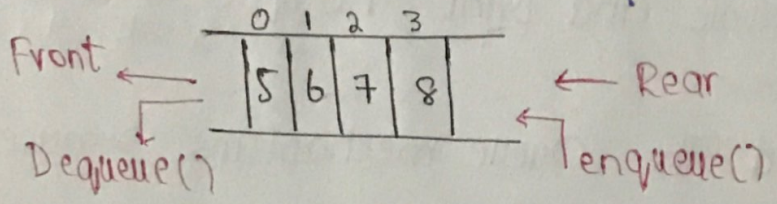
Queue:

- It is an linear data Structure
- It is an Abstract data type (ADT).
- First In First Out (FIFO) mean first value in the Queue delete first from Queue.
- Insertion happens from one end and that is known as rear end. / tail.
- Deletion happens from second / other end and that is known as front end. / head.

- Insertion operation in Queue is known as enqueue().
- Deletion operation in Queue is known as dequeue().
- Logically, we say Queue has two open ends.

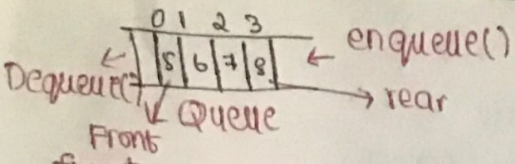


FIFO | LIFO

Operations:

- Enqueue ( ) → For Insertions
- Dequeue ( ) → For Deletions
- Peek ( ) | Front ( ) → For Front value
- Two more operation to check size and space but you can sum up these two in insertion and deletion.
  - IsFull()
  - IsEmpty()

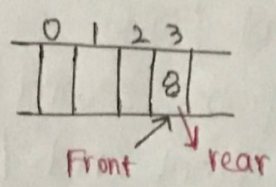
- One drawback of queue is when the front and rear point to the same elements and suppose this is the last one and now you insert more value it show you the overflow but previous all space are empty. This is the drawback of Queue but solution of this Queue is Circular Queue.



front = rear = -1 → means queue is empty

- Dequeue()
- Dequeue()
- Dequeue()

Note: When u call 3 times dequeue() operation and for 4 time front and rear point to the same, then u insert more it show u overflow but First 3 spaces are empty because delete the values.



size = 4

Applications:

- When more than one PCs are connected to the same printer but printer is already buzy but they saved the PC command in Queue and print the file according to the command.
- Processors hold the Queue mechanisms because processor is the shared resource in OS.

Implementation of Queue using Array:

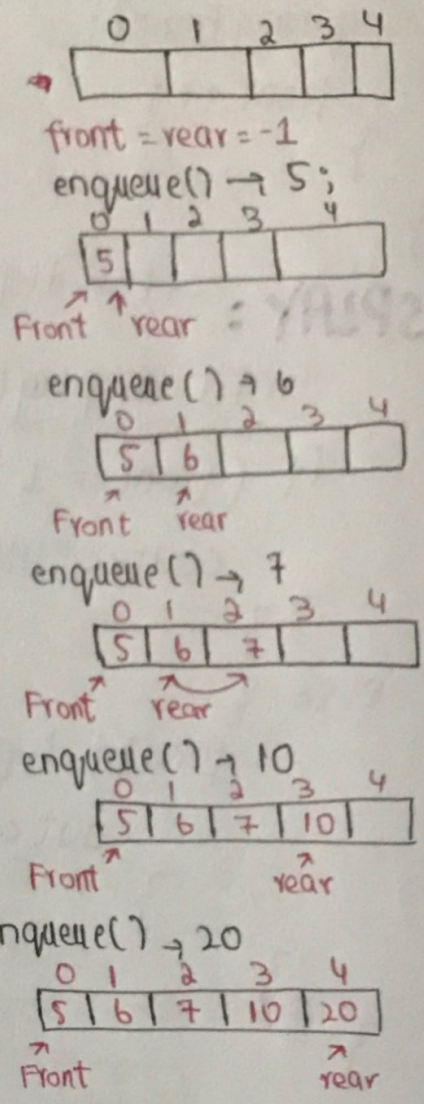
- Static memory location because specify the size.

```
#include<iostream>
#define N 5
using namespace std;
int queue[N];
int front = -1;
int rear = -1;
```

→ These all are written before creating any method.

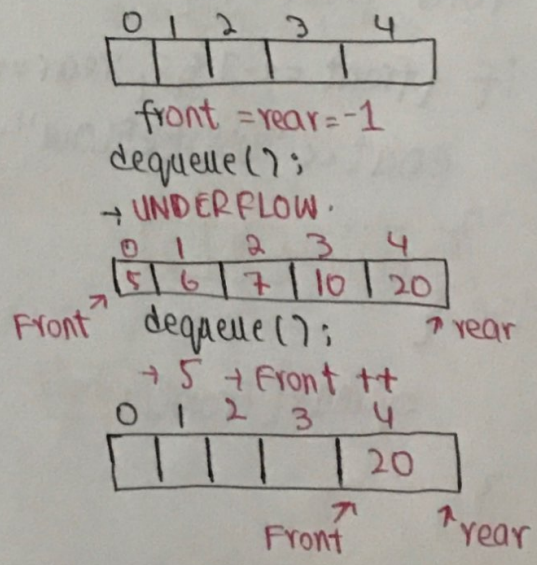
ENQUEUE: → For Insertion

```
void enqueue () {
        int x;
    cout<< "Enter value to insert";
        cin>> x;
  if (rear == N-1) {
        cout<< "OVERFLOW";
      }
  else if (front==-1 && rear ==-1) {
      front = rear =0;
      queue [0] = x;
    }
 else {  rear ++;
      queue [rear];
    }
}
```



DEQUEUE: → FOR Deletion.

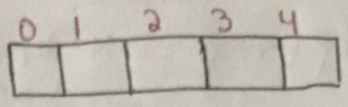
```
void dequeue () {
if (front ==-1 && rear =-1) {
    cout<< "UNDERFLOW";
  }
else if (rear = front) {
  cout<< queue [rear];
  front = rear = 0 -1;
}
```



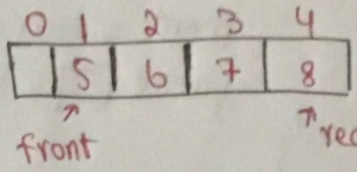
```
else {
    queue[front];
    front++;
  }
}
```

DISPLAY : → For display QUEUE

```
void display(){
if (front==-1 && rear==-1){
    cout<<"UNDERFLOW";
  }
else {
    for (int i=front;i<=rear;i++){
        cout<<queue[i];
      }
  }
}
```



front = rear = -1
display();
→ UNDERFLOW

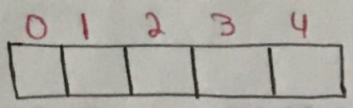


i = front = 1
i <= rear = 4
} → Loop for
cout<<queue[i];

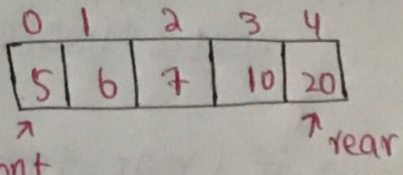
Print:- 5, 6, 7, 8

PEEK : → FOR display Front Value

```
void peek(){
if (front ==-1&& rear ==-1){
    cout<<"UNDERFLOW";
  }
else {
    queue[front];
  }
}
```



front = rear = -1
peek();
→ UNDERFLOW



Print:- 5

Implementation of Queue using Linked List:

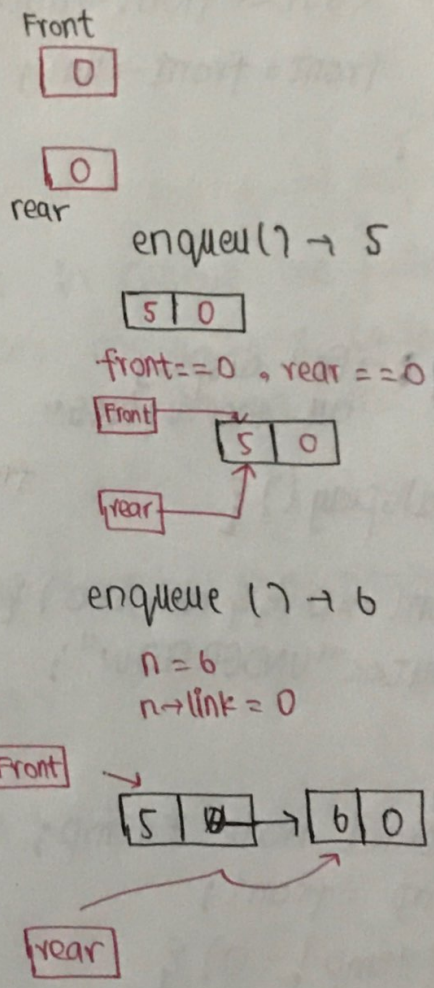
- To solve the static size declaration to store values, we use linked List in Queue.
- Use dynamic Memory Allocation means run time.

```
struct node {
    int data;
    struct node *link;
};
struct node *front = 0;
struct node *rear = 0;
```

→ write this code before writing any function.

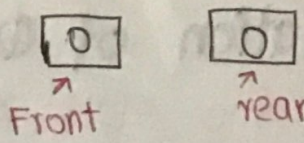
ENQUEUE: → For Insertion.

```
void enqueue () {
    int x;
cout<< "Enter value";
    cin>>x;
struct node *n = new node();
    n->data = x;
    n->link = 0;
if (front == 0 && rear == 0) {
    front = rear = n;
    }
else {
        rear->link = n;
        rear = n;
    }
}
```



DEQUEUE : → For deletion

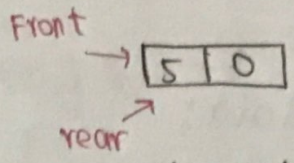
```
void dequeue () {
  if ( front == 0 && rear == 0) {
    cout<<"UNDERFLOW";
  }
 else if ( front == rear) {
    cout << front → data;
    front = rear = 0 ;
  }
 else {  cout << front → data;
    front = front → link;
  }
}
```



dequeue() –

front = rear = 0

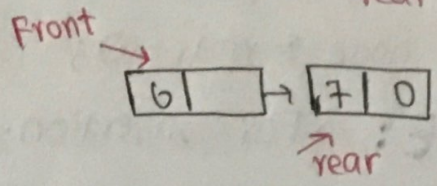
→ UNDERFLOW



dequeue();

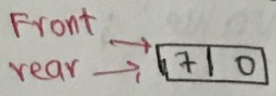
front == rear

Print: 5 → front = 0
rear = 0



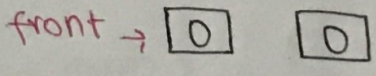
dequeue();

Print:- 6 → Front = front → link



Display : → For display all the values.

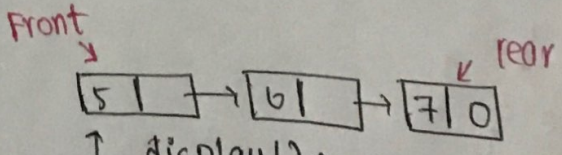
```
void display () {
if (front == 0 && rear == 0) {
   cout<<"UNDERFLOW";
  }
else {
   struct node *temp;
  temp = front;
while (temp != 0) {
   cout<< temp → data;
   temp = temp → link;
  }
 }
}
```



display();

front = rear = 0;

Print: UNDERFLOW



display();

temp = front

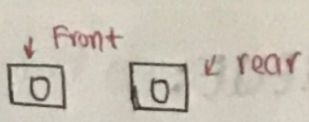
temp

Loop (temp != 0) {

- true, next → print 5
- true, next → print 6
- true, next → print 7

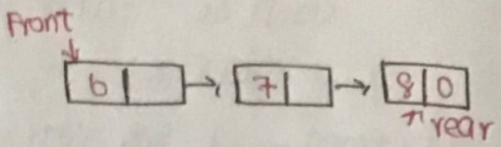
false, Exit

PEEK: → For display Front values.



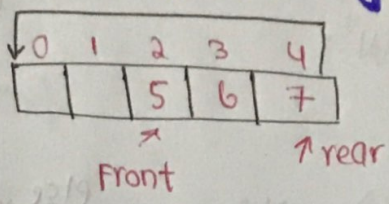

```
void peek() {
    if (front==0 && rear==0) {
        cout<< "UNDERFLOW";
    }
    else {
        cout<< front->data;
    }
}
```

peek();
front=rear=0
Print:- UNDERFLOW



peek();
Print:- 6

QUEUE using Circular Array:



Enqueue();
→ OVERFLOW → (But first spaces are Empty)

To Solve the drawback of arrays in Queue we follow the concept of Circular Array it can move in a cycle to check empty space. If space is empty insert the value, otherwise Overflow occur.

→ write this code All the time before the programe. functions.

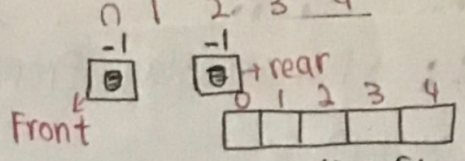
```
#include <iostream>
#define N 5
using namespace std;

int queue[N];
int front=-1;
int rear=-1;
```

...all values.

ENQUEUE: → For insertion.

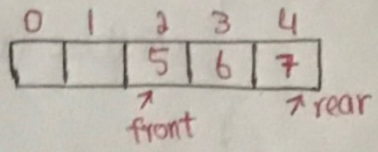
```
void enqueue() {
        int x;
        cout << "Enter value";
        cin >> x;
if(front == -1 && rear == -1){
      front = rear = 0;
      queue[rear] = x;
   }
else if( (rear+1) % N == front){
     cout << "OVERFLOW";
   }
else {  rear = (rear+1) % N;
        queue[rear] = x;
   }
}
```



enqueue() → 5;

front = rear = 0 → add to rear mean 0 index value.

→ All five values are added same.



enqueue(); → 10

→ check overflow:

(rear+1) % N == front

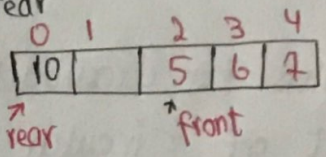
4+1 % 5 == 2

5 % 5 == 2

0 == 2 x

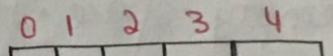
else rear = (rear+1) % N = 0

insert at 0 index and update rear



DEQUEUE: → For Deletion.

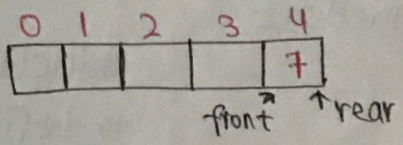
```
void dequeue() {
if( front == -1 && rear == -1) {
  cout << "UNDERFLOW";
}else if (front == rear) {
 cout << queue[front];
 front = rear = -1;
}
else {  cout << queue[front];
    front = (front + 1) % N;
}
```



front = rear = -1

dequeue();

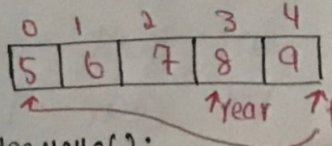
UNDERFLOW.



dequeue();

→ 7

→ front = rear = -1



dequeue();

→ After operation.

→ 9

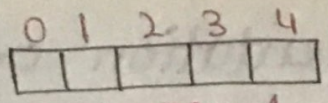
→ front = (front + 1) % N

4 + 1 % N

front = 5 % 5 = 0

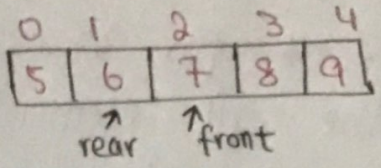
Display : → For display All values.

```
void display(){
if (front==-1 && rear==-1){
    cout<<"UNDERFLOW";
}
else {
        int i= front;
while ( i != rear){
    cout << queue[i];
    i = (i+1) % N;
}
cout << queue[rear];
}
```



front = rear = -1

display();
→ UNDERFLOW



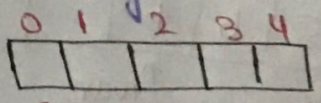
display();
f = front = 2 , rear = 1
2 != 1 ✓
→ 7 → i = 3 % 5 = 3
3 != 1 ✓
→ 8 → i = 4 % 5 = 4
4 != 1 ✓
→ 9 → i = 5 % 5 = 0
0 != 1
→ 5 → i = 1 % 5 = 1
1 != 1 ✗

→ This loop can only print value before rear after the iterations of loop just print rear value.

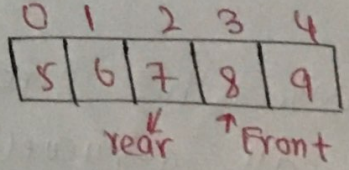
cout << queue[rear]. ∵ rear = 1
→ 6.

Peek(); → For display the Front value Only.

```
void peek(){
if (front==-1 && rear==-1){
    cout << "UNDERFLOW";
}
else {
    queue [Front];
}
}
```



front = rear = -1
peek();
→ UNDERFLOW.



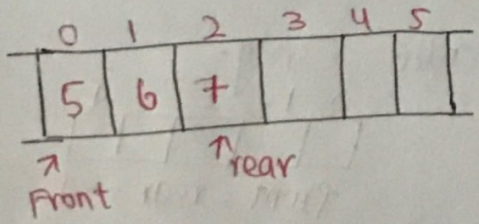
peek();
queue [Front];
→ 8

I Implementation of Queue using Stacks.

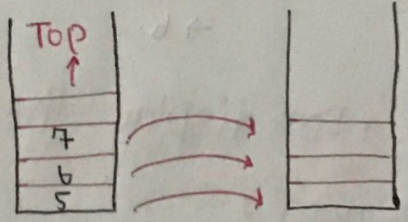
- Queue use the FIFO Mechanism.
- Stack use the LIFO Mechanism.
- We used to 2 stacks to implementation of Queuse.
- One stack for insertion and other for deletion.
- Pop the all elements of stack 1 to stack 2 and delete the top element because stack has only one open end, after ~~use~~ deletion move the elements again back to the stack 1.
- There are two methods for implementation of Queue compared to time complexity.

1st Method	2nd Method
• Enqueue O(1)	• Enqueue O(n)
• Dequeue O(n)	• dequeue O(1)

Queue (FIFO)



LIFO



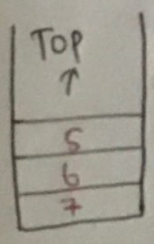
- dequeue().

→ 5

- dequeue()

→ 7

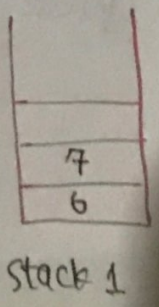
→ To solve this error we pop stack 1 to stack 2 and then call dequeue().



- dequeue().

→ 5

- Now shift/Pop remaining again to the stack 1.



Ist Method:

- Enqueue O(1)
- Dequeue O(n)

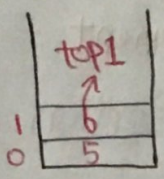
Write this code before implementation of functions.

```
#include<iostream>
# define N 5
using namespace std;
int stack1[N], stack2[N];
int top1=-1, top2=-1;
int count=0; → // For minimum iteration when convert one stack to another.
```

→we know we create queue using 2 stacks so first declare push and pop functions.

Push1(): →For insertion in the queue.

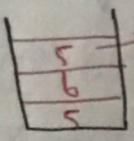
```
void push1(int x){
 if(top1==N-1){
    cout<<"OVERFLOW";
 }
 else{ top1++;
    stack1[top1]=x;
 }}
```

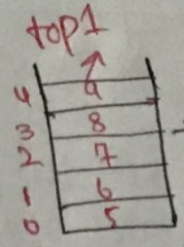


→ stack1
- push1(x) → 5

else → top1=1 → ++ = 2

stack2[2]=5; →





→ stack1
- push1(x) → 7

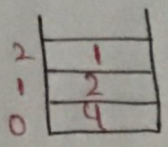
if (top1 == N-1)

4 == 5-1

4 == 4 → OVERFLOW.

Pop1(): →For delete stack1 value

```
int Pop1(){
  return stack1[top--];
}
```



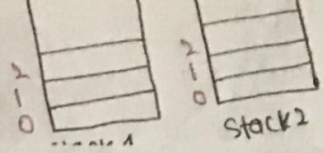
→ stack1
- pop1(); → int type.

return → 1

then --

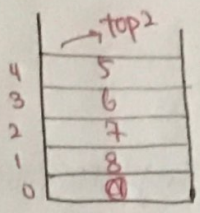
→ post Decrement

void dequeue(){

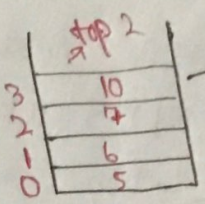


if **Push2():** → For insertion stack 1 elements.

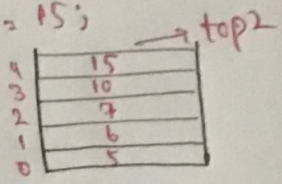
```
void push2(int x){
  if(top2==N-1){
    cout<<"OVERFLOW";
  }
  else{
    top2++;
  stack2[top2]=x;
  }
}
```



- push2(x) → 10
- if top2 == N-1
  4 == 5-1
  4 == 4
- OVERFLOW



- push2(x) → 15
- else top2 = 3
  top2++ = 4
- stack[4] = 15;

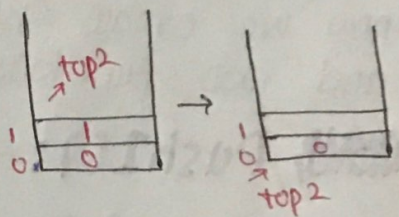


Pop2(): → For delete stack2 elements.

```
int pop2(){
  return stack2[top2--];
}
```

post decrement

- ~~push2~~
- pop2();
- → return 1
- then --



Enqueue(): → For insertion in the Queue.

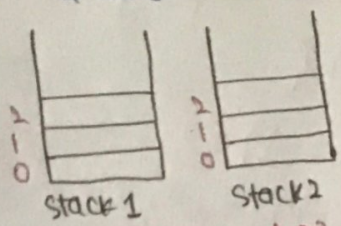
```
void enqueue(){
  int x;
cout<<"Enter value";
  cin>>x;
push1(x);
count++;
}
```

NOTE:- It is simply a enqueue function just calling the push1() function.

- We use only stack1 for insertions because stack2 is temprorary and use for deletion means shift stack1 to stack2 and then delete the value of stack2(top) after this shift back to stack1.

dequeue(): → For deletion in the Queue.

```
void dequeue(){
 if (top1==-1 && top2==-1){
 cout<<" Stack is Empty";
   }
 else {
  for(int i=0; i<count; i++){
   int a= pop1();
    push2(a);
  }
cout<<"Deleted is :"<<pop2();
   count--;
for (int i=0; i<count; i++){
    int b= pop2();
    push1(b);
}
  }
   }
```



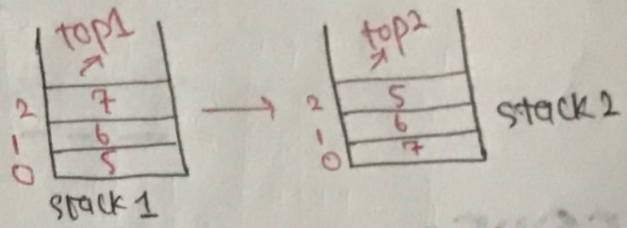
top1 = -1 , top2 = -1

• dequeue();

if top1==-1 && top2==-1

-1==-1 && -1==-1

→ Empty.



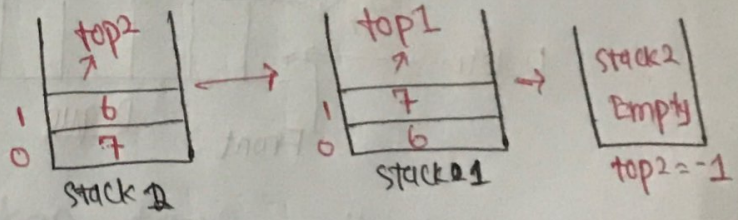
• dequeue();

else loop.

→ shift: stack 1 to stack 2

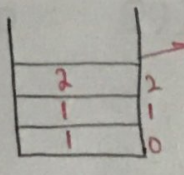
→ pop2() → 5 → deleted (--)

again shift to stack 1.



Display(): → For display all the elements in the Queue.
→ only use stack1.

```
void display(){
 if (top1==-1){
cout<<"Queue is Empty";
 } else {
   for (int i=0; i<=top1; i++){
   cout<<stack1[i];
} } }
```



• display();

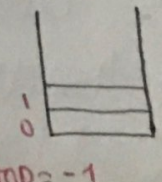
else i=0 , top1=2

0<=2 → 1 → i++

1<=2 → 1 → i++

2<=2 → 2 → i++

3<=2 x Exit



top = -1

• display();

if top1 == -1

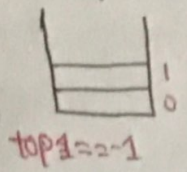
-1 == -1

• EMPTY.

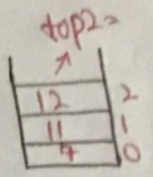
• output Restricted → Deletion only from one end only and

Peek(): For display the top element in the queue.

```
void peek(){
 if (top1==-1){
cout<<"UNDERFLOW";
}else{
cout<<stack1[top1];
}
}
```



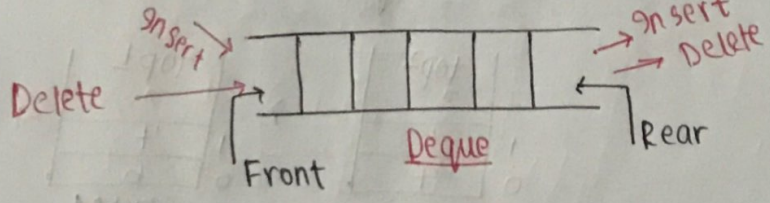

```
•peek();
if top1==-1
   -1==-1
•UNDERFLOW
```



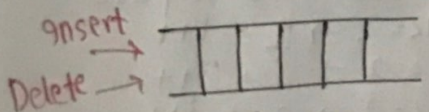

```
•peek();
else stack1[top].
     stack1[2]
→12
```

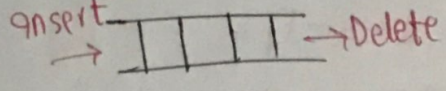
DEQUE:

- Double Ended List.
- It is also known as deck.
- In deque insertions and deletions can happen from the both side of the queue.



- We can set the front and rear on any side because insertion and deletion occur on both sides.
- It can use stack - LIFO and queue - FIFO

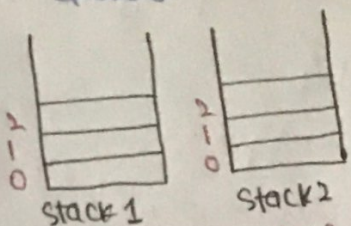




- In can ~~perform~~ have two ~~operations~~ types one is input restricted and the other is output restricted.
- Input restricted → Insertions only from one end and deletions from the both end.

dequeue(): → For deletion in the Queue.

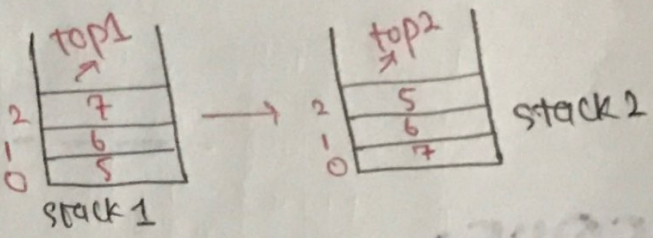
```
void dequeue(){
 if (top1==-1 && top2==-1){
 cout<<" Stack/Queue is Empty";
   }
 else {
  for(int i=0; i<count; i++){
   int a= pop1();
    push2(a);
   }
cout<<"Deleted is:"<<pop2();
  count--;
for (int i=0; i<count; i++){
   int b=pop2();
   push1(b);
 }
  }
   }
```



top1 = -1 , top2 = -1

• dequeue();

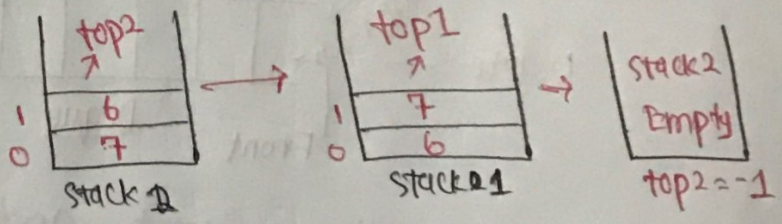
if top1==-1 && top2==-1
-1==-1 && -1==-1
→ Empty.



• dequeue();

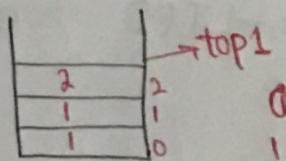
else loop.
→ shift: stack 1 to stack 2
→ pop2() → 5 → deleted. (--)

again shift to stack 1.



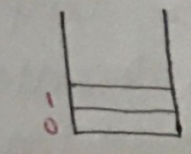
Display(): → For display all the elements in the Queue.
→ only use stack1.

```
void display(){
 if (top1==-1){
cout<<"Queue is Empty";
 } else {
   for (int i=0; i<=top1; i++){
   cout<<stack1[i];
} } }
```



• display();

else i=0, top1=2
0<=2 → 1 → i++
1<=2 → 1 → i++
2<=2 → 2 → i++
3<=2 x Exit



• display();

if top1 == -1
-1 == -1

• EMPTY.

- Output Restricted → Deletion only from one end only and insertions is possible on both end.

- Operations on Deque →
  - Insert at first
  - delete from front
  - Insert at rear
  - delete from rear

- getfront → front
- getrear → rear
- isFull() → return true if Deque is full.
- isEmpty() → return true if deque is Empty.
- We can implement Deque using Circular Array or Doubly Linked List.

Applications:

- Undo , Redo
- To check palindrome (same from front and last).
- Multiprocessor Scheduling