Data Structure

Data Management

- Data management is the practice of collecting, keeping, and using data securely, efficiently, and cost-effectively.
- The goal of data management is to help people, organizations, and connected things optimize the use of data within the bounds of policy and regulation so that they can make decisions and take actions that maximize the benefit to the organization.

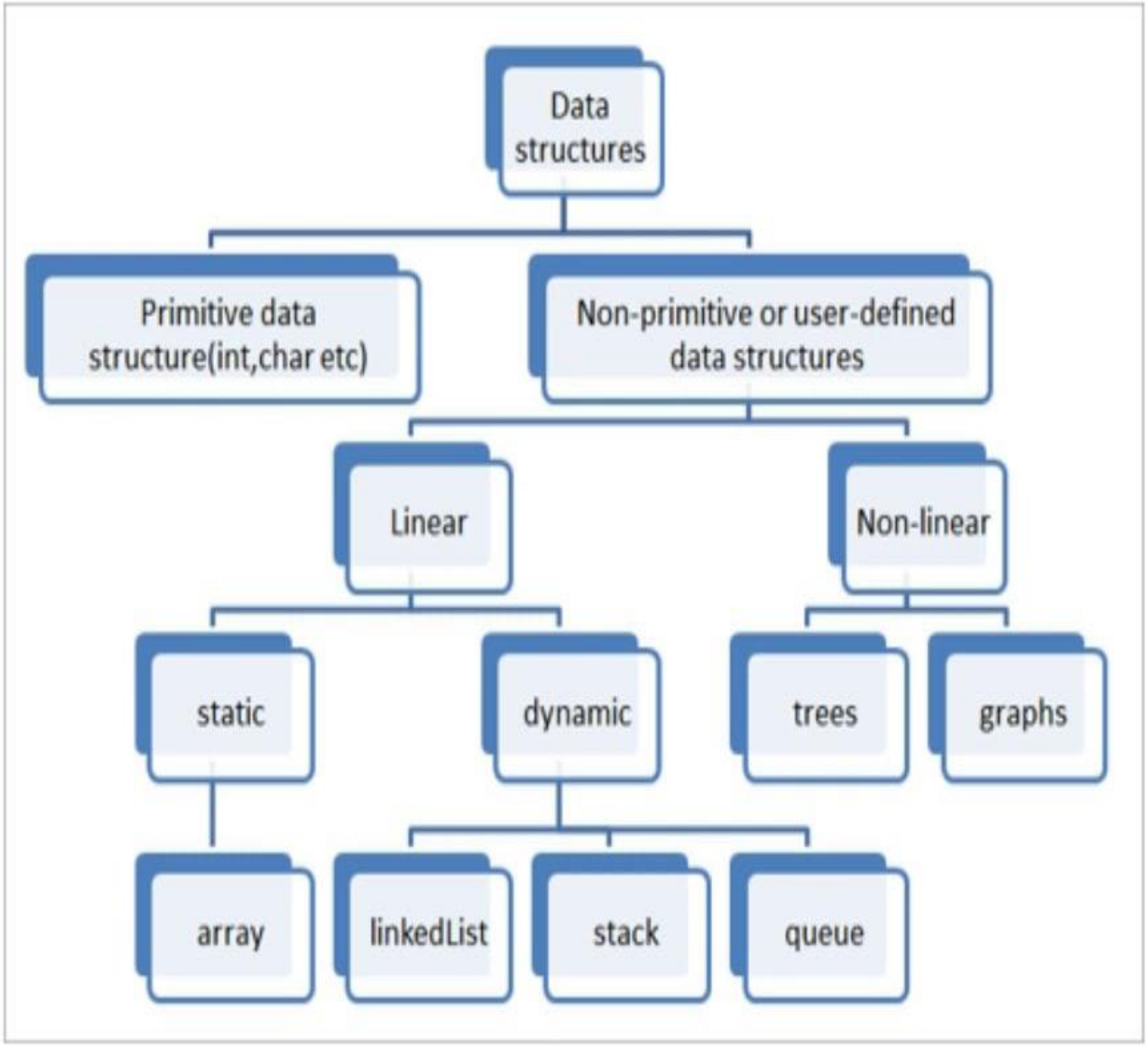
Data

- 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.
- That means Data is collection of information in row form.
- **Data Object:** is a term that refers to a set of elements such a set may be finite or infinite. Ex. Set of students studying in second year.
- **Data Type:** Built in, Derived data Type and User defined data type.

Data Structure

- A data structure is a storage that is used to store and organize data.
- It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
- In simple term , Organized collection of data in a particular format is called Data Structures.

Data Structure Classification


Data structures
Primitive data structure(int,char etc)
Non-primitive or user-defined data structures
Linear
Non-linear
static
dynamic
trees
graphs
array
linkedList
stack
queue

Primitive Data Structure

- Can hold single value in a specific location.
- Ex. float, int, char.

Non-Primitive Data Structure

- It is a kind of data structure that can hold multiple values either in a continuous or random location.
- There are two types of Non primitive data structure:

1. Linear
2. Non-Linear

Linear Data Structure

- Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
- That means we just need to start from one place and will find other data in a sequence.
- Examples of linear data structures are array, stack, queue, linked list, etc.

1. Static
2. Dynamic

Static and Dynamic Data Structure

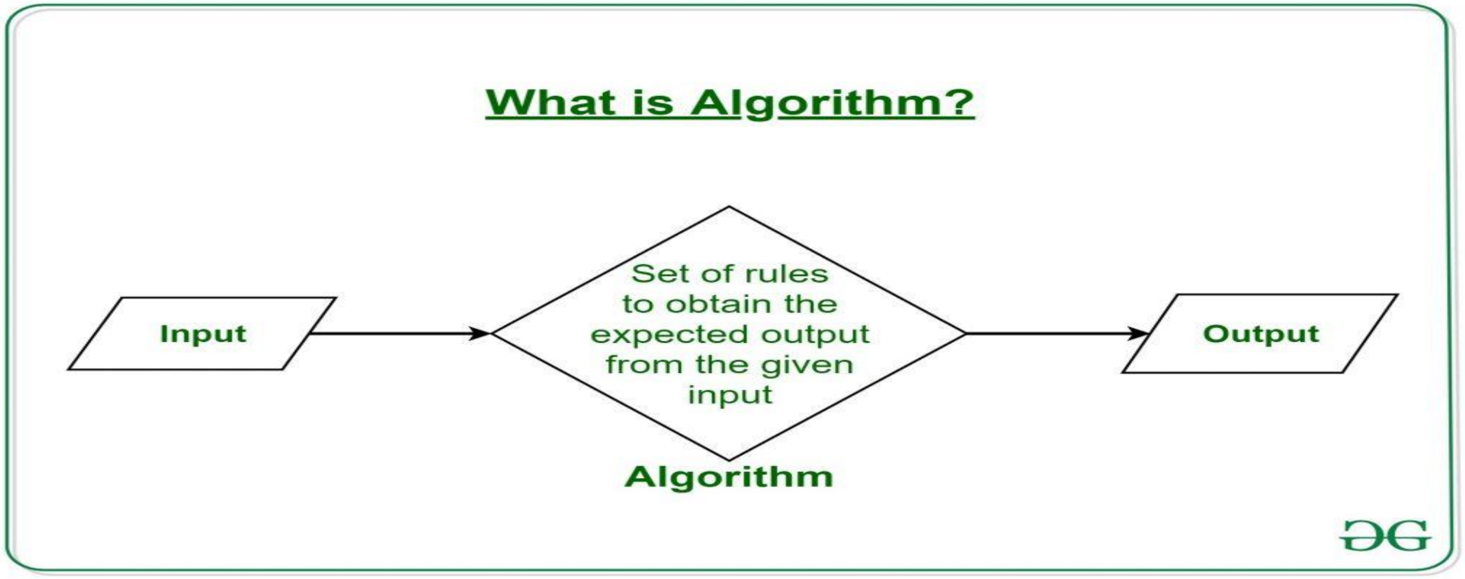
- **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
- An example of this data structure is an array.
- **Dynamic data structure:** In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime.
- Examples of this data structure are queue, stack, etc.

Non-Linear Data Structure

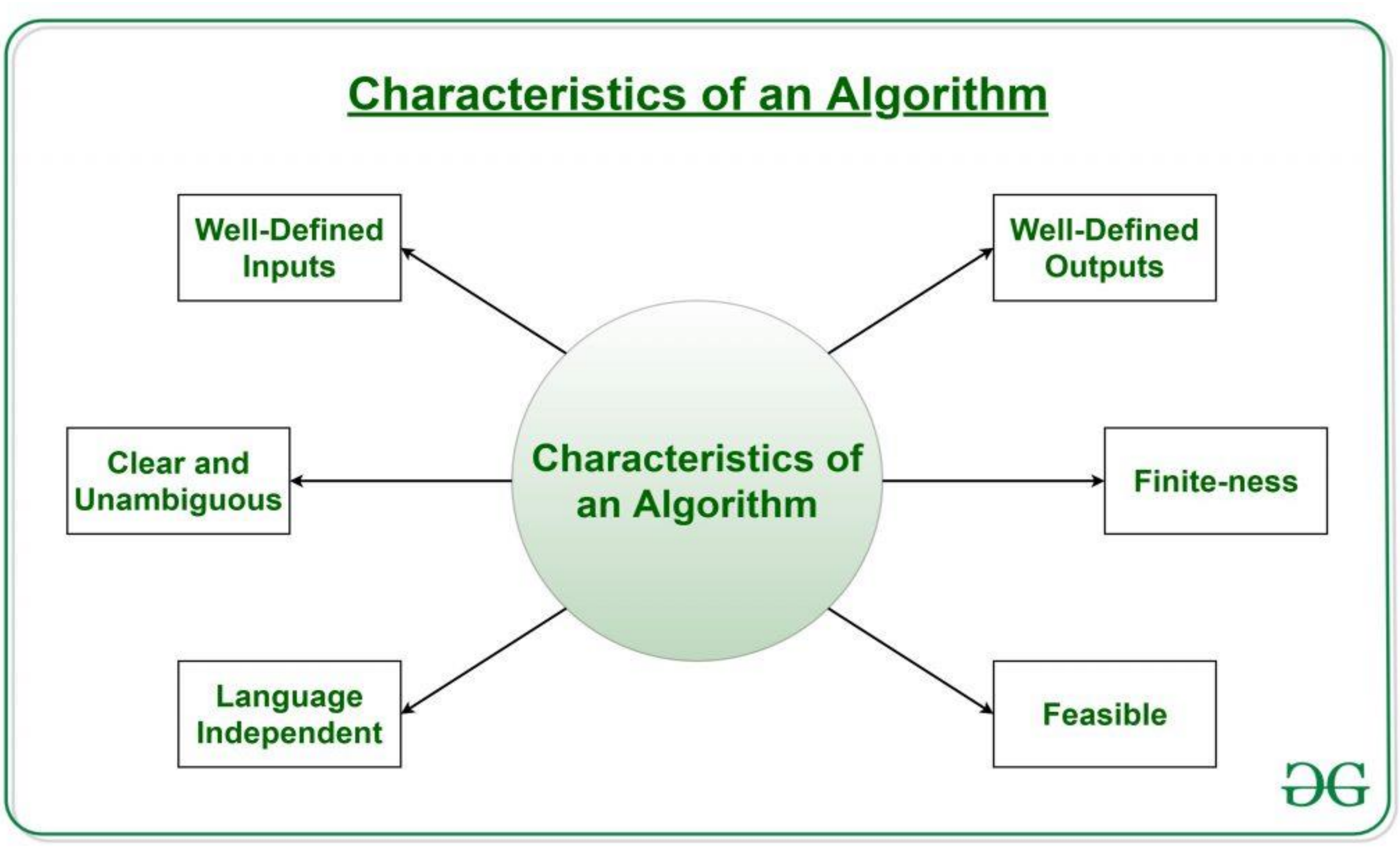
- Data structures where data elements are not placed sequentially or linearly are called non-linear data structures.
- In a non-linear data structure, we can’t traverse all the elements in a single run only.
- Each element can have multiple path to connect to other elements.
- Examples of non-linear data structures are trees and graphs.

Algorithm

- Algorithm is finite set of instruction for performing a particular task.
- The instruction are nothing but the statements in simple English language.


What is Algorithm?
Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Characteristics of Algorithm



Characteristics of Algorithm

- **Clear and Unambiguous:** The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.

Characteristics of Algorithm

- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Structure of Algorithm

Algorithm Heading
(Consists Name of Algorithm, Problem description, Input and Output)

Algorithm Body
(Consists of logical body)

Example

- **Adds two numbers and store the result in third variable:**
- **Step1** Start
- **Step2** Read the first no. is variable ‘a’.
- **Step3** Read the second no. is variable ‘b’.
- **Step4** Perform addition of both numbers and store result in variable ‘c’.
- **Step 5** Print the value of ‘c’ as a result of addition.
- **Step 6** Stop

Difference between Algorithm and Program

Algorithm	Program
It is step by step procedure of how to solve a problem.	Program is an implemented coding of solution to a problem on the Algorithm.
Designing phase .	Implementation phase.
It has to specify a finiteness condition.	It does not have to specify the finiteness condition.

Pseudo code

- An **algorithm** is defined as a well-defined sequence of steps that provides a solution for a given problem, whereas a pseudocode is one of the methods that can be used to represent an algorithm.
- While algorithms are generally written in a natural language or plain English language, **pseudocode** is written in a format that is similar to the structure of a high-level programming language.
- **Program** on the other hand allows us to write a code in a particular programming language.

Example

- **Find out maximum element in given array size n:**
- **Algorithm:ArrayMax (a[1….n],n)**

```
1. Max=a[1]
2. For i=2 to n do
3. If a[i]>max then
4. max<-a[i]
5. End if
6. End for
7. Return max
```

Abstract Data Type(ADT)

- Abstraction is an act of representing essential features.
- Data type defines type of data.
- Example: If i want to store date.
- ADT is a mathematical model that logically represents datatype.
- ADT only tells essential features without including background details.

Abstract Data Type(ADT)

- It represents set of data and set of operation that can be performed on the data.
- Example, List ADT, Stack ADT etc.

Performance Analysis of Algo.

- Different Algo. May complete the task with different set of instruction in less or more time and space.
- It is important to know how much of **particular resource** is required for given algorithm.
- Number of techniques are possible:

1. Measuring space complexity
2. Measuring time complexity
3. Measuring runtime complexity
4. Measuring input size
5. Computing best case , worst case and average case.

Complexity of Algorithm

- The efficiency of an algorithm can be decided by measuring the performance of an algorithm.
- We can measure performance of algorithm by computing two fectors:

1. Amount of time required by an algorithm to execute(Time Complexity).
2. Amount of storage required by an algorithm.(Space complexity).

Time Complexity

- The time complexity of a program is an amount of time needed to complete program execution.

$$t(n)= \text{compile time} + \text{runtime}$$

- Where time taken by program.

Space Complexity

- Space complexity of program is an amount of memory space which is needed to load program.so for find out the space complexity , we want to seem of following component:

1. A fixed part then include space for entire code.
2. A variable part that consists space needed by the component variable whose size is dependent on particular instance of problem which is being solved and stack space used by recursive procedure.

Analysis case(Case of analysis)

- **Best case(very rarely used):** Define the input for which algorithm takes less time or minimum time.
- Time complexity when algorithm runs for shortest time. calculate lower bound.
- For example: In the linear search when search data is present at the first location of large data then the best case occurs.
- **Worst Case(mostly used):** Define the input for which algorithm takes a long time or maximum time. Time complexity when algorithm runs for longest time. Calculate upper bound.

Analysis case(Case of analysis)

- For example: In the linear search when search data is not present at all then the worst case occurs.
- **Average case(rarely used):**In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.

Average case = all random case time / total no of case

THANK YOU

Linear Data Structure:Array

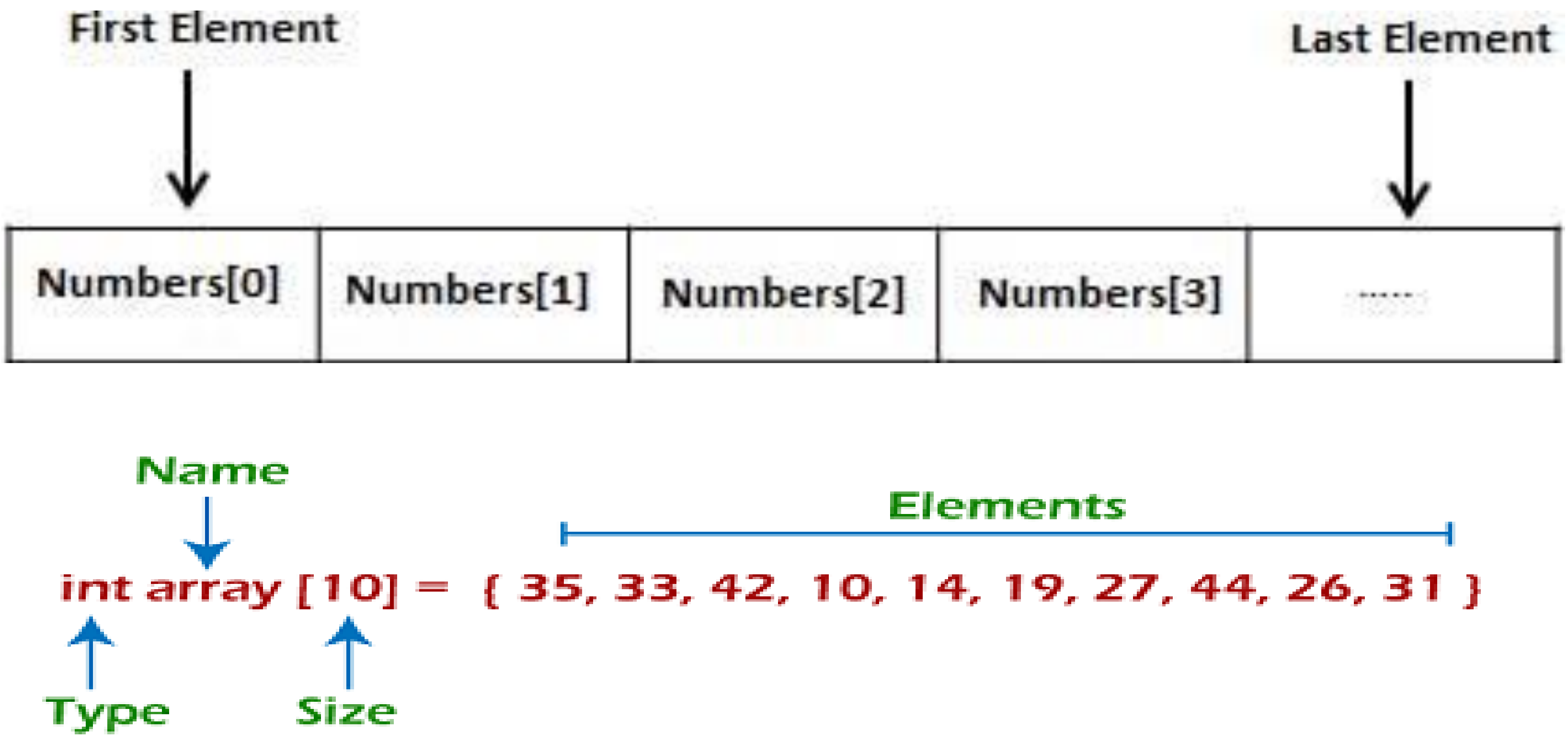
Outline

- Representation of arrays,
- Applications of arrays,
- Sparse matrix and its representation

Array

- An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations.
- Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.
- You can store group of data of same data type in an array.
- It is a kind of data structure that can store a fixed-size sequential collection of elements of the same type.
- Array might be belonging to any of the data types.

For Example,


First Element
Last Element
Numbers[0]
Numbers[1]
Numbers[2]
Numbers[3]
.....
Name
Elements
int array [10] = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 }
Type
Size

Types of Arrays

There are 2 types of C arrays. They are,

1. One dimensional array
2. Multi dimensional array
   1. Two dimensional array
   2. Multi dimensional array

Memory Allocation of 1 D Array

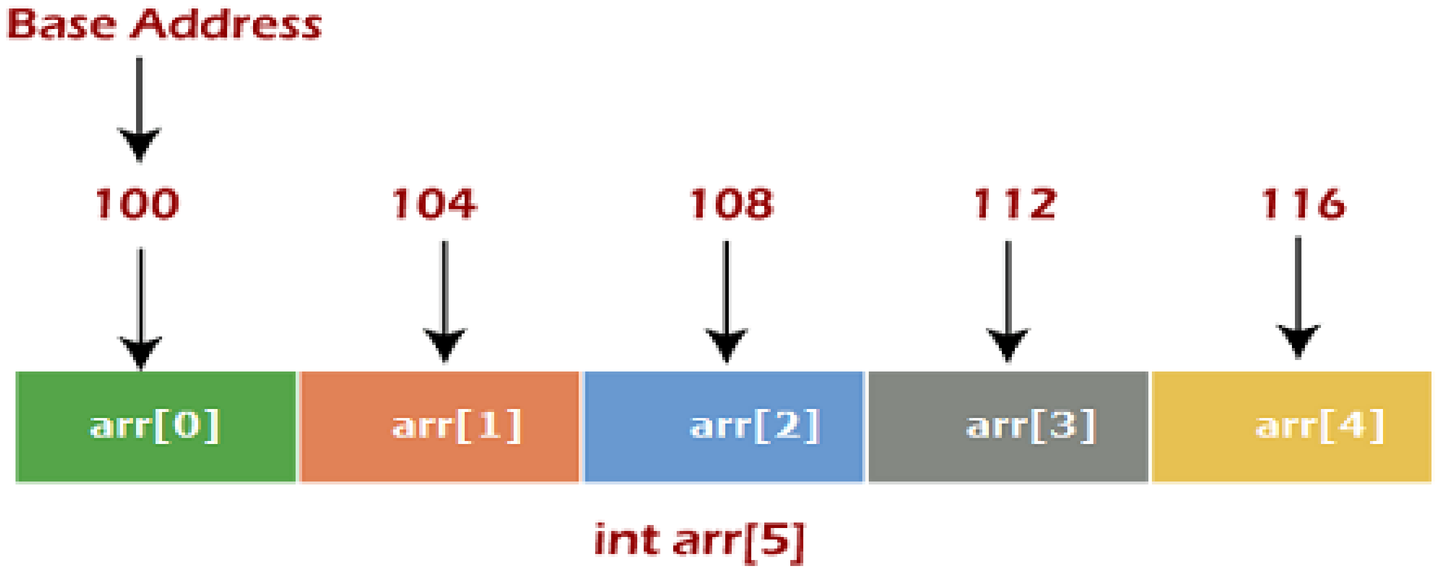
- All the data elements of an array are stored at contiguous locations in the main memory.
- The name of the array represents the base address or the address of the first element in the main memory.
- Each element of the array is represented by proper indexing.

Memory Allocation of 1 D Array

- We can define the indexing of an array in the below ways -

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

Memory Allocation of 1 D Array



Access Elements from 1 D Array

- We required the information given below to access any random element from the array –
  1. Base Address of the array.
  2. Size of an element in bytes.
  3. Type of indexing, array follows.

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

For Example,

- We can understand it with the help of an example -
- Suppose an array, A[-10 ..... +2] having Base address (BA) = 999 and size of an element = 2 bytes, find the location of A[-1].

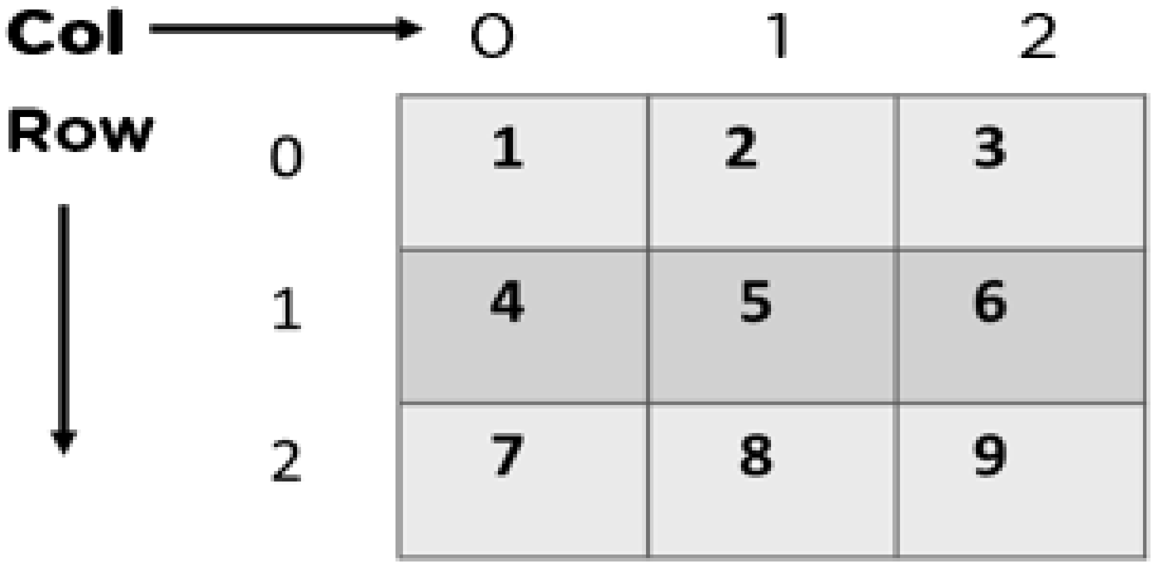
$$\begin{aligned} L(A[-1]) &= 999 + 2 \times [(-1) - (-10)] \\ &= 999 + 18 \\ &= 1017 \end{aligned}$$

Operations on Array

- Traversal
- Insertion
- Deletion
- Search
- Update
- Sorting

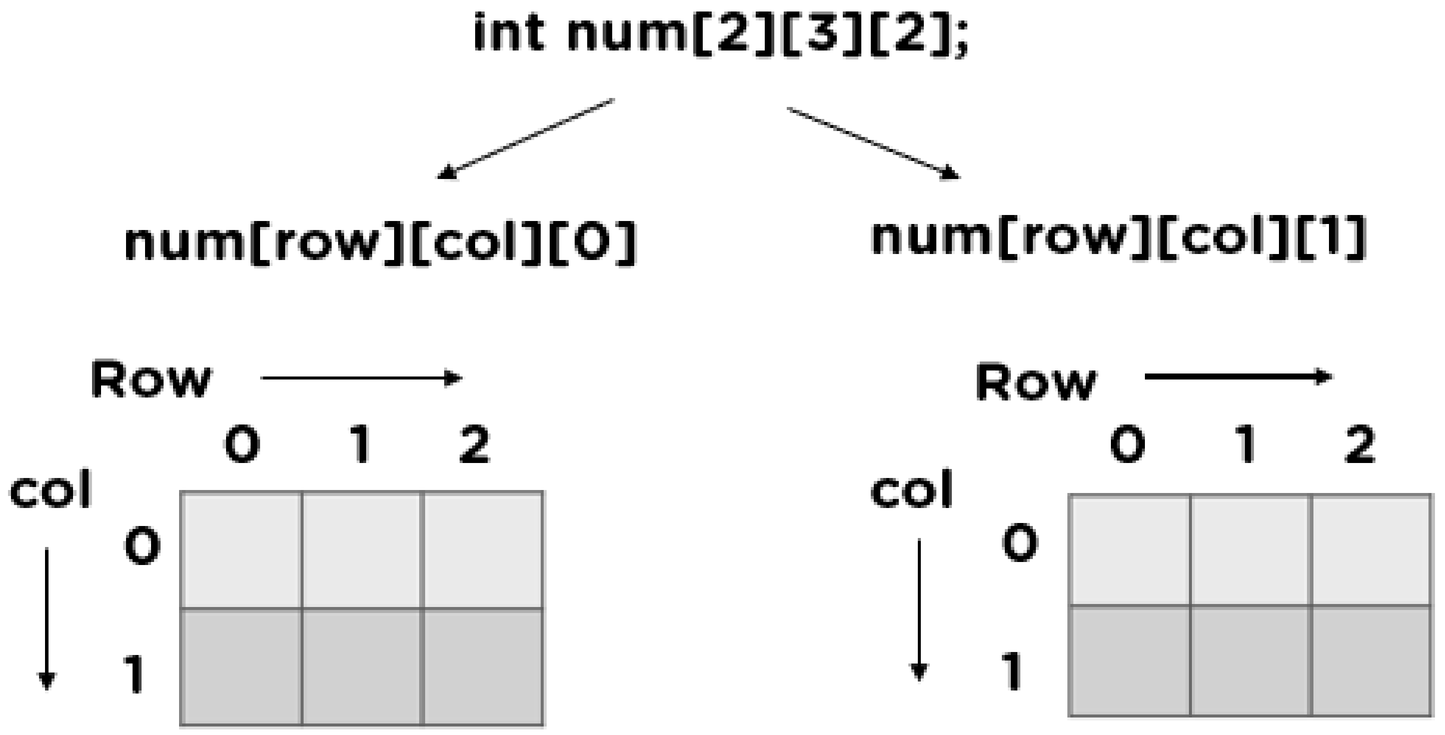
Two-Dimensional Array

- 2-D array like a table where each cell contains elements.
- The 2D array is organized as matrices which can be represented as the collection of rows and columns.



Three-Dimensional Array

- 3-D array like a cuboid made up of smaller cuboids where each cuboid can contain an element.



Array Declaration

- How to declare an array?

```
dataType arrayName[arraySize];
```

- For example,

```
float mark[5]; or
char str[5];
```

Array Initialization &Accessing

- How to initialize an array?

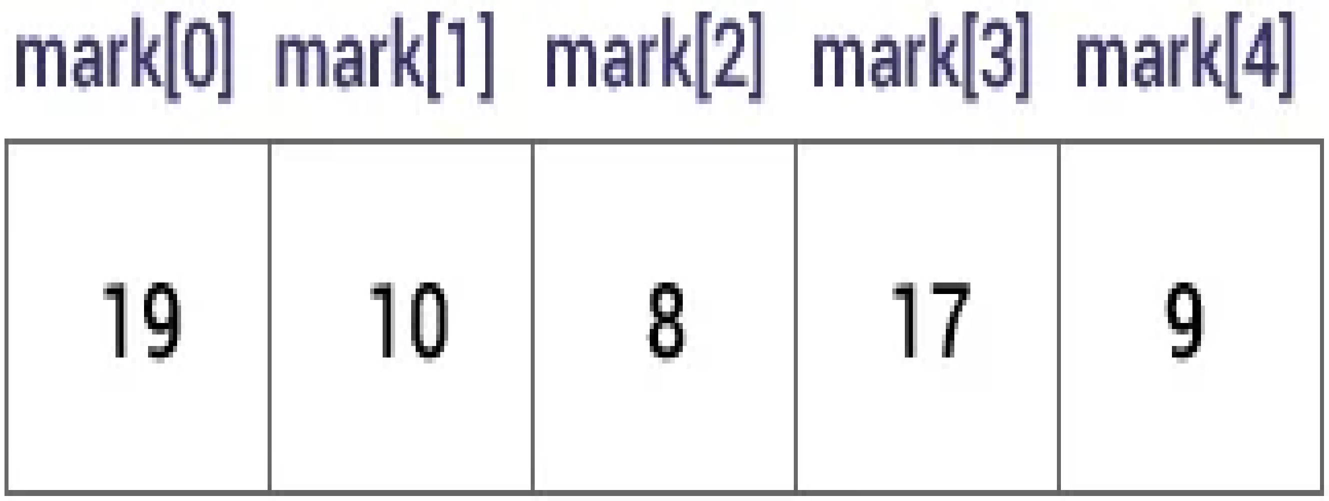
```
int age[5]={0, 1, 2, 3, 4}; or age[1]=1;
char str[10]={‘H’,‘a’,‘i’}; or
char str[0] = ‘H’;
```

- You can access elements of an array by indices.

```
age[1]; /*1 is accessed*/
str[1]; /*1 is accessed*/
```

For Example

- int mark[5] = {19, 10, 8, 17, 9};



Access elements out of its bound!

- Suppose you declared an array of 10 elements. Let's say,

  int testArray[10];

- You can access the array elements from testArray[0] to testArray[9] and you should never access elements of an array outside of its bound.

Two-dimensional Arrays

- The two-dimensional array is simplest form of multidimensional array. It is array of array.
- To declare a two-dimensional integer array of size [x][y], you would write something as follows –

```
type name[x][y];
```

- For example,

```
int twodim[5][10];
```

Initialization of 2-dimensional Arrays

- 2-dimensional array may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = { {0, 1, 2, 3} ,{4, 5, 6, 7} ,{8, 9, 10,
11}};      or
a[0][0]=0;
a[0][1]=1;
```

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

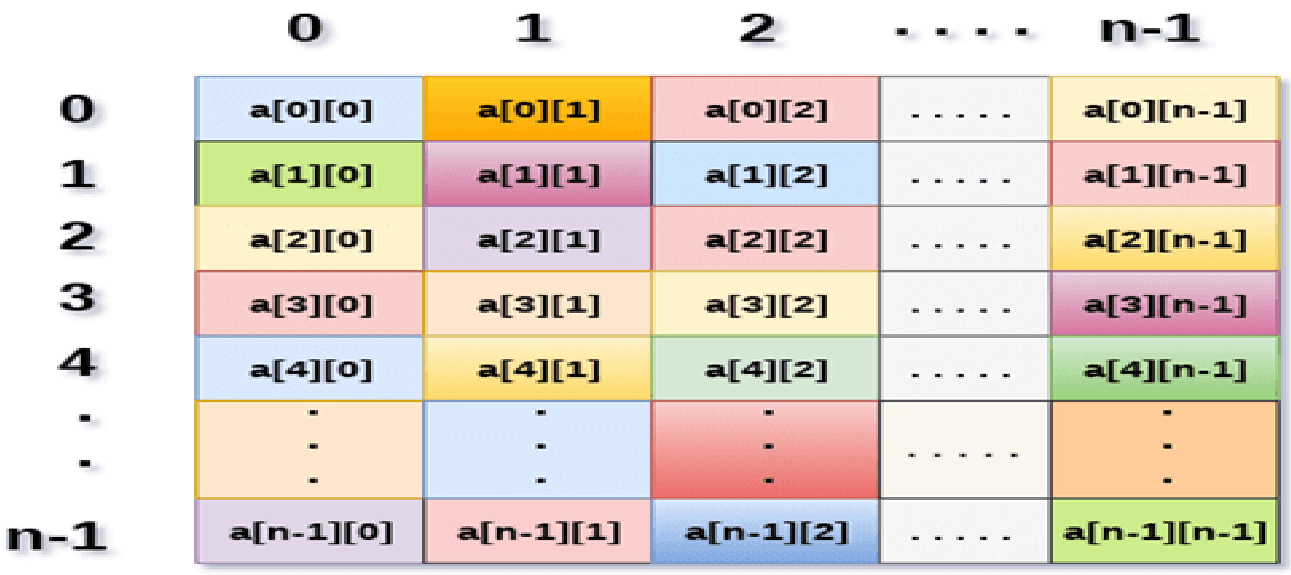
Accessing 2-Dimensional Array elements

- Two dimensional array element is accessed by using subscripts. i.e. , row index and column index of array.
- For example,

```
int val =a[0][0];
int val1= a[0][1];
```

For Example

- A two-dimensional array can be considered as a table which will have x number of rows and y number of columns.
- A two-dimensional array **a**, which contains three rows, and four columns can be shown as follows –



	0	1	2	. . . .	n-1
0	a[0][0]	a[0][1]	a[0][2]	. . . . .	a[0][n-1]
1	a[1][0]	a[1][1]	a[1][2]	. . . . .	a[1][n-1]
2	a[2][0]	a[2][1]	a[2][2]	. . . . .	a[2][n-1]
3	a[3][0]	a[3][1]	a[3][2]	. . . . .	a[3][n-1]
4	a[4][0]	a[4][1]	a[4][2]	. . . . .	a[4][n-1]
. .	. . .	. . .	. . .	. . . . . .	. . .
n-1	a[n-1][0]	a[n-1][1]	a[n-1][2]	. . . . .	a[n-1][n-1]

a[n][n]

Memory Allocation of 2 D Array

- However, 2 D arrays exists from the user point of view.
- In computer memory, the storage technique for 2D array is similar to that of an one dimensional array.
- The size of a two dimensional array is equal to the multiplication of number of rows and the number of columns present in the array.
- We do need to map two dimensional array to the one dimensional array in order to store them in the memory.

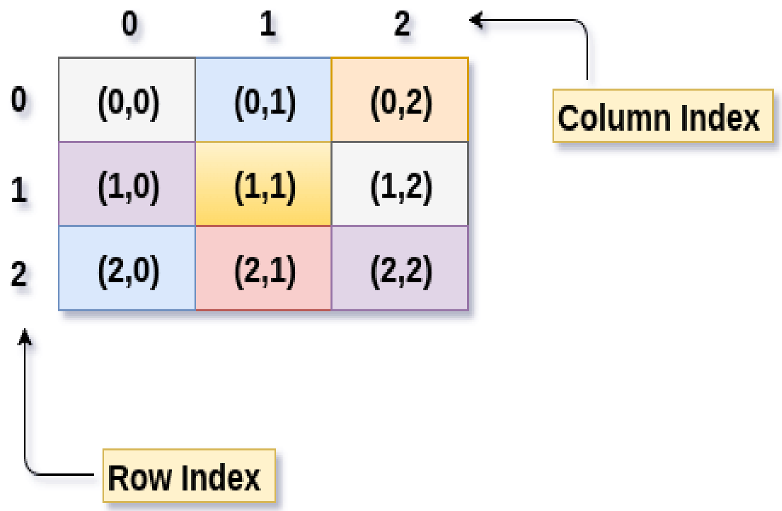
Mapping 2D array to 1D array

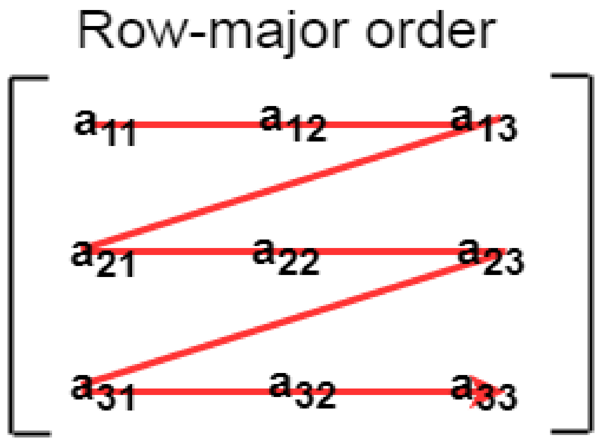
- There are two main techniques of storing 2D array elements into memory:

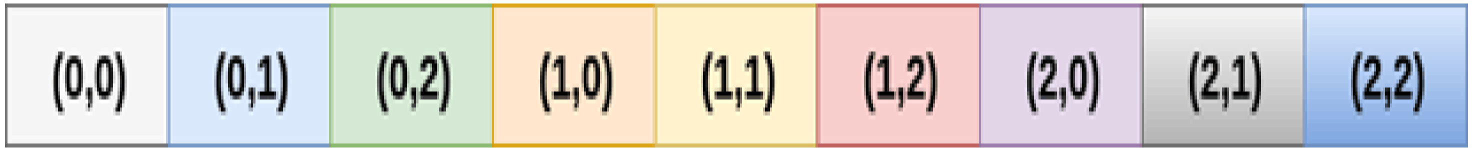
  1. Row Major ordering.
  2. Column Major ordering.

Row Major ordering

- In row major ordering, all the rows of the 2D array are stored into the memory contiguously.

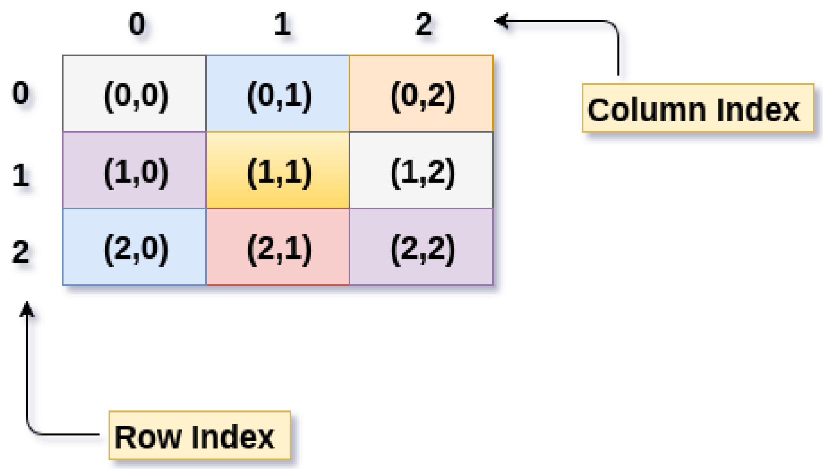




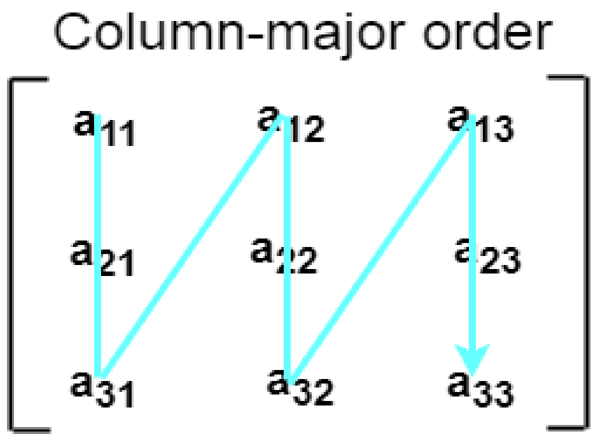


Column Major ordering

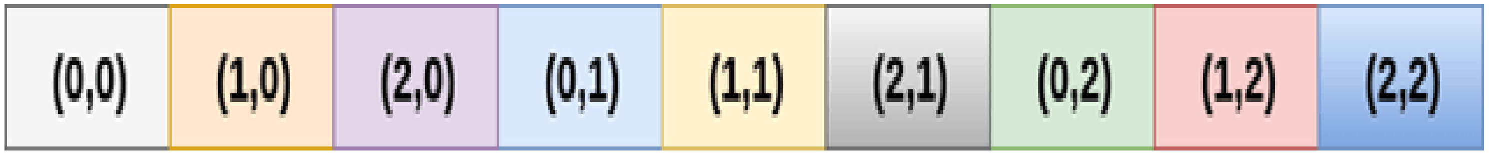
- According to the column major ordering, all the columns of the 2D array are stored into the memory contiguously.



	0	1	2
0	(0,0)	(0,1)	(0,2)
1	(1,0)	(1,1)	(1,2)
2	(2,0)	(2,1)	(2,2)



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$



(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

Applications of Array

- Arrays are used to implement data structures like a stack, queue, etc.
- Arrays are used for matrices and other mathematical implementations.
- Arrays are used in lookup tables in computers.
- Arrays can be used for CPU scheduling.
- **Real-Time Application:** Contact lists on mobile phones.
- Matrices use arrays which are used in different fields like image processing, computer graphics, and many more.

Applications of Array

- Arrays are used in online ticket booking portals.
- Pages of book.
- IoT applications use arrays as we know that the number of values in an array will remain constant, and also that the accessing will be faster.
- It is also utilized in speech processing, where each speech signal is represented by an array.
- The viewing screen of any desktop/laptop is also a multidimensional array of pixels.

Sparse Matrix

- In computer programming, a matrix can be defined with a 2-dimensional array.
- Any array with 'm' columns and 'n' rows represent a m X n matrix.
- There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.
- When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix.

Sparse Matrix

- For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements.
- In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero.
- That means, totally we allocate 100 X 100 X 4 = 40000 bytes of space to store this integer matrix and to access these 10 non-zero elements we must make scanning for 10000 times.
- To make it simple we use the following sparse matrix representation.

Sparse Matrix Representation

- A sparse matrix can be represented by using TWO representations, those are as follows:-

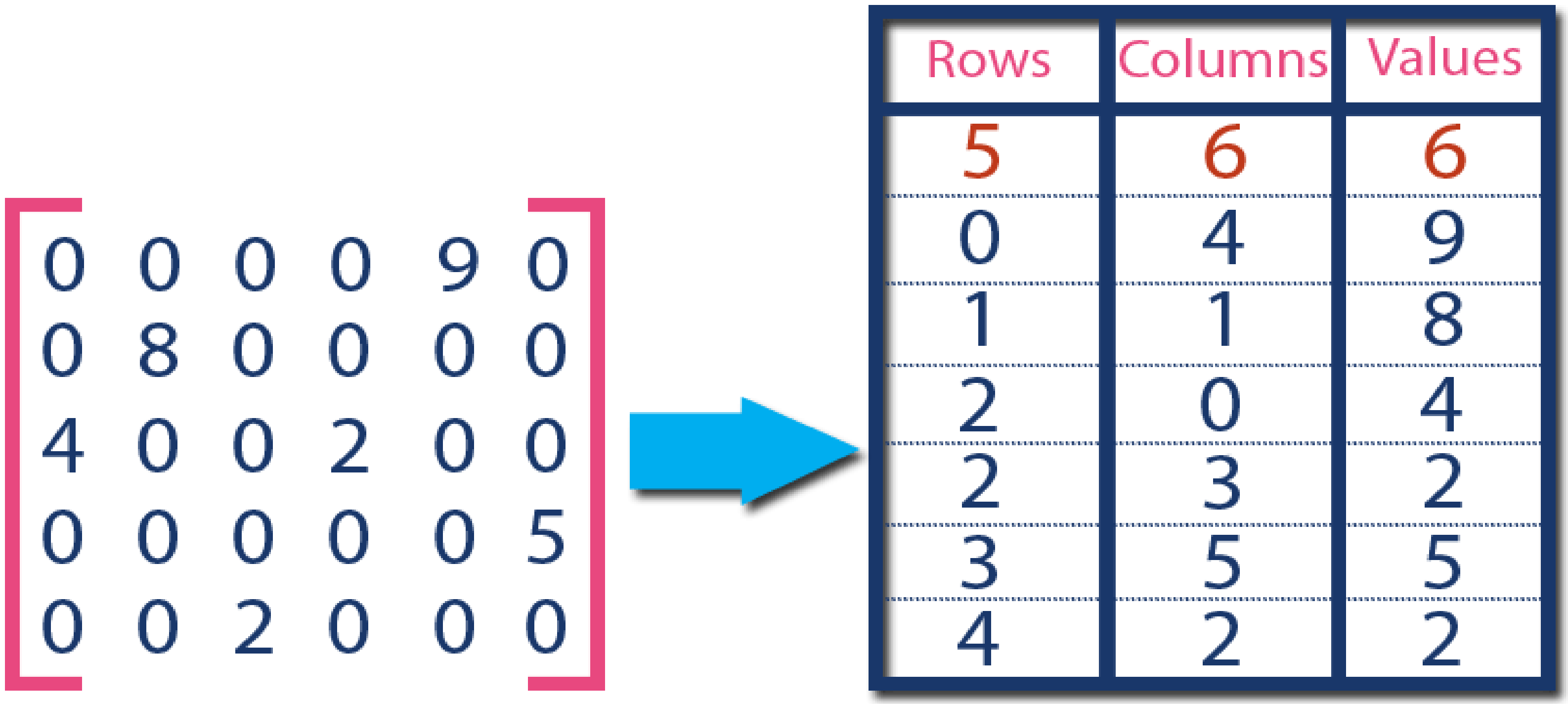
  1. Triplet Representation (Array Representation).
  2. Linked Representation

Triplet Representation (Array Representation)

- In this representation, we consider only non-zero values along with their row and column index values.
- In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse matrix.

Triplet Representation (Array Representation)

- For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

Linked Representation

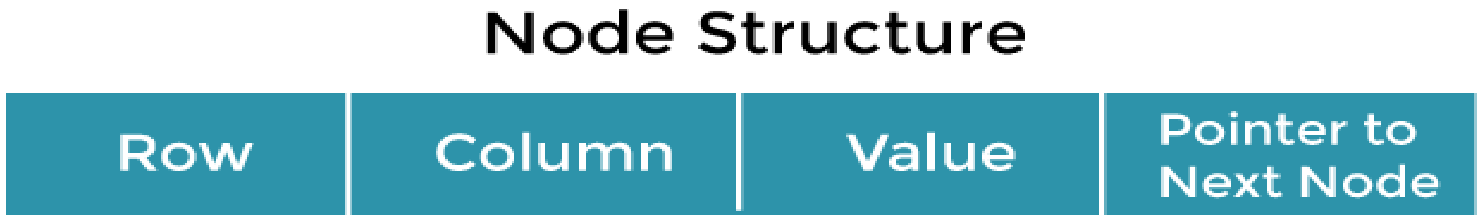
- In a linked list representation, the linked list data structure is used to represent the sparse matrix.
- Unlike the array representation, a node in the linked list representation consists of four fields. The four fields of the linked list are given as follows -

1. Row - It represents the index of the row where the non-zero element is located.
2. Column - It represents the index of the column where the non-zero element is located.
3. Value - It is the value of the non-zero element that is located at the index (row, column).

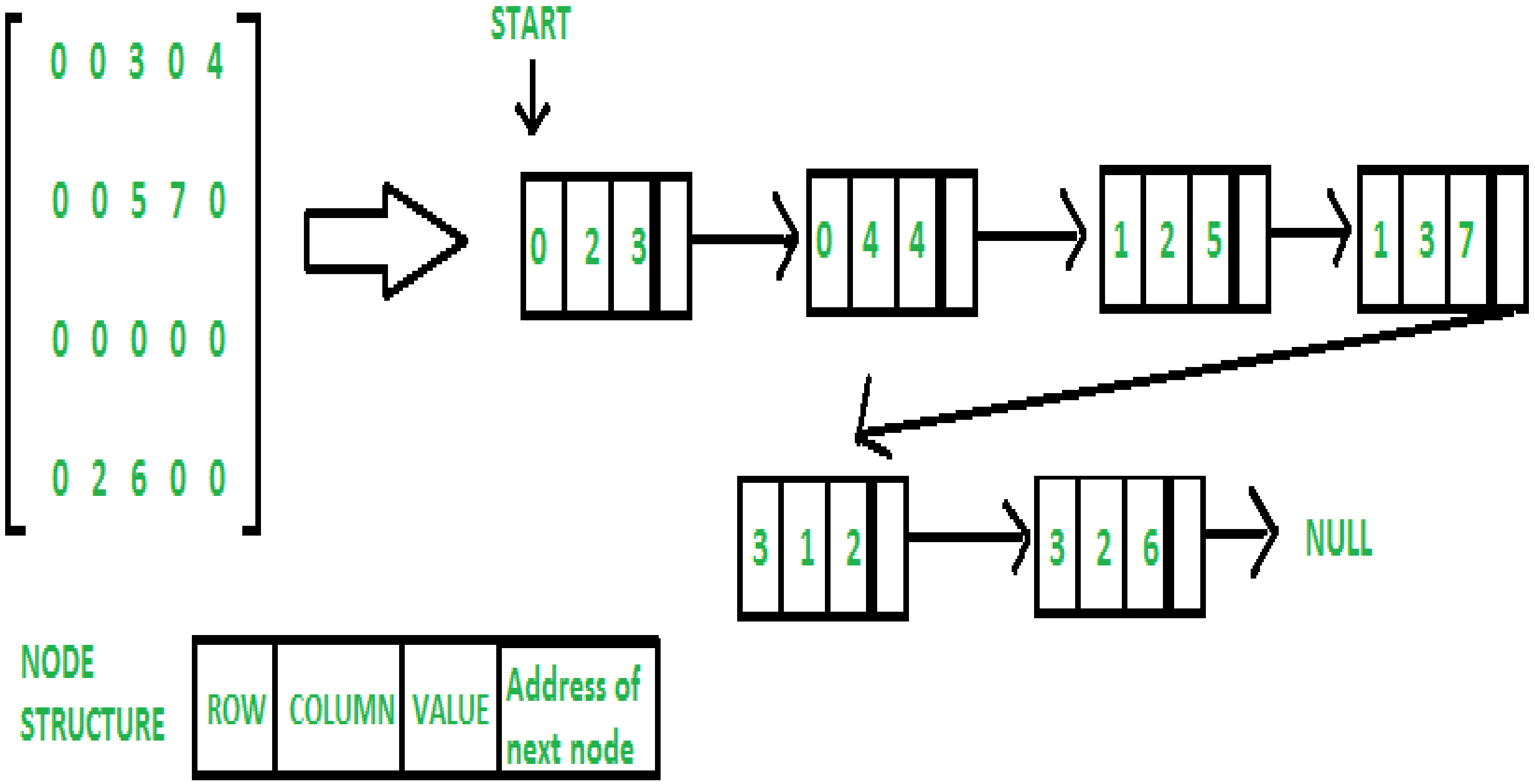
Linked Representation

4. Next node - It stores the address of the next node.

- The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.
- The node structure of the linked list representation of the sparse matrix is shown in the below image -



Linked Representation



THANK YOU

Stack

Topics

- Applications of Stacks
- Polish Expression
- Reverse Polish Expression And Their Compilation
- Recursion
- Tower of Hanoi

Polish Notation

- Polish Notation in the data structure is a method of expressing mathematical, logical, and algebraic equations universally.
- This notation is used by the compiler to evaluate mathematical equations based on their order of operations.

Polish Notation

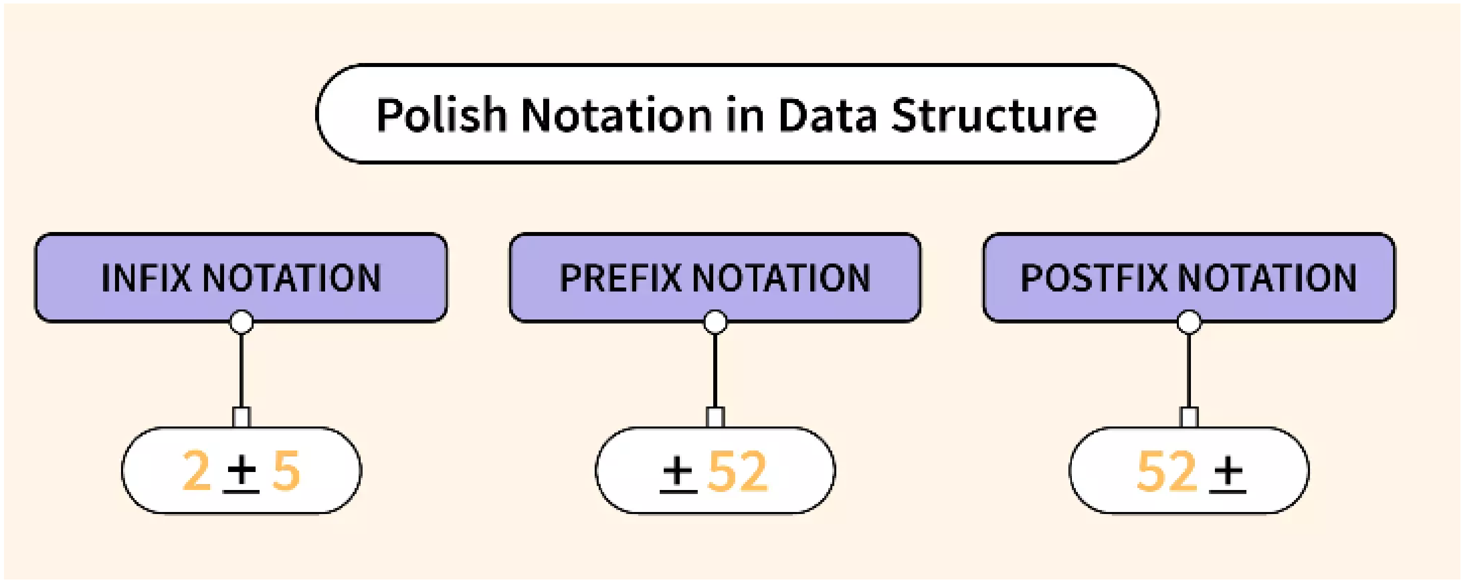
- This type of notation was introduced by the **Polish mathematician Lukasiewicz**.
- Polish Notation in data structure tells us about different ways to write an arithmetic expression.
- An arithmetic expression contains 2 things, i.e., operands and operators.

Operands and Operator

- **Operands** are either numbers or variables that can be replaced by numbers to evaluate the expressions.
- **Operators** are symbols symbolizing the operation to be performed between operands present in the expression.

Polish Notation

- When parsing mathematical expressions, three types of notations are commonly used : **Infix Notation, Prefix Notation, and Postfix Notation.**



Types of Notations: Infix

- This polish notation in data structure states that the operator is written in between the operands.

```
(3+7)
(1*(2+3))
```

- It is the most common type of notation we generally use to represent expressions. It's the fully parenthesized notation.

Types of Notations: Infix

- It much easier to write mathematical expressions in Infix Notation.
- But it is difficult to parse expressions on computers in the form of infix Polish Notation.

Types of Notations: Prefix

- This polish notation in data structure states that the operator should be present as a prefix or before the operands.
- This notation is also known as "**Polish Notation**".

3+7 will convert into +37
1*(2+3) will convert into *1(+23)

Types of Notations: Postfix

- This notation states that the operator should be present as a suffix, postfix, or after the operands.
- It is also known as **Suffix notation or Reverse Polish Notation**.

3+7 will convert into 37+

1*(2+3) will convert into (23+)1*

Types of Notations: Postfix

- In general, a computer can easily understand postfix expressions.
- This notation is universally accepted and is preferred for designing and programming the arithmetic and logical units of a CPU (Central Processing Unit).
- All the expressions that are entered into a computer are converted into Postfix or Reverse Polish Notation, stored in a stack, and then computed.

Operator Precedence and Associativity

- To convert each expression from infix to postfix, we assign priority to each of the operators present in the expression.

Operator	Symbols	Associativity
Parenthesis	{ }, ( ), [ ]	Left to right
Exponential notation	^	Right to Left
Multiplication and Division	*,/	Left to right
Addition and Subtraction	+,-	Left to right

Conversion of an Infix to Postfix

- Generally, humans find infix polish notation much easier to understand than postfix or reverse polish notation.
- To convert each expression from infix to postfix, we assign priority to each of the operators present in the expression.
- Two ways to convert infix to postfix:

1. Without using stack (Multiple scan).
2. Using stack (Single scan).

Rules for Infix to postfix using stack

1. Scan Expression from Left to Right.
2. Print OPERANDs as they arrive.
3. If OPERATOR arrives & Stack is empty, push this operator onto the stack.
4. IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, push it on stack.
5. IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and print the TOP. Then test the incoming operator against the NEW TOP of stack.

Rules for Infix to postfix using stack

6. IF incoming OPERATOR has EQUAL precedence with TOP of Stack, use ASSOCIATIVITY Rules.
7. For ASSOCIATIVITY of LEFT to RIGHT –

   POP and print the TOP of stack, then push the incoming OPERATOR.

   Then test the incoming OPERATOR against the NEW TOP of stack.
8. For ASSOCIATIVITY of RIGHT to LEFT –

   PUSH incoming OPERATOR on stack.

Rules for Infix to postfix using stack

9. At the end of Expression, POP & print all OPERATORS from the stack.
10. IF incoming SYMBOL is ‘(‘ PUSH it onto Stack.
11. IF incoming SYMBOL is ‘)’ POP the stack and print OPERATORs till ‘(‘ is found. POP that ‘(‘.
12. IF TOP of stack is ‘(‘ PUSH OPERATOR on Stack.

Pseudocode for Infix to postfix

```
FUNCTION InfixToPostfix (stack,infix)
string  postfix.
LOOP  i = 0 to i < infix.length
1.   IF infix[i] -> OPERAND then
         postfix+=infix[i]
2.   ELSE IF  infix[i] -> '(' then PUSH to stack
3.   ELSE IF infix[i] ->  ')' then POP&PRINT stack
          till stack gets empty OR '(' is found. POP
         that"(".
```

Pseudocode for Infix to postfix

```
4. ELSE IF infix[i] -> OPERATOR (+,-,*,/,^) then
4.1.  IF stack-> EMPTY then PUSH infix[i] on stack
4.2.  ELSE IF stack -> NOTEMPTY
4.2.1 IF precedence(infix[i]) > precedence(stack.top)
            then -> PUSH infix[i] on Stack
4.2.2 ELSE  IF  (precendence(infix[i])  ==
          precendence (stack.top))
            IF (infix[i] == ‘^’)  then
               -> PUSH infix[i] on Stack
            ELSE
```

Pseudocode for Infix to postfix

```
        --> POP & PRINT TOP of Stack till this
        condition is true.
        --> PUSH infix[i] on Stack
4.2.3  ELSE
4.2.3.1  WHILE stack  -> (NOT EMPTY)  &&
precendence (infix[i] <= precendence (stack.top))
then POP and PRINT stack
4.2.3.2 PUSH current OPERATOR on Stack
END LOOP
POP & PRINT remaining O·PERATORS in  the
stack
```

Rules for Infix to prefix using stack

1. Reverse infix expression.
2. Scan Expression from Left to Right.
3. Print OPERANDs as the arrive.
4. If OPERATOR arrives & Stack is empty, PUSH to stack.
5. IF incoming OPERATOR has HIGHER precedence than the TOP of the Stack, PUSH it on stack.
6. IF incoming OPERATOR has LOWER precedence than the TOP of the Stack, then POP and PRINT the TOP. Then test the incoming OPERATOR against the NEW TOP of stack.

Rules for Infix to prefix using stack

7. IF incoming OPERATOR has EQUAL precedence with TOP of Stack, use ASSOCIATIVITY Rules.
8. For ASSOCIATIVITY of LEFT to RIGHT –

   PUSH incoming OPERATOR on stack.
9. For ASSOCIATIVITY of RIGHT to LEFT –

   POP and print the TOP of stack, then push the incoming OPERATOR.

   Then test the incoming OPERATOR against the NEW TOP of stack.

Rules for Infix to prefix using stack

10. At the end of Expression, POP & print all OPERATORS from the stack.
11. IF incoming SYMBOL is ‘)‘ PUSH it onto Stack.
12. IF incoming SYMBOL is ‘(’ POP the stack & PRINT OPERATORs till ‘)‘ is found or Stack Empty. POP out that ‘)‘ from stack.
13. IF TOP of stack is ‘)‘ PUSH OPERATOR on Stack.
14. At the end Reverse output string again.

Pseudocode for Infix to prefix

```
FUNCTION InfixToPrefix (stack,infix)
infix = reverse(infix).
LOOP  i=0 to i<infix.length
   IF infix[i] is OPERAND--> prefix+=infix[i]
   ELSE IF infix[i] is ‘)‘ --> Stack.push(infix[i])
   ELSE IF infix[i] is ’(’ --> POP & PRINT Stack
   values till '('is found & stack NOT EMPTY. POP
   that’(‘
   ELSE IF infix[i] IS A OPERATOR(+,-,*,/,^) -->
      IF Stack IS EMPTY--> PUSH OPERATOR on
      stack
```

Pseudocode for Infix to prefix

```
ELSE -->
IF precedence(infix[i])>precedence(stack.top)
    --> PUSH infix[i] on Stack
ELSE   IF  precedence(infix[i]) == precedence
(stack.top)
        IF   infix[i]==‘^’
            --> POP & PRINT TOP of Stack till
            this  condition is true.
            --> PUSH infix[i] on Stack
        ELSE
            --> PUSH infix[i] on Stack
```

Pseudocode for Infix to prefix

```
    ELSE IF precedence (infix[i]) <precedence (stack.top)
        --> POP & PRINT till Stack NOT EMPTY && precedence(infix[i]) < precedence (stack.top)
        --> PUSH infix[i] onto Stack
END LOOP

POP & PRINT Remaining Elements of Stack
prefix= reverse(prefix)& RETURN
```

Evaluation of Postfix Expression

```
Begin
   for each character in postfix expr, do
     if operand is encountered,
        push it onto stack
    else if operator is encountered ,
        pop 2 elements
        A <- Top element
        B <- Next to top element
```

Evaluation of Postfix Expression

```
        result = B operator A
        push result onto stack
    return element of stack top
End
```

Evaluation of Prefix Expression

```
Begin
Scan prefix expression from Right to Left
   for each character in prefix expr, do
     if operand is encountered,
        push it onto stack
    else if operator is encountered ,
        pop 2 elements
        A <- Top element
        B <- Next to top element
```

Evaluation of Prefix Expression

```
        result = A operator B
        push result onto stack
    return element of stack top
End
```

Rules for postfix to infix using stack

1. Scan POSTFIX expression from LEFT to RIGHT.
2. IF the incoming symbol is a OPERAND, PUSH it onto the Stack.
3. IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR in between the 2 OPERANDs, ADD '(' & ')' to the whole expression & PUSH this whole new expression string back into the Stack.
4. At the end POP and PRINT the full INFIX expression from the Stack.

Pseudocode for Postfix to Infix

```
Function PostfixToInfix (string postfix)
1.  stack s
2.  LOOP: i=0 to  postfix.length
        2.1.  IF postfix[i] is OPERAND ->
            2.1.1   s.push (postfix[i])
        2.2.  ELSE IF postfix[i] is OPERATOR ->
            2.2.1   op1 = s.top()
            2.2.2   s.pop()
            2.2.3   op2 = s.top()
            2.2.4   s.pop()
```

Pseudocode for Postfix to Infix

```
        2.2.5   exp = ‘(' + op2+ postfix[i] + op1 + ‘)’
        2.2.6   s.push (exp)
END LOOP
RETURN s.top
```

Rules for prefix to infix using stack

1. Scan PREFIX expression from RIGHT to LEFT OR REVERSE the PREFIX expression and scan it from LEFT to RIGHT.
2. IF the incoming symbol is a OPERAND, PUSH it onto the Stack
3. IF the incoming symbol is a OPERATOR, POP 2 OPERANDs from the Stack, ADD this incoming OPERATOR in between the 2 OPERANDs, ADD ‘(‘ & ‘)’ to the whole expression & PUSH this whole new expression string back into the Stack.
4. At the end POP and PRINT the full INFIX expression from the Stack.

Pseudocode for Prefix to Infix

```
Function PrefixToInfix (string prefix)
1.  stack s
2.  LOOP: i = prefix.length-1 to 0
        2.1.  IF prefix[i] is OPERAND ->
            2.1.1   s.push (prefix[i])
        2.2.  ELSE IF prefix[i] is OPERATOR ->
            2.2.1   op1 = s.top()
            2.2.2   s.pop()
            2.2.3   op2 = s.top()
            2.2.4   s.pop()
```

Pseudocode for Prefix to Infix

```
        2.2.5   exp = ‘(' + op1+ postfix[i] + op2 + ‘)’
        2.2.6   s.push (exp)
END LOOP
RETURN s.top
```

THANK YOU

Queue

Topics

- Representation Of Queue
- Operations On Queue
- Circular Queue
- Priority Queue
- Array representation of Priority Queue
- Double Ended Queue
- Applications of Queue

Difference between Stack & Queue

Stack	Queue
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue.
Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion.	Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion.
Stack is called as Last In First Out (LIFO) List.	Queue is called as First In First Out (FIFO) List.
The most and least accessible elements are called TOP of the stack	Insertion of element is performed at FRONT end and deletion is performed from REAR end.
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store

Difference between Stack & Queue

Stack	Queue
Insertion operation is referred as PUSH and deletion operation is referred as POP.	Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE.
Function calling in any languages uses Stack.	Task Scheduling by Operating System uses queue.

Linear Queue

- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) of FCFS (first come first served) pattern.
- **Front** is the end of queue from that deletion is to be performed.
- **Rear** is the end of queue at which new element is to be inserted.

Linear Queue

- The process to add an element into queue is called **Enqueue.**
- The process of removal of an element from queue is called **Dequeue**.
- The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checkedout.

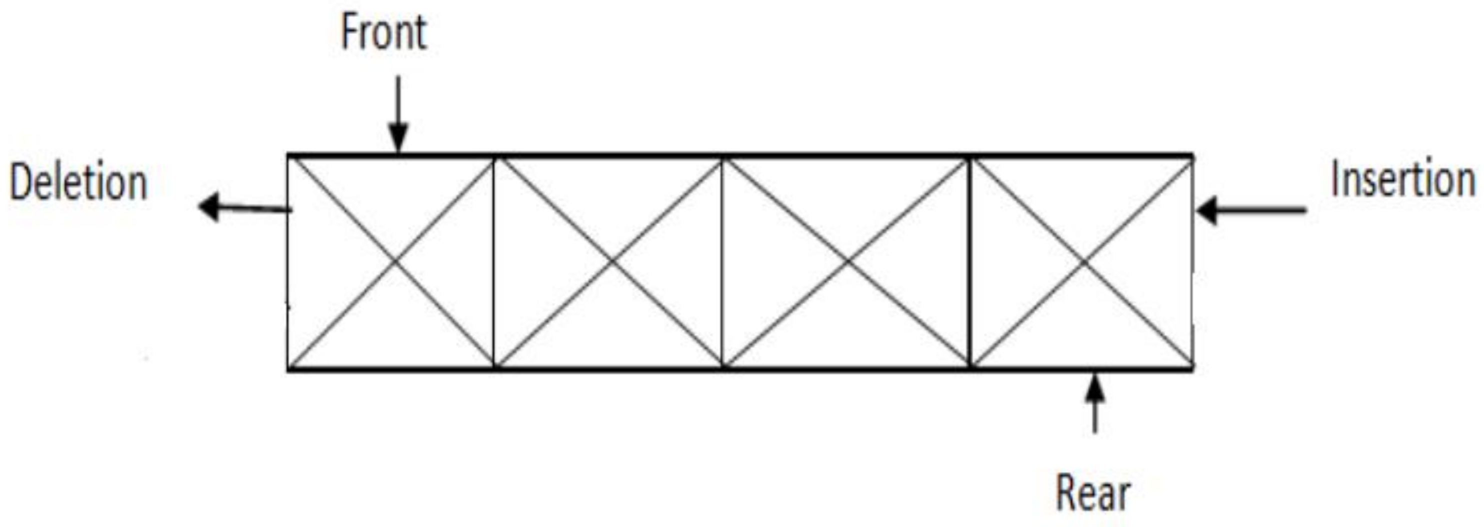
Array Representation Of Linear Queue

- Like stacks, Queues can also be represented in an array.
- In this representation, the Queue is implemented using the array. Variables used in this case are:-

1. Queue: the name of the array storing queue elements.
2. Rear: the index where the first element is stored in the array representing the queue.

Array Representation Of Linear Queue

3. Front: the index where the last element is stored in an array representing the queue.



Drawback of Linear Queue

- The major drawback of using a linear Queue is that insertion is done only from the rear end.
- If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.
- In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Operations on Queue

- The fundamental operations that can be performed on queue are listed as follows -

1. **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
2. **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.

Operations on Queue

3. **Peek/front:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
4. **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
5. **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Insertion in Queue

Procedure: **QINSERT(Q, F, R, N,Y)**

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This procedure inserts Y at rear end of Queue.

```
1. [Overflow]
   IF R>=N-1
   Then write ('OVERFLOW')
   Return
2. [Increment REAR pointer]
   R<-R+1
```

Insertion in Queue

```
3. [Insert element]
   Q[R]<-Y
4. [Is front pointer properly set]
   IF F==-1
   Then F<-0
   Return
```

Deletion from Queue

Function: **QDELETE(Q, F, R)**

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This function deletes an element from front end of the Queue.

```
1. [Underflow]
   IF F==-1
   Then write ('UNDERFLOW')
   Return(0) (0 denotes an empty Queue)
2. [Delete element]
   Y<-Q[F]
```

Deletion from Queue

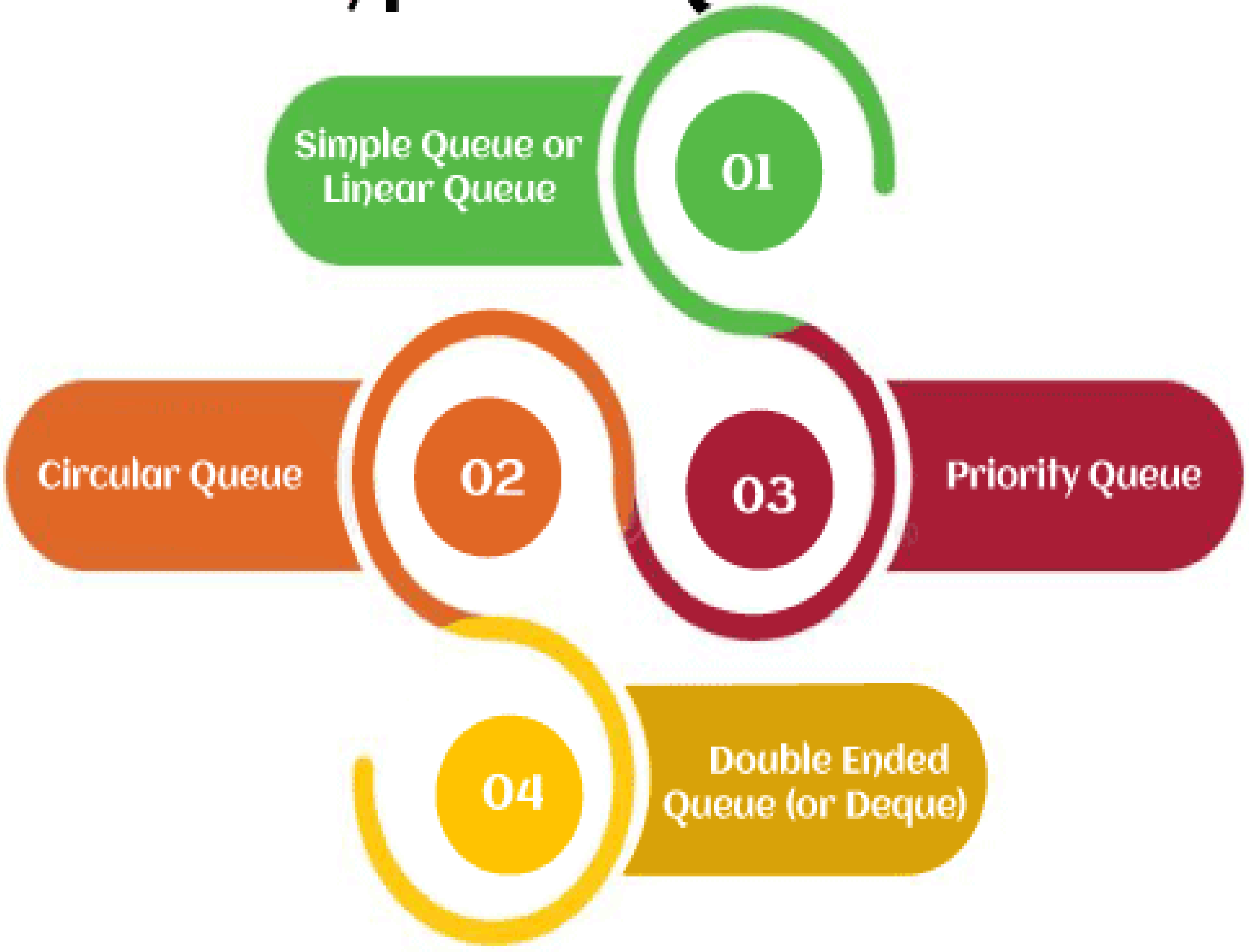
```
3. [Queue empty?]
   IF F=R
   Then F<-R<-0
   Else F<-F+1 (increment front pointer)
4. [Return element]
   Return(Y)
```

Ways to implement the Queue

- There are two ways of implementing the Queue:

1. Implementation using array: The sequential allocation in a Queue can be implemented using an array.
2. Implementation using Linked list: The linked list allocation in a Queue can be implemented using a linked list.

Types of Queues



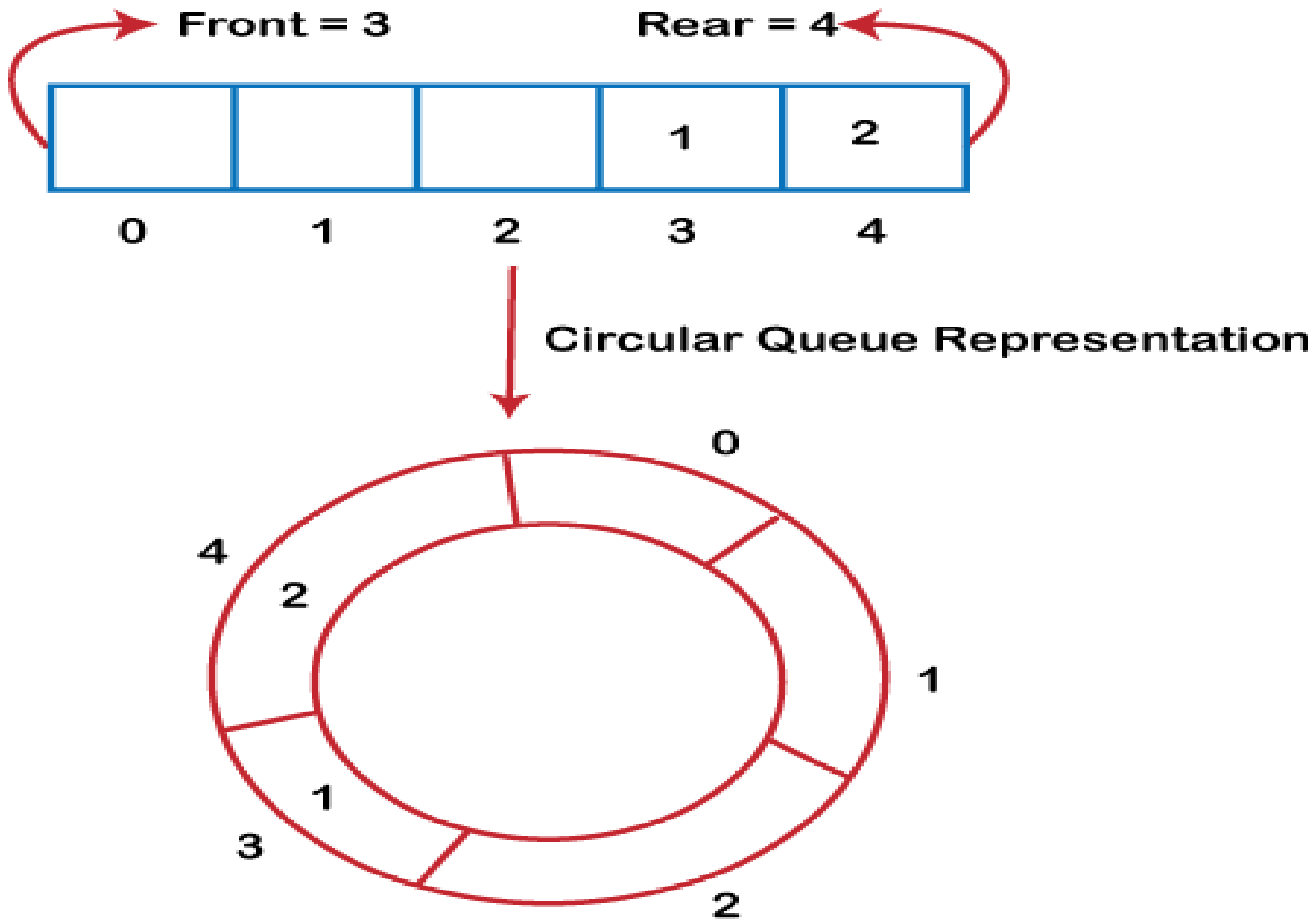
Circular Queue

- There was one limitation in the array implementation of Queue.
- If the rear reaches to the end position of the Queue, then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized.
- So, to overcome such limitations, the concept of the circular queue was introduced.
- Circular queue is a linear data structure. It follows FIFO principle.

Circular Queue

- In circular queue the last node is connected back to the first node to make a circle.
- Elements are added at the rear end and the elements are deleted at front end of the queue.
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.

Circular Queue



Insertion in Circular Queue

Procedure: **CQINSERT (F, R, Q, N, Y)**

- Given F and R pointers to the front and rear elements of a circular queue, respectively. Circular queue Q consisting of N elements. This procedure inserts Y at rear end of Circular queue.

```
1. [Reset Rear Pointer]
   IF R==N-1
   Then R<-0
   Else R<-R+1
```

Insertion in Circular Queue

```
2. [Overflow]
   If F==R
   Then Write('OVERFLOW')
   Return
3. [Insert element]
   Q[R]<-Y
4. [Is front pointer properly set?]
   If F==-1
   Then F<-0
   Return
```

Deletion from Circular Queue

Function: **CQDELETE (F, R, Q, N)**

- Given F and R pointers to the front and rear elements of a Circular queue respectively. Circular Queue Q consisting of N elements. This function deletes an element from front end of the Circular Queue. Y is temporary pointer variable.

```
1. [Underflow?]
   If F==-1
   Then Write('UNDERFLOW')
   Return(0)
```

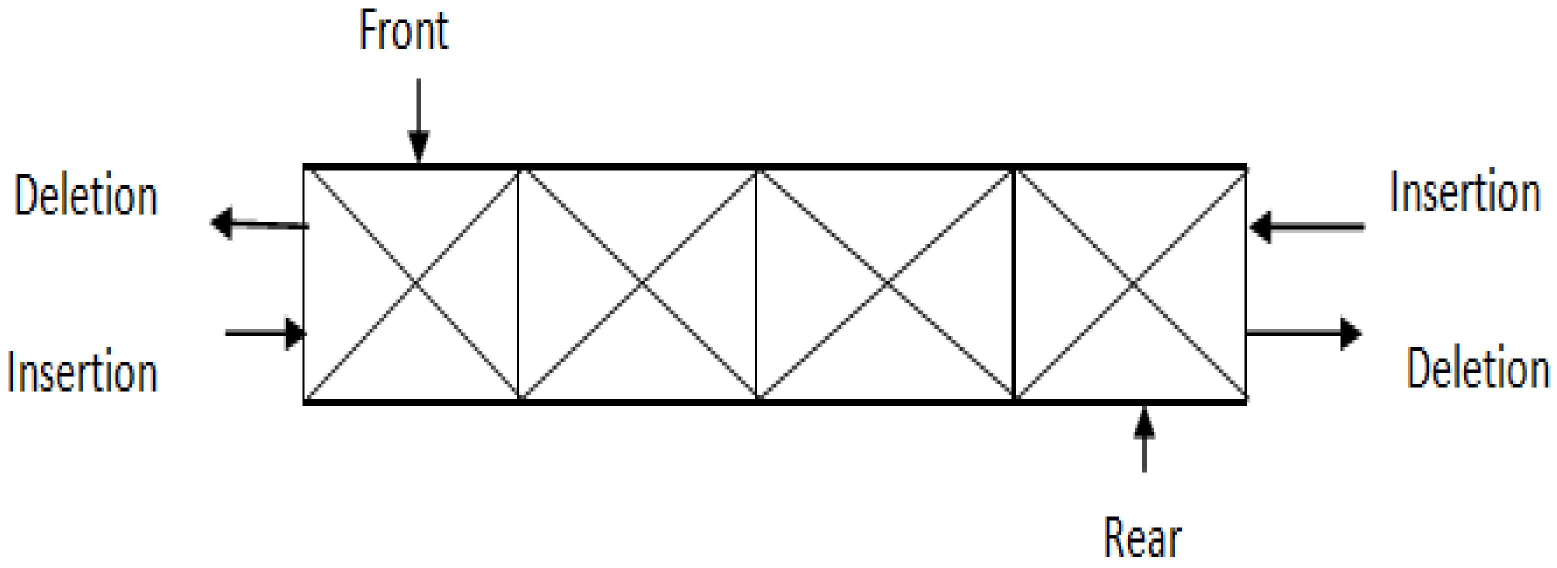
Deletion from Circular Queue

```
2. [Delete element]
    Y <- Q[F]
3. [Queue Empty?]
    If F = =R
    Then F <- R <- -1
4. [Increment front pointer]
    If F = =N-1
   Then F <- 0
   Else F <- F+1
5. Return(Y)
```

Double Ended Queue(DeQue)

- The dequeue stands for Double Ended Queue.
- Deque is a linear data structure where the insertion and deletion operations are performed from both ends.
- We can say that deque is a generalized version of the queue.
- Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule.

Double Ended Queue (DeQueue)



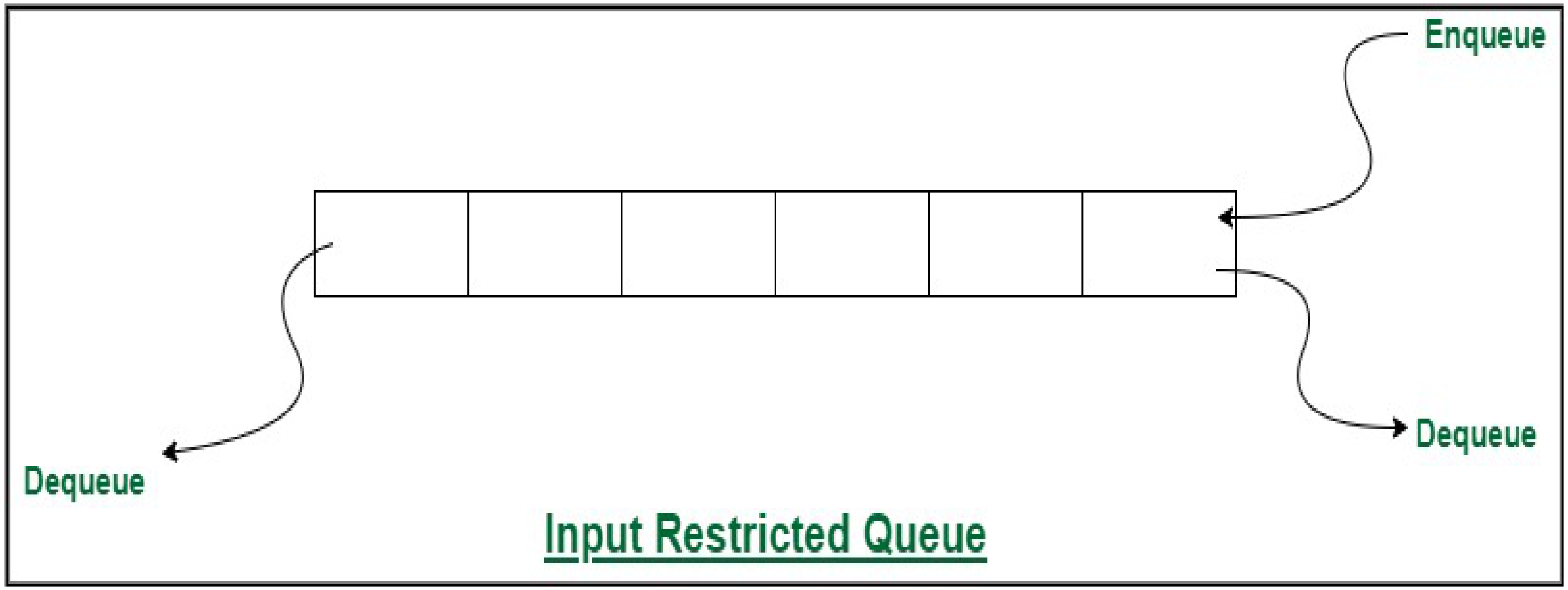
Types of DeQue

- There are two types of deque-

1. Input restricted queue.
2. Output restricted queue.

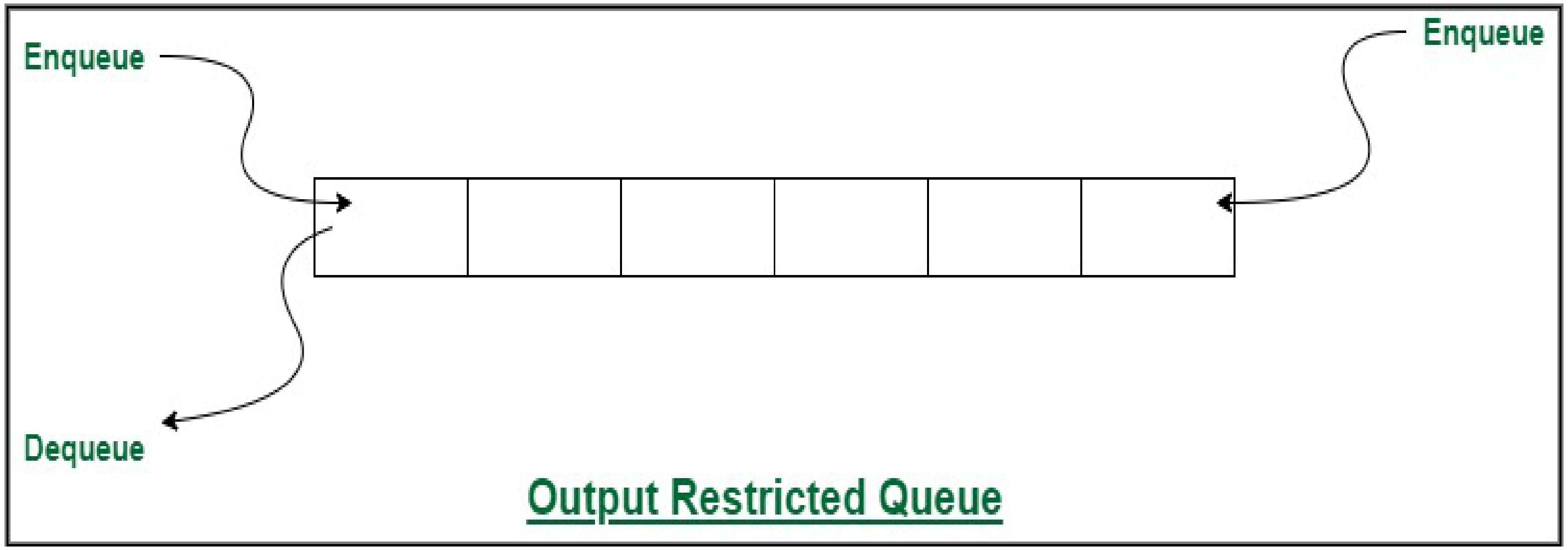
Input restricted DeQue

- In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

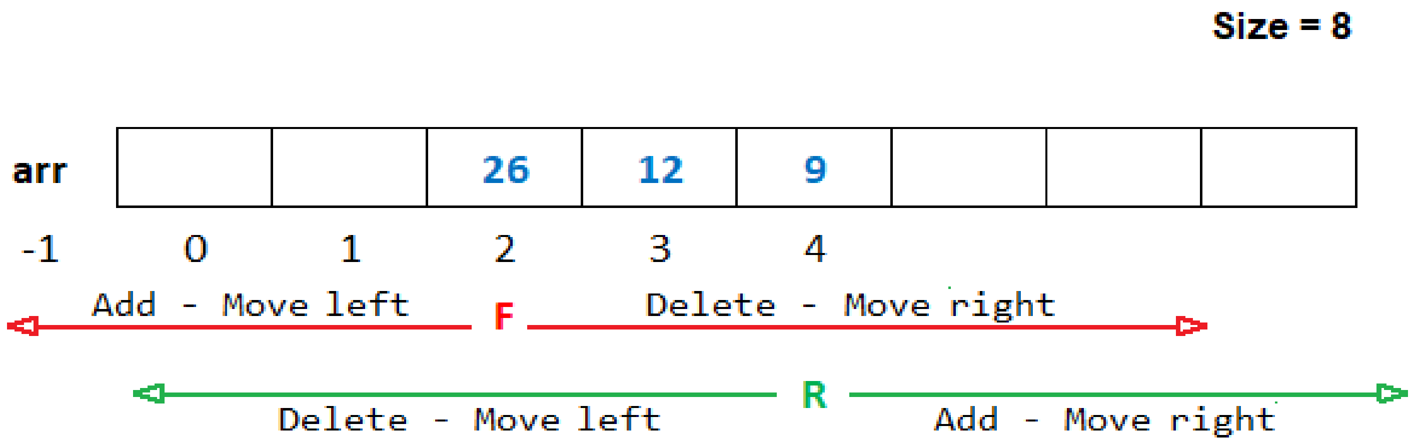


Output restricted DeQue

- In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Operations performed on DeQue



- From the above image of the deque, we can see that when we add an element from the rear end, the R moves towards the right and,

Operations performed on DeQue

- when we delete an element from the rear end, the R moves towards the left.
- Similarly, when we delete an element from the front end, the F moves towards the right and when we add an element from the front end, the F moves towards the left.

Operations performed on DeQue

1. Insertion at front
2. Insertion at rear
3. Deletion at front
4. Deletion at rear

Insertion in DeQue

Procedure: **DQINSERT_FRONT (Q, F, R, N,Y)**

- Given F and R pointers to the front and rear elements of a queue, a queue consisting of N elements and an element Y, this procedure inserts Y at the front of the queue.

```
1. [Overflow]
        IF F==0
        Then write ('OVERFLOW')
        Return
```

Insertion in DeQue

```
2. [Decrement front pointer]
   F <- F-1
3. [Is front pointer properly set]
   IF F==-1 && R==-1
   Then F<-R<-0
4. [Insert element ]
   Q[F] <-Y
```

Deletion from DeQue

Function: **DQDELETE_REAR (Q, F, R)**

- Given F and R pointers to the front and rear elements of a queue. And a queue Q to which they correspond, this function deletes and returns the last element from the front end of a queue. And Y is temporary variable.

```
1. [Underflow]
    IF R= -1
   Then write ('UNDERFLOW')
   Return(0)
```

Deletion from DeQue

```
2. [Delete element]
    Y <- Q[R]
3. [Queue empty?]
    IF R=F
    Then R <- F <- -1
    Else R <- R-1 (decrement front pointer)
4. [Return element]
    Return (Y)
```

Priority Queue

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is often referred as priority queue.
- The priority of the elements in a priority queue determines the order in which elements are served (i.e., the order in which they are removed).
- If in any case the elements have same priority, they are served as per their ordering in the queue.

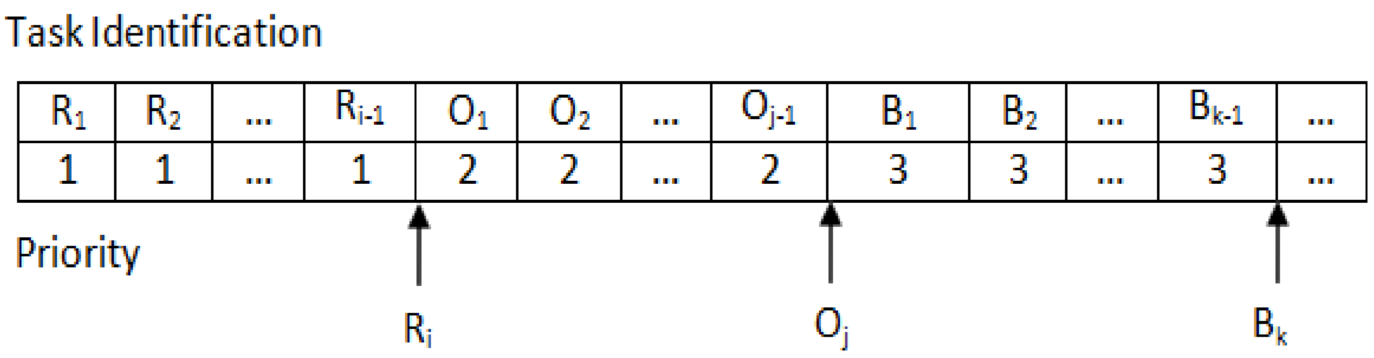
Properties of Priority Queue

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Example

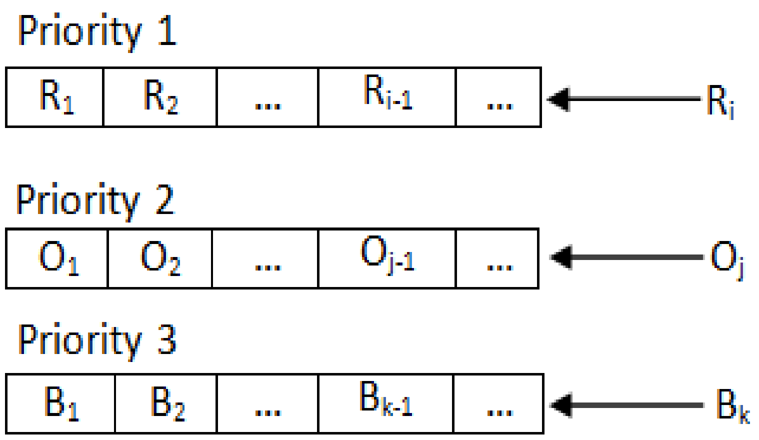
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively.
- Therefore if a job is initiated with priority i, it is inserted immediately at the end of list of other jobs with priorities i.
- Here jobs are always removed from the front of queue.

Example



Task Identification
R_1 R_2 ... R_{i-1} O_1 O_2 ... O_{j-1} B_1 B_2 ... B_{k-1} ...
1 1 ... 1 2 2 ... 2 3 3 ... 3 ...
Priority
R_i
O_j
B_k

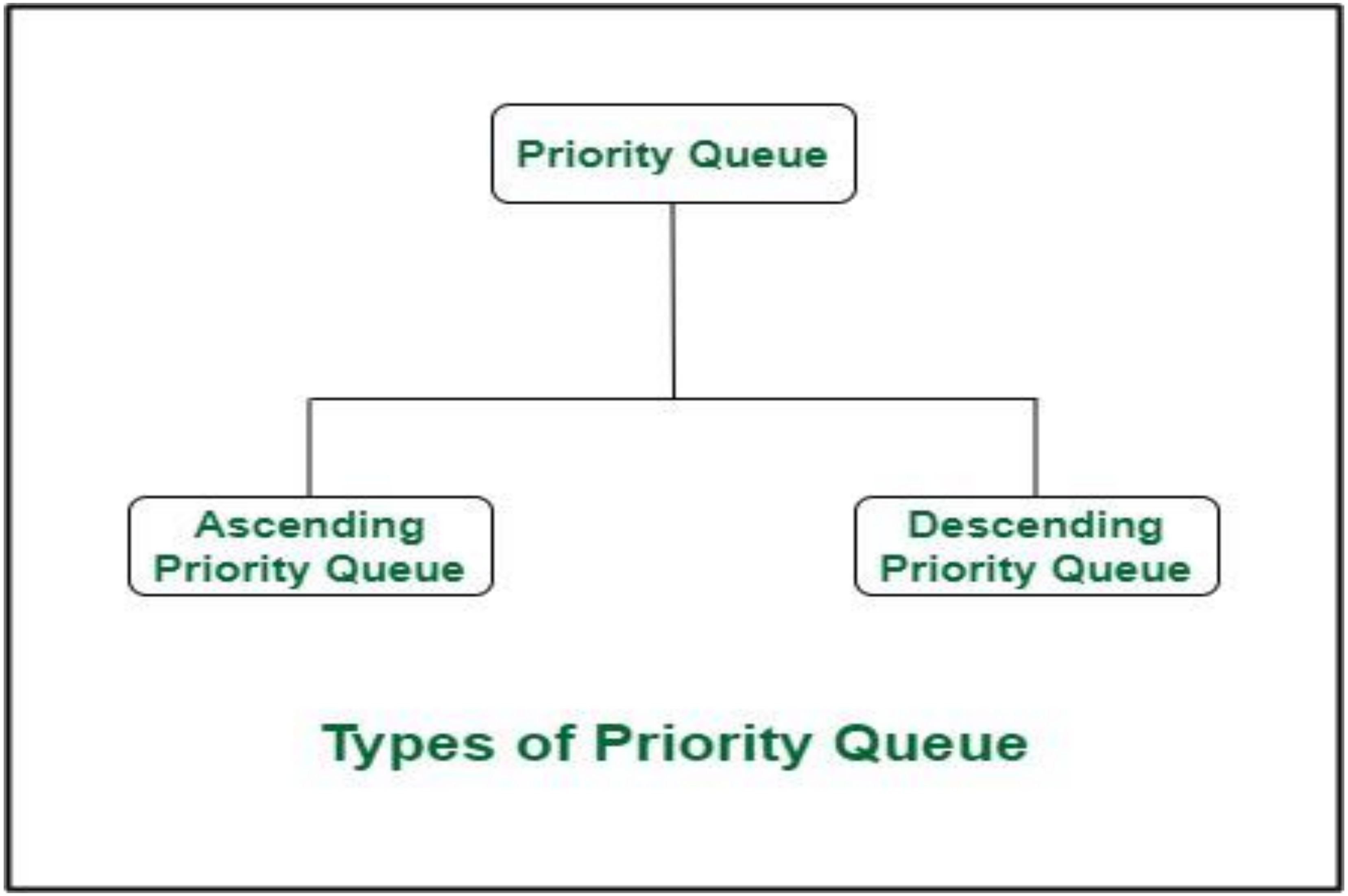
Fig (a) : Priority Queue viewed as a single queue with insertion allowed at any position.



Priority 1
R_1 R_2 ... R_{i-1} ...
R_i
Priority 2
O_1 O_2 ... O_{j-1} ...
O_j
Priority 3
B_1 B_2 ... B_{k-1} ...
B_k

Fig (b) : Priority Queue viewed as a Viewed as a set of queue

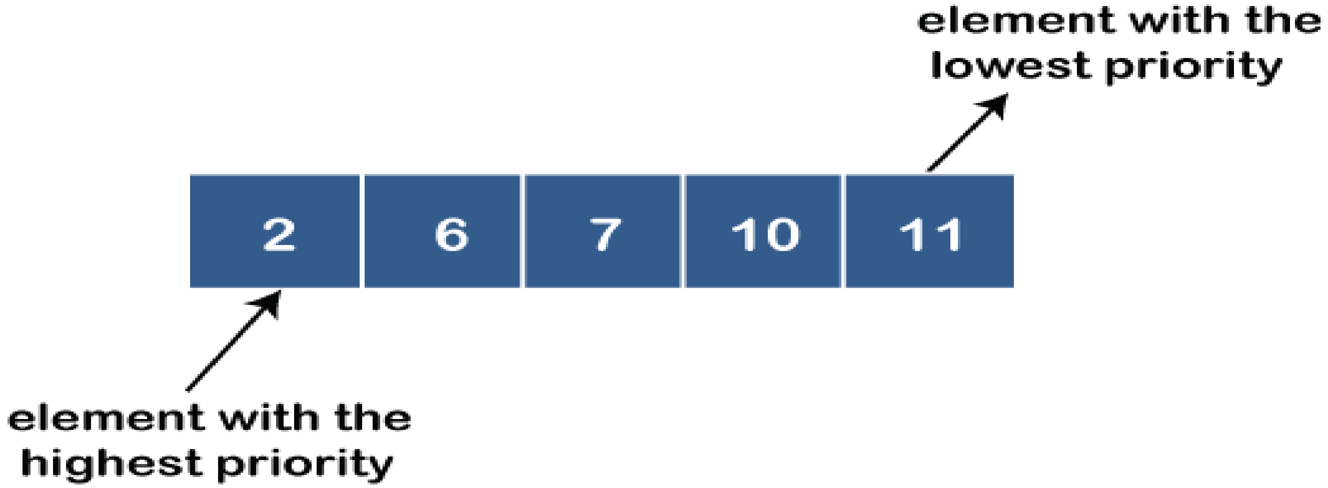
Types of Priority Queue



Types of Priority Queue

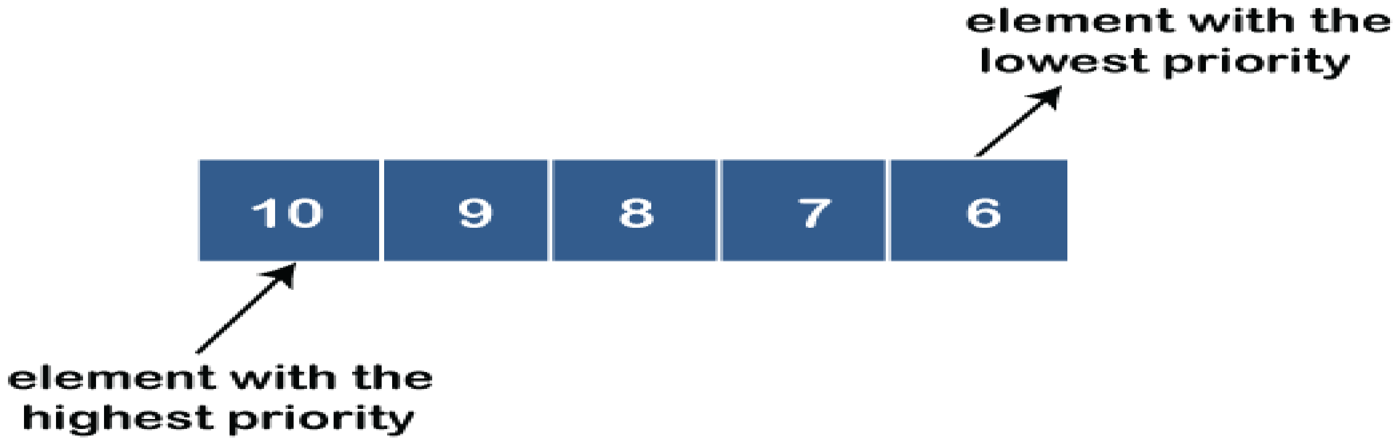
Ascending order Priority Queue

- In ascending order priority queue, a lower priority number is given as a higher priority in a priority.
- For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

Descending order Priority Queue

- In this type of priority queue, a higher priority number is given as a higher priority in a priority.
- For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

Implementation of Priority Queue

- Priority queue can be implemented using the following data structures:

1. Arrays
2. Linked list
3. Heap data structure
4. Binary search tree

Priority Queue using Array

- A simple implementation is to use an array of the following structure.

```
struct item {
        int item;
        int priority;
}
```

- enqueue(): This function is used to insert new data into the queue.
- dequeue(): This function removes the element with the highest priority from the queue.

Applications of Queue

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- Managing requests on a single shared resource such as CPU scheduling and disk scheduling.
- Handling hardware or real-time systems interrupts.
- Handling website traffic.
- Routers and switches in networking.
- Maintaining the playlist in media players.

THANK YOU

Sorting

Outline

- Sorting
- Bubble Sort
- Selection Sort
- Quick Sort

Sorting

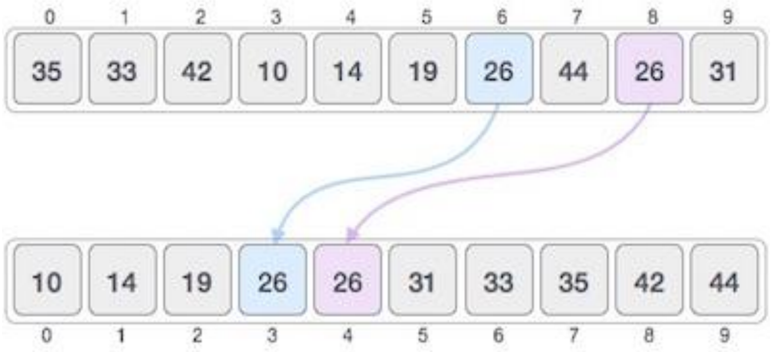
- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- For example, consider an array A = {A1, A2, A3, A4, ….., An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > …..> An.
- Sorting algorithm specifies the way to arrange data in a particular order.

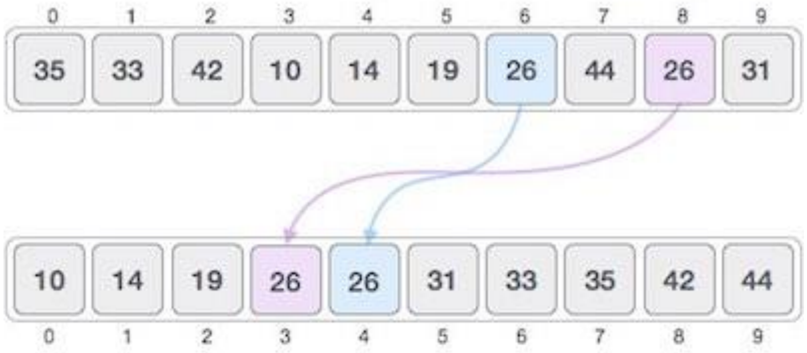
In-place Sorting and Not-in-place Sorting

- Sorting algorithms may require some extra space for comparison and temporary storage of few data elements.
- These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**.
- Bubble sort is an example of in-place sorting.
- However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted.
- Sorting which uses equal or more space is called **not-in-place sorting.**
- Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**. Bubble and Merge sort.
- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting.** Quick sort.





Adaptive and Non-Adaptive Sorting Algorithm

- A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted.
- That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them. Ex. quick sort and bubble sort.
- A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness. ex. Selection sort and merge sort.

Bubble Sort

- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared, and the elements are swapped if they are not in order.
- It is called **bubble sort** because the movement of array elements is just like the movement of air bubbles in the water.

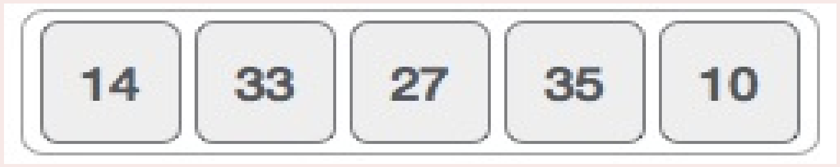
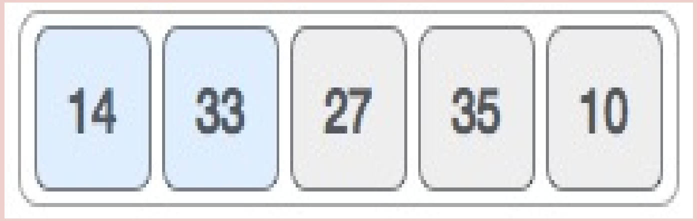
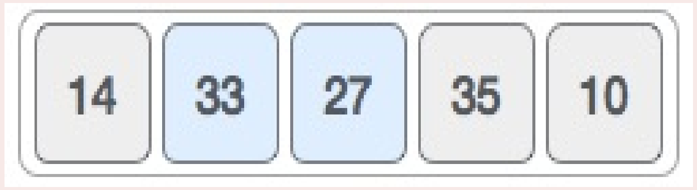
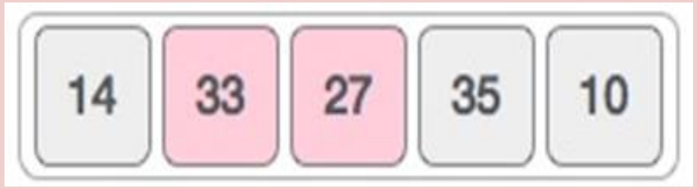
Bubble Sort

- Bubbles in water rise to the surface. similarly, the array elements in bubble sort move to the end in each iteration.
- It is not suitable for large data set as its average and worst-case time complexity is quite high (O(n2)).
- Best case time complexity is (O(n)).

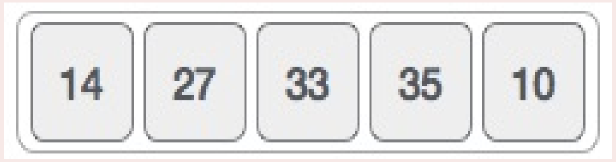
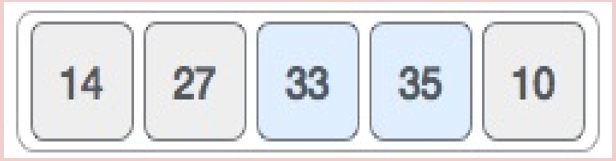
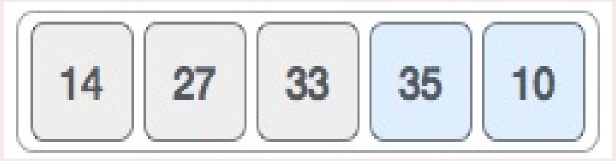
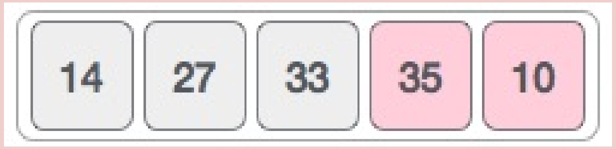
Algorithm

```
begin BubbleSort(arr)
   for all array elements
      if arr[i] > arr[i+1]
         swap(arr[i], arr[i+1])
      end if
   end for
   return arr
end BubbleSort
```

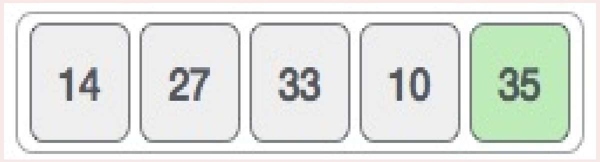
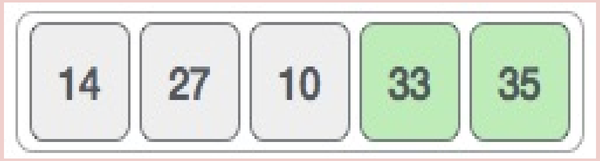
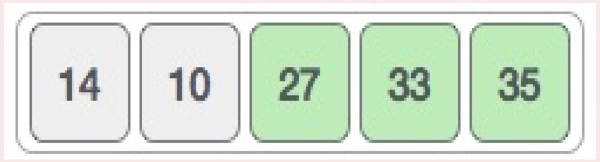
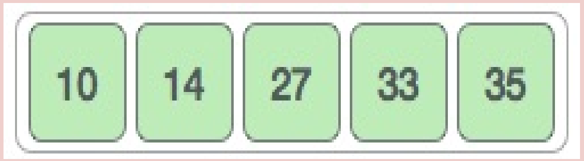
How Bubble Sort Works?

Unsorted Array=  14 33 27 35 10	
 14 33 27 35 10	Bubble sort starts with very first two elements, comparing them to check which one is greater.
 14 33 27 35 10	In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.
 14 33 27 35 10	We find that 27 is smaller than 33 and these two values must be swapped.

How Bubble Sort Works?

	The new array should look like this –
	Next we compare 33 and 35. We find that both are in already sorted positions.
	Then we move to the next two values, 35 and 10.
	We know then that 10 is smaller 35. Hence they are not sorted.

How Bubble Sort Works?

	We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −
	To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −
	Notice that after each iteration, at least one value moves at the end.
	And when there's no swap required, bubble sorts learns that an array is completely sorted.

Selection Sort

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

Selection Sort

- In selection sort, the first smallest element is selected from the unsorted array and placed at the first position.
- After that second smallest element is selected and placed in the second position.
- The process continues until the array is entirely sorted.
- The average and worst-case complexity of selection sort is O(n2).

Algorithm

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted

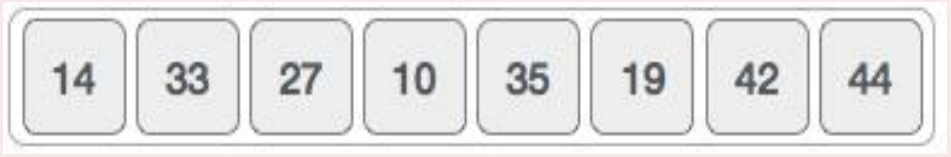
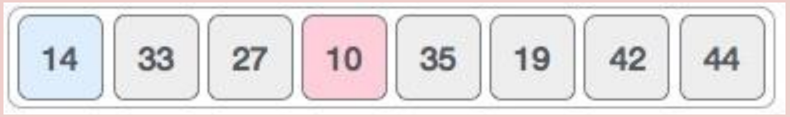
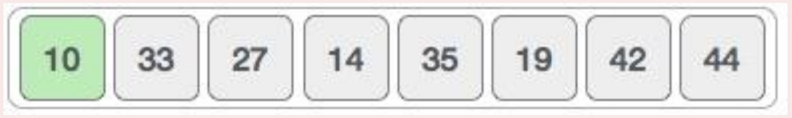
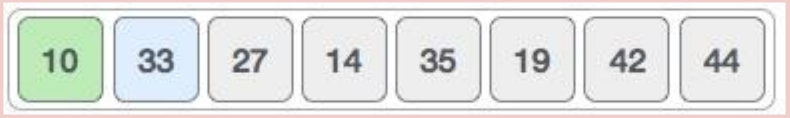
Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list
   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i
      /* check the element to be minimum */
      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for
```

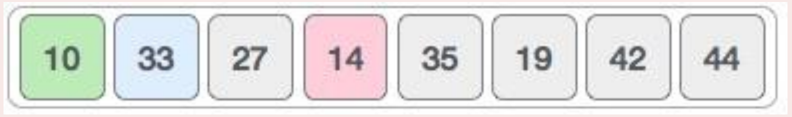
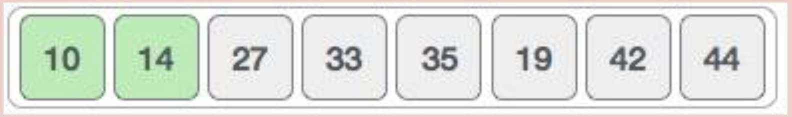
Pseudocode

```
/* swap the minimum element with the current element*/
      if indexMin != i then
         swap list[min] and list[i]
      end if
   end for
end procedure
```

How Selection Sort Works?

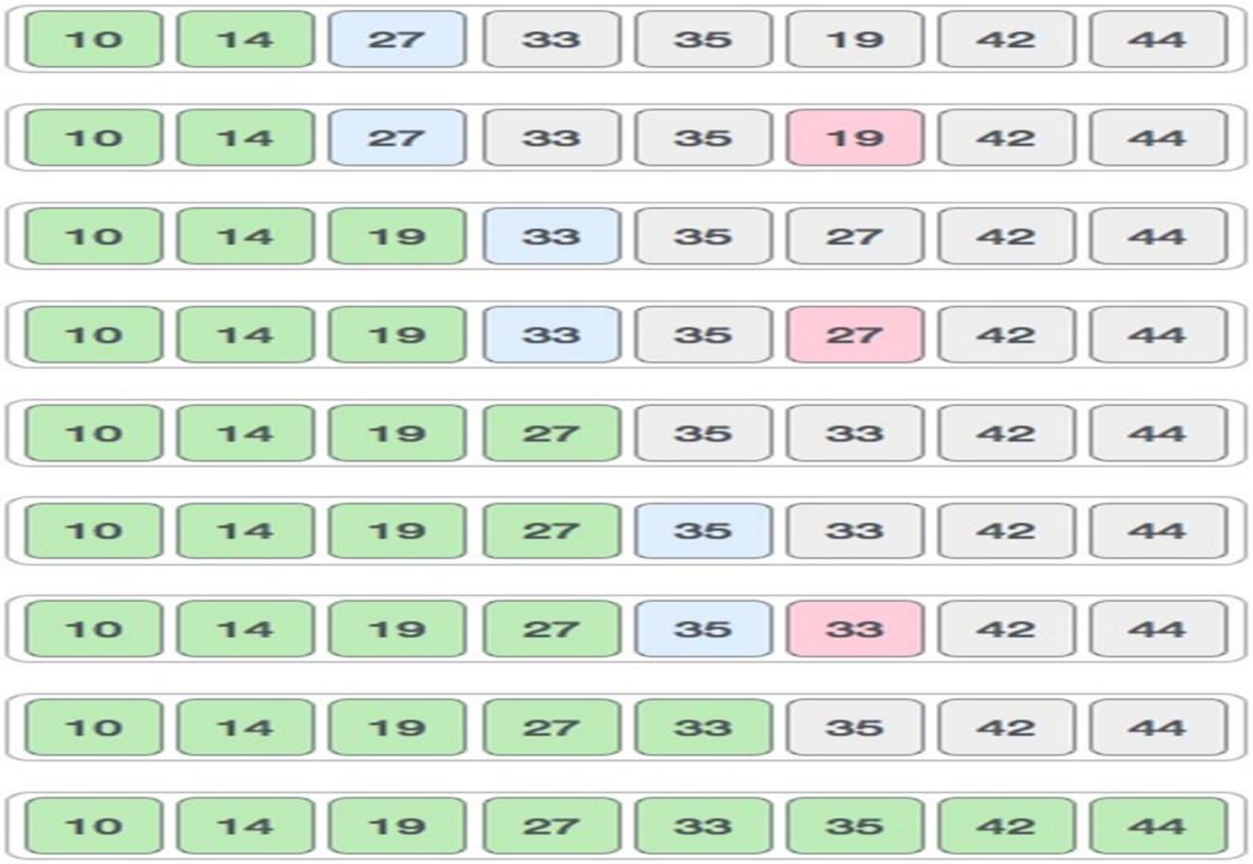
Unsorted Array= 	
	For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.
	So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.
	For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

How Selection Sort Works?

	We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.
	After two iterations, two least values are positioned at the beginning in a sorted manner.

How Selection Sort Works?

- The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process.



THANK YOU