:: UNIT – 3 ::

TOPIC ➔ Introduction to Data Structure

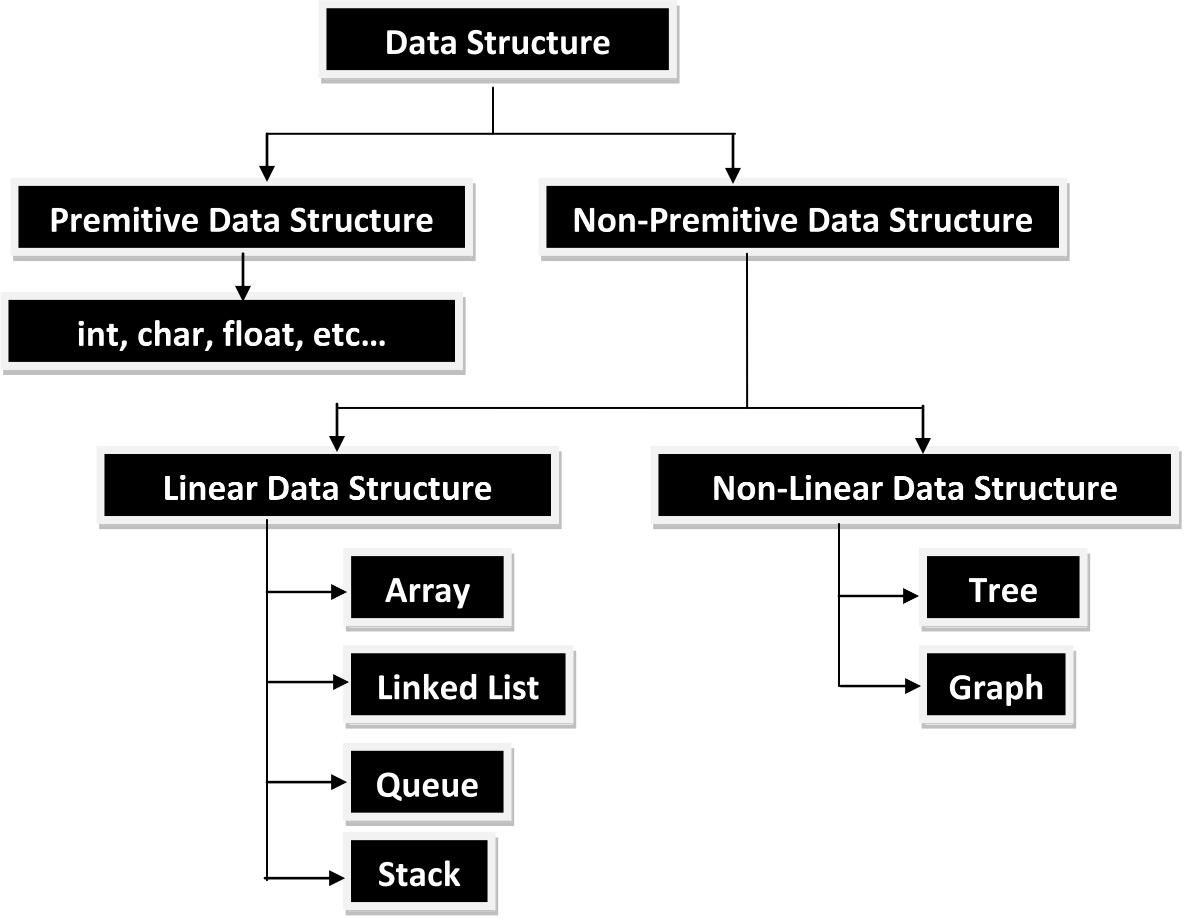
CONTENTS:

1) **Primitive and Simple Data Structures**
2) **Linear and Non Linear Structures File Organization**

❖ INTRODUCTION:

- Knowledge of data structures is required to people who design and develop computer programs of any kind: system software and application software.
- As we know, data are represented by data values held in memory or stored permanently on a file.
- Often the different data values are related to each other.
- To enable programs to make use of these relationships, these data values must be in an organized form.
- The organized collection of data is called a data structure.
- The programs have to follow certain rules to access and process the structured data.
- We may, therefore, say data are represented that:

Data Structure = Organized Data + Allowed Operation



Primitive Data Structures (Types):

- Primitive data types are data types provided by a programming language (like C) as building blocks.
- Primitive types are also known as *built-in* or *basic* types.
- The primitive data structures directly operate by machine level instructions.
- Examples: char, int, long, double, float etc.

Non - Primitive Or Composite Data Structures (Types):

- Non – primitive data types are those that can be constructed by using one or more primitive data structure.
- Non – primitive types are also known as composite types.
- The non – primitive data structures use primitive data types to access resources or data.
- Example: Array, Structure, Stack, Queue, Linked List etc.

- **Linear Data Structures:**
  - A data structure is said to be linear if its elements form a sequence or linear list. OR In linear data structure the data items are arranged in a linear sequence like in an array.
  - Two contiguous elements of linear data structure are stored in contiguous memory location and hence can be accessed easily.
  - Examples: Array, Linked List, Stack, Queues.

- **Non – Linear Data Structures:**
  - In a non-linear, the data items are not in sequence. An example of a non-linear data structure is a tree.
  - Data structures may also be classified as homogeneous and non-homogeneous data structures.
  - An array is a homogeneous structure in which all elements are of same type.
  - In non-homogeneous structures the elements may or may not be of the same type.
  - Records (Structures) are common example of non-homogeneous data structures.
  - Another way of classifying data structures is as static and dynamic data structures.
  - Static data structures are one whose sizes and structures associated memory location are fixed at compile time.
  - Dynamic structures are one which expand or shrink as required during the program execution and their associated memory locations change.

❖ Storage Structure for Array:

- There are mainly three types of arrays as follows:
  - One Dimensional Array:

    Syntax : Datatype array_name [size];

    Example : int arr[5]; (In this an integer array is declared with named arr and having 5 elements.)

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
100	102	103	106	108

Representation of an Array

  - Two Dimensional Array:

    Syntax : Datatype array_name [row_size][col_size];

    Example : int arr[5][5]; (In this an integer array is declared with named arr and having 5 rows and 5 columns.)

arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]
arr[4][0]	arr[4][1]	arr[4][2]	arr[4][3]	arr[4][4]

Representation of 2D Array

  - Multi Dimensional Array:

    Syntax : Datatype array_name [exp 1][exp 2]…..[exp N];

❖ Structure and Array of Structure:

- There are two ways through which array and structure are related with each other:
  - Array as member of structure:

    Example:

```
struct student
{
        int rno;
        char name[20];
        int marks[5];
}
```

  - Array of structure means structure variable as an array:

    Example: If we have structure of student and want to store result of 10 students, then array of structure variable can be defined as follow: struct student stud[10];

    Now, if we want to access rno and marks[5] for the stud[0] then, we have to write: stud[0].rno, stud[0].marks[0], stud[0].marks[1],stud[0].marks[2] and so on…

:: UNIT – 3 ::

TOPIC ➔ Elementary Data Structure

CONTENTS:

❖ INTRODUCTION OF STACK:

- Stack is a linear data structure and works same like as an array in which the latest data will be processed first.
- STACK is a special type of data structure where the insertion and deletion of an element is done from one end called Top Of Stack (TOS).
- Here, the last element inserted will be on the top of the stack. Since deletion is done from the same end, last element inserted is the first element to be deleted. So, stack is also called Last In First Out (LIFO) data structure.
- As we find that there must be facility to push the items on the stack, or to remove them (pop) from the top of the stack.
- Also we should have a way by which we will be able to know about the status of the stack as whether it is full or empty.
- We can check the size of the stack, and we can change the element.

❖ Implementation of Stacks:

- As we know that stack is linear list and to implement, it supports any list implementation techniques. There are mainly two types of techniques we implement for stack is as follows:
- *Static Stack*: with the use of an array means a stack with finite size.
- *Dynamic Stack*: With the use of pointer (linked list) means a stack with infinite size.
- The array implementation technique is very simple and easy to implement. But there is one potential risk that we need to declare that the size of an array before start the operation.
- It is usually easy to declare the array to be large enough without wasting too much memory spaces. Associated with each stack there is a top of stack, TOS that is -1 for an empty stack.

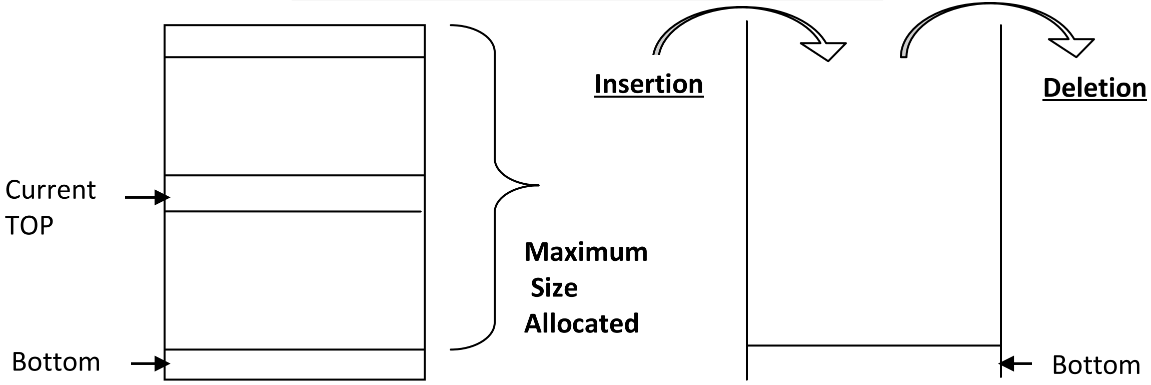
❖ Implementation of Stacks using Arrays:

- Stacks are one of the most important linear data structures of variable size. This is an important subclass of the lists (arrays) that permit the insertion and deletion from only one end TOP.

❖ Some Terminologies:

- *Insertion*: this operation is also called push
- *Deletion*: this operation is also called pop
- *Top*: A pointer, which keeps track of the top element in the Stack. If an array of size N is declared to be a stack, then *TOP will be -1 when the stack is empty* and *is N when the stack is full.*

Memory Representation of Stacks



❖ Function To Insert an element into the stack PUSH( ) :-

- Before inserting any element into the stack we must check whether the stack is full.
- In such case we cannot enter the element into the stack.
- If the stack is not full, we must increment the position of the TOP and insert the element. So, first we will write a function that checks whether the stack is full.

```
/* function to check whether stack is full */
    int stack_full(int top)
    {
            if(top >= SIZE – 1 )
                    return (1);
            else
                    return (0);
    }
```

- The function returns 1, if, the stack is full.
- Since we place TOP at –1 to denote stack empty condition, the top varies from 0 to SIZE -1 when it stores elements. Therefore TOP at SIZE -1 denotes that the stack is full.

```
/* function to insert (push) an element into the stack */
        void push ( )
        {
                int no;
                if ( top >= size -1)
                        printf("\n STACK IS FULL");
                 else
                {       printf (" Enter Number into the Stack::");
                        scanf("%d",&no);
                        top++
                        stk[top]=no;
                }
        }
```

❖ Function to delete an element from the stack

- Before deleting any element from the stack we must check whether the stack is empty.
- In such case we cannot delete the element from the stack.
- If the stack is not empty, we must delete the element by decrementing the position of the TOP. So, first we will write a function that checks whether the stack is empty.

/* function to check whether stack is empty */

```
 int stack_empty (int top)
{            if(top<0)
                        return (1);
             else
                        return (0);
}
```

- This function return 1 if the stack is empty.
- Since the elements are stored from positions 0 to SIZE-1, the empty condition is considered when the TOP has -1 in it.

/* function to delete an element from the stack */

```
void pop ( )
{
            int no;
            if(top<0)
                        printf("\n STACK IS EMPTY");
            else
            {
                        no=stk[top];
                        printf("Deleted item is %d", no);
                        top--;
            }
}
```

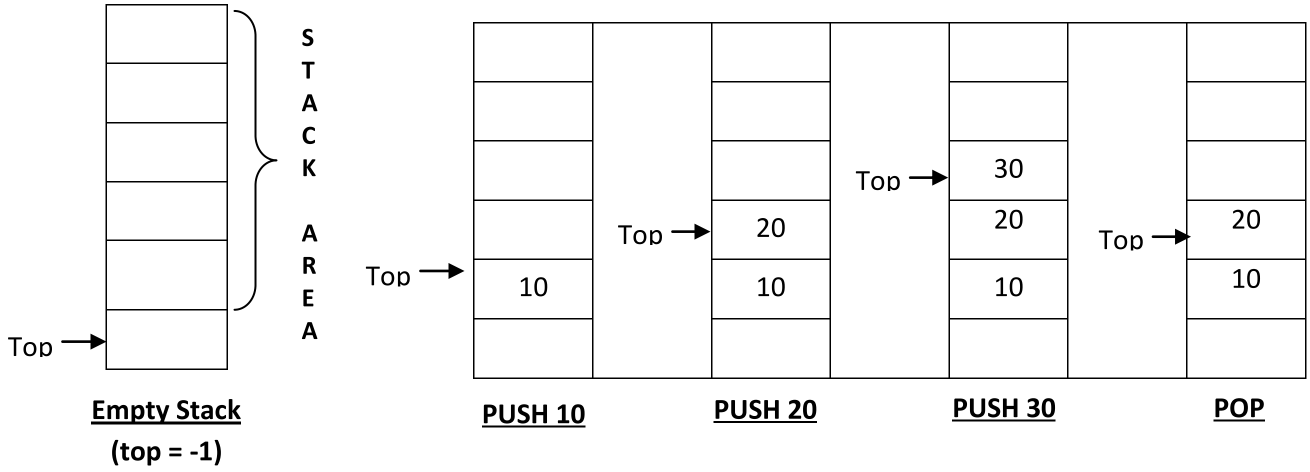
- Since the TOP points to the current top item, first we store this value in a variable and then decrements the TOP.

/* displays from top to bottom */

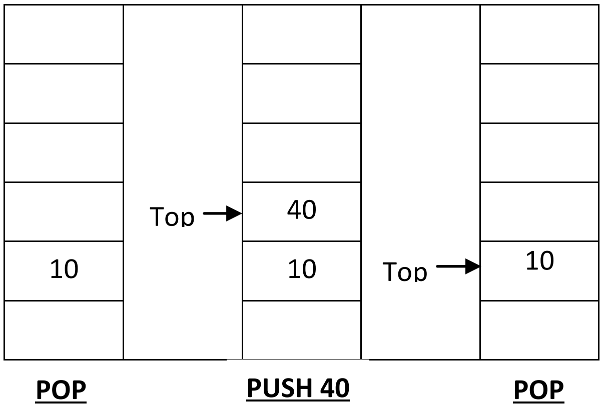
```
void display ( )
{
            for(i=top;i>=0;i--)
            {
                        printf("%f\n", stk[top]);
            }
}
```

```
/* displays from bottom to top */
    void display ( )
    {
            int i;
            for(i=0; i<=top;i++)
            {
                    printf("%f\n", stk[i]);
            }
    }
```

Now we will see working of the stack with diagrams.



Trace of stack values with push and pop functions (operations)



❖ Function to peep an element from the stack

- If we want to access some information stored at some location in a stack then peep operation is required.
- In this operation we just move the pointer to the desired location and then fetch the information related with that location.
- So, to fetch elements from the required location we need algorithm which describe below. For this operation, index value is subtracted from the TOS.

```
void peep( )
{
            int s;
            printf("Enter the location for searching::");
            scanf("%d",&s);
            printf("Value is %d",stk[top-s+1]);
}
```

❖ Function to change an element in the stack

- If anyone want to change the content of some location in a stack then update operation is required.
- Suppose one wants to update information at the ith location in the stack stk.
- We move TOS pointer to the ith location from the top of stack and input the new value of that location.

```
void change( )
{
            int i,c,new1;
            printf("\n Enter the position where you want to update/change ::: ");
            scanf("%d,&c);
            printf('\n\n The element of the location %d is ::: %d",c,stk[top-c+1]);
            printf("\n\n Enter the new record ::: ");
            scanf("%d",&new1);
            stk[top-c+1]=new1;
            printf("The new stack is ::: \n");
            for(i=top;i>=0;i--)
            {
              Printf("\t\t\t%d\n",stk[i]);
            }
}
```

❖ Write a C program that implement stack using Array

```
#include<stdio.h>
#include<conio.h>
#define size 100

int stk[size], top = -1;

void push();
void pop();
void display();
void peep();
void change();

void main()
{
        int ch;
        while(ch!=6)
        {
                clrscr();
                printf("\n Main Menu");
                printf("\n 1. Push");
                printf("\n 2. Pop");
                printf("\n 3. Display");
                printf("\n 4. Peep");
                printf("\n 5. Change");
                printf("\n 6. Exit");

                printf("\n Enter Your Choice: ");
                scanf("%d",&ch);

                switch(ch)
                {
                        case 1:
                                push();
                                display();
                                getch();
                                break;
                        case 2:
                                pop();
                                getch();
                                break;
                        case 3:
                                display();
                                getch();
                                break;
                        case 4:
                                peep();
                                getch();
                                break;
                        case 5:
                                change();
                                getch();
                                break;
                        case 6:
                                exit(0);
                                break;
                        default:
                                printf("End...");
                                break;
                }
        }
        getch();
}

void push()
{
        int no;
        if(top >= size-1)
                printf("\n Stack is FULL");
        else
        {
                printf("Enter Number : ");
                scanf("%d",&no);
                top++;
                stk[top] = no;
        }
}

void pop()
{
        int no;
        if(top < 0)
                printf("Stack is EMPTY");
```

```
    else
    {
        no = stk[top];
        printf("\nDeleted Item is %d",no);
        top--;
    }
}
void display()
{
    int i;
    if(top < 0)
        printf("Stack is EMPTY");
    else
    {
        printf("Content of the Stack : \n");
        for(i=top;i>=0;i--)
            printf("%d\n",stk[i]);
    }
}
void peep()
{
    int s;
    printf("Enter the Location to Search : ");
    scanf("%d",&s);
    printf("Value is %d",stk[top-s+1]);
}
void change()
{
    int i,c,new1;

    printf("\nEnter the position where you want to update / change :");
    scanf("%d",&c);

    printf("\n\nThe element of the location %d is %d",c,stk[top-c+1]);
    printf("\n\nEnter the new record :");
    scanf("%d",&new1);
    stk[top-c+1] = new1;
    printf("\nThe new stack is : \n");

    for(i=top;i>=0;i--)
        printf("\t\t\t%d\n",stk[i]);
}
```

❖ Implementing Stacks using Linked Lists:

- Advantages of Stacks using Arrays:
    - It is more efficient. And occupies lesser memory space, as pointers don't need have to be stored.
    - The implementation of data structure using array is easy to use.
- Disadvantages of Stacks using Linked Lists:
    - The size of array is fixed in advance, before they are implemented as stacks.
    - That means the size of the stack can't be increased or decreased.
- To overcome this situation stack with linked list is used.
- While using linked list, we shall push and pop nodes from one end of a linked list.
- The stack as linked list is represented as a singly connected list.
- Each node in the linked list contains the data and a pointer that gives location of the next node in the list.
- The structure for the LIFO data structure stack is as follows:

```
struct stack
{
        int info;
        struct stack *next;
} *start, *node;
```

- The linked list will have header node, which will always point to the top of the stack.
- Another important thing for this implementation is that the new element will be always added at the beginning of the linked list and while deleting the element it will be deleted from the same end i.e. the beginning of the linked list.

❖ Function to check whether the stack is empty

- To check whether the stack is empty, we will pass the header node which always point to top of the stack. Thus, when the header points to NULL, the stack will be empty.

```
int stack_empty(struct stack *st1)
{
        if(st1 == NULL)
                return (1);
        else
                return (0);
}
```

❖ Function to check whether the stack is full

- There is no condition based on any existing pointer, but if the new_node function returns NULL, i.e. no more memory is available for locations.

```
int (new1 == NULL)
{
        return(new1);
}
```

- The funtion should be used very carefully, we should not generate a new node when the stack is not full.
- The stack_full function must have returned the address of the new allotted node.

❖ Function to push an element into the stack

```
struct stack *push(struct stack *st1)
{
        struct stack *new1;
        new1 = (struct stack *) malloc (sizeof(struct stack));
        if(new1 == NULL)
        {
                return (new1);
        }
        new1->next = st1;
        start = new1;
        st1 = new1;

        printf("Enter the value : ");
        scanf("%d", &st1->info);
        return(st1);
}
```

❖ Function to pop an element from the stack

```
void pop(struct stack *st1)
{
        if(st1 == NULL)
        {
                printf("\nStack is EMPTY");
                return;
        }
        printf("\nThe %d element is popped", start -> info);
        start = start -> next;
        free(st1);
}
```

❖ Write a C program that implement stack using Linked List OR Stack using Pointer / Dynamic Stack

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

struct node
{
   int data;
   struct node *next;
};
struct node *top=NULL,*temp;
void push();
void pop();
void display();

void main()
{
   int ch;
   while(1)
   {
         clrscr();
         printf("\n\t1.Push");
         printf("\n\t2.Pop");
         printf("\n\t3.Display");
         printf("\n\t4.Exit\n");
         printf("\n\tEnter ur choice:");
         scanf("%d",&ch);

         switch(ch)
         {
               case 1:
                     push();
                     getch();
                     break;
               case 2:
                     pop();
                     getch();
                     break;
               case 3:
                     display();
                     getch();
```

```
            break;
        case 4:
            exit(0);
    }
  }

}

void push()
{
    temp=(struct node *)malloc(sizeof(struct node));
    printf("\n\tEnter a number :");
    scanf("%d",&temp->data);
    temp->next=top;
    top=temp;
}
void pop()
{
    if(top!=NULL)
    {
        printf("\n\tThe popped element is %d\t",top->data);
        top=top->next;
    }
    else
    {
        printf("\n\tStack is empty\n");
    }
}
void display()
{
    temp=top;
    if(temp==NULL)
    {
        printf("\n\tStack is empty\n");
    }

    while(temp!=NULL)
    {
        printf("\n\t%d",temp->data);
        temp=temp->next;
    }
}
```

❖ Recursion and Stacks

- Recursion is powerful tool in C. It is a process by which a function calls itself frequently, until a function specified condition has been fulfilled.
- When a called function in turn calls another function a process of chaining occurs.
- Recursion is a special case of this process, where a function calls itself. The process is used for repetitive computation in which each action in terms of a previous result.
- Many iterative problems can be within in this form.
- The recursion is based on LIFO mechanism.
- We can also define recursion as a process in which a function calls itself with reduced input and has a base condition to stop the process. i.e. any recursive function must satisfy two conditions:
  - It must have a terminal condition.
  - After each recursive call it should reach a value nearing the terminal condition.
- Following is the example to find the factorial of a number.

$$\text{Fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{fact}(n-1) & \text{otherwise} \end{cases}$$

We can compute 4! As shown below

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1

By definition 0! is 1. So, 0! will not be expressed in terms of itself. Now, the computations will be carried out in reverse order as shown.

0! = 1 (i.e. 1)
1! = 1 * 0! (i.e. 1)
2! = 2 * 1! (i.e. 2)
3! = 3 * 2! (i.e. 6)
4! = 4 * 3! (i.e. 24)

This characteristics of recursion suggests that we can use stack to push (store) returned value by each step of recursion and then can pop (retrieve) them back at the end. When a function is called, the return address, the values of local and formal variables are pushed onto the stack, a block of memory of contiguous locations, set aside for this purpose. After this, the control enters into the function. Once the return statement is encountered, control comes back to the previous call, by using return value present in the stack. And it substitutes the return value to the call. If the function does not return any value, control goes to the statement that follows the function call.

- **Iteration V/s. Recursion**

  In recursion, every time a function is called, all the local variables, formal variables and return address will be pushed on the stack. So, it occupies more stacks and most of the time is spent in pushing and popping. Whereas, the non-recursive function execute much faster and are easy to design.

Application of Stack:

- **Polish Notation:**
  - **Prefix, Infix and Postfix Expression:**

    Infix notation is the common arithmetic and logical formula notation, in which operators are written infix-style between the operands they act on. (E.g. A + B).

    In Postfix notation, the operator comes after the operand. For Example, the infix expression A + B will be written as AB+ in its postfix notation.

    In Prefix notation, the operator comes before the operand. The infix expression A+B will be written as +AB in its prefix notation.

    Prefix notation is also known as 'Polish Notation' and Postfix notation is also known as 'Reverse Polish Notation'.

    In computers it is much easier to parse the infix and postfix notation than parsing the infix notation. So computers use mathematical expressions in either the Polish (Prefix) or Reverse Polish Notation (Postfix) notations, but many programming languages use it due to its familiarity.

  - **Conversion from Infix to Postfix:**

    Stack is commonly used to convert an expression from infix to postfix notation. The conversion algorithm uses stack. While converting the expression BODMAS rule is used.
    BODMAS (Brackets, Orders (powers and square root), Division, Multiplication, Addition, Subtraction).

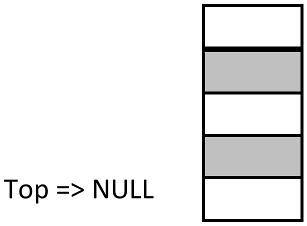
    Algorithm:

    - Scan the infix expression from left to right for tokens (Operators, Operands & Parentheses).
    - If token is operand, append it in postfix expression.
    - If token is a left parentheses " (", push it in stack.
    - If token is an operator, pop all the operators which are of higher or equal precedence than the incoming token and append them to the output expression. After popping out all such operators, push the new token on stack.
    - If right parentheses ") " is found, pop all the operators from the stack and append them to output string.

- When all tokens of infix expression have been scanned. Pop all the elements from the stack and append them to the output string.

Example: Infix Notation: A * (B + C) – D / E

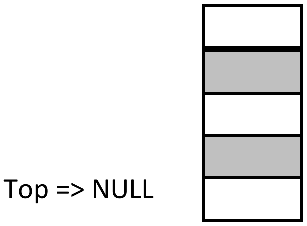
- Initially, stack is empty and we have only infix notation.



Infix Notation:
A * (B + C) – D / E

Postfix Notation:

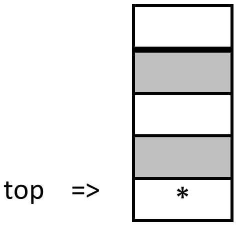
- The first token is A and as it is an operand, it is appended to the output.



Infix Notation:
*** (B + C) – D / E**

Postfix Notation:
A

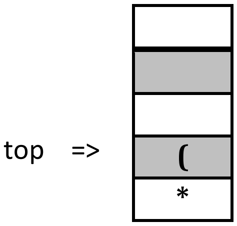
- Next token is *. Since stack is empty, it is pushed into the stack.



Infix Notation:
*** (B + C) – D / E**

Postfix Notation:
A

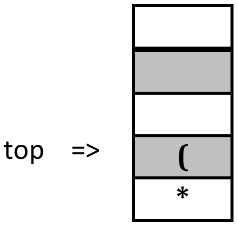
- Next token is (, the precedence of open-parentheses, as it is maximum, it will be pushed to the stack. But when another operator is to come on the top of '(' then its precedence is least.



Infix Notation:
B + C) – D / E

Postfix Notation:
A

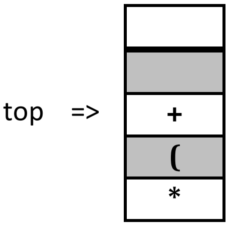
- Next token is B and as it is an operand, appended to output string.



Infix Notation:
+ C) – D / E

Postfix Notation:
AB

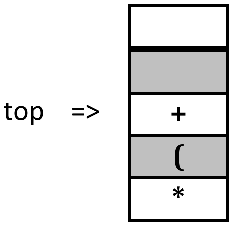
- Now, the next token is + and as it is operator, it will be pushed to the stack.



Infix Notation:
C) – D / E

Postfix Notation:
AB

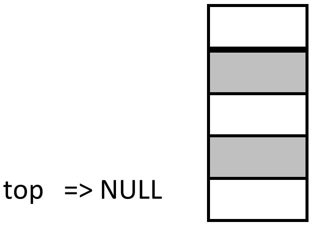
- The next token is C and as it is an operand, appended to output string.



Infix Notation:
– D / E

Postfix Notation:
ABC

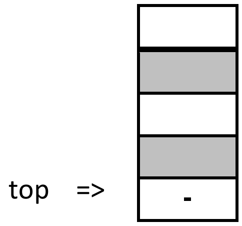
- The next token is ‘) ‘ - is an operator and is with higher or equal precedence than other operators in the stack. So, pop all the elements from stack and append them to the output string.



Infix Notation:
– D / E

Postfix Notation:
ABC + *

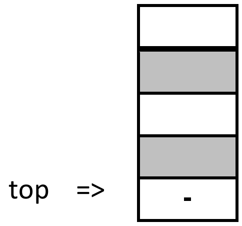
- Now, the next token is ‘ – ‘ which is an operator and pushed to the stack.



Infix Notation:
D / E

Postfix Notation:
ABC + *

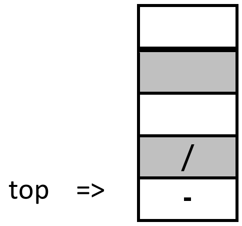
- Next token is D and as it is operand, it is appended to output string.



Infix Notation:
/ E

Postfix Notation:
ABC + * D

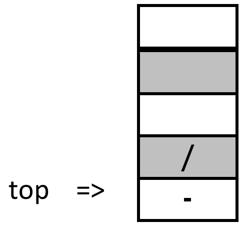
- Next, we have / as token and which is an operator and pushed to the stack.



Infix Notation:
E

Postfix Notation:
ABC + * D

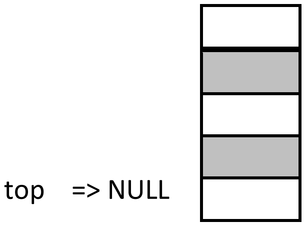
- After this, the next token is E and as it is operand, it is appended to output string.



Infix Notation:

Postfix Notation:
ABC + * DE

- Now, the input expression is complete. So, pop all the elements from the stack and append them to the output string.



Infix Notation:

Postfix Notation:
ABC + * D / -

- So finally we get output as ABC + * DE/-

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	Multiply A and B, divide C by D, add the results.
A * (B + C) / D	A B C + * D /	/ * A + B C D	Add B and C, multiply by A, divide by D.
A * (B + C / D)	A B C D / + *	* A + B / C D	Divide C by D, add B, multiply by A

Converting between these notations

The most straightforward method is to start by inserting all the implicit brackets that show the order of evaluation. E.g.

Infix	Postfix	Prefix
((A * B) + (C / D))	((A B *) (C D /) +)	(+(* A B) (/ C D))
((A * (B + C)) / D)	((A (B C +) *) D /)	(/(*A (+ B C))D)
(A * (B + (C / D)))	(A (B (C D /) +) *)	(* A (+ B (/ C D)))

You can convert directly between these bracketed forms simply by moving the operator within the brackets. That is, (x + y) or (x y +) or (+ x y). Repeat this for all the operators in an expression, and finally remove any superfluous brackets.

: STATIC Stack ALGORITHMS:

- *PUSH Operation:*

```
Step – 1: If (top >= size-1)        (Checking whether the stack is full or not)
          Output: "Stack is full" and exit.
Step–2: top = top + 1
Step –3: stk[top] = no;
Step– 4: exit
```

- *POP Operation:*

```
Step – 1: If (top == -1)             (Checking whether the stack is empty or not)
                    Output: “Stack is empty” and exit.
Step – 2: no = stk[top];
Step –3:top = top – 1;
Step – 4: exit
```

- *PEEP Operation:*

```
Step – 1:  If (top == -1)             (Checking whether the stack is empty or not)
                    Output: “Stack is empty” and exit.
Step – 2:  Print: “Enter position to search”
           read search_no;
Step – 3: no = stk[top – search_no + 1];
Step – 4: exit
```

- *UPDATE Operation:*

```
Step – 1:  If (top == -1)             (Checking whether the stack is empty or not)
                    Output: “Stack is empty” and exit.
Step – 2:  Print: “Enter Position to update”
           read search_no;
Step – 3:  stk[top – search_no + 1] = search_no;
Step – 4: exit
```

: DYNAMIC Stack ALGORITHMS:

- *PUSH Operation:*

```
Step – 1: ptr = Create a new node (Allocate free space)
Step–2: if (ptr == NULL)
                    Output: “Insufficient Memory” and exit.
Step –3: ptr -> data = value;
Step– 4: ptr -> next = top;
Step – 5: top = temp;
Step – 6: exit
```

- *POP Operation:*

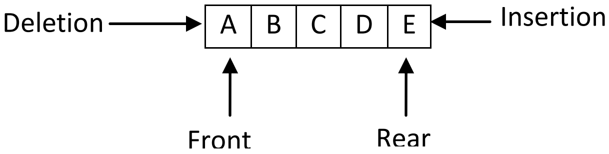
```
Step – 1: if (top == NULL)
                    Output: “Stack is empty” and exit.
Step – 2: Print: “The node deleted is “, top->data
Step –3: top = top->next            (Moving top to the next node).
```

  - Step – 4: exit

:: UNIT – 3 ::

❖ INTRODUCTION OF QUEUE:

- It is same as an array and stack.
- Queue is a linear data structure as it stores data in sequential manner.
- Queues are of two types: Simple Queue and Circular Queue.
- The insert operation of queue (also known as ENQUEUE) is performed at one end known as rear.
- The delete operation of queue (also known as DEQUEUE) is performed at other end known as front.
- In short, front and rear are the two variables used to keep track of queue.
- The example of queue is a queue of persons at airline reservation window.



- There are two common ways in which queues may be implemented :
  - Static Implementation (with the use of arrays)
  - Dynamic Implementation (with the use of Pointers (Or Linked Lists).

: STATIC Queue ALGORITHMS:

- *Insert Operation:*

```
Step – 1: If (rear >= size-1)     (Checking whether the queue is full or not)
              Output: “Queue is full” and exit.
Step–2: rear = rear + 1
Step –3: q[rear] = no;
Step– 4: if front = 0             (Set the front variable …)
              front = 1;
```

- *Delete Operation:*

```
Step – 1: If (rear >= size-1)     (Checking whether the queue is full or not)
              Output: “Queue is full” and exit.
Step–2: rear = rear + 1
Step –3: q[rear] = no;
Step– 4: if front = 0             (Set the front variable …)
              front = 1;
```

: DYNAMIC Queue ALGORITHMS:

- *Insert Operation:*

```
Step – 1: temp = Create a new node (Allocate free space)
Step –2:  temp->next = NULL;
Step –3: temp->data = value;
Step– 4: if(*front == NULL && *rear == NULL)
                (Check if the created node is 1st node or not)
                *front = *rear =temp;
Step – 5: else
Step – 6:
        *rear ->next = temp;
        *rear = temp;
```

- *Delete Operation:*

```
Step – 1: if (*front == NULL && *rear == NULL)
                (Check whether the queue is empty or not.)
                Output: “Queue is empty” and exit.
Step –2: temp = *front;
Step – 2: Print: “The node deleted is “, temp->data
Step –3: *front = *front->next    (Moving *front to the next node).
Step – 4: exit.
```

❖ CIRCULAR QUEUE:

- In simple queue, all insertion of an element is performed at rear (last position) and all the deletions are possible at front (first position).
- The limitation of simple queue is that once the queue is FULL and when we are removing elements using front, these array elements become blank.
- So, when queue is FULL, the value of rear becomes equals to the size of the array. When we are removing elements using front, the value of front is incremented. And Though we have blank element in array, we can’t insert any new element value as rear == size.
- Now to overcome this problem, the new technique *‘Circular Queue’* is used.
- In this technique, when value of rear reaches the queue’s size the first element will become the queue’s new value for rear.
- That means, once the queue is full, and we try to remove an element, then front will move forward and rear will move to first position of queue.

: Algorithm for Circular Queue:

- *Insert Operation:*

```
Step – 1: if(front==0 && rear == size-1 || rear+1 ==front)
                (Check whether Queue is Full or not)
                Output: “Queue is full” and exit.
Step –2:  else if(front == -1 && rear == -1)
                front = rear = 0;
```

```
Step – 3: else if(rear == size-1)
                rear = 0;
Step – 4: else
                rear = rear + 1;
Step – 5: q[rear] = no;
```

- *Delete Operation:*

```
Step – 1: if(front==-1 && rear == -1)      (Check whether Queue is Empty or not)
                Output: “Queue is empty” and exit.
Step – 2: Print: “The element deleted is “, q[front].
Step – 3: if(front == size-1  && rear < front)
                front = 0;
Step – 4: else if(front  == rear)
                front = rear = -1;
Step – 5: else
                front = front + 1;
```

❖ **PROGRAMS:**

- **STATIC QUEUE:**

```
//USE OF QUEUE (STATIC QUEUE)....
#include<stdio.h>
#include<conio.h>
#define size 100
int q[size], front = -1,rear = -1;

void insert();
void delete1();
void display();

void main()
{
        int ch;
        while(ch!=6)
        {
                clrscr();
                printf("\n Main Menu");
                printf("\n 1. Insert");
                printf("\n 2. Delete");
                printf("\n 3. Display");
                printf("\n 4. Exit");
        printf("\n Enter Your Choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert();
                display();
                getch();
                break;
            case 2:
                delete1();
                getch();
                break;
            case 3:
                display();
                getch();
                break;
            case 4:
                exit(0);
            default:
                printf("End...");
                break;
        }
    }
    getch();
}

void insert()
{
    int num;
    if(rear >= size -1)
        printf("\nQueue is FULL...");
    else
    {
        printf("\nEnter Number : ");
        scanf("%d",&num);
        rear++;
        q[rear] = num;
        if(rear == 0)
            front =0;
    }
}
```

```
void delete1()
{
    if(front < 0)
        printf("\nQueue is EMPTY...");
    else
    {
        printf("\nThe element %d is deleted",q[front]);
        if(front == rear)
            front = rear = -1;
        else
            front ++;
    }
}
void display()
{
    int i;
    if(front < 0)
        printf("\nQueue is EMPTY...");
    else
    {
        printf("\nThe Elements of Queue are ...\n");
        for(i=front;i<=rear;i++)
            printf("\n%d",q[i]);
    }
}
```

- **DYNAMIC QUEUE:**

```
//DYNAMIC QUEUE OR QUEUE WITH LINKED LIST OR QUEUE USING POINTER...
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct queue
{
        int data;
        struct queue *next;
};
typedef struct queue q;
void insert(q**,q**);
void delete1(q**,q**);
void display(q*);
void main()
{
   int ch;
   q *front, *rear;
   rear=front=NULL;
   while(ch!=4)
   {
          clrscr();
          printf("\n\t1. Insert");
          printf("\n\t2. Delete");
          printf("\n\t3. Display");
          printf("\n\t4. Exit");
          printf("\n\tEnter Your Choice : ");
          scanf("%d",&ch);
          switch(ch)
          {
                 case 1:
                        insert(&front,&rear);
                        getch();
                        break;
                 case 2:
                        delete1(&front,&rear);
                        getch();
                        break;
                 case 3:
                        display(front);
                        getch();
                        break;
                 case 4:
                        exit(0);
          }
   }
   getch();
}

void insert(q **front,q **rear)
{
          q *temp;
          temp=(struct queue*)malloc(sizeof(struct queue));
          temp->next=NULL;
          printf("\n\tEnter Number : ");
          scanf("%d",&(temp->data));
          if((*front)==NULL || (*rear)==NULL)
          {
                *front=temp;
                *rear=temp;
        }
        else
        {
                (*rear)->next=temp;
                *rear=temp;
        }
}

void delete1(q **front,q **rear)
{
        q *temp;
        if((*front)==NULL || (*rear)==NULL)
                printf("\n\tQueue is empty.");
        else
        {
                temp=*front;
                printf("\n\tThe element %d is deleted",temp->data);
                (*front)=(*front)->next;
                free(temp);
        }
}
void display(q *temp)
{
        if(temp==NULL)
                printf("\n\tQueue is empty.");
        else
        {
                while(temp!=NULL)
                {
                        printf("\n\t%d",temp->data);
                        temp=temp->next;
                }
        }
}
```

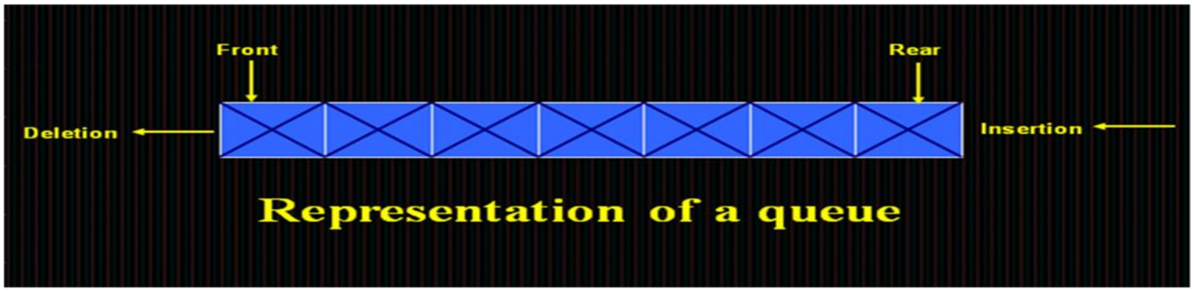
- **CIRCULAR QUEUE:**

```
//CIRCULAR QUEUE.....
#include<stdio.h>
#include<conio.h>
#define size 5

void insert();
void delete1();
void display();
void search();
void change();

int q[size];
int front = -1;
int rear = -1;

void main()
{
        int ch;
        clrscr();
        while(ch!=6)
        {
                clrscr();
                printf("\n\t1. Insert ");
                printf("\n\t2. Delete ");
                printf("\n\t3. Display ");
                printf("\n\t4. Search ");
                printf("\n\t5. Change ");
                printf("\n\t6. Exit");
                printf("\n\tEnter Your Choice :: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1:
                                insert();
                                display();
                                getch();
                                break;
                        case 2:
                                delete1();
                                getch();
                                break;
```

```
            case 3:
                display();
                getch();
                break;
            case 4:
                search();
                getch();
                break;
            case 5:
                change();
                getch();
                break;
            case 6:
                exit(0);
            default:
                break;
        }
    }
}

void insert()
{
    if(front==0 && rear == size-1 || rear+1 ==front)
    {
        printf("\tQueue is Full");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else if(rear == size-1)
    {           rear = 0;       }
    else
    {
        rear = rear + 1;
    }
    printf("\tEnter Element : ");
    scanf("%d",&q[rear]);
}
```

```
void delete1()
{
        if(front == -1 && rear == -1)
        {
                printf("\tQueue is Empty");
                return;
        }
        printf("The element to be deleted is : %d",q[front]);
        if(front==size-1 && rear < front)
        {               front = 0;      }
        else if(front == rear)
        {               front = rear = -1;      }
        else
        {               front = front + 1;      }
}

void display()
{
        int i;
        if(front == -1 && rear == -1)
        {
                printf("\tQueue is empty");
                return;
        }
        if(rear >= front)
        {
                for(i=front;i<=rear;i++)
                        printf("\n\t%d",q[i]);
        }
        else
        {
                for(i=front;i<=size-1;i++)
                {
                        printf("\n\t%d",q[i]);
                }
                for(i=0;i<=rear;i++)
                {
                        printf("\n\t%d",q[i]);
                }
        }
}
```

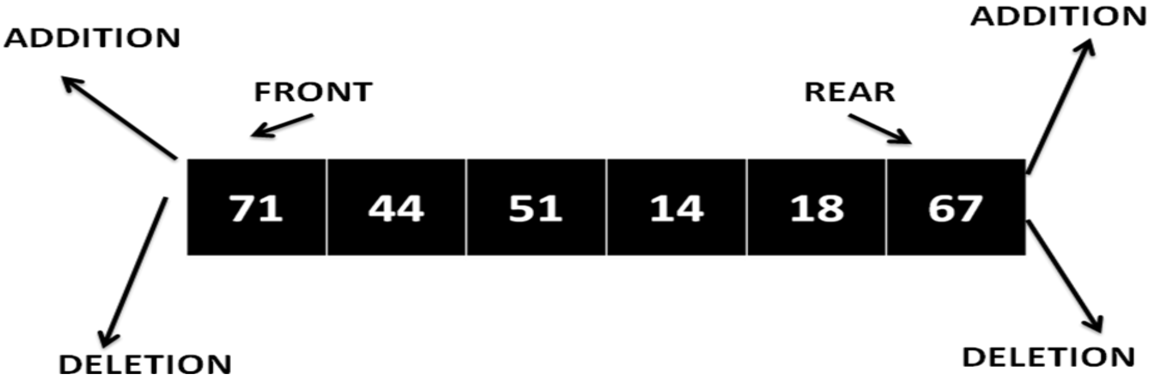
```
void search()
{
        int position;
        if(front == -1)
                printf("\n\tQueue is Empty");
        else
        {
                printf("\n\tEnter the position that you want to search : ");
                scanf("%d",&position);
                printf("The element at %d is %d",position,q[position-1]);
        }
}

void change()
{
        int position,new1;
        if(front == -1)
                printf("\n\tQueue is Empty");
        else
        {
                printf("\n\tEnter the position that you want to update / change : ");
                scanf("%d",&position);
                printf("\n\tThe value at %d is %d",position,q[position-1]);
                printf("\n\tEnter New Value : ");
                scanf("%d",&new1);
                q[position-1]=new1;
                display();
        }
}
```



Representation of a queue

DEQUES

- A Double Ended Queue is works same like as simple queue but insertion and deletions are possible at either end.
- That means, the insertion and deletions are possible at both the ends.
- We can add / remove a new element @ rear / front end.
- Therefore, it is known as Double Ended Queues.



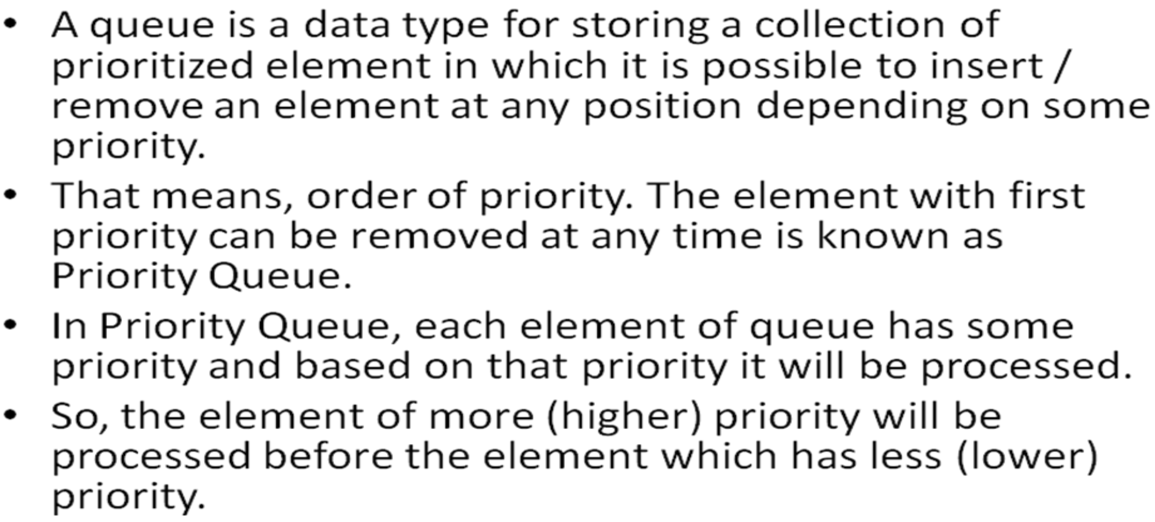
- There are two types of DEQUES:
  - INPUT RESTRICTED DEQUE:
    - An input restricted deque allows insertion at only one end (rear end) whereas deletion at both ends.
  - OUTPUT RESTRICTED DEQUE:
    - An output restricted deque allows deletion at only one end (front end) whereas insertion at both ends.

Possible Operations in DEQUE are:

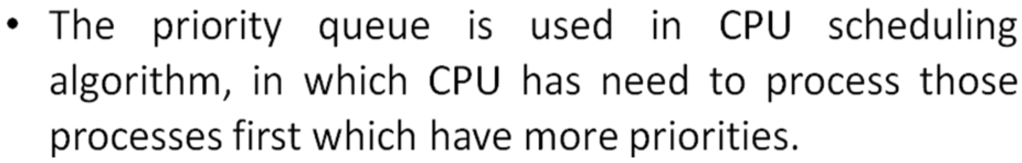
1. Enqueue Front : Add an element @ the front end.
2. Enqueue Rear : Add an element @ the rear end.
3. Dequeue Front : Remove an element @ the front end.
4. Dequeue Rear : Remove an element @ the rear end.

- Among the above operations 2nd, 3rd, 4th are performed by input restricted deque and 1st, 2nd, and 3rd are performed by output restricted deque.

PRIORITY QUEUES



- A queue is a data type for storing a collection of prioritized element in which it is possible to insert / remove an element at any position depending on some priority.
- That means, order of priority. The element with first priority can be removed at any time is known as Priority Queue.
- In Priority Queue, each element of queue has some priority and based on that priority it will be processed.
- So, the element of more (higher) priority will be processed before the element which has less (lower) priority.



- The priority queue is used in CPU scheduling algorithm, in which CPU has need to process those processes first which have more priorities.