Fundamentals of Programming CCS1063/CSE1062

Lecture 12 –Introduction to Data Structures

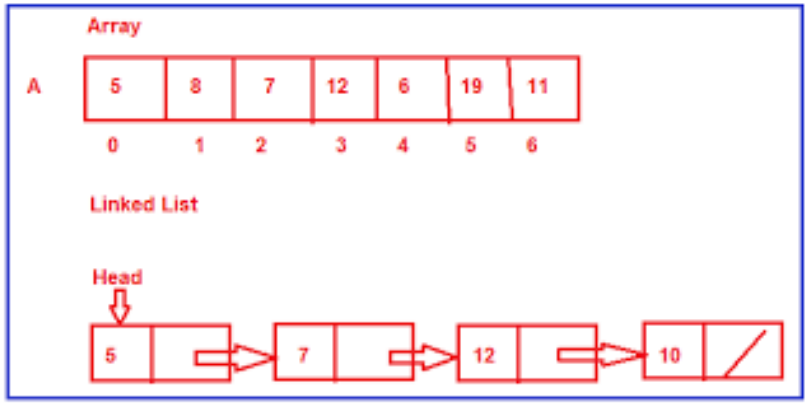
Professor Noel Fernando

UCSC

What is data structures?

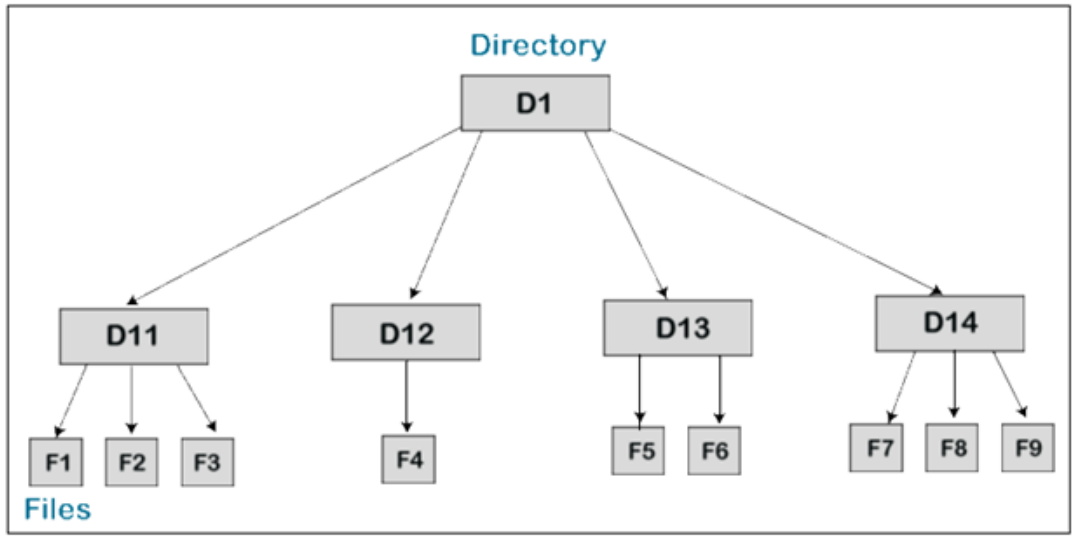
What is data structures?

- Data Structure can be defined as the group of **data elements** which provides an **efficient** way of **storing and organizing data** in the computer so that it can be **used efficiently**.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.



What is data structures?

- Data Structures are widely used in almost every aspect of Computer Science
- i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.



Examples of data structures

1. Stack
2. Queue
3. Linked List
4. Tree
5. Hash Table
6. Priority Queue
7. Heap
8. Binary Search Tree

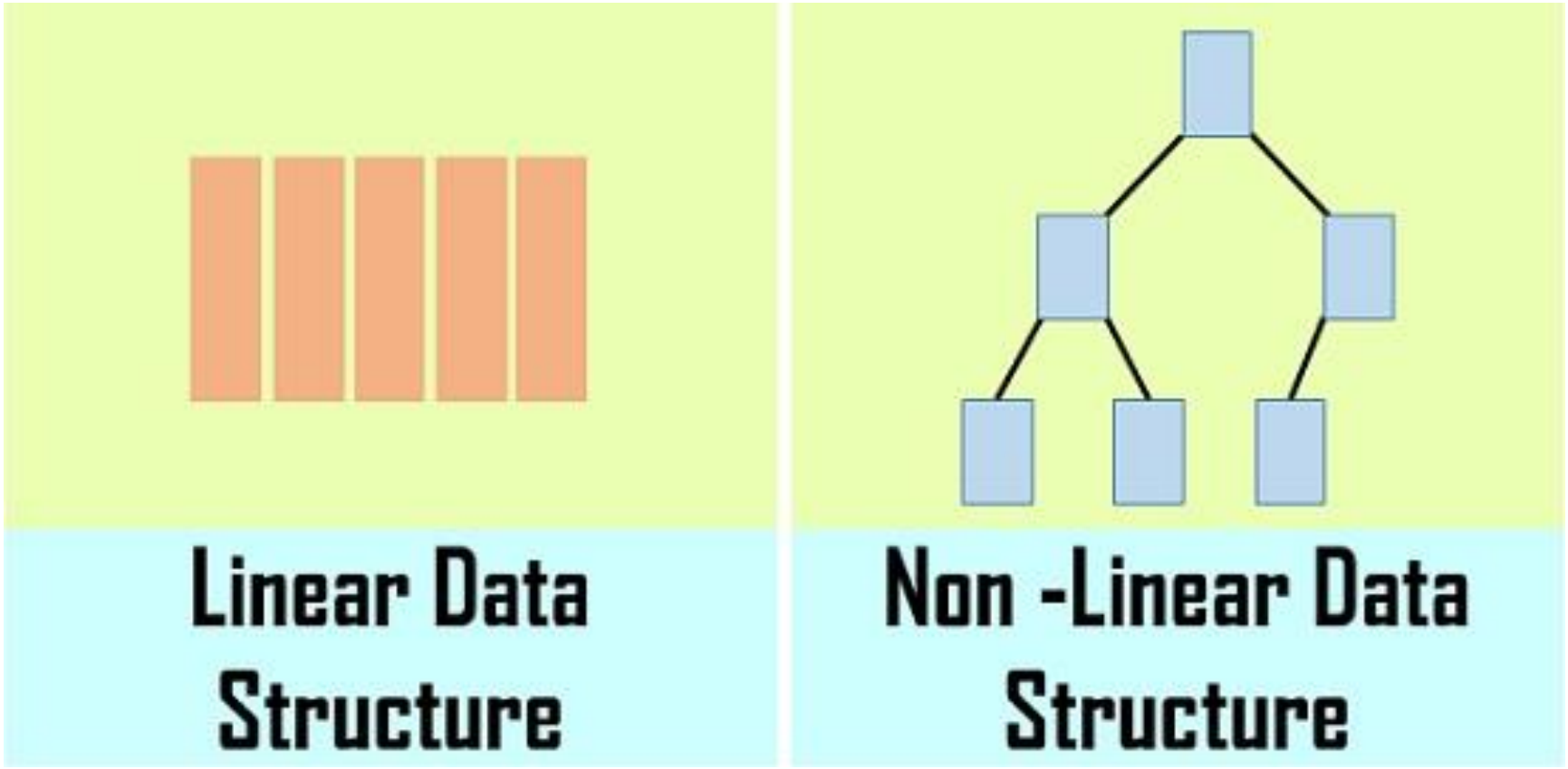
Categories of data structure

- Data structures can be subdivided into two major types.
- Linear Data structure
- Non-Linear Data structure

- Linear Data structure : A data structure is said to be linear if its elements combine to form any specific order.
- E.g. Arrays and Linked List.

- Non-Linear Data structure : This structure mainly represents data with a hierarchical relationship between different elements.
- E.g. Trees and Graphs

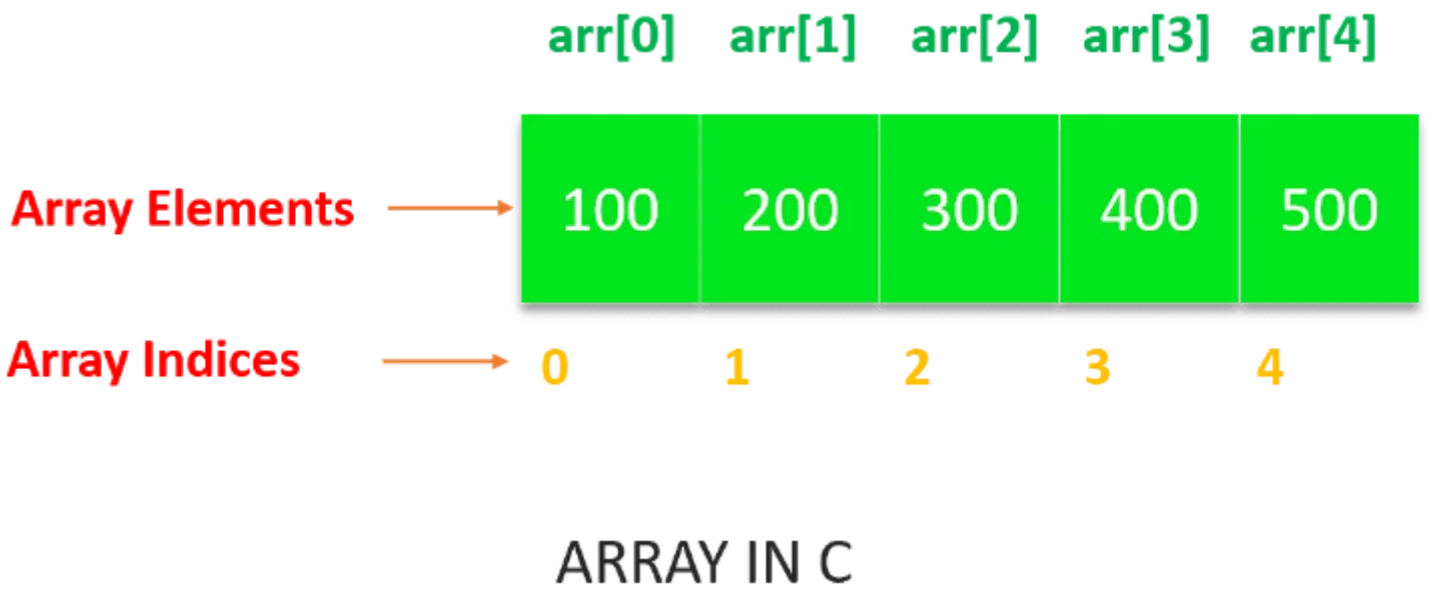
Difference between Linear and Non-linear Data Structures



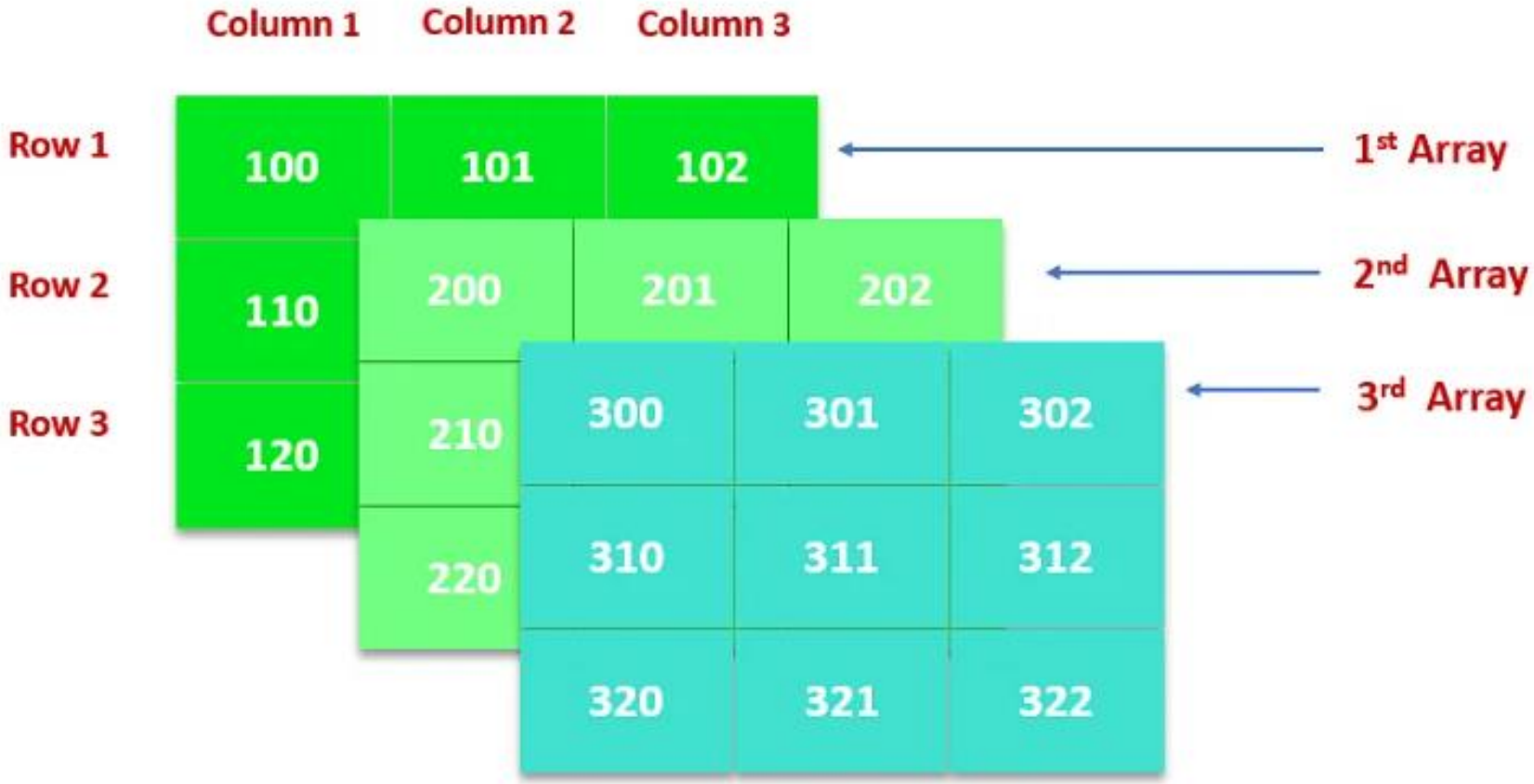
Properties Linear and Non-linear Data Structures

BASIS FOR COMPARISON	LINEAR DATA STRUCTURE	NON-LINEAR DATA STRUCTURE
Basic	The data items are arranged in an orderly manner where the elements are attached adjacently.	It arranges the data in a sorted order and there exists a relationship between the data elements.
Traversing of the data	The data elements can be accessed in one time (single run).	Traversing of data elements in one go is not possible.
Ease of implementation	Simpler	Complex
Levels involved	Single level	Multiple level
Examples	Array, queue, stack, linked list, etc.	Tree and graph.
Memory utilization	Ineffective	Effective

1-D array


arr[0]
arr[1]
arr[2]
arr[3]
arr[4]
Array Elements
100
200
300
400
500
Array Indices
0
1
2
3
4
ARRAY IN C

Multi-dimensional Arrays



An array of structure

- As example , we can consider a ***S*** is array of structure.
- The size of the array is 10
- It could store information of **10** different variables of type *student*.
- So we don't need to take **10** different variables instead we could use array of structure *student*.

An array of structure

Structure student S[10] ;

	Name	Class	Roll_number	Marks [5]
S[0]	S[0].name	S[0].class	S[0].roll_number	S[0].marks[5]
S[1]	S[1].name	S[1].class	S[1].roll_number	S[1].marks[5]
⋮	⋮	⋮	⋮	⋮
S[9]	S[9].name	S[9].class	S[9].roll_number	S[9].marks[5]

Question

- Write a C program to read the following information and display the it on the screen.

Name	Roll No	Class	Marks				
Anuradha	1	A	100.00	98.00	97.00	99.00	87.00
Sarath	2	A	89.00	92.00	98.00	87.00	78.00

Answer

```
#include<stdio.h>
struct student {
  char name[50];
  char Class[100];
  int roll_number;
  float marks[5];
};
```

Answer

```
int main() {
  struct student s[2];
  for (int i = 0; i < 2; i++) {
    printf("\nEnter details of student %d\n", i + 1);
    printf("Enter name: ");
    scanf("%s", s[i].name);
    printf("\nEnter roll no: ");
    scanf("%d", &s[i].roll_number);
    printf("\nEnter class: ");
    scanf("%s", s[i].Class);
    for (int j = 0; j < 5; j++) {
      printf("\nEnter the marks in subject %d (out of 100): ", j + 1);
      scanf("%f", &s[i].marks[j]);
    }
    printf("\n");
  }
```

Answer

```
printf("\n");
  printf("Name\t\tRoll no\t\t\tClass\t\t\t\tMarks\n");
  for (int i = 0; i < 2; i++) {
    printf("%s\t\t%d\t\t\t%s\t\t",
      s[i].name, s[i].roll_number, s[i].Class);
    for (int j = 0; j < 5; j++) {
      printf("%.2f\t", s[i].marks[j]);
    }
    printf("\n");
  }
   return 0;
}
```

Basic Operations in the Arrays

- Following are the basic operations supported by an array.
- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.
- **Display** − Displays the contents of the array.

1) Traversing

- Traversing an Array means going through each element of an Array exactly once.
- We start from the first element and go to the last element

1) Traversing

```
#include <stdio.h>
void main() {
int array[] = {1,2,3,4,5};
int i, n = 5;
printf(" The array elements are: \n " );
for( i=0;i < n; i++) {
 printf(" array[%d] = %d \n " , i, array[i] );
}
}
```

```
The array elements are:
 array[0] = 1
 array[1] = 2
 array[2] = 3
 array[3] = 4
 array[4] = 5
```

(2) Searching

- The search operation finds a particular data item or element in an Array.
- We can search in an unsorted array with the help of traversal of the Array.
- The linear traversal from the first element to the last element can be used to search if a given number is present in an Array and can also be used to find its position if present.

Searching

```
#include<stdio.h>
int findElement(int arr[], int n, int key) {
 int i; for (i = 0; i < n; i++)
if (arr[i] == key
return i;
return -1; }
```

Searching

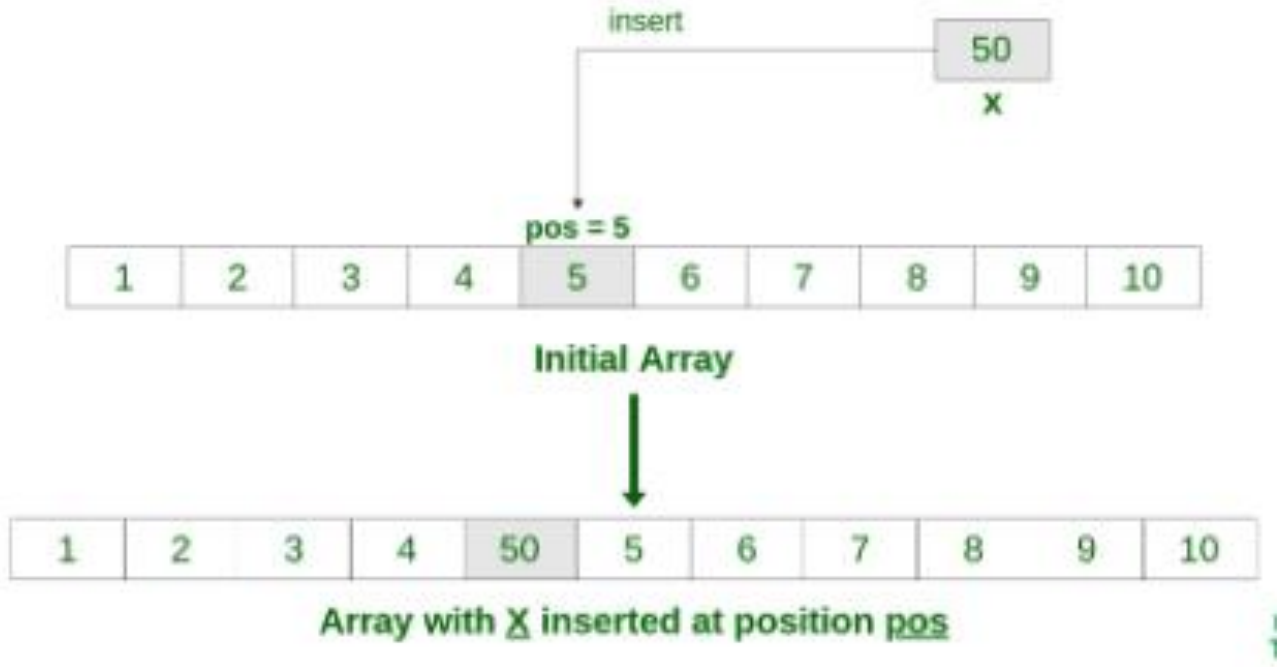
```
int main() {
 int arr[] = {1, 4, 0, 6, 3};
int n = sizeof(arr) / sizeof(arr[0]);
 int key = 4;
int position = findElement(arr, n, key);
 if (position == - 1)
printf("Element not found");
else printf("Element Found at Position: %d", position + 1 );
return 0;
}
```

Output:

```
Element Found at Position: 2
```

How to Insert an element at a specific position

- Given an array **arr** of size **n**,
- insert an element **x** in this array **arr** at a specific position **pos**.



OUTPUT

```
Initial Array:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Array with 50 inserted at position 5:
[1, 2, 3, 4, 50, 5, 6, 7, 8, 9, 10]
```

How to Insert an element at a specific position

- First get the element to be inserted, say **x**
- Then get the position at which this element is to be inserted, say **pos**
- Create a **new array** with the size one greater than the previous size
- Copy all the elements from previous array into the new array till the position pos
- Insert the element x at position pos
- Insert the rest of the elements from the previous array into the new array after the pos.

3) Deletion - Remove a specific element from array

- **Input**
- Array : {1, 20, 5, 78, 30}
- Element : 78
- **Output**
- Array : {1, 20, 5, 30}

3) Deletion - Remove a specific element from array

Algorithm

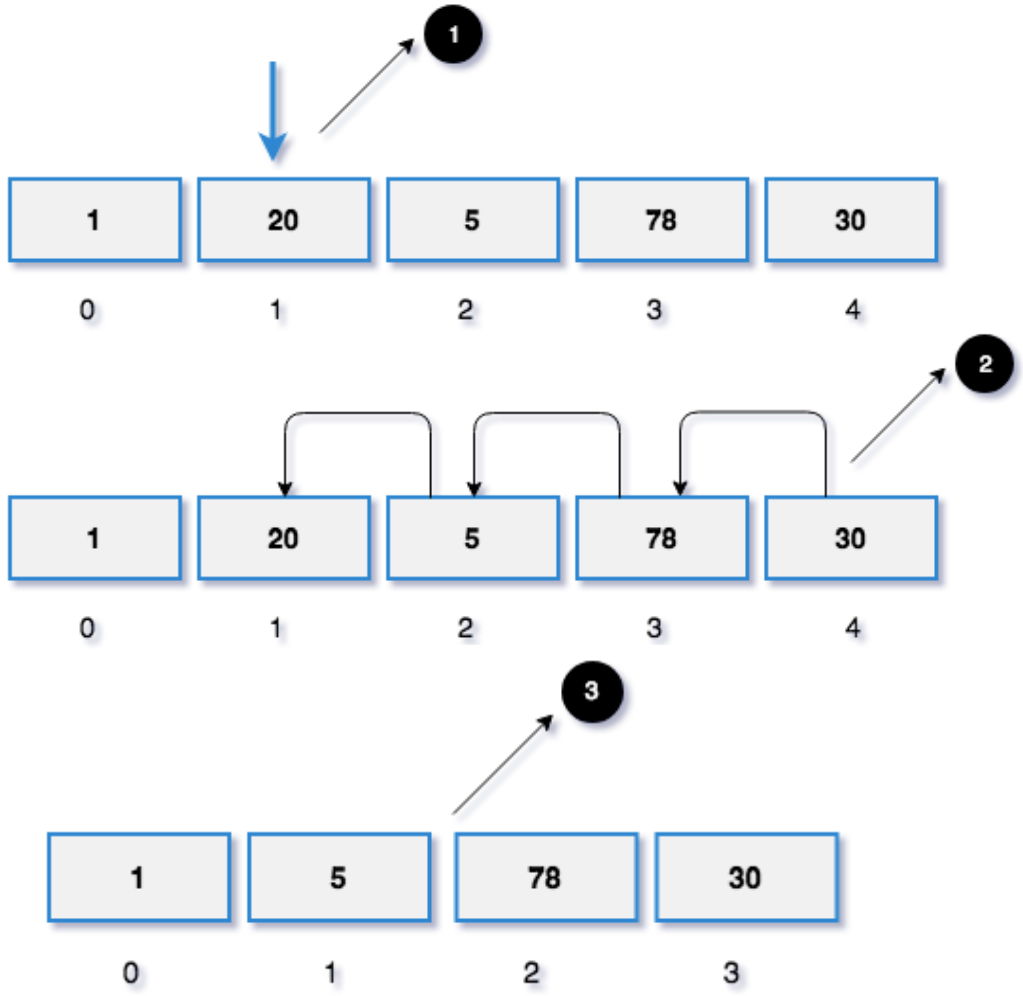
1. Find the given element in the given array and note the index.
2. If the element found,

   Shift all the elements from index + 1 by 1 position to the left.

   Reduce the array size by 1.
3. Otherwise, print "Element Not Found"

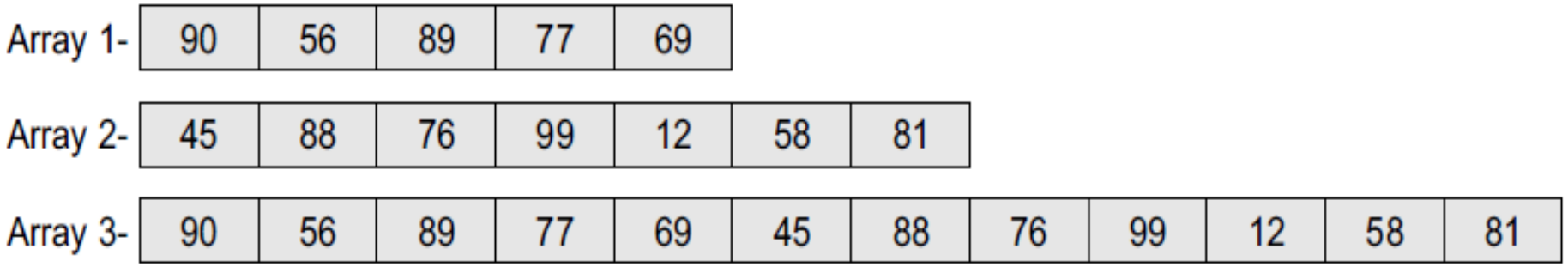
3) Deletion - Remove a specific element from array

- Visual Representation
- Let's take an array of 5 elements.
- 1, 20, 5, 78, 30.
- If we remove element 20 from the array, the execution will be,

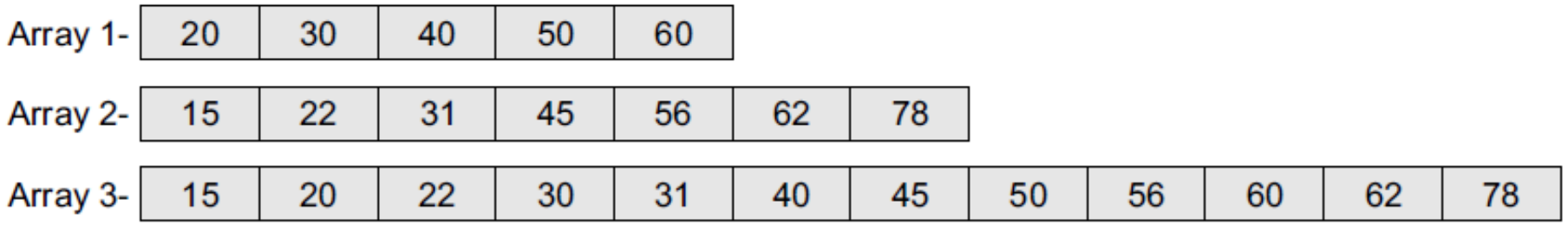


Merging Two Arrays

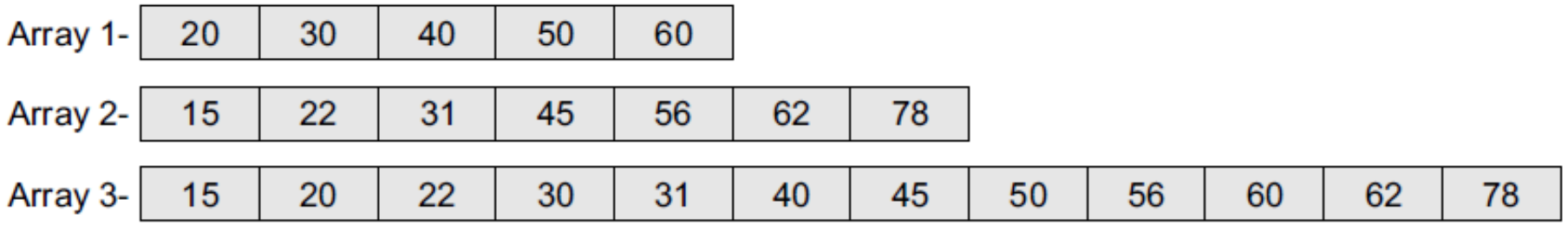
- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array.



Merging of two sorted arrays

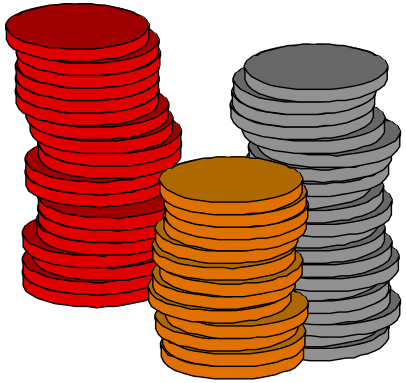


Merging of two sorted arrays



STACK

- A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.
- Stacks are sometimes known as LIFO (last in, first out) lists.



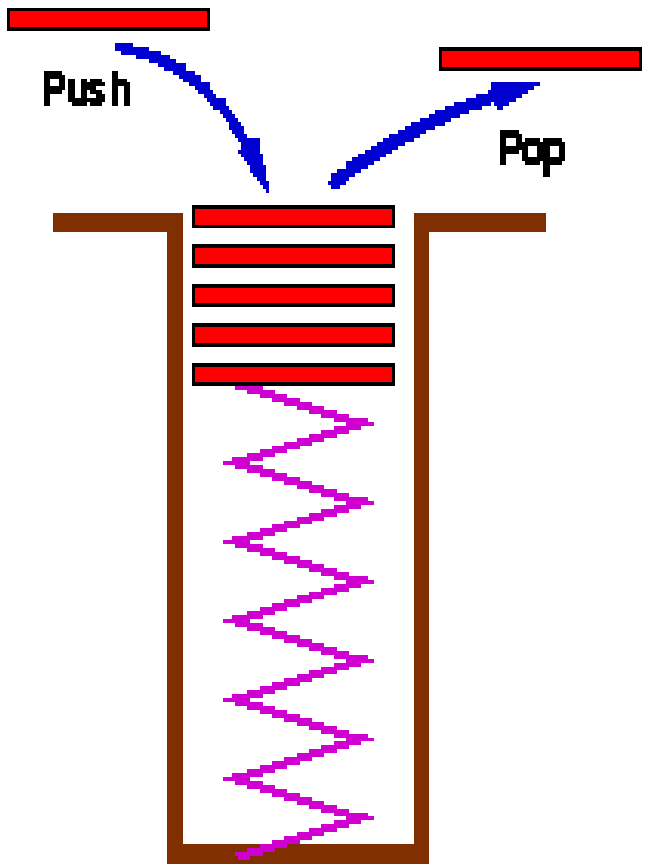
Stacks

- More examples from the real world?
- Stack of plates in a buffet table
- Stack of chairs
- The tennis balls in their container
- stack of trays in a criteria
- Stack of Books, Clothes piled

What is a stack?

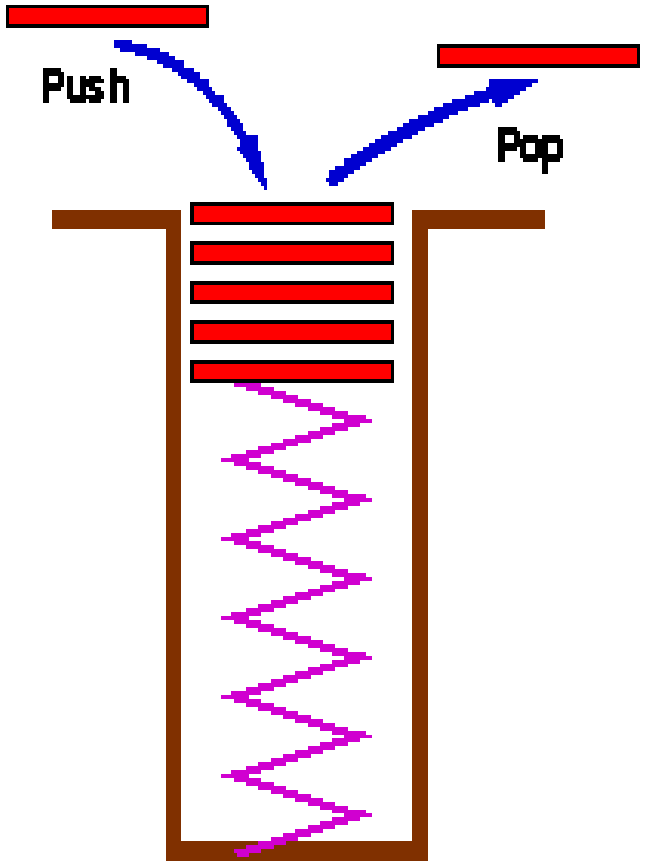
- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- (LIFO)
- It means: the last element inserted is the first one to be removed

Examples



- A common model of a stack is a plate or coin stacker. Plates are "pushed “onto to the top and "popped" off from the top.

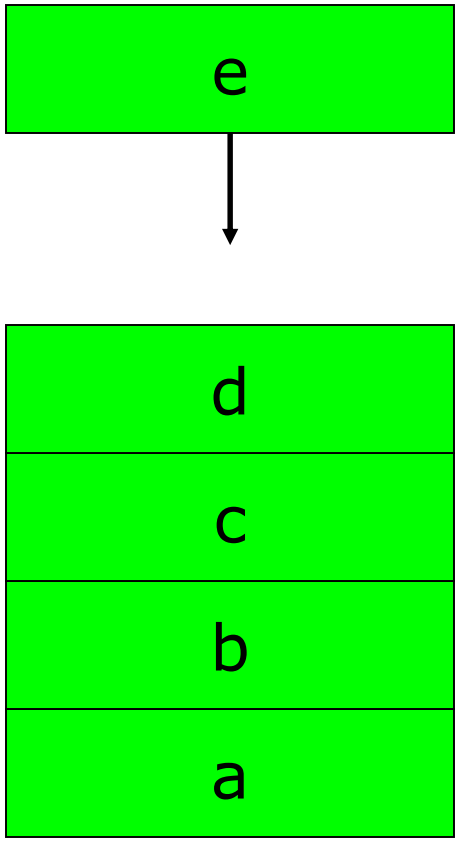
Examples (Cont.)



- Stacks form Last-In-First-Out (LIFO) queues.

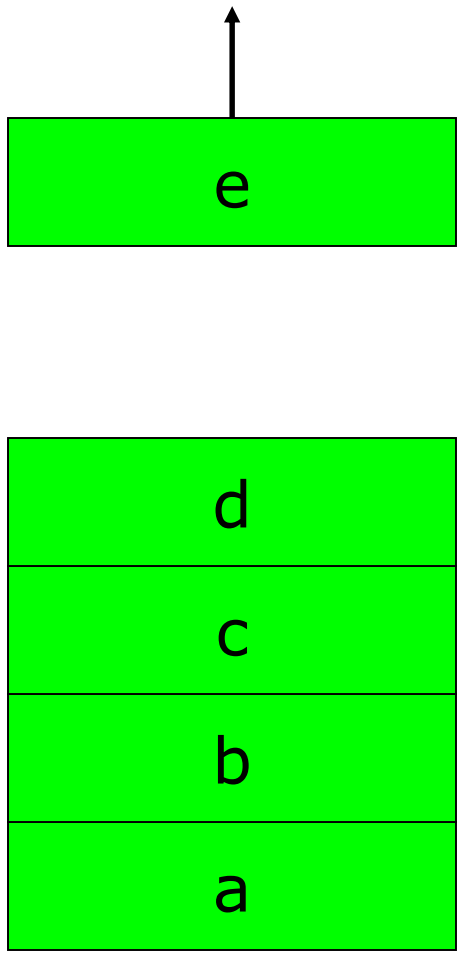
Stacks are LIFO

Push operations:



Stacks are LIFO

Pop operation:



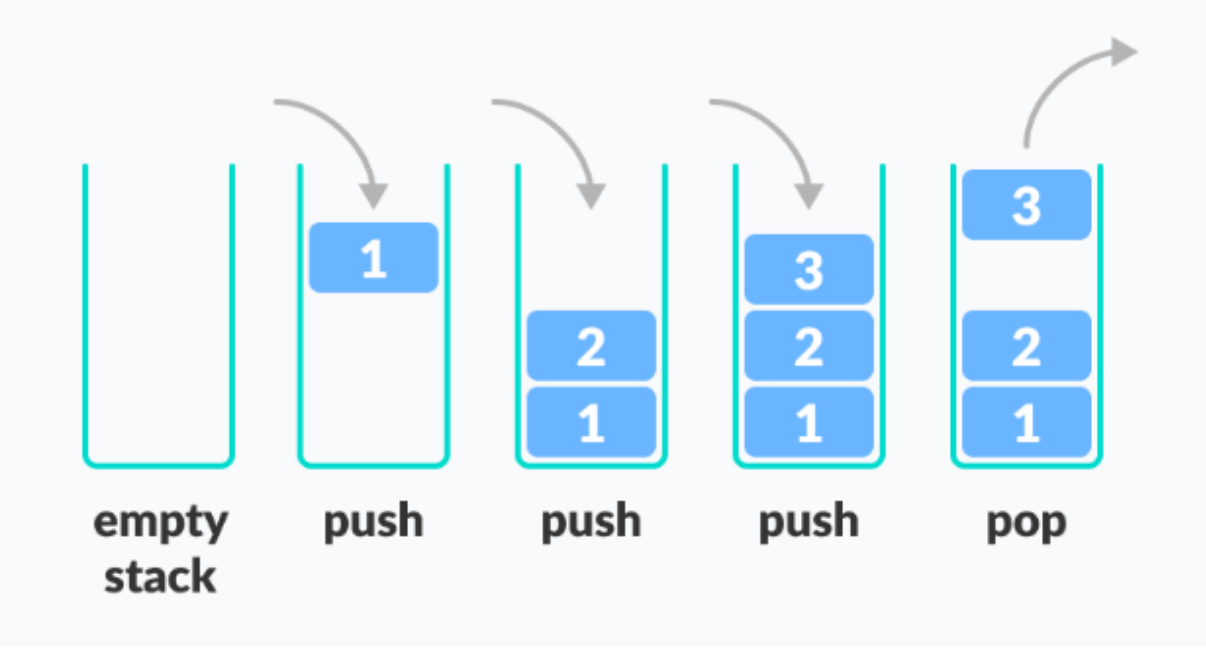
Last element that was pushed is the first to be popped.

Stacks

- *There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle*

Linear Data Structure

-E.g. Stack



Stack applications

- “Back” button of Web Browser
  - History of visited web pages is pushed onto the stack and popped when “back” button is clicked
- “Undo” functionality of a text editor
- Reversing the order of elements in an array
- Saving local variables when one function calls another, and this one calls another, and so on.

Stacks

- A stack is a linear data structure that can be accessed only at one of its ends for storing and retrieving data.
- There are two ways of implementing a stack : Array (Static) and linked list (dynamic).

Basic operations on Stacks

- A stack is an object or more specifically an ADT that allows the following operations:

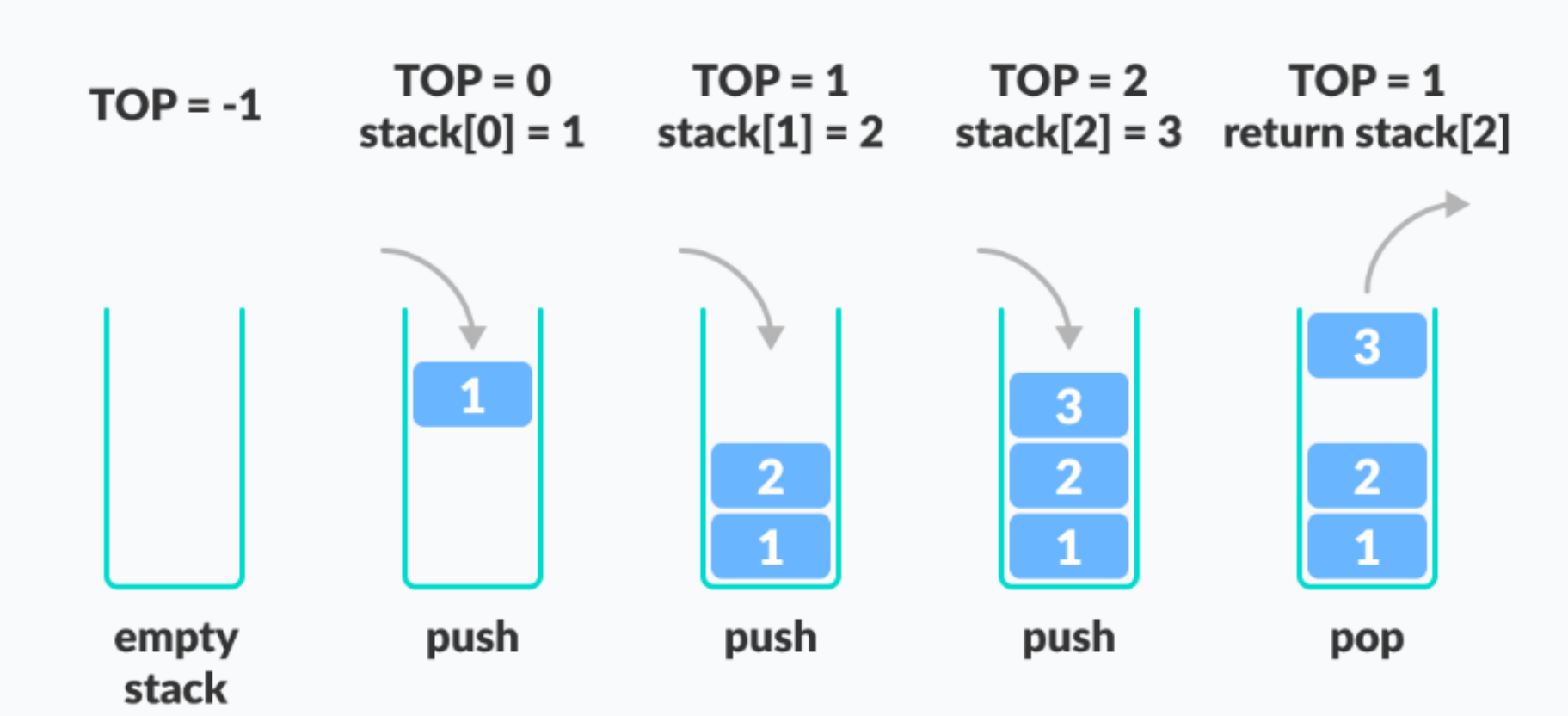
  - `Push` : Add an element to the top of a stack
  - `Pop` : Remove an element from the top of a stack
  - `IsEmpty` : Check if the stack is empty
  - `IsFull` : Check if the stack is full
  - `Peek` : Get the value of the top element without removing it

How a Stack Works

- The operations work as follows:

1. A pointer called `TOP` is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of `TOP` and place the new element in the position pointed to by `TOP`.
4. On popping an element, we return the element pointed to by `TOP` and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty

How a Stack Works



Stack Implementation

- Implementation can be done in two ways
  - Static implementation
  - Dynamic Implementation
- Static Implementation
  - Stacks have **fixed size**, and are implemented as **arrays**
  - It is also inefficient for utilization of memory
- Dynamic Implementation
  - Stack **grow in size** as needed, and implemented as **linked lists**
  - Dynamic Implementation is done through pointers
  - The memory is efficiently utilize with Dynamic Implementations

Stack-Related Terms

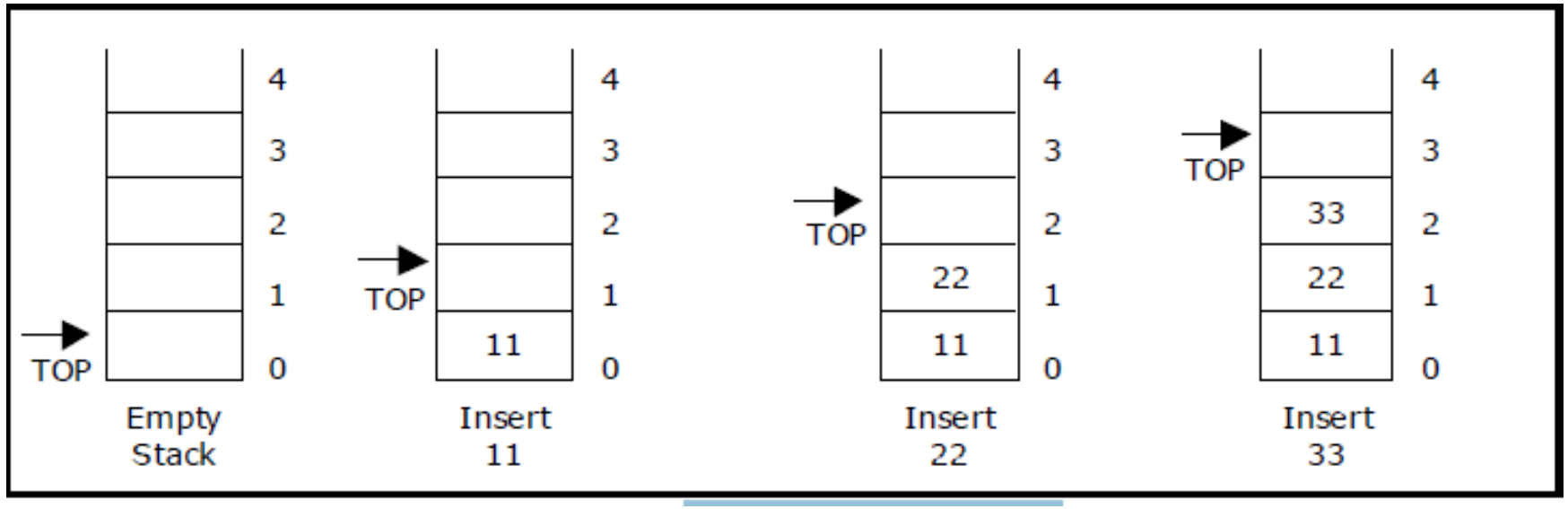
- Top
  - A pointer that points the top element in the stack.
- Stack Underflow
  - When there is no element in the stack, the status of stack is known as stack underflow.
- Stack Overflow
  - When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as stack overflow

Representation of Stack

- Consider a stack with 6 elements capacity, This is called as the size of the stack.
- The number of elements to be added should not exceed the maximum size of the stack.
- If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition.
- Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition

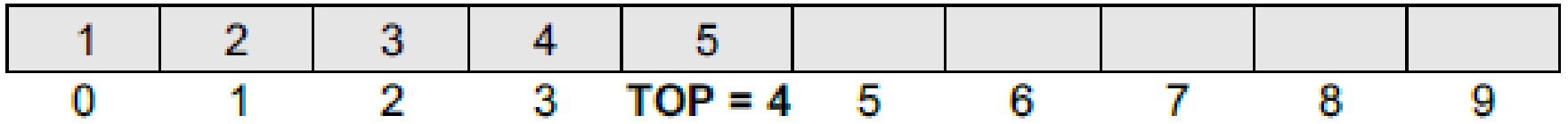
Example : Push(Insert) operations on stack -push()

Push(11)
Push(22)
Push(33)



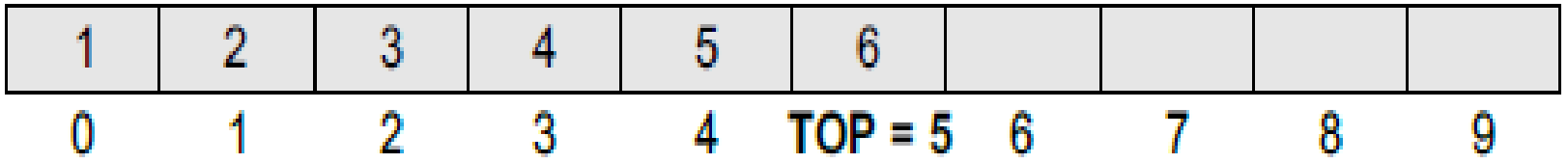
Algorithm to insert an element in a stack

E.g.1 Consider the following stack.



To insert an element with value 6, we first check if TOP=MAX–1.

If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].



Algorithm to insert an element in a stack

Example 2

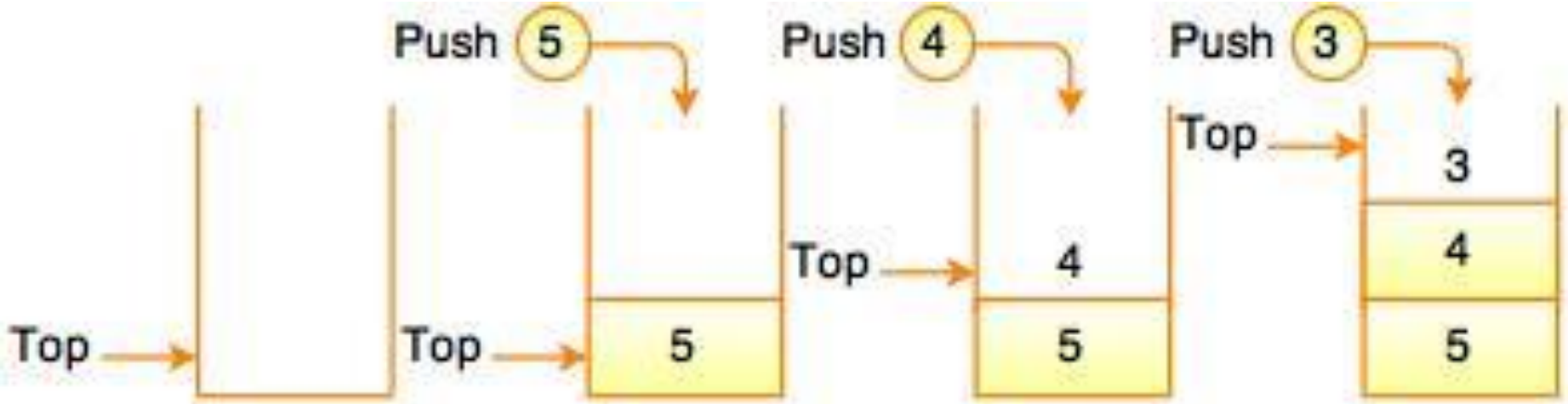


Fig. Insertion of Elements in a Stack

Algorithm to insert an element in a stack

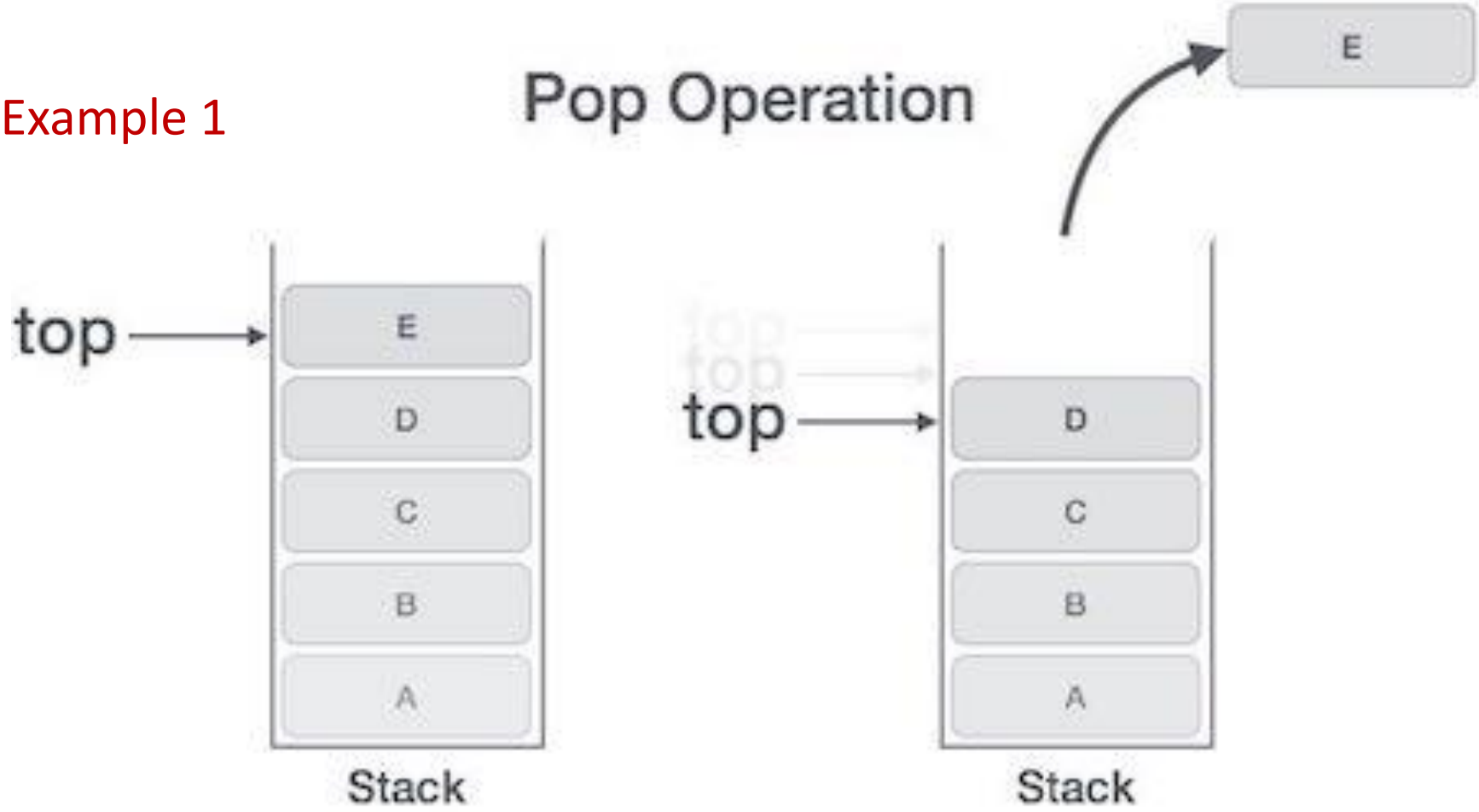
```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means 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.

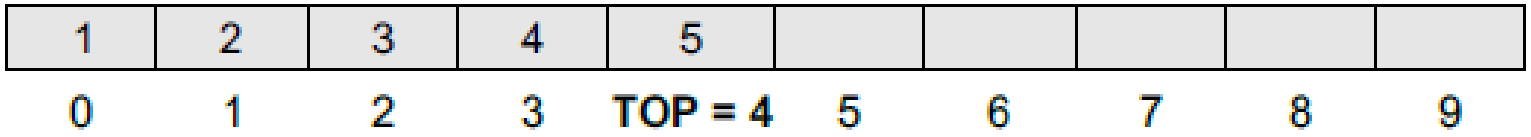
Pop Operation

Example 1

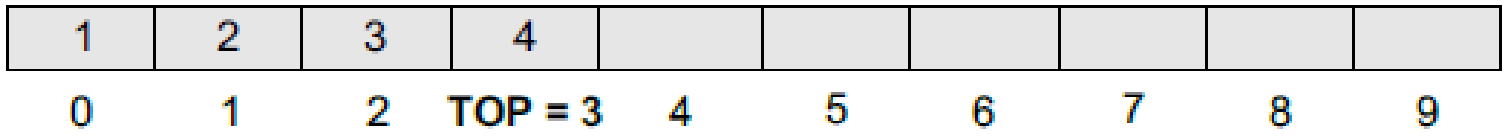


Pop Operation

- Example 2
- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP.

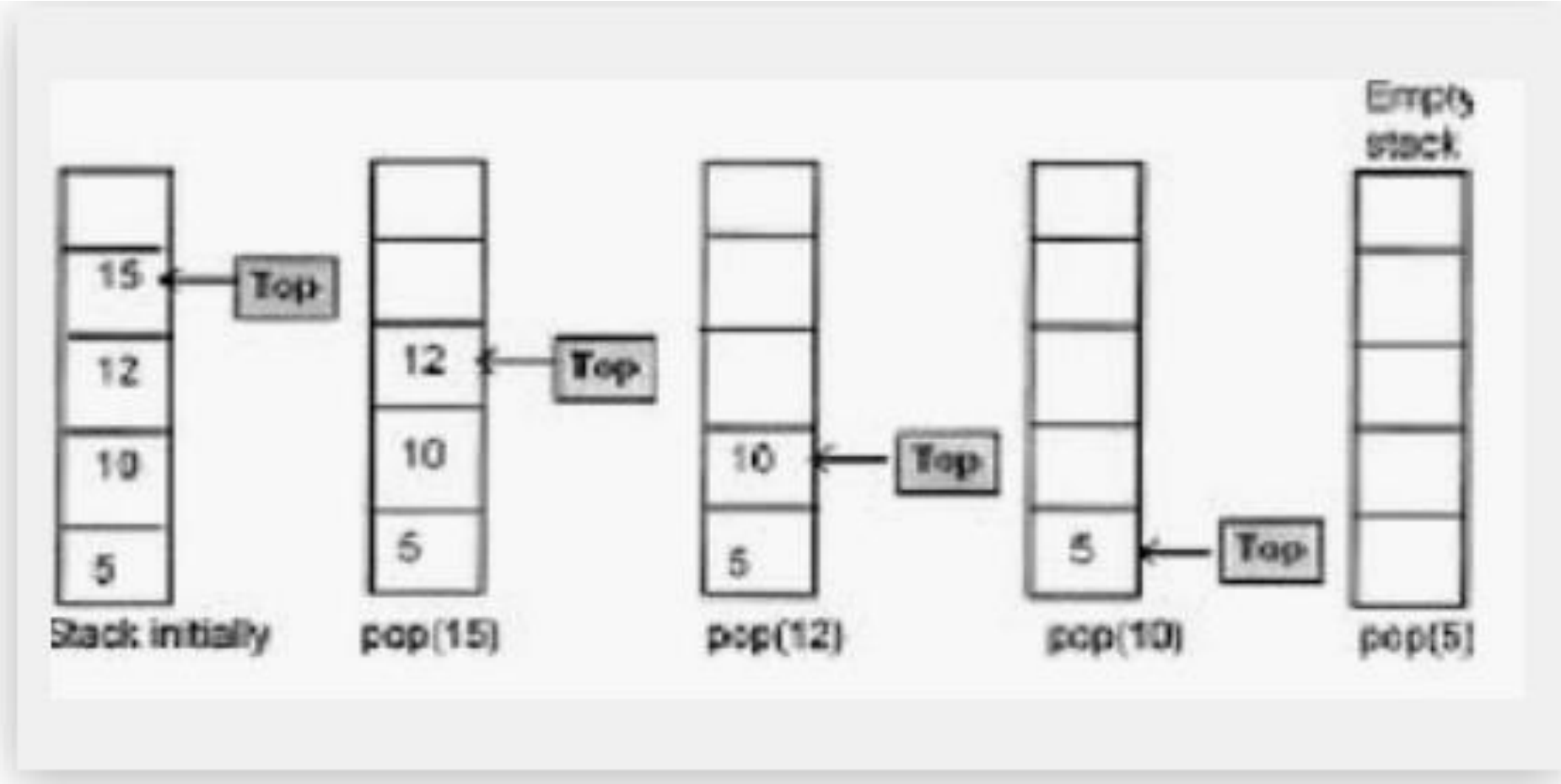


- Thus, the updated stack becomes as shown in below



Pop operations on stack

When an element is taken off from the stack, the operation is performed by pop().



Pop Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

Peek Operation

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

isempty()-Pseudocode

isEmpty() – check if stack is empty

```
Begin Procedure IsEmpty
        If top is less than 1
                return Ture
        else
                return False
        endif
End Procedure
```

Isempty using C

Implementation of isempty() function in C programming language is slightly different.

We initialize top at -1, as the index in array starts from 0.

```
Boolean Isempty()
{
        if (top==-1)
                return true;
        else
                return false;
}
```

Pesudocode Algorithm of isfull() function

```
Begin Procedure IsFull
        If top is equal to MAXSIZE
                return true
        else
                return false
        endif
End procedure
```

C code of isfull() function

Implementation of isfull() function in C programming language

```
Boolean isfull()
{
If (top=MAXSIZE-1)
return true
else
return false
}
```

Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created

int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);
```

Write a program to perform Push, Pop, and Peek operations on a stack. Cont....

```
int main(int argc, char *argv[]) {
        int val, option;
        do
        {
                printf("\n *****MAIN MENU*****");
                printf("\n 1. PUSH");
                printf("\n 2. POP");
                printf("\n 3. PEEK");
                printf("\n 4. DISPLAY");
                printf("\n 5. EXIT");
                printf("\n Enter your option: ");
                scanf("%d", &option);
                switch(option)
```

Write a program to perform Push, Pop, and Peek operations on a stack. Cont....

```
{
case 1:
        printf("\n Enter the number to be pushed on stack: ");
        scanf("%d", &val);
        push(st, val);
        break;
case 2:
        val = pop(st);
        if(val != -1)
        printf("\n The value deleted from stack is: %d", val);
        break;
case 3:
        val = peek(st);
        if(val != -1)
```

Write a program to perform Push, Pop, and Peek operations on a stack. Cont....

```
                        printf("\n The value stored at top of stack is: %d", val);
                        break;
                case 4:
                        display(st);
                        break;
                }
        }while(option != 5);
        return 0;
}
void push(int st[], int val)
{
        if(top == MAX-1)
        {
                printf("\n STACK OVERFLOW");
        }
        else
        {
                top++;
                st[top] = val;
        }
}
```

Write a program to perform Push, Pop, and Peek operations on a stack. Cont....

```
int pop(int st[])
{
        int val;
        if(top == -1)
        {
                printf("\n STACK UNDERFLOW");
                return -1;
        }
        else
        {
                val = st[top];
                top--;
                return val;
        }
}
```

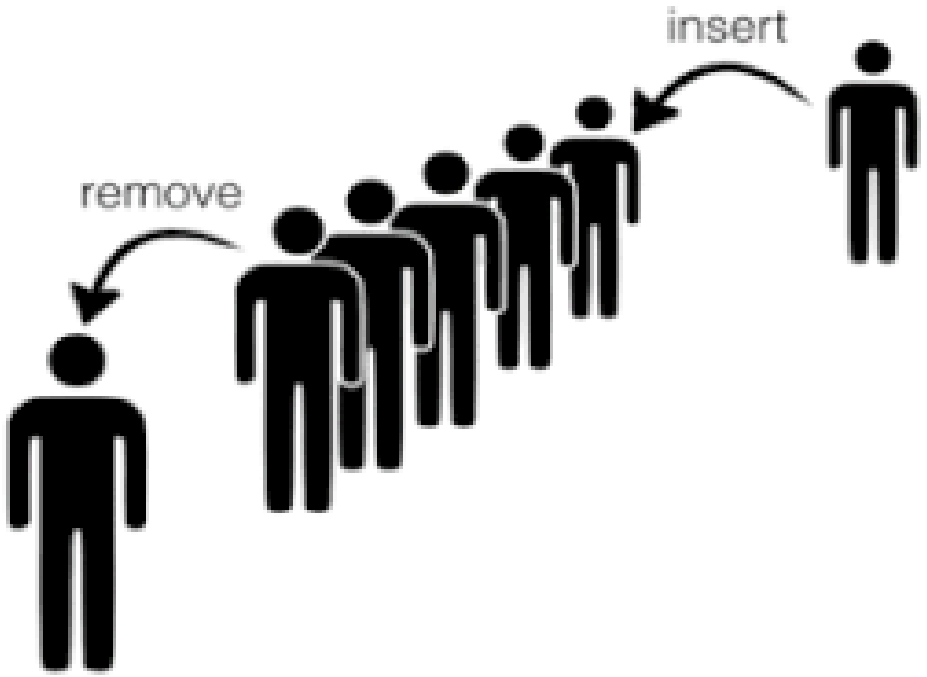
Write a program to perform Push, Pop, and Peek operations on a stack. Cont....

```
void display(int st[])
{
        int i;
        if(top == -1)
        printf("\n STACK IS EMPTY");
        else
        {
                for(i=top;i>=0;i--)
                printf("\n %d",st[i]);
                printf("\n"); // Added for formatting purposes
        }
}
int peek(int st[])
{
        if(top == -1)
        {
                printf("\n STACK IS EMPTY");
                return -1;
        }
        else
        return (st[top]);
}
```

What is a Queue?

What is a Queue?

Queues



Queues

- A queue is another special kind of list, where items are inserted at one end called the rear(back) and deleted at the other end called the front.
- Another name for a queue is a “FIFO” or “First-in-first-out” list

The queue data structure

- A queue is used in computing in much the same way as it is used in every day life: allow a sequence of items to be processed on a **first-come-first-served basis.**

- In most computer installations, for example, one printer is connected to several different several machines. So that more than one user can submit printing jobs to the same printer, it maintains a queue.

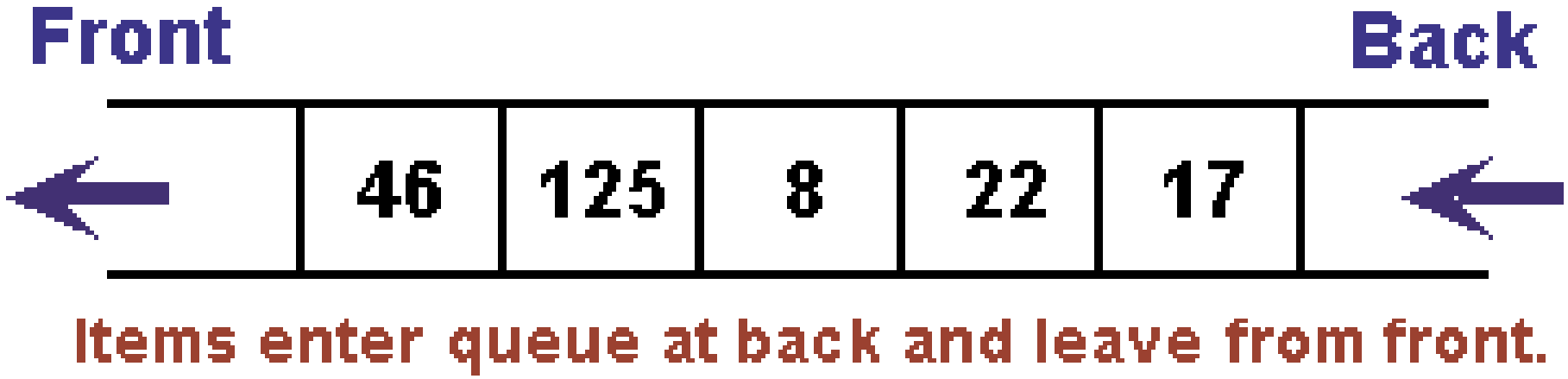
Queues in computer science

- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Abstract data type: Queue

- **queue**: a more restricted List with the following constraints:
  - elements are stored by order of insertion from *front* to *back*
  - items can only be added to the *back* of the queue
  - only the *front* element can be accessed or removed

- goal: every operation on a queue should be O(1)



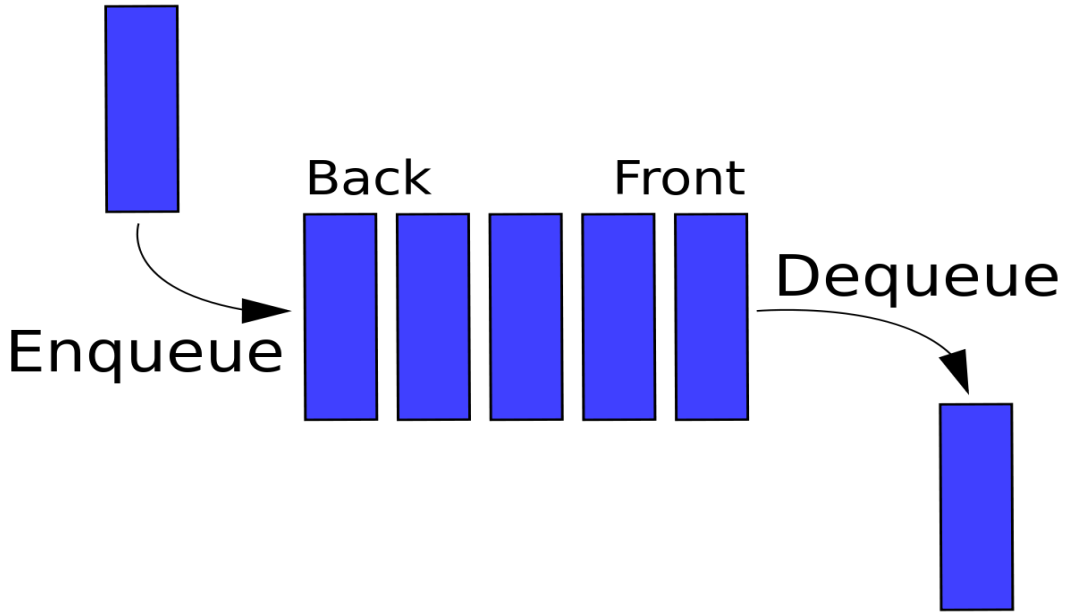
ARRAY REPRESENTATION OF QUEUES

- The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning.
- Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

Main Operations on a queue

enqueue: which inserts an element at the end of the queue (rear).

dequeue: which deletes an element at the start of the queue (front)

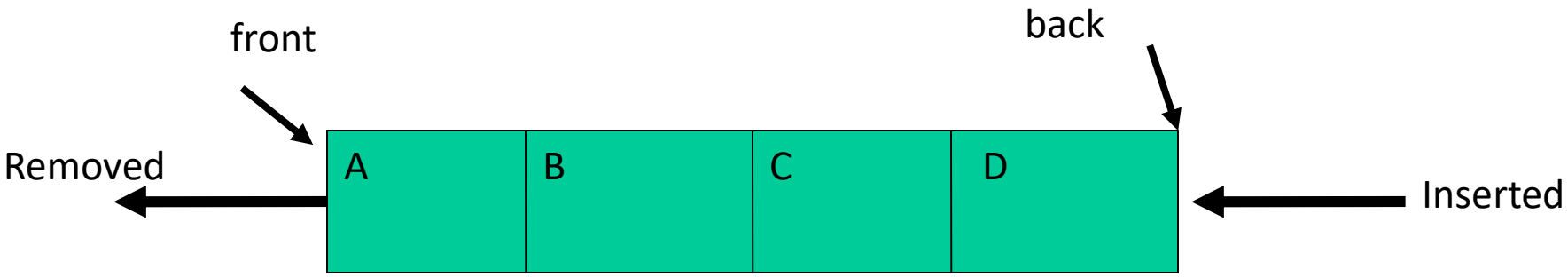


Components of a queue

- **Front** is a variable which refers to first position in queue.
- **Rear** is a variable which refers to last position in queue.
- **Element** is component which has data.
- **MaxQueue** is variable that describes maximum number of elements in a queue.

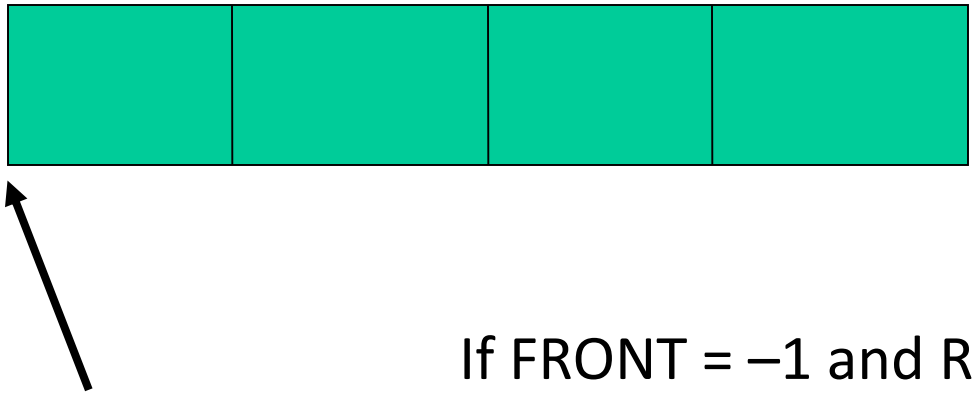
Queue Definition

- A queue is an ordered collection of items from which items may be deleted at one end (called front of the queue) and into which items may be inserted at the other end (called the rear of the queue)



Queue initialisation

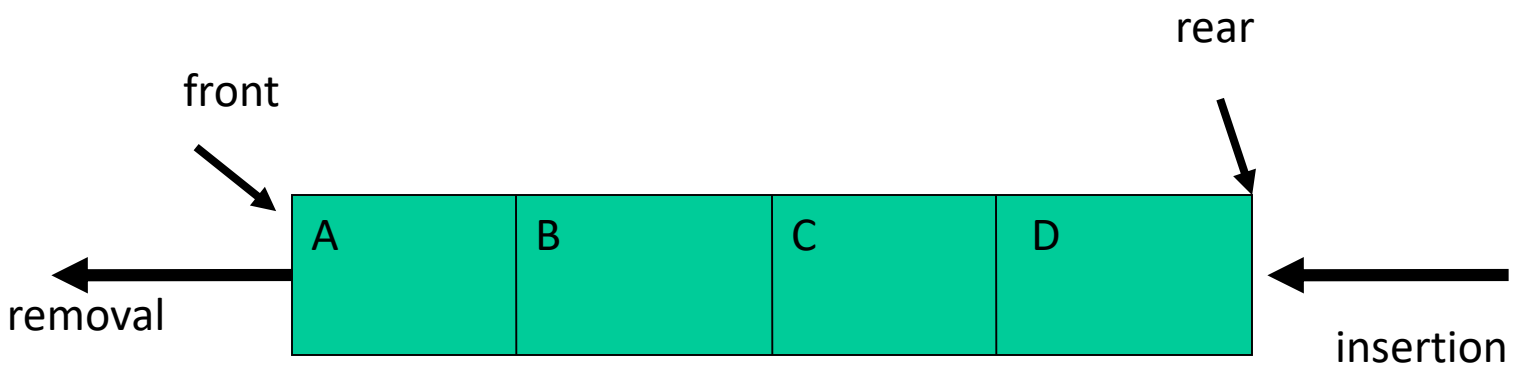
- For an empty queue



Front, Rear

If FRONT = –1 and REAR = –1, it means there is no element in the queue

Basic Array implementation of queues (general situation)

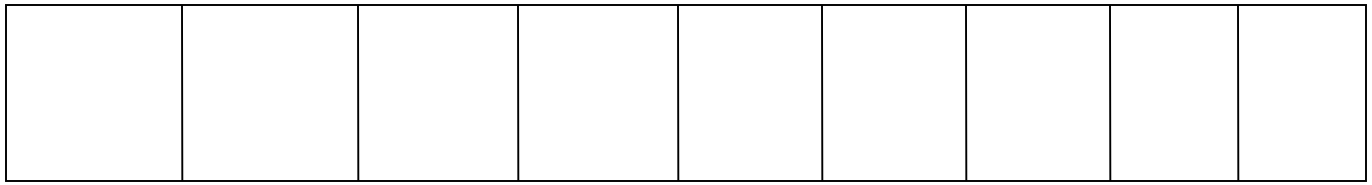


After insertion of A,B,C and D

Front=0 and rear=3

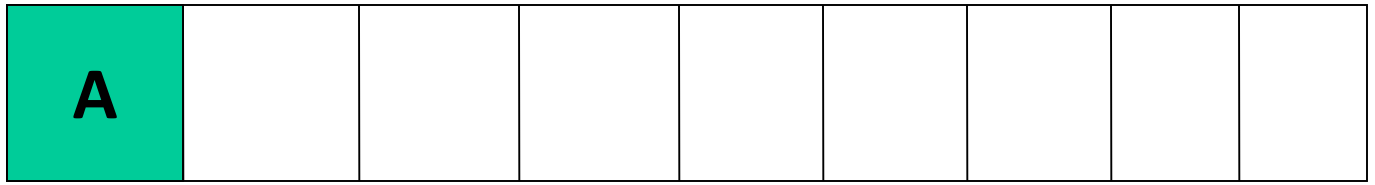
Array implementation of queues

Eg:



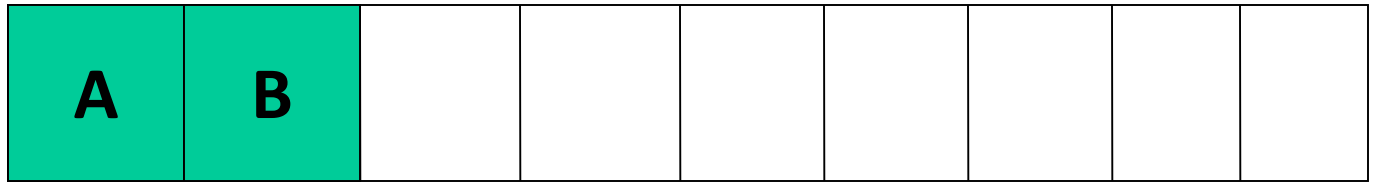
If FRONT = –1 and REAR = –1, it means there is no element in the queue

Enqueue(A)



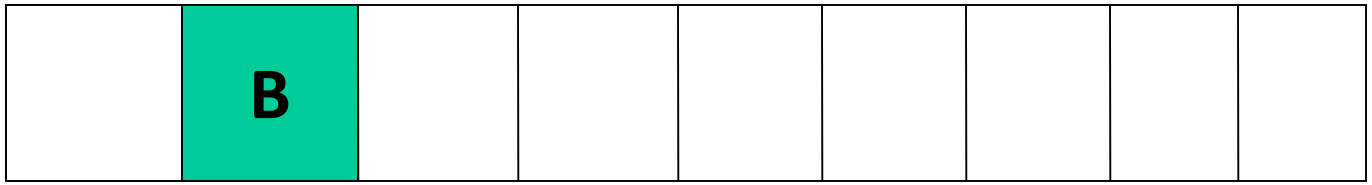
FRONT = 0 and REAR = 0

Enqueue(B)



FRONT = 0 and REAR = 1

Dequeue()



FRONT=1, REAR=1

Array implementation of queues Cont.……

- **B**efore inserting an element in a queue, we must check for overflow conditions.
- An overflow will occur when we try to insert an element into a queue that is already full(REAR = MAX – 1)
- Before deleting an element from a queue, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a queue that is already empty.
- If FRONT = –1 and REAR = –1, it means there is no element in the queue.

Declaration of arrays

Const

Maxqueue=Value { value is a integer number}

Type

- Queue_array=array[1.. Maxqueue] of datatype
- Queue : Queue_array
- Front, Rear: integer (pointer of the queue)

Declaration of arrays

Example :

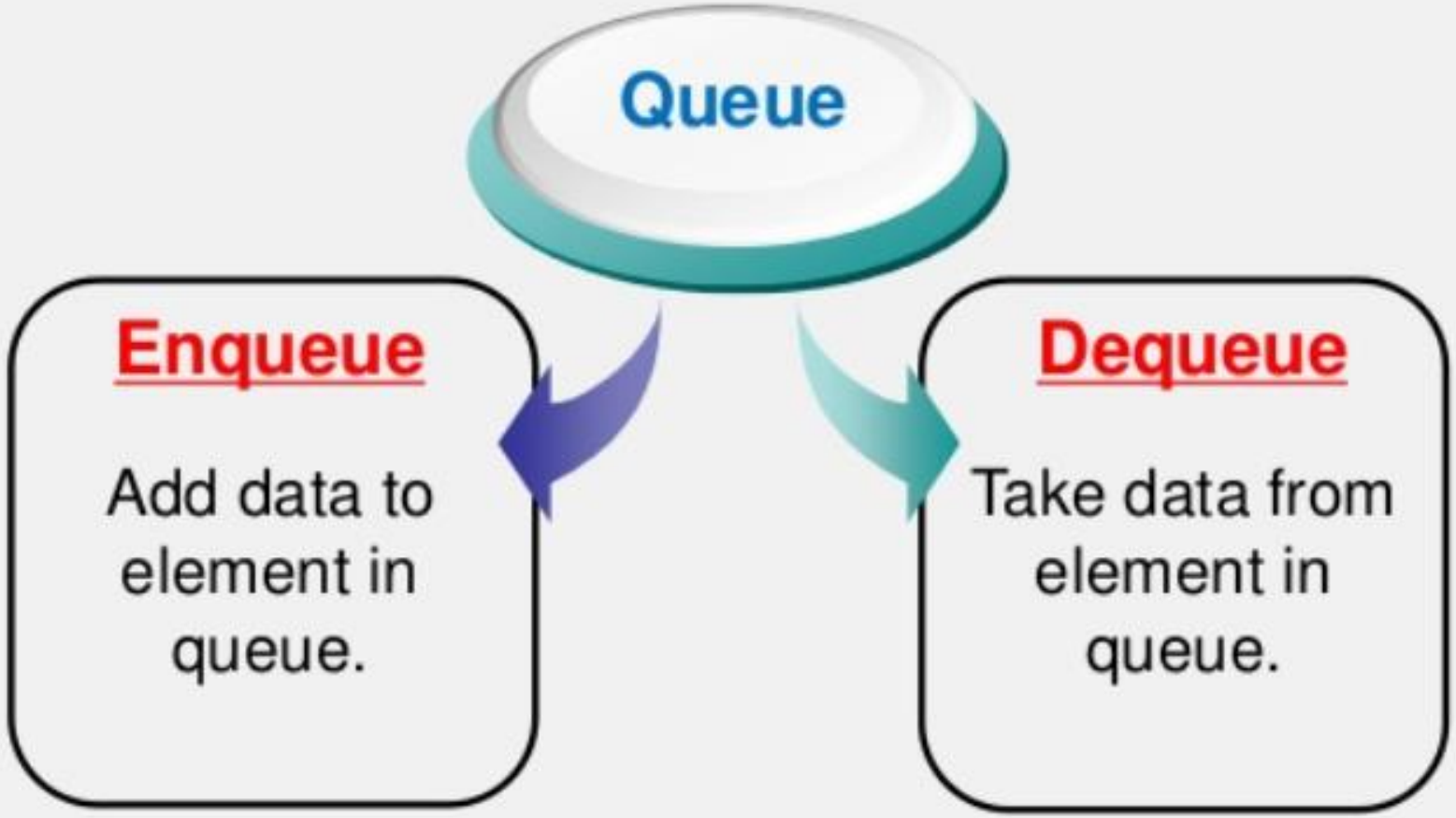
Const

Maxqueue=4

Type

- Queue_array=array[1.. Maxqueue] of integer
- Queue : Queue_array
- Front, Rear: integer

Main Operation



Enqueue

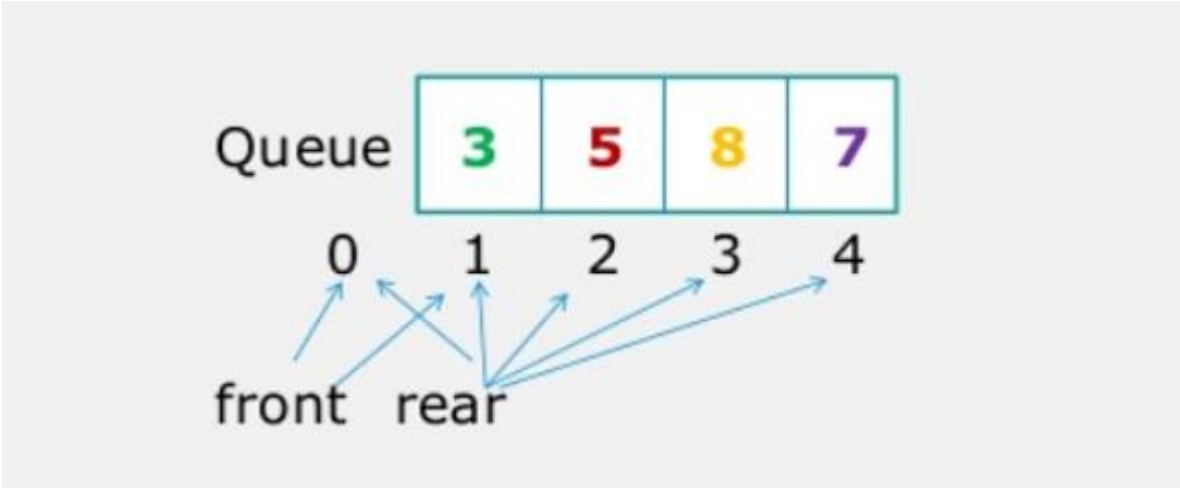
- Steps in queue operations:
- Queue can be added when it’s not full.
- If queue is empty, then front and rear is -1

Enqueue

Steps in enqueue operation:

- Queue can be added when it's not full
- If queue is empty then front and rear is added by 1. For the contrary, rear is added by 1.
- Queue element, which was refered by rear pointer, is filled with new data.

Enqueue



if queue is empty front=rear=-1

Enqueue(3) , front=0,rear=0

Enqueue(5) , front=0,rear=1

Enqueue(8) , front=0,rear=2

Enqueue(7) , front=0,rear=3

Enqueue(2) , front=0,rear=4, No enqueue, queue is full

Algorithm to insert an element in a queue

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to delete an element from a queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Queue implementation

- **Source code for Queue operations using array:**
- In order to create a queue, we require a one-dimensional array Q(1:n) and two variables *front* and *rear*.

Program

```
##include <stdio.h>
#include <conio.h>
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
```

Main program

```
int main()
{
        int option, val;
        do
        {
                printf("\n\n ***** MAIN MENU *****");
                printf("\n 1. Insert an element");
                printf("\n 2. Delete an element");
                printf("\n 3. Peek");
                printf("\n 4. Display the queue");
                printf("\n 5. EXIT");
                printf("\n Enter your option : ");
                scanf("%d", &option);
```

```
                switch(option)
                {
                case 1:
                     insert();
                     break;
                case 2:
                     val = delete_element();
                     if (val != -1)
                     printf("\n The number deleted is : %d", val);
                     break;
                case 3:
                     val = peek();
                     if (val != -1)
                     printf("\n The first value in queue is : %d", val);
                     break;
                case 4:
                     display();
                     break;
                }
        }while(option != 5);
        getch();
        return 0;
}
```

insert

```
void insert()
{
        int num;
        printf("\n Enter the number to be inserted in the queue : ");
        scanf("%d", &num);
        if(rear == MAX-1)
        printf("\n OVERFLOW");
        else if(front == -1 && rear == -1)
        front = rear = 0;
        else
        rear++;
        queue[rear] = num;
}
```

Delete

```
int delete_element()
{
        int val;
    if(front == -1 || front>rear)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    else
    {
        val = queue[front];
        front++;
        if(front > rear)
        front = rear = -1;
        return val;
    }
}
```

peek

```
int peek()
{
        if(front==-1 || front>rear)
        {
                printf("\n QUEUE IS EMPTY");
                return -1;
        }
        else
        {
                return queue[front];
        }
}
```

Display

```
void display()
{
        int i;
        printf("\n");
        if(front == -1 || front > rear)
        printf("\n QUEUE IS EMPTY");
        else
        {
                for(i = front;i <= rear;i++)
                printf("\t %d", queue[i]);
        }
}
```