2nd - Sem - Data structures and Algorithm Analysis

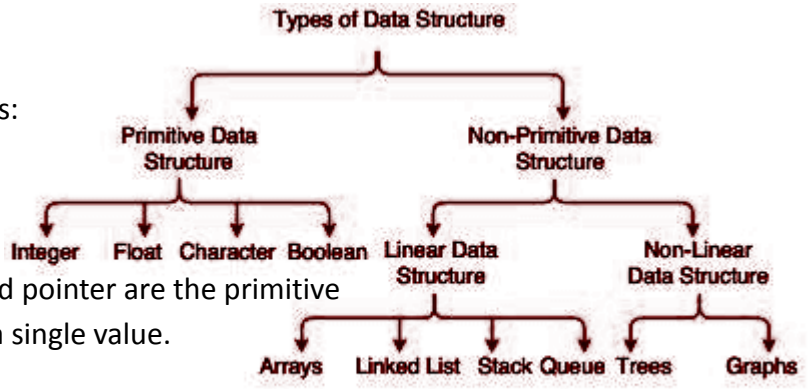
Module-1

#Data Structures:-Data may be organized in many different ways, the logicål or mathematical model of a particular organization of data is called a **data structure.**

-A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data.

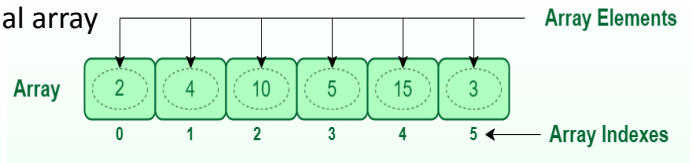
->Classification of Data Structure:

-There are two types of data structures:

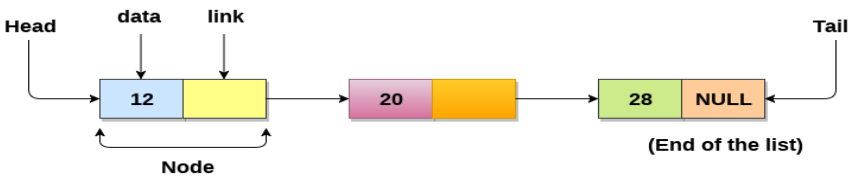


- **Primitive data structure:-**
  The primitive data structures are primitive data types.
  -The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

- **Non-primitive data structure:-** The non-primitive data structure is divided into two types:
  - ➢ Linear data structure:-The arrangement of data in a sequential manner is known as a linear data structure.
    -The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues.
    -In these data structures, one element is connected to only one another element in a linear form.
    - ➔ Array:-The simplest type of data structure is a linear(one dimensional) array.
      -Arrays is a collection of similar types of data items.
      -It is one of the simplest data structures where each data element can be randomly accessed by using its index number.
      -eg of one dimensional array



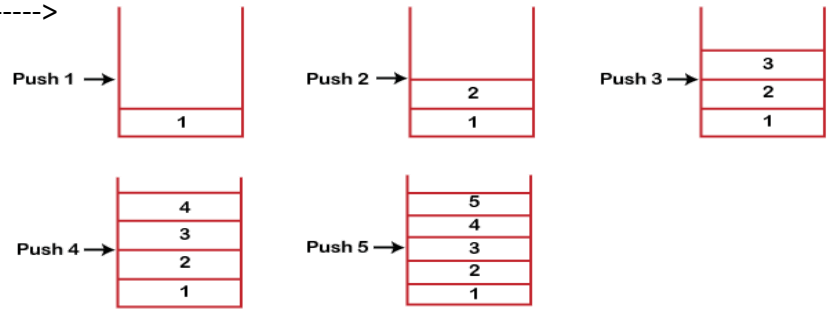
    - ➔ Linked List:-Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
      -A node contains two fields i.e. data storage and the address of the next node in the memory.
      -A null or X symbol means that the list is ended.



➔ Stack:-A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.

-It's insertion and deletion can take place only at one end called the **top** .

-eg:--------->



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

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

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

-eg:-

➢ 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.

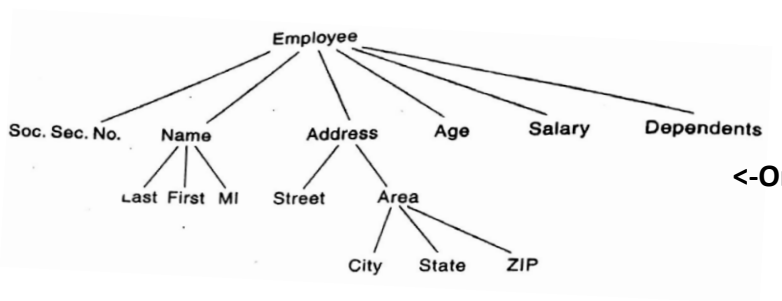
-In this case, the elements are arranged in a random manner.

-The example is trees and graphs.

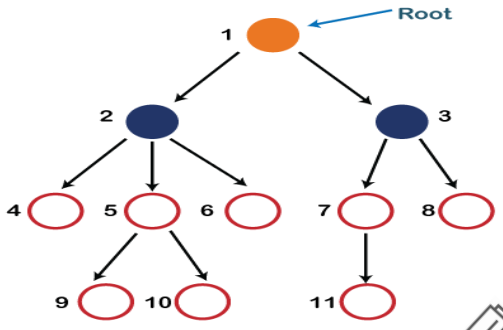
➔ Trees:-A tree is also one of the data structures that represent **hierarchical data**.

-In this structure, the root is at the top, and its branches are moving in a downward direction.

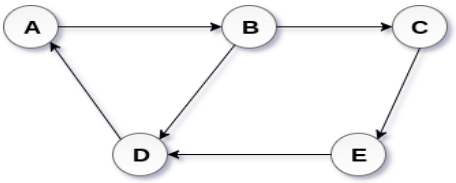
-eg:-



<-Or->



➔ Graphs:-A graph can be defined as group of vertices and edges that are used to connect these vertices.
-A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.
-Ex: A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



***Data Structure operations:**-This are the 4 common operations that can be performed on the data structures are:

- **Traversing:-** Traversing a Data Structure means to visit the element stored in it.
  -For example, traversing is required while printing the names of all the employees in a department.
- **Searching:-** Searching means to find a particular element in the given data-structure.
  -It is considered as successful when the required element is found.
  - For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
  -We can search for any element in a data structure.
- **Insertion:-** We can also insert the new element in a data structure.
  -For example, we can use the insertion operation to add the details of a new employee the company has recently hired.
- **Deletion:-** We can also perform the delete operation to remove the element from the data structure.
  -For example, we can use the deleting operation to delete the name of an employee who has left the job.

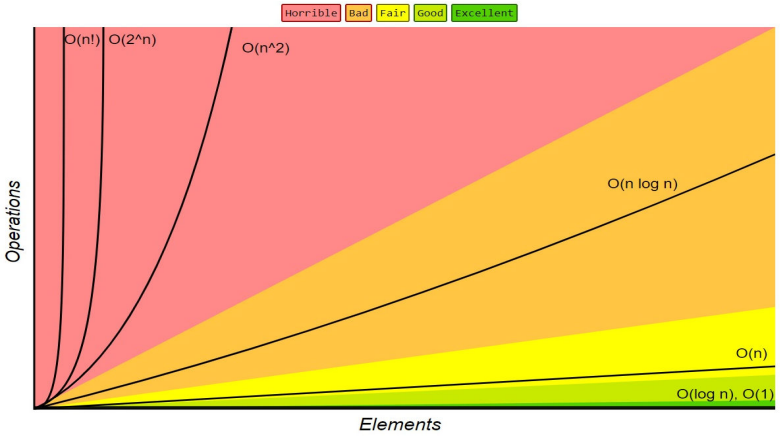
-The Following two operations ,which are used in specific situations,they are;

- **Sorting:**- We can sort the elements of a data structure either in an ascending or descending order.
  -For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order
- **Merging:-**Combining the records in two different sorted files into a single sorted file.

#Algorithms:-An algorithm is a well defined list of steps for solving a particular problem.
-The time and space it uses are the two major measures of the efficiency of an algorithm.
-The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task.
-It is not the complete program or code; it is just a solution (logic) of a problem.
-Ex: Algorithm to add two numbers and display the result.

Step 1 – START
Step 2 – declare three integers a, b & c
Step 3 – define values of a & b
Step 4 – add values of a & b
Step 5 – store output of step 4 to c
Step 6 – print c
Step 7 – STOP

***Algorithm Complexity:**-Big-O notation represents the algorithmic complexity.
-The performance of the algorithm can be measured in two factors:



- **Time complexity:** The time complexity of an algorithm is the amount of time required to complete the execution.
  -The time complexity of an algorithm is denoted by the big O notation.
  -The time complexity is mainly calculated by counting the number of steps to finish the execution.
  -the worst-time complexity as it is the maximum time taken for any given input size.
  -The time complexity of an algorithm is the amount of time it takes for each statement to complete.
  -As a result, it is highly dependent on the size of the processed data.
  -The input size has a strong relationship with time complexity. As the size of the input increases, so does the runtime, or the amount of time it takes the algorithm to run.

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

->Asymptotic notation:-Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by observing its behaviour as the input size increases.
-Asymptotic notations are classified into three types:

1. **Big-Oh (O) notation:**-Used e to describe the performance or complexity of an algorithm.
   - Describes the **worst-case scenario**, and can be used to describe the maximum execution time (asymptotic upper bounds) required or the space used (e.g. in memory or on disk) by an algorithm.
2. **Big Omega (Ω) notation:**-it is also used to describe the performance or complexity of an algorithm.
   -specifically describes the **best-case scenario,** and can be used to describe the minimum execution time (asymptotic lower bounds) required or the space used (e.g. in memory or on disk) by an algorithm
3. **Big Theta (Θ) notation:**-it is also Used to describe the performance or complexity of an algorithm.
   -Theta specifically describes the **average-case scenario**, and can be used to describe the average execution time required or the space used by an algorithm.

-There are three cases in Asymptotic Analysis they are;

- **Best Case:** It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.
- **Average Case:** You add the running times for each possible input combination and take the average in the average case.
- **Worst Case:** It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time possible.

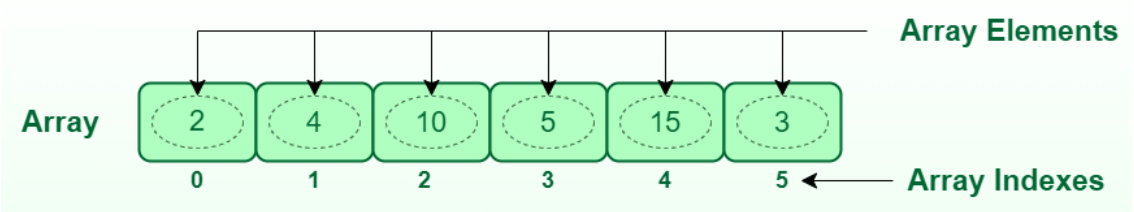
->Advantages of Algorithms:

-It is easy to understand.
-An algorithm is a step-wise representation of a solution to a given problem.
-In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.
-It helps in programming.
-It helps in debugging.

->Disadvantages of Algorithms:

-Writing an algorithm takes a long time so it is time-consuming.
-Understanding complex logic through algorithms can be very difficult.
-Branching and Looping statements are difficult to show in Algorithms(imp).
-No standard form to write an algorithm text.

#Arrays:-An array is a collection of items stored at contiguous memory locations.
-The idea is to store multiple items of the same type together.
-It is one of the simplest data structures where each data element can be randomly accessed by using its index number.



-basic operations supported in the array -

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular inde
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

->Advantage of array

- Easier to declare and use
- Can be used with most of the data type
- Easy to find elements
- Multi dimensional

->Disadvantage of array

- Wastage of memory space if all array elements are not used.
- Size of an array is fixed once declared
- Array is homogenous. It means that the elements with similar data type can be stored in it.

*** Ordered lists:**-One of the simplest and most commonly found data object is the ordered or linear list.

-Eg: (monday,tue,wed,thu,fri,sat,sun)

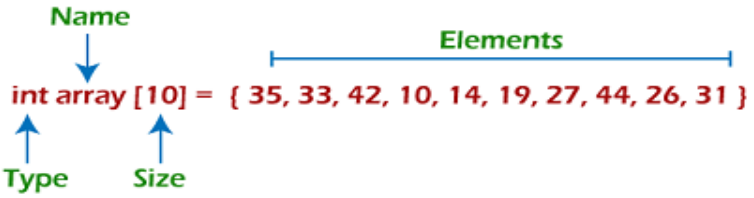
-or the values in a card deck (2,3,4,5,6,7,8,9,10,jack,queen,king,Ace).

-This are the various operations that can be performed on the ordered lists they are;

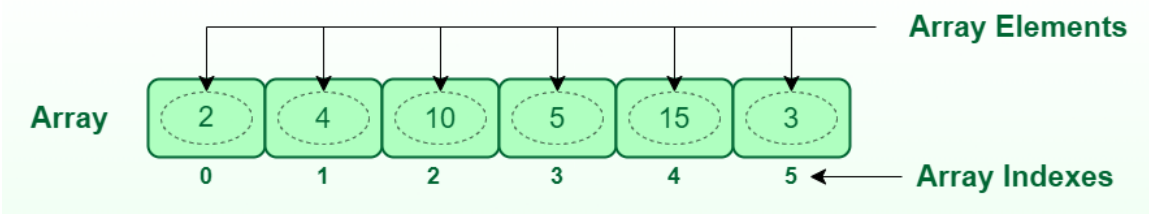
- Find the length of the list
- Read the list from left to right or right to left.
- Recover the i-th element
- Store a new value into the i-th position
- Delete the element from the i-th position.

-The most common way to represent an ordered list is by an array.

***representation of array:**-We can represent an array in various ways in different programming languages.

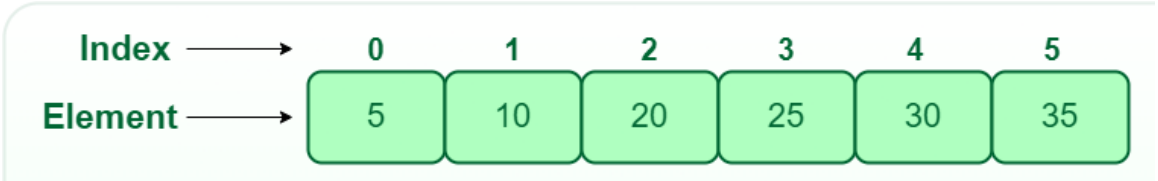


- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

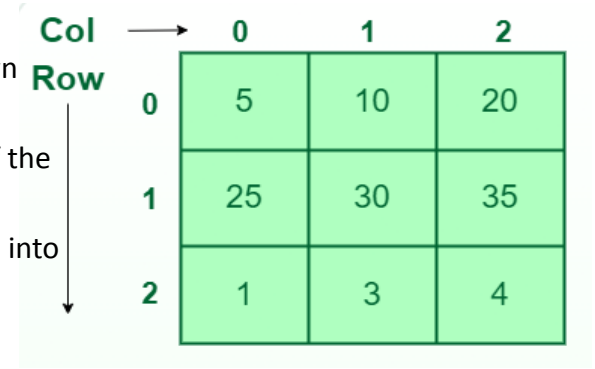


→**Types of array:**-There are majorly two types of arrays they are;

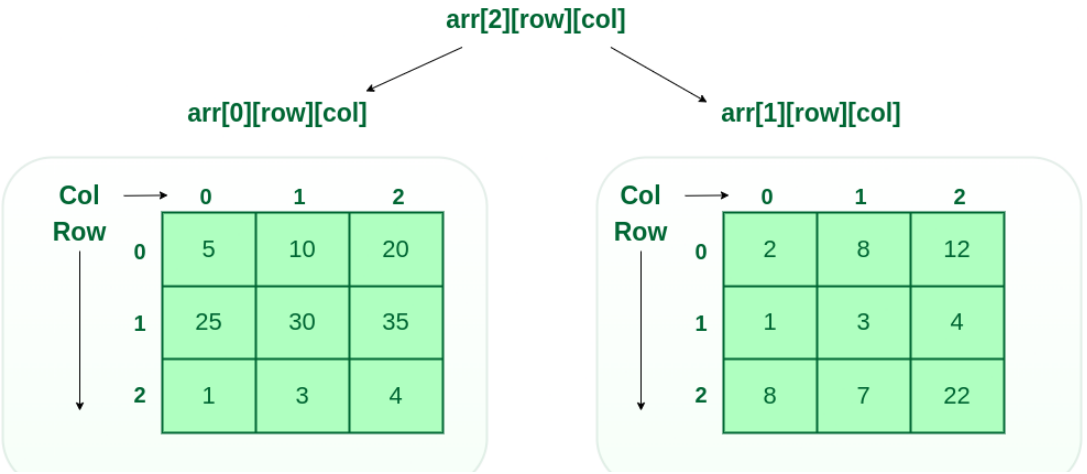
1. **One dimensional array**:-A one-dimensional array is a kind of linear array.
   -It involves single sub-scripting.
   -The [] (brackets) is used for the subscript of the array and to declare and access the elements from the array.
   -Eg:int a[10];



2. **Multi-dimensional array**:-In multi-dimensional arrays, we have two categories:
   - **Two-Dimensional Arrays**:-An array involving two subscripts [] [] is known as a two-dimensional array.
     -They are also known as the array of the array.
     -Two-dimensional arrays are divided into rows and columns
     -Example: int arr[5][5];

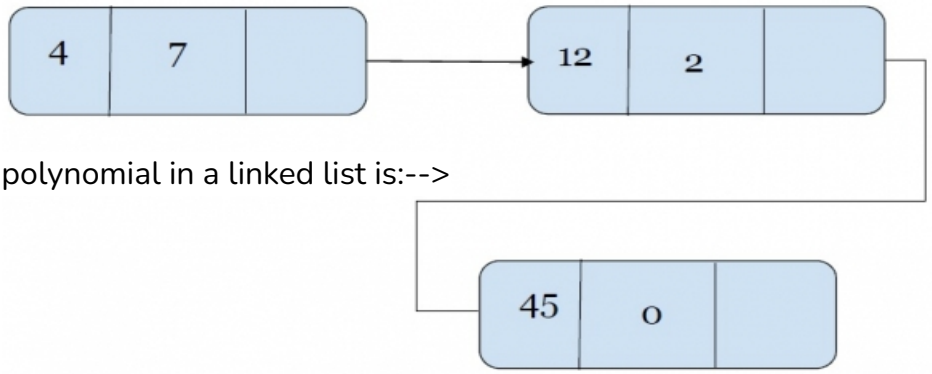


   - **Three-Dimensional Arrays**:- A Three Dimensional array is a collection of Two dimensional arrays.
     -It can be visualized as multiple 2D array stacked on top of each other .
     - Example: int a[5][5][5];



***polynomial addition**:-polynomial is a mathematical expression that consists of variables and coefficients.

-for example:-$4x^7 + 12x^2 + 45$



-representing polynomial in a linked list is:-->

-Eg:

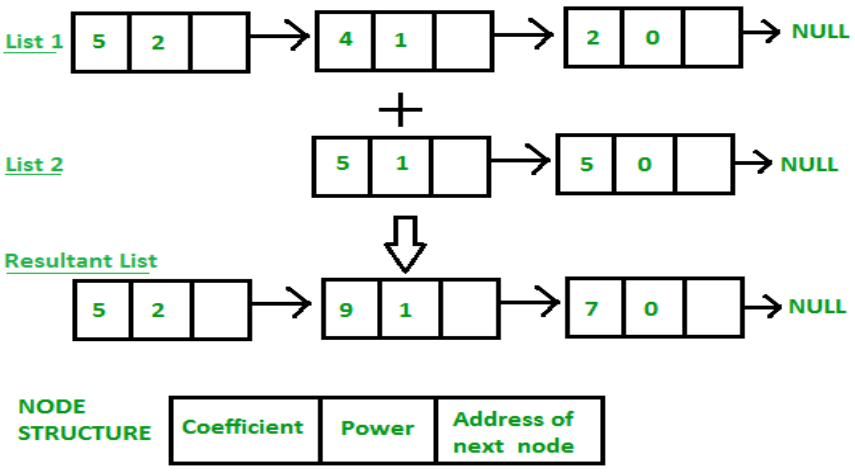
Input:

1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$



-Adding two polynomials that are represented by a linked list.

–We check values at the exponent value of the node.

-For the same values of exponent, we will add the coefficients.

-google ill nokkanam

#Stacks and Queues

***Stack:**- Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.

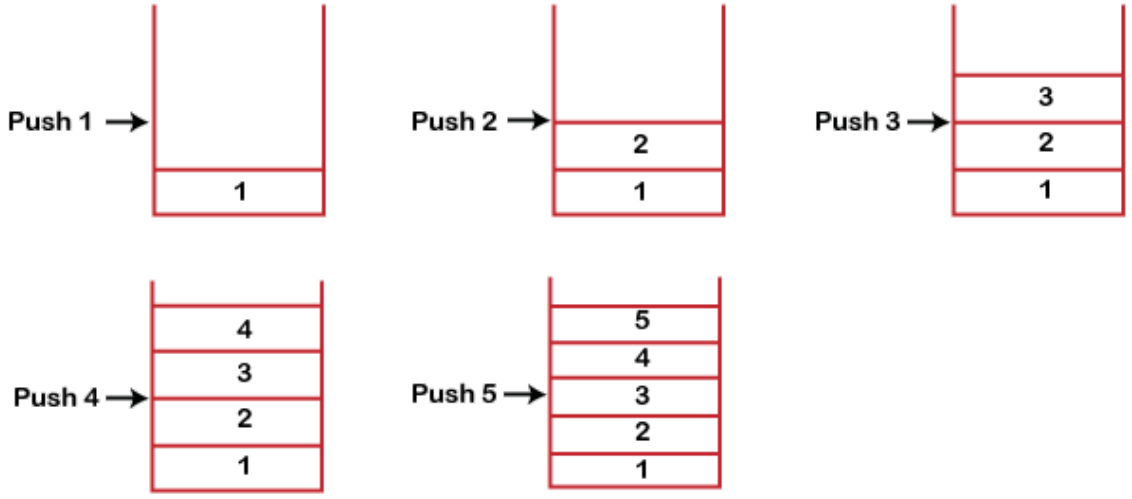
-Stack has one end, whereas the Queue has two ends (front and rear).

-It contains only one pointer top pointer pointing to the topmost element of the stack.

-Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the top of stack.

-In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

→**Working of Stack:**-Stack works on the LIFO pattern.
-As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
-Suppose we want to store the elements in a stack and let's assume that stack is empty.
-We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



- insert an element in a stack then the operation is known as a **push**.
-delete an element from the stack, the operation is known as a **pop**.
-when we insert an element into the stack but the stack is full that condition is called **overflow**.
-when we delete an element from the stack but the stack is empty that condition is called **underflow**

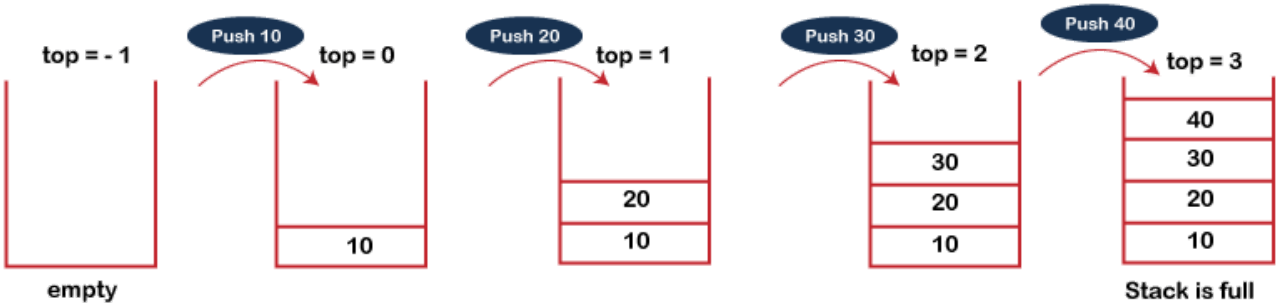
***Operations on stacks:**-The basic operations of stack are;

- **push():**- to insert an element into the stack
- **pop():**- to remove an element from the stack
- **top():-** Returns the top element of the stack.
- **isEmpty():-** returns true if stack is empty else false.
- **isFull():-** It determines whether the stack is full or not.'
- **size():**- returns the size of stack.
- **count():-** It returns the total number of elements available in a stack.
- **display():-** It prints all the elements available in the stack.

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

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.

- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.



-Algorithm of Push

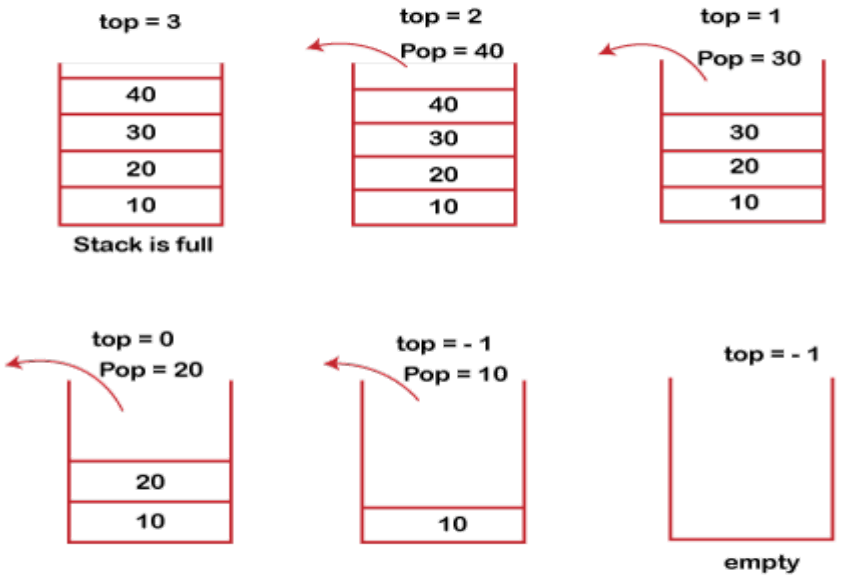
PUSH(STACK, TOP, MAXSTACK, ITEM)

This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTACK, then: Print: OVERFLOW, and Return.
2. Set TOP: TOP+ 1. [Increases TOP by 1.]
3. Set STACK[TOP]:= ITEM. [Inserts ITEM in new TOP position.]
4. Return.

->PUSH operation:-The steps involved in the POP operation is given below:

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



-Algorithm of Pop

POP(STACK, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP= 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM: STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP TOP-1. [Decreases TOP by I.]
4. Return.

***Application of stacks**

- Evaluation of Arithmetic Expressions:- explanation 👇
- Expression Conversion:-explanation 👇
- Backtracking:- explain cheyandaa
- Memory Management:- explain cheyandaa

–>Evaluation of Arithmetic Expressions:-A stack is a very effective data structure for evaluating arithmetic expressions in programming languages.

-An arithmetic expression consists of operands and operators.

-In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: A + (B - C)

-To evaluate the expressions, we need to be aware of the standard precedence rules for arithmetic expression.

-The precedence rules for the arithmetic operators are:

Operator	Priority
**, unary −, unary +, ¬	6
*, /	5
+, −	4
<, ≮, ≤, =, ≠, ≥, >, ≯	3
and	2
or	1

-There are three notations to represent an arithmetic expression:

1. **Prefix Notation**:-The prefix notation places the operator before the operands.
   -This notation was introduced by the Polish mathematician and hence often referred to as polish notation.
   -Example: + A B, -CD ,Z+Y etc.

-The first two expressions are in prefix notation because the operator comes before the operands.
-The last one is not a prefix notation because the operator is in between the operands

2. **Infix Notation**:-The infix notation is an expression in which each operator is placed between the operands.
   -Example: A + B, (C - D) etc.
   -All these expressions are in infix notation because the operator comes between the operands.
3. **Postfix Notation**:-The postfix notation places the operator after the operands.
   -This notation is just the reverse of Polish notation and also known as Reverse Polish notation.
   -Example: AB +, CD+, etc.
   -All these expressions are in postfix notation because the operator comes after the operands.
   -it also know as suffix notation.

-Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

→**Expression Conversion:**-Converting one form of expression to another is one of the important applications of stacks.

1. Infix to Postfix:- ethu matram explain cheythal mathii
2. Infix to Prefix
3. Postfix to Infix
4. Prefix to Infix

► **Infix to Postfix:**-To convert infix expression to postfix expression, first Scan the infix expression from left to right.
-Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.

- **Algorithm**
  - ☐ Step 1: Consider the next element in the input.
  - ☐ Step 2: If it is operand, display it.
  - ☐ Step 3: If it is opening parenthesis, insert it on stack.
  - ☐ Step 4: If it is an operator, then
    If stack is empty, insert operator on stack.
    If the top of stack is opening parenthesis, insert the operator on stack
    If it has higher priority than the top of stack, insert the operator on stack.
    Else, delete the operator from the stack and display it, repeat Step 4.
  - ☐ Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.
  - ☐ Step 6: If there is more input, go to Step 1.
  - ☐ Step 7: If there is no more input, delete the remaining operators to output.

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

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*+**

Example 2: converting **K + L - M*N + (O^P) * W/U/V * T + Q** expression into postfix form.

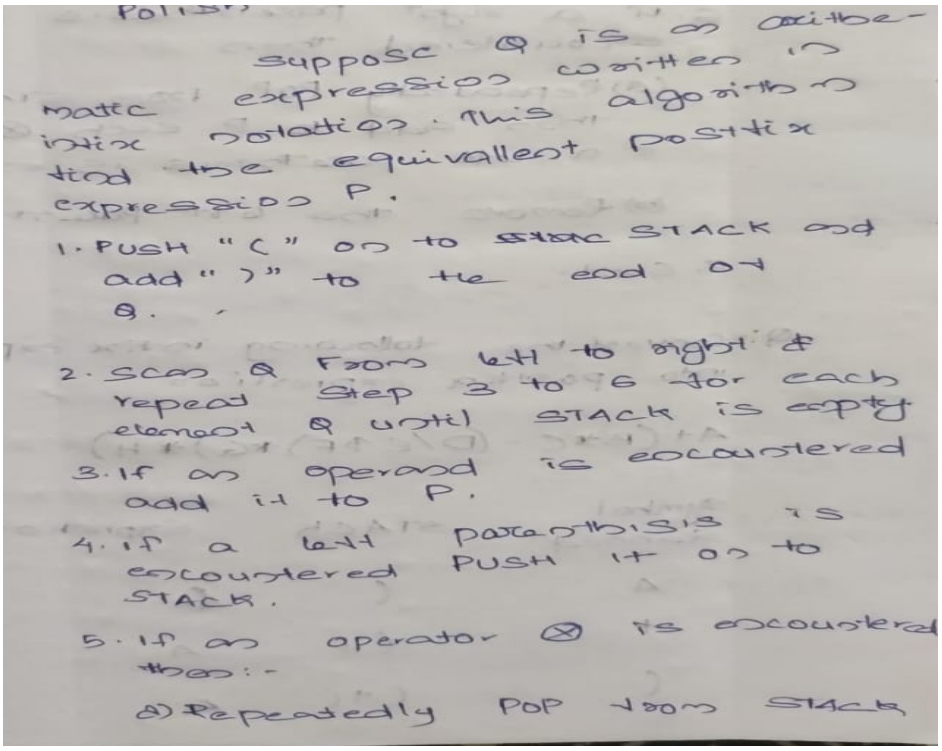
Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(	+ (	K L + M N *-

O	+ (	K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

Or

Note: first add parentheses to the expression

->Algorithm



add to P each ⊗ which has the same precedence as or higher precedence than ⊗

↑ precedence

(b) Add ⊗ to stack

6. If a right parenthesis is encountered then

(a) Repeatedly POP from STACK then add to P each ⊗ until a left parenthesis is encountered

(b) Remove the left parenthesis

7. Exit.

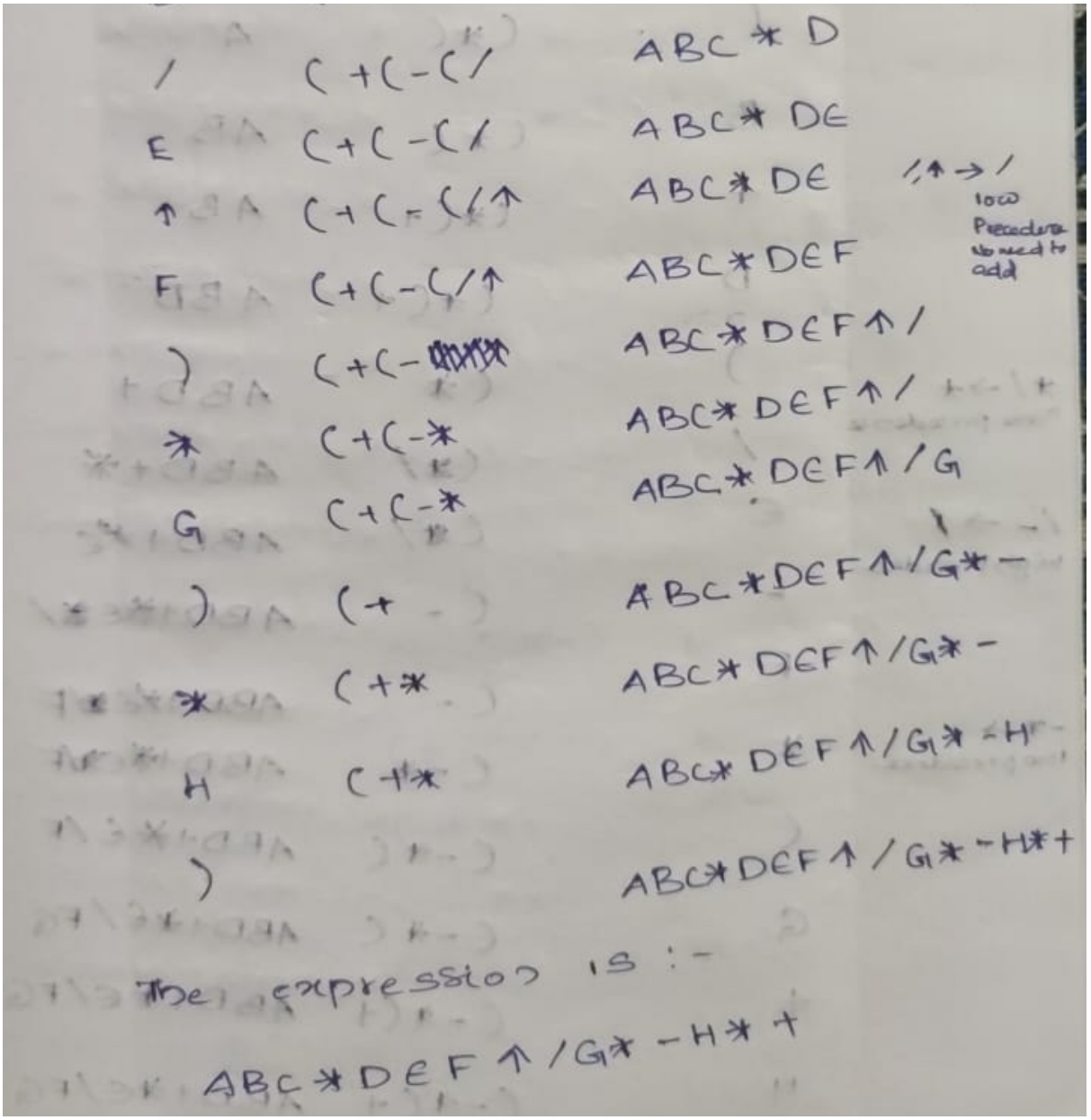
Q:- Convert the following infix expression into postfix expression

A+(B*C-(D/E↑F)*G)*H)

Symbol Scanned	STACK	Expression (P)
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D

(*, -) → * high Precedence

(, / → same Precedence



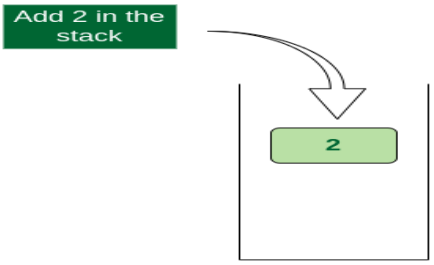
-Example 2: **A * (B + D) / E - F * (G + H / K)**

–>evaluation of postfix expressions:-Scan the expression from left to right and keep on storing the operands into a stack.

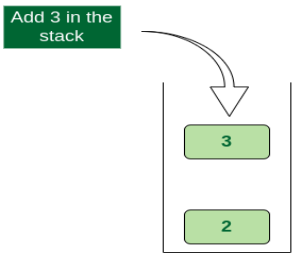
-Once an operator is received, pop the two topmost elements and evaluate them and push the result in the stack again.

-Consider the expression: Ex : 2 3 1 * + 9 -

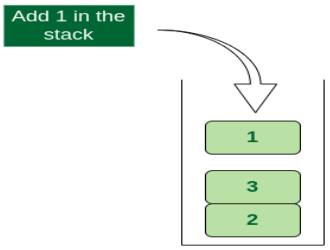
- Scan 2, it’s a number, So push it into the stack. Stack contains ‘2’.



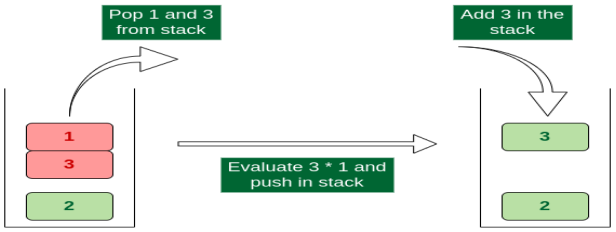
- Scan 3, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)



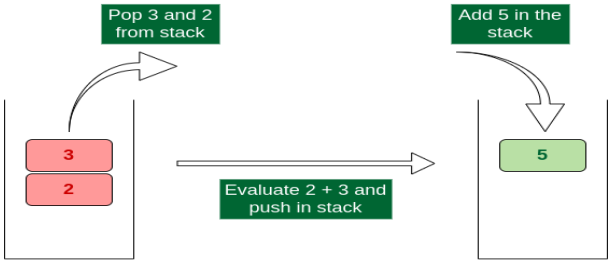
- Scan 1, again a number, push it to stack, stack now contains ‘2 3 1’



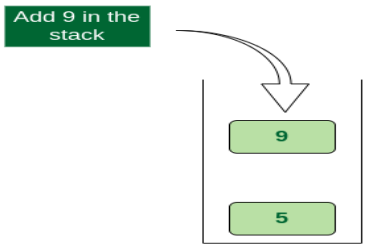
- Scan *, it’s an operator. Pop two operands from stack, apply the * operator on operands.
  -We get 3*1 which results in 3. We push the result 3 to stack.
  -The stack now becomes ‘2 3’.



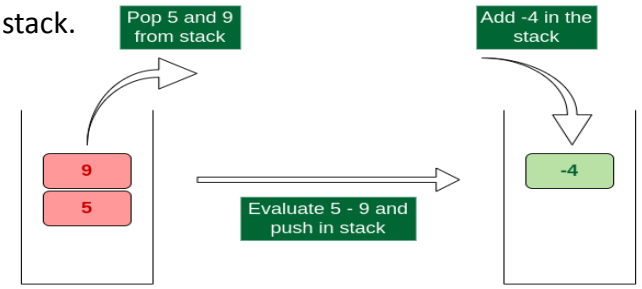
- Scan +, it’s an operator. Pop two operands from stack, apply the + operator on operands.
  -We get 3 + 2 which results in 5. We push the result 5 to stack.
  -The stack now becomes ‘5’.



- Scan 9, it’s a number. So we push it to the stack.
  -The stack now becomes ‘5 9’.



- Scan -, it's an operator, pop two operands from stack, apply the – operator on operands, we get 5 – 9 which results in -4.
  -We push the result -4 to the stack.
  -The stack now becomes '-4'.



- There are no more elements to scan, we return the top element from the stack (which is the only element left in a stack).
  -So the result becomes -4.

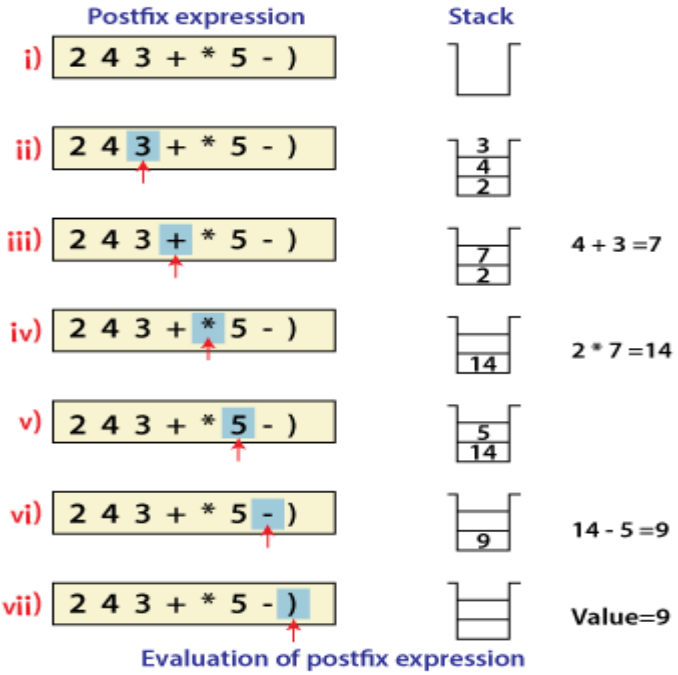
►Algorithm

- ☐ Step 1:Read the expression from left to right.
- ☐ Step 2:If the element encountered is an operand, push it into the stack.
- ☐ Step 3:If the element encountered is an operator, pop two operands a and b from the stack, apply the operator (b operator a) and push the result back into the stack.

Or

- ☐ Step 1:Scan the given expression from left to right and do the following for every scanned element.
- ☐ Step 2:If the element is a number, push it into the stack.
- ☐ Step 3:If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack.
- ☐ Step 4:When the expression is ended, the number in the stack is the final answer.

-Example :2 4 3 + * 5 -)



->Advantages of Stack:-

- Easy implementation
- A Stack helps to manage the data in the 'Last in First out' method.
- It allows you to control and handle memory allocation and deallocation.
- Insertion and deletion from one place.

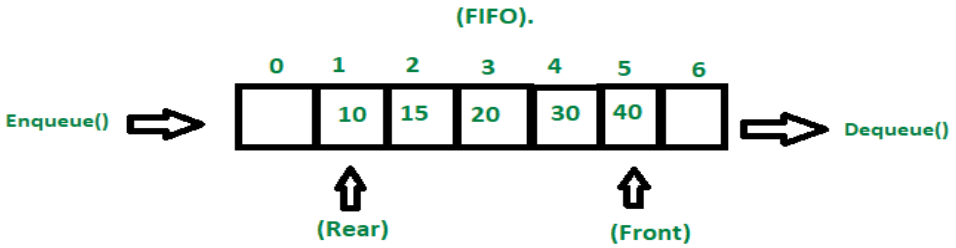
->Disadvantages of Stack:-

- It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.
- It has very limited memory.
- In Stack, random access is not possible.
- Stack overflow and underflow

***Queue:**-A queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end.

Or

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



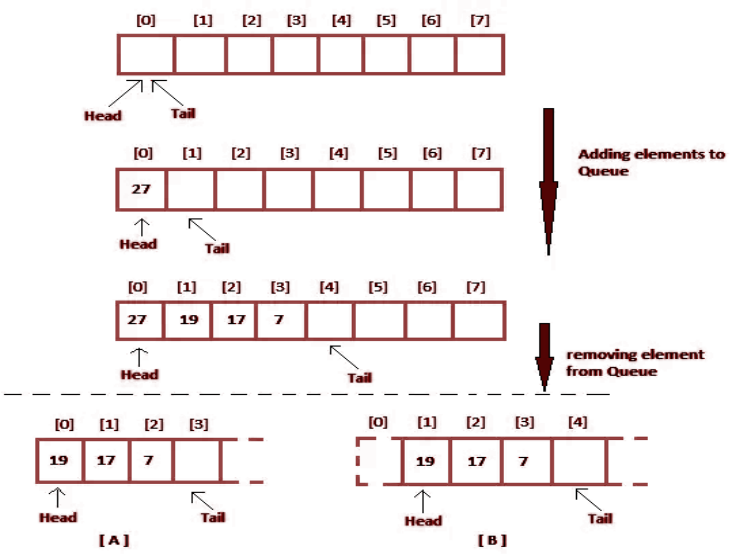
-Queue is referred to be as First In First Out list.

-The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue.**

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

- Cashier line in a store
- A car wash line

-Example



->Algorithm of Insertion (Enqueue)

1 – START
2 – Check if the queue is full.
3 – If the queue is full, produce overflow error and exit.
4 – If the queue is not full, increment rear pointer to point the next empty space.
5 – Add data element to the queue location, where the rear is pointing.
6 – return success.
7 – END

Or

1-Check if the queue is full or not.
2-If the queue is full, then print overflow error and exit the program.
3-If the queue is not full, then increment the tail and add the element.
4-Exit

->Algorithm of Deletion (Dequeue)

1 – START
2 – Check if the queue is empty.
3 – If the queue is empty, produce underflow error and exit.
4 – If the queue is not empty, access the data where front is pointing.
5 – Increment front pointer to point to the next available data element.
6 – Return success.
7 – END

Or

1-Check if the queue is empty or not.
2-If the queue is empty, then print underflow error and exit the program.
3-If the queue is not empty, then print the element at the head and increment the head.
4-Exit

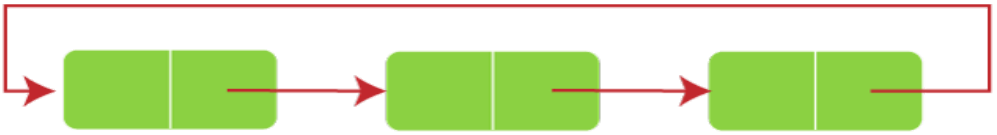
Delete cheyumbol front will become front+1 .
Add cheyumbol rear will become real +1

→**Types of Queue:-**There are four different types of queue that are listed as follows

1. **Simple Queue or Linear Queue:**-In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.
   -The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.
   -It strictly follows the FIFO rule.

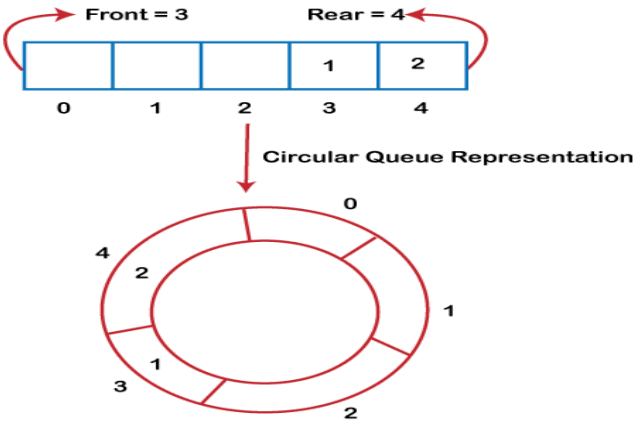
-The major drawback of using a linear Queue is that insertion is done only from the rear end.
-If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.
- In this case, the linear Queue shows the overflow condition as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

2. **Circular Queue:**-It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as Ring Buffer, as all the ends are connected to another end.
- The representation of circular queue is shown in the below image

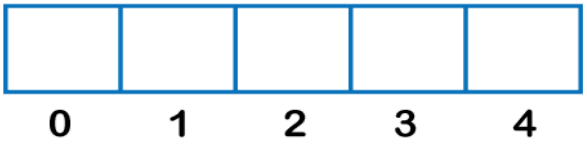


Circular Queue

-The drawback that occurs in a linear queue is overcome by using the circular queue.
-If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.
-The main advantage of using the circular queue is better memory utilization.

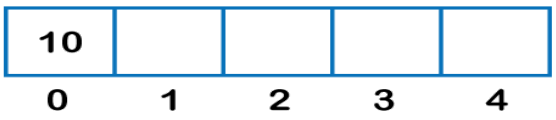


-Example:



Front= -1
Rear= -1

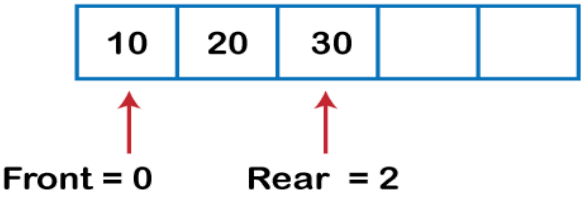
-Add 10



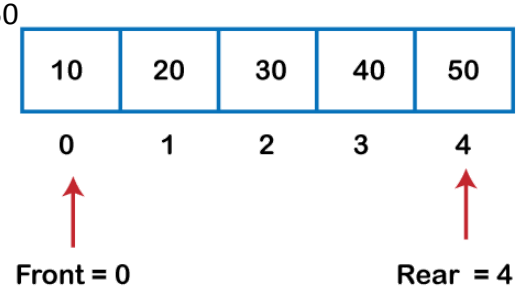
Front= 0

Rear= 0

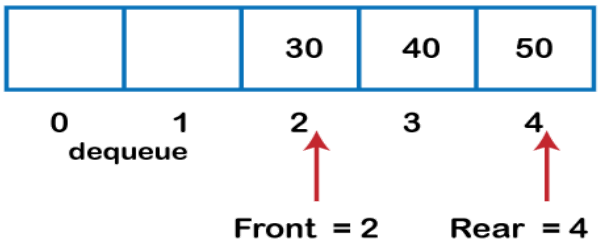
-Add 20 and 30



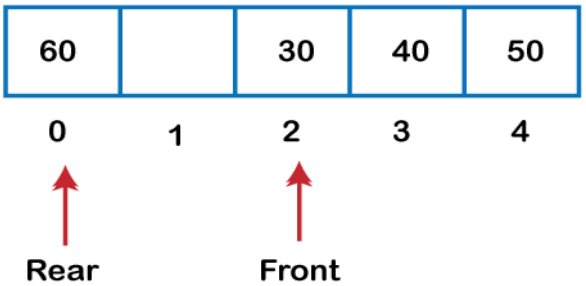
-Add 40 and 50



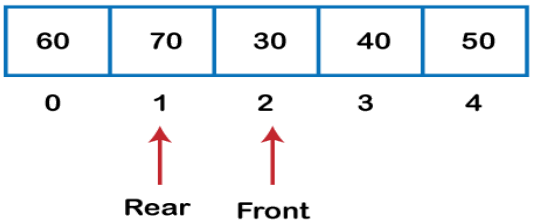
-dequeue 10 and 20



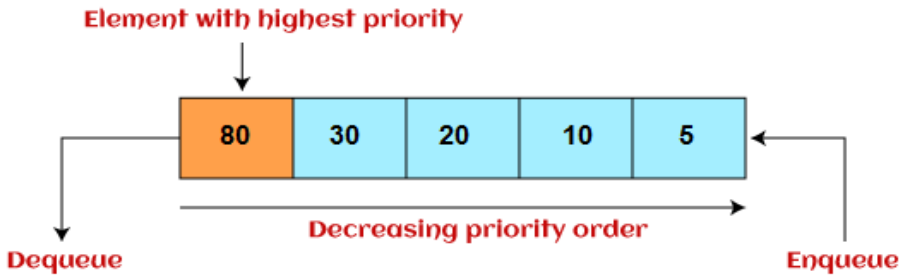
-Add 60



-Add 70



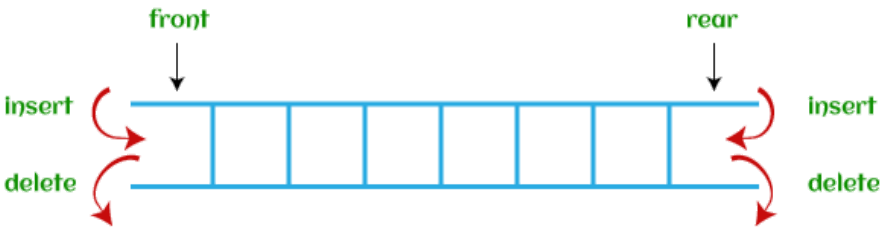
3. **Priority Queue:**-It is a special type of queue in which the elements are arranged based on the priority.
   -It is a special type of queue data structure in which every element has a priority associated with it.
   -Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle.
   -The representation of priority queue is shown in the below image



   -Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority.
   -There are two types of priority queue that are discussed as follows -

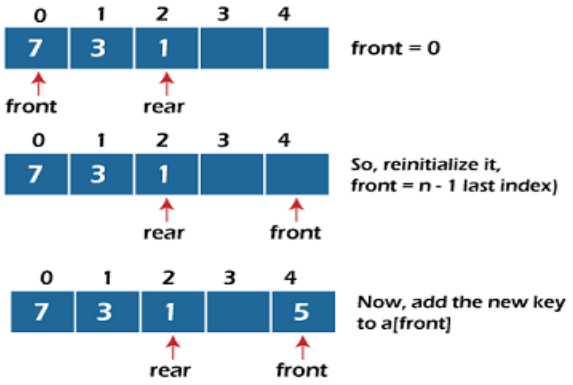
   - **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first.
     -Suppose an array with elements 7, 5, and 3 in the same order,
     -so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
   - **Descending priority queue** - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first.
     -Suppose an array with elements 7, 3, and 5 in the same order,
     -so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

4. **Deque/Double Ended Queue :**-In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear.
   -It means that we can insert and delete elements from both front and rear ends of the queue.
   -The representation of the deque is shown in the below image



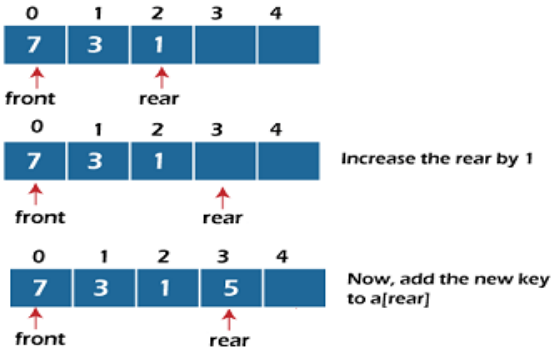
-There are the following operations that can be applied on a deque

- **Insertion at front**

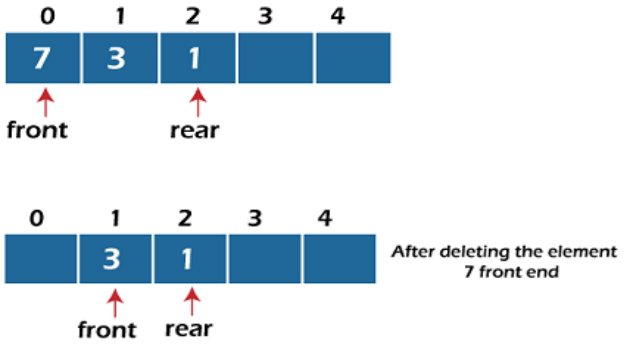


Example important alle

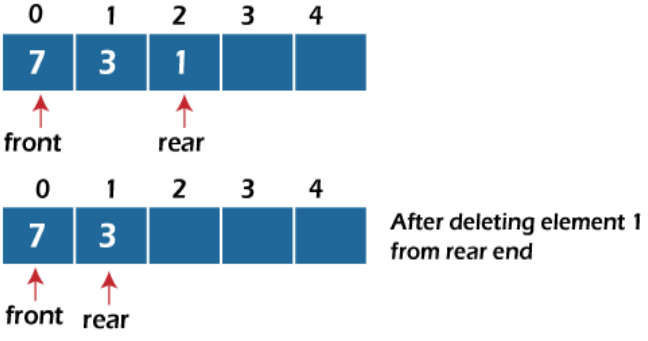
- **Insertion at rear**



- **Deletion at front**

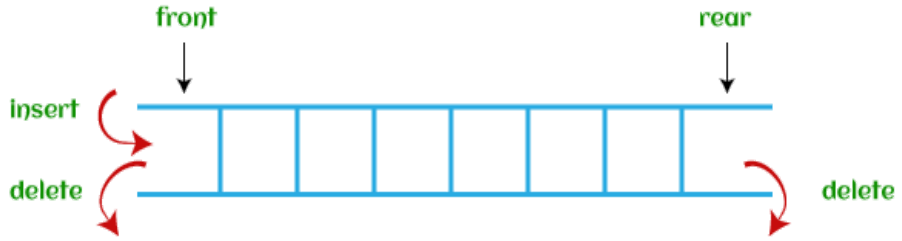


- **Deletion at rear**

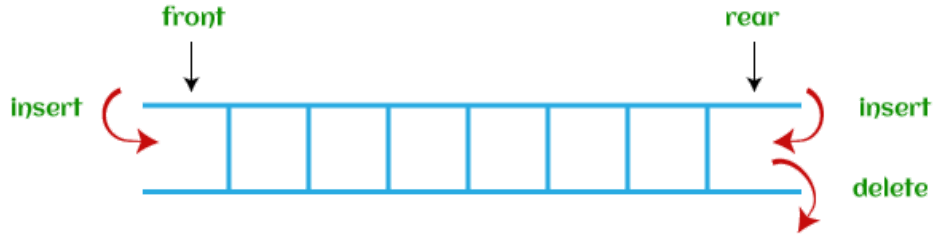


-There are two types of deque that are discussed as follows -

- **Input restricted deque** - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



- **Output restricted deque** - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



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

- **Enqueue**: The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **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.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull)**: It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

->Applications of Queue data structure :

1. **Task Scheduling:** Queues can be used to schedule tasks based on priority or the order in which they were received.
2. **Resource Allocation**: Queues can be used to manage and allocate resources, such as printers or CPU processing time.
3. **Batch Processing:** Queues can be used to handle batch processing jobs, such as data analysis or image rendering.

4. **Message Buffering:** Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
5. **Event Handling:** Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.
6. **Traffic Management:** Queues can be used to manage traffic flow in transportation systems, such as airport control systems or road networks.
7. **Operating systems**: Operating systems often use queues to manage processes and resources. For example, a process scheduler might use a queue to manage the order in which processes are executed.
8. **Network protocols:** Network protocols like TCP and UDP use queues to manage packets that are transmitted over the network. Queues can help to ensure that packets are delivered in the correct order and at the appropriate rate.
9. **Printer queues** :In printing systems, queues are used to manage the order in which print jobs are processed. Jobs are added to the queue as they are submitted, and the printer processes them in the order they were received.
10. **Web servers:** Web servers use queues to manage incoming requests from clients. Requests are added to the queue as they are received, and they are processed by the server in the order they were received.
11. **Breadth-first search algorithm:** The breadth-first search algorithm uses a queue to explore nodes in a graph level-by-level. The algorithm starts at a given node, adds its neighbors to the queue, and then processes each neighbor in turn.

***Difference between Stack and Queue**

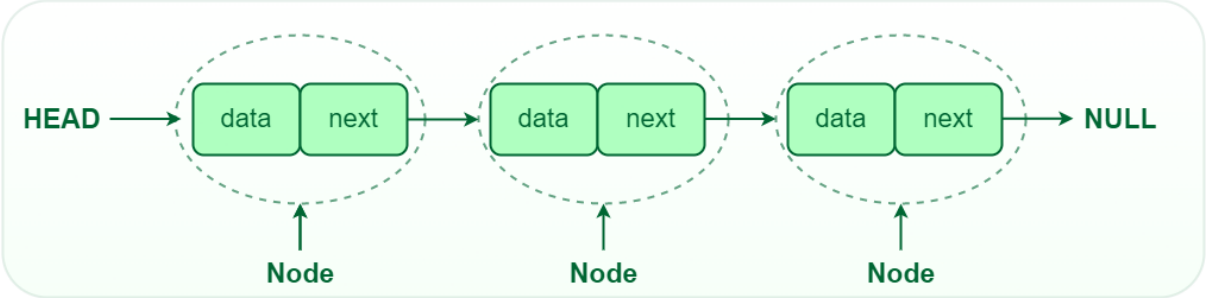
Stack	Queue
Push and pop at the top of the stack	Enqueue elements at the back of the queue And dequeue elements from the front of the queue
It follows LIFO	It follows FIFO
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.
Stack does not have any types.	Queue is of three types – 1. Circular Queue 2. Priority queue 3. double-ended queue.

***sparse matrices:** google

Module-2

#Linked List:-A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.

-The elements in a linked list are linked using pointers.



-A linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

-If start / head is null it means linked list is empty.

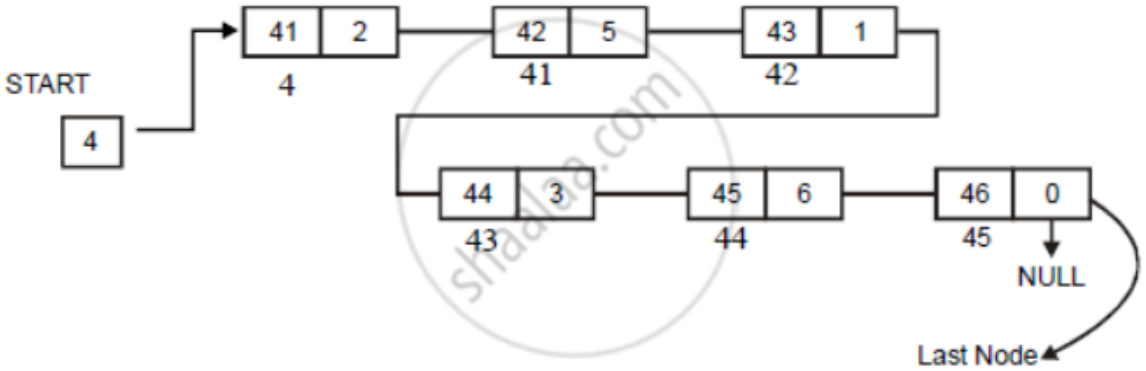
→**Representation of Linked list in memory:**-Linked lists can be represented in memory by using two arrays respectively known as **INFO** and **LINK.**

-INFO contains information of element and LINK next node address respectively.

-The Start contains address of first node.

-Pointer field of last node denoted by NULL which indicates the end of list.

e.g., Consider a linked list given below:



Start =4

INFO[4]=41 LINK[4]=2

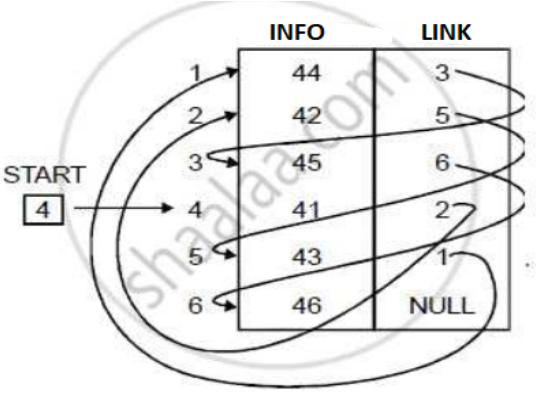
INFO[2]=42 LINK[2]=5

INFO[5]=43 LINK[5]=1

INFO[1]=44 LINK[1]=3

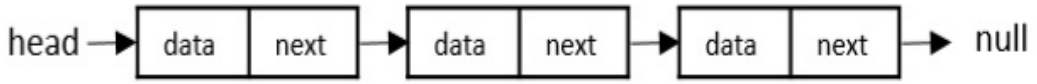
INFO[3]=45 LINK[3]=6

INFO[6]=46 LINK[6]=0 /end of the linked list



-There are three types of linked lists They are;

1. **Singly Linked List** – The nodes only point to the address of the next node in the list.



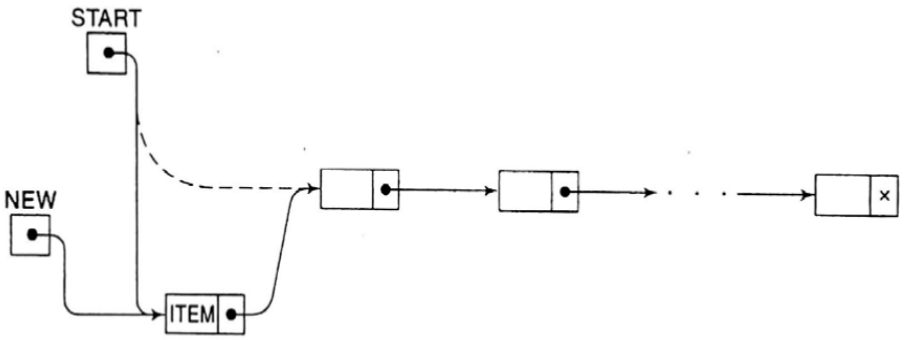
->Operations on Singly Linked List:-There are various operations which can be performed on singly linked list they are; Insertion ,Deletion ,Traverse and Searching.

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

  ➢ Insertion at beginning

INSFIRST(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the first node in the list.

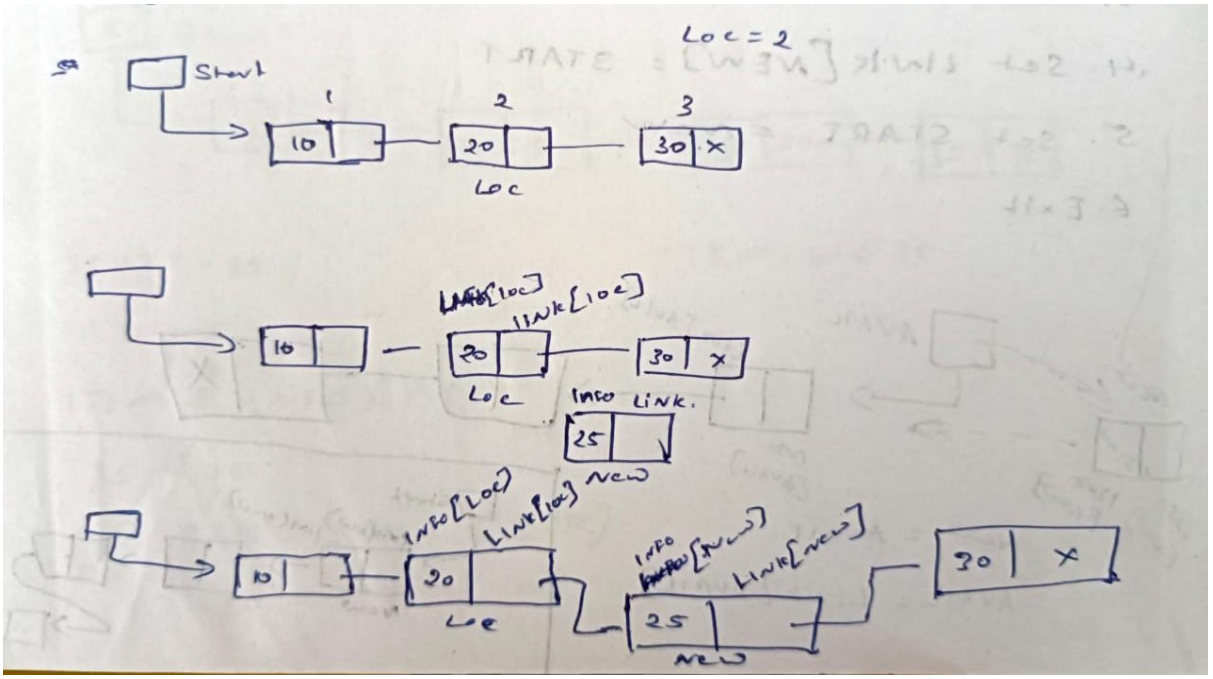
1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node]
4. Set LINK[NEW] := START. [New node now points to original first node.]
5. Set START := NEW. [Changes START so it points to the new node.]
6. Exit.



  ➢ Insertion after a given node

INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node.]
4. If LOC = NULL, then: [Insert as first node.]
   Set LINK[NEW] := START and START := NEW.
   Else: [Insert after node with location LOC.]
   Set LINK [NEW] := LINK[LOC] and LINK[LOC] := NEW.
   [End of If structure.]
5. Exit.



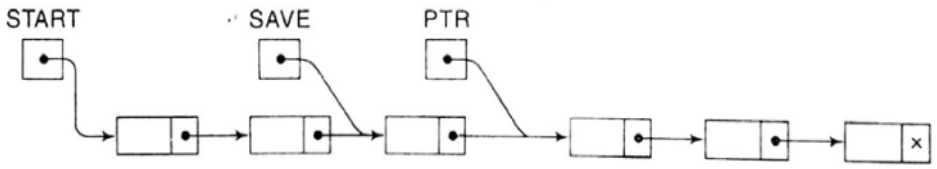
➢ Inserting into a sorted list

FINDA(INFO, LINK, START, ITEM, LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC = NULL.

1. [List empty?] If START = NULL, then: Set LOC := NULL, and Return.
2. [Special case?] If ITEM < INFO[START], then: Set LOC := NULL, and Return.
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]

4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If ITEM < INFO[PTR], then:
   Set LOC := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := SAVE.
8. Return.

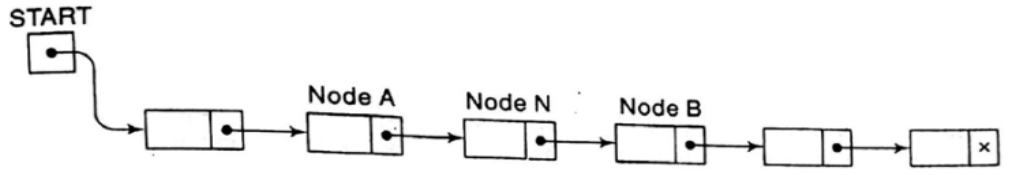
SAVE := PTR and PTR := LINK[PTR]



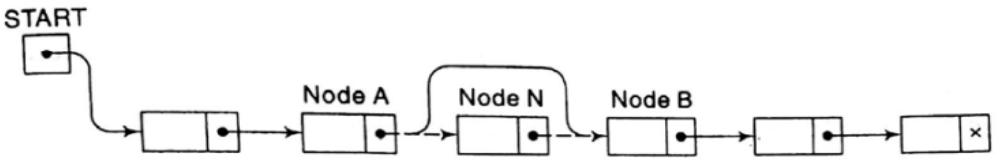
- **Deletion:**-The Deletion of a node from a singly linked list can be performed at different positions.

DEL(INFO, LINK, START, AVAIL, LOC, LOCP)
This algorithm deletes the node N with location LOC. LOCP is the location of the node which precedes N or, when N is the first node, LOCP = NULL.

1. If LOCP = NULL, then:
   Set START := LINK[START]. [Deletes first node.]
   Else:
   Set LINK[LOCP] := LINK[LOC]. [Deletes node N.]
   [End of If structure.]
2. [Return deleted node to the AVAIL list.]
   Set LINK[LOC] := AVAIL and AVAIL := LOC.
3. Exit.



(a) Before deletion



(b) After deletion

- **Traverse:**-In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

(Traversing a Linked List) Let LIST be a linked list in memory. This algorithm traverses LIST, applying an operation PROCESS to each element of LIST. The variable PTR points to the node currently being processed.

1. Set PTR := START. [Initializes pointer PTR.]
2. Repeat Steps 3 and 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR].
4. Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of Step 2 loop.]
5. Exit.

- **Searching:**-In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned.
  -Searching can be performed in sorted list and unsorted list.
  - ➢ <u>List is unsorted</u>

SEARCH(INFO, LINK, START, ITEM, LOC)
LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

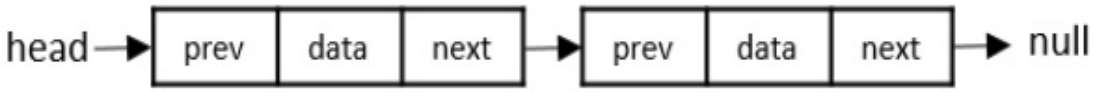
1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3. If ITEM = INFO[PTR], then:
   Set LOC := PTR, and Exit.
   Else:
   Set PTR := LINK[PTR]. [PTR now points to the next node.]
   [End of If structure.]
   [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.

➢ LIst is sorted

```
SRCHSL(INFO, LINK, START, ITEM, LOC)
LIST is a sorted list in memory. This algorithm finds the location LOC of the
node where ITEM first appears in LIST, or sets LOC = NULL.

1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3.      If ITEM < INFO[PTR], then:
            Set PTR := LINK[PTR]. [PTR now points to next node.]
        Else if ITEM = INFO[PTR], then:
            Set LOC := PTR, and Exit. [Search is successful.]
        Else:
            Set LOC := NULL, and Exit. [ITEM now exceeds INFO[PTR].]
        [End of If structure.]
    [End of Step 2 loop.]
4. Set LOC := NULL.
5. Exit.
```

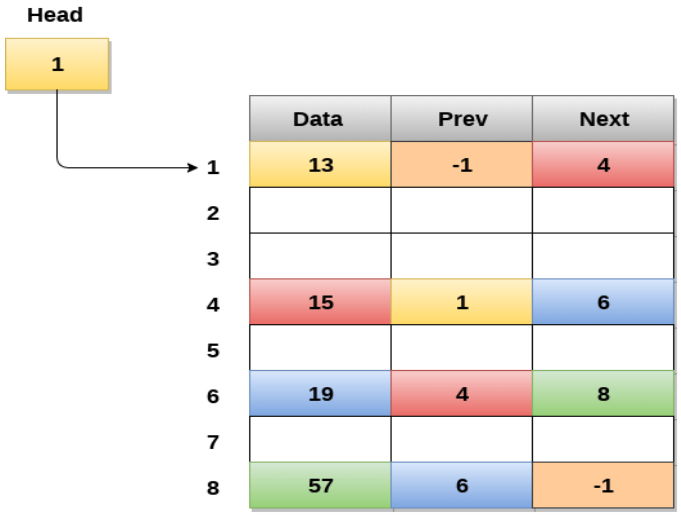
2. **Doubly Linked List** – The nodes point to the addresses of both previous and next nodes.
   -Doubly Linked Lists contain three "buckets" one bucket holds the data and the other two buckets hold the addresses of the previous and next nodes in the list.



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

→**Memory Representation of a doubly linked list:**-Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion.
-However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
-Below figure, the first element of the list that is i.e. 13 stored at address 1.
-The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the prev of the list contains null.
-The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.

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



→**Operations on doubly linked list:**- There are various operations which can be performed on doubly linked list they are; Insertion ,Deletion ,Traverse and Searching.

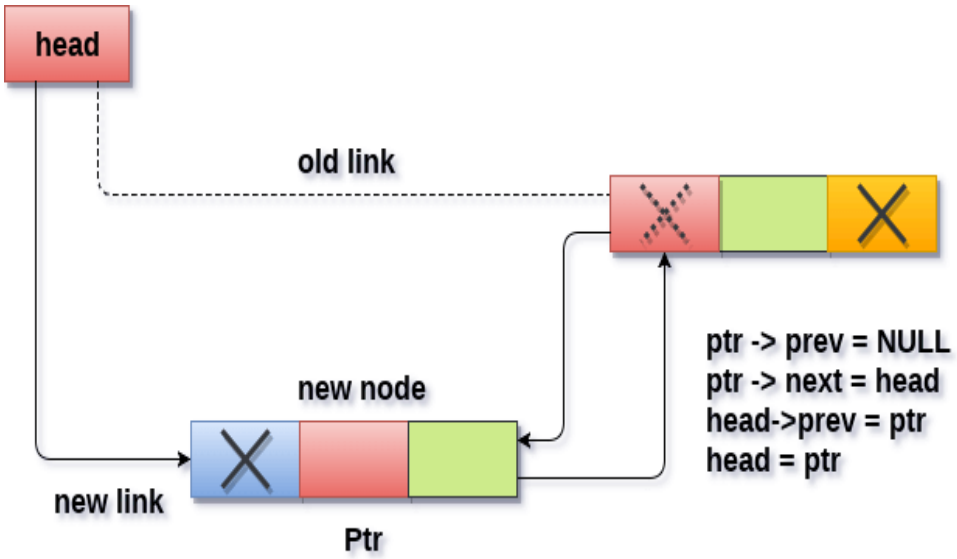
- Insertion:- The insertion into a doubly linked list can be performed at different positions.
  -Based on the position of the new node being inserted, the insertion is categorized into the following categories they are;

  ➢ <u>Insertion at beginning</u> :-Adding the node into the linked list at beginning.
  -Algorithm

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

<u>Explanation</u>

-Allocate the space for the new node in the memory.
-Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
-In the second scenario, the condition head == NULL become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.
-Since, the node being inserted is the first node of the list and therefore it must contain NULL in its prev pointer. Hence assign null to its previous part and make the head point to this node.



➢ Insertion at end:-Adding the node into the linked list to the end.
-Algorithm

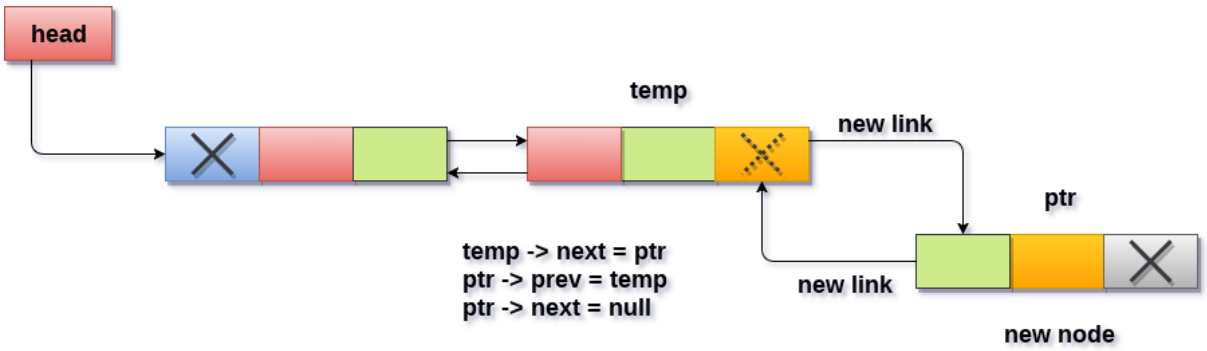
★ Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 11
[END OF IF]

★ Step 2: SET NEW_NODE = PTR
★ Step 3: SET PTR = PTR -> NEXT
★ Step 4: SET NEW_NODE -> DATA = VAL
★ Step 5: SET NEW_NODE -> NEXT = NULL
★ Step 6: SET TEMP = START
★ Step 7: Repeat Step 8 while
TEMP -> NEXT != NULL
★ Step 8: SET TEMP = TEMP -> NEXT
[END OF LOOP]

★ Step 9: SET TEMP -> NEXT = NEW_NODE
★ Step 10C: SET NEW_NODE -> PREV = TEMP
★ Step 11: EXIT

Explanation

-Allocate the memory for the new node. Make the pointer ptr point to the new node being inserted.
-Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.
-In the second scenario, the condition head == NULL become false. The new node will be inserted as the last node of the list. For this purpose, we have to traverse the whole list in order to reach the last node of the list. Initialize the pointer temp to head and traverse the list by using this pointer.
-the pointer temp point to the last node at the end of this while loop. Now, we just need to make a few pointer adjustments to insert the new node ptr to the list. First, make the next pointer of temp point to the new node being inserted i.e. ptr.
-make the previous pointer of the node ptr point to the existing last node of the list i.e. temp.
-make the next pointer of the node ptr point to the null as it will be the new last node of the list.



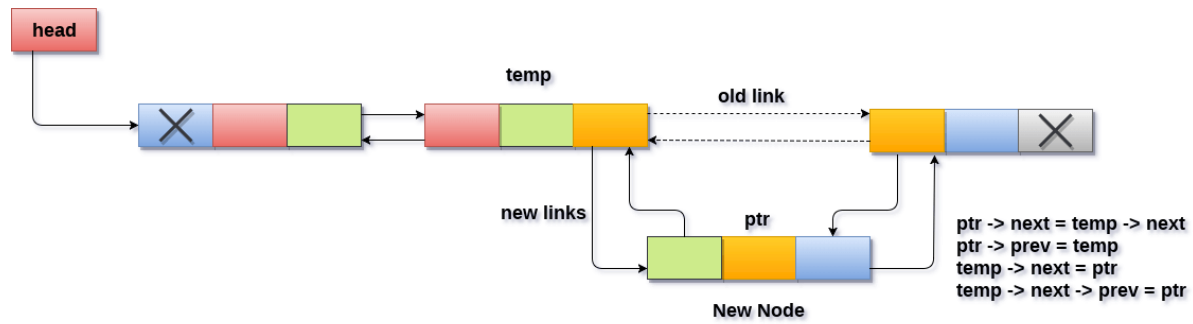
➢ Insertion after specified node:-Adding the node into the linked list after the specified node.
-Algorithm

★ Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 15
[END OF IF]

★ Step 2: SET NEW_NODE = PTR
★ Step 3: SET PTR = PTR -> NEXT
★ Step 4: SET NEW_NODE -> DATA = VAL
★ Step 5: SET TEMP = START
★ Step 6: SET I = 0
★ Step 7: REPEAT 8 to 10 until I
★ Step 8: SET TEMP = TEMP -> NEXT
★ STEP 9: IF TEMP = NULL
★ STEP 10: WRITE "LESS THAN DESIRED NO. OF ELEMENTS"
GOTO STEP 15
[END OF IF]
[END OF LOOP]

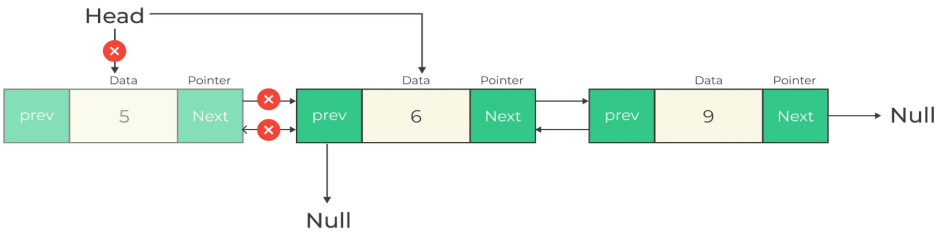
★ Step 11: SET NEW_NODE -> NEXT = TEMP -> NEXT
★ Step 12: SET NEW_NODE -> PREV = TEMP
★ Step 13 : SET TEMP -> NEXT = NEW_NODE
★ Step 14: SET TEMP -> NEXT -> PREV = NEW_NODE
★ Step 15: EXIT

Explanation
-Allocate the memory for the new node.
-Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.
-The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a fer pointer adjustments here. Make the next pointer of ptr point to the next node of temp.
-make the prev of the new node ptr point to temp.
-make the next pointer of temp point to the new node ptr.
-make the previous pointer of the next node of temp point to the new node.

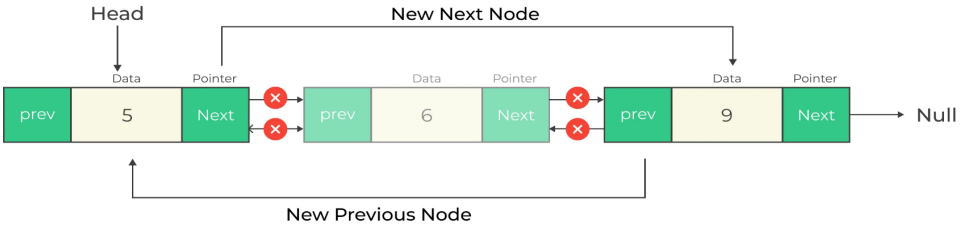


Insertion into doubly linked list after specified node

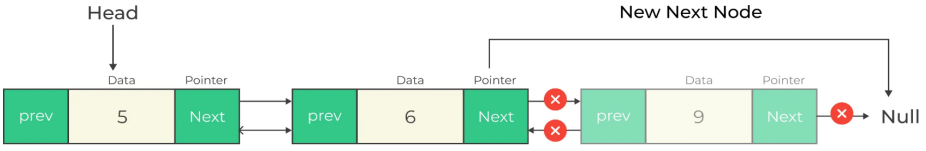
- Deletion :-we want to delete the node in the doubly linked list we have three ways to delete the node in another position.
  - ➢ <u>Deletion at beginning</u>



  - ➢ <u>Deletion at middle/specified position</u>



  - ➢ <u>Deletion at last</u>



-In Disadvantages, Doubly linked list occupy more space and often more operations are required for the similar tasks as compared to singly linked lists.
-It is easy to reverse the linked list.

- Traverse :-Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.
  -Algorithm

★ IF HEAD == NULL
WRITE "UNDERFLOW"
★ Copy the head pointer in any of the temporary pointer ptr.
★ then, traverse through the list by using while loop.
-Keep shifting value of pointer variable ptr until we find the last node.
-The last node contains null in its next part
★ Although, traversing means visiting each node of the list once to perform some specific operation.

-Here, we are printing the data associated with each node of the list.

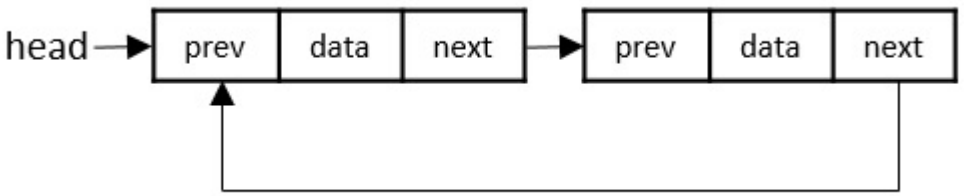
- **Step 1:** IF HEAD == NULL

  WRITE "UNDERFLOW"
  GOTO STEP 6
  [END OF IF]

- **Step 2:** Set PTR = HEAD
- **Step 3:** Repeat step 4 and 5 while PTR != NULL
- **Step 4:** Write PTR → data
- **Step 5:** PTR = PTR → next
- **Step 6:** Exit

- Searching:-Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

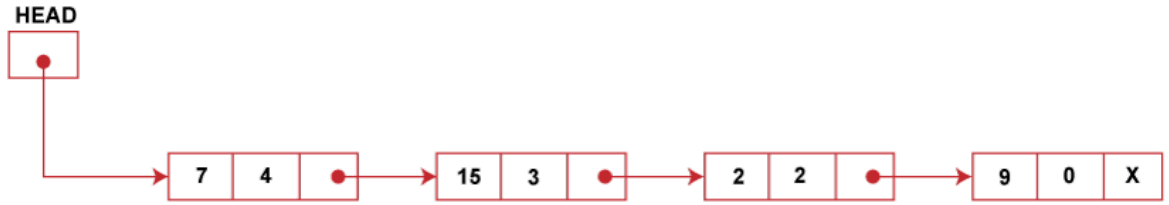
3. **Circular Linked List –** The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.
   -circular linked lists can exist in both singly linked list and doubly linked list.
   -Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



→**Representation of polynomials using Linked list:**-Polynomial are one of the most important applications of linked lists.
- Each node of a linked list representing polynomials parts they are;
-The first part contains the value of the coefficient of the term.
-The second part contains the value of the exponent.
-The third part, LINK points to the next term (next node).

-Consider a polynomial P(x) = 7x2 + 15x3 - 2 x2 + 9. Here 7, 15, -2, and 9 are the coefficients, and 4,3,2,0 are the exponents of the terms in the polynomial.

-On representing this polynomial using a linked list, we have

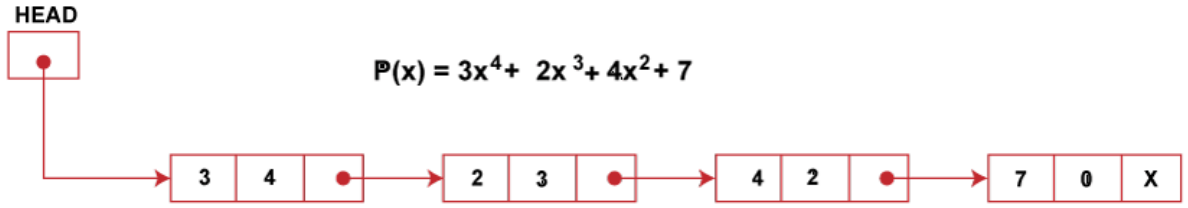


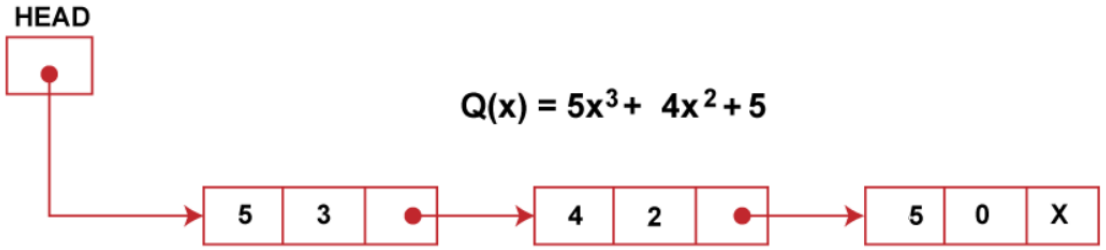
-Addition of Polynomials:-Let us consider an example to show how the addition of two polynomials is performed,

$$P(x) = 3x^4 + 2x^3 + 4x^2 + 7$$

$$Q(x) = 5x^3 + 4x^2 + 5$$

-These polynomials are represented using a linked list

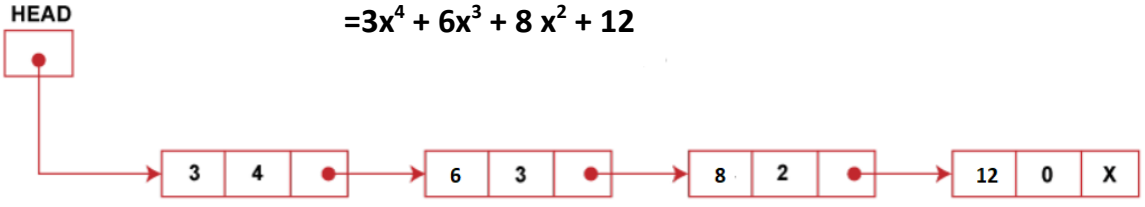




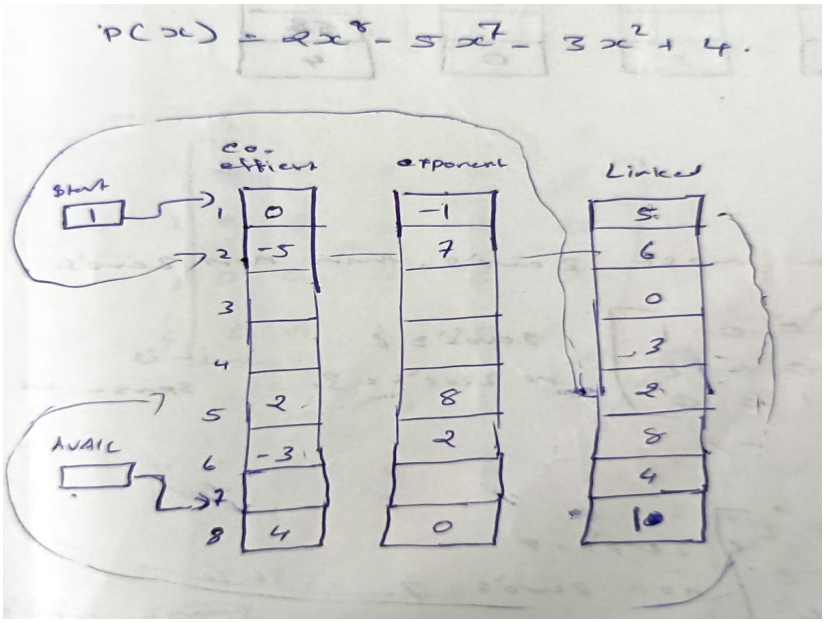
$$S(x) = [3x^4 + 2x^3 + 4x^2 + 7] + [5x^3 + 4x^2 + 5]$$

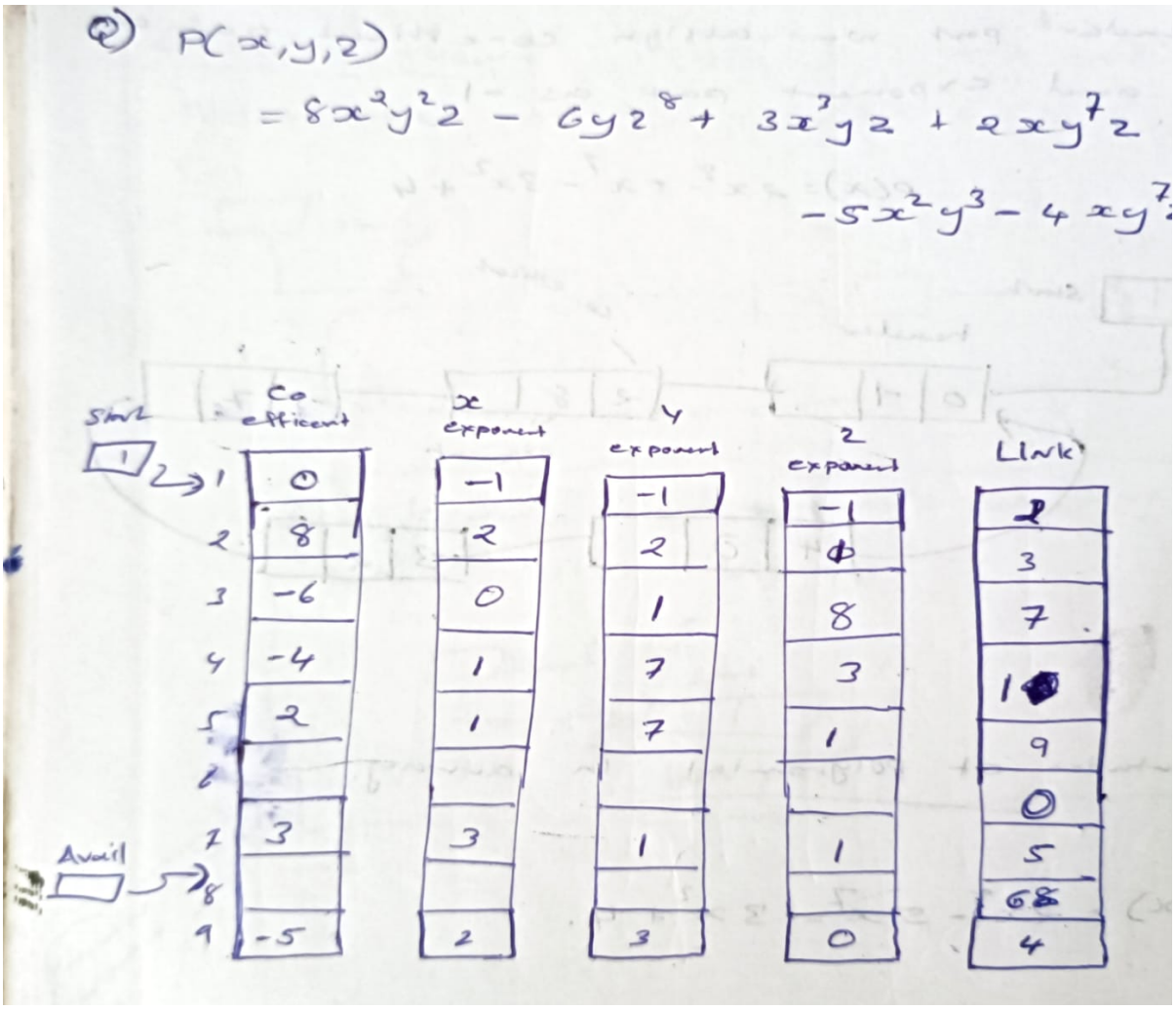
$$= 3x^4 + 6x^3 + 8x^2 + 12$$

-



→**representation of polynomial in array:-**





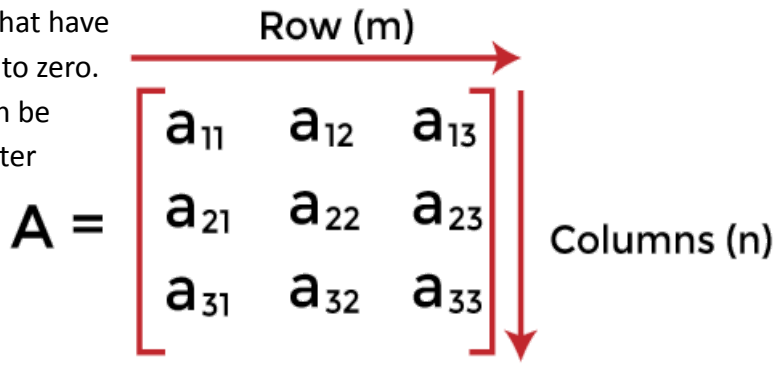
***Sparse matrix:-**A matrix can be defined as a two-dimensional array having 'm' rows and 'n' columns.

-A matrix with m rows and n columns is called m × n matrix.

-Sparse matrices are those matrices that have the majority of their elements equal to zero.

-In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

-Eg;



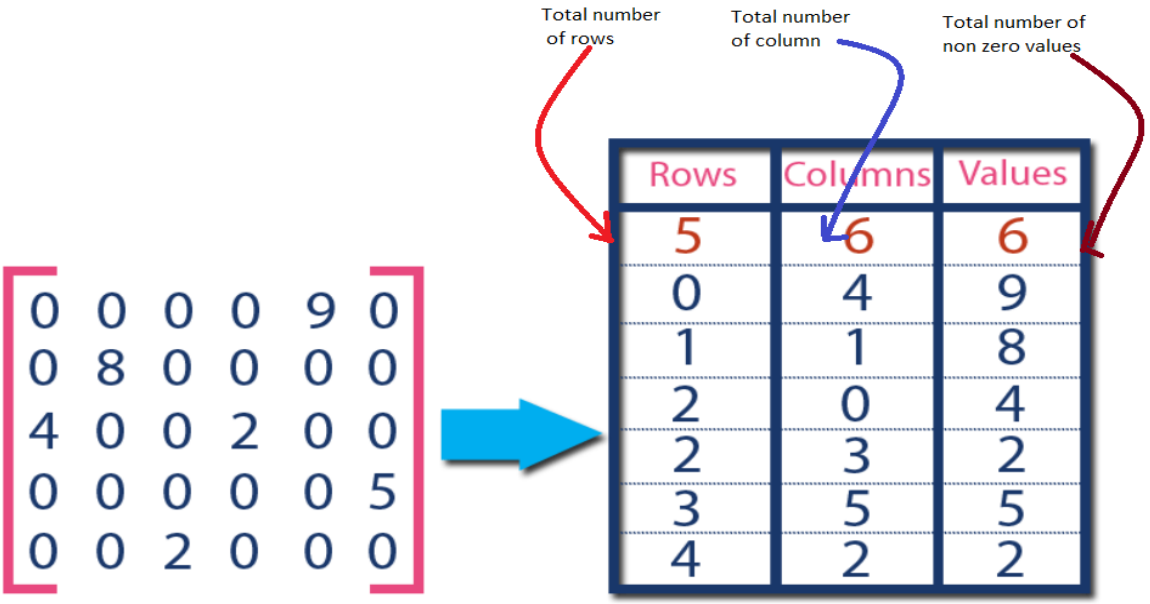
	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	0	0	0
4	1	0	0	0

-In given figure, we can observe a 5x4 sparse matrix containing 7 non-zero elements and 13 zero elements.

-Therefore it is a sparse matrix because here we have Number of zeros is greater than number of non zeros.

-Sparse Matrix Representations can be done in many ways following are two common representations:

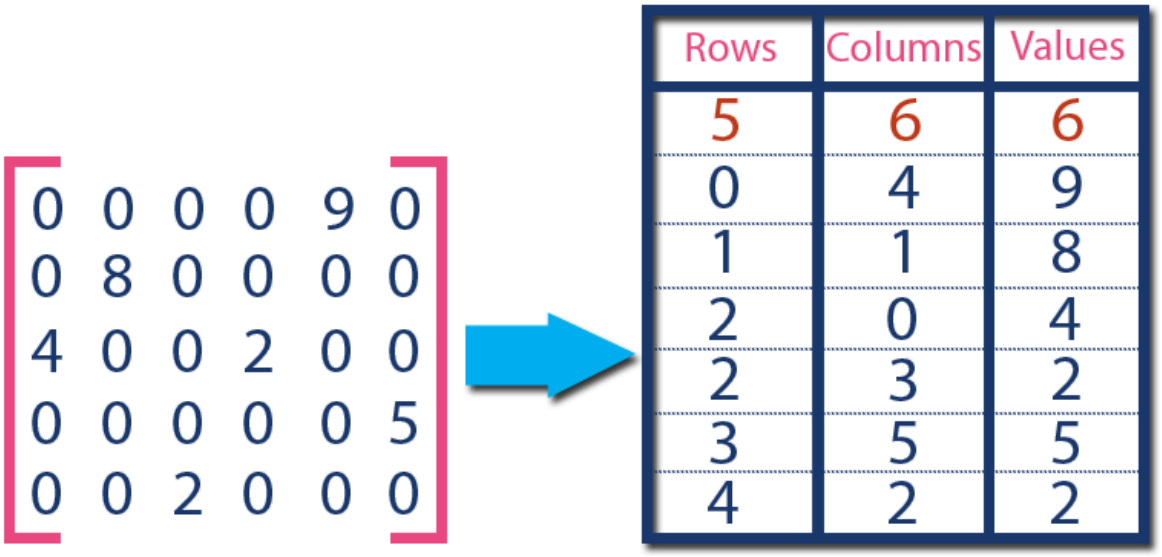
- **Array representation/Triplet Representation**:-In this representation, we consider only non-zero values along with their row and column index values.
  -In this representation, the 0th row stores the total number of rows, total number of columns and the total number of non-zero values in the sparse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



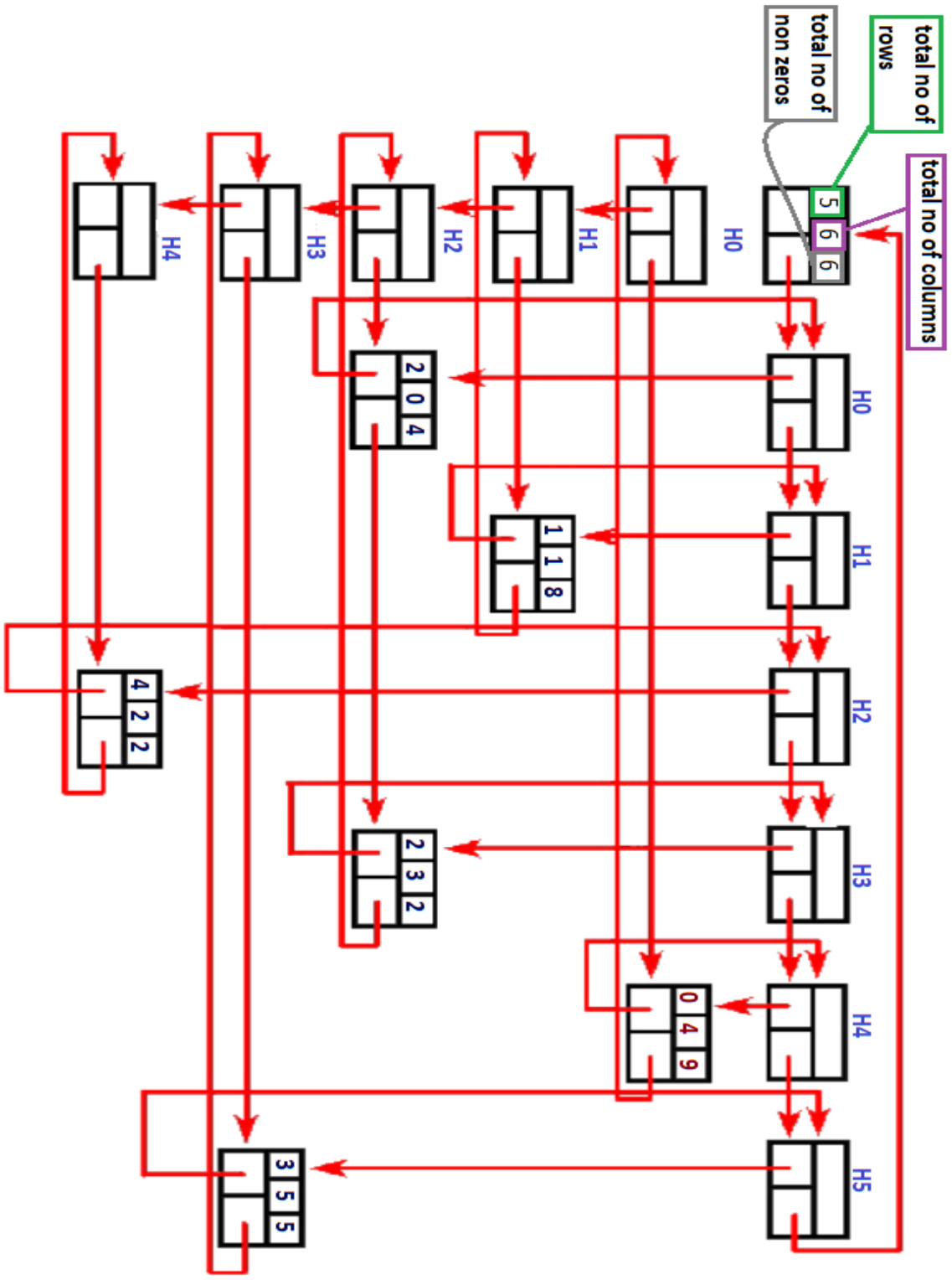
-Here the first row in the right side table is filled with values 5, 6 & 6 which indicates that it is a sparse matrix with 5 rows, 6 columns & 6 non-zero values.
-The second row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix and so on.

- **Linked list representation**:- In this linked list, we use two different nodes namely **header node/index node** and **element node**.
  - Header node consists of three fields and element node consists of five fields as shown in the image…

-Consider the sparse matrix used in the Triplet/array representation.



-This sparse matrix can be represented using linked representation as shown in the below image.



-The very first node which is used to represent abstract information of the sparse matrix (i.e., It is a matrix of 5 X 6 with 6 non-zero elements).

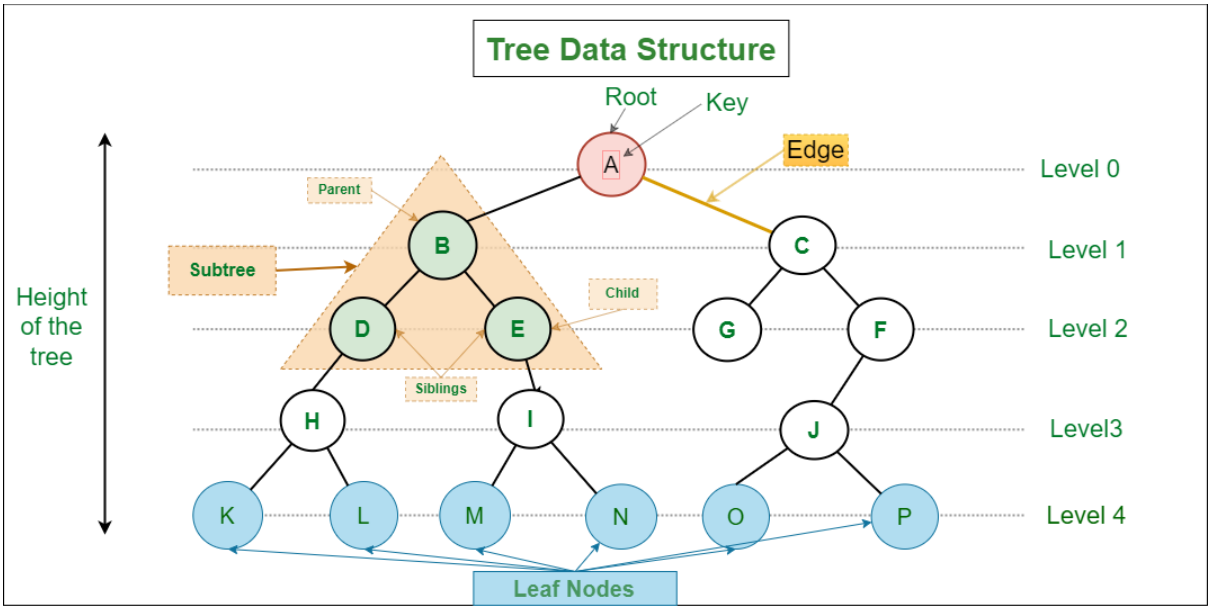
-In this representation, in each row and column, the last node right field points to its respective header node.

#Trees:-A tree is also one of the data structures that represent hierarchical data.

-It is a collection of nodes that are connected by edges .

-The topmost node of the tree is called the **root,** and the nodes below it are called the **child** nodes.

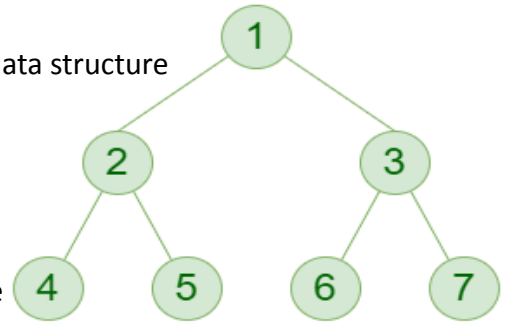
-Each node can have multiple child nodes, and these child nodes can also have their own child nodes.



→Basic Terminologies In Tree Data Structure are;

- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P} are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.

- Subtree: Any node of the tree along with its descendant.

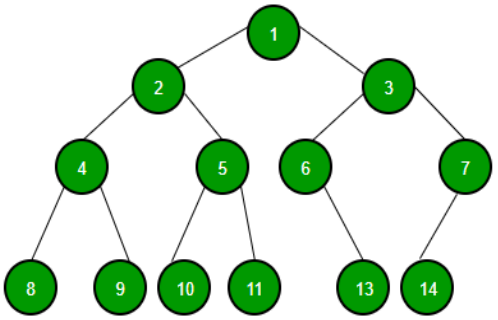


***Binary trees:**-Binary Tree is defined as a tree data structure where each node has at most 2 children.

-Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

-Or The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

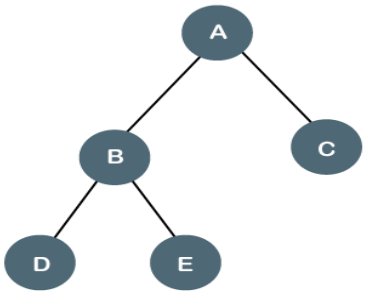
-Example of binary tree (Fig1 and fig2)—---------->



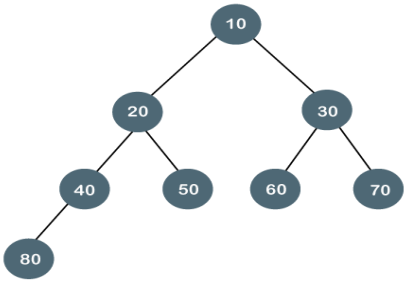
→Types of Binary Tree

There are the types of Binary tree:

1. Full/ proper/ strict Binary tree:-The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children



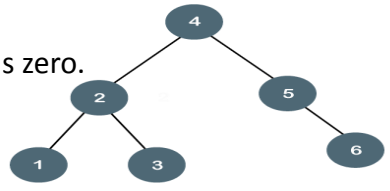
2. Complete Binary Tree:-The complete binary tree is a tree in which all the nodes are completely filled except the last level.



3. Balanced Binary Tree:-The balanced binary tree is a tree in which both the left and right trees differ by atmost 1.

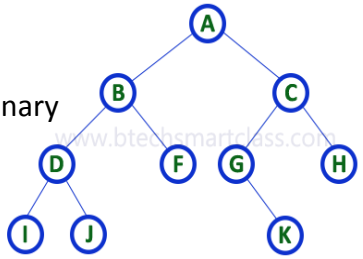
   -For example, AVL and Red-Black trees are balanced binary tree.

   -The given tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



***binary tree representation:**-A binary tree data structure is represented using two methods.

-These are the methods;

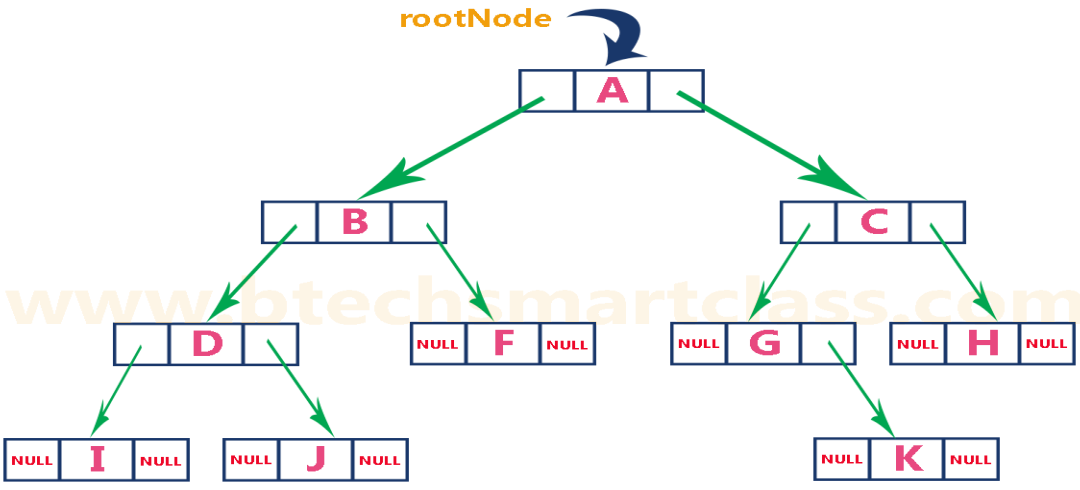


1. **Array Representation:**-In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
   -Consider the above example of a binary tree and it is represented as follows.

   -To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

2. **Linked List Representation:**-We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields.
   -First field for storing left child address, second for storing actual data and third for storing right child address.
   -In this linked list representation, a node has the following structure…

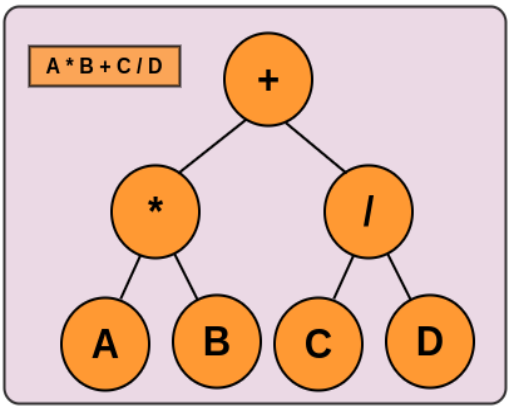
   -The above example of the binary tree represented using Linked list representation is shown as follows…



***algebraic expressions:**-algebraic Expression trees are used to express a mathematical expression in the form of a binary tree.

-Algebraic Expression trees are binary trees in which each internal (non-leaf) node is an operator and each leaf node is an operand.

-For example, if we have an expression A * B + C / D.



***binary tree traversals:**- Traversal is a process to visit all the nodes of a tree and may print their values too.

-Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.

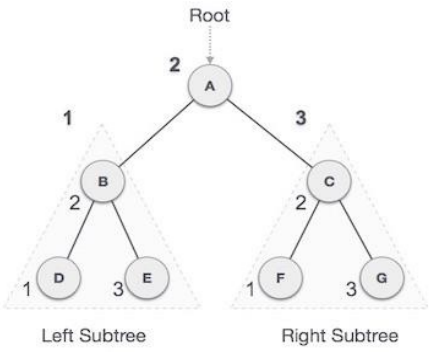
-There are three ways which we use to traverse a tree

1. In - Order Traversal:-In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.

   -We should always remember that every node may represent a subtree itself.

   D ⟶ B ⟶ E ⟶ A ⟶ F ⟶ C ⟶ G

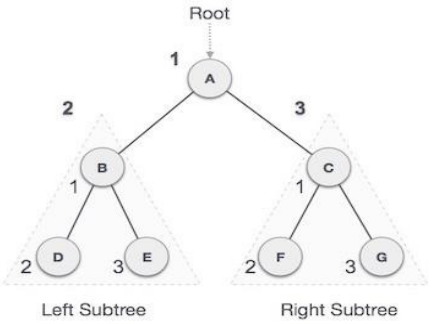
   [L-N-R]



2. Pre - Order Traversal:-In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

   A ⟶ B ⟶ D ⟶ E ⟶ C ⟶ F ⟶ G

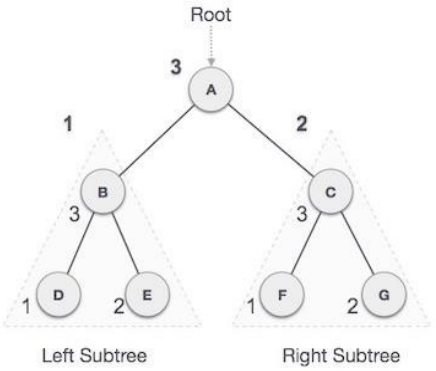
   [N-L-R]



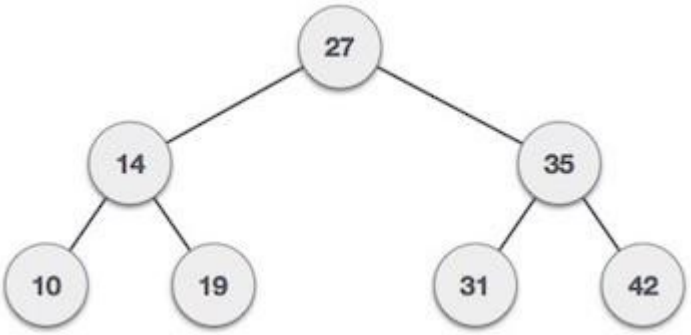
3. Post - Order Traversal:-In this traversal method, the root node is visited last, hence the name.
   -First we traverse the left subtree, then the right subtree and finally the root node.

D → E → B → F → G → C → A

[L-R-N]



-Example



- Preorder traversal: 27 14 10 19 35 31 42
- Inorder traversal: 10 14 19 27 31 35 42
- Post order traversal: 10 19 14 31 42 35 27

→Algorithm of pre-order:-

PREORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. The algorithm does a preorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Initially push NULL onto STACK, and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Apply PROCESS to INFO[PTR].
4. [Right child?]
   If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
   [End of If structure.]

5. [Left child?]
   If LEFT[PTR] ≠ NULL, then:
      Set PTR := LEFT[PTR].
   Else: [Pop from STACK.]
      Set PTR := STACK[TOP] and TOP := TOP – 1.
   [End of If structure.]
[End of Step 2 loop.]
6. Exit.

→**Algorithm of in-order:-**

INORD(INFO, LEFT, RIGHT, ROOT)
A binary tree is in memory. This algorithm does an inorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]
   (a) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
   (b) Set PTR := LEFT[PTR]. [Updates PTR.]

   [End of loop.]
3. Set PTR := STACK[TOP] and TOP := TOP – 1. [Pops node from STACK.]
4. Repeat Steps 5 to 7 while PTR ≠ NULL: [Backtracking.]
5. Apply PROCESS to INFO[PTR].

6. [Right child?] If RIGHT[PTR] ≠ NULL, then:
   (a) Set PTR := RIGHT[PTR].
   (b) Go to Step 3.
   [End of If structure.]
7. Set PTR := STACK[TOP] and TOP := TOP –1. [Pops node.]
   [End of Step 4 loop.]
8. Exit.

→**Algorithm of post-order:-**

POSTORD(INFO, LEFT, RIGHT, ROOT)

A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

1. [Push NULL onto STACK and initialize PTR.]
   Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2. [Push left-most path onto STACK.]
   Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Set TOP := TOP + 1 and STACK[TOP] := PTR.
   [Pushes PTR on STACK.]
4. If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
   Set TOP := TOP + 1 and STACK[TOP] := –RIGHT[PTR].
   [End of If structure.]
5. Set PTR := LEFT[PTR]. [Updates pointer PTR.]
   [End of Step 2 loop.]
6. Set PTR := STACK[TOP] and TOP := TOP – 1.
   [Pops node from STACK.]

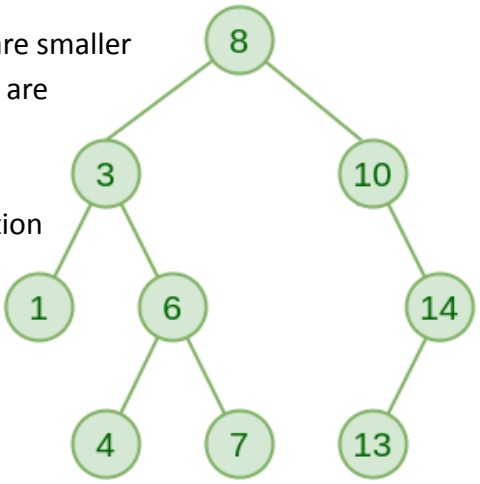
7. **Repeat while PTR > 0:**
   (a) Apply PROCESS to INFO[PTR].
   (b) Set PTR := STACK[TOP] and TOP := TOP – 1.
   [Pops node from STACK.]
   [End of loop.]
8. If PTR < 0, then:
   (a) Set PTR := –PTR.
   (b) Go to Step 2.
   [End of If structure.]
9. Exit.

***Binary Search Tree (BST) :**-A binary search tree follows some order to arrange the elements.

-In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.

-This rule is applied recursively to the left and right subtrees of the root.

►fig:



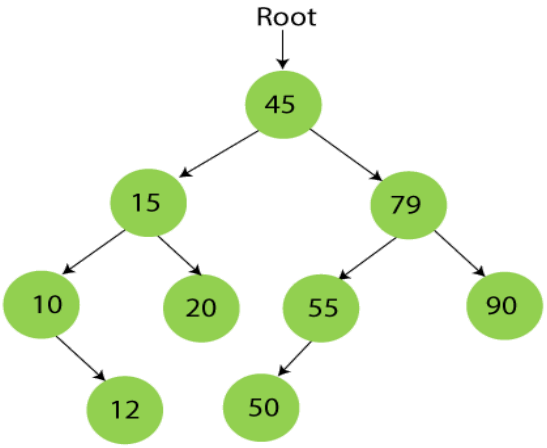
-The root node is 8, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

-Searching an element in the Binary search tree is easy .

-As compared to array and linked lists, insertion and deletion operations are faster in BST.

-Q) Construct a BST using the given values

45, 15, 79, 90, 10, 55, 12, 20, 50.



→Operations perform on the binary search tree are:

1. **Search** :-Searching means to find or locate a specific element or node in a data structure.

   -In Binary search tree, searching a node is easy because elements in BST are stored in a specific order.

   -The steps of searching a node in Binary Search tree are

   - First, compare the element to be searched with the root element of the tree.
   - If root is matched with the target element, then return the node's location.
   - If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
   - If it is larger than the root element, then move to the right subtree.
   - Repeat the above procedure recursively until the match is found.
   - If the element is not found or not present in the tree, then return NULL.

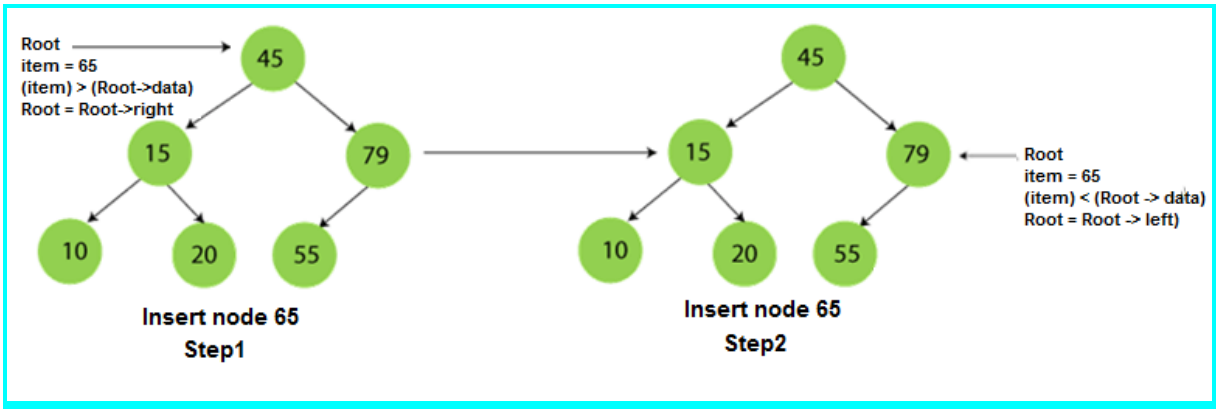
2. **Insert**:-A new key in BST is always inserted at the leaf.

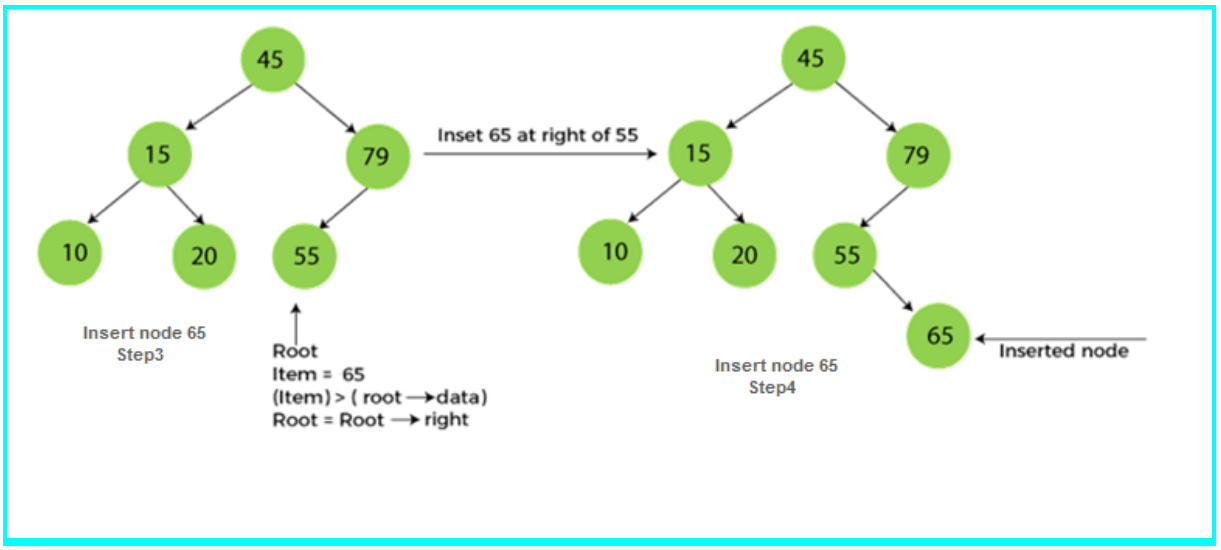
-To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree.

-Else, search for the empty location in the right subtree and insert the data.

-Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

-the process of inserting a node into BST using an example.





3. Delete :- **(syllabus ill padikkan unde e 3 operation um)**

***Balanced Trees:**-A balanced binary tree is also known as height balanced tree.

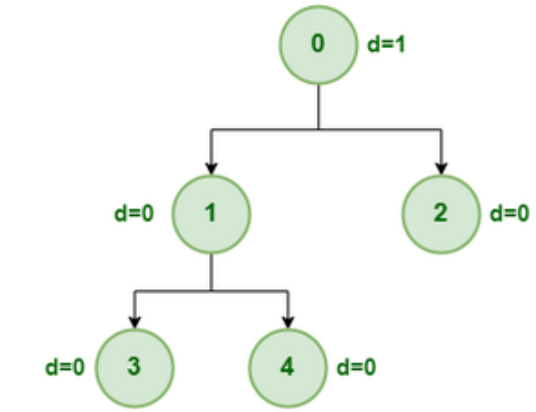
-It is defined as binary tree in when the difference between the height of the left subtree and right subtree is equal to 1.

The height of a tree is the number of edges on the longest path between the root node and the leaf node.

-A balanced binary tree is a binary tree that follows the 3 conditions:

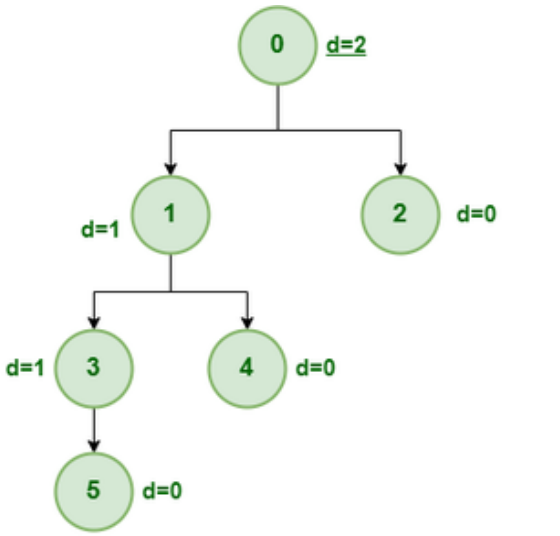
- **The height of the left and right tree for any node does not differ by more than 1.**
- **The left subtree of that node is also balanced.**
- **The right subtree of that node is also balanced.**

-Example:



Balanced Binary Tree with depth at each level indicated

Depth of a node = |height of left child - height of right child|



Unbalanced Binary Tree with depth at each level indicated

Depth of a node = |height of left child - height of right child|

-It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.

- In the figure above, the root node having a value 0 is unbalanced with a depth of 2 units.

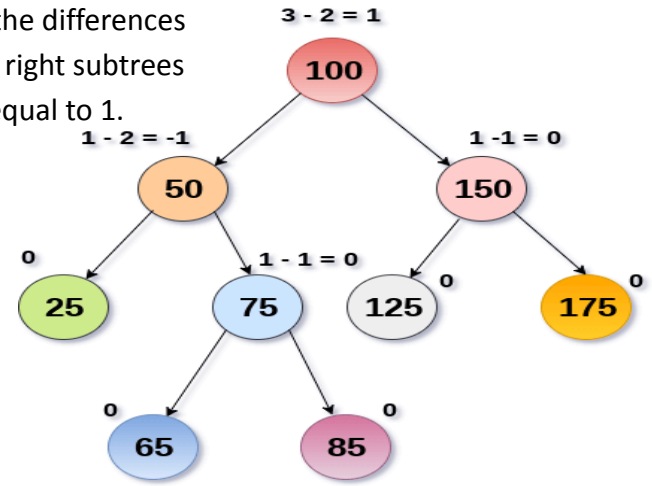
→**Application of Balanced Binary Tree:**

1. **AVL Trees:**-AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one (0,-1,1) for all nodes.

   -The difference between the heights of the left subtree and the right subtree for any node is known as the **balance factor** of the node.

   -Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

   -The above tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.



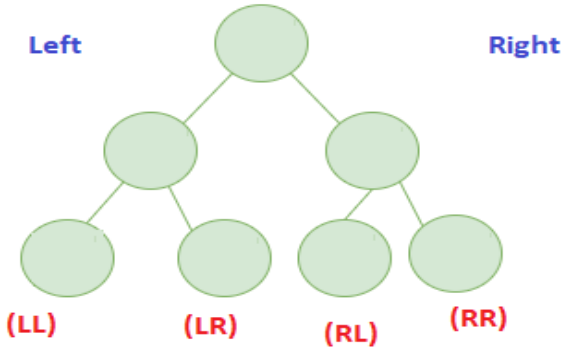
Balance Factor= $h(T^L)$ - $h(T^R)$

► $h(T^L)$=hight of left subtree.
► $h(T^R)$=hight of right subtree.

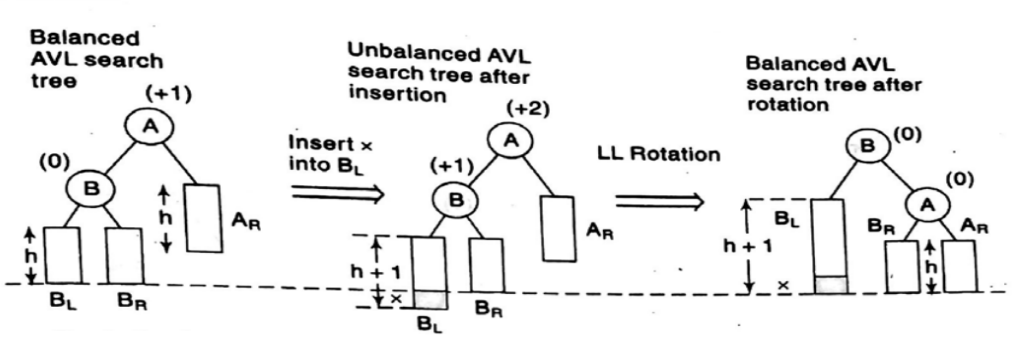
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

→**AVL Rotations:**-After the insertion of element the balance factor of any node in the AVL tree can lead to an unbalanced stage.
-To restore the balance of the BST we use some technique is called rotation.
- There are basically four types of rotations which are as follows:

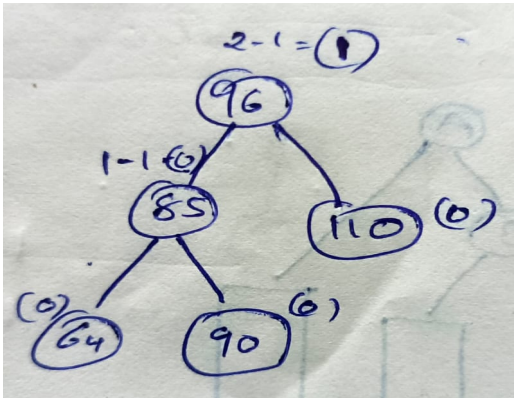
- **L L rotation**: Inserted node is in the left subtree of left subtree of A.
- **R R rotation** : Inserted node is in the right subtree of right subtree of A.
- **L R rotation** : Inserted node is in the right subtree of left subtree of A.
- **R L rotation** : Inserted node is in the left subtree of right subtree of A.



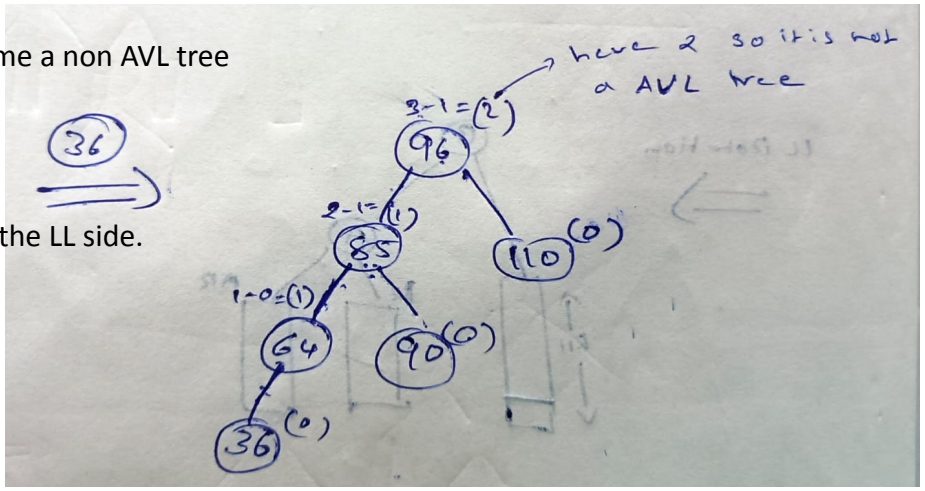
►L L rotation



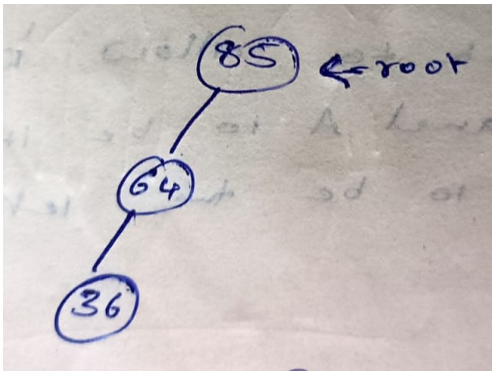
-The new element X is inserted in the left subtree of left subtree of A,
-the closest ancestor node whose BF(A) becomes +2 after insertion.
-To rebalance the search tree, it is rotated so as to allow B to be the root with B_L and A to be its left subtree and right child, and B_R and A_R to be the left and right subtrees of A.
-Eg: given figure is a BST and AVL tree
-All AVL tree are BST



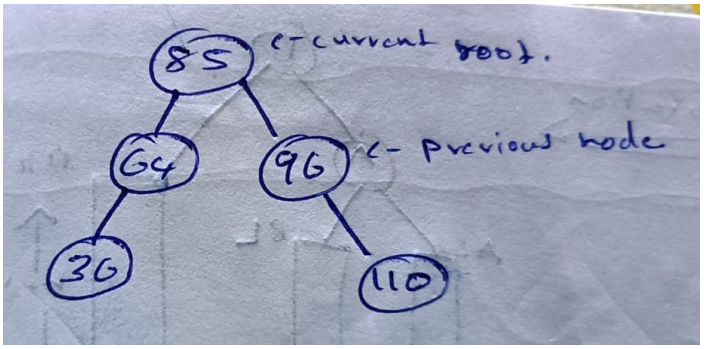
-insert 36.
-After insertion it became a non AVL tree Because of 2.
-To balance we have to apply LL rotation.
-Because we add 36 at the LL side.



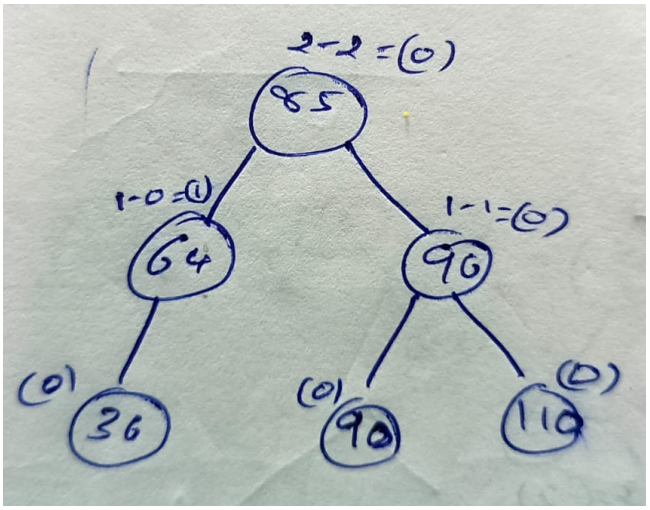
-So set 85 as root and hold left sub tree of 85.



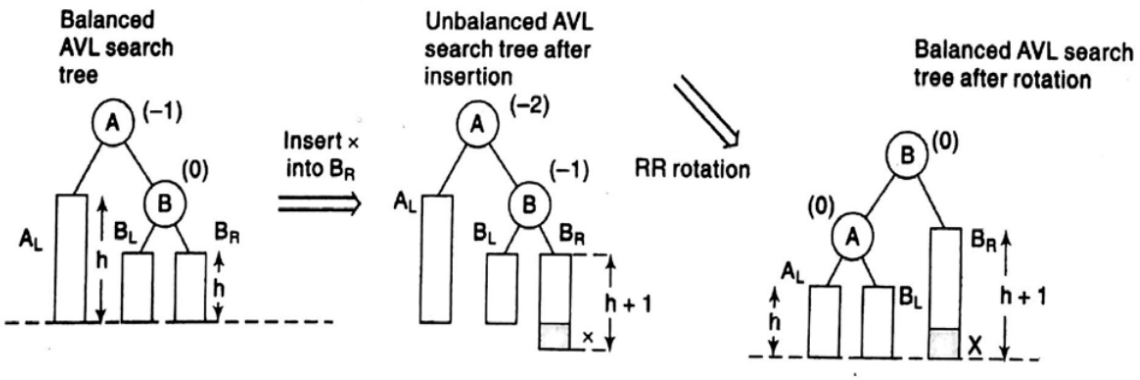
-And add previous root 96 as the right subtree of 85.



-Right sub tree of root 85 is 90 and place 90 at the left sub tree on the previous root 96.



►R R rotation



-Here the new element X is in the right subtree of A.

-The rebalancing rotation pushes B up to the root with A as its left child and B_R as its right subtree, and A_L and B_L as the left and right subtrees of A.

-Eg:

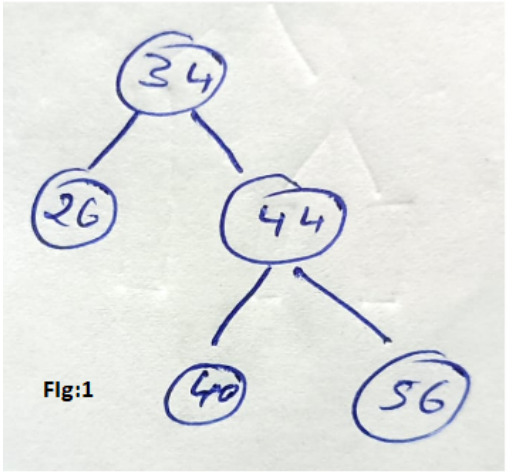


Fig:1

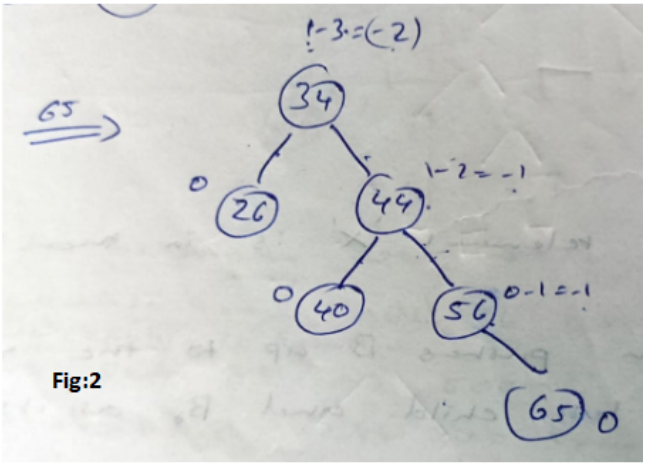


Fig:2

-Fig 1 is a balanced

-The inserted node 65 cause it a unbalanced (fig:2)

-so we apply RR rotation

-because we add 55 at the right right subtree.

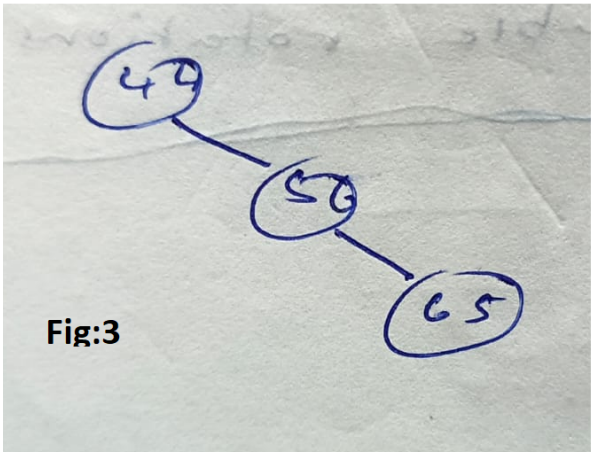


Fig:3

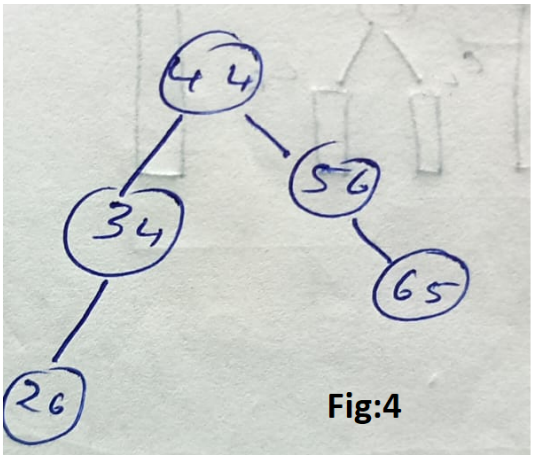


Fig:4

-And we set 44 as root node.

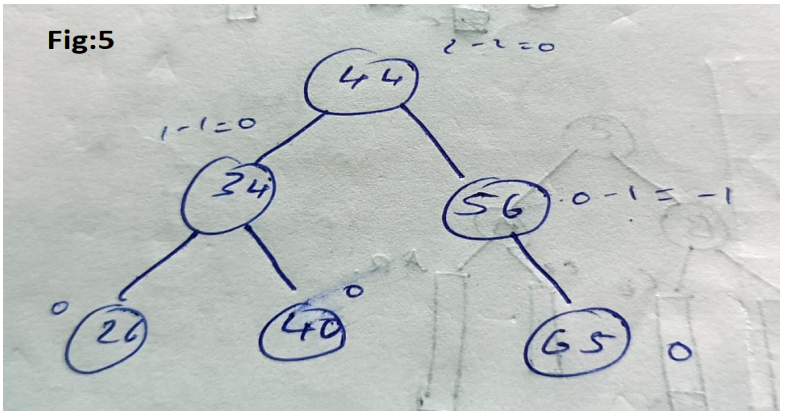
-And hold its right subtree. (fig:3)

-Then add previous root 35 as the left child of the current root 44 (fig:4)

-And add left subtree of 44 to the right of the previous root 34 (fig :5).

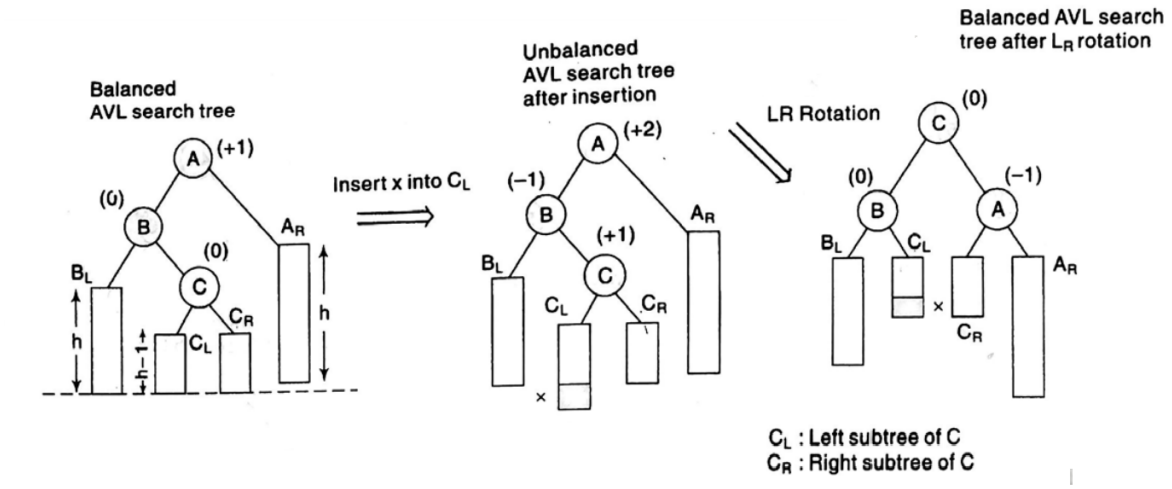
-Know it became balanced.

Fig:5

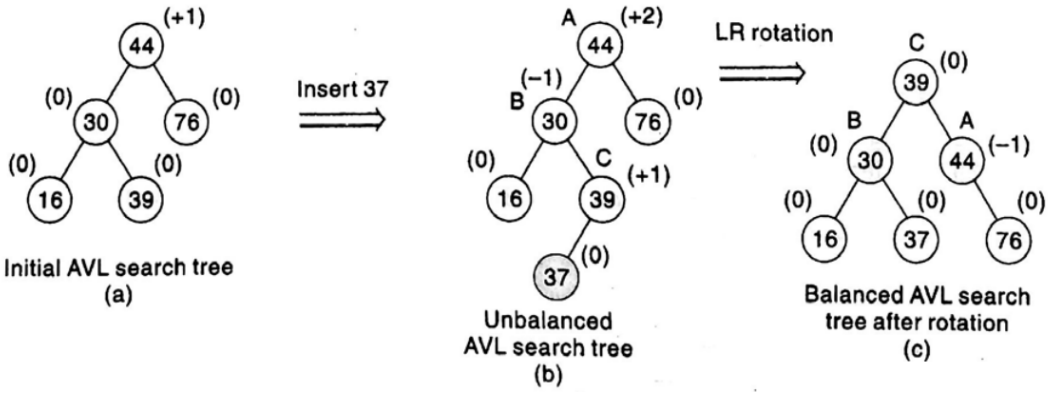


Note: LL and RR rotation are called as single rotation
LR and RL rotation are called double rotation

►**L R rotation** :-LR can be accomplished by RR followed by LL rotation.



Eg:



-Fig a is balanced
-After insertion it become unbalanced.
-We apply LR rotation because we add 37 at the LR side.
-And we se 39 as root.

►**R L rotation:**-RL can be accomplished by LL followed by RR rotation.
-Explain cheyandaa

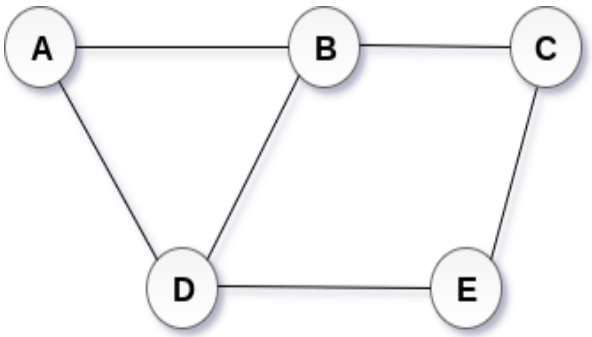
#Graphs:-Graph is a non-linear data structure.

It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs).

-Here edges are used to connect the vertices.

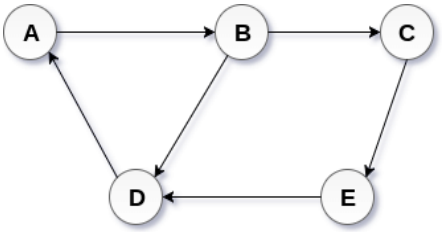
-A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.

Eg:A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.

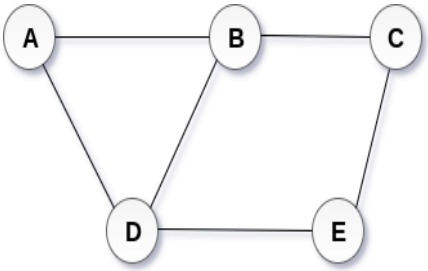


→types of graph

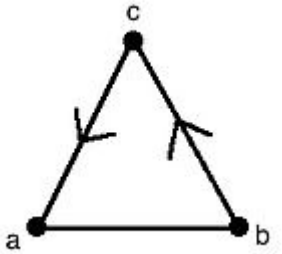
- Directed Graph :-In a directed graph, Edges represent a specific path from some vertex A to another vertex B.
  -Node A is called initial node while node B is called terminal node.
  -A directed graph is shown in the following figure.



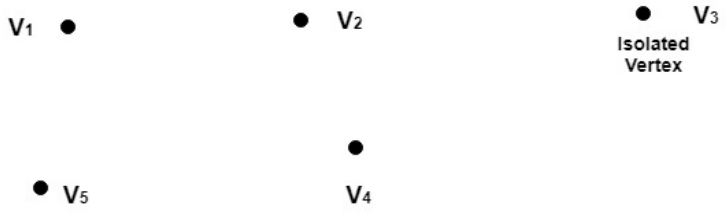
- Undirected Graph:-A graph with only undirected edges is said to be undirected graph.
  -Example



- Mixed Graph:-A graph with both undirected and directed edges is said to be mixed graph.



- Null Graph: A null graph is defined as a graph which consists only the isolated vertices.
  -Example: The graph shown in fig is a null graph, and the vertices are isolated vertices.



→**Graph Terminology**:-

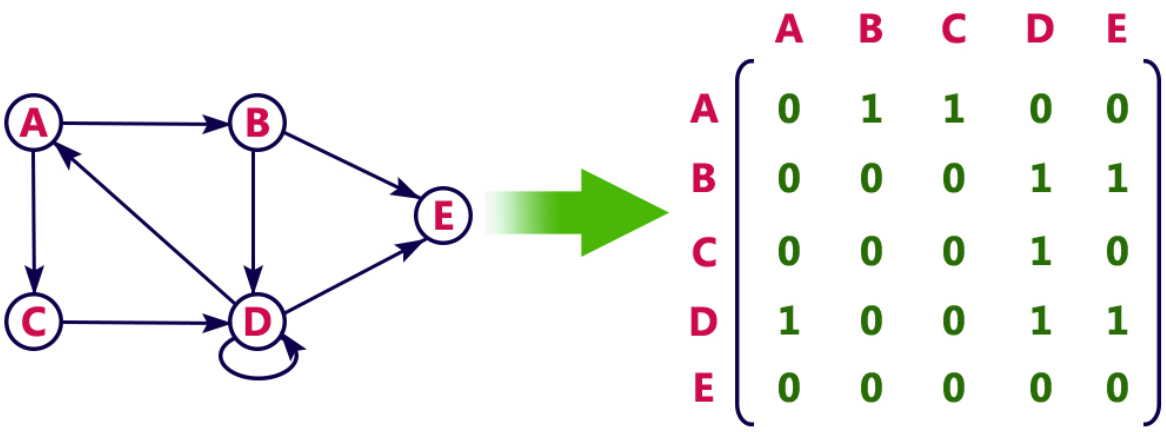
1. **Path:-**A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
2. **Closed Path:-**A path will be called as closed path if the initial node is same as terminal node.
   -A path will be closed path if V0=VN.
3. **Simple Graph:**-A graph is said to be simple if there are no parallel and self-loop edges.
4. **Parallel edges or Multiple edges:-**If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.
5. **Self-loop:-**Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.
6. **Complete Graph:-**A complete graph is the one in which every node is connected with all other nodes.
   -A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.
7. **Weighted Graph:-**In a weighted graph, each edge is assigned with some data such as length or weight.
8. **Adjacent Nodes:-**If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
9. **Degree of the Node:-**A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.
10. **Isolated vertex:** It is the vertex that is not connected to any other vertices in the graph.

→**Representations of Graphs/sequential Representations of Graphs**:-y Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

-A graph is a data structure that consist a sets of vertices (called nodes) and edges.

-There are two ways to store Graphs into the computer's memory:

1. **Sequential representation**:-In sequential representation, an **adjacency matrix or path matrix** is used to store the graph.
   A. Adjacency matrix:-In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices.
   -That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices.
   -This matrix is filled with either 1 or 0.
   -Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.
   -For example:

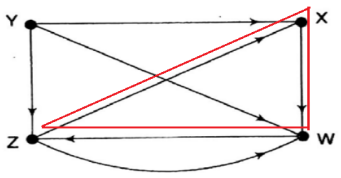


	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

   B. Path matrix:-Let G be a simple directed graph with N nodes v,,v_2,...$V_{N.}$
   -The path matrix or reachability matrix of G is the N*N matrix
   where P= (P_{ij}) defined as follows:

$$P_{ij} = \begin{cases} 1 & \text{if there is a path from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

-Eg:



$$P = \begin{array}{c} \\ x \\ y \\ z \\ w \end{array} \begin{array}{c} \begin{array}{cccc} X & Y & Z & W \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \end{array}$$

-here x can reach x by travelling through x→w→z→x.
-And we mark 1 at the matrix.
-Travelling is according to direction.

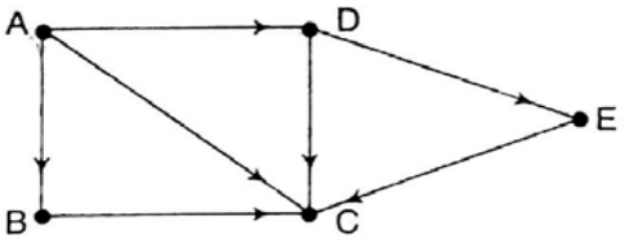
2. **Linked list representation**:- In linked list representation, there is a use of an **adjacency list** to store the graph.
   -The linked representation contain two lists they are; Node list and Edge list
   - Node list:-Each element in the Node list is corresponds to the nodes in the graph.
     -And its recording form is

Node	Next	Adjacent	

     Node:-It is the name or key value of the node.
     Next:-It will be pointed to the next node in the list.
     Adjacent:-It points to the first elements in the adjacency list.
   - Edge list:-Each element in the edge list is corresponds to an edge in the graph.
     -And its recording form is

Destination	Link

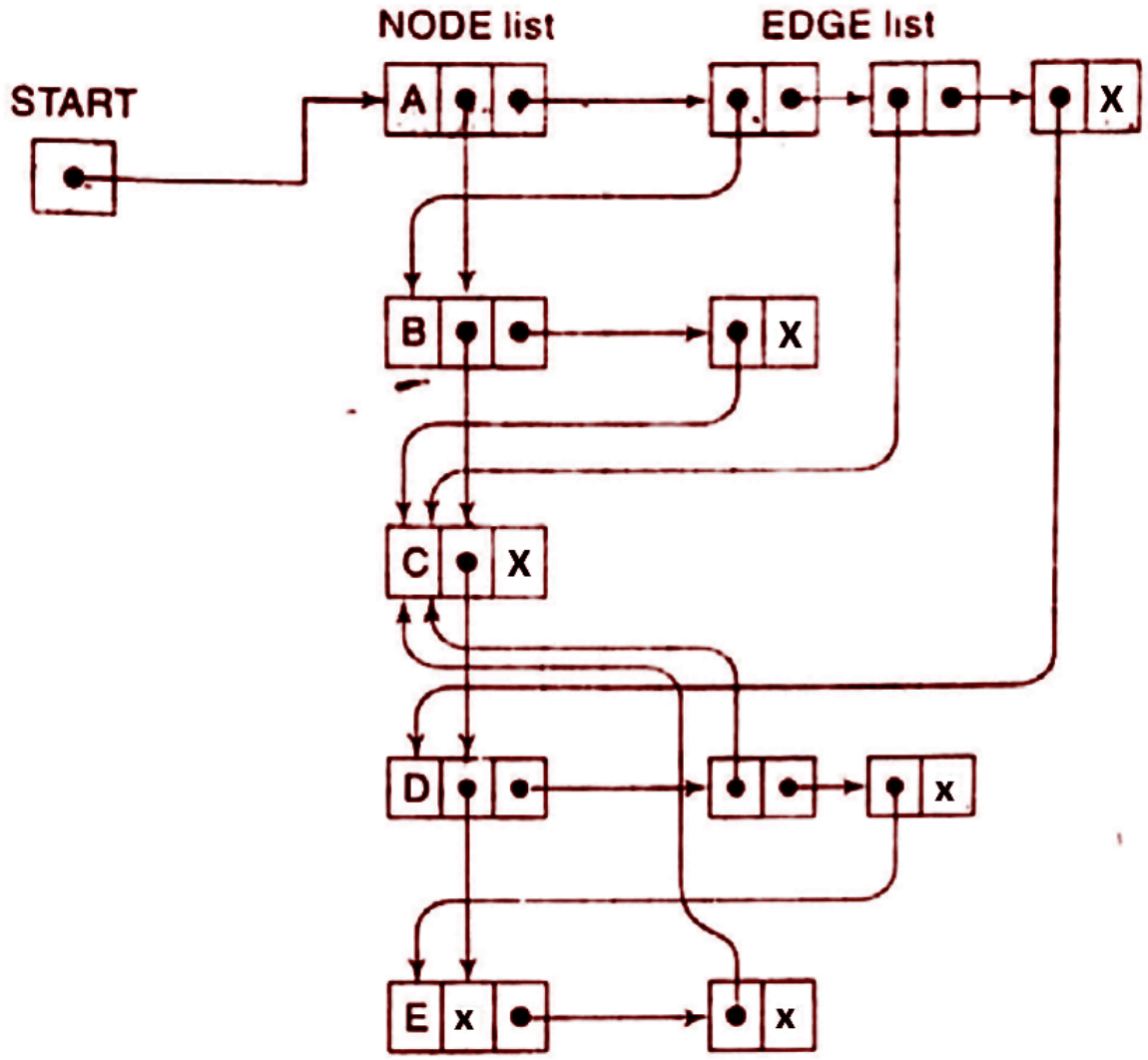
     →Adjacency list:-Consider the graph and the table shows the adjacency of the node.



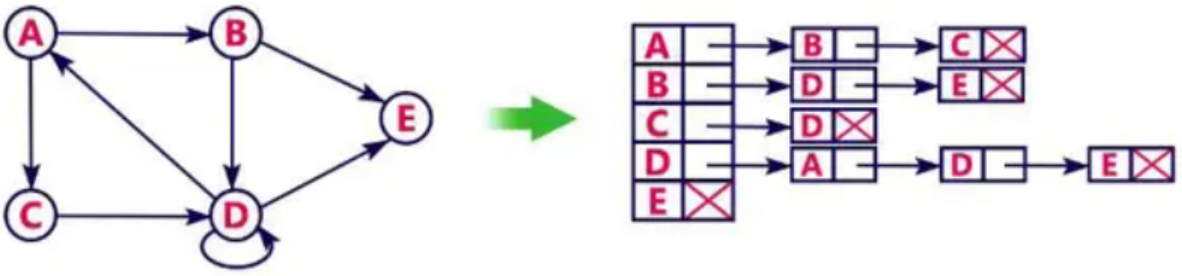
(a) Graph *G*

Node	Adjacency List
A	B, C, D
B	C
C	
D	C, E
E	C

(b) Adjacency list of *G*


NODE list
EDGE list
START
A
X
B
X
C
X
D
x
E
x
x

-Example 2:


A
B
E
C
D
A
B
C
D
E
B
C
D
E
D
A
D
E

***Graph Traversals:**-Graph traversal is a technique used to search for a vertex in a graph.

-The graph has two types of traversal they are;.

1. **Breadth First Search (BFS)**:-The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph.

   -In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one.

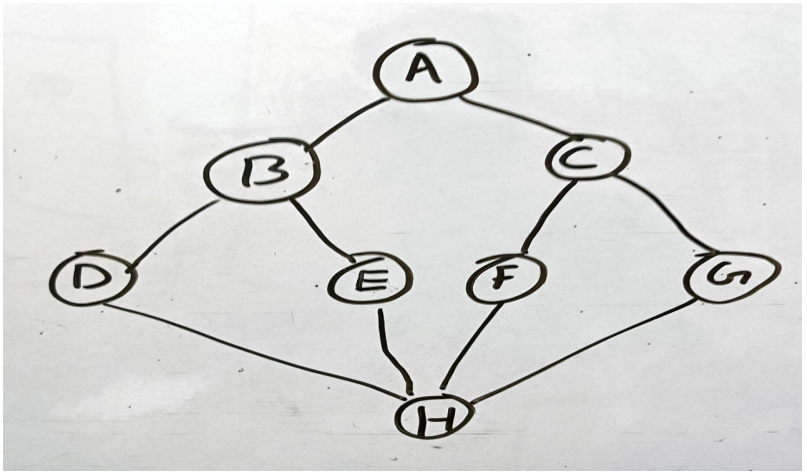
   -After completing all of the adjacent vertices, it moves further to check another vertices and checks its adjacent vertices again.

   →Algorithm of BFS

   - Step 1:-
   - Step 2:-visit its adjacent unvisited node.
   - Step 3:-mark it as visited and display it.
   - Step 4:-insert the visited node into the queue.
   - Step 5:-if there are no adjacent node then remove first node from the queue.
   - Step 6:-repeat the above step until the queue is empty.

     And exit.

   -Example:



	A	B	C	D	E	F	G	H
Array:								

Queue:								

   -At start queue is empty.

   -Know let us start from node A once we visited node A.

   -we mark it as visited.

   -And placed it in a queue and mark 1 in the array means visited.

   (0 indicate unvisited)

	A	B	C	D	E	F	G	H
Array:	1	0	0	0	0	0	0	0

Queue:	A							

-Visit the adjacent node and mark it as visited.
-Here B and C is the adjacent node of A.
-So place B and C in queue and mark 1 in the array .

	A	B	C	D	E	F	G	H
Array:	1	1	1	0	0	0	0	0

Queue:	A	B	C					

-Here node A have no adjacent node unvisited so remove the node A from the queue.
-because we visited all the adjacent node of A .
-And display node A in the output.

	A	B	C	D	E	F	G	H
Array:	1	1	1	0	0	0	0	0

Queue:		B	C					

Output: A

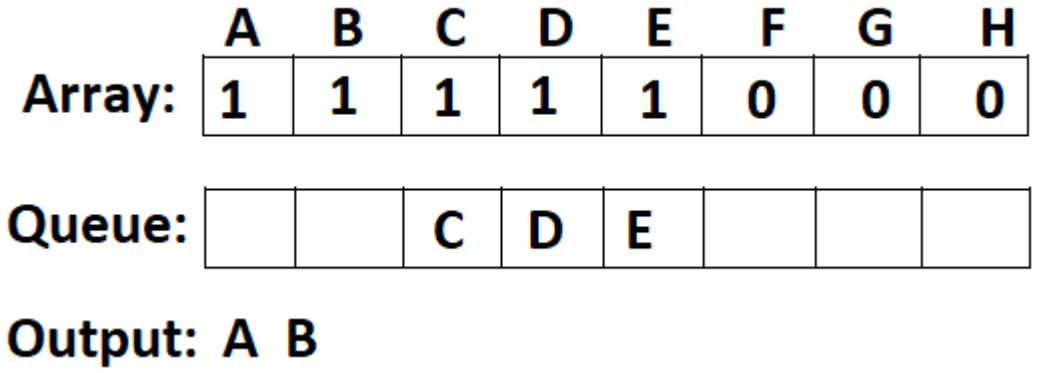
-Here next element in the queue is B so we take adjacent node of B.
-Adjacent node of B is D and E so place it in the queue.
-And mark 1 for visiting D and E in the array.

	A	B	C	D	E	F	G	H
Array:	1	1	1	1	1	0	0	0

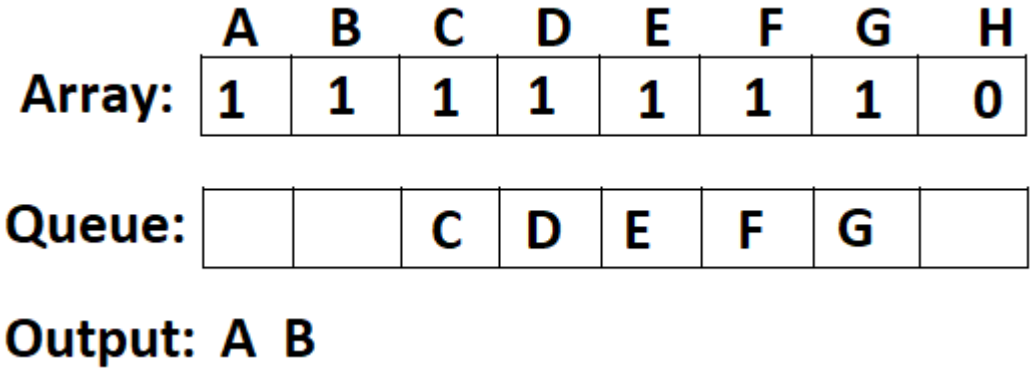
Queue:		B	C	D	E			

Output: A

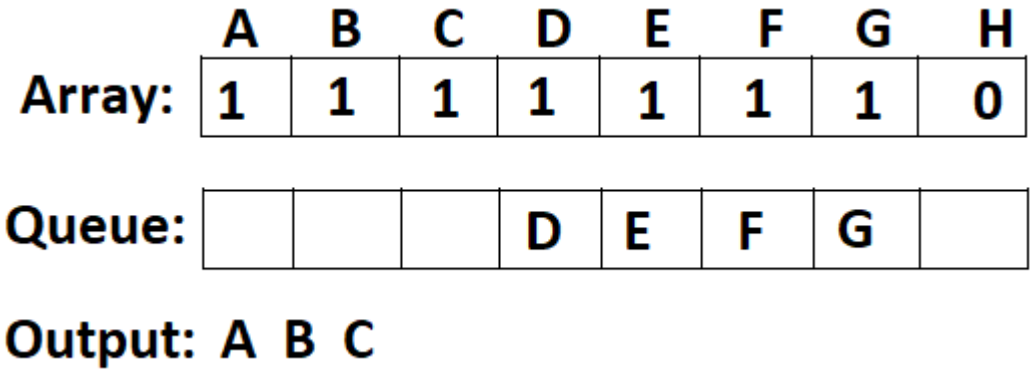
-And take B from the queue because there is no remaining adjacent node to visit.
-And place it in the output section.



-Here next element in the queue is C so we take adjacent node of C.
-Adjacent node of C is F and G so place it in the queue.
-And mark 1 for visiting F and G in the array.

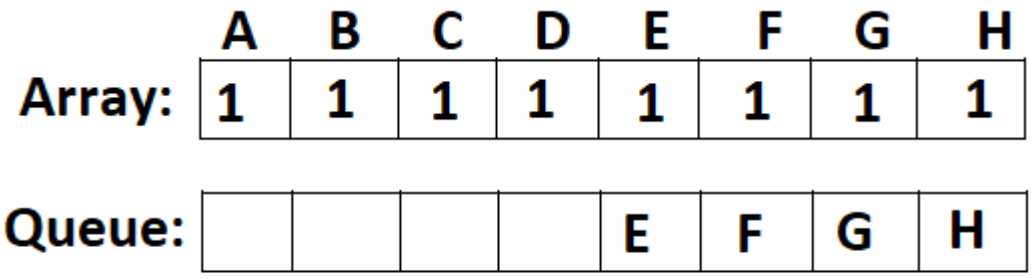


-Here node C have no more unvisited adjacent node therefore remove C from the queue and add it to output.



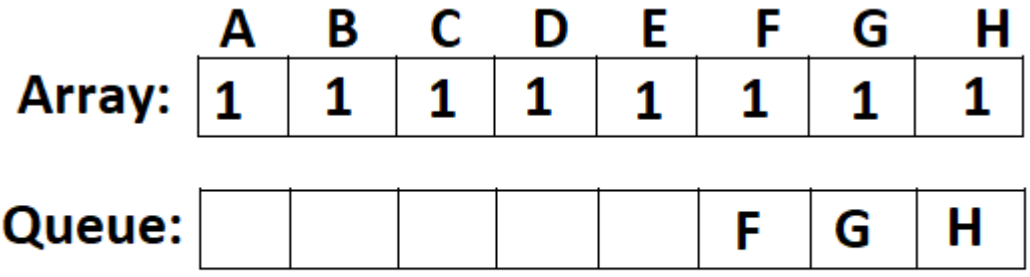
--Here next element in the queue is D so we take adjacent node of D.
-Adjacent node of C is H ,so place it in the queue.
-And mark 1 for visiting H in the array.

-And there is no more unvisited adjacent node in D therefore remove D from the queue and add it to output.



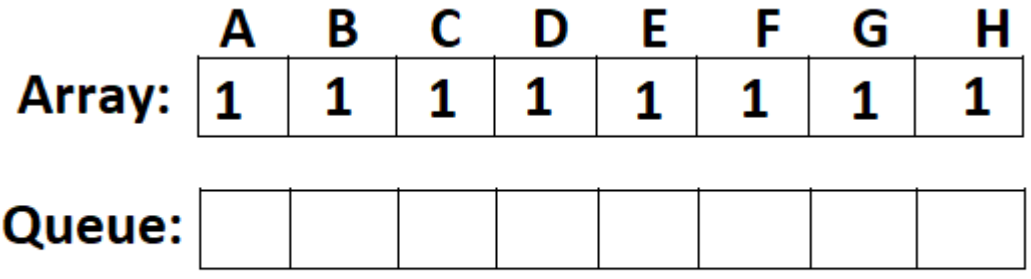
Output: A B C D

-Next element in the queue is E and its adjacent node is already visited and there is no more unvisited adjacent node remain.
-so remove node E from queue and add it to output.



Output: A B C D E

-Next element is F,G and H and there is no more unvisited adjacent node remains to this node.
-so remove it from queue and add it to output



Output: A B C D E F G H

-When queue is empty then exit.
-So the output is

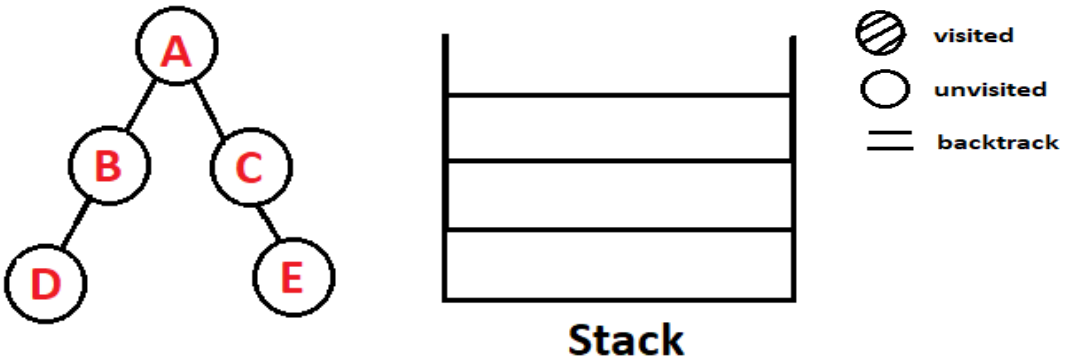
Output: A B C D E F G H

2. **Depth First Search (DFS)**:-The Depth First Search (DFS) is a graph traversal algorithm.
   -In this algorithm one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and try to traverse in the same manner.

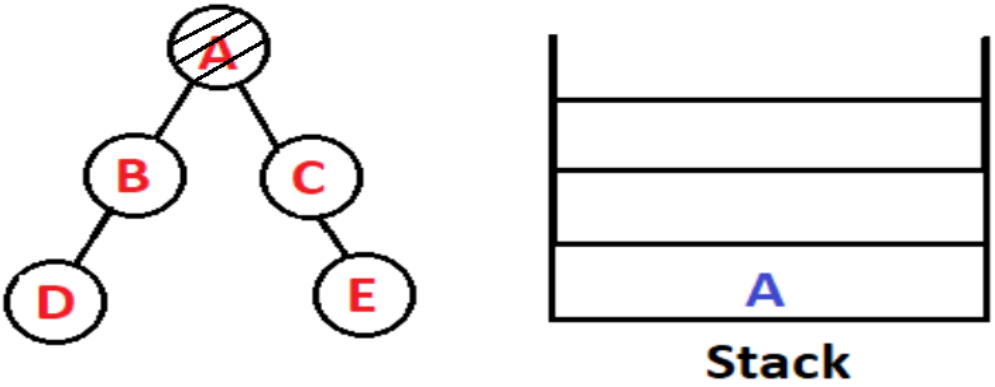
   →Algorithm for DFS

   - Step 1 :- Mark the starting node as visited.
   - Step 2 :-Examine the adjacent node of the starting node.
   - Step 3 :-choose an unvisited adjacent node and visit it.
   - Step 4 :-repeat step 2 and 3 for newly visited node.
   - Step 5 :-if there is no unvisited adjacent node.
     -Then back track to the previous node and repeat step 2 and step 3.
   - Step 6 :-The algorithm terminated when all the nodes are visited And exit.

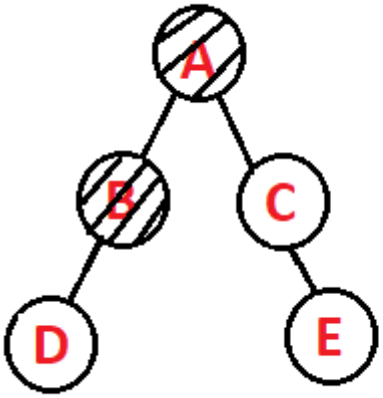
   -Example:

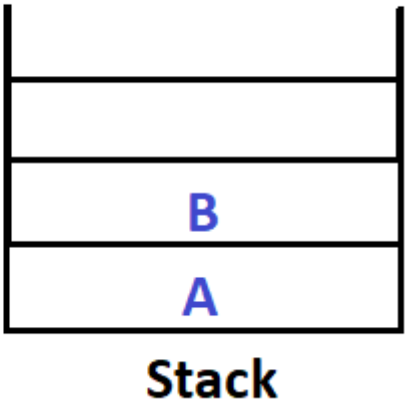


   -Mark node A as visited and push it on the stack.

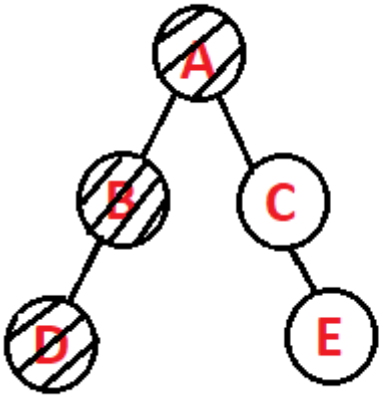


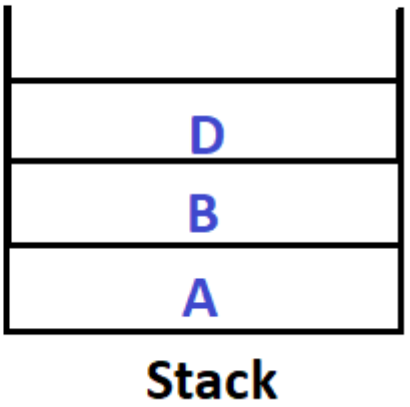
-Process the child of node A and mark node B as visited and push it on the stack.



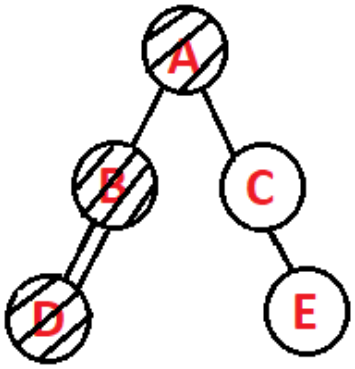


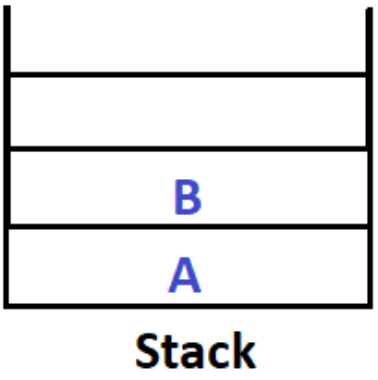
-Process the children of node B and mark node D as visited and push it on the stack.

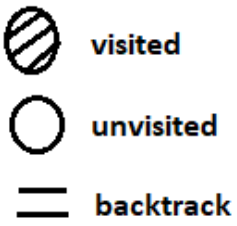




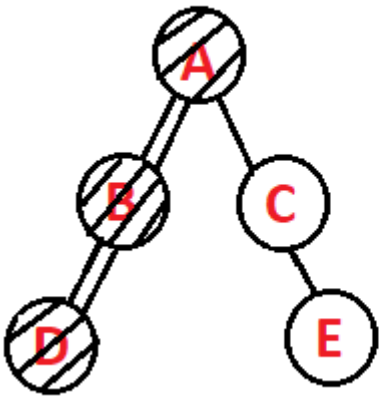
-Pop node from stack and process it because node D have no adjacent node.
-and we backtrack by popping it from stack.

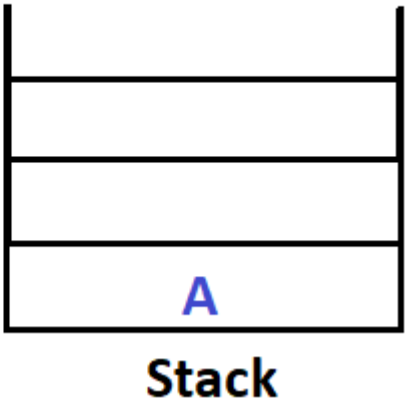




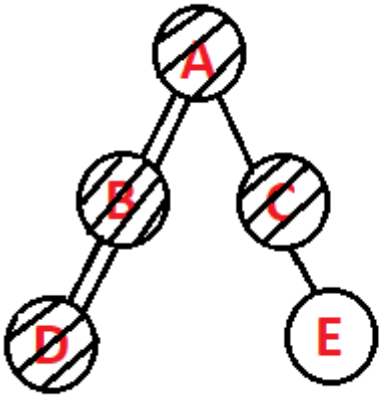


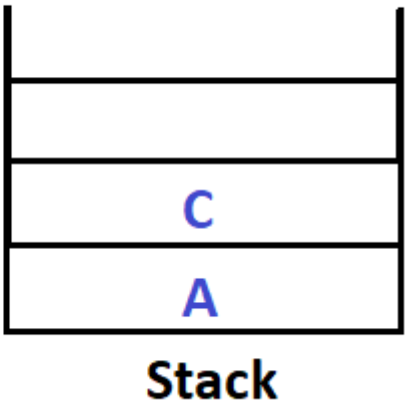
-Read node B from stack and process it.
-Since node B have no other adjacent node we backtrack by popping it from stack.



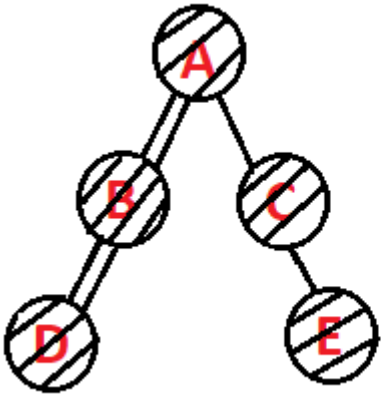


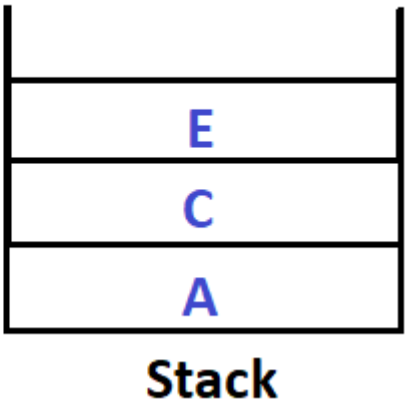
-Read the top node from the stack that is A and process it.
-Here node A have one unvisited adjacent node that is C.
-So push node C into the stack.





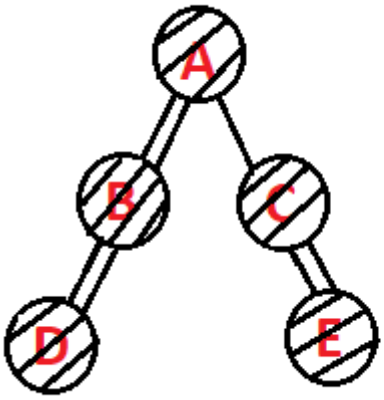
-Process node C in the stack.
-And here node C have a adjacent node E to visit.
-So we push Node E to the stack.

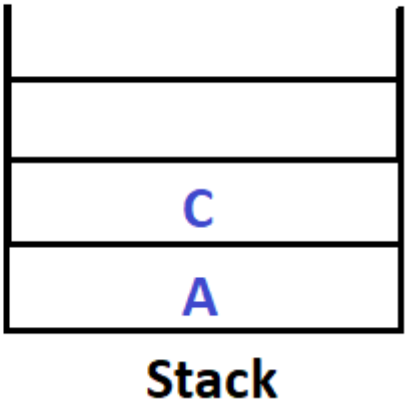




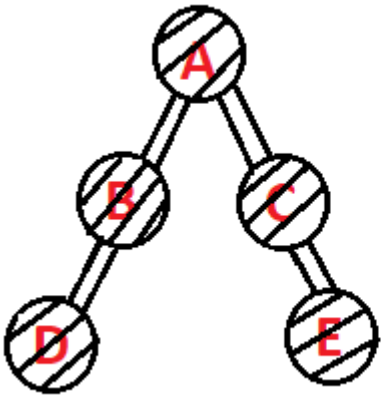
-Read E from stack and process it.

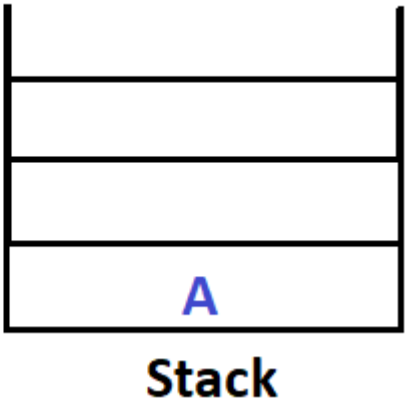
-Since E has no adjacent node to visit.
-Therefore we backtrack by popping it from stack.



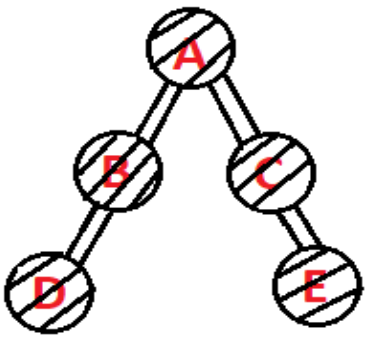


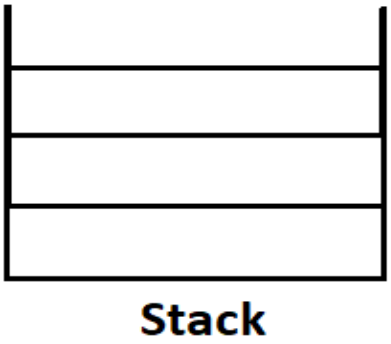
-Read the node C from stack and process it.
-Since C has no adjacent node to visit.
-Therefore we backtrack by popping it from stack.





--Read the node A from stack and process it.
-Since all the children of the node A is visited.
-Therefore we backtrack by popping it from stack.
-And know the stack is empty.



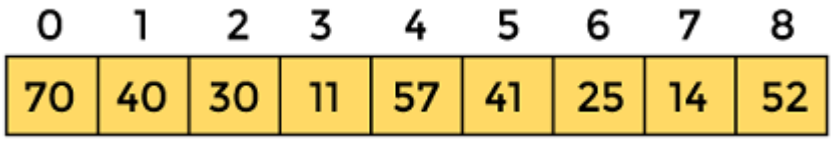


-Know all the node in the graph have been visited.
-so exit.

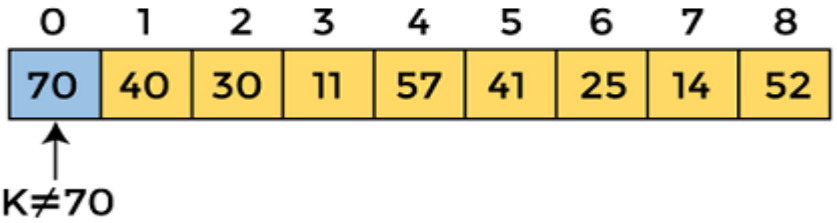
Module -3-Searching and Sorting

#Searching:-Searching in data structure refers to the process of finding the required information from a collection of items.
-Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.
-These algorithms are classified according to the type of search operation they perform, such as:

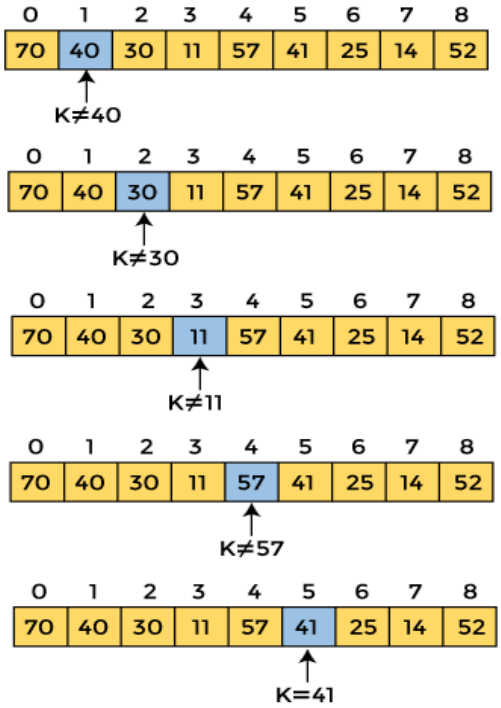
1. **Linear search/Sequential Search**:-Linear search is also called as sequential search algorithm.
   -It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
   -If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
   -It is widely used to search an element from the unordered list.
   →Working:-let's take an unsorted array.
   -The elements of array are -



   -Let the element to be searched is K = 41
   -Now, start from the first element and compare K with each element of the array.



   -The value of K, i.e., 41, is not matched with the first element of the array.
   -So, move to the next element. And follow the same process until the respective element is found.



-Now, the element to be searched is found. So algorithm will return the index of the element matched.

→ Algorithm:

- Assume we need to find an element that is in an array in random order.
- Start with the first position and compare it to the target in order to search for a target element.
- If the current element matches the target element, return the position of the current element.
- If not, move on to the next one until we reach the very end of an array.
- If still unable to find the target, return-1

2. **Binary Search**:-The Binary search method is **only suitable for searching in a sorted array**.

   -In this method, the element that has to be searched is compared to the array's middle element.

   -Search is considered successful only if it matches the target.

   -The binary search algorithm uses the **divide-and-conquer approach**, it does not scan every element in the list, it only searches half of the list instead of going through each element,

   -Hence said to be the best searching algorithm because it is faster to execute as compared to Linear search.

   →Working:-searching an element 40 in the 13-element array, following figure shows how binary search works:

   11 22 30 33 40 44 55 60 66 77 80 88 99.

upper

11 22 30 33 40 44 55 60 66 77, 80, 88, 99

1 2 3 4 5 6 7 8 9 10 11 12 13

Item is search to be 40?

1 - lower

13 - upper

$mid = \frac{upper + lower}{2}$

1	2	3	4	5	6	7
11	22	30	33	40	44	55

$\frac{13+1}{2} = 7$

mid = 7

ATA[7] = 55

∴ 40 < 50, so minus 1 from upper

1 = lower

upper = 7 - 1 = 6.

$mid = \frac{1+6}{2} = 3.5$

∴ 3

1	2	3	4	5	6
11	22	30	33	40	44

DATA[3] = 30.

30 > 40

so add 1 to lower.

lower = 3+1 = 4.

upper = 6.

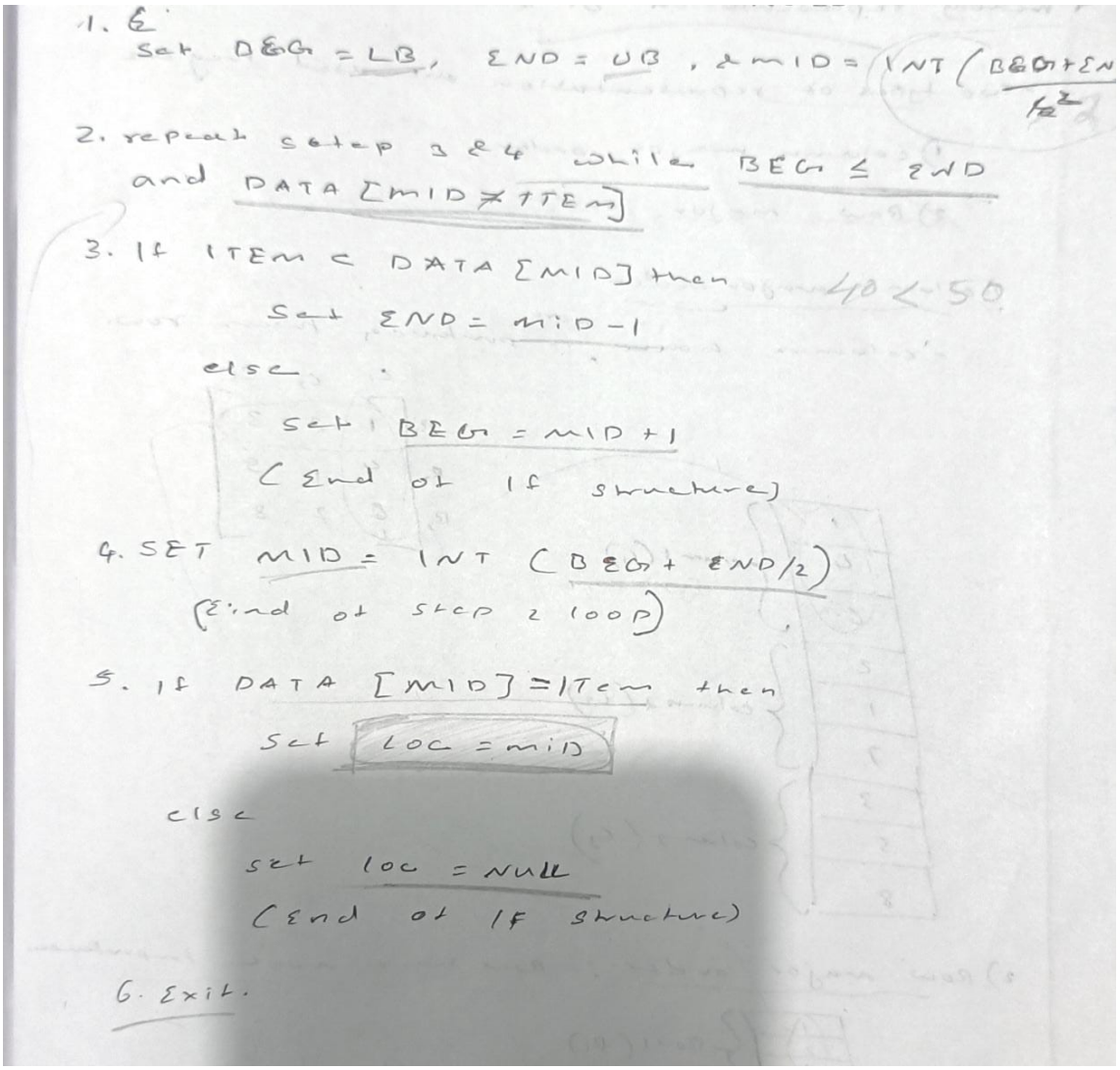
$mid = \frac{4+6}{2} = \frac{10}{2} = 5$

DATA[5] = 40

Item present at the 5th position

4	5	6
33	40	44

→Algorithm:



→Comparison of Linear search and Binary search

Linear Search	Binary Search
Commonly known as sequential search.	Commonly known as half-interval search.
Elements are searched in a sequential manner (one by one).	Elements are searched using the divide-and-conquer approach.
The elements in the array can be in random order.	Elements in the array need to be in sorted order.
Less Complex to implement.	More Complex to implement.
Linear search is a slow process.	Binary search is comparatively faster.
Single and Multidimensional arrays can be used.	Only single dimensional array can be used.
Does not Efficient for larger arrays.	Efficient for larger arrays.
The worst-case time complexity is O(n).	The worst case time complexity is O(log n).

→**complexities of Linear search and binary search (3 marks)**:-

- Linear search:
  - ➢ Space Complexity:-Space complexity for linear search is **O(n)** as it does not use any extra space where n is the number of elements in an array.
  - ➢ Time Complexity:-
    *Best- case complexity = O(1) occurs when the search element is present at the first element in the search array.
    *Worst- case complexity = O(n) occurs when the search element is not present in the set of elements or array.
    *Average complexity = O(n) is referred to when the element is present somewhere in the search array.

- Binary search:
  - ➢ Space Complexity :-The space complexity of binary search is **O(1)**.
  - ➢ Time Complexity:
    *Best Case Complexity - In Binary search, best case occurs when the element to search is found in first comparison, i.e., when the first middle element itself is the element to be searched. The best-case time complexity of Binary search is **O(1)**.
    *Average Case Complexity - The average case time complexity of Binary search is **O(logn)**.
    *Worst Case Complexity - In Binary search, the worst case occurs, when we have to keep reducing the search space till it has only one element. The worst-case time complexity of Binary search is **O(logn)**.

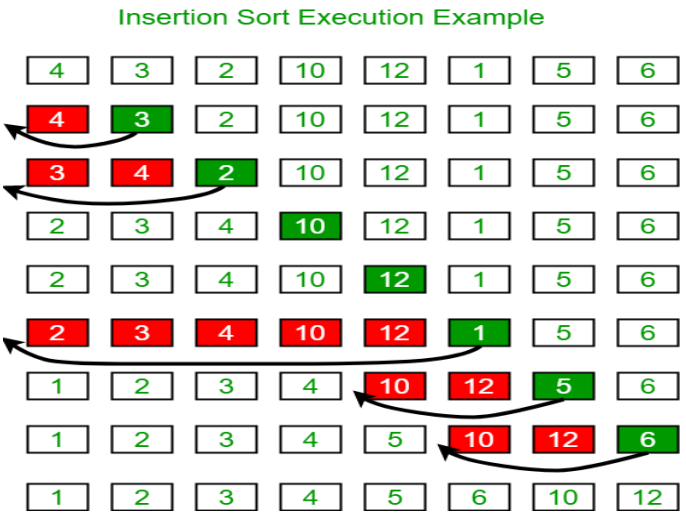
#Sorting:-Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
-For example, consider an array A = {A1, A2, A3, A4, ?? An }, the array is called to be in ascending order if element of A are arranged like A1 > A2 > A3 > A4 > A5 > ? > An .
Eg:int A[10] = { 5, 4, 10, 2, 30, 45, 34, 14, 18, 9)
-The Array sorted in ascending order will be given as;

A[] = { 2, 4, 5, 9, 10, 14, 18, 30, 34, 45 }

-There are many techniques by using which, sorting can be performed they are;

1. **Insertion Sort**:-insertion sort is a simple sorting technique.
   -Insertion sort works similar to the sorting of playing cards in hands.
   -It is assumed that the first card is already sorted in the card game, and then we select an unsorted card.
   -If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side.
   -Similarly, all unsorted cards are taken and put in their exact place.

-The same approach is applied in insertion sort.



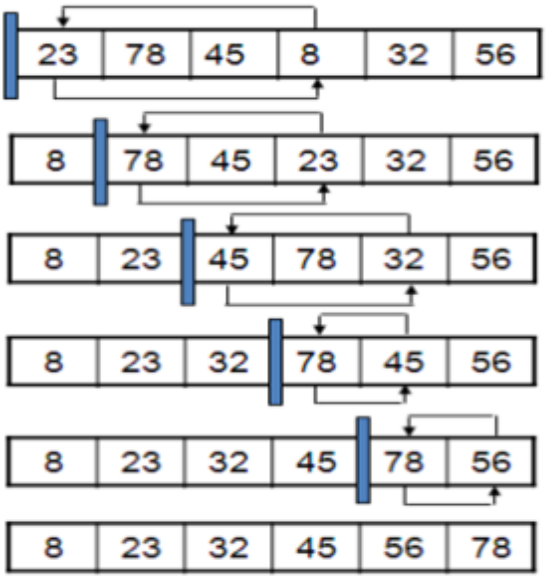
>Time Complexity of Insertion Sort:

- ➢ Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of insertion sort is O(n).
- ➢ Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of insertion sort is $O(n^2)$.
- ➢ Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is $O(n^2)$.
  -where n is the number of items.

>space complexity of Insertion Sort is O(1), It is because, in insertion sort, an extra variable is required for swapping.

2. **Selection sort**:-The working procedure of selection sort is also simple.
   -In selection sort the array is divided into two parts, first is sorted part, and another one is the unsorted part.
   -Initially, the sorted part of the array is empty, and unsorted part is the given array.
   -Sorted part is placed at the left, while the unsorted part is placed at the right.

   -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.



Here, smallest element is 8 and it checked with all from beginning.

8 < 23

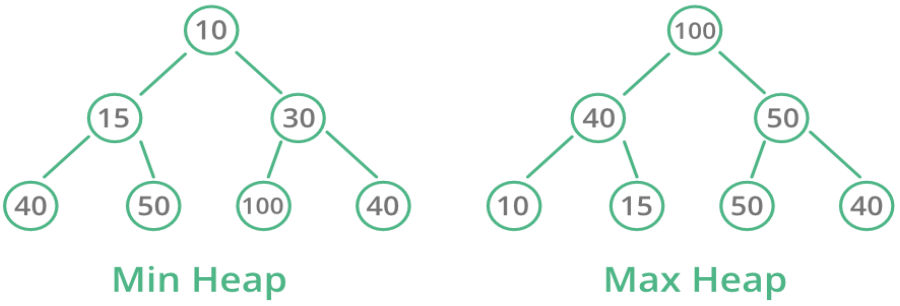
23 < 78

32 < 45

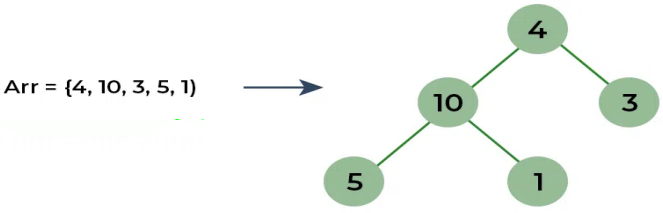
45 < 78

in the last 56 < 78 giving sorted array

3. **Heap sort**:-A heap is a **complete binary tree**, and the binary tree is a tree in which the node can have the utmost two children.
   -A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled.
   -It works by creating a max or min heap from an unsorted array.
   →Max heap:-The value of the root (N) >= The value of each of the children of N.
   →Min heap:-The value at N<= the value at any of the children of N.



Eg: let's take an unsorted array and try to sort it using heap sort.
Consider the array: arr[] = {4, 10, 3, 5, 1}.
-Build a complete binary tree from the array.

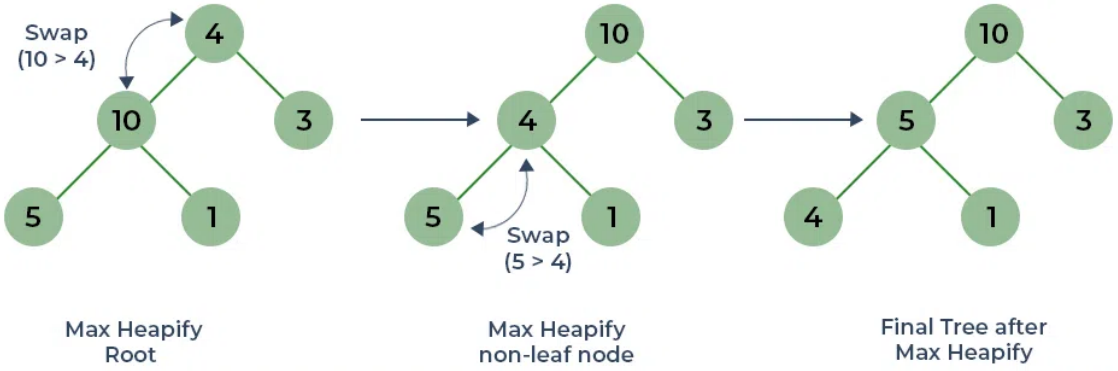


-To transform into max heap: After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.

-To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes

-Here, in this example, as the parent node 4 is smaller than the child node 10, thus, swap them to build a max-heap.

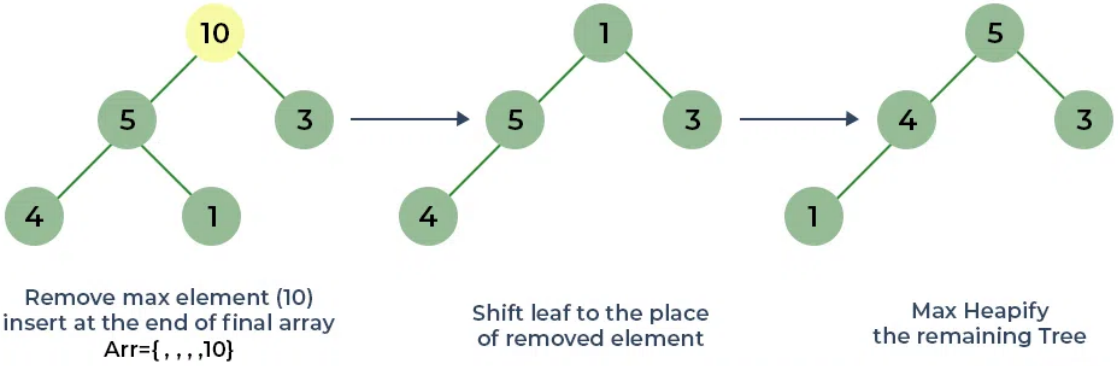
-Now, 4 as a parent is smaller than the child 5, thus swap both of these again and the resulted heap and array should be like this:



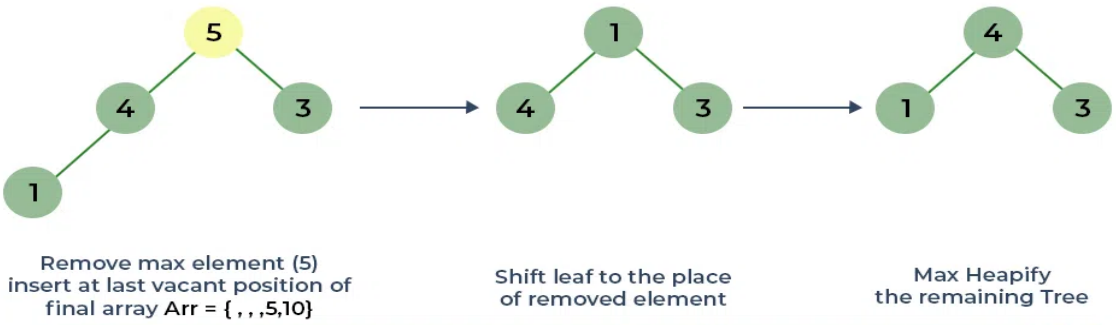
→**Perform heap sort**: Remove the maximum element in each step and then consider the remaining elements and transform it into a max heap.

-Delete the root element (10) from the max heap. In order to delete this node, try to swap it with the last node, i.e. (1). After removing the root element, again convert it into max heap.

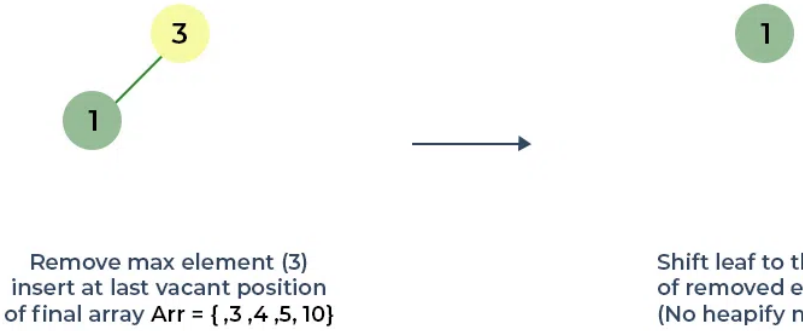
-Resulted heap and array should look like this:



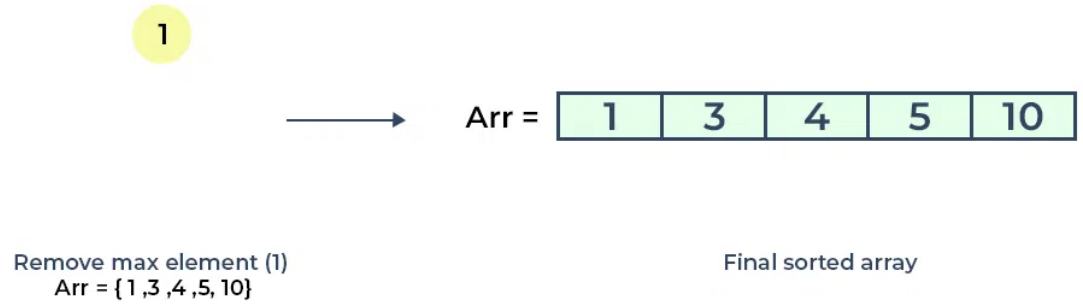
-Repeat the above steps and it will look like the following:



-Now remove the root (i.e. 3) again and perform heapify.



-Now when the root is removed once again it is sorted. and the sorted array will be like arr[] = {1, 3, 4, 5, 10}.



→Comparison between min heap and max heap (9marks)

	Min Heap	Max Heap
1.	In a Min-Heap the key present at the root node must be less than or equal to among the keys present at all of its children.	In a Max-Heap the key present at the root node must be greater than or equal to among the keys present at all of its children.
2.	In a Min-Heap the minimum key element present at the root.	In a Max-Heap the maximum key element present at the root.
3.	A Min-Heap uses the ascending priority.	A Max-Heap uses the descending priority.
4.	In the construction of a Min-Heap, the smallest element has priority.	In the construction of a Max-Heap, the largest element has priority.
5.	In a Min-Heap, the smallest element is the first to be popped from the heap.	In a Max-Heap, the largest element is the first to be popped from the heap.

Max Heap	Min Heap
1. The data at the root node should be greater than each of the child nodes.	1. The data at the root node should be smaller than each of the child nodes.
2. The element having the highest value has the highest priority assigned to it.	2. The element having the lowest value has the highest priority assigned to it.
3. The first value to be extracted is the maximum value.	3. The first value to be extracted is the minimum value.
4. It is used for the purpose of implementing Priority Queue.	4. It is used for the purpose of implementing Dijkstra Graph Algorithm and Minimum Spanning Trees.
5. It is used to sort the array in ascending order using Heap Sort.	5. It is used to sort the array in ascending order using Heap Sort.
6. Operations performed in Max Heap include Extract Maximum, Get Maximum and Insertion.	6. Operations performed in Min Heap include Extract Minimum, Get Minimum and Insertion.
7. The root of the tree must have the maximum value.	7. The root of the tree must have the minimum value.

4. **Radix Sort**:-Radix sort is the linear sorting algorithm that is **used for integers**.
   -In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.
   -The process of radix sort works similar to the sorting of students names, according to the alphabetical order.
   -In the first pass, the names of students are grouped according to the ascending order of the first letter of their names.
   -After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name.
   -And the process continues until we find the sorted list.

   In the first pass the 1,s digits are sorted.
   -Ih the second pas the 10's digits are sorted.
   -In the third pass the 100's digits are sorted.

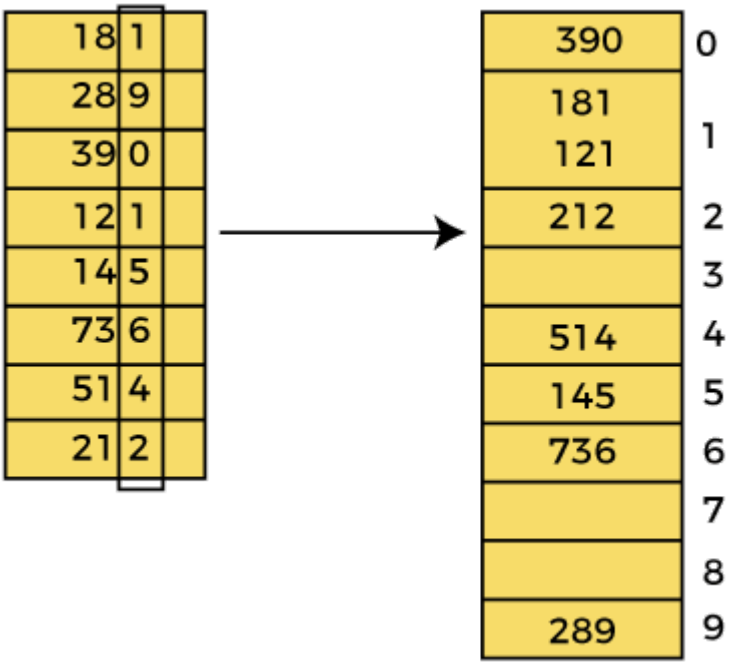
   -It is a in-place sort mechanism.

→**working:-**Now, let's see the working of Radix sort Algorithm.
-Given below is an unsorted array and try to sort it using radix sort.

181	289	390	121	145	736	514	212

-In the given array, the largest element is 736 that have 3 digits in it.
-So, the loop will run up to three times (i.e., to the hundreds place).
-That means three passes are required to sort the array.

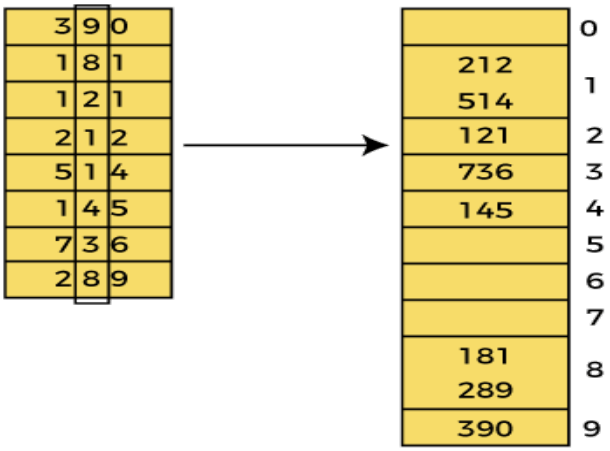
- Pass 1:In the first pass, the list is sorted on the basis of the digits at 1's place.



-After the first pass, the array elements are -

390	181	121	212	514	145	736	289

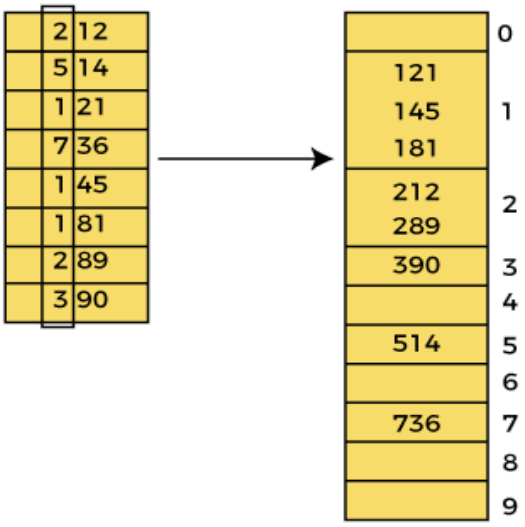
- Pass 2:In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).



-After the second pass, the array elements are -

212	514	121	736	145	181	289	390

- Pass 3:In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



-After the third pass, the array elements are -

121	145	181	212	289	390	514	736

-Now, the array is sorted in ascending order

→Radix sort complexity(3 marks):

- **Time Complexity**:-

➢ Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of Radix sort is **Ω(n+k)**.

➢ Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of Radix sort is **θ(nk).**

➢ Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of Radix sort is **O(nk).**

N-number of the digits in an array
-here we have 8 elements
K-number of digits in a number ,therefore K=3.
-because 121 have 3 numbers

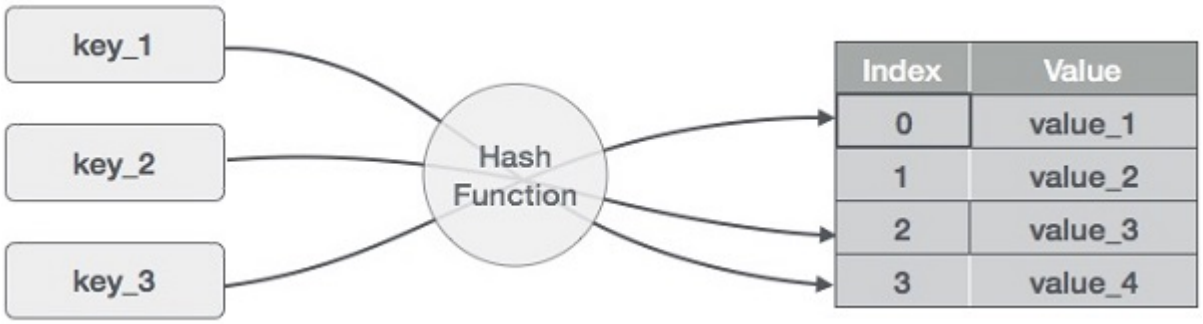
- **Space Complexity**:-The space complexity of Radix sort is O(n + k).

→Time and Space Complexity Comparison Table of various sorting methods:

Sorting methods	Time Complexity			Space Complexity
	Best case	Average Case	Worst Case	
Insertion Sort	Ω(N)	Θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	Θ(N^2)	O(N^2)	O(1)
Heap Sort	Ω(N log N)	Θ(N log N)	O(N log N)	O(1)
Radix Sort	Ω(N k)	Θ(N k)	O(N k)	O(N + k)

→Stable and unstable sorting:-google

#Hashing:-Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function.
-It is done for faster access to elements.
-A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., **key and value**.
-The hash table can be implemented with the help of an associative array
-The efficiency of mapping depends on the efficiency of the hash function used.
-Hashing is a technique to convert a range of key values into a range of indexes of an array.
-We're going to **use modulo operator to get a range of key values.**

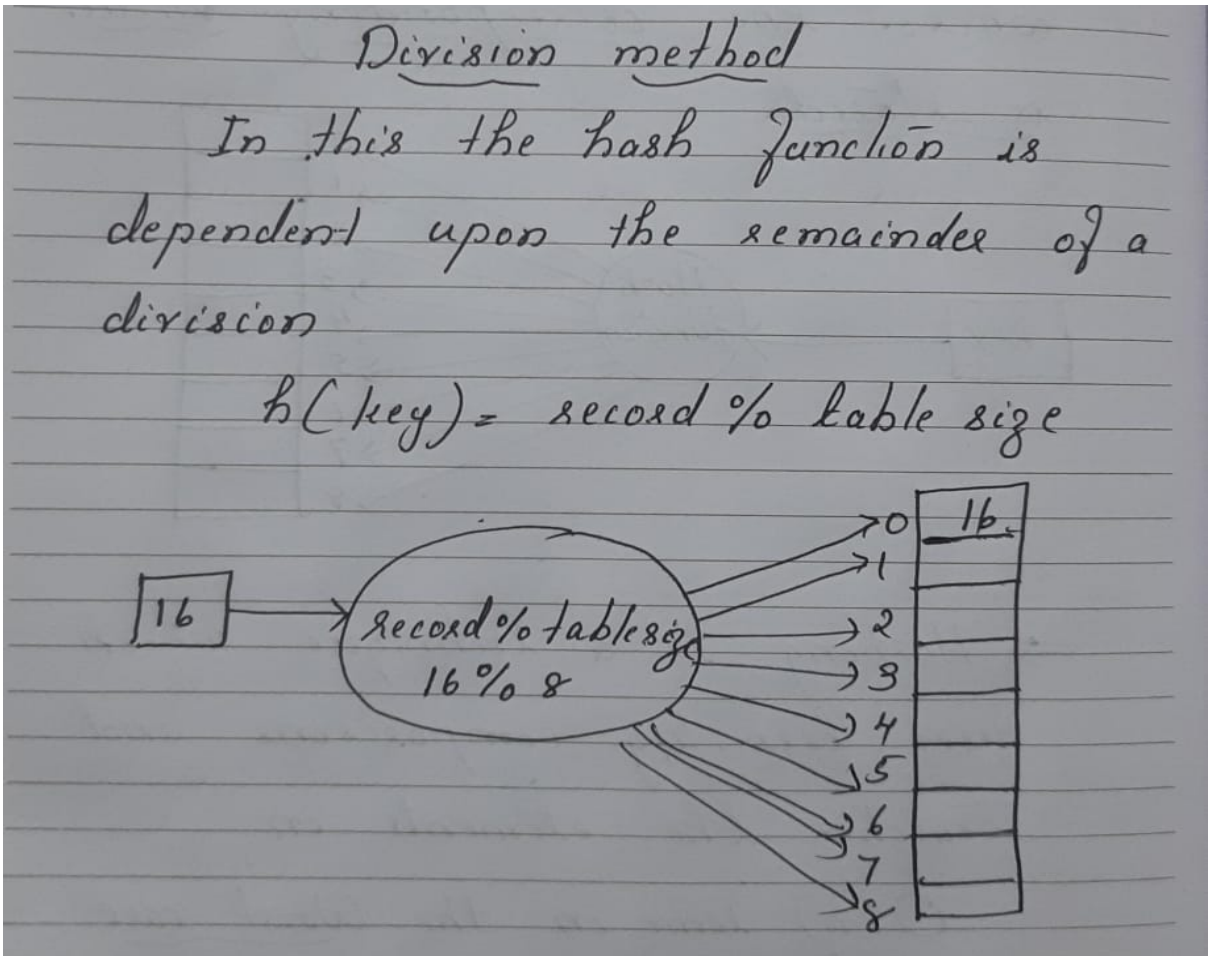


eg:Consider an example of hash table and the following items are to be stored.
Item are in the (key,value) format.
(1,20) , (2,70) , (42,80) , (4,25) , (12,44) , (14,32) , (17,11) , (13,78) , (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

***Hash functions (9 marks):**-The type of hash function are;

1. **Division Method**
2. **Mid Square Method**
3. **Digit folding method**



In the above example we store the data 16 in the 0th index location of the hash table.

Mid square method

⇒ In this method firstly key is squared and then mid part of the result is taken as the index.

⇒ For example, consider that if we want to place a record of 3101 and size of the table is 1000.

So 3101 * 3101 = 9616201.

ie, h(3101) = 162 (Middle 3 digits)

Digit folding method

⇒ In this method, the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key.

⇒ For example, consider a record of 124655012 then it will be divided into parts. ie, 124, 655, 012, After dividing the parts combine these parts by adding it.

H(key) = 124 + 655 + 012 = 791

***Collision Resolution**:-It is possible that two different key k1 and k2 will have the same hash address.

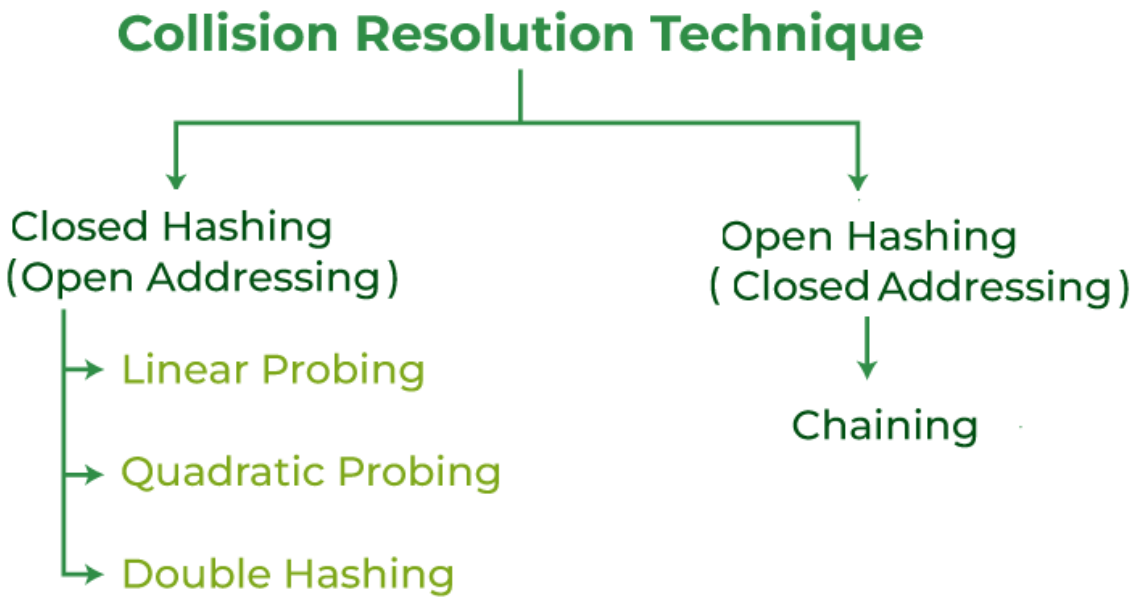
-This situation is called collision.

-or when two values are stored at the same index, and this leads to the collision problem.

-To resolve these collisions, we have some techniques known as collision techniques.

-The following are the collision techniques:

1. **Closed Hashing: It is also known as open addressing.**
2. **Open Hashing: It is also known as closed addressing.**



1. Closed Hashing: It is also known as open addressing.

   -Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left.

   -In Closed hashing, three techniques are used to resolve the collision:

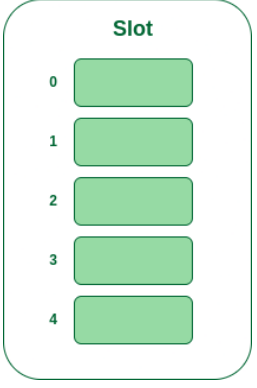
   - **Linear probing**:-Here ,if a record with key K is mapped to an address but the address is already occupied by another key.

     -Then linear probing technique start to search for the closest free locations .

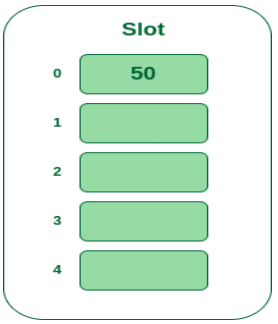
     -In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is found.

     -Eg:Let us consider a simple hash function as “key mod 5” and a sequence of keys that are to be inserted are 50, 70, 76, 85, 93.

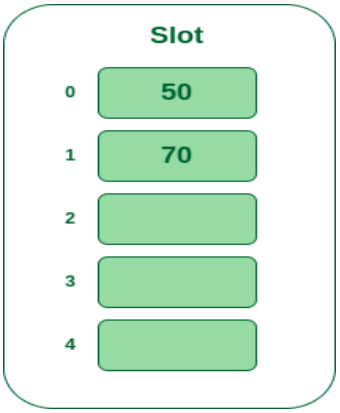
     ➢ Step 1:First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



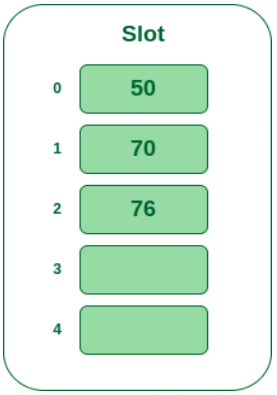
- Step 2: Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



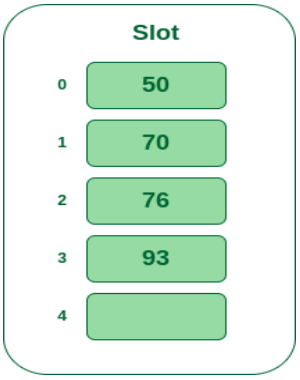
- Step 3: The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



- Step 4: The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



- Step 5: The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into lot number 3.



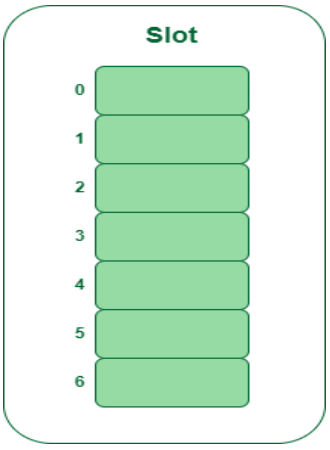
- **Quadratic probing**:-Quadratic probing operates by **taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.**

  -This method is also known as the **mid-square method.**

*If the slot hash(x) % n is full, then we try (hash(x) + 12) % n. *If (hash(x) + 12) % n is also full, then we try (hash(x) + 22) % n. *If (hash(x) + 22) % n is also full, then we try (hash(x) + 32) % n. *This process will be repeated for all the values of i until an empty slot is found

-Eg:Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i2 . Insert = 22, 30, and 50.

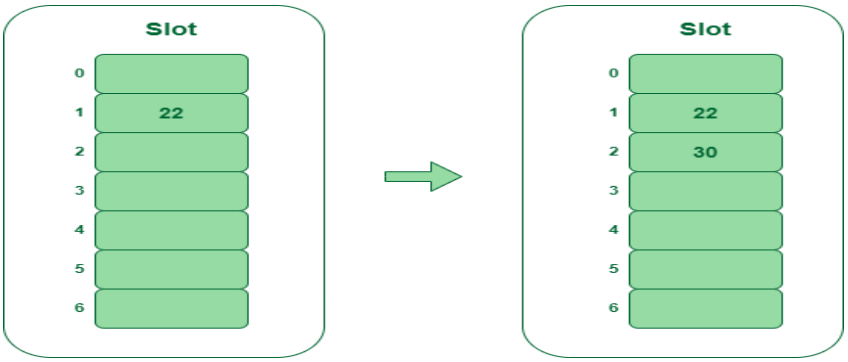
- Step 1: Create a table of size 7.



- Step 2 – Insert 22 and 30

  -Hash(25) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.

  -Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



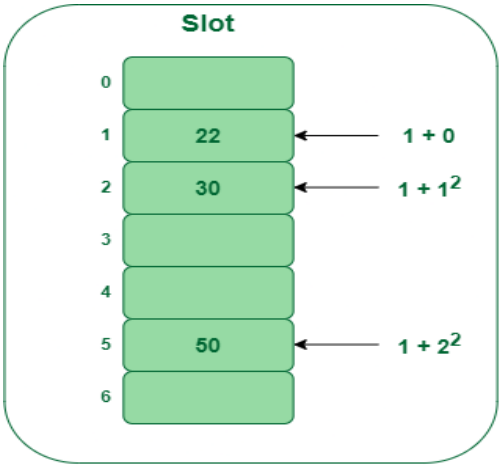
- Step 3: Inserting 50

  -Hash(25) = 50 % 7 = 1

  -In our hash table slot 1 is already occupied. So, we will search for slot 1+12, i.e. 1+1 = 2,

  -Again slot 2 is found occupied, so we will search for cell 1+22, i.e.1+4 = 5,

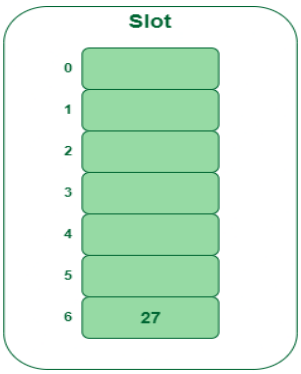
  -Now, cell 5 is not occupied so we will place 50 in slot 5.



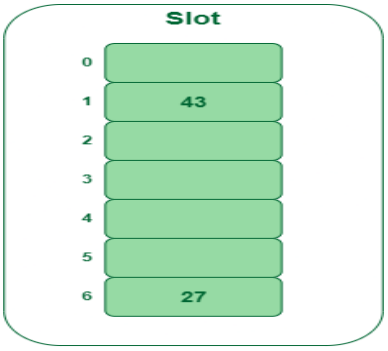
- **Double Hashing technique**:-It is the efficient method of collision resolution technique.
  -Because it uses double hash function to calculate the address of any key.
  -This combination of hash functions is of the form

  $$h(k, i) = (h1(k) + i * h2(k)) \% n$$

  -h1(k) and h2(k) are two different hash functions.
  -i indicates a collision number, n indicates the hash table size.
  -Eg:Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and
  second hash-function is **h2(k) = 1 + (k mod 5)**
  - ➢ Step 1: Insert 27
    -27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



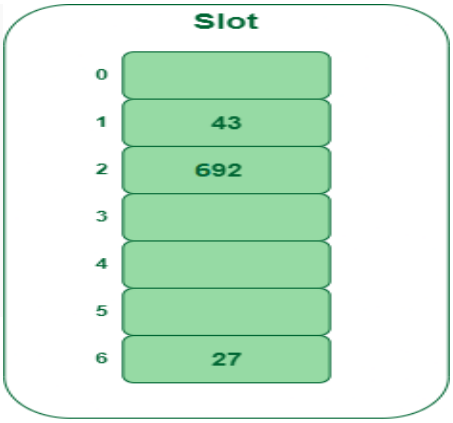
  - ➢ Step 2: Insert 43
    -43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



  - ➢ Step 3: Insert 692
    -692 % 7 = 6, but location 6 is already being occupied and this is a collision
    -So we need to resolve this collision using double hashing.

h_{new} = [h1(692) + i * (h2(692)] % 7
= [6 + 1 * (1 + 692 % 5)] % 7
= 9 % 7
= 2

Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.

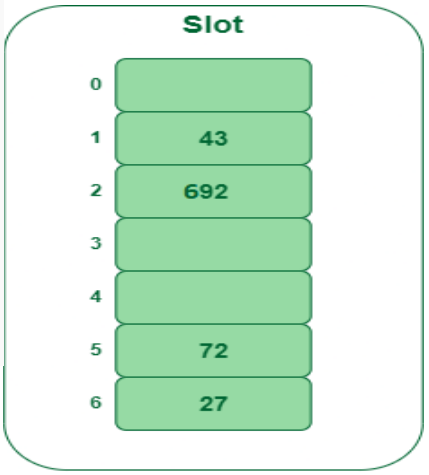


  - ➢ Step 4: Insert 72
    -72 % 7 = 2, but location 2 is already being occupied and this is a collision.

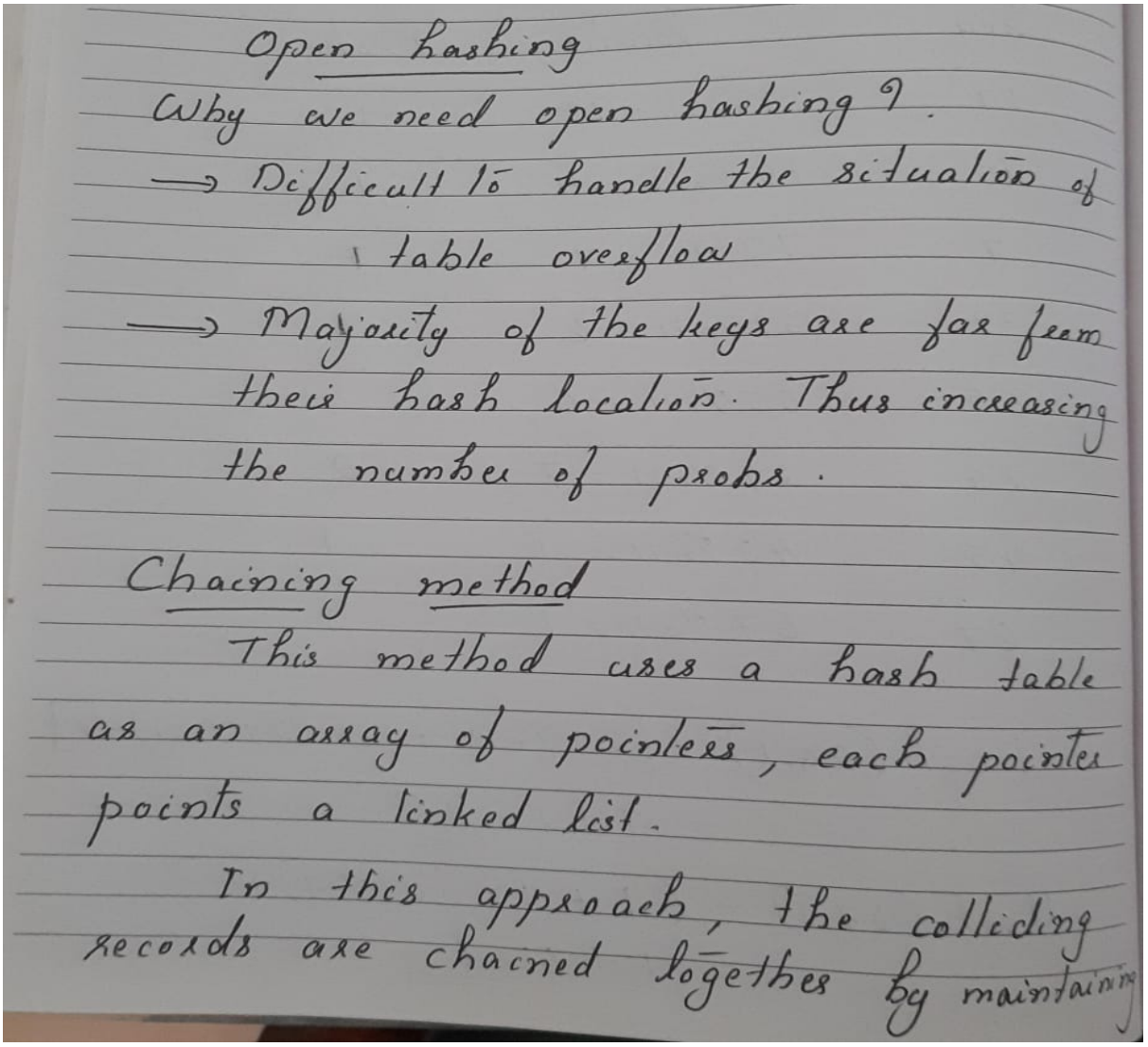
-So we need to resolve this collision using double hashing.

h_{new} = [h1(72) + i * (h2(72)] % 7
= [2 + 1 * (1 + 72 % 5)] % 7
= 5 % 7
= 5,

Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



2. Open Hashing: It is also known as closed addressing.


Open hashing

Why we need open hashing ?

→ Difficult to handle the situation of table overflow

→ Majority of the keys are far from their hash location. Thus increasing the number of probs.

Chaining method

This method uses a hash table as an array of pointers, each pointer points a linked list.

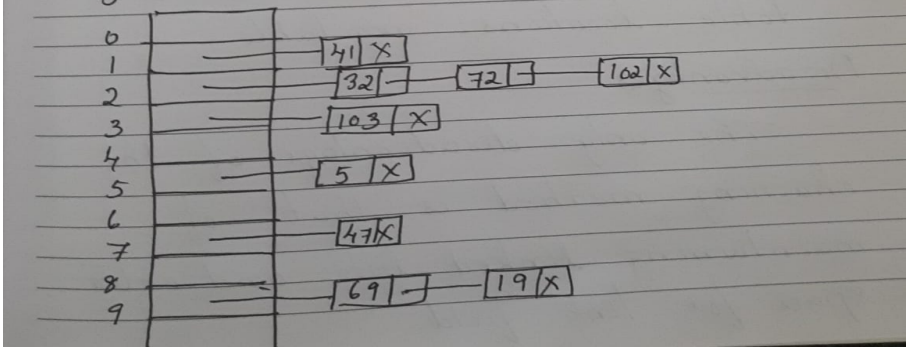
In this approach, the colliding records are chained together by maintaining

a linked list of such colliding records. A separate one way list is used for each set of colliding records.

Example

Insert 47, 32, 69, 41, 103, 72, 05, 19, 102 in hash table of size 10.

Suppose we are considering the hash address of the key is decided by its last digit.



Advantages of Chaining

→ Overflow situation never arises

→ collision ~~resto~~ resolution can be achieved very efficiently, free from clustering

→ Insertion and deletion become quick and easy task.

→ Remains effective even when the no. of key values to be store is much larger than the size of hash table locations available.

Disadvantages

The only disadvantage of the chaining method is that of maintaining linked lists and extra space for link field.

Join for more MCA short note : https://t.me/mgu_mca_shortnote

@MGU_MCA_SHORTNOTE