visited. Push the ones which are still not visited into the stack.
4. Repeat steps 2 and 3 until the stack is not empty.

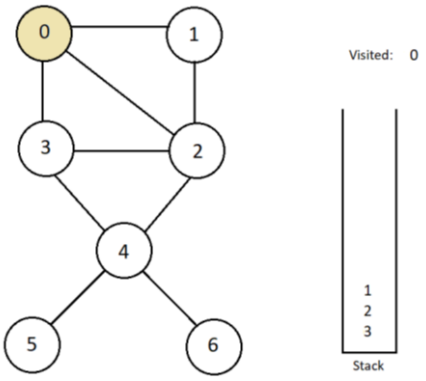
To understand the procedure of the Depth First Search, consider the graph illustrated below. For better contrast, we took the same graph as in the last lecture.



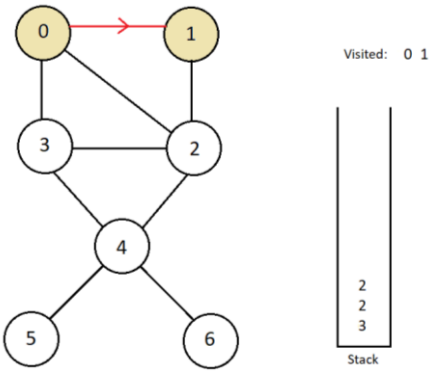
Considering the fact that we could begin traversing with any source node; we'll start with 0 only. So, following step1, we would push this node into the stack and begin our Depth First Search Traversal.



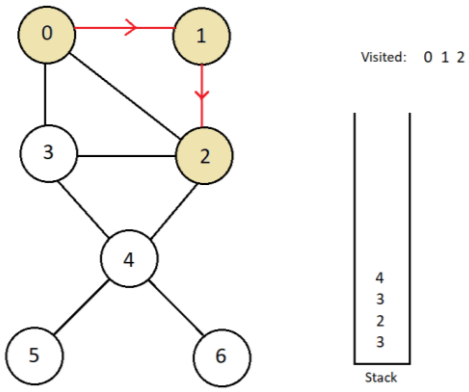
The next step says, pop the top element from the stack which is node 0 here, and mark it visited. Then, we'll start visiting all the nodes connected to node 0 which are not visited already, but before that, we are asked to push them all into the stack, and the order in which we push doesn't matter at all. Therefore, we will push nodes 3, 2, and 1 into the stack.



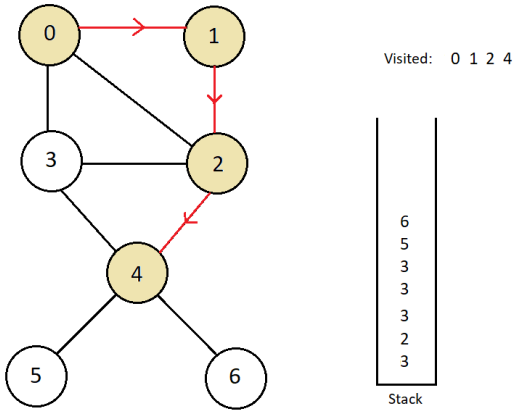
Repeating the steps above, we'll now pop the top element from the stack which is node 1, and mark it visited. Only nodes connected to node 1 were nodes 0 and 2, and since the only unvisited one is node 2. It's important to observe here, that although node 2 is in the stack, it is not visited. So, we'll push it into the stack again.



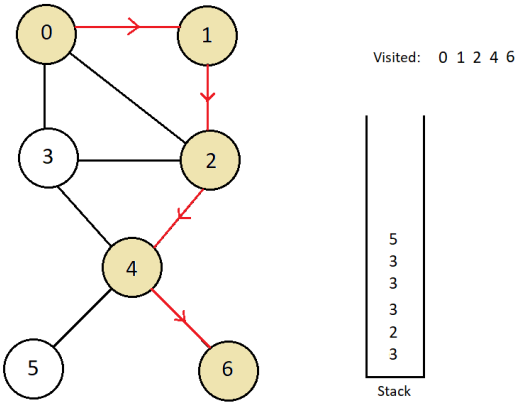
Next, we have node 2 at the top of the stack. We'll mark node 2 visited and unvisited nodes connected to node 2 are nodes 3 and 4, regardless of the fact that 3 is already there in the stack. So, we'll just push nodes 3 and 4 into the stack.



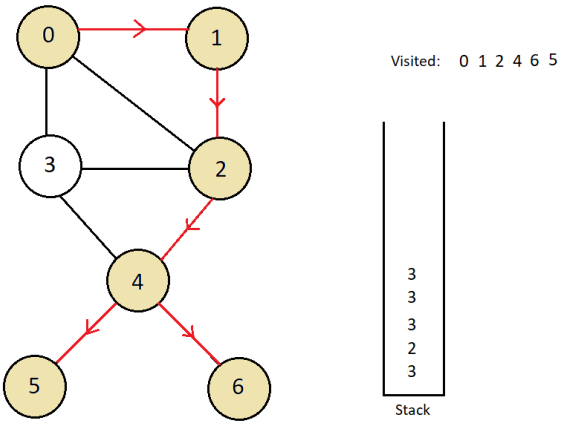
Node 4 is the next we have on the top. So, just mark it as visited. Since, all nodes 3, 5, and 6, except node 2, which are directly connected to it are not visited, we'll push them into the stack.



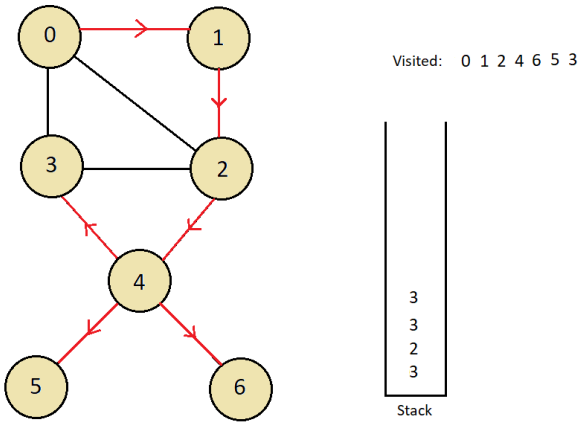
Next, we have node 6 on the top of the stack. Pop it and mark it visited. Since there are no nodes that are directed connected to node 6 and unvisited, we'll continue further without doing anything.



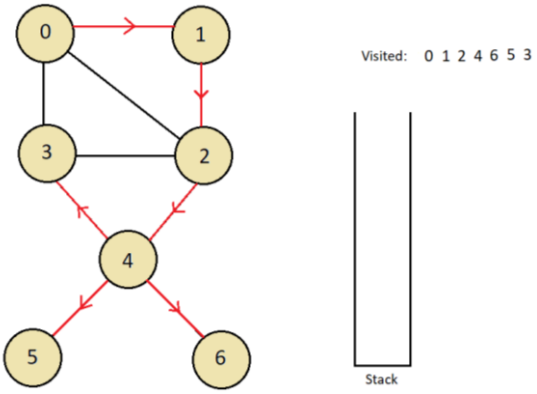
Next, we pop node 5 out of the stack and mark it visited. And since there is no unvisited node connected to it, we continue.



Node 3 comes next to be visited, being on the top in the stack. Mark node 3 visited and again there are no nodes left unvisited and connected to node 3. So, we just continue popping out elements from the stack.



Now, if we could observe, there are no nodes left to be visited. Although there are elements in the stack to be explored. So, we just pop them one by one and ignore finding them already visited. And this gets our stack emptied and every node traversed in Depth First Search manner, ultimately.



And the order in which we marked our nodes visited is the Depth First Search traversal order. Here, it is **0, 1, 2, 4, 6, 5, 3**. So basically, the visited array maintains whether the node itself is visited or not, and the stack maintains nodes whose exploration got suspended earlier. This was the difference.

One important reason why stack is used in the implementation of DFS and what makes it relatively easy is that it can directly be used in the form of functions since function calls use memory stacks.

Let's now see how the process we did manually above can be automated in C although we will be using pseudocode for now.

- **Input**: A graph G = (V,E) and source code s in V
- **Algorithm**:

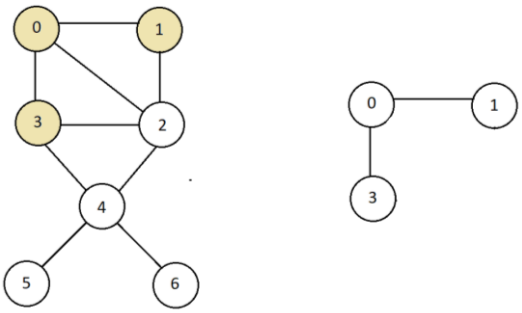
```
DFS(G,u)

u.visited = true
for each v ∈ G.adj(u)
   if v.visited == false
      DFS(G,v)


init()
  for each u ∈ G
     u.visited = false
  for each u ∈ G
     DFS(G,u)
```

Spanning Trees:

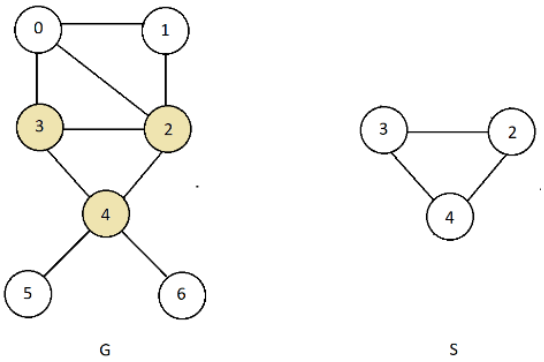
Before we dig into spanning trees, we should talk about a few things:

❖ Subgraphs:

A subgraph of a graph is a graph whose vertices and edges are subsets of the original graph.



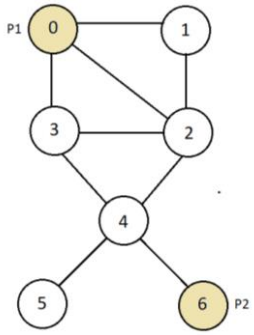
S is a subgraph of graph G because nodes 0, 1, and 3 form the subset of the set of nodes in G, and the edges connecting 0 to 3 and 0 to 1 also form the subset of the set of edges in G. Another subgraph of Graph G could be the one mentioned below.



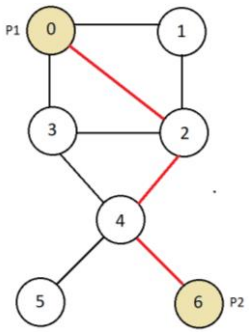
This is a subgraph of graph G too, because nodes 3, 2, and 4 form the subset of the set of nodes in G, and the edges connecting 3 to 4, 3 to 2, and 2 to 4 also form the subset of the set of edges in G.

❖ Connected Graphs:

A connected graph, as the word connected suggests, is a graph that is connected in the sense of a topological space, that there is a path from any point to any other point in the graph. And the graph which is not connected is said to be disconnected. Consider the graph below:



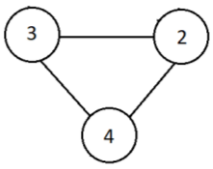
We have marked two random points P1 and P2, and we'll see if there is a path from P1 to P2. And if we could see, there is indeed a path from P1 to P2 via nodes 2 and 4.



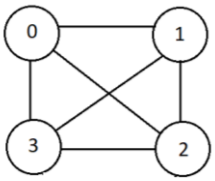
There could be a number of other ways to reach point P2 from P1, but there should exist at least one path from any point to another point for the graph to be called connected, otherwise disconnected. We should check for some other pair of vertices in the above graph.

❖ Complete Graphs:

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. Below, illustrated complete graphs with 3 and 4 nodes respectively.



Complete graph with 3 nodes



Complete graph with 4 nodes

As we can see, in any of the examples above, every pair of nodes is connected using a unique edge of their own. That is what makes a graph complete. We should consider making a complete graph with more vertices, say 5 or 7.

Note: Every complete graph is a connected graph, although every connected graph is not necessarily complete.

❖ Spanning Trees:

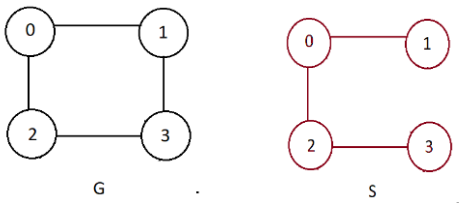
As we learnt, a subgraph of a graph G is a graph whose vertices and edges are subsets of the original graph G. Hence, a connected subgraph 'S' of a graph G (V, E) is said to be a spanning tree of the graph if and only if:

a. All the vertices of G are present in S,

b. No. of edges in S should be |V|-1, where |V| represents the number of vertices.

That is, for a subgraph of a graph to be called a spanning tree of that graph, it should **have all vertices of the original graph** and must have exactly **|V|-1 edges**, where |V| represents the number of vertices and the graph should be **connected.**

Consider a simple graph G and another graph S.



Now, let's find out, step by step, if graph S is a spanning tree of graph G or not, considering nodes 1 and 3 don't have an edge in common.

- ✓ Graph S is a subgraph of graph G. All nodes/vertices present in S are also a part of graph G, and all vertices exist in graph G too.
- ✓ Graph S is connected since we can go from any node to any other node via some edges in the graph.
- ✓ All vertices of graph G which are vertices 0, 1, 2, 3 are present in graph S.
- ✓ The number of edges in graph S equals the number of vertices in graph G - 1, since the number of vertices in graph G is 4, and the number of edges in graph S is 3.

Since graph G satisfies all the above conditions, it is a spanning tree of the graph G.

Therefore, whenever we are given a graph and are asked whether this graph is a spanning tree of another graph or not, we should just simply check for these four conditions, and declare it a spanning tree only if it satisfies all **four conditions**, and not a spanning tree even if it misses by any one of them.

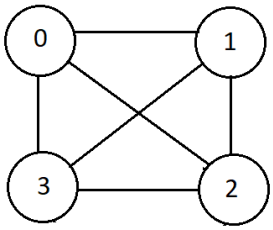
And if we are asked to create a spanning tree of a graph, we must first plot all the vertices of the original graph. And then create V-1 edges which were there in

the original graph and what makes our graph connected. This would be sufficient, and we can just ignore all other edges present in the original graph.

How many possible spanning trees are there for a graph: Well, there isn't a general formula for any random graph, but there is one for all complete graphs.

Number of Spanning Trees for Complete Graphs:

A complete graph has **n(n-2)** spanning trees where n represents the number of vertices in the graph. Consider the complete graph below with 4 nodes.

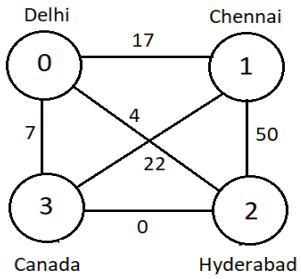


This complete graph has 4 vertices, hence the number of spanning trees it can have is, 4 raised to the power (4-2), that is, 42 which is 16.

Cost of Spanning Tree & Minimum Spanning Tree:

For easy understanding of things, we'll walk through an instance where there is a subject named Prem, and he's desperate to meet his beloved who is residing currently at someplace whose location he is not aware of. Although he has good options for places to visit and there are different routes to see all those places. Prem is not sort of a rich brat. He would definitely love to save money while he travels through places.

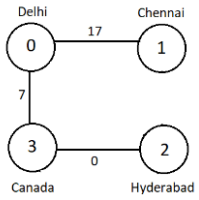
Consider the graph given below and the places are shown in the graph are the ones Prem has to visit. He wants to travel through all the places and at the same time for minimum possible expenditure. Each two places are connected through a route and every route would have some travel cost as well.



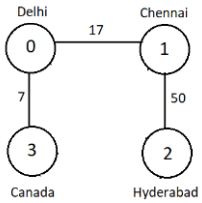
Prem, therefore, needs to come up with an algorithm that will help him find his beloved by wandering through all possible locations and at the least possible cost. And this is where finding a minimum cost spanning tree helps.

Cost of a Spanning Tree:

The cost of the spanning tree is the sum of the weights of all the edges in the tree. Now, consider the example of Prem we took at the beginning, and the graph he was to create a spanning tree of. Then, suppose we created a spanning tree S1 out of it which illustrated below.



S1



S2

What would be the cost of this spanning tree? Yes, 17+7+0 which is, 24. Let this cost for the spanning tree S1, be Cost1. Suppose another spanning tree S2 of the same graph.

This spanning tree has a cost worth 17+7+50, which is 74, and which is definitely greater than 24 what we calculated earlier for S1. Therefore, Prem would definitely not want to incur some cost greater than 24.

Minimum Spanning Tree:

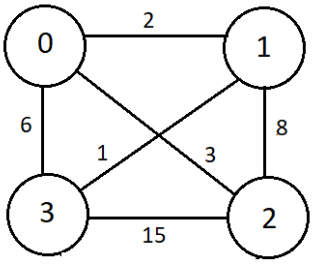
A Minimum Spanning tree, abbreviated as **MST**, is the spanning tree having the minimum cost. A graph can have a number of spanning trees all having some cost, say S1, S2, S3, S4, and S5 having cost 100, 104, 500, 400, and 10 respectively, but the one incurring the minimum cost is called the minimum spanning tree. Here, S5 having cost 10 would be the minimum spanning tree.

Applications of Minimum Spanning Tree:

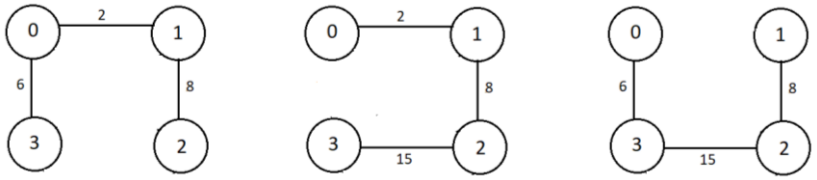
Applications of a minimum spanning tree must have gotten somewhat clear from the example of Prem, where we want to traverse all the nodes, which means the graph remains connected but it must have only bare minimum costing edges. So, this was a basic application of a minimum spanning tree. Things will gradually get clearer. Let's take some graphs and do some exercises on them ourselves.

Exercise 01:

Find the cost of any 3 spanning trees of the given graph.



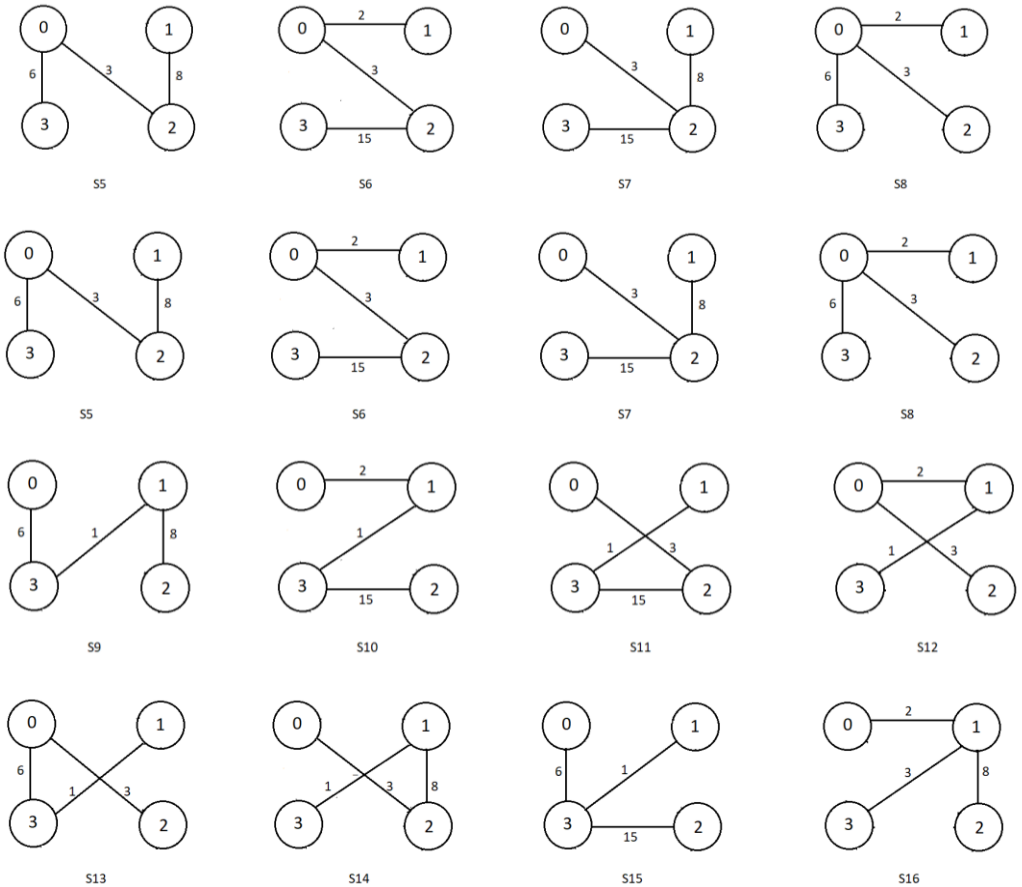
Consider the given graph be G. Our first step would be to create three spanning trees S1, S2, and S3 of the graph G. Then, mention weights to all the edges.



Now, let C1, C2, and C3 be the cost of spanning trees S1, S2, and S3 respectively. Therefore, C1 is (6+2+8) = 16, C2 is (2+8+15) = 25 and C3 is (6+15+8) = 29. Hence the cost of these spanning trees is 16, 25, 29 respectively. One thing we can infer from this is, C1 is less than C2, and C2 is less than C3, i.e., C1 $<$ C2 $<$ C3. As a result, C1 is preferred to both C2 and C3, and C2 is preferred to C3.

Exercise 02: Find the minimum spanning tree of the same graph we considered in the previous exercise.

Since graph G is a complete graph and it has a total of four nodes, the total number of spanning trees possible for graph G would be 4 raised to (4-2), which is 42, 16. So, the first step would be to create all the 16 possible spanning trees. All of them are illustrated below along with the edge weights.



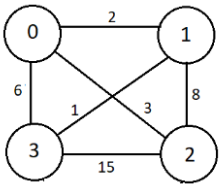
So, these were all the possible spanning trees we had. Now, first we have to count the cost of all these spanning trees, and then choose the minimum out of them.

Prims Minimum Spanning Tree Algorithm:

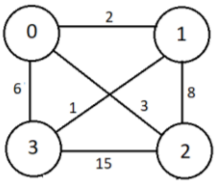
Prim's Algorithm uses the Greedy approach to find the minimum spanning tree. There are mainly **two steps** that this algorithm follows to find the minimum spanning tree of a graph:

1. We start with any node and start creating the Minimum Spanning Tree.
2. In Prim's Algorithm, we grow the spanning tree from a starting position until n-1 edges are or all nodes are covered.

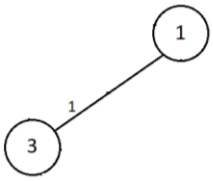
Let's take a simple example. Consider the complete graph with 4 vertices, K4



Now, as the first step says, we can start without any node in the graph. Arbitrarily, we will choose node 1. And since the edge between nodes 1 and 3 is the least weighted, we'll consider this edge in our minimum spanning tree.

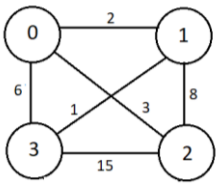


K_4

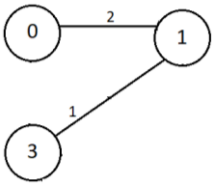


Spanning Tree

Next, we'll try involving node 0 in our spanning tree. Possible candidates for connecting node 0 from either node 3 or node 1 which compose the current spanning tree are edges having weights 6 and 2. Obviously, we'll choose the one with weight 2.

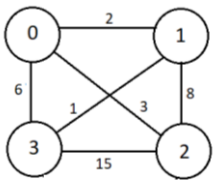


K_4

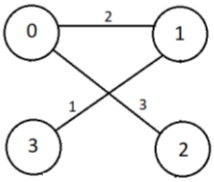


Spanning Tree

Now, the only node left is node 2. Possible candidates for connecting this node to the current spanning tree are edges with weights 15, 3, and 8. And there shouldn't be any doubt while considering the edge with weight 3.



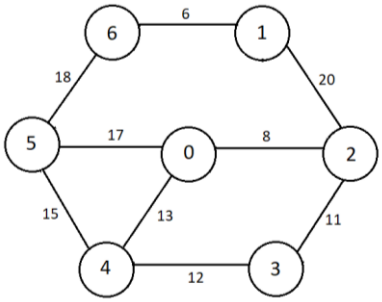
K_4



Spanning Tree

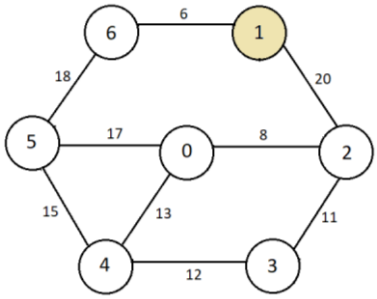
The cost of this spanning tree is (1+2+3) = 6 which is the minimum possible. And that finishes our construction of the spanning tree for graph K4. And this is exactly the way Prim's algorithm works.

Now, I'll give you one relatively tough example to get a better feel of the algorithm. Consider the graph I've illustrated below.



G

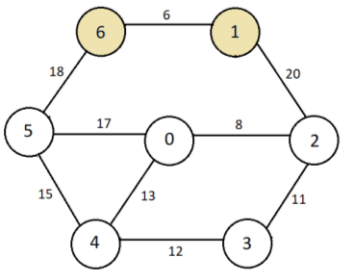
The first step, as the algorithm says, is to choose any arbitrary node and start creating the spanning tree. We have chosen node 1 for the same.



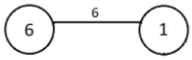
G

Spanning Tree

Colored nodes indicate that they have been considered for the spanning tree while the rest of the nodes are still left to cover. Here, in the Prim's algorithm, we maintain two sets of nodes, one for the nodes counted in the spanning tree, A, and another for the rest of them, V. Nodes connected to node 1 are 6 and 2, connected through edges of weights 6 and 20 respectively. Therefore, we will choose the one with a weight of 6.

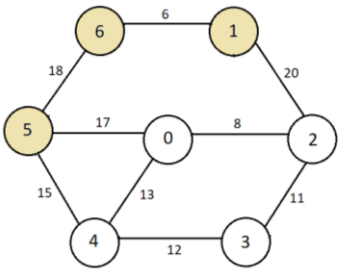


G

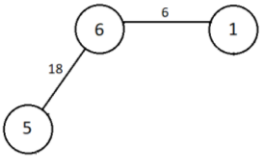


Spanning Tree

Now, 6 gets added to the set containing the covered elements. Nodes possible to get connected to the current spanning tree are 5 and 2. Node 5 and node 2 are connected to node 6 and node 1 respectively through edges of weights 18 and 20. We will obviously choose the one with less weight, i.e., weighted edge 18.

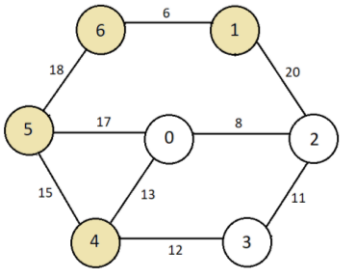


G

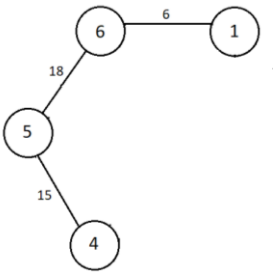


Spanning Tree

Now, 5 gets added to the set containing the covered elements. Possible nodes to get connected to the current spanning tree are 4, 0, and 2. Node 4 and node 0 are connected to node 5 through edges of weights 15 and 17 and node 2 is connected to node 1 through an edge of weight 20. We will, therefore, choose the one with weight min (15,17,20) i.e., 15. Hence, node 4 gets added to the current spanning tree.

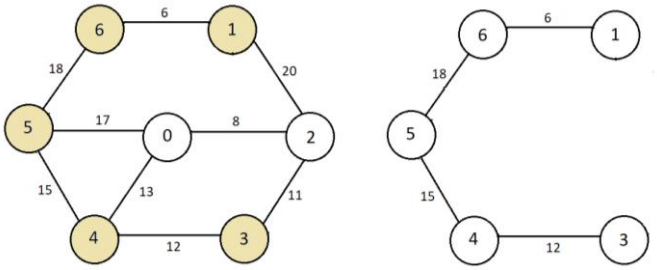


G

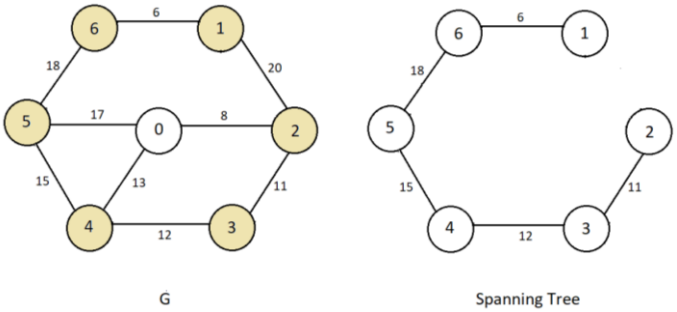


Spanning Tree

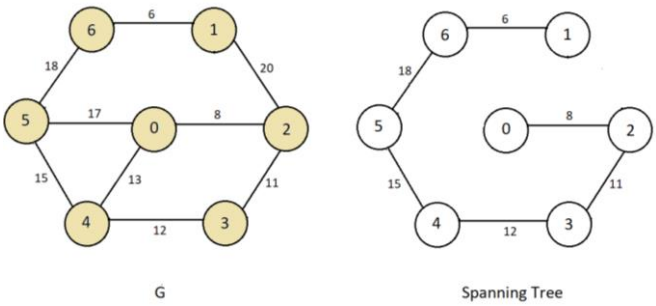
Now, we have nodes 0, 2, and 3 as possible candidates to get connected to the current spanning tree. Node 0 is connected to node 5 and node 4 through edges of weights 17 and 13 respectively. Node 3 is connected to node 4 through an edge of weight 12, and node 2 is connected to node 1 through an edge of weight 20. Therefore, we'll choose the one with weight min (17,13, 12, 20) i.e., 12. Hence, node 3 gets added to the current spanning tree.



Next, we are left with only two nodes to cover. Node 0 is connected to node 5 and node 4 through edges of weights 17 and 13 respectively. Similarly, Node 2 is connected to node 1 and node 3 through edges of weights 20 and 11 respectively. Therefore, we'll choose the one with weight min (17,13, 20, 11) i.e., 11. Hence, node 2 gets added to the current spanning tree through node 3.



The only node left now is node 0. We have 3 possible ways to connect this node to the rest of the tree. There are edges of weights 17, 13, and 8 though nodes 5, 4, and 2 respectively. Therefore, we'll choose the edge with a weight of 8.



This completes our spanning tree with 7 nodes and 6 edges. The cost of this spanning tree sums to (6+18+15+12+11+8), which is equal to 70. And we could verify if this is actually the minimum of all possible spanning trees. This is how Prim's algorithm works. It maintains the sets of elements covered and uncovered and sees the least possible edge to connect any one of the covered nodes to the uncovered ones. And the algorithm stops functioning once the set with uncovered nodes gets emptied.