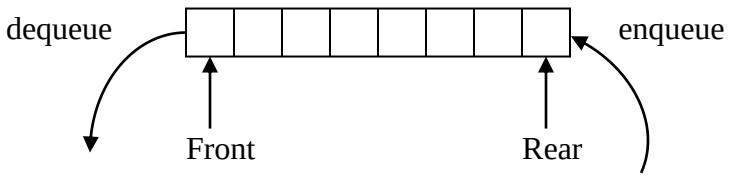
CHAPTER FIVE

Queue

- a data structure that has access to its data at the front and rear.
- operates on FIFO (Fast In First Out) basis.
- uses two pointers/indices to keep tack of information/data.
- has two basic operations:
    - enqueue - inserting data at the rear of the queue
    - dequeue – removing data at the front of the queue



Example:

Operation	Content of queue
Enqueue(B)	B
Enqueue(C)	B, C
Dequeue()	C
Enqueue(G)	C, G
Enqueue (F)	C, G, F
Dequeue()	G, F
Enqueue(A)	G, F, A
Dequeue()	F, A

5.1. Simple array implementation of enqueue and dequeue operations

Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int FRONT =-1,REAR =-1;
int QUEUESIZE=0;
```

- To enqueue data to the queue
    - check if there is space in the queue

REAR<MAX_SIZE-1 ?
Yes: - Increment REAR
- Store the data in Num[REAR]
- Increment QUEUESIZE
FRONT = = -1?
Yes: - Increment FRONT
No: - Queue Overflow

- To dequeue data from the queue
  - check if there is data in the queue
    QUEUESIZE > 0 ?
    Yes: - Copy the data in Num[FRONT]
    - Increment FRONT
    - Decrement QUEUESIZE
    No: - Queue Underflow

Implementation:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
{
        if(Rear<MAX_SIZE-1)
        {
            REAR++;
            Num[REAR]=x;
            QUEUESIZE++;
            if(FRONT = = -1)
                    FRONT++;
        }
        else
            cout<<"Queue Overflow";
}
int dequeue()
{
        int x;
        if(QUEUESIZE>0)
        {
            x=Num[FRONT];
            FRONT++;
            QUEUESIZE--;

        }
        else
            cout<<"Queue Underflow";
        return(x);
}
```

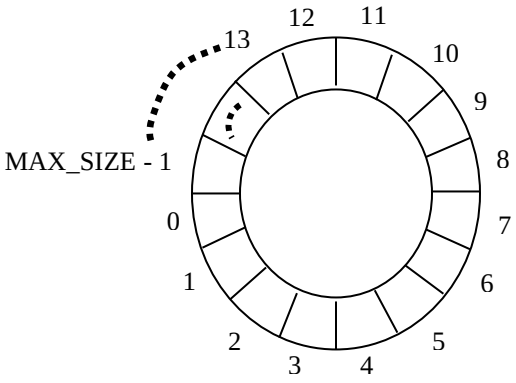
5.2. Circular array implementation of enqueue and dequeue operations

A problem with simple arrays is we run out of space even if the queue never reaches the size of the array. Thus, simulated circular arrays (in which freed spaces are re-used to store data) can be used to solve this problem.

Example: Consider a queue with MAX_SIZE = 4

Operation	Simple array							Circular array						
	Content of the array				Content of the Queue	QUEUE SIZE	Message	Content of the array				Content of the queue	QUEUE SIZE	Message
Enqueue(B)	B				B	1		B				B	1	
Enqueue(C)	B	C			BC	2		B	C			BC	2	
Dequeue()		C			C	1			C			C	1	
Enqueue(G)		C	G		CG	2			C	G		CG	2	
Enqueue (F)		C	G	F	CGF	3			C	G	F	CGF	3	
Dequeue()			G	F	GF	2				G	F	GF	2	
Enqueue(A)			G	F	GF	2	Overflow	A		G	F	GFA	3	
Enqueue(D)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	
Enqueue(C)			G	F	GF	2	Overflow	A	D	G	F	GFAD	4	Overflow
Dequeue()				F	F	1		A	D		F	FAD	3	
Enqueue(H)				F	F	1	Overflow	A	D	H	F	FADH	4	
Dequeue ()					Empty	0		A	D	H		ADH	3	
Dequeue()					Empty	0	Underflow		D	H		DH	2	
Dequeue()					Empty	0	Underflow			H		H	1	
Dequeue()					Empty	0	Underflow					Empty	0	
Dequeue()					Empty	0	Underflow					Empty	0	Underflow

The circular array implementation of a queue with MAX_SIZE can be simulated as follows:



Analysis:

Consider the following structure: int Num[MAX_SIZE];

We need to have two integer variables that tell:

- the index of the front element
- the index of the rear element

We also need an integer variable that tells:

- the total number of data in the queue

```
int FRONT =-1,REAR =-1;
int QUEUESIZE=0;
```

- To enqueue data to the queue
  - check if there is space in the queue
    QUEUESIZE<MAX_SIZE ?
    Yes: - Increment REAR
    REAR = = MAX_SIZE ?
    Yes: REAR = 0
    - Store the data in Num[REAR]
    - Increment QUEUESIZE
    FRONT = = -1?
    Yes: - Increment FRONT
    No: - Queue Overflow

- To dequeue data from the queue
  - check if there is data in the queue
    QUEUESIZE > 0 ?
    Yes: - Copy the data in Num[FRONT]
    - Increment FRONT
    FRONT = = MAX_SIZE ?
    Yes: FRONT = 0
    - Decrement QUEUESIZE
    No: - Queue Underflow

Implementation:

```
const int MAX_SIZE=100;
int FRONT =-1, REAR =-1;
int QUEUESIZE = 0;

void enqueue(int x)
{
    if(QUEUESIZE<MAX_SIZE)
    {
        REAR++;
        if(REAR = = MAX_SIZE)
            REAR=0;
        Num[REAR]=x;
        QUEUESIZE++;
        if(FRONT = = -1)
            FRONT++;
    }
    else
        cout<<"Queue Overflow";
}
int dequeue()
{
```

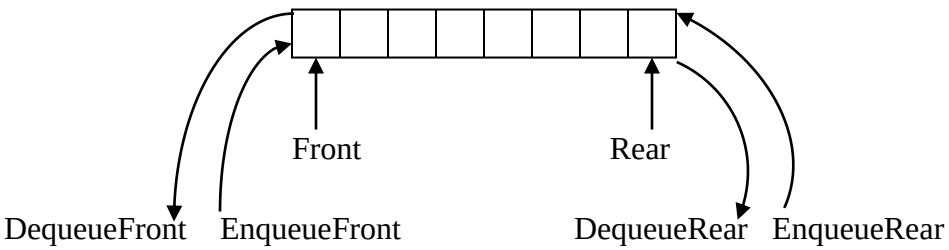
```
        int x;
        if(QUEUESIZE>0)
        {
            x=Num[FRONT];
            FRONT++;
            if(FRONT = = MAX_SIZE)
                    FRONT = 0;
            QUEUESIZE--;

        }
        else
            cout<<"Queue Underflow";
        return(x);
}
```

5.3. Linked list implementation of enqueue and dequeue operations

Enqueue- is inserting a node at the end of a linked list
Dequeue- is deleting the first node in the list

5.4. Deque (pronounced as Deck)

- is a Double Ended Queue
- insertion and deletion can occur at either end
- has the following basic operations
    - EnqueueFront – inserts data at the front of the list
    - DequeueFront – deletes data at the front of the list
    - EnqueueRear – inserts data at the end of the list
    - DequeueRear – deletes data at the end of the list
- implementation is similar to that of queue
- is best implemented using doubly linked list



5.5. *Priority Queue*

- is a queue where each data has an associated key that is provided at the time of insertion.
- Dequeue operation deletes data having highest priority in the list
- One of the previously used dequeue or enqueue operations has to be modified

Example: Consider the following queue of persons where females have higher priority than males (gender is the key to give priority).

Abebe	Alemu	Aster	Belay	Kedir	Meron	Yonas
Male	Male	Female	Male	Male	Female	Male

Dequeue()- deletes Aster

Abebe	Alemu	Belay	Kedir	Meron	Yonas
Male	Male	Male	Male	Female	Male

Dequeue()- deletes Meron

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Now the queue has data having equal priority and dequeue operation deletes the front element like in the case of ordinary queues.

Dequeue()- deletes Abebe

Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male

Dequeue()- deletes Alemu

Belay	Kedir	Yonas
Male	Male	Male

Thus, in the above example the implementation of the dequeue operation need to be modified.

5.5.1. Demerging Queues

- is the process of creating two or more queues from a single queue.
- used to give priority for some groups of data

Example: The following two queues can be created from the above priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Algorithm:

```
create empty females and males queue
while (PriorityQueue is not empty)
{
        Data=DequeuePriorityQueue(); // delete data at the front
        if(gender of Data is Female)
                EnqueueFemale(Data);
        else
                EnqueueMale(Data);
}
```

5.5.2. Merging Queues

- is the process of creating a priority queue from two or more queues.
- the ordinary dequeue implementation can be used to delete data in the newly created priority queue.

Example: The following two queues (females queue has higher priority than the males queue) can be merged to create a priority queue.

Aster	Meron
Female	Female

Abebe	Alemu	Belay	Kedir	Yonas
Male	Male	Male	Male	Male

Aster	Meron	Abebe	Alemu	Belay	Kedir	Yonas
Female	Female	Male	Male	Male	Male	Male

Algorithm:

```
create an empty priority queue
while(FemalesQueue is not empty)
        EnqueuePriorityQueue(DequeueFemalesQueue());
while(MalesQueue is not empty)
        EnqueuePriorityQueue(DequeueMalesQueue());
```

It is also possible to merge two or more priority queues.

Example: Consider the following priority queues and suppose large numbers represent high priorities.

ABC	CDE	DEF	FGH	HIJ
52	41	35	16	12

BCD	EFG	GHI	IJK	JKL
47	32	13	10	7

Thus, the two queues can be merged to give the following priority queue.

ABC	BCD	CDE	DEF	EFG	FGH	GHI	HIJ	IJK	JKL
52	47	41	35	32	16	13	12	10	7

5.6. Application of Queues

i. Print server- maintains a queue of print jobs

```
Print()
{
    EnqueuePrintQueue(Document)
}
EndOfPrint()
{
    DequeuePrintQueue()
}
```

ii. Disk Driver- maintains a queue of disk input/output requests

iii. Task scheduler in multiprocessing system- maintains priority queues of processes

iv. Telephone calls in a busy environment –maintains a queue of telephone calls

v. Simulation of waiting line- maintains a queue of persons