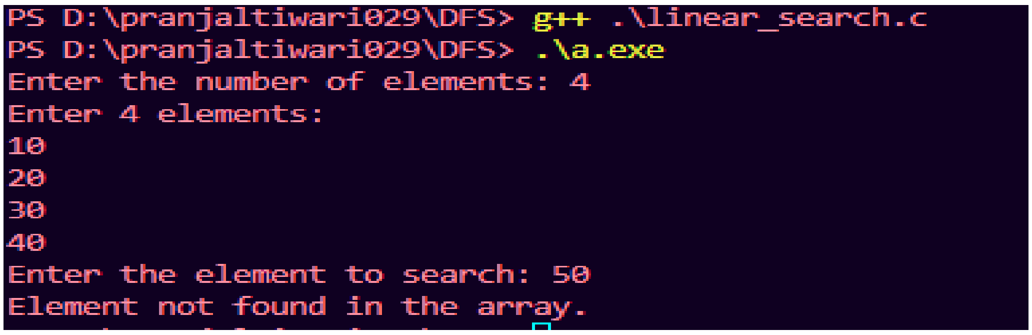
Q1) Write a program in C to implement Linear Search

Code

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return i;
        }
    }
    return -1;
}
int main() {
    int n, key;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Enter the element to search: ");
    scanf("%d", &key);

    int result = linearSearch(arr, n, key);

    if (result == -1) {
        printf("Element not found in the array.\n");
    } else {
        printf("Element found at index %d.\n", result);
    }
    return 0;
}
```

Output

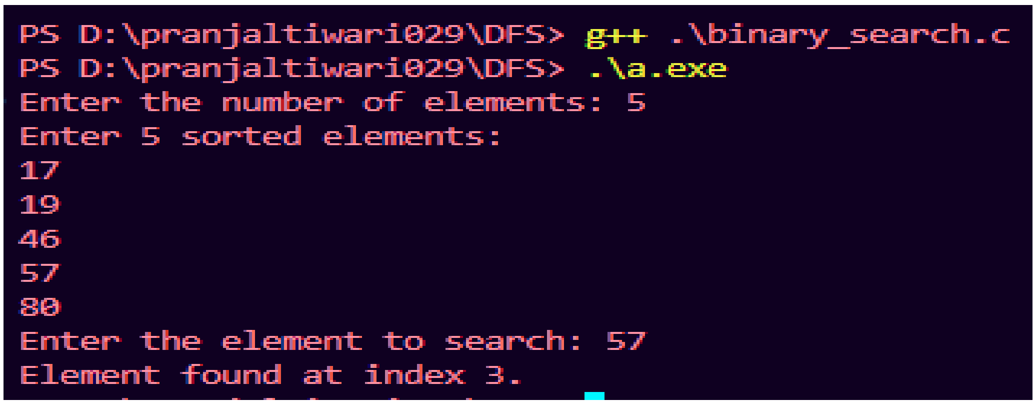


```
PS D:\pranjaltiwari029\DFS> g++ .\linear_search.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter the number of elements: 4
Enter 4 elements:
10
20
30
40
Enter the element to search: 50
Element not found in the array.
```

Q2) Write a program in C to implement Binary Search

```c
#include <stdio.h>

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key)
            return mid;
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }

    return -1;
}

int main() {
    int n, key;

    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter %d sorted elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

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

    int result = binarySearch(arr, n, key);

    if (result == -1) {
        printf("Element not found in the array.\n");
    } else {
        printf("Element found at index %d.\n", result);
    }

    return 0;
}
```

OUTPUT



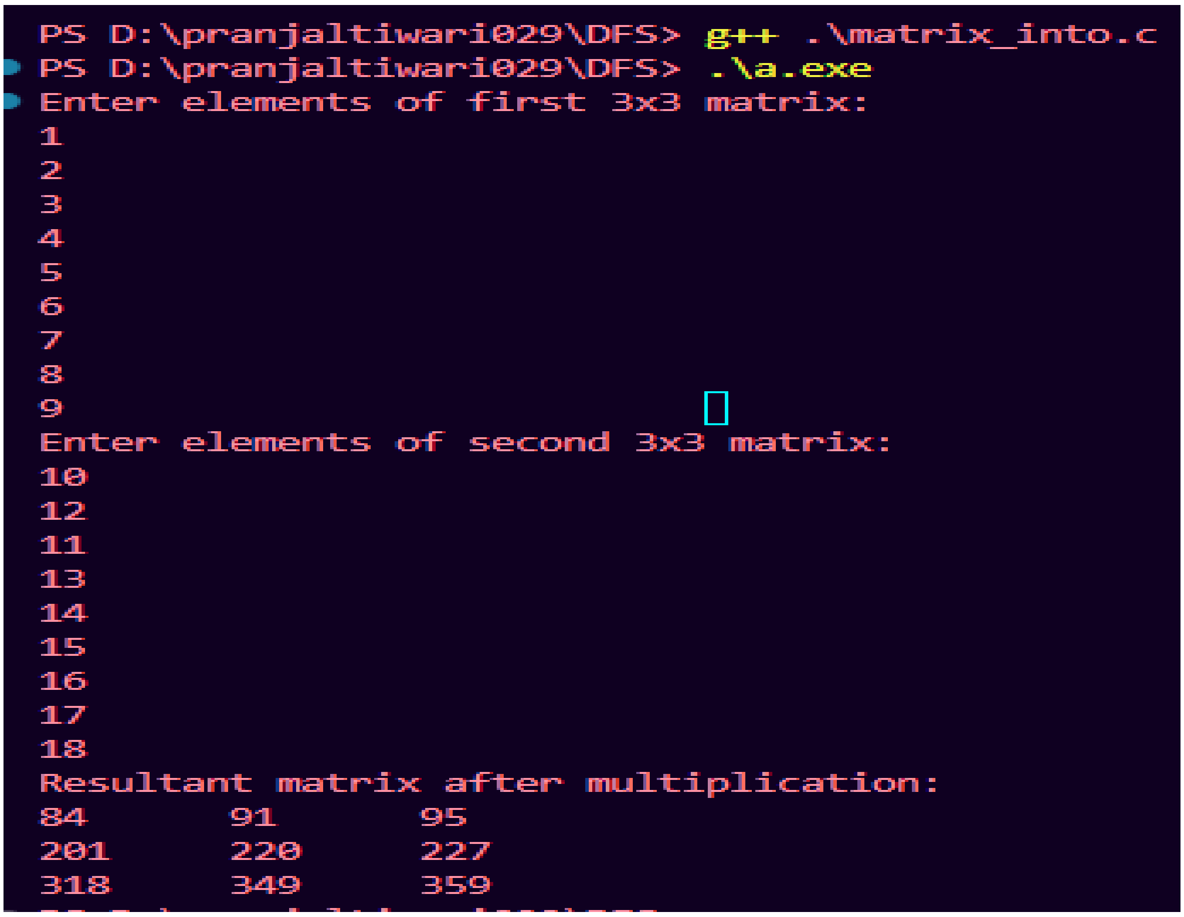
```
PS D:\pranjaltiwari029\DFS> g++ .\binary_search.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter the number of elements: 5
Enter 5 sorted elements:
17
19
46
57
80
Enter the element to search: 57
Element found at index 3.
```

Q3) Write a program in C to implement Matrix Multiplication on two Matrices of 3x3.

Code

```c
#include <stdio.h>

int main() {
    int a[3][3], b[3][3], result[3][3];
    int i, j, k;


    printf("Enter elements of first 3x3 matrix:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &a[i][j]);
        }
    }


    printf("Enter elements of second 3x3 matrix:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            scanf("%d", &b[i][j]);
        }
    }


    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            result[i][j] = 0;
        }
    }


    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            for (k = 0; k < 3; k++) {
                result[i][j] += a[i][k] * b[k][j];
            }
        }
    }


    printf("Resultant matrix after multiplication:\n");
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            printf("%d\t", result[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

OUTPUT



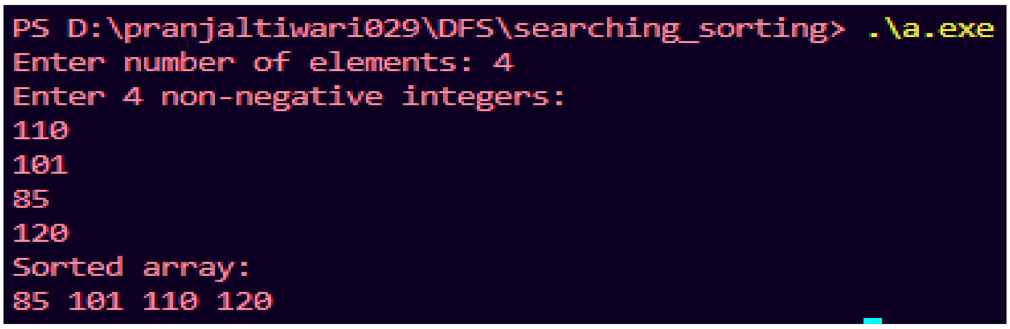
```
PS D:\pranjaltiwari029\DFS> g++ .\matrix_into.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter elements of first 3x3 matrix:
1
2
3
4
5
6
7
8
9
Enter elements of second 3x3 matrix:
10
12
11
13
14
15
16
17
18
Resultant matrix after multiplication:
84      91      95
201     220     227
318     349     359
```

Q4) Write a program in C to implement Selection Sort

Code

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int i, j, min_idx, temp;

    for (i = 0; i < n-1; i++) {
        min_idx = i; // Assume the current element is the minimum

        // Find the index of the minimum element in the remaining array
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element
        temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int n, i;

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

    int arr[n];

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    selectionSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT

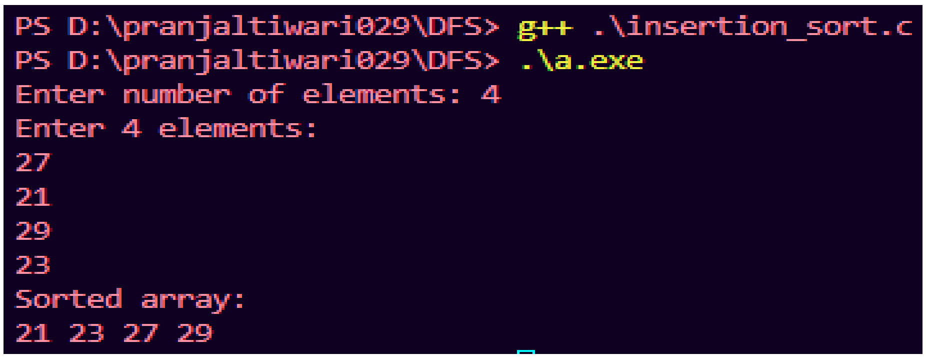


```
PS D:\pranjaltiwari029\DFS\searching_sorting> .\a.exe
Enter number of elements: 4
Enter 4 non-negative integers:
110
101
85
120
Sorted array:
85 101 110 120
```

Q5) Write a program in C to implement insertion sort

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    int i, key, j;

    for (i = 1; i < n; i++) {
        key = arr[i]; // current element to be inserted
        j = i - 1;

        // Move elements of arr[0..i-1] that are greater than key
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }

        arr[j + 1] = key; // Insert the key at correct position
    }
}

int main() {
    int n, i;

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

    int arr[n];

    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

OUTPUT

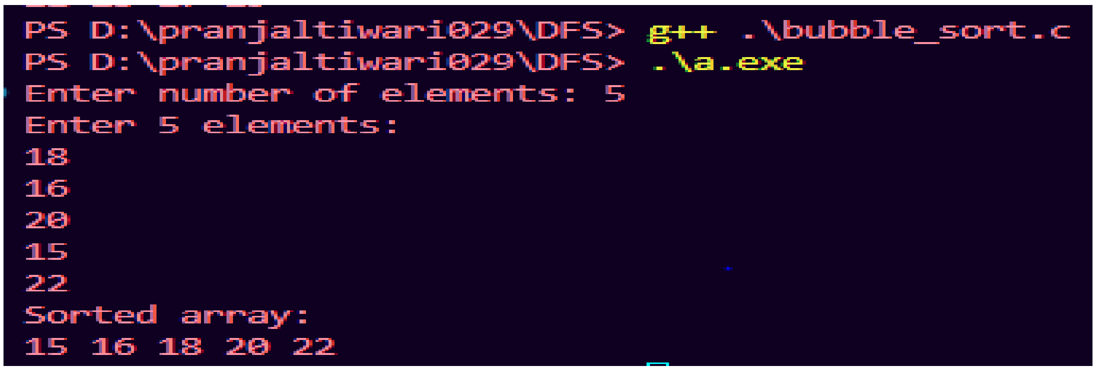

```
PS D:\pranjaltiwari029\DFS> g++ .\insertion_sort.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 4
Enter 4 elements:
27
21
29
23
Sorted array:
21 23 27 29
```

Q6) Write a program in C to implement Bubble Sort

Code

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        // Last i elements are already in place
        for (j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                // Swap arr[j] and arr[j+1]
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
int main() {
    int n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d elements:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    bubbleSort(arr, n);
    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Output

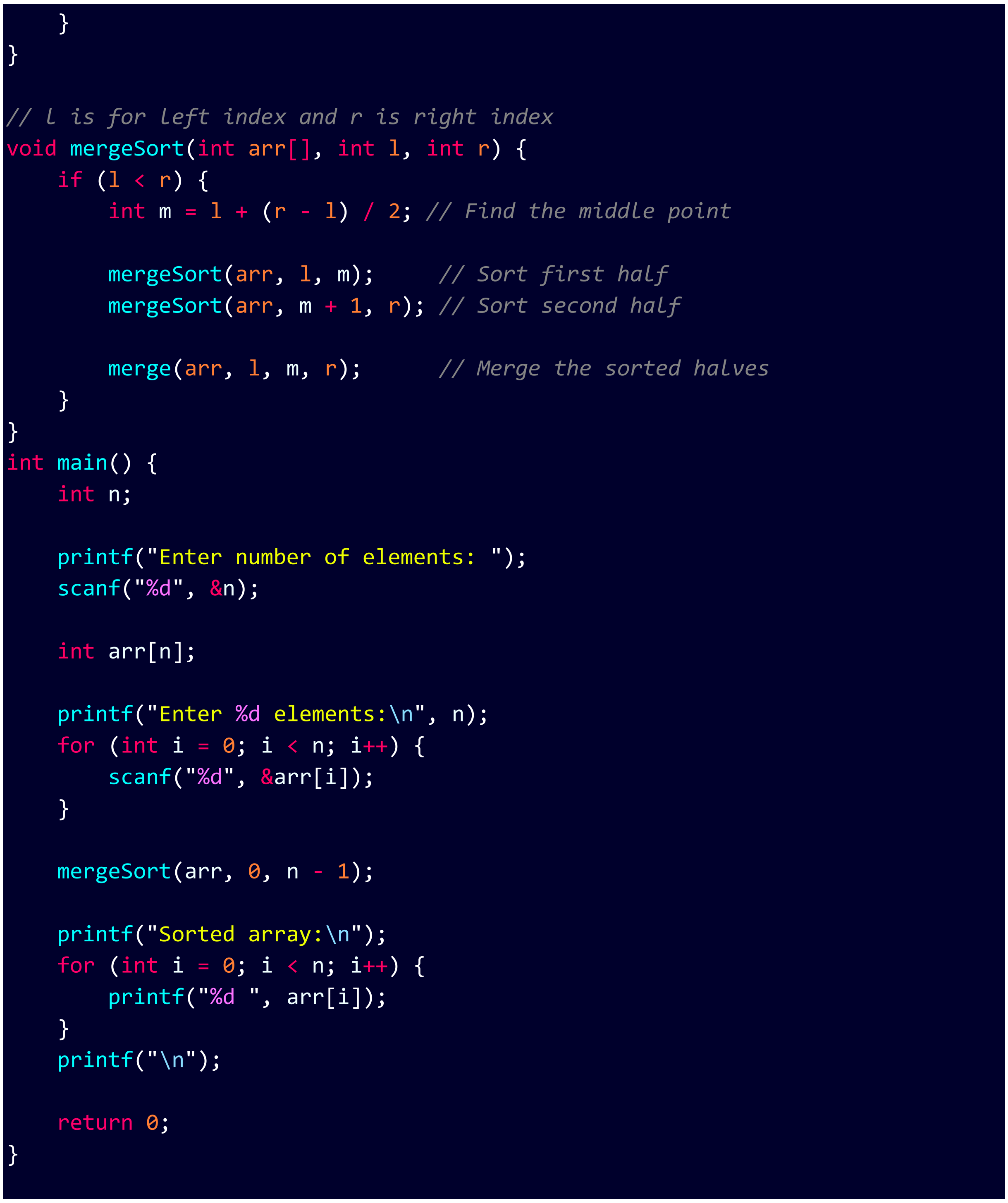


```
PS D:\pranjaltiwari029\DFS> g++ .\bubble_sort.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 5
Enter 5 elements:
18
16
20
15
22
Sorted array:
15 16 18 20 22
```

Q7) Write a program in C to implement Merge Sort

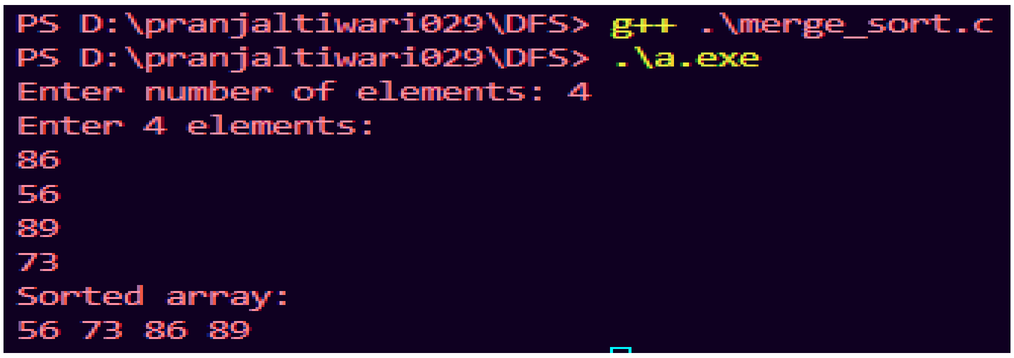
Code

```c
#include <stdio.h>

// Merge two subarrays of arr[]
void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;

    // Create temporary arrays
    int L[n1], R[n2];

    // Copy data to temp arrays L[] and R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Merge the temp arrays back into arr[l..r]
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray

    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
```



```c
    }
}

// l is for left index and r is right index
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2; // Find the middle point

        mergeSort(arr, l, m);      // Sort first half
        mergeSort(arr, m + 1, r); // Sort second half

        merge(arr, l, m, r);       // Merge the sorted halves
    }
}
int main() {
    int n;

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

    int arr[n];

    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    mergeSort(arr, 0, n - 1);

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

    return 0;
}
```

OUTPUT



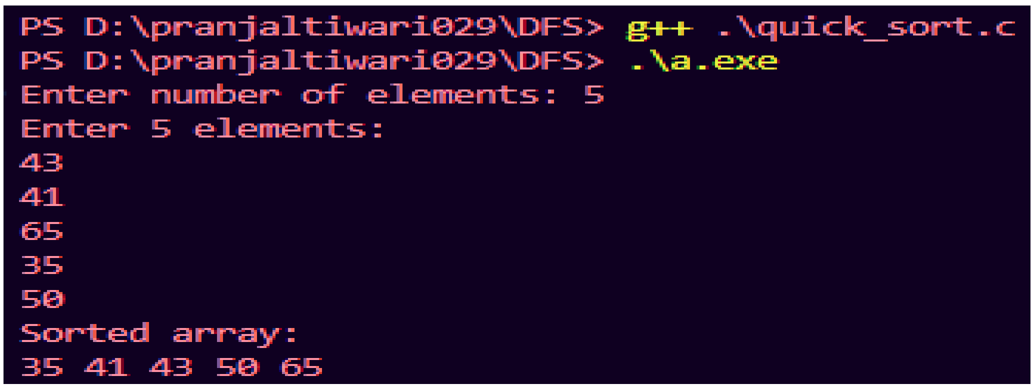
```
PS D:\pranjaltiwari029\DFS> g++ .\merge_sort.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 4
Enter 4 elements:
86
56
89
73
Sorted array:
56 73 86 89
```

Q8) Write a program in C to implement quick sort

Code

```c
#include <stdio.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // choosing the last element as pivot
    int i = (low - 1); // index of smaller element

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    swap(&arr[i + 1], &arr[high]);
    return (i + 1); // return the partition point
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high); // pi is partitioning index

        quickSort(arr, low, pi - 1); // sort elements before partition
        quickSort(arr, pi + 1, high); // sort elements after partition
    }
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    int arr[n];

    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);
    printf("Sorted array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output



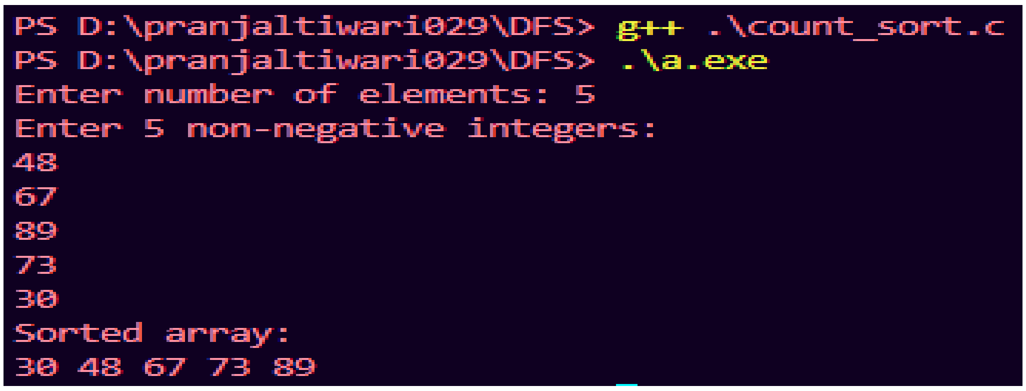
```
PS D:\pranjaltiwari029\DFS> g++ .\quick_sort.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 5
Enter 5 elements:
43
41
65
35
50
Sorted array:
35 41 43 50 65
```

Q9) Write a program in C to implement Count Sort.

Code

```c
#include <stdio.h>

void countSort(int arr[], int n) {
    int i;

    // Find the maximum element in the array
    int max = arr[0];
    for (i = 1; i < n; i++) {
        if (arr[i] > max)
            max = arr[i];
    }

    // Create a count array to store count of individual elements
    int count[max + 1];

    // Initialize count array with 0
    for (i = 0; i <= max; i++) {
        count[i] = 0;
    }

    // Store the count of each element
    for (i = 0; i < n; i++) {
        count[arr[i]]++;
    }

    // Modify the original array using the count array
    int index = 0;
    for (i = 0; i <= max; i++) {
        while (count[i] > 0) {
            arr[index++] = i;
            count[i]--;
        }
    }
}

int main() {
    int n, i;

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

    int arr[n];

    printf("Enter %d non-negative integers:\n", n);
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    countSort(arr, n);

    printf("Sorted array:\n");
    for (i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output



```
PS D:\pranjaltiwari029\DFS> g++ .\count_sort.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 5
Enter 5 non-negative integers:
48
67
89
73
30
Sorted array:
30 48 67 73 89
```

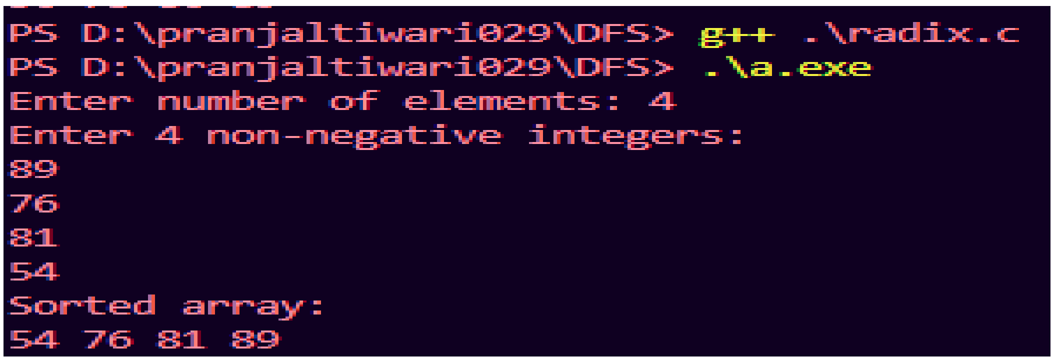
Q10) Write a program in C to implement Radix Sort.

Code

```c
#include <stdio.h>

// A utility function to get the maximum value in arr[]
int getMax(int arr[], int n) {
    int max = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > max)
            max = arr[i];
    return max;
}

// A function to do counting sort based on a specific digit (exp is 1, 10,
100, etc.)
void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int count[10] = {0};

    // Store count of occurrences
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that it contains actual position
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (int i = n - 1; i >= 0; i--) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy the output array back to arr[]
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] using Radix Sort
void radixSort(int arr[], int n) {
    int max = getMax(arr, n);

    // Do counting sort for every digit (exp = 1, 10, 100, ...)
    for (int exp = 1; max / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
```

```c
int main() {
    int n;

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

    int arr[n];

    printf("Enter %d non-negative integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    radixSort(arr, n);

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

    return 0;
}
```

Output



```
PS D:\pranjaltiwari029\DFS> g++ .\radix.c
PS D:\pranjaltiwari029\DFS> .\a.exe
Enter number of elements: 4
Enter 4 non-negative integers:
89
76
81
54
Sorted array:
54 76 81 89
```

Q11) Write a program in C to implement binary trees operations and traversals .

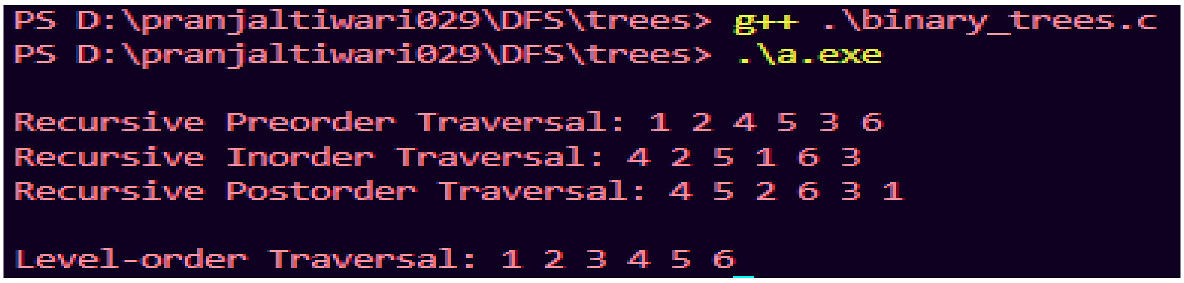
Code

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

// Define the structure for tree node
typedef struct Node {
    int data;
    struct Node* left;
    struct Node* right;
} Node;

// Define queue structure for level order operations
typedef struct QueueNode {
    Node* treeNode;
    struct QueueNode* next;
} QueueNode;

typedef struct Queue {
    QueueNode *front, *rear;
} Queue;

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

// ----------------- Queue Functions -----------------
Queue* createQueue() {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->front = q->rear = NULL;
    return q;
}

void enqueue(Queue* q, Node* node) {
    QueueNode* temp = (QueueNode*)malloc(sizeof(QueueNode));
    temp->treeNode = node;
    temp->next = NULL;
    if (q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
```

```
}

Node* dequeue(Queue* q) {
    if (q->front == NULL)
        return NULL;
    QueueNode* temp = q->front;
    Node* node = temp->treeNode;
    q->front = q->front->next;
    if (q->front == NULL)
        q->rear = NULL;
    free(temp);
    return node;
}

int isQueueEmpty(Queue* q) {
    return q->front == NULL;
}

// ----------------- Build Tree using Level Order -----------------
Node* buildTree(int arr[], int n) {
    if (n == 0)
        return NULL;

    Node* root = createNode(arr[0]);
    Queue* q = createQueue();
    enqueue(q, root);

    int i = 1;
    while (i < n) {
        Node* temp = dequeue(q);

        if (i < n) {
            temp->left = createNode(arr[i++]);
            enqueue(q, temp->left);
        }
        if (i < n) {
            temp->right = createNode(arr[i++]);
            enqueue(q, temp->right);
        }
    }
    return root;
}

// ----------------- Recursive Traversals -----------------
void preorderRecursive(Node* root) {
    if (root == NULL)
        return;
    printf("%d ", root->data);
```

```c
    preorderRecursive(root->left);
    preorderRecursive(root->right);
}

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

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

// ----------------- Level-order Traversal -----------------
void levelOrderTraversal(Node* root) {
    if (root == NULL)
        return;
    Queue* q = createQueue();
    enqueue(q, root);
    while (!isQueueEmpty(q)) {
        Node* temp = dequeue(q);
        printf("%d ", temp->data);
        if (temp->left)
            enqueue(q, temp->left);
        if (temp->right)
            enqueue(q, temp->right);
    }
}

// ----------------- Main Function -----------------
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6}; // Sample input
    int n = sizeof(arr) / sizeof(arr[0]);

    Node* root = buildTree(arr, n);

    printf("\nRecursive Preorder Traversal: ");
    preorderRecursive(root);

    printf("\nRecursive Inorder Traversal: ");
    inorderRecursive(root);
    printf("\nRecursive Postorder Traversal: ");
    postorderRecursive(root);

    printf("\n\nLevel-order Traversal: ");
    levelOrderTraversal(root);

    printf("\n");

    return 0;
}
```

OUTPUT



```
PS D:\pranjaltiwari029\DFS\trees> g++ .\binary_trees.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe

Recursive Preorder Traversal: 1 2 4 5 3 6
Recursive Inorder Traversal: 4 2 5 1 6 3
Recursive Postorder Traversal: 4 5 2 6 3 1

Level-order Traversal: 1 2 3 4 5 6
```

Q12) Write a program in C to implement binary search trees

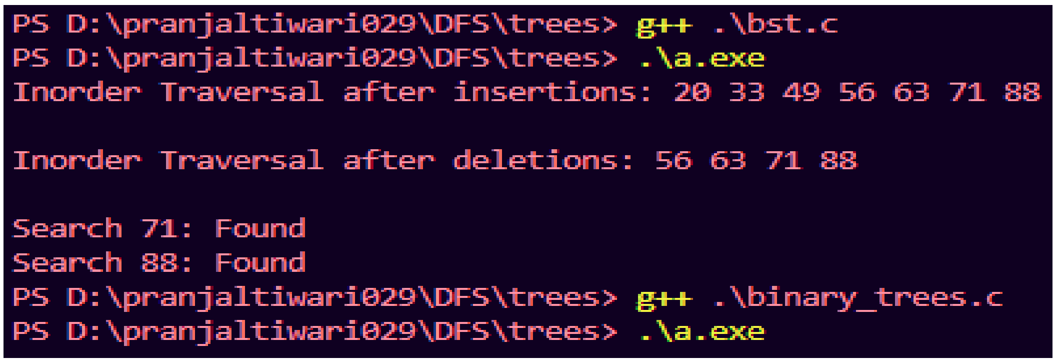
Code

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

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

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

// Insert into BST
Node* insert(Node* root, int data) {
    if (root == NULL)
        return createNode(data);

    if (data < root->data)
        root->left = insert(root->left, data);
    else if (data > root->data)
        root->right = insert(root->right, data);

    return root;
}

// Inorder Traversal
void inorderTraversal(Node* root) {
    if (root == NULL)
        return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}

// Search in BST
int search(Node* root, int key) {
    if (root == NULL)
        return 0;
    if (root->data == key)
```

```
        return 1;
    else if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}

// Find minimum value node
Node* findMin(Node* root) {
    while (root && root->left != NULL)
        root = root->left;
    return root;
}

// Delete a node from BST
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return NULL;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node found
        if (root->left == NULL) {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            Node* temp = root->left;
            free(root);
            return temp;
        }
        // Node with two children
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Main function
int main() {
    Node* root = NULL;

    // Insert elements
```

```c
    int elements[] = {56, 33, 71, 20, 49, 63, 88};
    int n = sizeof(elements) / sizeof(elements[0]);

    for (int i = 0; i < n; i++) {
        root = insert(root, elements[i]);
    }

    printf("Inorder Traversal after insertions: ");
    inorderTraversal(root);
    printf("\n");

    // Delete nodes:
    int toDelete[] = {20, 33, 49};
    for (int i = 0; i < 3; i++) {
        root = deleteNode(root, toDelete[i]);
    }

    printf("\nInorder Traversal after deletions: ");
    inorderTraversal(root);
    printf("\n");


    int toSearch[] = {71, 88};
    for (int i = 0; i < 2; i++) {
        if (search(root, toSearch[i]))
            printf("\nSearch %d: Found", toSearch[i]);
        else
            printf("\nSearch %d: Not Found", toSearch[i]);
    }

    printf("\n");
    return 0;
}
```

Output



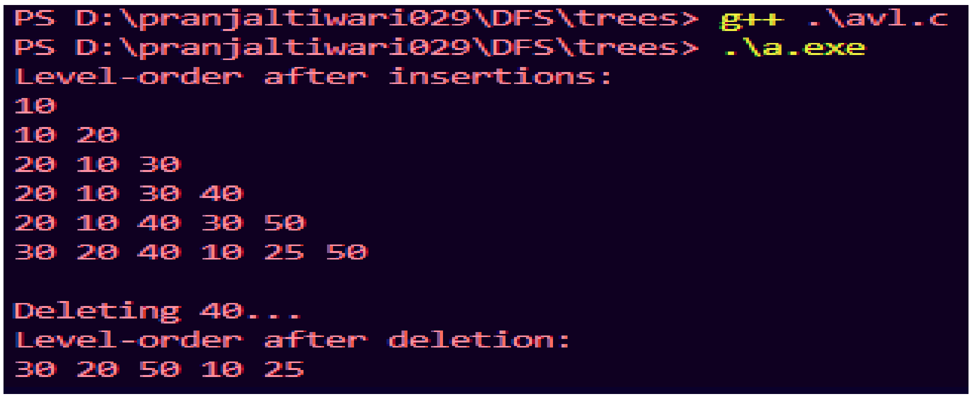
```
PS D:\pranjaltiwari029\DFS\trees> g++ .\bst.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe
Inorder Traversal after insertions: 20 33 49 56 63 71 88

Inorder Traversal after deletions: 56 63 71 88

Search 71: Found
Search 88: Found
PS D:\pranjaltiwari029\DFS\trees> g++ .\binary_trees.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe
```

Q13) Write a program in C to implement avl trees

Code

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

// AVL Tree Node
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
    int height;
} Node;

// Utility to get max
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Get height of node
int height(Node* N) {
    if (N == NULL)
        return 0;
    return N->height;
}

// Create a new node
Node* newNode(int key) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->key = key;
    node->left = node->right = NULL;
    node->height = 1; // New node is initially at height 1
    return node;
}

// Right rotate
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

// Left rotate
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

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

    return y;
}

// Get balance factor
int getBalance(Node* N) {
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Insert into AVL Tree
Node* insert(Node* node, int key) {
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else
        return node; // Equal keys not allowed

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

    int balance = getBalance(node);

    // If unbalanced, there are 4 cases:

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

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

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

    return node;
}

// Find minimum node
Node* minValueNode(Node* node) {
    Node* current = node;

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

// Delete from AVL Tree
Node* deleteNode(Node* root, int key) {
    if (root == NULL)
        return root;

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

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

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

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

            free(temp);
        }
        else {
            Node* temp = minValueNode(root->right);

            root->key = temp->key;

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

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

    int balance = getBalance(root);

    // Left Left
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }
    return root;
}
// Print Level Order Traversal
void printLevelOrder(Node* root) {
    if (root == NULL)
        return;

    Node* queue[100];
    int front = 0, rear = 0;
    queue[rear++] = root;

    while (front < rear) {
        Node* current = queue[front++];
        printf("%d ", current->key);

        if (current->left != NULL)
            queue[rear++] = current->left;
        if (current->right != NULL)
            queue[rear++] = current->right;
    }
}
int main() {
    Node* root = NULL;

    int insert_elements[] = {10, 20, 30, 40, 50, 25};
    int n = sizeof(insert_elements) / sizeof(insert_elements[0]);

    printf("Level-order after insertions:\n");
    for (int i = 0; i < n; i++) {
        root = insert(root, insert_elements[i]);
        printLevelOrder(root);
        printf("\n");
    }

    printf("\nDeleting 40...\n");
    root = deleteNode(root, 40);

    printf("Level-order after deletion:\n");
    printLevelOrder(root);
    printf("\n");

    return 0;
}
```

Output



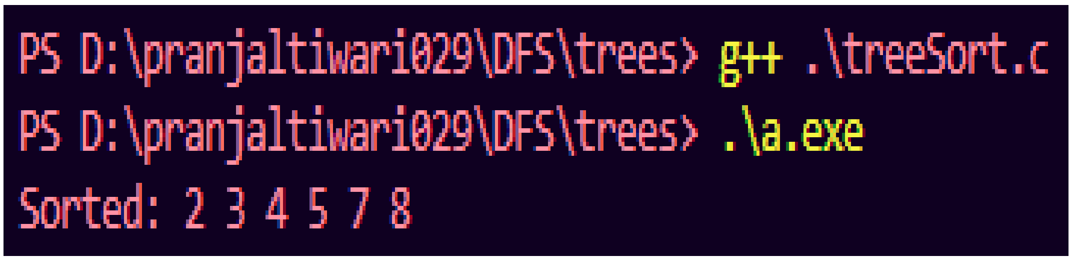
```
PS D:\pranjaltiwari029\DFS\trees> g++ .\avl.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe
Level-order after insertions:
10
10 20
20 10 30
20 10 30 40
20 10 40 30 50
30 20 40 10 25 50

Deleting 40...
Level-order after deletion:
30 20 50 10 25
```

Q14) Write a program in C to implement Tree Sort using BST.

Code

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

// Define a BST node
typedef struct Node {
    int key;
    struct Node* left;
    struct Node* right;
} Node;

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

// Insert into BST
Node* insert(Node* root, int key) {
    if (root == NULL)
        return newNode(key);

    if (key < root->key)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);

    return root;
}

// Inorder Traversal (prints sorted elements)
void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->key);
        inorderTraversal(root->right);
    }
}

int main() {
    int elements[] = {5, 3, 7, 2, 8, 4};
    int n = sizeof(elements) / sizeof(elements[0]);

    Node* root = NULL;
    // Insert elements into BST
    for (int i = 0; i < n; i++) {
        root = insert(root, elements[i]);
    }


    printf("Sorted: ");
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output

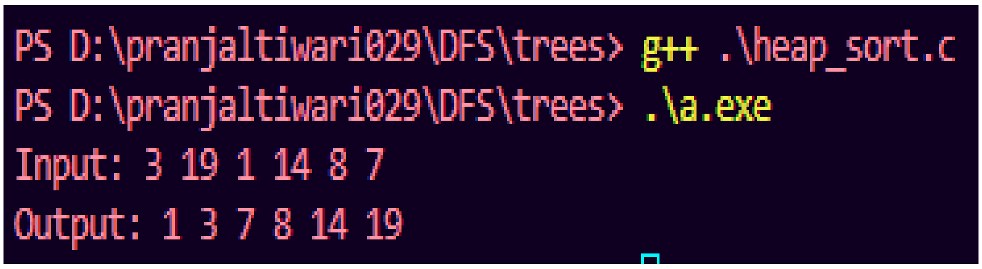


```
PS D:\pranjaltiwari029\DFS\trees> g++ .\treeSort.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe
Sorted: 2 3 4 5 7 8
```

Q16) Write a program in C to implement heap sort.
Code

```c
#include <stdio.h>

// Function to heapify a subtree rooted at index i in a max heap
void heapify(int arr[], int n, int i) {
    int largest = i;         // Initialize largest as root
    int left = 2 * i + 1;    // Left child index
    int right = 2 * i + 2;   // Right child index

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than the largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
        // Swap root and largest
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected subtree
        heapify(arr, n, largest);
    }
}

// Function to perform heap sort
void heapSort(int arr[], int n) {
    // Build a max heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract elements from the heap
    for (int i = n - 1; i > 0; i--) {
        // Swap current root with the end element
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print the array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {3, 19, 1, 14, 8, 7};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Input: ");
    printArray(arr, n);

    heapSort(arr, n);

    printf("Output: ");
    printArray(arr, n);

    return 0;
}
```

Output



```
PS D:\pranjaltiwari029\DFS\trees> g++ .\heap_sort.c
PS D:\pranjaltiwari029\DFS\trees> .\a.exe
Input: 3 19 1 14 8 7
Output: 1 3 7 8 14 19
```

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES

VIVEKANANDA SCHOOL OF INFORMATION TECHNOLOGY

BACHELOR OF COMPUTER APPLICATION

Data and File Structures

MCA-162

Guru Gobind Singh Indraprastha University
Sector - 16C, Dwarka, Delhi - 110078

SUBMITTED TO:	**SUBMITTED BY:**
Varun Bhatuda	Pranjal Tiwari
VSIT,VIPS	12017704424
	MCA 2-B