CS8381 – DATA STRUCTURES LABORATORY

Name of the Student : ……

Register Number : ……

Year / Semester / Section :……

Branch : ……

Department of Computer Science and Engineering

CHENNAI INSTITUTE OF TECHNOLOGY

Transforming Lives

CHENNAI INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Anna University Chennai), Sarathy Nagar, Kundrathur, Chennai - 600069.

CHENNAI INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Anna University Chennai), Sarathy Nagar, Kundrathur, Chennai - 600069.

Register No: | | | | | | | | | | | | |

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by Mr./Ms..... of …… semester B.E..... Engineering in the ..... Laboratory during the academic year .....

Staff – in – Charge **Head of the Department**

Submitted for the University Practical Examination held on .....

Internal Examiner
Date: ……

External Examiner
Date: ……...

COURSE OBJECTIVES AND OUTCOMES

OBJECTIVES

- To implement linear and non-linear data structures
- To understand the different operations of search trees
- To implement graph traversal algorithms
- To get familiarized to sorting and searching algorithms

1. Array implementation of Stack and Queue ADTs
2. Array implementation of List ADT
3. Linked list implementation of List, Stack and Queue ADTs
4. Applications of List, Stack and Queue ADTs
5. Implementation of Binary Trees and operations of Binary Trees
6. Implementation of Binary Search Trees
7. Implementation of AVL Trees
8. Implementation of Heaps using Priority Queues.
9. Graph representation and Traversal algorithms
10. Applications of Graphs
11. Implementation of searching and sorting algorithms
12. Hashing – any two collision techniques

OUTCOMES:

At the end of the course, the students will be able to:

- Write functions to implement linear and non-linear data structure operations
- Suggest appropriate linear / non-linear data structure operations for solving a given problem
- Appropriately use the linear / non-linear data structure operations for a given problem
- Apply appropriate hash functions that result in a collision free scenario for data storage and retrieval.

INDEX

EXPERIMENT NO	DATE OF EXPERIMENT	TITLE	DATE OF COMPLETION	SIGNATURE
1		Array implementation of stack and Queue ADTs		
2		Array implementation of list ADT		
3		Linked list implementation of List,Stack and queue ADTs		
4		Application of List,Stack and Queue ADTs		
5		Implementation of Binary trees and operations of Binary trees		
6		Implementation of Binary Search Tree		
7		Implemenation of AVL Tree		
8		Implementation of Heaps using Priority Queues		
9		Graph representation and traversal algorithms		
10		Applications of Graphs		
11		Implementation of searching and sorting alogorithms		
12		Hashing –any two collision techniques.		

CONTENT BEYOND THE SYLLABUS

EXPERIMENT NO	DATE OF EXPERIMENT	TITLE	DATE OF COMPLETION	SIGNATURE
13		Dijkstra's Algorithm		
14		Travelling salesman problem		

EX.NO:1(a)	ARRAY IMPLEMENTATION OF STACK ADTs
DATE:	

AIM:

To write a C program to implement stack ADTs using array.

ALGORITHM:

STEP 1: Define a structure stack with an array variable which stores stack elements and and top variable which acting as a array index variable.

STEP 2: initialize the top variable as -1 to intimate the stack is empty in main function.

STEP 3: Define a functions called stackfull, stack empty for showing the status of the array.

i. if the top value is greater than the size of stack then the stack is full.

ii. if the top value is less than zero of the size of stack, then stack is empty.

STEP 4: Define the push function.

i. check the stackfull condition by calling function.

ii. if not, increment the top value by one

iii. assign the element into the top specified location.

STEP 5: Define the pop function.

i. Check the stack empty by calling the function.

ii. if not, decrement the top value by one.

STEP 6: Define the display function to display the elements from the postion top to the index 0.

PROGRAM

```
#include<stdio.h>
 int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   //clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
```

```
    printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
    do
    {
       printf("\n Enter the Choice:");
       scanf("%d",&choice);
       switch(choice)
       {
          case 1:
          {
             push();
             break;
          }
          case 2:
          {
             pop();
             break;
          }
          case 3:
          {
             display();
             break;
          }
          case 4:
          {
             printf("\n\t EXIT POINT ");
             break;
          }
          default:
          {
             printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
          } }    }
    while(choice!=4);
    return 0;
}
void push()
{
    if(top>=n-1)
    {
       printf("\n\tSTACK is over flow");

    }
```

```
    else
    {
       printf(" Enter a value to be pushed:");
       scanf("%d",&x);
       top++;
       stack[top]=x;
    }  }
void pop()
{
    if(top<=-1)
    {
       printf("\n\t Stack is under flow");
    }
    else
    {
       printf("\n\t The popped elements is %d",stack[top]);
       top--;
    }      }
void display()
{
    if(top>=0)
    {
       printf("\n The elements in STACK \n");
       for(i=top; i>=0; i--)
          printf("\n%d",stack[i]);
       printf("\n Press Next Choice");
    }
    else
    {
       printf("\n The STACK is empty");
    }      }
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:1(b)	ARRAY IMPLEMENTATION OF QUEUE ADTs
DATE:	

AIM:

To write a C program to implement Queue ADTs using array.

ALGORITHM:

STEP 1: Define a structure queue with an array variable which stores queue elements and and rear and front variable which acting as a array index variable.

STEP 2: initialize the two index variables rear and front variable as -1 to denote the queue is empty in main function.

STEP 3: Define a functions called queuefull, queue empty for showing the status of the array.

i. if the front value is greater than the size of queue then the queue is full.

ii. if the rear value is less than zero of the size of queue, then queue is empty.

STEP 4: Define the push function.

i. check the queuefull condition by calling function.

ii. if not, increment the rear variable by one. If it is a first element of the queue increment front value by one.

iii. assign the element into the rear specified location.

STEP 5: Define the pop function.

i. Check the queue empty by calling the function.

ii. if not, increment the front value by one.

STEP 6. Define the display function to display the elements from the postion front to The rear specified array location.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define n 5
void main()
{
   int queue[n],ch=1,front=0,rear=0,i,j=1,x=n;
   //clrscr();
   printf("Queue using Array");
   printf("\n1.Insertion \n2.Deletion \n3.Display \n4.Exit");
```

```
while(ch)
{
   printf("\nEnter the Choice:");
   scanf("%d",&ch);
   switch(ch)
   {
   case 1:
      if(rear==x)
         printf("\n Queue is Full");
      else
      {
         printf("\n Enter no %d:",j++);
         scanf("%d",&queue[rear++]);
      }
      break;
   case 2:
      if(front==rear)
      {
         printf("\n Queue is empty");
      }
      else
      {
         printf("\n Deleted Element is %d",queue[front++]);
         x++;
      }
      break;
   case 3:
      printf("\n Queue Elements are:\n ");
      if(front==rear)
         printf("\n Queue is Empty");
      else
      {
         for(i=front; i<rear; i++)
         {
            printf("%d",queue[i]);
            printf("\n");
         }
         break;
      case 4:
         exit(0);
      default:
```

```
            printf("Wrong Choice: please see the options");
        }               }     }
   getch();
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:2	ARRAY IMPLEMENTATION OF LIST ADT
DATE:	

AIM:

To write a c program to implement list ADT using array.

ALGORITHM:

STEP 1: Declare an array variable as list and array index variable.
STEP 2: Declare the necessary functions for creating, inserting, deleting and searching an element to the list.
STEP 3: Define the create function to insert n number of elements into the array.
STEP 4: Define the insert function to insert an element into the specific position of the array.
STEP 5: Define the delete function by replacing the other elements from the deleting element location.
STEP 6: Define the Search function by comparing the elements of the list with the search elements.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int a[100],n,i;
void create();
void display();
void search();
void sort();
void insert();
void delet();
void reverse();
void main()
{
        int op;
        clrscr();
        printf("ARRAY IMPLEMENTATION OF ORDER LIST");
        printf("\n**************************************");
        printf ("\n\t1.CREATE\n\t2.DISPLAY\n\t3.SEARCH\n\t4.SORT\n\t5.INSERT\n\t
        6. DEL ETE\n\t7.REVERSE\n\t8.EXIT");
        printf("\n ENTER THE NUMBER OF ARRAY ELEMENTS:");
```

```
        scanf("%d",&n);
        do
        {
                printf("\n ENTER THE OPTION:");
                scanf("%d",&op);
                switch(op)
                {
                        case 1:
                                create();
                                break;
                        case 2:
                                display();
                                break;
                        case 3:
                                search();
                                break;
                        case 4:
                                sort();
                                break;
                        case 5:
                                insert();
                                break;
                        case 6:
                                delet();
                                break;
                        case 7:
                                reverse();
                                break;
                        case 8:
                                exit(0);
                        default:
                        {
                                printf("\n INVALID OUTPUT");
                                break;
                        }
                }
        }while(op!=8);
        getch();
}
void create()
{
```

```
	printf("\n ENTER THE ARRAY ELEMENTS:");
	for(i=0;i<n;i++)
			scanf("%d",&a[i]);
}
void display()
{
	printf("\n THE ARRAY ELEMENTS ARE:");
	for(i=0;i<n;i++)
			printf("\n a[%d]=%d",i,a[i]);
}
void search()
{
	int m;
	printf("\n PLEASE ENTER THE ELEMENT TO SEARCH:");
	scanf("%d",&m);
	for(i=0;i<n;i++)
	{
		if(a[i]==m)
		{
			printf("THE ELEMENT FOUND AT POSITION:%d",i);
			break;
		}
	}
	if(i==n)
		printf("\n ELEMENT IS NOT FOUND");
}
void sort()
{
	int j,temp;
	for(i=0;i<n;i++)
	{
		for(j=i+1;j<n;j++)
		{
			if(a[i]>a[j])
			{
				temp=a[i];
				a[i]=a[j];
				a[j]=temp;
			}
		}
	}
```

```
	printf("\n THE SORTED ARRAY IS: ");
	for(i=0;i<n;i++)
		printf("\n\ta[%d]=%d",i,a[i]);
}
void insert()
{
	int pos,m;
	printf("\n ENTER THE POSITION TO INSERT AN ELEMENT:");
	scanf("%d",&pos);
	printf("\n ENTER THE ELEMENT TO INSERT:");
	scanf("%d",&m);
	for(i=n-1;i>=pos-1;i--)
		a[i+1]=a[i];
	a[pos-1]=m;
	n++;
}
void delet()
{
	int m,pos;
	printf("\n ENTER THE ELEMENT TO BE DELETED:");
	scanf("%d",&m);
	for(i=0;i<n;i++)
	{
		if(a[i]==m)
		{
			pos=i;
			break;
		}
	}
	if(i==n)
		pos=-1;
	for(i=pos;i<n-1;i++)
		a[i]=a[i+1];
	n--;
}
void reverse()
{
	int i;
	printf("\n THE LIST IN REVERSE ORDER IS:");
	for(i=n-1;i>=0;i--)
		printf("\n%d",a[i]);}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:3(a)	LINKED LIST IMPLEMENTATION OF QUEUE ADTs
DATE:	

AIM:

To implement C program using Linked List implementation of queue ADT.

ALGORITHM:

STEP 1: Include all the header files which are used in the program. And declare all the user defined functions.

STEP 2: Define a 'Node' structure with two members data and next.

STEP 3: Define a Node pointer 'top' and set it to NULL.

STEP 4: Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

STEP 5: Increment the value as 1.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;
 int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();
 int count = 0;
 void main()    {
```

```
int no, ch, e;
printf("\n 1 - Enque");
printf("\n 2 - Deque");
printf("\n 3 - Front element");
printf("\n 4 - Empty");
printf("\n 5 - Exit");
printf("\n 6 - Display");
printf("\n 7 - Queue size");
create();
while (1)
{
   printf("\n Enter choice : ");
   scanf("%d", &ch);
   switch (ch)  {
   case 1:
      printf("Enter data : ");
      scanf("%d", &no);
      enq(no);
      break;
   case 2:
      deq();
      break;
   case 3:
      e = frontelement();
      if (e != 0)
         printf("Front element : %d", e);
      else
         printf("\n No front element in Queue as queue is empty");
      break;
   case 4:
      empty();
      break;
   case 5:
      exit(0);
   case 6:
      display();
      break;
   case 7:
      queuesize();
      break;
   default:
```

```
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }     } }
 void create( )  {
    front = rear = NULL;
}
void queuesize( )        {
    printf("\n Queue size : %d", count);
}
 void enq(int data)      {
    if (rear == NULL)
    {
        rear = (struct node *)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->info = data;
        front = rear;
    }
    else
    {
        temp=(struct node *)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->info = data;
        temp->ptr = NULL;
        rear = temp;
    }
    count++;     }
void display()
{
    front1 = front;
 if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty");
        return;         }
    while (front1 != rear)
    {
        printf("%d ", front1->info);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("%d", front1->info);
}
```

```
void deq()
{
   front1 = front;
   if (front1 == NULL)
   {
      printf("\n Error: Trying to display elements from empty queue");
      return;
   }
   else
      if (front1->ptr != NULL)
      {
         front1 = front1->ptr;
         printf("\n Dequed value : %d", front->info);
         free(front);
         front = front1;
      }
      else
      {
         printf("\n Dequed value : %d", front->info);
         free(front);
         front = NULL;
         rear = NULL;
      }
      count--;
}
int frontelement()
{
   if ((front != NULL) && (rear != NULL))
      return(front->info);
   else
      return 0;
}
void empty()
{
    if ((front == NULL) && (rear == NULL))
      printf("\n Queue empty");
   else
     printf("Queue not empty");
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:3(b)	LINKED LIST IMPLEMENTATION OF LIST ADTs
DATE:	

AIM:

To write a C program to implement list ADT using linked list.

ALGORITHM:

STEP 1: Start
STEP 2: Read the value of ch
STEP 3: If ch=1 call create list functions
STEP 4: If ch=2 call insert list functions
STEP 5: If ch=3 call delete list functions
STEP 6: If ch=4 call view list functions
STEP 7: Repeat while (ch! =5)
STEP 8: Stop

Algorithm for Create List

STEP 1: Read value of item
STEP 2: Allocate the memory far new node
STEP 3: Assign values to new node data part
STEP 4: Assign new node address part as null

Algorithm for Insert List

STEP 1: Read the value of item
STEP 2: Allocate the memory far new node
STEP 3: Assign values of data field as null else make link fields of all new nodes to point
STEP 4: starting node to new node
STEP 5: Set the external pointer from the starting node to new node

Algorithm or Delete List

STEP 1: If the link is empty then return else check whether the list contains more than one element
STEP 2: Move the start pointer to the next node
STEP 3: Free the first node

Algorithm for View List

STEP 1: Using the for loop i
STEP 2: Print the linked list
STEP 3: Stop

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct Node;
typedef struct Node * PtrToNode;
typedef PtrToNode List;
typedef PtrToNode Position;

struct Node
{
   int e;
   Position next;
};
void Insert(int x, List l, Position p)
{
   Position TmpCell;
   TmpCell = (struct Node*) malloc(sizeof(struct Node));
   if(TmpCell == NULL)
      printf("Memory out of space\n");
   else
   {
      TmpCell->e = x;
      TmpCell->next = p->next;
      p->next = TmpCell;
   }
}
int isLast(Position p)
{
   return (p->next == NULL);
}
Position FindPrevious(int x, List l)
```

```
{
   Position p = l;
   while(p->next != NULL && p->next->e != x)
      p = p->next;
   return p;
}
void Delete(int x, List l)
{
   Position p, TmpCell;
   p = FindPrevious(x, l);
   if(!isLast(p))
   {
      TmpCell = p->next;
      p->next = TmpCell->next;
      free(TmpCell);
   }
   else
      printf("Element does not exist!!!\n");
}
void Display(List l)
{
   printf("The list element are :: ");
   Position p = l->next;
   while(p != NULL)
   {
      printf("%d -> ", p->e);
      p = p->next;
   }
}
void Merge(List l, List l1)
{
   int i, n, x, j;
```

```
    Position p;
    printf("Enter the number of elements to be merged :: ");
    scanf("%d",&n);
    for(i = 1; i <= n; i++)
    {
       p = l1;
       scanf("%d", &x);
       for(j = 1; j < i; j++)
          p = p->next;
       Insert(x, l1, p);
    }
    printf("The new List :: ");
    Display(l1);
    printf("The merged List ::");
    p = l;
    while(p->next != NULL)
    {
       p = p->next;
    }
    p->next = l1->next;
    Display(l);
}
int main()
{
    int x, pos, ch, i;
    List l, l1;
    l = (struct Node *) malloc(sizeof(struct Node));
    l->next = NULL;
    List p = l;
    printf("LINKED LIST IMPLEMENTATION OF LIST ADT\n\n");
    do
    {
```

```
    printf("\n\n1. INSERT\t 2. DELETE\t 3. MERGE\t 4. PRINT\t 5. QUIT\n\nEnter the choice :: ");
    scanf("%d", &ch);
    switch(ch)
    {
    case 1:
       p = l;
       printf("Enter the element to be inserted :: ");
       scanf("%d",&x);
       printf("Enter the position of the element :: ");
       scanf("%d",&pos);
       for(i = 1; i < pos; i++)
       {
          p = p->next;
       }
       Insert(x,l,p);
       break;
    case 2:
       p = l;
       printf("Enter the element to be deleted :: ");
       scanf("%d",&x);
       Delete(x,p);
       break;
    case 3:
       l1 = (struct Node *) malloc(sizeof(struct Node));
       l1->next = NULL;
       Merge(l, l1);
       break;
    case 4:
       Display(l);
       break;
    }
}
```

```
    while(ch<5);
    return 0;
}
```

OUTPUT:

INFERENCE

RESULT:

EX.NO:3(a)	LINKED LIST IMPLEMENTATION OF STACK ADTs
DATE:	

AIM:

To demonstrate linked list implementation of stack using a C program.

ALGORITHM:

STEP 1: Create a structure called node with data and self referenced pointer variable next.

STEP 2: Create a pointer variable of type node called top to initialize into NULL.

STEP 3: Define the function push to insert the element into the stack.

i. Allocate a node amount of memory to create a new node
ii. Assign a data into the new node.
iii. If stack top is NULL
New node becomes the top
iv. Else define the newnode next as top and change the pointer top as newnode.

STEP 4: Define the function pop to delete the element from the stack.

v. Assign the top element address to the deleting node.
vi. Redfine the top pointer to the next of top.
vii. Deallocate the memory for the deleting node.

PROGRAM:

```
include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define CAPACITY 10000
// Define stack node structure
struct stack
{
   int data;
   struct stack *next;
} *top;
int size = 0;
void push(int element);
int  pop();
int main()
{
```

```
    int choice, data;
    while(1)
    {
        /* Menu */
        printf("-------------------------------------\n");
        printf("    STACK IMPLEMENTATION PROGRAM    \n");
        printf("-------------------------------------\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Size\n");
        printf("4. Exit\n");
        printf("-------------------------------------\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter data to push into stack: ");
                scanf("%d", &data);
                // Push element to stack
                push(data);
                break;
            case 2:
                data = pop();
                // If stack is not empty
                if (data != INT_MIN)
                    printf("Data => %d\n", data);
                break;
            case 3:
                printf("Stack size: %d\n", size);
                break;
            case 4:
                printf("Exiting from app.\n");
                exit(0);
                break;
            default:
                printf("Invalid choice, please try again.\n");
        }
        printf("\n\n");
    }
    return 0;
```

```
}
void push(int element)
{
    if (size >= CAPACITY)
    {
        printf("Stack Overflow, can't add more element to stack.\n");
        return;
    }
    struct stack * newNode = (struct stack *) malloc(sizeof(struct stack));
    newNode->data = element;
    newNode->next = top;
    top = newNode;
    size++;
    printf("Data pushed to stack.\n");
}
int pop()
{
    int data = 0;
    struct stack * topNode;
    if (size <= 0 || !top)
    {
        printf("Stack is empty.\n");
        return INT_MIN;
    }
    topNode = top;
    data = top->data;
    top = top->next;
    free(topNode);
    size--;
    return data;
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:4	APPLICATIONS OF LIST, STACK AND QUEUE ADTs

AIM:

To writet a C program implementing applications of list,stack and queue.

ALGORITHM:

Step 1: Include the header files
Step 2: Scan the Infix string from left to right.
Step 3: To Initialize an empty stack.
Step 4: If the scanned character is an operand, add it to the Postfix string.
Step 5: If the scanned character is an operator and if the stack is empty push the character to stack
Step 6: If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.

PROGRAM:

INFIX TO POSTFIX EXPRESSION:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define SIZE 50
char s[SIZE];
int top=-1;
push(char elem)
{
   s[++top]=elem;
}

char pop()
{
   return(s[top--]);
}

int pr(char elem)
{
```

```
    switch(elem)
    {
    case '#': return 0;
    case '(': return 1;
    case '+':
    case '-': return 2;
    case '*':
    case '/': return 3;
    }
}

void main()
{
    char infx[50],pofx[50],ch,elem;
    int i=0,k=0;
    clrscr();
    printf("\n\nRead the Infix Expression ? ");
    scanf("%s",infx);
    push('#');
    while( (ch=infx[i++]) != '\0')
    {
    if( ch == '(') push(ch);
    else
    if(isalnum(ch)) pofx[k++]=ch;
      else
    if( ch == ')')
    {
       while( s[top] != '(')
      pofx[k++]=pop();
       elem=pop();
    }
    else
    {
       while( pr(s[top]) >= pr(ch) )
      pofx[k++]=pop();
       push(ch);
    }
    }
    while( s[top] != '#')
  pofx[k++]=pop();
    pofx[k]='\0';
```

```
   printf("\n\nGiven Infix Expn: %s  Postfix Expn: %s\n",infx,pofx);
   getch();
}
```

POSTFIX TO INFIX

```
#include <stdio.h>
#include <stdlib.h>
int top = 10;
struct node
{
        char ch;
        struct node *next;
        struct node *prev;
}  *stack[11];
typedef struct node node;

void push(node *str)
{
        if (top <= 0)
        printf("Stack is Full ");
        else
        {
                stack[top] = str;
                top--;
        }
}

node *pop()
{
        node *exp;
        if (top >= 10)
                printf("Stack is Empty ");
        else
                exp = stack[++top];
        return exp;
}
void convert(char exp[])
{
        node *op1,  *op2;
        node *temp;
        int i;
```

```
	for (i=0;exp[i]!='\0';i++)
	if (exp[i] >= 'a'&& exp[i] <= 'z'|| exp[i] >= 'A' && exp[i] <= 'Z')
	{
		temp = (node*)malloc(sizeof(node));
		temp->ch = exp[i];
		temp->next = NULL;
		temp->prev = NULL;
		push(temp);
	}
	else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/' ||
exp[i] == '^')
	{
		op1 = pop();
		op2 = pop();
		temp = (node*)malloc(sizeof(node));
		temp->ch = exp[i];
		temp->next = op1;
		temp->prev = op2;
		push(temp);
	}
}

void display(node *temp)
{
	if (temp != NULL)
	{
		display(temp->prev);
		printf("%c", temp->ch);
		display(temp->next);
	}
}

void main()
{
	char exp[50];
	clrscr();
	printf("Enter the postfix expression :");
	scanf("%s", exp);
	convert(exp);
	printf("\nThe Equivalant Infix expression is:");
	display(pop());
```

```
        printf("\n\n");
        getch();
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:5	IMPLEMENTATION OF BINARY TREES AND OPERATIONS OF BINARY TREE
DATE:	

AIM:

To writet a C program implementing binary tree and its operation.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Include header files.
struct btnode
{
int value;
struct btnode *l;
struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1
STEP 3: Compare x with the middle element.
STEP 4: If x matches with middle element, we return the mid index.
STEP 5: Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
STEP 6: Else (x is smaller) recur for the left half.
STEP 7: Stop the program.

PROGRAM:

```
#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 preorder(struct btnode *t);
void postorder(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 - Preorder Traversal\n");
   printf("5 - Postorder Traversal\n");
   printf("6 - 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:
         preorder(root);
         break;
      case 5:
         postorder(root);
         break;
      case 6:
         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);
}

/* 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);
}
/* 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:

INFERENCE:

RESULT:

EX.NO:6	IMPLEMENTATION OF BINARY SERACH TREES
DATE:	

AIM:

To write a C program to implement binary search tree.

ALGORITHM:

STEP 1: Start the program.

STEP 2: Include header files.

Compare x with the middle element.

If x matches with middle element, we return the mid index.

A recursive binary search function. It returns

```
int binarySearch(int arr[], int l, int r, int x)
{
  if (r >= l)
  {
      int mid = l + (r - l)/2;
       if (arr[mid] == x)
          return mid;
      if (arr[mid] > x)
          return binarySearch(arr, l, mid-1, x);
       return binarySearch(arr, mid+1, r, x);
  }
  return -1;
}
```

STEP 3: Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.

STEP 4: Else (x is smaller) recur for the left half.

STEP 5: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int key;
   struct node *left, *right;
};
```

```
// A utility function to create a new BST node
struct node *newNode(int item)
{
   struct node *temp =  (struct node *)malloc(sizeof(struct node));
   temp->key = item;
   temp->left = temp->right = NULL;
   return temp;
}
void inorder(struct node *root)
{
   if (root != NULL)
   {
      inorder(root->left);
      printf("%d \n", root->key);
      inorder(root->right);
   }
}
  struct node* insert(struct node* node, int key)
{
     if (node == NULL) return newNode(key);
    if (key < node->key)
      node->left  = insert(node->left, key);
   else if (key > node->key)
      node->right = insert(node->right, key);
    return node;
}

int main()

{
   struct node *root = NULL;
   root = insert(root, 50);
   insert(root, 30);
   insert(root, 20);
   insert(root, 40);
   insert(root, 70);
   insert(root, 60);
   insert(root, 80);
   inorder(root);
   return 0;
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:7	IMPLEMENTATION OF AVL TREE
DATE:	

AIM:

To write a C program for AVL Tree implementation.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Include header files.
STEP 3: Insert the new element into the **tree** using **Binary** Search **Tree** insertion logic.
STEP 4: After insertion, check the Balance Factor of every node.
STEP 5: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
STEP 6: Stop the program.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
        int key;
        struct Node *left;
        struct Node *right;
        int height;
};
int max(int a, int b);
int height(struct Node *N)
{
        if (N == NULL)
                return 0;
        return N->height;
}
int max(int a, int b)
{
        return (a > b)? a : b;
}       NULL left and right pointers. */
struct Node* newNode(int key)
{
```

```
        struct Node* node = (struct Node*) malloc(sizeof(struct Node));
        node->key = key;
        node->left = NULL;
        node->right = NULL;
        node->height = 1; // new node is initially added at leaf
        return(node);
} struct Node *rightRotate(struct Node *y)
{
        struct Node *x = y->left;
        struct Node *T2 = x->right;
        x->right = y;
        y->left = T2;
        y->height = max(height(y->left), height(y->right))+1;
        x->height = max(height(x->left), height(x->right))+1;
        return x;
}
struct Node *leftRotate(struct Node *x)
{
        struct Node *y = x->right;
        struct Node *T2 = y->left;
        y->left = x;
        x->right = T2;
        x->height = max(height(x->left), height(x->right))+1;
        y->height = max(height(y->left), height(y->right))+1;
        return y;
}
int getBalance(struct Node *N)
{
        if (N == NULL)
                return 0;
        return height(N->left) - height(N->right);
}
struct Node* insert(struct Node* node, int key)
{
if (node == NULL)
                return(newNode(key));

        if (key < node->key)
                node->left = insert(node->left, key);
        else if (key > node->key)
                node->right = insert(node->right, key);
```

```
        else // Equal keys are not allowed in BST
                return node;
        node->height = 1 + max(height(node->left), height(node->right));
        int balance = getBalance(node);
        if (balance > 1 && key < node->left->key)
                return rightRotate(node);
        if (balance < -1 && key > node->right->key)
                return leftRotate(node
        if (balance > 1 && key > node->left->key)
        {
                node->left = leftRotate(node->left);
                return rightRotate(node);
        }
        if (balance < -1 && key < node->right->key)
        {
                node->right = rightRotate(node->right);
                return leftRotate(node);
        }
return node;
}
void preOrder(struct Node *root)
{
        if(root != NULL)
        {
                printf("%d ", root->key);
                preOrder(root->left);
                preOrder(root->right);
        }
}
int main()
{
struct Node *root = NULL;
root = insert(root, 10);
root = insert(root, 20);
root = insert(root, 30);
root = insert(root, 40);
root = insert(root, 50);
root = insert(root, 25);
printf("Preorder traversal of the constructed AVL"
                " tree is \n");
preOrder(root);
```

```
return 0;
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:8	IMPLEMENTATIONS OF HEAPS USING PRIORITY QUEUES
DATE:	

AIM:

To write a C program to implement heaps using priority queues..

ALGORITHM:

STEP 1: Start the program

STEP 2: Include header files.

PUSH(HEAD, DATA, PRIORITY)
Create new node with DATA and PRIORITY
Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.:
NEW -> NEXT = HEAD
HEAD = NEW
Set TEMP to head of the list
While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY
TEMP = TEMP -> NEXT
[END OF LOOP]
NEW -> NEXT = TEMP -> NEXT
TEMP -> NEXT = NEW
End
POP(HEAD)
Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.
Free the node at the head of the list
End

STEP 3 : Stop the program,

PROGRAM:

```
#include<stdio.h>
#include<malloc.h>
void insert();
void del();
void display();
struct node
{
int priority;
int info;
```

```
struct node *next;
}*start=NULL,*q,*temp,*new;
typedef struct node N;
int main()
{
int ch;
do
{
printf("\n[1] INSERTION\t[2] DELETION\t[3] DISPLAY [4] EXIT\t:");
scanf("%d",&ch);
switch(ch)
{
case 1:insert();
break;
case 2:del();
break;
case 3:display();
break;
case 4:
break;
}
}
while(ch<4);
}
void insert()
{
int item,itprio;
new=(N*)malloc(sizeof(N));
printf("ENTER THE ELT.TO BE INSERTED :\t");
scanf("%d",&item);
printf("ENTER ITS PRIORITY :\t");
scanf("%d",&itprio);
new->info=item;
new->priority=itprio;
new->next=NULL;
if(start==NULL )
{
//new->next=start;
start=new;
}
else if(start!=NULL&&itprio<=start->priority)
```

```
{  new->next=start;
start=new;
}
else
{
q=start;
while(q->next != NULL && q->next->priority<=itprio)
{q=q->next;}
new->next=q->next;
q->next=new;
}
}
void del()
{
if(start==NULL)
{
printf("\nQUEUE UNDERFLOW\n");

}
else
{
new=start;
printf("\nDELETED ITEM IS %d\n",new->info);
start=start->next;
//free(start);
}
}
 void display()
{
temp=start;
if(start==NULL)
printf("QUEUE IS EMPTY\n");
else
{
printf("QUEUE IS:\n");
if(temp!=NULL)
for(temp=start;temp!=NULL;temp=temp->next)
{
printf("\n%d priority =%d\n",temp->info,temp->priority);
//temp=temp->next;
}}}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:9	GRAPH REPRESENTATION AND TRAVERSAL ALGORITHMS
DATE:	

AIM:

To write a C program for implementation of graph representation and traversal .

ALGORITHM:

STEP 1: Start the program.
STEP 2: Include the header files
STEP 3: Declare the base class base
STEP 4: Call the BFS() function.
STEP 5: First move horizontally and visit all the nodes of the current layer over to the next layer
STEP 6: Call the DFS() function.
STEP 7: First move vertically and visit all the nodes of the current layer over to the next layer
STEP 8: Stop the program.

PROGRAM:

BFS:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue {
   int items[SIZE];
   int front;
   int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node
{
```

```
    int vertex;
    struct node* next;
};
struct node* createNode(int);
struct Graph
{
    int numVertices;
    struct node** adjLists;
    int* visited;
};
struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);
int main()
{
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    bfs(graph, 0);
    return 0;
}
void bfs(struct Graph* graph, int startVertex) {
    struct queue* q = createQueue();
    graph->visited[startVertex] = 1;
    enqueue(q, startVertex);
        while(!isEmpty(q)){
        printQueue(q);
        int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
      struct node* temp = graph->adjLists[currentVertex];
      while(temp) {
            int adjVertex = temp->vertex;
            if(graph->visited[adjVertex] == 0){
                graph->visited[adjVertex] = 1;
                enqueue(q, adjVertex);
```

```
        }           temp = temp->next;
    }    } }

struct node* createNode(int v)
{
   struct node* newNode = malloc(sizeof(struct node));
   newNode->vertex = v;
   newNode->next = NULL;
   return newNode;
}
struct Graph* createGraph(int vertices)
{
   struct Graph* graph = malloc(sizeof(struct Graph));
   graph->numVertices = vertices;
   graph->adjLists = malloc(vertices * sizeof(struct node*));
   graph->visited = malloc(vertices * sizeof(int));
   int i;
   for (i = 0; i < vertices; i++) {
   graph->adjLists[i] = NULL;
      graph->visited[i] = 0;
   }
   return graph;
}
void addEdge(struct Graph* graph, int src, int dest)
{
   // Add edge from src to dest
   struct node* newNode = createNode(dest);
   newNode->next = graph->adjLists[src];
   graph->adjLists[src] = newNode;
   // Add edge from dest to src
   newNode = createNode(src);
   newNode->next = graph->adjLists[dest];
   graph->adjLists[dest] = newNode;
}
struct queue* createQueue() {
   struct queue* q = malloc(sizeof(struct queue));
   q->front = -1;
   q->rear = -1;
   return q;
}
```

```
int isEmpty(struct queue* q) {
   if(q->rear == -1)
      return 1;
   else
      return 0;}
void enqueue(struct queue* q, int value){
   if(q->rear == SIZE-1)
      printf("\nQueue is Full!!");
   else {
      if(q->front == -1)
         q->front = 0;
      q->rear++;
      q->items[q->rear] = value;
   }
}
int dequeue(struct queue* q){
   int item;
   if(isEmpty(q)){
      printf("Queue is empty");
      item = -1;
   }
   else{
      item = q->items[q->front];
      q->front++;
      if(q->front > q->rear){
         printf("Resetting queue");
         q->front = q->rear = -1;
      }
   }
   return item;
}
void printQueue(struct queue *q) {
   int i = q->front;

   if(isEmpty(q)) {
      printf("Queue is empty");
   } else {
      printf("\nQueue contains \n");
      for(i = q->front; i < q->rear + 1; i++) {
            printf("%d ", q->items[i]);
      }
```

}}

DFS:

```
#include <stdio.h>

#include <stdlib.h>

struct node

{

   int vertex;

   struct node* next;

};

struct node* createNode(int v);

struct Graph

   int numVertices;

   int* visited;

   struct node** adjLists; // we need int** to store a two dimensional array. Similary, we need struct node** to store an array of Linked lists

};struct Graph* createGraph(int);

void addEdge(struct Graph*, int, int);

void printGraph(struct Graph*);

void DFS(struct Graph*, int);

int main()

{   struct Graph* graph = createGraph(4);

   addEdge(graph, 0, 1);

   addEdge(graph, 0, 2);

   addEdge(graph, 1, 2);

   addEdge(graph, 2, 3);

   printGraph(graph);
```

```
    DFS(graph, 2);
    return 0;
}oid DFS(struct Graph* graph, int vertex) {
        struct node* adjList = graph->adjLists[vertex];
        struct node* temp = adjList;
        graph->visited[vertex] = 1;
        printf("Visited %d \n", vertex);
        while(temp!=NULL) {
            int connectedVertex = temp->vertex;
            if(graph->visited[connectedVertex] == 0) {
                DFS(graph, connectedVertex);        }
            temp = temp->next;
        }  }
struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
     graph->adjLists = malloc(vertices * sizeof(struct node*));
```

```
    graph->visited = malloc(vertices * sizeof(int));
    int i;
    for (i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add edge from src to dest
    struct node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;
    // Add edge from dest to src
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}
void printGraph(struct Graph* graph)
{
    int v;
    for (v = 0; v < graph->numVertices; v++)
    {
        struct node* temp = graph->adjLists[v];
```

```
        printf("\n Adjacency list of vertex %d\n ", v);
        while (temp)
        {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:10	APPLICATIONS OF GRAPHS
DATE:	

AIM:

To write a C program for executing topological sorting in graph.

ALGORITHM:

STEP 1: Choose a vertex in a graph without any predecessors. In other words, it is a vertex with Zero Indegree.

STEP 2: Delete this vertex and all the outgoing edges from it.

STEP 3: Repeat the above process till all the vertices are not deleted from the DAG.

PROGRAM:

```
#include <stdio.h>
int main(){
int i,j,k,n,a[10][10],indeg[10],flag[10],count=0;
printf("Enter the no of vertices:\n");
scanf("%d",&n);
printf("Enter the adjacency matrix:\n");
for(i=0;i<n;i++){
printf("Enter row %d\n",i+1);
for(j=0;j<n;j++)
scanf("%d",&a[i][j]);
}
for(i=0;i<n;i++){
      indeg[i]=0;
      flag[i]=0;
   }
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
         indeg[i]=indeg[i]+a[j][i];
   printf("\nThe topological order is:");
   while(count<n){
      for(k=0;k<n;k++){
         if((indeg[k]==0) && (flag[k]==0)){
            printf("%d ",(k+1));
            flag [k]=1;
         }
```

```
            for(i=0;i<n;i++){
                if(a[i][k]==1)
                    indeg[k]--;
            }
        }

        count++;
    }

    return 0;
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:11	IMPLEMENTATION OF SEARCHING AND SORTING ALGORITHMS
DATE:	

AIM:

To write a C program to perform searching and sorting algorithms.

ALGORITHM:

STEP 1. Start the program.

STEP 2. Write two functions named linear() and binary().

STEP 3. Read the elements in the array

STEP 4. Linear search

Read the search element in the array

Check the search element with all the elements in the array if it is found means print element found otherwise print element not found

STEP 5.Binary search

Arrange the elements in ascending order

Find the mid position of the given array

Compare the search element to the mid element

If it is equal to the number we are searching then print element found

If it is less than move to the right.

If it is greater than move to the left.

Repeat the same step until element is found.

b) let ib=0 and ub=n-1

c) read the data to be searched X

d) find the mid position of the given array

mid=(ib+ub)/2

e) compare X with a[mid]

if equal then

goto step (g)

else

if X less than a[mid] then ub=mid-1

PROGRAM:

Binary Search

```
#include <stdio.h>
#include <conio.h>
void main( )
```

```
{
int m,i, n, j, temp, flg, low, middle, high, a[100];char ch;
clrscr( );
flg=0;
printf("Enter the size of the array \n");
scanf("%d",&n);
printf("Enter the elements of the array \n");
for (i=1; i<=n;i++)
{
scanf("%d", &a[i]);
}
do
{
printf("\nEnter the number which you want to search :");
scanf("%d", &m);
printf ("\nPress 'l' for linear search and 'b' for binary search :");
ch=getch( );
switch(ch)
{
case 'l':
for (i=1; i<=n;i++)
{
          if(a[i]==m)
            {
            flg=1;
            break;
                  }
}
if(flg== 1)
{
printf("\nThe element is found and the position is: %d",i);
printf("\n");
}
else
printf("\nElement is not found \n");
break;
case 'b':
for (i=1; i<=n-1;i++)
{
for (j=i+1;j<=n;j++)
{
```

```
if(a[i]>a[j])
{
temp=a[i];
a[i]=a[j];
a[j]=temp;
}
}
}
printf("\nThe sorted list is \n");
for (i=1; i<=n;i++)
{
printf("%d ",a[i]);

}
low=1;
high=n;
while (low<=high)
{
middle=(low+high)/2;
if(a[middle] == m)
{
flg = 1;
break;
}
else
{
if(a[middle]<m)
{
low=middle+1;
}
else
{
high=middle-1;
}
}
}
if(flg == 1)
printf("\nThe element is found and the position is : %d", middle);
else
printf("\nThe element is not found \n");
break;
```

```
default:
printf("\nWrong selection \n");
break;
}
flushall();
printf("\nPress 'Y' to continue and any other key to discontinue : ");
ch=getch( );
}while (ch == 'Y');
getch( );
}
```

Merge Sort

```
#include&lt;stdio.h&gt;
#include&lt;conio.h&gt;

#define MAX_SIZE 5

void merge_sort(int, int);
void merge_array(int, int, int, int);

int arr_sort[MAX_SIZE];

int main() {
  int i;

  printf("Simple Merge Sort Example - Functions and Array\n");
  printf("\nEnter %d Elements for Sorting\n", MAX_SIZE);
  for (i = 0; i &lt; MAX_SIZE; i++)
    scanf("%d", &amp;arr_sort[i]);

  printf("\nYour Data   :");
  for (i = 0; i &lt; MAX_SIZE; i++) {
    printf("\t%d", arr_sort[i]);
  }

  merge_sort(0, MAX_SIZE - 1);

  printf("\n\nSorted Data :");
  for (i = 0; i &lt; MAX_SIZE; i++) {
    printf("\t%d", arr_sort[i]);
  }
```

```
  getch();

}

void merge_sort(int i, int j) {
  int m;

  if (i &lt; j) {
    m = (i + j) / 2;
    merge_sort(i, m);
    merge_sort(m + 1, j);
    // Merging two arrays
    merge_array(i, m, m + 1, j);
  }
}
void merge_array(int a, int b, int c, int d) {
  int t[50];
  int i = a, j = c, k = 0;

  while (i &lt;= b &amp;&amp; j &lt;= d) {
    if (arr_sort[i] &lt; arr_sort[j])
      t[k++] = arr_sort[i++];
    else
      t[k++] = arr_sort[j++];
  }
  //collect remaining elements
  while (i &lt;= b)
    t[k++] = arr_sort[i++];

  while (j &lt;= d)
    t[k++] = arr_sort[j++];

  for (i = a, j = 0; i &lt;= d; i++, j++)
    arr_sort[i] = t[j];
}
```

Quick Sort

```
#include <stdio.h>

void quick_sort(int[],int,int);
int partition(int[],int,int);
```

```
int main()
{
int a[50],n,i;
printf("How many elements?");
scanf("%d",&n);
printf("\nEnter array elements:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
quick_sort(a,0,n-1);
printf("\nArray after sorting:");
for(i=0;i<n;i++)
printf("%d ",a[i]);
return 0;
}

void quick_sort(int a[],int l,int u)
{
int j;
if(l<u)
{
j=partition(a,l,u);
quick_sort(a,l,j-1);
quick_sort(a,j+1,u);
}
}

int partition(int a[],int l,int u)
{
int v,i,j,temp;
v=a[l];
i=l;
j=u+1;
do
{
do
i++;
while(a[i]<v&&i<=u);
do
j--;
```

```
while(v<a[j]);
if(i<j)
{
temp=a[i];
a[i]=a[j];

}
}while(i<j);
a[l]=a[j];
a[j]=v;
return(j);
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:12	HASHING ANY TWO COLLISION TECHNIQUES
DATE:	

AIM:

To implement a C program to analyse hashing method with two different collision techniques.

ALGORITHM:

STEP 1.Include the header files
STEP 2.Declare the basic variables
STEP 3.Declare and define the necessary function for searching
STEP 4. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.
STEP 5.Declare and define the function display().
STEP 6.Find the given input is presented or not in the list.

PROGRAM:

SEPARATE CHAINING:

hashing method with two different collision techniques

```
#include<stdio.h>
#include<stdlib.h>
#define size 7
struct node
   int data;
   struct node *next;
};
struct node *chain[size];
void init()
{
   int i;
   for(i = 0; i < size; i++)
      chain[i] = NULL;
}
void insert(int value)
{
   //create a newnode with value
   struct node *newNode = malloc(sizeof(struct node));
   newNode->data = value;
   newNode->next = NULL;
   //calculate hash key
```

```
    int key = value % size;
    //check if chain[key] is empty
    if(chain[key] == NULL)
        chain[key] = newNode;
    //collision
    else
    {
        //add the node at the end of chain[key].
        struct node *temp = chain[key];
        while(temp->next)
        {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}
void print()
{
    int i;
    for(i = 0; i < size; i++)
    {
        struct node *temp = chain[i];
        printf("chain[%d]-->",i);
        while(temp)
        {
            printf("%d -->",temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

int main()
{
    //init array of list to NULL
    init();

    insert(7);
    insert(0);
    insert(3);
    insert(10);
    insert(4);
    insert(5);
    print();
    return 0;
}
```

DOUBLE HASHING:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
struct data
{
        int key;
        int value;
};
 struct hashtable_item
{

        int flag;
        /*
         * flag = 0 : data not present
         * flag = 1 : some data already present
         * flag = 2 : data was present,but deleted
        */
        struct data *item;
 };
struct hashtable_item *array;
int max = 7;
int size = 0;
int prime = 3;
int hashcode1(int key)
{
        return (key % max);
}
 int hashcode2(int key)
{
        return (prime - (key % prime));
}
 void insert(int key, int value)
{
        int hash1 = hashcode1(key);
        int hash2 = hashcode2(key);
        int index = hash1;
 /* create new data to insert */
struct data *new_item = (struct data*) malloc(sizeof(struct data));
        new_item->key = key;
        new_item->value = value;
```

```
        if (size == max)
    {
            printf("\n Hash Table is full, cannot insert more items \n");
            return;
        }
        /* probing through other array elements */
        while (array[index].flag == 1) {
            if (array[index].item->key == key)
          {
                printf("\n Key already present, hence updating its value \n");
                array[index].item->value = value;
                return;
            }
            index = (index + hash2) % max;
            if (index == hash1)
          {
                printf("\n Add is failed \n");
                return;
            }
            printf("\n probing \n")
}
        array[index].item = new_item;
        array[index].flag = 1;
        size++;
        printf("\n Key (%d) has been inserted \n", key);
}

/* to remove an element from the array */
void remove_element(int key)
{
        int hash1 = hashcode1(key);
        int hash2 = hashcode2(key);
        int index = hash1;
        if (size == 0)
    {
            printf("\n Hash Table is empty \n");
            return;
        }

        /* probing through other elements */
```

```
        while (array[index].flag != 0)
    {
            if (array[index].flag == 1  &&  array[index].item->key == key)
        {
                array[index].item = NULL;
                array[index].flag = 2;
                size--;
                printf("\n Key (%d) has been removed \n", key);
                return;
            }
            index = (index + hash2) % max;
            if (index == hash1)
        {
            break;
            }
        }
        printf("\n Key (%d) does not exist \n", key);
}
int size_of_hashtable()
{
        return size;
}

/* displays all elements of array */
void display()
{
        int i;
        for (i = 0; i < max; i++)
    {
            if (array[i].flag != 1)
        {
                printf("\n Array[%d] has no elements \n", i);
            }
            else
        {
                printf("\n Array[%d] has elements \n Key (%d) and Value (%d)\n", i,
array[i].item->key, array[i].item->value);
            }
        }
}
```

```
/* initializes array */
void init_array()
{
        int i;
        for(i = 0; i < max; i++)
    {
                array[i].item = NULL;
                array[i].flag = 0;
        }
        prime = get_prime();
}

/* returns largest prime number less than size of array */
int get_prime()
{
        int i,j;
        for (i = max - 1; i >= 1; i--)
    {
                int flag = 0;
                for (j = 2; j <= (int)sqrt(i); j++)
            {
                        if (i % j == 0)
                {
                                flag++;
                        }
                }
                if (flag == 0)
            {
                        return i;
                }
        }
        return 3;

}

void main()
{
        int choice, key, value, n, c;
        clrscr();

        array = (struct hashtable_item*) malloc(max * sizeof(struct hashtable_item));
```

```
        init_array();

        do {
                printf("Implementation of Hash Table in C with Double Hashing.\n\n");
                printf("MENU-: \n1.Inserting item in the Hash Table"
                    "\n2.Removing item from the Hash Table"
                    "\n3.Check the size of Hash Table"
                    "\n4.Display Hash Table"
                     "\n\n Please enter your choice-:");

                scanf("%d", &choice);

                switch(choice)
           {    case 1:
                    printf("Inserting element in Hash Table\n");
                    printf("Enter key and value-:\t");
                    scanf("%d %d", &key, &value);
                    insert(key, value);
                    break;
                case 2:
                    printf("Deleting in Hash Table \n 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("Wrong Input\n");
                }
                printf("\n Do you want to continue-:(press 1 for yes)\t");
                scanf("%d", &c);
        }while(c == 1);
        getch();
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:13	DIJKSTRA'S ALGORITHM
DATE:	

AIM:

To implement C program for Dijikstra's Algorithm using adjacency matrix.

ALGORITHM:

STEP 1: Set the pathLength of all the vertices to Infinity, the predecessor of all the vertices to NIL and the status of all the vertices to temporary**.**

STEP 2: Set pathLength of source vertex to Zero**.**

STEP 3: Find the vertex that has the minimum value of pathLength by checking all the vertices in the graph and set its status as Permanent and consider it as the current vertex**.**

STEP 4: Check all the temporary vertices adjacent to the current vertex. Now, the value of the pathLength is recalculated for all these temporary successors of current vertex, and relabelling is done if needed.

STEP 5: Repeat the steps **3** and **4** until there is no temporary vertex remaining in the graph. In case if there are any vertices left, then they must have pathLength as Infinity**.**

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define INFINITY 9999
#define MAX 10

void dijkstra(int G[MAX][MAX],int n,int startnode);

int main()
{
int G[MAX][MAX],i,j,n,u;
printf("Enter no. of vertices:");
scanf("%d",&n);
printf("\nEnter the adjacency matrix:\n");
```

```
for(i=0;i<n;i++)
for(j=0;j<n;j++)
scanf("%d",&G[i][j]);
printf("\nEnter the starting node:");
scanf("%d",&u);
dijkstra(G,n,u);
return 0;
}

void dijkstra(int G[MAX][MAX],int n,int startnode)
{

int cost[MAX][MAX],distance[MAX],pred[MAX];
int visited[MAX],count,mindistance,nextnode,i,j;
//pred[] stores the predecessor of each node
//count gives the number of nodes seen so far
//create the cost matrix
for(i=0;i<n;i++)
for(j=0;j<n;j++)
if(G[i][j]==0)
cost[i][j]=INFINITY;
else
cost[i][j]=G[i][j];
//initialize pred[],distance[] and visited[]
for(i=0;i<n;i++)
{
distance[i]=cost[startnode][i];
pred[i]=startnode;
visited[i]=0;
}
distance[startnode]=0;
visited[startnode]=1;
count=1;
while(count<n-1)
{
mindistance=INFINITY;
//nextnode gives the node at minimum distance
for(i=0;i<n;i++)
if(distance[i]<mindistance&&!visited[i])
{
mindistance=distance[i];
nextnode=i;
}
//check if a better path exists through nextnode
visited[nextnode]=1;
for(i=0;i<n;i++)
if(!visited[i])
```

```
if(mindistance+cost[nextnode][i]<distance[i])
{
distance[i]=mindistance+cost[nextnode][i];
pred[i]=nextnode;
}
count++;
}

//print the path and distance of each node
for(i=0;i<n;i++)
if(i!=startnode)
{
printf("\nDistance of node%d=%d",i,distance[i]);
printf("\nPath=%d",i);
j=i;
do
{
j=pred[j];
printf("<-%d",j);
}while(j!=startnode);
}
}
```

OUTPUT:

INFERENCE:

RESULT:

EX.NO:14	TRAVELLING SALESMAN PROBLEM
DATE:	

AIM:

To write a C program for Travelling salesman problem.

ALGORITHM:

STEP 1: Start the program.
STEP 2: Consider city 1 as the starting and ending point.
STEP 3: Generate all (n-1)! Permutations of cities.
STEP 4: Calculate cost of every permutation and keep track of minimum cost permutation.
STEP 5: Return the permutation with minimum cost.
STEP 6: Time Complexity: Θ(n!)
STEP 7: Stop the program.

PROGRAM:

```
#include<stdio.h>
int ary[10][10],completed[10],n,cost=0;
void takeInput()
{
int i,j;
 printf("Enter the number of villages: ");
scanf("%d",&n);
 printf("\nEnter the Cost Matrix\n");
 for(i=0;i < n;i++)
{
printf("\nEnter Elements of Row: %d\n",i+1);
 for( j=0;j < n;j++)
scanf("%d",&ary[i][j]);
 completed[i]=0;
}
 printf("\n\nThe cost list is:");
 for( i=0;i < n;i++)
```

```
{
printf("\n");
 for(j=0;j < n;j++)
printf("\t%d",ary[i][j]);
}
}

void mincost(int city)
{
int i,ncity;
 completed[city]=1;
 printf("%d--->",city+1);
ncity=least(city);
 if(ncity==999)
{
ncity=0;
printf("%d",ncity+1);
cost+=ary[city][ncity];
 return;
}
 mincost(ncity);
}
int least(int c)
{
int i,nc=999;
int min=999,kmin;

for(i=0;i < n;i++)
{
if((ary[c][i]!=0)&&(completed[i]==0))
if(ary[c][i]+ary[i][c] < min)
{
min=ary[i][0]+ary[c][i];
kmin=ary[c][i];
nc=i;
}
}
if(min!=999)
cost+=kmin;
return nc;
}
```

```
int main()
{
takeInput();
printf("\n\nThe Path is:\n");
mincost(0); //passing 0 because starting vertex
printf("\n\nMinimum cost is %d\n ",cost);
return 0;
}
```

OUTPUT:

INFERENCE:

RESULT: