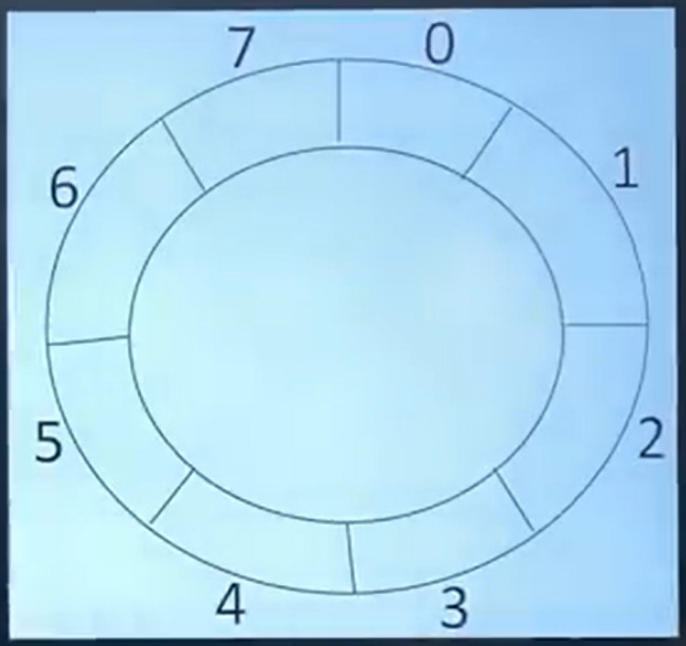
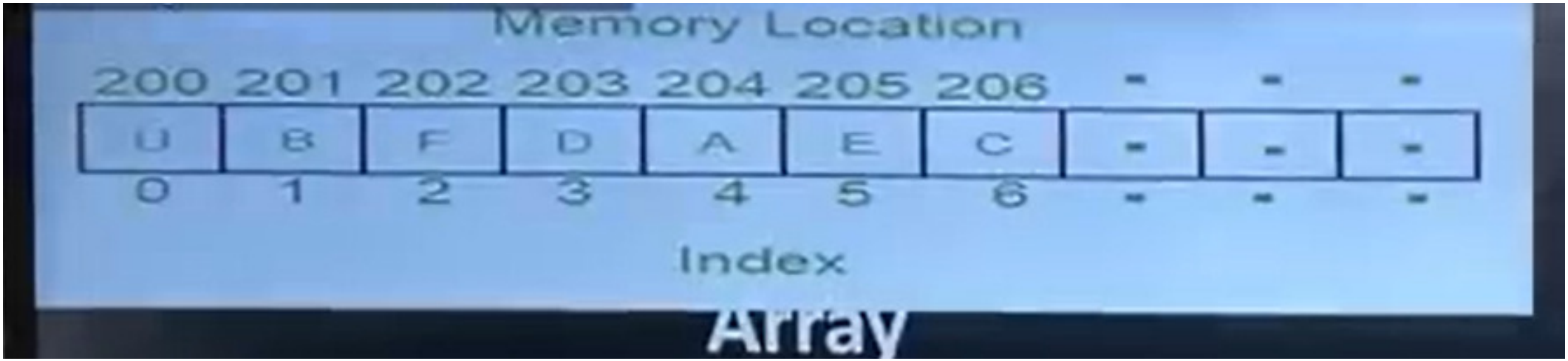
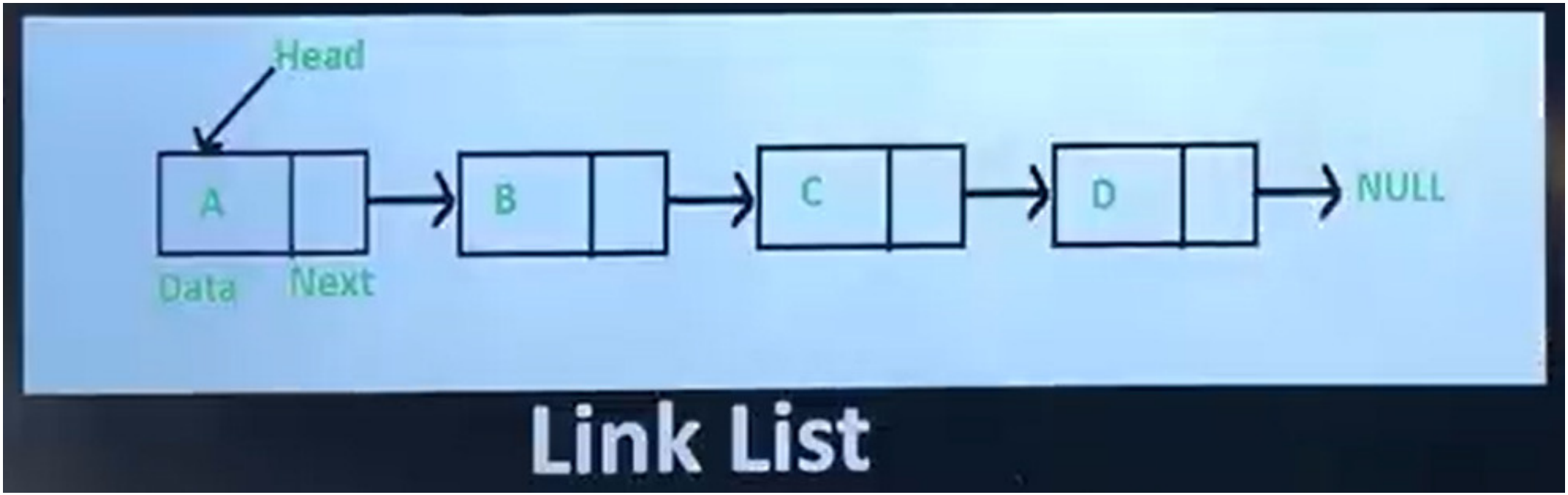
Circular Queue

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
}
```




Memory Location
200 201 202 203 204 205 206 - - -
U B F D A E C - - -
0 1 2 3 4 5 6 - - -
Index
Array


Head
A
B
C
D
NULL
Data
Next
Link List

(Ch-3) Linked lists

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

Implementation of link list

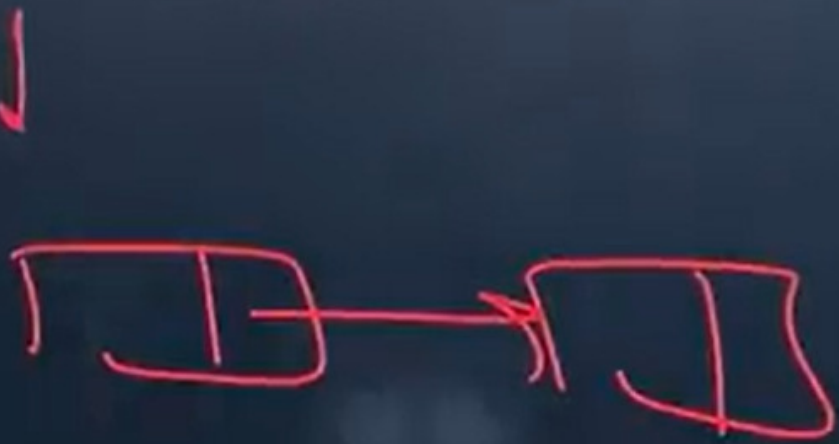
```
struct node
{
    int data;
    struct node *next;
};
```

(Ch-3) Linked lists

```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;
}
```

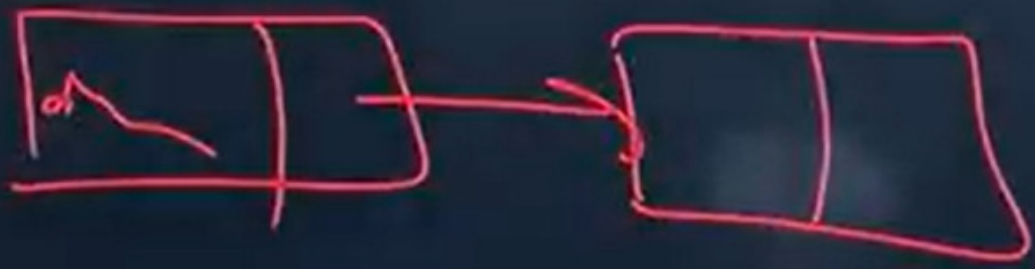
(Ch-4) Stack

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

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



(Ch-4) Stack

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

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



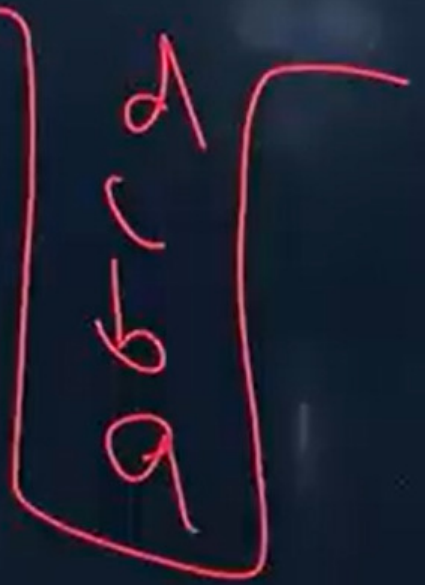
(Ch-4) Stack

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

(Ch-4) Stack

```
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);
    }
}
```

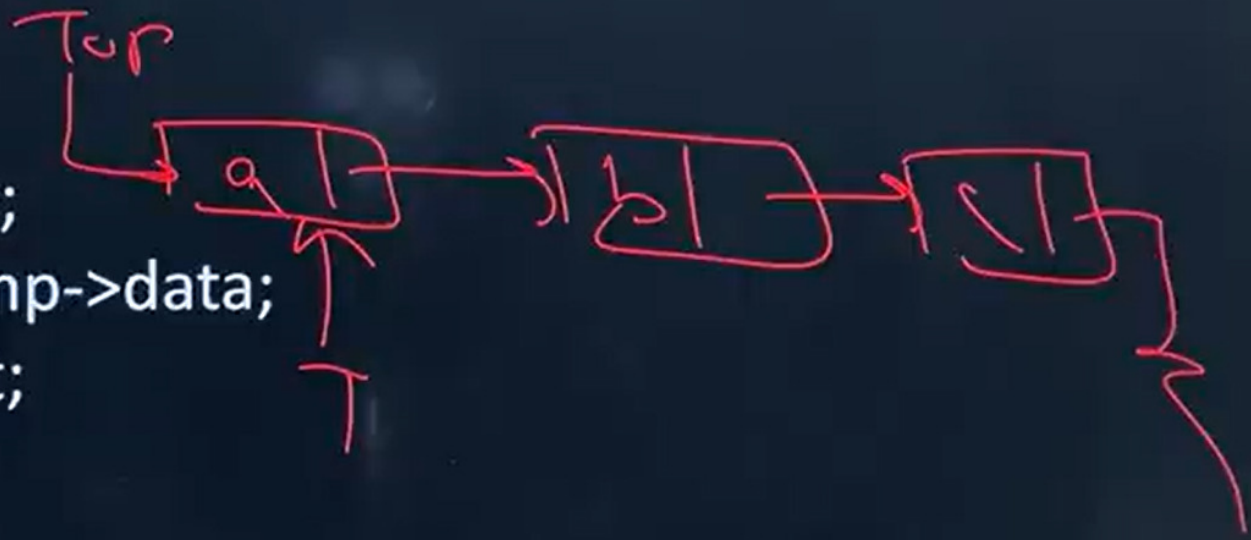
a b c d



(Ch-4) Stack

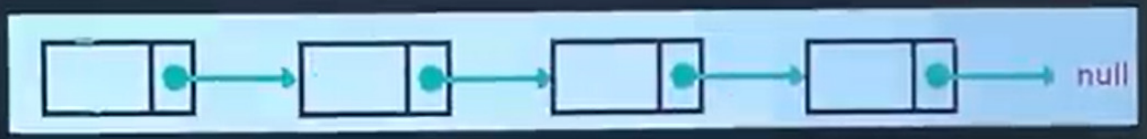
```
void push(Stack* s, int item)
{
    Node* newNode = (Node*)
    malloc(sizeof(Node));
    if (newNode == NULL)
    {
        printf("Stack overflow!\n");
        exit(1); // Exit with an error code
    }
    newNode->data = item;
    newNode->next = s->top;
    s->top = newNode;
}
```

```
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;
}
```

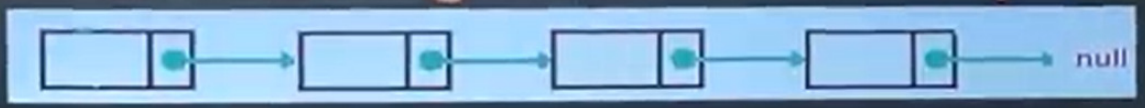


Q Write a C-style pseudocode for searching a key in a link list recursively, where pointer head have the address of the first node of the list?

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



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

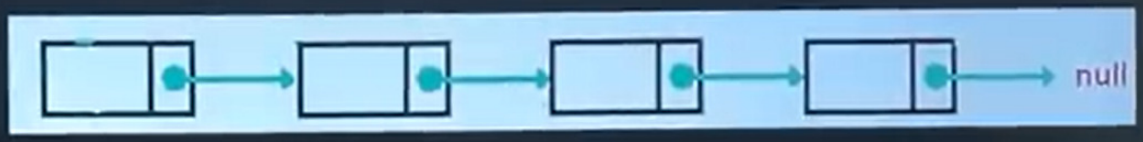



```
void insertAtBeginning(Node** head, int key)
{
    Node* newNode = createNode(key);
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
```

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?



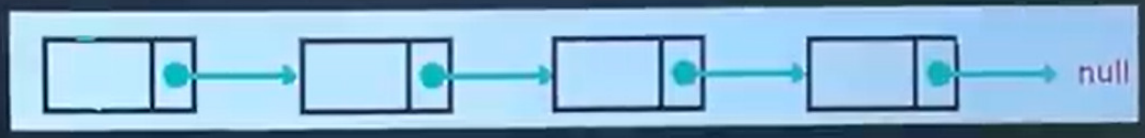

```
void deleteAtBeginning(Node** head)
{
    if (*head == NULL)
    {
        printf("List is already empty.\n");
        return;
    }
    Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}
```

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);
}
```

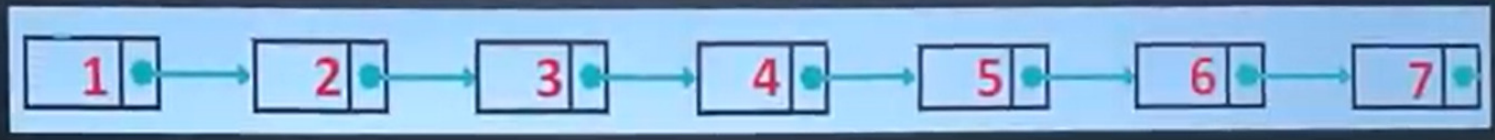
Q Write a C-style pseudocode for reversing a link-list in a iteratively?

```
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;

}
```



Basic Operations on a Linked List

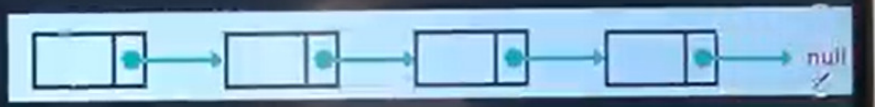
Q The following C function takes a single-linked list of integers as a parameter and rearranges the elements of the list. The function is called with the list containing the integers 1, 2, 3, 4, 5, 6, 7 in the given order. What will be the contents of the list after the function completes execution?




```
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;
        }
}
```

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

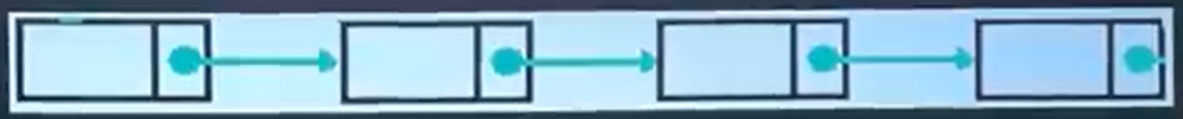
```
Node* reverseListRecursive(Node* head)
{
    if (head == NULL || head->next == NULL)
    {
        return head;
    }
    Node* rest = reverseListRecursive(head->next);
    head->next->next = head;
    head->next = NULL;
    return rest;
}
```



(Ch-3) Linked lists

Basic Operations on a Linked List

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.

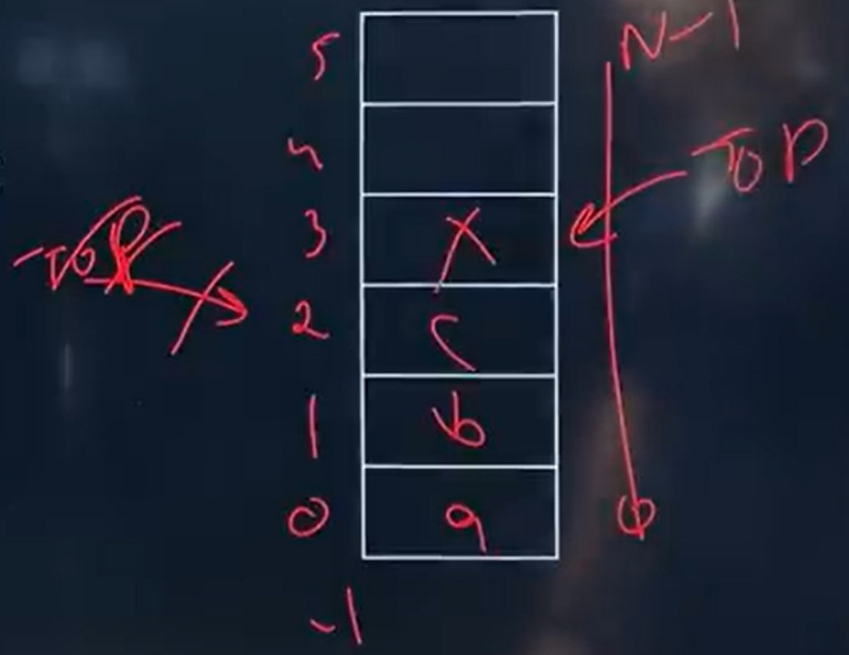


```
typedef struct node
{
        int value;
        struct node *next;
}Node;

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;

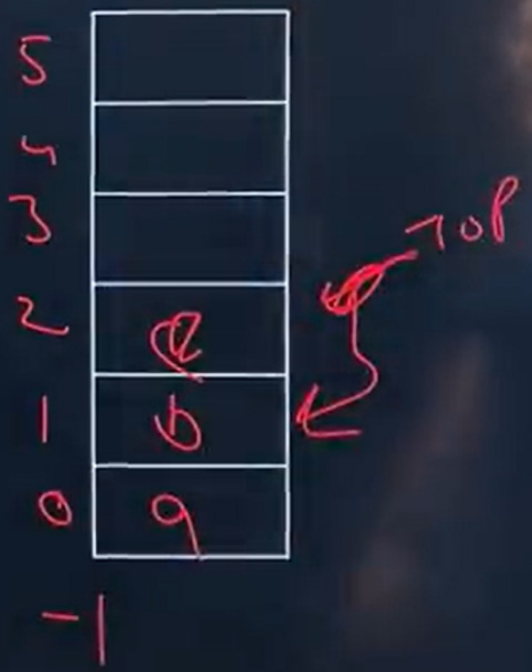
Push operation: - The process of adding new element to the top of stack is called push operation. the new element will be inserted at the top after every push operation the top is incremented by one. in the case the array is full and no new element can be accommodated it is called over-flow condition.

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

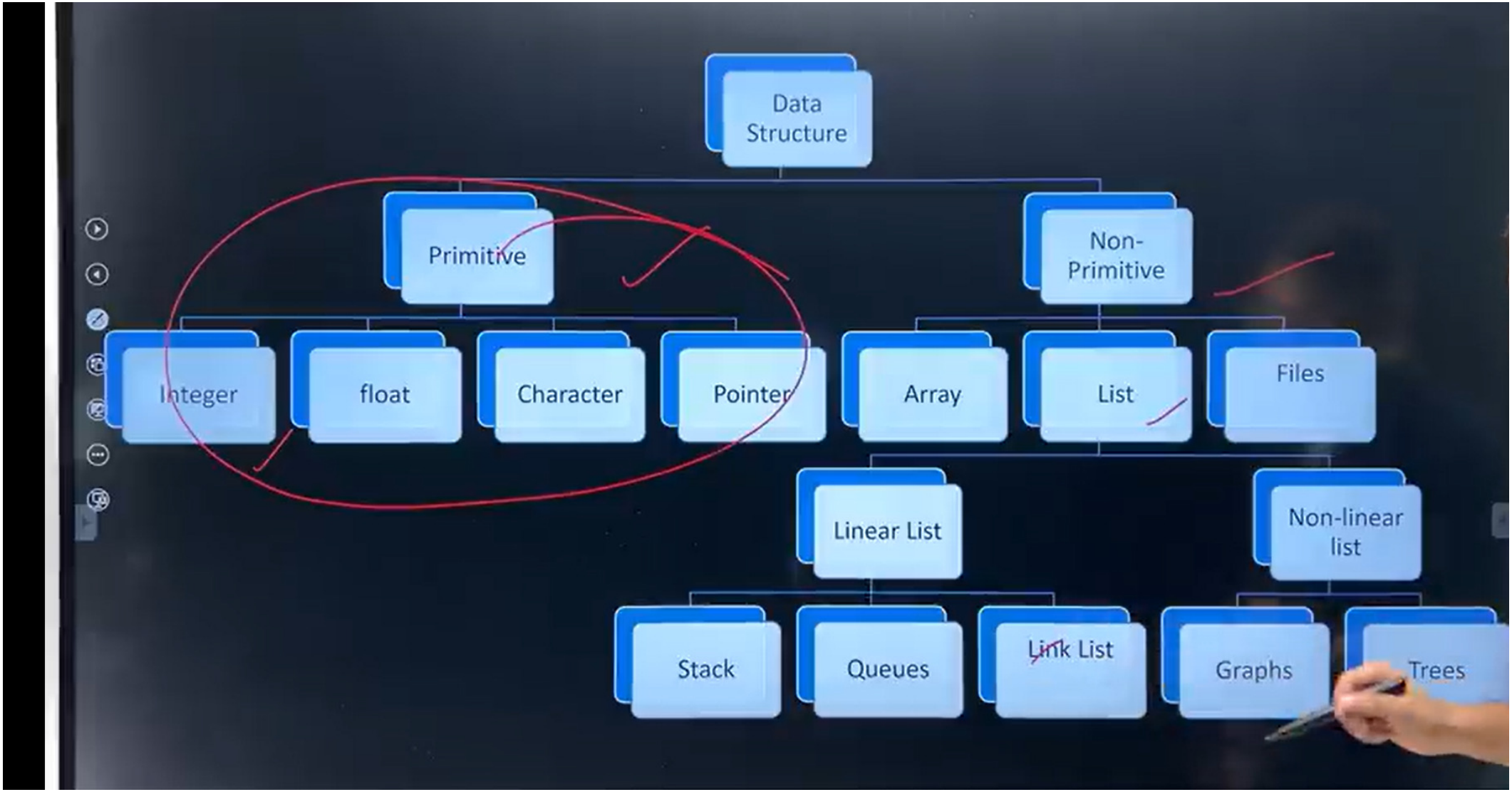


Pop: - The process of deleting an element. from the top of stack is called POP operation, after every POP operation the stack is decremented by one if there is no element in the stack and the POP operation is requested then this will result into a stack underflow condition.

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

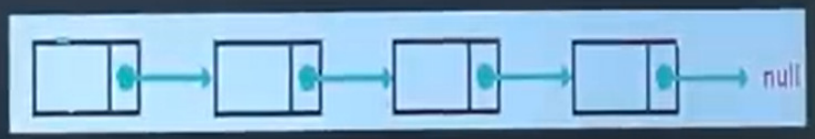


Aspect	Recursion	Iteration
Basic Concept	Function calls itself to solve sub-problems.	Uses loops to repeatedly execute code blocks.
Memory Usage	Typically uses more memory due to call stack.	Uses less memory as it doesn't rely on the call stack.
Termination	Requires a base case to prevent infinite loops.	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.


Data Structure
Primitive
Non-Primitive
Integer
float
Character
Pointer
Array
List
Files
Linear List
Non-linear list
Stack
Queues
Link List
Graphs
Trees

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.

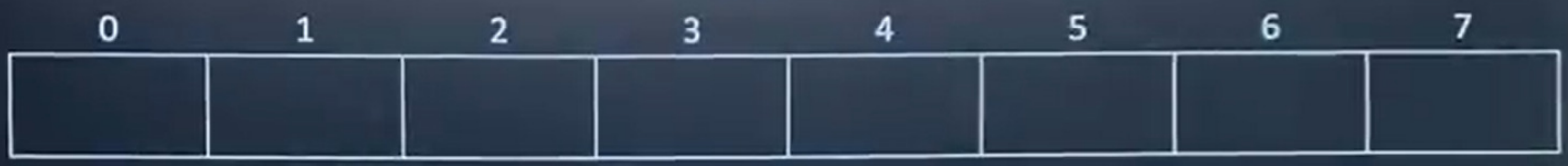
Q Write a C-style pseudocode for Traversing a link list iteratively, where pointer head have the address of the first node of the list?



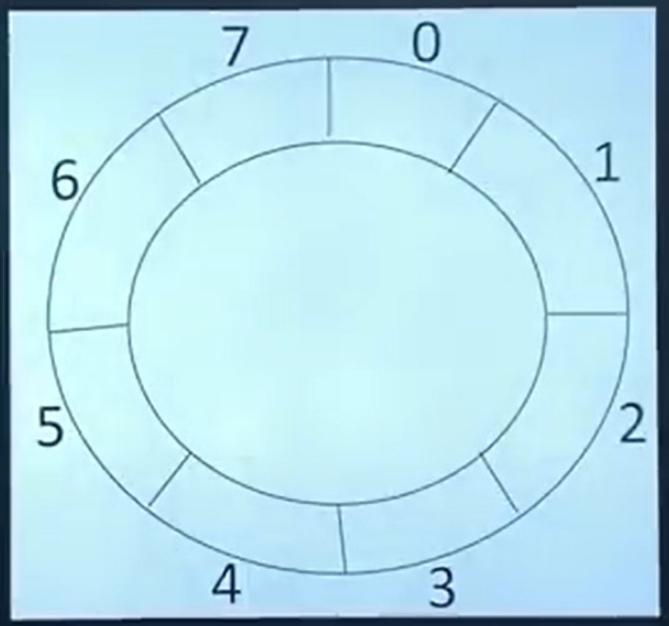

```
void traverseList(Node* head)
{
    Node* current = head;
    while (current != NULL)
    {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
```

Circular Queue

Deletion

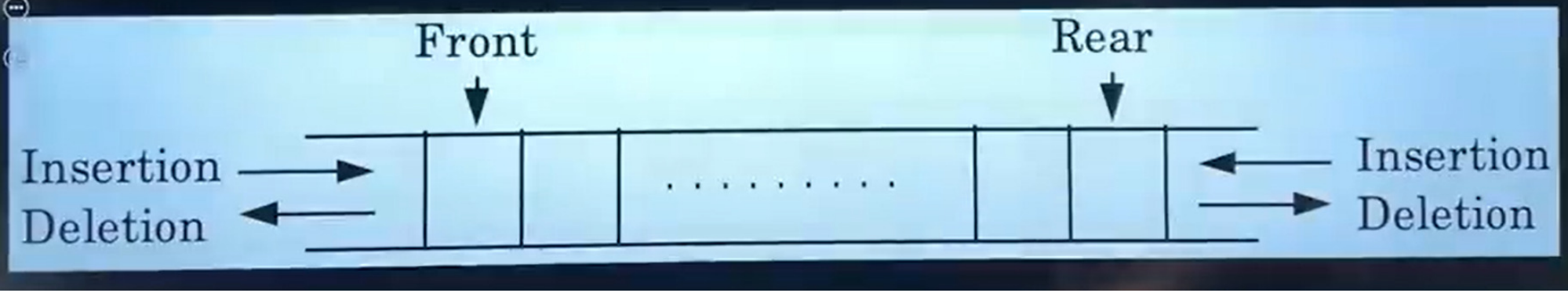



```
Dequeue(QUEUE, N, F, R, ITEM)
{
    if (F == - 1)
        Write under flow and exit
    ITEM = QUEUE[F]
    if (F = = R)
        Set F = -1 && R = -1
    Else
        F = (F + 1)%N
    Return item
}
```

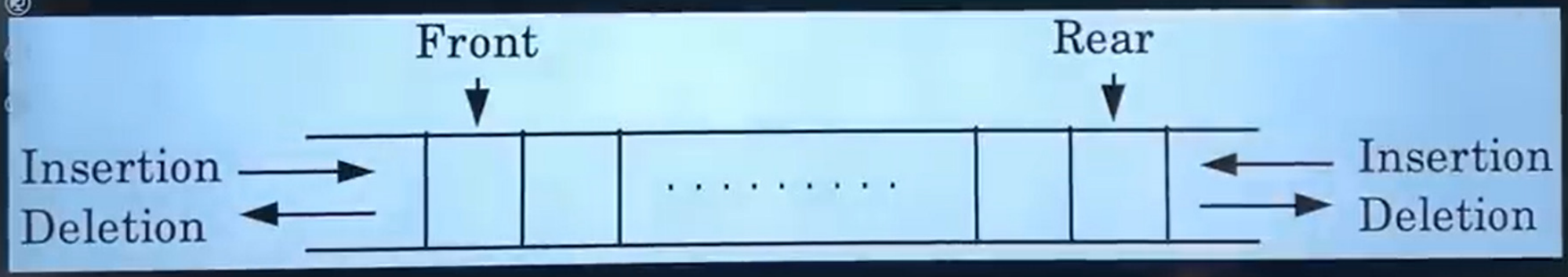


Dequeue

- 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.

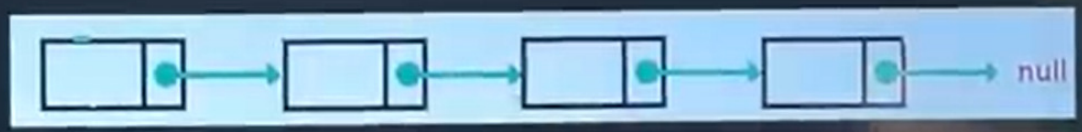


Priority Queue

- 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.

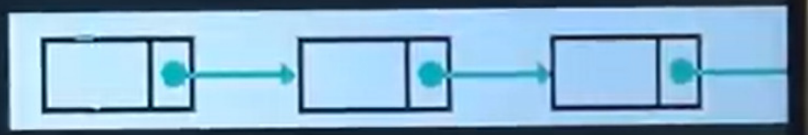
Q Write a C-style pseudocode for searching a key in a link list iteratively, where pointer head have the address of the first node of the list?

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



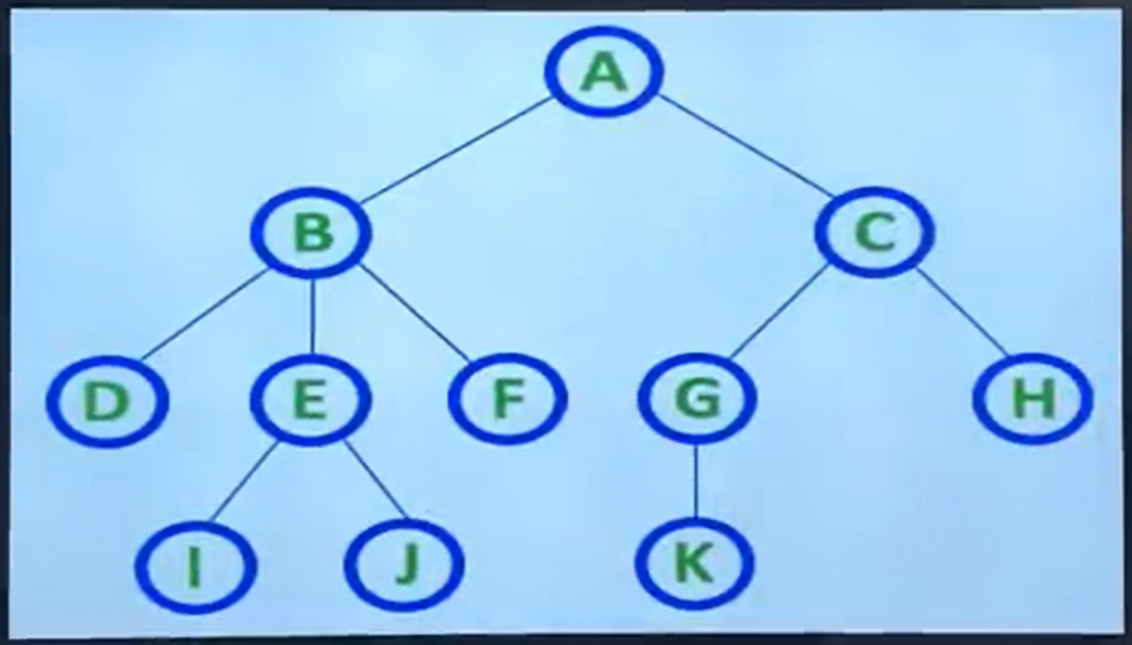
Q Write a C-style pseudocode for Traversing a link list recursively, where pointer head have the address of the first node of the list?

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

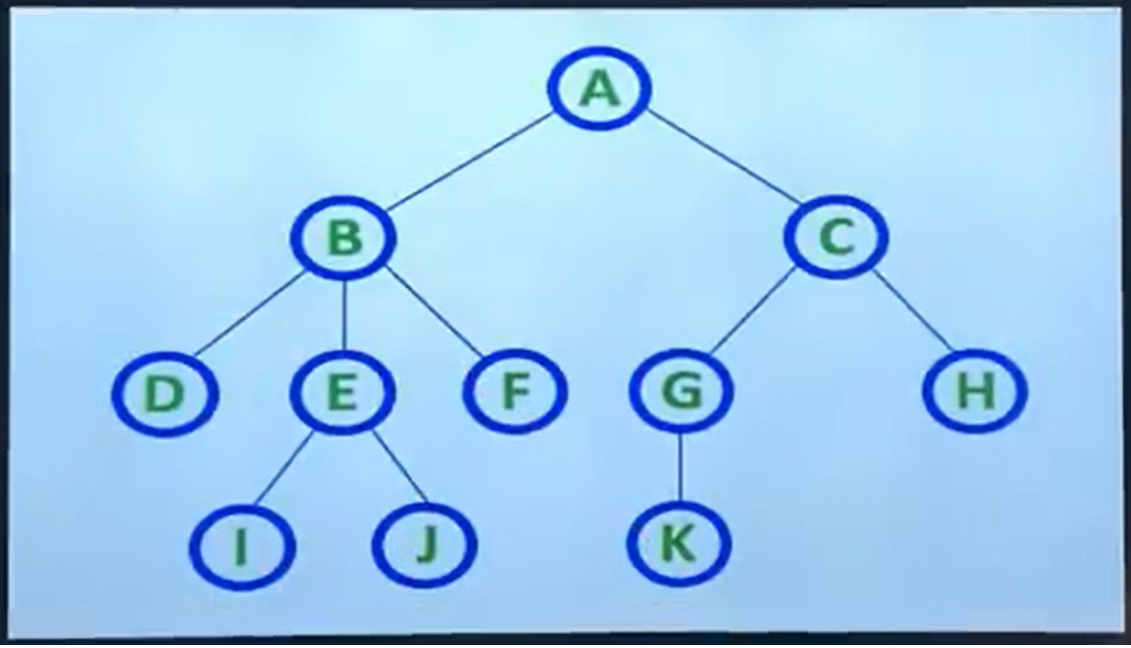


Tree

- 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.

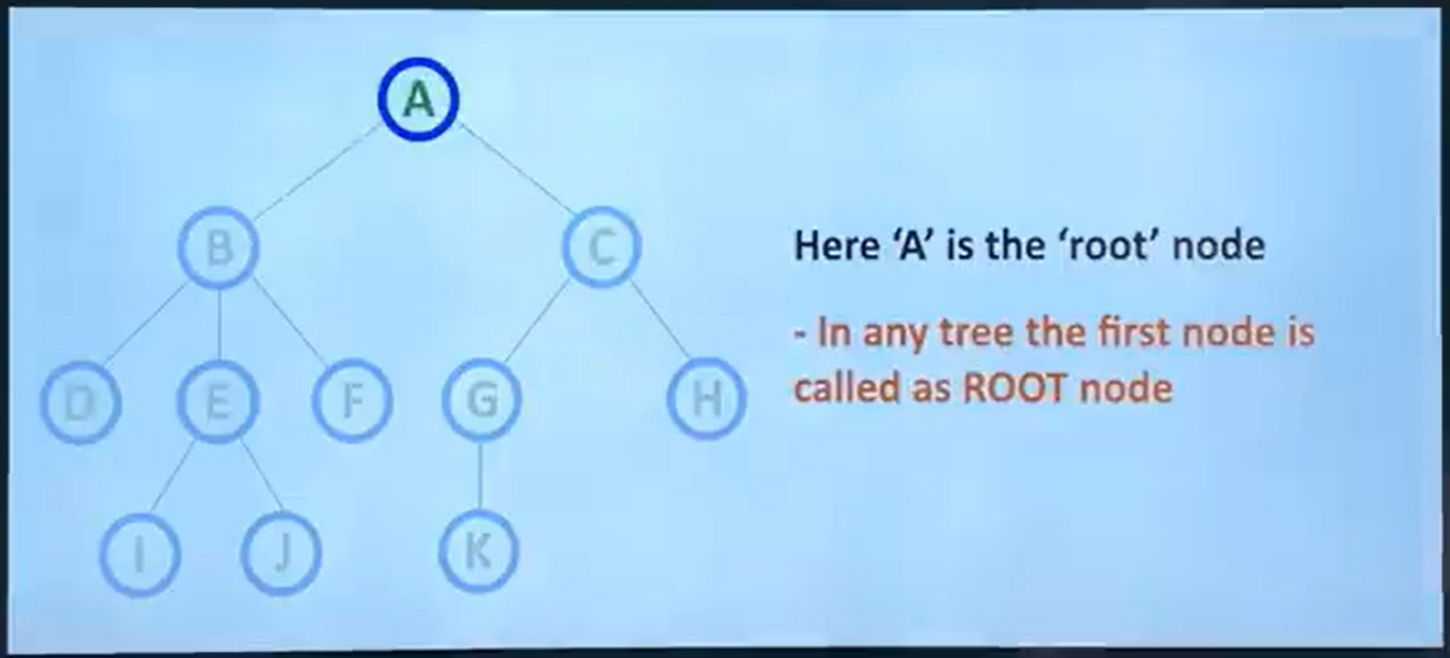


- A tree is a finite set of one or more data items(nodes) such that
  - There is a special data item called root of the tree
  - And its remaining data items are partitioned into number of mutually exclusive (disjoint) subsets, each of which is itself a tree and they are called subtree. i.e. Every node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



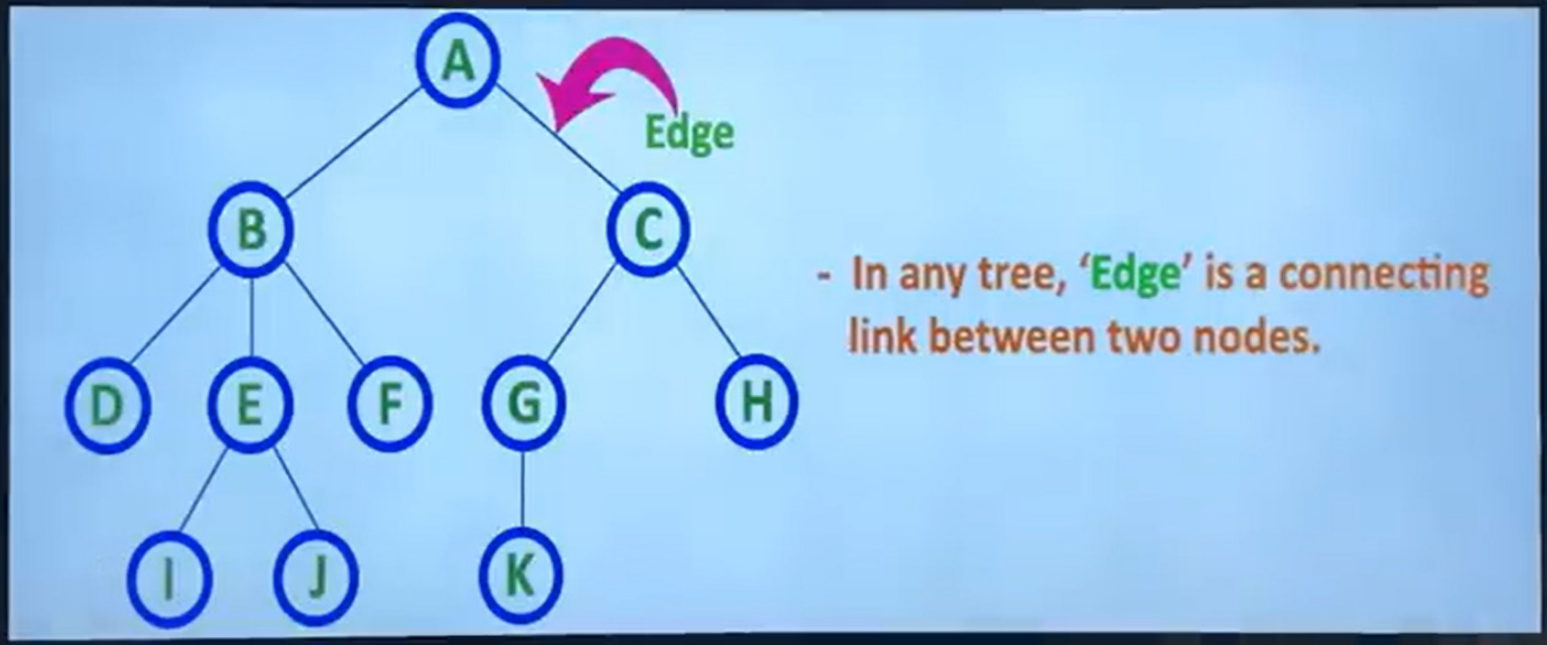
Root

- 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.



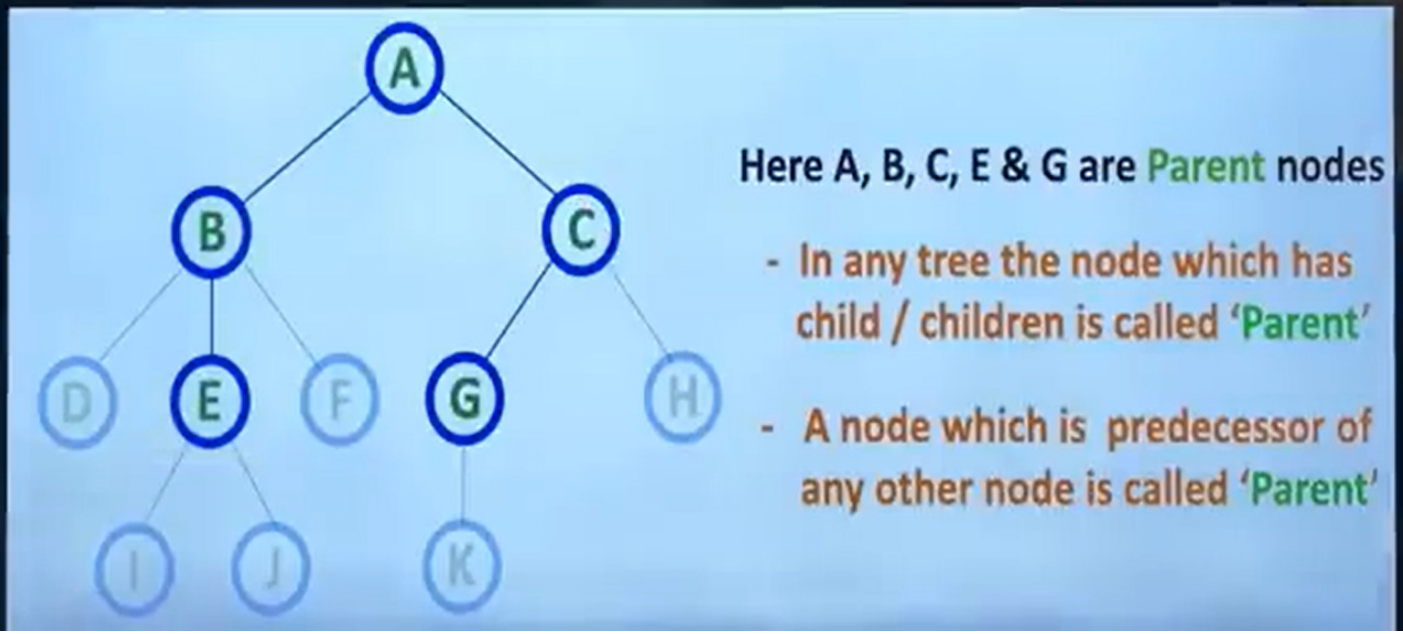
Edge

- 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.



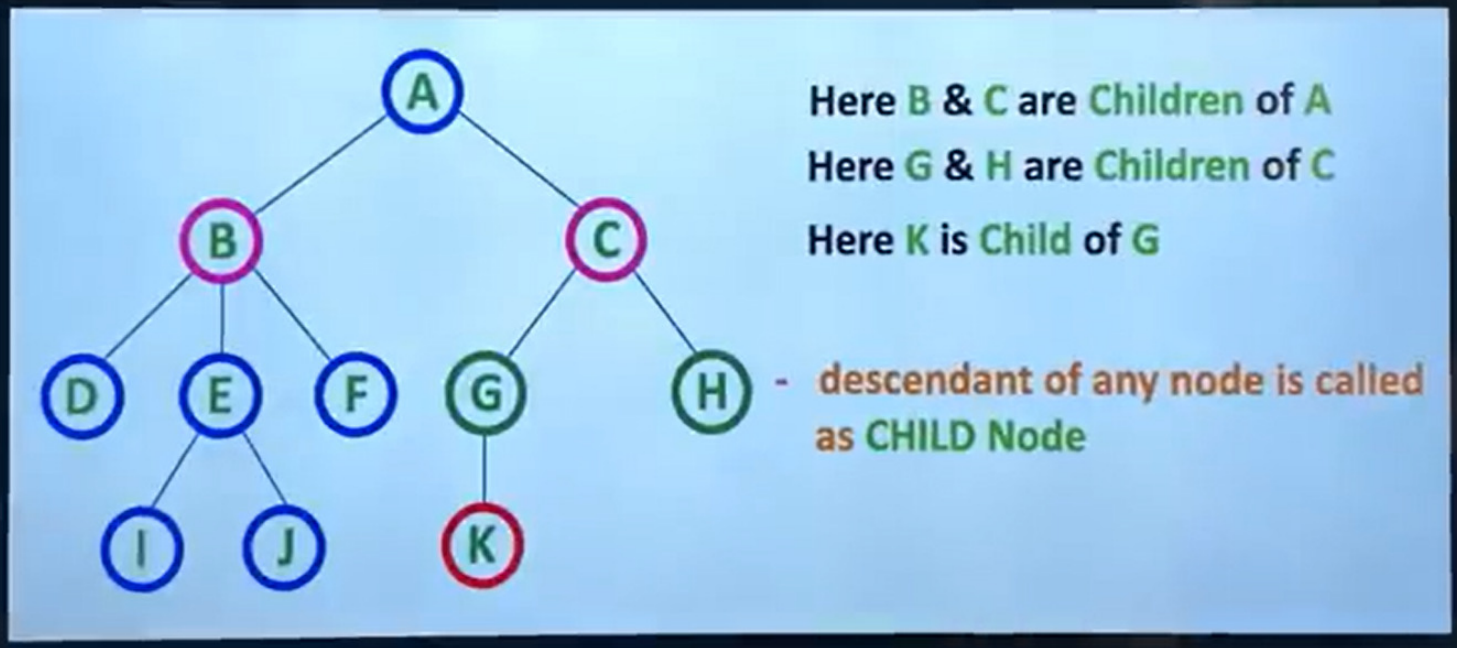
Parent

- 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".



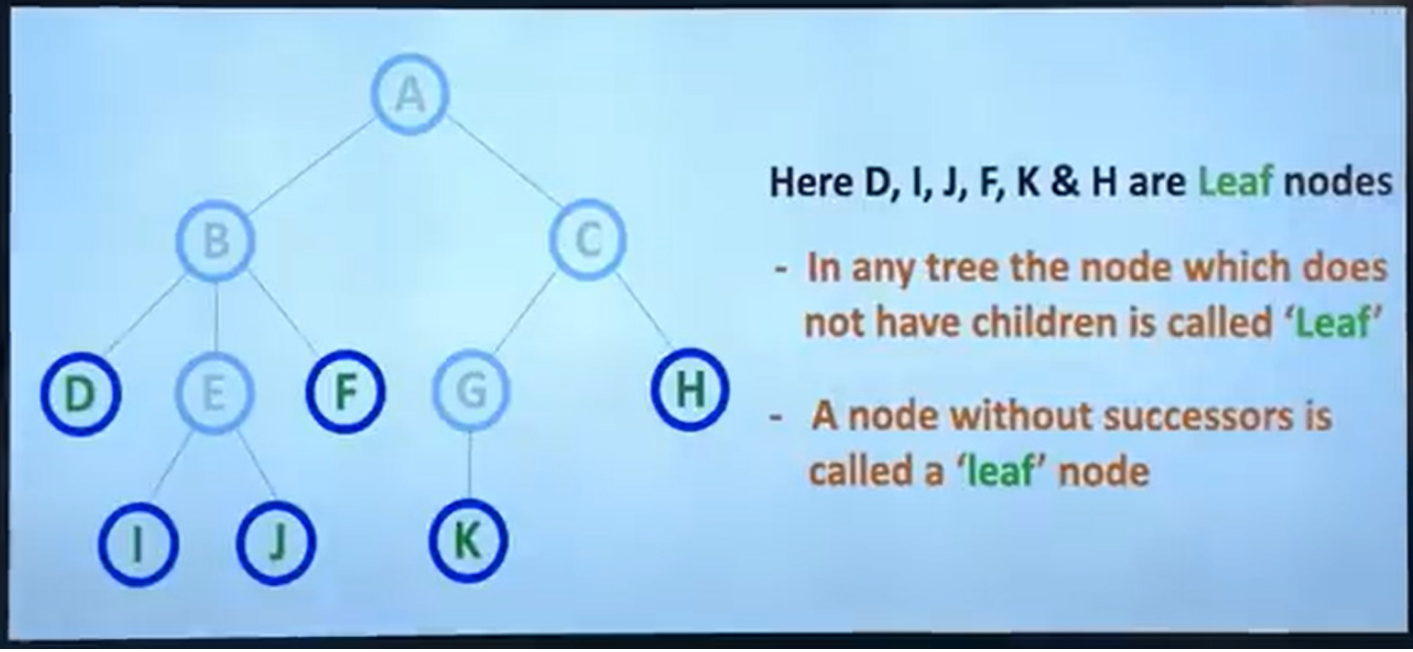
Child

- 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.



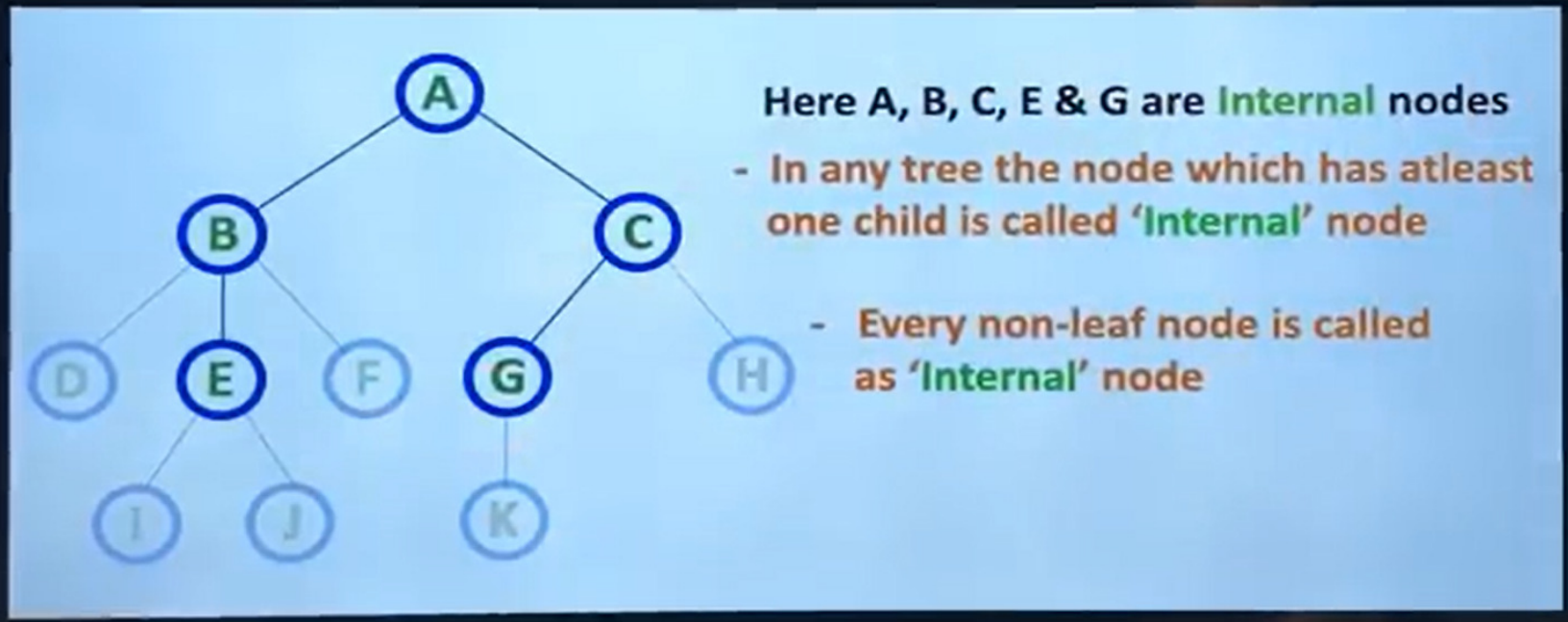
Leaf / External

- 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.



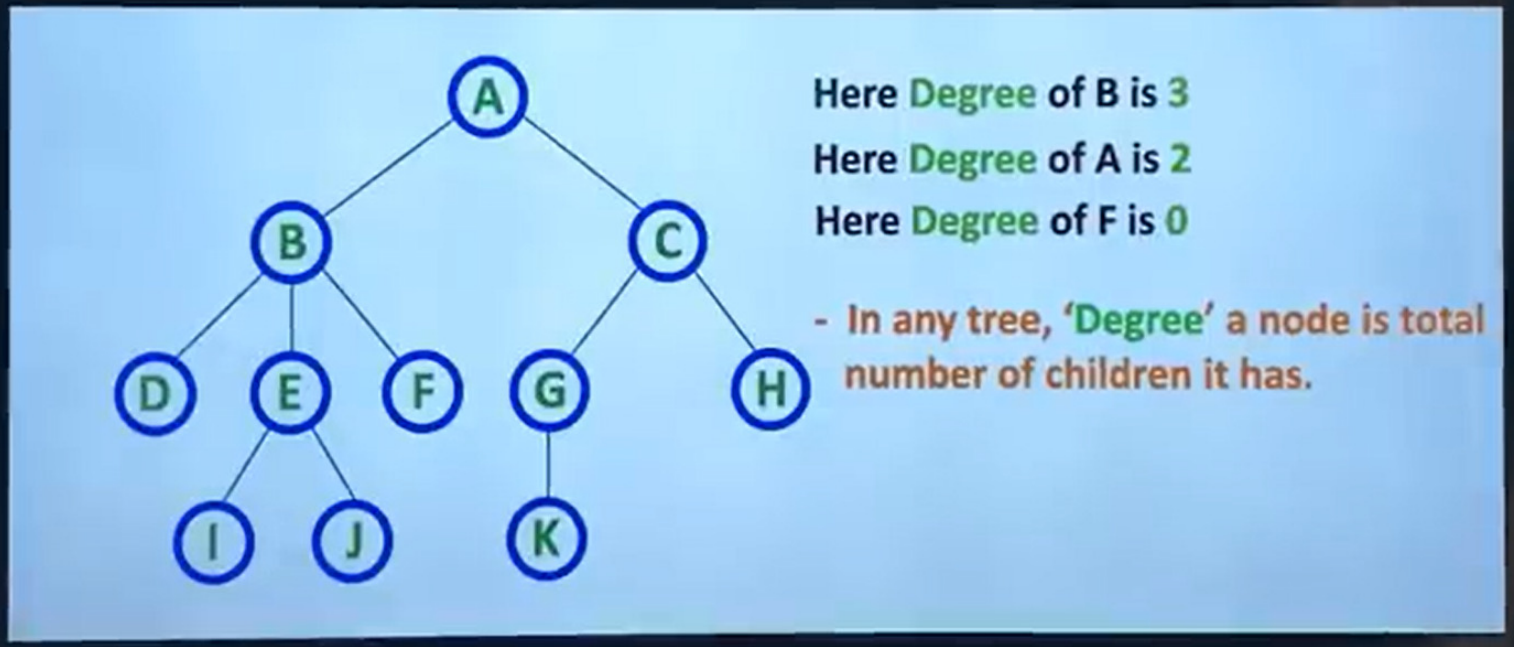
Internal Nodes

- 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.



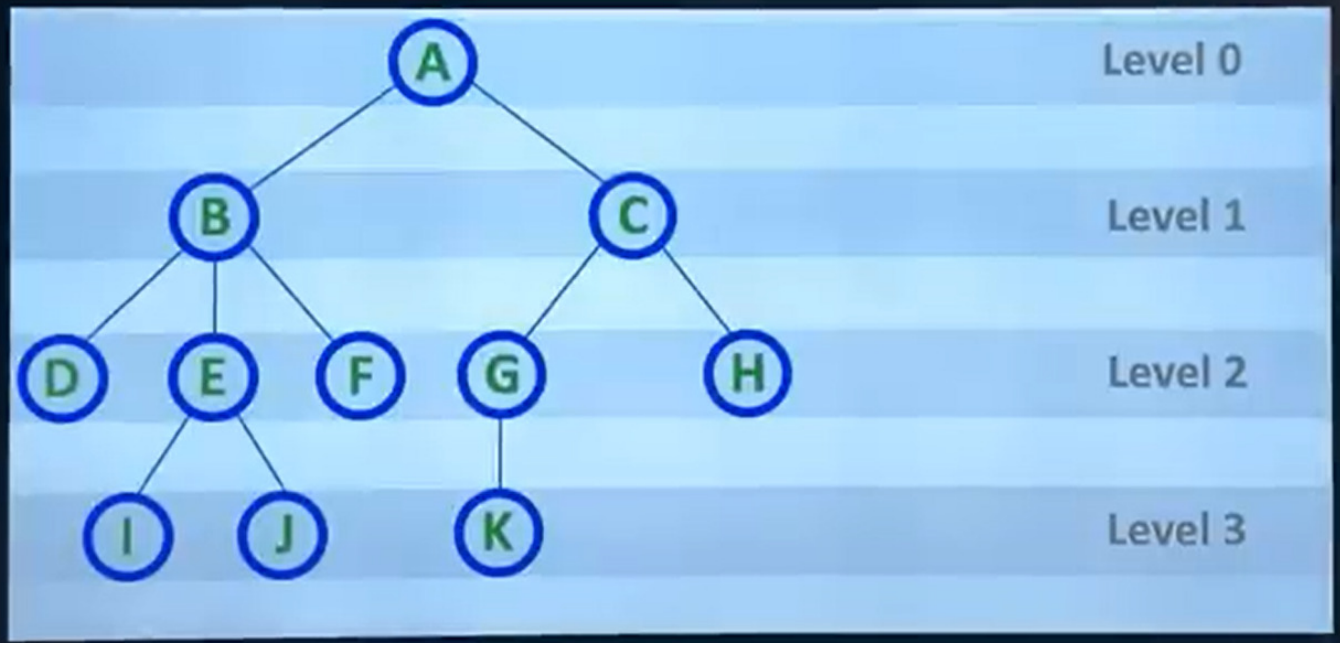
Degree

- 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'



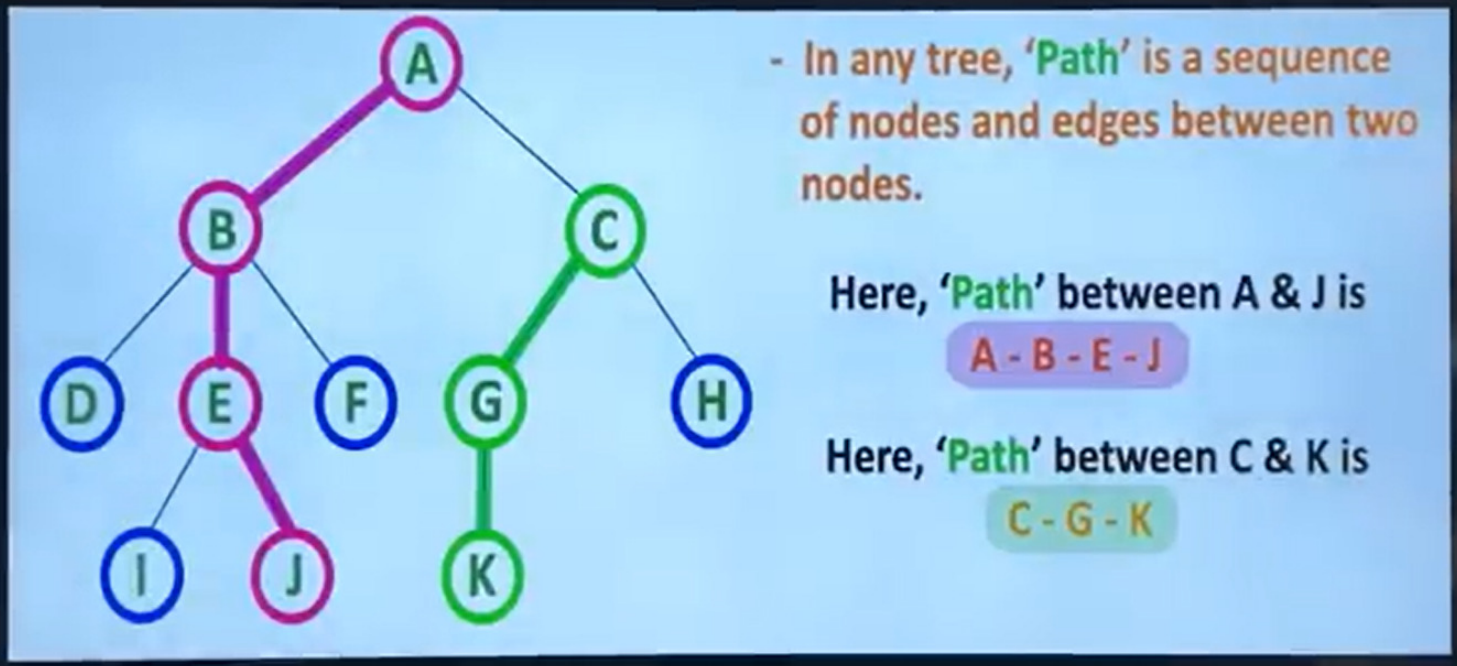
Level / Depth / Height

- 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).



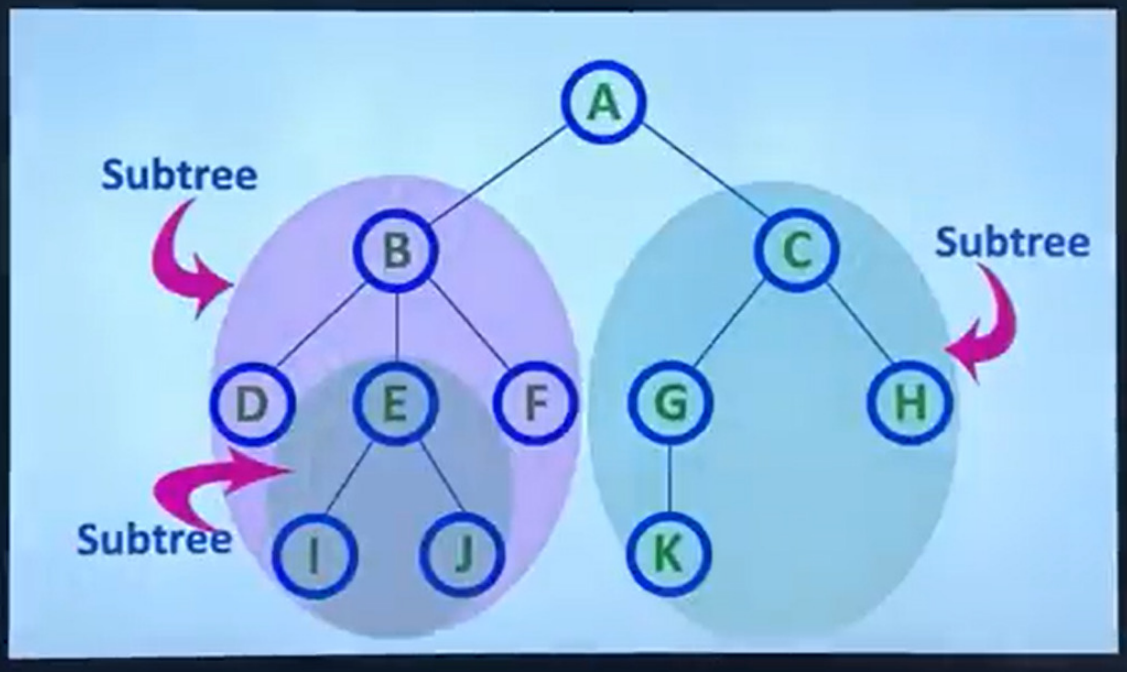
Path

- 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.



Sub Tree

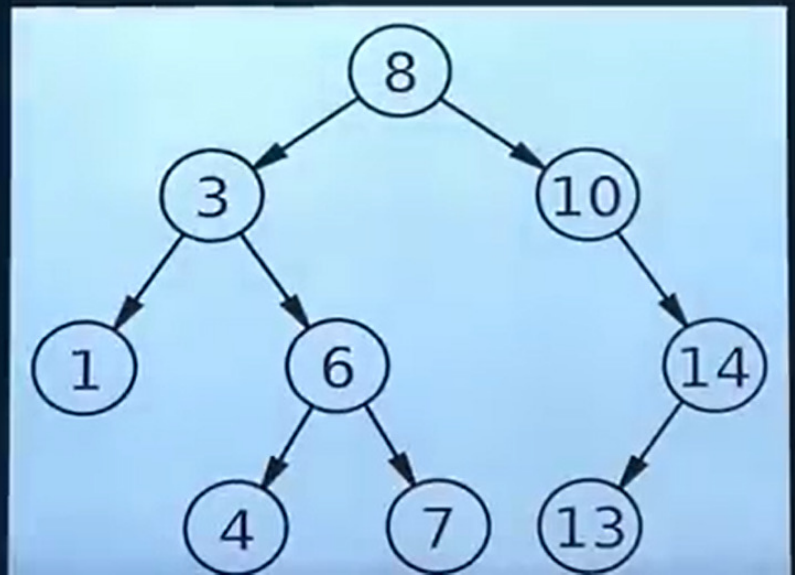
- In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Binary tree

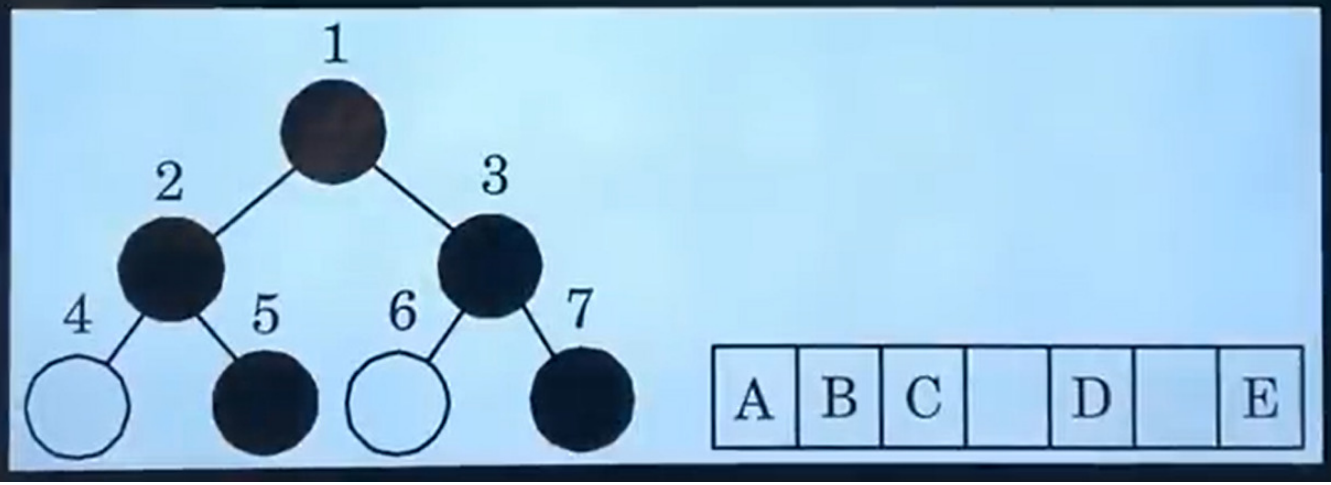
- A binary tree T is defined as a finite set of elements called nodes such that,
  - T is empty (null tree)
  - T contain a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary tree T_1 and T_2
- Direct: - A tree T in which any node can have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

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



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)



Linked representation of binary tree

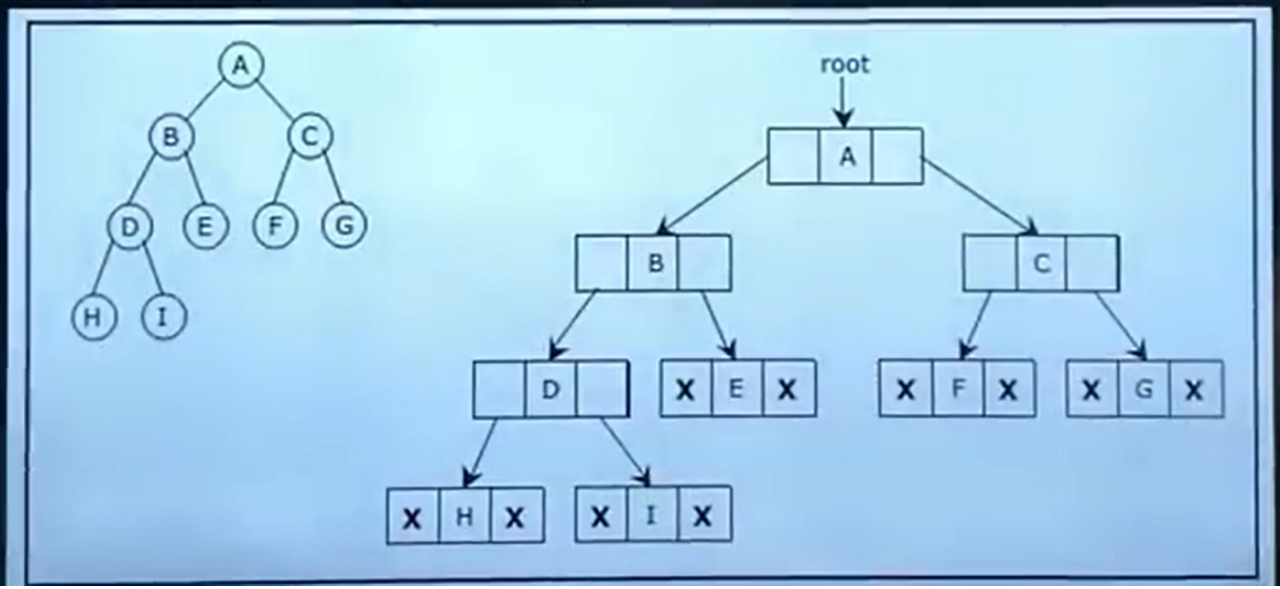
1. Consider a binary tree T which uses three parallel arrays, INFO, LEFT and RIGHT, and a pointer variable ROOT.
2. First of all, each node N of T will correspond to a location K such that :
   a. INFO[K] contains the data at the node N.
   b. LEFT[K] contains the location of the left child of node N.
   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.

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.

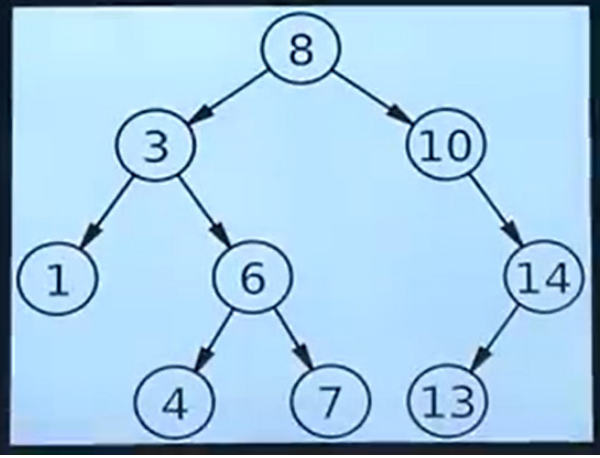
Each node in the linked list contains the following components:

- **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.



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.



- 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)

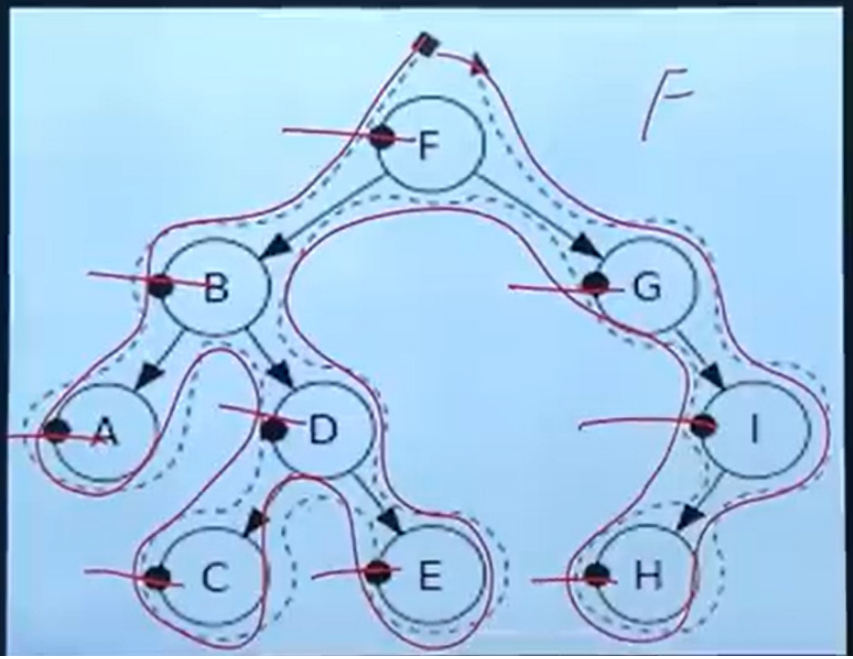
Root L R

L Root R

L R Root

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

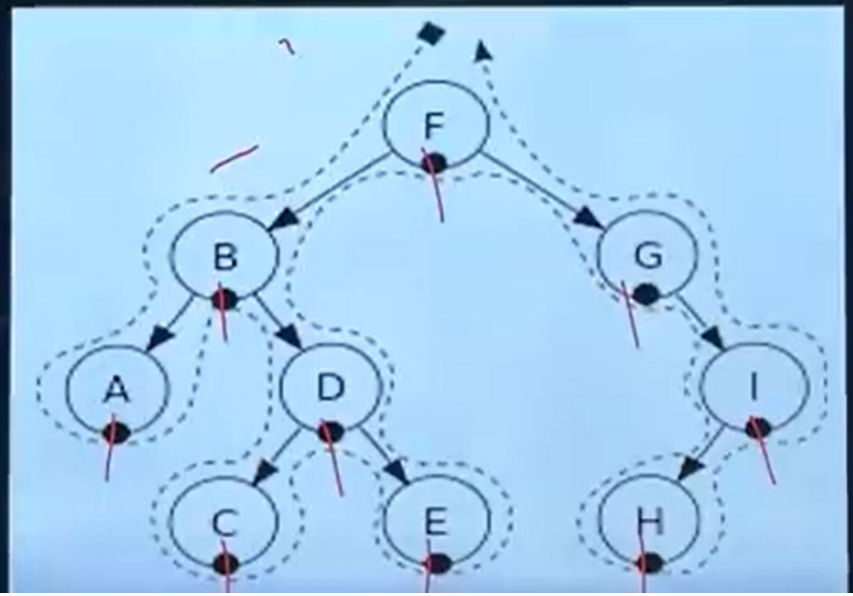
- Check if the current node is empty or null.
- Display the data part of the root (or current node).
- Traverse the left subtree by recursively calling the pre-order function.
- Traverse the right subtree by recursively calling the pre-order function.



In-order (L root R)

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

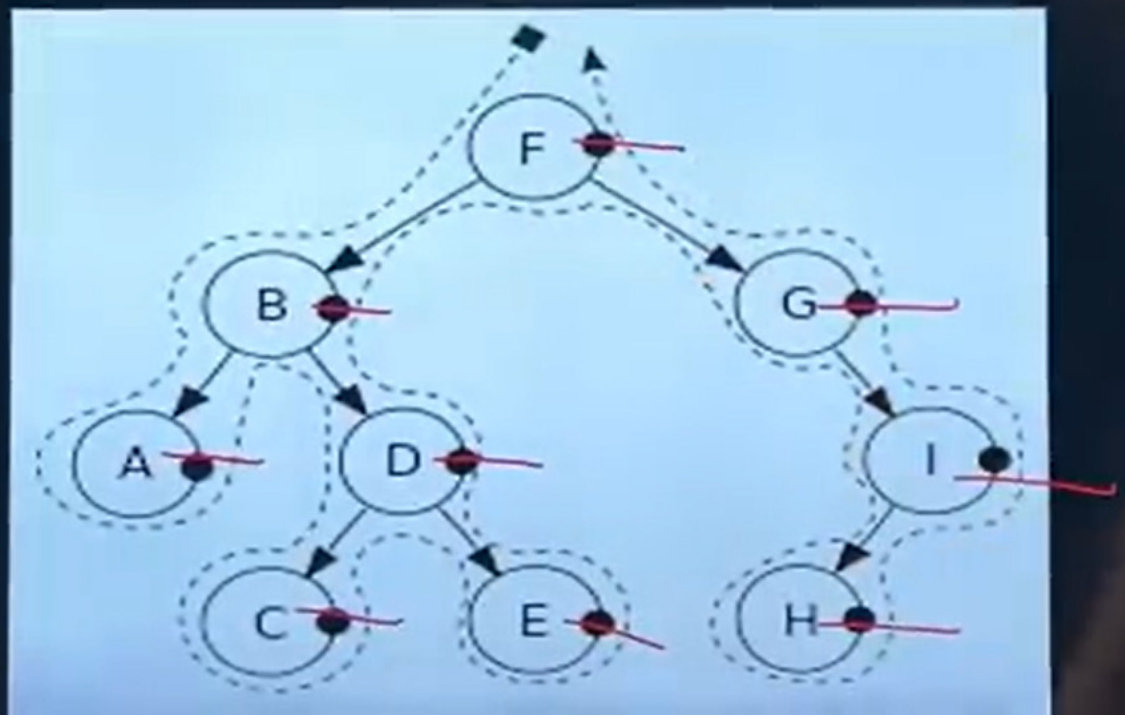
- Check if the current node is empty or null.
- Traverse the left subtree by recursively calling the in-order function.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively calling the in-order function.
- In a binary search tree, in-order traversal retrieves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



Post-order (L R Root)

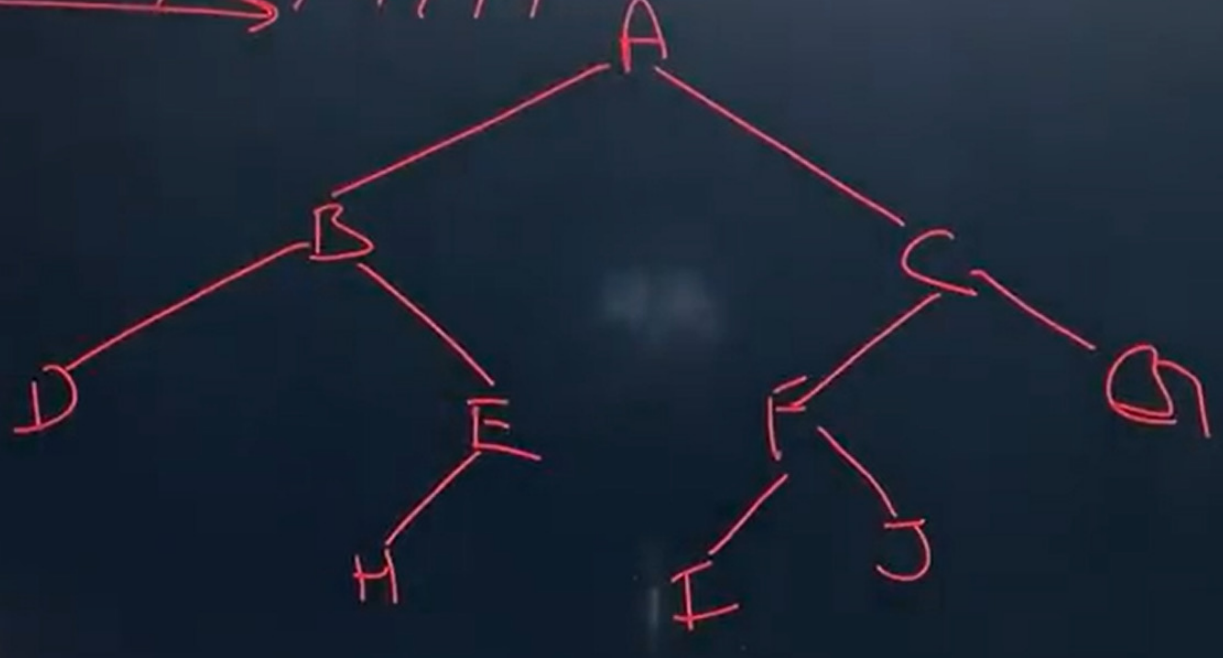
Post-order: A, C, E, D, B, H, I, G, F.

- 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).



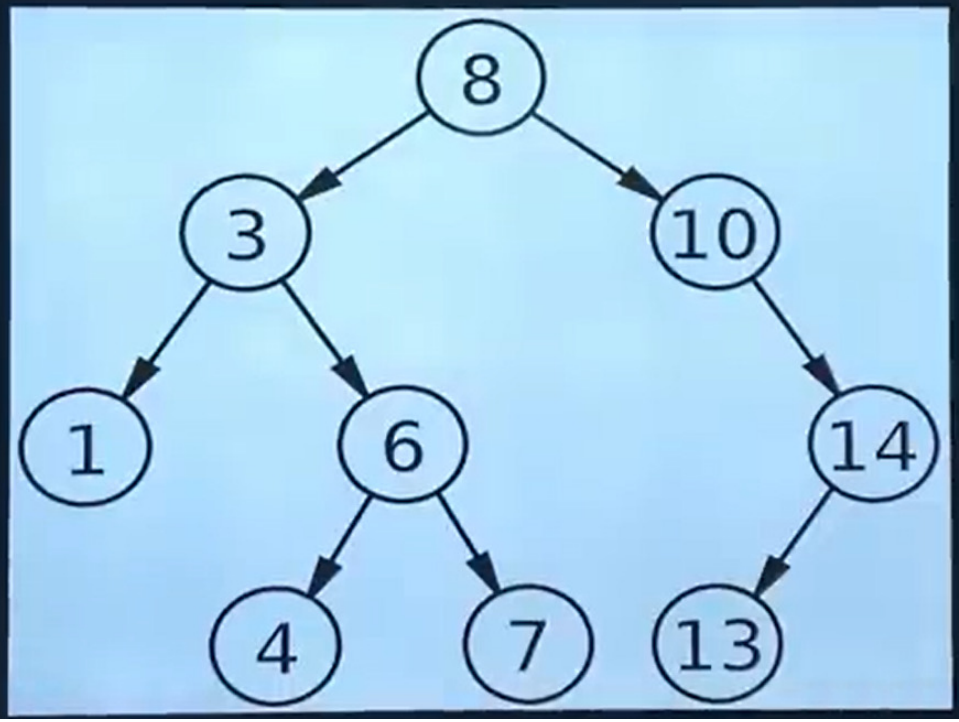
Inorder : D B H E A I F J C G

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



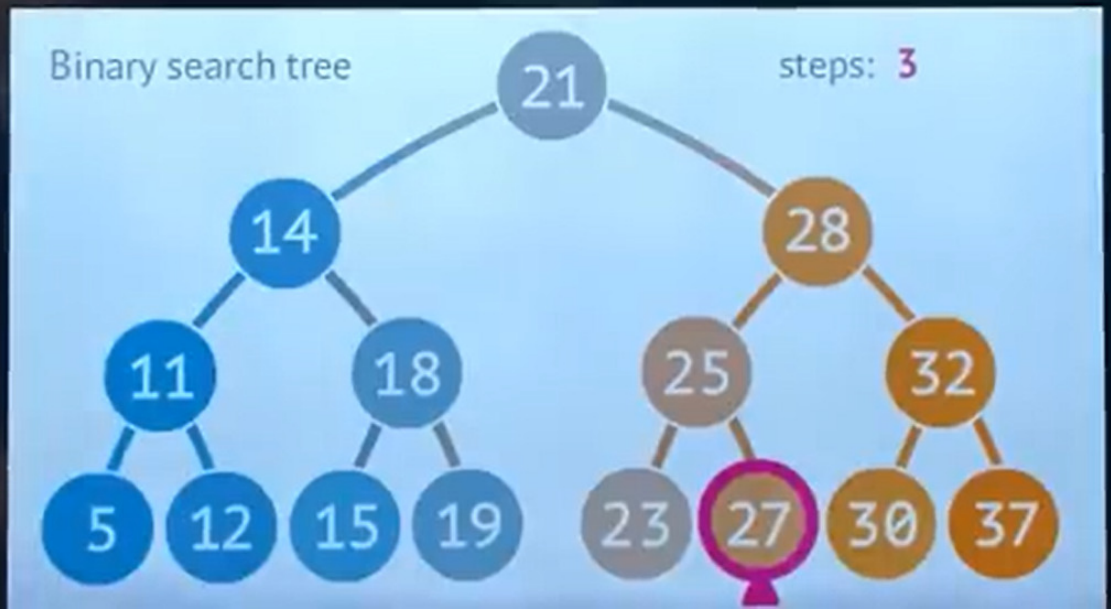
Binary search tree / Ordered tree / Sorted binary tree

- 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.



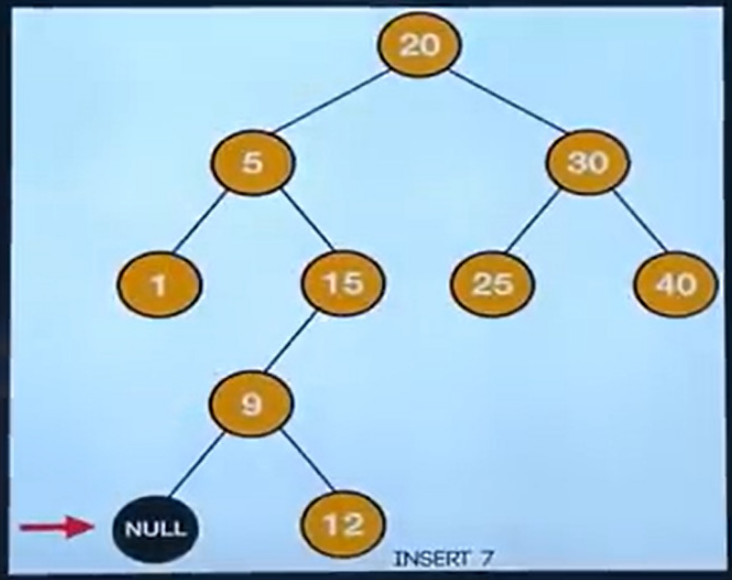
Searching

- 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.



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.

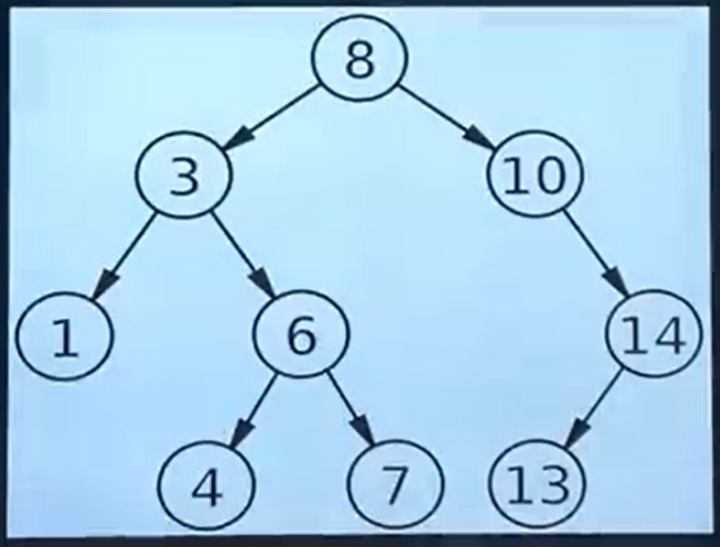


Deletion

- 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.

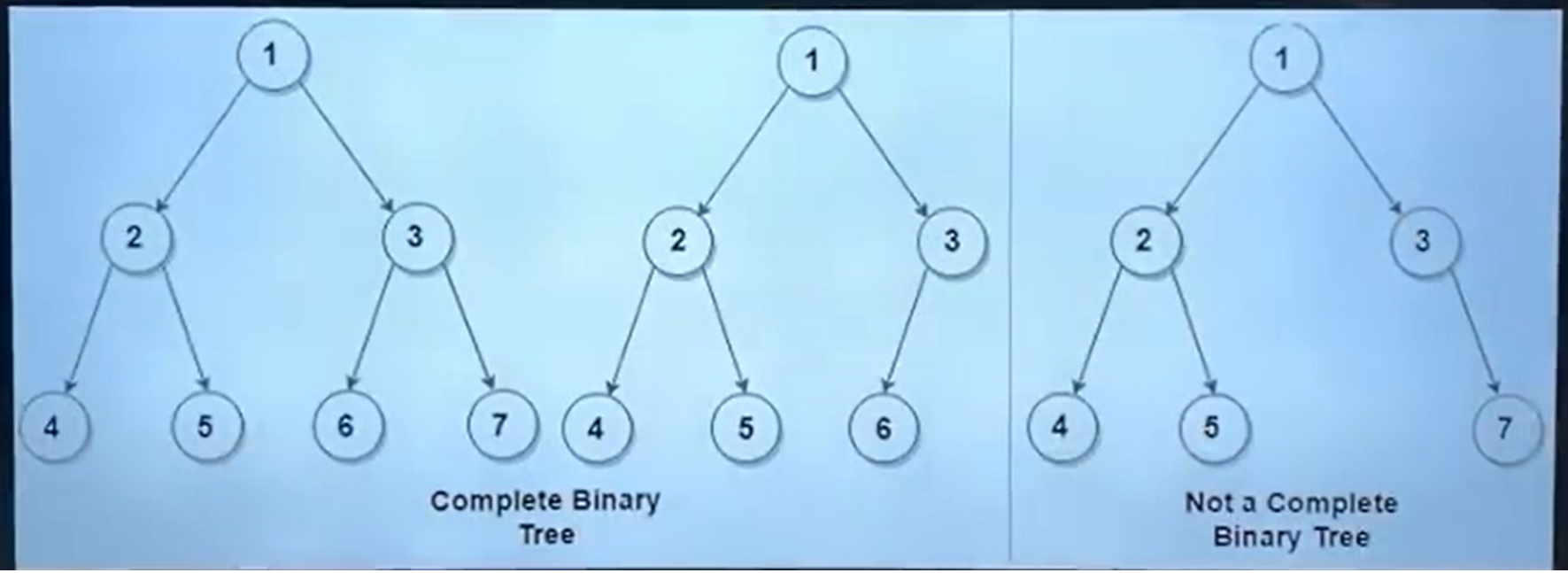


Algorithm	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

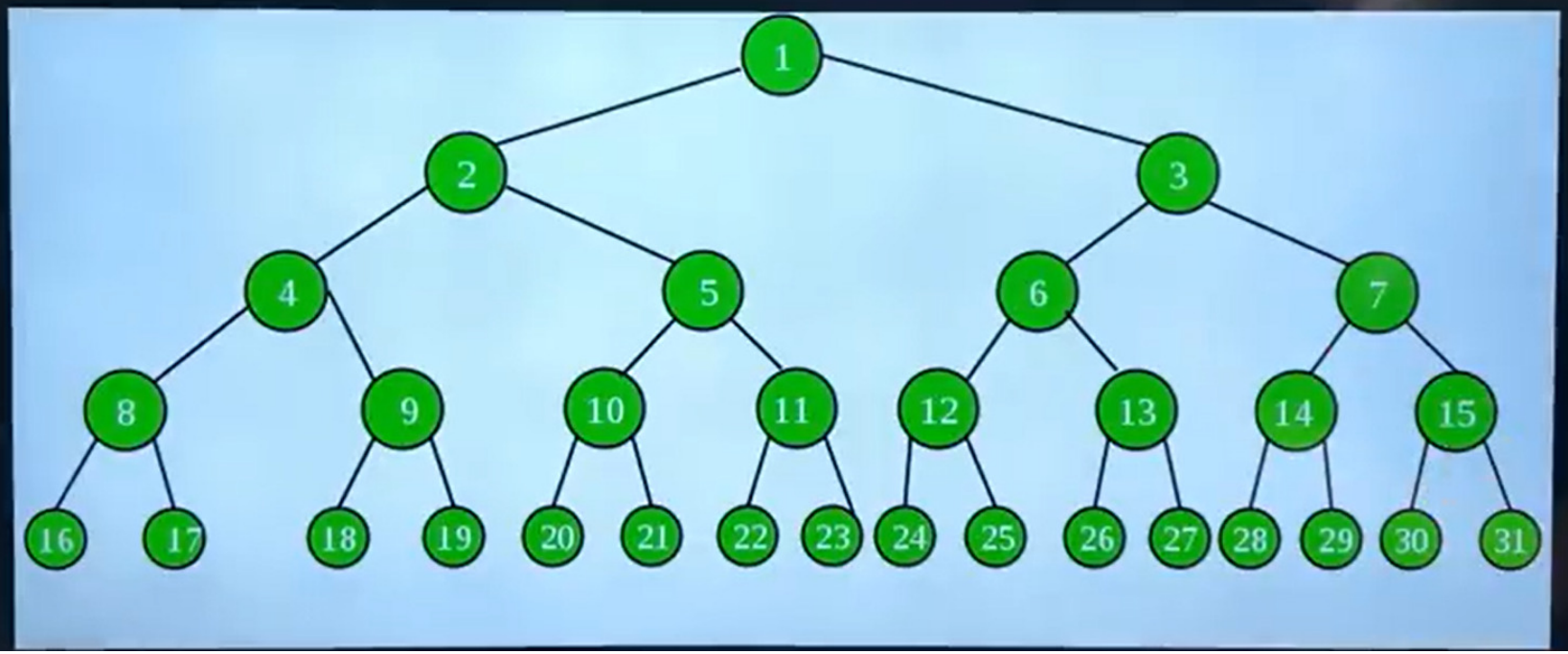
The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithm such as in-order traversal can be very efficient; they are also easy to code

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.

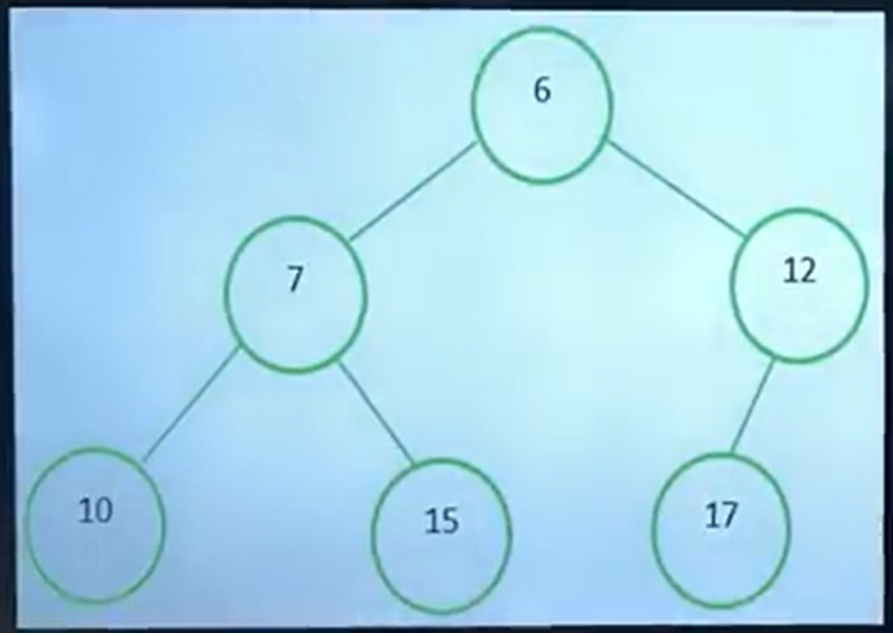


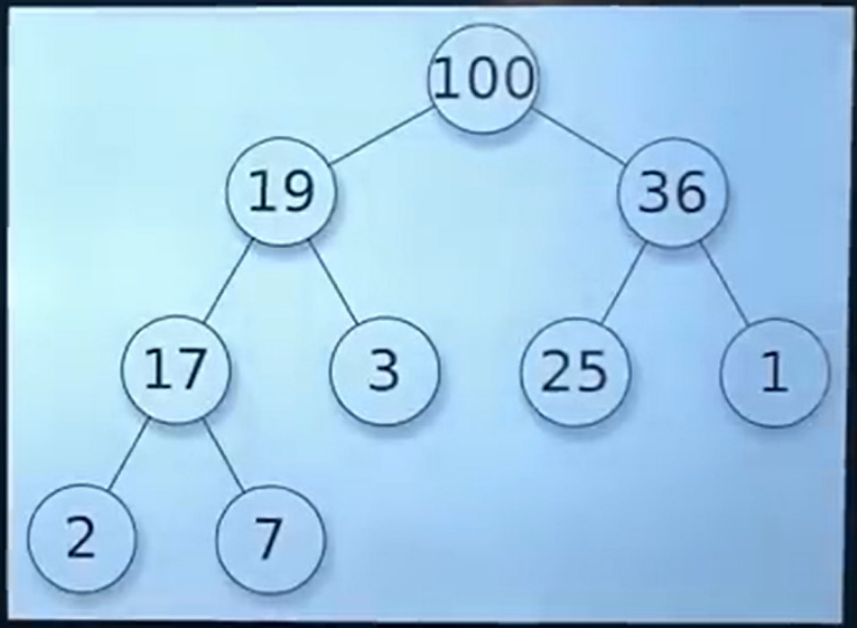
- One can easily determine the children and parent of a node k in any complete tree T
- Specially the left and right children of the node K are 2*k, 2*k + 1 and the parent of k is the node lower bound(k/2)



Heap

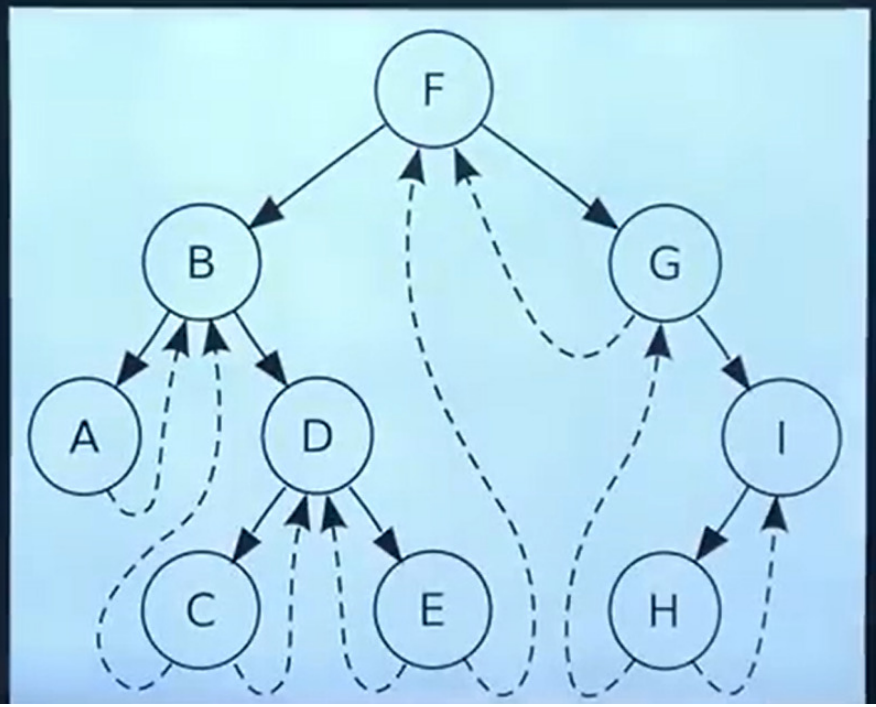
- Suppose H is a complete binary tree with n elements, H is called a Heap, if each node N of H has following properties:
  - The value of N is greater than to the value at each of the children of N then it is called Max heap.
  - A min heap is defined as the value at N is less than the value at any of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



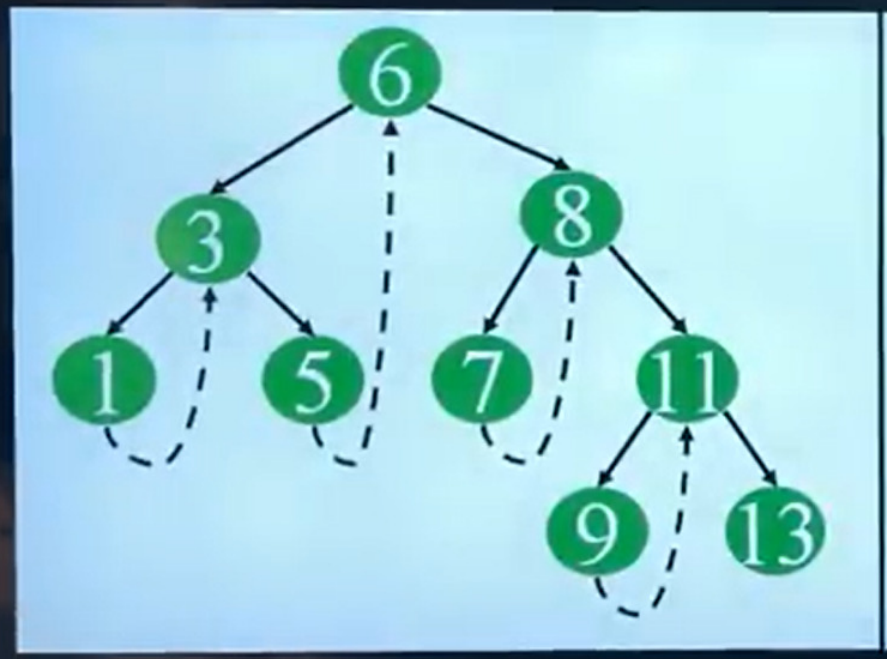


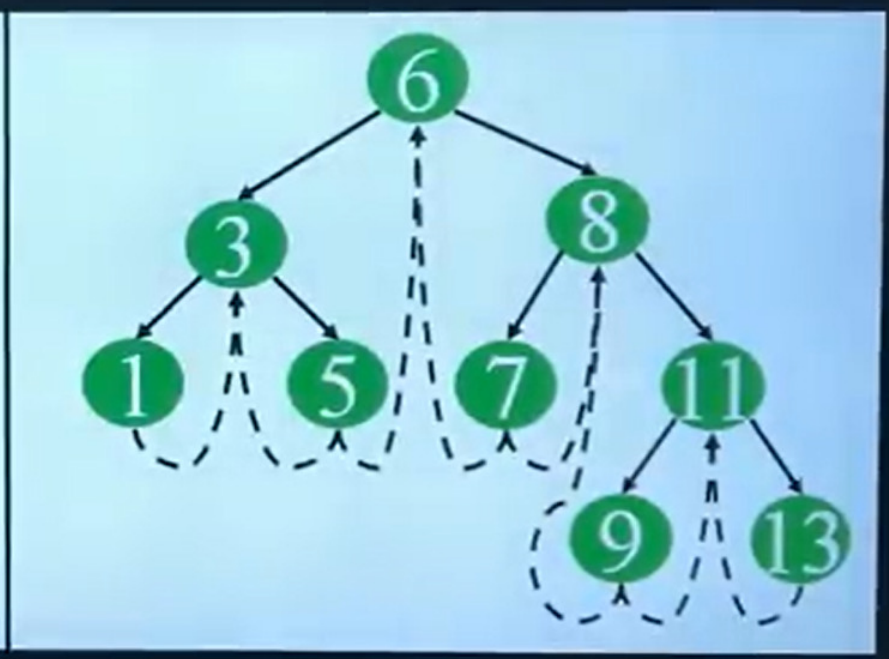
Threaded binary tree

- 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.



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

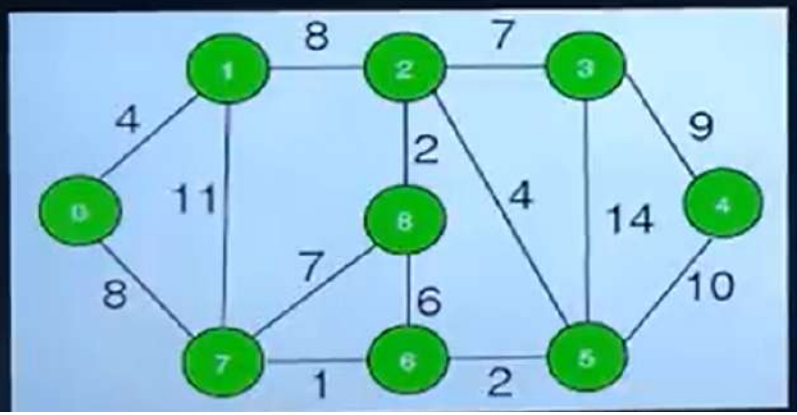


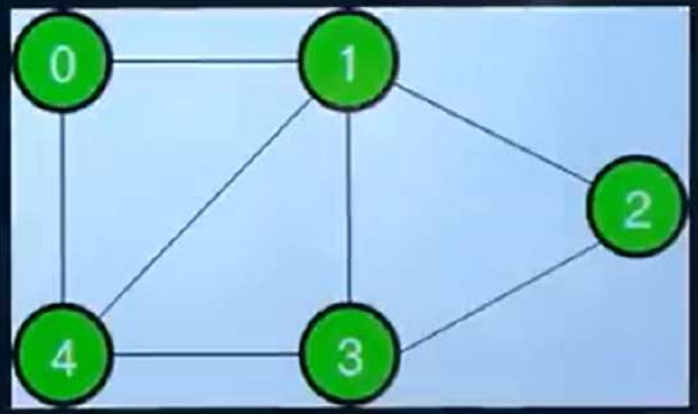


- **Benefits**:
  - Allows stack-less in-order traversal.
  - Makes efficient use of memory by replacing null pointers with threads.
- This tree variant is beneficial when recursive or stack-based traversals aren't feasible. However, its popularity has decreased with newer data structures.

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.





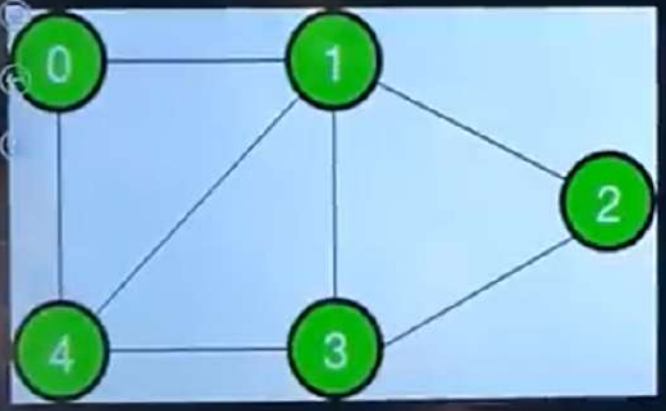
- Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network.
- Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented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

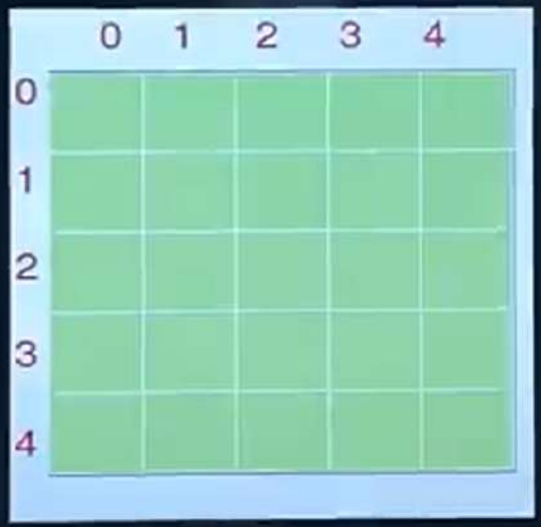
Representation of Graph in Memory

- 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.





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.

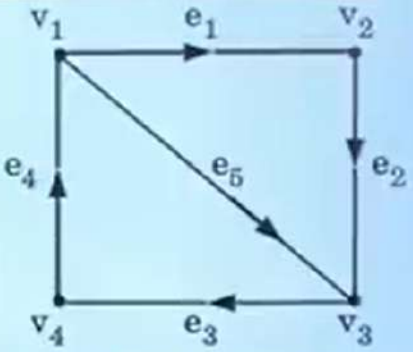
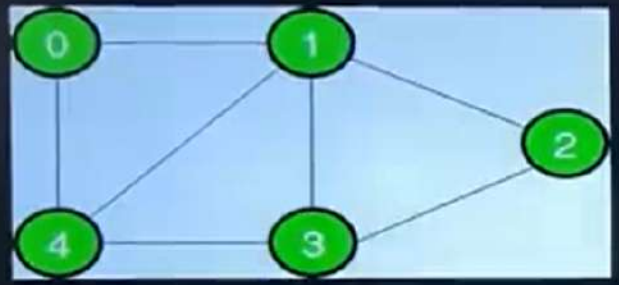


Fig. 4.3.3.

The incidence matrix of the digraph of Fig. 4.3.3 is

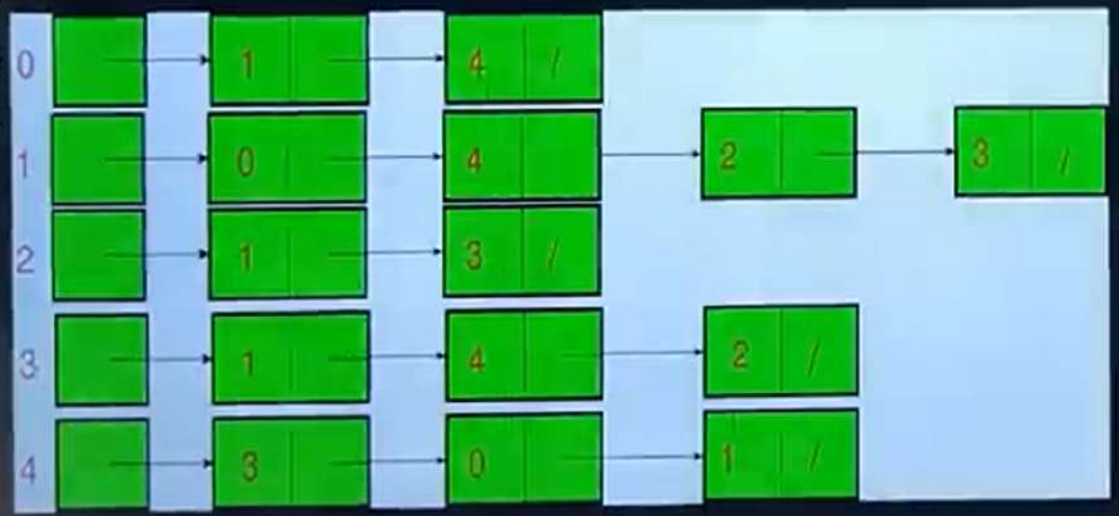
$$I(D) = \begin{bmatrix} 1 & 0 & 0 & -1 & 1 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 & 0 \end{bmatrix}$$

- **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.



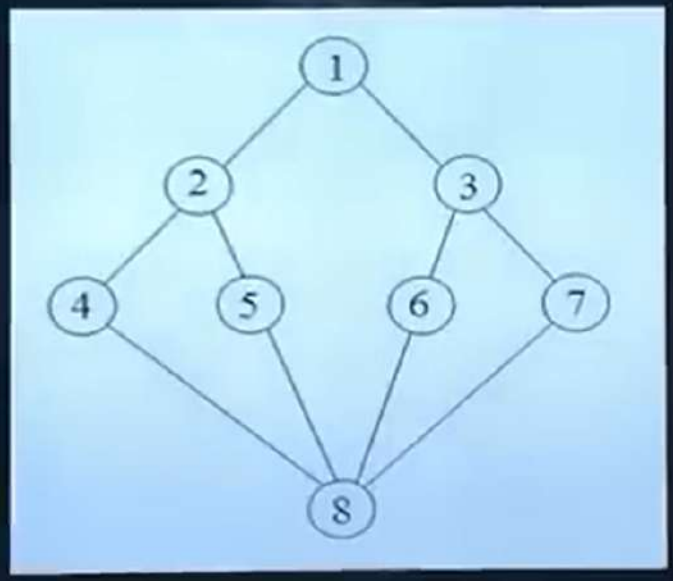
a. For non-weighted graph : | INFO | Adj-list |

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.



- 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.

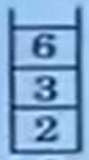
Numerical : Adjacency list of the given graph :

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

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

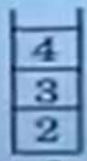
3. Pop 1 from stack, change its STATUS = 1 and
   Push 2, 7 onto stack and change their STATUS = 2; DFS = 1

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

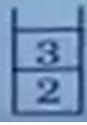


5. Pop 6 from stack, Push 5; DFS = 1, 7, 6

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 ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ 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 ∈ G.Adj[u]            // explore edge (u, v)
5      if v.color == WHITE
6          v.π = 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 $\Theta(V + E)$.

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.

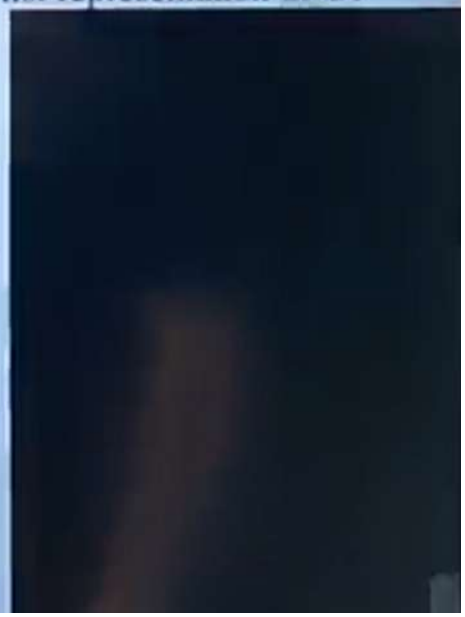
Breadth First Traversal (or Search)

- 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

- 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 ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each v ∈ G.Adj[u]
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = 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.



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.

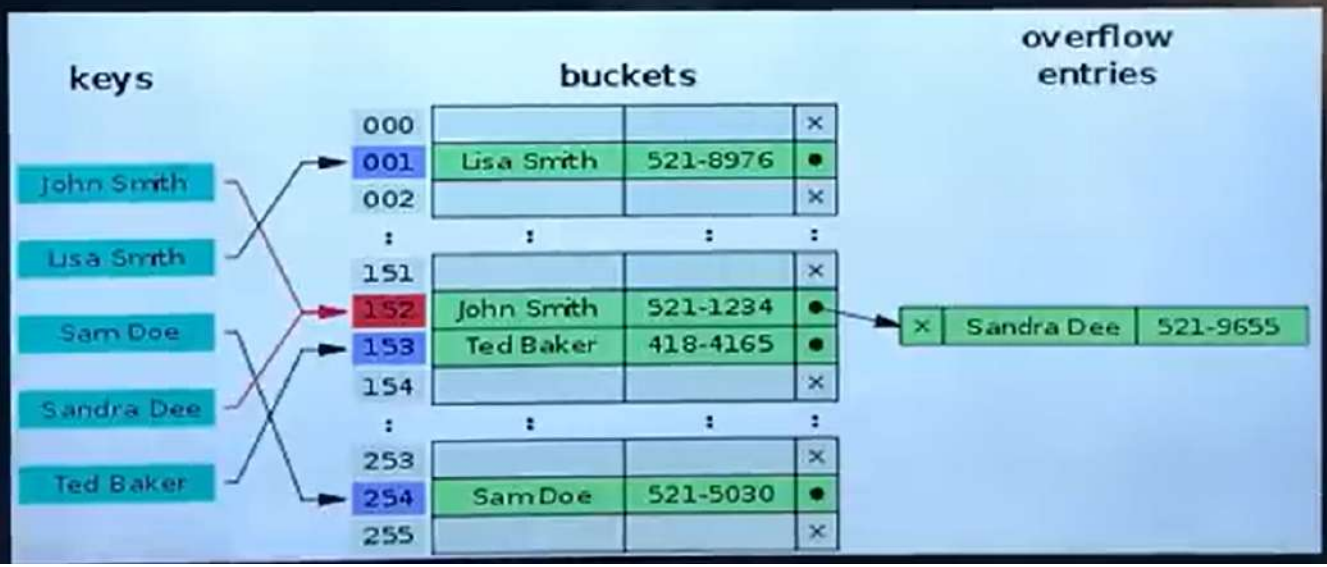
Application of BFS : Breadth first search can be used to solve many problems in graph theory, for example

- 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.



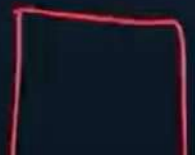
- This conversion called hash function which is from the set of K keys into the set of memory location L.
  - H: K→L
- In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table. An array that stores pointers to records corresponding to our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

- **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.)
  - $H(K) = K(mod\ m)$
  - $H(K) = K(mod\ m) + 1$



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.

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.

Collision Resolution Technique

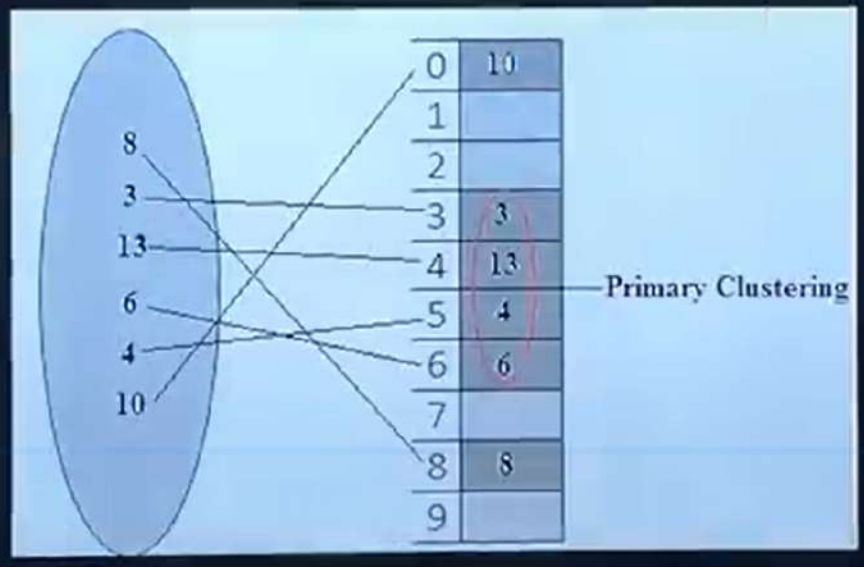
- **Open Addressing/closed hashing** - In Open Addressing, all elements are stored in the hash table itself. i.e. collision is resolved by **probing** or searching through alternate locations in the Hash table itself in a particular *sequence*.

- 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

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.



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

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

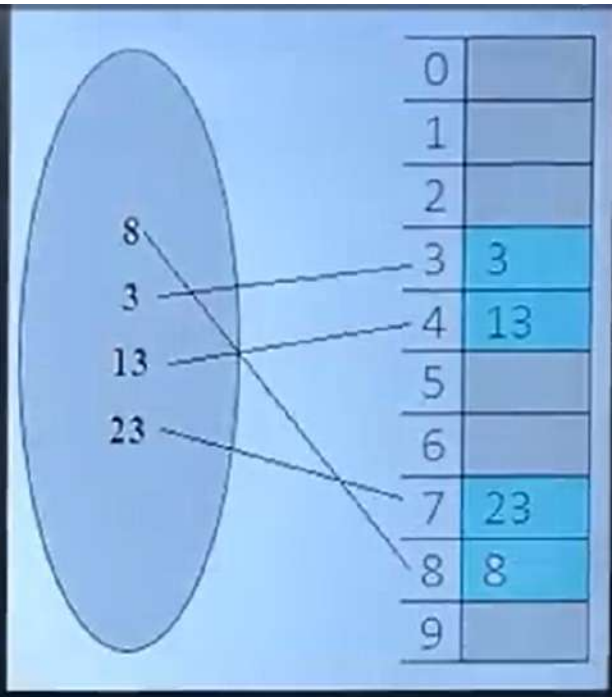
- Where, h' is an auxiliary hash function and i = 0, 1, ..., m - 1.

- **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.

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

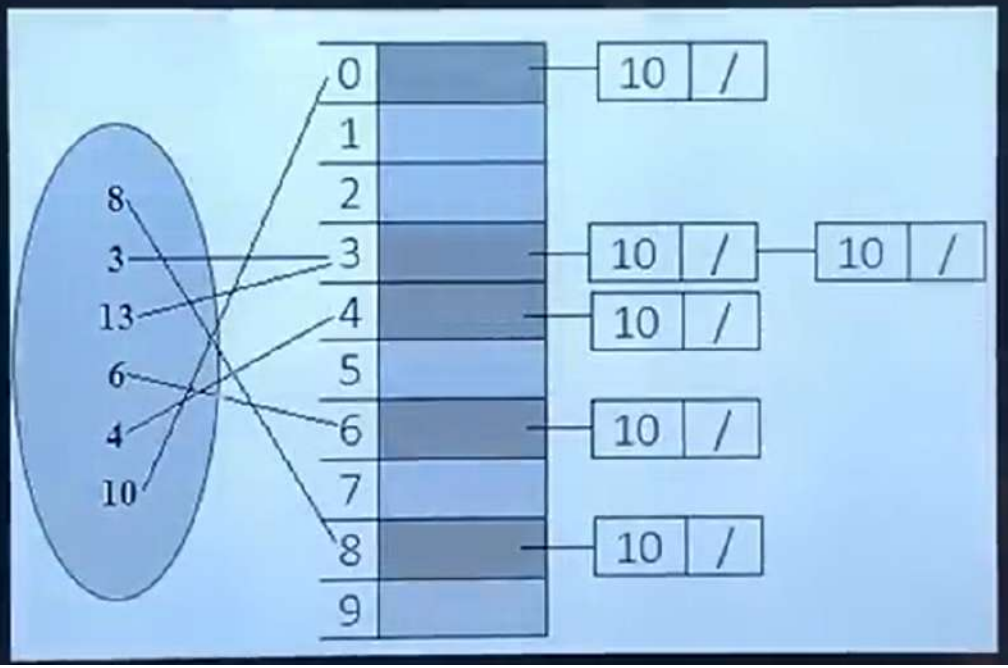
- 8 will be placed at: h (8) = [h (8) + f (0^2)] mod 10 = 8, so it gets placed at location 8.
- 3 will be placed at: h (3) = [h (3) + f (0^2)] mod 10 = 3, no collision, so it gets placed at location 3
- 13 will be placed at: h (13) = [h (13) + f (0^2)] mod 10 = 3, collision occurred, so we increase the 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.

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.



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

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 i^{th} 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