Bridge Course: Basics of C Programming & Data Structure.

Data Structures

Prepared By: Gloriya Mathew, Asst. Professor, Amal Jyothi College of Engineering.

Introduction to Data Structures, Basic operations on different Data Structures:

- Stack
- Queue
- Tree
- Basic Sorting and Searching techniques.

Data Structures:

- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- The idea is to reduce the space and time complexities of different tasks.
- An efficient data structure also uses minimum memory space and execution time to process the structure.
- A data structure is not only used for organizing the data.
- It is also used for processing, retrieving, and storing data.
- There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed.

Need Of Data Structure:

1. Data structure modification is easy.
2. It requires less time.
3. Save storage memory space.
4. Data representation is easy.
5. Easy access to the large database.

- *Data structures provide an easy way of organizing, retrieving, managing, and storing data.*

Classification/Types of Data Structures:

1. Linear Data Structure
2. Non-Linear Data Structure.

-

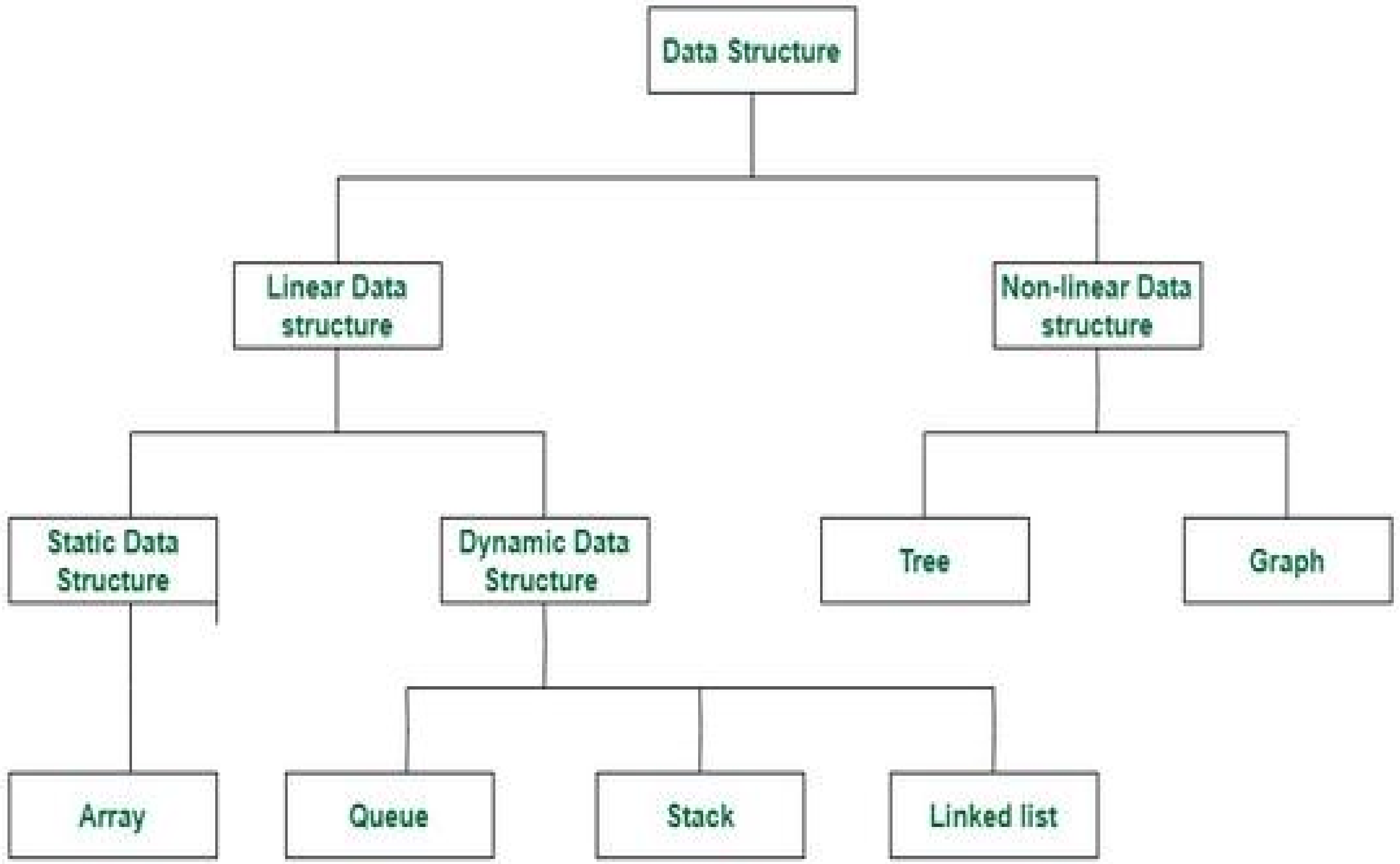
1. Linear Data Structure:

- Elements are arranged in one dimension ,also known as linear dimension.
- Example: lists, stack, queue, etc.

2. Non-Linear Data Structure

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.

Classification of Data Structure

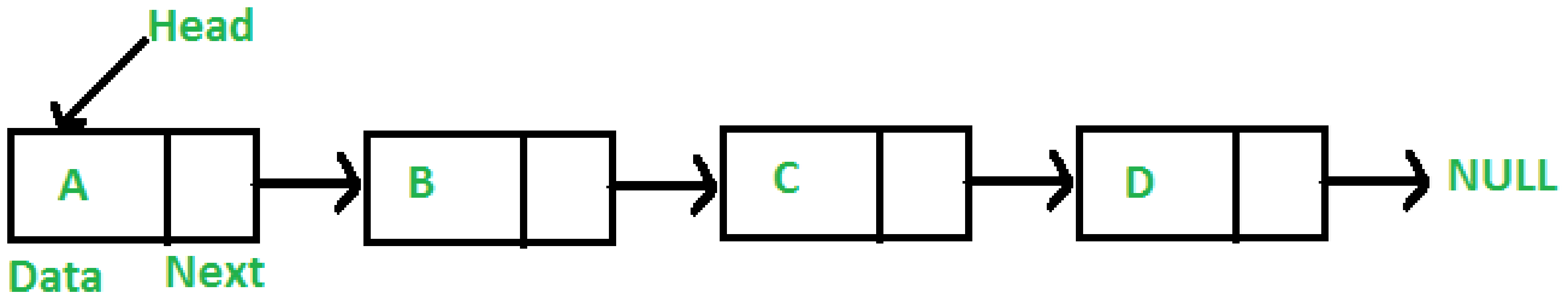


1. Array:

- An array is a collection of data items stored at contiguous memory locations.
- The idea is to store multiple items of the same type together.
- This makes it easier to calculate the position of each element by simply adding an offset to a base value i.e,the memory location of the first element of the array (generally denoted by the name of the array).

2. Linked Lists:

- Like arrays, Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location;
- The elements are linked using pointers.



Stack

Stack

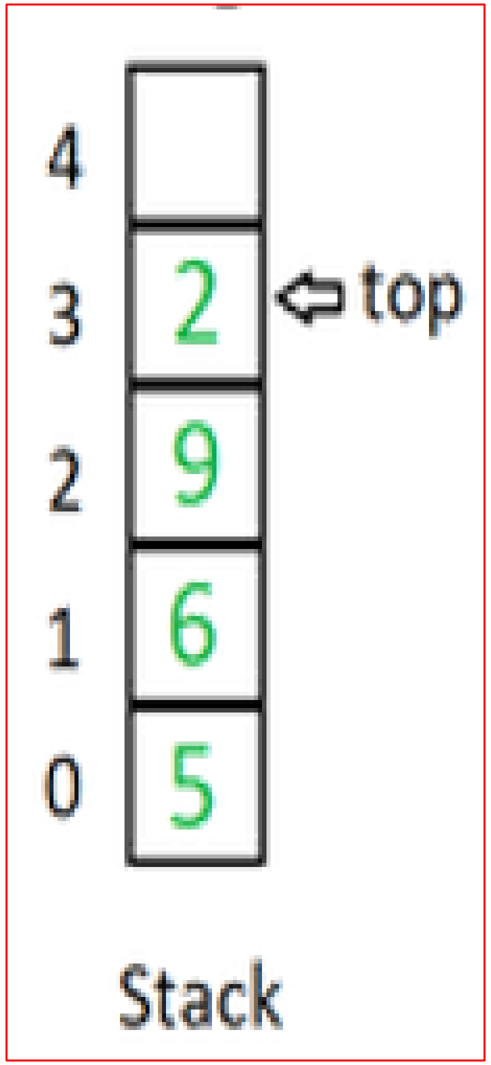
- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack;
- Eg:- a deck of cards or a pile of plates, etc.

Stack

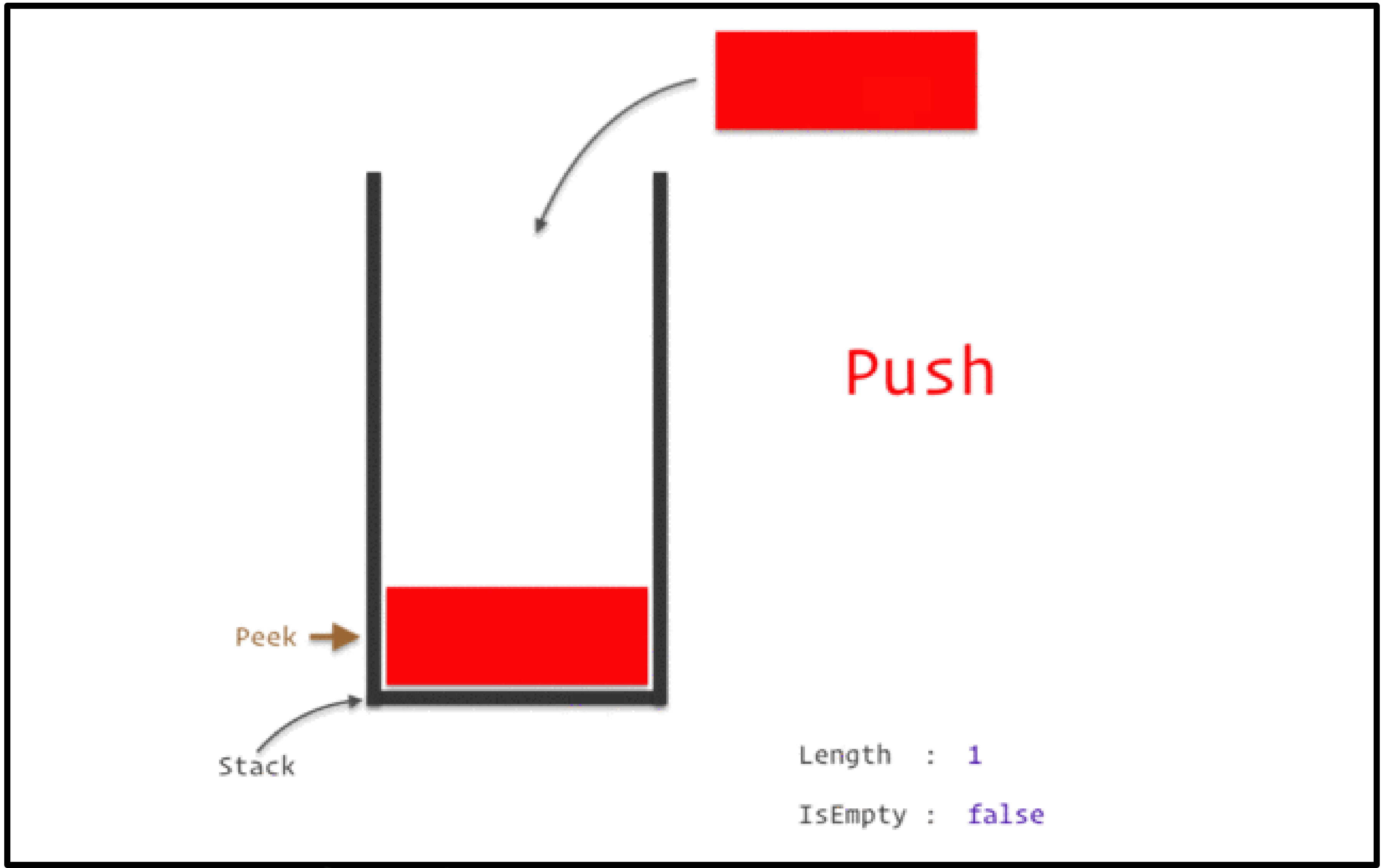
- A real-world stack allows operations at one end only.
- For example, we can place or remove a card or plate from the top of the stack only.
- Likewise, Stack ADT allows all data operations at one end only.
- At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure.
- LIFO stands for Last-in-first-out.
- Here, the element which is placed (inserted or added) last, is accessed first.
- In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

3. Stack:

- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).
- In stack, all insertion and deletion are permitted at only one end of the list.



Stack:



Stack - Basic operations :

1. **Initialize:** Make a stack empty.
2. **Push:** Adds an item in the stack.
   - If the stack is full, then it is said to be an Overflow condition.
3. **Pop:** Removes an item from the stack.
   - The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
4. **Peek** or **Top:** Returns top element of the stack.
5. **isEmpty**: Returns true → stack is empty; else false.
6. **isFull()** – check if stack is full.

Push Operation

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

Step 1 − Checks if the stack is full.

Step 2 − If the stack is full, produces an error and exit.

Step 3 − If the stack is not full, increments **top** to point to next empty space.

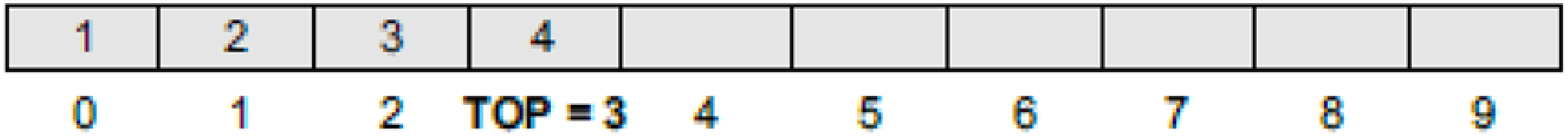
Step 4 − Adds data element to the stack location, where top is pointing.

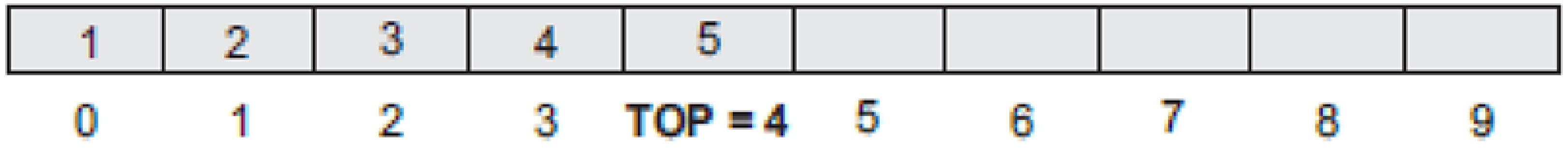
Step 5 − Returns success.

Algorithm-Push Operation-Array

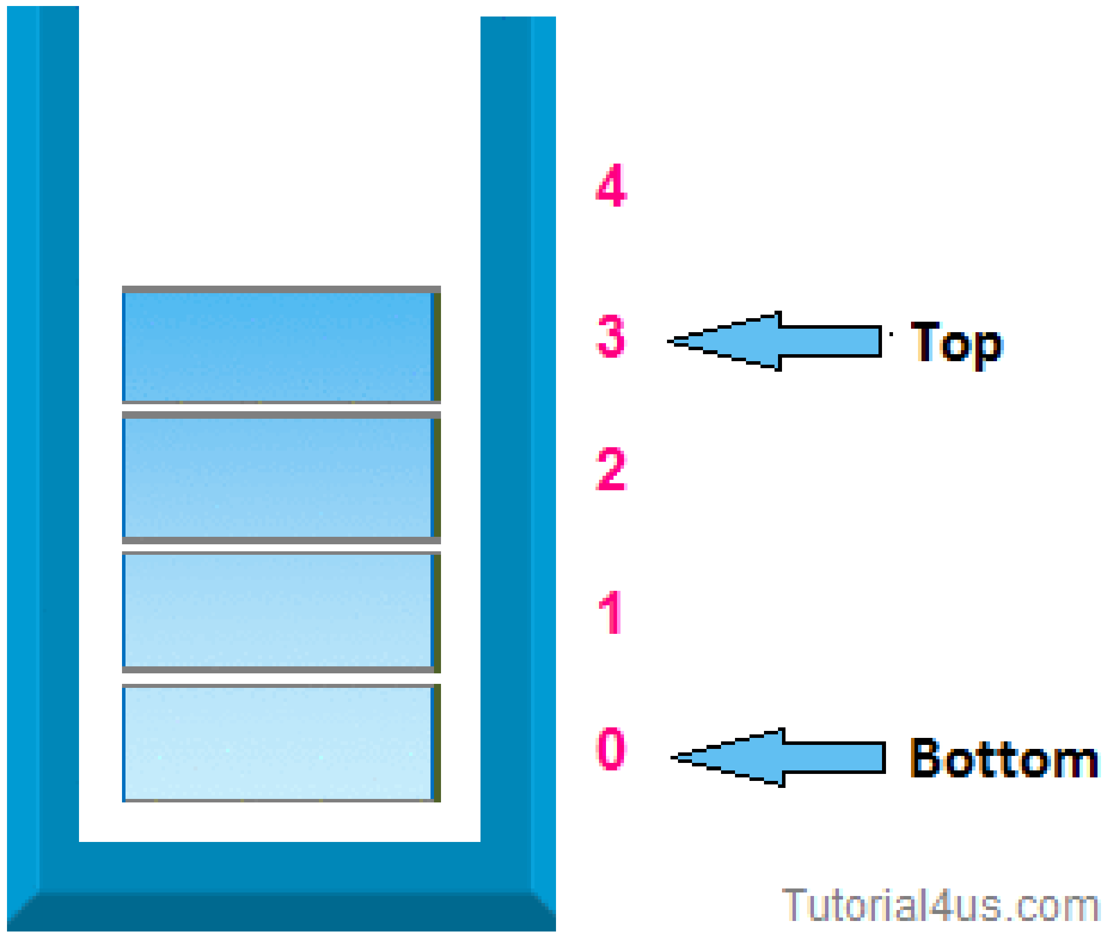
```
Step 1: IF TOP = MAX-1
                PRINT “OVERFLOW”
                Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP+1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Stack: Push Operation- Array





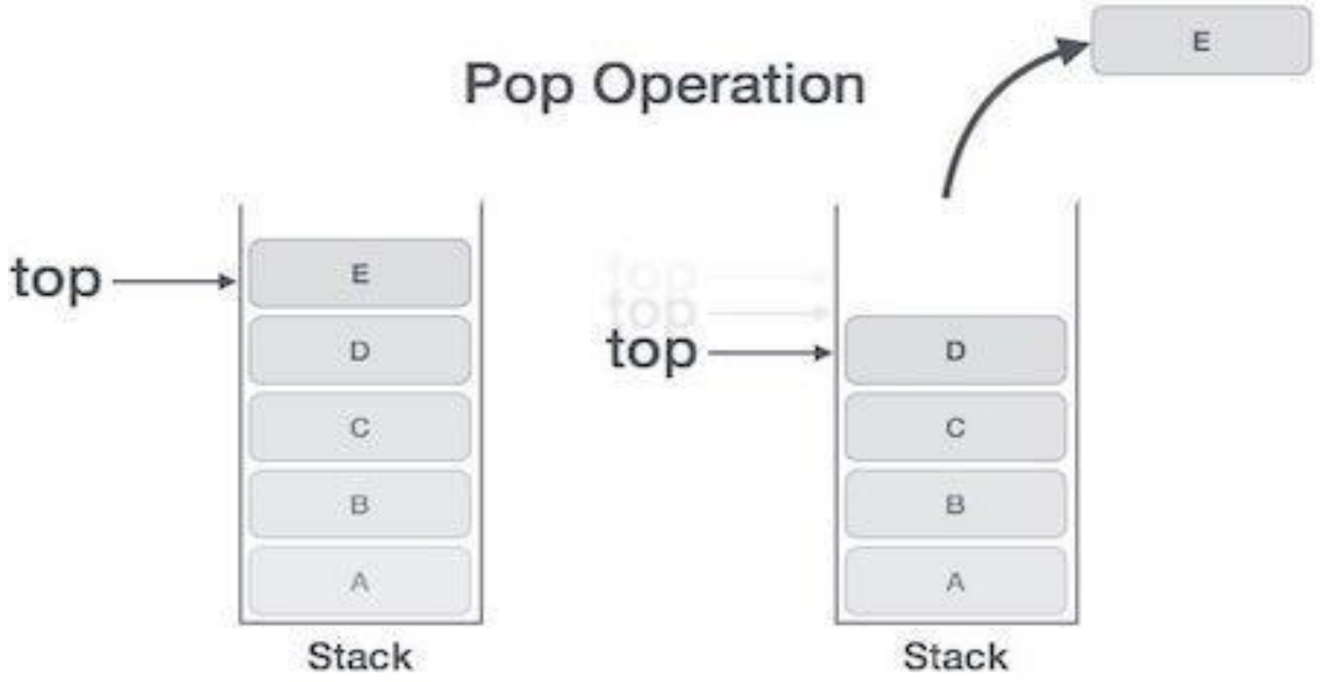
Stack:



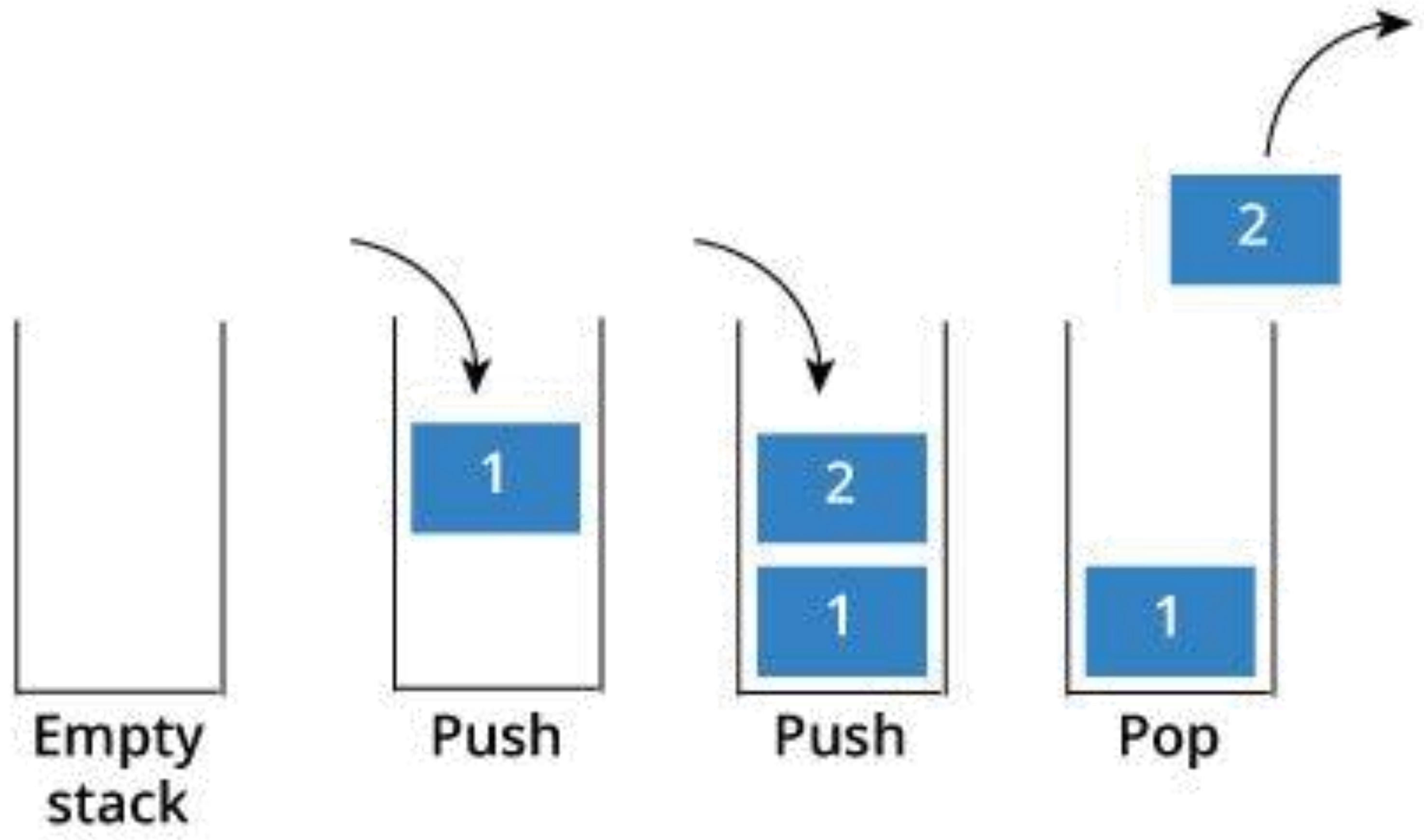
Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.
- But in linked-list implementation, pop() actually removes data element and de allocates memory space.

Stack



Stack:



Stack Implementation

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

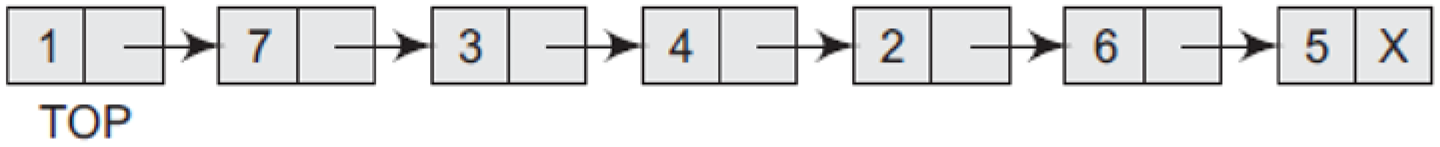
Linked Representation Of Stack:

Stack Implementation using Linkd List:

- The drawback of array representation is that the array must be declared to have some fixed size
- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- if the array size cannot be determined in advance, then the other alternative linked representation, is used.

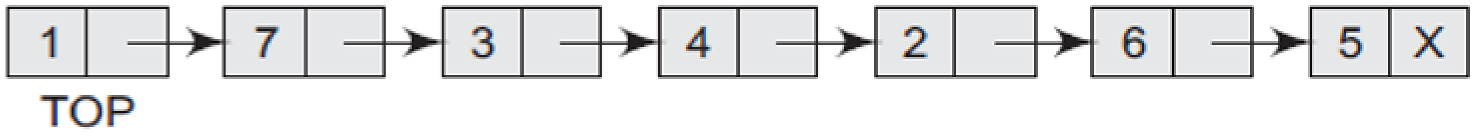
Linked Representation Of Stacks

- In a linked stack, every node has two parts:—
  - *i. one that stores data*
  - *ii. two that stores the address of the next node.*
- The START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the node pointed by TOP.
- If TOP = NULL, then it indicates that the stack is empty



Operations On A Linked Stack – Push()

- A linked stack supports all the three stack operations, that is, push, pop, and peek.
- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack



Steps:

1) To insert an element, we first check if **TOP=NULL**. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP.
2) If **TOP!=NULL,** then we insert the new node at the beginning of the linked stack and name this new node as TOP

Push Operation –Linked Stack

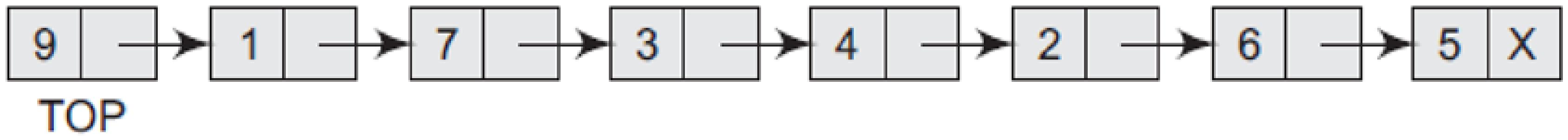
- In Step 1, memory is allocated for the new node.
- In Step 2, the DATA part of the new node is initialized with the value to be stored in the node.
- In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL.
- In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP.
- However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

Algorithm –Push Operation –Linked Stack

```
Step 1: Allocate memory for the new node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
Step 4: END
```

Pop Operation

- The pop operation is used to delete the topmost element from a stack.
- However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed
- In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack.

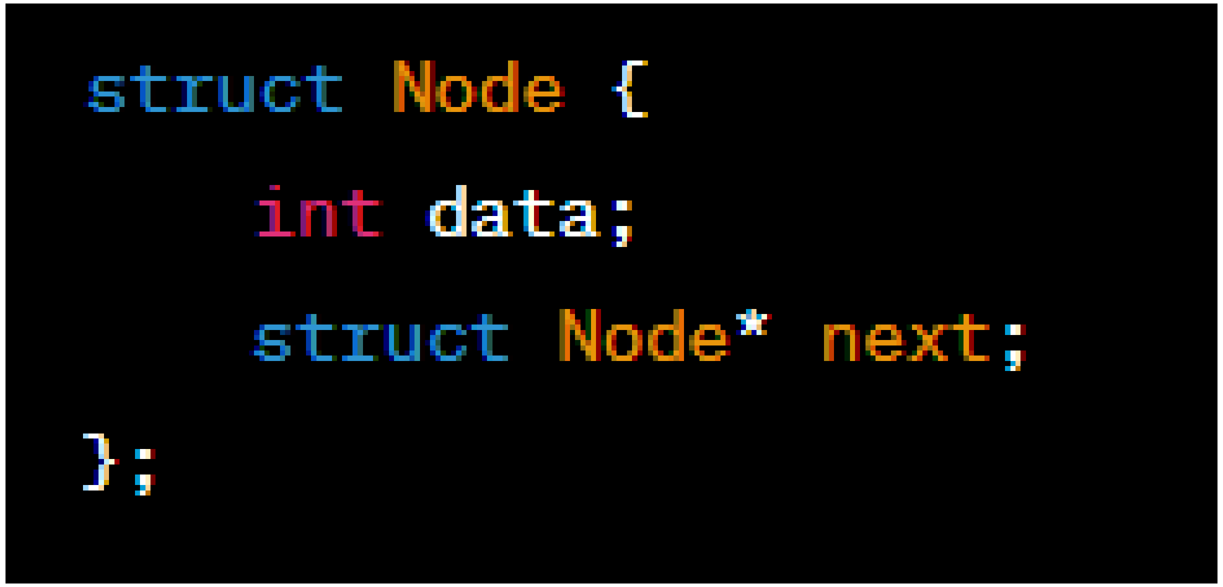


Algorithm –Pop Operation –Linked Stack

```
Step 1: IF TOP = NULL
                PRINT "UNDERFLOW"
                Goto Step 5
        [END OF IF]
Step 2: SET TEMP = TOP
Step 3: SET  TOP=TEMP-> NEXT
Step 4: FREE TEMP
Step 5: END
```

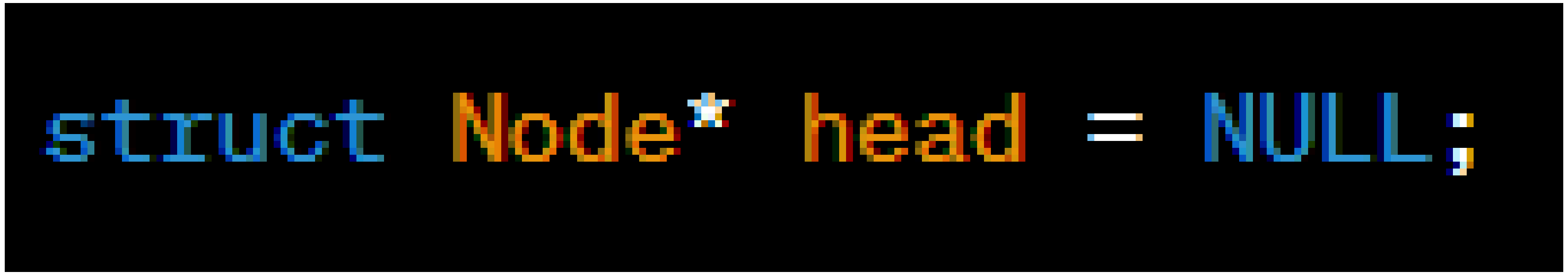
Implement A Stack Using A Linked List

- Define a structure for a node to hold the data and a pointer to the next node.
- This structure(Node) represents the elements of the stack.



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

2. Create a pointer to the top of the stack, often called "head."



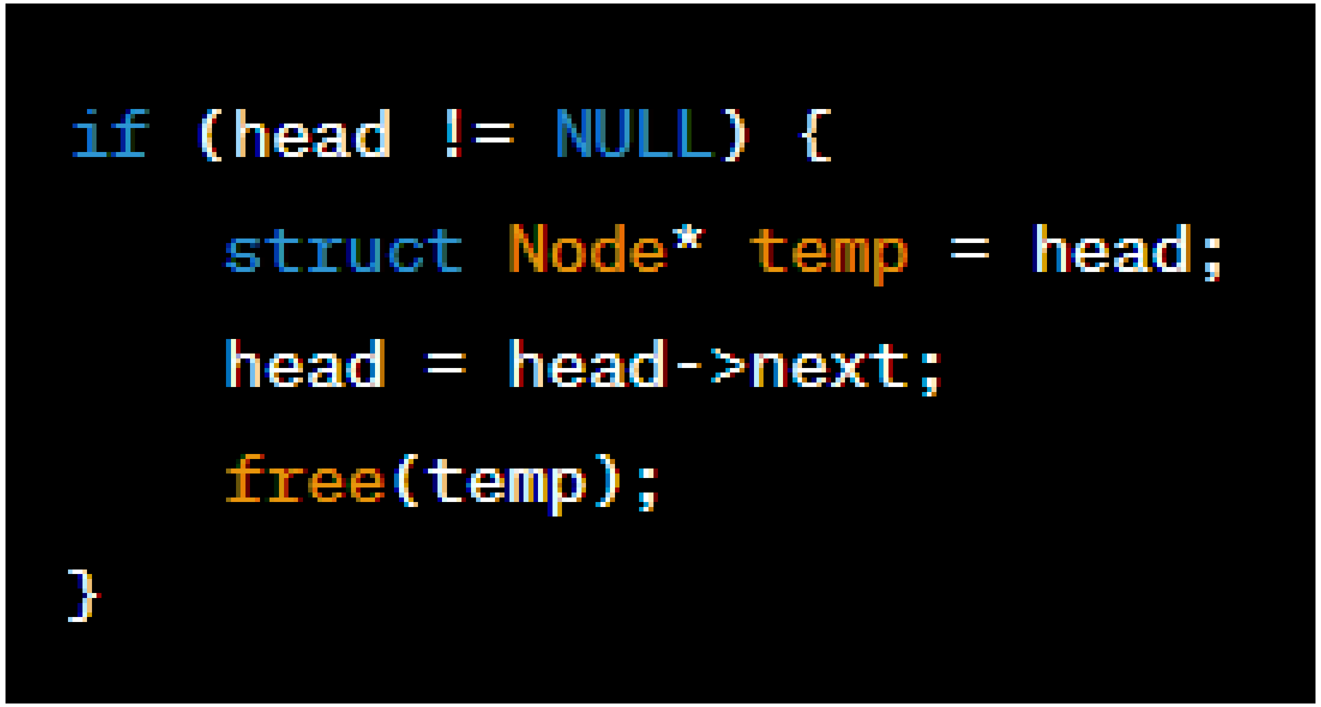
```
struct Node* head = NULL;
```

3. Implement the stack operations using linked list:

- **Push Operation:** To add an element to the stack, create a new node, set its data, and update the pointers.

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = head;
head = newNode;
```

- **Pop Operation:** To remove an element from the stack, update the "head" pointer.



```
if (head != NULL) {
    struct Node* temp = head;
    head = head->next;
    free(temp);
}
```

Queue.

Queue:

- Queue is also an Abstract Data Type (ADT) or a linear data structure.
- IT follows a particular order in which the operations are performed.
- The order is First In First Out (FIFO).
- In the queue, items are inserted at one end and deleted from the other end.
- Eg:- any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in removing.
- In a stack we remove the item the most recently added;
- In a queue, we remove the item the least recently added.

Queue:

Enqueue:

- process to Add an element into queue is called Enqueue.

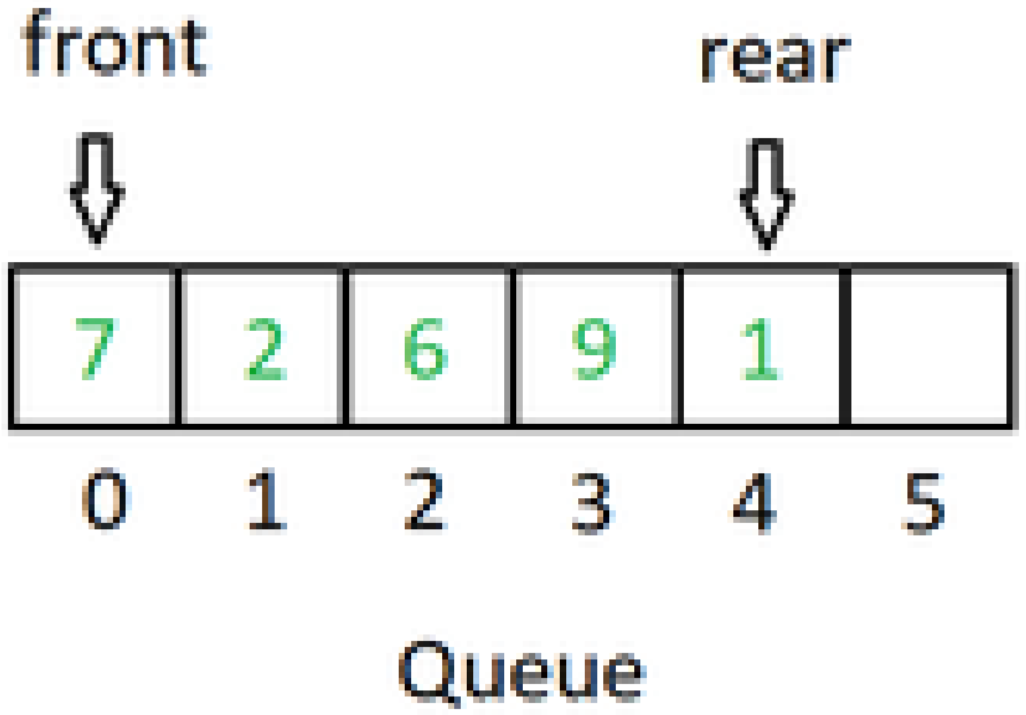
Dequeue:

- process of Removal of an element from queue.

Applications of Queue:

- Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios :
- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive

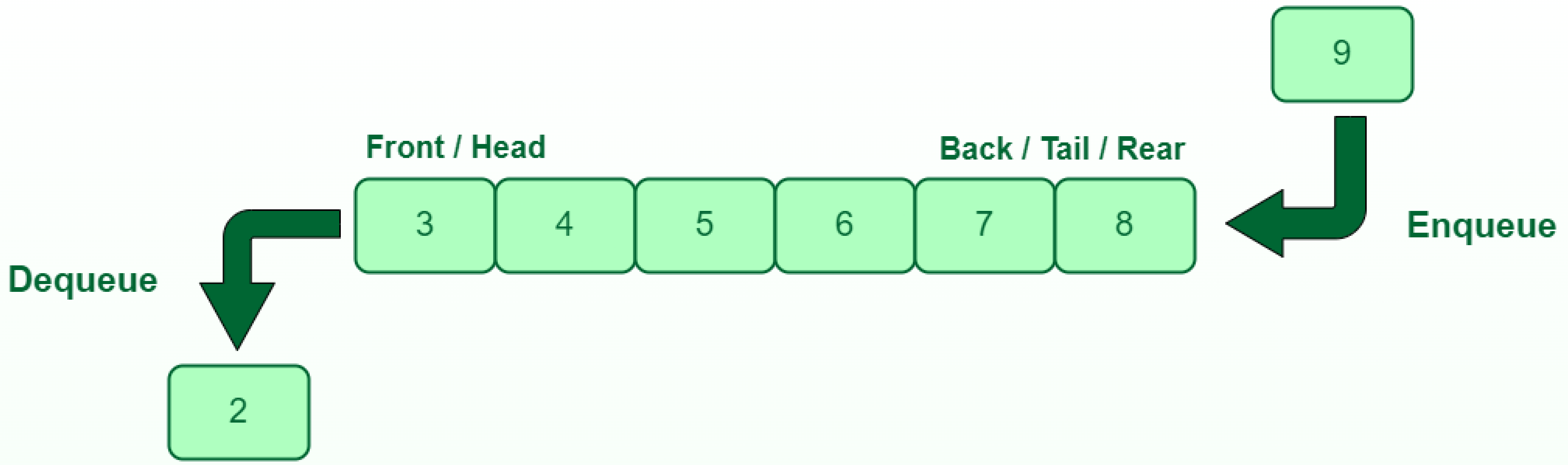
Queue:



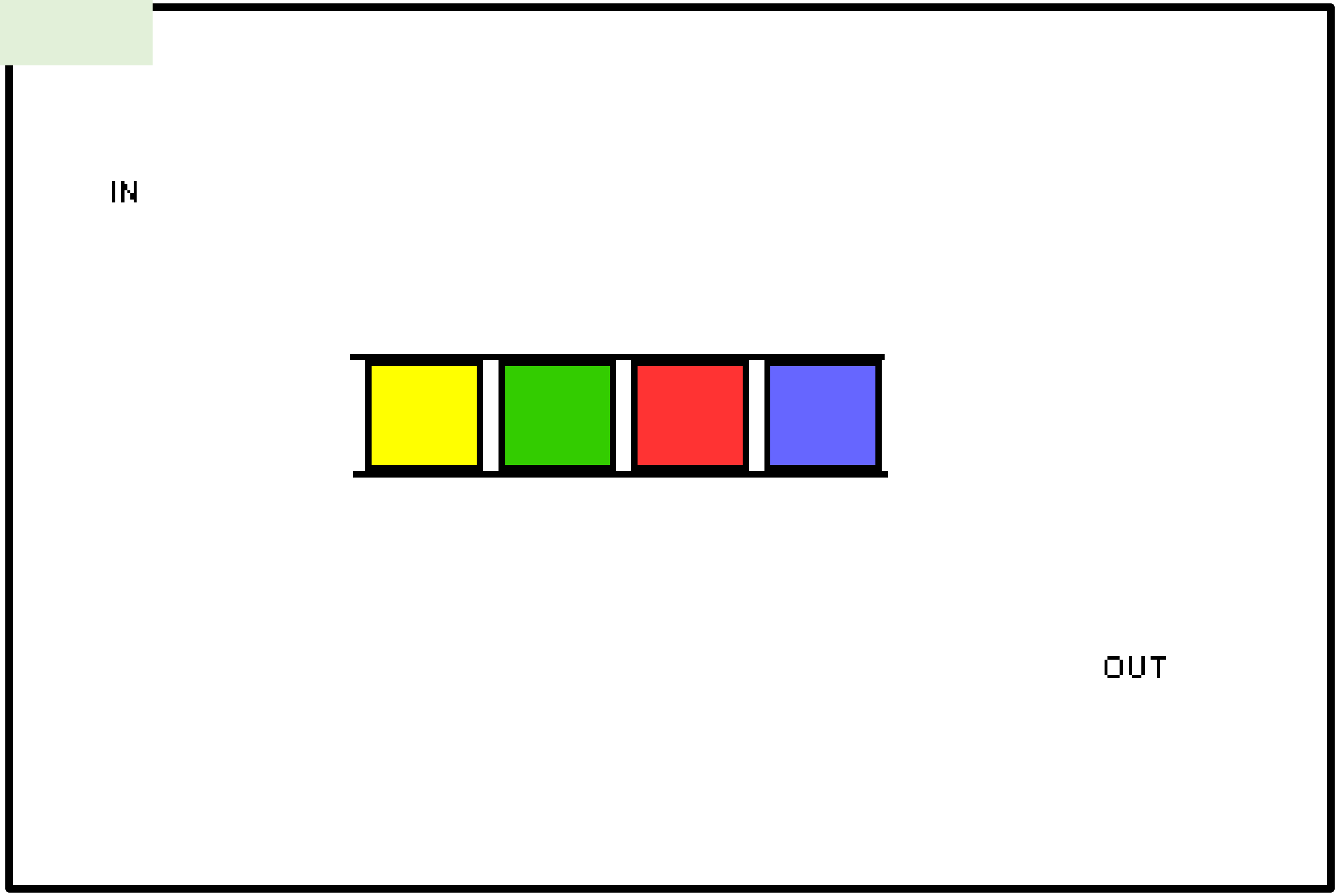
Queue - Basic Operations :

1) **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
2) **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
3) **Front:** Get the front item from the queue.
4) **Rear:** Get the last item from the queue.

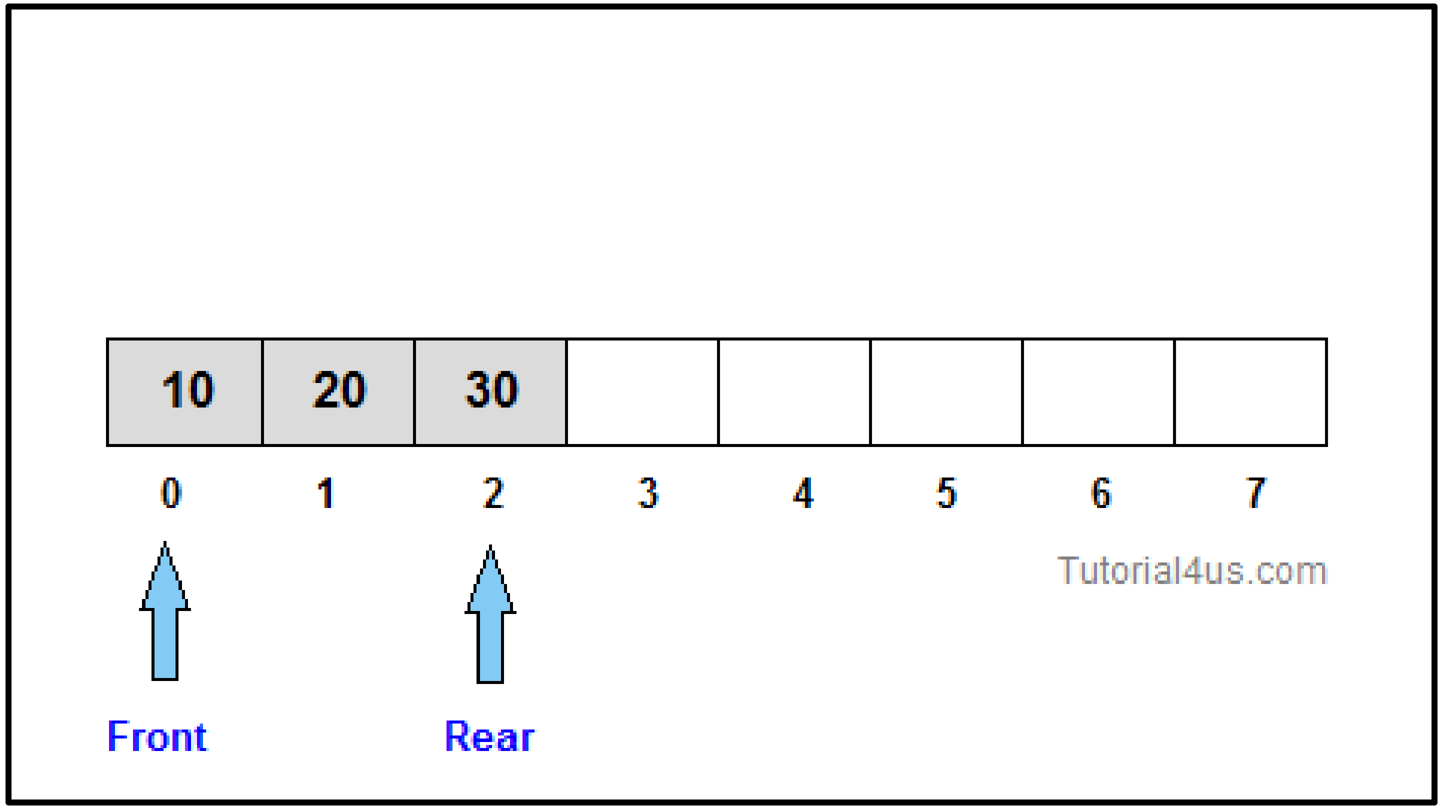
Queue:



Queue:



Queue:



Characteristics of Queue:

- Queue can **handle multiple data.**
- We can **access both ends.**
- They are **fast** and **flexible.**

Queue Representation:

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

➢ **Queue:** the name of the array storing queue elements.

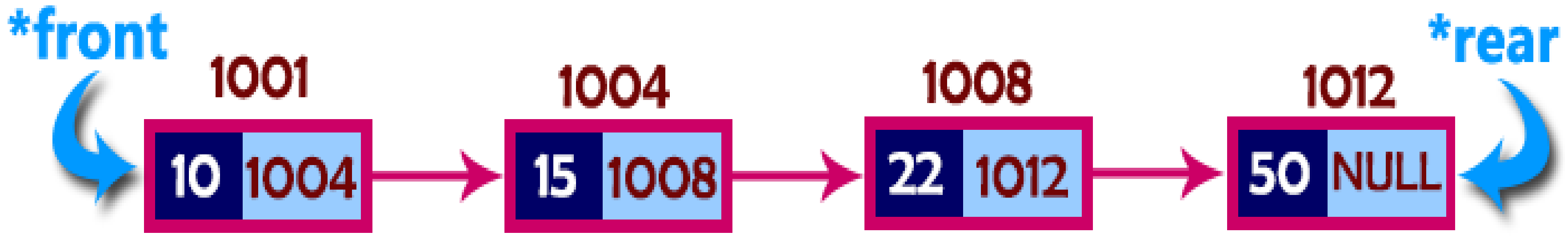
➢ **Front:** the index where the first element is stored in the array representing the queue.

➢ **Rear:** the index where the last element is stored in an array representing the queue.

Implementation of Queue

- using **Arrays**
- using **Linked List**

Linked list implementation of a QUEUE



- The last inserted node is always pointed by 'rear'
- The first node is always pointed by 'front'.

Operations on queue

- **Insertion- ENQUEUE**
- **Deletion- DEQUEUE**
- **Queue can be implemented using an *Array or Linked List.***

Implementation of QUEUE: Using Linked list

To implement queue using linked list, set the following things before implementing actual operations.

Step 1 - Include all the header files which are used in the program. And declare all the user defined functions.

Step 2 - Define a 'Node' structure with two members data and next.

Step 3 - Define two Node pointers 'front' and 'rear' and set both to NULL.

Step 4 - Implement the main method by displaying Menu of list of operations and make suitable function calls in the main method to perform user selected operation.

Implementation of QUEUE: Using Linked list

```
#include<stdio.h>
#include<conio.h>

struct Node
{
    int data;
    struct Node *next;
} *front = NULL,*rear = NULL;
void enQueue(int);
void deQueue();
void display();
```

enQueue(value) :- Inserting an element into the Queue

Step 1 - **Create a newNode** & set **newNode → data=value** and
Set **newNode → next= NULL.**

Step 2 - Check whether queue is Empty (**rear == NULL**)

Step 3 - If it is **Empty** then, set **front = newNode** and **rear = newNode**.

Step 4 - If it is **Not Empty** then, set **rear → next = newNode** and
rear = newNode.

deQueue() - Deleting an Element from Queue

Step 1 - Check whether queue is Empty (**front == NULL**).

Step 2 - If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function

Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set **temp=front**

Step 4 - Then set **front = front → next** and **delete** '**temp**', ie **(free(temp))**

display() - Displaying the elements of Queue

Step 1 - Check whether queue is **Empty (front == NULL)**.

Step 2 - If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.

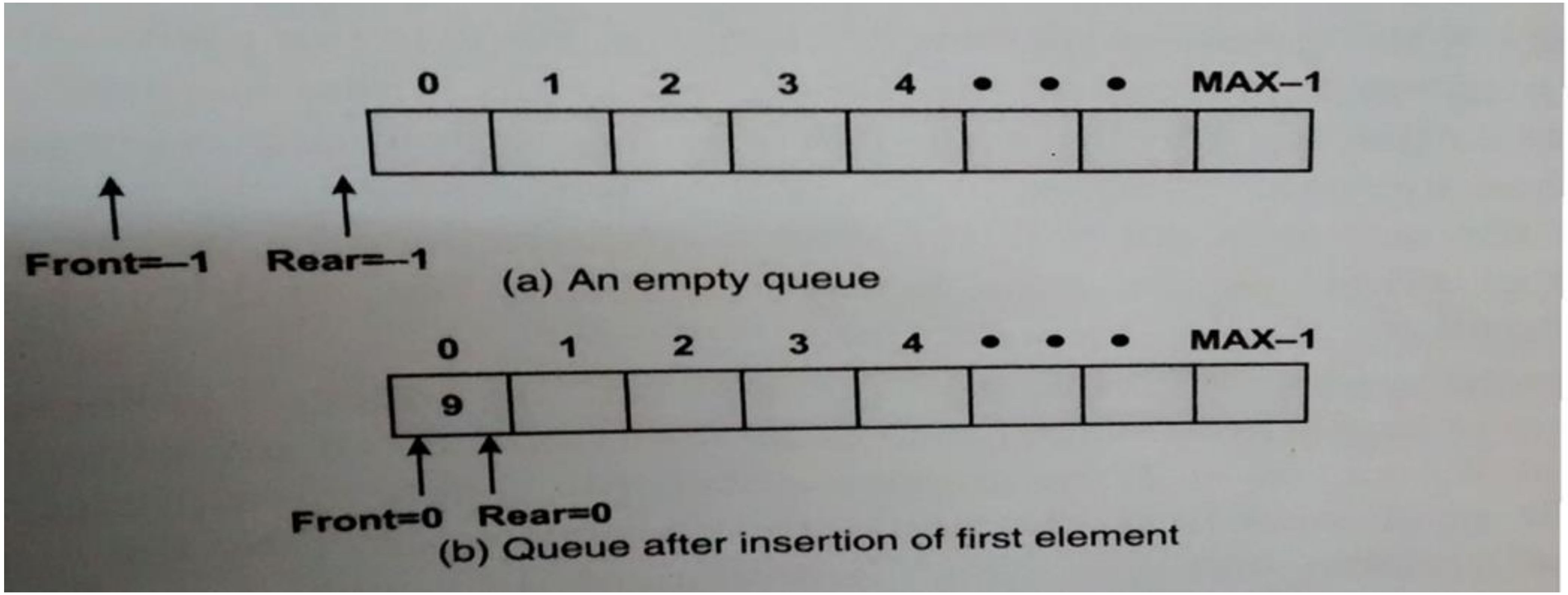
Step 4 - Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).

Step 5 - Finally! Display '**temp → data** ---> **NULL**'.

Implementation of Queue using Array

- Initially the FRONT and the REAR of the queue points to -1 to indicate an empty queue.
- Before we insert a new element, test overflow condition(Queue full)
- If queue is full, then REAR= MAX-1; (MAX → maximum size of the array)
- If the Queue is not full, insertion can be performed.
- To insert an element, increment the REAR pointer, and element is inserted at that position.
- As we add elements to the queue, the REAR keeps on moving ahead, always pointing to the position where the next element will be inserted, while the FRONT remains at the first index.
- Before deleting an element , check underflow condition.
- i.e, If queue is empty , then value of FRONT is -1

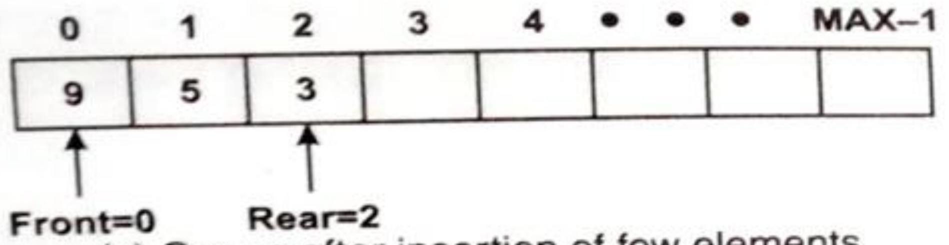
Implementation of Queue using Array



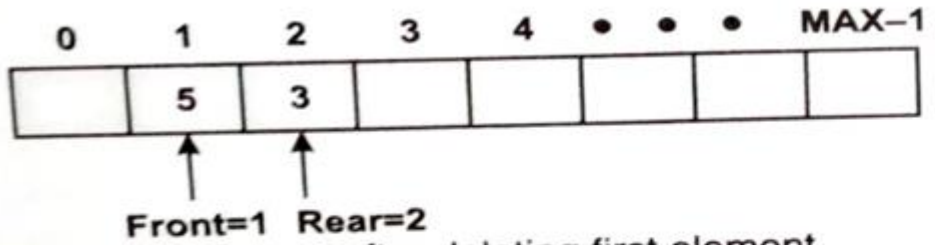
(a) An empty queue

(b) Queue after insertion of first element

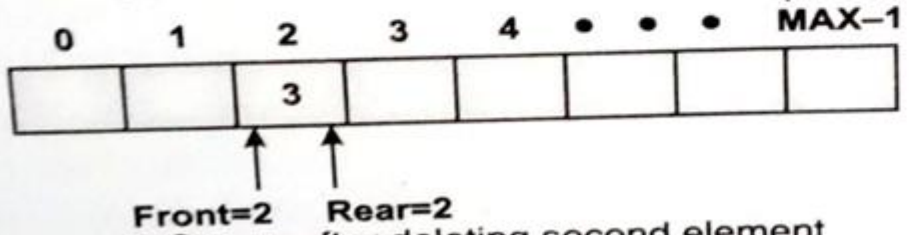
Implementation of Queue using Array



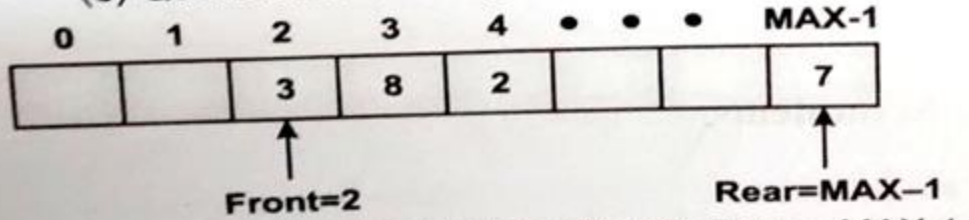
(c) Queue after insertion of few elements



(d) Queue after deleting first element



(e) Queue after deleting second element



(f) Queue having vacant space though Rear = MAX-1

Fig. 6.2 *Various States of Queue after Insert and Delete Operations*

Queue using Array: - Enqueue operation

```
Algorithm enQueue(queue, front, rear)
1.  if queue is full  if(rear == SIZE-1)
    ▪ then
    ▪ print "Queue is Full! Insertion is not  possible!!!";
2. if Queue is empty if(front == -1)
    ▪ front = 0;
    ▪ rear= rear+1;
3. queue[rear] = value;
4. Print "Enqueued the Element into the queue!!";
```

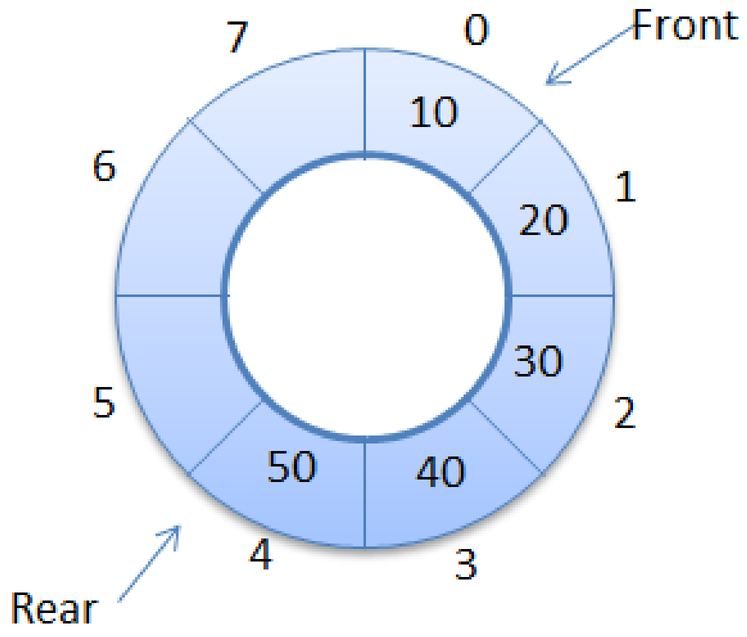
Queue using Array:- Dequeue operation

- Algorithm *deQueue*(queue, front, rear)

```
1.  if queue is empty if(front = = -1)    //check if queue is empty
        2.  print " Queue is Empty!!!!";
3.  Set val=queue[front];
4.  if (front == rear)  // there is only one element in queue
5.              front = rear = -1;
6.  Else    front= front+1;
7.  Return val;
```

Circular Queue

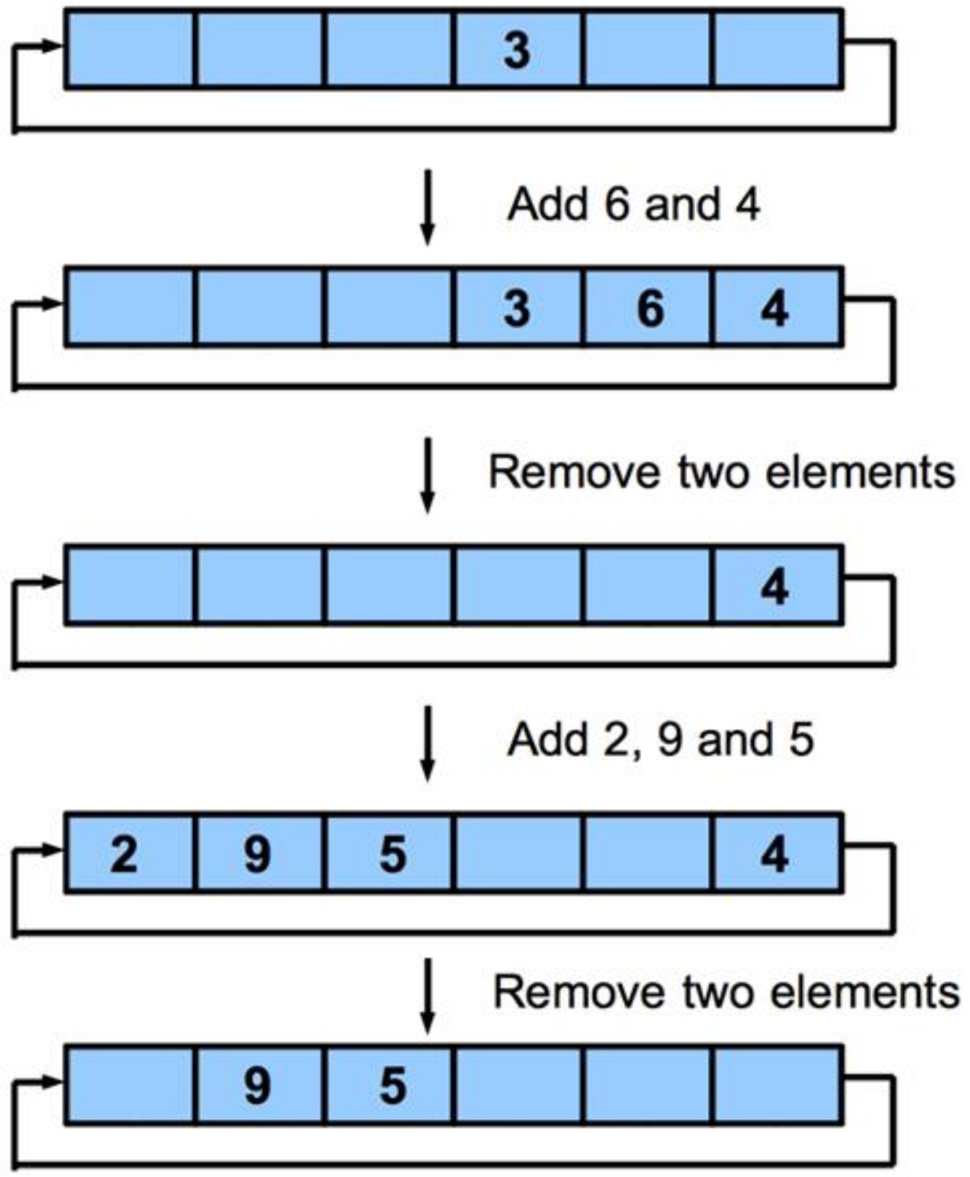
- Circular Queue is a linear data structure;
- operations are performed based on FIFO (First In First Out) principle and
- Last position is connected back to the first position to make a circle.
- It is also called 'Ring Buffer'.



Circular Queue

Drawback of Linear Queue:

- In a normal Queue, we can insert elements until queue becomes full.
- Once the queue is full, even though few elements from the front are deleted and
- some occupied **space is relieved**, it is **not possible to add** anymore new elements, as the **rear has already reached** the **Queue's rear most position.**



Circular Queue

- This queue is not linear but circular.
- Its structure can be like the following figure:
- In circular queue, once the Queue is full the "First" element of the Queue becomes the "Rear" most element, if and only if the "Front" has moved forward.
- otherwise it will again be a "Queue overflow" state.

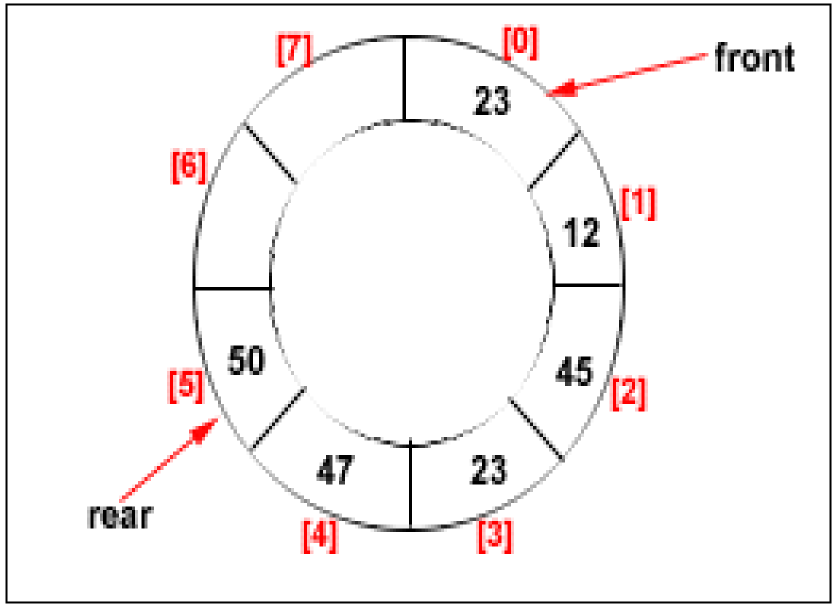


Figure: Circular Queue having Rear = 5 and Front = 0

Circular Queue Implementation

- The **FRONT** pointer points to the **first element** of the queue.
- The **REAR** pointer points to the **last element** of the queue.
- *Initially, **FRONT** and **REAR** are **set to -1.***

Circular Queue Implementation:

Queue empty condition:

- `FRONT == -1`

Queue full condition:

1. `FRONT == 0 and REAR == (SIZE - 1)`
2. `FRONT == (REAR + 1)`

These two conditions can be merged into single using modulo operation

3. `FRONT == (REAR+1) % SIZE`

Circular Queue- Enqueue Algorithm:

Step 1:

Check whether **queue** is **FULL**.

if(**front==(rear+1)%SIZE)**

Step 2:

If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.

Step 3:

If it is **NOT FULL**, then check if it is empty **if(front == - 1)** if it is **TRUE**, then set **front=rear = 0 and queue[rear]=value** and terminate

Step 4:

```
Else check if(rear=SIZE-1 && front != 0)
          set  rear=0;
               queue[rear] = value
else set rear=rear+1 ;
          queue[rear] = value; and terminate
```

Circular Queue-Enqueue Operation

```
Step 1:  IF (REAR+1)%SIZE = FRONT
          Write " Queue is Full " and exit
          [End OF IF]

Step 2: IF (FRONT == -1)
          SET FRONT = REAR = 0
          ELSE IF (REAR == SIZE - 1 and FRONT ! = 0)
                    SET REAR = 0
          ELSE
                    SET REAR = (REAR + 1)
          [END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT
```

Circular Queue-Dequeue Operation

In a circular queue, the element is always deleted from front position.
The deQueue() function doesn't take any value as parameter

Step 1: Check whether queue is EMPTY. (front == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element
Then check whether front == SIZE-1, if it is TRUE, then set front = 0.

Step4: else check whether both front and rear are equal (front == rear), if it isTRUE, then set both front = rear = -1)
Else and increment the front, front ++

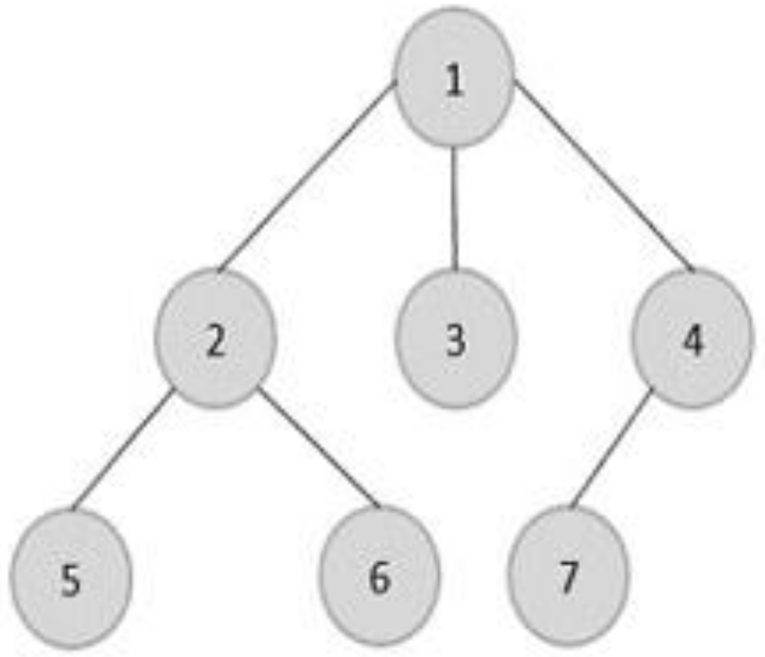
Circular Queue-Dequeue Operation

```
• Step 1: IF FRONT = -1
                Print " UNDERFLOW "
                Goto Step 4
          [END of IF]
• Step 2: SET VAL = QUEUE[FRONT]
• Step 3: IF FRONT = REAR
                    SET FRONT = REAR = -1
                ELSE
                IF FRONT = MAX -1
                    SET FRONT = 0
                ELSE
                    SET FRONT = FRONT + 1
                [END of IF]
                [END OF IF]
• Step 4: EXIT
```

Tree

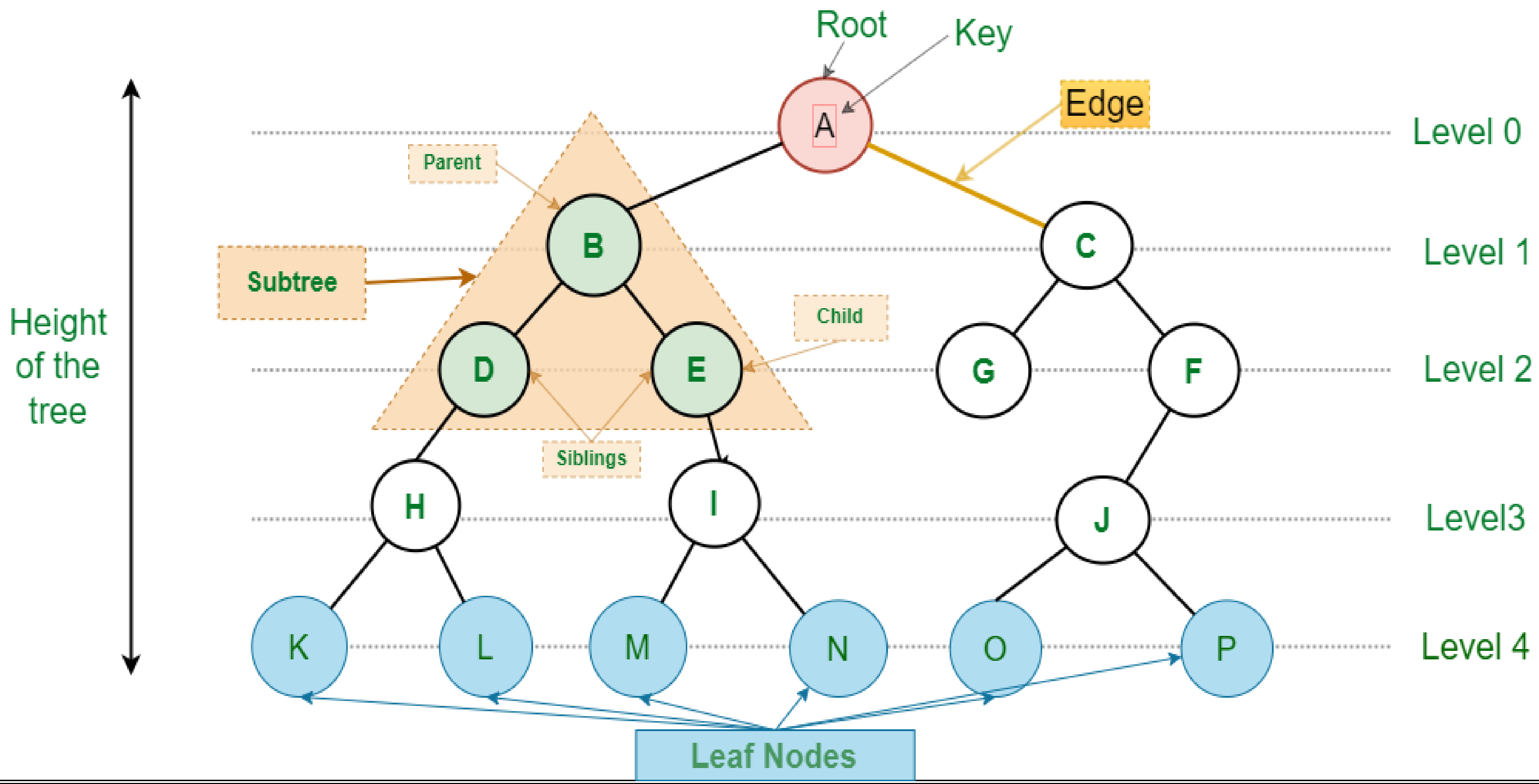
Tree

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- The topmost node of the tree is called the root, and the nodes below it are called the child nodes.
- Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

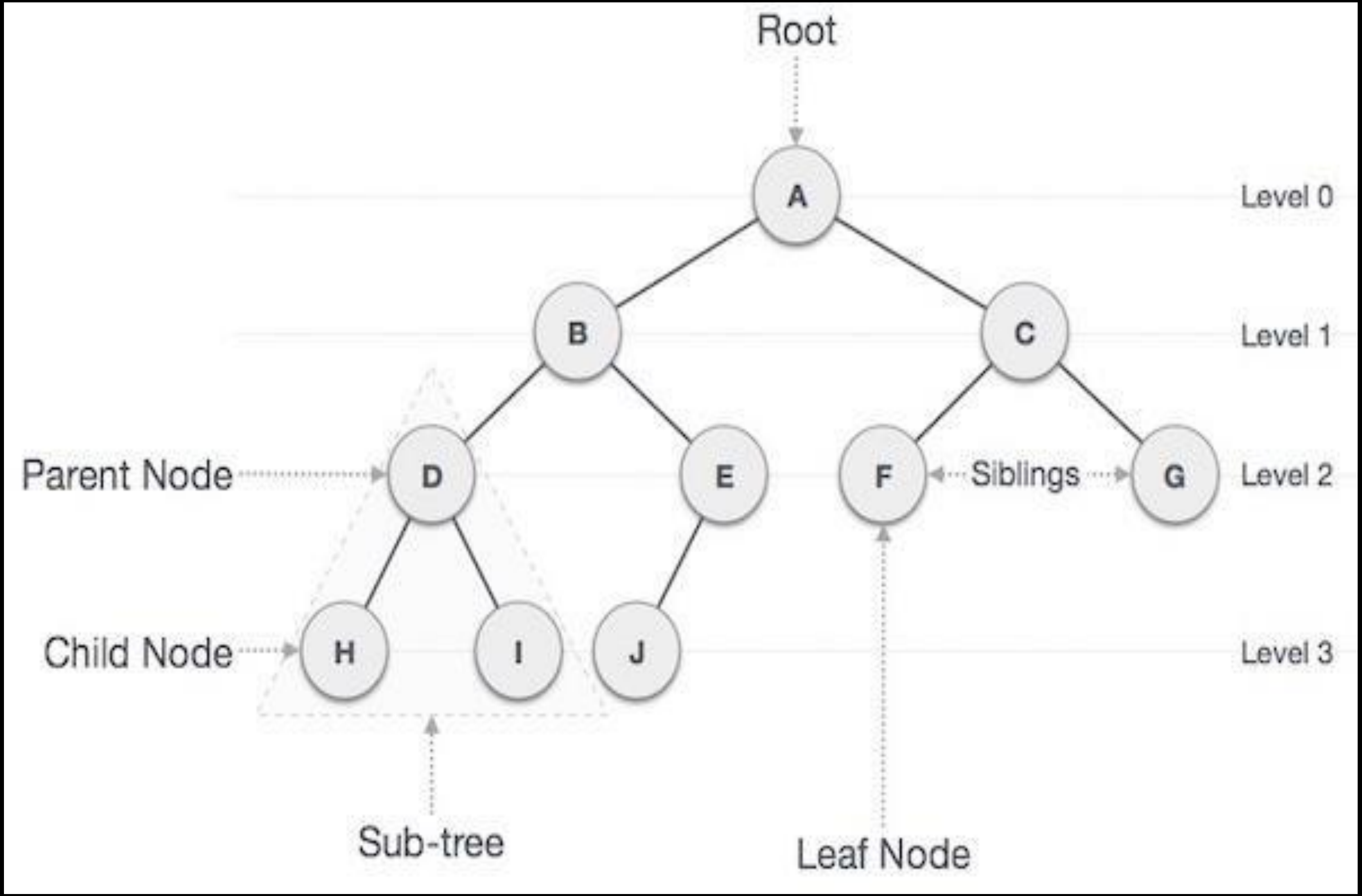


General Tree Data Structure

Tree Data Structure



Tree


Root
A
Level 0
B
C
Level 1
Parent Node
D
E
F
Siblings
G
Level 2
Child Node
H
I
J
Level 3
Sub-tree
Leaf Node

Tree

Root

- It is the topmost node of a tree.

Height of a Node

- The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

- The depth of a node is the number of edges from the root to the node.

Height of a Tree

- The height of a Tree is the height of the root node or the depth of the deepest node.

Basic Terminologies In Tree Data Structure:

i. **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.

ii. **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.

iii. **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.

iv. **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {K, L, M, N, O, P} are the leaf nodes of the tree.

v. **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}

Basic Terminologies In Tree Data Structure:

vi. **Descendant:** Any successor node on the path from the leaf node to that node. {E,I} are the descendants of the node {B}.

vii. **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.

viii. **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.

ix. **Internal node:** A node with at least one child is called Internal Node.

x. **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.

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

Basic Terminologies In Tree Data Structure:

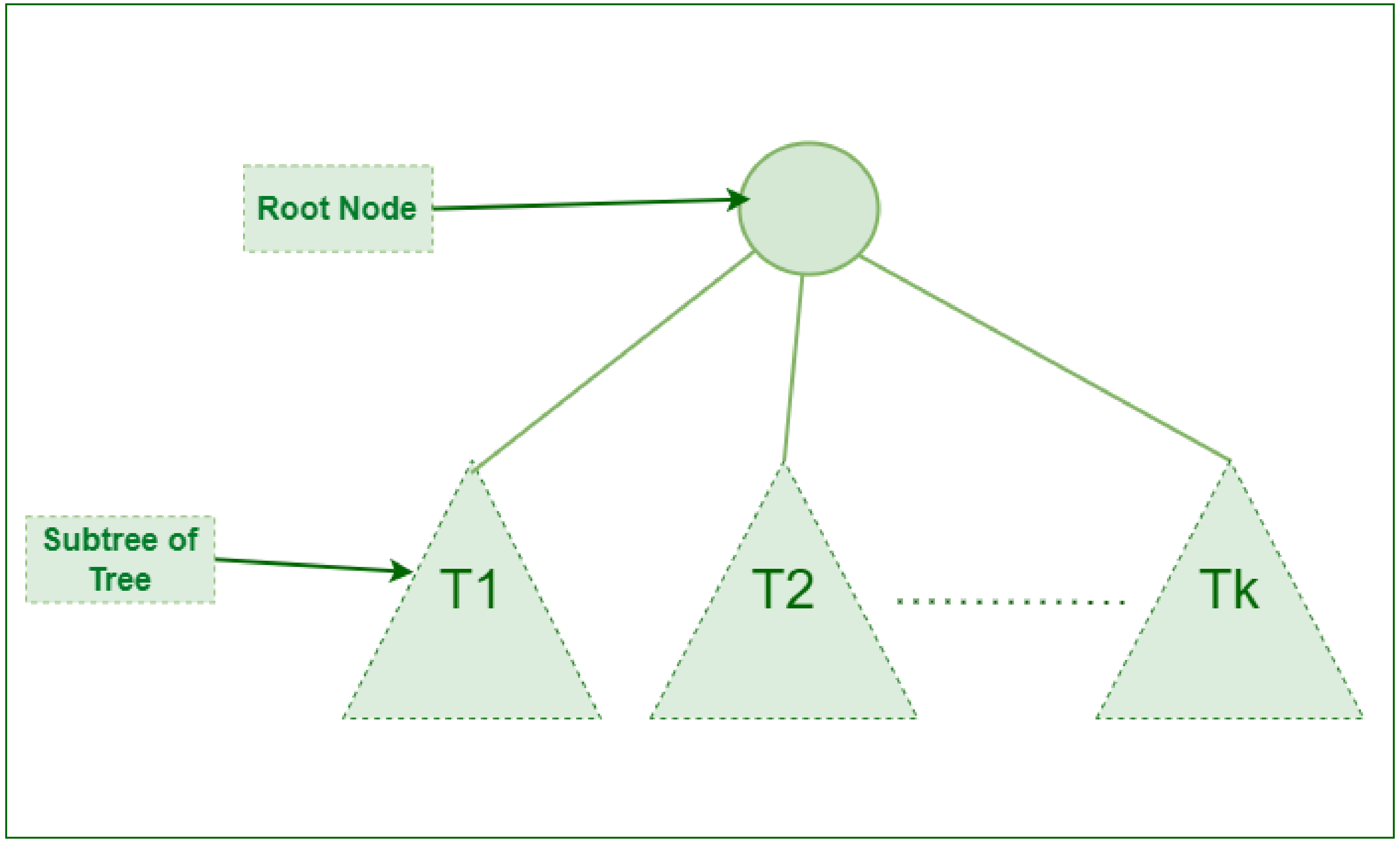
xii. Visiting – Visiting refers to checking the value of a node when control is on the node.

xiii. Traversing – Traversing means passing through nodes in a specific order.

xiv. Keys – Key represents a value of a node based on which a search operation is to be carried out for a node.

Representation of Tree Data Structure:

- ***A tree consists of a root, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root of the tree to the root of each subtree.***



Representation of a Node in Tree:

```
struct Node
{
        int data;
        struct Node *first_child;
        struct Node *second_child;
        struct Node *third_child;
        .
        .
        .
        struct Node *nth_child;
};
```

Types of Trees

- three types :

1. General Trees
2. Binary Trees
3. Binary Search Trees