GOVERNMENT POLYTECHNIC SHEOHAR

LABORATORY MANUAL

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

III SEMESTER

DATA STRUCTURE AND ALGORITHM

SUBJECT CODE: P2418301

LIST OF EXPERIMENTS

1	Write Program to find size of different data types.	CO-1
2	**a).** Write a program to insert an element in a given array. **b).** Write a program to delete an element from a given array **c).** Write a program to modify a character in a string **d).** Write a program to insert a node at beginning, mid, and end of a given singly linked list **e).** Write a program to insert a node at beginning, mid, and end of a given circular linked list **f).** Write a program using stack for a given expression evaluation. **g).** Write a program to perform enqueue and dequeue operations on Queue	CO-2
3	**a).** Write programs to perform in order pre order, and post order traversal on a tree.). **b).** Write functions to perform the following operations on the heap: i. Insert an element into the heap. ii. Delete the root element (highest priority) from the heap. iii. Retrieve the root element without removing it. iv. Check if the heap is empty **c).** Write a program to perform following operation on a given Priority Queue: i. Enqueue of an element ii. Dequeue of an element i. Find the element with highest priority iii. Determine the size of Priority Queue iv. Empty check of Priority Queue **d).** Write programs to perform following operation on graph i. To detect a cycle in a given graph using DFS ii. To find an articulation point in a graph using DFS iii. To find the shortest path between two given nodes using BFS **e).** Apply Bellman-Ford algorithm to find shortest path for a given negative edge graph.	CO-3

	f). Apply Floyd-Warshall algorithm to find shortest path for a given weighted directed graph.	
4	**a).** Develop a Program to: i. Apply insertion sort, quicksort, and merge sort on given dataset. ii. Apply binary search to find an element in given array. **b).** Write a program to create a hash table using array data structure. **C).** Write a program to perform the following operations on the hash table: i. Insert a key-value pair into the hash table. ii. Retrieve the value associated with a given key from the hash table. iii. Delete a key-value pair from the hash table. i. iv. Check if a key exists in the hash table.	CO-4
5	Develop Program to: i. Find Longest common sequence in given strings. Expected Output: [('item3', '24.5'), ('item2', '15.10'), ('item1', '12.20')] ii. Find shortest path using Bellman-Ford algorithm for a given graph. iii. iii. Find minimum and maximum value from n elements using divide and conquer method.	CO-5

PRACTICAL OUTCOMES

CO	Statement	Revised Bloom Taxonomy
1	Analyze the efficiency of algorithm	Analyze
2	Implement operations on linear data structures	Apply
3	Implement operations on non-linear data structures	Apply
4	Apply different searching, sorting and hashing techniques to solve real world problems.	Apply
5	Design efficient algorithms to solve the real-world problems.	Apply

PROGRAM OUTCOMES (POS)

PO1 - Basic and Discipline specific knowledge: Apply knowledge of basic mathematics, science and engineering fundamentals and engineering specialization to solve the engineering problems.

PO2 - Problem analysis: Identify and analyse well-defined engineering problems using codified standard methods.

PO3 - Design/ development of solutions: Design solutions for well-defined technical problems and assist with the design of systems components or processes to meet specified needs.

PO4 - Engineering Tools, Experimentation and Testing: Apply modern engineering tools and appropriate technique to conduct standard tests and measurements.

PO5 - Engineering practices for society, sustainability and environment: Apply appropriate technology in context of society, sustainability, environment and ethical practices.

PO6 - Project Management: Use engineering management principles individually, as a team member or a leader to manage projects and effectively communicate about well- defined engineering activities.

PO7 - Life-long learning: Ability to analyze individual needs and engage in updating in the context of technological changes.

PROGRAM SPECIFIC OUTCOMES

PSO1: Apply the fundamentals of computer science, including algorithms and programming, to analyze complex problems and develop robust solutions.

PSO2: Design and implement computer science solutions with emphasis on energy efficiency, sustainability, and eco-friendly technologies.

MAPPING

S. No.	CO-Statement	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PSO1	PSO2
CO1	Analyze the efficiency of algorithm	1						1	2	2
CO2	Implement operations on linear data structures	2	2	1	1				1	
CO3	Implement operations on non-linear data structures	2	2	1	1				2	
CO4	Apply different searching, sorting and hashing techniques to solve real world problems.	2	3	1	1				2	2
CO5	Design efficient algorithms to solve the real-world problems.	2	3	1	1			1	2	2

EXPERIMENT NO: 1

WRITE PROGRAM TO FIND SIZE OF DIFFERENT DATA TYPES.

AIM: - To write a C program to find and display the size of different data types.

APPARATUS / TOOLS REQUIRED: -

- Computer system with C compiler (e.g., Turbo C / GCC / Code: Blocks)
- Vs Code
- Keyboard and Monitor
- Operating System: Windows / Linux

INSTALLATION STEPS: -

Installing MinGW Compiler

1. Open your web browser and go to: https://www.mingw-w64.org/.
2. Download the MinGW-w64 installer for Windows.
3. Run the downloaded setup file.
4. During installation:
    - Select Architecture as x86_64 (for 64-bit system).
    - Select Threads as posix.
    - Select Exception as seh.
    - Choose an installation directory (e.g., C:\MinGW).
5. After installation, open the Environment Variables:
    - Go to This PC → Properties → Advanced System Settings → Environment Variables.
    - Under System Variables, find and select Path → Click Edit.
    - Click New, and add:
    - C:\MinGW\bin
    - Click OK on all windows to save the changes.
6. Verify installation:
    - Open Command Prompt and type:

- gcc --version
- If you see a GCC version message, installation is successful.

Installing and Setting Up Visual Studio Code (VS Code)

1. Download Visual Studio Code from https://code.visualstudio.com/.
2. Run the installer and follow the on-screen steps to install.
3. Open VS Code after installation.
4. Go to the Extensions panel (press Ctrl + Shift + X).
5. In the search bar, type C/C++ and install the extension by Microsoft.
6. (Optional) Install Code Runner extension to run programs easily.
7. Create a new folder for your C programs (e.g., CPrograms).
8. Open that folder in VS Code (File → Open Folder).
9. Create a new file and save it as size.c.
10. Write your C program in the editor.
11. Open the terminal in VS Code (Terminal → New Terminal).
12. Compile the program using: gcc size.c -o size
13. Run the program using: ./size
14. Observe the output in the terminal.

THEORY: -

In C, each data type occupies a specific amount of memory depending on the compiler and system architecture.

The size of() operator in C is used to determine the memory size (in bytes) occupied by a data type or variable.

This helps programmers understand how much memory is allocated for storing different data types.

Common Data Types and Their Typical Sizes:

Data Type	Description	Typical Size (in Bytes)
Int	Integer type	4
Float	Floating-point number	4
Double	Double-precision float	8
Char	Character type	1
Long Int	Long integer	8
Short Int	Short integer type	2

(Note: Sizes may vary depending on the compiler and system architecture.)

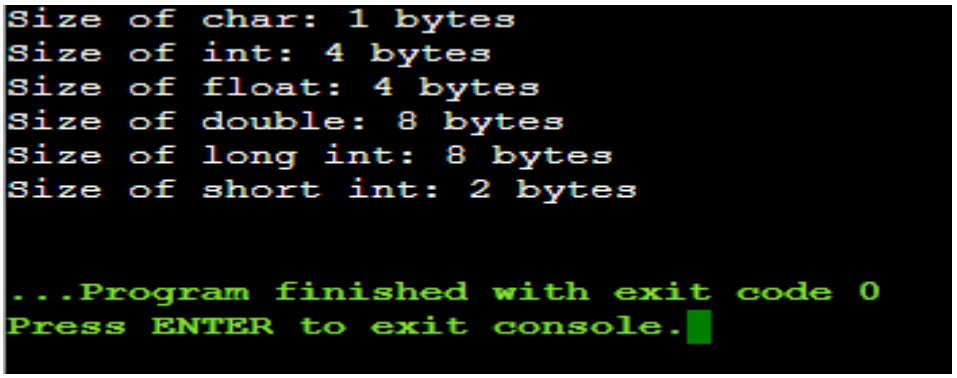
PROCEDURE: -

1. Open Visual Studio Code.
2. Create a new C file named size.c.
3. Write the C program using sizeof() operator.
4. Save the file.
5. Open the terminal and compile the program using GCC.
6. Execute the program to display data type sizes.
7. Record the output.

PROGRAM (CODE): -

```
#include <stdio.h>
  int main() {
     printf("Size of char: %lu bytes\n", sizeof(char));
     printf("Size of int: %lu bytes\n", sizeof(int));
     printf("Size of float: %lu bytes\n", sizeof(float));
     printf("Size of double: %lu bytes\n", sizeof(double));
     printf("Size of long int: %lu bytes\n", sizeof(long int));
     printf("Size of short int: %lu bytes\n", sizeof(short int));
     return 0;
  }
```

OUTPUT: -



RESULT: -

The program successfully displayed the size (in bytes) of different data types using the size of() operator in C.

EXPERIMENT NO: 2

TO STUDY AND IMPLEMENT VARIOUS PROGRAMS RELATED TO ARRAYS, STRINGS, LINKED LISTS, STACKS, AND QUEUES.

AIM: - To study and implement various programs related to arrays, strings, linked lists, stacks, and queues.

APPARATUS / TOOLS REQUIRED: -

- Computer system with C compiler (e.g., Turbo C / GCC / Code: Blocks)
- Vs Code
- Keyboard and Monitor
- Operating System: Windows / Linux

THEORY: -

1. **Array:**
    - An array is a collection of elements of the same data type stored in contiguous memory locations.
    - Operations include insertion, deletion, traversal, and searching.
    - To insert, elements are shifted right; to delete, elements are shifted left.
2. **String:**
    - A string is a sequence of characters terminated by a null character (\0).
    - Operations include insertion, deletion, and modification of characters.
3. **Linked List:**
    - A linked list is a linear data structure where each node contains data and a pointer to the next node.
    - Types:
        - **Singly Linked List** – each node points to the next node.
        - **Circular Linked List** – last node points back to the head.
    - Operations include insertion and deletion at the beginning, middle, and end.
4. **Stack:**
    - Stack is a LIFO (Last In, First Out) data structure.

- The main operations are **push** (insert) and **pop** (delete).
- Used in expression evaluation and function call management.

5. **Queue:**
   - Queue is a FIFO (First In, First Out) data structure.
   - The main operations are **enqueue** (insert at rear) and **dequeue** (delete from front).
   - Commonly used in scheduling and buffering.

(a) Program to Insert an Element in a Given Array

PROCEDURE:

1. Declare an array and read its size.
2. Input the array elements.
3. Take the position and new element from the user.
4. Shift elements one position to the right.
5. Insert the new element.
6. Display the updated array.

PROGRAM (CODE): -

```
#include <stdio.h>
int main() {
    int arr[100], n, i, pos, element;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter position to insert element: ");
    scanf("%d", &pos);
    printf("Enter element to insert: ");
    scanf("%d", &element);
```

```
    for (i = n; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[pos - 1] = element;
    n++;
    printf("Array after insertion:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT: -

```
Enter number of elements: 5
Enter elements: 10
12
15
16
19
Enter position to insert element: 3
Enter element to insert: 50
Array after insertion:
10 12 50 15 16 19

...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

The element was successfully inserted at the specified position in the array.

(b) program to delete an element from a given array

PROCEDURE: -

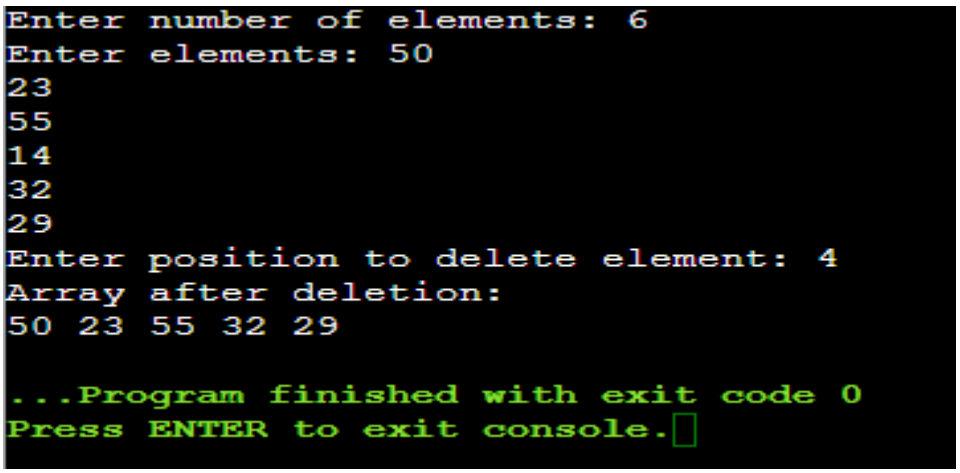
1. Start the program and declare an array.
2. Input the number of elements and their values.
3. Ask for the position of the element to delete.
4. Shift all subsequent elements one place to the left.
5. Decrease the total element count.
6. Display the updated array.

7. Stop the program.

PROGRAM (CODE): -

```
#include <stdio.h>
int main() {
    int arr[100], n, i, pos;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter position to delete element: ");
    scanf("%d", &pos);
    for (i = pos - 1; i < n - 1; i++)
        arr[i] = arr[i + 1];
    n--;
    printf("Array after deletion:\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT: -



RESULT: -

The program successfully deleted an element from the array.

(c) Program to Modify a Character in a String

PROCEDURE: -

1. Read a string from the user.
2. Input the position of the character to modify.
3. Input the new character.
4. Replace the old character at that position.
5. Display the modified string.

PROGRAM (CODE): -

```
#include <stdio.h>
#include <string.h>
int main() {
    char str[100];
    int pos;
    char ch;
    // Use fgets() instead of gets() to avoid buffer overflow
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);
```

```
    // Remove the trailing newline character if it exists
    str[strcspn(str, "\n")] = '\0';
    printf("Enter position to modify: ");
    scanf("%d", &pos);
    // Check if position is valid
    if (pos < 1 || pos > strlen(str)) {
       printf("Invalid  position.\n");
       return 1;
    }
    printf("Enter new character: ");
    scanf(" %c", &ch); // The space before %c is to consume any leftover whitespace
    // Modify the string at the specified position
    str[pos - 1] = ch;
    printf("Modified string: %s\n", str);
    return 0;
}
```

OUTPUT: -

```
Enter a string: hello
Enter position to modify: 1
Enter new character: java
Modified string: jello


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

The program successfully modified a character in the given string.

(d) Program to Insert a Node at Beginning, Middle, and End of a Singly Linked List

PROCEDURE: -

1. Define a structure Node with data and next pointer.
2. Create an initial linked list.
3. Create and insert a new node at:
    - Beginning
    - Middle
    - End
4. Update the links properly.
5. Display the linked list after each insertion.

PROGRAM (CODE): -

```
#include  <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node *next;
};

void display(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node *head = NULL, *newNode, *temp;
    int i;
    // Create initial list
    newNode = (struct Node*)malloc(sizeof(struct Node));
```

```
newNode->data = 10;
newNode->next = NULL;
head = newNode;
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = 20;
newNode->next = NULL;
head->next = newNode;
printf("Initial List:\n");
display(head);
// Insert at beginning
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = 5;
newNode->next = head;
head = newNode;
printf("\nAfter inserting at beginning:\n");
display(head);
// Insert in middle
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = 15;
temp = head;
for (i = 1; i < 2; i++)
    temp = temp->next;
newNode->next = temp->next;
temp->next = newNode;
printf("\nAfter inserting in middle:\n");
display(head);
// Insert at end
newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = 25;
newNode->next = NULL;
temp = head;
while (temp->next != NULL)
    temp = temp->next;
```

```
    temp->next = newNode;
    printf("\nAfter inserting at end:\n");
    display(head);
    return 0;
}
```

OUTPUT: -

```
Initial List:
10 -> 20 -> NULL

After inserting at beginning:
5 -> 10 -> 20 -> NULL

After inserting in middle:
5 -> 10 -> 15 -> 20 -> NULL

After inserting at end:
5 -> 10 -> 15 -> 20 -> 25 -> NULL


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

Nodes were successfully inserted at beginning, middle, and end of a singly linked list.

(e) Program to Insert a Node at Beginning, Middle, and End of a Circular Linked List

PROCEDURE: -

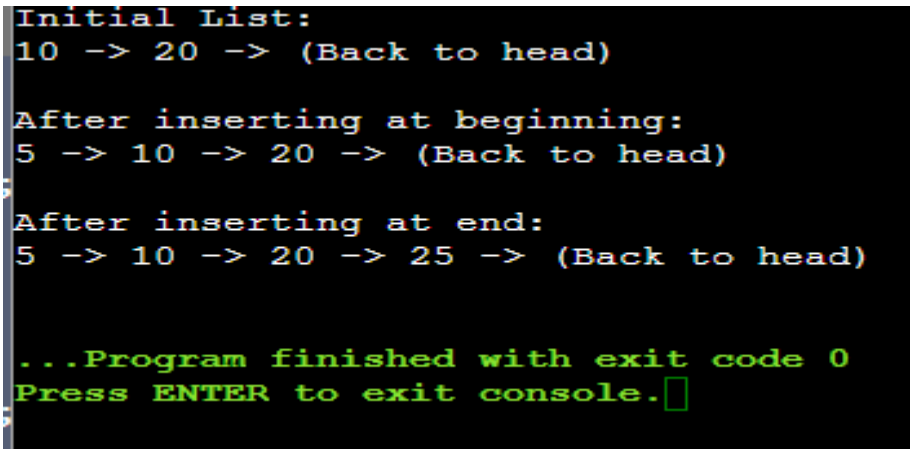
1. Define a structure Node with data and next pointer.
2. Create a circular linked list where the last node points to the head.
3. Insert nodes at:
    - Beginning
    - Middle
    - End
4. Update the circular links correctly.
5. Display the list after each operation.

PROGRAM (CODE): -

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

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

void display(struct Node *head) {
    struct Node *temp = head;
    if (head == NULL) return;
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(Back to head)\n");
}

int main() {
    struct Node *head = NULL, *newNode, *temp;

    // Create circular linked list with one node
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = 10;
    newNode->next = newNode;
    head = newNode;
    // Insert at end
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = 20;
```

```
    newNode->next = head;
    head->next = newNode;
    printf("Initial List:\n");
    display(head);
    // Insert at beginning
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = 5;
    newNode->next = head;
    temp = head;
    while (temp->next != head)
        temp = temp->next;
    temp->next = newNode;
    head = newNode;

    printf("\nAfter inserting at beginning:\n");
    display(head);

    // Insert at end
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = 25;
    newNode->next = head;
    temp = head;
    while (temp->next != head)
        temp = temp->next;
    temp->next = newNode;

    printf("\nAfter inserting at end:\n");
    display(head);
    return 0;
}
```

OUTPUT: -



RESULT: -

Nodes were successfully inserted at beginning, middle, and end of a circular linked list.

(F) program using stack for expression evaluation

PROCEDURE: -

1. Declare a stack and initialize top = -1.
2. Read a postfix expression from the user.
3. Traverse each character:
    - If it's a digit, push it on the stack.
    - If it's an operator, pop two elements, perform the operation, and push the result.
4. Display the final result.

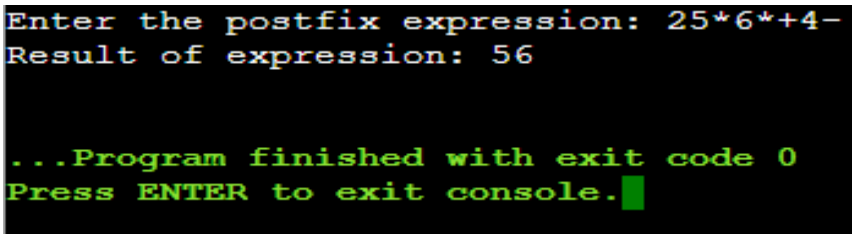
PROGRAM CODE: -

```
#include <stdio.h>
#include <ctype.h>
#define MAX 100
int stack[MAX];
int top = -1;
void push(int x) {
```

```
    stack[++top] = x;
}
int pop() {
    return stack[top--];
}
int main() {
    char exp[MAX];
    char *e;
    int n1, n2, n3, num;
    printf("Enter the postfix expression: ");
    scanf("%s", exp);
    e = exp;

    while (*e != '\0') {
        if (isdigit(*e)) {
            num = *e - '0';
            push(num);
        } else {
            n1 = pop();
            n2 = pop();
            switch (*e) {
                case '+': n3 = n2 + n1; break;
                case '-': n3 = n2 - n1; break;
                case '*': n3 = n2 * n1; break;
                case '/': n3 = n2 / n1; break;
            }
            push(n3);
        }
        e++;
    }
```

```
    printf("Result of expression: %d\n", pop());
    return 0;
}
```

OUTPUT: -



RESULT: -

The postfix expression was successfully evaluated using a stack.

(G) program to perform enqueue and dequeue operations on queue

PROCEDURE: -

1. Declare a queue array with front and rear pointers initialized to -1.
2. Implement functions for:
    - **enqueue(x):** Add element at the rear.
    - **dequeue():** Remove element from the front.
    - **display():** Show all elements.
3. Perform multiple enqueue and dequeue operations.
4. Display the final queue.

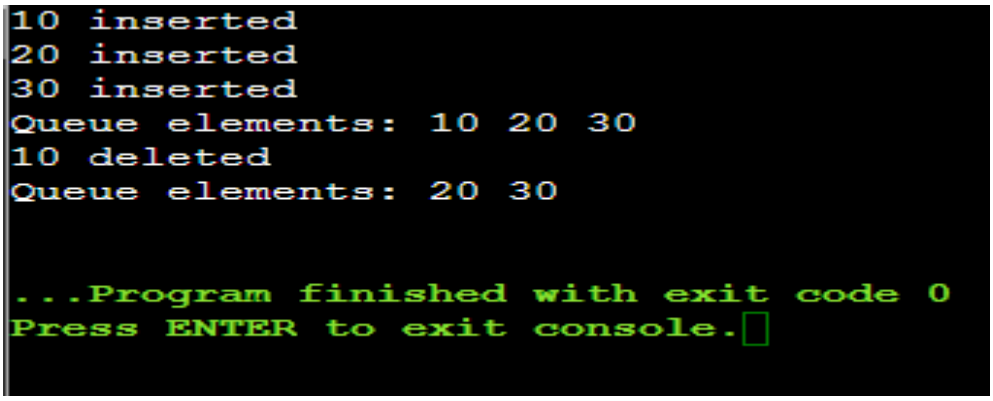
PROGRAM CODE: -

```
#include <stdio.h>
#define MAX 5

int queue[MAX];
int front = -1, rear = -1;
```

```
void enqueue(int x) {
    if (rear == MAX - 1)
        printf("Queue Overflow\n");
    else {
        if (front == -1)
            front = 0;
        queue[++rear] = x;
        printf("%d inserted\n", x);
    }
}

void dequeue() {
    if (front == -1 || front > rear)
        printf("Queue Underflow\n");
    else
        printf("%d deleted\n", queue[front++]);
}

void display() {
    if (front == -1)
        printf("Queue is empty\n");
    else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++)
            printf("%d ", queue[i]);
        printf("\n");
    }
}

int main() {
```

```
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();
    dequeue();
    display();
    return 0;
}
```

OUTPUT: -



RESULT: -

The program successfully performed enqueue and dequeue operations on a queue.

EXPERMENT NO: 3

IMPLEMENTATION OF TREE, HEAP, PRIORITY QUEUE, AND GRAPH ALGORITHMS

AIM: - To study and implement tree traversals, heap operations, priority queue operations, and graph algorithms such as DFS, BFS, Bellman-Ford, and Floyd-Warshall using C programming.

APPARATUS / TOOLS REQUIRED: -

- Computer System (Windows/Linux)
- MinGW Compiler
- Visual Studio Code (VS Code)

THEORY: -

1. **Tree Traversals:**
   A tree is a non-linear hierarchical structure.
   - Inorder (Left, Root, Right)
   - Preorder (Root, Left, Right)
   - Postorder (Left, Right, Root)
2. **Heap:**
   A heap is a complete binary tree that satisfies the heap property.
   - Max-Heap: Parent node ≥ child nodes.
   - Min-Heap: Parent node ≤ child nodes.
   - Common operations: Insertion, Deletion of root, Peek, and Empty check.
3. **Priority Queue:**
   A queue where each element has a priority, and elements with higher priority are dequeued before others.
4. **Graph:**
   A collection of nodes (vertices) connected by edges.

- DFS (Depth First Search): Explores as far as possible before backtracking.
- BFS (Breadth First Search): Explores all neighbors before moving deeper.
- Applications: Cycle detection, shortest paths, and articulation points.

5. **Bellman-Ford Algorithm:**
   Finds the shortest path from a source vertex to all others, even with negative edge weights.
   - Works in O(V × E) time complexity.
6. **Floyd-Warshall Algorithm:**
   A dynamic programming algorithm for finding shortest paths between all pairs of vertices in a weighted directed graph.
   - Works in $O(V^3)$ time complexity.

(A) program to perform in order, preorder, and post order traversal on a tree

PROCEDURE: -

1. Define a structure for a binary tree node.
2. Create nodes and link them to form a binary tree.
3. Write recursive functions for in order, preorder, and post order traversals.
4. Display the traversal orders.

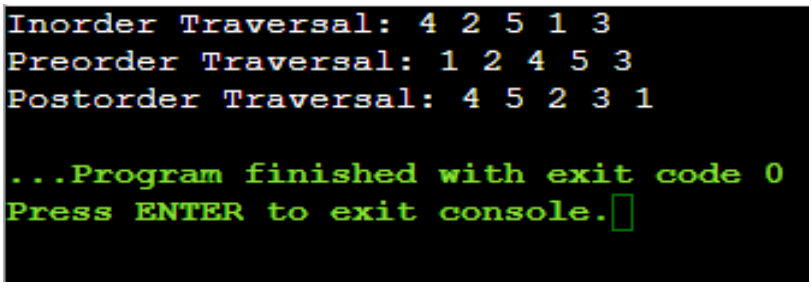
PROGRAME (CODE): -

```
#include  <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node *left, *right;
};
```

```
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}
void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}
void preorder(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void postorder(struct Node* root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
```

```
        printf("%d ", root->data);
    }
}
int main() {
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    printf("Inorder Traversal: ");
    inorder(root);
    printf("\nPreorder Traversal: ");
    preorder(root);
    printf("\nPostorder Traversal: ");
    postorder(root);
    return 0;
}
```

OUTPUT: -



RESULT: -

Tree traversals were successfully implemented using recursion.

(B) program to perform heap operations

PROCEDURE: -

1. Define an array to represent the heap.
2. Implement functions to:
    - Insert an element.
    - Delete the root.
    - Retrieve (peek) the root.
    - Check if heap is empty.
3. Use Max-Heap property for comparisons.

PROGRAM (CODE): -

```
#include <stdio.h>

#define MAX 100

int heap[MAX];

int size = 0;

void insert(int val) {

    int i = size++;

    heap[i] = val;

    while (i != 0 && heap[(i - 1) / 2] < heap[i]) {

        int temp = heap[i];

        heap[i] = heap[(i - 1) / 2];

        heap[(i - 1) / 2] = temp;

        i = (i - 1) / 2;

    }

    printf("%d inserted.\n", val);
```

```
}
void heapifyDown(int i) {
    int largest = i, left = 2*i+1, right = 2*i+2;
    if (left < size && heap[left] > heap[largest]) largest = left;
    if (right < size && heap[right] > heap[largest]) largest = right;
    if (largest != i) {
        int temp = heap[i];
        heap[i] = heap[largest];
        heap[largest] = temp;
        heapifyDown(largest);
    }
}
void deleteRoot() {
    if (size <= 0) {
        printf("Heap is empty!\n");
        return;
    }
    printf("Deleted root: %d\n", heap[0]);
    heap[0] = heap[--size];
    heapifyDown(0);
}
int peek() {
    if (size == 0) return -1;
    return heap[0];
```

```
}
int isEmpty() {
    return size == 0;
}
int main() {
    insert(40);
    insert(30);
    insert(50);
    insert(10);
    printf("Root element: %d\n", peek());
    deleteRoot();
    printf("Root after deletion: %d\n", peek());
    printf("Heap empty: %s\n", isEmpty() ? "Yes" : "No");
    return 0;
}
```

OUTPUT: -

```
40 inserted.
30 inserted.
50 inserted.
10 inserted.
Root element: 50
Deleted root: 50
Root after deletion: 40
Heap empty: No

...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

Heap operations (insert, delete root, peek, empty check) were successfully implemented.

(C) program to perform priority queue operations

PROCEDURE: -

1. Define a structure with data and priority.
2. Implement functions for:
    - Enqueue: Add element with priority.
    - Dequeue: Remove element with highest priority.
    - Peek: Display highest priority element.
    - Size and Empty check.
3. Display the queue after operations.

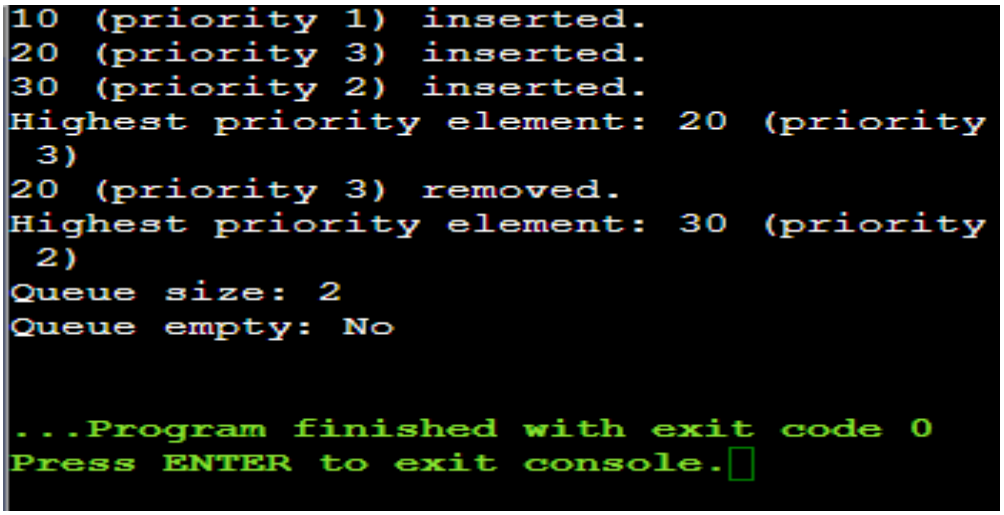
PROGRAM (CODE): -

```
#include <stdio.h>

#define MAX 100

int data[MAX], priority[MAX];

int size = 0;

void enqueue(int val, int pr) {

    int i = size++;

    data[i] = val;

    priority[i] = pr;

    printf("%d (priority %d) inserted.\n", val, pr);

}

void dequeue() {

    if (size == 0) {

        printf("Queue is empty.\n");
```

```
        return;
    }
    int highest = 0;
    for (int i = 1; i < size; i++)
        if (priority[i] > priority[highest])
            highest = i;
    printf("%d (priority %d) removed.\n", data[highest], priority[highest]);
    for (int i = highest; i < size - 1; i++) {
        data[i] = data[i + 1];
        priority[i] = priority[i + 1];
    }
    size--;
}
void peek() {
    if (size == 0) printf("Queue empty.\n");
    else {
        int highest = 0;
        for (int i = 1; i < size; i++)
            if (priority[i] > priority[highest])
                highest = i;
        printf("Highest priority element: %d (priority %d)\n", data[highest], priority[highest]);
    }
}
int main() {
```

```
    enqueue(10, 1);
    enqueue(20, 3);
    enqueue(30, 2);
    peek();
    dequeue();
    peek();
    printf("Queue size: %d\n", size);
    printf("Queue empty: %s\n", size == 0 ? "Yes" : "No");
    return 0;
}
```

OUTPUT: -



RESULT: -

Priority queue operations were successfully performed.

(D) program to perform graph operations

PROCEDURE: -

1. Represent graph using adjacency list.
2. Implement DFS for:

- Cycle detection
- Finding articulation points

3. Implement BFS for shortest path.

Program Code (Cycle Detection using DFS):

```
#include  <stdio.h>
#include <stdlib.h>
#define MAX 100
int adj[MAX][MAX], visited[MAX], V;
int dfs(int v, int parent) {
   visited[v] = 1;
   for (int i = 0; i < V; i++) {
      if (adj[v][i]) {
         if (!visited[i]) {
            if (dfs(i, v))
               return 1;
         } else if (i != parent)
            return 1;
      }
   }
   return 0;
}
int main() {
   printf("Enter number of vertices: ");
   scanf("%d", &V);
```

```
    printf("Enter adjacency matrix:\n");
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            scanf("%d", &adj[i][j]);
    for (int i = 0; i < V; i++)
        if (!visited[i] && dfs(i, -1)) {
            printf("Cycle detected.\n");
            return 0;
        }
    printf("No cycle found.\n");
    return 0;
}
```

OUTPUT: -

```
Enter number of vertices: 3
Enter adjacency matrix:
0
1
1
1
0
1
1
1
0
Cycle detected.

...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT:-

Cycle detection in a graph was successfully performed using DFS.

(E) PROGRAM TO APPLY BELLMAN-FORD ALGORITHM

PROCEDURE:-

1. Represent the graph as an edge list.
2. Initialize distances with infinity.
3. Relax all edges (V–1) times.
4. Check for negative weight cycles.
5. Display shortest paths.

PROGRAME (CODE): -

```
#include  <stdio.h>
#include <stdlib.h>
struct Edge {
    int src, dest, weight;
};
int main() {
    int V, E, i, j;
    printf("Enter vertices and edges: ");
    scanf("%d %d", &V, &E);
    struct Edge edge[E];
    printf("Enter edges (src dest weight):\n");
    for (i = 0; i < E; i++)
        scanf("%d %d %d", &edge[i].src, &edge[i].dest, &edge[i].weight);

    int dist[V];
    for (i = 0; i < V; i++)
```

```
        dist[i] = 9999;
    dist[0] = 0;
    for (i = 1; i < V; i++)
        for (j = 0; j < E; j++) {
            int u = edge[j].src, v = edge[j].dest, w = edge[j].weight;
            if (dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }
    printf("Vertex Distance from Source:\n");
    for (i = 0; i < V; i++)
        printf("%d\t%d\n", i, dist[i]);
    return 0;
}
```

OUTPUT: -

```
Enter vertices and edges: 2
3
Enter edges (src dest weight):
0
1
-1
0
2
4
1
2
3
Vertex Distance from Source:
0          0
1          -1


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

Bellman-Ford algorithm successfully found shortest paths with negative edges.

(F) program to apply floyd-warshall algorithm

PROCEDURE: -

1. Represent graph as adjacency matrix.
2. Initialize distance matrix.
3. Update distances for each vertex as intermediate.
4. Display the final shortest path matrix.

PROGRAM CODE:

```
#include <stdio.h>
#define INF 9999
#define V 4
void floydWarshall(int graph[V][V]) {
   int dist[V][V];
   for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
         dist[i][j] = graph[i][j];
   for (int k = 0; k < V; k++)
      for (int i = 0; i < V; i++)
         for (int j = 0; j < V; j++)
            if (dist[i][k] + dist[k][j] < dist[i][j])
               dist[i][j] = dist[i][k] + dist[k][j];
   printf("Shortest distance matrix:\n");
   for (int i = 0; i < V; i++) {
      for (int j = 0; j < V; j++)
         if (dist[i][j] == INF)
            printf("INF ");
         else
```

```
                printf("%d ", dist[i][j]);
            printf("\n");
        }
    }
    int main() {
        int graph[V][V] = {
            {0, 5, INF, 10},
            {INF, 0, 3, INF},
            {INF, INF, 0, 1},
            {INF, INF, INF, 0}
        };
        floydWarshall(graph);
        return 0;
    }
```

OUTPUT: -

```
Shortest distance matrix:
0  5  8  9
INF  0  3  4
INF  INF  0  1
INF  INF  INF  0


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

Floyd-Warshall algorithm successfully computed the shortest paths between all pairs of vertices.

EXPERMENT NO: 4

PROGRAMS TO IMPLEMENT SORTING, SEARCHING, AND HASHING TECHNIQUES

AIM: - To implement sorting, searching, and hashing techniques using C language for efficient data organization and retrieval.

APPARATUS / TOOLS REQUIRED: -

- Computer system (Windows/Linux)
- MinGW Compiler
- Visual Studio Code (VS Code)

THEORY: -

1. Sorting Algorithms

Sorting is the process of arranging data in a specific order (ascending or descending).

- **Insertion Sort:** Builds the sorted array one element at a time by comparing and inserting elements into their correct position.
- **Quicksort:** A divide-and-conquer algorithm that partitions the array around a pivot and recursively sorts the partitions.
- **Merge Sort:** Divides the array into halves, recursively sorts each half, and then merges the sorted halves.

2. Searching Algorithm

Binary Search: Efficiently finds an element in a sorted array by repeatedly dividing the search interval in half.

3. Hash Table

A hash table stores key–value pairs. It uses a **hash function** to compute an index for storing and retrieving data efficiently.

PROCEDURE: -

1. Start the program and include required header files.
2. Create functions for sorting (Insertion, Quick, Merge).
3. Input elements from the user into an array.
4. Perform sorting operations and display sorted arrays.
5. Implement Binary Search to find a given element.
6. Implement Hash Table functions (insert, retrieve, delete, check).
7. Display results of each operation.
8. Stop the program.

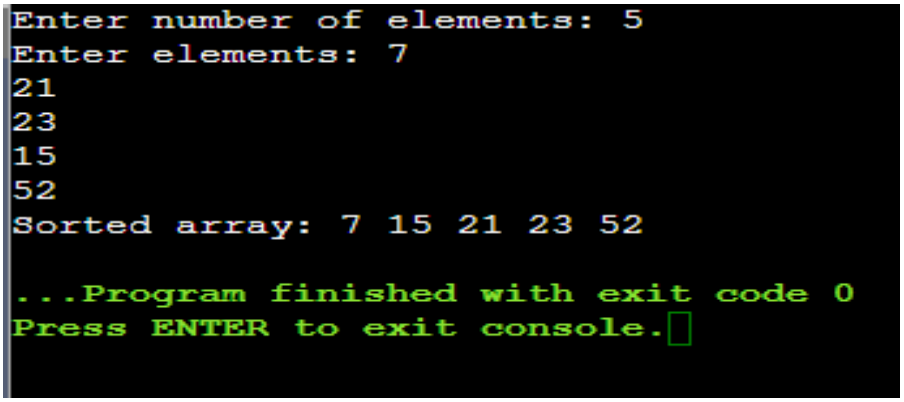
PROGRAM (CODE): -

(a) i. Program to Apply Insertion Sort

```
#include <stdio.h>

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

```
int main() {
    int arr[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    insertionSort(arr, n);
    printf("Sorted array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT: -



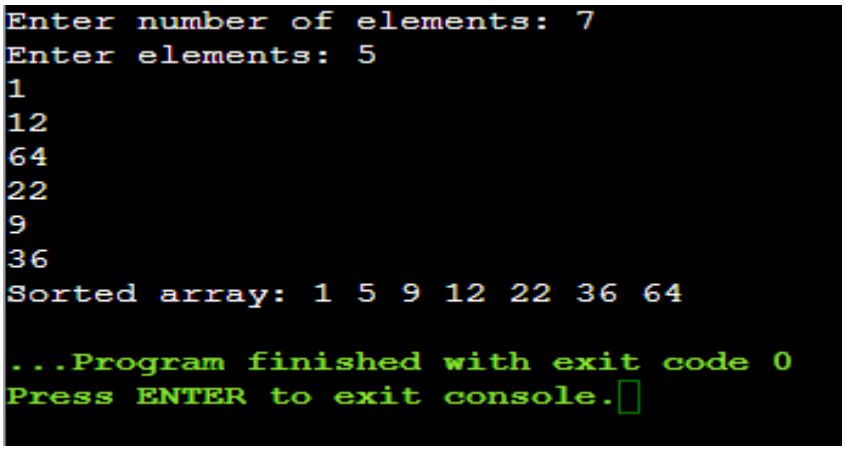
(a) i. Program to Apply Quicksort

```
#include <stdio.h>

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

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
int main() {
    int arr[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    quickSort(arr, 0, n - 1);
    printf("Sorted array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT: -



(a) i. Program to Apply Merge Sort

```
#include <stdio.h>
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m;
    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)
        arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];
    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = (l + r) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main() {
    int arr[100], n, i;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    mergeSort(arr, 0, n - 1);
    printf("Sorted array: ");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

OUTPUT: -

```
Enter number of elements: 4
Enter elements: 2
5
3
11
Sorted array: 2 3 5 11

...Program finished with exit code 0
Press ENTER to exit console.
```

(a) i. Program to Apply Binary Search

```
#include <stdio.h>
int binarySearch(int arr[], int n, int x) {
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (arr[mid] == x)
            return mid;
        else if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
int main() {
    int arr[100], n, i, x, result;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter sorted elements: ");
    for (i = 0; i < n; i++)
        scanf("%d", &arr[i]);
    printf("Enter element to search: ");
```

```
    scanf("%d", &x);
    result = binarySearch(arr, n, x);
    if (result == -1)
        printf("Element not found");
    else
        printf("Element found at position %d", result + 1);
    return 0;
}
```

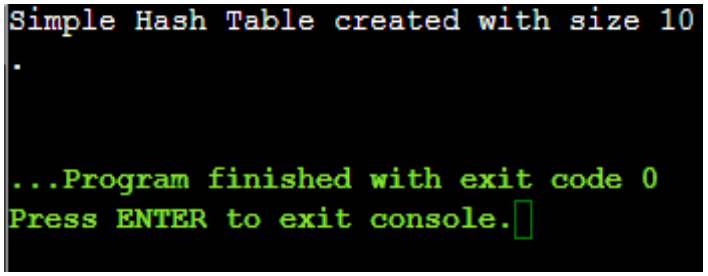
OUTPUT: -

```
Enter number of elements: 6
Enter sorted elements: 10
20
30
50
60
40
Enter element to search: 30
Element found at position 3

...Program finished with exit code 0
Press ENTER to exit console.
```

(b) Program to Create a Hash Table

```
#include <stdio.h>
#define SIZE 10

int hashTable[SIZE];

int hashFunction(int key) {
    return key % SIZE;
}

int main() {
    for (int i = 0; i < SIZE; i++)
        hashTable[i] = -1;

    printf("Simple Hash Table created with size %d.\n", SIZE);
    return 0;
}
```

OUTPUT: -



(c) Program to Perform Operations on Hash Table

```
#include <stdio.h>
#define SIZE 10
int hashTable[SIZE];

int hashFunction(int key) {
    return key % SIZE;
}
void insert(int key) {
     int index = hashFunction(key);
     while (hashTable[index] != -1)
         index = (index + 1) % SIZE;
    hashTable[index] = key;
    printf("Inserted %d at index %d\n", key, index);
}
int search(int key) {
    int index = hashFunction(key);
    int start = index;
    while (hashTable[index] != -1) {
        if (hashTable[index] == key)
            return index;
        index = (index + 1) % SIZE;
        if (index == start)
```

```
            return -1;
        }
        return -1;
    }
    void deleteKey(int key) {
        int index = search(key);
        if (index != -1) {
            hashTable[index] = -1;
            printf("Deleted %d\n", key);
        } else
            printf("Key not found\n");
    }
    int main() {
        for (int i = 0; i < SIZE; i++)
            hashTable[i] = -1;
        insert(10);
        insert(20);
        insert(30);
        int pos = search(20);
        if (pos != -1)
            printf("Element 20 found at index %d\n", pos);
        deleteKey(20);
        pos = search(20);
        if (pos == -1)
            printf("Element 20 not found after deletion\n");
        return 0;
    }
```

OUTPUT: -

```
Inserted 10 at index 0
Inserted 20 at index 1
Inserted 30 at index 2
Element 20 found at index 1
Deleted 20
Element 20 not found after deletion


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

The programs were successfully executed to perform sorting, searching, and hashing operations.

EXPERMENT NO: 5

PROGRAMS TO IMPLEMENT DYNAMIC PROGRAMMING AND DIVIDE & CONQUER TECHNIQUES

AIM: - To implement dynamic programming and divide-and-conquer techniques for solving string matching, shortest path, and min-max problems efficiently.

APPARATUS / TOOLS REQUIRED: -

- Computer System (Windows/Linux)
- MinGW Compiler
- Visual Studio Code (VS Code)

THEORY: -

1. **Longest Common Subsequence (LCS):**
    - The LCS of two strings is the longest sequence of characters that appear left-to-right (but not necessarily consecutively) in both strings.
    - Dynamic Programming is used to find the LCS efficiently by solving overlapping subproblems.
2. **Bellman-Ford Algorithm:**
    - The Bellman-Ford algorithm finds the shortest paths from a single source vertex to all other vertices in a weighted graph.
    - It works even when the graph contains negative edge weights.
3. **Divide and Conquer for Min and Max:**
    - The problem of finding the minimum and maximum element from an array can be solved using the divide and conquer strategy, which divides the array into subarrays, finds min and max for each, and then combines results.

PROCEDURE: -

1. Start the program and include necessary header files.
2. Input required data (strings, graph edges, or array elements).
3. Implement the algorithm for each problem as per requirement:
    - Use Dynamic Programming for LCS.
    - Use Bellman-Ford for shortest path detection.
    - Use Divide and Conquer for Min and Max finding.
4. Execute and display results.
5. Stop the program.

PROGRAM: -

(i) Program to Find Longest Common Subsequence (LCS)

```
#include <stdio.h>
#include <string.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int main() {
    char X[100], Y[100];
    int m, n, i, j;
    int L[100][100];
    printf("Enter first string: ");
    scanf("%s", X);
    printf("Enter second string: ");
    scanf("%s", Y);
```

```
    m = strlen(X);
    n = strlen(Y);
    for (i = 0; i <= m; i++) {
        for (j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;
            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
    printf("Length of Longest Common Subsequence: %d\n", L[m][n]);
    return 0;
}
```

OUTPUT: -

```
Enter first string: ABCDGH
Enter second string: AEDFIRH
Length of Longest Common Subsequence: 3


...Program finished with exit code 0
Press ENTER to exit console.
```

(ii) Program to Find Shortest Path using Bellman-Ford Algorithm

```
#include  <stdio.h>
#include <stdlib.h>
#include <limits.h>
struct Edge {
    int src, dest, weight;
};

struct Graph {
    int V, E;
    struct Edge* edge;
};
void BellmanFord(struct Graph* graph, int src) {
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }
```

```
    for (int j = 0; j < E; j++) {
        int u = graph->edge[j].src;
        int v = graph->edge[j].dest;
        int weight = graph->edge[j].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle\n");
            return;
        }
    }
    printf("Vertex\tDistance from Source\n");
    for (int i = 0; i < V; i++)
        printf("%d\t\t%d\n", i, dist[i]);
}
int main() {
    int V = 5, E = 8;
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*)malloc(E * sizeof(struct Edge));
    // Example graph
    graph->edge[0] = (struct Edge){0, 1, -1};
    graph->edge[1] = (struct Edge){0, 2, 4};
    graph->edge[2] = (struct Edge){1, 2, 3};
    graph->edge[3] = (struct Edge){1, 3, 2};
    graph->edge[4] = (struct Edge){1, 4, 2};
    graph->edge[5] = (struct Edge){3, 2, 5};
    graph->edge[6] = (struct Edge){3, 1, 1};
    graph->edge[7] = (struct Edge){4, 3, -3};
```

```
    BellmanFord(graph, 0);
    return 0;
}
```

OUTPUT: -

```
Vertex   Distance from Source
0                0
1                -1
2                2
3                -2
4                1


...Program finished with exit code 0
Press ENTER to exit console.
```

(iii) Program to Find Minimum and Maximum Using Divide and Conquer

```
#include <stdio.h>
struct Pair {
    int min;
    int max;
};
struct Pair findMinMax(int arr[], int low, int high) {
    struct Pair minmax, mml, mmr;
    int mid;
    if (low == high) {
        minmax.min = arr[low];
        minmax.max = arr[low];
        return minmax;
    }

    if (high == low + 1) {
```

```
        if (arr[low] > arr[high]) {
            minmax.max = arr[low];
            minmax.min = arr[high];
        } else {
            minmax.max = arr[high];
            minmax.min = arr[low];
        }
        return minmax;
    }
    mid = (low + high) / 2;
    mml = findMinMax(arr, low, mid);
    mmr = findMinMax(arr, mid + 1, high);
    minmax.min = (mml.min < mmr.min) ? mml.min : mmr.min;
    minmax.max = (mml.max > mmr.max) ? mml.max : mmr.max;

    return minmax;
}
int main() {
    int arr[] = {1000, 11, 445, 1, 330, 3000};
    int n = 6;
    struct Pair minmax = findMinMax(arr, 0, n - 1);
    printf("Minimum element: %d\n", minmax.min);
    printf("Maximum element: %d\n", minmax.max);
    return 0;
}
```

OUTPUT: -

```
Minimum element: 1
Maximum element: 3000


...Program finished with exit code 0
Press ENTER to exit console.
```

RESULT: -

The programs were successfully executed to demonstrate the use of Dynamic Programming and Divide & Conquer strategies in solving complex computational problems.

- The LCS algorithm correctly identified the longest common subsequence between two input strings.
- The Bellman-Ford algorithm efficiently computed the shortest path in a weighted graph, including negative edges.
- The Divide and Conquer approach accurately determined the minimum and maximum values from the given dataset with reduced comparisons.