303105201 - Design of Data Structures

UNIT-2

->Stacks, Recursion and Queue:

->Stacks: Definition, Stack Operations

  Array Representation of Stacks

  Stacks using Dynamic Arrays

->Stack Applications: Polish notation

  Infix to postfix conversion

  Evaluation of postfix expression.

->Recursion -Factorial, GCD

  Fibonacci sequence

  Tower of Hanoi

->Queues: Definition

  Array Representation

  Queue Operations

  Circular Queues

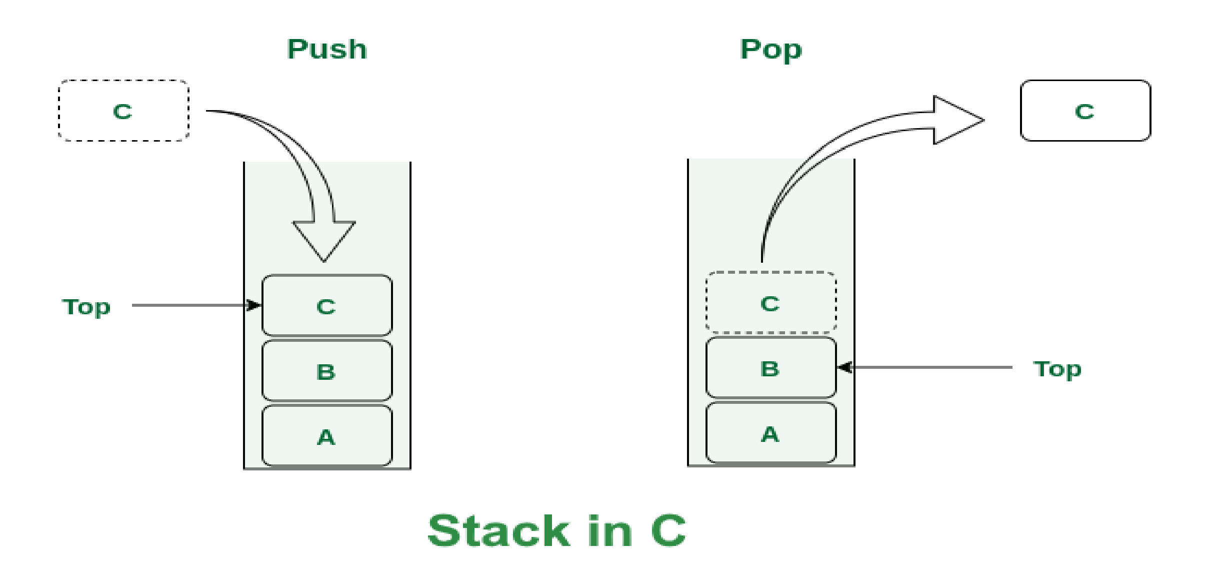
  Circular queues using Dynamic arrays

  Deque

  Priority Queues and its problems

Stacks:

Definition: A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. It means that the element which is inserted last is the first one to be removed.



Stack Operations:

Push: Adds an element to the top of the stack.

Pop: Removes the top element from the stack.

Peek (or Top): Returns the top element without removing it.

isEmpty: Checks if the stack is empty.

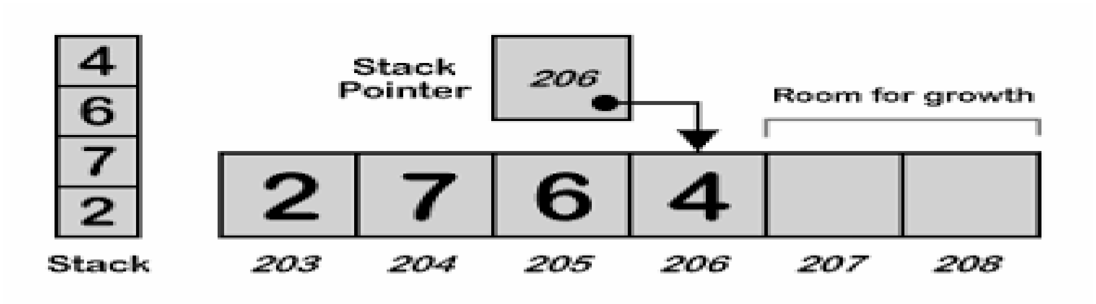
isFull: Checks if the stack is full (applicable in fixed-size implementations).

Array Representation of Stacks:

Stacks can be implemented using arrays. In this representation, a stack pointer (top) indicates the top element of the stack.

When pushing an element, increment top and store the element at that position.

When popping an element, return the element at top and decrement top.



Stacks using Dynamic Arrays:

Dynamic arrays can be used to implement stacks that can grow and shrink dynamically.

Initially, allocate memory for a small array. When the stack grows beyond its capacity, reallocate memory to expand it.

This implementation requires tracking the capacity and the current size of the stack.

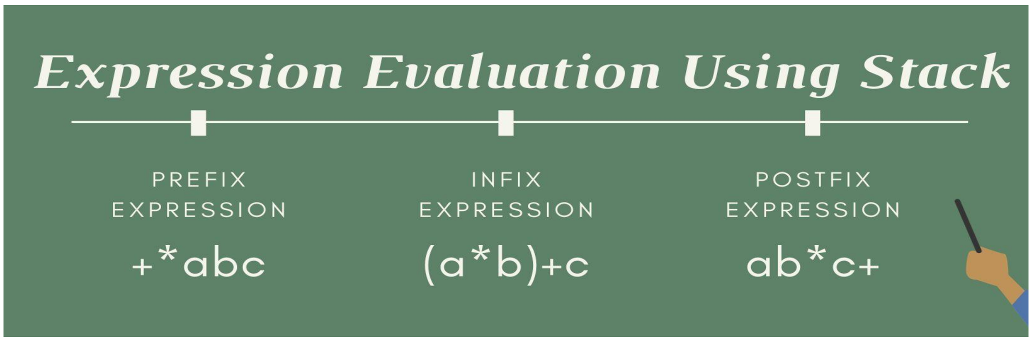
Polish Notation (Prefix, Infix, Postfix):

Polish notation, also known as prefix notation, is a mathematical notation in which every operator follows all of its operands. There are three forms of notation:

Prefix: Operators precede their operands (e.g., +AB).

Infix: Operators are placed between their operands (e.g., A+B).

Postfix: Operators follow their operands (e.g., AB+).



<u>Infix to Postfix Conversion:</u>

Infix to postfix conversion involves converting an infix expression to its postfix equivalent. This conversion is typically performed using a stack data structure. Here's a simple algorithm:

- Create an empty stack to hold operators.
- Scan the infix expression from left to right.
- If the scanned character is an operand, add it to the output.
- If the scanned character is an operator:
- Pop operators from the stack and add them to the output until an operator with lower precedence is encountered or the stack is empty.
- Push the scanned operator onto the stack.
- After scanning the entire expression, pop any remaining operators from the stack and add them to the output.

<u>Evaluation of Postfix Expression:</u>

Once the infix expression is converted to postfix notation, it can be evaluated using a stack. Here's the algorithm for evaluating a postfix expression:

- Create an empty stack to hold operands.
- Scan the postfix expression from left to right.
- If the scanned character is an operand, push it onto the stack.
- If the scanned character is an operator, pop the top two operands from the stack, apply the operator, and push the result back onto the stack.
- After scanning the entire expression, the result will be the top element of the stack.

Example:

Let's consider an infix expression "A + B * C" and perform infix to postfix conversion and evaluate the postfix expression:

Infix to Postfix Conversion:

Infix: A + B * C

Postfix: ABC*+

Evaluation of Postfix Expression:

Postfix: ABC*+

Scan from left to right:

Push A onto the stack: [A]

Push B onto the stack: [A, B]

Encounter *, pop B and A, calculate B * A = BA, push BA onto the stack: [BA]

Push C onto the stack: [BA, C]

Encounter +, pop C and BA, calculate BA + C = BAC, push BAC onto the stack: [BAC]

Result: BAC

Factorial

The factorial of a number n n (denoted as $n!$ n!) is the product of all positive integers less than or equal to n n.

```
#include <stdio.h>

// Function prototype

int factorial(int n);

int main() {
    int n;
    printf("Enter a non-negative integer: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("Factorial is not defined for negative numbers.\n");
    } else {
        int result = factorial(n);
        printf("Factorial of %d = %d\n", n, result);
    }
    return 0;
}
// Function to calculate factorial recursively
int factorial(int n) {
    // Base case: factorial of 0 is 1
    if (n == 0) {
        return 1;
    }
    // Recursive case: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}
```

Diagram representation

```
factorial(5)
  -> 5 * factorial(4)
        -> 4 * factorial(3)
              -> 3 * factorial(2)
                    -> 2 * factorial(1)
                          -> 1 (base case)
```

GCD (Greatest Common Divisor)

The GCD of two numbers is the largest number that divides both of them without leaving a remainder.

```
#include <stdio.h>
// Function prototype
int gcd(int a, int b);
int main() {
    int num1, num2;
    printf("Enter two integers: ");
    scanf("%d %d", &num1, &num2);
    // Calculate and display GCD
    int result = gcd(num1, num2);
    printf("GCD of %d and %d is %d\n", num1, num2, result);
    return 0;
}
// Recursive function to calculate GCD
int gcd(int a, int b) {
    // Base case
    if (b == 0) {
        return a;
    }
    // Recursive case
    else {
        return gcd(b, a % b);
    }
}
```

Diagram representation

```
gcd(56, 98)
  -> gcd(98, 56 % 98)
        -> gcd(56, 98 % 56)
              -> gcd(56, 42)
                    -> gcd(42, 56 % 42)
                          -> gcd(42, 14)
                                -> gcd(14, 42 % 14)
                                      -> gcd(14, 0)
                                            -> 14 (base case)
```

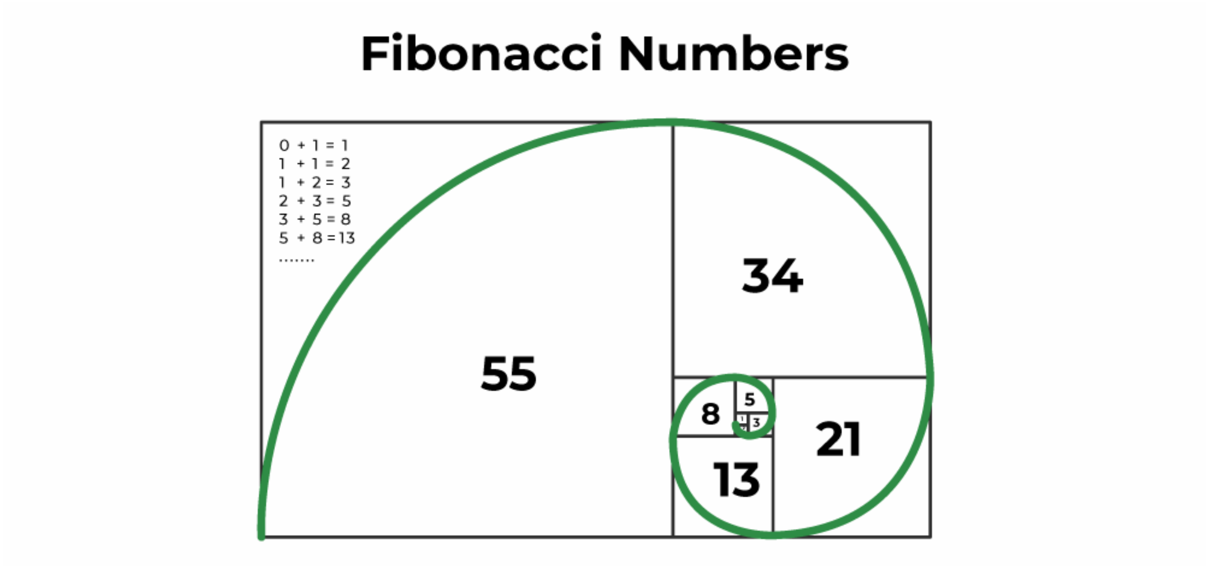
Fibonacci sequence

The Fibonacci sequence is a series of numbers where the next number is found by adding up the two numbers before it. The sequence starts with 0 and 1.

```
#include<stdio.h>
void printFibonacci(int n){
static int n1=0,n2=1,n3;
   if(n>0){
       n3 = n1 + n2;
       n1 = n2;
       n2 = n3;
       printf("%d ",n3);
       printFibonacci(n-1);
   }
}
int main(){
   int n;
   printf("Enter the number of elements: ");
   scanf("%d",&n);
   printf("Fibonacci Series: ");
   printf("%d %d ",0,1);
   printFibonacci(n-2);//n-2 because 2 numbers are already printed
  return 0;
}
```

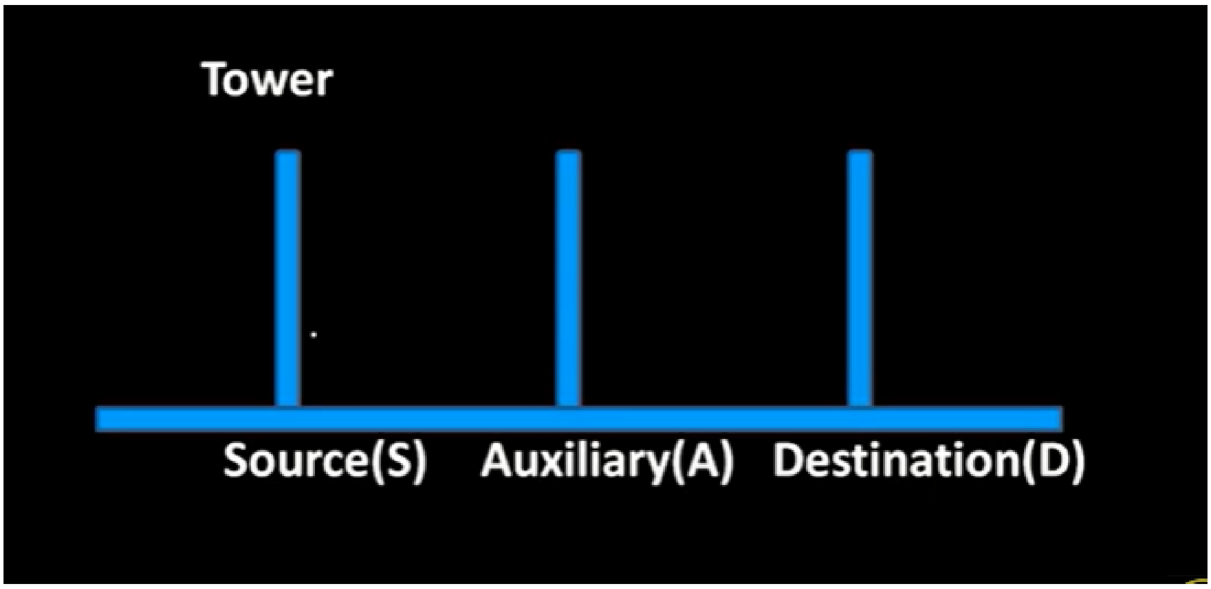
Diagram representation

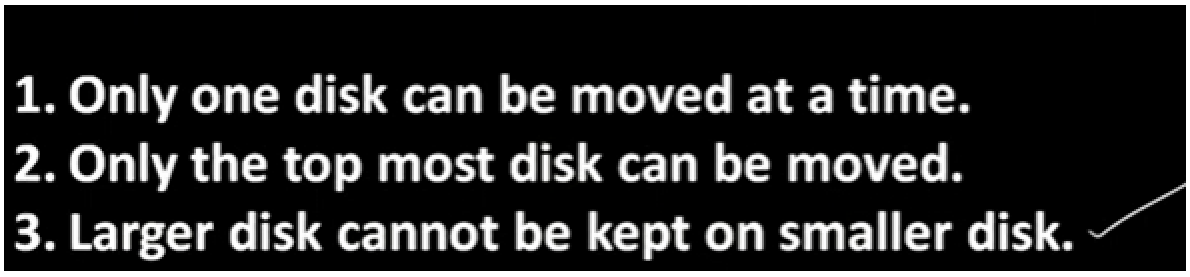
```
fibonacci(5)
  -> fibonacci(4) + fibonacci(3)
        -> (fibonacci(3) + fibonacci(2)) + (fibonacci(2) + fibonacci(1))
              -> ((fibonacci(2) + fibonacci(1)) + (fibonacci(1) + fibonacci(0)))
+ ((fibonacci(1) + fibonacci(0)) + 1)
                  -> (((fibonacci(1) + fibonacci(0)) + 1) + (1 + 0)) + ((1 + 0) + 1)
                      -> ((1 + 0 + 1) + 1) + (1 + 1)
                          -> (2 + 1) + 2
                              -> 3 + 2
                                  -> 5
```



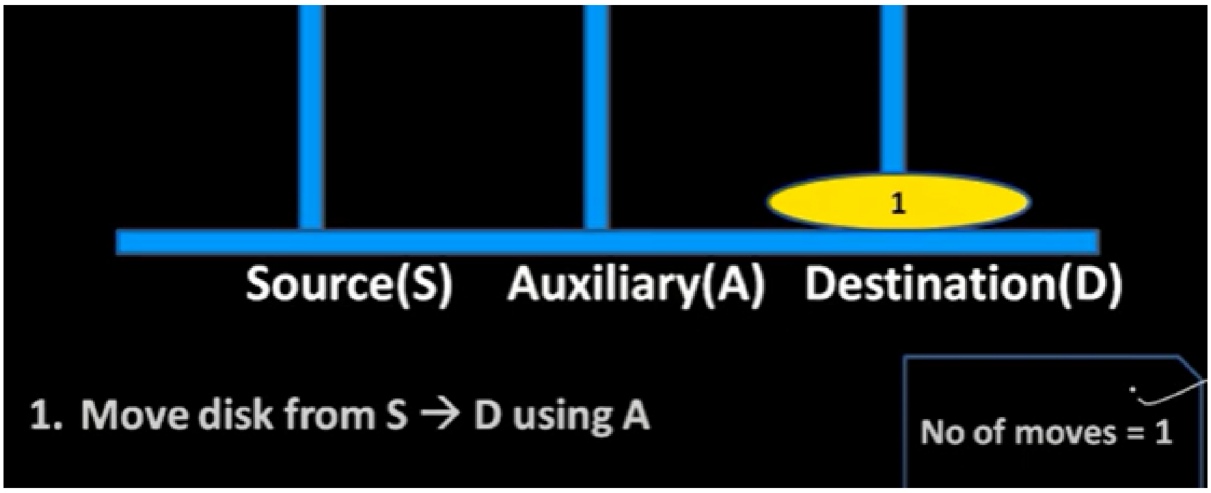
Tower of Hanoi

The Tower of Hanoi is a mathematical puzzle where you have three rods and a number of disks of different sizes that can slide onto any rod. The puzzle starts with the disks neatly stacked in ascending order of size on one rod, the smallest at the top, making a conical shape.




1. Only one disk can be moved at a time.
2. Only the top most disk can be moved.
3. Larger disk cannot be kept on smaller disk.

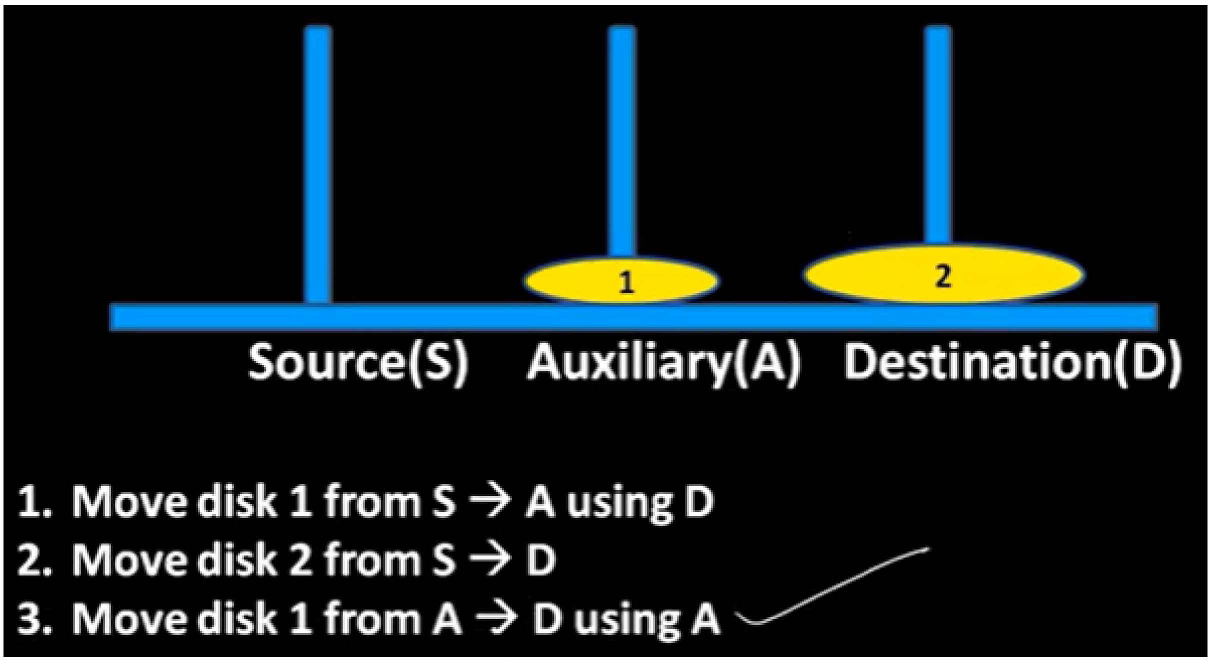
NO.OF DISKS-1



1. Move disk from S → D using A

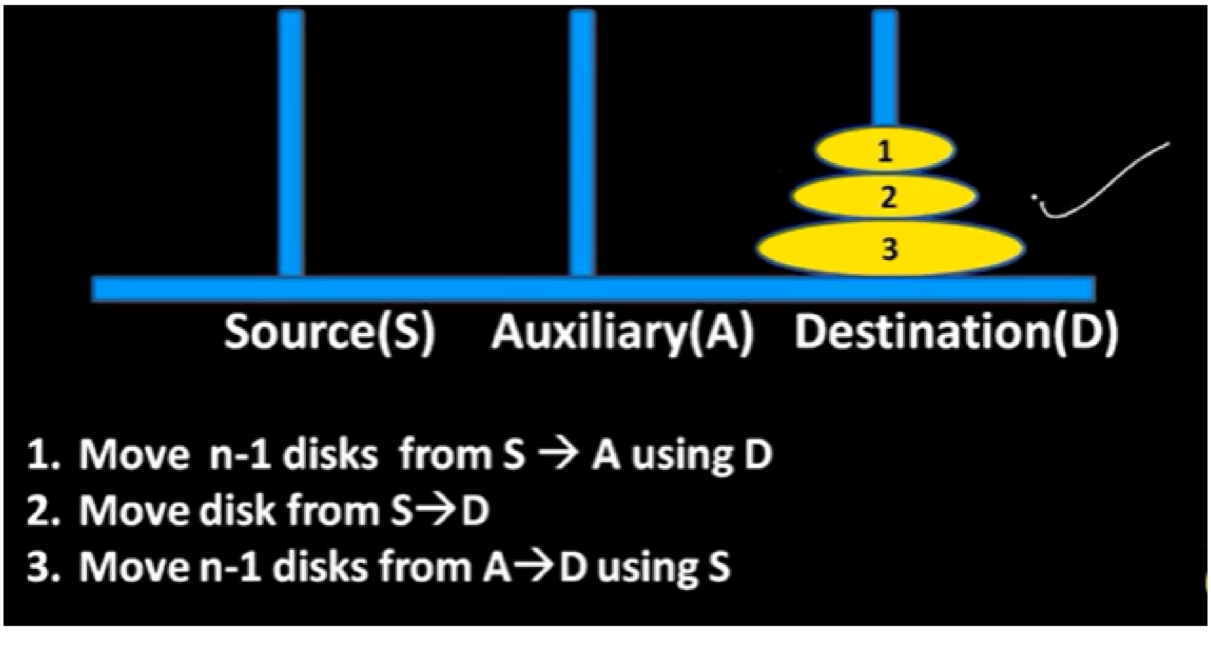
No of moves = 1

NO.OF DISKS-2



1. Move disk 1 from S → A using D
2. Move disk 2 from S → D
3. Move disk 1 from A → D using A

NO.OF DISKS-3

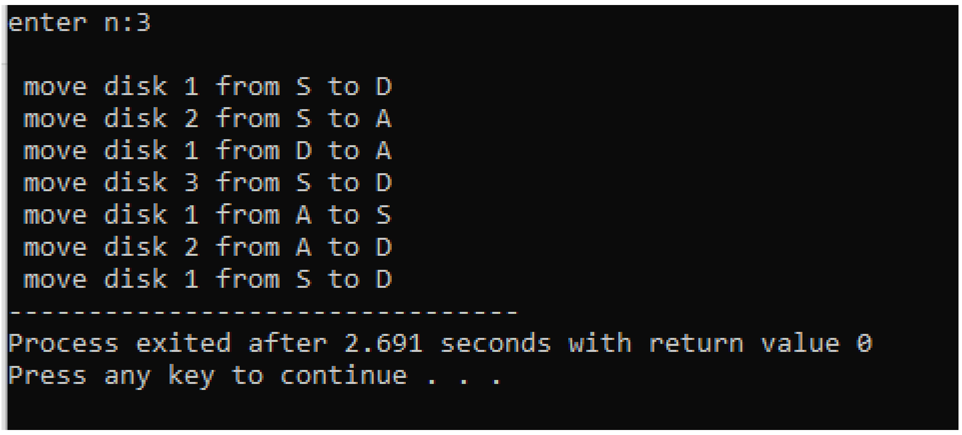


1. Move n-1 disks from S → A using D
2. Move disk from S→D
3. Move n-1 disks from A→D using S

Program:

```
#include<stdio.h>
#include<conio.h>
void TOH(int,char,char,char);
int main(){
        int n;
        //clrscr();
        printf("enter n:");
        scanf("%d",&n);
        TOH(n,'S','A','D');
        //getch();
}
void TOH(int n,char S,char A,char D){
        if(n>0){
                TOH(n-1,S,D,A);
                printf("\n move disk %d from %c to %c",n,S,D);
                TOH(n-1,A,S,D);
        }
}
```

OUTPUT:



```
enter n:3

 move disk 1 from S to D
 move disk 2 from S to A
 move disk 1 from D to A
 move disk 3 from S to D
 move disk 1 from A to S
 move disk 2 from A to D
 move disk 1 from S to D
--------------------------------
Process exited after 2.691 seconds with return value 0
Press any key to continue . . .
```

Diagram representation

For n=3

n=3:

Step 1: Move disk 1 from S to D

Step 2: Move disk 2 from S to A

Step 3: Move disk 1 from D to A

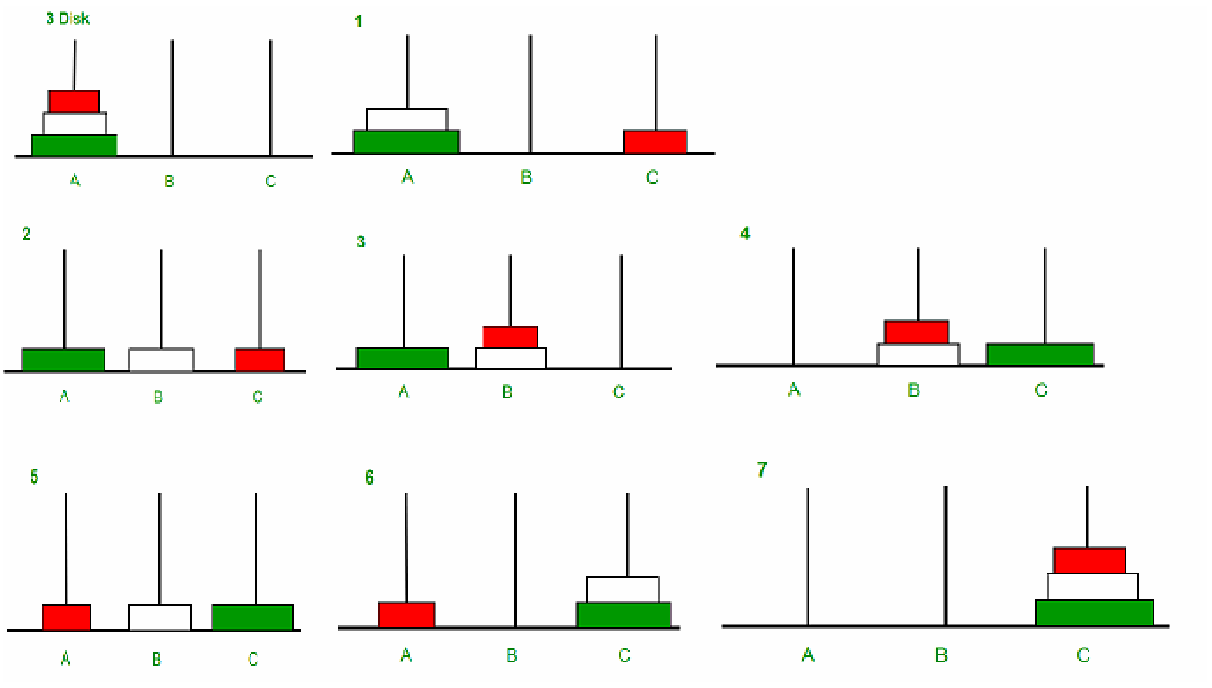
Step 4: Move disk 3 from S to D

Step 5: Move disk 1 from A to S

Step 6: Move disk 2 from A to D

Step 7: Move disk 1 from S to D

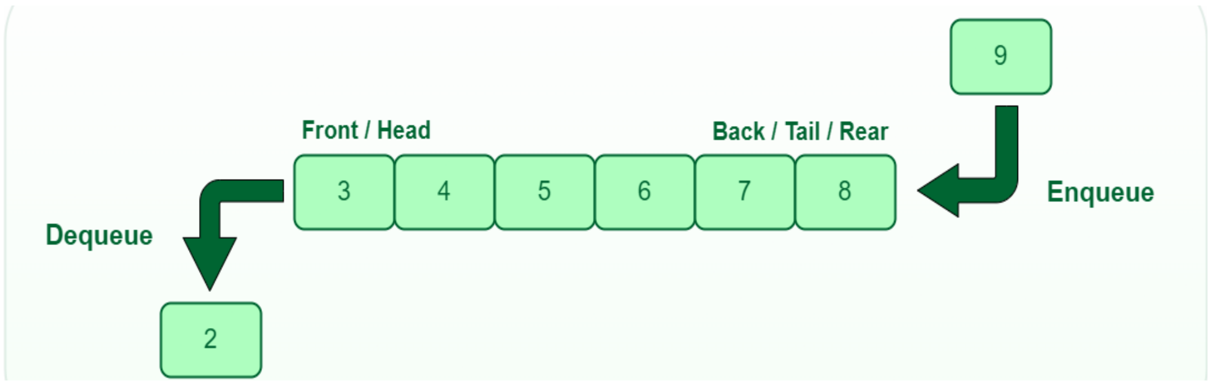
These steps show the sequence of moves to solve the Tower of Hanoi problem for 3 disks. Each step is determined recursively, breaking down the problem into smaller sub-problems.



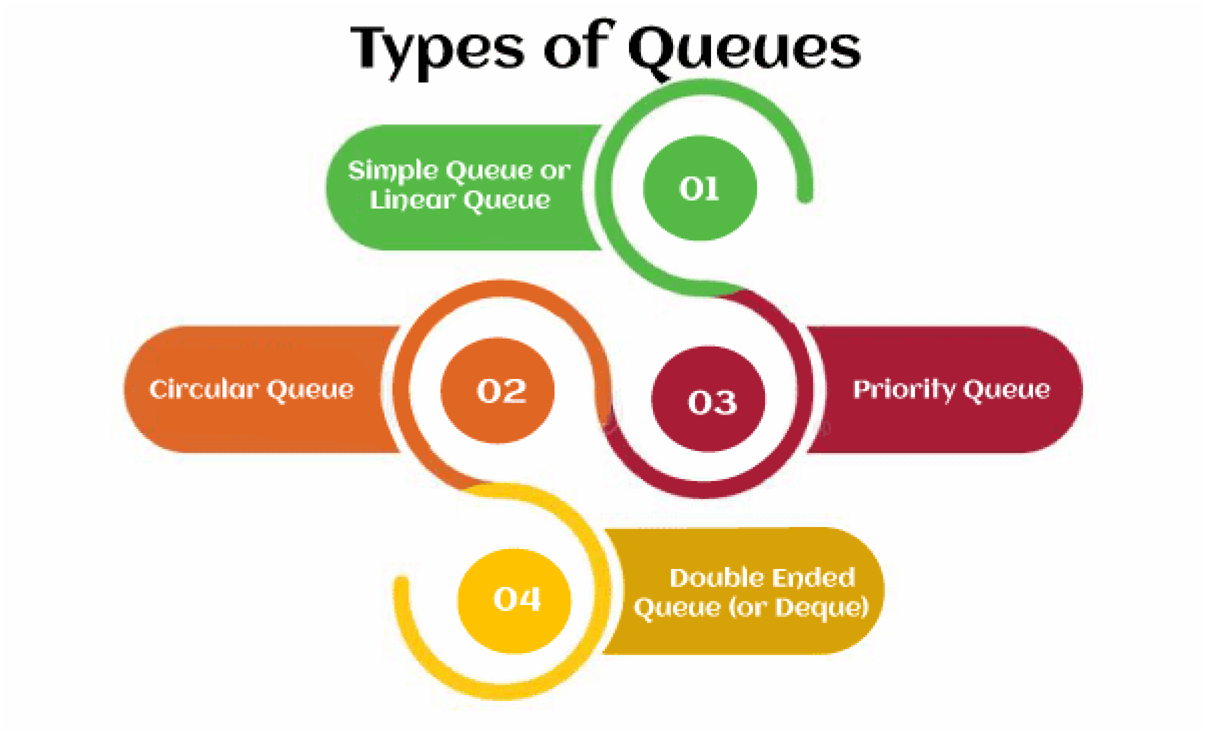
Queues:

Definition

A queue is a linear data structure that follows the First In First Out (FIFO) principle. This means that the element added first will be removed first.

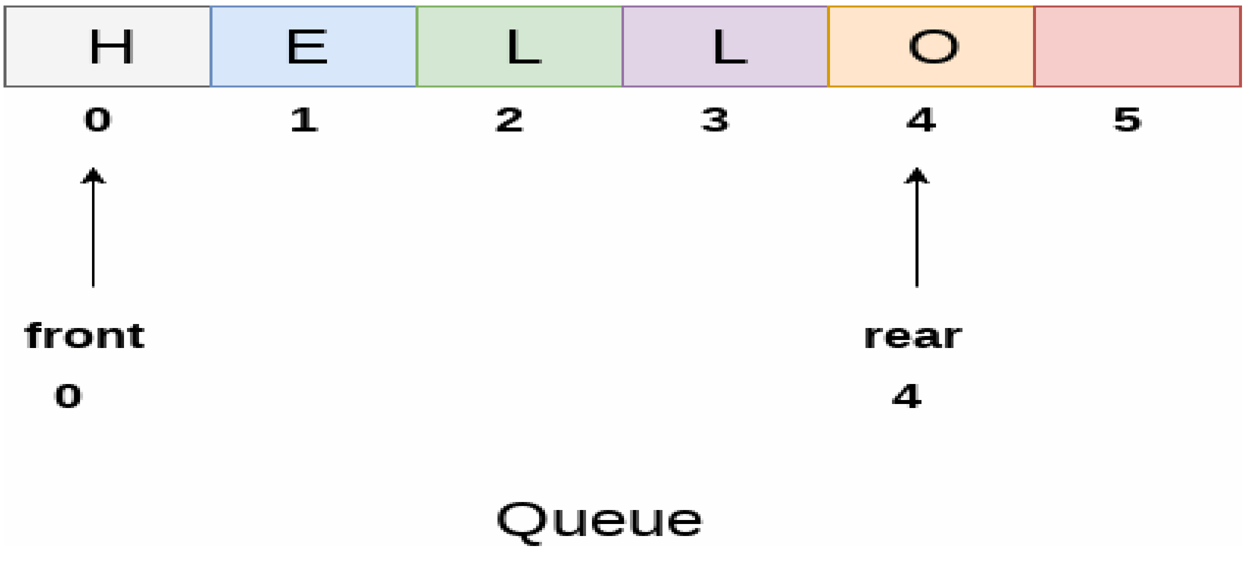


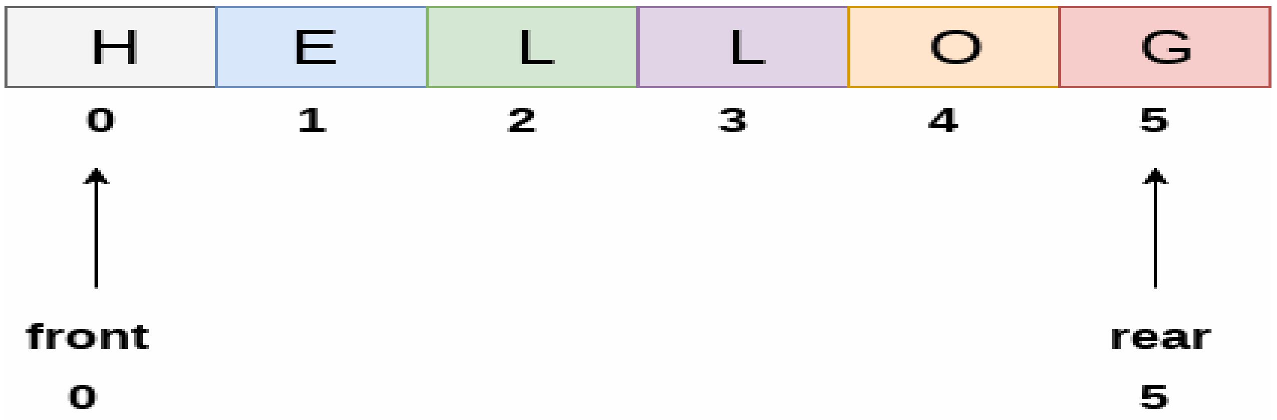
TYPES OF QUEUE:



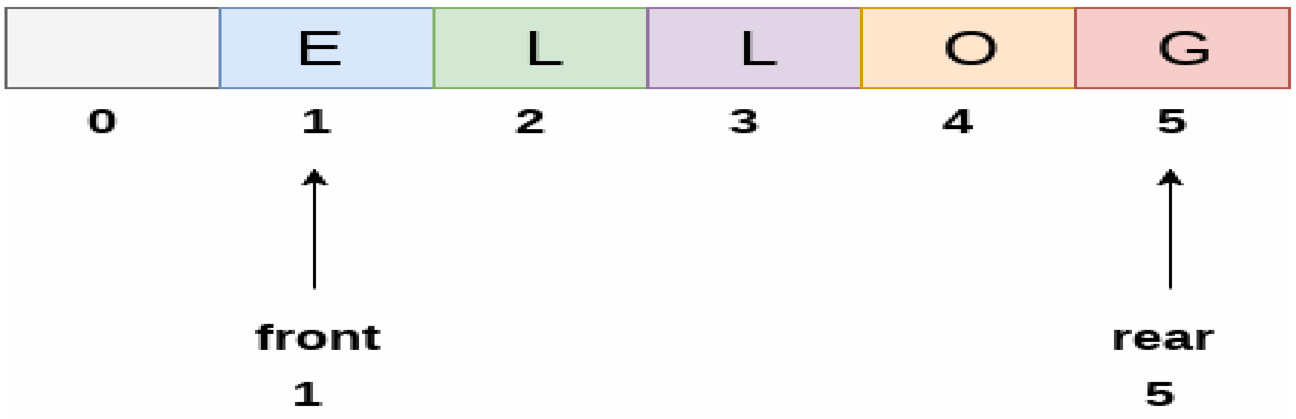
Array Representation of queue:

A queue can be represented using an array. An array is a fixed-size data structure that can store elements of the same type. In the context of a queue, the array will have two pointers: front and rear.





Queue after inserting an element



Queue after deleting an element

<u>Queue Operations</u>

Enqueue: Adding an element to the rear of the queue.

Dequeue: Removing an element from the front of the queue.

Peek/Front: Getting the element at the front of the queue without removing it.

IsEmpty: Checking if the queue is empty.

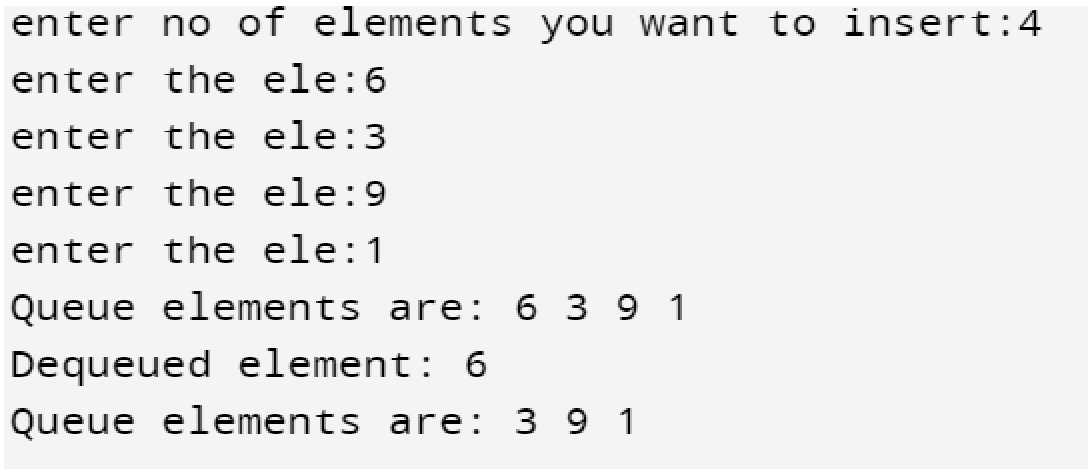
IsFull: Checking if the queue is full.

Example: (Implementation of Queue using Array in C)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
// Structure for Queue
typedef struct {
   int items[MAX_SIZE];
   int front;
   int rear;
} Queue;
// Function prototypes
void initializeQueue(Queue *q);
void enqueue(Queue *q, int value);
int dequeue(Queue *q);
int isEmpty(Queue *q);
int isFull(Queue *q);
void displayQueue(Queue *q);
// Initialize queue
void initializeQueue(Queue *q) {
   q->front = -1;
   q->rear = -1;
}
// Check if queue is empty
int isEmpty(Queue *q) {
    return (q->front == -1 && q->rear == -1);
}
// Check if queue is full
int isFull(Queue *q) {
    return (q->rear == MAX_SIZE - 1);
}
// Enqueue operation
void enqueue(Queue *q, int value) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue %d\n", value);
        return;
    }
    if (isEmpty(q)) {
        q->front = 0; // Initialize front if queue is empty
    }
    q->items[++(q->rear)] = value;
}
// Dequeue operation
int dequeue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue\n");
        exit(EXIT_FAILURE);
    }
    int dequeued = q->items[(q->front)++];
    if (q->front > q->rear) { // Reset front and rear if queue becomes
empty
        q->front = -1;
        q->rear = -1;
    }
    return dequeued;
}
// Function to display the elements in the queue
void displayQueue(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements are: ");
    for (int i = q->front; i <= q->rear; ++i) {
        printf("%d ", q->items[i]);
    }
    printf("\n");
}
// Main function
int main() {
    Queue q;
    int n,ele;
    printf("enter no of elements you want to insert:");
```

```
 scanf("%d",&n);
 initializeQueue(&q);
 for(int k=0;k<n;k++){
 printf("enter the ele:");
 scanf("%d",&ele);
 enqueue(&q,ele);
}
 displayQueue(&q);
 int dequeued = dequeue(&q);
 printf("Dequeued element: %d\n", dequeued);
 displayQueue(&q);
 return 0;
}
```

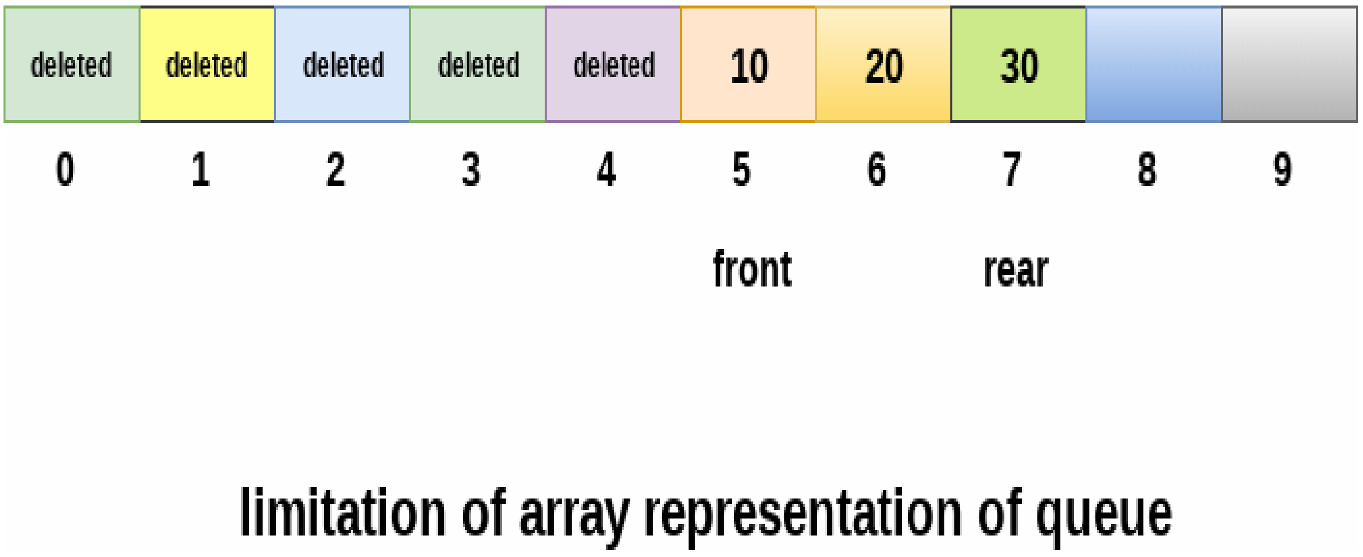
Output:



```
enter no of elements you want to insert:4
enter the ele:6
enter the ele:3
enter the ele:9
enter the ele:1
Queue elements are: 6 3 9 1
Dequeued element: 6
Queue elements are: 3 9 1
```

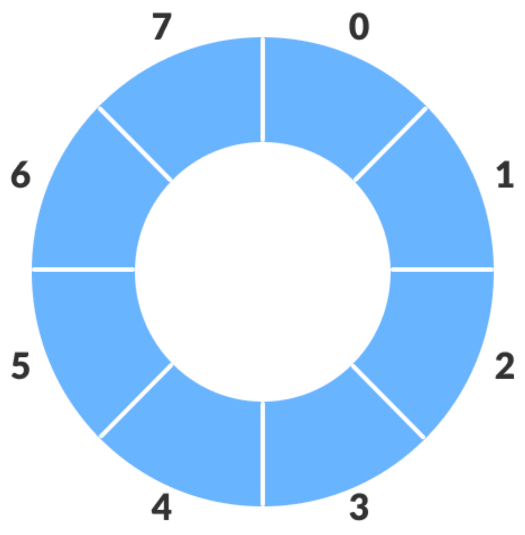
Drawback of array implementation:

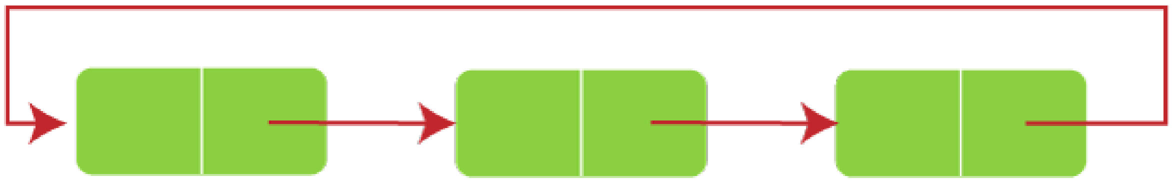
- Memory wastage



Circular Queues:

A circular queue overcomes the limitation of the array queue by treating the array as circular. This means the position after the last element connects to the first element of the array.



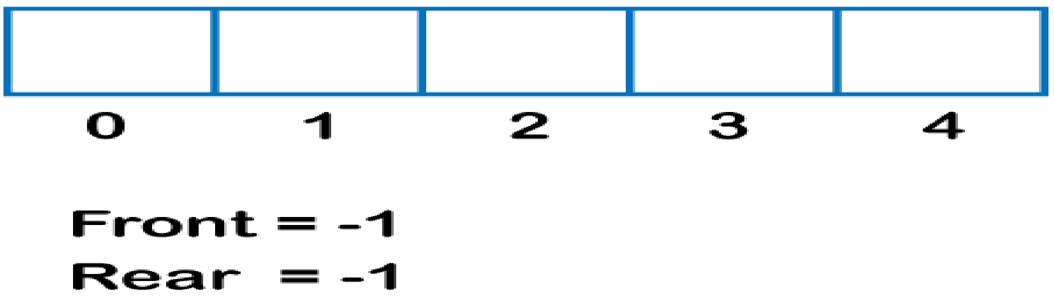


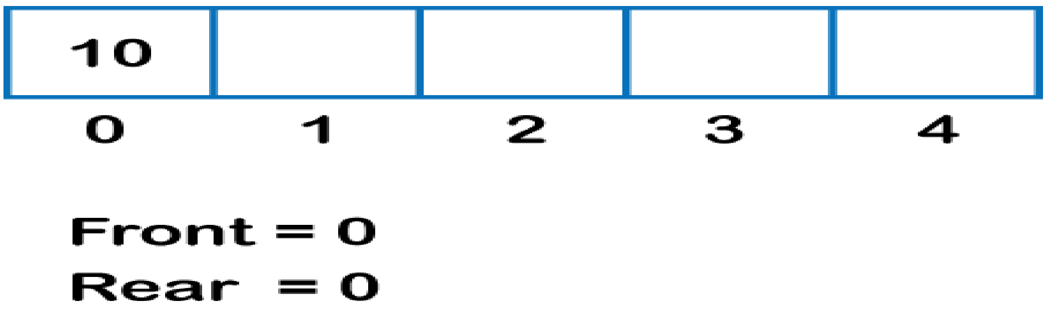
Circular Queue

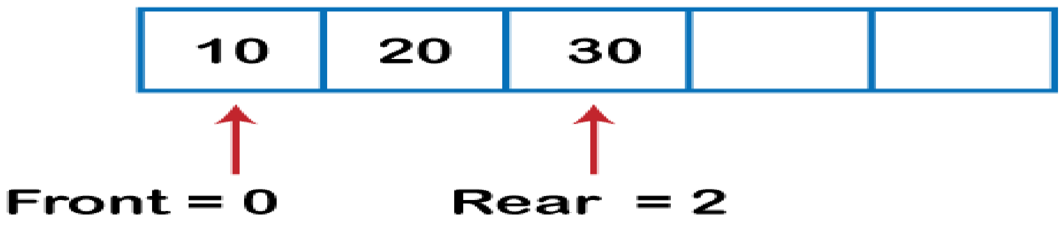
Circular Queue Operations:

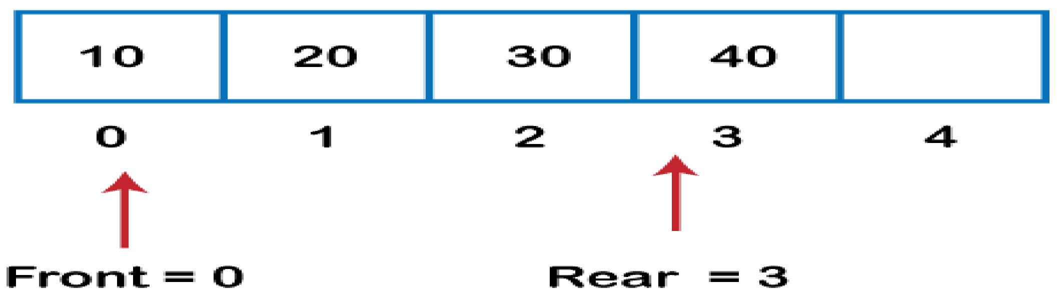
Enqueue: Similar to the regular queue but with circular adjustments.

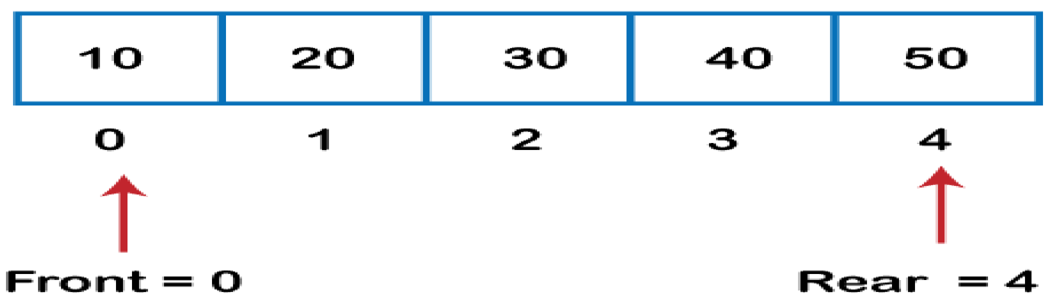
Dequeue: Similar to the regular queue but with circular adjustments.

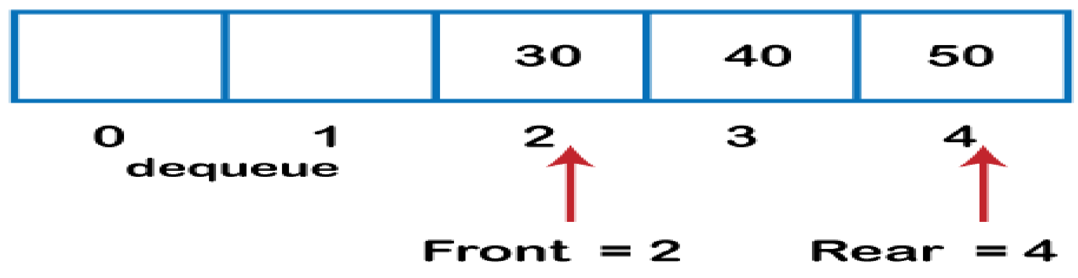


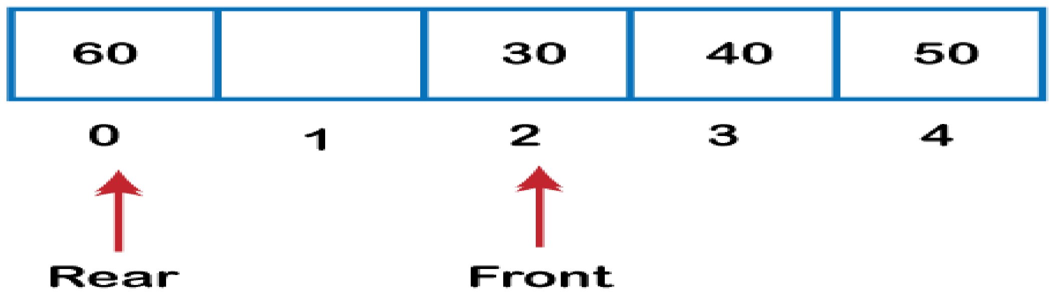


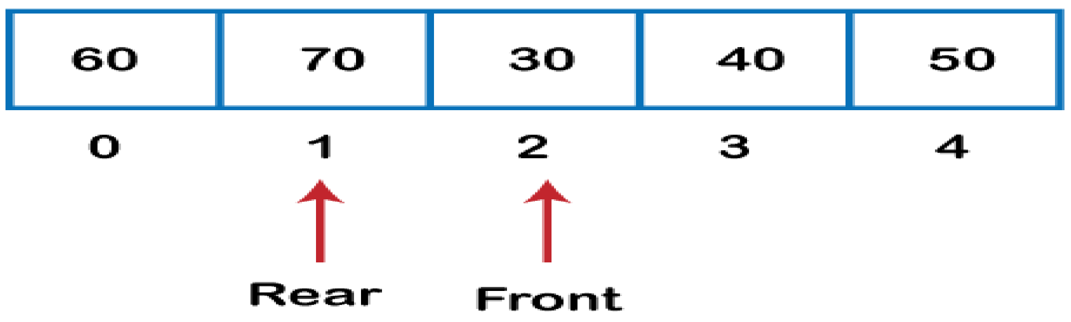



10
20
30
40
0
1
2
3
4
Front = 0
Rear = 3


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4


30
40
50
0
1
dequeue
2
3
4
Front = 2
Rear = 4


60
30
40
50
0
1
2
3
4
Rear
Front



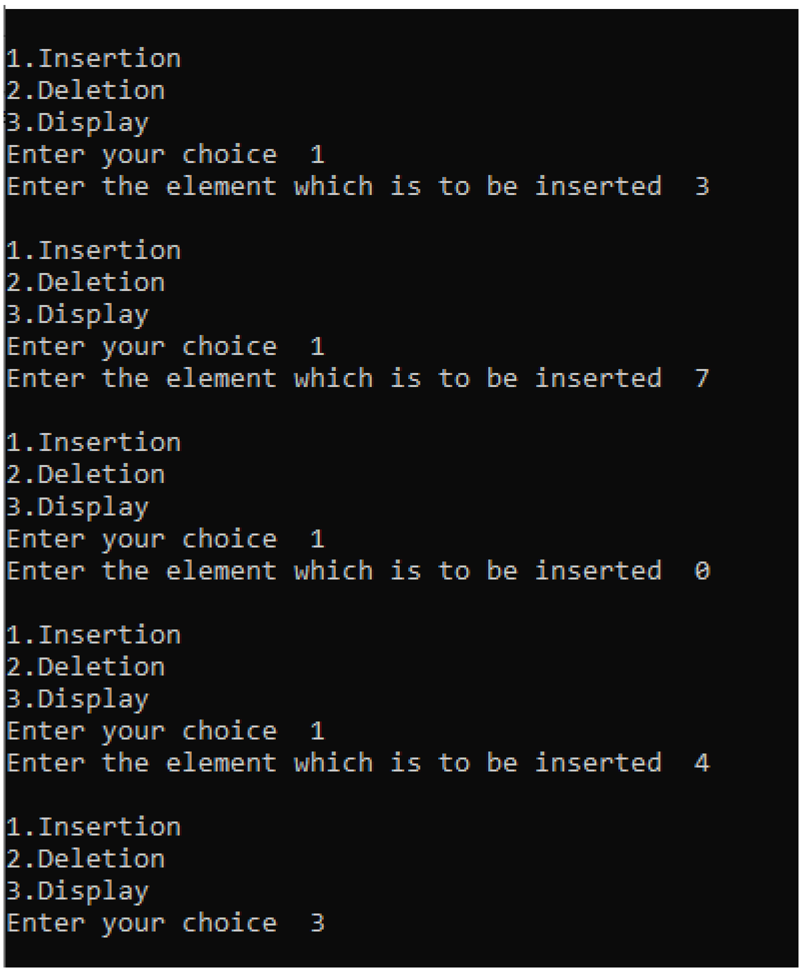
__Example: (Implementation of circular queue using Array)__

```
#include <stdio.h>
# define max 10
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
   {
      printf("Queue is overflow..");
   }
```

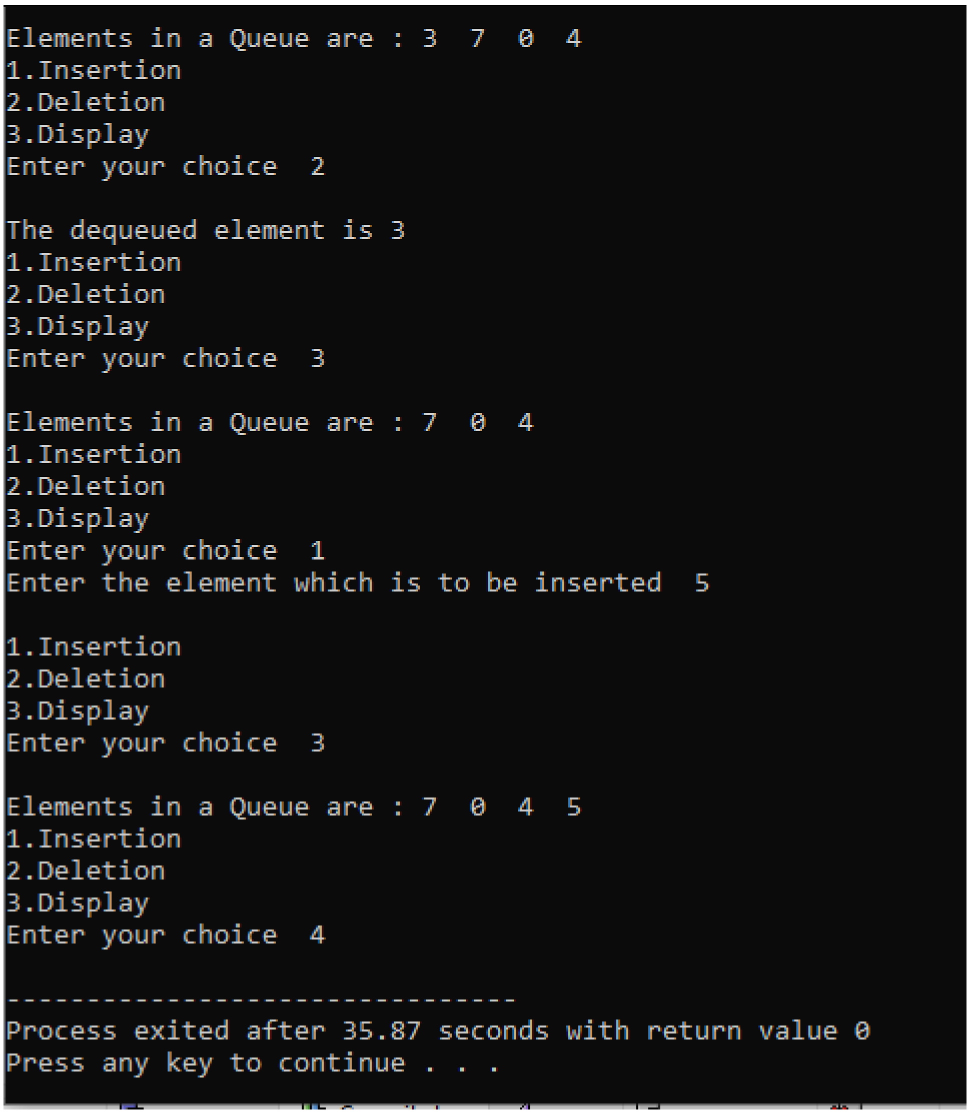
```
    else
    {
        rear=(rear+1)%max;       // rear is incremented
        queue[rear]=element;     // assigning a value to the queue at the
rear position.
    }
}
// function to delete the element from the queue
int dequeue()
{
    if((front==-1) && (rear==-1))  // condition to check queue is empty
    {
        printf("\nQueue is underflow..");
    }
 else if(front==rear)
{
   printf("\nThe dequeued element is %d  ", queue[front]);
   front=-1;
   rear=-1;
}
else
{
    printf("\nThe dequeued element is %d  ", queue[front]);
   front=(front+1)%max;
}
}
// function to display the elements of a queue
void display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("\nElements in a Queue are : ");
        while(i<=rear)
        {
            printf("%d  ", queue[i]);
            i=(i+1)%max;
        }
    }
}
int main()
{
    int choice=1,x;   // variables declaration
    while(choice<4 && choice!=0)   // while loop
```

```
   {
   printf("\n1.Insertion \n2.Deletion \n3.Display ");
   printf("\nEnter your choice  ");
   scanf("%d", &choice);
   switch(choice)
   {
      case 1:
      printf("Enter the element which is to be inserted  ");
      scanf("%d", &x);
      enqueue(x);
      break;
      case 2:
      dequeue();
      break;
      case 3:
      display();
   }
}
   return 0;
}
```

Output:



```

1.Insertion
2.Deletion
3.Display
Enter your choice  1
Enter the element which is to be inserted  3

1.Insertion
2.Deletion
3.Display
Enter your choice  1
Enter the element which is to be inserted  7

1.Insertion
2.Deletion
3.Display
Enter your choice  1
Enter the element which is to be inserted  0

1.Insertion
2.Deletion
3.Display
Enter your choice  1
Enter the element which is to be inserted  4

1.Insertion
2.Deletion
3.Display
Enter your choice  3
```



```
Elements in a Queue are : 3   7   0   4
1.Insertion
2.Deletion
3.Display
Enter your choice   2

The dequeued element is 3
1.Insertion
2.Deletion
3.Display
Enter your choice   3

Elements in a Queue are : 7   0   4
1.Insertion
2.Deletion
3.Display
Enter your choice   1
Enter the element which is to be inserted   5

1.Insertion
2.Deletion
3.Display
Enter your choice   3

Elements in a Queue are : 7   0   4   5
1.Insertion
2.Deletion
3.Display
Enter your choice   4

--------------------------------
Process exited after 35.87 seconds with return value 0
Press any key to continue . . .
```

Applications of Circular Queue:

The circular Queue can be used in the following scenarios:

Memory management: The circular queue provides memory management.

CPU Scheduling: The operating system also uses the circular queue to insert the processes and then execute them.

Traffic system: In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Circular Queues using Dynamic Arrays

- Dynamic arrays allow resizing, which can be beneficial for queues that need to grow or shrink dynamically.
- It involves managing the array size dynamically as elements are added or removed from the queue.

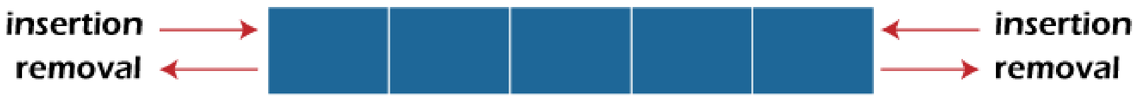
Program:

```
#include <stdio.h>

#include <stdlib.h>

typedef struct {

   int *arr;        // Array to store elements of the queue

   int front;       // Index of the front element

   int rear;        // Index of the rear element

   int capacity;    // Maximum capacity of the queue

   int size;        // Current number of elements in the queue

} CQueue;

// Function to create a circular queue with initial capacity

CQueue* createCQueue(int capacity) {

   CQueue *q = (CQueue*) malloc(sizeof(CQueue));

   q->capacity = capacity;

   q->arr = (int*) malloc(q->capacity * sizeof(int));

   q->front = -1;   // Initialize front index

   q->rear = -1;    // Initialize rear index

   q->size = 0;     // Initialize size of the queue
```

```
    return q;
}
// Function to check if the queue is empty
int isEmpty(CQueue *q) {
    return q->size == 0;
}
// Function to check if the queue is full
int isFull(CQueue *q) {
    return q->size == q->capacity;
}
// Function to enqueue an element into the circular queue
void enqueue(CQueue *q, int item) {
    if (isFull(q)) {
        printf("Queue is full. Cannot enqueue %d\n", item);
        return;
    }
    if (isEmpty(q)) {
        q->front = 0;   // If queue is empty, set front to 0
        q->rear = 0;    // Set rear to 0 when enqueueing the first element
    } else {
        q->rear = (q->rear + 1) % q->capacity;  // Circular increment of
rear index
    }
    q->arr[q->rear] = item;   // Enqueue the item at the rear index
    q->size++;   // Increase the size of the queue
}
// Function to dequeue an element from the circular queue
int dequeue(CQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    }
    int dequeuedItem = q->arr[q->front];   // Get the element at the
front index
    if (q->front == q->rear) {
        q->front = -1;   // Reset front and rear indices when queue
becomes empty
        q->rear = -1;
    } else {
        q->front = (q->front + 1) % q->capacity;   // Circular increment of
front index
    }
    q->size--;   // Decrease the size of the queue
    return dequeuedItem;   // Return the dequeued element
}
// Function to display the elements of the circular queue
void display(CQueue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Elements of the queue are: ");
    int i;
    for (i = q->front; i != q->rear; i = (i + 1) % q->capacity) {
        printf("%d ", q->arr[i]);   // Print elements from front to rear
    }
    printf("%d\n", q->arr[i]);   // Print the last element
}
// Function to free memory allocated to the circular queue
void deleteCQueue(CQueue *q) {
    free(q->arr);   // Free the array storing elements
    free(q);   // Free the queue structure
}
// Main function to test the circular queue implementation
int main() {
    int capacity;
    printf("Enter the capacity of the circular queue: ");
    scanf("%d", &capacity);   // Input the capacity of the circular queue
    CQueue *q = createCQueue(capacity);   // Create circular queue
with specified capacity
    int choice, item;
    do {
        // Display menu options
        printf("\n1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);   // Input choice from the user
        switch (choice) {
            case 1:
                printf("Enter element to enqueue: ");
                scanf("%d", &item);   // Input element to enqueue
                enqueue(q, item);   // Enqueue the input element
                break;
            case 2:
                item = dequeue(q);   // Dequeue an element from the
queue
                if (item != -1) {
                    printf("Dequeued element: %d\n", item);   // Display the
dequeued element
                }
                break;
            case 3:
                display(q);   // Display all elements in the queue
                break;
            case 4:
                printf("Exiting program.\n");   // Exit the program
                break;
            default:
                printf("Invalid choice. Please enter a valid choice.\n");   //
Handle invalid choice
        }
    } while (choice != 4);   // Repeat until user chooses to exit
    deleteCQueue(q);   // Free memory allocated to the circular queue
    return 0;
}
```

Deque (Double-ended Queue)

- A deque is a data structure that allows insertion and deletion at both ends (front and rear).
- It combines the features of both stacks (Last In First Out) and queues (First In First Out).



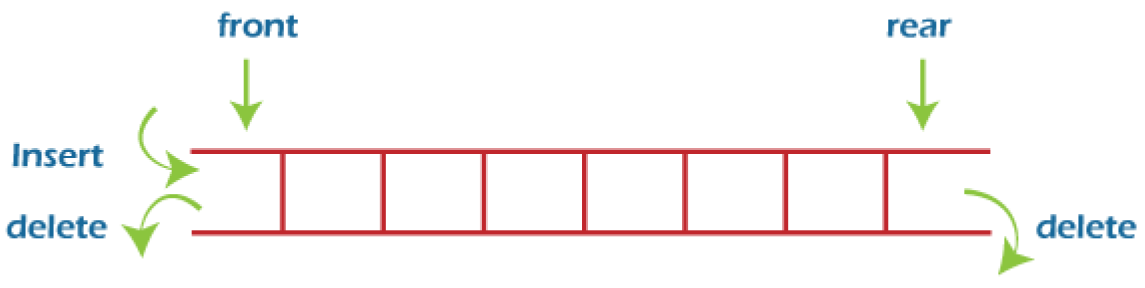
Representation of deque

There are two types of deque

1. Input restricted queue
2. Output restricted queue

<u>INPUT RESTICTED QUEUE:</u>

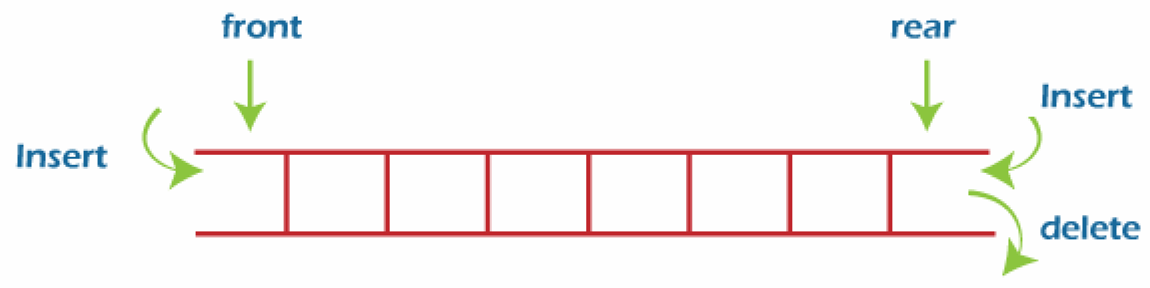
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

OUTPUT RESTICTED QUEUE:

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Deque Operations:

insertFront: Insert an element at the front end.

insertRear: Insert an element at the rear end.

deleteFront: Delete an element from the front end.

deleteRear: Delete an element from the rear end.

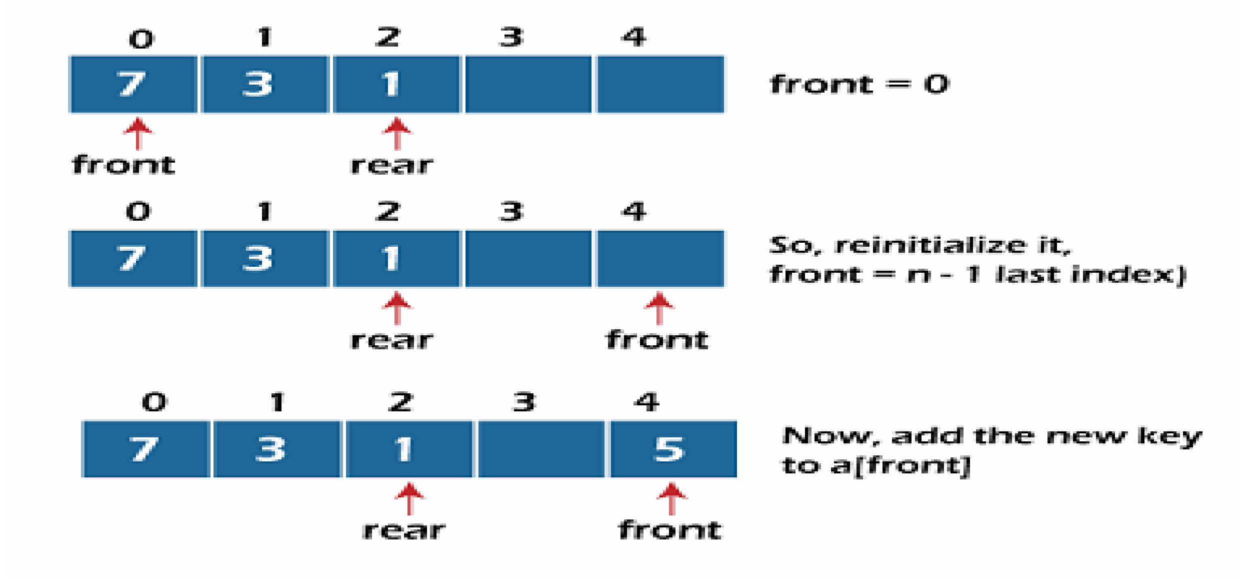
getFront: Get the front element.

getRear: Get the rear element.

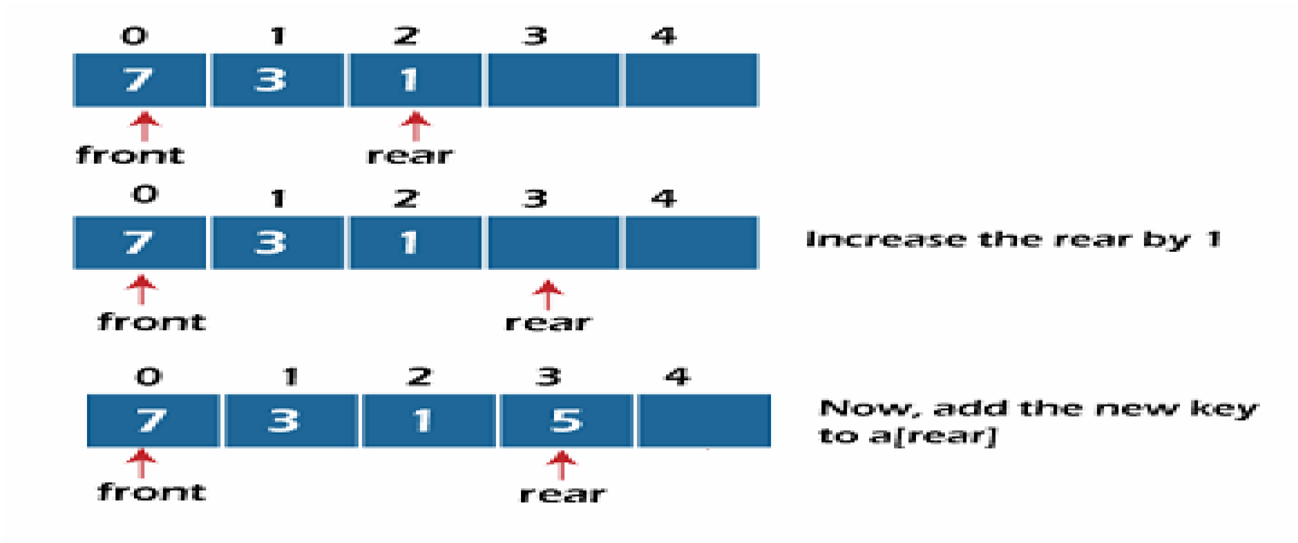
isFull: Check if the deque is full.

isEmpty: Check if the deque is empty.

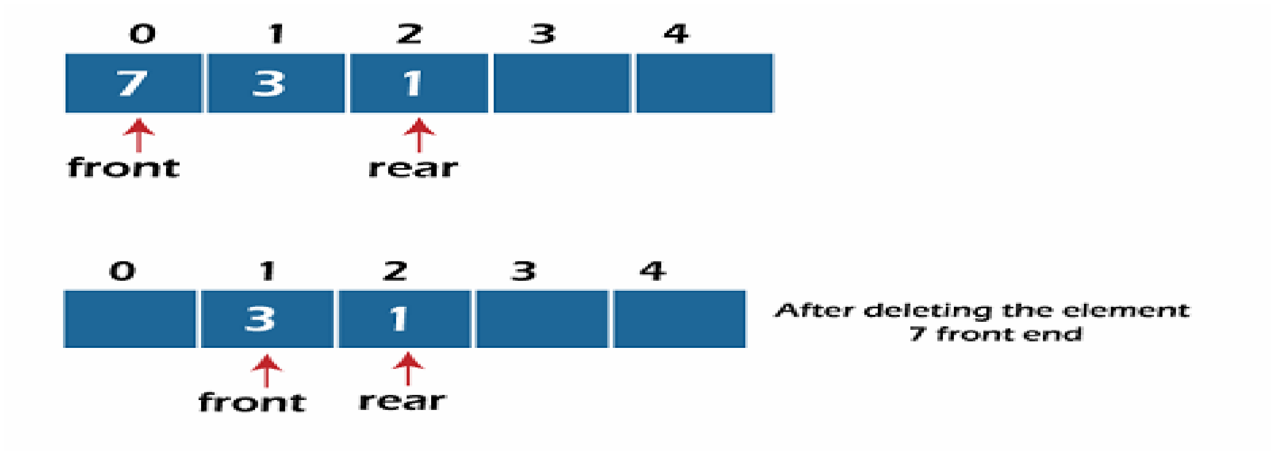
Insertion at the front end:



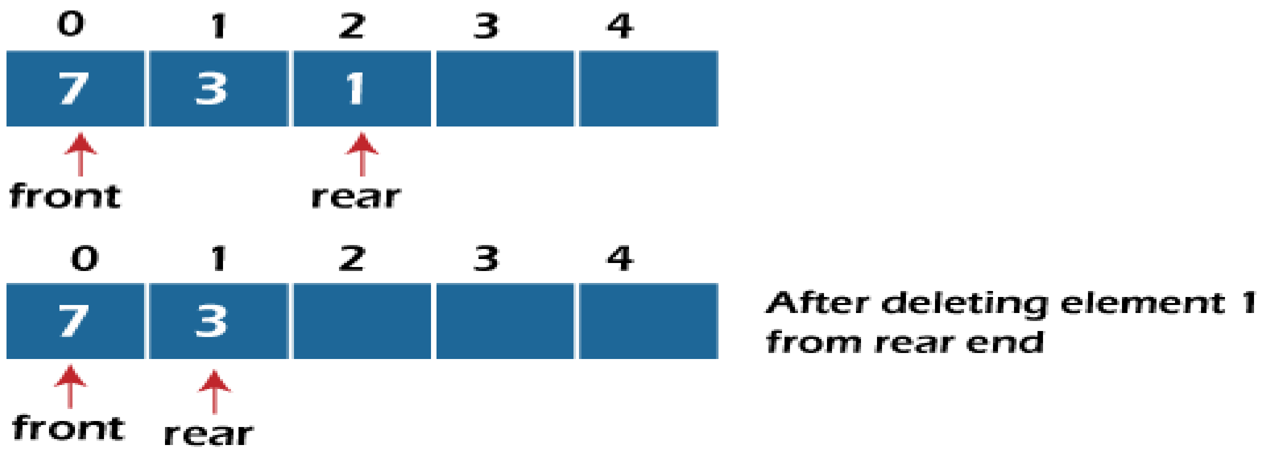
Insertion at the rear end:



Deletion at the front end



Deletion at the rear end



Applications of deque:

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.
- Job Scheduling
- History Management
- Optimized Data Structures

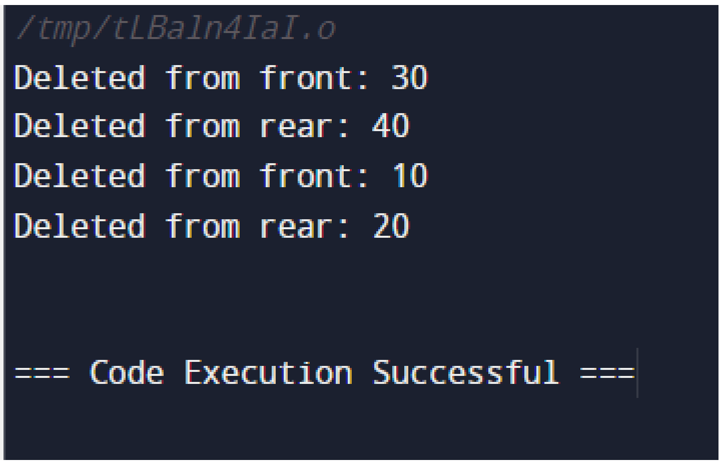
Example: (implementation of a deque using a circular array)

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
typedef struct {
   int items[MAX];  // Array to store elements of the deque
   int front, rear; // Indices for front and rear elements of the deque
} Deque;
// Function to initialize the deque
void initializeDeque(Deque* dq) {
    dq->front = -1;  // Initialize front index to -1 (indicating empty
deque)
    dq->rear = 0;    // Initialize rear index to 0 (indicating empty deque)
}
// Function to check if the deque is full
int isFullDeque(Deque* dq) {
    return ((dq->front == 0 && dq->rear == MAX - 1) ||  // Check if
deque is circularly full
            dq->front == dq->rear + 1);
}
// Function to check if the deque is empty
int isEmptyDeque(Deque* dq) {
    return dq->front == -1;  // Check if front index is -1 (indicating
empty deque)
}
// Function to insert an element at the front of the deque
void insertFront(Deque* dq, int value) {
    if (isFullDeque(dq)) {
        printf("Deque is full!\n");  // Print message if deque is full
        return;
    }
    if (isEmptyDeque(dq)) {
        dq->front = dq->rear = 0;  // Initialize deque if it's empty
    } else if (dq->front == 0) {
        dq->front = MAX - 1;  // Wrap around to the end of array
(circular)
    } else {
        dq->front--;  // Move front index backward
    }
    dq->items[dq->front] = value;  // Insert value at the front of deque
}
// Function to insert an element at the rear of the deque
void insertRear(Deque* dq, int value) {
    if (isFullDeque(dq)) {
        printf("Deque is full!\n");  // Print message if deque is full
        return;
    }
    if (isEmptyDeque(dq)) {
        dq->front = dq->rear = 0;  // Initialize deque if it's empty
    } else if (dq->rear == MAX - 1) {
        dq->rear = 0;  // Wrap around to the beginning of array (circular)
    } else {
        dq->rear++;  // Move rear index forward
    }
    dq->items[dq->rear] = value;  // Insert value at the rear of deque
}
// Function to delete an element from the front of the deque
int deleteFront(Deque* dq) {
    if (isEmptyDeque(dq)) {
        printf("Deque is empty!\n");  // Print message if deque is empty
        return -1;
    }
    int item = dq->items[dq->front];  // Retrieve item at the front of
deque
    if (dq->front == dq->rear) {
        dq->front = dq->rear = -1;  // Reset deque indices if it's the last
element
    } else if (dq->front == MAX - 1) {
        dq->front = 0;  // Wrap around to the beginning of array
(circular)
    } else {
        dq->front++;  // Move front index forward
    }
    return item;  // Return deleted item
}
// Function to delete an element from the rear of the deque
int deleteRear(Deque* dq) {
    if (isEmptyDeque(dq)) {
        printf("Deque is empty!\n");  // Print message if deque is empty
        return -1;
    }
    int item = dq->items[dq->rear];  // Retrieve item at the rear of
deque
```

```
    if (dq->front == dq->rear) {
        dq->front = dq->rear = -1;  // Reset deque indices if it's the last
element
    } else if (dq->rear == 0) {
        dq->rear = MAX - 1;  // Wrap around to the end of array
(circular)
    } else {
        dq->rear--;  // Move rear index backward
    }
    return item;  // Return deleted item
}
// Main function to test the deque implementation
int main() {
    Deque dq;  // Declare a deque instance
    initializeDeque(&dq);  // Initialize the deque
    // Test operations
    insertFront(&dq, 10);
    insertRear(&dq, 20);
    insertFront(&dq, 30);
    printf("Deleted from front: %d\n", deleteFront(&dq));
    printf("Deleted from rear: %d\n", deleteRear(&dq));
    printf("Deleted from front: %d\n", deleteFront(&dq));
    printf("Deleted from rear: %d\n", deleteRear(&dq));
    return 0;  // Return from main
```

}

OUTPUT:



```
/tmp/tLBaln4IaI.o
Deleted from front: 30
Deleted from rear: 40
Deleted from front: 10
Deleted from rear: 20


=== Code Execution Successful ===
```

Initialization:

initializeDeque initializes the front and rear pointers to -1, indicating an empty deque.

Check Full and Empty:

isFullDeque checks if the deque is full by comparing the positions of front and rear.

isEmptyDeque checks if the deque is empty by checking if front is -1.

Insert Front:

insertFront inserts an element at the front. If the deque is full, it prints an error message. If the deque is empty, it sets both front and rear to 0. If the front is at 0, it wraps around to MAX - 1. Otherwise, it decrements in front.

Insert Rear:

insertRear inserts an element at the rear. The logic is similar to insertFront, but it updates the rear pointer instead.

Delete Front:

deleteFront removes an element from the front. If the deque becomes empty, it resets both front and rear to -1. If the front reaches MAX - 1, it wraps around to 0. Otherwise, it increments front.

Delete Rear:

deleteRear removes an element from the rear. The logic is similar to deleteFront, but it updates the rear pointer instead.

Get Front and Rear:

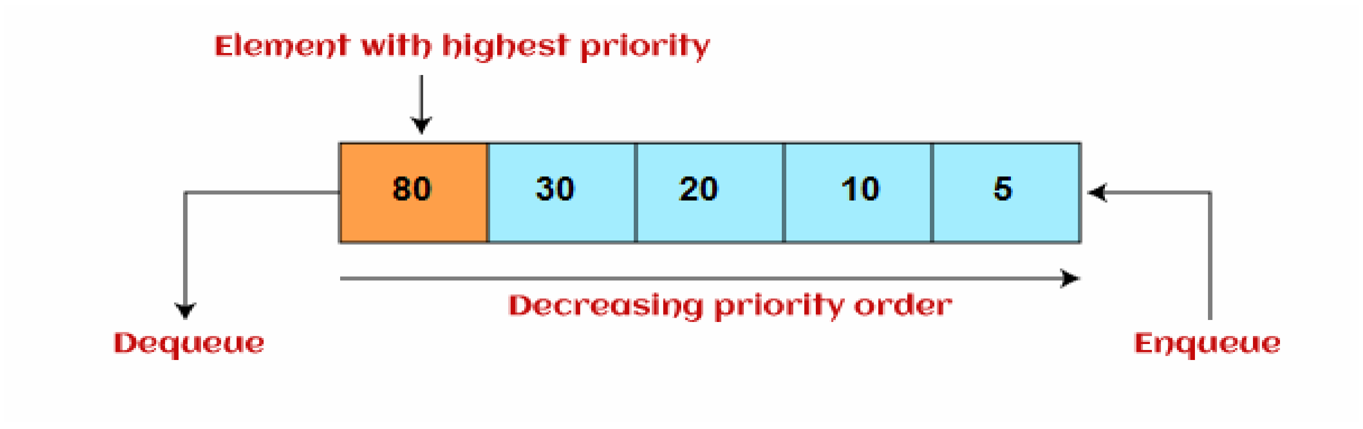
getFront and getRear return the elements at the front and rear of the deque, respectively.

Main Function:

The main function demonstrates the usage of the deque by inserting and deleting elements from both ends and printing the front and rear elements.

Priority Queues

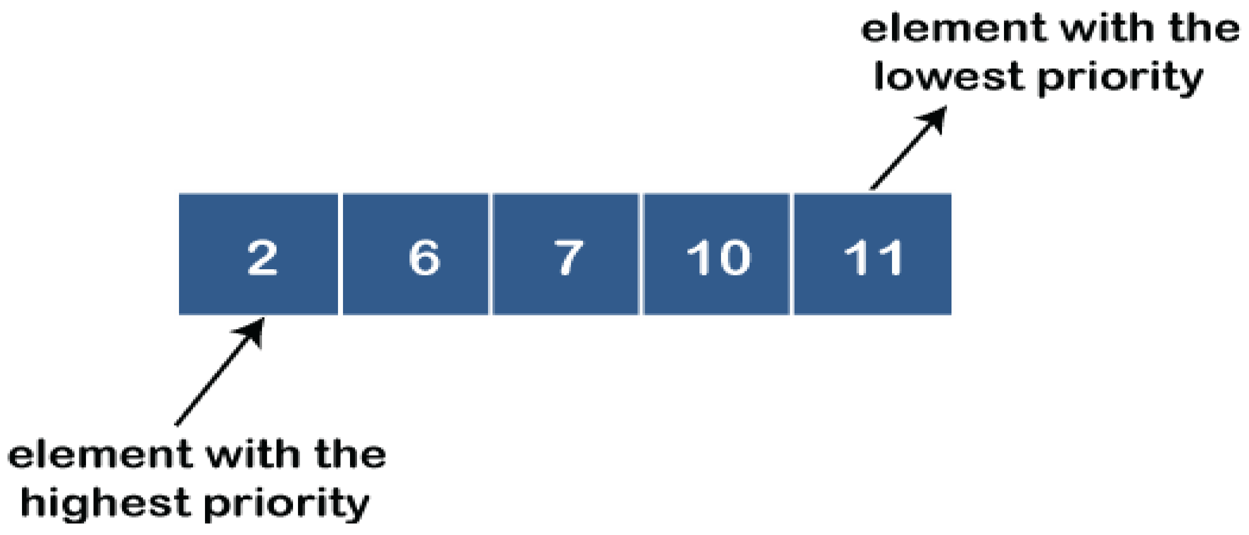
- A priority queue is a special type of queue where each element is associated with a priority, and elements are dequeued based on their priority.
- The priority queue stores elements in a sorted manner, ensuring that the highest priority element is always at the end of the array for easy removal.
- A priority queue is a type of queue that arranges elements based on their priority values.
- Elements with higher priority values are typically retrieved before elements with lower priority values.
- In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value.



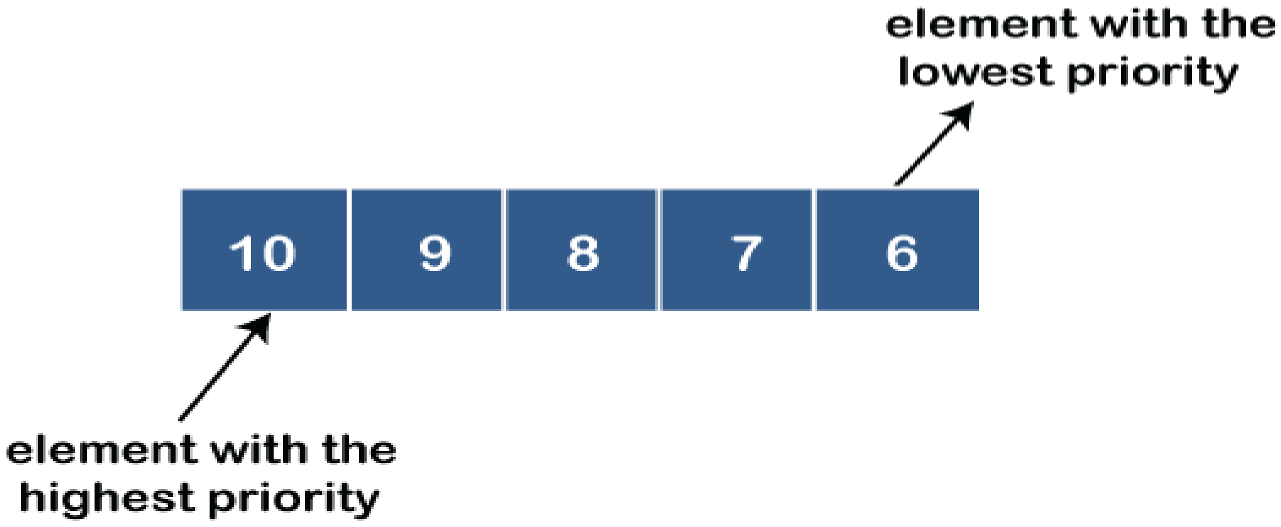
Types of Priority Queue:

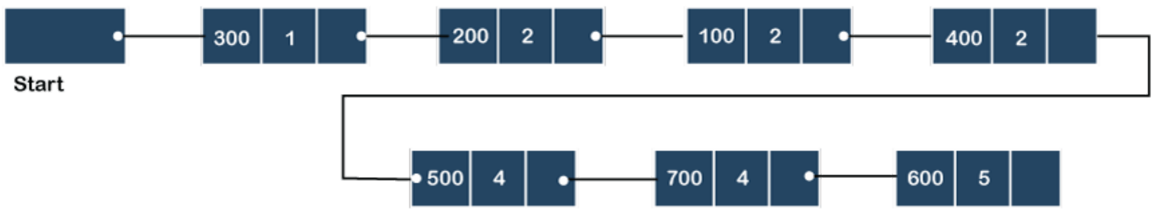
- Ascending order priority queue
- Descending order priority queue

Ascending order priority queue:


element with the lowest priority
2
6
7
10
11
element with the highest priority

Descending order priority queue:


element with the lowest priority
10
9
8
7
6
element with the highest priority



<u>*Priority Queue Operations:*</u>

Insert: Add an element to the priority queue.

Extract Max: Remove and return the element with the highest priority.

Peek: Return the element with the highest priority without removing it. This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

IsEmpty: Check if the priority queue is empty.

IsFull: Check if the priority queue is full.

<u>*Types of Priority Queue:*</u>

1) Ascending Order Priority Queue

2) Descending order Priority Queue

Problems with Priority Queues:

Implementation Complexity: Managing the priorities can be complex.

Efficiency: Insertion and deletion operations can be less efficient compared to normal queues.

Example: (implementation of a priority queue using an array)

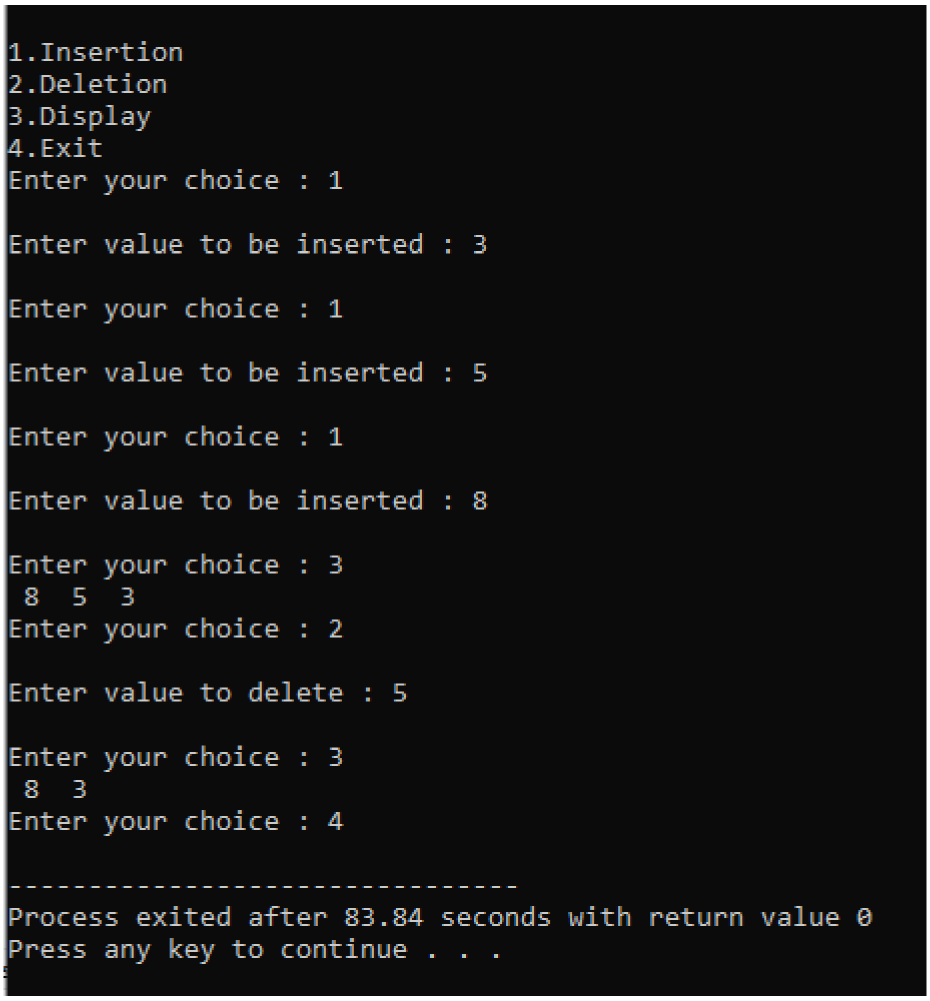
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();
int pri_que[MAX];
int front, rear;
void main()
{
   int n, ch;
    printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
   create();
   while (1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
```

```
        switch (ch)
        {
        case 1:
            printf("\nEnter value to be inserted : ");
            scanf("%d",&n);
            insert_by_priority(n);
            break;
        case 2:
            printf("\nEnter value to delete : ");
            scanf("%d",&n);
            delete_by_priority(n);
            break;
        case 3:
            display_pqueue();
            break;
        case 4:
            exit(0);
        default:
            printf("\nChoice is incorrect, Enter a correct choice");
        }
    }
}
/* Function to create an empty priority queue */
void create()
```

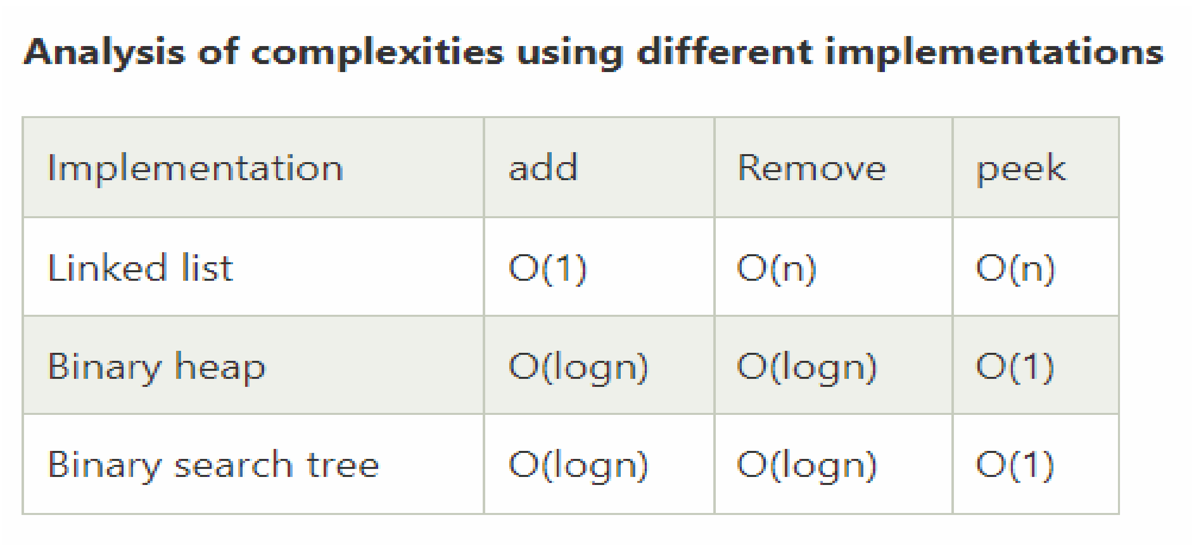
```
{
   front = rear = -1;
}
/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
   if (rear >= MAX - 1)
   {
      printf("\nQueue overflow no more elements can be inserted");
      return;
   }
   if ((front == -1) && (rear == -1))
   {
      front++;
      rear++;
      pri_que[rear] = data;
      return;
   }
   else
      check(data);
   rear++;
}
/* Function to check priority and place element */
void check(int data)
```

```
{
    int i,j;
    for (i = 0; i <= rear; i++)
    {
        if (data >= pri_que[i])
        {
            for (j = rear + 1; j > i; j--)
            {
                pri_que[j] = pri_que[j - 1];
            }
            pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}
/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;
    if ((front==-1) && (rear==-1))
    {
        printf("\nQueue is empty no elements to delete");
        return;
    }
    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }
        pri_que[i] = -99;
        rear--;
        if (rear == -1)
            front = -1;
        return;
        }
    }
    printf("\n%d not found in queue to delete", data);
}
/* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
        return;
    }
    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }
    front = 0;
}
```

Output:



```
1.Insertion
2.Deletion
3.Display
4.Exit
Enter your choice : 1

Enter value to be inserted : 3

Enter your choice : 1

Enter value to be inserted : 5

Enter your choice : 1

Enter value to be inserted : 8

Enter your choice : 3
 8  5  3
Enter your choice : 2

Enter value to delete : 5

Enter your choice : 3
 8  3
Enter your choice : 4

--------------------------------
Process exited after 83.84 seconds with return value 0
Press any key to continue . . .
```



Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

Applications of Priority queue:

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.