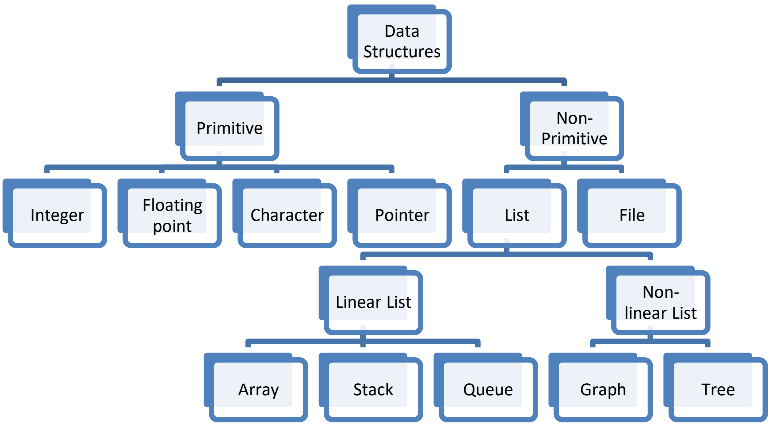
MODULE -1

Introduction to Data Structures: Basic Terminology – Classification - Operations on Data Structures, Linear Data Structures, Stacks: Introduction to Stacks - Array Representation of Stacks - Operations on a Stack - Applications of Stacks - Infix-to-Postfix Conversion - Evaluating Postfix Expressions, Queues: Introduction to Queues - Array Representation of Queues - Operations on a Queue - Types of Queues: Circular Queue – Dequeue – Priority queue.

Data Structures

Data Structure: is a way of organizing all data items that considers not only the elements stored but also their relationships to each other.

Classification of data structures



Linear Data structures: A data structure is said to be linear if its elements are connected in linear fashion in sequence of memory locations.

Non-Linear Data structures: Non-linear data structures are those data structures in which data items are not arranged in a sequence.

Difference between linear and non-linear data structures

Linear	**Non-Linear**
1. Every item is related to its previous and next item.	Every item is attached with many other items.
2. Data is arranged in linear sequence	Data is not arranged in a sequence.
3. Data items can be traversed in a single run.	Data cannot be traversed in a single run
4. Eg: Array, stack, queue	Eg: Tree and Graph
5. Implementation is easy	Implementation is difficult

ARRAY

Array is a homogeneous collection of elements arranged in a sequential manner. The basic operations of array are:

1. Traversal (accessing): Accessing elements in the array.
2. Insertion: Inserting a new element at a specified position.
3. Deletion : Deleting an element from a specific position or deleting a specified element.
4. Searching: Searching for an element in the array.
5. Sorting: Arranging the elements in ascending/descending order.

PROGRAM

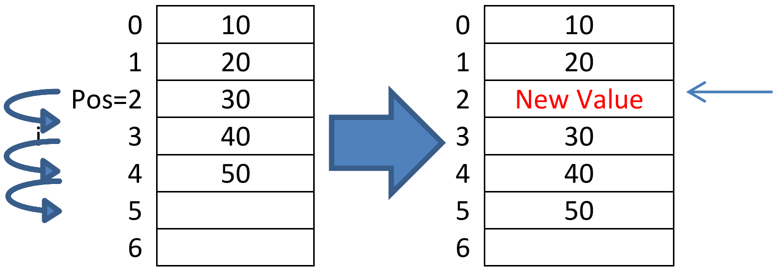
```
#include<stdio.h>
#include<stdlib.h>
int a[20],n;
void read_array();
void traverse_array();
void insertion();
void deletion;
void searching();
void sorting();

    int main()
    {
    int op;
    read_array();
    printf(“\nMenu: 1.Traversal 2.Insertion 3.Deletion 4.Searching 5. Sorting 6. Exit");
    do
    {
    printf(“\nEnter the option:");
    scanf(“%d”,&op);
    switch(op)
    {
    case 1:
    traverse_array();
    break;
    case 2:
    insertion();
    break;
    case 3:
    deletion();
    break;
    case 4:
    searching();
    break;
    case 5:
    sorting();
    break;
    case 6:
    exit(0);
default:
printf("Invalid choice";
}
}while(1);
return 0;
}
```

1. **Traversal**

```
void traverse_array()
{
int i;
printf("\nThe elements in the array are:");
for(i=0;i<n;i++)
printf("%d ",a[i]);
}
```

2. **Insertion**

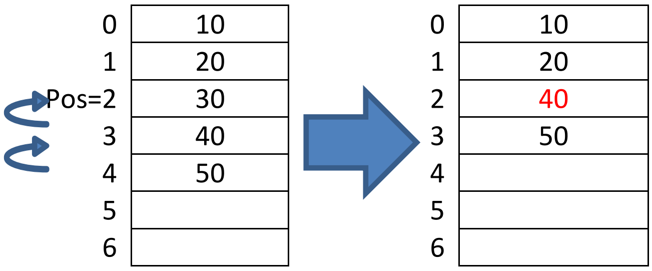
Here all elements below the specified position are shifted down one position before inserting the new element into the specified position. Assume that we want to insert the new element at position 2.



```
void insertion()
{
int pos, val,i;
printf("\nEnter the position :");
scanf("%d",&pos);
printf("\n Enter the value to be inserted:");
scanf("%d",&val);

for(i=n-1;i>=pos;i--)
{
a[i+1]=a[i];
}
a[pos]=val;
n++;
}
```

3. Deleting an element from a specified position

Here all elements below the specified position are shifted upwards by one position. Assume that we want to delete the element at position 2.



0	10
1	20
Pos=2	30
3	40
4	50
5	
6	

0	10
1	20
2	40
3	50
4	
5	
6	

```
void deletion()
{
int val,pos,i;
printf("\nEnter the position from which the element is to be deleted:");
scanf("%d",&pos);
val=a[pos];
for(i=pos;i<=n-2;i++)
{
a[i]=a[i+1];
}
n--;
printf("\nThe deleted element is: %d",val);
}
```

4. Searching

If the required element is present in the array, the position is returned.

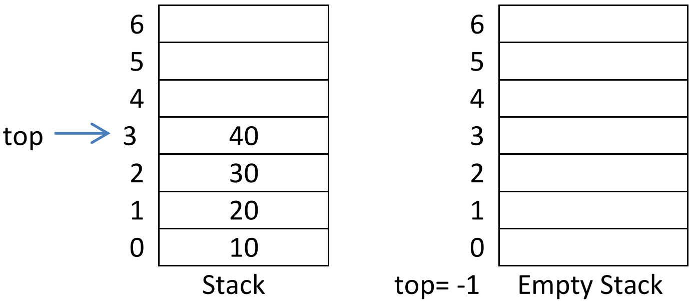
```
void searching()
{
int val,i;
printf("\nEnter the value to be searched");
scanf("%d",&val);
for(i=0;i<n;i++)
{
if(a[i]==val)
break;
}
if(i==n)
printf("\nThe element is not present in the array");
else
printf("\nThe element is present at position : %d",i);
}

void sorting()
{
int i,j,t;
for(i=0;i<n-1;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
t=a[i];
a[i]=a[j];
a[j]=t;
}
}
}
printf("\nThe elements after sorting:");
traverse_array();
}
```

STACK

Stack is a linear list in which elements are arranged in sequential manner. It is a special type of array in which insertion and deletion operations can only be done at one end of the array. That end is called **top** of the stack.

The insertion operation in a stack is called **push** and the deletion operation is called **pop.**

In case of empty stack the value of top is equal to -1, otherwise it will point to the top element of the stack.



The stack is also called **LIFO List (Last In First Out List)** means the last inserted element will be the first to be removed from the stack.

The basic operations in a stack are

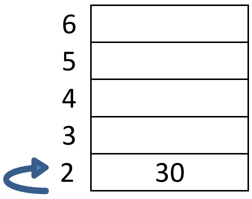
1. Push, 2. Pop and 3. Traversal

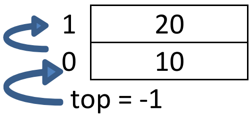
PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 5;
int S[20], top=-1;
    void push();
    void pop();
    void traversal();
    int main()
    {
    int op;
    printf(“\nMenu: 1.Push 2.Pop 3.Traversal 4. Exit");
    do
    {
    printf(“\nEnter the option:");
    scanf(“%d”,&op);
    switch(op)
    {
    case 1:
    push();
    break;
    case 2:
    pop();
    break;
    case 3:
    traversal();
    break;
    case 4:
    exit(0);
    default:
    printf("Invalid choice");
    }
    }while(1);
    return 0;
    }
```

1. PUSH Operation

In Push operation the top pointer is incremented by one and the new element is inserted at that position.



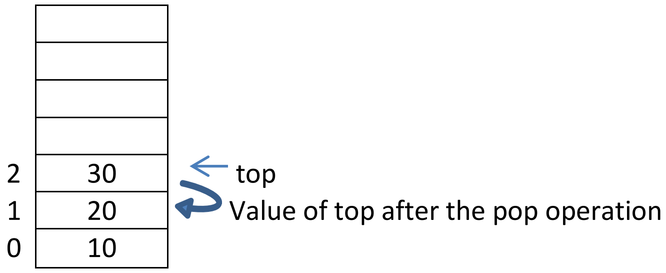


Before insertion we have to check whether the value of top=MAXSIZE-1, where MAXSIZE is the maximum size of the stack. If so, then the stack is full and no new element can be inserted into the stack. Such condition is called **stack overflow**.

```
void push()
{
int val;
if(top==MAX-1)
printf("\nStack Overflow");
else
{
printf("\nEnter the value to be inserted: ");
scanf("%d",&val);
top++;
s[top]=val;
}
}
```

2. **POP Operation**

   In Pop operation we just decrement the value of top by one. That means after removing the top most element, the top will point to the next below element. Before performing pop operation we have to check whether stack is empty. This will occur when the value of top=-1 and such condition is called **stack underflow.**



```
void pop()
{
int val;
if(top==-1)
printf("\nStack is empty");
else
{
val=s[top];
top--;
printf("\n The deleted element is: %d",val);
}
}
```

3. **Traversal Operation**

```
void traversal()
{
int i;
if(top==-1)
printf("\n stack is empty");
else
{
printf("\nThe elements are: ");
for(i=0;i<=top;i++)
{
printf("%d ",s[i]);
}
}
}
```

APPLICATIONS OF STACK

Some applications of stack are:

1. Conversion of an infix expression into its corresponding postfix form.
2. Evaluation of arithmetic expressions.
3. Recursion.
4. Parenthesis checker.
5. Reversing a list.

ARITHMETIC EXPRESSIONS

Infix, postfix and prefix notations are three different but equivalent notations of writing algebraic expressions.

Infix : In infix expression, the operator is placed in between the operands. For example, in a+b the operator + is placed between the two operands a and b.

Postfix (Polish) : In postfix or polish expression, the operator is placed after the operands. It is parenthesis free. For example,

Example 1: A+B can be written as AB+

Example 2: A+B *C can be written as
⇨ A+(BC*)
⇨ ABC*+

Example 3: (A+B) *C can be written as
⇨ (AB+) * C
⇨ AB+C*

Example 4: (A+B) – (C+D) can be written as
⇨ (AB+) – (CD+)

⇨ AB+ CD+ -

Example 5: (A-B) * (C+D) can be written as
⇨ (AB-) * (CD+)
⇨ AB- CD+ *

Example 6: (A+B) / (C+D) – (D*E) can be written as
⇨ (AB+) / (CD+) – (DE*)
⇨ (AB+ CD+ /) – (DE*)
⇨ AB+ CD+ /DE* -

Example 7: A – (B/C + (D/E * F) /G) *H can be written as
⇨ A – ((BC/) + ((DE/) * F)/G) *H
⇨ A-((BC/) + ((DE/ F*)/G)) *H
⇨ A-((BC/) + (DE/F*G/)) *H
⇨ A-(BC/DE/F*G/+)*H
⇨ A-(BC/DE/F*G/+H*)
⇨ ABC/DE/F*G/+H*-

Prefix (Reverse Polish) : In prefix or reverse polish expression, the operator is placed before the operands. It is also parenthesis free. For example,

Example 1: A+B can be written as +AB

Example 2: A+B *C can be written as
⇨ A+(*BC)
⇨ +A*BC

Example 3: (A+B) *C can be written as
⇨ (+AB) * C
⇨ *+ABC

Example 4: (A+B) – (C+D) can be written as
⇨ (+AB) – (+CD)
⇨ - +AB +CD

Example 5: (A-B) * (C+D) can be written as
⇨ (-AB) * (+CD)
⇨ * -AB +CD

Example 6: (A+B) / (C+D) – (D*E) can be written as
⇨ (+AB) / (+CD) – (*DE)
⇨ (/ +AB +CD) – (*DE)
⇨ - /+AB+CD*DE

Example 7: A – (B/C + (D/E * F) /G) *H can be written as
⇨ A – ((/BC) + ((/DE) * F)/G) *H
⇨ A-((/BC) + ((*/DE F)/G)) *H
⇨ A-((/BC) + (/*/DEFG)) *H
⇨ A-(+/BC/*/DEFG) *H
⇨ A-(*+/BC/*/DEFGH)
⇨ -A*+/BC/*/DEFGH

ALGORITHM TO CONVERT AN INFIX EXPRESSION INTO ITS CORRESPONDING POSTFIX FORM

Step 1 : Add a closing bracket) to the end of the given infix expression.
Step 2 : Push an open bracket (onto the stack.
Step 3 : Repeat the following steps until each character in the infix expression is scanned.

a) If an open bracket (is encountered, push it onto the stack.
b) If an operand (digit or alphabet) is encountered, add it to the postfix notation.
c) If a closing bracket) is encountered, then repeatedly pop from stack and store it to the postfix expression until an open bracket (is encountered and then remove the open bracket (from the stack.
d) If an operator 'op' is encountered, then repeatedly pop the operators which has the higher or equal priority than 'op' from stack ,add each operator poped from the stack to the postfix expression and then push the operator 'op' onto the stack.

Step 4 : Check whether the stack is empty. If it is not empty, the given expression is invalid.
Step 5 : Exit.

Example: Convert the infix expression A+B*(C/D + E/F) – (G+H)

After adding a closing bracket) to the end of the given infix expression, the expression becomes: A+B*(C/D + E/F) – (G+H))

Steps	Characters Scanned	Action performed	Stack	Postfix
1		Push (onto the stack	(	
2	A	Store A into the postfix expression	(	A
3	+	Push + onto the stack	(+	A
4	B	Store B into the postfix expression	(+	AB
5	*	Push * onto the stack	(+*	AB
6	(	Push (onto the stack	(+*(	AB
7	C	Store C into the postfix expression	(+*(	ABC
8	/	Push / onto the stack	(+*(/	ABC
9	D	Store D into the postfix expression	(+*(/	ABCD
10	+	Pop the higher priority operator / from the stack, store it into the postfix expression and push + onto the stack	(+*(+	ABCD/
11	E	Store E into the postfix expression	(+*(+	ABCD/E
12	/	Push / onto the stack	(+*(+/	ABCD/E
13	F	Store F into the postfix expression	(+*(+/	ABCD/EF
14	)	Pop / and + from the stack, store them into the postfix expression and pop (from the stack	(+*	ABCD/EF/+
15	-	Pop the higher and equal priority operators * and + from the stack, store it into the postfix expression and push - onto the stack	(-	ABCD/EF/+*+
16	(	Push (onto the stack	(-(	ABCD/EF/+*+
17	G	Store G into the postfix expression	(-(	ABCD/EF/+*+G
18	+	Push + onto the stack	(-(+	ABCD/EF/+

				*+G
19	H	Store H into the postfix expression	(-(+	ABCD/EF/+*+GH
20	)	Pop + from the stack, store it into the postfix expression and pop (from the stack	(-	ABCD/EF/+*+GH+
21	)	Pop - from the stack, store it into the postfix expression and pop (from the stack	Empty	ABCD/EF/+*GH+-

Program

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
char s[25],in[25],post[25];
int top=-1,MAX=25;
void convert();
int priority(char);
void push(char);
char pop();

int priority(char op)
{
switch(op)
{
case '(':
case ')':
return 1;
case '+':
case '-':
return 2;
case '*':
case '/':
return 3;
default:
printf("\ninvalid operator");
}
}
void push(char ch)
{
if(top==MAX-1)
{
printf("\nstack is full") ;
exit(0);
}
else
{
```

```
top++;
s[top]=ch;
}
}
char pop()
{
char ch;
if(top==-1)
{
printf("\nStack is empty. Given expression is invalid");
exit(0);
}
else
{
ch=s[top];
top--;
return ch;
}
}
void convert()
{
int len,i-=,j=0;
char ch;
len=strlen(in);
in[len]=')';
in[len+1]='\0';
push('(');
while(in[i]!='\0')
{
char ch=in[i];
if(ch=='(')
{
push(ch);
}
else if(ch==')')
{
char c=pop();
while(c!='(')
{
post[j]=c;
j++;
c=pop();
}
}
else if(isalpha(ch))
{
post[j]=ch;
j++;
}
else
{
int p1=priority(s[top]);
int p2=priority(ch);
while(p1>=p2)
{
char c=pop();
post[j]=c;
j++;
p1=priority(s[top]);
}
push(ch);
}
i++;
}
if(top!=-1)
{
printf("\n Given expression is invalid");
exit(0);
}
post[j]='\0';
}

int main()
{
printf("\nEnter the infix expression");
scanf(%s”,in);
convert();
printf("\n The postfix expression is: %s",post);
return 0;
}
```

EVALUATION OF POSTFIX EXPRESSION

Given algebraic expression written in infix notation, the computer first convert the expression into the equivalent postfix form and then evaluates the postfix expression. Both the tasks make use of stack as the primary tool.

Step1: Scan every character of the postfix expression and repeat step 2 until ‘\0’ is encountered.

Step2: If an alphabet is encountered push its value onto the stack. If an operator ‘op’ is encountered then

(a) Pop the top 2 elements from the stack as op2 and op1.

(b) Evaluate ‘op1 op op2’, where op2 is the topmost element and op1 is the element below op2.

(c) push the result of evaluation onto the stack.

Step 3: Set result = the topmost element of the stack.

Step 4: Exit.

Example

Suppose the infix expression is

a-((b*c)+d)/e

The corresponding postfix expression is

abc*d+e/-

Let a=9, b=3, c=4, d=8 and e=4

Characters Scanned	Action performed	Stack
a	Push value of a onto the stack	9
b	Push value of b onto the stack	9 3
c	Push value of c onto the stack	9 3 4
*	Pop the top 2 elements 3 and 4 from stack and push 3*4 onto the stack	9 12
d	Push value of d onto the stack	9 12 8
+	Pop the top 2 elements 12 and 8 from stack and push 12+8 onto the stack	9 20
e	Push value of e onto the stack	9 20 4
/	Pop the top 2 elements 20 and 4 from stack and push 20/4 onto the stack	9 5
-	Pop the top 2 elements 9 and 5 from stack and push 9-5 onto the stack	4

The topmost element of the stack is **4** and it will be the result of the postfix expression.

Program

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
char post[25];
int stack[25],top=-1,MAX=25;
void push(int);
int pop();
void evaluate();

void evaluate()
{
int i=0,val,result,op1,op2;
char c;
while(post[i]!='\0')
{
```

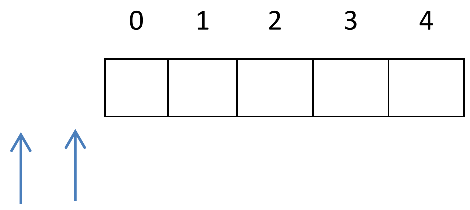
```
c=post[i];
if(isalpha(c))
{
printf("\nEnter the value for %c",c);
scanf("%d",&val);
push(val);
}
else
{
op2=pop();
op1=pop();
switch(c)
{
case '+': result=op1+op2;
break;
case '-': result=op1-op2;
break;
case '*': result=op1*op2;
break;
case '/': result=op1/op2;
break;
default: printf("\n Invalid operator");
}
push(result);
}

i++;
}

if(top!=0)
{
printf("\n Given Expression is Invalid");
exit(0);
}
else
printf(<<"\n Result is : %d",pop());
}

void push(int val)
{
if(top==MAX-1)
{
printf("\n Stack overflow");
exit(0);
}
```

```
else
{
top++;
stack[top]=val;
}
}

int pop()
{
int val;
if(top==-1)
{
printf("\n Stack is empty. Given expression is invalid");
exit(0);
}
else
{
val=stack[top];
top--;
return val;
}
}

int main()
{
printf("\nEnter the postfix expression:");
scanf(“%s”,post);
evaluate();
return 0;
}
```

QUEUE

Queue is also a special type of array in which insertion is performed at one end called **rear** and deletion is performed at the other end called **front**. Initially front and rear pointers are set to -1 which represents an empty queue. Queue is also called **FIFO list (First In First Out List)** since the first inserted element will be the first to be removed.

Example of an empty queue



front= -1   rear = -1

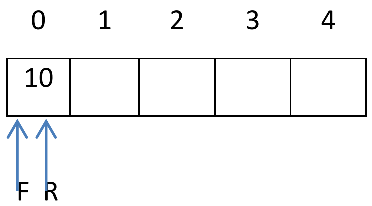
The basic operations in a queue are

1. Insertion, 2. Deletion and 3. Traversal

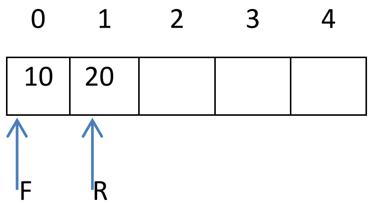
1. **<u>Insertion Operation (also called enqueue)</u>**

The new element is inserted at the rear end of the queue.

If Queue is empty both front and rear pointers are incremented by one and the new element is inserted at that position.



Otherwise only the rear pointer is incremented by one and the new element is inserted at that position. The front pointer remains the same.

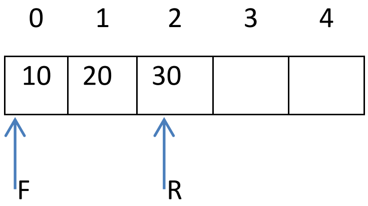


Before insertion we have to check whether the queue is full. If rear=MAXSIZE-1,where MAXSIZE is the maximum size of the queue, then the queue is full and further insertion is not possible. Such condition is called queue overflow.

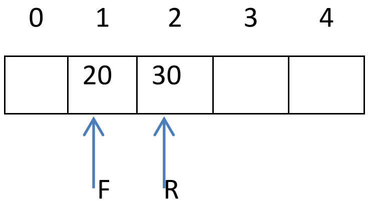
```
void insertion()
{
int val;
printf("\n Enter the element to be inserted:");
scanf(“%d”,&val);

if(rear==maxsize-1)
printf("\nQueue Overflow. Insertion is not Possible";
else
{
rear++;
q[rear]=val;
if(front==-1)
front++;
}
}
```

2. **<u>Deletion Operation(also called dequeue)</u>**

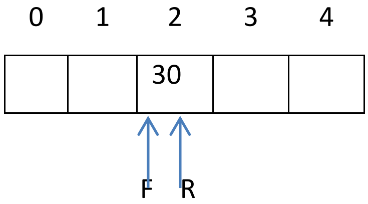
   The element is deleted from the front end of the queue. Before performing deletion operation we have to check whether the queue is empty. The queue becomes empty if both front and rear pointers are -1. Otherwise just increment the value of front by one for deletion. For example,



   After the deletion queue becomes



   Consider the following case:



Here the queue contains only one element and both the front and rear pointers are equal. In this case, after deletion operation the queue becomes empty and both front and rear pointers are set to -1.

```
void deletion()
{
int val;
if(front==-1)
printf("\n Queue is empty";
else
{
val=q[front];
if(front==rear)
{
front=-1;
rear=-1;
}
else
front++;
printf("\n The deleted element is: "<<val;
}
}
```

3. **Traversal Operation**

```
void traversal()
{
int i;
if(rear==-1)
printf("\n Queue is empty";
else
{
printf("\n The elements are: ";
for(i=front;i<=rear;i++)
printf(q[i]<<" ";
}
}
```

MAIN Function

```
int main()
{
int ch, val;
printf("MENU\n 1. Insertion 2. Deletion 3. Traversal 4. Exit");
do
{
printf("\n Enter the Choice");
scanf(“%d”,&ch);
switch(ch)
{
case 1:
insertion();
break;
case 2:
deletion();
break;
case 3:
traversal();
break;
case 4:
exit(1);
default:
printf("Invalid choice");
}
}while(1);
return(0);
}
```

CIRCULAR QUEUE

Consider the following case in a simple queue of maximum size 5.

0	1	2	3	4
		30	40	50
		F		R

Here Front points to the index 2 and rear points to 4 ie MAXSIZE-1. If we try to insert a new element into the above queue, it will not be possible because rear=MAXSIZE-1 eventhough two vacant positions are there at front. This is the limitation of a simple queue. To avoid such limitation we can use circular queue. In this queue the rear is incremented in such a way that after rear points to MAXSIZE-1 it will point to 0 if that position is empty. In this case the queue becomes full only if rear+1 points to front.

The pictorial representation of a circular queue is:

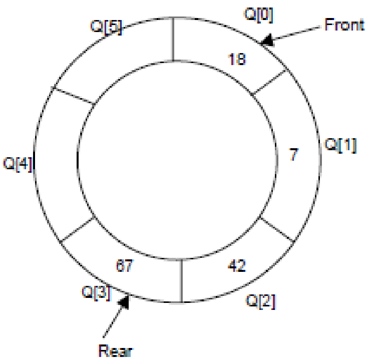


Fig. 4.11. A circular queue after inserting 18, 7, 42, 67.

After deleting 18 and 7, the queue becomes:

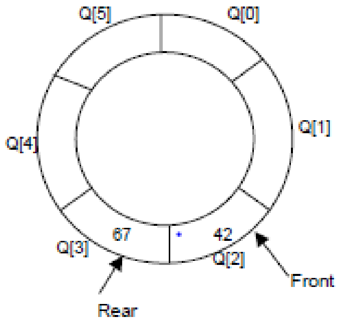


Fig. 4.12. A circular queue after popping 18, 7

Now we can insert 4 more elements into the above queue. But in a simple queue we can insert only 2 more elements.

After inserting three elements 30, 47 and 14 queue becomes

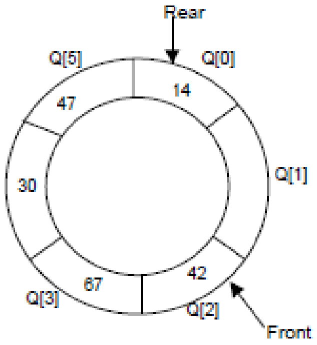


Fig. 4.13. A circular queue after pushing 30, 47, 14

The basic operations are 1. Inserion 2. Deletion and 3. Traversal

1. **Insertion Operation**

   Before inserting an element into the queue we have to check whether the queue is full. If (rear+1)%maxsize = front, the queue becomes full and further insertion is not possible. Otherwise increment the value of rear using the equation

   rear = (rear+1) % maxsize

   and the new element is inserted into that position. If the new element inserted is the first element in the queue, the value of front is set to zero.

```
void insertion()
{
int val;
if((rear+1)%maxsize == front)
printf("\nQueue Overflow. Insertion is not Possible");
else
{
printf(“\nEnter the element to be inserted”);
scanf(“%d”,&val);
rear = (rear+1)%maxsize;
q[rear]=val;
if(front==-1)
front++;
}
}
```

2. **Deletion Operation**

   Before deleting an element from the queue we have to check whether the queue is empty. If the value of front is -1 the queue is empty and deletion is not possible. Otherwise check whether the queue contains only a single element. In this case, the value of front and rear will be the same and after the deletion reset the value of front and rear to -1. Otherwise increment the value of front by using the equation:

   front = (front+1) % maxsize.

```
void deletion()
{
if(front==-1)
printf("\n Queue is empty");
else
{
printf("\n The deleted element is : "<<q[front]);
if(front==rear)
{
front=-1;
rear=-1;
}
else
front = (front+1) % maxsize;
}
}
```

3. **Traversal Operation**

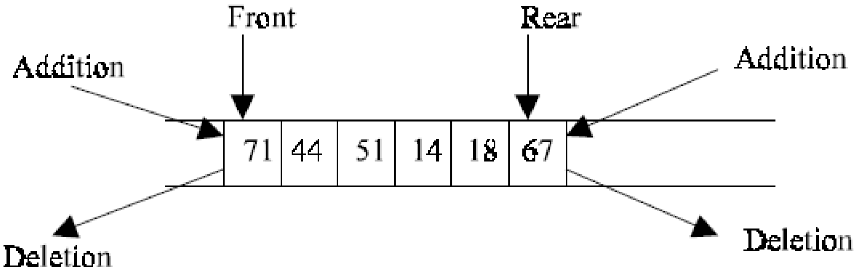
```
void traversal()
{
int i;
if(rear==-1)
printf("\n Queue is empty");
else
{
printf("\n The elements are: ");
i = front;
do
{
printf("%d ",q[i]);
if(i==rear)
break;
else
i = (i+1) % maxsize;
}while(1);
}
}
```

MAIN Function

```
int main()
{
int ch, val;
printf("MENU\n 1. Insertion 2. Deletion 3. Traversal 4. Exit");
do
{
printf("\n Enter the Choice");
scanf(“%d”,&ch);
switch(ch)
{
case 1:
insertion();
break;
case 2:
deletion();
break;
case 3:
traversal();
break;
case 4:
exit(1);
default:
printf("Invalid choice");

}
}while(1);
return(0);
}
```

DOUBLE ENDED QUEUE (DEQUEUE)

In double ended queue insertions and deletions can be done from both the ends.



There are five operations in double ended queue.

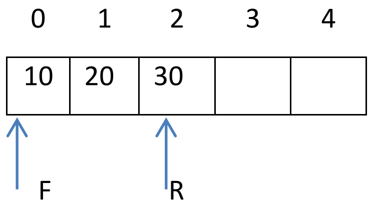
1. Insertion at rear end.
2. Insertion at front end.

3. Deletion from rear end.
4. Deletion from front end.
5. Traversal.

The operations, Insertion at the rear end and deletion from front end are same as insertion and deletion operations respectively of simple queue. The other two operations are explained below:

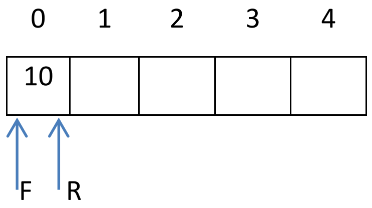
Insertion at front end

Before insertion we have to check whether queue is full. Consider the following situation:

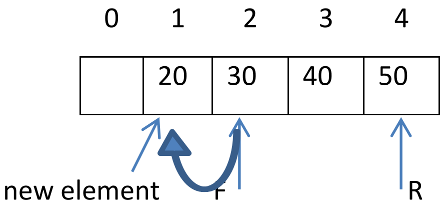


In this case, front pointer points to 0, no further insertion at front end is possible. So this case is considered as queue overflow condition.

If Queue is empty (ie, if front=rear=-1) both front and rear pointers are incremented by one and the new element is inserted at that position.



Otherwise decrement the value of front by 1 and insert the element at that position.



```
void insertAtFront()
{
int val;
if(front==0)
printf(”Queue is full and insertion at front end is not possible\n”);
else
{
printf(“\nEnter the element to be inserted”);
scanf(“%d”,&val);
if(front == -1)
{
front++;
rear++;
q[front] = val;
}
else
{
front--;
q[front] = val;
}
}
```

Deletion from rear end

Before deleting an element from the queue, first we have to check whether the queue is empty. Otherwise if it contains only one element the queue will be empty after the deletion. If so reset the values of front and rear to -1. If it contains more than one element just decrement the value of rear by 1.

```
void deleteAtRear()
{
if(front == -1)
printf(”Queue is empty\n”);
else
{
printf("\n The deleted element is: %d",q[rear];
if(front==rear)
{
front=-1;
rear=-1;
}
else
rear--;
}
}
```

Traversal Operation

```
void traversal()
{
int i;
if(rear==-1)
printf("\n Queue is empty");
else
```

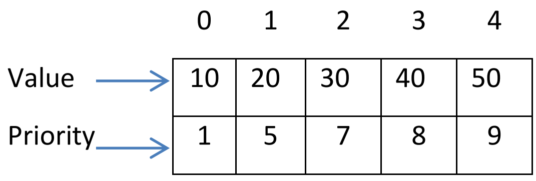
```
    {
    printf("\n The elements are: ");
    for(i=front; i<=rear; i++)
    printf("%d ",q[i]);
    }
    }
```

MAIN Function

```
int main()
{
int ch;
int val;
printf("\n 1. Insertion at Front 2. Insertion at Rear 3. Deletion from Front 4. Deletion from
Rear 5.  Traversal 6. Exit");
do
{
printf("\n Enter the Choice");
cin>>ch;
switch(ch)
{
case 1:
insertAtFront();
break;
case 2:
insertAtRear();
break;
case 3:
deleteAtFront();
break;
case 4:
deleteAtRear();
break;
case 5:
traversal();
break;
case 6:
exit(1);
default:
printf("Invalid choice");
}
}while(ch!=6);
return(0);
}
```

PRIORITY QUEUE

Priority queue is a variation of simple queue in which each element has a value associated with it called its priority. When deletion operation is performed we remove the element with highest priority. In a priority queue the elements are stored in the order of priority value so that the element with highest priority will always be at the front position. So elements are inserted into the queue in the order of priority.

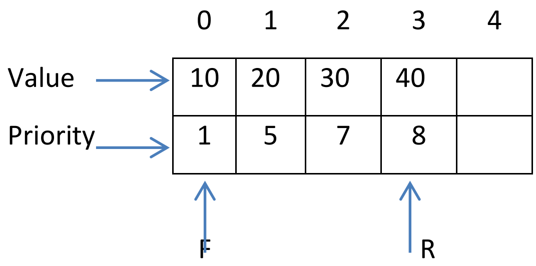


	0	1	2	3	4
Value	10	20	30	40	50
Priority	1	5	7	8	9

The basic operations are

1. Insertion, 2. Deletion and 3. Traversal

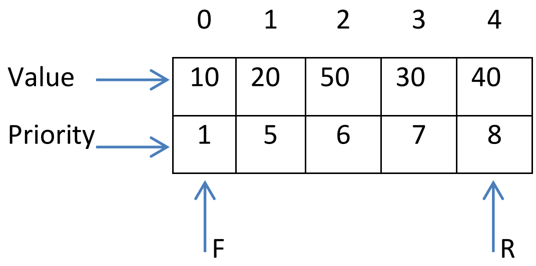
1. Insertion Operation



	0	1	2	3	4
Value	10	20	30	40	
Priority	1	5	7	8	

Suppose we have to insert an element with value 50 and priority 6. First check whether the queue is full. If not full, find the position at which the new element is inserted based on its priority and then the element is inserted at that position by shifting all the elements from that position right by one position.

After the insertion the above queue becomes:



	0	1	2	3	4
Value	10	20	50	30	40
Priority	1	5	6	7	8

The rear pointer is incremented by one after every insertion.

2. Deletion Operation

Deletion operation is same as that of simple queue. The element is deleted from the front end of the queue. Before performing deletion operation we have to check whether the queue is empty. The queue becomes empty if both front and rear pointers are -1. Otherwise just increment the value of front by one for deletion.

If the queue contains only one element, both the front and rear pointers are equal. In this case, after deletion operation the queue becomes empty and both front and rear pointers are set to -1.