Page No.
Date

Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

Practical No : 13

Theory:

Q.i) Explain simple Queue with representation, examples, advantages, disadvantages, time complexity.

→ i) A simple queue is a data structure that follows the FIFO principle. In this structure, the first element added is the first to be removed, much like a line of people where the person at the front is served first.

Representation:

A simple queue can be represented using:

1. Arrays :
2. Linked lists.

ii) In simple queue, two pointers are used:

- Front: Points to the first element in the queue.
- Rear: Points to the last element in the queue.

iii) Operations:

Operations such as enqueue, dequeue, peek / front, IsEmpty and IsFull can be performed on simple queue.

iv) Advantages:

1. Simple to implement: Especially with arrays or linked lists.
2. Efficient Access: Access to the first element is constant time O(1).
3. FIFO order: Useful for tasks needing FIFO processing like CPU scheduling or printing tasks.

v) Disadvantages:

1. Fixed size in arrays: With array implementation, the size is fixed and the queue may fill up.

Q.2) Explain queue as an ADT. (with C++ function.)

→ i) A queue as an ADT is a type of data structure that operates on the FIFO principle.

ii) It provides essential operations like enqueue, dequeue, peek and checking if the queue is empty or full.

Basic implementation of a queue:

```
#include <iostream>
#define SIZE 5

class Queue {
```

```
private:
        int items[SIZE], front, rear;

public:
    Queue() {
        front = -1;
        rear  = -1;
    }

bool isfull(){
    return rear == SIZE - 1;
    }

void enqueue (int value){
    if (isfull()) {
std::cout<<"Queue is full\n";
    }else {
    if (front == -1)
        front = 0;
        rear++;
    item[rear] = value;
std:: cout << "Enqueue: "<< value << "\n";}

int dequeue() {
    if (isEmpty()) {
std::cout<<"Queue is empty\n";
        return value;
    }
}
```

```
int peek() {
    if (isEmpty()) {
std::cout << "queue is empty\n";
        return -1;
    } else {
    return items[front];
    }
  }
};
int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
std::cout << "Front element:" << q.peek();
    q.dequeue();
    q.dequeue();
    q.dequeue();
    q.dequeue();

    return 0;
}
```

Page No.
Date

Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

Q.3) Difference between stack & Queue.

	Feature	stack	Queue
i)	Principle	LIFO (Last in, first out)	FIFO (First in, first out)
ii)	Main operation	push, pop	enqueue, dequeue,
iii)	Access order	Last added element is accessed first	First added element is accessed first.
iv)	Use case	Can be implemented with arrays or linked lists.	Can be implemeted with linked lists.
v)	Use case	Call stack, undo operations	Task scheduling printing.
vi)	Memory wastage	No memory wastage	Possible memory wastage.

16

Algorithms:

a) Check if Queue is Empty:

1. If front == -1 OR front > rear:
2. RETURN True
3. ELSE:
4. RETURN False.

b) Check if Queue is full:

1. If rear == capacity - 1:
2. RETURN True
3. ELSE:
4. RETURN False.

c) Insert Element in Queue:

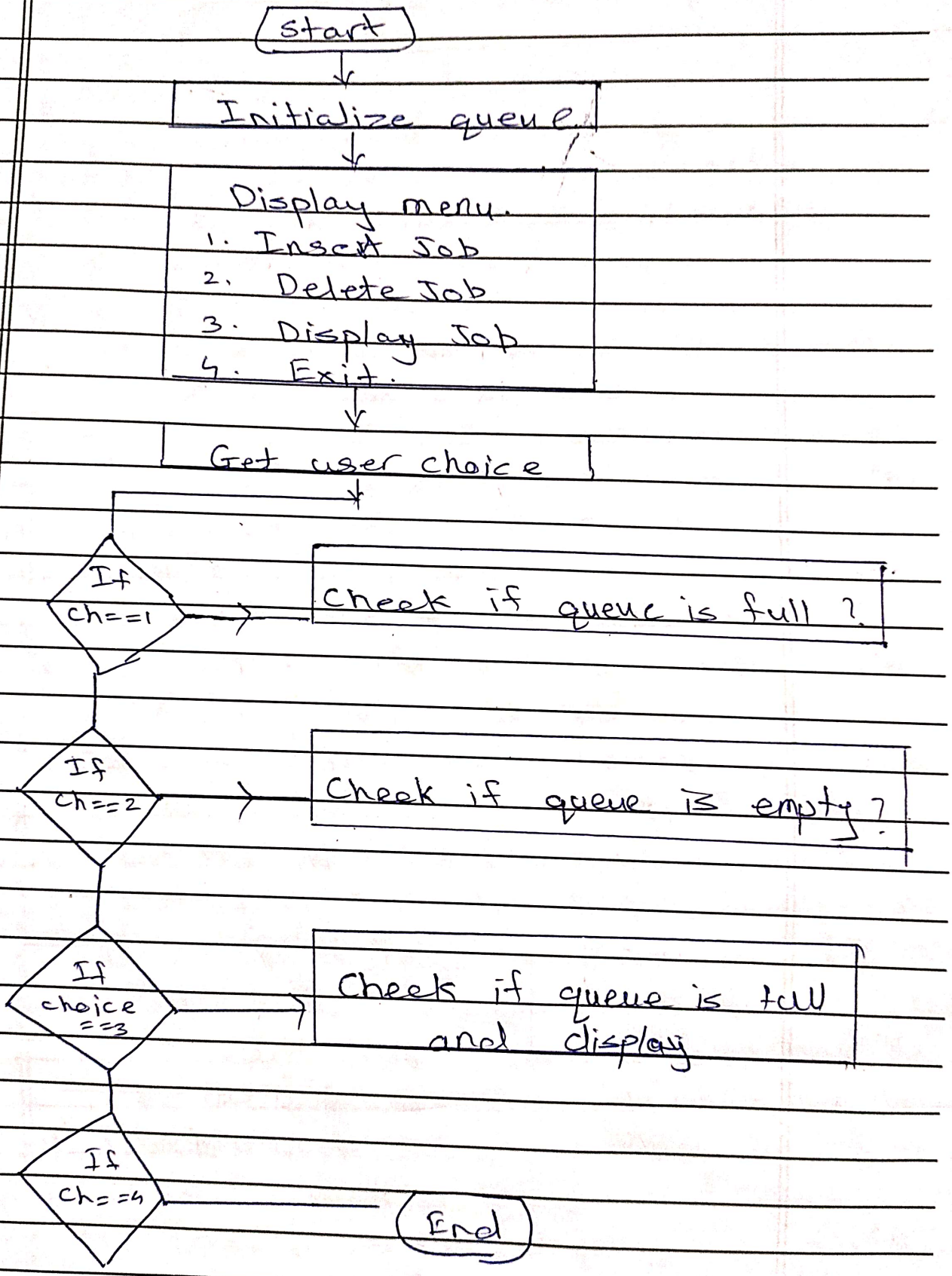
1. If ISFULL (queue, capacity):
2. PRINT "Queue overflow"
3. ELSE:
4. If front == -1:
5. SET front = 0
6. INCREMENT rear
7. queue[rear] = element
8. PRINT "Job added successfully"

d) Delete element from Queue:

```
1. If  IS EMPTY (queue):
2.         PRINT " Queue underflow"
3. Else :
4.         PRINT " Job removed:", queue[front]
5.         INCREMENT front
6.         IF front > rear :
7.                 SET front = rear = -1.
```

e) Display Elements of Queue.

```
1. IF IS EMPTY (queue):
2.         PRINT " Queue is empty"
3. ELSE :
4.         FOR i FROM front To rear:
5.                 PRINT queue[i]
```



Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

Practical no: 14

Theory:

Q.1) Explain dequeue with representation, examples, advantages, disadvantages, time complexity.

→ a) Representation:

1. Arrays: Fixed size with circular arrangements.
2. Doubly linked list: Dynamic size with each node pointing to the next & previous node.

b) Examples:

1. Input-restricted dequeue: Allow insertion only at the rear and deletion at both ends.
2. Output-restricted Dequeue: Allows deletion only from front and insertion at both ends.

c) Advantages:

1. Flexibility: Allows insertion and deletions at both ends.
2. Efficient use: Helps in managing situations requires flexible aecess.

d) Disadvantages:

1. More complex implementation: Managing complex for both ends increases code complexity.

2. Memory usages: When implemented using arrays, circular indexing may lead to wasted space.

e) Time complexity:
- Time complexity for insertion, deletion & checking full/empty is O(1).

Q.2) Explain dequeue as an ADT.

→ i) A dequeue is an ADT that allows insertion & deletion of elements at both the front and rear ends.

ii) It can operate in both directions, unlike a simple queue that only allows insertion at the rear & deletion at the front.

iii) Program:

```
#include <iostream>
#define SIZE 5

class Dequeue {
private:
    int arr[SIZE];
    int front, rear;
public:
    Dequeue() {
        front = -1;
        rear = 0;
    }
```

```
bool isFull() {
return (front == 0 && rear == SIZE-1 )
|| (front == rear+1);
        }
bool isEmpty() {
    return front == -1;
    }
void insertFront(int value)
{
    if (isFull()) {
std::cout << " Dequeue is full \n";
    } else {
    if (front == -1) { front = rear = 0; }
else if {
(front == front = SIZE -1 }
else {
        front--;
        }
arr[front] = value;
std::cout << " Inserted at front : " << value;
    }
}
int main() {
  Dequeue dq;
  dq.insertRear(10);
  dq.insertFront(20);
    return 0;
}
```

Page No.
Date

Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

Q.3) Difference between Queue & dequeue.

	Features	Dequeue	Queue
i)	Insertion	Both at the front & rear.	only at the rear
ii)	Deletion	Both from the front & rear.	only from the front.
iii)	Flexibility	More flexible	Limited ope.
iv)	Use case	Palindrome checking, job scheduling.	Printer queues, task scheduling.
v)	Types	Input-restricted, Output-restricted	simple & circular queue.
vi)	Complexity	More complex operations	simpler structure

Algorithms :

1) Check if the queue is empty.

```
1. IF front == -1 :
2.     RETURN True
3. ELSE
4.     RETURN False
```

2) ~~If~~ Check if the queue is full:

```
1. If (front == 0 AND rear == capacity-1)
2.    OR (Front == rear+1):
3.    RETURN True
4. ELSE
5.    RETURN False.
```

3) Insert element in queue from rear:

```
1. IF ISFULL() :
2.     PRINT " Queue overflow"
3. ELSE
4.     If front == -1 :
5.         SET front = 0.
6.     INCREMENT rear (circularly) : rear =
           (rear+1) % capacity.
7. ~~PRINT~~ queue [rear] = element
8. PRINT " Element added at rear ".
```

d) Insert Element in Queue from front.

```
1. IF ISFULL():
2.     PRINT "Queue overflow"
3. ELSE:
4.     IF front == -1:
5.         SET front = rear = 0
6.     ELSE:
7.         DECREMENT front (circularly):
           front = (front - 1 + capacity) % capacity
8. queue[front] = element.
9. PRINT "Element added at front"
```

e) Delete front element from Queue.

```
1. IF ISEMPTY():
2.     PRINT "Queue underflow"
3. ELSE:
       PRINT "Delete element:", queue[front]
       IF front == rear:
           SET front = rear = -1
       ELSE:
           INCREMENT front (circularly):
           front = (front + 1) % capacity.
```

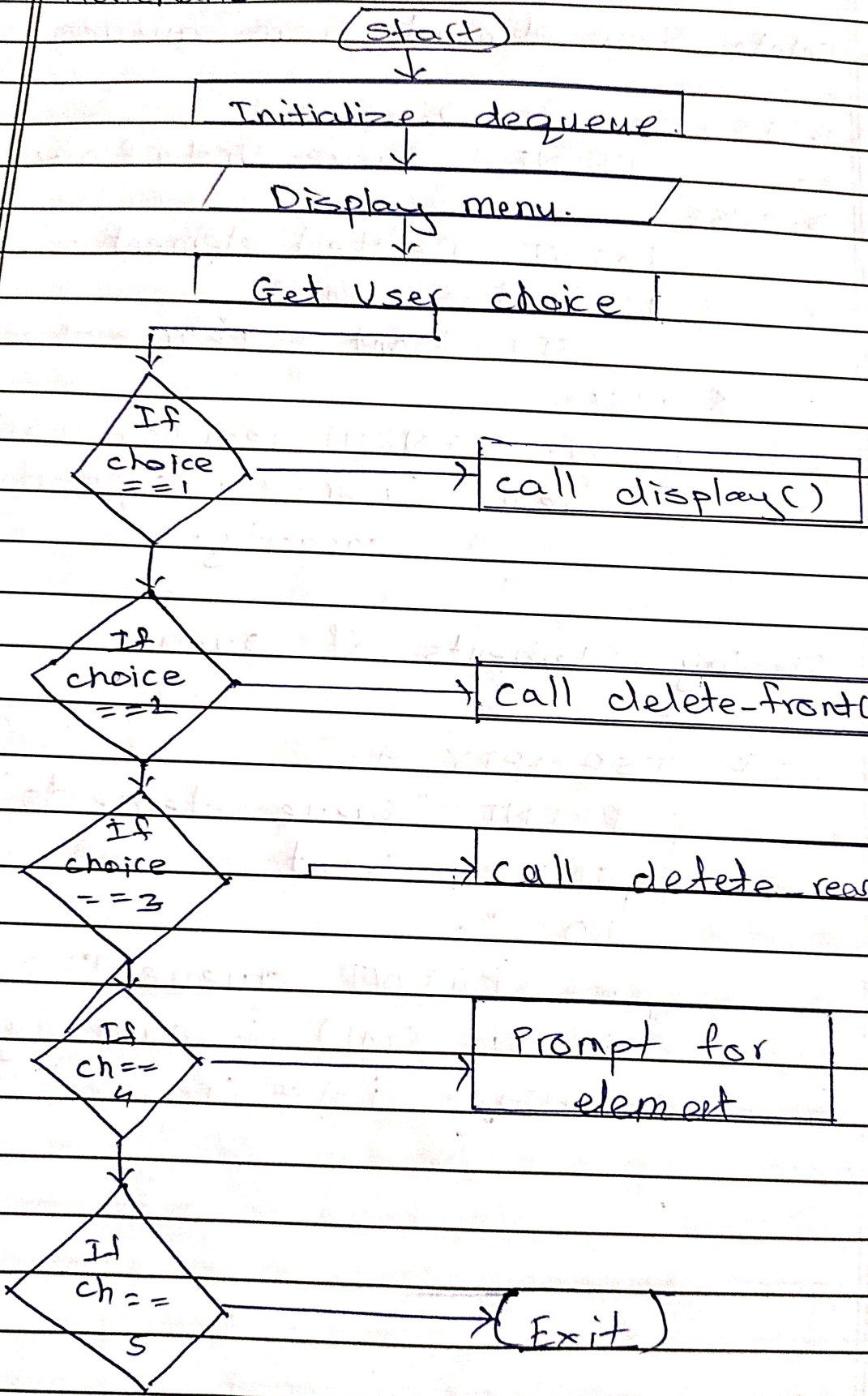
f) Delete Rear element from queue:

```
1. IF ISEMPTY():
2.     PRINT "Queue Underflow"
3. ELSE:
       PRINT "Deleted element :", queue[rear]
       IF front == rear:
           SET front = rear = -1
       ELSE:
           DECREMENT rear (circularly):
           rear = (rear - 1 + capacity) %
                  capacity.
```

g) Display Elements of queue:

```
1. If ISEMPTY():
       PRINT "Queue elements"
       SET i = front.
       DO:
           PRINT queue[i]
           i = (i+1) % capacity.
       WHILE i != (rear+1) % capacity.
```

Flowcharts:


Start
Initialize dequeue
Display menu.
Get User choice
If choice ==1
call display()
If choice ==2
call delete_front()
If choice ==3
call delete_rear()
If ch== 4
Prompt for element
If ch== 5
Exit

Page No.
Date

Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

NAAC

Practical no : 15

Theory:

Q.1) Explain circular queue with representation, Examples, Advantages, disadvantages.

→ i) A circular queue is a linear data structure that connects the last position back to the first position, forming a layout.

ii) Unlike a simple queue, which has a fixed beginning & end, a circular queue wraps around when it reaches the end of the allocated memory space, which helps utilize space efficiency.

iii) Representation:

A circular queue can be represented using arrays & linked lists.

In a circular queue, when the rear pointer reaches the last index of the array, it wraps around to index 0 if there is free space.

iv) Advantages:

1) Efficient use of space.

2) Avoids wastage.

v) Disadvantages:

1) Slightly complexity is more.

O(1).

Q.2) Explain circular queue as an ADT.

→

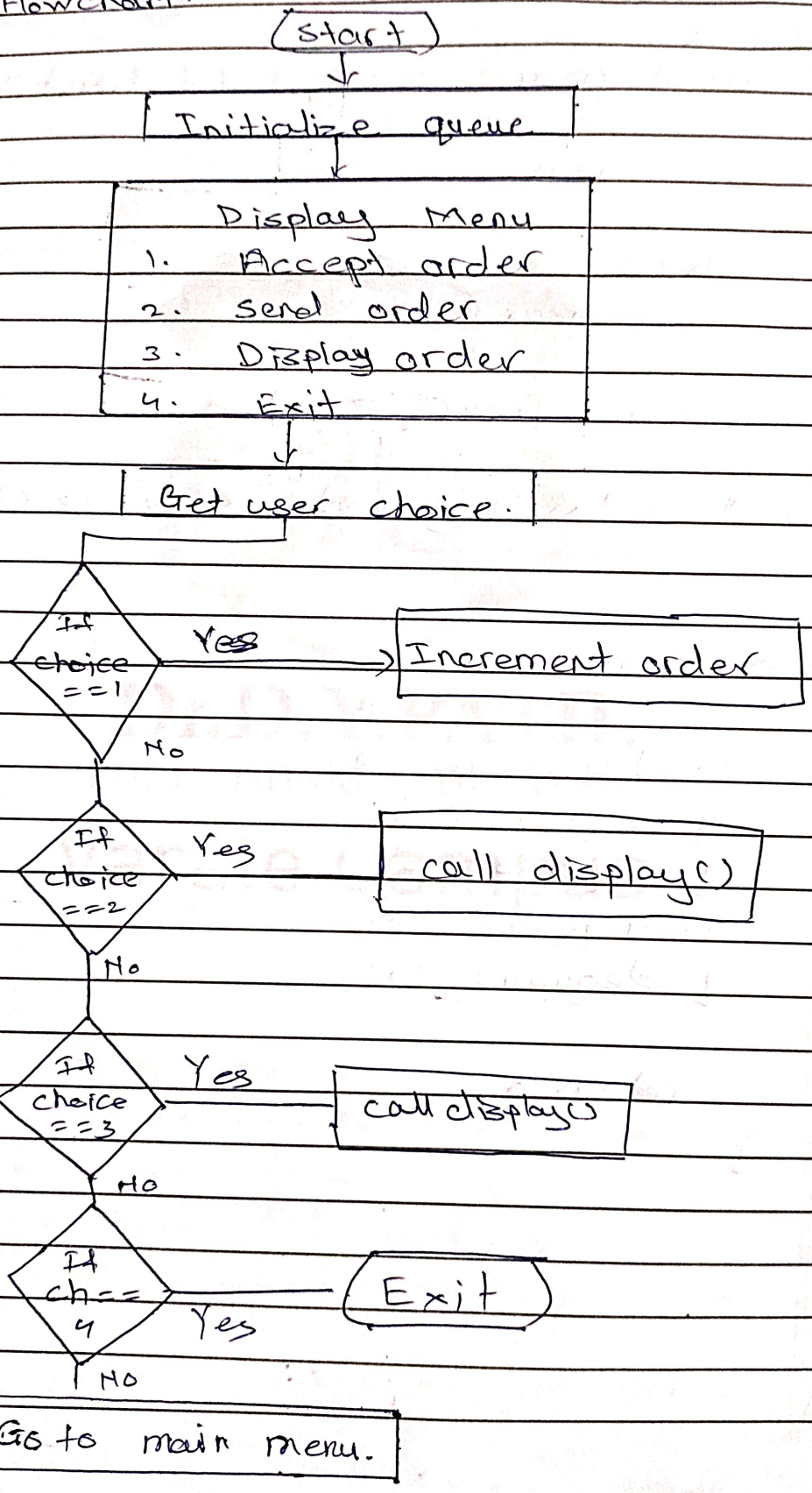
```
#include<iostream>
#define SIZE 5

class CircularQueue {

private:
        int items[SIZE];
        int front, rear;

public:
        CircularQueue() {
            front = -1;
            rear = -1;
        }
bool isFull() {
 return (front == 0 && rear == SIZE -1) ||
    (rear == (front - 1) % (SIZE -1));
 }
bool isEmpty() {
        return front == -1;
 }
void enqueue(int value) {
    if (isFull()) {
std::cout<<" Queue is full\n";
        }
else {
  if (front == -1) {
```

```
        front = rear = 0;
    }
    else if (rear == SIZE-1 && front != 0)
    {
        rear = 0;
    }
    else {
        rear++;
    }
    items[rear] = value;

    std::cout << "Enqueued: " << value << "\n";
    }
};
int main() {

    circularQueue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.dequeue();

    return 0;
}
```

Page No.
Date

Dr.D.Y.PATIL EDUCATIONAL FEDERATION Varale Campus

* Flowchart:



Page No.
Date

Dr.D.Y.PATIL
Technical Campus

Algorithms:

a. Check if the queue is empty.

1. If front == -1:
2. PRINT True
3. ELSE:
4. PRINT False

b. Check if the queue is full:

1. If (rear + 1) % capacity == front:
2. RETURN True
3. ELSE:
4. RETURN False

c. Insert element in queue from Rear:

1. If ISFULL():
2. PRINT "Queue overflow"
3. ELSE:
4. IF front == -1:
5. SET front = 0.
6. rear = (rear + 1) % capacity.
7. queue [rear] = element
9. PRINT "Element added"
9. END.

d) Delete front Element from queue:

```
1. IF ISEMPTY:
2.     PRINT " Queue underflow"
3. ELSE:
4.     PRINT " Deleted element:" queue[front]
5.     IF front == rear:
6.         SET front = rear = -1
7.     Else :
8.         front = (front + 1) %. capacity.
```