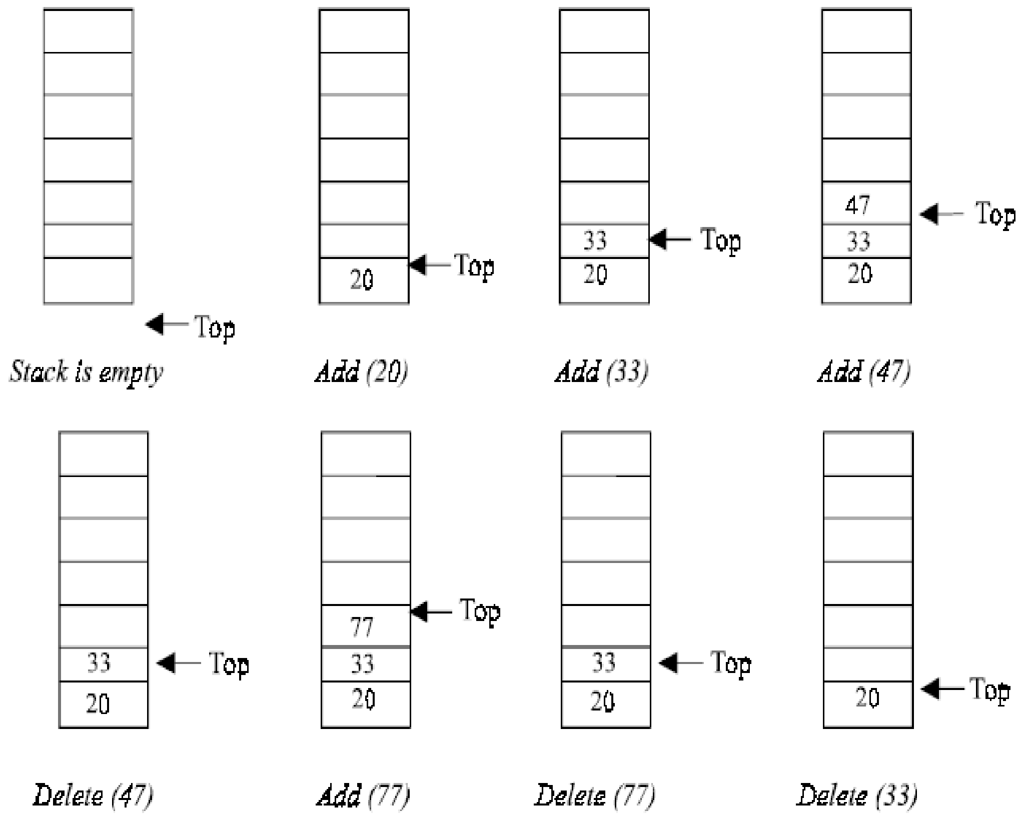
Experiment 4

Aim: Write a program that implement stack (its operations) using i) Arrays ii) Pointers

Objectives:

- To understand and implement the stack data structure
- To design, implement, and analyze stack implanted using array and linked list
- To understand different operations on stacks
- To understand the implementation of stack operations using array and linked list

Stack is a non primitive linear data structure in which a new data element may be added/inserted and deleted at only one end, called top of the stack. the last added element will be first removed from the stack. That is why the stack is also called ***Last-in-First-out (LIFO)***. Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack.
The operation of the stack can be illustrated in the following figure:



. Fig 1: **Stack operations**

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

***PUSH**:* The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

***POP**:* The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs**Stack using linked list:** Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.

Stack Implementation

Stack can be implemented in two ways:
1. Static implementation (using arrays)
2. Dynamic implementation (using pointers)

4.1 Static implementation using arrays

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm:

Push(Stack [MAXSIZE], item)

```
Let stack[Maxsize]is an array for implementing the stack.
1 [check for stack overflow?]
   If Top=Maxsize-1, then print: overflow and exit.
2 Set TOP=TOP+1 [increase TOP by 1]
3 Set STACK[TOP]:=ITEM [inserts item in new TOP position]
4 EXIT
```

Pop(Stack [MAXSIZE], item)

```
1 [check for stack underflow?]
   If Top<0, then print: stack underflow and exit.
   Else [remove the Top element]
   Set item=stack[TOP]
2 Decrement the stack top
   Set Top=Top-1
3 Return the deleted item from the stack
4 EXIT
```

Implementation:
/* Program of stack using array*/

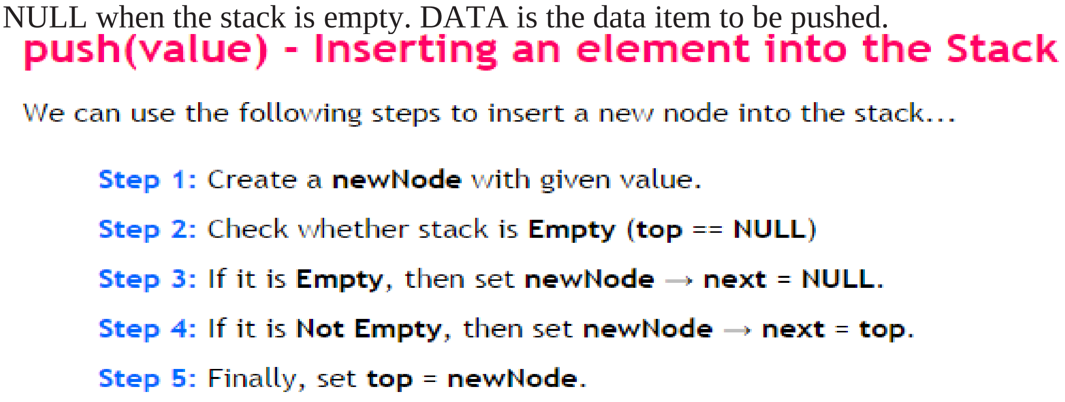
```
#include<stdio.h>
#include<stdlib.h>
int top=-1;
#define MAX_SIZE 101
int S[MAX_SIZE];
void Push(int x)
{
        if(top == MAX_SIZE -1)
        {
                printf("Stack Overflow");
                return;
        }else{
                S[++top]=x;
        }
}
void Pop()
{
        if(top==-1)
        {
                printf("Stack is EMPTY");
                return;
        }else{
                top--;
        }
}
int Top()
{
        return S[top];
}
void printStack()
{
        int i;
        printf("Stack Elements are:\n");
        for(i=0;i<=top;i++)
        {
                printf("%d\n",S[i]);
        }
}
int main()
{
        int choice,data;

        while(1)
        {
                printf("\n0.Exit\n1. PUSH an element into Stack\n2. Pop an Element from Stack\n3. 
Top of the Stack\n");
                printf("4. Display\n");
                printf("Enter your Choice:\t");
```

```
		while(scanf("%d",&choice)==0)
		{
			printf("Only Numbers!!!");
			int c;
			while((c=getchar())!='\n' && c!=EOF); //clear the stdin
		}
		switch(choice)
		{
			case 0:
				exit(0);
			case 1:
				printf("\nEnter an Element:\t");
				while(scanf("%d",&data)==0)
				{
					printf("Only Numbers!!!");
					int c;
					while((c=getchar())!='\n' && c!=EOF); //clear the stdin
				}
				Push(data);
				break;
			case 2:
				Pop();
				break;
			case 3:
				printf("Top of the Elements in Stack %d",Top());
				break;
			case 4:
				printStack();
				break;
			default:
				printf("WRONG ENTRY");
				break;
		}
	}
	return 0;
}
```

Output:

Stack implementation using linked list

Algorithm for PUSH operation:
Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.



push(value) - Inserting an element into the Stack

We can use the following steps to insert a new node into the stack...

Step 1: Create a **newNode** with given value.

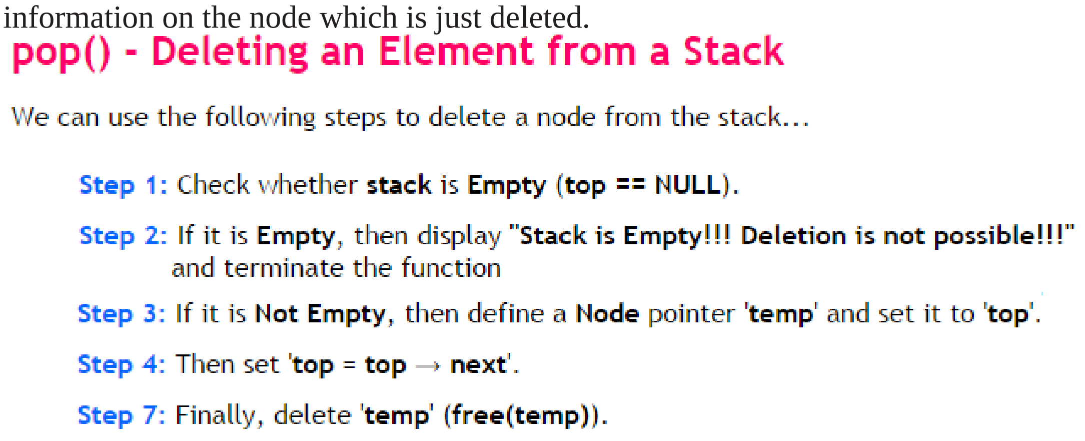
Step 2: Check whether stack is **Empty** (**top** == **NULL**)

Step 3: If it is **Empty**, then set **newNode** → **next** = **NULL**.

Step 4: If it is **Not Empty**, then set **newNode** → **next** = **top**.

Step 5: Finally, set **top** = **newNode**.

Algorithm for POP operation
Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.



pop() - Deleting an Element from a Stack

We can use the following steps to delete a node from the stack...

Step 1: Check whether **stack** is **Empty** (**top == NULL**).

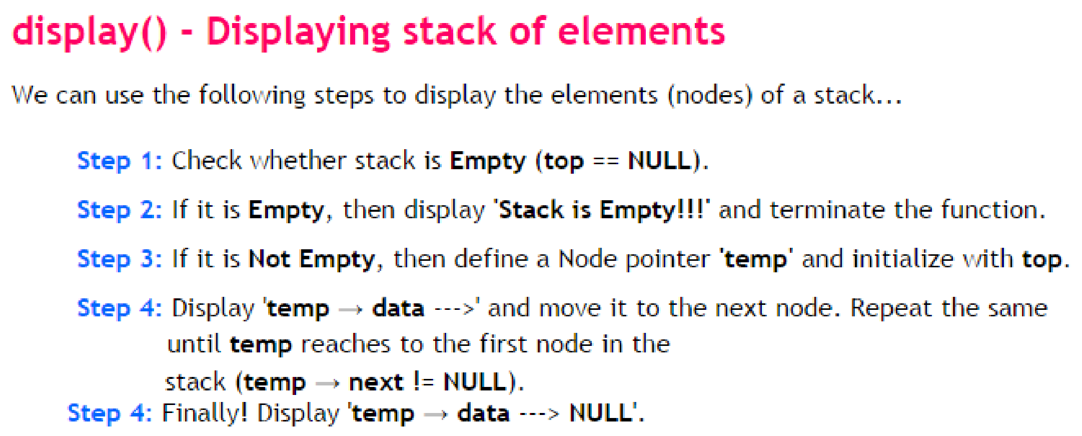
Step 2: If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function

Step 3: If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

Step 4: Then set '**top** = **top** → **next**'.

Step 7: Finally, delete '**temp**' (**free(temp)**).

Algorithm for deletion operation



display() - Displaying stack of elements

We can use the following steps to display the elements (nodes) of a stack...

Step 1: Check whether stack is **Empty** (**top** == **NULL**).

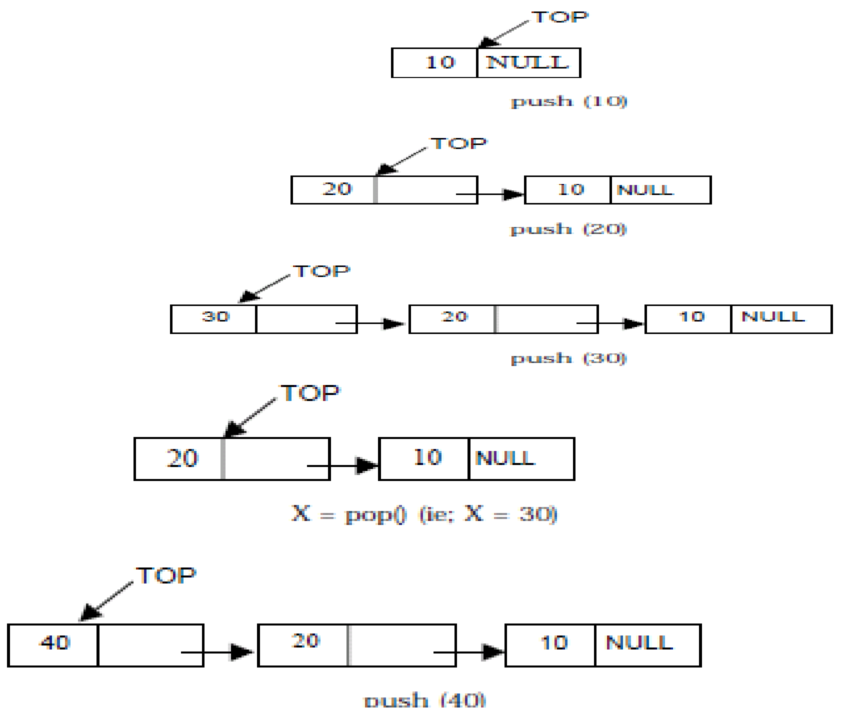
Step 2: If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.

Step 3: If it is **Not Empty**, then define a Node pointer '**temp**' and initialize with **top**.

Step 4: Display '**temp** → **data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack (**temp** → **next** != **NULL**).

Step 4: Finally! Display '**temp** → **data** ---> **NULL**'.

Illustration of stack operations using linked list:



Program:

/* Write a program to perform operations on stack using linked list */

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct node
{
        int item;
        struct node *next;
}*top;
int count;
int empty()
{
        if(top==NULL)
                return 1;
        else
                return 0;
}
void show()
{
        int i;
        struct node *p;
        if(empty())
        {
                printf("\nNo elements in the stack:");
                return;
        }
```

```
	printf("\nTotal Elements present in the stack  %d :",count);
	p=top;
	while(p!=NULL)
	{
		printf("\n%d",p->item);
		p=p->next;
	}
}
void push()
{
	int x,i;
	struct node *temp;
	printf("\nEnter the element to push into stack: ");
	scanf("%d",&x);
	temp=(struct node*)malloc(sizeof(struct node));
	temp->item=x;
	temp->next=NULL;
	if(top==NULL)
		top=temp;
	else
	{
		temp->next=top;
		top=temp;
	}
	count++;
}
void pop()
{
	struct node *temp;
	if(empty())
	{
		printf("\n NO element in stack to delete:");
		return;
	}
	temp=top;
	top=top->next;
	free(temp);
	count--;
}
void main()
{
	int choice,ch;
	do
	{
		printf("\n1.push\n2.pop\n3.show\nenter choice: ");
		scanf("%d",&choice);
		switch(choice)
		{
			case 1: push();break;
```

```
                        case 2: pop();break;
                        case 3: show(); break;
                        default: printf("\n Wrong Choice: ");
                }
                printf("\n Do You Want To Continue:  (1/0)   : ");
                scanf("%d",&ch);
        }       while(ch==1);
        getch();
}
```

output:

Experiment 5

Aim: Write a program that implement Linear Queue (its operations) using i) Arrays ii) Pointers.

Objectives:

- To understand and implement the queue data structure
- To design, implement, and analyze queue operations implemented using array and linked list
- To understand different operations on queue
- To understand the implementation of queue operations using array and linked list

QUEUES

Queue is a linear data structure.

- In which data will be inserted and deleted through two different ends.
- Follows the mechanism of FIFO (first in first out).
- Insertions are allowed at rear end.
- Deletions are allowed at front end.
- Queue allows insertion, deletion and display operations.
- Insertion operation is also called as enqueue operation.
- The deletion operation is also called as dequeue operation.
- Overflow occurs whenever the queue is full
- Under flow occurs whenever the queue is empty.

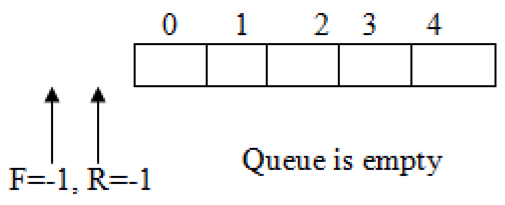
A queue can be represented by two ways.

- ➢ Array representation
- ➢ Linked list representation.

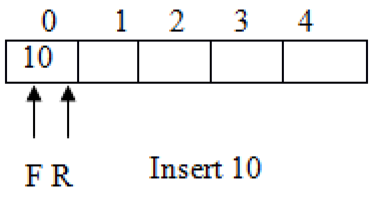
Queue implementation using Array

Following figure will illustrate the basic operations on queue.

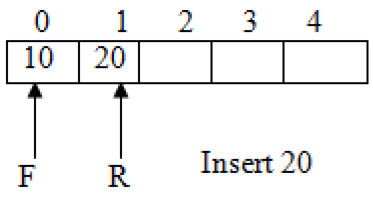
Consider a queue Q with 5 elements Q[5]. Initially there is no element in the queue front and rear pointers pointing to -1.



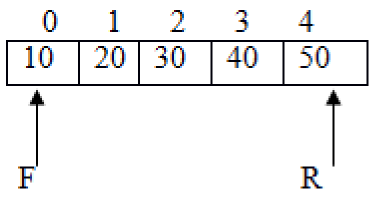
Insert element 10 to the queue, when ever we are inserting element 10 to the queue before that we need to increase front and rear values by one, now front and rear points to array index zero.



Insert element 20 to the queue, when ever we are inserting element 20 to the queue before that we need to increase rear values by one, rear points to array index one.



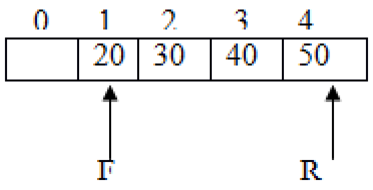
Similarly After inserting 30, 40,50 to the queue front is pointing to zero and rear is pointing to four.



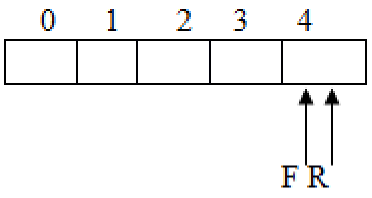
After inserting 30, 40, 50

Now queue is full, after this point it is not possible to insert a new element to the queue, this condition is called overflow condition

next if you call the delete operation first inserted element is deleted (i.e. 10) by incrementing front pointer now queue is look like



After deleting remaining elements 20,30,40,50 from the queue



Algorithm:

Let Q be the array of some specified size say SIZE

INSERTING AN ELEMENT INTO THE QUEUE

```
1. Initialize front=0 rear = –1
2. Input the value to be inserted and assign to variable “data”
3. If (rear >= SIZE)
   (a) Display “Queue overflow”
   (b) Exit
4. Else
       (a) Rear = rear +1
5. Q[rear] = data
6. Exit
```

DELETING AN ELEMENT FROM QUEUE

```
1. If (rear< front)
   (a) Front = 0, rear = –1
   (b) Display “The queue is empty”
   (c) Exit
2. Else
       (a) Data = Q[front]
3. Front = front +1
4. Exit
```

Applications of Queues:

a. Round Robin technique for processor scheduling is implemented using queues.

b. All types of customer service center software’s are designed using queues to store customer information.(ex. Train reservation)

c. Printer server routines are designed using queues.

QUEUE implementation using ARRAY

Program:

```
 # include<stdio.h>
# define MAX 5
intqueue_arr[MAX];
int rear = -1;
int front = -1;
main()
{
	int choice;
	while(1)
	{
		printf("1.Insert\n");
		printf("2.Delete\n");
		printf("3.Display\n");
		printf("4.Quit\n");
		printf("Enter your choice : ");
		scanf("%d",&choice);

		switch(choice)
		{
		case 1 :
			insert();
			break;
		case 2 :
			del();
			break;
		case 3:
			display();
			break;
		case 4:
			exit(1);
		default:
			printf("Wrong choice\n");
		}/*End of switch*/
	}/*End of while*/
}/*End of main()*/

insert()
{
	intadded_item;
	if (rear==MAX-1)
		printf("Queue Overflow\n");
	else
	{
		if (front==-1)  /*If queue is initially empty */
			front=0;
		printf("Input the element for adding in queue : ");
```

```
                scanf("%d", &added_item);
                rear=rear+1;
                queue_arr[rear] = added_item ;
        }
}/*End of insert()*/

del()
{
        if (front == -1 || front > rear)
        {
                printf("Queue Underflow\n");
                return ;
        }
        else
        {
                printf("Element deleted from queue is : %d\n", queue_arr[front]);
                front=front+1;
        }
}/*End of del() */

display()
{
        int i;
        if (front == -1)
                printf("Queue is empty\n");
        else
        {
                printf("Queue is :\n");
                for(i=front;i<= rear;i++)
                        printf("%d  ",queue_arr[i]);
                printf("\n");
        }
}/*End of display() */
```

Output:

QUEUE implementation using linked list

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct Node
{
 int data;
 struct Node *next;
}*front = NULL,*rear = NULL;
void insert(int);
void delete();
void display();
void main()
{
   int choice, value;
   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
          case 1: printf("Enter the value to be insert: ");
                  scanf("%d", &value);
                  insert(value);
                  break;
          case 2: delete(); break;
          case 3: display(); break;
          case 4: exit(0);
          default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
```

```
        rear = newNode;
    }
    printf("\nInsertion is Success!!!\n");
}
void delete()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        front = front -> next;
        printf("\nDeleted element: %d\n", temp->data);
        free(temp);
    }
}
void display()
{
    if(front == NULL)
        printf("\nQueue is Empty!!!\n");
    else{
        struct Node *temp = front;
        while(temp->next != NULL){
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL\n",temp->data);
    }  }
```

Output:

Experiment 6

Aim: Write a C program that implement Deque (its operations).

```
#include<stdio.h>
#include<process.h>
#define MAX 30
typedef struct dequeue
{
   int data[MAX];
   int rear,front;
}dequeue;
void initialize(dequeue *p);
int empty(dequeue *p);
int full(dequeue *p);
void enqueueR(dequeue *p,int x);
void enqueueF(dequeue *p,int x);
int dequeueF(dequeue *p);
int dequeueR(dequeue *p);
void print(dequeue *p);
void main()
{
   int i,x,op,n;
   dequeue q;

initialize(&q);
   do
   {
printf("\n1.Create\n2.Insert(rear)\n3.Insert(front)\n4.Delete(rear)\n5.Delete(front");
         printf("\n6.Print\n7.Exitn\n Enter your choice:");

scanf("%d",&op);
switch(op)
         {
 case 1:
printf("/nEnter number of elements:");
scanf("%d",&n);
initialize(&q);
printf("\nEnter the data:");
for(i=0;i<n;i++)
  {
scanf("%d",&x);
if(full(&q))
{
printf("\nQueue is full!!");
exit(0);
}
enqueueR(&q,x);
   }
break;
case 2:
printf("\nEnter element to be inserted:");
scanf("%d",&x);
if(full(&q))
   {
printf("\nQueue is full!!");
exit(0);
   }
enqueueR(&q,x);
break;
case 3: printf("\nEnter the element to be inserted:");
scanf("%d",&x);
if(full(&q))
{
printf("\nQueue is full!!");
   exit(0);
}
enqueueF(&q,x);
break;
case 4: if(empty(&q))
{
printf("\nQueue is empty!!");
 exit(0);
}
x=dequeueR(&q);
printf("\nElement deleted is %dn",x);
break;
case 5: if(empty(&q))
{
printf("\nQueue isempty!!");
exit(0);

}
x=dequeueF(&q);
printf("\nElement deleted is %dn",x);
break;
case 6: print(&q);
break;
default: break;
         }
   }while(op!=7);
}
void initialize(dequeue *P)
{
   P->rear=-1;
   P->front=-1;
}
int empty(dequeue *P)
{
   if(P->rear==-1)
          return(1);
   return(0);
}
int full(dequeue *P)
{
if((P->rear+1)%MAX==P->front)
          return(1);
   return(0);
}
void enqueueR(dequeue *P,int x)
{
   if(empty(P))
   {
          P->rear=0;
          P->front=0;

P->data[0]=x;
   }
   else
   {
P->rear=(P->rear+1)%MAX;

P->data[P->rear]=x;
   }
}
void enqueueF(dequeue *P,int x)
{
   if(empty(P))
   {
          P->rear=0;
          P->front=0;
P->data[0]=x;
   }
   else
   {
P->front=(P->front-1+MAX)%MAX;
P->data[P->front]=x;
   }
}
int dequeueF(dequeue *P)
{
   int x;
x=P->data[P->front];
if(P->rear==P->front)
//delete the last element
initialize(P);
   else
P->front=(P->front+1)%MAX;
   return(x);
```

```
}
int dequeueR(dequeue *P)
{
   int x;
x=P->data[P->rear];
if(P->rear==P->front)
         initialize(P);
   else
P->rear=(P->rear-1+MAX)%MAX;
   return(x);
}
void print(dequeue *P)
{
   if(empty(P))
   {
printf("\nQueue is empty!!");
         exit(0);
   }
   {
     int i;
   i=P->front;
while(i!=P->rear)
   {
printf("\n%d",P->data[i]);
i=(i+1)%MAX;
   }
   }
printf("\n%dn",P->data[P->rear]);
}
```

OUTPUT:

Write a program to implement all the functions of a dictionary using hashing.

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

struct set
{
  int key;
  int data;
};
struct set *array;
int capacity = 10;
int size = 0;

int hashFunction(int key)
{
  return (key % capacity);
}
int checkPrime(int n)
{
  int i;
  if (n == 1 || n == 0)
  {
  return 0;
  }
  for (i = 2; i < n / 2; i++)
  {
  if (n % i == 0)
  {
    return 0;
  }
  }
  return 1;
}
int getPrime(int n)
{
  if (n % 2 == 0)
  {
  n++;
  }
  while (!checkPrime(n))
  {
  n += 2;
  }
  return n;
}
void init_array()
{
  capacity = getPrime(capacity);
  array = (struct set *)malloc(capacity * sizeof(struct set));
  for (int i = 0; i < capacity; i++)
  {
  array[i].key = 0;
  array[i].data = 0;
  }
}

void insert(int key, int data)
{
  int index = hashFunction(key);
  if (array[index].data == 0)
  {
  array[index].key = key;
  array[index].data = data;
  size++;
  printf("\n Key (%d) has been inserted \n", key);
  }
  else if (array[index].key == key)
  {
  array[index].data = data;
  }
  else
  {
  printf("\n Collision occured  \n");
  }
}

void remove_element(int key)
{
  int index = hashFunction(key);
  if (array[index].data == 0)
  {
  printf("\n This key does not exist \n");
  }
  else
  {
  array[index].key = 0;
  array[index].data = 0;
  size--;
  printf("\n Key (%d) has been removed \n", key);
  }
}
void display()
{
  int i;
  for (i = 0; i < capacity; i++)
  {
  if (array[i].data == 0)
  {
    printf("\n array[%d]: / ", i);
  }
  else
  {
    printf("\n key: %d array[%d]: %d \t", array[i].key, i, array[i].data);
  }
  }
}

int size_of_hashtable()
{
  return size;
}

int main()
{
  int choice, key, data, n;
  int c = 0;
  init_array();
```

```
do
{
printf("1.Insert item in the Hash Table"
   "\n2.Remove item from the Hash Table"
   "\n3.Check the size of Hash Table"
   "\n4.Display a Hash Table"
   "\n\n Please enter your choice: ");

scanf("%d", &choice);
switch (choice)
{
case 1:

  printf("Enter key -:\t");
  scanf("%d", &key);
  printf("Enter data -:\t");
  scanf("%d", &data);
  insert(key, data);

  break;

case 2:

  printf("Enter the key to delete-:");
  scanf("%d", &key);
  remove_element(key);

  break;

case 3:

  n = size_of_hashtable();
  printf("Size of Hash Table is-:%d\n", n);

  break;

case 4:

  display();
```

```
    break;

  default:

    printf("Invalid Input\n");
  }

  printf("\nDo you want to continue (press 1 for yes): ");
  scanf("%d", &c);

  } while (c == 1);
}
```

OUTPUT:

```
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
Enter key -:     20
Enter data -:    34

 Key (20) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 1
Enter key -:     66
Enter data -:    49

 Key (66) has been inserted

Do you want to continue (press 1 for yes): 1
1.Insert item in the Hash Table
```

```
2.Remove item from the Hash Table
3.Check the size of Hash Table
4.Display a Hash Table

 Please enter your choice: 4

 key: 66 array[0]: 49
 array[1]: /
 array[2]: /
 array[3]: /
 array[4]: /
 array[5]: /
 array[6]: /
 array[7]: /
 array[8]: /
 key: 20 array[9]: 34
 array[10]: /
Do you want to continue (press 1 for yes): 0
```

Write a program that implement Binary Search Trees to perform the following operations i) Creation ii) Insertion iii) Deletion iv) Traversal

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

struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
   int ch;

   printf("\nOPERATIONS ---");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Delete an element from the tree\n");
   printf("3 - Inorder Traversal\n");
   printf("4 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         delete();
         break;
      case 3:
         inorder(root);
         break;
      case 4:
         exit(0);
      default :
         printf("Wrong choice, Please enter correct choice  ");
         break;
      }
   }
}

/* To insert a node in the tree */
```

```
void insert()
{
   create();
   if (root == NULL)
      root = temp;
   else
      search(root);
}

/* To create a node */
void create()
{
   int data;

   printf("Enter data of node to be inserted : ");
   scanf("%d", &data);
   temp = (struct btnode *)malloc(1*sizeof(struct btnode));
   temp->value = data;
   temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
   if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node value insert
at right */
      search(t->r);
   else if ((temp->value > t->value) && (t->r == NULL))
      t->r = temp;
   else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root node value insert
at left */
      search(t->l);
   else if ((temp->value < t->value) && (t->l == NULL))
      t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
   if (root == NULL)
```

```
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To check for the deleted node */
void delete()
{
    int data;

    if (root == NULL)
    {
        printf("No elements in a tree to delete");
        return;
    }
    printf("Enter the data to be deleted : ");
    scanf("%d", &data);
    t1 = root;
    t2 = root;
    search1(root, data);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
{
    if ((data>t->value))
    {
        t1 = t;
        search1(t->r, data);
    }
    else if ((data < t->value))
    {
        t1 = t;
        search1(t->l, data);
    }
    else if ((data==t->value))
    {
        delete1(t);
    }
}

/* To delete a node */
void delete1(struct btnode *t)
{
    int k;

    /* To delete leaf node */
    if ((t->l == NULL) && (t->r == NULL))
    {
        if (t1->l == t)
        {
            t1->l = NULL;
        }
        else
        {
            t1->r = NULL;
        }
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having one left hand child */
    else if ((t->r == NULL))
    {
        if (t1 == t)
        {
            root = t->l;
            t1 = root;
        }
        else if (t1->l == t)
        {
            t1->l = t->l;
        }
        else
        {
            t1->r = t->l;
        }
        t = NULL;
        free(t);
        return;
    }

    /* To delete node having right hand child */
    else if (t->l == NULL)
    {
        if (t1 == t)
        {
            root = t->r;
            t1 = root;
        }
        else if (t1->r == t)
            t1->r = t->r;
        else
            t1->l = t->r;
        t == NULL;
        free(t);
        return;
    }

    /* To delete node having two child */
    else if ((t->l != NULL) && (t->r != NULL))
    {
        t2 = root;
        if (t->r != NULL)
        {
            k = smallest(t->r);
            flag = 1;
        }
        else
        {
            k =largest(t->l);
            flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
    if (t->r != NULL)
    {
        t2 = t;
        return(largest(t->r));
    }
    else
        return(t->value);
}
```

OUTPUT:

```
OPERATIONS ---
1 - Insert an element into tree
2 - Delete an element from the tree
3 - Inorder Traversal
4 - Exit
```

```
Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 5

Enter your choice : 1
Enter data of node to be inserted : 34

Enter your choice : 1
Enter data of node to be inserted : 88

Enter your choice : 1
Enter data of node to be inserted : 2

Enter your choice : 3
2 -> 5 -> 12 -> 34 -> 88 ->
Enter your choice : 2
Enter the data to be deleted : 12

Enter your choice : 3
2 -> 5 -> 34 -> 88 ->
Enter your choice : 2
Enter the data to be deleted : 34

Enter your choice :
3
2 -> 5 -> 88 ->
Enter your choice : 4
```

Write a program to implement the tree traversal methods using recursion.

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

struct btnode
```

```
{
   int value;
   struct btnode *l;
   struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void insert();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);

int flag = 1;

void main()
{
   int ch;
 1
   printf("\n TREE TRAVERSAL METHODS USING RECURSION");
   printf("\n1 - Insert an element into tree\n");
   printf("2 - Inorder Traversal\n");
   printf("3 - Preorder Traversal\n");
   printf("4 - Postorder Traversal\n");
   printf("5 - Exit\n");
   while(1)
   {
      printf("\nEnter your choice : ");
      scanf("%d", &ch);
      switch (ch)
      {
      case 1:
         insert();
         break;
      case 2:
         inorder(root);
         break;
      case 3:
         preorder(root);
         break;
```

```
        case 4:
            postorder(root);
            break;
        case 5:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}

/* To create a node */
void create()
{
    int data;

    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root node value insert
at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))    /* value less than root node value insert
at left */
        search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    printf("%d -> ", t->value);
    if (t->l != NULL)
        preorder(t->l);
    if (t->r != NULL)
        preorder(t->r);
}

/* To find the postorder traversal */
```

```
void postorder(struct btnode *t)
{
   if (root == NULL)
   {
      printf("No elements in a tree to display ");
      return;
   }
   if (t->l != NULL)
      postorder(t->l);
   if (t->r != NULL)
      postorder(t->r);
   printf("%d -> ", t->value);
}
```

OUTPUT:

```
 TREE TRAVERSAL METHODS USING RECURSION
1 - Insert an element into tree
2 - Inorder Traversal
3 - Preorder Traversal
4 - Postorder Traversal
5 - Exit

Enter your choice : 1
Enter data of node to be inserted : 12

Enter your choice : 1
Enter data of node to be inserted : 5

Enter your choice : 34
Wrong choice, Please enter correct choice
Enter your choice : 1
Enter data of node to be inserted : 34

Enter your choice : 1
Enter data of node to be inserted : 2

Enter your choice : 1
Enter data of node to be inserted : 88
Enter your choice : 2
2 -> 5 -> 12 -> 34 -> 88 ->
Enter your choice : 3
12 -> 5 -> 2 -> 34 -> 88 ->
Enter your choice : 4
2 -> 5 -> 88 -> 34 -> 12 ->
Enter your choice : 5
```

10. Write a program that implements the following sorting methods to sort a given list of integers in ascending order
i) Heap sort
ii) Merge sort

```
// Heap Sort in C

  #include <stdio.h>
#include <conio.h>
#define MAX 10
void RestoreHeapUp(int *,int);
     ●   void RestoreHeapDown(int*,int,int);
int main()
{
int Heap[MAX],n,i,j;
//clrscr();
printf("\n Enter the number of elements : ");
scanf("%d",&n);
printf("\n Enter the elements : ");
for(i=1;i<=n;i++)
{
scanf("%d",&Heap[i]);
RestoreHeapUp(Heap, i); // Heapify
}
// Delete the root element and heapify the heap
j=n;
for(i=1;i<=j;i++)
{
int temp;
temp=Heap[1];
Heap[1]= Heap[n];
Heap[n]=temp;
n = n-1; // The element Heap[n] is supposed to be deleted
RestoreHeapDown(Heap,1,n); // Heapify
}
n=j;
printf("\n The sorted elements are: ");
for(i=1;i<=n;i++)
printf("%4d",Heap[i]);
return 0;
}
void RestoreHeapUp(int *Heap,int index)
{
int val = Heap[index];
while( (index>1) && (Heap[index/2] < val) )// Check parent's value
{
Heap[index]=Heap[index/2];
index /= 2;
}
Heap[index]=val;
}
void RestoreHeapDown(int *Heap,int index,int n)
{Pº
int val = Heap[index];
int j=index*2;
while(j<=n)
{
if( (j<n) && (Heap[j] < Heap[j+1]))// Check sibling's value
j++;
if(Heap[j] < Heap[j/2]) // Check parent's value
break;
Heap[j/2]=Heap[j];
j=j*2;
}
Heap[j/2]=val;
}
```

Output:

```
 Enter the number of elements : 8

 Enter the elements : 23
44
1
67
22
12
66
99
```

```
 The sorted elements are:    1  12  22  44  23  66  67  99
```

MERGE SORT

```
#include <stdio.h>
#include <conio.h>
#define size 100
void merge(int a[], int, int, int);
void merge_sort(int a[],int, int);
void main()
{
int arr[size], i, n;
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]);
}
merge_sort(arr, 0, n-1);
printf("\n The sorted array is: \n");
for(i=0;i<n;i++)
printf(" %d\t", arr[i]);
getch();
}
void merge(int arr[], int beg, int mid, int end)
{
int i=beg, j=mid+1, index=beg, temp[size], k;
while((i<=mid) && (j<=end))
{
 if(arr[i] < arr[j])
 {
temp[index] = arr[i];
i++;
 }
 else
 {
temp[index] = arr[j];
j++;
 }
index++;
}
if(i>mid)
{
 while(j<=end)
 {
     temp[index] = arr[j];
j++;
index++;
 }
}
else
{
 while(i<=mid)
 {
temp[index] = arr[i];
i++;
index++;
 }
}
for(k=beg;k<index;k++)
arr[k] = temp[k];
}
void merge_sort(int arr[], int beg, int end)
{
int mid;
if(beg<end)
{
mid = (beg+end)/2;
 merge_sort(arr, beg, mid);
merge_sort(arr, mid+1, end);
 merge(arr, beg, mid, end);
}
}
```

Output:

```
 Enter the number of elements in the array : 6

 Enter the elements of the array: 88
12
33
18
85
22

 The sorted array is:
 12	18	22	33	85	88
```

11. Write a program to implement the graph traversal methods such as BFS and DFS.

```
#include
#define MAX 10
void breadth_first_search(int adj[][MAX],int visited[],int start)
{
 int queue[MAX],rear = –1,front =– 1, i;
queue[++rear] = start;
 visited[start] = 1;
 while(rear != front)
 {
 start = queue[++front];
if(start == 4)
printf("5\t");
 else
 printf("%c \t",start + 65);
 for(i = 0; i < MAX; i++)
 {
 if(adj[start][i] == 1 && visited[i] == 0)
 {
 queue[++rear] = i; visited[i] = 1;
}
 }
 }
 }
 int main()
```

```
{
int visited[MAX] = {0};
            int adj[MAX][MAX], i, j;
printf("\n Enter the adjacency matrix: ");
            for(i = 0; i < MAX; i++)
                        for(j = 0; j < MAX; j++)
 scanf("%d", &adj[i][j]);
breadth_first_search(adj,visited,0);
            return 0;
}
```

 Output:

```
Output
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
A B D C E
```

```
#include <stdio.h>
#define MAX 5
void depth_first_search(int adj[][MAX],int visited[],int start)
{
            int stack[MAX];
int top = –1, i;
printf("%c–",start + 65);
visited[start] = 1;
stack[++top] = start;
while(top ! = –1)
{
 start = stack[top];
                        for(i = 0; i < MAX; i++)
 {
            if(adj[start][i] && visited[i] == 0)
 {
```

```
 stack[++top] = i;
printf("%c–", i + 65);
visited[i] = 1;
break;
 }
 }
                    if(i == MAX)
 top—;
}
}
int main()
{
          int adj[MAX][MAX];
          int visited[MAX] = {0}, i, j;
printf("\n Enter the adjacency matrix: ");
          for(i = 0; i < MAX; i++)
                    for(j = 0; j < MAX; j++)
 scanf("%d", &adj[i][j]);
printf("DFS Traversal: ");
          depth_first_search(adj,visited,0);
printf("\n");
          return 0;
}
```

Output

```
Enter the adjacency matrix:
0 1 0 1 0
1 0 1 1 0
0 1 0 0 1
1 1 0 0 1
0 0 1 1 0
DFS Traversal: A –> C –> E –>
```

12. Write a program to implement the Knuth-Morris-Pratt pattern matching algorithm.

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

void computeLPSArray(char* pat, int M, int* lps);

// Prints occurrences of txt[] in pat[]
void KMPSearch(char* pat, char* txt)
{
	int M = strlen(pat);
	int N = strlen(txt);

	// create lps[] that will hold the longest prefix suffix
	// values for pattern
	int lps[M];

	// Preprocess the pattern (calculate lps[] array)
	computeLPSArray(pat, M, lps);

	int i = 0; // index for txt[]
	int j = 0; // index for pat[]
	while (i < N) {
		if (pat[j] == txt[i]) {
			j++;
			i++;
		}

		if (j == M) {
			printf("Found pattern at index %d \n", i - j );
			j = lps[j - 1];
		}

		// mismatch after j matches
		else if (i < N && pat[j] != txt[i]) {
			// Do not match lps[0..lps[j-1]] characters,
                // they will match anyway
                if (j != 0)
                    j = lps[j - 1];
                else
                    i = i + 1;
            }
        }
}

// Fills lps[] for given pattern pat[0..M-1]
void computeLPSArray(char* pat, int M, int* lps)
{
    // length of the previous longest prefix suffix
    int len = 0;

    lps[0] = 0; // lps[0] is always 0

    // the loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < M) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }
        else // (pat[i] != pat[len])
        {
            // This is tricky. Consider the example.
            // AAACAAAA and i = 7. The idea is similar
            // to search step.
            if (len != 0) {
                len = lps[len - 1];

                // Also, note that we do not increment
                // i here
            }
            else // if (len == 0)
            {
                lps[i] = 0;
                i++;
```

```
                        }
                }
        }
}

// Driver program to test above function
int main()
{
        char txt[] = "AABAACAADAABAABA";
        char pat[] = "AABA";
        KMPSearch(pat, txt);
        return 0;
}
```

OUTPUT:

```
Found pattern at index 0
Found pattern at index 9
Found pattern at index 12
```