**Day-17 :**

**Stack :**

* LIFO (Last In First Out)
* It goes from bottom to top
* Size/Limit/ of stack 🡪 fixed
* Top will move from bottom to top
* Insert/push --- adding element by checking whether the stack is full or not
* Remove/pop/pull ---deleting element by checking whether the stack is empty or not

**QUEUE:**

* FIFO
* - Front == rear 🡪 empty queue (rear is increasing)
* - Rear == MAX 🡪 queue full
* - Front increases till MAX while deleting
* - When front reaches rear reset front and rear i.e f =0, r = 0 during removing element
* - Push here is enqueue and pop here is dequeue

**Basic Operations:**

* push () to insert an element into the stack
* pop () to remove an element from the stack
* top () Returns the top element of the stack.
* isEmpty () returns true if stack is empty else false.
* isFull () returns true if the stack is full else false.

……………………………………………………………………….

**static stack**. This is because the stack size is fixed at compile time with the #define MAX 100 directive, and the stack is implemented using a fixed-size array int stack[MAX].

In a **dynamic stack**, the stack size can grow or shrink at runtime, typically using dynamic memory allocation functions like malloc and free in C.

……………………………………………………………………….

(static memory allocation)

#include <stdio.h>

#include<stdlib.h>

#include<stack

#define MAX 5

int bottom=-1;

int top=0;

int push(int);

int pop();

int stackCount[MAX];

int push(int v)

{

if(top==MAX-1)

{

printf("\nStack/container is Full\n");

return 0;

}

stackCount[top]=v;

top++;

}

int poo()

{

int v;

if(top==bottom)

{

printf("\nStack/Container is empty\n");

return 0;

}

v = stackCount[top];

v--;

return v;

}

void dispStack()

{

int iter;

if(top==bottom)

{

printf("\nStack/Container is empty\n");

return;

}

printf("\nSatck Elements are\n");

for(iter=top;iter>bottom;iter--)

{

printf("\n%d",stackCount(iter));

}

printf("\n\n");

}

………………………………………………………………………….

**Queue Data Structure :**

* used for storing and managing data in a specific order.
* It follows the principle of "First in, First out" (FIFO)

A diagram of a type of queue

Description automatically generated

A diagram of a queue data structure

Description automatically generated

|  |
| --- |
| If(Front = = rear)  {  Empty;  } |

[Basic Operations in Queue Data Structure:](https://www.geeksforgeeks.org/basic-operations-for-queue-in-data-structure)

Some of the basic operations for Queue in Data Structure are:

1. **Enqueue:** Adds (or stores) an element to the end of the queue.
2. **Dequeue:** Removal of elements from the queue.
3. **Peek or front:** Acquires the data element available at the front node of the queue without deleting it.
4. **rear:** This operation returns the element at the rear end without removing it.
5. **isFull**: Validates if the queue is full.
6. **isEmpty**: Checks if the queue is empty.

Example :

#include <stdio.h>

#define MAX 5

int rear=0;

int front=0;

int q[MAX];

void enqueue(int);

void dequeue();

void dispQueue();

int main()

{

enqueue(10);

enqueue(20);

enqueue(30);

enqueue(40);

enqueue(50);

enqueue(60);

dispQueue();

dequeue();

dequeue();

dequeue();

dequeue();

dequeue();

dequeue();

dispQueue();

printf("\n\n");

return 0;

}

void dispQueue()

{

int i;

if(front == rear)

{

printf("\nEmpty Queue\n");

return;

}

printf("\nQueue is\n");

for(i=front;i<rear;i++)

printf("%d ",q[i]);

printf("\n\n");

}

void enqueue(int val)

{

if(rear == MAX)

{

printf("\nQueue is full\n");

return;

}

q[rear] = val;

rear++;

}

void dequeue()

{

if(front==rear)

{

front = 0; //reset the queue

rear = 0;

printf("\nEmpty Queue\n");

return;

}

printf("\n%d is dequeued from the queue",q[front]);

front++;

}

Queue is full.

Queue is

10 20 30 40 50

10 is dequeued from the queue

20 is dequeued from the queue

30 is dequeued from the queue

40 is dequeued from the queue

50 is dequeued from the queue

Empty Queue

Empty Queue

;

Create the employee record dynamically display it using stack operation