SRM Institute of Science and Technology
Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204
Department of Computer Applications
Circular – 2023-24
BCA DS 2nd Sem

Fundamentals of Data Structures and Algorithms (UDS23202J)

Lab Manual

Lab 1: Recursion

Title: Recursion

Aim: To write and execute a program to demonstrate recursion.

Procedure:

1. Write a C/C++/Java program to implement a recursive function (e.g., factorial, Fibonacci sequence, Tower of Hanoi).
2. Compile the program.
3. Execute the program and provide input.
4. Observe the output and verify the recursive calls.

Source Code:

```
#include <iostream>
using namespace std;

int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}

int main() {
    int num;
    cout << "Enter a non-negative number: ";
    cin >> num;
    if(num < 0) {
      cout << "Factorial is not defined for negative numbers." << endl;
    } else {
        cout << "Factorial of " << num << " is " << factorial(num) << endl;
    }
    return 0;
}
```

Input: 5

Expected Output: Factorial of 5 is 120

Lab 2: Arrays

Title: Arrays

Aim: To implement various operations on arrays (e.g., insertion, deletion, searching, sorting).

Procedure:

1. Write a C/C++/Java program to perform array operations.
2. Implement functions for insertion, deletion, searching (linear, binary), and sorting (bubble sort, selection sort).
3. Compile and execute the program.
4. Provide an array and operation choice as input.
5. Observe the output after each operation.

Source Code:

```
#include <iostream>
using namespace std;

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

void insertElement(int arr[], int &n, int pos, int val) {
    if (pos > n || pos < 0) {
        cout << "Invalid position" << endl;
        return;
    }
    for (int i = n; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[pos] = val;
    n++;
}

void deleteElement(int arr[], int &n, int pos) {
    if (pos >= n || pos < 0) {
        cout << "Invalid position" << endl;
        return;
    }
    for (int i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];
    n--;
}

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

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
}
```

```cpp
int main() {
    int arr[100] = {1, 5, 2, 8, 3};
    int n = 5;
    int choice, val, pos;

    cout << "Original array: ";
    printArray(arr, n);

    cout << "Enter operation (1-Insert, 2-Delete, 3-Search, 4-Sort): ";
    cin >> choice;

    switch (choice) {
        case 1:
            cout << "Enter position and value: ";
            cin >> pos >> val;
            insertElement(arr, n, pos, val);
            cout << "Array after insertion: ";
            printArray(arr, n);
            break;
        case 2:
            cout << "Enter position: ";
            cin >> pos;
            deleteElement(arr, n, pos);
            cout << "Array after deletion: ";
            printArray(arr, n);
            break;
        case 3:
            cout << "Enter value to search: ";
            cin >> val;
            int index = linearSearch(arr, n, val);
            if (index != -1)
                cout << "Value found at index " << index << endl;
            else
                cout << "Value not found" << endl;
            break;
        case 4:
            bubbleSort(arr, n);
            cout << "Sorted array: ";
            printArray(arr, n);
            break;
        default:
            cout << "Invalid choice" << endl;
    }
    return 0;
}
```

Input:

- Operation: 1 (Insert)
- Position: 2
- Value: 10

Expected Output:

- Original array: 1 5 2 8 3
- Array after insertion: 1 5 10 2 8 3

Lab 3: Linked List

Title: Linked List

Aim: To implement a singly linked list and perform operations like insertion, deletion, and traversal.

Procedure:

1. Define a structure for a linked list node.
2. Implement functions for creating, inserting, deleting, and traversing the linked list.
3. Write a main function to test these operations.
4. Compile and execute the program.
5. Provide input for the operations.
6. Observe the output.

Source Code:

```
#include <iostream>
using namespace std;

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

Node* head = NULL;

void insertAtBeginning(int val) {
    Node* newNode = new Node;
    newNode->data = val;
    newNode->next = head;
    head = newNode;
}

void insertAtEnd(int val) {
    Node* newNode = new Node;
    newNode->data = val;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL)
        temp = temp->next;
    temp->next = newNode;
}

void deleteNode(int val) {
    if (head == NULL)
        return;
    if (head->data == val) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL && temp->next->data != val)
```

```
        temp = temp->next;
    if (temp->next == NULL) {
        cout << "Value not found" << endl;
        return;
    }
    Node* toDelete = temp->next;
    temp->next = temp->next->next;
    delete toDelete;
}

void displayList() {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    insertAtEnd(1);
    insertAtEnd(2);
    insertAtEnd(3);
    insertAtBeginning(0);
    cout << "Linked List: ";
    displayList();
    deleteNode(2);
    cout << "Linked List after deleting 2: ";
    displayList();
    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

Linked List: 0 1 2 3

Linked List after deleting 2: 0 1 3

Lab 4: Stack and its Applications

Title: Stack and its Applications

Aim: To implement a stack and demonstrate its applications (e.g., parenthesis matching, expression evaluation).

Procedure:

1. Implement a stack using an array or linked list.
2. Implement push, pop, and isEmpty operations.
3. Write a program to demonstrate an application of the stack. Common examples include:

   Parenthesis matching: Check if parentheses in an expression are balanced.

   Expression evaluation: Convert infix to postfix and evaluate.

4. Compile and execute the program.
5. Provide an expression as input.
6. Observe if the parentheses are balanced or the evaluated result.

Source Code:

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

bool isMatching(char a, char b) {
    return ((a == '(' && b == ')') || (a == '{' && b == '}') || (a == '[' && b
== ']'));
}

bool areParenthesesBalanced(string expr) {
    stack<char> s;
    for (char c : expr) {
        if (c == '(' || c == '{' || c == '[')
            s.push(c);
        else if (c == ')' || c == '}' || c == ']') {
            if (s.empty() || !isMatching(s.top(), c))
                return false;
            s.pop();
        }
    }
    return s.empty();
}

int main() {
    string expression;
    cout << "Enter an expression with parentheses: ";
    cin >> expression;
    if (areParenthesesBalanced(expression))
        cout << "Parentheses are balanced" << endl;
    else
        cout << "Parentheses are not balanced" << endl;
    return 0;
}
```

Input: "{[()()]}"

Expected Output: Parentheses are balanced

Lab 5: Queue Implementation using Array and Pointers

Title: Queue Implementation using Array and Pointers

Aim: To implement a queue data structure using both arrays and pointers.

Procedure:

1. Implement a queue using an array (static implementation).
2. Implement a queue using pointers (dynamic implementation).
3. For each implementation, provide functions for enqueue, dequeue, isEmpty, and isFull (for array).
4. Write a main function to test both implementations.
5. Compile and execute the program.
6. Provide input for enqueue and dequeue operations.
7. Observe the output.

Source Code:

```
#include <iostream>
using namespace std;

// Queue implementation using array
const int MAX_SIZE = 100;
class ArrayQueue {
private:
    int arr[MAX_SIZE];
    int front, rear;
public:
    ArrayQueue() {
        front = -1;
        rear = -1;
    }

    bool isEmpty() {
        return front == -1 && rear == -1;
    }

    bool isFull() {
        return rear == MAX_SIZE - 1;
    }

    void enqueue(int val) {
        if (isFull()) {
            cout << "Queue is full" << endl;
            return;
        }
        if (isEmpty()) {
            front = rear = 0;
        } else {
            rear++;
        }
        arr[rear] = val;
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1; // Or throw an exception
        }
```

```
        int val = arr[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front++;
        }
        return val;
    }

    void display() {
      if(isEmpty()){
        cout << "Queue is empty" << endl;
        return;
      }
      for(int i = front; i <= rear; i++){
        cout << arr[i] << " ";
      }
      cout << endl;
    }
};

// Queue implementation using pointers
struct Node {
    int data;
    Node* next;
};

class PointerQueue {
private:
    Node* front, * rear;
public:
    PointerQueue() {
        front = NULL;
        rear = NULL;
    }

    bool isEmpty() {
        return front == NULL;
    }

    void enqueue(int val) {
        Node* newNode = new Node;
        newNode->data = val;
        newNode->next = NULL;
        if (isEmpty()) {
            front = rear = newNode;
            return;
        }
        rear->next = newNode;
        rear = newNode;
    }

    int dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty" << endl;
            return -1; // Or throw an exception
        }
        int val = front->data;
        Node* temp = front;
        front = front->next;
        if (front == NULL)
            rear = NULL;
        delete temp;
        return val;
    }

    void display() {
```

```
      if(isEmpty()){
        cout << "Queue is empty" << endl;
        return;
      }
      Node* temp = front;
      while(temp != NULL){
        cout << temp->data << " ";
        temp = temp->next;
      }
      cout << endl;
    }
};

int main() {
    ArrayQueue aq;
    aq.enqueue(10);
    aq.enqueue(20);
    aq.enqueue(30);
    cout << "Array Queue: ";
    aq.display();
    cout << "Dequeued: " << aq.dequeue() << endl;
    cout << "Array Queue after dequeue: ";
    aq.display();

    PointerQueue pq;
    pq.enqueue(100);
    pq.enqueue(200);
    pq.enqueue(300);
    cout << "Pointer Queue: ";
    pq.display();
    cout << "Dequeued: " << pq.dequeue() << endl;
    cout << "Pointer Queue after dequeue: ";
    pq.display();

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

Array Queue: 10 20 30

Dequeued: 10

Array Queue after dequeue: 20 30

Pointer Queue: 100 200 300

Dequeued: 100

Pointer Queue after dequeue: 200 300

Lab 6: Implementation of Binary Tree using Arrays

Title: Implementation of Binary Tree using Arrays

Aim: To implement a binary tree using an array.

Procedure:

1. Represent a binary tree using an array. The root is at index 0, left child of node i is at 2i+1, and right child is at 2i+2.
2. Implement functions for inserting a node, and displaying the tree. Display can be inorder, preorder, or postorder.
3. Write a main function to create a binary tree and perform operations.
4. Compile and execute the program.
5. Provide input for the tree structure.
6. Observe the output.

Source Code:

```
#include <iostream>
using namespace std;

const int MAX_SIZE = 100;
int tree[MAX_SIZE];
int nextFree = 0;

void insertNode(int val) {
    if (nextFree >= MAX_SIZE) {
        cout << "Tree is full" << endl;
        return;
    }
    tree[nextFree] = val;
    nextFree++;
}

void displayTree(int index, int level = 0) {
    if (index >= nextFree)
        return;

    // Right subtree first (for better visualization)
    displayTree(2 * index + 2, level + 1);

    // Print the current node with indentation for level
    for (int i = 0; i < level * 4; i++) // Adjust spacing as needed
        cout << "  ";
    cout << tree[index] << endl;

    // Left subtree
    displayTree(2 * index + 1, level + 1);
}

int main() {
    for (int i = 0; i < MAX_SIZE; i++)
      tree[i] = -1; // Initialize the tree array; -1 can represent null

    insertNode(1);      // Root
    insertNode(2);      // Left child of 1
    insertNode(3);      // Right child of 1
    insertNode(4);      // Left child of 2
    insertNode(5);      // Right child of 2
```

```
    insertNode(6);     // Left child of 3
    insertNode(7);     // Right child of 3

    cout << "Binary Tree (visual representation):\n";
    displayTree(0);

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

```
Binary Tree (visual representation):
        7
    3
        6
1
        5
    2
        4
```

Lab 7: Tree Traversals

Title: Tree Traversals

Aim: To implement and demonstrate different tree traversal techniques (inorder, preorder, postorder).

Procedure:

1. Represent a binary tree using nodes and pointers.
2. Implement recursive functions for inorder, preorder, and postorder traversals.
3. Write a main function to create a binary tree and call the traversal functions.
4. Compile and execute the program.
5. Observe the output for each traversal.

Source Code:

```
#include <iostream>
using namespace std;

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

Node* createNode(int val) {
    Node* newNode = new Node;
    newNode->data = val;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

void inorderTraversal(Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        cout << root->data << " ";
        inorderTraversal(root->right);
    }
}

void preorderTraversal(Node* root) {
    if (root != NULL) {
        cout << root->data << " ";
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}

void postorderTraversal(Node* root) {
    if (root != NULL) {
        postorderTraversal(root->left);
        postorderTraversal(root->right);
        cout << root->data << " ";
    }
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
```

```
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);
    root->right->left = createNode(6);
    root->right->right = createNode(7);

    cout << "Inorder Traversal: ";
    inorderTraversal(root);
    cout << endl;

    cout << "Preorder Traversal: ";
    preorderTraversal(root);
    cout << endl;

    cout << "Postorder Traversal: ";
    postorderTraversal(root);
    cout << endl;

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

Inorder Traversal: 4 2 5 1 6 3 7

Preorder Traversal: 1 2 4 5 3 6 7

Postorder Traversal: 4 5 2 6 7 3 1

Lab 8: Implementation of BST Heap Data Structure

Title: Implementation of BST and Heap Data Structure

Aim: To implement a Binary Search Tree (BST) and a Heap data structure.

Procedure:

1. Implement a BST with insert, delete, and search operations.
2. Implement a Heap (min-heap or max-heap) with insert and deleteMin/deleteMax operations.
3. Write a main function to test both implementations.
4. Compile and execute the program.
5. Provide input for operations on the BST and Heap.
6. Observe the output.

Source Code:

```
#include <iostream>
#include <vector>
#include <climits> // For INT_MIN and INT_MAX
using namespace std;

// Binary Search Tree (BST) Implementation
struct BSTNode {
    int data;
    BSTNode* left;
    BSTNode* right;
};

BSTNode* createBSTNode(int val) {
    BSTNode* newNode = new BSTNode;
    newNode->data = val;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}

BSTNode* insertBST(BSTNode* root, int val) {
    if (root == NULL)
        return createBSTNode(val);
    if (val < root->data)
        root->left = insertBST(root->left, val);
    else if (val > root->data)
        root->right = insertBST(root->right, val);
    return root;
}

BSTNode* searchBST(BSTNode* root, int val) {
    if (root == NULL || root->data == val)
        return root;
    if (val < root->data)
        return searchBST(root->left, val);
    else
        return searchBST(root->right, val);
}

BSTNode* findMin(BSTNode* node) {
    while (node->left != NULL)
        node = node->left;
```

```
    return node;
}

BSTNode* deleteBST(BSTNode* root, int val) {
    if (root == NULL)
        return root;
    if (val < root->data)
        root->left = deleteBST(root->left, val);
    else if (val > root->data)
        root->right = deleteBST(root->right, val);
    else {
        if (root->left == NULL) {
            BSTNode* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == NULL) {
            BSTNode* temp = root->left;
            delete root;
            return temp;
        }
        BSTNode* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteBST(root->right, temp->data);
    }
    return root;
}

void inorderTraversalBST(BSTNode* root) {
    if (root != NULL) {
        inorderTraversalBST(root->left);
        cout << root->data << " ";
        inorderTraversalBST(root->right);
    }
}

// Max Heap Implementation
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
        largest = left;
    if (right < n && arr[right] > arr[largest])
        largest = right;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}

void insertHeap(vector<int>& arr, int val) {
    arr.push_back(val);
    int i = arr.size() - 1;
    while (i > 0 && arr[(i - 1) / 2] < arr[i]) {
        swap(arr[i], arr[(i - 1) / 2]);
        i = (i - 1) / 2;
    }
}

int deleteMax(vector<int>& arr) {
    if (arr.empty())
        return INT_MIN;
    int maxVal = arr[0];
    arr[0] = arr.back();
    arr.pop_back();
    heapify(arr, arr.size(), 0);
    return maxVal;
```

```
}

void displayHeap(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << endl;
}

int main() {
    // BST operations
    BSTNode* root = NULL;
    root = insertBST(root, 50);
    insertBST(root, 30);
    insertBST(root, 20);
    insertBST(root, 40);
    insertBST(root, 70);
    insertBST(root, 60);
    insertBST(root, 80);

    cout << "Inorder traversal of BST: ";
    inorderTraversalBST(root);
    cout << endl;

    cout << "Search 60 in BST: " << (searchBST(root, 60) != NULL ? "Found" :
"Not Found") << endl;root = deleteBST(root, 20);
    cout << "Inorder traversal of BST after deleting 20: ";
    inorderTraversalBST(root);
    cout << endl;

    // Heap operations
    vector<int> heap;
    insertHeap(heap, 10);
    insertHeap(heap, 5);
    insertHeap(heap, 17);
    insertHeap(heap, 20);
    insertHeap(heap, 1);

    cout << "Max Heap: ";
    displayHeap(heap);

    cout << "Deleted max: " << deleteMax(heap) << endl;
    cout << "Max Heap after deletion: ";
    displayHeap(heap);

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

Inorder traversal of BST: 20 30 40 50 60 70 80

Search 60 in BST: Found

Inorder traversal of BST after deleting 20: 30 40 50 60 70 80

Max Heap: 20 17 10 5 1

Deleted max: 20

Max Heap after deletion: 17 5 10 1

Lab 9: Heap Implementation

Title: Heap Implementation

Aim: To implement a heap data structure (min-heap or max-heap).

Procedure:

1. Implement a heap using an array or a vector.
2. Implement functions for insert, deleteMin/deleteMax, and display.
3. Write a main function to test the heap implementation.
4. Compile and execute the program.
5. Provide input for heap operations.
6. Observe the output.

Source Code: (See Lab 8. The Heap implementation is already provided there.)

Input: (See Lab 8)

Expected Output: (See Lab 8)

Lab 10: Implementation of Bubble and Insertion Sort

Title: Implementation of Bubble and Insertion Sort

Aim: To implement bubble sort and insertion sort algorithms.

Procedure:

1. Implement the bubble sort algorithm.
2. Implement the insertion sort algorithm.
3. Write a main function to test both sorting algorithms.
4. Compile and execute the program.
5. Provide an unsorted array as input.
6. Observe the sorted output.

Source Code:

```
#include <iostream>
#include <vector>
using namespace std;

void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
        }
    }
}

void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> arr1 = {5, 1, 4, 2, 8};
    cout << "Original array: ";
    printArray(arr1);
    bubbleSort(arr1);
    cout << "Sorted array (Bubble Sort): ";
    printArray(arr1);

    vector<int> arr2 = {5, 1, 4, 2, 8};
     cout << "Original array: ";
    printArray(arr2);
```

```
    insertionSort(arr2);
    cout << "Sorted array (Insertion Sort): ";
    printArray(arr2);

    return 0;
}
```

Input: 5 1 4 2 8

Expected Output:

Original array: 5 1 4 2 8

Sorted array (Bubble Sort): 1 2 4 5 8

Original array: 5 1 4 2 8

Sorted array (Insertion Sort): 1 2 4 5 8

Lab 11: Implementation of Quick Sort and Merge Sort; Introduction to Searching

Title: Implementation of Quick Sort and Merge Sort; Introduction to Searching

Aim: To implement quick sort and merge sort algorithms, and to introduce linear and binary search.

Procedure:

1. Implement the quick sort algorithm.
2. Implement the merge sort algorithm.
3. Implement linear search.
4. Implement binary search.
5. Write a main function to test the sorting and searching algorithms.
6. Compile and execute the program.
7. Provide an unsorted array for sorting and a key for searching.
8. Observe the sorted output and the search result.

Source Code:

```
#include <iostream>
#include <vector>
using namespace std;

// Quick Sort
int partition(vector<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);
}

void quickSort(vector<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);
    }
}

// Merge Sort
void merge(vector<int>& arr, int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    int i = 0, j = 0, k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
```

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

void mergeSort(vector<int>& arr, int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

// Linear Search
int linearSearch(const vector<int>& arr, int key) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == key)
            return i;
    }
    return -1;
}

// Binary Search (requires sorted array)
int binarySearch(const vector<int>& arr, int key) {
    int low = 0, high = arr.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

void printArray(const vector<int>& arr) {
    for (int val : arr)
        cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> arr = {5, 1, 4, 2, 8, 10, 7, 9, 3, 6};

    cout << "Original array: ";
    printArray(arr);

    vector<int> arr1 = arr; // Copy for Quick Sort
    quickSort(arr1, 0, arr1.size() - 1);
    cout << "Sorted array (Quick Sort): ";
    printArray(arr1);
```

```
    vector<int> arr2 = arr; // Copy for Merge Sort
    mergeSort(arr2, 0, arr2.size() - 1);
    cout << "Sorted array (Merge Sort): ";
    printArray(arr2);

    int key = 7;
    int index = linearSearch(arr, key);
    if (index != -1)
        cout << key << " found at index " << index << " (Linear Search)" <<
endl;
    else
        cout << key << " not found (Linear Search)" << endl;

    sort(arr.begin(), arr.end()); // Sort for binary search
    index = binarySearch(arr, key);
    if (index != -1)
        cout << key << " found at index " << index << " (Binary Search)" <<
endl;
    else
        cout << key << " not found (Binary Search)" << endl;

    return 0;
}
```

Input: 5 1 4 2 8 10 7 9 3 6, Search Key = 7

Expected Output:

Original array: 5 1 4 2 8 10 7 9 3 6

Sorted array (Quick Sort): 1 2 3 4 5 6 7 8 9 10

Sorted array (Merge Sort): 1 2 3 4 5 6 7 8 9 10

7 found at index 6 (Linear Search)

7 found at index 6 (Binary Search)

Lab 12: Implementation of Graph using Array

Title: Implementation of Graph using Array

Aim: To implement a graph data structure using an adjacency matrix (array).

Procedure:

1. Represent a graph using a 2D array (adjacency matrix).
2. Implement functions to add an edge, remove an edge, and display the graph.
3. Write a main function to create a graph and perform operations.
4. Compile and execute the program.
5. Provide input for the graph structure and operations.
6. Observe the output.

Source Code:

```
#include <iostream>
#include <vector>
using namespace std;

const int MAX_VERTICES = 100;
int adjMatrix[MAX_VERTICES][MAX_VERTICES];
int numVertices;

void initializeGraph() {
    for (int i = 0; i < MAX_VERTICES; i++)
        for (int j = 0; j < MAX_VERTICES; j++)
            adjMatrix[i][j] = 0;
    numVertices = 0;
}

void addVertex() {
    if (numVertices < MAX_VERTICES)
        numVertices++;
    else
        cout << "Graph is full" << endl;
}

void addEdge(int u, int v) {
    if (u >= 0 && u < numVertices && v >= 0 && v < numVertices) {
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // For an undirected graph
    } else
        cout << "Invalid vertices" << endl;
}

void removeEdge(int u, int v) {
    if (u >= 0 && u < numVertices && v >= 0 && v < numVertices) {
        adjMatrix[u][v] = 0;
        adjMatrix[v][u] = 0; // For an undirected graph
    } else
        cout << "Invalid vertices" << endl;
}

void displayGraph() {
    cout << "Adjacency Matrix:" << endl;
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++)
            cout << adjMatrix[i][j] << " ";
        cout << endl;
```

```
    }
}

int main() {
    initializeGraph();
    addVertex(); // 0
    addVertex(); // 1
    addVertex(); // 2
    addVertex(); // 3

    addEdge(0, 1);
    addEdge(0, 2);
    addEdge(1, 2);
    addEdge(2, 3);

    cout << "Graph:" << endl;
    displayGraph();

    removeEdge(1, 2);
    cout << "Graph after removing edge (1,2):" << endl;
    displayGraph();

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

Graph:

```
Adjacency Matrix:
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
```

Graph after removing edge (1,2):

```
Adjacency Matrix:
0 1 1 0
1 0 0 0
1 0 0 1
0 0 1 0
```

Lab 13: Implementation of Shortest Path Algorithm

Title: Implementation of Shortest Path Algorithm

Aim: To implement an algorithm to find the shortest path between two nodes in a graph (e.g., Dijkstra's algorithm).

Procedure:

1. Represent a graph using an adjacency matrix or adjacency list.
2. Implement Dijkstra's algorithm (or another shortest path algorithm).
3. Write a main function to create a graph and find the shortest path between two given nodes.
4. Compile and execute the program.
5. Provide input for the graph structure, source node, and destination node.
6. Observe the shortest path and its cost.

Source Code:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

const int MAX_VERTICES = 100;

void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int numVertices, int
startVertex, int endVertex) {
    int distance[MAX_VERTICES];
    bool visited[MAX_VERTICES];
    for (int i = 0; i < numVertices; i++) {
        distance[i] = INT_MAX;
        visited[i] = false;
    }
    distance[startVertex] = 0;

    for (int count = 0; count < numVertices - 1; count++) {
        int minDistance = INT_MAX, minIndex;
        for (int v = 0; v < numVertices; v++) {
            if (!visited[v] && distance[v] <= minDistance) {
                minDistance = distance[v];
                minIndex = v;
            }
        }
        int u = minIndex;
        visited[u] = true;
        for (int v = 0; v < numVertices; v++) {
            if (!visited[v] && graph[u][v] && distance[u] != INT_MAX &&
distance[u] + graph[u][v] < distance[v]) {
                distance[v] = distance[u] + graph[u][v];
            }
        }
    }

    cout << "Shortest distance from vertex " << startVertex << " to vertex " <<
endVertex << " is " << distance[endVertex] << endl;
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
```

```
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };
    int numVertices = 9;
    int startVertex = 0;
    int endVertex = 6;

    dijkstra(graph, numVertices, startVertex, endVertex);

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output: Shortest distance from vertex 0 to vertex 6 is 11

Lab 14: Implementation of Minimum Spanning Tree

Title: Implementation of Minimum Spanning Tree

Aim: To implement an algorithm to find a minimum spanning tree (MST) of a graph (e.g., Prim's algorithm, Kruskal's algorithm).

Procedure:

1. Represent a graph using an adjacency matrix or adjacency list.
2. Implement Prim's algorithm (or Kruskal's algorithm).
3. Write a main function to create a graph and find its MST.
4. Compile and execute the program.
5. Provide input for the graph structure.
6. Observe the edges of the MST and its total cost.

Source Code:

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

const int MAX_VERTICES = 100;

int primMST(int graph[MAX_VERTICES][MAX_VERTICES], int numVertices) {
    int parent[MAX_VERTICES];
    int key[MAX_VERTICES];
    bool mstSet[MAX_VERTICES];
    for (int i = 0; i < numVertices; i++) {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
    key[0] = 0;
    parent[0] = -1;

    for (int count = 0; count < numVertices - 1; count++) {
        int minKey = INT_MAX, minIndex;
        for (int v = 0; v < numVertices; v++) {
            if (!mstSet[v] && key[v] < minKey) {
                minKey = key[v];
                minIndex = v;
            }
        }
        int u = minIndex;
        mstSet[u] = true;
        for (int v = 0; v < numVertices; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] < key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    cout << "Edges of Minimum Spanning Tree:" << endl;
    int mstCost = 0;
    for (int i = 1; i < numVertices; i++) {
        cout << parent[i] << " - " << i << "  Weight: " << graph[i][parent[i]]
<< endl;
        mstCost += graph[i][parent[i]];
    }
```

```
    cout << "Total cost of MST: " << mstCost << endl;
    return mstCost;
}

int main() {
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0},
    };
    int numVertices = 5;

    primMST(graph, numVertices);

    return 0;
}
```

Input: (Implicit - within main function)

Expected Output:

```
Edges of Minimum Spanning Tree:
0 - 1  Weight: 2
1 - 2  Weight: 3
0 - 3  Weight: 6
1 - 4  Weight: 5
Total cost of MST: 16
```

Lab 15: To Implement Binary Search using Divide and Conquer Strategy

Title: Binary Search using Divide and Conquer

Aim: To implement the binary search algorithm using the divide and conquer strategy.

Procedure:

1. Implement the binary search algorithm recursively or iteratively.
2. Write a main function to test the binary search implementation.
3. Compile and execute the program.
4. Provide a sorted array and a key to search for as input.
5. Observe the index of the key if found, or a message if not found.

Source Code: (This is the same as the Binary Search in Lab 11)

```
#include <iostream>
#include <vector>
using namespace std;

// Binary Search (requires sorted array)
int binarySearch(const vector<int>& arr, int key) {
    int low = 0, high = arr.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid;
        if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int main() {
    vector<int> arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}; // Sorted array
    int key = 23;
    int index = binarySearch(arr, key);
    if (index != -1)
        cout << key << " found at index " << index << " (Binary Search)" <<
endl;
    else
        cout << key << " not found (Binary Search)" << endl;
    return 0;
}
```

Input: Array: 2, 5, 8, 12, 16, 23, 38, 56, 72, 91, Key: 23

Expected Output: 23 found at index 5 (Binary Search)