DAYANANDA SAGAR ACADEMY OF TECHNOLOGY AND MANAGEMENT

DEPARTMENT OF MASTER OF COMPUTER APPLICATIONS

PROBLEM-SOLVING USING C LAB MANUAL

Academic year 2024-2025 (odd semester)

Semester –I

Course code: **MMCPB16**

PROBLEM SOLVING USING C

Prepared by

Dr. ENOCH ARUL PRAKASH

Mrs. PRIYANKA MOHAN

Dr. Manjula Sanjay Dr. Ravishankar M

HOD-MCA Principal

Vision of the Institute

To be a centre of excellence in education, research & training and to produce citizens with exceptional leadership qualities to serve national and global needs

Mission of the Institute

To achieve our objectives in an environment that enhances creativity, innovation and scholarly pursuits while adhering to our vision.

Vision of MCA Department

Nurture Continuous Learning through research and innovations in the field of Computer Science, Technology and Applications, to build competent professionals.

Mission of MCA Department

- Create a learning environment to motivate students to build strong technology skills.
- Promote value based ethical practices in all facets of learning
- Instill Entrepreneurial collaborative thinking through structured interventions and industry participation.

Program Education Outcome (PEO's):

PEO1: Analyse real life problems, design computing systems appropriate to its solutions that are technically sound, economically feasible and socially acceptable.

PEO2: Exhibit professionalism, ethical attitude, communication skills, team work in their profession and adapt to current trends by engaging in lifelong learning.

PEO3: Demonstrate Leadership and Entrepreneurship Skills by incorporating organizational goals.

Program Outcome (PO's):

PO1 (Foundation Knowledge): Apply knowledge of mathematics, programming logic and coding fundamentals for solution architecture and problem solving.

PO2 (Problem Analysis): Identify, review, formulate and analyse problems for primarily focussing on customer requirements using critical thinking frameworks.

PO3 (Development of Solutions): Design, develop and investigate problems with as an innovative approach for solutions incorporating ESG/SDG goals.

PO4 (Modern Tool Usage): Select, adapt and apply modern computational tools such as development of algorithms with an understanding of the limitations including human biases.

PO5 (Individual and Teamwork): Function and communicate effectively as an individual or a team leader in diverse and multidisciplinary groups. Use methodologies such as agile.

PO6 (Project Management and Finance): Use the principles of project management such as scheduling, work breakdown structure and be conversant with the principles of Finance for profitable project management.

PO7 (Ethics): Commit to professional ethics in managing software projects with financial aspects. Learn to use new technologies for cyber security and insulate customers from malware

PO8 (Life-long learning): Change management skills and the ability to learn, keep up with contemporary technologies and ways of working.

Program Specific Outcomes (PSO's):

PSO1: The graduates of the Program will have skills to develop, deploy and maintain applications for desktop, web, mobile, cloud, and cross platforms using modern tools and technologies.

PSO2: The graduates of the program analyze the societal needs to provide novel solutions through technological-based research.

Course Outcomes

CO	Course Outcomes	RBT Level	Level Indicator
CO1	Understand the techniques for evaluating the given expression	Understand	L1, L2
CO2	Apply sorting/searching techniques and validate input/output for the given problem.	Apply	L3
CO3	Evaluate data structures like Stacks, Queues, Linked list , Trees and Graphs, its operations, and algorithms	Analyse	L4
CO4	Implement the algorithm to find whether the given graph is connected or not and conclude on the performance of the technique implemented	Evaluate	L5

Mapping of Course Outcomes with Program Outcomes and PSO

Correlation levels: 1-Slight (Low) 2-Moderate (Medium) 3-Substantial (High)

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	-	-	-	-	-	-	-	-	-	-	-	-	2	-
CO2	3	-	-	-	-	-	-	-	-	-	-	-	2	-
CO3	-	3	-	-	-	-	-	-	-	-	2	-	2	-
CO4	-	-	2		-	-	-	2	-	-	-	-	-	2

Computer Lab Rules and Regulations

DO's

- Come prepared to the Lab.
- Submit your Records to the faculty and sign in the Log Book on entering the Lab
- Observation books have to be brought for all the labs.
- Backlog exercises to be executed after completing regular exercises.
- Regularly attend all the labs
- Put the chairs back to its position before you leave.
- Treat all the devices with care and consideration.
- Behave in a responsible manner at all times and maintain silence.
- Before leaving the lab shut down the system and rearrange the chairs
- Keep your premises clean

DON'T

- Use Mobile phones and pen drives
- Move around in the lab during the lab session.
- Tamper System Files or Try to access the Server.
- Write Records in the Lab
- Change the system assigned to you without the notice of the Lab Staff.
- Write on the table or mouse pads.
- Do not install or download any software or modify or delete any system files on any lab computers.

Assessment Pattern (both CIE and SEE)

Credit Course								
Assessment Method	**Component**	**Type of Assessment**	**Assessment Type used**	**Max. Marks**	**Evaluation Details**	**Reduced Marks**	**Min. Marks**	**Total**
CIE	Practical	Continuous Evaluation	Observation, Record, Execution , Viva	50	Observation -10 Record – 10 Execution of Programs - 20 Viva – 10 Total – 50 marks , Scored marks are scaled down to 30 marks	30	15	30
		Test-1	Practical	100	Average of two Internal Assessment Tests each of 100 Marks, Scored Marks are scaled down to 20 marks	20	10	20
		Test-2	Practical	100				
	Total CIE Practical						**25**	**50**
SEE				**100**	SEE Exam is Theory Exam, conducted for 100 Marks, scored marks are scaled down to 50 marks	**50**	**20**	**50**
CIE+SEE							**40**	**100**

COURSE PLAN

List of program

Problem Solving Using C: MMCPB16

PROBLEM SOLVING USING C	
Semester :I	**CIE Marks:40**
Subject Code: MMCPB16	**SEE Exam:60**
Credits (L: T:P:PJ):0:0:2:0	**Exam Hours:03**

1. Write a C program to Implement the following searching techniques
a. Linear Search b. Binary Search.
SOLUTIONS

```
#include <stdio.h>

// Linear Search
int linearSearch(int arr[], int n, int key) {
    int i;
    for (i = 0; i < n; ++i) {
        if (arr[i] == key)
            return i; // Return the index if key is found
    }
    return -1; // Return -1 if key is not found
}

// Binary Search
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid; // Return the index if key is found
        else if (arr[mid] < key)
            low = mid + 1; // Search in the right half
        else
            high = mid - 1; // Search in the left half
    }
    return -1; // Return -1 if key is not found
}

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

    // Linear search
    int linearIndex = linearSearch(arr, n, key);
    if (linearIndex != -1)
        printf("Linear Search: Element %d found at index %d.\n", key, linearIndex);
    else
        printf("Linear Search: Element %d not found in the array.\n", key);
```

```
    // Binary search (Array must be sorted)
    int binaryIndex = binarySearch(arr, 0, n - 1, key);
    if (binaryIndex != -1)
        printf("Binary Search: Element %d found at index %d.\n", key, binaryIndex);
    else
        printf("Binary Search: Element %d not found in the array.\n", key);

    return 0;
}
```

OUTPUT:

```
Linear Search: Element 23 found at index 5.
Binary Search: Element 23 found at index 5.
```

2. Write a C program to implement the following sorting algorithms using user defined functions: a. Bubble sort (Ascending order) b. Selection sort (Descending order).

SOLUTIONS

```
#include <stdio.h>
// Function to perform bubble sort
void bubbleSort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; ++i) {
        for (j = 0; j < n - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to perform selection sort
void selectionSort(int arr[], int n) {
    int i, j, minIndex;
    for (i = 0; i < n - 1; ++i) {
        minIndex = i;
        for (j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap arr[i] and arr[minIndex]
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
```

```
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    // Bubble Sort
    bubbleSort(arr, n);
    printf("Array after bubble sort (ascending order):\n");
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");

    // Selection Sort
    selectionSort(arr, n);
    printf("Array after selection sort (descending order):\n");
    for (int i = 0; i < n; ++i)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

OUTPUT:

```
Array before sorting:
64 34 25 12 22 11 90
Array after bubble sort (ascending order):
11 12 22 25 34 64 90
Array after selection sort (descending order):
90 64 34 25 22 12 11
```

3. Write a C Program implement STACK with the following operations a. Push an Element onto Stack b. Pop an Element from Stack

SOLUTIONS:

```
#include <stdio.h>
#define MAX_SIZE 100
// Structure to represent a stack
typedef struct {
    int top;
    int array[MAX_SIZE];
} Stack;
```

```
// Function to create an empty stack
Stack* createStack() {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->top = -1;
    return stack;
}

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

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

// Function to push an element onto the stack
void push(Stack* stack, int item) {
    if (isFull(stack)) {
        printf("Stack overflow. Cannot push.\n");
        return;
    }
    stack->array[++stack->top] = item;
}

// Function to pop an element from the stack
int pop(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow. Cannot pop.\n");
        return -1;
    }
    return stack->array[stack->top--];
}

// Function to display the elements of the stack
void display(Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack elements: ");
    int i;
    for (i = stack->top; i >= 0; --i)
        printf("%d ", stack->array[i]);
    printf("\n");
```

```
}

// Main function
   int main() {
   Stack* stack = createStack();
   int choice, item;

   do {
      printf("\nStack Operations:\n");
      printf("1. Push\n");
      printf("2. Pop\n");
      printf("3. Display\n");
      printf("4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &choice);

      switch (choice) {
         case 1:
            printf("Enter element to push: ");
            scanf("%d", &item);
            push(stack, item);
            break;
         case 2:
            item = pop(stack);
            if (item != -1)
               printf("Popped element: %d\n", item);
            break;
         case 3:
            display(stack);
            break;
         case 4:
            printf("Exiting...\n");
            break;
         default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   } while (choice != 4);

   return 0;
}
OUTPUT:
Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
```

```
Enter element to push: 5

Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter element to push: 8

Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 8 5

Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
Popped element: 8

Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements: 5

Stack Operations:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting...
```

4. Implement a Program in C for converting an Infix Expression to Postfix Expression.

SOLUTIONS:

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

```
#include<string.h>
#include<ctype.h>
#define MAX_SIZE 100

// Structure to represent a stack
typedef struct {
    int top;
    char array[MAX_SIZE];
} Stack;

// Function to create a stack
Stack* createStack() {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->top = -1;
    return stack;
}

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

// Function to return the top element of the stack
char peek(Stack* stack) {
    return stack->array[stack->top];
}

// Function to push an element onto the stack
void push(Stack* stack, char item) {
    stack->array[++stack->top] = item;
}

// Function to pop an element from the stack
char pop(Stack* stack) {
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '\0'; // Return null character if the stack is empty
}

// Function prototype for precedence function
int precedence(char op);

// Function to convert infix expression to postfix expression
void infixToPostfix(char* infix, char* postfix) {
    Stack* stack = createStack();
    int i, k;
    for (i = 0, k = -1; infix[i]; ++i) {
```

```
        char ch = infix[i];
        if (isalnum(ch))
            postfix[++k] = ch;
        else if (ch == '(')
            push(stack, ch);
        else if (ch == ')') {
            while (!isEmpty(stack) && peek(stack) != '(')
                postfix[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return; // Invalid expression
            else
                pop(stack); // Discard '('
        } else { // Operator
            while (!isEmpty(stack) && peek(stack) != '(' && precedence(ch) <=
precedence(peek(stack)))
                postfix[++k] = pop(stack);
            push(stack, ch);
        }
    }
    while (!isEmpty(stack))
        postfix[++k] = pop(stack);
    postfix[++k] = '\0';
}

// Function to get the precedence of an operator
int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}

// Main function
int main() {
    char infix[MAX_SIZE];
    printf("Enter an infix expression: ");
    fgets(infix, MAX_SIZE, stdin);

    // Remove newline character from input
    infix[strcspn(infix, "\n")] = '\0';

    char postfix[MAX_SIZE];
    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);
    return 0;
}
```

OUTPUT:
1)Enter an infix expression: A+B*C+D
Postfix expression:ABC*+D+

2)Enter an infix expression: ((A+B)-C*(D/E))+F
Postfix expression:AB+CDE/*-F+

5. Implement a Program in C for evaluating a Postfix Expression.

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

#define MAX_SIZE 100

// Structure to represent a stack
typedef struct {
    int top;
    int array[MAX_SIZE];
} Stack;

// Function to create a stack
Stack* createStack() {
    Stack* stack = (Stack*)malloc(sizeof(Stack));
    stack->top = -1;
    return stack;
}

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

// Function to push an element onto the stack
void push(Stack* stack, int item) {
    stack->array[++stack->top] = item;
}

// Function to pop an element from the stack
int pop(Stack* stack) {
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return -1; // Return -1 if the stack is empty
}

// Function to evaluate postfix expression
```

```
int evaluatePostfix(char* expression) {
    Stack* stack = createStack();
    int i, operand1, operand2;
    for (i = 0; expression[i]; ++i) {
        char ch = expression[i];
        if (isdigit(ch))
            push(stack, ch - '0'); // Convert char to int
        else { // Operator
            operand2 = pop(stack);
            operand1 = pop(stack);
            switch (ch) {
                case '+': push(stack, operand1 + operand2); break;
                case '-': push(stack, operand1 - operand2); break;
                case '*': push(stack, operand1 * operand2); break;
                case '/': push(stack, operand1 / operand2); break;
            }
        }
    }
    return pop(stack);
}

// Main function
    int main() {
    char expression[MAX_SIZE];
    printf("Enter a valid postfix expression: ");
    fgets(expression, MAX_SIZE, stdin);



    // Remove newline character from input
    expression[strcspn(expression, "\n")] = '\0';

    int result = evaluatePostfix(expression);
    printf("Result: %d\n", result);
    return 0;
}
```

OUTPUT:

1)Enter a valid postfix expression:245+*
  Result:18

2) Enter a valid postfix expression:12345*+*+
  Result:47

6. Write a C program to simulate the working of a singly linked list providing the following operations: a. Display & Insert b. Delete from the beginning/end c. Delete a given element

SOLUTIONS:

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

// Structure to represent a node in the linked list
typedef struct Node {
    int data;
    struct Node* next;
} Node;

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

// Function to display the elements of the linked list
void display(Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked List elements: ");
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

// Function to insert an element at the beginning of the linked list
Node* insertAtBeginning(Node* head, int data) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head = newNode;
    }
    return head;
}
```

```
// Function to insert an element at the end of the linked list
Node* insertAtEnd(Node* head, int data) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    return head;
}

// Function to delete an element from the beginning of the linked list
Node* deleteFromBeginning(Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
    } else {
        Node* temp = head;
        head = head->next;
        free(temp);
    }
    return head;
}

// Function to delete an element from the end of the linked list
Node* deleteFromEnd(Node* head) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
    } else {
        Node* temp = head;
        Node* prev = NULL;
        while (temp->next != NULL) {
            prev = temp;
            temp = temp->next;
        }
        if (prev == NULL)
            head = NULL;
        else
            prev->next = NULL;
        free(temp);
    }
    return head;
}

// Function to delete a given element from the linked list
```

```
Node* deleteElement(Node* head, int data) {
    if (head == NULL) {
        printf("List is empty. Cannot delete.\n");
    } else {
        Node* temp = head;
        Node* prev = NULL;
        while (temp != NULL && temp->data != data) {
            prev = temp;
            temp = temp->next;
        }
        if (temp == NULL) {
            printf("Element not found in the list.\n");
        } else {
            if (prev == NULL)
                head = head->next;
            else
                prev->next = temp->next;
            free(temp);
        }
    }
    return head;
}

// Main function
int main() {
    Node* head = NULL;
    int choice, data;

    do {
        printf("\nLinked List Operations:\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Delete from beginning\n");
        printf("4. Delete from end\n");
        printf("5. Delete a given element\n");
        printf("6. Display\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter element to insert at beginning: ");
                scanf("%d", &data);
                head = insertAtBeginning(head, data);
                break;
            case 2:
```

```
            printf("Enter element to insert at end: ");
            scanf("%d", &data);
            head = insertAtEnd(head, data);
            break;
        case 3:
            head = deleteFromBeginning(head);
            break;
        case 4:
            head = deleteFromEnd(head);
            break;
        case 5:
            printf("Enter element to delete: ");
            scanf("%d", &data);
            head = deleteElement(head, data);
            break;
        case 6:
            display(head);
            break;
        case 7:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice! Please enter a valid option.\n");
      }
   } while (choice != 7);

   return 0;
}
```

OUTPUT

```
Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 1
Enter element to insert at beginning: 5

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
```

```
7. Exit
Enter your choice: 2
Enter element to insert at end: 8

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 6
Linked List elements: 5 8

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 3

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 6
Linked List elements: 8

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 4

Linked List Operations:
```

```
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 6
List is empty.

Linked List Operations:
1. Insert at beginning
2. Insert at end
3. Delete from beginning
4. Delete from end
5. Delete a given element
6. Display
7. Exit
Enter your choice: 7
Exiting...
```

7. Check whether a given graph is connected or not using DFS method using C program

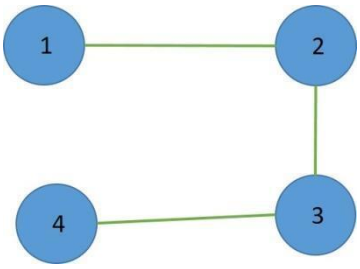
Solution:

```
#include <iostream>
#include <vector>

using namespace std;

class Graph {
   int V; // Number of vertices
   vector<vector<int>> adj; // Adjacency list

   void DFSUtil(int v, vector<bool>& visited) {
      visited[v] = true;

      // Recur for all the vertices adjacent to this vertex
      for (int i : adj[v]) {
         if (!visited[i]) {
            DFSUtil(i, visited);
         }
      }
   }

public:
```

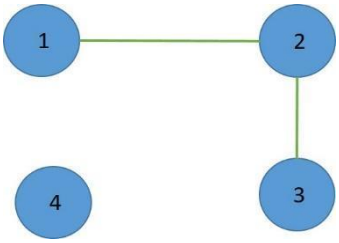
```
    Graph(int V) {
        this->V = V;
        adj.resize(V);
    }
    void addEdge(int v, int w) {
        adj[v].push_back(w); // Add w to v's list.
        adj[w].push_back(v); // Since the graph is undirected, add v to w's list.
    }

    bool isConnected() {
        vector<bool> visited(V, false);
        // Start DFS from the first vertex with an edge
        int i;
        for (i = 0; i < V; i++) {
            if (!adj[i].empty()) {
                break;
            }
        }

        // If there are no edges in the graph, it's considered connected
        if (i == V) {
            return true;
        }

        DFSUtil(i, visited);

        // Check if all non-isolated vertices are visited
        for (int i = 0; i < V; i++) {
            if (!visited[i] && !adj[i].empty()) {
                return false;
            }
        }

        return true;
    }
};

int main() {
    Graph g(5); // 5 vertices numbered from 0 to 4
    g.addEdge(0, 1);
    g.addEdge(1, 2);
    g.addEdge(2,3);
    g.addEdge(3, 4);

    if (g.isConnected()) {
        cout << "The graph is connected." << endl;
    } else {
```

```
        cout << "The graph is not connected." << endl;
    }

    return 0;
}
```

Output



The Graph is Connected



The Graph is not connected

8. **From a given vertex in a weighted connected graph, find shortest paths to other vertices using Dijkstra's algorithm.**

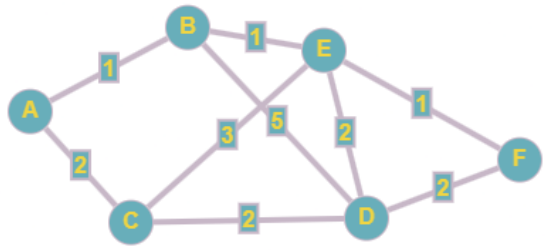
Solution:

```
#include<iostream>
#include<limits.h>
using namespace std;
int miniDist(int distance[], bool fin[]) // finding minimum distance
{
    int minimum=INT_MAX,ind;

    for(int k=0;k<6;k++)
    {
        if(fin[k]==false && distance[k]<=minimum)
        {
            minimum=distance[k];
```

```
            ind=k;
        }
    }
    return ind;
}

void DijkstraAlgo(int graph[6][6],int src) // adjacency matrix
{
    int distance[6]; // // array to calculate the minimum distance for each node
    bool fin[6];// boolean array to mark visited and unvisited for each node

    for(int k = 0; k<6; k++)
    {
        distance[k] = INT_MAX;
        fin[k] = false;
    }
        distance[src] = 0;   // Source vertex distance is set 0

    for(int i = 0; i<6; i++)
    {
        int m=miniDist(distance,fin);
        fin[m]=true;
        for(int k = 0; k<6; k++)
        {
            // updating the distance of neighbouring vertex
            if(!fin[k] && graph[m][k] && distance[m]!=INT_MAX &&
distance[m]+graph[m][k]<distance[k])
                distance[k]=distance[m]+graph[m][k];
        }
    }
    cout<<"Vertex\t\tDistance from source vertex"<<endl;
    for(int k = 0; k<6; k++)
    {
        char str=65+k;
        cout<<str<<"\t\t\t"<<distance[k]<<endl;
    }
}
int main()
{
    int graph[6][6]={
        {0, 1, 2, 0, 0, 0},
        {1, 0, 0, 5, 1, 0},
        {2, 0, 0, 2, 3, 0},
        {0, 5, 2, 0, 2, 2},
        {0, 1, 3, 2, 0, 1},
        {0, 0, 0, 2, 1, 0}};
    DijkstraAlgo(graph,0);
```

```
    return 0;
}
```

Output



Vertex	Distance from source vertex
A	0
B	1
C	2
D	4
E	2
F	3

9. **Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.**

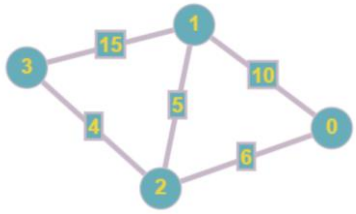
Solution:

```
#include <iostream>
#include <vector>
//#include <algorithm>
using namespace std;
// Function to find the parent of a node
int find(int parent[], int i) {
    if (parent[i] != i)
        parent[i] = find(parent, parent[i]); // Path compression
    return parent[i];
}
// Function to unite two subsets
void unionSets(int parent[], int rank[], int x, int y) {
    int rootX = find(parent, x);
    int rootY = find(parent, y);
    if (rootX != rootY) {
        // Union by rank
        if (rank[rootX] > rank[rootY])
            parent[rootY] = rootX;
```

```
        else if (rank[rootX] < rank[rootY])
            parent[rootX] = rootY;
        else {
            parent[rootY] = rootX;
            rank[rootX]++;
        }
    }
}

// Kruskal's algorithm to find the Minimum Spanning Tree
void kruskal(int V, vector<vector<int>>& edges) {
    vector<pair<int, pair<int, int>>> result; // {weight, {u, v}}
    int parent[V], rank[V];
    // Initialize parent and rank
    for (int i = 0; i < V; ++i) {
        parent[i] = i;
        rank[i] = 0;
    }
    // Sort edges by weight
    sort(edges.begin(), edges.end());
    int minCost = 0; // Total cost of MST
    for (auto& edge : edges) {
        int weight = edge[0];
        int u = edge[1];
        int v = edge[2];
        // Check if including this edge would form a cycle
        if (find(parent, u) != find(parent, v)) {
            unionSets(parent, rank, u, v);
            result.push_back({weight, {u, v}});
            minCost += weight;
        }
    }
    // Print the edges in the MST
    cout << "Edges in the MST:\n";
    for (auto& edge : result)
        cout << edge.second.first << " -- " << edge.second.second << " == " << edge.first << endl;
    cout << "Minimum cost of MST: " << minCost << endl;
}
int main() {
    int V = 4; // Number of vertices
    // Define edges in the form {weight, u, v}
    vector<vector<int>> edges = {
        {10, 0, 1},
        {6, 0, 2},
        {15, 1, 3},
        {4, 2, 3},
        {5, 1, 2}
```

```
    };
    kruskal(V, edges);
    return 0;
}
```

Output




```
Edges in the MST:
2 -- 3 == 4
1 -- 2 == 5
0 -- 2 == 6
Minimum cost of MST: 15
```

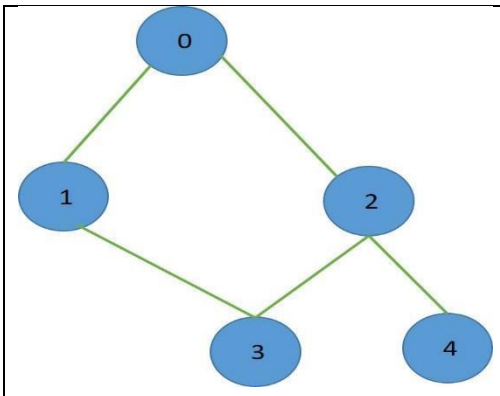
10. **Print all the nodes reachable from a given starting node in a digraph using BFS method**

PROGRAM

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_map>
using namespace std;
// Function to perform BFS and print all reachable nodes
void BFS(int start, unordered_map<int, vector<int>>& adj) {
    // To keep track of visited nodes
    unordered_map<int, bool> visited;
    // Queue to perform BFS
    queue<int> q;
    // Start BFS from the given start node
    visited[start] = true;
    q.push(start);
    cout << "Nodes reachable from node " << start << ": ";
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        // Traverse all adjacent nodes
        for (int neighbor : adj[node]) {
```

```
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
    cout << endl;
}
int main() {
    // Number of nodes and edges
    int n, e;
    cout << "Enter number of nodes: ";
    cin >> n;
    cout << "Enter number of edges: ";
    cin >> e;
    // Adjacency list to store the graph
    unordered_map<int, vector<int>> adj;
    cout << "Enter edges (format: u v, meaning u -> v):" << endl;
    for (int i = 0; i < e; ++i) {
        int u, v;
        cin >> u >> v;
        // Ensure u and v are within the valid node range
        if (u >= 0 && u < n && v >= 0 && v < n) {
            adj[u].push_back(v);
        } else {
            cout << "Invalid edge (" << u << ", " << v << "). Skipping." << endl;
        }
    }
    // Starting node for BFS
    int start;
    cout << "Enter the starting node: ";
    cin >> start;

    // Check if the starting node is within the valid range
    if (start >= 0 && start < n) {
        // Perform BFS from the start node
        BFS(start, adj);
    } else {
        cout << "Invalid starting node." << endl;
    }
    return 0;
}
```

Output




```
Enter number of nodes: 5
Enter number of edges: 5
Enter edges (format: u v, meaning u -> v):
0
1
0
2
1
3
2
3
2
4
Enter the starting node: 0
Nodes reachable from node 0: 0 1 2 3 4
```

DATA STRUCTURES AND ALGORITHM VIVA QUESTIONS

1. What is Data Structure?

A data structure is a model for organizing and storing data. Stacks, Queue, Linked Lists, and Trees are the examples of different data structures. Data structures are classified as either linear or nonlinear:

- A data structure is said to be linear if only one element can be accessed from an element directly. Eg: Stacks, Lists, Queues.
- A data structure is non-linear if more than one elements can be accessed from an element directly. Eg: Trees, Graphs, Heap.

2. Define ADT?

An abstract data type is a set of objects together with a set of operations. Abstract data types are mathematical abstraction. Objects such as lists, sets, graphs, trees can be viewed as abstract data types.

3. What do you mean by LIFO and FIFO?

LIFO stands for 'Last In First Out', it says how the data to be stored and retrieved. It means the data or element which was stored last should come out first. Stack follows LIFO.

FIFO stands for 'First In First Out', here the data(or element) which was stored first should be the one to come out first. It is implemented in Queue.

4. What is Stack?

A stack is a list of elements in which insertion and deletions can take place only at one end, this end is called as stack 'top'. Only top element can be accessed. A stack data structure has LIFO (Last In First Out) property. A stack is an example of linear data structure.

5. What operations can be done on Stack?

The following operations can be performed on stack:

- **Push** operation inserts new element into stack.
- **Pop** operation deletes the top element from the stack.
- **Peek** returns or give the top element without deleting it.
- **Isempty()** operation returns whether stack is empty or not.

6. List some applications of stack?

Some well-known applications of stack are as follow:

1. Infix to Postfix conversion.
2. Evaluatiion of postfix expression.
3. Check for balanced parantheses in an expression.
4. Stack is very important data structure being used to implement function calls efficiently.
5. Parsing
6. Simulation of recursion function.
7. String reverse using stack.

7. What is a Queue?

A Queue is a particular data structure in which the elements in the collection are kept in order. Unlike Stack, queue is opened at both end. Queue follows 'FIFO' method where element is inserted from rear end and deleted or removed from front end.

8. What operations can be done on Queue?

The following operations can be performed on queue:

- **Enqueue** operation inserts new element in rear of the list.
- **Dequeue** operation deletes the front element from the list.
- **Isempty()** operation checks whether queue is empty or not.

9. Where do we use Queue?

Some well-known applications of queue are as follow:

1. For findind level order traversal efficiently.
2. For implementing BFS efficiently.
3. For implementing Kruskal algorithm efficiently.

10. What is a bubble sort?

Bubble sort is a simple sorting algorithm, it works by repeatedly stepping through the list to be sorted comparing each pair of adjacent items and swapping if they are in the worng order.

First pass bubbles out the largest element and places it in the last position and second pass places the second largest element in the second last position and so on. Thus in the last pass smallest element is placed in the first position.

The running time is the total number of comparisons that is n(n-1)/2 which implies $O(n^2)$ time complexity.

11. What is Insertion Sort?

Insertion sort is a simple comparison sorting algorithm, every iteration of insertion sort removes an element from the input data and insert it into the correct position in the already sorted list until no input element remains.

The running time is the total number of comparisons that is n(n-1)/2 which implies $O(n^2)$ time complexity.

12. What is Selection Sort?

Insertion sort is a simple comparison sorting algorithm, the algorithm works as follows:

1. Find the minimum value in the list.
2. Swap it with the minimum value in the first position.
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time).

The running time is the total number of comparisons that is n(n-1)/2 which implies $O(n^2)$ time complexity.

13. What is Merge Sort?

Merge sort is an O(nlogn) comparison-based divide and conquer sorting algorithm, it works as follows:

1. If the list is of length 0 or 1, then it is already sorted.
2. Divide the unsorted list into two sub lists of about half the size.
3. Sort each sublist recursively by re-applying merge sort algorithm.
4. Merge the two sublists back into one sorted list.

14. What is Heap Sort?

Heap Sort is a comparison-based sorting algorithm which is much more efficient version of selection sort, it works by determining the largest (or smallest) element of the list, placing that at the end (or beginning)of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap. Once the data list has been made into a heap, the root node is gurantedd to be the largest (or smallest) element.

15. What is Quick Sort?

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists. The steps are:

1. Pick an element, called a pivot, from the list.
2. Reorder the list so the elements which are less than the pivot come before the pivot and the elements greater than pivot come after it. After this partitioning the pivot is in its final positon.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The running time complexity for worst case is $O(n^2)$ and for best and average case it is same i.e., O(nlogn).

16. What is a Graph?

A graph is a pair of sets (V,E), where V is the sets of vertices and E is the set of edges connecting the pair of vertices.

17. What is a Directed and Undirected Graph?

A graph is directed if each edge of graph has a direction between vertices.

A graph is undirected if there are no direction between vertices.

18**. What is a Connected Graph?**

A undirected graph is connected if there is a path from every vertex to every other vertex. A directed graph with this property is called strongly connected.

19.What is Greedy Technique?

The greedy technique is an algorithmic approach that makes a series of choices, each of which looks best at the moment, with the hope of finding a global optimum. In other words, a greedy algorithm makes the locally optimal choice at each step, aiming to find the overall best solution.

20.What is Dijkstra's algorithm?

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph by maintaining a set of vertices to explore and iteratively updating the shortest known distances.

21.What is Kruskal's algorithm?

Kruskal's algorithm finds the Minimum Cost Spanning Tree (MST) by sorting all edges and adding them one by one to the MST, ensuring no cycles are formed, until all vertices are connected.

22.**What is the BFS method?**

Breadth-First Search (BFS) is an algorithm for traversing or searching tree or graph. It explores all neighbors of a node before moving to the next level using queue, effectively finding all reachable nodes from a starting point.

23.How does Depth-First Search (DFS) check graph connectivity?

DFS traverses the graph by exploring as far as possible along each branch before backtracking. If all vertices are visited during the traversal from a starting vertex, the graph is connected.