

Data Structures and Algorithms:

Let's clear up our basics with these terms before deep diving into DSA. Data Structures and Algorithms are two different things.

Data Structures – These are like the ingredients you 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. You don't need to worry about these names. These topics will be covered in detail in the upcoming tutorials.

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.

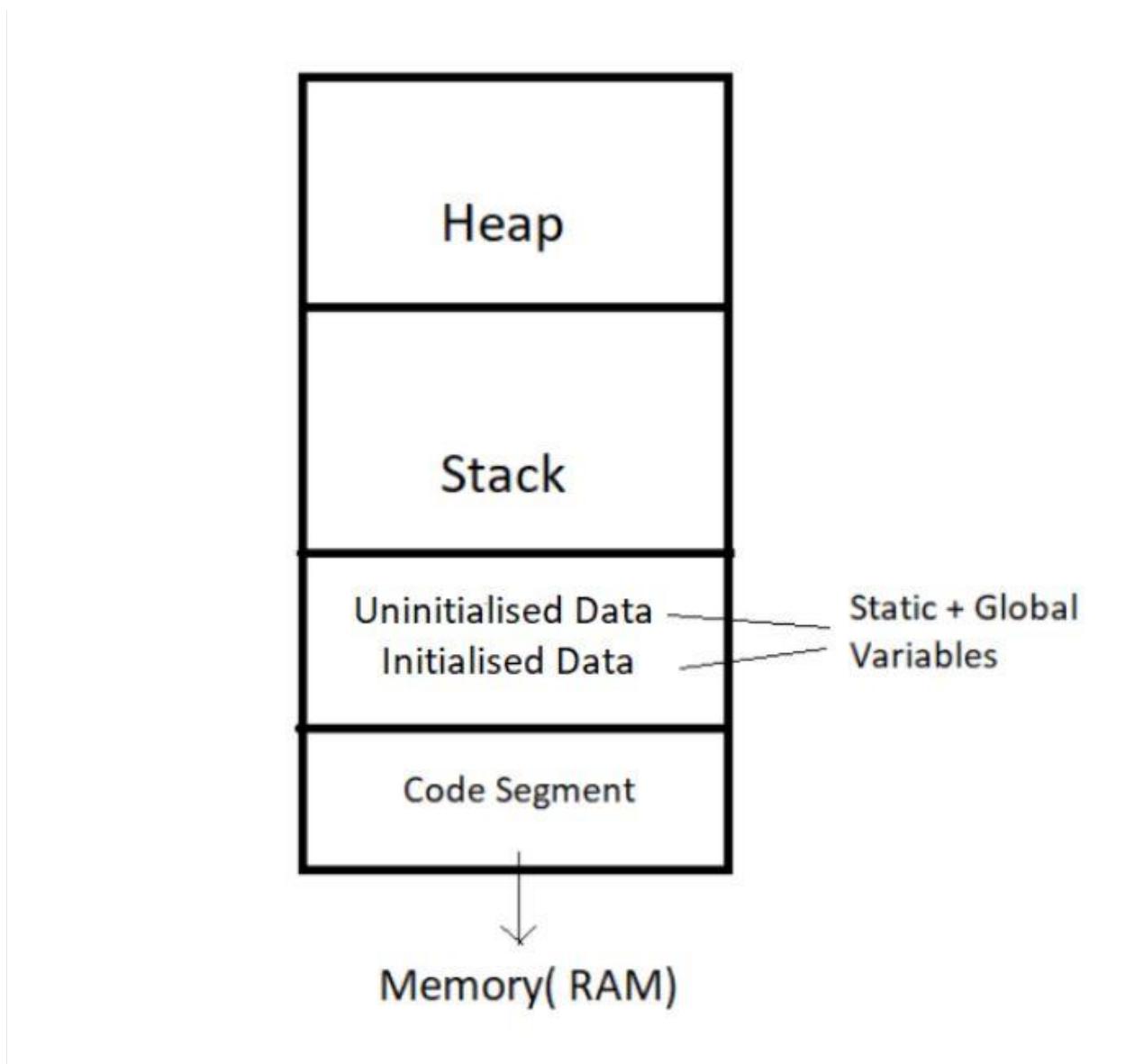
Some other Important terminologies:

1. **Database** – Collection of information in permanent storage for faster retrieval and updation. Examples are MySql, MongoDB, etc.
2. **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.
3. **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.

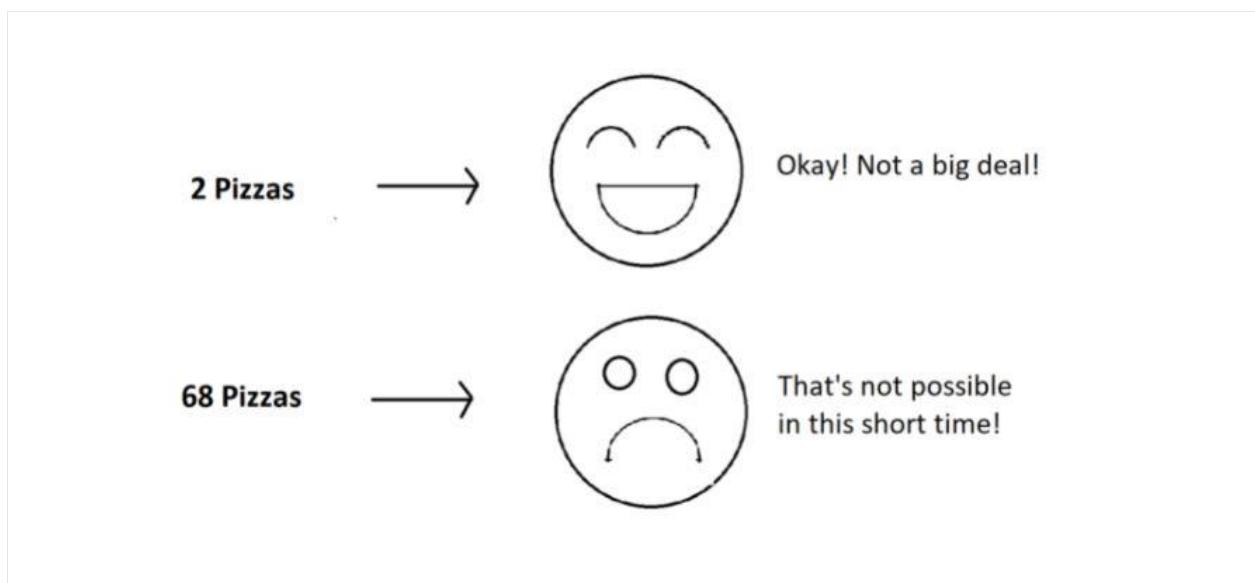
Take a look at the below diagram for a better understanding:



Time Complexity and Big O Notation (with notes)

An analogy to a real-life issue:

- This morning I wanted to eat some pizza; So, I asked my brother to get me some from Dominos, which is 3 km away.
- He got me the pizza, and I was happy only to realize it was too little for 29 friends who came to my house for a surprise visit!
- My brother can get 2 pizzas for me on his bike, but pizza for 29 friends is too huge of an input for him, which he cannot handle.



What is Time Complexity?

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. Let's understand this concept with the help of an example:

Consider two developers Shubham and Rohan, who created an algorithm to sort 'n' numbers independently. When I made the program run for some input size n, the following results were recorded:

No. of elements (n)	Time Taken By Shubham's Algo	Time Taken By Rohan's Algo
10 elements	90 ms	1 ms
70 elements	110 ms	10 ms
110 elements	180 ms	20 ms
1000 elements	2s	0.1s

We can see that at first, Shubham's algorithm worked well with smaller inputs; however, as we increase the number of elements, Rohan's algorithm performs much better.

Quick Quiz: Who's algorithm is better?

Time Complexity: Sending GTA 5 to a friend:

- Imagine you have a friend who lives 5 km away from you. You want to send him a game. Since the final exams are over and you want him to get this 60 GB file worth of game from you. How will you send it to him in the shortest time possible?
- Note that both of you are using JIO 4G with a 1 Gb/day data limit.

- The best way would be to send him the game by delivering it to his house. Copy the game to a hard disk and make it reach him physically.
- Would you do the same for sending some small-sized game like MineSweeper which is in KBS of size? Of Course no, because you can now easily send it via the Internet.
- As the file size grows, the time taken to send the game online increases linearly – $O(n)$ while the time taken by sending it physically remains constant. $O(n^0)$ or $O(1)$.

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:

$$k_1 n^2 + k_2 n + 36 = O(n^2)$$

Highest Order Term Can ignore other lower order terms

Algo 2:

$$k_1 k_2^2 + k_3 k_2 + 8 = O(n^0) = O(1)$$

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.

Note that these are the formulas for the time taken by their program.

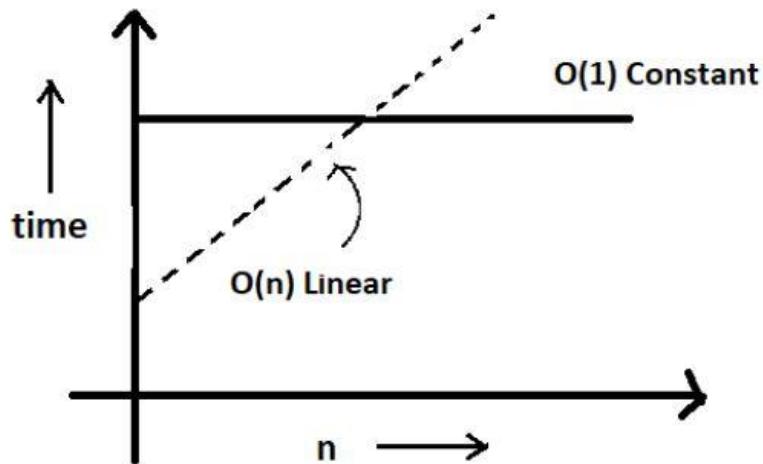
What is a Big O?

Putting it simply, 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, they would look something like this:



Asymptotic Notations: Big O, Big Omega and Big Theta Explained (With Notes)

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

Now let's look at the mathematical definition of 'order of.' Primarily there are three types of widely used asymptotic notations.

1. Big oh notation (O)
2. Big omega notation (Ω)
3. Big theta notation (Θ) - Widely used one

Big oh notation (O):

- Big oh notation is used to describe an asymptotic upper bound.
- Mathematically, if $f(n)$ describes the running time of an algorithm; $f(n) = O(g(n))$ if and only if there exist positive constants c and

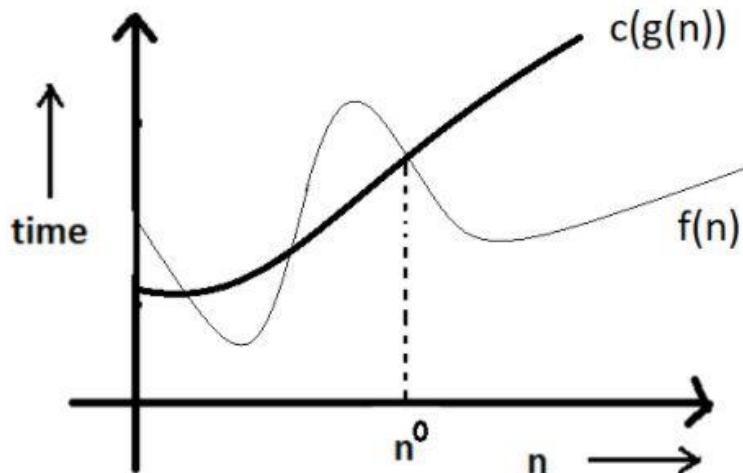
n° such that:

$$\theta \leq f(n) \leq c g(n) \quad \text{for all } n \geq n^\circ.$$

Copy

- Here, n is the input size, and $g(n)$ is any complexity function, for, e.g. n , n^2 , etc. (It is used to give upper bound on a function)
- If a function is $O(n)$, it is automatically $O(n^2)$ as well! Because it satisfies the equation given above.

Graphic example for Big oh (O):



Big Omega Notation (Ω):

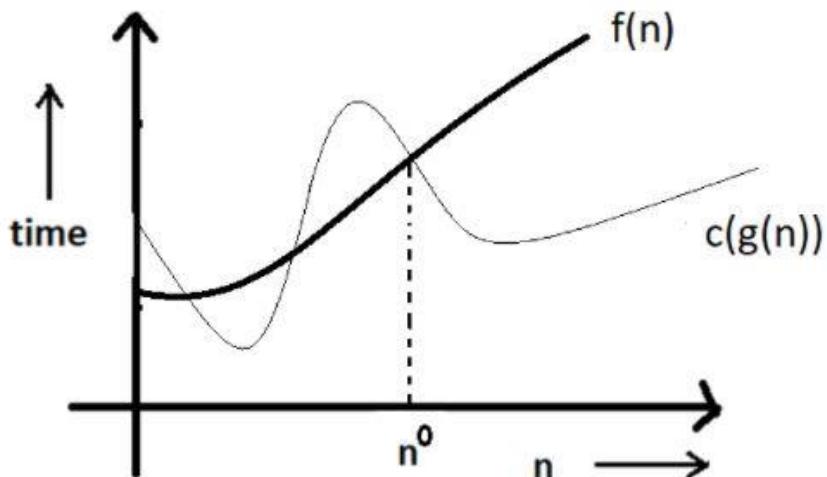
- Just like O notation provides an asymptotic upper bound, Ω notation provides an asymptotic lower bound.
- Let $f(n)$ define the running time of an algorithm; $f(n)$ is said to be $\Omega(g(n))$ if and only if there exist positive constants c and n° such that:

$$\theta \leq c g(n) \leq f(n) \quad \text{for all } n \geq n^{\circ}.$$

Copy

- It is used to give the lower bound on a function.
- If a function is $\Omega(n^2)$ it is automatically $\Omega(n)$ as well since it satisfies the above equation.

Graphic example for Big Omega (Ω):



Big theta notation (θ):

- Let $f(n)$ define the running time of an algorithm.
- $f(n)$ is said to be $\theta(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$ both.

Mathematically,

$$0 \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_o$$

$$0 \leq c_2 g(n) \leq f(n) \quad \forall n \geq n_o$$

for sufficiently large value of n

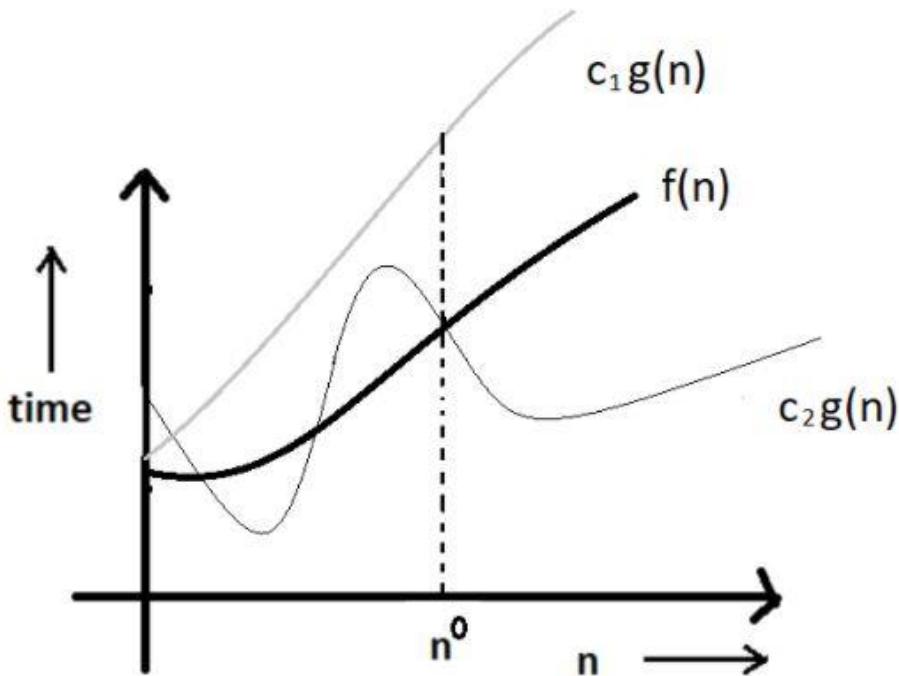
Merging both the equations, we get:

$$\theta \leq c_2 g(n) \leq f(n) \leq c_1 g(n) \quad \forall n \geq n_o.$$

[Copy](#)

The equation simply means that there exist positive constants c_1 and c_2 such that $f(n)$ is sandwiched between $c_2 g(n)$ and $c_1 g(n)$.

[Graphic example of Big theta \(θ \):](#)



Which one of these to use?

Big theta provides a better picture of a given algorithm's run time, which is why most interviewers expect you to answer in terms of Big theta when they ask "order of" questions. And what you provide as the answer in Big theta, is already a Big oh and a Big omega. It is recommended for this reason.

Quick Quiz: Prove that n^2+n+1 is $O(n^3)$, $\Omega(n^2)$, and $\theta(n^2)$ using respective definitions.

Hint: You can approach this both graphically, making some rough graphs and mathematically, finding valid constants c_1 and c_2 .

Increasing order of common runtimes:

Below mentioned are some common runtimes which you will come across in your coding career.

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

↑ ↑
Better Worse

Common runtimes from better to worse



Best Case, Worst Case and Average Case Analysis of an Algorithm (With Notes)

Life can sometimes be lucky for us:

- Exams getting canceled when you are not prepared, a surprise test when you are prepared, etc. → **Best case**

Occasionally, we may be unlucky:

- Questions you never prepared being asked in exams, or heavy rain during your sports period, etc. → **Worst case**

However, life remains balanced overall with a mixture of these lucky and unlucky times. → **Expected case**

Those were the analogies between the study of cases and our everyday lives. Our fortunes fluctuate from time to time, sometimes for the better and sometimes for the worse. Similarly, 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.

1. **Algorithm 1** - Start from the first element until an element greater than or equal to the number to be searched is found.
2. **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)

- o 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 = $O(1)$
- o 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)

- o If we get really lucky, the first element will be the only element that gets compared. Hence, a constant time.
 - Best case complexity = $O(1)$
- o If we get unlucky, we will have to keep dividing the array into halves until we get a single element. (that is, the array gets finished)
 - o Hence the time taken : $n + n/2 + n/4 + \dots + 1 = \log n$ with base 2
- Worst-case complexity = $O(\log n)$

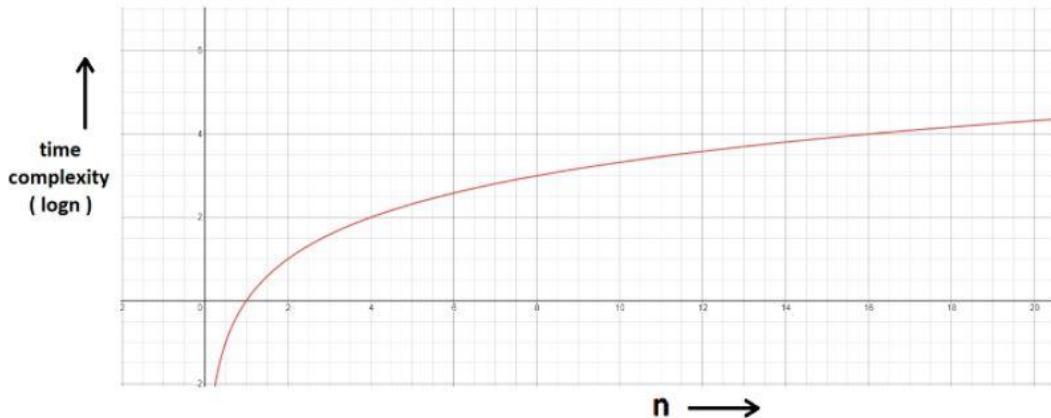
What is $\log(n)$?

Log refers to how many times I need to divide n units until they can no longer be divided (into halves).

- $\log_2 8 = 3 \Rightarrow 8/2 + 4/2 + 2/2 \rightarrow \text{Can't break anymore.}$

- $\log_4 4 = 2 \Rightarrow 4/2 + 2/2 \rightarrow$ Can't break anymore.

You can refer to the graph below, and you will find how slowly the time complexity (Y-axis) increases when we increase the input n (X-axis).



Space Complexity:

- Time is not the only thing we worry about while analyzing algorithms. Space is equally important.
- Creating an array of size n (size of the input) $\rightarrow O(n)$ Space
- If a function calls itself recursively n times, its space complexity is $O(n)$.

Quiz Quiz: Calculate the space complexity of a function that calculates the factorial of a given number n.

Hint: Use recursion.

You might have wondered at some point why we can't calculate complexity in seconds when dealing with time complexities. Here's why:

- Not everyone's computer is equally powerful. So we avoid handling absolute time taken. We just measure the growth of time with an increase in the input size.
- Asymptotic analysis is the measure of how time (runtime) grows with input.

How to Calculate Time Complexity of an Algorithm + Solved Questions (With Notes)

In previous videos, we had discussed what time complexity is and how it helps in dealing with what is most efficient for our programs. Our task today will be to find out how to calculate the time complexity of our programs. Here are some tips and tricks about the same, followed by a discussion of some questions.

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:

$$T(n) = n^2 \rightarrow O(n^2)$$

$$T(n) = \log n \rightarrow O(\log n)$$

Copy

Here are some tricks to calculate complexities:

- Drop the constants:

Anything you might think is $O(kn)$ (where k is a constant) is $O(n)$ as well. 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 you represent as $O(n^2+n)$ can be written as $O(n^2)$. 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 $m * n \rightarrow O(kmn) \rightarrow O(mn)$

Time Complexity – Competitive Practice Sheet:

Question1: Fine the time complexity of the *func1* function in the program shown in the snippet below:

```
#include<stdio.h>
```

```
void func1(int array[], int length)
```

```
{
```

```
int sum=0;
int product =1;
for (int i = 0; i <length; i++)
{
    sum+=array[i];
}
```

```
for (int i = 0; i < length; i++)
{
    product*=array[i];
}
}
```

```
int main()
{
    int arr[] = {3,4,66};
    func1(arr,3);
    return 0;
}
```

Copy

Question 2: Find the time complexity of the *func* function in the program from program2.c as follows:

```
void func(int n)
{
    int sum=0;
    int product =1;
    for (int i = 0; i <n; i++)
    {
```

```

for (int j = 0; j < n; j++)
{
    printf("%d , %d\n", i,j);
}
}

```

[Copy](#)

Question 3: Consider the recursive algorithm below, where the random(int n) spends one unit of time to return a random integer where the probability of each integer coming as random is evenly distributed within the range [0,n]. If the average processing time is T(n), what is the value of T(6)?

```

int function(int n)
{
    int i = 0;
    if (n <= 0)
    {
        return 0;
    }
    else
    {
        i = random(n - 1);
        printf("this\n");
        return function(i) + function(n - 1 - i);
    }
}

```

[Copy](#)

Question 4: Which of the following are equivalent to O(N) and why?

1. $O(N + P)$, where $P < N/9$
2. $O(9N-k)$
3. $O(N + 8\log N)$
4. $O(N + M^2)$

Question 5: The following simple code sums the values of all the nodes in a balanced binary search tree (don't worry about what it is, we'll learn them later). What is its runtime?

```
int sum(Node node)
{
    if (node == NULL)
    {
        return 0;
    }
    return sum(node.left) + node.value + sum(node.right);
}
```

[Copy](#)

Question 6: Find the complexity of the following code which tests whether a given number is prime or not?

```
int isPrime(int n)
{
    if (n == 1)
    {
        return 0;
    }
    for (int i = 2; i * i < n; i++)
    {
        if (n % i == 0)
        {
```

```
        return 0;  
    }  
}  
return 1;  
}
```

Copy

Question 7: What is the time complexity of the following snippet of code?

```
int isPrime(int n)  
{  
    for (int i = 2; i * i < 10000; i++)  
    {  
        if (n % i == 0)  
        {  
            return 0;  
        }  
    }  
  
    return 1;  
}  
isPrime();
```

Copy

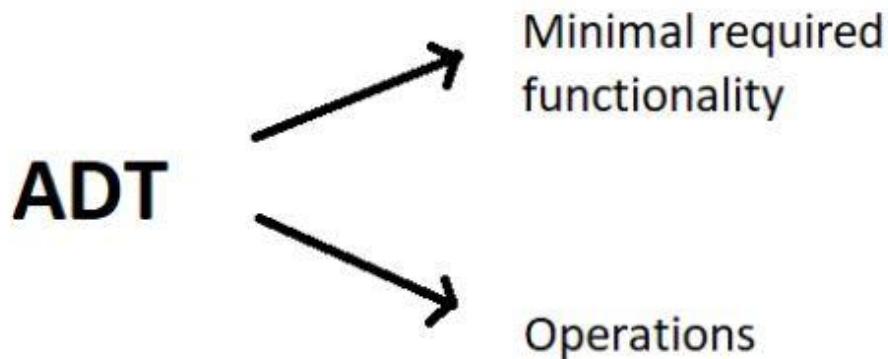
Arrays and Abstract Data Type in Data Structure (With Notes)

Today, we will learn about what an abstract data type is. This will just be an introduction. We'll implement these ideas in our next tutorial. Let's start with the basics.

Abstract Data Types and Arrays:

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

An array ADT holds the collection of given elements (can be int, float, custom) accessible by their index.

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.

- `get (i)` – get element i
- `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.

- `Max()`
- `Min()`
- `Search (num)`
- `Insert (i, num)`
- `Append (x)`

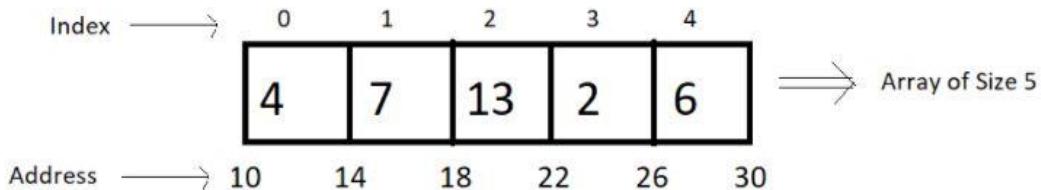
Static and Dynamic Arrays:

- Static arrays - Size cannot be changed
- Dynamic arrays - Size can be changed

Quick Quiz- Code the operations mentioned above in C language by creating array ADT.

Hint: Use structures.

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.

Array as An Abstract Data Type in Data Structures(With Notes)

In the last video, we learned what abstract data types are. In this video, we will be interested in implementing an array as an abstract data type. Giving it a quick revision, 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.

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,

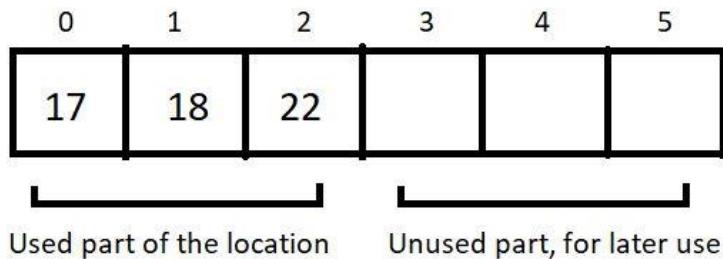
- max()
- get(i)
- set(i,num)
- add(another_array)

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, do:

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.

Let the below-illustrated array be an example of what we are talking about.



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

Implementing Array as an Abstract Data Type in C Language

In the last tutorial, we discussed the blueprint of our customized abstract data type, myArray. In case you missed it, make sure you check it out. Today, we will learn to write the code to implement that array with all the previously defined sets of values and operations.

Editor settings:

I will recommend you to use MinGW w64-bit compiler to compile your C programs and VS Code as your code editors. VS Code is highly recommended for its versatility with all the programming languages in the market. You can even check out my Youtube video covering all of this. Let's, for now, assume that you all have your setup

ready. I have attached the code snippet for creating the above ADT array below. Let's check it out.

Understanding the snippet below:

1. First, we will define a structure. You can use a class and its methods in C++, but in C, a structure is used to define customized data types.
2. Keep the blueprint we made in the last tutorial by your side. Define the structure elements, integer variables total_size and used_size, and an integer pointer to point at the address of the first element.
3. We are now ready with our customized data type. Let's define some functions, which will feature
 - Creating an array of this data type,
 - Printing the contents of this array,
 - Setting values in this array.

Create a void function *createArray* by passing the address of a struct data type *a*, and integers tSize and uSize. We can very easily assign this tSize and uSize given from the main, to the total_size and used_size of the struct myArray *a* by either of the methods defined below.

```
(*a).total_size = tSize;  
or  
a->total_size = tSize;
```

Copy

Code Snippet 1: Syntax for assigning structure elements to structure pointers.

Similarly, assign the integer pointer ptr, the address of the reserved memory location using malloc. Do use the header file <stdlib.h> for using malloc.

```
a->ptr = (int *)malloc(tSize * sizeof(int));
```

Copy

Code Snippet 2: Using malloc

4. We will now create a *show* function to display all the elements of the struct myArray. We will simply pass the address of the struct myArray *a*. To print all the elements, we will traverse through the whole struct and print each struct element till the iterator reaches the last element. We will use *a->used_size* to define the loop size. Use *(a->ptr)[i]* to access each element.

5. We will now create a setVal function to set values to this struct myArray a and pass the address of the same. Use scanf to assign values to each element via `(a→ptr)[i]`.

```
#include<stdio.h>
#include<stdlib.h>

struct myArray
{
    int total_size;
    int used_size;
    int *ptr;
};

void createArray(struct myArray * a, int tSize, int uSize){
    // (*a).total_size = tSize;
    // (*a).used_size = uSize;
    // (*a).ptr = (int *)malloc(tSize * sizeof(int));

    a->total_size = tSize;
    a->used_size = uSize;
    a->ptr = (int *)malloc(tSize * sizeof(int));
}

void show(struct myArray *a){
    for (int i = 0; i < a->used_size; i++)
    {
        printf("%d\n", (a->ptr)[i]);
    }
}

void setVal(struct myArray *a){
```

```

int n;
for (int i = 0; i < a->used_size; i++)
{
    printf("Enter element %d", i);
    scanf("%d", &n);
    (a->ptr)[i] = n;
}

}

int main(){
    struct myArray marks;
    createArray(&marks, 10, 2);
    printf("We are running setVal now\n");
    setVal(&marks);

    printf("We are running show now\n");
    show(&marks);

    return 0;
}

```

[Copy](#)

Code Snippet 3: A program to implement the ADT array

So, these were the basic methods we could define for this struct. We'll check if these work by running it. We'll call the *createArray*, and *setVal*/functions first to create an array of size 2, and assign some values to it. And then call the *show* function to see if it works.

Output of the above program:

```

Enter element 0 : 12
Enter element 1 : 13
We are running show now

```

12

13

PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

Copy

And this was implementing the myArray ADT. I hope you all could follow it. Possibly there were some syntaxes you were not familiar with, but don't worry, take your time. Watch the other courses regarding them on my YouTube channel.

Thank you for being with me throughout. I hope you enjoyed the tutorial. If you appreciate my work, please let your friends know about this course too. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where we'll learn to operate on this array using several operators. Till then keep learning.

Here is the source code we wrote in the video!

```
#include<stdio.h>
#include<stdlib.h>

struct myArray
{
    int total_size;
    int used_size;
    int *ptr;
};

void createArray(struct myArray * a, int tSize, int uSize){
    // (*a).total_size = tSize;
    // (*a).used_size = uSize;
    // (*a).ptr = (int *)malloc(tSize * sizeof(int));

    a->total_size = tSize;
    a->used_size = uSize;
    a->ptr = (int *)malloc(tSize * sizeof(int));
}
```

```
void show(struct myArray *a){
    for (int i = 0; i < a->used_size; i++)
    {
        printf("%d\n", (a->ptr)[i]);
    }
}

void setVal(struct myArray *a){
    int n;
    for (int i = 0; i < a->used_size; i++)
    {
        printf("Enter element %d", i);
        scanf("%d", &n);
        (a->ptr)[i] = n;
    }
}

int main(){
    struct myArray marks;
    createArray(&marks, 10, 2);
    printf("We are running setVal now\n");
    setVal(&marks);

    printf("We are running show now\n");
    show(&marks);

    return 0;
}
```

Operations on Arrays in Data Structures: Traversal, Insertion, Deletion and Searching

In the last tutorial, we discussed implementing our abstract data type array and its set of values. In today's lesson, we'll explore how we can operate on these arrays. For example: traversing through the array, sorting the array, and many more. We'll start with the primary ones.

Operations on an Array:

While there are many operations that can be implemented and studied, we only need to be familiar with the primary ones at this point. An array supports the following operations:

- Traversal
- Insertion
- Deletion
- Search

Other operations include sorting ascending, sorting descending, etc. Let's follow up on these individually.

Traversal:

Visiting every element of an array once is known as **traversing** the array.

Why Traversal?

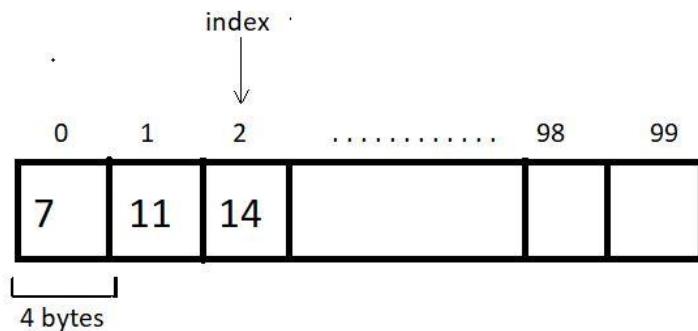
For use cases like:

- Storing all elements - Using `scanf()`
- Printing all elements - Using `printf()`
- Updating elements.

An array can easily be traversed using a *for* loop in C language.

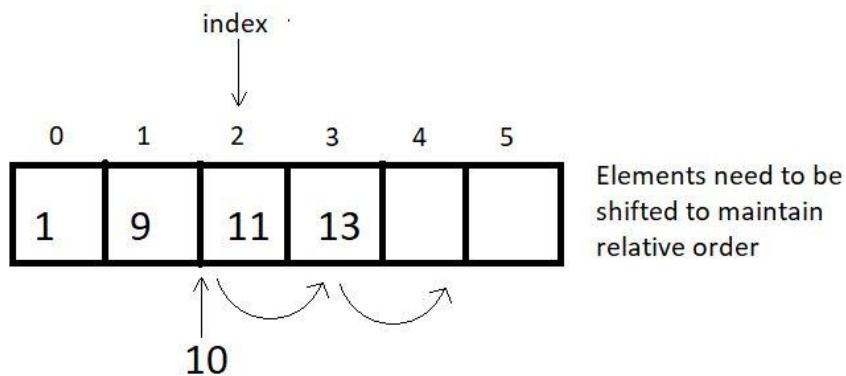
An important note on Arrays:

If we create an array of length 100 using `a[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 cannot go beyond 100 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.

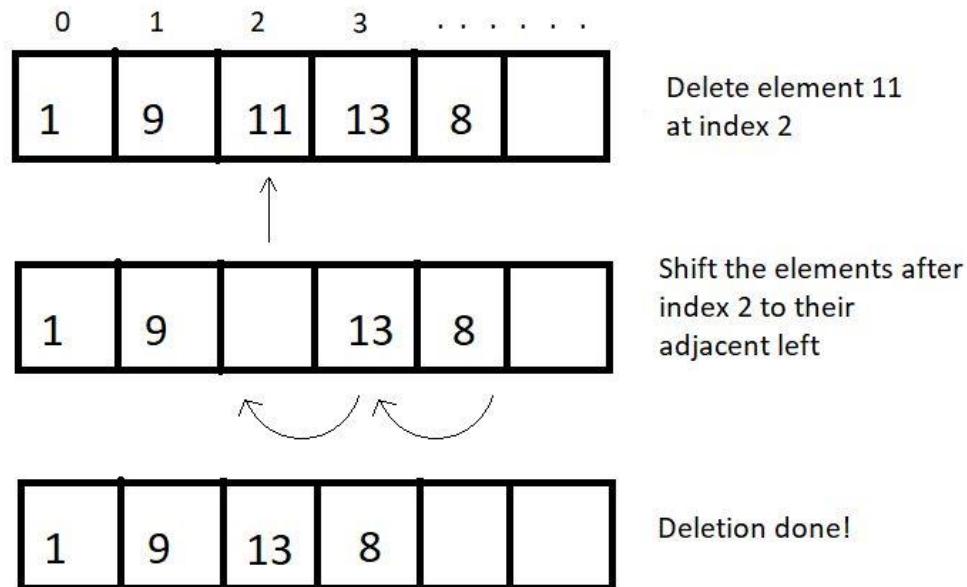


When no position is specified, it's best to insert the element at the end to avoid shifting, and this is when we achieve the best runtime $O(1)$.

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.

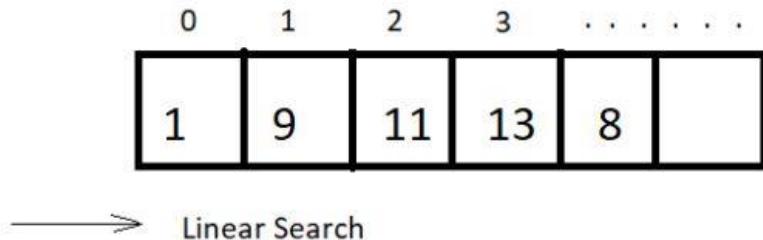
We can also bring the last element of the array to fill the void if the relative ordering is not important. :)



Quick Quiz: What is the best and the worst runtime for a delete operation?

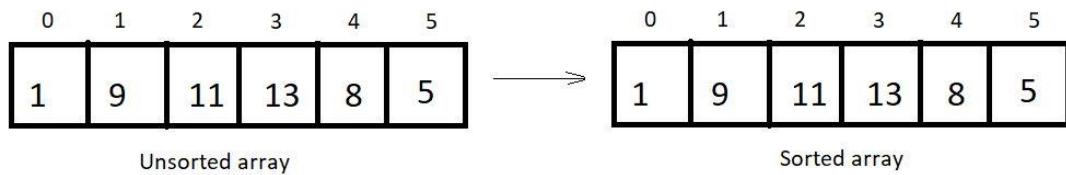
Searching:

Searching can be done by traversing the array until the element to be searched is found. $O(n)$ There is still a better method. As you may remember, we talked about binary search in some previous tutorials. Don't forget to look it up if you missed it. We had analyzed both linear and binary search. This search method is only applicable for sorted arrays. 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). We have different algorithms to sort arrays. We'll see various sorting techniques later in the course.



Coding Insertion Operation in Array in Data Structures in C language

In the last tutorial, we discussed all the primary operators and the concepts behind each. Today, we will learn how to code their algorithms. But before that, let's give ourselves a quick revision.

We talked about four operations-

basically, **traversal**, **insertion**, **deletion**, and **searching**. As already mentioned, traversal is not any big a deal. It can just be achieved by using a *for loop*. Our main objective today would be to implement insertion. So, let's slide our chairs to our coding arena. I have attached the code snippet below.

Understanding code snippet 1:

1. We will start by declaring an array *ar* of length 100. Initialize this array with some 4-5 elements. This will be our used memory.

2. We'll create a void *display* function using the method of traversal. Pass this array to the display function by value or by reference. And print the elements. Printing the elements of an array has already been covered in my C playlist. Visit now if you haven't yet.
3. We'll now create an integer function *indInsertion* (integer, just to check if the operation succeeds). Before that, create an integer variable *size* to store the used size of the array. Pass into this void function the array and its used size, the element to be inserted and the total size, and the index where it is inserted.

```
indInsertion(arr, size, element, 100, index);
```

[Copy](#)

4. In the *indInsertion* function, write the case of validity. Here, we'll check if the index is within the range [0,100]. We'll continue if it's valid; otherwise, return -1.
5. Create a *for* loop to shift the elements from the index to the last element to their adjacent right. This way, we'll create a void at the index we want to insert in.
6. Insert the element in the index. Return 1 on completion.

```
7. #include<stdio.h>
8.
9.
10.    void display(int arr[], int n){
11.        // Code for Traversal
12.        for (int i = 0; i < n; i++)
13.        {
14.            printf("%d ", arr[i]);
15.        }
16.        printf("\n");
17.    }
18.
19.    int indInsertion(int arr[], int size, int element, int
capacity, int index){
20.        // code for Insertion
21.        if(size>=capacity){
22.            return -1;
```

```
23.         }
24.         for (int i = size-1; i >=index; i--)
25.     {
26.         arr[i+1] = arr[i];
27.     }
28.     arr[index] = element;
29.     return 1;
30. }
31.
32. int main(){
33.     int arr[100] = {7, 8, 12, 27, 88};
34.     int size = 5, element = 45, index=1;
35.     display(arr, size);
36.     indInsertion(arr, size, element, 100, index);
37.     size +=1;
38.     display(arr, size);
39.     return 0;
40. }
```

[Copy](#)

Code Snippet 1: Insertion Operation Algorithm

Output of the above program:

```
7 8 12 27 88
```

```
7 45 8 12 27 88
```

[Copy](#)

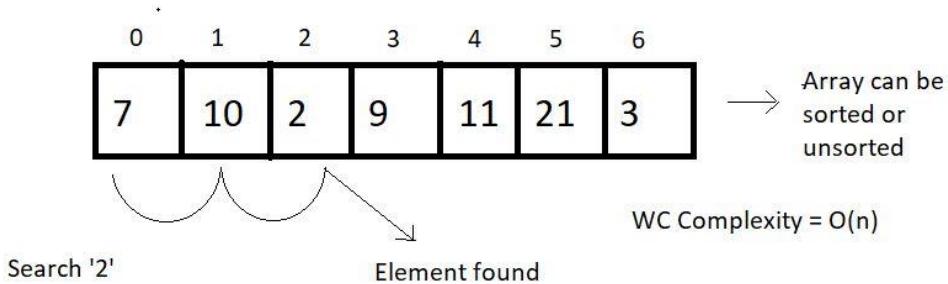
So, as you can see, element 45 got inserted at index 1, and the rest of the elements from this index to the last shifted to their right. And this is how we do an insertion in an array. We may come across a lot of variations to insert into an array, but we'll go slow for now.

Linear Vs Binary Search + Code in C Language (With Notes)

We have already covered the first three operators in an array, namely traversal, insertion, and deletion. Today, we will learn about the search operations in an array. You must already be familiar with these two methods we have for searching in an array, linear and binary search. We had used them quite a bit in our previous tutorials. We had analyzed them and got the result that for a sorted array, the fastest method to search is the binary one. Today, we'll learn how to code them and practically use them to search.

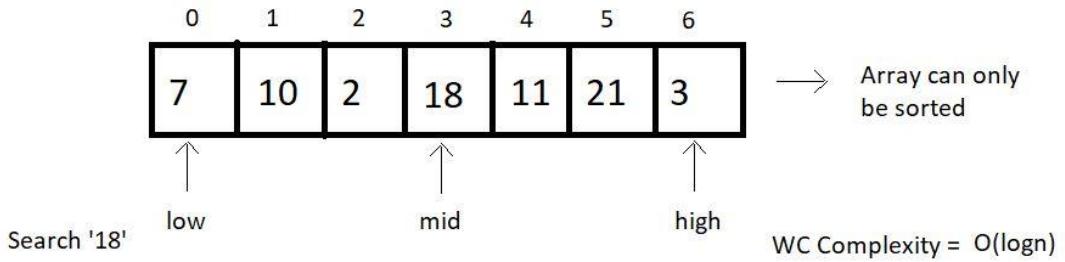
Linear Search:

This 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:

This 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.



From the above illustrations, we can draw a comparison between both the search methods based on their choice of arrays, operations, and worst-case complexities.

Linear Search	Binary Search
1. Works on both sorted and unsorted arrays	Works only on sorted arrays
2. Equality operations	Inequality operations
3. $O(n)$ WC Complexity	$O(\log n)$ WC Complexity

Table 1: Linear Search VS Binary Search

Let us now move on to the coding part of these methods. I have attached the snippet below. Refer to it while understanding.

Understanding the code snippet 1:

Linear Search:

1. We'll start with coding the linear search. Create an integer function `linearSearch`. This function will receive the array, its size, and the element to be searched as its parameters.
2. Run a `for` loop from its 0 to the last index, checking the `if` condition at every index whether the element at that index equals the search element. If yes, return the index, else continue the search.
3. If the element could not be found until the last, return -1.

Binary Search:

1. Create a function named `binarySearch` and pass the same three parameters as we did in linear search. Here, we will maintain three integer variables `low`, `mid`, and `high`. `Low` stores are the beginning of the search space, and `high` stores the end. `Mid` stores the middle element of our search space, which is $mid = (low+high)/2$.
2. Check whether the `mid` element equals the search element. If yes, return `mid`, else if the `mid` element is greater than the search element, then the search element must lie on the left side of the current space and `high` becomes `mid-1`, else if the `mid`

element is less than the search element, then we'll shift to the right side, and *low* becomes *mid+1*.

3. This way, we reduce our search space into half every time we repeat step 2. Now our new *mid* becomes $(\text{low}+\text{high})/2$, and we repeat step 2. And keep repeating until either we find the search element or the *low* becomes greater than the *high*.

```
#include<stdio.h>
```

```
int linearSearch(int arr[], int size, int element){  
    for (int i = 0; i < size; i++)  
    {  
        if(arr[i]==element){  
            return i;  
        }  
    }  
    return -1;  
}
```

```
int binarySearch(int arr[], int size, int element){  
    int low, mid, high;  
    low = 0;  
    high = size-1;  
    // Keep searching until low <= high  
    while(low<=high){  
        mid = (low + high)/2;  
        if(arr[mid] == element){  
            return mid;  
        }  
        if(arr[mid]<element){  
            low = mid+1;  
        }  
        else{  
            high = mid -1;  
        }  
    }  
}
```

```

        }
    }

    return -1;
}

int main(){
    // Unsorted array for linear search
    // int arr[] = {1,3,5,56,4,3,23,5,4,54634,56,34};

    // int size = sizeof(arr)/sizeof(int);

    // Sorted array for binary search
    int arr[] = {1,3,5,56,64,73,123,225,444};
    int size = sizeof(arr)/sizeof(int);
    int element = 444;
    int searchIndex = binarySearch(arr, size, element);
    printf("The element %d was found at index %d \n", element,
searchIndex);
    return 0;
}

```

[Copy](#)

Code Snippet 1: Linear search and Binary search codes.

Let's check if it works. Refer to the output below:

The element 444 was found at index 8

Introduction to Linked List in Data Structures (With Notes)

Linked lists are the new data structure we'll explore today. The study of linked lists will certainly be detailed, but first, I would like to inform you about one of 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:

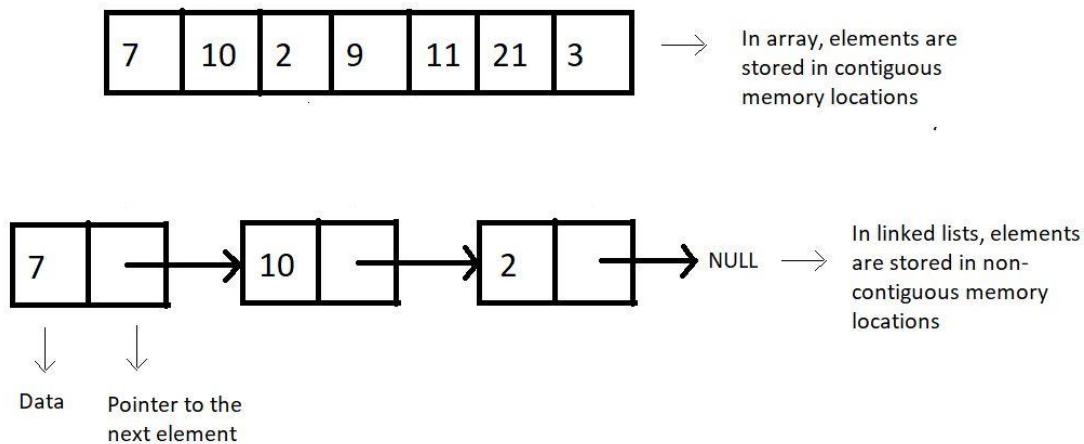


Figure 1: Arrays vs. Linked lists

Why 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.

Implementations

Linked lists are implemented in C language using a structure. You can refer to the snippet below.

Understanding the snippet below:

1. We construct a structure named *Node*.
2. 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
};
```

Copy

Linked List Data Structure: Creation and Traversal in C Language

In the last tutorial, we saw the differences between a linked list and an array. We saw the advantages and the limitations of a linked list. Today, we'll cover more on a linked lists' creation and learn how to traverse through it. If you haven't already, I recommend that you first go through the last tutorial.

I would anyway want to point some important things we learned about linked lists:

1. These are stored in non-contiguous memory locations.
2. Insertion and deletion in a linked list are very efficient in comparison to arrays.
3. An element called node holds the value as well as a pointer to the next element.

We can now move onto coding them. I've attached the snippet below for your referral. Follow them while understanding the same.

Understanding the snippet below:

1. An element in a linked list is a *struct Node*. It is made to hold integer *data* and a pointer of data type *struct Node**, as it has to point to another *struct Node*.
2. We'll create the below illustrated linked list.

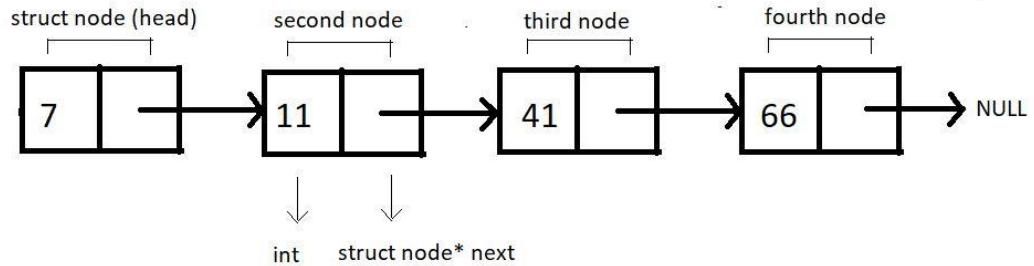


Figure 1: Illustration of the below implemented linked list.

3. We will always create individual nodes and link them to the next node via the arrow operator ' \rightarrow '.

4. First, we'll define a structure *Node* and create two of its members, an *int* variable *data*, to store the current node's value and a *struct node** pointer variable *next*.

5. Now, we can move on to our *main()* and start creating these nodes. We'll name the first node, *head*. Define a pointer to head node by *struct node* head*. And similarly for the other nodes. Request the memory location for each of these nodes from heap via *malloc* using the below snippet.

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

Copy

6. Link these nodes using the arrow operator and call the traversal function.

7. Create a void function *linkedlistTraversal* and pass into it the pointer to the head node.

8. Run a while loop while the pointer doesn't point to a NULL. And keep changing the pointer *next* each time you are done printing the data of the current node.

```
#include <stdio.h>
#include <stdlib.h>

struct Node
{
    int data;
```

```
    struct Node *next;
}

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

int main()
{
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 7;
    head->next = second;

    // Link second and third nodes
    second->data = 11;
```

```

second->next = third;

// Link third and fourth nodes
third->data = 41;
third->next = fourth;

// Terminate the list at the third node
fourth->data = 66;
fourth->next = NULL;

linkedListTraversal(head);
return 0;
}

```

[Copy](#)

Code Snippet 1: Creating and traversing in a linked list

Let's check if it works all fine. Refer to the output below.

```

Element: 7
Element: 11
Element: 41
Element: 66

```

Insertion of a Node in a Linked List Data Structure

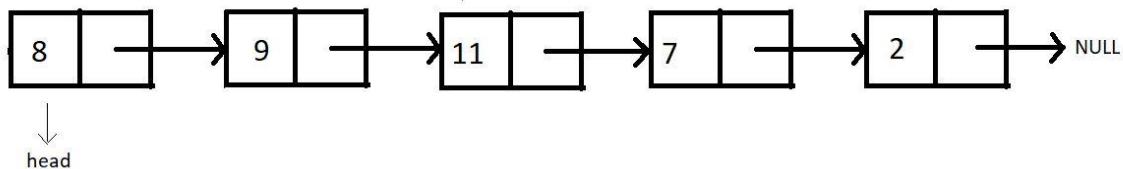
In the last tutorial, we had learned about creating a linked list using C structures and traversing through them while printing the values at each node. Today, we'll learn how to insert a node at some position and how that is more efficient than inserting an element in an array.

Inserting in an array has already been covered, and the following remarks were made:

1. A void has to be made to insert an element.
2. Creating a void causes the rest of the elements to shift to their adjacent right.
3. Time complexity: O(no. of elements shifted)

Inserting 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))
```

Copy

The above syntax will create a node, and the next thing one would need to do is set the data for this node.

```
ptr -> data = 9
```

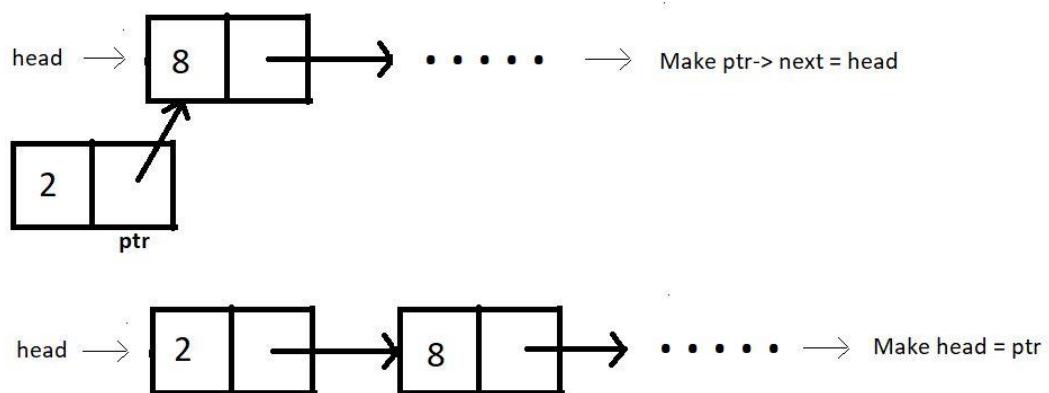
Copy

This will set the data.

Now, let's begin with each of these cases of insertion.

Case 1: Insert at the beginning

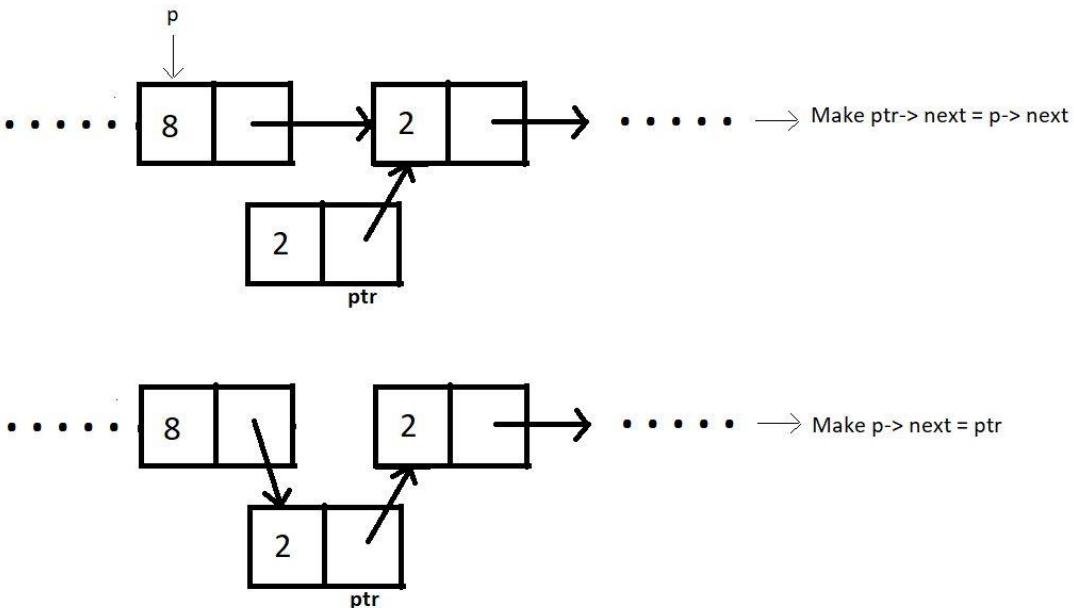
In order to insert the new node at the beginning, we would need to have the head pointer pointing to this new node and the new node's pointer to the current head.



Case 2: Insert in between:

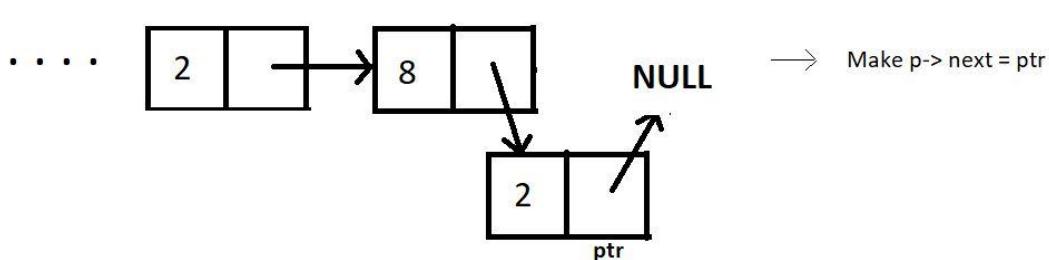
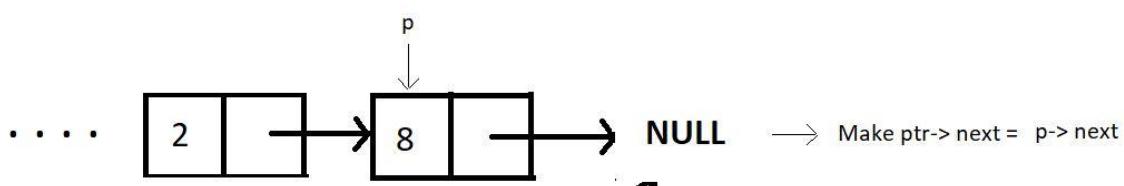
Assuming index starts from 0, we can insert an element at index $i > 0$ as follows:

1. Bring a temporary pointer `p` pointing to the node before the element you want to insert in the linked list.
2. Since we want to insert between 8 and 2, we bring pointer `p` to 8.



Case 3: 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.

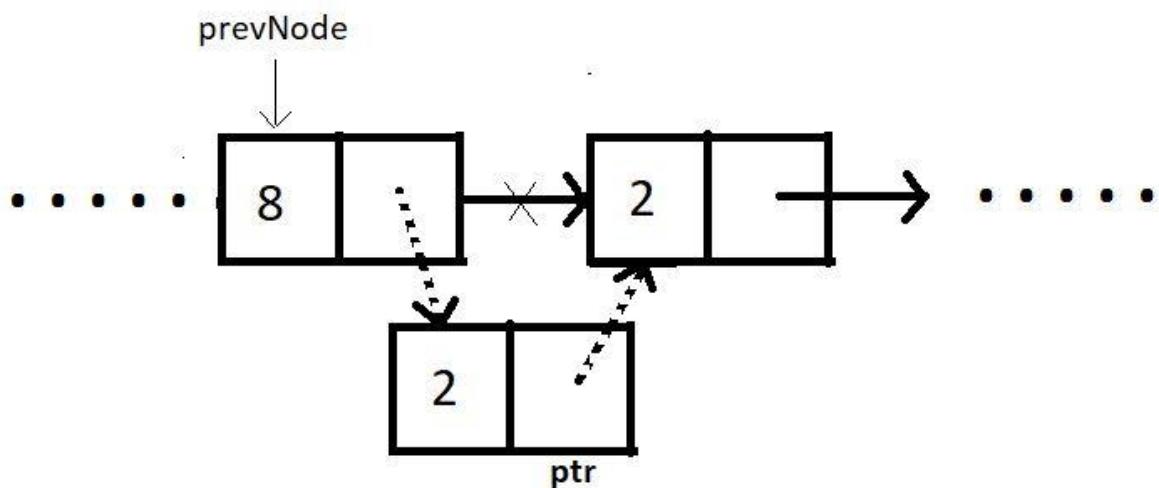


Case 4: Insert after a node:

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

ptr->next = prevNode->next;

prevNode->next = ptr;



Summarizing, 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 $O(1)$.

Insertion in a Linked List in C Language

So, since we are already finished learning about all the cases one would have encountered while inserting a new node into a linked list, we can now code them individually in C language.

Before we code, let's recall all the cases:

1. Inserting at the beginning -> Time complexity: $O(1)$
2. Inserting in between -> Time complexity: $O(n)$
3. Inserting at the end -> Time complexity: $O(n)$
4. Inserting after a given Node -> Time complexity: $O(1)$

Let's now code. I have attached the snippet below. Refer to it while understanding the steps.

Understanding the snippet below:

1. So, the first thing would be to create a struct *Node*. This is a known thing to us. We have covered this in our traversal video.
2. Create the *linkedlistTraversal* function. Earlier tutorials can be referred to.
3. Do include the header file `<stdlib.h>`, since we'll be using malloc to reserve memory locations.
4. As we did last time, create the same four nodes, the first node being the *head*. Define a pointer to head node by `struct node* head`. And similarly for the other nodes. Request the memory location for each of these nodes from the heap via malloc. Link these nodes using the arrow operator.
5. Now that we have created a linked list, we can create functions according to the different cases.

Insertion at the beginning:

1. Create a struct *Node** function *insertAtFirst* which will return the pointer to the new head.
2. We'll pass the current head pointer and the data to insert at the beginning, in the function.
3. Create a new struct *Node** pointer *ptr*, and assign it a new memory location in the heap.
4. Assign *head* to the next member of the *ptr* structure using *ptr-> next = head*, and the given data to its *data* member.
5. Return this pointer *ptr*.

6. // Case 1

```

7. struct Node * insertAtFirst(struct Node *head, int data){
8.     struct Node * ptr = (struct Node *) malloc(sizeof(struct
9.         Node));
10.
11.    ptr->data = data;
12.
13.    return ptr;
14. }
```

[Copy](#)

Code Snippet 1: Implementing insertAtFirst.

Insertion in between:

1. Create a struct Node* function *insertAtIndex* which will return the pointer to the head.
2. We'll pass the current head pointer and the data to insert and the index where it will get inserted, in the function.
3. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
4. Create a new struct Node* pointer pointing to *head*, and run a loop until this pointer reaches the index, where we are inserting a new node.
5. Assign *p->next* to the next member of the *ptr* structure using *ptr->next = p->next*, and the given data to its data member.
6. Break the connection between *p* and *p->next* by assigning *p->next* the new pointer. That is, *p->next = ptr*.
7. Return head.

```

// Case 2

struct Node * insertAtIndex(struct Node *head, int data, int index){

    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    struct Node * p = head;
    int i = 0;

    while (i!=index-1)
    {
        p = p->next;
```

```

    i++;
}

ptr->data = data;
ptr->next = p->next;
p->next = ptr;
return head;
}

```

[Copy](#)

Code Snippet 2: Implementing insertAtIndex.

Insertion at the end:

1. Inserting at the end is very similar to inserting at any index. The difference holds in the limit of the while loop. Here we run a loop until the pointer reaches the end and points to NULL.
2. Assign NULL to the next member of the new ptr structure using `ptr->next = NULL`, and the given data to its data member.
3. Break the connection between p and NULL by assigning `p->next` the new pointer. That is, `p->next = ptr`.
4. Return head.

```

// Case 3

struct Node * insertAtEnd(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;
    struct Node * p = head;

    while(p->next!=NULL){
        p = p->next;
    }

    p->next = ptr;
    ptr->next = NULL;
    return head;
}

```

```
}
```

[Copy](#)

Code Snippet 3: Implementing *insertAtEnd*.

Insertion after a given node:

1. Here, we already have a struct Node* pointer to insert the new node just next to it.
2. Create a struct Node* function *insertAfterNode* which will return the pointer to the head.
3. Pass into this function, the head node, the previous node, and the data.
4. Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap.
5. Since we already have a struct Node* *prevNode* given as a parameter, use it as *p* we had in the previous functions.
6. Assign *prevNode->next* to the next member of the *ptr* structure using *ptr->next = prevNode->next*, and the given data to its data member.
7. Break the connection between *prevNode* and *prevNode->next* by assigning *prevNode->next* the new pointer. That is, *prevNode->next = ptr*.
8. Return head.

```
9. // Case 4
10. struct Node * insertAfterNode(struct Node *head, struct
    Node *prevNode, int data){
11.     struct Node * ptr = (struct Node *)
        malloc(sizeof(struct Node));
12.     ptr->data = data;
13.
14.     ptr->next = prevNode->next;
15.     prevNode->next = ptr;
16.
17.     return head;
18. }
```

[Copy](#)

Code Snippet 4: Implementing *insertAfterNode*.

So those were the cases we had in insertion. Below is the whole source code.

```

#include<stdio.h>
#include<stdlib.h>

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

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

// Case 1
struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    ptr->next = head;
    return ptr;
}

// Case 2
struct Node * insertAtIndex(struct Node *head, int data, int index){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    struct Node * p = head;
    int i = 0;

```

```

while (i!=index-1)
{
    p = p->next;
    i++;
}
ptr->data = data;
ptr->next = p->next;
p->next = ptr;
return head;
}

// Case 3
struct Node * insertAtEnd(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;
    struct Node * p = head;

    while(p->next!=NULL){
        p = p->next;
    }
    p->next = ptr;
    ptr->next = NULL;
    return head;
}

// Case 4
struct Node * insertAfterNode(struct Node *head, struct Node
*prevNode, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;
}

```

```
ptr->next = prevNode->next;
prevNode->next = ptr;

return head;
}

int main(){
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 7;
    head->next = second;

    // Link second and third nodes
    second->data = 11;
    second->next = third;

    // Link third and fourth nodes
    third->data = 41;
    third->next = fourth;
```

```
// Terminate the list at the third node  
fourth->data = 66;  
fourth->next = NULL;
```

```
printf("Linked list before insertion\n");  
linkedListTraversal(head);  
// head = insertAtFirst(head, 56);  
// head = insertAtIndex(head, 56, 1);  
// head = insertAtEnd(head, 56);  
head = insertAfterNode(head, third, 45);  
printf("\nLinked list after insertion\n");  
linkedListTraversal(head);
```

```
return 0;
```

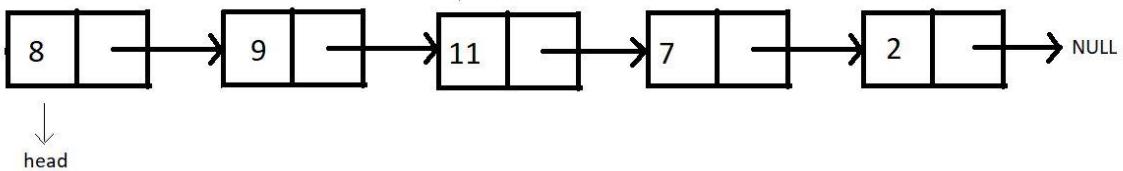
```
}
```

Deletion in a Linked List | Deleting a node from Linked List Data Structure

In the last two tutorials, we got to see how one can insert a node in a linked list. Today, we'll learn how to delete a node at some position. It will draw quite similarities with inserting a node, so this might be easy to you as well.

Inserting in a linked list:

Consider the following Linked List:



Insertion 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:

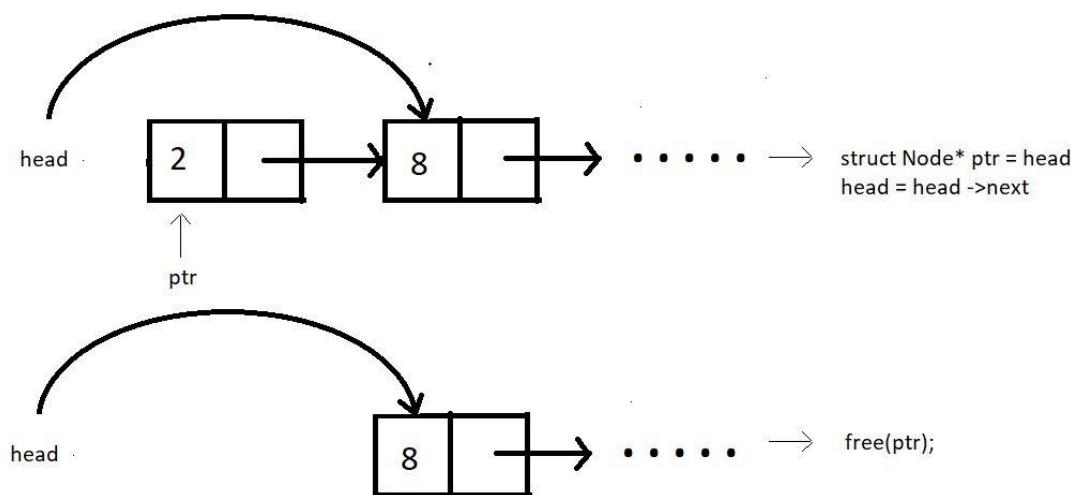
```
free(ptr);
```

Copy

The above syntax will free this node, that is, remove its reserved location in the heap.
Now, let's begin with each of these cases of insertion.

Case 1: Insert at the beginning:

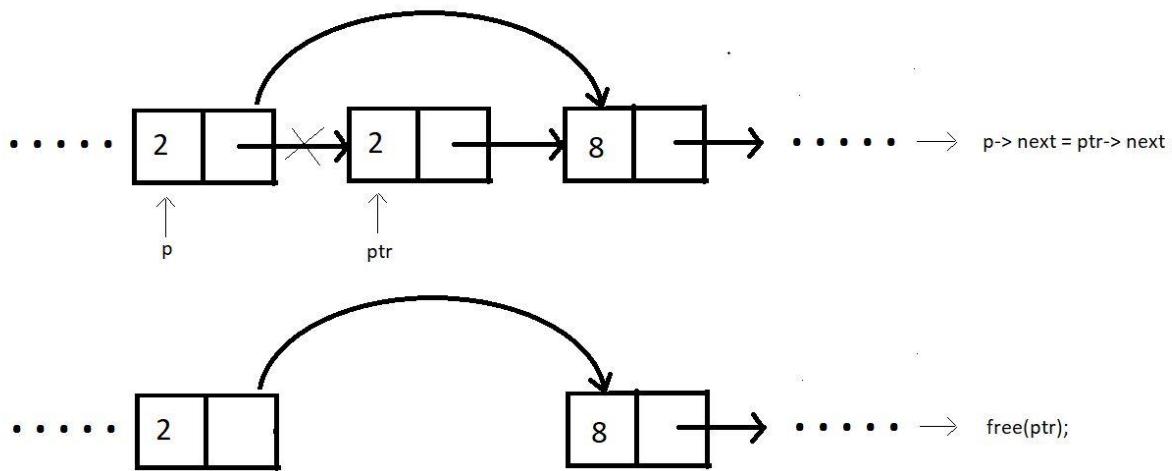
In order to delete the node at the beginning, we would need to have the head pointer pointing to the node second to the head node, that is, head-> next. And we would simply free the node that's left.



Case 2: Deleting at some index in between:

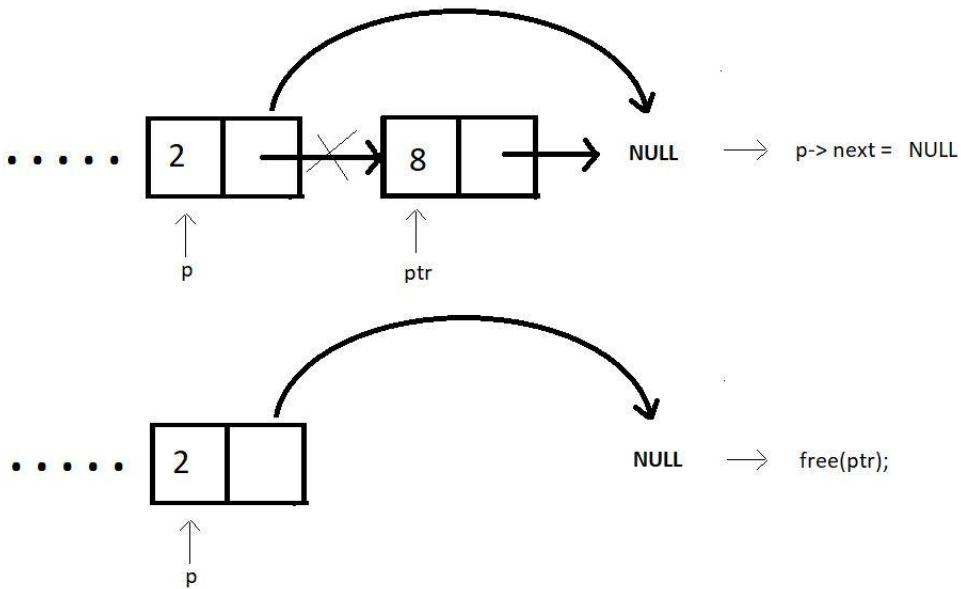
Assuming index starts from 0, we can delete an element from index $i > 0$ as follows:

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



Case 3: Deleting at 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 .

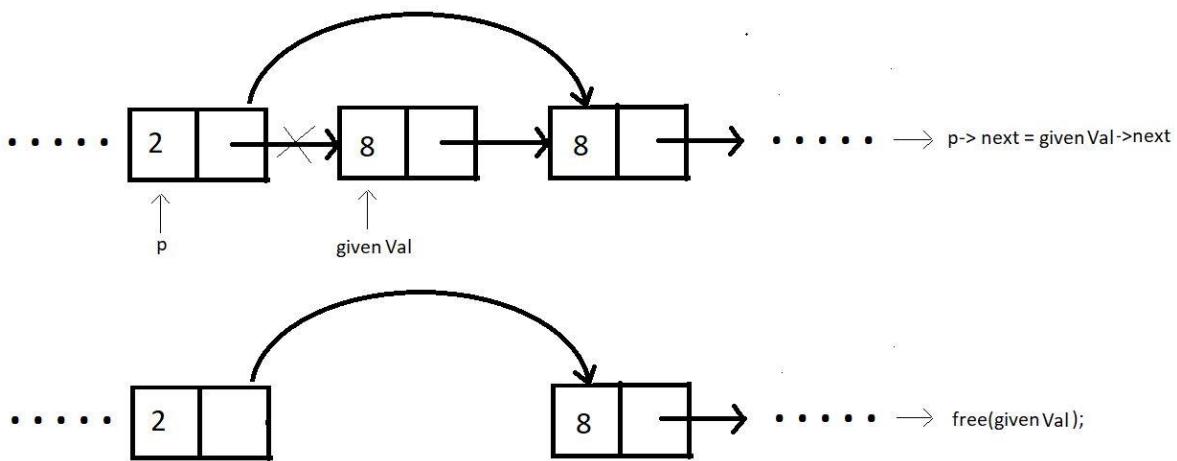


Case 4: Delete the first node with a given value:

Similar to the other cases, *ptr* can be deleted for a given value as well by following few steps:

1. *p->next = givenVal-> next;*
2. *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, i.e $O(1)$. 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

Delete a Node from Linked List (C Code For Deletion From Beginning, End, Specified Position & Key)

Now that we have a thorough understanding of all the cases that we will encounter when deleting an existing node from a linked list, we can code each one in C language.

Before we code, let's recall all the cases:

1. Deleting the first node -> Time complexity: $O(1)$
2. Deleting a node in between -> Time complexity: $O(n)$
3. Deleting the last node -> Time complexity: $O(n)$
4. Deleting the element with a given value from the linked list -> Time complexity: $O(n)$

Now, let's move on to the coding part. I have attached the snippet below. Refer to it while understanding the steps.

Understanding the snippet below:

1. You should have a good understanding of how to declare struct Nodes and traverse linked lists by now.
2. So, the first thing would be to create a struct *Node* and create the *linkedlistTraversal* function.
3. Do include the header file `<stdlib.h>`, since we'll use malloc to reserve memory locations.
4. Similar to what we did in the insertion video, create the four(choose any number) nodes. Request the memory location for each of these nodes from the heap via malloc. Link these nodes using the arrow operator.
5. And this is how we create a linked list of size 4. Let's see the cases of deletion.

Deleting the first node :

1. Create a struct *Node** function *deleteFirst* which will return the pointer to the new head after deleting the current head.
2. We'll pass the current head pointer in the function.
3. Create a new struct *Node** pointer *ptr*; and make it point to the current head.
4. Assign *head* to the next member of the list, by *head = head->next*, because this is going to be the new head of the linked list.
5. Free the pointer *ptr*. And return the head.

```
6. // Case 1: Deleting the first element from the linked list
```

```
7. struct Node * deleteFirst(struct Node * head){
```

```
8.     struct Node * ptr = head;
9.     head = head->next;
10.    free(ptr);
11.    return head;
12. }
```

Copy

Code Snippet 1: Deleting the first node

Deleting a node in between:

1. Create a struct Node* function *deleteAtIndex* which will return the pointer to the head.
2. In the function, we'll pass the current head pointer and the index where the node is to be deleted.
3. Create a new struct Node* pointer *p* pointing to *head*.
4. Create a new struct Node* pointer *q* pointing to *head->next*, and run a loop until this pointer reaches the index, from where we are deleting the node.
5. Assign *q->next* to the next member of the *p* structure using *p->next = q->next*.
6. Free the pointer *q*, because it has zero connections with the list now.
7. Return head.

```
8. // Case 2: Deleting the element at a given index from the
   linked list
9. struct Node * deleteAtIndex(struct Node * head, int index){
10.    struct Node *p = head;
11.    struct Node *q = head->next;
12.    for (int i = 0; i < index-1; i++)
13.    {
14.        p = p->next;
15.        q = q->next;
16.    }
17.
18.    p->next = q->next;
19.    free(q);
20.    return head;
```

```
21. }
```

Copy

Code Snippet 2: Deleting a node in between

Deleting the last node :

1. Deleting the last node is quite similar to deleting from any other index. The difference holds in the limit of the while loop. Here we run a loop until the pointer reaches the end and points to NULL.
2. Assign NULL to the next member of the p structure using p->next = NULL.
3. Break the connection between q and NULL by freeing the ptr q.
4. Return head.

```
5. // Case 3: Deleting the last element
6. struct Node * deleteAtLast(struct Node * head){
7.     struct Node *p = head;
8.     struct Node *q = head->next;
9.     while(q->next !=NULL)
10.    {
11.        p = p->next;
12.        q = q->next;
13.    }
14.
15.    p->next = NULL;
16.    free(q);
17.    return head;
18. }
```

Copy

Code Snippet 3: Deleting the last node

Deleting the element with a given value from the linked list :

1. Here, we already have a value that needs to be deleted from the list. The main thing is that we'll delete only the first occurrence.

2. Create a struct Node* function *deleteByValue* which will return the pointer to the head.
3. Pass into this function the head node, the value which needs to be deleted.
4. Create a new struct Node* pointer *p* pointing to the head.
5. Create another struct Node* pointer *q* pointing to the next of head.
6. Run a while loop until the pointer *q* encounters the given value or the list finishes.
7. If it encounters the value, delete that node by making *p* point the next node, skipping the node *q*. And free *q* from memory.
8. And if the list just finishes, it means there was no such value in the list. Continue without doing anything.
9. Return head.

```

10.    // Case 4: Deleting the element with a given value from
        the linked list
11.    struct Node * deleteByValue(struct Node * head, int
        value){
12.        struct Node *p = head;
13.        struct Node *q = head->next;
14.        while(q->data!=value && q->next!= NULL)
15.        {
16.            p = p->next;
17.            q = q->next;
18.        }
19.
20.        if(q->data == value){
21.            p->next = q->next;
22.            free(q);
23.        }
24.        return head;
25.    }

```

[Copy](#)

Code Snippet 4: Deleting the element with a given value from the linked list

So, this was all about deletion in the linked list data structure. Here is the whole source code:

```
#include <stdio.h>
#include <stdlib.h>

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

```
void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
```

```
// Case 1: Deleting the first element from the linked list
struct Node * deleteFirst(struct Node * head){
    struct Node * ptr = head;
    head = head->next;
    free(ptr);
    return head;
}
```

```
// Case 2: Deleting the element at a given index from the linked list
struct Node * deleteAtIndex(struct Node * head, int index){
    struct Node *p = head;
    struct Node *q = head->next;
    for (int i = 0; i < index-1; i++)
        p = p->next;
    q = p->next;
    p->next = q->next;
    free(q);
    return head;
}
```

```

    {
        p = p->next;
        q = q->next;
    }

    p->next = q->next;
    free(q);
    return head;
}

// Case 3: Deleting the last element
struct Node * deleteAtLast(struct Node * head){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->next !=NULL)
    {
        p = p->next;
        q = q->next;
    }

    p->next = NULL;
    free(q);
    return head;
}

// Case 4: Deleting the element with a given value from the linked
list
struct Node * deleteAtIndex(struct Node * head, int value){
    struct Node *p = head;
    struct Node *q = head->next;
    while(q->data!=value && q->next!= NULL)

```

```

    {
        p = p->next;
        q = q->next;
    }

    if(q->data == value){
        p->next = q->next;
        free(q);
    }
    return head;
}

int main()
{
    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 4;
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;
}

```

```

        // Link third and fourth nodes
        third->data = 8;
        third->next = fourth;

        // Terminate the list at the third node
        fourth->data = 1;
        fourth->next = NULL;

        printf("Linked list before deletion\n");
        linkedListTraversal(head);

        // head = deleteFirst(head); // For deleting first element of the
linked list
        // head = deleteAtIndex(head, 2);
        head = deleteAtLast(head);
        printf("Linked list after deletion\n");
        linkedListTraversal(head);

        return 0;
}

```

Circular Linked List and Operations in Data Structures (With Notes)

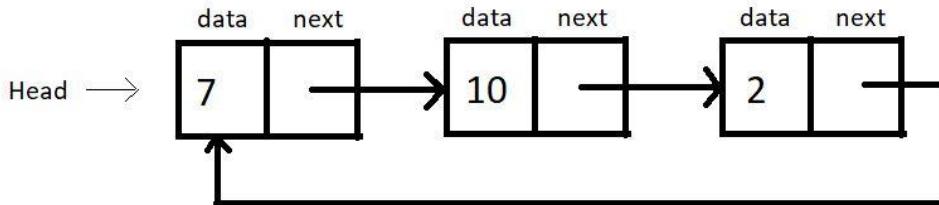
Till now, we have covered linked lists, which consist of a head, the body, and an end pointing to NULL. Basically, it was linear. We could do traversal, insertion, deletion, searching, and many more operations while traversing to the end of it. Today, we'll see a variant of it, circular linked lists. We'll also see all those operations that we could do in a linear linked list and their implementations in a circular linked list.

Introduction:

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 on 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.

Traversal:

- Traversal in a circular linked list can be achieved by creating a new struct Node* pointer p, which starts from the head and goes through the list until it points again at the head. So, this is how we go through this circle only once, visiting each node.
- And since traversal is achieved, all the other operations in a circular linked list become as easy as doing things in a linear linked list.
- One thing that may have sounded confusing to you is that there is a head but no starting of this circular linked list. Yes, that is the case; we have this head pointer just to start or inception in this list and for our convenience while operating on it. There is no first element here.

Circular Linked Lists: Operations in C Language

In the last tutorial, we learned about this new data structure, the circular linked lists. Additionally, we discussed the difference and similarities between a circular linked list and a linear linked list.

Let me quickly summarize some of the most important points:

1. Unlike singly-linked lists, a circular linked list has no node pointing to NULL. Hence it has no end. The last element points at the head node.

2. All the operations can still be done by maintaining an extra pointer fixed at the head node.
3. A circular linked list has a head node, but no starting node.

We even learned traversing through the circular linked list using the do-while approach. Today, we'll see one of the operations, insertion in a doubly-linked list with the help of C language.

Now, let's move on to the coding part. I have attached the snippet below. Refer to it while understanding the steps.

Creating the circular linked list:

1. Creating a circular linked list is no different from creating a singly linked list. One thing we do differently is that instead of having the last element to point to NULL, we'll make it point to the head.
2. Refer to those previous tutorials while creating these nodes and connecting them. This is the third time we are doing it, and I believe you must have gained that confidence.

```

3. struct Node
4. {
5.     int data;
6.     struct Node *next;
7. };
8. int main(){
9.
10.    struct Node *head;
11.    struct Node *second;
12.    struct Node *third;
13.    struct Node *fourth;
14.
15.    // Allocate memory for nodes in the linked list in
16.    // Heap
17.    head = (struct Node *)malloc(sizeof(struct Node));
18.    second = (struct Node *)malloc(sizeof(struct Node));
19.    third = (struct Node *)malloc(sizeof(struct Node));
20.    fourth = (struct Node *)malloc(sizeof(struct Node));

```

```

20.
21.         // Link first and second nodes
22.         head->data = 4;
23.         head->next = second;
24.
25.         // Link second and third nodes
26.         second->data = 3;
27.         second->next = third;
28.
29.         // Link third and fourth nodes
30.         third->data = 6;
31.         third->next = fourth;
32.
33.         // Terminate the list at the third node
34.         fourth->data = 1;
35.         fourth->next = head;
36.
37.         return 0;
38.     }

```

Code Snippet 1: Creating the circular linked list

Traversing the circular linked list:

1. Create a void function *linkedListTraversal* and pass the head pointer of the linked list to the function.
2. In the function, create a pointer *ptr* pointing to the head.
3. Run a *do-while* loop until *ptr* reaches the last node, and *ptr->next* becomes *head*, i.e. *ptr->next = head*. And keep printing the data of each node.
4. So, this is how we traverse through a circular linked list. And *do-while* was the key to make it possible.

```

5. void linkedListTraversal(struct Node *head){
6.     struct Node *ptr = head;

```

```

7.     do{
8.         printf("Element is %d\n", ptr->data);
9.         ptr = ptr->next;
10.    }while(ptr!=head);
11.

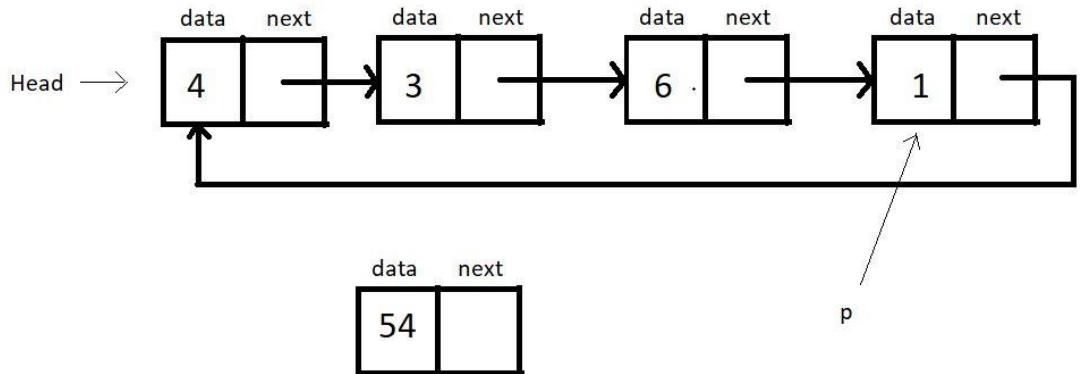
```

[Copy](#)

Code Snippet 2: Traversing the circular linked list

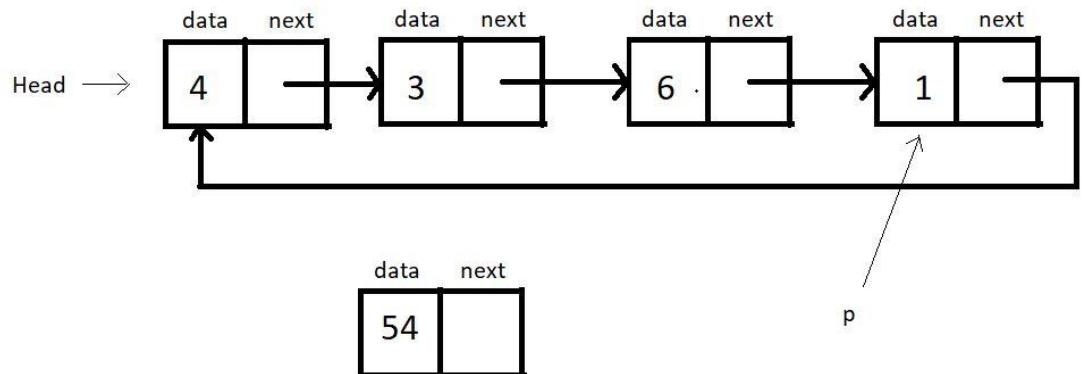
Inserting into a circular linked list:

- I'll just cover the insertion part, and that too on the head. Rest of the variations, I believe, you'll be able to do yourselves. Things are very similar to that of singly-linked lists.
- Create a struct Node* function *insertAtFirst* which will return the pointer to the new head.
- We'll pass the current head pointer and the data to insert at the beginning, in the function.
- Create a new struct Node* pointer *ptr*, and assign it a new memory location in the heap. This is our new node pointer. Make sure you don't forget to include the header file <stdlib.h>.
- Create another struct node * pointer *p* pointing to the next of the head. *p = head->next*.
- Run a *while* loop until the *p* pointer reaches the end element and *p->next* becomes the head.



- Now, assign *ptr* to the next of *p*, i.e. *p->next = ptr*. And *head* to the next of *ptr*, i.e. *ptr->next = head*.

8. Now, the new head becomes ptr. $head = ptr$.



9. Return head.

```
10. struct Node * insertAtFirst(struct Node *head, int data){  
11.     struct Node * ptr = (struct Node *)  
12.         malloc(sizeof(struct Node));  
13.         ptr->data = data;  
14.         struct Node * p = head->next;  
15.         while(p->next != head){  
16.             p = p->next;  
17.         }  
18.         // At this point p points to the last node of this  
19.         // circular linked list  
20.         p->next = ptr;  
21.         ptr->next = head;  
22.         head = ptr;  
23.         return head;  
24.     }  
25. }
```

Copy

Code Snippet 3: Inserting into a circular linked list

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

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

void linkedListTraversal(struct Node *head){
    struct Node *ptr = head;
    do{
        printf("Element is %d\n", ptr->data);
        ptr = ptr->next;
    }while(ptr!=head);
}

struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    struct Node * p = head->next;
    while(p->next != head){
        p = p->next;
    }

    // At this point p points to the last node of this circular linked
list

    p->next = ptr;
    ptr->next = head;
```

```
    head = ptr;
    return head;

}

int main(){

    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 4;
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;

    // Link third and fourth nodes
    third->data = 6;
    third->next = fourth;

    // Terminate the list at the third node
    fourth->data = 1;
```

```
    fourth->next = head;  
  
    return 0;  
}
```

Copy

Code Snippet 4: Insertion and traversal in a circular linked list

We'll now see whether the functions work accurately. Let's insert a few nodes at the beginning.

```
printf("Circular Linked list before insertion\n");  
linkedListTraversal(head);  
head = insertAtFirst(head, 54);  
head = insertAtFirst(head, 58);  
head = insertAtFirst(head, 59);  
printf("Circular Linked list after insertion\n");  
linkedListTraversal(head);
```

Copy

Code snippet 5: Using the insertAtFirst function

Refer to the output below:

```
Circular Linked list before insertion  
Element is 4  
Element is 3  
Element is 6  
Element is 1  
Circular Linked list after insertion  
Element is 59  
Element is 58  
Element is 54  
Element is 4  
Element is 3
```

Element is 6

Element is 1

Copy

As you can see, all the elements we passed into the `insertAtFirst` function got added at the beginning. So, it is indeed working.

And this was all about a circular linked list. We won't go doing other operations. You should all carry out other operations yourselves. Believe me, it ain't a tough job. We have next in the series another variant of a linked list called the doubly linked lists.

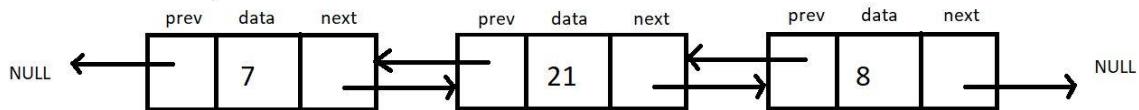
Doubly Linked Lists Explained With Code in C Language

So, we have already talked about a lot of things under linked lists. We talked about the singly-linked lists, which had both a head node and the last node pointing to the NULL. We also talked about circular linked lists, which had no ending but an arbitrary head node. We also learned about all the basic operations (traversal, insertion, deletion, search) that we could do on both these variants of a linked list. Our takeaway from all this is that we can perform all these operations on any variant of a linked list regardless of their structure and properties. We'll see one such thing today as well. We'll draw out similarities between the structure we'll handle today and the ones we did before.

What is a 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 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.
- A node comprises three parts, the data, a pointer to the next node, and a pointer to the previous node.
- Head node has the pointer to the previous node pointing to NULL.

Implementation in C:

Let's try implementing a doubly linked list in our codes. We'll have a struct *Node* as before. The only information added to this struct *Node* is a struct *Node** pointer to the previous node. Let's name this *prev*.

This new information makes us travel in both directions, but using it follows the use of more memory space for a single node that now comprises three members. It is because of this we have a singly linked list.

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

Copy

Code Snippet 1: Implementation of a doubly linked list.

Operations on 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. It very much follows the fact, "With great power, comes great responsibility." :)

Task: Try implementing a traversal algorithm to traverse once to the right and once to the left. Print the data in both cases.

So this was all we had in linked lists. It was a great segment. Let's give it an end here. We have so many things coming. So don't miss out on this ever. Keep revising things at regular intervals.

Thank you for being with me throughout. I hope you enjoyed the tutorial. If you appreciate my work, please let your friends know about this course too. Don't forget to download the notes from the link given below. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. See you all in the next tutorial where try to code these inserting methods. Till then, keep learning.

[Download Notes here](#)

Code as described/written in the video:

```
#include<stdio.h>  
#include<stdlib.h>
```

```

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

void linkedListTraversal(struct Node *head){
    struct Node *ptr = head;
    do{
        printf("Element is %d\n", ptr->data);
        ptr = ptr->next;
    }while(ptr!=head);
}

struct Node * insertAtFirst(struct Node *head, int data){
    struct Node * ptr = (struct Node *) malloc(sizeof(struct Node));
    ptr->data = data;

    struct Node * p = head->next;
    while(p->next != head){
        p = p->next;
    }
    // At this point p points to the last node of this circular linked
list

    p->next = ptr;
    ptr->next = head;
    head = ptr;
    return head;

}

```

```
int main(){

    struct Node *head;
    struct Node *second;
    struct Node *third;
    struct Node *fourth;

    // Allocate memory for nodes in the linked list in Heap
    head = (struct Node *)malloc(sizeof(struct Node));
    second = (struct Node *)malloc(sizeof(struct Node));
    third = (struct Node *)malloc(sizeof(struct Node));
    fourth = (struct Node *)malloc(sizeof(struct Node));

    // Link first and second nodes
    head->data = 4;
    head->next = second;

    // Link second and third nodes
    second->data = 3;
    second->next = third;

    // Link third and fourth nodes
    third->data = 6;
    third->next = fourth;

    // Terminate the list at the third node
    fourth->data = 1;
    fourth->next = head;

    printf("Circular Linked list before insertion\n");
    linkedListTraversal(head);
}
```

```
    head = insertAtFirst(head, 54);
    head = insertAtFirst(head, 58);
    head = insertAtFirst(head, 59);
    printf("Circular Linked list after insertion\n");
    linkedListTraversal(head);
    return 0;
}
```

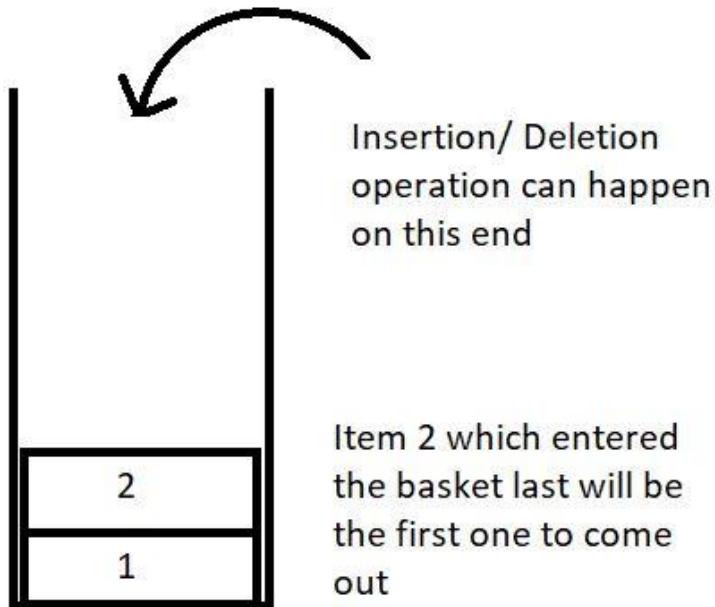
Introduction to Stack in Data Structures

It has been a while since we started this DSA course. We saw array ADT, linked lists and their variants, their implementation, and their operations. From this tutorial on, we will start learning about stack data structures.

Introduction:

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 below. 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:

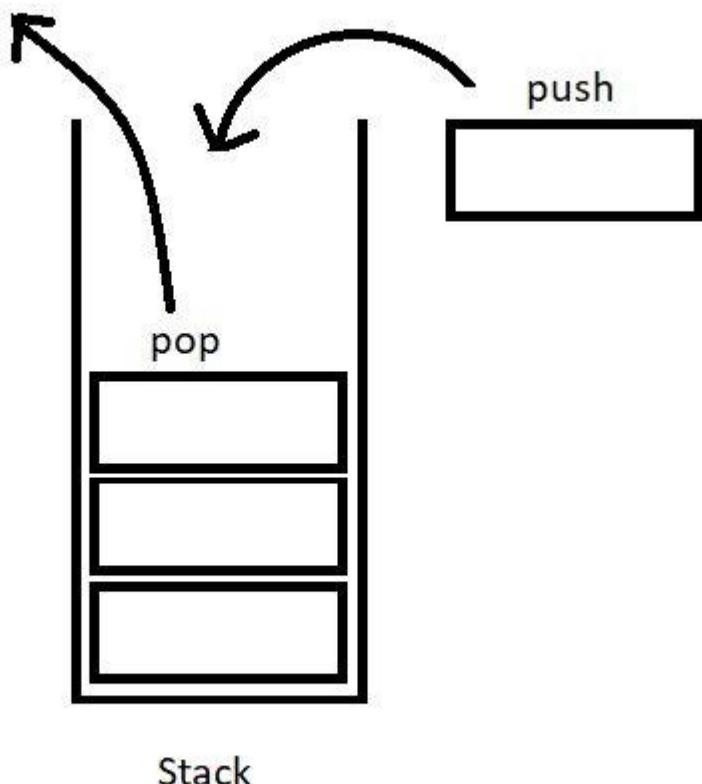
1. We have talked about function calls before as well. A function until it returns reserves a space in the memory stack. Any function embedded in some function comes above the parent function in the stack. So, first, the embedded function ends, and then the parent one. Here, the function called last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

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 out. Along with that, we need the space for the other elements to get in and their data.

Here are some of the basic operations we would want to perform on stacks:

1. `push()`: to push an element into the stack
2. `pop()`: to remove the topmost element from the stack



3. `peek(index)`: to return the value at a given index
4. `isempty()` / `isfull()` : to determine whether the stack is empty or full to carry efficient push and pull operations.

Implementing Stack Using Array in Data Structures

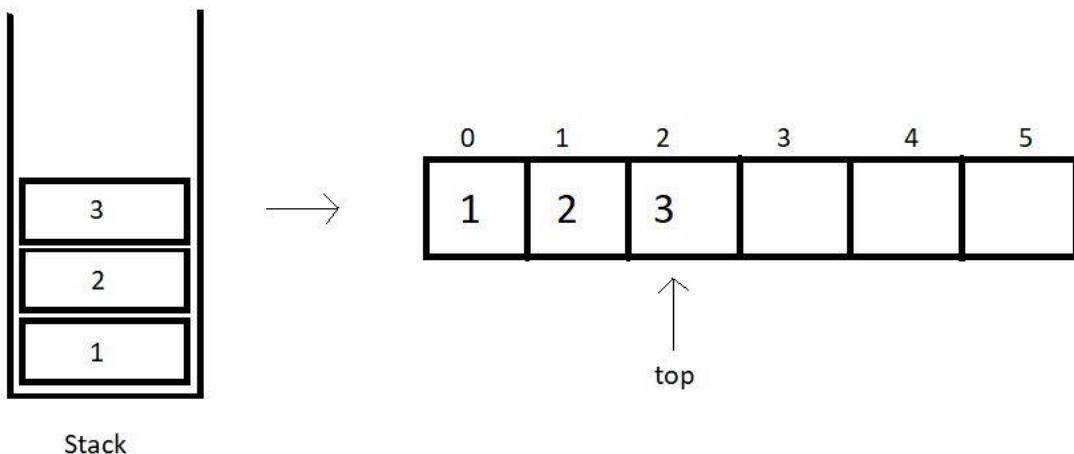
In the last tutorial, we learned about the stack data structure and its applications in several programming phases. We also discussed some of the operations possible on

a stack. Today, we'll try to implement these ideas on a stack using arrays. Although we have another choice of linked lists.

If you remember, a stack is a collection of elements following LIFO(Last In First Out); the element that gets pushed the last is the first one to come out of the stack.

Stack Using an Array

If we recall, arrays are linear data structures whose elements are indexed, and the elements can be accessed in constant time with their index. 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.

1. 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.

2. 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;  
}
```

Copy

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,

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

Follow the example below for defining a stack:

```
struct stack S;  
S.size = 80;  
S.top = -1;  
S.arr = (int*)malloc(S.size*sizeof(int));
```

Copy

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

We can now move on implementing the stack ADT, particularly their operators. We have in the list, push and pull, peek, and isempty/full operation. Let's visit them one by one.

push():

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():

Pop means to remove the last element entered in the stack, and that element has the index top. So, this becomes an easy job. 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.

We will also see the other operations in our tutorial dedicated to operations on Stack. Today, we just used an array to implement stacks. You might have found this difficult to digest, but you should go over it again and again. Read the notes provided.

C Code For Implementing Stack Using Array in Data Structures

In the last tutorial, we covered how to implement stacks by using arrays. We also dealt with the basic structure behind defining a stack with all the

customizations. We also learned about some of the operations one could do while handling stacks. Today, we'll try implementing stacks using arrays in C.

I've attached the snippet below. Keeping that in mind will help you understand the implementation.

Understanding the code snippet 1:

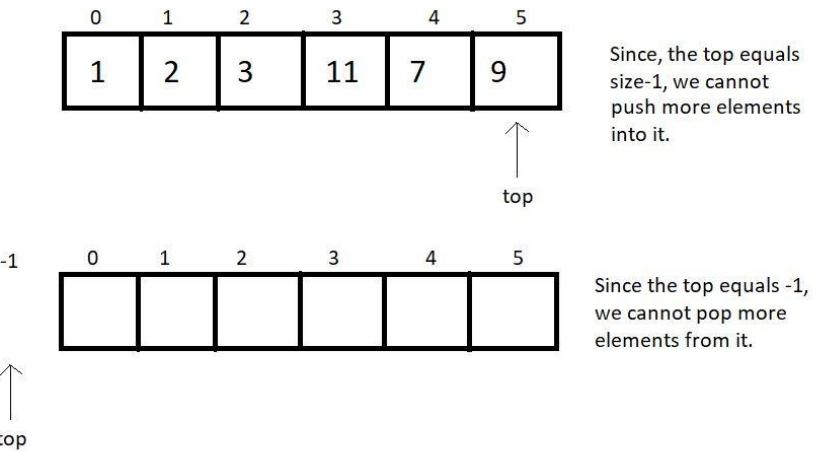
1. So, the first thing would be to create the struct *Stack* we discussed in the previous tutorial. Include three members, an integer variable to store the size of the stack, an integer variable to store the index of the topmost element, and an integer pointer to hold the address of the array.

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

Copy

Code Snippet 1: Creating stack

2. In *main*, create a struct stack *s*, and assign a value 80(you can assign any value of your choice) to its size, -1 to its top, and reserve memory in heap using malloc for its pointer *arr*. Don't forget to include <stdlib> .
3. We have one more method to declare these stacks. We can define a struct stack pointer *s*, and use the arrow operators to deal with their members. The advantage of this method is that we can pass these pointers as references into functions very conveniently.
4. Before we advance to pushing elements in this stack, there are a few conditions to deal with. We can only push an element in this stack if there is some space left or the top is not equal to the last index. Similarly, we can only pop an element from this stack if some element is stored or the top is not equal to -1.



5. So, let us first write functions to check whether these stacks are empty or full.

6. Create an integer function *isEmpty*, and pass the pointer to the stack as a parameter. In the function, check if the top is equal to -1. If yes, then it's empty and returns 1; otherwise, return 0.

```
int isEmpty(struct stack *ptr)
{
    if (ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
```

Copy

Code Snippet 2: Implementing isEmpty

7. Create an integer function *isFull*, and pass the pointer to the stack as a parameter. In the function, check if the top is equal to (size-1). If yes, then it's full and returns 1; otherwise, return 0.

```
int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Copy

Code Snippet 3: Implementing isFull

Here is the whole Source Code:

```
#include <stdio.h>
#include <stdlib.h>

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

int isEmpty(struct stack *ptr)
{
```

```
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int main()
{
    // struct stack s;
    // s.size = 80;
    // s.top = -1;
    // s.arr = (int *) malloc(s.size * sizeof(int));
}
```

```
    struct stack *s;
    s->size = 80;
    s->top = -1;
    s->arr = (int *)malloc(s->size * sizeof(int));

    return 0;
}
```

Copy

Code Snippet 4: Implementing isEmpty and isFull

Since there is no element inside the stack, we can now check if it's empty.

```
// Check if stack is empty
if(isEmpty(s)){
    printf("The stack is empty");
}
else{
    printf("The stack is not empty");
}
```

Copy

Code Snippet 5: Calling the function isEmpty

Output:

```
The stack is empty
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

Copy

Figure 1: Output of the above program

So, yes, that worked fine. Now, we can easily push some elements inside this stack manually to test this *isEmpty* function. This should not be a tough job. Just insert an element at top+1 and increment top by 1.

```
// Pushing an element manually
    s->arr[0] = 7;
    s->top++;

// Check if stack is empty
if(isEmpty(s)){
    printf("The stack is empty");
}
else{
    printf("The stack is not empty");
}
```

Copy

Code Snippet 6: Inserting an element in the stack

Output after inserting an element:

```
The stack is not empty
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

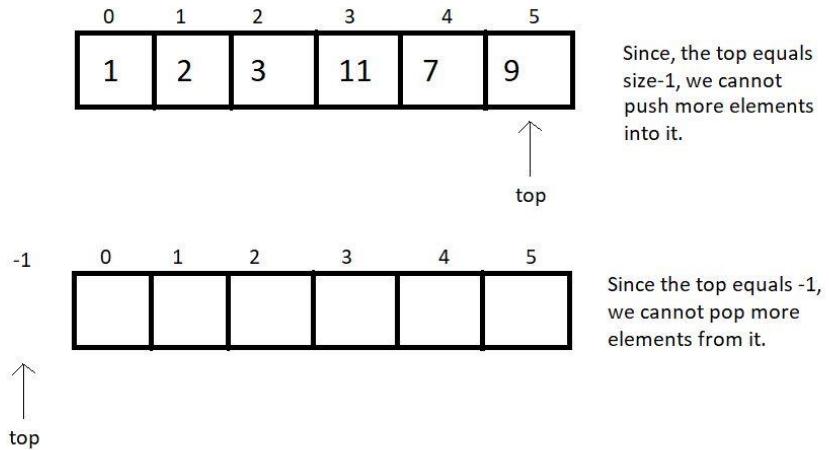
Copy

Push, Pop and Other Operations in Stack Implemented Using an Array

We had already finished the basics of a stack, and its implementation using arrays. We have gained enough confidence by writing the codes for implementing stacks using arrays in C. Now we can learn about the operations one can perform on a stack while executing them using arrays.

We concluded our last tutorial with two of the most important points:

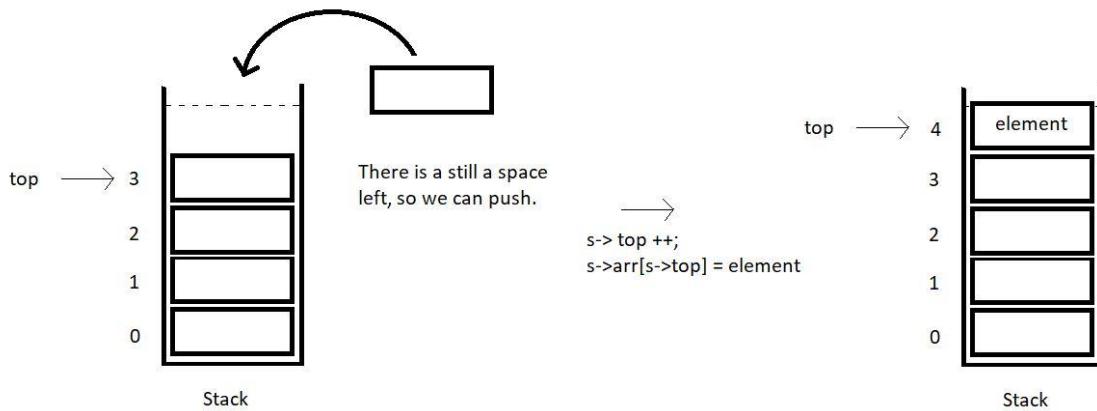
1. One cannot push more elements into a full-stack.
2. One cannot pop any more elements from an empty stack.



Declaring a stack was done in the last tutorial as well. Let's keep that in mind as we proceed.

Operation 1: Push-

1. The first thing is to define a stack. Suppose we have the creating stack and declaring its fundamentals part done, then pushing an element requires you to first check if there is any space left in the stack.
2. Call the *isFull* function which we did in the previous tutorial. 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.

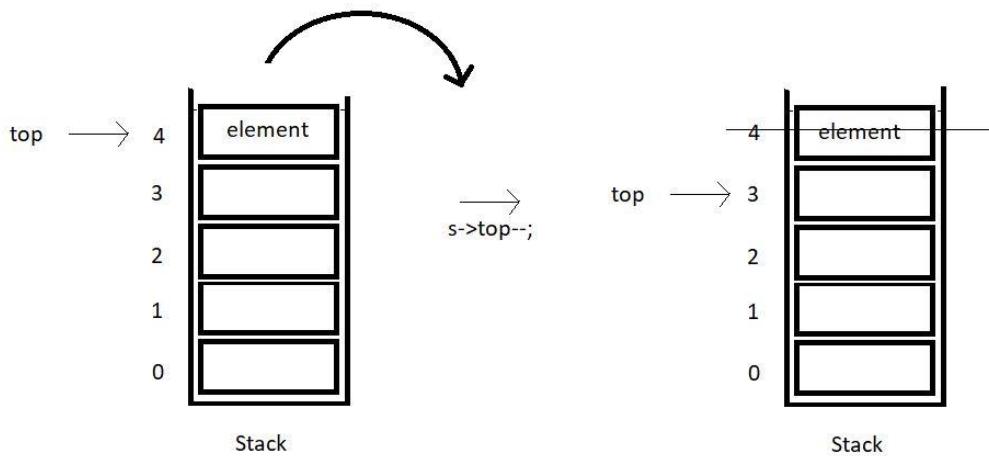


3. So, this is how we push an element in a stack array. Suppose we have an element x to insert in a stack array of size 4. We first checked if it was full, and found it was not full. We retrieved its top which was 3 here. We made it 4 by increasing it once. Now, just insert the element x at index 4, and we are done.

Operation 2: Pop-

Popping an element is very similar to inserting an element. You should first give it a try yourself. There are very subtle changes.

1. The first thing again is to define a stack. Get over with all the fundamentals. You must have learnt that by now. Then popping an element requires you to first check if there is any element left in the stack to pop.
2. Call the `isEmpty` function which we practiced in the previous tutorial. 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.



3. So, this is how we pop an element from a stack array. Suppose we make a `pop` call in a stack array of size 4. We first checked if it was empty, and found it was not empty. We retrieved its `top` which was 4 here. Stored the `element` at 4. We made it 3 by decreasing it once. Now, just return the element `x`, and popping is done.

So, this was known about the concepts behind pushing and popping elements in a stack. You'll enjoy it more when we'll implement their codes in C in the next tutorial. I always encourage self-motivation. Try them yourselves first and let me know if you could. We still have a lot to do. We still have many operations to complete. We'll cover them all slowly and steadily. Enjoy the ride.

Coding Push(), Pop(), isEmpty() and isFull() Operations in Stack Using an Array| C Code For Stack

In the last tutorial, we covered the concepts behind the push and the pop operations on a stack implemented with an array. We saw how easy it is, to push an element in a non-full array, and to pop an element from a non-empty array. Today, we'll be interested in coding these implementations in C.

If you didn't follow me in the last tutorial, I would recommend visiting that first. Because it not only covered the concepts but the implementation part

as well. I have attached the code snippet below. Refer to it while we learn to code:

Understanding the code snippet 1:

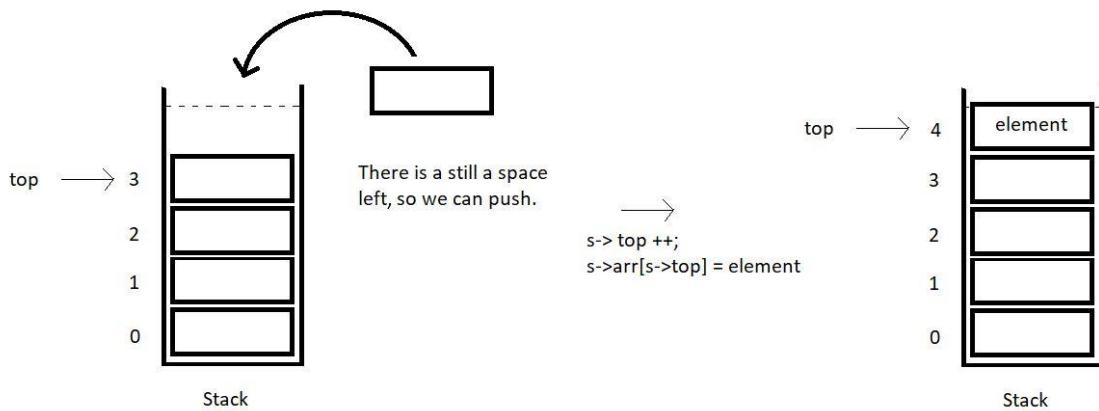
1. There is nothing new now. You can just construct a struct *stack*, with all its three members, *size*, to store the size of the array used to handle this stack, *top*, to store the index of the topmost element in the stack and *arr*, a pointer to store the address of the array used. I will skip over this because we have done it before.

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

Copy

Code Snippet 1: Creating stack struct

2. In the main, define a struct stack pointer *sp*, which will store the address of the stack. Since we are using malloc to reserve the memory in heap for this stack, don't forget to include the header file <stdlib.h>.
3. Initialize all the elements of the stack with some values.
4. Create the integer functions *isFull* and *isEmpty*. We have covered them in detail [here](#). These functions are a must, while we use the push or the pop operations.
5. Create a void function *push*, and pass into it the address of the stack using the pointer *sp* and the value which is to be pushed.
6. Don't forget to first check if our stack still has some space left to push elements. Use *isFull* function for that. If it returns 1, this is the case of stack overflow, otherwise, increase the top element of the stack by 1, and insert the value at this new top of the array.



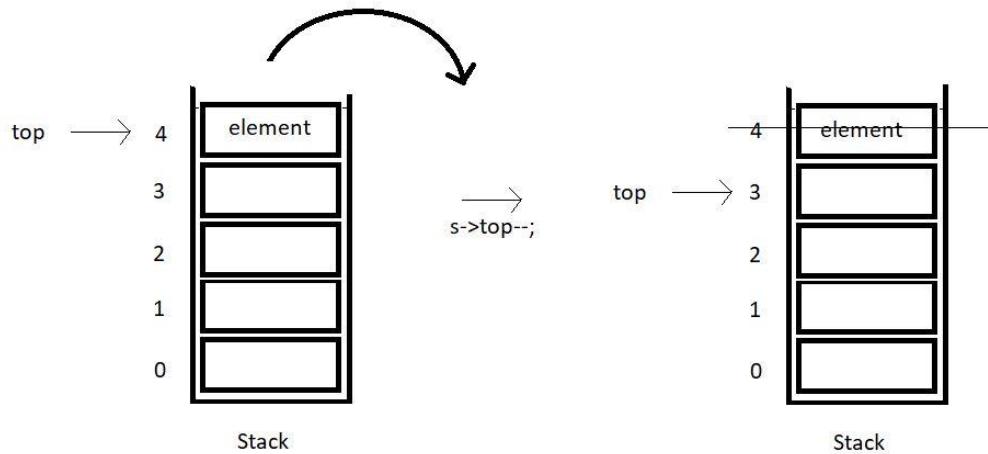
```
void push(struct stack* ptr, int val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the
stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
```

[Copy](#)

Code Snippet 2: Implementing the push operation.

7. Create another void function *pop*, and pass into it the same address of the stack using the pointer *sp*. This is the only parameter since the pop operation pops only the topmost element.
8. Don't forget to first check if our stack still has some elements left to pop elements. Use *isEmpty* function for that. If it returns 1, this is the case of stack underflow, otherwise, just decrease the top element of stack by 1, and we are done. The next time we push an element, we'll overwrite the present

element at that index. So, that's basically ignored and acts as if it got deleted.



```
int pop(struct stack* ptr){  
    if(isEmpty(ptr)){  
        printf("Stack Underflow! Cannot pop from the  
stack\n");  
        return -1;  
    }  
    else{  
        int val = ptr->arr[ptr->top];  
        ptr->top--;  
        return val;  
    }  
}
```

Copy

Code Snippet 3: Implementing the pop operation.

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>
```

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

```
int isEmpty(struct stack* ptr){
    if(ptr->top == -1){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
int isFull(struct stack* ptr){
    if(ptr->top == ptr->size - 1){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
void push(struct stack* ptr, int val){
```

```
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the
stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
```

```
int pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the
stack\n");
        return -1;
    }
    else{
        int val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

```
int main(){
    struct stack *sp = (struct stack *) malloc(sizeof(struct
stack));
    sp->size = 10;
    sp->top = -1;
    sp->arr = (int *) malloc(sp->size * sizeof(int));
    printf("Stack has been created successfully\n");
```

```
    return 0;
```

```
}
```

Copy

Code Snippet 4: Implementing the pop and the push operations.

Now let's check if everything is working properly. We'll first check if the *isFull* and the *isEmpty* functions work. Call these functions after declaring the stack *sp*.

```
printf("Before pushing, Full: %d\n", isFull(sp));
```

```
printf("Before pushing, Empty: %d\n", isEmpty(sp));
```

Copy

Code Snippet 5: Calling the *isEmpty* and the *isFull* functions

The output we received, was:

```
0
```

```
1
```

```
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

Copy

Figure 1: Output of the above program

So, since the stack is empty, it returned 1. Now, let's push 10 elements into this stack array using the *push* function. And then call the *isFull* and the *isEmpty* functions.

```
push(sp, 1);
```

```
push(sp, 23);
```

```
push(sp, 99);
```

```
    push(sp, 75);
    push(sp, 3);
    push(sp, 64);
    push(sp, 57);
    push(sp, 46);
    push(sp, 89);
    push(sp, 6); // ---> Pushed 10 values
    // push(sp, 46); // Stack Overflow since the size of the
stack is 10
    printf("After pushing, Full: %d\n", isFull(sp));
    printf("After pushing, Empty: %d\n", isEmpty(sp));
```

Copy

Code Snippet 6: Using the *push* function

The output we received, was:

```
1
0
```

```
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

Copy

Figure 2: Output of the above program

Since the stack is now full, it returned 1 from *isFull* function. This means our *push* function is working well. Now, let's pop some elements.

```
    printf("Popped %d from the stack\n", pop(sp)); // --> Last
in first out!
    printf("Popped %d from the stack\n", pop(sp)); // --> Last
in first out!
```

```
    printf("Popped %d from the stack\n", pop(sp)); // --> Last  
in first out!
```

Copy

Code Snippet 7: Using the *pop* function

The output we received was:

```
Popped 6 from the stack  
Popped 89 from the stack  
Popped 46 from the stack  
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

Copy

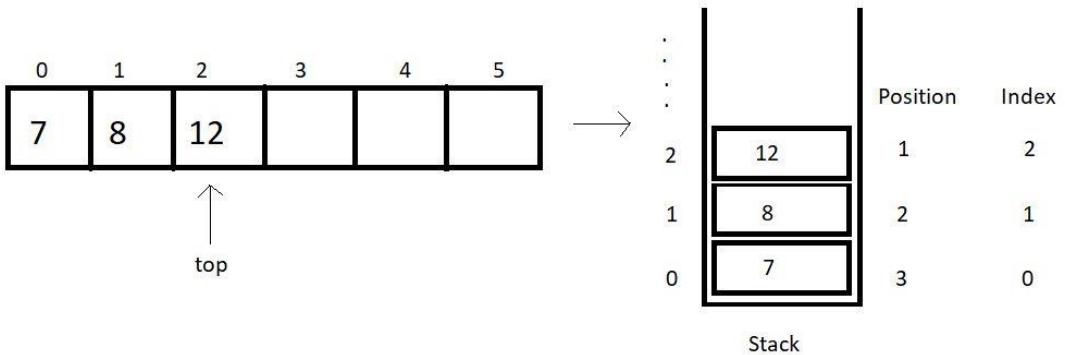
Figure 3: Output of the above program

Peek Operation in Stack Using Arrays (With C Code & Explanation)

Now that we've finished the push and pop operations, we'll move on to the peek operation in stacks. Peeking into something literally means to quickly see what's there at someplace. In a similar way, it refers to looking for the element at a specific index in a stack.

If you could remember, pushing an element into a stack needs you to first check if the stack is not full, and then insert the element at the incremented value of the top. And similarly, popping from a stack, needs you to first check if it is not empty, and then you just decrease the value of the top by 1.

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. I'll make you visualize this via a few 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.

Note: peek(1) returns 12 here.

Now, since we are done with all the basics of the peek operation, we can try writing its code as well. Here, we'll focus mainly on the peek operation, so you can just copy the codes from the last tutorial, where we learned writing *push* and *pop*, *isFull* and *isEmpty*.

I have attached the snippet below for your reference.

Understanding the code snippet 1:

1. I hope you have copied everything from the last tutorial. That'll save us some time. And this was important since we are focusing just on the peek operation.
2. Create an integer function *peek*, and pass the reference to the stack, and the position to peek in, as its parameters.
3. Inside the function, create an integer variable *arrayInd* which will store the index of the array to be returned. This is just $(top - position + 1)$.
4. Before we return anything, we'll check if the *arrayInd* is a valid index. If it's less than 0, it is invalid and we report an error. Otherwise, we just return the element at the index, $(top - position + 1)$.

```
int peek(struct stack* sp, int i){
    int arrayInd = sp->top - i + 1;
```

```
if(arrayInd < 0){  
    printf("Not a valid position for the stack\n");  
    return -1;  
}  
else{  
    return sp->arr[arrayInd];  
}  
}
```

Copy

Code Snippet 1: Writing the peek function

Here is the whole source code:

```
#include<stdio.h>  
#include<stdlib.h>  
  
struct stack{  
    int size ;  
    int top;  
    int * arr;  
};  
  
int isEmpty(struct stack* ptr){  
    if(ptr->top == -1){  
        return 1;  
    }  
    else{  
        return 0;  
    }
```

```
}
```

```
int isFull(struct stack* ptr){  
    if(ptr->top == ptr->size - 1){  
        return 1;  
    }  
    else{  
        return 0;  
    }  
}
```

```
void push(struct stack* ptr, int val){  
    if(isFull(ptr)){  
        printf("Stack Overflow! Cannot push %d to the  
stack\n", val);  
    }  
    else{  
        ptr->top++;  
        ptr->arr[ptr->top] = val;  
    }  
}
```

```
int pop(struct stack* ptr){  
    if(isEmpty(ptr)){  
        printf("Stack Underflow! Cannot pop from the  
stack\n");  
        return -1;  
    }  
    else{
```

```
    int val = ptr->arr[ptr->top];
    ptr->top--;
    return val;
}

}

int peek(struct stack* sp, int i){
    int arrayInd = sp->top - i + 1;
    if(arrayInd < 0){
        printf("Not a valid position for the stack\n");
        return -1;
    }
    else{
        return sp->arr[arrayInd];
    }
}

int main(){
    struct stack *sp = (struct stack *) malloc(sizeof(struct
stack));
    sp->size = 50;
    sp->top = -1;
    sp->arr = (int *) malloc(sp->size * sizeof(int));
    printf("Stack has been created successfully\n");

    printf("Before pushing, Full: %d\n", isFull(sp));
    printf("Before pushing, Empty: %d\n", isEmpty(sp));

    return 0;
}
```

[Copy](#)

Code Snippet 2: Implementing the peek function

This is how we peek into a stack array. We'll see how properly the functions work. First, we'll push a few elements into the empty stack we created.

```
push(sp, 1);
    push(sp, 23);
    push(sp, 99);
    push(sp, 75);
    push(sp, 3);
    push(sp, 64);
    push(sp, 57);
    push(sp, 46);
    push(sp, 89);
    push(sp, 6);
    push(sp, 5);
    push(sp, 75);
```

[Copy](#)

Code Snippet 3: Pushing a few elements in the stack

Now, we can peek into this stack array and print all the elements using a loop.

```
// Printing values from the stack
for (int j = 1; j <= sp->top + 1; j++)
{
    printf("The value at position %d is %d\n", j, peek(sp,
j));
}
```

[Copy](#)

Code Snippet 4: Calling the peek function

The output we received was:

```
The value at position 1 is 75
The value at position 2 is 5
The value at position 3 is 6
The value at position 4 is 89
The value at position 5 is 46
The value at position 6 is 57
The value at position 7 is 64
The value at position 8 is 3
The value at position 9 is 75
The value at position 10 is 99
The value at position 11 is 23
The value at position 12 is 1
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

[Copy](#)

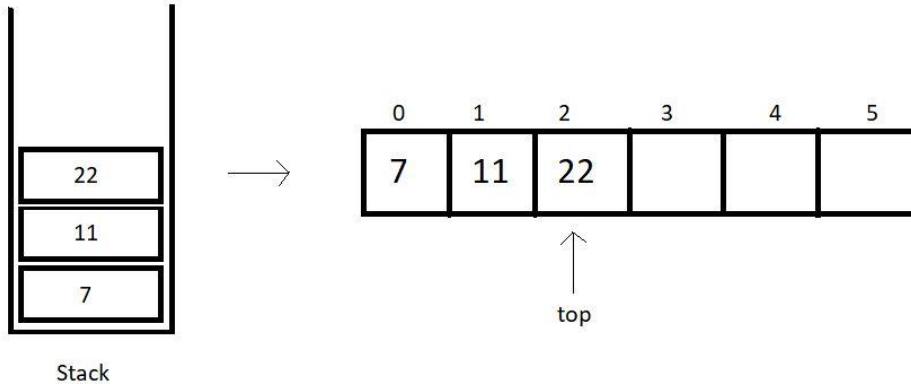
Figure 1: Output of the above program

stackTop, stackBottom & Time Complexity of Operations in Stack Using Arrays

In the last tutorial, we talked about the peek operation and implemented it in C using arrays. Today, we will explore some new stack operations.

Take a deep breath of relief since the things we are discussing today are the easiest of all. We will first start by learning about two operations we have in stacks, stackTop and stackBottom.

Let's consider a stack array for the understanding purpose.



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.

Time complexities of other 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 you 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 you 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.

Implementation:

I would suggest you all implement them on your own before moving ahead. I have attached the snippet below, for your referral.

Understanding the snippet below:

1. First of all, copy everything we have covered up to this point in your IDEs. I don't want to repeat them all and make this lengthy.
2. I suppose you have all the functions and declarations done.
3. Create an integer function *stackTop*, and pass the reference to the stack you created as a parameter. Just return the element at the index top of the array. And that's it.

```
int stackTop(struct stack* sp){
    return sp->arr[sp->top];
}
```

Copy

Code Snippet 1: Implementing *stackTop*

4. Create an integer function *stackBottom*, and pass the reference to the stack you created as a parameter. And then return the element at the index 0 of the array.

```
int stackBottom(struct stack* sp){
```

```
    return sp->arr[0];  
}
```

Copy

Code Snippet 2: Implementing *stackBottom*

Here is the whole source code:

```
#include<stdio.h>  
#include<stdlib.h>  
  
struct stack{  
    int size ;  
    int top;  
    int * arr;  
};  
  
int isEmpty(struct stack* ptr){  
    if(ptr->top == -1){  
        return 1;  
    }  
    else{  
        return 0;  
    }  
}  
  
int isFull(struct stack* ptr){  
    if(ptr->top == ptr->size - 1){  
        return 1;  
    }  
}
```

```
    else{
        return 0;
    }
}

void push(struct stack* ptr, int val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the
stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

int pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the
stack\n");
        return -1;
    }
    else{
        int val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

```
int peek(struct stack* sp, int i){  
    int arrayInd = sp->top - i + 1;  
    if(arrayInd < 0){  
        printf("Not a valid position for the stack\n");  
        return -1;  
    }  
    else{  
        return sp->arr[arrayInd];  
    }  
}
```

```
int stackTop(struct stack* sp){  
    return sp->arr[sp->top];  
}
```

```
int stackBottom(struct stack* sp){  
    return sp->arr[0];  
}
```

```
int main(){  
    struct stack *sp = (struct stack *) malloc(sizeof(struct  
stack));  
    sp->size = 50;  
    sp->top = -1;  
    sp->arr = (int *) malloc(sp->size * sizeof(int));  
    printf("Stack has been created successfully\n");
```

```
    printf("Before pushing, Full: %d\n", isFull(sp));  
    printf("Before pushing, Empty: %d\n", isEmpty(sp));
```

```
    push(sp, 1);
    push(sp, 23);
    push(sp, 99);
    push(sp, 75);
    push(sp, 3);
    push(sp, 64);
    push(sp, 57);
    push(sp, 46);
    push(sp, 89);
    push(sp, 6);
    push(sp, 5);
    push(sp, 75);
```

```
// // Printing values from the stack
// for (int j = 1; j <= sp->top + 1; j++)
// {
//     printf("The value at position %d is %d\n", j,
peek(sp, j));
// }

return 0;
}
```

Copy

Code Snippet 3: Implementing *stackTop* & *stackBottom*

Now, since we have done pushing elements into the stack, we can call our functions, *stackTop* and *stackBottom*.

```
printf("The top most value of this stack is %d\n",
stackTop(sp));
```

```
    printf("The bottom most value of this stack is %d\n",
stackBottom(sp));
```

Copy

Code Snippet 4: Calling functions *stackTop* & *stackBottom*

And the output we received was:

```
The top most value of this stack is 75
The bottom most value of this stack is 1
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

Copy

Figure 1: Output of the above program

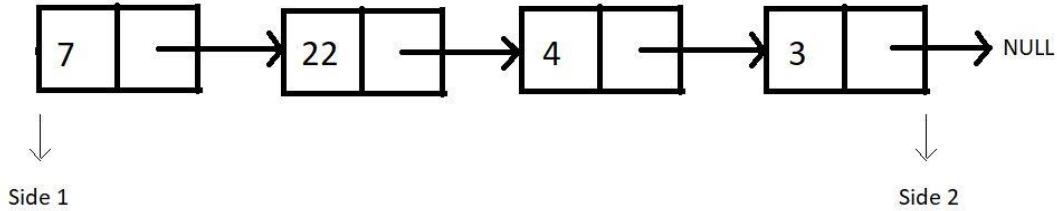
How to Implement Stack Using Linked List?

Earlier before, whenever we discussed stacks, we used arrays. We saw how good an array is while implementing stacks using them. We saw it follows constant time complexity for each of the operations we discussed. Today, we'll begin implementing stacks using a different data structure, linked lists.

Linked-lists is surely not a new term for you all. We have come here only after discussing all the basics. So, if you haven't come across the linked lists, you must have skipped them. I highly recommend you all to go through the videos discussing them in the playlist. Assuming you are done, we'll proceed.

Implementing stacks using linked lists:

We can now consider a singly linked list. Follow the illustration below.

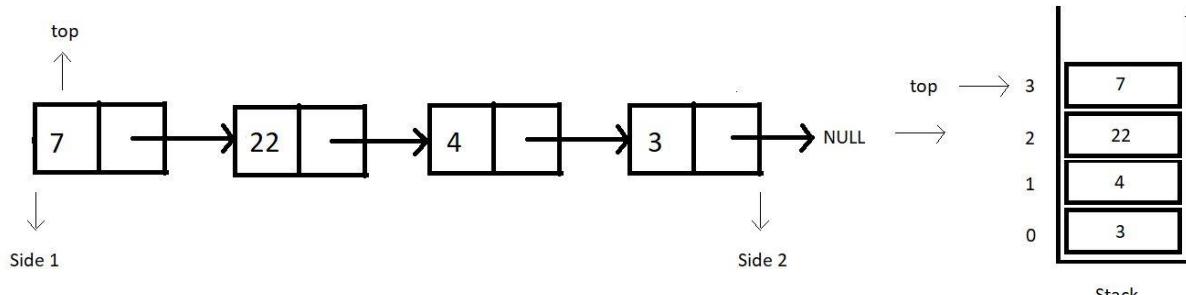


Consider this linked list functioning as a stack. And as you know, we have two sides of a linked list, one the head, and the other pointing to NULL. Which side do you feel should we consider as the top of the stack, where we push and pop from? After following me all the way through here, you would say the 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:



Let's revise how we used to define a struct Node in linked lists. We had a struct, and two structure members, data and a struct Node pointer to store the address of the next node.

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

Copy

Code Snippet 1: Structure of a Node in a Linked List

When is our stack empty or full?

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. Whereas stacks become empty when there is no node in the linked list, hence when the top equals to NULL.

1. Condition for stack full: When heap memory is exhausted
2. Condition for stack empty: top == NULL

One change I would like to implement before we proceed; 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, you can always set a custom size for it.

Implementing all the Stack Operations using Linked List (With Code in C)

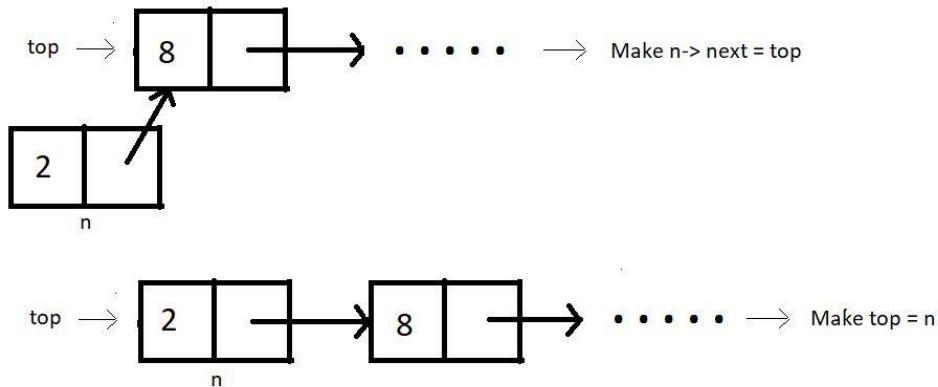
In the last tutorial, we started learning about implementing stacks using linked lists. We saw the benefits of using the head side of the linked list as the stack top. We figured out the conditions for the stack linked lists to be empty or full. Today, we'll discuss more of these operations, and write their codes in C.

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

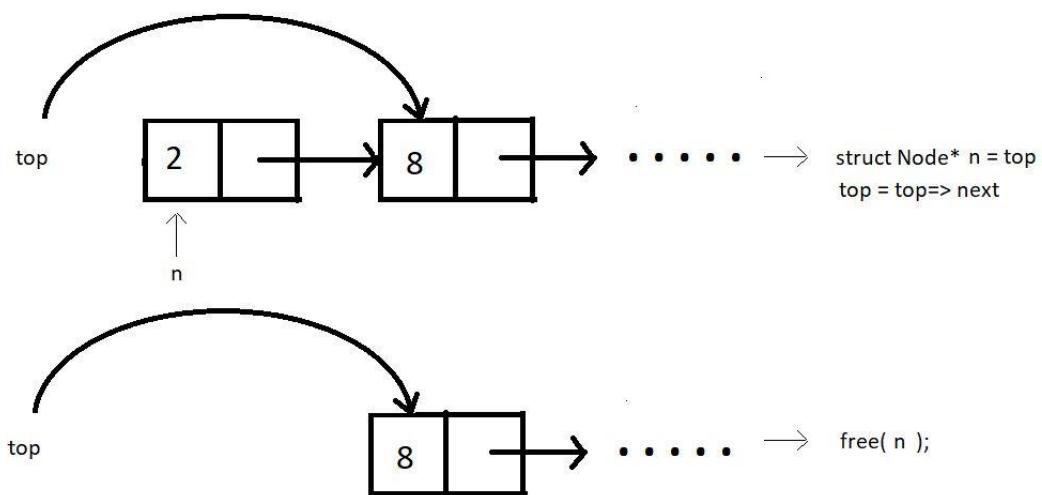
1. isEmpty : It just checks if our top element is NULL.

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.



We'll limit ourselves to these four operations for today. We'll now move to our editors to code them. We have already covered the tough parts of today's tutorial; these are the easy ones remaining. I have attached the code snippet below, refer to them while you code:

Understanding the code snippet below:

1. Create the structure for nodes. We'll use struct in C, name its *Node*, and make two members of this struct; an integer variable to store the data, and a struct Node pointer to store the address of the next element.

2. First of all, we'll create the *isEmpty* and the *isFull* functions.

3. *isEmpty()*:

- Create an integer function *isEmpty*, and pass the pointer to the top node as the parameter. If this top node equals NULL, return 1, else 0.

```
int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
}
```

Copy

Code Snippet 1: Implementing isEmpty function

4. isFull():

- Create an integer function *isFull*, and pass the pointer to the top node as the parameter.
- Create a new struct Node* pointer *p*, and assign it a new memory location in the heap. If this newly created node *p* is NULL, return 1, else 0.

```
int isFull(struct Node* top){  
    struct Node* p = (struct Node*)malloc(sizeof(struct  
Node));  
    if(p==NULL){  
        return 1;  
    }  
    else{  
        return 0;  
    }  
}
```

Copy

Code Snippet 2: Implementing isFull function

5. Push():

- Create a struct Node* function *push* which will return the pointer to the new top node.
- We'll pass the current top pointer and the data to push in the stack, in the function.

- Check if the stack is already not full, if full, return the condition stack overflow.
- Create a new struct Node* pointer *n*, and assign it a new memory location in the heap.
- Assign top to the next member of the n structure using *n->next = top*, and the given data to its data member.
- Return this pointer *n*, since this is our new top node.

```
struct Node* push(struct Node* top, int x){
    if(isFull(top)){
        printf("Stack Overflow\n");
    }
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}
```

[Copy](#)

Code Snippet 3: Implementing Push function

6. Pop() :

- Create an integer function *pop* which will return the element we remove from the top.
- We'll pass the reference of the current top pointer in the function. We are passing the reference this time, because we are not returning the updated top from the function.
- Check if the stack is already not empty, if empty, return the condition stack underflow.

- Create a new struct Node* pointer *n*, and make it point to the current top. Store the data of this node in an integer variable *x*.
- Assign top to the next member of the list, by top = top->next, because this is going to be our new top.
- Free the pointer *n*. And return *x*.

```
int pop(struct Node** top){
    if(isEmpty(*top)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = *top;
        *top = (*top)->next;
        int x = n->data;
        free(n);
        return x;
    }
}
```

[Copy](#)

Code Snippet 4: Implementing pop function

7. Now, since we would always need a traversal function to see if our operations are functioning all well, we'll just bring our codes from the linked list tutorial, named *linkedListTraversal*.

```
void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
```

```
    }
}
```

Copy

Code Snippet 5: LinkedListTraversal function

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

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

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
}
```

```
    else{
        return 0;
    }
}
```

```
int isFull(struct Node* top){
    struct Node* p = (struct Node*)malloc(sizeof(struct
Node));
    if(p==NULL){
        return 1;
    }
    else{
        return 0;
    }
}
```

```
struct Node* push(struct Node* top, int x){
    if(isFull(top)){
        printf("Stack Overflow\n");
    }
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct
Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}
```

```

int pop(struct Node** top){
    if(isEmpty(*top)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = *top;
        *top = (*top)->next;
        int x = n->data;
        free(n);
        return x;
    }
}

int main(){
    struct Node* top = NULL;
    return 0;
}

```

Copy

Code Snippet 6: Implementing a stack and its operations using linked list

We have just created a stack using a linked list. We have assigned NULL to the top node. Let's first push some elements and see if the changes reflect in the stack. We'll use traversal for that.

```

top = push(top, 78);
top = push(top, 7);
top = push(top, 8);

```

```
    linkedListTraversal(top);
```

Copy

Code Snippet 7: Pushing elements in a stack.

The output we received was:

```
Element: 8
```

```
Element: 7
```

```
Element: 78
```

```
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

Copy

Figure 1: Output of the above program

So, the push function worked all good. Let's pop one element out from the stack. And then again traverse through it.

```
int element = pop(&top);  
printf("Popped element is %d\n", element);  
linkedListTraversal(top);
```

Copy

Code Snippet 8: Popping elements from a stack.

The output we received then was:

```
Popped element is 8
```

```
Element: 7
```

```
Element: 78
```

```
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

[Copy](#)

Figure 2: Output of the above program

You must have observed we used the pointer to a pointer while popping elements from the stack. We referenced and unreferenced twice. So, to avoid all these complexities, I still have a better way to implement that thing. We can declare the *top* pointer globally. Earlier we used to declare it under main. Declaring it globally gives its access to all our functions without passing them as a parameter.

Refer to the second implementation of stacks below. They are more or less the same, just subtle changes. Follow them carefully. You are wise enough to understand them on your own.

```
#include<stdio.h>
#include<stdlib.h>

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

struct Node* top = NULL;

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
```

```
int isEmpty(struct Node* top){  
    if (top==NULL){  
        return 1;  
    }  
    else{  
        return 0;  
    }  
}  
  
int isFull(struct Node* top){  
    struct Node* p = (struct Node*)malloc(sizeof(struct  
Node));  
    if(p==NULL){  
        return 1;  
    }  
    else{  
        return 0;  
    }  
}  
  
struct Node* push(struct Node* top, int x){  
    if(isFull(top)){  
        printf("Stack Overflow\n");  
    }  
    else{  
        struct Node* n = (struct Node*) malloc(sizeof(struct  
Node));  
        n->data = x;
```

```
n->next = top;
top = n;
return top;
}

}

int pop(struct Node* tp){
if(isEmpty(tp)){
    printf("Stack Underflow\n");
}
else{
    struct Node* n = tp;
    top = (tp)->next;
    int x = n->data;
    free(n);
    return x;
}
}

int main(){
    top = push(top, 78);
    top = push(top, 7);
    top = push(top, 8);

    // linkedListTraversal(top);

    int element = pop(top);
    printf("Popped element is %d\n", element);
    linkedListTraversal(top);
}
```

```
    return 0;  
}
```

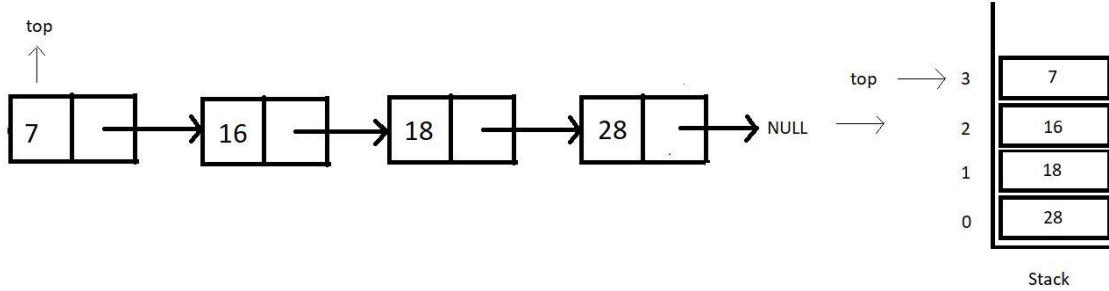
Copy

Code Snippet 9: Implementing a stack and its operations using linked list

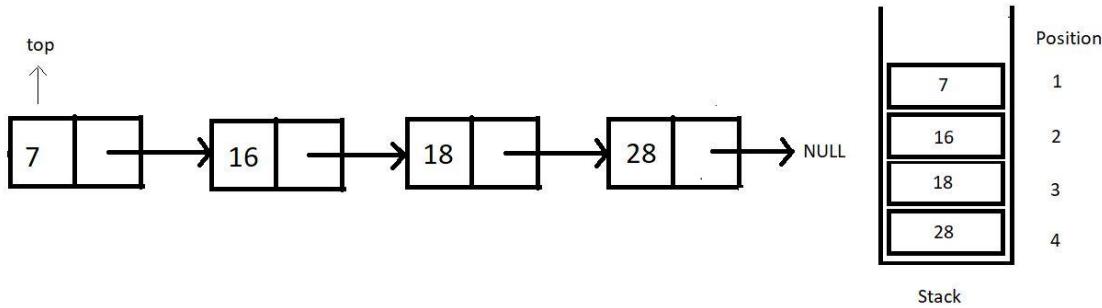
peek(), stackTop() and Other Operations on Stack Using Linked List (with C Code)

In the last tutorial, we learned to implement stacks using linked lists. We saw how efficiently we can push and pop elements in a stack-linked list. We saw a few other operations, *isEmpty*, *isFull*, *traversal*. Today, we will cover the remaining operations. They are: peek, stackTop, etc.

Similar to what we did last time, we will first understand the algorithm behind the operations, followed by the coding section. Let's see them individually, but before that, let's have an example illustration of the stack we'll go into within today's tutorial.



1. 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.

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

3. stackBottom:

I will leave the last operation, *stackBottom*, for your homework. Try implementing this on your own, and let me know if you could. You should be able to code this since we have covered the concepts already in the stack arrays.

So, these were the only operations we had in mind to discuss with you all. You will come across several variations of these. Nevertheless, you are intelligent enough to be able to change your codes if necessary. We'll now move to our editors to code the operations we discussed today. I have attached the code snippet below. Refer to them while you code:

Understanding the code snippet below:

1. Copy everything we did in the last tutorial. This will save us some time. It will also prevent repetitions in the course. Our main focus for today is to discuss these three operations. So, creating the stack and other operations can be ignored since they have already been covered.

2. We'll start with the *peek* function.

3. *peek()*:

- Create an integer function *peek*, and pass the position you want to peek in as a parameter.
- Since we have made the stack pointer global, we should not use that pointer to traverse; otherwise, we will lose the pointer to the top node. Rather create a new struct Node pointer *ptr* and give it the value of *top*.
- Run a loop from 0 to *pos*-1, since we are already at the first position.
- If our pointer reaches NULL at some point, we must have reached the last node, and the position asked was beyond the available positions, hence breaking the loop.
- If the current pointer found the position and it is not equal to NULL, return the data at that node, else -1.

```
int peek(int pos){  
    struct Node* ptr = top;  
    for (int i = 0; (i < pos-1 && ptr!=NULL); i++)  
    {  
        ptr = ptr->next;  
    }  
    if(ptr!=NULL){  
        return ptr->data;  
    }  
    else{  
        return -1;  
    }  
}
```

Copy

Code Snippet 1: Implementing peek function

4. stackTop():

- Create an integer function *stackTop*, and we are no longer passing any parameter since the top pointer is declared globally.
- Simply return the data member of the struct Node pointer *top*, and that's it.

```
int stackTop(){  
    return top->data;  
}
```

Copy

Code Snippet 2: Implementing stackTop function

Here is the whole source code:

```
#include<stdio.h>  
#include<stdlib.h>
```

```

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

struct Node* top = NULL;

void linkedListTraversal(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}

int isEmpty(struct Node* top){
    if (top==NULL){
        return 1;
    }
    else{
        return 0;
    }
}

int isFull(struct Node* top){
    struct Node* p = (struct Node*)malloc(sizeof(struct Node));
    if(p==NULL){
        return 1;
    }
    else{

```

```
        return 0;
    }
}

struct Node* push(struct Node* top, int x){
    if(isFull(top)){
        printf("Stack Overflow\n");
    }
    else{
        struct Node* n = (struct Node*) malloc(sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
}

int pop(struct Node* tp){
    if(isEmpty(tp)){
        printf("Stack Underflow\n");
    }
    else{
        struct Node* n = tp;
        top = (tp)->next;
        int x = n->data;
        free(n);
        return x;
    }
}

int peek(int pos){
```

```

    struct Node* ptr = top;
    for (int i = 0; (i < pos-1 && ptr!=NULL); i++)
    {
        ptr = ptr->next;
    }
    if(ptr!=NULL){
        return ptr->data;
    }
    else{
        return -1;
    }
}

int main(){
    top = push(top, 28);
    top = push(top, 18);
    top = push(top, 15);
    top = push(top, 7);

    linkedListTraversal(top);
    for (int i = 1; i <= 4; i++)
    {
        printf("Value at position %d is : %d\n", i, peek(i));
    }
    return 0;
}

```

[Copy](#)

Code Snippet 3: Using peek function

Let's now push some elements into the stack and see if the operations work all good.

```
top = push(top, 28);
```

```
top = push(top, 18);
top = push(top, 15);
top = push(top, 7);
```

Copy

Code Snippet 4: Using push function to put some elements inside the stack

Since we have pushed the elements, we can call our peek function in a loop, printing the whole array.

```
for (int i = 1; i <= 4; i++)
{
    printf("Value at position %d is : %d\n", i, peek(i));
}
```

Copy

Code Snippet 5: Using peek function to print the whole stack

The output we received was:

```
Value at position 1 is : 7
Value at position 2 is : 15
Value at position 3 is : 18
Value at position 4 is : 28
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

Parenthesis Matching Problem Using Stack Data Structure (Applications of Stack)

I was very excited to bring this topic to your attention. Parenthesis matching is one of the basic applications of the stack we learned about in our last ten lectures. This will be thought-provoking to you all as well. Since the dawn of programming, parenthesis matching has been a favorite topic. It is a must learn. So, today, we'll

start learning about parenthesis matching and how it gets implemented using stacks.

Parenthesis matching has always been threatening to beginners. But realizing its implementation using stacks makes it very intuitive and easy to deal with.

What is parenthesis matching?

If you remember learning mathematics in school, we had BODMAS there, which required you 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.

Below given illustrations would surely make it clear for you.

$$\left(\left(3 * 2 \right) - 1 \left(8 - 2 \right) \right)$$

Balanced Parentheses

$$1 - 3) * 4 (8$$

↑ ↑

No coresponding No corresponding
opening bracket closing bracket

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 to you as a character array.

0	1	2	3	4	5	6	7	8	9
3	*	2	-	(8	+	1)	\0

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

3. Conditions for unbalanced parentheses:

- When you find a closing parenthesis and try achieving the pop operation in 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 you 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.

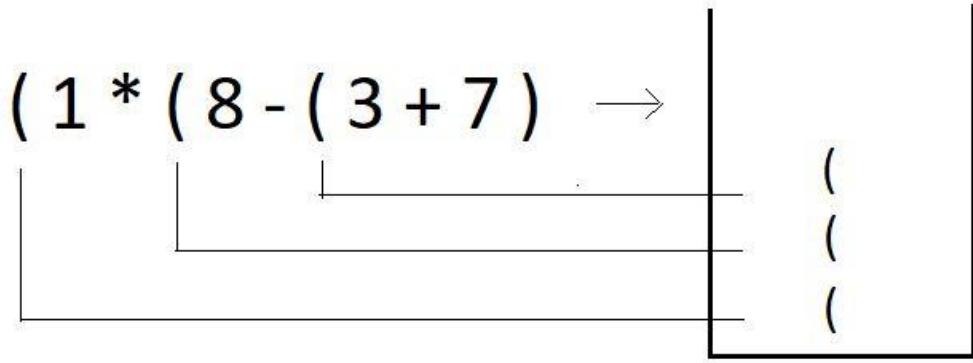
4. Note: Counting and matching the opening and closing brackets numbers is not enough to conclude if the parentheses are balanced. For eg: **1+3)*6(6+2**.

Example:

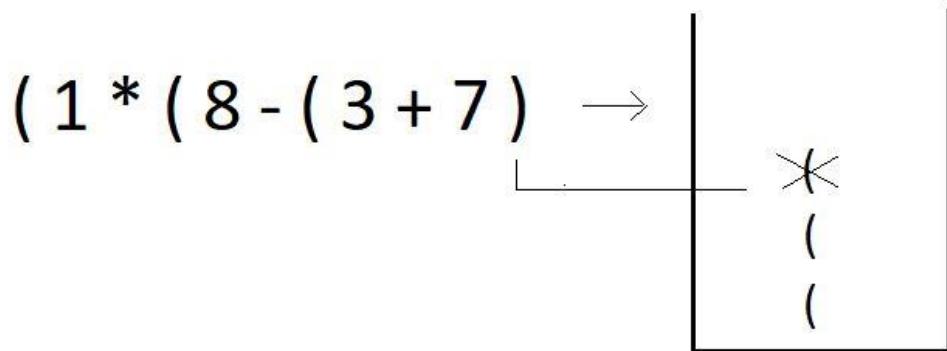
0	1	2	3	4	5	6	7	8	9	10
(1	*	(8	-	(3	+	7)

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 you 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**.

I have one task for you as well. Try checking if these expressions are balanced or not. And also, tell the number of times you had to push or pop in the stack. Also, comment on the time complexity of this algorithm. Answer the best and the worst runtime complexity for an expression of size n.

1. $7 - (8(3 * 4) + 11 + 12) - 8$

I would just recommend everyone to try coding the algorithm for parentheses matching on their own, and do tell me if you could. We'll see this segment in the next tutorial anyways. Do let me know if you are enjoying the course. Bigger things are waiting, just stay!

Parenthesis Checking Using Stack in C Language

In the last tutorial, we tried making parentheses matching intuitive and more understandable using stacks. We followed one simple algorithm to accomplish that. The algorithm states:

- Everytime you come across an opening parenthesis, push it in the stack.
- Everytime you come across a closing parenthesis, pop one opening parenthesis out from the stack.
- We call this match of parentheses unbalanced when we encounter either of the two of these troubles:
 1. There is no more opening bracket inside the stack to pop, and you come across a closing bracket.
 2. The stack size is not zero, or there are still more than zero opening brackets present in the stack after you come across EOE(end-of-expression).

So, that was a quick revision of the things we learned in the previous tutorial. We did enough examples in the previous tutorial; you can check them as well. In today's lesson, we will program the algorithm in C.

Understanding the code snippet below:

1. Start by creating an integer function *paranthesisMatch*, and pass the reference to a character array(expression) *exp* in the function as a parameter. This function will return 1 if the parentheses are balanced and zero otherwise.
2. Inside that function, create a stack pointer *sp*. And initialize the size member to some big number, let it be 100. Initialize the top to -1, and assign the array pointer a memory location in the heap. You have the freedom to choose any data structure you want to implement this stack. We have learned stacks using both arrays and linked lists very efficiently.

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

Copy

Code Snippet 1: Creating and Initialising stack array.

3. So, it would be better if you just copy everything of stack implementation because it will more or less remain the same for that part. I'll use the array one.

4. Change the datatype of the array from integer to char. Accordingly, change everything from integer to char. And *arr* to *exp*.

5. Run a loop starting from the beginning of the expression till it reaches EOE.

6. If the current character of the expression is an opening parenthesis, '(', push it into the stack using the push operation.

7. Else if the current character is a closing parenthesis ')', see if the stack is not empty, using *isEmpty*, and if it is, return 0 there itself, else pop the topmost character using *pop* operation.

8. In the end, if the stack becomes empty, return 1, else 0.

9. In the main, define a random character array expression and just passing this expression to *parenthesisMatch* would do our job.

Code for parentheses matching:

```
int parenthesisMatch(char * exp){  
    // Create and initialize the stack  
    struct stack* sp;  
    sp->size = 100;  
    sp->top = -1;  
    sp->arr = (char *)malloc(sp->size * sizeof(char));  
  
    for (int i = 0; exp[i]!='\0'; i++)  
    {  
        if(exp[i]=='('){  
            push(sp, '(');  
        }  
        else if(exp[i]==')'){  
            if(isEmpty(sp)){  
                return 0;  
            }  
            pop(sp);  
        }  
    }  
}
```

```
if(isEmpty(sp)){
    return 1;
}
else{
    return 0;
}
```

Copy

Code Snippet 2: Creating the parenthesisMatch function

Here is the whole source code:

```
#include <stdio.h>
#include <stdlib.h>

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

int isEmpty(struct stack *ptr)
{
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

```
}
```

```
int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

```
void push(struct stack* ptr, char val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}
```

```
char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
```

```
    ptr->top--;
    return val;
}
}
```

```
int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));
```

```
for (int i = 0; exp[i]!='\0'; i++)
{
    if(exp[i]=='('){
        push(sp, '(');
    }
    else if(exp[i]==')'){
        if(isEmpty(sp)){
            return 0;
        }
        pop(sp);
    }
}
```

```
if(isEmpty(sp)){
    return 1;
}
else{
    return 0;
```

```
    }

}

int main()
{
    char * exp = "((8)(*--$$9))";
    // Check if stack is empty
    if(parenthesisMatch(exp)){
        printf("The parenthesis is matching");
    }
    else{
        printf("The parenthesis is not matching");
    }
    return 0;
}
```

Copy

Code Snippet 3: A program to check for balanced parentheses.

Let's now just see if the functions work properly. We will give it some expressions of our choice.

```
char * exp = "((8)(*--$$9))";
// Check if stack is empty
if(parenthesisMatch(exp)){
    printf("The parenthesis is matching");
}
else{
    printf("The parenthesis is not matching");
}
```

Copy

Code Snippet 4: Calling the parenthesisMatch function

The output we received was:

```
The parenthesis is matching
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

Let's see for some another expression:

```
char * exp = "8)*(9);  
// Check if stack is empty  
if(parenthesisMatch(exp)){  
    printf("The parenthesis is matching");  
}  
else{  
    printf("The parenthesis is not matching");  
}
```

Copy

Code Snippet 5: Calling the parenthesisMatch function for another expression

The output we received was:

```
The parenthesis is not matching
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 2: Output of the above program

Multiple Parenthesis Matching Using Stack with C Code

In the last tutorial, we saw the implementation of parentheses matching using stacks in C. One thing you must have observed is that we used only one type of parenthesis throughout the tutorial. But in mathematics, we have expressions consisting of all three types of parenthesis. Today 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.

If you remember, parenthesis matching has nothing to do with the validity of the expression. It just tells whether an expression has all the parentheses balanced or

not. A balanced parentheses expression has a corresponding closing parenthesis to all of its opening parentheses. When we talk about matching multi parenthesis, our focus is mainly on the three types of an opening parenthesis, [{ (and their corresponding closing parentheses,) }]. So, basically, this tutorial is just an extension of what we learned in the previous two.

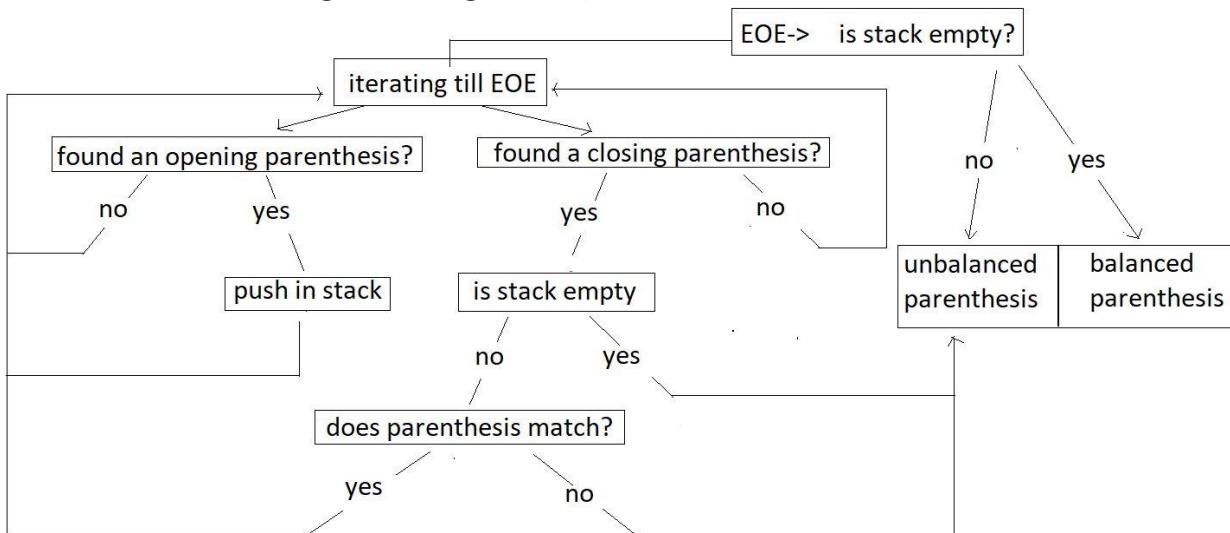
Modifying what we did earlier to make it work for multi-matching needs very little attention. Just follow these steps:

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:

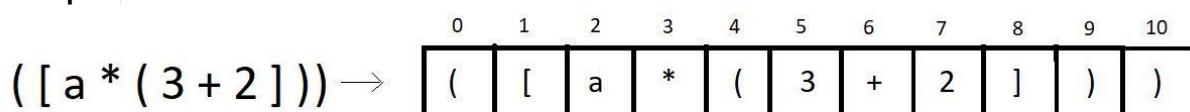
- Before we pop, this size of the stack must not be zero.
- The topmost parenthesis of the stack must match the type of closing parenthesis we encountered.

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.

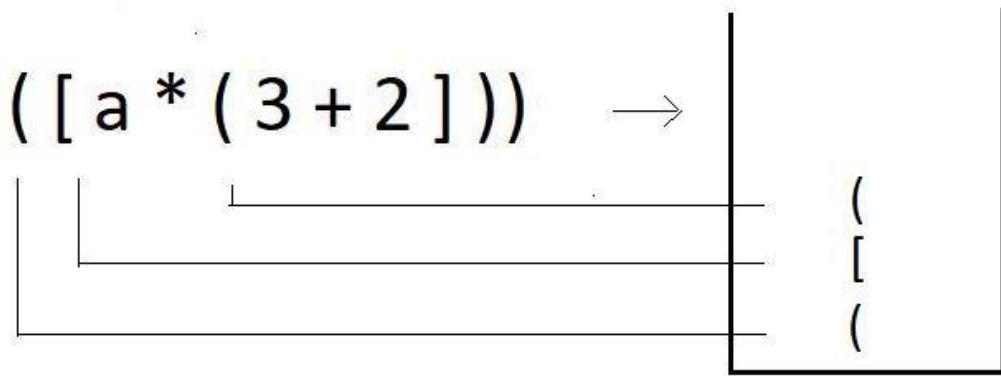


Example:

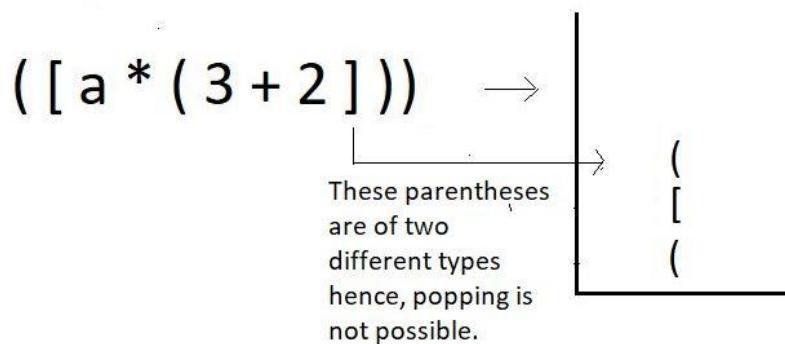


We'll try checking if the above expression has balanced multi-parentheses or not.

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 you encounter a closing bracket of any type in the expression, try checking if the kind of closing bracket you 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**.

The modified function should follow this algorithm. Let's now move to our editors.

Understanding the code snippet below:

1. Since in this tutorial, our main focus is to modify the code for matching parenthesis of a single type to matching multi parentheses., we'll copy the whole

thing from our last tutorial, from creating the function *parenthesisMatch* to the stack inside.

2. It is important to copy everything because a lot of things will remain the same. We make zero changes in the declaration of the stack and its members.

3. Run a loop starting from the beginning of the expression till it reaches EOE.

4. If the current character of the expression is an opening parenthesis, be it of any type, '(', '[', '{', push it into the stack using the push operation.

5. Else if the current character is a closing parenthesis of any type ')', ']', '}', see if the stack is not empty, using isEmpty, and if it is, return 0 there itself, else pop the topmost character using pop operation and store it in another character variable named *popped_ch* declared globally.

6. Create an integer function, *match* which will get the characters, *popped_ch*, and the current character of the expression as two parameters. Inside this function, check if these two characters are the same. If they are the same, return 1, else 0.

```
int match(char a, char b){  
    if(a=='{' && b=='}'){  
        return 1;  
    }  
    if(a=='(' && b==')'){  
        return 1;  
    }  
    if(a=='[' && b==']'){  
        return 1;  
    }  
    return 0;  
}
```

Copy

Code Snippet 1: Creating the match function

6. If the *match* function returns 1, our pop operation is successful, and we can continue checking further characters; else, if it returns 0, end the program here itself and return 0 to the main.

7. And if things went well throughout, and in the end, if the stack becomes empty, return 1, else 0.

Code for multi parentheses matching:

```
int parenthesisMatch(char * exp){
```

```

// Create and initialize the stack

struct stack* sp;
sp->size = 100;
sp->top = -1;
sp->arr = (char *)malloc(sp->size * sizeof(char));
char popped_ch;

for (int i = 0; exp[i]!='\0'; i++)
{
    if(exp[i]=='(' || exp[i]=='{' || exp[i]=='['){
        push(sp, exp[i]);
    }
    else if(exp[i]==')' || exp[i]=='}' || exp[i]==']'){
        if(isEmpty(sp)){
            return 0;
        }
        popped_ch = pop(sp);
        if(!match(popped_ch, exp[i])){
            return 0;
        }
    }
}

if(isEmpty(sp)){
    return 1;
}
else{
    return 0;
}
}

```

Copy

Code Snippet 2: Creating the modified parenthesisMatch function

Here is the whole source code:

```
#include <stdio.h>
#include <stdlib.h>

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

int isEmpty(struct stack *ptr)
{
    if (ptr->top == -1)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}

int isFull(struct stack *ptr)
{
    if (ptr->top == ptr->size - 1)
    {
        return 1;
    }
    else
```

```
{  
    return 0;  
}  
}  
  
void push(struct stack* ptr, char val){  
    if(isFull(ptr)){  
        printf("Stack Overflow! Cannot push %d to the stack\n", val);  
    }  
    else{  
        ptr->top++;  
        ptr->arr[ptr->top] = val;  
    }  
}  
  
char pop(struct stack* ptr){  
    if(isEmpty(ptr)){  
        printf("Stack Underflow! Cannot pop from the stack\n");  
        return -1;  
    }  
    else{  
        char val = ptr->arr[ptr->top];  
        ptr->top--;  
        return val;  
    }  
}  
  
char stackTop(struct stack* sp){  
    return sp->arr[sp->top];  
}
```

```

int match(char a, char b){
    if(a=='{' && b=='}'){
        return 1;
    }
    if(a=='(' && b==')'){
        return 1;
    }
    if(a=='[' && b==']'){
        return 1;
    }
    return 0;
}

int parenthesisMatch(char * exp){
    // Create and initialize the stack
    struct stack* sp;
    sp->size = 100;
    sp->top = -1;
    sp->arr = (char *)malloc(sp->size * sizeof(char));
    char popped_ch;

    for (int i = 0; exp[i]!='\0'; i++)
    {
        if(exp[i]=='(' || exp[i]=='{' || exp[i]=='['){
            push(sp, exp[i]);
        }
        else if(exp[i]==')' || exp[i]=='}' || exp[i]==']'){
            if(isEmpty(sp)){
                return 0;
            }
            popped_ch = pop(sp);
        }
    }
}

```

```

        if(!match(popped_ch, exp[i])){
            return 0;
        }
    }
}

if(isEmpty(sp)){
    return 1;
}
else{
    return 0;
}

}

int main()
{
    char * exp = "[4-6][(8){(9-8)})";

    if(parenthesisMatch(exp)){
        printf("The parenthesis is balanced");
    }
    else{
        printf("The parenthesis is not balanced");
    }
    return 0;
}

```

[Copy](#)

Code Snippet 3: A program to check for balanced multi-parentheses.

Let's try the functions now and see if they work. We will give it some random expressions of our choice.

```
char * exp = "((8){(9-8)})";
// Check if stack is empty
if(parenthesisMatch(exp)){
    printf("The parenthesis is matching");
}
else{
    printf("The parenthesis is not matching");
```

Copy

Code Snippet 4: Calling the parenthesisMatch function

The output we received was:

```
The parenthesis is matching
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

Let's see for some another expression:

```
char * exp = "[[4-6][(8){(9-8)}]]";
if(parenthesisMatch(exp)){
    printf("The parenthesis is balanced");
}
else{
    printf("The parenthesis is not balanced");
}
```

Copy

Code Snippet 5: Calling the parenthesisMatch function for another expression

The output we received was:

```
The parenthesis is not matching
```

PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

Copy

Figure 2: Output of the above program

Infix, Prefix and Postfix Expressions

We have finished learning matching parentheses in the last tutorial. It is always great to see the applications of what you learn, and parentheses matching was one such application of stacks. Today we'll start another one, called infix, prefix, and postfix expressions.

What are these?

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:

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 eg: $2 + 3$, $a * b$, $6 / 3$ etc.

< operand1 >< operator >< operand2 >

Prefix:

This method might seem new to you, but we have vocally used them a lot as well. 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.

< operator >< operand1 >< operand2 >

Postfix:

This is the method that might as well seem new to you, but we have used even this in our communication. 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.

< operand1 >< operand2 >< operator >

To understand the interchangeability of these terms, please refer to the table below.

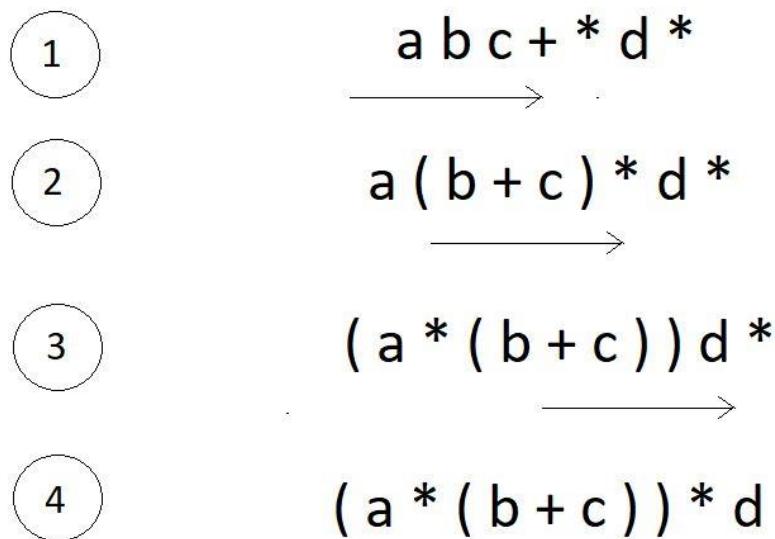
	Infix	Prefix	Postfix
1.	$a * b$	$* a b$	$a b *$
2.	$a - b$	$- a b$	$a b -$

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 you do when you get to evaluate a 1000 character long-expression, or even longer one? You 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.

Consider the expression $a * (b + c) * d$; since computers go left to right while evaluating an expression, we'll convert this infix expression to its postfix form. Its postfix form is, $a\ b\ c\ +\ * d\ *$. You must be wondering how we got here. Refer to the illustration below.



We have successfully reached what we wanted the machine to do. Now the kick is in converting infixes to postfixes and prefixes.

Converting infix to prefix:

Consider the expression, $x - y * z$.

1. Parenthesize 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. $(x - [* y z]) \rightarrow [- x * y z]$.

4. And we are done.

Converting infix to postfix:

Consider the same expression, $x - y * z$.

5. Parenthesize the expression as we did previously. Our expression now becomes $(x - (y * z))$.

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

7. Similarly, keep converting one by one, from the innermost to the outer parentheses. $(x - [yz^*]) \rightarrow [xyz^*-]$.

8. And we are done.

Similarly the expression $p - q - r / a$, follows the following conversions to become a prefix expression:

- $p - q - r / a \rightarrow ((p - q) - (r / a)) \rightarrow ([- p q] - [/ r a]) \rightarrow -- p q / r a$

Quick Quiz: Convert the above infix expression into its postfix form.

Note: You cannot change the expression given to you. For eg. $(p - q)^*(m - n)$ cannot be changed to something like $(p - (q^* m) - n)$.

Let's change this to its postfix equivalent.

- $(p - q)^*(m - n) \rightarrow ((p - q)^*(m - n)) \rightarrow ([pq^-][mn^-]) \rightarrow pq-mn-$

We didn't go to the programming part here. It was intended to be as simple as possible. I hope you understood everything. We'll see automating this process in our next tutorial. Stay connected.

Infix To Postfix Using Stack

In the last tutorial, we had learned to convert an infix expression to its postfix and prefix equivalents manually. Following were the simple steps we followed.

1. Parenthesize the expression following the operators' precedence and their associativity.
2. From the innermost to outermost, keep converting the expressions.

But we didn't talk about their implementation using stacks; rather, we didn't even mention stacks in our last class. Today, we will learn how to convert an infix expression into its postfix equivalent using stacks.

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

1. Start moving left to right from the beginning of the expression.
2. The moment you receive an operand, concatenate it to the postfix expression string.

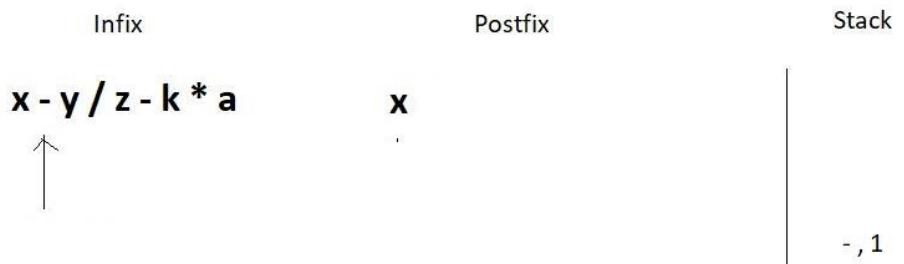
3. And the moment you encounter an operator, move to the stack along with its relative precedence number and see if the topmost operator in the stack has higher or lower precedence. If it's lower, push this operator inside the stack. Else, keep popping operators from the stack and concatenate it to the postfix expression until the topmost operator becomes weaker in precedence relative to the current operator.
4. If you reach the EOE, pop every element from the stack, and concatenate them as well. And the expression you 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.

Infix	Postfix	Stack		
$x - y / z - k * a$	xy	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>/, 2</td> </tr> <tr> <td>-, 1</td> </tr> </table>	/, 2	-, 1
/, 2				
-, 1				

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.

Infix	Postfix	Stack	
$x - y / z - k * a$	$xyz/-$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>-, 1</td> </tr> </table>	-, 1
-, 1			

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.

Infix	Postfix	Stack		
$x - y / z - k * a$	$xyz/-k$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>*, 2</td> </tr> <tr> <td>-, 1</td> </tr> </table>	*, 2	-, 1
*, 2				
-, 1				

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.

Infix	Postfix	Stack
$x - y / z - k * a$	$xyz/-ka^*$	

Follow every step meticulously, and you will find it very easy to master this. You can see if the answer we found at the end is correct manually.

- $x - y / z - k * a \rightarrow ((x - (y / z)) - (k * a)) \rightarrow ((x - [yz /]) - [ka^*]) \rightarrow [xyz/-] - [ka^*] \rightarrow xyz/-ka^*$

And it is indeed a correct conversion. I would now want you to follow the same steps and convert the expression $x + y * z - k$, using the stack method, and verify your answer manually using parentheses.

This was visualizing the conversion process of infix to postfix using stacks. We will see the programming part in the next tutorial. It would be best if you practiced converting a few expressions of your own. If you still feel diffident about using stacks, you must check out the previous videos where we discussed stacks in detail.

Coding Infix to Postfix in C using Stack

We saw earlier how infix expressions can be converted to their other equivalents manually. But when it came to automating the process, we took a different path. We used stacks to take hold of the operators we encountered in the expression. We followed an algorithm to convert an infix expression to its postfix equivalent, which in short, said:

1. We create a string variable that will hold our postfix expression. We start moving from the left to the right. And the moment we receive an operand, we concatenate it to the postfix string. And whenever we encounter an operator, we proceed with the following steps:

- Keep in account the operator and its relative precedence.
- If either the stack is empty or its topmost operator has lower relative precedence, push this operator-precedence pair 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.

2. If you reach the EOE, pop every element from the stack, if there is any, and concatenate them as well. And there, you'll have your postfix expression.

Let us now see the program pursuing the conversion. I have attached the snippets alongwith. Keep checking them while you understand the codes.

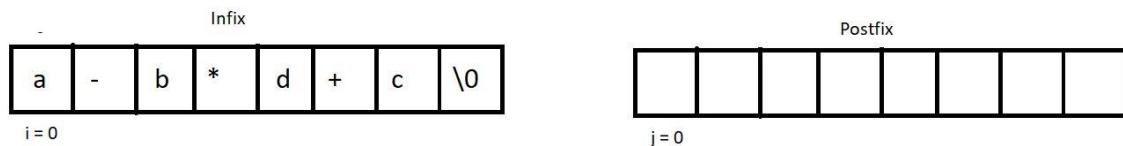
Understanding the program for infix to postfix conversion:

1. First of all, create a character pointer function *infixToPostfix* since the function has to return a character array. And now pass into this function the given expression, which is also a character pointer.

2. Define a struct stack pointer variable *sp*. And give it the required memory in the heap. Create the instance. It's safe to assume that a struct stack element and all its basic operations, *push*, *pop*, etc., have already been defined. You better copy everything from the stack tutorial.

3. Create a character array/pointer *postfix*, and assign it sufficient memory to hold all the characters of the infix expression in the heap.

4. Create two counters, one to traverse through the *infix* and another to traverse and insert in the *postfix*. Refer to the illustration below, which describes the initial conditions.



5. Run a while loop until we reach the EOE of the *infix*. And inside that loop, check if the current index holds an operator, and if it's not, add that character into the *postfix* and increment both the counters by 1. And if it does hold an operator, call another function that would check if the precedence of the *stackTop* is less than the precedence of the current operator. If yes, push it inside the stack. Else, pop the *stackTop*, and add it back into the *postfix*. Increment *j* by 1.

```
char* infixToPostfix(char* infix){  
    struct stack * sp = (struct stack *) malloc(sizeof(struct stack));  
    sp->size = 10;  
    sp->top = -1;  
    sp->arr = (char *) malloc(sp->size * sizeof(char));  
    char * postfix = (char *) malloc((strlen(infix)+1) *  
    sizeof(char));  
    int i=0; // Track infix traversal  
    int j = 0; // Track postfix addition  
    while (infix[i]!='\0')
```

```

    {
        if(!isOperator(infix[i])){
            postfix[j] = infix[i];
            j++;
            i++;
        }
        else{
            if(precedence(infix[i]) > precedence(stackTop(sp))){
                push(sp, infix[i]);
                i++;
            }
            else{
                postfix[j] = pop(sp);
                j++;
            }
        }
    }

    while (!isEmpty(sp))
    {
        postfix[j] = pop(sp);
        j++;
    }

    postfix[j] = '\0';
    return postfix;
}

```

[Copy](#)

Code Snippet 1: Creating the function infixToPostfix

6. It's now time to create the two functions to make this conversion possible. *isOperator* & *precedence* which checks if a character is an operator and compares the precedence of two operators respectively.
7. Create an integer function *isOperator* which takes a character as its parameter and returns 2, if it's an operator, and 0 otherwise.

```
int isOperator(char ch){  
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/'){  
        return 1;  
    } else  
        return 0;  
}
```

Copy

Code Snippet 2: Creating the function *isOperator*

8. Create another integer function *precedence*, which takes a character as its parameter, and returns its relative precedence. It returns 3 if it's a '/' or a '*'. And 2 if it's a '+' or a '-'.

9. If we are still left with any element in the stack at the end, pop them all and add them to the *postfix*.

```
int precedence(char ch){  
    if(ch == '*' || ch=='/'){  
        return 3;  
    } else if(ch == '+' || ch=='-'){  
        return 2;  
    } else  
        return 0;  
}
```

Copy

Code Snippet 3: Creating the function *precedence*

And we have successfully finished writing the codes.

Here is the whole source code:

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
  
struct stack
```

```
{  
    int size;  
    int top;  
    char *arr;  
};
```

```
int stackTop(struct stack* sp){  
    return sp->arr[sp->top];  
}
```

```
int isEmpty(struct stack *ptr)  
{  
    if (ptr->top == -1)  
    {  
        return 1;  
    }  
    else  
    {  
        return 0;  
    }  
}
```

```
int isFull(struct stack *ptr)  
{  
    if (ptr->top == ptr->size - 1)  
    {  
        return 1;  
    }  
    else  
    {  
        return 0;  
    }
```

```
    }
}

void push(struct stack* ptr, char val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n", val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

char pop(struct stack* ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}

int precedence(char ch){
    if(ch == '*' || ch=='/')
        return 3;
    else if(ch == '+' || ch=='-')
        return 2;
    else
        return 0;
}
```

```
}
```

```
int isOperator(char ch){
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
        return 1;
    else
        return 0;
}

char* infixToPostfix(char* infix){
    struct stack * sp = (struct stack *) malloc(sizeof(struct stack));
    sp->size = 10;
    sp->top = -1;
    sp->arr = (char *) malloc(sp->size * sizeof(char));
    char * postfix = (char *) malloc((strlen(infix)+1) * sizeof(char));
    int i=0; // Track infix traversal
    int j = 0; // Track postfix addition
    while (infix[i]!='\0')
    {
        if(!isOperator(infix[i])){
            postfix[j] = infix[i];
            j++;
            i++;
        }
        else{
            if(precedence(infix[i])> precedence(stackTop(sp))){
                push(sp, infix[i]);
                i++;
            }
            else{
                postfix[j] = pop(sp);
                j++;
            }
        }
    }
}
```

```

        }
    }
}

while (!isEmpty(sp))
{
    postfix[j] = pop(sp);
    j++;
}
postfix[j] = '\0';
return postfix;
}

int main()
{
    char * infix = "x-y/z-k*d";
    printf("postfix is %s", infixToPostfix(infix));
    return 0;
}

```

[Copy](#)

Code Snippet 4: Source code for the function infixToPostfix

We now need to check the function for some expressions to see if it works.

```

char * infix = "x-y/z-k*d";
printf("postfix is %s", infixToPostfix(infix));

```

[Copy](#)

Code Snippet 5: Calling the function infixToPostfix

```

postfix is xyz/-kd*-
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

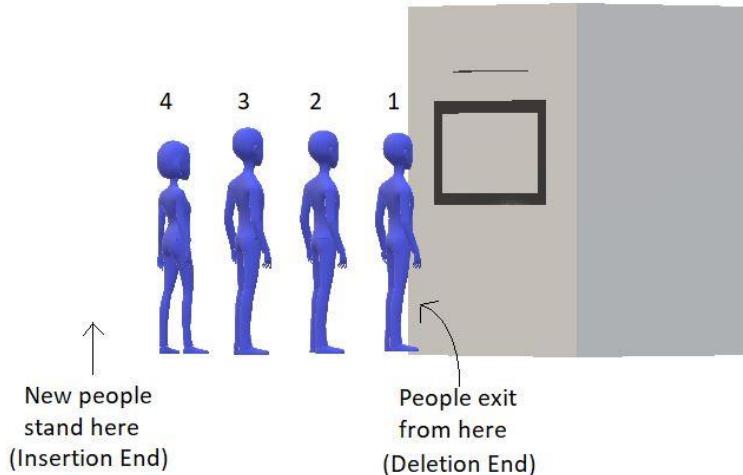
[Copy](#)

Figure 1: Output of the above program

Queue Data Structure

In the last tutorial, we finished learning stacks. And today, we will start a new data structure named queue. Queue as an English word must be a well-known thing to you. We stand in a queue while waiting for our turn to come. Indian railway is one of the places where people stand in a long queue, waiting for their chance to buy a ticket. One important thing to observe, which is quite intuitive, is that your chance comes first when you come first in the queue. And the people standing last, who have joined the queue last, get to buy the ticket in the end.

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 end.

Queue ADT

Data:

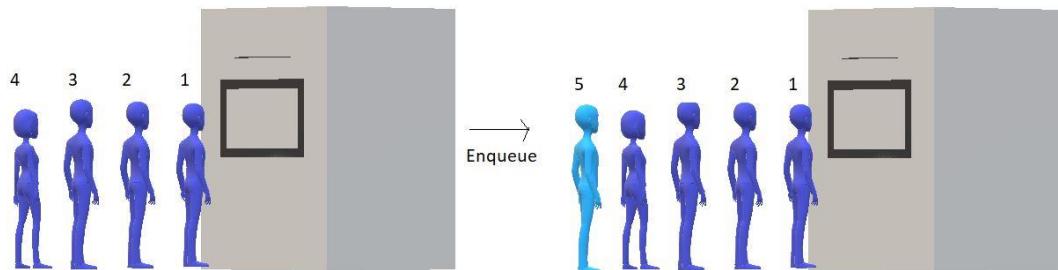
In order to create a queue, we need two pointers, one pointing to the insertion end, to gain knowledge about the address where the new element will be inserted to. And the other pointer pointing to the deletion end, which holds the address of the

element which will be deleted first. Along with that, we need the storage to hold the element itself.

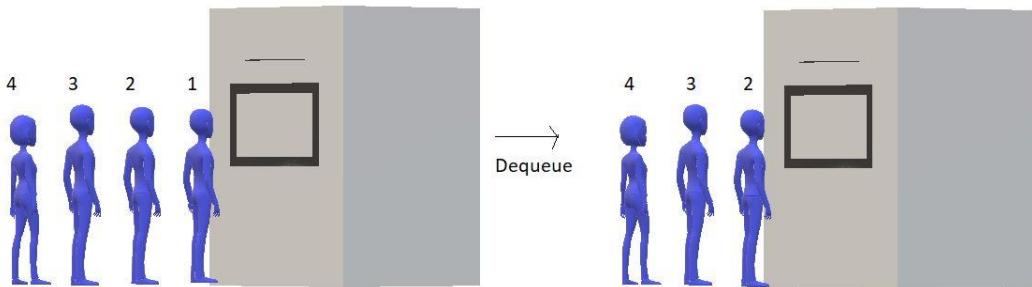
Methods:

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.

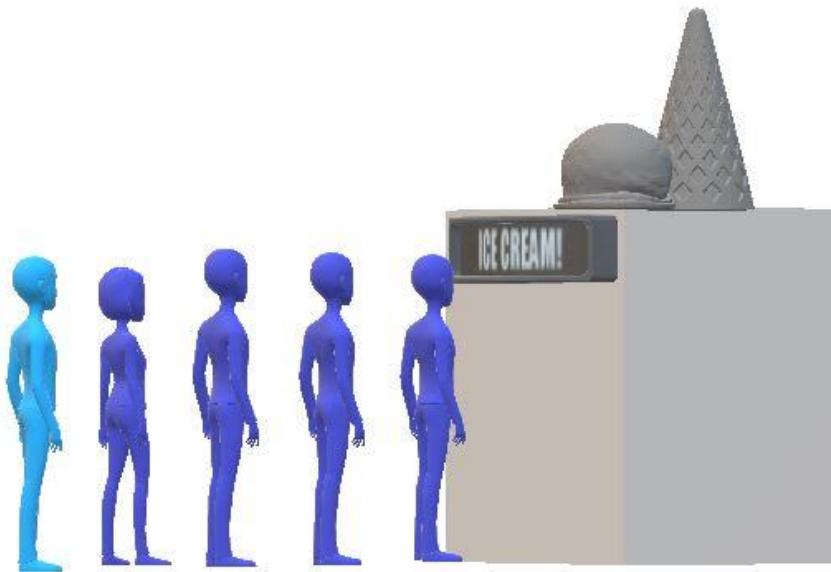
This was our abstract data type, queue. We have in this what we thought would suffice our needs for now. The list could be longer, but in my opinion, this is sufficient.

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. A queue is not limited to ticket counters or shops/malls, it has much wider applications, and you will yourself realize that while we proceed.

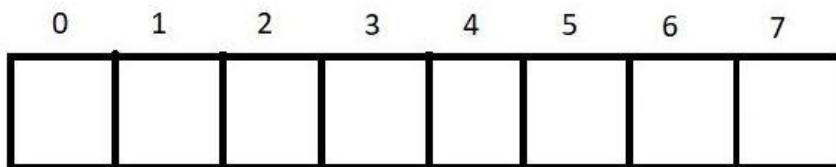
A queue is a collection of elements with certain operations following FIFO (First in First Out) discipline. We insert at one end and delete from the other. And this is what you have to keep in mind for now.

Queue Implementation: Array Implementation of Queue in Data Structure

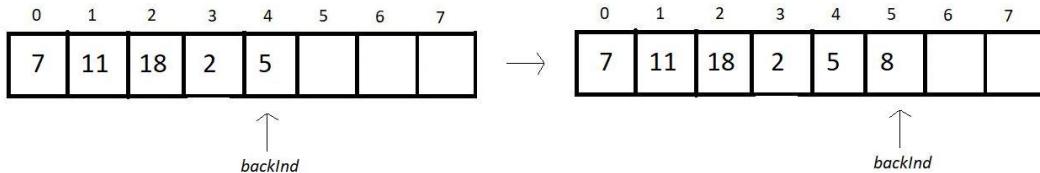
In the last lecture, we introduced to you a new data structure, queue. Today, we'll learn how to implement queues ADT using arrays. During our discussion, we compared its representation to our own lives. It is analogous to a queue in front of any ticket counter or an ice cream shop illustrated below.



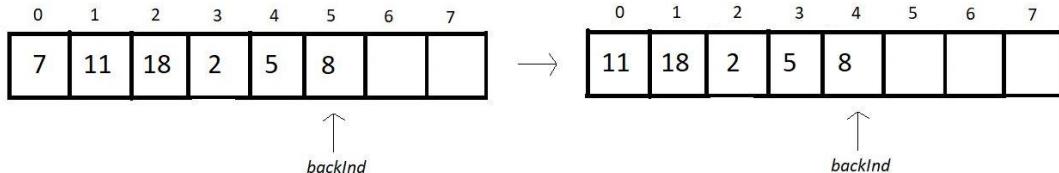
Here, we have shown a branded ice cream shop that is famous enough to have a queue of people waiting to get one of their choices. And the shop owner wants to store the information of these people, so he uses an array to accomplish that. Assuming that we have 8 people and we want to store their information, 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: $backInd = size-1$.

Can there be a better way to accomplish these tasks? The answer is yes. 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):

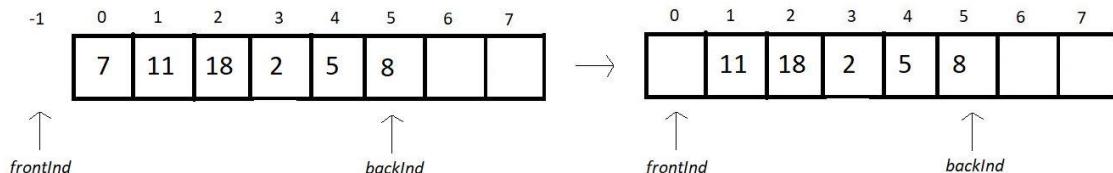
- Remove the element at the zeroth index(no need for that in an array)
- Increment *frontInd* by 1.
- Time complexity: O(1)

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 its Operations in Data Structure

In the last tutorial, we discussed the idea of the implementation of queues using arrays. We talked about the basic operations and their best methods. And we came to the conclusion that if we maintain two index variables, *frontInd* & *backInd*, we can accomplish both enqueue(insertion) and dequeue(deletion) in constant time complexity. Let me just enlist the method we prepared:

1. Insertion(enqueue):

- Increment *backInd* by 1.

- Insert the element
- Time complexity: O(1)

2. Deletion(dequeue):

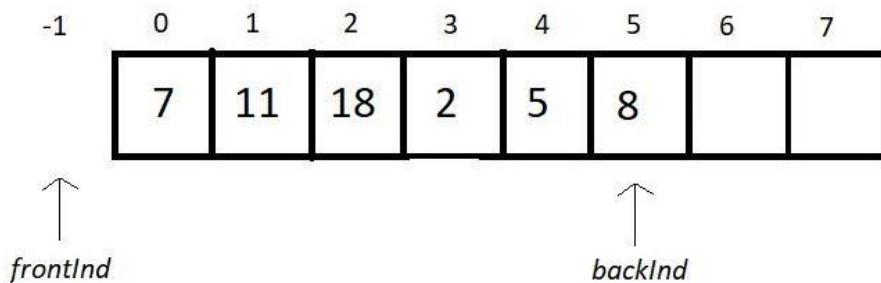
- Remove the element at the zeroth index(no need for that in an array)
- Increment *frontInd* by 1.
- Time complexity: O(1)

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.

Given array below represents a queue:



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

1. *size*: to store the size of the array
2. *frontInd*: to store the index prior to the first element.
3. *backInd*: to store the index of the rearmost element.
4. **arr*: to store the address of the array dynamically allocated in heap.

```
struct queue
{
    int size;
    int frontInd;
    int backInd;
    int* arr;
```

```
};
```

Copy

Now to use this struct element as a queue, you just need to initialize its instances as:

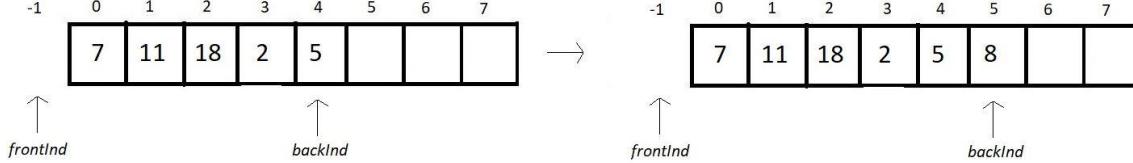
1. struct Queue q; (we are not dynamically allocating q here for now, as we did in stacks).
2. Use dot here, and not arrow operator to assign values to struct members, since q is not a pointer.
3. q.size = 10; (this gives size element the value 10)
4. q.frontInd = q.backInd = -1;(this gives both the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

And this is how you 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.

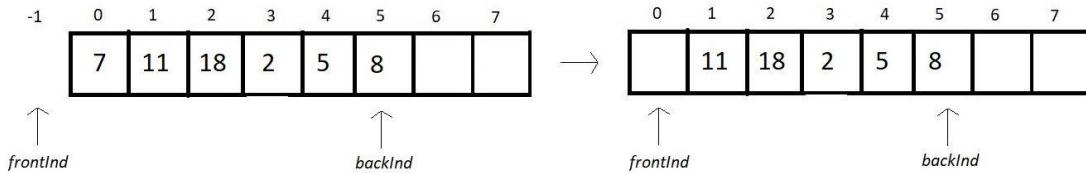
1. First, check if the queue is already not full.
2. If it is, it is the case of queue overflow, else just increment *backInd* by 1, insert the new element there. Follow the illustration below.



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:

3. First, check if the queue is already not empty.
4. 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.

So, these are the basic operations of a queue. We have implemented everything using arrays. We'll see the programming part in the next tutorial. Keep up with the playlist, and you won't miss anything.

C Code For Queue and its Operations Using Arrays in Data Structure

In the last tutorial, we finished learning concepts behind implementing the basic operations of a queue ADT using arrays. Those were enqueue, dequeue, *isEmpty* and *isFull*. Today we will learn how to program all of those. Without further ado, let's move to our editors!

I have attached the whole source code for your referral. Follow it while understanding.

Understanding the code snippet below:

- First of all, start by creating a struct named *queue*, and define all of its four members we discussed yesterday. An integer variable *size* to store the size of the array, another integer variable *f* to store the front end index, and an integer variable *r* to store the index of the rear end. Then, define an integer pointer *arr* to store the address of the dynamically allocated array.

```
struct queue
{
    int size;
```

```
int f;  
int r;  
int* arr;  
};
```

[Copy](#)

Code Snippet 1: Declaring struct queue

2. In main, declare a struct queue *q*, and initialize its instances. Declare some size of the array, let 100. Initialize both *f* and *r* with -1. And allocate memory in heap for *arr* using malloc. Don't forget to include the header file <stdlib.h>

```
struct queue q;  
q.size = 4;  
q.f = q.r = 0;  
q.arr = (int*) malloc(q.size*sizeof(int));
```

[Copy](#)

Code Snippet 2: Defining and initializing a struct element q

3. Creating Enqueue:

Create a void function *enqueue*, pass the pointer to the struct queue *q*, and the value to insert as parameters. First of all, check if the queue is full by calling the *isFull* function. If it returns 1, then print the condition of the queue overflow and return. Else, increase the *r* value of *q* using the arrow operator, and insert the new value at the index *r* of the array *arr*.

```
void enqueue(struct queue *q, int val){  
    if(isFull(q)){  
        printf("This Queue is full\n");  
    }  
    else{  
        q->r++;  
        q->arr[q->r] = val;  
        printf("Enqueued element: %d\n", val);  
    }  
}
```

```
}
```

[Copy](#)

Code Snippet 3: Creating the enqueue function

4. Creating isFull:

Create an integer function *isFull*, and pass into it the pointer to the struct queue *q* as the only parameter. In the function, check if the *r*element of struct queue *q* is equal to the (size element)-1. If it is, then there is no space left in the queue to insert elements anymore, hence return 1, else 0.

```
int isFull(struct queue *q){  
    if(q->r==q->size-1){  
        return 1;  
    }  
    return 0;  
}
```

[Copy](#)

Code Snippet 4: Creating the isFull function

5. Creating Dequeue:

Create an integer function *dequeue*, and pass the pointer to the struct queue *q*, as the only parameter into it. In the function, first of all, check if the queue is already not empty by calling the *isEmpty* function. If it returns 1, then print the condition of the queue underflow and return. Else, increase the *f*value of *q* using the arrow operator, and store the value at the index *f*of the array in some integer variable *a*. Later, return *a*.

```
int dequeue(struct queue *q){  
    int a = -1;  
    if(isEmpty(q)){  
        printf("This Queue is empty\n");  
    }  
    else{  
        q->f++;  
        a = q->arr[q->f];  
    }  
}
```

```
    return a;  
}
```

Copy

Code Snippet 5: Creating the dequeue function

6. Creating isEmpty:

Create an integer function *isEmpty*, and pass into it the pointer to the struct queue *q*, as the only parameter. Inside the function, check if the *r*element of the *q* is equal to the *f*element of the *q*. Intuitively speaking, the difference between the values of *f* & *r* is the size of the queue. And if they both are equal, the size is 0. Therefore, if they are equal, return 1, else return 0.

```
int isEmpty(struct queue *q){  
    if(q->r==q->f){  
        return 1;  
    }  
    return 0;  
}
```

Copy

Code Snippet 6: Creating the isEmpty function

Here is the whole source code:

```
#include<stdio.h>  
#include<stdlib.h>  
  
struct queue  
{  
    int size;  
    int f;  
    int r;  
    int* arr;  
};
```

```
int isEmpty(struct queue *q){  
    if(q->r==q->f){  
        return 1;  
    }  
    return 0;  
}
```

```
int isFull(struct queue *q){  
    if(q->r==q->size-1){  
        return 1;  
    }  
    return 0;  
}
```

```
void enqueue(struct queue *q, int val){  
    if(isFull(q)){  
        printf("This Queue is full\n");  
    }  
    else{  
        q->r++;  
        q->arr[q->r] = val;  
        printf("Enqueued element: %d\n", val);  
    }  
}
```

```
int dequeue(struct queue *q){  
    int a = -1;  
    if(isEmpty(q)){  
        printf("This Queue is empty\n");  
    }  
    else{
```

```
    q->f++;
    a = q->arr[q->f];
}
return a;
}

int main(){
    struct queue q;
    q.size = 4;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // Enqueue few elements
    enqueue(&q, 12);
    enqueue(&q, 15);
    enqueue(&q, 1);
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    printf("Dequeuing element %d\n", dequeue(&q));
    enqueue(&q, 45);
    enqueue(&q, 45);
    enqueue(&q, 45);

    if(isEmpty(&q)){
        printf("Queue is empty\n");
    }
    if(isFull(&q)){
        printf("Queue is full\n");
    }

    return 0;
}
```

```
}
```

[Copy](#)

Code Snippet 7: Implementing a queue and its operations using arrays

Let's now check if our functions work all good. Since we have not inserted any element in the queue yet, we'll see what the *isEmpty* function has to say.

```
if(isEmpty(&q)){  
    printf("Queue is empty\n");  
}
```

[Copy](#)

Code Snippet 8: Using the isEmpty function

The output we received was:

```
Queue is empty
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above program

Let us now insert/enqueue some elements inside the queue.

```
enqueue(&q, 12);  
enqueue(&q, 15);
```

[Copy](#)

Code Snippet 9: Using the enqueue function

Our terminal had the following output:

```
Enqueued element: 12  
Enqueued element: 15
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 2: Output of the above program

Let's dequeue the elements we inserted.

```
printf("Dequeuing element %d\n", dequeue(&q));  
printf("Dequeuing element %d\n", dequeue(&q));
```

Copy

Code Snippet 10: Using the dequeue function

And the output we received was:

```
Dequeuing element 12  
Dequeuing element 15
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 3: Output of the above program

Introduction to Circular Queue in Data Structures

We have completed learning the basics of queues and its operations so far. Before proceeding to the next section, let's catch up on what we covered in queues. Using a real-life example, we explored the meaning of queue. We learn that it follows the FIFO principle. We implemented a queue ADT and its basic operations using arrays. We wrote their code in C.

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.

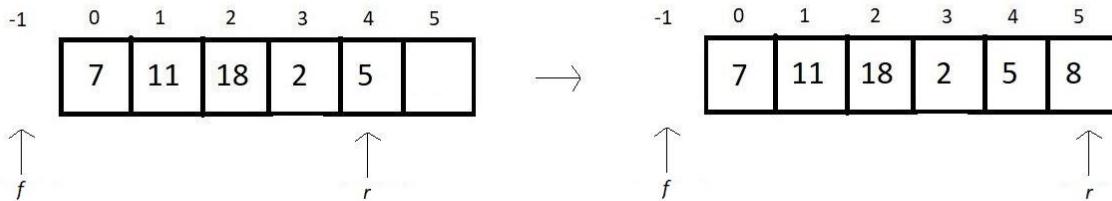


Figure 1: Using two integer variables to maintain the ends of a queue

Now, we start dequeuing some elements. Let's remove the first three elements. And now, if you 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.

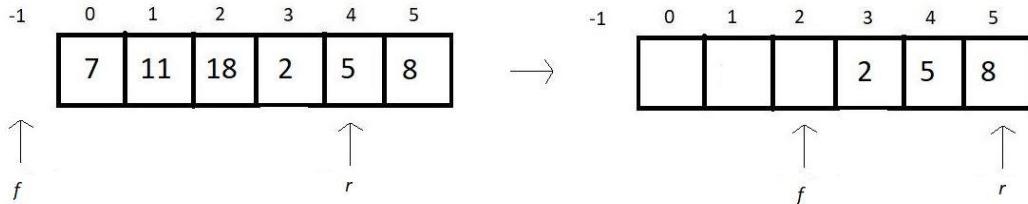


Figure 2: Dequeueing leaves vacant spaces behind

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.

1. 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.

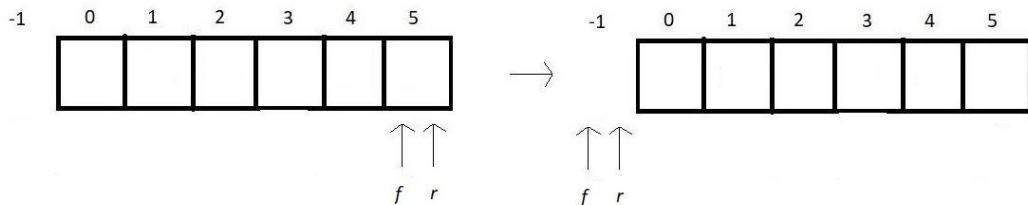


Figure 3: Resetting the index variables to -1

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 $(i+1) \% \text{size}$. 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.

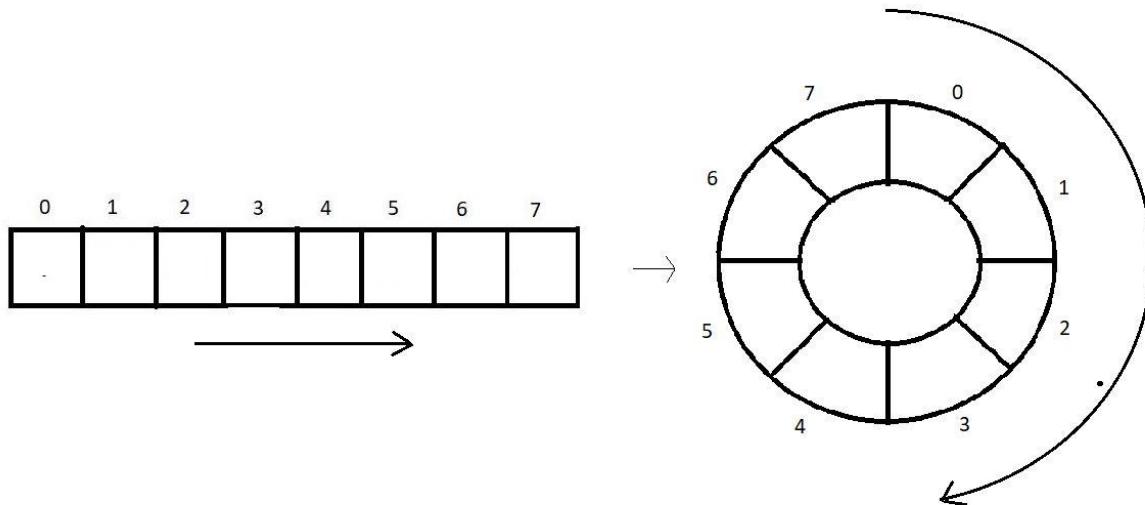


Figure 4: Conversion of a linear queue to a circular queue

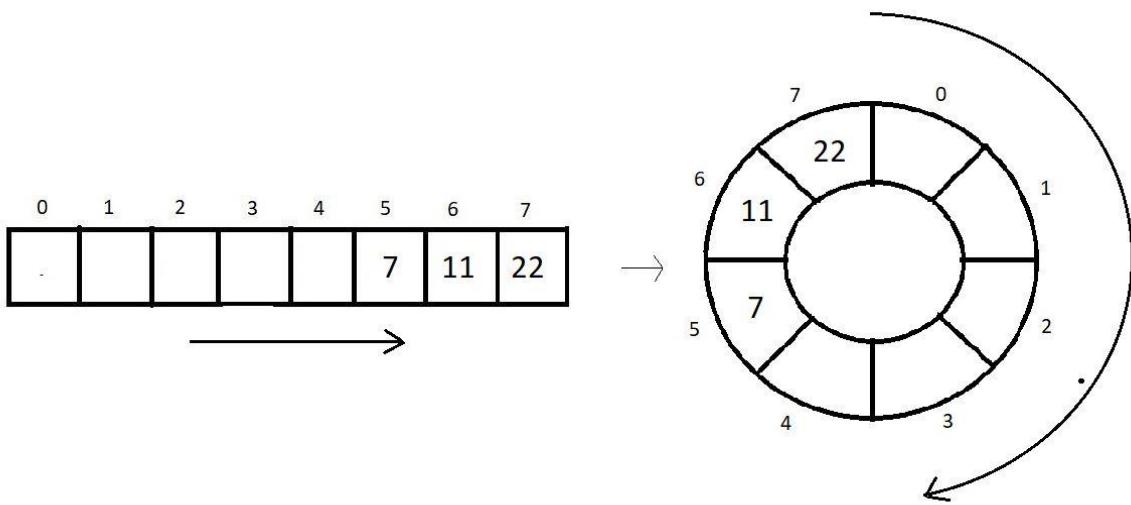
And this is the circular implementation of the same array we used to implement linearly. This allows the leftover spaces to be used again. This wheel type array is called the **circular queue**.

enqueue(), dequeue() & other Operations on Circular Queue

In the last tutorial, we gave you a basic introduction to circular queues and their necessity. We made you visualize the differences between a linear and a circular queue. We saw the advantages of a circular queue over a linear/normal queue. Today, we'll finish the implementation part of a circular queue and its operations using arrays.

If you remember, we converted a linear queue into a circular queue using a mathematical tool called **modulus**. This enables the feature of incrementing the indices circularly, where 0 comes after every *size - 1* index. See this illustration below, and realize how a queue implemented using a linear array of size 8 couldn't utilize the memory space efficiently. Once the rear index variable reaches the limit, the queue disables further enqueueing even if the spaces behind go unused.

But once you convert this linear/normal queue into a circular one, it enables further enqueueing until the queue actually gets full. We could now reuse the vacant spaces left after a dequeue operation by going through them again and again circularly.



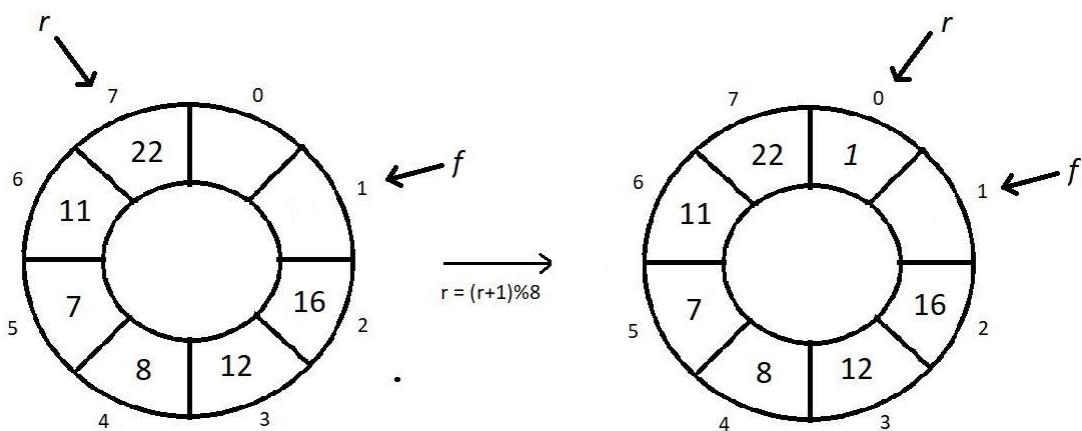
Note: Circular increment lets us access the queue indices circularly, which means, after we finish visiting the 7th index in the above illustration, we again come at the zeroth index.

Let us now see the operations one by one.

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,

1. 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.
2. 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.

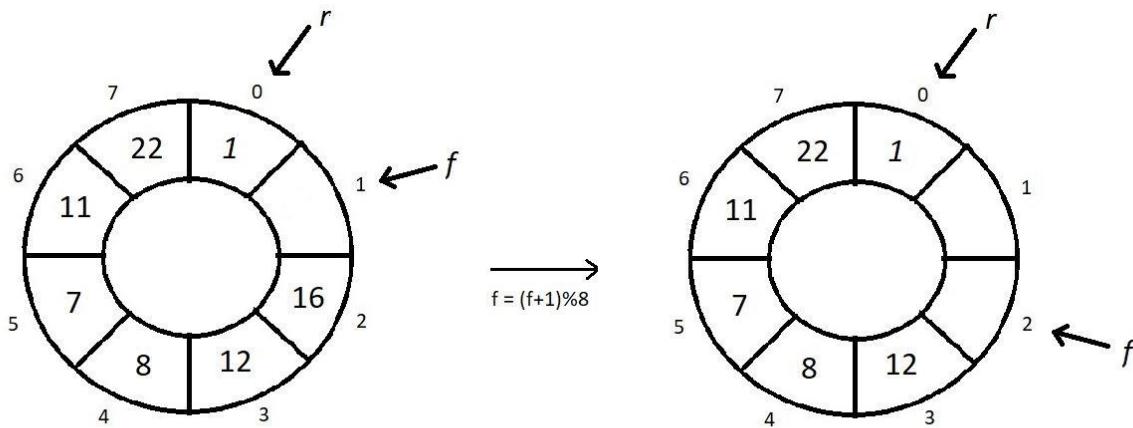


Now, since the f is just next to the r , the queue is full, and no more elements can get pushed.

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,

1. 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 you'll be amazed to know that it works here as well. There are zero modifications here.
2. 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.



Condition for *isEmpty*:

1. 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*:

1. 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.

C Code For Circular Queue & Operations on Circular Queue

We have already finished learning about the implementation of circular queues using arrays. We saw the algorithms behind enqueueing and dequeuing elements in a circular queue. We saw the conditions for circular queues to be declared full and

empty. Before proceeding, if you somehow missed the last lecture, I would recommend seeing that first, since we discussed key concepts there. Today's lecture will be solely focusing on the programming part.

Note: In circular queues, the *f* is always an index behind the first element which means there is always a vacant index in circular queues.

Let's get your editors involved. I have attached the source code below. Keep it handy while understanding the code.

Understanding the code snippet below:

1. First of all, I would like you all to copy everything from the queue implementation program since things are more or less the same, and circular queues are just a variation of normal queues. So, we would just make subtle modifications and things will work well.

2. Now, since it was a queue, replace queues with circular queues. Start by changing the struct named *queue* to a struct named *circularQueue*, and all the four members remain the same as queue. (An integer variable *size* to store the size of the array, another integer variable *f* to store the index of the front end, an integer variable *r* to store the index of the rear end. And an integer pointer *arr* to store the address of the dynamically allocated array.)

```
struct circularQueue
{
    int size;
    int f;
    int r;
    int* arr;
};
```

Copy

Code Snippet 1: Declaring struct circularQueue

3. In main, we had declared a struct circularQueue *q*, and initialized its instances. Here is a subtle change, we don't initialize circular queues' *f* and *r* with -1, rather 0. Since -1 is unreachable in circular incrementation. Leave everything as it is.

```
struct circularQueue q;
q.size = 4;
q.f = q.r = 0;
```

```
q.arr = (int*) malloc(q.size*sizeof(int));
```

Copy

Code Snippet 2: Defining and initialising a struct element *q*

4. Modifying isEmpty:

If you remember, the condition for isEmpty remains the same for both queues and circular queues. So, no modifications are needed here. Leave this as well.

```
int isEmpty(struct circularQueue *q){  
    if(q->r==q->f){  
        return 1;  
    }  
    return 0;  
}
```

Copy

Code Snippet 3: Modifying the *isEmpty* function

5. Modifying isFull:

Earlier, isFull checked if our rear has reached the limit of the array. And if it did, we returned the overflow statement. But now, the queue isn't full until technically. So, just see if the index next to the rear becomes front or not. Use circular increment (modulus) to pursue any increment in a circular queue.

So, check if $(r + 1) \bmod \text{size}$ is equal to the f . If it is, then there is no space left in the queue to insert anymore elements, hence return 1, else 0.

```
int isFull(struct circularQueue *q){  
    if((q->r+1)%q->size == q->f){  
        return 1;  
    }  
    return 0;  
}
```

Copy

Code Snippet 4: Modifying the *isFull* function

6. Modifying Enqueue:

In the function *enqueue*, first of all, check if the queue is full by calling the *isFull* function. If it returns 1, then print the condition of the queue overflow and return. Else, increase the *r* value of *q* circularly using the arrow operator and modulus, and insert the new value at the increased index *r* of the array *arr*.

```
void enqueue(struct circularQueue *q, int val){  
    if(isFull(q)){  
        printf("This Queue is full");  
    }  
    else{  
        q->r = (q->r +1)%q->size;  
        q->arr[q->r] = val;  
        printf("Enqueued element: %d\n", val);  
    }  
}
```

[Copy](#)

Code Snippet 5: Modifying the *enqueue* function

7. Modifying Dequeue:

Earlier when we dequeued in a queue, we would simply increase the value of *f* by 1. We would now increase but circularly, and that would be it.

In the function *dequeue*, first, check whether the circular queue is already not empty by calling *isEmpty*. If it returns 1, then print the condition of the queue underflow and return. Else, increase the *f* value of *q* using the arrow operator circularly, and store the value at the index *f* of the array in some integer variable *a*. Later, return *a*.

```
int dequeue(struct circularQueue *q){  
    int a = -1;  
    if(isEmpty(q)){  
        printf("This Queue is empty");  
    }  
    else{  
        q->f = (q->f +1)%q->size;  
        a = q->arr[q->f];  
    }  
}
```

```
    }
    return a;
}
```

[Copy](#)

Code Snippet 6: Modifying the *dequeue* function

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

struct circularQueue
{
    int size;
    int f;
    int r;
    int* arr;
};

int isEmpty(struct circularQueue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}

int isFull(struct circularQueue *q){
    if((q->r+1)%q->size == q->f){
        return 1;
    }
}
```

```
    return 0;
}

void enqueue(struct circularQueue *q, int val){
    if(isFull(q)){
        printf("This Queue is full");
    }
    else{
        q->r = (q->r +1)%q->size;
        q->arr[q->r] = val;
        printf("Enqueued element: %d\n", val);
    }
}

int dequeue(struct circularQueue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty");
    }
    else{
        q->f = (q->f +1)%q->size;
        a = q->arr[q->f];
    }
    return a;
}

int main(){
    struct circularQueue q;
    q.size = 4;
    q.f = q.r = 0;
```

```

q.arr = (int*) malloc(q.size*sizeof(int));

// Enqueue few elements
enqueue(&q, 12);
enqueue(&q, 15);
enqueue(&q, 1);
printf("Dequeuing element %d\n", dequeue(&q));
printf("Dequeuing element %d\n", dequeue(&q));
printf("Dequeuing element %d\n", dequeue(&q));
enqueue(&q, 45);
enqueue(&q, 45);
enqueue(&q, 45);

if(isEmpty(&q)){
    printf("Queue is empty\n");
}
if(isFull(&q)){
    printf("Queue is full\n");
}

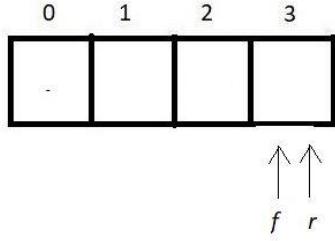
return 0;
}

```

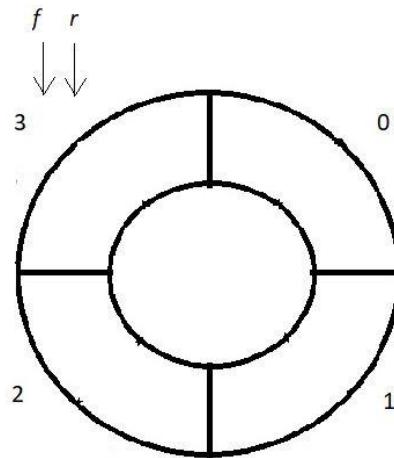
[Copy](#)

Code Snippet 7: Implementing a circular queue and its operations using arrays

Now you can actually see what happened here. Earlier when we enqueued elements to the full and dequeued everything again as seen in the illustration below, the queue still remained full.



You can even use the queue implementation code to see that. But now when you enqueue elements to its full and delete everything, the circular queue becomes empty, unlike the normal queue. You can very easily now insert at $(3+1)\%4$ index which is the zeroth index. I'll show you that using the code.



Let us now insert/enqueue three elements inside the queue. And see if it reverts the **full** message.

```

enqueue(&q, 12);
enqueue(&q, 15);
enqueue(&q, 1);
if(isFull(&q)){
    printf("Queue is full\n");
}

```

[Copy](#)

Code Snippet 8: Using the *enqueue* function

Our terminal had the following output:

```
Enqueued element: 12
Enqueued element: 15
Enqueued element: 1
Queue is full
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above program

Now let's dequeue everything and see if the queue again becomes empty or not.

```
printf("Dequeuing element %d\n", dequeue(&q));
printf("Dequeuing element %d\n", dequeue(&q));
printf("Dequeuing element %d\n", dequeue(&q));
if(isEmpty(&q)){
    printf("Queue is empty\n");
}
```

[Copy](#)

Code Snippet 9: Using the *dequeue* function

And the output we received was:

```
Dequeuing element 12
Dequeuing element 15
Dequeuing element 1
Queue is empty
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

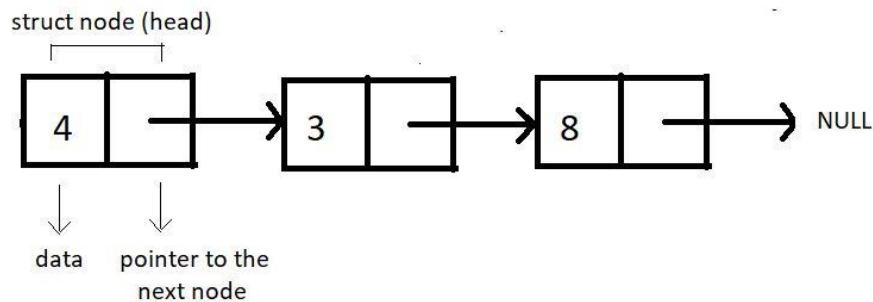
Figure 2: Output of the above program

Queue Using Linked Lists

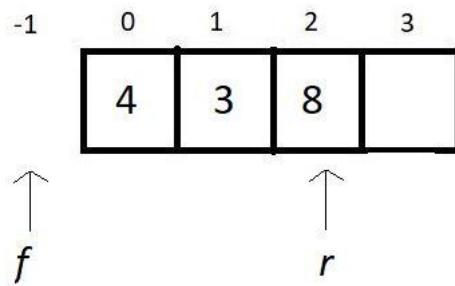
Up until now, queues had been implemented using arrays. We have another alternative that is a must to learn and has been examiners' favorite topic, implementing queues using linked lists. Today, we'll see how to implement queues using linked lists and some of its operations.

Implementing queues using linked lists tests your proficiency in using/ handling both queues and linked lists. And, in case if you have missed either or both of these, I would recommend visiting them first before proceeding.

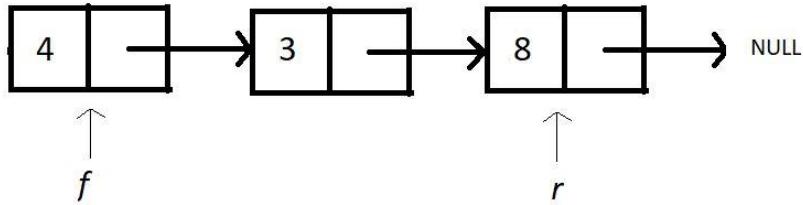
If you remember, linked lists are chain-like structures with nodes having two parts, an integer variable to hold data and another node pointer to hold the address of the next node. Below illustrated is a linked list with three nodes. The last node points to NULL. And the first node is called the *head*.



Moving to the basics of a queue, a queue represents a line or sequence of elements where the elements follow the FIFO discipline. The element inserting the first gets removed the first. 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.



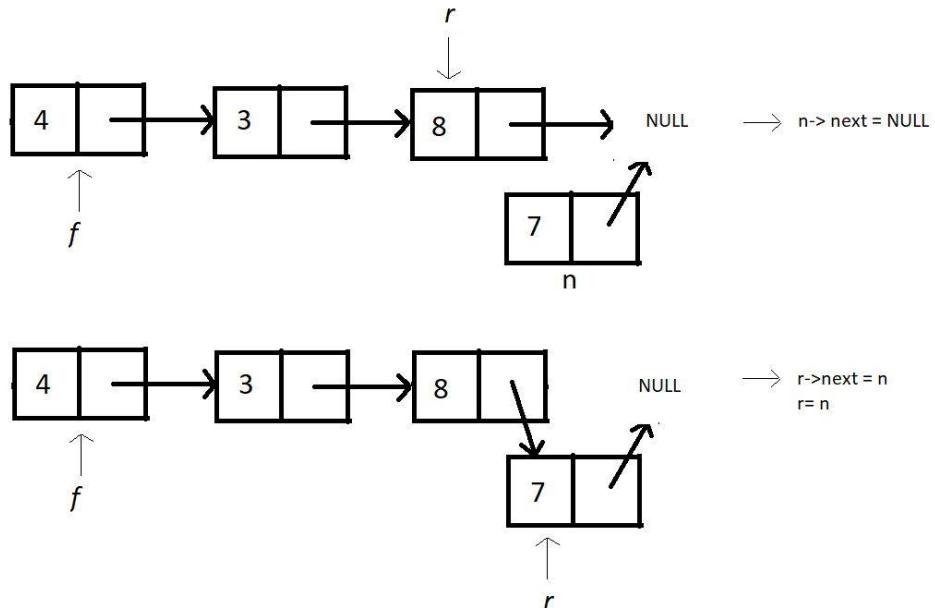
Since we are implementing this queue using a linked list, the index variables are no longer integers. These become the pointers to the front and the rear nodes. And the queue somewhat starts looking like this.



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. As we discussed this thoroughly in our past lectures, you should not find this difficult. Inserting a new node at the end requires you to follow few steps:

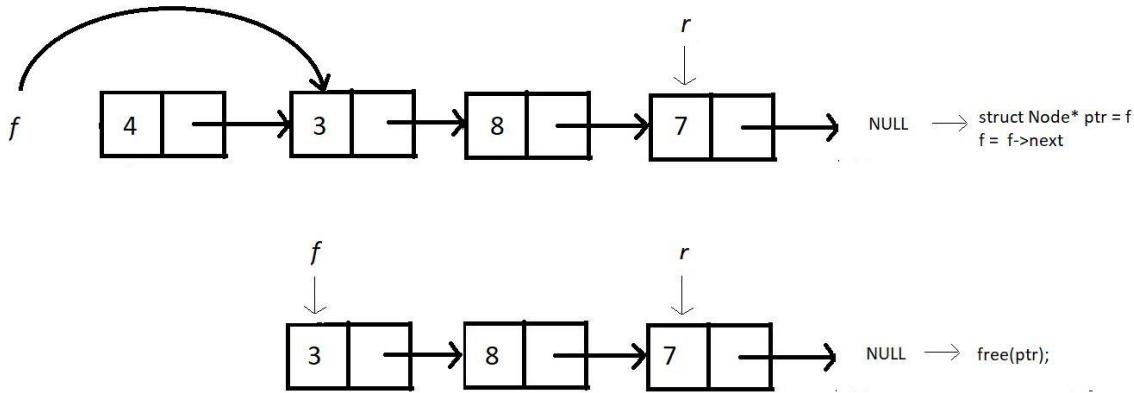
1. Check if there is a space left in the heap for a new node.
2. If there is, create a new node n , assign it memory in heap, and fill its data with the new value the user has given.
3. Point the next member of this new node n to NULL, and point the next member of the r to n . And make r equal to n . And we are done.
4. 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 you to follow few steps:

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



Condition for *isEmpty*:

The only condition for the queue linked list to be empty is that the *fnode* 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 *newnode* becomes NULL when created.

Implementing Queue Using Linked List in C Language (With Code)

In the last lecture, we learned to implement queues using linked lists. We talked about the enqueue and dequeue methods. In the queue linked list, we saw the conditions for its full or empty state. Today, we'll code these implementations in C. Before proceeding, make sure you have finished till here. I would recommend seeing that first if you somehow missed the last lecture since we discussed the concepts there.

Let's move onto our editors. I have attached the source code below for your reference.

Understanding the below code snippet:

1. First of all, I'll make you all aware of the things we have already completed. And I'll make you feel confident about linked lists and queues.
2. We had studied linked lists before, where we studied the traversal methods, insertion at different positions, deletion at different positions, cases of empty and full. And we have completed queues as well and their basic operations.
3. Today, we'll integrate our knowledge of both to implement queues using linked lists.
4. We don't need to copy everything we learned there in the linked lists. We will move with the basics, and this might feel like a revision of the past lectures.
5. Create a struct Node with two of its members, one integer variable `data` to store the data, and another struct Node pointer `next` to store the address of the next node.

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

[Copy](#)

Code Snippet 1: Creating the struct Node

6. Globally, create two struct Node pointers `f` and `r`, which would be used to mark the front and the rear ends. Declaring globally helps us use them in functions.

Creating Enqueue:

We learned in the last lecture that to enqueue, we only use the rear pointer and add a new node at the end of the list. So, create a void function `enqueue`, and the value to enqueue is the only parameter since we have declared the pointers `f` and `r` globally. In the function, create a new struct Node pointer `n`, and assign its memory in heap dynamically using malloc. Don't forget to include the header file `<stdlib.h>`. Then check if the queue is full or, in other words, if there is no space in the heap. And that can be done by checking if the new pointer `n` equals NULL. If it does, then print the condition of the queue overflow and return. Else, insert the val in the `data` member of `n`, and make this node point to NULL. If you recall, we discussed a special case, where we were inserting in the list for the first time, when both `f` and `r` equals NULL. For this case, make both `f` and `r` equal to `n`, and for all the other cases, just make the `r` point the new node `n`. Ultimately make `r` equal to `n` since `n` becomes our new rear end. And that would be all.

```
void enqueue(int val){
    struct Node *n = (struct Node *) malloc(sizeof(struct Node));
```

```

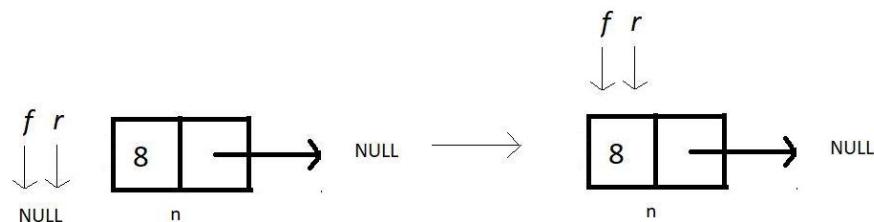
if(n==NULL){
    printf("Queue is Full");
}
else{
    n->data = val;
    n->next = NULL;
    if(f==NULL){
        f=r=n;
    }
    else{
        r->next = n;
        r=n;
    }
}
}

```

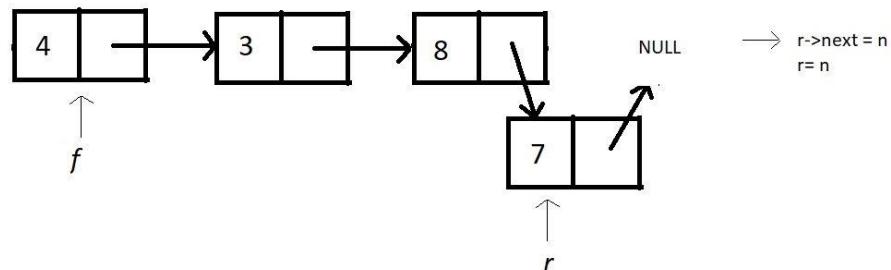
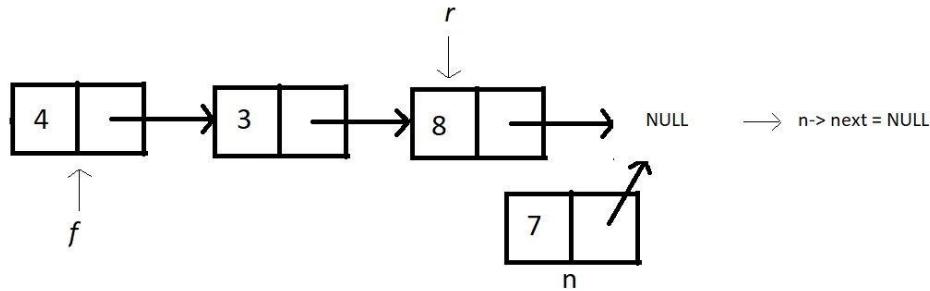
[Copy](#)

Code Snippet 2: Creating the enqueue function

Exception case:



All the other cases:



Creating Dequeue:

As we discussed in the last lecture, Dequeue needs you to just delete the head node, which is the *f node here*. So, create an integer function *dequeue*. And we have no parameters to pass. Create a struct Node pointer *ptr* to hold the node we will delete. Make *ptr* equal to *f*. In the function, check if the queue is already not empty by checking if our front *f* is *NULL* or not. If it is *NULL*, then print the condition of the queue underflow and return. Else, make *f* equal to the next node to *f*. Store the data of *ptr* in an integer variable *val*. We can now free the pointer *ptr*. And return *val*, which is the data of the node we deleted.

```
int dequeue()
{
    int val = -1;
    struct Node *ptr = f;
    if(f==NULL){
        printf("Queue is Empty\n");
    }
    else{
        f = f->next;
        val = ptr->data;
```

```
    free(ptr);  
}  
return val;  
}
```

[Copy](#)

Code Snippet 3: Creating the dequeue function

After every operation, we may need to have the traversal function, which you can copy from the previous lectures. Nothing in that needs to be modified.

Here is the whole source code:

```
#include <stdio.h>  
#include <stdlib.h>  
  
struct Node *f = NULL;  
struct Node *r = NULL;  
  
struct Node  
{  
    int data;  
    struct Node *next;  
};  
  
void linkedListTraversal(struct Node *ptr)  
{  
    printf("Printing the elements of this linked list\n");  
    while (ptr != NULL)  
    {  
        printf("Element: %d\n", ptr->data);  
        ptr = ptr->next;  
    }  
}
```

```
void enqueue(int val)
{
    struct Node *n = (struct Node *) malloc(sizeof(struct Node));
    if(n==NULL){
        printf("Queue is Full");
    }
    else{
        n->data = val;
        n->next = NULL;
        if(f==NULL){
            f=r=n;
        }
        else{
            r->next = n;
            r=n;
        }
    }
}
```

```
int dequeue()
{
    int val = -1;
    struct Node *ptr = f;
    if(f==NULL){
        printf("Queue is Empty\n");
    }
    else{
        f = f->next;
        val = ptr->data;
        free(ptr);
    }
}
```

```
    return val;  
}  
  
int main()  
{  
    linkedListTraversal(f);  
    printf("Dequeuing element %d\n", dequeue());  
    enqueue(34);  
    enqueue(4);  
    enqueue(7);  
    enqueue(17);  
    printf("Dequeuing element %d\n", dequeue());  
    printf("Dequeuing element %d\n", dequeue());  
    printf("Dequeuing element %d\n", dequeue());  
    printf("Dequeuing element %d\n", dequeue());  
    linkedListTraversal(f);  
    return 0;  
}
```

[Copy](#)

Code Snippet 4: Implementing queues using linked lists

Now, let's check if these methods work well. First, we'll enqueue some elements in the queue, and to check if that actually happens, we'll display the elements using traversal.

```
enqueue(34);  
enqueue(4);  
enqueue(7);  
enqueue(17);  
linkedListTraversal(f);
```

[Copy](#)

Code Snippet 5: Using the enqueue function

And the output we received was:

```
Printing the elements of this linked list  
Element: 34  
Element: 4  
Element: 7  
Element: 17
```

Copy

Figure 1: Output of the above program

Let us now dequeue everything. And focus on the order of elements being dequeued.

```
printf("Dequeuing element %d\n", dequeue());  
printf("Dequeuing element %d\n", dequeue());  
printf("Dequeuing element %d\n", dequeue());  
printf("Dequeuing element %d\n", dequeue());  
linkedListTraversal(f);
```

Copy

Code Snippet 6: Using the dequeue function

And the output this time was;

```
Dequeuing element 34  
Dequeuing element 4  
Dequeuing element 7  
Dequeuing element 17  
Printing the elements of this linked list
```

Copy

Figure 2: Output of the above program

Double-Ended Queue in Data Structure (DE-Queue Explained)

In the last lecture, we finished learning about queues. We saw both queues and circular queues. We implemented queues using both arrays and linked lists. We saw all their operations. There is actually nothing left there in

queues except one interesting topic, which is DE-Queue. It should not be confused with the dequeue we learnt. It is **Double Ended Queues**.

We had certain characteristics in normal queues, which I would like to summarize here:

1. A queue is very similar to the real-life queue, where you stand in the last and wait for your turn.
2. Similarly, the elements get inserted from one end and exit from the other.
3. We had two pointers cum index variables to maintain the two ends of this queue.
4. We followed the FIFO principle throughout the lectures.

And now, in **DEQueue**, we don't follow the FIFO principle. 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.

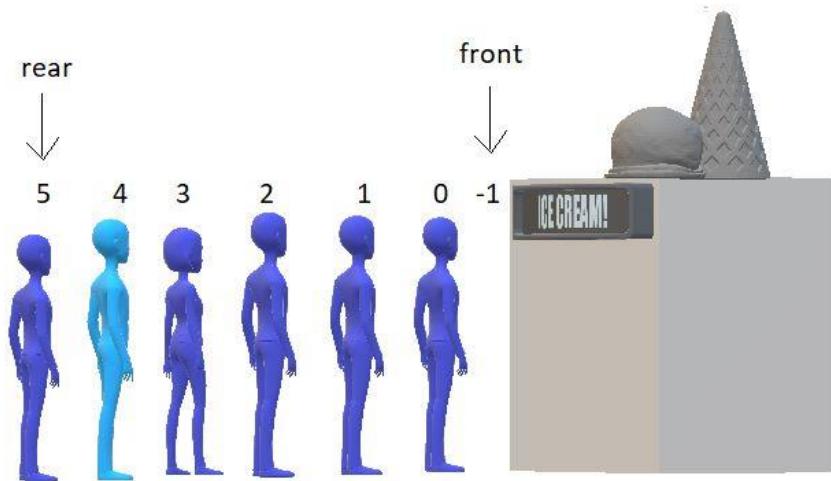
Double-ended queues, hence, have the following characteristics:

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

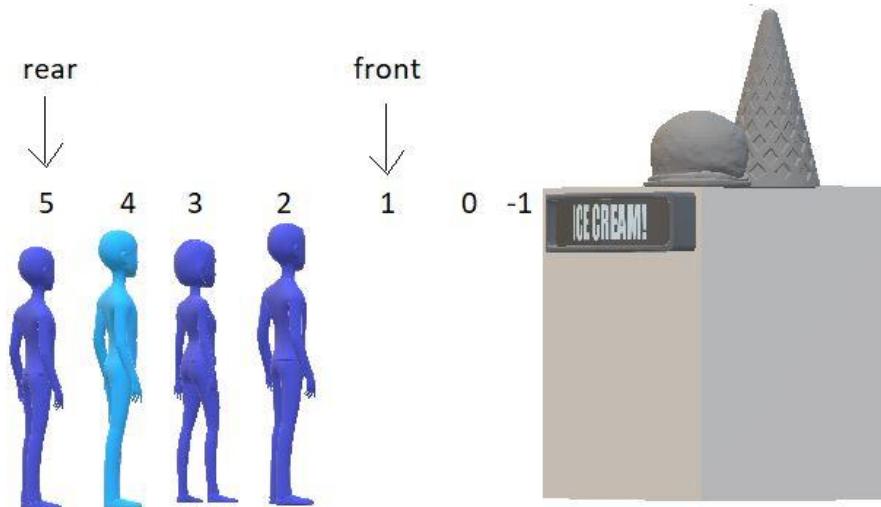
You would assume the implementation part of double-ended queues to be on the tough side, but believe me, it is straightforward to consume. I'll use illustrations to make you understand things better.

Insertion in a DEQueue:

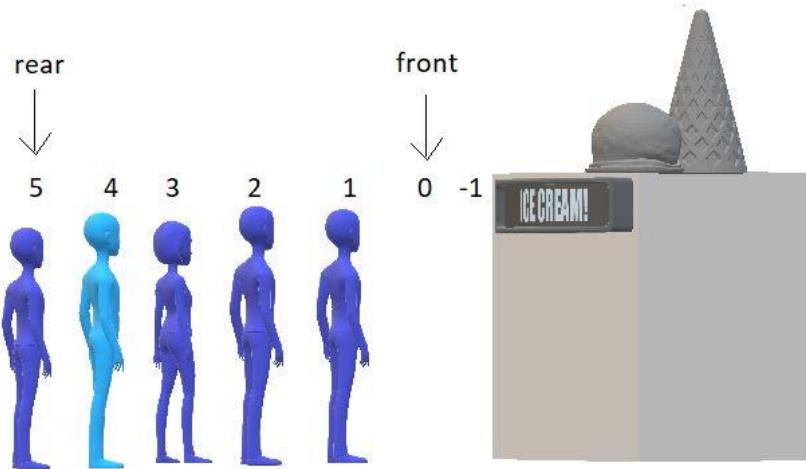
Insertion in a DEQueue is very intuitive. Follow the illustration below:



Now since the front has no space to insert, you 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:

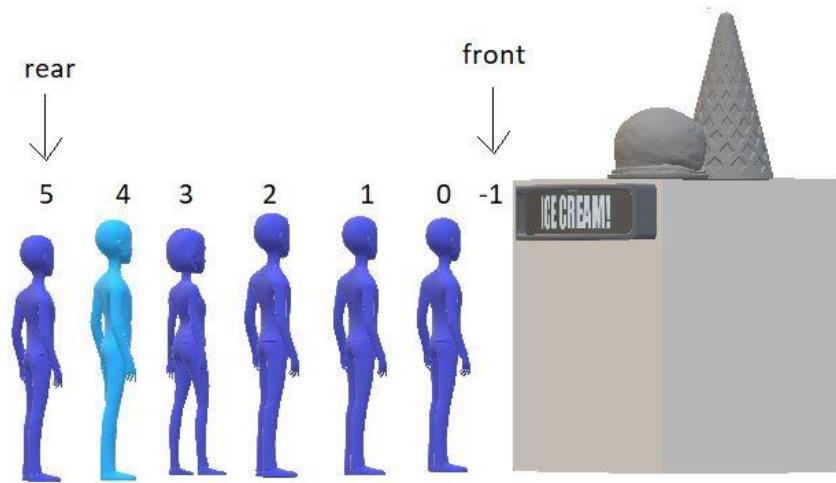


Now, we have 2 places to fill in front as well. And in DEQueue, we have no restrictions. We would just fill our new element at the front and decrease its value by 1. And that would be it. See the results below:

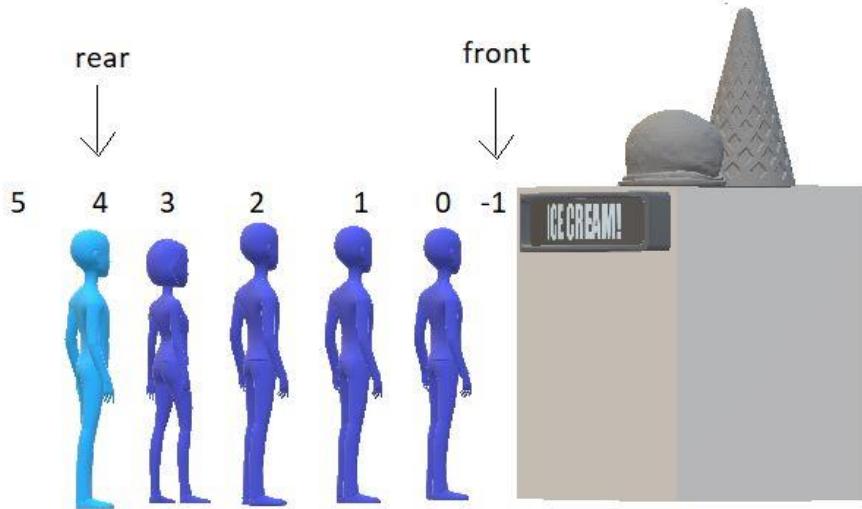


Deletion in a DEQueue:

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



Now, for one moment, think of the rear as the front end. You 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, DEQueues are of two types:

1. Restricted Input DEQueue
2. Restricted Output DEQueue

Restricted Input DEQueue:

Input restricted DEQueues don't allow insertion on the front end. But you can delete from both ends.

Restricted Output DEQueue:

Output restricted DEQueues don't allow deletion from the rear end. But you can perform the insertion on both the ends.

Now the main part is that you would write the program for implementing the Double Ended Queue ADT yourself this time! I know you are capable of doing that. For your convenience, I would like to discuss the ADT part. I would mention all the functionalities one would expect in DEQueues. So, yeah, let's see the DEQueue ADT.

DEQueue ADT:

The data part would be the same as the queue. I wouldn't repeat things.

Refer to the Queue ADT from [here](#).

Methods:

All the operations except the enqueue and dequeue will remain the same as that of the queue. In place of enqueue and dequeue, we would have:

1. enqueueF()
2. enqueueR()
3. dequeueF()
4. dequeueR()

Introduction to Sorting Algorithms

Following our discussion on data structures as queues and linked lists, one of the most important topics, we, now delve into a new one from the arena of algorithms named **sorting**. The session today will just be an introduction. We'll answer a few of the basic questions like, what is sorting? What is it being taught? What are the applications? And many more. So, just hold on and follow the trail.

What is sorting?

Even though you are familiar with sorting, allow me to reiterate the basics. So, sorting is a method to arrange a set of elements in either increasing or decreasing order according to some basis/relationship among the elements. Sorting are of two types, as you could deduce from the definition:

1. Sorting in ascending order, and
2. Sorting in descending order.

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. E.g., the set(1, 9, 2, 8, 7), when sorted in ascending order, becomes (1, 2, 7, 8, 9).

Sorting in descending order:

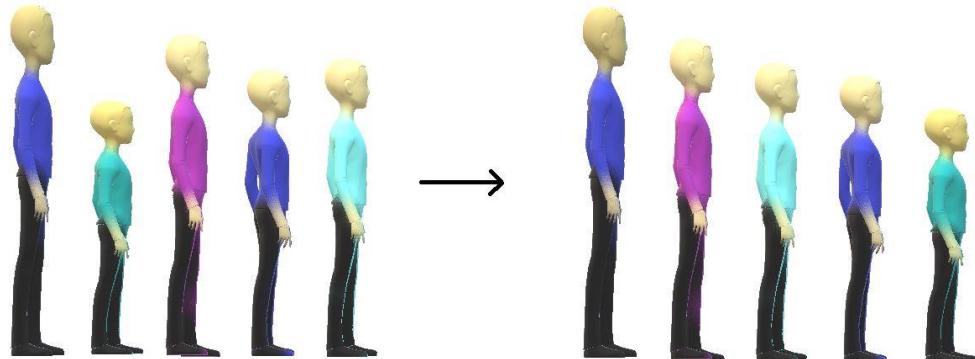
Sorting any set of elements in descending order refers to arranging the elements, let them be numbers, from the largest to the smallest. E.g., the same set(1, 9, 2, 8, 7), when sorted in descending order, becomes (9, 8, 7, 2, 1).

Another question that might cross your mind is why you are being taught this. So, let's explore the need of sorting methods.

Why do we need sorting?

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

1. There are social media applications, news applications, even your emails or file managers, where you want things to be arranged according to dates. You want the newest on top and oldest at the end. And this feature uses the method of sorting. And more specifically, sorting based on the date of publishing/modification.
2. Another example is the product delivery applications, be it delivering food like Swiggy, Zomato, or other shopping applications such as Amazon and Flipkart. You want the top-rated products on the top for your convenience. Sometimes you would need the products to be sorted according to their prices, be it the cheapest at first or the costliest at first. So, every one of these uses the sorting algorithm.
3. The third and most useful application is the dictionary. In a dictionary, the words are sorted lexicographically for you to find any word easily.
4. Another easy concept is that of binary search. If you remember, we discussed in the beginning that searching in a sorted array takes at most $O(\log N)$ time. And when it's not sorted, it can take up to $O(n)$. So, when an array is sorted, it minimizes the effort to find an element. Retrieval becomes much faster.
5. School assembly. If you recall the days of your high school, you stood height-wise during your morning assembly. The basis of sorting here is your height.



Criteria For Analysis of Sorting Algorithms

In the last lecture, we introduced to you the basics of sorting, its definition, its different types, and several examples to make you confident about its applications in real life. Today, in this lesson, you will learn how to come up with criteria for analyzing different sorting algorithms and why one differs from the other.

Before we proceed, make sure you have been through the basics. There are some old concepts, which I'll probably rush through. So, please check out the first 10-12 lectures before jumping to advance.

We will discuss each of the below-mentioned criteria in detail:

1. Time Complexity
2. Space Complexity
3. Stability
4. Internal & External Sorting Algorithms
5. Adaptivity
6. Recursiveness

Time Complexity:

- 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.

Note: Lesser the time complexity, the better is the algorithm.

Space Complexity:

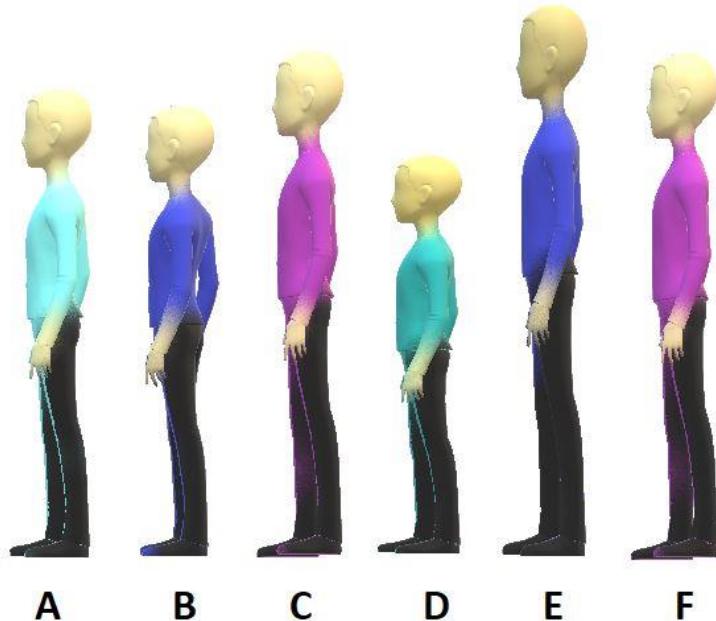
- The space complexity criterion helps us compare the space the algorithm uses to sort any data set. If an algorithm consumes a lot of space for larger inputs, it is considered a poor algorithm for sorting large data sets. In some cases, we might prefer a higher space complexity algorithm if it proposes 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. Inplace sorting algorithms mostly use swapping and rearranging techniques to sort a data set. One example is Bubble Sort (will be covered in the incoming videos).

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 me explain.

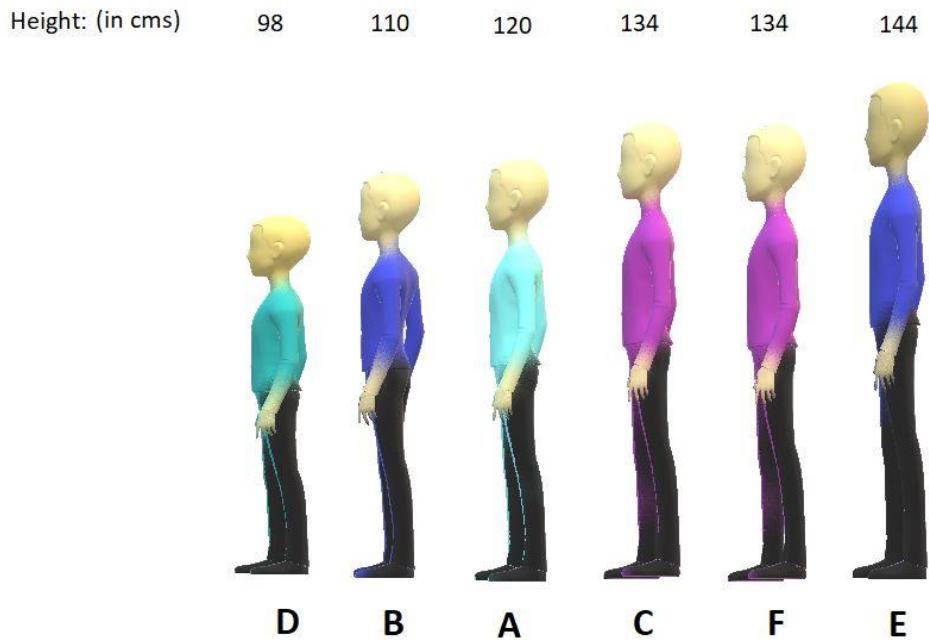
Suppose you have a set of numbers, 6, 1, 2, 7, 6, and we want to sort them in increasing order by using an algorithm. Then the result would be 1, 2, 6, 6, 7. But the key thing to look at is whether the 6s follow the same order as that given in the input or they have changed. That is, whether the first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

Height: (in cms) 120 110 134 98 144 134

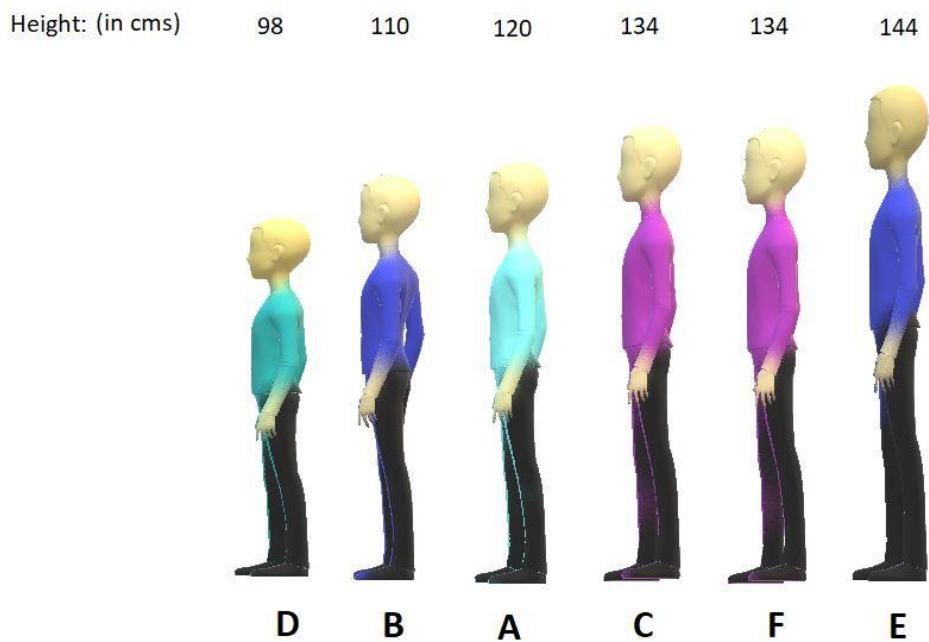


Suppose we called 6 students from a class and made them stand on the first-come basis. And then we measured their heights. And now, we used two different algorithms to assign them a position based on their increasing heights.

Sorting by algorithm A:



Sorting by algorithm B:



Algorithm A is stable, whereas Algorithm B is unstable because algorithm A preserved the order between students C and F having equal heights, and algorithm B couldn't.

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.

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:

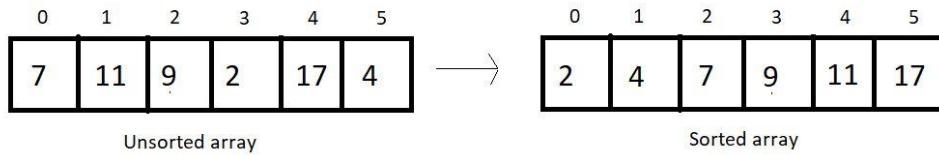
If the algorithm uses recursion to sort a data set, then it is called a recursive algorithm. Otherwise, non-recursive.

And these were the most general criteria to analyze our sorting algorithms. If some of these terminologies were not known to you, I would recommend you first clear these basics.

Bubble Sort Algorithm

In the last tutorial, we discussed different criteria to analyze our sorting algorithms. We made our basis for judging the efficiency of different sorting algorithms for different situations. Today, we are starting all these different sorting algorithms, and we will start with the **Bubble Sort Algorithm**.

Suppose we are given an array of integers and are asked to sort them using the bubble sort algorithm, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. In fact, whichever algorithm you follow, the result would be the same. The below figure shows the same.



The difference lies in the algorithm we follow. With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration. It's important for us to know how that will be pursued.

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. Take a look at the unsorted array above, and I'll walk you through each pass one by one, so you can feel how it gets sorted.

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 you 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.

0	1	2	3	4	5
7	11	9	2	17	4

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

0	1	2	3	4	5
7	11	9	2	17	4

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

0	1	2	3	4	5
7	9	11	2	17	4

Again, we swap the positions of 11 and 2.

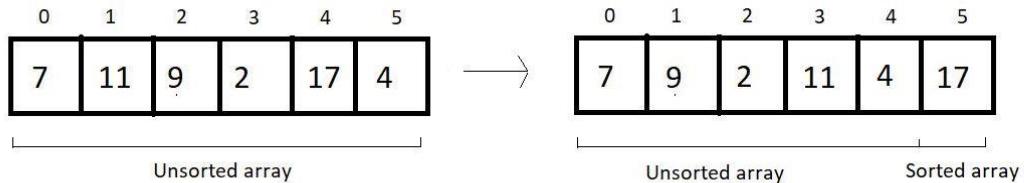
0	1	2	3	4	5
7	9	2	11	17	4

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

0	1	2	3	4	5
7	9	2	11	17	4

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.

0	1	2	3	4	5
7	9	2	11	4	17

No changes to make.

0	1	2	3	4	5
7	9	2	11	4	17

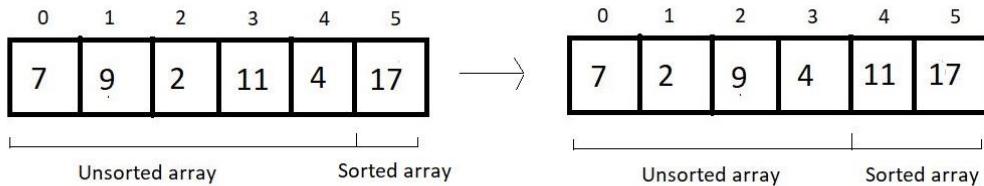
Yes, here we make a swap, since $9 > 2$.

0	1	2	3	4	5
7	2	9	11	4	17

Since $9 < 11$, we move further.

0	1	2	3	4	5
7	2	9	11	4	17

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.

0	1	2	3	4	5
7	2	9	4	11	17

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

0	1	2	3	4	5
2	7	9	4	11	17

We move ahead without making any change.

0	1	2	3	4	5
2	7	9	4	11	17

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

And that was our third pass. And the result at the end was:

0	1	2	3	4	5
2	7	9	4	11	17

4th Pass:

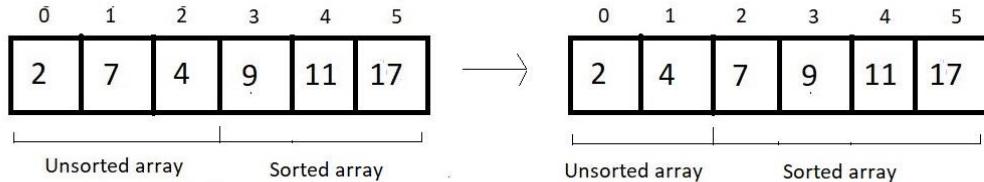
We just have the unsorted part of length 3, and that would cause just 2 comparisons. So, let's see them.

0	1	2	3	4	5
2	7	4	9	11	17

No changes here.

0	1	2	3	4	5
2	7	4	9	11	17

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

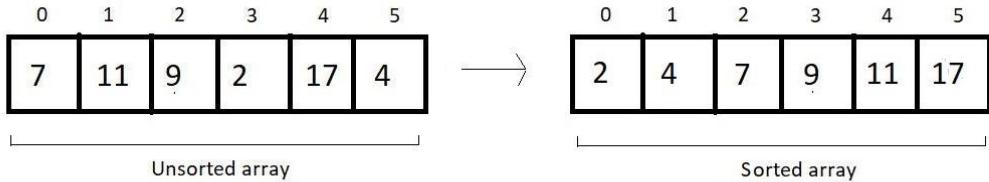


5th (last) pass:

We have only one comparison to make here.

0	1	2	3	4	5
2	4	7	9	11	17

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:

1. If you 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 $(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$ comparison and possible swaps.
2. 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 $O(n^2)$.
3. And if you could observe, we never made a swap when two elements of a pair become equal. Hence the algorithm is a **stable algorithm**.
4. It is not a recursive algorithm since we didn't use recursion here.
5. This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness. Although it can be modified to make it adaptive, it's not adaptive by default. We'll see in the next lecture how it can be made adaptive

Bubble Sort Program in C

In the last lecture, we learnt what bubble sort is and how it is used to sort a linear collection of elements. Towards the end, we drew some conclusions regarding bubble sorting. Before we move on to the programming part, let's review some important notes concerning bubble sort.

1. Time Complexity of the bubble sort algorithm is $O(n^2)$.
2. It is a stable algorithm, because it preserves the order of equal elements.
3. It is not a recursive algorithm.
4. Bubble sort is not adaptive by default, but can be made adaptive by modifying the program. I'll show this part too.

Writing the program for implementing bubble sort is as easy as pie. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. The first step is to define an array of elements, such as integers or characters. I am taking an array of integers.

2. Define an integer variable for storing the size/length of the array.
3. Before we proceed to write the function for Bubble Sort, we would first make a function for displaying the contents of the array.
4. Create a void function *printArray*, and pass the address of the array and its length as its parameters. It doesn't take much to use a for loop to print the array elements. So, we'll skip that.

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

[Copy](#)

Code Snippet 1: Creating the *printArray* function

5. Create another void function *bubbleSort* and pass the address of the array and its length as its parameters. Now, create a *for* loop which would track the number of passes. If you recall, to sort an array of length 6, we made a total of 5 passes, which is obviously, $6-1$. So, for length n , we would make $(n-1)$ passes. So, make this loop run from 0 to $(n-1)$. Inside this loop, make another *for* loop to track the index we are making a comparison at.

Can you decode the limit of this loop? It is obvious that we start from 0, but to which index? In the last lecture, we saw that with each pass, we reduced the size of the unsorted array by 1. In the first pass, we had the size of the unsorted array, 6, hence we made 5 comparisons. And for every subsequent pass, we made 4, 3, 2, and 1 comparison. Let i be the variable to store the pass we are at. Then the number of comparisons for i th pass would be $(n-i)$, where n is the length of the array. Since we started from $i=0$ in the program, it would be $(n-i-1)$ number of comparisons.

Inside this nested *for* loop, check if the j th element of the array is greater than the $(j+1)$ th element. Here, j is the counter variable of the second *for* loop. So, if the j th element of the array is greater than the $(j+1)$ th element, then swap their positions, since we want these to be sorted. Swapping needs you to define a temporary integer variable *temp*. Use it to swap the j th and the $(j+1)$ th element. And the array would itself get sorted. All we had to do was this.

```
void bubbleSort(int *A, int n){
    int temp;
```

```

int isSorted = 0;
for (int i = 0; i < n-1; i++) // For number of pass
{
    printf("Working on pass number %d\n", i+1);
    for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
    {
        if(A[j]>A[j+1]){
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}

```

[Copy](#)

Code Snippet 2: Creating the *bubbleSort* function

Modifying *bubbleSort* to make it adaptive

6. What would an adaptive bubble sort do? Once it detects that our array has already been sorted, it will not perform any more comparisons. So, just a single pass should do the job.

Therefore, the catch here is that the array is already sorted if we didn't have to perform any swapping during any of the passes. This is where we will stop making any more passes.

Create another void function *bubbleSortAdaptive*, and pass the address of the array and its length as its parameters. Create the same two loops, one nested in the other.

First one runs from 0 to $n-1$, and another from 0 to $n-i-1$. We will make an integer variable *isSorted* which would hold 1 if our array is sorted and 0 otherwise.

Make *isSorted* equal to 1 prior to starting any comparison in each pass. If any of our comparisons demands swapping of elements, we switch *isSorted* to 0.

At the end of each pass, check if the *isSorted* changed to 0. If it did, our array was not yet sorted; otherwise, end the comparison there itself, since our array was already sorted.

And this makes our bubble sort algorithm adaptive.

```
void bubbleSortAdaptive(int *A, int n){
```

```

int temp;
int isSorted = 0;
for (int i = 0; i < n-1; i++) // For number of pass
{
    printf("Working on pass number %d\n", i+1);
    isSorted = 1;
    for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
    {
        if(A[j]>A[j+1]){
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
            isSorted = 0;
        }
    }
    if(isSorted){
        return;
    }
}
}

```

[Copy](#)

Code Snippet 3: Creating the *bubbleSortAdaptive* function

Here is the whole source code:

```

#include<stdio.h>

void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
}

```

```

    printf("\n");
}

void bubbleSort(int *A, int n){
    int temp;
    int isSorted = 0;
    for (int i = 0; i < n-1; i++) // For number of pass
    {
        printf("Working on pass number %d\n", i+1);
        for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
        {
            if(A[j]>A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}

```

```

void bubbleSortAdaptive(int *A, int n){
    int temp;
    int isSorted = 0;
    for (int i = 0; i < n-1; i++) // For number of pass
    {
        printf("Working on pass number %d\n", i+1);
        isSorted = 1;
        for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
        {
            if(A[j]>A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}

```

```

        A[ j+1 ] = temp;
        isSorted = 0;
    }
}

if( isSorted ){
    return;
}
}

}

int main(){
// int A[] = {12, 54, 65, 7, 23, 9};
int A[] = {1, 2, 5, 6, 12, 54, 625, 7, 23, 9, 987};
// int A[] = {1, 2, 3, 4, 5, 6};
int n = 11;
printArray(A, n); // Printing the array before sorting
bubbleSort(A, n); // Function to sort the array
printArray(A, n); // Printing the array before sorting
return 0;
}

```

[Copy](#)

Code Snippet 4: Program to implement the Bubble Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 11.

```

int A[] = {1, 2, 5, 6, 12, 54, 625, 7, 23, 9, 987};
int n = 11;
printArray(A, n);
bubbleSort(A, n);
printArray(A, n);

```

[Copy](#)

Code Snippet 5: Using the *bubbleSort* function

And the output we received was:

```
1 2 5 6 12 54 625 7 23 9 987  
Working on pass number 1  
Working on pass number 2  
Working on pass number 3  
Working on pass number 4  
Working on pass number 5  
Working on pass number 6  
Working on pass number 7  
Working on pass number 8  
Working on pass number 9  
Working on pass number 10
```

```
1 2 5 6 7 9 12 23 54 625 987
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

So, our array got sorted, and it made 10 passes as you can see. Let us now put the same array in the adaptive bubble sort function, and see if it still makes 10 comparisons.

```
bubbleSortAdaptive(A, n);  
printArray(A, n);
```

Copy

Code Snippet 6: Using the *bubbleSortAdaptive* function

In fact, it only took one pass to detect it was already sorted.

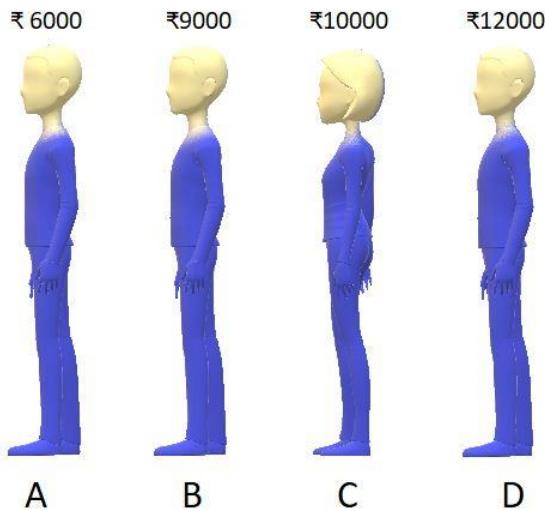
```
Working on pass number 1  
1 2 5 6 7 9 12 23 54 625 987
```

Copy

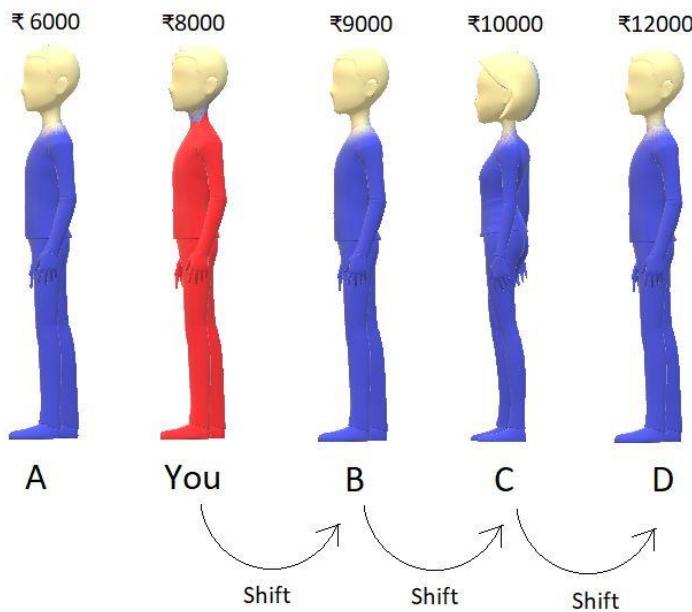
Figure 2: Output of the above program

Insertion Sort Algorithm

In the last lecture, we finished learning the Bubble Sort Algorithm. We learned how to write a program for the same in C language. We saw its characteristics as well. Today, we will learn about a new sorting method, called the Insertion Sort Algorithm. I will make it very intuitive for you to understand. I will use some real-life applications to make the process easy and will steadily dive into the technical part. Suppose you were 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 you suppose you have ₹8000 in your pocket, and you want to be a part of this queue. You don't know where to stand. So, now you start from the last and keep asking the person standing there whether he has more money than you or less money than you. If you find someone with more money, you simply ask him/her to shift backward. And the moment you find a person having less money than you, you stand just behind him/her. So, after doing all this, you find a position in the 2nd place in the queue. The final situation is:



So, this was one of the examples I had in mind. 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. And basically, what did we learn? We learned to insert an element in a sorted array. Although it felt very intuitive to just put yourself in the second position, what would you do if the queue had a thousand people? Not easy, right? And this is where we need a proper algorithm.

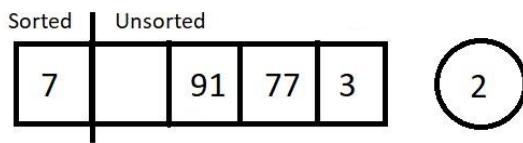
Insert Sort Algorithm:

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

Consider the given array below:

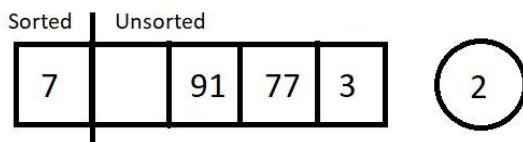
0	1	2	3	4
7	2	91	77	3

And what have we already learned? 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.

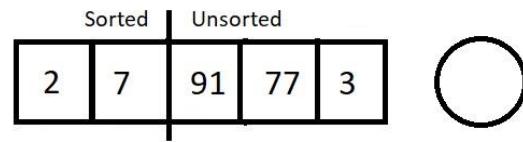


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let's call each of these insertions and the traversal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

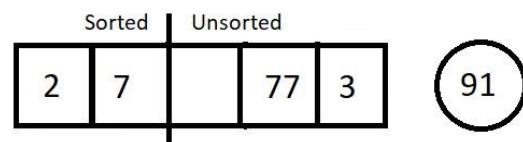
So, let's start with pass 1, which is to insert 2 in the sorted array of length 1.



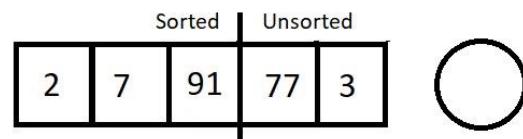
So, we plucked the first element from the unsorted part. Let's insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



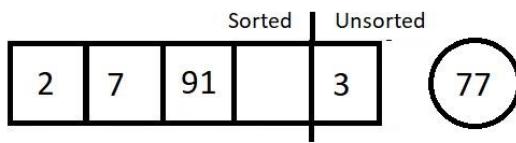
Let's proceed to the next pass.



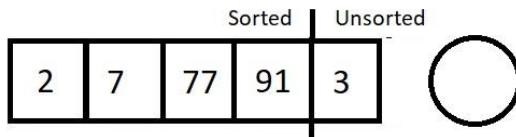
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



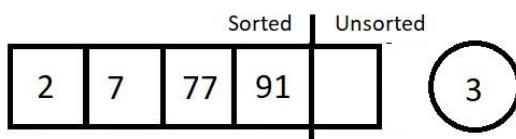
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let's proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



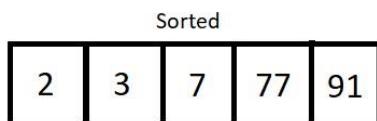
We started checking its best fit, and found the place next to element 7. So this time it would cause just a single shift of element 91.



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 *ith* pass, we would have *i* number of comparisons, and *i* possible swaps.

1. Time Complexity of Insertion Sort Algorithm:

Let's now calculate the time complexity of the algorithm. We made 4 passes for this array of length 5, and for *ith* 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 $1+2+3+4+\dots+(n-1)$ which is $n(n-1)/2$, which ultimately is $O(n^2)$.

2. 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.

3. 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.

And that was all about the insertion sort algorithm. I expect you all to take your own unsorted array, and use the insert sort algorithm this time to sort it. Rather, use both bubble sort and insertion sort, and see if it matches, and tell me which one you found convenient. If something seems unclear, go through the lectures again.

Insertion Sort in C Language (With Explanation)

In the last lecture, we learned what insertion sort is and how it is used to sort an array of elements by using the method of inserting an element in a sorted array. Finally, we enlisted the key characteristics of the algorithm. Before we move on to the programming part, let's review these characteristics of the insertion sort algorithm.

1. Time Complexity of the insertion sort algorithm is $O(n^2)$ in the worst case and $O(n)$ in the best case.
2. It is a stable algorithm since it preserves the order of equal elements.
3. It is not a recursive algorithm.
4. Insertion sort is adaptive by default and no extra effort is needed to make it adaptive. The time complexity itself gets reduced from $O(n^2)$ to $O(n)$ when the algorithm finds an array already sorted.

Having finished revising the insertion sort algorithm, let's now move to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. The first step is to define an array of elements. I am defining an array of integers.
2. Define an integer variable for storing the size/length of the array.
3. Before we proceed to write the function for Insertion Sort, we would first make a function for displaying the contents of the array.
4. Since we did that already in the Bubble Sort lecture, we would skip that here. Rather just copy the function `printArray` from the previous programming lecture.

```
void printArray(int* A, int n){  
    for (int i = 0; i < n; i++)  
    {
```

```

    printf("%d ", A[i]);
}
printf("\n");
}

```

[Copy](#)

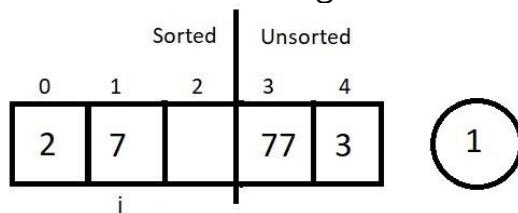
Code Snippet 1: Creating the *printArray* function

5. Create a void function *insertionSort* and pass the address of the array and its length as its parameters. Now, create a *for* loop which would track the number of passes. If you recall, to sort an array of length 5 using insertion sort algorithm, we made a total of 4 passes, which is obviously, $5-1$. So, for an array of length n , we would make $(n-1)$ passes. But this time the loop starts from the 1st index, and not from the 0th since the first element is sorted whatsoever. So, make this loop run from 1 to $(n-1)$. Inside this loop, collect the element at the index i in an integer variable *key*. This *key* is the element we would insert in the sorted subarray.

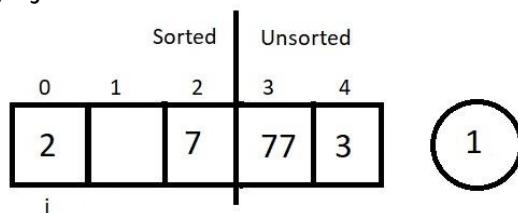
Create another index variable j , which would be used to iterate through the sorted subarray, and to find a perfect position for the *key*. The index variable j holds the value $i-1$.

Make a *while* loop run until either we finish through the sorted subarray and reach the last position, or else we find an index fit for the *key*. And until we come out of the loop, keep shifting the elements to their right and reduce j by 1. And once we come out, insert the *key* at the current value of $j+1$. And this would go on for $n-1$ passes.

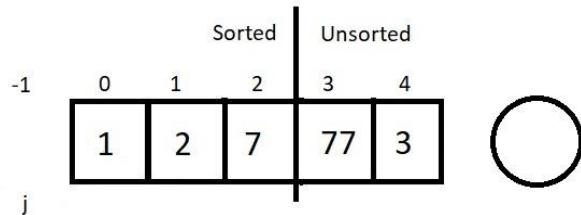
To understand the functioning of the above while loop, follow the illustrations below:



Here, $i=2$, and the element we have at index i is 1. No, this element needs to be given a position in the sorted subarray. So, $j = i - 1$ which is 1. We run a while loop up until j becomes 0 or we find a position for the *key* 1. So, when $j=1$, we check if the element at the j^{th} index is smaller than the *key* or not. Since it's not, we shift it to the right and reduce j by 1.



And now $j=0$, and we have element 2 at the j^{th} index, and it is bigger than the key, hence, we shift even this. And then reducing j makes it equal to -1. So, we stop there itself. And insert the key at $j+1$ which is at 0.



And at the end, we would receive a sorted array. All we had to do was this.

```
void insertionSort(int *A, int n){  
    int key, j;  
    // Loop for passes  
    for (int i = 1; i <= n-1; i++)  
    {  
        key = A[i];  
        j = i-1;  
        // Loop for each pass  
        while(j>=0 && A[j] > key){  
            A[j+1] = A[j];  
            j--;  
        }  
        A[j+1] = key;  
    }  
}
```

[Copy](#)

Code Snippet 2: Creating the *insertionSort* function

Here is the whole source code:

```
#include<stdio.h>  
  
void printArray(int* A, int n){  
    for (int i = 0; i < n; i++)
```

```

    {
        printf("%d ", A[i]);
    }
    printf("\n");
}

void insertionSort(int *A, int n){
    int key, j;
    // Loop for passes
    for (int i = 1; i <= n-1; i++)
    {
        key = A[i];
        j = i-1;
        // Loop for each pass
        while(j>=0 && A[j] > key){
            A[j+1] = A[j];
            j--;
        }
        A[j+1] = key;
    }
}

int main(){
    // -1   0   1   2   3   4   5
    //      12, | 54, 65, 07, 23, 09 --> i=1, key=54, j=0
    //      12, | 54, 65, 07, 23, 09 --> 1st pass done (i=1)!

    //      12, 54, | 65, 07, 23, 09 --> i=2, key=65, j=1
    //      12, 54, | 65, 07, 23, 09 --> 2nd pass done (i=2)!

    //      12, 54, 65, | 07, 23, 09 --> i=3, key=7, j=2
}

```

```
//      12, 54, 65,| 65, 23, 09 --> i=3, key=7, j=1  
//      12, 54, 54,| 65, 23, 09 --> i=3, key=7, j=0  
//      12, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1  
//      07, 12, 54,| 65, 23, 09 --> i=3, key=7, j=-1--> 3rd pass  
done (i=3)!
```

// Fast forwarding and 4th and 5th pass will give:

```
//      07, 12, 54, 65,| 23, 09 --> i=4, key=23, j=3  
//      07, 12, 23, 54,| 65, 09 --> After the 4th pass
```

```
//      07, 12, 23, 54, 65,| 09 --> i=5, key=09, j=4  
//      07, 09, 12, 23, 54, 65| --> After the 5th pass
```

```
int A[] = {12, 54, 65, 7, 23, 9};  
int n = 6;  
printArray(A, n);  
insertionSort(A, n);  
printArray(A, n);  
return 0;  
}
```

Copy

Code Snippet 3: Program to implement the Insertion Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 6.

```
int A[] = {12, 54, 65, 7, 23, 9};  
int n = 6;  
printArray(A, n);  
insertionSort(A, n);  
printArray(A, n);
```

Copy

Code Snippet 4: Using the *insertionSort* function

And the output we received was:

```
12 54 65 7 23 9  
7 9 12 23 54 65
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

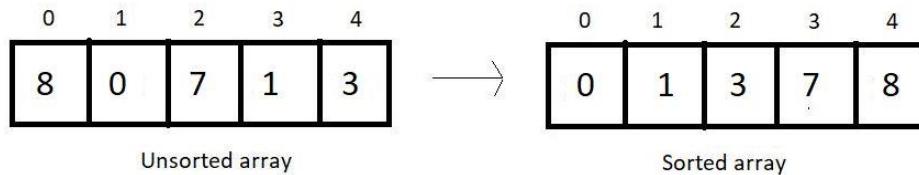
So, our array got sorted. Follow the dry run of the sorting process in the source commented. I have shown each of the 5 passes, and all the comparisons, and the shifting we did.

And, this was all about the insertion sort algorithm. We have finished writing the program for insertion sort, and have analyzed it as well. Once again, we are ready to move on to another sorting algorithm. Keep practicing everything we have learned so far by taking some random array of your own and sorting them using these algorithms. Do use the dry run method to gain confidence.

Selection Sort Algorithm

We have already finished learning about two sorting algorithms so far, the bubble sort algorithm and the insertion sort algorithm. In the last tutorial, we implemented the selection sort algorithm in the C language. Today we are interested in learning a new sorting algorithm called the **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

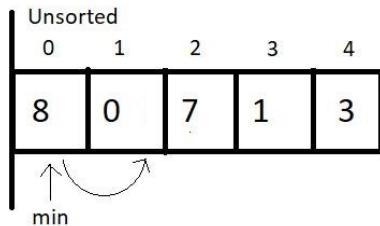
element at the first index of the unsorted subarray. This algorithm reduces the size of the unsorted part by 1 and increases the size of the sorted part by 1 at each respective pass. Let's see how these work. Take a look at the unsorted array above, and I'll walk you through each pass one by one and you will 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.

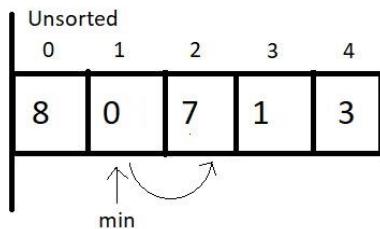
Since our array is of length 5, we will make 4 passes. You must have realized by now the reason why it would take just 4 passes.

1st Pass:

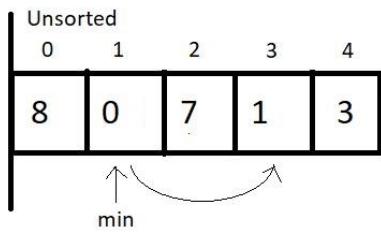
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the *min* index. Now, we'll have to check among the remaining 4 elements if there is still a lesser element than the first one.



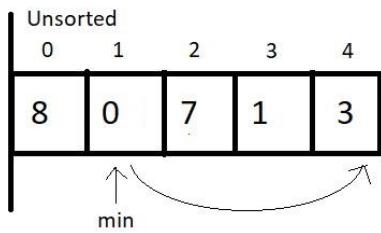
And when we compared the element at *min* index with the element at index 1, we found that 0 is less than 8 and hence we update our *min* index to 1.



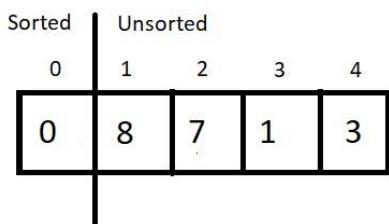
And now we keep checking with the updated *min*. Since 7 is not less than 0, we move ahead.



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.

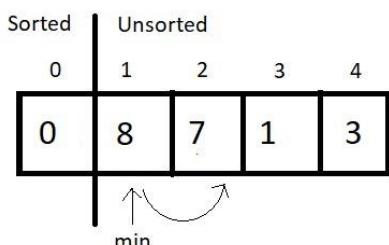


And now we compared the element at the *min* index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at *0th* index with the element at the *min* index. And this gives us our first sorted subarray of size 1. And this is where our first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

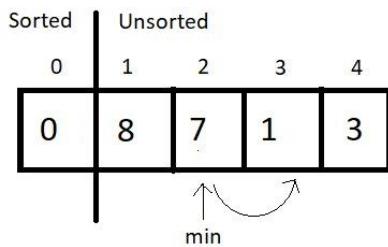


2nd Pass:

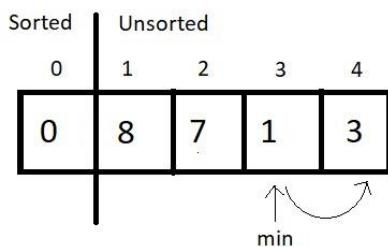
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the *min* index and start iterating to the right for finding the minimum element.



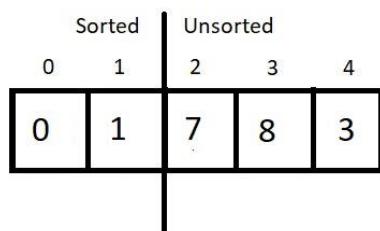
Since 7 is less than 8, we update our *min* index with 2. And move further.



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.

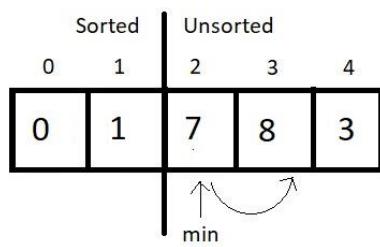


And since 3 is greater than 1, we don't make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at *min* index. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

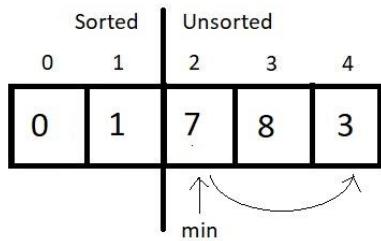


3rd Pass:

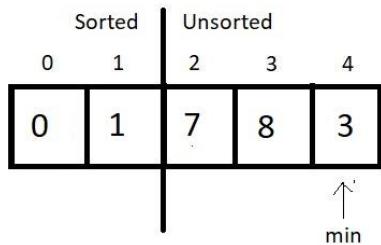
We'll again start from the beginning of the unsorted subarray which is from the index 2, and make the *min* index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.



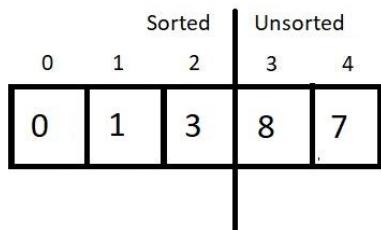
Since 8 is greater than 7, we would make no change, but move ahead.



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.

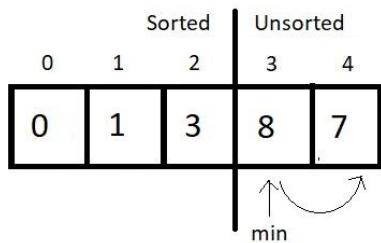


And since that was the last comparison of the third pass, we make a swap of the indices 2 and *min*. And the result at the end would be:

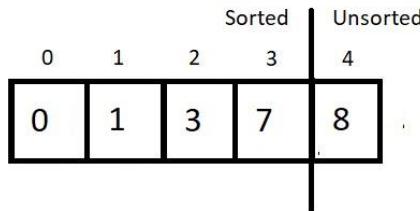


4th Pass:

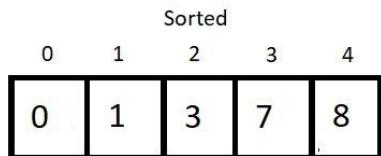
We now have the sorted subarray of length 3, hence the new *min* would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let's see that.



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:

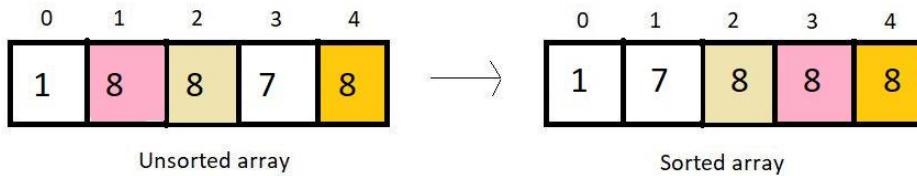
1. Time Complexity of Selection Sort:

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 $(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$ comparisons.

Sum from 1 to $n-1$, we get $\frac{n(n-1)}{2}$, and hence the time complexity of the algorithm would be $O(n^2)$.

2. 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:



3. It is not a recursive algorithm, since we didn't use recursion here.
4. 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.
5. This algorithm offers the benefit of making the least number of swaps to sort an array. We don't make any redundant swaps here.

Selection Sort Program in C

In the last lecture, we learned what selection sort is and how it is used to sort an array of elements by selecting the smallest of all from the unsorted part and replacing it with its final position. Finally, we enlisted our conclusions made after analyzing the algorithm using our defined criteria. Before we move on to the programming part, let's review these characteristics of the selection sort algorithm.

1. The time complexity of the selection sort algorithm is $O(n^2)$ in all its cases.
2. It is not a stable algorithm since it fails to preserve the original order of equal elements. We saw one example the other day.
3. It is not a recursive algorithm.
4. Selection sort is not an adaptive algorithm. It anyways makes comparisons regardless of whether the array given is sorted or not.

That being all that was known about the selection sort algorithm, let us move on to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. First few steps remain the same as we did for the previous two algorithms. The first step is to define an array of elements. We define an array of integers.
2. Define an integer variable for storing the size/length of the array.
3. Before we proceed to write the function for Selection Sort, we would first make a function for displaying the array's content.
4. Since we did that already in previous lectures, we would just copy the function *printArray* from the previous programming lecture.

```
void printArray(int* A, int n){
```

```

for (int i = 0; i < n; i++)
{
    printf("%d ", A[i]);
}
printf("\n");
}

```

[Copy](#)

Code Snippet 1: Creating the *printArray* function

5. Create a void function *selectionSort* and pass the array's address and the array's length as its parameters. Create two integer variables, one for maintaining the *min* index, called the *indexOfMin*, and another for swapping purposes called the *temp*. Now, create a *for* loop that tracks the number of passes. If you recall, to sort an array of length 5 using the selection sort algorithm, we made a total of 4 passes. So, for an array of length *n*, we would make $(n-1)$ passes. And the loop starts from the 0th index and ends at $(n-1)^{th}$.

And if you remember, at each pass, we first initialize the *indexOfMin* to be the first index of the unsorted part. So, inside this loop, initialize the *indexOfMin* to be *i*, which is always the first index of the unsorted part of the array.

Create another loop to iterate over the rest of the elements in the unsorted part to find if there is any lesser element than the one at *indexOfMin*. Make this loop run from *i+1* to the last. And compare the elements at every index. If you find an element at index *j*, which is less than the element at *indexOfMin*, then update *indexOfMin* to *j*.

And finally, when you finish iterating through the second loop, just swap the elements at indices *i* & *indexOfMin*. Swap using the *temp* variable. Follow the same steps at each pass.

And at the end, when you finish iterating through both the *i* and the *j* loops, you would receive a sorted array. All we had to do was this.

```

void selectionSort(int *A, int n){
    int indexOfMin, temp;
    printf("Running Selection sort...\\n");
    for (int i = 0; i < n-1; i++)
    {
        indexOfMin = i;
        for (int j = i+1; j < n; j++)
        {
            if (A[j] < A[indexOfMin])
                indexOfMin = j;
        }
        if (indexOfMin != i)
            swap(A[i], A[indexOfMin]);
    }
}

```

```

    {
        if(A[j] < A[indexOfMin]){
            indexOfMin = j;
        }
    }

    // Swap A[i] and A[indexOfMin]
    temp = A[i];
    A[i] = A[indexOfMin];
    A[indexOfMin] = temp;
}

}

```

[Copy](#)

Code Snippet 2: Creating the *selectionSort* function

Here is the whole source code:

```

#include<stdio.h>

void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}

void selectionSort(int *A, int n){
    int indexOfMin, temp;
    printf("Running Selection sort...\\n");
    for (int i = 0; i < n-1; i++)
    {
        indexOfMin = i;

```

```

        for (int j = i+1; j < n; j++)
    {
        if(A[j] < A[indexOfMin]){
            indexOfMin = j;
        }
    }

    // Swap A[i] and A[indexOfMin]
    temp = A[i];
    A[i] = A[indexOfMin];
    A[indexOfMin] = temp;
}

}

int main(){
    // Input Array (There will be total n-1 passes. 5-1 = 4 in this
    case!)
    // 00 01 02 03 04
    // |03, 05, 02, 13, 12

    // After first pass
    // 00 01 02 03 04
    // 02,|05, 03, 13, 12

    // After second pass
    // 00 01 02 03 04
    // 02, 03,|05, 13, 12

    // After third pass
    // 00 01 02 03 04
    // 02, 03, 05,|13, 12

    // After fourth pass
}

```

```
// 00 01 02 03 04  
// 02, 03, 05, 12, |13
```

```
int A[ ] = {3, 5, 2, 13, 12};  
int n = 5;  
printArray(A, n);  
selectionSort(A, n);  
printArray(A, n);  
  
return 0;  
}
```

Copy

Code Snippet 3: Program to implement the Selection Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 5.

```
int A[ ] = {3, 5, 2, 13, 12};  
int n = 5;  
printArray(A, n);  
selectionSort(A, n);  
printArray(A, n);
```

Copy

Code Snippet 4: Using the *selectionSort* function

And the output we received was:

```
3 5 2 13 12  
Running Selection sort...  
2 3 5 12 13  
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

Visualize each pass individually using the dry run method. This would give you a lot more confidence. The dry run of the above example is there in the source code itself.

QuickSort Algorithm (With Code in C)

Having finished three of the sorting algorithms, our next concern would be to learn the QuickSort algorithm. We have already finished the bubble sort algorithm, the insertion sort algorithm, and the selection sort algorithm. If you have missed any, please check out the previous videos first. Today we are interested in learning a new sorting algorithm called the **QuickSort Algorithm**.

The QuickSort 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 you 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. QuickSort is implemented using both these concepts. And I'll help you master them very soon.

Suppose we are given an array of integers, and we are asked to sort them using the quicksort algorithm, then the very first task you 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.

0	1	2	3	4	5	6	7	8	9
2	4	3	9	1	4	8	7	5	6

Unsorted array

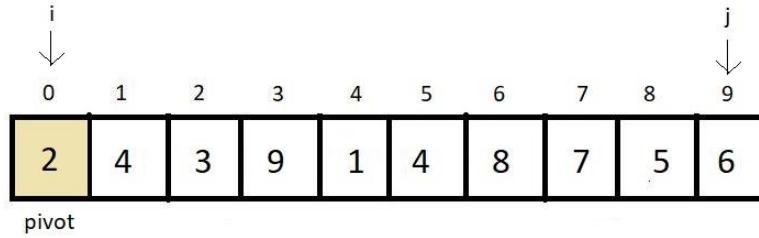
In the quicksort algorithm, every time you get a fresh unsorted subarray, you do a partition on it. Partition asks you to first choose an element as a *pivot*. And as already decided, we would choose the first element of the unsorted subarray as the *pivot*. We would need two more index variables, *i*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

1. 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.
2. Set the *pivot* as the element at the *low* index i since that is the first index of the unsorted subarray.
3. Increase i by 1 until you reach an element greater than the pivot element.
4. Decrease j by 1 until you reach an element smaller than or equal to the pivot element.
5. Having fixed the values of i and j , interchange the elements at indices i and j .
6. Repeat steps 3, 4, and 5 until j becomes less than or equal to i .
7. Finally, swap the pivot element and the element at the index.

This was the partitioning algorithm. Every time you 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.

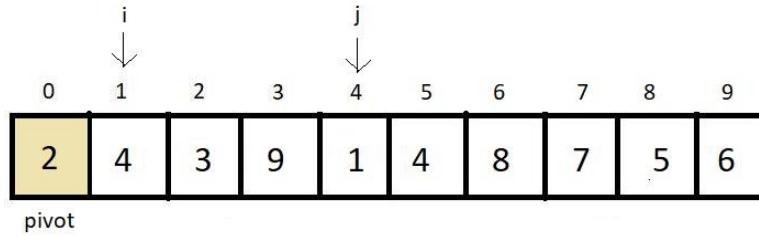
Now let's look at how the array we received at the beginning gets sorted using partitioning and divide and conquer recursively for smaller subarrays.

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 you to follow all the above steps we discussed.



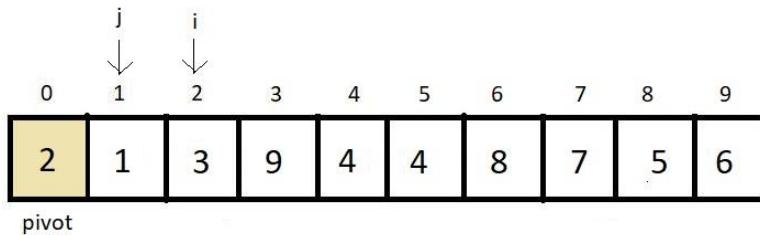
Current Unsorted Subarray

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.

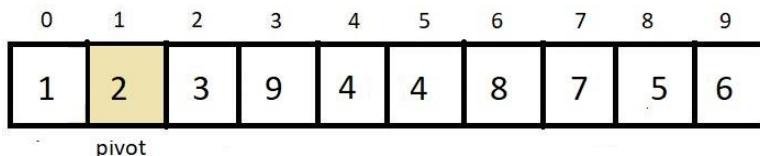


Current Unsorted Subarray

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

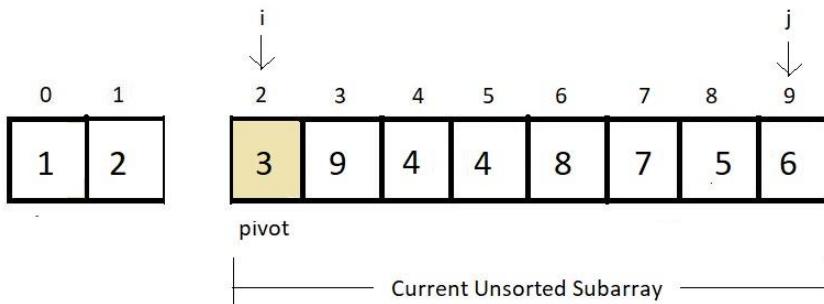


As j crossed i while searching, we followed the final step of swapping the pivot element and the element at j .

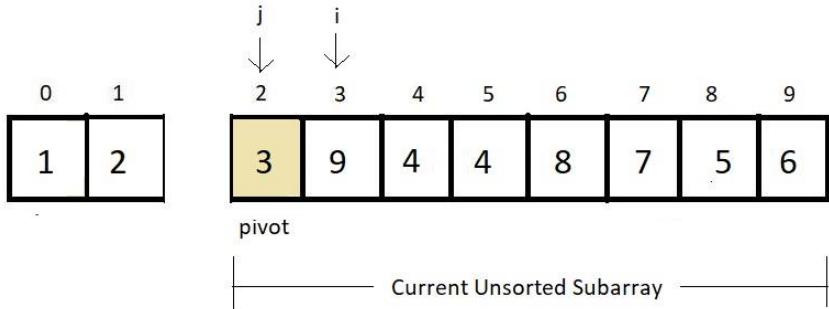


And this would be the final position of the current pivot even in our sorted array. As you can see, all the elements smaller than 2 are on the left, and the rest greater than 2 are on the right. Here comes the role of divide and conquer. We separate our focus from the whole array to just the subarrays, 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.

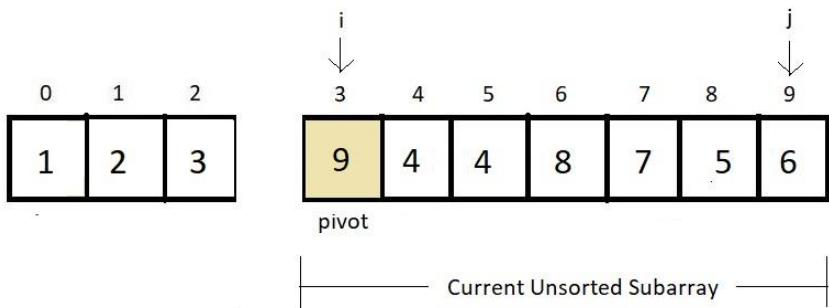
Now since the first subarray has just a single element, we consider it sorted. Let's now sort the second subarray. Follow all the partition steps from the beginning.



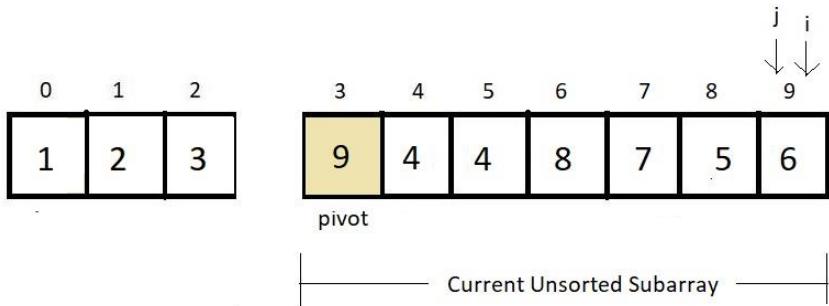
Now, our new pivot is the element at index 2. And i and j are 2 and 9, respectively, marking the start and the end of the subarray. Follow steps 3 and 4.



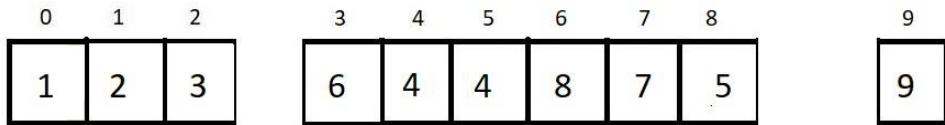
And since there were no elements smaller than 3, j crosses i in the very first iteration. This means 3 was already at its sorted index. And there are no elements to its left; the only unsorted subarray is {9, 4, 4, 8, 7, 5, 6}. And our new situation becomes:



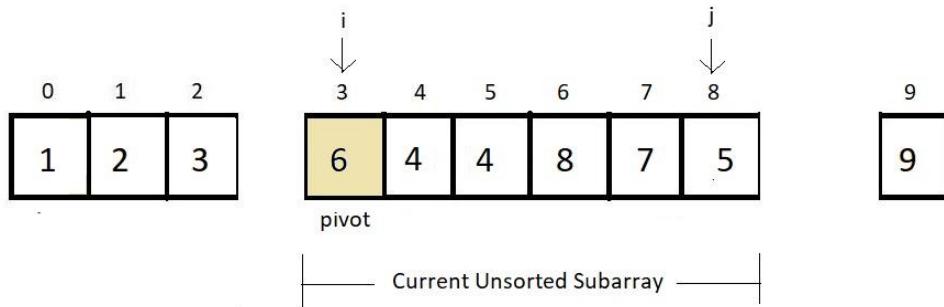
Repeating steps 3 and 4.



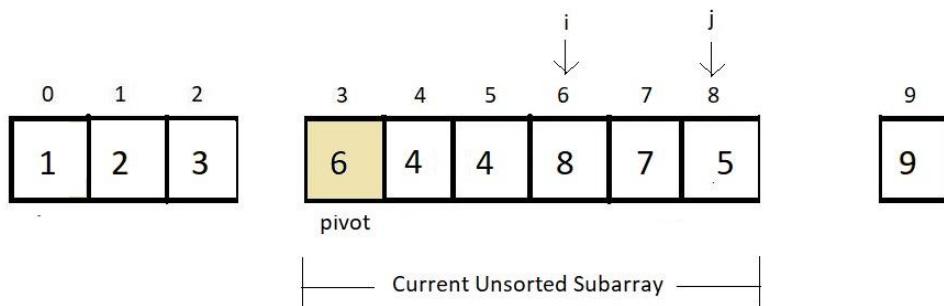
We found that there was no element greater than 9, and hence i reached the last. And 6 was the first element j found to be smaller than 9, and they collided. And this is where we do step 7. Swap the pivot element and the element at index j.



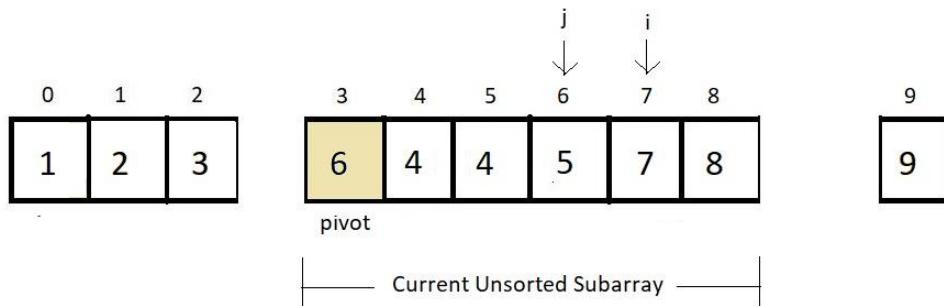
And since there are no elements to the right of element 9, we just have one sorted subarray {6, 4, 4, 8, 7, 5}. Let's call a partition on this as well.



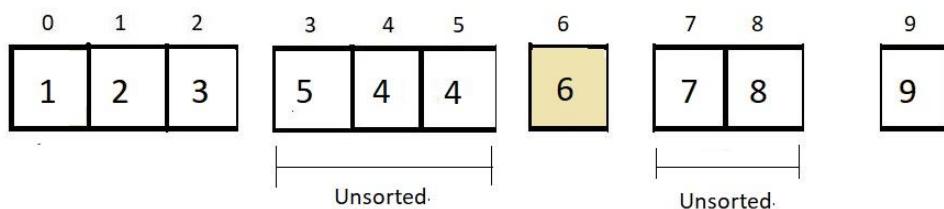
Repeat steps 3 and 4.



And since the condition $j \leq i$ 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.

0	1	2	3	4	5	6	7	8	9
1	2	3	4	4	5	6	7	8	9

Sorted Array

Things may not have made much sense since you are here for the first time. Go through the concepts again. And if you are concerned about the divide and conquer thing, we really don't have to worry about how things will go till the end. Recursions are meant to work like this. Rather we will see the program for implementing the QuickSort algorithm, and things will automatically become clear to you. So, let's just appreciate the tough part and try making it simpler by programming its implementation.

Having finished discussing the functioning of the quick sort algorithm, let's now move to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Before we proceed with the core concepts, let's just copy the `printArray` part in our current programs. This just helps a lot seeing the contents of the array before and after the sorting. Anyways, I have attached the snippet for `printArray` as well.

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

[Copy](#)

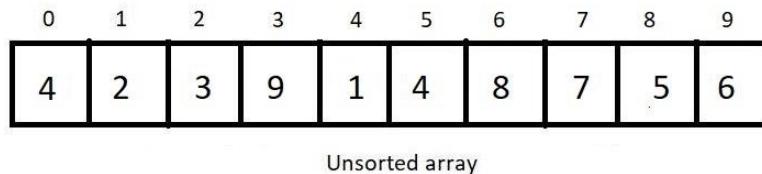
Code Snippet 1: Creating the `printArray` function

2. The next step is to define an array of elements. As always, we define an array of integers.

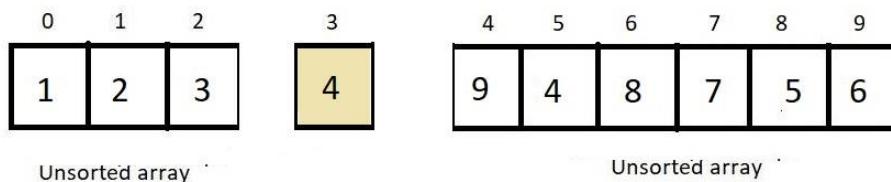
3. Define an integer variable for storing the size/length of the array.

Understanding the `quickSort` function:

4. If you recall what we did every time we were given an unsorted subarray, we just applied a partition on it. Now, since partition is a different job, we will have a different function for that. But the *quicksort* function just intends to follow things to partition. Like, if you pass an array to *quicksort*, it further passes it to the *partition* function, and the partition returns the pivot index after applying all the steps we discussed earlier. *Quicksort* stores this index and recursively calls itself with smaller subarrays which lie on the left and the right of the pivot index.



For example, if you call *quicksort* passing the above array, It would pass it further to the partition, and the partition would return the new index of the pivot element, which is 4. Partition returns 3, the new position of 4. Now, quicksort recursively calls itself on the left and the right subarrays highlighted below.



Creating the quickSort function:

5. Create a void function *quickSort* and pass the address of the array and the lower index, which would be 0 for the first time, and the higher index, which would be *length -1* for the first time, as parameters. Create an integer variable *partitionIndex* for holding the index provided by the *partition*. Now recursively call the *quickSort* function twice but with parameters changed to (*low*, *partitionIndex-1*) for the left subarray and (*partitionIndex+1*, *high*) for the right subarray, instead of just (*low*, *high*). But ain't we forgetting something? The basics of recursion demand a base condition to stop the recursion. Hence, the base condition here would be when our *low* becomes greater than or equal to our *high*. This is when our recursion stops.

```
void quickSort(int A[], int low, int high)
{
```

```

int partitionIndex; // Index of pivot after partition

if (low < high)
{
    partitionIndex = partition(A, low, high);
    quickSort(A, low, partitionIndex - 1); // sort left subarray
    quickSort(A, partitionIndex + 1, high); // sort right subarray
}
}

```

[Copy](#)

Code Snippet 2: Creating the *quickSort* function

Creating the partition function:

6. Create a void function *partition*, and pass the address of the array and the *low* and the *high* of the subarray as parameters. Create an integer variable *pivot* that takes the element at the low index. Create two index variables, *i* and *j*, and make them hold *low+1* and *high*.

Create a while loop and run until the index *i* reaches an element greater than or equal to the pivot or the array finishes. Till then, keep increasing *i* by 1. Similarly, create another while loop and run until our index *j* reaches an element smaller than the pivot or the array finishes. Till then, keep decreasing *j* by 1. After finishing all the above tasks, we swap the elements at indices *i* and *j*.

The above process is repeated using a do-while loop until *i* becomes greater than *j*. And when it does, the loop breaks, and before we return, we swap our pivot with the element at index *j*. And that should finish our job.

```

int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
    int temp;

    do
    {

```

```

        while (A[i] <= pivot)
    {
        i++;
    }

        while (A[j] > pivot)
    {
        j--;
    }

    if (i < j)
    {
        temp = A[i];
        A[i] = A[j];
        A[j] = temp;
    }
} while (i < j);

// Swap A[low] and A[j]
temp = A[low];
A[low] = A[j];
A[j] = temp;
return j;
}

```

[Copy](#)

Code Snippet 3: Creating the *partition* function

Here is the whole source code:

```

#include <stdio.h>

void printArray(int *A, int n)

```

```
{  
    for (int i = 0; i < n; i++)  
    {  
        printf("%d ", A[i]);  
    }  
    printf("\n");  
}
```

```
int partition(int A[], int low, int high)
```

```
{  
    int pivot = A[low];  
    int i = low + 1;  
    int j = high;  
    int temp;
```

```
    do
```

```
    {
```

```
        while (A[i] <= pivot)
```

```
        {
```

```
            i++;
```

```
        }
```

```
        while (A[j] > pivot)
```

```
        {
```

```
            j--;
```

```
        }
```

```
        if (i < j)
```

```
        {
```

```
            temp = A[i];
```

```
            A[i] = A[j];
```

```

        A[j] = temp;
    }
} while (i < j);

// Swap A[low] and A[j]
temp = A[low];
A[low] = A[j];
A[j] = temp;
return j;
}

void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1); // sort left subarray
        quickSort(A, partitionIndex + 1, high); // sort right subarray
    }
}

int main()
{
    //int A[] = {3, 5, 2, 13, 12, 3, 2, 13, 45};
    int A[] = {9, 4, 4, 8, 7, 5, 6};
    // 3, 5, 2, 13, 12, 3, 2, 13, 45
    // 3, 2, 2, 13i, 12, 3j, 5, 13, 45
    // 3, 2, 2, 3j, 12i, 13, 5, 13, 45 --> first call to partition
    returns 3
    int n = 9;
}

```

```
n =7;  
printArray(A, n);  
quickSort(A, 0, n - 1);  
printArray(A, n);  
return 0;  
}
```

Copy

Code Snippet 4: Program to implement the Quick Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 7.

```
int A[ ] = {9, 4, 4, 8, 7, 5, 6};  
int n = 7;  
printArray(A, n);  
quickSort(A, 0, n-1);  
printArray(A, n);
```

Copy

Code Snippet 5: Using the *quickSort* function

And the output we received was:

```
9 4 4 8 7 5 6  
4 4 5 6 7 8 9
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

So, our array got sorted. Follow the dry run we did in the comments in the source code. Practice sorting small arrays using the partition method on your own, and you'll feel confident about it.

Analysis of QuickSort Sorting Algorithm

In the last video, we learned to use the quick sort algorithm. We saw how we used the methods of divide and conquer and partitioning to achieve sorting. Later, in the video, we even saw the program to implement all these methods to sort an array using the quick sort algorithm. Today, we'll see the analysis of the quicksort algorithm on all criteria we defined earlier.

1. Time Complexity:

Let's start with the runtime complexity of the 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.

0	1	2	3	4
1	2	4	8	12

Unsorted array

In the first step, you 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.

0	1	2	3	4
1	2	4	8	12

Unsorted array

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.

0	1		2	3	4
1	2		4	8	12

Unsorted array

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

0	1	2		3	4
1	2	4		8	12

Unsorted array

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,

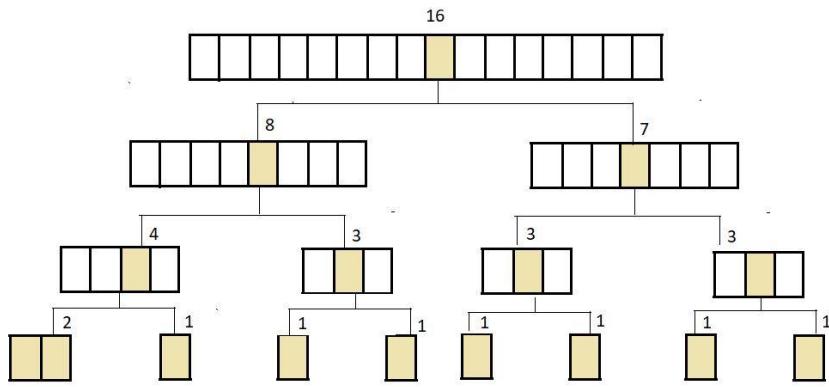
0	1	2	3	4
1	2	4	8	12

Sorted Array

So, if you 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 $O(n^2)$. This is our worst-case complexity.

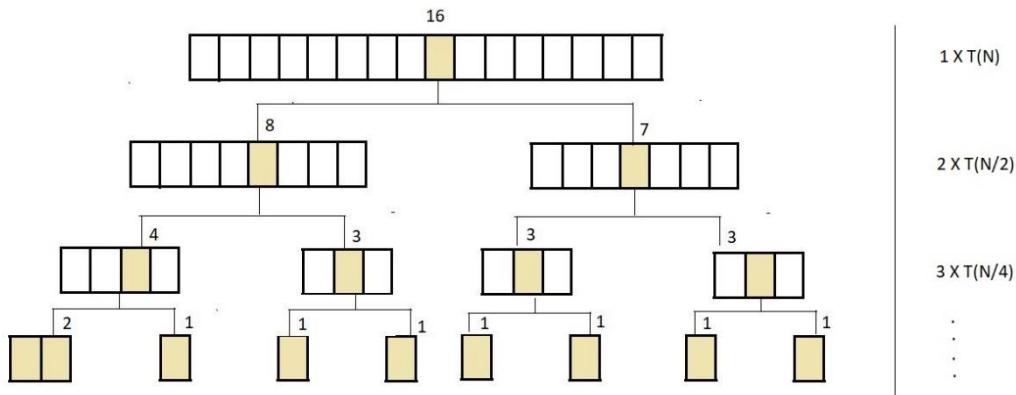
Best Case:

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 = $1(n) + 2(n/2) + 4(n/4) + \dots + \text{until the height of the tree}(h)$

Total time = $n * h$

Now, can you decide what h is? H is the height of the tree, and the height of the tree, if you remember, is $\log_2(n)$, where n is the size of the given array. In the above example, $h = 4$, since $\log_2(16)$ equals 4. Hence the time complexity of the algorithm in its best case is **$O(n\log n)$** .

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

2. Stability:

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

0	1	2	3	4
2	8	9	12	2

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

0	1	2	3	4
2	2	9	12	8

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

3. 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.
4. There is no hard and fast rule to choose only the first element as the pivot; rather, you can have any random element as its pivot using the rand() function and that you wouldn't believe actually reduces the algorithm's complexity.

MergeSort Sorting Algorithm

We have so far covered all our sorting algorithms except one or two. We learned about the bubble sort, the insertion sort, the selection sort, and the quicksort. Now it's time to move onto our next sorting algorithm, the merge sort algorithm. You will understand it very easily once I explain the working of the algorithm using a few intuitive examples to you.

But before we proceed, I would like to give you the reason why we call it the **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. I'll show you first how merging two sorted subarrays works.

Merging Procedure:

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

0	1	2	3	4
7	9	18	19	22

Sorted array A

0	1	2	3
1	6	9	11

Sorted Array B

1. 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.
2. 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.

i	0	1	2	3	4
	7	9	18	19	22

Sorted array A

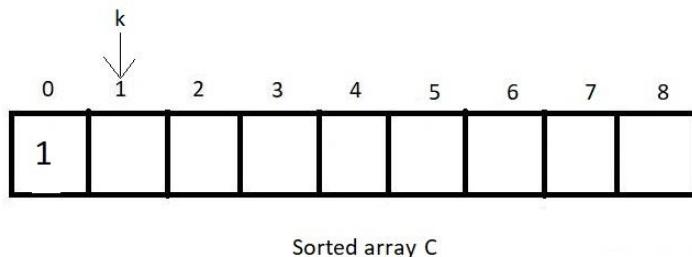
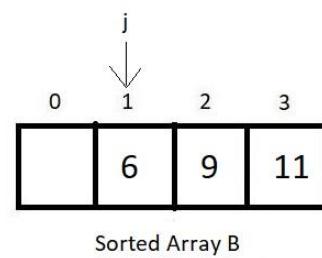
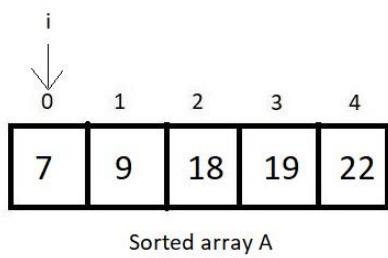
j	0	1	2	3
	1	6	9	11

Sorted Array B

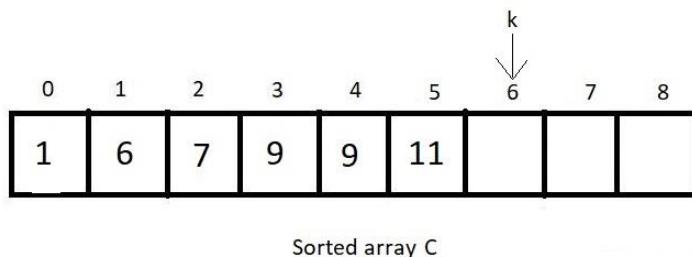
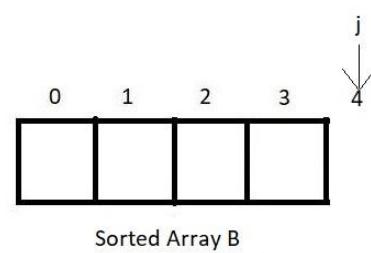
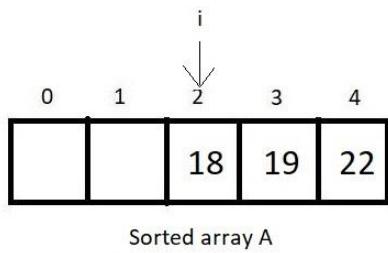
k	0	1	2	3	4	5	6	7	8

Sorted array C

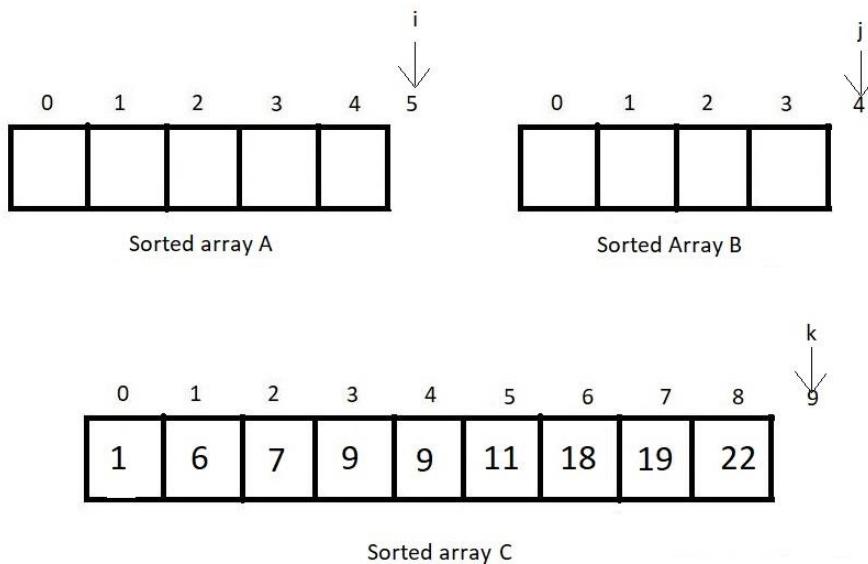
3. Initially, all i, j, and k are equal to 0.
4. Now, we compare the elements at index i of array A and index j of array B and see which one is smaller. Fill in the smaller element at index k of array C and increment k by 1. Also, increment the index variable of the array we fetched the smaller element from.
5. Here, $A[i]$ is greater than $B[j]$. Hence we fill $C[k]$ with $B[j]$ and increase k and j by 1.



6. 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.



I hope you understood the merging procedure. This is an important concept in learning the merge sort algorithm. Be sure not to skip this. Even the programming part of the merge procedure is not that tough. You just follow these steps:

1. 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.
2. Create three index variables i, j & k . And initialize all of them with 0.
3. And then run a while loop with the condition that both the index variables i and j don't exceed their respective array limits.
4. 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.
5. And when the loop finishes, either array A or B or both get finished. And now you run two while loops for each array A and B, and insert all the remaining elements as they are in the array C. And you 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++;
    }
}

```

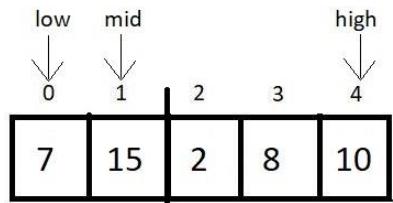
Now, this would quite not be our situation when sorting an array using the merge sort. We wouldn't have two different arrays A and B, rather a single array having two sorted subarrays. Now, I'd show you how to merge two sorted subarrays of a single array in the array itself.

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

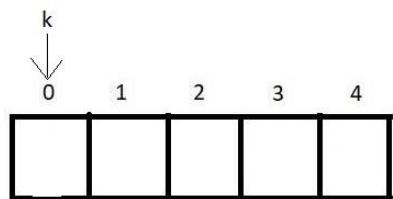
0	1	2	3	4
7	15	2	8	10

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*.

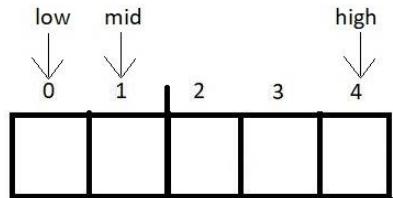


Unsorted Array A

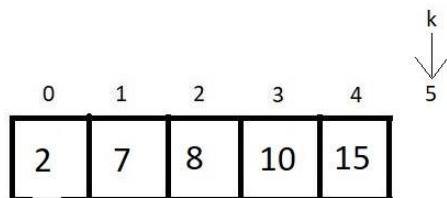


Sorted array B

Previously we had index variables *i*, *j*, and *k* initialised with 0 of their respective arrays. But here, *i* gets initialised with *low*, *j* gets initialised with *mid+1*, and *k* gets initialised 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.



Unsorted Array A



Sorted array B

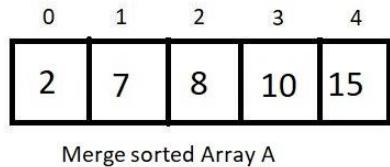
After all the elements get filled in array C, we revert back to our original array A and fill the sorted elements again from low to high, making our array merge-sorted.

```

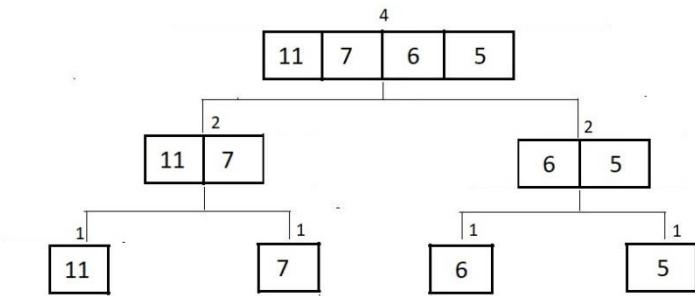
void merge(int A[], mid, low, high)
{
    int i, j, k, B[high+1];
    i = low;
    j = mid + 1;
    k = low;
    while (i <= mid && j <= high){
        if (A[i] < A[j]){
            B[k] = A[i];
            i++;
            k++;
        }
        else{
            B[k] = A[j];
            j++;
            k++;
        }
    }
    while (i <= mid){
        B[k] = A[i];
        k++;
        i++;
    }
    while (j <= high){
        B[k] = A[j];
        k++;
        j++;
    }
    for (int i = low; i <= high; i++)
        A[i] = B[i];
}

```

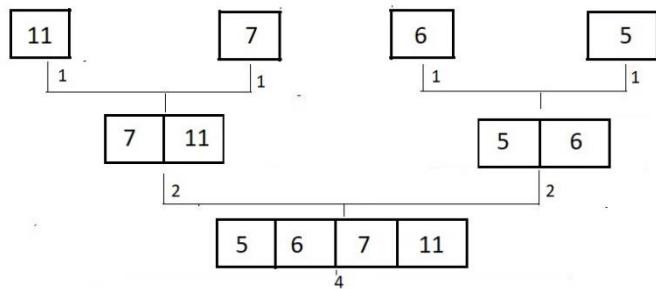
There were few changes we had to make in the pseudocode.



So that was our merging segment. That was the best I could do to make things clear to you. I hope you all understood everything I said. But things have not finished yet. This was just the merging part yet the core of the lecture. It's just the easy part left. Whenever you receive an unsorted array, you 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);
    }
}

```

MergeSort Source Code in C (Helpful Explanation)

In the last tutorial, we deeply covered the concepts behind the merge sort algorithm. We saw its implementation as well via some pseudocodes. We implemented the merge sort algorithm to sort a few arrays. I hope you did learn everything till here. If you couldn't, I recommend first going through the last lecture before proceeding to the programming part. Today, we'll solely look at the programming part of the merge sort algorithm in C.

Before we move on to the programming part, let's revise a few things

1. Whenever you are asked to sort an array using the merge sort algorithm, first divide the array until the size of each subarray becomes 1.
2. Now, since an array or subarray of size 1 is considered already sorted, we call our merge function, which merges these subarrays into bigger sorted subarrays.
3. And finally, you end up with your array fully sorted. Voila!

I will use an example from the last lecture to illustrate points 1 and 2.

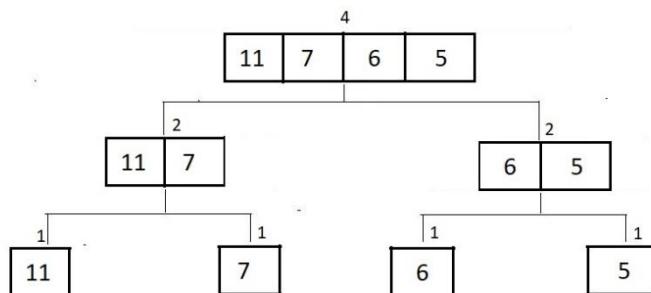


Figure: Point-1

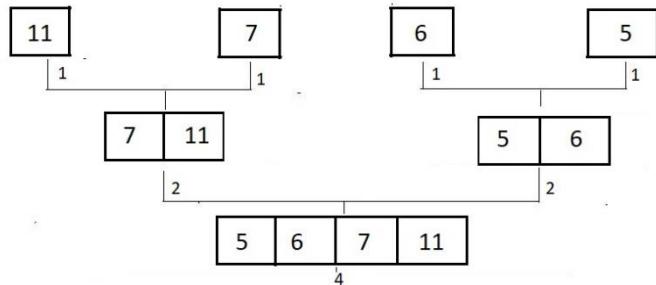


Figure: Point-2

That being all that we did yesterday, let us now move on to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Before we proceed with the functions `merge` and `mergeSort`, let's copy the `printArray` part in our current programs. Copying would save us some time. Having a print function helps a lot seeing the contents of the array before and after the sorting. Anyways, I have attached the snippet for `printArray` as well.

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

[Copy](#)

Code Snippet 1: Creating the `printArray` function

2. The next step is to define an array of elements. I'd rather copy that from our previous lecture. Then define an integer variable for storing the size/length of the array.

Creating the `merge` function:

3. This is just the `merge` function, whose only job is to merge two sorted arrays into a bigger sorted array. I'd recommend keeping the pseudo code with you for better understanding. Create a void function `merge` that takes the array and the integer indices `low, mid, and high` as its parameters. Create integer variables `i, j, and k` for

iterating through the array A and an auxiliary array B. Create this integer array B of size, but for now, we would choose some larger size, say 100.

Initialize *i* with *low*, *j* with *mid+1*, and *k* with *low*. Here, *i* marks the current element of the first subarray of array A, and *j* marks the first element of the second subarray. And, *k* is the iterator for array B to insert the smaller of elements at indices *i* and *j*. Run a while loop until either *i or j* or both reaches the threshold of their corresponding subarray. Inside the loop, see if the element at index *i* is smaller than the one at index *j*. If it is, insert element at index *i* in index *k* of array B i.e., *B[k] = A[i]* and increment both *i* and *k* by 1, else, insert element at index *j* in index *k* of array B i.e. *B[k] = A[j]* and increment both *j* and *k* by 1.

The above ends when either *i or j* or both reach its corresponding subarray's end. Now, run two separate while loops for inserting the remaining elements, if left, in both the subarrays. And this would finish filling all the elements in sorted order in array B. The only thing left is just to copy the sorted array back again to array A. And we are done.

```
void merge(int A[], int mid, int low, int high)
{
    int i, j, k, B[100];
    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)
{
    B[k] = A[i];
    k++;
    i++;
}

while (j <= high)
{
    B[k] = A[j];
    k++;
    j++;
}

for (int i = low; i <= high; i++)
{
    A[i] = B[i];
}
}

```

[Copy](#)

Code Snippet 2: Creating the *merge* function

Creating the *mergeSort* function:

4. Create a void function *mergeSort* and pass the address of the array and the index variables *low* and *high* as its parameters. Here, the lower index would be 0 for the first time, and the higher index would be *length -1* for the first time.

We would recursively call this function only if *low* is less than *high*; that is, there are at least two elements in the subarray. Otherwise, we break off from the loop.

Create an integer variable *mid* for holding the index of the mid element, which would be. Now recursively call the *mergeSort* function twice but with parameters changed to (*low*, *mid-1*) for the left subarray and (*mid+1*, *high*) for the right subarray. Applying *mergeSort* sorts the left half and the right half separately. This is where we would merge them back in the array. Call the *merge* function and pass the array, its index variables *low*, *mid*, and *high*. And this would return a sorted array.

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

[Copy](#)

Code Snippet 3: Creating the *mergeSort* function

Here is the whole source code:

```
#include <stdio.h>  
  
void printArray(int *A, int n)  
{  
    for (int i = 0; i < n; i++)  
    {  
        printf("%d ", A[i]);  
    }  
    printf("\n");  
}  
  
void merge(int A[], int mid, int low, int high)  
{  
    int i, j, k, B[100];  
    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)
{
    B[k] = A[i];
    k++;
    i++;
}
while (j <= high)
{
    B[k] = A[j];
    k++;
    j++;
}
for (int i = low; i <= high; i++)
{
    A[i] = B[i];
```

```
}
```

```
}
```

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

```
int main()  
{  
    // int A[] = {9, 14, 4, 8, 7, 5, 6};  
    int A[] = {9, 1, 4, 14, 4, 15, 6};  
    int n = 7;  
    printArray(A, n);  
    mergeSort(A, 0, 6);  
    printArray(A, n);  
    return 0;  
}
```

Copy

Code Snippet 4: Program to implement the Merge Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 7.

```
int A[] = {9, 1, 4, 14, 4, 15, 6};  
int n = 7;  
printArray(A, n);
```

```
mergeSort(A, 0, 6);  
printArray(A, n);
```

Copy

Code Snippet 5: Using the *mergeSort* function

And the output we received was:

```
9 1 4 14 4 15 6  
1 4 4 6 9 14 15
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 1: Output of the above program

So, our array got sorted. Practice sorting your own arrays using the merge sort method, and you'll feel confident about it. It's always possible to watch a segment again if you missed something. Make sure you understand everything.

Count Sort Algorithm

In the last lecture, we finished learning the merge sort algorithm and its implementation in C. Finally, we have moved to our last sorting algorithm. I know things must have felt tough at times, but believe me, you have finished learning a few of the most important algorithms of all. Today, we'll start with 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. We will discuss constraints later, which would make you wonder why we don't use just this. We will do all the analysis, but before that, let's see what count sort is. The below figure shows the array given.

0	1	2	3	4	5	6
3	1	9	7	1	2	4

Unsorted array

1. The algorithm first asks you 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 no. of occurrences of some number in the given array. We will have to initialize all count array elements with 0 for that to work.
2. 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, you 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.
3. Now traverse through the count array, and look for the nonzero elements. The moment you 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 you fill it in the resultant array. And then move further. This way, you 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.

↓
i

0	1	2	3	4	5	6
3	1	9	7	1	2	4

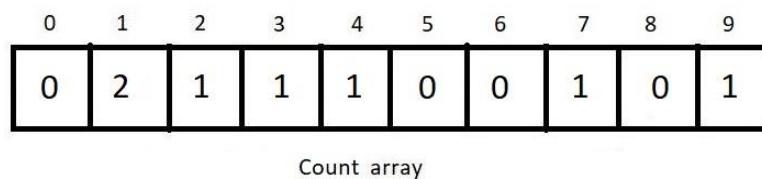
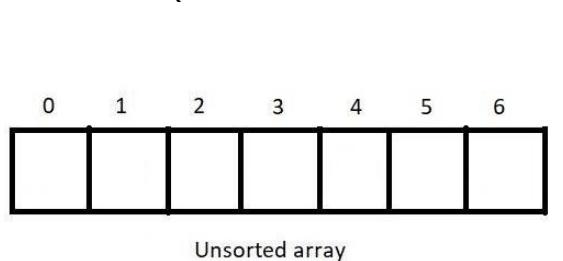
Max = 9

Unsorted array

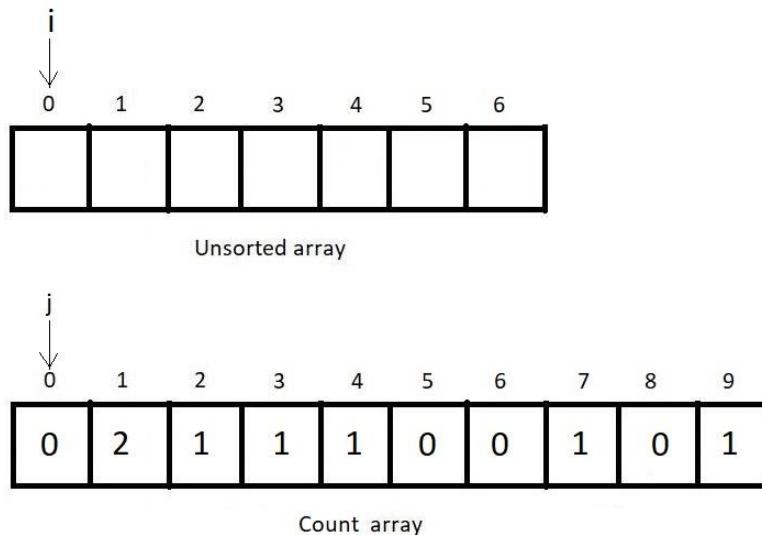
0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Count array

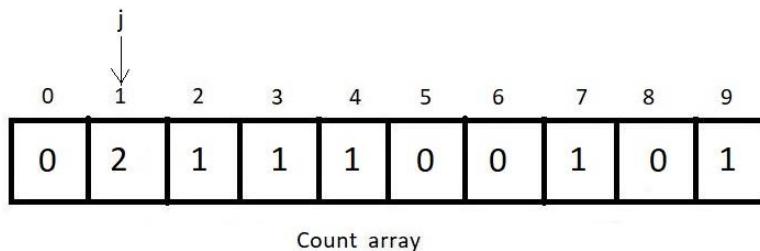
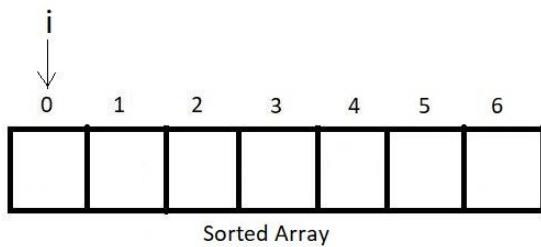
Now, let's iterate through the given array and count the no. of occurrences of each number less than equal to the maximum element.



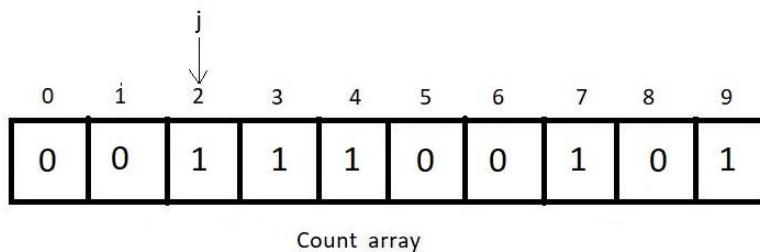
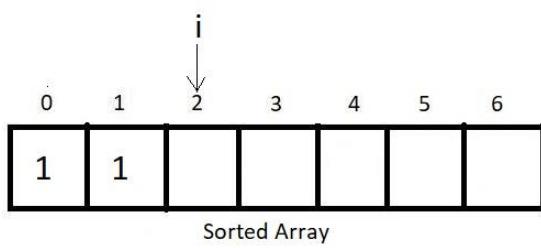
We would iterate through the count array and fill the unsorted array with the index we encounter having a non-zero number of occurrences.



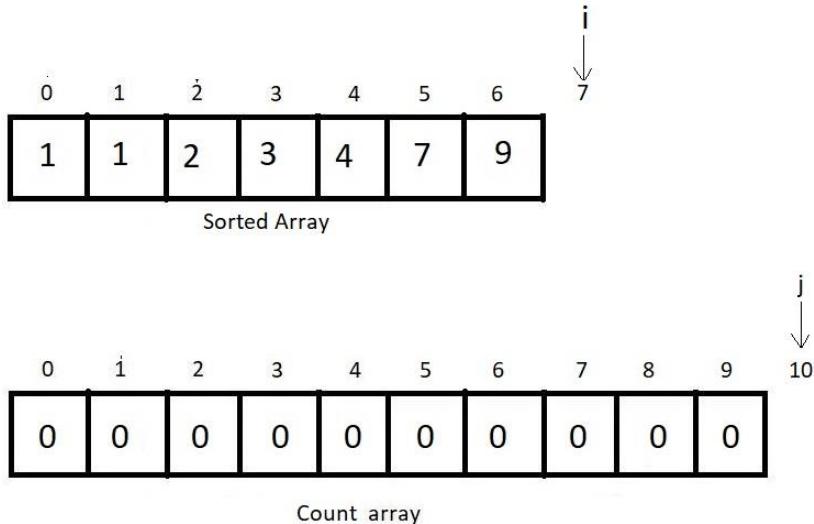
Now since there are zero numbers of zeros in the given array, we move further to index 1.



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.



And it was this easy to sort the array. Tell me you found it as easy as I promised in the beginning. Having finished discussing the functioning of the count sort algorithm, let's now move to the programming part. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Before we proceed with the actual function related to count sort, let's copy the *printArray* and the array declaration part from the previous lecture in our current program as well. This saves us time and helps us a lot to see the contents of the array before and after the sorting. I have attached the snippet for *printArray*

```
void printArray(int* A, int n){
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

[Copy](#)

Code Snippet 1: Creating the *printArray* function

2. Now, to proceed with the *countSort* function, we would need a *maximum* function, which would return the maximum of all elements in the array given.

Creating the maximum function:

3. Create an integer function `maximum` and pass the array and its length as its parameters. Create an integer variable `max` to store the maximum element. Initialize this `max` with the least possible number we have, which is 0. To use this identifier, you must include `<limits.h>`. Now iterate through the whole array using a for loop, and if you find an element greater than the current `max`, update `max`. At the end, return `max`.

```
int maximum(int A[], int n){  
    int max = INT_MIN;  
    for (int i = 0; i < n; i++)  
    {  
        if (max < A[i])  
        {  
            max = A[i];  
        }  
    }  
    return max;  
}
```

[Copy](#)

Code Snippet 2: Creating the `maximum` function

Creating the `countSort` function:

4. Create a void function `countSort` and pass the array and its length as its parameters. Create an integer variable `max` to store the maximum element which you get by calling the `maximum` function we made above. Next, create an integer array `count` and assign it memory dynamically using `malloc` of the size `max+1`. Don't forget to include `<stdlib.h>` to be able to use `malloc`.

Initialize the whole `count` array by simply using a for a loop.

Run another for loop to iterate through the given array and increase the value of the corresponding element index in the `count` array by 1.

5. Now, since the count array has been populated, create two index variables, `i` and `j`, to iterate through the count and the given array, respectively. Run a while loop until we reach the end of the count array. Inside the loop, check if the element at the current index in the count array is non-zero. If it is, insert `i` at the `jth` index of the given array and decrement the element in the count array at `ith` index by 1 and simultaneously increase the value of `j` by 1. Repeat this until the element at the `ith` index becomes zero or if it is already zero, increase `i` by 1.

The array becomes sorted once all the processes listed above are complete.

```
void countSort(int * A, int n){  
    int i, j;  
    // Find the maximum element in A  
    int max = maximum(A, n);  
  
    // Create the count array  
    int* count = (int *) malloc((max+1)*sizeof(int));  
  
    // Initialize the array elements to 0  
    for (i = 0; i < max+1; i++)  
    {  
        count[i] = 0;  
    }  
  
    // Increment the corresponding index in the count array  
    for (i = 0; i < n; i++)  
    {  
        count[A[i]] = count[A[i]] + 1;  
    }  
  
    i = 0; // counter for count array  
    j = 0; // counter for given array A  
  
    while(i<= max){  
        if(count[i]>0){  
            A[j] = i;  
            count[i] = count[i] - 1;  
            j++;  
        }  
        else{  
            i++;  
        }  
    }  
}
```

```
    }
}
}
```

[Copy](#)

Code Snippet 3: Creating the *countSort* function

Here is the whole source code:

```
#include<stdio.h>
#include<limits.h>
#include<stdlib.h>

void printArray(int *A, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
}
```

```
int maximum(int A[], int n){
    int max = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        if (max < A[i]){
            max = A[i];
        }
    }
    return max;
```

```
}
```

```
void countSort(int * A, int n){
```

```
    int i, j;
```

```
    // Find the maximum element in A
```

```
    int max = maximum(A, n);
```



```
    // Create the count array
```

```
    int* count = (int *) malloc((max+1)*sizeof(int));
```



```
    // Initialize the array elements to 0
```

```
    for (i = 0; i < max+1; i++)
```

```
    {
```

```
        count[i] = 0;
```

```
    }
```



```
    // Increment the corresponding index in the count array
```

```
    for (i = 0; i < n; i++)
```

```
    {
```

```
        count[A[i]] = count[A[i]] + 1;
```

```
    }
```



```
    i = 0; // counter for count array
```

```
    j = 0; // counter for given array A
```



```
    while(i<= max){
```

```
        if(count[i]>0){
```

```
            A[j] = i;
```

```
            count[i] = count[i] - 1;
```

```
            j++;
```

```
        }
```

```
        else{
```

```

        i++;
    }
}

int main(){
    int A[ ] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printArray(A, n);
    countSort(A, n);
    printArray(A, n);
    return 0;
}

```

[Copy](#)

Code Snippet 4: Program to implement the count Sort Algorithm

Let us now check if our functions work well. Consider an array A of length 7.

```

int A[ ] = {9, 1, 4, 14, 4, 15, 6};
int n = 7;
printArray(A, n);
countSort(A, n);
printArray(A, n);

```

[Copy](#)

Code Snippet 5: Using the *countSort* function

And the output we received was:

```

9 1 4 14 4 15 6
1 4 4 6 9 14 15

```

PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

[Copy](#)

Figure 1: Output of the above program

So, our array got sorted. That was easy for you to understand, I believe. And it was indeed as easy as pie. We'll now quickly move to the analysis part and wrap up our sorting algorithm series.

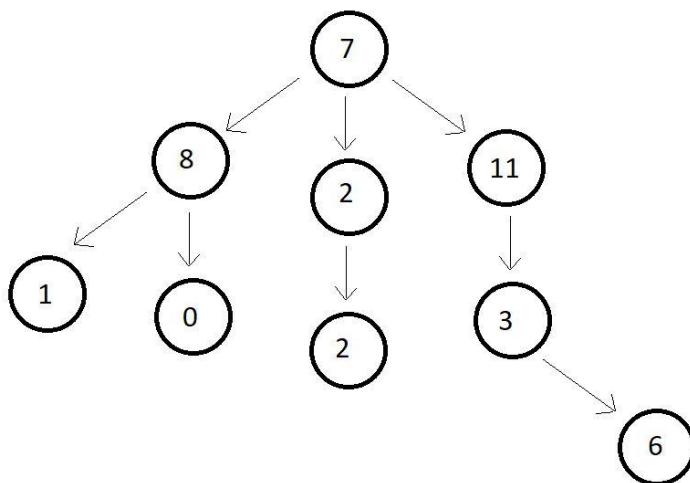
Time Complexity of Count Sort:

Calculating the time complexity of the count sort algorithm is one of the easiest jobs to do. If you 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.

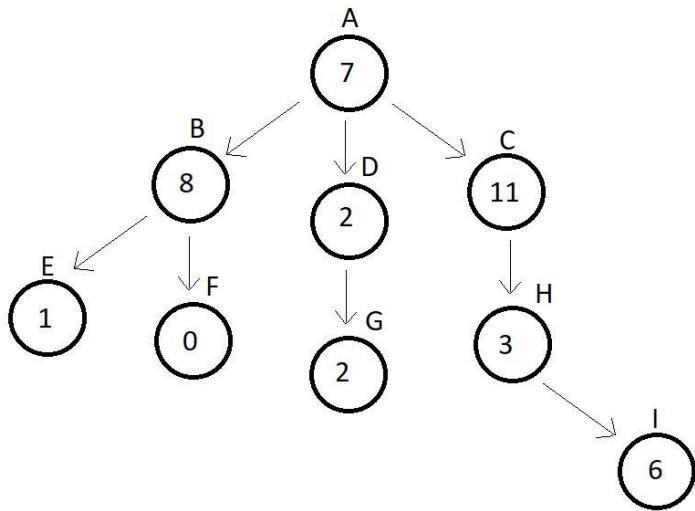
Introduction to Trees

In the wake of such an extensive and lengthy study of sorting algorithms, we would now like to move on to the study of some more advanced data structures. Today, we'll begin discussing trees. It is unlikely that I won't be able to explain trees to you after I finish this lecture. The flow of the concept will remain the same, first the theory and then their applications and problems.

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 give you a better idea of how a tree looks like, let me give an example:



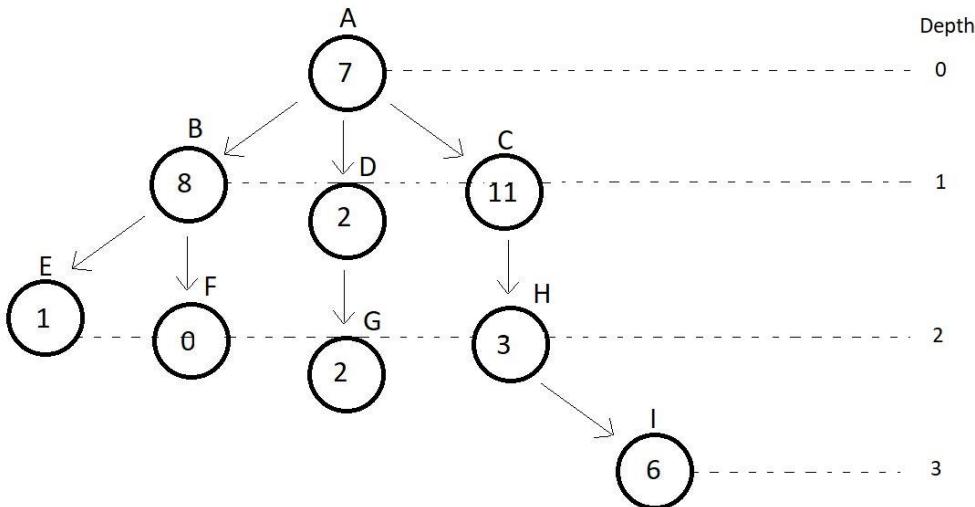
Every circle represents a node, and every arrow represents the hierarchy. For you to be able to understand the terminology associated with trees, I will further name these nodes.



Now, you can very easily say that node C is the child of node A or node B is the parent of node E. You would be wondering what a parent or child is. I was coming to that only.

Terminologies used in trees:

1. **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.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Internal node:** Nodes with at least one child. Here, nodes B, D and C are internal nodes.
9. **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.



10. **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.

There are still a lot of concepts left. To help you get a better understanding of trees, I presented this brief introduction. These were the basics you had to learn to move to the advanced topics. To understand what lies ahead, you need to learn the terminologies. Make sure you do that at least.

What is a Binary Tree?

In the last lecture, we started our new chapter, trees. There, we saw the basics of trees, their structure, and some of the terminology that we would need to know to understand what was to come. In this lesson, we will learn about a special kind of tree called the binary tree.

Before we proceed to learn what binary trees are, we'll give ourselves a quick revision of what we learned in the previous lecture.

1. A tree is made up of nodes and edges.
2. The topmost node is called the **Root** node which points to another node, is the **parent** of that node and the node which the parent is pointing at is the **child** of that parent node. And nodes having the same parents are called **siblings** of each other.
3. Nodes having zero children are the **leaf nodes or the external nodes**, and nodes having at least one child are the **internal nodes**.
4. **Ancestors** of a node are the nodes accessible by traversing upwards along the edges. They are either the parents or the parents of the parents.
5. **Descendants** of a node are the nodes accessible by traversing downwards along the edges. They are either the children or the children of the children.

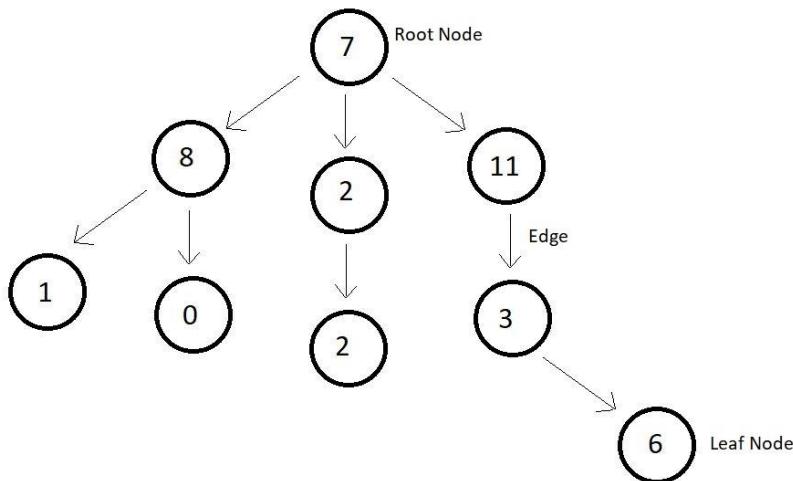
6. **Height** of a node is the number of edges in between the deepest leaf and that node. And **depth** of a node is the number of edges between the root and that node.

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

1. A tree with n nodes has $n-1$ Why $n-1$?

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.

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

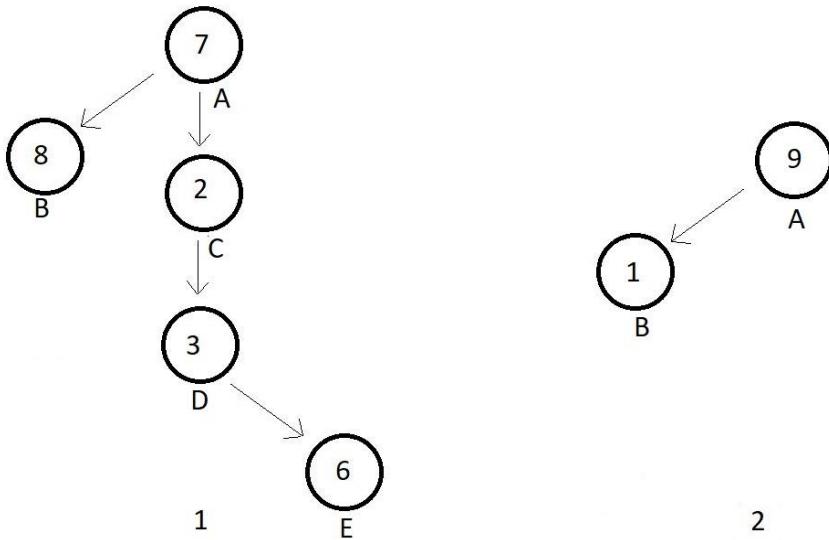


In the above tree, the number of nodes is 9, and hence the number of edges are 8. You can even count and verify the fact that a tree with n nodes has $n-1$ edges. Moreover, the highest degree of a node is that of the root node, which has 3 children. Hence the degree of the tree is also 3.

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.

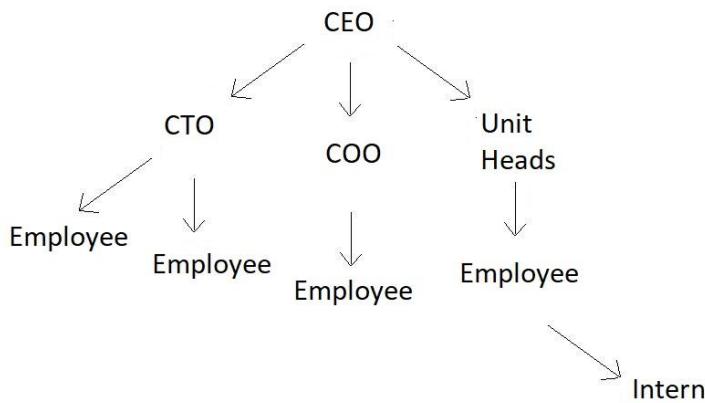
Few examples of a binary tree are mentioned below:



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

In the previous lecture, we learned about a special variety of trees, binary trees. We saw key characteristics of a binary tree that makes it different from other trees. Today, we'll see more about trees, binary trees, and different types of binary trees. Earlier we learned about trees, and how it offers a non-linear representation of data. Most of the data structures we discussed in this playlist had represented data in a linear fashion, be it arrays, queues, stacks, or linked lists. Trees are ideally used to represent hierarchical data. Although I should have said most of this in the first video, few things actually become clearer as time goes on. So, trees are useful in representing an organizational hierarchy. Refer to the illustration below.

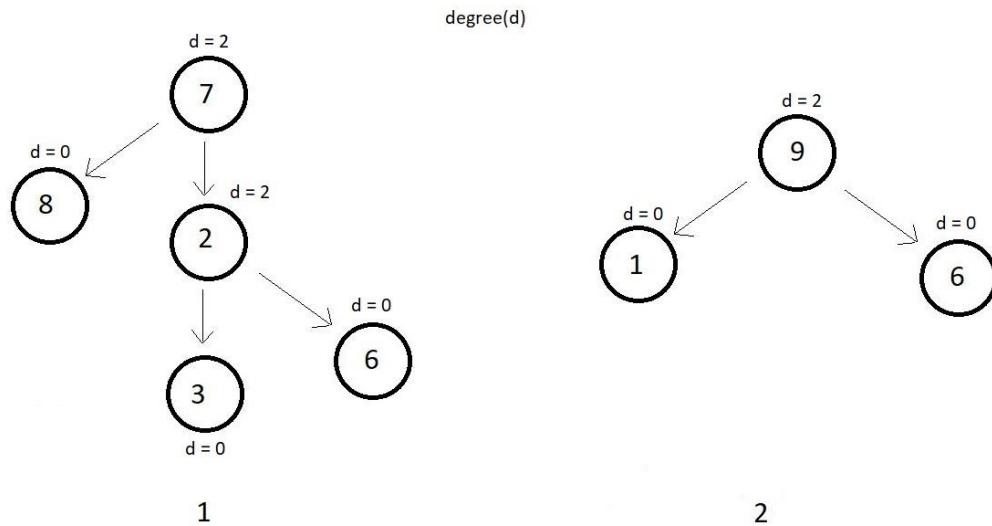


Another good example would be our file manager in desktops. There is a C Drive, it contains folders Windows, Program Files, etc. which further contain folders. So, they are hierarchically represented using a tree.

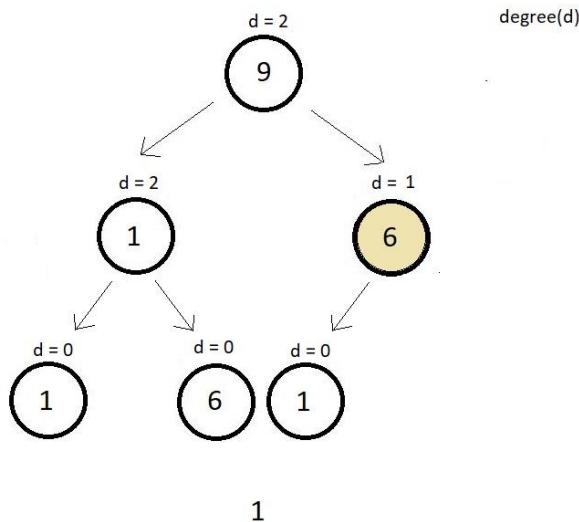
So, these were few things I had in mind, but today's motive is to explain to you all about the various types of binary trees we have.

1. Full or Strict Binary trees:

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:

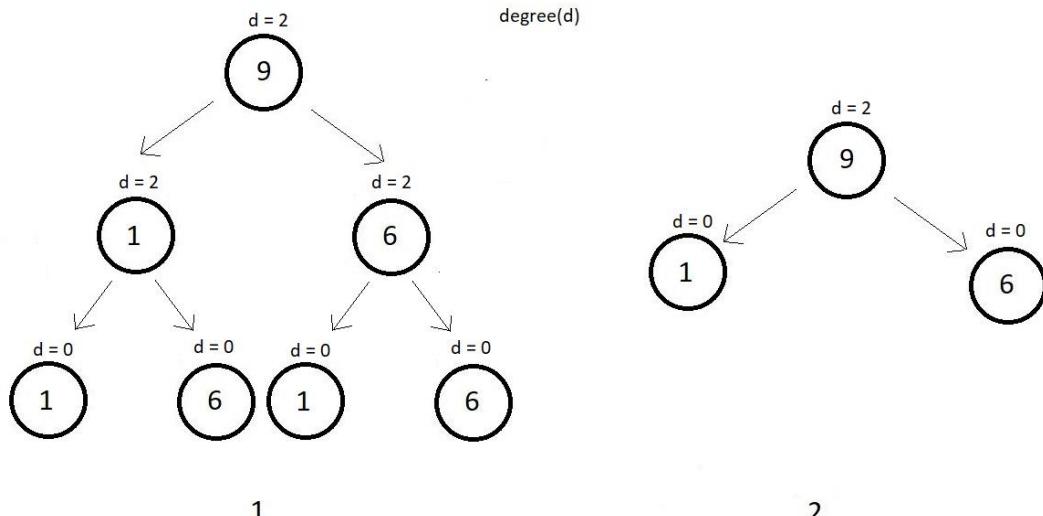


Below illustrated is a binary which is not a strict or full binary tree because the colored node has a degree of just 1.

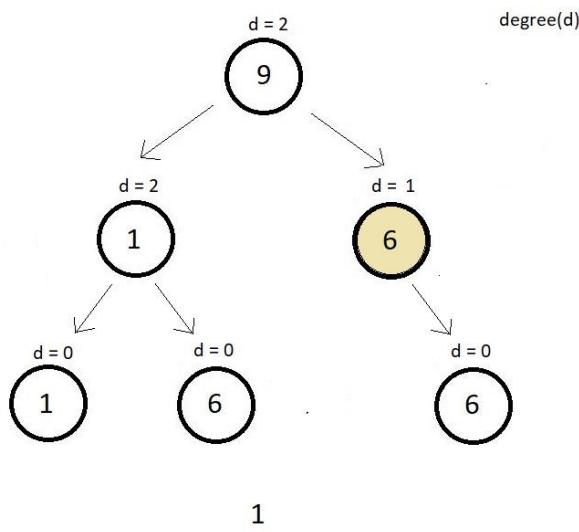


2. Perfect Binary Tree:

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:

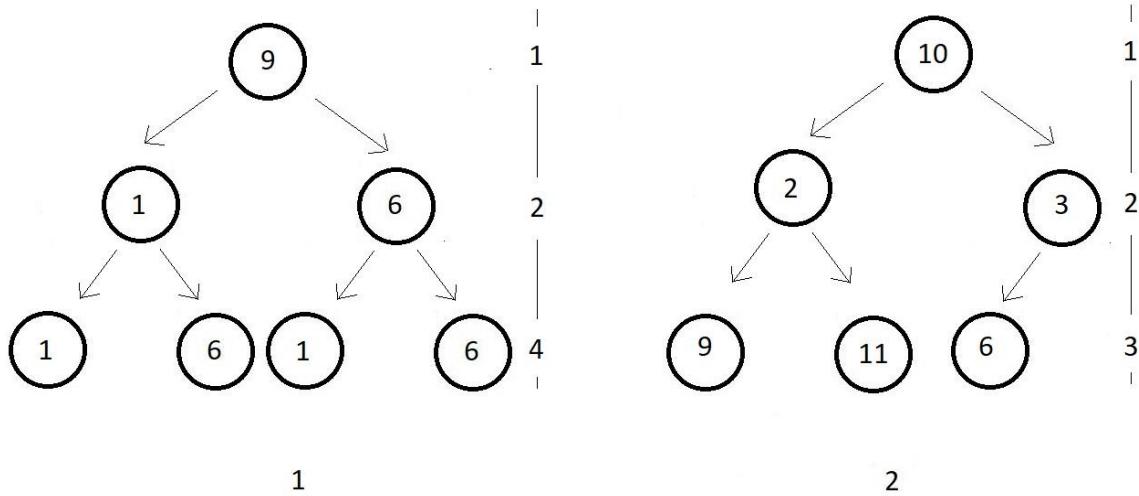


Every leaf node above in both the examples are on the same level and all the internal nodes have a degree 2. Below illustrated is a binary which is not a perfect binary tree because the colored node is an internal node and has a degree of just 1, although each leaf node is on the same level.

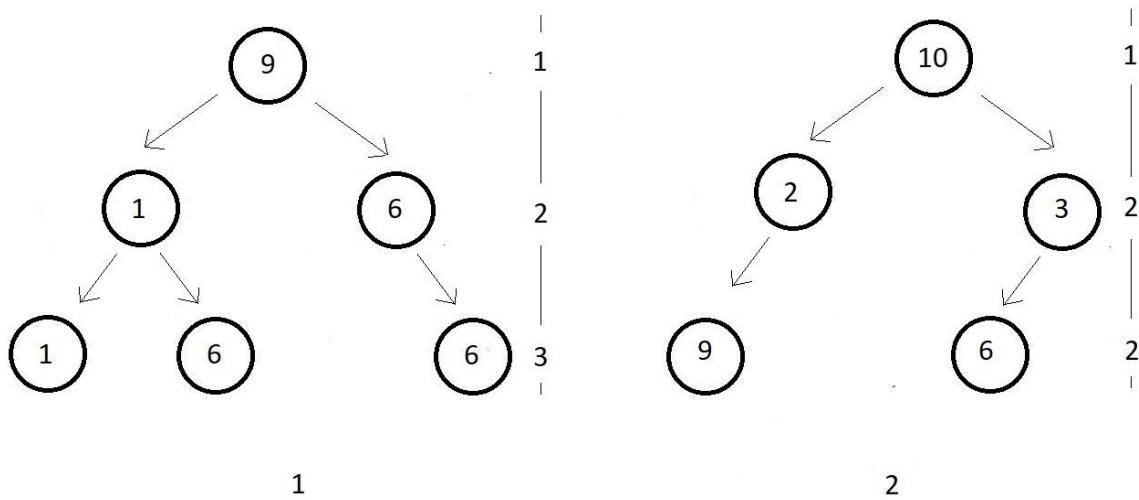


3. Complete Binary Tree:

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 your confusion.



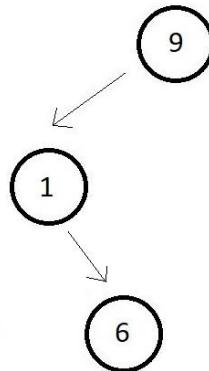
As indicated in figure 1, all levels are completely filled, so nothing further needs to be done. It is a complete binary tree. In figure 2, all nodes are completely filled except the last level which has just 3 keys. It is nonetheless a complete binary tree because all keys are left-aligned. Below illustrated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



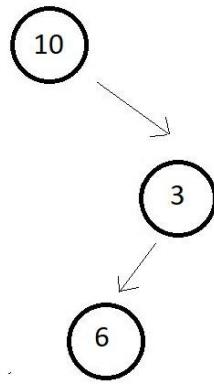
In both the figures, the last level is not complete. And hence we check if all the nodes are aligned to the left. But they aren't in both cases. And hence both of them are not complete.

4. 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:



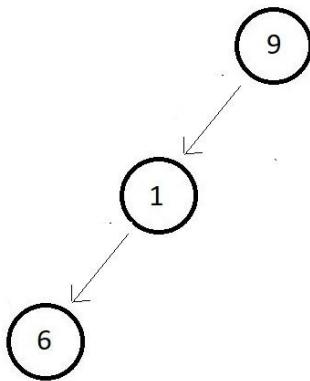
1



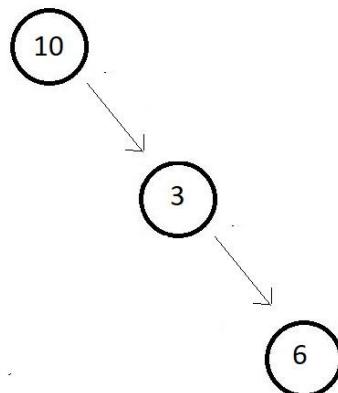
2

5. Skewed trees:

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. Skewed 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.



Left Skewed Tree



Right Skewed Tree

And those were all the various types of binary trees we had. You really don't have to memorize them all, rather just prepare some notes for now, and keep them for your interviews/exams. We already have quite a lot to keep in mind, and I'll not further stress you out by making you memorize these. Keep your concepts straight and you can learn things as per your requirement.

Representation of a Binary Tree

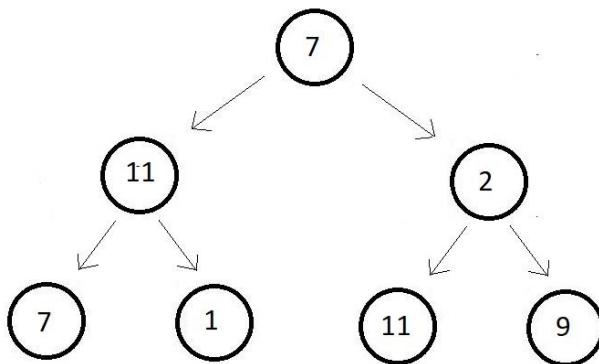
In the previous lecture, we saw the various types of binary trees we have. We looked at their examples, and I am confident that all of you understood everything very clearly. In today's lesson, we'll consider different techniques for representing binary trees in programming, and see which one best suits our needs.

So, the first way to represent a binary tree is by using arrays. We call this 'array representation'. And this method is not very recommended for representing binary trees. You will very soon know why.

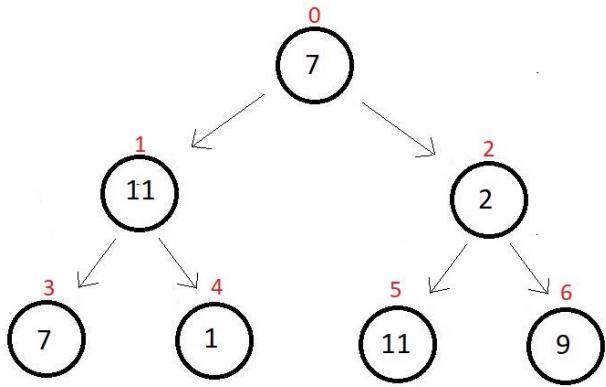
Array representation of Binary trees:

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 you declare an array of size 100, and after storing 100 nodes in it, you cannot even insert a single element further, regardless of all the spaces left in the memory. Another way you'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.



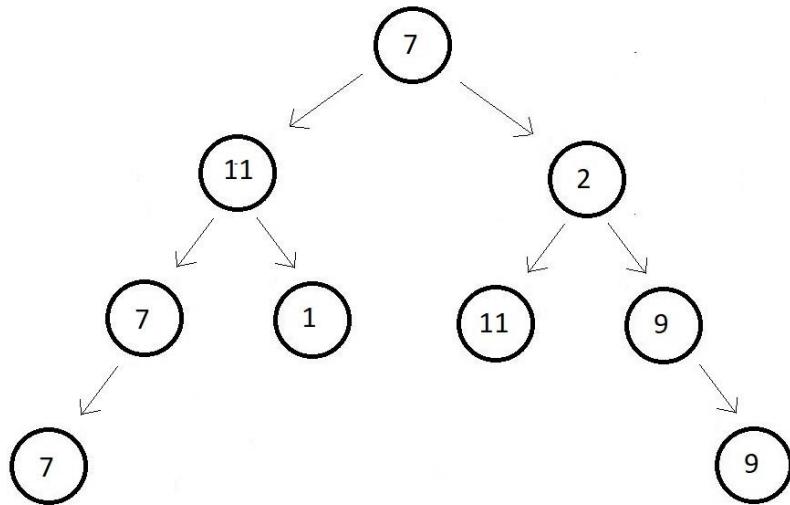
And there are actually a number of ways to represent these nodes via an array. I'll use the most convenient one where we traverse each level starting from the root node and from left to right and mark them with the indices these nodes would belong to.



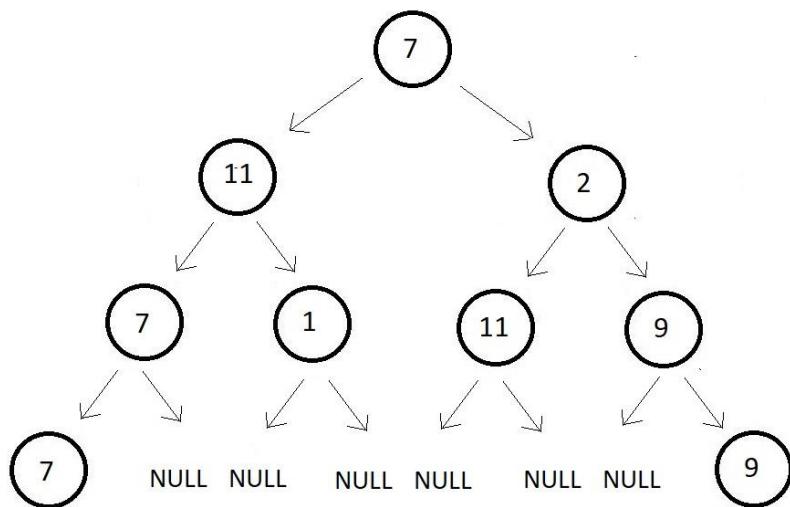
And now we can simply make an array of length 7 and store these elements at their corresponding indices.

0	1	2	3	4	5	6
7	11	2	7	1	11	9

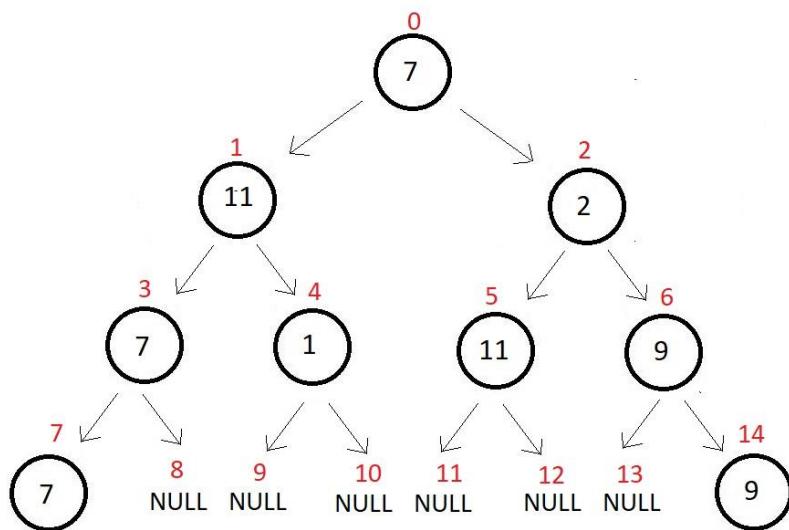
And you might be wondering about the cases where the binary is just not perfect. What if the last level has distributed leaves? Then let me tell you, there is a way out for that as well. Let's consider one case here. A binary tree with 9 nodes, and the last two nodes on the extremities of the last level.



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

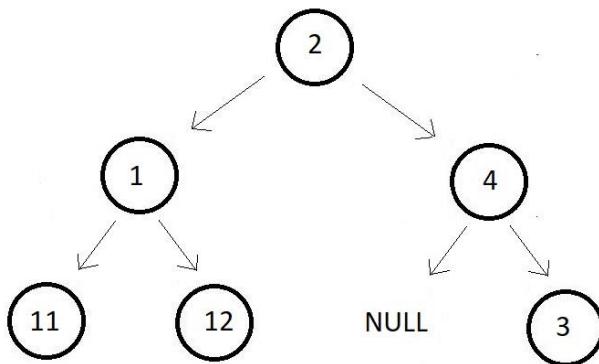
But was this even an efficient approach? Like Binary Trees are made only for efficient traversal and insertion and deletion and using an array for that really makes the process troublesome. Each of these operations becomes quite costly to accomplish. And that size constraint was already for making things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

We have another method to represent binary trees called the linked representation of binary trees. Don't confuse this with linked lists you have studied. And the reason why I am saying that is because linked lists are lists that are linear data structures.

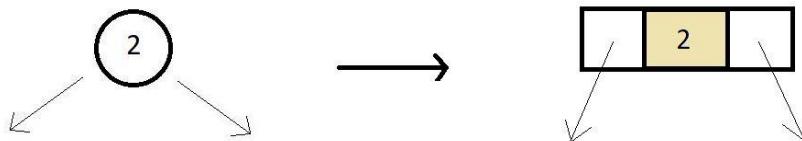
Linked Representation of Binary Trees:

This method of representing binary trees using linked nodes is considered the most efficient method of representation. For this, we use doubly-linked lists. I just hope you recall what doubly-linked lists are. We studied that here in the same playlist [Doubly Linked Lists Explained With Code in C Language](#).

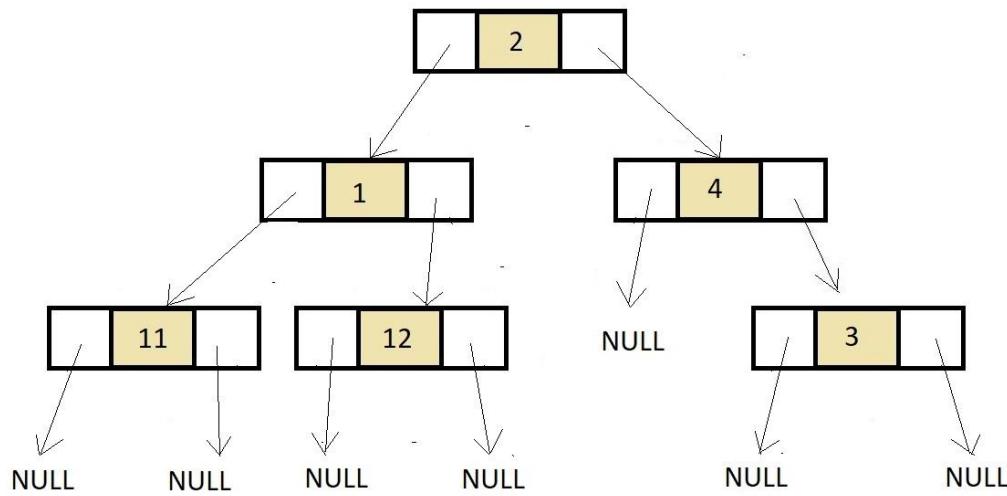
Using links makes the understanding of a binary tree very easy. It actually makes us visualize the tree even. Suppose we have a binary tree of 3 levels.



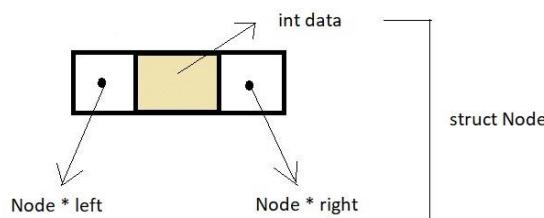
Now if you remember a doubly linked list helped us traversing both to the left and the right. And using that we would create a similar node here, pointing both to the left and the right child node. Follow the below representation of a node here in the linked representation of a binary tree.



You 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.



And let me show you that struct Node definition part in C language:

```

struct node{
    int data;
    struct node* left;
    struct node* right;
};

```

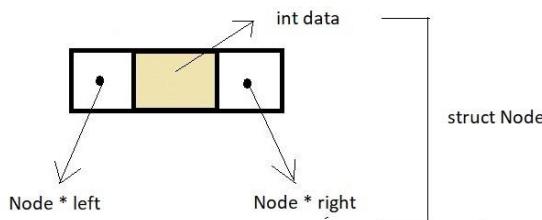
[Copy](#)

Code Snippet 1: Creating the struct Node

Linked Representation Of Binary Tree in C

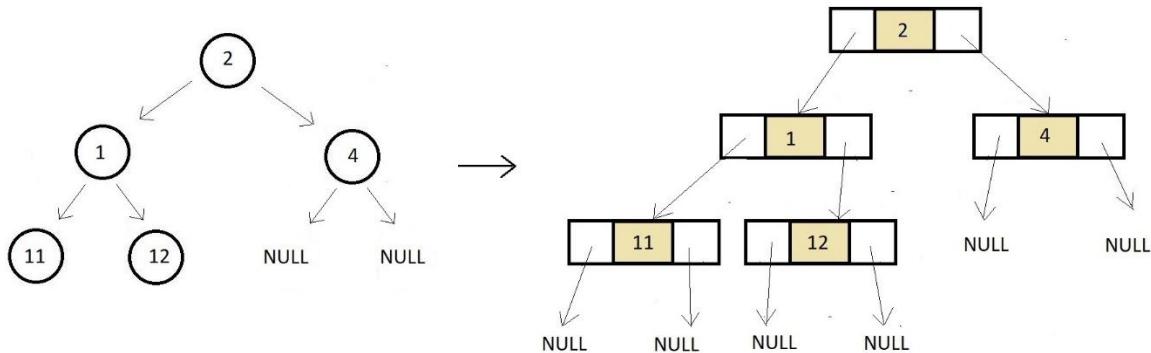
In the last lecture, we saw different representations of a binary tree. We saw its array representation. We saw its linked representation. In contrast to the linked representation of binary trees, we found its array representation to be futile at the end because of its size constraint and the complexity of its implementation. And since the array representation is not that tough to implement I believe, I'll continue with the linked representation, and a quick revision on that should work.

1. We use the concept of doubly-linked lists to represent a node.



2. The left pointer of this node points to the left child and the right pointer points to the right child, and if there is no left or right child, we represent that using a NULL.

See how a tree looks in real life, and how efficiently the linked representation of a tree helps us visualize the same.



We also saw the declaration of the structure Node in C. Today we'll see all that in detail. We'll see how these nodes get linked, and the creation of a binary tree via that. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. The first thing would be to create the struct Node we discussed yesterday. So, create a struct Node. Inside the structure, we have three embedded elements, first is an integer variable `data` to store the data of the node, second is a struct Node pointer

called `left` to store the address of the left child node, and third is again a struct Node pointer called `right` to store the address of the right child node.

```
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};
```

Copy

Code Snippet 1: Creating the struct Node

2. Our next job would be to create the root node. In main, create struct Node pointer `p`, and assign to it the memory in heap using malloc. Don't forget to include the header file `<malloc.h>` or `<stdlib.h>`. And then using the pointer operator `→`, assign some data to its `data` element and NULL to both the left and the right element of the struct Node `p`.

3. And now we can proceed to creating the further nodes. We have different ways to do that. First one is to copy the whole thing we did for the root node twice for both the children and name them `p1` and `p2`. This will create three separate nodes `p`, `p1` and `p2`. Then just link them together by changing the left element of `p` from NULL to the left node's pointer `p1` and the right element of `p` from NULL to the right node's pointer `p2`.

4. But this is somewhere redundant and not considered a good practice as we are copying the whole thing again and again. So, we would create a dedicated function for creating a node.

Creating the `createNode` function:

5. Create a struct Node* function `createNode`, and pass the data for the node as the only parameter. Create a struct Node pointer `n`. Reserve a memory in heap for `n` using malloc. Basically, do the same thing we did above. Point the left and the right element of the struct `n` to NULL, and fill the data in the `data` element. And return the node pointer `n` you created. This would simply create a node, and you can now very easily link them as per your wish via main to the other nodes.

```
struct node* createNode(int data){  
    struct node *n; // creating a node pointer  
    n = (struct node *) malloc(sizeof(struct node)); // Allocating  
    memory in the heap  
    n->data = data; // Setting the data  
    n->left = NULL; // Setting the left and right children to NULL
```

```
n->right = NULL; // Setting the left and right children to NULL  
return n; // Finally returning the created node  
}
```

Copy

Code Snippet 2: Creating the function createNode

Here is the whole source code:

```
#include<stdio.h>  
#include<malloc.h>  
  
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
struct node* createNode(int data){  
    struct node *n; // creating a node pointer  
    n = (struct node *) malloc(sizeof(struct node)); // Allocating  
    memory in the heap  
    n->data = data; // Setting the data  
    n->left = NULL; // Setting the left and right children to NULL  
    n->right = NULL; // Setting the left and right children to NULL  
    return n; // Finally returning the created node  
}  
  
int main(){  
    /*  
     * Constructing the root node  
     */  
    struct node *p;  
    p = (struct node *) malloc(sizeof(struct node));
```

```

    p->data = 2;
    p->left = NULL;
    p->right = NULL;

    // Constructing the second node
    struct node *p1;
    p1 = (struct node *) malloc(sizeof(struct node));
    p->data = 1;
    p1->left = NULL;
    p1->right = NULL;

    // Constructing the third node
    struct node *p2;
    p2 = (struct node *) malloc(sizeof(struct node));
    p->data = 4;
    p2->left = NULL;
    p2->right = NULL;
    */

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(2);
    struct node *p1 = createNode(1);
    struct node *p2 = createNode(4);

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    return 0;
}

```

[Copy](#)

Code Snippet 3: Implementing binary trees in C

Now we can see if our program doesn't revert any error while we try linking 3 nodes p, p1 and p2.

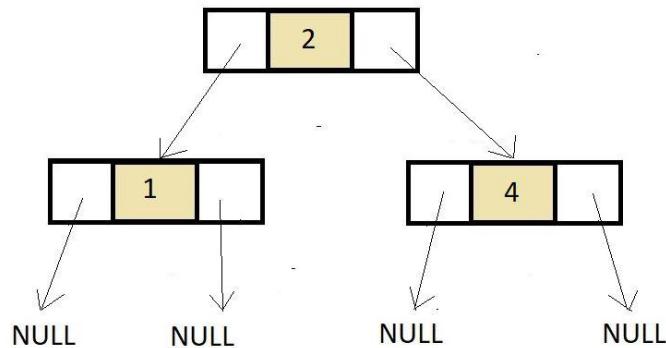
```
struct node *p = createNode(2);
struct node *p1 = createNode(1);
struct node *p2 = createNode(4);

p->left = p1;
p->right = p2;
```

Copy

Code Snippet 4: Using `createNode` to create nodes.

No, it didn't show any error, and this is how we created three nodes and linked two of them to the root node. You can imagine how it would have looked.



Traversal in Binary Tree (InOrder, PostOrder and PreOrder Traversals)

In the last tutorial, we learned to implement the linked representation of binary trees in C. We saw how easy it was for us to visualize the elements decorated as a real tree when implemented using linked representation. Today, we'll learn how to traverse in binary trees.

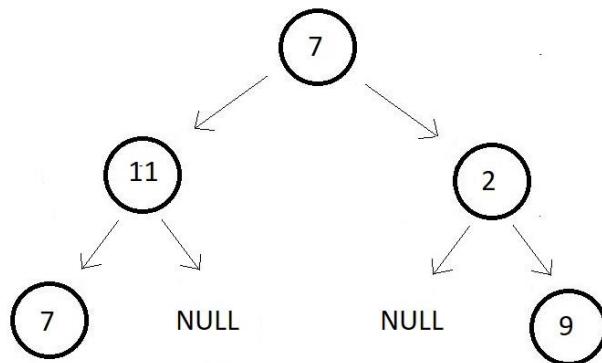
Traversal in a binary tree:

If you remember the previous lectures where we saw the traversal in linked lists, traversal meant visiting every node in the list at least once. Traversal has several applications, one might want to visit each node and print their values, another

might want to add each of those elements. It is, therefore, crucial to study the techniques of traversal.

But things were easy when we used to traverse through the lists, or the arrays, since they were linear structures, and the directions you choose to traverse were limited, either front to rear or from rear to front. But today we'll accomplish the complexity of traversing through a non-linear data structure, binary tree.

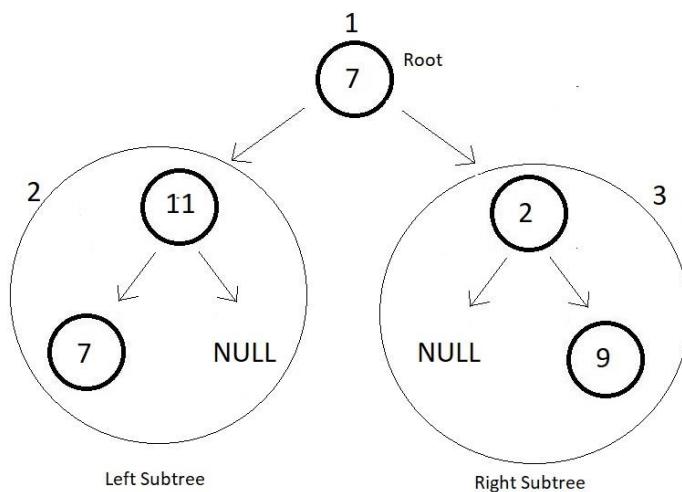
So basically, we have three different ways to traverse in a binary tree. They are InOrder, PostOrder, and PreOrder Traversals. Let's take a sample tree, and walk through it one by one, using each traversal technique for better understanding.



Let's start with the first one.

PreOrder Traversal in a Binary Tree:

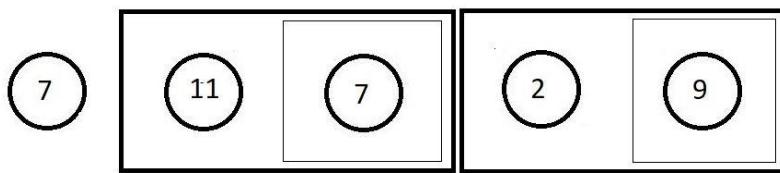
So in this traversal technique, the first node you start with is the root node. And thereafter you visit the left subtree and then the right subtree. Taking the above example, I'll mark your order of traversal as below. You first visit section 1, then 2, and then 3.



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

So, here you 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, you visit that and move further to the left subtree of this subtree. There you see a single element 7, you visit that, and then move to the right subtree which is a NULL. So, you're finished with the left subtree of the main tree. Now, you 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.

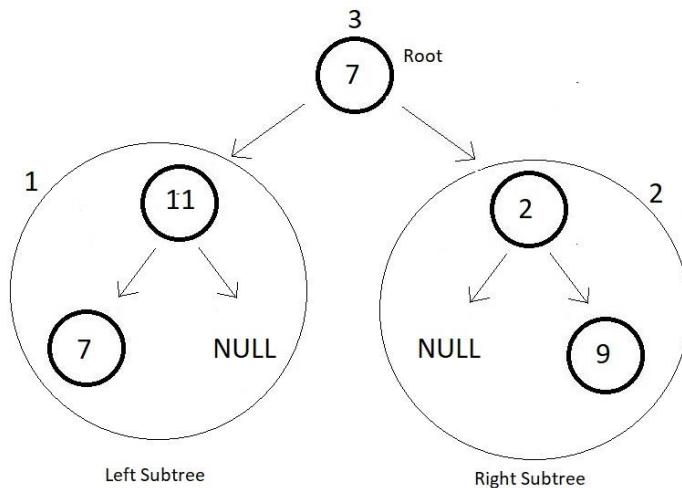
So basically, you recursively visit each subtree in the same order. And your final traversal order is:



Here, each block represents a different subtree.

PostOrder Traversal in a Binary Tree:

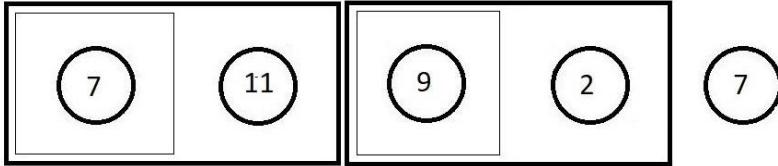
In this traversal technique, things are quite opposite to the PreOrder traversal. Here, you first visit the left subtree, and then the right subtree. So, the last node you'll visit is the root node. Taking the same above example, I'll mark your order of traversal as below. You first visit section 1,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



This was again a general idea of you traverse in a binary tree using PostOrder Traversal. Each time you get a tree, you first visit its left subtree, and then its right subtree, and then move to its root node.

I expect you to write the flow of the traversal in PostOrder yourself, and let me know if you could. We would anyway see them in detail.

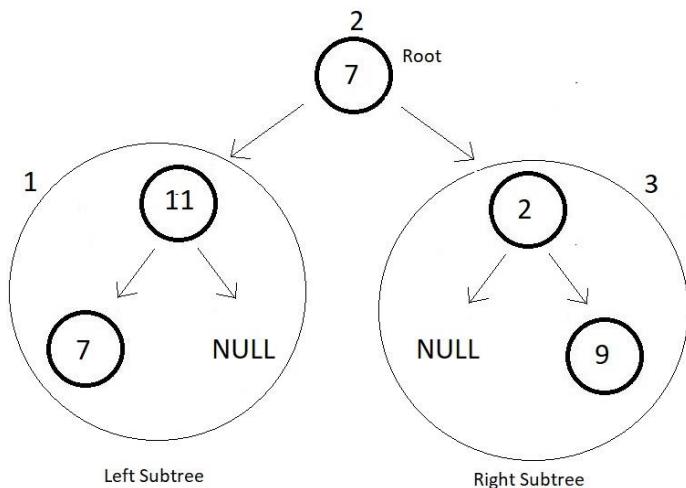
Hint: Your final traversal order should be.



Here, each block represents a different subtree.

InOrder Traversal in a Binary Tree:

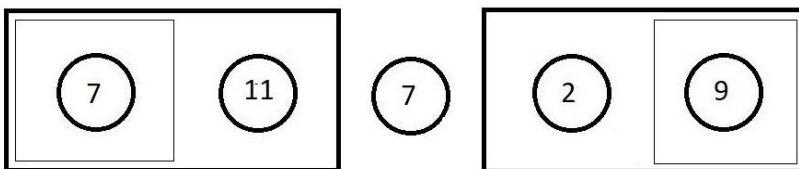
In this traversal technique, we simply start with the left subtree, that is you first visit the left subtree, and then go to the root node and then you'll visit the right subtree. Taking the same above example, I'll mark your order of traversal as below. You first visit section 1, then 2, and then 3.



This was a general idea of you traverse in a binary tree using InOrder Traversal. Each time you get a tree, you first visit its left subtree, and then its root node, and then finally its right subtree.

I expect you to write the flow of the traversal in InOrder yourself, and let me know if you could.

Hint: Your final traversal order should be:



Here, each block represents a different subtree.

So basically, the names PreOrder, PostOrder, and InOrder were given with respect to the position we keep our root node at while traversing in the Binary Tree. Since we

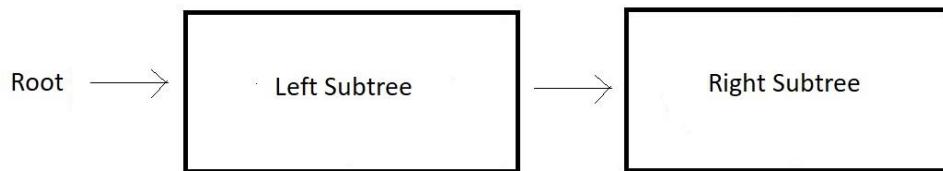
first visit the root node in the first technique, that's why the name PreOrder. And we visit the root node at last in the PostOrder and in between in InOrder. Each of these techniques has one thing in common: the left subtree is traversed before the right subtree.

And these were the traversal techniques we had. We'll cover the programming part for each of these three in our coming lectures and in great detail. You shouldn't miss out on any of these. And do not forget the two techniques I left for you to explore.

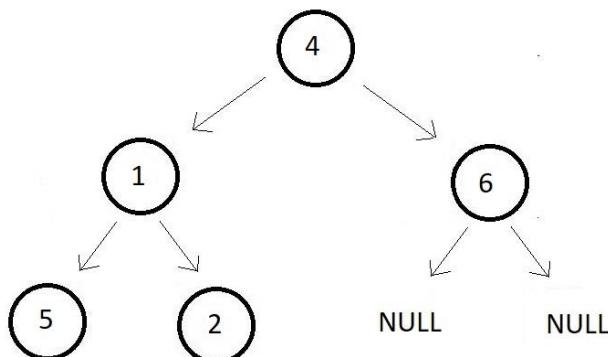
Preorder Traversal in a Binary Tree (With C Code)

In the last lecture, we saw why we traverse through trees for different purposes. We primarily discussed, in brief, all the three types of traversal techniques. I discussed the flow of the PreOrder Traversal there itself and asked you to cover the last two yourselves. We will surely cover them in detail starting today with the first one, PostOrder Traversal, and see its programming implementation in C.

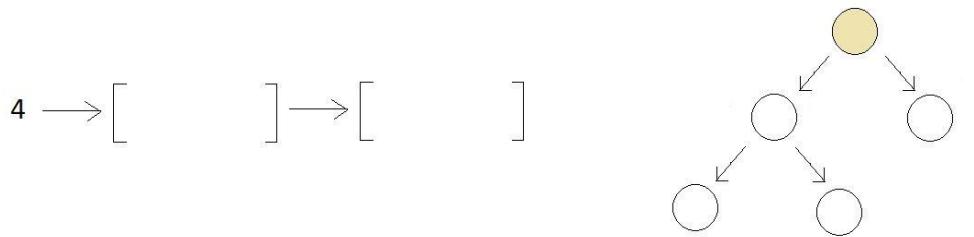
We basically have a basic idea of how PreOrder traversal works. Today we will see that in great detail. So, here you first start with the root node of the main tree and then get the hold of the left subtree. Now consider this left subtree as a new tree, and apply PreOrder on this. Recursively doing this with all the further subtrees, you will visit each node of this tree. Once you finish with the left subtree, you go to the right subtree and consider this as another tree and repeat the whole thing again.



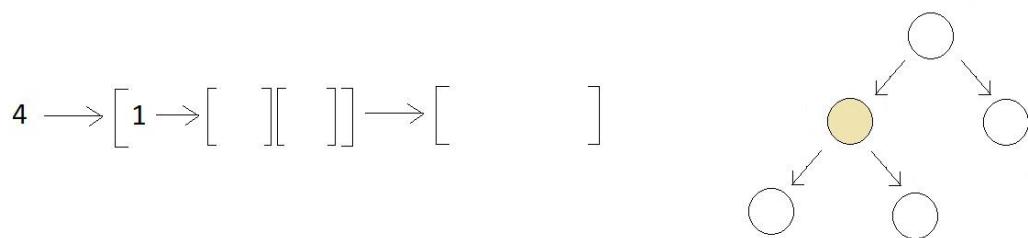
Let's first take an example binary tree, and apply PreOrder Traversal on the same.



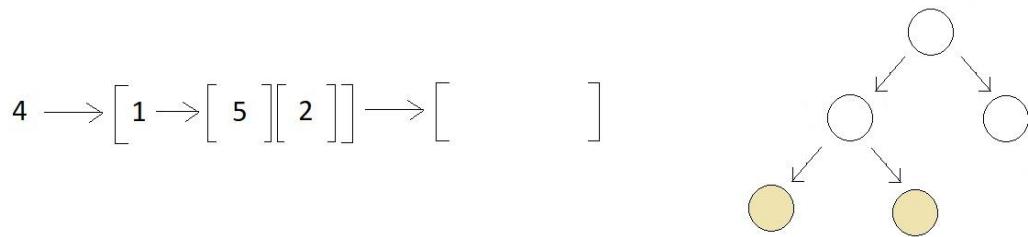
In the first step, you visit the root node and mark the presence of the left and the right subtree as separate individual trees.



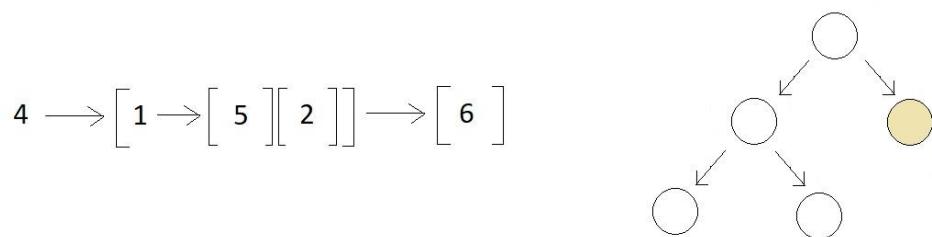
After you visit the root node, you move to the left subtree considering it as a different tree, and start with its root node.



And then you 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, you 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: $4 \rightarrow 1 \rightarrow 5 \rightarrow 2 \rightarrow 6$.

Now we are ready to implement its programming having studied the flow in detail. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. First of all, we wouldn't start from scratch creating the struct Node and the createNode function and everything. So just copy the whole thing we did in our previous programming lecture and paste them here. This would save us a lot of time.
2. Create all the five nodes, using the createNode function, and link them using the arrow operator, and altering their left and right pointer elements. This creates our tree. The next thing would be to create the preOrder function.

```
// Constructing the root node - Using Function (Recommended)

struct node *p = createNode(4);
struct node *p1 = createNode(1);
struct node *p2 = createNode(6);
struct node *p3 = createNode(5);
struct node *p4 = createNode(2);

// Finally The tree looks like this:

//      4
//     / \
//    1   6
//   /   \
//  5   2

// Linking the root node with left and right children

p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;
```

[Copy](#)

Code Snippet 1: Creating the Binary tree

Creating the preOrder function:

3. Create a void function preOrder and pass the pointer to the root node of the tree you want to traverse as the only parameter. Inside the function, check if the pointer is not NULL, otherwise we wouldn't do anything. So, if it is not NULL, print the data element of the root struct node by using the arrow operator.
4. After you finish visiting the root node, simply call the same function recursively on the left and the right subtrees and you're done. Applying recursion does your job in

its own subtle ways. It considers the left subtree as an individual tree and applies preorder on it, and the same goes for the right subtree.

```
void preOrder(struct node* root){  
    if(root!=NULL){  
        printf("%d ", root->data);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

[Copy](#)

Code Snippet 2: Creating the preOrder function

Here is the whole source code:

```
#include<stdio.h>  
#include<malloc.h>  
  
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
struct node* createNode(int data){  
    struct node *n; // creating a node pointer  
    n = (struct node *) malloc(sizeof(struct node)); // Allocating  
    memory in the heap  
    n->data = data; // Setting the data  
    n->left = NULL; // Setting the left and right children to NULL  
    n->right = NULL; // Setting the left and right children to NULL  
    return n; // Finally returning the created node  
}
```

```

void preOrder(struct node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main(){

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(4);
    struct node *p1 = createNode(1);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(5);
    struct node *p4 = createNode(2);

    // Finally The tree looks like this:
    //      4
    //     / \
    //    1   6
    //   /   \
    //  5   2

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;

    preOrder(p);
}

```

```
    return 0;  
}
```

Copy

Code Snippet 3: Implementing the preOrder function

Now simply call the preOrder function passing the pointer to the root node as its parameter and see if it actually visits each node.

```
preOrder(p);
```

Copy

Code Snippet 4: Using the preOrder function

And the output we received was:

```
4 1 5 2 6
```

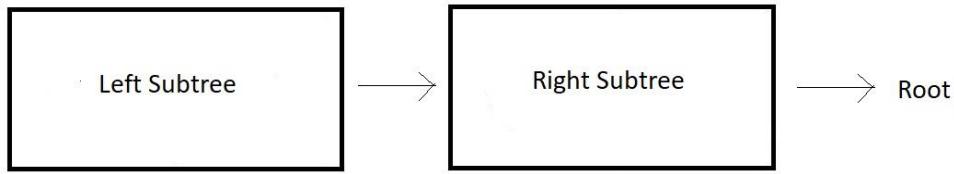
```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

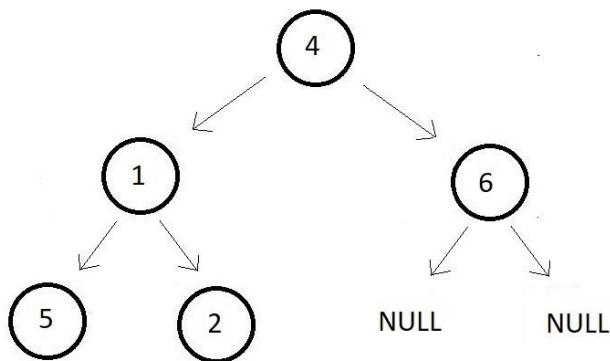
Figure 1: Output of the above code

PostOrder Traversal in a Binary Tree (With C Code)

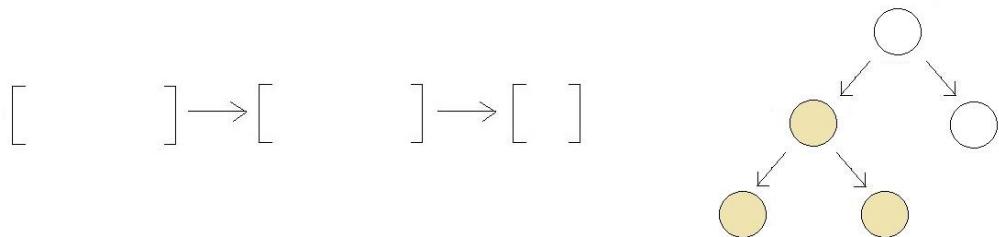
In the last lecture, we discussed in detail our first traversal technique, the PreOrder Traversal. We saw its programming implementation in C. If you haven't checked it out already, I would suggest doing so before proceeding. You can then get an idea of the general approach we are taking. Today, we'll see the PostOrder Traversal in detail, and also cover its programming part. We clearly have a basic idea of how PostOrder traversal works. I had asked you all previously to explore the flow of this technique yourselves. Today we will see that in detail. So, here you first start with the left subtree and consider that as another tree in itself. You then apply PostOrder on this subtree. You then swiftly move to the right subtree and repeat the whole thing again considering this as another individual tree. At the end, you visit the root node of the main tree and here you finish visiting the whole tree.



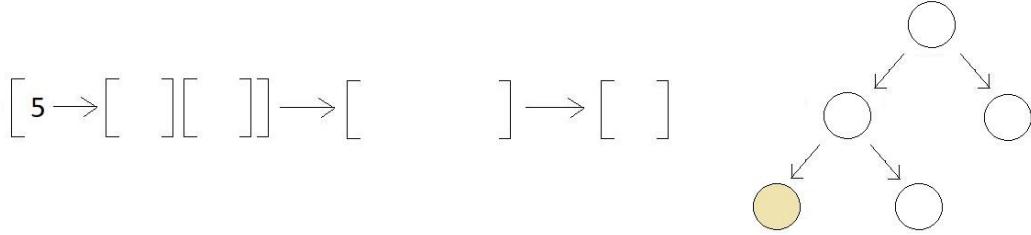
I would like to follow the same approach as I did in my last lecture. First, we will take an example binary tree, and apply PostOrder Traversal on the same.



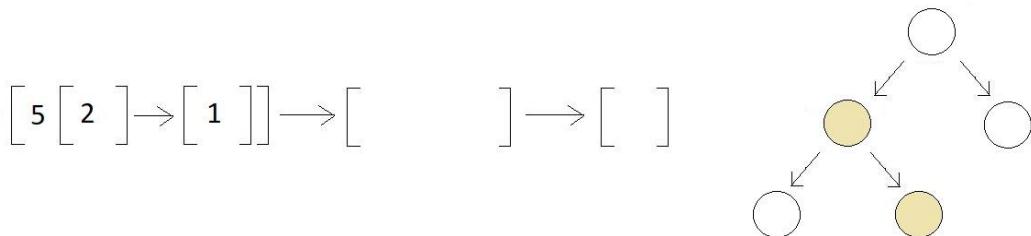
In the first step, you visit the left subtree considering it as a totally different tree. But before we start with the left subtree, we mark the presence of the right subtree and the root as following:



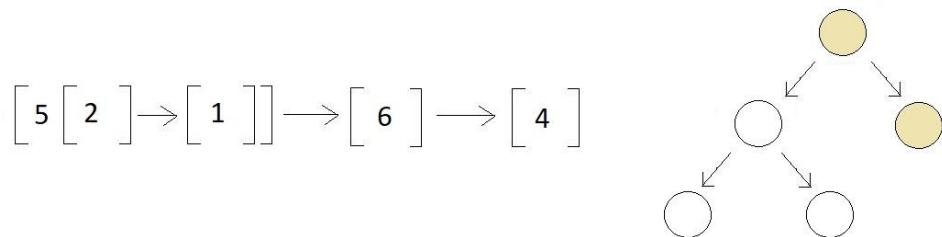
Now, you go through the left subtree, and visit its left node first.



And then you 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, you 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: $5 \rightarrow 2 \rightarrow 1 \rightarrow 6 \rightarrow 4$.

Following a thorough study of its flow, we are now ready to implement its programming. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Following what we did before, we wouldn't start from scratch creating the struct Node and the createNode function and everything. We would just copy the whole thing we did in the PreOrder Traversal programming part and paste them here. This would save us a lot of time and help us compare both Pre and Post Order traversals.
2. Create all the five nodes, using the createNode function, and link them using the arrow operator and altering their left and right pointer element.

This creates our tree. The next thing would be to create the postOrder function.

```
// Constructing the root node - Using Function  
(Recommended)
```

```
struct node *p = createNode(4);  
struct node *p1 = createNode(1);  
struct node *p2 = createNode(6);  
struct node *p3 = createNode(5);  
struct node *p4 = createNode(2);
```

```
// Finally The tree looks like this:
```

```
//      4  
//      / \\  
//      1   6  
//      / \\  
//      5   2
```

```
// Linking the root node with left and right children
```

```
p->left = p1;  
p->right = p2;  
p1->left = p3;  
p1->right = p4;
```

[Copy](#)

Code Snippet 1: Creating the Binary tree

Creating the postOrder function:

3. Create a void function postOrder and pass the pointer to the root node of the tree you want to traverse as the only parameter. Inside the function, check if the pointer is not NULL, otherwise we wouldn't do anything. If it is not NULL, we would not directly print the data of the root since this time it's the last one to get visited.

4. So first, you simply call the same function recursively on the left subtree and then the right subtree using the left and the right elements of the root struct. Once called, recursively, the function now considers the left subtree as an individual tree and applies postorder on it, and the same goes for the right subtree.

5. After visiting them both, you just print the data element of the root node marking it visited. And you are done.

```
void postOrder(struct node* root){  
    if(root!=NULL){  
        postOrder(root->left);  
        postOrder(root->right);  
        printf("%d ", root->data);  
    }  
}
```

Copy

Code Snippet 2: Creating the postOrder function

Here is the whole source code:

```
#include<stdio.h>  
#include<malloc.h>  
  
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
struct node* createNode(int data){  
    struct node *n; // creating a node pointer
```

```
n = (struct node *) malloc(sizeof(struct node)); //  
Allocating memory in the heap  
n->data = data; // Setting the data  
n->left = NULL; // Setting the left and right children to  
NULL  
n->right = NULL; // Setting the left and right children to  
NULL  
return n; // Finally returning the created node  
}
```

```
void preOrder(struct node* root){  
if(root!=NULL){  
    printf("%d ", root->data);  
    preOrder(root->left);  
    preOrder(root->right);  
}  
}
```

```
void postOrder(struct node* root){  
if(root!=NULL){  
    postOrder(root->left);  
    postOrder(root->right);  
    printf("%d ", root->data);  
}  
}
```

```
int main(){  
  
    // Constructing the root node - Using Function  
(Recommended)
```

```
struct node *p = createNode(4);
struct node *p1 = createNode(1);
struct node *p2 = createNode(6);
struct node *p3 = createNode(5);
struct node *p4 = createNode(2);

// Finally The tree looks like this:
//      4
//      / \
//     1   6
//     / \
//    5   2
```

```
// Linking the root node with left and right children
p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;
```

```
preOrder(p);
printf("\n");
postOrder(p);
return 0;
}
```

Copy

Code Snippet 3: Implementing the postOrder function

Now simply call both preOrder and the postOrder function passing the pointer to the root node as their parameter and see how their results vary.

```
preOrder(p);
```

```
    printf("\n");
    postOrder(p);
```

Copy

Code Snippet 4: Using the preOrder and the postOrder function

And the output we received was:

```
4 1 5 2 6
5 2 1 6 4
```

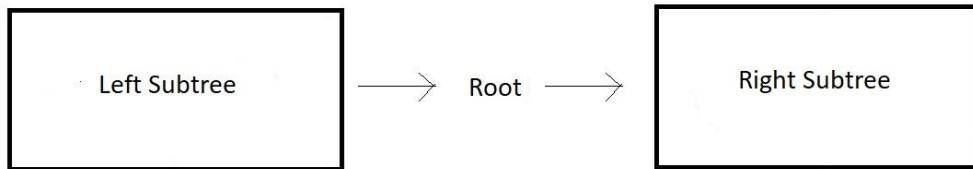
```
PS D:\MyData\Business\code playground\Ds & Algo with
Notes\Code>
```

Copy

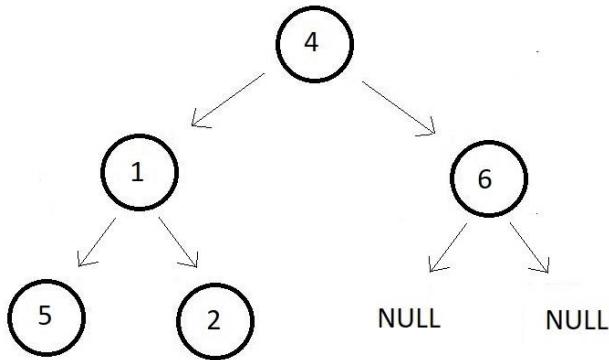
Figure 1: Output of the above code

InOrder Traversal in a Binary Tree (With C Code)

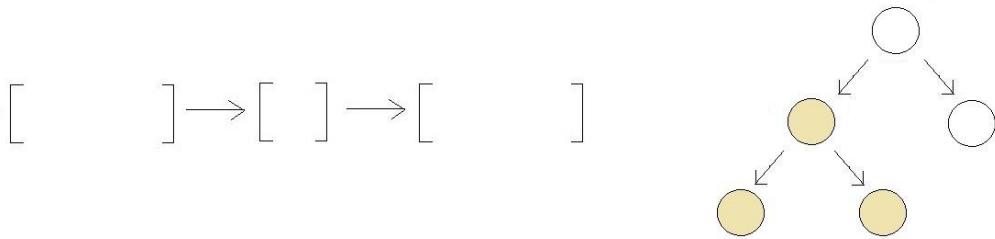
In the last lecture, we discussed in detail our second traversal technique, the PostOrder Traversal. We saw its programming implementation in C. If you have not seen the last two lectures already, I suggest checking them out before continuing. Since today we are going to see our last traversal technique, it is a must to see them. Today, we'll see the InOrder Traversal in detail, and also cover its programming part. We did study the basic idea of how InOrder traversal works. I had even asked you all previously to explore the flow of this technique yourselves. Today we will see that in detail. So, here you first start with the left subtree and consider that as another tree in itself. You then apply InOrder on this subtree. You then come back to the root node. And then swiftly move to the right subtree and repeat the whole thing again considering this as another individual tree. And this would finish your job of visiting each node. Since the root is in between here, hence the name InOrder.



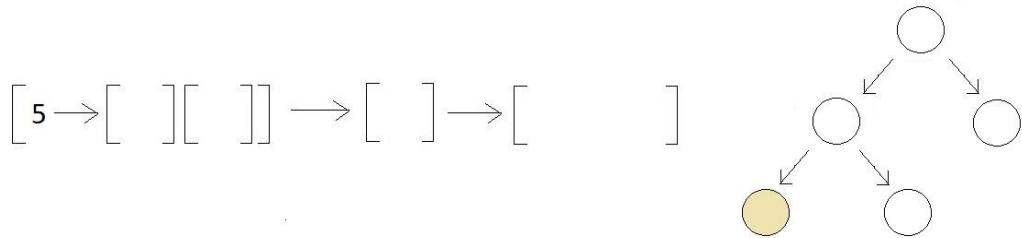
Our flow would remain unchanged, as we follow the same approach as the last two lectures. First, we will take an example binary tree, and apply InOrder Traversal on the same.



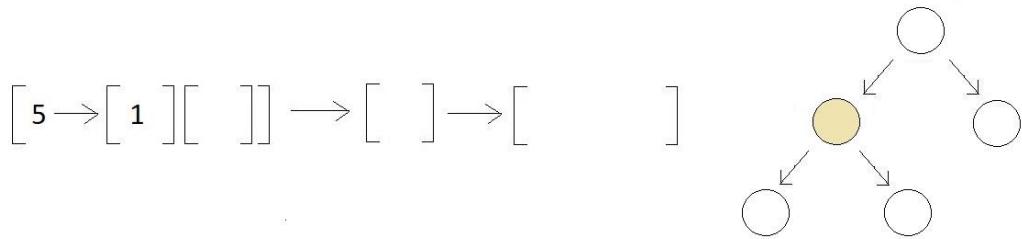
In the first step, you consider the left subtree as a completely different one and apply InOrder separately to it. But before you start with the left subtree, make sure to mark the presence of the root node and the right subtree as following:



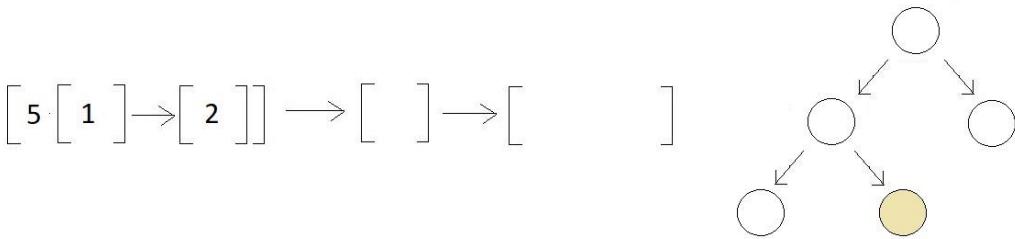
Now, you go through the left subtree, and further visit the left subtree of this new tree first.



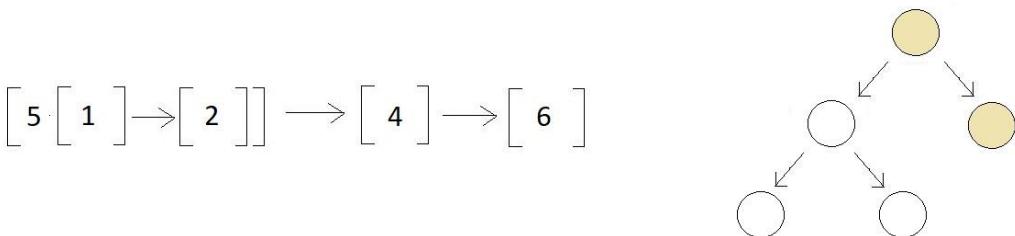
And then you proceed with the root of this new tree we considered.



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



With that being visited you 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: $5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 6$.

Having completed the flow of the InOrder Traversal, we are now ready to implement its programming. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. You know what to do first, right? We avoid doing repetitions, so we wouldn't start from scratch creating the whole struct Node and the createNode thing again rather we would just copy the whole thing we did in the PostOrder Traversal programming part and paste them here. This would get us even the codes for PreOrder and PostOrder.
2. You should now be able to create Binary Tree yourselves by now. So, just create the same binary tree we observed above using the createNode function. And if you are still not sure about creating a binary tree, follow the previous lectures. Let's now move to create the InOrder function.

Creating the inOrder function:

3. Create a void function inOrder and pass the pointer to the root node of the tree you want to traverse as the only parameter. Inside the function, check if the pointer is not NULL, as we are doing every time, since this is the base case for the recursion to stop. If it is NULL, we wouldn't do anything but if it isn't we would call the same function recursively on the left subtree using the left element of the root struct.
4. After we finish visiting the left subtree, we print the data element of the root node indicating it as visited.
5. Having visited both the left subtree and the root, we now move to the right subtree and call it recursively. This completes our flow. And we are done visiting all the nodes.

```
void inOrder(struct node* root){  
    if(root!=NULL){  
        inOrder(root->left);  
        printf("%d ", root->data);  
        inOrder(root->right);  
    }  
}
```

[Copy](#)

Code Snippet 1: Creating the InOrder function

Here is the whole source code:

```
#include<stdio.h>  
#include<malloc.h>  
  
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
struct node* createNode(int data){  
    struct node *n; // creating a node pointer  
    n = (struct node *) malloc(sizeof(struct node)); // Allocating  
    memory in the heap  
    n->data = data; // Setting the data  
    n->left = NULL; // Setting the left and right children to NULL  
    n->right = NULL; // Setting the left and right children to NULL  
    return n; // Finally returning the created node  
}
```

```
void preOrder(struct node* root){  
    if(root!=NULL){  
        printf("%d ", root->data);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

```
void postOrder(struct node* root){  
    if(root!=NULL){  
        postOrder(root->left);  
        postOrder(root->right);  
        printf("%d ", root->data);  
    }  
}
```

```
void inOrder(struct node* root){  
    if(root!=NULL){  
        inOrder(root->left);  
        printf("%d ", root->data);  
        inOrder(root->right);  
    }  
}
```

```
int main(){
```

```
// Constructing the root node - Using Function (Recommended)  
    struct node *p = createNode(4);  
    struct node *p1 = createNode(1);  
    struct node *p2 = createNode(6);  
    struct node *p3 = createNode(5);
```

```

struct node *p4 = createNode(2);
// Finally The tree looks like this:
//      4
//     / \
//    1   6
//   / \
//  5   2

// Linking the root node with left and right children
p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;

// preOrder(p);
// printf("\n");
// postOrder(p);
// printf("\n");
inOrder(p);
return 0;
}

```

[Copy](#)

Code Snippet 2: Implementing the inOrder function

Now simply call all the traversal functions, preOrder, postOrder, and inOrder passing the pointer to the root node as their parameter and see how they work.

```

preOrder(p);
printf("\n");
postOrder(p);
printf("\n");
inOrder(p);

```

[Copy](#)

Code Snippet 3: Using the preOrder, the postOrder, and the inOrder functions

And the output we received was:

```
4 1 5 2 6  
5 2 1 6 4  
5 1 2 4 6
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above code

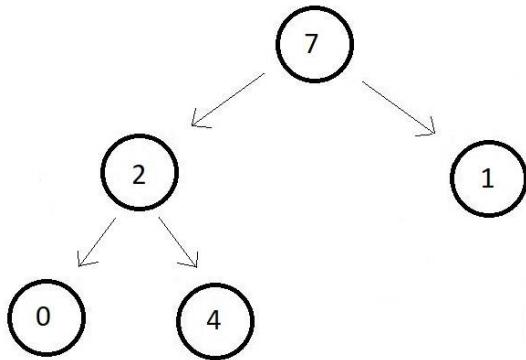
And it worked accurately to our knowledge. You should feel and observe the differences in all the traversal techniques. Things matched with our manual traversal above. You must verify the functions yourself by creating your own binary tree and by applying all the techniques. And here we finish all our traversal methods.

Best Trick To Find PreOrder, InOrder & PostOrder Traversal

In the previous tutorial, we covered our last traversal technique, the InOrder Traversal. We dealt with its programming in C. Earlier to that, we saw the other two traversal techniques. In the end, I did tell you that we will pursue a faster method to traverse through a Binary Tree. So today we'll see a faster, in fact, the best trick to find the PreOrder, the InOrder, and the PostOrder Traversal in binary trees.

This method shall help you in case you are in a hurry, or you have an assessment the next day. If it's just about marks, you can count entirely on the lecture and get good grades, I promise you. Throughout this lecture, it is assumed you have jumped right into this lecture, and haven't gone to the previous lectures of the series.

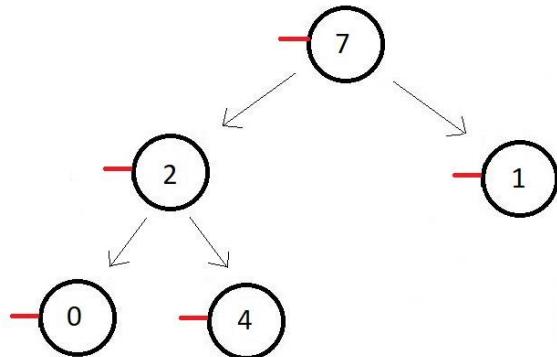
I would like to provide an example of a Binary Tree. We will then traverse through it using different techniques.



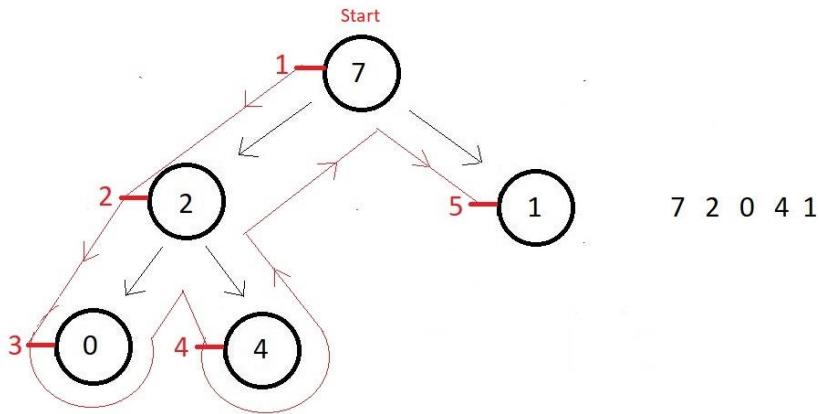
First, we'll see traversal using the PreOrder approach.

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.



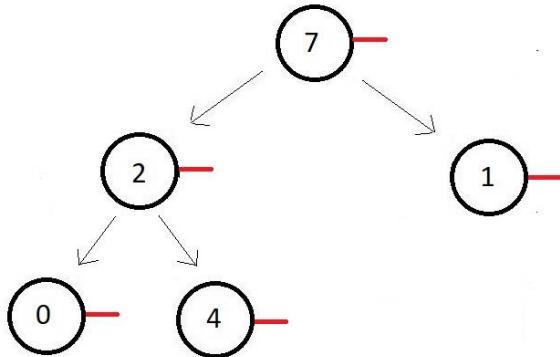
And now, start driving from the root to the left of the root node, and whenever you intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been made to direct you to the path we have followed.



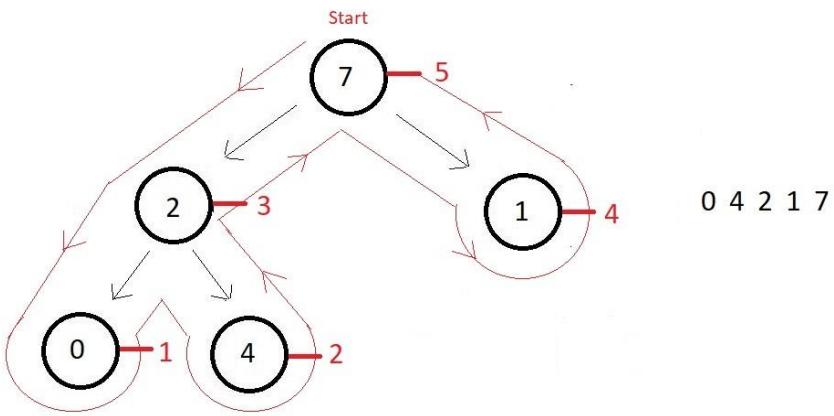
And this is how you were quickly able to write the order of nodes you visited first in the PreOrder Traversal. It was $7 \rightarrow 2 \rightarrow 0 \rightarrow 4 \rightarrow 1$. Let's see how easy PostOrder Traversal is.

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.



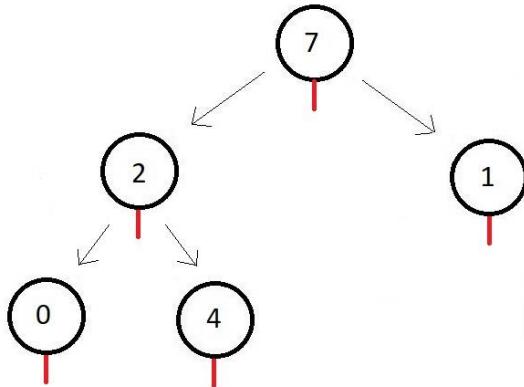
And now, drive the same way we did earlier. Start driving from the root to the left of the root node and whenever you intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been drawn to direct you to the path we have taken.



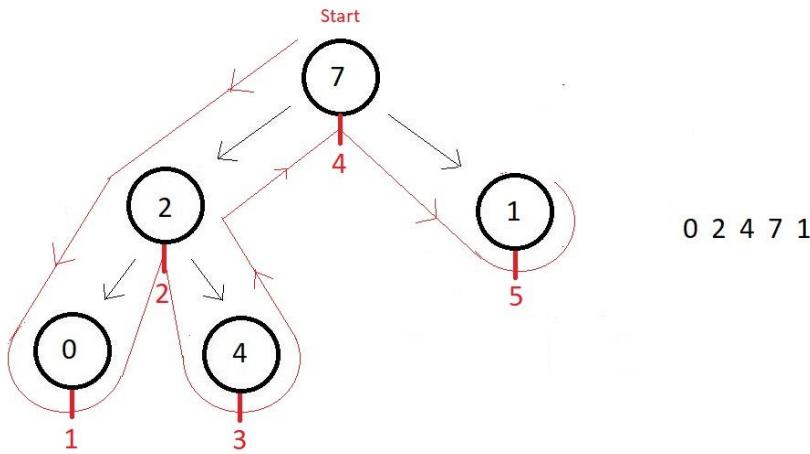
And this is how you were quickly able to write the order of nodes you visited first in the PostOrder Traversal. It was $0 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 7$. Let's see lastly how easy InOrder Traversal is.

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 you intersect a red edge while driving, print its value. Refer to the illustration below. Arrows have been drawn to direct you to the path we have taken.



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

You can do one thing now. Create the binary tree in the above example and use the programs we created in earlier lectures for each of these traversals to verify if this trick actually works. And I can assure you the authenticity of this trick for all the larger binary trees you will get in your assessments. Use this blindly. But make sure you cover up the concepts sooner or later. Because that is what helps you in the end.

Binary Search Trees: Introduction & Properties

In the last lecture, we finished learning about all the traversal methods in binary trees. To make it easier for you, we even found a way to keep track of the order of nodes you visit in every traversal method. Today, we'll start learning about what a binary search tree is and what features they possess.

Earlier we concluded a few points about binary trees. Binary trees are normal trees with each parent having either two or less than two children, that is the degree of the tree should be two or less than two.

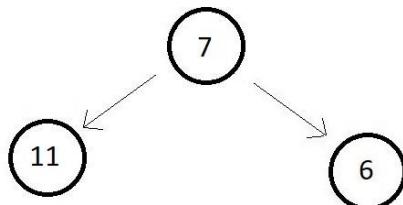


Figure 1

Tell me if the above tree is a binary tree or not. Of Course, it is. Since all the nodes have degrees either two or zero. But is this binary tree a binary search tree? Wait. A

distinction between binary trees and binary search trees is what I am seeking to establish. Following are the properties of a binary search tree:

Binary Search Trees:

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.

Having discussed all the properties, you must now tell me if the above binary tree was a binary search tree or not. The answer should be no. 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. Let me take one more example.

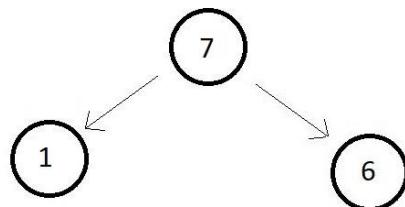


Figure 2

Tell me 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. Actually, I have always found you all to be smart enough to answer these petty questions. Let's consider a big one and see if it's a binary search tree or not.

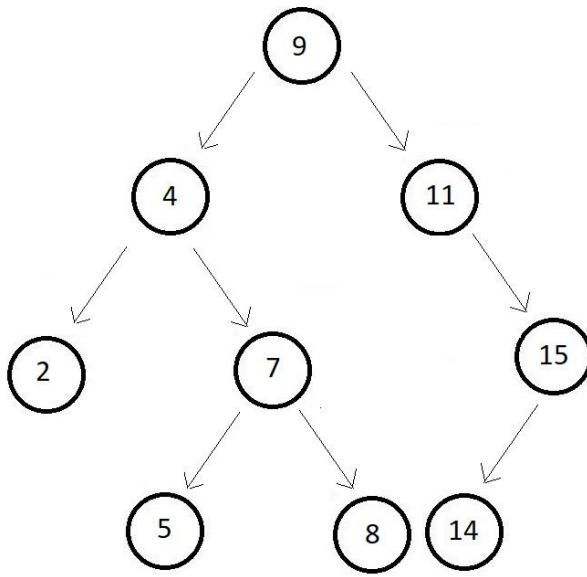


Figure 3

Take your time and analyze the different properties of a binary search tree and tell if this is a binary search tree or not.

Solution: YES.

Why?

The very first thing to observe here is the properties of a Binary Search Tree. You would check if all the properties are satisfied for each of the nodes of the tree. So, you 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. Had there been even one violation for any of the nodes, we would have said, no.

Lastly, there is one more amazing property that I've been keeping all along to amaze you. It 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.

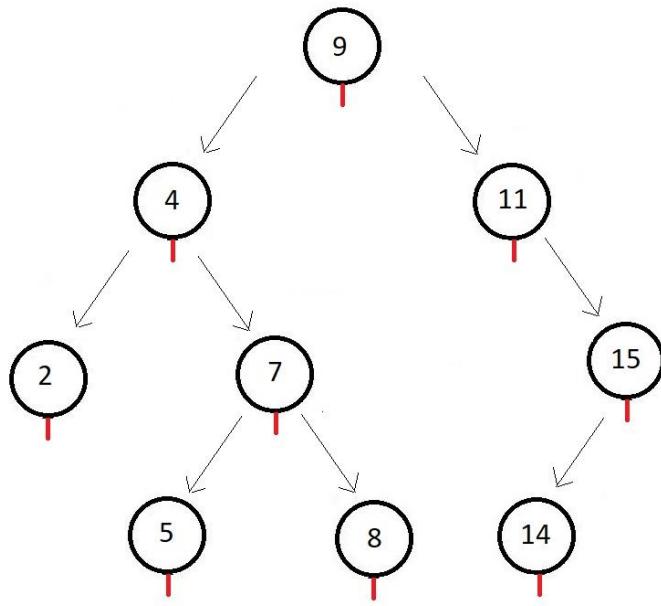


Figure 4.

Now, we'll start from the root node, and drive towards the left subtree and follow the arrows I've drawn for your convenience.

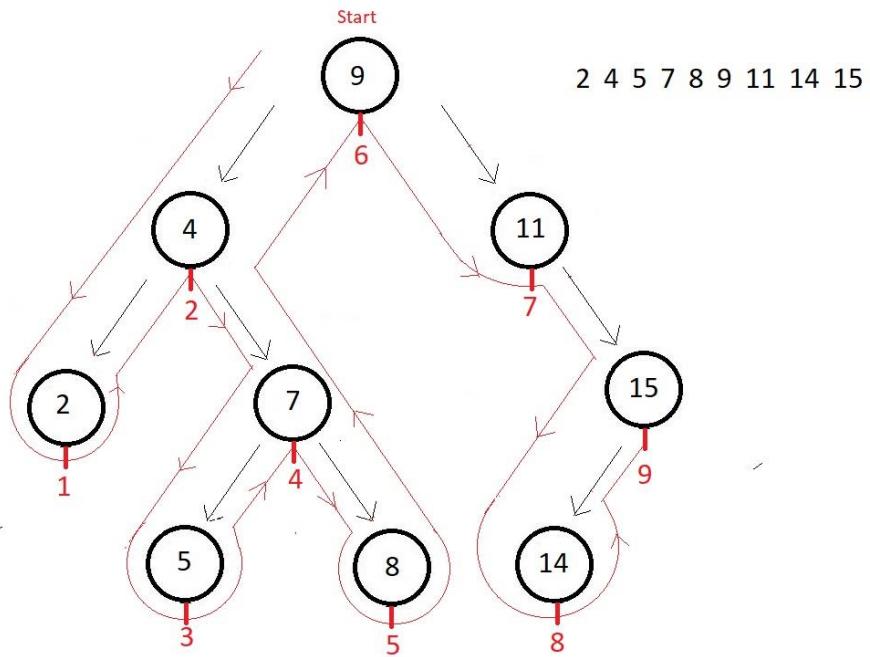


Figure 5

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. And this is how easy we have made checking if a tree is a binary search tree or not.

We still haven't seen how to make a binary search tree which shall be our next job. We'll also make programs for creating a binary search tree, and for checking if a tree is a binary search tree or not. This was all about the basics of binary search trees. We introduced this to you and discussed its most important properties.

Checking if a binary tree is a binary search tree or not!

In the last lecture, we got to know about one more variety of binary tree, called binary search trees. We discussed all the properties of a binary search tree. We drew a fine line between binary trees and binary search trees. Today, we'll delve into their differences and distinguish a binary search tree from a normal binary tree. And check if a binary tree is a binary search tree or not.

But before that let's briefly revise the properties of a binary search tree once.

Properties of a binary search tree:

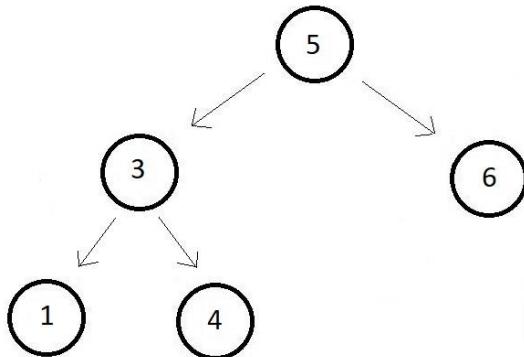
1. All nodes of the left subtree are lesser than the node itself.
2. All nodes of the right subtree are greater than the node itself.
3. Left and Right subtrees are also binary trees.
4. There are no duplicate nodes.
5. The InOrder traversal of a binary search tree gives an ascending sorted array.

And the last one is of utmost importance to us. And that is what we use the most to check if a binary tree is a binary search tree or not. So, basically, we'll check if a binary tree is a binary search tree or not by making an InOrder Traversal in the tree and analyzing its order. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. Since we'll create a binary search tree at first which is nothing but a binary tree only and we'll use the InOrder traversal function as well, we'll simply copy everything we did in our last programming tutorial where we learned to create the function inOrder. And we did this because we avoided doing repetitions in the course and starting from scratch, creating the whole struct Node and the createNode thing again and all the functions would have made the lecture redundant. And this has also saved us a lot of time.
2. I just hope that you've followed all the way from the start of the Binary Search lecture and not just jumped right here. Make sure you check them all before proceeding.

You should now be able to create Binary Tree yourselves. Create the binary search tree I've illustrated below using the createNode function. And if you are still not sure about creating a binary search tree, follow the previous lectures.



3. Let's now just create a function to check if the InOrder traversal of the binary tree is in ascending order or not.

Creating the function isBST:

4. Create an integer function `isBST` and pass the pointer to the root node of the tree you want to check as the only parameter. Inside the function, check if the pointer is not NULL, as we have been checking the whole time, and this is also considered as the base case for the recursion to stop. If it is NULL, we would simply return 1 since an empty tree is always a binary search tree. Else, this is a complex yet understandable part. You should follow what I am saying.
5. Create a static struct Node pointer `prev` initialised with NULL. This maintains the pointer to the parent node. And since the root node doesn't have any parent, we have initialized it with NULL and made it static.
6. Now, see if the left subtree is a Binary Search Tree or not, by calling it recursively. If it is not a BST, return 0 here itself. Else, see if the `prev` is not NULL otherwise this is the root node of the whole tree and we won't check this condition. If the `prev` node is not NULL and the current node, which is the root node of the current subtree, is smaller than or equal to the `prev` node, then we would return 0. Since this violates the increasing orderliness.
7. If it still passes all the if conditions we have structured above, we will store the current node in the `prev` and move it recursively to the right subtree. And this is nothing but a modified version of the InOrder Traversal.

```

int isBST(struct node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
    }
}
  
```

```

    }

    if(prev!=NULL && root->data <= prev->data){

        return 0;

    }

    prev = root;

    return isBST(root->right);

}

else{

    return 1;

}

}

```

[Copy](#)

Code Snippet 1: Creating the isBST function

I suggest ignoring the recursion and do not trace how the function isBST works if the function isBST is at all confusing. Just note that we have done nothing but the InOrder traversal of the tree, and we have checked if the previous value is smaller than the current value, that's it.

Note: Static variables are used when we don't want our value for that variable to change every time that function is called.

Here is the whole source code:

```

#include<stdio.h>

#include<malloc.h>

struct node{

    int data;

    struct node* left;

    struct node* right;

};

struct node* createNode(int data){

    struct node *n; // creating a node pointer

```

```
n = (struct node *) malloc(sizeof(struct node)); // Allocating  
memory in the heap  
n->data = data; // Setting the data  
n->left = NULL; // Setting the left and right children to NULL  
n->right = NULL; // Setting the left and right children to NULL  
return n; // Finally returning the created node  
}
```

```
void preOrder(struct node* root){  
    if(root!=NULL){  
        printf("%d ", root->data);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

```
void postOrder(struct node* root){  
    if(root!=NULL){  
        postOrder(root->left);  
        postOrder(root->right);  
        printf("%d ", root->data);  
    }  
}
```

```
void inOrder(struct node* root){  
    if(root!=NULL){  
        inOrder(root->left);  
        printf("%d ", root->data);  
        inOrder(root->right);  
    }  
}
```

```

int isBST(struct node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}

```

```
int main(){
```

```

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);

    // Finally The tree looks like this:
    //      5
    //     / \
    //    3   6
    //   / \
    //  1   4

```

```

// Linking the root node with left and right children
p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;

// preOrder(p);
// printf("\n");
// postOrder(p);
// printf("\n");
inOrder(p);
printf("\n");
// printf("%d", isBST(p));
if(isBST(p)){
    printf("This is a bst" );
}
else{
    printf("This is not a bst");
}
return 0;
}

```

[Copy](#)

Code Snippet 2: Implementing the isBST function

Let's now check if our function actually works, and it reverts whether our binary tree is a BST or not. We would also like to print the InOrder traversal of the tree.

```

inOrder(p);
printf("\n");
if(isBST(p)){
    printf("This is a bst" );
}

```

```
    }
else{
    printf("This is not a bst");
}
```

[Copy](#)

Code Snippet 3: Using the `isBST` function

And the output we received was:

```
1 3 4 5 6
This is a bst
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above code

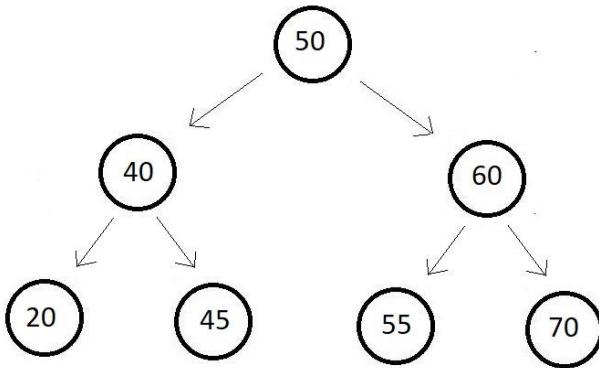
And yes, it says, our function is a Binary Search Tree and that was what we expected. The InOrder traversal is also in ascending order. So, things are pretty accurate and consistent.

Searching in a Binary Search Trees (Search Operation)

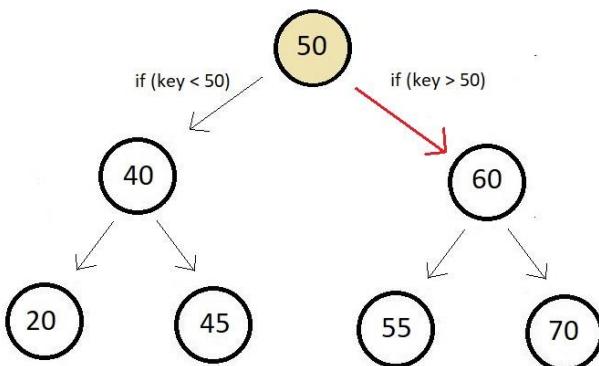
In the last tutorial, we saw the implementation of the `isBST` function in C. We thoroughly discussed the programming. We saw the modifications we had to make in the `inOrder` function to check if the InOrder traversal is in ascending order or not, and in fact, to check if a binary tree is a binary search tree or not. And today, we are going to look after our first binary search tree operation, the **search operation**. We'll see if any given key lies in the binary search tree or not.

Now you might be wondering what exactly is the significance of learning about binary search trees since binary trees were equally good. But let me tell you that one of the major applications of using a binary search tree, is to be able to search some key in the tree in **log_n** time complexity in the best case where n is the number of nodes. Each time you compare a node with your key, you divide the search space to its half. But let's not proceed without an example.

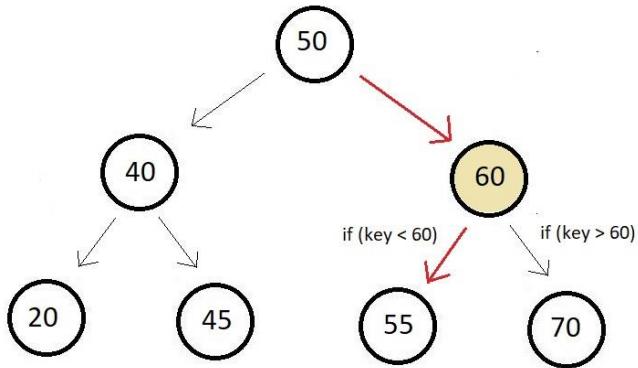
For our purpose of understanding, we will first examine a sample binary search tree. Suppose we have a Binary Search Tree illustrated below.



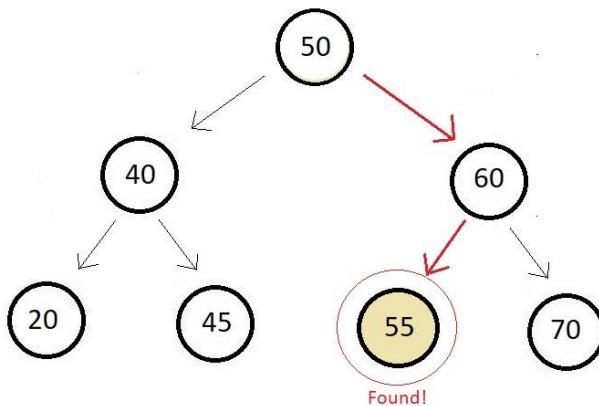
And let's say the key we want to search in this binary search tree is 55. Let's start our search. So, we'll first compare our key with the root node itself, which is 50.



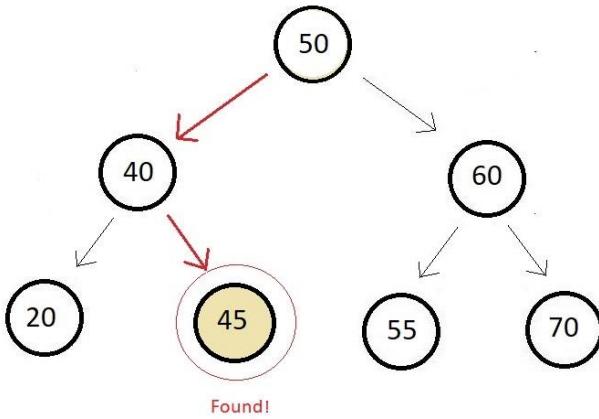
But since 50 is less than 55, which side should we proceed with? Left, or Right? Of course, right. Since all the elements in the right subtree of a node are greater than that node, we'll move to the right. The first element we check our key with is 60.



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, I'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. Let's analyze the time complexity of the searching algorithm we have discussed above.

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

If you remember, we had studied an algorithm called the Binary Search. We could use that algorithm only if the array we were searching for some key in was sorted. And that algorithm had the time complexity of $O(\log n)$ where n was the length of the array. Because we were always dividing our search space into half on the basis of whether our key was smaller or greater than the mid . And as you might have guessed, searching in a binary search tree is very much similar to that.

Searching in a binary search tree holds $O(\log n)$ 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 you 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 $T \propto h$, that is, our time complexity is proportional to the height of the tree. Therefore, the time complexity becomes $O(h)$.

Now, if you remember, the height of a tree ranges from $\log n$ to n , that is

$$(\log n) \leq h \leq n$$

So, the best-case time complexity is $O(\log n)$ and the worst-case time complexity is $O(n)$.

Our next aim would be to automate the searching process. The way we compared the nodes and decided to move to the left or to the right needs to be programmed. So today I'll just brief you all about the way we program this search operation and will implement them in C in the next lecture. We will also write pseudocode for better understanding.

Pseudocode for searching in a Binary Search Tree:

1. There will be a struct node pointer function *search* which will take the pointer to the root node and the key you want to search in the tree. And before you do anything, just check if the root node is not NULL. If it is, return NULL here itself. Otherwise, proceed further.
2. Now, check if the node you are at is the one you 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.

```
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);
    }
}
```

Figure 1: Pseudocode for the search function

We'll see the programming segment in the next lecture in detail. We will make use of our IDEs. And program our own and run them to actually search for a key in a binary search tree.

C Code For Searching in a BST

In the last lecture, we saw one important application of binary search trees. It is its search operation that allows us to search any key in the binary search tree in $\log n$ best case time complexity. We saw how searching is done by reducing the search space after every comparison, and how searching time is proportional to the height of the tree. Today, we'll see the programming of this search function in C and automate the searching process.

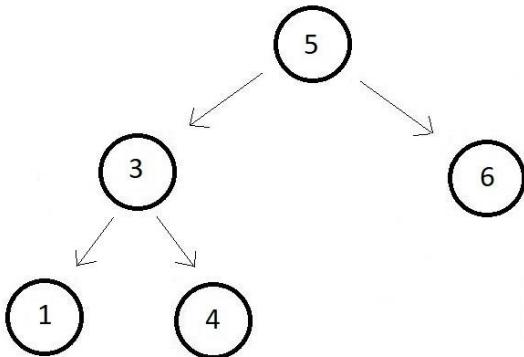
We did see the pseudocode of the search function in the last class. Follow it while we understand the procedure. I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. First of all, we'll anyway have to create a binary search tree to be able to use it to search some keys. And since we created everything the last time, we'll just copy everything we have done till the last programming lecture. It covered everything from creating a node to building a tree. It also features all the traversal methods

and the isBST function. We copied everything so as to keep everything in one place, and to avoid repeating things in the course, and this has also saved us a lot of time.

2. Let's create a binary search tree I've illustrated below using the createNode function. Creating a binary search tree should be an issue anymore. And if you are still not sure about creating a binary search tree, follow the previous lectures.



3. We'll do each of the other operations we usually do. But today our focus is on the search operation. Let's now just create a function to search a key in the binary search tree we created.

Creating the function search:

4. Create a struct Node pointer function `search` and pass into it the pointer to the root and the key you want to search as two of its parameters.
5. First of all, check if the root is NULL. If the root is NULL, we haven't found our key, and we'll simply return NULL.
6. Now, check if the node we are currently at has the data element equal to the key we were searching for. If it is the case, we have found the key, and we'll simply return the pointer to the current root.
7. But if we still haven't found the key, it is probably in the left subtree of the root, or in the right subtree of the root. To judge which side to proceed with, we'll check if the current root is less than or greater than the data element of the root. If it is less than the root, the key probability lies on the left subtree, and hence we would reduce our search space and recursively start searching in the left subtree. Otherwise, we would start searching in the right subtree. Anyways, we are reducing our search space after each comparison.

```
struct node * search(struct node* root, int 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);
    }
}

```

[Copy](#)

Code Snippet 1: Creating the search function

Recursion in the above function is pretty simple this time, unlike the isBST function we did early. Here, we simply decide every time which subtree to pursue next. And recursively go with that.

Here is the whole source code:

```

#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    struct node* left;
    struct node* right;
};

struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating
    memory in the heap
}

```

```
n->data = data; // Setting the data
n->left = NULL; // Setting the left and right children to NULL
n->right = NULL; // Setting the left and right children to NULL
return n; // Finally returning the created node
}
```

```
void preOrder(struct node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
void postOrder(struct node* root){
    if(root!=NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}
```

```
void inOrder(struct node* root){
    if(root!=NULL){
        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}
```

```
int isBST(struct node* root){
```

```
static struct node *prev = NULL;

if(root!=NULL){
    if(!isBST(root->left)){
        return 0;
    }
    if(prev!=NULL && root->data <= prev->data){
        return 0;
    }
    prev = root;
    return isBST(root->right);
}
else{
    return 1;
}
```

```
struct node * search(struct node* root, int 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);
    }
}
```

```
int main(){

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);

    // Finally The tree looks like this:
    //      5
    //     / \
    //    3   6
    //   / \
    //  1   4

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;

    struct node* n = search(p, 10);
    if(n!=NULL){
        printf("Found: %d", n->data);
    }
    else{
        printf("Element not found");
    }
    return 0;
}
```

Copy

Code Snippet 2: Implementing the search function

Now, make sure you store the value returned by the search function in a struct node pointer, say n. We'll see if our function actually works, and it reverts the pointer to the node it found, or a NULL if it didn't find the key. Let's search 10, in the above function.

```
struct node* n = search(p, 10);
if(n!=NULL){
    printf("Found: %d", n->data);
}
else{
    printf("Element not found");
}
```

[Copy](#)

Code Snippet 3: Using the search function

And the output we received was:

```
Element not found
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above code

And since 10 was not there in the binary search tree we created, the function `search` returned a NULL. Let's make it work for something which is there. Let's use the function to find 3 in the above BST.

```
struct node* n = search(p, 3);
if(n!=NULL){
    printf("Found: %d", n->data);
}
else{
    printf("Element not found");
}
```

[Copy](#)

Code Snippet 4: Using the search function again

And the output we received was:

```
Found: 3
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 2: Output of the above code

And yes, it says, our function search found the key 3 in the above binary search tree. It was simple to digest I believe.

Iterative Search in a Binary Search Tree

In the last lecture, we learned how to program the `search` function to be able to search keys in a binary search tree. The function returns the pointer to the node if it finds the key, otherwise returns NULL. We saw how easy binary search trees make the process of searching and offers a best-case runtime complexity of $\log n$.

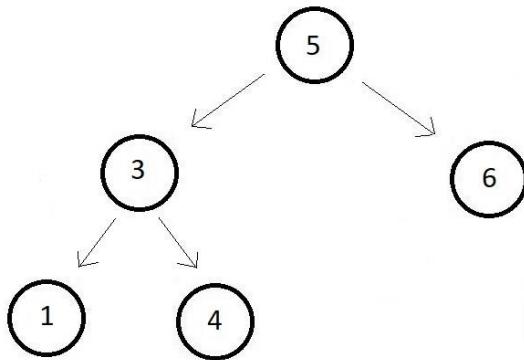
The `search` function we studied was a **recursive** function, calling the function recursively on the left or the right subtree. But today, we'll see the concept of searching **iteratively** in a binary search tree and its programming in C.

In the recursive approach, you start with the root and compare your key with the root node's data, and if it's smaller than the key, you select the whole left subtree and start searching in it thinking of it as another smaller tree. But in an iterative approach, we have the tree intact the whole time. We just make a pointer run from the root to the place we expect our key to be. Let's directly see the code for iterative search.

I have attached the source code below. Follow it as we proceed.

Understanding the code snippet below:

1. There is nothing much to explain in the code. Although for the people coming right here, we have just copied everything we had done till the last programming lecture which dealt with the search function. It had covered everything from creating a node, to building a tree. It also featured all the traversal methods, and all the other functions. We did this to save ourselves some time.
2. Let's create a binary search tree I've illustrated below using the `createNode` function. And without doing any further ado, we would move to creating the `searchIter`



Creating the function `searchIter`:

3. Create a struct Node pointer function `searchIter` and pass into it the pointer to the root and the key you want to search as two of its parameters.
4. First of all, check if the root's data itself is the key we were looking for. If the root data is equal to the key, we have found the key, and we'll return the pointer to the root.
5. If we couldn't find the key on the root, we'll further see if our key is greater than or smaller than the root data. If it is smaller than the root data, we will make the root node pointer now point to the left child of the root, using the arrow operator. And if the key is greater than the root data, we'll make the root node pointer point to the right child of the root
6. And we'll use a while loop to run steps 4 and 5 iteratively until our root becomes NULL, or we find the key and return from the function.
7. And in the end, if we couldn't find the key after iterating through the tree, or the root we received in the starting itself was a NULL, we would return NULL.

```

struct node * searchIter(struct node* root, int key){
    while(root!=NULL){
        if(key == root->data){
            return root;
        }
    }
  
```

```

        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    return NULL;
}

```

[Copy](#)

Code Snippet 1: Creating the searchIter function

Iteration is more intuitive because you are able to see what's exactly happening and how things end. Even so, recursion is considered a powerful tool.

[Here is the whole source code:](#)

```

#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    struct node* left;
    struct node* right;
};

struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating memory in the heap
    n->data = data; // Setting the data
}

```

```
    n->left = NULL; // Setting the left and right children to  
NULL  
  
    n->right = NULL; // Setting the left and right children to  
NULL  
  
    return n; // Finally returning the created node  
}
```

```
void preOrder(struct node* root){  
    if(root!=NULL){  
        printf("%d ", root->data);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

```
void postOrder(struct node* root){  
    if(root!=NULL){  
        postOrder(root->left);  
        postOrder(root->right);  
        printf("%d ", root->data);  
    }  
}
```

```
void inOrder(struct node* root){  
    if(root!=NULL){  
        inOrder(root->left);  
        printf("%d ", root->data);  
        inOrder(root->right);  
    }  
}
```

```
}
```

```
int isBST(struct node* root){  
    static struct node *prev = NULL;  
    if(root!=NULL){  
        if(!isBST(root->left)){  
            return 0;  
        }  
        if(prev!=NULL && root->data <= prev->data){  
            return 0;  
        }  
        prev = root;  
        return isBST(root->right);  
    }  
    else{  
        return 1;  
    }  
}
```

```
struct node * searchIter(struct node* root, int key){  
    while(root!=NULL){  
        if(key == root->data){  
            return root;  
        }  
        else if(key<root->data){  
            root = root->left;  
        }  
        else{  
            root = root->right;  
        }  
    }  
}
```

```
    }
}
return NULL;
}
```

```
int main(){
```

```
// Constructing the root node - Using Function  
(Recommended)
```

```
struct node *p = createNode(5);  
struct node *p1 = createNode(3);  
struct node *p2 = createNode(6);  
struct node *p3 = createNode(1);  
struct node *p4 = createNode(4);
```

```
// Finally The tree looks like this:
```

```
//      5  
//      / \br/>//      3   6  
//      / \br/>//      1   4
```

```
// Linking the root node with left and right children
```

```
p->left = p1;  
p->right = p2;  
p1->left = p3;  
p1->right = p4;
```

```
struct node* n = searchIter(p, 6);  
if(n!=NULL){
```

```
    printf("Found: %d", n->data);  
}  
else{  
    printf("Element not found");  
}  
return 0;  
}
```

Copy

Code Snippet 2: Implementing the searchIter function

Now, make sure you store the value returned by the searchIter function in a struct node pointer, say n. We'll see if our function actually works, and it reverts the pointer to the node it found, or a NULL if it didn't find the key. Let's search 10, in the above function. We did all this time as well.

```
struct node* n = searchIter(p, 10);  
if(n!=NULL){  
    printf("Found: %d", n->data);  
}  
else{  
    printf("Element not found");  
}
```

Copy

Code Snippet 3: Using the searchIter function

And the output we received was:

```
Element not found  
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

Copy

Figure 1: Output of the above code

And since 10 was not there in the binary search tree we created, the function `searchIter` returned a NULL. Let's see if it works for something else. Let's use the function to find 6 in the above BST.

```
struct node* n = searchIter(p, 6);
if(n!=NULL){
    printf("Found: %d", n->data);
}
else{
    printf("Element not found");
}
```

Copy

Code Snippet 4: Using the searchIter function again

And the output we received was:

```
Found: 6
```

```
PS D:\MyData\Business\code playground\Ds & Algo with  
Notes\Code>
```

Copy

Figure 2: Output of the above code

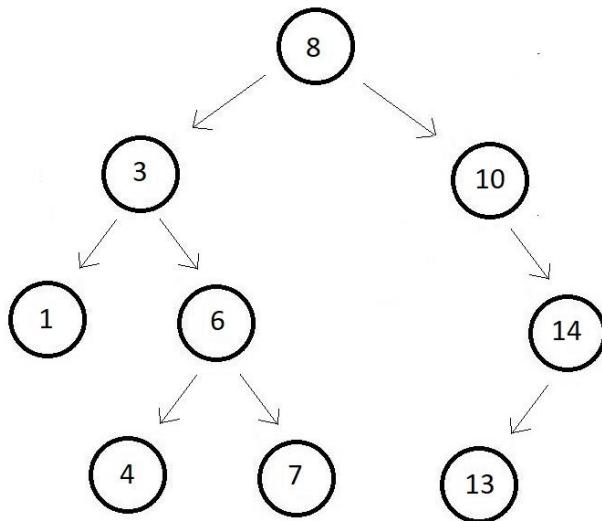
Ah yes. The function is working all good. Although we changed our approach from recursive to iterative, things remained pretty much the same. We still stop our searching after reaching the NULL. So, more or less the base case and the flow remain the same. And it did feel more intuitive.

Insertion in a Binary Search Tree

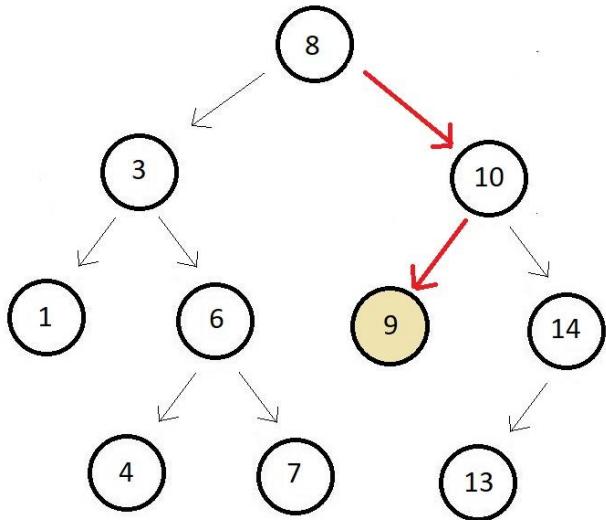
In the last lecture, we learned how to program the search function which will search for a key the user gives in the binary search tree, and revert its position if it is present in the tree, else return NULL. Today we'll learn our second operation in binary search trees which is the insertion in a binary search tree. Inserting in a binary search tree is really not a big deal, but yet very important.

Your half the job is already done if you have diligently followed the search operation. I am saying this because in the algorithm we will follow to insert an element, there are a lot of things in common to the algorithm we learn to search an element. We know a few facts about binary search trees. And I'll only mention the ones we will focus on, while we learn the insertion operation.

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



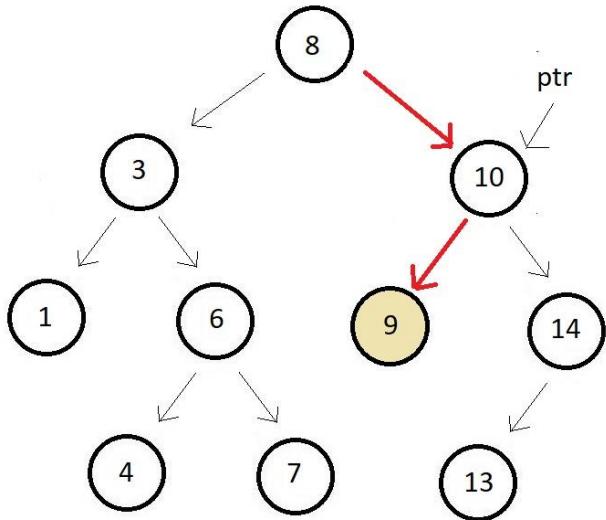
Now, you would simply start from the root node, and see if the element you want to insert is greater than or less than. And since 9 is greater than 8, we move to the right of the root. And then the root is the element 10, and since this time 9 is less than 10, we move to the left of it. And since there are no elements to its left, we simply insert element 9 there.



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

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

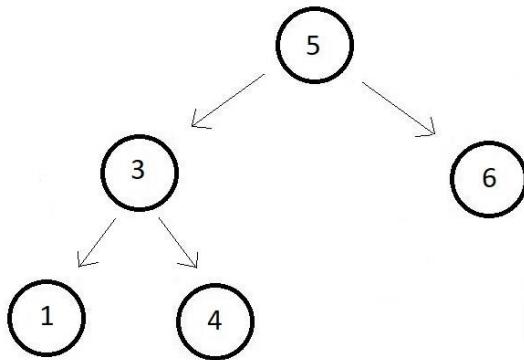
And another important thing to see here is the pointer you would follow the correct position with. In the above example, to be able to insert at that position, the pointer must be at node 10.



And then you check whether going to the left side is good, or the right. Here you 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. We'll see all this via our program. That would actually make things simpler to understand. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. Now, since our main focus would be to create the `insert` function, we could just copy everything we did in the last programming lecture where we learned the iterative search operation. And this would save us a lot of time. And we would have the `createNode` and other functions we did before at one place for our exigency.
2. Create a binary search tree of your choice, we would rather go with the one we already had in the program. Now, let's see the `insert`



Creating the *insert* function:

3. So, the first thing we would like to know is whether inserting this new node is even possible or not. For that, create a void function *insert* and pass the pointer to the root node, and the data of the node you want to insert as its parameters. We will call it
4. Now, we would use two struct pointers to traverse through the array. One of them would be our *root* which we would traverse through the nodes, and the other one would be *prev* which stores the pointer to the previous root. SO, just create a struct Node pointer *prev* to maintain the node you were previously at, at some point in time.
5. Run a while loop that is for until we reach some leaf, and couldn't traverse further. So, run that loop until the root becomes NULL. And inside that loop, make the *prev* equal to the current root since we would definitely move further because this root is not a NULL. We would either move to the left of this root or to the right of this root. But before that check, if this root itself is not equal to the node we are trying to insert. That is, write an if condition to see if there are any duplicates here. If there is, return from the function here itself.
6. Further in the loop, check if the element you want to insert is less than the current root. If it is, update the root to the left element of the struct root. And if it isn't, update the root to the right element of the struct root. And since we have already stored this root in the *prev* node, there isn't any issue updating.
7. And finally, you will have a *prev* node as the outcome at the end after this loop finishes. Now, the only procedure left now is to link these nodes together, that is the *prev* node, the *new* node, and the node *next* to the *prev*
8. Now, before you insert, make sure you create that new struct node using malloc, or simply the *createNode*. Fill in the *key* data into this *new* node. Now, simply check if the *prev->data* is less than the *key* or greater than the *key*. If it is less, insert our new node to the left of *prev*, else to right of *prev*. And that would be it. We are done inserting our new node.

```
void insert(struct node *root, int key){
```

```

    struct node *prev = NULL;
    while(root!=NULL){
        prev = root;
        if(key==root->data){
            printf("Cannot insert %d, already in BST", key);
            return;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    struct node* new = createNode(key);
    if(key<prev->data){
        prev->left = new;
    }
    else{
        prev->right = new;
    }
}

```

[Copy](#)

Code Snippet 1: Creating the *insert* function

Here is the whole source code:

```

#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
}

```

```
    struct node* left;
    struct node* right;
};

struct node* createNode(int data){
    struct node *n; // creating a node pointer
    n = (struct node *) malloc(sizeof(struct node)); // Allocating
    memory in the heap
    n->data = data; // Setting the data
    n->left = NULL; // Setting the left and right children to NULL
    n->right = NULL; // Setting the left and right children to NULL
    return n; // Finally returning the created node
}

void preOrder(struct node* root){
    if(root!=NULL){
        printf("%d ", root->data);
        preOrder(root->left);
        preOrder(root->right);
    }
}

void postOrder(struct node* root){
    if(root!=NULL){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->data);
    }
}

void inOrder(struct node* root){
    if(root!=NULL){
```

```

        inOrder(root->left);
        printf("%d ", root->data);
        inOrder(root->right);
    }
}

int isBST(struct node* root){
    static struct node *prev = NULL;
    if(root!=NULL){
        if(!isBST(root->left)){
            return 0;
        }
        if(prev!=NULL && root->data <= prev->data){
            return 0;
        }
        prev = root;
        return isBST(root->right);
    }
    else{
        return 1;
    }
}

struct node * searchIter(struct node* root, int key){
    while(root!=NULL){
        if(key == root->data){
            return root;
        }
        else if(key<root->data){
            root = root->left;
        }
    }
}

```

```
        else{
            root = root->right;
        }
    }

    return NULL;
}

void insert(struct node *root, int key){
    struct node *prev = NULL;
    while(root!=NULL){
        prev = root;
        if(key==root->data){
            printf("Cannot insert %d, already in BST", key);
            return;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    struct node* new = createNode(key);
    if(key<prev->data){
        prev->left = new;
    }
    else{
        prev->right = new;
    }
}
```

```

int main(){

    // Constructing the root node - Using Function (Recommended)
    struct node *p = createNode(5);
    struct node *p1 = createNode(3);
    struct node *p2 = createNode(6);
    struct node *p3 = createNode(1);
    struct node *p4 = createNode(4);

    // Finally The tree looks like this:
    //      5
    //     / \
    //    3   6
    //   /   \
    //  1   4

    // Linking the root node with left and right children
    p->left = p1;
    p->right = p2;
    p1->left = p3;
    p1->right = p4;

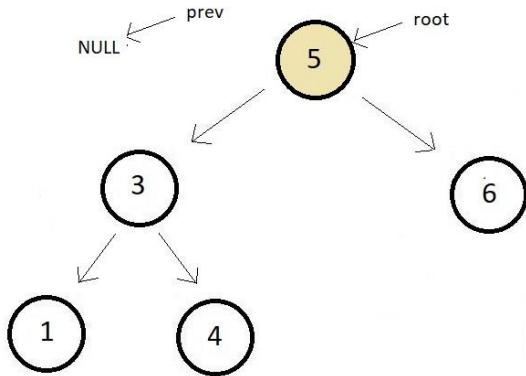
    insert(p, 16);
    printf("%d", p->right->right->data);
    return 0;
}

```

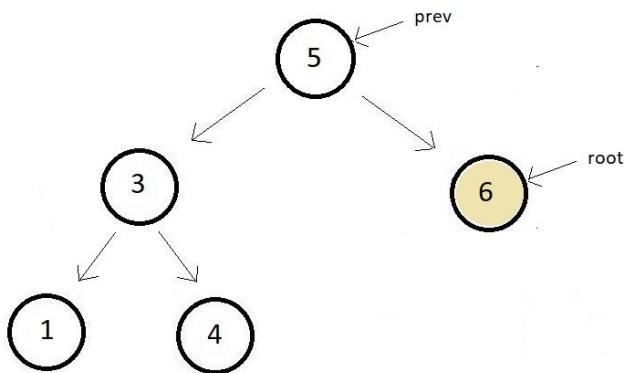
[Copy](#)

Code Snippet 2: Implementing the *insert* function in C

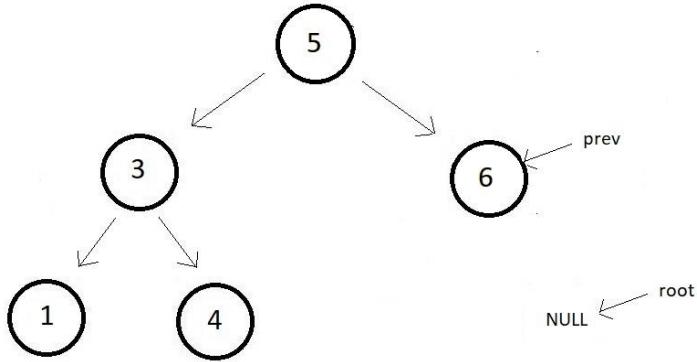
Now, suppose we want to insert element 7 in the above BST we had in the program. Now, let's first dry run that ourselves. And then we'll verify the position of the inserted node via the program. So, firstly, *prev* is NULL. And we start driving in the loop.



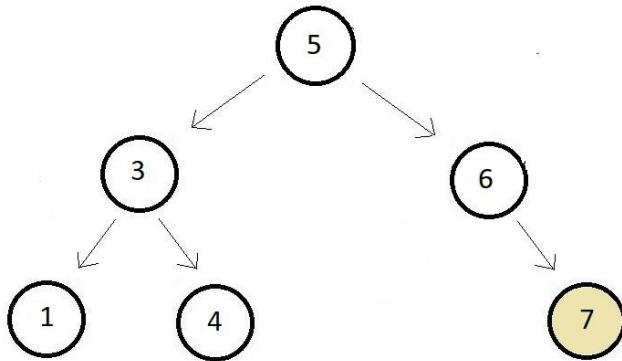
Now, first, we'll make our *prev* point to the root node, and since 7 is greater than 5, we move to the right of the root.



Now, we'll again update our *prev* with root, and since root is still smaller than 7, we'll move to the root's right which is a NULL. So, our loop ends here, and we get node 6 as our *prev* node.



And since 6 is smaller than 7, we'll insert node 7 to the right of node 6. And hence our element 7 gets placed at the position relative to the root as root -> right -> right.



Now, let's verify this by running the `insert` function, and print the data at the root -> right -> right position.

```

insert(p, 7);
printf("%d", p->right->right->data);
  
```

[Copy](#)

Code Snippet 3: Using the `insert` function

And the output we received was:

7

Copy

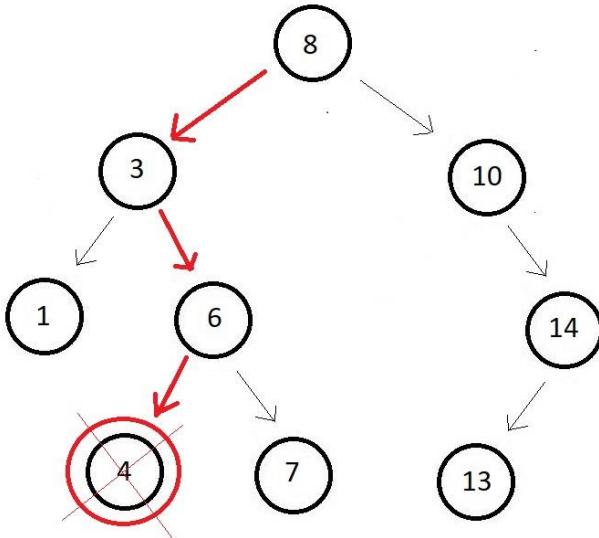
Figure 1: Output of the above code

Heh, so that got inserted with this ease. And that is all we had in the insertion operation. It is possible to learn this on your own if you practice. Next, we have the deletion function. Having learnt the insertion and the search operations would make learning deletion a cakewalk. You just have to be patient with me.

Deletion in a Binary Search Tree

In the last lecture, we saw insertion in a binary search tree. We learned how to program the insertion function in C. We drew similarities between the insertion and the search operation making it easy to digest. Today, we'll learn our third operation, the deletion operation in a binary search tree.

You might be wondering how big a deal is deleting some nodes in a binary search tree. Well, for cases like the one I have drawn below, where you'll be asked to remove say element 4 seems quite an easy job. You'll just search for the element and remove it.



But deleting in a binary search tree is no doubt a tough job like if you consider a case where the node, we'll have to remove might not be a leaf node or the node is the root node. But you should not be the one worrying here. I have some easy way out. We'll explore them in detail now.

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

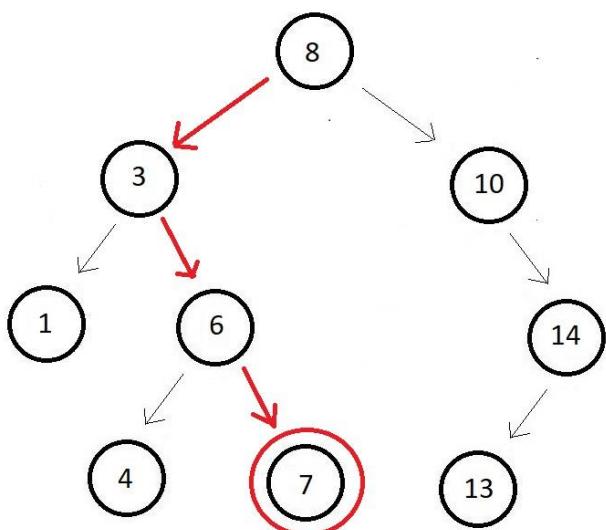
1. The node is a leaf node.
2. The node is a non-leaf node.
3. The node is the root node.

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

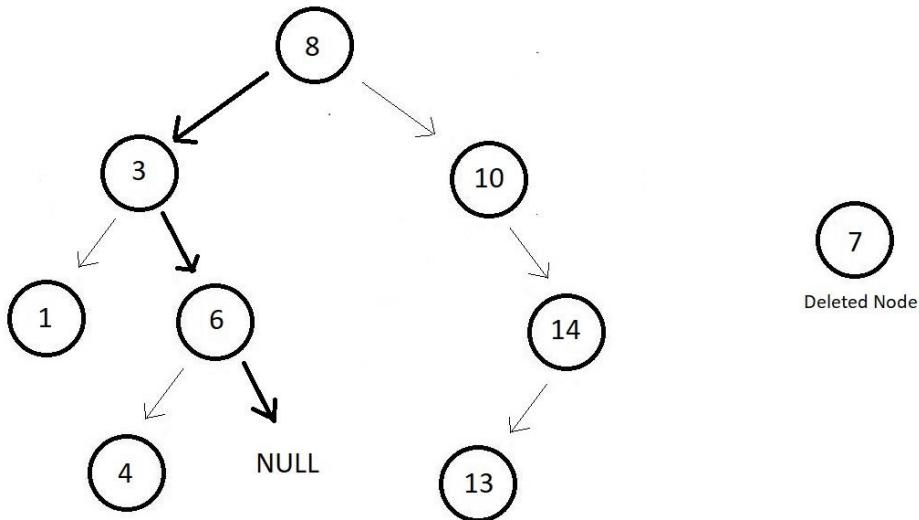
Deleting a leaf node:

Deleting a leaf node is the simplest case in deletion in binary search trees where the only thing you 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.



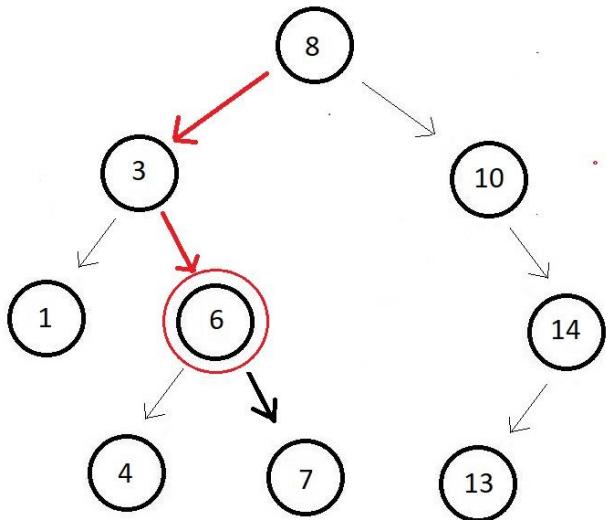
2. Delete the node.



Deleting a non-leaf node:

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

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

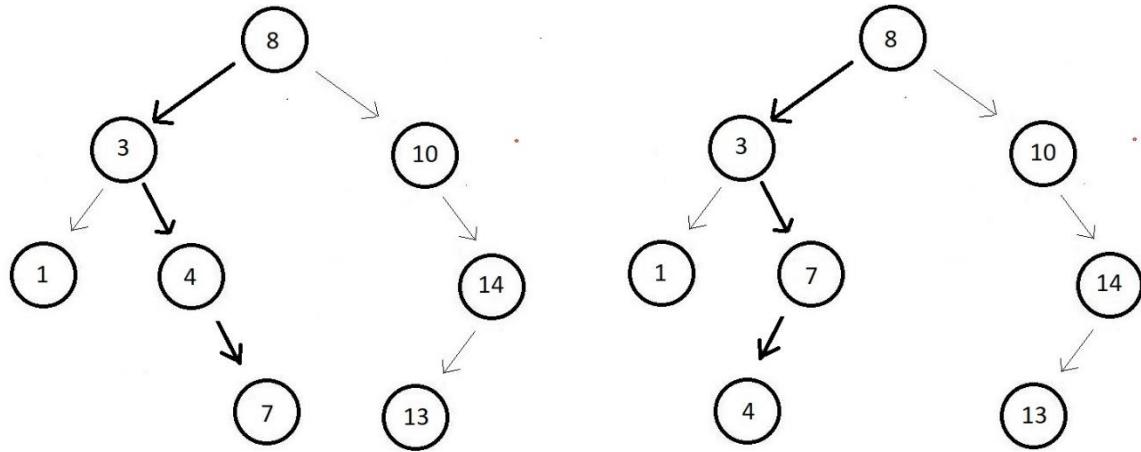


2. Now the dilemma is, which node will replace the position of node 6. Well, there is a simple answer to it. It says, when you delete a node that is not a leaf node, you replace its position with its **InOrder predecessor or Inorder successor**.

So, what does that mean? It means that if you write the InOrder traversal of the above tree, which I have already taught in my previous lectures, the nodes coming immediately before or after node 6, will be the one replacing it. So, if you write the InOrder traversal of the tree, you will get:

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

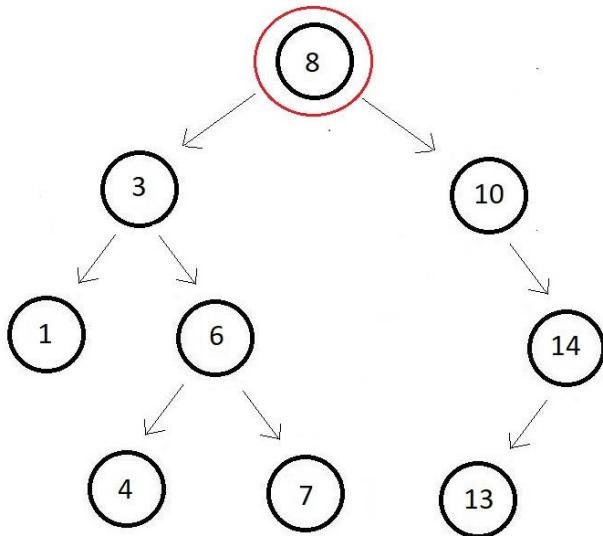
So, the InOrder predecessor and the Inorder successor of node 6 are 4 and 7 respectively. Hence you can substitute node 6 with any of these nodes, and the tree will still be a valid binary search tree. Refer to how it looks below.



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:

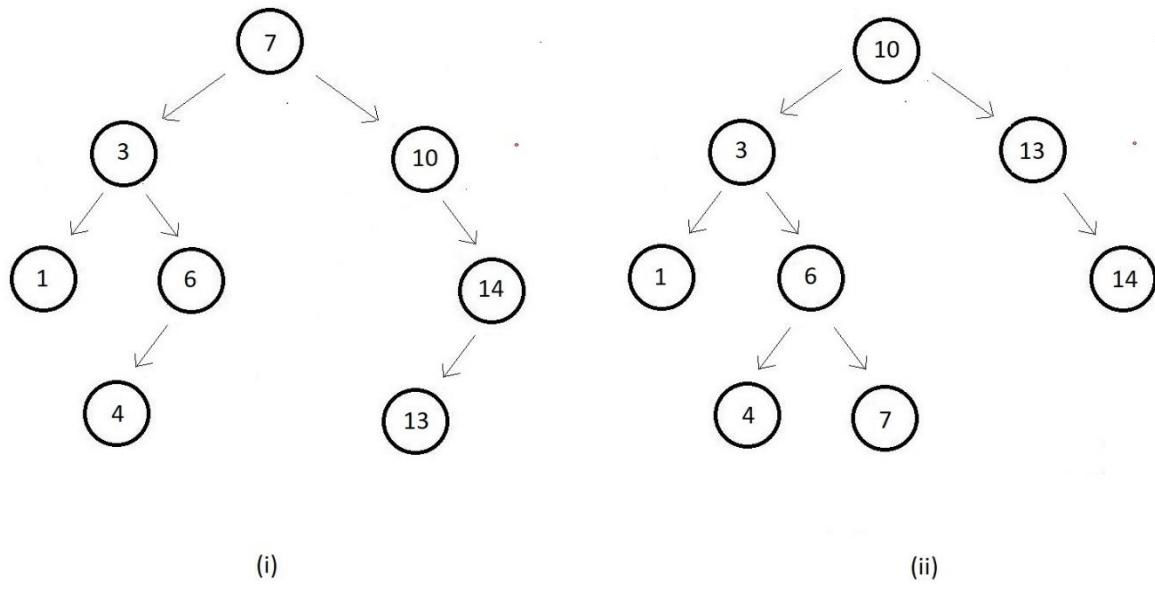
- Now, if you carefully observe, the root node is still another non-leaf node. So, the basics to delete the root node remains the same as what we did for a general non-leaf node. But since the root node holds a big size of subtrees along with, we have put this as a separate case here.



2. So, the first thing you 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 you can substitute node 8 with any of these nodes, but there is a catch here. So, if you substitute the root node here, with its InOrder predecessor 7, the tree will still be a binary search tree, but when you 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.

So, that was all we had. We saw all three cases in detail. We applied all our previous knowledge of the traversal series to help ourselves here. Next, we will see the programming part of the deletion operation. Practice on your own BST. Try deleting nodes in between the trees.

C Code For Deletion in a Binary Search Tree

In the last lecture, we dealt with all the cases of deletion operations in a binary search tree. Despite the complexity of the cases, we made them all appear easy. We saw examples of all the cases. Today, we'll see how to program those deletion cases and overall, the implementation of the deletion operation in a binary search tree using C language.

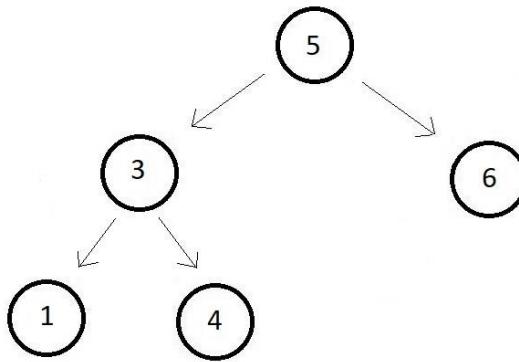
So, let's just move on to our editors without any further ado. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. Now, since our main focus would be to create the `deleteNode` function, we could just copy everything we did in the last programming lecture where we learnt the `insertion` operation in a binary search tree. And this would save us a lot of time.

And we would then have the InOrder and other traversal functions we did before at one place for our convenience.

2. You can always create a binary search tree of your own, and check if it is a BST using the InOrder traversal, and see if it's in ascending order or not. We would rather go with the one we already had in the program. Now, let's see the `deleteNode`



Creating the `deleteNode` function:

3. So, the first thing we would like to know is whether deleting this new node is even possible or not. For that, create a struct Node pointer function `deleteNode` and pass the pointer to the root node, and the data of the node you want to delete as its parameters. We will call it `value`

4. So, the next thing you would do is see whether our `value` is greater than or less than the root node's data. If it's less than the root node's data, we'll simply recursively call the function to its left subtree, and pass the same `value` into it. And if it's greater, we'll do the same thing with the right subtree. So, this basically reduces our problem. We do this until we reach the node we want to delete, or to the NULL, signifying the node was not there in the tree.

5. So, the deletion strategy says, when you find the node you want to delete, if you remember what we learnt, we can substitute it with its InOrder predecessor or with its InOrder successor. And we prefer doing it here using the InOrder predecessor. So, the first thing you would do here is to find the Inorder predecessor of the root node you want to delete and store it in a struct node pointer `iPre`. We would have a separate function for this, named `inOrderPredecessor` which we'll see later. Suppose it returns the pointer to the InOrder Predecessor of the root node.

6. Having received the pointer to the InOrder Predecessor of the root node, you just copy the data of that predecessor node to our root node. And call the `deleteNode` function recursively to the left subtree of the root node, and with the value of the InOrder predecessor's data. Now, observe this carefully. Why did we choose the left subtree? Because the InOrder predecessor of a node always lies on the left subtree of the node. And since you have replaced your root node with its InOrder predecessor, you have to now delete it from the left subtree.

7. And since these functions are recursive, we would need a base condition, and if you could realize, this recursive deletion of the replacing nodes must stop when it reaches a leaf node. So, the base conditions are when our given root is NULL, here we simply return NULL or both it's left and the right subtree is NULL, this is where the node we want to delete is the leaf node, so we'll simply free this node. And simply return the root node at the end.

```
struct node *deleteNode(struct node *root, int value){  
  
    struct node* iPre;  
    if (root == NULL){  
        return NULL;  
    }  
    if (root->left==NULL&&root->right==NULL){  
        free(root);  
    }  
  
    //searching for the node to be deleted  
    if (value < root->data){  
        deleteNode(root->left,value);  
    }  
    else if (value > root->data){  
        deleteNode(root->right,value);  
    }  
    //deletion strategy when the node is found  
    else{  
        iPre = inOrderPredecessor(root);  
        root->data = iPre->data;  
        deleteNode(root->left, iPre->data);  
    }  
}
```

[Copy](#)

Code Snippet 1: Creating the *deleteNode* function

8. Now, this deleteNode function would not simply work alone. We would need an *inOrderPredecessor* function we have used in the program.

Creating the *inOrderPredecessor* function:

9. Create a struct node pointer function *inOrderpredecessor* and pass into it the pointer to the root node you want to find the InOrder predecessor of as its parameter. I hope you all would have realised the fact that an Inorder predecessor of a node is the rightmost node of its left subtree.

10. So, simply update root to its left subtree, and use a while loop to iterate through all the right subtree, until we reach a leaf node. This gives our InOrder predecessor of the given node. Here, return the root.

```
struct node *inOrderPredecessor(struct node* root){  
    root = root->left;  
    while (root->right!=NULL)  
    {  
        root = root->right;  
    }  
    return root;  
}
```

Copy

Code Snippet 2: Creating the *inOrderPredecessor* function

Here is the whole source code:

```
#include<stdio.h>  
#include<malloc.h>  
  
struct node{  
    int data;  
    struct node* left;  
    struct node* right;  
};  
  
struct node* createNode(int data){  
    struct node *n; // creating a node pointer
```

```
n = (struct node *) malloc(sizeof(struct node)); // Allocating  
memory in the heap  
n->data = data; // Setting the data  
n->left = NULL; // Setting the left and right children to NULL  
n->right = NULL; // Setting the left and right children to NULL  
return n; // Finally returning the created node  
}
```

```
void preOrder(struct node* root){  
    if(root!=NULL){  
        printf("%d ", root->data);  
        preOrder(root->left);  
        preOrder(root->right);  
    }  
}
```

```
void postOrder(struct node* root){  
    if(root!=NULL){  
        postOrder(root->left);  
        postOrder(root->right);  
        printf("%d ", root->data);  
    }  
}
```

```
void inOrder(struct node* root){  
    if(root!=NULL){  
        inOrder(root->left);  
        printf("%d ", root->data);  
        inOrder(root->right);  
    }  
}
```

```
int isBST(struct node* root){  
    static struct node *prev = NULL;  
    if(root!=NULL){  
        if(!isBST(root->left)){  
            return 0;  
        }  
        if(prev!=NULL && root->data <= prev->data){  
            return 0;  
        }  
        prev = root;  
        return isBST(root->right);  
    }  
    else{  
        return 1;  
    }  
}
```

```
struct node * searchIter(struct node* root, int key){  
    while(root!=NULL){  
        if(key == root->data){  
            return root;  
        }  
        else if(key<root->data){  
            root = root->left;  
        }  
        else{  
            root = root->right;  
        }  
    }  
    return NULL;  
}
```

```

void insert(struct node *root, int key){
    struct node *prev = NULL;
    while(root!=NULL){
        prev = root;
        if(key==root->data){
            printf("Cannot insert %d, already in BST", key);
            return;
        }
        else if(key<root->data){
            root = root->left;
        }
        else{
            root = root->right;
        }
    }
    struct node* new = createNode(key);
    if(key<prev->data){
        prev->left = new;
    }
    else{
        prev->right = new;
    }
}

struct node *inOrderPredecessor(struct node* root){
    root = root->left;
    while (root->right!=NULL)
    {
        root = root->right;
    }
}

```

```
    return root;
}

struct node *deleteNode(struct node *root, int value){

    struct node* iPre;
    if (root == NULL){
        return NULL;
    }
    if (root->left==NULL&&root->right==NULL){
        free(root);
        return NULL;
    }

    //searching for the node to be deleted
    if (value < root->data){
        root->left = deleteNode(root->left,value);
    }
    else if (value > root->data){
        root->right = deleteNode(root->right,value);
    }
    //deletion strategy when the node is found
    else{
        iPre = inOrderPredecessor(root);
        root->data = iPre->data;
        root->left = deleteNode(root->left, iPre->data);
    }
    return root;
}

int main(){
```

```

// Constructing the root node - Using Function (Recommended)

struct node *p = createNode(5);
struct node *p1 = createNode(3);
struct node *p2 = createNode(6);
struct node *p3 = createNode(1);
struct node *p4 = createNode(4);

// Finally The tree looks like this:

//      5
//     / \
//    3   6
//   / \
//  1   4

// Linking the root node with left and right children

p->left = p1;
p->right = p2;
p1->left = p3;
p1->right = p4;

inOrder(p);
printf("\n");
deleteNode(p, 3);
inOrder(p);

return 0;
}

```

[Copy](#)

Code Snippet 3: Implementing the *deleteNode* function in C

Now, let's verify this function by running the `deleteNode` function with the value being 3, and to see if the node really got deleted, we'll run the `inOrder` function before and after the deletion operation.

```
inOrder(p);
printf("\n");
deleteNode(p, 3);
inOrder(p);
```

[Copy](#)

Code Snippet 4: Using the `deleteNode` function

And the output we received was:

```
1 3 4 5 6
1 4 5 6
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above code

So, yes, node 3 got deleted from the tree. And our tree didn't lose its identity, it remained a binary search tree. You can run the same function for all the other nodes to see if it works for all the cases. And try this for even bigger trees. Go through the last lecture again if you couldn't get hold of the basics of deletion. Next, we have our new data structure in the list, called the AVL trees.

AVL Trees - Introduction

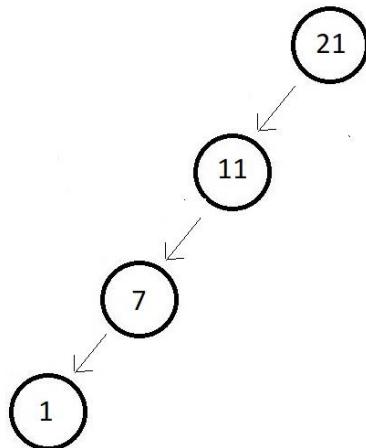
In the last lecture, we finished learning our operations on a binary search tree. We learned to search, insert, and all the cases of deletion in a binary search tree. Today, we'll start learning about the AVL trees.

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 lost to log and the complexity tends to $O(\log n)$, 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. I'll explain more on why AVL trees are considered a different topic worth teaching and why we even need AVL trees.

Why AVL trees?

I'll take an example to help you all understand the concept. Suppose you are told to create a Binary Search tree out of some elements given to you. Suppose the numbers

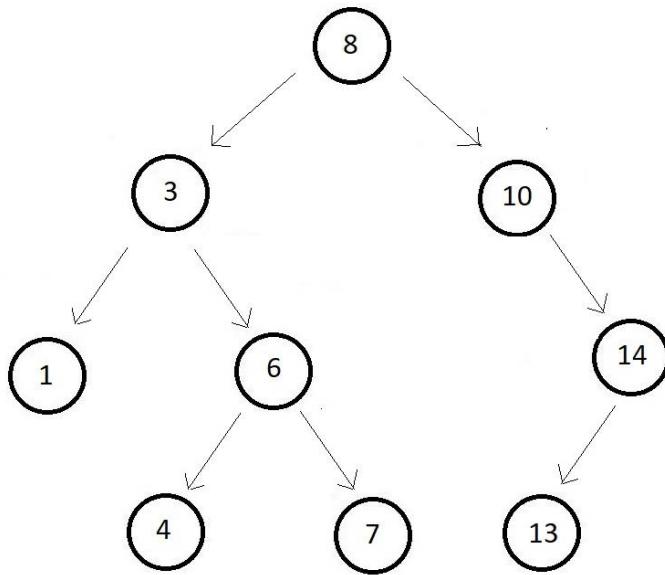
were $\{1, 11, 7, 21\}$. And to avoid any labour, you simply sort the elements and write them as shown in the figure below.



And to be honest, you 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 $O(n)$. But I can guarantee, you expected something more balanced, something more dense. Well, that is what an AVL tree is. An AVL tree is needed because:

1. Almost all the operations in a binary search tree are of order $O(h)$ where h is the height of the tree.
2. 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).
3. So, to guarantee an upper bound of $O(\log n)$ for all these operations we use balanced trees.

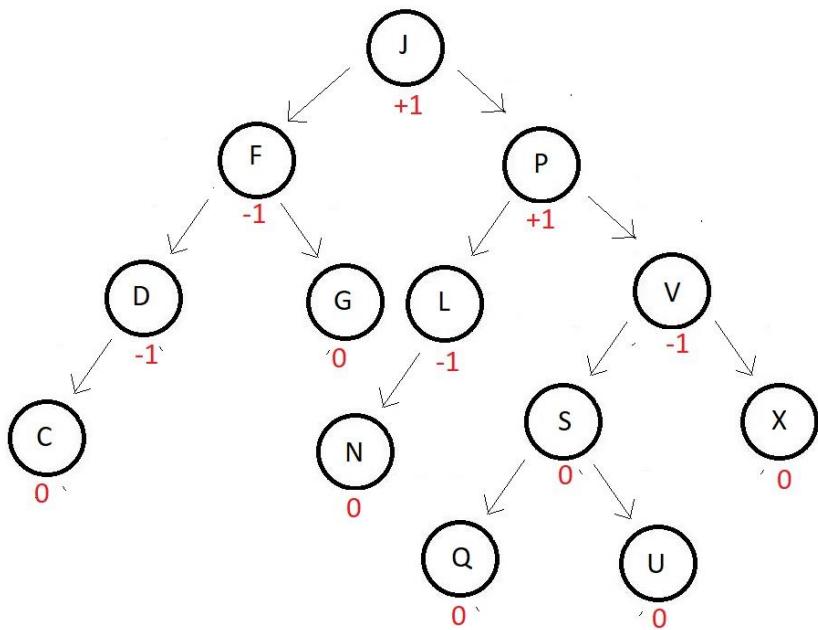
This is actually very practical. Because when a binary search tree takes the form of a list, our operations, say searching, starts taking more time. Consider the Binary search tree below.



To search 1, in this binary search tree, you 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?

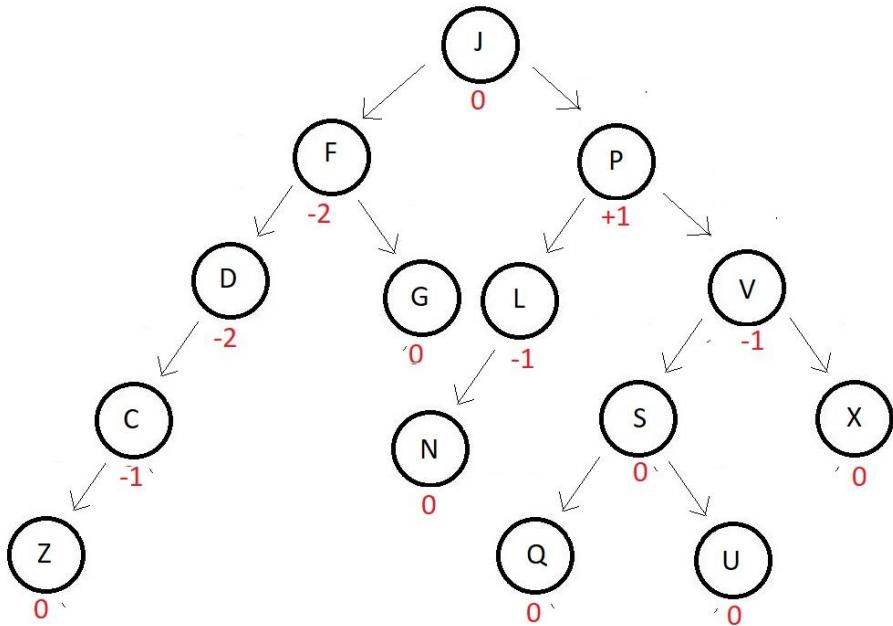
1. AVL trees are height balanced binary search trees. Because most of the operations work on $O(h)$, we would want the value of h to be minimum possible, which is $\log(n)$.
2. Height difference between the left and the right subtrees is less than 1 or equal in an AVL tree.
3. For AVL trees, there is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



You 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, i.e., $|BF| \leq 1$.

4. 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. I'll show you how in the next video.

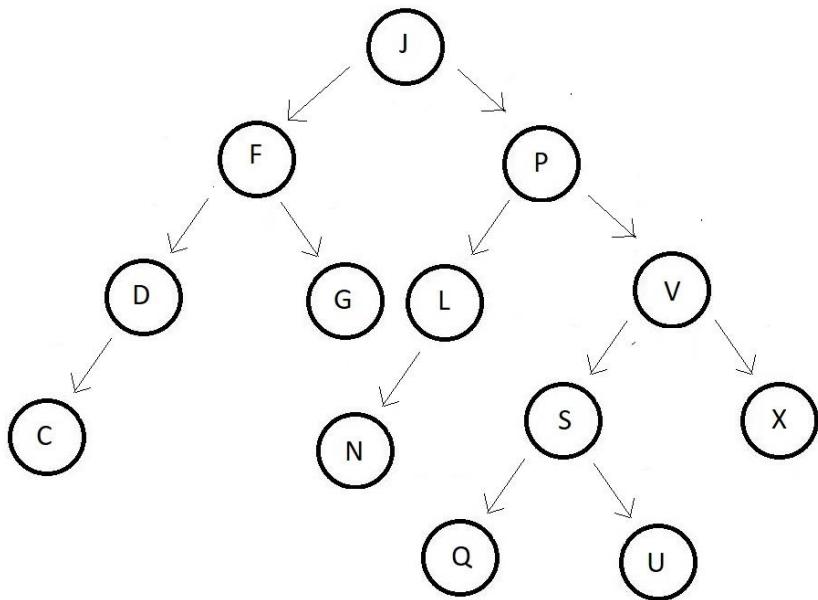
Insertion and Rotation in AVL Tree

In the last lecture, we got introduced to a new variant of the binary search tree, called the AVL trees. We saw why we needed AVL trees and how we judge whether a binary search tree is an AVL tree or not. Today, we'll learn about the rotations in an AVL tree, and why rotation is needed when you insert a new element in an AVL tree.

Before we see insertion and rotations in AVL trees, we'll give ourselves a quick brief revision of the characteristics of an AVL tree. These are:

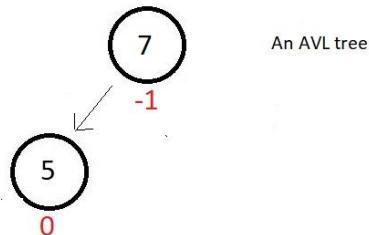
1. AVL trees are height-balanced binary search trees.
2. It gives an upper bound of $O(\log n)$ to all the operations possible in a Binary Search Tree.
3. Balance factor, $BF = \text{Height of Right Subtree} - \text{Height of Left Subtree}$, and for a binary search tree to be an AVL tree, $|BF|$ should be less than or equal to 1 for all of its nodes.

An example of an AVL tree is:

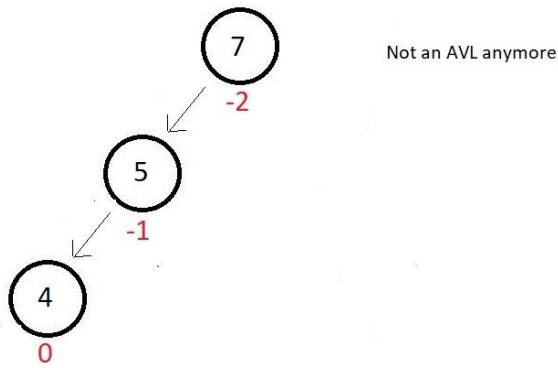


Now, before we proceed to see what different types of rotation in an AVL tree we have, we would first like to know why rotation is even done.

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.



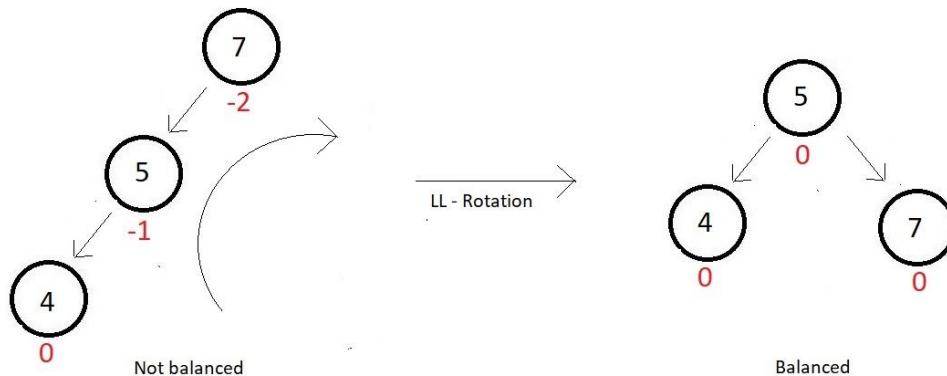
And you can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 4, our updated tree becomes unbalanced to the left. The absolute balance factor of node 7 becomes greater than 1.



So, the method you 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:

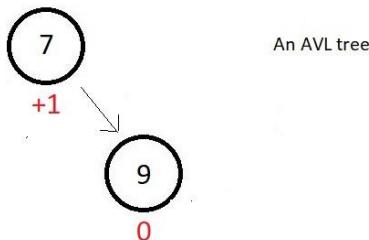
The name LL, just because we inserted the new element to the left subtree of the root. In this rotation technique, you just simply rotate your tree one time in the clockwise direction as shown below.



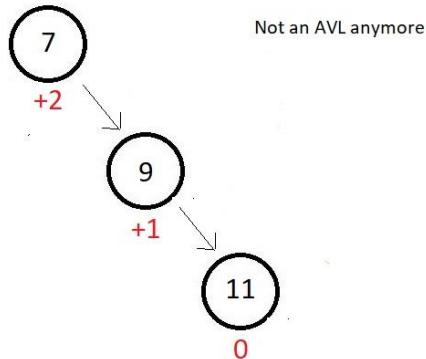
So, our tree got balanced again, with a perfect balance factor at each of its nodes. Next, we have the RR rotation.

RR Rotation:

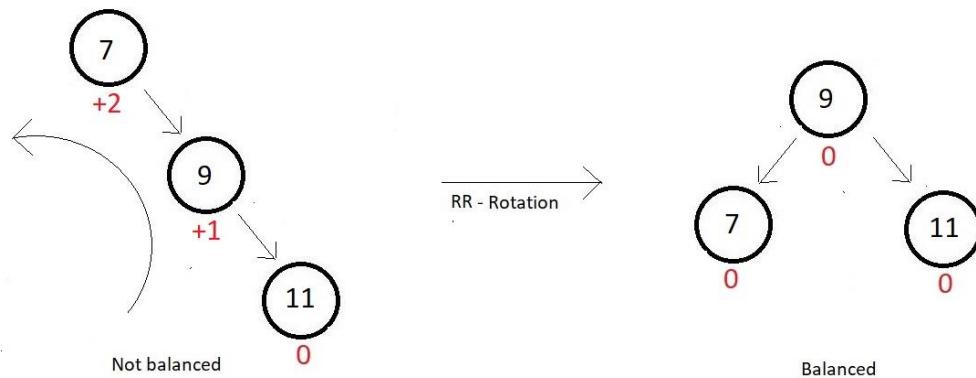
Now, suppose we have a small AVL tree having just these two nodes.



And you 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.



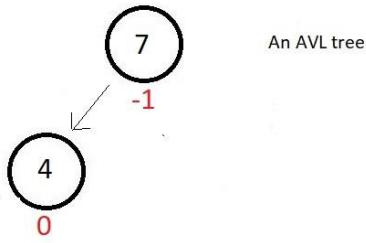
So, the method you will follow now to make this tree an AVL again is called the **RR rotation**. The name RR, just because we inserted the new element to the right subtree of the root. In this rotation technique, you just simply rotate your tree one time in an anti-clockwise direction as shown below.



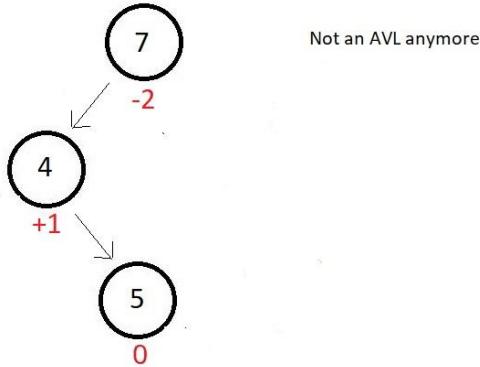
So, our tree got balanced again, with a perfect balance factor at each of its nodes. Next, we have the LR rotation.

LR Rotation:

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

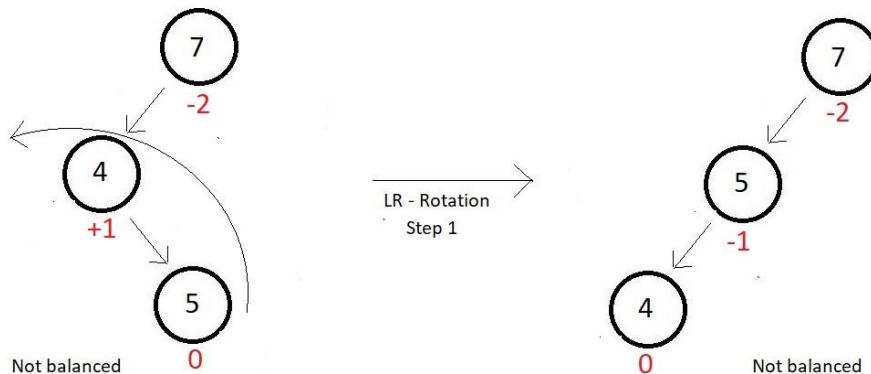


And you can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 5, our updated tree becomes unbalanced to the left. The absolute balance factor of node 7 becomes greater than 1.

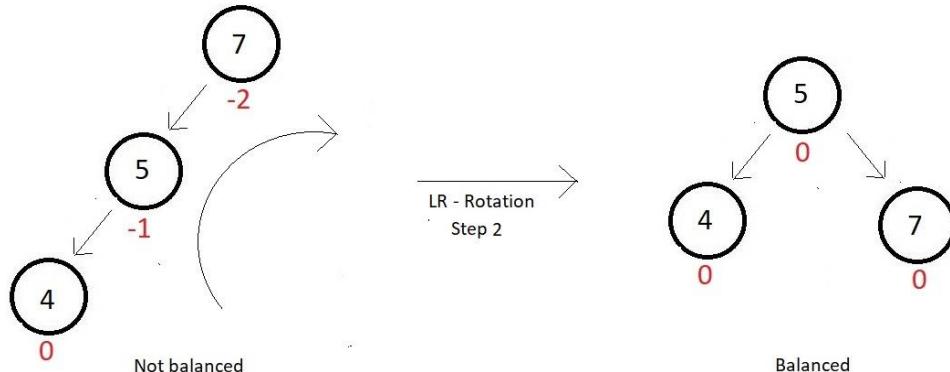


So, the method you will follow now to make this tree an AVL again is called the **LR rotation**. The name LR, just because we inserted the new element to the right to the left subtree of the root. In this rotation technique, there is a subtle complexity, which says, first rotate the left subtree in the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

Step 1:



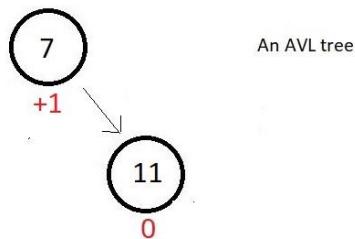
Step 2:



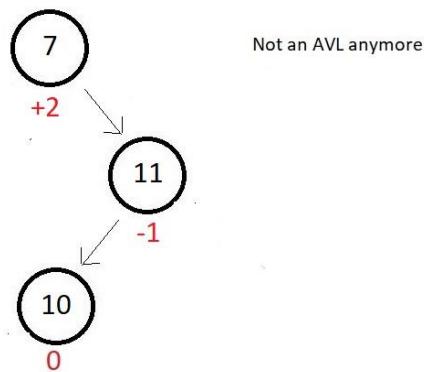
So, our tree got balanced again, with a perfect balance factor at each of its nodes. Although it was a bit clumsy, it was achievable. Next, we have the RL rotation.

RL Rotation:

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

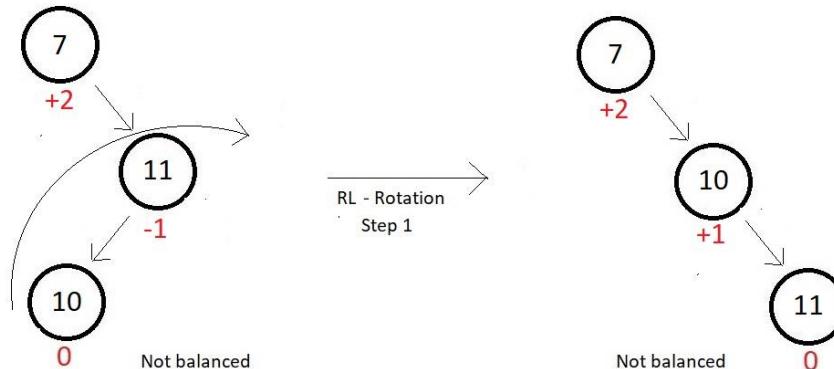


And you can see, the balance factor of both the nodes are good, but as soon as we insert a new node having data 10, our updated tree becomes unbalanced to the right. The absolute balance factor of node 7 becomes greater than 1.

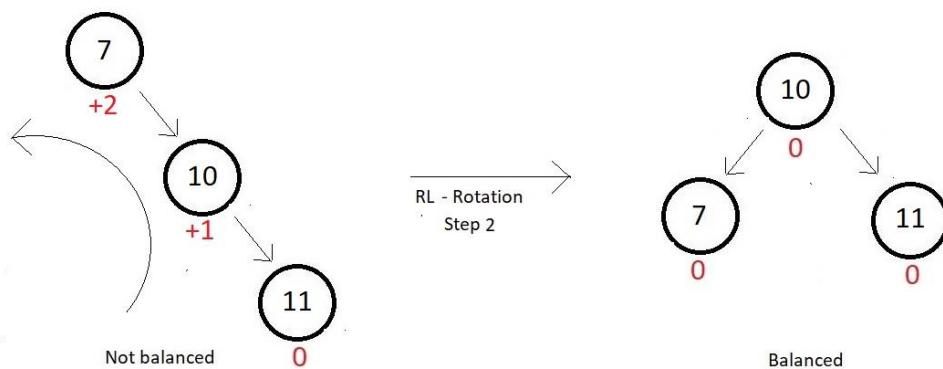


So, the method you will follow now to make this tree an AVL again is called the **RL rotation**. The name RL, just because we inserted the new element to the left to the right subtree of the root. We follow the same technique we used above, which says, first rotate the right subtree in the clockwise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

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

In the last lecture, we saw why AVL trees are important and how rotations in an AVL tree balances it back after an unbalanced insertion. We learned all the different types of rotations we have in AVL trees. We practiced them on the smallest possible AVL tree, but today, we'll push ourselves further with some complex AVL trees and complex situations that arise after insertion.

So, before we proceed with the examples of the rotations we learned, we should know how a binary search tree that is unbalanced can be balanced with a few Rotate Operations.

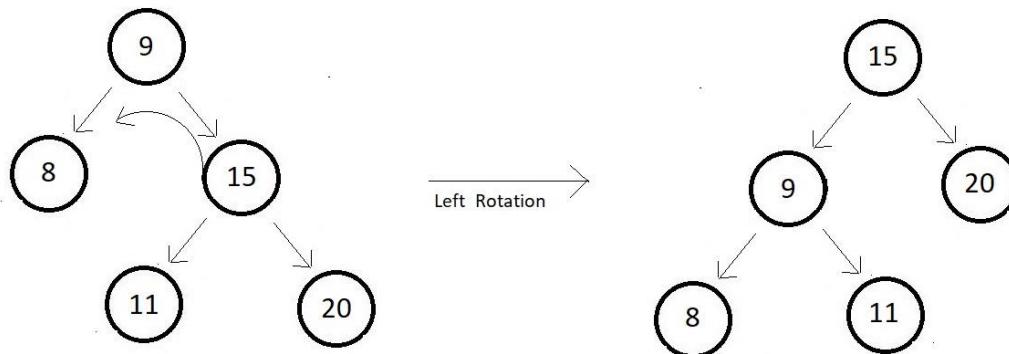
Rotate Operations:

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:

1. Left Rotate Operations.
2. Right Rotate Operation

Left Rotate Operations:

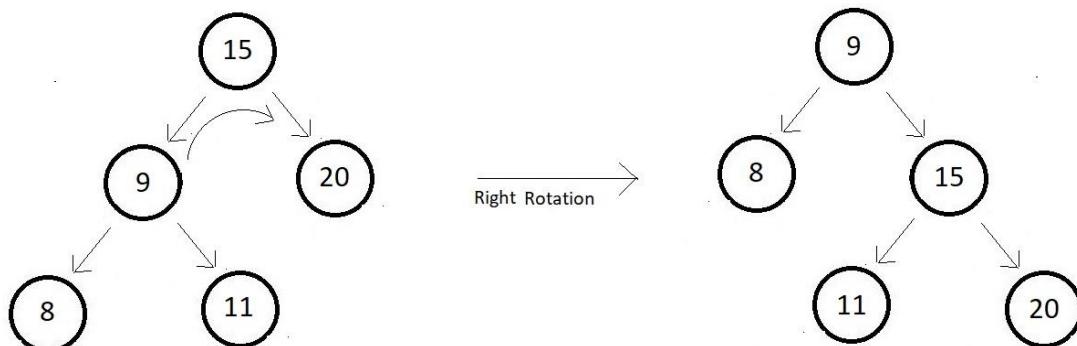
In this Rotate Operation, we move our unbalanced node to the left. Consider a binary search tree given below, and the newly formed tree after its left rotation with respect to the root.



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.



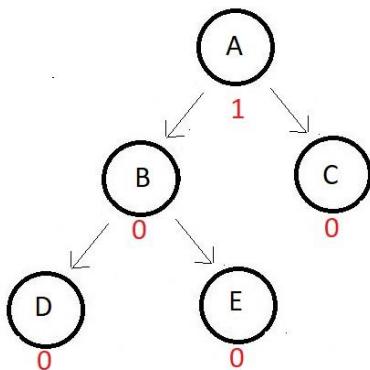
Again, as you can see that node 11 had to change its parent after the rotation to be able to maintain the balance of the tree. And rotating a tree to its left, and then again

to the right, yields the same original tree as you can see from the above two examples.

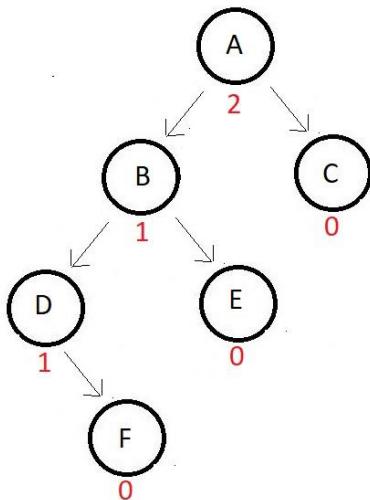
Let's move back to the balancing of the AVL tree after insertion. So, when it comes to complex trees, in order to balance an AVL tree after insertion, we can follow the below-mentioned rules:

1. For Left-Left insertion - Right rotate once with respect to the first imbalance node.
2. For Right-Right insertion - Left rotate once with respect to the first imbalance node.
3. For Left-Right insertion - Left rotate once and then Right rotate once.
4. 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:

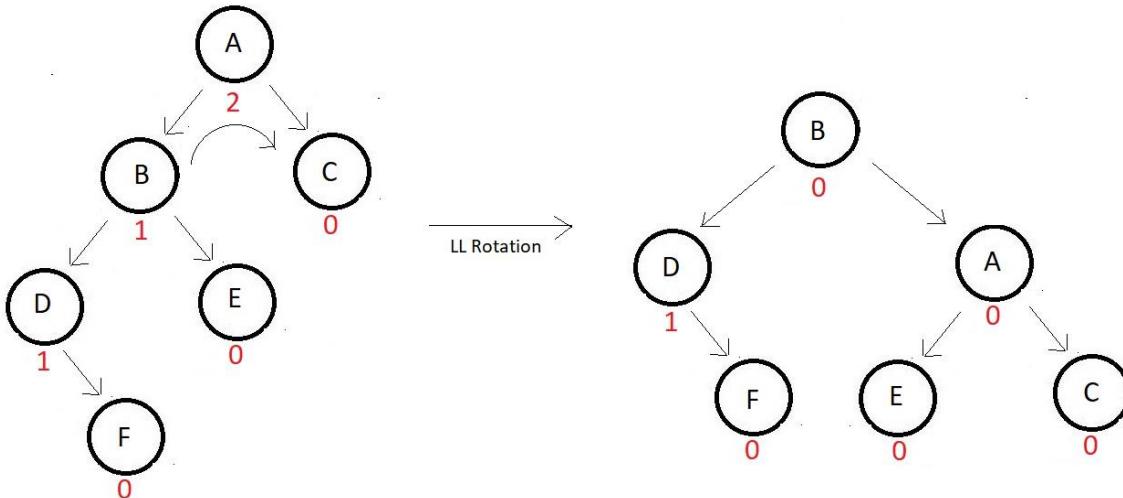


The absolute balance factor of each node is written beside, and you can see how balanced the values are. Now suppose we need to insert an element that gets its position to the right of node D. Now the updated tree looks something like this.



And the tree got imbalanced. Now, you follow these steps.

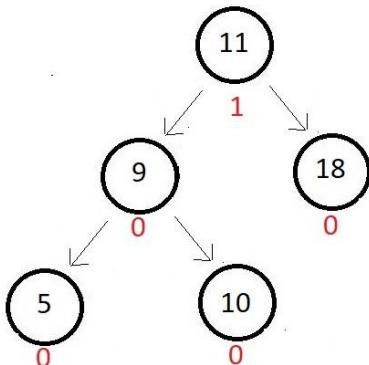
1. The first thing you would do is search for the node which got imbalanced first. We start iterating from the node we inserted at and move upwards looking for that first imbalance node. Here node A is the one we were searching for.
2. Second, you see what type of insertion was this with respect to the 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.

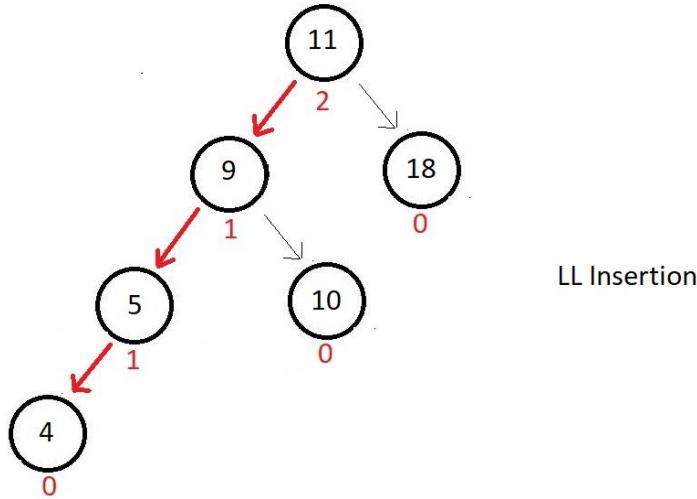
LL Rotation:

LL rotations were discussed already in the last lecture. We would just use it to balance a relatively complex AVL tree. Consider the one given below.



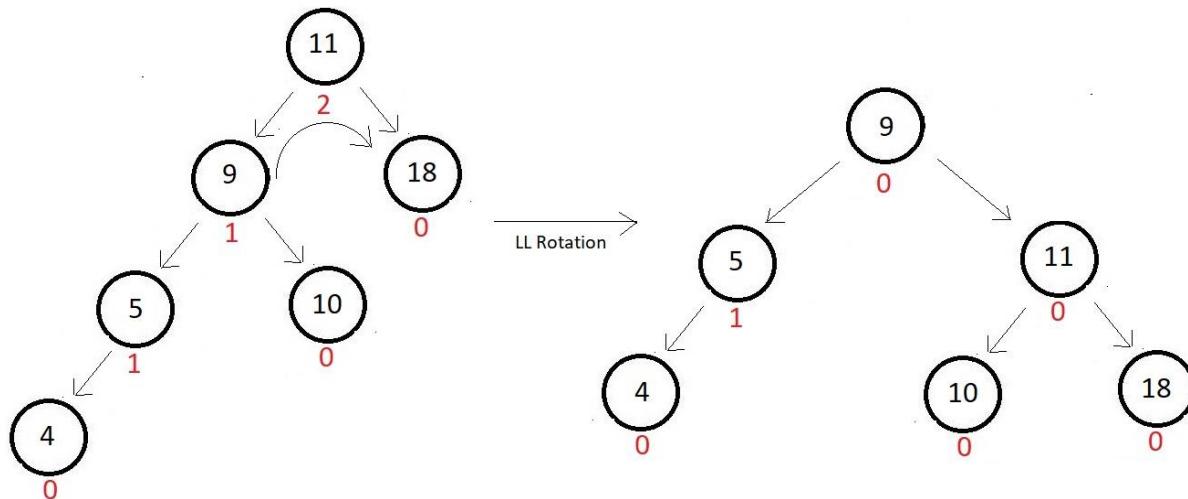
Absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 4. So, that

would get inserted to the left to node 5. The updated tree and their balance factors are:



LL Insertion

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.

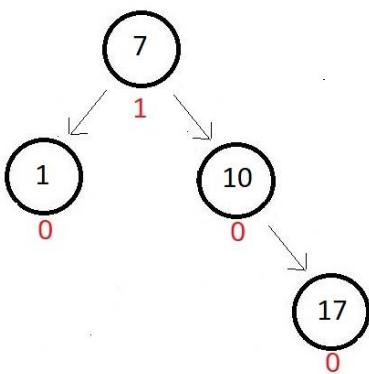


This is just a coincidence that our root node is the one we are rotating with respect to. We could come across examples where the first imbalanced node is not the root. So, we would rotate with respect to the one we'll find first, not the root.

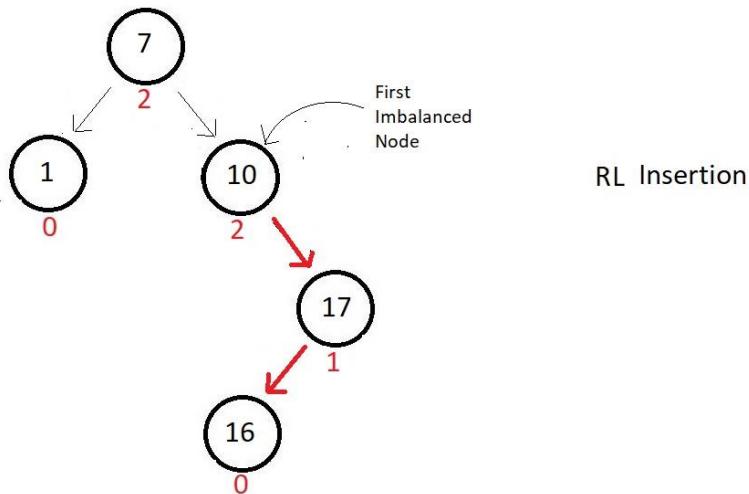
Let's now deal with one of the RL insertion cases. LR would be more or less the same, so, we'll just ignore that.

RL Rotation:

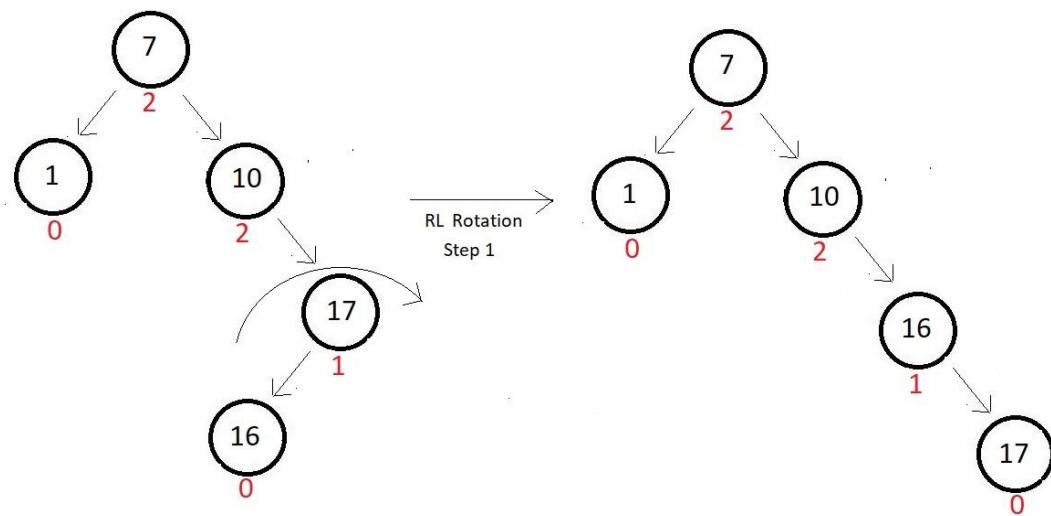
RL rotations were also discussed in the last lecture. We would just use it to balance a relatively complex AVL tree. Consider the one given below.



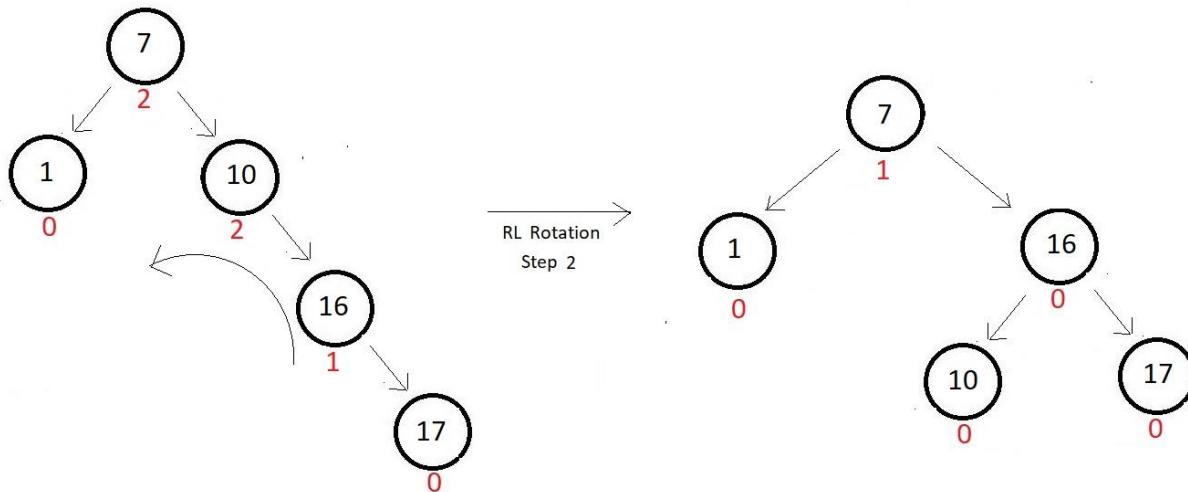
The absolute balance factors of each of the nodes are written beside. The tree is balanced and good for now. But now, we want to insert a node with data 16. So, that would get inserted to the left of node 17. The updated tree and their balance factors are:



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.



C Code For AVL Tree Insertion & Rotation (LL, RR, LR & RL Rotation)

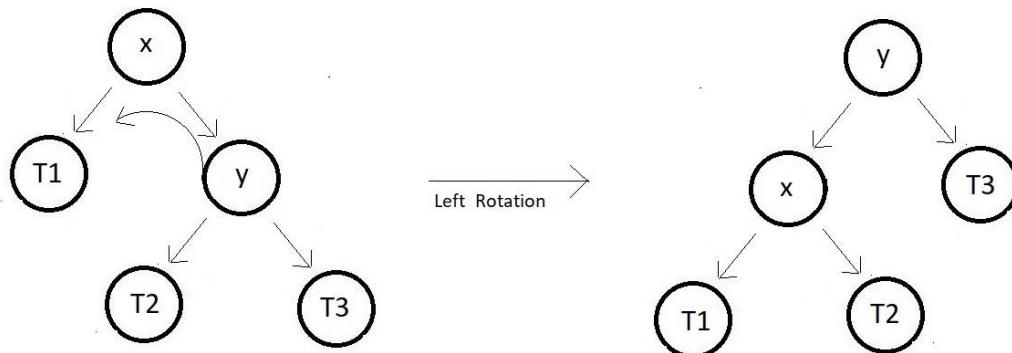
In the last lecture, we saw some good examples and complex ones to fix violations that arise after insertion operation using different rotation techniques. We learned using all the different types of strategies we use in AVL trees to balance an unbalanced tree caused after the four types of rotation situations. Today, we'll see the implementation of the same in C language.

So, before we proceed with the programming implementation of the things we learned, let's first give ourselves a very quick revision of a few things.

The two basic Rotate operations were:

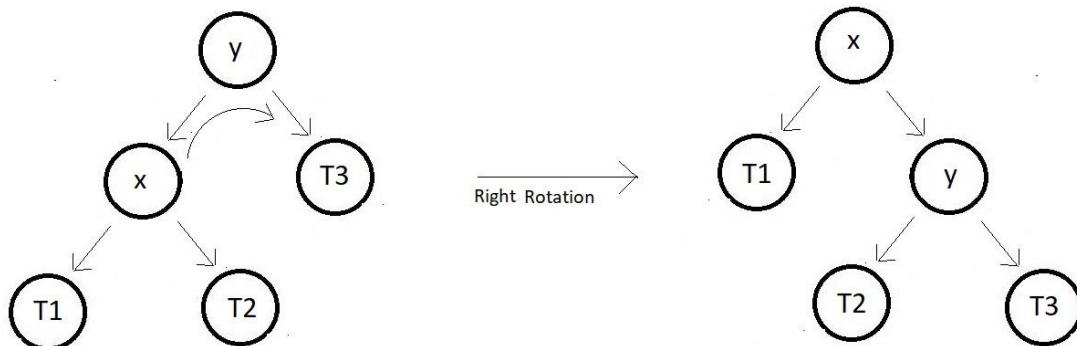
1. Left Rotate Operations.

Here, we move our unbalanced node to the left. Follow the example below.



2. Right Rotate Operation

Here, we move our unbalanced node to the right. Follow the example below.



We'll use the same tree we have illustrated above to guide us through our implementation of left and right rotation functions. We have already learnt in the last lecture the situations we had to use these two rotations techniques in. When you study the code, keep these illustrations in mind.

So, let's just move on to our editors without any further ado. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. The first thing would be to create a struct **Node** which we earlier used to create a node in a tree. So, create a struct **Node**. Inside the structure, we would have four embedded elements, first is an integer variable **data** to store the data of the node, second is a struct **Node** pointer called **left** to store the address of the left child node, and third is again a struct **Node** pointer called **right** to store the address of the right child node. Fourth is an integer variable storing the height of that node for the ease of calculating the absolute balance factor. Absolute balance factor, if you remember, should not exceed 1, otherwise, that node becomes imbalanced.

```
struct Node
```

```
{  
    int key;  
    struct Node *left;  
    struct Node *right;  
    int height;  
};
```

Copy

Code Snippet 1: Creating the struct Node

Creating the getHeight function:

2. Create an integer *getHeight*, and pass the pointer to the struct node you want to get the height of as the only parameter. Check if the struct node pointer is not NULL, return 0 if it is NULL, otherwise return the height element of the struct node. And that would be it.

```
int getHeight(struct Node *n){  
    if(n==NULL)  
        return 0;  
    return n->height;  
}
```

Copy

Code Snippet 2: Creating the getHeight function

Creating the createNode function:

3. Create a struct Node* function *createNode*, and pass the data/key for the node as the only parameter. Create a struct Node pointer *node*. Reserve a memory in the heap for the *node* using malloc. Make the left and the right element of the struct *node* point to NULL, and fill the data in the data element. Now, set the height element of this *node* as 1, because any new node would be a leaf node that has a height of 1. And now, return the pointer *node* you created. This would simply create a node as a part of the tree.

```
struct Node *createNode(int key){  
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));  
    node->key = key;  
    node->left = NULL;
```

```
    node->right = NULL;  
    node->height = 1;  
    return node;  
}
```

[Copy](#)

Code Snippet 3: Creating the `createNode` function

Here come the three most important functions of all when we implement the AVL tree. First is the `getBalanceFactor`, the second is the `leftRotate` and the third is the `rightRotate`.

Creating the `getBalanceFactor` function:

4. Create a struct `Node*` function `getBalanceFactor`, and pass the pointer to the struct node you want to get the balance factor of as the only parameter. Now the balance factor of a node is just the value we receive after subtracting the height of the right child with that of the left child of the node. So, first check whether the node is not `NULL`, if it is, simply returns 0. Otherwise, return the value received after subtracting the height of the right child with that of the left child. This was a utility function to find the balance factor for a node.

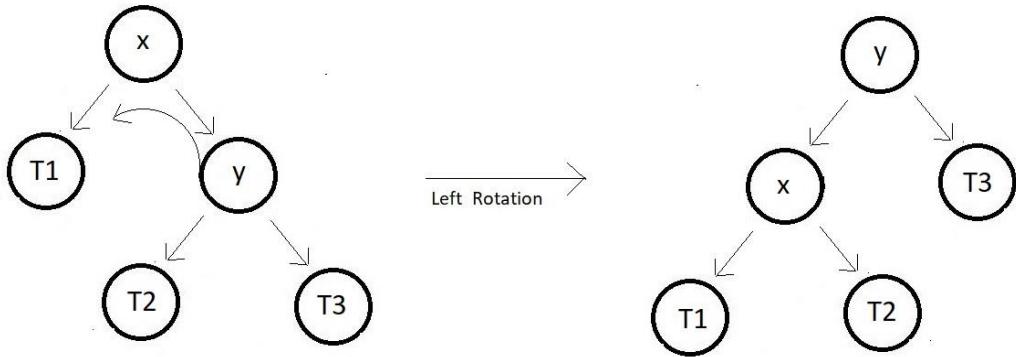
```
int getBalanceFactor(struct Node * n){  
    if(n==NULL){  
        return 0;  
    }  
    return getHeight(n->left) - getHeight(n->right);  
}
```

[Copy](#)

Code Snippet 4: Creating the `getBalanceFactor` function

Follow the illustrations below while we write the left and the right rotate functions respectively.

Creating the `leftRotate` function:



5. Create a struct Node* function leftRotate, and pass the pointer to the imbalanced struct node you want to left rotate as the only parameter. Consider this node *x* as represented in the figure above. As you could see, the left child of *x* is *y*. So, define a struct node pointer variable *y* and initialize it with *x->right*. As seen in the figure, we would make *x* the left child of *y*, and *T2* the right child of *x*. So, we would also store *y->left* in another struct node pointer variable *T2*.

Now, simply assign *x* to the left element of *y*, and *T2* to the right element of *x*. And this would finish our left rotation function. We must, however, remember to change the new heights of the nodes that have been shifted, which are *x* and *y*. Height of *x* becomes one added to the maximum of the heights of both of its children. Similarly, the height of *y* becomes one added to the maximum of the heights of both of its children. And now simply return *y*, since *y* becomes the new root for the subtree. Don't forget to explicitly define a max function before using it.

```
struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;

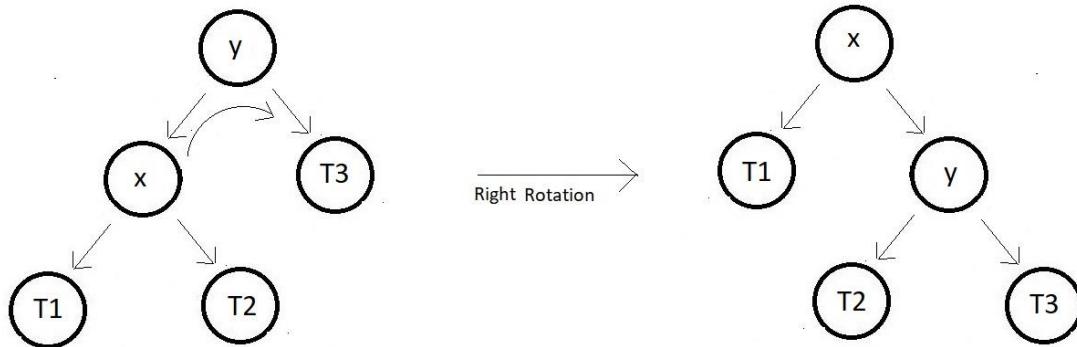
    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return y;
}
```

[Copy](#)

Code Snippet 5: Creating the leftRotate function

Creating the rightRotate function:



6. Create a struct Node* function rightRotate, and pass the pointer to the imbalanced struct node you want to right rotate as the only parameter. Consider this node y as represented in the figure above. As you could see, the left child of y is x . So, define a struct node pointer variable x and initialize it with $y->left$. As seen in the figure, we would make y the right child of x , and $T2$ the left child of y . So, we would also store $x->right$ in another struct node pointer variable $T2$.

Now, simply assign y to the right element of x , and $T2$ to the left element of y . And this would finish our right rotation function. In addition, we must remember to adjust the new height for the nodes that have been shifted, which are x and y . Height of x becomes one added to the maximum of the heights of both of its children. Similarly, the height of y becomes one added to the maximum of the heights of both of its children. And now simply return x , since x becomes the new root for the subtree.

```
struct Node* rightRotate(struct Node* y){  
    struct Node* x = y->left;  
    struct Node* T2 = x->right;  
  
    x->right = y;  
    y->left = T2;  
  
    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;  
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;  
  
    return x;  
}
```

Copy

Code Snippet 6: Creating the rightRotate function

Creating the insert function:

7. Create a struct Node* function *insert*, and pass the pointer to the root of the AVL tree and the data we want to insert as two of its parameters. Now, we would follow the simple Binary Search Tree insertion operation which we had already covered in our past lectures. There, we would just recursively iterate to the best-fit position where the new data should be inserted, and once found, we create a new node, and return the pointer to this node.

And since we have recursively gone down the tree, we would backtrack after inserting. While backtracking, we update the heights of all the nodes we followed to reach the apt position.

Once this new node gets inserted, the balance of the tree might get disturbed, so we would first create an integer variable *bf*, and store in it the balance factor of the current node we are on while backtracking. And if this node gets imbalanced, four of the possible cases arise.

Left-Left case:

Here, the new node would have been inserted on the left to the left child of the current node. So, if our *bf* has a value greater than 1, and the new node has data less than the data of the left child of the node itself, it is the case of left-left rotation, and hence we call the *rightRotate* function once to fix this disturbance.

Right-Right case:

Here, the new node would have been inserted on the right to the right child of the current node. So, if our *bf* has a value less than -1, and the new node has data greater than the data of the right child of the node itself, it is the case of right-right rotation, and hence we call the *leftRotate* function once to fix this disturbance.

Left-Right case:

Here, the new node would have been inserted on the right to the left child of the current node. So, if our *bf* has a value greater than 1, and the new node has data greater than the data of the left child of the node itself, it is the case of left-right rotation, and hence we call first the *leftRotate* function passing the left subtree and then call the *rightRotate* function on this updated node to fix this disturbance.

Right-Left case:

Here, the new node would have been inserted on the left to the right child of the current node. So, if our *bf* has a value less than -1, and the new node has data less than the data of the right child of the node itself, it is the case of right-left rotation, and hence we call first the *rightRotate* function passing the right subtree and then call the *leftRotate* function on this updated node to fix this disturbance.

And then finally return this node.

8. The Last step would be to fetch the *preOrder* traversal function from our previous programming lecture, to actually see how our tree would get all balanced after all the insertions we would do.

```

struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return createNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int bf = getBalanceFactor(node);

    // Left Left Case
    if(bf>1 && key < node->left->key){
        return rightRotate(node);
    }

    // Right Right Case
    if(bf<-1 && key > node->right->key){
        return leftRotate(node);
    }

    // Left Right Case
    if(bf>1 && key > node->left->key){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if(bf<-1 && key < node->right->key){
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
}

```

```
    return node;  
}
```

[Copy](#)

Code Snippet 7: Creating the insert function

Here is the whole source code:

```
#include <stdio.h>  
#include <stdlib.h>  
  
struct Node  
{  
    int key;  
    struct Node *left;  
    struct Node *right;  
    int height;  
};  
  
int getHeight(struct Node *n){  
    if(n==NULL)  
        return 0;  
    return n->height;  
}  
  
struct Node *createNode(int key){  
    struct Node* node = (struct Node *) malloc(sizeof(struct Node));  
    node->key = key;  
    node->left = NULL;  
    node->right = NULL;  
    node->height = 1;  
    return node;  
}
```

```

int max (int a, int b){
    return (a>b)?a:b;
}

int getBalanceFactor(struct Node * n){
    if(n==NULL){
        return 0;
    }
    return getHeight(n->left) - getHeight(n->right);
}

struct Node* rightRotate(struct Node* y){
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
    y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

    return x;
}

struct Node* leftRotate(struct Node* x){
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    y->left = x;
    x->right = T2;
}

```

```

x->height = max(getHeight(x->right), getHeight(x->left)) + 1;
y->height = max(getHeight(y->right), getHeight(y->left)) + 1;

return y;
}

struct Node *insert(struct Node* node, int key){
    if (node == NULL)
        return createNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    int bf = getBalanceFactor(node);

    // Left Left Case
    if(bf>1 && key < node->left->key){
        return rightRotate(node);
    }

    // Right Right Case
    if(bf<-1 && key > node->right->key){
        return leftRotate(node);
    }

    // Left Right Case
    if(bf>1 && key > node->left->key){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
}

```

```
// Right Left Case
if(bf<-1 && key < node->right->key){
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
return node;
}
```

```
void preOrder(struct Node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
int main(){
    struct Node * root = NULL;

    root = insert(root, 1);
    root = insert(root, 2);
    root = insert(root, 4);
    root = insert(root, 5);
    root = insert(root, 6);
    root = insert(root, 3);
    preOrder(root);
    return 0;
}
```

[Copy](#)

Code Snippet 8: Implementing AVL tree and its insertion operation

And now having finished all the functions we needed, we can create an AVL tree in main and verify if all these functions actually work. So, simply create a struct node pointer root and initialize it with NULL. And now we are good to insert as many nodes as we want. We would insert 6 of them. And then call the preOrder traversal on the same.

```
struct Node * root = NULL;  
  
root = insert(root, 1);  
root = insert(root, 2);  
root = insert(root, 4);  
root = insert(root, 5);  
root = insert(root, 6);  
root = insert(root, 3);  
preOrder(root);
```

[Copy](#)

Code Snippet 9: Creating an AVL tree and using the insert function

The output we received was:

```
4 2 1 3 5 6
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output of the above program

So, our AVL tree is actually working all fine. And this was the implementation of the rotations we learned. I know this was something actually I would grade tough, but giving respect to the complexity of the topic, I would recommend you all to go through these again and practice writing the code yourselves.

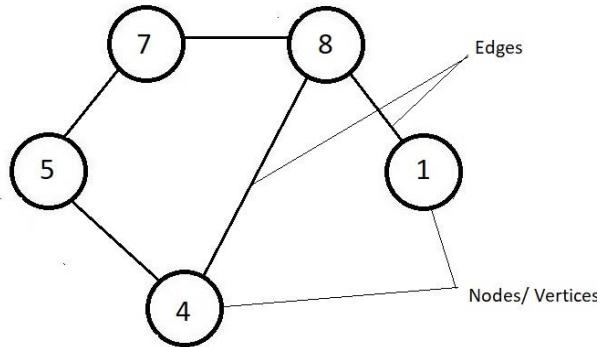
Introduction to Graphs | Graph Data Structure

As we continue on to our topic of data structures and algorithms, we'll examine one of the most important subjects of all, **graphs**. Graphs are an important data structure we should learn. Today, we'll start with the introduction of graphs, and the reasons why we should learn them.

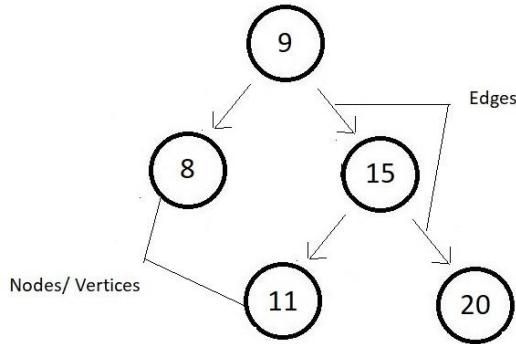
What is a graph?

All the data structures we have learned so far have either been linear data structures or nonlinear data structures. Linear data structures included arrays, linked lists, and stacks, while nonlinear hierarchical data structures included Binary Search Trees and AVL Trees. 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. You might be surprised to learn that trees are also a type of graph. Well, there is nothing as such to be surprised about. Follow the illustration of a tree below.

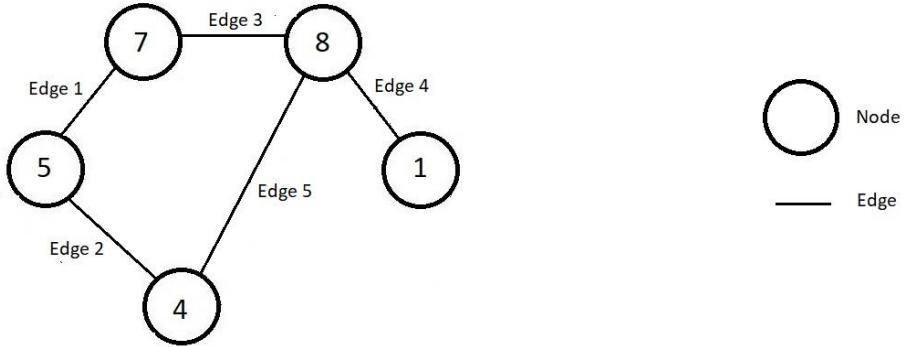


A tree is also a collection of nodes and edges, and hence is a graph only. Let's now quickly see all those elements of a graph.

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. You can take the analogy of people in a network, or a widespread chain, where people could be

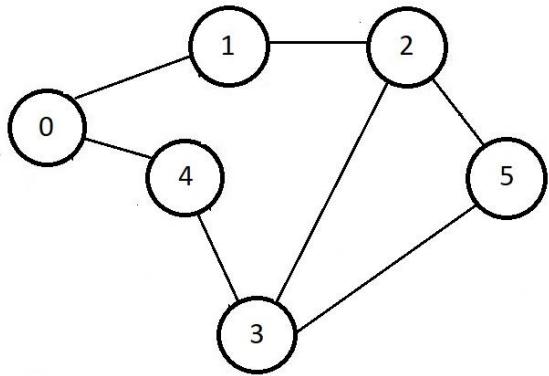
representing the nodes and their connections in the network could be represented using edges. It is a structure.



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:

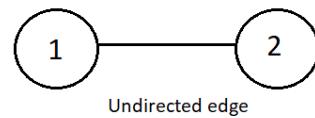
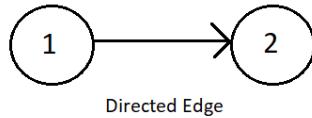
Graphs have a wide range of applications in our world. Learning graphs also becomes a necessity for anyone pursuing software development due to its applications. Graphs are used to model paths in a city, as often seen in Google Maps. They are used to model social networks such as Facebook or LinkedIn.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

Types of Edges:

1. Directed Edge - A directed edge is an edge connecting two nodes but strictly defines the way it is connected from and to. Below example shows node 1 connecting to node 2, and not vice - versa. You can take the analogy of Facebook's follow feature, where if you follow someone, then it's not that the other person who

followed you 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 your website to their page :)

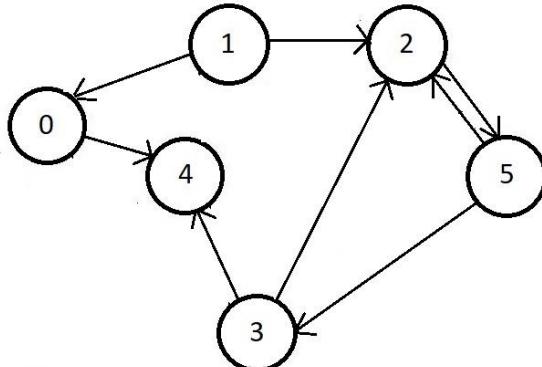
2. Undirected Edge - An undirected edge is an edge connecting two nodes from the way. Below example shows node 1 connecting to node 2, and at the same time node 2 connecting to node 1. You can take the analogy of Facebook's friend feature, where if you make someone a friend, then you too become their friend, so it's both ways.



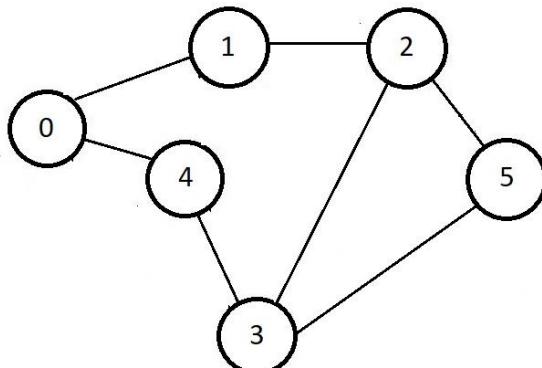
According to the type of edges that a graph has, graphs can be further divided into two types.

Types of Graphs:

1. Directed graph - Directed graphs are graphs having each of its edges directed. One of the examples of directed graphs is:



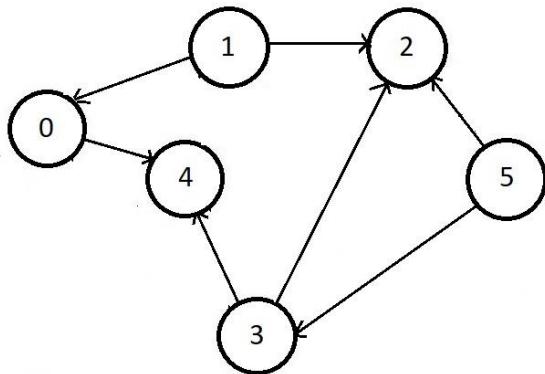
2. Undirected graph - Directed graphs are graphs having each of its edges undirected. One of the examples of directed graphs is:



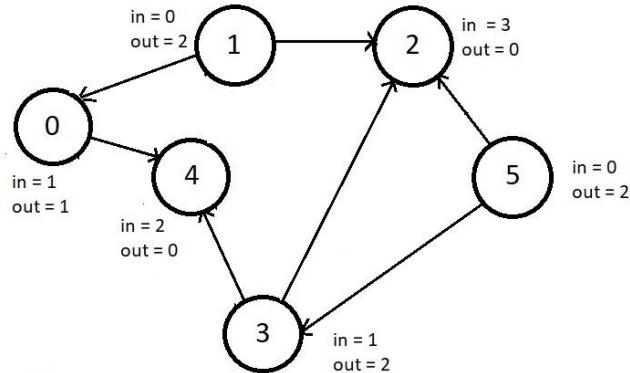
You may sometimes get to see both directed and undirected edges in the same graph, but those are rarely studied. For now, we would restrict ourselves to directed and undirected graphs only.

Indegree and Outdegree of a node:

As their name suggests, 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 below:



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

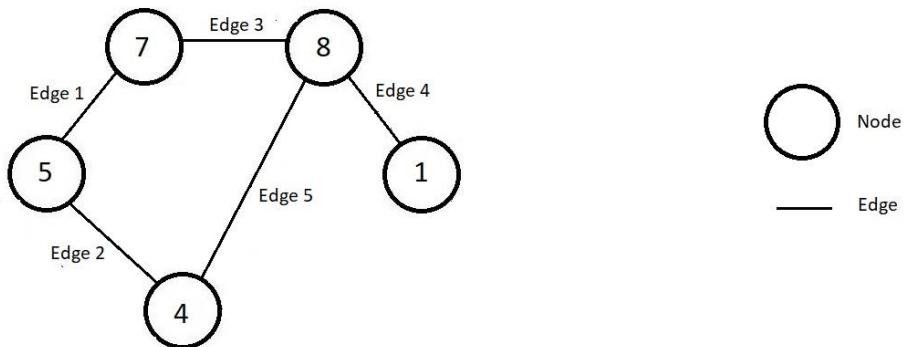
Representation of Graphs - Adjacency List, Adjacency Matrix & Other Representations

In the last lecture, we introduced to you our new topic, graphs. We saw the applications of graphs and discussed their elements including their nodes/ vertices and edges. Moving ahead with our discussion on graphs, today we'll learn how graphs are represented, and various terminologies related to graphs.

Representation of graphs is actually a very important concept and once you understand how graphs are represented, learning other graph algorithms becomes much easier. The first lecture detailed the following points:

1. Graphs and their nodes & edges, directed and undirected edges and graphs.
2. Applications including their use to model real-world problems like managing a social network, website links, etc.
3. Another major application is their use to solve problems like is there a path between two locations on a map and if there is, which one is the shortest.

A simple undirected graph looks like this:



Let's now move to see how we represent graphs in various ways.

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** - $A_{ij} = 1$, if there is an edge between i and j , 0 otherwise.

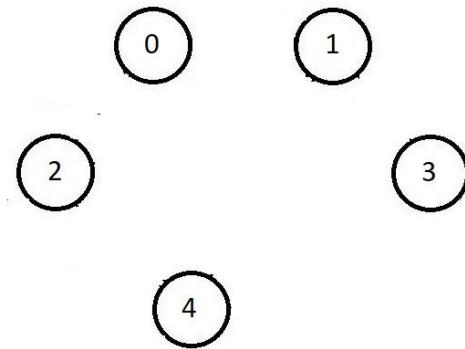
We'll see the above two in detail. Meanwhile, other representations include:

1. **Edge Set** - Store the pair of nodes/vertices connected with an edge. Example: $\{(0, 1), (0, 4), (1, 4)\}$.
2. Other implementations to represent a graph also exist. For example, Compact list representation, cost adjacency list, cost adjacency matrix, etc.

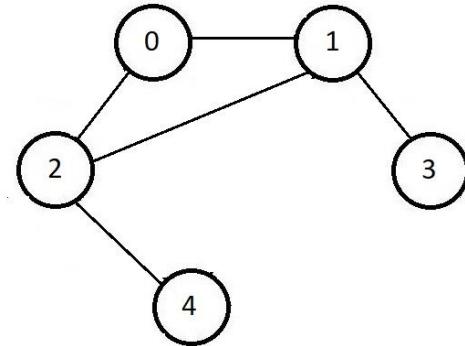
Let's now deal with each of these individually.

Adjacency List:

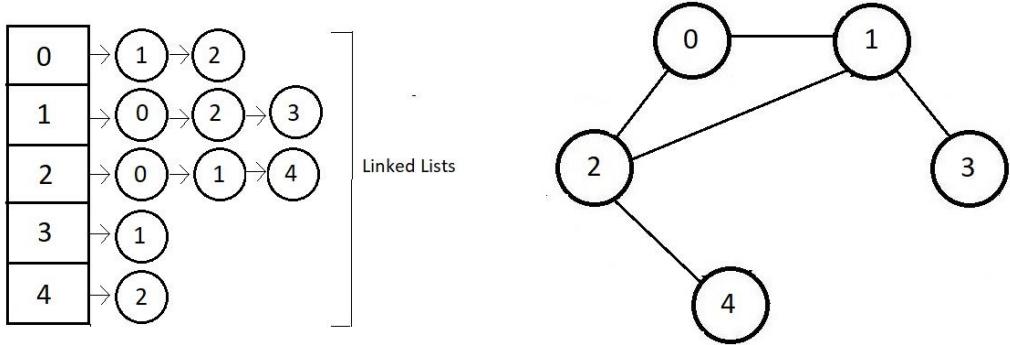
In this method of representation of graphs, we store the nodes, along with the list of their neighbors. Basically, we maintain a list of all the nodes, and along with it, we store the list of nodes a particular node is connected with. Consider the nodes I've illustrated below.



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, you 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.



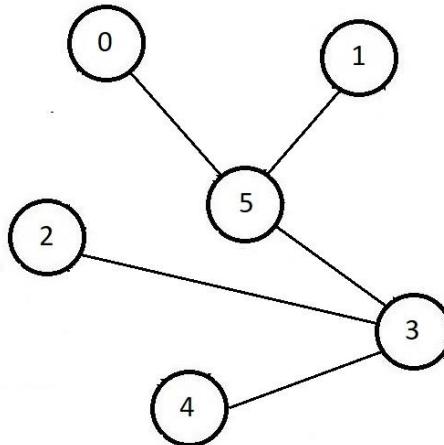
And the information about the connections of each of the nodes gets stored in separate linked lists as shown in the figure above. And each of the nodes itself acts as a pointer stored in an array pointing to the head of each of the linked lists. So, in this case, we would have an array of length 5 where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

And this was the adjacency list representation of graphs, and I can say that this is one of the most used representation methods. Next is the adjacency matrix.

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 i th row and j th column be A_{ij} , 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 a 0.

Consider the graph illustrated below:



Now, we'll make a 6×6 matrix as follows:

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

Now, 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 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:

	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

Now, one can very easily find 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. So, this was the adjacency matrix representation of graphs.

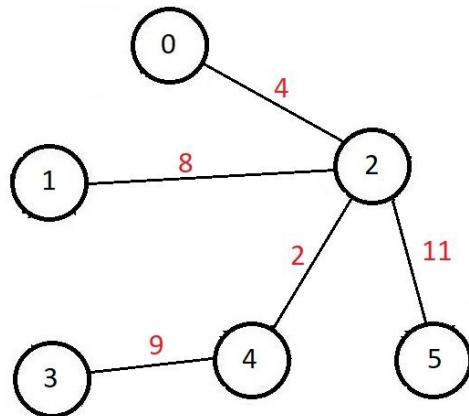
We can further extrapolate the application of the adjacency matrix by replacing these ones in the matrix with the weights for a weighted graph. Weighted graphs have a value/cost for each of the respective edges. These costs could represent

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 i th row and j th column be A_{ij} , 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.

Consider the graph illustrated below:



Now, we'll make a 6×6 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.

This was all about representing graphs. We'll be mainly using the adjacency list and adjacency matrix for all our purposes in the coming lectures. We'll see their programming as well in C and a lot more about graphs in the upcoming lecture.

Graph traversal & Graph traversal algorithms

In the last lecture, we discussed in detail the different ways to represent a graph. We saw two of the most popular representation methods namely the adjacency list and the adjacency matrix. Today, we'll be interested in learning what graph traversal is and about different graph traversal algorithms. We'll also look at the reasons why these traversal techniques are so important.

What is graph traversal?

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 you 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:

1. Breadth-First Search (BFS)
2. Depth First Search (DFS)

We'll see these algorithms in much detail in the coming lectures, but for today, we'll just sketch out a rough idea about these algorithms.

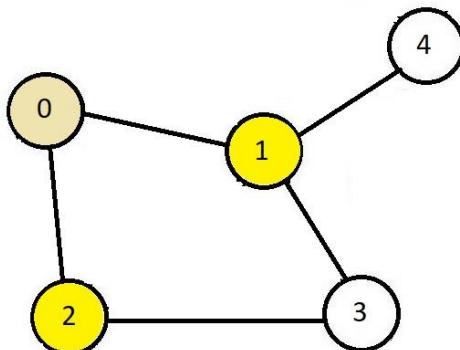
Breadth-First Search (BFS)

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, there are a few terminologies we need to familiarise ourselves with.

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.

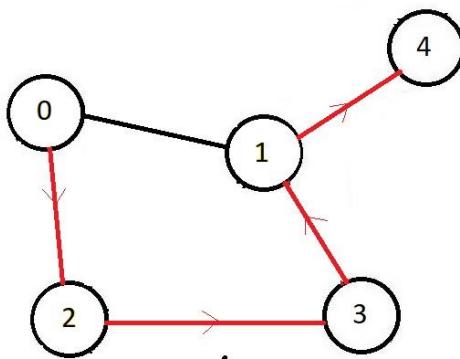
Follow the illustration of the graph I have mentioned below. We'll use this for understanding the breadth-first search.



In breadth-first search, we use the queue data structure, and let me tell you that 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 we get 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)

Follow the illustration of a graph I have mentioned below. We'll use this for understanding the depth-first search.



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.

So, this was the basic idea of each of these two algorithms. We'll discuss them in great detail in the upcoming lectures. We'll first understand the algorithm, then followed by their applications and the places where we prefer one over the other.

Breadth First Search (BFS) Graph Traversal in Data Structures

In the last lecture, we learned about what graph traversal is as well as two of the traversal techniques in brief. These were the Breadth-First Search and the Depth First Search. Today, we'll be dealing solely with the Breadth-First Search abbreviated as BFS in greater detail. I'll also provide you all with some techniques to derive the Breadth-First Search of any graph while only looking at it.

Before we actually move on to writing the algorithm of the Breadth-First Search or programming its implementation, we'll first visualize how Breadth-First Search works.

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.

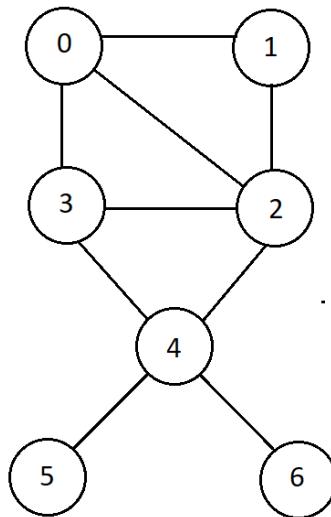
What is Breadth-First Search?

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.

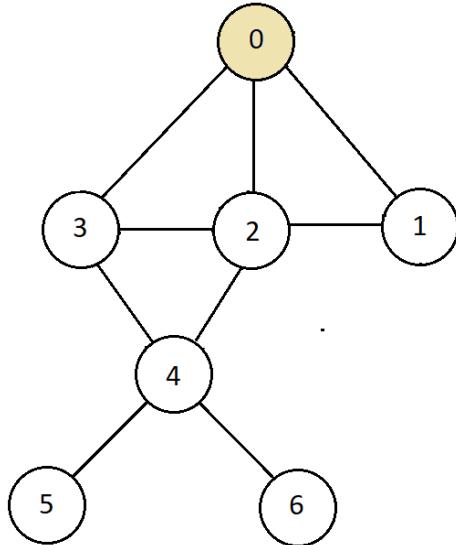
We should first learn the concept of the BFS spanning tree in order to understand the Breadth-First Search in a very intuitive way.

Method 1: BFS Spanning Tree:

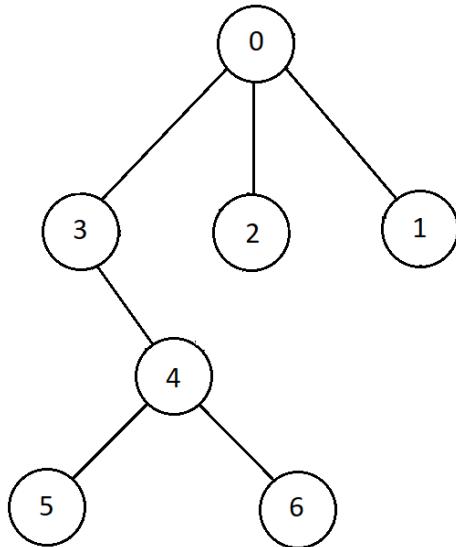
To understand what BFS Spanning Tree means, consider the graph I've illustrated below.



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. You 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,

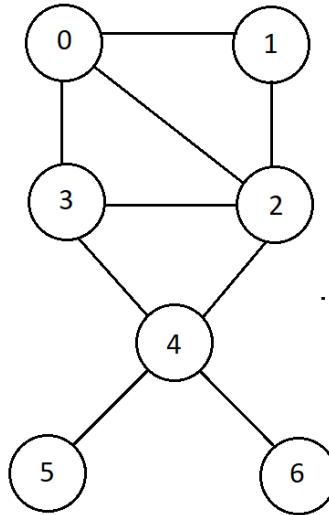


This constructed tree above is known as a BFS Spanning Tree. And surprisingly, the level order traversal of this BFS spanning tree gives us the Breadth-First Search traversal of the graph we started originally with. And if you remember what a level order traversal of a tree is, we simply write the nodes 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.

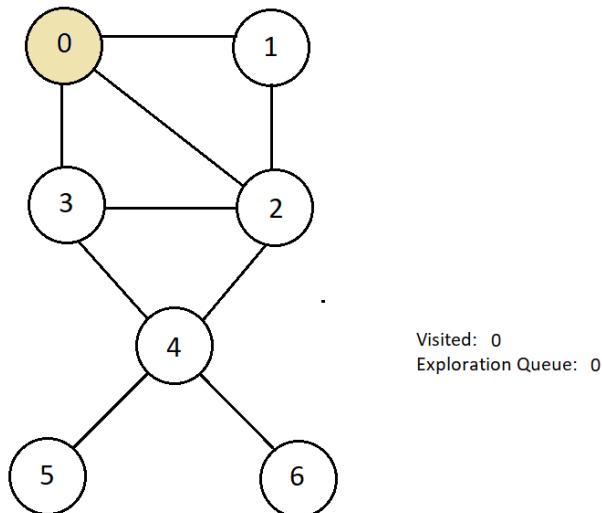
Therefore, given a graph, this is probably the easiest way, I believe, to write the Breadth-First Search traversal of the graph. Here, the chances of making a mistake are minimal. Despite the simplicity of the above technique, there is a very convenient method we follow when we implement the Breadth-First Search traversal algorithm in our program. Let us now discuss that.

Method 2: Conventional Breadth-First Search Traversal Algorithm:

Consider the same graph we covered above. Let me illustrate that again.

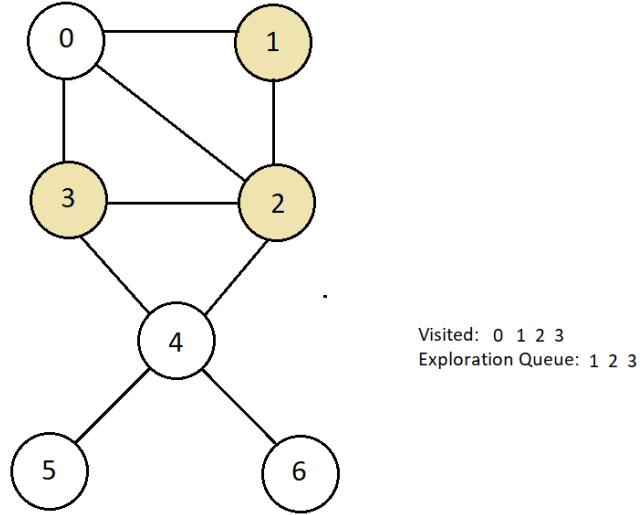


Considering we could begin with any source node; we'll start with 0 only. Let's define a queue named **exploration queue** which would hold the nodes we'll be exploring one by one. We would maintain another array holding the status of whether a node is already **visited** or not. Since we are starting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

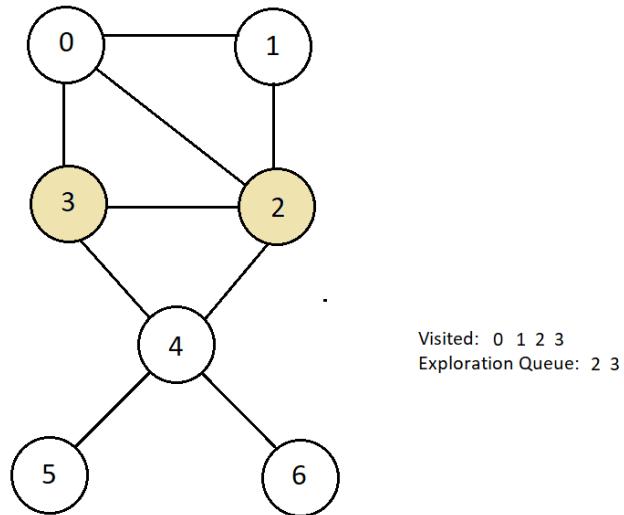


Now, we'll start visiting all the nodes connected to node 0, and remove node 0 from the exploration queue, enqueueing all the currently visited nodes which were nodes 1, 2, and 3. We are pushing them inside the exploration queue because these might

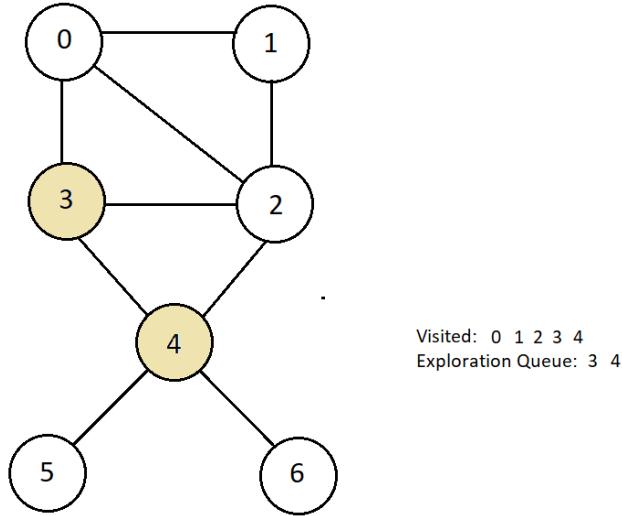
further have some unvisited nodes connected to them. Mark these nodes visited as well.



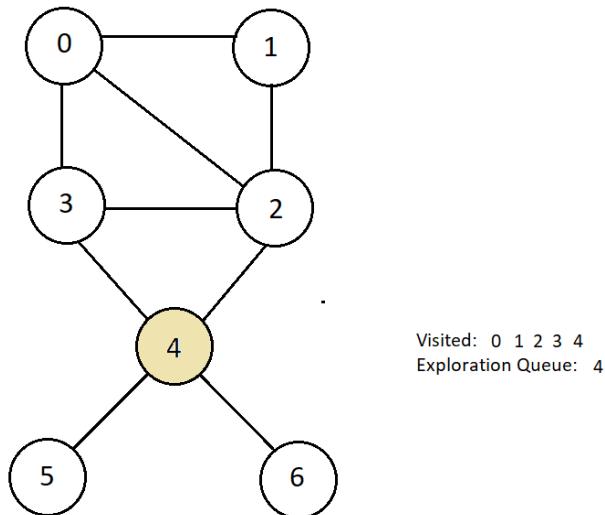
After this, we have node 1 at the top in the exploration queue, so we'll take it out and visit all unvisited nodes connected to it. Unfortunately, there aren't any. Therefore, we'll continue exploring further.



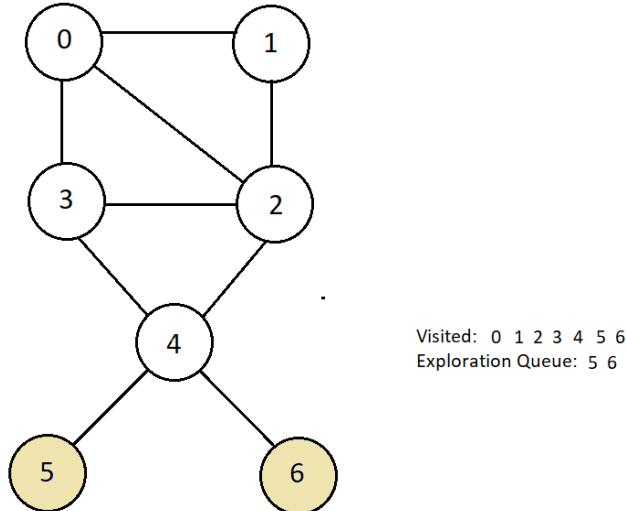
Next, we have node 2. And the only unvisited node connected to node 2 is node 4. So, we'll mark it visited and will also enqueue it in our exploration queue.



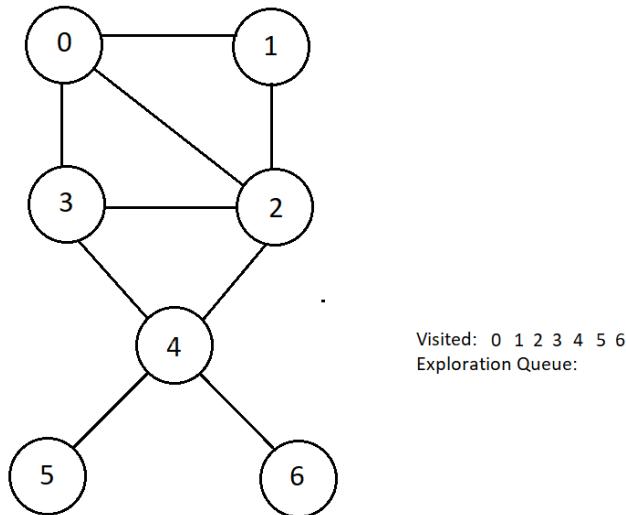
Node 3 is the next in line. Since, all nodes 1, 2, and 4 which are the nodes connected to it are already visited, we have nothing to do here while we are on node 3.



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.



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.



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. In the next lecture, we'll discuss it in more detail.

1. We'll take a whole graph and the information about its nodes and edges as the input along with the source node s .
2. We'll mark the node s visited and then create a queue, and enqueue s in it.

3. 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.
4. Below is the pseudocode, and I encourage you all to read it and try to implement the same in C.

```

- 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:

1. We can start with any vertex.
2. There can be multiple Breadth-First Search results for a given graph
3. The order of visiting the vertices can be anything.
4. **Quiz:** Try to find another valid Breadth-First Search for the same graph we discussed above. (Hint: Start with nodes other than 0).

We are now just left with the programming part, which will be covered super soon in the next lecture. Keep revising the concept until then.

BFS Implementation in C | C Code For Breadth First Search

In the last lecture, we learned about the concepts of Breadth-First Search in graph traversal. We saw two of its methods, first one using the BFS spanning tree, and the second one being one of the most conventional ones where we maintained a visited array and an exploration queue while we explored new nodes to see what we have visited already and what we have yet to explore respectively. Today we will implement Breadth-First Search in C language.

If you recall, we gave you all a pseudocode in the end and asked you all to read and try implementing the same in C. So, let me know if you could. The pseudocode was:

```

- 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);
    }
}

```

1. Now, the first thing we did was, we took the input. The input comprises the information concerning the graph, its nodes/vertices, and edges, and the source node we'll start the traversal with.
2. Then, we'll mark the source node s visited and then create an exploration queue, and enqueue the source code s in it.
3. 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 using the utility function dequeue and suppose that element is u . Then, we visit all the vertices which are directly connected to u and are already not visited.
4. And every time we visit a new node, we enqueue them in our exploration queue. Eventually, the queue becomes empty, and our traversal ends. And the visited array is the order of our Breadth-First Search traversal.

So, this was the theory part. We are now ready to move on to our programming segment. Having said that, let's move directly to our editors. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. The first thing you should do is, bring the implementation part of the queue from our previous lectures. In this way, we can save time and reduce repetition. I have literally copied everything from my previous snippet including the queue, and its utility functions implemented using arrays.

Initializing a queue:

2. Next thing you should do is initialize a queue, and dynamically assign it a memory in heap using malloc and define its size, say 400. Mark both its front and rear at zeroth index.

```
// Initializing Queue (Array Implementation)
struct queue q;
```

```
q.size = 400;  
q.f = q.r = 0;  
q.arr = (int*) malloc(q.size*sizeof(int));
```

Copy

Code Snippet 1: Initializing a queue

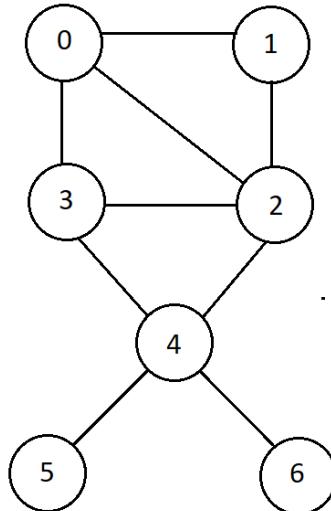
BFS Implementation:

3. Before we start programming the BFS implementation of the graph, we should first define a graph. And out of all the methods we have studied to represent a graph, we will be using the adjacency matrix one to define our graph here.

4. Define an integer variable *node* which we will use to traverse the graph. Define another integer variable *i* which is the node we are currently at. So, since we would be the default starting with any node of our choice, we would initialize it with node 1.

5. Now, define an integer array *visited* to store the status of a node (of size 7 here), and the values corresponding to a node is 0, if it is unvisited and 1, if it is visited. And since, no node is already visited when we first start, therefore we'll fill this array with all zeros.

6. Create an adjacency matrix corresponding to the graph I've illustrated below.



An adjacency matrix holds a value 1 for a cell that is at the intersection of the *i*th row and *j*th column if there is an edge between node *i* and node *j*. So, the adjacent matrix corresponding to the graph above is:

	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

7. Next, we would print the source node we have chosen, here node 1. Also, mark this node visited, which means make the i th index of the visited array 1. And since we are now interested in exploring the nodes connected to this source node, we would first enqueue this node into the queue q we created in step 2.

8. Create a while loop, and make it run while our queue is not empty. Queue becoming empty is the condition where we finish our traversal. We use the utility function `isEmpty` for the same.

Inside this loop, pluck the top element of the queue using `dequeue` and store it in an integer variable `node`. Run another `for` loop using `j` as the iterator, and since the size of our graph is 7, we'll make this loop run from 0 to 7.

Now, every time we find edges `node & j` connected (that is cell `a[node][j]` equal to 1) and the node `j` unvisited, we mark it visited, and enqueue this newly visited node `j` in the queue. We'll print these nodes while we visit them to determine the BFS traversal order of the graph.

This way we will explore all the nodes, and the queue will ultimately become empty.

```
// BFS Implementation
int node;
int i = 1;
int visited[7] = {0,0,0,0,0,0,0};
int a [7][7] = {
    {0,1,1,1,0,0,0},
    {1,0,1,0,0,0,0},
    {1,1,0,1,1,0,0},
    {1,0,1,0,1,0,0},
```

```

    {0,0,1,1,0,1,1},
    {0,0,0,0,1,0,0},
    {0,0,0,0,1,0,0}
};

printf("%d", i);
visited[i] = 1;
enqueue(&q, i); // Enqueue i for exploration
while (!isEmpty(&q))
{
    int node = dequeue(&q);
    for (int j = 0; j < 7; j++)
    {
        if(a[node][j] == 1 && visited[j] == 0){
            printf("%d", j);
            visited[j] = 1;
            enqueue(&q, j);
        }
    }
}
}

```

[Copy](#)

Code Snippet 2: Implementing Breadth-First Search

Here is the whole source code:

```

#include<stdio.h>
#include<stdlib.h>

struct queue
{
    int size;
    int f;
    int r;
}

```

```
    int* arr;  
};
```

```
int isEmpty(struct queue *q){  
    if(q->r==q->f){  
        return 1;  
    }  
    return 0;  
}
```

```
int isFull(struct queue *q){  
    if(q->r==q->size-1){  
        return 1;  
    }  
    return 0;  
}
```

```
void enqueue(struct queue *q, int val){  
    if(isFull(q)){  
        printf("This Queue is full\n");  
    }  
    else{  
        q->r++;  
        q->arr[q->r] = val;  
        // printf("Enqueued element: %d\n", val);  
    }  
}
```

```
int dequeue(struct queue *q){  
    int a = -1;
```

```
if(isEmpty(q)){
    printf("This Queue is empty\n");
}
else{
    q->f++;
    a = q->arr[q->f];
}
return a;
}
```

```
int main(){
    // Initializing Queue (Array Implementation)
    struct queue q;
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
    int i = 1;
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };
    printf("%d", i);
```

```

    visited[i] = 1;
    enqueue(&q, i); // Enqueue i for exploration
    while (!isEmpty(&q))
    {
        int node = dequeue(&q);
        for (int j = 0; j < 7; j++)
        {
            if(a[node][j] == 1 && visited[j] == 0){
                printf("%d", j);
                visited[j] = 1;
                enqueue(&q, j);
            }
        }
    }
    return 0;
}

```

[Copy](#)

Code Snippet 3: Implementing BFS using queue in C

We'll see if our algorithm actually works, and if it reverts the Breadth-First Search traversal order of the graph we fed into the program. As you could observe, we have started our traversal considering node 0 as the root node. And when we ran the program, the output we received was:

```
0123456
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

[Copy](#)

Figure 1: Output when the root node is 0

But if we change our root node from 0 to 1, the output changes to:

```
1023456
```

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

Figure 2: Output when root node is 1

I encourage you all to try this yourself. It shall work for any node you start with, and any graph you feed into the program. I hope I made this clear for you all. You can go through the lectures again for a better understanding of things. Depth First Search is what we'll cover next.

Depth First Search (DFS) Graph Traversal in Data Structures

In the last lecture, we completed learning about the first method of graph traversal which was the Breadth-First Search as well as its implementation in C. Today, we'll be dealing solely with the Depth First Search abbreviated as DFS in greater detail. Before we actually move on to writing the algorithm of the Depth First Search or programming its implementation, we'll first visualize how Depth First Search works. As we know, graph traversal becomes impossible when done manually for very large graphs. Therefore, some algorithms are used to automate this task. One such algorithm we learned was BFS, and another one, which we would learn today is the Depth First Search.

What is Depth First Search?

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.

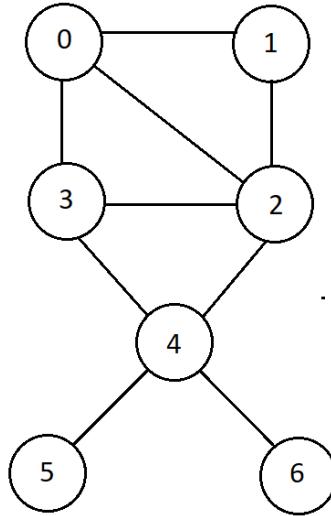
In BFS, we explore all the nodes connected to a node, then explore the nodes we visited in a horizontal manner, while in DFS, we start with the first connected node, and similarly go deep, so this looks like visiting the nodes vertically. This might sound confusing for now, so let's make it more intuitive for you to understand.

DFS Procedure:

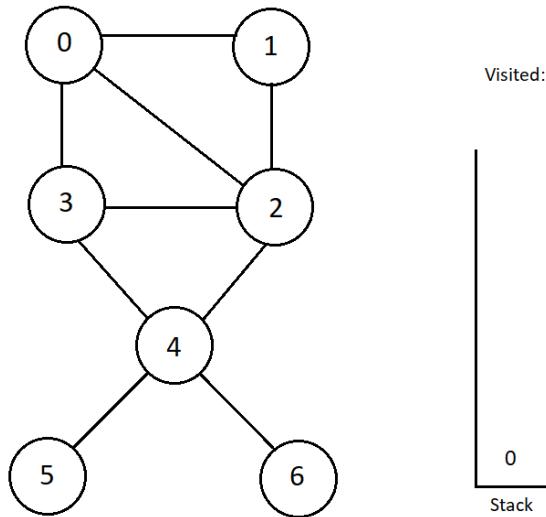
The procedure we follow to accomplish the depth-first search traversal of a graph is:

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 you remember, is a data structure in which the element you 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.

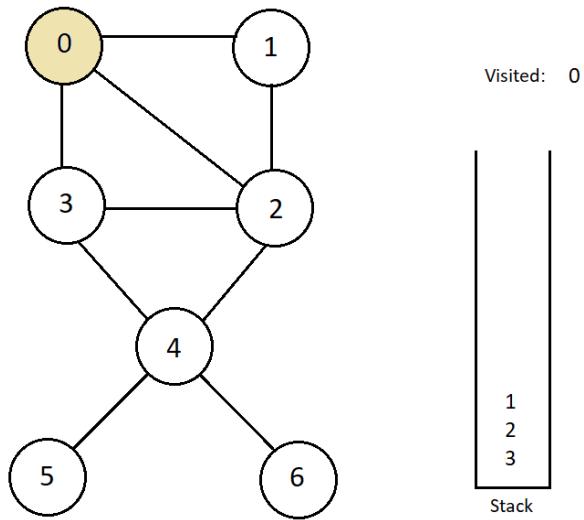
To understand the procedure of the Depth First Search, consider the graph I've illustrated below. For better contrast, I took the same graph as in the last lecture.



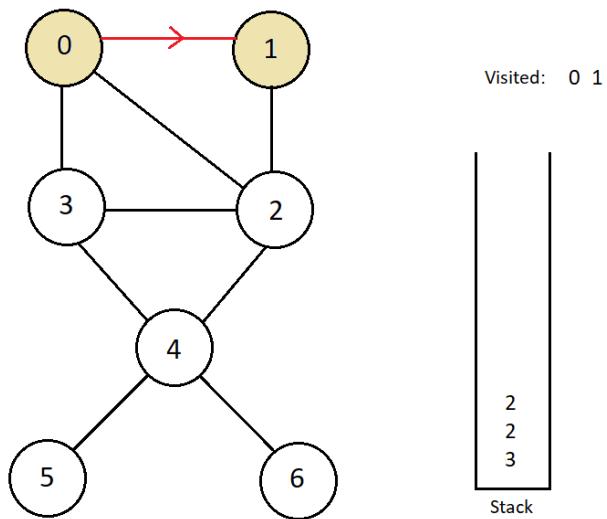
Considering the fact that we could begin traversing with any source node; we'll start with 0 only. So, following step1, we would push this node into the stack and begin our Depth First Search traversal.



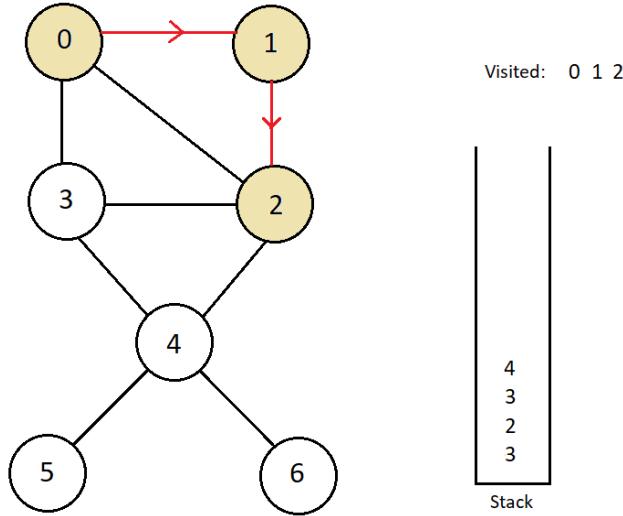
The next step says, pop the top element from the stack which is node 0 here, and mark it visited. Then, we'll start visiting all the nodes connected to node 0 which are not visited already, but before that, we are asked to push them all into the stack, and the order in which you push doesn't matter at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



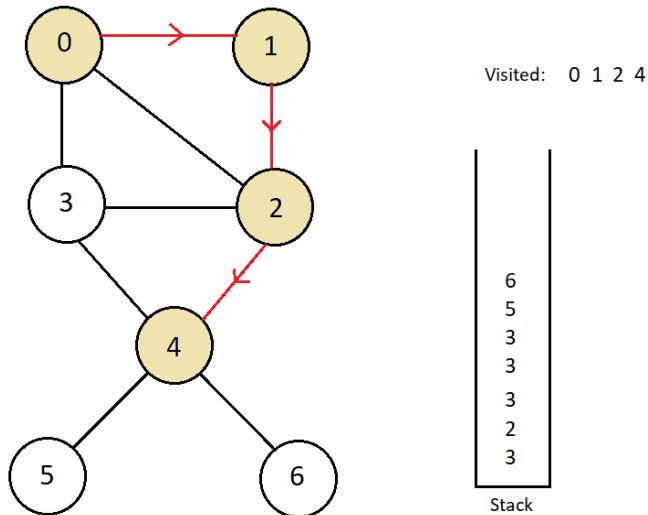
Repeating the steps above, we'll now pop the top element from the stack which is node 1, and mark it visited. Only nodes connected to node 1 were nodes 0 and 2, and since the only unvisited one is node 2. It's important to observe here, that although node 2 is in the stack, it is not visited. So, we'll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



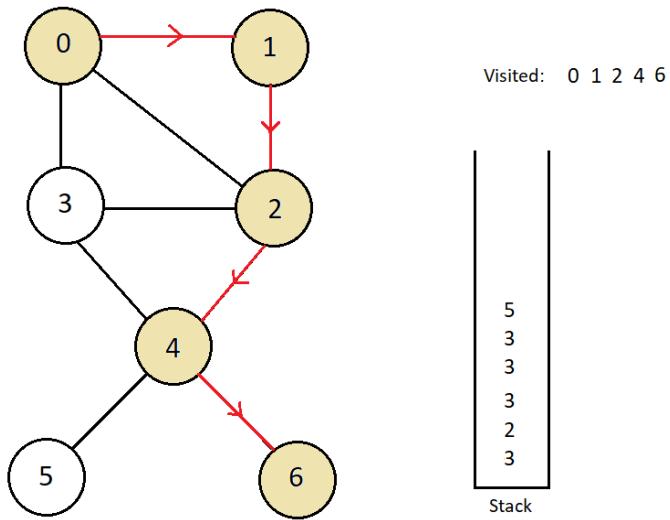
Next, we have node 2 at the top of the stack. We'll mark node 2 visited and unvisited nodes connected to node 2 are nodes 3 and 4, regardless of the fact that 3 is already there in the stack. So, we'll just push nodes 3 and 4 into the stack.



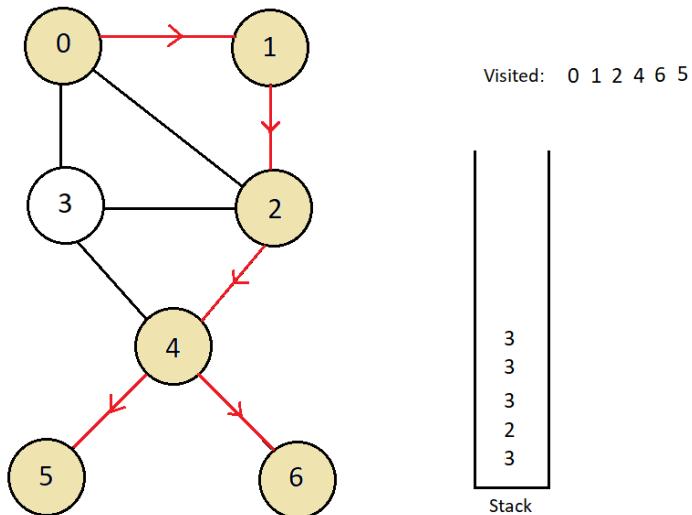
Node 4 is the next we have on the top. So, just mark it as visited. Since, all nodes 3, 5, and 6, except node 2, which are directly connected to it are not visited, we'll push them into the stack.



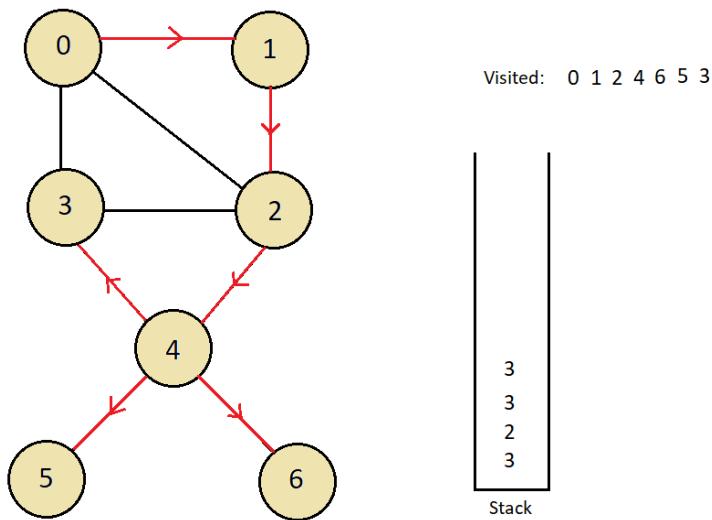
Next, we have node 6 on the top of the stack. Pop it and mark it visited. Since there are no nodes that are directed connected to node 6 and unvisited, we'll continue further without doing anything.



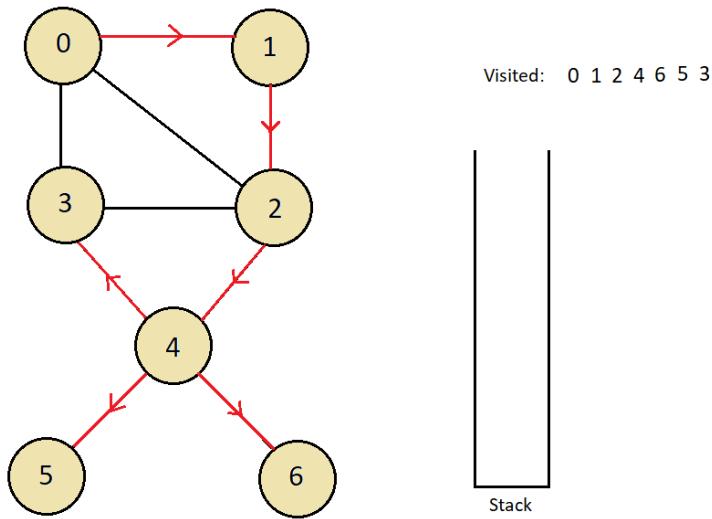
Next, we pop node 5 out of the stack and mark it visited. And since there is no unvisited node connected to it, we continue.



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 you 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. I will tell you how in the next lecture.

Let's now see how the process we did manually above can be automated in C although we will be using pseudocode for now. In the next lecture, we'll discuss it in more detail.

- ```
- 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)
```

We are now just left with the programming part, which will be covered in the next segment. Keep practicing the concepts until then. Try writing the algorithm in C yourselves using the pseudocode I have provided above. We would anyway do that in the next lecture.

## DFS Implementation in C | C Code For Depth First Search

In the last lecture, we learned about the concepts of Depth First Search in graph traversal. We discussed the differences we observed in both the graph traversal methods. We discussed the algorithm to automate the DFS traversal of a graph. Today we will implement the same Depth First Search algorithm in C language. Implementing Depth First Search, as I told you all in the last lecture, is one of the easiest jobs to do. DFS implementation asks you to use stacks, but if you know, a function call itself uses a memory stick, so calling a function recursively will do our work. And If you recall, we gave you all a pseudocode in the end and asked you all to read and try implementing the same in C. So, let me know if you could. The pseudocode was:

- 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)

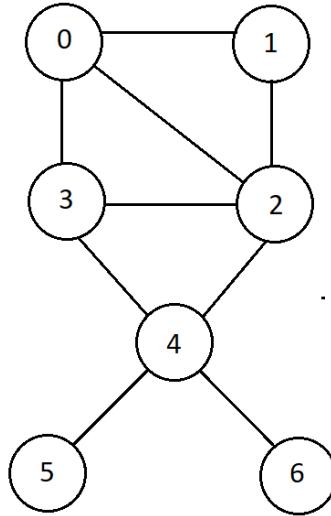
```

1. Now, the first thing we did was, we took the input. The input comprises the information concerning the graph, its nodes/vertices, and edges, and the source node we'll start the traversal with.
2. Then, we'll call the function DFS feeding a source node into it.
3. In the function, we'll mark the node visited, and then initiate a for loop which will access the connected nodes to this node, and see if they are visited or not.
4. And every time we find a node connected and not visited already, we call the DFS function recursively filling out our function stack. Eventually, the function stack becomes empty, and our traversal ends. And the visited array is the order of our Depth First Search traversal.

Well, this was the theory part. We're now ready to get started on the programming part. Let's move directly to our editors. I have attached the source code below. Follow it as we proceed.

#### **Understanding the source code below:**

1. Since we will be using functions to be able to use stacks directly, we don't really have to bring the implementation part of the stack from our previous lectures. Let's jump to our DFS implementation directly.
2. Before we start programming the DFS implementation of the graph, we should first define a graph. And out of all the methods we have studied to represent a graph, we will be using the adjacency matrix one to define our graph here. And since we have already done this in our last programming lecture where we implemented the Breadth-First Search, we'll bring the same graph here as well. This would save us a lot of time. So, basically, we got ourselves an adjacency matrix  $A[7][7]$  corresponding to the graph I've illustrated below.



3. Now, define an integer array *visited* to store the status of a node (of size 7 here), and the values corresponding to a node is 0 if it is unvisited and 1 if it is visited. We will fill this array with all zeros since no node has already been visited when we start.

#### **Creating function DFS:**

4. Now, we wish to define a function that would carry the Depth First Search traversal of the graph for us. So, create a void function *DFS* which will have the node *i* we want to visit as the only parameter. Inside that function, since that node which the function received as the parameter is the one to be visited now, we would mark it visited and print it at the same time.

5. Next step requires you to see all nodes connected directly to this node. Run a for loop of size 7 with an iterator *j*, since that is the number of nodes we have. If any of them is not visited and there is an edge in between *i* and *j*, we would recursively call *DFS* passing this node *j*. And that's all. Trust your function and it will do things in its own way. Our job has finished.

```
void DFS(int i){
 printf("%d ", i);
 visited[i] = 1;
 for (int j = 0; j < 7; j++)
 {
 if(A[i][j]==1 && !visited[j]){
 DFS(j);
 }
 }
}
```

[Copy](#)

### Code Snippet 1: Creating the DFS function

Here is the whole source code:

```
#include<stdio.h>
#include<stdlib.h>

int visited[7] = {0,0,0,0,0,0,0};
int A [7][7] = {
 {0,1,1,1,0,0,0},
 {1,0,1,0,0,0,0},
 {1,1,0,1,1,0,0},
 {1,0,1,0,1,0,0},
 {0,0,1,1,0,1,1},
 {0,0,0,0,1,0,0},
 {0,0,0,0,1,0,0}
};

void DFS(int i){
 printf("%d ", i);
 visited[i] = 1;
 for (int j = 0; j < 7; j++)
 {
 if(A[i][j]==1 && !visited[j]){
 DFS(j);
 }
 }
}

int main(){
 // DFS Implementation
}
```

```
 DFS(0);
 return 0;
}
```

Copy

### Code Snippet 2: Implementing DFS using recursive function in C

We'll see if our algorithm actually works, and if it reverts the Depth First Search traversal order of the graph we fed into the program. If you could observe, we have started our traversal considering node 0 as the root node. And when we ran the program, the output we received was:

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

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

### Figure 1: Output when the root node is 0

But if we change our root node from 0 to 4, the output changes to:

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

```
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Copy

### Figure 2: Output when the root node is 4

And this is how we implemented Depth First Search traversal in C without even explicitly using stacks just via a recursive function. You should all give it a shot too. No matter what node you start with, and whatever graph you feed the program with, it will work. Hopefully, that was clear. You can go through the lectures again for a better understanding of things. Stay tuned, as we are about to dive deep into graph theory.

## Spanning Trees & maximum no of possible spanning trees for complete graphs

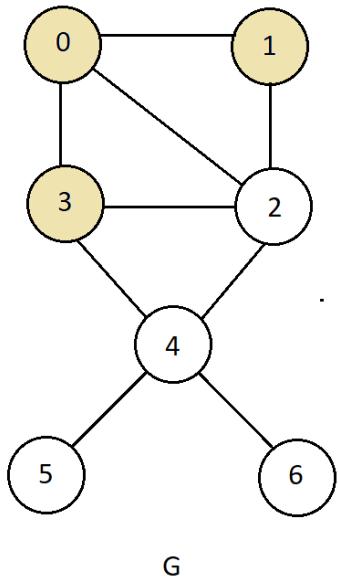
In the last lecture, we finished learning both the traversal algorithms, the Breadth-First Search and the Depth First Search. We implemented both of these in C using queues and stacks respectively. Today, we'll learn what spanning trees are, and the concepts behind them, and other small topics

such as connected graphs and others needed to understand spanning trees.

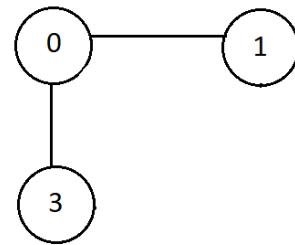
While we'll be learning more advanced topics, you will realize that applications of the algorithms we wrote BFS and DFS are not limited to just finding the traversal order of a graph, rather much more. Let's first start with what spanning trees are. But before we dig into spanning trees, we should talk about a few things you should know. First one being, a subgraph.

### **Subgraphs:**

A subgraph of a graph G is a graph whose vertices and edges are subsets of the original graph G. It means that if we consider the graph G illustrated below, then its subgraphs could be the graph S.

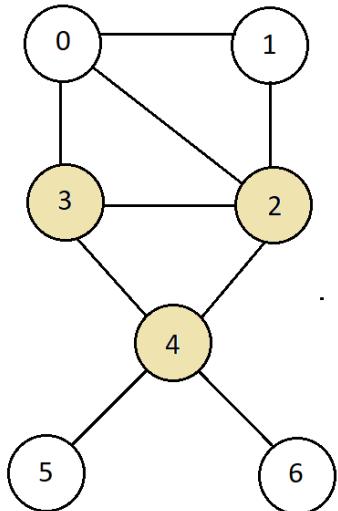


G

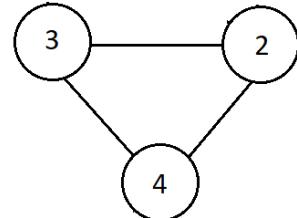


S

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.



G



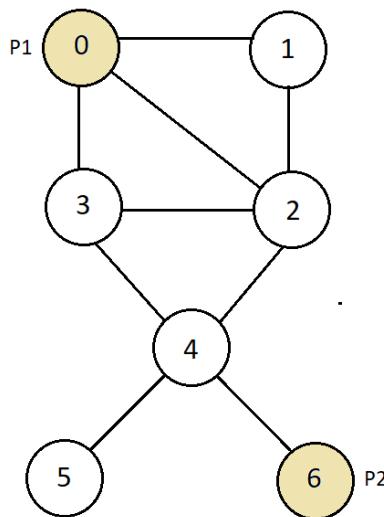
S

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.

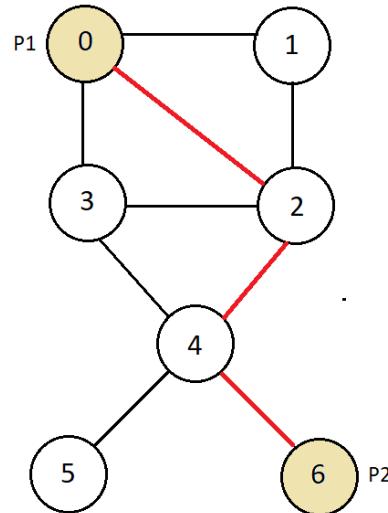
That should be enough for you to understand what subgraphs of a graph are. Moving ahead, we'll now talk about connected and complete graphs.

### Connected Graphs:

A connected graph, as the word connected suggests, is a graph that is **connected** in the sense of a topological space, i.e., 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:



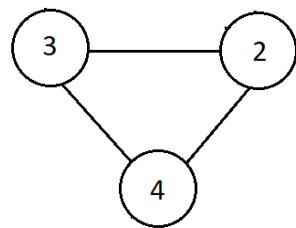
We have marked two random points P1 and P2, and we'll see if there is a path from P1 to P2. And if you could see, there is indeed a path from P1 to P2 via nodes 2 and 4.



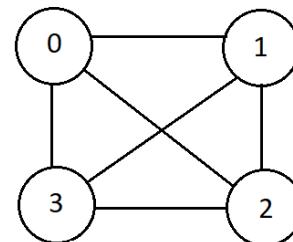
There could be a number of other ways to reach point P2 from P1, but there should exist at least one path from any point to another point for the graph to be called connected, otherwise disconnected. You should check for some other pair of vertices in the above graph.

### Complete Graphs:

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



Complete graph with 3 nodes

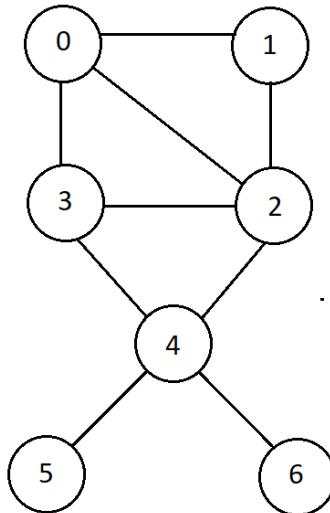


Complete graph with 4 nodes

As you 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. You 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.

**Quick Quiz:**



1. Is the graph illustrated above connected?
2. Is the same graph complete?

**Hint:** Nodes 3 and 1 are not directly connected.

And now, having done subgraphs, connected and complete graphs, we are good to learn about spanning trees. Let's proceed with them.

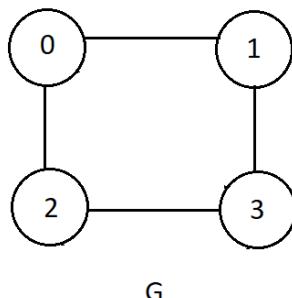
**What are spanning trees?**

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:

1. All the vertices of G are present in S,
2. No. of edges in S should be  $|V|-1$ , where  $|V|$  represents the number of vertices.

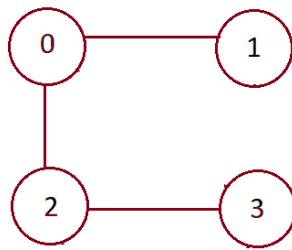
That is, for a **subgraph** of a graph to be called a **spanning tree** of that graph, it should have **all vertices of the original graph** and must have exactly  **$|V|-1$  edges**, where  $|V|$  represents the number of vertices and the graph should be **connected**.

You might find this difficult to comprehend, but let me give you an example to help you understand. Consider a simple graph G I have illustrated below:



G

Now, consider another graph S.



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.

- Is graph S a subgraph of graph G? ✓

Yes, 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.

- Is graph S connected? ✓

Yes, graph S is connected since we can go from any node to any other node via some edges in the graph.

- Are all vertices of graph G present in graph S? ✓

Yes, all vertices of graph G which are vertices 0, 1, 2, and 3, are present in graph S.

- Does the number of edges in graph S equal the number of vertices in graph G - 1? ✓

Yes, 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 you are given a graph and are asked whether this graph is a spanning tree of another graph or not, you 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.

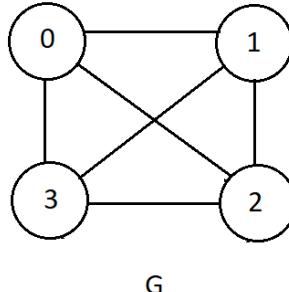
And if you are asked to create a spanning tree of a graph, you must first plot all the vertices of the original graph. And then create V-1 edges which were there in the original graph and what makes your graph connected.

This would be sufficient, and you can just ignore all other edges present in the original graph.

Now, you must be wondering about 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 I have made below with 4 nodes.

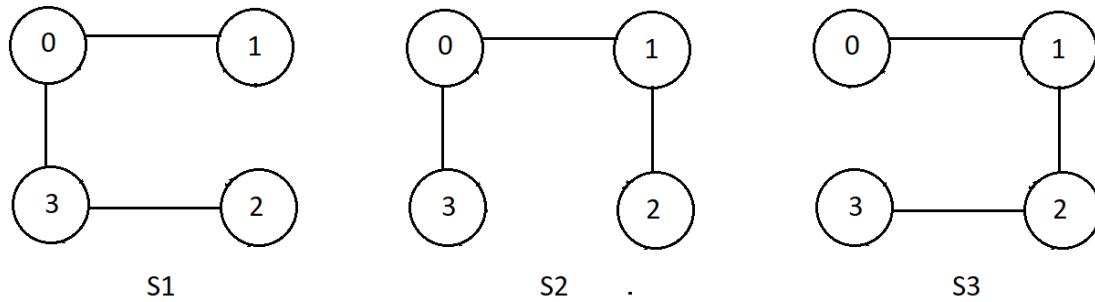


This complete graph has 4 vertices, hence the number of spanning trees it can have is, 4 raised to the power (4-2), i.e.,  $4^2$  which is 16.

### **Quick Quiz:**

1. Draw any 3 spanning trees of the graph illustrated above.

**Answer:**



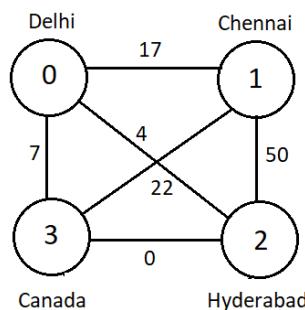
So, that was it for today. We learned what spanning trees are and other graph terminologies. Yet, so much more is left to the present. Next, we will learn about minimum spanning trees and spanning tree cost calculation, and more. Stay tuned.

## Calculating Spanning tree Cost & Minimum Spanning Tree

In the last lecture, we introduced to you the concept of a spanning tree. We discussed the conditions required for a graph to be called a spanning tree for another graph and several other terminologies related to graphs. Today, we'll delve deep into spanning trees and see their applications as well as learn what minimum cost spanning trees are.

For easy understanding of things, I'll walk you all 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. Let's give ourselves a quick revision of a few things:

- **Connected Graphs:**

A connected graph is a graph where 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.

- **Complete Graphs:**

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

- **Spanning trees:**

We learned that any graph S which satisfies below mentioned four conditions would be considered a spanning tree for another graph G. Those four conditions were:

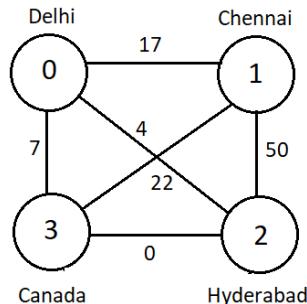
1. Graph S is a subgraph of graph G.
2. Graph S is a connected graph.
3. All vertices of graph G are present in graph S.
4. The number of edges in graph S equals the number of vertices in graph G - 1.

Having done the revision, we'll now look to proceed with what we mean by the cost of a spanning tree.

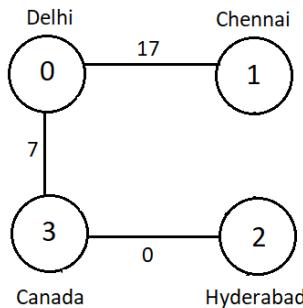
#### **Cost of a Spanning Tree:**

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.

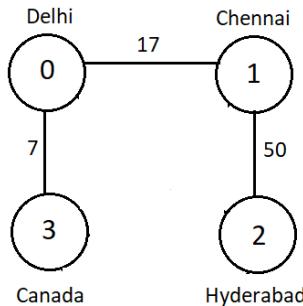


Now, suppose we created a spanning tree out of it which I've illustrated below.



S1

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 of the same graph be:



S2

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. And therefore, Prem would definitely not want to incur some cost greater than 24. Let's move ahead now and see what a minimum spanning tree is.

#### **Minimum spanning tree:**

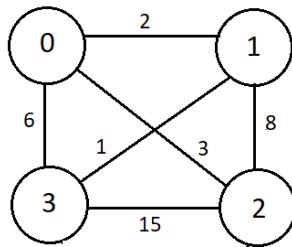
A Minimum Spanning tree, abbreviated as MST, is the spanning tree having the minimum cost. A graph can have a number of spanning trees all having some cost, say S1, S2, S3, S4, and S5 having cost 100, 104, 500, 400, and 10 respectively, but the one incurring the minimum cost is called the minimum spanning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

#### **Applications of Minimum Spanning Tree:**

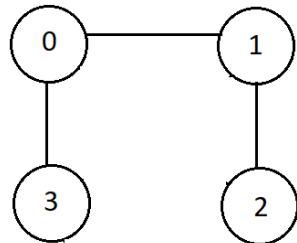
Applications of a minimum spanning tree must have gotten somewhat clear from the example of Prem, where we want to traverse all the nodes, which means the graph remains connected but it must have only bare minimum costing edges. So, this was a basic application of a minimum spanning tree. Things will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

#### **Exercise:**

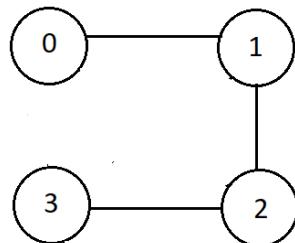
- Find the cost of any 3 spanning trees of the graph given below.



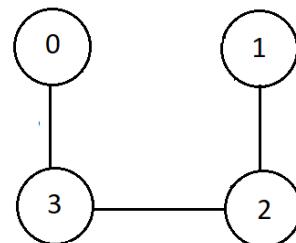
Consider the given graph be  $G$ , and our first step would be to create three spanning trees  $S_1, S_2$ , and  $S_3$  of the graph  $G$  which we have already learned in the last class.



$S_1$

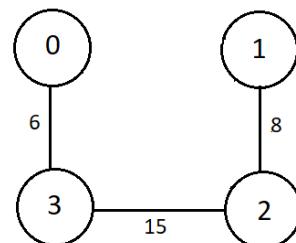
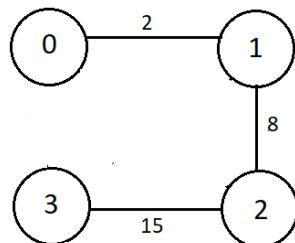
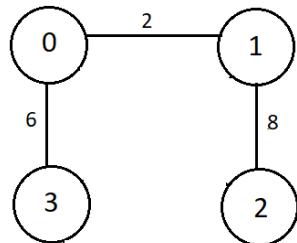


$S_2$



$S_3$

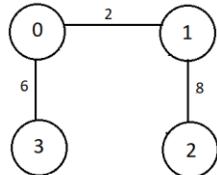
Then, mention weights to all the edges.



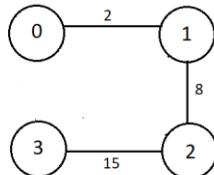
Now, let  $C_1, C_2$ , and  $C_3$  be the cost of spanning trees  $S_1, S_2$ , and  $S_3$  respectively. Therefore,  $C_1$  is  $(6+2+8) = 16$ ,  $C_2$  is  $(2+8+15) = 25$  and  $C_3$  is  $(6+15+8) = 29$ . Hence the cost of these spanning trees is 16, 25, 29 respectively. One thing we can infer from this is,  $C_1$  is less than  $C_2$ , and  $C_2$  is less than  $C_3$ , i.e.,  $C_1 < C_2 < C_3$ . As a result,  $C_1$  is preferred to both  $C_2$  and  $C_3$ , and  $C_2$  is preferred to  $C_3$ .

- Find the minimum spanning tree of the same graph we considered in the previous exercise.

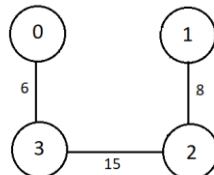
Since graph G is a complete graph and it has a total of four nodes, the total number of spanning trees possible for graph G would be 4 raised to (4-2), which is 42, 16. So, the first step would be to create all the 16 possible spanning trees. All of them are illustrated below along with the edge weights.



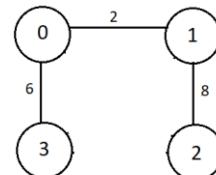
S1



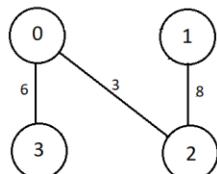
S2



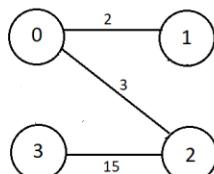
S3



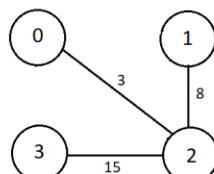
S4



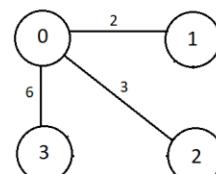
S5



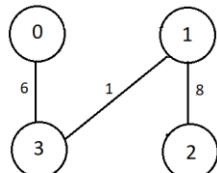
S6



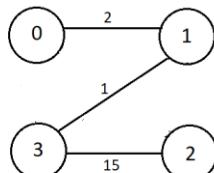
S7



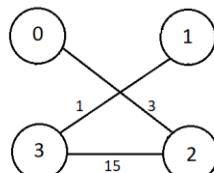
S8



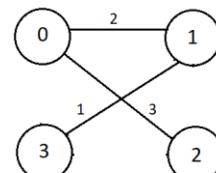
S9



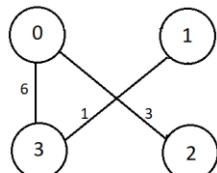
S10



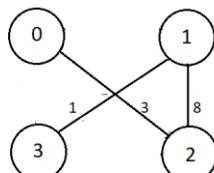
S11



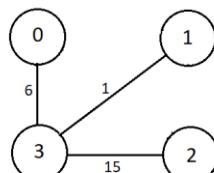
S12



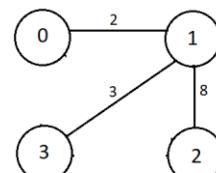
S13



S14



S15



S16

So, these were all the possible spanning trees we had. The rest I'll leave to you. You first have to count the cost of all these spanning trees, and then choose the minimum out of them. Tell me in the comments if you did all of them. I'll be more than happy to know if you could.

## Prims Minimum Spanning Tree Algorithm (Step by Step with examples)

In the last lecture, we learnt about the cost of a spanning tree, and we discussed what a minimum spanning tree is along with some of its applications. Today, we will see Prim's algorithm which is one of the algorithms to find the minimum spanning tree of a graph.

If you remember, we took the example of *Prem* and solved his dilemma of choosing the best possible route to traverse every city in search of his beloved while spending the least possible money. It is here that his big brother, *Prim*, steps in to help. His brother, *Prim*, has an algorithm named after him, the Prim's algorithm.

Rather than diving into the details of the algorithm right now, let's take a moment to revisit the concepts we've learned so far.

**Connected graphs** are graphs where every node in the graph has a path to every other node. A **complete graph** is one in which every pair of nodes has a unique edge. You should know those four conditions for a tree to be a **spanning tree** of some other graph. The **Cost of a spanning tree** is the sum of the weights of all the edges in the tree. And a **minimum spanning tree** has the minimum cost over all spanning trees of a graph.

**Quick Quiz:** Why is a spanning tree called '*spanning*' 'tree'?

**Hint:** It spans over the whole graph, and it is a tree.

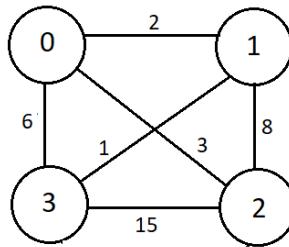
This gives us the green light to move forward with Prim's algorithm.

### Prim's 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:

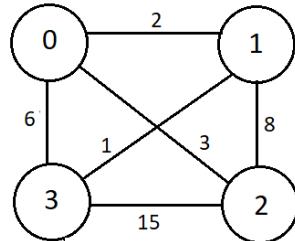
1. We start with any node and start creating the Minimum Spanning Tree.
2. In Prim's Algorithm, we grow the spanning tree from a starting position until  $n-1$  edges are or all nodes are covered.

Let me now take a simple example. Consider the complete graph with 4 vertices,  $K_4$ , illustrated below:

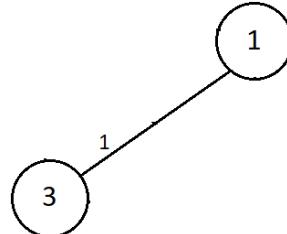


$K_4$

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.

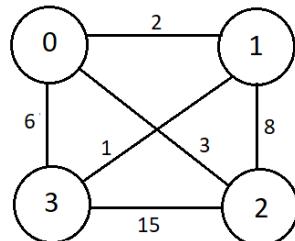


$K_4$

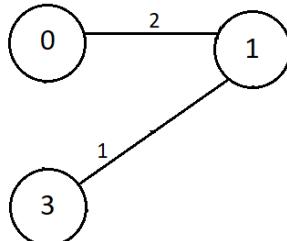


Spanning Tree

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.

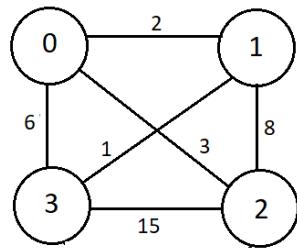


$K_4$

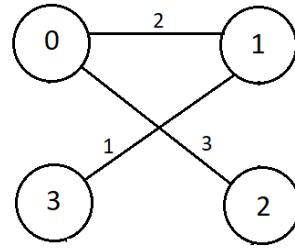


Spanning Tree

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.



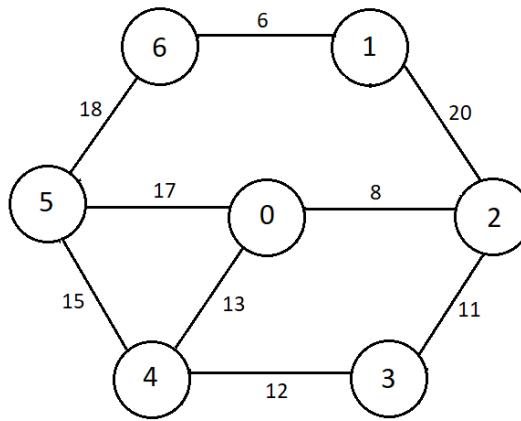
$K_4$



Spanning Tree

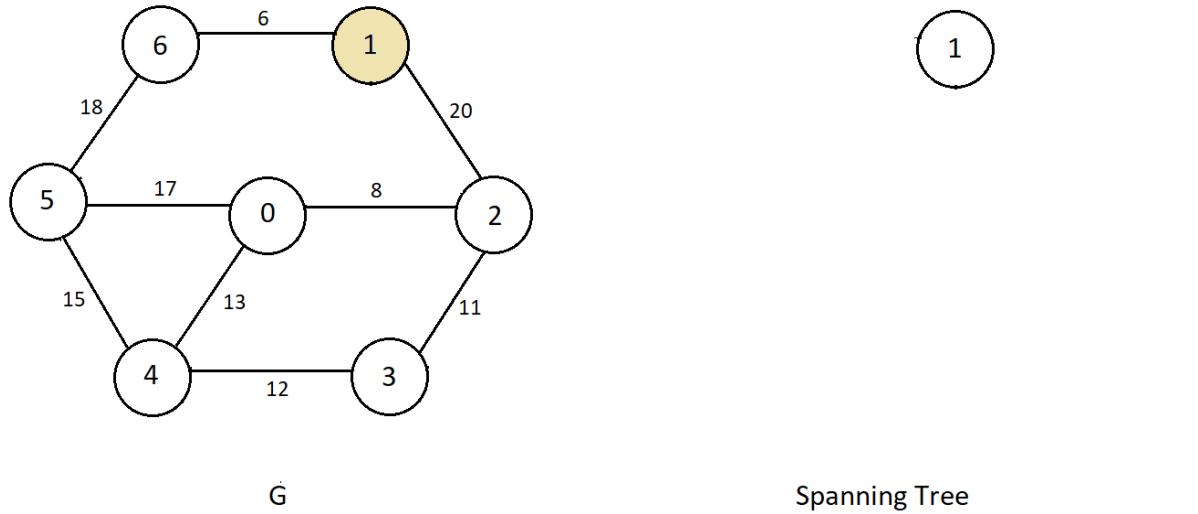
The cost of this spanning tree is  $(1+2+3)$ , i.e., 6 which is the minimum possible. And that finishes our construction of the spanning tree for graph  $K_4$ . And this is exactly the way Prim's algorithm works.

Now, I'll give you one relatively tough example to get a better feel of the algorithm. Consider the graph I've illustrated below.



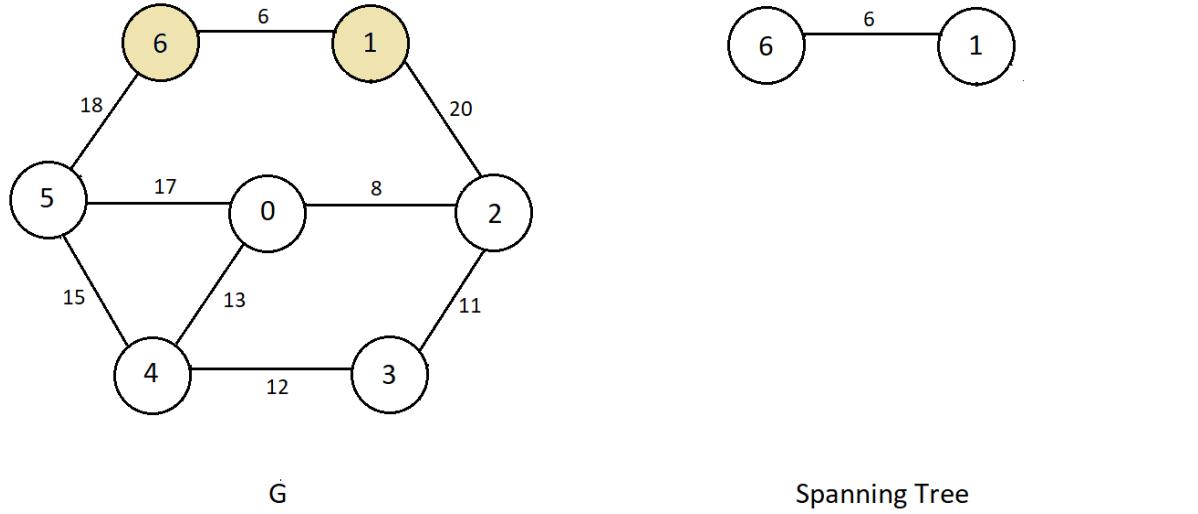
$G$

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.



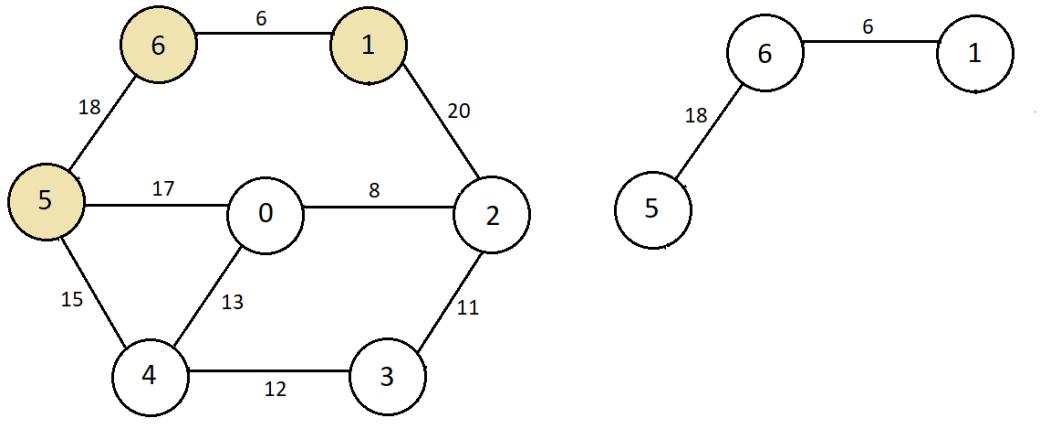
Spanning Tree

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.



Spanning Tree

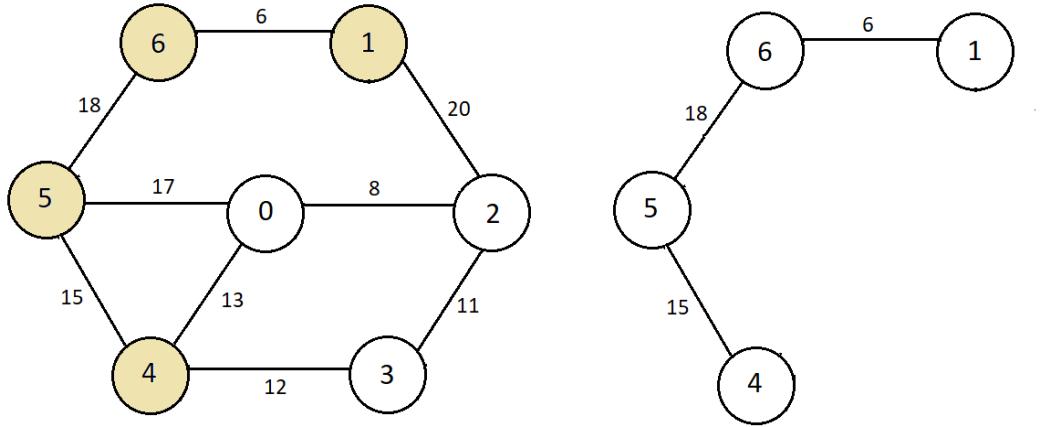
Now, 6 gets added to the set containing the covered elements. Nodes possible to get connected to the current spanning tree are 5 and 2. Node 5 and node 2 are connected to node 6 and node 1 respectively through edges of weights 18 and 20. We will obviously choose the one with less weight, i.e., weighted edge 18.



G

Spanning Tree

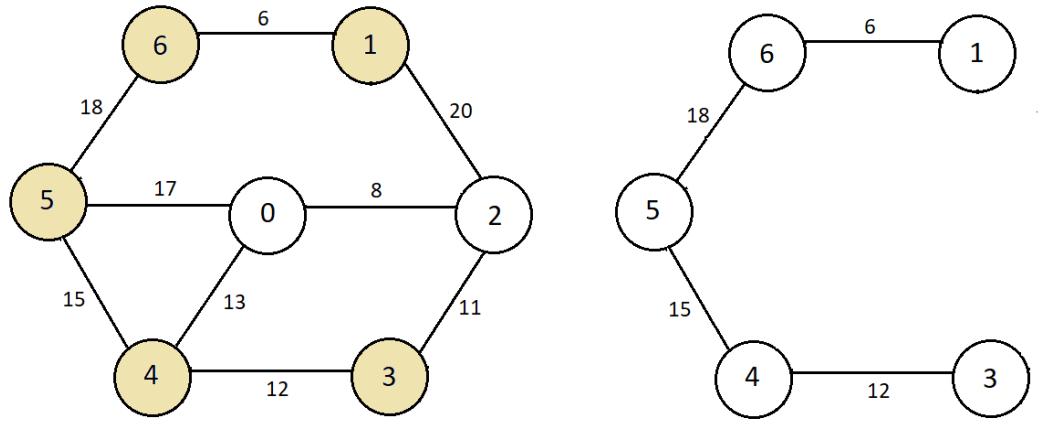
Now, 5 gets added to the set containing the covered elements. Possible nodes to get connected to the current spanning tree are 4, 0, and 2. Node 4 and node 0 are connected to node 5 through edges of weights 15 and 17 and node 2 is connected to node 1 through an edge of weight 20. We will, therefore, 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

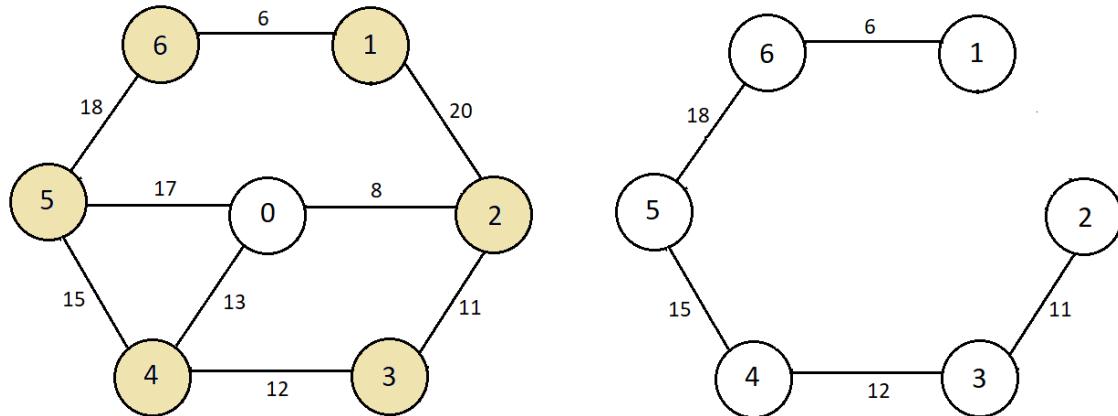
Now, we have nodes 0, 2, and 3 as possible candidates to get connected to the current spanning tree. Node 0 is connected to node 5 and node 4 through edges of weights 17 and 13 respectively. Node 3 is connected to node 4 through an edge of weight 12, and node 2 is connected to node 1 through an edge 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.



G

Spanning Tree

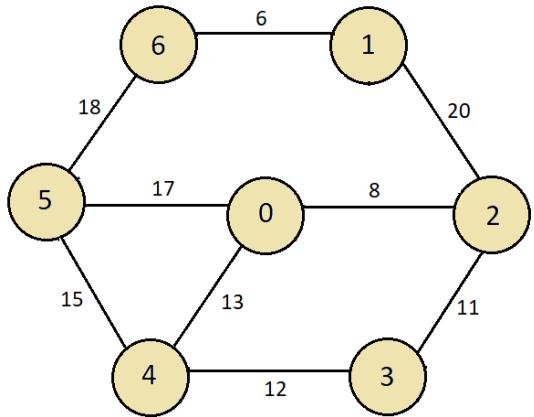
Next, we are left with only two nodes to cover. Node 0 is connected to node 5 and node 4 through edges of weights 17 and 13 respectively. Similarly, Node 2 is connected to node 1 and node 3 through edges of weights 20 and 11 respectively. Therefore, we'll choose the one with weight min (17, 13, 20, 11) i.e., 11. Hence, node 2 gets added to the current spanning tree through node 3.



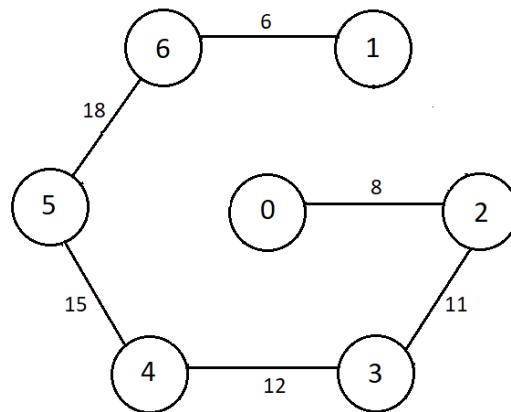
G

Spanning Tree

The only node left now is node 0. We have 3 possible ways to connect this node to the rest of the spanning 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.



G



Spanning Tree

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 you 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