UNIT-1

Introduction to Data Structures

1. Introduction to theory of data structures

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

Basic Terminology

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data: 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.

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.

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 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.

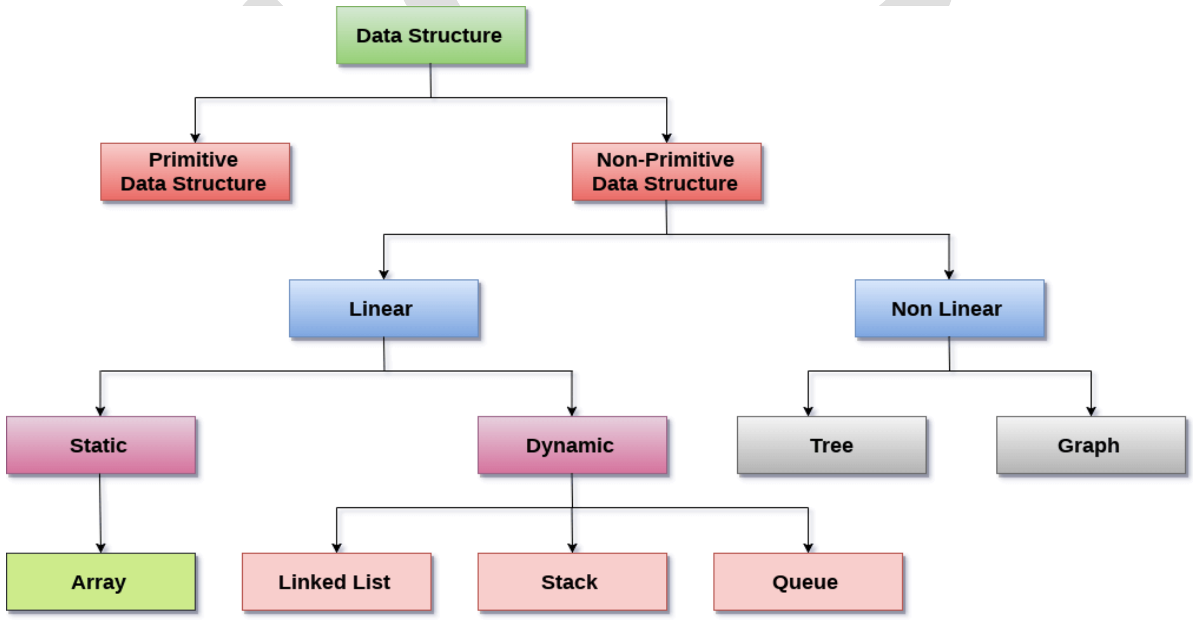
Advantages of Data Structures

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 particular 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.

2. CLASSIFICATION OF DATA STRUCTURES (OR) DATA STRUCTURE AND STRUCTURED TYPES



1. Linear Data Structures: A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element.

Types of Linear Data Structures are given below:

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].

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.

2. 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:

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.

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.

3. Operations on data structure:-

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subjects, we need to traverse the complete array of marks and calculate the total sum, and then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging.

Differences between the linear data structure and non-linear data structure:-

S.NO	Linear Data Structure	Non-linear Data Structure
1	In a linear data structure, data elements are arranged in a linear order where each and every element is attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not	While in a non-linear data

S.NO	Linear Data Structure	Non-linear Data Structure
	utilized in an efficient way.	structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

4. Data representation

Computer does not understand human language. Any data, viz., letters, symbols, pictures, audio, videos, etc., fed to computer should be converted to machine language first. Computers represent data in the following three forms −

Number System

We are introduced to concept of numbers from a very early age. To a computer, everything is a number, i.e., alphabets, pictures, sounds, etc., are numbers. Number system is categorized into four types −

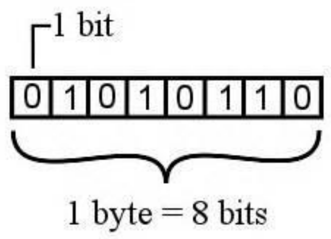
- Binary number system consists of only two values, either 0 or 1
- Octal number system represents values in 8 digits.
- Decimal number system represents values in 10 digits.
- Hexadecimal number system represents values in 16 digits.

Number System		
System	**Base**	**Digits**
Binary	2	0 1
Octal	8	0 1 2 3 4 5 6 7
Decimal	10	0 1 2 3 4 5 6 7 8 9
Hexadecimal	16	0 1 2 3 4 5 6 7 8 9 A B C D E F

Bits and Bytes

Bits − A bit is a smallest possible unit of data that a computer can recognize or use. Computer usually uses bits in groups.

Bytes − group of eight bits is called a byte. Half a byte is called a nibble.



The following table shows conversion of Bits and Bytes −

Byte Value	Bit Value
1 Byte	8 Bits
1024 Bytes	1 Kilobyte
1024 Kilobytes	1 Megabyte
1024 Megabytes	1 Gigabyte
1024 Gigabytes	1 Terabyte
1024 Terabytes	1 Petabyte
1024 Petabytes	1 Exabyte
1024 Exabytes	1 Zettabyte
1024 Zettabytes	1 Yottabyte
1024 Yottabytes	1 Brontobyte
1024 Brontobytes	1 Geopbytes

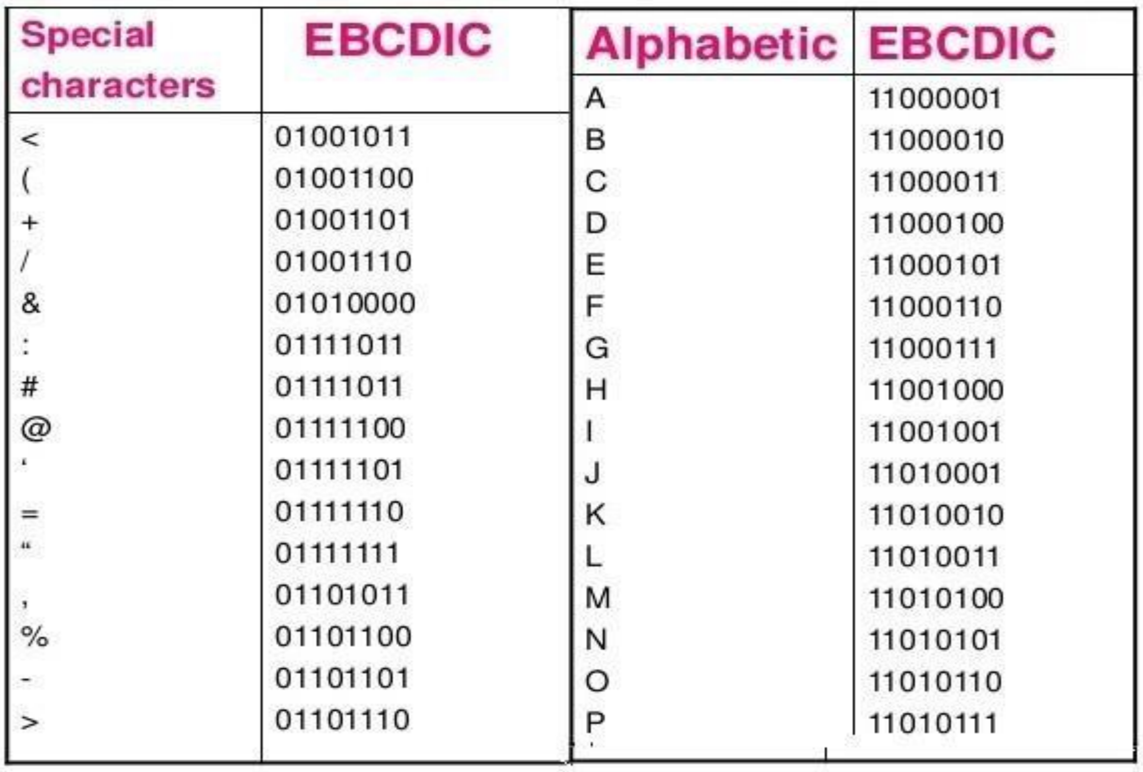
Text Code

Text code is format used commonly to represent alphabets, punctuation marks and other symbols. Four most popular text code systems are −

- EBCDIC
- ASCII
- Extended ASCII
- Unicode

EBCDIC

Extended Binary Coded Decimal Interchange Code is an 8-bit code that defines 256 symbols. Given below is the EBCDIC **Tabular column**



Special characters	EBCDIC
<	01001011
(	01001100
+	01001101
/	01001110
&	01010000
:	01111011
#	01111011
@	01111100
'	01111101
=	01111110
"	01111111
,	01101011
%	01101100
-	01101101
>	01101110

Alphabetic	EBCDIC
A	11000001
B	11000010
C	11000011
D	11000100
E	11000101
F	11000110
G	11000111
H	11001000
I	11001001
J	11010001
K	11010010
L	11010011
M	11010100
N	11010101
O	11010110
P	11010111

ASCII

American Standard Code for Information Interchange is an 8-bit code that specifies character values from 0 to 127.

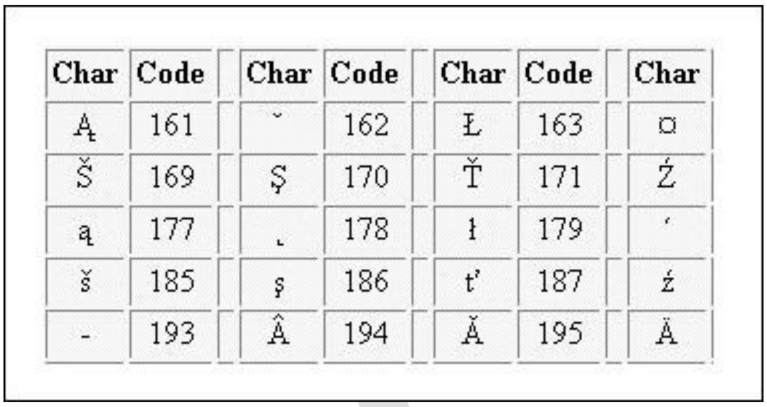
ASCII Tabular column

ASCII Code	Decimal Value	Character
0000 0000	0	Null prompt
0000 0001	1	Start of heading
0000 0010	2	Start of text
0000 0011	3	End of text
0000 0100	4	End of transmit
0000 0101	5	Enquiry
0000 0110	6	Acknowledge
0000 0111	7	Audible bell
0000 1000	8	Backspace
0000 1001	9	Horizontal tab
0000 1010	10	Line Feed

Extended ASCII

Extended American Standard Code for Information Interchange is an 8-bit code that specifies character values from 128 to 255.

Extended ASCII Tabular column

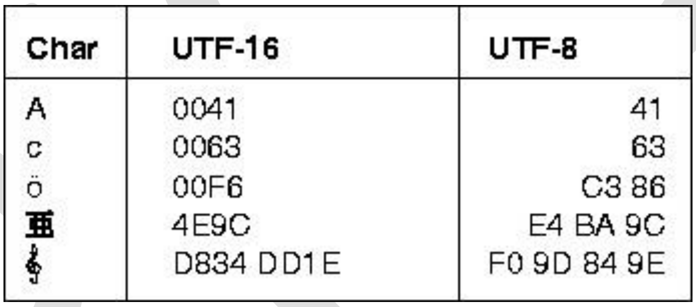


Char	Code	Char	Code	Char	Code	Char
Ą	161	˘	162	Ł	163	¤
Š	169	Ş	170	Ť	171	Ź
ą	177	˛	178	ł	179	´
š	185	ş	186	ť	187	ź
-	193	Â	194	Ă	195	Ä

Unicode

Unicode Worldwide Character Standard uses 4 to 32 bits to represent letters, numbers and symbol.

Unicode Tabular Column



Char	UTF-16	UTF-8
A	0041	41
c	0063	63
ö	00F6	C3 86
亜	4E9C	E4 BA 9C
𝄞	D834 DD1E	F0 9D 84 9E

5. Abstract data types

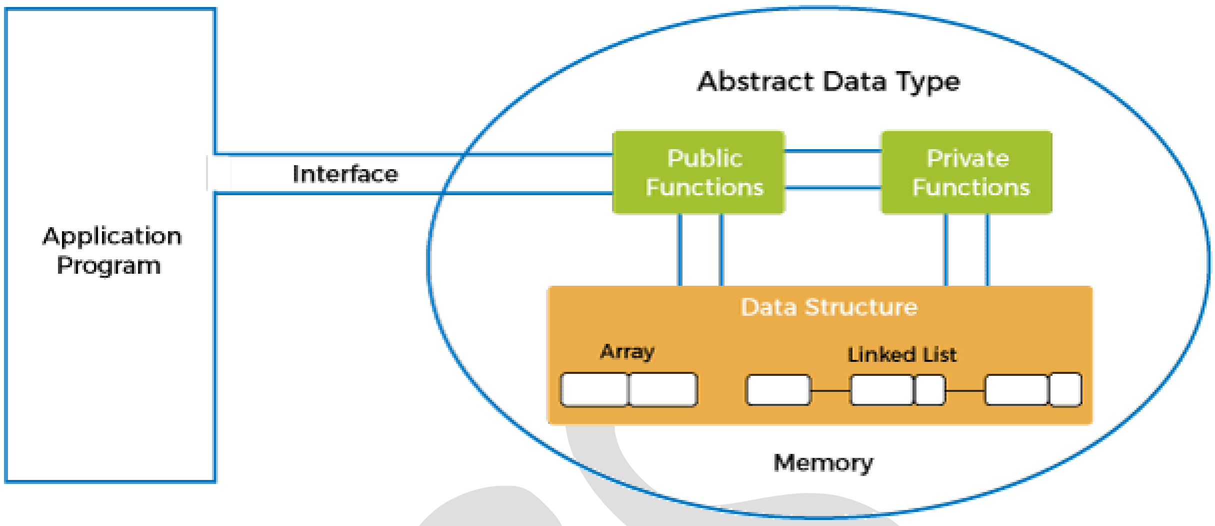
An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.
In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.
For example, a List is an abstract data type that is implemented using a dynamic array and linked list. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue. A Map is implemented using Tree map, hash map, or hash table.

Abstract data type model

Before knowing about the abstract data type model, we should know about abstraction and encapsulation.

Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.

Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program

Example:

Stack ADT: Here, the stack consists of elements of the same type arranged in sequential order. The following are the operations that can be performed on the stack are:

- initialize(): This method initializes the stack to be empty.
- push(): It is a method used for inserting an element into the stack.
- pop(): It is a method used for removing the element from the stack.
- isEmpty(): This method is used to check whether the stack is empty or not.
- isfull(): It checks whether the stack is full or not.

6. PRIMITIVE DATA TYPES

Primitive is the most fundamental data type usable in the Programming language. There are eight primitive data types: Boolean, byte, character, short, int, long, float, and double. In a Programming language, these data types serve as the foundation for data manipulation.

All basic data types are built-in into the majority of programming languages. Furthermore, many languages provide a set of composite data types. Primitive data types may or may not have a one-to-one correspondence with objects in the computer's memory, depending on the language and its implementation. However, operations on basic primitive data types are typically thought to be the fastest language constructs.

For example, integer addition can be performed as a single machine instruction, and some processors provide specific instructions for processing character sequences with a single instruction. The C standard specifically states that "a 'plain' int object has the natural size suggested by the execution environment's architecture". On a 32-bit architecture, this means that int will most likely be 32 bits long. Value types are always basic primitive types.

Most programming languages do not allow programmes to change the behaviour or capabilities of primitive (built-in or basic) data types. Smalltalk is an exception, allowing all data types to be extended within a programme, expanding the operations that can be performed on them or even redefining the built-in operations.

Such data types serve a single purpose: they contain pure, simple values of a type. Because these data types are defined by default in the Programming languages type system, they come with a set of predefined operations. Such primitive types cannot have new operations defined. There are three more types of primitives in the Java type system:

Short, int, long, float, and double are the numerical primitives. These primitive data types can only store numbers. Simple arithmetic (addition, subtraction, etc.) or comparison operations are associated with such data types (greater than, equal to, etc.)

Textual primitives include bytes and characters. Characters are stored in these primitive data types (that can be Unicode alphabets or even numbers). Textual manipulation operations are associated with data types (comparing two words, joining characters to make words, etc.). However, byte and char can perform arithmetic operations as well.

Primitives with boolean and null values: boolean and null.

7. Data type v/s data structure

Data Types	Data Structures
Data type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

Principles of programming and analysis of Algorithms

8. What is an 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 Pseudo code.

Characteristics of an Algorithm

The following are the characteristics of an algorithm:

- **Input:** An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output:** We will get 1 or more output at the end of an algorithm.
- **Unambiguity:** An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness:** An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent:** An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Dataflow of an Algorithm

- **Problem:** A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- **Algorithm:** An algorithm will be designed for a problem which is a step by step procedure.
- **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit:** The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output:** The output is the outcome or the result of the program.

Need of Algorithms

We need algorithms because of the following reasons:

- **Scalability:** It helps us to understand the scalability. When we have a big real-world problem, we need to scale it down into small-small steps to easily analyze the problem.
- **Performance:** The real-world is not easily broken down into smaller steps. If the problem can be easily broken into smaller steps means that the problem is feasible.

Example of an algorithm in programming:-

We will write an algorithm to add two numbers entered by the user.

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

9. Different Approaches of an Algorithm:-

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

- **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.
- **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.
- **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.
- **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:
  1. It breaks down the problem into a sub problem to find the optimal solution.

2. After breaking down the problem, it finds the optimal solution out of these sub problems.
3. Stores the result of the sub problems is known as memorization.
4. Reuse the result so that it cannot be recomputed for the same sub problems.
5. 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. 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:

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

10. Algorithm Analysis

The algorithm can be analyzed in two levels, i.e., first is before creating the algorithm, and second is after creating the algorithm. The following are the two analysis of an algorithm:

- **Priori Analysis**: Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- **Posterior Analysis**: Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any

programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

11. 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;
// Suppose we have to calculate the sum of n numbers.
for i=1 to n
sum=sum+i;
// when the loop ends then sum holds the sum of the n numbers
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.

So,

Space complexity = Auxiliary space + Input size.

12. Asymptotic Notations

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

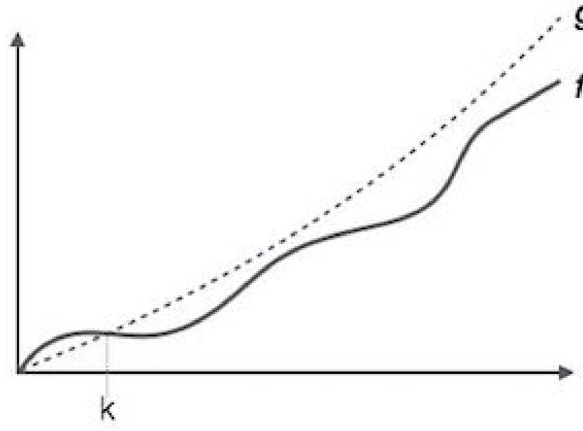
- **Big oh Notation (?)**

- Omega Notation (Ω)
- 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:

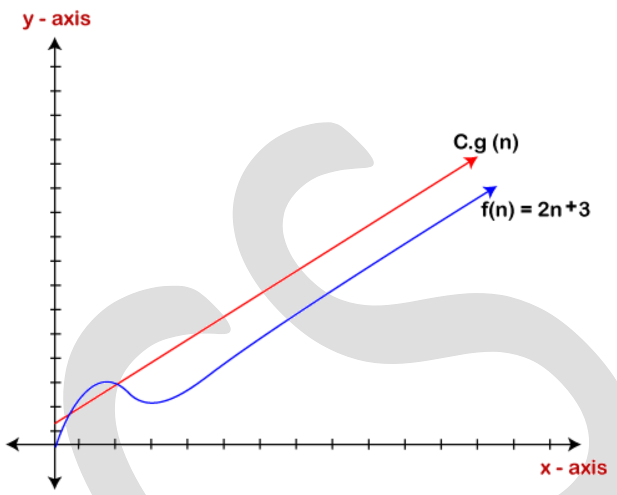


For example:
If **f(n)** and **g(n)** are the two functions defined for positive integers,
then **f(n) = O(g(n))** as **f(n) is big oh of g(n)** or f(n) is on the order of g(n)) if there exists constants c and no such that:
$f(n) \leq c.g(n)$ for all $n \geq n_o$
This implies that f(n) does not grow 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 have to 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 n, 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.

ARRAYS IN DATA STRUCTURES

ARRAY DEFINITION

- 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 int, char, double, float, etc.
- Array is the simplest data structure where each data element can be randomly accessed by using its index number.
- For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject. Instead of that, we can define an array which can store the marks in each subject at contiguous memory locations.

The array **marks[10]** defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. **marks[0]** denotes the marks in first subject, **marks[1]** denotes the marks in 2nd subject and so on.

Properties of the Array

1. Each element is of same data type and carries a same size i.e. int = 4 bytes.
2. Elements of the array are stored at contiguous memory locations where the first element is stored at the smallest memory location.
3. Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of data element.

for example, in C language, the syntax of declaring an array is like following:

1. int arr[10]; char arr[10]; float arr[5]

Program without array:

```
 #include <stdio.h>
 void main ()
 {
 int marks_1 = 56, marks_2 = 78, marks_3 = 88, marks_4 = 76, marks_5 = 56, marks_6 = 89;
 float avg = (marks_1 + marks_2 + marks_3 + marks_4 + marks_5 +marks_6) / 6 ;
 printf(avg);
 }
```

Program by using array:

```
    #include <stdio.h>
     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

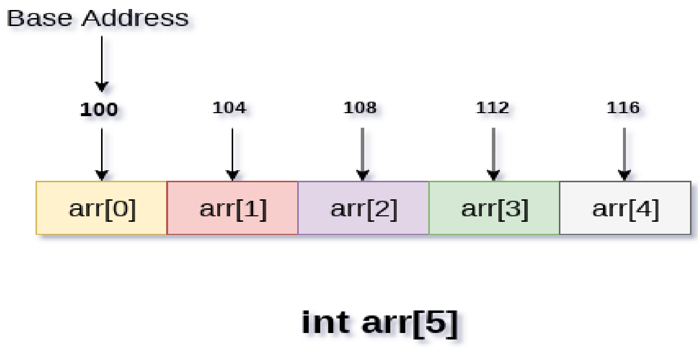
- ❖ Array provides the single name for the group of variables of the same type therefore; it is easy to remember the name of all the elements of an array.
- ❖ Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.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

Memory Allocation of the array

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.

- 0 (zero - based indexing) : The first element of the array will be arr[0].
- 1 (one - based indexing) : The first element of the array will be arr[1].
- 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. The following example illustrates how the array can be passed to a function.

Example:

```
#include <stdio.h>
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.

Types of Arrays

There are two types of Arrays

- One Dimensional Arrays
- Two Dimensional Arrays

One Dimensional Array in C:

One dimensional array is an array that has only one subscript specification that is needed to specify a particular element of an array. A one-dimensional array is a structured collection of components (often called array elements) that can be accessed individually by specifying the position of a component with a single index value.

Syntax: data-type arr_name[array_size];

Rules for Declaring One Dimensional Array

1. An array variable must be declared before being used in a program.
2. The declaration must have a data type (int, float, char, double, etc.), variable name, and subscript.
3. The subscript represents the size of the array. If the size is declared as 10, programmers can store 10 elements.
4. An array index always starts from 0. For example, if an array variable is declared as s[10], then it ranges from 0 to 9.

5. Each array element stored in a separate memory location.

Initialization of One-Dimensional Array in C

An array can be initialized at either following states:

1. At compiling time (static initialization)
2. Dynamic Initialization

Compiling time initialization:

The compile-time initialization means the array of the elements are initialized at the time the program is written or array declaration.

Syntax: data_type array_name [array_size]=(list of elements of an array);
Example: int n[5]={0, 1, 2, 3, 4};

Program:

```
#include<stdio.h>
int main()
{
 int n[5]={0, 1, 2, 3, 4};
 printf("%d", n[0]);
 printf("%d", n[1]);
 printf("%d", n[2]);
 printf("%d", n[3]);
 printf("%d", n[4]);
}
```

Output: 0 1 2 3 4

Two dimensional (2D) arrays

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns. Before we discuss more about two Dimensional array let's have a look at the following C program.

Simple Two dimensional (2D) Array Example

This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int disp[2][3];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("Enter value for disp[%d][%d]:", i, j);
      scanf("%d", &disp[i][j]);
    }
  }
  //Displaying array elements
  printf("Two Dimensional array elements:\n");
  for(i=0; i<2; i++) {
    for(j=0;j<3;j++) {
      printf("%d ", disp[i][j]);
      if(j==2){
        printf("\n");
      }
    }
  }
  return 0;
}
```

Output:

```
Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6
```

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```
int disp[2][4] = {
   {10, 11, 12, 13},
   {14, 15, 16, 17}
};
OR
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, I recommend you to use the first method as it is more readable, because you can visualize the rows and columns of 2d array in this method.

Things that you must consider while initializing a 2D array

We already know, when we initialize a normal array during declaration, we need not to specify the size of it. However that’s not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration. Let’s understand this with the help of few examples –

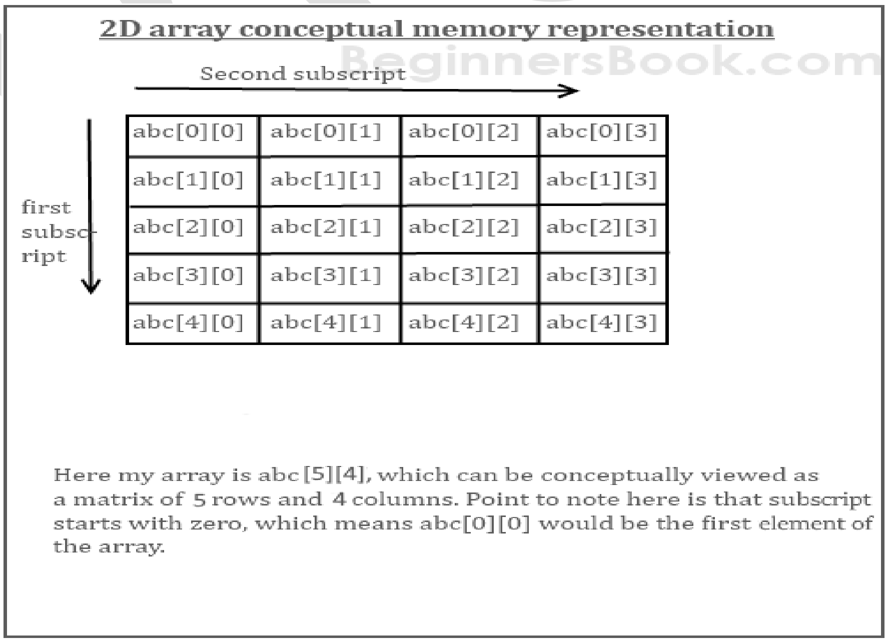
```
/* Valid declaration*/
int abc[2][2] = {1, 2, 3 ,4 }
/* Valid declaration*/
int abc[][2] = {1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[][] = {1, 2, 3 ,4 }
/* Invalid because of the same reason mentioned above*/
int abc[2][] = {1, 2, 3 ,4 }
```

How to store user input data into 2D array

We can calculate how many elements a two dimensional array can have by using this formula:The array arr[n1][n2] can have n1*n2 elements. The array that we have in the example below is having the dimensions 5 and 4. These dimensions are known as subscripts. So this array has **first subscript** value as 5 and **second subscript** value as 4.
So the array abc[5][4] can have 5*4 = 20 elements.
To store the elements entered by user we are using two for loops, one of them is a nested loop. The outer loop runs from 0 to the (first subscript -1) and the inner for loops runs from 0 to the (second subscript -1). This way the the order in which user enters the elements would be abc[0][0], abc[0][1], abc[0][2]…so on.

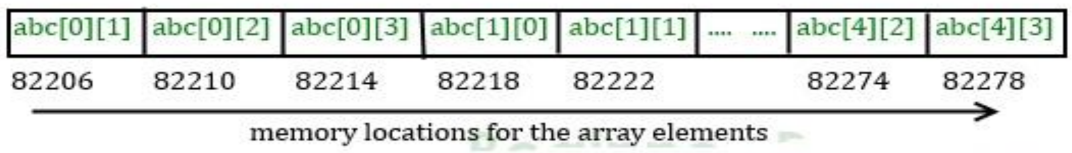
```
#include<stdio.h>
int main(){
  /* 2D array declaration*/
  int abc[5][4];
  /*Counter variables for the loop*/
  int i, j;
  for(i=0; i<5; i++) {
    for(j=0;j<4;j++) {
      printf("Enter value for abc[%d][%d]:", i, j);
      scanf("%d", &abc[i][j]);
    }
  }
  return 0;
}
```

In above example, I have a 2D array abc of integer type. Conceptually you can visualize the above array like this:


2D array conceptual memory representation
BeginnersBook.com
Second subscript
first subscript
abc[0][0] abc[0][1] abc[0][2] abc[0][3]
abc[1][0] abc[1][1] abc[1][2] abc[1][3]
abc[2][0] abc[2][1] abc[2][2] abc[2][3]
abc[3][0] abc[3][1] abc[3][2] abc[3][3]
abc[4][0] abc[4][1] abc[4][2] abc[4][3]

Here my array is abc[5][4], which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first element of the array.

However the actual representation of this array in memory would be something like this:



Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguos locations, so that you can understand that the address difference between each element is equal to the size of one element(int size 4). For better understanding see the program below.

Actual memory representation of a 2D array

ARRAY OPERATIONS IN DATA STRUCTURES

The basic operations supported by an array are:

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then the default values are assigned to the elements in the order as shown below:

Data Type	Default Value
bool	False
char	0
int	0
float	0.0
double	0.0f
void	0

Traversing in Linear Array

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms.

Traverse – print all the array elements one by one or process the each element one by one. Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A with given property.

This can be accomplished by traversing A, that is, by accessing and processing (frequently called visiting) each element of an exactly once.

Definition of traversing

In traversing operation of an array, each element of an array is accessed exactly for once for processing.

This is also called visiting of an array.

Algorithm

```
Step 1 :   [Initialization]  Set I = LB
Step 2 :    Repeat Step 3 and Step 4 while I  < = UB
step 3 :     [ processing ] Process the A[I] element
Step 4 :    [ Increment the counter ] I = I + 1
            [ End of the loop of step 2 ]
Step 5:     Exit
```

(*Here LB is lower Bound and UB is Upper Bound* A[ ] is linear array)

Example

Let LA is a Linear Array (unordered) with N elements.

```
//Write a Program which perform traversing operation.
#include <stdio.h>
void main()
{
  int LA[] = {2,4,6,8,9};
  int i, n = 5;
  printf("The array elements are:\n");
  for(i = 0; i < n; i++)
  {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

```
The array elements are:
LA[0] = 2
LA[1] = 4
LA[2] = 6
LA[3] = 8
LA[4] = 9
```

Insertion Operation

Inserting one or more data elements into an array is known as Insert operation. A new element is either added at beginning, end or at any given index of array, based on the requirement.

It is better understood with the practical implementation of insertion operation, where the data is added at the end of the array.

Algorithm

Let Array be a linear unordered array of **MAX** elements.

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA –

```
1. Start
2. Set J = N
3. Set N = N+1
```

```
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
int LA[] = {1,3,5,7,8};
int item = 10, k = 3, n = 5;
int i = 0, j = n;
printf("The original array elements are :\n ");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
n = n + 1;
while( j >= k) {
LA[j+1] = LA[j];
j = j - 1;
}
LA[k] = item;
printf("The array elements after insertion : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
}
```

When the above program is compiled and executed, the output produced is as follows:

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion Operation

To remove an existing element from the array and to re-organize the other elements of an array, is known as Deletion operation.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
int LA[] = {1,3,5,7,8};
int k = 3, n = 5;
int i, j;
printf("The original array elements are : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
j = k;
while( j < n) {
LA[j-1] = LA[j];
j = j + 1;
}
n = n -1;
printf("The array elements after deletion : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
}
```

When the above program is compiled and executed, the output produced is as follows:

Output

```
The original array elements are:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion:
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

Search Operation

A search is performed for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
int LA[] = {1,3,5,7,8};
int item = 5, n = 5;
int i = 0, j = 0;
printf("The original array elements are : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
while( j < n){
if( LA[j] == item ) {
break;
}
j = j + 1;
}
printf("Found element %d at position %d \n", item, j+1);
}
```

When the above program is compiled and executed, the output produced is as follows:

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

Update Operation

Updating of an existing element from the array at a given index is known as Update Operation.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

Example

Following is the implementation of the above algorithm −

```
#include <stdio.h>
main() {
int LA[] = {1,3,5,7,8};
int k = 3, n = 5, item = 10;
int i, j;
printf("The original array elements are : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
LA[k-1] = item;
printf("The array elements after updation : \n");
for(i = 0; i<n; i++) {
printf("LA[%d] = %d \n ", i, LA[i]);
}
}
```

When the above program is compiled and executed, the output produced is as follows:

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

Multi Dimensional Array in C

An Array having more than one dimension is called Multi Dimensional array in C.

In C Programming Language, by placing n number of brackets [ ], we can declare n-dimensional array where n is dimension number. For example,

int a[2][3][4] = Three Dimensional Array

int a[2][2][3][4] = Four Dimensional Array

Syntax of a Multi Dimensional Array in C Programming

The basic syntax or, the declaration of multi dimensional array in C Programming is

Data_Type Array_Name[Tables][Row_Size][Column_Size]

- **Data_type:** It will decide the type of elements it will accept. For example, If we want to store integer values then we declare the Data Type as int, If we want to store Float values then we declare the Data Type as float etc
- **Array_Name:** This is the name you want to give it to Multi Dimensional array in C.
- **Tables:** It will decide the number of tables an array can accept. Two Dimensional Array is always a single table with rows and columns. In contrast, Multi Dimensional array in C is more than 1 table with rows and columns.
- **Row_Size:** Number of Row elements an array can store. For example, Row_Size =10, the array will have 10 rows.
- **Column_Size:** Number of Column elements an array can store. For example, Column_Size = 8, the array will have 8 Columns.

We can calculate the maximum number of elements in a Three Dimensional using: [Tables] * [Row_Size] * [Column_Size]

For Example

int Employees[2][4][3];

1. Here, we used int as the data type to declare an array. So, the above array will accept only integers. If you try to add float values, it throws an error.
2. Employees is the array name
3. The number of tables = 2. So, this array will hold a maximum of 2 levels of data (rows and columns).
4. The Row size of an Array is 4. It means Employees array only accept 4 integer values as rows.
   - If we try to store more than 4, it throws an error.
   - We can store less than 4. For Example, If we store 2 integer values, the remaining two will assign with the default value (Which is 0).
5. The Column size of an Array is 3. It means Employees array will only accept 3 integer values as columns.
   - If we try to store more than 3, it throws an error.
   - We can store less than 3. For Example, If we store 1 integer values, the remaining 2 values will assign with the default value (Which is 0).
6. Finally, Employees array can hold a maximum of 24 integer values (2 * 4 * 3 = 24).

C Multi Dimensional Array Initialization

We can initialize the C Multi Dimensional Array in **C**

int Employees[2][4][3] = { { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },{ {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} } };

Here, We have 2 tables and the 1st table holds 4 Rows * 3 Columns, and the 2nd table also holds 4 Rows * 3 Columns

The first three elements of the first table will be 1st row, the second three elements will be 2nd row, the next three elements will be 3rdrow, and the last 3 elements will be 4th row. Here we divided them into 3 because our column size = 3, and we surrounded each row with curly braces ({}). It is always good practice to use the curly braces to separate the rows. Same for the second table.

Accessing Multi Dimensional Array in C

We can access the C Multi Dimensional array elements using indexes. Index starts at 0 and ends at n-1, where n is the size of a row or column.

For example, if an Array_name[4][8][5] will store 8-row elements and 5 column elements in each table where table size = 4. To access 1st value of the 1st table, use Array_name[0][0][0], to access 2nd row 3rd column value of the 3rd table then use Array_name[2][1][2] and to access the 8th row 5th column of the last table (4th table), use Array_name[3][7][4]. Lets see the example of C Multi Dimensional Array for better understanding:

```
int Employees[2][4][3] = { {10, 20, 30}, {15, 25, 35}, {22, 44, 66}, {33, 55, 77} },
                        { {1, 2, 3}, {5, 6, 7}, {2, 4, 6}, {3, 5, 7} }
                       };
```

//To Access the values in the Employees[2][4][3] array
//Accessing First Table Rows & Columns

```
Printf("%d", Employees[0][0][0]) = 10
Printf("%d", Employees[0][0][1]) = 20
Printf("%d", Employees[0][0][2]) = 30
Printf("%d", Employees[0][1][0]) = 15
Printf("%d", Employees[0][1][1]) = 25
Printf("%d", Employees[0][1][2]) = 35
Printf("%d", Employees[0][2][0]) = 22
Printf("%d", Employees[0][2][1]) = 44
Printf("%d", Employees[0][2][2]) = 66
Printf("%d", Employees[0][3][0]) = 33
Printf("%d", Employees[0][3][1]) = 55
Printf("%d", Employees[0][3][2]) = 77
```

//Accessing Second Table Rows & Columns

```
Printf("%d", Employees[1][0][0]) = 1
Printf("%d", Employees[1][0][1]) = 2
Printf("%d", Employees[1][0][2]) = 3
Printf("%d", Employees[1][1][0]) = 5
Printf("%d", Employees[1][1][1]) = 6
Printf("%d", Employees[1][1][2]) = 7
Printf("%d", Employees[1][2][0]) = 2
Printf("%d", Employees[1][2][1]) = 4
Printf("%d", Employees[1][2][2]) = 6
Printf("%d", Employees[1][3][0]) = 3
Printf("%d", Employees[1][3][1]) = 5
Printf("%d", Employees[1][3][2]) = 7
```

//To Alter the values in the Employees[4][3] array

```
Employees[0][2][1] = 98; // It will change the value of Employees[0][2][1] from 44 to 98
Employees[1][2][2] = 107; // It will change the value of Employees[1][2][2] from 6 to 107
```

Multi Dimensional Array in C Example

In this C program, we will declare three dimensional array and initialize it with some values. Using the for loop, we will display every individual value present in the array as per the index.

```
/* Example for Multi Dimensional Array in C programming */
#include<stdio.h>
```

```
int main()
{
 int tables, rows, columns;
 int Employees[2][2][3] = { { {9, 99, 999}, {8, 88, 888} },
                          { {225, 445, 665}, {333, 555, 777} }  };
  for (tables = 0; tables < 2; tables++)
 {
   for (rows = 0; rows < 2; rows++)
  {
    for (columns =0; columns < 3; columns++)
   {
      printf("Employees[%d][%d][%d] = %d\n", tables, rows, columns,
                              Employees[tables][rows][columns]);
   }
  }
 }
 return 0;
}
```

Linked Lists

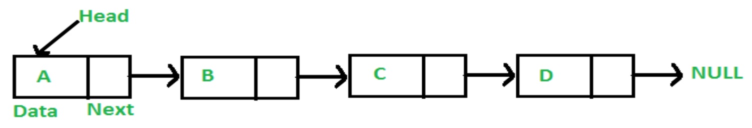
1. Introduction to lists and linked lists

We have studied that an array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. Elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

Definition of Linked Lists

A linked list, in simple terms, is a linear collection of data elements. These data elements are called *nodes*. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.



In the above fig, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node. The left part of the node which contains data may include a simple data type, an array, or a structure. The right part of the node contains a pointer to the next node (or address of the next node in sequence). The last node will have no next node connected to it, so it will store a special value called NULL. In Fig, the NULL pointer is represented by X. While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list. Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a *self-referential data type.*

Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code:

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

2. Dynamic memory allocation

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

1. malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Example:

```
ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
```

2. calloc() function in C

The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero.
It returns NULL if memory is not sufficient.
The syntax of calloc() function is given below:
ptr=(cast-type*)calloc(number, byte-size)
example
ptr=(int*)calloc(n,sizeof(int)); //memory allocated using calloc

3. realloc() function in C

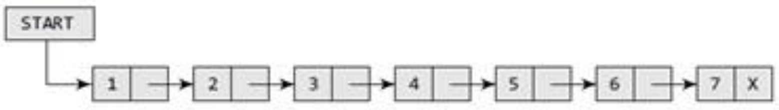
If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.
Let's see the syntax of realloc() function.
ptr=realloc(ptr, new-size)

4. free() function in C

The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.
Let's see the syntax of free() function.
free(ptr)

3. SINGLY LINKED LIST

A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type. By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way.



TRAVERSING A LINKED LIST

Traversing a linked list means accessing the nodes of the list in order to perform some processing on them. Remember a linked list always contains a pointer variable START which stores the address of the first node of the list. End of the list is marked by storing NULL or –1 in the NEXT field of the last node. For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. The algorithm to traverse a linked list is shown in Fig

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:             Apply Process to PTR -> DATA
Step 4:             SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: EXIT
```

Fig: Algorithm for traversing a linked list

In this algorithm, we first initialize PTR with the address of START. So now, PTR points to the first node of the linked list. Then in Step 2, a while loop is executed which is repeated till PTR processes the last node, that is until it encounters NULL. In Step 3, we apply the process (e.g., print) to the current node, that is, the node pointed by PTR. In Step

4, we move to the next node by making the PTR variable point to the node whose address is stored in the NEXT field.

Let us now write an algorithm to count the number of nodes in a linked list. To do this, we will traverse each and every node of the list and while traversing every individual node, we will increment the counter by 1. Once we reach NULL, that is, when all the nodes of the linked list have been traversed, the final value of the counter will be displayed. Below fig shows the algorithm to print the number of nodes in a linked list.

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:            SET COUNT = COUNT + 1
Step 5:            SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

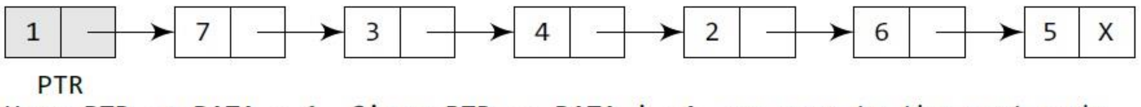
Fig: Algorithm to print the number of nodes in a linked list

Searching for a Value in a Linked List

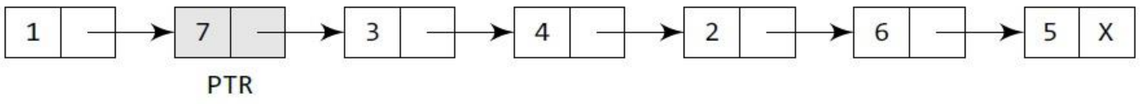
Searching a linked list means to find a particular element in the linked list. As already discussed, a linked list consists of nodes which are divided into two parts, the information part and the next part. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3:      IF VAL = PTR -> DATA
                    SET POS = PTR
                    Go To Step 5
             ELSE
                    SET PTR = PTR -> NEXT
             [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

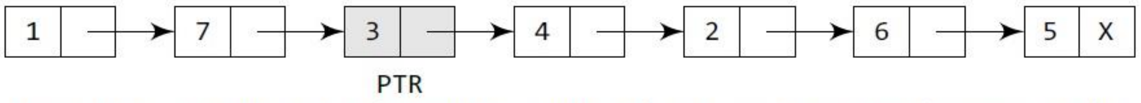
The above algorithm shows **to search a linked list**. In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every node's DATA with VAL for which the search is being made. If the search is successful, that is, VAL has been found, and then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list. Consider the linked list shown in Fig. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the following figure.



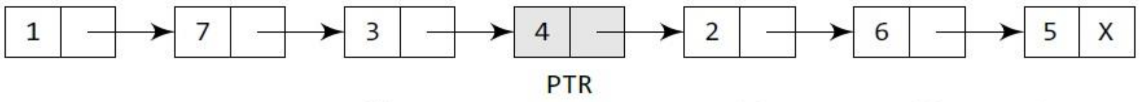
Here PTR -> DATA = 1. Since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 7. Since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 3. Since PTR -> DATA != 4, we move to the next node.



Here PTR -> DATA = 4. Since PTR -> DATA = 4, POS = PTR. POS now stores the address of the node that contains VAL

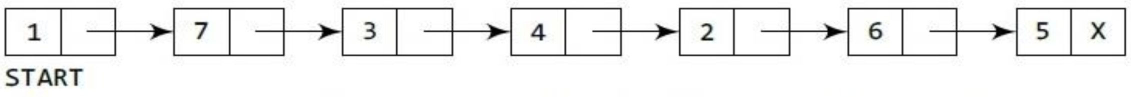
Inserting a New Node in a Linked List

In this section, we will see how a new node is added into an already existing linked list. We will take four cases and then see how insertion is done in each case.

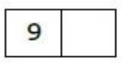
Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.
Before we describe the algorithms to perform insertions in all these four cases, let us first discuss an important term called OVERFLOW. Overflow is a condition that occurs when AVAIL = NULL or no free memory cell is present in the system. When this condition occurs, the program must give an appropriate message.

1. Inserting a Node at the Beginning of a Linked List

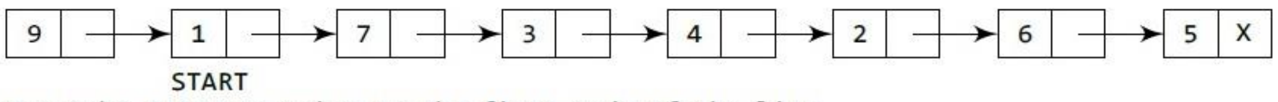
Consider the linked list shown in below. Suppose we want to add a new node with data 9 and add it as the first node of the list. Then the following changes will be done in the linked list.



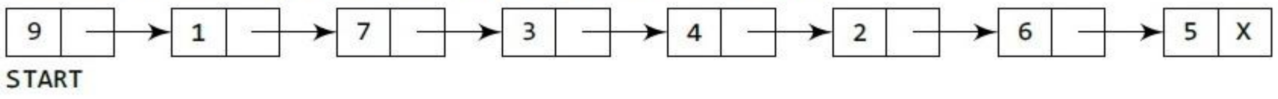
Allocate memory for the new node and initialize its DATA part to 9.



Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



Now make START to point to the first node of the list.



The algorithm shows, to insert a new node at the beginning of a linked list.

Step 1: IF AVAIL = NULL

```
Write OVERFLOW
Go to Step 7
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL NEXT
Step 4: SET DATA = VAL
Step 5: SET NEW_NODE NEXT = START
Step 6: SET START = NEW_NODE
Step 7: EXIT
```

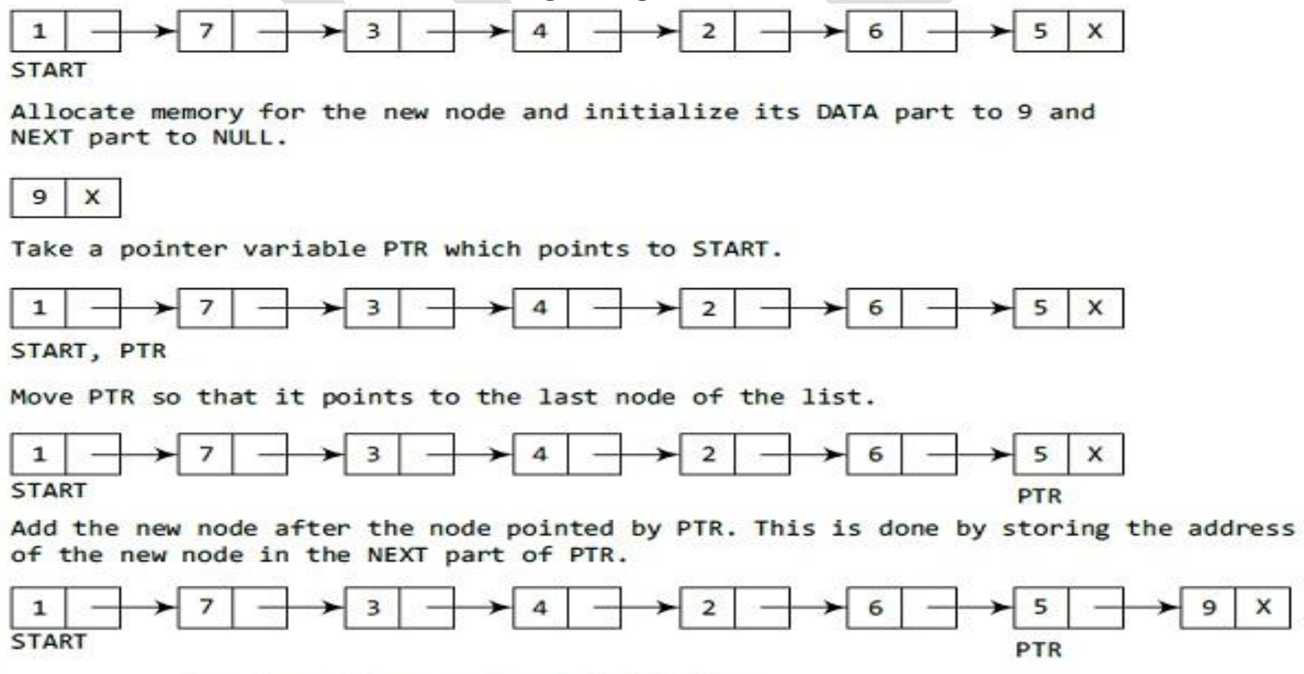
In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if a free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the next part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE. Note the following two steps:

```
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
```

These steps allocate memory for the new node. In C, there are functions like malloc(), alloc, and calloc() which automatically do the memory allocation on behalf of the user.

2. Inserting a Node at the End of a Linked List

Consider the linked list shown in Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.



Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

Take a pointer variable PTR which points to START.

Move PTR so that it points to the last node of the list.

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

Inserting an element at the end of a linked list

The following figure shows the algorithm to insert a new node at the end of a linked list.

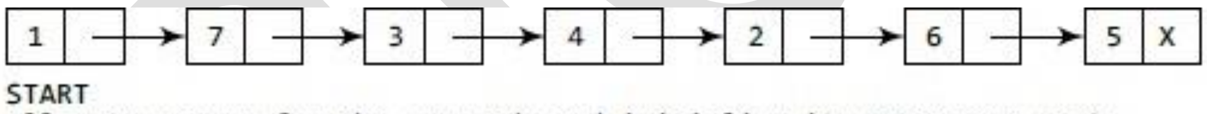
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET NEW_NODE->NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR->NEXT != NULL
Step 8:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 9: SET PTR->NEXT = NEW_NODE
Step 10: EXIT
```

Algorithm to insert a new node at the end

In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL, which signifies the end of the linked list.

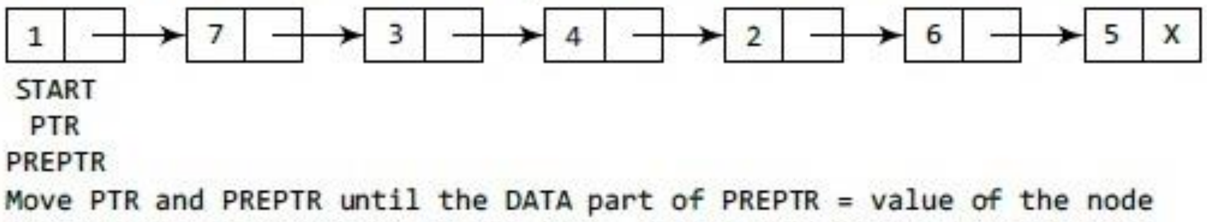
3. Inserting a Node After a Given Node in a Linked List

Consider the linked list shown in below. Suppose we want to add a new node with value 9 after the node containing data 3.

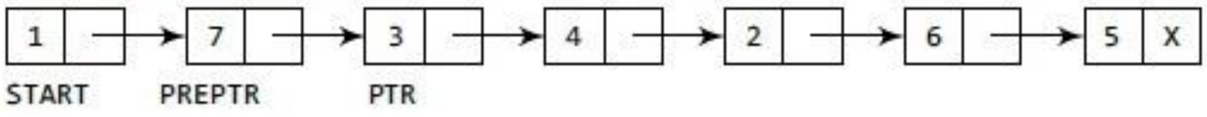


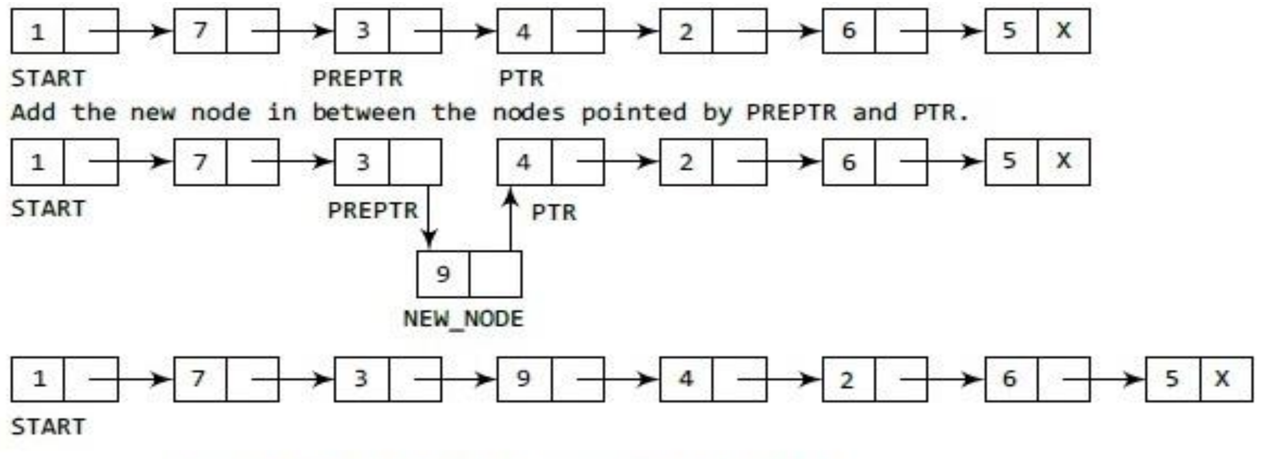
Allocate memory for the new node and initialize its DATA part to 9.

Take two pointer variables PTR and PREPTR and initialize them with START so that START, PTR, and PREPTR point to the first node of the list.



Move PTR and PREPTR until the DATA part of PREPTR = value of the node after which insertion has to be done. PREPTR will always point to the node just before PTR.




Add the new node in between the nodes pointed by PREPTR and PTR.

Inserting an element after a given node in a linked list

Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown below.

```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA
         != NUM
Step 8:       SET PREPTR = PTR
Step 9:       SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 10: PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

Algorithm to insert a new node after a node that has value NUM

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then we take another pointer variable PREPTR which will be used to store the address of the node preceding PTR. Initially, PREPTR is initialized to PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list.
In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that new node is inserted after the desired node.

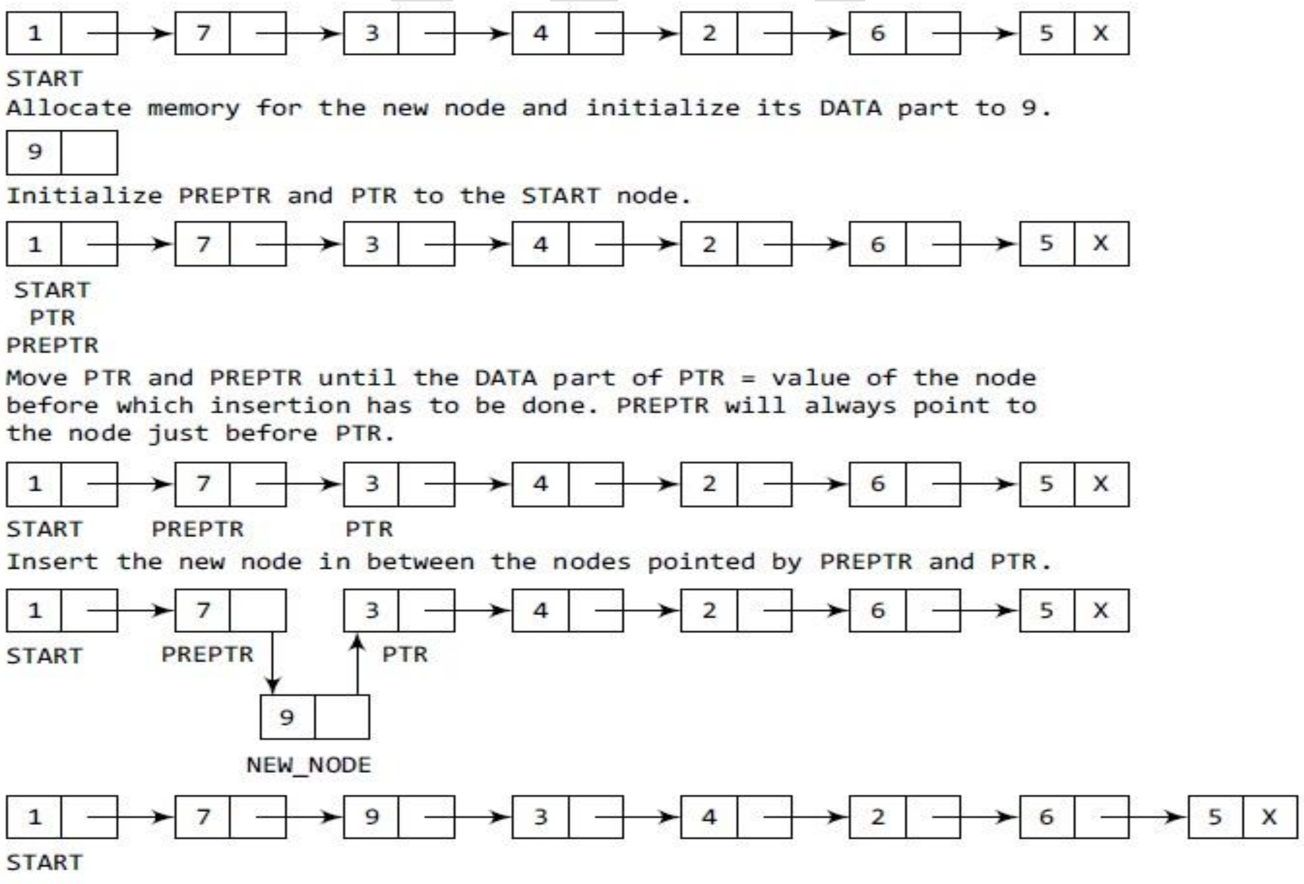
4. Inserting a Node Before a Given Node in a Linked List

Consider the linked list shown in below. Suppose we want to add a new node with value 9 before the node containing 3. Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown below.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 12
       [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL->NEXT
Step 4: SET NEW_NODE->DATA = VAL
Step 5: SET PTR = START
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PTR->DATA != NUM
Step 8:     SET PREPTR = PTR
Step 9:     SET PTR = PTR->NEXT
        [END OF LOOP]
Step 10: PREPTR->NEXT = NEW_NODE
Step 11: SET NEW_NODE->NEXT = PTR
Step 12: EXIT
```

Algorithm to insert a new node before a node that has value NUM

In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. Then, we take another pointer variable PREPTR and initialize it with PTR. So now, PTR, PREPTR, and START are all pointing to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, in Steps 10 and 11, we change the NEXT pointers in such a way that the new node is inserted before the desired node.



Allocate memory for the new node and initialize its DATA part to 9.

Initialize PREPTR and PTR to the START node.

Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.

Insert the new node in between the nodes pointed by PREPTR and PTR.

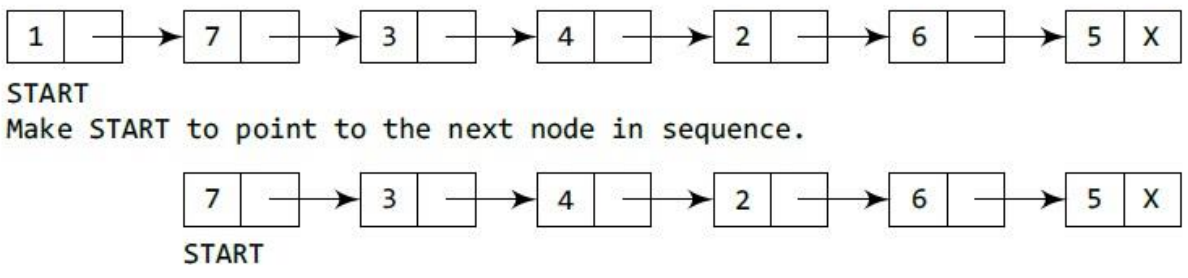
Inserting an element before a given node in a linked list

Deleting a Node from a Linked List

In this section, we will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.
Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.
Before we describe the algorithms in all these three cases, let us first discuss an important term called UNDERFLOW. Underflow is a condition that occurs when we try to delete a node from a linked list that is empty. This happens when START = NULL or when there are no more nodes to delete. Note that when we delete a node from a linked list, we actually have to free the memory occupied by that node. The memory is returned to the free pool so that it can be used to store other programs and data. Whatever be the case of deletion, we always change the AVAIL pointer so that it points to the address that has been recently vacated.

1. Deleting the First Node from a Linked List

Consider the linked list in the below Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.



Deleting the first node of a linked list

Below Figure shows the algorithm to delete the first node from a linked list.

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 5
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

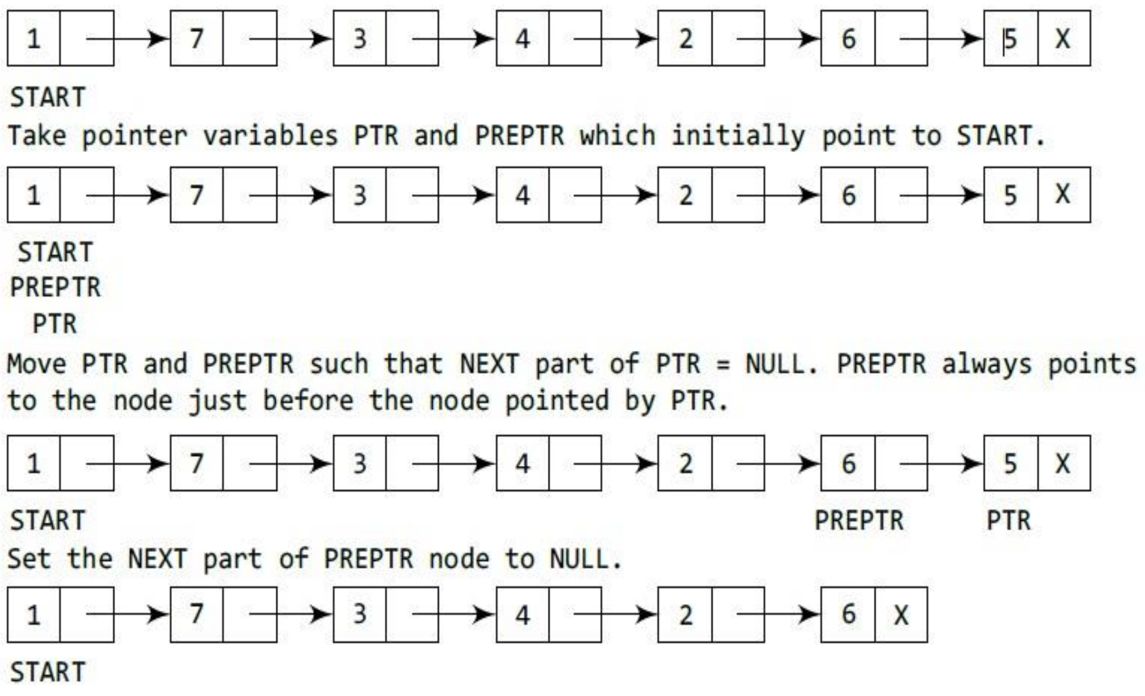
Algorithm to delete the first node

In Step 1, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by the node pointed by PTR (initially the first node of the list) is freed and returned to the free pool.

2. Deleting the Last Node from a Linked List

Consider the linked list shown in below. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

The below algorithm shows to delete the last node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the last node and the second last node, we set the NEXT pointer of the second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned back to the free pool.


Take pointer variables PTR and PREPTR which initially point to START.

Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.

Set the NEXT part of PREPTR node to NULL.

Deleting the last node of a linked list

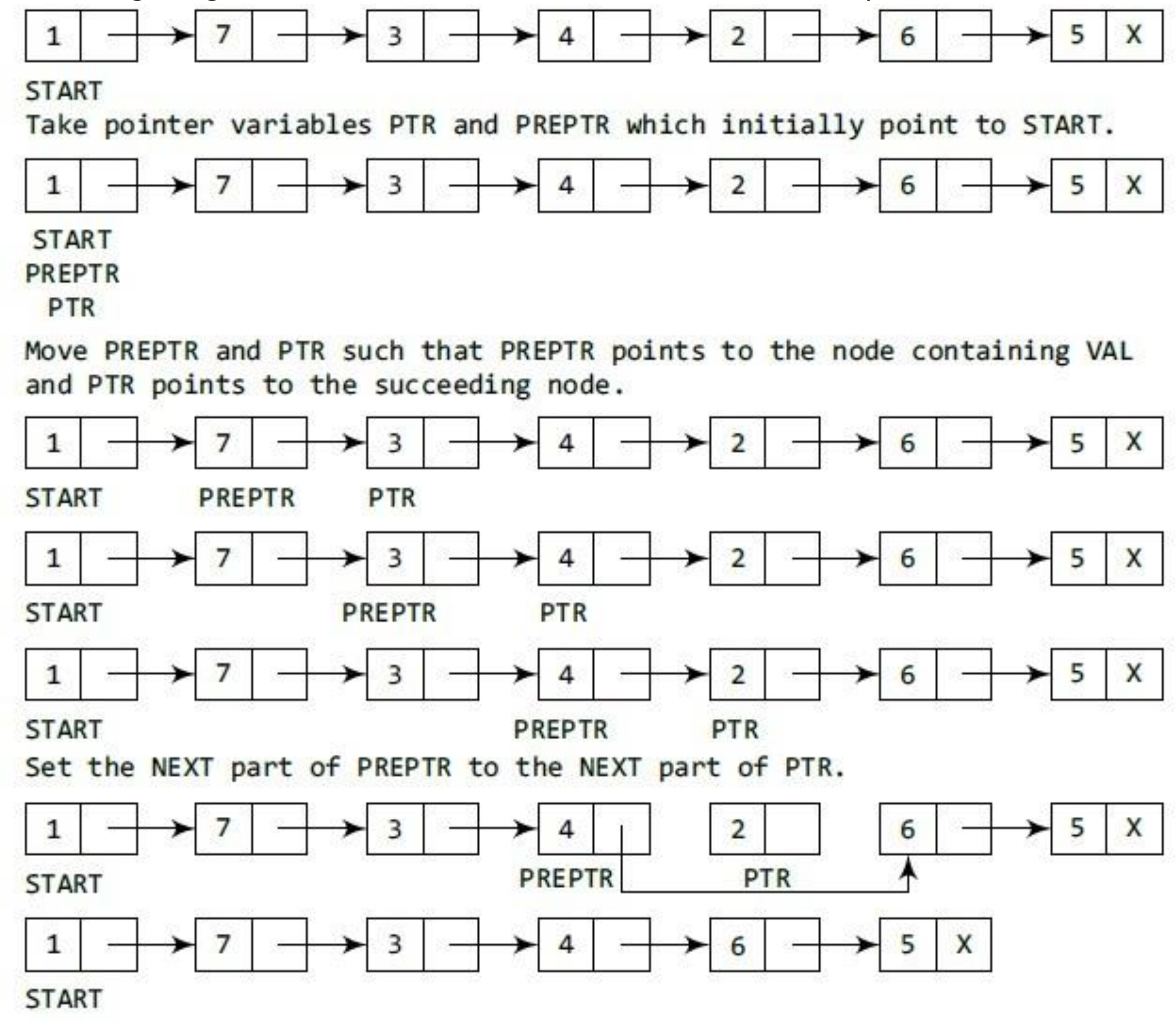
```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4:       SET PREPTR = PTR
Step 5:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Algorithm to delete the last node

3. Deleting the Node After a Given Node in a Linked List

Consider the linked list shown in Fig. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.

Figure 2 shows the algorithm to delete the node after a given node from a linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that it always points to one node before the PTR. Once we reach the node containing VAL and the node succeeding it, we set the next pointer of the node containing VAL to the address contained in next field of the node succeeding it. The memory of the node succeeding the given node is freed and returned back to the free pool.


1
7
3
4
2
6
5
X
START
Take pointer variables PTR and PREPTR which initially point to START.
START
PREPTR
PTR
Move PREPTR and PTR such that PREPTR points to the node containing VAL and PTR points to the succeeding node.
START
PREPTR
PTR
Set the NEXT part of PREPTR to the NEXT part of PTR.

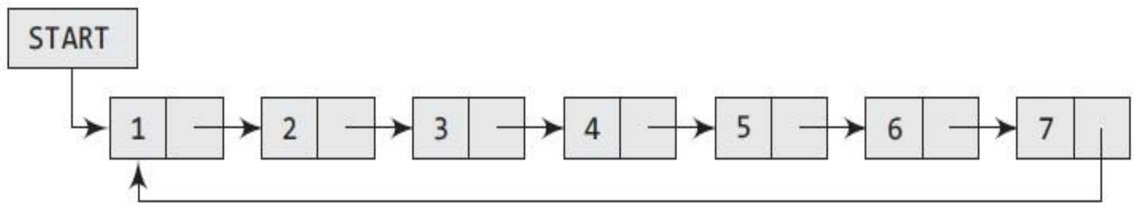
Deleting the node after a given node in a linked list

```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET PTR = START
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR->DATA != NUM
Step 5:       SET PREPTR = PTR
Step 6:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR->NEXT = PTR->NEXT
Step 9: FREE TEMP
Step 10: EXIT
```

Figure 2 Algorithm to delete the node after a given node

4. CIRCULAR LINKED LISTs

In a circular linked list, the last node contains a pointer to the first node of the list. We can have a circular singly linked list as well as a circular doubly linked list. While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending. Figure shows a circular linked list.



Circular linked list

Inserting a New Node in a Circular Linked List

In this section, we will see how a new node is added into an already existing linked list. We will take two cases and then see how insertion is done in each case.
Case 1: The new node is inserted at the beginning of the circular linked list.
Case 2: The new node is inserted at the end of the circular linked list.

1. Inserting a Node at the Beginning of a Circular Linked List

Consider the linked list shown in below. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list.

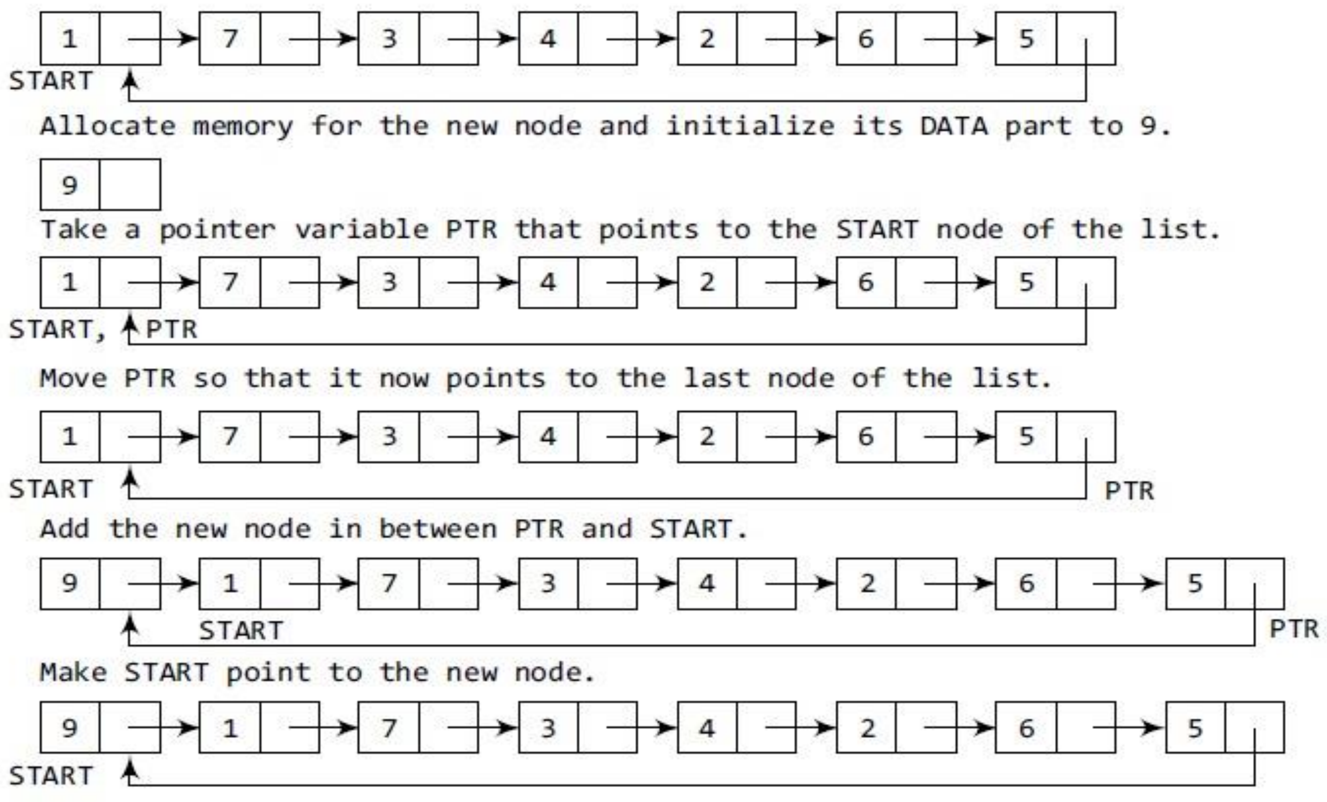

1 7 3 4 2 6 5
START
Allocate memory for the new node and initialize its DATA part to 9.
9
Take a pointer variable PTR that points to the START node of the list.
1 7 3 4 2 6 5
START, PTR
Move PTR so that it now points to the last node of the list.
1 7 3 4 2 6 5
START
PTR
Add the new node in between PTR and START.
9 1 7 3 4 2 6 5
START
PTR
Make START point to the new node.
9 1 7 3 4 2 6 5
START

Figure Inserting a new node at the beginning of a circular linked list

Figure shows the algorithm to insert a new node at the beginning of a linked list.

```
Step 1: IF AVAIL = NULL
             Write OVERFLOW
             Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:      PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

Figure Algorithm to insert a new node at the beginning

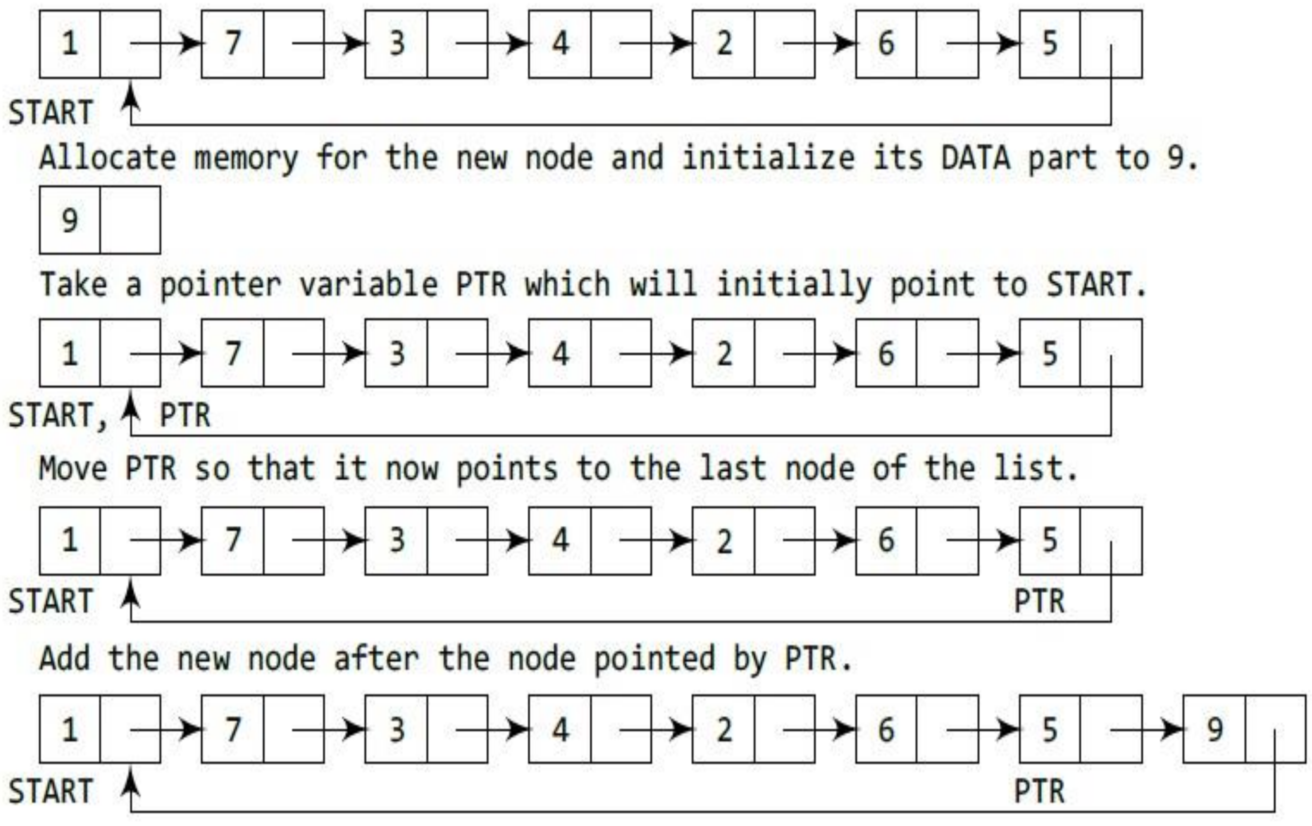
In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of the NEW_NODE. While inserting a node in a circular linked list, we have to use a while loop to traverse to the last node of the list. Because the last node contains a pointer to START, its NEXT field is updated so that after insertion it points to the new node which will be now known as START.

2. Inserting a Node at the End of a Circular Linked List

Consider the linked list shown in Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

Figure Algorithm to insert a new node at the end



Allocate memory for the new node and initialize its DATA part to 9.

Take a pointer variable PTR which will initially point to START.

Move PTR so that it now points to the last node of the list.

Add the new node after the node pointed by PTR.

Figure Inserting a new node at the end of a circular linked list

Figure shows the algorithm to insert a new node at the end of a circular linked list.

In Step 6, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains the address of the first node which is denoted by START.

Deleting a Node from a Circular Linked List

In this section, we will discuss how a node is deleted from an already existing circular linked list. We will take two cases and then see how deletion is done in each case. Rest of the cases of deletion is same as that given for singly linked lists.

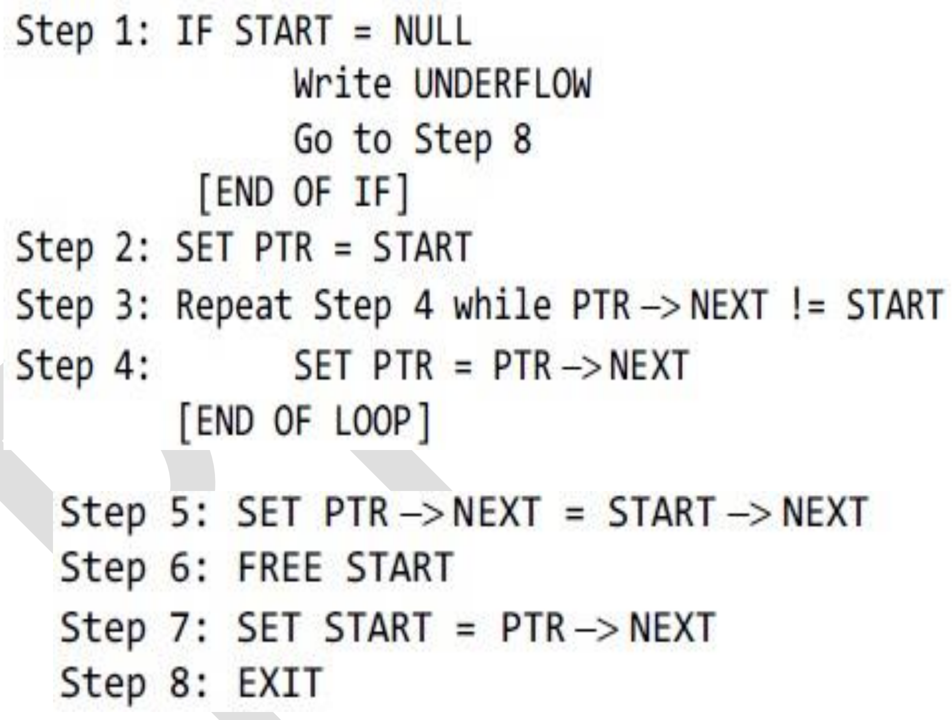
Case 1: The first node is deleted.

Case 2: The last node is deleted.

1. Deleting the First Node from a Circular Linked List

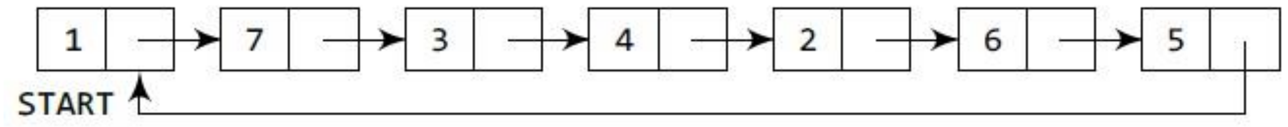
Consider the circular linked list shown in Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

The following algorithm shows how to delete the first node from a circular linked list.

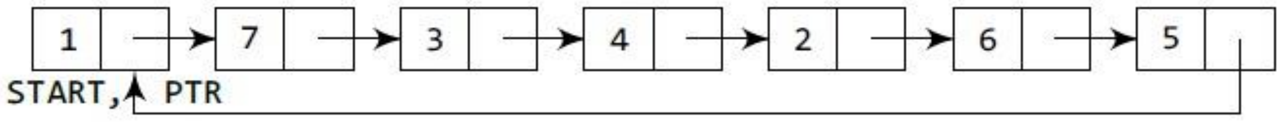


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```

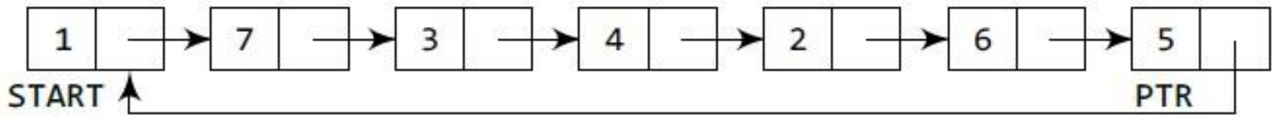
In Step 1 of the algorithm, we check if the linked list exists or not. If START = NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a pointer variable PTR which will be used to traverse the list to ultimately reach the last node. In Step 5, we change the next pointer of the last node to point to the second node of the circular linked list. In Step 6, the memory occupied by the first node is freed. Finally, in Step 7, the second node now becomes the first node of the list and its address is stored in the pointer variable START.



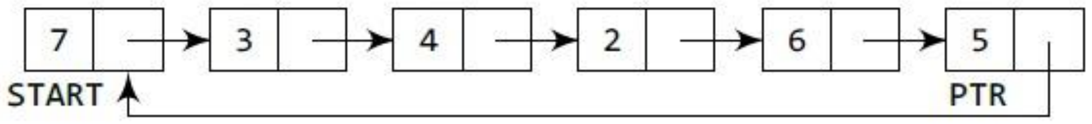
Take a variable PTR and make it point to the START node of the list.



Move PTR further so that it now points to the last node of the list.



The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.



Deleting the first node from a circular linked list

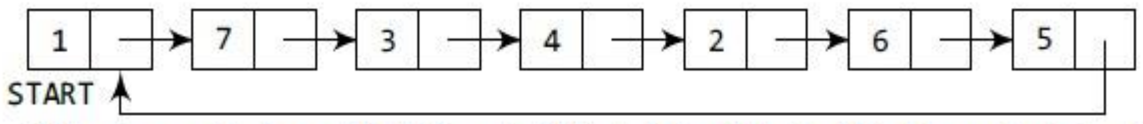
2. Deleting the Last Node from a Circular Linked List

Consider the circular linked list shown in Fig. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.

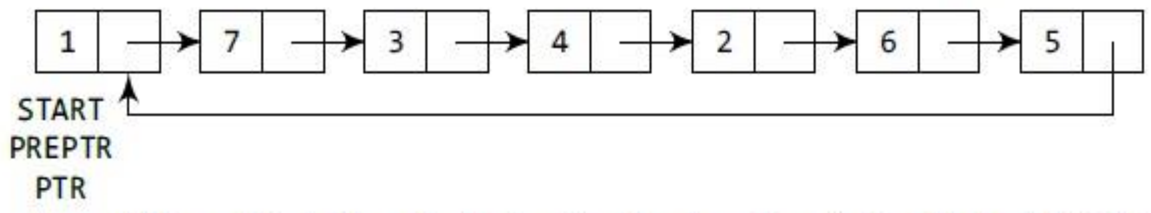
Figure shows the algorithm to delete the last node from a circular linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we take another pointer variable PREPTR such that PREPTR always points to one node before PTR. Once we reach the last node and the second last node, we set the next pointer of the second last node to START, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

```
Step 1: IF START = NULL
                Write UNDERFLOW
                Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:         SET PREPTR = PTR
Step 5:         SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```

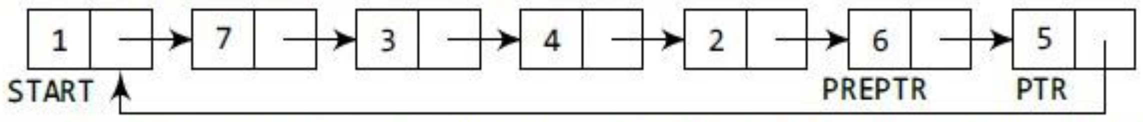
Algorithm to delete the last node



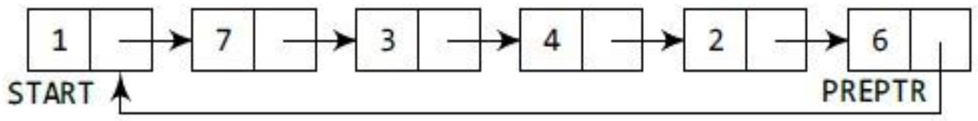
Take two pointers PREPTR and PTR which will initially point to START.



Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.



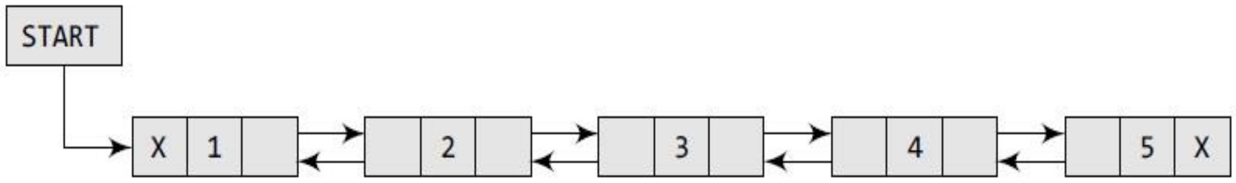
Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.



Deleting the last node from a circular linked list

5. DOUBLY LINKED LISTS

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Fig.



Doubly linked list

In C, the structure of a doubly linked list can be given as,

```
struct node
{
struct node *prev;
int data;
struct node *next;
};
```

The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction. Thus, we see that a doubly linked list calls for more space per node and more expensive basic operations. However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient.

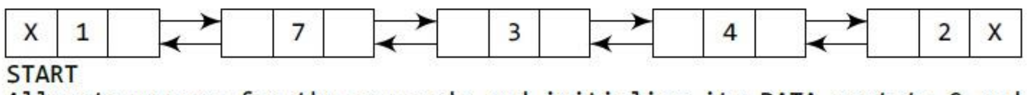
Inserting a New Node in a Doubly Linked List

In this section, we will discuss how a new node is added into an already existing doubly linked list. We will take four cases and then see how insertion is done in each case.

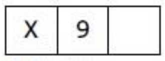
Case 1: The new node is inserted at the beginning.
Case 2: The new node is inserted at the end.
Case 3: The new node is inserted after a given node.
Case 4: The new node is inserted before a given node.

1. Inserting a Node at the Beginning of a Doubly Linked List

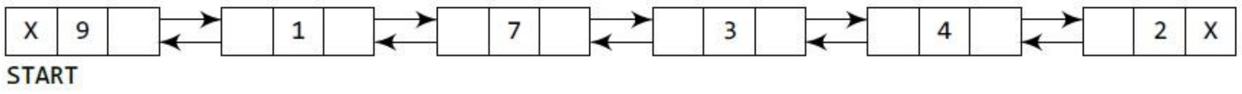
Consider the doubly linked list shown in Fig. Suppose we want to add a new node with data 9 as the first node of the list. Then the following changes will be done in the linked list



Allocate memory for the new node and initialize its DATA part to 9 and PREV field to NULL.

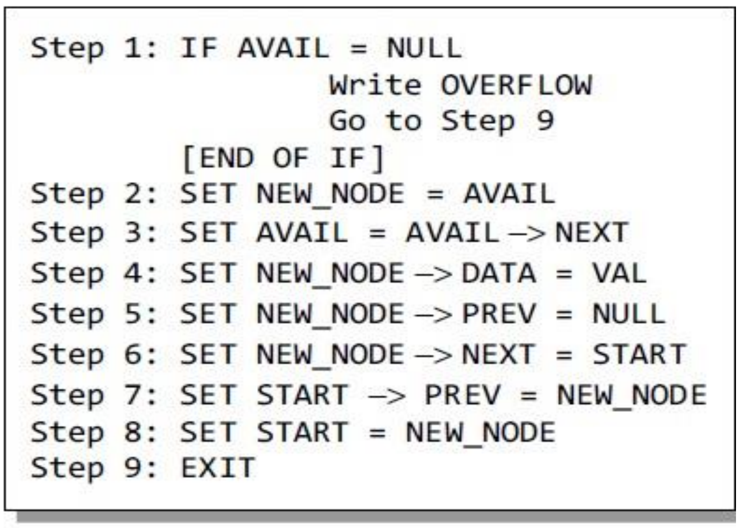


Add the new node before the START node. Now the new node becomes the first node of the list.



Inserting a new node at the beginning of a doubly linked list

Figure shows the algorithm to insert a new node at the beginning of a doubly linked list.



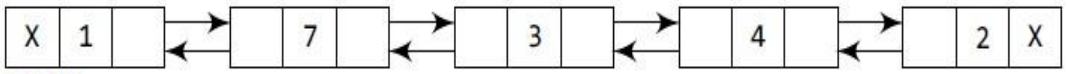
```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 9
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> PREV = NULL
Step 6: SET NEW_NODE -> NEXT = START
Step 7: SET START -> PREV = NEW_NODE
Step 8: SET START = NEW_NODE
Step 9: EXIT
```

Algorithm to insert a new node at the beginning

In Step 1, we first check whether memory is available for the new node. If the free memory has exhausted, then an OVERFLOW message is printed. Otherwise, if free memory cell is available, then we allocate space for the new node. Set its DATA part with the given VAL and the NEXT part is initialized with the address of the first node of the list, which is stored in START. Now, since the new node is added as the first node of the list, it will now be known as the START node, that is, the START pointer variable will now hold the address of NEW_NODE.

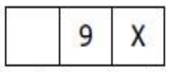
2. Inserting a Node at the End of a Doubly Linked List

Consider the doubly linked list shown in Fig. Suppose we want to add a new node with data 9 as the last node of the list. Then the following changes will be done in the linked list.

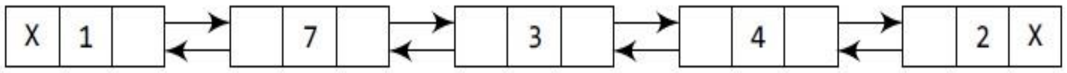


START

Allocate memory for the new node and initialize its DATA part to 9 and its NEXT field to NULL.

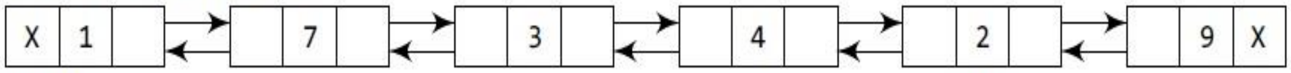


Take a pointer variable PTR and make it point to the first node of the list.



START,PTR

Move PTR so that it points to the last node of the list. Add the new node after the node pointed by PTR.



START PTR

Inserting a new node at the end of a doubly linked list

Figure shows the algorithm to insert a new node at the end of a doubly linked list.

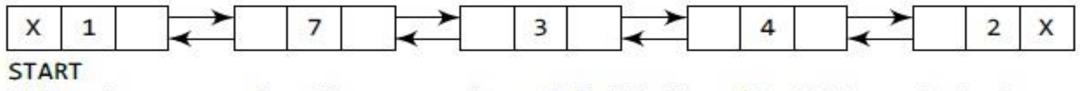
```
Step 1: IF AVAIL = NULL
             Write OVERFLOW
             Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET NEW_NODE -> PREV = PTR
Step 11: EXIT
```

Algorithm to insert a new node at the end

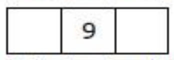
In Step 6, we take a pointer variable PTR and initialize it with START. In the while loop, we traverse through the linked list to reach the last node. Once we reach the last node, in Step 9, we change the NEXT pointer of the last node to store the address of the new node. Remember that the NEXT field of the new node contains NULL which signifies the end of the linked list. The PREV field of the NEW_NODE will be set so that it points to the node pointed by PTR (now the second last node of the list).

3. Inserting a Node After a Given Node in a Doubly Linked List

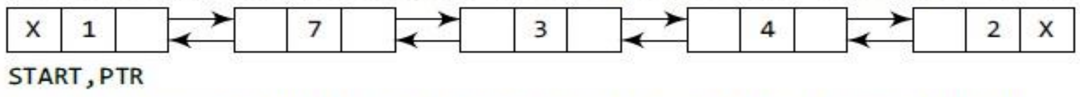
Consider the doubly linked list shown in Fig. Suppose we want to add a new node with value 9 after the node containing 3.



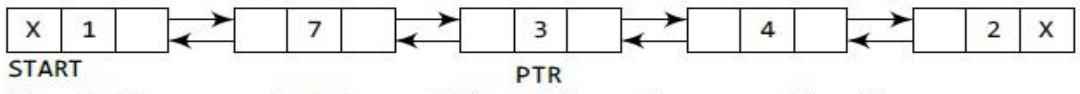
Allocate memory for the new node and initialize its DATA part to 9.



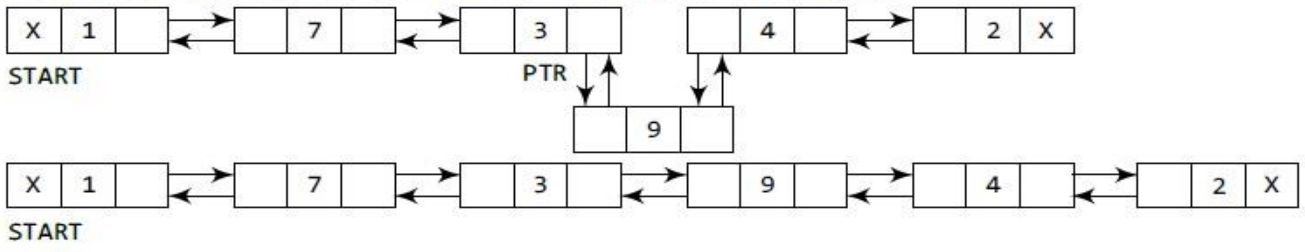
Take a pointer variable PTR and make it point to the first node of the list.



Move PTR further until the data part of PTR = value after which the node has to be inserted.



Insert the new node between PTR and the node succeeding it.



Inserting a new node after a given node in a doubly linked list

Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig.

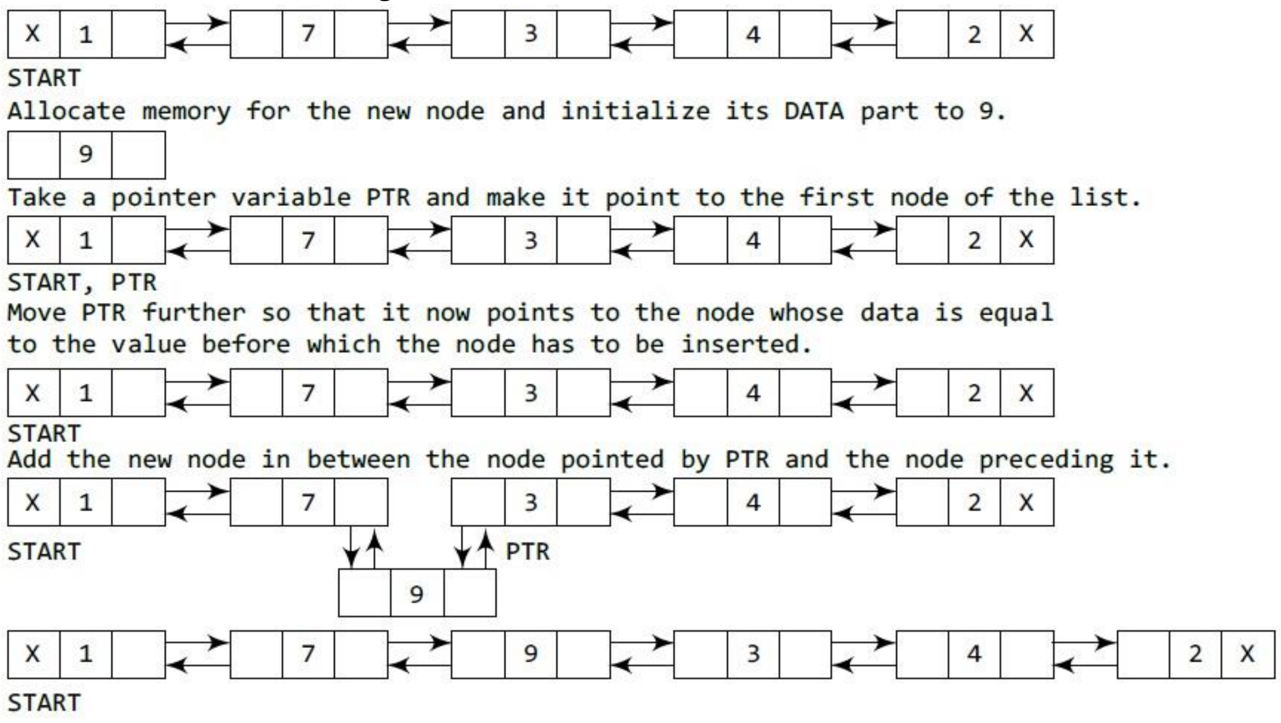
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR -> NEXT
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET PTR -> NEXT = NEW_NODE
Step 11: SET PTR -> NEXT -> PREV = NEW_NODE
Step 12: EXIT
```

Algorithm to insert a new node after a given node

The above algorithm shows how to insert a new node after a given node in a doubly linked list. In Step 5, we take a pointer PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted after this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted after the desired node.

4. Inserting a Node Before a Given Node in a Doubly Linked List

Consider the doubly linked list shown in Fig. Suppose we want to add a new node with value 9 before the node containing 3.



Allocate memory for the new node and initialize its DATA part to 9.

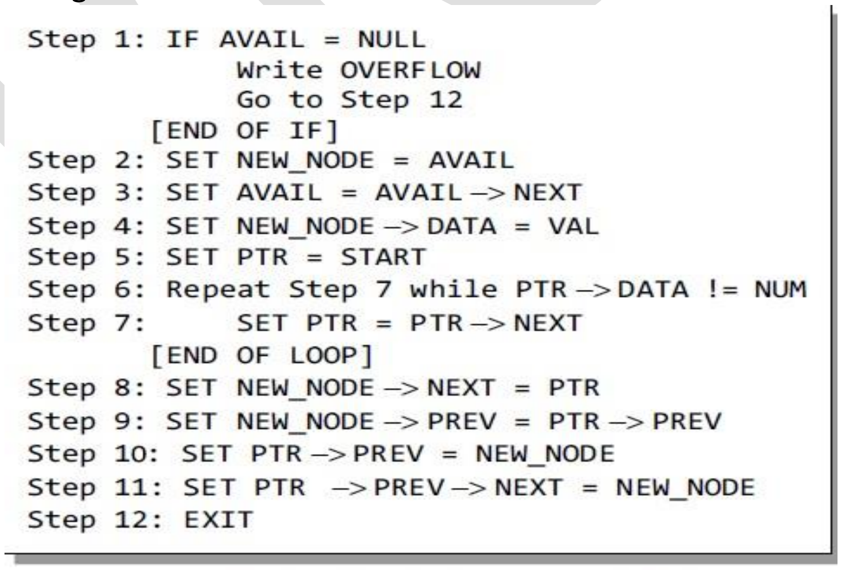
Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that it now points to the node whose data is equal to the value before which the node has to be inserted.

Add the new node in between the node pointed by PTR and the node preceding it.

Inserting a new node before a given node in a doubly linked list

Before discussing the changes that will be done in the linked list, let us first look at the algorithm shown in Fig.



```
Step 1: IF AVAIL = NULL
             Write OVERFLOW
             Go to Step 12
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> DATA != NUM
Step 7:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = PTR
Step 9: SET NEW_NODE -> PREV = PTR -> PREV
Step 10: SET PTR -> PREV = NEW_NODE
Step 11: SET PTR  -> PREV -> NEXT = NEW_NODE
Step 12: EXIT
```

Algorithm to insert a new node before a given node

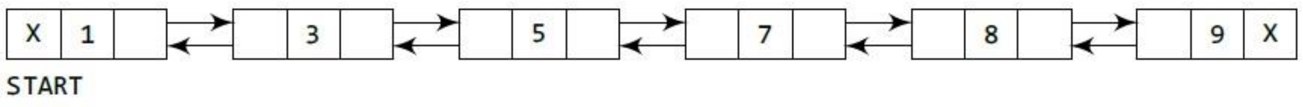
In Step 1, we first check whether memory is available for the new node. In Step 5, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. In the while loop, we traverse through the linked list to reach the node that has its value equal to NUM. We need to reach this node because the new node will be inserted before this node. Once we reach this node, we change the NEXT and PREV fields in such a way that the new node is inserted before the desired node.

Deleting a Node from a Doubly Linked List

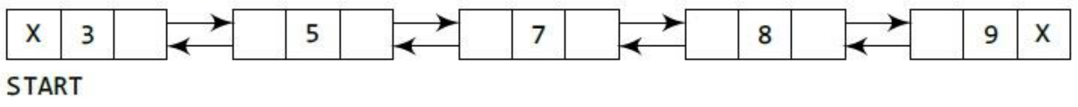
In this section, we will see how a node is deleted from an already existing doubly linked list. We will take four cases and then see how deletion is done in each case.
Case 1: The first node is deleted.
Case 2: The last node is deleted.
Case 3: The node after a given node is deleted.
Case 4: The node before a given node is deleted.

1. Deleting the First Node from a Doubly Linked List

Consider the doubly linked list shown in Fig. When we want to delete a node from the beginning of the list, then the following changes will be done in the linked list.

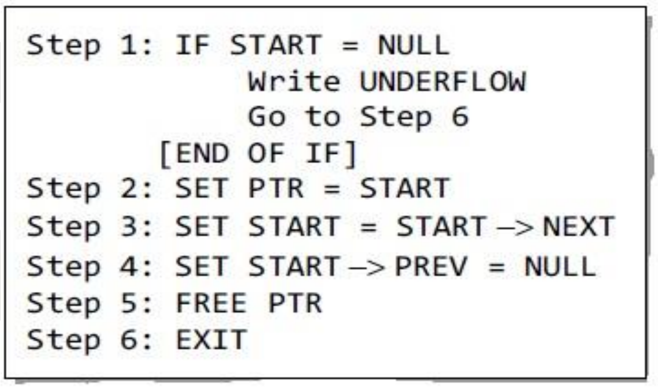


Free the memory occupied by the first node of the list and make the second node of the list as the START node.



Deleting the first node from a doubly linked list

Below figure shows the algorithm to delete the first node of a doubly linked list.



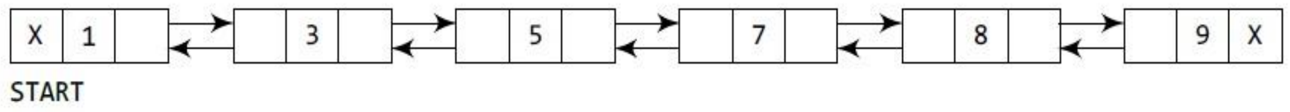
```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 6
          [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START -> NEXT
Step 4: SET START -> PREV = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Algorithm to delete the first node

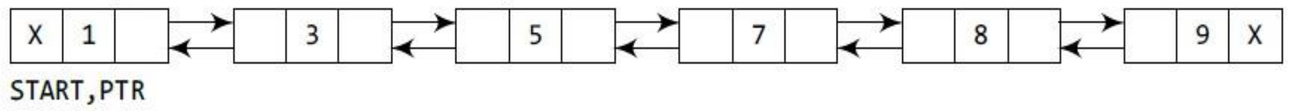
In Step 1 of the algorithm, we check if the linked list exists or not. If START =NULL, then it signifies that there are no nodes in the list and the control is transferred to the last statement of the algorithm. However, if there are nodes in the linked list, then we use a temporary pointer variable PTR that is set to point to the first node of the list. For this, we initialize PTR with START that stores the address of the first node of the list. In Step 3, START is made to point to the next node in sequence and finally the memory occupied by PTR (initially the first node of the list) is freed and returned to the free pool.

2. Deleting the Last Node from a Doubly Linked List

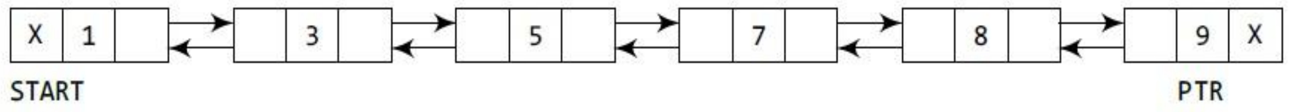
Consider the doubly linked list shown in Fig. Suppose we want to delete the last node from the linked list, then the following changes will be done in the linked list.



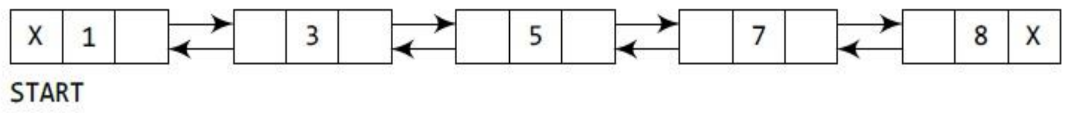
Take a pointer variable PTR that points to the first node of the list.



Move PTR so that it now points to the last node of the list.

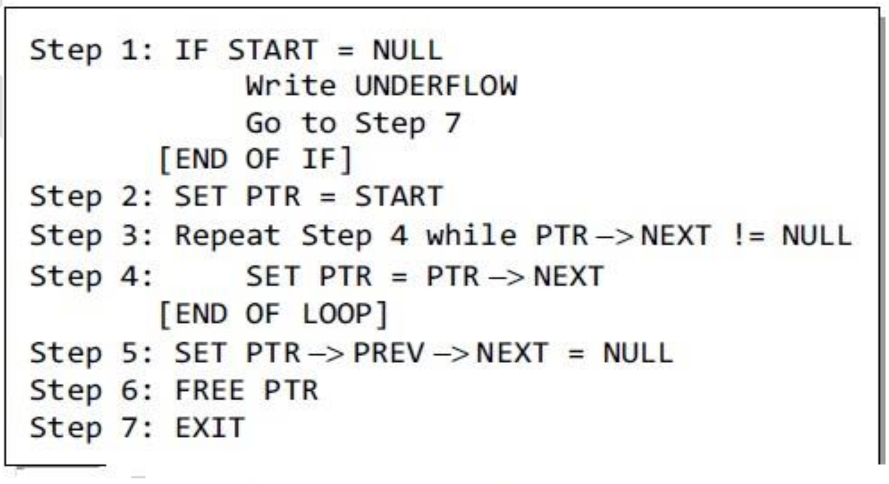


Free the space occupied by the node pointed by PTR and store NULL in NEXT field of its preceding node.



Deleting the last node from a doubly linked list

Figure shows the algorithm to delete the last node of a doubly linked list. In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the list to reach the last node. Once we reach the last node, we can also access the second last node by taking its address from the PREV field of the last node. To delete the last node, we simply have to set the next field of second last node to NULL, so that it now becomes the (new) last node of the linked list. The memory of the previous last node is freed and returned to the free pool.

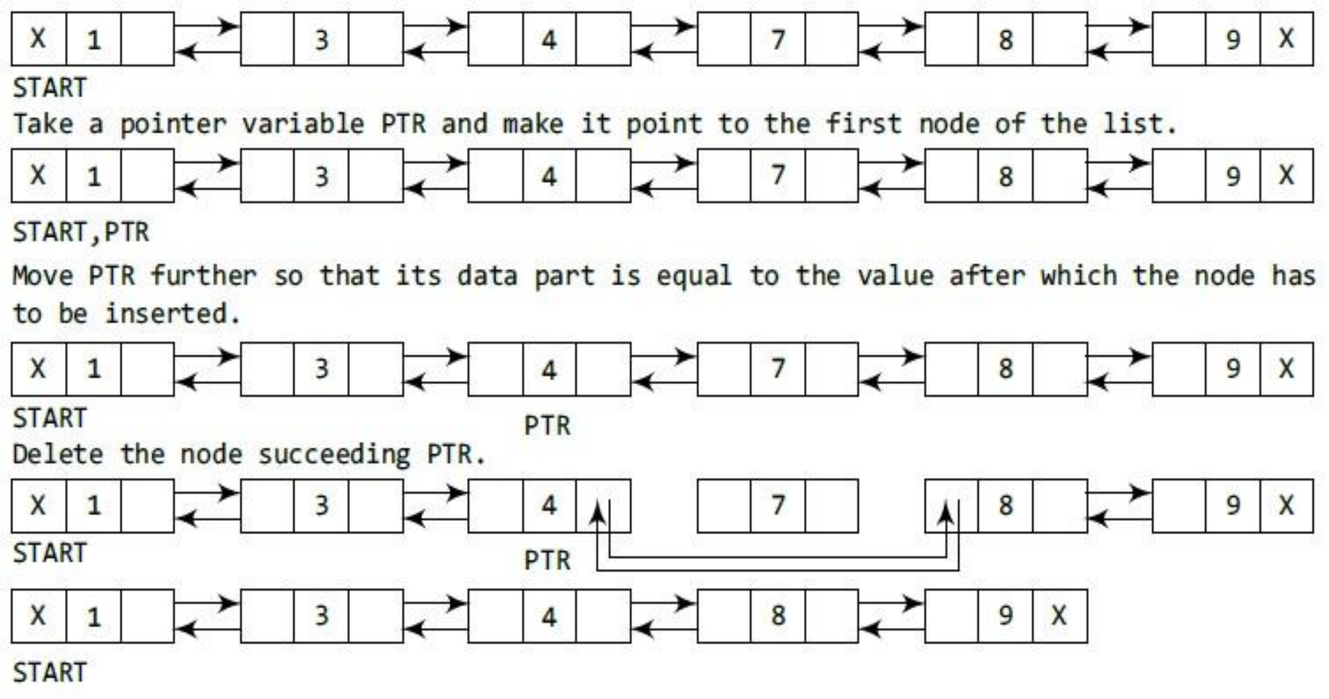


```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 7
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->NEXT != NULL
Step 4:       SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: SET PTR->PREV->NEXT = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Algorithm to delete the last node

3. Deleting the Node After a Given Node in a Doubly Linked List

Consider the doubly linked list shown in Fig. Suppose we want to delete the node that succeeds the node which contains data value 4. Then the following changes will be done in the linked list.



Take a pointer variable PTR and make it point to the first node of the list.

Move PTR further so that its data part is equal to the value after which the node has to be inserted.

Delete the node succeeding PTR.

Deleting the node after a given node in a doubly linked list

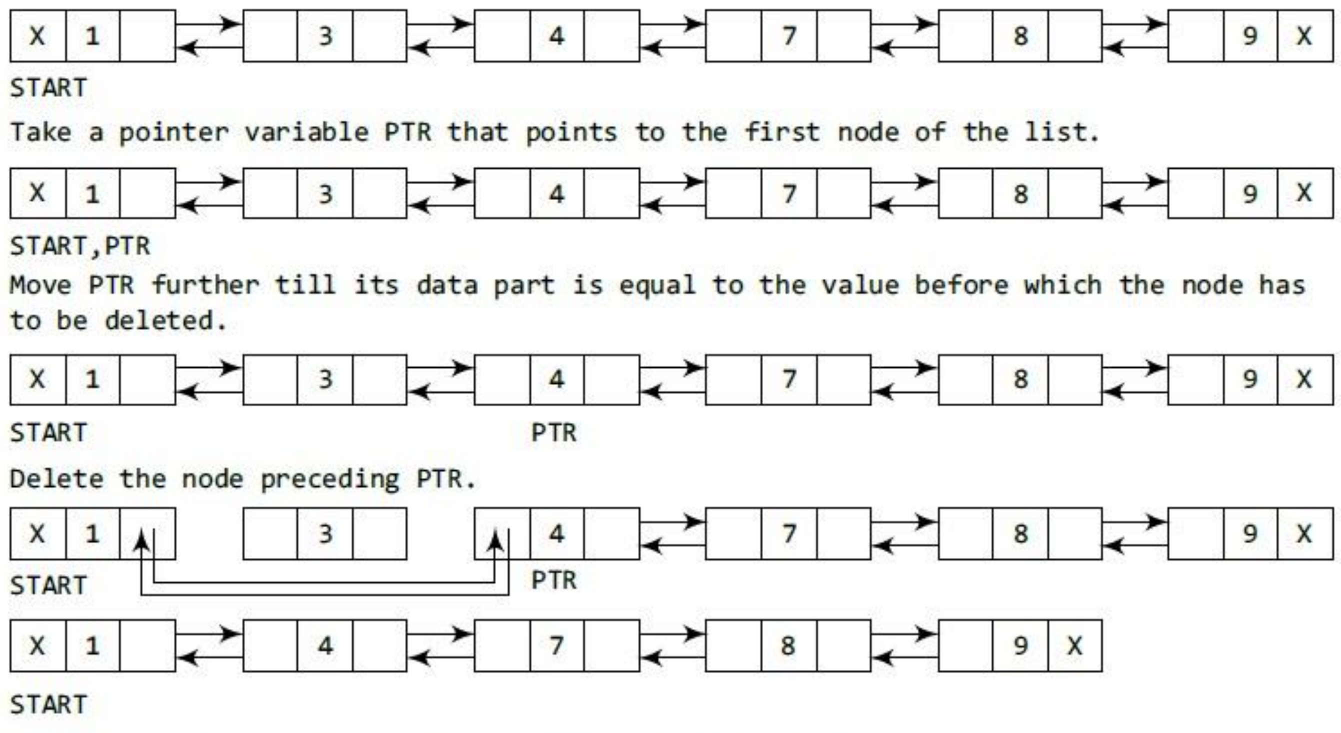
Figure shows the algorithm to delete a node after a given node of a doubly linked list.

```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> NEXT
Step 6: SET PTR -> NEXT = TEMP -> NEXT
Step 7: SET TEMP -> NEXT -> PREV = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

In Step 2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the doubly linked list. The while loop traverses through the linked list to reach the given node. Once we reach the node containing VAL, the node succeeding it can be easily accessed by using the address stored in its NEXT field. The NEXT field of the given node is set to contain the contents in the NEXT field of the succeeding node. Finally, the memory of the node succeeding the given node is freed and returned to the free pool.

4. Deleting the Node Before a Given Node in a Doubly Linked List

Consider the doubly linked list shown in Fig. Suppose we want to delete the node preceding the node with value 4.



Take a pointer variable PTR that points to the first node of the list.

Move PTR further till its data part is equal to the value before which the node has to be deleted.

Delete the node preceding PTR.

Deleting a node before a given node in a doubly linked list

Before discussing the changes that will be done in the linked list, let us first look at the algorithm.

```
Step 1: IF START = NULL
             Write UNDERFLOW
             Go to Step 9
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> DATA != NUM
Step 4:      SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET TEMP = PTR -> PREV
Step 6: SET TEMP -> PREV -> NEXT = PTR
Step 7: SET PTR -> PREV = TEMP -> PREV
Step 8: FREE TEMP
Step 9: EXIT
```

Figure 6.54 Algorithm to delete a node before a given node

In Step2, we take a pointer variable PTR and initialize it with START. That is, PTR now points to the first node of the linked list. The while loop traverses through the linked list to reach the desired node. Once we reach the node containing VAL, the PREV field of PTR is set to contain the address of the node preceding the node which comes before PTR. The memory of the node preceding PTR is freed and returned to the free pool. Hence, we see that we can insert or delete a node in a constant number of operations given only that node's address. Note that this is not possible in the case of a singly linked list which requires the previous node's address also to perform the same operation.

Difference between Array List and Linked List

ArrayList and LinkedList both implement the List interface and maintain insertion order. Both are non-synchronized classes.
However, there are many differences between the ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.
7) To be precise, an ArrayList is a resizable array.	LinkedList implements the doubly linked list of the list interface.

********************************END OF UNIT-1********************************