ANNA UNIVERSITY

UNIVERSITY COLLEGE OF ENGINEERING DINDIGUL

DINDIGUL - 624 622

DEPARTMENT OF :.....

Name _____

Register No _____

Branch _____

Subject _____

ANNA UNIVERSITY

UNIVERSITY COLLEGE OF ENGINEERING DINDIGUL

DINDIGUL - 624 622

BONAFIDE CERTIFICATE

Register No. | |

ertified that this is a Bonafide Record of the Practical work done by

..... Reg. No. .....

of ..... Semester, B.E. .....

in the ..... Laboratory

during the Academic year 2024 - 2025

Staff In-charge **Head of the Department**

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

In University College of Engineering, Dindigul.

Internal Examiner **External Examiner**

INDEX

EXPT NO : 01 PAGE NO :05 DATE :	**SINGLY LINKED LIST**

AIM:

To write a c program to implement singly linked list.

ALGORITHM:

Node structure

Element	Pointer

Element-data
Pointer-point to the next variable

Insert at beginning:

- Get a new node (allocate space), and get the element to be inserted.
- Set the pointer of the new node to header.
- Set the current node as header.
- Display the content of the list.

Insert at end:

- Get a new node (allocate space), get the element to be inserted.
- Move the pointer to the last node of the list.
- Set the pointer of the last node to the new node.
- Display the content of the list.

Insert any value:

- Get a new node (allocate space), and get the element to be inserted.
- Move the pointer of the list to the node that contain element < than the given new element.
- Set the pointer of the current node to new node and set the pointer of the new node to current+1 node.
- Display the content of the list.

from beginning:

- Set the pointer of the head node as the header.
- Display the content of the list.

Delete from end:

- Move the pointer of the list up to just before the last node.
- Set the pointer of the (n-1)th node as NULL.
- Display the content of the list.

Delete any element:

- Get the element to be deleted.
- Move the pointer of the list up to find the given element.
- If it is ith element then set the pointer of the (I-1)th element as (I+1). Display the content of the list.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
int i,j,m,n;
int create();
int deletion(int a1);
int insertion(int a2);
void display(int a3);
struct node
{
int data;
struct node *next;
}*head,*temp,*temp1;
void main()
{
int n=0,choice;
clrscr();
printf("\n SINGLY LINKED LIST");
while(1)
{
printf("\n 1.CREATE\n 2.INSERTION\n 3.DELETION\n 4.DISPLAY\n 5.EXIT");
printf("\n ENTER THE CHOICE:");
scanf("%d",&choice);
switch(choice)
{
case 1:
    n=create();
    break;
case 2:
    n=insertion(n);
    break;
case 3:
```

```
    n=deletion(n);
    break;
case 4:
    display(n);
    break;
case 5:
    exit(0);
    }getch();
}
}
int create()
{
int no,i;
head=malloc(sizeof(struct node));
head->next=NULL;
head->data=0;
printf("\nENTER THE NO. OF ELEMENTS");
scanf("%d",&no);
if (head->next==NULL)
{
printf("\nENTER THE ELEMENTS");
temp=malloc(sizeof(struct node));
scanf("%d",&temp->data);
head->next=temp;
temp->next=NULL;
for (i=1;i<no;i++)
{
temp1=malloc(sizeof(struct node));
scanf("%d",&temp1->data);
temp->next=temp1;
temp1->next=NULL;
temp=temp1;
}
}return no;
}
int deletion(int b1)
{
temp=head;
printf("\nENTER THE PORTION TO BE DELETED:");
scanf("%d",&m);
b1=b1-1;
for(j=1; j<m; j++)
{
temp=temp->next;
}
temp->next=temp->next->next;
return b1;
}
int insertion(int b2)
{
```

```
int b5=b2+1;
temp=head;
temp1=malloc(sizeof(struct node));
printf("\n ENTER THE PORTION TO BE INSERTED:");
scanf("%d", &m);
if (m<=b5)
{
b2=b2+1;
printf("\n ENTER THE DATA");
scanf("%d", &temp1->data);
for (j=1; j<=m; j++)
{
temp=temp->next;
}
temp1->next=temp->next;
temp->next=temp1;
}return b2;
}
void display(int b3)
{
int i=1;
temp=head;
printf("\n ELEMENTS IN THE SINGLY LINKED LIST ");
while(i<=b3)
{
printf("\t-->%d", temp->next->data);
temp=temp->next;
i=i+1;
}
getch();
}
```

RESULT:

The implementation of insert node, delete node and final node operation on singly liked list was executed and the output was verified

EXPT NO : 02 PAGE NO :09 DATE :	DOUBLY LINKED LIST

AIM:

To write a C program to implement doubly linked list

ALGORITHM:

Node structure:

Left ptr	Element	Right ptr

Element –data
Left ptr -Point to the left pointer
Right ptr –Point to the right pointer

Insert at beginning:

- Get a new node (allocate space), and get the element to be inserted.
- Set the pointer of the new node to header and left pointer to NULL.
- Set the left pointer of the header to new node.
- Set the new node as header.
- Display the content of the list.

Insert at end:

- Get a new node (allocate space), and get the element to be inserted.
- Move the pointer to the last of the list.
- Set the right pointer of the last node to new node and left pointer of the new node to last node.
- Set the right pointer of the new node to NULL.
- Display the content of the list.

Insert any value:

- Get a new node (allocate space), and get the element to be inserted.

- Move the pointer the list to the node that contain the element < than the given element.
- Set the right pointer of the current node to new node and set the left pointer of the new node to current node.
- Set the right pointer of the new node to current+1 node and set the left pointer of the current+1 node to new node.
- Display the content of the list.

Delete from beginning:

- Set pointer of the head node as header.
- Set the left pointer of left node as NULL.
- Display the content of the list.

Delete from end:

- Move the pointer of the list up to just before the last node.
- Set the right pointer of the (n-1)th node as NULL.
- Display the content of the list.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
int create();
int insertion(int);
int deletion(int);
void display();
struct node
{
int data;
struct node *next,*prev;
}*head,*temp,*temp1;
void main()
{
int n=0,choice;
clrscr();
printf("\n Doubly linked list");
while(1)
{
```

```
printf("\n1.create\n2.Insertion\n3.Deletion\n4.display\n5.Exit");
printf("\n Enter the choice;");
scanf("%d",& choice);
switch(choice)
{
case 1:
     n=create();
     break;
case 2:
     n=insertion(n);
     break;
case 3:
     n=Deletion(n);
     break;
case 4:
     display(n);
     break;
case 5:
     exit(0);
     }getch();
}
}
int create()
{
int no;
head=malloc(sizeof(struct node));
head->data=0;
head->next=NULL;
printf("Enter the number of nodes;");
scanf("%d",&no);
if(head->next==NULL)
{
int i;
temp=malloc(sizeof(struct node));
printf("\n Enter the elements:");
scanf("%d",&temp->data);
head->next=temp;
temp->next=NULL;
temp->prev=NULL;
for (i=1;i<no;i++)
{
temp1=malloc(sizeof(struct node));
scanf("%d",&temp1->data);
temp->next=temp1;
temp1->next=NULL;
temp1->prev=temp;
```

```
temp=temp1;
}
}return no;
}
int insertion(int pos)
{
int p,a=pos+1,j;
temp1=malloc(sizeof(struct node));
temp=head;
printf("\n Enter the position:");
scanf("%d",&p);
if(p<=a)
{
pos=pos+1;
printf("Enter the data:");
scanf("%d",&temp1->data);
for(j=1;j<=p-1;j++)
{
temp=temp->next;
temp1->next=temp->next;
temp1->prev=temp;
temp->next=temp1;
}
}return pos;
}
int Deletion(int pos)
{int m,j;
temp=head;
printf("Enter the position to delete:");
scanf("%d",&m);
pos=pos-1;
for(j=1;j<m;j++)
temp=temp->next;
temp->next=temp->next->next;
temp->next->prev=temp;
return pos;
}

void display(int b6)
{
int i=1;
temp=head;
printf("\n Elements in Doubly Linked List");
while(i<=b6)
{
printf("\t<->%d",temp->next->data);
```

```
temp=temp->next;
i=i+1;
}
getch();
}
```

RESULT:

The implementation of insert node, delete node and final node operation on doubly linked list was executed and the output was verified.

EXPT NO : 03 PAGE NO :13 DATE :	**IMPLEMENTATION OF STACK**

AIM:

To implement a polynomial as a linked list and write a function for polynomial addition.

ALGORITHM:

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");
    }

}
```

RESULT:

The implementation of stack operation was executed and output was verified.

EXPT NO : 04 PAGE NO :17 DATE :	**POLYNOMIAL ADDITION**

AIM:

To implement a polynomial as a linked list and write a function for polynomial addition.

ALGORITHM:

- Define all the variables and functions
- Define the structure and declare the variables.
- Get the first polynomial expression
- Get the second polynomial expression
- Call the polyadd() function
- Perform the operation
- Show the result.

PROGRAM:

```
#include<stdio.h>
#include<malloc.h>
#include<conio.h>
struct link{
      int coeff;
      int pow;
      struct link *next;
      };
struct link *poly1=NULL,*poly2=NULL,*poly=NULL;
void create(struct link *node)
{
 char ch;
 do
 {
  printf("\n enter coeff:");
  scanf("%d",&node->coeff);
  printf("\n enter power:");
  scanf("%d",&node->pow);
  node->next=(struct link*)malloc(sizeof(struct link));
  node=node->next;
  node->next=NULL;
```

```
  printf("\n continue(y/n):");
  ch=getch();
 }
 while(ch=='y' || ch=='Y');
}
void show(struct link *node)
{
 while(node->next!=NULL)
 {
  printf("%dx^%d",node->coeff,node->pow);
  node=node->next;
  if(node->next!=NULL)
   printf("+");
 }
}
void polyadd(struct link *poly1,struct link *poly2,struct link *poly)
{
    while(poly1->next &&  poly2->next)
    {
     if(poly1->pow>poly2->pow)
     {
      poly->pow=poly1->pow;
      poly->coeff=poly1->coeff;
      poly1=poly1->next;
      }
     else if(poly1->pow<poly2->pow)
     {
      poly->pow=poly2->pow;
      poly->coeff=poly2->coeff;
      poly2=poly2->next;
      }
     else
     {
      poly->pow=poly1->pow;
      poly->coeff=poly1->coeff+poly2->coeff;
      poly1=poly1->next;
      poly2=poly2->next;
      }
     poly->next=(struct link *)malloc(sizeof(struct link));
     poly=poly->next;
     poly->next=NULL;
    }
    while(poly1->next || poly2->next)
    {
     if(poly1->next)
     {
      poly->pow=poly1->pow;
      poly->coeff=poly1->coeff;
      poly1=poly1->next;
      }
```

```
    if(poly2->next)
    {
     poly->pow=poly2->pow;
     poly->coeff=poly2->coeff;
     poly2=poly2->next;
     }
     poly->next=(struct link *)malloc(sizeof(struct link));
     poly=poly->next;
     poly->next=NULL;
     }
}
main()
{
    char ch;
    do{
    poly1=(struct link *)malloc(sizeof(struct link));
    poly2=(struct link *)malloc(sizeof(struct link));
    poly=(struct link *)malloc(sizeof(struct link));
    printf("\nenter 1st number:");
    create(poly1);
    printf("\nenter 2nd number:");
    create(poly2);
    printf("\n1st Number:");
    show(poly1);
    printf("\n2nd Number:");
    show(poly2);
    polyadd(poly1,poly2,poly);
    printf("\nAdded polynomial:");
    show(poly);
    printf("\n add two more numbers:");
    ch=getch();
    }
    while(ch=='y' || ch=='Y');
}
```

RESULT:

The implementation of a polynomial as a linked list and function for polynomial addition was executed and output was verified.

EXPT NO : 05 PAGE NO :20 DATE :	**INFIX TO POSTFIX CONVERSION**

AIM:

To write a C program to implement infix to postfix conversion.

ALGORITHM:

- It is a ordered collection of homogenous data elements.
- It is also called as LIFO (last in first out)

The insertion and deletion are performed at one end are called top of the stack.

- Operands immediately go directly to output
- Operators are pushed in to tag (paranthiesis)
- Check to see if stack operators is less than current operator.
- If the top operator is less than push current operator to the stack.
- If the top operator is greater than the current, pop top operator and push on to the stack.
- If we encounter the right paranthesis pop from the stack and until we match the left paranthesis (output paranthesis).

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#define operand(x) (x>='a'&&x<='z'||x>='A'&&x<='Z'||x>='0'&&x<='9')
char infix[30],postfix[30],stack[30];
int top,i=0;
void init()
{
top=-1;
}
void push(char x)
{
stack[++top]=x;
}
char pop()
{
return(stack[top--]);
}
int isp(char x)
{
int y;
y=(x=='('?0:x=='^'?4:x=='*'?2:x=='/'?2:x=='+'?1:x=='-'?1:x==')'?6:-1);
return y;
```

```
}
int icp(char x)
{
int y;
y=(x=='('?0:x=='^'?4:x=='*'?2:x=='/'?2:x=='+'?1:x=='-'?1:x==')'?6:-1);
return y;
}
void infixtopostfix()
{
int j,l=0;
char x,y;
stack[++top]='\0';
for(j=0;(x=infix[i++])!='\0';j--)
if(operand(x))
postfix[l++]=x;
else
if(x==')')
while((y=pop())!='(')
postfix[l++]=y;
else
{
while(isp(stack[top])>=icp(x))
postfix[l++]=pop();
push(x);
}
while(top>=0)
postfix[l++]=pop();
}
int main()
{
clrscr();
init();
printf("\nEnter the infix Expression:");
scanf("%s",infix);
infixtopostfix();
printf("\n The resulting post fix Expression:%s",postfix);
getch();
return 0;
}
```

RESULT:

The implementation of infix to postfix operations was executed and the output was verified.

EXPT NO : 06 PAGE NO :22 DATE :	**DEQUE**

AIM:

To write a c program to implement Deque.

ALGORITHM:

It is a double ended queue.

- Insertion and deletion operations can be performed at both ends.

Operations on a deque

- PUSH DQ (item) - insert item at a front end.
- POP DQ () – remove item at front end.
- INJECT (item)-insert an item at rear end.
- EJECT () - remove the item from rear end.

Insert at front end (PUSH):

- Get a new data (allocate space), and get the element to be inserted.
- Set the pointer of the new data.
- Set the current data.
- Display the content of the list.

Insert at rear end (INJECT):

- Get a new data (allocate space), get the element to be inserted.
- Move the pointer to the last data of the list.
- Set the pointer of the last data to the new node.
- Display the content of the list.

Insert any value:

- Get a new data (allocate space), and get the element to be inserted.
- Move the pointer of the list that contain element.
- Set the pointer of the current data.
- Display the content of the list.

Remove from front end(POP):

- Set the pointer of the data.
- Display the content of the list.

Remove from rear end (EJECT):

- Move the pointer of the list up to just before the last data.
- Display the content of the list.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
typedef struct dq_t
{
int front,rear;
int item[SIZE];
}deque;
void create(deque*);
void display(deque*);
void insert_rear(deque*, int);
void insert_front(deque*,int);
int delete_front(deque*, int);
int delete_rear(deque*, int);
void main()
{
        int x,data,ch;
        deque DQ;
        clrscr();
        create(&DQ);
        printf("\n\t\t program shows working of double ended queue");
        do
{
        printf("\n\t\t menu");
        printf("\n\t\t 1: insert at rear end");
        printf("\n\t\t 2: insert at front end");
        printf("\n\t\t 3: delete from front end");
        printf("\n\t\t 4: delete from rear end");
        printf("\n\t\t 5: exit");
        printf("\n\t\t enter the choice:");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:
```

```
        if(DQ.rear >= SIZE)
        {
        printf("\n Deque is full at rear end");
        continue;
}
else
{
        printf("\n enter the element to be added at rear end");
        scanf("%d",&data);
        insert_rear(&DQ,data);
        printf("\n elements in a deque are:");
        display(&DQ);
        continue;
}
        case 2:
        if(DQ.front <=0)
        {
        printf("\n deque is full at front end");
        continue;
        }
else
        {
        printf("\n enter element to be added at front end:");
        scanf("%d",&data);
        insert_front(&DQ,data);
        printf("\n elements in a deque are:");
        display(&DQ);
        continue;
}
        case 3:
        x = delete_front(&DQ,data);
        if (DQ.front==0)
        {
        continue;
        }
else
        {
        printf("\n elements in a deque are");
        display(&DQ);
        continue;
        }
        case 4:
        x = delete_rear(&DQ,data);
        if (DQ.rear==0)
        {
        continue;
        }
else
        {
        printf("\n elements in a deque are");
```

```
                display(&DQ);
                continue;
                }
                case 5:
                printf("\n finish"); return;
                }
                }while(ch!=5);
                getch();
                }
        void create(deque *DQ)
        {
        DQ->front=0;
        DQ->rear=0;
        }
        void insert_rear(deque *DQ, int data)
        {
        if ((DQ->front == 0) &&(DQ->rear == 0))
        {
        DQ->item[DQ->rear] = data;
        DQ->rear = DQ->rear+1;
        }
        else
        {
        DQ->item[DQ->rear] = data;
        DQ->rear = DQ->rear +1;
        }
        }
        int delete_front(deque *DQ,int data)
        {
        if ((DQ->front == 0) && (DQ->rear == 0))
        {
        printf("\n underflow");
        return(0);
        }
else
        {
        data = DQ->item[DQ->front];
        printf("\n element %d is deleted from front:", data);
        DQ->front = DQ->front+1;
        }
        if (DQ->front==DQ->rear)
        {
        DQ->front=0;
        DQ->rear=0;
        printf("\n deque is empty(front end)");
        }
        return data;
        }
        void insert_front(deque *DQ, int data)
        {
```

```
        if (DQ->front > 0)
        {
        DQ->front = DQ->front-1;
        DQ->item[DQ->front] = data;
        }
        }
        int delete_rear(deque *DQ, int data)
        {
        if (DQ->rear == 0)
        {
        printf("\n underflow");
        return(0);
        }
else
        {
        DQ->rear = DQ->rear-1;
        data = DQ->item[DQ->rear];
        printf("\n elements %d is deleted from rear:", data);
        }
        if(DQ->front==DQ->rear)
        {
        DQ->front=0;
        DQ->rear=0;
        printf("\n deque is empty(rear end)");
        }
        return data;
        }
        void display(deque *DQ)
{
        int x;
        for(x=DQ->front;x<DQ->rear;x++)
{
        printf("%d\t",DQ->item[x]);
}
printf("\n\n");
}
```

RESULT:

The implementation of insert node, delete node operation on deque was executed and the output was obtained.

EXPT NO : 07 PAGE NO :27 DATE :	BINARY TREE TRAVERSAL

AIM:

To write a C program to implement the binary tree traversal.

ALGORITHM:

Structure

LEFT CHILD	DATA	RIGHT CHILD

OPERATIONS

- Pre order traversal.
- In order traversal.
- Post order traversal.

PRE ORDER TRAVERSAL

- Visit the root(V)
- Transverse the left sub-tree(L)
- Transverse the right sub-tree(R)

IN ORDER TRAVERSAL

- Transverse the left sub-tree(L)
- Visit the root(V)
- Transverse the right sub-tree(R)

POST ORDER TRAVERSAL

- Transverse the left sub-tree(L)
- Transverse the right sub-tree(R)
- Visit the root

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>
#include<stdlib.h>
int found=0;
struct tree_element
{
 int item;
 struct tree_element *left;
 struct tree_element *right;
};
typedef struct tree_element node;

void inorder(node *record)
{
 if(record!=NULL)
 {
  inorder(record->left);
  printf("\t%d",record->item);
  inorder(record->right);
 }
 return;
}

void preorder(node *record)
{
 if(record!=NULL)
 {
  printf("\t%d",record->item);
  preorder(record->left);
  preorder(record->right);
 }
 return;
}

void postorder(node *record)
{
 if(record!=NULL)
 {
  postorder(record->left);
  postorder(record->right);
```

```
  printf("\t%d",record->item);
 }
 return;
}
void create(node *record,int n)
{
 if(n>record->item)
 {
  if(record->right==NULL)
  {
   record->right=(node *) malloc(sizeof(node));
   record=record->right;
   record->item=n;
   record->right=NULL;
   record->left=NULL;
  }
  else
  {
   create(record->right,n);
  }
 }
 else
 if(n<record->item)
 {
  if(record->left==NULL)
  {
   record->left=(node *) malloc(sizeof(node));
   record=record->left;
   record->item=n;
   record->left=NULL;
   record->right=NULL;
  }
  else
  {
   create(record->left,n);
  }
 }
 else
 {
  printf(" \n the number is a duplicate ");
 }
  return;
}
```

```
 void main()
{
 int num,ch,n,f;
 node *tree;
 void create();
 void inorder();
 void preorder();
 void postorder();
clrscr();
 tree=(node *) malloc(sizeof(node));
 printf("type numbers one per line \n ");
 printf(" \ncntrl_z for EOF");
 scanf("%d",&num);
 tree->item=num;
 tree->right=NULL;
 tree->left=NULL;
 while((scanf("%d",&num))!=EOF)
 {
  create(tree,num);
 }
 do
 {
  printf("\nTYPE OF TRAVERSAL ");
  printf("\n1.INORDER");
  printf("\n2.PREORDER");
  printf("\n3.POSTORDER");
  printf("\n4.STOP");
  printf("\nENTER UR CHOICE \n ");
  scanf("%d",&ch);
  switch(ch)
  {
   case 1: printf("\nInorder traversal\n");
           inorder(tree);
           break;
   case 2: printf("\nPreorder traversal\n");
           preorder(tree);
           break;
   case 3: printf("\nPostorder traversal\n");
           postorder(tree);
           break;
   default:
           exit(0);
  }
```

```
  printf("\n");
 }while(ch!=5);
 getch();
}
```

RESULT:

The implementation of tree traversal operation was executed and the output was verified.

EXPT NO : 08 PAGE NO :32 DATE :	BINARY SEARCH TREE

AIM:

To write a C program to implement binary search tree.

ALGORITHM:

Insert node:

- Get the node with element.
- If the tree is empty then make the node as header.
- Otherwise, search the appropriate position of the element.
- If the new value is < tree then move on left sub tree.
- Else move on right sub tree.
- Insert the new node as leaf node.
- Display the tree [in order].

Delete queue:

- Get the element to be deleted.
- Search the element from header.
- If element< tree element move on left sub tree.
- Else move on right sub tree.

If find the element remove the node and convert the tree, with follow the binary tree properties.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
struct node
{
int data;
struct node *left,*right;
};
```

```
typedef struct node *nodeptr;
nodeptr root=NULL,t,p,q;
void add();
void search();
void disp(nodeptr,int,int,int);
void main()
{
int ch,num;
while(1)
{
clrscr();
printf("\n\t\t\t\tBINARY SEACH TREE");
printf("\n\t\t\t**********************************\n\n");
printf("\n1.Add a Node\n2.Search a Node\n3.Display\n4.Exit");
printf("\n Enter the Choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
add();
break;
case 2:
search();
getch();
break;
case 3:
disp(root,40,15,10);
getch();
break;
case 4:
exit(0);
break;
}
getch();
}
}
void add()
{
int x;
t=malloc(sizeof(struct node));
printf("\nEnter the Data:");
scanf("%d",&x);
t->data=x;
t->left=NULL;
t->right=NULL;
if(root==NULL)
```

```
root=t;
else
{
p=q=root;
while(q!=NULL&&x!=p->data)
{
p=q;
if(x<p->data)
q=p->left;
else
q=p->right;
}
if(x==p->data)
printf("\n%d is Duplicate Number",x);
else if(x<p->data)
p->left=t;
else

p->right=t;
}}
void search()
{
int x;
printf("\n Enter the Element to search:");
scanf("%d",&x);
p=q=root;
while(q!=NULL&&x!=p->data)
{
p=q;
if(x<p->data)
q=p->left;
else
q=p->right;
}
if(x==p->data)
{
printf("Element is present");
 }
 else
printf("Element is not Present");
}
void disp(nodeptr root,int col,int row,int wid)
{
gotoxy(col,row);
if(root!=NULL)
{
```

```
printf("%d",root->data);
disp(root->left,col-wid,row+2,wid/2);
disp(root->right,col+wid,row+2,wid/2);
}
}
```

RESULT:

The implementation of insert node, delete node and transverse operation on binary search tree was executed and the output was verified.

EXPT NO : 09(A) PAGE NO :36 DATE :	**LINEAR SEARCH**

AIM:

To write a C program to implement linear search.

ALGORITHM:

1. Step 1: set pos = -1
2. Step 2: set i = 1
3. Step 3: repeat step 4 while i <= n
4. Step 4: if a[i] == val
5. set pos = i
6. print pos
7. go to step 6
8. [end of if]
9. set ii = i + 1
10. [end of loop]
11. Step 5: if pos = -1
12. print "value is not present in the array "
13. [end of if]
14. Step 6: exit

PROGRAM:

```
#include <stdio.h>

void main()
{  int num;

   int i,  keynum, found = 0;

   printf("Enter the number of elements ");
   scanf("%d", &num);
   int array[num];
   printf("Enter the elements one by one \n");
   for (i = 0; i < num; i++)
   {
      scanf("%d", &array[i]);
   }

   printf("Enter the element to be searched ");
   scanf("%d", &keynum);
```

```
   /*  Linear search begins */
   for (i = 0; i < num ; i++)
   {
      if (keynum == array[i] )
      {
         found = 1;
         break;
      }
   }
   if (found == 1)
      printf("Element is present in the array at position %d",i+1);
   else
      printf("Element is not present in the array\n");
}
```

RESULT:

The implementation of linear search operation was executed and the output was verified.

EXPT NO : 09(B) PAGE NO :38 DATE :	**BINARY SEARCH**

AIM:

To write a C program to implement binary search.

ALGORITHM:

1. Step 1: set pos = -1
2. Step 2: set i = 1
3. Step 3: repeat step 4 while i <= n
4. Step 4: if a[i] == val
5. set pos = i
6. print pos
7. go to step 6
8. [end of if]
9. set ii = i + 1
10. [end of loop]
11. Step 5: if pos = -1
12. print "value is not present in the array "
13. [end of if]
14. Step 6: exit

PROGRAM:

```
#include <stdio.h>

int main()

{

int i, low, high, mid, n, key, array[100];

printf("Enter number of elementsn");

scanf("%d",&n);

printf("Enter %d integersn", n);

for(i = 0; i < n; i++)

scanf("%d",&array[i]);

printf("Enter value to findn");

scanf("%d", &key);

low = 0;
```

```
high = n - 1;
mid = (low+high)/2;
while (low <= high) {
if(array[mid] < key)
low = mid + 1;
else if (array[mid] == key) {
printf("%d found at location %d.n", key, mid+1);
break;
}
else
high = mid - 1;
mid = (low + high)/2;
}
if(low > high)
printf("Not found! %d isn't present in the list.n", key);
return 0;
}
```

RESULT:

The implementation of linear search operation was executed and the output was verified.

EXPT NO : 10 PAGE NO :40 DATE :	PRISM'S ALGORITHM

AIM:

To write a C program to find the shortest path in an undirected graph using prim's algorithm.

ALGORITHM:

- Get no. of vertices.
- Enter weighted matrix.
- Start from first node, find the minimum cost node if path is available from already visited node and make the path with the node[cycle not allowed].
- Print the closest path.
- Repeat until visit all nodes.
- Display total weight of the maximum cost spanning tree.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
int cost[50][50],source[10][3];
int nv,ne,u,v,position,initial,mincost,disnode;
void getdata();
void process();
int minsourcevertex(int);
int check(int,int);
void main()
{
clrscr();
printf("\t\t\tPRIM'S ALGORITHM\n");
printf("**************************");
getdata();
process();
getch();
}
void getdata()
{
```

```
int i,j;
printf("\n Enter the number of vertices==>");
scanf("%d",&nv);
printf("\n Enter the number of edges==>");
scanf("%d",&ne);
for(i=1;i<=nv;i++)
{
for(j=1;j<=ne;j++)
{
cost[i][j]=999;
}
}
for(i=1;i<=ne;i++)
{
printf("\nEnter the edges(u,v)==>");
scanf("%d%d",&u,&v);
printf("\n Enter the cost value==>");
scanf("%d",&cost[u][v]);
cost[v][u]=cost[u][v];
}
printf("\nEnter the source vertex==>");
scanf("%d",&initial);
}
void process()
{
int i,vertex;
source[1][2]=initial;
for(i=2;i<=nv;i++)
{
vertex=minsourcevertex(i-1);
source[i][1]=disnode;
source[i][2]=vertex;
source[i][3]=mincost;
}
printf("\n Minimum spanning tree\n");
printf("\n source\t distination\t cost\n ");
for(i=2;i<=nv;i++)
{
printf("\t%d\t %d \t %d\n",source[i][1],source[i][2],source[i][3]);
}
}
int minsourcevertex(int arrsize)
{
int i,j,temp;
mincost=cost[1][1];
for(i=1;i<=arrsize;i++)
```

```
{
position=source[i][2];
for(j=1;j<=nv;j++)
{
if(cost[position][j]<mincost)
{
if(!check(j,arrsize))
{
temp=j;
disnode=position ;
mincost=cost[position][temp];
}
}}}
return temp;
}
int check(int value,int size)
{
int i,retvalue=0;
for(i=1;i<=size;i++)
{
if(value==source[i][2])
{
retvalue=1;
break;
}
}
return retvalue;
}
```

RESULT:

The implementation of shortest path in an undirected graph using prim's algorithm was executed and the output was verified.

EXPT NO : 11 PAGE NO :43 DATE :	**BINARY HEAPS**

AIM:

To write a C program to implement the priority queue using Binary Heaps

ALGORITHM:

- Start the program
- Insertion,deletion,display operations are done.
- Declare the queue structure
- Input the queue size.
- Then enter the choice to either insert,delete,display and exit
- Enter the particular element to which operation is to be done
- Perform insertion operation and then deletion operation.
- If the elements are greater than queue size then queue overflow will occur.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
struct queue
{
  int element;
  int pri;
}dpq[30];

void pqinsert(struct queue[],int,int,int),pqdelete(struct queue[],int),adjustheap(int,int);
int smallson(int,int);

void main()
{
  int size,c=1,k=0,elt,p,i;
  clrscr();
  printf("Enter queue size:");
  scanf("%d",&size);
  while(c<4)
```

```
 {
        printf("1.insert\n2.delete\n3.display\n4.exit\nenter choice");
        scanf("%d",&c);
        switch(c)
        {
        case 1:
         if(k<size)
        {
           printf("Enter element");
          scanf("%d",&elt);
          printf("enter priority");
          scanf("%d",&p);
          pqinsert(dpq,k,elt,p);
          k++;
        }
        else
        printf("Queue Overflow");
        break;
        case 2:
         if(k!=0)
        {
           pqdelete(dpq,k);
          k--;
        }
        else
           printf("Underflow");
        break;
        case 3:
         if(k!=0)
          for(i=0;i<k;i++)
             printf("%d\t%d\n",dpq[i].element,dpq[i].pri);
        else
           printf("Queue is empty\n");
          break;
        case 4:
           break;
        }
    }
}

void pqinsert(struct queue dpq[],int k,int elt,int p)
{
         int s,f;
        s=k;
        f=(s-1)/2;
        while(s>0&&dpq[f].pri>p)
```

```
        {       dpq[s]=dpq[f];
                s=f;
                f=(s-1)/2;
        }
        dpq[s].element=elt;
        dpq[s].pri=p;
}

void pqdelete(struct queue dpq[],int k)
{
        printf("Deleted element is:%d\n",dpq[0].element);
        printf("Deleted priority is:%d\n",dpq[0].pri);
        adjustheap(0,k-1);
}
void adjustheap(int root,int k)
{
        int f,s;
        f=root;
        s=smallson(f,k-1);
        if(s>=0&&dpq[k].pri>dpq[s].pri)
        {       dpq[f]=dpq[s];
                adjustheap(s,k);
        }
        else
                dpq[f]=dpq[k];
}
int smallson(int p,int m)
{       int s;
        s=2*p+1;
        if(s+1<=m&&dpq[s].pri>dpq[s+1].pri)
                s=s+1;
        if(s>m)         return(-1);
        else return(s);
}
```

RESULT

The implementation of priority queue by using binary heaps was executed successfully and the output was verified.

EXPT NO : 12 PAGE NO :46 DATE :	HASHING WITH OPEN ADDRESSING

AIM:

To write a C program to implement the hashing with open addressing.

ALGORITHM:

Opening Hashing:

- Create a new node as structure.
- For insert option to enter the element and insert into the hash. Its hash value is calculated and is stored in the list corresponding to the hash value.
- For delete option, enter which element to be deleted. Its hash value is calculated and I is removed from the list corresponding to the value.
- To insert the next element, repeat step2 and to delete next step of 3.

Delete queue:

- To check whether the array is not full and enter element.
- If the array position corresponding to the hash value of that element is empty, then the element is inserted at the nearest empty space incrementally.
- If the array position corresponding to the hash value of that element is not empty, at the nearest empty space incrementally.
- To delete an element, enter the element to delete. If the element is not found then the element is deleted from the array.

PROGRAM:

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void linear_prob(int[],int,int);
void display(int[]);
```

```
int create(int);
int a[100],num,key,I;
int Tablesize=100,ch;
char ans;
void main()
{ clrscr();
printf(”\n Hashing – Collision handling by Linear probing\n”);
printf(”\n Enter the Hash Table Size:”);
scanf(”%d”,&Tablesize);
for(i=0;i<Tablesize;i++)
a[i]=-1;
printf("\n Menu\n");
printf("\n1.Insertion\n2.Display\n3.Exit\n");
do
{
printf("\nEnter your choice:");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\nEnter the number:");
scanf("%d",&num);
key=create(num);
linear_prob(a,key,num);
break;
case 2: display(a); break;
default: printf("\nInvalid Choice");
} }while(ch!=3);
}
int create(int num)
{ int key;
key=num%10;
return key;
}
void linear_prob(int a[100],int key,int num)
{ int flag,i,count=0;
flag=0;
if(a[key]==-1)
a[key]=num;
else
{ i=0;
while(i<Tablesize)
{ if(a[i]!=-1)
count++;
i++;
}
```

```
if(count==Tablesize)
{ printf("\n Hash Table Is Full");
display(a);
getch();
}
for(i=key+1;i<Tablesize;i++)
if(a[i]==-1)
{ a[i]=num;
flag=1;
break;
}
for(i=0;i<key&&flag==0;i++)
if(a[i]==-1)
{ a[i]=num;
flag=1;
break;
}} }
void display(int a[100])
{
int i;
printf("\n The Hash Table is….\n");
for(i=0;i<Tablesize;i++)
printf("\n%d %d",i,a[i]);
}
```

RESULT:

The implementation of hashing with open addressing was executed and the output was verified.

EXPT NO : 13(A) PAGE NO :49 DATE :	**HEAP SORT**

AIM:

To write a C program to implement the heap sort.

ALGORITHM:

- Insert the element in top if array is empty.
- Check up to the root, whether the top value is greater than its parents, if not swap the content.
- Consume the element with highest priority, i.e. put the top value in the first position and decrement the top value.
- Check up to top/2, whether the root value is smaller than the children, or it is not so swap the contents.
- Display the heap.

PROGRAM:

```
#include<stdio.h>
void heap(int a[],int n);
void create_heap(int a[],int n);
void main()
{
int a[50],i,n;
clrscr();
printf("\nEnter the limit:");
scanf("%d",&n);
printf("\nEnter the Element:\n");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
heap(a,n);
for(i=0;i<n;i++)
printf("%d\t",a[i]);
}
```

```
void create_heap(int a[],int n)
{
int i,j,q,key;
for(q=1;q<n;q++)
{
i=q;
key=a[q];
j=(int)(i/2);
while((i>0)&&(key>a[j]))
{
a[i]=a[j];
i=j;
j=(int)(i/2);
if(j<0)
j=0;
}
a[i]=key;
}
}
void heap(int a[],int n)
{
int I,i,j,q,key,temp;
create_heap(a,n);
for(q=n-1;q>=1;q--)
{
temp=a[0];
a[0]=a[q];
a[q]=temp;
i=0;
key=a[0];
j=1;
if((j+1)<q)
if(a[j+1]<a[j])
j=j+1;
while((j<=(q-1))&&(a[j]>key))
{
a[i]=a[j];
i=j;
j=2*I;
if((j+1)<q)
if(a[j+1]>a[j])
j=j+1;
else if(j>n-1)
j=n-1;
a[i]=key;
}
```

```
}
}
```

RESULT:

The implementation of heap sort operation was executed and the output was verified.

EXPT NO : 13(B) PAGE NO :52 DATE :	RADIX SORT

AIM:

To write a C program to implement the heap sorting.

ALGORITHM:

1. radixSort(arr)
2. max = largest element in the given array
3. d = number of digits in the largest element (or, max)
4. Now, create d buckets of size 0 - 9
5. for i -> 0 to d
6. sort the array elements using counting sort (or any stable sort) according to the digits at
7. the ith place

PROGRAM:

```
#include<stdio.h>
int get_max (int a[], int n){
  int max = a[0];
  for (int i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
void radix_sort (int a[], int n){
  int bucket[10][10], bucket_cnt[10];
  int i, j, k, r, NOP = 0, divisor = 1, lar, pass;
  lar = get_max (a, n);
  while (lar > 0){
    NOP++;
    lar /= 10;
  }
```

```
  for (pass = 0; pass < NOP; pass++){
    for (i = 0; i < 10; i++){
      bucket_cnt[i] = 0;
    }
    for (i = 0; i < n; i++){
      r = (a[i] / divisor) % 10;
      bucket[r][bucket_cnt[r]] = a[i];
      bucket_cnt[r] += 1;
    }
    i = 0;
    for (k = 0; k < 10; k++){
      for (j = 0; j < bucket_cnt[k]; j++){
        a[i] = bucket[k][j];
        i++;
      }
    }
    divisor *= 10;
    printf ("After pass %d : ", pass + 1);
    for (i = 0; i < n; i++)
      printf ("%d ", a[i]);
    printf ("
");
  }
}
int main (){
  int i, n, a[10];
  printf ("Enter the number of items to be sorted: ");
  scanf ("%d", &n);
  printf ("Enter items: ");
```

```
  for (i = 0; i < n; i++){
    scanf ("%d", &a[i]);
  }
  radix_sort (a, n);
  printf ("Sorted items : ");
  for (i = 0; i < n; i++)
    printf ("%d ", a[i]);
  printf ("
");
  return 0;
}
```

RESULT:

The implementation of radix sort operation was executed and the output was verified.

EXPT NO : 13(C) PAGE NO :55 DATE :	BUBBLE SORT

AIM:

To write a C program to implement the heap sorting.

ALGORITHM:

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
 return arr
end BubbleSort
```

PROGRAM:

```
// Bubble sort in C

#include <stdio.h>

// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two adjacent elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
```

```
  }
}

// print array
void printArray(int array[], int size) {
  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}

int main() {
  int data[] = {-2, 45, 0, 11, -9};

  // find the array's length
  int size = sizeof(data) / sizeof(data[0]);

  bubbleSort(data, size);

  printf("Sorted Array in Ascending Order:\n");
  printArray(data, size);
}
```

RESULT:

The implementation of bubble sort operation was executed and the output was verified.

EXPT NO : 13(D) PAGE NO :57 DATE :	MERGE SORT

AIM:

To write a C program to implement the heap sorting.

ALGORITHM:

Start

Declare an array and left, right, mid variable

Perform merge function.

mergesort(array,left,right)

mergesort (array, left, right)

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

Stop

PROGRAM:

```
#include<stdlib.h>
#include<stdio.h>
// Merge Function
void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
int L[n1], R[n2];
```

```
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1+ j];
i = 0;
j = 0;
k = l;
while (i < n1 && j < n2)
{
if (L[i] <= R[j])
{
arr[k] = L[i];
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
while (i < n1)
{
arr[k] = L[i];
i++;
k++;
}
while (j < n2)
{
arr[k] = R[j];
j++;
k++;
}
}
```

RESULT:

The implementation of merge sort operation was executed and the output was verified

EXPT NO : 14 PAGE NO :60 DATE :	CIRCULAR QUEUE ADT

AIM:

To write a C program to implement Circular Queue ADT to perform insertion and deletion operations.

ALGORITHM:

Start
Declare a CircularQueue structure with array, front, rear
Initialize front and rear to -1
Check if the queue is full using isFull()
Check if the queue is empty using isEmpty()
For insertion (enqueue):
If queue is full, display overflow message
If queue is empty, set front = 0
Increment rear circularly using (rear + 1) % SIZE
Insert element at rear
For deletion (dequeue):
If queue is empty, display underflow message
Store element at front
If front == rear, set front = rear = -1
Else increment front circularly using (front + 1) % SIZE
Display elements from front to rear circularly
Stop

PROGRAM:

```
#include <stdio.h>
 #include <stdlib.h>
 #define SIZE 5

typedef struct {
 int items[SIZE];
 int front;
 int rear;
 } CircularQueue;

void initialize(CircularQueue *q) {
 q->front = -1;
 q->rear = -1;
 }
```

```
int isFull(CircularQueue *q) {
 return (q->front == (q->rear + 1) % SIZE);
 }

int isEmpty(CircularQueue *q) {
 return (q->front == -1);
 }

void enqueue(CircularQueue *q, int value) {
 if (isFull(q)) {
 printf("Queue is full. Cannot insert %d\n", value);
 return;
 }
 if (isEmpty(q)) q->front = 0;
 q->rear = (q->rear + 1) % SIZE;
 q->items[q->rear] = value;
 printf("Inserted %d\n", value);
 }

int dequeue(CircularQueue *q) {
 if (isEmpty(q)) {
 printf("Queue is empty. Cannot delete\n");
 return -1;
 }
 int value = q->items[q->front];
 if (q->front == q->rear)
 q->front = q->rear = -1;
 else
 q->front = (q->front + 1) % SIZE;
 printf("Deleted %d\n", value);
 return value;
 }

void display(CircularQueue *q) {
 if (isEmpty(q)) {
 printf("Queue is empty\n");
 return;
 }
 printf("Queue elements: ");
 int i = q->front;
 while (1) {
 printf("%d ", q->items[i]);
 if (i == q->rear) break;
 i = (i + 1) % SIZE;
 }
 printf("\n");
 }
```

```
int main() {
 CircularQueue q;
 initialize(&q);
 int choice, value;
 while (1) {
 printf("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit\n");
 printf("Enter your choice: ");
 scanf("%d", &choice);
 switch (choice) {
 case 1:
 printf("Enter value to insert: ");
 scanf("%d", &value);
 enqueue(&q, value);
 break;
 case 2:
 dequeue(&q);
 break;
 case 3:
 display(&q);
 break;
 case 4:
 exit(0);
 default:
 printf("Invalid choice. Try again.\n");
 }
 }
 return 0;
 }
```

RESULT:

The implementation of circular queue operations (insertion and deletion) was executed and the output was verified.

EXPT NO : 15 PAGE NO :63 DATE :	AVL TREES AND OPERATIONS

AIM:

To write a C program to implement AVL Tree and perform insertion, deletion, and traversal operations.

ALGORITHM:

Start
Declare a node structure with data, left, right, height
Initialize root to NULL
Create a function to get height of a node
Create a function to calculate maximum of two integers
Create functions for rightRotate and leftRotate for balancing
Create a function to get balance factor of a node
For insertion:
Perform normal BST insertion
Update height of ancestor nodes
Check balance factor
Perform rotations if unbalanced (LL, RR, LR, RL)
For deletion:
Perform normal BST deletion
Update height of ancestor nodes
Check balance factor
Perform rotations if unbalanced (LL, RR, LR, RL)
Create functions for preorder, inorder, and postorder traversal
Stop

PROGRAM:

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

typedef struct Node {
 int data;
 struct Node *left;
 struct Node *right;
 int height;
 } Node;
```

```
int max(int a, int b) {
 return (a > b) ? a : b;
 }

int height(Node *n) {
 if (n == NULL) return 0;
 return n->height;
 }

Node* createNode(int data) {
 Node* node = (Node*)malloc(sizeof(Node));
 node->data = data;
 node->left = node->right = NULL;
 node->height = 1;
 return node;
 }

Node* rightRotate(Node *y) {
 Node *x = y->left;
 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;
 }

Node* leftRotate(Node *x) {
 Node *y = x->right;
 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(Node *n) {
 if (n == NULL) return 0;
 return height(n->left) - height(n->right);
 }

Node* insert(Node* node, int data) {
 if (node == NULL) return createNode(data);
 if (data < node->data)
 node->left = insert(node->left, data);
 else if (data > node->data)
```

```
node->right = insert(node->right, data);
else
return node;
node->height = 1 + max(height(node->left), height(node->right));
int balance = getBalance(node);
if (balance > 1 && data < node->left->data)
return rightRotate(node);
if (balance < -1 && data > node->right->data)
return leftRotate(node);
if (balance > 1 && data > node->left->data) {
node->left = leftRotate(node->left);
return rightRotate(node);
}
if (balance < -1 && data < node->right->data) {
node->right = rightRotate(node->right);
return leftRotate(node);
}
return node;
}

Node* minValueNode(Node* node) {
Node* current = node;
while (current->left != NULL)
current = current->left;
return current;
}

Node* deleteNode(Node* root, int data) {
if (root == NULL) return root;
if (data < root->data)
root->left = deleteNode(root->left, data);
else if (data > root->data)
root->right = deleteNode(root->right, data);
else {
if ((root->left == NULL) || (root->right == NULL)) {
Node* temp = root->left ? root->left : root->right;
if (temp == NULL) {
temp = root;
root = NULL;
} else *root = temp;
free(temp);
} else {
Node temp = minValueNode(root->right);
root->data = temp->data;
root->right = deleteNode(root->right, temp->data);
}
}
```

```
 if (root == NULL) return root;
 root->height = 1 + max(height(root->left), height(root->right));
 int balance = getBalance(root);
 if (balance > 1 && getBalance(root->left) >= 0)
 return rightRotate(root);
 if (balance > 1 && getBalance(root->left) < 0) {
 root->left = leftRotate(root->left);
 return rightRotate(root);
 }
 if (balance < -1 && getBalance(root->right) <= 0)
 return leftRotate(root);
 if (balance < -1 && getBalance(root->right) > 0) {
 root->right = rightRotate(root->right);
 return leftRotate(root);
 }
 return root;
 }

void inorder(Node* root) {
 if (root != NULL) {
 inorder(root->left);
 printf("%d ", root->data);
 inorder(root->right);
 }
 }

void preorder(Node* root) {
 if (root != NULL) {
 printf("%d ", root->data);
 preorder(root->left);
 preorder(root->right);
 }
 }

void postorder(Node* root) {
 if (root != NULL) {
 postorder(root->left);
 postorder(root->right);
 printf("%d ", root->data);
 }
 }

int main() {
 Node* root = NULL;
 int choice, val;
 while (1) {
 printf("\n1. Insert\n2. Delete\n3. Inorder\n4. Preorder\n5. Postorder\n6. Exit\n");
```

```
printf("Enter your choice: ");
scanf("%d", &choice);
switch(choice) {
case 1:
printf("Enter value to insert: ");
scanf("%d", &val);
root = insert(root, val);
break;
case 2:
printf("Enter value to delete: ");
scanf("%d", &val);
root = deleteNode(root, val);
break;
case 3:
printf("Inorder: ");
inorder(root);
printf("\n");
break;
case 4:
printf("Preorder: ");
preorder(root);
printf("\n");
break;
case 5:
printf("Postorder: ");
postorder(root);
printf("\n");
break;
case 6:
exit(0);
default:
printf("Invalid choice\n");
}
}
return 0;
}
```

RESULT:

The implementation of AVL Tree operations (insertion, deletion, and traversal) was executed and the output was verified.

EXPT NO : 16 PAGE NO :68 DATE :	POSTFIX EVALUATION

AIM:

To write a C program to evaluate a postfix expression using stack.

ALGORITHM:

Start
Declare a stack and top variable
Read the postfix expression
For each character in the expression:
If the character is an operand, push it onto the stack
If the character is an operator, pop two elements from the stack
Apply the operator on the popped elements
Push the result back onto the stack
After reading the expression, the stack will have the final result
Pop and display the result
Stop

PROGRAM:

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

#define SIZE 100

int stack[SIZE];
 int top = -1;

void push(int val) {
 if (top >= SIZE - 1) {
 printf("Stack overflow\n");
 return;
 }
 stack[++top] = val;
 }

int pop() {
 if (top == -1) {
 printf("Stack underflow\n");
```

```
 return -1;
 }
 return stack[top--];
 }

int evaluatePostfix(char exp[]) {
 int i = 0;
 char ch;
 int op1, op2, res;
 while ((ch = exp[i++]) != '\0') {
 if (isdigit(ch)) {
 push(ch - '0');
 } else {
 op2 = pop();
 op1 = pop();
 switch(ch) {
 case '+': res = op1 + op2; break;
 case '-': res = op1 - op2; break;
 case '*': res = op1 * op2; break;
 case '/': res = op1 / op2; break;
 case '^': res = pow(op1, op2); break;
 }
 push(res);
 }
 }
 return pop();
 }

int main() {
 char exp[SIZE];
 int result;
 printf("Enter postfix expression: ");
 scanf("%s", exp);
 result = evaluatePostfix(exp);
 printf("Result: %d\n", result);
 return 0;
 }
```

RESULT:

The implementation of postfix expression evaluation was executed and the output was verified.

EXPT NO : 17 PAGE NO :70 DATE :	DIJKSTRA'S ALGORITHM

AIM:

To write a C program to implement Dijkstra's algorithm for finding the shortest path from a source vertex to all other vertices in a weighted graph.

ALGORITHM:

Start
Declare a graph as adjacency matrix
Initialize distance array to INFINITY for all vertices except source (0)
Initialize a visited array to keep track of visited vertices
For all vertices:
Find the unvisited vertex with minimum distance
Mark it as visited
Update the distance of adjacent vertices if a shorter path is found through the current vertex
Repeat until all vertices are visited
Print the distance from source to all vertices
Stop

PROGRAM:

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

#define V 5

int minDistance(int dist[], int visited[]) {
 int min = INT_MAX, min_index;
 for (int v = 0; v < V; v++) {
 if (!visited[v] && dist[v] <= min) {
 min = dist[v];
 min_index = v;
 }
 }
 return min_index;
 }

void dijkstra(int graph[V][V], int src) {
 int dist[V];
```

```
 int visited[V];
 for (int i = 0; i < V; i++) {
 dist[i] = INT_MAX;
 visited[i] = 0;
 }
 dist[src] = 0;
 for (int count = 0; count < V - 1; count++) {
 int u = minDistance(dist, visited);
 visited[u] = 1;
 for (int v = 0; v < V; v++) {
 if (!visited[v] && graph[u][v] && dist[u] != INT_MAX
 && dist[u] + graph[u][v] < dist[v]) {
 dist[v] = dist[u] + graph[u][v];
 }
 }
 }
 printf("Vertex \t Distance from Source\n");
 for (int i = 0; i < V; i++)
 printf("%d \t %d\n", i, dist[i]);
 }

int main() {
 int graph[V][V] = {
 {0, 10, 0, 0, 5},
 {0, 0, 1, 0, 2},
 {0, 0, 0, 4, 0},
 {7, 0, 6, 0, 0},
 {0, 3, 9, 2, 0}
 };
 dijkstra(graph, 0);
 return 0;
 }
```

RESULT:

The implementation of Dijkstra's algorithm was executed and the shortest distances from the source vertex to all other vertices were verified.

EXPT NO : 18 PAGE NO :72 DATE :	HASHING TECHNIQUE

AIM:

To write a C program to implement hashing using chaining (non-open addressing) for insertion, search, and display operations.

ALGORITHM:

Start
Declare a hash table as an array of pointers to linked lists
Create a node structure with data and next pointer
Initialize all hash table entries to NULL
For insertion:
Compute the hash index using key % table_size
Create a new node with the key
Insert the node at the beginning of the linked list at the hash index
For search:
Compute the hash index using key % table_size
Traverse the linked list at that index to find the key
Return success if found, else failure
For display:
For each index in the table, traverse and print all nodes in the linked list
Stop

PROGRAM:

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

#define SIZE 10

typedef struct Node {
 int data;
 struct Node* next;
 } Node;

Node* hashTable[SIZE];

void initialize() {
 for (int i = 0; i < SIZE; i++)
 hashTable[i] = NULL;
 }
```

```
int hashFunction(int key) {
 return key % SIZE;
 }

void insert(int key) {
 int index = hashFunction(key);
 Node* newNode = (Node*)malloc(sizeof(Node));
 newNode->data = key;
 newNode->next = hashTable[index];
 hashTable[index] = newNode;
 printf("Inserted %d at index %d\n", key, index);
 }

void search(int key) {
 int index = hashFunction(key);
 Node* temp = hashTable[index];
 while (temp != NULL) {
 if (temp->data == key) {
 printf("%d found at index %d\n", key, index);
 return;
 }
 temp = temp->next;
 }
 printf("%d not found\n", key);
 }

void display() {
 for (int i = 0; i < SIZE; i++) {
 printf("Index %d: ", i);
 Node* temp = hashTable[i];
 while (temp != NULL) {
 printf("%d -> ", temp->data);
 temp = temp->next;
 }
 printf("NULL\n");
 }
 }

int main() {
 initialize();
 int choice, key;
 while (1) {
 printf("\n1. Insert\n2. Search\n3. Display\n4. Exit\n");
 printf("Enter your choice: ");
 scanf("%d", &choice);
 switch(choice) {
 case 1:
```

```
printf("Enter key to insert: ");
scanf("%d", &key);
insert(key);
break;
case 2:
printf("Enter key to search: ");
scanf("%d", &key);
search(key);
break;
case 3:
display();
break;
case 4:
exit(0);
default:
printf("Invalid choice\n");
}
}
return 0;
}
```

RESULT:

The implementation of hashing using chaining (non-open addressing) was executed and the insertion, search, and display operations were verified.