Data Structure and Algorithm

Data Structures

**Data:**

Data can be defined as a representation of facts, concepts, or instructions in a formalized manner, which should be suitable for communication, interpretation, or processing by human or electronic machine. Data is represented with the help of characters such as alphabets (A-Z, a-z), digits (0-9) or special characters (+,-,/,\*,<,>,= etc.) .

Data can be defined as an elementary value or the collection of values.

For example, student's name and its id are the data about the student. Catch

**Group Items:**

Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:**

 Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**File:**

A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:**

An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:**

Field is a single elementary unit of information representing the attribute of an entity.

What are Data Structures?

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.  Data Structures are

1. arrays,
2. Linked List,
3. Stack,
4. Queue

**Need of Data Structures:**

As applications are getting complexed and amount of data is increasing day by day, there may arise the following problems:

**Processor speed:**

To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

**Data Search:**

Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process.

**Multiple requests:**

 If thousands of users are searching for the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process in order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

**Advantages of Data Structures**

**Efficiency:**

  Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a perticular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.

**Reusability:**

Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.

**Abstraction:**

  Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

**Types of Data Structures**



There are two types of data structures:

1. Primitive data structure
2. Non-primitive data structure

**Primitive Data structure :**

The primitive data structures are primitive data types that can hold a single value. They are ---

1. int,
2. char,
3. float,
4. double, and
5. pointer

**Non-Primitive Data structure** :

The non-primitive data structure is divided into two types:

1. Linear data structure (Arrays, linked list, Stacks, and Queues)
2. Non-linear data structure(**trees and graphs)**

**Linear Data Structure**

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures. It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

**Linear data structures can also be classified as:**

1. **Static data structure:**
2. **Dynamic data structure:**

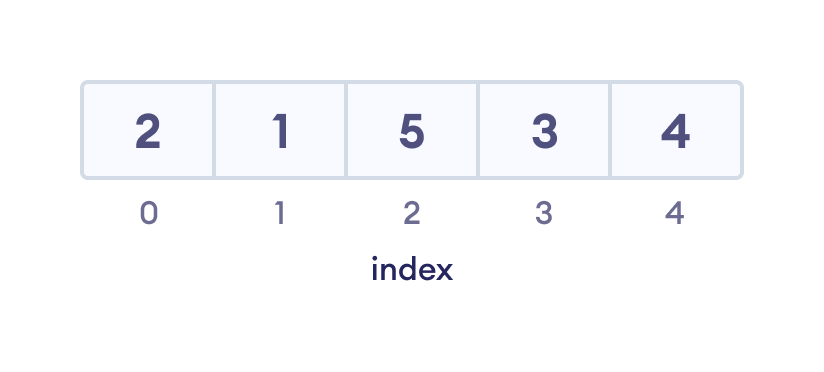
**Arrays:**

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].



An array with each element represented by an index

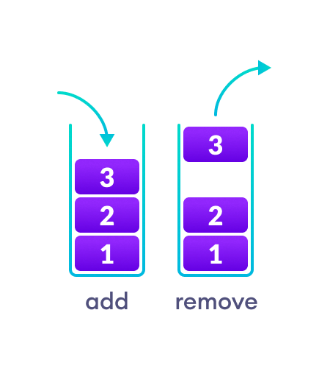
**Linked List:**

 Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.



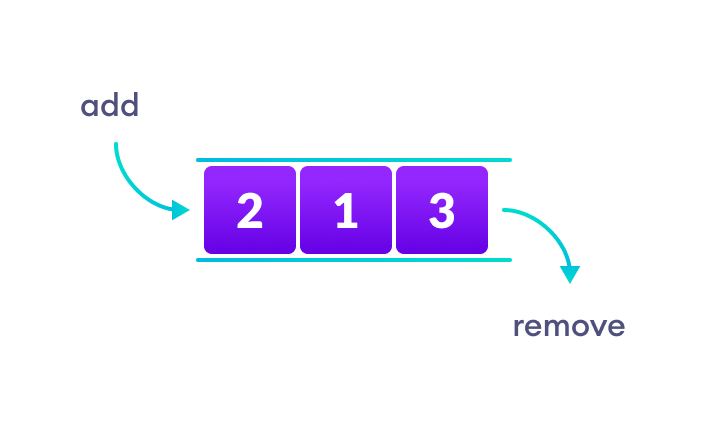
**Stack:**

Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**. A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.



**Queue:**

Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**. It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.



**Non-Linear Data Structures:**

This data structure does not form a sequence i.e., each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

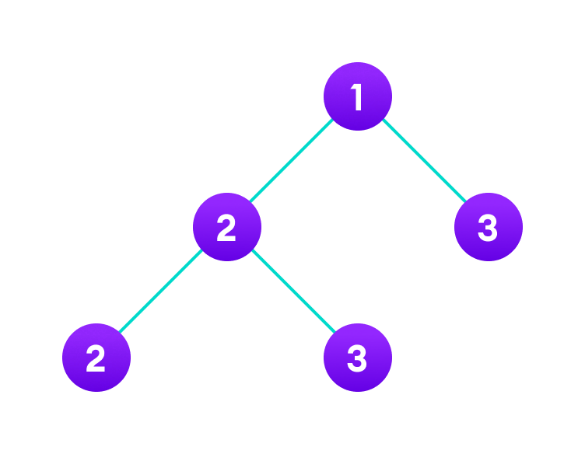
Types of Non-Linear Data Structures are given below:

1. **Trees:**
2. **Graphs:**

**Trees:**

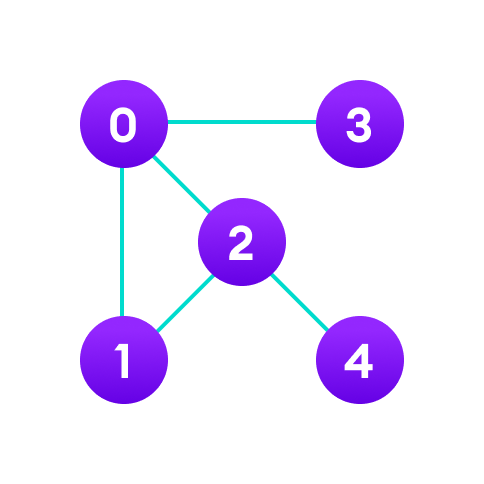
Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one child except the leaf nodes whereas each node can have at most one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.



**Graphs:**

Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.



**Algorithm**

An algorithm is a process, or a set of rules required to perform calculations or some other problem-solving operations especially by a computer. The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task. It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

**We need algorithms because of the following reasons:**

1. **Time is precious**
2. Scalability
3. **Performance:**
4. Memory

Algorithm Example:

The following are the steps required to add two numbers entered by the user:

Step 1: Start

Step 2: Declare three variables a, b, and sum.

Step 3: Enter the values of a and b.

Step 4: Add the values of a and b and store the result in the sum variable, i.e., sum= a + b.

Step 5: Print sum

Step 6: Stop

**Approaches of Algorithm**

**The following are the approaches used after considering both the theoretical and practical importance of designing an algorithm:**

1. **Brute force algorithm:** The general logic structure is applied to design an algorithm. It is also

known as an exhaustive search algorithm that searches all the possibilities to provide the required solution. Such algorithms are of two types:

1. **Optimizing:** Finding all the solutions of a problem and then take out the best solution or if the value of the best solution is known then it will terminate if the best solution is known.
2. **Sacrificing:** As soon as the best solution is found, then it will stop.
3. **Divide and conquer:** It is a very implementation of an algorithm. It allows you to design an algorithm in a step-by-step variation. It breaks down the algorithm to solve the problem in different methods. It allows you to break down the problem into different methods, and valid output is produced for the valid input. This valid output is passed to some other function.
4. **Greedy algorithm:** It is an algorithm paradigm that makes an optimal choice on each iteration with the hope of getting the best solution. It is easy to implement and has a faster execution time. But there are very rare cases in which it provides the optimal solution.
5. **Dynamic programming:** It makes the algorithm more efficient by storing the intermediate results. It follows five different steps to find the optimal solution for the problem:
6. It breaks down the problem into a subproblem to find the optimal solution.
7. After breaking down the problem, it finds the optimal solution out of these subproblems.
8. Stores the result of the subproblems is known as memorization.
9. Reuse the result so that it cannot be recomputed for the same subproblems.
10. Finally, it computes the result of the complex program.

**Branch and Bound Algorithm:**

The branch and bound algorithm can be applied to only integer programming problems. This approach divides all the sets of feasible solutions into smaller subsets. These subsets are further evaluated to find the best solution.

**Randomized Algorithm:**

As we have seen in a regular algorithm, we have predefined input and required output. Those algorithms that have some defined set of inputs and required output, and follow some described steps are known as deterministic algorithms. What happens that when the random variable is introduced in the randomized algorithm? In a randomized algorithm, some random bits are introduced by the algorithm and added in the input to produce the output, which is random in nature. Randomized algorithms are simpler and efficient than the deterministic algorithm.

**Backtracking:**

Backtracking is an algorithmic technique that solves the problem recursively and removes the solution if it does not satisfy the constraints of a problem.

The major categories of algorithms are given below:

1. Sort: Algorithm developed for sorting the items in a certain order.
2. Search: Algorithm developed for searching the items inside a data structure.
3. Delete: Algorithm developed for deleting the existing element from the data structure.
4. Insert: Algorithm developed for inserting an item inside a data structure.
5. Update: Algorithm developed for updating the existing element inside a data structure.

Algorithm Complexity

The performance of the algorithm can be measured in two factors:

**Time complexity:**

The time complexity of an algorithm is the amount of time required to complete the execution. The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity. The time complexity is mainly calculated by counting the number of steps to finish the execution. Let's understand the time complexity through an example.

sum=0;

**for** i=1 to n

sum= sum + i ;

**return** sum;

In the above code, the time complexity of the loop statement will be at least n, and if the value of n increases, then the time complexity also increases. While the complexity of the code, i.e., return sum will be constant as its value is not dependent on the value of n and will provide the result in one step only. We generally consider the worst-time complexity as it is the maximum time taken for any given input size.

**Space complexity:**

An algorithm's space complexity is the amount of space required to solve a problem and produce an output. Similar to the time complexity, space complexity is also expressed in big O notation.

For an algorithm, the space is required for the following purposes:

1. To store program instructions
2. To store constant values
3. To store variable values
4. To track the function calls, jumping statements, etc.

Auxiliary space: The extra space required by the algorithm, excluding the input size, is known as an auxiliary space. The space complexity considers both the spaces, i.e., auxiliary space, and space used by the input.

**Space complexity = Auxiliary space + Input size.**

Types of Algorithms

**The following are the types of algorithms:**

1. **Search Algorithm**
2. **Sort Algorithm**

**Search Algorithm**

On each day, we search for something in our day-to-day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user. There are mainly two techniques available to search the data in an array:

1. **Linear search**
2. **Binary search**

**Linear Search**

Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found. It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1. This algorithm can be implemented on the unsorted list.

**Binary Search**

A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list. The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner. It is used to find the middle element of the list.

Sorting Algorithms

Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.

Why do we need a sorting algorithm?

1. An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
2. It produces information in a sorted order, which is a human-readable format.
3. Searching a particular element in a sorted list is faster than the unsorted list.

Asymptotic Analysis

The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm. The efficiency is measured with the help of asymptotic notations.

**Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.**

C++ vs Java

**How to find the Time Complexity or running time for performing the operations?**

The measuring of the actual running time is not practical at all. The running time to perform any operation depends on the size of the input.

Suppose we have an array of five elements, and we want to add a new element at the beginning of the array. To achieve this, we need to shift each element towards right, and suppose each element takes one unit of time. There are five elements, so five units of time would be taken. Suppose there are 1000 elements in an array, then it takes 1000 units of time to shift. It

So, that time complexity depends upon the input size.

Therefore, if the input size is n, then f(n) is a function of n that denotes the time complexity.

How to calculate f(n)?

Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values. We can compare the data structures by comparing their f(n) values. We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes. Now, how to find f(n).

f(n) = 5n2 + 6n + 12

where n is the number of instructions executed, and it depends on the size of the input.

When n=1

% of running time due to 5n2 = Asymptotic Analysis \* 100 = 21.74%

% of running time due to 6n = Asymptotic Analysis \* 100 = 26.09%

% of running time due to 12 = Asymptotic Analysis \* 100 = 52.17%

From the above calculation, it is observed that most of the time is taken by 12. But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).

|  |  |  |  |
| --- | --- | --- | --- |
| n | 5n2 | 6n | 12 |
| 1 | 21.74% | 26.09% | 52.17% |
| 10 | 87.41% | 10.49% | 2.09% |
| 100 | 98.79% | 1.19% | 0.02% |
| 1000 | 99.88% | 0.12% | 0.0002% |

As we can observe in the above table that with the increase in the value of n, the running time of 5n2 increases while the running time of 6n and 12 also decreases. Therefore, it is observed that for larger values of n, the squared term consumes almost 99% of the time. As the n2 term is contributing most of the time, so we can eliminate the rest two terms.

**Therefore,**

f(n) = 5n2

Here, we are getting the approximate time complexity whose result is very close to the actual result. And this approximate measure of time complexity is known as an Asymptotic complexity. Here, we are not calculating the exact running time, we are eliminating the unnecessary terms, and we are just considering the term which is taking most of the time.

In mathematical analysis, asymptotic analysis of algorithm is a method of defining the mathematical bound of its run-time performance. Using the asymptotic analysis, we can easily conclude the average-case, best-case and worst-case scenario of an algorithm.

It is used to mathematically calculate the running time of any operation inside an algorithm.

**Example:** Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.

Usually, the time required by an algorithm comes under three types:

1. **Worst case:** It defines the input for which the algorithm takes a huge time.
2. **Average case:** It takes average time for the program execution.
3. **Best case:** It defines the input for which the algorithm takes the lowest time

Asymptotic Notations

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

1. Big oh Notation (O)
2. Omega Notation (Ω)
3. Theta Notation (θ)

Big oh Notation (O)

Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function. This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



**For example:**

If **f(n)** and **g(n)** are the two functions defined for positive integers. then

**f(n)** = **O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n))

such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

**Let's understand through examples**

Example 1: f(n)=2n+3 , g(n)=n

Now, we must find **Is f(n)=O(g(n))?**

To check f(n)=O(g(n)), it must satisfy the given condition:

**f(n)<=c.g(n)**

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then

2\*1+3<=5\*1

5<=5

For n=1, the above condition is true.

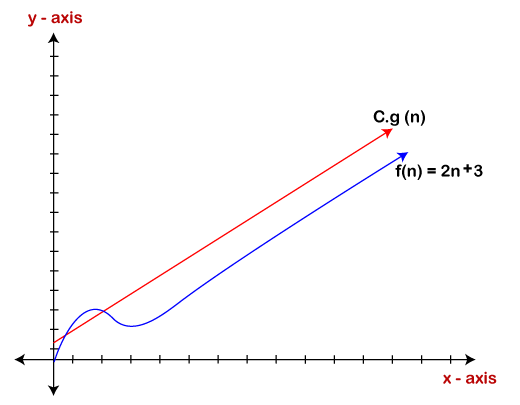
If n=2

2\*2+3<=5\*2

7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n). It can be represented graphically as:



The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity. It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

Omega Notation (Ω)

Omega Notation (Ω) describes lower bound of an algorithm's running time. It measures the best amount of time an algorithm can possibly take to complete or the best-case time complexity. It determines what is the fastest time that an algorithm can run.

If we required that an algorithm takes at least certain amount of time without using an upper bound, we use big- Ω notation i.e. the Greek letter "omega". It is used to bound the growth of running time for large input size.

If  **f(n)** and **g(n)** are the two functions defined for positive integers. then,  **f(n) = Ω (g(n))** as **f(n) is Omega of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:

**f(n) >= c.g(n) for all n ≥ no and c>0**

**Let's consider a simple example.**

If f(n) = 2n+3, g(n) = n,

Is f(n)= **Ω** (g(n))?

It must satisfy the condition:

**f(n) >= c.g(n)**

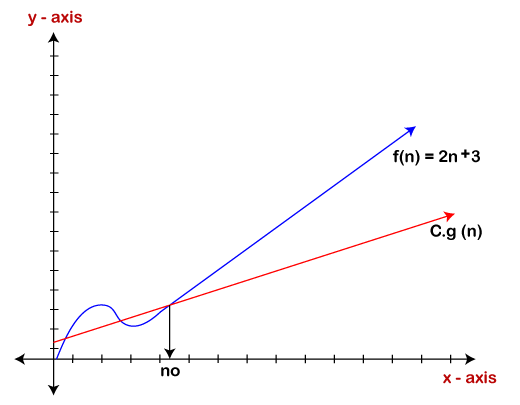
To check the above condition, we first replace f(n) by 2n+3 and g(n) by n.

**2n+3>=c\*n**

Suppose c=1

**2n+3>=n** (This equation will be true for any value of n starting from 1).

Therefore, it is proved that g(n) is big omega of 2n+3 function.



As we can see in the above figure that g(n) function is the lower bound of the f(n) function when the value of c is equal to 1. Therefore, this notation gives the fastest running time. But, we are not more interested in finding the fastest running time, we are interested in calculating the worst-case scenarios because we want to check our algorithm for larger input that what is the worst time that it will take so that we can take further decision in the further process.

Theta Notation (θ)

The theta notation mainly describes the average case scenarios. It represents the realistic time complexity of an algorithm. Every time, an algorithm does not perform worst or best, in real-world problems, algorithms mainly fluctuate between the worst-case and best-case, and this gives us the average case of the algorithm. Big theta is mainly used when the value of worst-case and the best-case is same. It is the formal way to express both the upper bound and lower bound of an algorithm running time.

Let's understand the big theta notation mathematically:

Let f(n) and g(n) be the functions of n where n is the steps required to execute the program then:

**f(n)= θg(n)**

The above condition is satisfied only if when

**c1.g(n)<=f(n)<=c2.g(n)**

where the function is bounded by two limits. Upper and Lower limit, and f(n) comes in between. The condition **f(n)= θg(n)** will be true if and only if c1.g(n) is less than or equal to f(n) and c2.g(n) is greater than or equal to f(n). The graphical representation of theta notation is given below:



Let's consider the same example where f(n)=2n+3 g(n)=n

As c1.g(n) should be less than f(n) so c1 has to be 1 whereas c2.g(n) should be greater than f(n) so c2 is equal to 5. The c1.g(n) is the lower limit of the of the f(n) while c2.g(n) is the upper limit of the f(n).

c1.g(n)<=f(n)<=c2.g(n)

Replace g(n) by n and f(n) by 2n+3

c1.n <=2n+3<=c2.n

if c1=1, c2=2, n=1

1\*1 <=2\*1+3 <=2\*1

**1** <= **5** <= **2** // for n=1, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

**If n=2**

1\*2<=2\*2+3<=2\*2

2<=7<=4 // for n=2, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n)

Therefore, we can say that for any value of n, it satisfies the condition c1.g(n)<=f(n)<=c2.g(n). Hence, it is proved that f(n) is big theta of g(n). So, this is the average-case scenario which provides the realistic time complexity.

Why we have three different asymptotic analysis?

As we know that big omega is for the best case, big oh is for the worst case while big theta is for the average case. Now, we will find out the average, worst and the best case of the linear search algorithm. Suppose we have an array of n numbers, and we want to find the element in an array using the linear search. In the linear search, every element is compared with the searched element on each iteration. Suppose if the match is found in a first iteration only, then the best case would be Ω(1), if the element matches with the last element, i.e., nth element of the array then the worst-case would-be O(n). The average case is the mid of the best and the worst-case, so it becomes **θ(n/1). The constant terms can be ignored in the time complexity so average case would be θ(n)**.

So, three different analyses provide the proper bounding between the actual functions. Here, bounding means that we have upper as well as lower limit which assures that the algorithm will behave between these limits only, i.e., it will not go beyond these limits.

Common Asymptotic Notations

|  |  |  |
| --- | --- | --- |
| constant | - | ?(1) |
| linear | - | ?(n) |
| logarithmic | - | ?(log n) |
| n log n | - | ?(n log n) |
| exponential | - | 2?(n) |
| cubic | - | ?(n3) |
| polynomial | - | n?(1) |
| quadratic | - | ?(n2) |

**Pointer**

Pointer is used to points the address of the value stored anywhere in the computer memory. To obtain the value stored at the location is known as dereferencing the pointer. Pointer improves the performance for repetitive process such as:

1. Traversing String
2. Lookup Tables
3. Control Tables
4. Tree Structures

**Pointer Details**

1. **Pointer arithmetic:** There are four arithmetic operators that can be used in pointers: ++, --, +, -
2. **Array of pointers:** You can define arrays to hold a number of pointers.
3. **Pointer to pointer:** C allows you to have pointer on a pointer and so on.
4. **Passing pointers to functions in C:** Passing an argument by reference or by address enable the passed argument to be changed in the calling function by the called function.
5. **Return pointer from functions in C:** C allows a function to return a pointer to the local variable, static variable and dynamically allocated memory as well.



**Example:**

**int** main( )

{

**int** a = 5;

**int** \*b;

b = &a;

printf ("value of a = %d\n", a);

printf ("value of a = %d\n", \*(&a));

printf ("value of a = %d\n", \*b);

printf ("address of a = %u\n", &a);

printf ("address of a = %d\n", b);

printf ("address of b = %u\n", &b);

printf ("value of b = address of a = %u", b);

**return** 0;

}

OUTPUT:

value of a = 5

value of a = 5

address of a = 3010494292

address of a = -1284473004

address of b = 3010494296

value of b = address of a = 3010494292

Example:

**int** main( )

{

**int** a = 5;

**int** \*b;

**int** \*\*c;

b = &a;

c = &b;

printf ("value of a = %d\n", a);

printf ("value of a = %d\n", \*(&a));

printf ("value of a = %d\n", \*b);

printf ("value of a = %d\n", \*\*c);

printf ("value of b = address of a = %u\n", b);

printf ("value of c = address of b = %u\n", c);

printf ("address of a = %u\n", &a);

printf ("address of a = %u\n", b);

printf ("address of a = %u\n", \*c);

printf ("address of b = %u\n", &b);

printf ("address of b = %u\n", c);

printf ("address of c = %u\n", &c);

**return** 0;

}

**Pointer to Pointer**

value of a = 5

value of a = 5

value of a = 5

value of a = 5

value of b = address of a = 2831685116

value of c = address of b = 2831685120

address of a = 2831685116

address of a = 2831685116

address of a = 2831685116

address of b = 2831685120

address of b = 2831685120

address of c = 2831685128

**Structure**

A structure is a composite data type that defines a grouped list of variables that are to be placed under one name in a block of memory. It allows different variables to be accessed by using a single pointer to the structure.

**Syntax**

struct structure\_name

{

    data\_type member1;

    data\_type member2;

    .

    .

    data\_type memeber;

};

**Advantages:**

1. It can hold variables of different data types.
2. We can create objects containing different types of attributes.
3. It allows us to re-use the data layout across programs.
4. It is used to implement other data structures like linked lists, stacks, queues, trees, graphs etc.

**Program:**

**void** main( )

{

struct employee

{

**int** id ;

**float** salary ;

**int** mobile ;

} ;

struct employee e1,e2,e3 ;

clrscr();

printf ("\nEnter ids, salary & mobile no. of 3 employee\n"

scanf ("%d %f %d", &e1.id, &e1.salary, &e1.mobile);

scanf ("%d%f %d", &e2.id, &e2.salary, &e2.mobile);

scanf ("%d %f %d", &e3.id, &e3.salary, &e3.mobile);

printf ("\n Entered Result ");

printf ("\n%d %f %d", e1.id, e1.salary, e1.mobile);

printf ("\n%d%f %d", e2.id, e2.salary, e2.mobile);

printf ("\n%d %f %d", e3.id, e3.salary, e3.mobile);

getch();

}

**Array:**

Arrays are defined as the collection of similar type of data items stored at contiguous memory locations. Arrays are the derived data type in C programming language which can store the primitive type of data such as,

1. int
2. char
3. double
4. float

Array is the simplest data structure where each data element can be randomly accessed by using its index number.

**Program without array:**

**void** main ()

{

**int** marks\_1 = 56, marks\_2 = 78, marks\_3 = 88, marks\_4 = 76;

**float** avg = (marks\_1 + marks\_2 + marks\_3 + marks\_4 ) / 4 ;

printf(avg);

}

**Program by using array:**

**void** main ()

{

**int** marks[6] = {56,78,88,76,56,89);

**int** i;

**float** avg;

**for** (i=0; i<6; i++ )

    {

        avg = avg + marks[i];

    }

    printf(avg);

}

**Complexity of Array operations**

Time and space complexity of various array operations are described in the following table.

**Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Average Case** | **Worst Case** |
| Access | O(1) | O(1) |
| Search | O(n) | O(n) |
| Insertion | O(n) | O(n) |
| Deletion | O(n) | O(n) |

**Space Complexity**

In array, space complexity for worst case is **O(n)**.

**Advantages of Array**

1. Array provides the single name for the group of variables of the same type therefore, it is easy to remember the name of all the elements of an array.
2. Traversing an array is a very simple process, we just need to increment the base address of the array in order to visit each element one by one.
3. Any element in the array can be directly accessed by using the index.

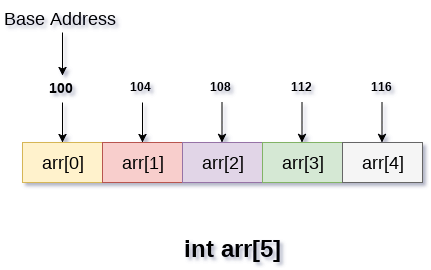
**Memory Allocation of the array**

As we have mentioned, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of first element in the main memory. Each element of the array is represented by a proper indexing.

The indexing of the array can be defined in three ways.

1. 0 (zero - based indexing) : The first element of the array will be arr[0].
2. 1 (one - based indexing) : The first element of the array will be arr[1].
3. n (n - based indexing) : The first element of the array can reside at any random index number.

In the following image, we have shown the memory allocation of an array arr of size 5. The array follows 0-based indexing approach. The base address of the array is 100th byte. This will be the address of arr[0]. Here, the size of int is 4 bytes therefore each element will take 4 bytes in the memory.



In 0 based indexing, If the size of an array is n then the maximum index number, an element can have is **n-1**. However, it will be n if we use **1** based indexing.

**Accessing Elements of an array:**

To access any random element of an array we need the following information:

1. Base Address of the array.
2. Size of an element in bytes.
3. Which type of indexing, array follows.

Address of any element of a 1D array can be calculated by using the following formula:

Byte address of element A[i]  = base address + size \* ( i - first index)

**Example :**

In an array, A[-10 ..... +2 ], Base address (BA) = 999, size of an element = 2 bytes,

find the location of A[-1].

L(A[-1]) = 999 + [(-1) - (-10)] x 2

       = 999 + 18

       = 1017

**Passing array to the function :**

As we have mentioned earlier that, the name of the array represents the starting address or the address of the first element of the array. All the elements of the array can be traversed by using the base address.

**Example:**

**int** summation(**int**[]);

**void** main ()

{

**int** arr[5] = {0,1,2,3,4};

**int** sum = summation(arr);

    printf("%d",sum);

}

**int** summation (**int** arr[])

{

**int** sum=0,i;

**for** (i = 0; i<5; i++)

    {

        sum = sum + arr[i];

    }

**return** sum;

}

The above program defines a function named as summation which accepts an array as an argument. The function calculates the sum of all the elements of the array and returns it.

**2D Array**

The syntax of declaring two-dimensional array is very much similar to that of a one dimensional array, given as follows.

**int** arr[max\_rows][max\_columns];

**void** main ()

{

**int** arr[3][3],i,j;

**for** (i=0;i<3;i++)

    {

**for** (j=0;j<3;j++)

        {

            printf("Enter a[%d][%d]: ",i,j);

            scanf("%d",&arr[i][j]);

        }

    }

    printf("\n printing the elements ....\n");

**for**(i=0;i<3;i++)

    {

        printf("\n");

**for** (j=0;j<3;j++)

        {

            printf("%d\t",arr[i][j]);

        }

    }

}

Linked list:

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list:

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below.



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

**How to declare a linked list?**

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure.**

The declaration of linked list is given as follows -

struct node

{

int data;

struct node \*next;

}

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

**Types of Linked list**

Linked list is classified into the following types -

1. **Singly linked list -** Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
2. **Doubly linked list -** Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
3. **Circular singly linked list -** In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
4. **Circular doubly linked list -** Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

**Advantages of Linked list**

The advantages of using the Linked list are given as follows -

1. **Dynamic data structure -** The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
2. **Insertion and deletion -** Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just must update the address of the pointer of the node.
3. **Memory efficient -** The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
4. **Implementation -** We can implement both stacks and queues using linked list.

**Disadvantages of Linked list**

The limitations of using the Linked list are given as follows -

1. **Memory usage -** In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
2. **Traversal -** Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
3. **Reverse traversing -** Backtracking or reverse traversing is difficult in a linked list. In a doubly linked list, it is easier but requires more memory to store the back pointer.

**Applications of Linked list**

The applications of the Linked list are given as follows -

1. With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
2. A linked list can be used to represent the sparse matrix.
3. The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
4. Using linked list, we can implement stack, queue, tree, and other various data structures.
5. The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
6. A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.

**Operations performed on Linked list**

The basic operations that are supported by a list are mentioned as follows -

1. **Insertion -** This operation is performed to add an element into the list.
2. **Deletion -** It is performed to delete an operation from the list.
3. **Display -** It is performed to display the elements of the list.
4. **Search -** It is performed to search an element from the list using the given key.

**Complexity of Linked list**

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. **Time Complexity**

|  |  |  |
| --- | --- | --- |
| **Operations** | **Average case time complexity** | **Worst-case time complexity** |
| **Insertion** | O(1) | O(1) |
| **Deletion** | O(1) | O(1) |
| **Search** | O(n) | O(n) |

Where 'n' is the number of nodes in the given tree.

**2. Space Complexity**

|  |  |
| --- | --- |
| **Operations** | **Space complexity** |
| **Insertion** | O(n) |
| **Deletion** | O(n) |
| **Search** | O(n) |

The space complexity of linked list is **O(n).**

Types of Linked list

**The following are the types of linked list:**

1. Singly Linked list
2. Doubly Linked list
3. Circular Linked list
4. Doubly Circular Linked list

Singly linked list or One way chain:

Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor. One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we cannot traverse the list in the reverse direction.

Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

DS Singly Linked List

In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|  | **Average** | | | | **Worst** | | | | **Worst** |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| Singly Linked List | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation

struct node

{

**int** data;

    struct node \*next;

};

struct node \*head, \*ptr;

ptr = (struct node \*)malloc(sizeof(struct node \*));

Insertion:

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-singly-linked-list-at-beginning) | It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list. |
| 2 | [Insertion at end of the list](https://www.javatpoint.com/insertion-in-singly-linked-list-at-end) | It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-singly-linked-list-after-specified-node) | It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. . |

Deletion and Traversing:

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-singly-linked-list-at-beginning) | It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers. |
| 2 | [Deletion at the end of the list](https://www.javatpoint.com/deletion-in-singly-linked-list-at-end) | It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios. |
| 3 | [Deletion after specified node](https://www.javatpoint.com/deletion-in-singly-linked-list-after-specified-node) | It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-singly-linked-list) | In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list. |
| 5 | [Searching](https://www.javatpoint.com/searching-in-singly-linked-list) | In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. . |

Linked List in C: Menu Driven Program

struct node

{

**int** data;

    struct node \*next;

};

struct node \*head;

**void** beginsert ();

**void** lastinsert ();

**void** randominsert();

**void** begin\_delete();

**void** last\_delete();

**void** random\_delete();

**void** display();

**void** search();

**void** main ()

{

**int** choice =0;

**while**(choice != 9)

    {

        printf("\n\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");

        printf("\nChoose one option from the following list ...\n");

        printf("\n===============================================\n");

        printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4.Delete from Beginning\n

        5.Delete from last\n6.Delete node after specified location\n7.Search **for** an element\n8.Show\n9.Exit\n");

        printf("\nEnter your choice?\n");

        scanf("\n%d",&choice);

**switch**(choice)

        {

**case** 1:

            beginsert();

**break**;

**case** 2:

            lastinsert();

**break**;

**case** 3:

            randominsert();

**break**;

**case** 4:

            begin\_delete();

**break**;

**case** 5:

            last\_delete();

**break**;

**case** 6:

            random\_delete();

**break**;

**case** 7:

            search();

**break**;

**case** 8:

            display();

**break**;

**case** 9:

            exit(0);

**break**;

**default**:

             printf("Please enter valid choice..");

        }

    }

}

**void** beginsert()

{

    struct node \*ptr;

**int** item;

    ptr = (struct node \*) malloc(sizeof(struct node \*));

**if**(ptr == NULL)

    {

        printf("\nOVERFLOW");

    }

**else**

    {

        printf("\nEnter value\n");

        scanf("%d",&item);

        ptr->data = item;

        ptr->next = head;

        head = ptr;

        printf("\nNode inserted");

    }

}

**void** lastinsert()

{

    struct node \*ptr,\*temp;

**int** item;

    ptr = (struct node\*)malloc(sizeof(struct node));

**if**(ptr == NULL)

    {

        printf("\nOVERFLOW");

    }

**else**

    {

        printf("\nEnter value?\n");

        scanf("%d",&item);

        ptr->data = item;

**if**(head == NULL)

        {

            ptr -> next = NULL;

            head = ptr;

            printf("\nNode inserted");

        }

**else**

        {

            temp = head;

**while** (temp -> next != NULL)

            {

                temp = temp -> next;

            }

            temp->next = ptr;

            ptr->next = NULL;

            printf("\nNode inserted");

        }

    }

}

**void** randominsert()

{

**int** i,loc,item;

    struct node \*ptr, \*temp;

    ptr = (struct node \*) malloc (sizeof(struct node));

**if**(ptr == NULL)

    {

        printf("\nOVERFLOW");

    }

**else**

    {

        printf("\nEnter element value");

        scanf("%d",&item);

        ptr->data = item;

        printf("\nEnter the location after which you want to insert ");

        scanf("\n%d",&loc);

        temp=head;

**for**(i=0; I < loc; i++)

        {

            temp = temp->next;

**if**(temp == NULL)

            {

                printf("\ncan't insert\n");

**return**;

            }

        }

        ptr ->next = temp ->next;

        temp ->next = ptr;

        printf("\nNode inserted");

    }

}

**void** begin\_delete()

{

    struct node \*ptr;

**if**(head == NULL)

    {

        printf("\nList is empty\n");

    }

**else**

    {

        ptr = head;

        head = ptr->next;

        free(ptr);

        printf("\nNode deleted from the begining ...\n");

    }

}

**void** last\_delete()

{

    struct node \*ptr,\*ptr1;

**if**(head == NULL)

    {

        printf("\nlist is empty");

    }

**else** **if**(head -> next == NULL)

    {

        head = NULL;

        free(head);

        printf("\nOnly node of the list deleted ...\n");

    }

**else**

    {

        ptr = head;

**while**(ptr->next != NULL)

        {

            ptr1 = ptr;

            ptr = ptr ->next;

        }

        ptr1->next = NULL;

        free(ptr);

        printf("\nDeleted Node from the last ...\n");

    }

}

**void** random\_delete()

{

    struct node \*ptr,\*ptr1;

**int** loc,i;

    printf("\n Enter the location of the node after which you want to perform deletion \n");

    scanf("%d",&loc);

    ptr=head;

**for**(i=0;i<loc;i++)

    {

        ptr1 = ptr;

        ptr = ptr->next;

**if**(ptr == NULL)

        {

            printf("\nCan't delete");

**return**;

        }

    }

    ptr1 ->next = ptr ->next;

    free(ptr);

    printf("\nDeleted node %d ",loc+1);

}

**void** search()

{

    struct node \*ptr;

**int** item,i=0,flag;

    ptr = head;

**if**(ptr == NULL)

    {

        printf("\nEmpty List\n");

    }

**else**

    {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

**while** (ptr!=NULL)

        {

**if**(ptr->data == item)

            {

                printf("item found at location %d ",i+1);

                flag=0;

            }

**else**

            {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

**if**(flag==1)

        {

            printf("Item not found\n");

        }

    }

}

**void** display()

{

    struct node \*ptr;

    ptr = head;

**if**(ptr == NULL)

    {

        printf("Nothing to print");

    }

**else**

    {

        printf("\nprinting values . . . . .\n");

**while** (ptr!=NULL)

        {

            printf("\n%d",ptr->data);

            ptr = ptr -> next;

        }

    }

}

Doubly linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



In C, structure of a node in doubly linked list can be given as :

struct node

{

    struct node \*prev;

**int** data;

    struct node \*next;

}

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

Keep Watching

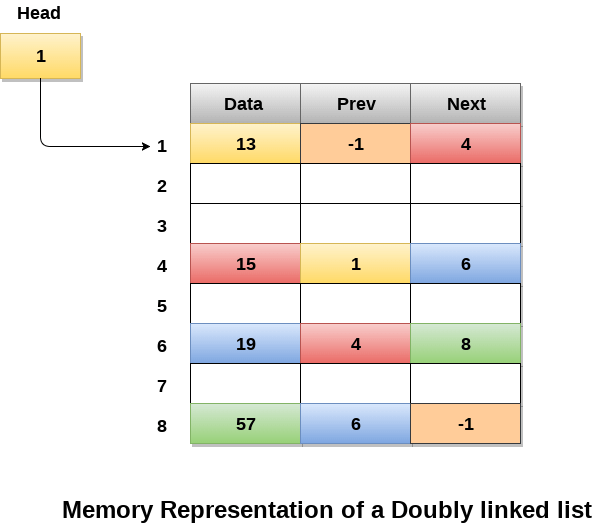
In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).

In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

We can traverse the list in this way until we find any node containing null or -1 in its next part.



Operations on doubly linked list

**Node Creation**

struct node

{

    struct node \*prev;

**int** data;

    struct node \*next;

};

struct node \*head;

All the remaining operations regarding doubly linked list are described in the following table.

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-beginning) | Adding the node into the linked list at beginning. |
| 2 | [Insertion at end](https://www.javatpoint.com/insertion-in-doubly-linked-list-at-the-end) | Adding the node into the linked list to the end. |
| 3 | [Insertion after specified node](https://www.javatpoint.com/insertion-in-doubly-linked-list-after-specified-node) | Adding the node into the linked list after the specified node. |
| 4 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-beginning) | Removing the node from beginning of the list |
| 5 | [Deletion at the end](https://www.javatpoint.com/deletion-in-doubly-linked-list-at-the-end) | Removing the node from end of the list. |
| 6 | [Deletion of the node having given data](https://www.javatpoint.com/deletion-in-doubly-linked-list-after-the-specified-node) | Removing the node which is present just after the node containing the given data. |
| 7 | [Searching](https://www.javatpoint.com/searching-in-doubly-linked-list) | Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null. |
| 8 | [Traversing](https://www.javatpoint.com/traversing-in-doubly-linked-list) | Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc. |

Menu Driven Program in C to implement all the operations of doubly linked list

struct node

{

    struct node \*prev;

    struct node \*next;

**int** data;

};

struct node \*head;

**void** insertion\_beginning();

**void** insertion\_last();

**void** insertion\_specified();

**void** deletion\_beginning();

**void** deletion\_last();

**void** deletion\_specified();

**void** display();

**void** search();

**void** main ()

{

**int** choice =0;

}

**void** insertion\_beginning()

{

   struct node \*ptr;

**int** item;

   ptr = (struct node \*)malloc(sizeof(struct node));

**if**(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

**else**

   {

    printf("\nEnter Item value");

    scanf("%d",&item);

**if**(head==NULL)

   {

       ptr->next = NULL;

       ptr->prev=NULL;

       ptr->data=item;

       head=ptr;

   }

**else**

   {

       ptr->data=item;

       ptr->prev=NULL;

       ptr->next = head;

       head->prev=ptr;

       head=ptr;

   }

   printf("\nNode inserted\n");

}

}

**void** insertion\_last()

{

   struct node \*ptr,\*temp;

**int** item;

   ptr = (struct node \*) malloc(sizeof(struct node));

**if**(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

**else**

   {

       printf("\nEnter value");

       scanf("%d",&item);

        ptr->data=item;

**if**(head == NULL)

       {

           ptr->next = NULL;

           ptr->prev = NULL;

           head = ptr;

       }

**else**

       {

          temp = head;

**while**(temp->next!=NULL)

          {

              temp = temp->next;

          }

          temp->next = ptr;

          ptr ->prev=temp;

          ptr->next = NULL;

          }

       }

     printf("\nnode inserted\n");

    }

**void** insertion\_specified()

{

struct node \*ptr,\*temp;

**int** item,loc,i;

ptr = (struct node \*)malloc(sizeof(struct node));

**if**(ptr == NULL)

{

printf("\n OVERFLOW");

}

**else**

{

temp=head;

printf("Enter the location");

scanf("%d",&loc);

**for**(i=0;i<loc;i++)

{

temp = temp->next;

**if**(temp == NULL)

{

printf("\n There are less than %d elements", loc);

**return**;

}

}

printf("Enter value");

scanf("%d",&item);

ptr->data = item;

ptr->next = temp->next;

ptr -> prev = temp;

temp->next = ptr;

temp->next->prev=ptr;

printf("\nnode inserted\n");

}

}

**void** deletion\_beginning()

{

    struct node \*ptr;

**if**(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

**else** **if**(head->next == NULL)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {

        ptr = head;

        head = head -> next;

        head -> prev = NULL;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

**void** deletion\_last()

{

    struct node \*ptr;

**if**(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

**else** **if**(head->next == NULL)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {

        ptr = head;

**if**(ptr->next != NULL)

        {

            ptr = ptr -> next;

        }

        ptr -> prev -> next = NULL;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

**void** deletion\_specified()

{

    struct node \*ptr, \*temp;

**int** val;

    printf("\n Enter the data after which the node is to be deleted : ");

    scanf("%d", &val);

    ptr = head;

**while**(ptr -> data != val)

    ptr = ptr -> next;

**if**(ptr -> next == NULL)

    {

        printf("\nCan't delete\n");

    }

**else** **if**(ptr -> next -> next == NULL)

    {

        ptr ->next = NULL;

    }

**else**

    {

        temp = ptr -> next;

        ptr -> next = temp -> next;

        temp -> next -> prev = ptr;

        free(temp);

        printf("\nnode deleted\n");

    }

}

**void** display()

{

    struct node \*ptr;

    printf("\n printing values...\n");

    ptr = head;

**while**(ptr != NULL)

    {

        printf("%d\n",ptr->data);

        ptr=ptr->next;

    }

}

**void** search()

{

    struct node \*ptr;

**int** item,i=0,flag;

    ptr = head;

**if**(ptr == NULL)

    {

        printf("\nEmpty List\n");

    }

**else**

    {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

**while** (ptr!=NULL)

        {

**if**(ptr->data == item)

            {

                printf("\nitem found at location %d ",i+1);

                flag=0;

**break**;

            }

**else**

            {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

**if**(flag==1)

        {

            printf("\nItem not found\n");

        }

    }

}

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.

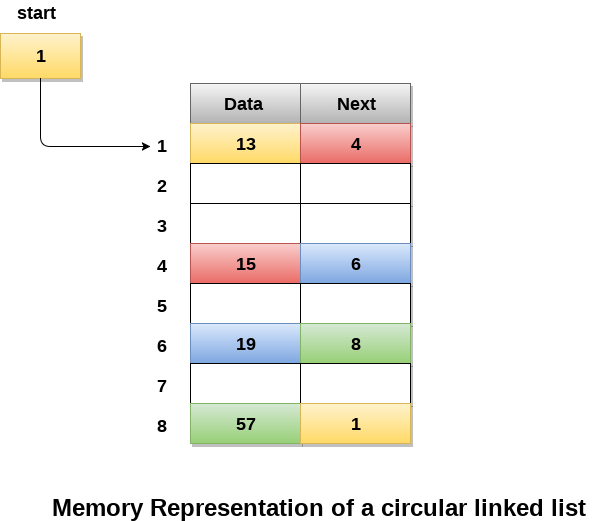


Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.

However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.



We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.

Operations on Circular Singly linked list:

Insertion

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-circular-singly-list-at-beginning) | Adding a node into circular singly linked list at the beginning. |
| 2 | [Insertion at the end](https://www.javatpoint.com/insertion-in-circular-singly-linked-list-at-end) | Adding a node into circular singly linked list at the end. |

Deletion & Traversing

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-circular-singly-linked-list-at-beginning) | Removing the node from circular singly linked list at the beginning. |
| 2 | [Deletion at the end](https://www.javatpoint.com/deletion-in-circular-singly-linked-list-at-end) | Removing the node from circular singly linked list at the end. |
| 3 | [Searching](https://www.javatpoint.com/searching-in-circular-singly-linked-list) | Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null. |
| 4 | [Traversing](https://www.javatpoint.com/traversing-in-circular-singly-linked-list) | Visiting each element of the list at least once in order to perform some specific operation. |

Menu-driven program in C implementing all operations

on circular singly linked list

#include<stdio.h>

#include<stdlib.h>

struct node

{

**int** data;

    struct node \*next;

};

struct node \*head;

**void** beginsert ();

**void** lastinsert ();

**void** randominsert();

**void** begin\_delete();

**void** last\_delete();

**void** random\_delete();

**void** display();

**void** search();

**void** main ()

{

**int** choice =0;

**while**(choice != 7)

    {

        printf("\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");

        printf("\nChoose one option from the following list ...\n");

        printf("\n===============================================\n");

        printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete from last\n5.Search for an element\n6.Show\n7.Exit\n");

        printf("\nEnter your choice?\n");

        scanf("\n%d",&choice);

**switch**(choice)

        {

**case** 1:

            beginsert();

**break**;

**case** 2:

            lastinsert();

**break**;

**case** 3:

            begin\_delete();

**break**;

**case** 4:

            last\_delete();

**break**;

**case** 5:

            search();

**break**;

**case** 6:

            display();

**break**;

**case** 7:

            exit(0);

**break**;

**default**:

            printf("Please enter valid choice..");

        }

    }

}

**void** beginsert()

{

    struct node \*ptr,\*temp;

**int** item;

    ptr = (struct node \*)malloc(sizeof(struct node));

**if**(ptr == NULL)

    {

        printf("\nOVERFLOW");

    }

**else**

    {

        printf("\nEnter the node data?");

        scanf("%d",&item);

        ptr -> data = item;

**if**(head == NULL)

        {

            head = ptr;

            ptr -> next = head;

        }

**else**

        {

            temp = head;

**while**(temp->next != head)

                temp = temp->next;

            ptr->next = head;

            temp -> next = ptr;

            head = ptr;

        }

        printf("\nnode inserted\n");

    }

}

**void** lastinsert()

{

    struct node \*ptr,\*temp;

**int** item;

    ptr = (struct node \*)malloc(sizeof(struct node));

**if**(ptr == NULL)

    {

        printf("\nOVERFLOW\n");

    }

**else**

    {

        printf("\nEnter Data?");

        scanf("%d",&item);

        ptr->data = item;

**if**(head == NULL)

        {

            head = ptr;

            ptr -> next = head;

        }

**else**

        {

            temp = head;

**while**(temp -> next != head)

            {

                temp = temp -> next;

            }

            temp -> next = ptr;

            ptr -> next = head;

        }

        printf("\nnode inserted\n");

    }

}

**void** begin\_delete()

{

    struct node \*ptr;

**if**(head == NULL)

    {

        printf("\nUNDERFLOW");

    }

**else** **if**(head->next == head)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {   ptr = head;

**while**(ptr -> next != head)

            ptr = ptr -> next;

        ptr->next = head->next;

        free(head);

        head = ptr->next;

        printf("\nnode deleted\n");

    }

}

**void** last\_delete()

{

    struct node \*ptr, \*preptr;

**if**(head==NULL)

    {

        printf("\nUNDERFLOW");

    }

**else** **if** (head ->next == head)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {

        ptr = head;

**while**(ptr ->next != head)

        {

            preptr=ptr;

            ptr = ptr->next;

        }

        preptr->next = ptr -> next;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

**void** search()

{

    struct node \*ptr;

**int** item,i=0,flag=1;

    ptr = head;

**if**(ptr == NULL)

    {

        printf("\nEmpty List\n");

    }

**else**

    {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

**if**(head ->data == item)

        {

        printf("item found at location %d",i+1);

        flag=0;

        }

**else**

        {

**while** (ptr->next != head)

        {

**if**(ptr->data == item)

            {

                printf("item found at location %d ",i+1);

                flag=0;

**break**;

            }

**else**

            {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

        }

**if**(flag != 0)

        {

            printf("Item not found\n");

        }

    }

}

**void** display()

{

    struct node \*ptr;

    ptr=head;

**if**(head == NULL)

    {

        printf("\nnothing to print");

    }

**else**

    {

        printf("\n printing values ... \n");

**while**(ptr -> next != head)

        {

            printf("%d\n", ptr -> data);

            ptr = ptr -> next;

        }

        printf("%d\n", ptr -> data);

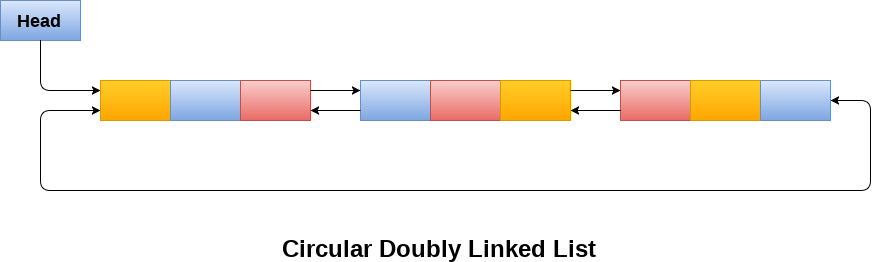
    }

}

Circular Doubly Linked List

Circular doubly linked list is a more complexed type of data structure in which a node contain pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the node. The last node of the list contains the address of the first node of the list. The first node of the list also contain address of the last node in its previous pointer.

A circular doubly linked list is shown in the following figure.

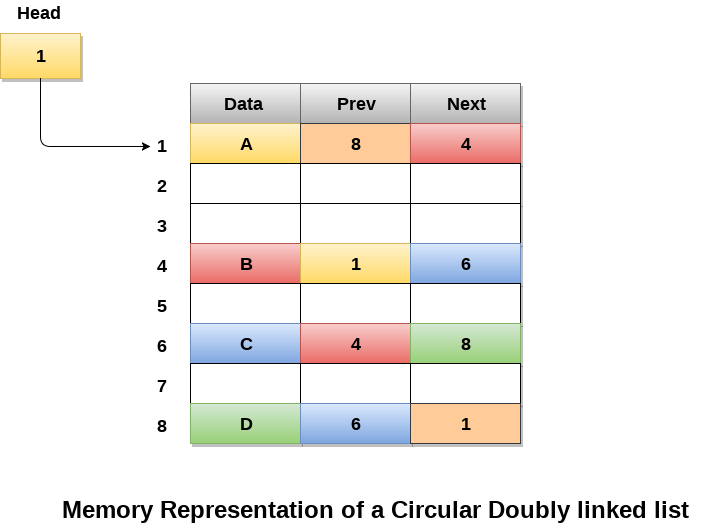


Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations. However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Memory Management of Circular Doubly linked list

The following figure shows the way in which the memory is allocated for a circular doubly linked list. The variable head contains the address of the first element of the list i.e. 1 hence the starting node of the list contains data A is stored at address 1. Since, each node of the list is supposed to have three parts therefore, the starting node of the list contains address of the last node i.e. 8 and the next node i.e. 4. The last node of the list that is stored at address 8 and containing data as 6, contains address of the first node of the list as shown in the image i.e. 1. In circular doubly linked list, the last node is identified by the address of the first node which is stored in the next part of the last node therefore the node which contains the address of the first node, is actually the last node of the list.

Difference between JDK, JRE, and JVM



Operations on circular doubly linked list :

There are various operations which can be performed on circular doubly linked list. The node structure of a circular doubly linked list is similar to doubly linked list. However, the operations on circular doubly linked list is described in the following table.

|  |  |  |
| --- | --- | --- |
| SN | Operation | Description |
| 1 | [Insertion at beginning](https://www.javatpoint.com/insertion-in-circular-doubly-linked-list-at-beginning) | Adding a node in circular doubly linked list at the beginning. |
| 2 | [Insertion at end](https://www.javatpoint.com/insertion-in-circular-doubly-linked-list-at-end) | Adding a node in circular doubly linked list at the end. |
| 3 | [Deletion at beginning](https://www.javatpoint.com/deletion-in-circular-doubly-linked-list-at-beginning) | Removing a node in circular doubly linked list from beginning. |
| 4 | [Deletion at end](https://www.javatpoint.com/deletion-in-circular-doubly-linked-list-at-end) | Removing a node in circular doubly linked list at the end. |

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

C program to implement all the operations on circular doubly linked list

#include<stdio.h>

#include<stdlib.h>

struct node

{

    struct node \*prev;

    struct node \*next;

**int** data;

};

struct node \*head;

**void** insertion\_beginning();

**void** insertion\_last();

**void** deletion\_beginning();

**void** deletion\_last();

**void** display();

**void** search();

**void** main ()

{

**int** choice =0;

**while**(choice != 9)

    {

        printf("\n\*\*\*\*\*\*\*\*\*Main Menu\*\*\*\*\*\*\*\*\*\n");

        printf("\nChoose one option from the following list ...\n");

        printf("\n===============================================\n");

        printf("\n1.Insert in Beginning\n2.Insert at last\n3.Delete from Beginning\n4.Delete from last\n5.Search\n6.Show\n7.Exit\n");

        printf("\nEnter your choice?\n");

        scanf("\n%d",&choice);

**switch**(choice)

        {

**case** 1:

            insertion\_beginning();

**break**;

**case** 2:

                    insertion\_last();

**break**;

**case** 3:

            deletion\_beginning();

**break**;

**case** 4:

            deletion\_last();

**break**;

**case** 5:

            search();

**break**;

**case** 6:

            display();

**break**;

**case** 7:

            exit(0);

**break**;

**default**:

            printf("Please enter valid choice..");

        }

    }

}

**void** insertion\_beginning()

{

   struct node \*ptr,\*temp;

**int** item;

   ptr = (struct node \*)malloc(sizeof(struct node));

**if**(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

**else**

   {

    printf("\nEnter Item value");

    scanf("%d",&item);

    ptr->data=item;

**if**(head==NULL)

   {

      head = ptr;

      ptr -> next = head;

      ptr -> prev = head;

   }

**else**

   {

       temp = head;

**while**(temp -> next != head)

    {

        temp = temp -> next;

    }

    temp -> next = ptr;

    ptr -> prev = temp;

    head -> prev = ptr;

    ptr -> next = head;

    head = ptr;

   }

   printf("\nNode inserted\n");

}

}

**void** insertion\_last()

{

   struct node \*ptr,\*temp;

**int** item;

   ptr = (struct node \*) malloc(sizeof(struct node));

**if**(ptr == NULL)

   {

       printf("\nOVERFLOW");

   }

**else**

   {

       printf("\nEnter value");

       scanf("%d",&item);

        ptr->data=item;

**if**(head == NULL)

       {

           head = ptr;

           ptr -> next = head;

           ptr -> prev = head;

       }

**else**

       {

          temp = head;

**while**(temp->next !=head)

          {

              temp = temp->next;

          }

          temp->next = ptr;

          ptr ->prev=temp;

          head -> prev = ptr;

      ptr -> next = head;

        }

   }

     printf("\nnode inserted\n");

}

**void** deletion\_beginning()

{

    struct node \*temp;

**if**(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

**else** **if**(head->next == head)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {

        temp = head;

**while**(temp -> next != head)

        {

            temp = temp -> next;

        }

        temp -> next = head -> next;

        head -> next -> prev = temp;

        free(head);

        head = temp -> next;

    }

}

**void** deletion\_last()

{

    struct node \*ptr;

**if**(head == NULL)

    {

        printf("\n UNDERFLOW");

    }

**else** **if**(head->next == head)

    {

        head = NULL;

        free(head);

        printf("\nnode deleted\n");

    }

**else**

    {

        ptr = head;

**if**(ptr->next != head)

        {

            ptr = ptr -> next;

        }

        ptr -> prev -> next = head;

        head -> prev = ptr -> prev;

        free(ptr);

        printf("\nnode deleted\n");

    }

}

**void** display()

{

    struct node \*ptr;

    ptr=head;

**if**(head == NULL)

    {

        printf("\nnothing to print");

    }

**else**

    {

        printf("\n printing values ... \n");

**while**(ptr -> next != head)

        {

            printf("%d\n", ptr -> data);

            ptr = ptr -> next;

        }

        printf("%d\n", ptr -> data);

    }

}

**void** search()

{

    struct node \*ptr;

**int** item,i=0,flag=1;

    ptr = head;

**if**(ptr == NULL)

    {

        printf("\nEmpty List\n");

    }

**else**

    {

        printf("\nEnter item which you want to search?\n");

        scanf("%d",&item);

**if**(head ->data == item)

        {

        printf("item found at location %d",i+1);

        flag=0;

        }

**else**

        {

**while** (ptr->next != head)

        {

**if**(ptr->data == item)

            {

                printf("item found at location %d ",i+1);

                flag=0;

**break**;

            }

**else**

            {

                flag=1;

            }

            i++;

            ptr = ptr -> next;

        }

        }

**if**(flag != 0)

        {

            printf("Item not found\n");

        }

    }

}

Stack

**What is a Stack?**

A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle. Stack has one end, whereas the Queue has two ends (**front and rear**). It contains only one pointer **top pointer** pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack. In other words, a **stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.**

**Some key points related to stack**

It is called as stack because it behaves like a real-world stack, piles of books, etc.

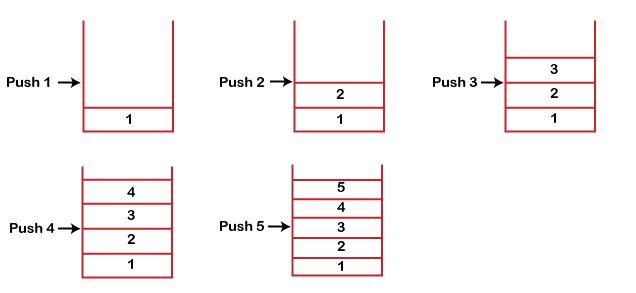
A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.

It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

**Working of Stack**

Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

**Standard Stack Operations**

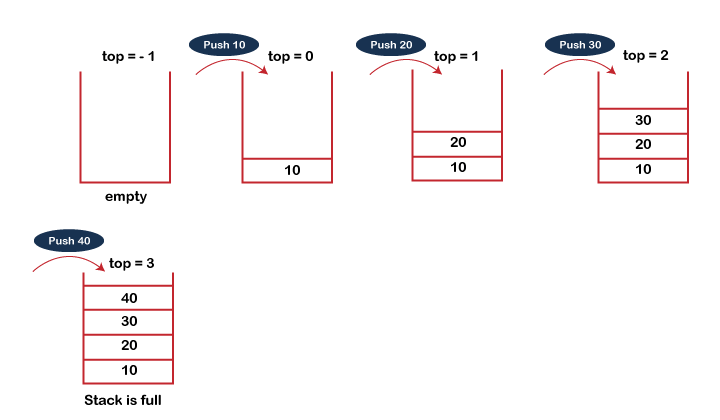
**The following are some common operations implemented on the stack:**

1. **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
2. **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. **isEmpty():** It determines whether the stack is empty or not.
4. **isFull():** It determines whether the stack is full or not.'
5. **peek():** It returns the element at the given position.
6. **count():** It returns the total number of elements available in a stack.
7. **change():** It changes the element at the given position.
8. **display():** It prints all the elements available in the stack.

**PUSH operation**

**The steps involved in the PUSH operation is given below:**

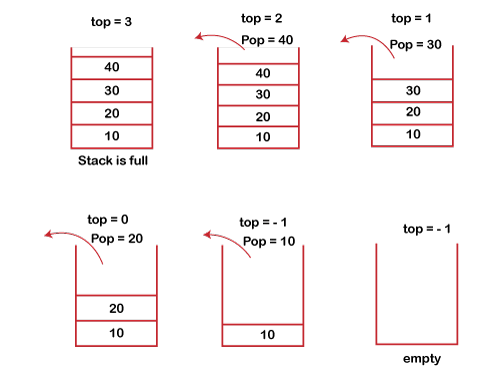
1. Before inserting an element in a stack, we check whether the stack is full.
2. If we try to insert the element in a stack, and the stack is full, then the **overflow** condition occurs.
3. When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
4. When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
5. The elements will be inserted until we reach the **max** size of the stack.



**POP operation**

**The steps involved in the POP operation is given below:**

1. Before deleting the element from the stack, we check whether the stack is empty.
2. If we try to delete the element from the empty stack, then the **underflow** condition occurs.
3. If the stack is not empty, we first access the element which is pointed by the **top**
4. Once the pop operation is performed, the top is decremented by 1, i.e., **top = top-1**.



**Applications of Stack**

**The following are the applications of the stack:**

1. **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:
   * + 1. **int** main()
       2. {
       3. cout<<"Hello";
       4. cout<<"javaTpoint";
       5. }
2. As we know, each program has an opening and closing braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.
3. **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack.  
   First, we push all the characters of the string in a stack until we reach the null character.  
   After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
4. **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.  
   If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
5. **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
6. **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
7. **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
8. **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
   * 1. Infix to prefix
     2. Infix to postfix
     3. Prefix to infix
     4. Prefix to postfix
     5. Postfix to infix
9. **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

2. Queue is referred to be as First In First Out list.

3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

OOPs Concepts in Java

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Complexity

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Time Complexity | | | | | | | | Space Compleity |
|  | **Average** | | | | **Worst** | | | | **Worst** |
|  | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |  |
| Queue | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

Types of Queues

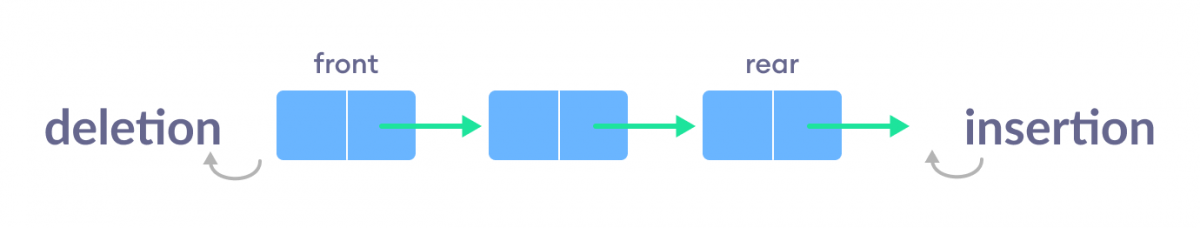
A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Double Ended Queue

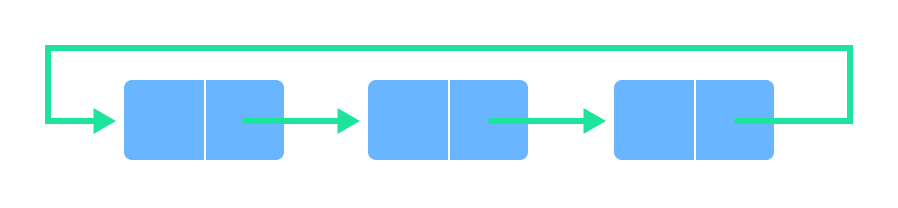
Simple Queue

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



Circular Queue

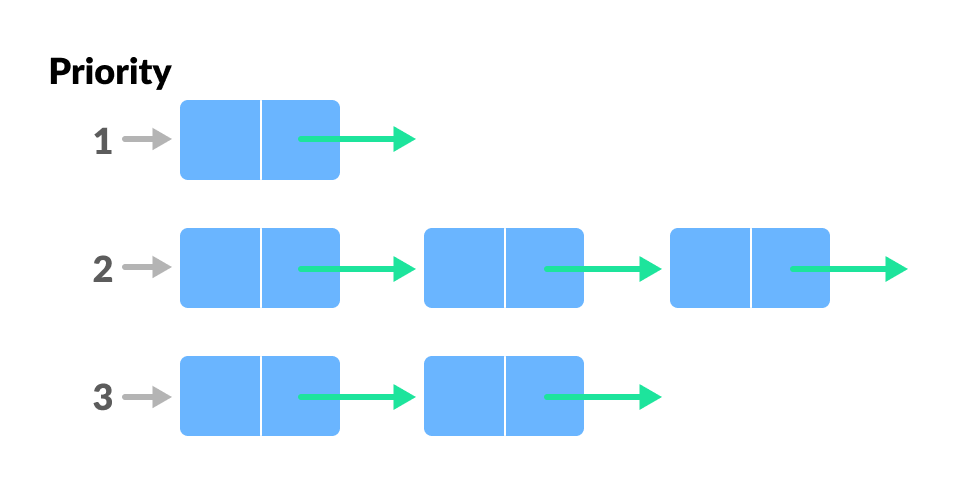
In a circular queue, the last element points to the first element making a circular link.



The main advantage of a circular queue over a simple queue is better memory utilization. If the last position is full and the first position is empty, we can insert an element in the first position. This action is not possible in a simple queue.

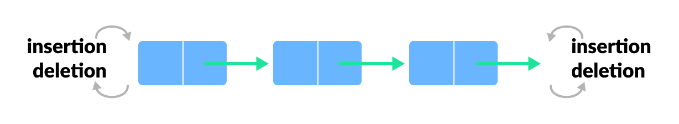
Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.



Deque (Double Ended Queue)

In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



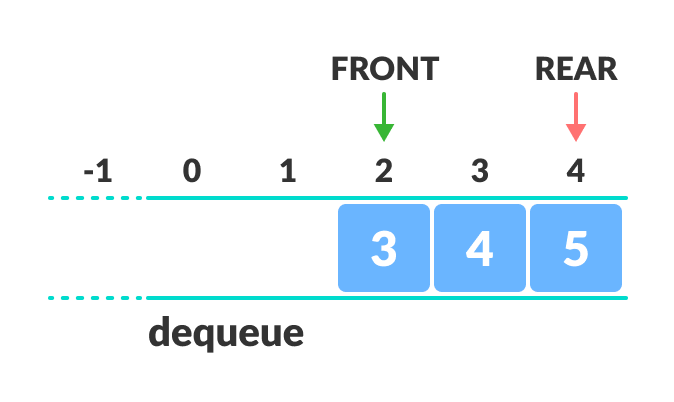
# Circular Queue Data Structure

In this tutorial, you will learn what a circular queue is. Also, you will find implementation of circular queue in C, C++, Java and Python.

A circular queue is the extended version of a [regular queue](https://www.programiz.com/data-structures/queue) where the last element is connected to the first element. Thus forming a circle-like structure.

Circular queue representation

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.

Limitation of the regular Queue

Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

## How Circular Queue Works

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

Here, the circular increment is performed by modulo division with the queue size. That is,

if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

## Circular Queue Operations

The circular queue work as follows:

* two pointers FRONT and REAR
* FRONT track the first element of the queue
* REAR track the last elements of the queue
* initially, set value of FRONT and REAR to -1

### 1. Enqueue Operation

* check if the queue is full
* for the first element, set value of FRONT to 0
* circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
* add the new element in the position pointed to by REAR

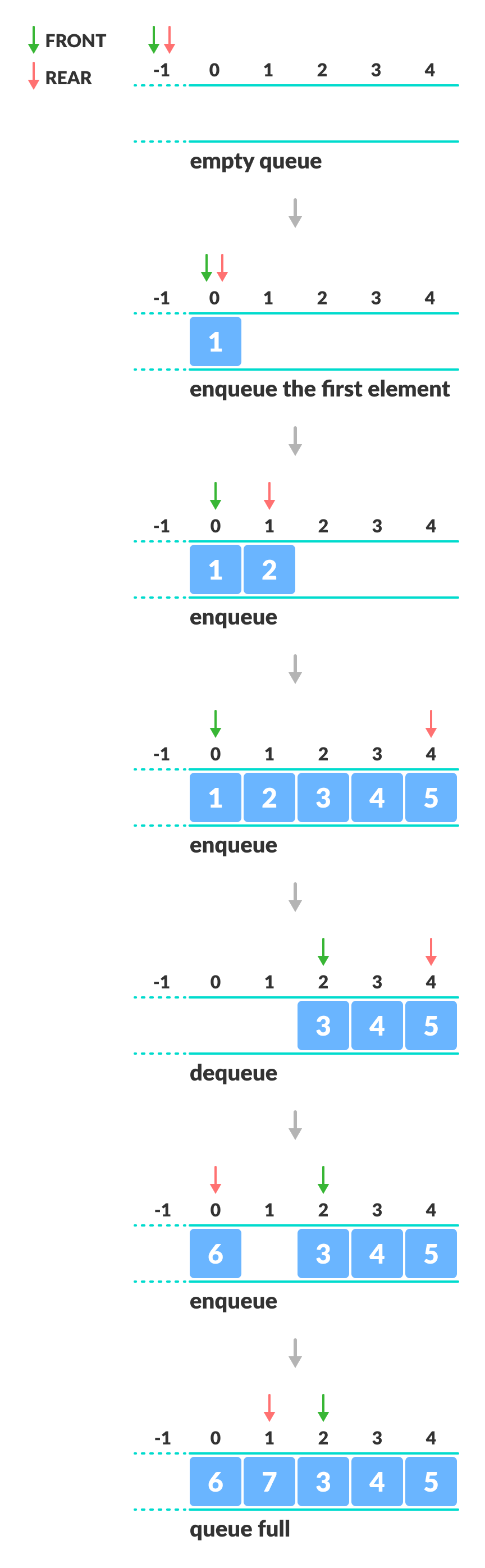
### 2. Dequeue Operation

* check if the queue is empty
* return the value pointed by FRONT
* circularly increase the FRONT index by 1
* for the last element, reset the values of FRONT and REAR to -1

However, the check for full queue has a new additional case:

* Case 1: FRONT = 0 && REAR == SIZE - 1
* Case 2: FRONT = REAR + 1

The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

Enque and Deque Operations

## Circular Queue Implementations in Python, Java, C, and C++

The most common queue implementation is using arrays, but it can also be implemented using lists.

[Python](https://www.programiz.com/dsa/circular-queue#python-code)

[Java](https://www.programiz.com/dsa/circular-queue#java-code)

[C](https://www.programiz.com/dsa/circular-queue#c-code)

[C+](https://www.programiz.com/dsa/circular-queue#cpp-code)

# Circular Queue implementation in Python

class MyCircularQueue():

def \_\_init\_\_(self, k):

self.k = k

self.queue = [None] \* k

self.head = self.tail = -1

# Insert an element into the circular queue

def enqueue(self, data):

if ((self.tail + 1) % self.k == self.head):

print("The circular queue is full\n")

elif (self.head == -1):

self.head = 0

self.tail = 0

self.queue[self.tail] = data

else:

self.tail = (self.tail + 1) % self.k

self.queue[self.tail] = data

# Delete an element from the circular queue

def dequeue(self):

if (self.head == -1):

print("The circular queue is empty\n")

elif (self.head == self.tail):

temp = self.queue[self.head]

self.head = -1

self.tail = -1

return temp

else:

temp = self.queue[self.head]

self.head = (self.head + 1) % self.k

return temp

def printCQueue(self):

if(self.head == -1):

print("No element in the circular queue")

elif (self.tail >= self.head):

for i in range(self.head, self.tail + 1):

print(self.queue[i], end=" ")

print()

else:

for i in range(self.head, self.k):

print(self.queue[i], end=" ")

for i in range(0, self.tail + 1):

print(self.queue[i], end=" ")

print()

# Your MyCircularQueue object will be instantiated and called as such:

obj = MyCircularQueue(5)

obj.enqueue(1)

obj.enqueue(2)

obj.enqueue(3)

obj.enqueue(4)

obj.enqueue(5)

print("Initial queue")

obj.printCQueue()

obj.dequeue()

print("After removing an element from the queue")

obj.printCQueue()

## Circular Queue Complexity Analysis

The complexity of the enqueue and dequeue operations of a circular queue is O(1) for (array implementations).

## Applications of Circular Queue

* CPU scheduling
* Memory management
* Traffic Management