Lab 1 Program Code: Selection sort with dynamic memory allocation

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

void selectionsort(int a[], int n) {
   int i, j, temp, pos;

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

      for(j = i+1; j < n; j++) {

         if(a[j] < a[pos]) {
            pos = j;
         }

         if(i != pos) {
            temp = a[i];
            a[i] = a[pos];
            a[pos] = temp;
         }
      }
   }
}

void main() {
   int i, n, *a;

   printf("Enter the array size: ");
   scanf("%d", &n);

   a = (int *) malloc (sizeof(int) * n);

   printf("\nEnter the array elements: ");
   for(i = 0; i < n; i++)
      scanf("%d", &a[i]);

   selectionsort(a, n);

   printf("\nThe sorted array is: ");
   for(i = 0; i < n; i++)
      printf("%d\n", a[i]);
}
```

Lab 2 Program Code: Stack and its Operations

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

int max, *a, top = -1;

void push(int ele);
int pop();
void display();

void main() {
    int choice, ele;

    printf("Enter the size of the stacks: ");
    scanf("%d", &max);

    a = (int *) malloc (sizeof(int) * max);

    while(1) {
        printf("\nEnter your choice: ");
        printf("\n1: Push\t2: Pop\t3: Display\t4: Exit\n");
        scanf("%d", &choice);

        switch(choice) {
            case 1: printf("Enter the element to insert: "); scanf("%d", &ele); push(ele); break;

            case 2: ele = pop(); if(ele == -99) printf("Deleted element is: %d", ele); break;

            case 3: display(); break;

            case 4: free(a); exit(0);

            default: printf("Invalid input");
        }
    }
}

void push(int ele) {
    if(top == max-1) {
        printf("Stack Overflow");
        printf("\tDoubling the size...");
        a = realloc(a, 2 * max * sizeof(int));
    }

    top++;
    a[top] = ele;
}
```

```
int pop() {
    int ele;
    if(top == -1) {
        printf("Stack Underflow");
        return -99;
    }

    else {
        ele = a[top];
        top--;

        return ele;
    }
}

void display() {
    int i;
    if(top == -1) {
        printf("Stack Underflow");
    }

    else {
        printf("\nThe Stack elements are: \n");
        for(i = top; i >= 0; i--) {
            printf("%d", a[i]);
        }
    }
}
```

Lab 3 Program Code: Infix to Postfix to Infix to Evaluation to Answer

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

char istack[20];
float stack[20];
int tos = -1;
int top = -1;

void ipush(char s)
{
    tos++;
    istack[tos] = s;
}

void push(float ele)
{
    top++;
    stack[top] = ele;
}

char ipop()
{
    // int ele = istack[tos];
    // tos--;
    return istack[tos--];
}

float pop()
{
    // float ele = stack[top];
    // top--;
    return stack[top--];
}
```

```
int precd(char s) {
    switch(s) {
        case '^': return 4;
        case '*':
        case '/':
        case '%': return 3;
        case '+':
        case '-': return 2;
        case '(':
        case ')':
        case '#': return 1;
    }
    return 0;
}

void convertip(char infix[20], char postfix[20])
{
    int i, j = 0;
    char symbol;
    ipush('#');
            for(i = 0; i < strlen(infix); i++) {
                    symbol = infix[i];
                    switch (symbol)
        {
        case '(': ipush(symbol);
            break;
        case ')':
            while(istack[tos] != '(')
                postfix[j++] = ipop();
                ipop();
                break;
        case '^':
        case '*':
        case '/':
        case '%':
        case '+':
        case '-':
            while(precd(symbol) <= precd(istack[tos]))
            postfix[j++] = ipop();
            ipush(symbol);
            break;
        default: postfix[j++] = symbol;
        }
    }

    while(istack[tos] != '#')
    postfix[j++] = ipop();

    postfix[j] = '\0';      }
```

```
void evaluate(char postfix[50]) {

    int i;
    char symbol;
    float op1, op2, result, x;

    for(i = 0; i < strlen(postfix); i++) {
                symbol = postfix[i];
    if(isalpha(symbol)) {
                printf("Enter the value for %c: \t", symbol);
                scanf("%f", &x);
                push(x);
}
else
{
        op2 = pop();
        op1 = pop();

        switch(symbol) {
            case '+':
                push(op1+op2); break;

            case '-':
                push(op1-op2); break;

            case '*':
                push(op1*op2); break;

            case '/':
                push(op1/op2); break;

            case '%':
                push((int)op1%(int)op2); break;

            case '^':
                push(pow(op1, op2)); break;

            default: printf("Invalid postfix expression.");
                exit(0);
        }
      }
    }

    result = pop();
    printf("The result is: %f", result);
}
```

```
void main() {
    char infix[20], postfix[20];

    printf("Enter infix expression: ");
    gets(infix);

    convertip(infix, postfix);
    printf("Postfix expression is: %s", postfix);

    evaluate(postfix);
}
```

<u>Lab 4 Program Code:</u> Queue and its operations

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

int rear = -1;
int front = 0;
int max;
int count = 0;
int *q;

void dequeue() {
    int ele;

    if(count == 0) {
        printf("Queue Underflow");
    }
    else {
        ele = q[front];

        front = (front + 1) % max;
        count--;

        printf("\nDeleted element is: %d", ele);
    }
}

void display() {
    int i, k;

    if(count == 0) {
        printf("Queue is empty");
    }
    else {
        printf("\nThe Queue elements are: \n");
        k = front;

        for(i = 0; i < count; i++) {
            printf("%d\t", q[k]);
            k = (k + 1) % max;
        }
    }
}
```

```c
void enqueue() {
    int ele;

    if(count == max)
        printf("Queue Overflow");

    else {
    printf("\nEnter element to be inserted: ");
    scanf("%d", &ele);

    rear = (rear + 1) % max;
    q[rear] = ele;
    count++;
    }
}



void main() {
    int i, ele, choice;

    printf("Enter max queue size: ");
    scanf("%d", &max);

    q = (int *) malloc (sizeof(int) * max);

    while(1) {
        printf("\nEnter your choice: \n1: Enqueue\t2: Dequeue\t3: Display\t4: Exit");
        scanf("%d", &choice);

        switch(choice) {
            case 1: enqueue(); break;

            case 2: dequeue(); break;

            case 3: display(); break;

            case 4: free(q); exit(0);

            default: printf("Invalid input");
        }
    }
}
```

Lab 5 Program Code: Change of Branch

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

struct SLL {
    int usn;
    char name[20];
    char mode[20];
    struct SLL *next;
};

typedef struct SLL node;
node *start = NULL;

node *createNode();
void insertEnd();
void display();
void deleteKey();


void main() {
    int i, n, choice, ele;

    while(1) {
        printf("Enter your choice: ");
        printf("1: Insert\t2: Display\t3: Delete\t4: Add to branch\t5: Exit\n");
        scanf("%d", &choice);

        switch(choice) {
            case 1:
            case 4: printf("Enter number of students: \n");
            scanf("%d", &n);
            for(i = 0; i < n; i++) {
                insertEnd();
            } break;

            case 2: display(); break;

            case 3: deleteKey(); break;

            case 5: exit(0);

            default: printf("Invalid choice");
        }
    }
}
```

```
node *createNode() {
   node *newnode;

   newnode = (node *) malloc (sizeof(node));
   printf("Enter USN | Name | Mode: ");
   scanf("%d %s %s", &newnode->usn, newnode->name, newnode->mode);

   newnode->next = NULL;

   return newnode;
}


void display() {
   node *temp = start;

   if(start == NULL) {
      printf("List empty, cant delete");
      return ;
   }

   printf("Students in class are: \n");

   while(temp != NULL) {
   printf("USN: %d\tName: %s\tMode: %s\n", temp->usn, temp->name, temp->mode);
   temp = temp->next;
   }
}

void insertEnd() {
   node *newnode, *temp;

   newnode = createNode();
   if(start == NULL) {
      start = newnode;
      return;
   }

   temp = start;
   while(temp->next != NULL) {
      temp = temp->next;
   }

   temp->next = newnode;
}
```

```
void deleteKey() {
    node *temp = start, *prev;
    int key;

    if(start == NULL) {
        printf("\nList is empty, cant delete");
    }
    printf("Enter key: ");
    scanf("%d", &key);

    if(start->usn == key) {
        start = start->next;
        printf("deleted: %d", temp->usn);
        free(temp);
    }

    else {
        while(temp != NULL && temp->usn != key) {
            prev = temp;
            temp = temp->next;
        }

        if(temp == NULL) {
            printf("USN invalid");
            return;
        }

        prev->next = temp->next;
        printf("Deleted element is: %d", temp->usn);
        free(temp);
    }
}
```

Lab 6 Program code:

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

struct BST {
    int data;
    struct BST *left, *right;
};

typedef struct BST node;

node *insert(node *root, int key) {
    if(root == NULL) {
        root = (node *) malloc (sizeof(node));
        root->data = key;
        root->left = root->right = NULL;
        return root;
    }
    if(key < root->data)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

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

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

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

```
void search(node *root, int key) {

    if(root == NULL) {
        printf("Key not found"); return;
    }

    if(root->data == key) {
        printf("Key found"); return;
    }

    if(key < root->data) {
        search(root->left, key);
    }
    else
        search(root->right, key);
}

void main() {
    int n, i, key, choice;

    node *temp, *root = NULL;
    printf("Enter number of nodes: ");
    scanf("%d", &n);

    printf("Enter elements: ");
    for(i = 0; i < n; i++) {
        scanf("%d", &key);
        root = insert(root, key);
    }

    while(1) {
        printf("\nEnter choice: ");
        printf("\n1: Inorder\t2: Preorder\t3: Postorder\t4: Search\t5: Exit\n");
        scanf("%d", &choice);

        switch(choice) {

            case 1:
            printf("\nInorder: ");
            inorder(root); break;

            case 2:
            printf("\nPre-order: ");
            preorder(root); break;

            case 3:
            printf("\nPost-order: ");
            postorder(root); break;
```

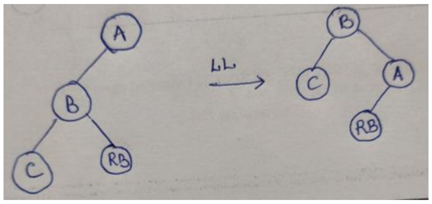
```
        case 4:
        printf("Enter search element: ");
        scanf("%d", &key);
        search(root, key); break;

        case 5: exit(0);

        default: printf("Invalid choice!");
      }
    }
}
```

Module -4 – AVL Tree

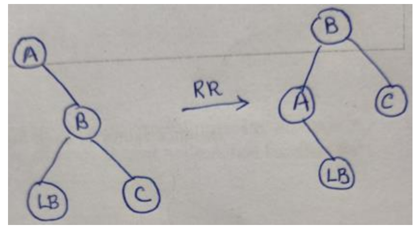
AVL Tree Implementation

Implementation 1(as done in class):

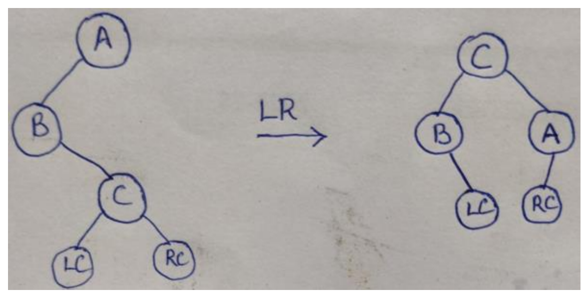
```
/*   Program - AVL Tree Implementation */
#include <stdio.h>
#include <stdlib.h>
struct AVL
{
   int key;
   struct AVL *left;
   struct AVL *right;
   int height;
};
typedef struct AVL node;
int getHeight(node *n)
{
   if(n==NULL)
      return 0;
   return n->height;
}
node *createNode(int key)
{
   node* n = (node *) malloc(sizeof(node));
   n->key = key;
   n->left = NULL;
   n->right = NULL;
   n->height = 1;
   return n;
}
int max (int a, int b)
{
   return (a>b)?a:b;
}
int getBalanceFactor(node * n)
{
   if(n==NULL)
      return 0;
   return getHeight(n->left) - getHeight(n->right);
}
node* leftRotate(node* A)
{
   node* B = A->left;
   node* RB = B->right;
   B->right = A;
   A->left = RB;
```



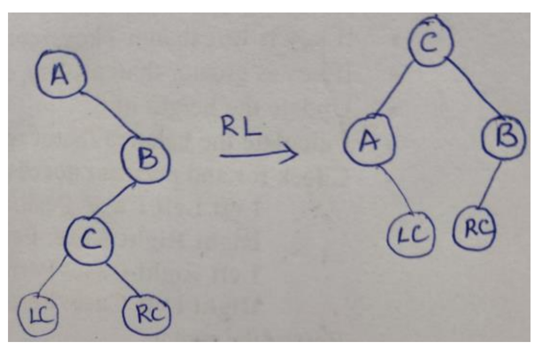
```
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   return B;
}
```




```
node* rightRotate(node* A)
{
   node* B = A->right;
   node* LB = B->left;
   B->left = A;
   A->right = LB;
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   return B;
}
```




```
node* leftrightRotate(node* A)
{
   node* B = A->left;
   node* C = B->right;
   node* LC = C->left;
   node* RC = C->right;
   C->left = B;
   C->right = A;
   B->right = LC;
   A->left = RC;
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   C->height = max(getHeight(C->right), getHeight(C->left)) + 1;
   return C;
}
```

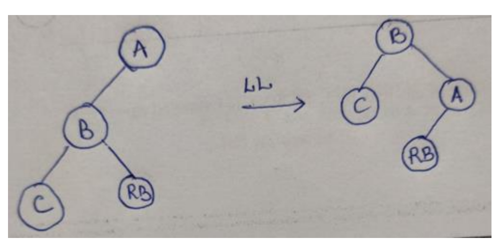


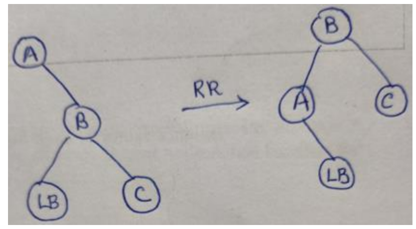

```
node* rightleftRotate(node* A)
{
   node* B = A->right;
   node* C = B->left;
   node* LC = C->left;
   node* RC = C->right;
   C->left = A;
   C->right = B;
   A->right = LC;
   B->left = RC;
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   C->height = max(getHeight(C->right), getHeight(C->left)) + 1;
   return C;
}
node *insert(node* n, int key)
{       int bf;
        if (n == NULL)
```

```
            return  createNode(key);
        if (key < n->key)
            n->left  = insert(n->left, key);
        else if (key > n->key)
            n->right = insert(n->right, key);
        n->height = 1 + max(getHeight(n->left), getHeight(n->right));
        bf = getBalanceFactor(n);
        // Left Left Case
        if(bf>1 && key < n->left->key)
            return leftRotate(n);
        // Right Right Case
        if(bf<-1 && key > n->right->key)
            return rightRotate(n);
        // Left Right Case
        if(bf>1 && key > n->left->key)
            return leftrightRotate(n);
        // Right Left Case
        if(bf<-1 && key < n->right->key)
            return rightleftRotate(n);
        return n;
}
void printtree(node *root, int space,int n)
{
        int i;
        if (root == NULL)
            return;
        space +=n;
        printtree(root->right, space,n);
        printf("\n");
        for (i =n; i < space; i++)
            printf(" ");
        printf("(%d)\n", root->key,space);
        printtree(root->left, space,n);
}
int main()
{
        node * root = NULL;
        int n,i,key;
        printf("enter the number of nodes\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("enter the key\n");
            scanf("%d",&key);
            root = insert(root, key);
        }
        printtree(root, 0, n) ;
        return 0;
}
```

Implementation 2:

```
/*   Program - AVL Tree Implementation */
#include <stdio.h>
#include <stdlib.h>
struct AVL
{
   int key;
   struct AVL *left;
   struct AVL *right;
   int height;
};
typedef struct AVL node;
int getHeight(node *n)
{
   if(n==NULL)
      return 0;
   return n->height;
}
node *createNode(int key)
{
   node* n = (node *) malloc(sizeof(node));
   n->key = key;
   n->left = NULL;
   n->right = NULL;
   n->height = 1;
   return n;
}
int max (int a, int b)
{
   return (a>b)?a:b;
}
int getBalanceFactor(node * n)
{
   if(n==NULL)
        return 0;
   return getHeight(n->left) - getHeight(n->right);
}
node* leftRotate(node* A)
{
   node* B = A->left;
   node* RB = B->right;
   B->right = A;
   A->left = RB;
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   return B;
}
```






```
node* rightRotate(node* A)
{
   node* B = A->right;
   node* LB = B->left;
   B->left = A;
   A->right = LB;
   A->height = max(getHeight(A->right), getHeight(A->left)) + 1;
   B->height = max(getHeight(B->right), getHeight(B->left)) + 1;
   return B;
}
node *insert(node* n, int key)
{
   int bf;
        if (n == NULL)
                return  createNode(key);
        if (key < n->key)
                n->left  = insert(n->left, key);
        else if (key > n->key)
                n->right = insert(n->right, key);
        n->height = 1 + max(getHeight(n->left), getHeight(n->right));
        bf = getBalanceFactor(n);
        // Left Left Case
        if(bf>1 && key < n->left->key)
                return leftRotate(n);
        // Right Right Case
        if(bf<-1 && key > n->right->key)
                return rightRotate(n);
        // Left Right Case
        if(bf>1 && key > n->left->key)
        {
                n->left = rightRotate(n->left);
                return leftRotate(n);
        }
        // Right Left Case
        if(bf<-1 && key < n->right->key)
        {
                n->right = leftRotate(n->right);
                return rightRotate(n);
        }
        return n;
}
void printtree(node *root, int space,int n)
{
int i;
        if (root == NULL)
                return;
        space +=n;
        printtree(root->right, space,n);
        printf("\n");
```

```
        for (i =n; i < space; i++)
                printf(" ");
        printf("(%d)\n", root->key,space);
        printtree(root->left, space,n);
}
int main()
{
        node * root = NULL;
        int n,i,key;
        printf("enter the number of nodes\n");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("enter the key\n");
                scanf("%d",&key);
                root = insert(root, key);
        }
        printtree(root, 0, n) ;
        return 0;
}
```

Ex : 8

Title: Design, Develop and Implement a Program in C to create N router connections and display the routers by removal of which will disconnect the network completely.

Problem Description: create a data structure to represent the routers and its connections between the routers and display the router numbers on removal which it disconnects the network completely.

Method: choose the best data structure to represent the routers and a function to find the articulation point.

Theory Reference: Module 5

Explanation:

- **Articulation Point**: A vertex that, when removed, disconnects the graph.
- **DFS**: A traversal technique used to explore nodes and edges of a graph. The `dfnlow` function performs the main DFS operation while tracking discovery and low values.
- **dfn[u]**: Discovery time of vertex `u`.
- **low[u]**: The lowest discovery time reachable from `u`.
- **parent[u]**: The parent vertex of `u` in the DFS tree.
- **Adjacency Matrix**: The graph is represented as an adjacency matrix `adj`, where `adj[u][v]` is `1` if there's an edge between routers `u` and `v`.

Following Initializations are done:

- Arrays are initialized to keep track of:
  - Discovery times (`dfn`)
  - Low values (`low`)
  - Parent vertices (`parent`)
  - Visited vertices (`visited`)
  - Articulation points (`articulation`)

Algorithm:

Step 1: main Function - To initialize the graph and collect user input for the number of vertices and edges. Update the adjacency matrix to mark the edge between u and v as present (set adj[u][v] and adj[v][u] to 1).

Step 2: findArticulationPoints Function - To find articulation points in the graph using Depth-First Search (DFS).

1. For i=0 to V−1:
    - Initialize `parent[i]` to -1 (indicating no parent).
    - Initialize `visited[i]` to 0 (indicating not visited).
    - Initialize `articulation[i]` to 0 (indicating not an articulation point).
2. For i=0 to V−1:
    - If `visited[i]` is 0, call `dfnlow(i, adj, V)` to perform DFS from vertex i.
3. Print the articulation points:
    - For i=0 to V−1:
        - If `articulation[i]` is 1, print "Router i".

Step 3. dfnlow Function - To perform DFS and update dfn and low values, identifying articulation points.

1. Initialize `children` to 0 (to count child vertices of the current vertex u).
2. Mark `u` as visited: set `visited[u]` to 1.
3. Set `dfn[u]` and `low[u]` to `time` (incremented by 1).
4. For each vertex v from 0 to V−1:
    - If there is an edge from u to v (`adj[u][v] == 1`):
        - If v is not visited:
            1. Increment `children` by 1.
            2. Set `parent[v]` to u.
            3. Recursively call `dfnlow(v, adj, V)`.
            4. Update `low[u]`: set `low[u]` to the minimum of `low[u]` and `low[v]`.
            5. Check if u is an articulation point:
                - If u is the root of the DFS tree (i.e., `parent[u] == -1`) and has two or more children, mark u as an articulation point.
                - If u is not the root and `low[v] >= dfn[u]`, mark u as an articulation point.
        - Else if v is visited and v is not the parent of u:
            - Update `low[u]`: set `low[u]` to the minimum of `low[u]` and `dfn[v]`.

Program:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int dfn[MAX],low[MAX],parent[MAX],time = 0;

int visited[MAX],articulation[MAX];

void dfnlow(int u, int adj[MAX][MAX], int V)

{

        int children = 0,v;

        visited[u] = 1;

        dfn[u] = low[u] = ++time;

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

        {

                if (adj[u][v])

                {

                        // If there's an edge between u and v

                        if (!visited[v])

                        {

                                children++;

                                parent[v] = u;

                                dfnlow(v, adj, V);
```

```
                    // Check if the subtree rooted at v has a connection back to one of the ancestors of u
                    low[u] = (low[u] < low[v]) ? low[u] : low[v];
                    // u is an articulation point in the following cases:
                    // Case 1: u is the root of DFS tree and has two or more children.
                    if (parent[u] == -1 && children > 1)
                    articulation[u] = 1;
                    // Case 2: u is not the root and low value of one of its children is more than dfn value of u.
                    if (parent[u] != -1 && low[v] >= dfn[u])
                    articulation[u] = 1;
                }
                else if (v != parent[u])
                {
                    // Update low value of u for parent function calls.
                    low[u] = (low[u] < dfn[v]) ? low[u] : dfn[v];
                }
            }
        }
}
void findArticulationPoints(int adj[MAX][MAX], int V)
{
    int i;
    for (i = 0; i < V; i++)
```

```
    {
        parent[i] = -1;
        visited[i] = 0;
        articulation[i] = 0;
    }
    for (i = 0; i < V; i++)
    {
        if (!visited[i])
        dfnlow(i, adj, V);
    }
    printf("Articulation points are:\n");
    for (i = 0; i < V; i++)
    {
        if (articulation[i])
            printf("Router %d\n", i);
    }
}
int main()
{
    int V,E,i,adj[MAX][MAX] = {0},u,v;
    printf("Enter the number of routers (vertices): ");
```

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

    printf("Enter the number of connections (edges): ");

    scanf("%d",&E);

    printf("Enter the connections (u v):\n");

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

    {

        scanf("%d %d", &u,&v);

        adj[u][v] = 1;

        adj[v][u] = 1;

    }

    findArticulationPoints(adj, V);

    return 0;

}
```

Output:

```
Enter the number of routers (vertices): 10
Enter the number of connections (edges): 11
Enter the connections (u v):
0 1
1 2
1 3
2 4
3 4
3 5
5 7
5 6
6 7
7 8
7 9
Articulation points are:
Router 1
Router 3
Router 5
Router 7
```

Ex : 9

Title: Design, Develop and Implement a Program in C to perform polynomial addition, output the polynomial result and evaluate the resultant polynomial.

Problem Description: Read two polynomials and perform addition. Evaluate the resultant polynomial by providing x value.

Method: A Polynomial is a collection of terms, and each term consist of coefficient, variable and exponent. Represent it using structures. Consider the polynomial with one variable x, Input the vale of x to evaluate the resultant polynomial after addition.

Theory Reference: Module 1

<u>Explanation:</u>

The program defines a polynomial structure and implements functions to read, add, print, and evaluate polynomials.

<u>Structure definition</u>

```
struct polynomial
{
        int coeff; // Coefficient of the polynomial term
        int expo; // Exponent of the polynomial term
};
```

<u>Algorithm:</u>

Step 1: readpolynomial Function

Reads polynomial terms from user input.

Step 2: addpolynomial Function

1. Initialize three indices: i for p1, j for p2, and k for p3.
2. While both i and j are within bounds:
    - If the exponents of the current terms in both polynomials are equal:
        - Add their coefficients and store in p3.
        - Increment i, j, and k.
    - If the exponent of p1 is greater than that of p2:

        - Copy the term from p1 to p3.
        - Increment i and k.
    - If the exponent of p2 is greater:
        - Copy the term from p2 to p3.
        - Increment j and k.
3. After one of the polynomials is fully processed, add the remaining terms from the other polynomial.
4. Return the number of terms in the resultant polynomial.

Step 3: printpolynomial Function

1. Loop from 0 to n−2:
    - Print each term in the format coeff(x^expo) followed by a plus sign.
2. Print the last term without a plus sign.

Step 4: evaluate Function

1. Initialize sum to 0.
2. Prompt the user to enter the value of x.
3. Loop through each term in the polynomial:
    - Calculate the term's contribution as coeff * (x ^ expo).
    - Add this contribution to sum.
4. Print the result.

Step 5. main Function

1. Create three arrays to store the two input polynomials and the result.
2. Read the number of terms and coefficients/exponents for the first polynomial.
3. Read the number of terms and coefficients/exponents for the second polynomial.
4. Print both input polynomials.
5. Call `addpolynomial` to get the resultant polynomial.
6. Print the resultant polynomial.
7. Call `evaluate` to compute the value of the resultant polynomial at a user-defined x.

Program:

```
#include<stdio.h>

#include<math.h>

struct polynomial

{

   int coeff;
```

```
    int expo;
};
typedef struct polynomial poly;
void readpolynomial(poly p[], int n);
int addpolynomial(poly p1[], poly p2[], int n1, int n2, poly p3[]);
void printpolynomial(poly p[], int n);
void evaluate(poly p[],int n);
void readpolynomial(poly p[], int n)
{
    int i;
   for (i = 0; i < n; i++)
{
        printf("\n Enter the Coefficient and the Exponent");
        scanf("%d%d",&p[i].coeff,&p[i].expo);
    }
}
int addpolynomial(poly p1[], poly p2[], int n1, int n2, poly p3[])
{
    int i=0, j=0, k=0;
    while (i < n1 && j <n2)
   {
            if (p1[i].expo == p2[j].expo)
            {
            p3[k].coeff = p1[i].coeff + p2[j].coeff;
                        p3[k].expo = p1[i].expo;
                        i++;
                        j++;
                        k++;
        }
```

```
       else if (p1[i].expo > p2[j].expo)
       {
                  p3[k]=p1[i];
                  i++;
                  k++;
        }
        else
           {
                  p3[k] = p2[j];
                  j++;
                  k++;
        }
    }
    while (i < n1)
    {
        p3[k] = p1[i];
        i++;
        k++;
     }
   while (j < n2)
  {
        p3[k] = p2[j];
          j++;
        k++;
    }
    return (k);
}
void printpolynomial(poly p[], int n)
{
```

```
    int i;
    for (i = 0; i < n - 1; i++)
        printf("%d(x^%d)+", p[i].coeff, p[i].expo);
    printf("%d(x^%d)", p[n - 1].coeff, p[n - 1].expo);
}
void evaluate(poly p[],int n)
{
        int sum=0,i,x;
        printf("\n enter the value of x:");
        scanf("%d",&x);
        for (i = 0; i < n; i++)
                sum=sum+p[i].coeff*pow(x,p[i].expo);
        printf("\n Result=%d\n",sum);
}
int main()
{
        int n1, n2, n3;
        poly p1[10],p2[10],p3[10];
        printf("enter number of terms in first polynomial");
        scanf("%d",&n1);
        readpolynomial(p1,n1);
        printf("enter number of terms in second polynomial");
        scanf("%d",&n2);
        readpolynomial(p2,n2);
        printf(" \n First polynomial : ");
        printpolynomial(p1, n1);
        printf(" \n Second polynomial : ");
        printpolynomial(p2, n2);
        n3 = addpolynomial(p1,p2,n1,n2,p3);
```

```
    printf(" \n Resultant polynomial after addition : ");
    printpolynomial(p3, n3);
    evaluate(p3, n3);
    return 0;
}
```

Output

```
enter number of terms in first polynomial3

 Enter the Coefficient and the Exponent2 2

 Enter the Coefficient and the Exponent3 1

 Enter the Coefficient and the Exponent5
0
enter number of terms in second polynomial3

 Enter the Coefficient and the Exponent2 2

 Enter the Coefficient and the Exponent3 1

 Enter the Coefficient and the Exponent5 0

 First polynomial : 2(x^2)+3(x^1)+5(x^0)
 Second polynomial : 2(x^2)+3(x^1)+5(x^0)
 Resultant polynomial after addition : 4(x^2)+6(x^1)+10(x^0)
 enter the value of x:2

 Result=38

--------------------------------
Process exited after 46.87 seconds with return value 0
Press any key to continue . . .
```

Ex : 9b

Title: Design, Develop and Implement a Program in C to read a sparse matrix to search a particular value and to display the transpose of the matrix in its triplet format.

Problem Description: To manage sparse matrices using a triplet representation and finding its transpose.

Method: The Program must allow the user to read a sparse matrix, display it in triplet format, search for specific values, and transpose the matrix.

Theory Reference: Module 1

Explanation:

Structure Definition

```
struct sparse
{
    int row; // Row index of the non-zero element
    int col; // Column index of the non-zero element
    int val; // Value of the non-zero element
} s[10];
```

This structure represents a non-zero element of a sparse matrix, storing its row index, column index, and value. An array s of this structure is used to store the matrix.

Algorithm:

Step 1: readsparsematrix Function

Reads a sparse matrix from user input and stores its non-zero elements in triplet form.

Step 2: triplet Function

Displays the triplet representation of the sparse matrix.

Step 3: search Function

Searches for a specified key (value) in the sparse matrix.

1. Prompt the user to enter a value `key` to search for.
2. Initialize a flag `found` to 0.
3. Loop from `1` to `s[0].val`:
   - If `s[i].val` matches the `key`:
     - Print the row and column of the found element.
     - Set `found` to 1 and break the loop.
4. If `found` is still 0 after the loop, print "element not found."

Step 4: transpose Function

1. Create a temporary structure array `trans` to store the transposed matrix.
2. Set the dimensions of the transposed matrix:
   - `trans[0].row` = `s[0].col` (new number of rows)
   - `trans[0].col` = `s[0].row` (new number of columns)
   - `trans[0].val` = `s[0].val` (same number of non-zero elements)
3. Initialize a variable `k` to 1 to index into the transposed array.
4. Loop over each column iii of the original matrix:
   - For each non-zero element in `s`:
     - If the column index matches iii:
       - Assign the transposed values:
         - `trans[k].row` = `s[j].col`
         - `trans[k].col` = `s[j].row`
         - `trans[k].val` = `s[j].val`
       - Increment `k`.
5. Copy the transposed matrix back to `s` (overwriting the original).
6. Call `triplet()` to display the transposed matrix.

Step 5: main Function

1. Call `readsparsematrix()` to read the sparse matrix from user input.
2. Call `triplet()` to display the original sparse matrix.
3. Call `search()` to allow the user to search for a specific value in the matrix.
4. Call `transpose()` to compute and display the transposed matrix.

Program:

```
#include <stdio.h>
struct sparse
{
    int row;
    int col;
    int val;
} s[10];
void readsparsematrix()
{
    int i,j,r,c,ele,pos=0;
    printf("Enter rows and cols:\n");
    scanf("%d%d", &r,&c);
    for (i = 0; i < r; i++)
{
        for (j = 0; j < c; j++)
{
                scanf("%d", &ele);
                if (ele != 0)
{
                        pos++;
                        s[pos].row = i;
                        s[pos].col = j;
                        s[pos].val = ele;
                }
            }
        }
    s[0].row = r;
    s[0].col = c;
```

```
    s[0].val = pos;
}
void triplet()
{
            int i;
            printf("\nTriplet representation:\n");
            printf("Row  Col  Value\n");
            for (i = 0; i <= s[0].val; i++)
{
        printf("%d    %d    %d\n", s[i].row, s[i].col, s[i].val);
            }
}
void search()
{
            int found=0,key,i;
            printf("Enter key:\n");
            scanf("%d", &key);
            for (i = 1; i <= s[0].val; i++)
{
        if (s[i].val == key)
   {
            printf("%d    %d    \n", s[i].row, s[i].col);
            found=1;
            break;
          }
    }
    if(found==0)
            printf("\n element not  found:");
```

```
}
void transpose()
{
   int i,j;
   struct sparse trans[10];
   trans[0].row = s[0].col;
   trans[0].col = s[0].row;
   trans[0].val = s[0].val;
   int k = 1;
   for (i = 0; i < s[0].col; i++)
   {
      for (j = 1; j <= s[0].val; j++)
       {
         if (s[j].col == i)
          {
             trans[k].row = s[j].col;
             trans[k].col = s[j].row;
             trans[k].val = s[j].val;
             k++;
          }
       }
    }
   for (i = 0; i <= s[0].val; i++)
  {
      s[i] = trans[i];
   }
   triplet();
}
```

```
int main()
{
        readsparsematrix();
        triplet();
        search();
        transpose();
        return 0;
}
```

<u>Output:</u>

```
Enter rows and cols:
4 5
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

Triplet representation:
Row   Col   Value
4     5     6
0     2     3
0     4     4
1     2     5
1     3     7
3     1     2
3     2     6
Enter key:
7
1     3

Triplet representation:
Row   Col   Value
5     4     6
1     3     2
2     0     3
2     1     5
2     3     6
3     1     7
4     0     4

--------------------------------
Process exited after 87.55 seconds with return value 0
Press any key to continue . . .
```

Ex : 10

Title: Design and develop a Program in C that uses Hash function H: K ->L as H(K)=K mod m and implement hashing technique to map a given key K to the address space L. Resolvethe collision (if any).

Problem Description: Given a set K of Keys (4-digit) which uniquely determines the records in file F. Assume that file F ismaintained in memory by a Hash Table (HT) of m memory locations with L as the set of memory addresses (2-digit) of locations in HT. Let the keys in K and addresses in L are Integers.

Method: Ensure that the program generates the 4-digit key randomly and generates the L using hash function, demonstratesif there is a collision and its resolution by using linear probing.

Theory Reference: Module 5

Explanation:

1. **Linear Probing**: Collisions are resolved by checking the next available index (index1 = (index1 + 1) % m;), which continues until an empty slot (-1) is found.
2. **Memory Allocation**: The hash table (a) is dynamically allocated using malloc.
3. **Handling Table Full Condition**: If the hash table is full, it stops inserting further keys and reports the issue.

Algorithm:

Step 1: Initialize the Hash Table

1. **Create an array** `a` of size `m` (in this case, `m = 20`), and initialize all elements to `-1` to represent empty slots.
2. Set `count = 0` to keep track of the number of elements inserted into the hash table.

Step 2: Input the Number of Keys and Validate

1. Ask the user for the number of keys to insert into the hash table.
2. **Validate the input**:
    - If the number of keys `n` is greater than `m`, print an error message and terminate the program (since the hash table cannot accommodate more keys than its size).

Step 3: Input the Keys

1. Ask the user to input `n` keys (each key is a 4-digit integer).
2. Store the keys in an array `key[20]`.

Step 4: Insert Keys into the Hash Table Using Linear Probing

For each key in the array `key[]`:

1. **Compute the index** where the key should be inserted:

  - Use the hash function: `index1 = key % m` (this gives an index in the range `0` to `m-1`).
2. **Linear Probing** to resolve collisions:
   - Check if the slot at index `index1` is occupied (`a[index1] != -1`):
     - If occupied, move to the next slot: `index1 = (index1 + 1) % m`.
     - Repeat this process until an empty slot is found (i.e., `a[index1] == -1`).
3. **Insert the key**:
   - Once an empty slot is found, insert the key at that index: `a[index1] = key`.
   - Increment the `count` to indicate that a key has been successfully inserted.
4. **Check if the hash table is full**:
   - If `count == m`, print a message indicating the table is full and stop inserting further keys.

Step 5: Display the Hash Table

1. Traverse the array `a[]` and print the keys stored in the hash table. For each index:
   - If the slot is empty (`a[i] == -1`), print "Empty".
   - Otherwise, print the key stored at that index.

Program

```
#include<stdio.h>

#include<stdlib.h>

#define m 20

int key[20],n;

int *a,index1;

int count = 0;

void linearprobing(int key)

{

   index1 = key % m;

   while (a[index1] != -1)
```

```
    {
        index1 = (index1 + 1) % m;// collision resolved using linear probing
    }
    a[index1] = key;
    count++;
}
void display()
{
    int i;
    if (count == 0)
    {
        printf("\nHash Table is empty");
        return;
    }
    printf("\nHash Table contents are:\n ");
    for (i = 0; i < m; i++)
        printf("\n T[%d] --> %d ", i, a[i]);
}
void main()
{
```

```
    int i;
    printf("\nEnter the number of four digit key values: ");
    scanf("%d",&n);
    a = (int *) malloc(m * sizeof(int));
    for (i = 0; i < m; i++)
        a[i] = -1;
    printf("\nEnter the four digit key values (K):\n ");
    for (i = 0; i < n; i++)
        scanf("%4d", &key[i]);
    for (i = 0; i < n; i++)
    {
        if (count == m)
        {
            printf("\n-----Hash table is full. Cannot insert the record %d key-----", i + 1);
            break;
        }
        linearprobing(key[i]);
    }
    display();
}
```

Output

```
Enter the number of four digit key values: 10

Enter the four digit key values (K):
 4020
  4560
  9908
  6785
  0423
  7890
  6547
  3342
  9043
  6754

Hash Table contents are:

 T[0] --> 4020
 T[1] --> 4560
 T[2] --> 3342
 T[3] --> 423
 T[4] --> 9043
 T[5] --> 6785
 T[6] --> -1
 T[7] --> 6547
 T[8] --> 9908
 T[9] --> -1
 T[10] --> 7890
 T[11] --> -1
 T[12] --> -1
 T[13] --> -1
 T[14] --> 6754
 T[15] --> -1
 T[16] --> -1
 T[17] --> -1
 T[18] --> -1
 T[19] --> -1
--------------------------------
Process exited after 16.61 seconds with return value 15
Press any key to continue . . .
```