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

1.Single linked list

Aim:

The aim of the program is single linked list is executed

Algorithm:

1. Start
2. Create a structure and functions for each operations
3. Display the main menu
4. Read user choice
5. Execute choice operation
6. Display operation completion
7. Back to main menu
8. Check for exit, if no Execute the operation for the given choice
9. Otherwise end

Program:

```
#include<stdio.h>
#include<stdlib.h>
Struct node{
    Int data;
    Struct node *next;
};
Void insert_begin(struct node*L,int x){
    Struct node *new=(struct node*)malloc(sizeof(struct node));
    If(new!=NULL){
        New->data=x;
        New->next=L->next;
        L->next=new;
    }
    Else{
        Printf("Memory not allocated");
}


}
Void insert_after_p(struct node*L,int x,int pos){
    Struct node *new=(struct node*)malloc(sizeof(struct node));
```

```
        If(new!=NULL){
                New->data=x;
                Struct node *p;
                P=L->next;
                Int i=1;
                While(p!=NULL&&i<pos)
                {
                        P=p->next;
                        I++;
                }
                New->next=p->next;
                p->next=new;
                }
        Else{
                Printf("Menory not allocated");
}
}
Void insert_end(struct node *L,int x)
{
        Struct node *new=(struct node*)malloc(sizeof(struct node));
        If(new!=NULL){
                New->data=x;
                Struct node *p;
                P=L->next;
                While(p->next!=NULL)
                {
                        P=p->next;
                }
                New->next=NULL;
                p->next=new;
        }
}
Int  find(struct node *L,int x)
{
```

```
        Struct node *p;
        P=L->next;int i=1;
        While(p!=NULL&&p->data!=x)
        {
              P=p->next;
              I++;
        }
        If(p!=NULL)
        Printf("Element %d is found at position %d",x,i);
        Else
        Printf("element not found");
}
Int find_next(struct node *L,int x){
        Struct node *p=L->next;

        While(p!=NULL&&p->data!=x){

         P=p->next;

        }
         If(p!=NULL)
         Printf("Next element after %d is %d",x,p->next->data);
         Else
         Printf("NO next element");

}
Int find_prev(struct node*L,int x)
{
        Struct node *p=L->next;
        While(p!=NULL&&p->next->data!=x){
              P=p->next;}
        If(p!=NULL)
        Printf("Previous element before %d is %d",x,p->data);
        Else
```

```
        Printf("NO previous element");
}
Int islast(struct node*L,int x){
        Struct node *p=L->next;
        While(p->data!=x&&p!=NULL){
                P=p->next;}
        If(p->next==NULL){
                Return 1;


        }


        Else
                Return 0;}
Int isempty(struct node *L){
        If(L->next==NULL){
                Return 1;
        }
        Else{
                Return 0;
        }}
Void delete_beginning(struct node *L){
        Struct node *p=L->next;
        L->next=p->next;
        Free(p);}
Void delete_after_p(struct node*L,int pos){
        Struct node *p;
Int i=0;
Struct node *s=L;
While(i<=pos)
{
        P=s;
        S=s->next;
        I++;
}
```

```
        p->next=s->next;
        free(s);
}
Void delete_end(struct node *L){
        Struct node *p;
        P=L->next;
        Struct node *s=p->next;
        While(s->next!=NULL){
                P=p->next;
                S=p->next;
        }
        p->next=NULL;
        free(s);
}
Void delete_list(struct node *L)
{
        Struct node *p=L->next;
        Struct node *s=p->next;
        While(p!=NULL)
        {
                S=p->next;
                Free(p);
                P=s;
        }
        L->next=NULL;}
Void display(struct node *L){
        Struct node *temp=L->next;
        While(temp!=NULL){
                Printf("%d",temp->data);
                Temp=temp->next;
        }}

Int main(){
        Int n,opt,data,position;
```

```
Printf(“Enter the no. Of nodes:”);
Scanf(“%d”,&n);
Struct node *head=(struct node*)malloc(sizeof(struct node));
Struct node*p,*temp=head;
For(int i=0;i<n;i++)
{
    P=(struct node*)malloc(sizeof(struct node));
    Printf(“Enter the nodes:”);
    Scanf(“%d”,&p->data);
    p->next=NULL;
    if(head==NULL){
        head=p=temp;
    }
    Else{
        Temp->next=p;
        Temp=p;
    }
}
Do{
    Printf(“\n1.Insert at first\n2.Insert after p\n3.Insert at end\n4.Find an element\n5.find next element\n6.find previous element\n7.check whether last or not\n8.Check whether empty or not\n9.delete first node\n10.delete after p\n11.delete at the end\n12.delete list\n13.display\n14.exit\n”);
    Printf(“Enter your option:”);
    Scanf(“%d”,&opt);
Switch(opt)
{
    Case 1:
        Printf(“Enter data:”);
        Scanf(“%d”,&data);
        Insert_begin(head,data);
        Display(head);
        Break;
    Case 2:
        Printf(“Enter data”);
        Scanf(“%d”,&data);
```

```
        Printf("Enter position after which to insert:");
        Scanf("%d",&position);
        Insert_after_p(head,data,position);
        Display(head);
        Break;
    Case 3:
     Printf("Enter data:");
     Scanf("%d",&data);
     Insert_end(head,data);
     Display(head);
     Break;
    Case 4:
     Printf("Enter element to be found:");
     Scanf("%d",&data);
     Find(head,data);

     Break;
    Case 5:
     Printf("Enter element to be find next:");
     Scanf("%d",&data);
     Find_next(head,data);

    Break;
   Case 6:
    Printf("Enter element to find previous:");
    Scanf("%d",&data);
    Find_prev(head,data);

    Break;
   Case 7:
    Printf("enter the data to check if its last");
    Scanf("%d",&data);
    Int a= islast(head,data);
    If(a)
```

```
        Printf("%d is the last node",data);
    Else
        Printf("%d is not the last node.",data);
    Break;
   Case 8:
    If(isempty(head))
        Printf("The list is empty");
    Else
        Printf("The list is not empty");
    Break;
   Case 9:
    Delete_beginning(head);
    Display(head);
    Break;
   Case 10:
    Printf("enter position after which to delete a node:");
    Scanf("%d",&position);
    Delete_after_p(head,position);
    Display(head);
    Break;
   Case 11:
    Delete_end(head);
    Display(head);
    Break;
   Case 12:
    Delete_list(head);
    Display(head);
    Break;
   Case 13:
    Display(head);
    Break;
    Case 14:
    Printf("\nExiting the program\n");
    Break;
```

```
                    Default:
    Printf("invalid option");
    Break;
     }
}while(opt!=14);
Return 0;
}
```

Output:

Enter the no. Of nodes:2

Enter the nodes:1

Enter the nodes:3

1.Insert at first

2.Insert after p

3.Insert at end

4.Find an element

5.find next element

6.find previous element

7.check whether last or not

8.Check whether empty or not

9.delete first node

10.delete after p

11.delete at the end

12.delete list

13.display

14.exit

Enter your option:3

Enter data:1

131

1.Insert at first

2.Insert after p

3.Insert at end

4.Find an element

5.find next element

6.find previous element

7.check whether last or not

8.Check whether empty or not

9.delete first node

10.delete after p

11.delete at the end

12.delete list

13.display

14.exit

Enter your option:14

Exit

Result:

The program successfully implemented and excuted

2.DOUBLE LINKED LIST

Aim: The aim of the program is to implement the concepts ;insert,delete,find and dsplay elements in a double linked list

ALGORITHM: Step-by-Step Algorithm

Step 1: Define the Structure for the Doubly Linked List Node

- Define a struct node that contains:
  - int data for storing the value.
  - struct node *next for pointing to the next node.
  - struct node *prev for pointing to the previous node.

Step 2: Initialize Global Variables

- Initialize the head pointer struct node *l = NULL.
- Initialize a temporary node pointer struct node *newnode.

Step 3: Insert at the Beginning (insert_first)

1. Allocate memory for a new node.
2. Check if the allocation is successful.
3. Get the data for the new node from the user.
4. If the list is not empty:
   - Set newnode->next to point to the current head.
   - Set newnode->prev to NULL.
   - Update the current head's previous pointer to newnode.
   - Update the head pointer to newnode.
5. If the list is empty:
   - Set newnode->next and newnode->prev to NULL.
   - Update the head pointer to newnode.

Step 4: Insert at the End (insert_last)

1. Allocate memory for a new node.
2. Check if the allocation is successful.
3. Get the data for the new node from the user.
4. Set newnode->next to NULL.
5. If the list is not empty:
   - Traverse to the last node.
   - Set newnode->prev to point to the last node.
   - Update the last node's next pointer to newnode.
6. If the list is empty:
   - Set newnode->prev to NULL.
   - Update the head pointer to newnode.

Step 5: Insert After a Given Position (insert_afterp)

1. Allocate memory for a new node.
2. Check if the allocation is successful.
3. Get the data for the new node from the user.
4. Get the position after which to insert the new node from the user.
5. Traverse to the node just before the given position.
6. Update pointers to insert the new node after the given position:
   - Set newnode->next to point to the next node.
   - Set newnode->prev to point to the current node.
   - Update the next node's previous pointer to newnode.
   - Update the current node's next pointer to newnode.

Step 6: Delete the First Node (delete_first)

1. If the list is not empty:
   - Set a temporary pointer to the current head.
   - Update the head pointer to the next node.
   - Set the new head's previous pointer to NULL.
   - Free the memory of the temporary node.

Step 7: Delete the Last Node (delete_last)

1. If the list is not empty:
   - Traverse to the second last node.
   - Set a temporary pointer to the last node.
   - Update the second last node's next pointer to NULL.
   - Free the memory of the temporary node.

Step 8: Delete After a Given Position (delete_afterp)

1. Get the position after which to delete the node from the user.
2. Traverse to the node just before the given position.
3. Set a temporary pointer to the node to be deleted.
4. If the node to be deleted is not the last node:
   - Update pointers to remove the node:
     - Set the current node's next pointer to the node after the node to be deleted.
     - Update the next node's previous pointer to the current node.
   - Free the memory of the temporary node.
5. If the node to be deleted is the last node:
   - Update the current node's next pointer to NULL.
   - Free the memory of the temporary node.

Step 9: Find Element at a Given Position (find)

1. Get the position from the user.

2. Traverse to the node at the given position.
3. Print the data of the node at the given position.

Step 10: Find Next Element (find_next)

1. Get the position from the user.
2. Traverse to the node at the given position.
3. Print the data of the node next to the given position.

Step 11: Find Previous Element (find_previous)

1. Get the position from the user.
2. Traverse to the node at the given position.
3. Print the data of the node previous to the given position.

Step 12: Display the List (display)

1. If the list is not empty:
   - Traverse through the list from the head to the last node.
   - Print the data of each node.
2. If the list is empty, print "List is empty".

Step 13: Main Function (main)

1. Display the menu options.
2. Repeatedly:
   - Get the user's choice.
   - Call the appropriate function based on the user's choice.
3. Exit the loop if the user selects the exit option.

PROGRAM:

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


struct node {
   int data;
   struct node *next;
   struct node *prev;
 };
struct node *l=NULL;
struct node *newnode;
```

```
void insert_first(){
    newnode=(struct node*)malloc(sizeof(struct node));
    if(newnode!=NULL){
        printf("Enter the element : ");
        scanf("%d",&newnode->data);
        if (l!=NULL){
        newnode->next=l;
        newnode->prev=NULL;
        l->prev=newnode;
        l=newnode;
        }
        else
        {
            newnode->next=NULL;
            newnode->prev=NULL;
            l=newnode;
        }

    }

}

void insert_last(){
    newnode=(struct node*)malloc(sizeof(struct node));
    struct node*p;
    if(newnode!=NULL){
```

```
        printf("enter no");
        scanf("%d",&newnode->data);
        newnode->next=NULL;
        if (l!=NULL){
            p=l;
            while(p->next!=NULL){
                p=p->next;
            }
            newnode->prev=p;
            p->next=newnode;

        }
        else
        {
            newnode->prev=NULL;
            l=newnode;
        }
    }

}

void insert_afterp(){
    newnode=(struct node*)malloc(sizeof(struct node));
    struct node *p=l;
    int pos,c=1;
    if(newnode!=NULL){
        printf("Enter the element : ");
```

```
        scanf("%d",&newnode->data);
        printf("enter the positon");
        scanf("%d",&pos);
        while(c<pos-1){
            p=p->next;
            c++;
        }
        newnode->next=p->next;
        newnode->prev=p;
        p->next=newnode;
        p->next->prev=newnode;
    }
}

void delete_first(){
    struct node*p=l;
    l=l->next;
    l->prev=NULL;
    free(p);
}

void delete_last(){
    struct node*p=l;
    struct node*temp;
    while(p->next->next!=NULL){
        p=p->next;
    }
```

```
    temp=p->next;
    p->next=NULL;
    free(temp);

}

void delete_afterp(){
    struct node*p=l;
    struct node *temp;
    int pos,c=1;
    printf("Enter position : ");
    scanf("%d",&pos);
    while(c<pos){
        p=p->next;
        c++;
    }
    temp=p->next;
    if (temp->next==NULL){
        free(temp);
        p->next=NULL;
    }
    else{
        p->next=temp->next;
        temp->next->prev=p;
        free(temp);
    }
}
```

```
void find(){
    struct node *p=l;
    int pos,c=1;
    printf("enter position ");
    scanf("%d",&pos);
    while(c<pos){
        p=p->next;
        c++;
    }
    printf("The element present at the position %d is %d",pos,p->data);
}

void find_next(){
    struct node *p=l;
    int pos,c=1;
    printf("Enter the position : ");
    scanf("%d",&pos);
    while(c<pos){
        p=p->next;
        c++;
    }
    printf("The element present next to the position %d is %d",pos,p->next->data);
}

void find_previous(){
    struct node *p=l;
```

```
    int pos,c=1;
    printf("enter position ");
    scanf("%d",&pos);
    while(c<pos){
        p=p->next;
        c++;
    }
    printf("The element present previous to the position %d is %d",pos,p->prev->data);
}
void display(){
    if (l!=NULL){
        struct node*p;
        p=l;
        while(p!=NULL){
            printf("%d",p->data);
            p=p->next;
        }
    }
    else
        printf("List is empty");
}

void main(){
    int n;
    struct node *l=NULL;
    printf("options\n1.enter at first\n2.enter at last\n3.insert after p\n4.delete first element\n5.delete last\n6.delete
after p\n7.find\n8.find next\n9.find previous\n10.display\n11.exit\n");
    do{
```

```
printf("\nEnter your option:");
scanf("%d",&n);
switch(n){
   case 1:
      insert_first();
      break;
   case 2:
      insert_last();
      break;
   case 3:
      insert_afterp();
      break;
   case 4:
      delete_first();
      break;
   case 5:
      delete_last();
      break;
   case 6:
      delete_afterp();
      break;
   case 7:
      find();
      break;
   case 8:
      find_next();
      break;
```

```
            case 9:
                find_previous();
                break;
            case 10:
                display();
                break;
        }
    }while(n!=11);
}
```

OUTPUT:

options

1.enter at first

2.enter at last

3.insert after p

4.delete first element

5.delete last

6.delete after p

7.find

8.find next

9.find previous

10.display

11.exit

Enter your option:1

Enter the element : 10

Enter your option:2

enter no20

Enter your option:10

10 20

Enter your option:4

Enter your option:10

20

Enter your option:1

Enter the element : 15

Enter your option:7

enter position 1

The element present at the position 1 is 15

Enter your option:6

Enter position : 1

Enter your option:10

15

Enter your option:2

Result: The program implements the concepts ;insert,delete,find and display elements in a double linked list successfully

3.Polynomial manipulation

Aim:

To implement polynomial manipulation in c, namely , polynomial addition , polynomial subtraction and polynomial multiplication using single linked list.

Algorithm:

1. **Start:** Begin the program.

2. **Declare Structures and Function Prototypes:** Define a structure for polynomial terms (struct Term) and declare function prototypes for creating terms, inserting terms into polynomials, displaying polynomials, and performing polynomial operations (addition, subtraction, multiplication).

3. **Main Function:** Start the main function.

4. **Initialize Variables:** Declare variables for the choice of operation (choice), coefficients, exponents, and polynomial pointers (poly1, poly2, result).

5. **Operation Menu:** Display a menu for the user to choose the operation they want to perform (addition, subtraction, multiplication, or exit).

6. **Input Polynomials:** Prompt the user to input coefficients and exponents for the two polynomials based on the chosen operation.

7. **Perform Operation:** Depending on the user's choice, call the corresponding function (addPolynomials, subtractPolynomials, multiplyPolynomials) to perform the operation on the input polynomials.

8. **Display Result:** Display the resultant polynomial after the operation.

9. **Free Memory:** Free memory allocated for the polynomials after each operation to prevent memory leaks.

10. **Repeat or Exit:** Ask the user if they want to continue with another operation or exit the program. If they choose to continue, repeat steps 5 to 9. If they choose to exit, end the program.

Code:

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

```
#include <stdlib.h>

// Define a structure for the polynomial term
Struct Term {
Int coefficient;
Int exponent;
Struct Term* next;
};

// Function prototypes
Struct Term* createTerm(int coefficient, int exponent);
Void insertTerm(struct Term** poly, int coefficient, int exponent);
Void displayPolynomial(struct Term* poly);
Struct Term* addPolynomials(struct Term* poly1, struct Term* poly2);
Struct Term* subtractPolynomials(struct Term* poly1, struct Term* poly2);
Struct Term* multiplyPolynomials(struct Term* poly1, struct Term* poly2);
Void freePolynomial(struct Term* poly);

Int main() {
Struct Term *poly1 = NULL, *poly2 = NULL, *result = NULL;
Int choice, coefficient, exponent;

While (1) {
Printf("\n1. Add polynomials\n2. Subtract polynomials\n3. Multiply polynomials\n4. Exit\nEnter your choice:
");
Scanf("%d", &choice);

Switch (choice) {
Case 1:
Printf("Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):\n");
While (1) {
Scanf("%d %d", &coefficient, &exponent);
If (coefficient == 0 && exponent == 0)
Break;
```

```
insertTerm(&poly1, coefficient, exponent);
}
Printf(“Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):\n”);
While (1) {
Scanf(“%d %d”, &coefficient, &exponent);
If (coefficient == 0 && exponent == 0)
Break;
insertTerm(&poly2, coefficient, exponent);
}
Printf(“Resultant polynomial after addition: “);
Result = addPolynomials(poly1, poly2);
displayPolynomial(result);
freePolynomial(result);
break;
case 2:
printf(“Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):\n”);
while (1) {
scanf(“%d %d”, &coefficient, &exponent);
if (coefficient == 0 && exponent == 0)
break;
insertTerm(&poly1, coefficient, exponent);
}
Printf(“Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):\n”);
While (1) {
Scanf(“%d %d”, &coefficient, &exponent);
If (coefficient == 0 && exponent == 0)
Break;
insertTerm(&poly2, coefficient, exponent);
}
Printf(“Resultant polynomial after subtraction: “);
Result = subtractPolynomials(poly1, poly2);
displayPolynomial(result);
freePolynomial(result);
break;
```

```
case 3:
printf(“Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):\n”);
while (1) {
scanf(“%d %d”, &coefficient, &exponent);
if (coefficient == 0 && exponent == 0)
break;
insertTerm(&poly1, coefficient, exponent);
}
Printf(“Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):\n”);
While (1) {
Scanf(“%d %d”, &coefficient, &exponent);
If (coefficient == 0 && exponent == 0)
Break;
insertTerm(&poly2, coefficient, exponent);
}
Printf(“Resultant polynomial after multiplication: “);
Result = multiplyPolynomials(poly1, poly2);
displayPolynomial(result);
freePolynomial(result);
break;
case 4:
exit(0);
default:
printf(“Invalid choice!\n”);
}
// Clear the polynomials after each operation
freePolynomial(poly1);
freePolynomial(poly2);
poly1 = NULL;
poly2 = NULL;
}

Return 0;
}
```

```
// Function to create a polynomial term
Struct Term* createTerm(int coefficient, int exponent) {
Struct Term* newTerm = (struct Term*)malloc(sizeof(struct Term));
If (!newTerm) {
Printf("Memory allocation failed.\n");
Exit(1);
}
newTerm->coefficient = coefficient;
newTerm->exponent = exponent;
newTerm->next = NULL;
return newTerm;
}

// Function to insert a term into a polynomial in sorted order of exponents
Void insertTerm(struct Term** poly, int coefficient, int exponent) {
Struct Term* newTerm = createTerm(coefficient, exponent);
If (*poly == NULL || exponent > (*poly)->exponent) {
newTerm->next = *poly;
*poly = newTerm;
} else {
Struct Term* temp = *poly;
While (temp->next != NULL && temp->next->exponent > exponent) {
Temp = temp->next;
}
newTerm->next = temp->next;
temp->next = newTerm;
}
}

// Function to display a polynomial
Void displayPolynomial(struct Term* poly) {
While (poly != NULL) {
Printf("%dx^%d ", poly->coefficient, poly->exponent);
If (poly->next != NULL)
```

```
Printf(“+ “);
Poly = poly->next;
}
Printf(“\n”);
}

// Function to add two polynomials
Struct Term* addPolynomials(struct Term* poly1, struct Term* poly2) {
Struct Term* result = NULL;
While (poly1 != NULL && poly2 != NULL) {
If (poly1->exponent > poly2->exponent) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
} else if (poly1->exponent < poly2->exponent) {
insertTerm(&result, poly2->coefficient, poly2->exponent);
poly2 = poly2->next;
} else {
insertTerm(&result, poly1->coefficient + poly2->coefficient, poly1->exponent);
poly1 = poly1->next;
poly2 = poly2->next;
}
}
While (poly1 != NULL) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
}
While (poly2 != NULL) {
insertTerm(&result, poly2->coefficient, poly2->exponent);
poly2 = poly2->next;
}
Return result;
}

// Function to subtract two polynomials
```

```
Struct Term* subtractPolynomials(struct Term* poly1, struct Term* poly2) {
Struct Term* result = NULL;
While (poly1 != NULL && poly2 != NULL) {
If (poly1->exponent > poly2->exponent) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
} else if (poly1->exponent < poly2->exponent) {
insertTerm(&result, -(poly2->coefficient), poly2->exponent);
poly2 = poly2->next;
} else {
insertTerm(&result, poly1->coefficient – poly2->coefficient, poly1->exponent);
poly1 = poly1->next;
poly2 = poly2->next;
}
}
While (poly1 != NULL) {
insertTerm(&result, poly1->coefficient, poly1->exponent);
poly1 = poly1->next;
}
While (poly2 != NULL) {
insertTerm(&result, -(poly2->coefficient), poly2->exponent);
poly2 = poly2->next;
}
Return result;
}

// Function to multiply two polynomials
Struct Term* multiplyPolynomials(struct Term* poly1, struct Term* poly2) {
Struct Term* result = NULL;
Struct Term* temp1 = poly1;
While (temp1 != NULL) {
Struct Term* temp2 = poly2;
While (temp2 != NULL) {
insertTerm(&result, temp1->coefficient * temp2->coefficient, temp1->exponent + temp2->exponent);
```

```
temp2 = temp2->next;
}
Temp1 = temp1->next;
}
Return result;
}

// Function to free memory allocated for polynomial
Void freePolynomial(struct Term* poly) {
Struct Term* temp;
While (poly != NULL) {
Temp = poly;
Poly = poly->next;
Free(temp);
}
}
```

Output:

1.Add polynomials

2. Subtract polynomials

3. Multiply polynomials

4. Exit

Enter your choice: 1

Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):

2 3 4 2 1 1 0 0

Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):

3 3 1 2 5 1 0 0

Resultant polynomial after addition : 5x^3 + 5x^2 + 6x^1

1. Add polynomials

2. Subtract polynomials

3. Multiply polynomials

4. Exit

Enter your choice: 2

Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):

7 3 5 1 0 0

Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):

8 3 2 2 3 1 0 0

Resultant polynomial after subtract ion: -1x^3 + -2x^2 + 2x^1

1.Add polynomials

2. Subtract polynomials

3. Multiply polynomials

4. Exit

Enter your choice: 3

Enter the first polynomial (coefficient followed by exponent, enter 0 0 to finish):

2 3 2 2 1 1

0 0

Enter the second polynomial (coefficient followed by exponent, enter 0 0 to finish):

3 2 1 3 0 0

Resultant polynomial after multiplication: 2x^6 + 2x^5 + 6x^5 + 1 x^4 + 6x^4 + 3x^3

Result:

Thus, the program was implemented successfully using Singly linked list .

4.IMPLEMENTATION OF STACK

Aim:

The aim of the program is to execute stack using array and linked list.

ALGORITHM:

1.Start

2.Initialize: Create an empty stack and set its maximum size if applicable.

3.Push: To add an element onto the stack, increment the stack pointer and place the new element into the location pointed to by the stack pointer.

4.Pop: To remove an element from the stack, return the element at the current stack pointer location and then decrement the stack pointer.

5.Peek: To view the top element of the stack without removing it, return the element at the current stack pointer location.

6.isEmpty: Check if the stack is empty by examining if the stack pointer is pointing to the base of the stack.

7.isFull (if there's a maximum size): Check if the stack is full by comparing the stack pointer to the maximum size.

8.End

PROGRAM USING ARRAY:

```
#include <stdio.h>

#define MAX_SIZE 10

typedef struct {
   int data[MAX_SIZE];
   int top;
```

```
} Stack;

void initStack(Stack* stack) {
    stack->top = -1;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

int isFull(Stack* stack) {
    return stack->top == MAX_SIZE - 1;
}

void push(Stack* stack, int element) {
    if (isFull(stack)) {
        printf("Stack overflow!\n");
        return;
    }
    stack->data[++stack->top] = element;
}

int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        return -1;
    }
    return stack->data[stack->top--];
}

int top(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return -1;
```

```
    }
    return stack->data[stack->top];
}

void displayStack(Stack* stack) {
    int i;
    for (i = 0; i <= stack->top; i++) {
        printf("%d ", stack->data[i]);
    }
    printf("\n");
}

int main() {
    Stack stack;
    initStack(&stack);

    int choice, element;

    while (1) {
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Top\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &element);
                push(&stack, element);
                break;
            case 2:
```

```
            element = pop(&stack);
            if (element != -1) {
                printf("Popped element: %d\n", element);
            }
            break;
        case 3:
            element = top(&stack);
            if (element != -1) {
                printf("Top element: %d\n", element);
            }
            break;
        case 4:
            displayStack(&stack);
            break;
        case 5:
            return 0;
        default:
            printf("Invalid choice!\n");
      }
   }

   return 0;
}
```

OUTPUT:

```
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter element to push: 10
1. Push
2. Pop
```

3. Top

4. Display

5. Exit

Enter your choice: 5

PROGRAM USING LINKED LIST:

```
#include <stdio.h>

#define MAX_SIZE 10

typedef struct {
    int data[MAX_SIZE];
    int top;
} Stack;

void initStack(Stack* stack) {
    stack->top = -1;
}

int isEmpty(Stack* stack) {
    return stack->top == -1;
}

int isFull(Stack* stack) {
    return stack->top == MAX_SIZE - 1;
}

void push(Stack* stack, int element) {
    if (isFull(stack)) {
        printf("Stack overflow!\n");
        return;
    }
```

```
    stack->data[++stack->top] = element;
}

int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        return -1;
    }
    return stack->data[stack->top--];
}

int top(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return stack->data[stack->top];
}

void displayStack(Stack* stack) {
    int i;
    for (i = 0; i <= stack->top; i++) {
        printf("%d ", stack->data[i]);
    }
    printf("\n");
}

int main() {
    Stack stack;
    initStack(&stack);

    int choice, element;

    while (1) {
```

```
printf("1. Push\n");
printf("2. Pop\n");
printf("3. Top\n");
printf("4. Display\n");
printf("5. Exit\n");
printf("Enter your choice: ");
scanf("%d", &choice);

switch (choice) {
    case 1:
        printf("Enter element to push: ");
        scanf("%d", &element);
        push(&stack, element);
        break;
    case 2:
        element = pop(&stack);
        if (element!= -1) {
            printf("Popped element: %d\n", element);
        }
        break;
    case 3:
        element = top(&stack);
        if (element!= -1) {
            printf("Top element: %d\n", element);
        }
        break;
    case 4:
        displayStack(&stack);
        break;
    case 5:
        return 0;
    default:
        printf("Invalid choice!\n");
}
```

```
    }

    return 0;
}
```

OUTPUT:

```
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 1
Enter element to push: 10
1. Push
2. Pop
3. Top
4. Display
5. Exit
Enter your choice: 5
```

RESULT:

The output is verified successfully for the above program.

5.Infix to postfix

Aim:

The aim of the provided code is to convert an infix expression to a postfix expression in c

Algorithm:

- **input**: Read the infix expression from the user.
- **Process the Expression**:

- Initialize an empty stack.
- Loop through each character of the expression.
- If it's an operand (alphabet), output it directly.
- If it's an operator:
- While the stack is not empty and the precedence of the current operator is less than or equal to the precedence of the operator at the top of the stack, pop and output the top operator.
- Push the current operator onto the stack.
- If it's an opening parenthesis, push it onto the stack.
- If it's a closing parenthesis:
- Pop and output operators from the stack until an opening parenthesis is encountered.
- Discard the opening parenthesis.
- Ignore spaces.
- If it's any other character, print "Invalid expression" and terminate.

- **Output**: After processing all characters, pop and output any remaining operators from the stack.

Program:

```
#include <stdio.h>
char str[25];
int stack[25];
int top=-1;

int priority(char sym)
{
   if(sym=='*'||sym=='/'||sym=='%')
   return 2;
   else if(sym=='+'||sym=='-')
   return 1;
   else
   return 0;
```

```
}

void push(int num)
{
   top++;
   stack[top]=num;
}

void pop()
{
   if(top<0)
   {
      printf("\n---Stack Underflow---\n");
   }
   else
   {
      top--;
   }
}

char peek()
{
   if(top<0)
   return '\0';
   else
   return stack[top];
}

void display()
{
   if(top<0)
   printf("\nThe stack is empty\n");

   for(int i=top;i>=0;i--)
```

```
    {
        printf(" | %d |\n",stack[i]);
    }
}

void check_precedence(char in_exp)
{
    char in_stack=peek();

    if(priority(in_stack) >= priority(in_exp))
    {
        printf("%c",in_stack);
        pop();
        check_precedence(in_exp);
    }

    else
    {
        push(in_exp);
    }
}

int main()
{
    printf("Enter expression containing lowercase alphabets and operators (+,-,*,/,%)\n");
    scanf("%[^\n]s",str);
    for(int i=0;str[i]!='\0';i++)
    {
        if((str[i]>='a'&& str[i]<='z')||(str[i]>='A' && str[i]<='Z'))
        {
            printf("%c",str[i]);
        }

        else if(str[i]=='+'||str[i]=='-'||str[i]=='*'||str[i]=='/')
```

```
        {
            check_precedence(str[i]);
        }

        else if(str[i]==' ')
        continue;

        else if(str[i]=='(')
        push(str[i]);

        else if(str[i]==')')
        {
            while(peek()!='(')
            {
                printf("%c",peek());
                pop();
            }
            pop();
        }
        else
        {
            printf("Invalid expression ");
            break;
        }
    }

    while(top!=-1)
    {
        printf("%c",peek());
        pop();
    }
    return 0;
}
```

Output:

Enter expression containing lowercase alphabets and operators (+,-,*,/,%)

a b +

ab+

Result:

The output is verified successfully for the above program.

6.APPLICATION OF STACK

AIM:

The code aims to evaluating arithmetic expressions efficiently handles operator precedence

And parenthesis by using stack.

ALGORITHM:

1. Start

2. Create an empty stack to hold operands.

3. Initialize a variable top to -1 which represents the top of the stack

4. Read the input from user

5. Iterate through each character in the expression

6. If the character is a digit, convert the character to its integer value and push the integer into stack.

7. if the character is an operator, pop the top two operands from the stack and perform the corresponding operation. 8. Get the result and display it

9. End

PROGRAM:

```
/*Evaluation of postfix expression*/
#include<stdio.h>
#include<ctype.h>

int stack[1000],top=-1;

void push(char x){
   top++;
   stack[top]=x;
}
int pop(){
   int x=stack[top];
   top--;
   return x;
}

int main(){
   char ep[1000];
   printf("Enter the postfix expression: ");
```

```
    scanf("%s",ep);
    for(int i=0;ep[i]!='\0';i++){
        if(isdigit(ep[i])){
            push((ep[i]-'0'));
        }
        else {
            int a = pop();
            int b = pop(),c;
            switch(ep[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;

            }
        }
    }
    printf("The answer is %d".,pop());
    return 0;

}
```

OUTPUT:

Enter the postfix expression: 12+

The answer is 3.

RESULT: Thus, the program has been successfully executed.

7.IMPLEMENTATION OF QUEUE USING ARRAY AND LINKED LIST.

Aim:

The aim of the program is to execute queue using array and linked list.

Algorithm:

- Start
- To enqueuer an element read the value
- If rear is equal to MAX_SIZE-1, print Queue is full
- Otherwise if front is -1, set front to 0 , increment rear by 1 and assign the value to queue[rear].
- To dequeuer an element if front is -1 print Queue is emprty and return 1
- Otherwise assign element as queue[front], increment front by 1
- End

Program using array:

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

#define MAX_SIZE 100

int queue[MAX_SIZE];
 int front = -1, rear = -1;

void enqueue(int value);
 int dequeue();
void display();

int main() {
     enqueue(10);
     enqueue(20);
     enqueue(30);

   display();

   dequeue();

   display();
   return 0;
}
```

```
void enqueue(int value) {
    if (rear == MAX_SIZE - 1) {
                printf("Queue is full.\n");
   } else {
       if (front == -1) {
           front = 0;
       }
        rear++;
        queue[rear] = value;
   }
}

int dequeue() {
    int element;
    if (front == -1) {
        printf("Queue is empty.\n");
        return -1;
    } else {
        element = queue[front];
        front++;
        if (front > rear) {
           front = rear = -1;
        }
       return element;
   }
}

void display() {
    if (front == -1) {
        printf("Queue is empty.\n");
   } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
           printf("%d ", queue[i]);
        }
        printf("\n");
   }
```

```
}
```

Output:

Queue elements: 10 20 30

Queue elements: 20 30

Program using linked list:

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

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

struct Node* front = NULL;
struct Node* rear = NULL;

void enqueue(int value);
 int dequeue();
 void display();

int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);

   display();

   dequeue();

   display();

   return 0;
}
```

```
void enqueue(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
     newNode->data = value;
    newNode->next = NULL;

    if (rear == NULL) {
         front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
     if (front == NULL) {
         printf("Queue is empty.\n");
        return -1;
     } else {
        struct Node* temp = front;
         int element = temp->data;
         front = front->next;
         free(temp);
         if (front == NULL) {
             rear = NULL;
        }
        return element;
    }
}

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

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

Output:

Queue elements: 10 20 30

Queue elements: 20 30

Result:

The output is verified successfully for the above program.

8.TREE TRAVERSAL-Techniques

Aim:

The aim of the provided C code is to implement a binary tree data structure along with functions for creating the tree, and performing pre-order, in-order, and post-order traversals on the tree.

Algorithm:

1. Start
2. Defines a structure Node representing a node in the binary search tree. Each node contains data, left child pointer, and right child pointer.
3. Provides a function to 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. To delete a node from the binary search tree while preserving the BST property. Create a function to handle cases where the node has zero, one, or two children by finding the successor node and replacing the node to be deleted with it.
6. Create a function to find the node with the minimum value in a subtree, which is used in deletion operation.
7. To search for a value in the binary search tree, Create a recursive function to traverses the tree, comparing the value with each node's data until the value is found or the tree is exhausted.
8. Provide a function to perform an inorder traversal of the binary search tree, printing the nodes in sorted order.
9. End.

Program:

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

struct Node
{
    int data;
    struct Node* left;
```

```
    struct Node* right;
};

void create_tree(struct Node*root)
{
    int l,r;
    printf("Enter the value for left child of %d: ",root->data);
    scanf("%d",&l);
    if(l!=0)
    {
        struct Node*left_node = (struct Node*) malloc (sizeof (struct Node));
        left_node->data=l;
        root->left=left_node;
        create_tree(left_node);
    }
    else root->left=NULL;

    printf("Enter the value for right child of %d: ",root->data);
    scanf("%d",&r);
    if(r!=0)
    {
        struct Node*right_node = (struct Node*) malloc (sizeof (struct Node));
        right_node->data=r;
        root->right=right_node;
        create_tree(right_node);
    }
    else root->right=NULL;
}

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);
    }
}

void main()
{
    struct Node tree;
    tree.left = NULL;
    tree.right = NULL;
    printf("Enter the data for root node: ");
    scanf("%d",&tree.data);
    create_tree(&tree);
    printf("\nPreorder traversal:\n");
```

```
    preorder(&tree);
    printf("\nInorder traversal:\n");
    inorder(&tree);
    printf("\nPostorder traversal:\n");
    postorder(&tree);
    return;
}
```

Output:

Enter the data for root node: 1

Enter the value for left child of 1: 2

Enter the value for left child of 2: 4

Enter the value for left child of 4: 8

Enter the value for left child of 8: 0

Enter the value for right child of 8: 0

Enter the value for right child of 4: 0

Enter the value for right child of 2: 5

Enter the value for left child of 5: 0

Enter the value for right child of 5: 10

Enter the value for left child of 10: 0

Enter the value for right child of 10: 0

Enter the value for right child of 1: 3

Enter the value for left child of 3: 6

Enter the value for left child of 6: 0

Enter the value for right child of 6: 0

Enter the value for right child of 3: 7

Enter the value for left child of 7: 0

Enter the value for right child of 7: 9

Enter the value for left child of 9: 0

Enter the value for right child of 9: 0

Preorder:

1 2 4 8 5 10 3 6 7 9

Inorder:

8 4 2 5 10 1 6 3 7 9

Postorder:

8 4 10 5 2 6 9 7 3 1

Result:

The output is verified successfully for the above program.

****

9.IMPLEMENTATION OF BINARY SEARCH TREE

Aim:

To implement a binary search tree (BST) with insertion, deletion, search, and in-order traversal operations using C programming language.

Algorithm:

1. Start.
2. Define the structure of the tree node.
3. Implement a function to create a new node.
4. Implement a function to insert a node into the BST.
5. Implement a function to find the minimum value node in the BST.
6. Implement a function to delete a node from the BST.
7. Implement a function to perform in-order traversal of the BST.
8. Implement a function to search for a value in the BST.
9. Create a main function to provide a menu-driven interface for inserting, deleting, searching, and displaying the BST.
10. End.

Program:

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

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

struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->left = NULL;
   newNode->right = NULL;
   return newNode;
}


struct Node* insertNode(struct Node* root, int data) {
   if (root == NULL) {
      root = createNode(data);
   } else if (data < root->data) {
      root->left = insertNode(root->left, data);
   } else {
      root->right = insertNode(root->right, data);
   }
   return root;
}

struct Node* findMin(struct Node* root) {
   while (root->left != NULL) root = root->left;
   return root;
}

struct Node* deleteNode(struct Node* root, int data) {
```

```
    if (root == NULL) {
        printf("Element not present in tree.\n");
        return root;
    } else 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) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

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

struct Node* search(struct Node* root, int data) {
    if (root == NULL || root->data == data) {
        return root;
    } else if (data < root->data) {
        return search(root->left, data);
    } else {
        return search(root->right, data);
    }
}

int main() {
    struct Node* root = NULL;
    int choice, value;
    do {
        printf("\nMenu:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Search\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
 printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insertNode(root, value);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                root = deleteNode(root, value);
                break;
            case 3:
                printf("Enter value to search: ");
                scanf("%d", &value);
                if (search(root, value) != NULL) {
                    printf("Node with value %d found in the binary search tree.\n", value);
                } else {
                    printf("Node with value %d not found in the binary search tree.\n", value);
                }
                break;
            case 4:
                printf("In-order traversal of the binary search tree:\n");
                if (root == NULL) {
                    printf("Tree is empty.\n");
                } else {
                    inorderTraversal(root);
                    printf("\n");
                }
                break;
            case 5:
                printf("Exiting...\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);

    return 0;
}
```

Output:

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter value to insert: 10

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter value to insert: 5

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter value to insert: 15

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 1

Enter value to insert: 2

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

In-order traversal of the binary search tree:

2 5 10 15

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 3

Enter value to search: 10

Node with value 10 found in the binary search tree.

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 2

Enter value to delete: 5

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 4

In-order traversal of the binary search tree:

2 10 15

Menu:

1. Insert

2. Delete

3. Search

4. Display

5. Exit

Enter your choice: 5

Exiting.....

Result:

The program successfully implemented a binary search tree with operations to insert, delete, search, and display nodes using in-order traversal.

10.Implementation of AVL Tree

Aim:

To implement and execute AVL tree and to record the output.

Algorithm:

1. Right Rotation (RR)

- Set p as the right child of T.
- Make T the left child of p.
- Set the right child of T to NULL.
- Return p.

2. Left Rotation (LL)

- Set p as the left child of T.
- Make T the right child of p.
- Set the left child of T to NULL.
- Return p.

3. Right-Left Rotation (RL)

- Set p as the right child of T.
- Set the right child of T to the left child of p.
- Set the left child of p to NULL.
- Set the right child of the new right child of T to p.
- Perform right rotation on T (RR).
- Return the result.

4. Left-Right Rotation (LR)

- Set p as the left child of T.
- Set the left child of T to the right child of p.
- Set the right child of p to NULL.
- Set the left child of the new left child of T to p.
- Perform left rotation on T (LL).
- Return the result.

5. Rotation Based on Balance Factor

- If balance factor (bf) > 1:
- If left child has left child, perform LL rotation.
- Else, perform LR rotation.
- If bf < -1:
- If right child has right child, perform RR rotation.
- Else, perform RL rotation.

- Return the rotated subtree.

6. Calculate Height (ht)

- If rt is NULL, return 0.
- Calculate height of left subtree (lh).
- Calculate height of right subtree (rh).
- Return 1 + max(lh, rh).

7. Balance the AVL Tree (balance)

- Balance left subtree.
- Balance right subtree.
- Calculate heights and balance factor.
- Rotate based on balance factor.
- Return balanced tree.

Program:

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

typedef struct Node Node;
struct Node {
    int data;
    Node* l;
    Node* r;
};

Node* ins(Node* rt, int d) {
    if (rt == NULL) {
        Node* n = (Node*)malloc(sizeof(Node));
        n->data = d;
        n->l = NULL;
        n->r = NULL;
        return n;
    }
    if (d < rt->data)
        rt->l = ins(rt->l, d);
    else if (d > rt->data)
        rt->r = ins(rt->r, d);

    return rt;
}
```

```
Node* find(Node* rt, int d) {
    if (rt == NULL || rt->data == d)
        return rt;
    if (d < rt->data)
        return find(rt->l, d);
    else
        return find(rt->r, d);
}

Node* min(Node* rt) {
    Node* cur = rt;
    while (cur && cur->l != NULL)
        cur = cur->l;
    return cur;
}

Node* del(Node* rt, int d) {
    if (rt == NULL)
        return rt;

    if (d < rt->data) {
        rt->l = del(rt->l, d);
    } else if (d > rt->data) {
        rt->r = del(rt->r, d);
    } else {
        if (rt->l == NULL) {
            Node* tmp = rt->r;
            free(rt);
            return tmp;
        } else if (rt->r == NULL) {
            Node* tmp = rt->l;
            free(rt);
            return tmp;
        }

        Node* tmp = min(rt->r);
        rt->data = tmp->data;
        rt->r = del(rt->r, tmp->data);
    }
```

```
    return rt;
}

Node* RR(Node* T) {
    Node* p = T->r;
    p->l = T;
    T->r = NULL;
    return p;
}

Node* LL(Node* T) {
    Node* p = T->l;
    p->r = T;
    T->l = NULL;
    return p;
}

Node* RL(Node* T) {
    Node* p = T->r;
    T->r = p->l;
    p->l = NULL;
    T->r->r = p;
    return RR(T);
}

Node* LR(Node* T) {
    Node* p = T->l;
    T->l = p->r;
    p->r = NULL;
    T->l->l = p;
    return LL(T);
}

Node* rot(Node* rt, int bf) {
    if (bf > 1) {
        if (rt->l->l != NULL)
            return LL(rt);
        else
            return LR(rt);
    } else if (bf < -1) {
```

```
        if (rt->r->r != NULL)
            return RR(rt);
        else
            return RL(rt);
    }
    return rt;
}

int ht(Node* rt) {
    if (rt == NULL)
        return 0;
    int lh = ht(rt->l);
    int rh = ht(rt->r);
    return 1 + (lh > rh ? lh : rh);
}

Node* balance(Node* rt) {
    if (rt != NULL) {
        rt->l = balance(rt->l);
        rt->r = balance(rt->r);

        int lh = ht(rt->l);
        int rh = ht(rt->r);
        int bf = lh - rh;

        rt = rot(rt, bf);
    }
    return rt;
}

void disp(Node* rt) {
    if (rt == NULL)
        return;
    printf("%d ", rt->data);
    if (rt->l != NULL || rt->r != NULL) {
        printf("{ ");
        disp(rt->l);
        printf(", ");
        disp(rt->r);
        printf("} ");
```

```
    }
}

int main() {
    int ch = 1, val;
    Node* rt = (Node*)malloc(sizeof(Node));
    rt->l = NULL;
    rt->r = NULL;
    printf("Enter the data for root node: ");
    scanf("%d", &rt->data);

    printf("\nOperations:\n 1) Insert\n 2) Delete\n 3) Find\n 4) Display\nPress 0 to exit\n");
    while (ch) {
        printf("\nEnter choice: ");
        scanf("%d", &ch);
        switch (ch) {
        case 1:
            printf("Enter node data to be attached: ");
            scanf("%d", &val);
            rt = ins(rt, val);
            rt = balance(rt);
            break;

        case 2:
            printf("Enter node data to be deleted: ");
            scanf("%d", &val);
            rt = del(rt, val);
            rt = balance(rt);
            break;

        case 3:
            printf("Enter node data to be searched: ");
            scanf("%d", &val);
            Node* fnd = find(rt, val);

            if (fnd)
                printf("\nElement found at address %p\n", fnd);
            else
                printf("\nElement not found\n");
            break;
```

```
        case 4:
            disp(rt);
            printf("\n");
            break;

        case 0:
            printf("Operation terminated\n");
            break;

        default:
            printf("Invalid operation\n");
        }
    }
    return 0;
}
```

Output:

```
Enter the data for root node: 10

Operations:
 1) Insert
 2) Delete
 3) Find
 4) Display
Press 0 to exit

Enter choice: 1
Enter node data to be attached: 9

Enter choice: 1
Enter node data to be attached: 20

Enter choice: 1
Enter node data to be attached: 223

Enter choice: 1
Enter node data to be attached: 26

Enter choice: 4
```

10 { 9 , 26 { 20 , 223 } }

Enter choice: 2

Enter node data to be deleted: 223

Enter choice: 3

Enter node data to be searched: 26

Element found at address 0x6e4b40

Enter choice: 4

10 { 9 , 26 { 20 , } }

Enter choice: 0

Operation terminated

Result:

Thus the code implemented and executed successfully.

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.

Coding:

```
#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:

The program has been successfully implemented.

12.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:

The program has been successfully implemented.

13.TOPOLOGICAL SORT

AIM:

To implement a topological sorting algorithm on a graph using a queue data structure, and to display the sorted vertices.

ALGORITHM:

Step 1: Start the program.

Step 2: Implement the ‘CreateGraph’ function to create a graph with n vertices.

Step 3: Input the number of vertices (n) and the adjacency matrix representing the graph.

Step 4: Initialize the indegree array to store the indegree of each vertex.

Step 5: Define the ‘AddEdge’ function to add edges to the graph and update the indegree array.

Step 6: Create the ‘Topsort’ function to perform the topological sort.

Step 7: Initialize a queue to store vertices with an indegree of 0.

Step 8: Enqueue vertices with an indegree of 0 and dequeue them when their indegree becomes 0.

Step 9: Update the ‘topnum’ array to store the topological order of the vertices.

Step 10: Define ‘DisplayTopSort’ function to display the topological order of the vertices.

Step 11: End the program.

PROGRAM:

```
    #include <stdio.h>
    #include <stdlib.h>
    #define MAX 5
struct Queue {
   int data[MAX];
   int front, rear;
};
struct Graph {
   int vertices[MAX];
   int edges[MAX][MAX];
   int indegree[MAX];
   int topnum[MAX];
};
struct Queue* CreateQueue() {
   struct Queue* q = (struct Queue*)malloc(sizeof(struct Queue));
```

```
    q->front = q->rear = 0;
    return q;
}

void MakeEmpty(struct Queue* q) {
    q->front = q->rear = 0;
}

int IsEmpty(struct Queue* q) {
    return q->front == q->rear;
}


void Enqueue(int vertex, struct Queue* q) {
    q->data[q->rear++] = vertex;
}


int Dequeue(struct Queue* q) {
    return q->data[q->front++];
}

void CreateGraph(struct Graph* g) {
    for (int i = 0; i < MAX; i++) {
        g->vertices[i] = i;
        g->indegree[i] = 0;
        for (int j = 0; j < MAX; j++) {
            g->edges[i][j] = 0;
        }
    }
}

void AddEdge(struct Graph* g, int src, int dest) {
    g->edges[src][dest] = 1;
    g->indegree[dest]++;
}

void Topsort(struct Graph* g) {
    struct Queue* q = CreateQueue();
    MakeEmpty(q);
    int counter = 0;
    for (int i = 0; i < MAX; i++) {
        if (g->indegree[i] == 0) {
            Enqueue(i, q);
        }
    }
    while (!IsEmpty(q)) {
        int v = Dequeue(q);
        g->topnum[v] = ++counter;
        for (int w = 0; w < MAX; w++) {
            if (g->edges[v][w] && --g->indegree[w] == 0) {
                Enqueue(w, q);
            }
        }
    }
    if (counter != MAX) {
        printf("Graph has a cycle\n");
    }
    free(q);
}
```

```
void DisplayTopSort(struct Graph* g) {
    printf("Topological Sorting: ");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            if (g->topnum[j] == i + 1) {
                printf("%d ", j);
                break;
            }
        }
    }
    printf("\n");
}

int main() {
    struct Graph g;
    int numEdges, src, dest;
    CreateGraph(&g);
    printf("Enter the number of edges: ");
    scanf("%d", &numEdges);
    printf("Enter the edges one by one:\n");
    for (int i = 0; i < numEdges; i++) {
        scanf("%d %d", &src, &dest);
        if (src >= MAX || dest >= MAX || src < 0 || dest < 0) {
            printf("Invalid edge. Please enter vertices between 0 and %d.\n", MAX - 1);
            i--;
        } else {
            AddEdge(&g, src, dest);
        }
    }
    Topsort(&g);
    DisplayTopSort(&g);
    return 0;
}
```

OUTPUT:

Enter the number of edges: 4

Enter the edges one by one:

0 1

1 2

2 3

3 4

Topological Sorting: 0 1 2 3 4

RESULT:

Thus, the program has been successfully executed and verified.

14.Prim's Algorithm

Aim:

The aim of the provided code is to implement Prim's algorithm in C to find the minimum spanning tree (MST) of a given graph.

Algorithm:

1. Start

2. Input the number of vertices and the adjacency matrix representing the graph.

3.Find the vertex with the minimum key value among the vertices not yet included in the MST.

4. Initializes key values and mstset for all vertices , then iteratively select the vertex with the minimum key value and updates the key values of its adjacent vertices of a shorter edge is found.

5.Print the edges of the MST along with their weights.

6. End.

Program:

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

#define MAX_VERTICES 10
#define INF 999999

int graph[MAX_VERTICES][MAX_VERTICES];
int vertices;

void createGraph() {
    int i, j;
    printf("Enter the number of vertices: ");
    scanf("%d", &vertices);
    printf("Enter the adjacency matrix:\n");
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            scanf("%d", &graph[i][j]);
```

```
        }
    }
}

int findMinKey(int key[], bool mstSet[]) {
    int min = INF, min_index;
    for (int v = 0; v < vertices; v++) {
        if (mstSet[v] == false && key[v] < min) {
            min = key[v];
            min_index = v;
        }
    }
    return min_index;
}

void printMST(int parent[]) {
    printf("Edge \tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d \t%d \n", parent[i], i, graph[i][parent[i]]);
    }
}

void primMST() {
    int parent[vertices];
    int key[vertices];
    bool mstSet[vertices];

    for (int i = 0; i < vertices; i++) {
        key[i] = INF;
        mstSet[i] = false;
    }

    key[0] = 0;
    parent[0] = -1;
```

```
    for (int count = 0; count < vertices - 1; count++) {
        int u = findMinKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent);
}

int main() {
    createGraph();
    primMST();
    return 0;
}
```

Output:

```
Enter the number of vertices: 5
Enter the 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
Edge    Weight
0 - 1   2
1 - 2   3
0 - 3   6
1 - 4   5
```

Result:

The output is verified successfully for the above program.

15.DIJIKSTRA'S ALGORITHM

AIM:

The code aims to create a graph and to find the shortest path between two vertices using Dijkstra's Algorithm.

ALGORITHM:

1. Start.

2. The main function calls the create graph function.

3. Input the number of vertices and the adjacency matrix representing the graph.

4. Find the vertex with the minimum distance from the source vertex among the vertices.

5. Then, it implements Dijkstra's algorithm, initializes distance values and sptset for all vertices, and iteratively updates the distance values until all vertices are included in the shortest path tree.

6. Display the Output

7. End.

PROGRAM:

```
#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define MAX 100

#define INF INT_MAX


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

   int min = INF, min_index;

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

      if (sptSet[v] == 0 && dist[v] <= min) {

         min = dist[v];

         min_index = v;

      }

   }

   return min_index;

}


void printSolution(int dist[], int V) {
```

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

int main() {
    int V, E;
    printf("Enter the number of vertices: ");
    scanf("%d", &V);
    int graph[MAX][MAX] = {0};
```

```
    printf("Enter the number of edges: ");
    scanf("%d", &E);
    printf("Enter the edges (source destination weight):\n");
    for (int i = 0; i < E; i++) {
        int src, dest, weight;
        scanf("%d %d %d", &src, &dest, &weight);
        graph[src][dest] = weight;
        graph[dest][src] = weight;
    }
    int srcVertex;
    printf("Enter the source vertex: ");
    scanf("%d", &srcVertex);
    printf("Adjacency Matrix of the graph:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            printf("%d ", graph[i][j]);
        }
        printf("\n");
    }
    printf("\nShortest paths from vertex %d using Dijkstra's algorithm:\n", srcVertex);
    dijkstra(graph, V, srcVertex);
    return 0;
}
```

OUTPUT:

```
Enter the number of vertices: 4
Enter the number of edges: 5
Enter the edges (source destination weight):
1 2 5
1 3 3
2 3 8
2 4 2
```

3 4 6

Enter the source vertex: 1

Adjacency Matrix of the graph:

0 0 0 0

0 0 5 3

0 5 0 8

0 3 8 0

Shortest paths from vertex 1 using Dijkstra's algorithm:

Vertex	Distance from source
1	0
2	5
3	3
4	7

RESULT:
Thus the program has been executed successfully

16. SORTING-QUICK SORT

AIM: THE AIM OF THE PROGRAM IS TO IMPLEMENT QUICK SORT-SORTING ALGORITHM

ALGORITHM: Step-by-Step Algorithm

Step 1: *Select a Pivot*

- Any element at random.
- The first or last element.
- Middle element.

Step 2: Rearrange the Array

- The pivot element is compared to all of the items starting with the first index. If the element is greater than the pivot element, a second pointer is appended.
- When compared to other elements, if a smaller element than the pivot element is found, the smaller element is swapped with the larger element identified before.

Step 3: *Divide the Subarrays*

- sort the segment of the array to the left of the pivot, and then sort the segment of the array to the right of the pivot.

Step 4: *Final Sorted Array*

- After sorting both subarrays and placing the pivot in its correct position, we combine the elements to form the sorted array

PROGRAM:

```
#include <stdio.h>

int partition(int arr[], int low, int high)

{

   int pivot = arr[high];

   int i = (low - 1);

   for (int j = low; j <= high - 1; j++)

   {

      if (arr[j] < pivot)

      {
```

```
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
int main()
{
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
```

```
    {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
    printf("\nInitially array elemnts are:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    quickSort(arr, 0, n - 1);
    printf("\nSorted array using Quick Sort:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT:

Original array - 2 11 9 4 13 5

Array after sorting - 2 4 5 9 11 13

RESULT:

THE PROGRAM IMPLEMENTS THE QUICK SORT-SORTING ALGORITHM SUCCESSFULLY.

MERGE SORT

AIM: THE AIM OF THE PROGRAM IS TO IMPLEMENT MERGE SORT-SORTING ALGORITHM

ALGORITHM: Step-by-Step Algorithm

1. If the list has 0 or 1 elements, it is already sorted. Return the list.
2. Find the middle index of the list.
3. Recursively apply merge sort to the left half of the list.
4. Recursively apply merge sort to the right half of the list.
5. Create an empty result list to store the merged elements.
6. Initialize two pointers for the left and right halves, starting at the first element of each half.
7. Compare the elements at the pointers in the left and right halves.
8. Add the smaller element to the result list and move the corresponding pointer.
9. Continue comparing and adding elements until all elements from one half are added to the result list.
10. Add any remaining elements from the other half to the result list and return it.

PROGRAM:

```
#include <stdio.h>

void merge(int arr[], int l, int m, int r)

{

   int i, j, k;

   int n1 = m - l + 1;

   int n2 = r - m;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
```

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

```
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main()
{
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter element %d: ", i + 1);
```

```
        scanf("%d", &arr[i]);
    }
    printf("\nInitially array elements are:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    mergeSort(arr, 0, n - 1);
    printf("\nSorted array using Merge Sort:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

OUTPUT:

Original array - 2 11 9 4 13 5

Array after sorting - 2 4 5 9 11 13

RESULT:

THE PROGRAM IMPLEMENTS THE MERGE SORT-SORTING ALGORITHM SUCCESSFULLY.

17. COLLISION RESOLUTION TECHNIQUES-

AIM: To implement Hash data structure using Open Addressing as a collision resolution technique.

ALGORITHM :

STEP 1:- Declare an array of a linked list with the hash table size.

STEP 2:- Initialize the table with the size and the cells.

STEP 3:- Implement linear probing, quadratic probing, and double hashing functions.

STEP 4:- Implement rehashing to increase table size and reinsert entries, once half of the table is filled.

STEP 5:- Insert the keys entered by user, using anyone of the above collision resolution techniques, as the desire of user.

STEP 6:- Display the final hash table.

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
enum kind_of_entry {empty,delete,legitimate};
typedef struct node{
   int data;
   enum kind_of_entry info;
}*cells;


typedef struct table{
  int t_size;
   cells *list;
}H_table;


int nextprime(int num)
{
   int f;
   if( num==1)
   {
      return 2;
```

```
    }
    while(num)
    {
        for(int i=2;i<num;i++)
        {   f=1;
            if(num%i==0)
            {f=0;
              break;}
        }
        if(f)
        {

          return num;
        }
        num ++;
    }
}
int isprime(int num)
{
    if (num==2)
    {
        return 1;
    }
    for(int i=2;i<num;i++)
    {
        if(num%i==0)
        {
            return 0;
        }
    }
    return 1;
}
H_table* initialize_Table(int size)
{
```

```
    size  = nextprime(size);
    H_table  *H;
    H= (H_table*)malloc(sizeof(H_table));
    if(H!=NULL)
    {
       H->t_size  =size;
       H->list = ( cells*)malloc(sizeof( cells)*H->t_size);
       if(H->list!=NULL)
       {
          for(int i=0;i<H->t_size;i++)
          {
             H->list[i]=(struct node*)malloc(sizeof(struct node));
             if(H->list[i]==NULL)
             {
                printf("Fatal Error\n");
                return NULL;
             }
             H->list[i]->info = empty;

          }
       }
       return H;
    }
    return NULL;

}
int hash(int key,int h_size)
{
    return key%h_size;
}
int  linear_index(int key,H_table *H)
{
     int index;
     index = hash(key,H->t_size);
```

```
    while(H->list[index]->info==legitimate && H->list[index]->data!=key)
      index = (index+1)%H->t_size;
    return index;

}
void linear_probing(int key,H_table *H)
{
    int i = linear_index(key,H);
    if(H->list[i]->info==empty)
    {
        H->list[i]->data = key;
        H->list[i]->info = legitimate;
    }

}
int quadratic_index(int key , H_table *h)
{
   int index;
   int collision_num =0 ;
   index = hash(key,h->t_size);

   while(h->list[index]->info==legitimate && ( h->list[index]->data!=key ))
   {
    {
      index = (2*(++collision_num)-1)+index %h->t_size;}
   }

   return index;
}
void quadratic_probing(int key,H_table *H)
{
   int i = quadratic_index(key , H);
   if(H->list[i]->info==empty)
   {
```

```
        H->list[i]->data=key;
        H->list[i]->info = legitimate;
    }
}
int hash2(int key,int size)
{
  int R = size-1;
  while(!isprime(R))
  {
     R--;
  }

  return R-(key%R);
}
int find_double(int key, H_table *h)
{
    int pos=hash(key,h->t_size);
    int step=hash2(key,h->t_size);

        while(h->list[pos]->data!=key && h->list[pos]->info!=empty)
    {
       pos=(pos+step)%h->t_size;
    }

    return pos;

}
void double_hashing(int key,H_table *H)
{
  int index = find_double(key,H);
  if(H->list[index]->info==empty)
  {
    H->list[index]->data = key;
    H->list[index]->info=legitimate;
```

```
   }

}

H_table* rehashing(H_table *H)
{
    cells *oldlist;
    int i,old_size;
    old_size = H->t_size;
    oldlist = H->list;
    H = initialize_Table(2*H->t_size);

    for(i=0;i<old_size;i++)
    {
      if(oldlist[i]->info==legitimate)
      {
        linear_probing(oldlist[i]->data,H);

      }
    }
    free(oldlist);
    return H;
}
void display(H_table *h)
{
     for(int i=0;i<h->t_size;i++)
    {
        if(h->list[i]->info == legitimate)
        {
            printf("index %d: %d\n",i,h->list[i]->data);
        }
        else
        {
            printf("index %d: NULL\n",i);
```

```
        }
    }
}
int main()
{
  H_table *H;
  int table_size,n=0,count=0;
  printf("Enter the table size:");
  scanf("%d",&table_size);
  H = initialize_Table(table_size);
  printf("You can perform\n1.Linear Probing\n2.Quadratic Probing\n3.Double Hashing\n4.Dispaly\n5.EXIT");
  printf("\nNOTE:Rehashing will be performed if table is filled by more than half!!");
  int opt,key;
  do
  { printf("\nEnter your option:");
    scanf("%d",&opt);
    switch(opt)
    {
      case 1:
      printf("Enter the no. of keys to insert:");
      scanf("%d",&n);
      count +=n;
      while(n--)
      {
         printf("Enter the key:");
         scanf("%d",&key);
         linear_probing(key,H);
      }
      if(count >(H->t_size/2))
        { printf("Rehashing is performed!!\n");
         H= rehashing(H);}
      break;

      case 2:
```

```
printf("Enter the no. of keys to insert:");
scanf("%d",&n);
count +=n;
while(n--)
{
    printf("Enter the key:");
    scanf("%d",&key);
    quadratic_probing(key,H);
}
if(count >(H->t_size/2))
  { printf("Rehashing is performed!!\n");
   H= rehashing(H);}
break;
case 3:
printf("Enter the no. of keys to insert:");
scanf("%d",&n);
count +=n;
while(n--)
{
    printf("Enter the key:");
    scanf("%d",&key);
    double_hashing(key,H);
}
if(count >(H->t_size/2))
  { printf("Rehashing is performed!!\n");
   rehashing(H);}
break;

case 4:
display(H);
break;

 case 5:
 printf("EXITING...");
```

```
        break;

      }
   }while(opt!=5);
   return 0;

}
```

OUTPUT:

You can perform

1.Linear Probing

2.Quadratic Probing

3.Double Hashing

4.Dispaly

5.EXIT

NOTE: Rehashing will be performed if table is filled by more than half!!

Enter your option:1

Enter the no. of keys to insert:6

Enter the key:34

Enter the key:78

Enter the key:34

Enter the key:24

Enter the key:23

Enter the key:71

Rehashing is performed!!

Enter your option:1

Enter the no. of keys to insert:4

Enter the key:89

Enter the key:54

Enter the key:83

Enter the key:02

Enter your option:4

index 0: 23

index 1: 24

index 2: 71

index 3: 2

index 4: NULL

index 5: NULL

index 6: NULL

index 7: NULL

index 8: 54

index 9: 78

index 10: NULL

index 11: 34

index 12: NULL

index 13: NULL

index 14: 83

index 15: NULL

index 16: NULL

index 17: NULL

index 18: NULL

index 19: NULL

index 20: 89

index 21: NULL

index 22: NULL

Enter your option:5

EXITING....

RESULT: Thus, the program was executed successfully.

//Separate chaining

18.COLLISION RESOLUTION TECHNIQUES-Separate Chaining

AIM: To implement Hash data structure using Separate chaining as a collision resolution technique.

ALGORITHM:

STEP 1:- Declare an array of a linked list with the hash table size.

STEP 2:- Initialize an array of a linked list to NULL.

STEP 3:- Find hash key.

STEP 4:- If chain[key] == NULL Make chain[key] points to the key node.

STEP 5:- Otherwise(collision), Insert the key node at the end of the chain[key].

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#define MINSIZE 7

 struct node{
  int data;
  struct node *next;
};
typedef struct node *cell;
typedef struct node *position;
typedef struct table
{
    int t_size;
     cell *list;
}H_table;
int nextprime(int num)
{
    int f;
    if( num==1)
    {
```

```
        return 2;
    }
    while(num)
    {
        for(int i=2;i<num;i++)
        {   f=1;
            if(num%i==0)
            {f=0;
              break;}
        }
        if(f)
        {

          return num;
        }
        num ++;
    }
}
H_table* initialize_table(int size)
{
    if(size < MINSIZE)
    {
        printf("Table size too small\n");
        return NULL;
    }

        H_table *H;
        int i;
        H = (H_table*)malloc(sizeof(H_table));
        if(H==NULL)
        {
            printf("No memory allocated!!\n");
          return NULL;
        }
```

```
        H->t_size = nextprime(size);
        H->list =(cell*)malloc(H->t_size*sizeof(cell));
        for(i=0;i<H->t_size;i++)
        {
          H->list[i] = (struct node*)malloc(sizeof(struct node));

          if(H->list[i]==NULL)
          {
           printf("Error\n");
           return NULL;
          }
          else{
              H->list[i]->next=NULL;
          }
        }
        return H;
}

int hash_index(int key,int h_size)
{
    return key%h_size;
}
position find(int key,H_table *H)
{
    position p;
    cell l;
    l = H->list[hash_index(key,H->t_size)];
    p = l->next;
    while(p!=NULL && p->data!=key)
      p= p->next;
    return p;
}
void insert(int key,H_table *H)
{
```

```
    cell pos,new,l;
    pos = find(key,H);

    if(pos==NULL)/*key is not found*/
    {
        new = (struct node*)malloc(sizeof(struct node));

        if(new==NULL)
        printf("Out of space!!\n");
        else{
            l= H->list[hash_index(key,H->t_size)];
            new->data = key;
            new->next = l->next;
            l->next= new;

        }
    }
}
void display(H_table *H)
{
    int i;
    cell temp;
    for (i=0;i<H->t_size;i++)
    {
        temp = H->list[i]->next;
        if(temp!=NULL)
        printf("\nKeys in index %d:",i);
        while(temp!=NULL)
        {
            printf("%d",temp->data);
            if(temp->next!=NULL)
            {
                printf("->");
            }
```

```
            temp = temp->next;
        }

    }
}
int main()
{
    H_table *H;
    int table_size,n,key;
    printf("Enter the size of the table:");
    scanf("%d",&table_size);
    H = initialize_table(table_size);
    printf("Enter the No. of keys to insert:");

    scanf("%d",&n);
    while(n--){
        printf("Enter the data:");
        scanf("%d",&key);
        insert(key,H);

    }
    printf("\nThe elements in the table are:");
    display(H);
    return 0;

}
```

OUTPUT:

```
Enter the size of the table:9
Enter the No. of keys to insert:7
Enter the data:2
Enter the data:56
Enter the data:78
```

Enter the data:22

Enter the data:69

Enter the data:82

Enter the data:61

The elements in the table are:

Keys in index 0:22

Keys in index 1:78->56

Keys in index 2:2

Keys in index 3:69

Keys in index 5:82

Keys in index 6:61

RESULT: Thus, the program was executed successfully.