RAJALAKSHMI ENGINEERING COLLEGE

An Autonomous Institution, Affiliated to Anna University
Rajalakshmi Nagar, Thandalam – 602 105

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CS23231 – DATA STRUCTURES

(*Regulation 2023*)

LAB MANUAL

Name : . . . . . .

Register No. : . . . . . .

Year / Branch / Section : . . . . . .

Semester : . . . . . .

Academic Year : . . . . . .

LESSON PLAN

Course Code	Course Title (Laboratory Integrated Theory Course)	L	T	P	C
CS23231	Data Structures	1	0	6	4

LIST OF EXPERIMENTS	
Sl. No	**Name of the experiment**
Week 1	Implementation of Single Linked List (Insertion, Deletion and Display)
Week 2	Implementation of Doubly Linked List (Insertion, Deletion and Display)
Week 3	Applications of Singly Linked List (Polynomial Manipulation)
Week 4	Implementation of Stack using Array and Linked List implementation
Week 5	Applications of Stack (Infix to Postfix)
Week 6	Applications of Stack (Evaluating Arithmetic Expression)
Week 7	Implementation of Queue using Array and Linked List implementation
Week 8	Implementation of Binary Search Tree
Week 9	Performing Tree Traversal Techniques
Week 10	Implementation of AVL Tree
Week 11	Performing Topological Sorting
Week 12	Implementation of BFS, DFS
Week 13	Implementation of Prim's Algorithm
Week 14	Implementation of Dijkstra's Algorithm
Week 15	Program to perform Sorting
Week 16	Implementation of Open Addressing (Linear Probing and Quadratic Probing)
Week 17	Implementation of Rehashing

EX.NO.1 SINGLY LINKED LIST

AIM:

To implement singly linked list for performing various operations like insertion, deletion, search and display.

ALGORITHM:

1. Start.
2. Initialize a head or dummy node pointing towards NULL.
3. Display the operations and using switch case, select the desired operation to be executed.
4. Separate functions for each operations is created like
    - *Insertion in beginning.
    - *Insertion after p.
    - *Insertion in the end.
    - *Deletion in beginning
    - *Deletion after p.
    - *Deletion in the end.
    - *Find an element.
    - *Find the next element.
    - *Find previous element.
    - *Display.
    - *Delete list.
    - *Is empty and Is last.
5. The operation ends when user enters 0.
6. End.

PROGRAM:

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

void insert_begin(struct Node* L)
{
   int x;
   printf("Enter the value to be inserted: ");
   scanf("%d",&x);
   struct Node*n=(struct Node*) malloc(sizeof(struct Node));

   if(n!=NULL)
   {
      n->data=x;
      if(L->next!=NULL)
      {
         n->next=L->next;
         L->next=n;
      }
      else
      L->next=n;
   }
}

void insert_last(struct Node* L)
{
   int x;
```

```
    printf("Enter the value to be inserted: ");
    scanf("%d",&x);
    struct Node*n=(struct Node*) malloc(sizeof(struct Node));
    if(n!=NULL)
    {
        n->data=x;
        struct Node*temp = L->next;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=n;
    }
}

void insert_after(struct Node* L)
{
    int x,pos;
    printf("\nEnter the value to be inserted: ");
    scanf("%d",&x);
    printf("Enter the position to be inserted: ");
    scanf("%d",&pos);
    struct Node*n=(struct Node*) malloc(sizeof(struct Node));
    if(n!=NULL)
    {
        n->data=x;
        struct Node*temp = L->next;
        for(int i=0;i<pos;i++)
        {
            temp=temp->next;
        }
        n->next=temp->next;
        temp->next=n;
    }
}

void delete_begin(struct Node* L)
{
    struct Node* temp=L->next;
    L->next = L->next->next;
    free(temp);
}

void delete_last(struct Node* L)
{
    struct Node* temp=L->next;
    while(temp->next->next!=NULL)
    {
        temp=temp->next;
    }
    free(temp->next);
    temp->next=NULL;
}

void delete_after(struct Node* L)
{
    int pos;
    struct Node* temp=L->next;
    printf("\nEnter the position to be deleted: ");
    scanf("%d",&pos);
```

```
    for(int i=0;i<pos;i++)
    {
       temp=temp->next;
    }
    struct Node* p=temp->next;
    temp->next=temp->next->next;
    free(p);

}

struct Node* find(struct Node* L)
{
    int num;
    struct Node* temp=L->next;
    printf("\nEnter the number to be searched: ");
    scanf("%d",&num);

    while((temp->next!=NULL)&&(temp->data!=num))
    {
       temp=temp->next;
    }
    if(temp->data==num)
       return temp;
    else
       return NULL;
}

struct Node* find_next(struct Node* L)
{
    int num;
    struct Node* temp=L->next;
    printf("\nEnter the number to be searched: ");
    scanf("%d",&num);

    while((temp->next!=NULL)&&(temp->data!=num))
    {
       temp=temp->next;
    }
    if(temp)
    return temp->next;
    else
    return NULL;
}

struct Node*find_prev(struct Node* L)
{
    int num;
    struct Node* temp=L;
    printf("\nEnter the number to be searched: ");
    scanf("%d",&num);

    while((temp->next!=NULL)&&(temp->next->data!=num))
    {
       temp=temp->next;
    }
    if(temp!=L&&temp->next!=NULL)
       return temp;
    else
       return NULL;
}
```

```
void delete_all(struct Node* L)
{
   struct Node* p=L->next;
   L->next=NULL;
   struct Node* temp=p;
   while(p!=NULL)
   {
      temp=p->next;
      free(p);
      p=temp;
   }
}
void is_empty(struct Node* L)
{
   if(L->next==NULL)
   printf("The list is empty");
   else
   printf("The list is not empty");
}

void is_last(struct Node* L)
{
   int pos;
   printf("Enter the position to be inserted: ");
   scanf("%d",&pos);
   struct Node*temp = L->next;
   for(int i=0;i<pos;i++)
   {
      temp=temp->next;
   }
   if(temp->next==NULL)
   printf("This is the last index");
   else
   printf("This is not the last index");
}

void display(struct Node* L)
{
   struct Node* temp=L->next;
   while(temp!=NULL)
   {
      printf("%d -> ",temp->data);
      temp=temp->next;
   }
   printf("\n");
}


void main() {
   int choice=1;
   struct Node list,*ptr;
   list.next=NULL;

   printf("OPERATIONS:\n\n");
   printf(" 1.Insert at the first\n 2.Insert at the end\n 3.Insert after position p\n 4.Delete the first\n 5.Delete the
end\n 6.Delete at the position p\n 7.Find the element\n 8.Find next element\n 9.Find previous element\n
10.Display the list\n 11.Delete list\n 12.Check if list is empty\n 13.Check if the given index is the last\n\nSelect
'0' to exit\n");
   while(choice)
```

```
{
   printf("\nEnter choice: ");
   scanf("%d",&choice);

   switch(choice)
   {
      case 1:
      insert_begin(&list);
      break;

      case 2:
      insert_last(&list);
      break;

      case 3:
      insert_after(&list);
      break;

      case 4:
      delete_begin(&list);
      break;

      case 5:
      delete_last(&list);
      break;

      case 6:
      delete_after(&list);
      break;

      case 7:
      ptr=find(&list);
      if(ptr)
      printf("Element %d found at location %x\n",ptr->data,ptr);
      else
      printf("Element not found\n");
      break;

      case 8:
      ptr=find_next(&list);
      if(ptr)
      printf("Element %d found at location %x\n",ptr->data,ptr);
      else
      printf("Element not found\n");
      break;

      case 9:
      ptr=find_prev(&list);
      if(ptr)
      printf("Element %d found at location %x\n",ptr->data,ptr);
      else
      printf("Element not found\n");
      break;

      case 10:
      display(&list);
      break;

      case 11:
      delete_all(&list);
```

```
            break;

            case 12:
            is_empty(&list);
            break;

            case 13:
            is_last(&list);
            break;

            case 0:
            printf("\nOperation terminated...");
            break;

            default:
            printf("Invalid operation\n");

        }
    }
}
```

OUTPUT:

OPERATIONS:

1.Insert at the first
2.Insert at the end
3.Insert after position p
4.Delete the first
5.Delete the end
6.Delete at the position p
7.Find the element
8.Find next element
9.Find previous element
10.Display the list
11.Delete the entire list
12.Check if list is empty
13.Check the given index is the last

Select '0' to exit

Enter choice: 1
Enter the value to be inserted: 1

Enter choice: 2
Enter the value to be inserted: 2

Enter choice: 10
1 -> 2 ->

Enter choice: 0

Operation terminated...

RESULT:

Thus, the singly linked list is implemented and executed successfully for performing the desired operations…

EX.NO.2 IMPLEMENTATION OF DOUBLY LINKED LIST

AIM:

To perform the doubly linked list operation (insertion ,deletion ,searching, display)

ALGORITHM:

1. Start

2. Create a structure and functions for each operations

3. Declare the variables

4. Create a do-while loop to display the menu and execute operations based on your input until the user chooses to exit

5. Inside the loop display the menu options

6. Prompt the user to enter their choice

7. Use switch statement to perform different operations based on the user's choice and display it.

8. Repeat the loop until the user chooses to exit

9. Exit

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
// Define a node structure for doubly linked list
struct Node {
int data;
struct Node* prev;
struct Node* next;
};
// Function to create a new node
struct Node* createNode(int data)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->prev = NULL;
newNode->next = NULL;
return newNode;
}
// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head_ref, int data)
```

```
 {
 struct Node* newNode = createNode(data);
 if (*head_ref == NULL)
 {
 *head_ref = newNode;
 return;
}
newNode->next = *head_ref;
 (*head_ref)->prev = newNode;
 *head_ref = newNode;
 }
 // Function to insert a node at the end of the list
 void insertAtEnd(struct Node** head_ref, int data)
{
struct Node* newNode = createNode(data);
 struct Node* temp = *head_ref;
 if (*head_ref == NULL)
 {
 *head_ref = newNode;
 return;
}
 while (temp->next != NULL)
{
 temp = temp->next;
}
 temp->next = newNode;
 newNode->prev = temp;
 }
 // Function to insert a node at a specific position
 void insertAtPosition(struct Node** head_ref, int data, int position)
 {
 if (position < 1)
 {
 printf("Invalid position\n");
```

```
 return;
 }
 if (position == 1)
 {
insertAtBeginning(head_ref, data);
 return;
 }
 struct Node* newNode = createNode(data);
 struct Node* temp = *head_ref;
 for (int i = 1; i < position - 1 && temp != NULL; i++)
{
 temp = temp->next;
}
 if (temp == NULL)
 {
 printf("Position out of range\n");
 return;
 }
 newNode->next = temp->next;
 if (temp->next != NULL)
{
 temp->next->prev = newNode;
 }
 temp->next = newNode;
 newNode->prev = temp;
 }
 // Function to delete the first node
void deleteFirstNode(struct Node** head_ref)
{
if (*head_ref == NULL)
 {
 printf("List is empty\n");
 return;
}
```

```
struct Node* temp = *head_ref;
 *head_ref = temp->next;
 if (*head_ref != NULL)
{
(*head_ref)->prev = NULL;
 }
 free(temp);
 }
 // Function to delete the last node
 void deleteLastNode(struct Node** head_ref)
 {
 if (*head_ref == NULL)
 {
 printf("List is empty\n");
 return;
}
 struct Node* temp = *head_ref;
 while (temp->next != NULL)
 { temp = temp->next;
 }
 if (temp->prev != NULL)
{
 temp->prev->next = NULL;
 }
 else
{
 *head_ref = NULL;
 }
 free(temp);
 }
 // Function to delete a node at a specific position
void deleteAtPosition(struct Node** head_ref, int position)
 {
 if (*head_ref == NULL)
```

```
{
 printf("List is empty\n");
 return;
}
 if (position < 1)
 { printf("Invalid position\n");
return;
 }
 if (position == 1)
 {
deleteFirstNode(head_ref);
 return;
 }
 struct Node* temp = *head_ref;
 for (int i = 1; i < position && temp != NULL; i++)
{
temp = temp->next;
 }
 if (temp == NULL)
{
 printf("Position out of range\n");
 return;
 }
 if (temp->next != NULL)
{
 temp->next->prev = temp->prev;
 }
 temp->prev->next = temp->next;
free(temp);
 }
 // Function to search for a node with a given value
struct Node* searchNode(struct Node* head, int key)
 {
 struct Node* temp = head;
```

```
while (temp != NULL)
{
 if (temp->data == key)
 {
 return temp;
}
 temp = temp->next;
}
 return NULL;
 }
 // Function to display the doubly linked list
void displayList(struct Node* head)
{
 struct Node* temp = head;
 while (temp != NULL)
 {
 printf("%d ", temp->data);
 temp = temp->next;
 }
 printf("\n");
 }
 int main()
{
 struct Node* head = NULL;
int choice, data, position, key;
do
 {
 printf("\n1. Insert at Beginning\n");
 printf("2. Insert at End\n");
 printf("3. Insert at Position\n");
 printf("4. Delete First Node\n");
printf("5. Delete Last Node\n");
 printf("6. Delete at Position\n");
 printf("7. Search for a Node\n");
```

```
 printf("8. Display List\n");
 printf("9. Exit\n");
 printf("Enter your choice: ");
 scanf("%d", &choice);
 switch(choice)
{
case 1:
            printf("Enter data to insert at beginning: ");
            scanf("%d", &data);
           insertAtBeginning(&head, data); break;
case 2:
            printf("Enter data to insert at end: ");
            scanf("%d", &data);
            insertAtEnd(&head, data);
           break;
case 3:
            printf("Enter data to insert: ");
            scanf("%d", &data);
            printf("Enter position to insert at: ");
           scanf("%d", &position);
            insertAtPosition(&head, data, position);
           break;
case 4:
           deleteFirstNode(&head);
            break;
case 5:
            deleteLastNode(&head);
           break;
 case 6:
            printf("Enter position to delete: ");
            scanf("%d", &position);
            deleteAtPosition(&head, position);
            break;
case 7:
```

```
            printf("Enter value to search: ");
            scanf("%d", &key);
            struct Node* result = searchNode(head, key);
            if (result != NULL)
            {
            printf("%d found in the list.\n", key);
            }
            else
            {
            printf("%d not found in the list.\n", key);
            }
            break;
case 8:
            printf("Doubly linked list: ");
            displayList(head);
            break;
case 9:
            printf("Exiting...\n");
            break;
default:
 printf("Invalid choice\n");
 }
 } while (choice != 9);
return 0;
 }
```

OUTPUT:

1. Insert at Beginning
2. Insert at End
3. Insert at Position
4. Delete First Node
5. Delete Last Node
6. Delete at Position
7. Search for a Node
8. Display List

9. Exit

Enter your choice: 2

Enter data to insert at end: 1

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 2

Enter data to insert at end: 2

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 2

Enter data to insert at end: 3

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 8

Doubly linked list: 1 2 3

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 6

Enter position to delete: 3

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 8

Doubly linked list: 1 2

1. Insert at Beginning

2. Insert at End

3. Insert at Position

4. Delete First Node

5. Delete Last Node

6. Delete at Position

7. Search for a Node

8. Display List

9. Exit

Enter your choice: 9

Exiting...

=== Code Execution Successful ===

RESULT:

Thus, the implementation of doubly linked list is executed successfully...

EX.NO.3 POLYNOMIAL MANUPILATION

AIM:

To perform arithmetic operations on polynomials entered by the user.

ALGORITHM:

1. Start

2. Loop (repeat until user stops)

3. Get Polynomials

4. Ask user for the first polynomial (coefficients and powers).

5. Store terms in a list (highest power first), removing duplicates (coefficients of 0).

6. Repeat for the second polynomial (storing in a separate list).

7. Add the polynomials (corresponding terms), store the sum in another list, removing duplicates.

8. Subtract the polynomials (corresponding terms), store the difference in another list, removing duplicates.

9. Multiply the polynomials (each term from one with each term from the other), combine terms by power, store the product in another list, removing duplicates.

10. Show the original polynomials entered by the user.

11. Show the sum, difference, and product of the polynomials.

12. Ask the user if they want to continue (0 to exit).

13. Free the memory used by all the polynomial lists.

14. Stop.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


struct Node{

   int coeff;

   int power;

   struct Node* next;

};
```

```
void insert(struct Node*L,int coeff,int power)
{
   struct Node* n = (struct Node*) malloc(sizeof(struct Node));
   if (n!=NULL)
   {
      struct Node*p=L->next;
      n->coeff=coeff;
      n->power=power;
      if(p==NULL) L->next=n;
      else if(p->next==NULL && p->power<n->power)
      {
         n->next=p;
         L->next=n;
      }
      else
      {
         if(p->power<n->power)
         {
            n->next=p;
            L->next=n;
            return;
         }
           while(p->next!=NULL && p->next->power>n->power)
         {
            p=p->next;
         }
         n->next=p->next;
         p->next=n;
      }
```

```
    }

    else
    printf("Memory not allocated");
}
void remove_duplicates(struct Node*L)
{
    struct Node*p=L->next;
    while(p!=NULL && p->coeff==0)
    {
        L->next=p->next;
        free(p);
        p=L->next;
    }

    while(p!=NULL && p->next!=NULL)
    {
        if(p->next!=NULL && p->next->coeff==0)
        {
            struct Node*temp=p->next;
            p->next=p->next->next;
            free(temp);
        }

        else if(p->power==p->next->power)
        {
            p->coeff += p->next->coeff;
            struct Node*temp=p->next;
            p->next=p->next->next;
            free(temp);
```

```
        }

        else
        p=p->next;
    }
}

struct Node add(struct Node*L1,struct Node*L2)
{
    struct Node *p1=L1->next;
    struct Node *p2=L2->next;
    struct Node L3;
    L3.next=NULL;
    while(p1!=NULL && p2!=NULL)
    {
        if(p1->power > p2->power)
        {
            insert(&L3,p1->coeff,p1->power);
            p1=p1->next;
        }

        else if(p2->power > p1->power)
        {
            insert(&L3,p2->coeff,p2->power);
            p2=p2->next;
        }
        else
        {
            insert(&L3,p1->coeff + p2->coeff,p1->power);
            p1=p1->next;
```

```
            p2=p2->next;
        }
    }
    while (p1!=NULL)
    {
        insert(&L3,p1->coeff,p1->power);
        p1=p1->next;
    }
    while (p2!=NULL)
    {
        insert(&L3,p2->coeff,p2->power);
        p2=p2->next;
    }
    return L3;
}

struct Node sub(struct Node*L1,struct Node*L2)
{
    struct Node *p1=L1->next;
    struct Node *p2=L2->next;
    struct Node L3;
    L3.next=NULL;
    while(p1!=NULL && p2!=NULL)
    {
        if(p1->power > p2->power)
        {
            insert(&L3,p1->coeff,p1->power);
            p1=p1->next;
        }
```

```
        else if(p2->power > p1->power)
        {
            insert(&L3,-p2->coeff,p2->power);
            p2=p2->next;
        }
         else
        {
            p1=p1->next;
            p2=p2->next;
        }
    }
    while (p1!=NULL)
    {
        insert(&L3,p1->coeff,p1->power);
        p1=p1->next;
    }
    while (p2!=NULL)
    {
        insert(&L3,-p2->coeff,p2->power);
        p2=p2->next;
    }
    return L3;
}

struct Node multiply(struct Node*L1,struct Node*L2)
{
    struct Node*p1=L1->next;
    struct Node*p2;
    struct Node L3;
    L3.next=NULL;
```

```
    while(p1!=NULL)
    {
       p2=L2->next;
       while(p2!=NULL)
       {
          insert(&L3,p1->coeff * p2->coeff,p1->power+p2->power);
           p2=p2->next;
       }
       p1=p1->next;
    }
    return L3;
}
void get_nodes(int num,struct Node*L)
{
    int term,c,p;
    printf("\nEnter number of terms in polynomial %d:  ",num);
    scanf("%d",&term);
    printf("Enter the first polynomial terms in descending order(coeff x power):\n");
    while(term)
    {
       printf("=> ");
       scanf("%dx%d",&c,&p);
       insert(L,c,p);
       Term--;
    }
    remove_duplicates(L);
}

void delete_all(struct Node* L)
{
```

```
    struct Node* p=L->next;
    L->next=NULL;
    struct Node* temp=p;
    while(p!=NULL)
    {
        temp=p->next;
        free(p);
        p=temp;
    }
}

void display(struct Node*L)
{
    struct Node* ptr=L->next;

    while(ptr!=NULL)
    {
        if((ptr->power==0))
        {
            printf("%d ",ptr->coeff,ptr->power);
            ptr=ptr->next;
        }
        else if((ptr->power==1))
        {
            printf("%dx ",ptr->coeff,ptr->power);
            ptr=ptr->next;
        }
        else if(ptr->power!=0)
        {
            printf("%dx^%d ",ptr->coeff,ptr->power);
```

```
            ptr=ptr->next;
        }
        if(ptr!=NULL)       {
            printf("+ ");
        }
    }
    if(L->next==NULL)
    printf("0");
}
int main() {
    struct Node list1,list2,list3,list4,list5;
    list1.next=NULL;
    list2.next=NULL;
    list3.next=NULL;
    list4.next=NULL;
    list5.next=NULL;

    int end=1;
    while (end)
    {
        get_nodes(1,&list1);
        get_nodes(2,&list2);

        list3=add(&list1,&list2);
        remove_duplicates(&list3);

        list4=sub(&list1,&list2);
        remove_duplicates(&list4);

        list5=multiply(&list1,&list2);
```

```
        remove_duplicates(&list5);

        printf("\nFirst polynomial equation: p(x)\n");
        display(&list1);

        printf("\nSecond polynomial equation: q(x)\n");
        display(&list2);

        printf("\n\nSum of polynomial equations: p(x)+q(x)\n");
        display(&list3);

        printf("\n\nDifference of polynomial equations: p(x)-q(x)\n");
        display(&list4);

        printf("\n\nProduct of polynomial equations: p(x)*q(x)\n");
        display(&list5);

        printf("\n\nDo you wish to continue: (press 0 to exit, 1 to continue) ");
        scanf("%d",&end);

        if(end==0) printf("Operation terminated…");

        delete_all(&list1);
        delete_all(&list2);
        delete_all(&list3);
        delete_all(&list4);
        delete_all(&list5);
    }
    return 0;
}
```

OUTPUT:

Enter number of terms in polynomial 1: 2

Enter the first polynomial terms in descending order(coeff x power):

=> 3x^2

=> -1x

Enter number of terms in polynomial 2: 1

Enter the first polynomial terms in descending order(coeff x power):

⇨ 2x

First polynomial equation: p(x)

3x^2 – x

Second polynomial equation: q(x)

2x

Sum of polynomial equations: p(x)+q(x)

3x^2 + x

Difference of polynomial equations: p(x)-q(x)

3x^2 – 3x

Product of polynomial equations: p(x)*q(x)

6x^3 – 2x^2

Do you wish to continue: (press 0 to exit, 1 to continue):0

RESULT:

Thus, the above program runs successfully…

EX.NO.4 IMPLEMENTATION OF STACK USING ARRAY

AIM:

To implement a stack using Array.

ALGORITHM:

Step 1: Start.

Step 2: Define the maximum size of the Stack (MAX) and initialize the top index (top) to – 1

Step 3: Check if the Stack is full or empty (is Full () and Is Empty).

Add elements to the stack (Push ()).

Remove elements from the stack (Pop ()).

Retrieve the top elements without removing it (Top)).

Display all elements of the stack. (Display)).

Step 4: Continuously prompt the user for actions.

Step 5: Stop.

PROGRAM:

```
#include <stdio.h>
#define MAX 5
int Stack[MAX], top = -1;
int IsFull();
int IsEmpty();
void Push(int ele);
void Pop();
void Top();
void Display();
int main()
{
    int ch, e;
    do
    {
        printf("1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT");
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1:
```

```
                printf("Enter the element : ");
                scanf("%d", &e);
                Push(e);
                break;
            case 2:
                Pop();
                break;
            case 3:
                Top();
                break;
            case 4:
                Display();
                break;
        }
    } while(ch <= 4);
    return 0;
}
int IsFull()
{
    if(top == MAX - 1)
        return 1;
    else
        return 0;
}
int IsEmpty()
{
    if(top == -1)
        return 1;
    else
        return 0;
}
void Push(int ele)
{
    if(IsFull())
```

```
            printf("Stack Overflow...!\n");
        else
        {
            top = top + 1;
            Stack[top] = ele;
        }
}
void Pop()
{
        if(IsEmpty())
            printf("Stack Underflow...!\n");
        else
        {
            printf("%d\n", Stack[top]);
            top = top - 1;
        }
}
void Top()
{
         if(IsEmpty())
             printf("Stack Underflow...!\n");
         else
            printf("%d\n", Stack[top]);
}
void Display()
{
        int i;
        if(IsEmpty())
            printf("Stack Underflow...!\n");
        else
        {
            for(i = top; i >= 0; i--)
                printf("%d\t", Stack[i]);
            printf("\n");
```

```
        }
}
```

OUTPUT:

```
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element: 10
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT.
Enter your choice:1
Enter the element:20
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:30
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element: 40
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:50
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:60
Stack over flow…!
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
```

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:2

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:2

Stack under flow…!

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:5

Result:

Thus, the above program for Array Implementation for Stack is executed successfully & output is verified…

EX.NO.4 IMPLEMENTATION OF STACK USING LINKED LIST

AIM:

To implement a stack using Linked list.

ALGORITHM:

Step 1: Start.

Step 2: Define a structure for a stack node containing an integer element and a pointer to the next node.

Step 3: Functionality implementations.

Is Empty (), Push (int e), Pop (), Top (), Display ()

Step 4: Continuously prompt the user for actions: Push, Pop, Top, Display or Exit.

Step 5: Stop.

PROGRAM:

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int Element;
    struct node *Next;
}*List = NULL;
typedef struct node Stack;
int IsEmpty();
void Push(int e);
void Pop();
void Top();
void Display();
int main()
{
    int ch, e;
    do
    {
        printf("1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT");
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch(ch)
```

```
            {
                case 1:
                        printf("Enter the element : ");
                        scanf("%d", &e);
                        Push(e);
                        break;
                case 2:
                        Pop();
                        break;
                case 3:
                        Top();
                        break;
                case 4:
                        Display();
                        break;
             }
        } while(ch <= 4);
        return 0;
}
int IsEmpty()
{
      if(List == NULL)
            return 1;
      else
           return 0;
}
void Push(int e)
{
        Stack *NewNode = malloc(sizeof(Stack));
        NewNode->Element = e;
        if(IsEmpty())
              NewNode->Next = NULL;
        else
              NewNode->Next = List;
```

```
        List = NewNode;
}
void Pop()
{
        if(IsEmpty())
                printf("Stack is Underflow...!\n");
        else
        {
                Stack *TempNode;
                TempNode = List;
                List = List->Next;
                printf("%d\n", TempNode->Element);
                free(TempNode);
        }
}
void Top()
{
        if(IsEmpty())
                printf("Stack is Underflow...!\n");
        else
                printf("%d\n", List->Element);
}
void Display()
{
        if(IsEmpty())
                printf("Stack is Underflow...!\n");
        else
        {
                Stack *Position;
                Position = List;
                while(Position != NULL)
                {
                        printf("%d\t", Position->Element);
                        Position = Position->Next;
```

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

OUTPUT:

```
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element: 10
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT.
Enter your choice:1
Enter the element:20
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:30
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element: 40
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:50
1.PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:1
Enter the element:60
Stack over flow…!
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT
Enter your choice:2
```

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:2

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:2

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:2

Stack under flow…!

1,PUSH 2.POP 3.TOP 4.DISPLAY 5.EXIT

Enter your choice:5

Result:

Thus, the above program for linked list implementation of Stack is executed successfully and the output is verified…

EX.NO.5 INFIX TO POSTFIX CONVERSION

AIM:

To write a C program to perform infix to postfix conversion using stack.

ALGORITHM:

Step 1: Start.

Step 2: Make an empty stack.

Step 3: Read the infix expression one character at a time until it encounters end of expression.

Step 4: If the character is an operand, place it onto the output.

Step 5: If the character is an operator, push it onto the stack.

Step 6: If the stack operator has a higher or equal priority than input operator, then pop that operator from the stack and place it onto the output.

Step 7: If the character is a left parenthesis, push it onto the stack.

Step 8: If the character is a right parenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Step 9: After the infix expression has been read, pop all operators from the stack and append them to the output string.

Step 10: The output string is the postfix expression.

Step 11: Stop.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#define MAX 20

int Stack[MAX], top = -1;
char expr[MAX], post[MAX];
void Push(char sym);
char Pop();
char Top();
int Priority(char sym);

int main()
{
        int i;
        printf("Enter the infix expression : ");
        fgets(expr,MAX,stdin);
```

```
	int size=strlen(expr)-1;
	if(expr[size]=='\n')
	{
		expr[size]='\0';
	}

	for(i = 0; i < strlen(expr); i++)
	{
		if(expr[i] >= 'a' && expr[i] <= 'z')
			printf("%c", expr[i]);
		else if(expr[i] == '(')
			Push(expr[i]);
		else if(expr[i] == ')')
		{
			while(Top() != '(')
				printf("%c", Pop());
			Pop();
		}
		else
		{
			while(Priority(expr[i])<=Priority(Top()) && top!=-1)
				printf("%c", Pop());
			Push(expr[i]);
		 }
	}
	for(i = top; i >= 0; i--)
		printf("%c", Pop());
	return 0;
}

void Push(char sym)
{
	top = top + 1;
	Stack[top] = sym;
}
```

```
char Pop()
{
	char e;
	e = Stack[top];
	top = top - 1;
	return e;
}
char Top()
{
	return Stack[top];;
}

int Priority(char sym)
{
	int p = 0;
	switch(sym)
	{
		case '(':
			p = 0;
			break;
		case '+':
		case '-':
			p = 1;
			break;
		case '*':
		case '/':
		case '%':
			p = 2;
			break;
		case '^':
			p = 3;
			break;
	}
 return p;
}
```

OUTPUT:

Enter the infix expression : a/b^c+d*e-f*g

abc^/de*+fg*-

RESULT:

Thus, the C program to perform infix to postfix conversion in stack was executed successfully…

EX.NO.6 EVALUATING ARITHMETIC EXPRESSION

AIM:

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

ALGORITHM:

Step 1: Start.

Step 2: Make an empty stack.

Step 3: Read the postfix expression one character at a time until it encounters end of expression.

Step 4: If the character is an operand, push its associated value onto the stack.

Step 5: If the character is an operator, pop two values from the stack, apply the operator to them and push the result onto the stack.

Step 6: After the entire postfix expression has been read, the stack will contain one element which is the result of the expression.

Step 7: Pop the result to the output screen.

Step 8: Stop.

PROGRAM:

```
#include <stdio.h>
#include <string.h>
#define MAX 20

int Stack[MAX], top = -1;
char expr[MAX];
void Push(int ele);
int Pop();

int main()
{
        int i, a, b, c, e;
        printf("Enter the postfix expression : ");
        fgets(expr,MAX,stdin);
        int size=strlen(expr)-1;
        if(expr[size]=='\n')
        {
                expr[size]='\0';
        }
        for(i = 0; i < strlen(expr); i++)
        {
```

```
            if(expr[i]=='+'||expr[i]=='-'||expr[i]=='*'||expr[i]=='/')
            {
                b = Pop();
                a = Pop();
                switch(expr[i])
                {
                    case '+':
                        c = a + b;
                        Push(c);
                        break;
                    case '-':
                        c = a - b;
                        Push(c);
                        break;
                    case '*':
                        c = a * b;
                        Push(c);
                        break;
                    case '/':
                        c = a / b;
                        Push(c);
                        break;
                }
            }
            else
            {
                printf("Enter the value of %c : ", expr[i]);
                scanf("%d", &e);
                Push(e);
            }
        }
 printf("The result is %d", Pop());
 return 0;
}
void Push(int ele)
{
```

```
        top = top + 1;
        Stack[top] = ele;
}
int Pop()
{
        int e;
        e = Stack[top];
        top = top - 1;
        return e;
}
```

OUTPUT:

```
Enter the postfix expression : abc+*d*
Enter the value of a : 2
Enter the value of b : 3
Enter the value of c : 4
Enter the value of d : 5
The result is 70
```

RESULT:

Thus, the C program to evaluate an arithmetic expression using stack was executed successfully…

EX.NO.7 IMPLEMENTATION OF QUEUE USING ARRAY

AIM:

Array implementation of queue using c program.

ALGORITHM:

Step 1: Create an empty array to represent the queue.

Step 2: Add an element to the rear of the array.

Step 3: Remove an element from the front of the array.

Step 4: Retrieve the front element without removing it.

Step 5: Verify if the array is empty.

PROGRAM:

```
#include <stdio.h>

 #define MAX 5

int Queue[MAX], front = -1, rear = -1;
int IsFull();

int IsEmpty();

void Enqueue(int ele);

void Dequeue();

void Display();
int main()

{

int ch, e;

do

 {
printf("1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT");

printf("\nEnter your choice : ");

scanf("%d", &ch);

switch(ch)

 {

case 1:

printf("Enter the element : ");

scanf("%d", &e);
```

```
 Enqueue(e);
break;
case 2:
 Dequeue();
break;
case 3:
 Display();
break;
 }
 } while(ch <= 3);
return 0;
}
int IsFull()
{
if(rear == MAX - 1)
return 1;
else
return 0;
int IsEmpty()
{
if(front == -1)
return 1;
else
return 0;
}
void Enqueue(int ele)
{
if(IsFull())
printf("Queue is Overflow...!\n");
else
 {
```

```
 rear = rear + 1;
 Queue[rear] = ele;
if(front == -1)
 front = 0;
 }
}
void Dequeue()
{
if(IsEmpty())
printf("Queue is Underflow...!\n");
else
 {
printf("%d\n", Queue[front]);
if(front == rear)
 front = rear = -1;
else
 front = front + 1;
 }
}
void Display()
{
int i;
if(IsEmpty())
printf("Queue is Underflow...!\n");
else
 {
for(i = front; i <= rear; i++)
printf("%d\t", Queue[i]);
printf("\n");

 }
```

}

OUTPUT:

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 10

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 20

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 30

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 40

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 50

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 1

Enter the element : 60

Queue is Overflow...!

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 3

10 20 30 40 50

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

10

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

20

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

30

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

40

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

50

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

Queue is Underflow...!

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 3

Queue Underflow...!

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 4

RESULT:

Hence the program executed successfully…

EX.NO.7 IMPLEMENTATION OF QUEUE USING LINKED LIST

AIM:

Linked list implementation of queue using c program.

ALGORITHM:

Step 1: Create a new linked list node with the given value.

Step 2: If the rear value is None, set the front and rear to the newly created node. Otherwise, set the next of the rear to the newly created node and move the rear pointer to that newly created node.

Step 3: If the front is None, return (base case).Initialize a temporary pointer (temp) with the front node. Set the front to its next node.

Step 4: If the front becomes None, set the rear to None.Delete temp from memory.

PROGRAM:

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

struct node

{

int Element;

struct node *Next;

}*Front = NULL, *Rear = NULL;

typedef struct node Queue;

int IsEmpty(Queue *List);

void Enqueue(int e);

void Dequeue();

void Display();

int main()

{

int ch, e;

do

 {

printf("1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT");

printf("\nEnter your choice : ");

scanf("%d", &ch);
```

```
switch(ch)
 {
case 1:
printf("Enter the element : ");
scanf("%d", &e);
 Enqueue(e);
break;
case 2:
 Dequeue();
break;
case 3:
 Display();
 break;
 }
 } while(ch <= 3);
return 0;
}
int IsEmpty(Queue *List)
{
if(List == NULL)
return 1;
else
return 0;
}
void Enqueue(int e)
{
Queue *NewNode = malloc(sizeof(Queue));
 NewNode->Element = e;
 NewNode->Next = NULL;
if(Rear == NULL)
 Front = Rear = NewNode;
else
 {
```

```
 Rear->Next = NewNode;
 Rear = NewNode;
 }
}
void Dequeue()
{
if(IsEmpty(Front))
printf("Queue is Underflow...!\n");
else
 {
Queue *TempNode;
 TempNode = Front;
if(Front == Rear)
 Front = Rear = NULL;
else
 Front = Front->Next;
printf("%d\n", TempNode->Element);
free(TempNode);
 }
}
void Display()
{
if(IsEmpty(Front))
printf("Queue is Underflow...!\n");
else
 {
Queue *Position;
 Position = Front;
while(Position != NULL)
 {
```

```
printf("%d\t", Position->Element);
 Position = Position->Next;
 }
printf("\n");
 }
}
```

Output :

```
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 1
Enter the element : 10
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 1
Enter the element : 20
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 1
Enter the element : 30
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 1
Enter the element : 40
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 1
Enter the element : 50
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 3
10 20 30 40 50
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 2
10
1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT
Enter your choice : 2
```

20

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

30

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

40

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

50

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 2

Queue is Underflow...!

1.ENQUEUE 2.DEQUEUE 3.DISPLAY 4.EXIT

Enter your choice : 4

RESULT:

Hence the program executed successfully...

EX.NO.8 TREE TRAVERSAL

AIM:

The aim of this program is to implement and demonstrate tree traversal techniques (In-order, Pre-order, and Post-order) on a binary tree using C programming language

ALGORITHM:

1. Start.

2. Defines a structure Node.

3. Create a new node with the given value and initialize its pointers.

4. To insert a new node into the binary search tree while maintaining the BST property. Create a function to check if the value is less than the current node's data, it traverses to the left subtree; otherwise, it traverses to the right subtree.

5. Provide a function to perform an inorder traversal of the binary search tree, printing the nodes in sorted order.

6. Provide a function to perform an preorder traversal of the binary search tree, printing the nodes in sorted order.

7. Provide a function to perform an postorder traversal of the binary search tree, printing the nodes in sorted order.

8. End.

PROGRAM:

```
#include <stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *left;
   struct node *right;
};
struct node* newnode(int x)
{
   struct node *new=(struct node *)malloc(sizeof(struct node));
   new->data=x;
   new->left=NULL;
   new->right=NULL;
   return new;
}
```

```
struct node* insert(struct node *root,int x)
{
   if(root==NULL)
    root=newnode(x);
   else if (root->data>x)
    root->left=insert(root->left,x);
   else
    root->right=insert(root->right,x);
   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;
root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
printf("\nInorder : ");
   inorder(root);
    printf("\nPreorder : ");
    preorder(root);
    printf("\nPostorder : ");
    postorder(root);
return 0;
}
```

OUTPUT:

Inorder : 20 30 40 50 60 70 80

Preorder : 50 30 20 40 70 60 80

Postorder : 20 40 30 60 80 70 50

RESULT:

The output is verified successfully for the above program…

EX.NO.9 BINARY SEARCH TREE

AIM:

Implementation of Binary Search Tree using C

ALGORITHM:

1.Create an empty tree and insert the root node .

2.Insert the value to the right , if the root node is smaller than the value and in the left , if the root node is greater than the value.

3.Find the value in the right , if the root node is smaller than the value and in the left , if the root node is greater than the value.

4.To delete an element from a BST, the user first searches for the element using the search operation. If the element is found, there are three cases to consider:

- The node to be deleted has no children: In this case, simply remove the node from the tree.
- The node to be deleted has only one child: In this case, remove the child node from its original position and replace the node to be deleted with its child node.
- The node to be deleted has two children: In this case, get the in-order successor of that node, replace the node with the inorder successor, and then remove the inorder successor from its original position.

5.To display the binary search tree , we go with inorder,preorder and postorder traversal.

PROGRAM:

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

struct Node
{
   int data;
   struct Node* left;
   struct Node* right;
};

void insert(struct Node*root,int data)
{
   if(data<root->data)
   {
      if(root->left!=NULL)
      insert(root->left,data);
      else
      {
         struct Node*new_node=(struct Node*)malloc(sizeof(struct Node));
         new_node->data=data;
         new_node->left=NULL;
         new_node->right=NULL;
         root->left=new_node;
      }
   }

   else
   {
      if(root->right!=NULL)
```

```
        insert(root->right,data);
        else
        {
            struct Node*new_node=(struct Node*)malloc(sizeof(struct Node));
            new_node->data=data;
            new_node->left=NULL;
            new_node->right=NULL;
            root->right=new_node;
        }
    }
}

struct Node* find(struct Node*root,int data)
{
    struct Node*p;
    if(root->data==data)
        printf("%d Element found \n",data);
    else if(data<root->data)
    {
        if(root->left!=NULL)
            p=find(root->left,data);
        else
        {
            printf("Element not found\n");
        }
    }

    else
    {
        if(root->right!=NULL)
            p=find(root->right,data);
        else
        {
            printf("Element not found\n");
        }
    }
    return p;
}

struct Node* delete(struct Node* root, int data)
{
    if (root == NULL)
        return root;

    if (root->data > data)
    {
        root->left = delete(root->left, data);
    }
```

```
    else if (root->data < data)
    {
        root->right = delete(root->right, data);
    }

    else if (root->left == NULL)
    {
        struct Node* temp = root->right;
        free(root);
        return temp;
    }
    else if (root->right == NULL)
    {
        struct Node* temp = root->left;
        free(root);
        return temp;
    }
    else {

        struct Node* succParent = root;
        struct Node* succ = root->right;
        while (succ->left != NULL)
        {
            succParent = succ;
            succ = succ->left;
        }
        if (succParent != root)
            succParent->left = succ->right;
        else
            succParent->right = succ->right;
        root->data = succ->data;
        free(succ);
    }
    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);
    }
}
void main()
{
    int choice=1;
    struct Node root;
    root.left=NULL;
    root.right=NULL;
    printf("Enter the data for root node: ");
    scanf("%d",&root.data);
    while(choice!=5)
    {
        printf("\nChoose :-");
        printf("\n1.Insert");
        printf("\n2.Delete");
        printf("\n3.Find");
        printf("\n4.Display");
        printf("\n5.Exit");
        printf("\nEnter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                int x;
                printf("\nEnter element : ");
                scanf("%d",&x);
                insert(&root,x);
                break;
            case 2:
                int del;
                printf("\nEnter element to be deleted : ");
                scanf("%d",&del);
                delete(&root,del);
                break;
```

```
            case 3:
                int e;
                printf("\nEnter element to be searched : ");
                scanf("%d",&e);
                find(&root,e);
                break;
            case 4 :
                printf("\nInorder : ");
                inorder(&root);
                printf("\nPreorder : ");
                preorder(&root);
                printf("\nPostorder : ");
                postorder(&root);
                break;
        }
    }
}
```

OUTPUT:

```
Enter the data for root node: 30

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 1
Enter element : 20

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 1
Enter element : 10

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 1
Enter element : 60
```

```
Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 1
Enter element : 90

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 4

Inorder : 10 20 30 60 90
Preorder : 30 20 10 60 90
Postorder : 10 20 90 60 30

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 3

Enter element to be searched : 20
20 Element found

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 2
Enter element to be deleted : 60

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 4
```

Inorder : 10 20 30 90
Preorder : 30 20 10 90
Postorder : 10 20 90 30

Choose :-
1.Insert
2.Delete
3.Find
4.Display
5.Exit
Enter your choice : 5

RESULT:

Hence, the above code is executed and the output obtained…

EX.NO.10 AVL TREE

AIM:

Implementation of AVL Trees

ALGORITHM:

Implementing AVL trees involves several steps, including rotations and maintaining balance factors.

1. **Node Structure**: Define a structure for the AVL tree node. It typically includes:
    - Key: The value stored in the node.
    - Height: The height of the node in the tree.
    - Left and Right Pointers: Pointers to the left and right children.
2. **Rotation Functions**: Implement rotation functions for balancing the tree:
    - Left Rotation
    - Right Rotation
    - Left-Right Rotation (Double Rotation)
    - Right-Left Rotation (Double Rotation)
3. **Insertion**:
    - Perform a standard BST insertion.
    - Update heights of ancestors.
    - Balance the tree using rotation functions if necessary.
4. **Deletion**:
    - Perform a standard BST deletion.
    - Update heights of ancestors.
    - Balance the tree using rotation functions if necessary.
5. **Balancing**:
    - Maintain balance factor for each node (the difference in heights of the left and right subtrees).
    - Balance the tree by performing rotations when the balance factor of a node is greater than 1 or less than -1.
6. **Height Calculation**:
    - Update the height of a node after each insertion and deletion.

PROGRAM:

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

struct AVLNode {
   int key;
   struct AVLNode *left;
   struct AVLNode *right;
   int height;
};

int height(struct AVLNode *node) {
   if (node == NULL)
      return 0;
   return node->height;
}
```

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

struct AVLNode *newNode(int key) {
    struct AVLNode *node = (struct AVLNode *)malloc(sizeof(struct AVLNode));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

struct AVLNode *rightRotate(struct AVLNode *y) {
    struct AVLNode *x = y->left;
    struct AVLNode *T2 = x->right;


    x->right = y;
    y->left = T2;


    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;


    return x;
}


struct AVLNode *leftRotate(struct AVLNode *x) {
    struct AVLNode *y = x->right;
    struct AVLNode *T2 = y->left;


    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;


    return y;
}
```

```
int getBalance(struct AVLNode *node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}


struct AVLNode *insertNode(struct AVLNode *node, int key) {

    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insertNode(node->left, key);
    else if (key > node->key)
        node->right = insertNode(node->right, key);
    else
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);


    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

struct AVLNode *minValueNode(struct AVLNode *node) {
```

```
    struct AVLNode *current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}

struct AVLNode *deleteNode(struct AVLNode *root, int key) {
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);

    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {
        if ((root->left == NULL) || (root->right == NULL)) {
            struct AVLNode *temp = root->left ? root->left : root->right;


            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;

            free(temp);
        } else {

            struct AVLNode *temp = minValueNode(root->right);


            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }

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

int main() {
    struct AVLNode *root = NULL;
    int choice, value;

    while (1) {
        printf("\n1. Insert a value\n");
        printf("2. Delete a value\n");
        printf("3. Print AVL tree in inorder\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value to insert: ");
```

```
            scanf("%d", &value);
            root = insertNode(root, value);
            break;
        case 2:
            printf("Enter the value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            break;
        case 3:
            printf("Inorder traversal of the AVL tree:\n");
            inorder(root);
            printf("\n");
            break;
        case 4:
            exit(0);
        default:
            printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

OUTPUT:

```
1. Insert a value
2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 1
Enter the value to insert: 7

1. Insert a value
2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 1
Enter the value to insert: 9

1. Insert a value
2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 1
Enter the value to insert: 2

1. Insert a value
```

2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 1
Enter the value to insert: 8

1. Insert a value
2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 1
Enter the value to insert: 1

1. Insert a value
2. Delete a value
3. Print AVL tree in inorder
4. Exit
Enter your choice: 3
Inorder traversal of the AVL tree:
1 2 7 8 9

RESULT:

Hence the code is implemented and executed successfully...

EX.NO.11 BREADTH FIRST SEARCH-BFS

AIM:

The aim of the program is to implement Breadth First Search using C programming Language.

ALGORITHM:

1. Start.
2. Create an empty queue Q.
3. Mark all vertices as unvisited.
4. Mark S as visited.
5. Enqueue s into Q.
6. Dequeue the front vertex from Q.
7. Traverse the graph.
8. Mark v as visited.
9. Enqueue v into Q.
10. End.

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>

struct queue

{

   int size;

   int f;

   int r;

   int* arr;

};

int isEmpty(struct queue *q){

   if(q->r==q->f){

      return 1;

   }

   return 0;

}
```

```
int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        // printf("Enqued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
```

```
    }
    return a;
}

int main(){
    // Initializing Queue (Array Implementation)
    struct queue q;
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
    int i = 1;
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };
    printf("%d", i);
    visited[i] = 1;
```

```
    enqueue(&q, i); // Enqueue i for exploration
    while (!isEmpty(&q))
    {
        int node = dequeue(&q);
        for (int j = 0; j < 7; j++)
        {
            if(a[node][j] ==1 && visited[j] == 0){
                printf("%d", j);
                visited[j] = 1;
                enqueue(&q, j);
            }
        }
    }
    return 0;
}
```

OUTPUT:

1 0 2 3 4 5 6

RESULT:

Hence, the program has been successfully implemented…

EX.NO.11 DEPTH FIRST SEARCH-DFS

AIM:

The aim of the program is to implement Depth First Search using C programming Language.

ALGORITHM:

1. Start.
2. Create a stack and push the starting vertex.
3. Mark the starting vertex as visited.
4. Pop a vertex from the stack.
5. If the neighbour has not been visited, mark it as visited
6. Push the neighbour onto the stack.
7. End.

PROGRAM:

```
#include<stdio.h>

#include<stdlib.h>



int visited[7] = {0,0,0,0,0,0,0};

   int A [7][7] = {

      {0,1,1,1,0,0,0},

      {1,0,1,0,0,0,0},

      {1,1,0,1,1,0,0},

      {1,0,1,0,1,0,0},

      {0,0,1,1,0,1,1},

      {0,0,0,0,1,0,0},

      {0,0,0,0,1,0,0}

   };



void DFS(int i){

   printf("%d ", i);

   visited[i] = 1;

   for (int j = 0; j < 7; j++)
```

```
    {
        if(A[i][j]==1 && !visited[j]){
            DFS(j);
        }
    }
}

int main(){
    DFS(0);
    return 0;
}
```

OUTPUT:

0 1 2 3 4 5 6

RESULT:

Hence, the program has been successfully implemented…

EX.NO.12 TOPOLOGICAL SORTING

AIM:

Implementation of Topological sorting using c.

ALGORITHM:

1. **Compute In-degree**: Calculate the in-degree (number of incoming edges) for each vertex.
2. **Initialize Queue**: Enqueue all vertices with in-degree 0 into a queue.
3. **Process Queue**: Process each vertex in the queue by:

   - Removing it from the queue.
   - Outputting it (or storing it in the result list).
   - Decreasing the in-degree of its neighboring vertices.
   - If a neighboring vertex's in-degree becomes 0, enqueue it.

4. **Check for Cycle**: If all vertices are processed and the result list has fewer vertices than the total number of vertices, then a cycle exists in the graph (not a DAG).

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>


#define MAX_VERTICES 100

struct AdjListNode {

   int dest;

   struct AdjListNode* next;

};


struct AdjList {

   struct AdjListNode* head;

};


struct Graph {

   int numVertices;

   struct AdjList* array;

   int* inDegree;

};

struct AdjListNode* newAdjListNode(int dest) {

   struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct AdjListNode));

   newNode->dest = dest;

   newNode->next = NULL;

   return newNode;
```

```
}
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;

    graph->array = (struct AdjList*)malloc(numVertices * sizeof(struct AdjList));
    graph->inDegree = (int*)malloc(numVertices * sizeof(int));

    for (int i = 0; i < numVertices; ++i) {
        graph->array[i].head = NULL;
        graph->inDegree[i] = 0;
    }

    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    graph->inDegree[dest]++;
}

void topologicalSort(struct Graph* graph) {
    int* result = (int*)malloc(graph->numVertices * sizeof(int)); // To store the topological order
    int resultIndex = 0; // Index in result array

    int* queue = (int*)malloc(graph->numVertices * sizeof(int));
    int front = 0, rear = 0;

    for (int i = 0; i < graph->numVertices; ++i) {
        if (graph->inDegree[i] == 0) {
            queue[rear++] = i;
        }
```

```
    }

    while (front < rear) {
        int u = queue[front++]; // Dequeue a vertex from queue and add it to result

        result[resultIndex++] = u;
        struct AdjListNode* node = graph->array[u].head;
        while (node != NULL) {
            int v = node->dest;
            if (--graph->inDegree[v] == 0) {
                queue[rear++] = v;
            }

            node = node->next;
        }
    }
    if (resultIndex != graph->numVertices) {
        printf("Graph has a cycle!\n");
        return;
    }
    printf("Topological Sort: ");
    for (int i = 0; i < resultIndex; ++i) {
        printf("%d ", result[i]);
    }
    printf("\n");
    free(result);
    free(queue);
}

int main() {
    int numVertices = 6;
    struct Graph* graph = createGraph(numVertices);

    addEdge(graph, 5, 2);
```

```
    addEdge(graph, 5, 0);
    addEdge(graph, 4, 0);
    addEdge(graph, 4, 1);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 1);

    printf("Given graph:\n");
    for (int v = 0; v < numVertices; v++) {
        struct AdjListNode* temp = graph->array[v].head;
        while (temp != NULL) {
            printf("(%d -> %d) ", v, temp->dest);
            temp = temp->next;
        }
        printf("\n");
    }

    topologicalSort(graph);

    return 0;
}
```

OUTPUT:

```
Given graph:
(2 -> 3)
(3 -> 1)
(4 -> 1) (4 -> 0)
(5 -> 0) (5 -> 2)
Topological Sort: 4 5 0 2 3 1
```

RESULT:

Hence, the above code is executed and the output obtained…

EX.NO.13 PRIM'S ALGORITHM

AIM:

To find minimum spanning tree using Prims

ALGORITHM:

1. Start
2. Start with an arbitrary vertex as the initial tree. This vertex can be chosen randomly or by any specific criteria.
3. Mark this vertex as visited and add it to the MST.
4. Find the cheapest edge from any visited vertex to any unvisited vertex.
5. Add the unvisited vertex connected by this edge to the MST, and mark it as visited.
6. Repeat steps 3 and 4 until all vertices are visited.
7. The MST is formed by all the edges selected during the process.

PROGRAM:

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

typedef struct Sub_table_parts* Sub_Table;
typedef struct Table_parts* Table;
typedef struct Node Node;
typedef struct Node* pos;
typedef struct Node* Vert;
typedef struct Graph_parts* Graph;
typedef int Vertex;

struct Node
{
   int data;
   int weight;
   pos next;
};

struct Graph_parts
{
   int vertices;
   int edges;
   Vert* List;
};

struct Sub_table_parts
{
   Vert Adj_list;
   int known;
   int distance;
   int prev_node;
};
```

```
struct Table_parts
{
   int size;
   Sub_Table* S_table;
};

void Link(Graph G, int v1, int v2, int c)
{
   Vert n = (Vert)malloc(sizeof(Node));
   if(n!=NULL)
   {
       n->data = v2;
       n->weight=c;
       pos p=G->List[v1];
       n->next=p->next;
       p->next=n;
   }
}

void Get_edges(Graph G)
{
   int v1,v2,cost;
   printf("Enter the edges with cost:\n");
   for(int i=0;i<G->edges;i++)
   {
      printf("=> ");
      scanf("%d %d %d", &v1, &v2, &cost);
      Link(G,v1,v2,cost);
      Link(G,v2,v1,cost);
   }
}

Graph Create_graph()
{
   Graph G = (Graph)malloc(sizeof(struct Graph_parts));
   if(G!=NULL)
   {
      printf("Enter the number of vertices in graph: ");
      scanf("%d", &G->vertices);
      printf("Vertices are from 0 to %d\n",G->vertices-1);

      printf("\nEnter the number of edges: ");
      scanf("%d",&G->edges);
```

```
        G->List = (Vert*)malloc(sizeof(Node) * G->vertices);
        if(G->List!=NULL)
        {
            for (int i = 0; i < G->vertices; i++)
            {
                G->List[i] = (Vert)malloc(sizeof(struct Node));
            }
        }
        Get_edges(G);
        return G;
    }
}

Table Create_table(Graph G)
{
    Table T = (Table)malloc(sizeof(struct Table_parts));

    if(T!=NULL)
    {
        T->size=G->vertices;
        T->S_table=(Sub_Table*)malloc(sizeof(struct Sub_table_parts)*T->size);
        for(int i=0;i<T->size;i++)
        {
            T->S_table[i]=(Sub_Table)malloc(sizeof(struct Table_parts));
            T->S_table[i]->Adj_list=G->List[i];
            T->S_table[i]->known=0;
            T->S_table[i]->distance=1000;
            T->S_table[i]->prev_node=-1;
        }
    }
    return T;
}

void Display(Graph G)
{
    printf("\nDisplaying the Graph using List\n\n");
    for (int i = 0; i < G->vertices; i++)
    {
        struct Node* p = G->List[i];
        printf("%d => ", i);
        p=p->next;
        while (p != NULL)
        {
            printf("(%d,%d) ", p->data,p->weight);
            p = p->next;
```

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

void Display_Table(Table T)
{
    printf("\nDisplaying the Table using List\n\n");
    for (int i = 0; i < T->size; i++)
    {
        Sub_Table p = T->S_table[i];

        printf("%d | k=%d | d=%d | p=%d | List = ",i,p->known,p->distance,p->prev_node);
        pos s=p->Adj_list;
        while (s->next!= NULL)
        {
            s=s->next;
            printf("(%d,%d) ", s->data,s->weight);
        }
        printf("\n");
    }
}

Vertex find_min_distance(Table T)
{
    int min=1000;
    Vertex v=-1;
    for(int i=0;i<T->size;i++)
    {
        if(T->S_table[i]->distance < min && T->S_table[i]->known == 0)
        {
            min=T->S_table[i]->distance;
            v=i;
        }
    }
    return v;
}

void Min_Spanning_Tree(Table T, Vertex s)
{
    Sub_Table cur = T->S_table[s];
    if(cur->known==0)
    {
        pos ptr=cur->Adj_list;
        while(ptr->next!=NULL)
```

```
        {
            ptr=ptr->next;
            int node=ptr->data;
            int dist=ptr->weight;
            Sub_Table next=T->S_table[node];
            if(next->distance >= dist && next->known ==0)
            {
                next->distance=dist;
                next->prev_node=s;
            }
        }
        cur->known=1;
    }
    Vertex min=find_min_distance(T);
    if(min!=-1)
        Min_Spanning_Tree(T,min);
}

void main()
{
    int s;
    Graph G=Create_graph();
    Table T=Create_table(G);
    Display(G);

    printf("\nEnter the starting node for Prim's Algorithm: ");
    scanf("%d",&s);

    T->S_table[s]->distance=0;
    Min_Spanning_Tree(T,s);
    Display_Table(T);

    printf("\nMinimum spanning tree edges for the given graph\n\n");
    for(int i=0;i<T->size;i++)
    {
        int k=T->S_table[i]->prev_node;
        int m=T->S_table[i]->distance;
        if(k!=-1)
            printf("(%d,%d,c=%d) ",k,i,m);
    }
}
```

OUTPUT:

```
Enter the number of vertices in graph: 4
Vertices are from 0 to 3
```

Enter the number of edges: 3
Enter the edges with cost:
=> 1 2 3
=> 0 3 7
=> 0 2 6

Displaying the Graph using List

0 => (2,6) (3,7)
1 => (2,3)
2 => (0,6) (1,3)
3 => (0,7)

Enter the starting node for Prim's Algorithm: 0

Displaying the Table using List

0 | k=1 | d=0 | p=-1 | List = (2,6) (3,7)
1 | k=1 | d=3 | p=2 | List = (2,3)
2 | k=1 | d=6 | p=0 | List = (0,6) (1,3)
3 | k=1 | d=7 | p=0 | List = (0,7)

Minimum spanning tree edges for the given graph

(2,1, c=3) (0,2, c=6) (0,3, c=7)

RESULT:

Hence the code is implemented and executed successfully...

EX.NO.14 DIJIKSTRA'S ALGORITHM

AIM:

The main aim of Dijkstra's algorithm is to find the shortest path from a starting node to all other nodes in a weighted graph. It's often used in scenarios like finding the shortest route in a road network or the lowest cost in a communication network.

ALGORITHM:

1. Start

2. Initialize the distance from the start node to all other nodes as infinity, except for the start node itself, which is set to zero. Maintain a set of unvisited nodes.

3. Choose the unvisited node with the smallest tentative distance from the start node. Initially, this will be the start node itself.

4. For the selected node, consider all of its unvisited neighbours and calculate their tentative distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

5. Once all of the neighbours of the selected node have been considered, mark the selected node as visited and remove it from the unvisited set.

6. Repeat steps 2 and 3 until all the nodes have been visited. The algorithm terminates when all nodes have been visited.

7. Once the algorithm has visited all the nodes, the shortest distance from the start node to every other node in the graph will have been calculated.

8. End

PROGRAM:

```
#include <stdio.h>

#include <limits.h>

#define MAX_VERTICES 100

int minDistance(int dist[], int sptSet[], int vertices) {

   int min = INT_MAX, minIndex;

   for (int v = 0; v < vertices; v++) {

      if (!sptSet[v] && dist[v] < min) {

         min = dist[v];

         minIndex = v;

      }

   }
```

```
    return minIndex;
}
void printSolution(int dist[], int vertices) {
    printf("Vertex \tDistance from Source\n");
    for (int i = 0; i < vertices; i++) {
        printf("%d \t%d\n", i, dist[i]);
    }
}
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int src, int vertices) {
    int dist[MAX_VERTICES];
    int sptSet[MAX_VERTICES];
    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        sptSet[i] = 0;
    }
    dist[src] = 0;
    for (int count = 0; count < vertices - 1; count++) {
        int u = minDistance(dist, sptSet, vertices);
        sptSet[u] = 1;
        for (int v = 0; v < vertices; v++) {
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }
                printSolution(dist, vertices);
}
int main() {
    int vertices;
    printf("Input the number of vertices: ");
```

```
    scanf("%d", &vertices);
    if (vertices <= 0 || vertices > MAX_VERTICES) {
        printf("Invalid number of vertices. Exiting...\n");
        return 1;
    }
    int graph[MAX_VERTICES][MAX_VERTICES];
    printf("Input the adjacency matrix for the graph (use INT_MAX for infinity):\n");
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
        }
    }
    int source;
    printf("Input the source vertex: ");
    scanf("%d", &source);
    if (source < 0 || source >= vertices) {
        printf("Invalid source vertex. Exiting...\n");
        return 1;
    }
    dijkstra(graph, source, vertices);
    return 0;
}
```

OUTPUT:

```
Input the number of vertices: 5
Input the adjacency matrix for the graph (use INT_MAX for infinity):
0 3 2 0 0
3 0 0 1 0
2 0 0 1 4
```

0 1 1 0 2

0 0 4 2 0

Input the source vertex: 0

Vertex Distance from Source

0 0

1 3

2 2

3 3

4 5

RESULT:

The output is verified successfully for the above program…

EX.NO.15 MERGE SORTING

AIM:

To understand the working and implement merge sort in C.

ALGORITHM:

1. Base Case: If the array has 0 or 1 element, it is already sorted.

2. Divide: Split the array into two halves.

3. Recursion: Recursively apply the same logic to both halves.

4. Merge: Merge the two sorted halves to produce the sorted array:

- Compare the elements of the two halves one by one and build a new sorted array.
- These algorithms both have their own advantages and typical use cases.

PROGRAM:

```
#include <stdio.h>

void printArray(int *A, int n)

{

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

   {

      printf("%d ", A[i]);

   }

   printf("\n");

}


void merge(int A[], int mid, int low, int high)

{

   int i, j, k, B[100];

   i = low;

   j = mid + 1;

   k = low;


   while (i <= mid && j <= high)

   {

      if (A[i] < A[j])

      {

         B[k] = A[i];

         i++;
```

```
            k++;
        }
        else
        {
            B[k] = A[j];
            j++;
            k++;
        }
    }
    while (i <= mid)
    {
        B[k] = A[i];
        k++;
        i++;
    }
    while (j <= high)
    {
        B[k] = A[j];
        k++;
        j++;
    }
    for (int i = low; i <= high; i++)
    {
        A[i] = B[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, mid, low, high);
    }
}

int main()
{
    // int A[] = {9, 14, 4, 8, 7, 5, 6};
    int A[] = {9, 1, 4, 14, 4, 15, 6};
    int n = 7;
    printf("MERGE SORT=>\nUnsorted Array:\n");
    printArray(A, n);
    mergeSort(A, 0, 6);
    printf("Sorted array in ascending order:\n");
    printArray(A, n);
    return 0;
}
```

OUTPUT:

MERGE SORT=>

Unsorted Array:

9 1 4 14 4 15 6

Sorted array in ascending order:

1 4 4 6 9 14 1

RESULT:

The output is verified successfully for the above program…

EX.NO.15 QUICK SORTING

AIM:

To understand the working and implement quick sort in C.

ALGORITHM:

1. Base Case: If the array has 0 or 1 element, it is already sorted.

2. Pivot Selection: Select the pivot element (commonly the middle element).

3. Partitioning: Partition the array into three parts:

- Elements less than the pivot.
- Elements equal to the pivot.
- Elements greater than the pivot.

4. Recursion: Recursively apply the same logic to the left and right sub-arrays.

5. Concatenation: Combine the sorted sub-arrays and the pivot to get the final sorted array.

PROGRAM:

```
#include <stdio.h>

// function to swap elements
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}
// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {
```

```
      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

// function to print array elements
void printArray(int array[], int size) {
```

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

// main function
int main() {
  int data[] = {8, 7, 2, 1, 0, 9, 6};

  int n = sizeof(data) / sizeof(data[0]);

  printf("QUICK SORT=>\nUnsorted Array:\n");
  printArray(data, n);

  // perform quicksort on data
  quickSort(data, 0, n - 1);

  printf("Sorted array in ascending order:\n");
  printArray(data, n);
}
```

OUTPUT:

```
QUICK SORT=>
Unsorted Array:
8 7 2 1 0 9 6
Sorted array in ascending order:
0 1 2 6 7 8 9
```

RESULT:

The output is verified successfully for the above program…

EX.NO.16 HASHING

AIM:

To create a hash table and perform collision resolution using various techniques.

ALGORITHM:

Step 1: Start
Step 2: Define cell and hash_table structures, and implement primary and secondary hash functions (hash and hash2).
Step 3: Implement prime and prime2 functions to find the next and previous prime numbers respectively for hash table sizing.
Step 4: Allocate memory for the hash table and its cells, initializing cell statuses to empty.
Step 5: Compute the position using the hash function and resolve collisions using linear probing.
Step 6: Compute the position using the hash function and resolve collisions using quadratic probing.
Step 7: Compute the position using the hash function and secondary hash function for step size to resolve collisions.
Step 8: Insert elements into the hash table at the computed position based on the selected probing method.
Step 9: Traverse the hash table and print the status and elements of each cell.
Step 10: Create a new hash table of double the size (next prime number) and reinsert existing elements.
Step 11: Accept user input for table size, elements, and operations, perform insertions, display results, and handle rehashing if selected.
Step 12: Stop

PROGRAM:

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

enum kind_of_entry{legitimate,empty,delete};
struct cell
{
   int element;
   enum kind_of_entry info;
};

struct hash_table
{
   int table_size;
    struct cell *the_cells;
};

int hash(int key,int table_size)
{
   return key%table_size;
}

int prime(int n)
{
   int flag=0;
   for(int i=2;i<n;i++)
   {
      if(n%i==0)
      {
         flag=1;
         break;
      }
      else
      {
         continue;
      }
```

```
    }
    if(flag==0)
        return n;
    else
        return prime(n+1);
}
struct hash_table* initialize_table(int size)
{
    struct hash_table* h=(struct hash_table *)malloc(sizeof(struct hash_table));
    if(h==NULL)
    {
        printf("memory not allocated\n");
        return NULL;
    }
    else
    {
        h->table_size=size;
        h->the_cells=(struct cell *)malloc(sizeof(struct cell)*h->table_size);
        if(h->the_cells==NULL)
        {
            printf("memory not allocated for table cells\n");
            return NULL;
        }
        else
        {
            for(int i=0;i<h->table_size;i++)
            {
                h->the_cells[i].info=empty;
            }
        }
    }
    return h;
}
int find_linear(int key,struct hash_table *h)
{
    int pos=hash(key,h->table_size);
    while(h->the_cells[pos].element!=key && h->the_cells[pos].info!=empty)
            pos=(pos+1)%h->table_size;
    return pos;
}
int find_quadratic(int key,struct hash_table *h)
{
    int i=0;
    int pos=hash(key,h->table_size);
    while(h->the_cells[pos].element!=key && h->the_cells[pos].info!=empty)
    {
        pos=(pos+(2*(++i))-1)%h->table_size;
    }
    return pos;
}
int prime2(int n)
{
    int flag=0;
    for(int i=2;i<n;i++)
    {
      if(n%i==0)
```

```
        {
            flag=1;
            break;
        }
        else
        {
            continue;
        }
    }
    if(flag==0)
        return n;
    else
        return prime2(n-1);
}
int hash2(int key,int size)
{
    int r=prime2(size-1);
    return (r-(key%r));
}
int find_double(int key,struct hash_table *h)
{
    int pos=hash(key,h->table_size);
    int step=hash2(key,h->table_size);
     while(h->the_cells[pos].element!=key && h->the_cells[pos].info!=empty)
    {
        pos=(pos+step)%h->table_size;
    }
    return pos;

}
void insert(struct hash_table * h,int key,int op)
{
    int pos;
    if(op==1)
     pos=find_linear(key,h);
    else if(op==2)
     pos=find_quadratic(key,h);
     else if(op==3)
     pos=find_double(key,h);

    if(h->the_cells[pos].info==empty)
    {
        h->the_cells[pos].info=legitimate;
        h->the_cells[pos].element=key;
    }
}

void display(struct hash_table *h)
{
    for(int i=0;i<h->table_size;i++)
    {
        if(h->the_cells[i].info == legitimate)
        {
            printf("index %d: %d\n",i,h->the_cells[i].element);
        }
        else
```

```
        {
            printf("index %d: null\n",i);
        }
    }
}
void rehash(int a[],int n,int size)
{
    struct hash_table *h1=initialize_table(prime(2*size));
    for(int i=0;i<n;i++)
    {
        insert(h1,a[i],1);
    }
    display(h1);
}
void main() {
    int op;

    int size;
    printf("enter table size:");
    scanf("%d",&size);

    int n,x;
    printf("\nenter no of elemnts:");
    scanf("%d",&n);
    int a[n];
    int i=0;
    do
    {
        printf("enter element:");
        scanf("%d",&x);
        a[i]=x;
        i++;
    }while(i<n);
    printf("1 linear probing\n2 quadratic probing\n3 double hashing\n4 rehashing\n");
    int j=0;
    do
    {
        printf("enter opertaion:");
        scanf("%d",&op);
        if(op==4)
        {
            rehash(a,n,size);
        }
        struct hash_table *h=initialize_table(size);
        if(op>4)
        {
            printf("\ninvalid operation");
        }

        else
        {
            for(int k=0;k<n;k++)
                insert(h,a[k],op);
        }
        if(op!=4)
          display(h);
```

```
        free(h->the_cells);
        free(h);
        j++;
    }while(j<4);
}
```

OUTPUT:

```
enter table size:10

enter no of elemnts:5
enter element:13
enter element:5
enter element:23
enter element:9
enter element:35
1 linear probing
2 quadratic probing
3 double hashing
4 rehashing
enter opertaion:2
index 0: null
index 1: null
index 2: null
index 3: 13
index 4: 23
index 5: 5
index 6: 35
index 7: null
index 8: null
index 9: 9
enter opertaion:0
index 0: null
index 1: null
index 2: null
index 3: null
index 4: null
index 5: null
index 6: null
index 7: null
index 8: null
index 9: null
```

RESULT:

Thus the c program to create a hash table and perform collision resolution was executed successfully…