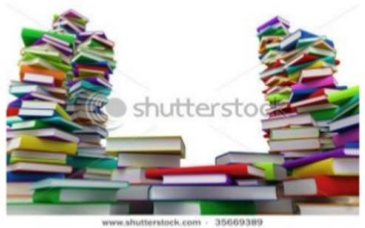
# Data Structure

# Sep23 : Day 8

**Kiran Waghmare**

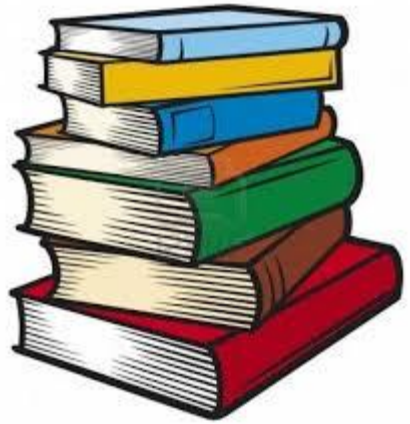
**CDAC Mumbai**

## Examples of stack



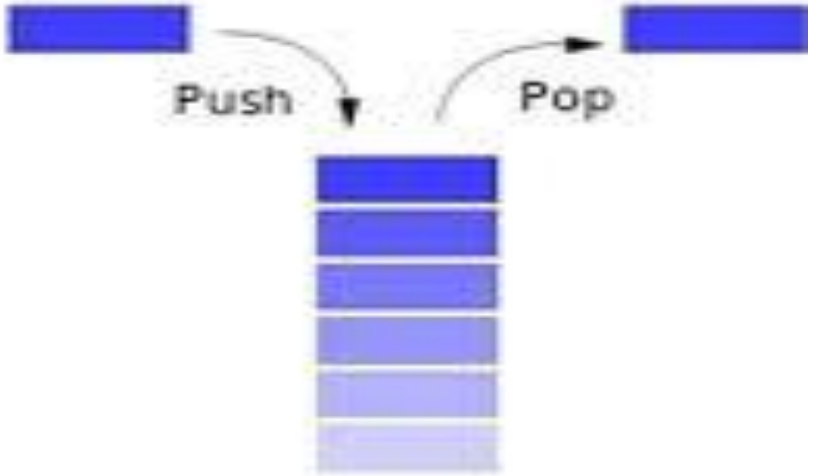
# Stacks

Kiran Waghmare



Stack of books

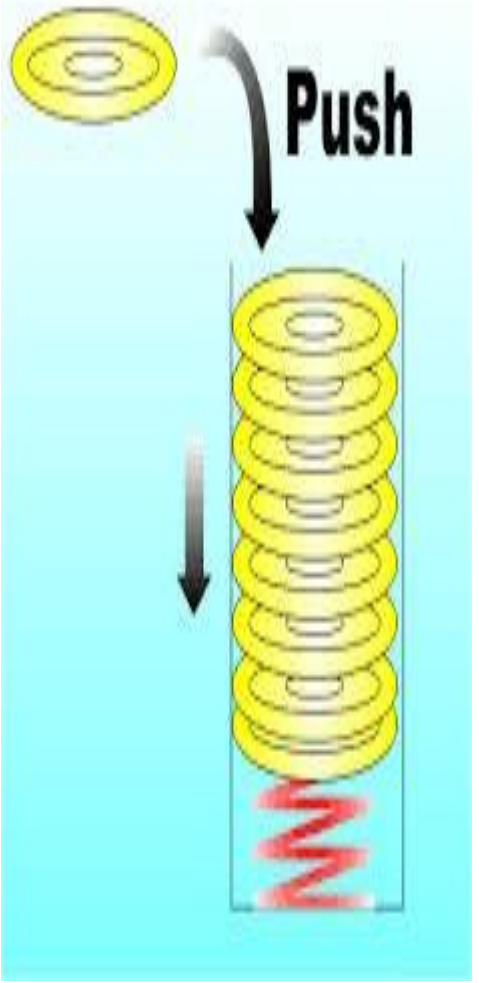
Stack of Coins




Memory stack

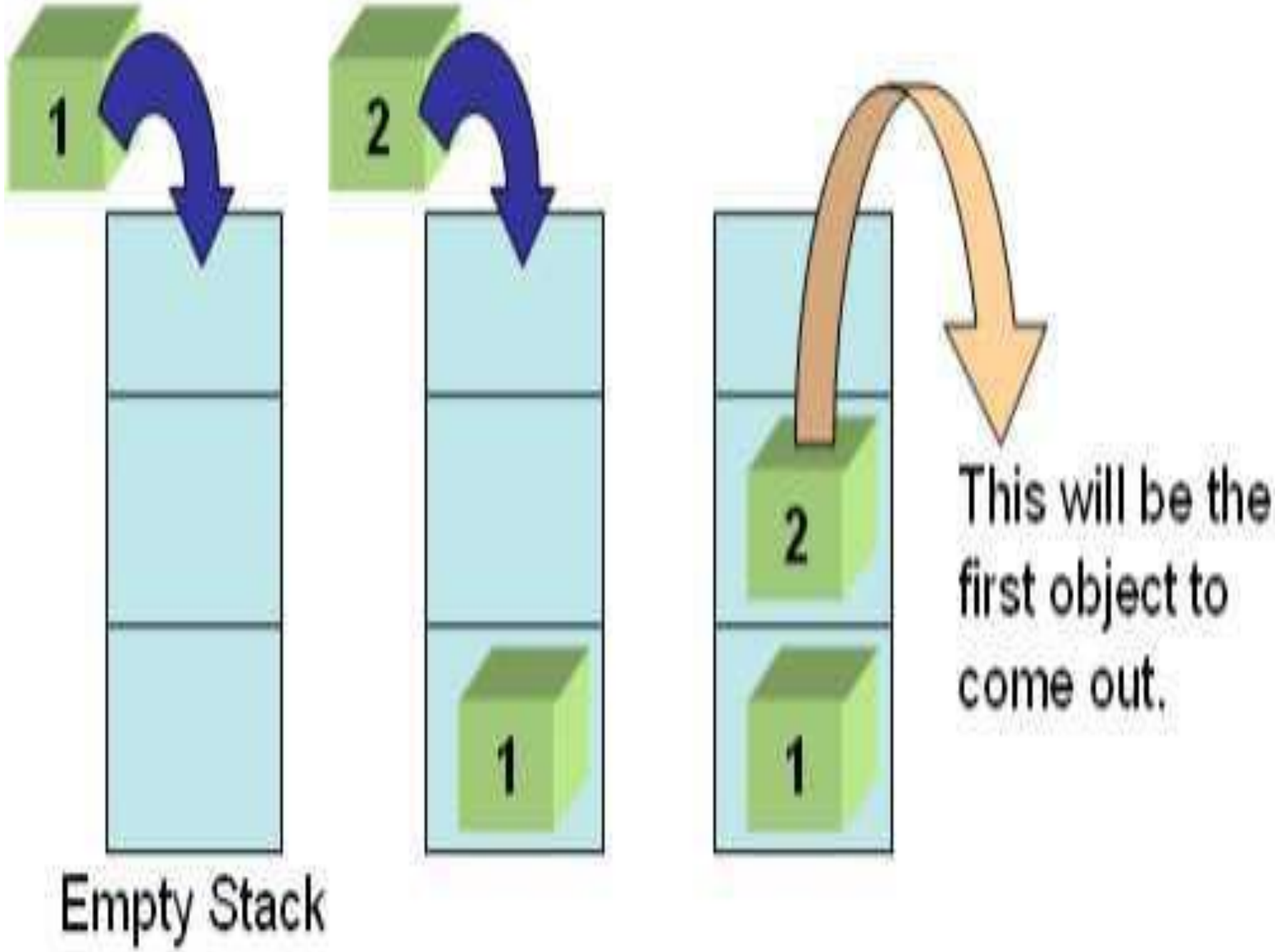
# Stack

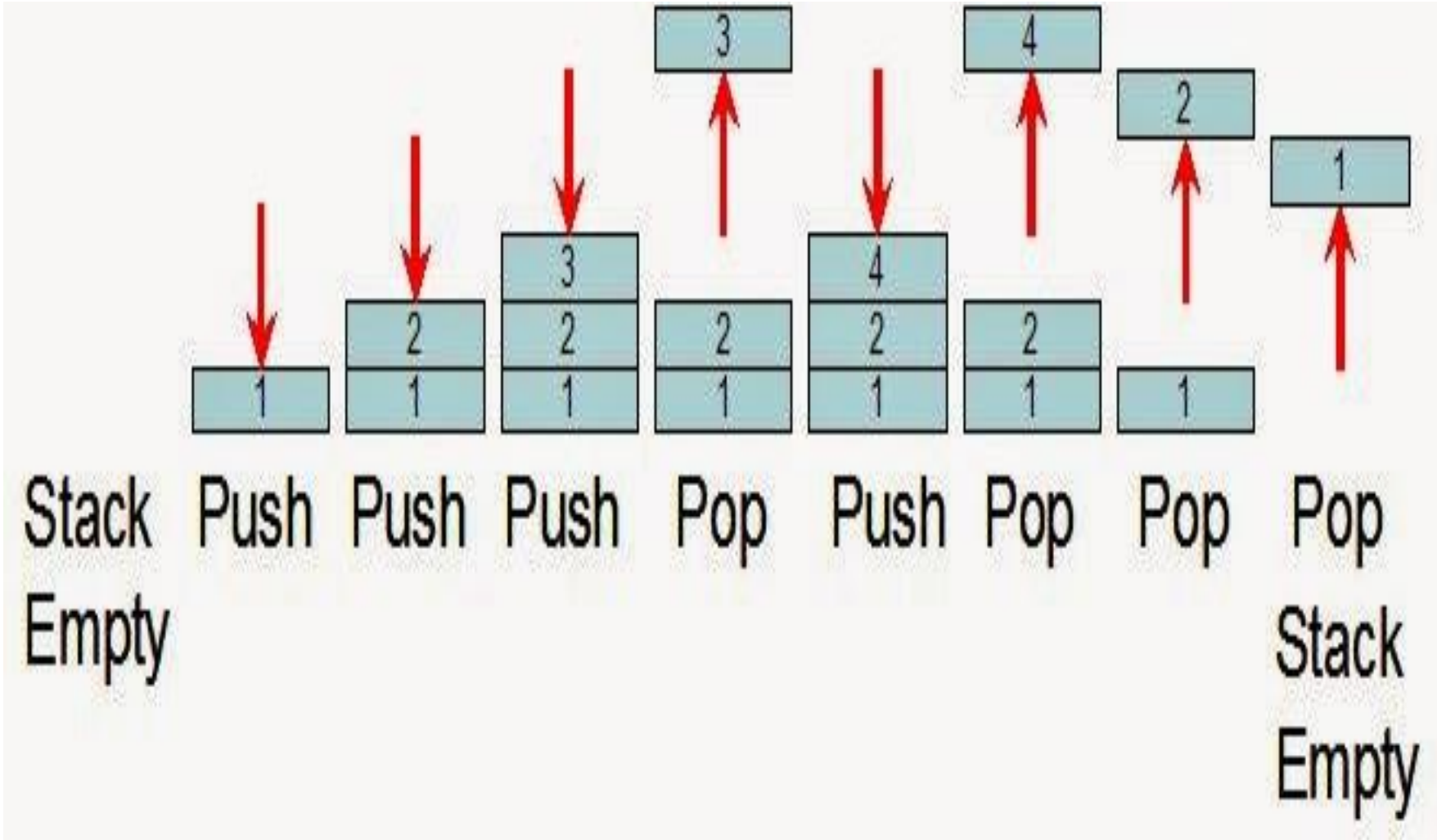
- Stack is an ordered list of similar data type.
- Stack is a LIFO structure. (Last in First out).
- push() function is used to insert new elements into the Stack and pop() is used to delete an element from the stack. Both insertion and deletion are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.



# Standard Stack Operations

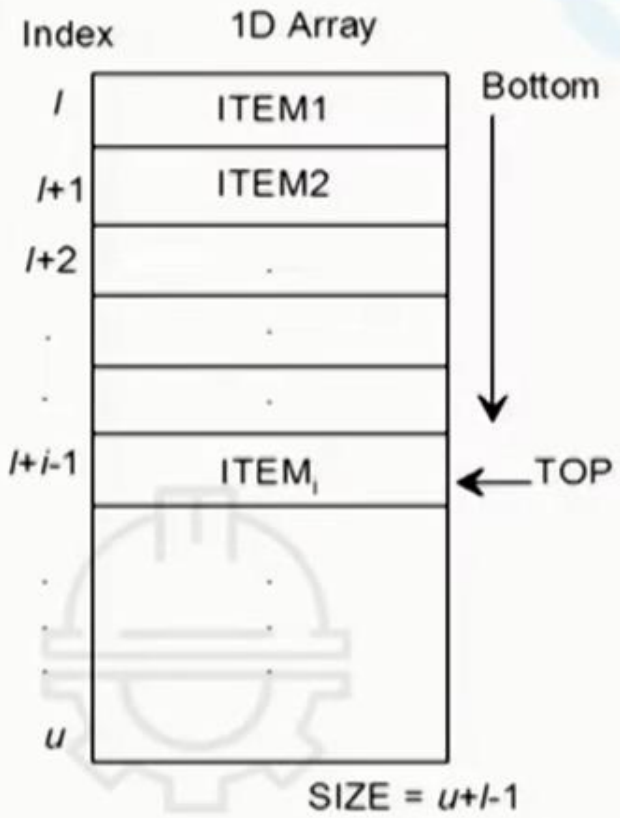
- **The following are some common operations implemented on the stack:**
- **push():**
  - When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- **pop():**
  - When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- **isEmpty():**
  - It determines whether the stack is empty or not.
- **isFull():**
  - It determines whether the stack is full or not.'
- **peek():**
  - It returns the element at the given position.
- **count():**
  - It returns the total number of elements available in a stack.
- **change():**
  - It changes the element at the given position.
- **display():**
  - It prints all the elements available in the stack.


1
2
2
1
2
1
This will be the first object to come out.
Empty Stack

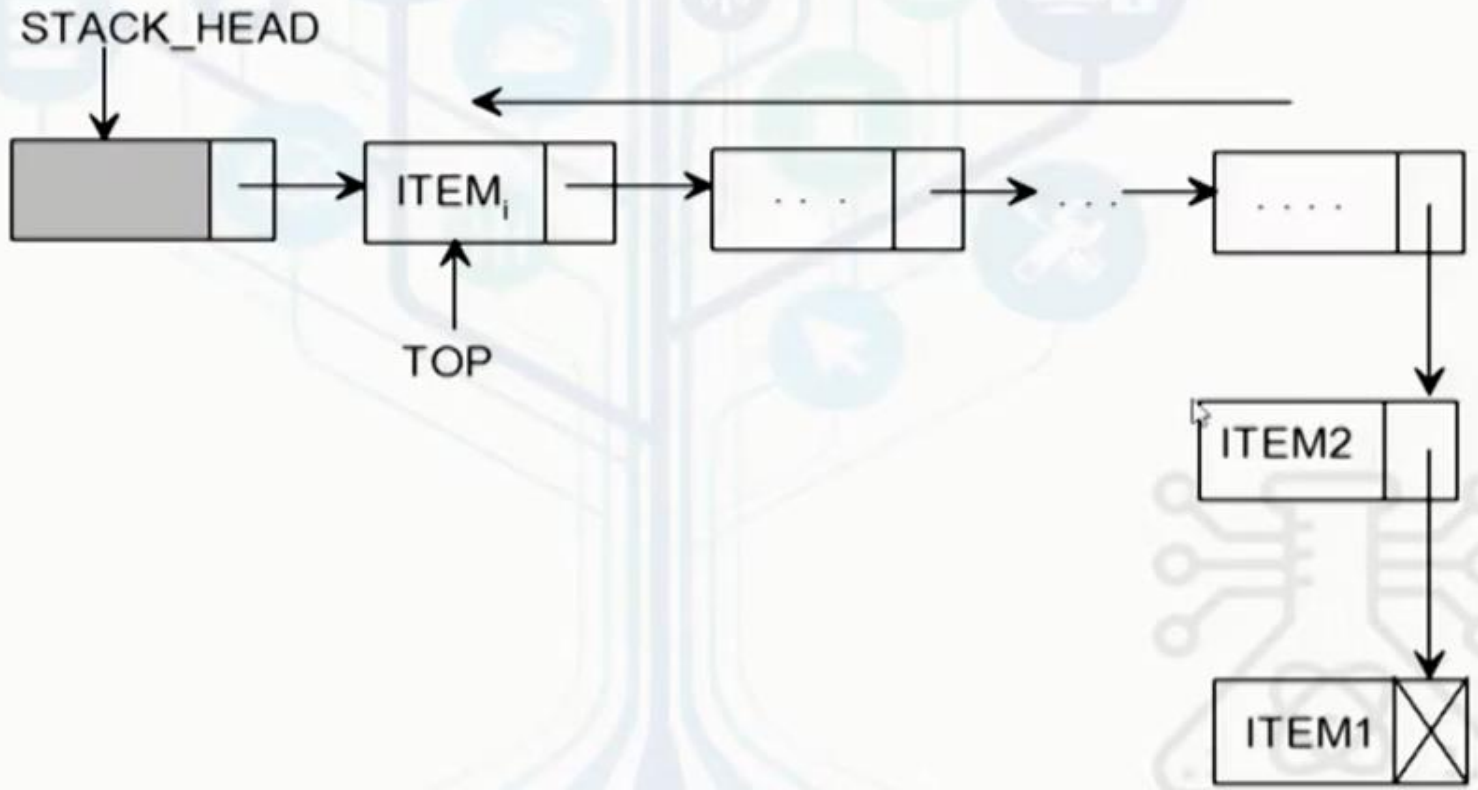

Stack Empty
Push
Push
Push
Pop
Push
Pop
Pop
Pop
Stack Empty
1
2
3
4

# Memory representations

## Array representation




## Linked list representation



```
void push(long j)
{
    stackArray[++top] = j;
}

long pop()
{
    return stackArray[top--];
}

long peek()
{
    return stackArray[top];
}

boolean isEmpty()
{
    return (top == -1);
}
```

# Applications of Stack

- **The following are the applications of the stack:**

- **Balancing of symbols:**
  - Stack is used for balancing a symbol. For example, we have the following program:
  - As we know, each program has an opening and closing braces;
  - when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack.
  - Therefore, the net value comes out to be zero.
  - If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:**
  - Stack is also used for reversing a string.
  - For example, we want to reverse a "javaTpoint" string, so we can achieve this with the help of a stack.
  - First, we push all the characters of the string in a stack until we reach the null character.
  - After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.

- **UNDO/REDO:**
  - It can also be used for performing UNDO/REDO operations.
  - For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc.
  - So, there are three states, a, ab, and abc, which are stored in a stack.
  - There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  - If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:**
  - The recursion means that the function is calling itself again.
  - To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):**
  - This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:**
  - Suppose we have to create a path to solve a maze problem.
  - If we are moving in a particular path, and we realize that we come on the wrong way.
  - In order to come at the beginning of the path to create a new path, we have to use the stack data structure.

- **Expression conversion:**
  - Stack can also be used for expression conversion.
  - This is one of the most important applications of stack.
  - The list of the expression conversion is given below:
    - Infix to prefix
    - Infix to postfix
    - Prefix to infix
    - Prefix to postfix
    - Postfix to infix
- **Memory management:**
  - The stack manages the memory.
  - The memory is assigned in the contiguous memory blocks.
  - The memory is known as stack memory as all the variables are assigned in a function call stack memory.
  - The memory size assigned to the program is known to the compiler.

  When the function is created, all its variables are assigned in the stack memory.

  When the function completed its execution, all the variables assigned in the stack are released.

# Parenthesis matching problem : Algorithm

1. For each character of input string
2. If character is opening parenthesis '(', put it on stack.
3. If character is closing parenthesis ')'
   a. Check top of stack, if it is '(' , pop and move to next character.
   b. If it is not '(', return false
4. After scanning the entire string, check if stack is empty. If stack is empty, return true else return false.

# Check for Balanced Brackets in an expression (well-formedness) using Stack

- Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in the given expression.

- Example:

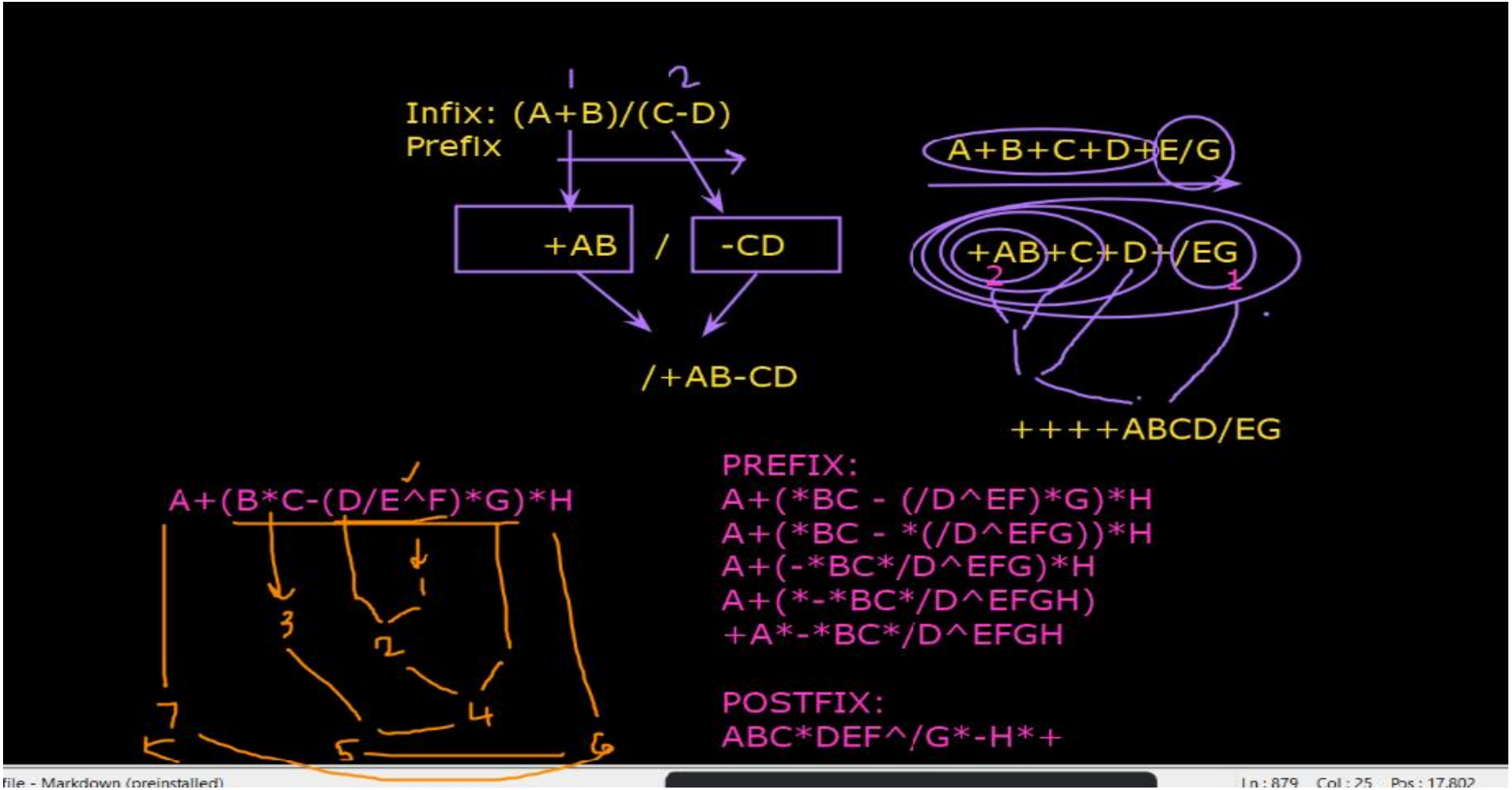
- Input: exp = “[()]{}{[()()]()}”
- Output: Balanced
- Explanation: all the brackets are well-formed

- Input: exp = “[(])”
- Output: Not Balanced
- Explanation: 1 and 4 brackets are not balanced because
- there is a closing ‘]’ before the closing ‘(‘

# Polish Notations

1. **Infix Notation : A+B**
2. **Prefix Notation: +AB**
3. **Postfix Notation : AB+**

- **Operator Precedence:**
  1. **BODMAS Rule**
  2. **Brackets, Exponential, (* / %), (+ -)**
- **Rules: Infix to Postfix Conversion**
  1. Parenthesize the expression starting from left to right.
  2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression B* C is parenthesized first before A+B.
  3. The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.
  4. Once the expression is converted to postfix from remove the parenthesis.


Infix: (A+B)/(C-D)
Prefix
1
2
+AB
/
-CD
/+AB-CD
A+B+C+D+E/G
+AB+C+D+/EG
2
1
++++ABCD/EG
A+(B*C-(D/E^F)*G)*H
1
2
3
4
5
6
7
PREFIX:
A+(*BC - (/D^EF)*G)*H
A+(*BC - *(/D^EFG))*H
A+(-*BC*/D^EFG)*H
A+(*-*BC*/D^EFGH)
+A*-*BC*/D^EFGH
POSTFIX:
ABC*DEF^/G*-H*+

**ALGORITHM:**

- **Scan infix expression from left to right.**
- **If there is a character as operand, output it.**
- **if not**

  **1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '(' ), push it.**

  **2 Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)**
- **If the scanned character is an '(', push it to the stack.**
- **If the character character is an ')', pop the stack and and output it until a '(' is encountered, and discard both the parenthesis.**
- **Repeat steps 2-6 until infix expression is scanned.**
- **display the output**
- **Pop and output from the stack until it is not empty.**

(A+(B*C)) Infix ---> Postfix

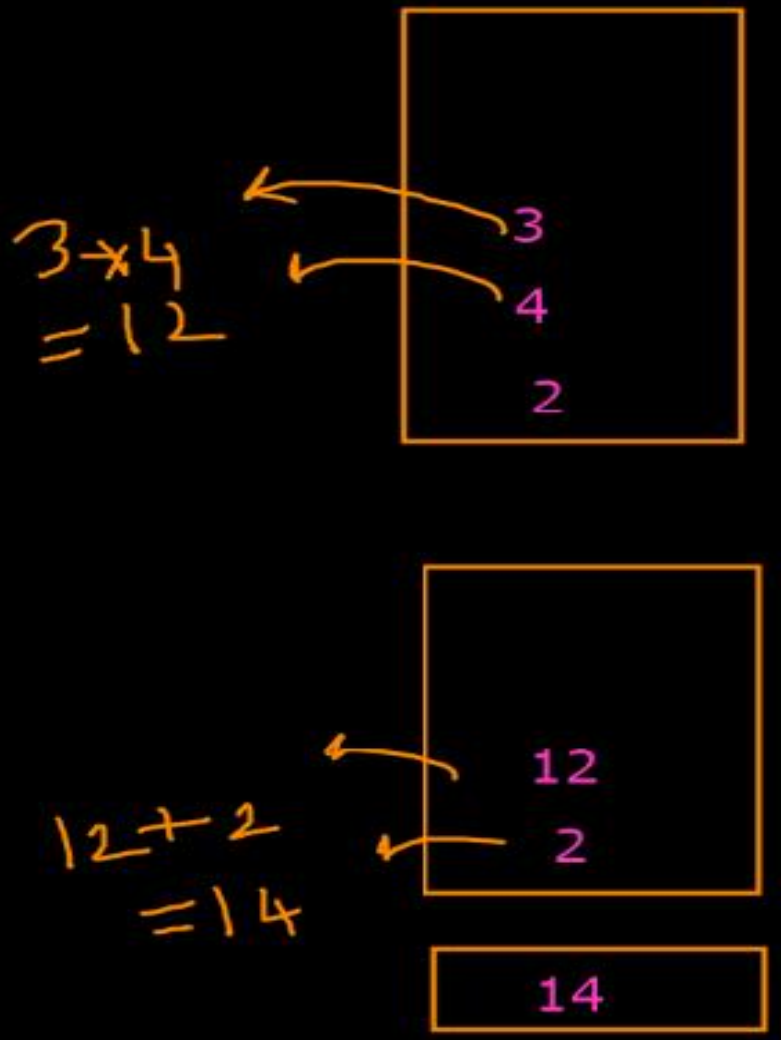
| INPUT | STACK | OUTPUT |
|---|---|---|
| A | | A |
| + | + | A |
| ( | +( | A |
| B | +( | AB |
| * | +(* | AB |
| C | +(* | ABC |
| ) | + | ABC* |
| | | ABC*+ |

Input =(a+(b*c)/(d-e))  Infix ---> Postfix

| Input | Stack | Output |
|---|---|---|
| ( | ( | |
| a | ( | a |
| + | (+ | a |
| ( | (+( | a |
| b | (+( | ab |
| * | (+(* | ab |
| c | (+(* | abc |
| ) | (+ | abc* |
| / | (+/ | abc* |
| ( | (+/( | abc* |
| d | (+/( | abc*d |
| - | (+/(- | abc*d |
| e | (+/(- | abc*de |
| ) | (+/ | abc*de- |
| ) | - | abc*de-/+ |

# Postfix

Infix : 2+4*3
Postfix: 2 4 3 * +



## ADVANTAGE OF POSTFIX:

- Any formula can be expressed without parenthesis.
- It is very convenient for evaluating formulas on computer with stacks.
- Postfix expression doesn’t has the operator precedence.
- Postfix is slightly easier to evaluate.
- It reflects the order in which operations are performed.
- You need to worry about the left and right associativity.

# Application: Evaluation of a postfix expression

***Steps:***

1. Append a special delimiter '#' at the end of the expression
2. item = *E*.**ReadSymbol**( ) // Read the first symbol from *E*
3. **While** (item ≠ '#') **do**
4. **If** (item = operand) **then**
5. **PUSH**(item) // Operand is the first push into the stack
6. **Else**
7. op = item // The item is an operator
8. *y* = **POP**( ) // The right-most operand of the current operator
9. *x* = **POP**( ) // The left-most operand of the current operator
10. *t* = *x* op *y* // Perform the operation with operator 'op' and operands *x*, *y*
11. **PUSH**(*t*) // Push the result into stack
12. **EndIf**
13. item = *E*.**ReadSymbol**( ) // Read the next item from *E*
14. **EndWhile**
15. value = **POP**( ) // Get the value of the expression
16. **Return**(value)
17. **Stop**

# Balance Parenthesis

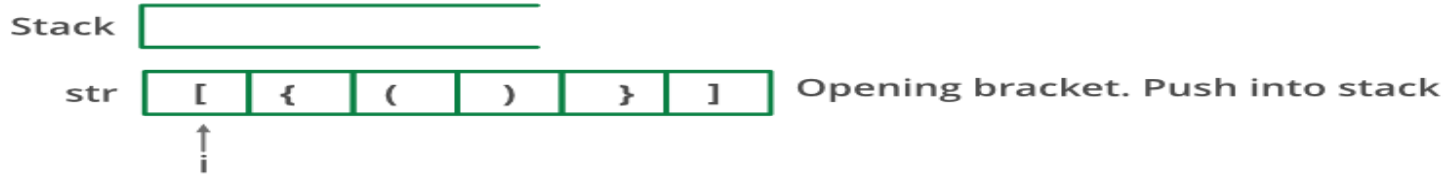
- Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in exp.
- **Example**:
- ***Input**: exp = “[(])”*
  ***Output**: Not Balanced*

- ***Input**: exp = “[()]{}{[()()]()}”*
  ***Output**: Balanced*

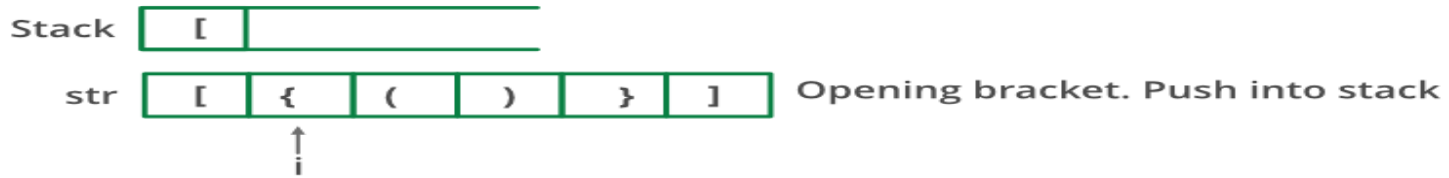
**Algorithm:**

- Declare a character stack S.
- Now traverse the expression string exp.
  - If the current character is a starting bracket (‘**(**‘ **or** ‘**{**‘ **or** ‘**[**‘) then push it to stack.
  - If the current character is a closing bracket (‘**)**’ **or** ‘**}**’ **or** ‘**]**’) then pop from stack and if the popped character is the matching starting bracket then fine else brackets are not balanced.
- After complete traversal, if there is some starting bracket left in stack then “not balanced”

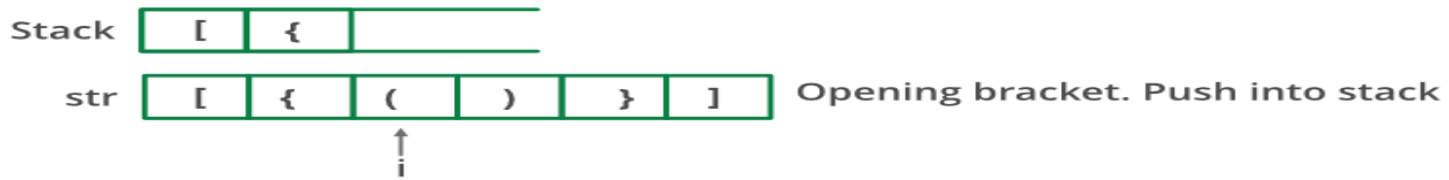
**Initially :**




**Step 1:**



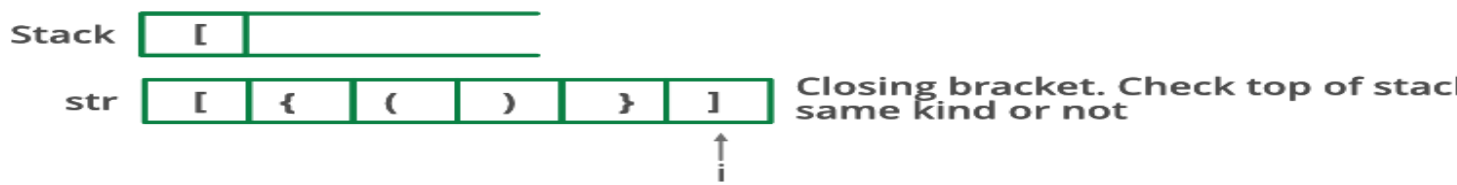

**Step 2:**




**Step 3:**


**Step 4:**


**Step 5:**



# Queue

Kiran Waghmare

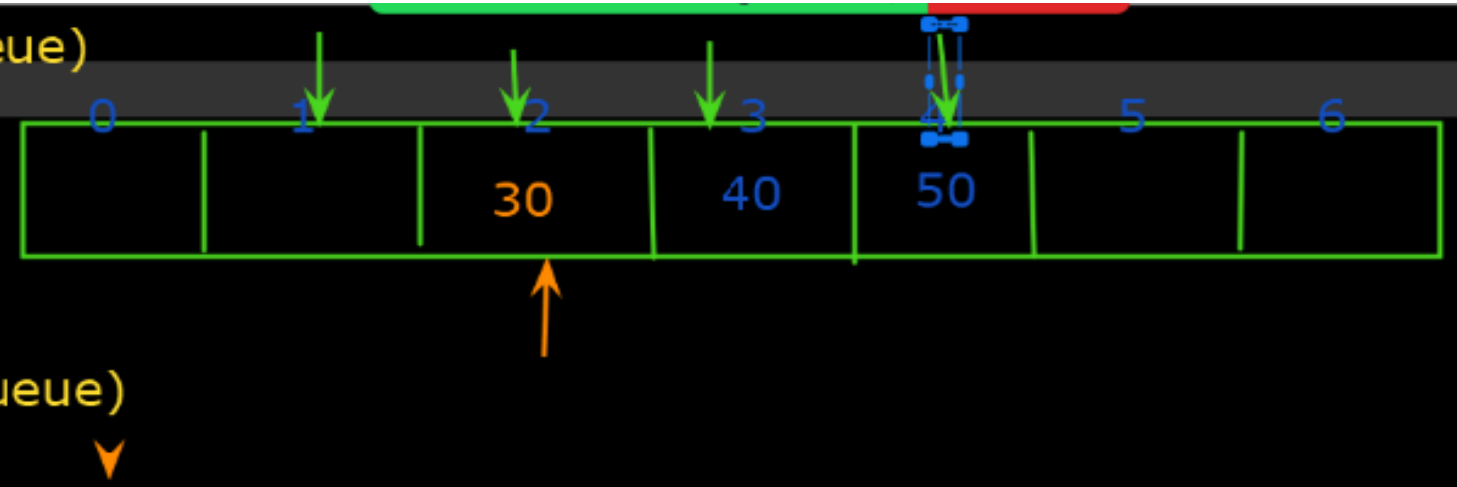
Agenda:

---

-Stack

-Queue

Queue:



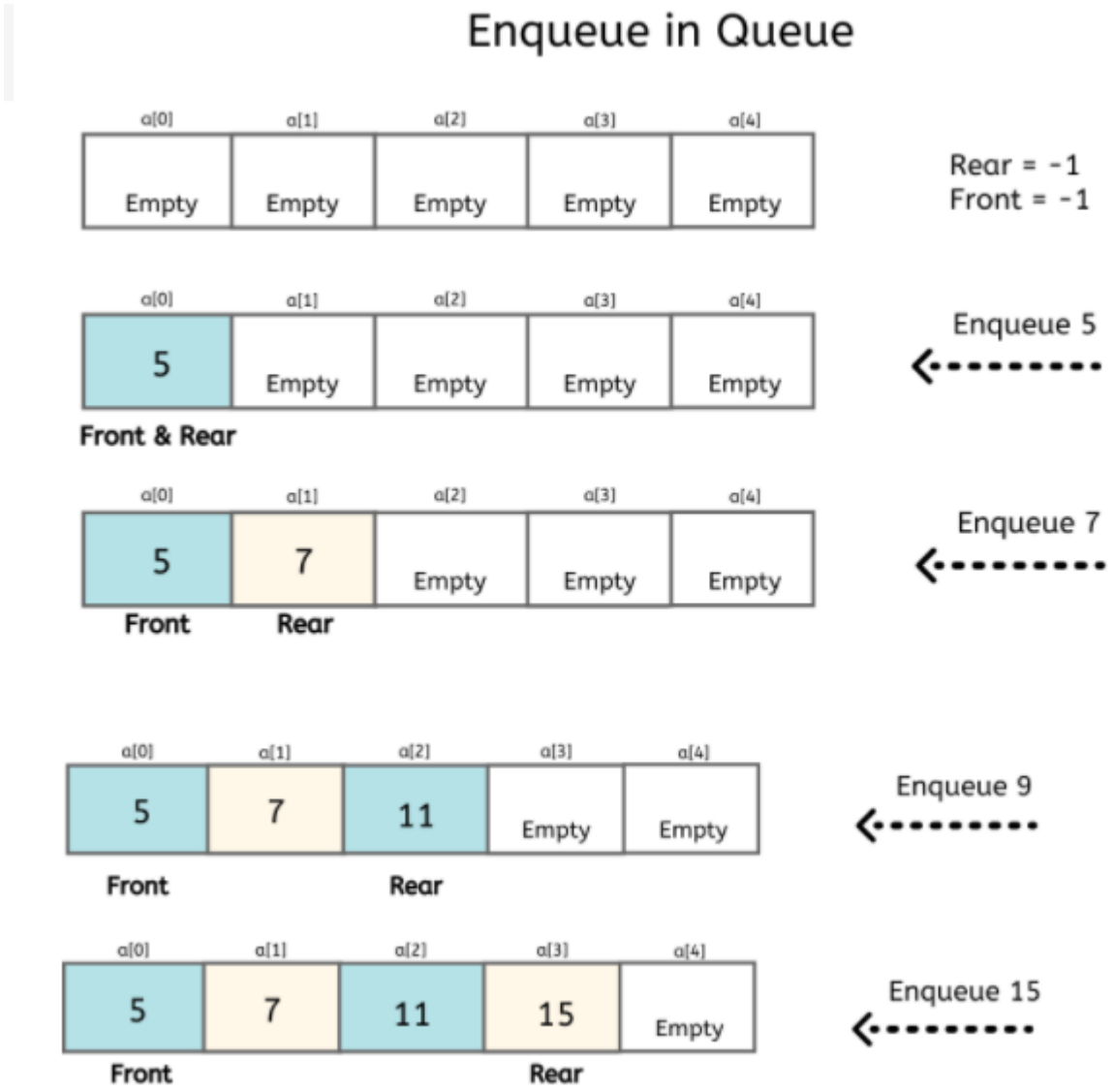
# Queue

- Ordered collection of homogeneous elements.
- Non-primitive linear data structure.
- A new element is added at one end called rear end and the existing elements are deleted from the other end called front end.
- This mechanism is called First-In-First-Out (FIFO)
- Total no. of elements in queue = rear-front+1

## 1. Enqueue()

**When we requirea to add an element to the Queue we perform Enqueue() operation.**

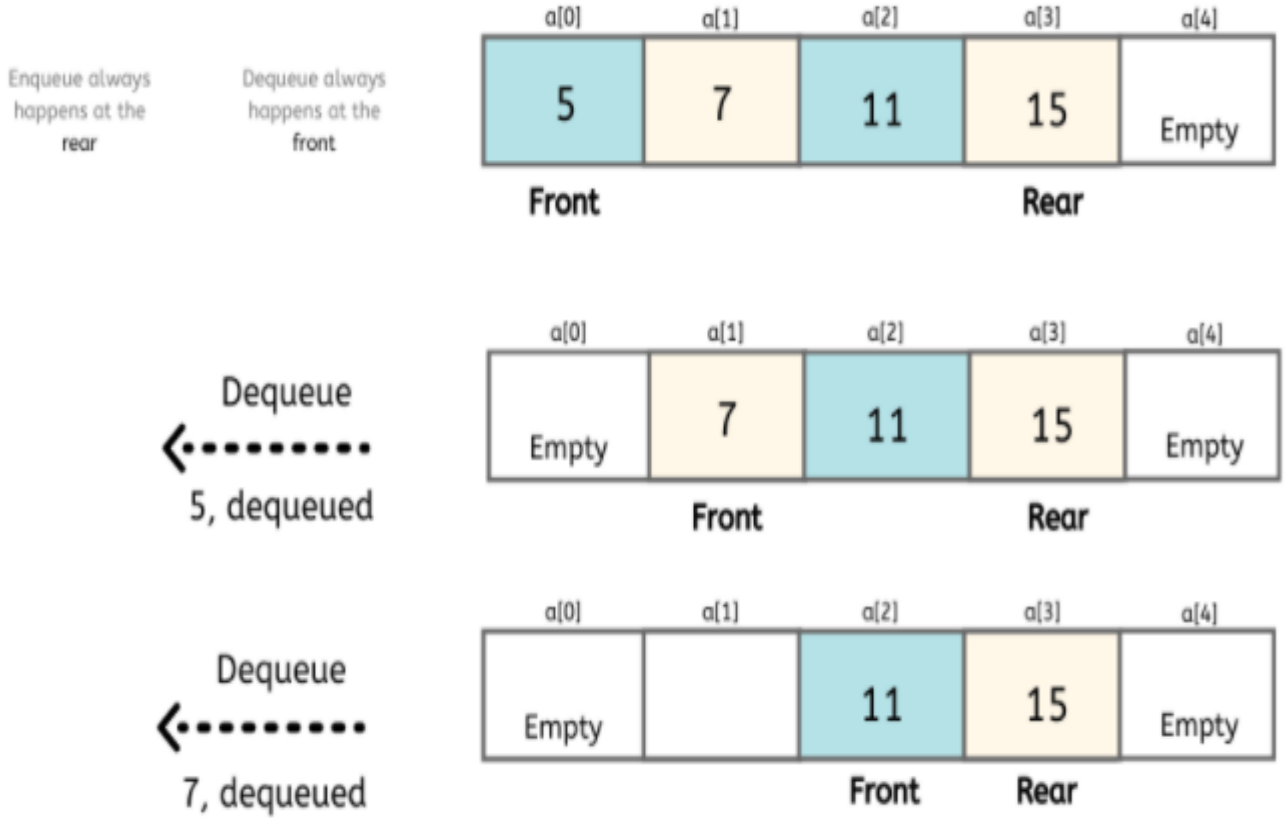
**Push() operation is synonymous of insertion/addition in a data structure.**



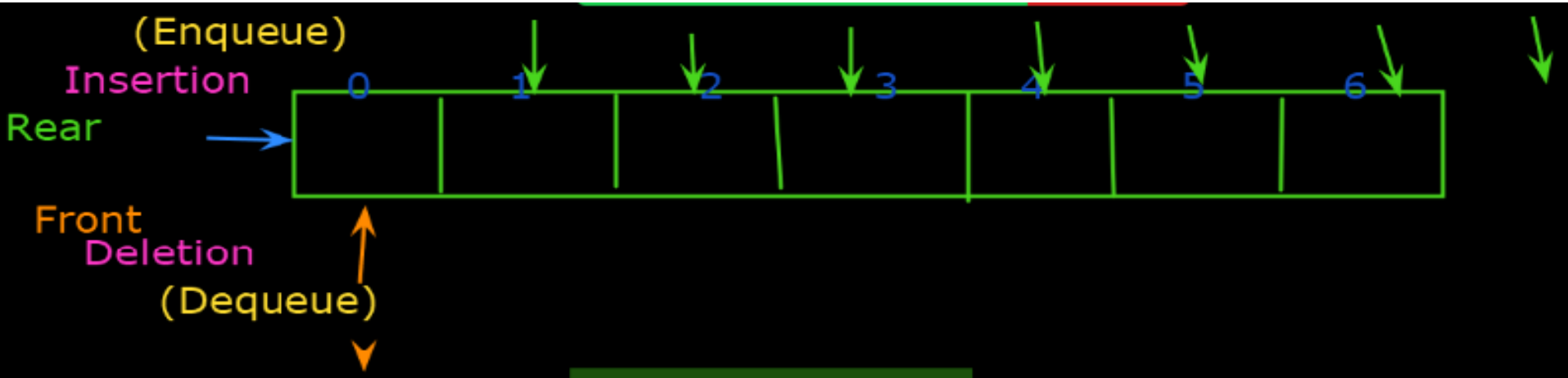
## 2. Dequeue()

**When we require to delete/remove an element to the Queue we perform Dequeue() operation.**

**Dequeue() operation is synonymous of deletion/removal in a data structure.**



Agenda:
---
-Stack
-Queue

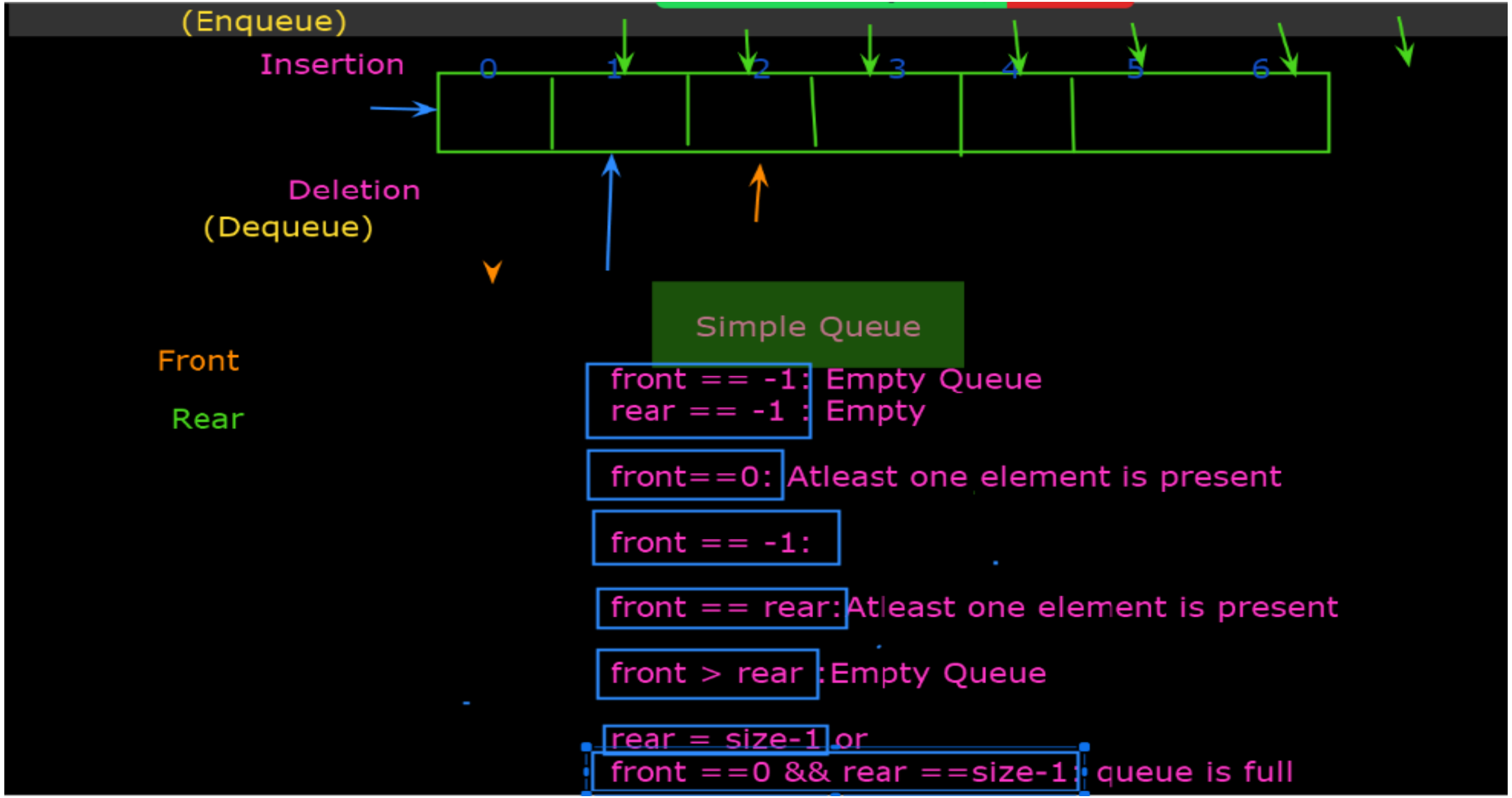



Queue:

Simple Queue

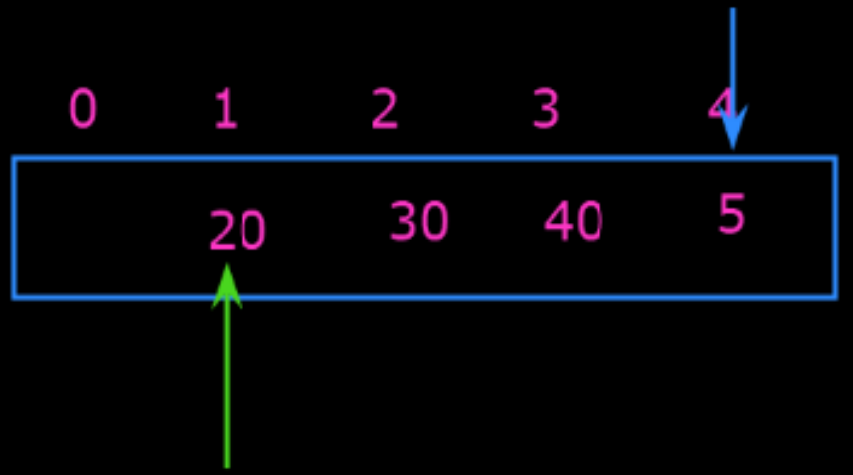
-data structure
-Queue follows the rule : FIFO (First In First Out)
-Insertion operation is called as Enqueue
-Deletion operation is called as Dequeue
-Useing 2 pointers:
    -Rear: Inserting an element in queue
    -Front: Deleting an element in queue
    -Rear = -1
    -Front = -1/ 0

Operations on th Queue:
---
-Enqueue
-Dequeue


(Enqueue)
Insertion
0
1
2
3
4
5
6
Deletion
(Dequeue)
Simple Queue
Front
Rear
front == -1: Empty Queue
rear == -1 : Empty
front==0: Atleast one element is present
front == -1:
front == rear: Atleast one element is present
front > rear :Empty Queue
rear = size-1 or
front ==0 && rear ==size-1: queue is full

```
    {
        for(int i=front;i<=rear;i++)
        {
            System.out.println(Q[i]+" ");
        }
    }
    //print the values of rear & front

    public static void main(String args[])
    {
    }
}
```




Time complexity:

---

Enqueue: O(1)
Dequeue: O(1)

# Circular Queue

- In a circular queue, all nodes are treated as circular. Last node is connected back to the first node.
- Circular queue is also called as **Ring Buffer.**
- It is an abstract data type.
- Circular queue contains a collection of data which allows insertion of data at the end of the queue and deletion of data at the beginning of the queue

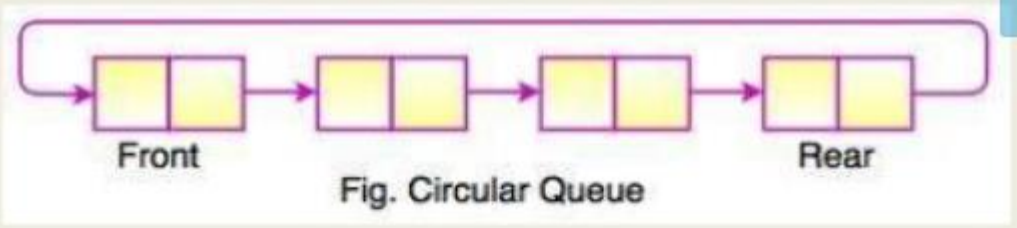


Fig. Circular Queue

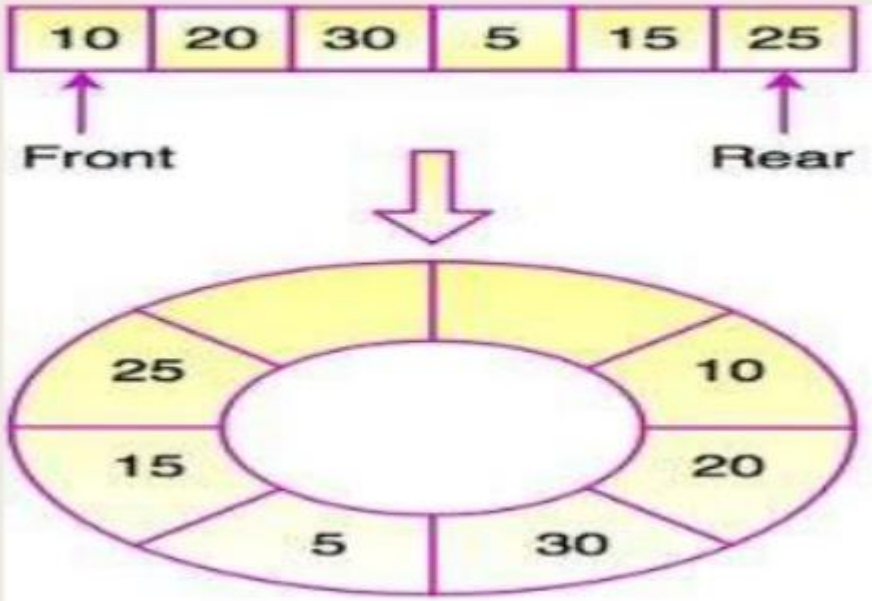
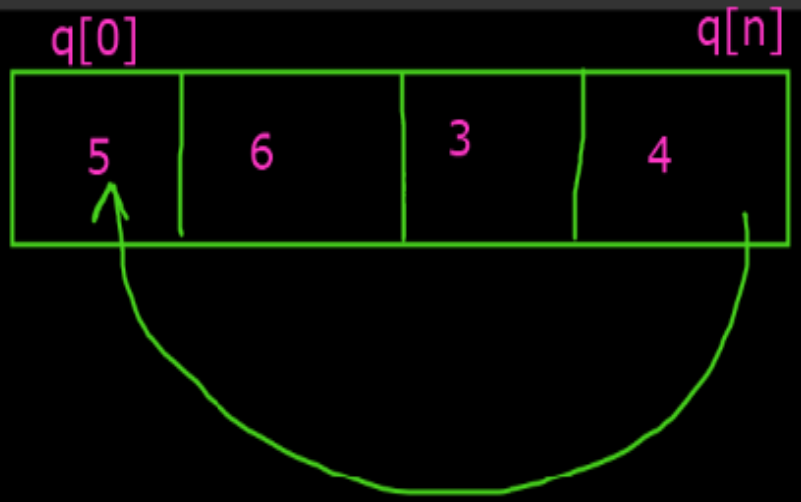


Fig. Circular Queue

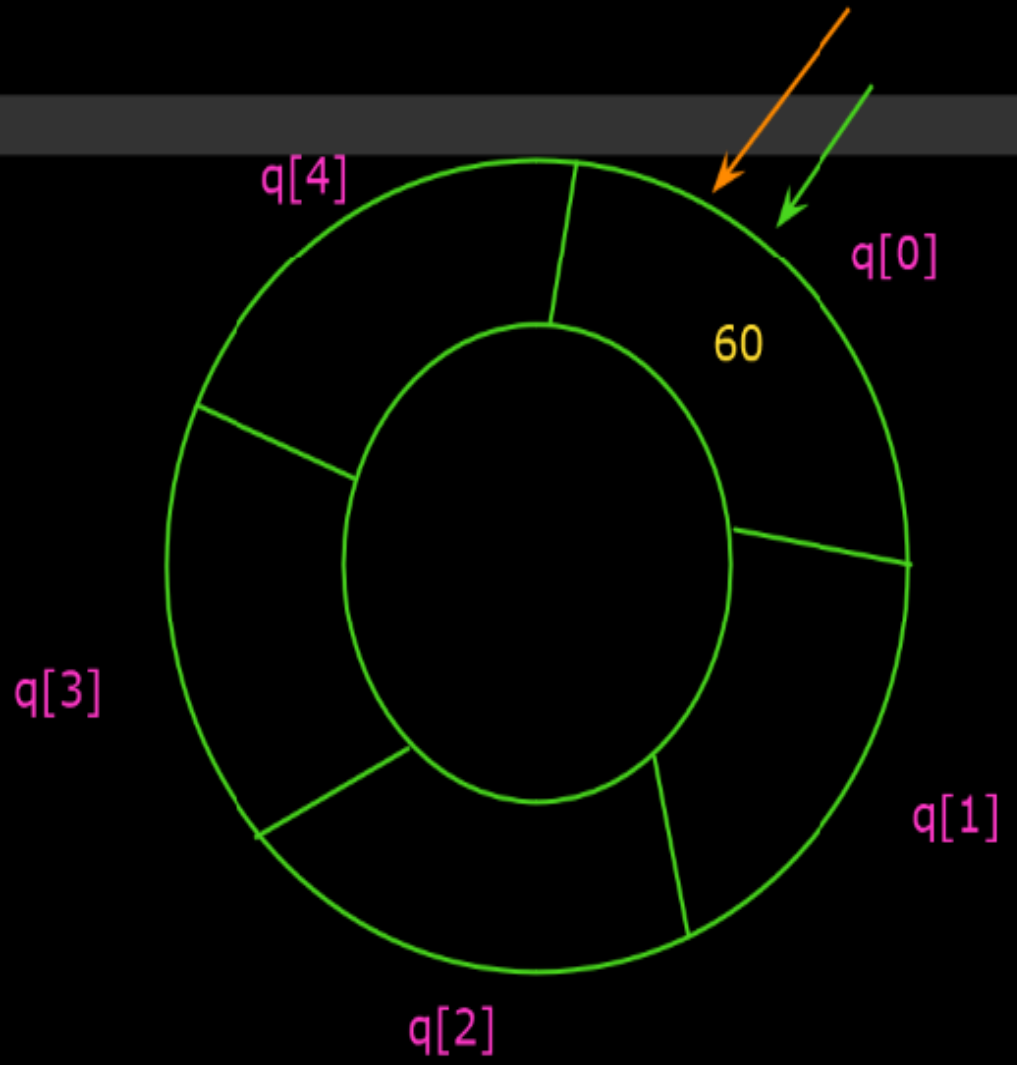
Circular Queue:




front = (front+1)%(size)

rear = (rear +1)%(size)

= (4+1)%5
=5%5
=0



```
            //Reset the queue
            front = -1;
            rear = -1;
        }
        else
        {
            front=(front+1)%size;
        }
        System.out.println("Deleted element is = "+x);
        return x;
    }

}

void display()
{
    for(int i=front;i<=rear;i++)
    {
        System.out.println(Q[i]+" ");
    }
}
```

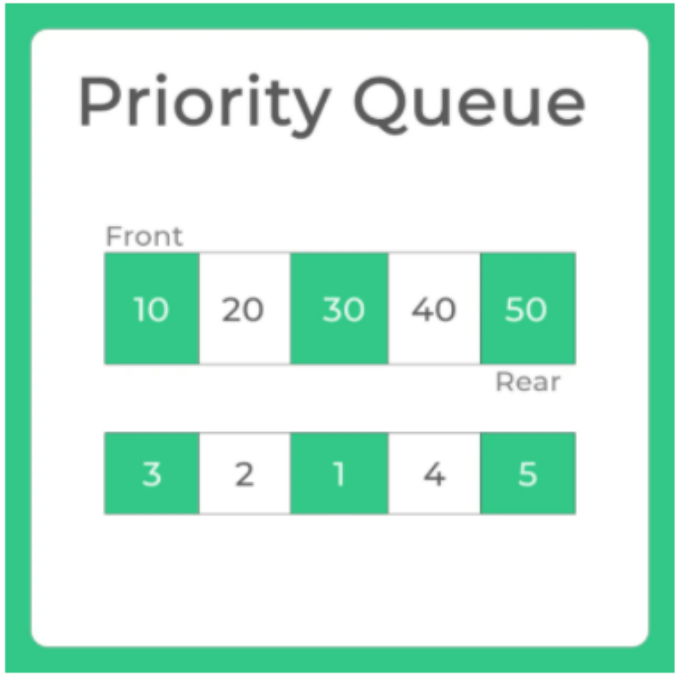
Time Complexity:

---

Enqueue: O(1)
Dequeue: O(1)

# Priority Queue

Priority Queue is a special type of queue in which elements are treated according to their priority. Insertion of an element take place at the rear of the queue but the deletion or removal of an element take place according to the priority of the element. Element with the highest priority is removed first and element with the lowest priority is removed last.

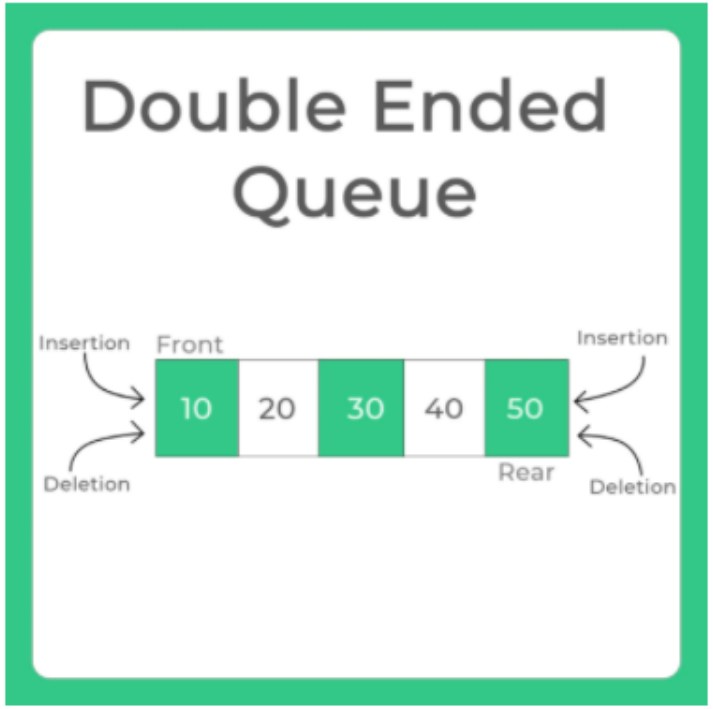



## Applications of Priority Queue

The applications of priority queue are:-

- Dijkstra's shortest path algorithm
- Data compression in huffman codes.
- Load balancing and interrupt handling in operating system.
- Sorting heap.

# Double Ended Queue




Double ended queue are also known as deque. In this type of queue insertion and deletion of an element can take place at both the ends. Further deque is divided into two types:-

- Input Restricted Deque :- In this, input is blocked at a single end but allows deletion at both the ends.
- Output Restricted Deque :- In this, output is blocked at a single end but allows insertion at both the ends.

## Applications of Double Ended Queue

The applications of double ended queue are:-

- To execute undo and redo operation.
- For implementing stacks.
- Storing the history of web browsers.

# Dequeue
# (Double ended Queue)

- In Double Ended Queue, insert and delete operation can be occur at both ends that is front and rear of the queue

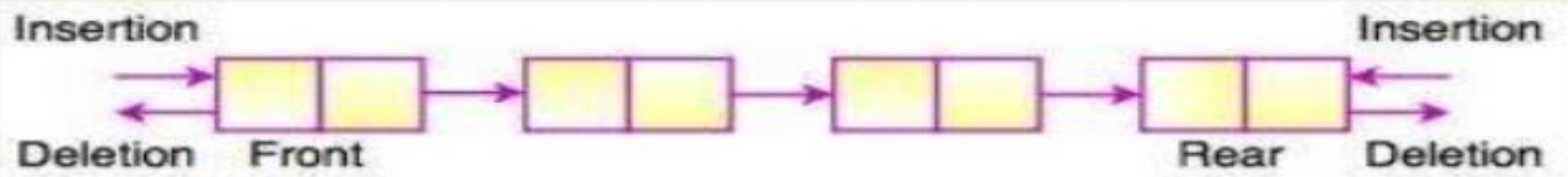
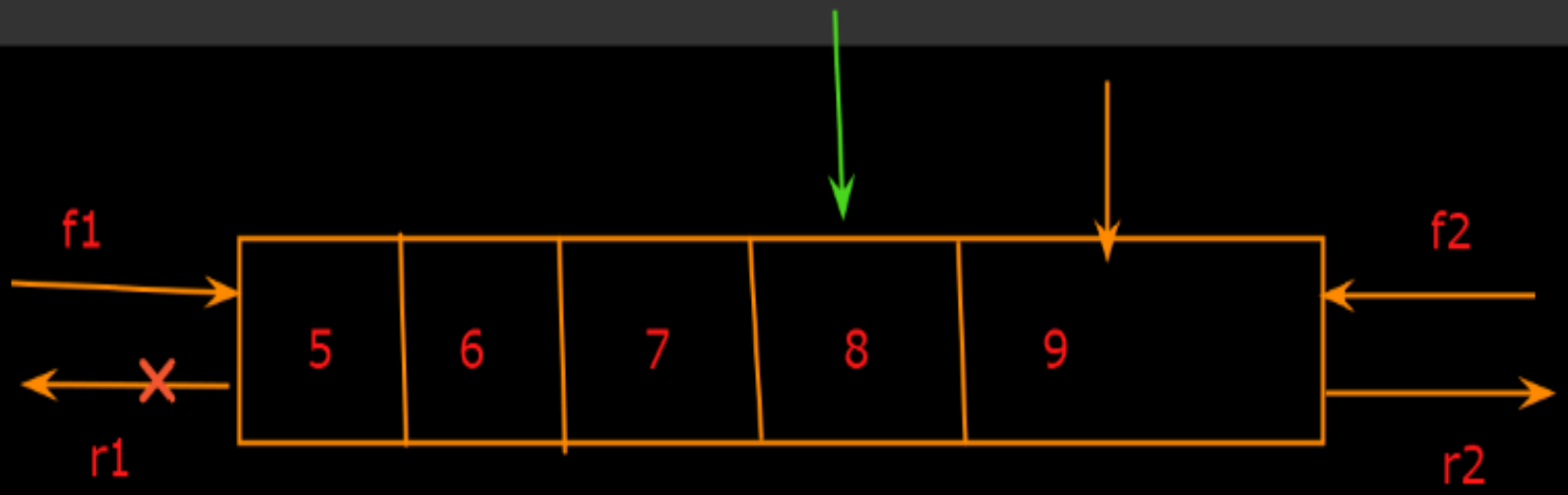



Fig. Double Ended Queue (Dequeue)

```
insertfront()
insertrear()
--------------
deletefront()
deleterear()
```




((front == 0 && rear == size-1) || front == rear + 1)  :Deque full

front == -1 || rear == -1  : Deque is empty

# Thanks