

Ex. NO: 1**IMPLEMENTATION OF LIST****DATE:****1.A. ARRAY IMPLEMENTATION OF LIST ADT****AIM:**

To write a C program to implement the array using the list ADT.

ALGORITHM:

1. Start
2. Define list using an array of size n.
3. First item with subscript or position = 0
4. Display menu on list operation
5. Accept user choice
6. If choice = 1 then
7. Locate node after which insertion is to be done
8. Create an empty location in array after the node by moving the existing elements one position ahead.
9. Insert the new element at appropriate position
10. Else if choice = 2
11. Get element to be deleted.
12. Locate the position of the element replace the element with the element in the following position.
13. Repeat the step for successive elements until end of the array.
14. Else
15. Traverse the list from position or subscript 0 to n.
16. Stop

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
#define MAX 10
void create();
void insert();
void deletion();
void search();
void display();
int a,b[20], n, p, e, f, i, pos;
void main()
{
//clrscr();
int ch;
char g='y';
do
{
printf("\n main Menu");
```

```
printf("\n 1.Create \n 2.Delete \n 3.Search \n 4.Insert \n 5.Display \n");
printf("\n Enter your Choice");
scanf("%d", &ch);
switch(ch)
{
case 1:
create();
break;
case 2:
deletion();
break;
case 3:
search();
break;
case 4:
insert();
break;
case 5:
display();
break;
default:
printf("\n Enter the correct choice:");
}
printf("\n Do u want to continue::");
scanf("\n%c", &g);
}
while(g=='y'||g=='Y');
}
void create()
{
printf("\n Enter the number of nodes");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the Element %d:",i+1);
scanf("%d", &b[i]);
}
}
void deletion()
{
printf("\n Enter the position u want to delete::");
scanf("%d", &pos);
if(pos>=n)
```

```
{
printf("\n Invalid Location::");
}
else
{
for(i=pos+1;i<n;i++)
{
b[i-1]=b[i];
}
n--;
}
printf("\n The Elements after deletion");
for(i=0;i<n;i++)
{
printf("\t%d", b[i]);
}
}
void search()
{
printf("\n Enter the Element to be searched:");
scanf("%d", &e);
for(i=0;i<n;i++)
{
if(b[i]==e)
{
printf("Value is in the %d Position", i);
}
else
{
printf("Value %d is not in the list::", e);
continue;
}
}
}
void insert()
{
printf("\n Enter the position u need to insert::");
scanf("%d", &pos);
if(pos>=n)
{
printf("\n invalid Location::");
}
else
```

```
{
for(i=MAX-1;i>=pos-1;i--)
{
b[i+1]=b[i];
}
printf("\n Enter the element to insert::\n");
scanf("%d",&p);
b[pos]=p;
n++;
}
printf("\n The list after insertion::\n");
display();
}
void display()
{
printf("\n The Elements of The list ADT are:");
for(i=0;i<n;i++)
{
printf("\n\n%d", b[i]);
}
}
```

OUTPUT:

```
main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display

Enter your Choice1

Enter the number of nodes5

Enter the Element 1:10

Enter the Element 2:20

Enter the Element 3:30

Enter the Element 4:40

Enter the Element 5:50
```

```
main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display

Enter your Choice3

Enter the Element to be searched:30
Value is in the 2 Position
Do u want to continue:::
Y
```

```
main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display

Enter your Choice4

Enter the position u need to insert::3

Enter the element to insert::
25

The list after insertion::

The Elements of The list ADT are:
10
20
30|
25

40
Do u want to continue::Y
```

```
main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display

Enter your Choice2

Enter the position u want to delete::4

The Elements after deletion    10  20  30  40
Do u want to continue::Y
```

```
main Menu
1.Create
2.Delete
3.Search
4.Insert
5.Display

Enter your Choice5

The Elements of The list ADT are:

10

20

30

25

40

Do u want to continue::N
```

1.B. DESIGN AND IMPLEMENT SINGLY LINKED LIST

AIM:

To design and implement a singly linked list and perform operations such as insertion, deletions, displaying dynamically.

ALGORITHM:

1. Start.
2. Define a node structure containing data and a next pointer, and initialize head=NULL.
3. Display the menu of linked list operations to the user.

4. Read the user's choice.
5. If the choice is Insert at Beginning, create a new node and make it the new head of the list.
6. If the choice is Insert at End, create a new node, traverse to the last node, and link the new node at the end.
7. If the choice is Insert at Position, count the number of nodes, validate the position, traverse to the required position, and insert the new node.
8. If the choice is Delete from Beginning, check whether the list is empty; otherwise, remove the first node and update the head.
9. If the choice is Delete from End, traverse to the last node, remove it, and update the second-last node's link.
10. If the choice is Delete from Position, validate the position, traverse to the specified node, remove it, and reconnect the links.
11. If the choice is Display, traverse the linked list from the head and print each node's data.
12. If the list is empty during any operation, display an appropriate error message.
13. Repeat the menu until the user selects the Exit option.
14. Release allocated memory for deleted nodes during deletion operations.
15. Stop.

PROGRAM:

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

struct node{
    int data;
    struct node*next;
}*head=NULL;

int count()
{
    struct node *temp;
    int i=1;
    temp=head;
    while(temp->next!=NULL)
    {
        temp=temp->next;
        i++;
    }
    return(i);
}

struct node *create(int value)
{
    struct node *temp;
    temp=(struct node*)malloc(sizeof(struct node));
```

```
temp->data=value;
temp->next=NULL;
return temp;
}
void insert_begin(int value)
{
    struct node *newnode;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        newnode->next=head;
        head=newnode;
    }
}

void insert_end(int value)
{
    struct node *newnode, *temp;
    newnode=create(value);
    if(head==NULL)
    {
        head=newnode;
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=newnode;
    }
}

void insert_pos(int value,int pos)
{
    struct node *newnode, *temp1,*temp2;
    int i,c=1;
    newnode=create(value);
```

```
i=count();
if(pos==1)
    insert_begin(value);
else if(pos>i+1)
{
    printf("insertion is not possible");
    return;
}
else
{
    temp1=head;
    while(c<=pos-1 && temp1!=NULL)
    {
        temp2=temp1;
        temp1=temp1->next;
        c++;
    }
    newnode->next=temp2->next;
    temp2->next=newnode;
}
}

void delete_begin()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
    else
    {
        temp=head;
        head=head->next;
        free(temp);
    }
}

void delete_end()
{
    struct node *temp1,*temp2;
    if(head==NULL)
    {
        printf("deletion is not possible");
    }
}
```

```
    else
    {
        temp1=head;
        while(temp1->next!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
        }
        temp2->next=NULL;
        free(temp1);
    }
}
void delete_pos(int pos)
{
    struct node *temp1,*temp2;
    int i,c=1;
    i=count();
    if(pos==1)
        delete_begin();
    else if(pos>i)
    {
        printf("Deletion is not possible");
        return;
    }
    else
    {
        temp1=head;
        while(c<=pos-1 && temp1->next!=NULL)
        {
            temp2=temp1;
            temp1=temp1->next;
            c++;
        }
        temp2->next=temp1->next;
        free(temp1);
    }
}
void display()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("list is empty");
    }
}
```

```
    }
    else
    {
        temp=head;
        while(temp->next!=NULL)
        {
            printf("%d-> ",temp->data);
            temp=temp->next;
        }
        printf("%d",temp->data);
    }
}

void main()
{
    int ch,pos,value;
    do
    {
        printf("\n1.insert begin\n2.insert end\n3.insert position\n4.delete begin\n5.delete
end\n6.delete position\n7.display\n8.exit\n");
        printf("enter your choice:");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("enter the value:");
                    scanf("%d",&value);
                    insert_begin(value);
                    display();
                    break;
            case 2: printf("enter value:");
                    scanf("%d",&value);
                    insert_end(value);
                    display();
                    break;
            case 3: printf("enter value:");
                    scanf("%d",&value);
                    printf("enter position you want to insert: ");
                    scanf("%d",&pos);
                    insert_pos(value,pos);
                    display();
                    break;
            case 4: delete_begin();
                    display();
```

```

        break;
    case 5: delete_end();
            display();
            break;
    case 6: printf("enter position you want to delete: ");
            scanf("%d",&pos);
            delete_pos(pos);
            display();
            break;
    case 7: display();
            break;
    case 8: break;
    default: printf("\nyour choice is wrong!.. ");
    }
}while(ch!=8);
}

```

OUTPUT:

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:1
enter the value:10
10

```

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:2
enter value:20
10-> 20

```

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:2
enter value:50
10-> 20-> 30-> 40-> 50

```

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:3
enter value:25
enter position you want to insert: 3
10-> 20-> 25-> 30-> 40-> 50

```

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:4
20-> 25-> 30-> 40-> 50

```

```

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:5
20-> 25-> 30-> 40

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:6
enter position you want to delete: 3
20-> 25-> 40

1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:7
20-> 25-> 40
1.insert begin
2.insert end
3.insert position
4.delete begin
5.delete end
6.delete position
7.display
8.exit
enter your choice:8

...Program finished with exit code 8

```

1.C. DESIGN AND IMPLEMENT DOUBLY LINKED LIST

AIM:

To design and implement a doubly linked list and perform operations such as insertion, deletions, displaying dynamically.

ALGORITHM:

1. Start.
2. Define a node with three fields: data, prev, and next.
3. Initialize the head pointer to NULL.
4. Create a new node by allocating memory, storing the data, and setting both prev and next to NULL.
5. Insert at the beginning by making the new node point to the current head, updating the head's prev pointer, and moving head to the new node.
6. Insert at the end by traversing to the last node, linking the new node after it, and updating the prev pointer of the new node.
7. Insert at a specified position by traversing to the required location and updating the prev and next pointers of the neighboring nodes and the new node.
8. Delete from the beginning by moving the head to the next node, setting its prev pointer to NULL, and freeing the first node.
9. Delete from the end by traversing to the last node, updating the second-last node's next pointer to NULL, and freeing the last node.

10. Delete from a specified position by traversing to the node, reconnecting its previous and next nodes, and freeing the selected node.
11. Count the number of nodes by traversing the list from the head until NULL is reached.
12. Display the list in forward direction by traversing from head to the last node using the next pointer.
13. Display the list in backward direction by first reaching the last node and then traversing to the head using the prev pointer.
14. Repeat the menu-driven operations until the user chooses the Exit option.
15. Stop.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *prev;
    struct node *next;
} *head = NULL;
struct node* create(int value)
{
    struct node *temp;
    temp = (struct node*)malloc(sizeof(struct node));
    temp->data = value;
    temp->prev = NULL;
    temp->next = NULL;
    return temp;
}
int count()
{
    int c = 0;
    struct node *temp = head;
    while(temp != NULL)
    {
        c++;
        temp = temp->next;
    }
    return c;
}
void insert_begin(int value)
{
    struct node *newnode = create(value);
    if(head == NULL)
    {
        head = newnode;
```

```
    }
    else
    {
        newnode->next = head;
        head->prev = newnode;
        head = newnode;
    }
}
void insert_end(int value)
{
    struct node *newnode = create(value);
    struct node *temp;
    if(head == NULL)
    {
        head = newnode;
    }
    else
    {
        temp = head;

        while(temp->next != NULL)
            temp = temp->next;
        temp->next = newnode;
        newnode->prev = temp;
    }
}
void insert_pos(int value,int pos)
{
    int c = count();
    if(pos<1 || pos>c+1)
    {
        printf("Insertion not possible\n");
        return;
    }
    if(pos==1)
    {
        insert_begin(value);
        return;
    }
    if(pos==c+1)
    {
        insert_end(value);
        return;
    }
    struct node *newnode = create(value);
    struct node *temp = head;
```

```
    int i;
    for(i=1;i<pos-1;i++)
        temp=temp->next;
    newnode->next = temp->next;
    newnode->prev = temp;
    temp->next->prev = newnode;
    temp->next = newnode;
}
void delete_begin()
{
    struct node *temp;
    if(head==NULL)
    {
        printf("List is empty\n");
        return;
    }
    temp=head;
    if(head->next==NULL)
    {
        head=NULL;
    }
    else
    {
        head=head->next;
        head->prev=NULL;
    }
    free(temp);
}
void delete_end()
{
    if(head==NULL)
    {
        printf("List is empty\n");
        return;
    }
    if(head->next==NULL)
    {
        free(head);
        head=NULL;
        return;
    }
    struct node *temp=head;
    while(temp->next!=NULL)
        temp=temp->next;
    temp->prev->next=NULL;
```

```
        free(temp);
    }
void delete_pos(int pos)
{
    int c=count();
    if(pos<1 || pos>c)
    {
        printf("Deletion not possible\n");
        return;
    }
    if(pos==1)
    {
        delete_begin();
        return;
    }
    if(pos==c)
    {
        delete_end();
        return;
    }
    struct node *temp=head;
    int i;
    for(i=1;i<pos;i++)
        temp=temp->next;
    temp->prev->next=temp->next;
    temp->next->prev=temp->prev;
    free(temp);
}
void display_forward()
{
    struct node *temp=head;
    if(head==NULL)
    {
        printf("List is empty\n");
        return;
    }
    while(temp!=NULL)
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }
}
void display_backward()
{
    struct node *temp=head;
    if(head==NULL)
```

```
{
    printf("List is empty\n");
    return;
}
while(temp->next!=NULL)
    temp=temp->next;
while(temp!=NULL)
{
    printf("%d ",temp->data);
    temp=temp->prev;
}
}
int main()
{
    int ch,value,pos;
    do
    {
        printf("\n1.Insert Begin");
        printf("\n2.Insert End");
        printf("\n3.Insert Position");
        printf("\n4.Delete Begin");
        printf("\n5.Delete End");
        printf("\n6.Delete Position");
        printf("\n7.Display Forward");
        printf("\n8.Display Backward");
        printf("\n9.Exit");
        printf("\nEnter choice : ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                printf("Enter value : ");
                scanf("%d",&value);
                insert_begin(value);
                printf("The Forward Traversal\n");
                display_forward();
                printf("\nThe Backward Traversal\n");
                display_backward();
                break;
            case 2:
                printf("Enter value : ");
                scanf("%d",&value);
                insert_end(value);
                printf("The Forward Traversal\n");
                display_forward();
                printf("\nThe Backward Traversal\n");
```

```
        display_backward();
        break;
case 3:
    printf("Enter value : ");
    scanf("%d",&value);
    printf("Enter position : ");
    scanf("%d",&pos);
    insert_pos(value,pos);
    printf("The Forward Traversal\n");
    display_forward();
    printf("\nThe Backward Traversal\n");
    display_backward();
    break;
case 4:
    delete_begin();
    printf("The Forward Traversal\n");
    display_forward();
    printf("\nThe Backward Traversal\n");
    display_backward();
    break;
case 5:
    delete_end();
    printf("The Forward Traversal\n");
    display_forward();
    printf("\nThe Backward Traversal\n");
    display_backward();
    break;
case 6:
    printf("Enter position : ");
    scanf("%d",&pos);
    delete_pos(pos);
    printf("The Forward Traversal\n");
    display_forward();
    printf("\nThe Backward Traversal\n");
    display_backward();
    break;
case 7:
    printf("The Forward Traversal\n");
    display_forward();
    break;
case 8:
    printf("\nThe Backward Traversal\n");
    display_backward();
    break;
case 9:
    break;
```

```

        default:
            printf("Invalid Choice");
        }
    }while(ch!=9);
    return 0;
}

```

OUTPUT:

<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 1 Enter value : 10 The Forward Traversal 10 The Backward Traversal 10 </pre>	<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 2 Enter value : 20 The Forward Traversal 10 20 The Backward Traversal 20 10 </pre>	<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 3 Enter value : 25 Enter position : 3 The Forward Traversal 10 20 25 30 40 50 The Backward Traversal 50 40 30 25 20 10 </pre>
<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 4 The Forward Traversal 20 25 30 40 50 The Backward Traversal 50 40 30 25 20 </pre>	<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 5 The Forward Traversal 20 25 30 40 The Backward Traversal 40 30 25 20 </pre>	<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 6 Enter position : 2 The Forward Traversal 20 30 40 The Backward Traversal 40 30 20 </pre>
<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 7 The Forward Traversal 20 30 40 </pre>	<pre> 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : 8 The Backward Traversal 40 30 20 1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Exit Enter choice : ...Program finished with exit code </pre>	

1.D. DESIGN AND IMPLEMENT CIRCULARLY LINKED LIST**AIM:**

To design and implement a circularly linked list and perform operations such as insertion, deletions, displaying dynamically.

ALGORITHM:

1. Start the program.
2. Define a node with two fields: data and next.
3. Initialize the head pointer to NULL.
4. Create a new node by allocating memory, storing the data, and setting its next pointer to NULL.
5. Insert at the beginning by linking the new node before the head, updating the last node's next pointer to the new node, and making the new node the head.
6. Insert at the end by traversing to the last node, linking the new node after it, and setting the new node's next pointer to the head.
7. Insert at a specified position by traversing to the required position and adjusting the next pointers of the surrounding nodes.
8. Delete from the beginning by moving the head to the next node, updating the last node's next pointer to the new head, and freeing the old head node.
9. Delete from the end by traversing to the last node, making the previous node point to the head, and freeing the last node.
10. Delete from a specified position by traversing to the desired node, updating the previous node's next pointer, and freeing the selected node.
11. Count the number of nodes by traversing the list from the head until the traversal returns to the head.
12. Display the list by traversing from the head and printing each node's data until the head is reached again.
13. Maintain the circular property by ensuring that the last node always points back to the head after every insertion and deletion.
14. Repeat the menu-driven operations until the user selects the Exit option.
15. Stop.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *prev;
    struct node *next;
} *head = NULL;
struct node* create(int value)
{
```

```
    struct node *newnode;
    newnode = (struct node*)malloc(sizeof(struct node));
    newnode->data = value;
    newnode->prev = NULL;
    newnode->next = NULL;
    return newnode;
}
int count()
{
    if(head == NULL)
        return 0;
    int c = 0;
    struct node *temp = head;
    do
    {
        c++;
        temp = temp->next;
    }while(temp != head);
    return c;
}
void insert_begin(int value)
{
    struct node *newnode = create(value);
    if(head == NULL)
    {
        head = newnode;
        head->next = head;
        head->prev = head;
    }
    else
    {
        struct node *last = head->prev;
        newnode->next = head;
        newnode->prev = last;
        last->next = newnode;
        head->prev = newnode;
        head = newnode;
    }
}
void insert_end(int value)
{
    struct node *newnode = create(value);
    if(head == NULL)
    {
        head = newnode;
        head->next = head;
    }
```

```
        head->prev = head;
    }
    else
    {
        struct node *last = head->prev;
        newnode->next = head;
        newnode->prev = last;
        last->next = newnode;
        head->prev = newnode;
    }
}

void insert_pos(int value, int pos)
{
    int n = count();
    if(pos < 1 || pos > n + 1)
    {
        printf("Insertion not possible\n");
        return;
    }
    if(pos == 1)
    {
        insert_begin(value);
        return;
    }
    if(pos == n + 1)
    {
        insert_end(value);
        return;
    }
    struct node *newnode = create(value);
    struct node *temp = head;
    int i;
    for(i = 1; i < pos - 1; i++)
        temp = temp->next;
    newnode->next = temp->next;
    newnode->prev = temp;
    temp->next->prev = newnode;
    temp->next = newnode;
}

void delete_begin()
{
    if(head == NULL)
    {
        printf("List is empty\n");
        return;
    }
}
```

```
    if(head->next == head)
    {
        free(head);
        head = NULL;
        return;
    }
    struct node *last = head->prev;
    struct node *temp = head;
    head = head->next;
    head->prev = last;
    last->next = head;
    free(temp);
}
void delete_end()
{
    if(head == NULL)
    {
        printf("List is empty\n");
        return;
    }
    if(head->next == head)
    {
        free(head);
        head = NULL;
        return;
    }
    struct node *last = head->prev;
    last->prev->next = head;
    head->prev = last->prev;
    free(last);
}
void delete_pos(int pos)
{
    int n = count();
    if(pos < 1 || pos > n)
    {
        printf("Deletion not possible\n");
        return;
    }
    if(pos == 1)
    {
        delete_begin();
        return;
    }
    if(pos == n)
    {
        delete_end();
        return;
    }
}
```

```
        delete_end();
        return;
    }
    struct node *temp = head;
    int i;
    for(i = 1; i < pos; i++)
        temp = temp->next;
    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    free(temp);
}
void display_forward()
{
    if(head == NULL)
    {
        printf("List is empty\n");
        return;
    }
    struct node *temp = head;
    do
    {
        printf("%d ", temp->data);
        temp = temp->next;
    }while(temp != head);
    printf("%d", head->data);
}
void display_backward()
{
    if(head == NULL)
    {
        printf("List is empty\n");
        return;
    }
    struct node *temp = head->prev;
    do
    {
        printf("%d ", temp->data);
        temp = temp->prev;
    }while(temp != head->prev);
    printf("%d", temp->data);
}
int main()
{
    int ch, value, pos;
    do
    {
```

```
printf("\n1.Insert Begin");
printf("\n2.Insert End");
printf("\n3.Insert Position");
printf("\n4.Delete Begin");
printf("\n5.Delete End");
printf("\n6.Delete Position");
printf("\n7.Display Forward");
printf("\n8.Display Backward");
printf("\n9.Count");
printf("\n10.Exit");
printf("\nEnter choice: ");
scanf("%d",&ch);
switch(ch)
{
    case 1:
        printf("Enter value: ");
        scanf("%d",&value);
        insert_begin(value);
        printf("The Forward Traversal\n");
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 2:
        printf("Enter value: ");
        scanf("%d",&value);
        insert_end(value);
        printf("The Forward Traversal\n");
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 3:
        printf("Enter value: ");
        scanf("%d",&value);
        printf("Enter position: ");
        scanf("%d",&pos);
        insert_pos(value,pos);
        printf("The Forward Traversal\n");
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 4:
        delete_begin();
        printf("The Forward Traversal\n");
```

```
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 5:
        delete_end();
        printf("The Forward Traversal\n");
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 6:
        printf("Enter position: ");
        scanf("%d",&pos);
        delete_pos(pos);
        printf("The Forward Traversal\n");
        display_forward();
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 7:
        printf("The Forward Traversal\n");
        display_forward();
        break;
    case 8:
        printf("\nThe Backward Traversal\n");
        display_backward();
        break;
    case 9:
        printf("Count = %d\n",count());
        break;
    case 10:
        printf("Program Ended\n");
        break;
    default:
        printf("Invalid Choice");
    }
}while(ch!=10);
return 0;
}
```

OUTPUT:

1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 1 Enter value: 10 The Forward Traversal 10 10 The Backward Traversal 10 10	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 2 Enter value: 20 The Forward Traversal 10 20 10 The Backward Traversal 20 10 20	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 3 Enter value: 25 Enter position: 3 The Forward Traversal 10 20 25 30 40 50 10 The Backward Traversal 50 40 30 25 20 10 50	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 4 The Forward Traversal 20 25 30 40 50 20 The Backward Traversal 50 40 30 25 20 50
--	--	---	---

1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 5 The Forward Traversal 20 25 30 40 20 The Backward Traversal 40 30 25 20 40	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 6 Enter position: 2 The Forward Traversal 20 30 40 20 The Backward Traversal 40 30 20 40	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 7 The Forward Traversal 20 30 40 20	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 8 The Backward Traversal 40 30 20 40
---	--	--	---

1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 9 Count = 3	1.Insert Begin 2.Insert End 3.Insert Position 4.Delete Begin 5.Delete End 6.Delete Position 7.Display Forward 8.Display Backward 9.Count 10.Exit Enter choice: 10 Program Ended ...Program finished with exit
---	---

RESULTS:

Thus the programs of Array implementation using list ADT, Singly Linked List, Doubly Linked List, Circularly Linked List and their operations are implemented and executed successfully.

Ex. NO: 2

IMPLEMENTATION OF STACK AND QUEUE

DATE:

2.A. DESIGN AND IMPLEMENT STACK AND ITS OPERATION USING LIST**AIM:**

To write a C program to design and implement stack and its operation using list.

ALGORITHM:

1. Start the program and initialize the stack by setting top = NULL.
2. Display the menu and read the user's choice (Push, Pop, Peek, Display, IsEmpty, Exit).
3. Perform the selected operation:
 - a. Push: Create a new node, insert the value at the top, and update top.
 - b. Pop: If the stack is empty, display "Stack Underflow"; otherwise, remove the top node and update top.
4. Peek: If the stack is not empty, display the top element; otherwise, display "Stack is empty".
5. Display/IsEmpty: Display all stack elements from top to bottom or check whether the stack is empty.
6. Repeat steps 2–5 until the user chooses Exit, then Stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *next;
};
struct Node *top = NULL;
void push(int value) {
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Stack Overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed into stack.\n", value);
}
void pop() {
    struct Node *temp;
    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }
    temp = top;
    top = top->next;
    free(temp);
}
```

```
    }
    temp = top;
    printf("%d popped from stack.\n", top-
>data); top = top->next;
    free(temp);
}
void peek() {
    if (top == NULL) {
        printf("Stack is empty.\n");
    }
    else {
        printf("Top element:
%d\n", top->data); }
}
void display() {
    struct Node *temp;
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements are:\n");
    temp = top;
    while (temp != NULL) {
        printf("%d\n", temp->data);
        temp = temp->next;
    }
}
void isEmpty() {
    if (top == NULL)
        printf("Stack is empty.\n");
    else
        printf("Stack is not empty.\n");
}
int main() {
    int choice, value;
    while (1) {
        printf("\n--- STACK USING LINKED LIST ---
\n"); printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Is Empty\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
```

```

        case 1:
            printf("Enter value: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            peek();
            break;
        case 4:
            display();
            break;
        case 5:
            isEmpty();
            break;
        case 6:
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice!\n");
    }
}
return 0;
}

```

OUTPUT:

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 1
Enter value: 10
10 pushed into stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 1
Enter value: 20
20 pushed into stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 1
Enter value: 30
30 pushed into stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 1
Enter value: 40
40 pushed into stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 1
Enter value: 50
50 pushed into stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 2
50 popped from stack.

```

```

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 3
Top element: 40

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 4
Stack elements are:
40
30
20
10

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 5
Stack is not empty.

--- STACK USING LINKED LIST ---
1. Push
2. Pop
3. Peek
4. Display
5. Is Empty
6. Exit
Enter your choice: 6
Exiting...
...Program finished with exit code 0

```

2.B. DESIGN AND IMPLEMENT QUEUE AND ITS OPERATION USING LIST

AIM:

To write a C program to design and implement queue and its operation using list.

ALGORITHM:

1. Start the program and initialize the queue by setting FRONT = NULL and REAR = NULL.
2. Display the menu and read the user's choice (Enqueue, Dequeue, Peek, Display, Exit).
3. If the choice is Enqueue, create a new node, insert it at the REAR of the queue, and update REAR (and FRONT if the queue is empty).
4. If the choice is Dequeue, check whether the queue is empty. If empty, display "Queue Underflow"; otherwise, remove the FRONT node and update FRONT (and REAR if the queue becomes empty).
5. If the choice is Peek, display the FRONT element. If the choice is Display, traverse the queue from FRONT to REAR and display all elements.
6. Repeat Steps 2–5 until the user selects Exit, then stop the program.

PROGRAM:

```

#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *next;
};
struct Node *front = NULL, *rear = NULL;
void enqueue(int value) {
    struct Node *newNode;
    newNode = (struct Node *)malloc(sizeof(struct Node));
    if (newNode == NULL) {

```

```
        printf("Queue Overflow\n");
        return;
    }
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
    printf("%d inserted into queue.\n", value);
}

void dequeue() {
    struct Node *temp;
    if (front == NULL) {
        printf("Queue Underflow\n");
        return;
    }
    temp = front;
    printf("%d deleted from queue.\n", front->data);
    front = front->next;
    if (front == NULL)
        rear = NULL;
    free(temp);
}

void peek() {
    if (front == NULL)
        printf("Queue is empty.\n");
    else
        printf("Front element: %d\n", front->data);
}

void display() {
    struct Node *temp;
    if (front == NULL) {
        printf("Queue is empty.\n");
        return;
    }
    printf("Queue elements are:\n");
    temp = front;
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}
```

```
int main() {
int choice, value;
while (1) {
printf("\n--- QUEUE USING LINKED LIST ---\n");
printf("1. Enqueue\n");
printf("2. Dequeue\n");
printf("3. Peek\n");
printf("4. Display\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);
switch (choice) {
    case 1:
        printf("Enter value: ");
        scanf("%d", &value);
        enqueue(value);
        break;
    case 2:
        dequeue();
        break;
    case 3:
        peek();
        break;
    case 4:
        display();
        break;
    case 5:
        printf("Exiting...\n");
        exit(0);
    default:
        printf("Invalid choice!\n");
        }
    }
return 0;
}
```

OUTPUT:

```
--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 10
10 inserted into queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 20
20 inserted into queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 30
30 inserted into queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 40
40 inserted into queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value: 50
50 inserted into queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
10 deleted from queue.

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element: 20

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements are:
20 30 40 50

--- QUEUE USING LINKED LIST ---
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting...

...Program finished with exit code 0
```

RESULT:

Thus the programs of stack and queue implementation and its operations using list are executed successfully.

Ex. NO: 3 APPLICATIONS OF LINEAR DATA STRUCTURES DATE:

3.A. DESIGN AND IMPLEMENT POLYNOMIAL ADT USING LIST

AIM:

To write a C program to design and implement polynomial ADT using list.

ALGORITHM:

1. Start the program and create two empty linked lists for the two polynomials.
2. Input the terms (coefficient and exponent) of both polynomials and insert each term in descending order of exponents, combining like terms.
3. Display the first and second polynomials.
4. Compare the exponents of corresponding terms in both polynomials and add the coefficients when exponents are equal; otherwise, copy the term with the larger exponent to the result.
5. Append any remaining terms from either polynomial to the result polynomial.
6. Display the resulting polynomial, free the allocated memory, and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node
{
    int coeff;
    int exp;
    struct Node *next;
} Node;

Node *createNode(int coeff, int exp)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;
    return newNode;
}

Node *insert(Node *head, int coeff, int exp)
{
    if (coeff == 0)
        return head;

    Node *newNode = createNode(coeff, exp);
```

```
if (head == NULL || exp > head->exp)
{
    newNode->next = head;
    return newNode;
}
Node *temp = head;
Node *prev = NULL;

while (temp != NULL && temp->exp > exp)
{
    prev = temp;
    temp = temp->next;
}
if (temp != NULL && temp->exp == exp)
{
    temp->coeff += coeff;
    free(newNode);
    if (temp->coeff == 0)
    {
        if (prev == NULL)
        {
            head = temp->next;
        }
        else
        {
            prev->next = temp->next;
        }
        free(temp);
    }
}
else
{
    newNode->next = temp;
    if (prev != NULL)
        prev->next = newNode;
}
return head;
}
Node *createPolynomial()
{
    Node *head = NULL;
    int n, coeff, exp;
    printf("Enter number of terms: ");
```

```
scanf("%d", &n);
printf("Enter coefficient and exponent:\n");
for (int i = 0; i < n; i++)
{
    scanf("%d %d", &coeff, &exp);
    head = insert(head, coeff, exp);
}
return head;
}
void display(Node *head)
{
    if (head == NULL)
    {
        printf("0\n");
        return;
    }
    while (head != NULL)
    {
        printf("%dx^%d", head->coeff, head->exp);
        if (head->next != NULL)
            printf(" + ");
        head = head->next;
    }
    printf("\n");
}
Node *add(Node *p1, Node *p2)
{
    Node *result = NULL;
    while (p1 != NULL && p2 != NULL)
    {
        if (p1->exp > p2->exp)
        {
            result = insert(result, p1->coeff, p1->exp);
            p1 = p1->next;
        }
        else if (p2->exp > p1->exp)
        {
            result = insert(result, p2->coeff, p2->exp);
            p2 = p2->next;
        }
        else
        {
            result = insert(result, p1->coeff + p2->coeff, p1->exp);
```

```
        p1 = p1->next;
        p2 = p2->next;
    }
}
while (p1 != NULL)
{
    result = insert(result, p1->coeff, p1->exp);
    p1 = p1->next;
}
while (p2 != NULL)
{
    result = insert(result, p2->coeff, p2->exp);
    p2 = p2->next;
}
return result;
}

void freeList(Node *head)
{
    Node *temp;
    while (head != NULL)
    {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main()
{
    Node *poly1, *poly2, *sum;
    printf("Create First Polynomial\n");
    poly1 = createPolynomial();
    printf("\nCreate Second Polynomial\n");
    poly2 = createPolynomial();
    printf("\nFirst Polynomial: ");
    display(poly1);
    printf("Second Polynomial: ");
    display(poly2);
    sum = add(poly1, poly2);
    printf("Sum Polynomial: ");
    display(sum);
    freeList(poly1);
    freeList(poly2);
}
```

```
    freeList(sum);  
    return 0;  
}
```

OUTPUT:

```
Create First Polynomial  
Enter number of terms: 3  
Enter coefficient and exponent:  
2 3  
3 2  
4 1  
  
Create Second Polynomial  
Enter number of terms: 3  
Enter coefficient and exponent:  
5 3  
7 2  
1 1  
  
First Polynomial: 2x^3 + 3x^2 + 4x^1  
Second Polynomial: 5x^3 + 7x^2 + 1x^1  
Sum Polynomial: 7x^3 + 10x^2 + 5x^1  
  
...Program finished with exit code 0
```

3.B. USES STACK OPERATIONS TO CONVERT INFIX EXPRESSION INTO POSTFIX EXPRESSION**AIM:**

To write a C program to that uses stack operations to convert infix expression into postfix expression.

ALGORITHM:

1. Start the program and input the infix expression.
2. Scan the infix expression from left to right, one character at a time.
3. If the scanned character is an operand, add it directly to the postfix expression; if it is '(', push it onto the stack.
4. If the scanned character is ')', pop and append operators from the stack until '(' is encountered, then remove '(' from the stack.
5. If the scanned character is an operator, pop and append all operators from the stack with higher or equal precedence, then push the current operator onto the stack. After scanning the entire expression, pop and append any remaining operators.
6. Display the postfix expression and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100
char stack[MAX];
int top = -1;
void push(char ch)
{
    stack[++top] = ch;
}
char pop()
{
    if (top == -1)
        return '\0';
    return stack[top--];
}
int precedence(char ch)
{
    switch (ch)
    {
        case '^':
            return 3;
        case '*':
        case '/':
            return 2;
        case '+':
        case '-':
            return 1;
        default:
            return 0;
    }
}
int main()
{
    char infix[MAX], postfix[MAX];
    int i = 0, j = 0;
    printf("Enter an infix expression: ");
    scanf("%s", infix);
    while (infix[i] != '\0')
    {
        char ch = infix[i];
        if (isalnum(ch))
        {
            postfix[j++] = ch;
        }
        else if (ch == '(')    {
```

```
        push(ch);
    }
    else if (ch == ')') {
        while (top != -1 && stack[top] != '(')
        {
            postfix[j++] = pop();
        }
        pop();
    }
    else {
        while (top != -1 && precedence(stack[top]) >= precedence(ch))
        {
            postfix[j++] = pop();
        }
        push(ch);
    }
    i++;
}
while (top != -1)
{
    postfix[j++] = pop();
}
postfix[j] = '\0';
printf("Postfix Expression: %s\n", postfix);
return 0;
}
```

OUTPUT:

```
Enter an infix expression: A+B*C/D-E
Postfix Expression: ABC*D/+E-
```

```
Enter an infix expression: (A-B)*(C+D)/E
Postfix Expression: AB-CD+*E/
```

```
Enter an infix expression: (2+3)/(5-1)*10
Postfix Expression: 23+51-/10*
```

3.C. USES STACK OPERATIONS FOR EVALUATING THE POSTFIX EXPRESSION**AIM:**

To write a C program to that uses stack operations for evaluating the postfix expression.

ALGORITHM:

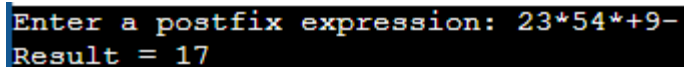
1. Start the program and input the postfix expression.
2. Scan the postfix expression from left to right, one character at a time.
3. If the scanned character is an operand, push it onto the stack.

4. If the scanned character is an operator, pop the top two operands from the stack, perform the required operation, and push the result back onto the stack.
5. Repeat Steps 3 and 4 until all characters in the postfix expression have been processed.
6. Pop the final value from the stack, display it as the result of the postfix expression, and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100
int stack[MAX];
int top = -1;
void push(int value)
{
    stack[++top] = value;
}
int pop()
{
    if (top == -1)
    {
        printf("Stack Underflow\n");
        return -1;
    }
    return stack[top--];
}
int main()
{
    char postfix[MAX];
    int i = 0, op1, op2, result;
    printf("Enter a postfix expression: ");
    scanf("%s", postfix);
    while (postfix[i] != '\0')
    {
        char ch = postfix[i];
        if (isdigit(ch))
        {
            push(ch - '0');
        }
        else
        {
            op2 = pop();
            op1 = pop();
            switch (ch)
            {
                case '+':
```

```
        result = op1 + op2;
        break;
    case '-':
        result = op1 - op2;
        break;
    case '*':
        result = op1 * op2;
        break;
    case '/':
        result = op1 / op2;
        break;
    case '%':
        result = op1 % op2;
        break;
    default:
        printf("Invalid Operator\n");
        return 1;
    }
    push(result);
}
i++;
}
printf("Result = %d\n", pop());
return 0;
}
```

OUTPUT:

```
Enter a postfix expression: 23*54*+9-
Result = 17
```

RESULT:

Thus the C program to design and implement polynomial ADT using list, conversion of infix expression into postfix expression and evaluating the postfix expression using stack operations were implemented and executed successfully.

Ex. NO: 4**IMPLEMENTATION OF TREE****DATE:****4.A. DESIGN AND IMPLEMENT BINARY SEARCH TREE****AIM:**

To write a C program to design and implement binary search tree.

ALGORITHM:

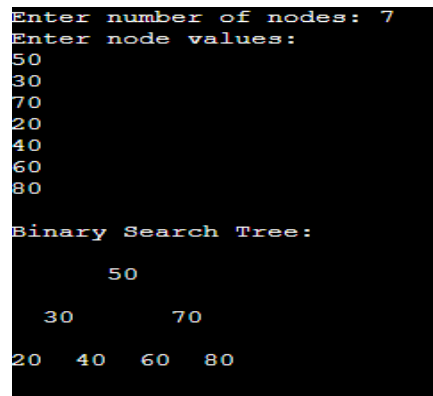
1. Start the program and create an empty Binary Search Tree (BST).
2. Read the number of nodes and insert each node into the BST by comparing the value with the current node and placing it in the left subtree if smaller or the right subtree if larger.
3. Calculate the height of the Binary Search Tree.
4. Traverse the tree level by level from the root to the last level.
5. Print each level with appropriate spacing so that the nodes are displayed in a hierarchical tree structure.
6. Display the Binary Search Tree and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
struct Node
{
    int data;
    struct Node *left, *right;
};
struct Node *createNode(int data)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}
struct Node *insert(struct Node *root, int data)
{
    if (root == NULL)
        return createNode(data);
    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);
    return root;
}
int height(struct Node *root)
{
    if (root == NULL)
```

```
        return 0;
    int l = height(root->left);
    int r = height(root->right);
    return (l > r ? l : r) + 1;
}
void printSpaces(int n)
{
    while (n--)
        printf(" ");
}
void printLevel(struct Node *root, int level, int space)
{
    if (level == 1) {
        if (root)
            printf("%2d", root->data);
        else
            printf(" ");
        printSpaces(space);    }
    else {
        if (root) {
            printLevel(root->left, level - 1, space);
            printLevel(root->right, level - 1, space); }
        else {
            printLevel(NULL, level - 1, space);
            printLevel(NULL, level - 1, space); }
    }
}
void printTree(struct Node *root)
{
    int h = height(root);
    printf("\nBinary Search Tree:\n\n");
    for (int i = 1; i <= h; i++) {
        int leading = (1 << (h - i)) - 1;
        int between = (1 << (h - i + 1)) - 1;
        printSpaces(leading * 2);
        printLevel(root, i, between * 2);
        printf("\n\n"); }
}
int main()
{
    struct Node *root = NULL;
    int n, value;
    printf("Enter number of nodes: ");
```

```
scanf("%d", &n);
printf("Enter node values:\n");
for (int i = 0; i < n; i++) {
    scanf("%d", &value);
    root = insert(root, value); }
printTree(root);
return 0;
}
```

OUTPUT:

```
Enter number of nodes: 7
Enter node values:
50
30
70
20
40
60
80

Binary Search Tree:

      50
     /  \
    30   70
   /  \ /  \
  20  40 60 80
```

4.B. TRAVERSE THE ABOVE BINARY TREE RECURSIVELY IN PRE-ORDER, POST-ORDER & IN-ORDER**AIM:**

To write a C program to traverse the above binary tree recursively in pre-order, post-order & in-order

ALGORITHM:

1. Start the program and create an empty Binary Search Tree (BST).
2. Read the number of nodes and insert each node into the BST according to BST properties.
3. Perform Preorder traversal recursively by visiting the root, then the left subtree, and finally the right subtree.
4. Perform Inorder traversal recursively by visiting the left subtree, then the root, and finally the right subtree.
5. Perform Postorder traversal recursively by visiting the left subtree, then the right subtree, and finally the root.
6. Display the preorder, inorder, and postorder traversal results and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct Node
{
    int data;
    struct Node *left;
    struct Node *right;
};
```

```
struct Node *createNode(int value)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct Node *insert(struct Node *root, int value)
{
    if (root == NULL)
        return createNode(value);
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    return root;
}
void preorder(struct Node *root)
{
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right); }
}
void inorder(struct Node *root)
{
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right); }
}
void postorder(struct Node *root)
{
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data); }
}
int main()
{
    struct Node *root = NULL;
    int n, value;
```

```
printf("Enter number of nodes: ");
scanf("%d", &n);
printf("Enter node values:\n");
for (int i = 0; i < n; i++) {
    scanf("%d", &value);
    root = insert(root, value); }
printf("\nPreorder Traversal: ");
preorder(root);
printf("\nInorder Traversal: ");
inorder(root);
printf("\nPostorder Traversal: ");
postorder(root);
printf("\n");
return 0;
}
```

OUTPUT:

```
Enter number of nodes: 7
Enter node values:
50
30
70
20
40
60
80

Preorder Traversal: 50 30 20 40 70 60 80
Inorder Traversal: 20 30 40 50 60 70 80
Postorder Traversal: 20 40 30 60 80 70 50
```

RESULT:

Thus the C program to design and implement Binary Search Tree(BST) and traverse that tree recursively in pre-order, post-order & in-order was executed successfully.

Ex. NO: 5

IMPLEMENTATION OF ADVANCED TREE

DATE:

5.A. DESIGN AND IMPLEMENT AVL TREE USING TEMPLATES.**AIM:**

To write a C program to design and implement AVL tree using templates.

ALGORITHM:

1. Start and initialize the AVL tree root as NULL.
2. Read the elements from the user and insert each element using normal Binary Search Tree (BST) insertion.
3. Calculate the height and balance factor of each node after insertion.
Balance Factor = Height(Left Subtree) – Height(Right Subtree).
4. Balance the tree if the balance factor becomes less than -1 or greater than +1 using the appropriate rotation: LL, RR, LR, or RL.
5. Perform required operations such as deletion and update the height and balance factor of affected nodes, applying rotations when necessary.
6. Display the AVL tree using Inorder, Preorder, or Postorder traversal and stop the program.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
    int height;
};
int height(struct Node *node) {
    if (node == NULL)
        return 0;
    return node->height;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}
struct Node* createNode(int data) {
    struct Node *newNode =
        (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1;
    return newNode;
}
int getBalance(struct Node *node) {
```

```
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

struct Node* rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *temp = x->right;
    x->right = y;
    y->left = temp;
    y->height = 1 + max(height(y->left), height(y->right));
    x->height = 1 + max(height(x->left), height(x->right));
    return x;
}

struct Node* leftRotate(struct Node *x) {
    struct Node *y = x->right;
    struct Node *temp = y->left;
    y->left = x;
    x->right = temp;
    x->height = 1 + max(height(x->left), height(x->right));
    y->height = 1 + max(height(y->left), height(y->right));
    return y;
}

struct Node* insert(struct 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 {
        printf("Duplicate value not allowed!\n");
        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;
}

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

struct Node* deleteNode(struct Node *root, int data) {
    if (root == NULL) {
        printf("Value not found!\n");
        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) {
            struct Node *temp;
            if (root->left != NULL)
                temp = root->left;
            else
                temp = root->right;
            if (temp == NULL) {
                temp = root;
                root = NULL;
            }
        }
        else {
            *root = *temp;
        }
        free(temp);
    }
    else {
        struct 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(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

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

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

int main() {
    struct Node *root = NULL;
```

```
int choice, value, n, i;
while (1) {
    printf("\n\n==== AVL TREE MENU =====\n");
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Inorder Traversal\n");
    printf("4. Preorder Traversal\n");
    printf("5. Postorder Traversal\n");
    printf("6. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
    switch (choice) {
        case 1:
            printf("Enter value to insert: ");
            scanf("%d", &value);
            root = insert(root, value);
            printf("Value inserted successfully.\n");
            break;
        case 2:
            printf("Enter value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            break;
        case 3:
            printf("Inorder Traversal: ");
            inorder(root);
            printf("\n");
            break;
        case 4:
            printf("Preorder Traversal: ");
            preorder(root);
            printf("\n");
            break;
        case 5:
            printf("Postorder Traversal: ");
            postorder(root);
            printf("\n");
            break;
        case 6:
            printf("Program terminated.\n");
            exit(0);
        default:
            printf("Invalid choice!\n");
    }
}
```

```

    }
}
return 0;
}

```

OUTPUT:

```

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 10
Value inserted successfully.

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 20
Value inserted successfully.

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 30
Value inserted successfully.

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 40
Value inserted successfully.

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 1
Enter value to insert: 25
Value inserted successfully.

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 3
Inorder Traversal: 10 20 25 30 35 40

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 4
Preorder Traversal: 30 20 10 25 40 35

===== AVL TREE MENU =====
1. Insert
2. Delete
3. Inorder Traversal
4. Preorder Traversal
5. Postorder Traversal
6. Exit
Enter your choice: 5
Postorder Traversal: 10 25 20 35 40 30

```

5.B. DESIGN AND IMPLEMENT HEAP TREE USING TEMPLATES.**AIM:**

To write a C program to design and implement heap tree using templates.

ALGORITHM:

1. Start and initialize an empty heap.
2. Read the elements from the user and insert each element into the heap.
3. Place the new element at the end of the heap and compare it with its parent node.
4. Adjust the heap by swapping the element with its parent until the heap property is satisfied (heapify-up).
5. Delete the root element when required, replace it with the last element, and restore the heap property using heapify-down.
6. Display the heap and stop the program.

PROGRAM:

```

#include <stdio.h>
#define MAX 100

```

```
int heap[MAX];
int size = 0;
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
void insert(int value) {
    int i;
    size++;
    i = size;
    heap[i] = value;
    while (i > 1 && heap[i] > heap[i / 2]) {
        swap(&heap[i], &heap[i / 2]);
        i = i / 2;
    }
}
void deleteMax() {
    int i, largest, left, right;
    if (size == 0) {
        printf("Heap is empty!\n");
        return;
    }
    printf("Deleted element: %d\n", heap[1]);
    heap[1] = heap[size];
    size--;
    i = 1;
    while (1) {
        left = 2 * i;
        right = 2 * i + 1;
        largest = i;
        if (left <= size && heap[left] > heap[largest])
            largest = left;
        if (right <= size && heap[right] > heap[largest])
            largest = right;
        if (largest != i) {
            swap(&heap[i], &heap[largest]);
            i = largest;
        }
        else {
            break;
        }
    }
}
```

```
}
void display() {
    int i;
    if (size == 0) {
        printf("Heap is empty!\n");
        return;
    }
    printf("Heap elements: ");
    for (i = 1; i <= size; i++)
        printf("%d ", heap[i]);
    printf("\n");
}
int main() {
    int choice, value, n, i;
    while (1) {
        printf("\n===== MAX HEAP MENU =====\n");
        printf("1. Insert\n");
        printf("2. Delete Maximum\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value: ");
                scanf("%d", &value);
                insert(value);
                printf("Element inserted successfully.\n");
                break;
            case 2:
                deleteMax();
                break;
            case 3:
                display();
                break;
            case 4:
                printf("Program terminated.\n");
                return 0;
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0; }
```

OUTPUT:

```
===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 1
Enter value: 50
Element inserted successfully.

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 1
Enter value: 30
Element inserted successfully.

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 1
Enter value: 70
Element inserted successfully.

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 1
Enter value: 20
Element inserted successfully.

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 3
Heap elements: 70 30 50 20

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 2
Deleted element: 70

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 3
Heap elements: 50 30 20

===== MAX HEAP MENU =====
1. Insert
2. Delete Maximum
3. Display
4. Exit
Enter your choice: 4
Program terminated.
```

RESULT:

Thus the C program to design and implement AVL tree and Heap tree was executed successfully.

Ex. NO: 6 IMPLEMENTATION OF SHORTEST PATH ALGORITHMS DATE:

6.A. DESIGN AND IMPLEMENT DIJKSTRA'S ALGORITHM

AIM:

To write a C program to design and implement Dijkstra's algorithm.

ALGORITHM:

1. Start the program and read the number of vertices and the cost adjacency matrix.
2. Initialize the distance of the source vertex as 0 and all other distances according to the adjacency matrix; mark all vertices as unvisited.
3. Select the unvisited vertex having the minimum distance and mark it as visited.
4. Update the distance of each unvisited adjacent vertex if a shorter path is obtained through the selected vertex.
5. Repeat Steps 3 and 4 until all reachable vertices are visited.
6. Display the shortest distance from the source vertex to every other vertex and stop the program.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
#define INF 9999
int main()
{
    int n, cost[MAX][MAX];
    int distance[MAX], visited[MAX];
    int source, i, j, count, min, next;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter the cost adjacency matrix:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &cost[i][j]);
            if (cost[i][j] == 0 && i != j)
                cost[i][j] = INF; } }
    printf("Enter source vertex (1-%d): ", n);
    scanf("%d", &source);
    source--;
    for (i = 0; i < n; i++)
    {
        distance[i] = cost[source][i];
        visited[i] = 0; }
    distance[source] = 0;
    visited[source] = 1;
```

```
for (count = 1; count < n; count++)
{
    min = INF;
    next = -1;
    for (i = 0; i < n; i++)
    {
        if (!visited[i] && distance[i] < min)
        {
            min = distance[i];
            next = i;        }    }
    if (next == -1)
        break;
    visited[next] = 1;
    for (i = 0; i < n; i++)
    {
        if (!visited[i] && distance[next] + cost[next][i] < distance[i])
            distance[i] = distance[next] + cost[next][i];
    } }
printf("\nShortest distances from vertex %d:\n", source + 1);
for (i = 0; i < n; i++)
{
    if (distance[i] == INF)
        printf("Vertex %d : INF\n", i + 1);
    else
        printf("Vertex %d : %d\n", i + 1, distance[i]); }
return 0;
}
```

OUTPUT:

```
Enter number of vertices: 5
Enter the cost adjacency matrix:
0 10 0 30 100
10 0 50 0 0
0 50 0 20 10
30 0 20 0 60
100 0 10 60 0
Enter source vertex (1-5): 1

Shortest distances from vertex 1:
Vertex 1 : 0
Vertex 2 : 10
Vertex 3 : 50
Vertex 4 : 30
Vertex 5 : 60
```

6.B. DESIGN AND IMPLEMENT FLOYD WARSHALL ALGORITHM**AIM:**

To write a C program to design and implement Floyd Warshall algorithm.

ALGORITHM:

1. Start the program and read the number of vertices and the cost adjacency matrix.
2. Initialize the matrix such that each diagonal element is 0 and unreachable vertices are represented by infinity.
3. Select each vertex k as an intermediate vertex one by one.
4. Compare the existing distance between every pair of vertices i and j with the distance through vertex k.
5. Update the distance using $D[i][j] = \min(D[i][j], D[i][k] + D[k][j])$ whenever a shorter path is found.
6. Display the final matrix containing the shortest distances between all pairs of vertices and stop the program.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
#define INF 9999
int main()
{
    int n, graph[MAX][MAX];
    int i, j, k;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter the cost adjacency matrix:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &graph[i][j]);

            if (graph[i][j] == 0 && i != j)
                graph[i][j] = INF;        }    }
    for (k = 0; k < n; k++)
    {
        for (i = 0; i < n; i++)
        {
            for (j = 0; j < n; j++)
            {
                if (graph[i][k] != INF && graph[k][j] != INF && graph[i][k] + graph[k][j] <
graph[i][j])
                {
                    graph[i][j] = graph[i][k] + graph[k][j];        }        }        }    }
    printf("\nAll-Pairs Shortest Path Matrix:\n");
    for (i = 0; i < n; i++)
    {
```

```
    for (j = 0; j < n; j++)  
    {  
        if (graph[i][j] == INF)  
            printf("INF\t");  
        else  
            printf("%d\t", graph[i][j]);    }  
    printf("\n"); }  
return 0;  
}
```

OUTPUT:

```
Enter number of vertices: 4  
Enter the cost adjacency matrix:  
0 5 0 10  
0 0 3 0  
0 0 0 1  
0 0 0 0  
  
All-Pairs Shortest Path Matrix:  
0      5      8      9  
INF     0      3      4  
INF    INF     0      1  
INF    INF    INF     0
```

RESULT:

Thus the C program to design and implementation of Dijkstra's and Floyd Warshall's algorithms were executed successfully.

Ex. NO: 7 IMPLEMENTATION OF MINIMUM SPANNING TREE DATE:

7.A. DESIGN AND IMPLEMENT KRUSKAL'S ALGORITHM

AIM:

To write a C program to design and implement Kruskal's algorithm.

ALGORITHM:

1. Start the program and read the number of vertices and edges with their weights.
2. Sort all edges in ascending order of their weights.
3. Initialize each vertex as a separate set.
4. Select the smallest edge and check whether it forms a cycle.
5. Add the edge to the MST if it does not form a cycle, and repeat until n-1 edges are selected.
6. Display the edges of the MST and its minimum cost.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
int parent[MAX];
int find(int i)
{   while (parent[i] != i)
    {   i = parent[i];
        return i;
    }
}
void unionSet(int i, int j)
{
    int a = find(i);
    int b = find(j);
    parent[a] = b;
}
int main()
{
    int n, e, i, j;
    int u, v, w;
    int count = 0, total = 0;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter number of edges: ");
    scanf("%d", &e);
    int edge[e][3];
    printf("Enter edges (source destination weight):\n");
    for (i = 0; i < e; i++)
        scanf("%d %d %d", &edge[i][0], &edge[i][1], &edge[i][2]);

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

```

    parent[i] = i;
    for (i = 0; i < e - 1; i++) {
        for (j = i + 1; j < e; j++) {
            if (edge[i][2] > edge[j][2]) {
                int temp0 = edge[i][0];
                int temp1 = edge[i][1];
                int temp2 = edge[i][2];
                edge[i][0] = edge[j][0];
                edge[i][1] = edge[j][1];
                edge[i][2] = edge[j][2];
                edge[j][0] = temp0;
                edge[j][1] = temp1;
                edge[j][2] = temp2;
            }
        }
    }
    printf("\nEdges in Minimum Spanning Tree:\n");
    for (i = 0; i < e && count < n - 1; i++) {
        u = edge[i][0];
        v = edge[i][1];
        w = edge[i][2];
        if (find(u) != find(v))
        {
            unionSet(u, v);
            printf("%d -- %d = %d\n", u, v, w);
            total += w;
            count++;
        }
    }
    printf("Minimum Cost = %d\n", total);
    return 0;
}

```

OUTPUT:

```

Enter number of vertices: 5
Enter number of edges: 7
Enter edges (source destination weight):
0 1 2
0 3 6
1 2 3
1 3 8
1 4 5
2 4 7
3 4 9

Edges in Minimum Spanning Tree:
0 -- 1 = 2
1 -- 2 = 3
1 -- 4 = 5
0 -- 3 = 6
Minimum Cost = 16

```

7.B. DESIGN AND IMPLEMENT PRIM'S ALGORITHM**AIM:**

To write a C program to design and implement Prim's algorithm.

ALGORITHM:

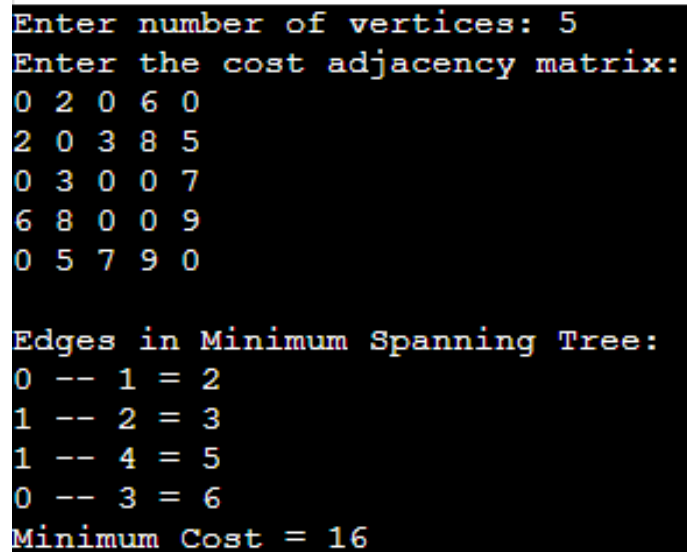
1. Start the program and read the number of vertices and cost adjacency matrix.
2. Select any vertex as the starting vertex and mark it as visited.
3. Find the minimum-cost edge connecting a visited vertex to an unvisited vertex.
4. Add the selected edge to the MST and mark the new vertex as visited.
5. Repeat steps 3 and 4 until n-1 edges are selected.
6. Display the edges of the MST and its minimum cost.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
#define INF 9999
int main()
{
    int n, cost[MAX][MAX];
    int visited[MAX] = {0};
    int i, j, edges = 0;
    int min, u = 0, v = 0, total = 0;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter the cost adjacency matrix:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            scanf("%d", &cost[i][j]);

            if (cost[i][j] == 0 && i != j)
                cost[i][j] = INF;        }    }
    visited[0] = 1;
    printf("\nEdges in Minimum Spanning Tree:\n");
    while (edges < n - 1)
    { min = INF;
      for (i = 0; i < n; i++)
      {
          if (visited[i])
          {
              for (j = 0; j < n; j++)
              {
                  if (!visited[j] && cost[i][j] < min)
```

```
        {
            min = cost[i][j];
            u = i;
            v = j;          }          }          }
    if (min == INF)
        break;
    printf("%d -- %d = %d\n", u, v, min);
    visited[v] = 1;
    total += min;
    edges++;
}
printf("Minimum Cost = %d\n", total);
return 0;}
```

OUTPUT:

```
Enter number of vertices: 5
Enter the cost adjacency matrix:
0 2 0 6 0
2 0 3 8 5
0 3 0 0 7
6 8 0 0 9
0 5 7 9 0

Edges in Minimum Spanning Tree:
0 -- 1 = 2
1 -- 2 = 3
1 -- 4 = 5
0 -- 3 = 6
Minimum Cost = 16
```

RESULT:

Thus the C program to design and implementation of Kruskal's and Prim's algorithms were executed successfully.

Ex. NO: 8

GRAPH TRAVERSAL & APPLICATIONS

DATE:

8.A. IMPLEMENTATION OF DEPTH FIRST SEARCH**AIM:**

To write a C program to implement Depth First Search (DFS) algorithm.

ALGORITHM:**PROGRAM:**

```
#include <stdio.h>
#define MAX 20
int graph[MAX][MAX], visited[MAX];
int n;
void DFS(int vertex)
{
    int i;
    visited[vertex] = 1;
    printf("%d ", vertex);
    for (i = 0; i < n; i++)
    {
        if (graph[vertex][i] == 1 && visited[i] == 0)
            DFS(i);
    }
}
int main()
{
    int i, j, start;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
    for (i = 0; i < n; i++)
        visited[i] = 0;
    printf("Enter starting vertex: ");
    scanf("%d", &start);
    printf("\nDFS Traversal: ");
    DFS(start);
    return 0;
}
```

OUTPUT:

```
Enter number of vertices: 5
Enter adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0
Enter starting vertex: 0
DFS Traversal: 0 1 3 4 2
```

8.B. IMPLEMENTATION OF BREADTH FIRST SEARCH**AIM:**

To write a C program to implement Breadth First Search (BFS) algorithm.

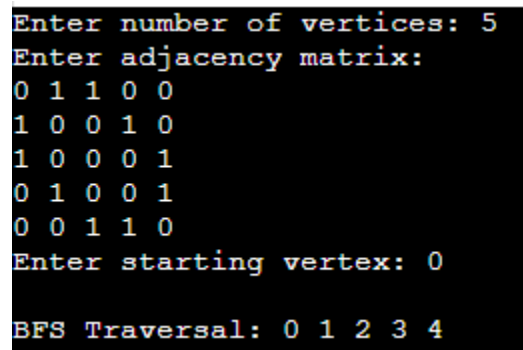
ALGORITHM:

1. Start the program and read the number of vertices and adjacency matrix.
2. Initialize all vertices as unvisited.
3. Select the starting vertex and mark it as visited.
4. Visit the selected vertex and recursively visit each unvisited adjacent vertex.
5. Continue until all reachable vertices are visited.
6. Display the DFS traversal and stop.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
int main()
{
    int graph[MAX][MAX], visited[MAX];
    int queue[MAX];
    int n, i, j, start;
    int front = 0, rear = -1;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &graph[i][j]);
    for (i = 0; i < n; i++)
        visited[i] = 0;
    printf("Enter starting vertex: ");
    scanf("%d", &start);
    queue[++rear] = start;
```

```
visited[start] = 1;
printf("\nBFS Traversal: ");
while (front <= rear)
{
    int vertex = queue[front++];
    printf("%d ", vertex);
    for (i = 0; i < n; i++)
    {
        if (graph[vertex][i] == 1 && visited[i] == 0)
        {
            queue[++rear] = i;
            visited[i] = 1;
        }
    }
}
return 0;
}
```

OUTPUT:

```
Enter number of vertices: 5
Enter adjacency matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 0 1
0 1 0 0 1
0 0 1 1 0
Enter starting vertex: 0
BFS Traversal: 0 1 2 3 4
```

8.C. IMPLEMENTATION OF TOPOLOGICAL SORTING**AIM:**

To write a C program to implement Breadth First Search (BFS) algorithm.

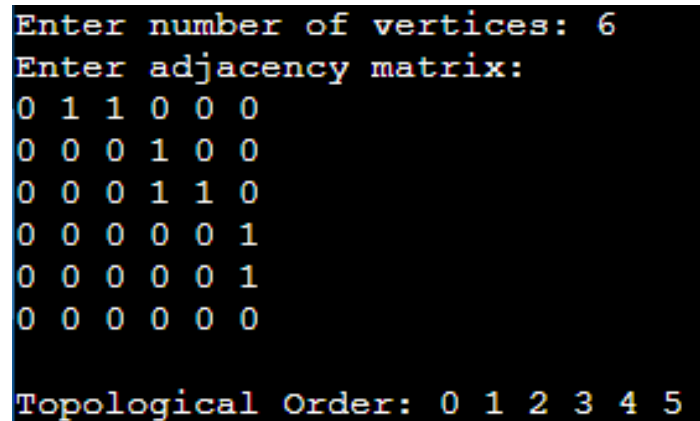
ALGORITHM:

1. Start the program and read the number of vertices and adjacency matrix.
2. Calculate the indegree of every vertex.
3. Insert all vertices having indegree zero into the queue.
4. Remove a vertex from the queue, add it to the topological order, and decrease the indegree of its adjacent vertices.
5. Insert newly zero-indegree vertices into the queue and repeat until the queue is empty.
6. Display the topological order; if all vertices are not processed, report that the graph contains a cycle.

PROGRAM:

```
#include <stdio.h>
#define MAX 20
int main()
{
    int graph[MAX][MAX], indegree[MAX];
    int queue[MAX], topo[MAX];
    int n, i, j;
    int front = 0, rear = -1;
    int count = 0;
    printf("Enter number of vertices: ");
    scanf("%d", &n);
    printf("Enter adjacency matrix:\n");
    for (i = 0; i < n; i++)
    { for (j = 0; j < n; j++)
        scanf("%d", &graph[i][j]);
    }
    for (i = 0; i < n; i++)
    {
        indegree[i] = 0;
        for (j = 0; j < n; j++)
        {
            if (graph[j][i] == 1)
                indegree[i]++;
        }
    }
    for (i = 0; i < n; i++)
    {
        if (indegree[i] == 0)
            queue[++rear] = i;
    }
    while (front <= rear)
    {
        int vertex = queue[front++];
        topo[count++] = vertex;
        for (i = 0; i < n; i++)
        {
            if (graph[vertex][i] == 1)
            {
                indegree[i]--;
                if (indegree[i] == 0)
                    queue[++rear] = i;
            }
        }
    }
    if (count != n)
        printf("\nTopological sorting is not possible.\nThe graph contains a cycle.\n");
}
```

```
    else
    {
        printf("\nTopological Order: ");
        for (i = 0; i < n; i++)
            printf("%d ", topo[i]);
        printf("\n");
    }
    return 0;
}
```

OUTPUT:

```
Enter number of vertices: 6
Enter adjacency matrix:
0 1 1 0 0 0
0 0 0 1 0 0
0 0 0 1 1 0
0 0 0 0 0 1
0 0 0 0 0 1
0 0 0 0 0 0

Topological Order: 0 1 2 3 4 5
```

RESULT:

Thus the C program to design and implementation of DFS, BFS and Topological sorting algorithms were executed successfully.

Ex. NO: 9

SORTING AND HASHTABLE IMPLEMENTATION

DATE:

9.A. IMPLEMENTATION OF SORTING TECHNIQUES**AIM:**

To write a C program to implementing the following sorting techniques to arrange a list of integers in ascending order. a) Insertion sort, b) Selection sort, c) Quick sort, d) Merge sort.

ALGORITHM:

1. Start the program and declare an array to store the given list of integers.
2. Read the number of elements n and read all the elements into the array.
3. Display the menu containing Insertion Sort, Selection Sort, Quick Sort, and Merge Sort.
4. Read the user's choice of sorting technique.
5. If Insertion Sort is selected, take each element as a key and insert it into its correct position in the sorted portion.
6. If Selection Sort is selected, find the smallest element from the unsorted portion and swap it with the first unsorted element.
7. If Quick Sort is selected, choose a pivot, partition the array around the pivot, and recursively sort the left and right subarrays.
8. If Merge Sort is selected, recursively divide the array into smaller subarrays and merge them in ascending order.
9. Display the resulting array after applying the selected sorting technique.

PROGRAM:

```
#include <stdio.h>
#define MAX 50
void insertionSort(int a[], int n)
{
    int i, j, key;
    for (i = 1; i < n; i++)
    {
        key = a[i];
        j = i - 1;
        while (j >= 0 && a[j] > key)
        {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}
void selectionSort(int a[], int n)
{
    int i, j, min, temp;
    for (i = 0; i < n - 1; i++)
    {
        min = i;
        for (j = i + 1; j < n; j++)
        {
            if (a[j] < a[min])
                min = j;
        }
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

```
        temp = a[i];
        a[i] = a[min];
        a[min] = temp;    }}
void quickSort(int a[], int low, int high)
{   int i, j, pivot, temp;
    if (low < high)
    {   pivot = a[low];
        i = low;
        j = high;
        while (i < j) {
            while (a[i] <= pivot && i < high)
                i++;
            while (a[j] > pivot)
                j--;
            if (i < j) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;    }    }
        temp = a[low];
        a[low] = a[j];
        a[j] = temp;
        quickSort(a, low, j - 1);
        quickSort(a, j + 1, high);    }    }
void merge(int a[], int low, int mid, int high) {
    int temp[MAX];
    int i = low;
    int j = mid + 1;
    int k = low;
    while (i <= mid && j <= high) {
        if (a[i] <= a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while (i <= mid)
        temp[k++] = a[i++];
    while (j <= high)
        temp[k++] = a[j++];
    for (i = low; i <= high; i++)
        a[i] = temp[i];
}
void mergeSort(int a[ ], int low, int high){
    int mid;
```

```
    if (low < high)
    {
        mid = (low + high) / 2;
        mergeSort(a, low, mid);
        mergeSort(a, mid + 1, high);
        merge(a, low, mid, high);    } }
void display(int a[ ], int n){
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
int main()
{
    int a[MAX], n, i, choice;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements:\n");
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);
    do{
        printf("\nSORTING TECHNIQUES\n");
        printf("1. Insertion Sort\n");
        printf("2. Selection Sort\n");
        printf("3. Quick Sort\n");
        printf("4. Merge Sort\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
                insertionSort(a, n);
                printf("\nUsing Insertion Sort:\n");
                printf("Sorted array in ascending order:\n");
                display(a, n);
                break;
            case 2:
                selectionSort(a, n);
                printf("\nUsing Selection Sort:\n");
                printf("Sorted array in ascending order:\n");
                display(a, n);
                break;
```

```
case 3:
    quickSort(a, 0, n - 1);
    printf("\nUsing Quick Sort:\n");
    printf("Sorted array in ascending order:\n");
    display(a, n);
    break;
case 4:
    mergeSort(a, 0, n - 1);
    printf("\nUsing Merge Sort:\n");
    printf("Sorted array in ascending order:\n");
    display(a, n);
    break;
case 5:
    break;
default:
    printf("Invalid choice\n");
    return 0;
}
}while(choice!=5);
return 0;
}
```

OUTPUT:

```
Enter number of elements: 6
Enter the elements:
20 42 13 10 50 8

SORTING TECHNIQUES
1. Insertion Sort
2. Selection Sort
3. Quick Sort
4. Merge Sort
Enter your choice: 1

Using Insertion Sort:
Sorted array in ascending order:
8 10 13 20 42 50
```

```
Enter your choice: 2
```

```
Using Selection Sort:
Sorted array in ascending order:
8 10 13 20 42 50
```

```
Enter your choice: 3
```

```
Using Quick Sort:
Sorted array in ascending order:
8 10 13 20 42 50
```

```
Enter your choice: 4
```

```
Using Merge Sort:
Sorted array in ascending order:
8 10 13 20 42 50
```

9.B. IMPLEMENTATION OF HASHING TECHNIQUES**AIM:**

To write a C program to implementing the following hashing techniques – a) Linear Probing, b) Quadratic Probing.

ALGORITHM:

1. Start the program and initialize both hash tables with empty values.
2. Read the number of keys and the keys to be inserted.
3. Calculate the hash index using key % SIZE.
4. For Linear Probing, check the next consecutive position when a collision occurs.
5. For Quadratic Probing, check positions using $(\text{index} + i^2) \% \text{SIZE}$ when a collision occurs.
6. Insert each key into the first available position in both hash tables.
7. Repeat the insertion process until all keys are inserted or the table becomes full.
8. Display the hash table obtained using Linear Probing.
9. Display the hash table obtained using Quadratic Probing.

PROGRAM:

```
#include <stdio.h>
#define SIZE 10
int linearHash[SIZE];
int quadraticHash[SIZE];
void initialize()
{
    int i;
    for (i = 0; i < SIZE; i++)
    {
        linearHash[i] = -1;
        quadraticHash[i] = -1;
    }
}
void linearProbing(int key)
{
    int index, i, pos;
    index = key % SIZE;
    for (i = 0; i < SIZE; i++)
    {
        pos = (index + i) % SIZE;
        if (linearHash[pos] == -1)
        {
            linearHash[pos] = key;
            return;
        }
    }
}
```

```
    printf("Linear Hash Table is full\n");
}
void quadraticProbing(int key)
{
    int index, i, pos;
    index = key % SIZE;
    for (i = 0; i < SIZE; i++)
    {
        pos = (index + i * i) % SIZE;
        if (quadraticHash[pos] == -1)
        {
            quadraticHash[pos] = key;
            return;
        }
    }
    printf("Quadratic Hash Table is full\n");
}
void display()
{
    int i;
    printf("\nLinear Probing:\n");
    for (i = 0; i < SIZE; i++)
    {
        if (linearHash[i] == -1)
            printf("Index %d : Empty\n", i);
        else
            printf("Index %d : %d\n", i, linearHash[i]);
    }
    printf("\nQuadratic Probing:\n");
    for (i = 0; i < SIZE; i++)
    {
        if (quadraticHash[i] == -1)
            printf("Index %d : Empty\n", i);
        else
            printf("Index %d : %d\n", i, quadraticHash[i]);
    }
}
int main()
{
    int n, key, i;
    initialize();
    printf("Enter number of keys: ");
    scanf("%d", &n);
    printf("Enter the keys:\n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &key);
```

```
    linearProbing(key);  
    quadraticProbing(key);  
}  
display();  
return 0;  
}
```

OUTPUT:

```
Enter number of keys: 6  
Enter the keys:  
25 35 15 45 55 65
```

Linear Probing:

```
Index 0 : 65  
Index 1 : Empty  
Index 2 : Empty  
Index 3 : Empty  
Index 4 : Empty  
Index 5 : 25  
Index 6 : 35  
Index 7 : 15  
Index 8 : 45  
Index 9 : 55
```

Quadratic Probing:

```
Index 0 : 65  
Index 1 : 55  
Index 2 : Empty  
Index 3 : Empty  
Index 4 : 45  
Index 5 : 25  
Index 6 : 35  
Index 7 : Empty  
Index 8 : Empty  
Index 9 : 15
```

RESULT:

Thus the c program to implementing the sorting and hashing techniques were executed successfully.