SEMESTER EXAM SERIES

DATA STRUCTURES

IN 7 HOURS + NOTES

Video Chapters

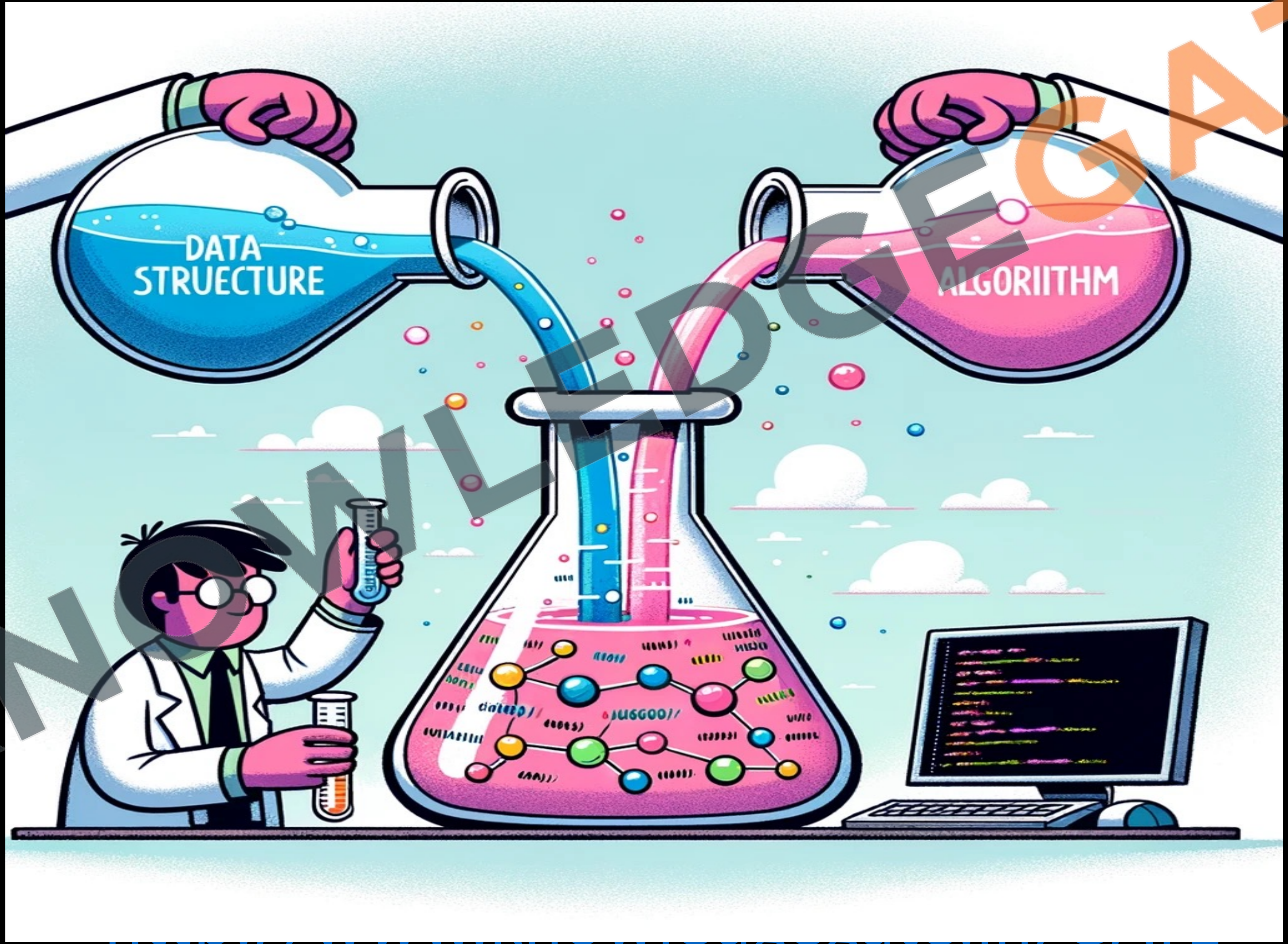
- **(Ch-1) Introduction** : Basic Terminology, Elementary Data Organization, Built in Data Types in C. Abstract Data Types (ADT)
- **(Ch-2) Array:** Definition, Single and Multidimensional Arrays, Representation of Arrays: Row Major Order, and Column Major Order, Derivation of Index Formulae for 1-D,2-D,3-D and n-D Array Application of arrays, Sparse Matrices and their representations.
- **(Ch-3) Linked lists**: Array Implementation and Pointer Implementation of Singly Linked Lists, Doubly Linked List, Circularly Linked List, Operations on a Linked List. Insertion, Deletion, Traversal, Polynomial Representation and Addition Subtraction & Multiplications of Single variable & Two variables Polynomial.
- **(Ch-4) Stack**: Abstract Data Type, Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack in C, Application of stack: Prefix and Postfix Expressions, Evaluation of postfix expression, Iteration and Recursion- Principles of recursion, Tail recursion, Removal of recursion Problem solving using iteration and recursion with examples such as binary search, Fibonacci numbers, and Hanoi towers. Trade offs between iteration and recursion.
- **(Ch-5) Queue**: Create, Add, Delete, Full and Empty, Circular queues, Array and linked implementation of queues in C, Dequeue and Priority Queue.
- **(Ch-6) Tree**: Basic terminology used with Tree, Binary Trees, Binary Tree Representation: Array Representation and Pointer(Linked List) Representation, Binary Search Tree, Strictly Binary Tree ,Complete Binary Tree . A Extended Binary Trees, Tree Traversal algorithms: Inorder, Preorder and Postorder, Constructing Binary Tree from given Tree Traversal, Operation of Insertion , Deletion, Searching & Modification of data in Binary Search . Threaded Binary trees, Traversing Threaded Binary trees. Huffman coding using Binary Tree. Concept & Basic Operations for AVL Tree , B Tree & Binary Heaps
- **(Ch-7) Graphs**: Terminology used with Graph, Data Structure for Graph Representations: Adjacency Matrices, Adjacency List, Adjacency. Graph Traversal: Depth First Search and Breadth First Search.
- **(Ch-8) Hashing**: Concept of Searching, Sequential search, Index Sequential Search, Binary Search. Concept of Hashing & Collision resolution Techniques used in Hashing

Idea of computer science

- Computer science deals with solving a problem correctly in the form of Algorithm which then can be converted into a program, in most efficient time and memory.
- Problem --> Solution(Algorithm) →Program (Efficient)

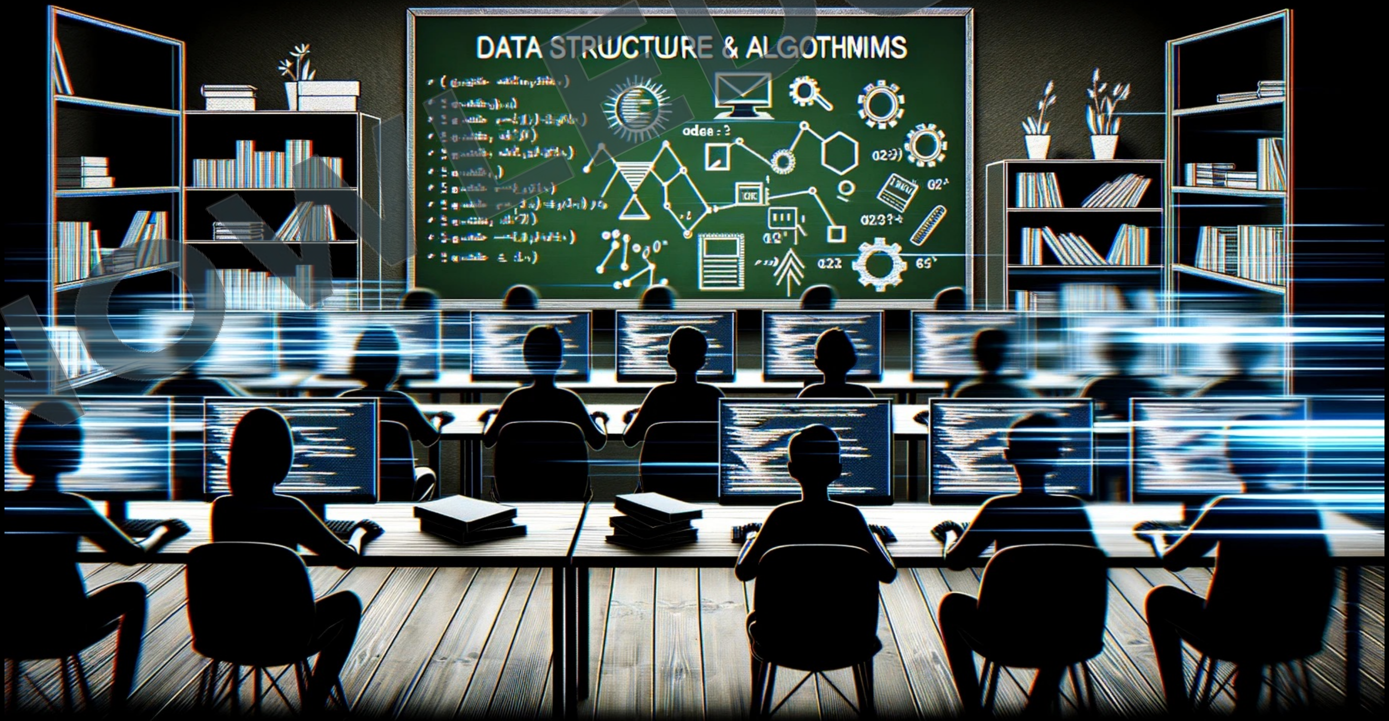
http: GATE

- To write an efficient program we need knowledge of both Data Structures and Algorithms.
  - DATA STRUCTURE + ALGORITHM = PROGRAM



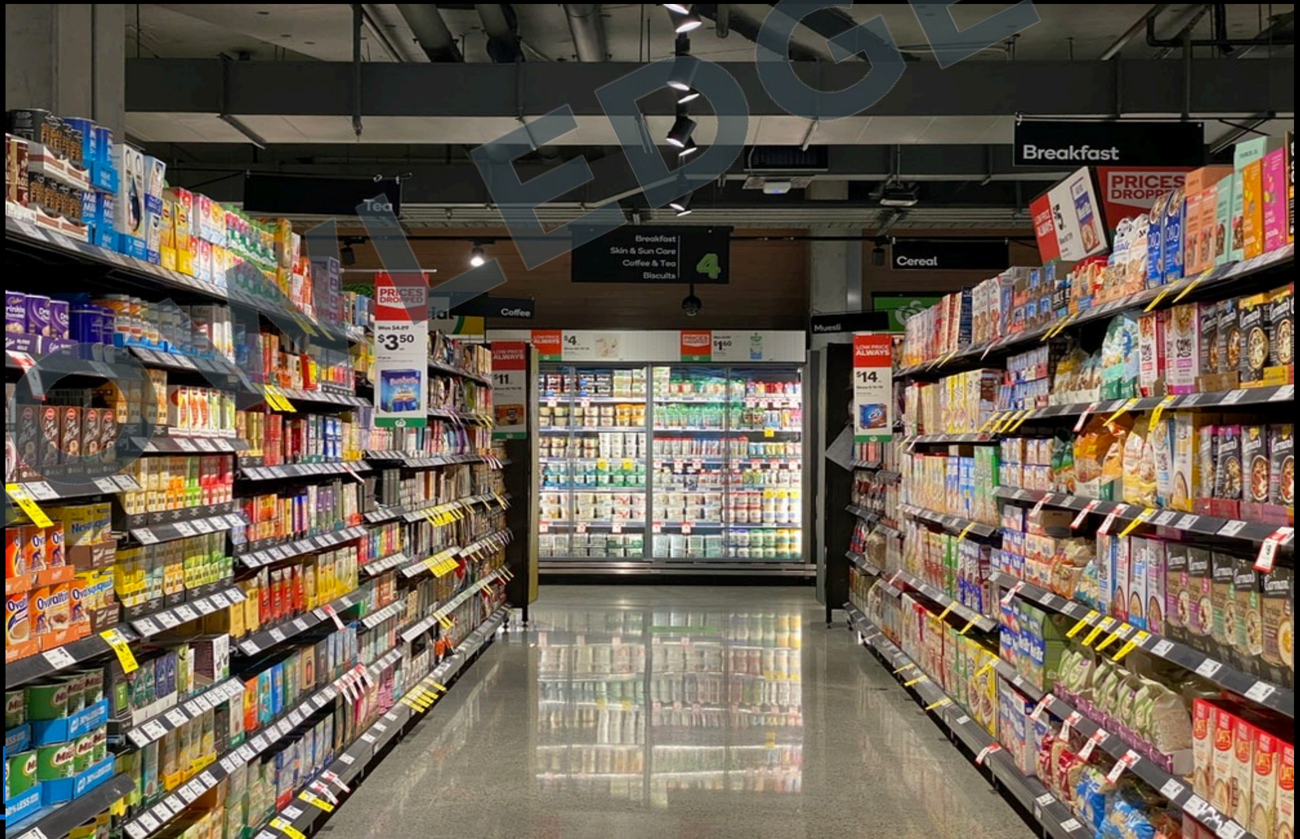
Why we study data structure and algorithms

- Course objective is to teach you how to code efficiently.
- What is the meaning of efficiency (time, space, battery, system buses, register etc) time is considered as most important.
- Better running time is obtained from the use of most appropriate data structure and algorithms, rather than through removing a few statements by clever coding.

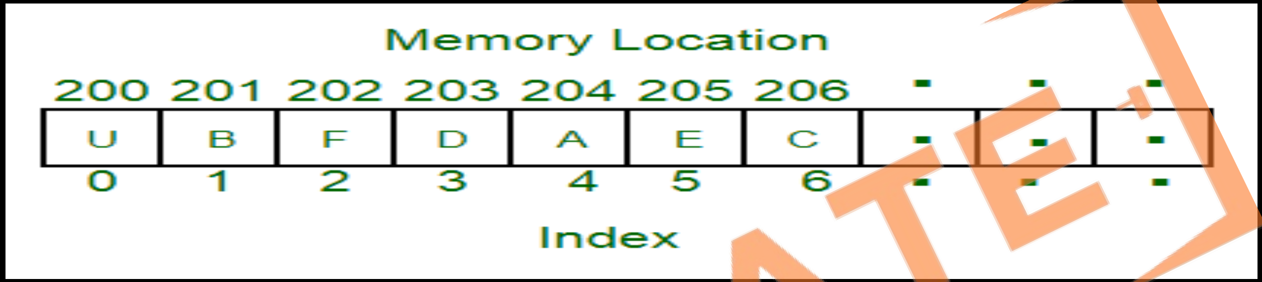


What is data structure

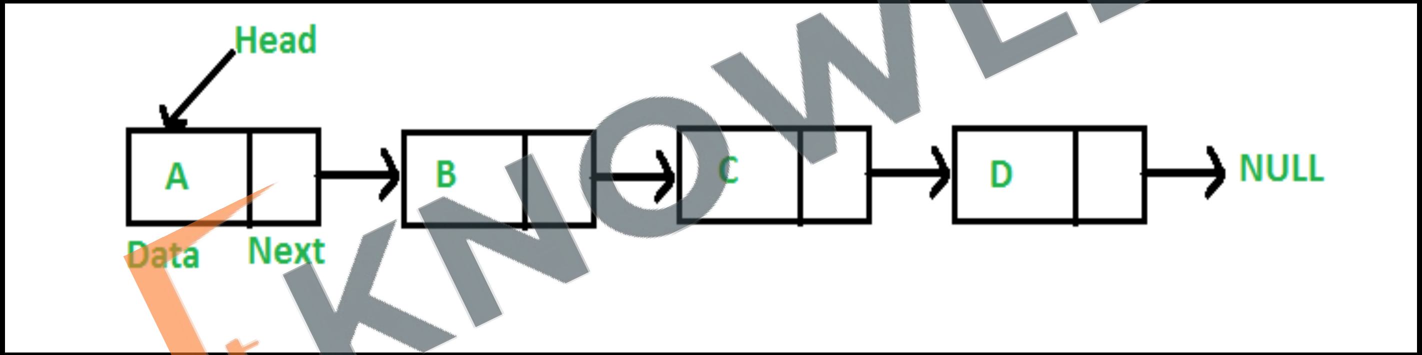
- **Data structure** is a particular way of organizing data in a computer memory (cache, main, secondary) so that Memory can be used efficiently both in terms of time and space.
- It is a logical relationship existing between individual elements of data, it considers elements stored and also their relationship to each other.



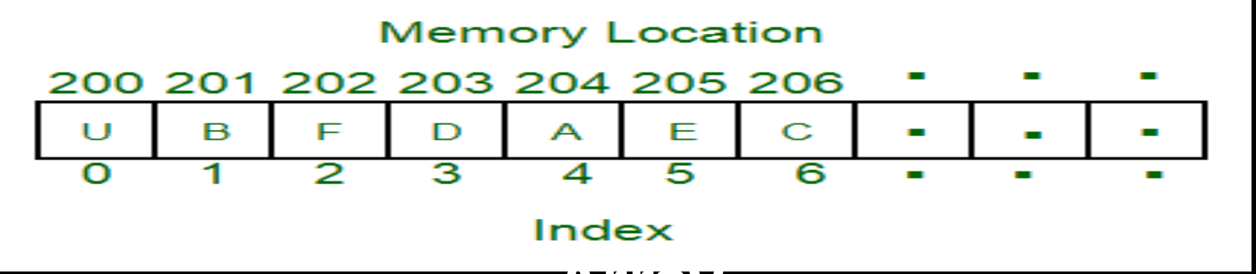
- Data structure mainly specifies the following four things: -
  - Organization of data
  - Accessing methods
  - Degree of association
  - Processing methods



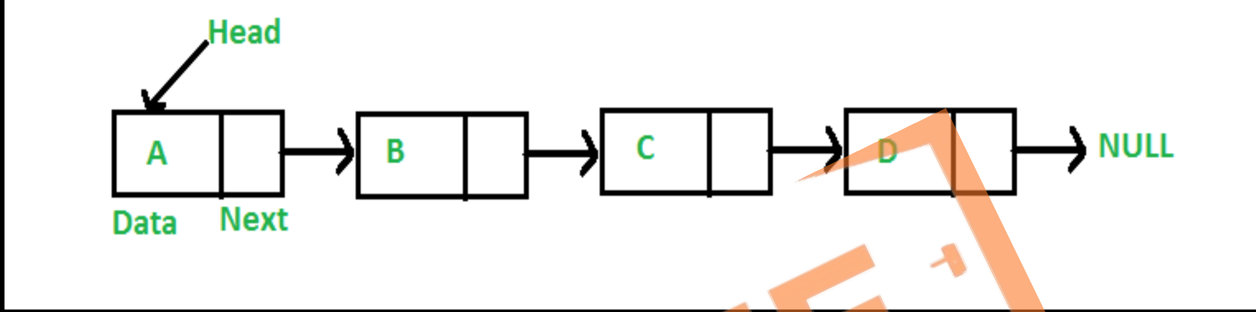
Array

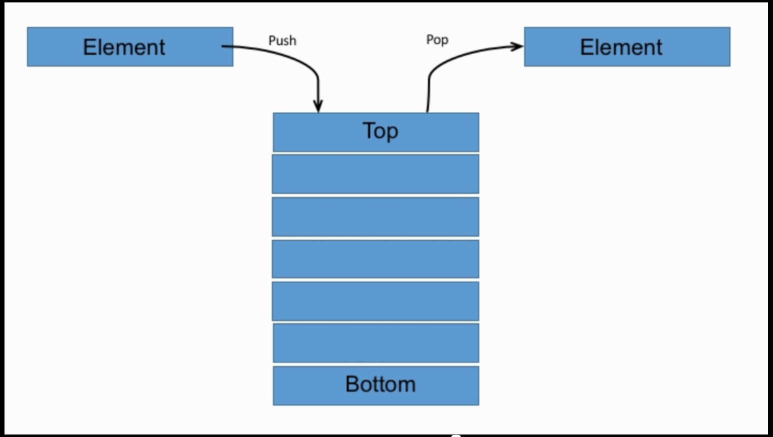


Link List

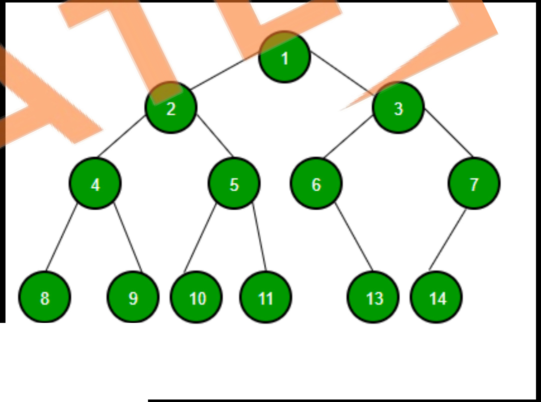


Array

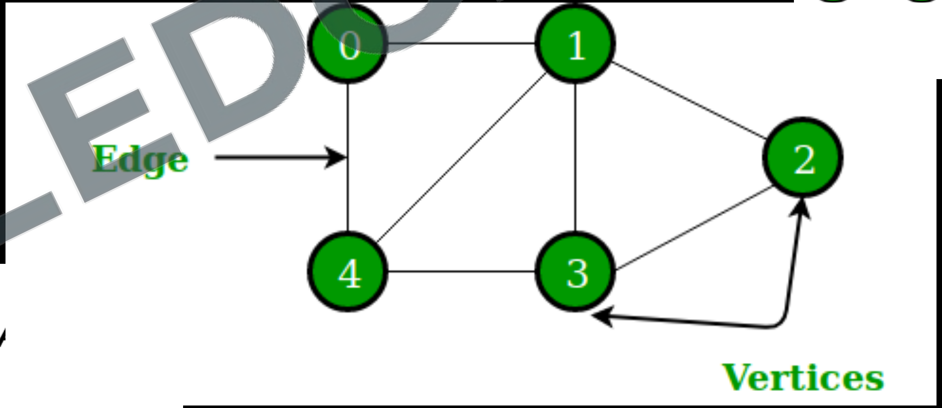




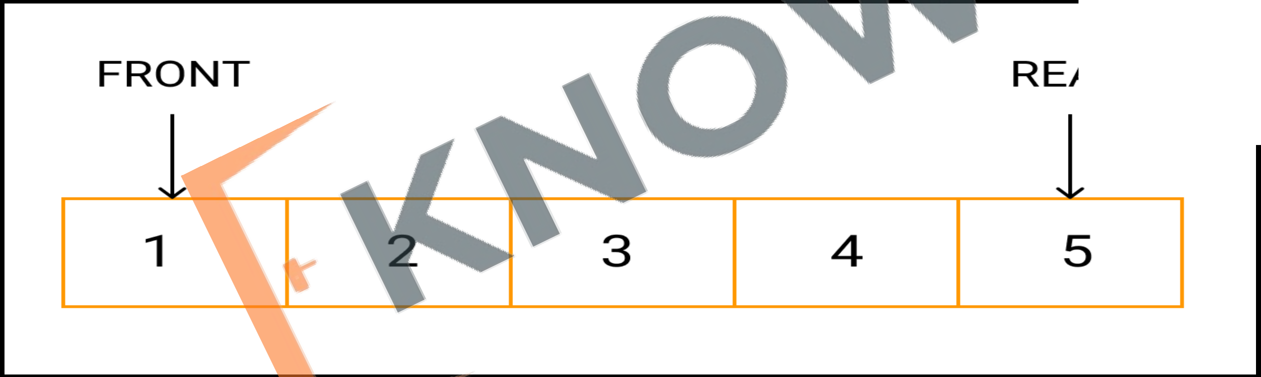
Stack



Tree



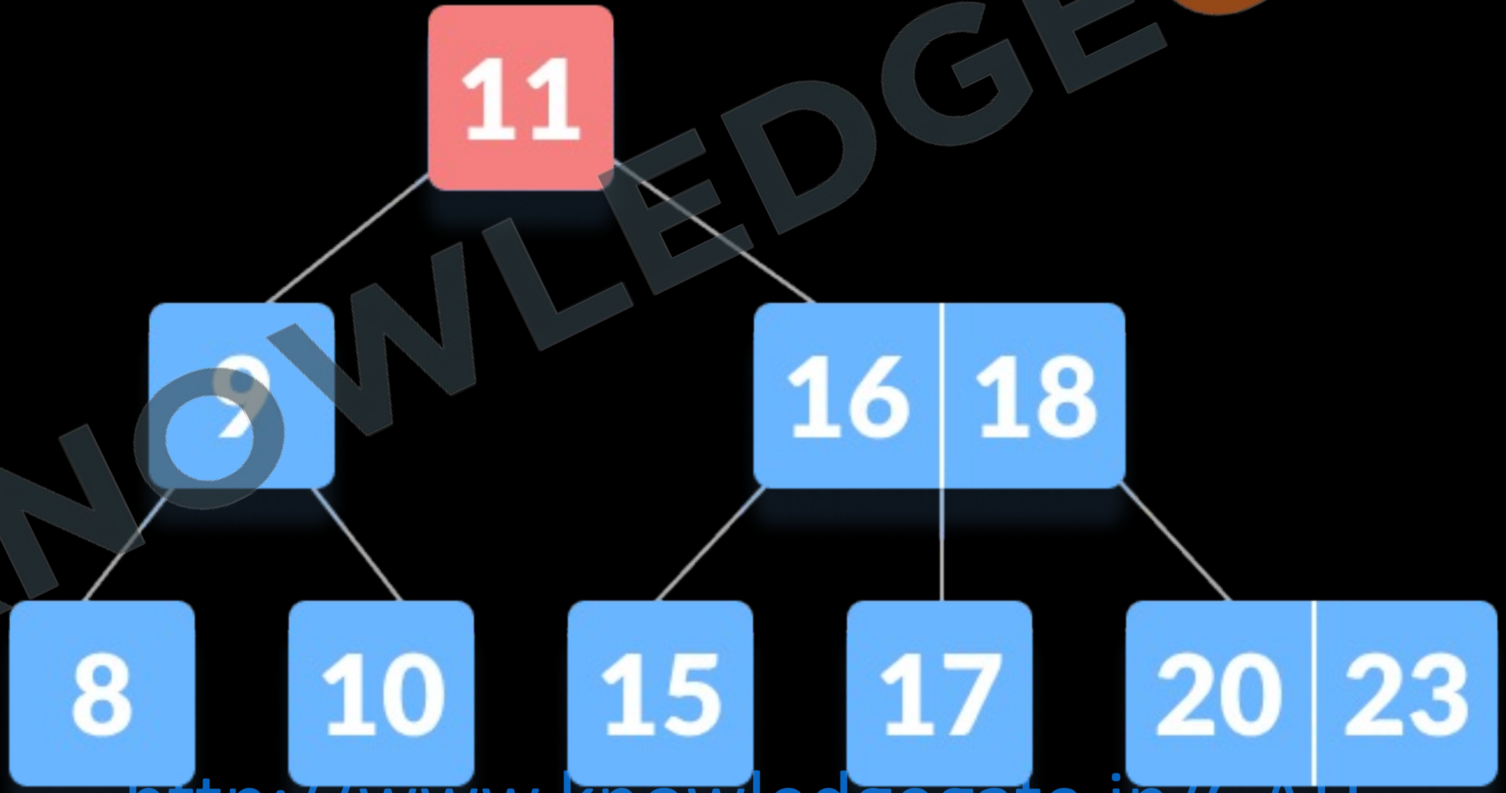
Graph



Queue

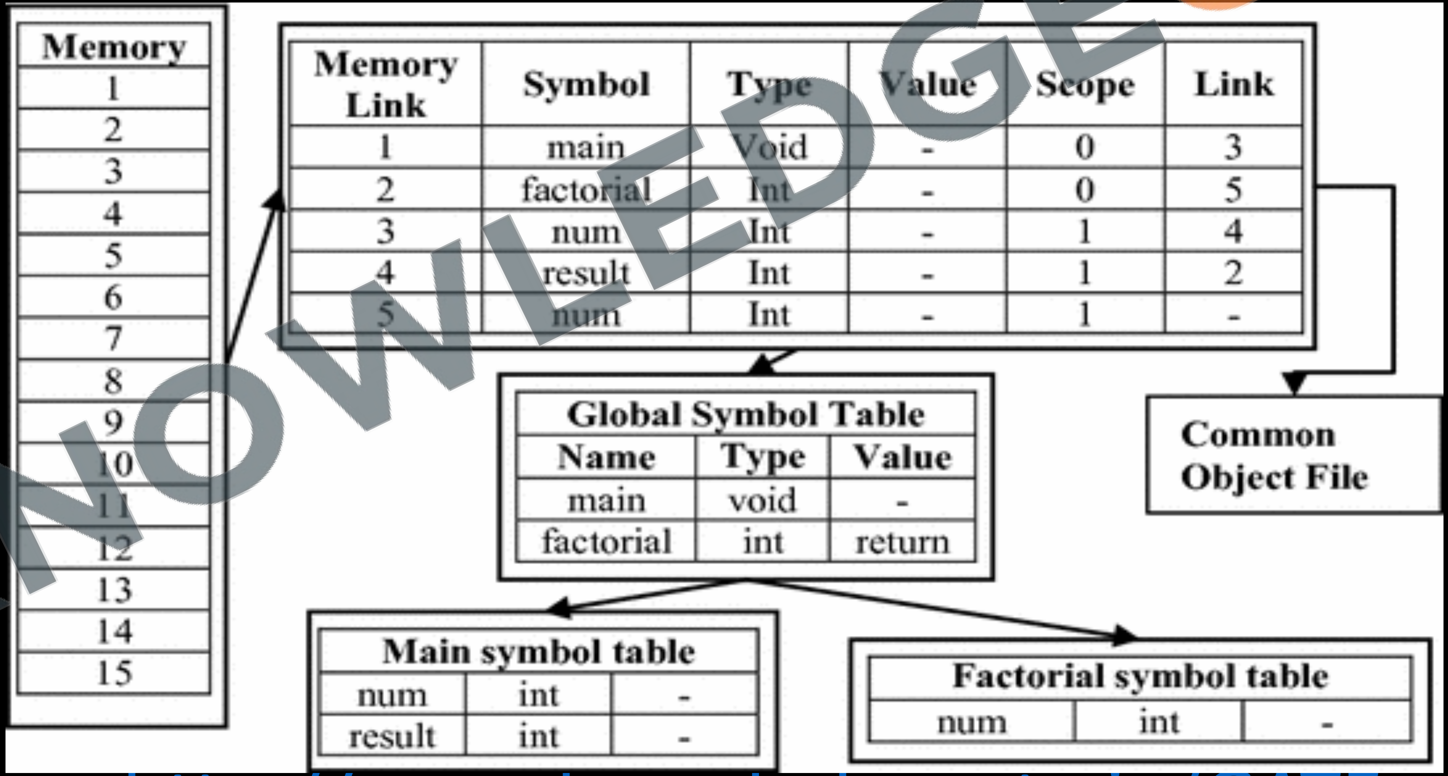
Effect of Data Structure

- Data structure effect both the structural and functional aspects of the program.
- Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks, For example.
  - Relational databases commonly use B-tree indexes for data retrieval
  - Compiler implementations usually use hash tables to look up identifiers.



- Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design.

- The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure.



Memory
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Memory Link	Symbol	Type	Value	Scope	Link
1	main	Void	-	0	3
2	factorial	Int	-	0	5
3	num	Int	-	1	4
4	result	Int	-	1	2
5	num	Int	-	1	-

Global Symbol Table

Name	Type	Value
main	void	-
factorial	int	return

Common Object File

Main symbol table

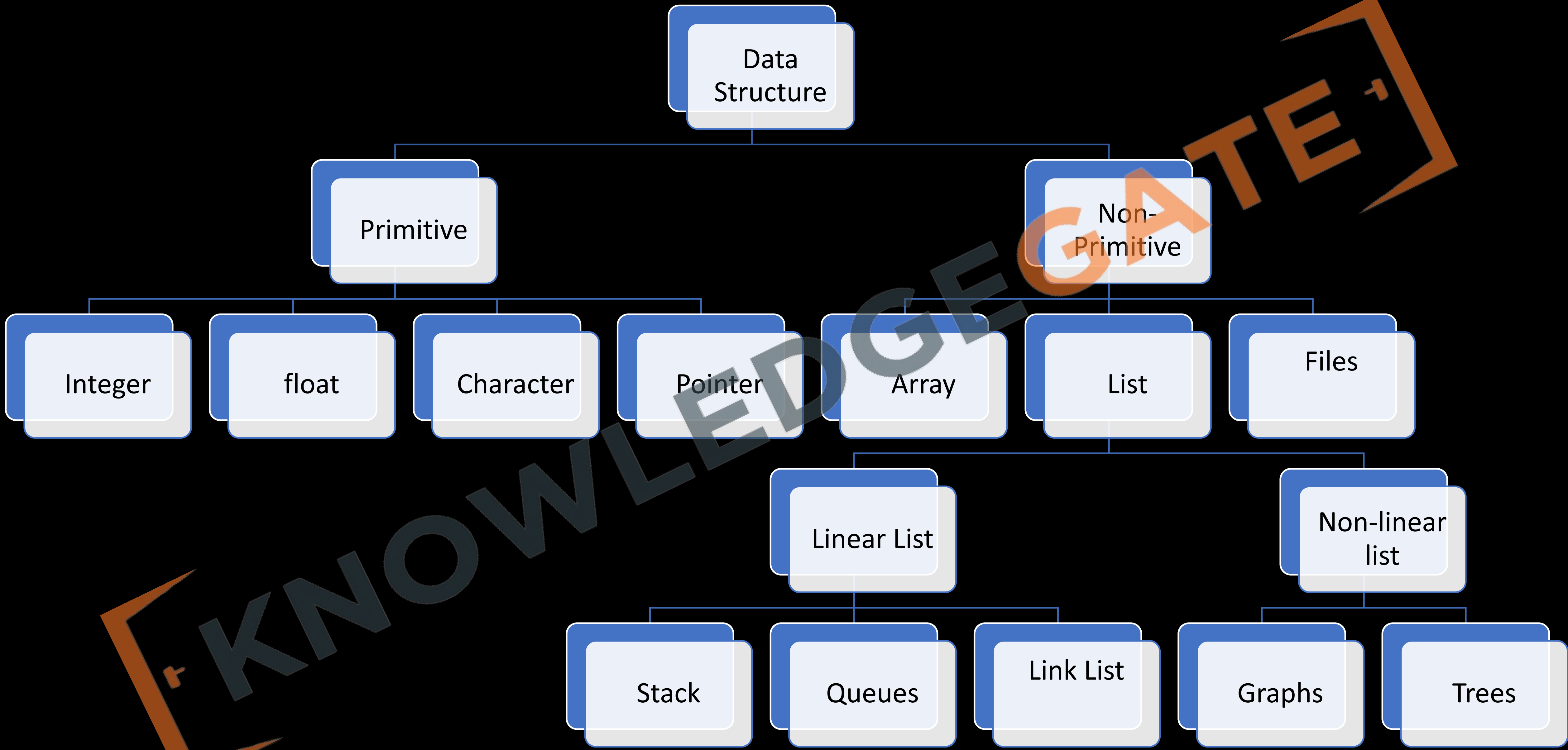
num	int	-
result	int	-

Factorial symbol table

num	int	-

Primitive data structure

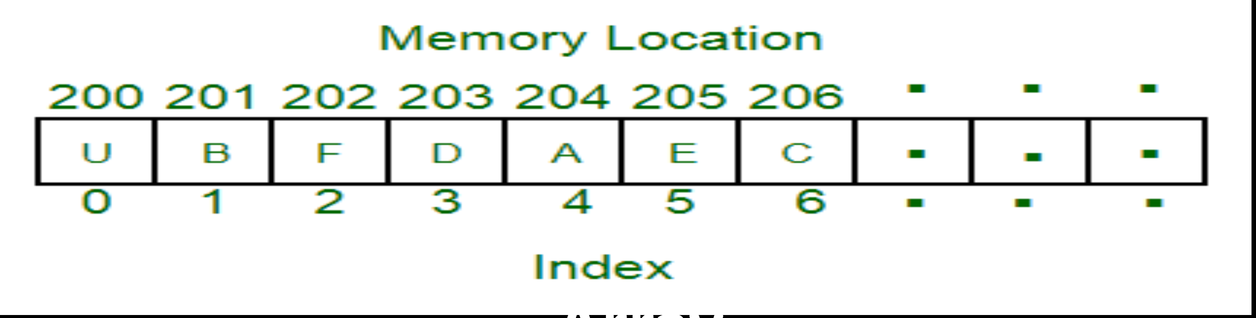
- Primitive data structures are those which have predefined way of storing data by the system. And the set of operations that can be performed on these data are also predefined. They are directly operated upon by the machine instruction.
- Primitive data structures are char, int, float, double. The predefined operations are addition, subtraction, etc.



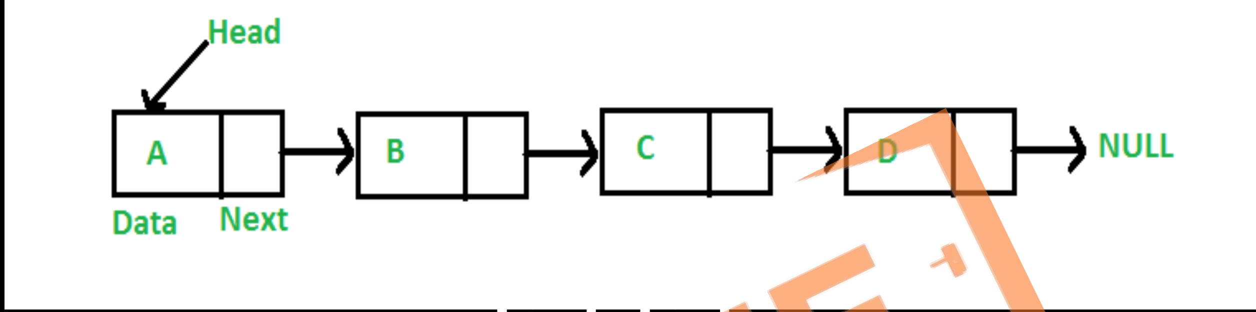
http://www.knowledgegate.in/GATE

Non-Primitive data structure

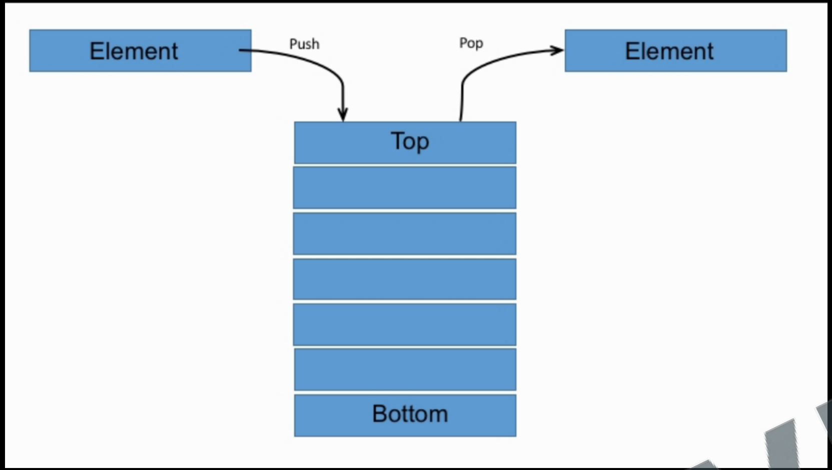
- But there are certain situations when primitive data structures are not sufficient for our job. There comes derived data structures and user defined data structures.
- Derived data structures are also provided by the system but are made using primitives like an array. It can be array of chars, array of int, etc. The set of operations that can be performed on derived data structures are also predefined.
- Finally, there are user defined data types which the user defines using the primitive and derived data types using language constructs like structure or class and uses according to their needs. And the user has to define the set or operations that we can perform on them. User defines data types are Linked Lists, Trees, etc.



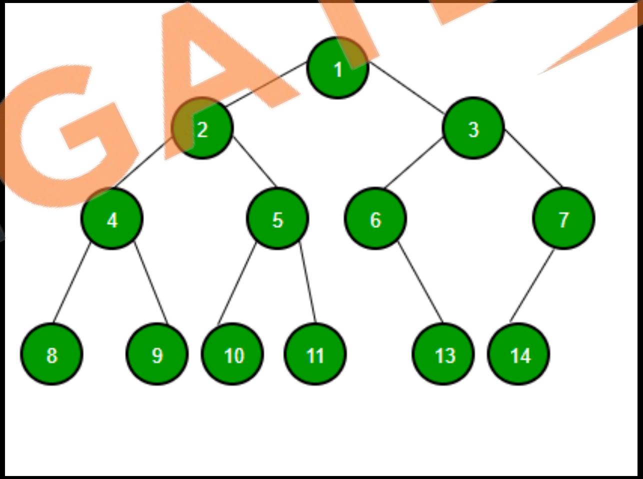
Array



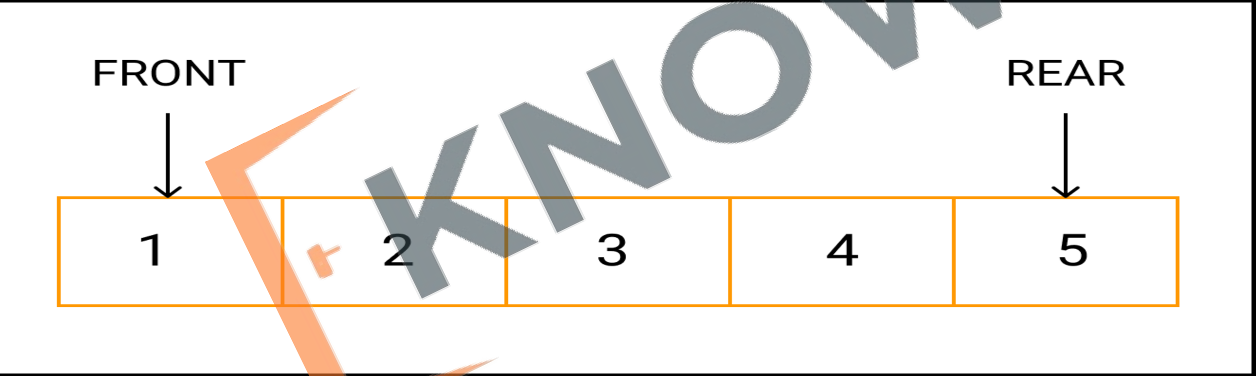
Link List



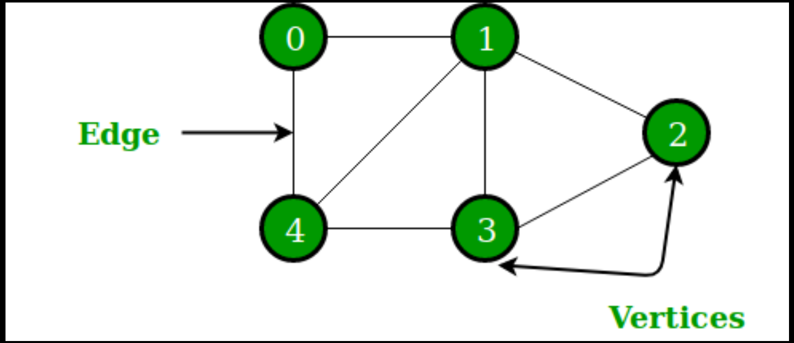
Stack



Tree



Queue



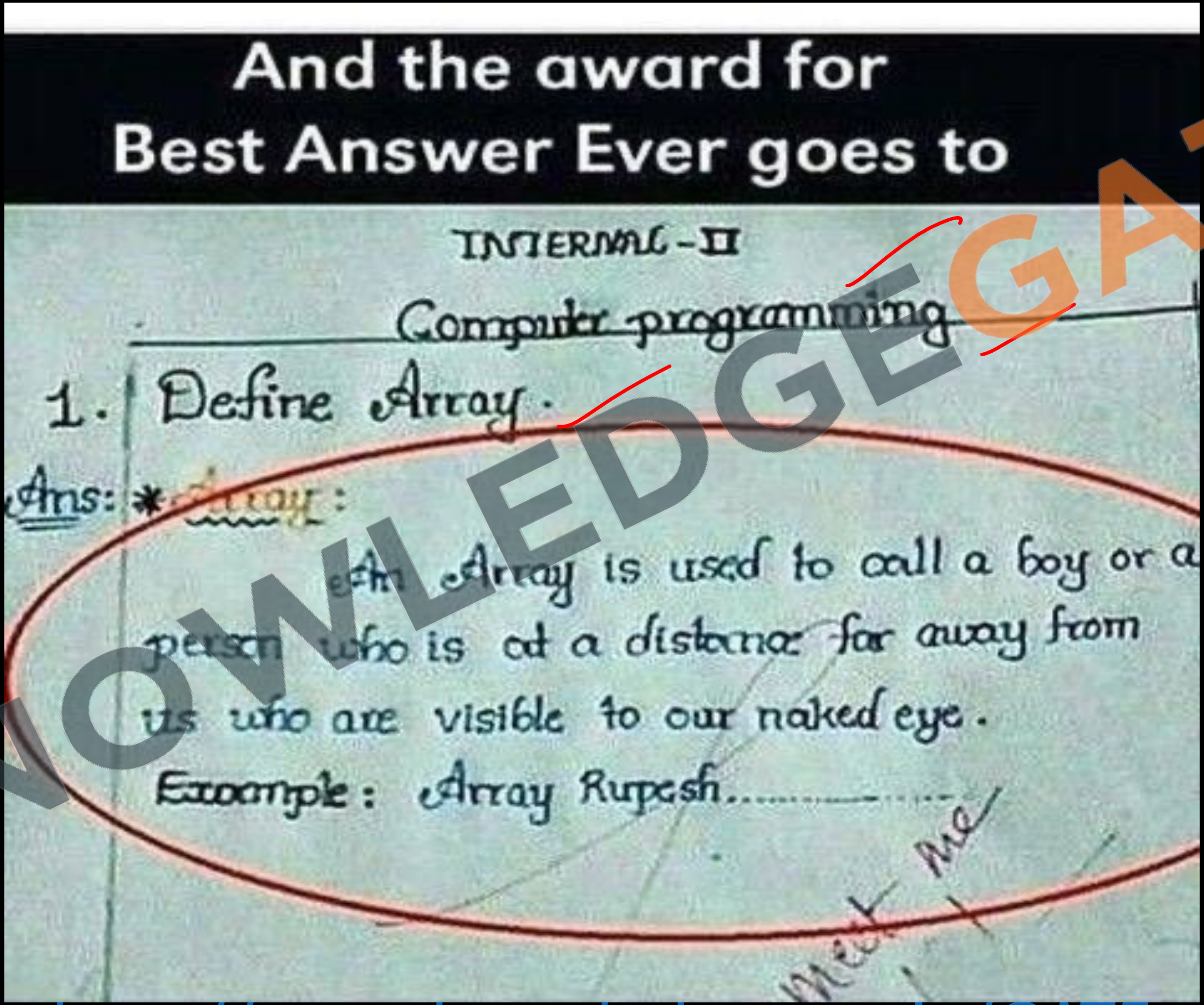
Graph

LINEAR DATA STRUCTURE	NON-LINEAR DATA STRUCTURE
• In a linear data structure, data elements are arranged in a linear order where each and every element are attached to its previous and next adjacent.	• In a non-linear data structure, data elements are attached in hierarchical manner.
• In linear data structure, single level is involved.	• Whereas in non-linear data structure, multiple levels are involved.
• Its implementation is easy in comparison to non-linear data structure.	• While its implementation is complex in comparison to linear data structure.
• In linear data structure, data elements can be traversed in a single run only.	• While in non-linear data structure, data elements can't be traversed in a single run only.
• Its examples are: array, stack, queue, linked list, etc.	• While its examples are: trees and graphs.

- **Homogeneous data:** Homogeneous data structures are those data structures that contain only similar type of data e.g. like a data structure containing only integer or float values. The simplest example of such type of data structures is an Array.

- **Heterogeneous Data:** Heterogeneous Data Structures are those data structures that contains a variety or dissimilar type of data, for e.g. a data structure that can contain various data of different data types like integer, float and character. The examples of such data structures include structures, union etc.

Array



And the award for
Best Answer Ever goes to

INTERNAL - II

Computer programming

1. Define Array.

Ans: * ~~Array~~ :

An Array is used to call a boy or a person who is at a distance far away from us who are visible to our naked eye.

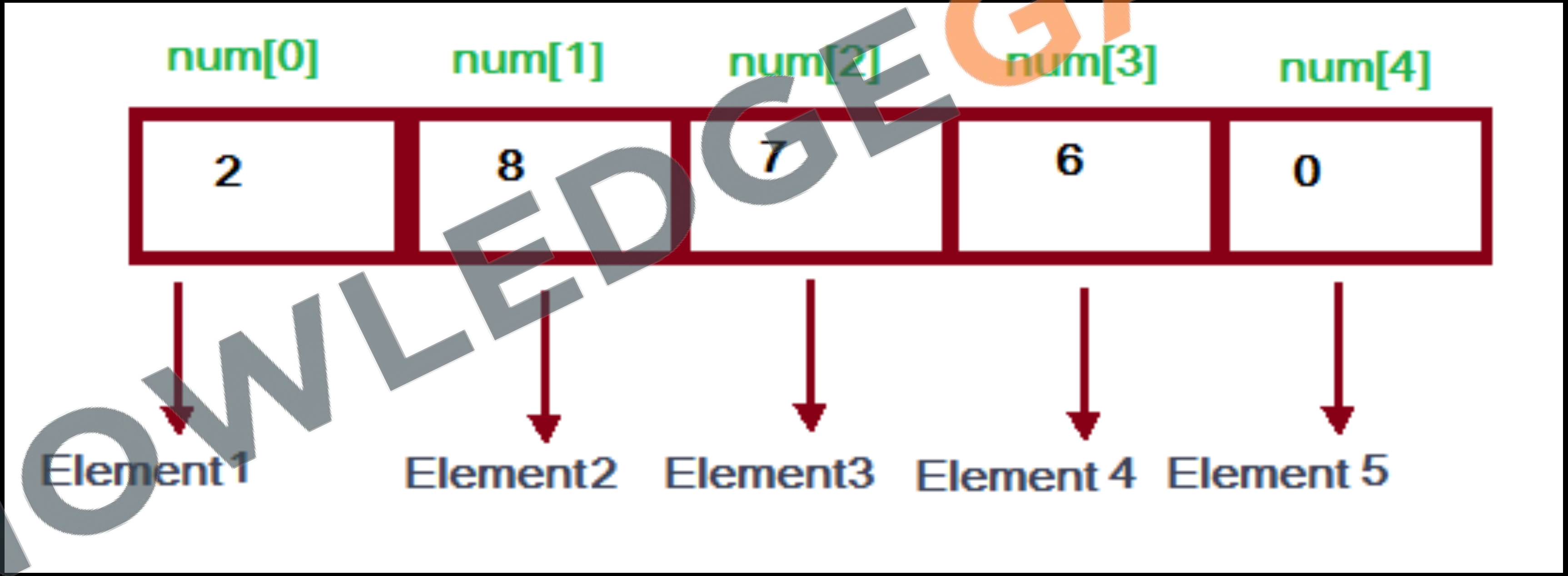
Example : Array Rupesh..........

http://www.knowledgegate.in/GATE

Array

- An array is a is a data structure that stores collection of elements of same type stored at contiguous memory locations and can be accessed using an index.

int num[5]; ⟶



How to declare an array in C

- datatype arrayName[array Size];
- int myarray[5];
- In C, the default value of the elements in an array is undefined or garbage. When an array is declared, the memory is allocated for the elements of the array, but the values of those elements are not initialized.
- It is important to note that some programming languages, like Java, automatically initialize the elements of an array to a default value (e.g., 0 for integers, false for booleans, and null for objects) if no initial values are specified.
- Note that in C, you cannot change the size of the array once it has been declared.

How to initialize an array in C

- dataType arrayName[arraySize] = {value$_1$, value$_2$, ..., value$_N$};

- int myarray[5] = {1, 2, 3, 4, 5};

- You can also initialize an array like this.
  - int myarray[] = {1, 2, 3, 4, 5};

- Here, we haven't specified the size. However, the compiler knows it’s size is 5 as we are initializing it with 5 elements.

Change Value of Array elements

- int myarray[5] = {1, 2, 3, 4, 5};
- make the value of the third element to -1
  - myarray[2] = -1;
- make the value of the fifth element to 0
  - myarray[4] = 0;

- Arrays have several advantages, like:
  - **Efficient storage and retrieval**: Arrays store elements in contiguous memory locations, which makes it easy to retrieve elements using their index. So very efficient with large amounts of data.
  - **Random access(fast access)**: Arrays allow access to individual elements using their index, which means that accessing any element of the array takes the same amount of time.
  - **Easy to sort and search**: Arrays can be easily sorted and searched using algorithms like binary search, which can be more efficient than searching through unsorted data.
  - **Flexibility**: Arrays can be used to represent a wide variety of data structures, including stacks, queues etc.
  - **Easy to use**: Arrays are a simple and easy-to-use data structure that can be easily understood by programmers of all skill levels.

- Arrays also have some disadvantages, likes:
  - **Fixed size**: In most programming languages, arrays have a fixed size that cannot be changed once they are created. This can make it difficult to work with data structures that need to grow or shrink dynamically**(Internal Fragmentation) (External Fragmentation)**.
  - **No built-in support for insertion or deletion**: Inserting or deleting an element in an array can be time-consuming and require shifting all the elements after the insertion or deletion point.
  - **Homogeneous elements**: Arrays can only store elements of the same type, which can be limiting for many requirements.
  - **Poor performance for some operations**: Some operations, such as searching or inserting elements in a sorted array, can have poor performance compared to other data structures like hash tables or binary search trees.

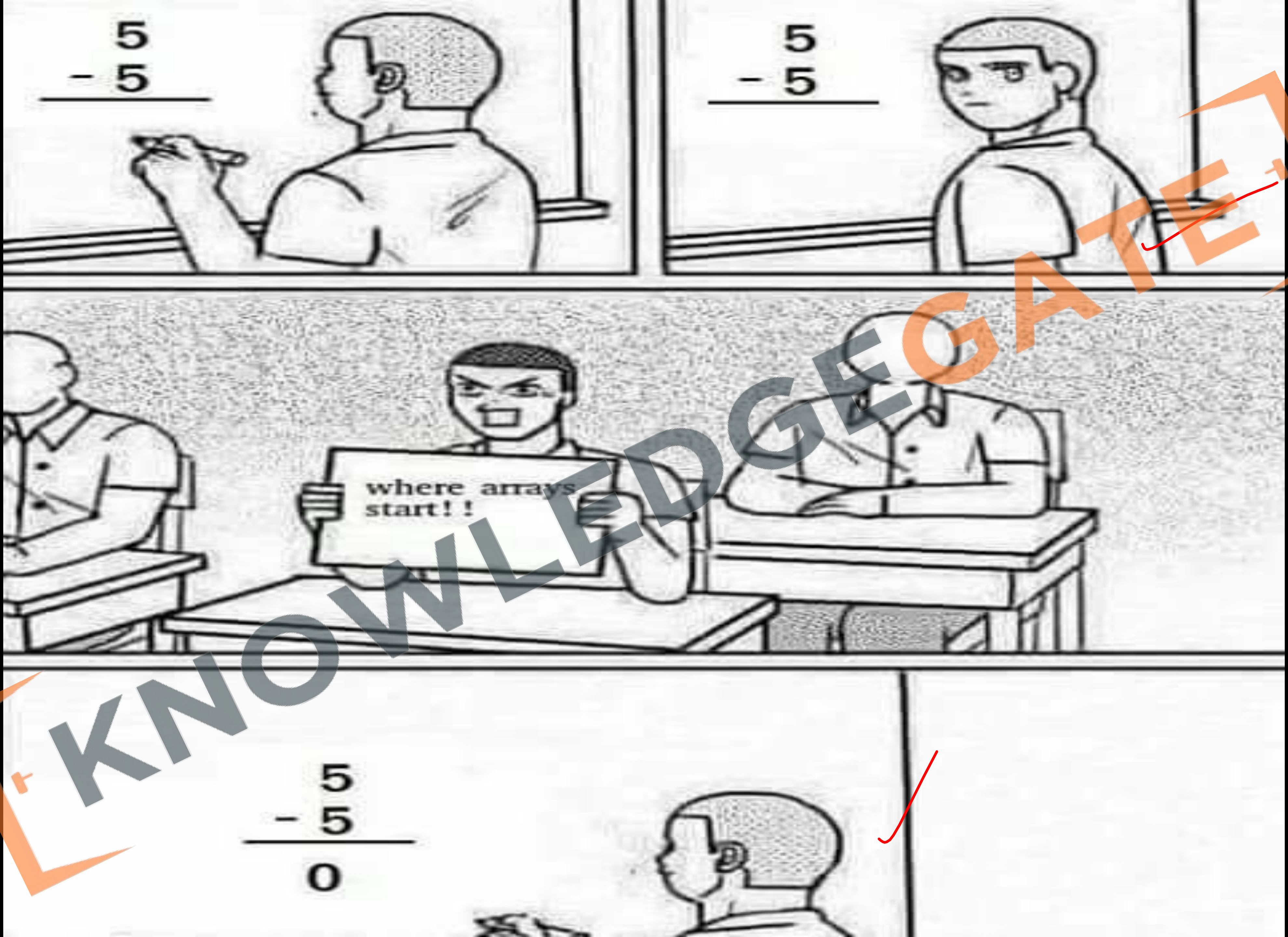
Applications of Arrays

- **Memory Management**: Arrays enable efficient storage of multiple items of the same type, especially when size is known beforehand.
- **Data Representation**: Used for vectors and matrices in mathematical operations like matrix multiplication.
- **Database Management**: Arrays store and manage datasets in relational databases, allowing efficient querying and updates.
- **Implementing Data Structures**: Arrays are foundational for structures like heaps, hash tables, and strings.
- **Caching & Buffering**: Arrays act as buffers in systems, storing data temporarily before writing to slower mediums or transmitting over networks.

We need to talk
OK
RUN CMD
C:\>_
You seem to have more time for your computer than me. I want to know how important I am to you
You're my number 1 in my life
Just that I start counting from 0

- Types of indexing in array:
  - 0 (zero-based indexing): The first element of the array is indexed by subscript of 0
  - 1 (one-based indexing): The first element of the array is indexed by subscript of 1
  - n (n-based indexing): The base index of an array can be freely chosen. Usually programming languages allowing n-based indexing also allow negative index values and other scalar data types like enumerations, or characters may be used as an array index.

-FATHER-
AS A FATHER, IT'S MY JOB TO RESPECT MY SON'S OPINION
UM... DAD?
YES, MY DEAR SON?
Arrays start at one
BANG! BANG! BANG

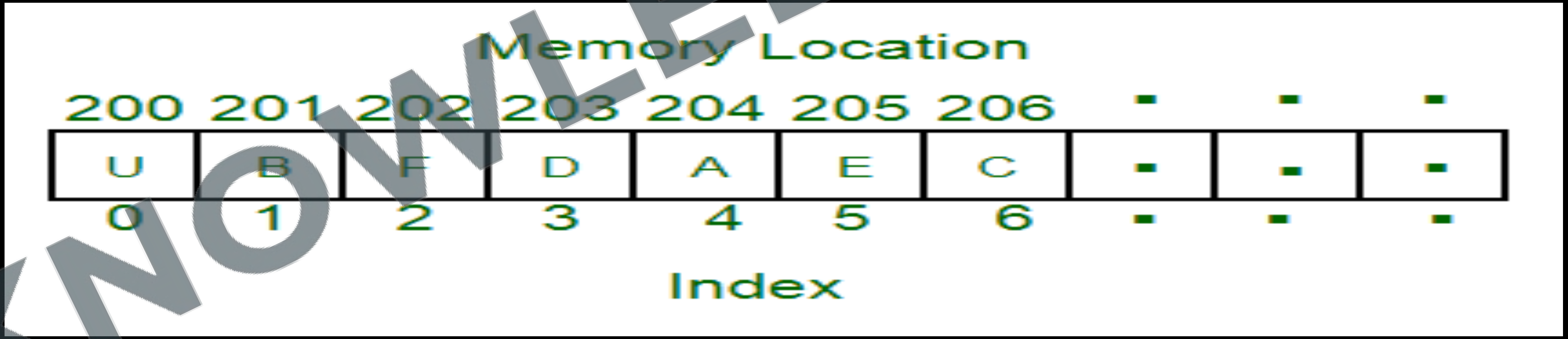

5
- 5
5
- 5
where arrays start!!
5
- 5
0

DONT SHOOT!
I AM A PROGRAMMER
PROVE IT!
ARRAYS START AT 1
imgflip.com

Size of an array

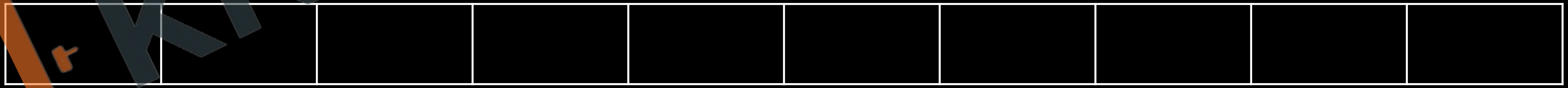
- Number of elements = (Upper bound – Lower Bound) + 1
  - Lower bound index of the first element of the array
  - Upper bound index of the last element of the array

- Size = number of elements * Size of each elements in bytes



One Dimensional array

- Address of the element at kth index
  - a[k] = B + W*k
  - a[k] = B + W*(k – Lower bound)
    - B is the base address of the array
    - W is the size of each element
    - K is the index of the element
    - Lower bound index of the first element of the array
    - Upper bound index of the last element of the array



Q Let the base address of the first element of the array is 250 and each element of the array occupies 3 bytes in the memory, then address of the fifth element of a one- dimensional array a[10] ?

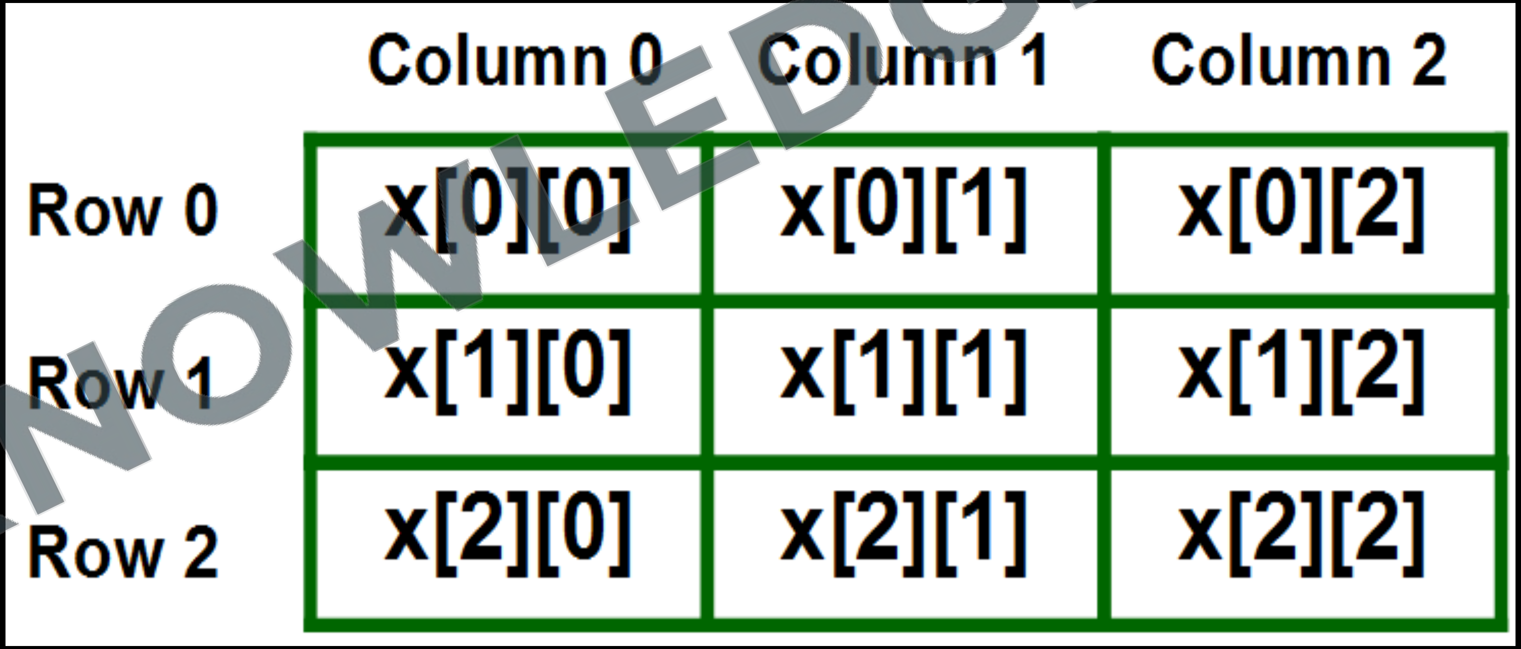
a[k] = B + W*(k – Lower bound)

Q An array has been declared as follows
A: array [-6--------6] of elements where every element takes 4 bytes, if the base address of the array is 3500 find the address of array[0]?

a[k] = B + W*(k – Lower bound)

Two-Dimensional array

- The two-dimensional array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.
- However, 2D arrays are created to implement a relational database look a like data structure. It provides ease of holding the bulk of data at once which can be passed to any number of functions wherever required.



	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

http://www.knowledgegate.in/GATE

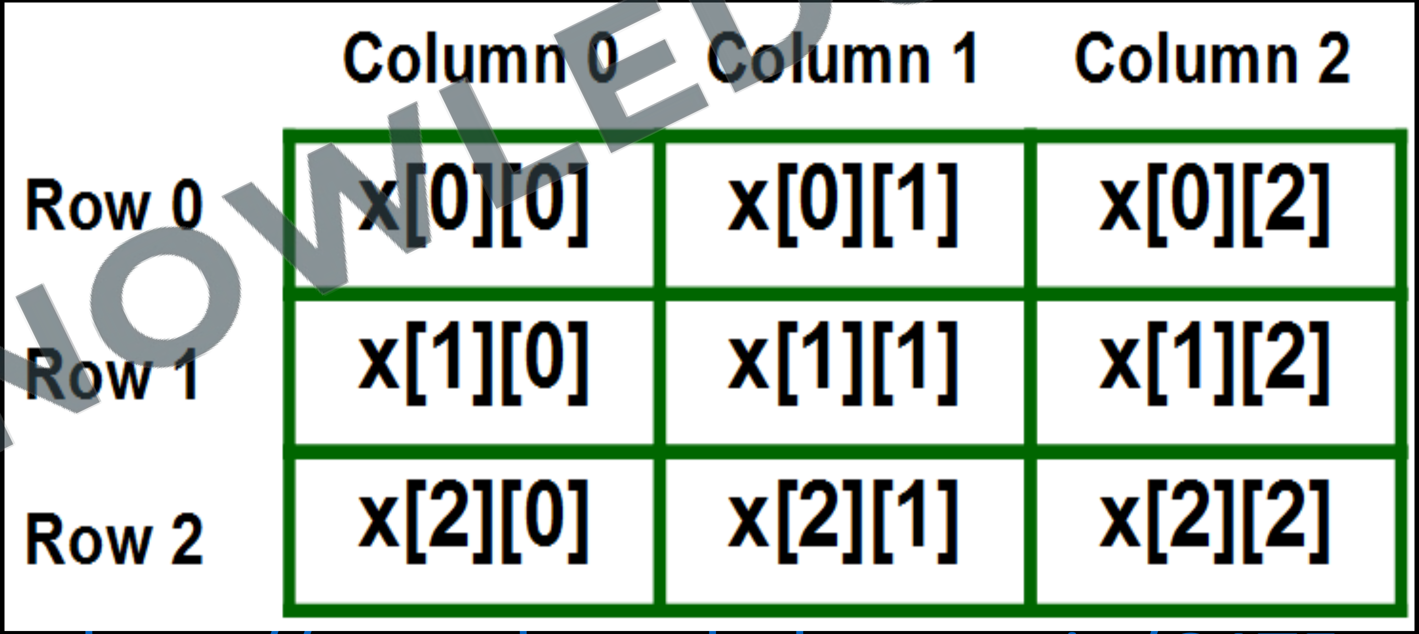
Two-Dimensional array

```
data_type array_name[rows][columns];

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

Implementation of 2D array

- In computing, row-major order and column-major order are methods for storing multidimensional arrays in linear storage such as random access memory.

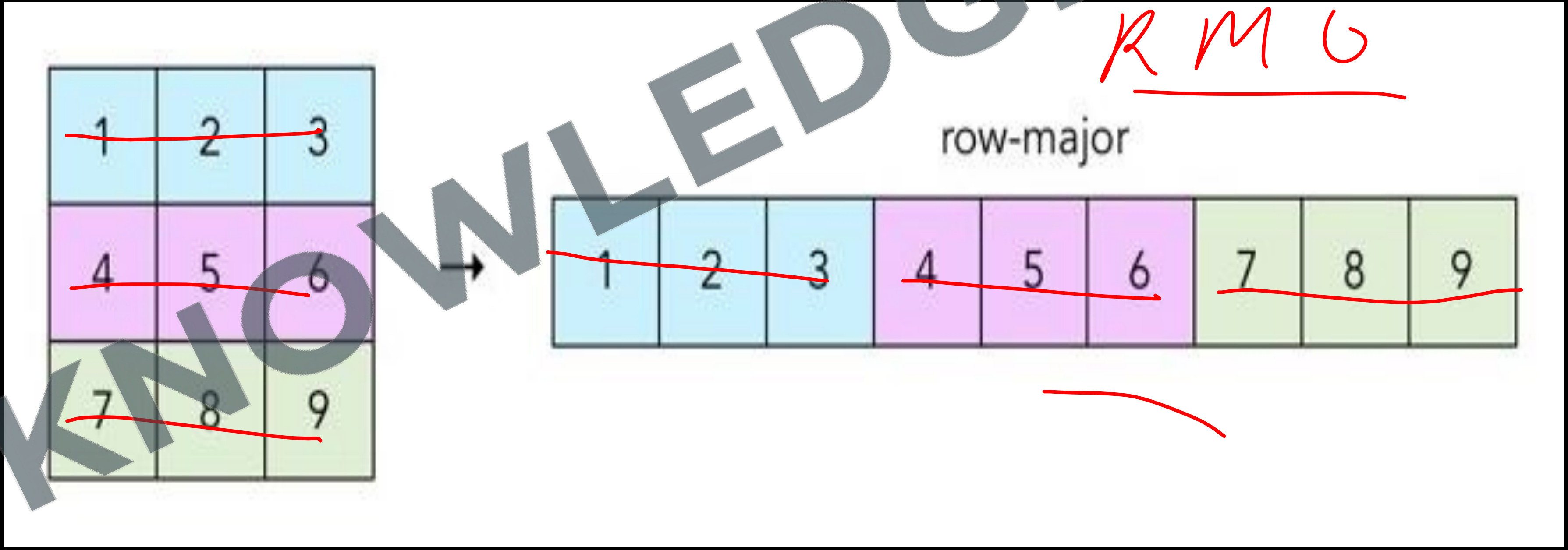
- The difference between the orders lies in which elements of an array are contiguous in memory.



$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Row Major implementation of 2D array

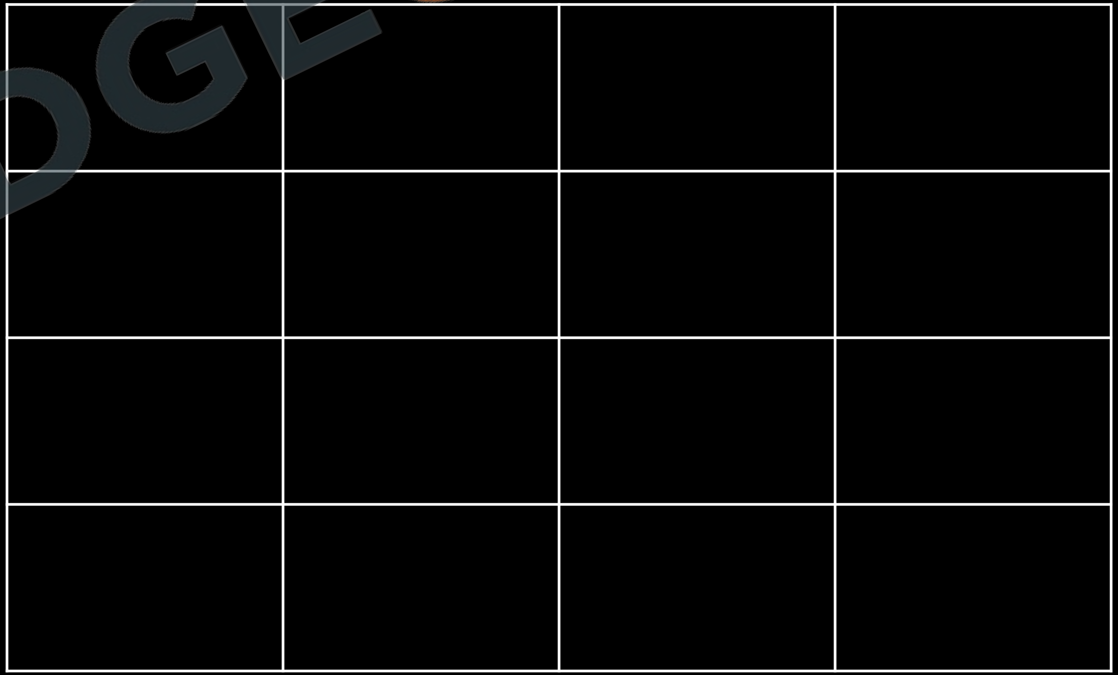
- In Row major method elements of an array are arranged sequentially row by row.
- Thus, elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.



http://www.knowledgegate.in/GATE

Row Major implementation of 2D array

$$\text{Address of } a[i][j] = B + W*[\ (U_2-L_2+1)\ (i-L_1) + (j-L_2)\ ]$$



B = Base address

W = Size of each element

L_1 = Lower bound of rows

U_1 = Upper bound of rows

L_2 = Lower bound of columns

U_2 = Upper bound of columns

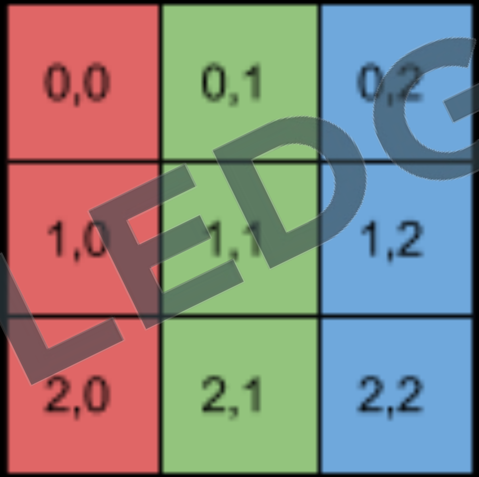
(U_2-L_2+1) = numbers of columns

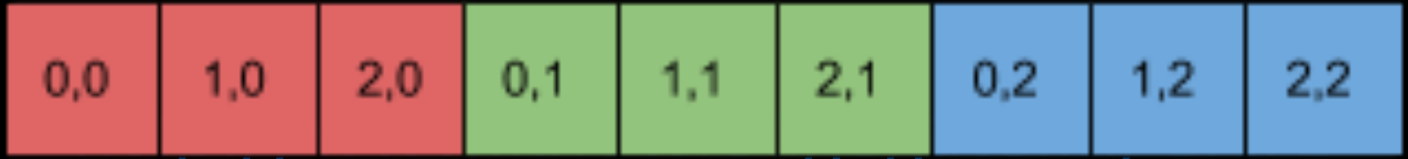
$(i-L_1)$ = number of rows before us

$(j-L_2)$ = number of elements before us in current row

Column Major implementation of 2D array

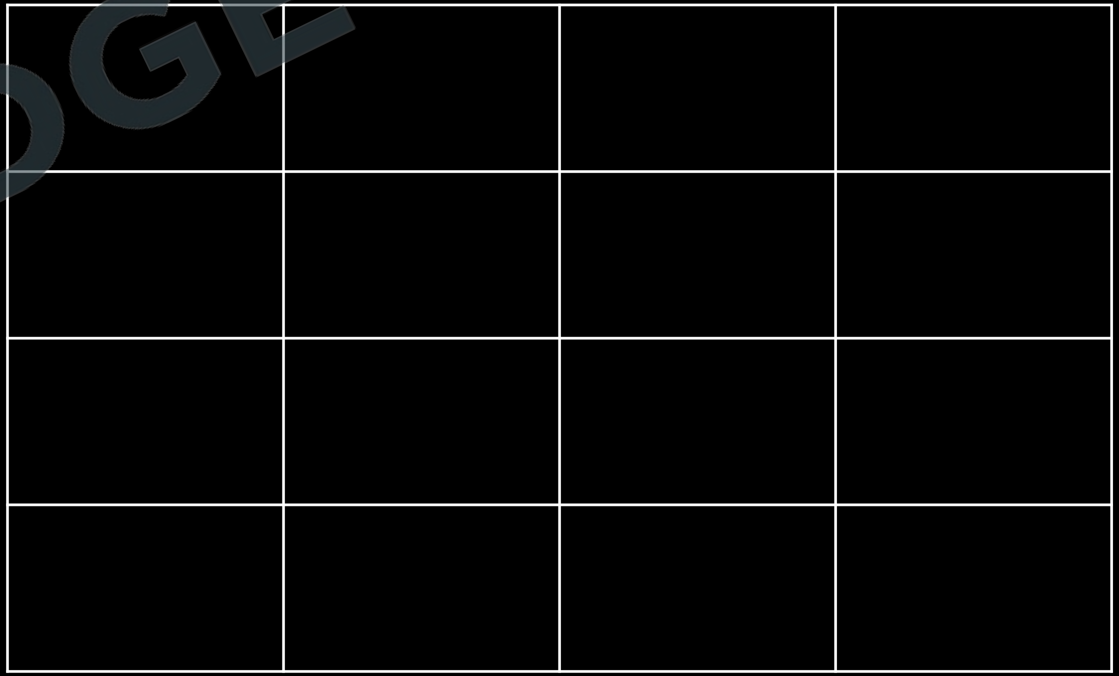
- In Column major method elements of an array are arranged sequentially column by column. Thus, elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on.





Column Major implementation of 2D array

Address of a[i][j] = B + W*[(U_1-L_1+1) $(j-L_2)$ + $(i-L_1)$]



B = Base address

W = Size of each element

L_1 = Lower bound of rows

U_1 = Upper bound of rows

L_2 = Lower bound of columns

U_2 = Upper bound of columns

(U_1-L_1+1) = numbers of rows

$(j-L_2)$ = number of columns before us

$(i-L_1)$ = number of elements before us in current column

Q An array VAL[1…15][1…10] is stored in the memory with each element requiring 4 bytes of storage. If the base address of the array VAL is 1500, determine the location of VAL[12][9] when the array VAL is stored

(i) Row wise

Address of a[i][j] = B + W*[$(U_2-L_2+1)(i-L_1) + (j-L_2)$]

(ii) Column wise

Address of a[i][j] = B + W*[$(U_1-L_1+1)(j-L_2) + (i-L_1)$]

3-Dimensional array

$A([L_1]---[U_1]), ([L_2]---[U_2]), ([L_3]---[U_3])$

Location of A [I, j, k] =

$$B + (i-L_1)(U_2-L_2+1)(U_3-L_3+1) + (j-L_2)(U_3-L_3+1) + (k-L_3)$$

http://www.knowledgegate.in/GATE

Q Suppose multidimensional arrays *Q is* declared *Q*(1: 8, – 5: 5, – 10 : 5) stored in column major order

- Find the length of each dimension of *Q*.
- The number of elements in *Q*.
- Assuming base address (*Q*) = 400, *W* = 4, find the effective indices *E*1, *E*2, *E*3 and address of the element *Q*[3, 3, 3].

$$A\ [I, j, k] = B + (i-L_1)\ (U_2-L_2+1)\ (U_3-L_3+1) + (j-L_2)(U_3-L_3+1) + (k-L_3)$$

N-Dimensional array

$A([L_1]---[U_1]), ([L_2]---[U_2]), ([L_3]---[U_3]), ([L_4]---[U_4])--------([L_N]---[U_N])$

Location of A [I, j, k, ----, x] =

$B + (i-L_1)(U_2-L_2+1)(U_3-L_3+1)(U_4-L_4+1) ----(U_n-L_n+1)$

$+ (j-L_2)(U_3-L_3+1)(U_4-L_4+1) ----(U_n-L_n+1)$

$+ (k-L_3)(U_4-L_4+1) ----(U_n-L_n+1)$

$+$

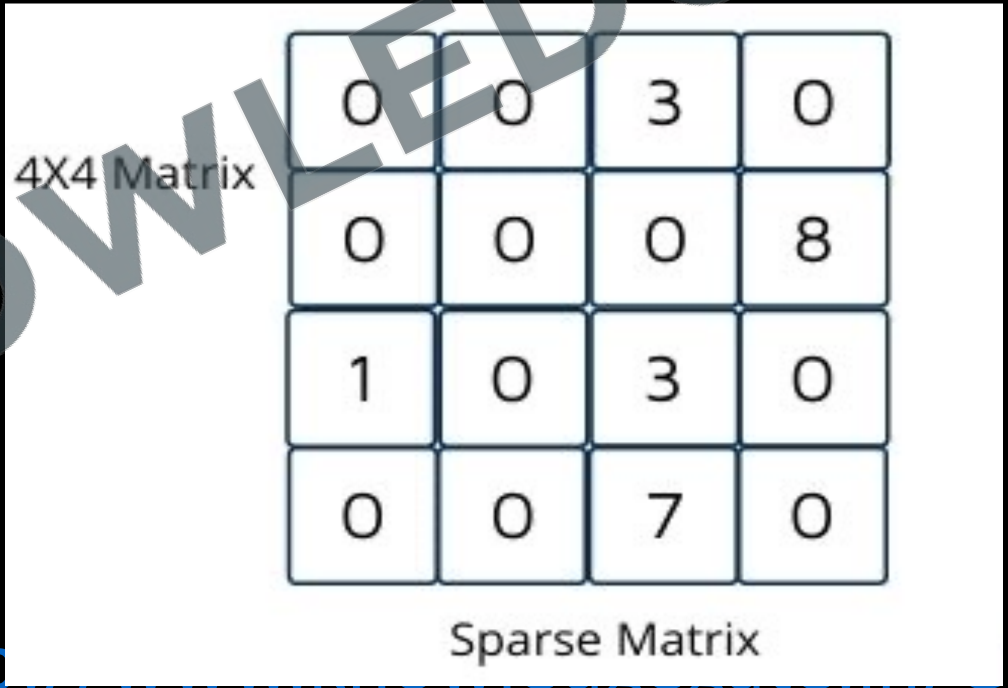
$+$

$+$

$+ (x-L_n)$

Sparse Matrix

- A matrix is considered sparse if a large number of its elements are zero Conversely, a matrix with most of its elements being non-zero is termed dense.
- Using a sparse matrix over a regular matrix has distinct advantages:
    - **Storage Efficiency**: Given that a majority of the elements are zeros, sparse matrices allow for memory conservation by only storing the non-zero elements.
    - **Computational Speed**: By structuring the data to only account for non-zero elements, operations become faster, as they skip over the zero values.

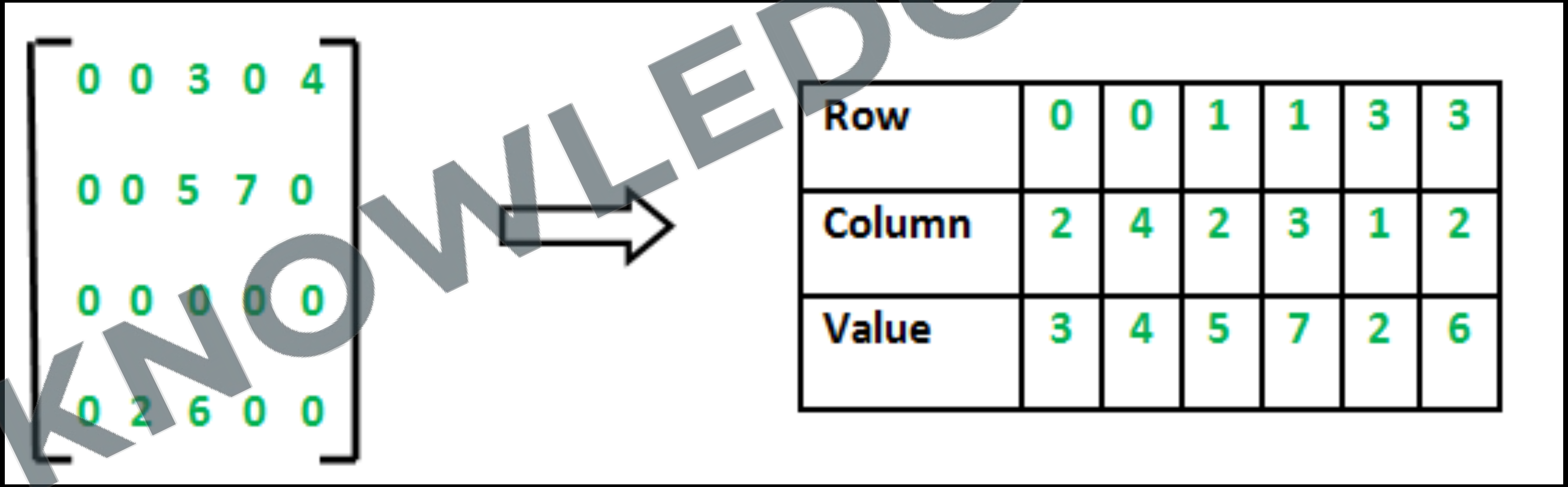

4X4 Matrix
Sparse Matrix

0	0	3	0
0	0	0	8
1	0	3	0
0	0	7	0

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

Array representation

- 2D array is used to represent a sparse matrix in which there are three rows named as
    - **Row:** Index of row, where non-zero element is located
    - **Column:** Index of column, where non-zero element is located
    - **Value:** Value of the non zero element located at index – (row, column)

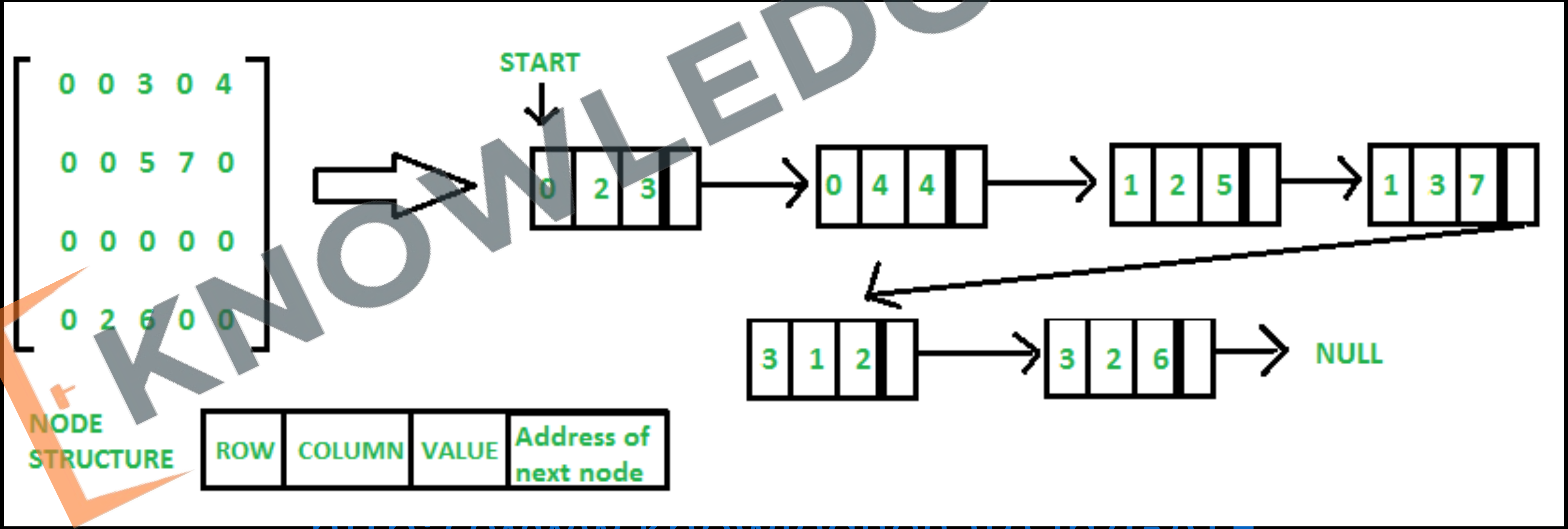


$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Linked List Representation

- In linked list, each node has four fields. These four fields are defined as: **Row:** Index of row, where non-zero element is located
  - **Column:** Index of column, where non-zero element is located
  - **Value:** Value of the non zero element located at index – (row,column)
  - **Next node:** Address of the next node

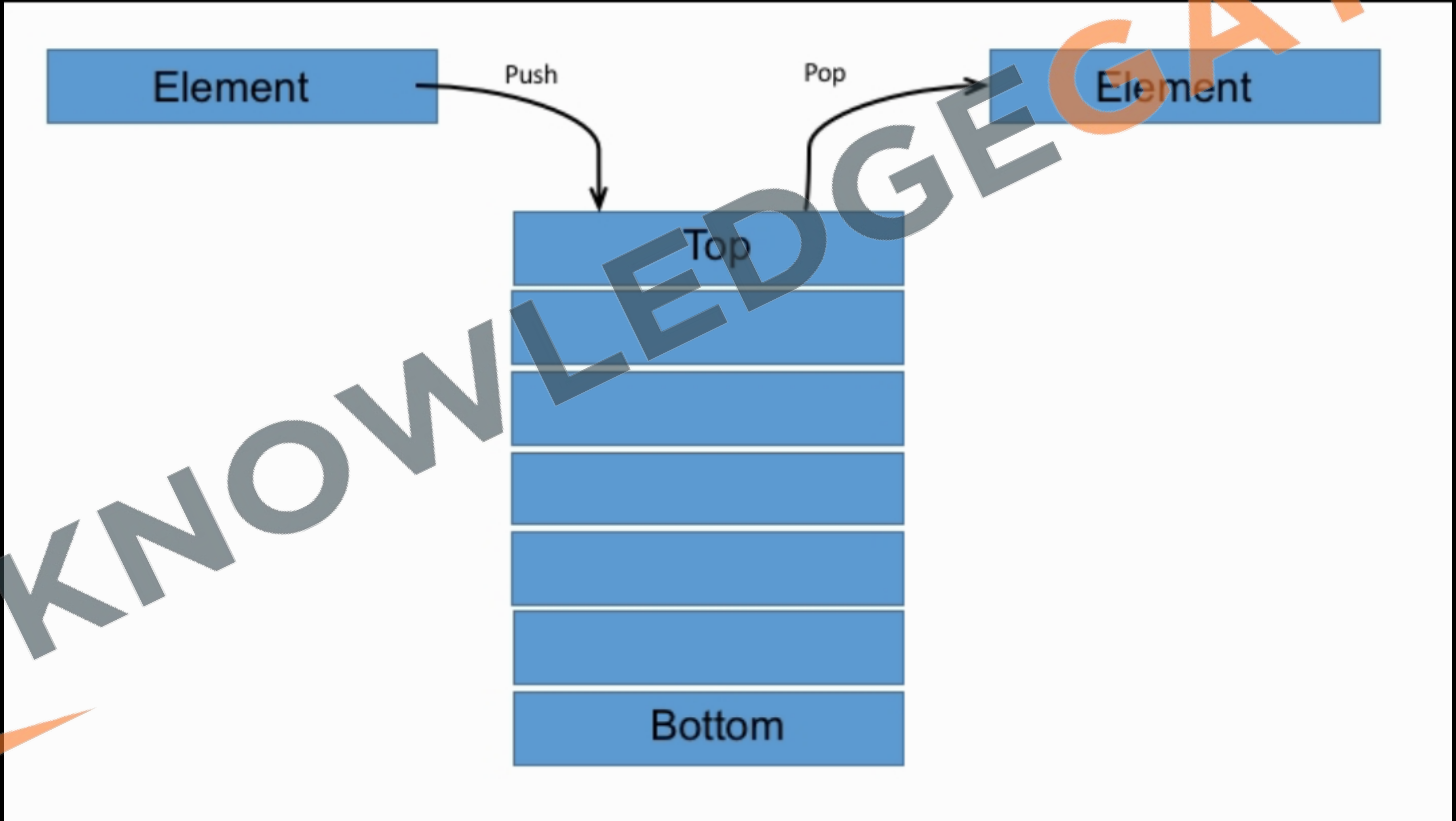


Basics of Stack

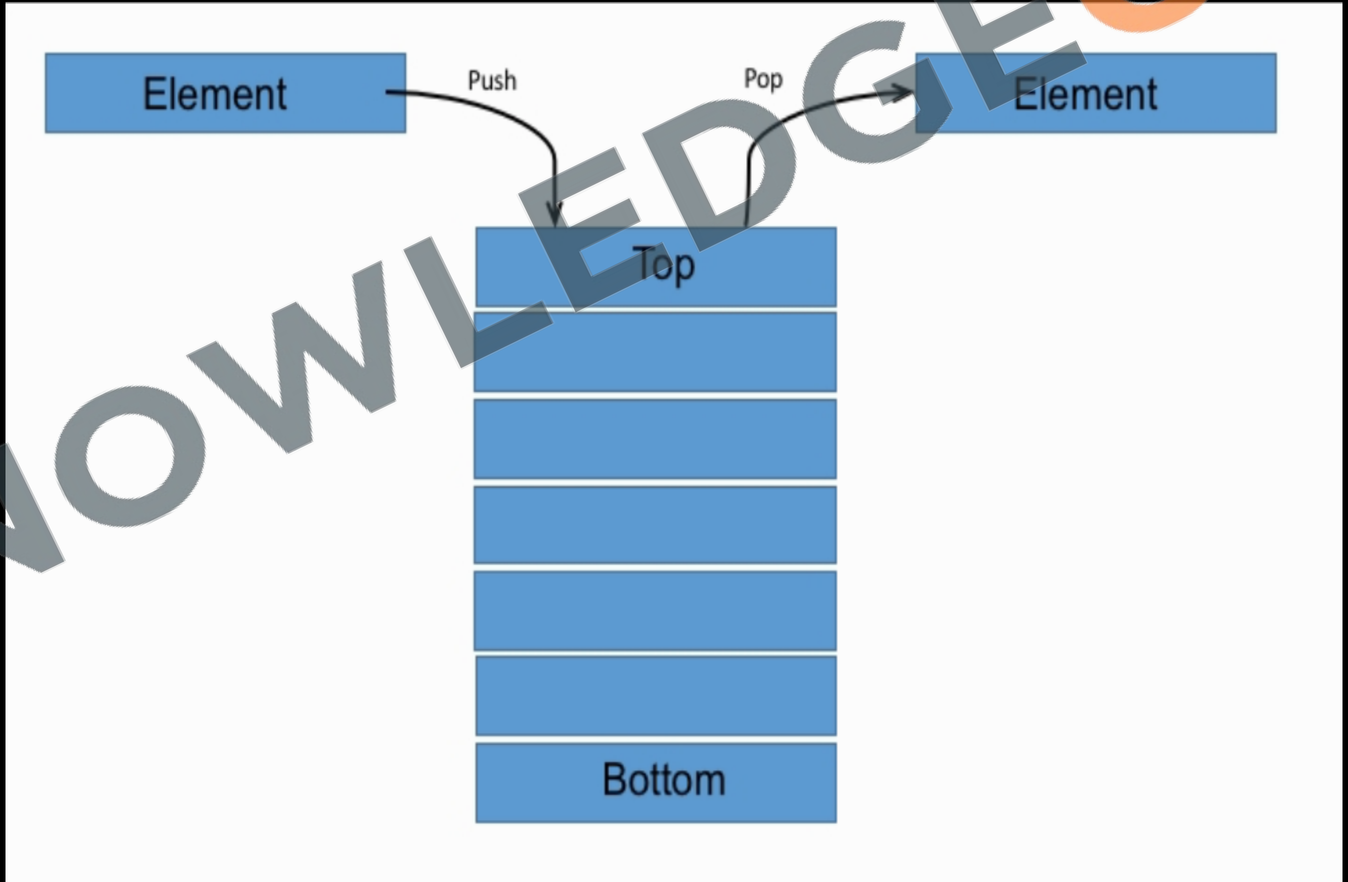
Basics of Stack

STACK

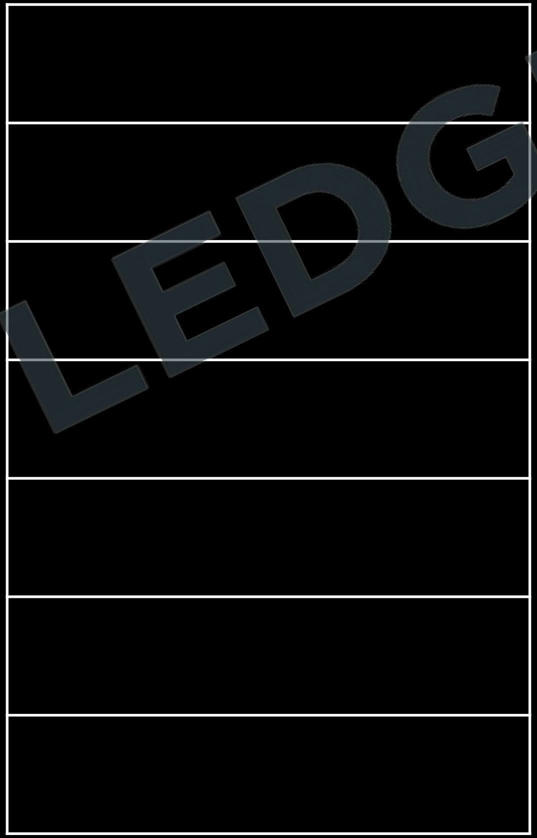
- A stack is a non-primitive linear data structure. it is an ordered list in which addition of a new data item and deletion of already existing data item is done from only one end known as top of stack (TOS).



- The element which is added in last will be first to be removed and the element which is inserted first will be removed in last.
- That is why it is called last in first out (LIFO) or first in last out (FILO) type of list.
- Most frequently accessible element in the stack is the top most element, whereas the least accessible element is the bottom of the stack.



Q Choose the correct alternatives (more than one may be correct) and write the corresponding letters only: The following sequence of operations is performed on a stack: PUSH (10), PUSH (20), POP, PUSH (10), PUSH (20), POP, POP, POP, PUSH (20), POP The sequence of values popped out is ?



http://www.knowledgegate.in/GATE

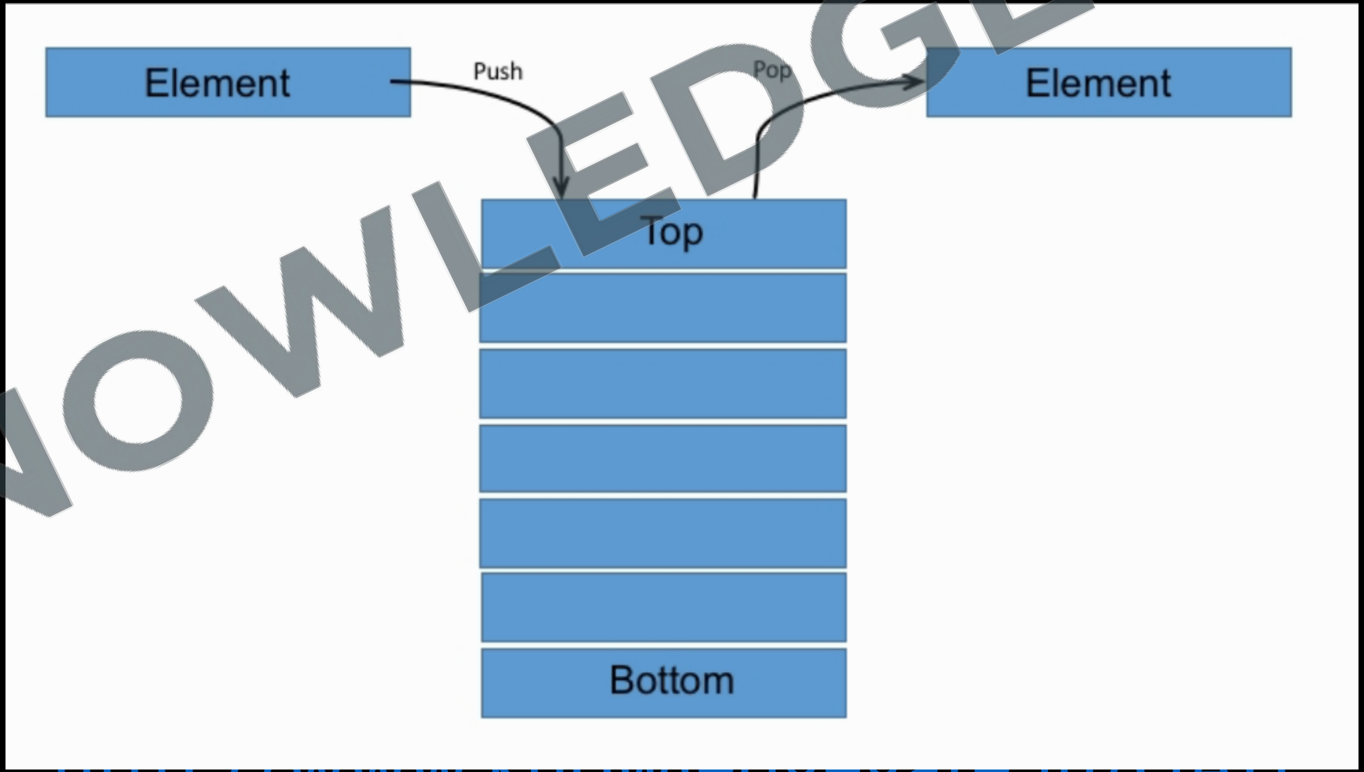
Applications of Stack

- **Expression Parsing:** Stacks help evaluate and check programming expressions, ensuring balanced parentheses.
- **Backtracking**: Used in algorithms like maze-solving and the "Eight Queens" puzzle.
- **Function Calls**: Manage function details during calls in programming languages.
- **Undo Feature**: Implement undo in text editors and browsers.
- **Syntax Checking**: Compilers use stacks to match syntax elements like 'if' with 'else'.

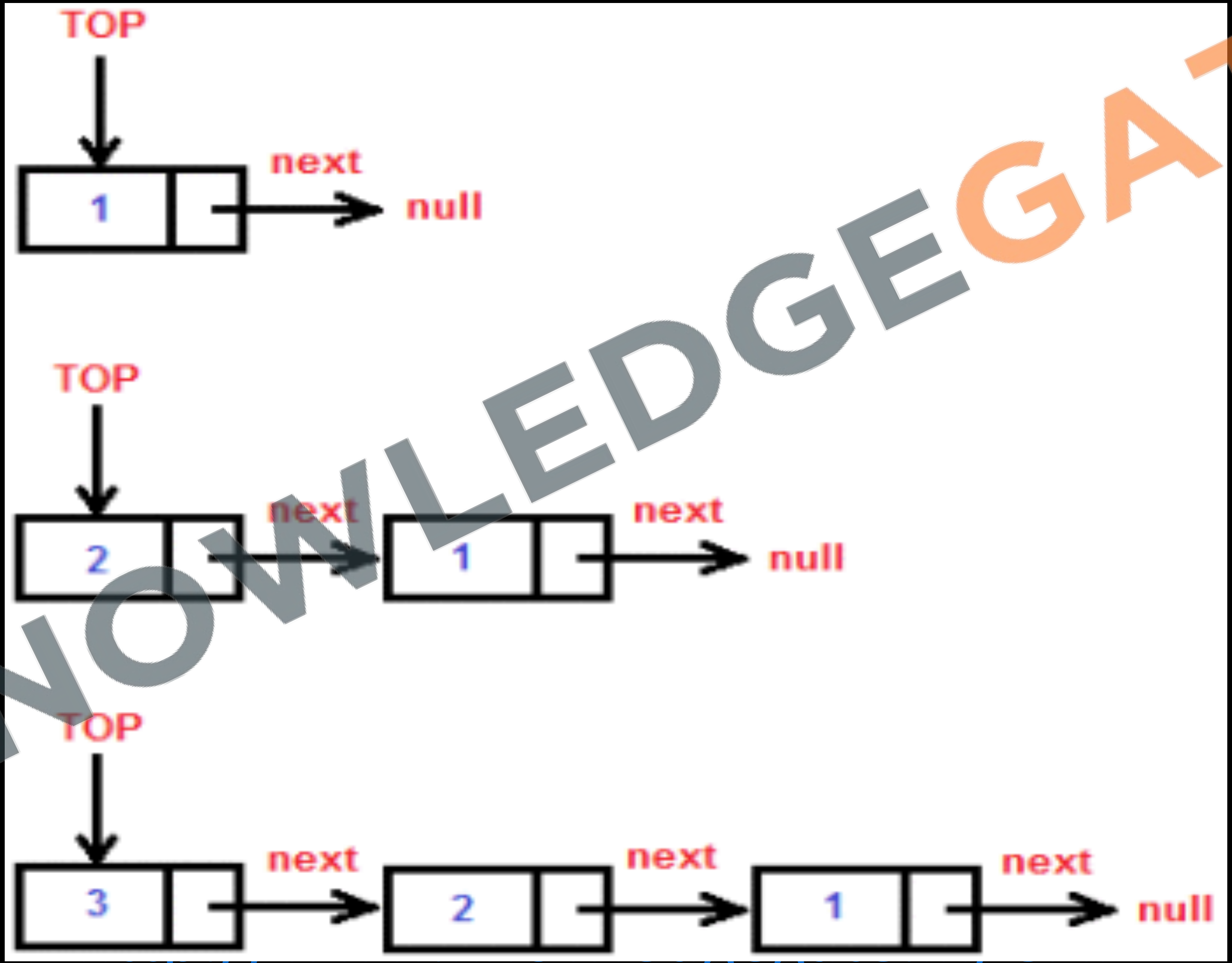
Stack Implementation

Stack is generally implemented in two ways.

- **Static Implementation**: - Here array is used to create stack. it is a simple technique but is not a flexible way of creation, as the size of stack has to be declared during program design, after that size implementation is not efficient with respect to memory utilization.

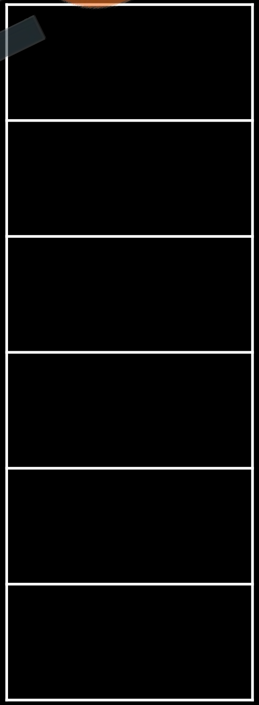


- **Dynamic implementation**: - It is also called linked list representation and uses pointer to implement the stack type of data structure.



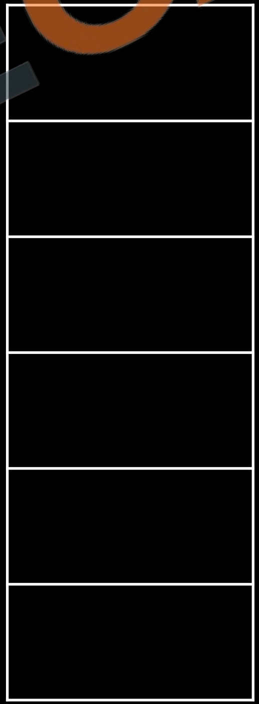
Push operation: - The process of adding new element to the top of stack is called push operation. the new element will be inserted at the top after every push operation the top is incremented by one. in the case the array is full and no new element can be accommodated it is called over-flow condition.

```
PUSH (S, N, TOP, x)
{
    if (TOP==N-1)
        Print stack overflow and exit
    TOP = TOP + 1
    S[TOP] = x
    exit
}
```



Pop: - The process of deleting an element. from the top of stack is called POP operation, after every POP operation the stack is decremented by one if there is no element in the stack and the POP operation is requested then this will result into a stack underflow condition.

```
POP (S, N, TOP)
{
    if (TOP==-1)
        print underflow and exit
    y = S[TOP]
    TOP=TOP-1
    return(y) and exit
}
```



```
typedef struct
{
    int arr[MAX_SIZE];
    int top;
} Stack;



void initialize(Stack *s)
{
     s->top = -1;
}
```

```
int isEmpty(Stack *s)
{
    return s->top == -1;
}



int isFull(Stack *s)
{
    return s->top == MAX_SIZE - 1;
}
```

```
void push(Stack *s, int item)
{
    if (isFull(s))
    {
        printf("Stack is full!\n");
        return;
    }
     s->arr[++(s->top)] = item;
}
```

```
int pop(Stack *s)
{
    if (isEmpty(s))
    {
        printf("Stack is empty!\n");
        exit(1);
    }
     return s->arr[(s->top)--];
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
int main()
{
    Stack s;
    initialize(&s);
    return 0;
}
```

```
typedef struct Node
{
    int data;
    struct Node* next;
} Node;

typedef struct
{
    Node* top;
} Stack;
```

```c
void initialize(Stack* s)
{
    s->top = NULL;
}

int isEmpty(Stack* s)
{
    return s->top == NULL;
}
```

```
void push(Stack* s, int item)
{
    Node* newNode = (Node*)
    malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Stack overflow!\n");
        exit(1); // Exit with an error code
    }
    newNode->data = item;
    newNode->next = s->top;
    s->top = newNode;
}
```

```
int pop(Stack* s)
{
    if (isEmpty(s))
    {
        printf("Stack underflow!\n");
        exit(1);
    }
    Node* temp = s->top;
    int poppedData = temp->data;
    s->top = s->top->next;
    free(temp);
    return poppedData;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    Stack s;
    initialize(&s);




    return 0;
}
```

```
void reverseString(char str[])
{
    int length = strlen(str);
    Stack s;
    initialize(&s);
    for (int i = 0; i < length; i++)
    {
        push(&s, str[i]);
    }
    for (int i = 0; i < length; i++)
    {
        str[i] = pop(&s);
    }
}
```

```c
int main()
{
    char str[] = "Hello, World!";
    printf("Original String: %s\n", str);
    reverseString(str);
    printf("Reversed String: %s\n", str);
    return 0;
}
```

Q if the input sequence is 1, 2, 3, 4, 5 then identify the wrong stack permutation (possible pop sequence)?

a) 3, 5, 4, 2, 1

b) 2, 4, 3, 5, 1

c) 4, 3, 5, 2, 1

d) 5, 4, 3, 1, 2

- **Infix notation**: the operator is written in between the operands. e.g. A+B. the reason why this notation is called infix is the place of operator in the expression.

- **Prefix notation:** In which the operator is written before the operands it is also called as polish notation. e.g. +AB

- **Postfix:** In the postfix notation the operator are written after the operands, so it is called the postfix notation. It is also known as suffix notation or reverse polish notation. AB+

- The description "Polish" refers to the nationality of logician Jan Łukasiewicz, who invented Polish notation in 1924.

http /GATE

- Postfix notation is type of notation which is most suitable for a computer to calculate any expression. It is universally accepted notation for designing arithmetic and logical unit (ALU) of the CPU.

- Any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

Q Consider an expression a + b * c / d ^ e ^ f * d – c + b, convert it into both prefix and post fix notation?

Q Consider an expression log(x!), convert it into both prefix and post fix notation?

Let Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

```
1.  Push "("onto STACK, and add ")" to end of $Q$.
2.  Scan $Q$ from left to right and repeat steps 3 to 6 for each element of $Q$
    until the STACK is empty.
3.  If an operand is encountered, add it to $P$.
4.  If a left parenthesis is encountered push it onto STACK.
5.  If an operator $\otimes$ is encountered, then :
    a.  Repeatedly pop from STACK and add to $P$ each operator (on the top
        of STACK) which has the same precedence as or higher precedence
        than $\otimes$.
    b.  Add $\otimes$ to STACK.
        [End if]
6.  If a right parenthesis is encountered, then :
    a.  Repeatedly pop from STACK and add to $P$ each operator (on the top
        of STACK) until a left parenthesis is encountered.
    b.  Remove the left parenthesis [Do not add it to $P$]
        [End if]
        [End of step 2]
7.  End.
```

(A + (B *C + D)/E)

Character	Stack	Postfix
(	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
+	(+(+	ABC*
D	(+(+	ABC*D
)	(+	ABC*D+
/	(+/	ABC*D+
E	(+/	ABC*D+E
)	(	ABC*D+E/+

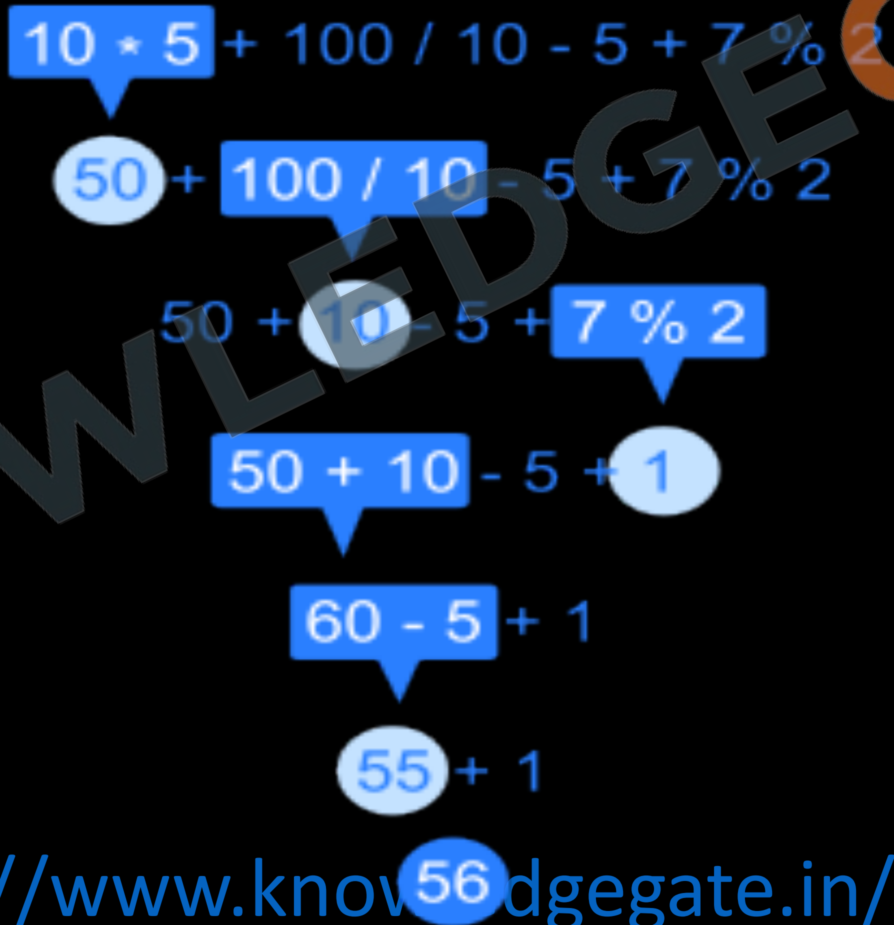
$A + (B*C - (D/E \wedge F)*H)$

Character	Stack	Postfix
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	AB
*	(+ (*	AB
C	(+ (*	ABC
–	(+ (– (	ABC*
(	(+ (– (	ABC*
D	(+ (– (	ABC*D
/	(+ (– (/	ABC*D
E	(+ (– (/	ABC*DE
^	(+ (– (/^	ABC*DE
F	(+ (– (/^	ABC*DEF
)	(+ (– (/^	ABC*DEF
*	(+ (– *	ABC*DEF ^/
H	(+ (– *	ABC*DEF ^/ H

Q Consider an expression a + (b * c) / (d ^ e) convert it into post fix notation using stack?

Evaluation of arithmetic expression

- An expression is defined as a number of operands or data items combined using several operators. There are basically three of notation to represent an expression.



Q The result evaluating the postfix expression

8 2 3 * 1 / + 4 1 * 2 / +

Q The result evaluating the prefix expression

$$+ + 8 / * 2 3 1 / * 4 1 2$$

```
1.  Add a right parenthesis ")" to $P$.
    [This acts as a sentinel]
2.  Scan $P$ from left to right and repeat step 3 and 4 for each element of
    $P$ until the sentinel ")" is encountered.
3.  If an operand is encountered, put it on STACK.
4.  If an operator $\otimes$ is encountered then :
    a.  Remove the top two elements of STACK, where $A$ is the top element
        and $B$ is the next-to-top element.
    b.  Evaluate $B \otimes A$.
    c.  Place the result of (b) back on STACK.
        [End of if structure]
        [End of step 2 loop]
5.  Set value equal to top element on STACK.
6.  End.
```

Recursion

- Recursion is defined as defining anything in terms of itself Recursion is a programming concept where a function calls itself in order to solve a larger problem by breaking it down into smaller, more manageable sub-problems. It's a fundamental idea in computer science and mathematics and is used to design algorithms and solve problems that have repetitive structures.
- **Base Case**: Essential to halt recursion. It provides a direct solution without further recursive calls.
- **Recursive Case**: The function calls itself to address smaller instances of the problem.
- **Call Stack**: Each recursive call is added to the program's call stack. Deep recursion might cause a "stack overflow" error.

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1;  // Base case: factorial of 0 is 1
    }
    else
    {
        return n * factorial(n-1);  // Recursive case
   }
}
```

Iteration

- Iteration refers to the process of repeatedly executing a set of statements as long as a specified condition remains true. In programming, iteration is commonly implemented using loops.

- **Loop Types**:
  - **For Loop**: Used for a known number of repetitions.
  - **While Loop**: Runs as long as a condition is true.
  - **Do-While Loop**: Executes at least once before checking the condition.

- **Control Statements**:
  - **Break**: Exits the loop.
  - **Continue**: Skips to the next iteration.

- **Nested Loops**: Loops within loops, often seen in matrix tasks.

```
int factorial(int n)
{
    int result = 1;
    for(int i = 1; i <= n; i++)
    {
        result *= i;
    }
    return result;
}
```

Recursion vs. Iteration

- Any problem that can be solved recursively can also be solved iteratively (using loops), and vice versa. However, the choice between the two often depends on problem characteristics, readability, and efficiency considerations.

Aspect	Recursion	Iteration
Basic Concept	Function calls itself to solve sub-problems.	Uses loops to repeatedly execute code blocks.
Memory Usage	Typically uses more memory due to call stack.	Uses less memory as it doesn't rely on the call stack.
Termination	Requires a base case to prevent infinite loops.	Requires a loop exit condition.
Ease of Implementation	Can be more intuitive for certain problems.	Often simpler and more straightforward for repetitive tasks.
Performance	Might be slower due to overhead of function calls.	Typically faster due to direct loop mechanics.

- Recursion can be categorized based on how and where functions call themselves or other functions. Here's a succinct breakdown of the types of recursion:

- **Direct Recursion**:
  - A function calls itself directly.
  - **Tail Recursion**: If the recursive call is the last operation in the function, before it returns a value. It's more memory efficient since it can be optimized by the compiler to use constant stack space.
  - **Head Recursion**: If the recursive call is made before any other operation in the function. The operations are executed after the recursive call, which makes it use more stack space compared to tail recursion.

- **Indirect Recursion**:
  - Two or more functions call each other in a cyclic manner. For example, function A calls function B, and function B calls function A.

Q Find the output of the following pseudo code?

```
Void main()
{
    fun(4);
}

Void fun(int x)
{
    if (x > 0)
    {
        Printf("%d", x);
        fun(x - 1);
    }
}
```

Q Find the output of the following pseudo code?

```
Void main()
{
    fun(4);
}

Void fun(int x)
{
    if (x > 0)
    {
        fun(x - 1);
        Printf("%d", x);
    }
}
```

Q Find the output of the following pseudo code?

```
Void main()
{
    fun(3);
}

Void fun(int x)
{
    if (x > 0)
    {
        Printf("%d", x);
        fun(x - 1);
        Printf("%d", x);
        fun(x - 1);
        Printf("%d", x);
    }
}
```

Q Find the output of the following pseudo code on n = 5?

```
int x(int n)
{
    if (n < 3)
        return 1;
    Else
        return x(n-1) + x(n-1) + 1;
}
```

Q Consider the following recursive C function. If get (5) function is being called in main () then how many times will the get () function be invoked before returning to the main ()?

```
void get (int n)
{
    if (n < 1)
        return;
    get(n-1);
    get(n-3);
    printf ("%d", n);
}
```

- In mathematics, the **Fibonacci numbers**, commonly denoted F_n, form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1.
  - if n==0, then f(n) = 0
  - if n==1, then f(n) = 1
  - if n > 1, then f(n-1) + f(n-2)

- if n==0, then f(n) = 0
- if n==1, then f(n) = 1
- if n > 1, then f(n-1) + f(n-2)

n	0	1	2	3	4	5	6	7	8	9	10	11	12
f(n)													
No of invocation													
No of addition													

no of invocation = $2f(n+1) - 1$

no of addition = $f(n+1) – 1$

- Fibonacci numbers are named after Italian mathematician Leonardo of Pisa, later known as Fibonacci. In his 1202 book *Liber Abaci*, Fibonacci introduced the sequence to Western European mathematics

- Although the sequence had been described earlier in Indian mathematics,as early as 200 BC in work by Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths.

http://wv e.in/GATE

- Fibonacci numbers appear unexpectedly often in mathematics, so much so that there is an entire journal dedicated to their study, the *Fibonacci Quarterly*.

- Applications of Fibonacci numbers include computer algorithms such as the Fibonacci search technique and the Fibonacci heap data structure, and graphs called Fibonacci cubes used for interconnecting parallel and distributed systems.

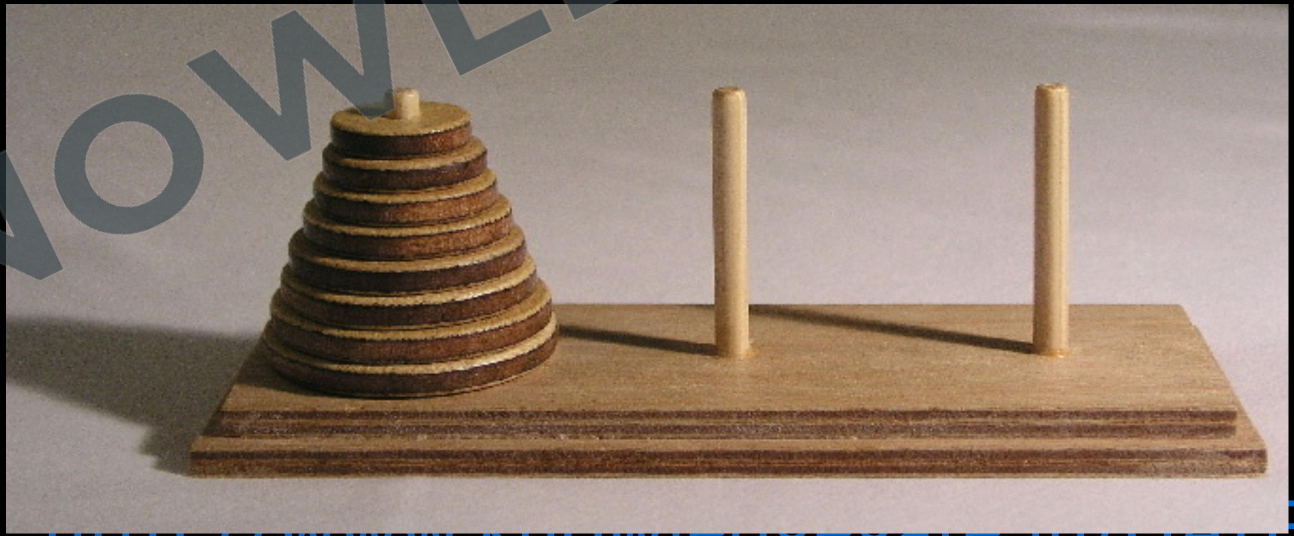
- They also appear in biological settings, such as branching in trees, the arrangement of leaves on a stem, the fruit sprouts of a pineapple, the flowering of an artichoke, an uncurling fern, and the arrangement of a pine cone's bracts.

- Although the sequence had been described earlier in Indian mathematics, as early as 200 BC in work by Acharya Pingala on enumerating possible patterns of Sanskrit poetry formed from syllables of two lengths.
- binary numeral system
- binomial theorem
- Pascal's triangle
- zero

http://www.knowledgegate.in/GATE

Tower of hanoi

- The **Tower of Hanoi** (also called the **Tower of Brahma**) is a mathematical game or puzzle.
- It consists of three rods and a number of disks of different sizes, which can slide onto any rod.
- The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.



Torres de Hanoi
GAP

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.

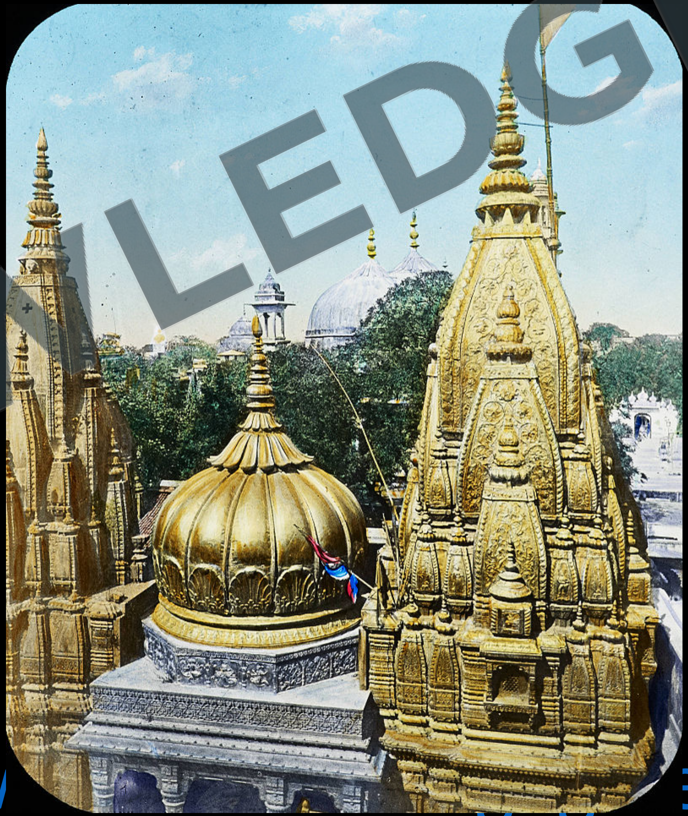
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.

3. No larger disk may be placed on top of a smaller disk.

- There is a story about an Indian temple in Kashi Vishwanath which contains a large room with three time-worn posts in it, surrounded by 64 golden disks.

- Brahmin priests, acting out the command of an ancient prophecy, have been moving these disks in accordance with the immutable rules of Brahma since that time.

- The puzzle is therefore also known as the Tower of Brahma puzzle.



http://v e.in/GATE

```
Tower(N, B, A, E)
{
    if(n = 1)
    {
        B → E
        return
    }
    tower(n-1, B, E, A);
    B → E
    tower(n-1, A, B, E);
    Return
}
```

total disk moves = $2^n - 1$

total number of function call = $2^{n+1} - 1$

how many invocation are required for the first disk to move = n

Q Find the output of the following pseudo code?

```
void Print_array(a, i, j)
{
    if (i = = j)
    {
        printf("%d', a[i]);
        return;
    }
    Else
    {
        printf("%d', a[i]);
        print_array(a, i+1, j)
    }
}
```

Q Find the output of the following pseudo code?

```
void Print_array(a, i, j)
{
    if (i = = j)
    {
        printf(“%d’, a[i]);
        return;
    }
    Else
    {
        print_array(a, i+1, j)
        printf(“%d’, a[i]);
    }
}
```

Q Find the output of the following pseudo code?

```
void Print_somthing(a, i, j)
{
    if (i = = j)
    {
        printf("%d', a[i]);
        return;
    }
    Else
    {
        if(a[i] < a[j])
            Print_somthing (a, i+1, j);
        Else
            Print_somthing (a, i, j-1);
    }
}
```

Q Find what this function is doing?

```
 void what(struct Bnode *t)
{
    if (t)
    {
        what(t → LC);
        printf(“%d’, t → data);
        what(t → RC);
    }
}
```

Q Find what this function is doing?

```
 void what(struct Bnode *t)
{
     if (t)
     {
            printf(‘’%d’, t → data);
            what(t → LC);
            what(t → RC);
     }
}
```

Q Find what this function is doing?

```
 void what(struct Bnode *t)
{
     if (t)
     {
          what(t → LC);
          what(t → RC);
          printf(“’%d’, t → data);
     }
}
```

Q Find what this function is doing?

```
 Void what(struct Bnode *t)
{
    if (t)
    {
            printf("%d', t → data);
            what(t → LC);
            printf("%d', t → data);
            what(t → RC);
            printf("%d', t → data);
    }
}
```

Q Find what this function is doing?

```
Void A(struct Bnode *t)
{
    if (t)
    {
        B(t → LC);
        printf("%d', t → data);
        B(t → RC);
    }
}

Void B(struct Bnode *t)
{
    if (t)
    {
        printf("%d', t → data);
        A(t → LC);
        A(t → RC);
    }
}
```

Q What does the following function print for n = 25?

```
void fun(int n)
{
     if (n == 0)
         return;
    printf("%d", n%2);
    fun(n/2);
}
```

(A) 11001
(B) 10011
(C) 11111
(D) 00000

Q Consider the following recursive function fun (x, y). What is the value of fun (4, 3)

```
int fun(int x, int y)
{
      if (x == 0)
            return y;
      return fun(x - 1,  x + y);
}
```

(A) 13

(B) 12

(C) 9

(D) 10

Q What does the following function do?

```
int fun(int x, int y)
{
    if (y == 0)
        return 0;
    return (x + fun(x, y-1));
}
```

(A) x + y

(B) x + x*y

(C) x*y

(D) x^y

http://www.knowledgegate.in/GATE

Q What does fun2() do in general?

```
int fun(int x, int y)
{
if (y == 0)
return 0;
    return (x + fun(x, y-1));
}

int fun2(int a, int b)
{
    if (b == 0)
        return 1;
    return fun(a, fun2(a, b-1));
}
```

(A) x*y **(B)** x+x*y

(C) x^y **(D)** y^x

Queue

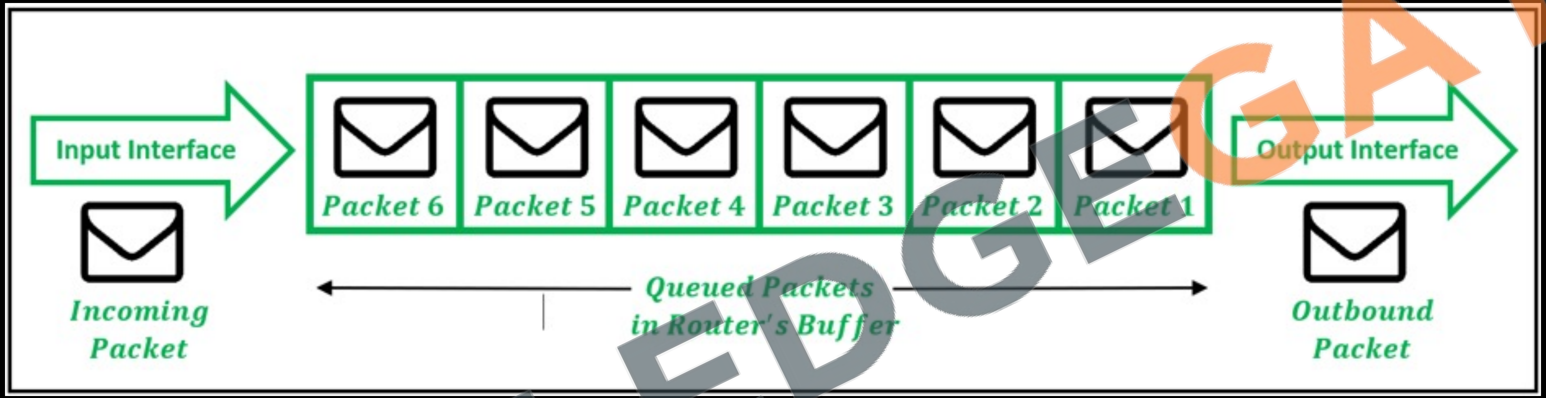
- A queue is a linear list of elements in which deletions can take place only at one end called the front, and insertions can take place only at the end called rear.
- Queue is a first in first out types of data structure(FIFO), the terms 'Front' and 'Rear' are used in describing a linear list only when it is implemented as a queue.

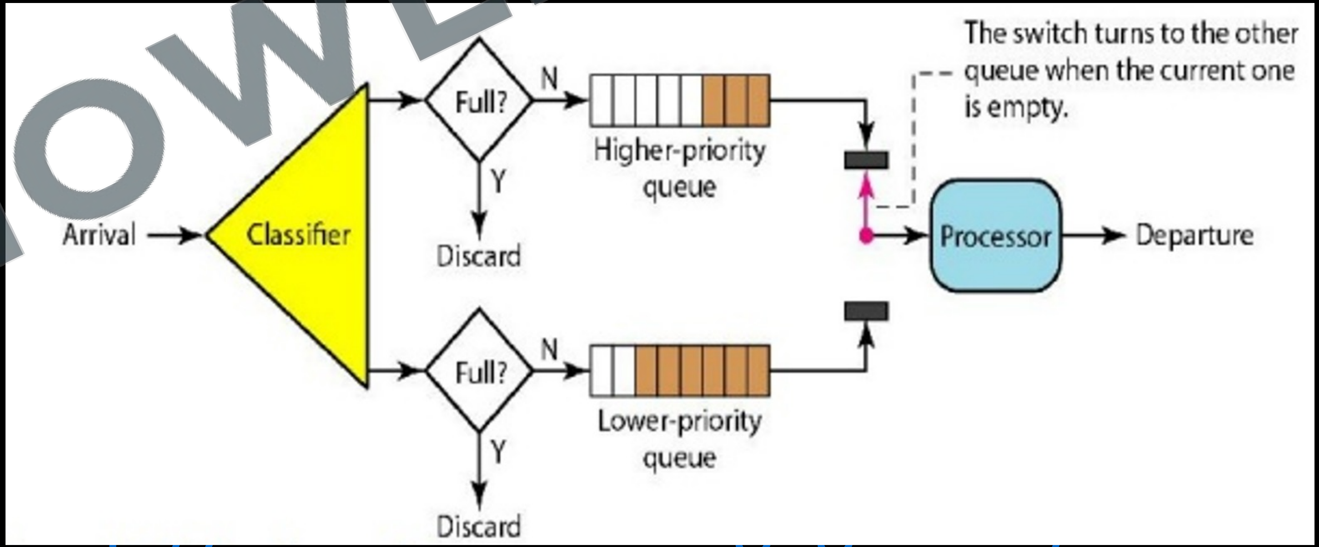
4GIFS.com

knowle

- In computer science queue are used in multiple places e.g. in time sharing system program with the same priority from a queue waiting to be executed.

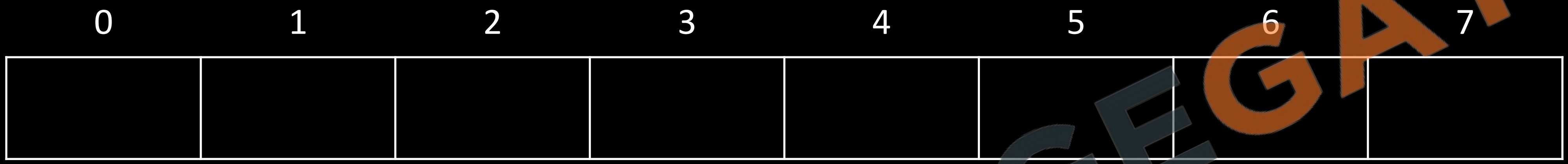
- A queue is a non-primitive linear data structure. it is homogeneous collection of elements.



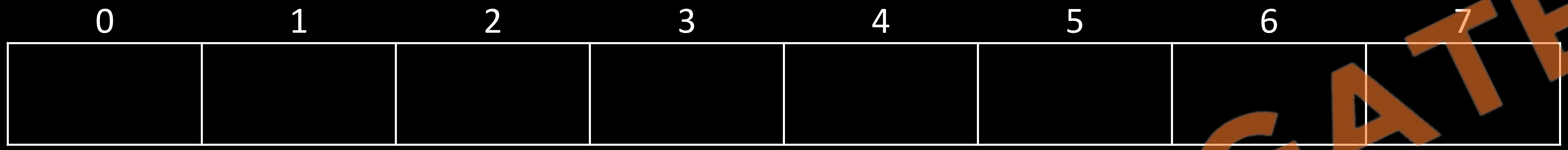


Representation of Queues

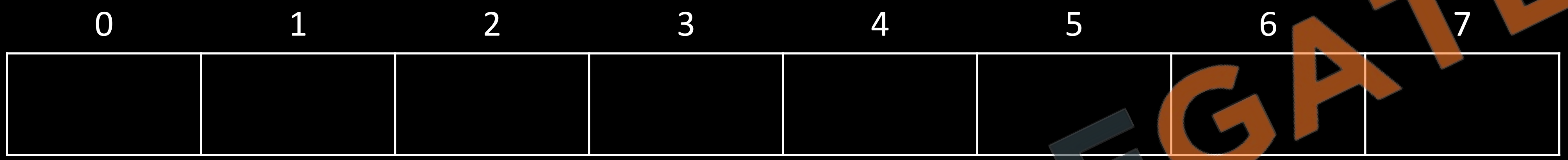
- Mostly each of our queues will be maintained by a linear array QUEUE and two pointer variables: FRONT containing the location of the Front element of the queue and REAR, containing the location of the rear element of the queue.



- Whenever an element is added to the queue, the value of REAR is increased by 1
  - REAR = REAR + 1



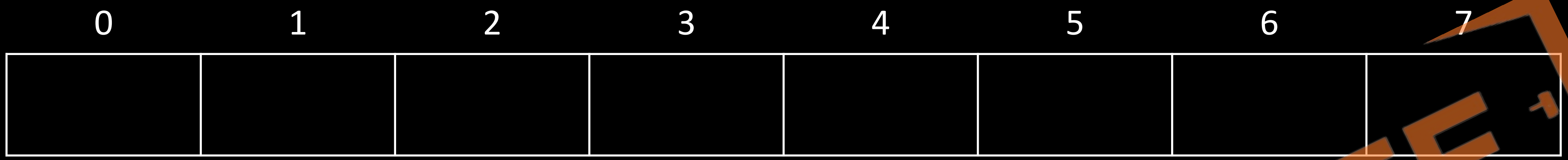
- The condition FRONT = null, will indicate that the queue is empty. Whenever an element is deleted from the queue, the value of FRONT is increased by 1
  - Front = Front + 1



- This means that after N insertion the rear element of the queue will occupy QUEUE[N] or queue will occupy the last part of the array. This may occur even though the queue itself may not contain many elements.

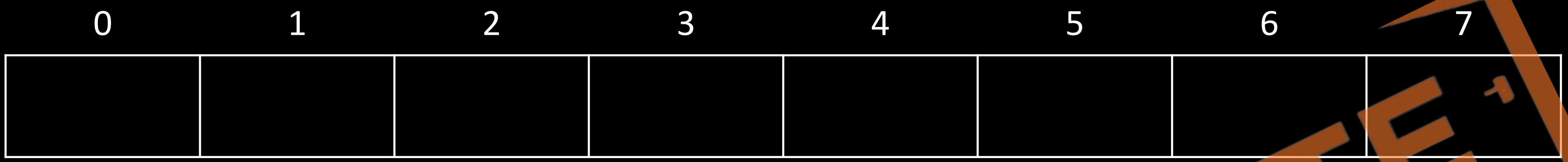
- Total number of elements in a queue
  - Rear – Front + 1

Insertion



```
Enqueue (QUEUE, N, F, R, ITEM)
{
    if (R == N - 1)
        Write over flow and exit
    if (F = = -1)
        Set F = 0 && R = 0
    Else
        R = R + 1
    Queue[R] = ITEM
}
```

Deletion



```
Deque (QUEUE, N, F, R, ITEM)
{
    if (F == - 1)
        Write under flow and exit
    ITEM = QUEUE[F]
    if (F = = R)
        Set F = -1 && R = -1
    Else
        F = F + 1
    Return item
}
```

```
typedef struct
{
    int arr[MAX_SIZE];
    int front;
    int rear;
} Queue;

void initialize(Queue *q)
{
    q->front = -1;
    q->rear = -1;
}
```

```
int isEmpty(Queue *q)
{
    return q->front == -1;
}


int isFull(Queue *q)
{
    return q->rear == MAX_SIZE - 1;
}
```

```
void enqueue(Queue *q, int item)
{
    if (isFull(q))
    {
        printf("Queue is full!\n");
        return;
    }
    if (isEmpty(q))
    {
        q->front = 0;
    }
    q->arr[++(q->rear)] = item;
}
```

```
int dequeue(CircularQueue *q)
{
    if (isEmpty(q))
    {
        printf("Queue is empty!\n");
        exit(1);
    }
    int dequeuedItem = q->arr[q->front];
    if (q->front == q->rear)
    {
        q->front = -1;
        q->rear = -1;
    }
    else
    {
        q->front = (q->front + 1) % MAX_SIZE;
    }
    return dequeuedItem;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
int main()
{
    Queue q;
    initialize(&q);



    return 0;
}
```

```
typedef struct Node
{
    int data;
    struct Node* next;
} Node;



typedef struct
{
    Node* front;
    Node* rear;
} Queue;
```

```
void initialize(Queue* q)
{
    q->front = NULL;
    q->rear = NULL;
}


int isEmpty(Queue* q)
{
    return q->front == NULL;
}
```

```
void enqueue(Queue* q, int item)
{
    Node* newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Queue overflow!\n");
        return;
    }
    newNode->data = item;
    newNode->next = NULL;
    if (isEmpty(q))
    {
        q->front = newNode;
    }
    else
    {
        q->rear->next = newNode;
    }
    q->rear = newNode;
}
```

```
int dequeue(Queue* q)
{
    if (isEmpty(q))
    {
        printf("Queue underflow!\n");
        exit(1);
    }
    Node* temp = q->front;
    int dequeuedItem = temp->data;
    q->front = q->front->next;
    if (q->front == NULL)
    {
        q->rear = NULL;
    }
    free(temp);
    return dequeuedItem;
}
```

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    Queue q;
    initialize(&q);




    return 0;
}
```

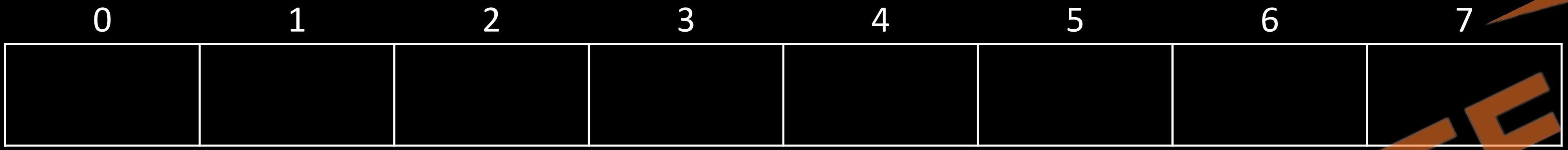
Q. Consider the following sequence of operations on an empty stack.
Push(54);push(52);pop();push(55);push(62);s=pop();

Consider the following sequence of operations on an empty queue.
enqueue(21);enqueue(24);dequeue();enqueue(28);enqueue(32);q=dequeue();
The value of s+q is ___________.

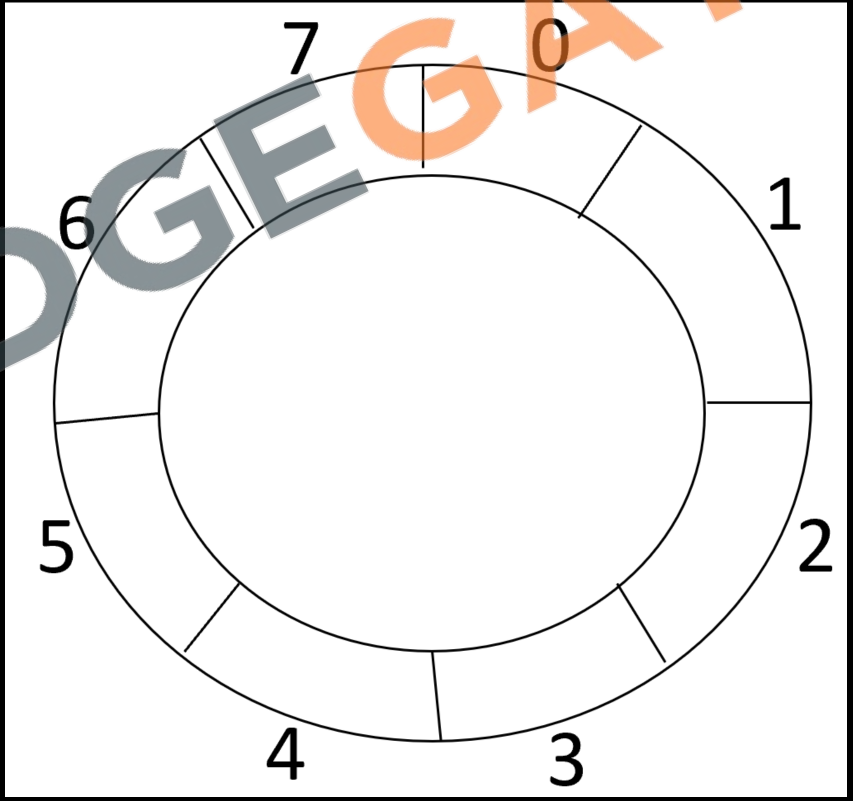
Analysis

0	1	2	3	4	5	6	7

Circular Queue



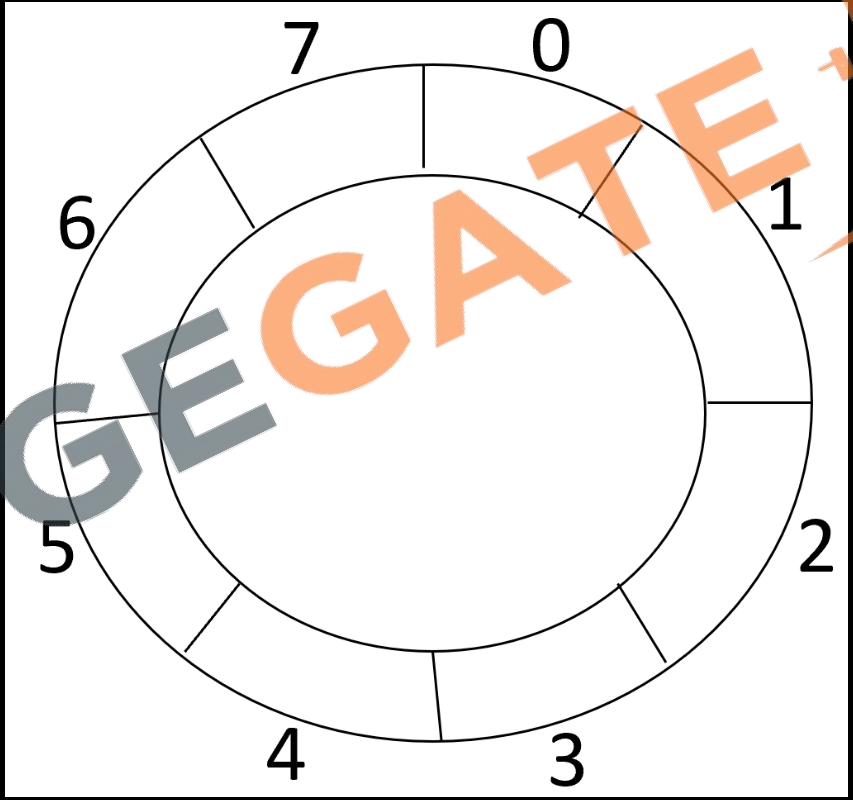
1. EnQ a, b, c
2. DeQ 1 element
3. EnQ d, e, f
4. EnQ g, h, I
5. DeQ 4 element
6. EnQ j, k, l, m, n



Circular Queue

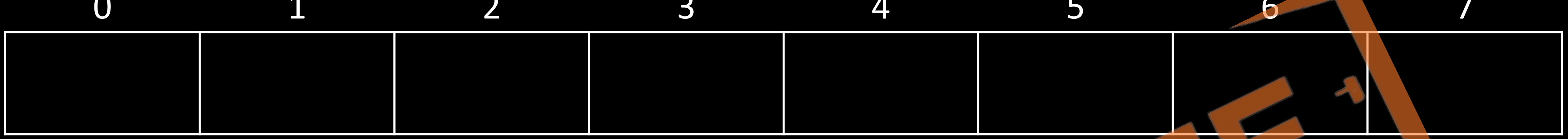
Insertion

```
Enqueue(QUEUE, N, F, R, ITEM)
{
    if ((F==0 && R==N-1) :: (F == R + 1))
        Write over flow and exit
    if (F = = -1)
        Set F = 0 && R = 0
    Else
        R = (R + 1)%N
    Queue[R] = ITEM
}
```

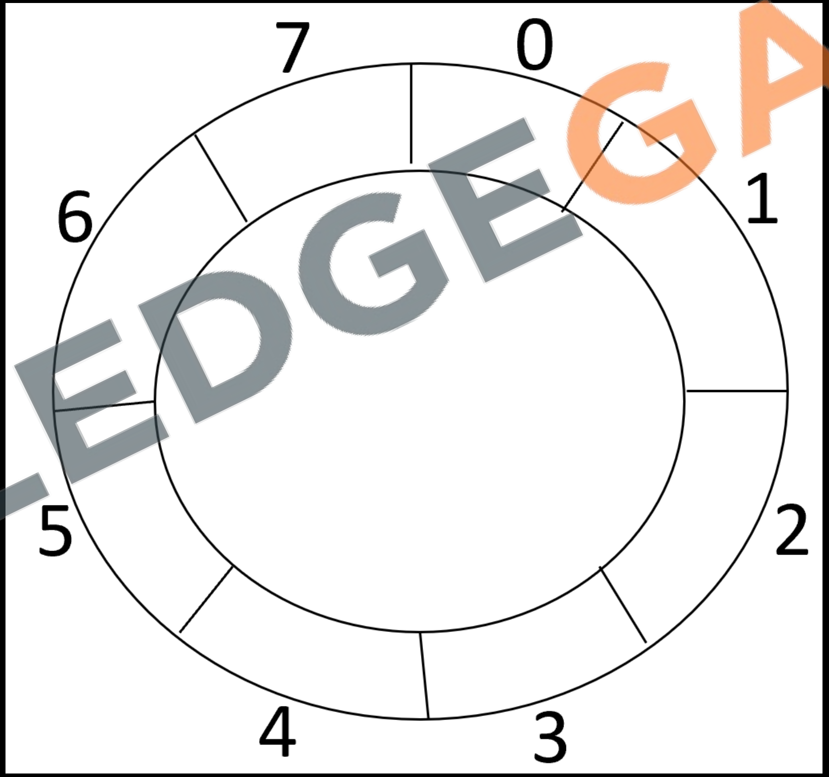


Circular Queue

Deletion



```
Dequeue(QUEUE, N, F, R, ITEM)
{
    if (F == - 1)
        Write under flow and exit
    ITEM = QUEUE[F]
    if (F = = R)
        Set F = -1 && R = -1
    Else
        F = (F + 1)%N
    Return item
}
```



```
typedef struct
{
    Int arr[MAX_SIZE];
    int front;
    int rear;
} CircularQueue;



void initialize(CircularQueue *q)
{
    q->front = -1;
    q->rear = -1;
}
```

```
Int isEmpty(CircularQueue *q)
{
    return q->front == -1;
}


int isFull(CircularQueue *q)
{
    return (q->rear + 1) % MAX_SIZE == q->front;
}
```

```
void enqueue(CircularQueue *q, int item)
{
    if (isFull(q))
    {
        printf("Queue is full!\n");
        return;
    }
    if (isEmpty(q))
    {
        q->front = 0;
        q->rear = 0;
    }
    else
    {
        q->rear = (q->rear + 1) % MAX_SIZE;
    }
    q->arr[q->rear] = item;
}
```

```
int dequeue(CircularQueue *q)
{
        if (isEmpty(q))
        {
                printf("Queue is empty!\n");
                exit(1);
        }
        int dequeuedItem = q->arr[q->front];
        if (q->front == q->rear)
        {
                q->front = -1;
                q->rear = -1;
        }
        else
        {
                q->front = (q->front + 1) % MAX_SIZE;
        }
        return dequeuedItem;
}
```

```
int main()
{
    CircularQueue q;
    initialize(&q);




    return 0;
}
```

```c
typedef struct Node
{
    int data;
    struct Node* next;
} Node;


typedef struct
{
    Node* front;
    Node* rear;
} CircularQueue;
```

```
void initialize(CircularQueue* q)
{
    q->front = NULL;
    q->rear = NULL;
}

int isEmpty(CircularQueue* q)
{
    return q->front == NULL;
}
```

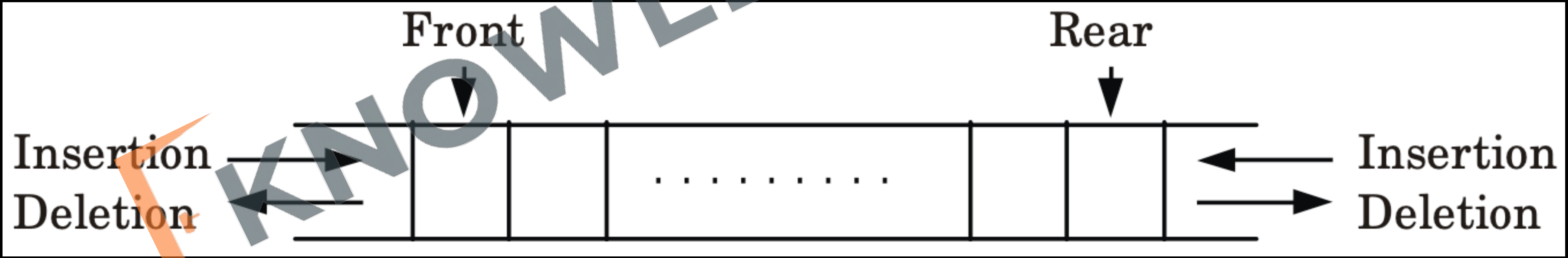
```
// Enqueue an item to the queue
void enqueue(CircularQueue* q, int item)
{
        Node* newNode = (Node*) malloc(sizeof(Node));
        if (newNode == NULL)
        {
                printf("Queue overflow!\n");
                return;
        }
        newNode->data = item;
        newNode->next = NULL;
        if (isEmpty(q))
        {
                q->front = newNode;
                q->rear = newNode;
                newNode->next = newNode;
        }
        else
        {
                newNode->next = q->front;
                q->rear->next = newNode;
                q->rear = newNode;
        }
}
```

```
// Dequeue an item from the queue
int dequeue(CircularQueue* q)
{
        if (isEmpty(q))
        {
                printf("Queue underflow!\n");
                exit(1);
        }
        Node* temp = q->front;
        int dequeuedItem = temp->data;
        if (q->front == q->rear)
        {
                q->front = NULL;
                q->rear = NULL;
        }
        else
        {
                q->front = q->front->next;
                q->rear->next = q->front;
        }
        free(temp);
        return dequeuedItem;
}
```

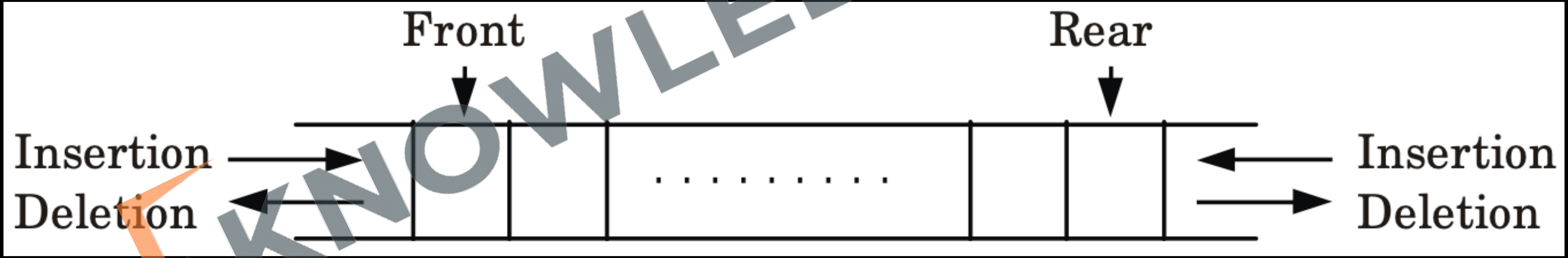
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    CircularQueue q;
    initialize(&q);




    return 0;
}
```

Dequeue

- In a dequeue, both insertion and deletion operations are performed at either end of the queues. That is, we can insert an element from the rear end or the front end. Also deletion is possible from either end.
- This dequeue can be used both as a stack and as a queue.
- There are various ways by which this dequeue can be represented. The most common ways of representing this type of dequeue are :
  - Using a doubly linked list
  - Using a circular array



- Types of dequeue :
  - **Input-restricted dequeue** : In input-restricted dequeue, element can be added at only one end but we can delete the element from both ends.
  - **Output-restricted dequeue** : An output-restricted dequeue is a dequeue where deletions take place at only one end but allows insertion at both ends.

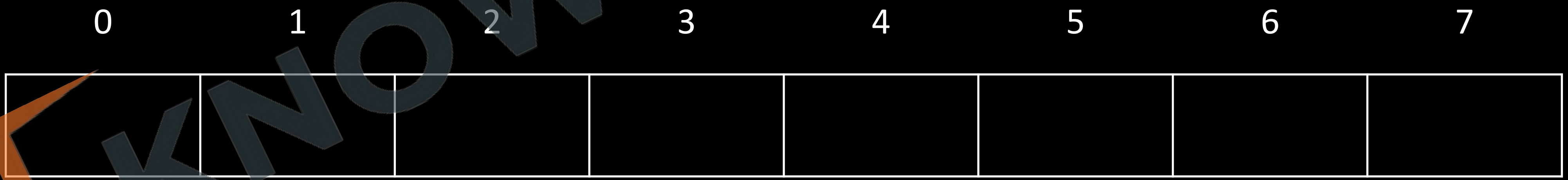


Priority Queue

- A priority queue is a collection of elements such that each element has been assigned a priority and such that the order in which elements are deleted and processed comes from the following rules.
    - An element of higher priority is processed before any element of lower priority
    - Two element with the same priority are processed according to the order in which they were added to the queue.

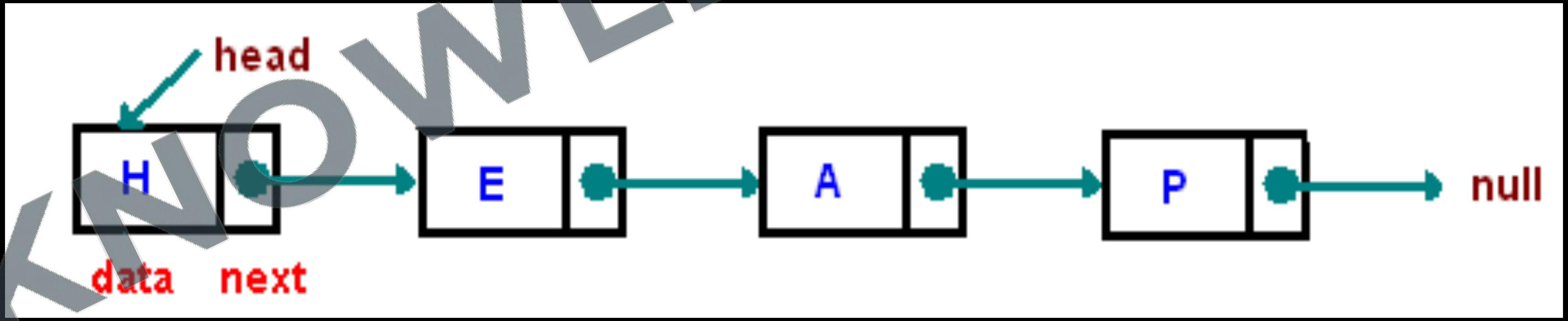
Problem with Array

- **Fixed size and reallocation**: Arrays have a fixed size, which can lead to memory waste if the allocated size is larger than the actual data. Resizing an array often requires creating a new one and copying elements, which can be inefficient.
- **Inefficient insertion and deletion**: Adding or removing elements in the middle of an array requires shifting the remaining elements, resulting in a higher time complexity (O(n)) compared to linked lists.
- **Less flexible**: Arrays can only store elements of the same data type, and their structure cannot be easily adapted to different types (e.g., singly, doubly, circular) like linked lists.



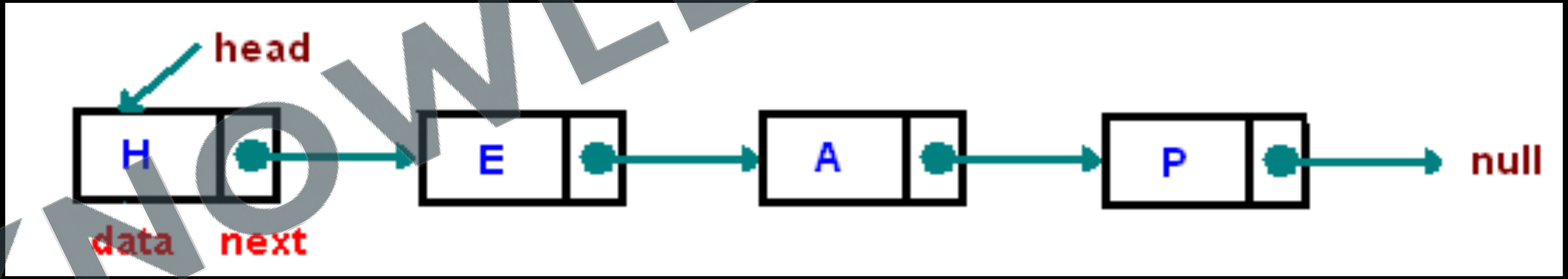
Linked list

- Solution is linked list
  - A linked list is a dynamic data structure that consists of elements called nodes, which are connected in a linear sequence. Each node contains two parts: data and a reference to the next node.
  - The first part is the information part of the node, which can store any type of information, such as integers, characters, or objects.
  - The second part called linked field or next pointer field, contains the address of the next node of the list.



http://www.knowledgegate.in/GATE

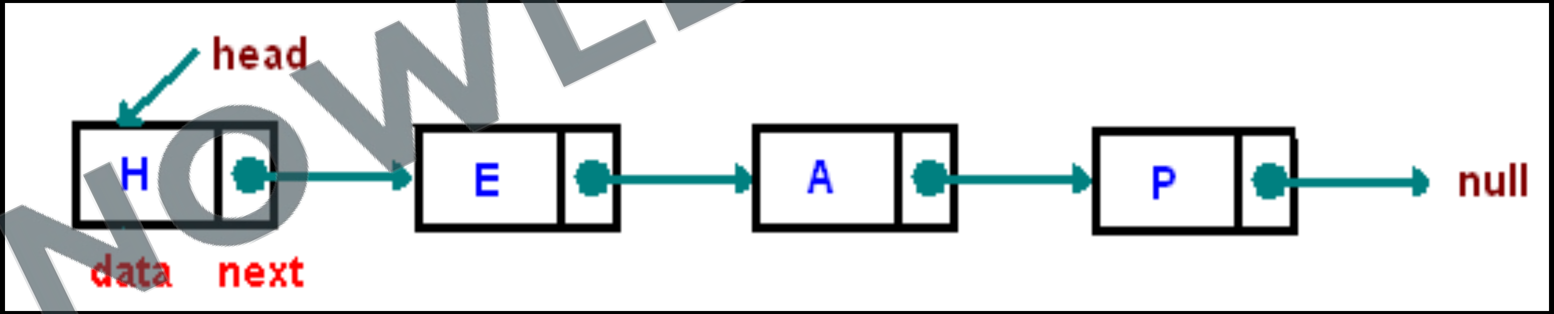
- The pointer of the last node contains a null pointer, which is an invalid address (0 or negative value).
- The linked also contains a list pointer variable called start/first/head which contain the address of the first node in the list.
- A special case is the list that has no nodes, such a list is called null list or empty list and is denoted by a null pointer in the variable start/first/head.



http://www.knowledgegate.in/GATE

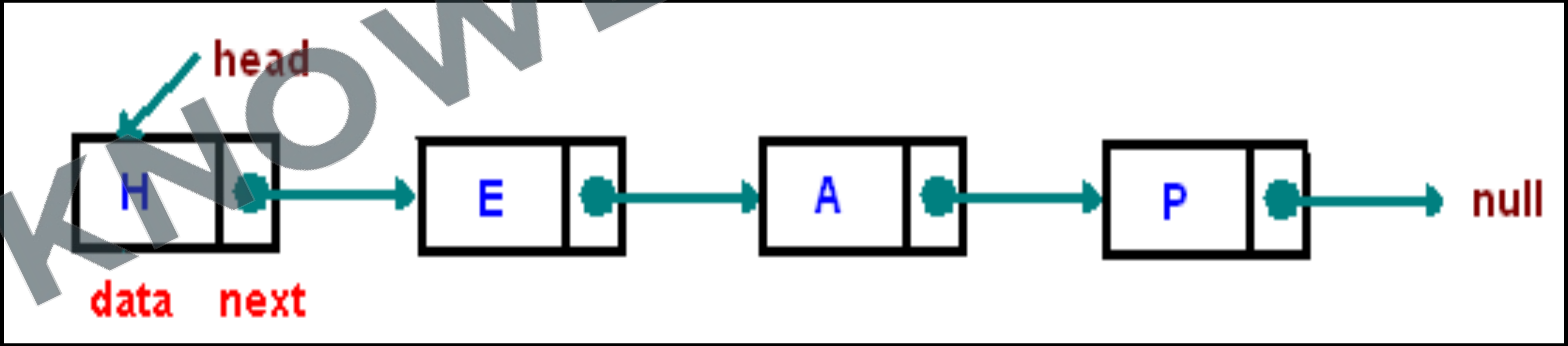
Implementation of link list

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



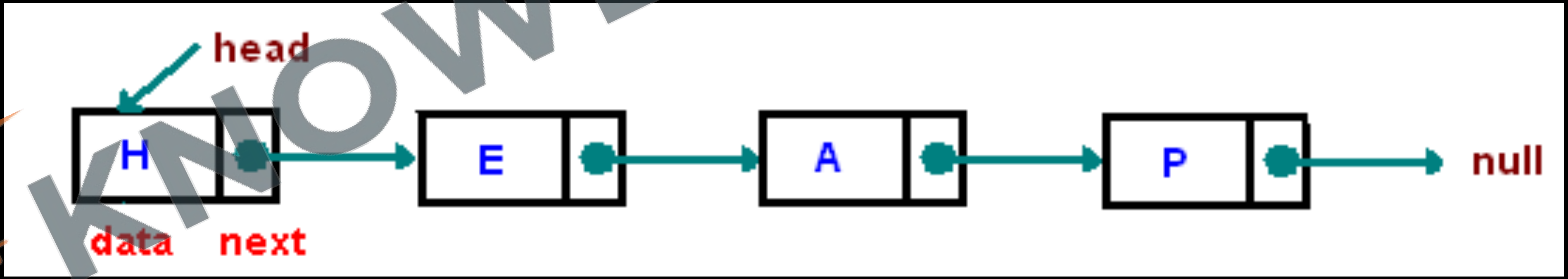
Advantage of link list

- **Dynamic size and efficient memory usage**: Linked lists can easily grow or shrink, allowing for efficient memory allocation and reduced waste as elements are added or removed.
- **Fast insertion and deletion**: Operations like inserting or removing elements can be performed in constant time (O(1)) if the position is known, offering better performance compared to array-based structures.
- **Versatility**: Linked lists can be adapted to various types (singly, doubly, circular) and can store elements of different data types or objects, providing a flexible data structure for a wide range of applications.



Disadvantage of link list

- **Slower access times**: Linked lists have a higher time complexity for element access (O(n)) compared to arrays, as elements must be accessed sequentially from the head of the list.
- **Memory overhead**: Each node in a linked list requires additional memory to store the reference (or pointer) to the next node, increasing the overall memory usage compared to array-based structures.
- **Pointer manipulation**: Implementing linked lists involves managing pointers, which can increase code complexity and lead to potential issues, such as memory leaks or segmentation faults, if not handled carefully.



http://www.knowledgegate.in/GATE

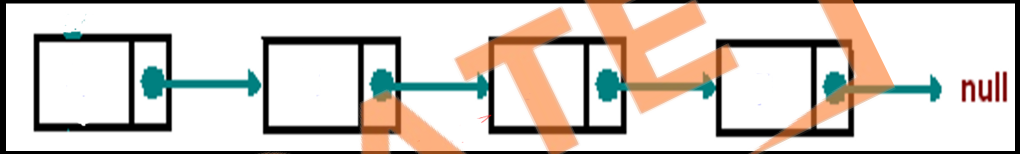
Aspect	Array	Linked List
Memory Allocation	Contiguous memory locations.	Non-contiguous memory locations.
Size Flexibility	Fixed size.	Dynamic size, can grow or shrink as required.
Access Time	O(1) for direct access due to indexing.	O(n) for accessing an element as it requires traversal.
Insertion/Deletion	O(n) in worst case as shifting may be required.	O(1) if the pointer to the node is known.
Memory Efficiency	More memory efficient for a known size of data.	Extra memory for pointers, which can be overhead for small data sizes.

```c
#include <stdio.h>
#include <stdlib.h>
// Define the Node structure
typedef struct Node
{
	int data;
	struct Node* next;
} Node;

Node* createNode(int data)
{
	Node* newNode = (Node*) malloc(sizeof(Node));
	if (!newNode)
	{
		printf("Memory error\n");
		exit(1);
	}
	newNode->data = data;
	newNode->next = NULL;
	return newNode;
}
```

```
int main()
{
    Node* head = createNode(10);
    head->next = createNode(20);
    head->next->next = createNode(30);
    printf("Linked List: ");
    traverseList(head);
    return 0;
}
```

Q Write a C-style pseudocode for Traversing a link list iteratively, where pointer head have the address of the first node of the list?

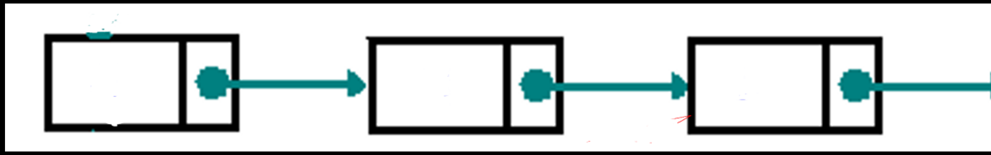
```
void traverseList(Node* head)
{
    Node* current = head;
    while (current != NULL)
    {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```



http://www.knowledgegate.in/GATE

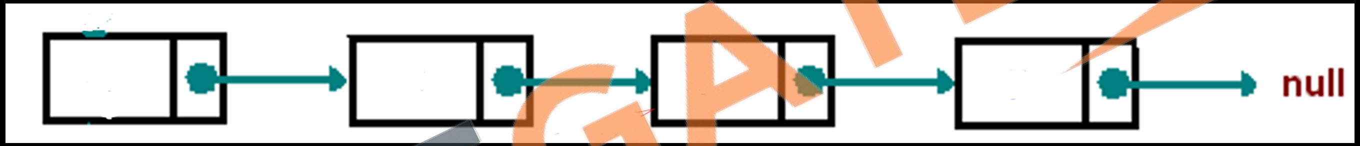
Q Write a C-style pseudocode for Traversing a link list recursively, where pointer head have the address of the first node of the list?

```
void traverseListRecursive(Node* current)
{
    if (current == NULL)
    {
        printf("NULL\n");
        return;
    }
    printf("%d -> ", current->data);
    traverseListRecursive(current->next);
}
```



Q Write a C-style pseudocode for searching a key in a link list iteratively, where pointer head have the address of the first node of the list?

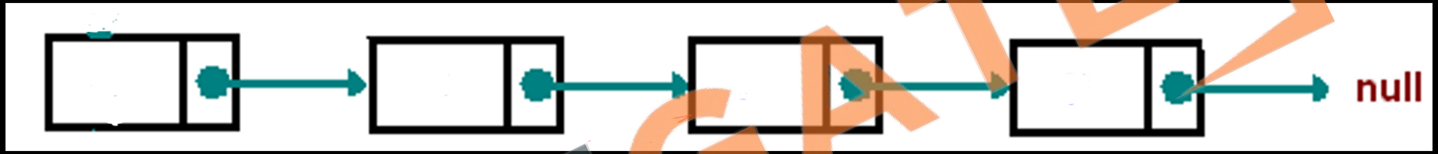
```
Node* searchKeyIterative(Node* head, int key)
{
    Node* current = head;
    while (current != NULL)
    {
        if (current->data == key)
        {
            return current;
        }
        current = current->next;
    }
    return NULL;
}
```



http://www.knowledgegate.in/GATE

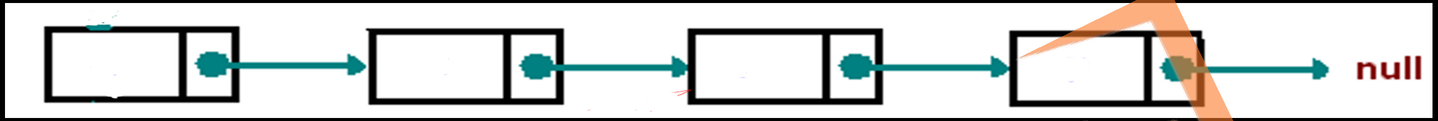
Q Write a C-style pseudocode for searching a key in a link list recursively, where pointer head have the address of the first node of the list?

```
Node* searchKeyRecursive(Node* current, int key)
{
    if (current == NULL)
    {
        return NULL;
    }
    if (current->data == key)
    {
        return current;
    }
    return searchKeyRecursive(current->next, key);
}
```



http://www.knowledgegate.in/GATE

Q Write a C-style pseudocode for inserting a node with a key in the starting of link-list?



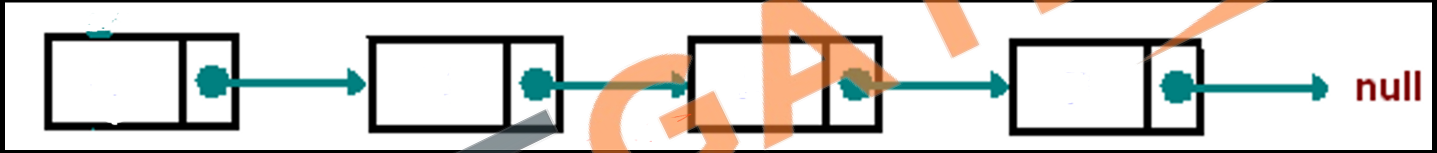
```
void insertAtBeginning(Node** head, int key)
{
    Node* newNode = createNode(key);
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
```

Q Write a C-style pseudocode for inserting a node with a key after a location in a link-list?

```
void insertAfter(Node* prevNode, int key)
{
    if (prevNode == NULL)
    {
        printf("The given previous node cannot be NULL.\n");
        return;
    }
    Node* newNode = createNode(key);
    newNode->data = data;
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}
```

Q Write a C-style pseudocode for deleting a node from the starting of link-list?

```
void deleteAtBeginning(Node** head)
{
    if (*head == NULL)
    {
        printf("List is already empty.\n");
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

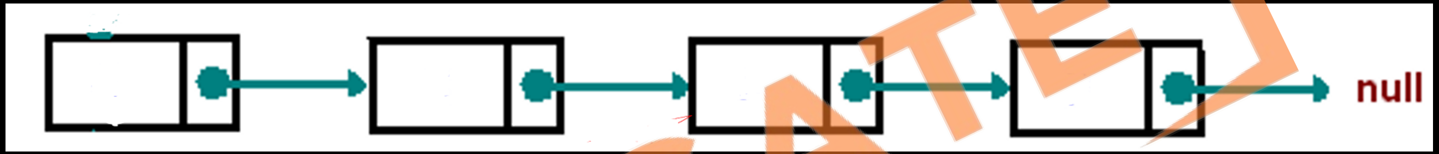


Q Write a C-style pseudocode for deleting a node after a given location from the starting of link-list?

```
void deleteAfter(Node* prevNode)
{
    if (prevNode == NULL || prevNode->next == NULL)
    {
        printf("The given node is NULL or there's no node after it to delete.\n");
        return;
    }
    Node* temp = prevNode->next;
    prevNode->next = temp->next;
    free(temp);
}
```

Q Write a C-style pseudocode for reversing a link-list in a iteratively?

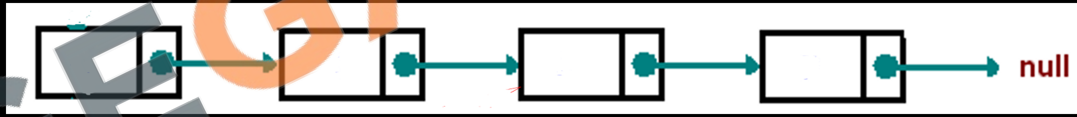
```
void reverseListIterative(Node** head)
{
    Node* prev = NULL;
    Node* current = *head;
    Node* next = NULL;
    while (current != NULL)
    {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head = prev;

}
```



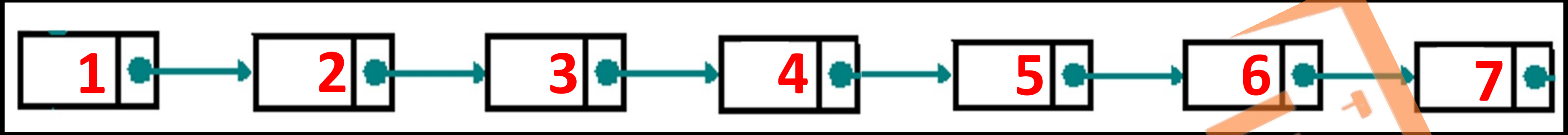
http://www.knowledgegate.in/GATE

Q Write a C-style pseudocode for reversing a link-list in a recursively?

```
Node* reverseListRecursive(Node* head)
{
    if (head == NULL || head->next == NULL)
    {
        return head;
    }
    Node* rest = reverseListRecursive(head->next);
    head->next->next = head;
    head->next = NULL;
    return rest;
}
```



Q The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?



```
struct node
{
        int value;
        struct node *next;
};
void rearrange(struct node *list)
{
        struct node *p, * q;
        int temp;
        if ((!list) || !list->next)
                return;
        p = list;
        q = list->next;
        while(q)
        {
                temp = p->value;
                p->value = q->value;
                q->value = temp;
                p = q->next;
                q = p ? p->next:0;
        }
}
```

http://www.knowledgegate.in/GATE

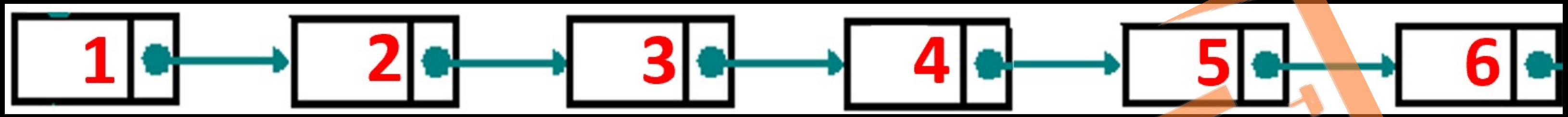
Q The following C function takes a simply-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some part of the code is left blank. Choose the correct alternative to replace the blank line.

```
typedef struct node
{
        int value;
        struct node *next;
}Node;

Node *move_to_front(Node *head)
{
        Node *p, *q;
        if ((head == NULL: || (head->next == NULL))
                return head;
        q = NULL; p = head;
        while (p-> next !=NULL)
        {
                q = p;
                 p = p->next;
        }
        _______________________________
        return head;
}
```

(A) q = NULL; p->next = head; head = p;
(B) q->next = NULL; head = p; p->next = head;
(C) head = p; p->next = q; q->next = NULL;
(D) q->next = NULL; p->next = head; head = p;

http://www.knowledgegate.in/GATE

Q What is the output of following function for start pointing to first node of following linked list?



1->2->3->4->5->6

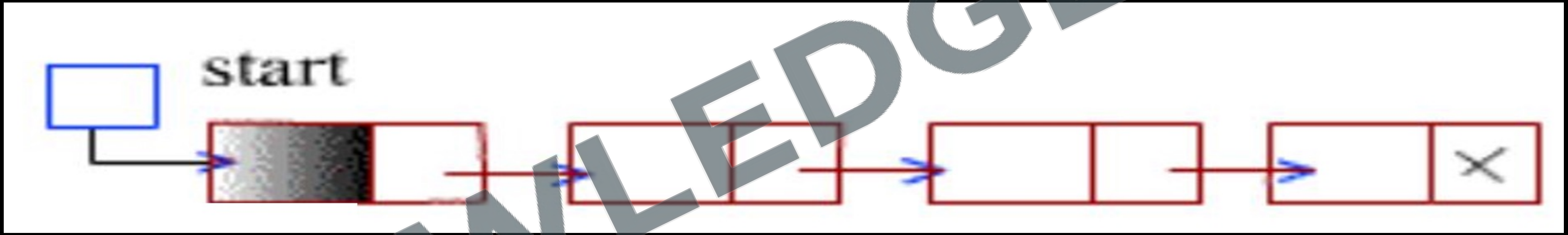
```
void fun(struct node* start)
{
    if(start == NULL)
        return;
    printf("%d  ", start->data);
    if(start->next != NULL )
        fun(start->next->next);
    printf("%d  ", start->data);
}
```

(A) 1 4 6 6 4 1 **(B)** 1 3 5 1 3 5

(C) 1 2 3 5 **(D)** 1 3 5 5 3 1

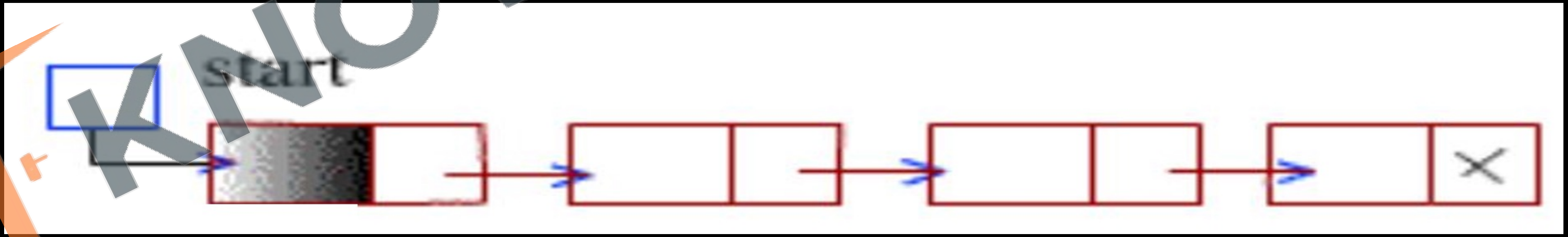
Header link list

- A header linked list is a variation of a standard linked list that includes a special node, called the header node, at the beginning of the list.
- The header node does not store any actual data; instead, it serves as a fixed reference point that simplifies some operations on the linked list like inserting or deleting elements at the beginning of the list.



Here are some features of header linked lists:

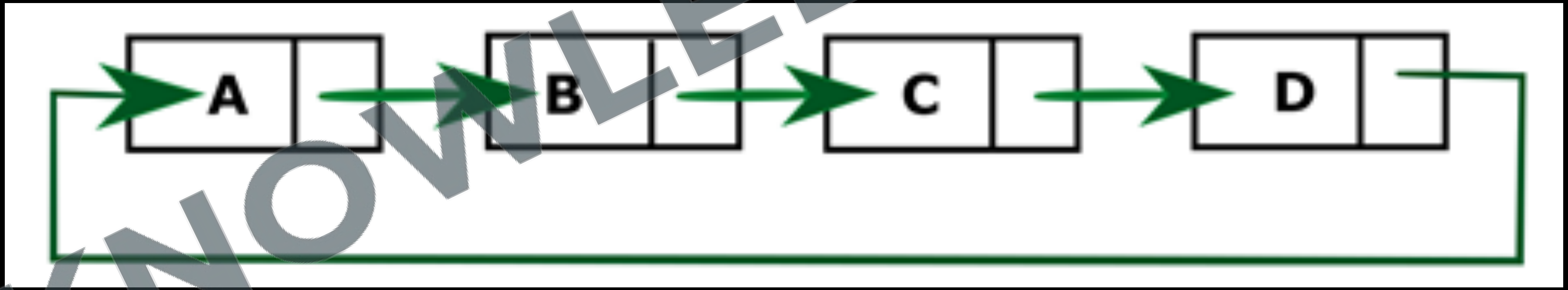
- The header node is always present, even when the list is empty.
- The header node's 'next' pointer points to the first actual data node in the list or 'Null' if the list is empty.
- The header node simplifies operations like insertion or deletion at the beginning, middle, or end of the list, as well as traversal, since the header node acts as a consistent starting point.
- The header node can also store metadata about the list, such as its length, although this is not a requirement.



```
void traverseHeaderLinkedList(Node* header)
{
    if (header == NULL)
    {
        printf("List is empty.\n");
        return;
    }
    Node* temp = header->next;
    while (temp != NULL)
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Circular Linked List

- A singly circular linked list is a variation of a singly linked list in which the last node in the list points back to the first node, creating a loop or circular structure.
- The primary difference between a standard singly linked list and a singly circular linked list is that the last node's 'Null' pointer refers to the first node in the list, rather than being 'Null'.



Here are some key features of singly circular linked lists:

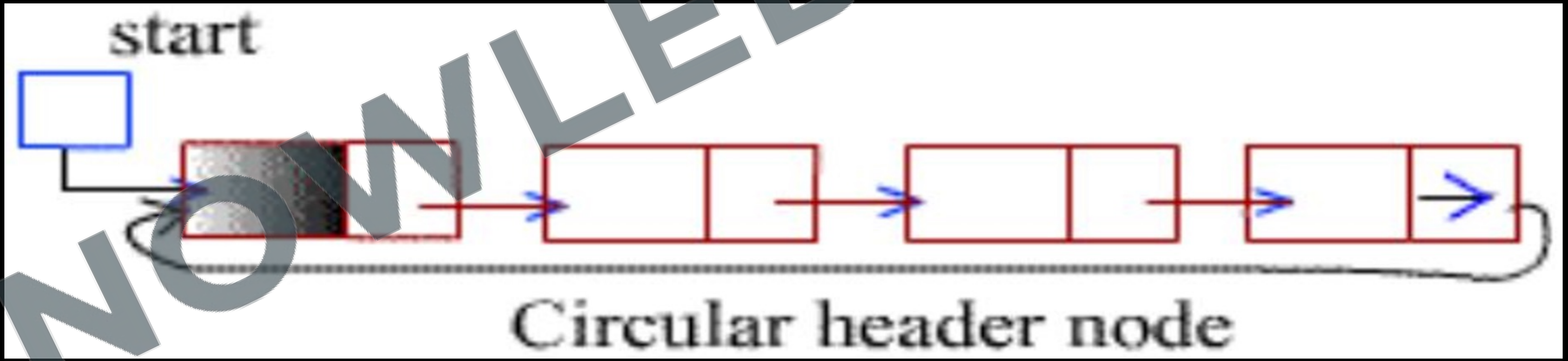
- Singly circular linked lists can be used to implement data structures like queues or circular buffers, where elements are added to the end and removed from the front, with constant-time complexity for both operations.
- Traversal of the list requires a stopping condition, such as iterating until you reach the starting node again or using a counter to limit the number of iterations, to avoid infinite loops.

```
void traverseCircularLinkedList(Node* head)
{
    if (head == NULL)
    {
        printf("List is empty.\n");
        return;
    }
    Node* temp = head;
    do
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    while (temp != head);
    printf("%d (head)\n", head->data);
}
```

Header circular link list

- A header singly circular linked list is a variation of a singly circular linked list that includes a special node, called the header node, at the beginning of the list.

- The header node does not store any actual data; instead, it serves as a fixed reference point that simplifies some operations on the linked list. The header node's primary purpose is to eliminate the need for special cases when performing certain operations like inserting or deleting elements at the beginning or end of the list.

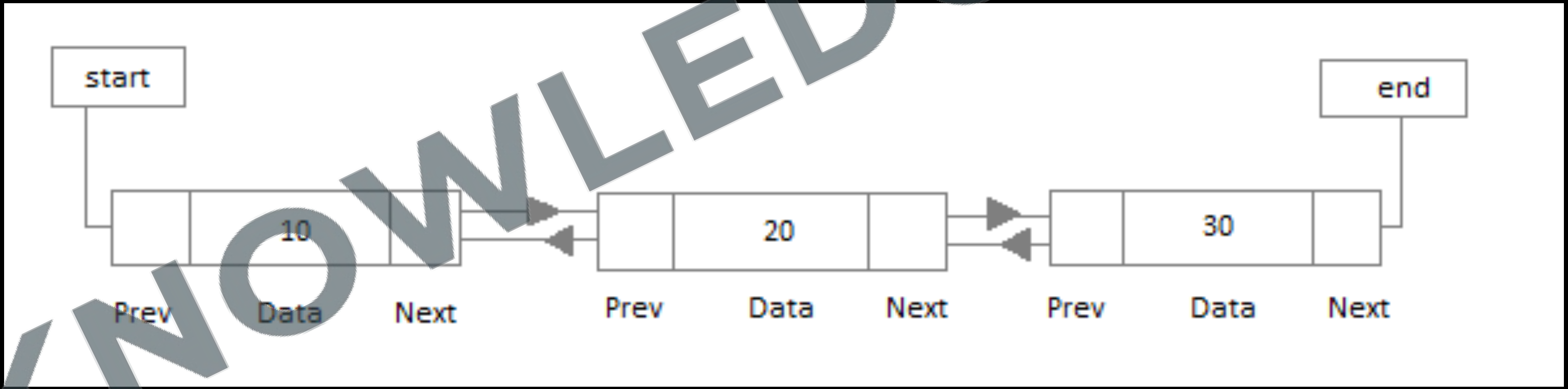


```c
void traverseHeaderCircularLinkedList(Node* header)
{
    if (header->next == header)
    {
        printf("List is empty.\n");
        return;
    }
    Node* temp = header->next;
    while (temp != header)
    {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("HEADER\n");
}
```

Doubly link list

- A doubly linked list is a data structure in which each node contains a data element and two pointers, one pointing to the previous node (the ‘previous’ pointer) and the other pointing to the next node (the ‘next’ pointer) in the sequence.

- This bidirectional linking allows for easier traversal and manipulation of the list in both forward and backward directions, as well as simplifying some operations such as insertion or deletion of nodes at any position in the list.



```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node
{
        int data;
        struct Node* prev;
        struct Node* next;
} Node;

Node* createNode(int data)
{
        Node* newNode = (Node*) malloc(sizeof(Node));
        if (!newNode)
        {
                printf("Memory error\n");
                exit(1);
        }
        newNode->data = data;
        newNode->prev = NULL;
        newNode->next = NULL;
        return newNode;
}
```

```
void insertAtBeginning(Node** head, int data)
{
        Node* newNode = createNode(data);
        if (*head != NULL)
        {
                (*head)->prev = newNode;
        }
        newNode->next = *head;
        *head = newNode;
}

void insertAtEnd(Node** head, int data)
{
        Node* newNode = createNode(data);
        if (*head == NULL)
        {
                *head = newNode;
                return;
        }
        Node* temp = *head;
        while (temp->next != NULL)
        {
                temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
}
```

```
void traverse(Node* head)
{
        Node* temp = head;
        while (temp != NULL)
        {
                printf("%d -> ", temp->data);
                temp = temp->next;
        }
        printf("NULL\n");
}

Node* search(Node* head, int key)
{
        Node* temp = head;
        while (temp != NULL)
        {
                if (temp->data == key)
                {
                        return temp;
                }
                temp = temp->next;
        }
        return NULL;
}
```

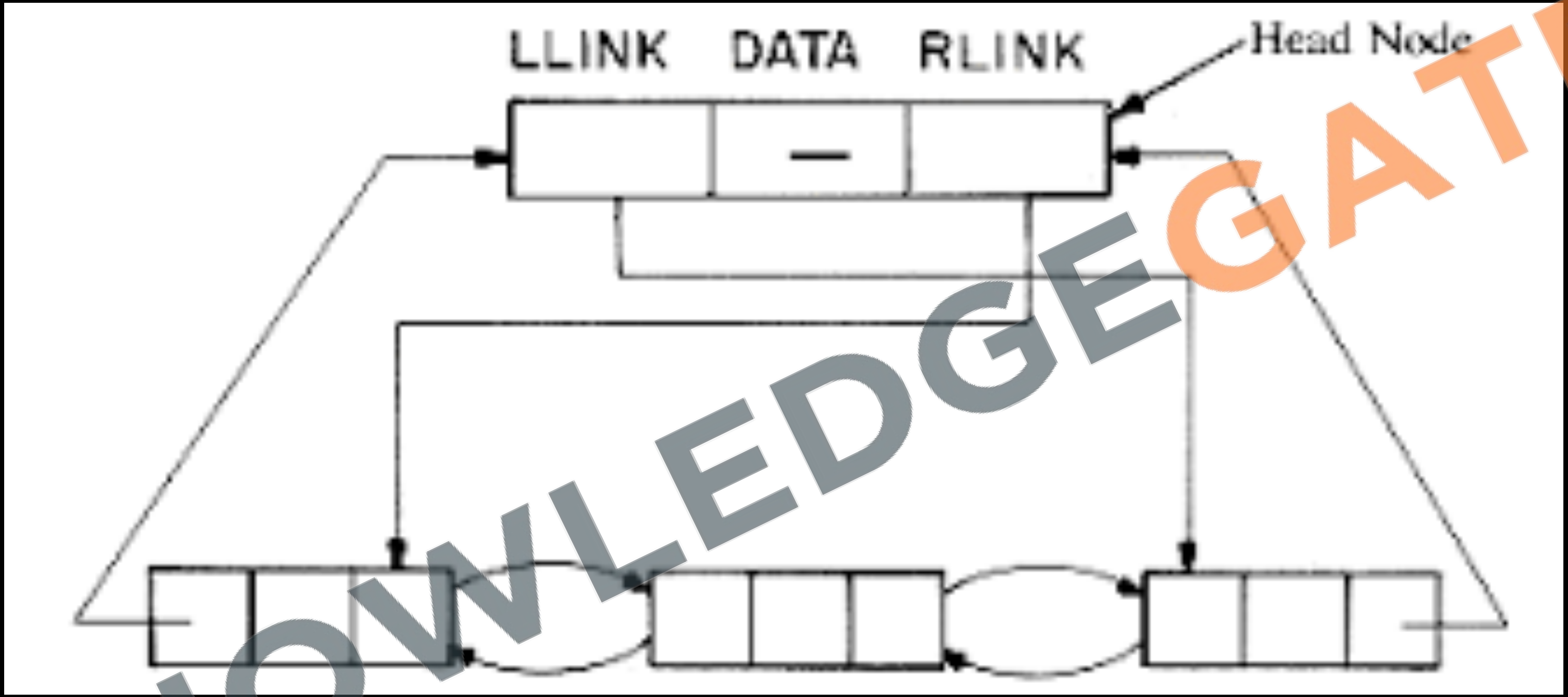
```
void deleteAtBeginning(Node** head)
{
        if (*head == NULL)
                return;
        Node* temp = *head;
        *head = (*head)->next;
        if (*head != NULL)
        {
                (*head)->prev = NULL;
        }
        free(temp);
}

void deleteAtEnd(Node** head)
{
        if (*head == NULL)
                return;
        Node* temp = *head;
        while (temp->next != NULL)
        {
                temp = temp->next;
        }
        if (temp->prev != NULL)
        {
                temp->prev->next = NULL;
        }
        else
        {
                *head = NULL;
        }
        Free(temp);
}
```

```
void deleteByPointer(Node** head, Node* loc)
{
    if (*head == NULL || loc == NULL)
        return;
    if (loc->prev == NULL)
    {
        *head = loc->next;
    }
    else
    {
        loc->prev->next = loc->next;
    }
    if (loc->next != NULL)
    {
        loc->next->prev = loc->prev;
    }
    free(loc);
}
```

Here are some key features of doubly linked lists:

- Each node has two pointers: ‘next’ pointing to the subsequent node and ‘previous’ pointing to the preceding node in the list.
- The first node’s ‘previous’ pointer and the last node’s ‘next’ pointer are set to ‘null’ indicating the beginning and end of the list, respectively.
- Doubly linked lists allow for easier traversal and manipulation in both forward and backward directions compared to singly linked lists.
- Doubly linked lists consume more memory than singly linked lists due to the additional ‘previous’ pointer.

Header Circular Doubly Link List



Q Consider the following function that takes reference to head of a Doubly Linked List as parameter. Assume that a node of doubly linked list has previous pointer as *prev* and next pointer as *next*.

```
void fun(struct node **head_ref)
{
        struct node *temp = NULL;
        struct node *current = *head_ref;
        while (current !=  NULL)
        {
                temp = current->prev;
                current->prev = current->next;
                current->next = temp;
                current = current->prev;
        }
        if(temp != NULL )
                *head_ref = temp->prev;
        }
```

Assume that reference of head of following doubly linked list is passed to above function

1 <--> 2 <--> 3 <--> 4 <--> 5 <-->6.

What should be the modified linked list after the function call?

(A) 2 <--> 1 <--> 4 <--> 3 <--> 6 <-->5

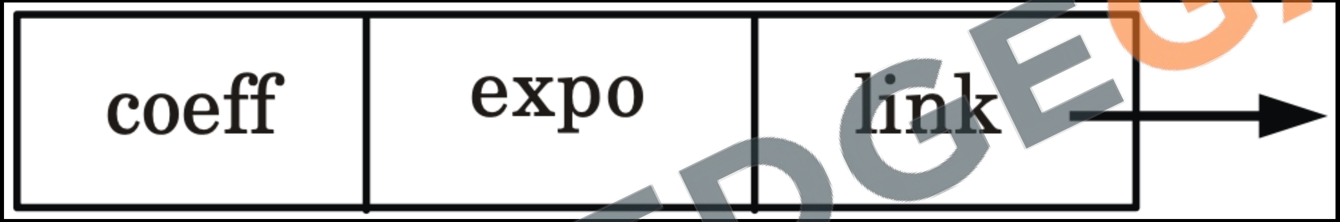
(B) 5 <--> 4 <--> 3 <--> 2 <--> 1 <-->6.

(C) 6 <--> 5 <--> 4 <--> 3 <--> 2 <--> 1.

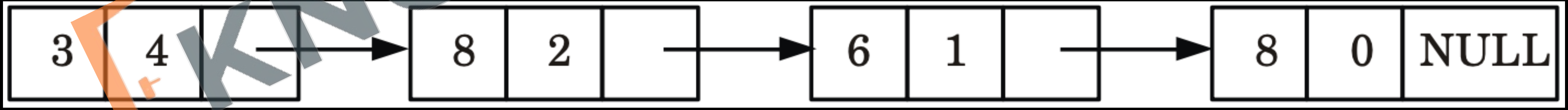
(D) 6 <--> 5 <--> 4 <--> 3 <--> 1 <--> 2

Polynomial Representation Using Linked List

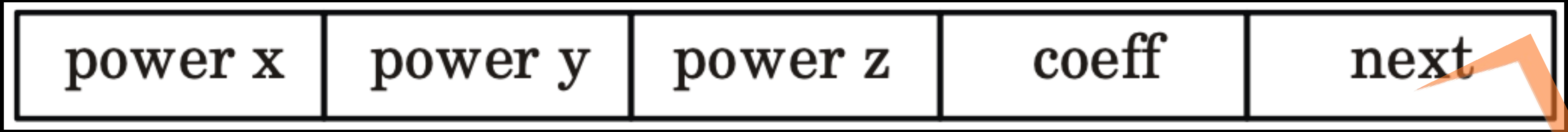
- In the linked representation of polynomials, each node should consist of three elements, namely coefficient, exponent and a link to the next term.
- The coefficient field holds the value of the coefficient of a term, the exponent field contains the exponent value of that term and the link field contains the address of the next term in the polynomial.



- $3x^4 + 8x^2 + 6x + 8$

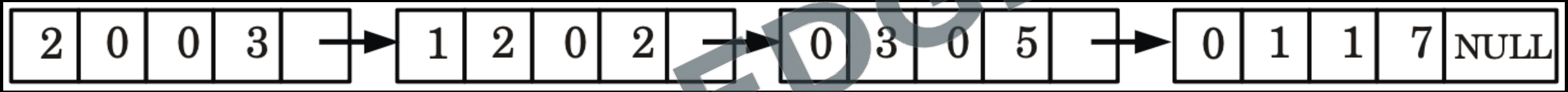


http://www.knowledgegate.in/GATE



power x	power y	power z	coeff	next

- $3x^2 + 2xy^2 + 5y^3 + 7yz$



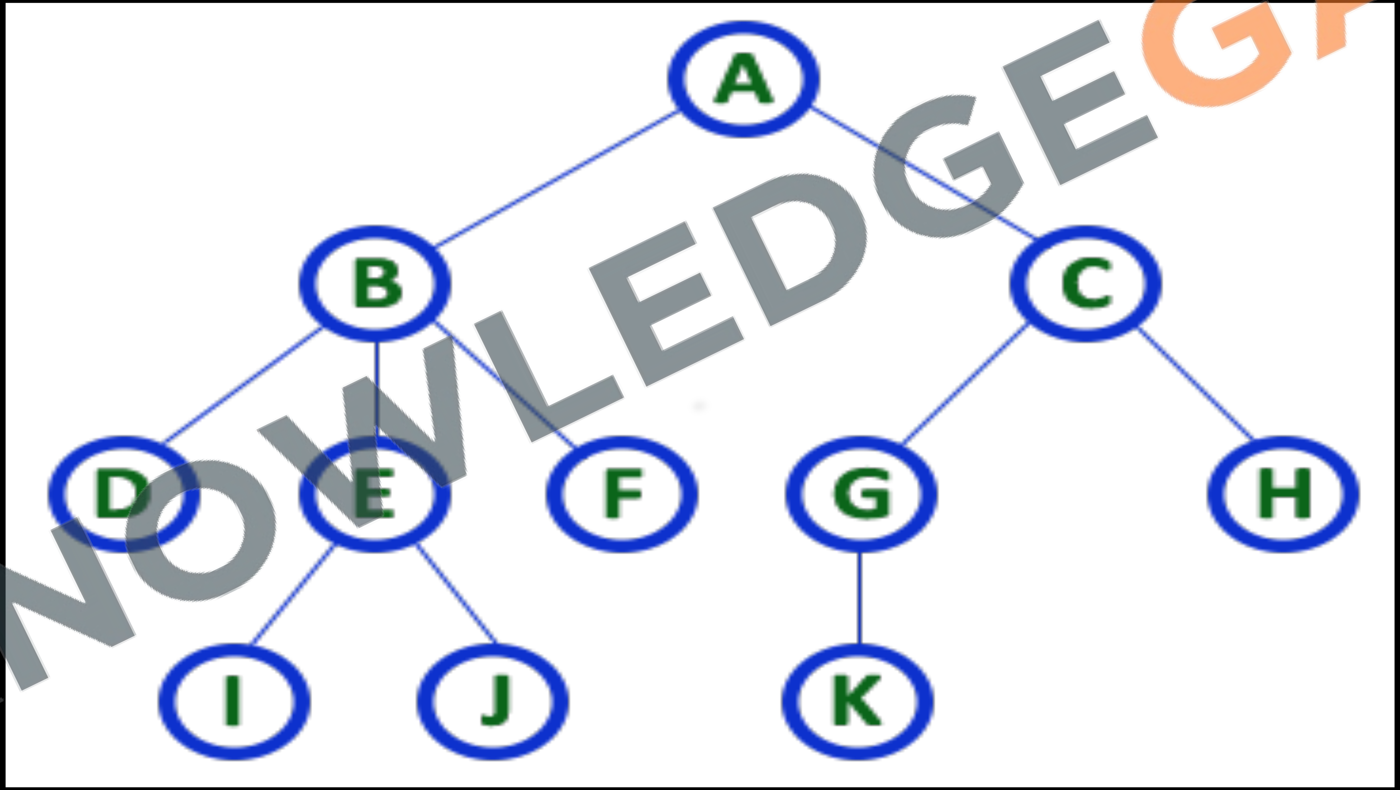
http://www.knowledgegate.in/GATE

Polynomial Addition Using Linked List

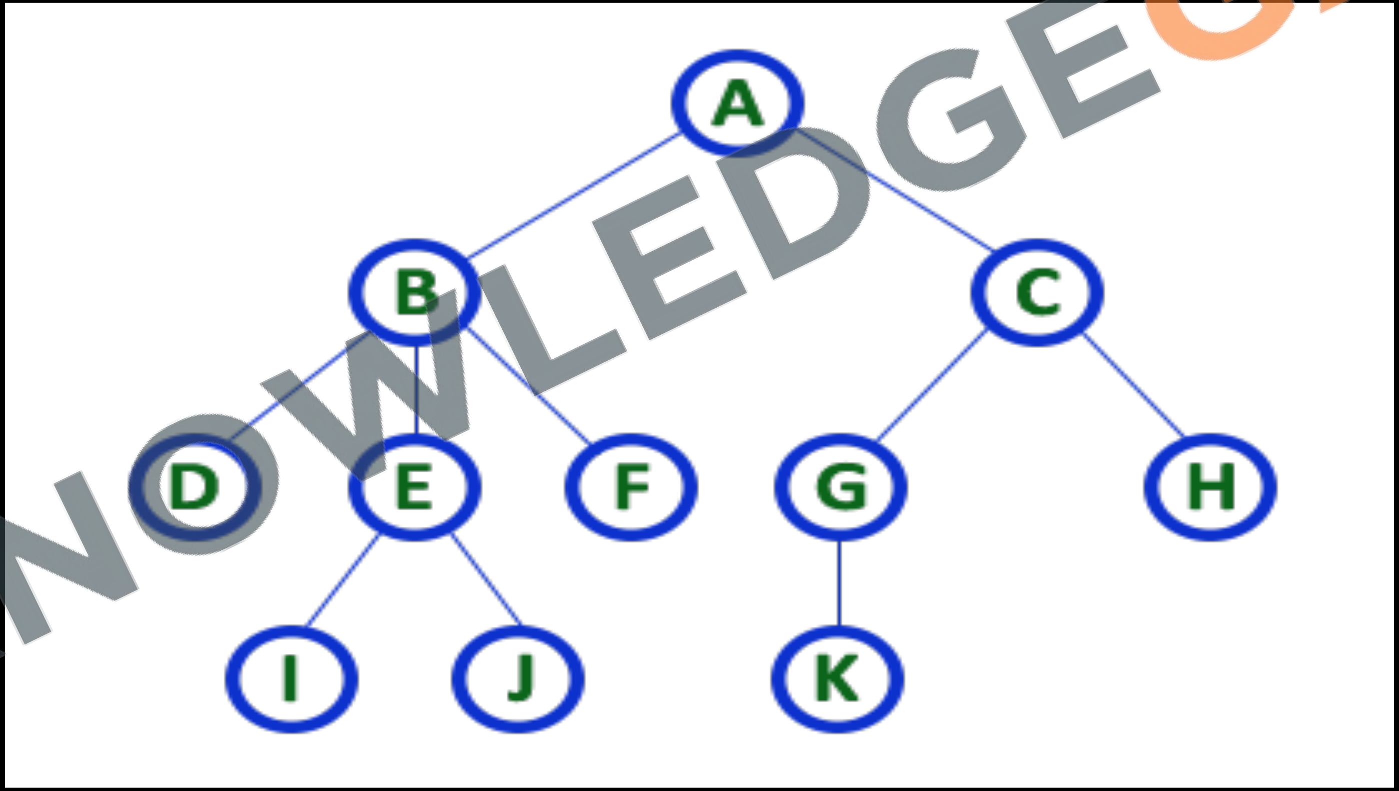
http://www.knowledgegate.in/GATE

Tree

- The tree is one of the most powerful, flexible, versatile and nonlinear advanced data structures, it represents hierarchical relationship existing between several data items. it is used in wide range of applications.

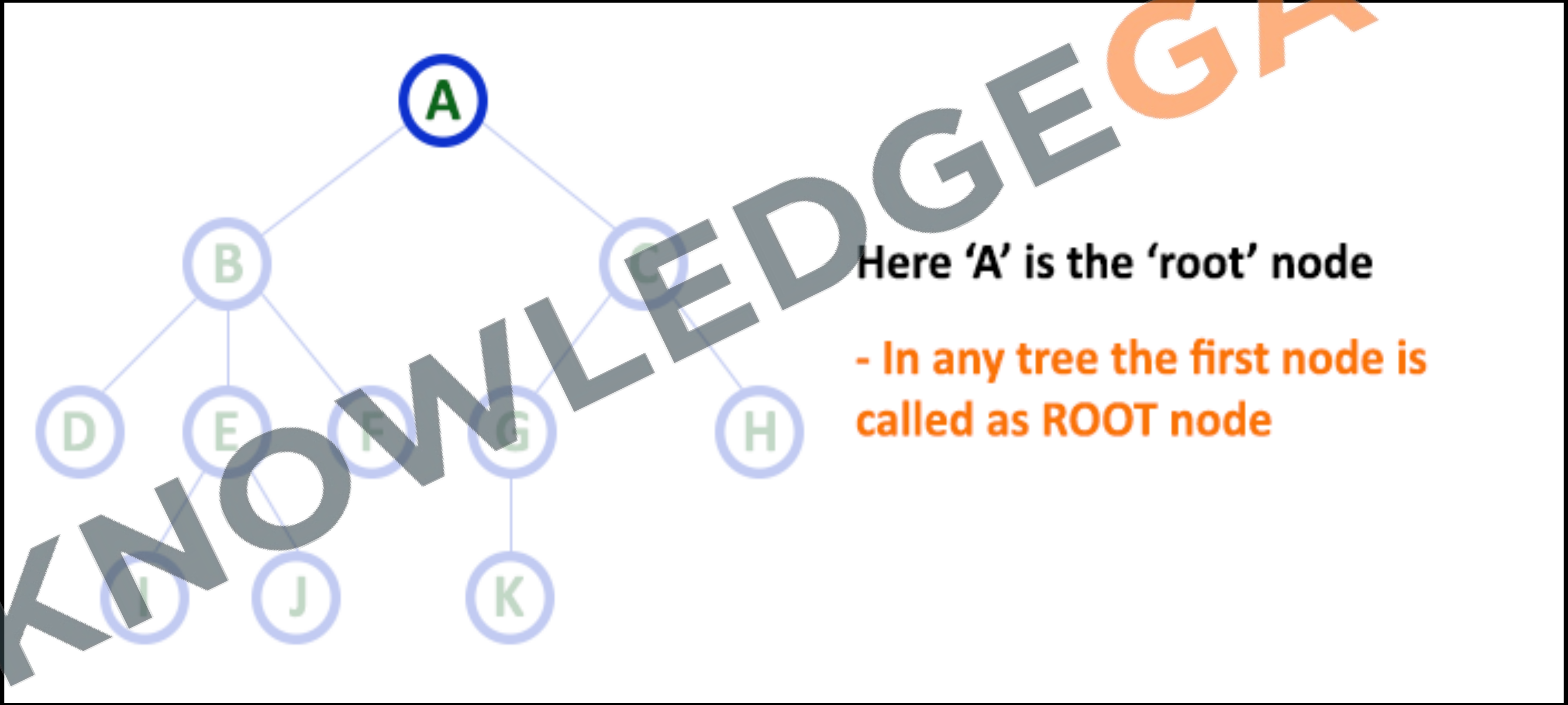


- A tree is a finite set of one or more data items(nodes) such that
  - There is a special data item called root of the tree
  - And its remaining data items are partitioned into number of mutually exclusive (disjoint) subsets, each of which is itself a tree and they are called subtree. i.e. Every node (exclude a root) is connected by a directed edge *from* exactly one other node; A direction is: *parent -> children*



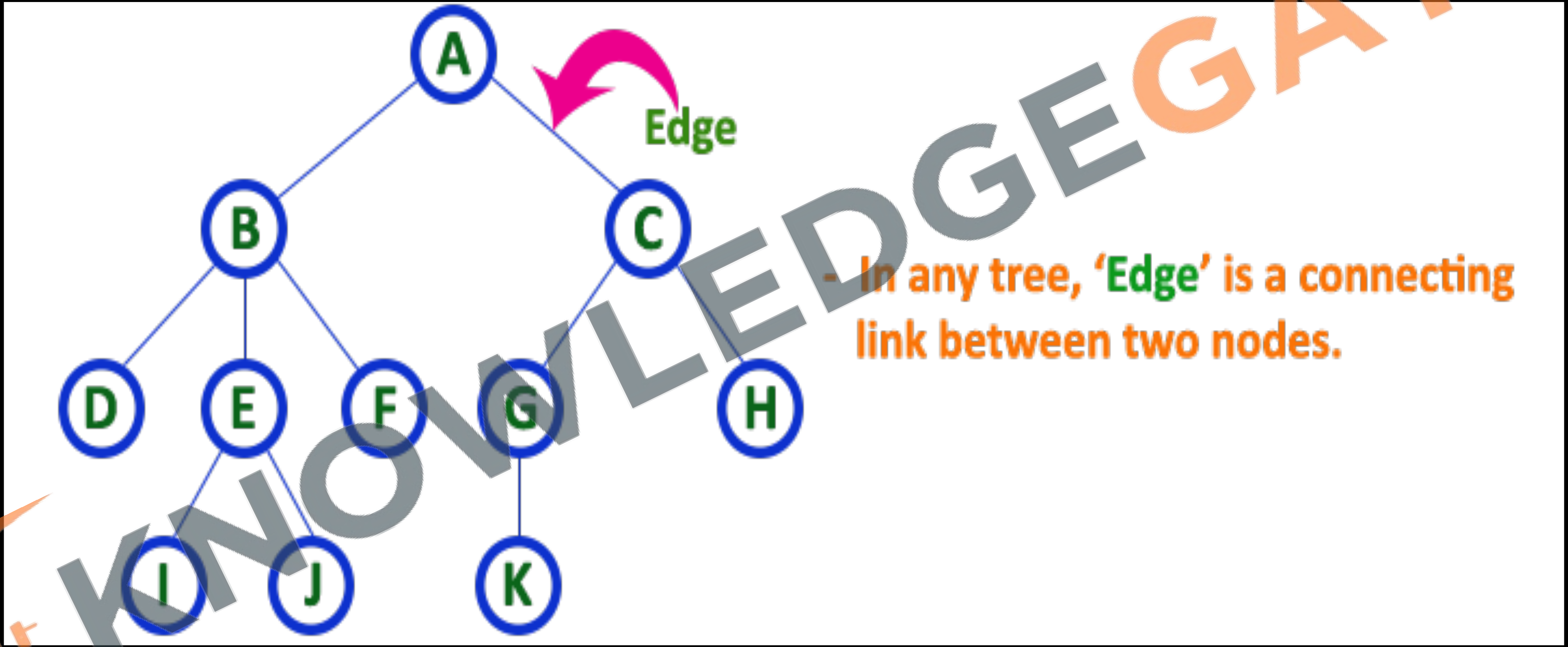
Root

- The first/Top most node is called as Root Node. We always have exactly one root node in every tree. We can say that root node is the origin of tree data structure.


A
B
C
D
E
F
G
H
I
J
K
Here 'A' is the 'root' node
- In any tree the first node is called as ROOT node

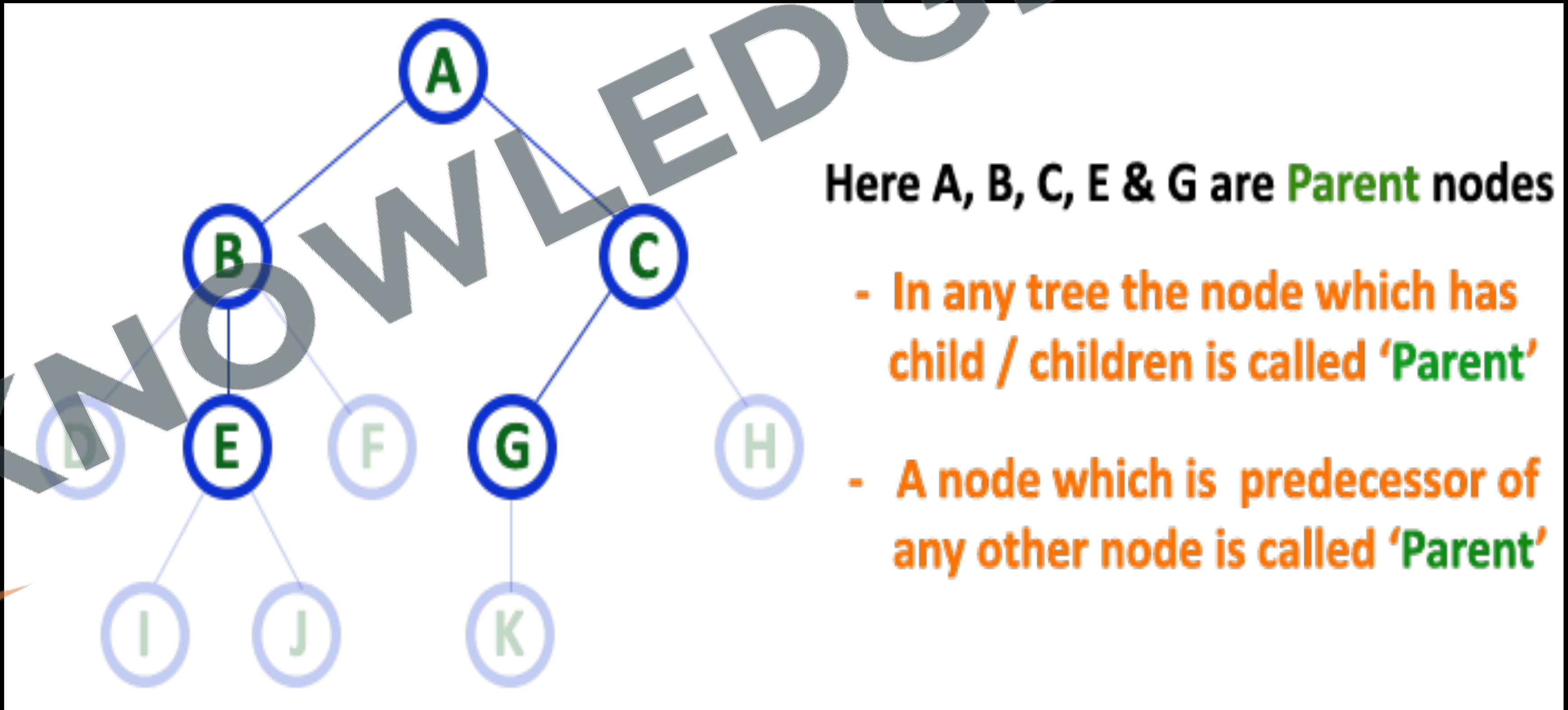
Edge

- In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with 'N' number of nodes there will be exactly of 'N-1' number of edges.


A
B
C
D
E
F
G
H
I
J
K
Edge
- In any tree, 'Edge' is a connecting link between two nodes.

Parent

- In a tree data structure, the node which is predecessor of any node is called as PARENT NODE.
- In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "The node which has child / children".

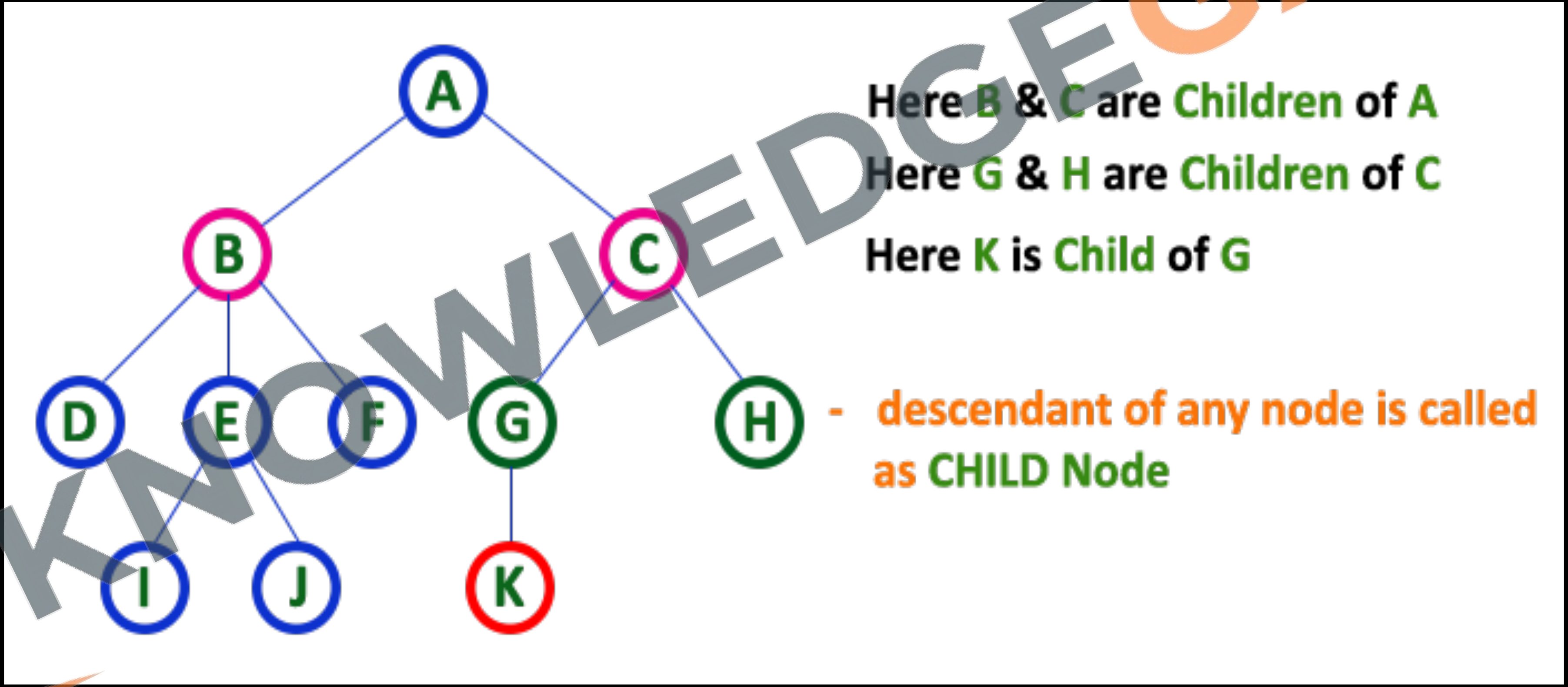


Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

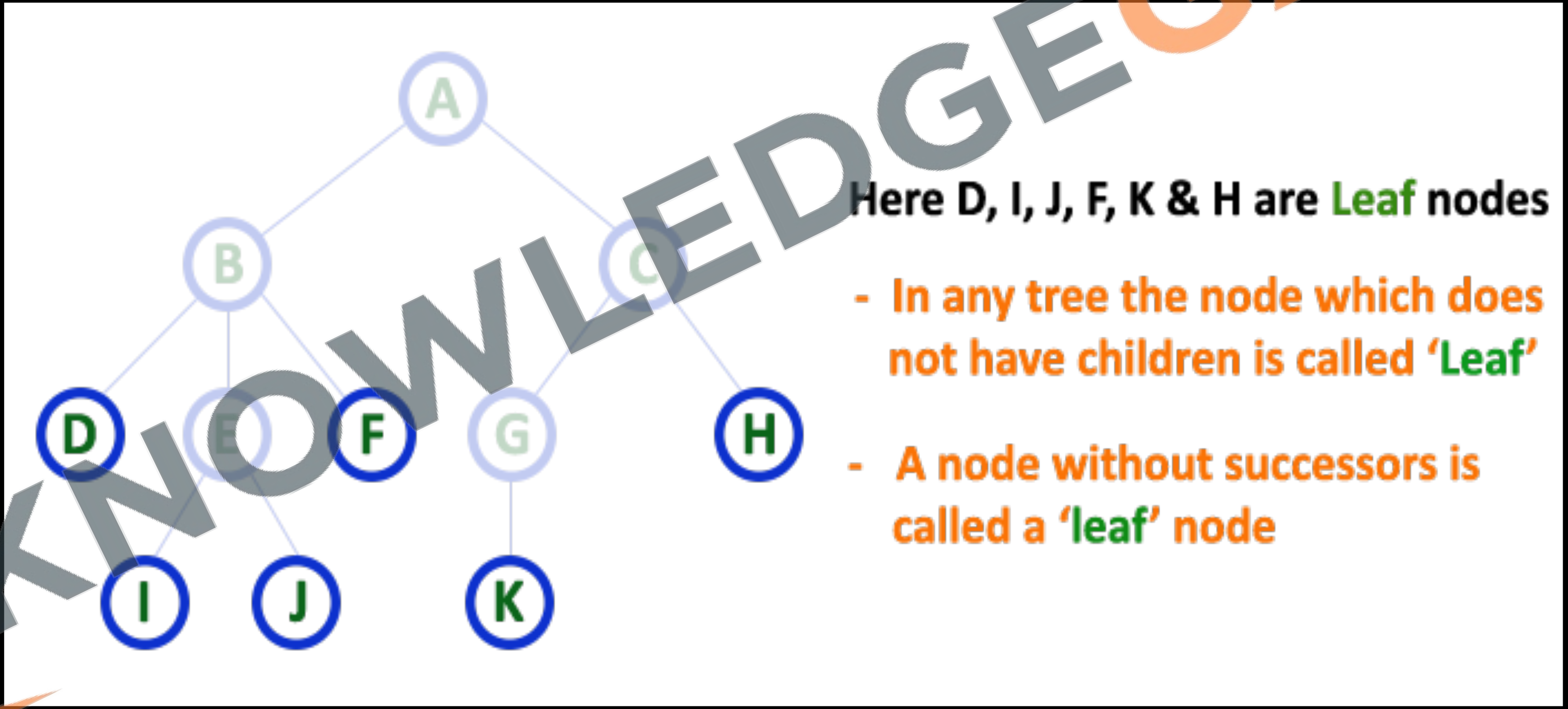
Child

- In a tree data structure, the node which is descendant of any node is called as CHILD Node.
- In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.


A
B
C
D
E
F
G
H
I
J
K
Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G
- descendant of any node is called as CHILD Node

Leaf / External

- In a tree data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child.
- In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



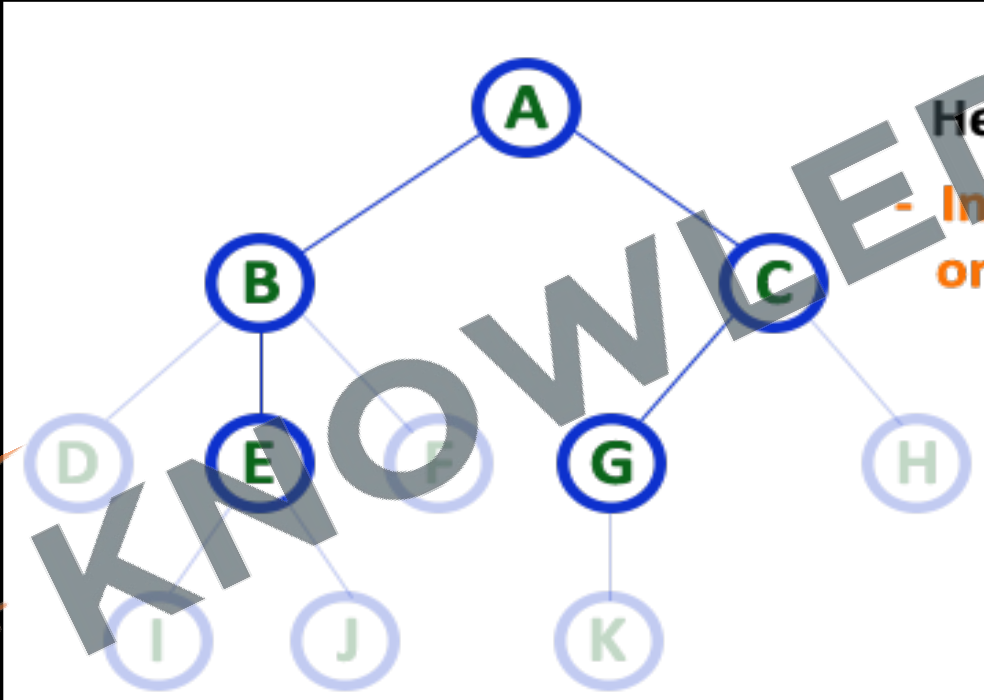
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

http://www.knowledgegate.in/GATE

Internal Nodes

- In a tree data structure, the node which has at least one child is called as INTERNAL Node. In simple words, an internal node is a node with at least one child.
- In a tree data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

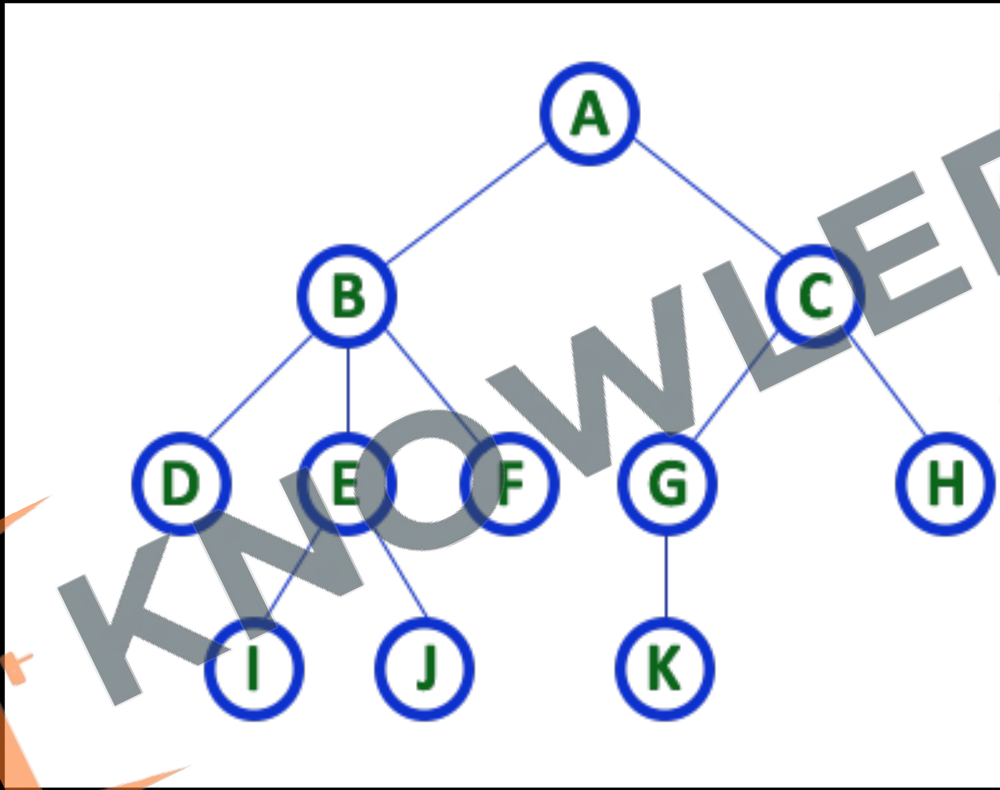


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

Degree

- In a tree data structure, the total number of children of a node is called as DEGREE of that Node. In simple words, the Degree of a node is total number of children it has.
- The highest degree allowed of a node in a tree is called as 'Degree of Tree'



Here Degree of B is 3

Here Degree of A is 2

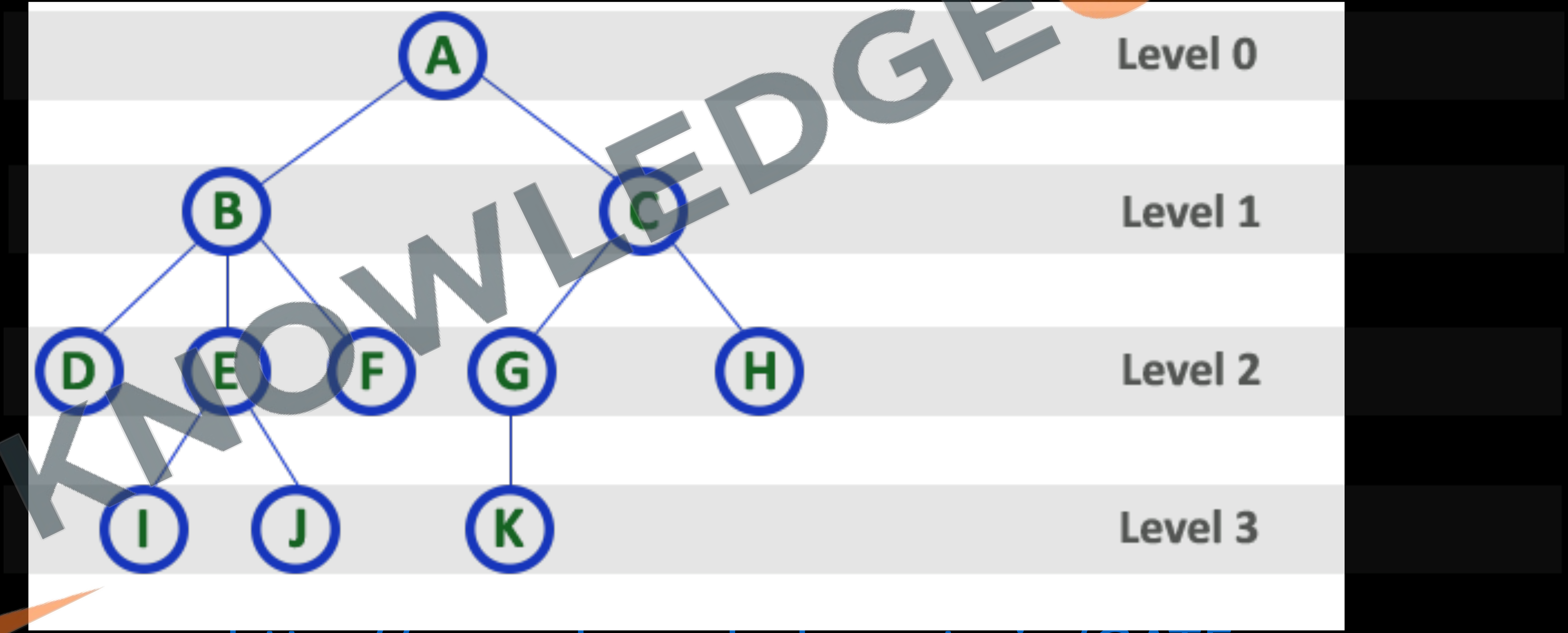
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

http://www.knowledgegate.in/GATE

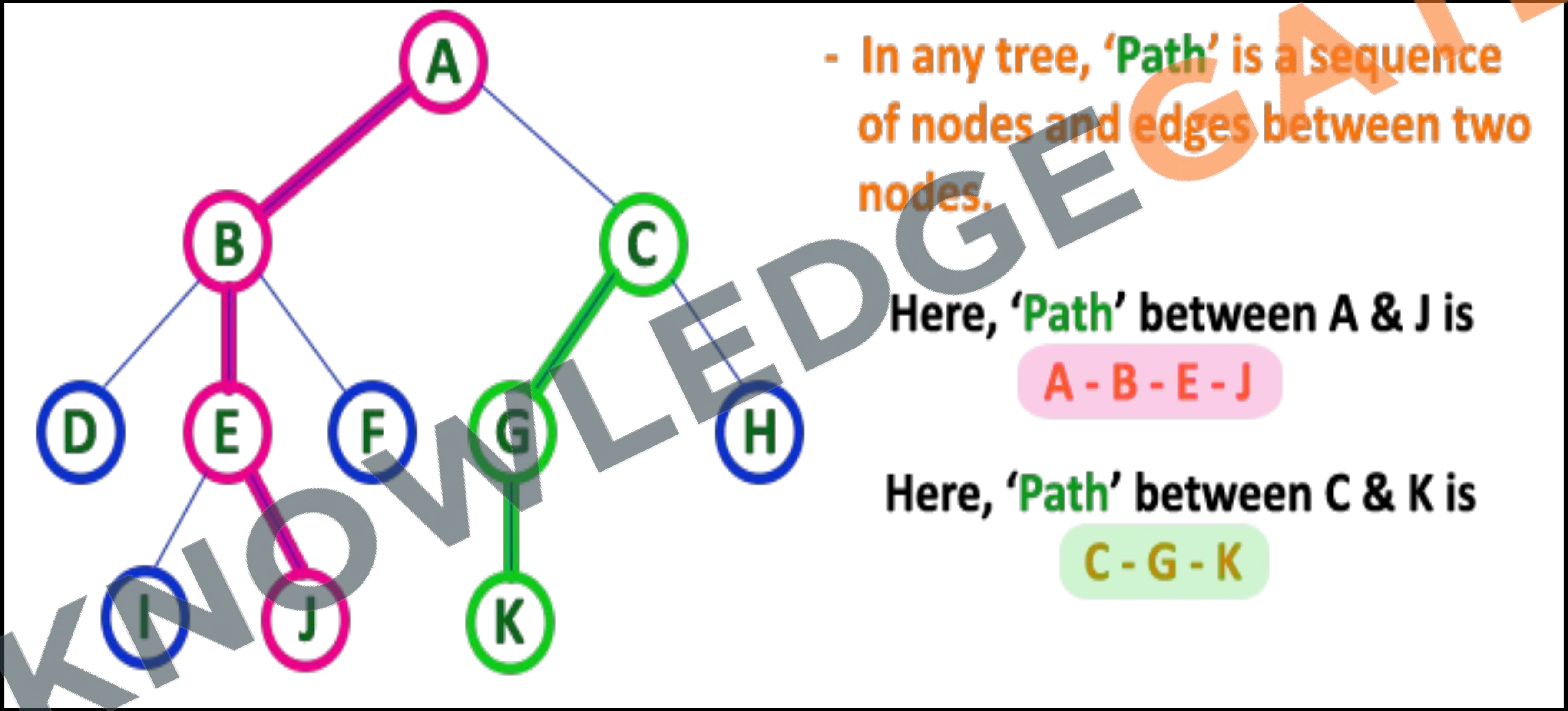
Level / Depth / Height

- In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on...
- In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



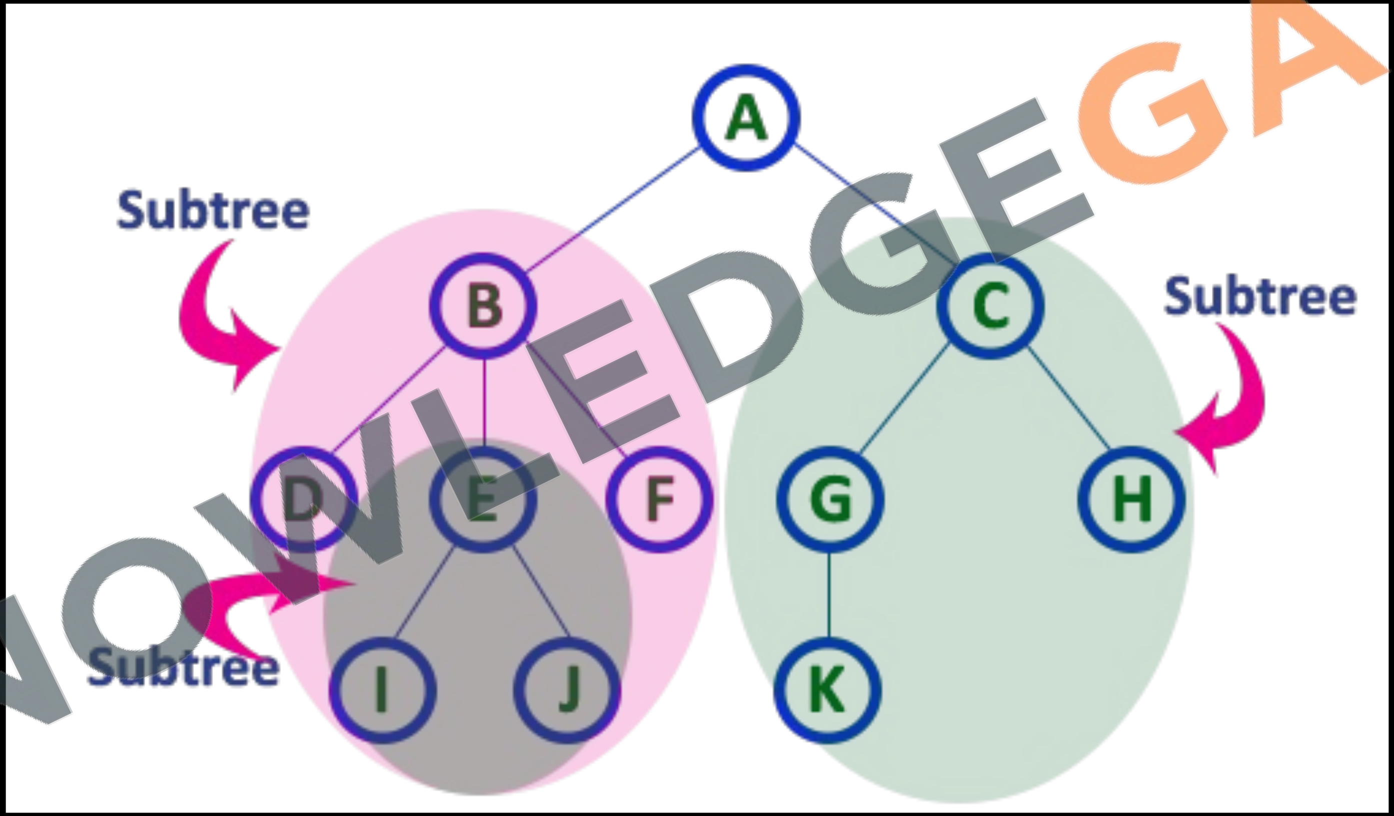
Path

- In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as PATH between that two Nodes. Length of a Path is total number of edge in that path. In below example the path A - B - E - J has length 4.


A
B
C
D
E
F
G
H
I
J
K
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.
Here, 'Path' between A & J is
A - B - E - J
Here, 'Path' between C & K is
C - G - K

Sub Tree

- In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

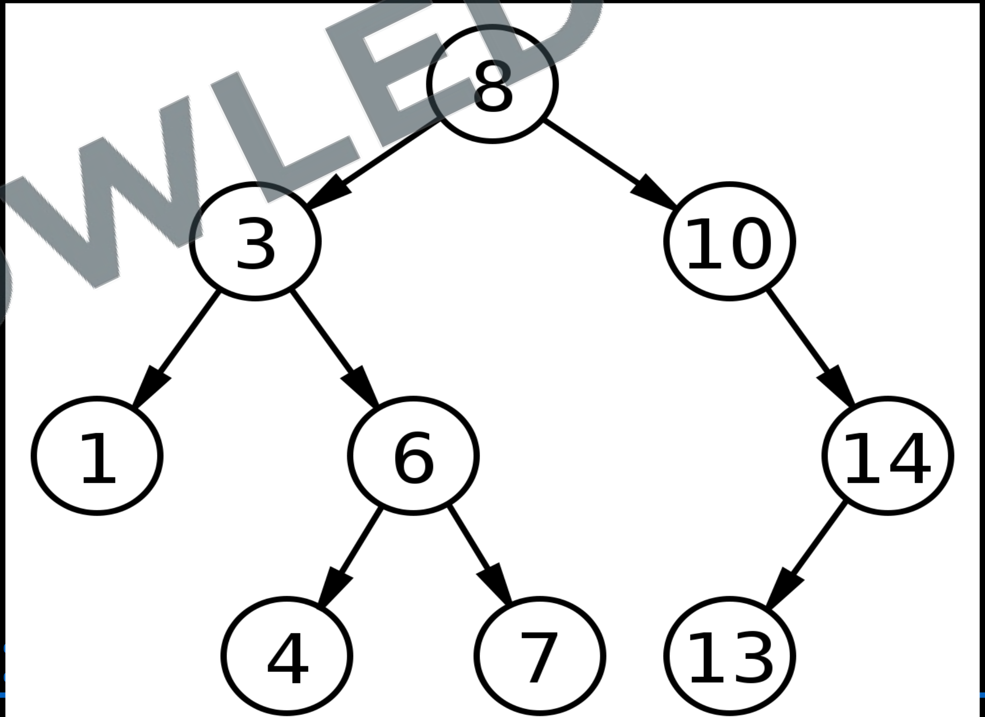


http://www.knowledgegate.in/GATE

Binary tree

- A binary tree T is defined as a finite set of elements called nodes such that,
  - T is empty (null tree)
  - T contain a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary tree T_1 and T_2
- Direct: - A tree T in which any node can have maximum two children (left and right)

```
struct node {
   int data;
   struct node* left;
   struct node* right;
}
```



http: ATE

Q Let T be a binary search tree with 15 nodes. The minimum and maximum possible heights of T are: ________

(A) 4 and 15 respectively
(B) 3 and 14 respectively
(C) 4 and 14 respectively
(D) 3 and 15 respectively

Q The height of a tree is the length of the longest root-to-leaf path in it. The maximum and minimum number of nodes in a binary tree of height 5 are

(A) 63 and 6, respectively

(B) 64 and 5, respectively

(C) 32 and 6, respectively

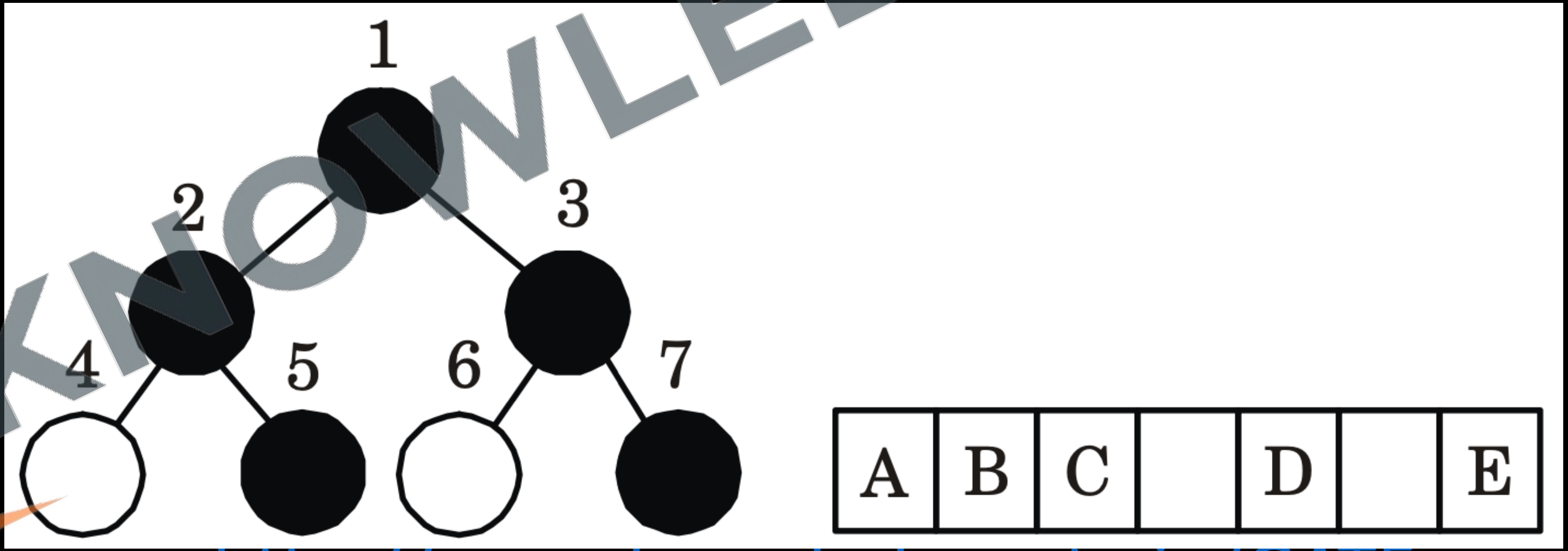
(D) 31 and 5, respectively

Q The height of a binary tree is the maximum number of edges in any root to leaf path. The maximum number of nodes in a binary tree of height h is:

a) $2^{h}-1$ **b)** $2^{h-1}-1$ **c)** $2^{h+1}-1$ **d)** 2^{h+1}

Binary tree representation using array

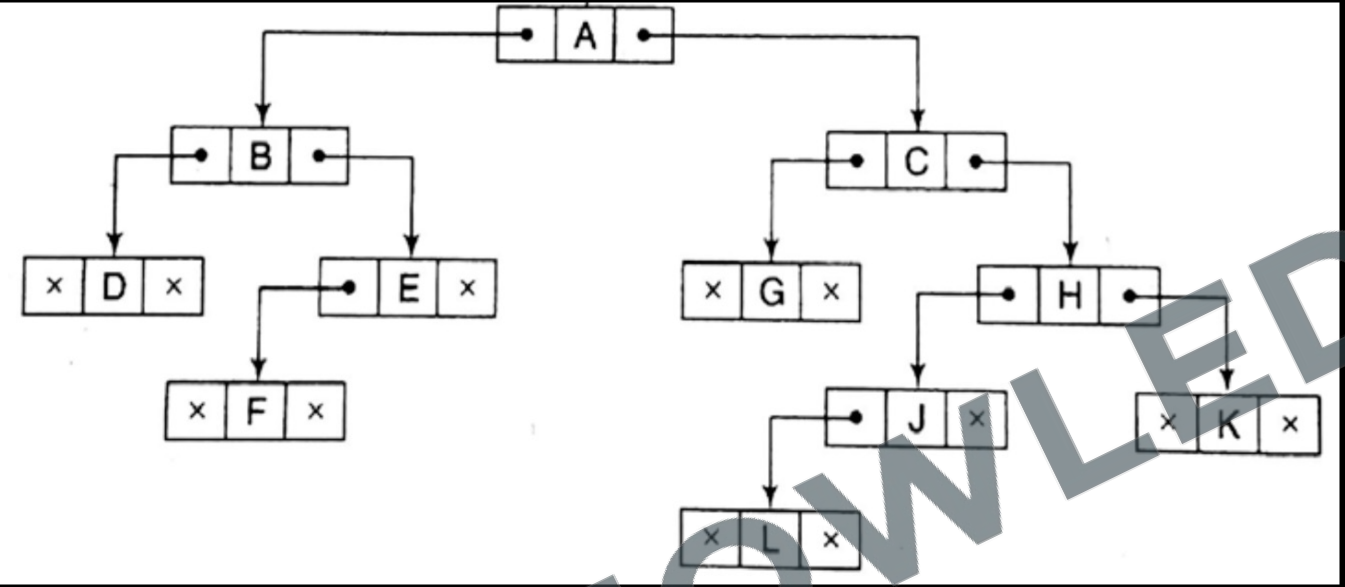
- Binary tree can be represented using an array
- General representation
- The root is at index ‘1’
- For any given node at position ‘i’
    - Left Child is at position 2*i
    - Right Child is at position 2*i + 1
- If a node does not have a left or right child, that position in the array remains empty or is filled with a special value indicating it's vacant (like null or -1)

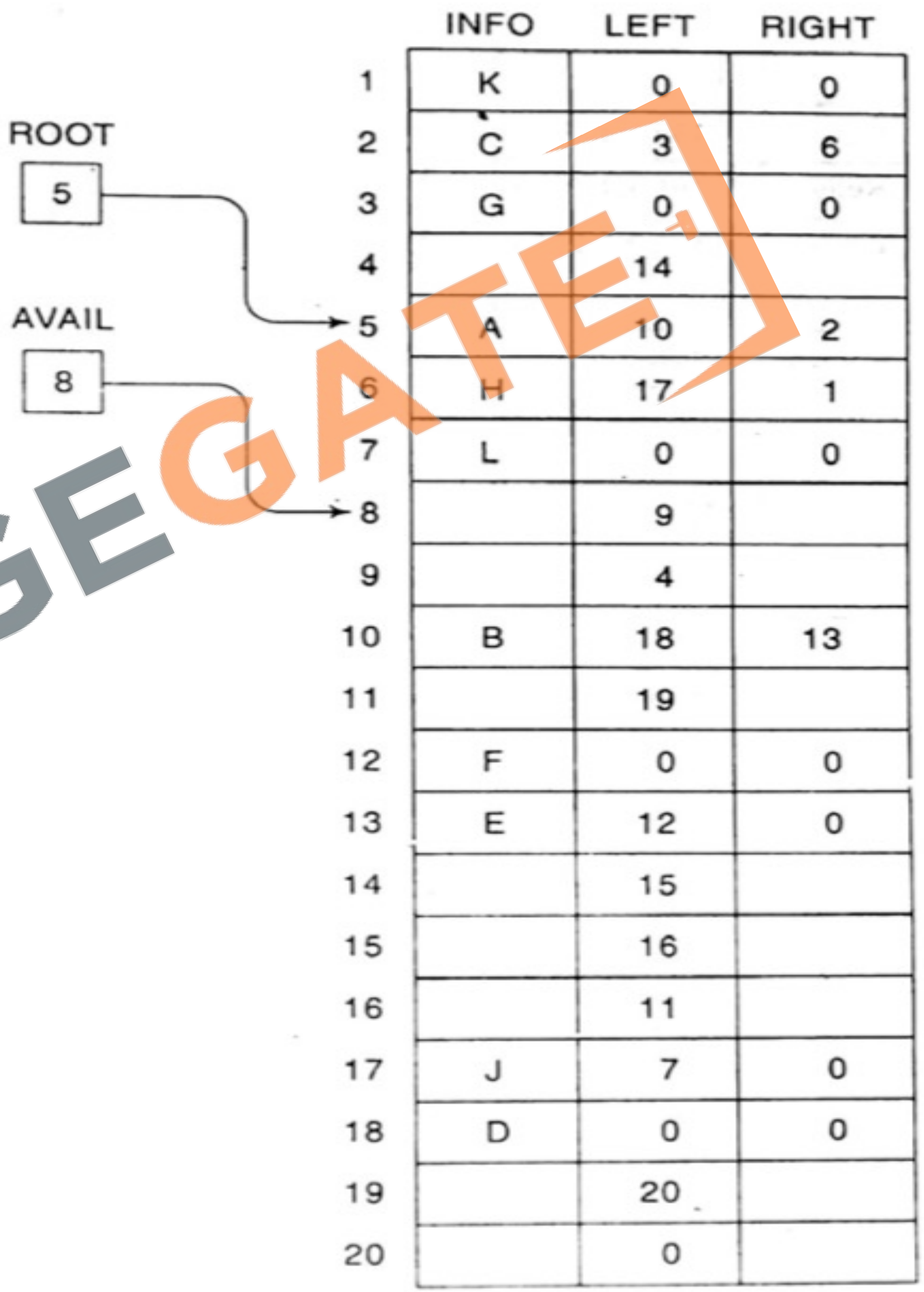


- In fact, a binary tree that has *n*-elements may require an array of size up to 2^n (including position 0) for its representation.

Linked representation of binary tree

1. Consider a binary tree *T* which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT.
2. First of all, each node *N* of *T* will correspond to a location *K* such that :
   a. INFO[*K*] contains the data at the node *N*.
   b. LEFT[*K*] contains the location of the left child of node *N*.
   c. RIGHT[*K*] contains the location of the right child of node *N*.
3. ROOT will contain the location of the root *R* of *T*.
4. If any subtree is empty, then the corresponding pointer will contain the null value.
5. If the tree *T* itself is empty, then ROOT will contain the null value.
6. INFO may actually be a linear array of records or a collection of parallel arrays.



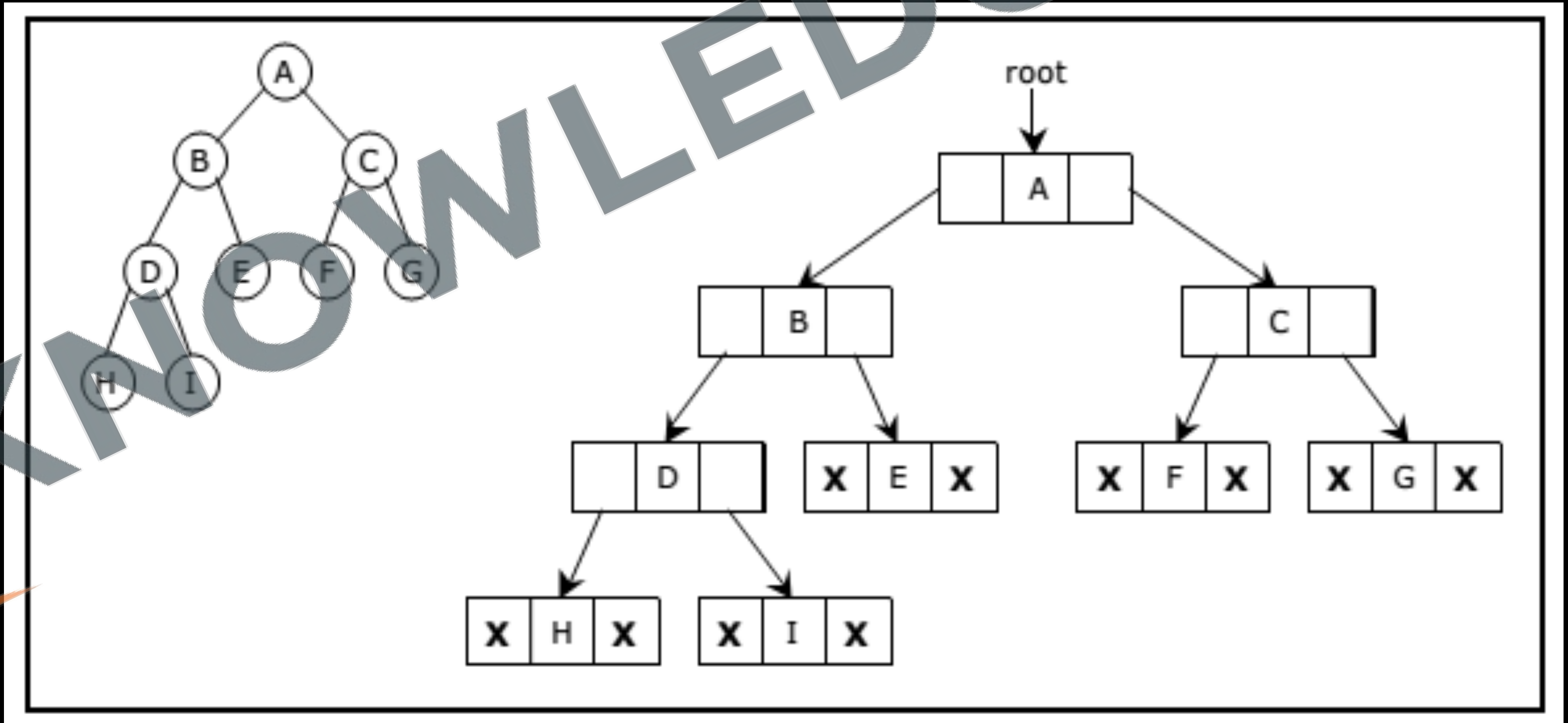


	INFO	LEFT	RIGHT
1	K	0	0
2	C	3	6
3	G	0	0
4		14	
5	A	10	2
6	H	17	1
7	L	0	0
8		9	
9		4	
10	B	18	13
11		19	
12	F	0	0
13	E	12	0
14		15	
15		16	
16		11	
17	J	7	0
18	D	0	0
19		20	
20		0	

Binary tree representation using Linked List

- A binary tree can be efficiently represented using a linked list structure where each node of the tree is represented by a separate node in the linked list. This linked structure is typically referred to as a "node-based" representation.

- Each node in the linked list contains the following components:
  - **Data**: The value stored in the node.
  - **Left Pointer**: A pointer pointing to the left child node.
  - **Right Pointer**: A pointer pointing to the right child node.

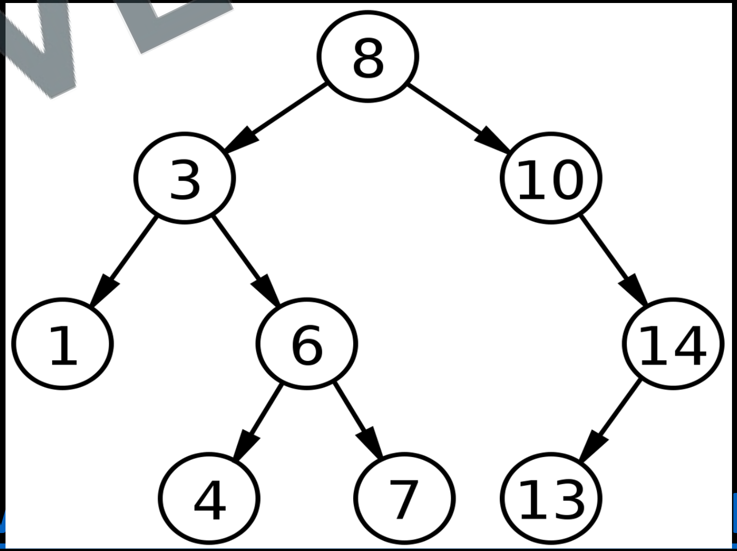


```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
        int data;
        struct Node* left;
        struct Node* right;
} Node;

Node* createNode(int data)
{
        Node* newNode = (Node*) malloc(sizeof(Node));
        if (!newNode)
        {
                printf("Memory error\n");
                exit(1);
        }
        newNode->data = data;
        newNode->left = NULL;
        newNode->right = NULL;
        return newNode;
}
```

```
void insert(Node** root, int data)
{
    if (*root == NULL)
    {
        *root = createNode(data);
        return;
    }
    if (data < (*root)->data)
    {
        insert(&((*root)->left), data);
    }
    else
    {
        insert(&((*root)->right), data);
    }
}
```

Traversal of binary tree

- The process of visiting (checking and/or updating) each node in a tree data structure, exactly once in called tree traversal. Such traversals are classified by the order in which the nodes are visited.
- Unlike linked lists, one-dimensional arrays and other linear data structures, which are canonically traversed in linear order, trees may be traversed in multiple ways.
- They may be traversed in depth-first or breadth-first order. There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order. Beyond these basic traversals, various more complex or hybrid schemes are possible, such as depth-limited searches like iterative deepening depth-first search.



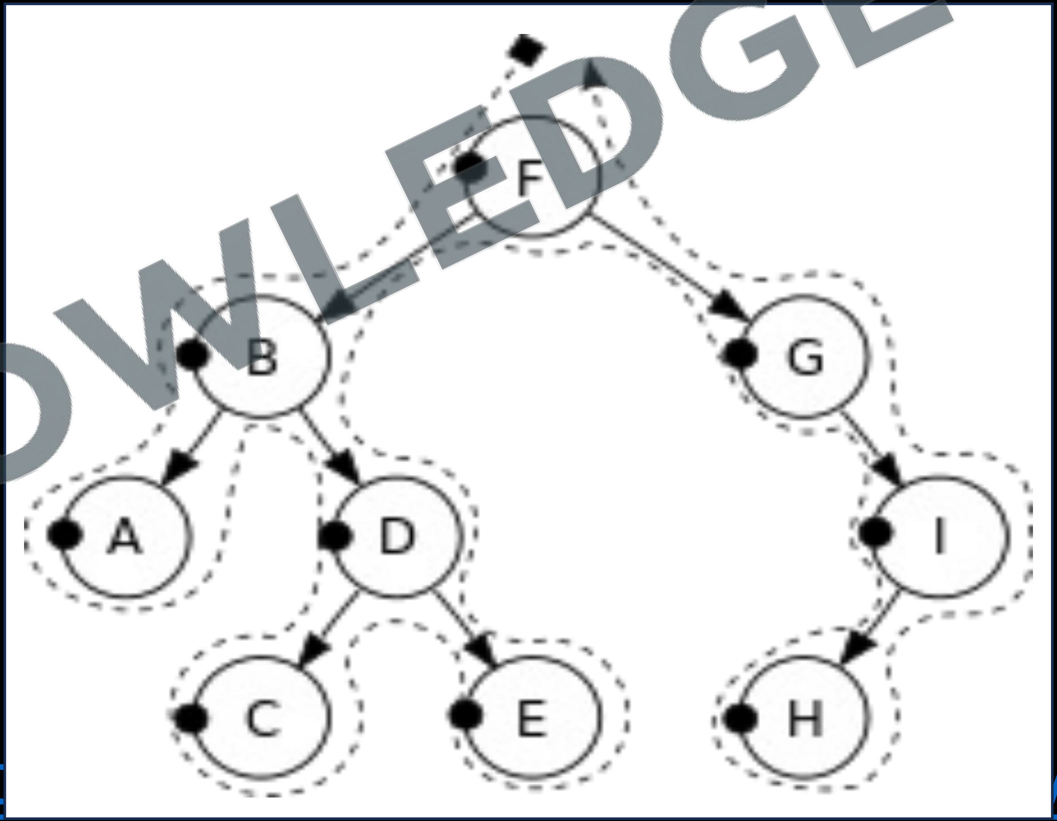
http://w n/GATE

- Some applications do not require that the nodes be visited in any particular order as long as each node is visited precisely once. For other applications, nodes must be visited in an order that preserves some relationship.
- These steps can be done *in any order*. If (L) is done before (R), the process is called left-to-right traversal, otherwise it is called right-to-left traversal. The following methods show left-to-right traversal:

Pre-order (Root L R)

Pre-order: F, B, A, D, C, E, G, I, H.

- Check if the current node is empty or null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.

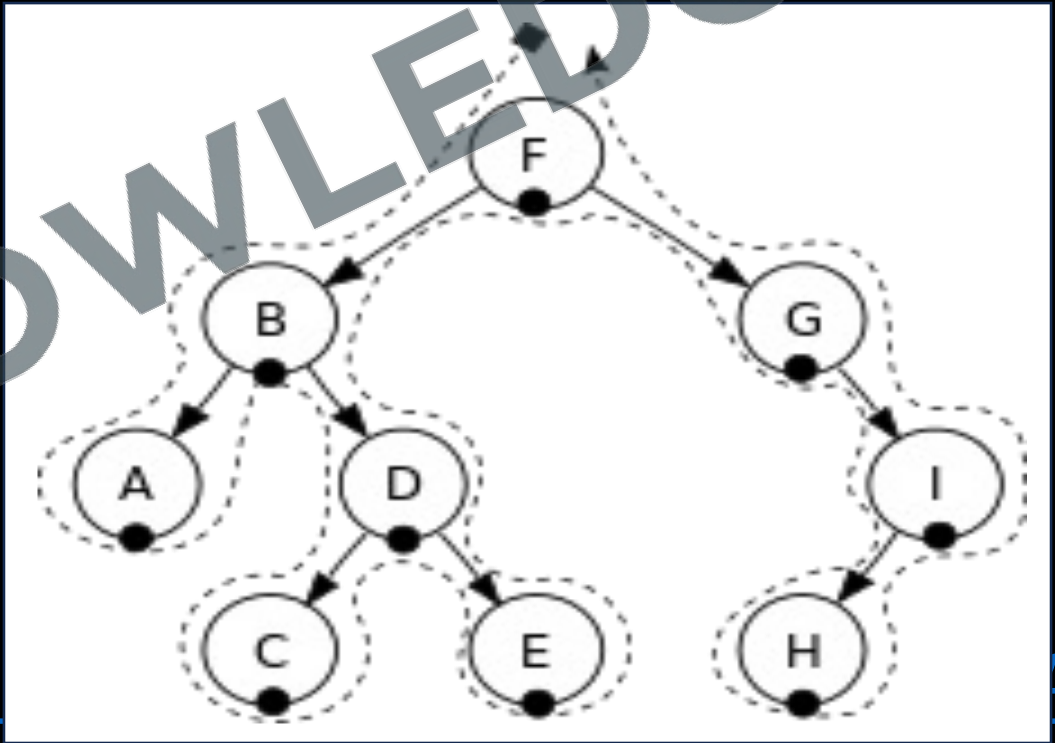


htt ATE

In-order (L root R)

In-order: A, B, C, D, E, F, G, H, I.

- Check if the current node is empty or null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.
- In a binary search tree, in-order traversal retrieves data in sorted order.

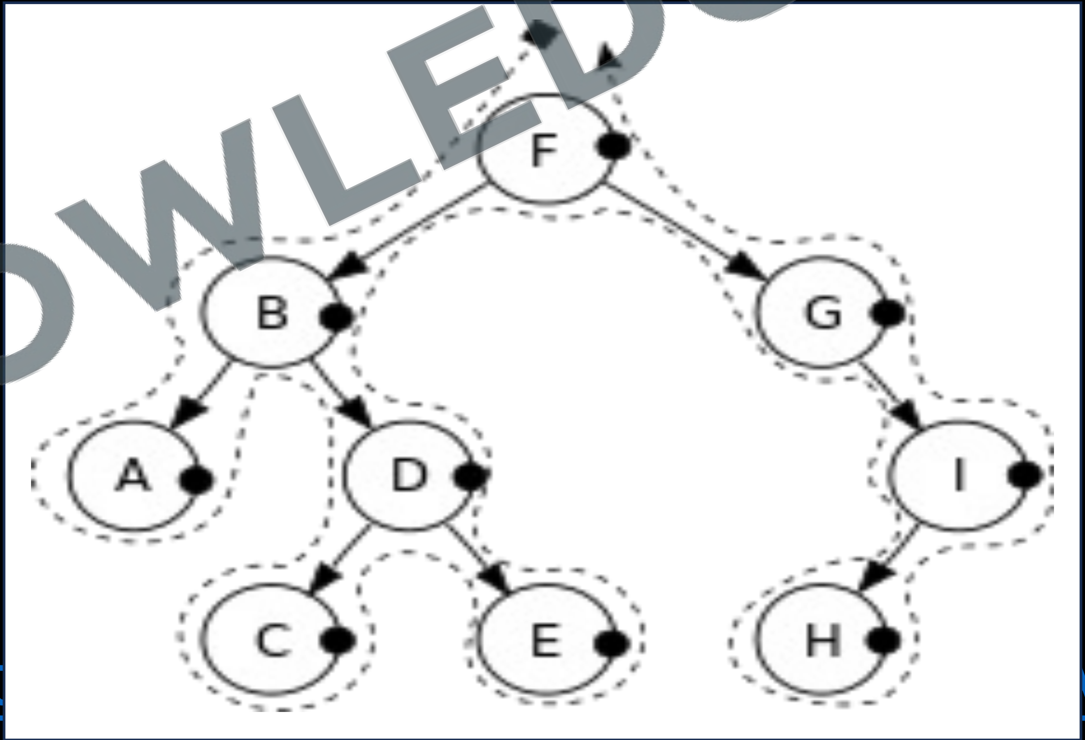


htt ATE

Post-order (L R Root)

Post-order: A, C, E, D, B, H, I, G, F.

- Check if the current node is empty or null.
- Traverse the left subtree by recursively calling the post-order function.
- Traverse the right subtree by recursively calling the post-order function.
- Display the data part of the root (or current node).

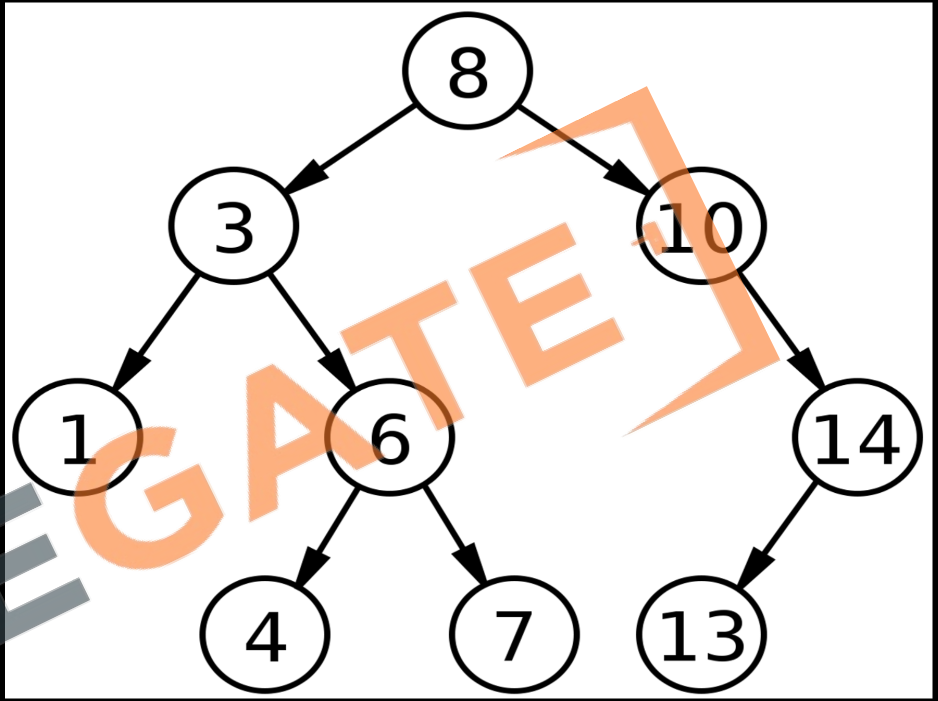


Q The post order traversal of a binary tree is 8, 9, 6, 7, 4, 5, 2, 3, 1. The inorder traversal of the same tree is 8, 6, 9, 4, 7, 2, 5, 1, 3. The height of a tree is the length of the longest path from the root to any leaf. The height of the binary tree above is ________.

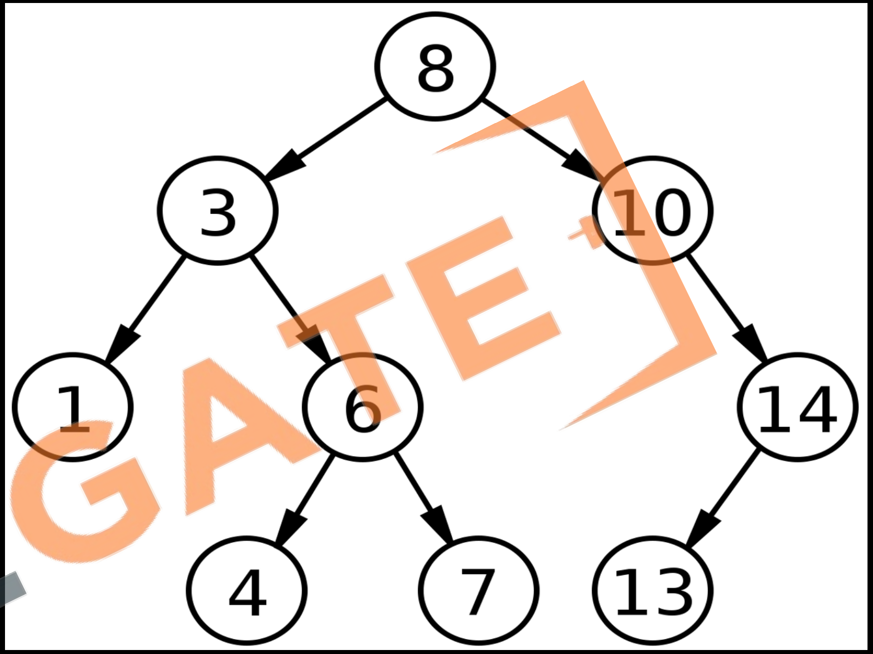
Inorder : D B H E A I F J C G

Preorder : A B D E H C F I J G

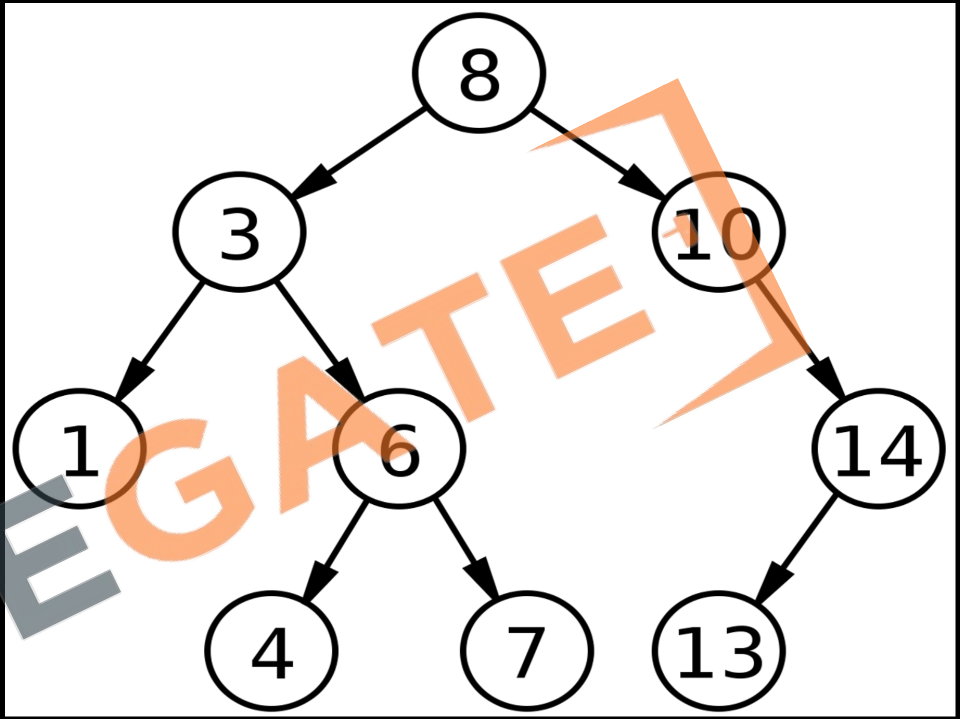
```
void inorderTraversal(Node* root)
{
    if (root == NULL)
        return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
```



```
void preorderTraversal(Node* root)
{
    if (root == NULL)
        return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
```

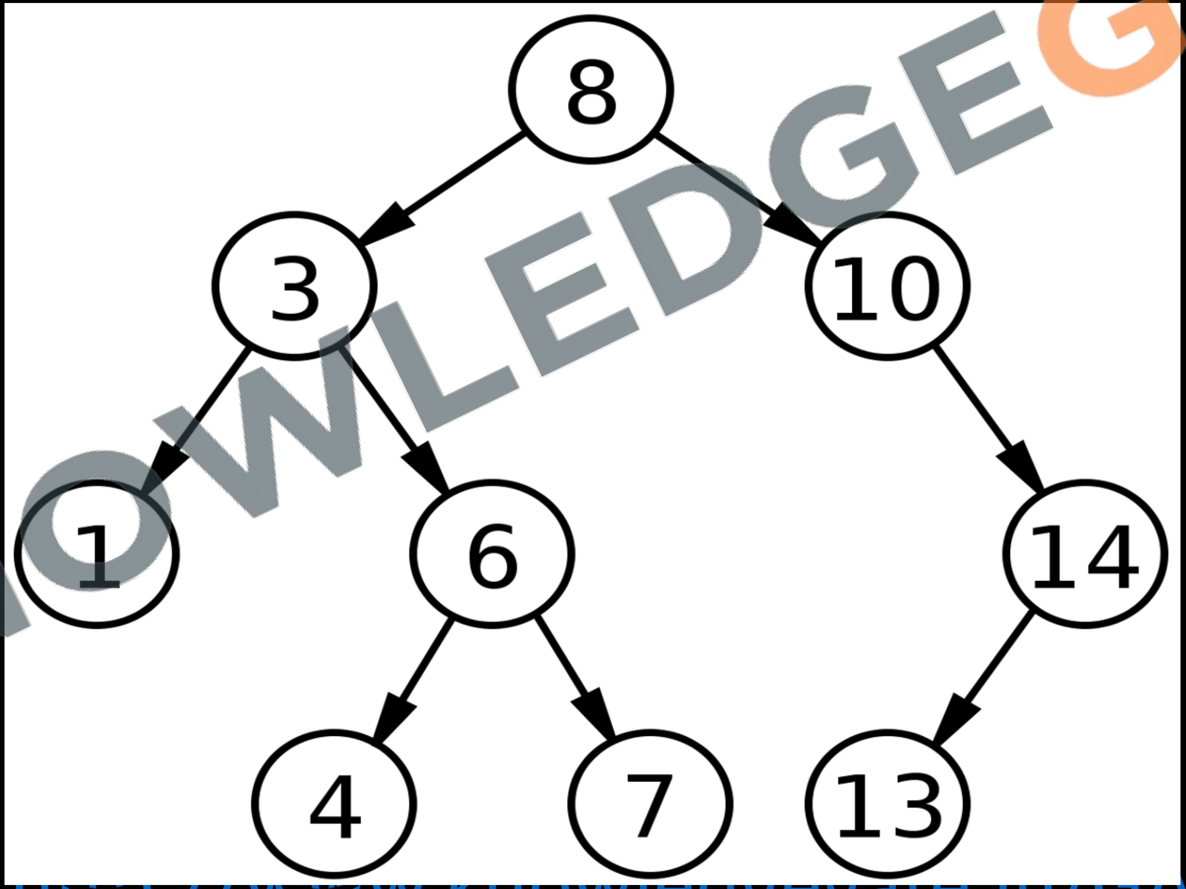


```
void postorderTraversal(Node* root)
{
    if (root == NULL)
        return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}
```



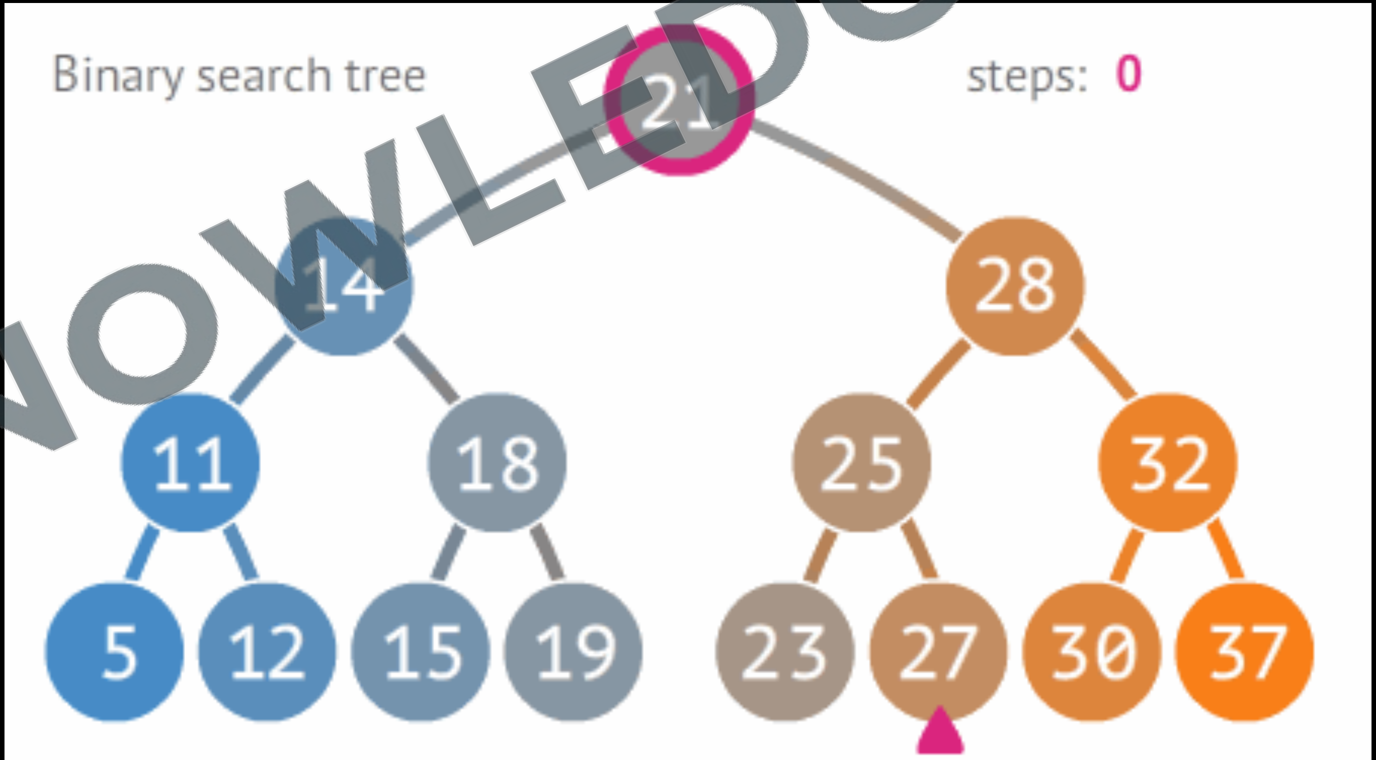
Binary search tree / Ordered tree / Sorted binary tree

- A binary search tree (BST) is a binary tree in which left subtree of a node contains a key less than the node’s key and right subtree of a node contains only the nodes with key greater than the node’s key. Left and right sub tree must each also be a binary search tree.



Searching

- We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node.
- If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree.
- This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found after a *null* subtree is reached, then the key is not present in the tree.



Operations

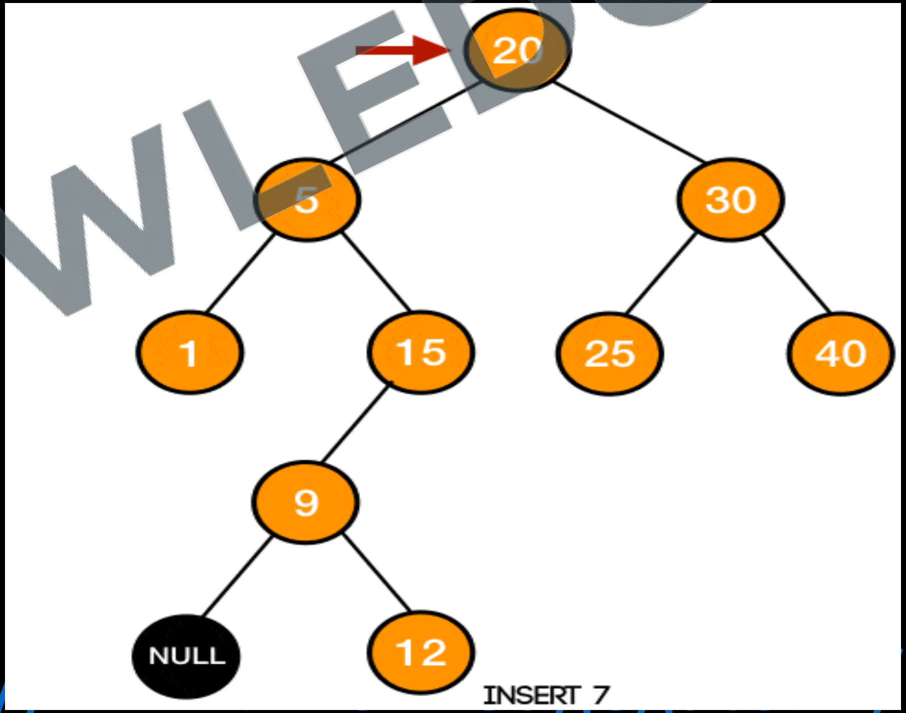
- Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

Q While inserting the elements 71, 65, 84, 69, 67, 83 in an empty binary search tree (BST) in the sequence shown, the element in the lowest level is

(A) 65 **(B)** 67 **(C)** 69 **(D)** 83

Insertion

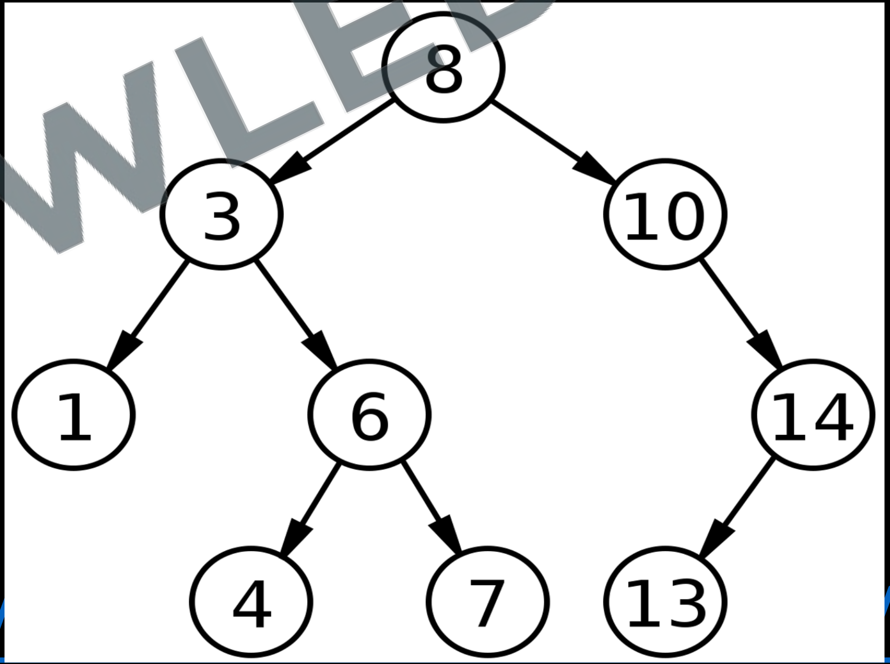
- Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before.
- Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'new Node') as its right or left child, depending on the node's key.
- In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

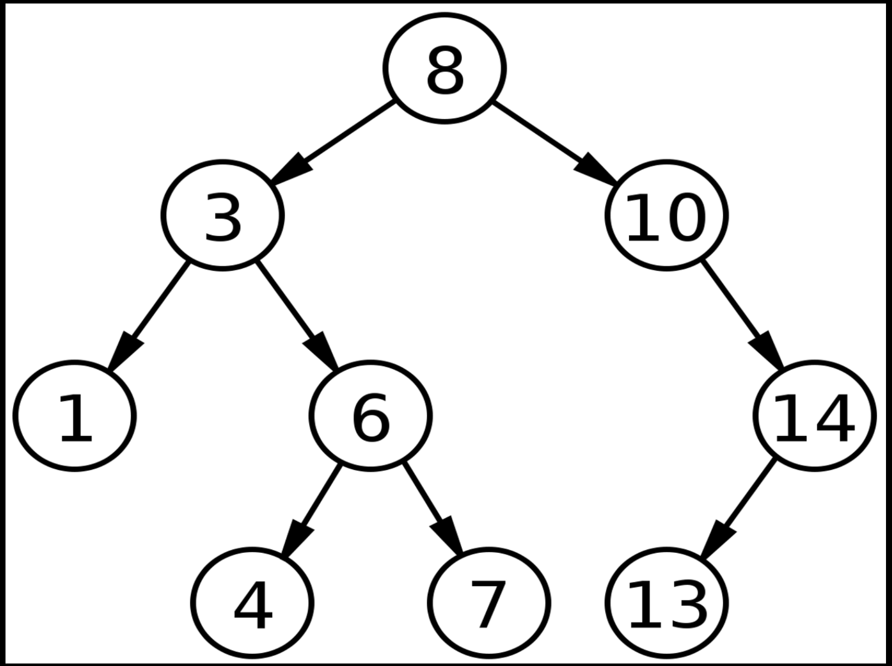


http:/ GATE

Deletion

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *D*. Do not delete *D*.
- Instead, choose either its in-order predecessor node or its in-order successor node as replacement node *E* (s. figure). Copy the user values of *E* to *D*.
- If *E* does not have a child simply remove *E* from its previous parent *G*. If *E* has a child, say *F*, it is a right child. Replace *E* with *F* at *E*'s parent.




8
3
10
1
6
14
4
7
13

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

- The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithm such as in-order traversal can be very efficient; they are also easy to code

Q The pre-order traversal of a binary search tree is given by 12, 8, 6, 2, 7, 9, 10, 16, 15, 19, 17, 20. Then the post-order traversal of this tree is:

a) 2, 6, 7, 8, 9, 10, 12, 15, 16, 17, 19, 20

b) 2, 7, 6, 10, 9, 8, 15, 17, 20, 19, 16, 12

c) 7, 2, 6, 8, 9, 10, 20, 17, 19, 15, 16, 12

d) 7, 6, 2, 10, 9, 8, 15, 16, 17, 20, 19, 12

Q Which of the following is/are correct inorder traversal sequence(s) of binary search tree(s)? **1**. 3, 5, 7, 8, 15, 19, 25
2. 5, 8, 9, 12, 10, 15, 25
3. 2, 7, 10, 8, 14, 16, 20
4. 4, 6, 7, 9, 18, 20, 25

a) 1 and 4 only **b)** 2 and 3 only **c)** 2 and 4 only **d)** 2 only

Q Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 55. Which of the following sequences CANNOT be the sequence of nodes examined?

(a) {10, 75, 64, 43, 60, 57, 55}

(b) 190, 12, 68, 34, 62, 45, 55}

(c) (9, 85, 47, 68, 43, 57, 55}

(d) {79, 14, 72, 56, 16, 53, 55}

```
Node* searchBST(Node* root, int data)
{
    if (root == NULL || root->data == data)
    {
        return root;
    }
    if (data < root->data)
    {
        return searchBST(root->left, data);
    }
    return searchBST(root->right, data);
}
```

```
void insertBST(Node** root, int data)
{
	if (*root == NULL)
	{
		*root = createNode(data);
		return;
	}
	if (data < (*root)->data)
	{
		insertBST(&((*root)->left), data);
	}
	else if (data > (*root)->data)
	{
		insertBST(&((*root)->right), data);
	}
	else
	{
		printf("Element %d already exists in the BST.\n", data);
	}
}
```

```
Node* deleteBST(Node* root, int data)
{
        if (!root)
                return root;
        if (data < root->data)
        {
                root->left = deleteBST(root->left, data);
        }
        else if (data > root->data)
        {
                root->right = deleteBST(root->right, data);
        }
        else
        {
                if (!root->left)
                {
                        Node* temp = root->right;
                        free(root);
                        return temp;
                }
                else if (!root->right)
                {
                        Node* temp = root->left;
                        free(root);
                        return temp;
                }
                Node* temp = findMinValueNode(root->right);
                root->data = temp->data;
                root->right = deleteBST(root->right, temp->data);
        }
        return root;
}
```

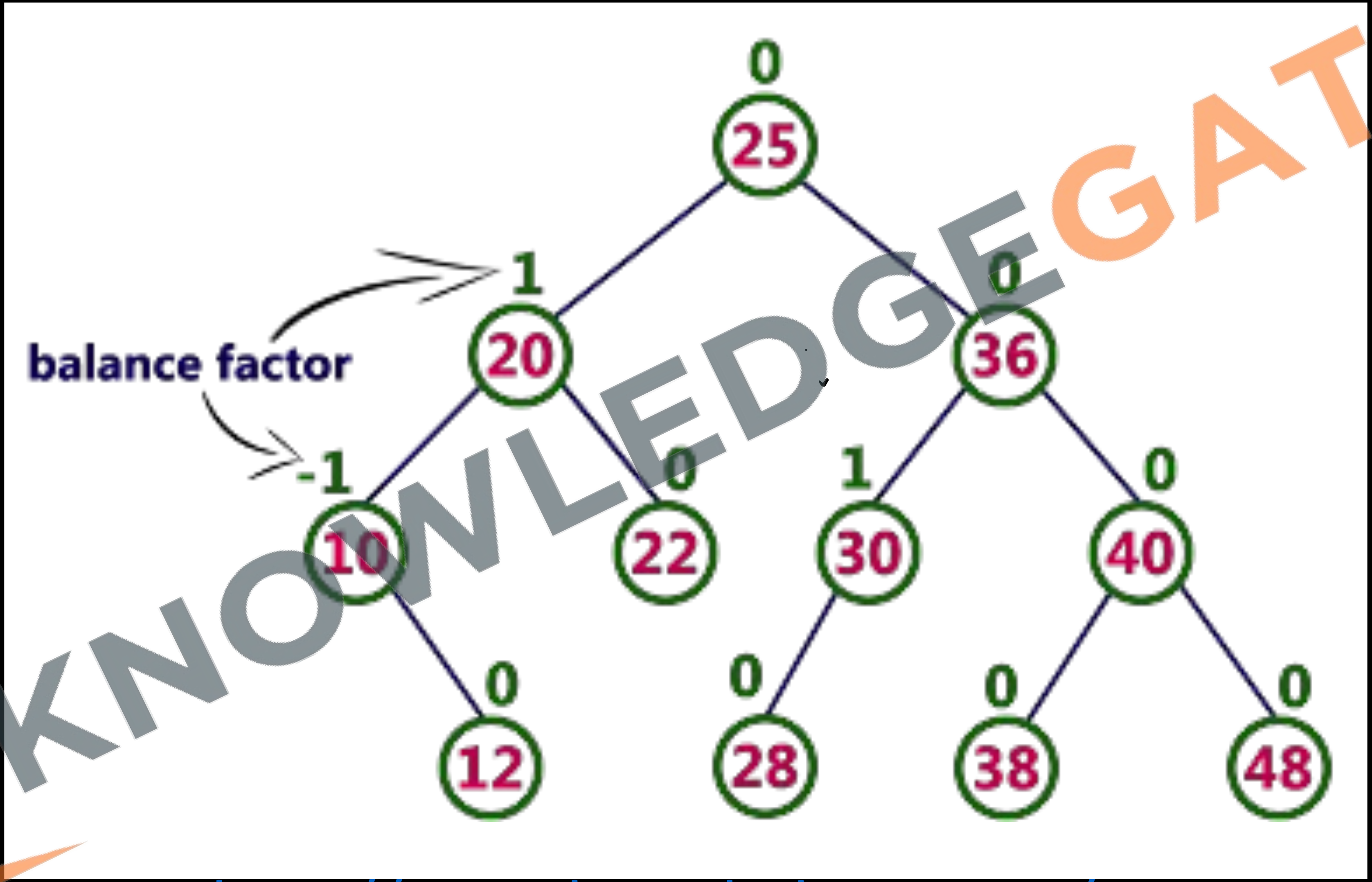
AVL tree

- In computer science, an **AVL tree** (named after inventors **A**delson-**V**elsky and **L**andis) is a self-balancing binary search tree. It was the first such data structure to be invented.

Adelson-Velsky

Landis

- In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.



http://www.knowledgegate.in/GATE

Balance factor

- In a binary tree the *balance factor* of a node N is defined to be the height difference Balance Factor(N): = Height (LeftSubtree(N)) – Height (RightSubtree(N)) of its two child subtrees.

- A binary tree is defined to be an *AVL tree* if the invariant Balance Factor(N) ∈ {–1, 0, +1} holds for every node N in the tree.

- A node N with Balance Factor(N) > 0 is called "left-heavy“
- One with Balance Factor(N) < 0 is called "right-heavy“
- One with Balance Factor(N) = 0 is sometimes simply called "balanced".

Insertion in an AVL tree

- Insert a node similarly as we do in binary search tree.
- After insertion start checking the balancing factor of each node in a bottom up fashion that is from newly inserted node towards the root.
- Stop on the first node whose balancing factor is violated and go two steps towards the newly inserted nodes. watch the movement, which is identified as the problem.

Problem	Solution
LL	R
RR	L
LR	LR
RL	RL

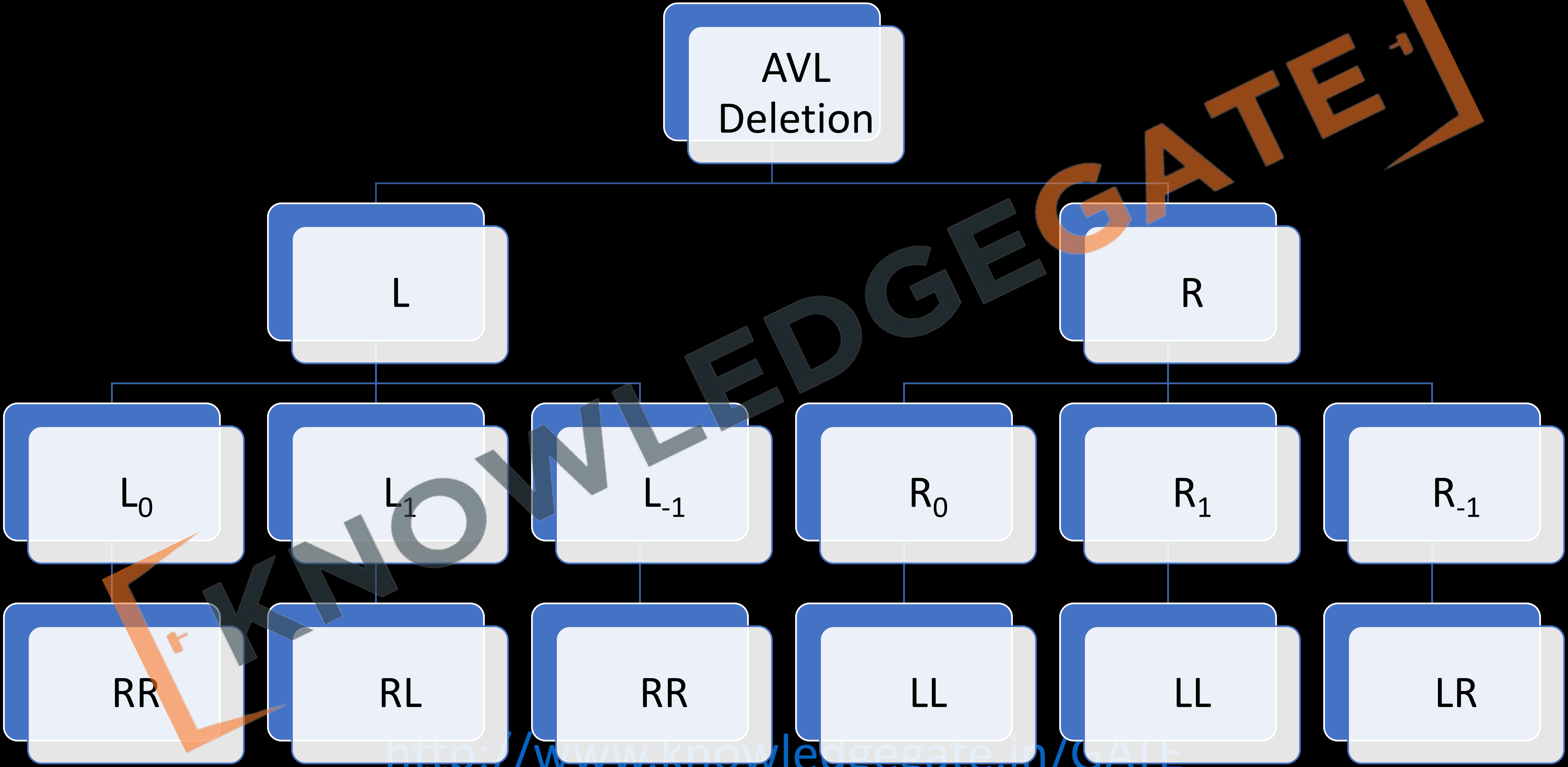
Q Consider an empty AVL tree and insert the following nodes in sequence 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7?

Insertion in an AVL tree

- After every insertion at most two rotations are sufficient to balance the AVL tree

Problem	Solution
LL	R
RR	L
LR	LR
RL	RL

Deletion in an AVL tree



Deletion in an AVL tree

Q delete the following nodes in sequence 2, 3, 10, 18, 4, 9, 14, 7, 15 ?

Analysis of AVL tree

- Lookup, insertion, and deletion all take O(log *n*) time in both the average and worst cases, where *n* is the number of nodes in the tree prior to the operation.
- Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

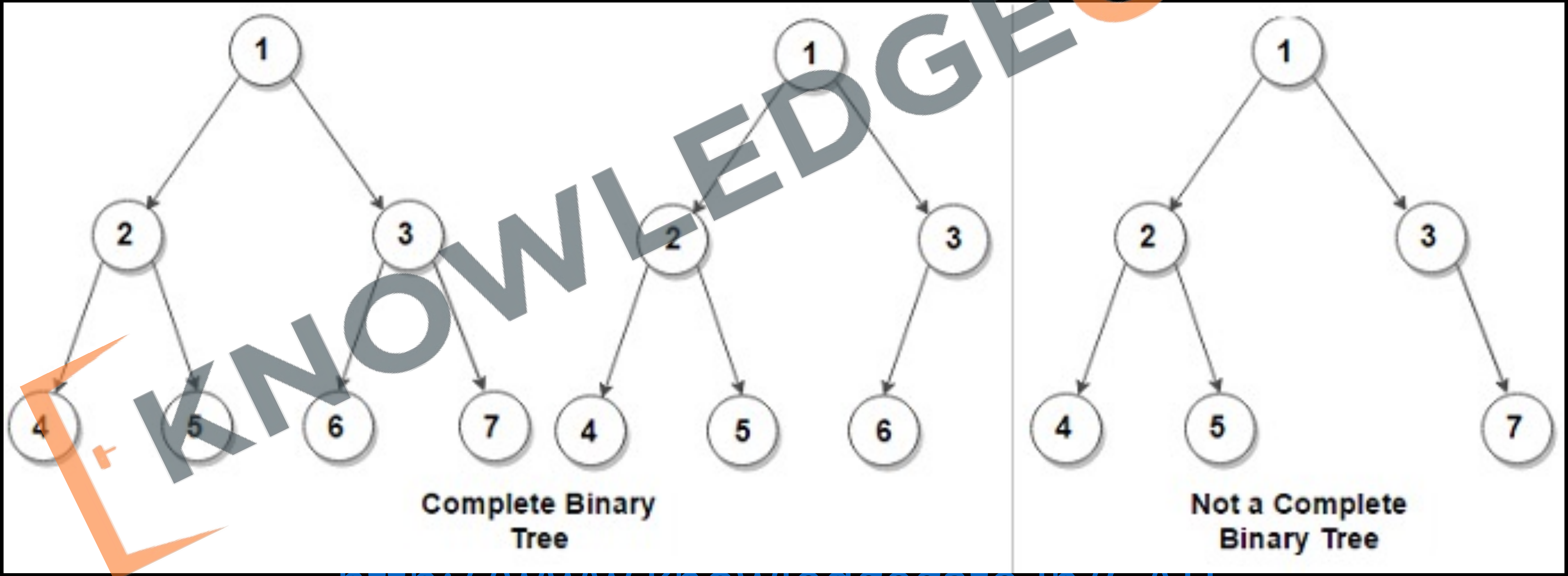
Binary search tree

Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

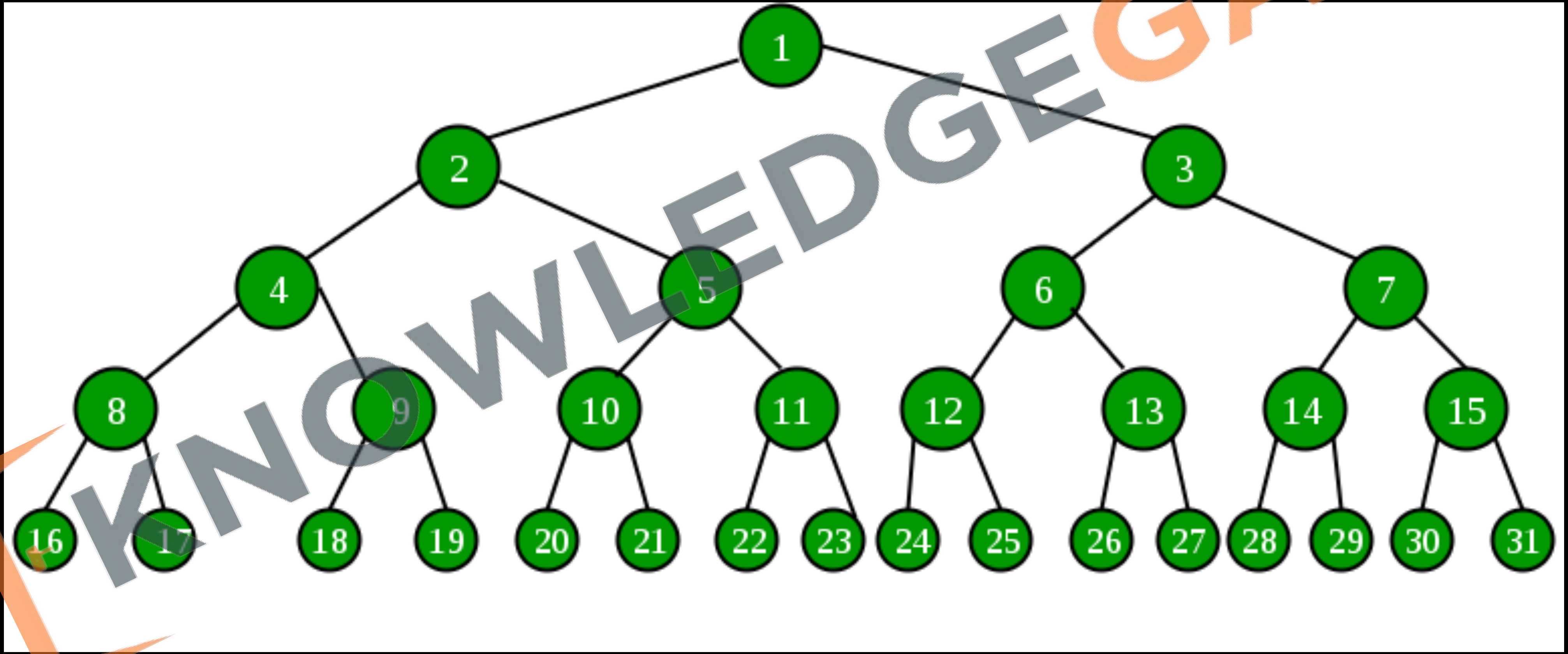
AVL tree

Complete Binary Tree

- Consider a binary tree T, the maximum number of nodes at height h is 2^h nodes.
- The binary tree T is said to be complete binary tree, if all its level except possibly the last, have the maximum number of nodes and if all the nodes at the last level appear as far left as possible.

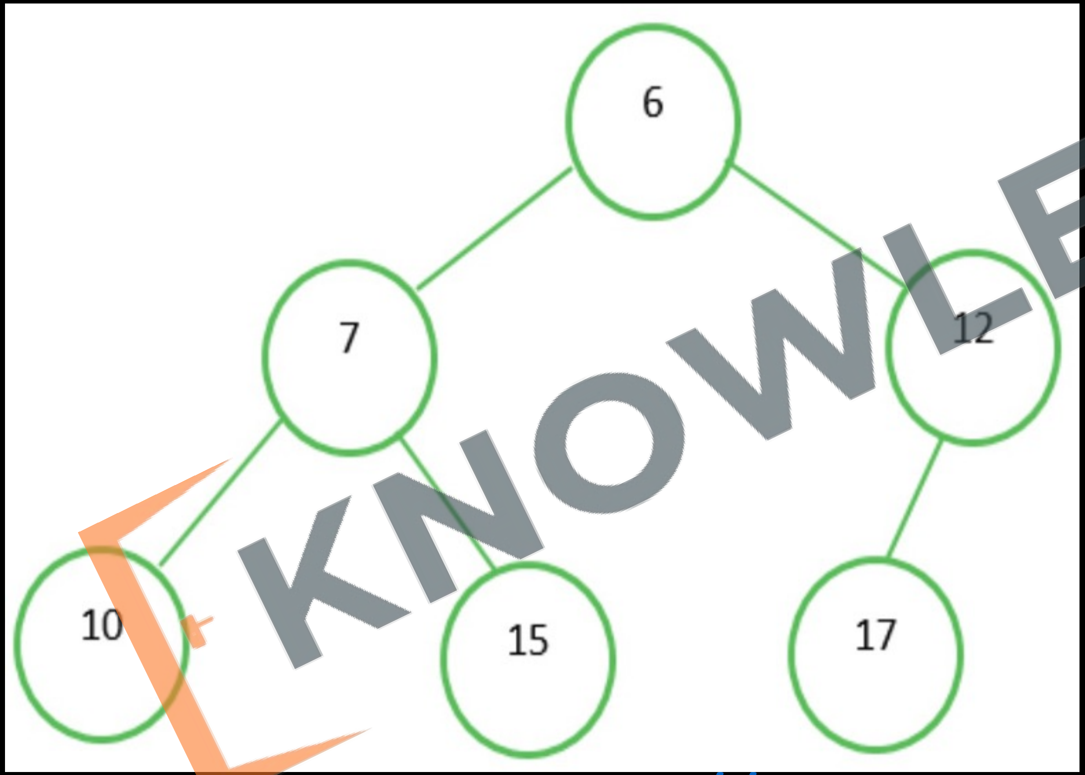


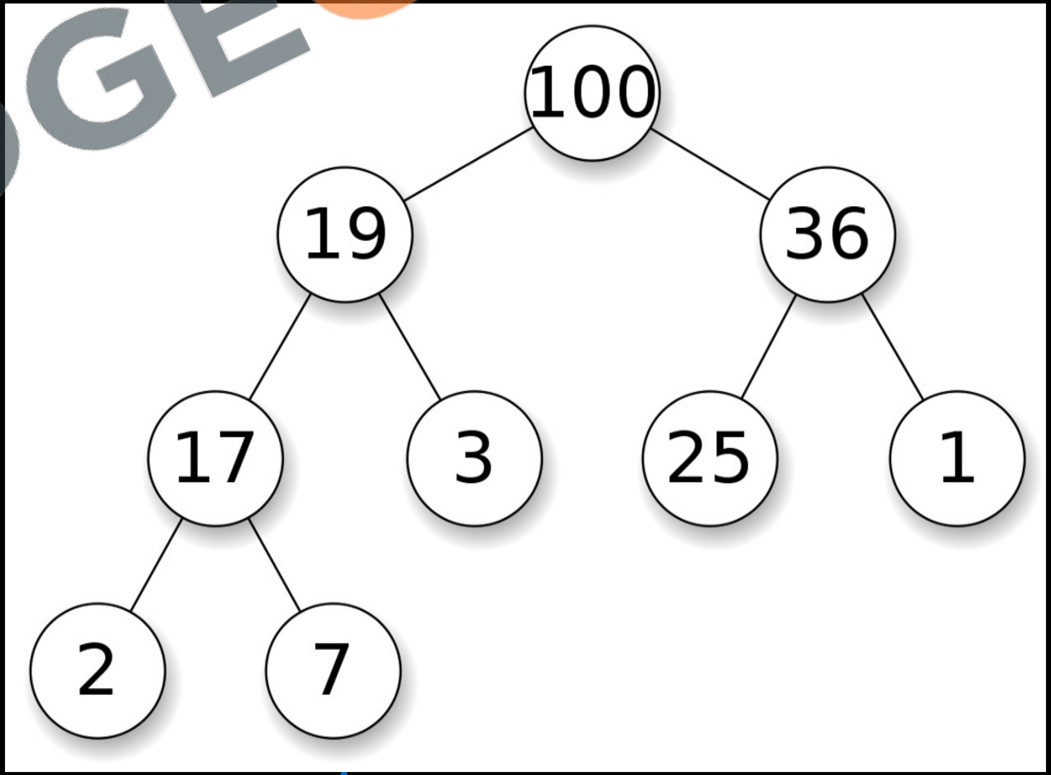
- One can easily determine the children and parent of a node k in any complete tree T
- Specially the left and right children of the node K are 2*k, 2*k + 1 and the parent of k is the node lower bound(k/2)



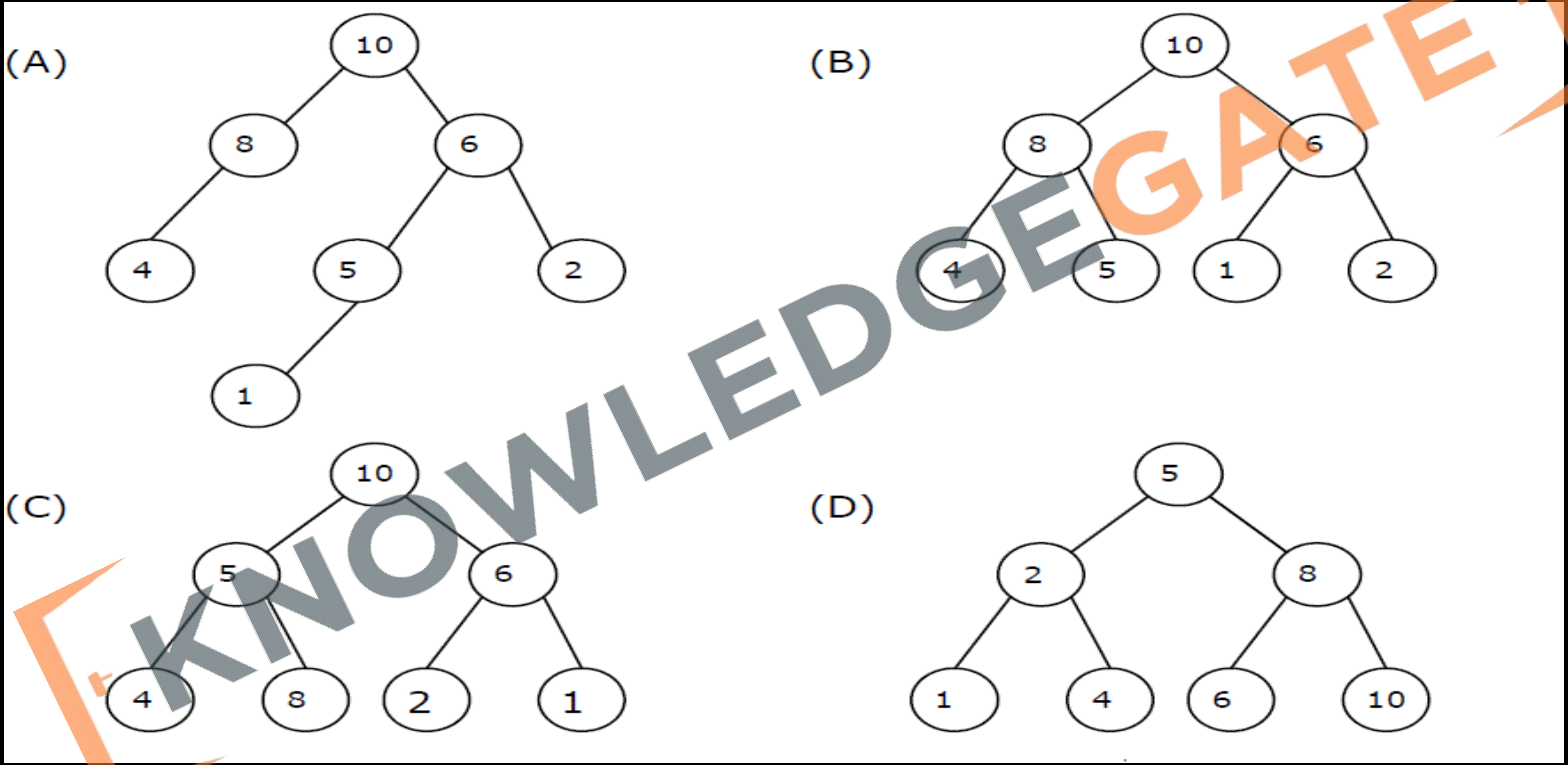
Heap

- Suppose H is a complete binary tree with n elements, H is called a Heap, if each node N of H has following properties:
  - The value of N is greater than to the value at each of the children of N then it is called Max heap.
  - A min heap is defined as the value at N is less than the value at any of the children of N.





Q A max-heap is a heap where the value of each parent is greater than or equal to the values of its children. Which of the following is a max-heap?



Q Consider a binary max-heap implemented using an array. Which one of the following arrays represents a binary max-heap?

(A) 23,17,14,6,13,10,1,12,7,5

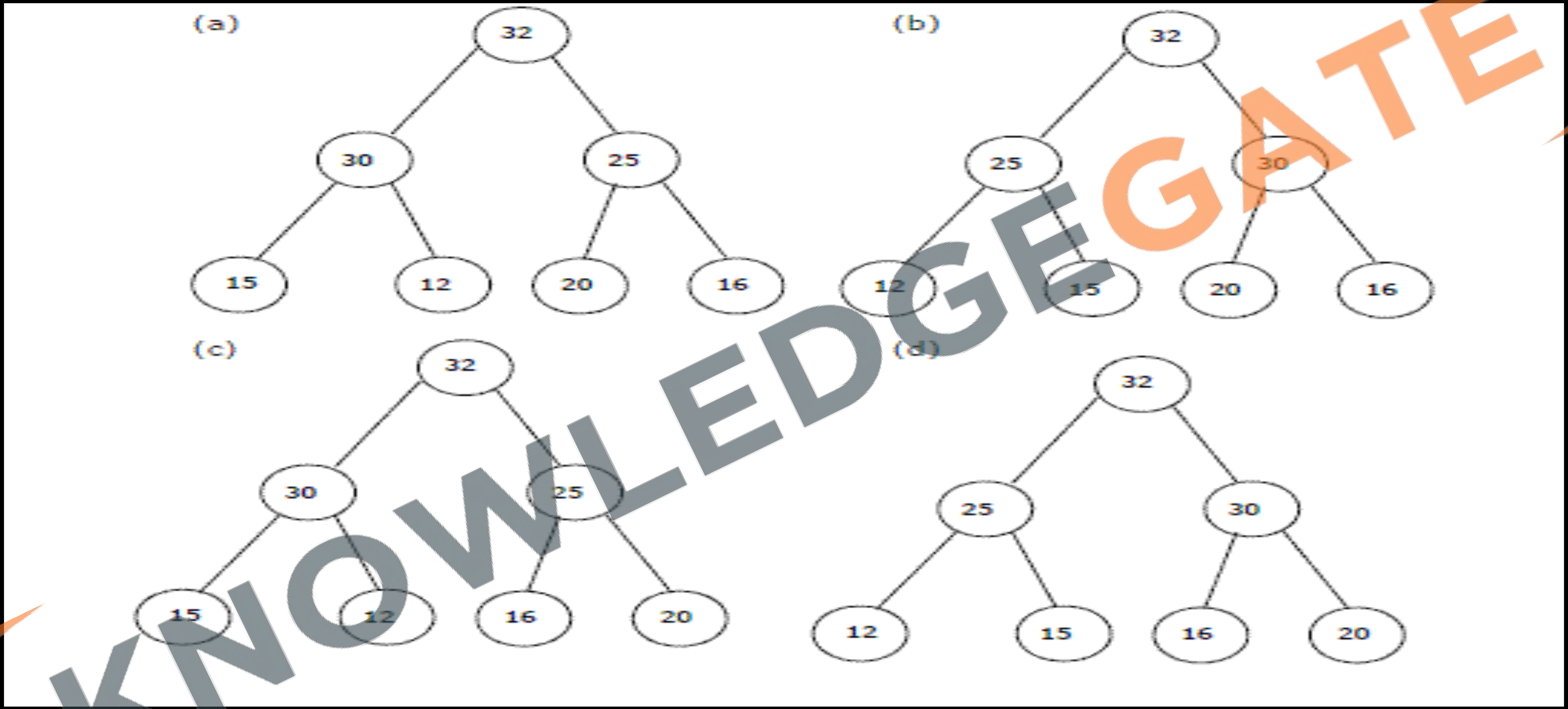
(B) 23,17,14,6,13,10,1,5,7,12

(C) 23,17,14,7,13,10,1,5,6,12

(D) 23,17,14,7,13,10,1,12,5,7

Q The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a Max Heap. The resultant Max Heap is.

Q The elements 32, 15, 20, 30, 12, 25, 16 are inserted one by one in the given order into a Max Heap. The resultant Max Heap is.



http://www.knowledgegate.in/GATE

Q Consider the following array of elements. ⟨89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100⟩. The minimum number of interchanges needed to convert it into a max-heap is

(A) 4 **(B)** 5 **(C)** 2 **(D)** 3

Q A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is: 10, 8, 5, 3, 2. Two new elements 1 and 7 are inserted into the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

(A) 10, 8, 7, 3, 2, 1, 5

(B) 10, 8, 7, 2, 3, 1, 5

(C) 10, 8, 7, 1, 2, 3, 5

(D) 10, 8, 7, 5, 3, 2, 1

Q Consider a binary max-heap implemented using an array. Which one of the following arrays represents a binary max-heap?

(A) 25,12,16,13,10,8,14

(B) 25,12,16,13,10,8,14

(C) 25,14,16,13,10,8,12

(D) 25,14,12,13,10,8,16

Q What is the content of the array after two delete operations on the correct answer to the previous question?

(A) 14,13,12,10,8

(B) 14,12,13,8,10

(C) 14,13,8,12,10

(D) 14,13,12,8,10

Q We are given a set of n distinct elements and an unlabeled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?

(A) 0 **(B)** 1 **(C)** n! **(D)** (1/(n+1)).2nCn

Q The maximum number of binary trees that can be formed with three unlabelled nodes is:

a) 1 **b)** 5 **c)** 4 **d)** 3

Q How many distinct binary search trees can be created out of 4 distinct keys?

(A) 4 **(B)** 14 **(C)** 24 **(D)** 42

Q how many distinct BST can be constructed with 3 distinct keys?

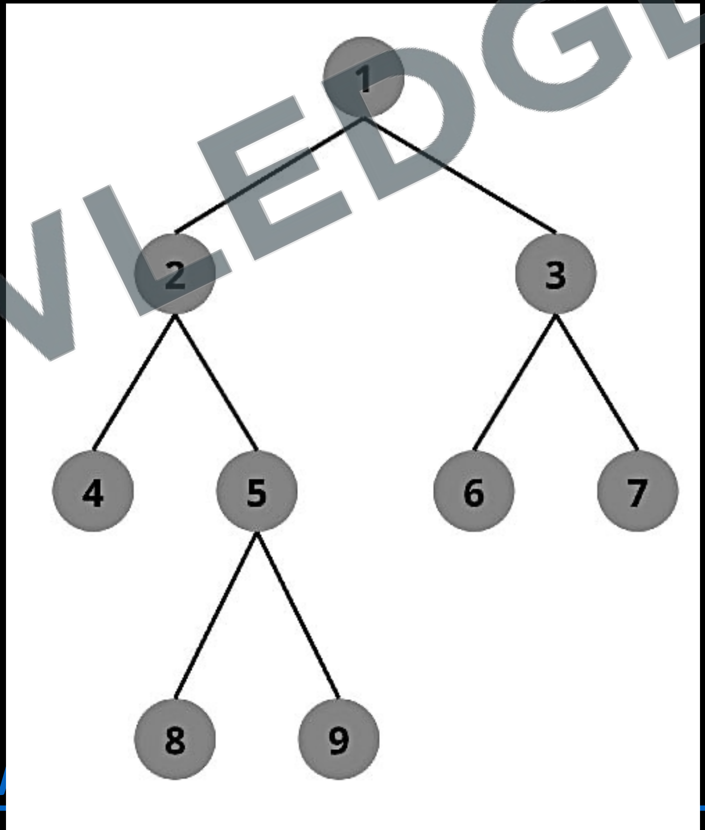
a) 4 **b)** 5 **c)** 6 **d)** 9

Q A complete n-ary tree is one in which every node has 0 or n sons. If x is the number of internal nodes of a complete n-ary tree, the number of leaves in it is given by

a) $x(n-1)+1$ **b)** $xn-1$ **c)** $xn+1$ **d)** $x(n+1)$

Strictly binary tree

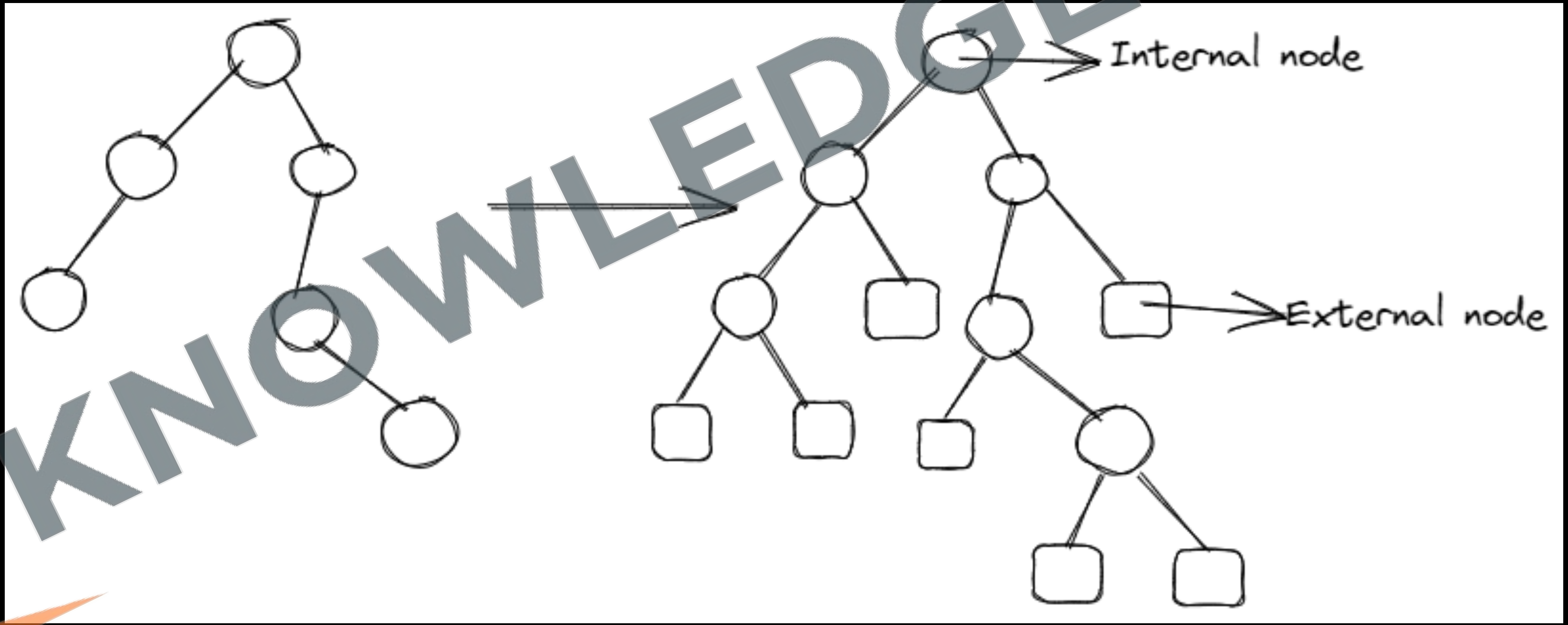
- If every non-leaf node in a binary tree has non-empty left and right subtree, the tree is termed as strictly binary tree.
- b. A strictly binary tree with n leaves always contains 2n - 1 nodes.
- c. If every non-leaf node in a binary tree has exactly two children, the tree is known as strictly binary tree.



http://w in/GATE

Extended binary tree

- A binary tree is said to be 2-tree or extended binary tree if each node has either 0 or 2 children
- Nodes with 2 children are called internal nodes and nodes with 0 children are called external nodes.

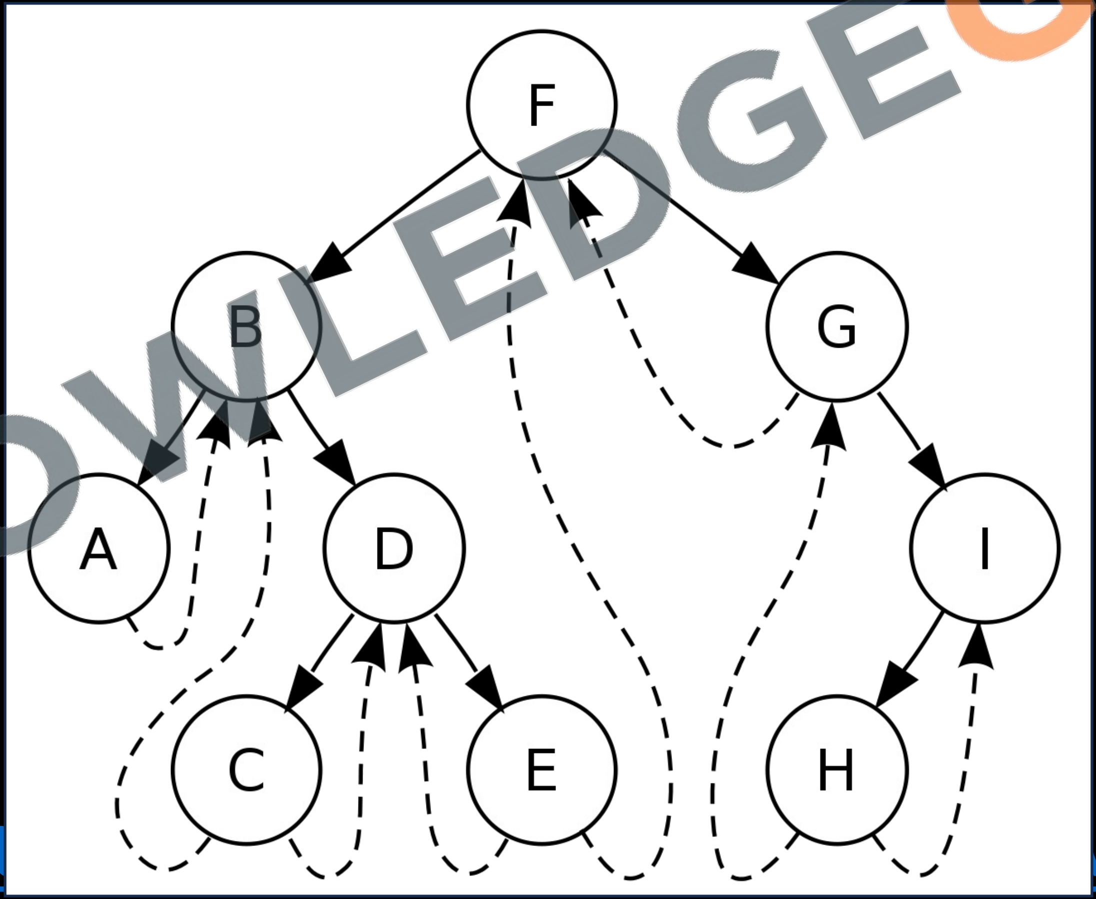


http://www.knowledgegate.in/GATE

Feature/Property	Strictly Binary Tree	Extended (Full) Binary Tree
Definition	Every non-leaf node has exactly two children.	Every node has either 0 or 2 children.
Nodes with One Child	No nodes with only one child.	No nodes with only one child.
Special Nodes	None.	May have external (dummy) nodes to make it full.
Typical Usage	Less common in practical applications.	Used in scenarios like Huffman coding trees where it's beneficial to consider the tree as full.

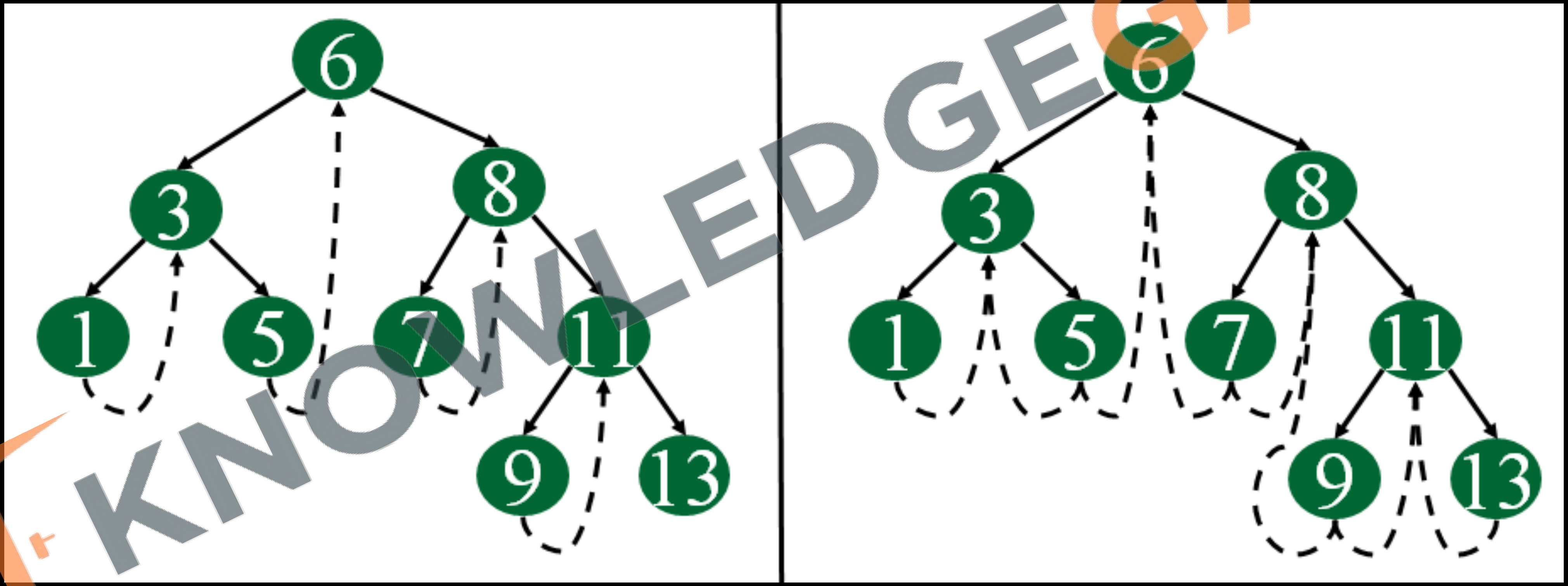
Threaded binary tree

- A threaded binary tree is a modified binary tree that uses null pointers to link to the next node in an in-order sequence, optimizing in-order traversal.
- **Purpose**: Utilizes null pointers to store references (threads) to nodes, aiding efficient in-order traversal without recursion or stacks.



htt ATE

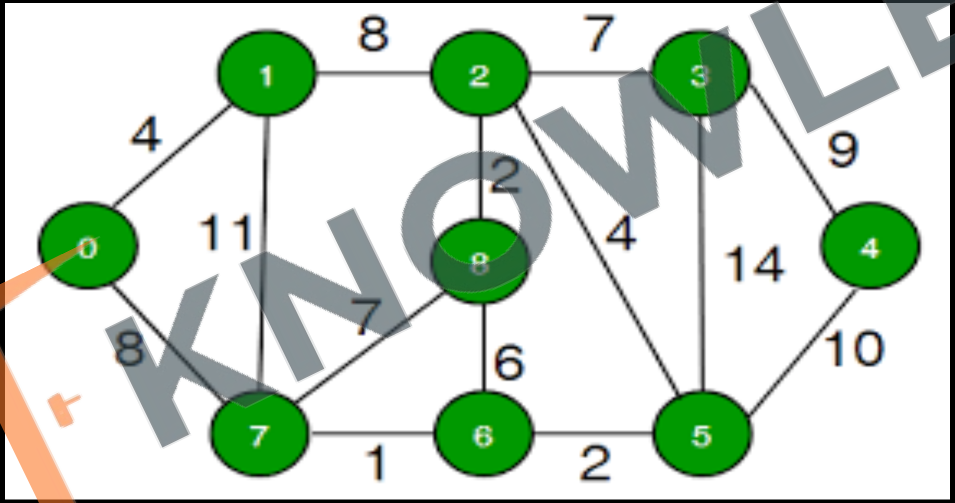
- **Types**:
  - **Single Threaded**: Nodes threaded towards either in-order predecessor or successor.
  - **Double Threaded**: Nodes threaded towards both predecessor and successor.

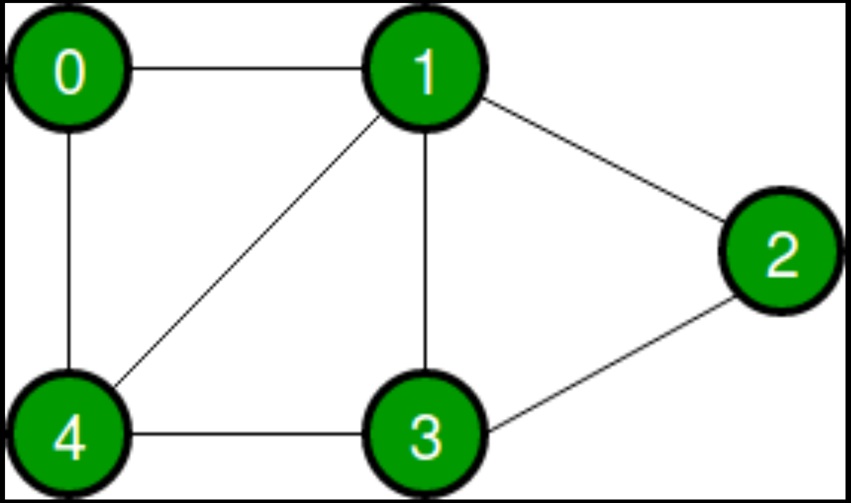


- **Benefits**:
  - Allows stack-less in-order traversal.
  - Makes efficient use of memory by replacing null pointers with threads.
- This tree variant is beneficial when recursive or stack-based traversals aren't feasible. However, its popularity has decreased with newer data structures.

Graph

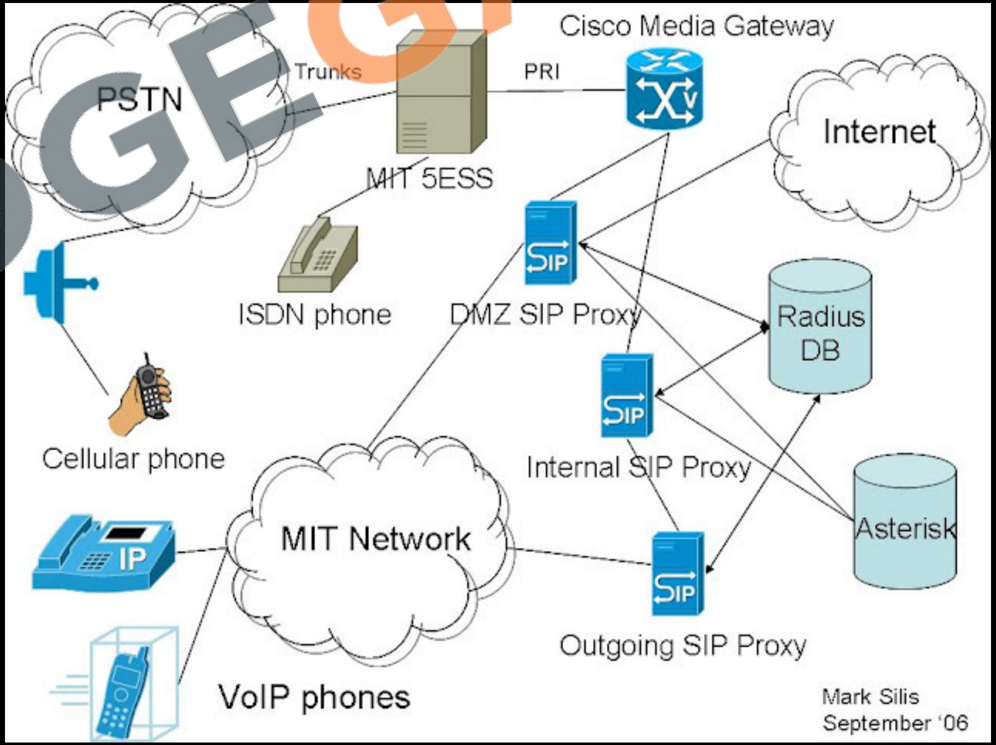
- Graph is a data structure that consists of following two components:
  - A finite set of vertices also called as nodes.
  - A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of a directed graph(di-graph).
  - The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.





http://www.knowledgegate.in/GATE

- Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale.

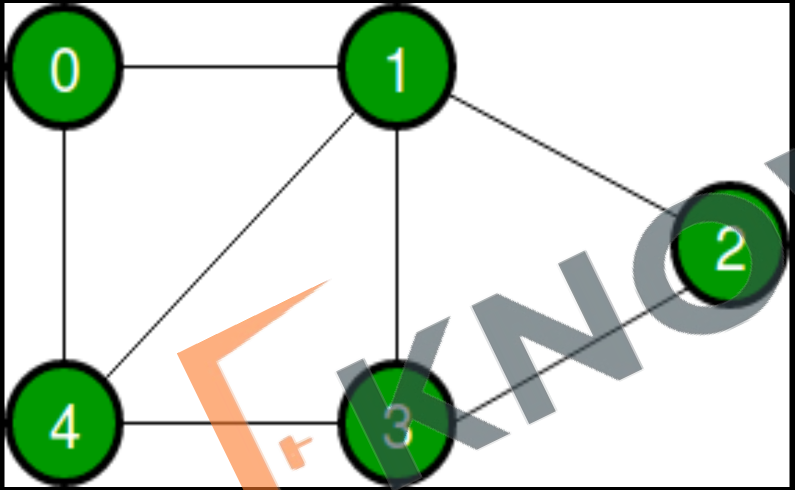


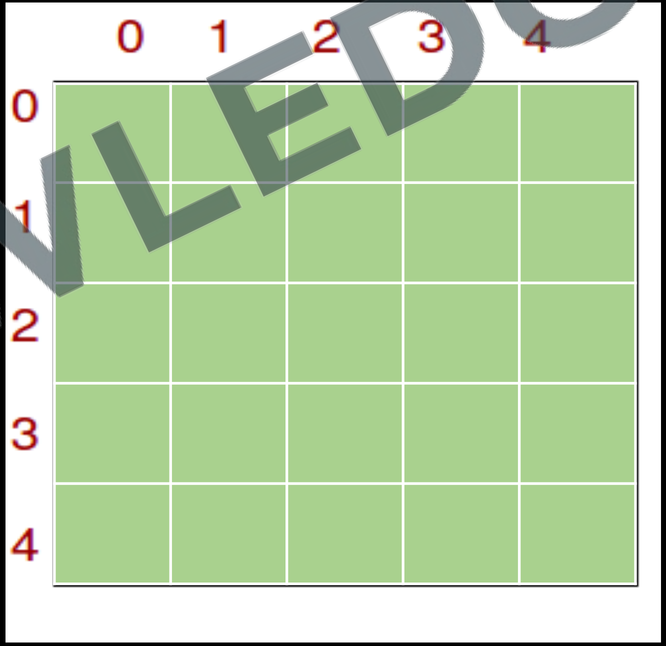
Representation of Graph in Memory

- Following two are the most commonly used representations of a graph.
  - Adjacency Matrix
  - Adjacency List

- There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

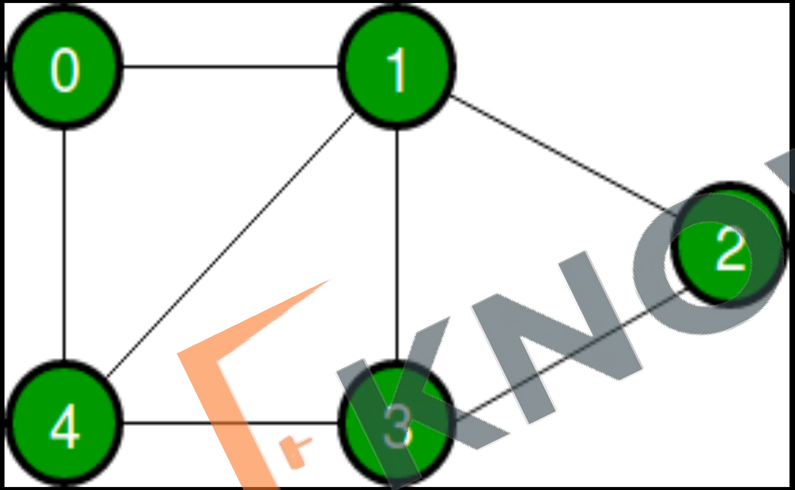
- **Adjacency Matrix:** Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric.
- Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.

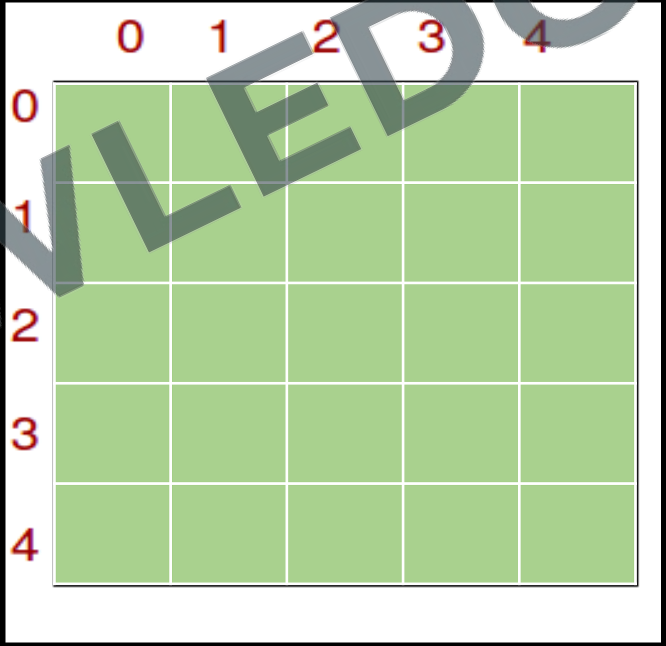




	0	1	2	3	4
0					
1					
2					
3					
4					

- For directed graph





	0	1	2	3	4
0					
1					
2					
3					
4					

Incidence Matrix

- **Representation of undirected graph** : Consider a undirected graph G = (V, E) which has n vertices and m edges all labelled. The incidence matrix I(G) = [bij], is then n x m matrix,
  - where bi,j=1 when edge ej is incident with vi
  - = 0 otherwise
- **Representation of directed graph** : The incidence matrix I(D) = [bij] of digraph D with n vertices and m edges is the n x m matrix in which.
  - Bi,j = 1 if arc j is directed away from vertex vi
  - =－1 if arc j is directed towards vertex vi
  - =0 otherwise.

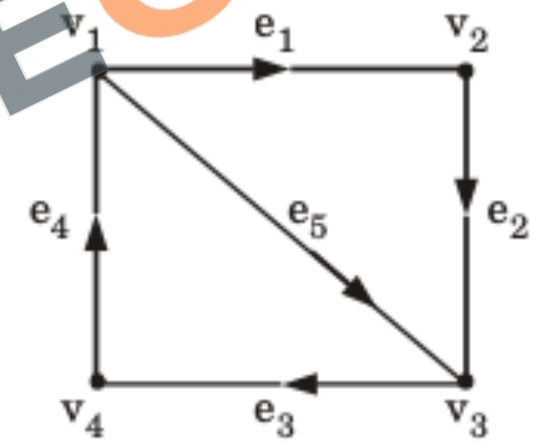


Fig. 4.3.3.

The incidence matrix of the digraph of Fig. 4.3.3 is

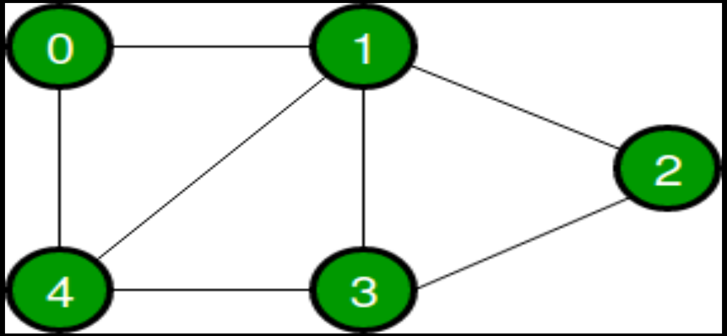
$$I(D) = \begin{bmatrix} 1 & 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix}$$

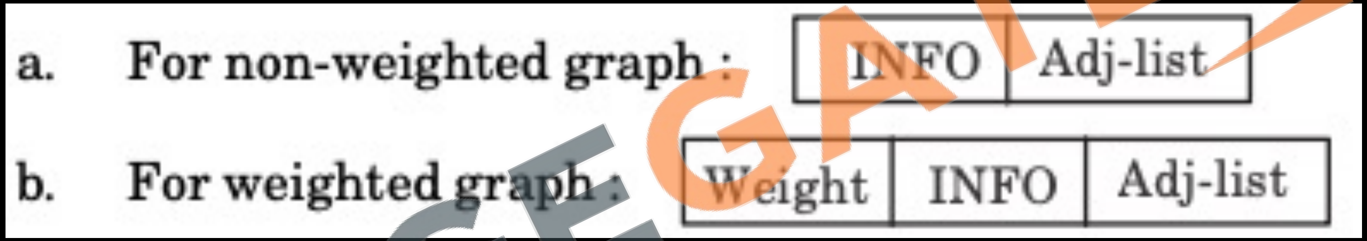
http://www.know

- **Pros:** Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).

- **Cons:** Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

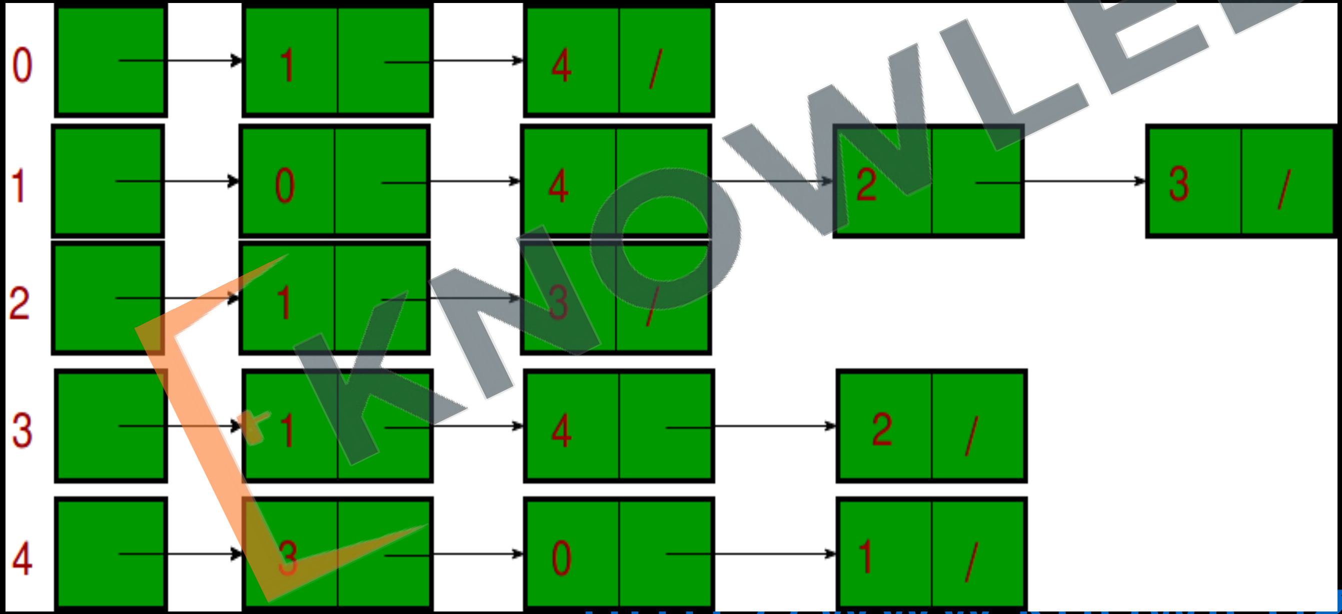
- **Adjacency List:** An array of lists is used. Size of the array is equal to the number of vertices. Let the array be array[]. An entry array[i] represents the list of vertices adjacent to the *i*th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.





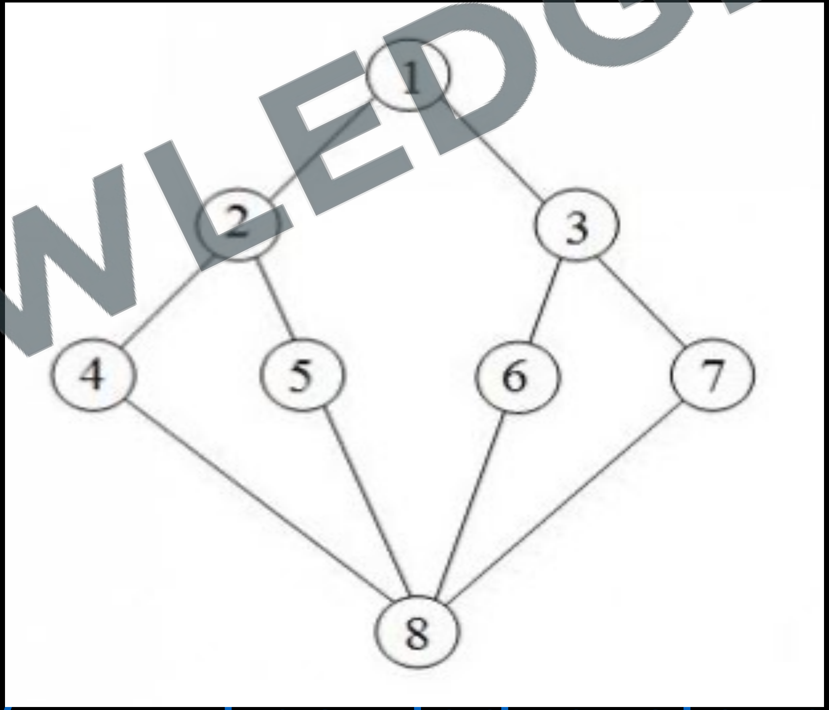
a. For non-weighted graph : | INFO | Adj-list |

b. For weighted graph : | Weight | INFO | Adj-list |



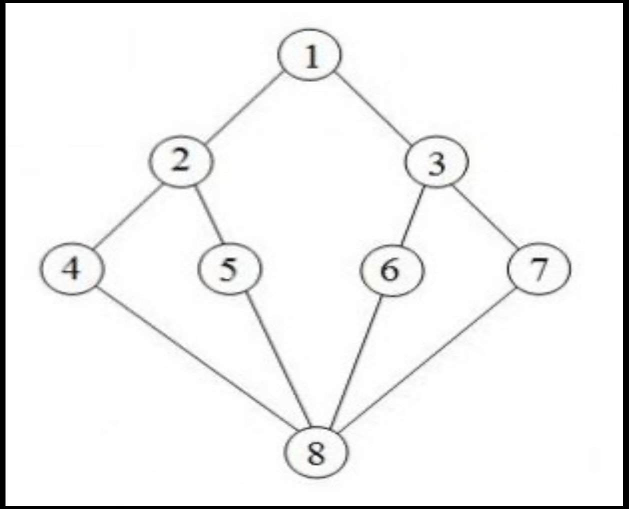
Graph Traversal

- Traversal means visiting all the nodes of a graph.
- Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree.
- The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array.

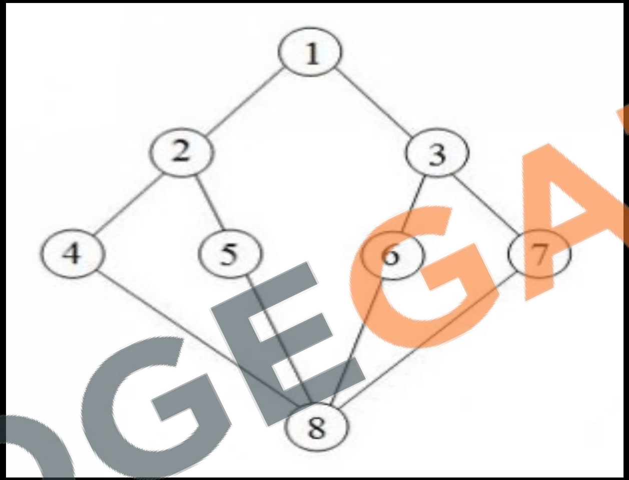


Q Which of the following are valid and invalid DFS traversal sequence

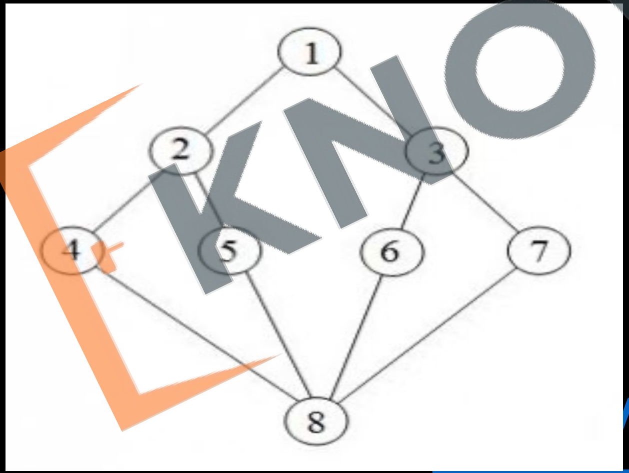
a) 1, 3, 7, 8, 5, 2, 4, 6



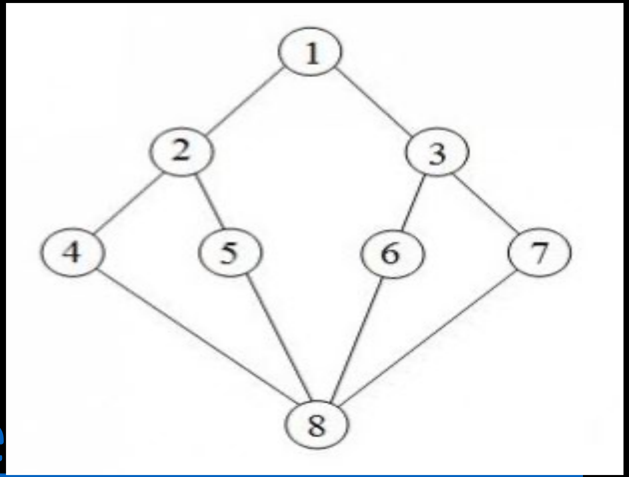
b) 1, 2, 5, 8, 6, 3, 7, 4



c) 1, 3, 6, 7, 8, 5, 2, 4



d) 1, 2, 4, 5, 8, 6, 7, 3

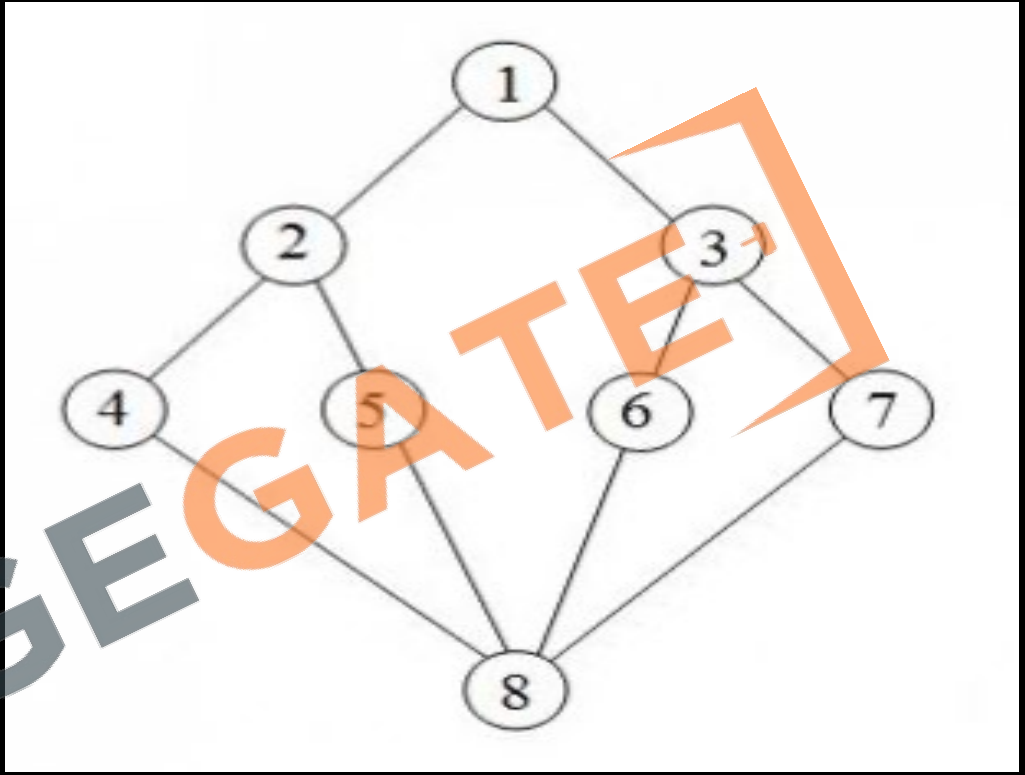


- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited

- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

- The DFS algorithm works as follows:
  - Start by putting any one of the graph's vertices on top of a stack.
  - Take the top item of the stack and add it to the visited list.
  - Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of stack.
  - Keep repeating steps 2 and 3 until the stack is empty.

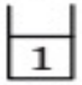
```
DFS(v)
{
    visited(v) = 1
    For all x adjacent to v
    {
        if (x is not visited)
            DFS(x)
    }
}
```



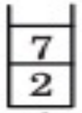
Numerical : Adjacency list of the given graph :

$1 \rightarrow 2, 7$
$2 \rightarrow 3$
$3 \rightarrow 5, 4, 1$
$4 \rightarrow 6$
$5 \rightarrow 4$
$6 \rightarrow 2, 5, 1$
$7 \rightarrow 3, 6$

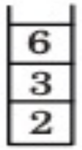
1. Initially set STATUS = 1 for all vertex
2. Push 1 onto stack and set their STATUS = 2



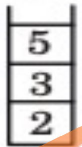
3. Pop 1 from stack, change its STATUS = 1 and
   Push 2, 7 onto stack and change their STATUS = 2;   DFS = 1



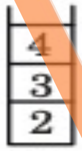
4. Pop 7 from stack, Push 3, 6;   DFS = 1, 7



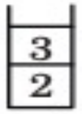
5. Pop 6 from stack, Push 5;   DFS = 1, 7, 6



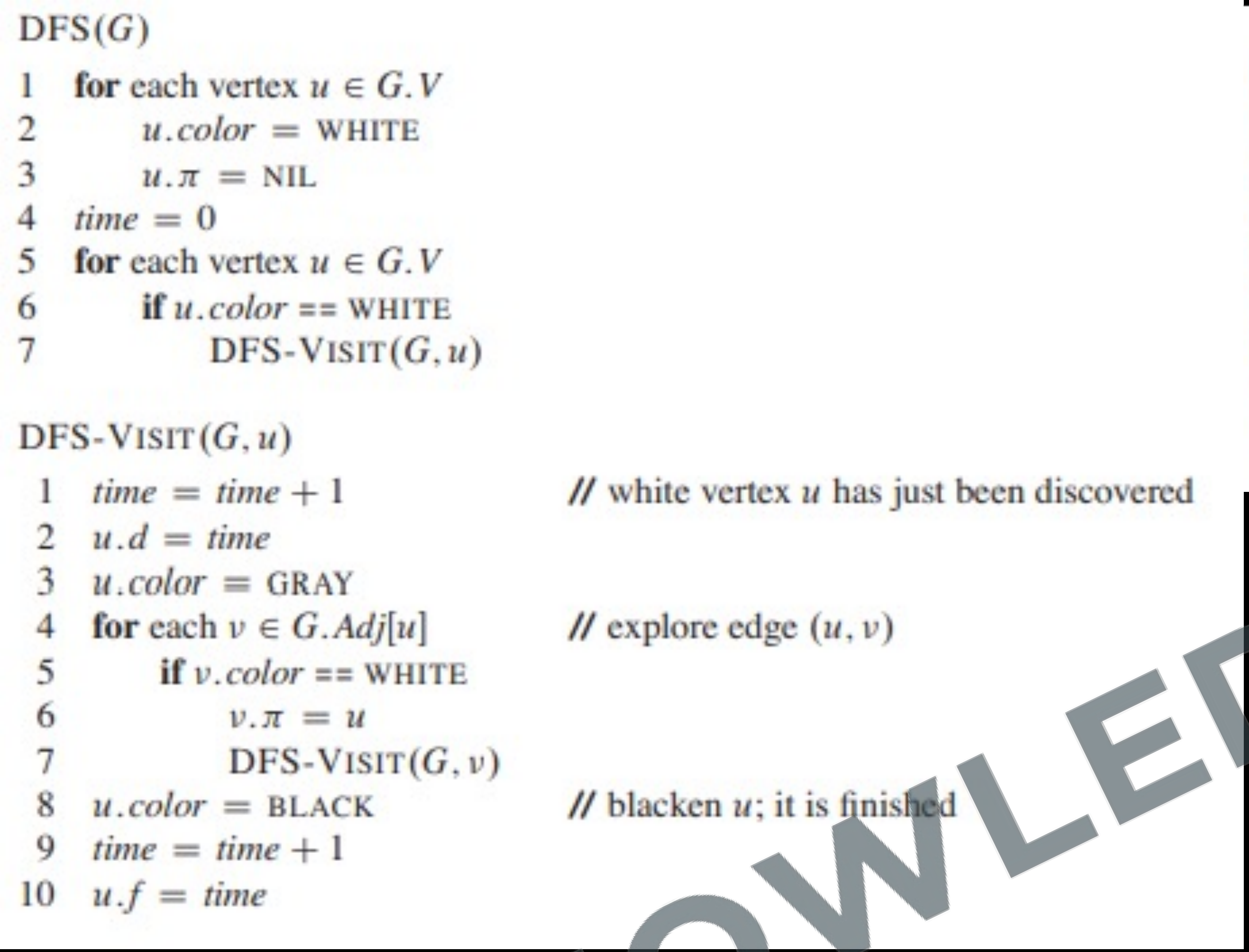
6. Pop 5 from stack, Push 4;   DFS = 1, 7, 6, 5



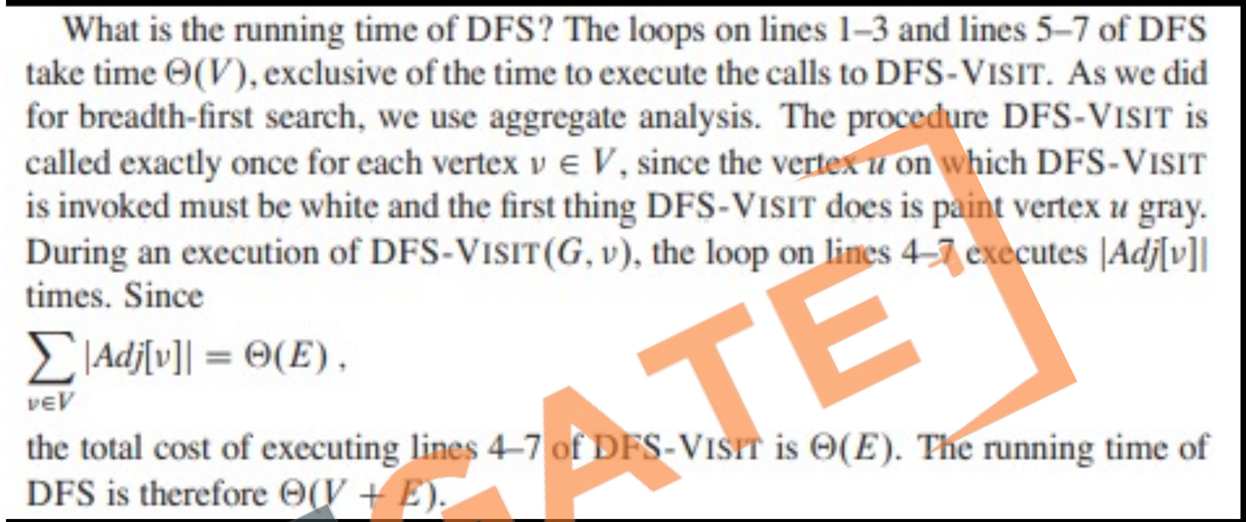
7. Pop 4 from stack;   DFS = 1, 7, 6, 5, 4



- The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time O(|V|+|E|), where |V|is the number of vertices and |E| the number of edges.



```
DFS(G)
1  for each vertex $u \in G.V$
2      $u.color$ = WHITE
3      $u.\pi$ = NIL
4  $time = 0$
5  for each vertex $u \in G.V$
6      if $u.color$ == WHITE
7          DFS-VISIT($G, u$)

DFS-VISIT($G, u$)
 1  $time = time + 1$            // white vertex $u$ has just been discovered
 2  $u.d = time$
 3  $u.color$ = GRAY
 4  for each $v \in G.Adj[u]$    // explore edge $(u, v)$
 5      if $v.color$ == WHITE
 6          $v.\pi = u$
 7          DFS-VISIT($G, v$)
 8  $u.color$ = BLACK            // blacken $u$; it is finished
 9  $time = time + 1$
10  $u.f = time$
```



What is the running time of DFS? The loops on lines 1–3 and lines 5–7 of DFS take time $\Theta(V)$, exclusive of the time to execute the calls to DFS-VISIT. As we did for breadth-first search, we use aggregate analysis. The procedure DFS-VISIT is called exactly once for each vertex $v \in V$, since the vertex u on which DFS-VISIT is invoked must be white and the first thing DFS-VISIT does is paint vertex u gray. During an execution of DFS-VISIT(G, v), the loop on lines 4–7 executes $|Adj[v]|$ times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E) ,$$

the total cost of executing lines 4–7 of DFS-VISIT is $\Theta(E)$. The running time of DFS is therefore $\Theta(V + E)$.

http://www.knowledgegate.in/GATE

```
DFS-iterative (G, s)
{
    let S be stack
    Push( s )
    while ( S is not empty)
    {
        v = pop(S)
        if v is not marked as visited
        {
            mark v as visited
            for all neighbors w of v in Graph G:
            {
                if w is not marked as visited:
                    push( w )
            }
        }
    }
}
```

Importance of DFS : DFS is very important algorithm as based upon DFS :

- Testing whether graph is connected.
- Computing a spanning forest of *G*.
- Computing the connected components of *G*.
- Computing a path between two vertices of *G* or reporting that no such
- path exists.
- Computing a cycle in *G* or reporting that no such cycle exists.

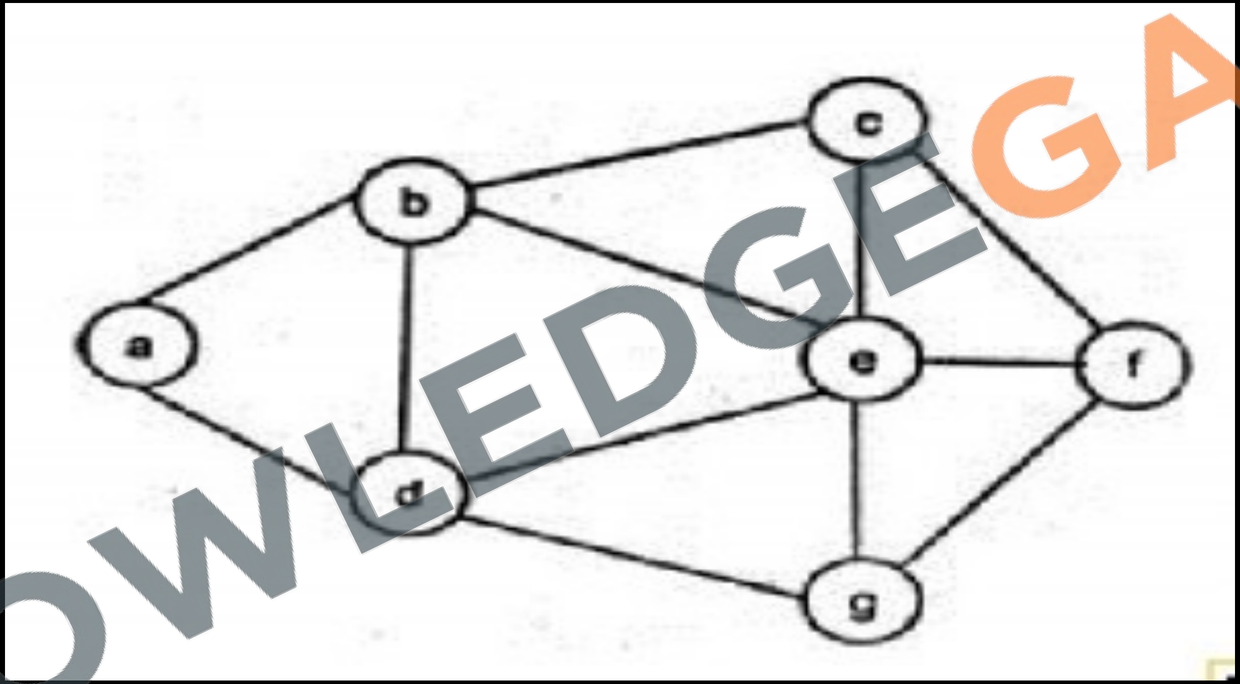
Application of DFS : Algorithms that use depth first search as a building block include :

- Finding connected components.
- Topological sorting.
- Finding 2-(edge or vertex)-connected components.
- Finding 3-(edge or vertex)-connected components.
- Finding the bridges of a graph.
- Generating words in order to plot the limit set of a group.
- Finding strongly connected components.

Q Consider the following sequence of nodes for the undirected graph given below.

1) a b e f d g c **2)** a b e f c g d **3)** a d g e b c f **4)** a d b c g e f

A Depth First Search (DFS) is started at node a. The nodes are listed in the order they are first visited. Which all of the above is (are) possible output(s)?



(A) 1 and 3 only
(B) 2 and 3 only
(C) 2, 3 and 4 only
(D) 1, 2, and 3

http://www.knowledgegate.in/GATE

Q Consider the following graph

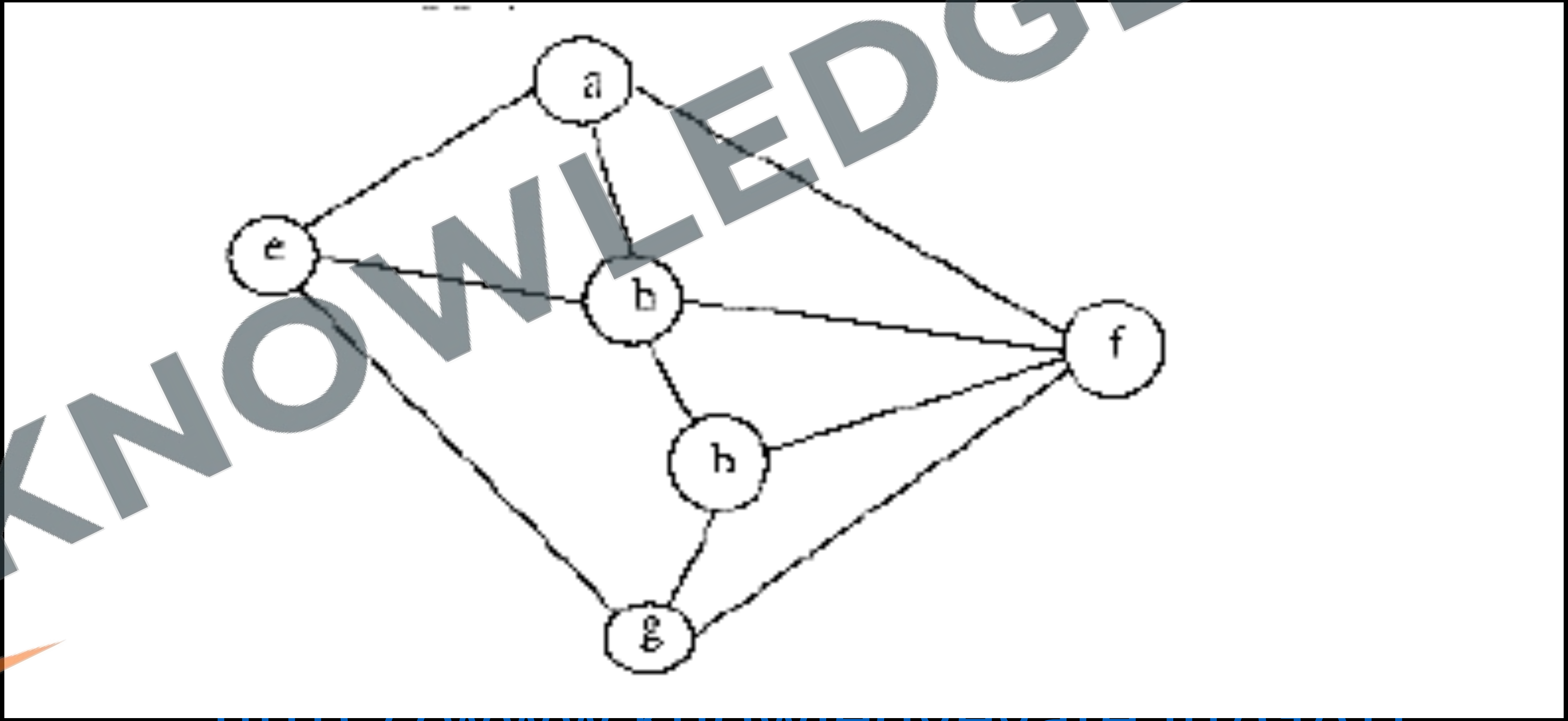
Among the following sequences

I) a b e g h f **II)** a b f e h g **III)** a b f h g e **IV)** a f g h b e

Which are depth first traversals of the above graph?

(A) I, II and IV only **(B)** I and IV only

(C) II, III and IV only **(D)** I, III and IV only



Breadth First Traversal (or Search)

- Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again.
- To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex, i.e. the graph is connected

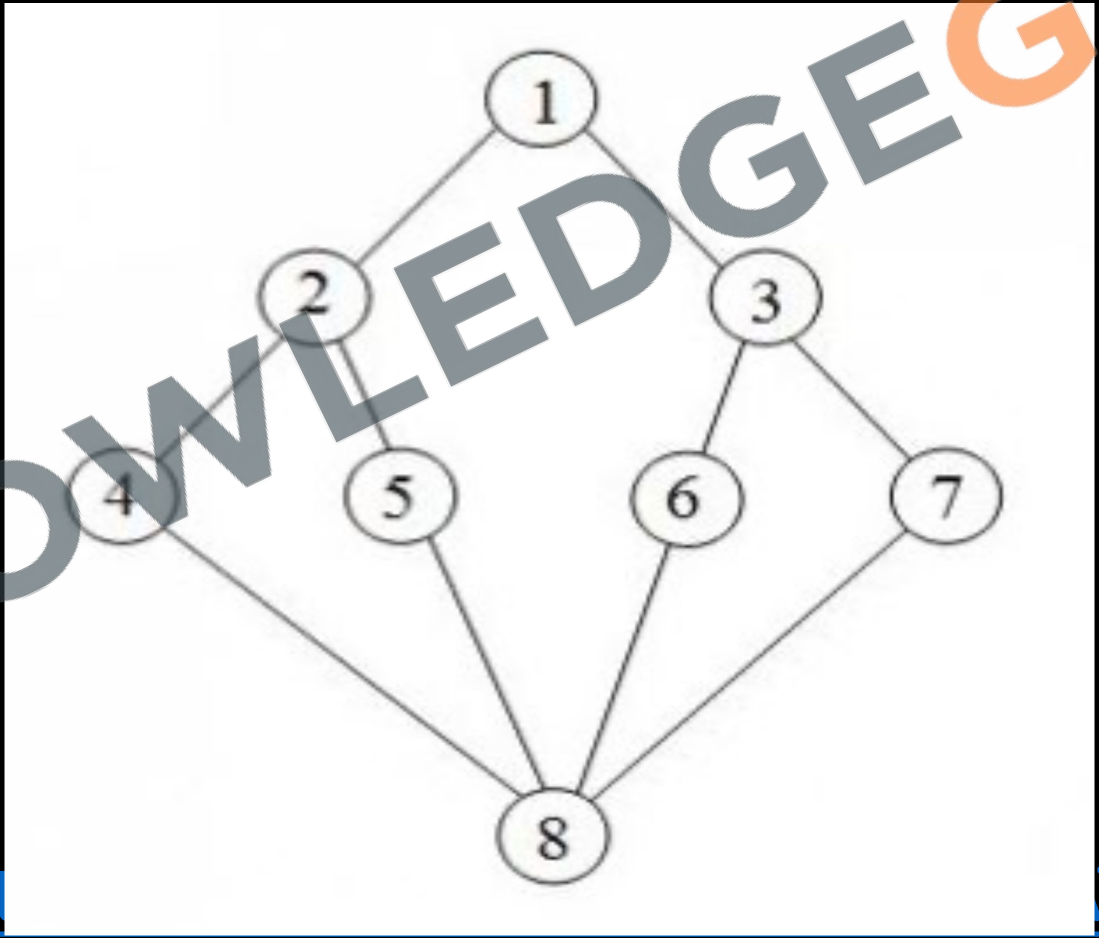
Q Which of the following are valid and invalid BFS traversal sequence

a) 1, 3, 2, 5, 4, 7, 6, 8

b) 1, 3, 2, 7, 6, 4, 5, 8

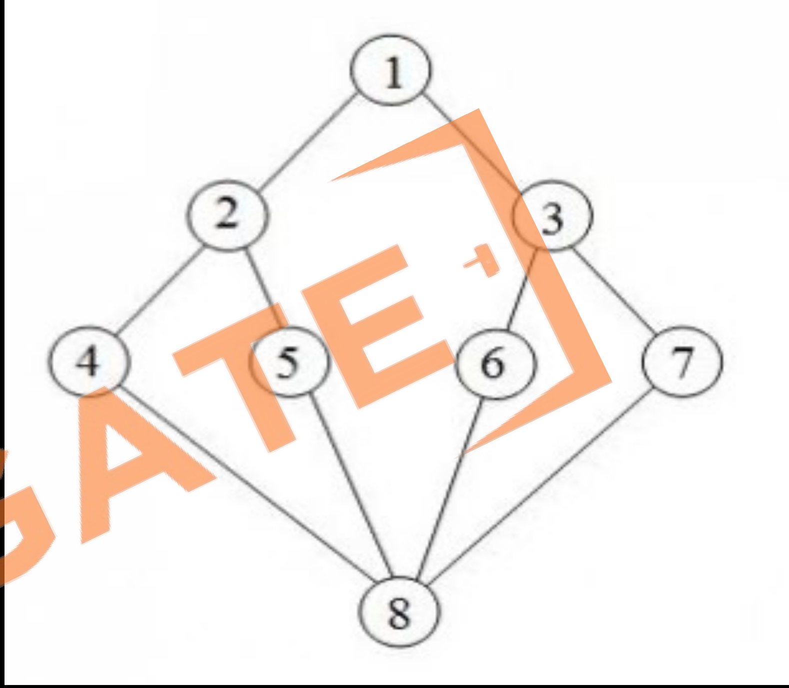
c) 1, 2, 3, 5, 4, 7, 6, 8

d) 1, 2, 3, 7, 5, 6, 4, 8



htt ATE

```
BFS(v)
{
    visited(v) = 1
    insert[V,Q]
    While(Q != Phi)
    {
        u = Delete(Q);
        for all x adjacent to u
        {
            if (x is not visited)
                {
                    visited(x) = 1
                    insert(x,Q)
                }
        }
    }
}
```



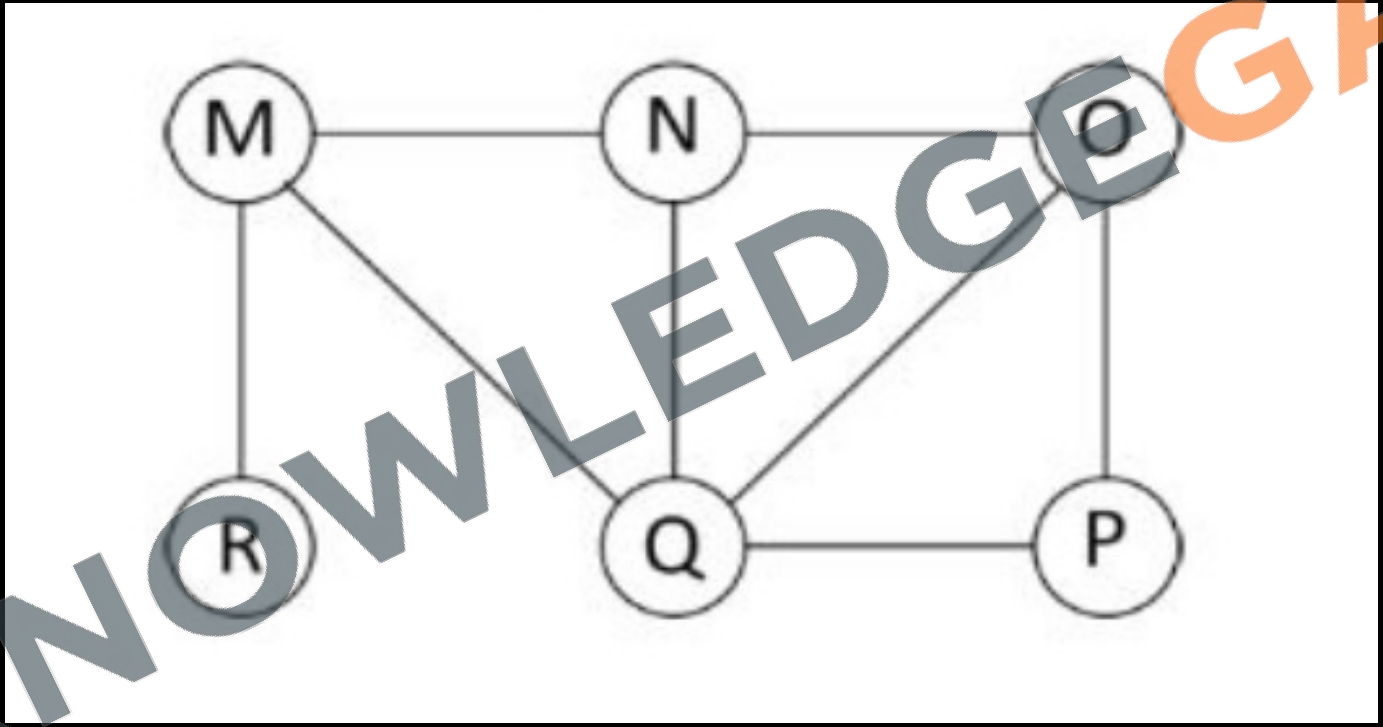
- The time complexity can be expressed as $O(|V|+|E|)$, since every vertex and every edge will be explored in the worst case. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$may vary between $O(1)$ and $O(|V|^2)$.

```
BFS($G, s$)
 1  for each vertex $u \in G.V - \{s\}$
 2      $u.color$ = WHITE
 3      $u.d = \infty$
 4      $u.\pi$ = NIL
 5  $s.color$ = GRAY
 6  $s.d = 0$
 7  $s.\pi$ = NIL
 8  $Q = \emptyset$
 9  ENQUEUE($Q, s$)
10  while $Q \neq \emptyset$
11      $u$ = DEQUEUE($Q$)
12      for each $v \in G.Adj[u]$
13          if $v.color$ == WHITE
14              $v.color$ = GRAY
15              $v.d = u.d + 1$
16              $v.\pi = u$
17              ENQUEUE($Q, v$)
18      $u.color$ = BLACK
```

Before proving the various properties of breadth-first search, we take on the somewhat easier job of analyzing its running time on an input graph $G = (V, E)$. We use aggregate analysis, as we saw in Section 17.1. After initialization, breadth-first search never whitens a vertex, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(V)$. Because the procedure scans the adjacency list of each vertex only when the vertex is dequeued, it scans each adjacency list at most once. Since the sum of the lengths of all the adjacency lists is $\Theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of the BFS procedure is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

Q Breath First Search (BFS) has been implemented using queue data structure. Which one of the following is a possible order of visiting the nodes in the graph above?

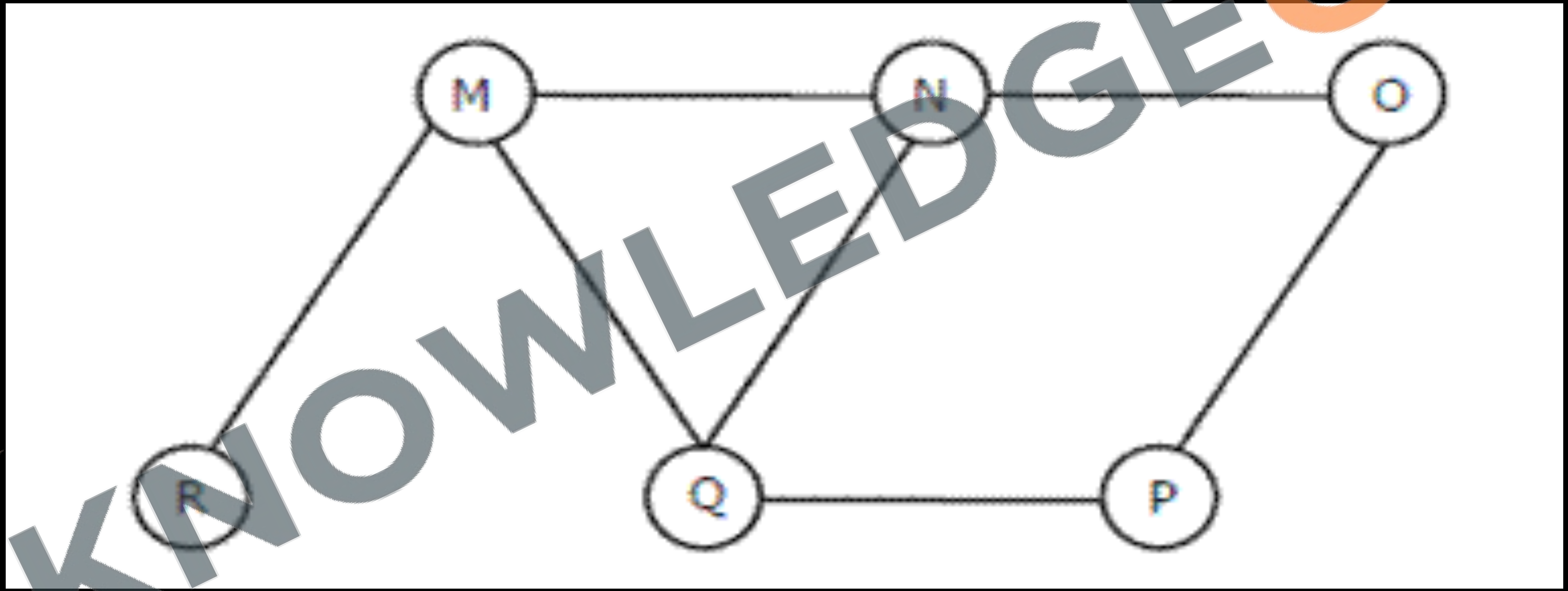
a) MNOPQR **b)** NQMPOR **c)** QMNROP **d)** POQNMR



http://www.knowledgegate.in/GATE

Q The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is

(A) MNOPQR **(B)** NQMPOR **(C)** QMNPRO **(D)** QMNPOR



http://www.knowledgegate.in/GATE

Importance of BFS :

- It is one of the single source shortest path algorithms, so it is used to
- compute the shortest path.
- It is also used to solve puzzles such as the Rubik's Cube.
- BFS is not only the quickest way of solving the Rubik's Cube, but also
- the most optimal way of solving it.

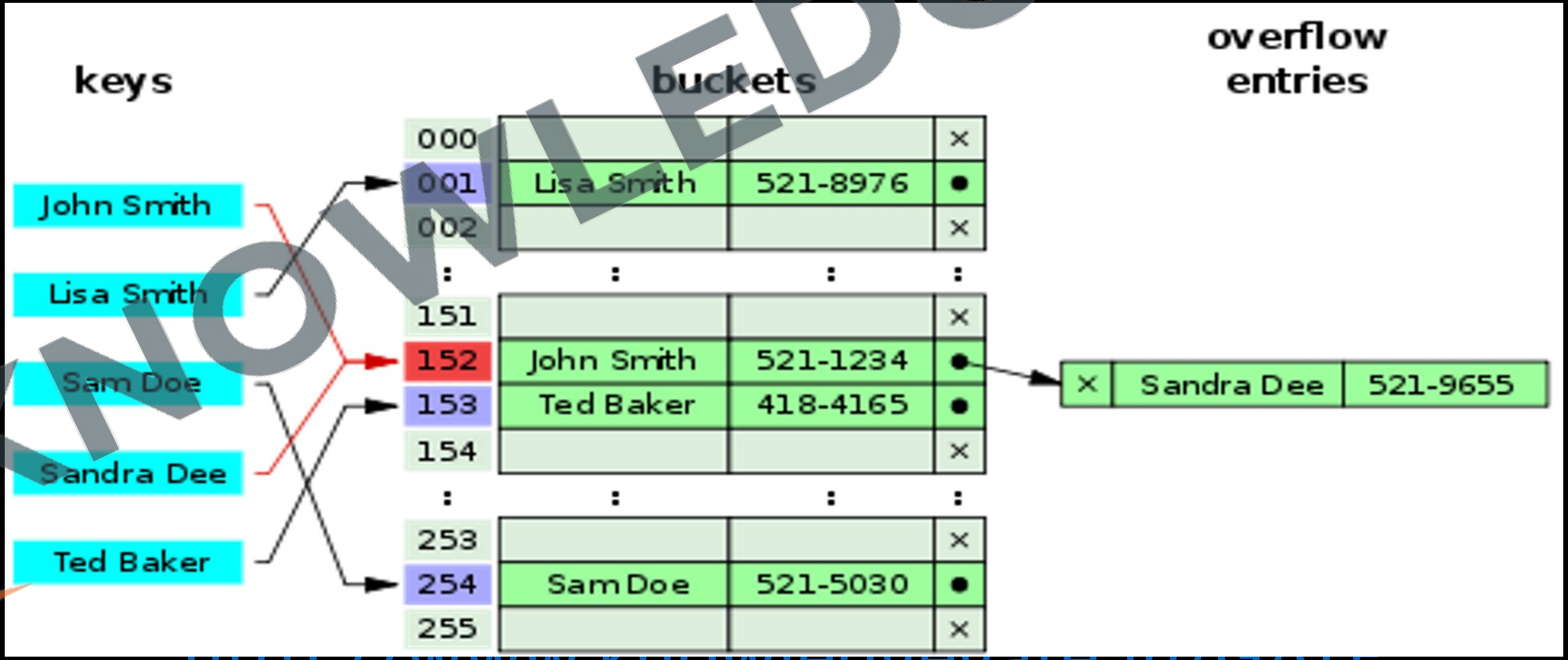
Application of BFS : Breadth first search can be used to solve many problems in graph theory, for example

- Copying garbage collection.
- Finding the shortest path between two nodes *u* and *v*, with path length measured by number of edges (an advantage over depth first search).
- Ford-Fulkerson method for computing the maximum flow in a flow network.
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.
- Construction of the failure function of the Aho-Corasick pattern matcher.
- Testing bipartiteness of a graph.

Introduction to hashing

- Main idea of data structure is to help us store the data. But Most common operation on any data structure is not insert or delete but actually search, as even for insertion and deletion search is also required.
- In any of the data structure the search time first depends on the number of elements which data structure contains and then on type of structure. for e.g.
    - Unsorted array – O(n)
    - sorted array – O(logn)
    - link list – O(n)
    - BT – O(n)
    - BST – O(n)
    - AVL – O(logn)

- So hashing is a technique where search time is independent of the number of items in which we are searching a data value.
- The basic idea is to use the key itself to find the address in the memory to make searching easy. For e.g. *to use phone number, roll no, Aadhar card, voter id or any other key and convert it into a smaller* practical *number (*but it must be modified so a great deal of space is not wasted*) and uses the small number as index in a table called hash table.*
- The values are then stored in **hash table**, By using that key you can access the element in **O(1)** time.



- This conversion called hash function which is from the set of K keys into the set of memory location L.
  - H: K→L
- In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table. An array that stores pointers to records corresponding to our search key. The remaining entries can be nil.

Q Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function x mod 10, which of the following statements are true?

i. 9679, 1989, 4199 hash to the same value
ii. 1471, 6171 has to the same value
iii. All elements hash to the same value
iv. Each element hashes to a different value

(A) i only

(B) ii only

(C) i and ii only

(D) iii or iv

- **Collision**: - It is possible that two different set of keys K_1 and K_2 will yield the same hash address. This situation is called collision. The technique to resolve collision is called collision resolution.

- Characteristics of good hash function
  - Easy to compute and understand
  - Efficiently computable- It must take less time to compute
  - Should uniformly distribute the keys (Each table position equally likely for each key) and should not result in clustering.
  - Must have low collision rate

Most popular hash function

- **Division-remainder method**: The size of the number of items in the table is estimated. That number is then used as a divisor into each original value or key to extract a quotient and a remainder.
- The remainder is the hashed value. (Since this method is liable to produce a number of collisions, any search mechanism would have to be able to recognize a collision and offer an alternate search mechanism.)
  - $H(K) = K(\text{mod } m)$
  - $H(K) = K(\text{mod } m) + 1$

- **Note**: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Q Which of the following statement(s) is TRUE?

I. A hash function takes a message of arbitrary length and generates a fixed length code.

II. A hash function takes a message of fixed length and generates a code of variable length.

III. A hash function may give the same hash value for distinct messages.

(a) I only **(b)** II and III only **(c)** I and III only **(d)** I only

Mid-Square Method

- The mid-square method is a technique used to generate hash codes by squaring the key and then extracting a portion of the resulting number. This method was popular for hash function design in early hashing techniques but has been superseded by more robust methods in modern systems.
- **Square the Key**: Take the key, square it (e.g., key 123 gives 15129).
- **Middle Extraction**: Extract middle digits from the squared result (e.g., from 15129, take 512).
- **Fit to Table**: Optionally, use modulus to fit the hash within table size.

- **Pros**:
  - Simple to use.
  - Effective for random, uniform keys.

- **Cons**:
  - Potential for collisions.
  - Hash quality varies with key distribution.

Folding Method

- The folding method is a technique used in hashing to partition the key into several parts, then combine these parts to determine the hash code.
- Here's how the folding method works:
  - **Partition the Key**: Divide the key into equal-sized parts. For example, for a key 123456789 and partition size of 3, you'd have 123, 456, and 789.
  - **Add the Partitions**: Sum these parts together. Continuing the example, 123 + 456 + 789 = 1368.
  - **Modulus Operation**: If the resulting sum is larger than the hash table size, a modulus operation will bring it within range. For instance, if the hash table has 1000 slots, 1368 % 1000 = 368 would be the final hash code.

- **Advantages**:
  - It distributes keys that are close in value across the hash table.
  - Simple and intuitive.

- **Disadvantages**:
  - Not as efficient for keys with certain patterns.
  - Might still lead to collisions if the table size isn't chosen wisely.
  - Like other simple hashing techniques, the folding method's usage has been largely superseded by more advanced hash functions in modern systems. However, it remains a basic technique useful for understanding foundational hashing concepts.

Collision Resolution Technique

- **Open Addressing/closed hashing** - In Open Addressing, all elements are stored in the hash table itself. i.e. collision is resolved by **probing** or searching through alternate locations in the Hash table itself in a particular *sequence.*
- When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table. So, at any point, size of table must be greater than or equal to total number of keys.
- It is of three types linear probing, quadratic probing, double hashing

Q Consider a hash table of size seven, with starting index zero, and a hash function (7x+3) mod 4. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Here "__" denotes an empty location in the table.

(a) 3, 10, 1, 8, __ , __ , __

(b) 1, 3, 8, 10, __ , __ , __

(c) 1, __ , 3, __ , 8, __ , 10

(d) 3, 10, __ , __ , 8, __ , _

Linear probing

- Linear probing is a method used in open addressing hashing. When a collision occurs, it searches the table sequentially from the hashed position to find an empty or matching slot.

- **Key Points**:
  - Uses a random hash function, ensuring constant expected time for operations.
  - Achieves O(1) time for insert, remove, and search if the load factor is kept below one.

- **Insert(k)**: Probe until an empty slot is found, then insert k.
- **Search(k)**: Probe until a matching key or empty slot is found.
- **Delete(k)**: Mark slots of deleted keys as "deleted". Inserts can use these slots, but searches don't stop at them.

Linear Probing

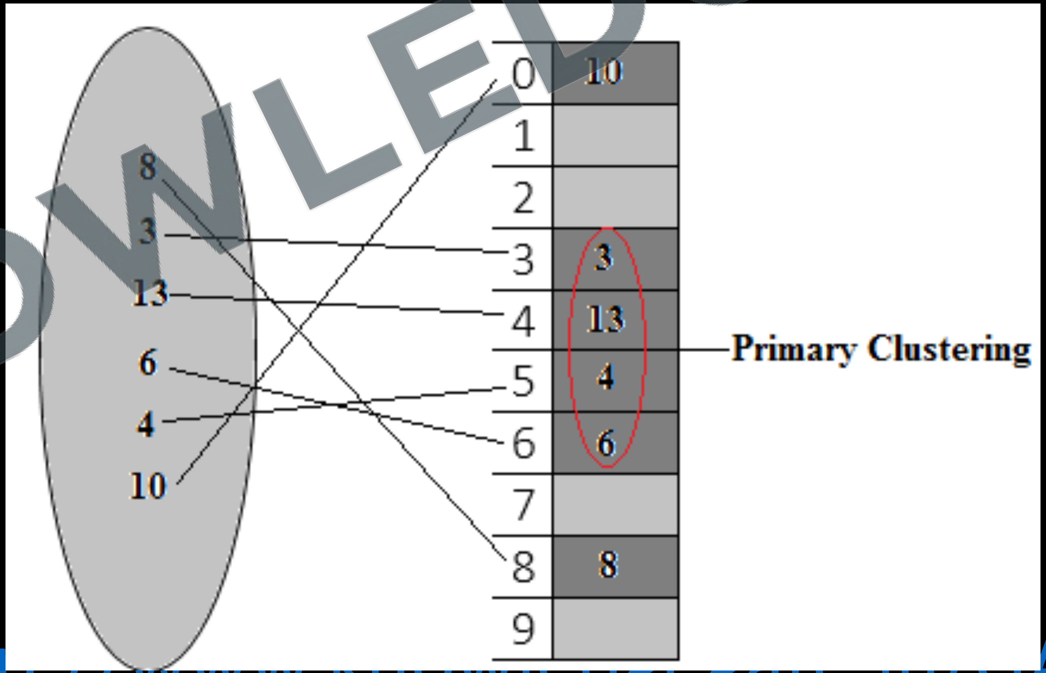
- In linear probing method, in case of a collision we find out the next free space and store the key that is causing collision in it.
- The method of *linear probing* uses the hash function

  $h(k, i) = (h'(k) + i) \bmod m;$

  for $i = 0, 1, \ldots, m - 1$.

Example: Let us take the previous example, where the key value 13 was causing the collision at location 3.

- h (13) = (h(13) + 0) mod 10 = 3, since it is causing collision we consider the next value of i, i.e. i =1.
- h (13) = (h(13) + 1) mod 10 = 4, now at this location there is no collision so we place the value 13 at location 4.



Q A hash table contains 10 buckets and uses linear probing to resolve collision. The key values are integers and the hash function used is key%10, if the values 43 165 62 123 142 are inserted in the table, in what location would the key value 142 be inserted?

A) 2 **b)** 3 **c)** 4 **d)** 6

Q Consider a hash table of size seven, with starting index zero, and a hash function (3x + 4)mod7. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing? Note that '_' denotes an empty location in the table.

(A) 8, _, _, _, _, _, 10

(B) 1, 8, 10, _, _, _, 3

(C) 1, _, _, _, _, _,3

(D) 1, 10, 8, _, _, _, 3

- **Advantage**:
  - Linear probing is fast, simple, and easy to implement, making it a popular choice on standard hardware.
  - It offers high performance due to its excellent locality of reference.

- **Disadvantage**:
  - It's sensitive to the quality of its hash function compared to other schemes.
  - Performance drops faster at high load factors due to primary clustering, leading to more nearby collisions and longer operation times.
  - Requires a superior hash function for optimal performance than some other methods.

- **Primary Clustering**: In open-addressing hash tables, especially with linear probing, collisions result in records being placed in the next available hash table cell. This creates a contiguous cluster of occupied cells. When another record hashes to any part of this cluster, the cluster size increases by one.

- **Secondary Clustering**: This occurs in open addressing modes, including linear and quadratic probing, where the probe sequence doesn't depend on the key. A subpar hash function can make many keys hash to the same spot. Consequently, these keys either follow the same probe sequence or land in the same hash chain, leading to slower access times.

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.

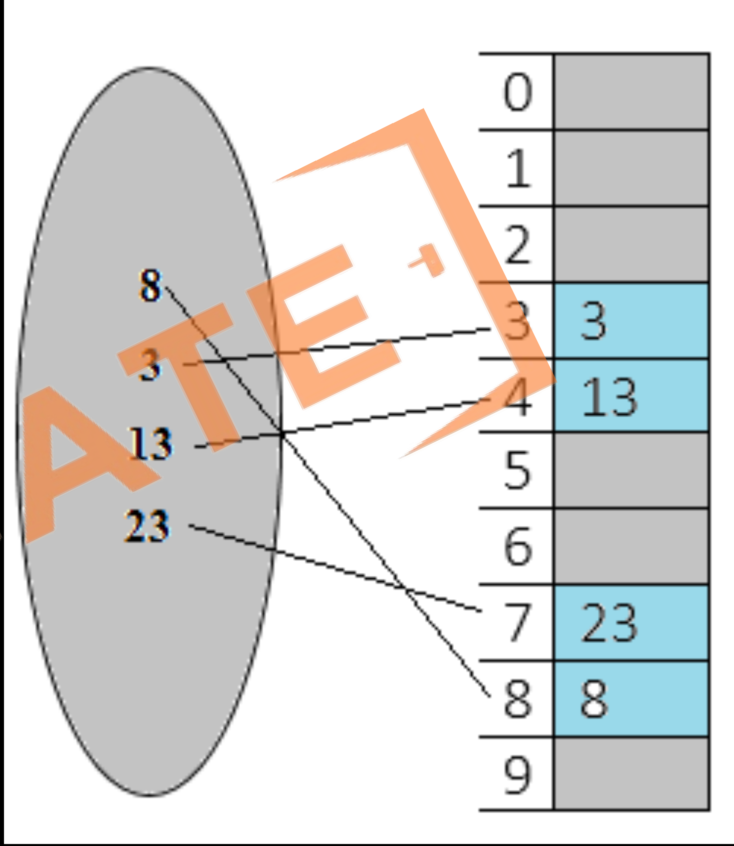
- ***Quadratic probing*** uses a hash function of the form

$$h(k, i) = (h'(k) + f(i^2)) \bmod m$$

Where, h' is an auxiliary hash function and $i = 0, 1, \ldots, m - 1$.

Example: Consider the key values 8, 3, 13, 23 and the hash table size is 10.

- 8 will be placed at: h (8) = [h (8) + f (0^2)] mod 10 = 8, so it gets placed at location 8.
- 3 will be placed at: h (3) = [h (3) + f (0^2)] mod 10 = 3, no collision, so it gets placed at location 3
- 13 will be placed at: h (13) = [h (13) + f (0^2)] mod 10 = 3, collision occurred, so we increase the value of i.
  - h (13) = [h (13) + f (1^2)] mod 10 = 4, no collision, so it gets placed at location 4.
- 23 will be placed at: h (23) = [h (23) + f (0^2)] mod 10 = 3, collision occurred, so we increase the value of i.
  - h (23) = [h (23) + f (1^2)] mod 10 = 4, again collision occurred, so we increase the value of i.
  - h (23) = [h (23) + f (2^2)] mod 10 = 3 + 4 = 7, no collision occurred, so it gets placed at location 7.
- Quadratic probing avoids clustering of elements and thus improves the searching time.



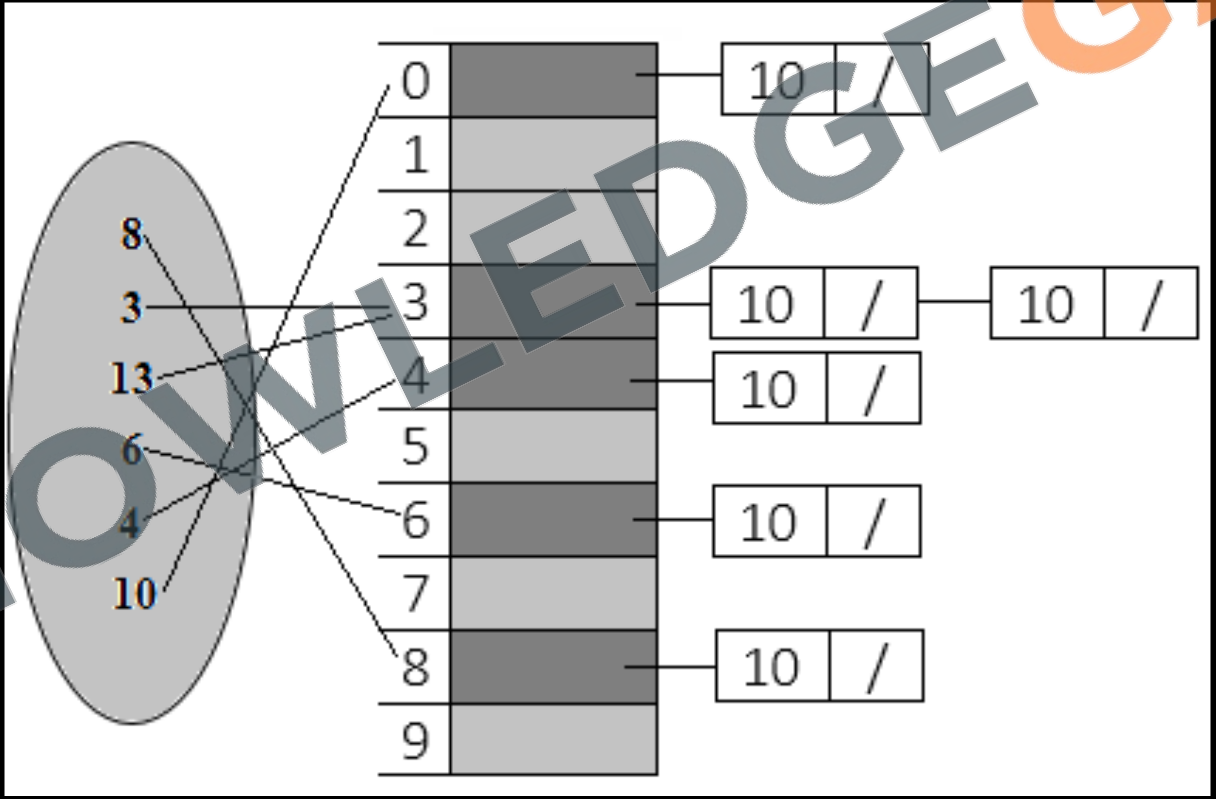
- **Advantage**
  - Quadratic probing can be a more efficient algorithm in a closed hashing table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune.
  - It also provides good memory caching because it preserves some locality of reference; however, linear probing has greater locality and, thus, better cache performance.

- **Disadvantage**
  - Quadratic probing lies between the two in terms of cache performance and clustering.

- **Performance of Open Addressing**: Like Chaining, performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of table (simple uniform hashing)
  - m = Number of slots in hash table
  - n = Number of keys to be inserted in hash table
  - Load factor $\alpha = n/m$ (< 1)
  - Expected time to search/insert/delete < $1/(1 - \alpha)$
  - So Search, Insert and Delete take $(1/(1 - \alpha))$ time

Chaining

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value. In *chaining*, we place all the elements that hash to the same slot into the same linked list.



http://www.knowledgegate.in/GATE

- **Advantage: -** Chaining is simple
- **Disadvantage: -**but requires additional memory outside the table.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

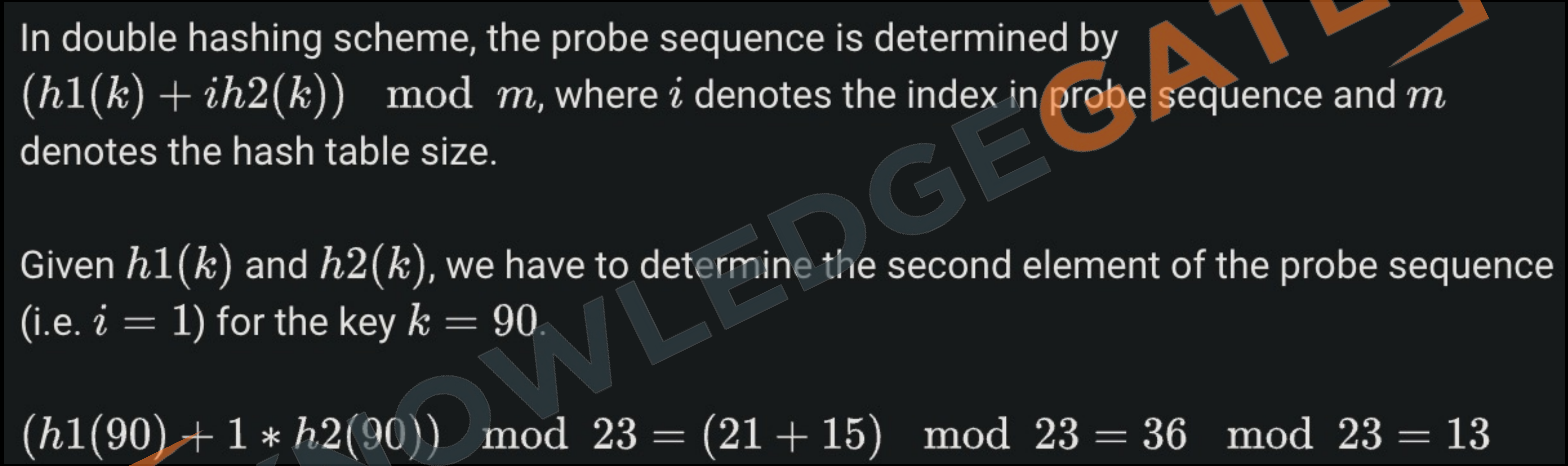
Q Consider a hash table with 9 slots. The hash function is $h(k) = k \bmod 9$. The collisions are resolved by chaining. The following 9 keys are inserted in the order: 5, 28, 19, 15, 20, 33, 12, 17, 10. The maximum, minimum, and average chain lengths in the hash table, respectively, are

(A) 3, 0, and 1 **(B)** 3, 3, and 3 **(C)** 4, 0, and 1 **(D)** 3, 0, and 2

Double Hashing

- Double hashing is used in hash tables to handle hash collisions using open addressing. It uses two hash values: the primary for table indexing and the secondary to set an interval for searching. This method differs from linear and quadratic probing. With double hashing, data mapped to the same location has varied bucket sequences, reducing repeated collisions.
- Given two random, uniform, and independent hash functions h_1 and h_2, the ith location in the bucket sequence for value k in a hash table of |T| buckets is: $h(i, k) = (h_1(k) + i \cdot h_2(k)) \bmod |T|$. Generally, h_1 and h_2 are selected from a set of universal hash functions; h_1 is selected to have a range of $\{0, |T| - 1\}$ and h_2 to have a range of $\{1, |T| - 1\}$. Double hashing approximates a random distribution; more precisely, pair-wise independent hash functions yield a probability of $(n/|T|)^2$ that any pair of keys will follow the same bucket sequence

Q Consider a double hashing scheme in which the primary hash function is $h_1(k)$ = k mod 23, and the secondary hash function is $h_2(k)$ = 1+(k mod 19). Assume that the table size is 23. Then the address returned by probe 1 in the probe sequence (assume that the probe sequence begins at probe 0) for key value k = 90 is _________ .?


In double hashing scheme, the probe sequence is determined by $(h1(k) + ih2(k)) \mod m$, where i denotes the index in probe sequence and m denotes the hash table size.

Given $h1(k)$ and $h2(k)$, we have to determine the second element of the probe sequence (i.e. $i = 1$) for the key $k = 90$.

$$(h1(90) + 1 * h2(90)) \mod 23 = (21 + 15) \mod 23 = 36 \mod 23 = 13$$

http://www.knowledgegate.in/GATE