UNIT-1

What is Data Structure?

- Data structure is an arrangement of data in computer's memory. It makes the data quickly available to the processor for required operations.
- It is a software artifact which allows data to be stored, organized and accessed.
- It is a structure program used to store ordered data, so that various operations can be performed on it easily.

Types of Data Structure

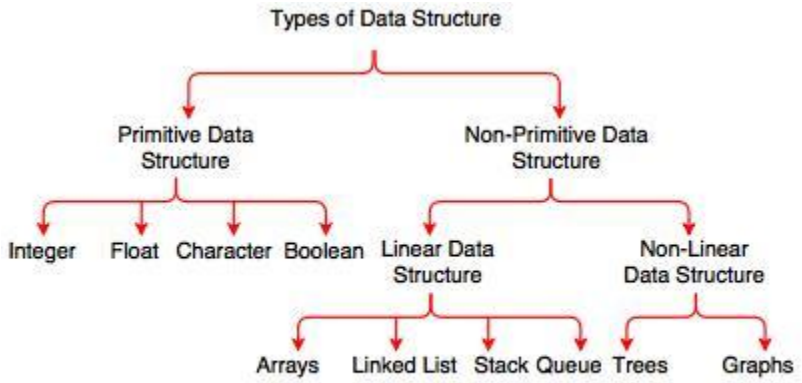


Fig. Types of Data Structure

A. Primitive Data Type

- Primitive data types are the data types available in most of the programming languages.
- These data types are used to represent single value.
- It is a basic data type available in most of the programming language.

Data type Description

Integer	Used to represent a number without decimal point.
Float	Used to represent a number with decimal point.
Character	Used to represent single character.
Boolean	Used to represent logical values either true or false.

B. Non-Primitive Data Type

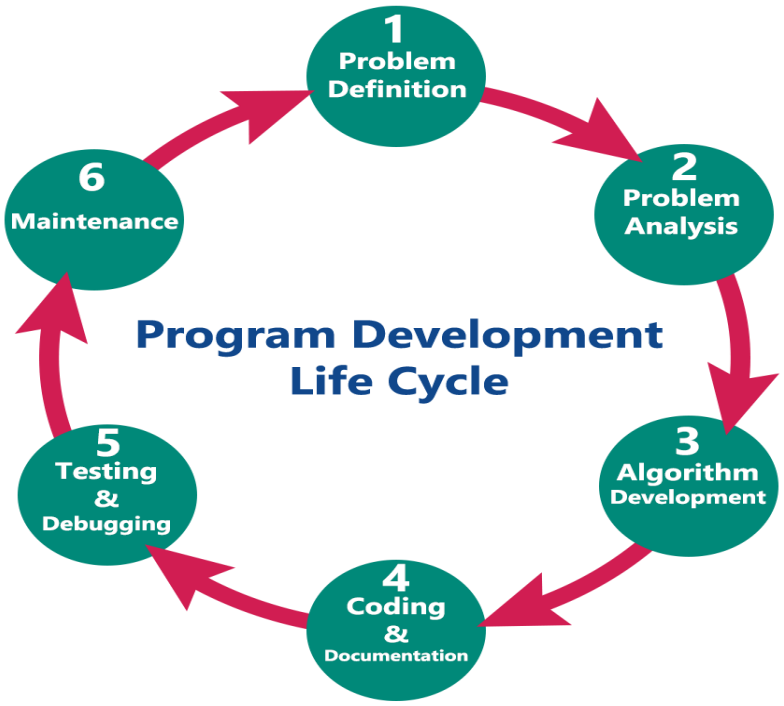
- Data type derived from primary data types are known as Non-Primitive data types.
- Non-Primitive data types are used to store group of values.

Program Development Life Cycle

When we want to develop a program using any programming language, we follow a sequence of steps. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language.

Generally, program development life cycle contains 6 phases, they are as follows….

- Problem Definition
- Problem Analysis
- Algorithm Development
- Coding & Documentation
- Testing & Debugging
- Maintenance



1. Problem Definition

In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirment, what should be the output of the problem solution. These are defined in this first phase of the program development life cycle.

2. Problem Analysis

In phase 2, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

3. Algorithm Development

During this phase, we develop a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means we write the solution in step by step statements.

4. Coding & Documentation

This phase uses a programming language to write or implement actual programming instructions for the steps defined in the previous phase. In this phase, we construct actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java etc.,

5. Testing & Debugging

During this phase, we check whether the code written in previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test that whether it is providing the desired output or not.

6. Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated again to make the enhancements. That means in this phase, the solution (program) is used by the end user. If the user encounters any problem or wants any enhancement,then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

Difference between Top Down and Bottom up approach of programming

Basis for comparison	Top-down Approach	Bottom-up Approach
Basic	Breaks the massive problem into smaller subproblems.	Solves the fundamental low-level problem and integrates them into a larger one.
Process	Submodules are solitarily analysed.	Examine what data is to be encapsulated, and implies the concept of information hiding.
Communication	Not required in the top-down approach.	Needs a specific amount of communication.
Redundancy	Contain redundant information.	Redundancy can be eliminated.
Programming languages	Structure/procedural oriented programming languages (i.e. C) follows the top-down approach.	Object-oriented programming languages (like C++, Java, etc.) follows the bottom-up approach.
Mainly used in	Module documentation, test case creation, code implementation and debugging.	Testing

Difference between Procedure Oriented and Object Oriented Language

Basis for comparison	Procedure Oriented Language	Object Oriented Language
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP,Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the	In OOP, data can not move easily from function to function,it can be kept public or private so we can

Basis for comparison	Top-down Approach	Bottom-up Approach
	system.	control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Structured Programming Language

Definition – It is a programming method which aimed at improving quality, clarity and access time of computer program by the use of block structures, subroutines, for and while loops. This programming features will be helpful when concept of exception handing is needed in the program. It uses various control structures, sub routines, blocks and theorem. The theorems involved in structure programming are Sequence, Selection, Iteration and Recursion. Most of the programming language uses structured programming language features such as ALGOL, Pascal, PL/I, Ada, C, etc. The structure programming enforces a logical structure on the program being written to make it more efficient and easy to modify and understand. What is Structured Programming Language is explained in simple and precise manner.

Main features of Structural Programming language

1. Division of Complex problems into small procedures and functions.
2. No presence of GOTO Statement
3. The main statement include – If-then-else, Call and Case statements.
4. Large set of operators like arithmetic, relational, logical, bit manipulation, shift and part word operators.
5. Inclusion of facilities for implementing entry points and external references in program.

Datatypes

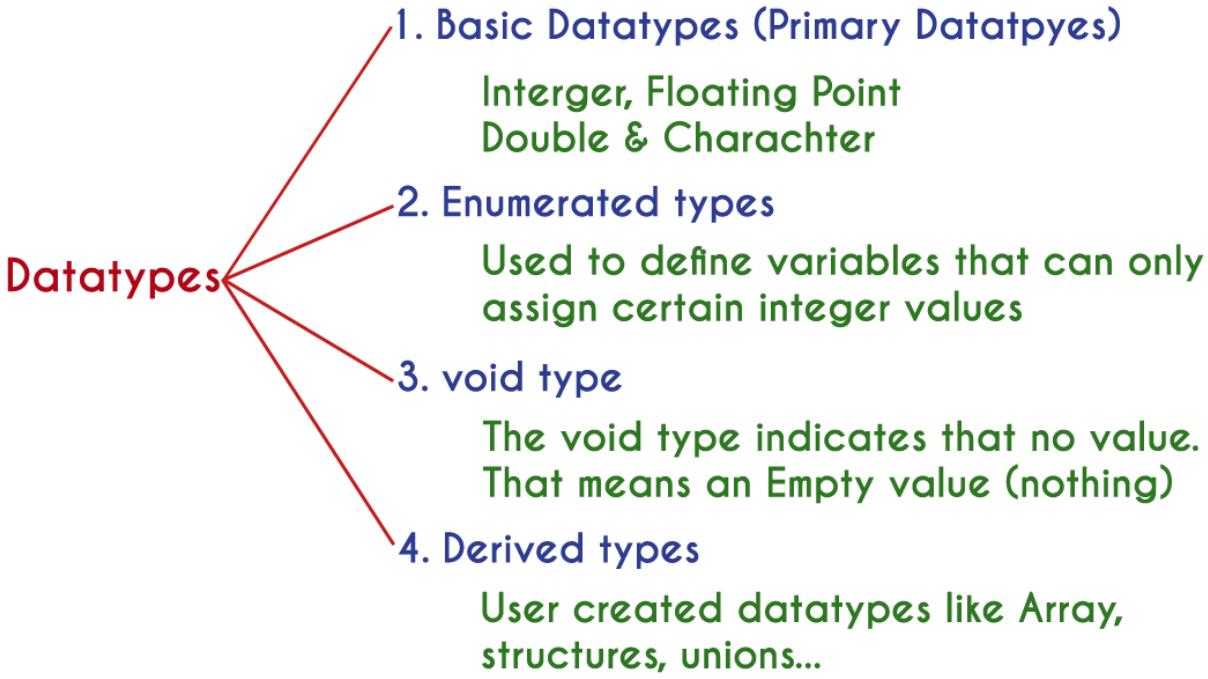
Data used in c program is classified into different types based on its properties. In c

programming langauge, datatype can be defined as a set of values with similar characteristics.

All the values in a datatype have the same properties.

Datatypes in c programming language are used to specify what kind of value can be stored in a variable. The memory size and type of value of a variable are determined by varible datatype. In a c program, each variable or constant or array must have a datatype and this datatype specifies how much memory is to be allocated and what type of values are to be stored in that variable or constant or array. In c programming language, datatypes are classified as follows...

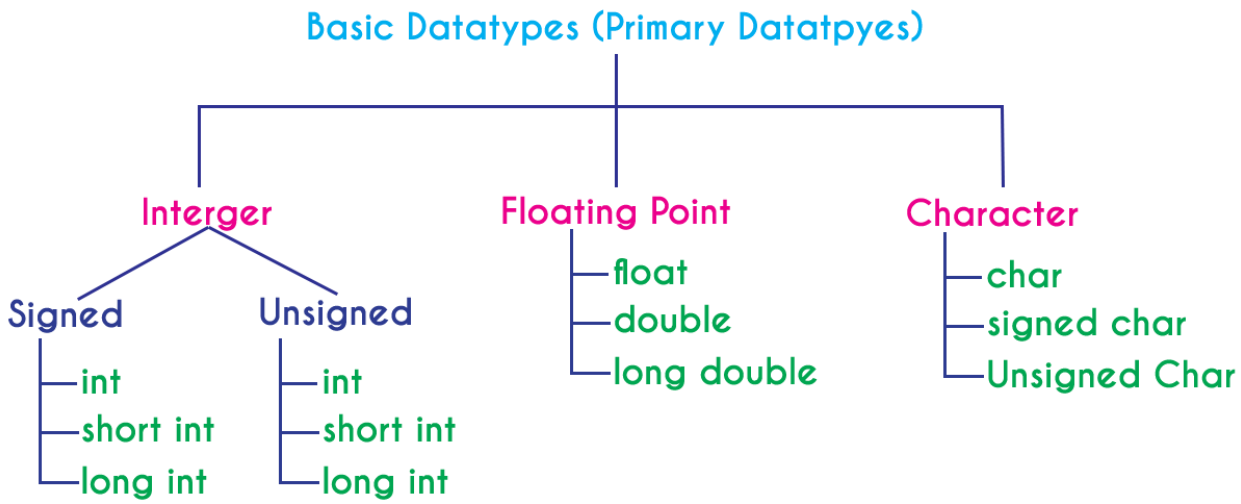
1. Primary Datatypes (Basic Datatypes OR Predefined Datatypes)
2. Derived Datatypes (Secondary Datatypes OR Userdefined Datatypes)
3. Enumeration Datatypes
4. Void Datatype



Primary Datatypes

The primary datatypes in C programming language are the basic datatypes. All the primary datatypes are already defined in the system. Primary datatypes are also called as Built-In datatypes. The following are the primary datatypes in c programming lanuage...

1. Integer Datatype
2. Floating Point Datatype
3. Double Datatype
4. Character Datatype



Integer Datatype

Integer datatype is a set of whole numbers. Every integer value does not have the decimal value. We use the keyword "**int**" to represent integer datatype in c.

Type	Size (Bytes)	Range	Specifier
int (signed short int)	2	-32768 to +32767	%d
short int (signed short int)	2	-32768 to +32767	%d
long int (signed long int)	4	-2,147,483,648 to +2,147,483,647	%d
unsigned int (unsigned short int)	2	0 to 65535	%u
unsigned long int	4	0 to 4,294,967,295	%u

Floating Point Datatypes

Floating point datatypes are set of numbers with decimal value. Every floating point value must contain the decimal value. The floating point datatype has two variants...

- float
- double

Type	Size (Bytes)	Range	Specifier
float	4	1.2E - 38 to 3.4E + 38	%f
double	8	2.3E-308 to 1.7E+308	%ld
long double	10	3.4E-4932 to 1.1E+4932	%ld

Character Datatype

Character datatype is a set of characters enclosed in single quotations. The following table provides complete details about character datatype

Type	Size (Bytes)	Range	Specifier
char (signed char)	1	-128 to +127	%c
unsigned char	1	0 to 255	%c

void Datatype

The void datatype means nothing or no value. Generally, void is used to specify a function which does not return any value. We also use the void datatype to specify empty parameters of a function.

Enumerated Datatype

An enumerated datatype is a user-defined data type that consists of integer constants and each integer constant is given a name. The keyword "**enum**" is used to define enumerated datatype.

Derived Datatypes

Derived datatypes are user-defined data types. The derived datatypes are also called as user defined datatypes or secondary datatypes. In c programming language, the derived datatypes are created using the following concepts...

- Arrays
- Structures
- Unions
- Enumeration

Variables

Variables in c programming language are the named memory locations where user can store different values of same datatype during the program execution.

A variable name may contain letters, digits and underscore symbol. The following are the rules to specify a variable name...

1. Variable name should not start with digit.

2. Keywords should not be used as variable names.
3. Variable name should not contain any special symbols except underscore(_).
4. Variable name can be of any length but compiler considers only the first 31 characters of the variable name.

Declaration of Variable

Declaration of a variable tells to the compiler to allocate required amount of memory with specified variable name and allows only specified datatype values into that memory location.

Declaration Syntax:

Datatype variablename;

Example

int number;

The above declaration tells to the compiler that allocate **2 bytes** of memory with the name **number** and allows only integer values into that memory location.

Constants

In C programming language, a constant is similar to the variable but constant holds only one value during the program execution.

In C programmig language, constant can be of any datatype like integer, floating point, character, string and double etc.,

Integer constants

An integer constant can be a decimal integer or octal integer or hexa decimal integer. A decimal integer value is specified as direct integer value whereas octal integer value is prefixed with 'o' and hexa decimal value is prefixed with 'OX'.

Floating Point constants

A floating point constant must contain both integer and decimal parts. Some times it may also contain exponent part. When a floating point constant is represented in exponent form, the value must be suffixed with 'e' or 'E'.

Example

The floating point value **3.14** is represented as **3E-14** in exponent form.

Character Constants

A character constant is a symbol enclosed in single quotation. A character constant has a maximum length of one character.

Example

'A'
'2'
'+'

In C programming language, there are some predefined character constants called escape sequences.

String Constants

A string constant is a collection of characters, digits, special symbols and escape sequences that are enclosed in double quotations.

Creating constants in C

In c programming language, constants can be created using two concepts...

1. **Using 'const' keyword**
2. **Using '#define' preprocessor**

Pointers in C

In c programming language, we use normal variables to store user data values. When we declare a variable, the compiler allocates required memory with specified name. In c programming language, every variable has name, datatype, value, storage class, and address. We use a special type of variable called pointer to store the address of another variable with same datatype.

Accessing the Address of Variables

In c programming language, we use the **reference operator "&"** to access the address of variable. For example, to access the address of a variable **"marks"** we use **"&marks"**. We use the following printf statement to display memory location address of variable **"marks"**...

Example Code

```
printf("Address : %u", &marks) ;
```

Declaring Pointers (Creating Pointers)

In c programming language, declaration of pointer variable is similar to the creation of normal variable but the name is prefixed with ***** symbol. We use the following syntax to declare a pointer variable...

datatype *pointerName ;

Example Code

int *ptr ;

Assigning Address to Pointer

To assign address to a pointer variable we use assignment operator with the following syntax...

pointerVariableName = & variableName ;

Memory Allocation of Pointer Variables

Every pointer variable is used to store the address of another variable. In computer memory address of any memory location is an **unsigned integer** value. In c programming language, unsigned integer requires **2 bytes** of memory. So, irrespective of pointer datatype every pointer variable is allocated with 2 bytes of memory.

UNIT-2

Array – In the C programming language an array is a fixed sequenced collection of elements of the same data type. It is simply a grouping of like type data. In the simplest form, an array can be used to represent a list of numbers, or a list of names. Some examples where the concept of an array can be used:

- List of temperatures recorded every hour in a day , or a month, or a year.
- List of employees in an organization
- List of products and their cost sold by a store
- Table of daily rainfall data

Since an array provides a convenient structure for representing data, it is classified as one of the data structures in C language.

There are following types of arrays in the C programming language –

There are following types of arrays in the C programming language –

1. One – dimensional arrays
2. Multidimensional arrays

1D Arrays – A list of items can be given one variable name using only one subscript and such a variable is called single sub-scripted variable or one dimensional array.

Declaration of 1D Arrays –

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in the memory. The syntax form of array declaration is –

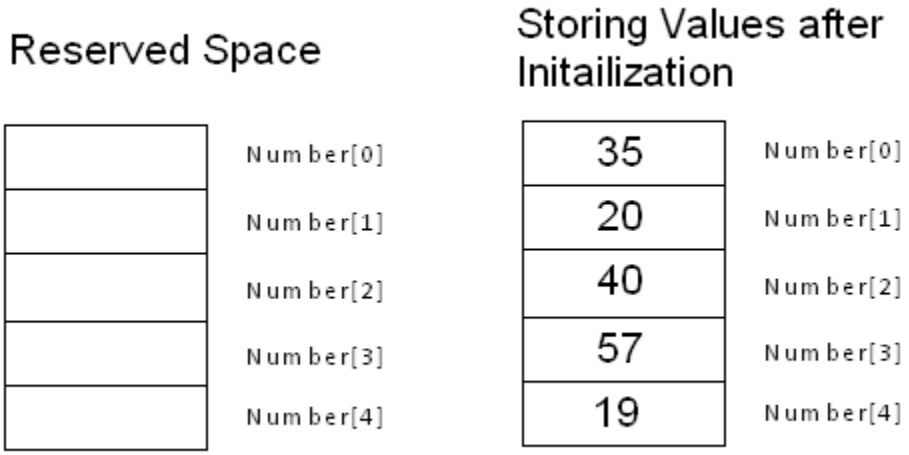
type variable-name[size];
Ex -
float height[50];
int groupt[10];
char name[10]

The type specifies the type of the element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored inside the array. Also the C programming language treats character strings simply as arrays of characters.

Now as we declare a array

int number[5];

Then the computer reserves five storage locations as the size of the array is 5 as shown below –



Initialization of 1D Array

After an array is declared, it's elements must be initialized. In C programming an array can be initialized at either of the following stages:

- At compile time
- At run time

Compile Time initialization

We can initialize the elements of arrays in the same was as the ordinary variables when they are declared. The general form of initialization of array is:

type array-name[size] = { list of values };

The values in the list are separated by commas. For ex, the statement

int number[3] = { 0,5,4 };

will declare the variable' number' as an array of size 3 and will assign the values to each element. If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero automatically.

Remember, if we have more initializers than the declared size, the compiler will produce an error.

Run time Initialization

An array can also be explicitly initialized at run time. For ex – consider the following segment of a C program.

```
for(i=0;i<10;i++)
{
    scanf(" %d ", &x[i] );
}
```

Above example will initialize array elements with the values entered through the keyboard. In the run time initialization of the arrays looping statements are almost compulsory. Looping statements are used to initialize the values of the arrays one by one by using assignment operator or through the keyboard by the user

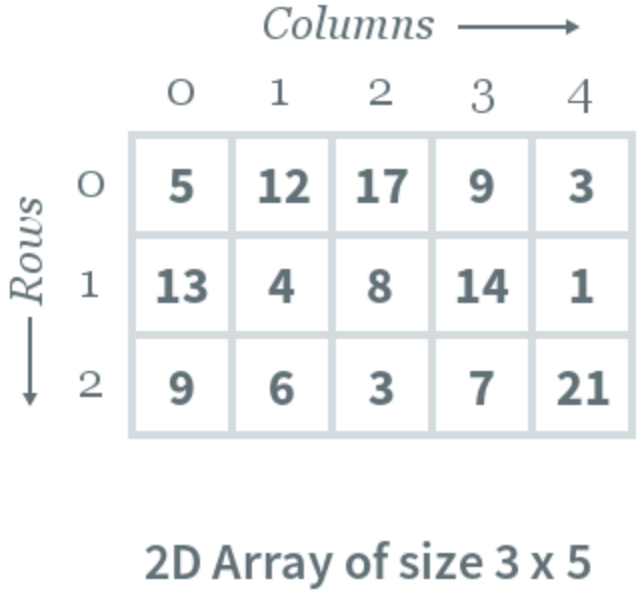
A multi-dimensional array

A multi-dimensional **array** is an array of arrays. 2-dimensional arrays are the most commonly used. They are used to store data in a tabular manner.

Consider following 2D array, which is of the size 3*5

For an array of size , the rows and columns are numbered from to and columns are numbered from to , respectively. Any element of the array can be accessed by where and . For example, in the following array, the value stored at is

.



2D Array of size 3 x 5

2D array declaration:

To declare a 2D array, you must specify the following:
Row-size: Defines the number of rows
Column-size: Defines the number of columns
Type of array: Defines the type of elements to be stored in the array i.e. either a number, character, or other such datatype. A sample form of declaration is as follows:

type arr[row_size][column_size]
array is declared as follows:
int arr[3][5];

2D array initialization:
An array can either be initialized during or after declaration. The format of initializing an array during declaration is as follows:

type arr[row_size][column_size] = {{elements}, {elements} ... }

An example in C++ is given below:

int arr[3][5] = {{5, 12, 17, 9, 3}, {13, 4, 8, 14, 1}, {9, 6, 3, 7, 21}};

Initializing an array after declaration can be done by assigning values to each cell of 2D array, as follows.

type arr[row_size][column_size]
arr[i][j] = 14

Basic Operations

Following are the basic operations supported by an array.

- **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.

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data 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 K^{th} 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

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

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 K^{th} position of LA.

1. Start
2. Set J = K

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

Search Operation

You can perform a search 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

Update Operation

Update operation refers to updating an existing element from the array at a given 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 update an element available at the K^{th} position of LA.

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

UNIT-3

Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.

- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

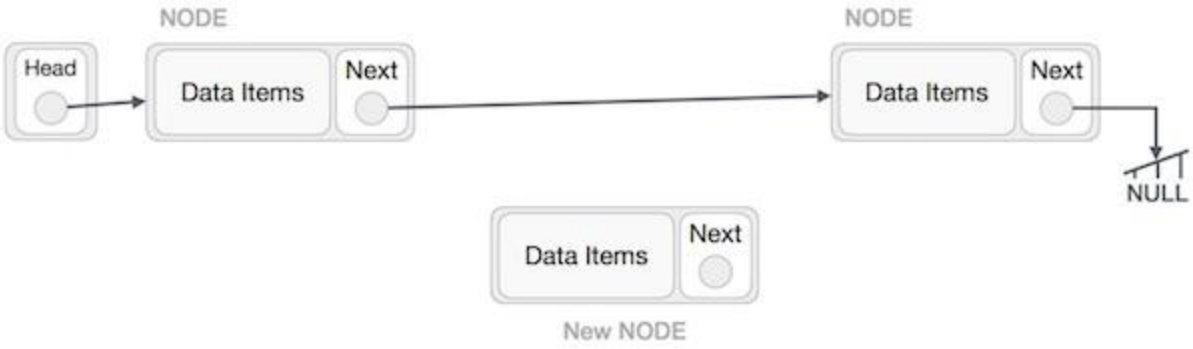
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

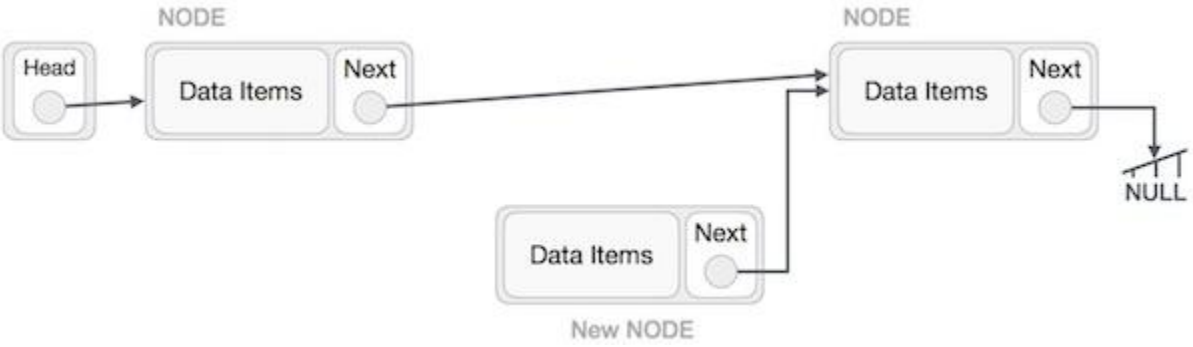
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

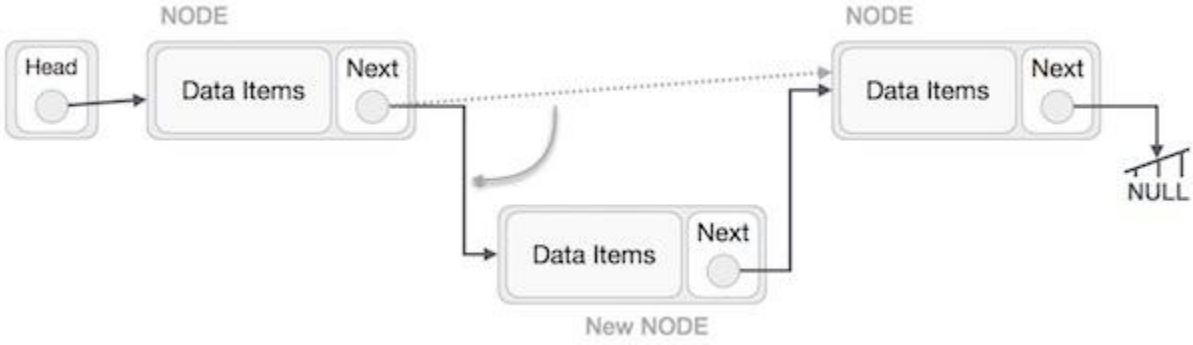
NewNode.next −> RightNode;

It should look like this −



Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

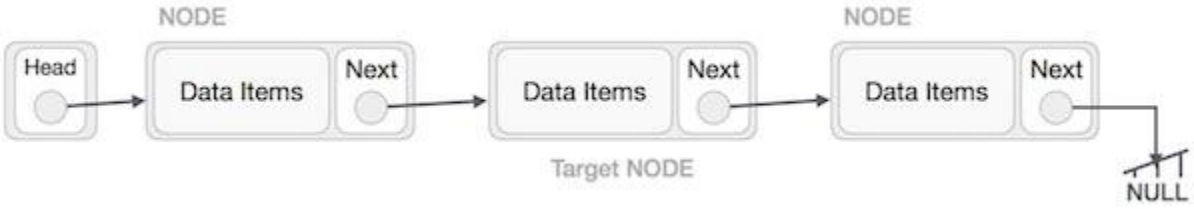


This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

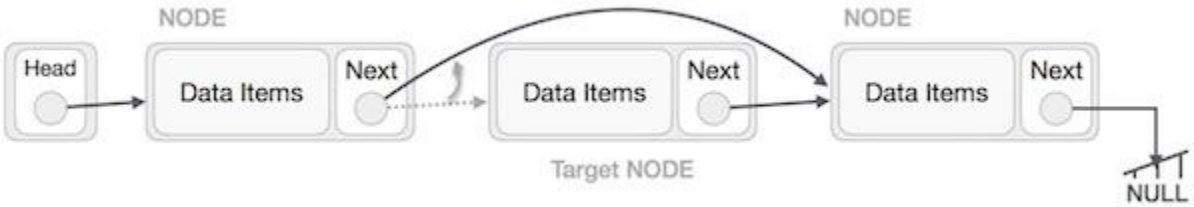
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



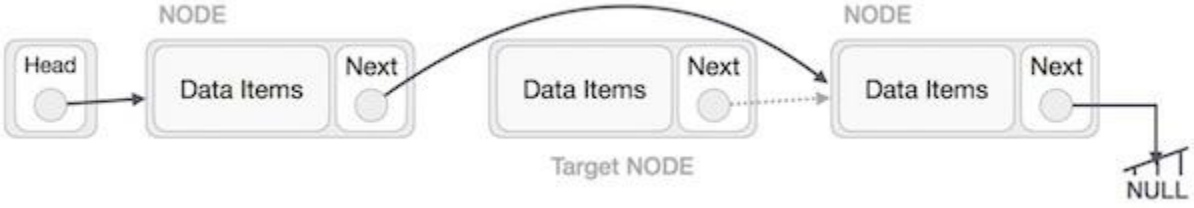
The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

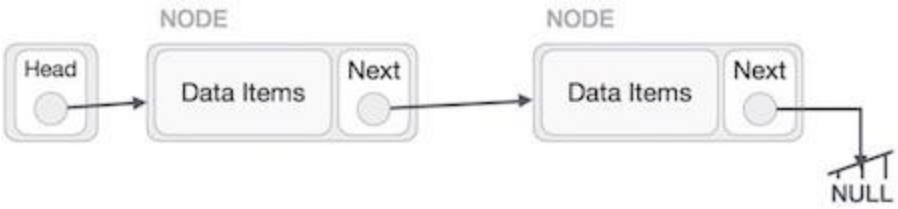


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

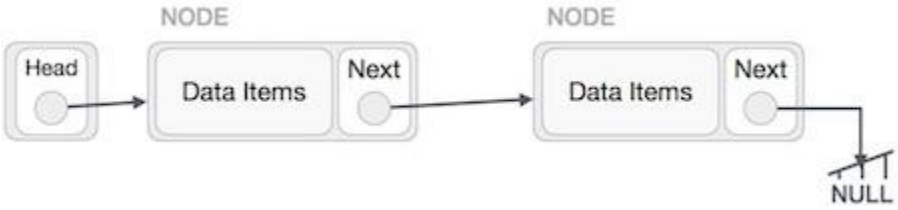


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

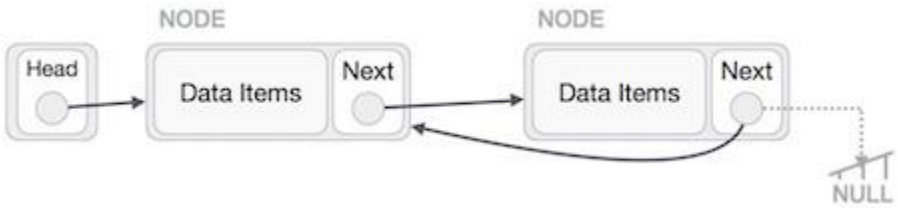


Reverse Operation

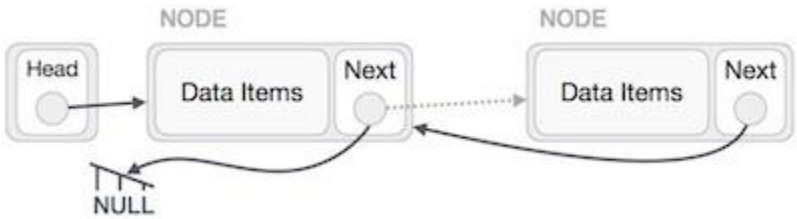
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



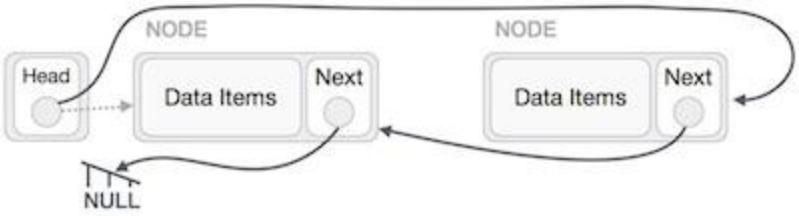
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



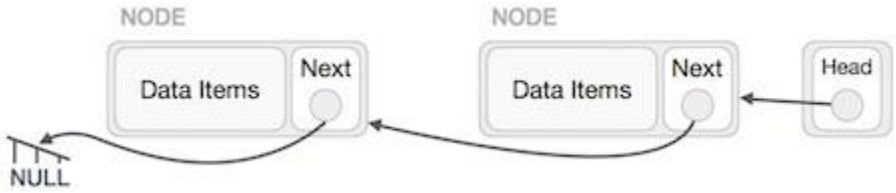
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



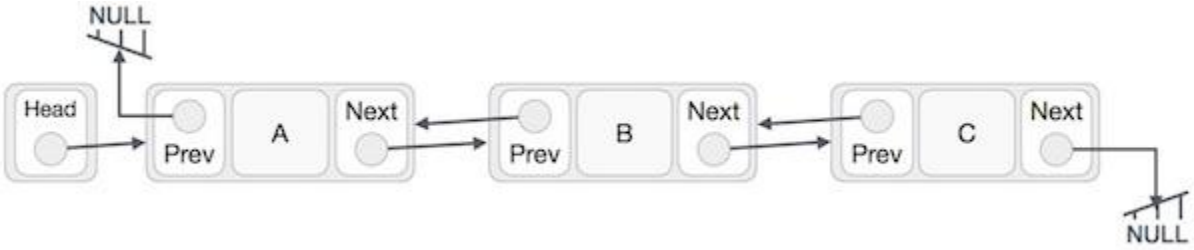
We'll make the head node point to the new first node by using the temp node.



Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **Prev** − Each link of a linked list contains a link to the previous link called Prev.
- **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List Representation



As per the above illustration, following are the important points to be considered.

- Doubly Linked List contains a link element called first and last.
- Each link carries a data field(s) and two link fields called next and prev.
- Each link is linked with its next link using its next link.
- Each link is linked with its previous link using its previous link.
- The last link carries a link as null to mark the end of the list.

Basic Operations

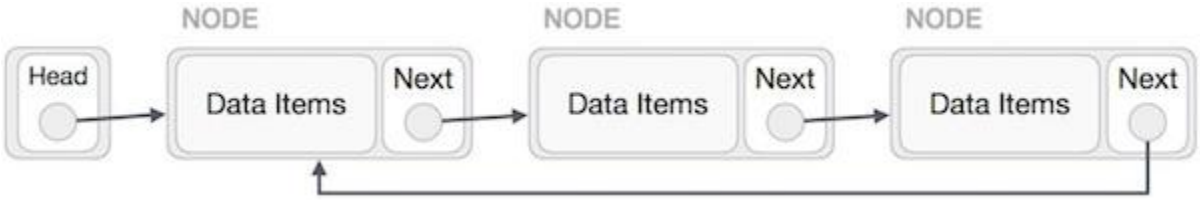
Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Insert Last** − Adds an element at the end of the list.
- **Delete Last** − Deletes an element from the end of the list.
- **Insert After** − Adds an element after an item of the list.
- **Delete** − Deletes an element from the list using the key.
- **Display forward** − Displays the complete list in a forward manner.
- **Display backward** − Displays the complete list in a backward manner.

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

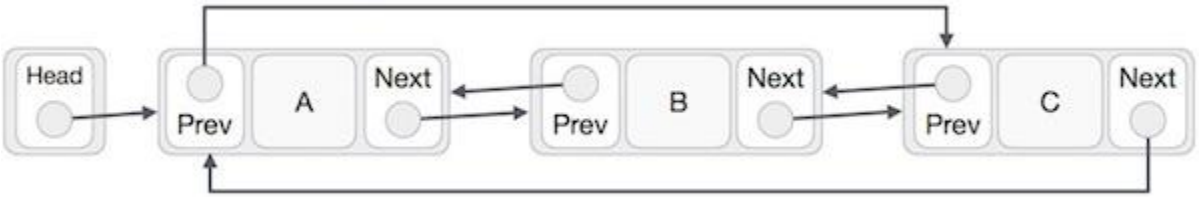
Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



As per the above illustration, following are the important points to be considered.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Basic Operations

Following are the important operations supported by a circular list.

- **insert** − Inserts an element at the start of the list.

- **delete** − Deletes an element from the start of the list.
- **display** − Displays the list.

UNIT-4

Stack

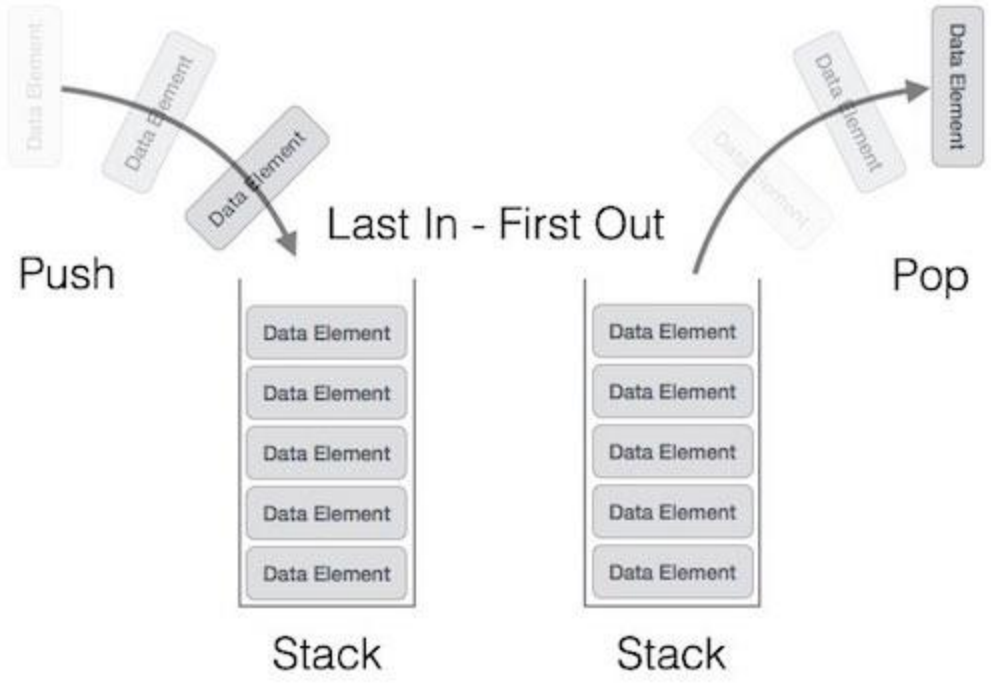
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

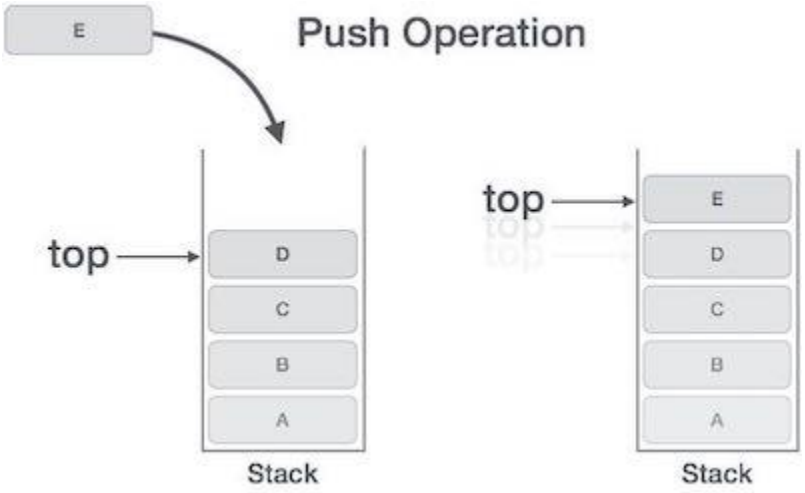
- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data

end procedure
```

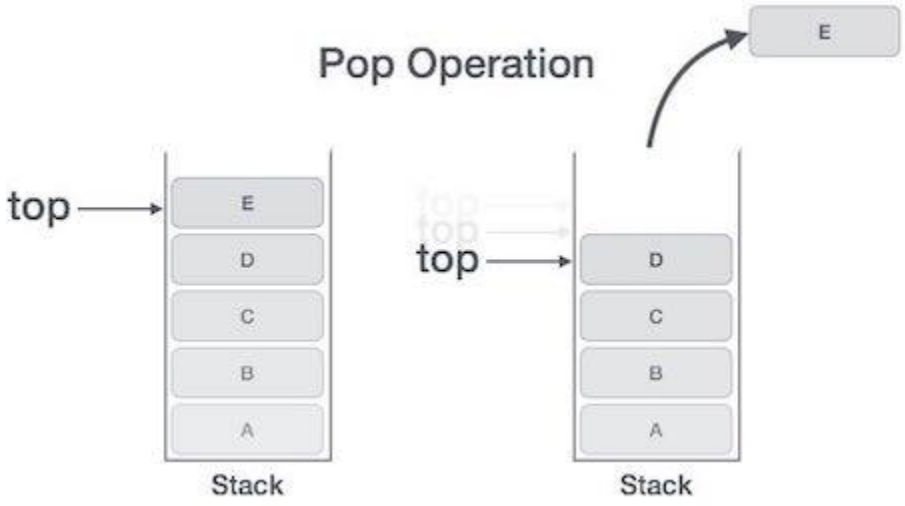
Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is

decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Applications of Stack

Expression Evaluation
Stack is used to evaluate prefix, postfix and infix expressions.

Expression Conversion
An expression can be represented in prefix, postfix or infix notation. Stack can be used to convert one form of expression to another.

Syntax Parsing
Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

Backtracking
Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

Parenthesis Checking
Stack is used to check the proper opening and closing of parenthesis.

String Reversal
Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack.

Function Call
Stack is used to keep information about the active functions or subroutines.

Expression Representation

There are three popular methods used for representation of an expression:

Infix	**A + B**	**Operator between operands.**
Prefix	+ AB	Operator before operands.
Postfix	AB +	Operator after operands.

1. Conversion of Infix to Postfix

Example: Suppose we are converting 3*3/(4-1)+6*2 expression into postfix form.

Following table shows the evaluation of Infix to Postfix:

Expression	Stack	Output
3	Empty	3
*	*	3
3	*	33
/	/	33*
(	/(	33*
4	/(	33*4
-	/(-	33*4
1	/(-	33*41
)	-	33*41-
+	+	33*41-/
6	+	33*41-/6
*	+*	33*41-/62
2	+*	33*41-/62
	Empty	**33*41-/62*+**

So, the Postfix Expression is **33*41-/62*+**

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

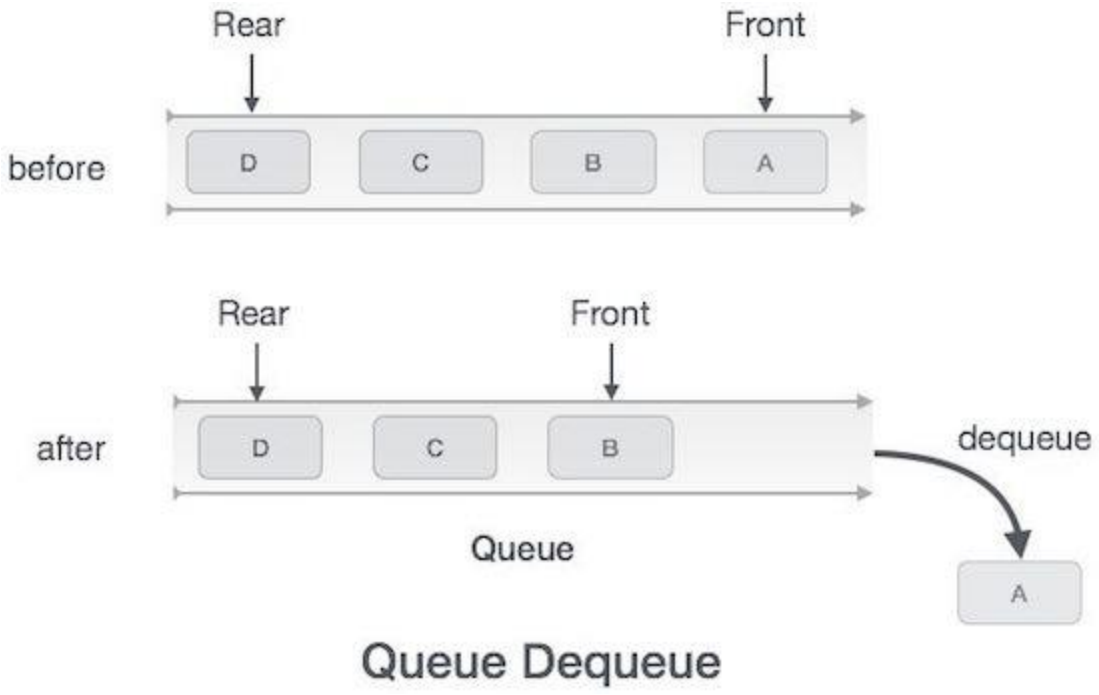
Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

procedure dequeue

```
  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

Applications of Queue

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
2. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

There are four types of Queue:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Dequeue (Double Ended Queue)

Simple Queue

Simple queue defines the simple operation of queue in which insertion occurs at the rear of the list and deletion occurs at the front of the list.

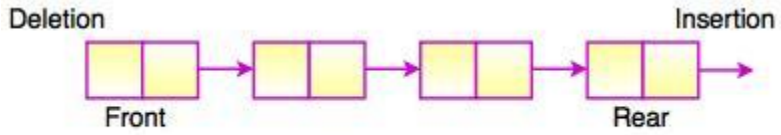


Fig. Simple Queue

Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue.

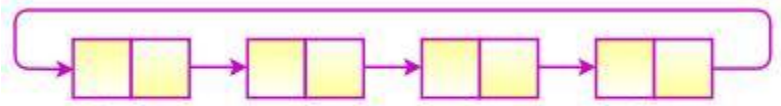


Fig. Circular Queue

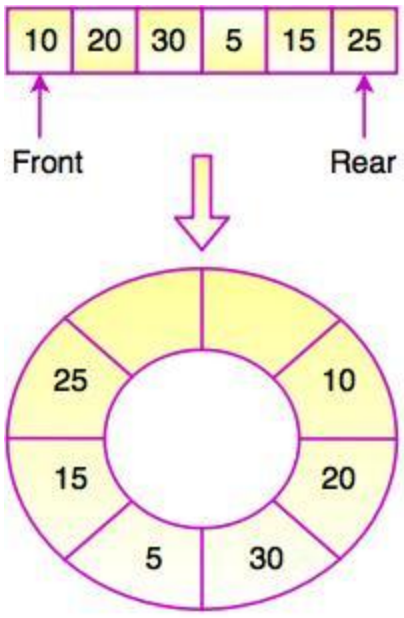


Fig. Circular Queue

The above figure shows the structure of circular queue. It stores an element in a circular way and performs the operations according to its FIFO structure.

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.

Priority Queue

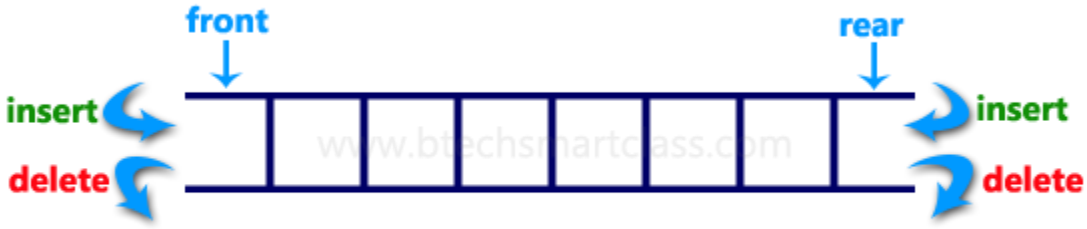
- Priority queue contains data items which have some preset priority. While removing an element from a priority queue, the data item with the highest priority is removed first.
- In a priority queue, insertion is performed in the order of arrival and deletion is performed based on the priority.

Applications of Priority Queue:

1) CPU Scheduling
2) Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

Double Ended Queue Data structure

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (**front** and **rear**). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

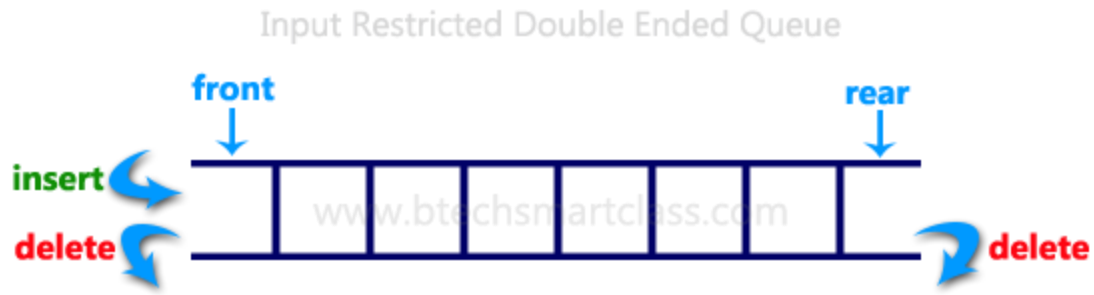


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

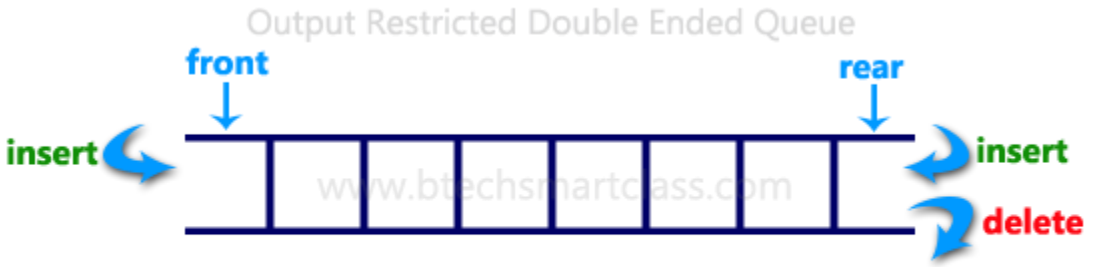
Input Restricted Double Ended Queue

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



Output Restricted Double Ended Queue

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



Recursion

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Base condition in recursion

In recursive program, the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

How a particular problem is solved using recursion?

The idea is represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop recursion. For example, we compute factorial n if we know factorial of (n-1). Base case for factorial would be n = 0. We return 1 when n = 0.