**Introduction to Data Structures & Algorithms**

**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:

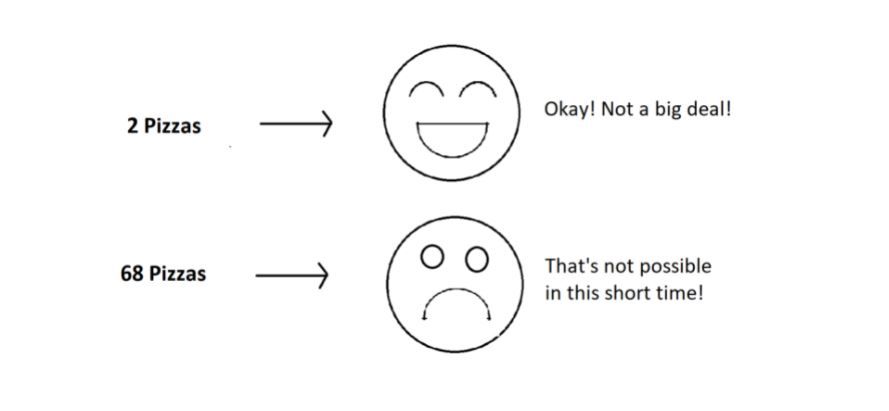


So, this was all for the beginning. Data Structures and Algorithms are not new concepts. If you have done programming in any language like C, you must have come across  Arrays – A data structure. And algorithms are just sequences of processing steps to solve a problem. :)

# Time Complexity and Big O Notation

**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 | 122 ms |
| 70 elements | 110 ms | 124 ms |
| 110 elements | 180 ms | 131 ms |
| 1000 elements | 2s | 800 ms |

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?

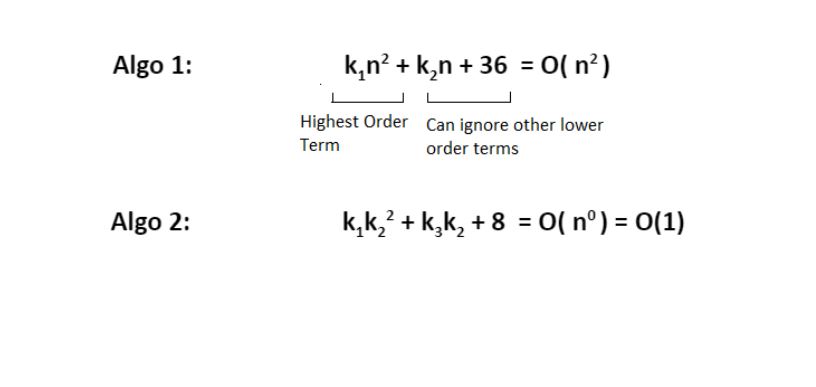
**Time Complexity: Sending GTA 5 to a friend:**

* 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(n0) 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:



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.

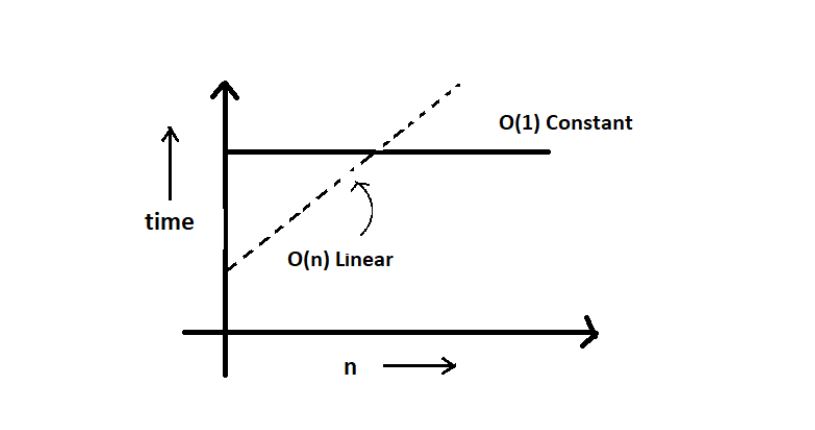
Note that these are the formulas for the time taken by their program.

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:



So, this was the basics of time complexities.

**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

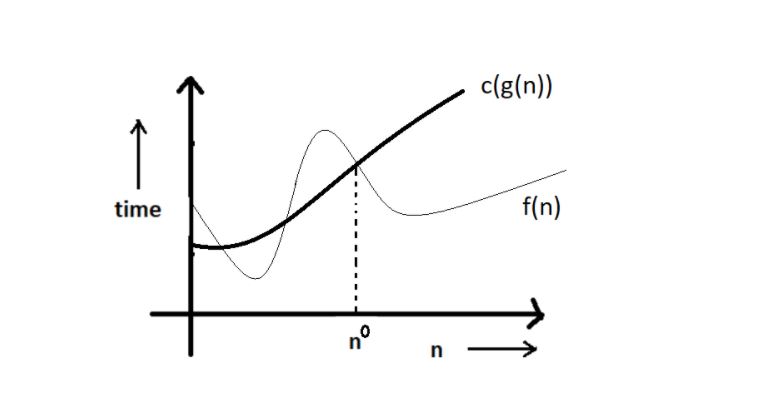
**Big oh notation ( O ):**

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

0 ≤ f(n) ≤ c g(n) for all n ≥ n°.

* Here, n is the input size, and g(n) is any complexity function, for, e.g. n, n2, etc. (It is used to give upper bound on a function)
* If a function is O(n), it is automatically O(n2) as well! Because it satisfies the equation given above.

**Graphic example for Big oh ( O ):**



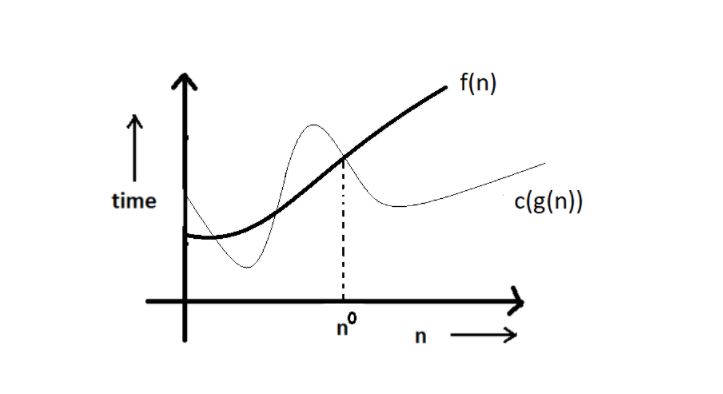
**Big Omega Notation ( Ω ):**

* 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 Ω (g(n)) if and only if there exist positive constants  c and n° such that:

0 ≤ c g(n) ≤ f(n) for all n ≥ n°.

* It is used to give the lower bound on a function.
* If a function is Ω (n2) it is automatically Ω (n) as well since it satisfies the above equation.

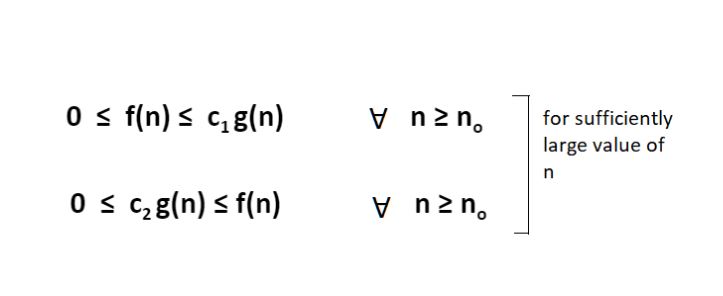
**Graphic example for Big Omega (Ω):**



**Big theta notation ( θ ):**

* Let f(n) define the running time of an algorithm.
* F(n) is said to be θ (g(n)) if f(n) is O (g(n)) and f(x) is Ω (g(n)) both.

Mathematically,

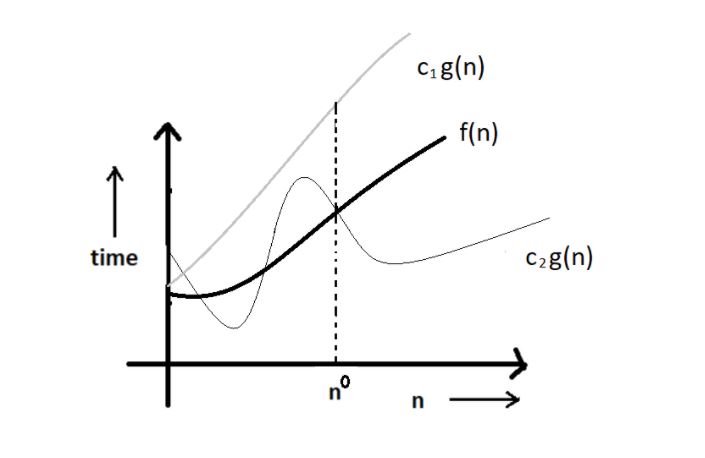


Merging both the equations, we get:

0 ≤ c2 g(n) ≤ f(n) ≤ c1 g(n) ∀ n ≥ no.

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

**Graphic example of Big theta ( θ ):**



**Which one of these to use?**

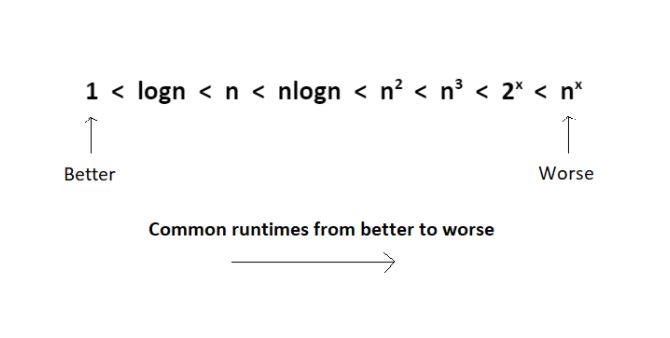
Big theta provides a better picture of a given algorithm's run time, which is why most interviewers expect you to answer in terms of Big theta when they ask "order of" questions. And what you provide as the answer in Big theta, is already a Big oh and a Big omega. It is recommended for this reason.

**Quick Quiz:** Prove that n2+n+1 is O(n3), Ω(n2), and θ(n2) using respective definitions.

**Hint:**You can approach this both graphically, making some rough graphs and mathematically, finding valid constants c1 and c2.

**Increasing order of common runtimes:**

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



So, this was all about the asymptotic notations.

**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)**

* We might get lucky enough to find our element to be the first element of the array. Therefore, we only made one comparison which is obviously constant for any size of the array.
* Best case complexity = O(1)
* If we are not that fortunate, the element we are searching for might be the last one. Therefore, our program made ‘n’ comparisons.
* Worst-case complexity = O(n)

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

**Analyzing Algorithm 2: (Binary Search)**

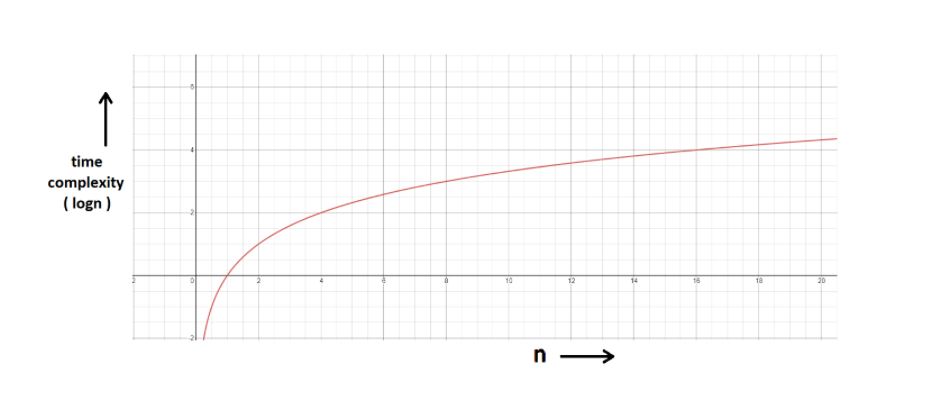
* If we get really lucky, the first element will be the only element that gets compared. Hence, a constant time.
* Best case complexity = O(1)
* If we get unlucky, we will have to keep dividing the array into halves until we get a single element. (that is, the array gets finished)
* Hence the time taken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

**What is log(n)?**

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

* log8 = 3  ⇒  8/2  + 4/2  + 2/2   →    Can’t break anymore.
* log4 = 2  ⇒  4/2  + 2/2   →    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) **→** 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) = n2 → O(n^2)

T(n) = logn → O(logn)

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

##### **Consider all variables which are provided as input:**

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

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

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

#### 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;

}

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);

}

}

}

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);

}

}

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

1. O(N + P), where P < N/9
2. 0(9N-k)
3. O(N + 8log N)
4. O(N + M2)

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);

}

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;

}

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();

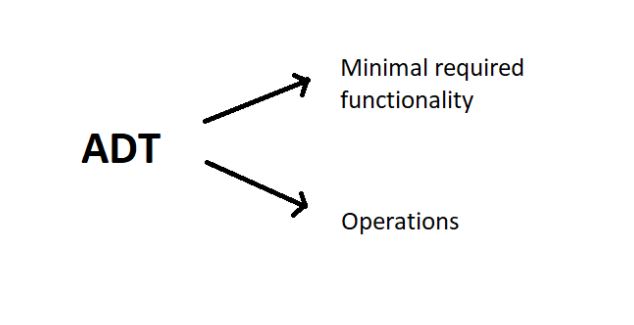
So, these were the few questions I felt like discussing. Try them on your own first. And if you have already given them enough thought, then move on to the video above where we have discussed their solutions in detail.

# 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:https://cwh-full-next-space.fra1.digitaloceanspaces.com/videos/data-structures-and-algorithms-in-hindi-6/Image_2.webp

* 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.

So, this was enough discussing the basics of an array ADT. It is now time to implement these data types through our codes. Even if you are not very familiar with these languages or need a quick brush-up, you can always move on to my easily accessible C or C++ playlist.

**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,

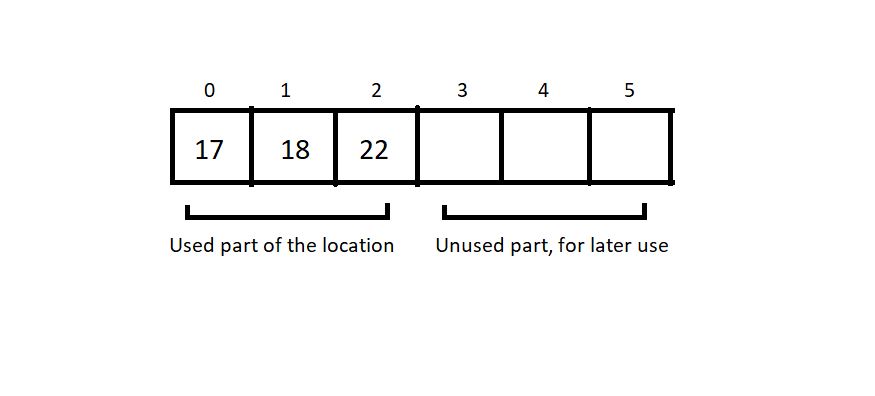
* max()
* get(i)
* set(i,num)
* add(another\_array)

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.

We will keep the code implementation of the above ADT for the next tutorial. You can even give it a try on your editors. Use structs and define that set of values with proper data types. The next session will teach us that anyway.

# 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;

***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));

***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;

}

***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>

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](https://www.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?

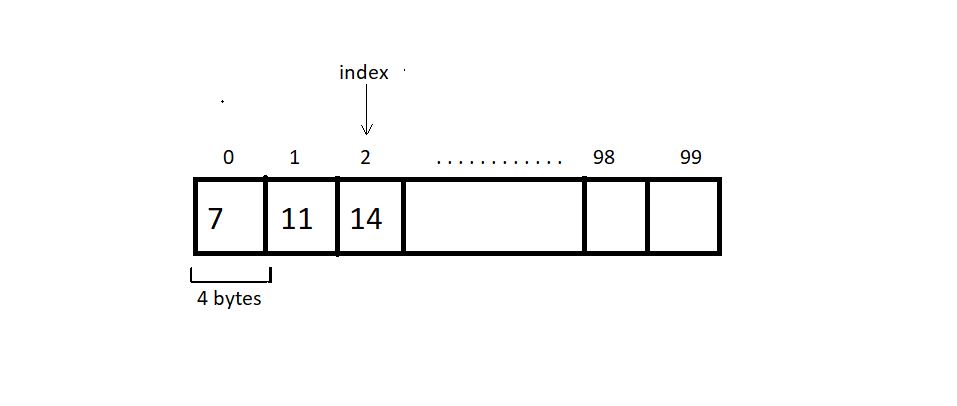
For use cases like:

* 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.

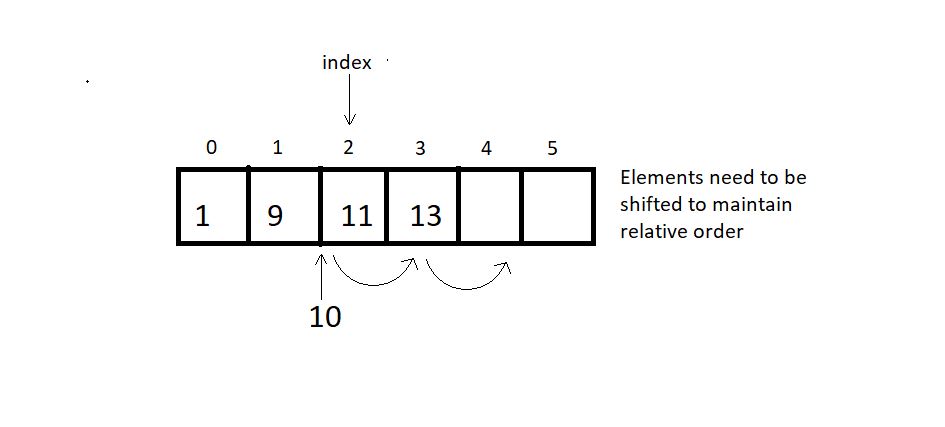
##### An important note on Arrays:

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.

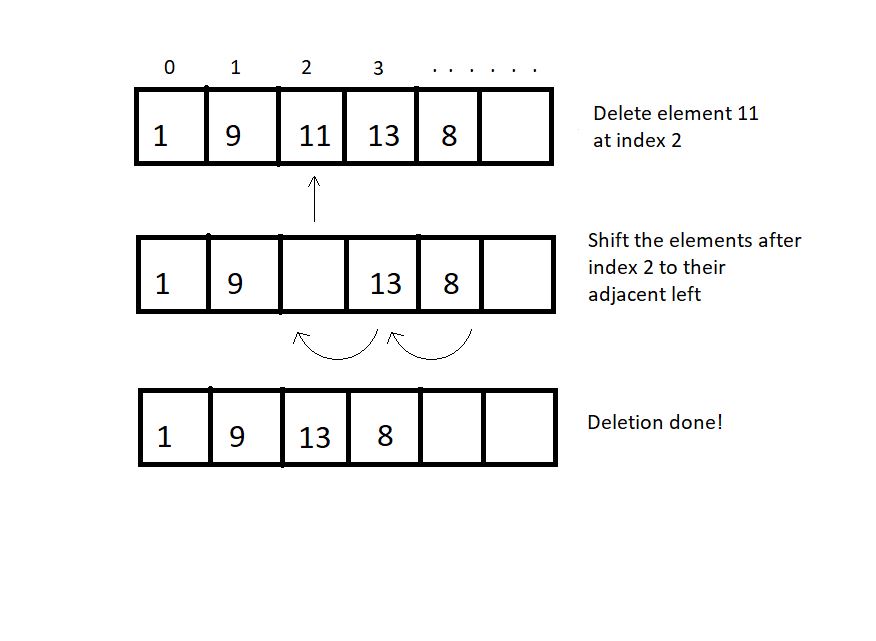


When no position is specified, it’s best to insert the element at the end to avoid shifting, and this is when we achieve the best runtime O(1).

#### 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.

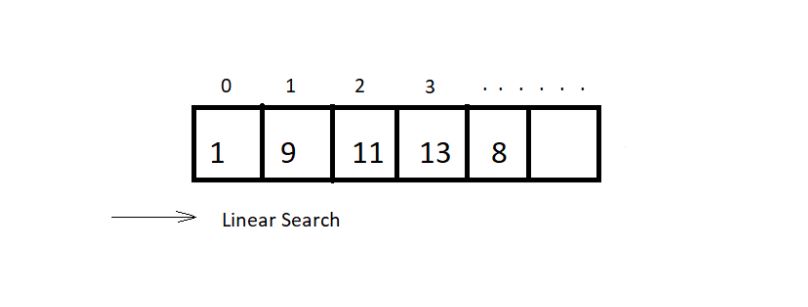
We can also bring the last element of the array to fill the void if the relative ordering is not important. :)



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

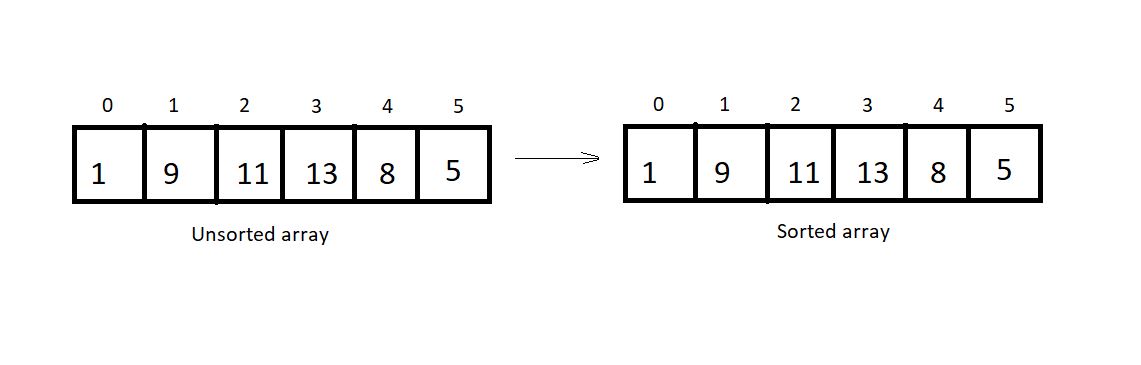
#### Searching:

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(logn)



#### 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.



So, these were few primary operators for an abstract data type array

**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 *arr*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);

1. 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.
2. 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.
3. Insert the element in the index. Return 1 on completion.
4. #include<stdio.h>

7. void display(int arr[], int n){
8. // Code for Traversal
9. for (int i = 0; i < n; i++)
10. {
11. printf("%d ", arr[i]);
12. }
13. printf("\n");
14. }
16. int indInsertion(int arr[], int size, int element, int capacity, int index){
17. // code for Insertion
18. if(size>=capacity){
19. return -1;
20. }
21. for (int i = size-1; i >=index; i--)
22. {
23. arr[i+1] = arr[i];
24. }
25. arr[index] = element;
26. return 1;
27. }
29. int main(){
30. int arr[100] = {7, 8, 12, 27, 88};
31. int size = 5, element = 45, index=1;
32. display(arr, size);
33. indInsertion(arr, size, element, 100, index);
34. size +=1;
35. display(arr, size);
36. return 0;
37. }

***Code Snippet 1: Insertion Operation Algorithm***

Output of the above program:

7 8 12 27 88

7 45 8 12 27 88

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.

**Coding Deletion Operation in Array Using C Language (With Notes)**

In the last tutorial, we had learned about the first two primary operations in an array ADT, traversal and insertion. Today, we will study the third one, *deletion*.

Programming a deletion differs very slightly from programming an insertion. In insertion, we had to shift elements to their adjacent right to create a void at the desired place to insert a new element, but in deletion, we’ll shift the elements to their adjacent left to fill the void created after deleting an element at some index.

Let us now code this out or rather transform the code we constructed to insert the element. I have attached the snippet below.

**Understanding code snippet 1:**

1. One thing which will remain as it is, is the display function.
2. We have to make minimal changes in the insertion function to make it a deletion function. Rename it *indDeletion.*The index and the array, and its size will be our only parameters this time.
3. Replace the right shift with the left shift. Just assign array[i], the value present in array[i+1].
4. And we are done deleting the element at some specified index.

#include <stdio.h>

void display(int arr[], int n)

{

// Code for Traversal

for (int i = 0; i < n; i++)

{

printf("%d ", arr[i]);

}

printf("\n");

}

void indDeletion(int arr[], int size, int index)

{

// code for Deletion

for (int i = index; i < size-1; i++)

{

arr[i] = arr[i + 1];

}

}

int main()

{

int arr[100] = {7, 8, 12, 27, 88};

int size = 5, element = 45, index = 0;

display(arr, size);

indDeletion(arr, size, index);

size -= 1;

display(arr, size);

return 0;

}

**Code Snippet 1: Deletion in an array:**

We can now check if the program actually works for deleting the element at some index. We’ll create an array with 5 elements and display it before and after deleting an element at index 0.

Refer to the output below:

7 8 12 27 88

8 12 27 88

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

**Figure 1: Output of the above program**

So, the code works fine. Element at index 0 got deleted, and the rest of the elements shifted left to fill the void created after deletion.  And this was all about the deletion. Only a subtle change of code helped transform insertion into deletion.

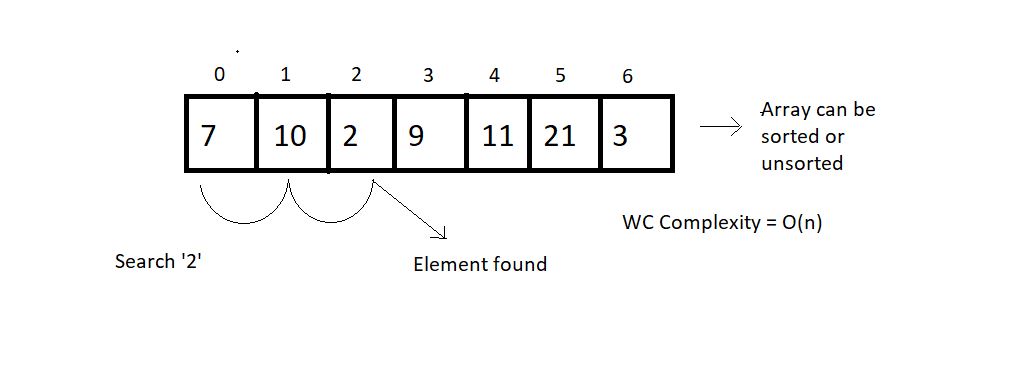
# Linear Vs Binary Search + Code in C Language (With Notes)

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.

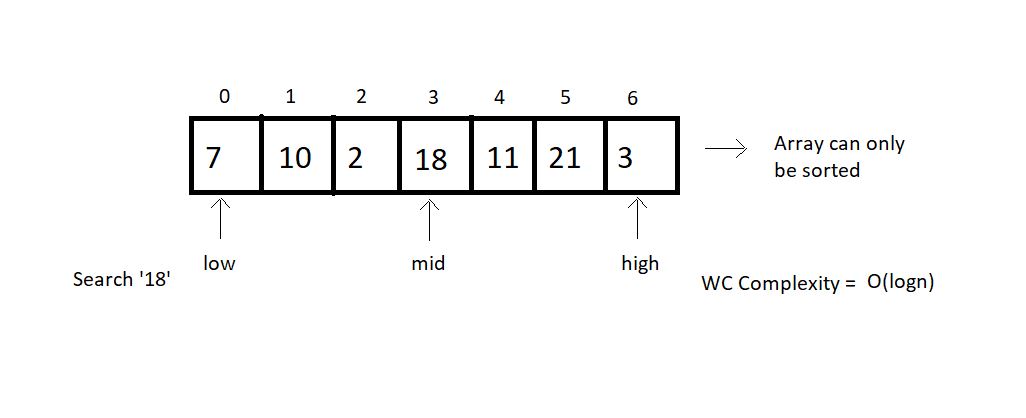
#### Linear Search:

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 (low+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;

}

**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

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

So from the above output, you can conclude that our code works all fine. So, we are done implementing both these search methods. Binary Search holds great importance in the world of programming. We’ll come across several algorithms following binary search.

# 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.

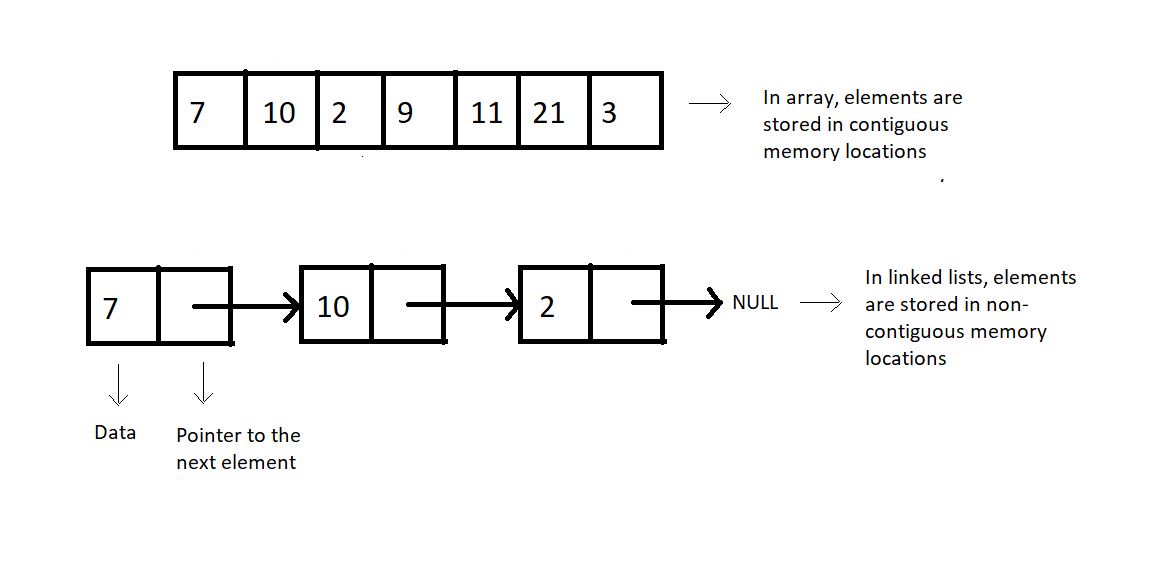
But linked lists are stored in a non-contiguous memory location. To add a new element, we just have to create a node somewhere in the memory and get it pointed by the previous element. And deleting an element is just as easy as that. We just have to skip pointing to that particular node. Lengthening 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

};

**Code Snippet 1: Implementation of a linked list**

**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:

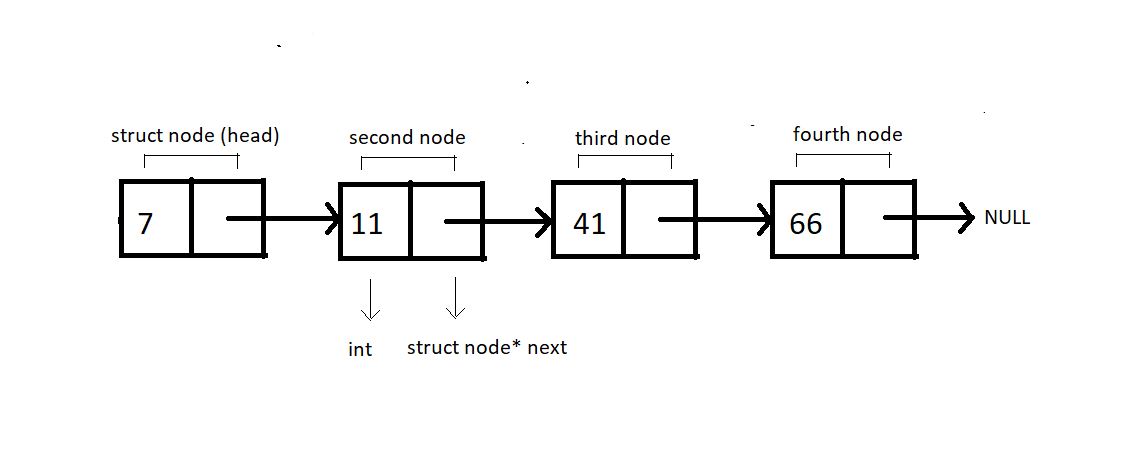
1. These are stored in non-contiguous memory locations.
2. Insertion and deletion in a linked list are very efficient in comparison to arrays.
3. An element called node holds the value as well as a pointer to the next element.

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 ‘→’.

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));

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;

}

**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

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

**Figure 2: Output of the above program**

So, this was successfully creating and traversing through the linked list.

**Insertion of a Node in a Linked List Data Structure**

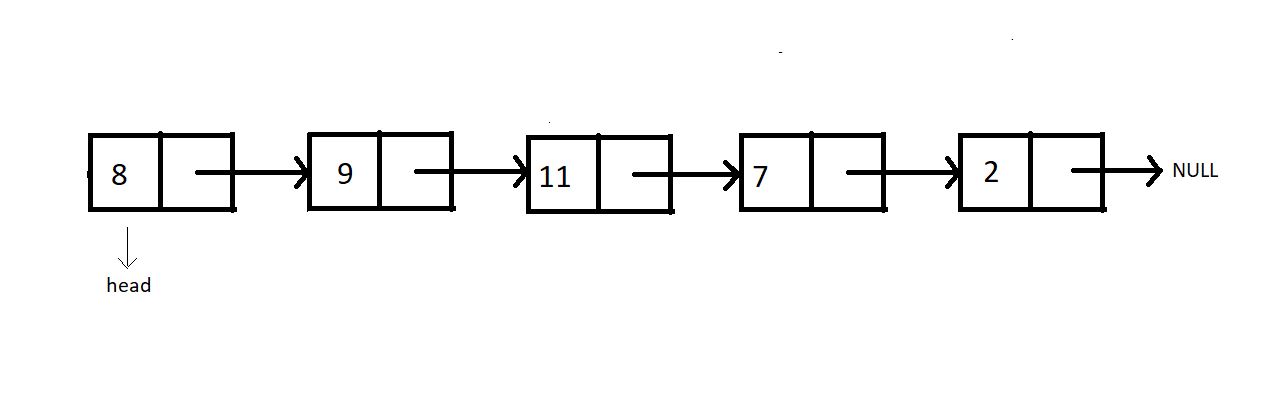
In the last tutorial, we had learned about creating a linked list using C structures and traversing through them while printing the values at each node. Today, we’ll learn how to insert a node at some position and how that is more efficient than inserting an element in an array.

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))

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

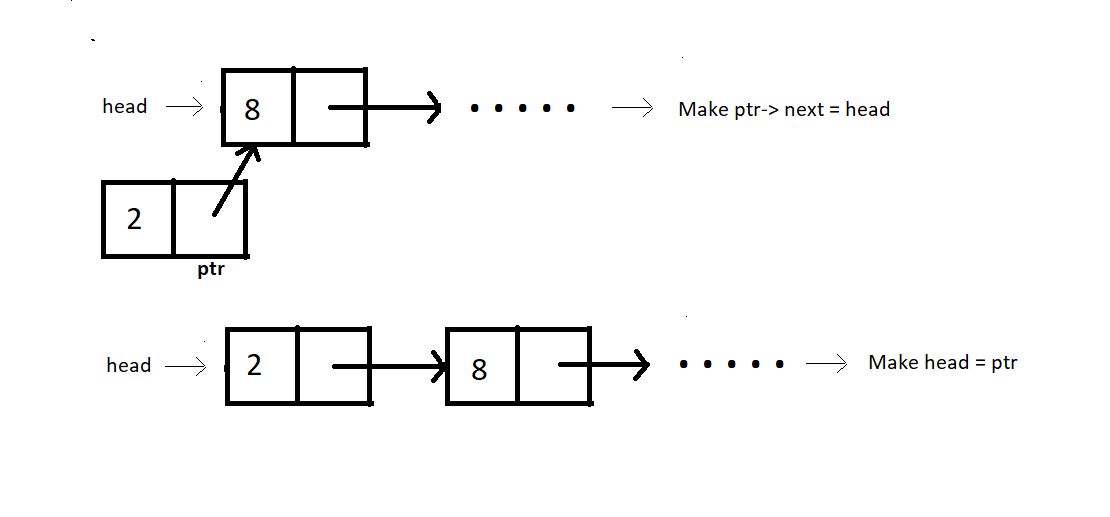
ptr -> data = 9

This will set the data.

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

**Case 1: Insert at the beginning**

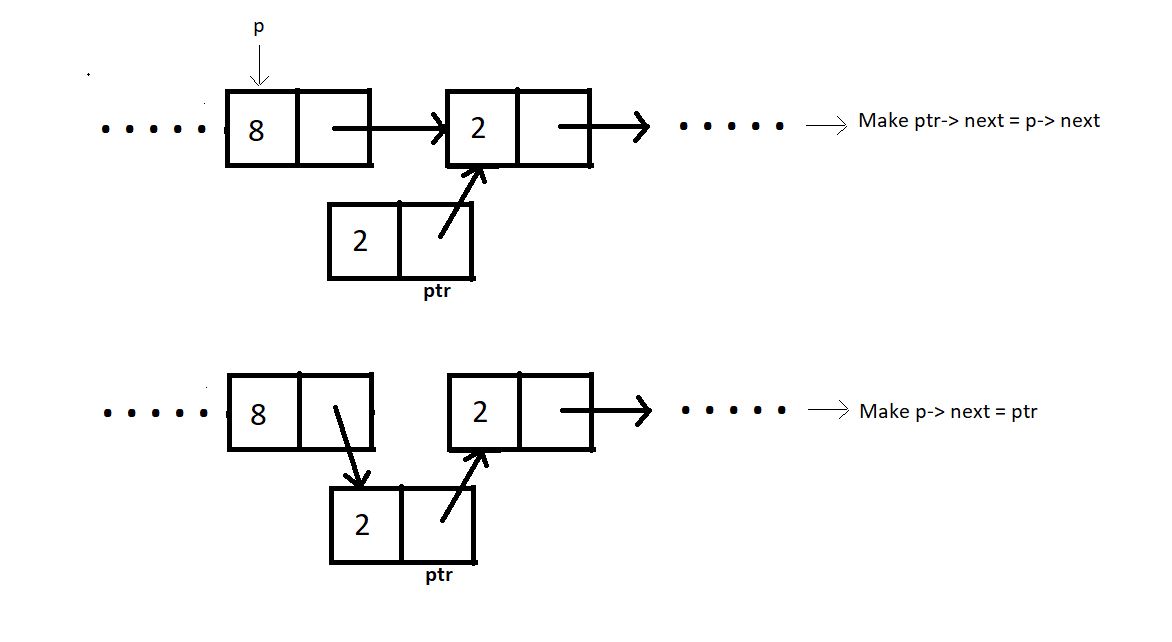
In order to insert the new node at the beginning, we would need to have the head pointer pointing to this new node and the new node’s pointer to the current head.



**Case 2: Insert in between:**

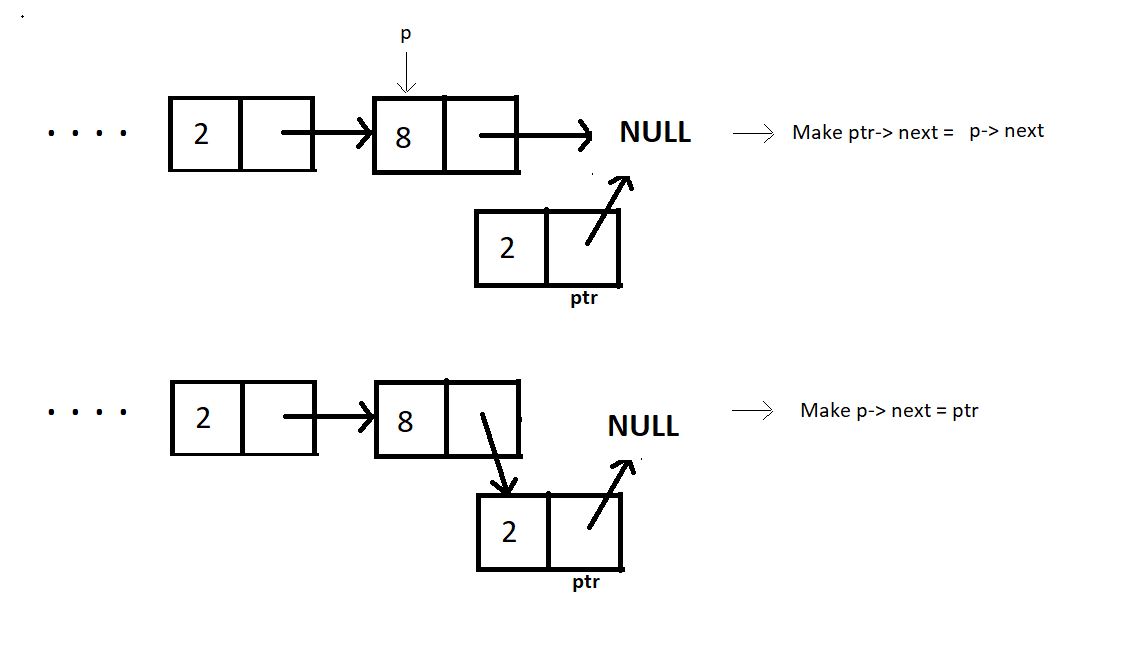
Assuming index starts from 0, we can insert an element at index i>0 as follows:

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.

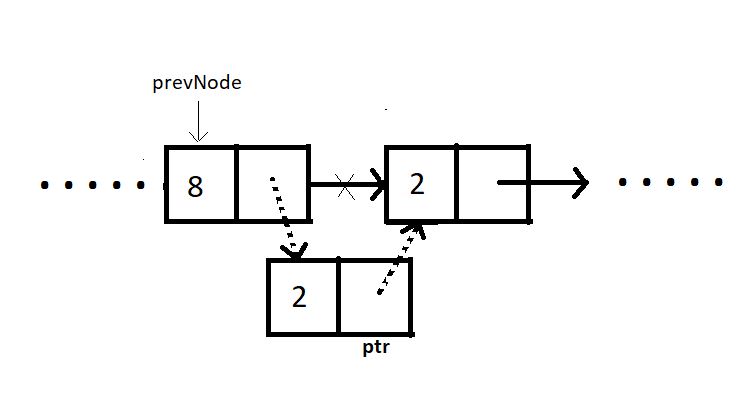


**Case 4: Insert after a node:**

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 Node));
9. ptr->data = data;
11. ptr->next = head;
12. return ptr;
13. }

***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;

}

***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;

}

***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;
14. ptr->next = prevNode->next;
15. prevNode->next = ptr;
17. return head;
18. }

**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);

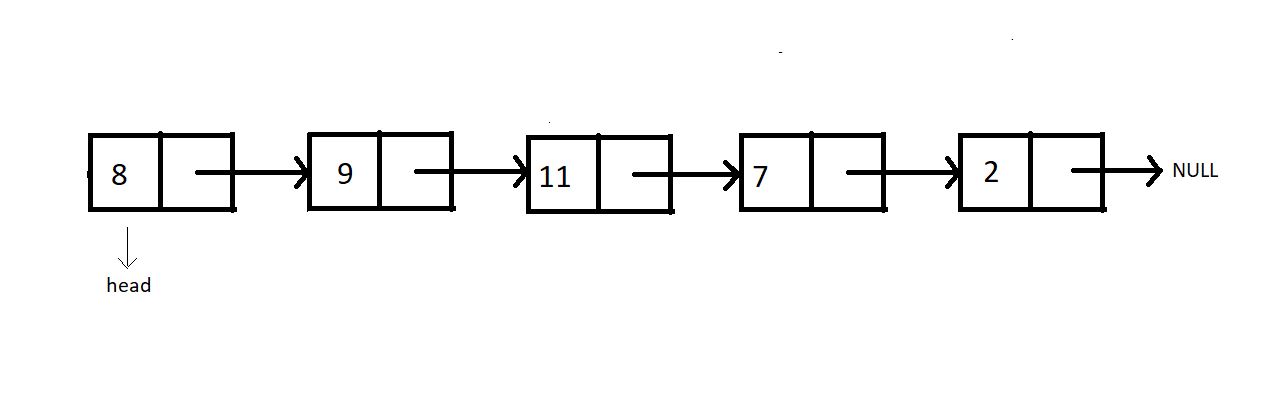
return 0;

}

# 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:

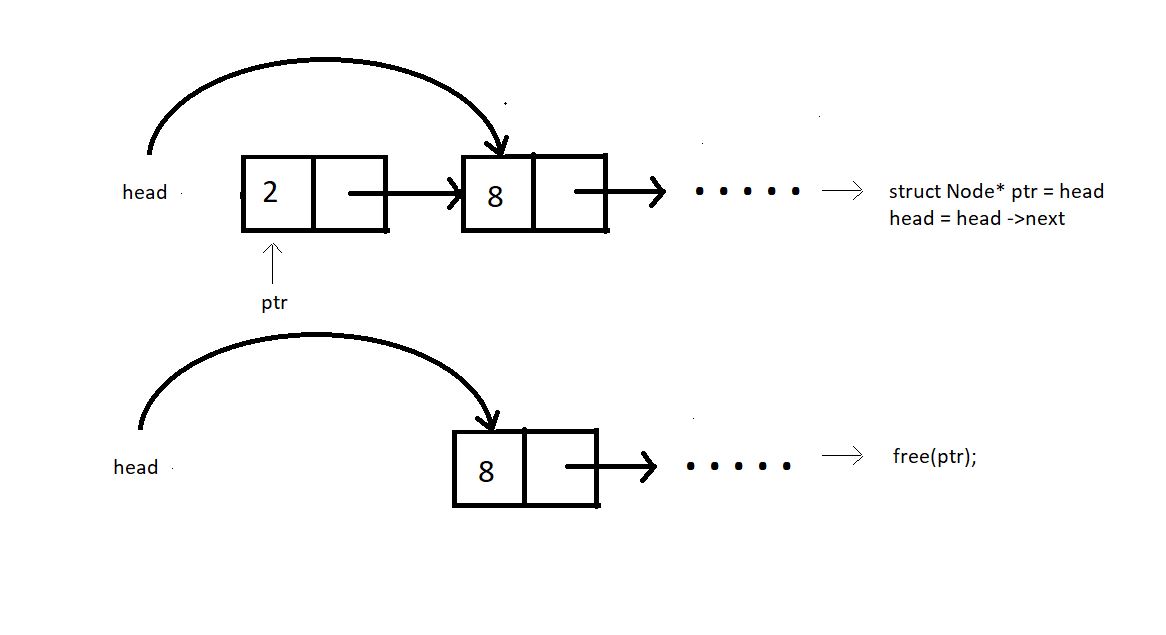
free(ptr);

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:

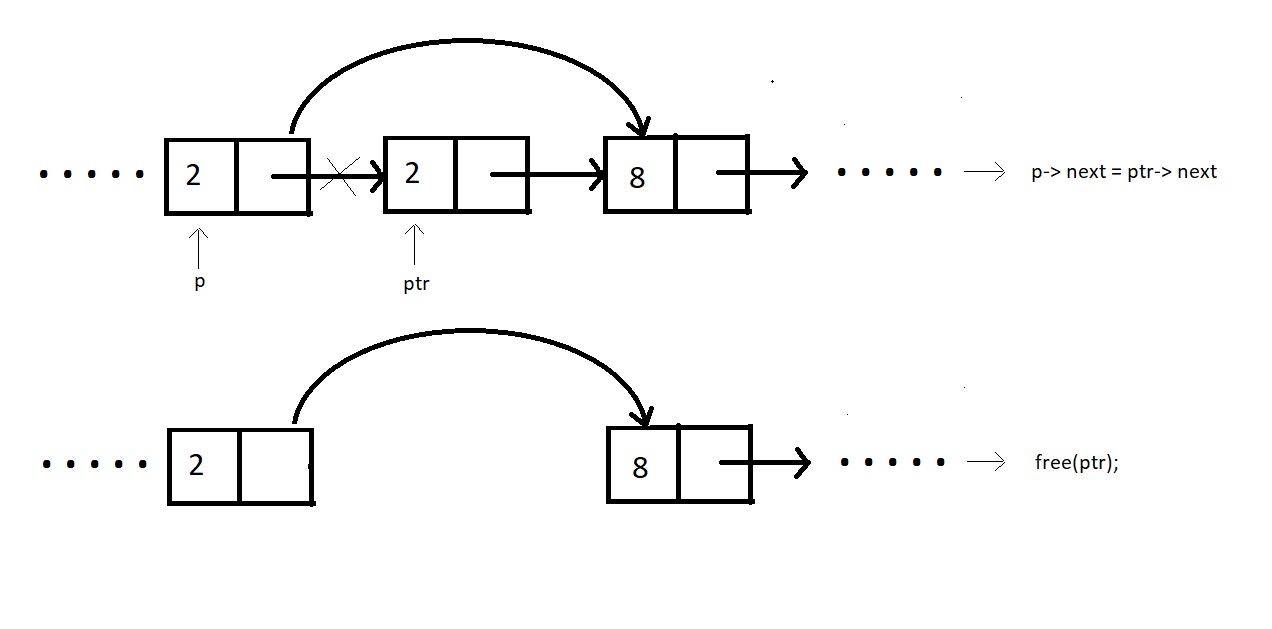
In order to delete the node at the beginning, we would need to have the head pointer pointing to the node second to the head node, that is, head-> next. And we would simply free the node that’s left.



#### 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.

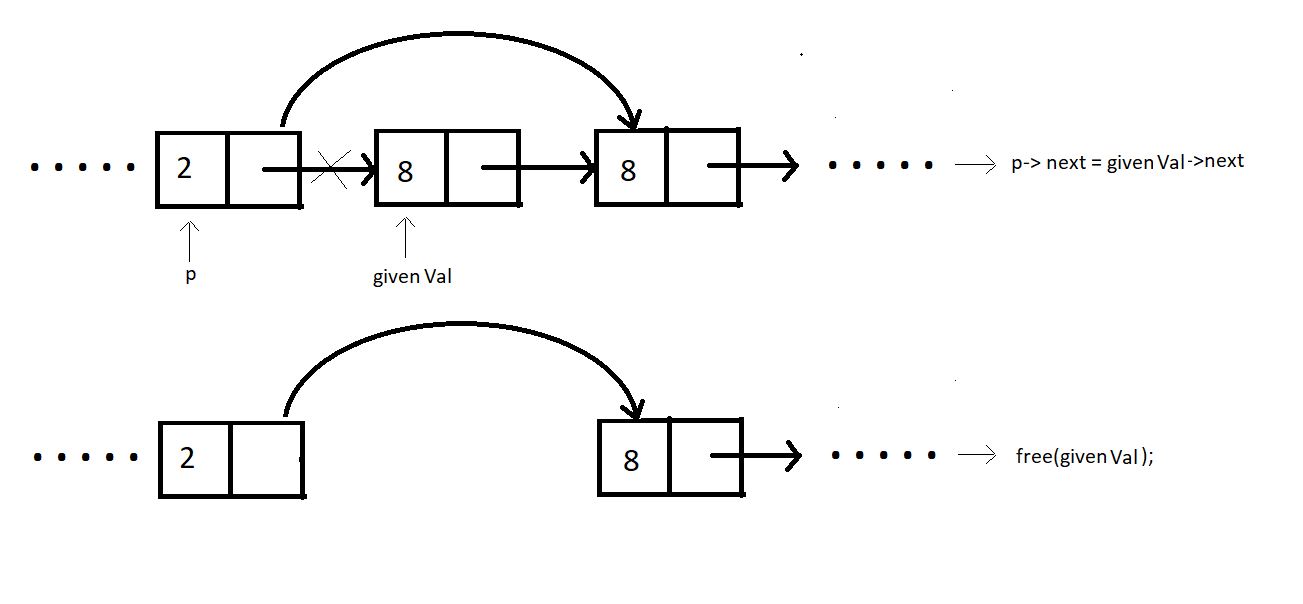


#### Case 4: Delete the first node with a given value:

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. }

***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. }
18. p->next = q->next;
19. free(q);
20. return head;
21. }

**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. }
15. p->next = NULL;
16. free(q);
17. return head;
18. }

***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. }
20. if(q->data == value){
21. p->next = q->next;
22. free(q);
23. }
24. return head;
25. }

***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 = 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 incept 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(){
10. struct Node \*head;
11. struct Node \*second;
12. struct Node \*third;
13. struct Node \*fourth;
15. // Allocate memory for nodes in the linked list in Heap
16. head = (struct Node \*)malloc(sizeof(struct Node));
17. second = (struct Node \*)malloc(sizeof(struct Node));
18. third = (struct Node \*)malloc(sizeof(struct Node));
19. fourth = (struct Node \*)malloc(sizeof(struct Node));
21. // Link first and second nodes
22. head->data = 4;
23. head->next = second;
25. // Link second and third nodes
26. second->data = 3;
27. second->next = third;
29. // Link third and fourth nodes
30. third->data = 6;
31. third->next = fourth;
33. // Terminate the list at the third node
34. fourth->data = 1;
35. fourth->next = head;
37. return 0;
38. }

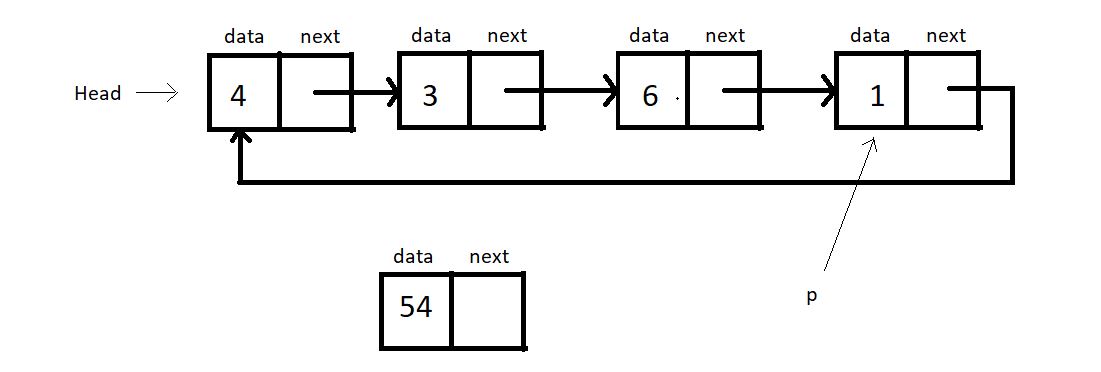
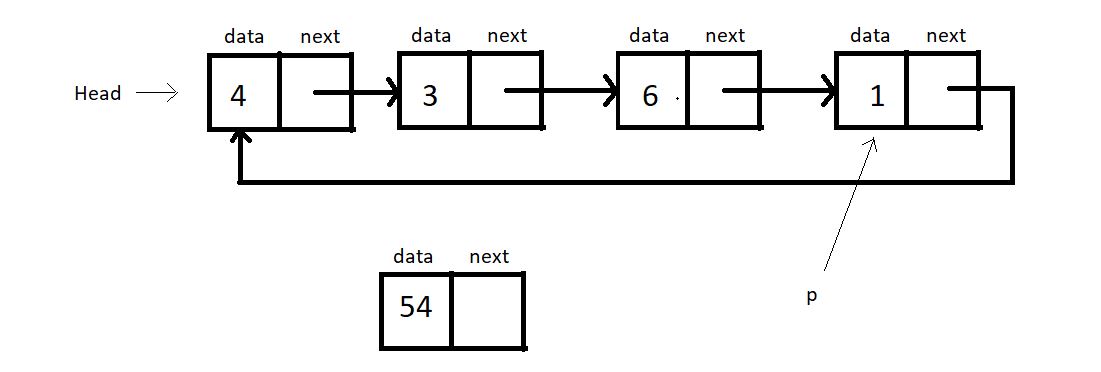
**Code Snippet 1: Creating the circular linked list**

#### 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. }

**Code Snippet 2: Traversing the circular linked list**

#### Inserting into a circular linked list:

1. 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.
2. Create a struct Node\* function insertAtFirst which will return the pointer to the new head.
3. We’ll pass the current head pointer and the data to insert at the beginning, in the function.
4. 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>.
5. Create another struct node \* pointer p pointing to the next of the head. p = head-> next.
6. Run a while loop until the p pointer reaches the end element and p-> next becomes the head.  
   
7. 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 \*) malloc(sizeof(struct Node));
12. 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 circular linked list
20. p->next = ptr;
21. ptr->next = head;
22. head = ptr;
23. return head;
25. }

**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;

}

***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);

***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

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.

# Doubly Linked Lists Explained With Code in C Language

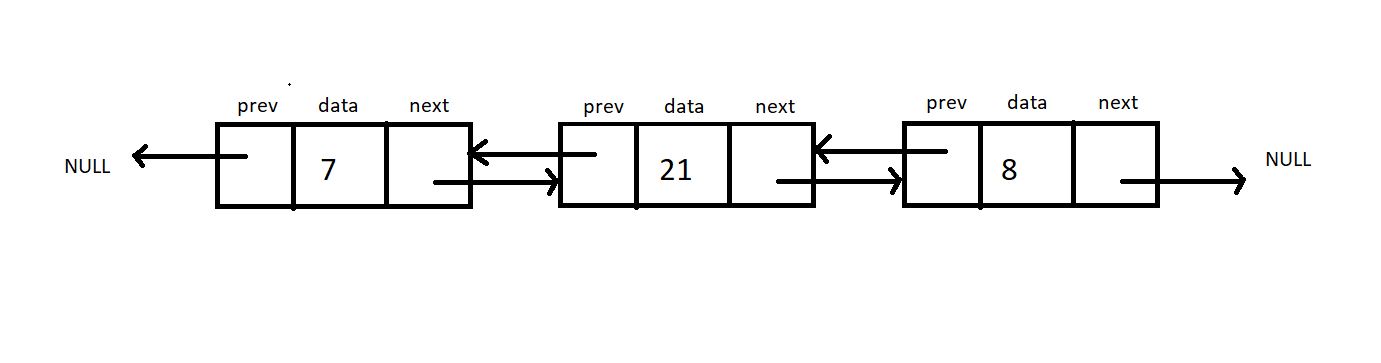
So, we have already talked about a lot of things under linked lists. We talked about the singly-linked lists, which had both a head node and the last node pointing to the NULL. We also talked about circular linked lists, which had no ending but an arbitrary head node. We also learned about all the basic 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;

};

***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.” :)

#### Code as described:

#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;

}

**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.

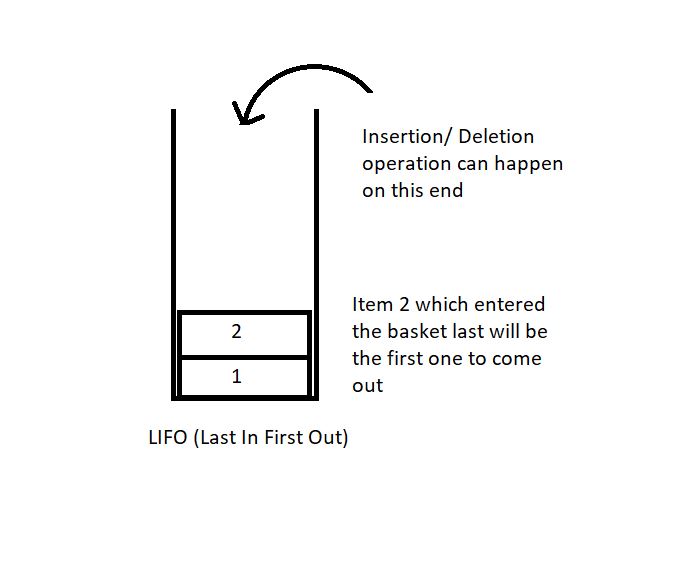
# Introduction to Stack in Data Structures

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.



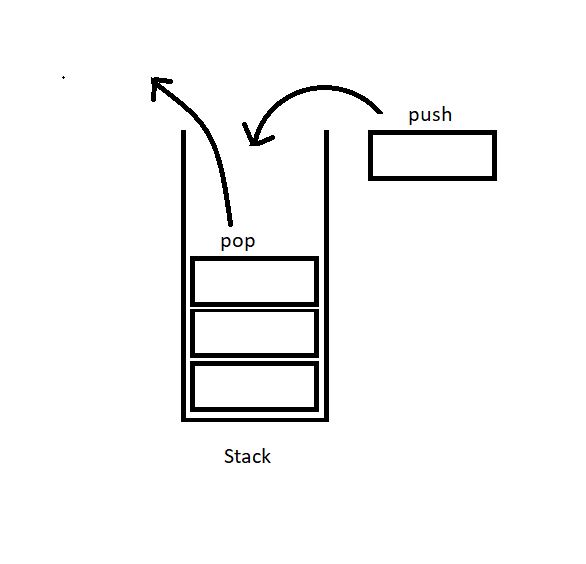
Applications of Stack:

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 about. 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.

#### Implementation:

A stack element can be implemented by both an array and a linked list. We’ll see both these methods in the coming tutorials.

A stack is a collection of elements with certain operations following the LIFO (Last in First Out) discipline.

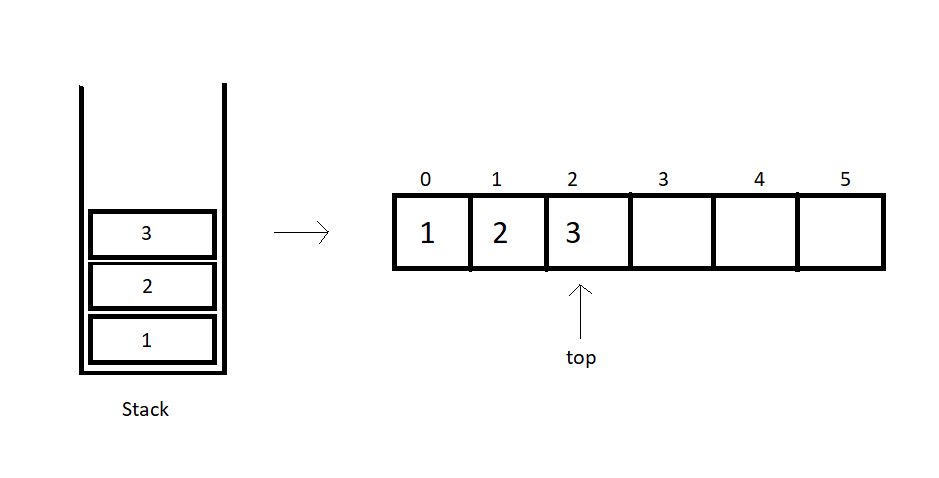
# Implementing Stack Using Array in Data Structures

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;

}

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));

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.

**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;

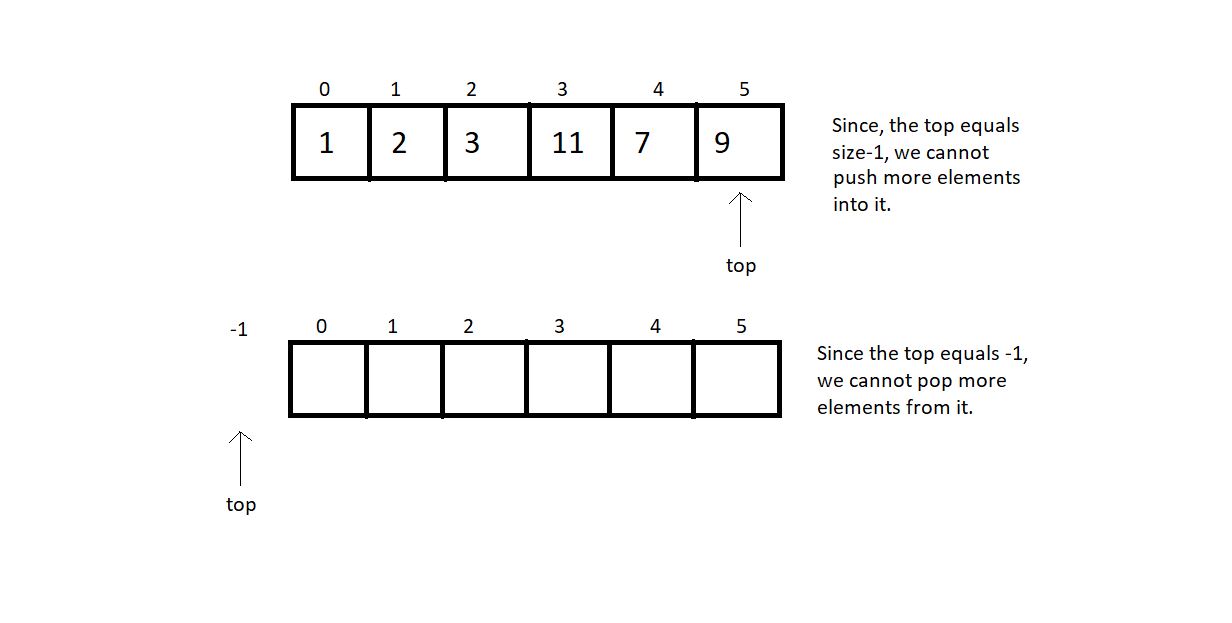
};

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;

}

}

***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;

}

}

***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;

}

***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");

}

***Code Snippet 5: Calling the function isEmpty***

Output:

The stack is empty

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

**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");

}

***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>

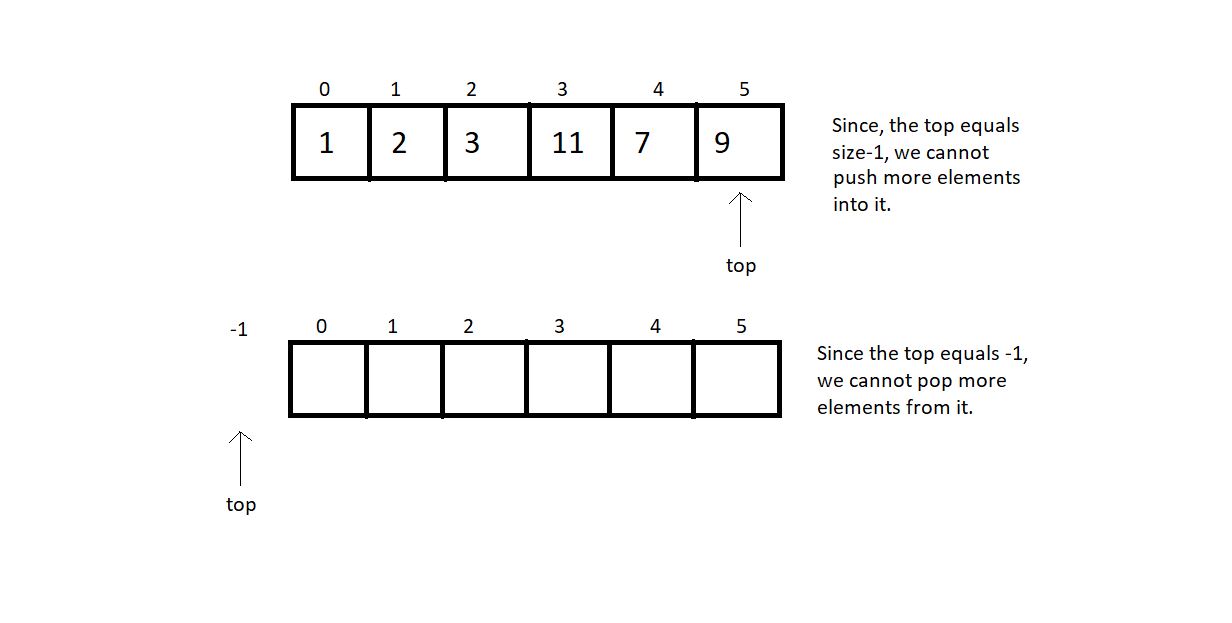
**Figure 2: Output of the above program**

# 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.

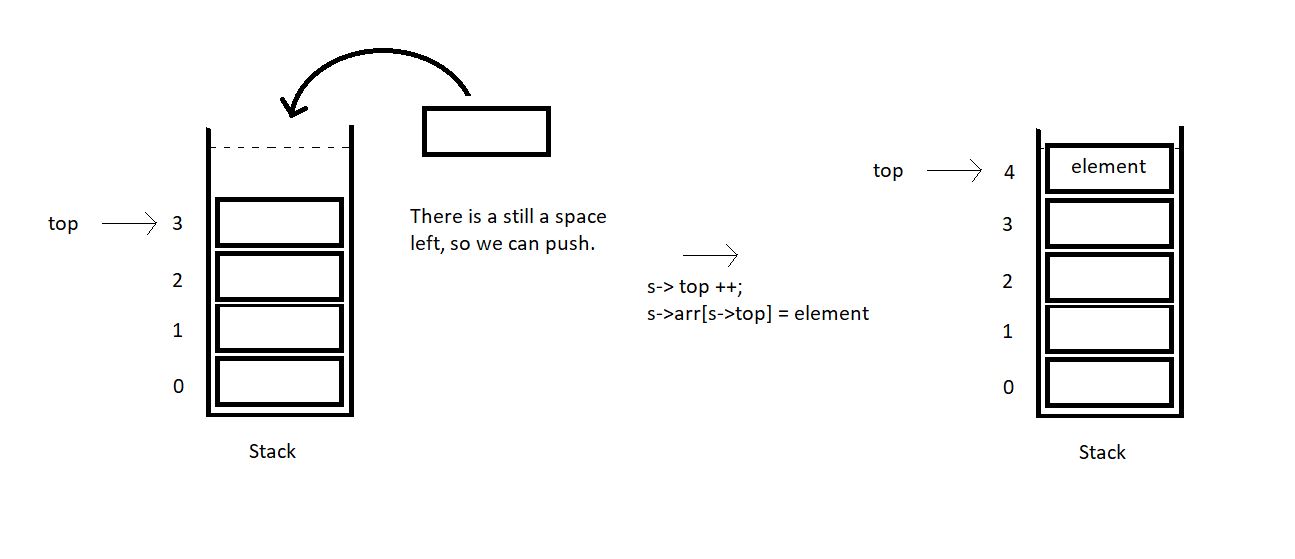


Declaring a stack was done in the last tutorial as well. Let's keep that in mind as we proceed.

**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.



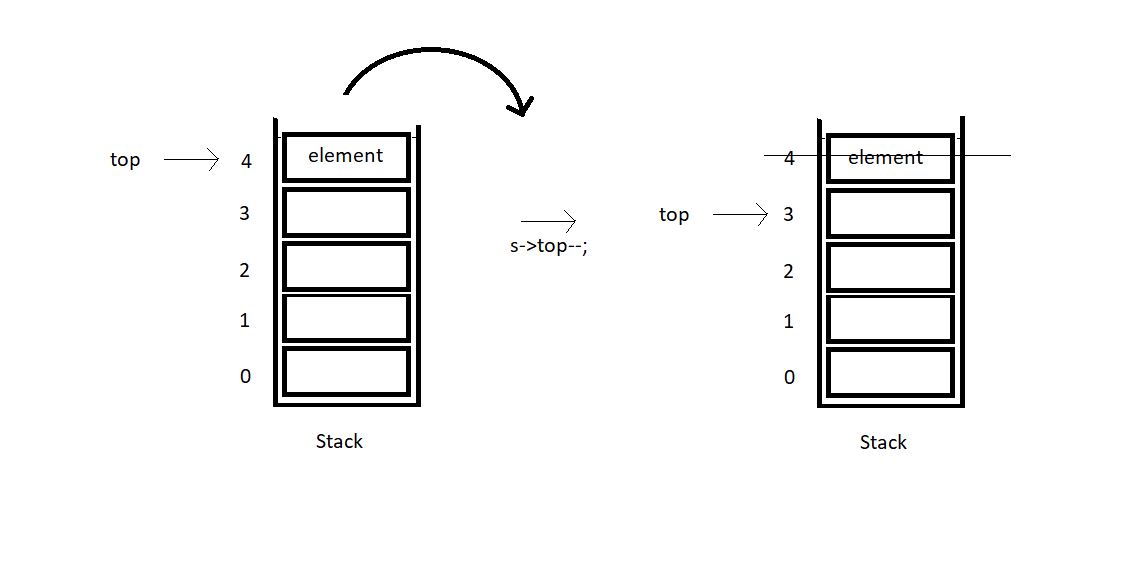
3. So, this is how we push an element in a stack array. Suppose we have an element x to insert in a stack array of size 4. We first checked if it was full, and found it was not full. We retrieved its top which was 3 here. We made it 4 by increasing it once. Now, just insert the element x at index 4,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

# 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;

};

**Code Snippet 1: Creating stack struct**

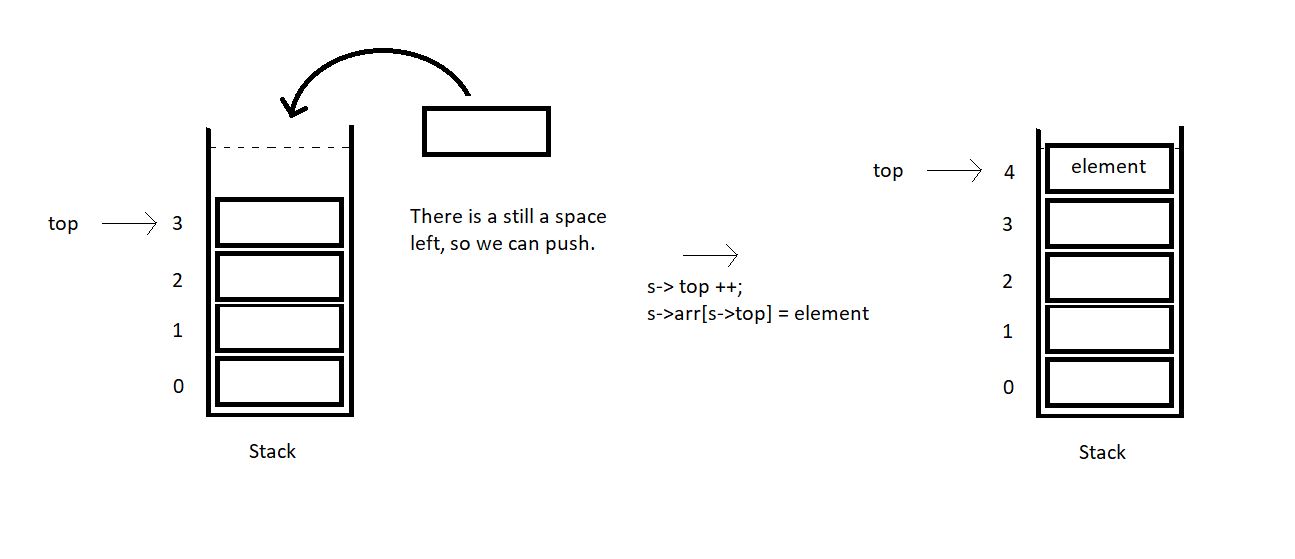
2. In the main, define a struct stack pointer sp, which will store the address of the stack. Since we are using malloc to reserve the memory in heap for this stack, don't forget to include the header file <stdlib.h>.

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](https://www.codewithharry.com/videos/data-structures-and-algorithms-in-hindi-24). 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;

}

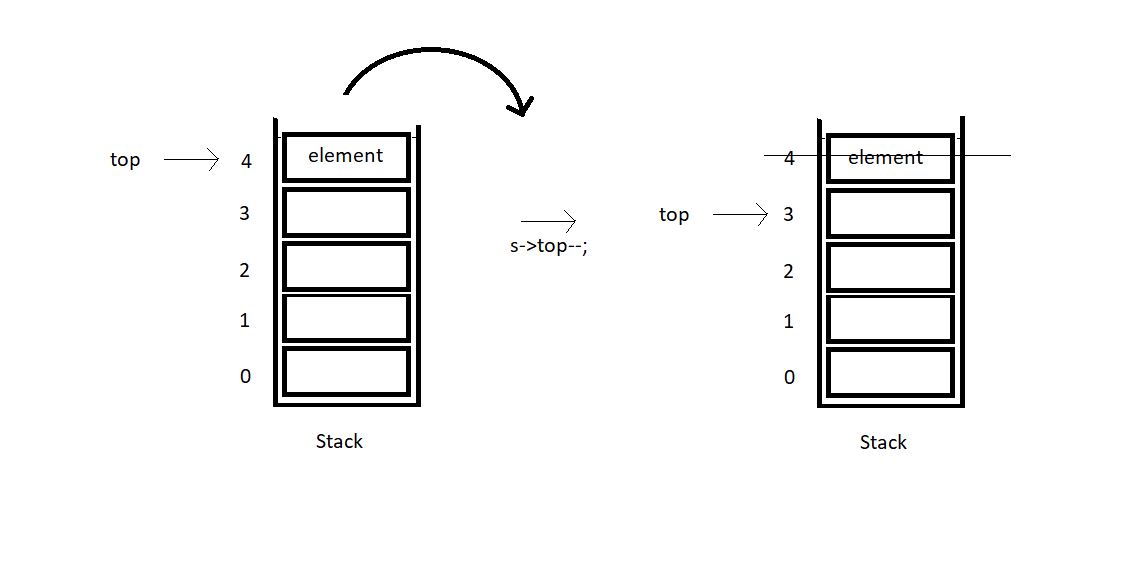
}

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;

}

}

**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;

}

**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));

**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>

**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));

**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>

**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!

**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>

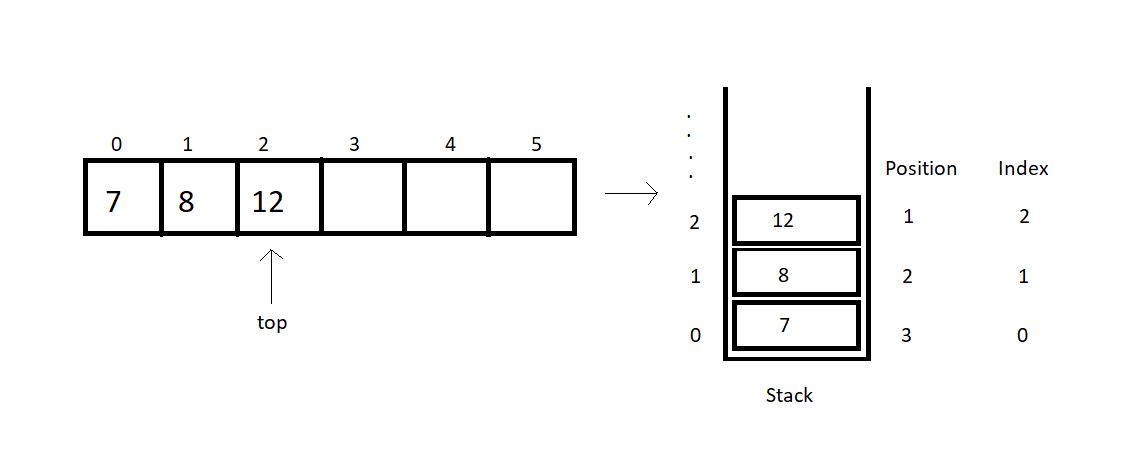
**Figure 3: Output of the above program**

**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];

}

}

**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;

}

**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);

**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));

}

**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>

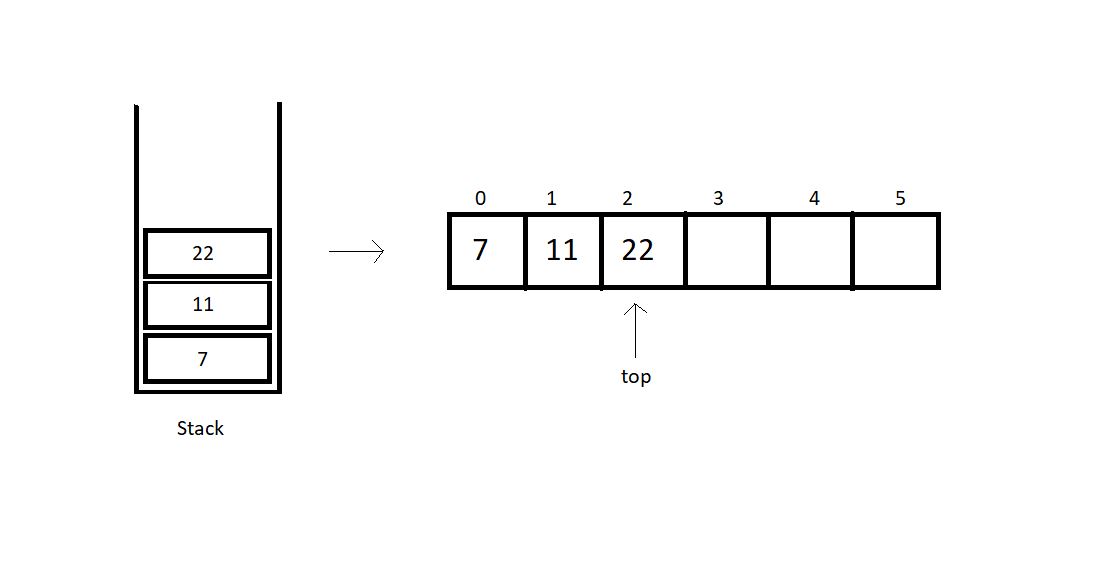
**Figure 1: Output of the above program**

# 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];

}

**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];

}

**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;

}

**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));

**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>

**Figure 1: Output of the above program**

Finally, I can say that it is all that we had when we implemented stacks using arrays.

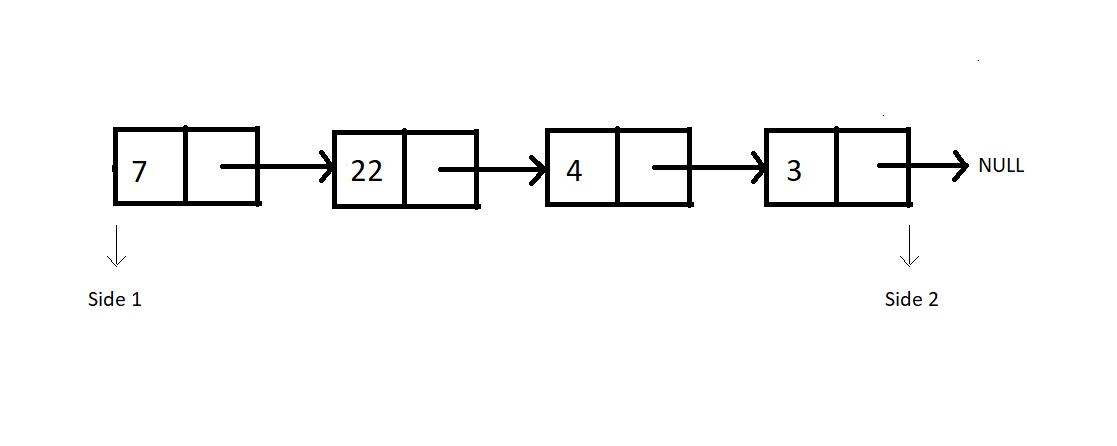
# How to Implement Stack Using Linked List?

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.

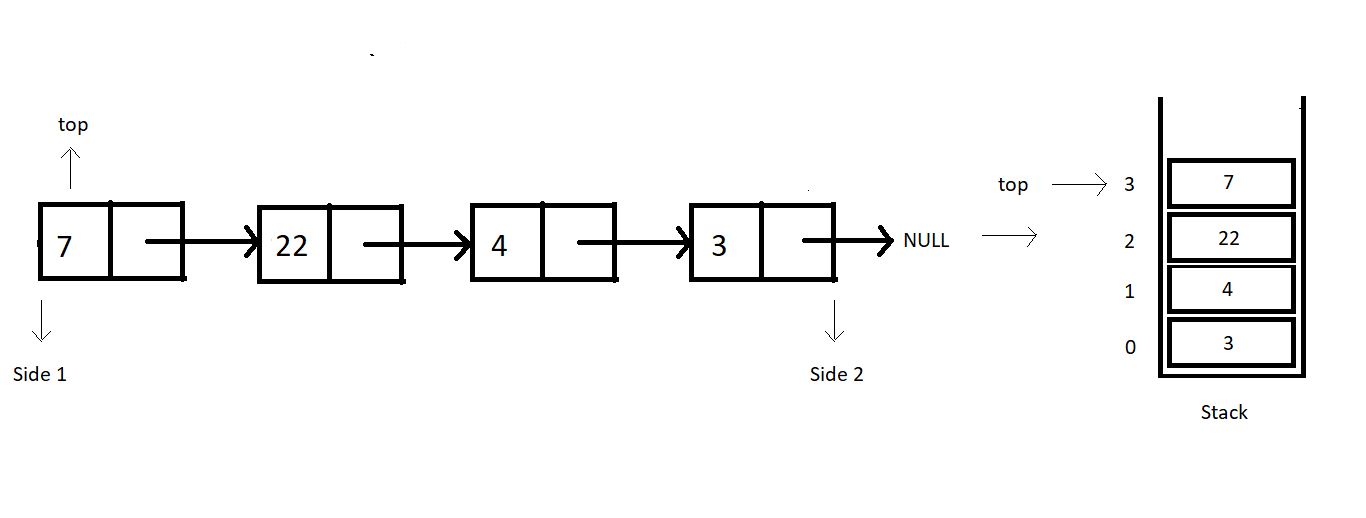


Consider this linked list functioning as a stack. And as you know, we have two sides of a linked list, one the head, and the other pointing to NULL. Which side do you feel should we consider as the top of the stack, where we push and pop from? After following me all the way through here, you would say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

}

**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.

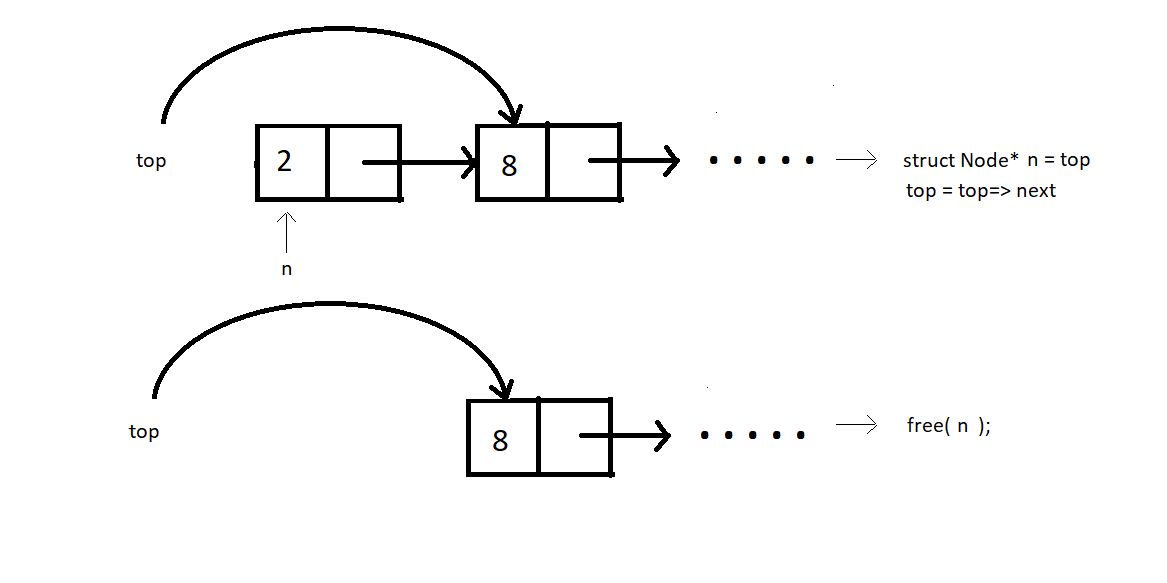
**1. isEmpty :**It just checks if our top element is NULL.

**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;

}

}

**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;

}

}

**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;

}

}

**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;

}

}

**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;

}

}

**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;

}

**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);

**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>

**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);

**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>

**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;

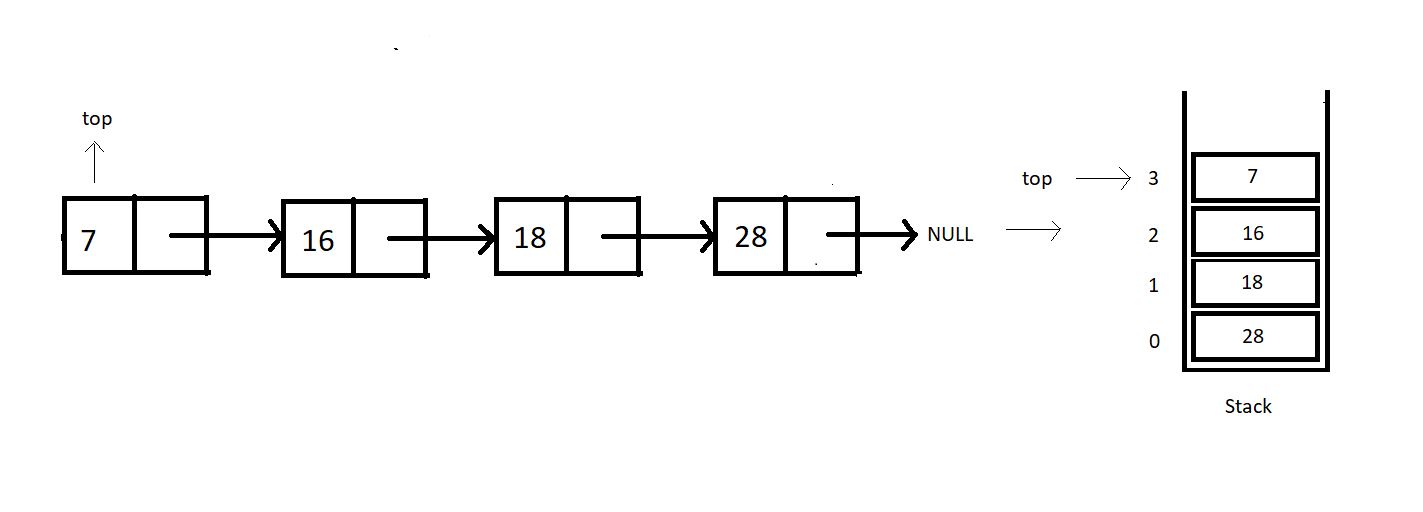
}

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

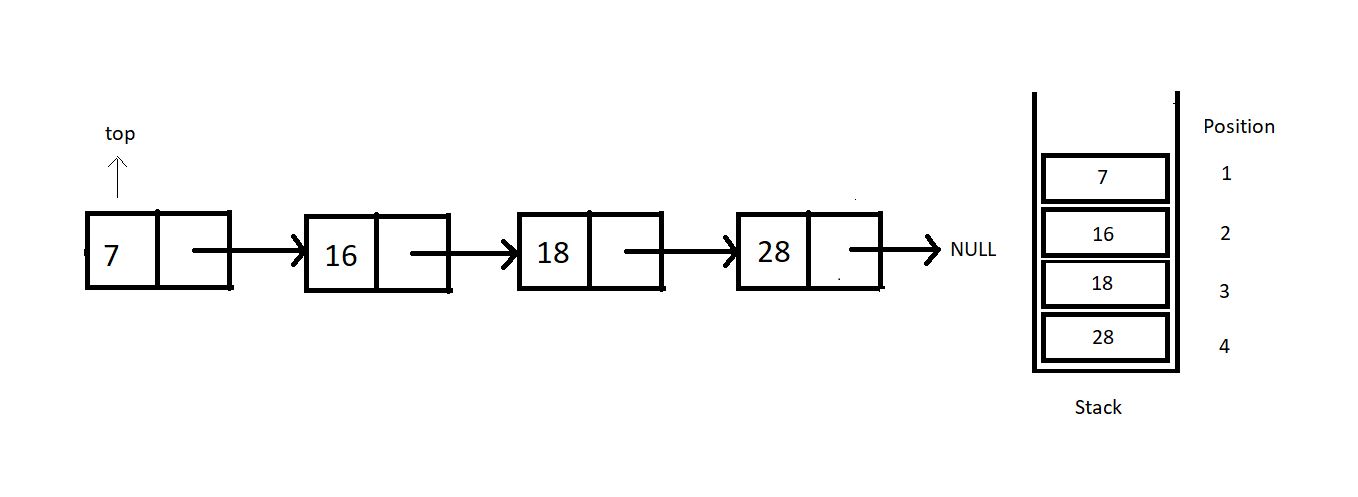
# peek(), stackTop() and Other Operations on Stack Using Linked List (with C Code)

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;

}

}

***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;

}

***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;

}

***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);

***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));

}

***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>

***Figure 1: Output of the above program***

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

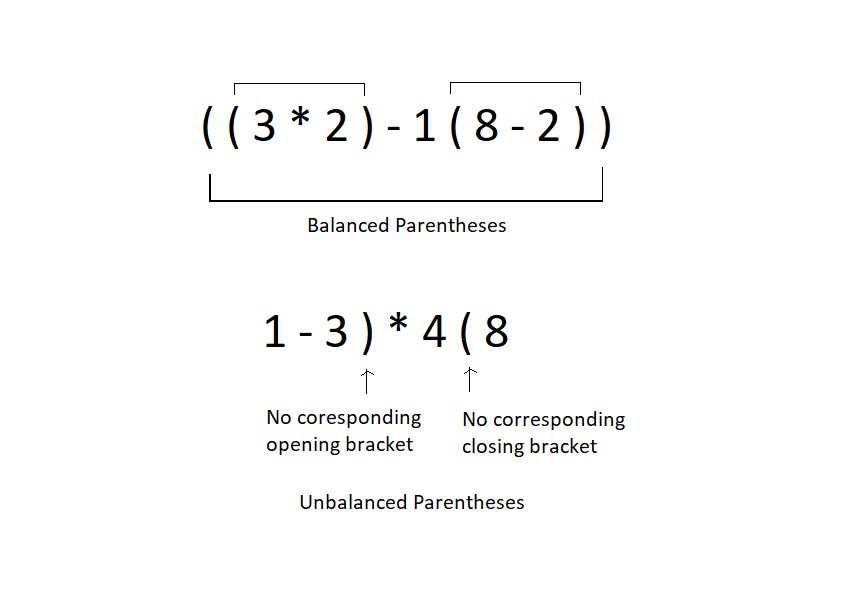
I was very excited to bring this topic to your attention. Parenthesis matching is one of the basic applications of the stack we learned about in our last ten lectures. This will be thought-provoking to you all as well. Since the dawn of programming, parenthesis matching has been a favorite topic. It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

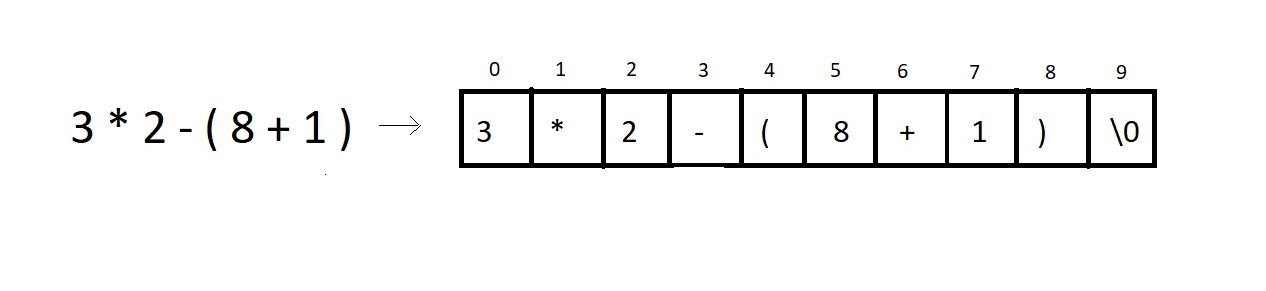
Below given illustrations would surely make it clear for you.



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.



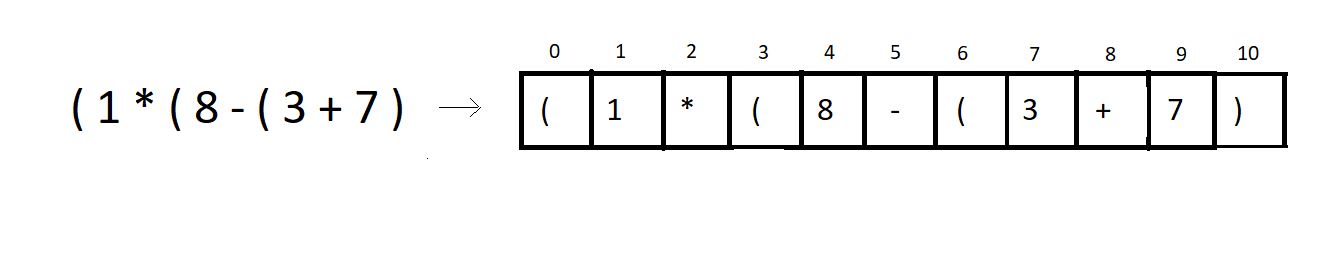
2. Iterate through the character array and ignore everything you find other than the opening and the closing parenthesis.  Every time you find an opening parenthesis, push it inside a character stack. And every time you find a closing parenthesis, pop from the stack, in which you pushed the opening bracket.

**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.

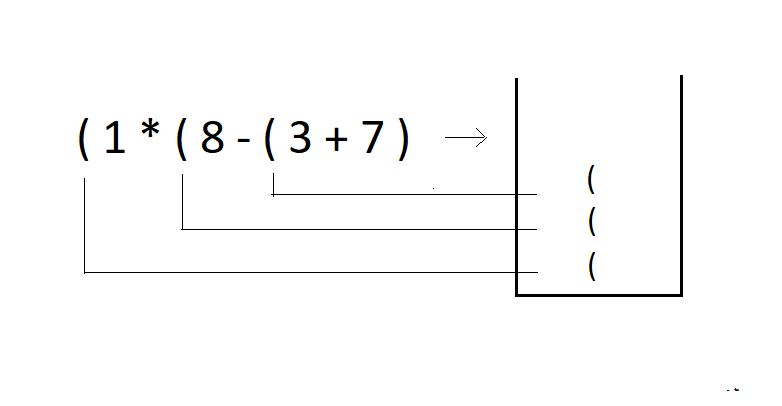
**4. Note:** Counting and matching the opening and closing brackets numbers is not enough to conclude if the parentheses are balanced. For eg: **1+3)\*6(6+2**.

**Example:**

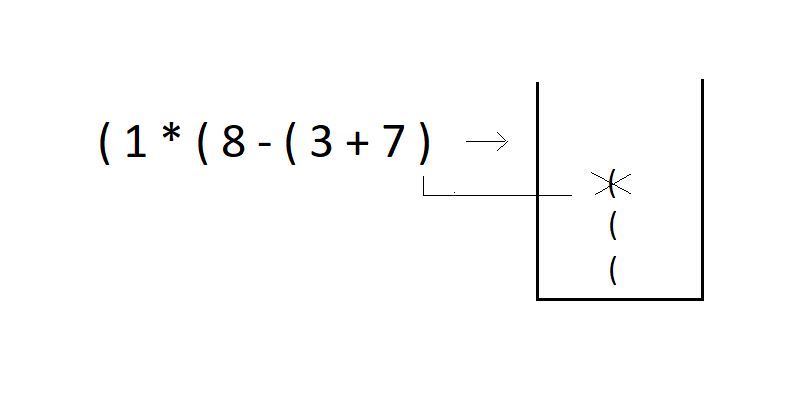


We’ll try checking if the above expression has balanced parentheses or not.

**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 )**

# 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));

***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;

}

}

**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;

}

***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");

}

***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>

***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");

}

***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>

***Figure 2: Output of the above program***

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

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

# 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:

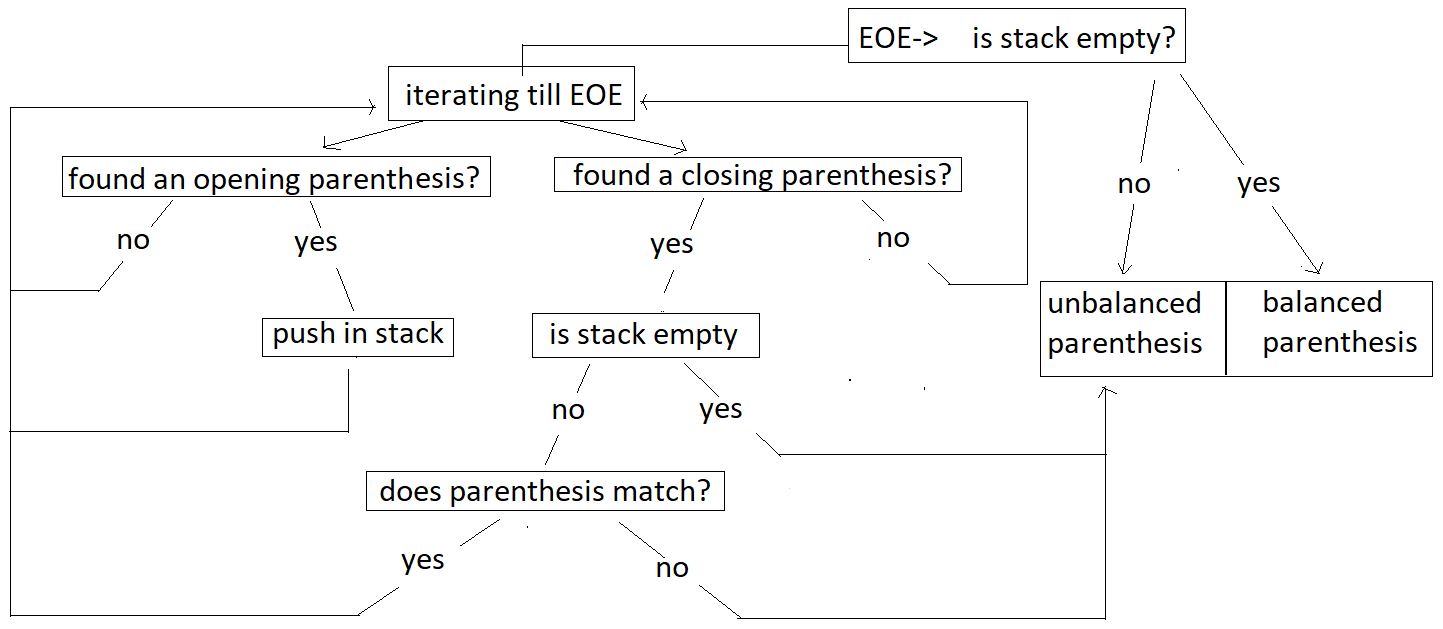
1. Whenever we encounter an opening parenthesis, we simply push it in the stack, similar to what we did earlier.

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

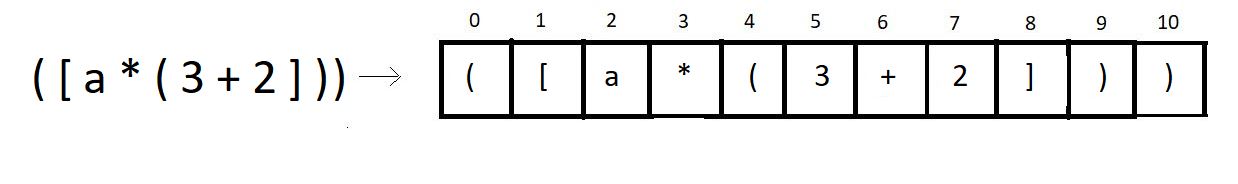
* Before we pop, this size of the stack must not be zero.
* The topmost parenthesis of the stack must match the type of closing parenthesis we encountered.

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.

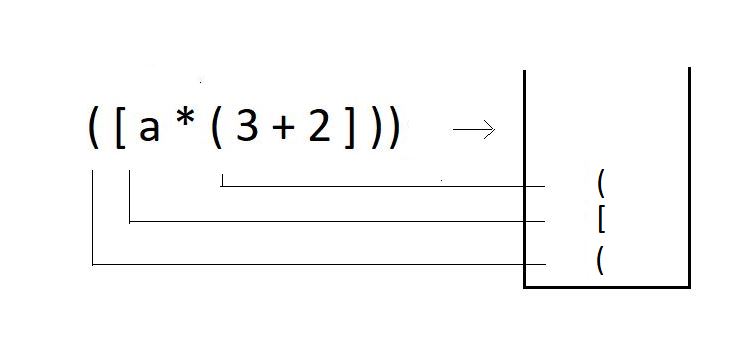


**Example:**

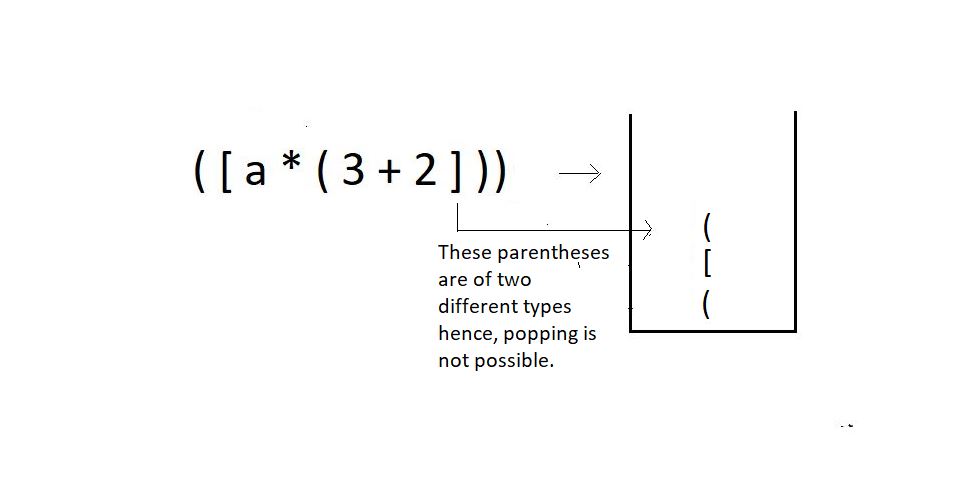


We’ll try checking if the above expression has balanced multi-parentheses or not.

**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;

}

***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;

}

}

***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;

}

***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");

***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>

**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");

}

***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>

***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.

< operand 1 >< **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**>< operand 1 >< 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.

< operand 1 >< 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?**

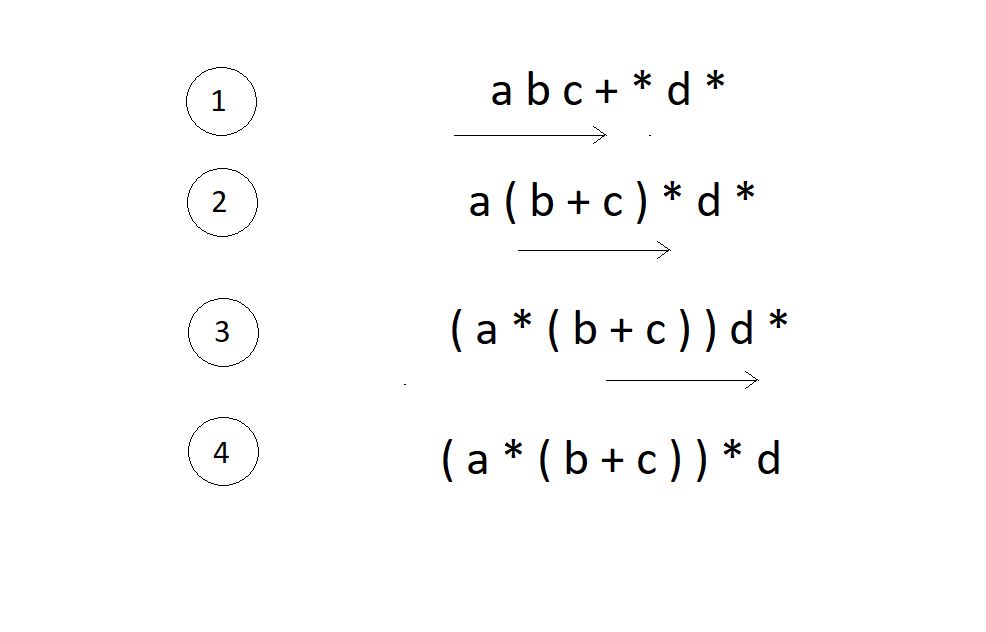
When we evaluate a mathematical expression, we have a rule in mind, named BODMAS, where we have operators’ precedence in this order; brackets, of, division, multiplication, addition, subtraction. But what would you do when you get to evaluate a 1000 character long-expression, or even longer one? You 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.

The following table lists the precedence and associativity of C operators. Operators are listed top to bottom, in descending precedence.

|  |  |  |  |
| --- | --- | --- | --- |
| **Precedence** | **Operator** | **Description** | **Associativity** |
| **1** | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| () | Function call |
| [] | Array subscripting |
| . | Structure and union member access |
| -> | Structure and union member access through pointer |
| (*type*){*list*} | Compound literal(C99) |
| **2** | ++ -- | Prefix increment and decrement[[note 1]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-1) | Right-to-left |
| + - | Unary plus and minus |
| ! ~ | Logical NOT and bitwise NOT |
| (*type*) | Cast |
| \* | Indirection (dereference) |
| & | Address-of |
| sizeof | Size-of[[note 2]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-2) |
| \_Alignof | Alignment requirement(C11) |
| **3** | \* / % | Multiplication, division, and remainder | Left-to-right |
| **4** | + - | Addition and subtraction |
| **5** | << >> | Bitwise left shift and right shift |
| **6** | < <= | For relational operators < and ≤ respectively |
| > >= | For relational operators > and ≥ respectively |
| **7** | == != | For relational = and ≠ respectively |
| **8** | & | Bitwise AND |
| **9** | ^ | Bitwise XOR (exclusive or) |
| **10** | | | Bitwise OR (inclusive or) |
| **11** | && | Logical AND |
| **12** | || | Logical OR |
| **13** | ?: | Ternary conditional[[note 3]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-3) | Right-to-left |
| **14**[[note 4]](https://en.cppreference.com/w/c/language/operator_precedence#cite_note-4) | = | Simple assignment |
| += -= | Assignment by sum and difference |
| \*= /= %= | Assignment by product, quotient, and remainder |
| <<= >>= | Assignment by bitwise left shift and right shift |
| &= ^= |= | Assignment by bitwise AND, XOR, and OR |
| **15** | , | Comma | Left-to-right |

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

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

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

4. And we are done.

**Converting infix to postfix:**

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

5. Parentheses 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 - [ y z \* ] )**.

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

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**  →  ( ( p - q ) -  ( r / a ) ) →  ( [ - p q ] - [ / r a ]  )  →**- - 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 )**→  ( ( p - q ) \* ( m - n ) ) →  ( [p q - ] *[m n - ] )  →****p q-m n -***

# 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.

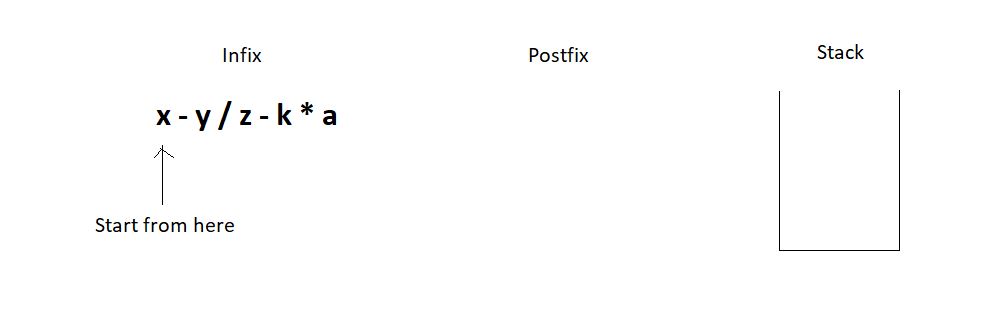
But we didn’t talk about their implementation using stacks; rather, we didn’t even mention stacks in our last class. Today, we will learn how to convert an infix expression into its postfix equivalent using stacks.

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

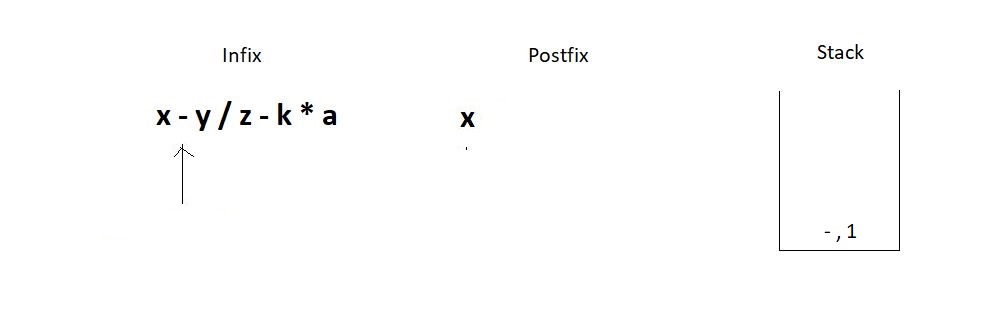
1. Start moving left to right from the beginning of the expression.
2. The moment you receive an operand, concatenate it to the postfix expression string.
3. And the moment you encounter an operator, move to the stack along with its relative precedence number and see if the topmost operator in the stack 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.

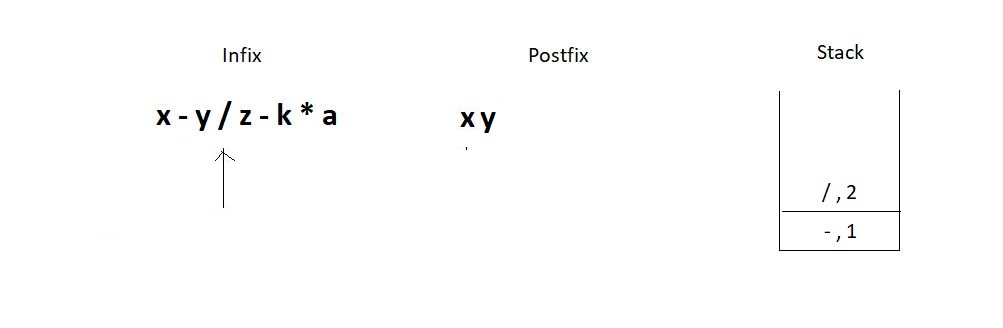
1. We will start traversing from the left.



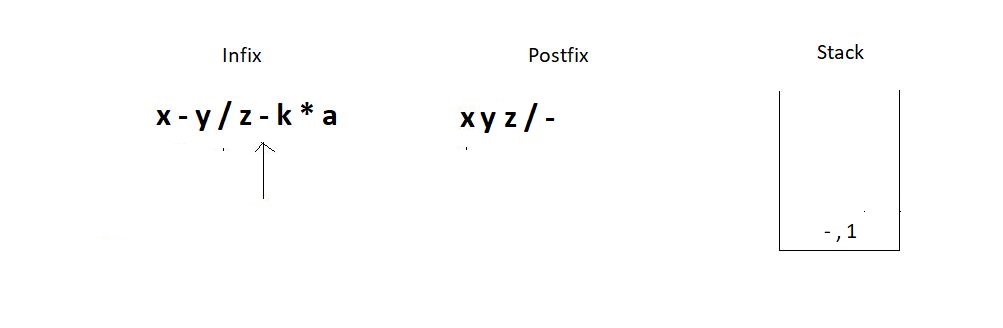
2. First, we got the letter ‘x’. We just pushed it into the postfix string. Then we got the subtraction symbol ‘-’, and we push it into the stack since the stack is empty.



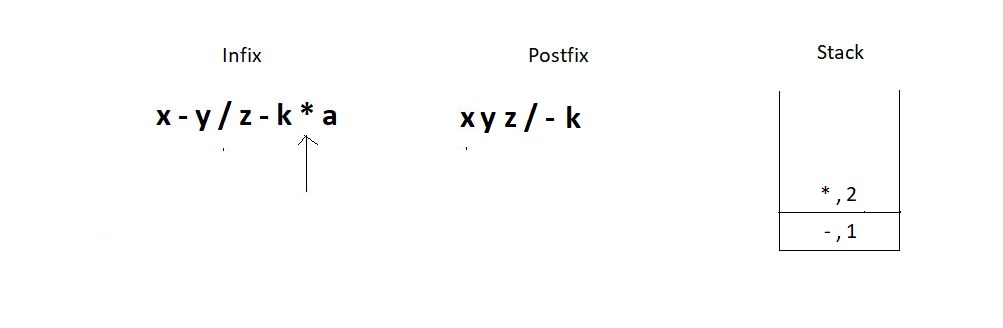
3. Similarly, we push the division operator in the stack since the topmost operator has a precedence number 1, and the division has 2.



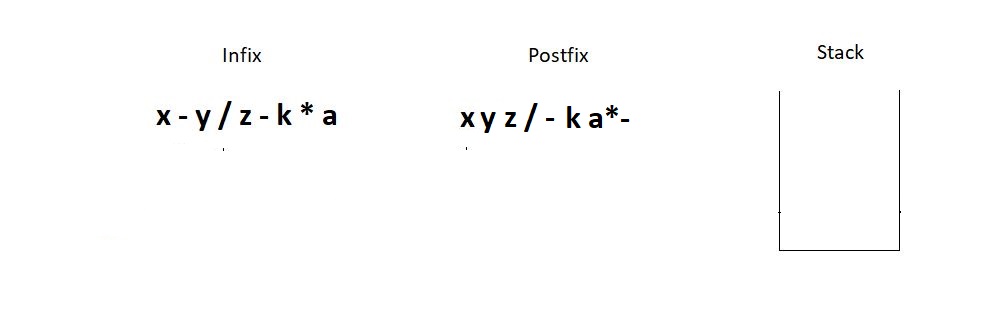
4. The next operator we encounter is again a subtraction. Since the topmost operator in the stack has an operator precedence number 2, we would pop elements out from the stack until we can push the current operator. This leads to removing both the present operators in the stack since they are both greater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



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



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



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** →  (( x - ( y / z )) - ( k \* a )) →  (( x - [ y z / ]) - [ k a \* ]  )  → [ x y z / - ] - [ k a \* ]  →**x y z / - k a \* -**

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.

# 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;

}

***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;

}

***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;

}

**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;

}

***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));

***Code Snippet 5: Calling the function infixToPostfix***

postfix is xyz/-kd\*-

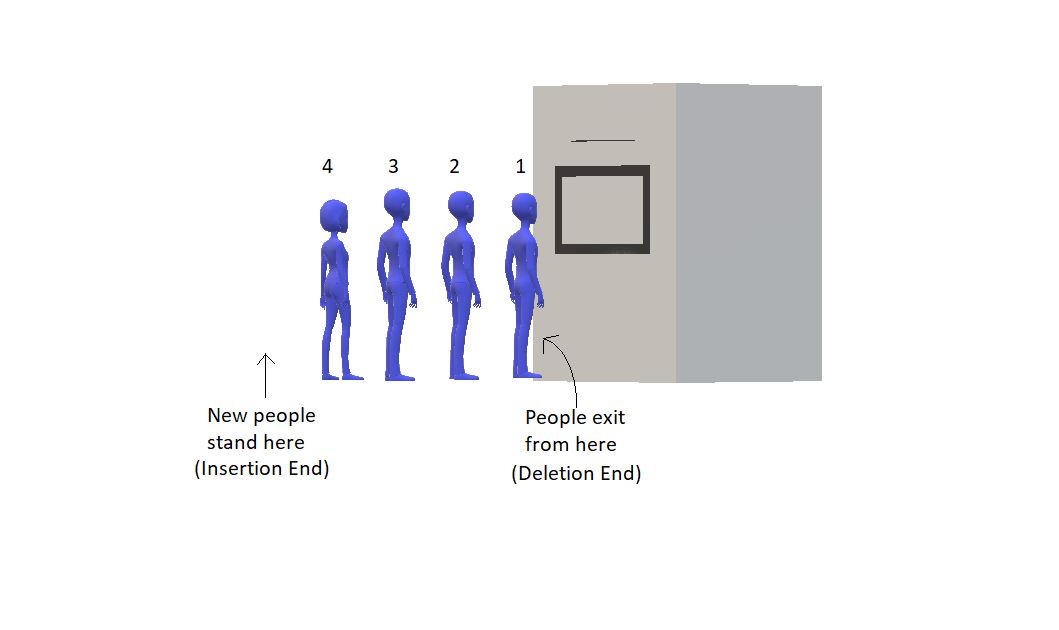
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

***Figure 1: Output of the above program***

# Queue Data Structure in Hindi

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

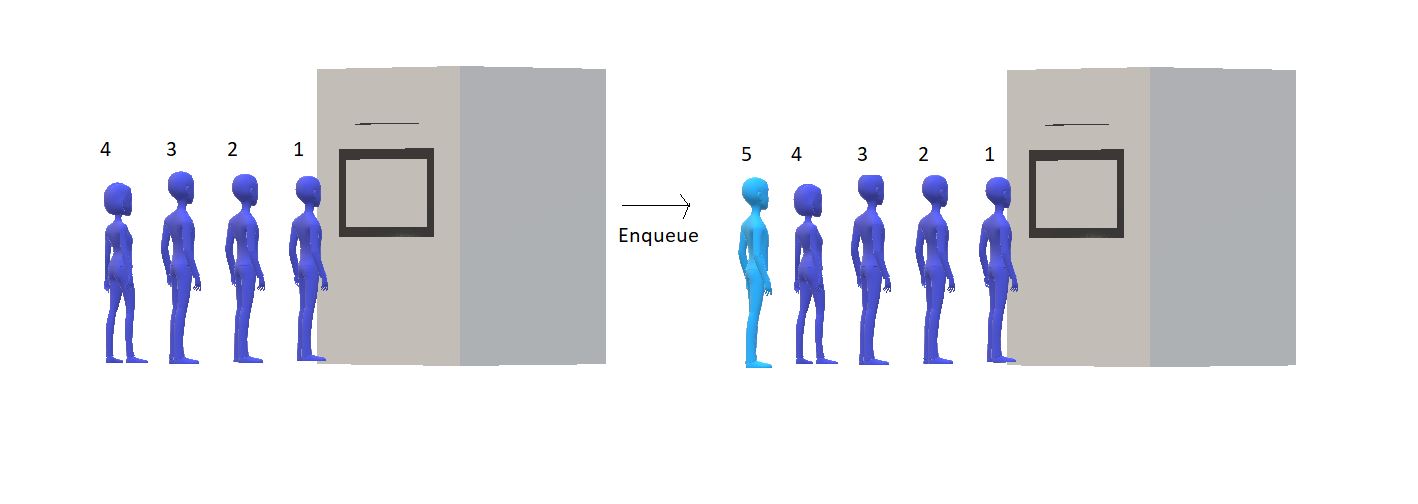
**Data:**

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.

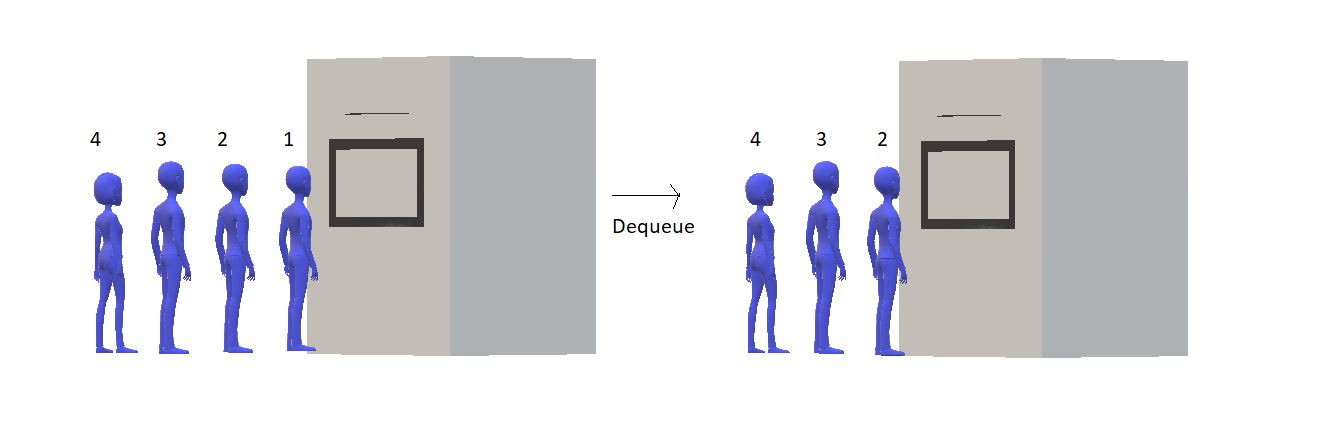
**Methods:**

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.

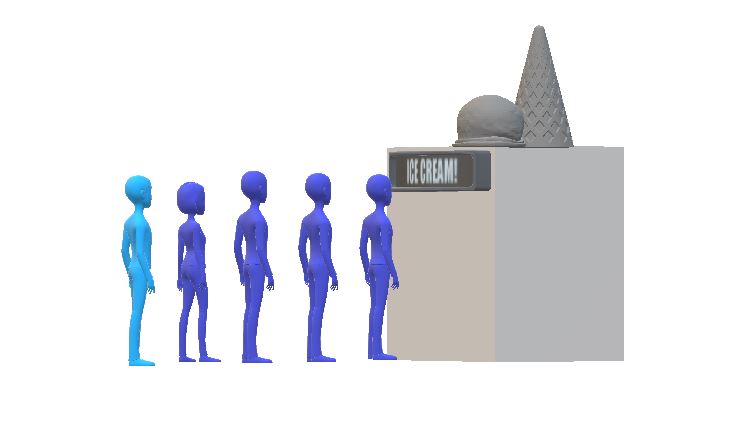
This was our abstract data type, queue. We have in this what we thought would suffice our needs for now. The list could be longer, but in my opinion, this is sufficient.

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.

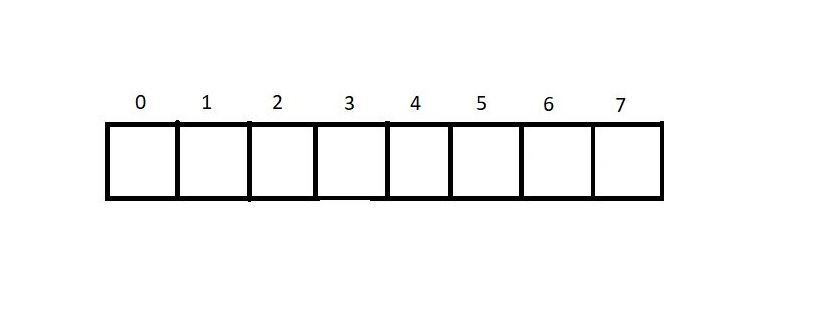
A queue is a collection of elements with certain operations following FIFO (First in First Out) discipline. We insert at one end and delete from the other. And this is what you have to keep in mind for now.

**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.



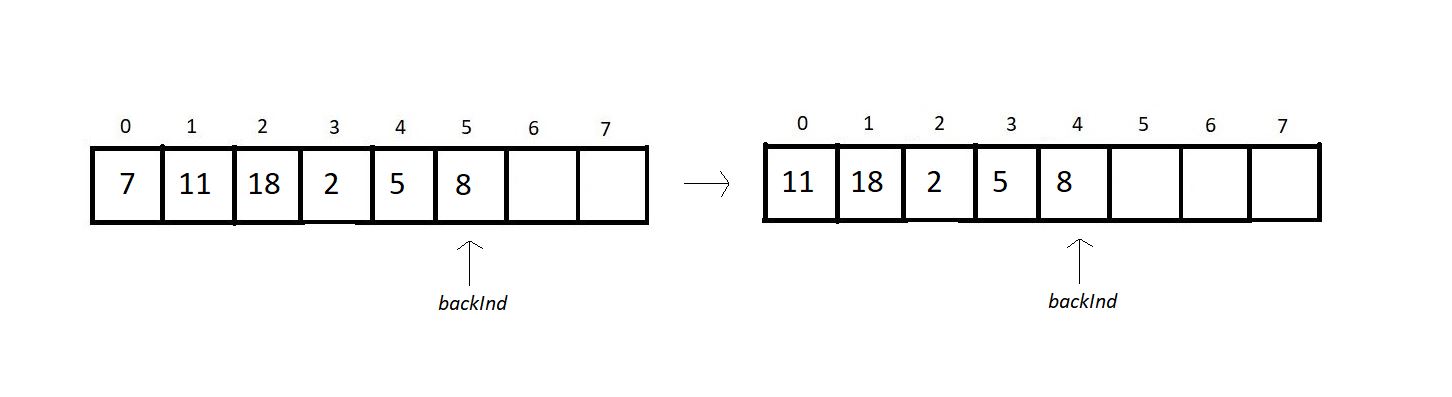
Here, we have shown a branded ice cream shop that is famous enough to have a queue of people waiting to get one of their choices. And the shop owner wants to store the information of these people, so he uses an array to accomplish that. Assuming that we have 8 people and we want to store their information,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

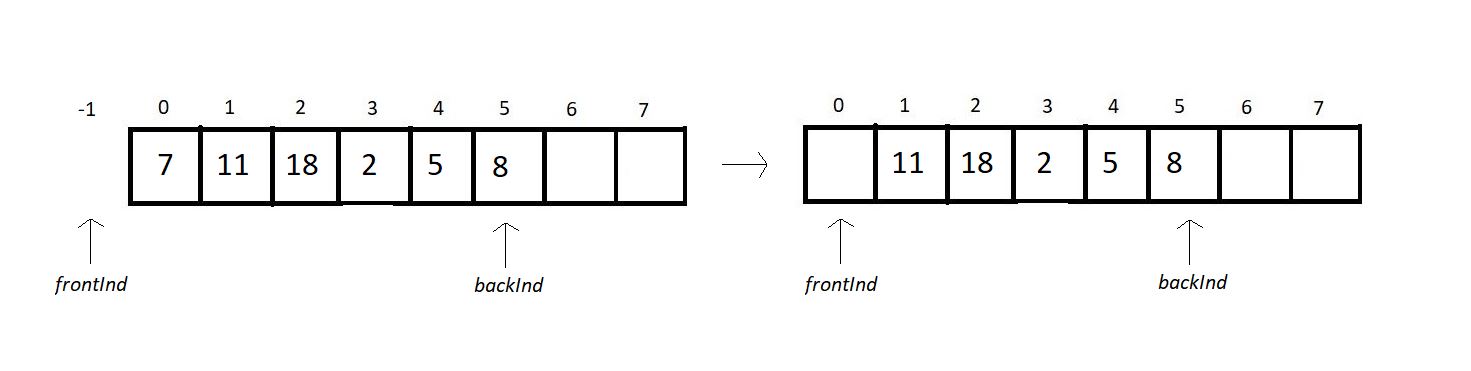
* Remove the element at the zeroth index( no need for that in an array )
* Increment *frontInd*by 1.
* Time complexity: O(1)

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:



The act of optimizing a solution/program is very important, and you should always strive for a better solution to a problem. And a solution that takes less time is always preferred. So, this is how we implement the queue ADT using an array.

# 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 ):

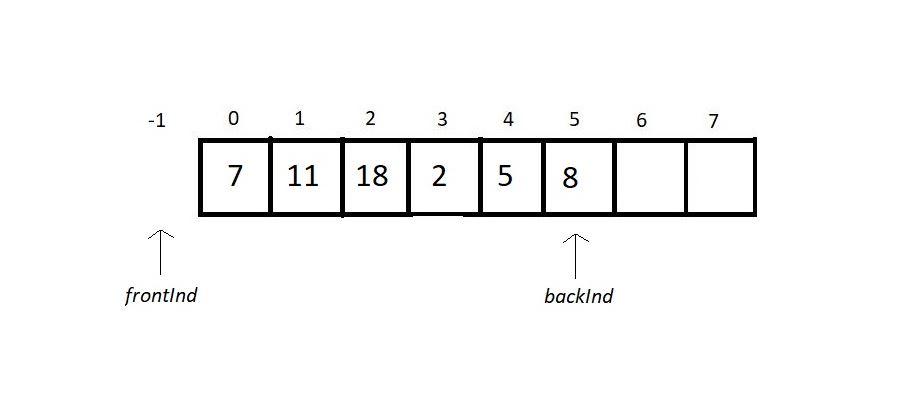
* Remove the element at the zeroth index( no need for that in an array )
* Increment frontInd by 1.
* Time complexity: O(1)

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;

};

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

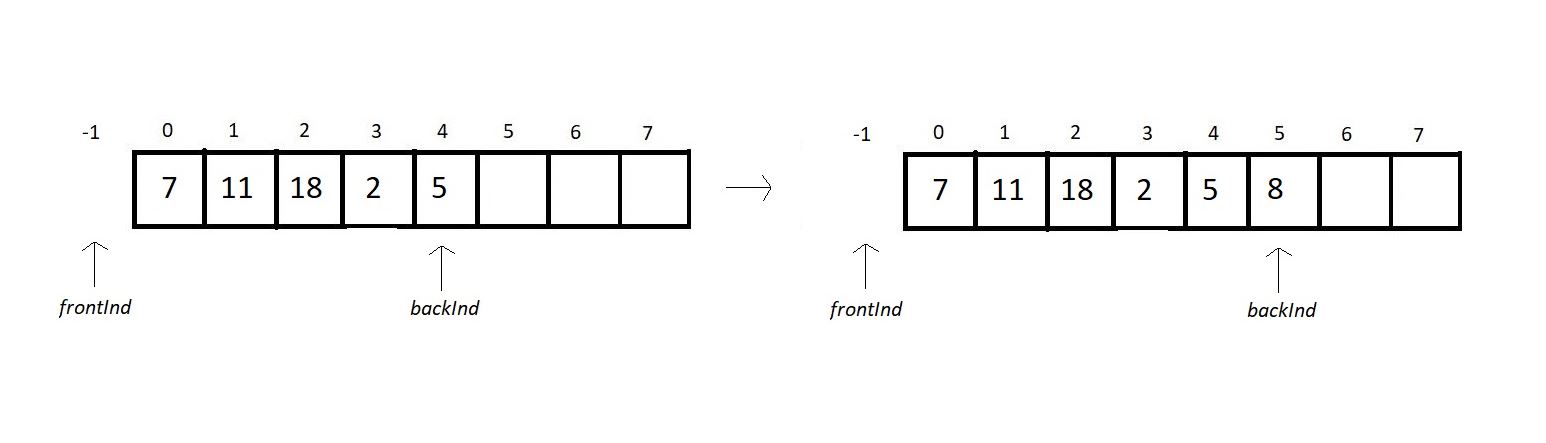
1. struct Queue q; (we are not dynamically allocating q here for now, as we did in stacks).
2. Use dot here, and not arrow operator to assign values to struct members, since q is not a pointer.
3. q.size = 10; (this gives size element the value 10)
4. q.frontInd = q.backInd = -1;(this gives both the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

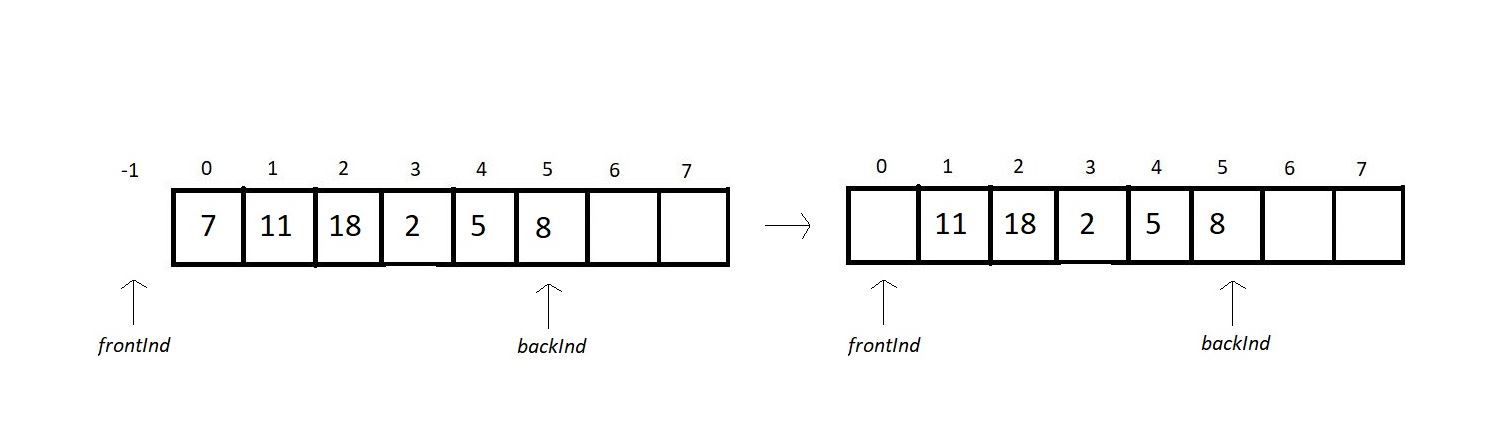


##### Dequeue:

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:**

1. 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:**

1. 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.

# 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:**

1. 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;

};

***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));

***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("Enqued element: %d\n", val);

}

}

***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;

}

***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;

}

***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;

}

***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("Enqued 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;

}

***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");

}

***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>

***Figure 1: Output of the above program***

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

enqueue(&q, 12);

enqueue(&q, 15);

***Code Snippet 9: Using the enqueue function***

Our terminal had the following output:

Enqued element: 12

Enqued element: 15

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

***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));

***Code Snippet 10: Using the dequeue function***

And the output we received was:

Dequeuing element 12

Dequeuing element 15

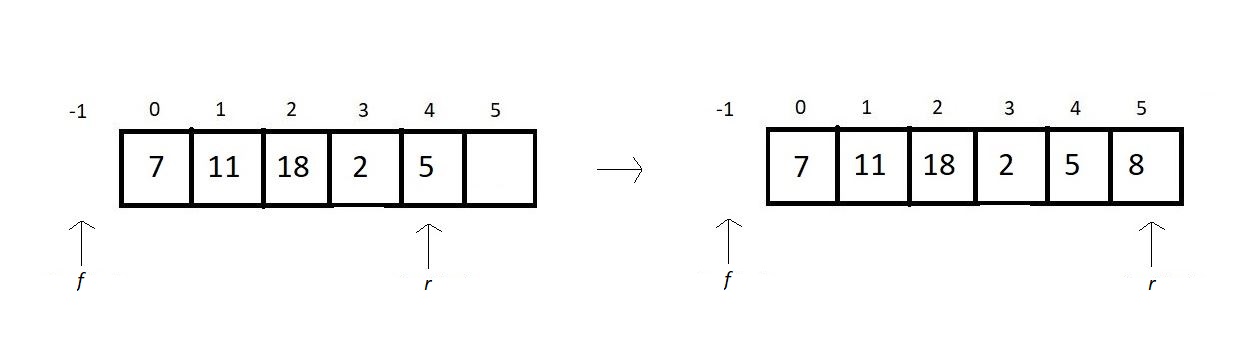
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

***Figure 3: Output of the above program***

# Introduction to Circular Queue in Data Structures

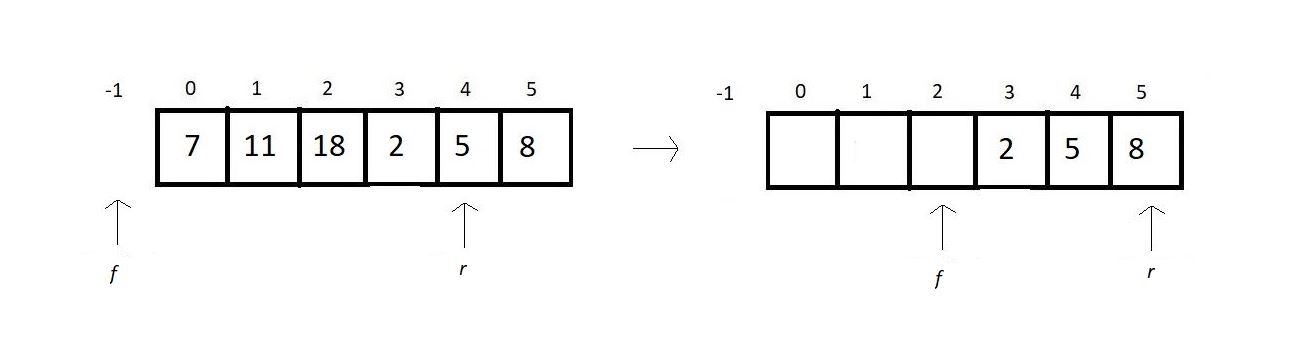
We have completed learning the basics of queues and its operations so far. Before proceeding to the next section, let's catch up on what we covered in queues. Using a real-life example, we explored the meaning of queue. We learn that it follows the FIFO principle. We implemented a queue ADT and its basic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

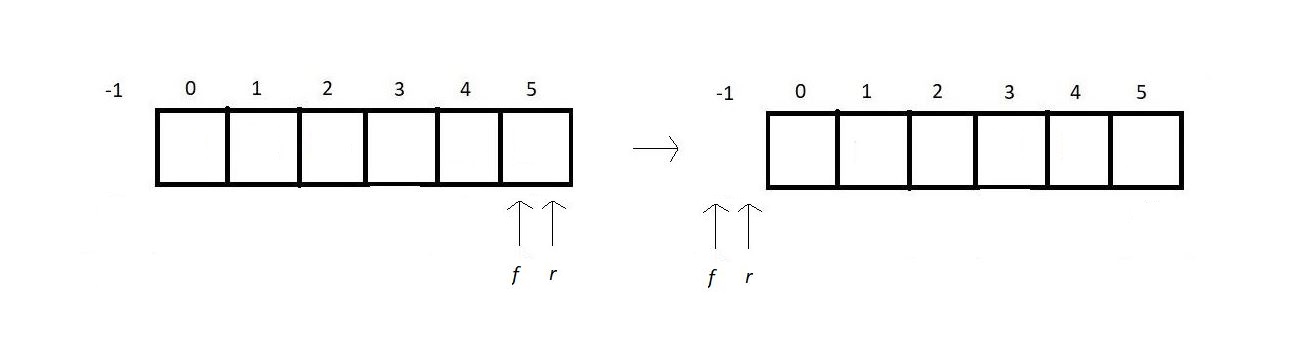
**

***Figure 2:  Dequeuing leaves vacant spaces behind***

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

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

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***

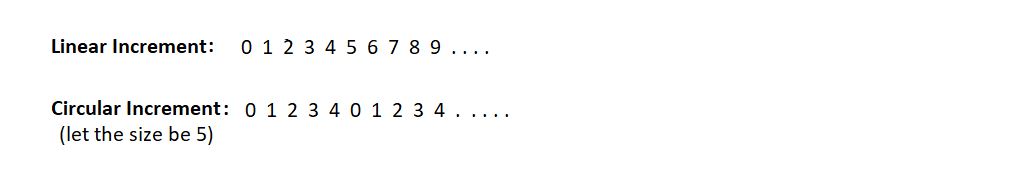
However, the efficiency of this method is limited by the requirement that the front and rear be the same. It is ineffective in the case of figure 2. Therefore we need a more optimized solution. This is when **circular queues** come to the rescue.

#### Circular queues:

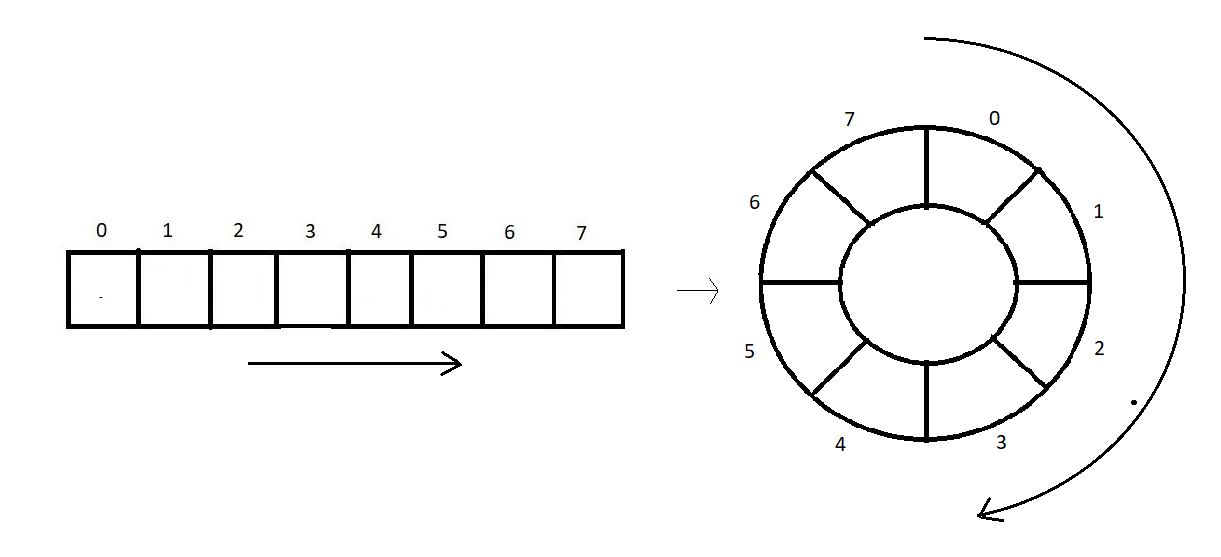
In circular queues, we mainly focus on the point that we don’t increment our indices linearly. Linearly increasing indices cause the case of overflow when our index reaches the limit, which is size-1.

In linear increment, i becomes i+1.

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



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***

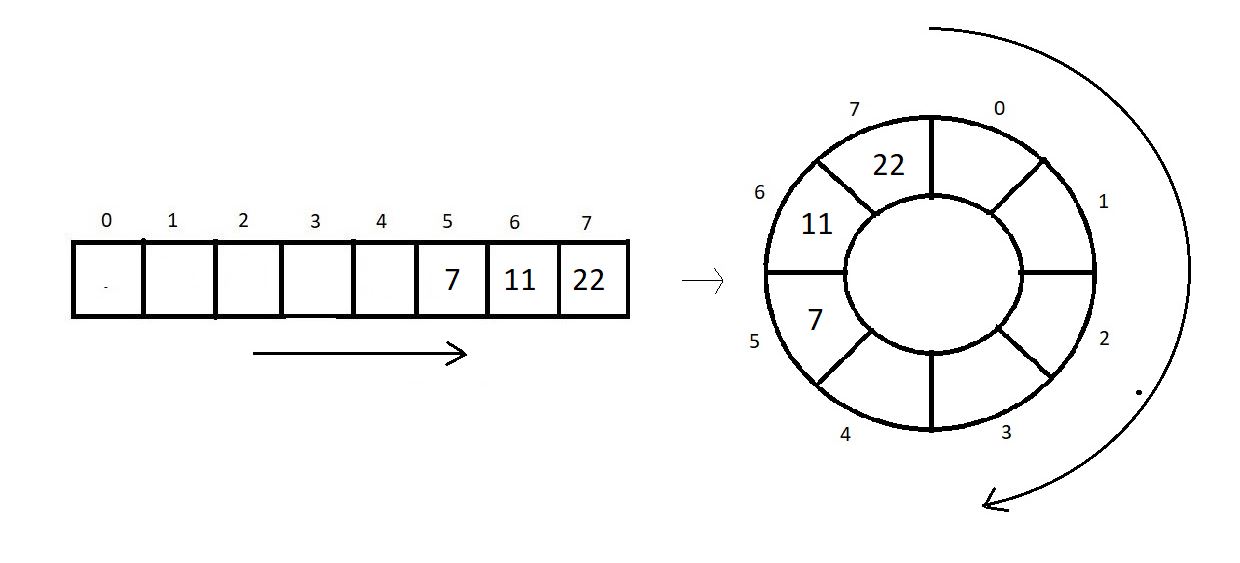
And this is the circular implementation of the same array we used to implement linearly. This allows the leftover spaces to be used again. This wheel type array is called the **circular queue**.

# 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 enqueuing even if the spaces behind go unused.

But once you convert this linear/normal queue into a circular one, it enables further enqueuing 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.



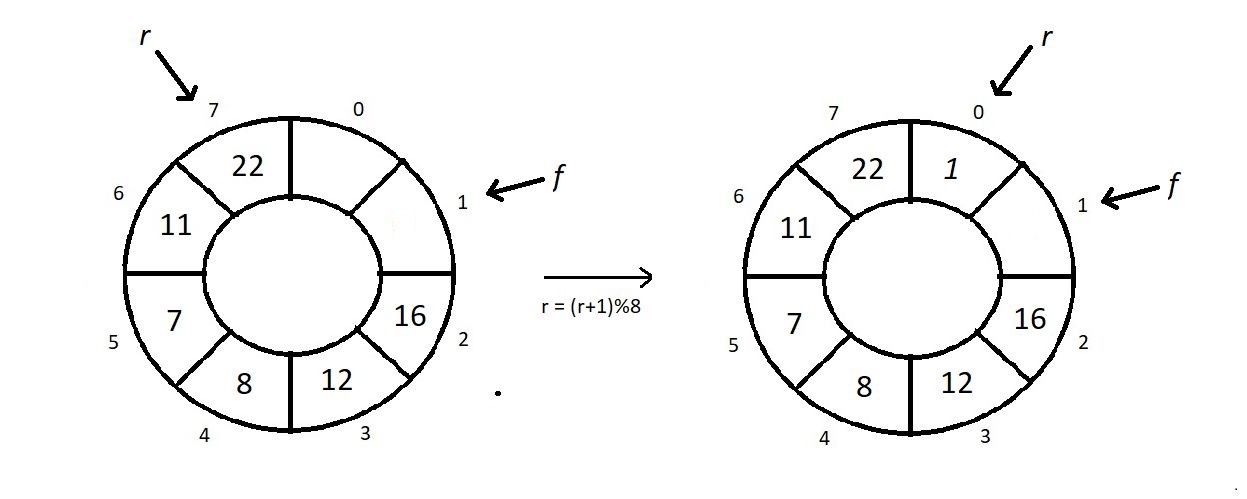
Note: Circular increment lets us access the queue indices circularly, which means, after we finish visiting the 7th index in the above illustration, we again come at the zeroth index.

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.

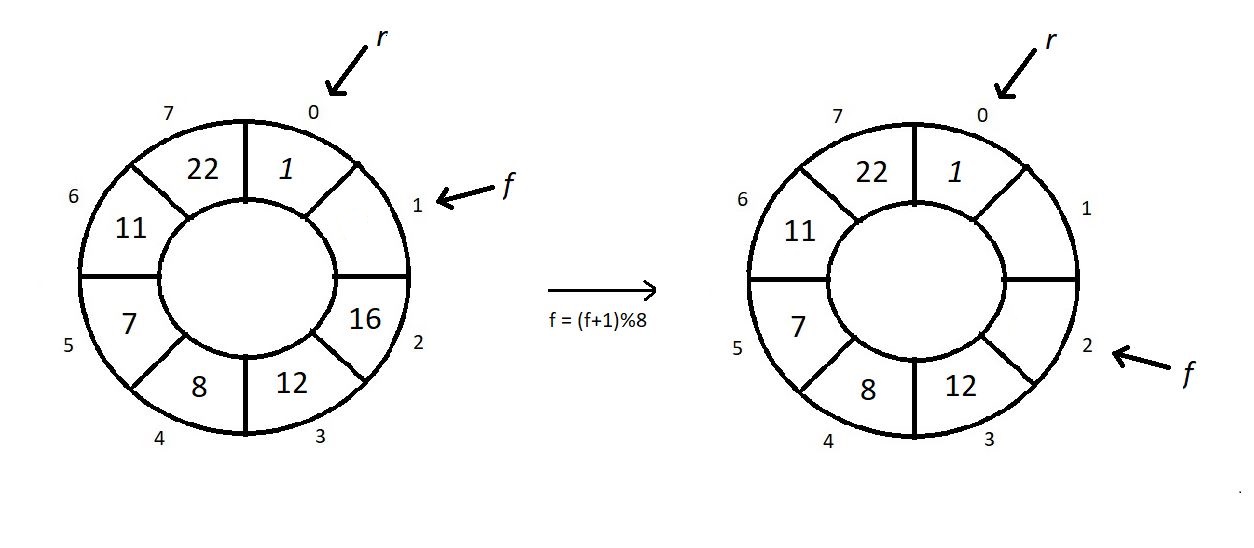


Now, since the f is just next to the r, the queue is full, and no more elements can get pushed.

##### **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 in Hindi

We have already finished learning about the implementation of circular queues using arrays. We saw the algorithms behind enqueuing 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;

};

**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));

**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;

}

**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 element of q) +1 is equal to the (f element of q). 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;

}

**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("Enqued element: %d\n", val);

}

}

**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;

}

**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("Enqued 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");

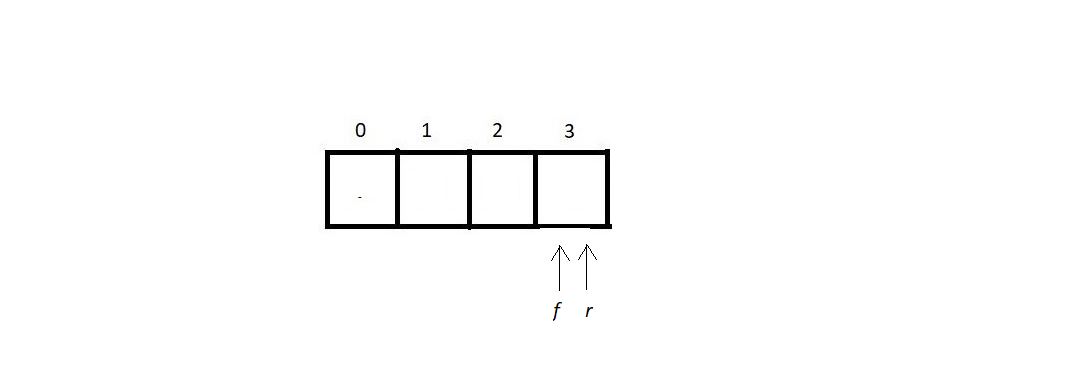
}

return 0;

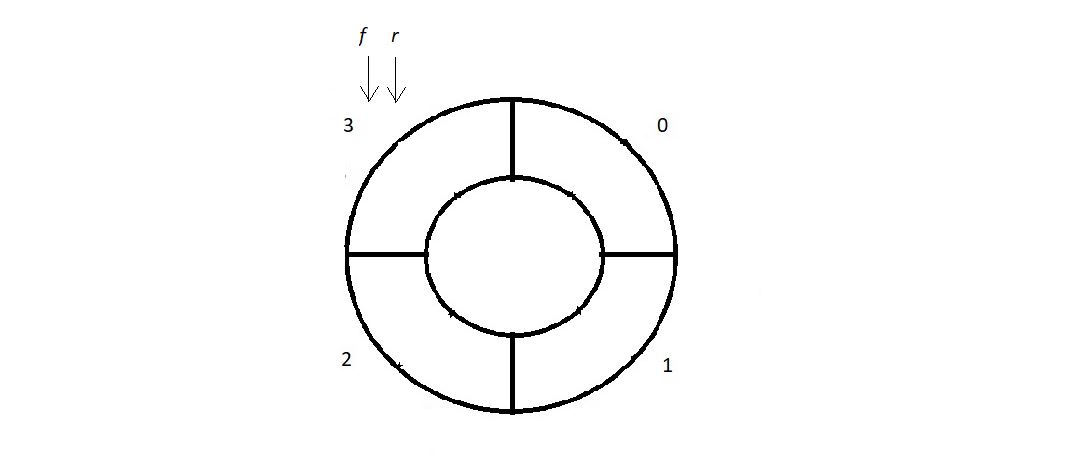
}

**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");

}

**Code Snippet 8: Using the *enqueue*function**

Our terminal had the following output:

Enqued element: 12

Enqued element: 15

Enqued element: 1

Queue is full

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

**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");

}

**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>

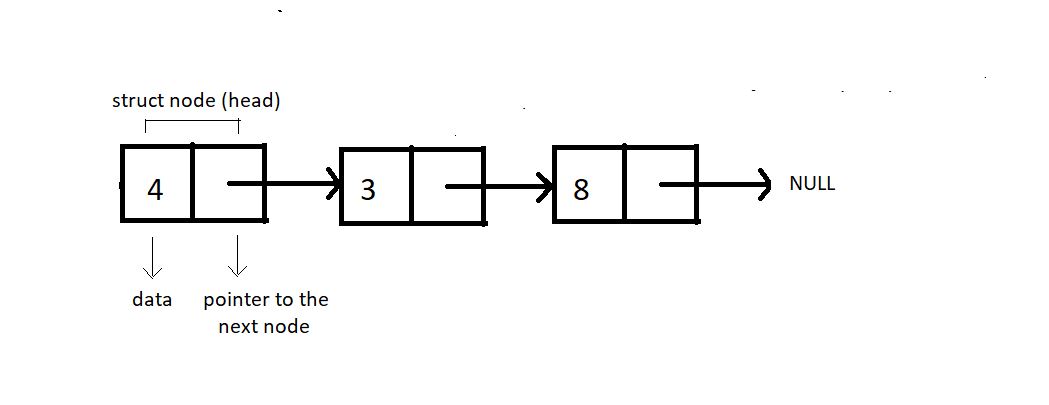
**Figure 2: Output of the above program**

# Queue Using Linked Lists

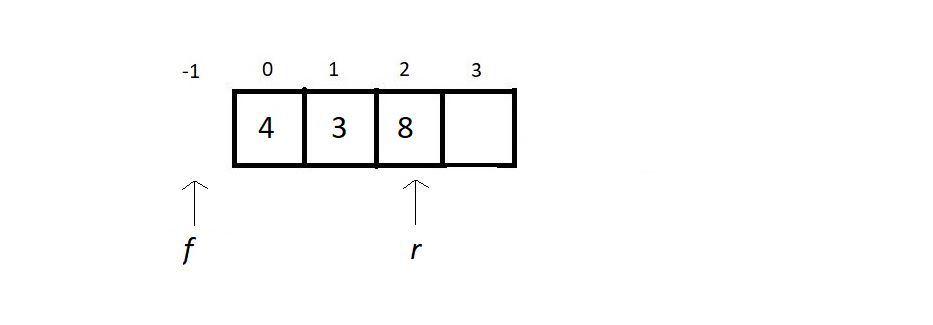
Up until now, queues had been implemented using arrays. We have another alternative that is a must to learn and has been examiners’ favorite topic, implementing queues using linked lists. Today, we’ll see how to implement queues using linked lists and some of its operations.

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.

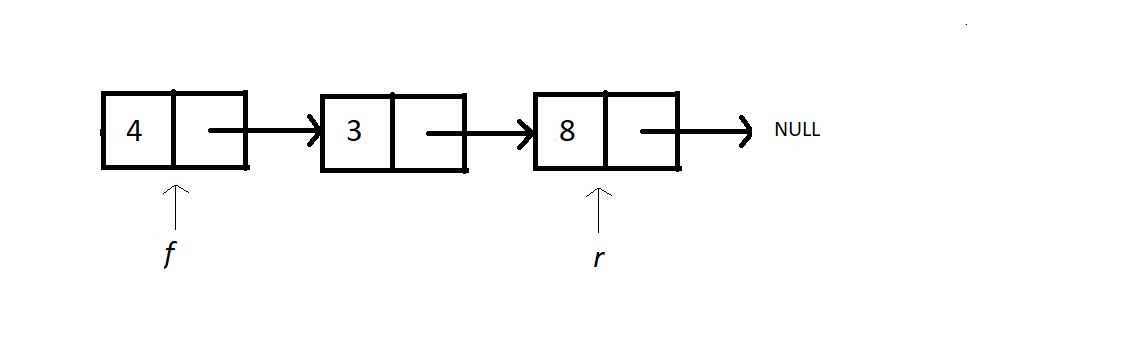
If you remember, linked lists are chain-like structures with nodes having two parts, an integer variable to hold data and another node pointer to hold the address of the next node. Below illustrated is a linked list with three nodes. The last node points to NULL. And the first node is called the head.



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.



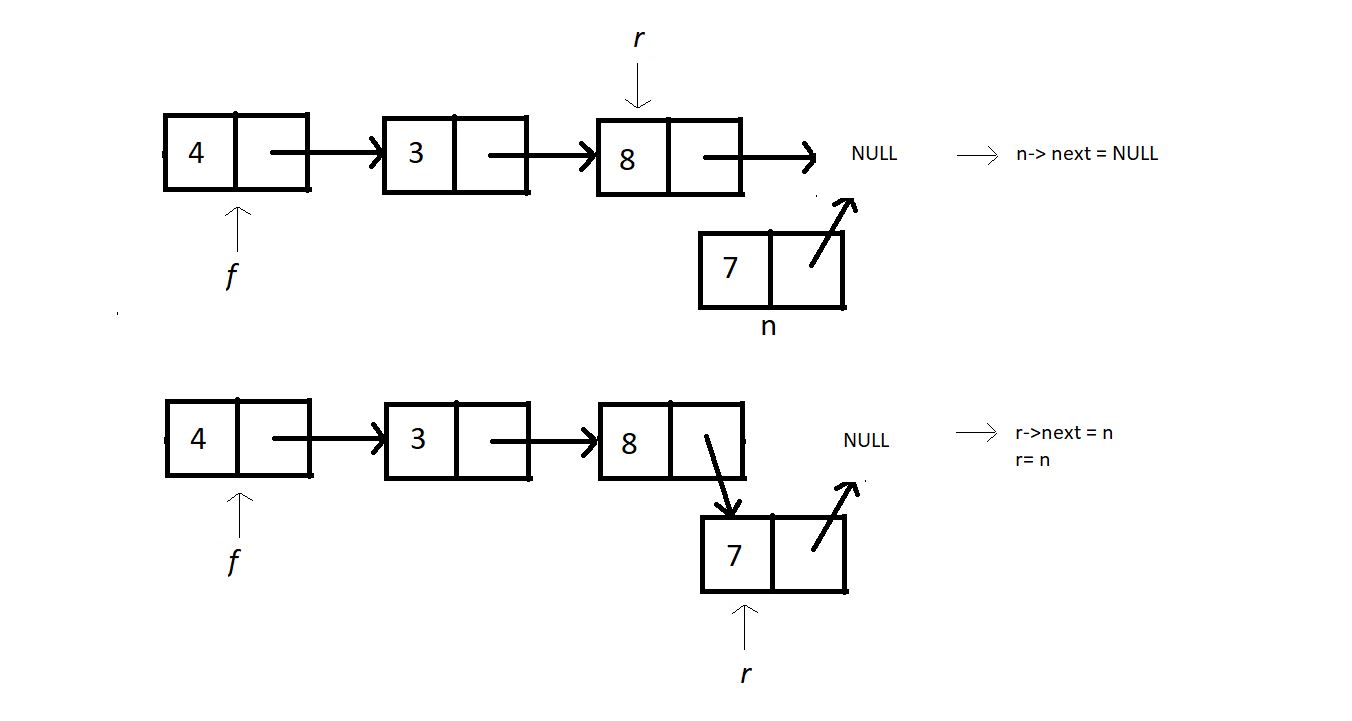
Since we are implementing this queue using a linked list, the index variables are no longer integers. These become the pointers to the front and the rear nodes. And the queue somewhat starts looking like this.



#### 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 f node. And don’t forget to store the data of the f node 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 f node is NULL, which means there is no beginning, hence no element.

##### **Condition for isFull:**

Queues implemented using linked lists never usually become full until the space in the heap memory is exhausted. Therefore, the only condition for the queue linked list to be full is that the new node becomes NULL when created.

# 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;

};

**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;

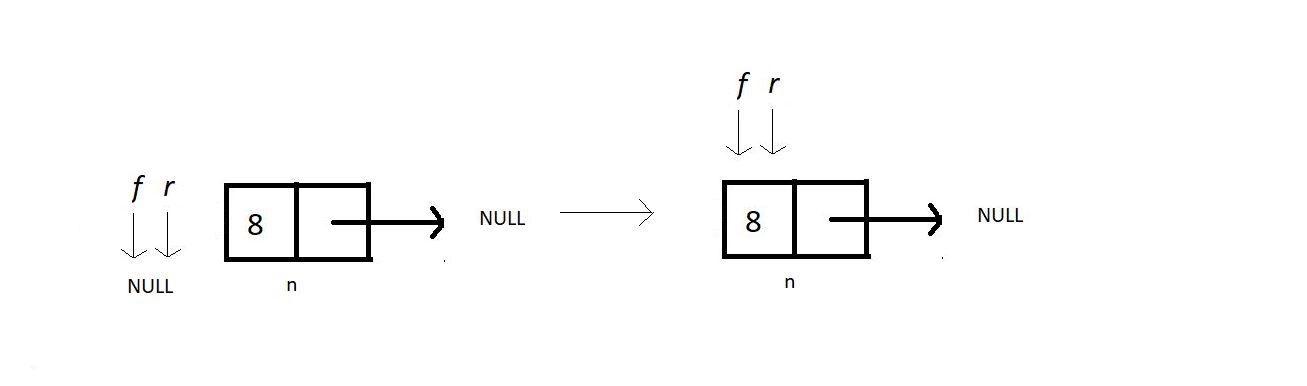
}

}

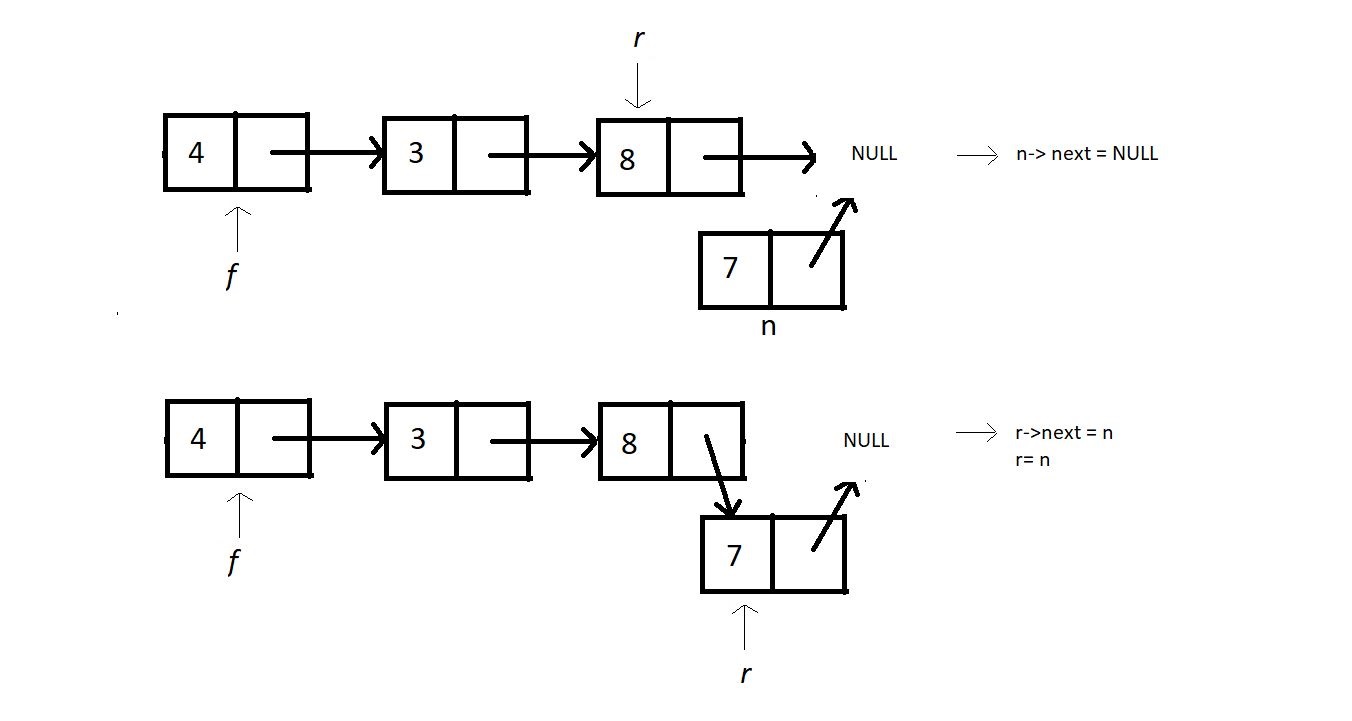
}

***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;

}

**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;

}

***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);

***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

***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);

***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

***Figure 2: Output of the above program***

As you can observe, there was no element left to display, and element 34 went out first since it entered the first.

# 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.

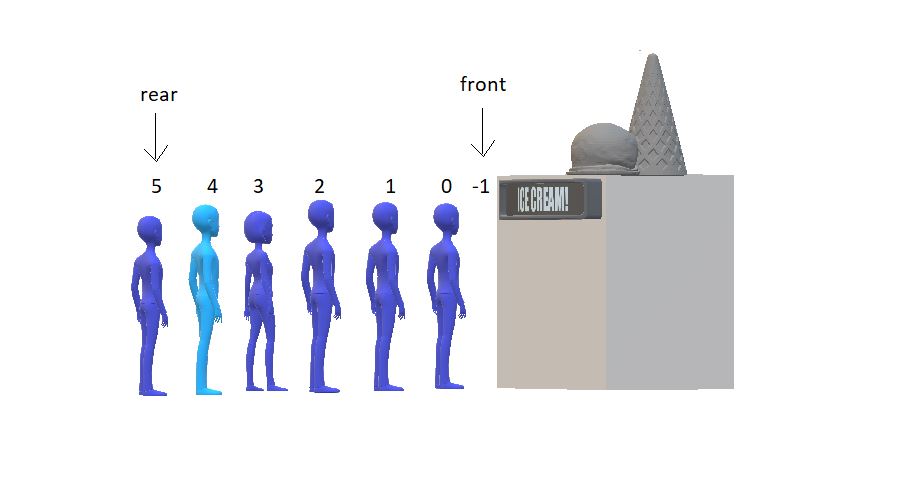
Double-ended queues, hence, have the following characteristics:

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.

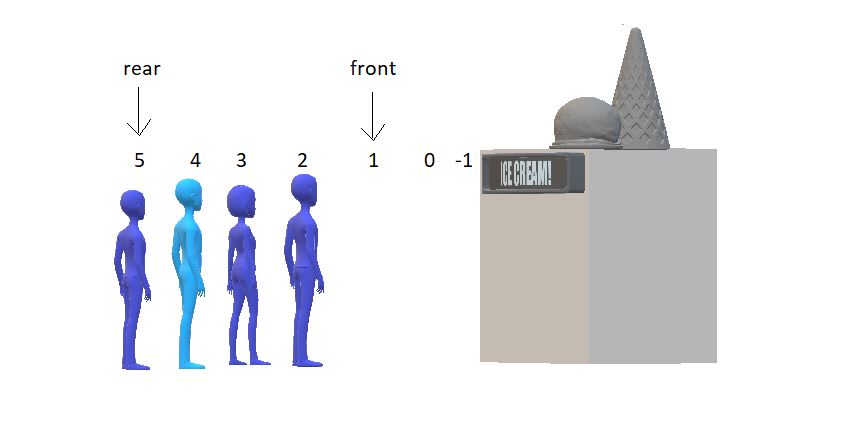
You would assume the implementation part of double-ended queues to be on the tough side, but believe me, it is straightforward to consume. I’ll use illustrations to make you understand things better.

#### Insertion in a DEQueue:

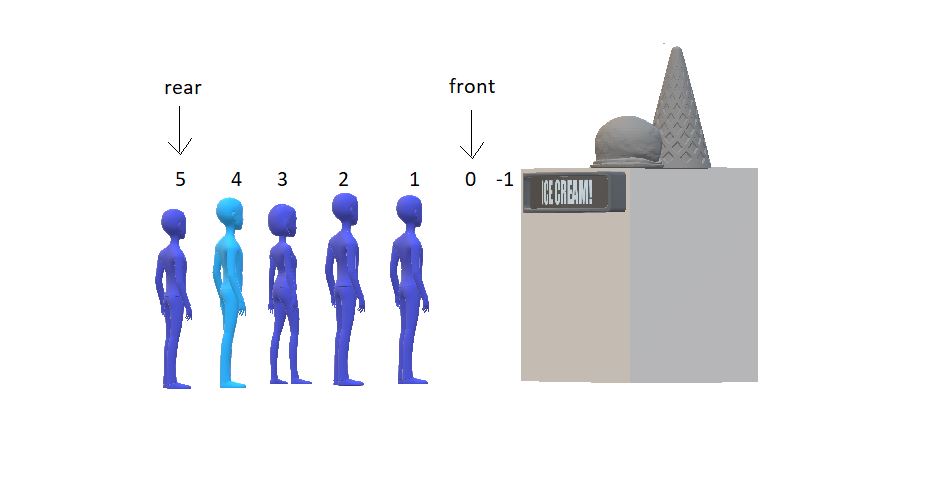
Insertion in a DEQueue is very intuitive. Follow the illustration below:



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:

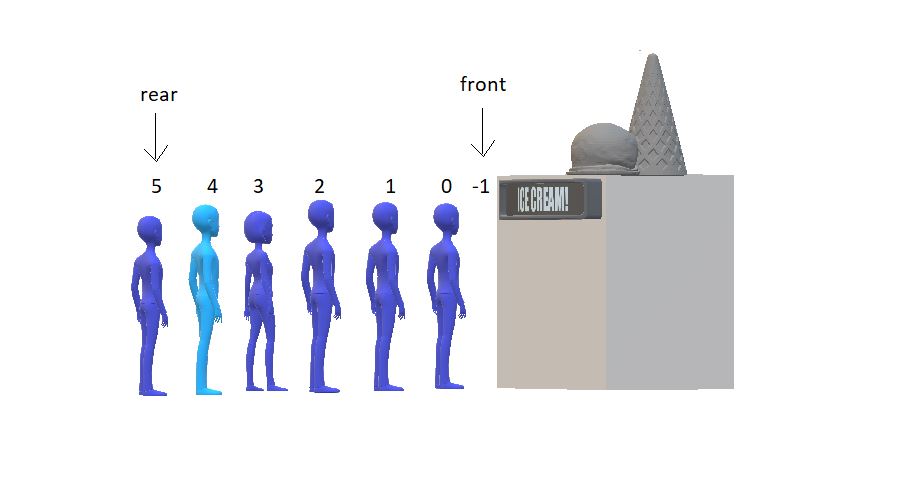


Now, we have 2 places to fill in front as well. And in DEQueue, we have no restrictions. We would just fill our new element at the front and decrease its value by 1. And that would be it. See the results below:

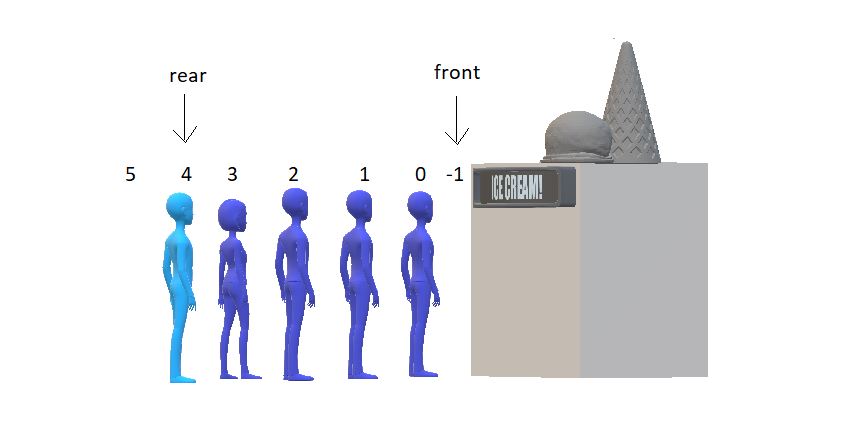


#### Deletion in a DEQueue:

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](https://www.codewithharry.com/videos/data-structures-and-algorithms-in-hindi-38).

**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()

You can even have more of these, as initialise(), print(), etc. This was our abstract data type, DEQueue. Do implement their programs, and had it not been under your capability,

**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:**

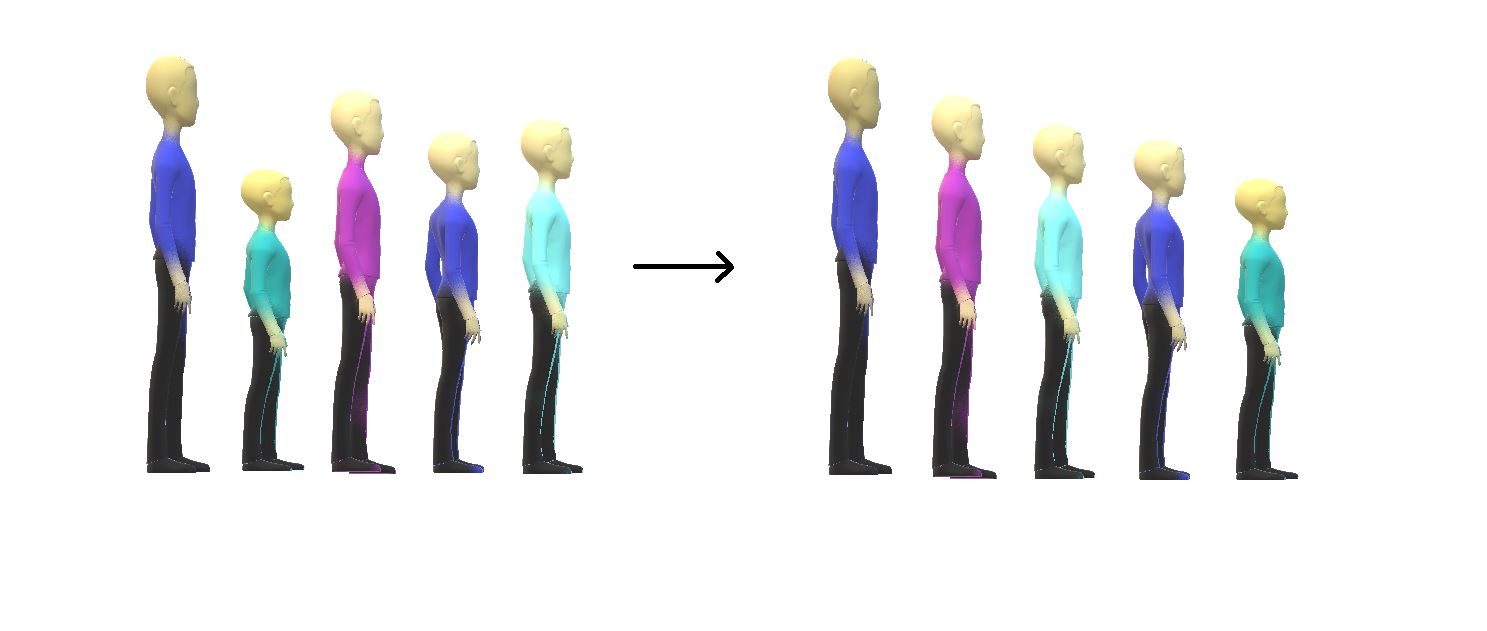
Sorting any set of elements in descending order refers to arranging the elements, let them be numbers, from the largest to the smallest. E.g., the same set(1, 9, 2, 8, 7), when sorted in descending order, becomes (9, 8, 7, 2, 1).

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(N2), and most of our algorithms’ time complexity revolves around these two.

**Note:**Lesser the time complexity, the better is the algorithm.

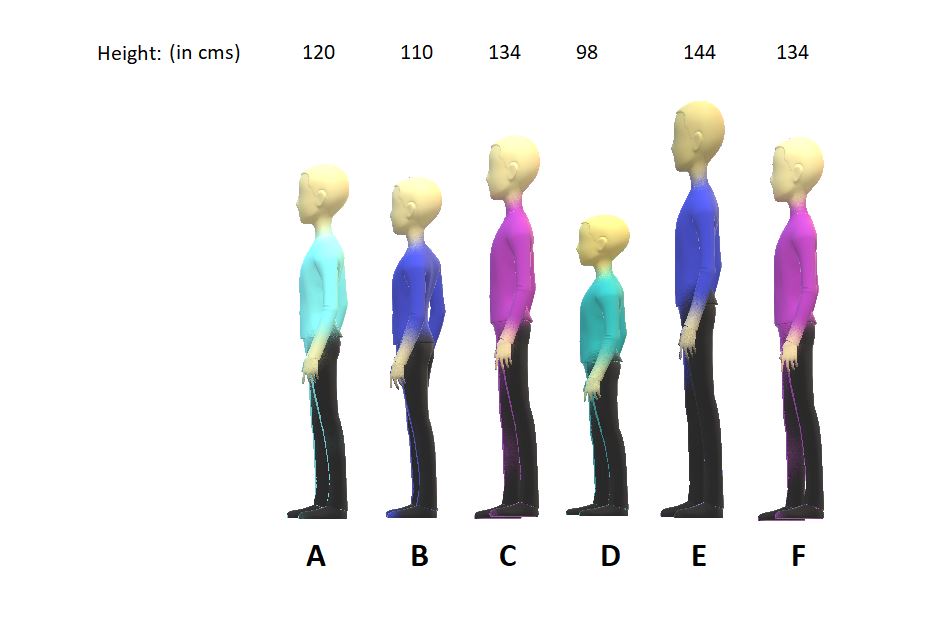
Space Complexity:

* 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:

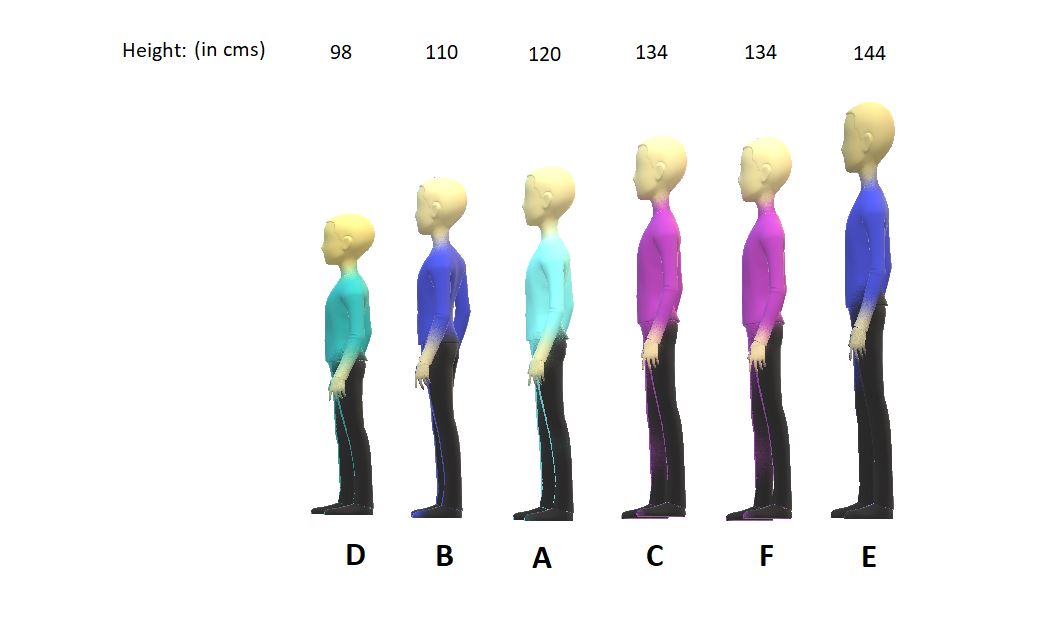
The stability of an algorithm is judged by the fact whether the order of the elements having equal status when sorted on some basis is preserved or not. It probably sounded technical, but let me explain.

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.

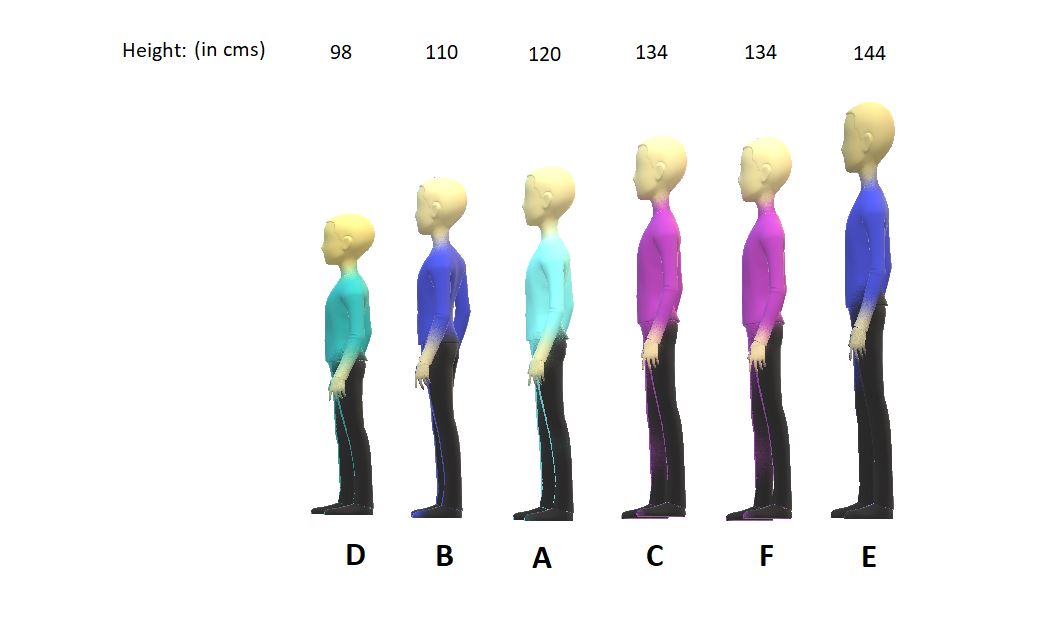
An illustration for your better understanding:  


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.

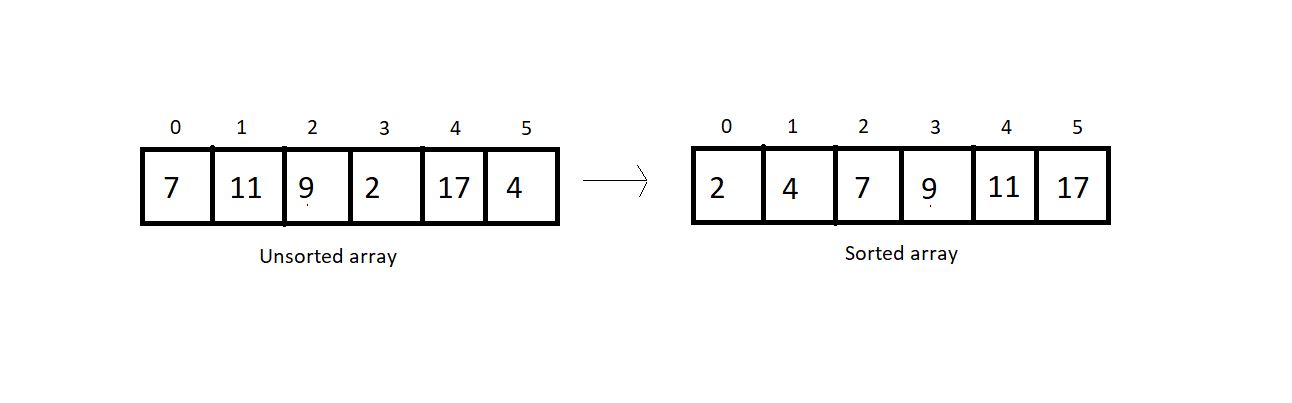
Recursiveness:

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

**Bubble Sort Algorithm in Hindi**

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.

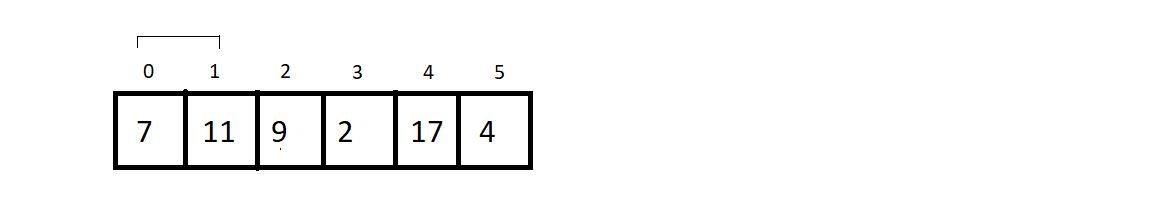
At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array 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.

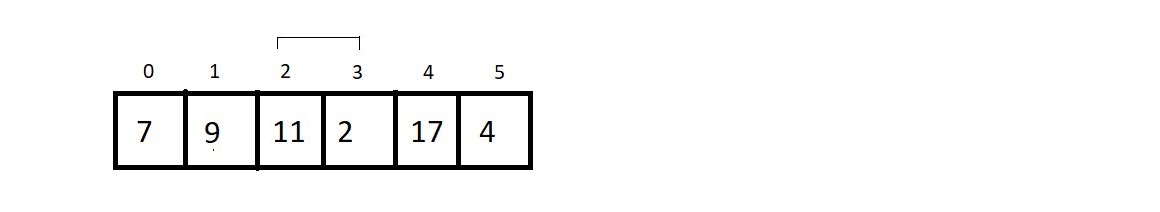
Let’s start with the first one.



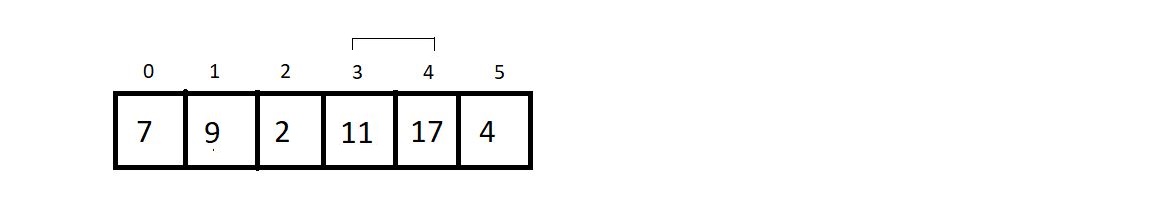
Since these two are already sorted, we move ahead without making any changes.



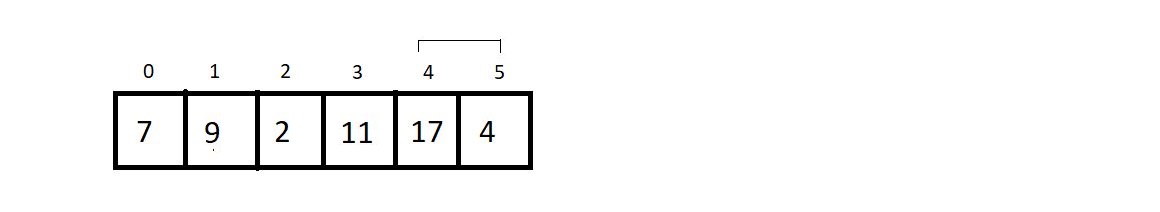
Now since 9 is less than 11, we swap their positions to make them sorted.



Again, we swap the positions of 11 and 2.

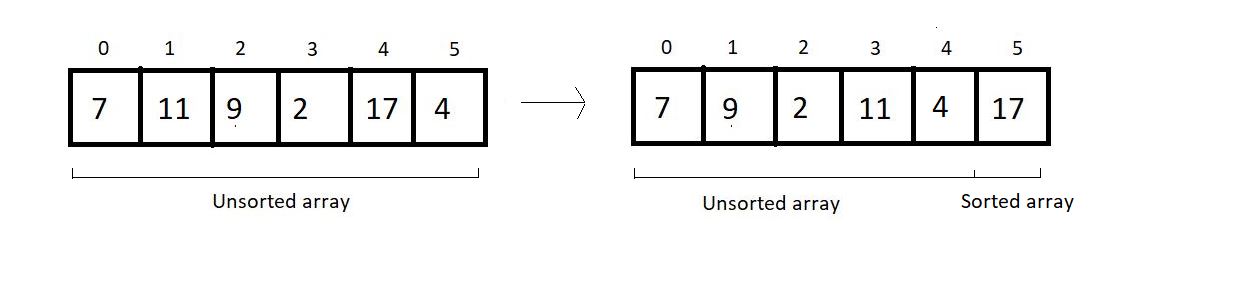


We move ahead without changing anything since they are already sorted.



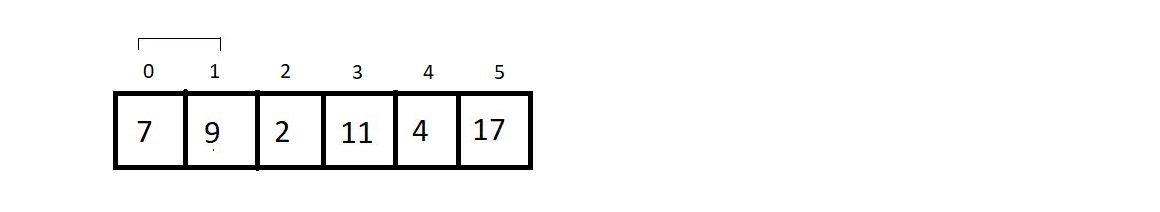
Here, we make a swap since 17 is greater than 4.

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

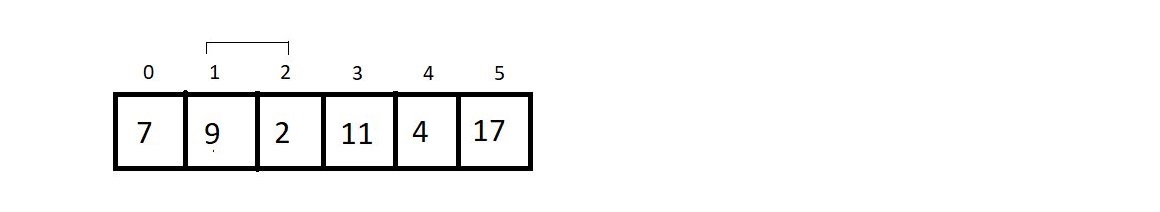


**2nd Pass:**

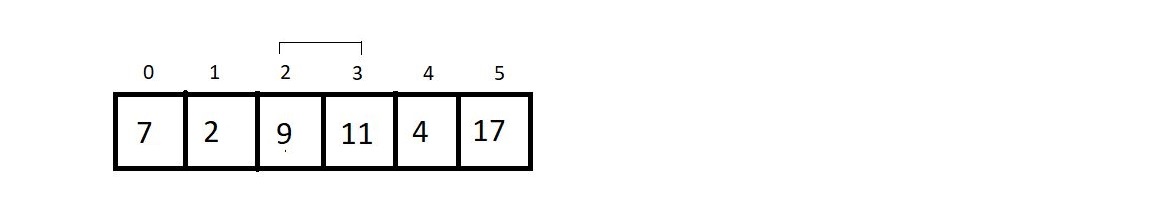
We again start from the beginning, with a reduced unsorted part of length 5. Hence the number of comparisons would be just 4.



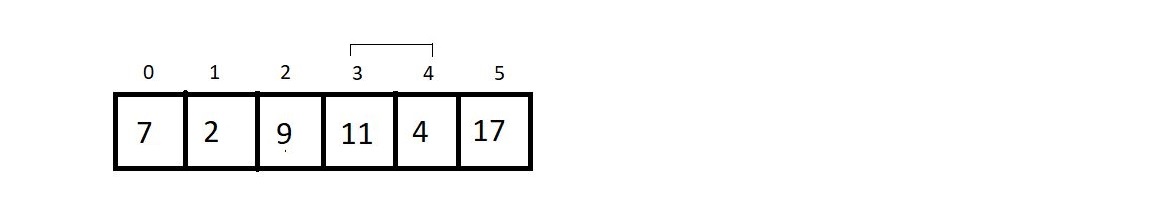
No changes to make.



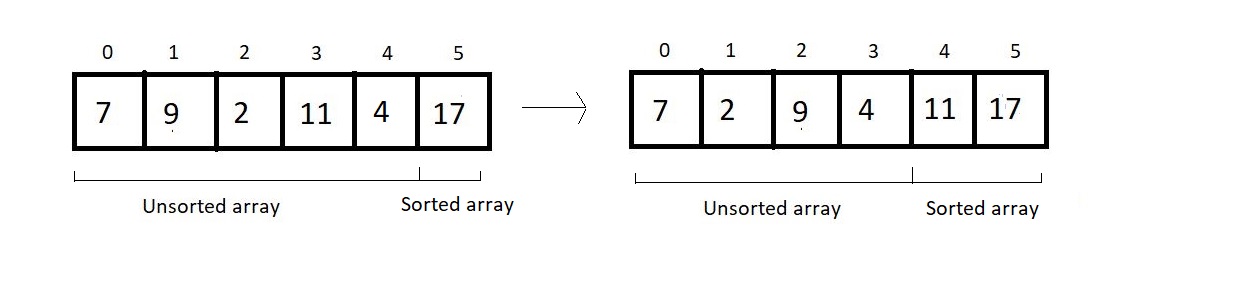
Yes, here we make a swap, since 9>2.



Since 9 < 11, we move further.

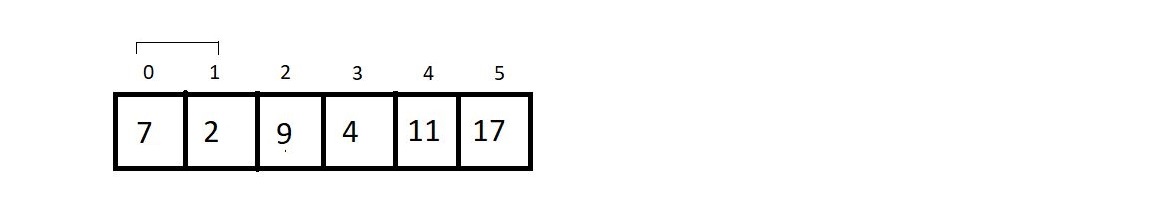


And since 11 is greater than 4, we make a swap again. And that would be it for the second pass. Let’s see how close we have reached to the sorted array.

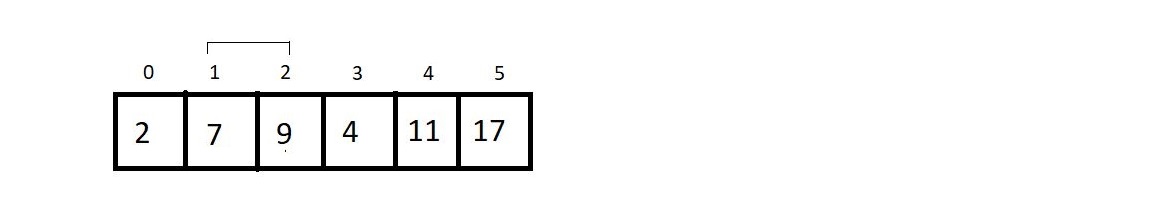


**3rd Pass:**

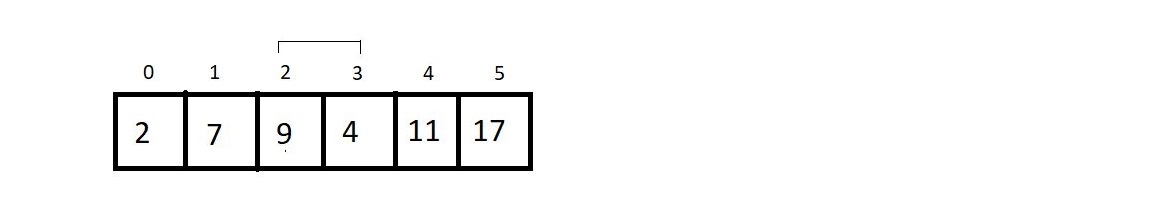
We’ll again start from the beginning, and this time our unsorted part has a length of 4; hence no. of comparisons would be 3.



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

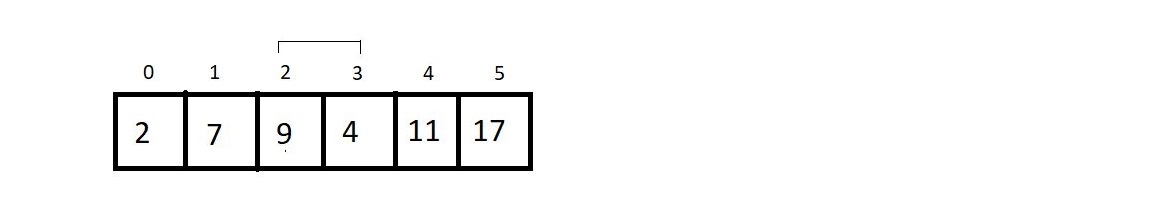


We move ahead without making any change.



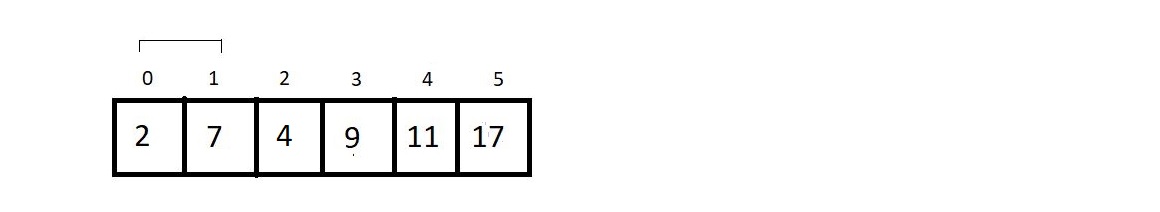
In this final comparison, we make a swap, since 9 > 4.

And that was our third pass. And the result at the end was:

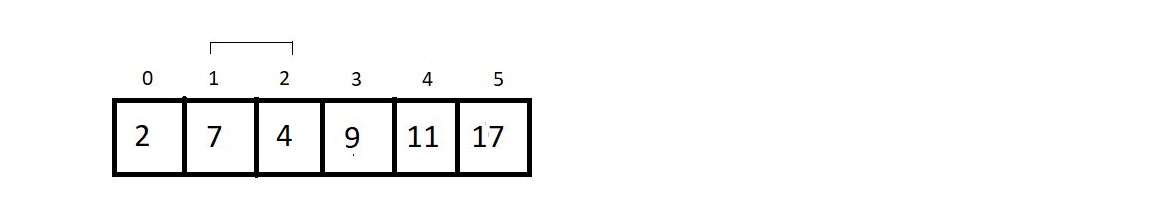


**4th Pass:**

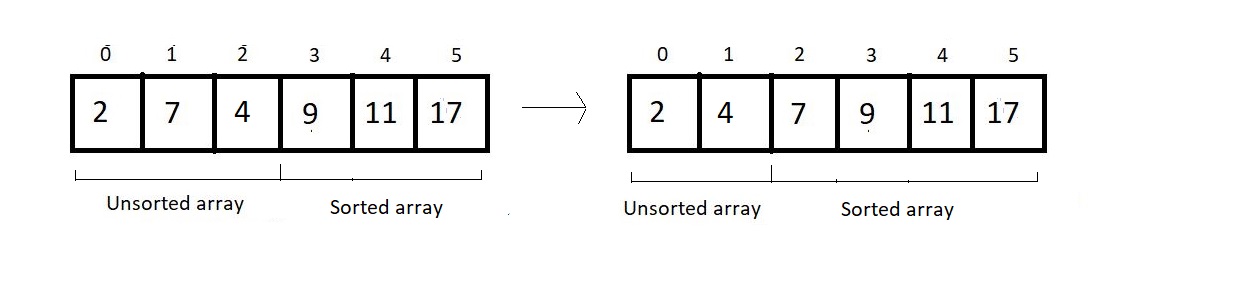
We just have the unsorted part of length 3, and that would cause just 2 comparisons. So, let’s see them.

****

No changes here.

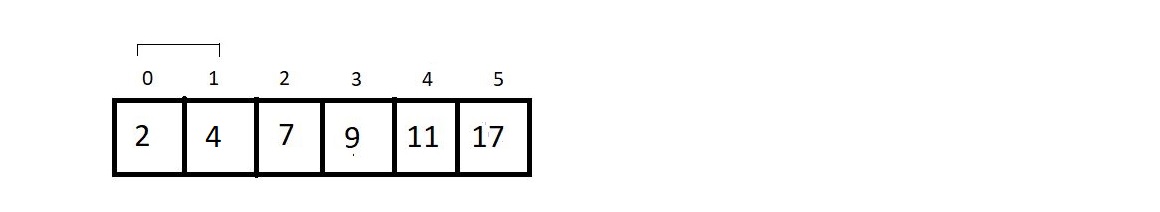


We swap their positions. And that is all in the 4th pass. The resultant array after the 4th pass is:

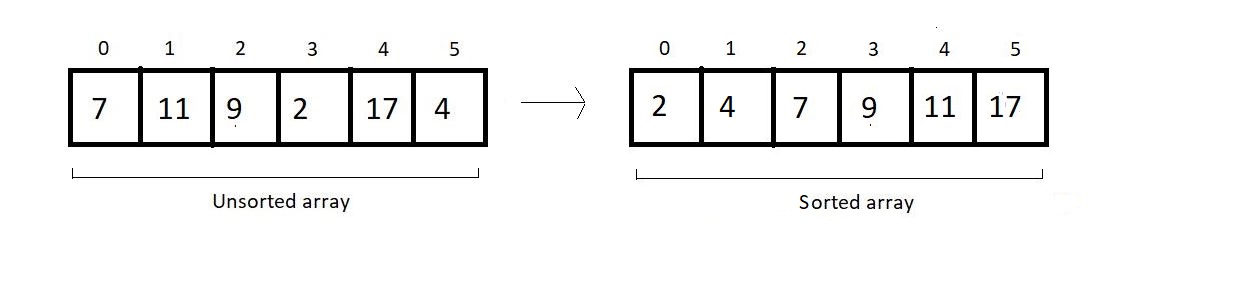


**5th (last) pass:**

We have only one comparison to make here.



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



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

Time Complexity of Bubble Sort:

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) + . . . . . + 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(n2).
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");

}

**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 ith 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 jth 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 jth 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 jth 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;

}

}

}

}

**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;

}

}

}

**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;

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;

}

**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);

**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>

**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);

**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

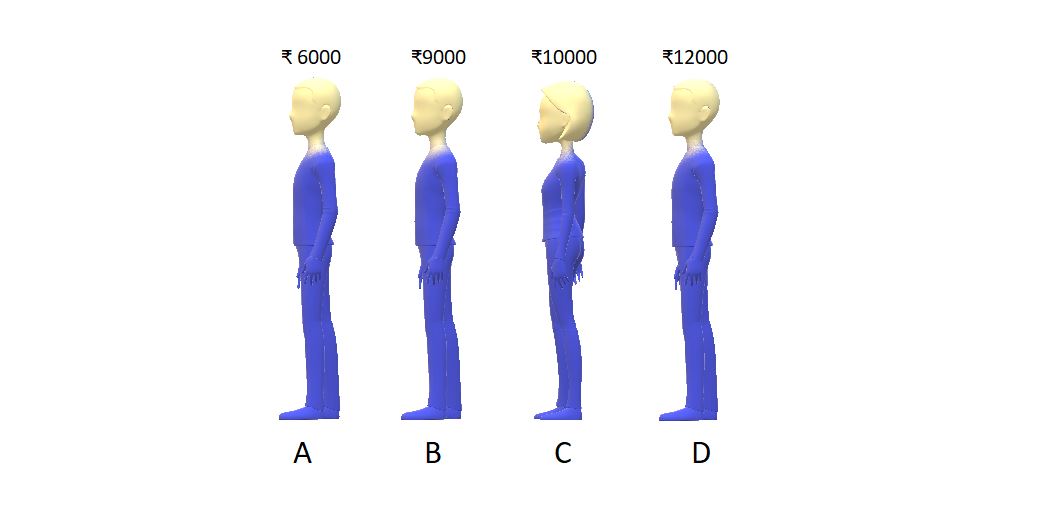
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>

**Figure 2: Output of the above program**

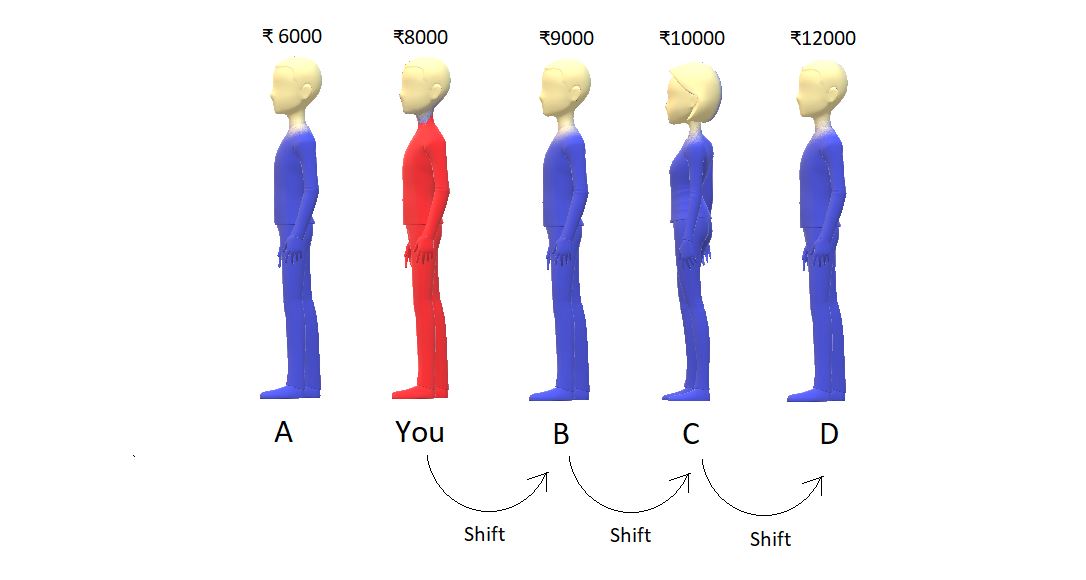
# Insertion Sort Algorithm in Hindi

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.

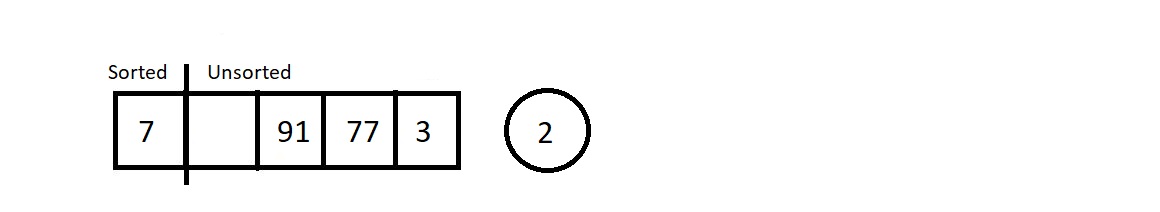
**Insert Sort Algorithm:**

Let’s just take an array, and use the insertion sort algorithm to sort its elements in increasing order.

Consider the given array below:

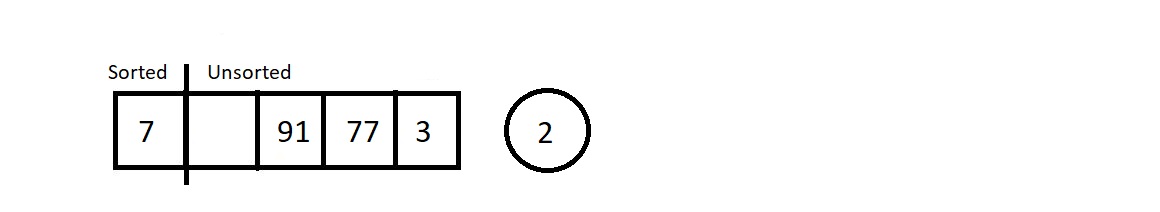


And what have we already learned? We have learned to put an arbitrary element inside a sorted array, using the insertion method we saw above. **And an array of a single element is always sorted.**So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

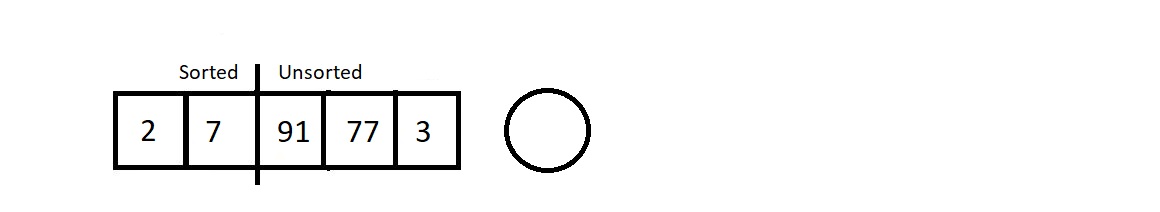


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let’s call each of these insertions and the traversal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

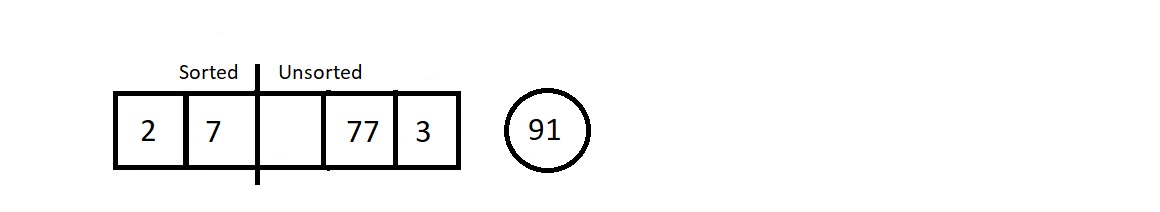
So, let’s start with pass 1, which is to insert 2 in the sorted array of length 1.



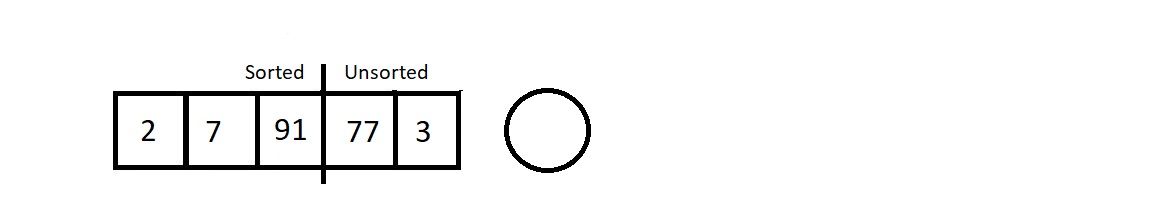
So, we plucked the first element from the unsorted part. Let’s insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



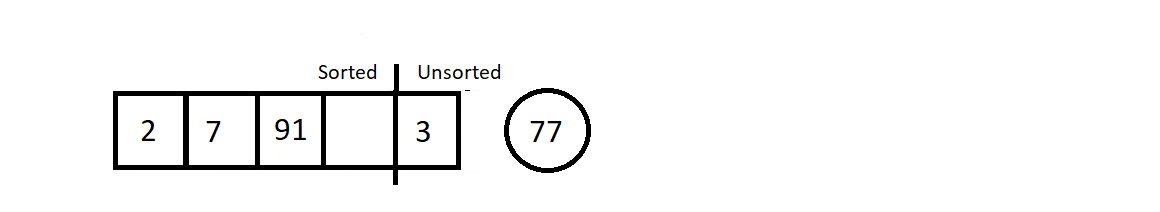
Let’s proceed to the next pass.



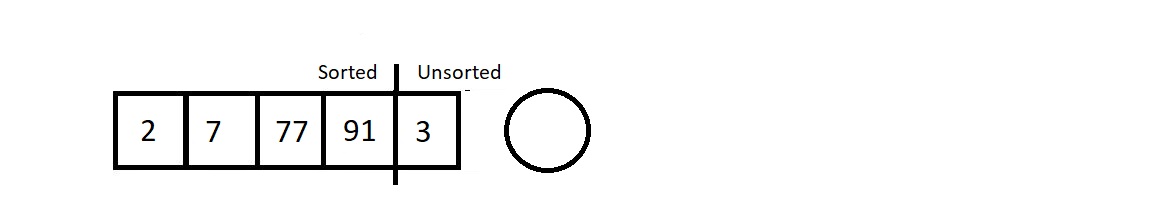
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



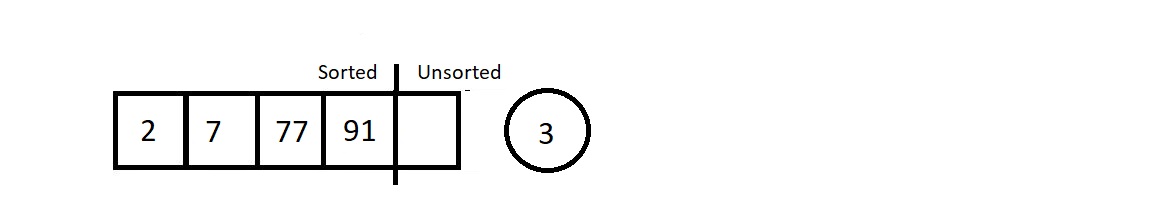
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let’s proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



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+ . . . + (n-1) which is n(n-1)/2, which ultimately is **O(n2)**.

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.

**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(n2) 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(n2) 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");

}

**Code Snippet 1: Creating the *printArray*function**

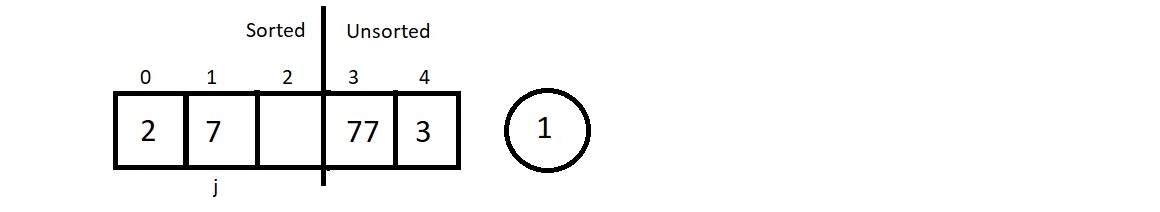
5. Create a void function *insertionSort*and pass the address of the array and its length as its parameters. Now, create a *for*loop which would track the number of passes. If you recall, to sort an array of length 5 using insertion sort algorithm, we made a total of 4 passes, which is obviously, 5-1. 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.

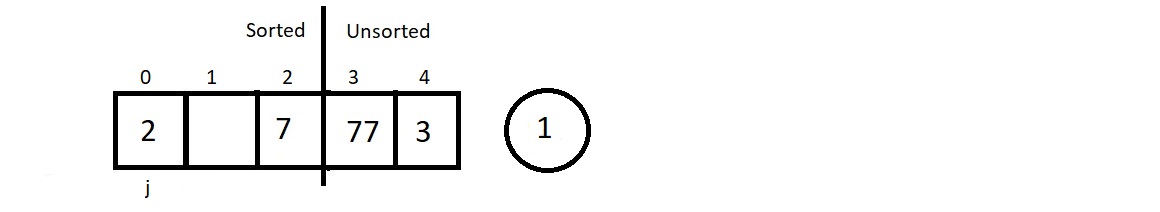
Create another index variable j, which would be used to iterate through the sorted subarray, and to find a perfect position for the *key*. The index variable *j* holds the value *i-1.*

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.

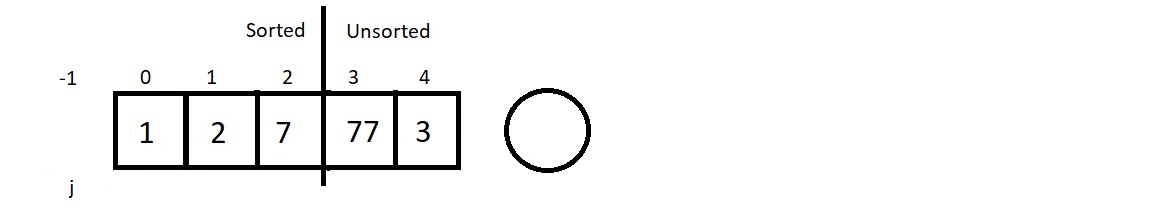
To understand the functioning of the above while loop, follow the illustrations below:



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 jth 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 jth 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;

}

}

**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;

}

**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);

**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>

**Figure 1: Output of the above program**

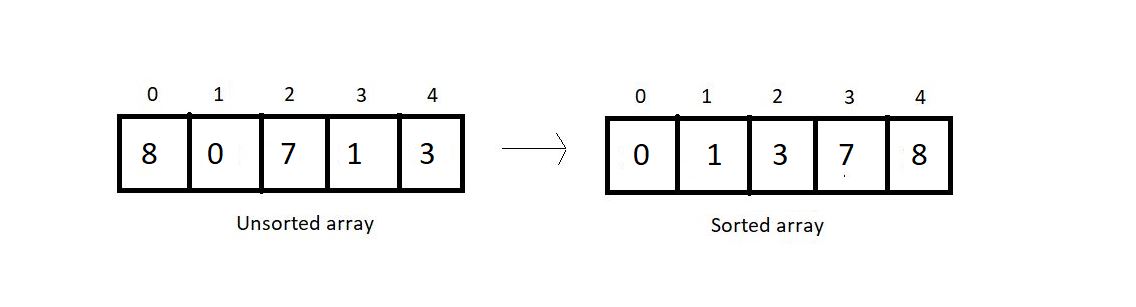
So, our array got sorted. Follow the dry run of the sorting process in the source commented. I have shown each of the 5 passes, and all the comparisons, and the shifting we did.

And, this was all about the insertion sort algorithm.

**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.



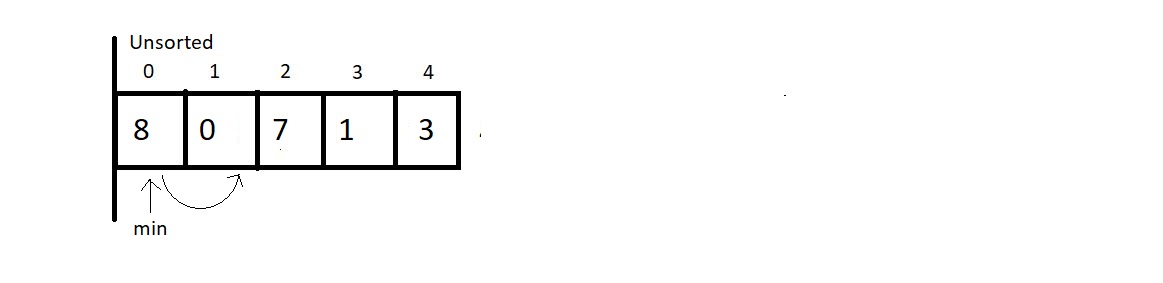
In selection sort, at each pass, we make sure that the smallest element of the current unsorted subarray reaches its final position. And this is pursued by finding the smallest element in the unsorted subarray and replacing it at the end with the element at the first index of the unsorted subarray. This 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.

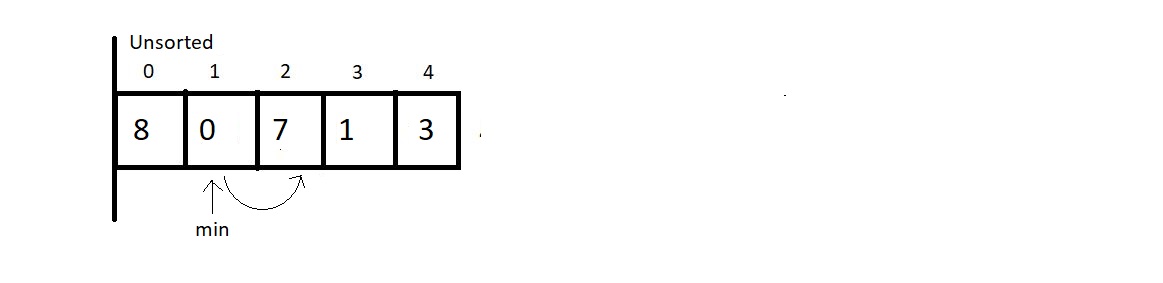
Since our array is of length 5, we will make 4 passes. You must have realized by now the reason why it would take just 4 passes.

**1st Pass:**

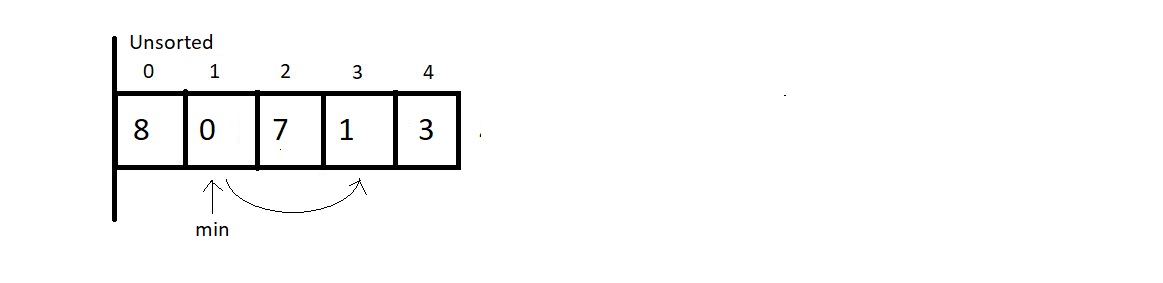
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the *min*index. Now, we’ll have to check among the remaining 4 elements if there is still a lesser element than the first one.



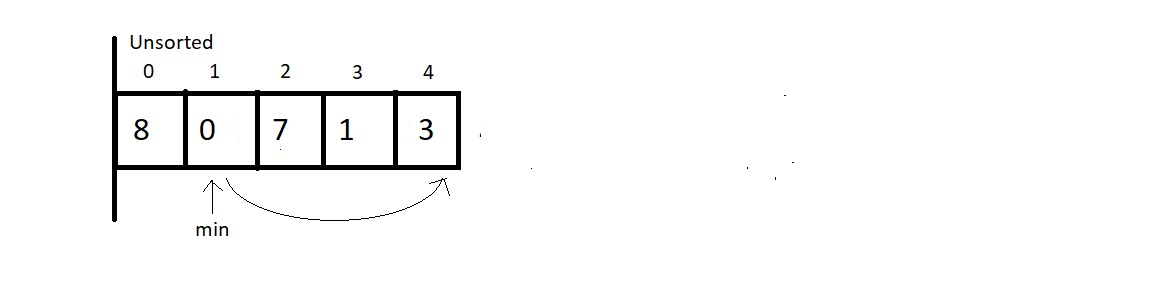
And when we compared the element at *min*index with the element at index 1, we found that 0 is less than 8 and hence we update our *min* index to 1.



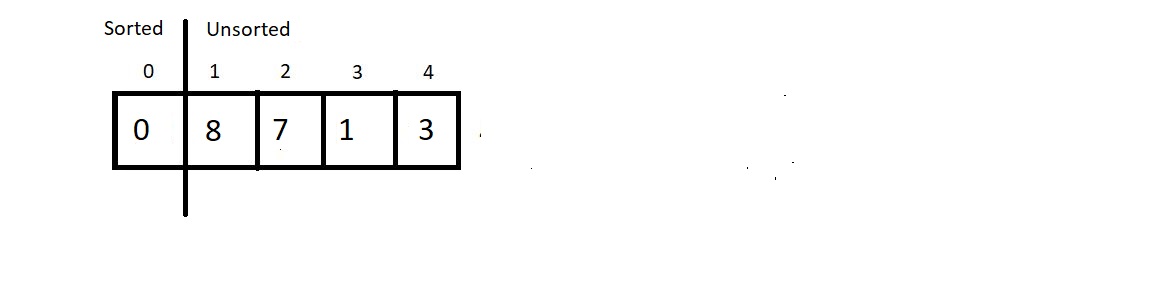
And now we keep checking with the updated *min.*Since 7 is not less than 0, we move ahead.



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.

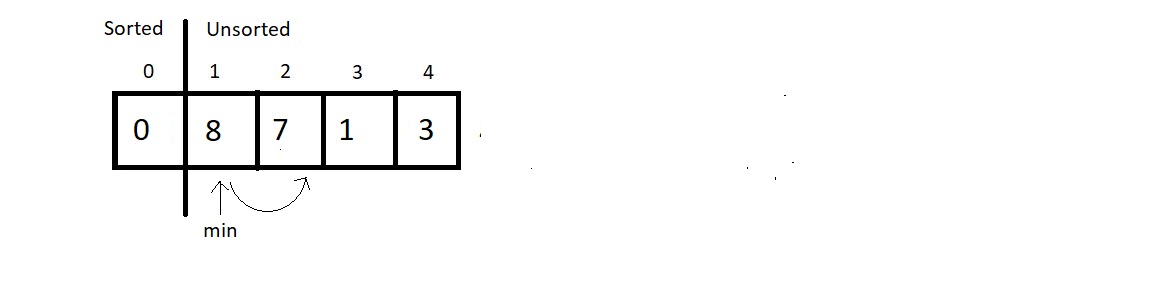


And now we compared the element at the *min*index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at *0th*index with the element at the *min*index. And this gives us our first sorted subarray of size 1. And this is where our first pass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

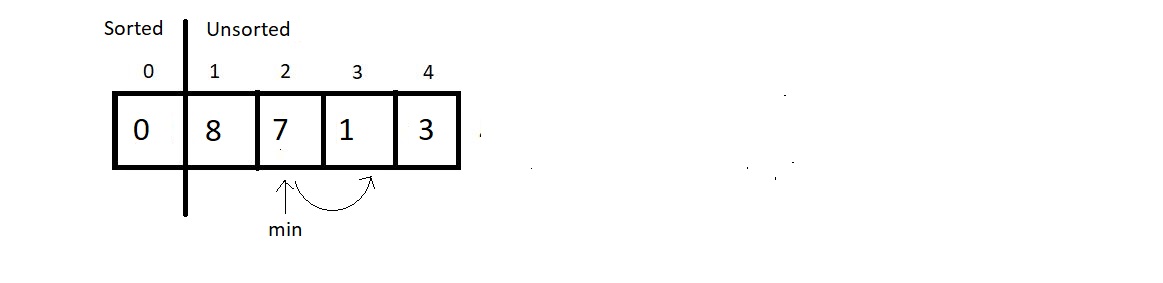


**2nd Pass:**

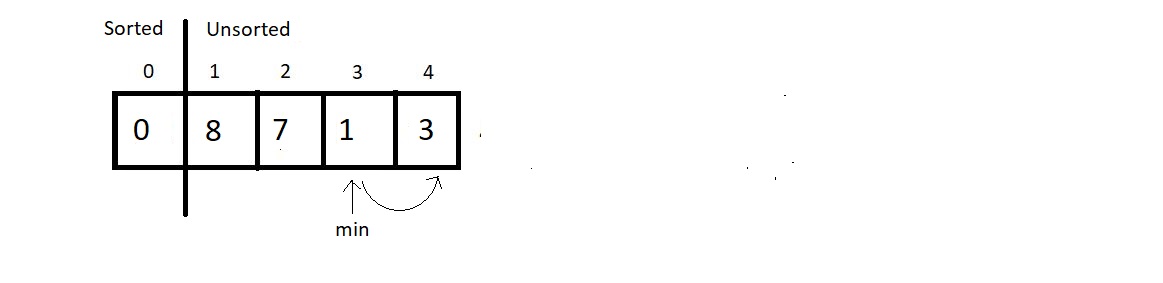
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the *min* index and start iterating to the right for finding the minimum element.



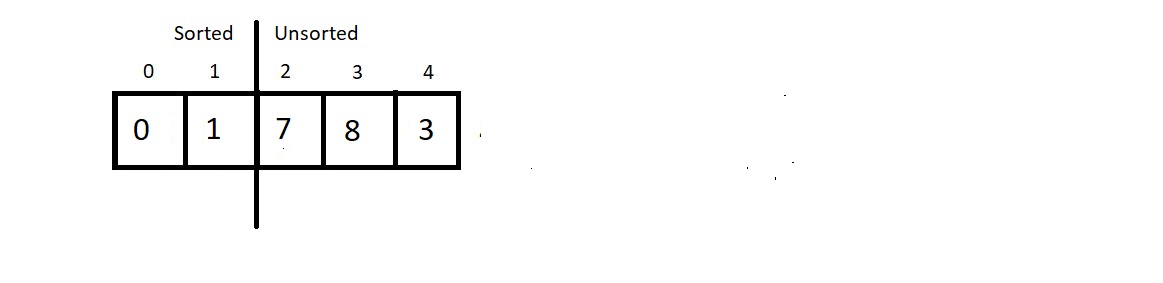
Since 7 is less than 8, we update our *min*index with 2. And move further.



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.

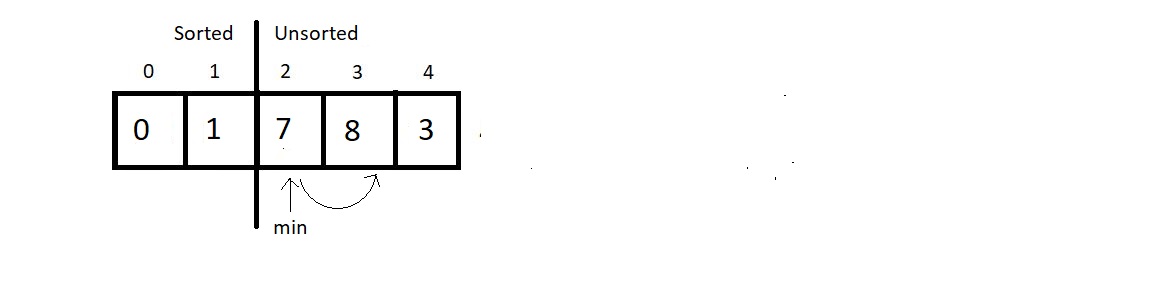


And since 3 is greater than 1, we don’t make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at *min*index. And that would be it for the second pass. Let’s see how close we have reached to the sorted array.

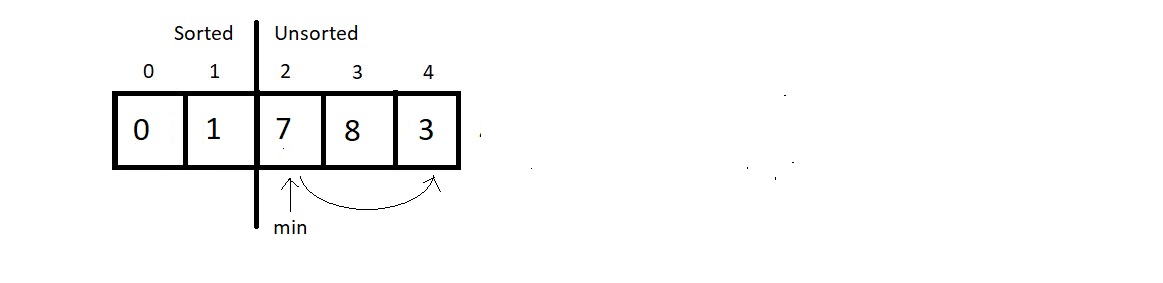


**3rd Pass:**

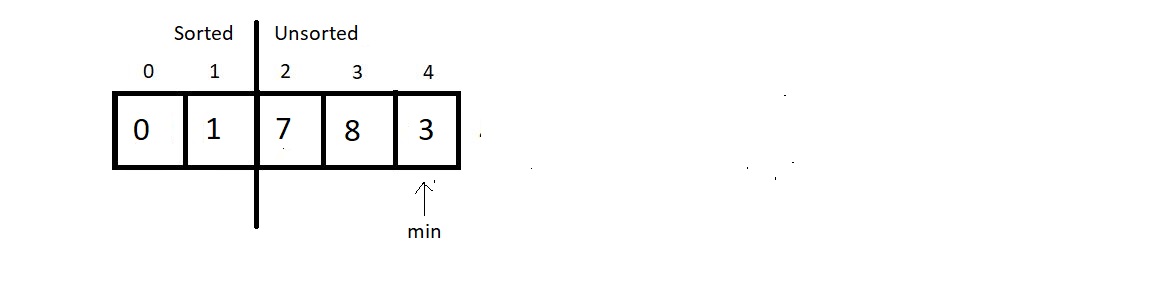
We’ll again start from the beginning of the unsorted subarray which is from the index 2, and make the *min*index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.



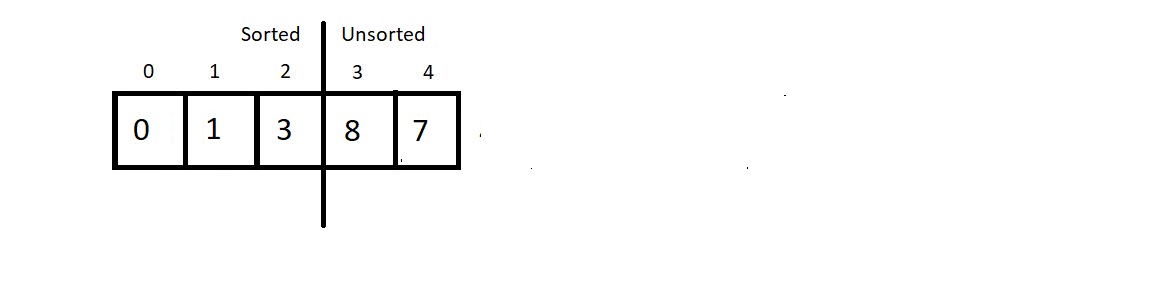
Since 8 is greater than 7, we would make no change, but move ahead.



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.

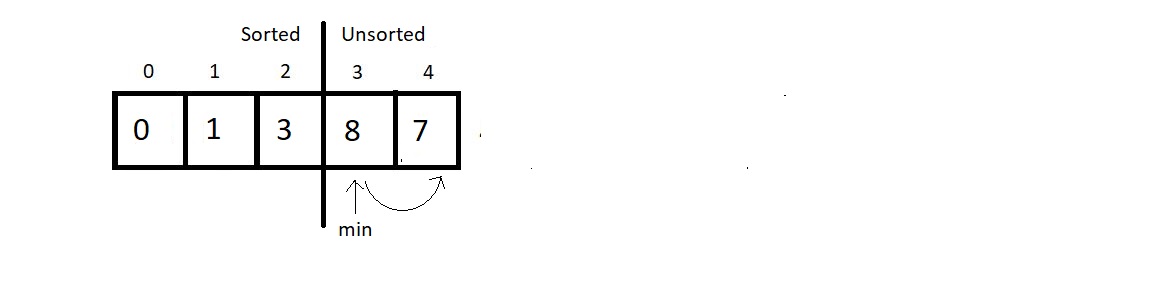


And since that was the last comparison of the third pass, we make a swap of the indices 2 and *min.*And the result at the end would be:

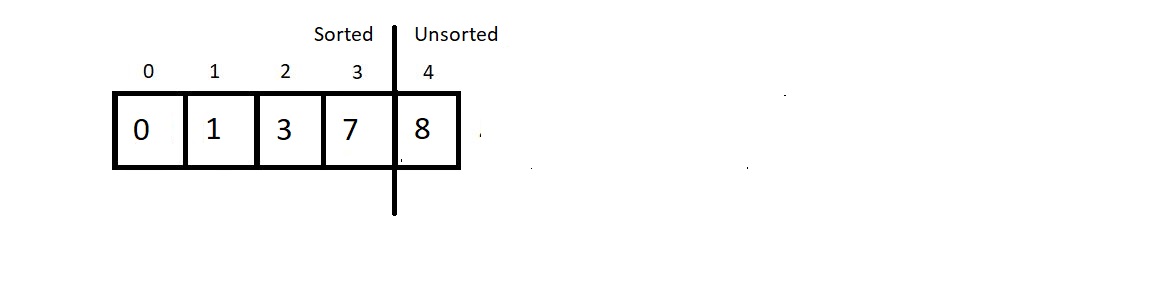


**4th Pass:**

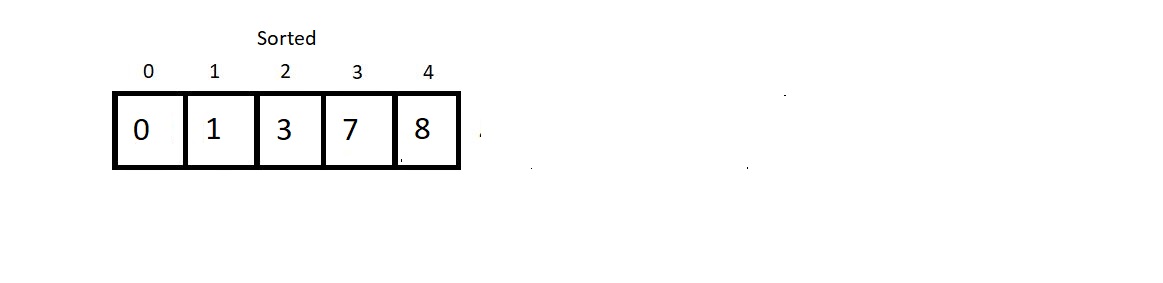
We now have the sorted subarray of length 3, hence the new *min*would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let’s see that.



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:

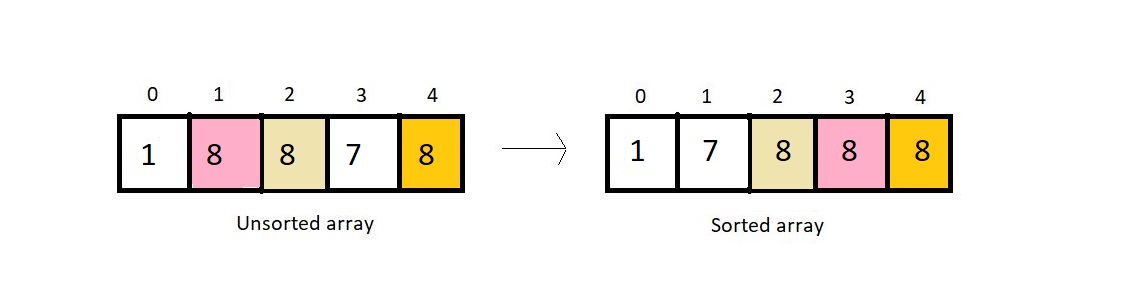
1. **Time Complexity of Selection Sort:**

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) + . . . . . + 1 comparisons.

Sum from 1 to n-1, we get , and hence the time complexity of the algorithm would be **O(n2)**.

1. 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:



1. It is not a recursive algorithm, since we didn’t use recursion here.
2. 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.
3. 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(n2) 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");

}

**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 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 thej 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;

}

}

// Swap A[i] and A[indexOfMin]

temp = A[i];

A[i] = A[indexOfMin];

A[indexOfMin] = temp;

}

}

**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;

}

**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);

**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>

**Figure 1: Output of the above program**

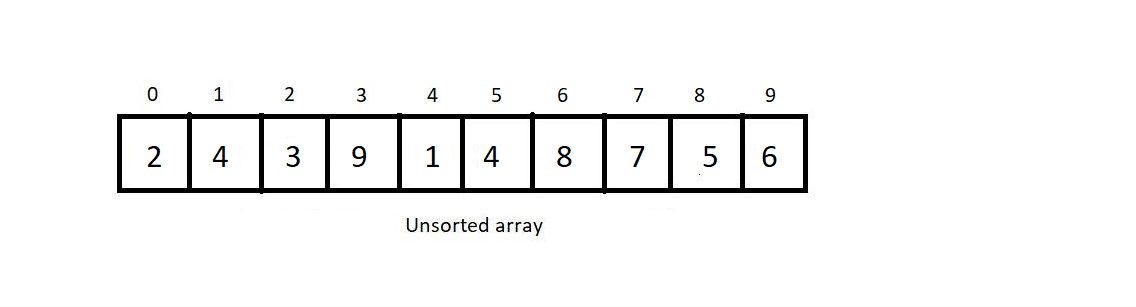
Visualize each pass individually using the dry run method. This would give you a lot more confidence. The dry run of the above example is there in the source code itself.

**QuickSort Algorithm in Hindi (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.



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.

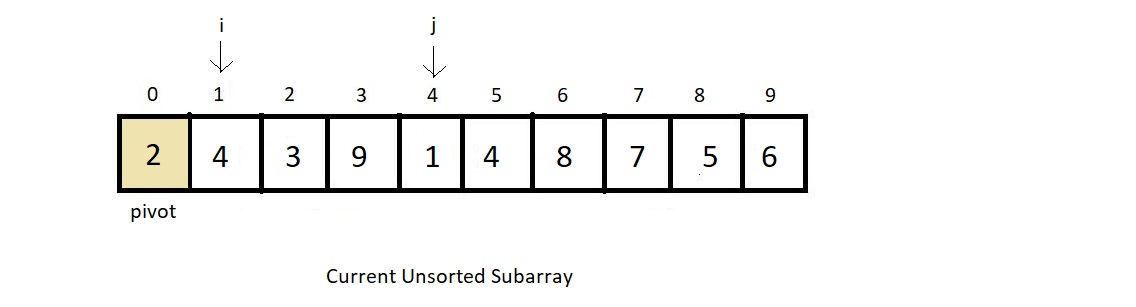
Now let's look at how the array we received at the beginning gets sorted using partitioning and divide and conquer recursively for smaller subarrays.

Firstly, the whole array is unsorted, and hence we apply quicksort on the whole array.

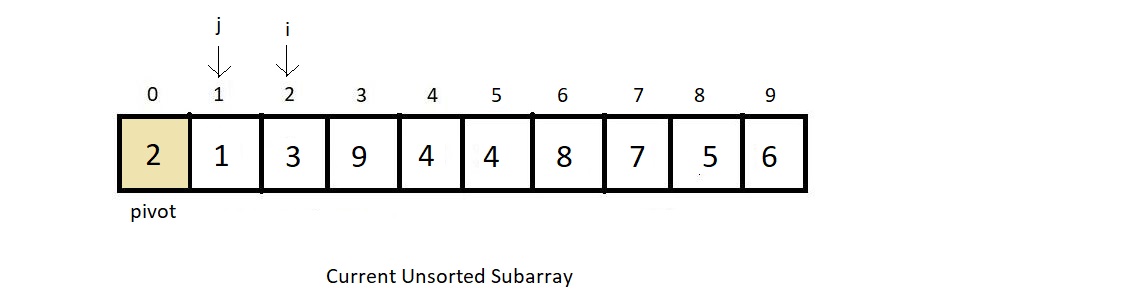
Now, we apply a partition in this array. Applying partition asks you to follow all the above steps we discussed.



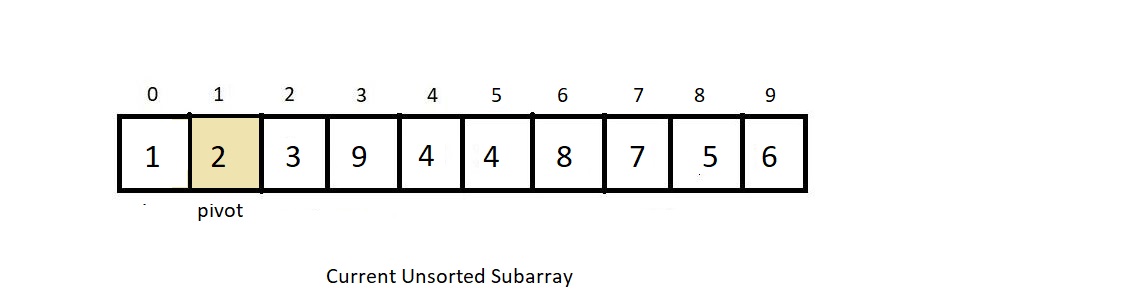
Keep increasing i until we reach an element greater than the pivot, and keep decreasing j until we reach an element smaller or equal to the pivot.



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

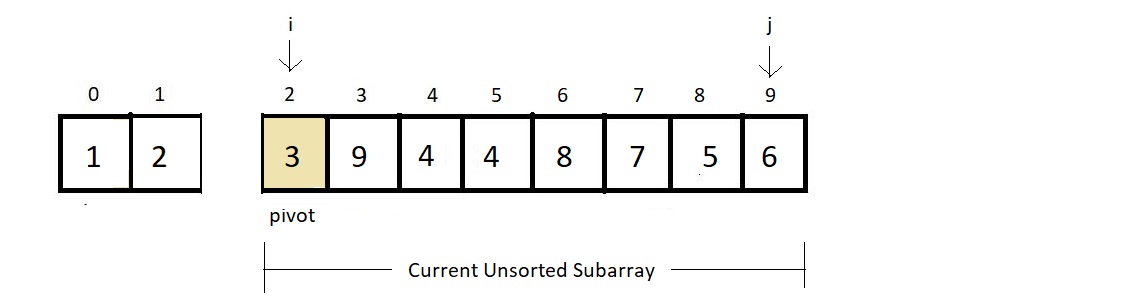


As j crossed i while searching, we followed the final step of swapping the pivot element and the element at j.

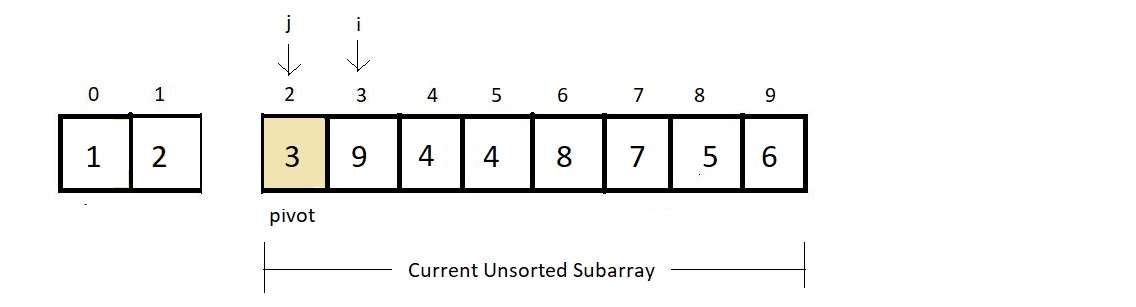


And this would be the final position of the current pivot even in our sorted array. As you can see, all the elements smaller than 2 are on the left, and the rest greater than 2 are on the right. Here comes the role of divide and conquer. We separate our focus from the whole array to just the subarrays, 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.

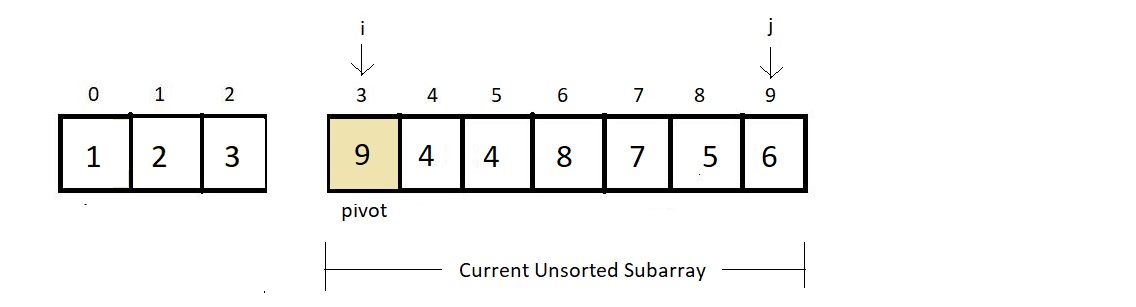
Now since the first subarray has just a single element, we consider it sorted. Let’s now sort the second subarray. Follow all the partition steps from the beginning.



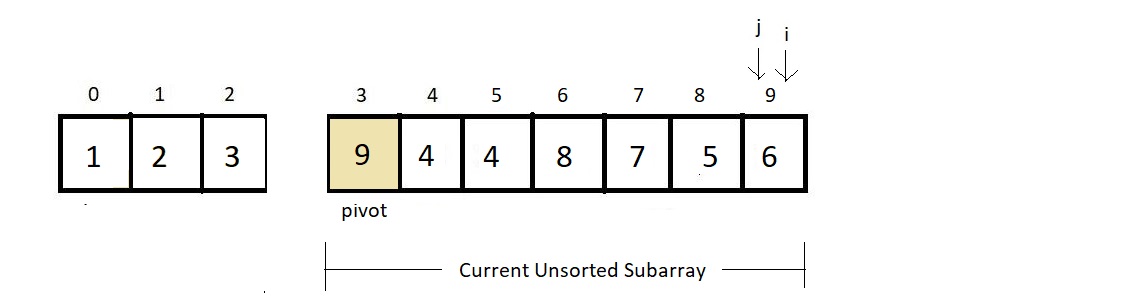
Now, our new pivot is the element at index 2. And *i* and *j* are 2 and 9, respectively, marking the start and the end of the subarray. Follow steps 3 and 4.



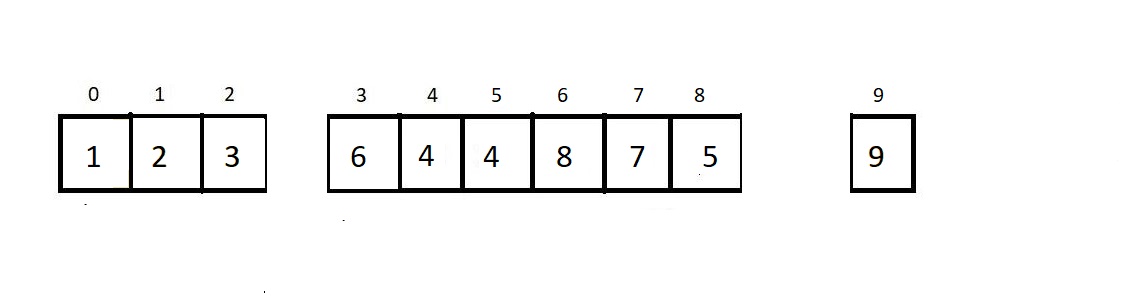
And since there were no elements smaller than 3, j crosses i in the very first iteration. This means 3 was already at its sorted index. And there are no elements to its left; the only unsorted subarray is {9, 4, 4, 8, 7, 5, 6}. And our new situation becomes:



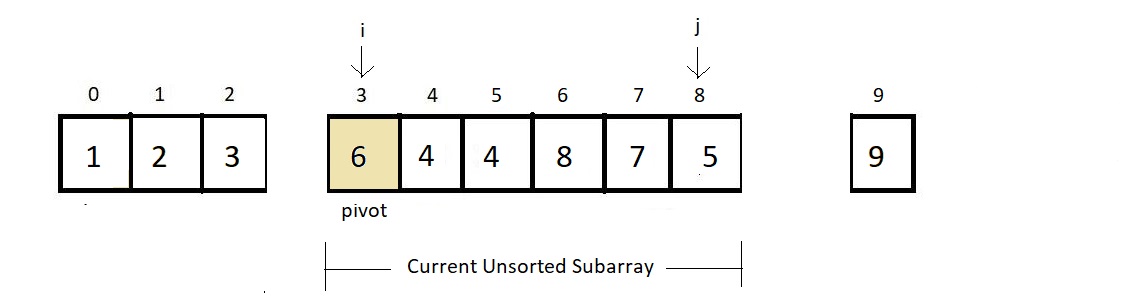
Repeating steps 3 and 4.



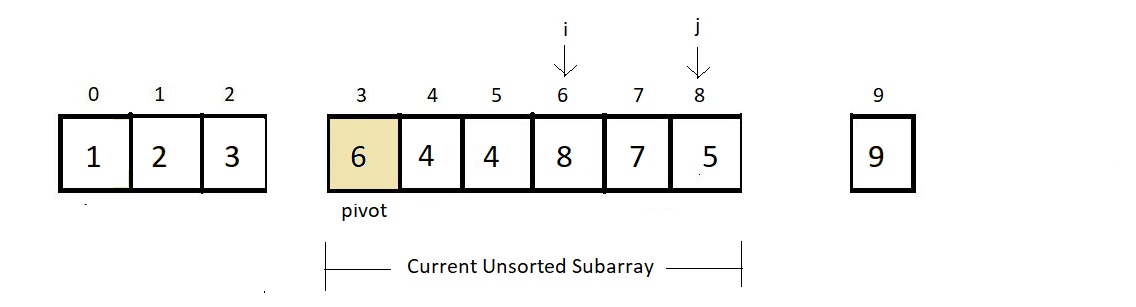
We found that there was no element greater than 9, and hence *i* reached the last. And 6 was the first element j found to be smaller than 9, and they collided. And this is where we do step 7. Swap the pivot element and the element at index j.



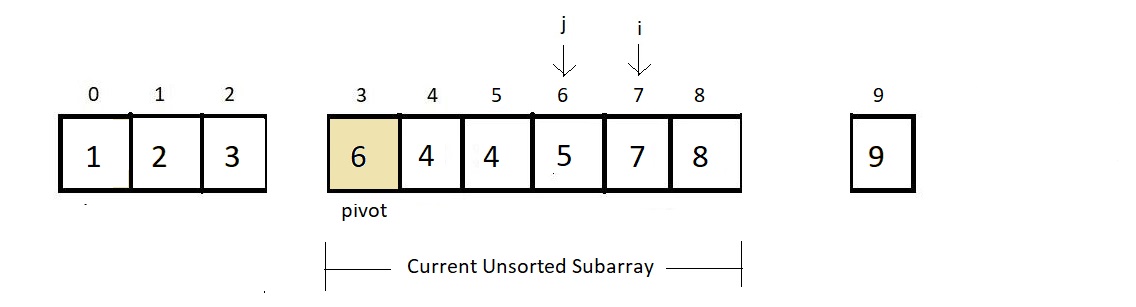
And since there are no elements to the right of element 9, we just have one sorted subarray {6, 4, 4, 8, 7, 5}. Let’s call a partition on this as well.



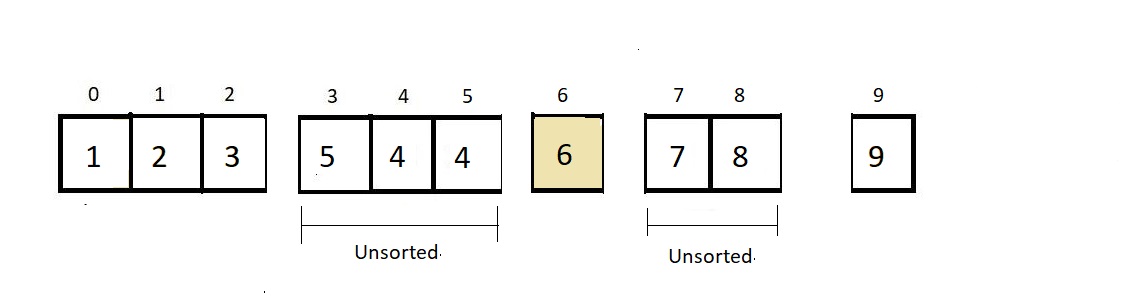
Repeat steps 3 and 4.



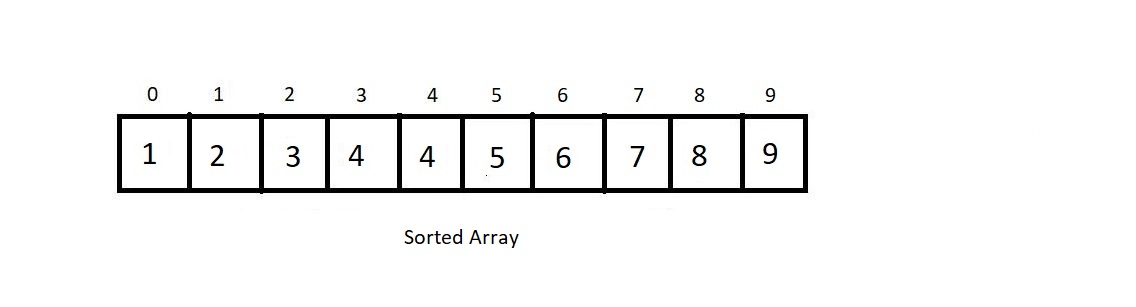
And since the condition j<=i has not been met yet, we just swap the elements at index i and j and continue our search.



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.



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");

}

**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.

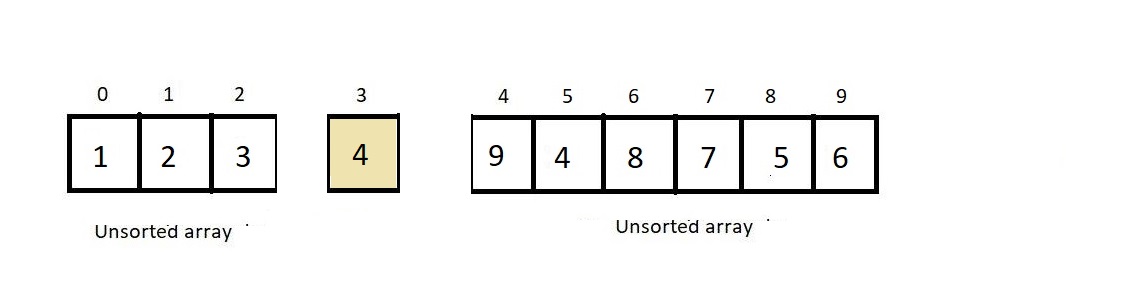
3. Define an integer variable for storing the size/length of the array.

**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

}

}

**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;

}

**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;

}

**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);

**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>

**Figure 1: Output of the above program**

**Analysis of QuickSort Sorting Algorithm**

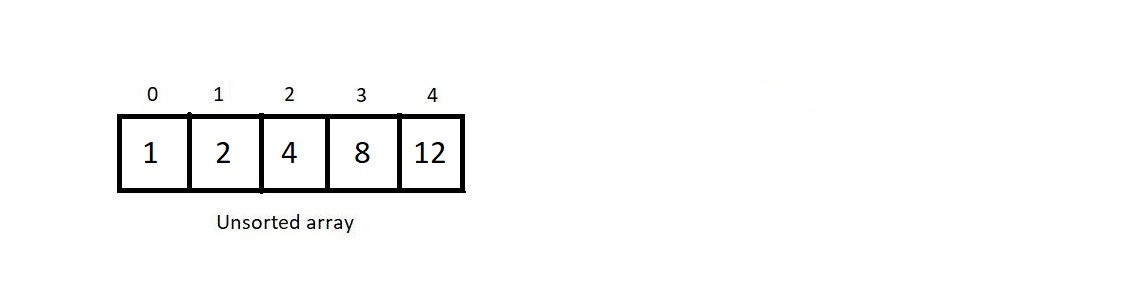
In the last video, we learned to use the quick sort algorithm. We saw how we used the methods of divide and conquer and partitioning to achieve sorting. Later, in the video, we even saw the program to implement all these methods to sort an array using the quick sort algorithm. Today, we’ll see the analysis of the quicksort algorithm on all criteria we defined earlier.

1. **Time Complexity:**

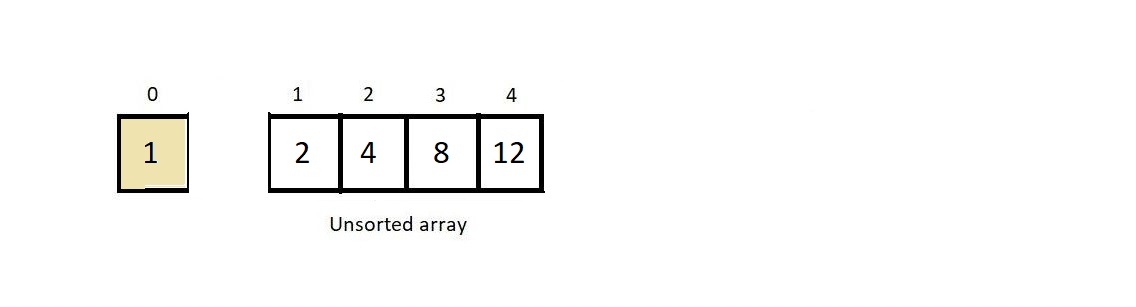
Let’s start with the runtime complexity of the algorithm:

**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.



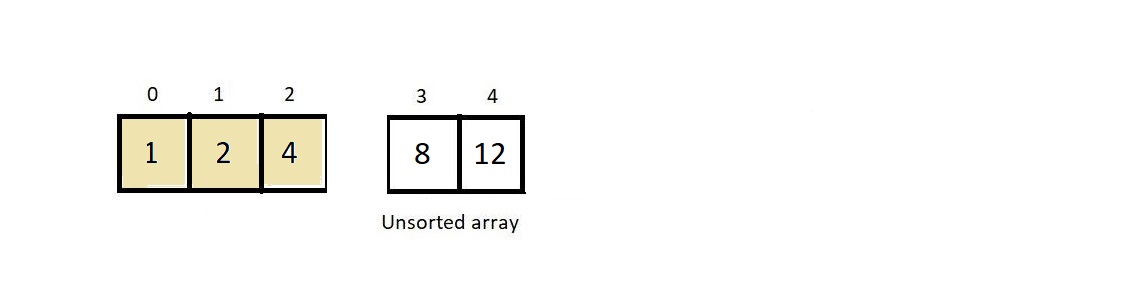
In the first step, you choose 1 as the pivot and apply a partition on the whole array. Since 1 is already at its correct position, we apply quicksort on the rest of the subarrays.



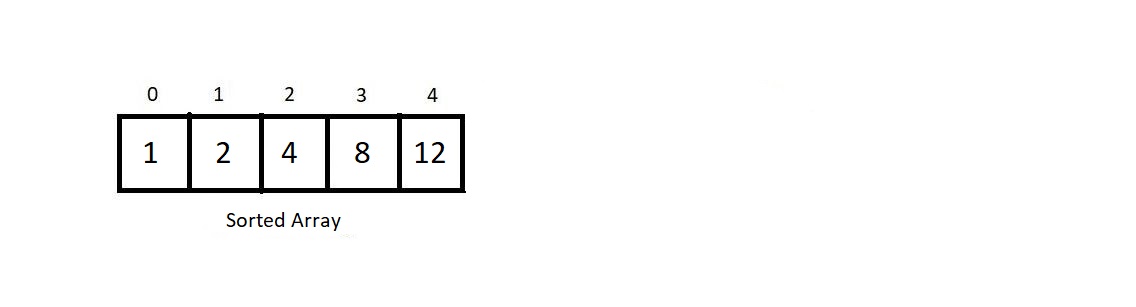
Next, the pivot is element 2, and when applied partition, we found that there is no element less than 2 in the subarray; hence, the pivot remains there itself. We further apply quicksort on the only subarray that is to the right.



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



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

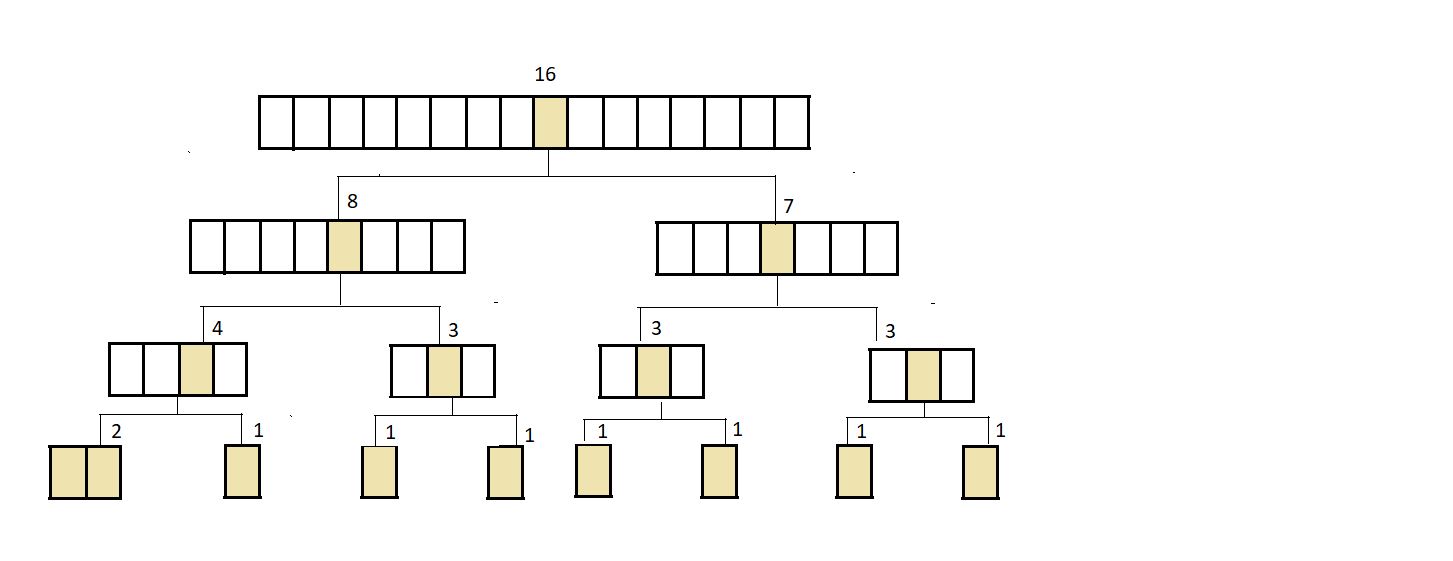


So, if you calculate carefully, for an array of size 5, we had to partition the subarray 4 times. That is, for an array of size n, there would be (n-1) partitions. Now, during each partition, long story short, we made our two index variables, *i and j*run from either direction towards each other until 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(n2)**. This is our worst-case complexity.

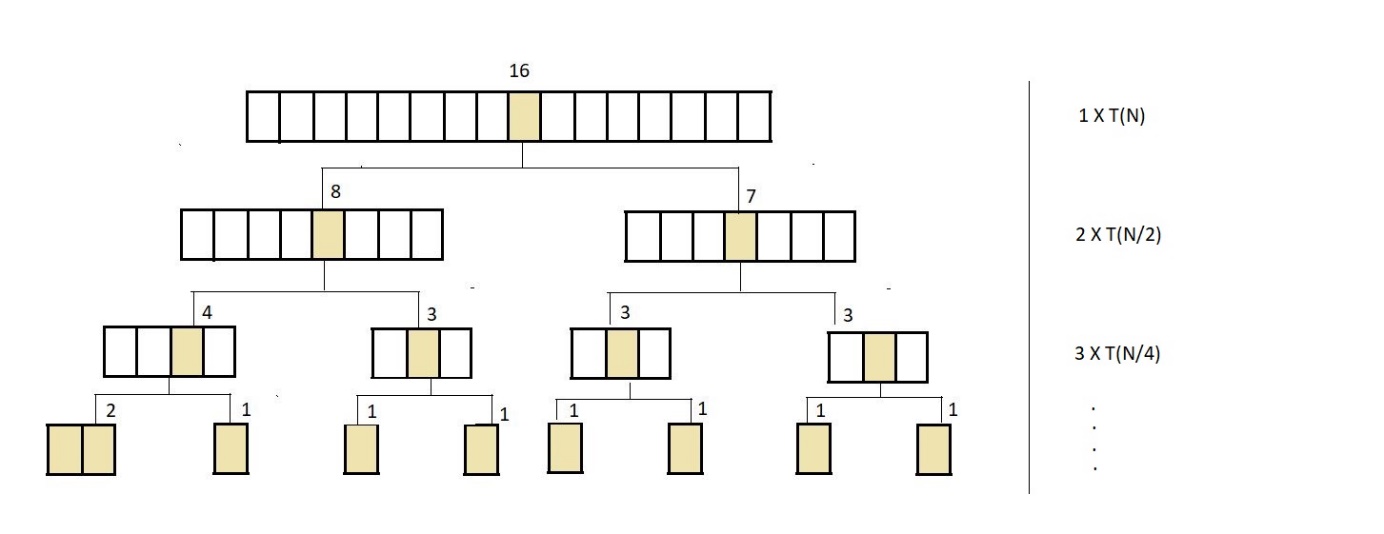
**Best Case:**

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



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

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



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

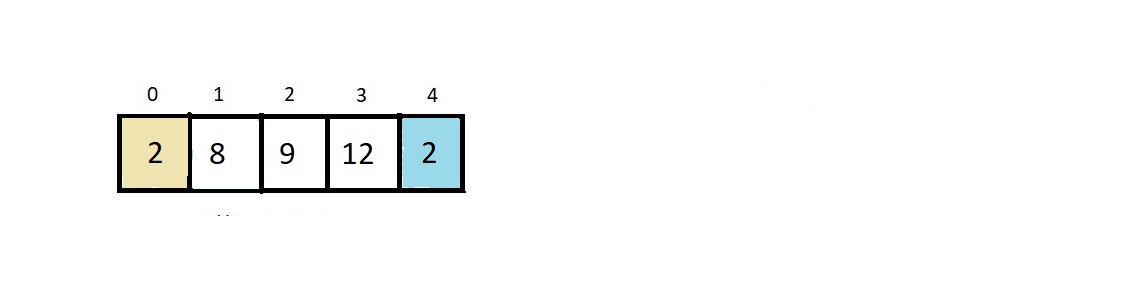
**Total time = 1(n) + 2(n/2) +4(n/4) + ..........+ until the height of the tree(h) Total time = n\*h**

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

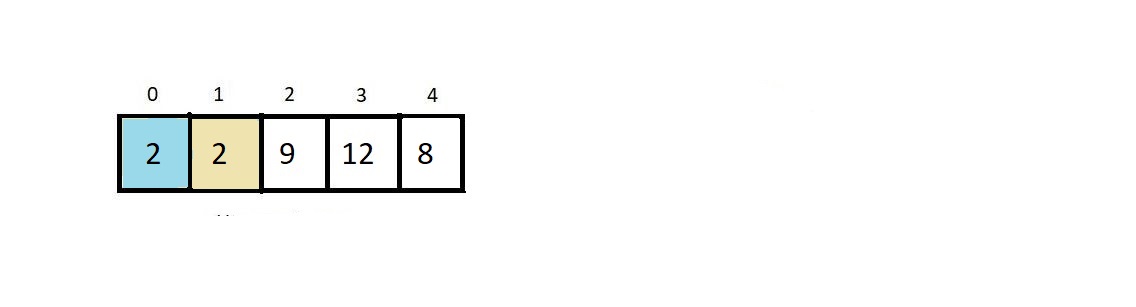
**Note: The average time complexity remains O(nlogn).**Calculations have been avoided here due to their complexity.

1. **Stability:**

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



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



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

1. 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.
2. 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.

So, that was all we had to discuss regarding the QuickSort algorithm.