1. Write a C program to implement iterative and recursive binary search algorithms. Define and use a macro to compare two integers in your program.

Program

```c
#include <stdio.h>

#define COMPARE(a, b) ((a) - (b))   // Macro to compare two integers

// Recursive Binary Search with Trace
int recursiveBinarySearch(int arr[], int left, int right, int key)
{
   if (left <= right)
{
      int mid = left + (right - left) / 2;
      int cmp = COMPARE(arr[mid], key);

      // Trace
      printf("Recursive Step -> left: %d, right: %d, mid: %d, arr[mid]: %d\n",
           left, right, mid, arr[mid]);

      if (cmp == 0)
         return mid;  // Key found
      else if (cmp > 0)
         return recursiveBinarySearch(arr, left, mid - 1, key); // Search left half
      else
         return recursiveBinarySearch(arr, mid + 1, right, key); // Search right half
   }
   return -1; // Key not found
}

// Iterative Binary Search with Trace
int iterativeBinarySearch(int arr[], int n, int key)
{
   int left = 0, right = n - 1;

   while (left <= right) {
      int mid = left + (right - left) / 2;
      int cmp = COMPARE(arr[mid], key);
```

```
        // Trace
        printf("Iterative Step -> left: %d, right: %d, mid: %d, arr[mid]: %d\n",
               left, right, mid, arr[mid]);

        if (cmp == 0)
            return mid;  // Key found
        else if (cmp > 0)
            right = mid - 1; // Search left half
        else
            left = mid + 1; // Search right half
    }
    return -1; // Key not found
}

int main() {
    int arr[] = {2, 5, 8, 12, 16, 23, 38, 45, 56, 72, 91};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key;

    printf("Enter element to search: ");
    scanf("%d", &key);

    // Iterative Binary Search
    printf("\n--- Iterative Binary Search Trace ---\n");
    int result_iter = iterativeBinarySearch(arr, n, key);
    if (result_iter != -1)
        printf("Iterative: Element %d found at index %d\n", key, result_iter);
    else
        printf("Iterative: Element %d not found\n", key);

    // Recursive Binary Search
    printf("\n--- Recursive Binary Search Trace ---\n");
    int result_rec = recursiveBinarySearch(arr, 0, n - 1, key);
    if (result_rec != -1)
        printf("Recursive: Element %d found at index %d\n", key, result_rec);
    else
        printf("Recursive: Element %d not found\n", key);

    return 0;
```

}

OUTPUT:
Enter element to search: 23

--- Iterative Binary Search Trace ---
Iterative Step -> left: 0, right: 10, mid: 5, arr[mid]: 23
Iterative: Element 23 found at index 5

--- Recursive Binary Search Trace ---
Recursive Step -> left: 0, right: 10, mid: 5, arr[mid]: 23
Recursive: Element 23 found at index 5

2. **Write a C program to find the fast transpose of a sparse matrix.**

Program:

```
#include <stdio.h>
#define MAX 100

// Function to read sparse matrix in 3-tuple form
void readSparse(int sparse[MAX][3])
 {
   int rows, cols, nonZero, i;
   printf("Enter number of rows, columns and non-zero elements: ");
   scanf("%d %d %d", &rows, &cols, &nonZero);

   // Store metadata
   sparse[0][0] = rows;
   sparse[0][1] = cols;
   sparse[0][2] = nonZero;

   printf("Enter row, column and value for each non-zero element:\n");
   for (i = 1; i <= nonZero; i++)
 {
      scanf("%d %d %d", &sparse[i][0], &sparse[i][1], &sparse[i][2]);
   }
}

// Function to print sparse matrix in 3-tuple form
void printSparse(int sparse[MAX][3])
{
   int i, nonZero = sparse[0][2];
   printf("\nRow\tCol\tVal\n");
   for (i = 0; i <= nonZero; i++)
 {
      printf("%d\t%d\t%d\n", sparse[i][0], sparse[i][1], sparse[i][2]);
   }
}

// Function for Fast Transpose
void fastTranspose(int sparse[MAX][3], int trans[MAX][3])
 {
```

```
    int rowTerms[MAX], startingPos[MAX];
    int rows, cols, nonZero;
    int i, j;

    rows = sparse[0][0];
    cols = sparse[0][1];
    nonZero = sparse[0][2];

    // Metadata for transpose
    trans[0][0] = cols;
    trans[0][1] = rows;
    trans[0][2] = nonZero;

    // Initialize rowTerms
    for (i = 0; i < cols; i++)
        rowTerms[i] = 0;

    // Count number of elements in each column (of original matrix)
    for (i = 1; i <= nonZero; i++)
        rowTerms[sparse[i][1]]++;

    // Compute starting positions for each row in transposed matrix
    startingPos[0] = 1;
    for (i = 1; i < cols; i++)
        startingPos[i] = startingPos[i - 1] + rowTerms[i - 1];

    // Place elements directly into transposed matrix
    for (i = 1; i <= nonZero; i++)
  {
        j = startingPos[sparse[i][1]]++;
        trans[j][0] = sparse[i][1];
        trans[j][1] = sparse[i][0];
        trans[j][2] = sparse[i][2];
    }
}

int main()
{
    int sparse[MAX][3], trans[MAX][3];
```

```
    readSparse(sparse);

    printf("\nOriginal Sparse Matrix (3-tuple form):");
    printSparse(sparse);

    fastTranspose(sparse, trans);

    printf("\nFast Transposed Sparse Matrix (3-tuple form):");
    printSparse(trans);

    return 0;
}
```

Input:

Enter number of rows, columns and non-zero elements: 3 3 4
Enter row, column and value for each non-zero element:
0 0 5
0 2 8
1 1 3
2 0 6

This means the matrix is:

5 0 8
0 3 0
6 0 0

Output:

Original Sparse Matrix (3-tuple form):

Row	Col	Val
3	3	4
0	0	5
0	2	8
1	1	3
2	0	6

Fast Transposed Sparse Matrix (3-tuple form):

Row	Col	Val
3	3	4

0	0	5
0	2	6
1	1	3
2	0	8

3. Write a C program to implement a circular queue using dynamically allocated array and perform the following operations on it.

(i)Insert an item (ii) Delete an item (iii) Display a circular queue

<u>Program:</u>

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

typedef struct {
   int *arr;     // dynamic array
   int front, rear, size, capacity;
} CircularQueue;

// Function to create a circular queue
CircularQueue* createQueue(int capacity) {
   CircularQueue *q = (CircularQueue*)malloc(sizeof(CircularQueue));
   q->capacity = capacity;
   q->front = q->rear = -1;
   q->arr = (int*)malloc(capacity * sizeof(int));
   return q;
}

// Check if queue is full
int isFull(CircularQueue *q) {
   return ((q->rear + 1) % q->capacity == q->front);
}

// Check if queue is empty
int isEmpty(CircularQueue *q) {
   return (q->front == -1);
}

// Insert an item
void enqueue(CircularQueue *q, int item) {
   if (isFull(q)) {
      printf("Queue is FULL! Cannot insert %d\n", item);
      return;
   }
   if (q->front == -1)   // first insertion
      q->front = 0;
   q->rear = (q->rear + 1) % q->capacity;
   q->arr[q->rear] = item;
   printf("Inserted: %d\n", item);
}
```

```c
// Delete an item
void dequeue(CircularQueue *q) {
   if (isEmpty(q)) {
      printf("Queue is EMPTY! Cannot delete\n");
      return;
   }
   printf("Deleted: %d\n", q->arr[q->front]);
   if (q->front == q->rear) {   // only one element
      q->front = q->rear = -1;
   } else {
      q->front = (q->front + 1) % q->capacity;
   }
}

// Display the circular queue
void display(CircularQueue *q) {
   if (isEmpty(q)) {
      printf("Queue is EMPTY!\n");
      return;
   }
   printf("Circular Queue: ");
   int i = q->front;
   while (1) {
      printf("%d ", q->arr[i]);
      if (i == q->rear) break;
      i = (i + 1) % q->capacity;
   }
   printf("\n");
}

// Driver program
int main() {
   int capacity, choice, item;
   printf("Enter size of Circular Queue: ");
   scanf("%d", &capacity);

   CircularQueue *q = createQueue(capacity);

   while (1) {
      printf("\n--- Circular Queue Menu ---\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
```

```
                printf("Enter element to insert: ");
                scanf("%d", &item);
                enqueue(q, item);
                break;
            case 2:
                dequeue(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                free(q->arr);
                free(q);
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Output:

```
Enter size of Circular Queue: 5

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 10
Inserted: 10

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 20
Inserted: 20

--- Circular Queue Menu ---
1. Insert
2. Delete
```

3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 30
Inserted: 30

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Circular Queue: 10 20 30

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted: 10

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Circular Queue: 20 30

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 40
Inserted: 40

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 50

Inserted: 50

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 60
Inserted: 60

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Circular Queue: 20 30 40 50 60

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1
Enter element to insert: 70
Queue is FULL! Cannot insert 70

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2
Deleted: 20

--- Circular Queue Menu ---
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3
Circular Queue: 30 40 50 60

4. Design, Develop and Implement a Program in C for the following Stack Applications

a. Evaluation of Suffix expression with single digit operands and operators: +, -, *, /, %, ^

b. Solving Tower of Hanoi problem with n disks.

<u>Program:</u>

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

#define MAX 100

// ---------------- Stack for Postfix Evaluation ----------------
typedef struct {
   int arr[MAX];
   int top;
} Stack;

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

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

int isFull(Stack *s) {
   return (s->top == MAX - 1);
}

void push(Stack *s, int val) {
   if (isFull(s)) {
      printf("Stack Overflow!\n");
      return;
   }
   s->arr[++s->top] = val;
}

int pop(Stack *s) {
   if (isEmpty(s)) {
      printf("Stack Underflow!\n");
      return -1;
   }
   return s->arr[s->top--];
}
```

```c
// Function to evaluate postfix expression
int evaluatePostfix(char exp[]) {
    Stack s;
    initStack(&s);

    for (int i = 0; exp[i] != '\0'; i++) {
        char ch = exp[i];

        if (isdigit(ch)) {
            push(&s, ch - '0');   // convert char to int
        } else {
            int val2 = pop(&s);
            int val1 = pop(&s);
            switch (ch) {
                case '+': push(&s, val1 + val2); break;
                case '-': push(&s, val1 - val2); break;
                case '*': push(&s, val1 * val2); break;
                case '/': push(&s, val1 / val2); break;
                case '%': push(&s, val1 % val2); break;
                case '^': push(&s, (int)pow(val1, val2)); break;
                default:
                    printf("Invalid Operator %c\n", ch);
                    exit(1);
            }
        }
    }
    return pop(&s);
}

// ---------------- Tower of Hanoi ----------------
void towerOfHanoi(int n, char src, char aux, char dest) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", src, dest);
        return;
    }
    towerOfHanoi(n - 1, src, dest, aux);
    printf("Move disk %d from %c to %c\n", n, src, dest);
    towerOfHanoi(n - 1, aux, src, dest);
}

// ---------------- Main Program ----------------
int main() {
    int choice, n;
    char exp[MAX];
```

```
    while (1) {
        printf("\n--- Stack Applications Menu ---\n");
        printf("1. Evaluate Postfix Expression\n");
        printf("2. Tower of Hanoi\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter postfix expression: ");
                scanf("%s", exp);
                printf("Result = %d\n", evaluatePostfix(exp));
                break;

            case 2:
                printf("Enter number of disks: ");
                scanf("%d", &n);
                printf("The sequence of moves:\n");
                towerOfHanoi(n, 'A', 'B', 'C');
                break;

            case 3:
                exit(0);

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

Output:

(a) **Postfix Evaluation**

```
--- Stack Applications Menu ---
1. Evaluate Postfix Expression
2. Tower of Hanoi
3. Exit
Enter your choice: 1
Enter postfix expression: 23*54*+9-
Result = 17
```

(b) **Tower of Hanoi (n=3)**

```
--- Stack Applications Menu ---
1. Evaluate Postfix Expression
2. Tower of Hanoi
```

```
3. Exit
Enter your choice: 2
Enter number of disks: 3
The sequence of moves:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

© Recursive Tree (n = 3)

```
Move 3 disks from A → C using B
|
|-- Move 2 disks from A → B using C
|   |
|   |-- Move 1 disk from A → C
|   |-- Move disk 2 from A → B
|   |-- Move 1 disk from C → B
|
|-- Move disk 3 from A → C
|
|-- Move 2 disks from B → C using A
    |
    |-- Move 1 disk from B → A
    |-- Move disk 2 from B → C
    |-- Move 1 disk from A → C
```

5. Write a C program to implement a doubly linked circular list with a header node and perform the following operations on it.

(i) Insert a node (iii) Display a doubly linked circular list in forward direction
(ii) Delete a node (iv) Display a doubly linked circular list in reverse direction

<u>Program:</u>

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

// Node structure
typedef struct Node {
   int data;
   struct Node *prev, *next;
} Node;

// Function to create header node
Node* createHeader() {
   Node* header = (Node*)malloc(sizeof(Node));
   header->data = -1;  // header doesn't store real data
   header->next = header;
   header->prev = header;
   return header;
}

// Insert a node at the end
void insertNode(Node* header, int value) {
   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->data = value;

   Node* last = header->prev;  // last node before header

   newNode->next = header;
   newNode->prev = last;
   last->next = newNode;
   header->prev = newNode;

   printf("Inserted: %d\n", value);
}

// Delete a node with given value
void deleteNode(Node* header, int value) {
   if (header->next == header) {
      printf("List is EMPTY!\n");
      return;
   }
```

```
    Node* temp = header->next;
    while (temp != header) {
        if (temp->data == value) {
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;
            free(temp);
            printf("Deleted: %d\n", value);
            return;
        }
        temp = temp->next;
    }
    printf("Element %d not found!\n", value);
}

// Display forward
void displayForward(Node* header) {
    if (header->next == header) {
        printf("List is EMPTY!\n");
        return;
    }

    Node* temp = header->next;
    printf("Forward List: ");
    while (temp != header) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

// Display reverse
void displayReverse(Node* header) {
    if (header->prev == header) {
        printf("List is EMPTY!\n");
        return;
    }

    Node* temp = header->prev;
    printf("Reverse List: ");
    while (temp != header) {
        printf("%d ", temp->data);
        temp = temp->prev;
    }
    printf("\n");
}
```

```c
// Driver program
int main() {
    Node* header = createHeader();
    int choice, val;

    while (1) {
        printf("\n--- Doubly Linked Circular List Menu ---\n");
        printf("1. Insert\n2. Delete\n3. Display Forward\n4. Display Reverse\n5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &val);
                insertNode(header, val);
                break;
            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &val);
                deleteNode(header, val);
                break;
            case 3:
                displayForward(header);
                break;
            case 4:
                displayReverse(header);
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
}
```

Output:

```
--- Doubly Linked Circular List Menu ---
1. Insert
2. Delete
3. Display Forward
4. Display Reverse
5. Exit
Enter your choice: 1
Enter value to insert: 10
```

Inserted: 10

Enter your choice: 1
Enter value to insert: 20
Inserted: 20

Enter your choice: 1
Enter value to insert: 30
Inserted: 30

Enter your choice: 3
Forward List: 10 20 30

Enter your choice: 4
Reverse List: 30 20 10

Enter your choice: 2
Enter value to delete: 20
Deleted: 20

Enter your choice: 3
Forward List: 10 30

6. Write a C program to implement multiple linked queues (at least 5) and perform the following operations on them.

(i) Add an item in ith queue (ii) Delete an item from ith queue (iii) Display ith queue

Program:

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

#define N 5   // number of queues

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

// Queue structure
typedef struct {
   Node* front;
   Node* rear;
} Queue;

// Initialize all queues
void initQueues(Queue q[]) {
   for (int i = 0; i < N; i++) {
      q[i].front = q[i].rear = NULL;
   }
}

// Check if ith queue is empty
int isEmpty(Queue* q) {
   return (q->front == NULL);
}

// Enqueue in ith queue
void enqueue(Queue q[], int i, int value) {
   if (i < 0 || i >= N) {
      printf("Invalid Queue Number!\n");
      return;
   }

   Node* newNode = (Node*)malloc(sizeof(Node));
   newNode->data = value;
   newNode->next = NULL;

   if (q[i].rear == NULL) {
```

```
        q[i].front = q[i].rear = newNode;
    } else {
        q[i].rear->next = newNode;
        q[i].rear = newNode;
    }
    printf("Inserted %d into Queue %d\n", value, i);
}

// Dequeue from ith queue
void dequeue(Queue q[], int i) {
    if (i < 0 || i >= N) {
        printf("Invalid Queue Number!\n");
        return;
    }

    if (isEmpty(&q[i])) {
        printf("Queue %d is EMPTY!\n", i);
        return;
    }

    Node* temp = q[i].front;
    printf("Deleted %d from Queue %d\n", temp->data, i);

    q[i].front = q[i].front->next;
    if (q[i].front == NULL)
        q[i].rear = NULL;

    free(temp);
}

// Display ith queue
void display(Queue q[], int i) {
    if (i < 0 || i >= N) {
        printf("Invalid Queue Number!\n");
        return;
    }

    if (isEmpty(&q[i])) {
        printf("Queue %d is EMPTY!\n", i);
        return;
    }

    printf("Queue %d: ", i);
    Node* temp = q[i].front;
    while (temp != NULL) {
        printf("%d ", temp->data);
```

```c
        temp = temp->next;
    }
    printf("\n");
}

// Driver program
int main() {
    Queue queues[N];
    int choice, qno, val;

    initQueues(queues);

    while (1) {
        printf("\n--- Multiple Linked Queues Menu ---\n");
        printf("1. Insert into ith Queue\n");
        printf("2. Delete from ith Queue\n");
        printf("3. Display ith Queue\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter queue number (0-%d): ", N-1);
                scanf("%d", &qno);
                printf("Enter value to insert: ");
                scanf("%d", &val);
                enqueue(queues, qno, val);
                break;
            case 2:
                printf("Enter queue number (0-%d): ", N-1);
                scanf("%d", &qno);
                dequeue(queues, qno);
                break;
            case 3:
                printf("Enter queue number (0-%d): ", N-1);
                scanf("%d", &qno);
                display(queues, qno);
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice!\n");
        }
    }
    return 0;
```

}

Output:

```
--- Multiple Linked Queues Menu ---
1. Insert into ith Queue
2. Delete from ith Queue
3. Display ith Queue
4. Display ALL Queues
5. Exit
Enter your choice: 1
Enter queue number (0-4): 0
Enter value to insert: 11
Inserted 11 into Queue 0

Enter your choice: 1
Enter queue number (0-4): 2
Enter value to insert: 22
Inserted 22 into Queue 2

Enter your choice: 4
Queue 0: 11
Queue 1: EMPTY
Queue 2: 22
Queue 3: EMPTY
Queue 4: EMPTY
```

7. Write a C program to implement a binary search tree using linked representation and perform the following operations on it.

(i) Insert an item (ii) Search an item (iii) Inorder Traversal

Program:

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

// Node structure
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Create a new node
Node* createNode(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Insert a node in BST
Node* insert(Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    } else {
        printf("Duplicate value %d not allowed in BST.\n", value);
    }
    return root;
}

// Search an item in BST
Node* search(Node* root, int value) {
    if (root == NULL || root->data == value)
        return root;

    if (value < root->data)
```

```c
        return search(root->left, value);
    else
        return search(root->right, value);
}

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

// Driver program
int main() {
    Node* root = NULL;
    int choice, val;
    Node* found;

    while (1) {
        printf("\n--- Binary Search Tree Menu ---\n");
        printf("1. Insert\n2. Search\n3. Inorder Traversal\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &val);
                root = insert(root, val);
                break;

            case 2:
                printf("Enter value to search: ");
                scanf("%d", &val);
                found = search(root, val);
                if (found != NULL)
                    printf("Value %d found in BST.\n", val);
                else
                    printf("Value %d not found in BST.\n", val);
                break;

            case 3:
                printf("Inorder Traversal: ");
                inorder(root);
```

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

        case 4:
            exit(0);

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

Output:

```
--- Binary Search Tree Menu ---
1. Insert
2. Search
3. Inorder Traversal
4. Exit
Enter your choice: 1
Enter value to insert: 50

Enter your choice: 1
Enter value to insert: 30

Enter your choice: 1
Enter value to insert: 70

Enter your choice: 1
Enter value to insert: 20

Enter your choice: 1
Enter value to insert: 40

Enter your choice: 1
Enter value to insert: 60

Enter your choice: 3
Inorder Traversal: 20 30 40 50 60 70

Enter your choice: 2
Enter value to search: 40
Value 40 found in BST.
```

8. Write a C program to implement Red black tree.

(i) Insert an item (ii) delete an item (iii) display the elements

Program:

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

typedef enum { RED, BLACK } Color;

typedef struct Node {
    int data;
    Color color;
    struct Node *left, *right, *parent;
} Node;

Node *root = NULL;
Node *TNULL;

void initializeTNULL() {
    TNULL = (Node *)malloc(sizeof(Node));
    TNULL->color = BLACK;
    TNULL->left = TNULL->right = TNULL->parent = NULL;
}

Node *newNode(int data) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->data = data;
    node->color = RED;
    node->left = node->right = node->parent = TNULL;
    return node;
}

void leftRotate(Node *x) {
    Node *y = x->right;
    x->right = y->left;
    if (y->left != TNULL)
        y->left->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        root = y;
    else if (x == x->parent->left)
        x->parent->left = y;
    else
        x->parent->right = y;
    y->left = x;
    x->parent = y;
```

```
}

void rightRotate(Node *x) {
    Node *y = x->left;
    x->left = y->right;
    if (y->right != TNULL)
        y->right->parent = x;
    y->parent = x->parent;
    if (x->parent == NULL)
        root = y;
    else if (x == x->parent->right)
        x->parent->right = y;
    else
        x->parent->left = y;
    y->right = x;
    x->parent = y;
}

void fixInsert(Node *k) {
    Node *u;
    while (k->parent->color == RED) {
        if (k->parent == k->parent->parent->right) {
            u = k->parent->parent->left;
            if (u->color == RED) {
                u->color = BLACK;
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->left) {
                    k = k->parent;
                    rightRotate(k);
                }
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                leftRotate(k->parent->parent);
            }
        } else {
            u = k->parent->parent->right;
            if (u->color == RED) {
                u->color = BLACK;
                k->parent->color = BLACK;
                k->parent->parent->color = RED;
                k = k->parent->parent;
            } else {
                if (k == k->parent->right) {
```

```
                k = k->parent;
                leftRotate(k);
            }
            k->parent->color = BLACK;
            k->parent->parent->color = RED;
            rightRotate(k->parent->parent);
        }
    }
    if (k == root)
        break;
  }
  root->color = BLACK;
}

void insert(int key) {
  Node *node = newNode(key);
  Node *y = NULL;
  Node *x = root;
  while (x != TNULL) {
    y = x;
    if (node->data < x->data)
      x = x->left;
    else
      x = x->right;
  }
  node->parent = y;
  if (y == NULL)
    root = node;
  else if (node->data < y->data)
    y->left = node;
  else
    y->right = node;
  if (node->parent == NULL) {
    node->color = BLACK;
    return;
  }
  if (node->parent->parent == NULL)
    return;
  fixInsert(node);
}

void inorderHelper(Node *root) {
  if (root != TNULL) {
    inorderHelper(root->left);
    printf("%d ", root->data);
    inorderHelper(root->right);
```

```
    }
}

void inorder() {
    inorderHelper(root);
}

int main() {
    initializeTNULL();
    insert(55);
    insert(40);
    insert(65);
    insert(60);
    insert(75);
    insert(57);

    printf("Inorder traversal: ");
    inorder();
    printf("\n");

    return 0;
}
```

Input:
55, 40, 65, 60, 75, 57

Inorder Traversal Output:

40 55 57 60 65 75

Console Output of the Program

Inorder traversal: 40 55 57 60 65 75

Inserted elements for Red-Black Tree

55, 40, 65, 60, 75, 57

Red-Black Tree Structure

- **BLACK nodes**: B
- **RED nodes**: R

```
          60(B)
         /      \
    55(R)      65(B)
   /           /   \
 40(B)       57(R) 75(R)
```

9. Write a C program to perform depth first search of a graph represented as an adjacency list.

Program:

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

#define MAX 100

// Node for adjacency list
typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

// Graph structure
typedef struct Graph {
    int numVertices;
    Node** adjLists;
    int* visited;
} Graph;

// Create a node
Node* createNode(int v) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

// Create a graph
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;

    graph->adjLists = (Node**)malloc(vertices * sizeof(Node*));
    graph->visited = (int*)malloc(vertices * sizeof(int));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }

    return graph;
}
```

```
// Add edge (undirected graph)
void addEdge(Graph* graph, int src, int dest) {
    // Add edge from src to dest
    Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Add edge from dest to src (undirected)
    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// DFS traversal
void DFS(Graph* graph, int vertex) {
    graph->visited[vertex] = 1;
    printf("%d ", vertex);

    Node* temp = graph->adjLists[vertex];
    while (temp != NULL) {
        int connectedVertex = temp->vertex;
        if (!graph->visited[connectedVertex]) {
            DFS(graph, connectedVertex);
        }
        temp = temp->next;
    }
}

// Display adjacency list
void displayGraph(Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        Node* temp = graph->adjLists[i];
        printf("%d: ", i);
        while (temp) {
            printf("%d -> ", temp->vertex);
            temp = temp->next;
        }
        printf("NULL\n");
    }
}

// Driver program
int main() {
    int vertices, edges, src, dest, start;

    printf("Enter number of vertices: ");
```

```
    scanf("%d", &vertices);
    Graph* graph = createGraph(vertices);

    printf("Enter number of edges: ");
    scanf("%d", &edges);

    for (int i = 0; i < edges; i++) {
        printf("Enter edge %d (source destination): ", i + 1);
        scanf("%d %d", &src, &dest);
        addEdge(graph, src, dest);
    }

    printf("\nAdjacency List of the Graph:\n");
    displayGraph(graph);

    printf("\nEnter starting vertex for DFS: ");
    scanf("%d", &start);

    printf("DFS traversal starting from vertex %d: ", start);
    DFS(graph, start);
    printf("\n");

    return 0;
}
```

Output:

```
Enter number of vertices: 5
Enter number of edges: 4
Enter edge 1 (source destination): 0 1
Enter edge 2 (source destination): 0 2
Enter edge 3 (source destination): 1 3
Enter edge 4 (source destination): 2 4

Adjacency List of the Graph:
0: 2 -> 1 -> NULL
1: 3 -> 0 -> NULL
2: 4 -> 0 -> NULL
3: 1 -> NULL
4: 2 -> NULL

Enter starting vertex for DFS: 0
DFS traversal starting from vertex 0: 0 2 4 1 3
```

10. Design and develop a program in C that uses Hash Function H:K->L as H(K)=K mod m(reminder method) and implement hashing technique to map a given key K to the address space L. Resolve the collision (if any) using linear probing

Program:

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

#define SIZE 10   // Size of hash table

// Function to insert a key using linear probing
void insert(int hashTable[], int key) {
    int index = key % SIZE;
    int originalIndex = index;
    int i = 0;

    // Linear probing
    while (hashTable[index] != -1) {
        index = (originalIndex + i) % SIZE;
        i++;
        if (i == SIZE) {
            printf("Hash table is full! Cannot insert %d\n", key);
            return;
        }
    }

    hashTable[index] = key;
    printf("Inserted %d at index %d\n", key, index);
}

// Function to display hash table
void display(int hashTable[]) {
    printf("\nHash Table:\n");
    for (int i = 0; i < SIZE; i++) {
        if (hashTable[i] != -1)
            printf("Index %d -> %d\n", i, hashTable[i]);
        else
            printf("Index %d -> NULL\n", i);
    }
}

// Function to search a key
void search(int hashTable[], int key) {
    int index = key % SIZE;
    int originalIndex = index;
```

```c
    int i = 0;

    while (hashTable[index] != -1) {
        if (hashTable[index] == key) {
            printf("Key %d found at index %d\n", key, index);
            return;
        }
        i++;
        index = (originalIndex + i) % SIZE;
        if (i == SIZE)
            break;
    }
    printf("Key %d not found in hash table.\n", key);
}

// Driver program
int main() {
    int hashTable[SIZE];

    // Initialize hash table
    for (int i = 0; i < SIZE; i++)
        hashTable[i] = -1;

    int choice, key;

    while (1) {
        printf("\n--- Hashing Menu ---\n");
        printf("1. Insert Key\n2. Search Key\n3. Display Hash Table\n4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter key to insert: ");
                scanf("%d", &key);
                insert(hashTable, key);
                break;
            case 2:
                printf("Enter key to search: ");
                scanf("%d", &key);
                search(hashTable, key);
                break;
            case 3:
                display(hashTable);
                break;
            case 4:
```

```
            exit(0);
          default:
            printf("Invalid choice!\n");
        }
    }

    return 0;
}
```

Output:

```
--- Hashing Menu ---
1. Insert Key
2. Search Key
3. Display Hash Table
4. Exit
Enter your choice: 1
Enter key to insert: 12
Inserted 12 at index 2

Enter your choice: 1
Enter key to insert: 22
Inserted 22 at index 3

Enter your choice: 1
Enter key to insert: 32
Inserted 32 at index 4

Enter your choice: 3

Hash Table:
Index 0 -> NULL
Index 1 -> NULL
Index 2 -> 12
Index 3 -> 22
Index 4 -> 32
Index 5 -> NULL
Index 6 -> NULL
Index 7 -> NULL
Index 8 -> NULL
Index 9 -> NULL

Enter your choice: 2
Enter key to search: 22
Key 22 found at index 3
```