23CSE111-DESIGN AND ANALYSIS OF ALGORTHIM

LAB-5

1.MERGE SORTING

Merge Sort is a **divide-and-conquer** sorting algorithm.

It:

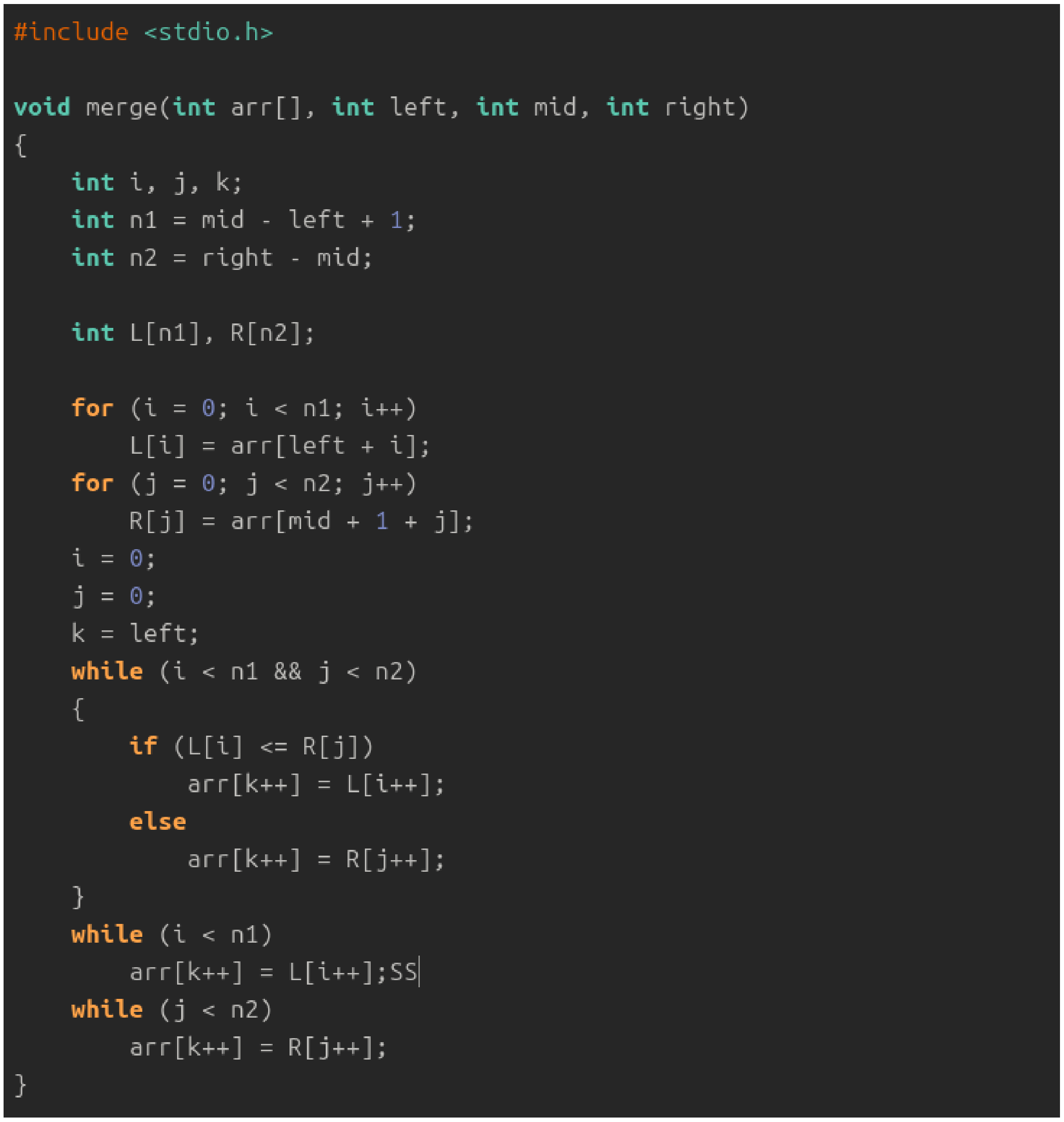
1. Divides the array into smaller subarrays
2. Sorts those subarrays
3. Merges them back together in sorted order

Merge Sort Algorithm (Step-by-Step):

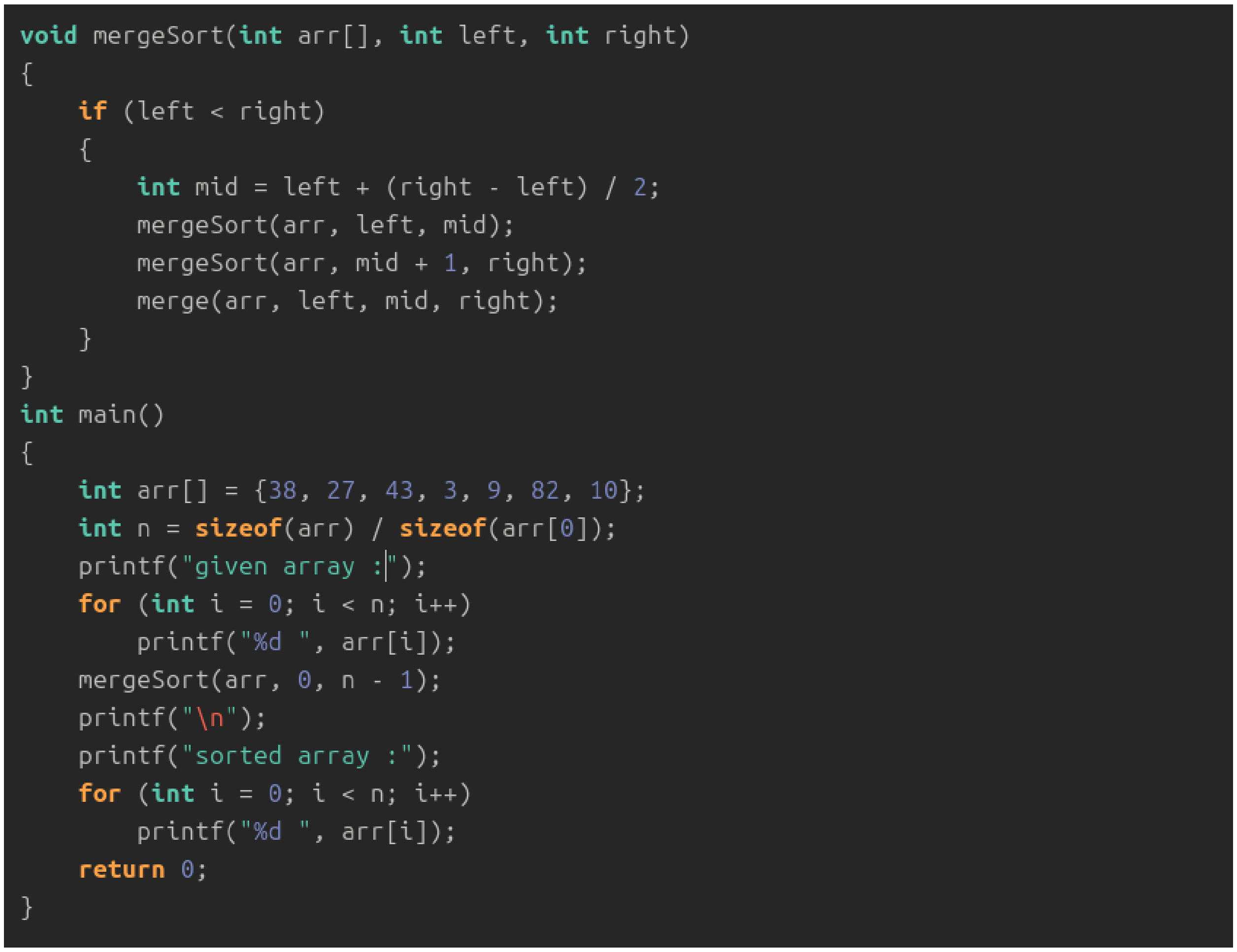
Algorithm:

1. If the array has **0 or 1 elements**, it is already sorted.
2. Divide the array into **two halves**.
3. Recursively apply merge sort to both halves.
4. Merge the two sorted halves into one sorted array.

Code:

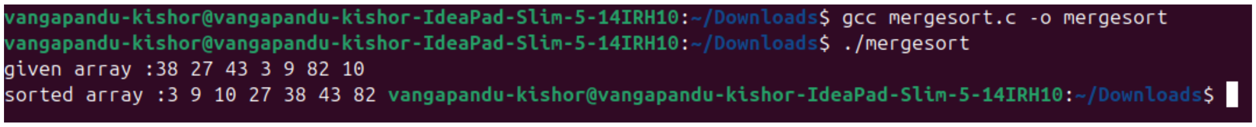


```c
#include <stdio.h>

void merge(int arr[], int left, int mid, int right)
{
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

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



```c
void mergeSort(int arr[], int left, int right)
{
    if (left < right)
    {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main()
{
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    printf("given array :");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    mergeSort(arr, 0, n - 1);
    printf("\n");
    printf("sorted array :");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Output:



```
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ gcc mergesort.c -o mergesort
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ ./mergesort
given array :38 27 43 3 9 82 10
sorted array :3 9 10 27 38 43 82 vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$
```

2.QUICK SORT:

Quick Sort is a **divide-and-conquer sorting algorithm**.

It:

- Selects a **pivot element** from the array
- Partitions the array around the pivot

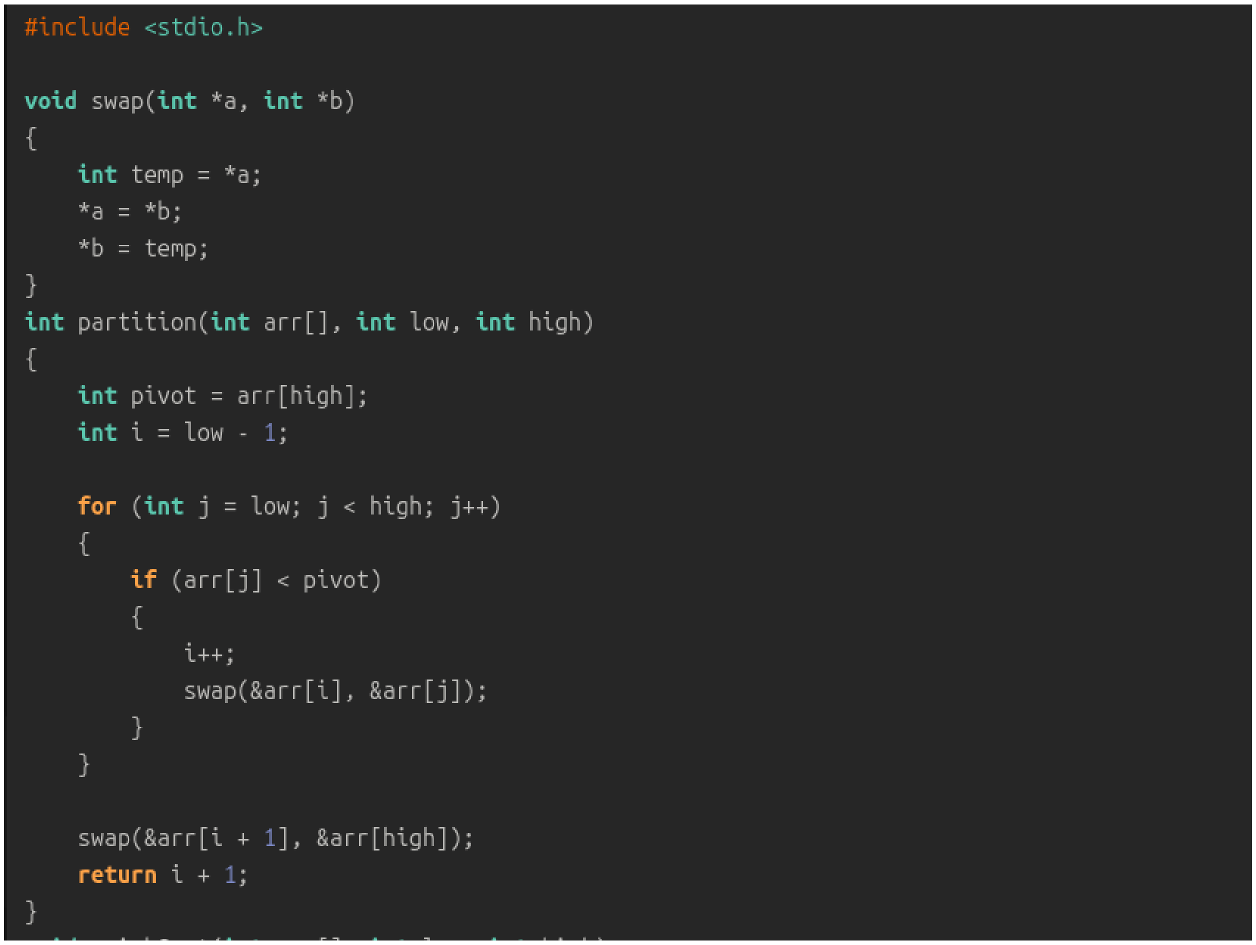
- Sorts the subarrays recursively

Quick Sort Algorithm (Step-by-Step):

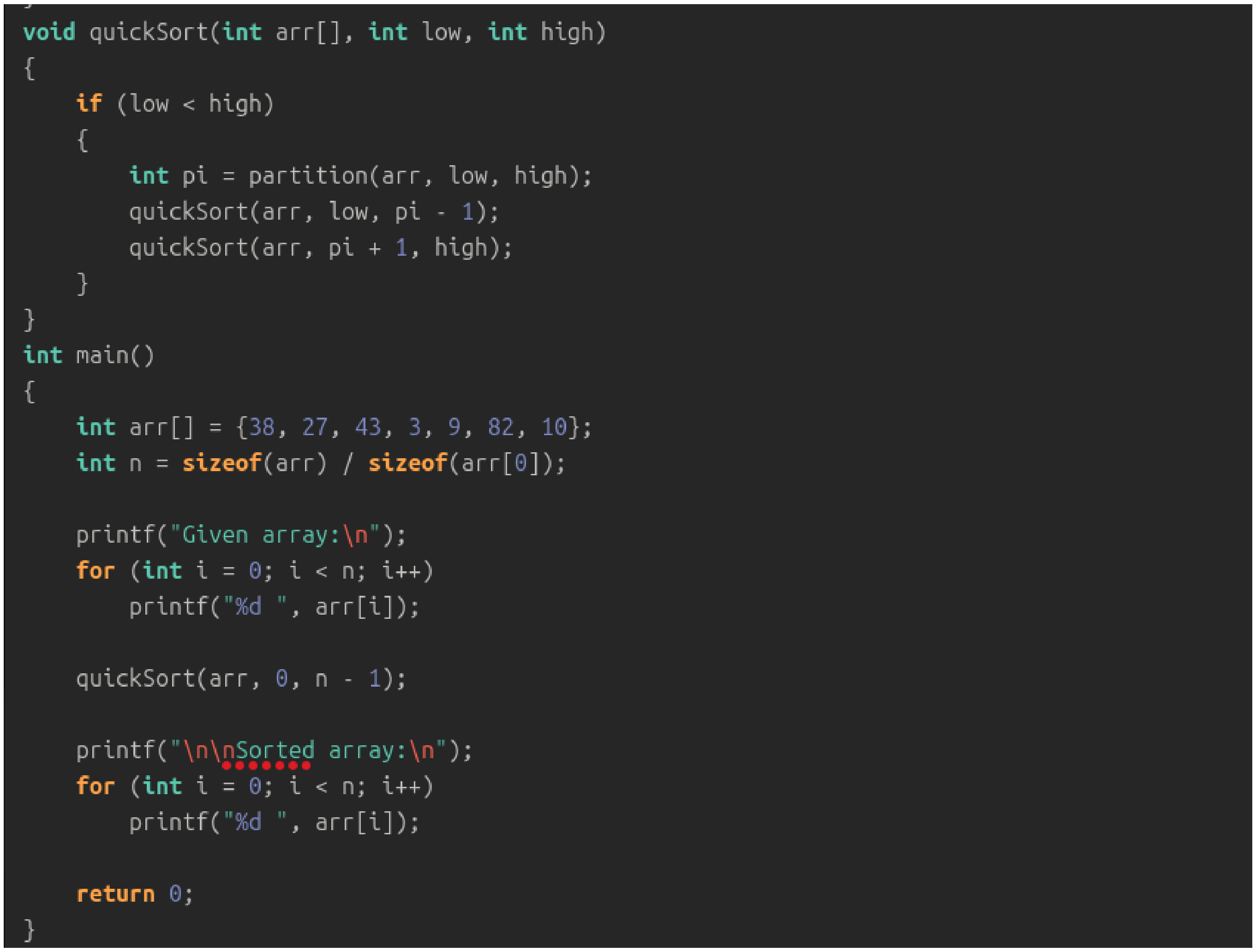
Algorithm:

- If the array has **0 or 1 elements**, it is already sorted.
- Choose an element from the array as the **pivot**.
- Rearrange the array so that:
    - Elements **less than the pivot** come before it.
    - Elements **greater than the pivot** come after it.
- Recursively apply **Quick Sort** to the left subarray.
- Recursively apply **Quick Sort** to the right subarray.
- The array becomes sorted when all recursive calls complete.

Code:

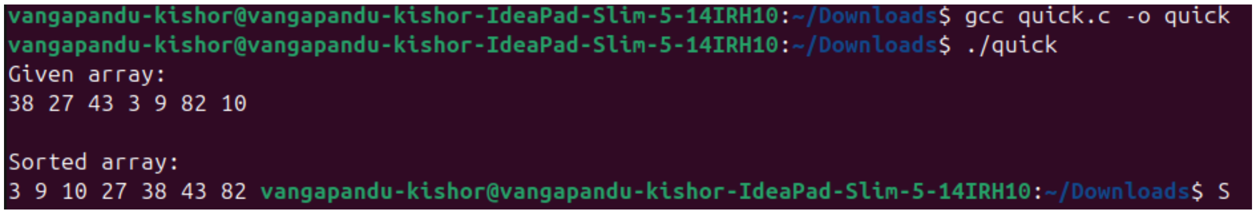


```
#include <stdio.h>

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];
    int i = low - 1;

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

    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
```



```c
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main()
{
    int arr[] = {38, 27, 43, 3, 9, 82, 10};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    quickSort(arr, 0, n - 1);

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

    return 0;
}
```

Output:



```
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ gcc quick.c -o quick
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ ./quick
Given array:
38 27 43 3 9 82 10

Sorted array:
3 9 10 27 38 43 82 vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ S
```

3.BST (BINARY SEARCH TREE)

A **Binary Search Tree (BST)** is a type of **binary tree** used for efficient searching, insertion, and deletion.

In a BST:

- The **left child** contains values **less than** the parent node.
- The **right child** contains values **greater than** the parent node.
- This property is true for **every node** in the tree.

Properties of Binary Search Tree:

- Each node has **at most two children**.
- Left subtree values < Root value
- Right subtree values > Root value
- No duplicate elements (in standard BST)
- In-order traversal gives elements in **sorted order**.

Operations on Binary Search Tree

1. Insertion:

- Compare the value with the root.
- If smaller, insert into the left subtree.
- If larger, insert into the right subtree.
- Repeat until the correct position is found.

2. Searching:

- Start from the root.
- If the value matches, search ends.
- If smaller, move to the left subtree.
- If larger, move to the right subtree.

3. Deletion:

Three cases:

1. Node with **no child** (leaf)
2. Node with **one child**
3. Node with **two children** (replace with inorder successor or predecessor)

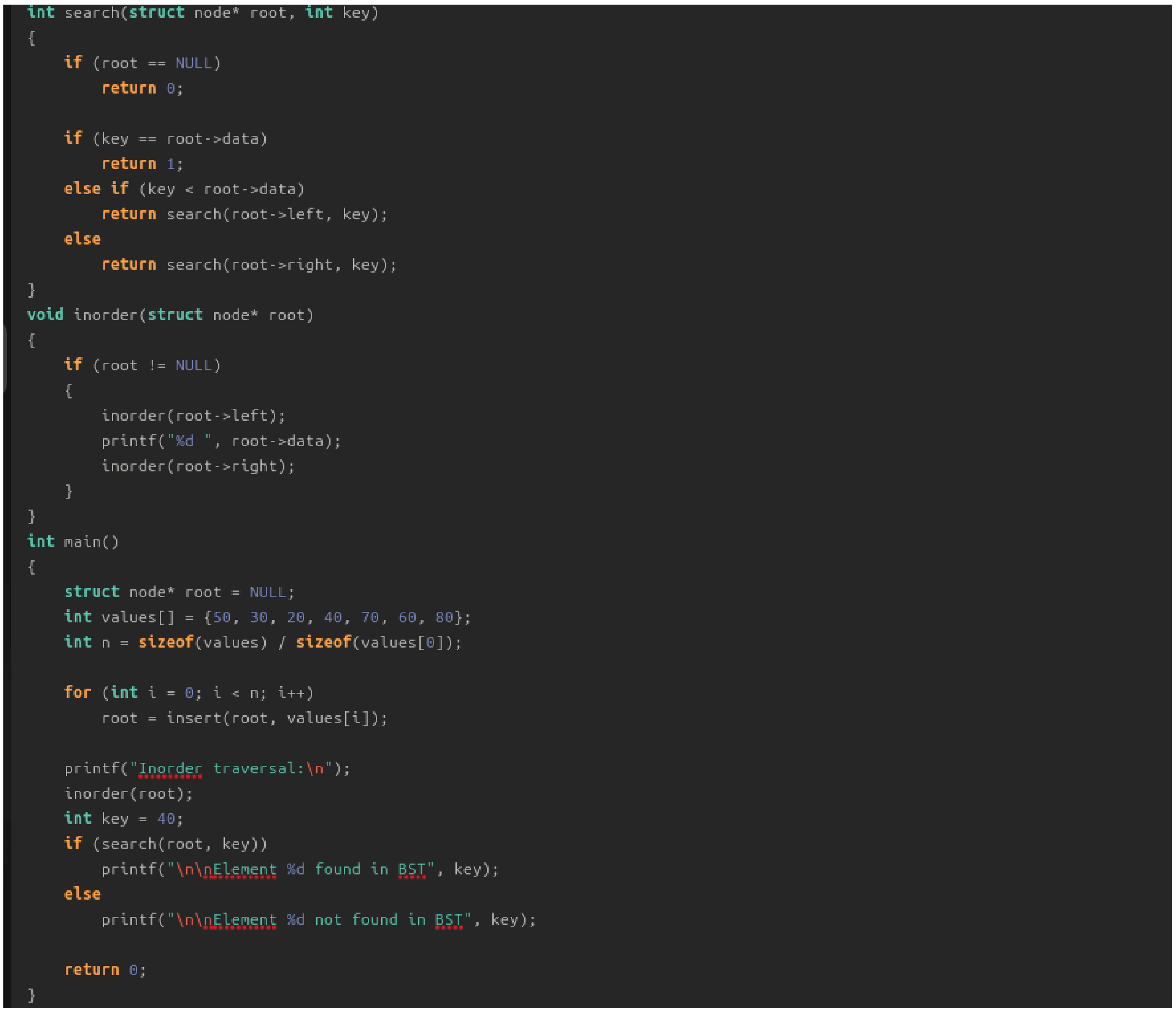
Traversal Methods:

- **In-order** (Left → Root → Right) → Sorted order

- **Pre-order** (Root → Left → Right)
- **Post-order** (Left → Right → Root)

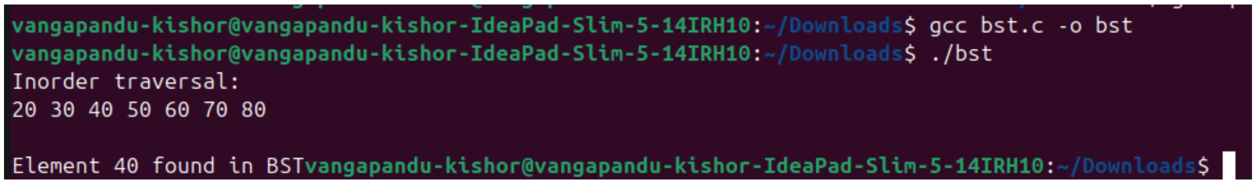
Code:

```c
#include <stdio.h>
#include <stdlib.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
struct node* createNode(int value)
{
    struct node* newNode = (struct node*)malloc(sizeof(struct node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
struct node* insert(struct 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);

    return root;
}
```



```c
int search(struct node* root, int key)
{
    if (root == NULL)
        return 0;

    if (key == root->data)
        return 1;
    else if (key < root->data)
        return search(root->left, key);
    else
        return search(root->right, key);
}
void inorder(struct node* root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
int main()
{
    struct node* root = NULL;
    int values[] = {50, 30, 20, 40, 70, 60, 80};
    int n = sizeof(values) / sizeof(values[0]);

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

    printf("Inorder traversal:\n");
    inorder(root);
    int key = 40;
    if (search(root, key))
        printf("\n\nElement %d found in BST", key);
    else
        printf("\n\nElement %d not found in BST", key);

    return 0;
}
```

Output:



```
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ gcc bst.c -o bst
vangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$ ./bst
Inorder traversal:
20 30 40 50 60 70 80

Element 40 found in BSTvangapandu-kishor@vangapandu-kishor-IdeaPad-Slim-5-14IRH10:~/Downloads$
```