Data Structure

Introduction

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artifical intelligence, Graphics and many more.

Basic Terminology

- Data structures are the building blocks of any program or the software.
- Choosing the appropriate data structure for a program is the most difficult task for a programmer.

Following terminology is used as far as data structures are concerned

Data: Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

Group Items: Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

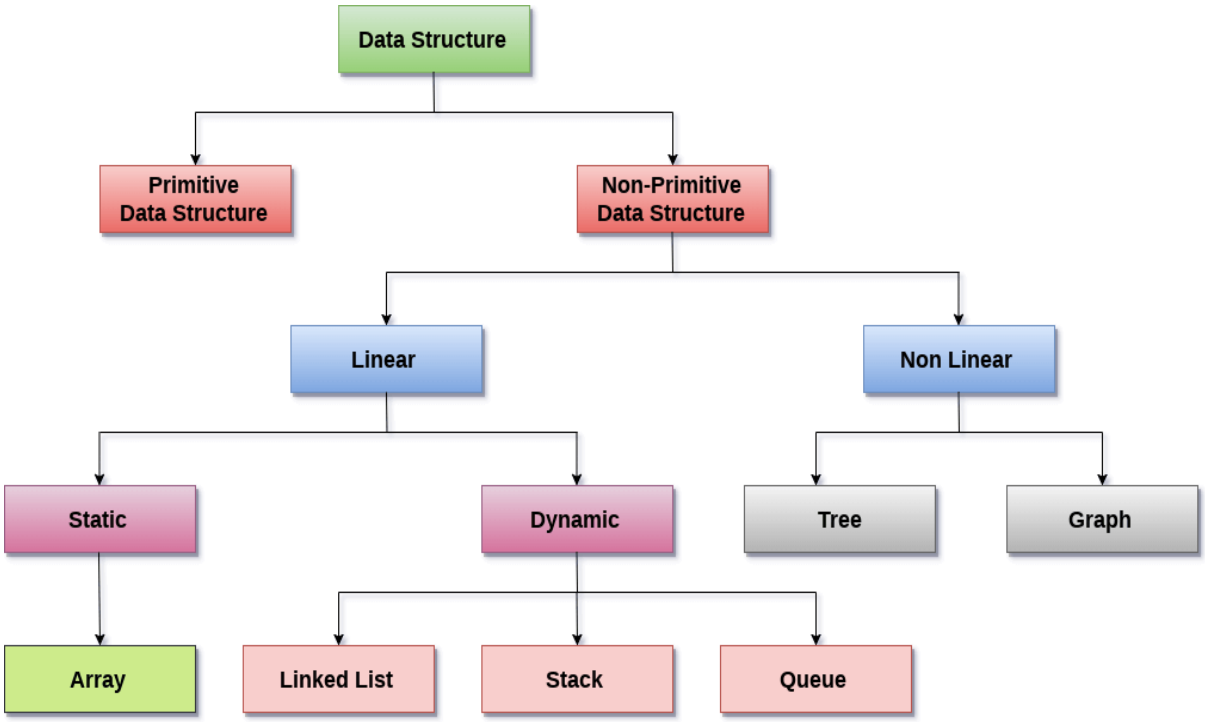
Record: Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

File: A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

Attribute and Entity: An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

Field: Field is a single elementary unit of information representing the attribute of an entity.

Data Structure Classification



Linear Data Structures:

- A data structure is called linear if all of its elements are arranged in the linear order.
- In linear data structures, the elements are stored in non-hierarchical order

Types of Linear Data Structures

Arrays:

- An array is a collection of similar type of data items and each data item is called an element of the array.
- The data type of the element may be any valid data type like char, int, float or double.
- The elements of array share the same variable name but each one carries a different index number known as subscript.
- The array can be of one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],..... age[98], age[99].

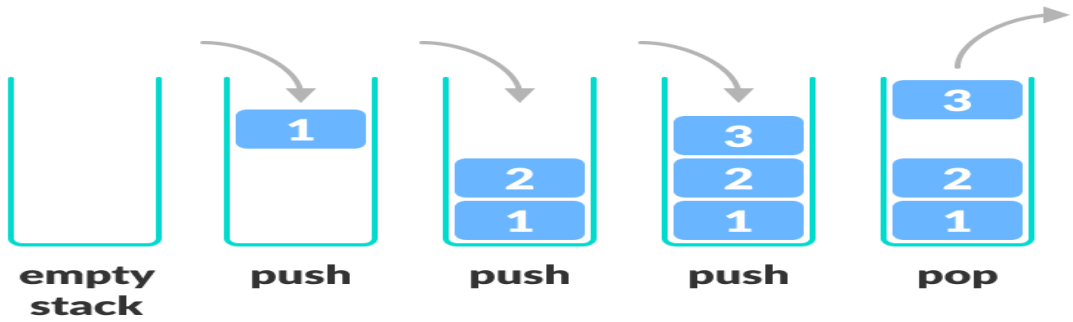
Linked List:

- Linked list is a linear data structure which is used to maintain a list in the memory.
- It can be seen as the collection of nodes stored at non-contiguous memory locations.
- Each node of the list contains a pointer to its adjacent node.

Stack:

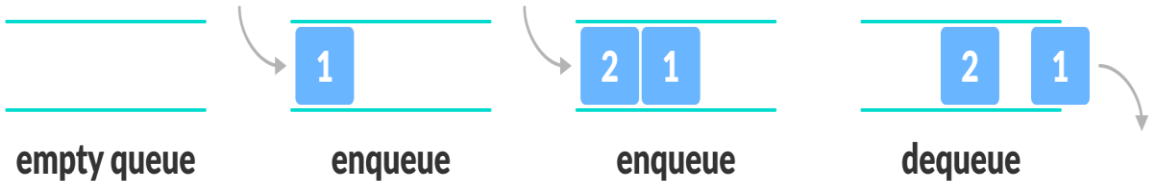
- Stack is a linear list in which insertion and deletions are allowed only at one end, called top.
- Stack works on the principle of LIFO(last in first out)/FILO(First in last out)
- A stack is an abstract data type (ADT), can be implemented in most of the programming languages.
- It is named as stack because it behaves like a real-world stack,

for example: - pile of plates or deck of cards etc.



Queue:

- Queue is a linear list in which elements can be inserted only at one end called rear and deleted only at the other end called front.
- It is an abstract data structure, similar to stack.
- Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.



Non Linear Data Structures:

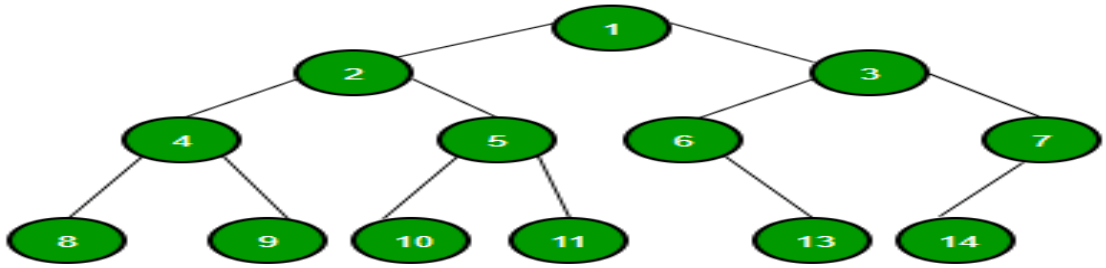
- This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement.
- The data elements are not arranged in sequential structure.

Types of Non Linear Data Structures are given below:

Trees:

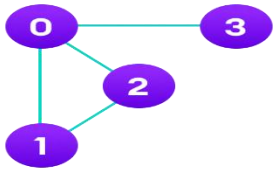
- Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes.
- The bottommost nodes in the herierchy are called leaf node while the topmost node is called root node.
- Each node contains pointers to point adjacent nodes.
- Tree data structure is based on the parent-child relationship among the nodes.

- Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node.
- Trees can be classfied into many categories which will be discussed later



Graphs:

- Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges.
- A graph is different from tree in the sense that a graph can have cycle while the tree can not have the one.



Operations on data structure

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will devide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

Stack

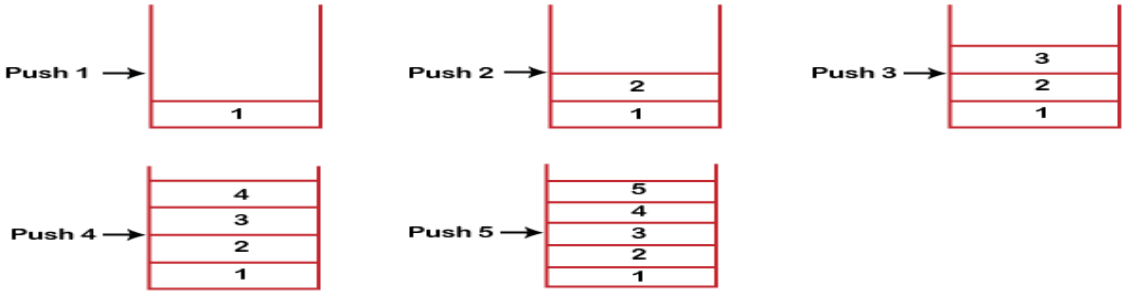
- A Stack is a linear data structure that follows the **LIFO (Last-In-First-Out)** principle.
- Stack has one end, whereas the Queue has two ends (**front and rear**).
- It contains only one pointer **top pointer** pointing to the topmost element of the stack.
- Whenever an element is added in the stack, it is added on the top of the stack, and
- the element can be deleted only from top of the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

Some key points related to stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

Working of Stack

- Stack works on the LIFO pattern.
- As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.
- Suppose we want to store the elements in a stack and let's assume that stack is empty.
- We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



- Since our stack is full as the size of the stack is 5.
- In the above cases, we can observe that it goes from the top to the bottom when we were entering the new element in the stack. The stack gets filled up from the bottom to the top.
- When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed.
- It follows the LIFO pattern, which means that the value entered first will be removed last.
- In the above case, the value 1 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

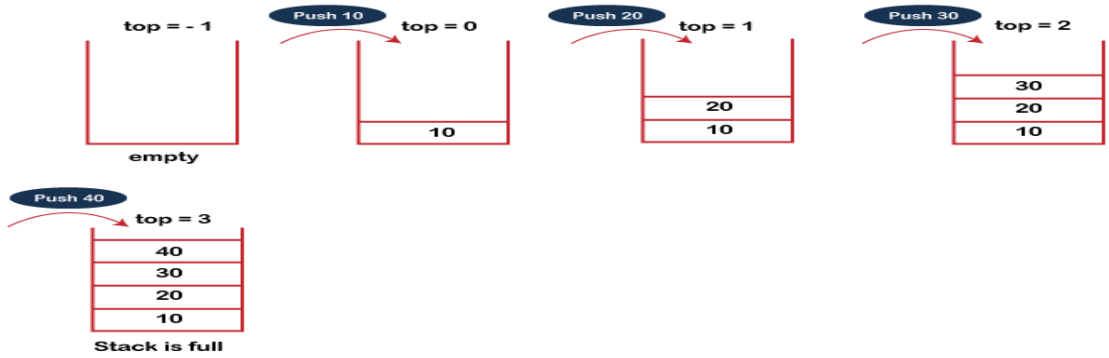
The following are some common operations implemented on the stack:

- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full even if you try to push one more element into the stack 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 condition occurs.
- **isEmpty():** It determines whether the stack is empty or not.
- **isFull():** It determines whether the stack is full or not.'
- **count():** It returns the total number of elements available in a stack.
- **display():** It prints all the elements available in the stack.

PUSH operation

The steps involved in the PUSH operation is given below:

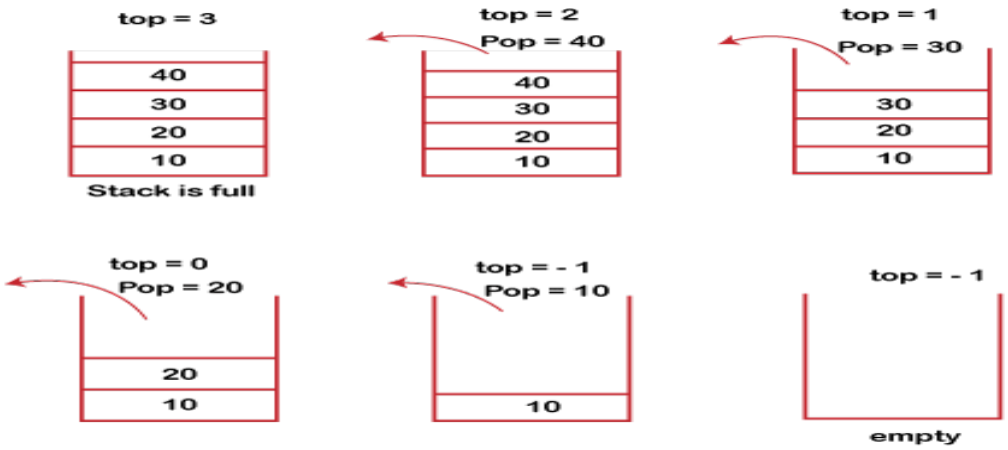
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the *overflow* condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the *max* size of the stack.



POP operation

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the *underflow* condition occurs.
- If the stack is not empty, we first access the element which is pointed by the *top*
- Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.



Representation of stacks

A stack can be represented in two ways

1. Array Representation

2. Linked List Representation

Example:

//Array representation of Stack

```
#include<stdio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         {
            push();
            break;
         }
         case 2:
         {
```

```
            pop();
            break;
         }
         case 3:
         {
            display();
            break;
         }
         case 4:
         {
            printf("\n\t EXIT POINT ");
            break;
         }
         default:
         {
            printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
         }
      }
   }
   while(choice!=4);
   return 0;
}
void push()
{
   if(top>=n-1)
   {
      printf("\n\tSTACK is over flow");
          }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
        }
```

```
}
void pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
   }
   else
   {
      printf("\n\t The popped elements is %d",stack[top]);
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);
      printf("\n Press Next Choice");
   }
   else
   {
      printf("\n The STACK is empty");
   }
 }
```

Output:

```
Enter the size of STACK[MAX=100]:10
        STACK OPERATIONS USING ARRAY
        --------------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
```

```
Enter the Choice:1
Enter a value to be pushed:12

Enter the Choice:1
Enter a value to be pushed:24

Enter the Choice:1
Enter a value to be pushed:98

Enter the Choice:3

The elements in STACK

98
24
12
Press Next Choice
Enter the Choice:2
        The popped elements is 98
Enter the Choice:3

The elements in STACK

24
12
Press Next Choice
Enter the Choice:4

        EXIT POINT
```

Example:

```
//Linked List Representation of Stacks
#include <stdio.h>
#include <stdlib.h>
struct node
 {
```

```
  int info;
  struct node *ptr;
}*top,*top1,*temp;
int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();
int count = 0;
void main()
 {
  int no, ch, e;
  printf("\n 1 - Push");
  printf("\n 2 - Pop");
  printf("\n 3 - Top");
  printf("\n 4 - Empty");
  printf("\n 5 - Exit");
  printf("\n 6 - Display");
  printf("\n 7 - Stack Count");
  printf("\n 8 - Destroy stack");
  create();
  while (1)
   {
    printf("\n Enter choice : ");
    scanf("%d", &ch);
    switch (ch){
      case 1:
            printf("Enter element : ");
            scanf("%d", &no);
            push(no);
            break;
      case 2:
           pop();
           break;
      case 3:
           if (top == NULL)
           printf("stack is empty");
```

```
            else
              {
                e = topelement();
                printf("\n Top element : %d", e);
              }
            break;
        case 4:
              empty();
              break;
        case 5:
             exit(0);
        case 6:
            display();
            break;
        case 7:
           stack_count();
           break;
        case 8:
           destroy();
           break;
        default :
           printf(" wrong choice:Try again ");
           break;
     }
  }
}
//empty stack
void create()
    {
  top = NULL;
       }
void stack_count()
    {
  printf("\n no: of elements in stack : %d", count);
       }
//push data
void push(int data)
   {
  if (top == NULL)
      {
```

```
      top =(struct node *)malloc(1*sizeof(struct node));
      top->ptr = NULL;
      top->info = data;
             }
   else
    {
      temp =(struct node *)malloc(1*sizeof(struct node));
      temp->ptr = top;
      temp->info = data;
      top = temp;
          }
   count++;
         }
void display()
    {
   top1 = top;
   if (top1 == NULL){
      printf("empty stack");
      return;
             }
   while (top1 != NULL)
       {
      printf("%d ", top1->info);
      top1 = top1->ptr;
   }
}
void pop()
   {
   top1 = top;
   if (top1 == NULL)
      {
      printf("\n error");
      return;
         }
   else
      top1 = top1->ptr;
      printf("\n Popped value : %d", top->info);
      free(top);
      top = top1;
      count--;
```

```
    }
  int topelement()
   {
    return(top->info);
       }
  //check stack empty or not
  void empty()
    {
    if (top == NULL)
      printf("\n empty stack");
    else
      printf("\n stack not empty with %d values", count);
    }
    void destroy()
       {
      top1 = top;
    while (top1 != NULL)
       {
      top1 = top->ptr;
      free(top);
    top = top1;
    top1 = top1->ptr;
        }
  free(top1);
  top = NULL;
  printf("\n all are destroyed");
  count = 0;
}
```

Output:

1 - Push

2 - Pop

3 - Top

4 - Empty

5 - Exit

6 - Display

7 - Stack Count

8 - Destroy stack

Enter choice : 1

Enter element : 10

```
Enter choice : 1
Enter element : 20

Enter choice : 1
Enter element : 30

Enter choice : 3

Top element : 30
Enter choice : 6
30 20 10
Enter choice : 7

no: of elements in stack : 3
Enter choice : 4

stack not empty with 3 values
Enter choice : 2
Popped value : 30
Enter choice : 8

all are destroyed
Enter choice : 6
empty stack
Enter choice : 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Stack Applications

In a stack, only limited operations are performed because it is restricted data structure.

Following are the applications of stack:

1. Expression Evaluation
2. Expression Conversion

Expression Evaluation

- In the C programming language, an expression is evaluated based on the operator precedence and associativity.
- When there are multiple operators in an expression, they are evaluated according to their precedence and associativity.
- The operator with higher precedence is evaluated first and the operator with the least precedence is evaluated last.

Parsing Expressions

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others.

Example:

a+b * c

will be evaluate multiplication first and later use addition operator.

Associativity

- Associativity describes the rule where operators with the same precedence appear in an expression.
- For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators.
- Here, both + and – are left associative, so the expression will be evaluated as

  (a + b) − c.
- Precedence and associativity determines the order of evaluation of an expression.

Following is an operator precedence and associativity table (highest to lowest) –

SNO	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (∗) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators.

At any point of time in expression evaluation, the order can be altered by using parenthesis.

Example:

In a + b*c, the expression part b*c will be evaluated first, with multiplication as precedence over addition.

We here use parenthesis for a + b to be evaluated first, like (a + b)*c.

Expression Conversion

- The way to write arithmetic expression is known as a notation.
- An arithmetic expression can be written in three different equivalent notations, i.e., without changing the essence or output of an expression.

These notations are −

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression.

Infix Notation

- When the operator is written in between the operands, then it is known as infix notation.

Example:

a - b + c, where operators are used in-between operands.

It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices.

Prefix Notation

- In prefix notation, an operator comes before the operands.
- Prefix notation is also known as Polish Notation.

Example:

+ab. This is equivalent to its infix notation a + b.

Postfix Notation

- This notation style is also known as Reversed Polish Notation.
- In this notation, the operator is postfixed to the operands i.e., the operator is written after the operands.
- The postfix expression is an expression in which the operator is written after the operands.

Example: ab+. This is equivalent to its infix notation a + b.

The following table briefly tries to show the difference in all three notations –

SNO	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) ∗ c	∗ + a b c	a b + c ∗
3	a ∗ (b + c)	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) ∗ (c + d)	∗ + a b + c d	a b + c d + ∗

6	((a + b) ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

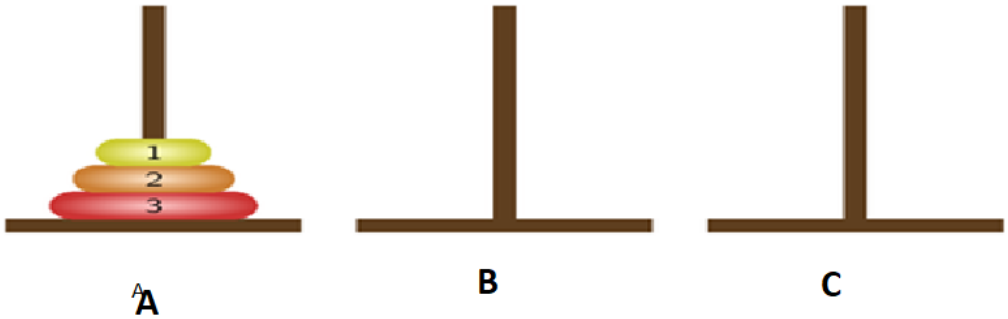
Tower of Hanoi Problem

- Before getting started, let's talk about what the Tower of Hanoi problem is.
- Well, this is a fun puzzle game where the objective is to move an entire stack of disks from the source position to another position.

Three simple rules are followed:

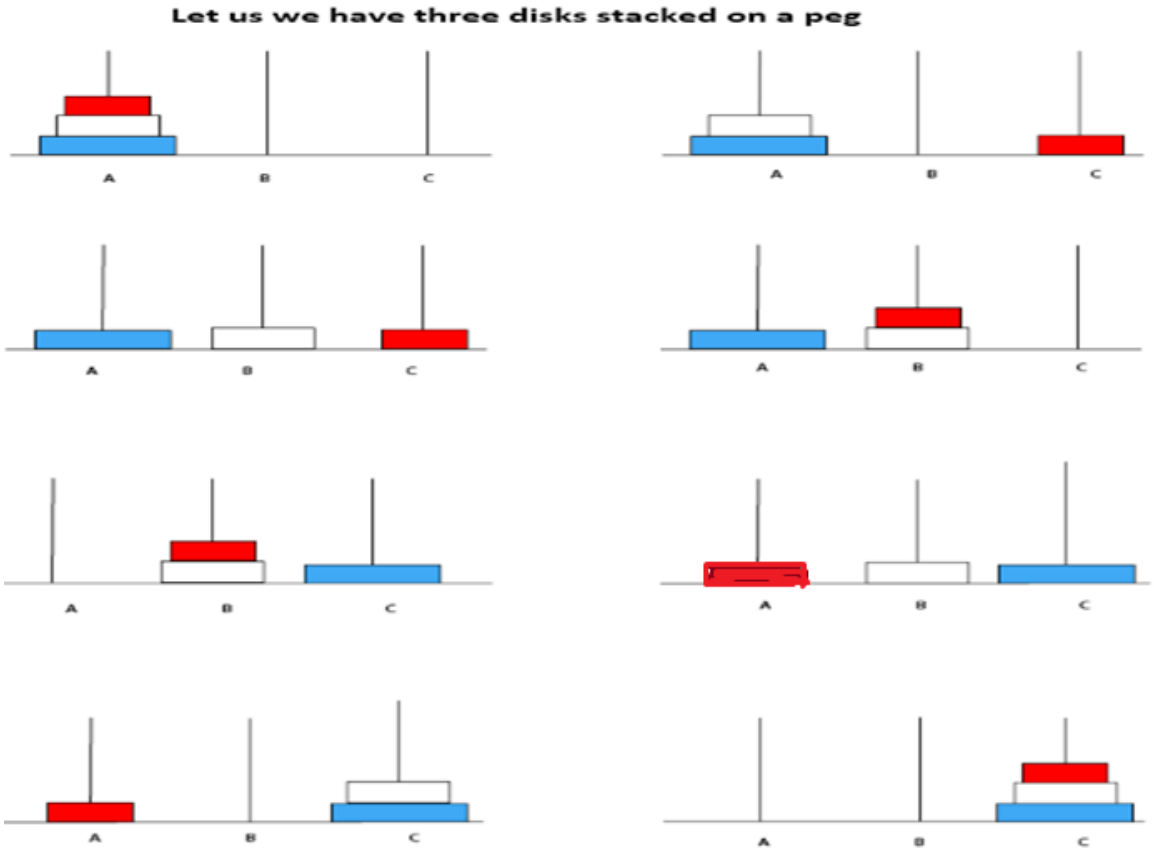
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack. In other words, a disk can only be moved if it is the uppermost disk on a stack.
3. No larger disk may be placed on top of a smaller disk.

Now, let's try to imagine a scenario. Suppose we have a stack of three disks. Our job is to move this stack from source A to destination C. How do we do this? Before we can get there, let's imagine there is an intermediate point B.



We can use B as a helper to finish this job. We are now ready to move on. Let's go through each of the steps:

1. Move the first disk from A to C
2. Move the first disk from A to B
3. Move the first disk from C to B
4. Move the first disk from A to C
5. Move the first disk from B to A
6. Move the first disk from B to C
7. Move the first disk from A to C



Queue

- Queue is an abstract data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.

For example, people waiting in line for a rail ticket form a queue.

Queue Representation

- A queue can also be implemented by using Arrays & Linked-lists
- We shall implement queues using one-dimensional array.

Operations on Queue

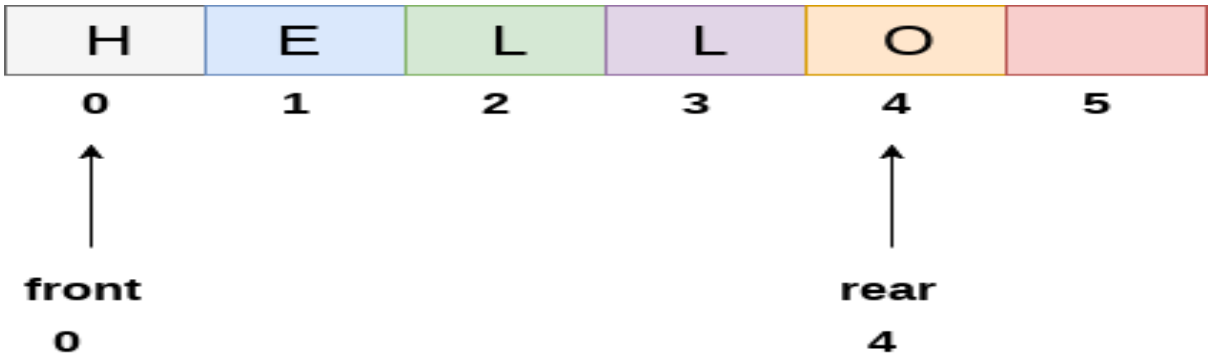
- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value. The dequeue operation can also be designed to void.
- **Queue overflow (isfull):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

A Queue can be represented as a container opened from both the sides in which the element can be enqueued from one side and dequeued from another side

Array representation of Queue

- We can easily represent queue by using linear arrays.
- There are two variables i.e. front and rear, that are implemented in the case of every queue.
- Front and rear variables point to the position from where insertions and deletions are performed in a queue.
- Initially, the value of front is -1 which represents an empty queue.

Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



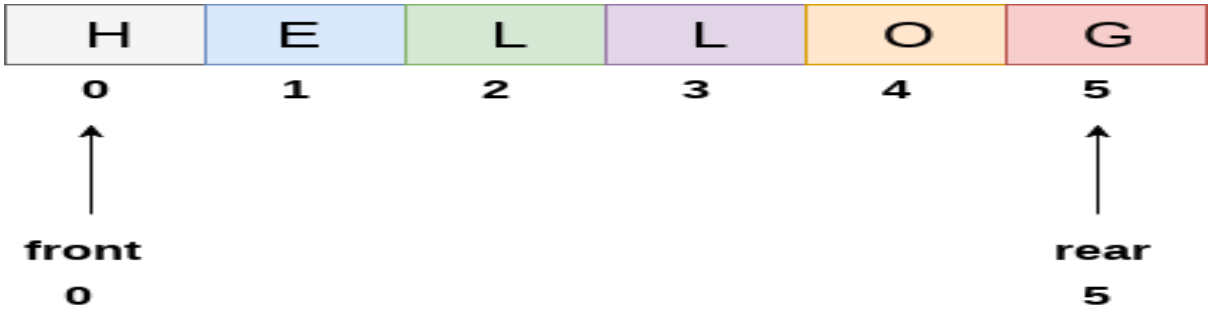
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**.

Since, No deletion is performed in the queue till now

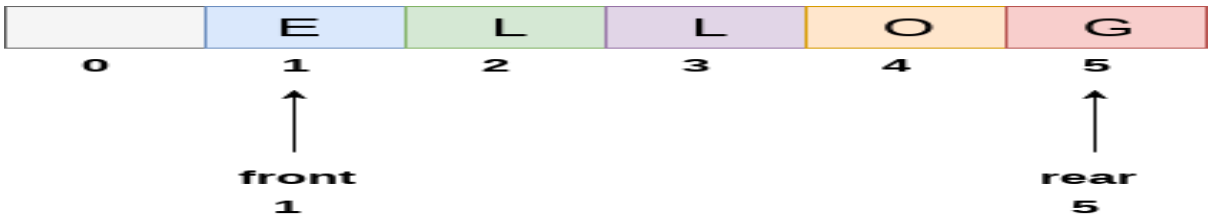
However, the value of rear increases by one every time an insertion is performed in the queue.

After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from 0 to 1. however, the queue will look something like following.



Queue after deleting an element

Example:

```
#include <stdio.h>
#include<stdlib.h>
#define MAX 5
void insert();
void delete();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
int main()
{
int choice;
while (1)
{
```

```
printf("\n 1.Insert ");
printf("\n 2.Delete ");
printf("\n 3.Display ");
printf("\n 4.Quit ");
printf("\n Enter your choice : ");
scanf("%d", &choice);
switch(choice)
{
case 1:
        insert();
        break;
case 2:
        delete();
        break;
case 3:
        display();
        break;
case 4:
        exit(1);
default:
         printf("Wrong choice n");
}
}
}
void insert()
{
int item;
if(rear == MAX - 1)
printf("Queue Overflow n");
else
{
if(front== - 1)
front = 0;
printf("Inset the element in queue : ");
scanf("%d", &item);
```

```
rear = rear + 1;
queue_array[rear] = item;
}
}
void delete()
{
if(front == - 1 || front > rear)
{
printf("Queue Underflow n");
return;
}
else
{
printf("Element deleted from queue is : %dn", queue_array[front]);
front = front + 1;
}
}
void display()
{
int i;
if(front == - 1)
printf("Queue is empty n");
else
{
printf("Queue is : n");
for(i = front; i <= rear; i++)
printf("%d ", queue_array[i]);
printf("n");
}
}
```

Output:

```
1.Insert
 2.Delete
 3.Display
 4.Quit
```

Enter your choice : 1

Inset the element in queue : 1000

1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1

Inset the element in queue : 2000

1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3

Queue is :
1000 2000
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1

Inset the element in queue : 3000

1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1

Inset the element in queue : 4000

1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3

Queue is :
1000 2000 3000 4000
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 2
Element deleted from queue is : 1000

1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3

Queue is :
2000 3000 4000
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 2
Element deleted from queue is : 2000

1.Insert
2.Delete
3.Display
4.Quit

Enter your choice : 3

Queue is :
3000 4000
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 4

...Program finished with exit code 0
Press ENTER to exit console.

Linked List implementation of Queue

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part.

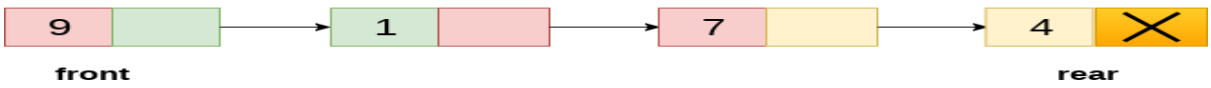
Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer.

The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively.

If front and rear both are NULL, it indicates that the queue is empty.



Linked Queue

Example:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
```

```
    int data;
    struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*************************Main
Menu*****************************\n");

printf("\n==========================================================
\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",& choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
```

```
            break;
            default:
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    struct node *ptr;
    int item;

    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    else
    {
        printf("\nEnter value?\n");
        scanf("%d",&item);
        ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
        else
        {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
        }
```

```
    }
}
void delete ()
{
    struct node *ptr;
    if(front == NULL)
    {
        printf("\nUNDERFLOW\n");
        return;
    }
    else
    {
        ptr = front;
        front = front -> next;
        free(ptr);
    }
}
void display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
    {
        printf("\nEmpty queue\n");
    }
    else
    {   printf("\nprinting values .....\n");
        while(ptr != NULL)
        {
            printf("\n%d\n",ptr -> data);
            ptr = ptr -> next;
        }
    }
}
```

Output:

**********Main Menu**********

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
100

**********Main Menu**********

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
200

**********Main Menu**********

==========

1.insert an element

2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter value?
300

**********Main Menu**********

==========

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

100

200

300
**********Main Menu**********

==========

1.insert an element
2.Delete an element

3.Display the queue
4.Exit

Enter your choice ?2

**********Main Menu**********

==========
=

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

200

300

**********Main Menu**********

==========
=

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4

...Program finished with exit code 0
Press ENTER to exit console.

Types of Queues

There are four types of Queues:

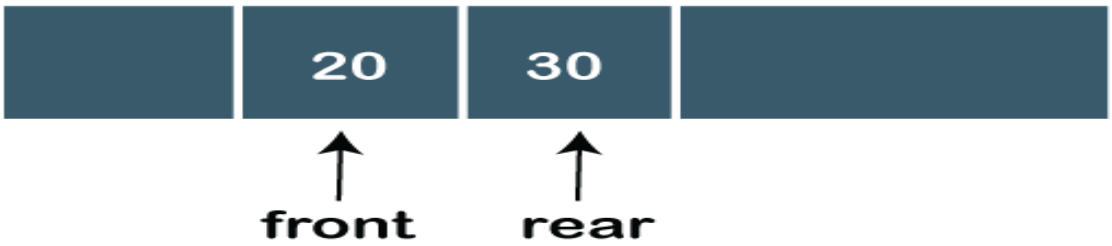
1.Linear Queue

- In Linear Queue, an insertion takes place from one end while the deletion occurs from another end.
- The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end.
- It strictly follows the FIFO rule.

The linear Queue can be represented, as shown in the below figure:

The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion.

If we want to show the deletion, then it can be represented as:



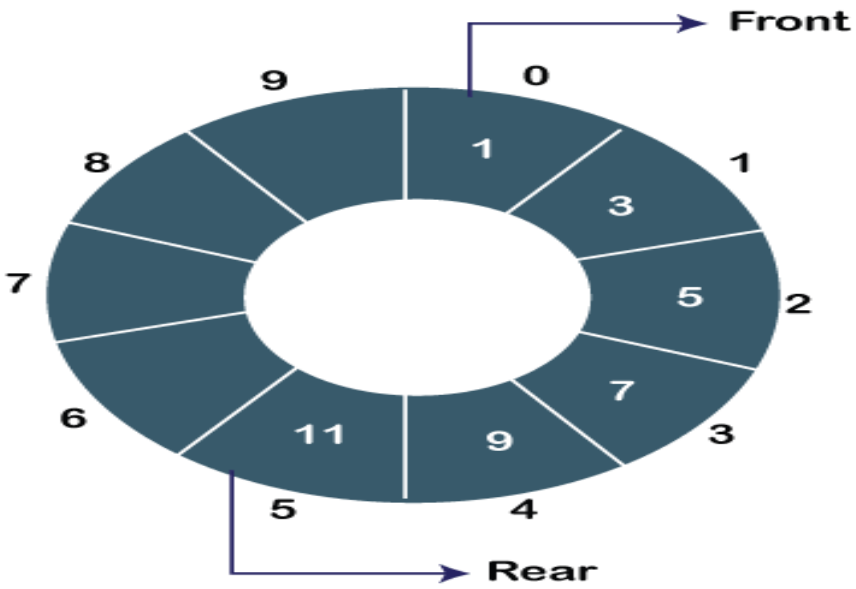
- In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

- The major drawback of using a linear Queue is that insertion is done only from the rear end.
- If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue.
- In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

2.Circular Queue

- In Circular Queue, all the nodes are represented as circular.
- It is similar to the linear Queue except that the last element of the queue is connected to the first element.
- It is also known as Ring Buffer as all the ends are connected to another end.

The circular queue can be represented as:



- The drawback that occurs in a linear queue is overcome by using the circular queue.
- If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear.

3. Priority Queue

A priority queue is another special type of Queue data structure in which each element has some priority associated with it.

Based on the priority of the element, the elements are arranged in a priority queue.

If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

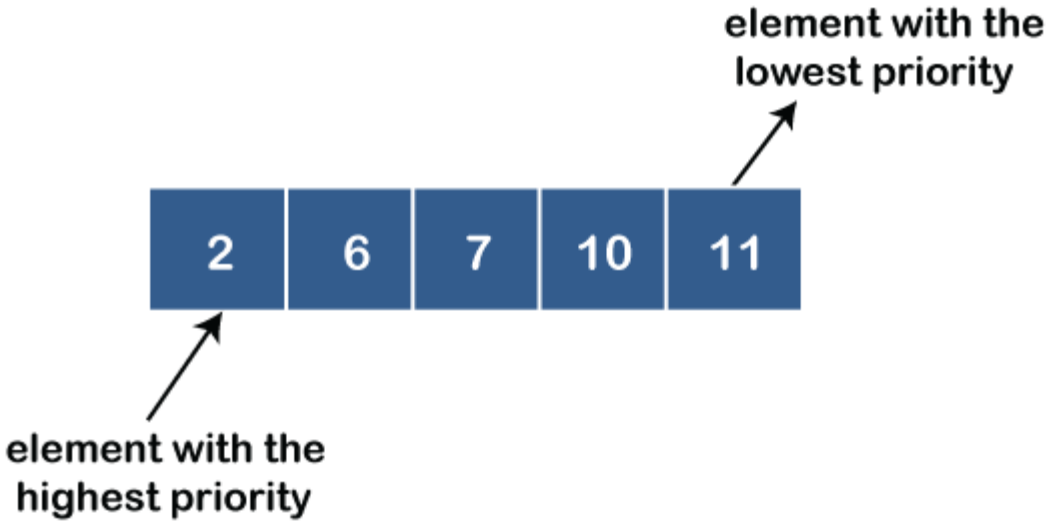
- poll(): This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- add(2): This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- poll(): It will remove '2' element from the priority queue as it has the highest priority queue.
- add(5): It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

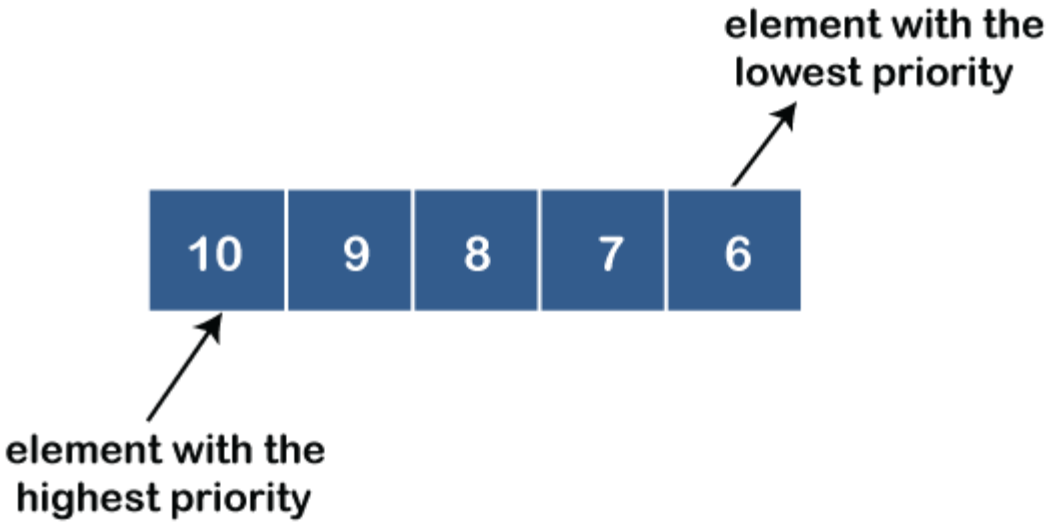
There are two types of priority queue:

- Ascending order priority queue: In ascending order priority queue, a lower priority number is given as a higher priority . For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5;

therefore, the smallest number, i.e., 1 is given as the highest priority.



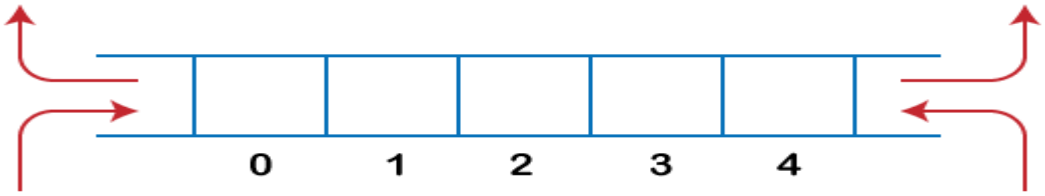
- Descending order priority queue: In descending order priority queue, a higher priority number is given as a higher priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority



4. Deque

The dequeue stands for Double Ended Queue.

In the queue, the insertion takes place from one end while the deletion takes place from another end. The end at which the insertion occurs is known as the **rear end** whereas the end at which the deletion occurs is known as **front end**.



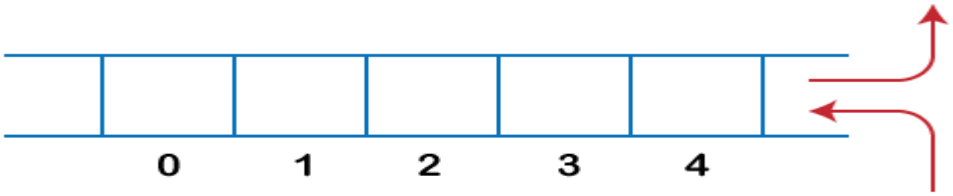
Deque is a linear data structure in which the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Let's look at some properties of deque.

- Deque can be used both as **stack** and **queue** as it allows the insertion and deletion operations on both ends.

In deque, the insertion and deletion operation can be performed from one side.

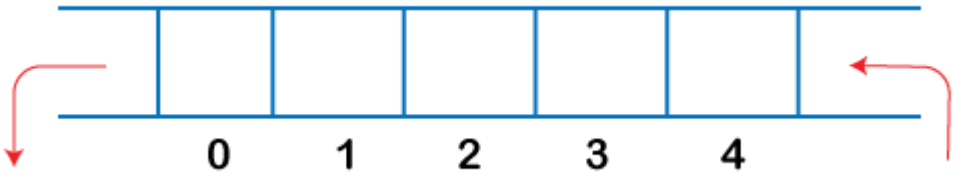
The stack follows the LIFO rule in which both the insertion and deletion can be performed only from one end; therefore, we conclude that deque can be considered as a stack.



In deque, the insertion can be performed on one end, and the deletion can be done on another end.

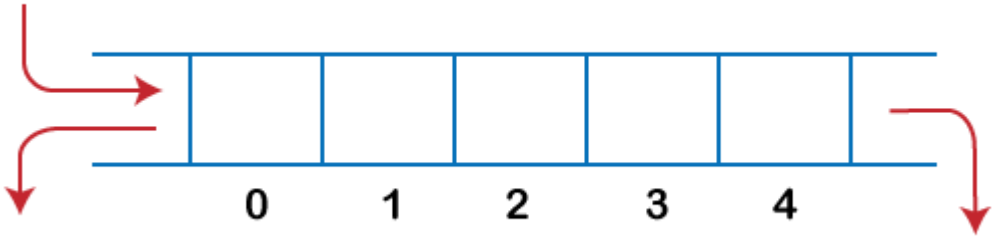
The queue follows the FIFO rule in which the element is inserted on one end and deleted from another end.

Therefore, we conclude that the deque can also be considered as the queue.

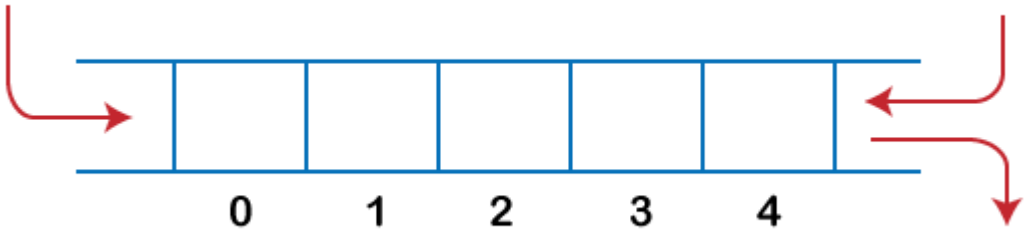


There are two types of Queues, **Input-restricted queue**, and **output-restricted queue**.

1. **Input-restricted queue:** The input-restricted queue means that some restrictions are applied to the insertion. In input-restricted queue, the insertion is applied to one end while the deletion is applied from both the ends.



2. **Output-restricted queue:** The output-restricted queue means that some restrictions are applied to the deletion operation. In an output-restricted queue, the deletion can be applied only from one end, whereas the insertion is possible from both ends.



Dynamic memory allocations

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*.

Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

- The malloc() function allocates single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type*)malloc(byte-size)

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
 {
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
   if(ptr==NULL)
   {
      printf("Sorry! unable to allocate memory");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
}
```

Output:

Enter number of elements: 3

Enter elements of array: 10

20

30

Sum=60

calloc() function in C

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

```
ptr=(cast-type*)calloc(number, byte-size)
```

Example:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int n,i,*ptr,sum=0;
   printf("Enter number of elements: ");
   scanf("%d",&n);
   ptr=(int*)calloc(n,sizeof(int));  //memory allocated using calloc
   if(ptr==NULL)
   {
      printf("Sorry! unable to allocate memory");
      exit(0);
   }
   printf("Enter elements of array: ");
   for(i=0;i<n;++i)
   {
      scanf("%d",ptr+i);
      sum+=*(ptr+i);
   }
   printf("Sum=%d",sum);
   free(ptr);
return 0;
```

}

Output:

Enter number of elements: 3

Enter elements of array: 100

200

300

Sum=600

realloc() function in C

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function.
- In short, it changes the memory size.

Let's see the syntax of realloc() function.

```
ptr=realloc(ptr, new-size)
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int n = 4, i, *p, s = 0;
  p = (int*) calloc(n, sizeof(int));
  if(p == NULL)
    {
     printf("\nError! memory not allocated.");
     exit(0);
  }
  printf("\nEnter elements of array : ");
```

```
  for(i = 0; i < n; ++i)
    {
    scanf("%d", p + i);
    s += *(p + i);
       }
  printf("\nSum : %d", s);
  p = (int*) realloc(p, 6);
  printf("\nEnter elements of array : ");
  for(i = 0; i < n; ++i)
    {
    scanf("%d", p + i);
    s += *(p + i);
        }
  printf("\nSum : %d", s);
  return 0;
   }
```

Output: Enter elements of array : 3 34 28 8

Sum : 73
Enter elements of array : 3 28 33 8 10 15
Sum : 145

free() function in C

- The memory occupied by malloc() or calloc() functions must be released by calling free() function.
- Otherwise, it will consume memory until program exit.

Let's see the syntax of free() function.

free(ptr)

Example:

```
#include <stdio.h>
int main()
 {
int* ptr = malloc(10 * sizeof(*ptr));
if (ptr != NULL)
 {
 *(ptr + 2) = 50;
 printf("Value of the 2nd integer is %d",*(ptr + 2));
      }
free(ptr);
    }
```

Output:

Value of the 2nd integer is 50