UNIT-III

QUEUES

1. Introduction of Queues

Let us explain the concept of queues using the analogies given below.

- People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

In all these examples, we see that the element at the first position is served first. Same is the case with queue data structure. A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists. In this section, we will see how queues are implemented using each of these data structures.

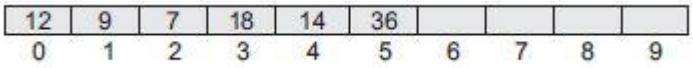
2. Representation Of Queues

In this we have two types of Representation Of Queues. They are:

1. Array Representation
2. Linked list Representation

1. Array Representation:

- Queues can be easily represented using linear arrays
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.



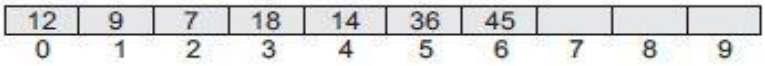
ARRAY REPRESENTATION OF QUEUE

OPERATION ON QUEUES

We have two types of operations on Queues. They are:

1. INSERTION
2. DELETION

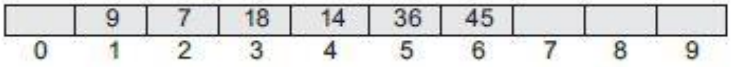
1.INSERTION:



QUEUE AFTER INSERTION OF A NEW ELEMENT

- FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.
- Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.
- However, before inserting an element in a queue, we must check for overflow conditions
- An overflow will occur when we try to insert an element into a queue that is already full.
- When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0.

2. DELETION:



QUEUE AFTER DELETION OF AN ELEMENT

- If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue.
- Similarly, before deleting an element from a queue, we must check for underflow conditions.
- An underflow condition occurs when we try to delete an element from a queue that is already empty.
- If FRONT = –1 and REAR = –1, it means there is no element in the queue.

ALGORITHM TO INSERT AN ELEMENT IN A QUEUE

Step 1: IF REAR = MAX-1
Write OVERFLOW
Goto step 4
[END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR =
ELSE
SET REAR = REAR + 1
[END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT

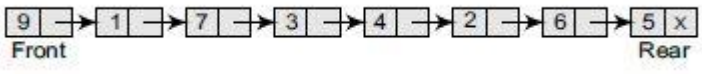
ALGORITHM TO DELETE AN ELEMENT FROM A QUEUE

Step 1: IF FRONT = -1 OR FRONT > REAR
Write UNDERFLOW
ELSE
SET FRONT = FRONT + 1
[END OF IF]
Step 2: EXIT

2. LINKED LIST REPRESENTATION:

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.
- All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.



LINKED QUEUE

OPERATION ON LINKED QUEUES

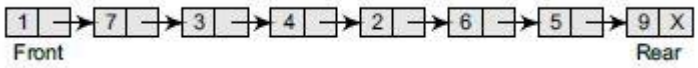
- A queue has two basic operations: insert and delete.
- The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue. Apart from

this, there is another operation peek which returns the value of the first element of the queue.

1. INSERT OPERATION:
    - The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue.

LINKED QUEUE

- To insert an element with value 9, we first check if FRONT=NULL.
- If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part.
- The new node will then be called both FRONT and REAR.
- However, if FRONT!= NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear.
- Thus the updated queue becomes as:



LINKED AFTER INSERTING A NEW NODE

ALGORITHM TO INSERT AN ELEMENT IN A LINKED QUEUE:

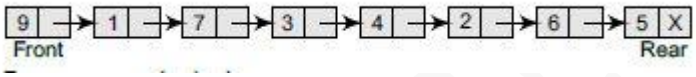
Step 1: Allocate memory for the new node and name
it as PTR

Step 2: SET PTR DATA = VAL

Step 3: IF FRONT = NULL
SET FRONT = REAR = PT
SET FRONT NEXT = REAR NEXT = NULL
ELSE
SET REAR NEXT = PTR
SET REAR = PTR
SET REAR NEXT = NULL
[END OF IF]

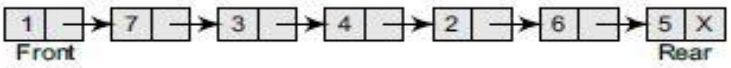
Step 4: END

2. DELETE OPERATION:
   - The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT.
   - However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done.
   - If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed.



LINKED QUEUE

- To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue.
- Thus the updated queue becomes as:



LINKED QUEUE AFTER DELETION OF AN ELEMENT

ALGORITHM TO DELETE AN ELEMENT FROM AN LINKED QUEUE:

Step 1: IF FRONT = NULL
Write Underflow
Go to Step 5
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END

3. IMPLEMENTATION OF QUEUES

PROGRAMMING EXAMPLE 1:

1. Write a program to implement a linear queue.

```
##include <stdio.h>
#include <conio.h>
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
int option, val;
do
{
printf("\n\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
case 1:
insert();
break;
```

```
case 2:
val = delete_element();
if (val != -1)
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if (val != -1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
}while(option != 5);
getch();
return 0;
}
void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(rear == MAX-1)
printf("\n OVERFLOW");
else if(front == -1 && rear == -1)
front = rear = 0;
else
rear++;
queue[rear] = num;
}
```

```
int delete_element()
{
int val;
if(front == -1 || front>rear)
{
printf("\n UNDERFLOW");
return -1;
}
else
{
val = queue[front];
front++;
if(front > rear)
front = rear = -1;
return val;
}
}
int peek()
{
if(front==-1 || front>rear)
{
printf("\n QUEUE IS EMPTY");
return -1;
}
else
{
return queue[front];
}
}
void display()
{
```

```
int i;
printf("\n");
if(front == -1 || front > rear)
printf("\n QUEUE IS EMPTY");
else
{
for(i = front;i <= rear;i++)
printf("\t %d", queue[i]);
}
}
```

Output

```
***** MAIN MENU *****"
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 50
```

PROGRAMMING EXAMPLE 2:

2. Write a program to implement a linked queue.

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
int data;
struct node *next;
};
struct queue
{
```

```
struct node *front;
struct node *rear;
};
struct queue *q;
void create_queue(struct queue *);
struct queue *insert(struct queue *,int);
struct queue *delete_element(struct queue *);
struct queue *display(struct queue *);
int peek(struct queue *);
int main()
{
int val, option;
create_queue(q);
clrscr();
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. INSERT");
printf("\n 2. DELETE");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to insert in the queue:");
scanf("%d", &val);
q = insert(q,val);
break;
```

```
case 2:
q = delete_element(q);
break;
case 3:
val = peek(q);
if(val! = –1)
printf("\n The value at front of queue is : %d", val);
break;
case 4:
q = display(q);
break;
}
}while(option != 5);
getch();
return 0;
}
void create_queue(struct queue *q)
{
q -> rear = NULL;
q -> front = NULL;
}
struct queue *insert(struct queue *q,int val)
{
struct node *ptr;
ptr = (struct node*)malloc(sizeof(struct node));
ptr -> data = val;
if(q -> front == NULL)
{
q -> front = ptr;
q -> rear = ptr;
q -> front -> next = q -> rear -> next = NULL;
```

```
}
else
{
q -> rear -> next = ptr;
q -> rear = ptr;
q -> rear -> next = NULL;
}
return q;
}
struct queue *display(struct queue *q)
{
struct node *ptr;
ptr = q -> front;
if(ptr == NULL)
printf("\n QUEUE IS EMPTY");
else
{
printf("\n");
while(ptr!=q -> rear)
{
printf("%d\t", ptr -> data);
ptr = ptr -> next;
}
printf("%d\t", ptr -> data);
}
return q;
}
struct queue *delete_element(struct queue *q){
struct node *ptr;
ptr = q -> front;
if(q -> front == NULL)
```

```
printf("\n UNDERFLOW");
else
{
q -> front = q -> front -> next;
printf("\n The value being deleted is : %d", ptr -> data);
free(ptr);
}
return q;
}
int peek(struct queue *q)
{
if(q->front==NULL)
{
printf("\n QUEUE IS EMPTY");
return –1;
}
else
return q->front->data;
```

Output

```
*****MAIN MENU*****
1. INSERT
2. DELETE
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

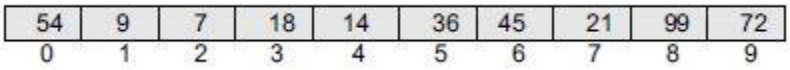
4.APPLICATIONS OF QUEUES

A queue data structure can be classified into the following types:

1. Circular Queue 2. Deque 3. Priority Queue 4. Multiple Queue

1. CIRCULAR QUEUE:

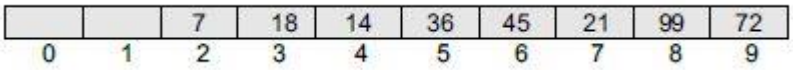
In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.



LINEAR QUEUE

Here, FRONT = 0 and REAR = 9.

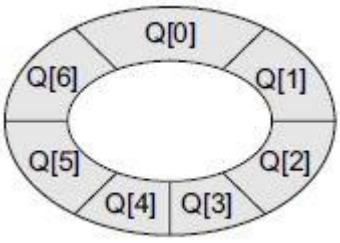
Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made.



Queue After Two Successive Deletions

Here, front = 2 and REAR = 9.

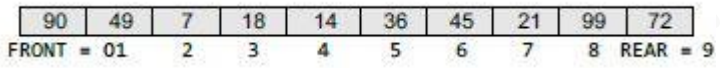
- Suppose we want to insert a new element in the queue shown in Fig. 8.14. Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue.
- To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.
- The second option is to use a circular queue. In the circular queue, the first index comes right after the last index.



CIRCULAR QUEUE

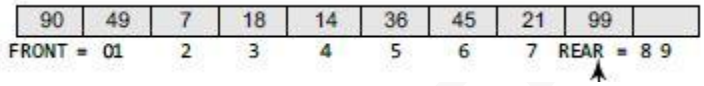
- The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations.
- For insertion, we now have to check for the following three conditions:

1. If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given .



FULL QUEUE

2. If rear != MAX – 1, then rear will be incremented and the value will be inserted



QUEUE WITH VACCANT LOCATIONS

3. If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there,

Inserting An Element In A Circular Queue

ALGORITHM TO INSERT AN ELEMENT IN CIRCULAR QUEUE

Step 1: IF FRONT = and Rear = MAX - 1

Write OVERFLOW

Goto step 4

[End OF IF]

STEP 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR =0

ELSE IF REAR = MAX - 1 and FRONT != 0

SET REAR =0

ELSE

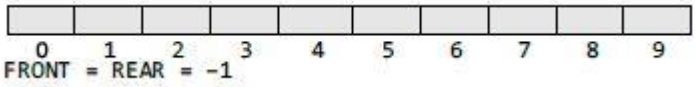
SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE [REAR] = VAL

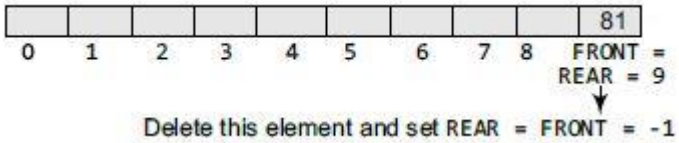
Step 4: EXIT

- After seeing how a new element is added in a circular queue, let us now discuss how deletions are performed in this case.
- To delete an element, again we check for three conditions.
  1. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.



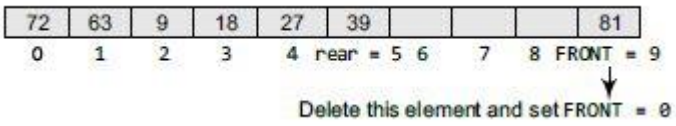
EMPTY QUEUE

  2. If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1.



QUEUE WITH A SINGLE ELEMENT

  3. If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0.



Queue Where Front=Max-1 Before Deletion

ALGORITHM TO DELETE AN ELEMENT IN CIRCULAR QUEUE

Step 1: IF FRONT = -1

Write UNDERFLOW

Goto Step 4

[END of IF]

Step 2: SET VAL = QUEUE [FRONT]

Step 3: IF FRONT = REAR

SET FRONT = REAR = -1

ELSE

IF FRONT = MAX -1

SET FRONT =0

```
        ELSE
            SET FRONT = FRONT + 1
        [END of IF]
        [END OF IF]
```

Step 4: EXIT

PROGRAMMING EXAMPLE:

3. Write a program to implement a circular queue.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int queue[MAX];
int front=–1, rear=–1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
int option, val;
clrscr();
do
{
printf("\n ***** MAIN MENU *****");
printf("\n 1. Insert an element");
printf("\n 2. Delete an element");
printf("\n 3. Peek");
printf("\n 4. Display the queue");
printf("\n 5. EXIT");
printf("\n Enter your option : ");
scanf("%d", &option);
switch(option)
```

```
{
case 1:
insert();
break;
case 2:
val = delete_element();
if(val!=–1)
printf("\n The number deleted is : %d", val);
break;
case 3:
val = peek();
if(val!=–1)
printf("\n The first value in queue is : %d", val);
break;
case 4:
display();
break;
}
}while(option!=5);
getch();
return 0;
}
void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(front==0 && rear==MAX–1)
printf("\n OVERFLOW");
else if(front==–1 && rear==–1)
{
```

```
front=rear=0;
queue[rear]=num;
}
else if(rear==MAX–1 && front!=0)
{
rear=0;
queue[rear]=num;
}
else
{
rear++;
queue[rear]=num;
}
}
int delete_element()
{
int val;
if(front==–1 && rear==–1)
{
printf("\n UNDERFLOW");
return –1;
}
val = queue[front];
if(front==rear)
front=rear=–1;
else
{
if(front==MAX–1)
front=0;
else
front++;
```

```
}
return val;
}
int peek()
{
if(front==–1 && rear==–1)
{
printf("\n QUEUE IS EMPTY");
return –1;
else
{
return queue[front];
}
}
void display()
{
int i;
printf("\n");
if (front ==–1 && rear= =–1)
printf ("\n QUEUE IS EMPTY");
else
{
if(front<rear)
{
for(i=front;i<=rear;i++)
printf("\t %d", queue[i]);
}
else
{
for(i=front;i<MAX;i++)
printf("\t %d", queue[i]);
```

```
for(i=0;i<=rear;i++)
printf("\t %d", queue[i]);
}
}
}
```

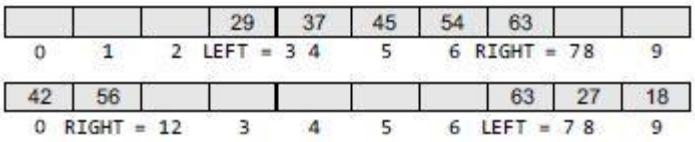
Output

```
***** MAIN MENU *****
1. Insert an element
2. Delete an element
3. Peek
4. Display the queue
5. EXIT
Enter your option : 1
Enter the number to be inserted in the queue : 25
Enter your option : 2
The number deleted is : 25
Enter your option : 3
QUEUE IS EMPTY
Enter your option : 5
```

2.<u>DEQUES:</u>

A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end. It is also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0].



DOUBLE END QUEUES

There are two variants of a double-ended queue. They include:

- **Input restricted deque:** In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
- **Output restricted deque:** In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

PROGRAMMING EXAMPLE:

4. Write a program to implement input and output restricted deques.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int deque[MAX];
int left = –1, right = –1;
void input_deque(void);
void output_deque(void);
void insert_left(void);
void insert_right(void);
void delete_left(void);
void delete_right(void);
void display(void);
int main()
{
int option;
clrscr();
printf("\n *****MAIN MENU*****");
printf("\n 1.Input restricted deque");
printf("\n 2.Output restricted deque");
printf("Enter your option : ");
```

```
scanf("%d",&option);
switch(option)
{
case 1:
input_deque();
break;
case 2:
output_deque();
break;
}
return 0;
}
void input_deque()
{
int option;
do
{
printf("\n INPUT RESTRICTED DEQUE");
printf("\n 1.Insert at right");
printf("\n 2.Delete from left");
printf("\n 3.Delete from right");
printf("\n 4.Display");
printf("\n 5.Quit");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1:
insert_right();
break;
case 2:
```

```
delete_left();
break;
case 3:
delete_right();
break;
case 4:
display();
break;
}
}while(option!=5);
}
void output_deque()
{
int option;
do
{
printf("OUTPUT RESTRICTED DEQUE");
printf("\n 1.Insert at right");
printf("\n 2.Insert at left");
printf("\n 3.Delete from left");
printf("\n 4.Display");
printf("\n 5.Quit");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1:
insert_right();
break;
case 2:
insert_left();
```

```
break;
case 3:
delete_left();
break;
case 4:
display();
break;
}
}while(option!=5);
}
void insert_right()
{
int val;
printf("\n Enter the value to be added:");
scanf("%d", &val);
if((left == 0 && right == MAX–1) || (left == right+1))
{
printf("\n OVERFLOW");
return;
}
if (left == –1) /* if queue is initially empty */
{
left = 0;
right = 0;
}
else
{
if(right == MAX–1) /*right is at last position of queue */
right = 0;
else
right = right+1;
```

```
}
deque[right] = val ;
}
void insert_left()
{
int val;
printf("\n Enter the value to be added:");
scanf("%d", &val);
if((left == 0 && right == MAX–1) || (left == right+1))
{
printf("\n Overflow");
return;
}
if (left == –1)/*If queue is initially empty*/
{
left = 0;
right = 0;
}
else
{
if(left == 0)
left=MAX–1;
else
left=left–1;
}
deque[left] = val;
}
void delete_left()
{
if (left == –1)
{
```

```
printf("\n UNDERFLOW");
return ;
}
printf("\n The deleted element is : %d", deque[left]);
if(left == right) /*Queue has only one element */
{
left = –1;
right = –1;
}
else
{
if(left == MAX–1)
left = 0;
else
left = left+1;
}
}
void delete_right()
{
if (left == –1)
{
printf("\n UNDERFLOW");
return ;
}
printf("\n The element deleted is : %d", deque[right]);
if(left == right) /*queue has only one element*/
{
left = –1;
right = –1;
}
else
```

```
{
if(right == 0)
right=MAX–1;
else
right=right–1;
}
}
void display()
{
int front = left, rear = right;
if(front == –1)
{
printf("\n QUEUE IS EMPTY");
return;
}
printf("\n The elements of the queue are : ");
if(front <= rear )
{
while(front <= rear)
{
printf("%d",deque[front]);
front++;
}
}
else
{
while(front <= MAX–1)
{
printf("%d", deque[front]);
front++;
}
```

```
front = 0;
while(front <= rear)
{
printf("%d",deque[front]);
front++;
}
}
printf("\n");
}
```

Output

```
***** MAIN MENU *****
1.Input restricted deque
2.Output restricted deque
Enter your option : 1
INPUT RESTRICTED DEQUEUE
1.Insert at right
2.Delete from left
3.Delete from right
4.Display
5.Quit
Enter your option : 1
Enter the value to be added : 5
Enter the value to be added : 10
Enter your option : 2
The deleted element is : 5
Enter your option : 5
```

3.PRIORITY QUEUES:

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

- An element with higher priority is processed before an element with a lower priority.
- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors.

Linked Representation of a Priority Queue

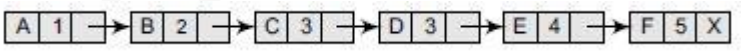
In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts:

(a) the information or data part,

(b) the priority number of the element, and

(c) the address of the next element.

If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority.



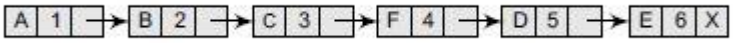
PRIORITY QUEUE

- Lower priority number means higher priority
- The priority queue in is a sorted priority queue having six elements.
- From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before because the list is not sorted based on FCFS.
- Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

Insertion: When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element.

PRIOIRTY QUEUE

If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element.



Priority Queue After Insertion A New Node

However, if we have a new element with data = F and priority number = 2, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS.

Priority Queue After Insertion A New Node

Deletion: Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

Array Representation of a Priority Queue

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed as.

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

PRIORITY QUEUE MATRIX

- FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number.Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

Insertion To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if w have to insert an element R with priority number 3,

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Priority Queue Matrix After Insertion Of A New Element

Deletion To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

PROGRAMMING EXAMPLE:

5. Write a program to implement a priority queue.

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
struct node
{
int data;
int priority;
```

```
struct node *next;
}
struct node *start=NULL;
struct node *insert(struct node *);
struct node *delete(struct node *);
void display(struct node *);
int main()
{
int option;
clrscr();
do
{
printf("\n *****MAIN MENU*****);
printf("\n 1. INSERT");
printf("\n 2. DELETE");
printf("\n 3. DISPLAY");
printf("\n 4. EXIT");
printf("\n Enter your option : ");
scanf( "%d", &option);
switch(option)
{
case 1:
start=insert(start);
break;
case 2:
start = delete(start);
break;
case 3:
display(start);
break;
}
```

```
}while(option!=4);
}
struct node *insert(struct node *start)
{
int val, pri;
struct node *ptr, *p;
ptr = (struct node *)malloc(sizeof(struct node));
printf("\n Enter the value and its priority : " );
scanf( "%d %d", &val, &pri);
ptr–>data = val;
ptr–>priority = pri;
if(start==NULL || pri < start–>priority )
{
ptr–>next = start;
start = ptr;
}
else
{
p = start;
while(p–>next != NULL && p–>next–>priority <= pri)
p = p–>next;
ptr–>next = p–>next;
p–>next = ptr;
}
return start;
}
struct node *delete(struct node *start)
{
struct node *ptr;
if(start == NULL)
{
```

```
printf("\n UNDERFLOW" );
return;
}
Else
{
ptr = start;
printf("\n Deleted item is: %d", ptr–>data);
start = start–>next;
free(ptr);
}
return start;
}
void display(struct node *start)
{
struct node *ptr;
ptr = start;
if(start == NULL)
printf("\nQUEUE IS EMPTY" );
else
{
printf("\n PRIORITY QUEUE IS : " );
while(ptr != NULL)
{
printf( "\t%d[priority=%d]", ptr–>data, ptr–>priority );
ptr=ptr–>next;
}
}
}
```

Output

*****MAIN MENU*****

1. INSERT

2. DELETE

3. DISPLAY

4. EXIT

Enter your option : 1

Enter the value and its priority : 5 2

Enter the value and its priority : 10 1

Enter your option : 3

PRIORITY QUEUE IS :

10[priority = 1] 5[priority = 2]

Enter your option : 4

4.MULTIPLE QUEUES:

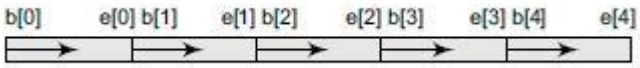
When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.

- In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.

MULTIPLE QUEUES

In the figure, an array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing—queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i].



MULTIPLE QUEUES

PROGRAMMING EXAMPLE:

6. Write a program to implement multiple queues.

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int QUEUE[MAX], rearA=–1,frontA=–1, rearB=MAX, frontB = MAX;
void insertA(int val)
{
if(rearA==rearB –1)
printf("\n OVERFLOW");
else
{
if(rearA ==–1 && frontA == –1)
{ rearA = frontA = 0;
QUEUE[rearA] = val;
}
else
QUEUE[++rearA] = val;
}
}
int deleteA()
{
int val;
if(frontA==–1)
{
```

```
printf("\n UNDERFLOW");
return –1;
}
else
{
val = QUEUE[frontA];
frontA++;
if (frontA>rearA)
frontA=rearA=–1
return val;
}
}
void display_queueA()
{
int i;
if(frontA==–1)
printf("\n QUEUE A IS EMPTY");
else
{
for(i=frontA;i<=rearA;i++)
printf("\t %d",QUEUE[i]);
}
}
void insertB(int val)
{
if(rearA==rearB–1)
printf("\n OVERFLOW");
else
{
if(rearB == MAX && frontB == MAX)
{ rearB = frontB = MAX–1;
```

```
QUEUE[rearB] = val;
}
else
QUEUE[—rearB] = val;
}
}
int deleteB()
{
int val;
if(frontB==MAX)
{
printf("\n UNDERFLOW");
return –1;
}
else
{
val = QUEUE[frontB];
frontB—;
if (frontB<rearB)
frontB=rearB=MAX;
return val;
}
}
void display_queueB()
{
int i;
if(frontB==MAX)
printf("\n QUEUE B IS EMPTY");
else
{
for(i=frontB;i>=rearB;i—)
```

```
printf("\t %d",QUEUE[i]);
}
}
int main()
{
int option, val;
clrscr();
do
{
printf("\n *******MENU******");
printf("\n 1. INSERT IN QUEUE A");
printf("\n 2. INSERT IN QUEUE B");
printf("\n 3. DELETE FROM QUEUE A");
printf("\n 4. DELETE FROM QUEUE B");
printf("\n 5. DISPLAY QUEUE A");
printf("\n 6. DISPLAY QUEUE B");
printf("\n 7. EXIT");
printf("\n Enter your option : ");
scanf("%d",&option);
switch(option)
{
case 1: printf("\n Enter the value to be inserted in Queue A : ");
scanf("%d",&val);
insertA(val);
break;
case 2: printf("\n Enter the value to be inserted in Queue B : ");
scanf("%d",&val);
insertB(val);
break;
case 3: val=deleteA();
if(val!=–1)
```

```
printf("\n The value deleted from Queue A = %d",val);
break;
case 4 : val=deleteB();
if(val!=–1)
printf("\n The value deleted from Queue B = %d",val);
break;
case 5: printf("\n The contents of Queue A are : \n");
display_queueA();
break;
case 6: printf("\n The contents of Queue B are : \n");
display_queueB();
break;
}
}while(option!=7);
getch();
}
```

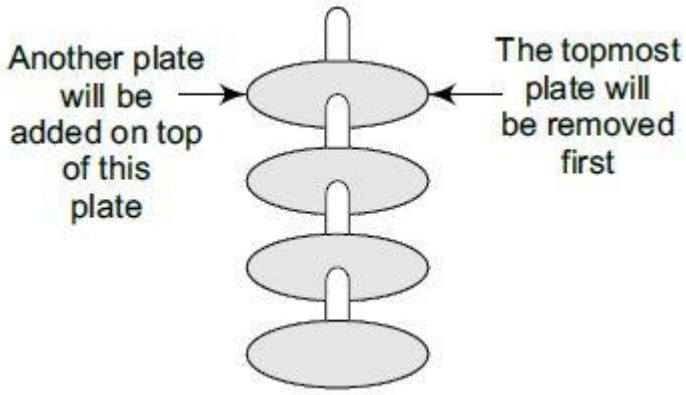
Output

```
*******MENU******"
1. INSERT IN QUEUE A
2. INSERT IN QUEUE B
3. DELETE FROM QUEUE A
4. DELETE FROM QUEUE B
5. DISPLAY QUEUE A
6. DISPLAY QUEUE B
7. EXIT
Enter your option : 2
Enter the value to be inserted in Queue B : 10
Enter the value to be inserted in Queue B : 5
Enter your option: 6
The contents of Queue B are : 10 5
Enter your option : 7
```

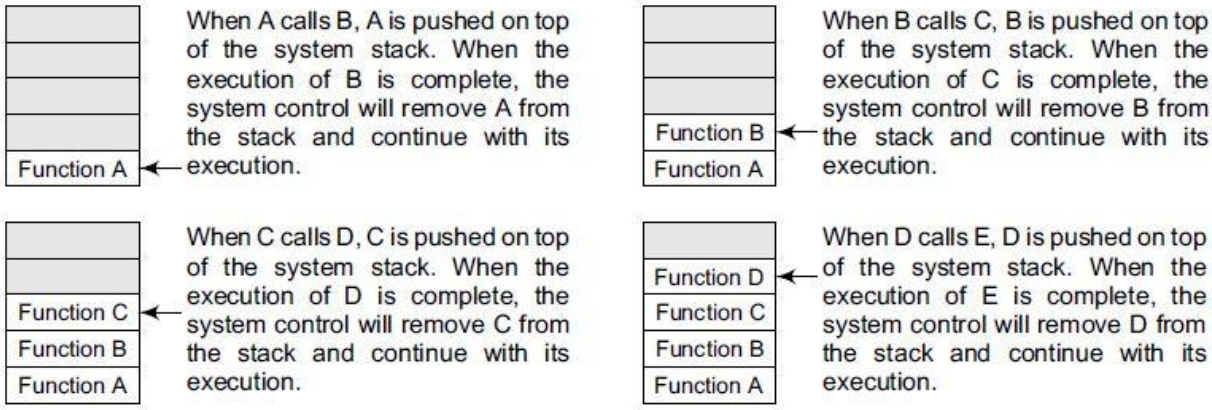
STACKS

1.INTRODUCTION TO STACKS

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.



STACK OF PLATES



SYSTEM STACK IN CASE OF FUNCTION CALL

2.ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.

(You must be wondering why we have written MAX–1. It is because array indices start from 0.)

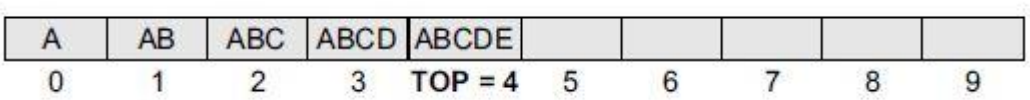


FIG. STACK

The stack in Fig. shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

OPERATIONS ON STACK:

A stack supports three basic operations: push, pop, and peek. The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack. The peek operation returns the value of the topmost element of the stack.

1. **PUSH OPERATION:**
   - The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
   - However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

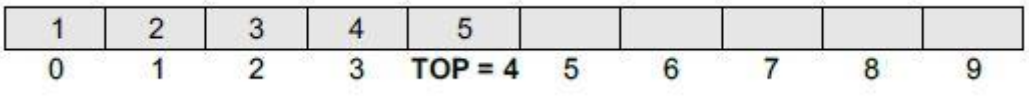


FIG. STACK

- To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as shown in Fig.

FIG. STACK AFTER INSERTION

ALGORITHM TO INSERT AN ELEMENT IN STACK:

Step 1: IF TOP = MAX-1

PRINT OVERFLOW

Goto Step 4

[END OF IF]

Step 2: SET TOP = TOP + 1

Step 3: SET STACK [TOP] = VALUE

Step 4: END

2. POP OPERATION:

- The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

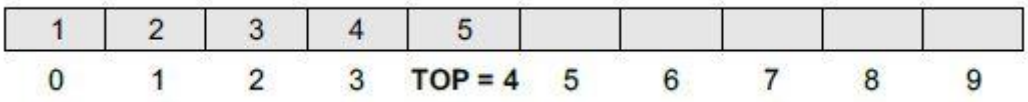


FIG. STACK

- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig.

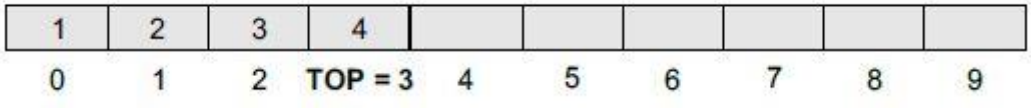


FIG. STACK AFTER FELETION

ALGORITHM TO DELETE AN ELEMENT FROM A STACK:

Step 1: IF TOP = NULL

PRINT UNDERFLOW

GOTO STEP 4

[END OF IF]

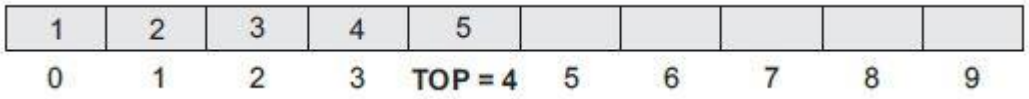
Step 2: SET VAL = STACK [TOP]

Step 3: SET TOP = TOP - 1

Step 4: END

3. PEEK OPERATION:

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.



STACK

- Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

ALGORITHM FOR PEEK OPERATION:

Step 1: IF TOP = NULL

PRINT STACK IS EMPTY

Goto Step 3

Step 2: RETURN STACK [TOP]

Step 3: END

PROGRAMMING EXAMPLE:

1. Write a program to perform Push, Pop, and Peek operations on a stack.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define MAX 3 // Altering this value changes size of stack created
int st[MAX], top=-1;
void push(int st[], int val);
int pop(int st[]);
int peek(int st[]);
void display(int st[]);
int main(int argc, char *argv[]) {
int val, option;
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option: ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to be pushed on stack: ");
scanf("%d", &val);
push(st, val);
break;
case 2:
```

```
val = pop(st);
if(val != -1)
printf("\n The value deleted from stack is: %d", val);
break;
case 3:
val = peek(st);
if(val != -1)
printf("\n The value stored at top of stack is: %d", val);
break;
case 4:
display(st);
break;
}
}while(option != 5);
return 0;
}
void push(int st[], int val)
{
if(top == MAX-1)
{
printf("\n STACK OVERFLOW");
}
else
{
top++;
st[top] = val;
}
}
int pop(int st[])
{
int val;
```

```
if(top == -1)
{
printf("\n STACK UNDERFLOW");
return -1;
}
else
{
val = st[top];
top--;
return val;
}
}
void display(int st[])
{
int i;
if(top == -1)
printf("\n STACK IS EMPTY");
else
{
for(i=top;i>=0;i--)
printf("\n %d",st[i]);
printf("\n"); // Added for formatting purposes
}
}
int peek(int st[])
{
if(top == -1)
{
printf("\n STACK IS EMPTY");
return -1;
}
```

```
else
return (st[top]);
}
```

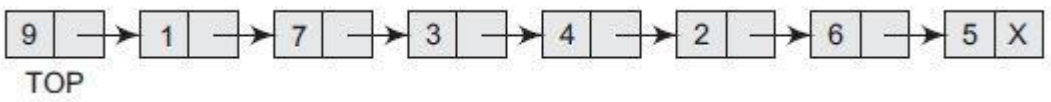
Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. PEEK
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 500
```

3.LINKED REPRESENTATION OF STACKS

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.

- But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

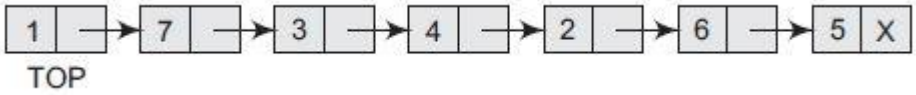


LINKED STACK

OPERATIONS ON A LINKED STACK:

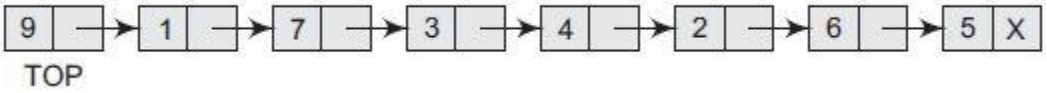
A linked stack supports all the three stack operations, that is, push, pop, and peek.

1. **PUSH OPERATION:**
   - The push operation is used to insert an element into the stack. The new element is added a topmost position of the stack.



LINKED STACK

   - To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP.



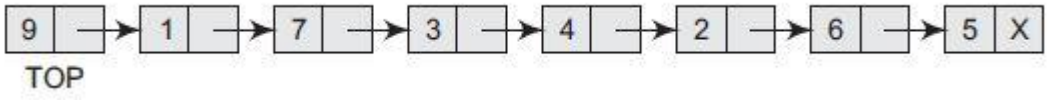
LINKED STACK AFTER INSERTING A NEW NODE

ALGORITHM TO INSERT AN ELEMENT IN A LINKED STACK:

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

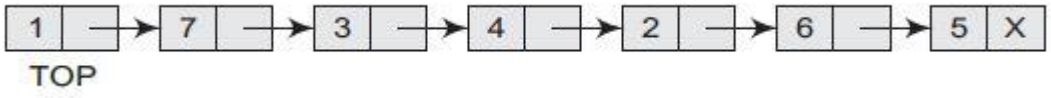
2. **POP OPERATION:**

- The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.



LINKED STACK

- In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack.



LINKED STACK AFTER DELETION OF THE TOPMOST ELEMENT

ALGORITHM TO DELETE AN ELEMENT FROM A LINKED LIST:

Step 1: IF TOP = NULL
PRINT UNDERFLOW
Goto Step 5
[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP -> NEXT

Step 4: FREE PTR

Step 5: END

PROGRAMMING EXAMPLE:

2. Write a program to implement a linked stack.

```
##include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
#include <malloc.h>
struct stack
{
int data;
struct stack *next;
};
struct stack *top = NULL;
struct stack *push(struct stack *, int);
struct stack *display(struct stack *);
struct stack *pop(struct stack *);
int peek(struct stack *);
int main(int argc, char *argv[]) {
int val, option;
do
{
printf("\n *****MAIN MENU*****");
printf("\n 1. PUSH");
printf("\n 2. POP");
printf("\n 3. PEEK");
printf("\n 4. DISPLAY");
printf("\n 5. EXIT");
printf("\n Enter your option: ");
scanf("%d", &option);
switch(option)
{
case 1:
printf("\n Enter the number to be pushed on stack: ");
scanf("%d", &val);
top = push(top, val);
break;
case 2:
```

```
top = pop(top);
break;
case 3:
val = peek(top);
if (val != -1)
printf("\n The value at the top of stack is: %d", val);
else
printf("\n STACK IS EMPTY");
break;
case 4:
top = display(top);
break;
}
}while(option != 5);
return 0;
}
struct stack *push(struct stack *top, int val)
{
struct stack *ptr;
ptr = (struct stack*)malloc(sizeof(struct stack));
ptr -> data = val;
if(top == NULL)
{
ptr -> next = NULL;
top = ptr;
}
else
{
ptr -> next = top;
top = ptr;
}
```

```
return top;
}
struct stack *display(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK IS EMPTY");
else
{
while(ptr != NULL)
{
printf("\n %d", ptr -> data);
ptr = ptr -> next;
}
}
return top;
}
struct stack *pop(struct stack *top)
{
struct stack *ptr;
ptr = top;
if(top == NULL)
printf("\n STACK UNDERFLOW");
else
{
top = top -> next;
printf("\n The value being deleted is: %d", ptr -> data);
free(ptr);
}
return top;
```

```
}
int peek(struct stack *top)
{
if(top==NULL)
return -1;
else
return top ->data;
}
```

Output

```
*****MAIN MENU*****
1. PUSH
2. POP
3. Peek
4. DISPLAY
5. EXIT
Enter your option : 1
Enter the number to be pushed on stack : 100
```

4.APPLICATIONS OF STACKS

1. REVERSING A LIST:

- A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

PROGRAMMING EXAMPLE:

3. Write a program to reverse a list of given numbers.

```
#include <stdio.h>
#include <conio.h>
int stk[10];
int top=–1;
int pop();
```

```
void push(int);
int main()
{
int val, n, i,
arr[10];
clrscr();
printf("\n Enter the number of elements in the array : ");
scanf("%d", &n);
printf("\n Enter the elements of the array : ");
for(i=0;i<n;i++)
scanf("%d", &arr[i]);
for(i=0;i<n;i++)
push(arr[i]);
for(i=0;i<n;i++)
{
val = pop();
arr[i] = val;
}
printf("\n The reversed array is : ");
for(i=0;i<n;i++)
printf("\n %d", arr[i]);
getche"();
return 0;
}
void push(int val)
{
stk[++top] = val;
}
int pop()
{
return(stk[top—]);
```

}

Output

Enter the number of elements in the array : 5

Enter the elements of the array : 1 2 3 4 5

The reversed array is : 5 4 3 2 1

2.CONVERTING INFIX TO POSTFIX EXPRESSION:

- Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators. The precedence of these operators can be given as follows:
    a. Higher priority *, /, %
    b. Lower priority +, –
- No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

EXAMPLE: Convert the following infix expressions into prefix expressions.

Solution

(a) (A + B) * C

(+AB)*C

*+ABC

(b) (A–B) * (C+D)

[–AB] * [+CD]

*–AB+CD

(c) (A + B) / (C + D) – (D * E)

[+AB] / [+CD] – [*DE]

[/+AB+CD] – [*DE]

–/+AB+CD*DE

ALGORITHM TO CONVERT INFIX EXPRESSION TO POSTFIX EXPRESSION

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
       IF a "(" is encountered, push it on the stack
       IF an operand (whether a digit or a character) is encountered, add it to the
       postfix expression.
       IF a ")" is encountered, then
         a. Repeatedly pop from stack and add it to the postfix expression until a
            "(" is encountered.
         b. Discard the "(". That is, remove the "(" from stack and do not
            add it to the postfix expression
       IF an operator O is encountered, then
         a. Repeatedly pop from stack and add each operator (popped from the stack) to the
            postfix expression which has the same precedence or a higher precedence than O
         b. Push the operator O to the stack
       [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

EXAMPLE: Convert the following infix expression into postfix expression using the algorithm given.

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)

SOLUTION:

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
–	(–	A
(	(– (	A
B	(– (	A B
/	(– (/	A B
C	(– (/	A B C
+	(– (+	A B C /
(	(– (+ (	A B C /
D	(– (+ (	A B C / D
%	(– (+ (%	A B C / D
E	(– (+ (%	A B C / D E
*	(– (+ (% *	A B C / D E
F	(– (+ (% *	A B C / D E F
)	(– (+	A B C / D E F * %
/	(– (+ /	A B C / D E F * %
G	(– (+ /	A B C / D E F * % G
)	(–	A B C / D E F * % G / +
*	(– *	A B C / D E F * % G / +
H	(– *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * –

PROGRAMMING EXAMPLE:

4.Write a program to convert an infix expression into its equivalent postfix notation.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char st[MAX];
int top=–1;
void push(char st[], char);
char pop(char st[]);
void InfixtoPostfix(char source[], char target[]);
int getPriority(char);
int main()
{
char infix[100], postfix[100];
clrscr();
printf("\n Enter any infix expression : ");
gets(infix);
strcpy(postfix, "");
InfixtoPostfix(infix, postfix);
printf("\n The corresponding postfix expression is : ");
puts(postfix);
getch();
return 0;
}
void InfixtoPostfix(char source[], char target[])
{
int i=0, j=0;
char temp;
strcpy(target, "");
```

```
while(source[i]!='\0')
{
if(source[i]=='(')
{
push(st, source[i]);
i++;
}
else if(source[i] == ')')
{
while((top!=–1) && (st[top]!='('))
{
target[j] = pop(st);
j++;
}
if(top==–1)
{
printf("\n INCORRECT EXPRESSION");
exit(1);
}
temp = pop(st);//remove left parenthesis
i++;
}
else if(isdigit(source[i]) || isalpha(source[i]))
{
target[j] = source[i];
j++;
i++;
}
else if (source[i] == '+' || source[i] == '–' || source[i] == '*' ||
source[i] == '/' || source[i] == '%')
{
```

```
while( (top!=–1) && (st[top]!= '(') && (getPriority(st[top])
> getPriority(source[i])))
{
target[j] = pop(st);
j++;
}
push(st, source[i]);
i++;
}
Else
{
printf("\n INCORRECT ELEMENT IN EXPRESSION");
exit(1);
}
}
while((top!=–1) && (st[top]!='('))
{
target[j] = pop(st);
j++;
}
target[j]='\0';
}
int getPriority(char op)
{
if(op=='/' || op == '*' || op=='%')
return 1;
else if(op=='+' || op=='–')
return 0;
}
void push(char st[], char val)
{
```

```
if(top==MAX–1)
printf("\n STACK OVERFLOW");
else
{
top++;
st[top]=val;
}
}
char pop(char st[])
{
char val=' ';
if(top==–1)
printf("\n STACK UNDERFLOW");
else
{
val=st[top];
top—;
}
return val;
}
```

Output

Enter any infix expression : A+B–C*D

The corresponding postfix expression is : AB+CD*–

3. EVALUATION OF POSTFIX EXPRESSION:

- Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.
- Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack.

- However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

ALGORITHM TO EVALUATE POSTFIX EXPRESSION:

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

EXAMPLE:

Let us now take an example that makes use of this algorithm. Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluate the expression.

The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation. Look at Table which shows the procedure.

Evaluation of a postfix expression.

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

PROGRAMMING EXAMPLE:

5.Write a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=–1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
float val;
char exp[100];
clrscr();
printf("\n Enter any postfix expression : ");
```

```
gets(exp);
val = evaluatePostfixExp(exp);
printf("\n Value of the postfix expression = %.2f", val);
getch();
return 0;
}
float evaluatePostfixExp(char exp[])
{
int i=0;
float op1, op2, value;
while(exp[i] != '\0')
{
if(isdigit(exp[i]))
push(st, (float)(exp[i]–'0'));
else
{
op2 = pop(st);
op1 = pop(st);
switch(exp[i])
{
case '+':
value = op1 + op2;
break;
case '–':
value = op1 – op2;
break;
case '/':
value = op1 / op2;
break;
case '*':
value = op1 * op2;
```

```
break;
case '%':
value = (int)op1 % (int)op2;
break;
}
push(st, value);
}
i++;
}
return(pop(st));
}
void push(float st[], float val)
{
if(top==MAX–1)
printf("\n STACK OVERFLOW");
else
{
top++;
st[top]=val;
}
}
float pop(float st[])
{
float val=–1;
if(top==–1)
printf("\n STACK UNDERFLOW");
else
{
val=st[top];
top—;
}
```

return val;

}

Output

Enter any postfix expression : 9 3 4 * 8 + 4 / –

Value of the postfix expression = 4.00

4. FACTORIAL CALCULATION:

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function. Every recursive solution has two major cases. They are

Base case, in which the problem is simple enough to be solved directly without making any further calls to the same function.

Recursive case, in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step. Third, the result is obtained by combining the solutions of simpler sub-parts.

Therefore, recursion is defining large and complex problems in terms of smaller and more easily solvable problems. In recursive functions, a complex problem is defined in terms of simpler problems and the simplest problem is given explicitly.

To understand recursive functions, let us take an example of calculating factorial of a number.

To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.

In other words, n! = n X (n–1)!

Let us say we need to find the value of 5!

5! = 5 X 4 X 3 X 2 X 1

= 120

This can be written as

5! = 5 X 4!, where 4!= 4 X 3!

Therefore,

5! = 5 X 4 X 3!

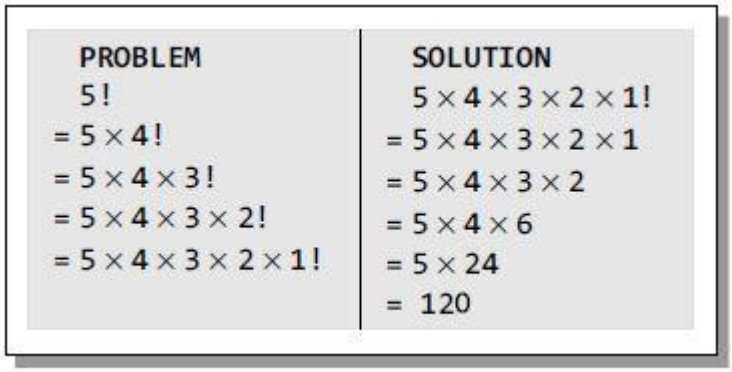
Similarly, we can also write,

5! = 5 X 4 X 3 X 2!

Expanding further

5! = 5 X 4 X 3 X 2 X 1!

We know, 1! = 1



RECURSIVE FACTORIAL FUNCTION

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. every recursive function must have a base case and a recursive case. For the factorial function,

Base case is when n = 1, because if n = 1, the result will be 1 as 1! = 1.

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as factorial(n) = n × factorial (n–1)

PROGRAMMING EXAMPLE:

6. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int); // FUNCTION DECLARATION
int main()
{
int num, val;
printf("\n Enter the number: ");
scanf("%d", &num);
```

```
val = Fact(num);
printf("\n Factorial of %d = %d", num, val);
return 0;
}
int Fact(int n)
{
if(n==1)
return 1;
else
return (n * Fact(n–1));
}
```

Output

```
Enter the number : 5
Factorial of 5 = 120
```