**Data Structure and Algorithms**

**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.

**Database** – Collection of information in permanent storage for faster retrieval and updation.

Examples are MySql, MongoDB, etc.

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

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

**Memory Layout of C Programs:**

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

* The stack holds the memory occupied by functions. It stores the activation records of the functions used in the program. And erases them as they get executed.
* The heap contains the data which is requested by the program as dynamic memory using pointers.
* Initialized and uninitialized data segments hold initialized and uninitialized global variables, respectively.

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



**Time Complexity and Big O Notation**

**What is Time Complexity?**

Time Complexity is the study of the efficiency of algorithms.

Consider two developers Yatharth and Riya, 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 Riya’s Algo** | **Time Taken By Yatharth’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, Riya’s algorithm worked well with smaller inputs; however, as we increase the number of elements, Yatharth’s algorithm performs much better.

**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:



**Asymptotic Notations: Big O, Big Omega and Big Theta**

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

Big oh notation ( O )

Big omega notation ( Ω )

Big theta notation ( θ ) – Widely used one

**Big oh notation ( O ):**

Big oh notation is used to describe an asymptotic upper bound.

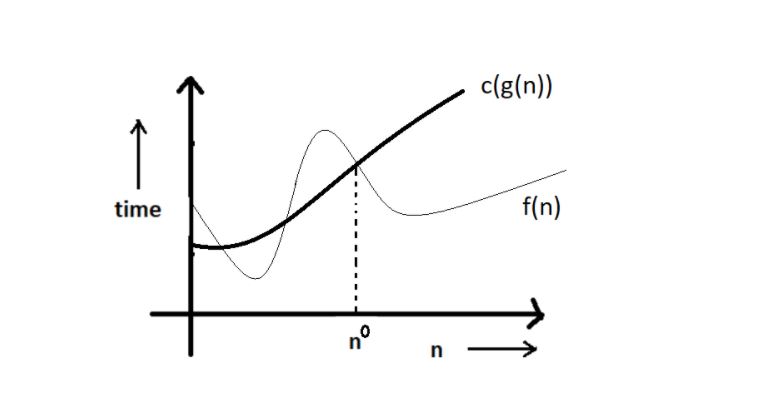
Mathematically, if f(n) describes the running time of an algorithm; f(n) is O(g(n)) if and only if there exist positive constants c and n° such that:

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.



Merging both the equations, we get:

0 ≤ f(n) ≤ c g(n) for all n ≥ n°.

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.

**Best Case, Worst Case and Average Case Analysis of an Algorithm**

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.

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

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

**Analyzing Algorithm 1: (Linear Search)**

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

**Best case complexity = O(1)**

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

**Worst-case complexity = O(n)**

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

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

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

**Best case complexity = O(1)**

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

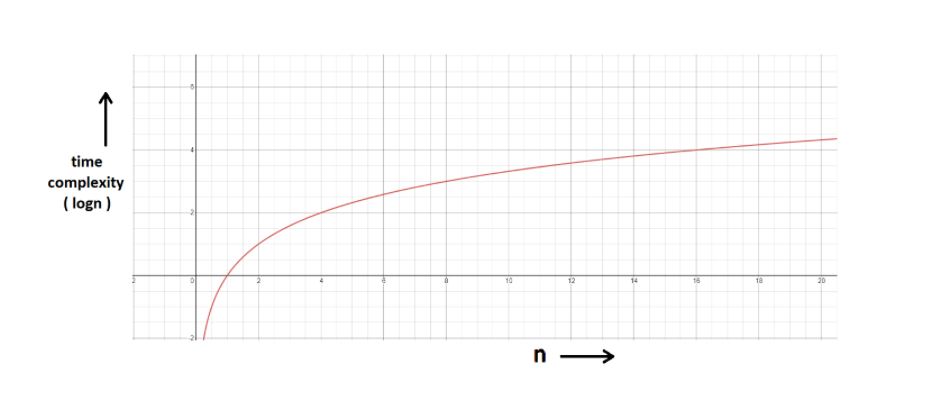
**What is log(n)?**

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).

**How to Calculate Time Complexity of an Algorithm**

**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.

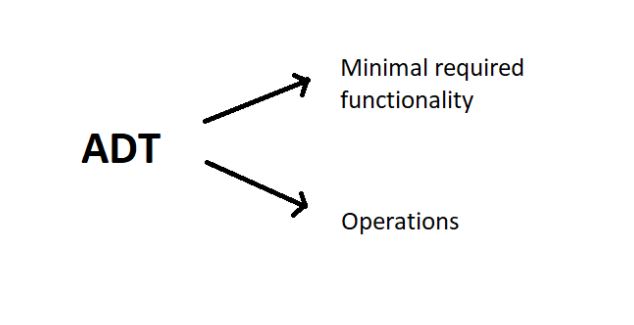
**Consider all variables which are provided as input:**

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

**Arrays and Abstract Data Type in Data Structure**

**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 )

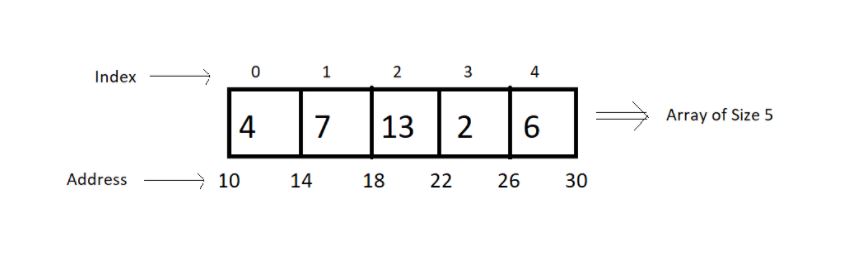
Append (x)

**Static and Dynamic Arrays:**

Static arrays – Size cannot be changed

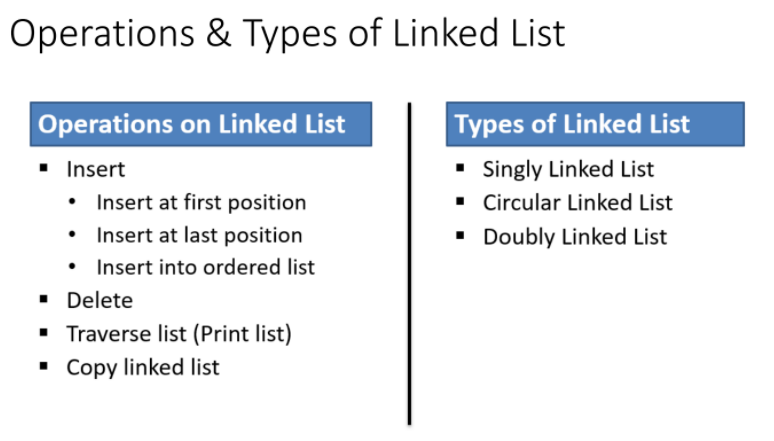
Dynamic arrays – Size can be changed

Memory Representations of Array:



* Elements in an array are stored in contiguous memory locations.
* Elements in an array can be accessed using the base address in constant time → O (1).
* Although changing the size of an array is not possible, one can always reallocate it to some bigger memory location. Therefore resizing in an array is a costly operation.
* Define Linked List. How is it differ from Array Data Structure?  
    
  A linked list is a **non-sequential collection** of data items.

A Linked List is a Linear Data Structure.

* + A LL is **the collection of nodes** that are randomly stored.
  + Each node is divided into **two parts**, the first part represents the **data** of the element and the second part contains the **address of next node**.
  + The last node of the list stored null value as the address.
  + It is possible for a list to have no nodes at all, such a list is called empty list.  
      
    

### **Differences between Array and Linked list**

|  |  |
| --- | --- |
| **Array** | **Linked list** |
| An array is a **collection of elements** of a similar data type. | A linked list is a **collection of objects** known as a **node** where node consists of two parts, i.e., data and address. |
| Array elements store in a **contiguous memory** location. | Linked list elements can be **stored anywhere in the memory** or randomly stored. |
| Array works with a **static memory.** Here static memory means that the **memory size is fixed and cannot be changed at the run time.** | The Linked list works with **dynamic memory**. Here, dynamic memory means that the **memory size can be changed at the run time** according to our requirements. |
| Array elements are **independent of each other.** | Linked list elements are **dependent on each other**. As each node contains the address of the next node so to access the next node, we need to access its previous node. |
| Array **takes more time** while performing any operation like **insertion, deletion**, etc. | Linked list **takes less time** while performing any operation like **insertion, deletion**, etc. |
| **Accessing any element** in an array is **faster** as the element in an array can be directly accessed through the index. | **Accessing an element** in a linked list is **slower** as it starts traversing from the first element of the linked list. |
| In the case of an array, **memory is allocated at compile-time.** | In the case of a linked list, **memory is allocated at run time.** |
| Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused. | Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement. |

**What is the limitation of Simple Queue? Which are the ways to overcome limitations of Simple Queue? Explain with suitable example**

The main limitation of queues is one of the basic operations of deleting an element from it is cumbersome.

When we add an element in Queue, the rear pointer is increased by 1 whereas, when we remove an element front pointer is increased by 1.

But an array implementation of queue this may cause the problem as follows:  
  
Consider operations performed on a Queue (with SIZE = 5) as follows:

1. Initially empty Queue is there so, front = 0 and rear = -1

2. When we add 5 elements to queue, the state of the queue becomes as follows with front = 0 and rear = 4

10

20

30

40

50

3. Now suppose we delete 2 elements from Queue then, the state of the Queue becomes as follows, with front = 2 and rear = 4

30

40

50

4. Now, actually we have deleted 2 elements from the queue so, there should be space for another 2 elements in the queue, but as the rear pointer is pointing at last position and Queue overflow condition  
(Rear == SIZE-1) is true, we can’t insert the new element in the queue even if it has an empty space.  
To overcome this problem there is another variation of queue called [CIRCULAR QUEUE](http://letusc-sharp.blogspot.com/2012/10/circular-queue.html)

**What are the advantages of Doubly Linked List over Singly Linked List?**

Advantages over singly linked list

1) A DLL can be **traversed in both forward and backward** direction.

2) The **delete operation** in DLL is more **efficient** if pointer to the node to be deleted is given.

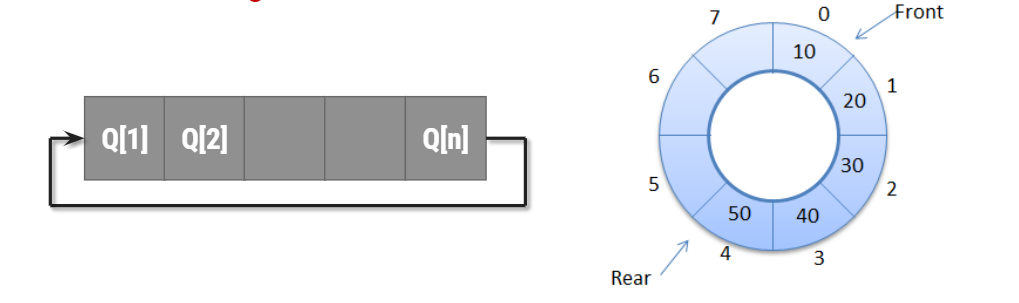
3) We can **quickly insert a new node before a given node.**

**Explain insertion operation in the Circular queue with all the conditions.**

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle

In circular queue the last node is connected back to the first node to make a circle.

It is also called as “Ring buffer”.



Operations on Circular Queue:

**Front**: Get the front item from queue.

**Rear**: Get the last item from queue.

**enQueue**(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

Check whether **queue is Full** – Check ((**rear == SIZE-1 && front == 0) || (rear == front-1)).**

If it is full then display Queue is full.

3. If queue is **not full** then, check if (**rear == SIZE – 1 && front != 0**) if it is true then set rear=0 and insert element.

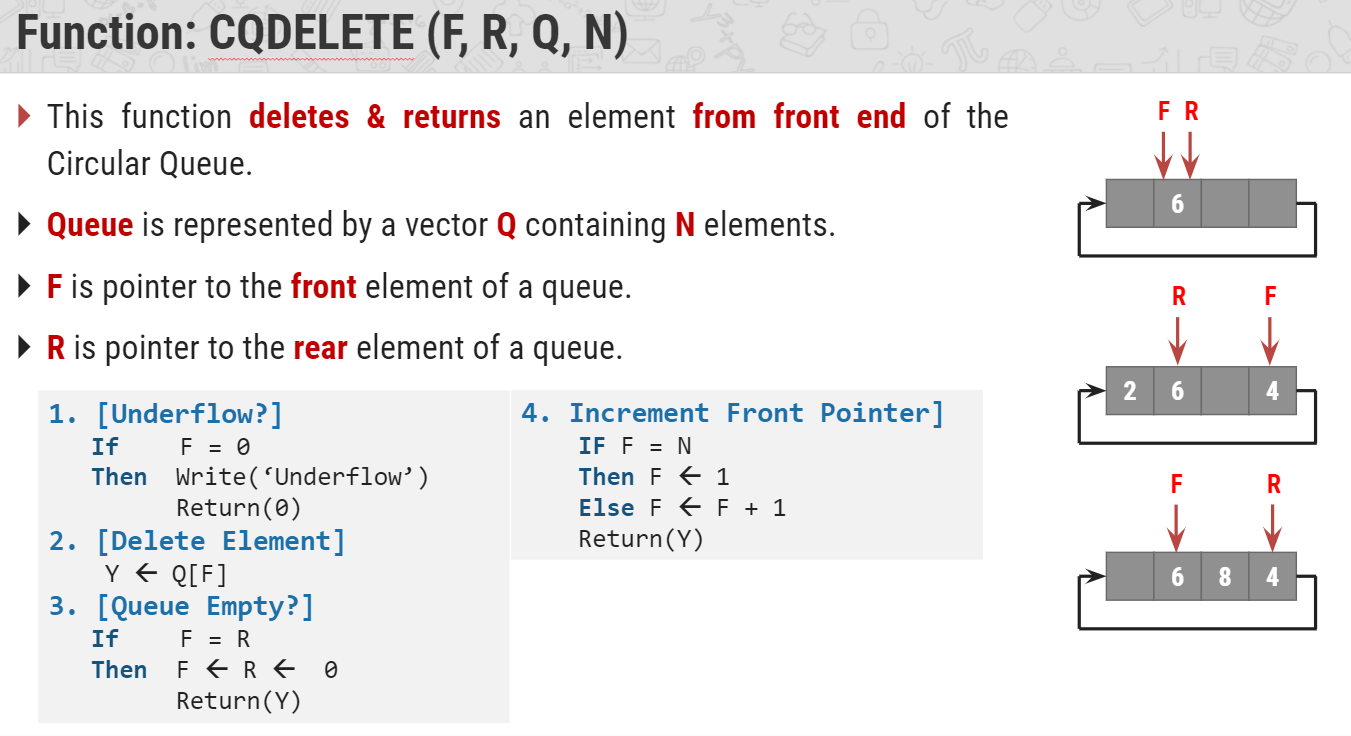
**deQueue**() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.

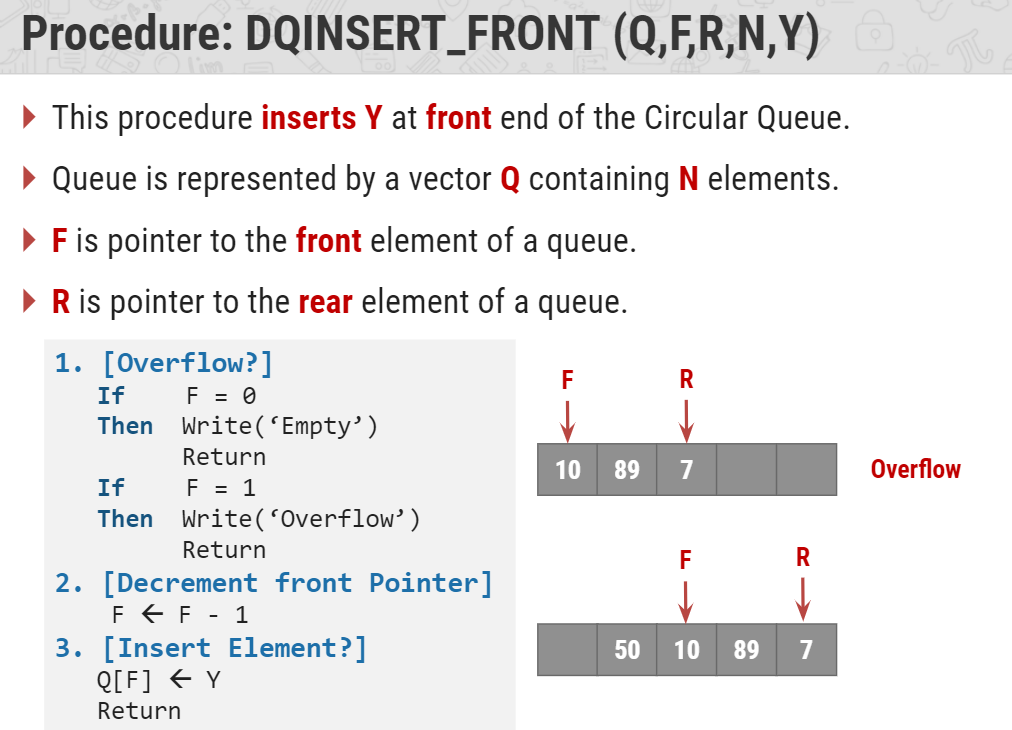
Check whether queue is **Empty** means check (**front==-1**).

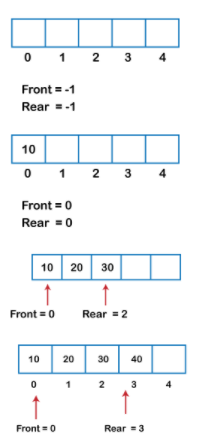
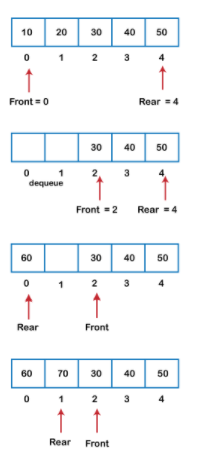
If it is empty then display Queue is empty.

If queue is not empty then step 3

Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

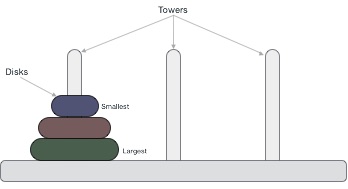






Discuss tower of hanoi problem for N=3 discs with recursive tracing.

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



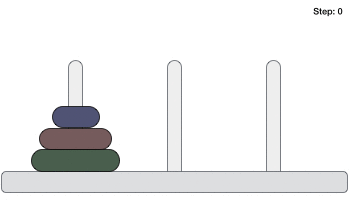
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

## Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

* Only one disk can be moved among the towers at any given time.
* Only the "top" disk can be removed.
* No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



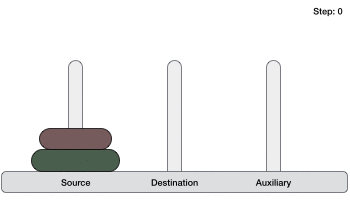
Tower of Hanoi puzzle with n disks can be solved in minimum **2n−1** steps. This presentation shows that a puzzle with 3 disks has taken **23 - 1 = 7** steps.

## Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

* First, we move the smaller (top) disk to aux peg.
* Then, we move the larger (bottom) disk to destination peg.
* And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

**Step 1** − Move n-1 disks from **source** to **aux**

**Step 2** − Move nth disk from **source** to **dest**

**Step 3** − Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

START

Procedure Hanoi(disk, source, dest, aux)

IF disk == 1, THEN

move disk from source to dest

ELSE

Hanoi(disk - 1, source, aux, dest) // Step 1

move disk from source to dest // Step 2

Hanoi(disk - 1, aux, dest, source) // Step 3

END IF

END Procedure

STOP

What is data structure? Explain types of data structure with an example.

<https://www.javatpoint.com/data-structure-introduction#:~:text=Data%20Structure%20can%20be%20defined,%2C%20Stack%2C%20Queue%2C%20etc>.

**Sorting**

Selection sort is a simple sorting algorithm.

This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

1. Start by finding the smallest entry.
2. Swap the smallest entry with the first entry.
3. Part of the array is now sorted.
4. Find the smallest element in the unsorted side.
5. Swap with the front of the unsorted side.
6. We have increased the size of the sorted side by one element.
7. The process continues...

## How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

Selection Sort

Selection Sort

Selection Sort

Selection Sort  
  


### **Pseudocode**

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

end for

end procedure

2. Insertion Sort

This is an in-place comparison-based sorting algorithm.

1. The Insertion Sort algorithm also views the array as having a sorted side and an unsorted side.
2. The sorted side starts with just the first element, which is not necessarily the smallest element.
3. The sorted side grows by taking the front element from the unsorted side...

...and inserting it in the place that keep the sort side arranged from small to large.

We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

Insertion Sort

Insertion Sort

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

## Pseudocode

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure