SRM Institute of Science and Technology

Delhi – Meerut Road, Sikri Kalan, Ghaziabad, Uttar Pradesh – 201204

Department of Computer Applications

Circular – 2024-25

BCA 2nd Sem

Data Structures and Algorithms (USA24201J)

Lab Manual

Lab 1: Recursion

Title: Recursion

Aim: To write and execute programs based on recursion.

Procedure:

1. Understand the concept of recursion (a function calling itself).
2. Identify the base case and the recursive step.
3. Write a C/C++/Python program to implement a recursive function (e.g., factorial, Fibonacci sequence, Tower of Hanoi).
4. Compile and execute the program.
5. Test with different inputs.

Source Code:

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

unsigned long long factorial(int n) {
  if (n == 0) {
    return 1; // Base case: 0! = 1
  } else {
    return n * factorial(n - 1); // Recursive step
  }
}

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

Input: A non-negative integer.

Expected Output: The factorial of the input integer.

Lab 2: Arrays

Title: Arrays

Aim: To implement programs using arrays for various operations.

Procedure:

1. Understand the concept of arrays (contiguous memory locations storing elements of the same data type).
2. Write a C/C++/Python program to perform array operations (e.g., insertion, deletion, searching, sorting, reversing).
3. Compile and execute the program.
4. Test with different array sizes and elements.

Source Code:

int main() { int arr[100], n, i, pos, val;

```
cout << "Enter the number of elements in the array: ";
cin >> n;

cout << "Enter the elements of the array: ";
for (i = 0; i < n; i++)
    cin >> arr[i];

cout << "Enter the position where you want to insert an element: ";
cin >> pos;

cout << "Enter the value you want to insert: ";
cin >> val;

if (pos <= 0 || pos > n + 1) {
    cout << "Invalid position!" << endl;
} else {
    // Make space for the new element
    for (i = n - 1; i >= pos - 1; i--)
        arr[i + 1] = arr[i];

    arr[pos - 1] = val; // Insert the element
    n++; // Increase the size of the array

    cout << "Array after insertion: ";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}
return 0;
```

}

</details>

Input: Array elements, position for insertion/deletion, value to insert/delete.

Expected Output: Modified array after the operation.

Lab 3: Implementation of Linked List

Title: Implementation of Linked List

Aim: To implement a linked list and perform operations on it.

Procedure:

1. Understand the concept of a linked list (nodes containing data and a pointer to the next node).
2. Write a C/C++/Python program to implement a linked list.
3. Implement operations like insertion, deletion, traversal, searching, and reversing the list.
4. Compile and execute the program.
5. Test with different data sets.

Source Code:

// Define the structure of a node in the linked list struct Node { int data; Node* next; };

// Function to insert a node at the beginning of the list void insertAtBeginning(Node** head, int data) { Node* newNode = new Node(); // Create a new node newNode->data = data; // Assign data to the new node newNode->next = *head; // Make the new node point to the current head *head = newNode; // Update the head to point to the new node }

// Function to display the linked list void displayList(Node* head) { Node* current = head; cout << "Linked List: "; while (current != NULL) { cout << current->data << " -> "; current = current->next; } cout << "NULL" << endl; }

int main() { Node* head = NULL; // Initialize an empty linked list

```
insertAtBeginning(&head, 3);
insertAtBeginning(&head, 7);
insertAtBeginning(&head, 1);
insertAtBeginning(&head, 9);

displayList(head); // Output: Linked List: 9 -> 1 -> 7 -> 3 -> NULL

return 0;
```

}

</details>

Input: Data to be inserted or deleted, position for insertion/deletion.

Expected Output: Modified linked list after the operation.

Lab 4: Implementation of Stack and Its Applications

Title: Implementation of Stack and Its Applications

Aim: To implement a stack data structure and use it to solve problems.

Procedure:

1. Understand the concept of a stack (LIFO - Last In, First Out).
2. Implement a stack using arrays or linked lists in C/C++/Python.
3. Implement stack operations: push, pop, peek, isEmpty, isFull.
4. Apply the stack to solve problems like:

    Balanced parenthesis checking.

    Expression evaluation (infix, postfix, prefix).

    Reversing a string.

5. Compile and execute the program.
6. Test with different inputs.

Source Code:

```
const int MAX_SIZE = 100; // Maximum size of the stack

// Structure to represent the stack struct Stack { int top; int items[MAX_SIZE]; };

// Function to initialize the stack void initialize(Stack* stack) { stack->top = -1; // Initialize top to -1 (empty stack) }

// Function to check if the stack is empty bool isEmpty(Stack* stack) { return (stack->top == -1); }

// Function to check if the stack is full bool isFull(Stack* stack) { return (stack->top == MAX_SIZE - 1); }

// Function to push an element onto the stack void push(Stack* stack, int value) { if (isFull(stack)) { cout << "Stack is full! Cannot push " << value << endl; } else { stack->items[++stack->top] = value; // Increment top and push the value cout << value << " pushed onto the stack" << endl; } }

// Function to pop an element from the stack int pop(Stack* stack) { if (isEmpty(stack)) { cout << "Stack is empty! Cannot pop" << endl; return -1; // Return a sentinel value to indicate error } else { int value = stack->items[stack->top--]; // Get the top element and decrement top cout << value << " popped from the stack" << endl; return value; } }

// Function to peek at the top element of the stack int peek(Stack* stack) { if (isEmpty(stack)) { cout << "Stack is empty! Cannot peek" << endl; return -1; // Return a sentinel value to indicate error } else { return stack->items[stack->top]; // Return the top element } }

int main() { Stack myStack; initialize(&myStack);
```

```
  push(&myStack, 5);
  push(&myStack, 10);
  push(&myStack, 15);

  cout << "Top element: " << peek(&myStack) << endl; // Output: Top
element: 15

  pop(&myStack);
  pop(&myStack);
  pop(&myStack);

  if(!isEmpty(&myStack))
    cout << "Top element: " << peek(&myStack) << endl;
  else
    cout<< "Stack is empty"<<endl;

  return 0;


}

</details>
```

Input: Data for push operation; for application problems, input varies (e.g., a string for parenthesis checking, an expression for evaluation).

Expected Output: Stack operations: pushed/popped elements. Application problems: the result of the problem (e.g., "balanced" or "unbalanced," the evaluated expression).

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. Understand the concept of a queue (FIFO - First In, First Out).
2. Implement a queue using an array in C/C++/Python.
3. Implement a queue using pointers (linked list) in C/C++/Python.
4. Implement queue operations: enqueue, dequeue, peek, isEmpty, isFull (for array implementation).
5. Compile and execute the program.
6. Test with different data sets.

Source Code:

const int MAX_SIZE = 100;

struct Queue { int front, rear; int items[MAX_SIZE]; };

void initialize(Queue* queue) { queue->front = -1; queue->rear = -1; }

bool isEmpty(Queue* queue) { return (queue->front == -1 && queue->rear == -1); }

bool isFull(Queue* queue) { return (queue->rear == MAX_SIZE - 1); }

void enqueue(Queue* queue, int value) { if (isFull(queue)) { cout << "Queue is full! Cannot enqueue " << value << endl; } else { if (isEmpty(queue)) { queue->front = 0; // Enqueue the first element } queue->items[++queue->rear] = value; cout << value << " enqueued" << endl; } }

int dequeue(Queue* queue) { if (isEmpty(queue)) { cout << "Queue is empty! Cannot dequeue" << endl; return -1; } else { int value = queue->items[queue->front]; if (queue->front == queue->rear) { // Dequeue the last element queue->front = queue->rear = -1; } else { queue->front++; } cout << value << " dequeued" << endl; return value; } }

int peek(Queue* queue) { if (isEmpty(queue)) { cout << "Queue is empty! Cannot peek" << endl; return -1; } else { return queue->items[queue->front]; } }

int main() { Queue myQueue; initialize(&myQueue);

```
enqueue(&myQueue, 5);
enqueue(&myQueue, 10);
enqueue(&myQueue, 15);

cout << "Front element: " << peek(&myQueue) << endl;

dequeue(&myQueue);
dequeue(&myQueue);
dequeue(&myQueue);

if(!isEmpty(&myQueue))
  cout << "Front element: " << peek(&myQueue) << endl;
```

```
    else
      cout << "Queue is empty" << endl;
    return 0;


  }

  </details>
```

Input: Data for enqueue operation.

Expected Output: Queue operations: enqueued/dequeued elements.

Lab 6: Implementation of Binary Tree Using Arrays

Title: Implementation of Binary Tree Using Arrays

Aim: To implement a binary tree using arrays.

Procedure:

1. Understand the concept of a binary tree and its representation using arrays.
2. Write a C/C++/Python program to represent a binary tree using an array.
3. Implement operations like inserting a node, deleting a node (limited in array representation), and traversing the tree.
4. Compile and execute the program.
5. Test with different tree structures.

Source Code:

const int MAX_SIZE = 100; int tree[MAX_SIZE]; // Array to represent the binary tree int size = 0; // Current number of nodes in the tree

// Function to insert a node into the binary tree void insertNode(int value, int index) { if (index >= MAX_SIZE) { cout << "Tree is full. Cannot insert " << value << endl; return; } if (index == 0 && size == 0){ tree[index] = value; size++; return; }

```
if(tree[(index-1)/2] == 0 && (index-1)/2 != index)
{
      cout<< "No parent for this node" <<endl;
      return;
}
tree[index] = value;
size++;
```

}

// Function to display the binary tree (array representation) void displayTree() { cout << "Binary Tree (Array Representation):" << endl; for (int i = 0; i < size; i++) { cout << "Index " << i << ": " << tree[i] << endl; } }

int main() { // Initialize the tree with 0 (representing empty nodes) for (int i = 0; i < MAX_SIZE; i++) { tree[i] = 0; } insertNode(1, 0); insertNode(2, 1); insertNode(3, 2); insertNode(4, 3); insertNode(5, 4); insertNode(6, 5); insertNode(7, 6); insertNode(8, 7); insertNode(9, 8); insertNode(10, 9);

```
displayTree();

return 0;
```

}

```
</details>
```

Input: Node values and their positions for insertion.

Expected Output: Array representation of the binary tree.

Lab 7: Implement All Three Types of Tree Traversals

Title: Implement All Three Types of Tree Traversals

Aim: To implement preorder, inorder, and postorder traversals of a binary tree.

Procedure:

1. Understand the different types of tree traversals (preorder, inorder, postorder).
2. Implement a binary tree using nodes and pointers in C/C++/Python.
3. Write functions for each traversal method (recursive or iterative).
4. Compile and execute the program.
5. Test with different binary tree structures.

Source Code:

// Structure for a binary tree node struct Node { int data; Node* left; Node* right; };

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

// Preorder traversal (Root-Left-Right) void preorderTraversal(Node* root) { if (root != NULL) { cout << root->data << " "; preorderTraversal(root->left); preorderTraversal(root->right); } }

// Inorder traversal (Left-Root-Right) void inorderTraversal(Node* root) { if (root != NULL) { inorderTraversal(root->left); cout << root->data << " "; inorderTraversal(root->right); } }

// Postorder traversal (Left-Right-Root) void postorderTraversal(Node* root) { if (root != NULL) { postorderTraversal(root->left); postorderTraversal(root->right); cout << root->data << " "; } }

int main() { // Create a sample binary tree 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 << "Preorder Traversal: ";
preorderTraversal(root);
cout << endl;

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

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

return 0;
```

}

</details>

Input: A binary tree structure.

Expected Output: The preorder, inorder, and postorder traversals of the tree.

Lab 8: Implementation of BST Heap Data Structure

Title: Implementation of BST Heap Data Structure

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

Procedure:

1. Understand the properties of a BST and a Heap.
2. Implement a BST with operations like insertion, deletion, and searching in C/C++/Python.
3. Implement a Heap (Min Heap or Max Heap) with operations like insertion, deletion, and heapify in C/C++/Python.
4. Compile and execute the program.
5. Test with various data sets.

Source Code:

```
// Structure for a BST node struct Node { int data; Node* left; Node* right; };

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

// Function to insert a node into the BST Node* insertBST(Node* root, int data) { if (root == NULL) { return createNode(data); } if (data < root->data) { root->left = insertBST(root->left, data); } else if (data > root->data) { root->right = insertBST(root->right, data); } return root; } //Function to find minimum node in BST Node* findMin(Node* root) { while (root->left != NULL) root = root->left; return root; }

// Function to delete a node from the BST Node* deleteBST(Node* root, int data) { if (root == NULL) { return root; } if (data < root->data) { root->left = deleteBST(root->left, data); } else if (data > root->data) { root->right = deleteBST(root->right, data); } else { // Node with only one child or no child if (root->left == NULL) { Node* temp = root->right; delete root; return temp; } else if (root->right == NULL) { Node* temp = root->left; delete root; return temp; } // Node with two children: Get the inorder successor (smallest // in the right subtree) Node* temp = findMin(root->right);

        // Copy the inorder successor's data to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteBST(root->right, temp->data);
    }
    return root;


}

// Function to search for a value in the BST bool searchBST(Node* root, int data) { if (root == NULL) { return false; } if (root->data == data) { return true; } else if (data < root->data) { return searchBST(root->left, data); } else { return searchBST(root->right, data); } }
```

// Inorder traversal of the BST (to print sorted order) void inorderTraversal(Node* root) { if (root != NULL) { inorderTraversal(root->left); cout << root->data << " "; inorderTraversal(root->right); } }

int main() { Node* root = NULL; root = insertBST(root, 50); root = insertBST(root, 30); root = insertBST(root, 20); root = insertBST(root, 40); root = insertBST(root, 70); root = insertBST(root, 60); root = insertBST(root, 80);

```
  cout << "Inorder traversal of the BST: ";
  inorderTraversal(root);
  cout << endl;

  cout << "Search for 60: " << (searchBST(root, 60) ? "Found" : "Not
Found") << endl;
  cout << "Search for 90: " << (searchBST(root, 90) ? "Found" : "Not
Found") << endl;

  root = deleteBST(root, 20);
  cout << "Inorder traversal after deleting 20: ";
  inorderTraversal(root);
  cout << endl;

  root = deleteBST(root, 30);
  cout << "Inorder traversal after deleting 30: ";
  inorderTraversal(root);
  cout << endl;
  root = deleteBST(root, 50);
  cout << "Inorder traversal after deleting 50: ";
  inorderTraversal(root);
  cout << endl;

  return 0;
```

}

```
</details>
```

Input: Data for insertion, deletion, and searching in the BST and Heap.

Expected Output: BST: The structure of the BST and the results of search/traversal operations. Heap: The structure of the Heap after insertions/deletions.

Lab 9: Implementation of Min and Max Heap

Title: Implementation of Min and Max Heap

Aim: To implement Min Heap and Max Heap data structures.

Procedure:

1. Understand the properties of Min Heap and Max Heap.
2. Implement a Min Heap in C/C++/Python.
3. Implement a Max Heap in C/C++/Python.
4. Implement operations like insertion, deletion, and heapify for both Min Heap and Max Heap.
5. Compile and execute the program.
6. Test with different data sets.

Source Code:

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

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

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

  // If largest is not root
  if (largest != i) {
      swap(arr[i], arr[largest]); // Swap root and largest
      maxHeapify(arr, n, largest);    // Recursively heapify the affected
sub-tree
  }
```

}

// Function to insert a key into the max heap void insertMaxHeap(vector& arr, int key) { int n = arr.size(); if (n == 0) { arr.push_back(key); // Insert the first element } else { arr.push_back(key); // Insert at the end int i = n; while (i != 0 && arr[i] > arr[(i - 1) / 2]) { swap(arr[i], arr[(i - 1) / 2]); i = (i - 1) / 2; } } }

// Function to delete the root (maximum element) from the max heap void deleteMaxHeap(vector& arr) { int n = arr.size(); if (n == 0) return; arr[0] = arr[n - 1]; // Move the last element to the root arr.pop_back(); // Remove the last element maxHeapify(arr, arr.size(), 0); // Heapify the root }

// Function to display the max heap void displayMaxHeap(const vector& arr) { cout << "Max Heap: "; for (int i = 0; i < arr.size(); i++) cout << arr[i] << " "; cout << endl; }

int main() { vector maxHeap;

```
  insertMaxHeap(maxHeap, 10);
```

```
    insertMaxHeap(maxHeap, 5);
    insertMaxHeap(maxHeap, 15);
    insertMaxHeap(maxHeap, 20);
    insertMaxHeap(maxHeap, 25);
    insertMaxHeap(maxHeap, 30);
    displayMaxHeap(maxHeap); // Output: 30 25 15 5 10 20

    deleteMaxHeap(maxHeap);
    displayMaxHeap(maxHeap); // Output: 25 20 15 5 10

    return 0;


}
```

</details>

Input: Data for insertion and deletion in the Min Heap and Max Heap.

Expected Output: The structure of the Min Heap and Max Heap after insertions/deletions.

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. Understand the working principles of Bubble Sort and Insertion Sort.
2. Write a C/C++/Python program to implement Bubble Sort.
3. Write a C/C++/Python program to implement Insertion Sort.
4. Compile and execute the programs.
5. Test with different input arrays (sorted, unsorted, reverse sorted).
6. Analyze the time complexity of both algorithms.

Source Code:

// Function to perform Bubble Sort void bubbleSort(vector& 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]); } } } }

// Function to display the array void displayArray(const vector& arr) { for (int i = 0; i < arr.size(); i++) cout << arr[i] << " "; cout << endl; }

int main() { vector arr = {64, 34, 25, 12, 22, 11, 90};

```
cout << "Unsorted array: ";
displayArray(arr);

bubbleSort(arr);

cout << "Sorted array (Bubble Sort): ";
displayArray(arr);

return 0;
```

}

```
</details>
```

Input: An unsorted array of integers.

Expected Output: The sorted array using Bubble Sort and Insertion Sort.

Lab 11: Implementation of Quick Sort and Merge Sort

Title: Implementation of Quick Sort and Merge Sort

Aim: To implement Quick Sort and Merge Sort algorithms.

Procedure:

1. Understand the working principles of Quick Sort and Merge Sort (divide-and-conquer algorithms).
2. Write a C/C++/Python program to implement Quick Sort (recursive).
3. Write a C/C++/Python program to implement Merge Sort (recursive).
4. Compile and execute the programs.
5. Test with different input arrays (sorted, unsorted, reverse sorted).
6. Analyze the time complexity of both algorithms.

Source Code:

// Function to partition the array for Quick Sort int partition(vector& arr, int low, int high) { int pivot = arr[high]; // Choose the last element as the pivot int i = (low - 1); // Index of smaller element

```
for (int j = low; j < high; j++) {
    // If current element is smaller than or equal to pivot
    if (arr[j] <= pivot) {
        i++; // Increment index of smaller element
        swap(arr[i], arr[j]);
    }
}
swap(arr[i + 1], arr[high]);
return (i + 1);
```

}

// Function to perform Quick Sort void quickSort(vector& arr, int low, int high) { if (low < high) { int pi = partition(arr, low, high); // pi is partitioning index quickSort(arr, low, pi - 1); // Recursively sort the left sub-array quickSort(arr, pi + 1, high); // Recursively sort the right sub-array } }

// Function to display the array void displayArray(const vector& arr) { for (int i = 0; i < arr.size(); i++) cout << arr[i] << " "; cout << endl; }

int main() { vector arr = {64, 34, 25, 12, 22, 11, 90};

```
cout << "Unsorted array: ";
displayArray(arr);

quickSort(arr, 0, arr.size() - 1);

cout << "Sorted array (Quick Sort): ";
displayArray(arr);

return 0;
```

}

</details>

Input: An unsorted array of integers.

Expected Output: The sorted array using Quick Sort and Merge Sort.

Lab 12: Linear Search and Binary Search

Title: Linear Search and Binary Search

Aim: To implement Linear Search and Binary Search algorithms.

Procedure:

1. Understand the working principles of Linear Search and Binary Search.
2. Write a C/C++/Python program to implement Linear Search.
3. Write a C/C++/Python program to implement Binary Search (requires a sorted array).
4. Compile and execute the programs.
5. Test with different input arrays (sorted and unsorted) and search keys.
6. Analyze the time complexity of both algorithms.

Source Code:

// Function to perform Binary Search (iterative) int binarySearch(const vector& arr, int key) { int low = 0, high = arr.size() - 1; while (low <= high) { int mid = low + (high - low) / 2; // More robust way to find mid if (arr[mid] == key) return mid; if (arr[mid] < key) low = mid + 1; else high = mid - 1; } return -1; // Key not found }

int main() { vector arr = {11, 22, 25, 34, 64, 90}; // Sorted array

```
int key = 25;
int index = binarySearch(arr, key);
if (index != -1)
    cout << "Element " << key << " found at index " << index << endl;
else
    cout << "Element " << key << " not found in the array" << endl;

key = 50;
index = binarySearch(arr, key);
if (index != -1)
    cout << "Element " << key << " found at index " << index << endl;
else
    cout << "Element " << key << " not found in the array" << endl;

return 0;
```

}

</details>

Input: A sorted/unsorted array of integers and a search key.

Expected Output: The index of the key if found, or a message indicating that the key is not found.

Lab 13: Implementation of Graph Using Array

Title: Implementation of Graph Using Array

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

Procedure:

1. Understand the concept of a graph (vertices and edges) and adjacency matrix representation.
2. Write a C/C++/Python program to represent a graph using a 2D array (adjacency matrix).
3. Implement operations like adding an edge, removing an edge, and displaying the graph.
4. Implement graph traversal algorithms (Depth-First Search (DFS) and Breadth-First Search (BFS)).
5. Compile and execute the program.
6. Test with different graph structures.

Source Code:

const int MAX_VERTICES = 100; // Maximum number of vertices

// Function to add an edge to the graph void addEdge(int graph[][MAX_VERTICES], int u, int v, bool isDirected) { graph[u][v] = 1; // Add edge from u to v if (!isDirected) { graph[v][u] = 1; // If graph is undirected, add edge from v to u as well } }

// Function to display the graph (adjacency matrix) void displayGraph(int graph[][MAX_VERTICES], int vertices) { cout << "Adjacency Matrix:" << endl; for (int i = 0; i < vertices; i++) { for (int j = 0; j < vertices; j++) { cout << graph[i][j] << " "; } cout << endl; } }

// Function to perform Depth First Search (DFS) void DFS(int graph[][MAX_VERTICES], int vertices, int startVertex, vector& visited) { cout << startVertex << " "; visited[startVertex] = true; for (int i = 0; i < vertices; i++) { if (graph[startVertex][i] == 1 && !visited[i]) { DFS(graph, vertices, i, visited); } } }

// Function to perform Breadth First Search (BFS) void BFS(int graph[][MAX_VERTICES], int vertices, int startVertex) { queue q; vector visited(vertices, false); // Initialize all vertices as not visited

```
q.push(startVertex);
visited[startVertex] = true;

while (!q.empty()) {
    int u = q.front();
    cout << u << " ";
    q.pop();

    for (int v = 0; v < vertices; v++) {
        if (graph[u][v] == 1 && !visited[v]) {
            q.push(v);
            visited[v] = true;
        }
    }
```

```
}
```

}

int main() { int vertices = 6; // Number of vertices in the graph int graph[MAX_VERTICES][MAX_VERTICES] = {0}; // Initialize adjacency matrix

```
addEdge(graph, 0, 1, false);
addEdge(graph, 0, 2, false);
addEdge(graph, 1, 3, false);
addEdge(graph, 2, 4, false);
addEdge(graph, 3, 4, false);
addEdge(graph, 3, 5, false);
addEdge(graph, 4, 5, false);

displayGraph(graph, vertices);

cout << "DFS starting from vertex 0: ";
vector<bool> visited(vertices, false);
DFS(graph, vertices, 0, visited);
cout << endl;

cout << "BFS starting from vertex 0: ";
BFS(graph, vertices, 0);
cout << endl;

return 0;
```

}

</details>

Input: Number of vertices, edges (pairs of vertices).

Expected Output: Adjacency matrix representation of the graph, DFS and BFS traversals.

Lab 14: Implementation of Shortest Path Algorithm

Title: Implementation of Shortest Path Algorithm

Aim: To implement an algorithm to find the shortest path between two vertices in a graph.

Procedure:

1. Understand the concept of shortest path algorithms (e.g., Dijkstra's algorithm, Bellman-Ford algorithm).
2. Implement Dijkstra's algorithm (for non-negative edge weights) in C/C++/Python.
3. (Optional) Implement Bellman-Ford algorithm (for graphs with negative edge weights) in C/C++/Python.
4. Represent the graph using an adjacency matrix or adjacency list.
5. Compile and execute the program.
6. Test with different graph structures and source-destination pairs.

Source Code:

const int MAX_VERTICES = 100;

// Function to find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree int minDistance(const vector& dist, const vector& sptSet, int vertices) { int min = INT_MAX, min_index; for (int v = 0; v < vertices; v++) { if (sptSet[v] == false && dist[v] <= min) { min = dist[v]; min_index = v; } } return min_index; }

// Function to implement Dijkstra's algorithm for finding shortest path from source to all other vertices void dijkstra(int graph[][MAX_VERTICES], int source, int vertices) { vector dist(vertices, INT_MAX); // Initialize distances to all vertices as infinite vector sptSet(vertices, false); // sptSet[i] is true if vertex i is included in shortest path tree

```
  dist[source] = 0; // Distance of source vertex from itself is always 0

  // Find shortest path for all vertices
  for (int count = 0; count < vertices - 1; count++) {
      int u = minDistance(dist, sptSet, vertices); // Pick the minimum
distance vertex from the set of vertices not yet processed
      sptSet[u] = true; // Mark the picked vertex as processed

      // Update dist values of the adjacent vertices of the picked vertex
      for (int v = 0; v < vertices; v++) {
          if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v]) {
              dist[v] = dist[u] + graph[u][v];
          }
      }
  }

  // Print the constructed distance array
  cout << "Vertex   Distance from Source" << endl;
  for (int i = 0; i < vertices; i++)
      cout << i << " \t\t" << dist[i] << endl;


}
```

int main() { int vertices = 9; 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} };

```
  dijkstra(graph, 0, vertices); // Find shortest path from source vertex 0
to all other vertices

  return 0;
```

}

```
</details>
```

Input: Graph representation (adjacency matrix or list), source and destination vertices.

Expected Output: The shortest path distance between the source and destination vertices.

Lab 15: Implementation of Minimum Spanning Tree

Title: Implementation of Minimum Spanning Tree

Aim: To implement an algorithm to find the Minimum Spanning Tree (MST) of a graph.

Procedure:

1. Understand the concept of Minimum Spanning Trees and algorithms to find them (e.g., Kruskal's algorithm, Prim's algorithm).
2. Implement either Kruskal's algorithm or Prim's algorithm in C/C++/Python.
3. Represent the graph using an adjacency matrix or adjacency list.
4. Compile and execute the program.
5. Test with different graph structures.

Source Code:

const int MAX_VERTICES = 100;

// Function to find the vertex with minimum key value, from the set of vertices not yet included in MST int minKey(const vector& key, const vector& mstSet, int vertices) { int min = INT_MAX, min_index; for (int v = 0; v < vertices; v++) { if (mstSet[v] == false && key[v] < min) { min = key[v]; min_index = v; } } return min_index; }

// Function to implement Prim's algorithm for finding Minimum Spanning Tree void primMST(int graph[][MAX_VERTICES], int vertices) { vector parent(vertices); // Array to store constructed MST vector key(vertices, INT_MAX); // Key values used to pick minimum weight edge in cut vector mstSet(vertices, false); // To represent set of vertices not yet included in MST

```
  key[0] = 0;      // Make key 0 so that this vertex is picked as first
vertex
  parent[0] = -1; // First node is always root of MST

  // The MST will have V vertices
  for (int count = 0; count < vertices - 1; count++) {
      int u = minKey(key, mstSet, vertices); // Pick the minimum key
vertex from the set of vertices not yet included in MST
      mstSet[u] = true; // Include the picked vertex in MST Set

      // Update key value and parent index of the adjacent vertices of the
picked vertex.
      // Consider only those vertices which are not yet included in MST
      for (int v = 0; v < vertices; v++) {
          if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v]) {
              parent[v] = u;
              key[v] = graph[u][v];
          }
      }
  }

  // Print the constructed MST
  cout << "Edge \tWeight\n";
  for (int i = 1; i < vertices; i++)
      cout << parent[i] << " - " << i << " \t" << graph[i][parent[i]] <<
endl;
```

}

int main() { int vertices = 9; 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} };

```
  primMST(graph, vertices);

  return 0;
```

}

</details>

Input: Graph representation (adjacency matrix or list).

Expected Output: The edges included in the Minimum Spanning Tree and their weights.