CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

REGULATION 2023

LAB MANUAL

CS3201-DATA STRUCTURES USING C++

II SEMESTER

COMMON TO CSE, IT,AI&DS,AI&ML,CSBS,CZ BRANCHES

Name of the Student	:
Register Number	:
Year / Semester / Section	:
Batch	:

CONTENT

13.		Binary search using a recursive function	CO5	PO1,PO2,PO3		
14.		C++ program to implement the Insertion Sort algorithm	CO5	PO1,PO2,PO3		
15.		C++ program to implement separate chaining technique in hashing	CO5	PO1,PO2,PO3		

EX.NO: 1	**C++ PROGRAM TO IMPLEMENT SINGLY LINKED LIST**
DATE:	

AIM:

To write a C++ program for implementing singly linked list.

ALGORITHM:

Step 1: Start the program.
Step 2: Create a class for the node.
Step 3: Create a node.
Step 4: Create a singly linked list
Step 5: Insert into the list
Step 6: Delete from the list.
Step 7: Search the list.
Step 8: Count the no of nodes in the list.
Step 9: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for the linked list
class Node {
public:
   int data;
   Node* next;

Node(int val) {
      data = val;
      next = NULL;
   }
};

// LinkedList class to implement the operations
class LinkedList {
private:
   Node* head;

public:
LinkedList() {
      head = NULL;
   }
```

```
    // Function to insert an element at the beginning of the list
    void insertElement(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
        } else {
newNode->next = head;
            head = newNode;
        }
cout<<val<< " inserted into the list." <<endl;
    }

    // Function to insert an element at the end of the list
    void insertElementAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
        }
cout<<val<< " inserted at the end of the list." <<endl;
    }

    // Function to insert an element at a given position in the list
    void insertElementAtPosition(int val, int position) {
        Node* newNode = new Node(val);
        if (position < 1) {
cout<< "Invalid position. Insertion failed." <<endl;
            return;
        }

        if (position == 1) {
newNode->next = head;
            head = newNode;
cout<<val<< " inserted at position " << position << "." <<endl;
            return;
        }

        Node* current = head;
        int count = 1;

        while (count < position - 1 && current) {
            current = current->next;
            count++;
        }

        if (!current) {
```

```
cout<< "Invalid position. Insertion failed." <<endl;
            return;
        }

newNode->next = current->next;
        current->next = newNode;

cout<<val<< " inserted at position " << position << "." <<endl;
    }

    // Function to insert an element after a given node in the list
    void insertElementAfterNode(int val, int key) {
        Node* newNode = new Node(val);
        Node* current = head;

        while (current && current->data != key) {
            current = current->next;
        }

        if (!current) {
cout<< key << " not found in the list. Insertion failed." <<endl;
            return;
        }

newNode->next = current->next;
        current->next = newNode;

cout<<val<< " inserted after " << key << "." <<endl;
    }

    // Function to insert an element before a given node in the list
    void insertElementBeforeNode(int val, int key) {
        Node* newNode = new Node(val);
        Node* current = head;
        Node* prev = nullptr;

        while (current && current->data != key) {
prev = current;
            current = current->next;
        }

        if (!current) {
cout<< key << " not found in the list. Insertion failed." <<endl;
            return;
        }

        if (!prev) {
newNode->next = head;
            head = newNode;
        } else {
newNode->next = current;
```

```
prev->next = newNode;
        }

cout<<val<< " inserted before " << key << "." <<endl;
    }

    // Function to delete the first element from the list
    void deleteElementAtBeginning() {
        if (!head) {
cout<< "List is empty. Deletion failed." <<endl;
            return;
        }

        Node* temp = head;
        head = head->next;

        delete temp;
cout<< "First element deleted from the list." <<endl;
    }

    // Function to delete the last element from the list
    void deleteElementAtEnd() {
        if (!head) {
cout<< "List is empty. Deletion failed." <<endl;
            return;
        }

        Node* current = head;
        Node* prev = nullptr;

        while (current->next) {
prev = current;
            current = current->next;
        }

        if (!prev) {
            // If there is only one element in the list
            head = nullptr;
        } else {
prev->next = nullptr;
        }

        delete current;
cout<< "Last element deleted from the list." <<endl;
    }

    // Function to delete an element at a given position in the list
    void deleteElementAtPosition(int position) {
        if (!head || position < 1) {
cout<< "List is empty or invalid position. Deletion failed." <<endl;
            return;
        }

        Node* current = head;
        Node* prev = nullptr;
        int count = 1;

        while (count < position && current) {
prev = current;
            current = current->next;
            count++;
        }

        if (!current) {
cout<< "Invalid position. Deletion failed." <<endl;
            return;
        }

        if (!prev) {
            head = current->next;
        } else {
prev->next = current->next;
        }

        delete current;
cout<< "Element at position " << position << " deleted from the list." <<endl;
    }

    // Function to delete an element by its value from the list
    void deleteElementByValue(int val) {
        if (!head) {
cout<< "List is empty. Deletion failed." <<endl;
            return;
        }

        Node* current = head;
        Node* prev = nullptr;

        while (current && current->data != val) {
prev = current;
            current = current->next;
        }

        if (!current) {
cout<<val<< " not found in the list. Deletion failed." <<endl;
            return;
        }

        if (!prev) {
            head = current->next;
        } else {
prev->next = current->next;
```

```
        }

        delete current;
cout<<val<< " deleted from the list." <<endl;
    }

    // Function to search for a key element in the list
    bool searchElement(int val) {
        Node* current = head;

        while (current) {
            if (current->data == val) {
cout<<val<< " found in the list." <<endl;
                return true;
            }
            current = current->next;
        }

cout<<val<< " not found in the list." <<endl;
        return false;
    }

    // Function to count the number of nodes in the list
    int countNodes() {
        int count = 0;
        Node* current = head;

        while (current) {
            count++;
            current = current->next;
        }

cout<< "Number of nodes in the list: " << count <<endl;
        return count;
    }

    // Function to traverse and display the elements of the list
    void traverseList() {
        Node* current = head;
cout<< "Linked List: ";
        while (current) {
cout<< current->data << " ";
            current = current->next;
        }
cout<<endl;
    }
};

int main() {
    LinkedList myList;
    char choice;
    int value, position, key;

    do {
cout<< "\nMenu:\n";
cout<< "a) Insert an element at the beginning of the list.\n";
cout<< "b) Insert an element at the end of the list.\n";
cout<< "c) Insert an element at a given position in the list.\n";
cout<< "d) Insert an element after a given node in the list.\n";
cout<< "e) Insert an element before a given node in the list.\n";
cout<< "f) Delete an element from the beginning of the list.\n";
cout<< "g) Delete an element from the end of the list.\n";
cout<< "h) Delete an element at a given position in the list.\n";
cout<< "i) Delete an element by value from the list.\n";
cout<< "j) Search for a key element in the list.\n";
cout<< "k) Count the number of nodes in the list.\n";
cout<< "l) Traverse and display the list.\n";
cout<< "m) Exit.\n";
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 'a':
cout<< "Enter the value to insert: ";
cin>> value;
myList.insertElement(value);
                break;
            case 'b':
cout<< "Enter the value to insert at the end: ";
cin>> value;
myList.insertElementAtEnd(value);
                break;
            case 'c':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the position to insert: ";
cin>> position;
myList.insertElementAtPosition(value, position);
                break;
            case 'd':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the key after which to insert: ";
cin>> key;
myList.insertElementAfterNode(value, key);
                break;
            case 'e':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the key before which to insert: ";
cin>> key;
myList.insertElementBeforeNode(value, key);
```

```
                break;
            case 'f':
myList.deleteElementAtBeginning();
                break;
            case 'g':
myList.deleteElementAtEnd();
                break;
            case 'h':
cout<< "Enter the position to delete: ";
cin>> position;
myList.deleteElementAtPosition(position);
                break;
            case 'i':
cout<< "Enter the value to delete: ";
cin>> value;
myList.deleteElementByValue(value);
                break;
            case 'j':
cout<< "Enter the key element to search: ";
cin>> value;
myList.searchElement(value);
                break;
            case 'k':
myList.countNodes();
                break;
            case 'l':
myList.traverseList();
                break;
            case 'm':
cout<< "Exiting the program.\n";
                break;
            default:
cout<< "Invalid choice. Please enter a valid option.\n";
        }
    } while (choice != 'm');

    return 0;
}
```

OUTPUT:

INFERENCE:
A singly linked list is a unidirectional linked list. So, you can only traverse it in one direction, i.e., from head node to tail node. Through this program we learnt how to implement some operations on singly linked list.

RESULT:

The program was successfully executed and the output was got.

EX.NO:2	**C++ PROGRAM TO IMPLEMENT DOUBLY LINKED LIST**
DATE:	

AIM:

To write a C++ program for implementing doubly linked list.

ALGORITHM:

```
Step 1: Start the program.
Step 2: Create a class for the node.
Step 3: Create a node.
Step 4: Create a doubly linked list
Step 5: Insert into the list
Step 6: Delete from the list.
Step 7: Print the list in reverse order.
Step 8: Stop the program.
```

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for the doubly linked list
class Node {
public:
    int data;
    Node* next;
    Node* prev;

Node(int val) {
        data = val;
        next = prev = NULL;
    }
};

// DoublyLinkedList class to implement the operations
class DoublyLinkedList {
private:
    Node* head;

public:
DoublyLinkedList() {
        head = NULL;
    }
    // Function to insert an element at the beginning of the list
    void insertElement(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
        } else {
newNode->next = head;
            head->prev = newNode;
            head = newNode;
        }
cout<<val<< " inserted into the list." <<endl;
    }

    // Function to insert an element at the end of the list
    void insertElementAtEnd(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
        } else {
            Node* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newNode;
newNode->prev = current;
        }
cout<<val<< " inserted at the end of the list." <<endl;
    }

    // Function to delete the first element from the list
    void deleteElementAtBeginning() {
        if (!head) {
cout<< "List is empty. Deletion failed." <<endl;
            return;
        }

        Node* temp = head;
        head = head->next;
        if (head) {
            head->prev = NULL;
        }

        delete temp;
cout<< "First element deleted from the list." <<endl;
    }

    // Function to insert an element at a given position in the list
    void insertElementAtPosition(int val, int position) {
        Node* newNode = new Node(val);
        if (!head || position < 1) {
cout<< "List is empty or invalid position. Insertion failed." <<endl;
```

```
            return;
        }

        Node* current = head;
        int count = 1;

        while (count < position - 1 && current) {
            current = current->next;
            count++;
        }

        if (!current) {
cout<< "Invalid position. Insertion failed." <<endl;
            return;
        }

newNode->next = current->next;
newNode->prev = current;
        if (current->next) {
            current->next->prev = newNode;
        }
        current->next = newNode;

cout<<val<< " inserted at position " << position << "." <<endl;
    }

    // Function to insert an element after a given node in the list
    void insertElementAfterNode(int val, int key) {
        Node* newNode = new Node(val);
        Node* current = head;

        while (current && current->data != key) {
            current = current->next;
        }

        if (!current) {
cout<< key << " not found in the list. Insertion failed." <<endl;
            return;
        }

newNode->next = current->next;
newNode->prev = current;
        if (current->next) {
            current->next->prev = newNode;
        }
        current->next = newNode;

cout<<val<< " inserted after " << key << "." <<endl;
    }

    // Function to insert an element before a given node in the list
    void insertElementBeforeNode(int val, int key) {
        Node* newNode = new Node(val);
        Node* current = head;

        while (current && current->data != key) {
            current = current->next;
        }

        if (!current) {
cout<< key << " not found in the list. Insertion failed." <<endl;
            return;
        }

newNode->next = current;
newNode->prev = current->prev;
        if (current->prev) {
            current->prev->next = newNode;
        } else {
            head = newNode;
        }
        current->prev = newNode;

cout<<val<< " inserted before " << key << "." <<endl;
    }

    // Function to delete the last element from the list
    void deleteElementAtEnd() {
        if (!head) {
cout<< "List is empty. Deletion failed." <<endl;
            return;
        }

        Node* current = head;
        while (current->next) {
            current = current->next;
        }

        if (current->prev) {
            current->prev->next = nullptr;
        } else {
            head = nullptr;
        }

        delete current;
cout<< "Last element deleted from the list." <<endl;
    }

    // Function to delete an element at a given position in the list
    void deleteElementAtPosition(int position) {
        if (!head || position < 1) {
cout<< "List is empty or invalid position. Deletion failed." <<endl;
```

```
            return;
        }

        Node* current = head;
        int count = 1;

        while (count < position && current) {
            current = current->next;
            count++;
        }

        if (!current) {
cout<< "Invalid position. Deletion failed." <<endl;
            return;
        }

        if (current->prev) {
            current->prev->next = current->next;
        } else {
            head = current->next;
        }

        if (current->next) {
            current->next->prev = current->prev;
        }

        delete current;
cout<< "Element at position " << position << " deleted from the list." <<endl;
    }

    // Function to traverse and display the elements of the list
    void traverseList() {
        Node* current = head;
cout<< "Doubly Linked List: ";
        while (current) {
cout<< current->data << " ";
            current = current->next;
        }
cout<<endl;
    }
};

int main() {
DoublyLinkedListmyList;
    char choice;
    int value, position, key;

    do {
cout<< "\nMenu:\n";
cout<< "a) Insert an element at the beginning of the list.\n";
cout<< "b) Insert an element at the end of the list.\n";
```

```
cout<< "c) Insert an element at a given position in the list.\n";
cout<< "d) Insert an element after a given node in the list.\n";
cout<< "e) Insert an element before a given node in the list.\n";
cout<< "f) Delete an element from the beginning of the list.\n";
cout<< "g) Delete an element from the end of the list.\n";
cout<< "h) Delete an element at a given position in the list.\n";
cout<< "i) Traverse and display the list.\n";
cout<< "j) Exit.\n";
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 'a':
cout<< "Enter the value to insert: ";
cin>> value;
myList.insertElement(value);
                break;
            case 'b':
cout<< "Enter the value to insert at the end: ";
cin>> value;
myList.insertElementAtEnd(value);
                break;
            case 'c':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the position to insert: ";
cin>> position;
myList.insertElementAtPosition(value, position);
                break;
            case 'd':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the key after which to insert: ";
cin>> key;
myList.insertElementAfterNode(value, key);
                break;
            case 'e':
cout<< "Enter the value to insert: ";
cin>> value;
cout<< "Enter the key before which to insert: ";
cin>> key;
myList.insertElementBeforeNode(value, key);
                break;
            case 'f':
myList.deleteElementAtBeginning();
                break;
            case 'g':
myList.deleteElementAtEnd();
                break;
            case 'h':
cout<< "Enter the position to delete: ";
```

```
cin>> position;
myList.deleteElementAtPosition(position);
                break;
            case 'i':
myList.traverseList();
                break;
            case 'j':
cout<< "Exiting the program.\n";
                break;
            default:
cout<< "Invalid choice. Please enter a valid option.\n";
        }
    } while (choice != 'j');

    return 0;
}
```

OUTPUT

INFERENCE:

A doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Through this program we learnt how to implement some operations on doubly linked list

RESULT

The program was successfully executed and the output was got.

EX.NO: 3	C++ PROGRAM TO PERFORM POLYNOMIAL ADDITION USING SINGLY LINKED LIST
DATE:	

AIM:

To write a C++ program to Perform Polynomial Addition using singly linked list.

ALGORITHM:

Step 1: Start the program.
Step 2: Insert a term into Polynomial 1.
Step 3: Insert a term into Polynomial 2
Step 4: Display Polynomial 1
Step 5: Display Polynomial 2
Step 6: Add Polynomials
Step 7: Print the result.
Step 8: Stop the program.

PROGRAM:

```
#include <iostream>
#include <cmath>

using namespace std;

// Node class for the polynomial term
class Term {
   public:
   int coefficient;
   int exponent;
   Term* next;

Term(int coeff, int exp) : coefficient(coeff), exponent(exp), next(nullptr) {}
};

// Polynomial class to perform polynomial operations
class Polynomial {
private:
   Term* head;

public:
Polynomial() : head(nullptr) {}

   // Function to insert a term into the polynomial
   void insertTerm(int coeff, int exp) {
      Term* newTerm = new Term(coeff, exp);
      if (!head) {
            head = newTerm;
        } else {
            Term* current = head;
            while (current->next) {
                current = current->next;
            }
            current->next = newTerm;
        }
    }

    // Function to display the polynomial
    void displayPolynomial() {
        if (!head) {
cout<< "Polynomial is empty." <<endl;
            return;
        }

        Term* current = head;
        while (current) {
cout<< current->coefficient << "x^" << current->exponent;
            if (current->next) {
cout<< " + ";
            }
            current = current->next;
        }
cout<<endl;
    }

    // Function to add two polynomials
    Polynomial addPolynomials(const Polynomial& poly) {
        Polynomial result;
        Term* term1 = head;
        Term* term2 = poly.head;

        while (term1 && term2) {
            if (term1->exponent > term2->exponent) {
result.insertTerm(term1->coefficient, term1->exponent);
                term1 = term1->next;
            } else if (term1->exponent < term2->exponent) {
result.insertTerm(term2->coefficient, term2->exponent);
                term2 = term2->next;
            } else {
                int sumCoeff = term1->coefficient + term2->coefficient;
result.insertTerm(sumCoeff, term1->exponent);
                term1 = term1->next;
                term2 = term2->next;
            }
        }

        // Add remaining terms from poly1 or poly2
        while (term1) {
```

```
result.insertTerm(term1->coefficient, term1->exponent);
            term1 = term1->next;
        }

        while (term2) {
result.insertTerm(term2->coefficient, term2->exponent);
            term2 = term2->next;
        }

        return result;
    }
};

int main() {
    Polynomial poly1, poly2, result;
    char choice;
    int coeff, exp;

    do {
cout<< "\nMenu:\n";
cout<< "a) Insert a term into Polynomial 1.\n";
cout<< "b) Insert a term into Polynomial 2.\n";
cout<< "c) Display Polynomial 1.\n";
cout<< "d) Display Polynomial 2.\n";
cout<< "e) Add Polynomials.\n";
cout<< "f) Exit.\n";
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 'a':
cout<< "Enter coefficient and exponent for Polynomial 1 term: ";
cin>>coeff>> exp;
                poly1.insertTerm(coeff, exp);
                break;
            case 'b':
cout<< "Enter coefficient and exponent for Polynomial 2 term: ";
cin>>coeff>> exp;
                poly2.insertTerm(coeff, exp);
                break;
            case 'c':
cout<< "Polynomial 1: ";
                poly1.displayPolynomial();
                break;
            case 'd':
cout<< "Polynomial 2: ";
                poly2.displayPolynomial();
                break;
            case 'e':
                result = poly1.addPolynomials(poly2);
cout<< "Resultant Polynomial (Polynomial 1 + Polynomial 2): ";
```

```
result.displayPolynomial();
                break;
            case 'f':
cout<< "Exiting the program.\n";
                break;
            default:
cout<< "Invalid choice. Please enter a valid option.\n";
        }
    } while (choice != 'f');

    return 0;
}
```

OUTPUT:

INFERENCE:

Polynomials are algebraic expressions that consist of constants and variables of different powers. In this program we learnt to add two polynomials.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 4	**C++ PROGRAM TO IMPLEMENT STACK OPERATIONS**
DATE:	

AIM:

To write a C++ program to implement stack operations.

ALGORITHM:

Step 1: Start the program.
Step 2: Create a node.
Step 3: Create a pointer to top of the stack.
Step 4: Do the push operation.
Step 5: Do the pop operation.
Step 6: Go to the top of the stack.
Step 7: Check if the stack is empty.
Step 8: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for linked list
class Node {
   public:
   int data;
   Node* next;
};

// Stack class
class Stack {
private:
   Node* top; // Pointer to the top of the stack

public:
   Stack(){
      top=NULL;
   }

   // Function to push a new element onto the stack
   void push(int value) {
      Node* newNode = new Node;
newNode->data = value;
newNode->next = top;
      top = newNode;
cout<< "Pushed: " << value <<endl;
   }

   // Function to pop the top element from the stack
   void pop() {
      if (isEmpty()) {
cout<< "Stack is empty. Cannot pop." <<endl;
         return;
      }
      Node* temp = top;
      top = top->next;
cout<< "Popped: " << temp->data <<endl;
      delete temp;
   }

   // Function to get the top element of the stack
   int getTop() {
      if (isEmpty()) {
cout<< "Stack is empty." <<endl;
         return -1; // Return a sentinel value to indicate an empty stack
      }
      return top->data;
   }

   // Function to check if the stack is empty
   bool isEmpty() {
      return top == NULL;
   }
};

int main() {
   Stack stack;
   int choice, value;

   do {
cout<< "Stack Operations Menu:" <<endl;
cout<< "1. Push" <<endl;
cout<< "2. Pop" <<endl;
cout<< "3. Top" <<endl;
cout<< "4. Is Empty" <<endl;
cout<< "5. Exit" <<endl;
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 1:
cout<< "Enter the value to push: ";
cin>> value;
stack.push(value);
                break;
            case 2:
stack.pop();
                break;
            case 3:
                value = stack.getTop();
                if (value != -1) {
cout<< "Top element: " << value <<endl;
                }
                break;
            case 4:
cout<< (stack.isEmpty() ? "Stack is empty." : "Stack is not empty.") <<endl;
                break;
            case 5:
cout<< "Exiting program." <<endl;
                break;
            default:
cout<< "Invalid choice. Please try again." <<endl;
        }
    } while (choice != 5);

    return 0;
}
```

OUTPUT:

INFERENCE:
A Stack is a linear data structure that holds a linear, ordered sequence of elements. In this program we learnt to implement stack operations.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 5	C++ PROGRAMS TO IMPLEMENT A QUEUE OPERATIONS USING A LINKED LIST
DATE:	

AIM:

To write a C++ program to implement a queue operations using a linked list

ALGORITHM:

Step 1: Start the program.
Step 2: Create a node.
Step 3: Create a queue.
Step 4: Add an element to the queue.
Step 5: Delete an element from the queue.
Step 6: Get the front element in the queue.
Step 7: Check if the queue is empty.
Step 8: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for linked list
class Node {
   public:
   int data;
   Node* next;
};

// Queue class
class Queue {
private:
   Node* front; // Pointer to the front of the queue
   Node* rear;  // Pointer to the rear of the queue

public:
   Queue(){
      front = NULL;
      rear = NULL;
   }
   // Function to enqueue a new element into the queue
   void enqueue(int value) {
      Node* newNode = new Node;
newNode->data = value;
newNode->next = NULL;

      if (isEmpty()) {
         front = rear = newNode;
      } else {
         rear->next = newNode;
         rear = newNode;
      }

cout<< "Enqueued: " << value <<endl;
   }

   // Function to dequeue the front element from the queue
   void dequeue() {
      if (isEmpty()) {
cout<< "Queue is empty. Cannot dequeue." <<endl;
         return;
      }

      Node* temp = front;
      front = front->next;

      if (front == NULL) {
         rear = NULL;
      }

cout<< "Dequeued: " << temp->data <<endl;
      delete temp;
   }

   // Function to get the front element of the queue
   int getFront() {
      if (isEmpty()) {
cout<< "Queue is empty." <<endl;
         return -1; // Return a sentinel value to indicate an empty queue
      }
      return front->data;
   }

   // Function to check if the queue is empty
   bool isEmpty() {
        return front == NULL;
    }

    // Function to display the elements of the queue
    void displayQueue() {
        if (isEmpty()) {
cout<< "Queue is empty." <<endl;
            return;
        }

        Node* current = front;
cout<< "Queue elements: ";
        while (current) {
cout<< current->data << " ";
            current = current->next;
        }
cout<<endl;
    }
};

int main() {
    Queue queue;
    int choice, value;

    do {
cout<< "\nQueue Operations Menu:" <<endl;
cout<< "1. Enqueue" <<endl;
cout<< "2. Dequeue" <<endl;
cout<< "3. Front" <<endl;
cout<< "4. Is Empty" <<endl;
cout<< "5. Display Queue" <<endl;
cout<< "6. Exit" <<endl;
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 1:
cout<< "Enter the value to enqueue: ";
cin>> value;
queue.enqueue(value);
                break;
            case 2:
queue.dequeue();
                break;
            case 3:
```

```
            value = queue.getFront();
            if (value != -1) {
cout<< "Front element: " << value <<endl;
            }
            break;
        case 4:
cout<< (queue.isEmpty() ? "Queue is empty." : "Queue is not empty.") <<endl;
            break;
        case 5:
queue.displayQueue();
            break;
        case 6:
cout<< "Exiting program." <<endl;
            break;
        default:
cout<< "Invalid choice. Please try again." <<endl;
      }
   } while (choice != 6);

   return 0;
}
```

OUTPUT:

INFERENCE:

A queue follows the FIFO (First In First Out) method and is open at both of its ends. In this program we learnt to implement queue operations.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 6	CONVERT INFIX TO POSTFIX EXPRESSION USING STACK
DATE:	

AIM:

To implement a C++ program to convert Infix to Postfix Expression using Stack

ALGORITHM:

Step 1: Start the program.
Step 2: Read the infix expression.
Step 3: Convert to postfix notation.
Step 4: Print the postfix expression.
Step 5: Stop the program.

PROGRAM:

```
#include <iostream>

#include <stack>

#include <string>

#include <cctype>

using namespace std;

// Function to check if a character is an operator

bool isOperator(char ch) {

    return (ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '%' || ch == '^');

}

// Function to get the precedence of an operator

int getPrecedence(char op) {

    if (op == '^')

        return 3;

    else if (op == '*' || op == '/' || op == '%')

        return 2;

    else if (op == '+' || op == '-')
```

```
        return 1;
    else
        return 0; // For non-operators
}

// Function to convert infix expression to postfix using stack STL
string infixToPostfix(const string& infix) {
    stack<char> operators;
    string postfix;

    for (char ch : infix) {
        if (isalnum(ch)) {
            // If character is an operand, add it to the postfix expression
            postfix += ch;
        } else if (ch == '(') {
            // Push opening parenthesis onto the stack
            operators.push(ch);
        } else if (ch == ')') {
            // Pop operators from the stack and add them to the postfix expression until '(' is encountered
            while (!operators.empty() && operators.top() != '(') {
                postfix += operators.top();
                operators.pop();
            }
            // Pop the '(' from the stack
            operators.pop();
        } else if (isOperator(ch)) {

            while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(ch)) {

                postfix += operators.top();

                operators.pop();

            }

            // Push the current operator onto the stack

            operators.push(ch);

        }

    }


    // Pop any remaining operators from the stack and add them to the postfix expression

    while (!operators.empty()) {

        postfix += operators.top();

        operators.pop();

    }


    return postfix;

}

int main() {

    string infixExpression;

    cout << "Enter an infix expression: ";

    getline(cin, infixExpression);

    cout << "Postfix Expression: " << infixToPostfix(infixExpression) << endl;

    return 0;}
```

OUTPUT:

INFERENCE:
Postfix expression can be defined as an expression in which all the operators are present after the operands. In this program we learnt to implement the conversion of infix to postfix notation.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 7	C++ PROGRAMS TO IMPLEMENT BINARY SEARCH TREE OPERATIONS
DATE:	

AIM:

. Write C++ programs to implement binary search tree operations.

ALGORITHM:

Step 1: Start the program.
Step 2: Create a node.
Step 3: Insert an element into a binary search tree.
Step 4: Delete an element from a binary search tree.
Step 5: Search for a key element in a binary search tree.
Step 6: Print the results.
Step 6: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for binary search tree
class TreeNode {
   public:
   int data;
TreeNode* left;
TreeNode* right;
};

// BinarySearchTree class
class BinarySearchTree {
private:
TreeNode* root;

public:
BinarySearchTree(){
       root=NULL;
   }

   // Function to insert an element into the binary search tree
TreeNode* insert(TreeNode* node, int value) {
        if (node == NULL) {
TreeNode* newNode = new TreeNode;
newNode->data = value;
newNode->left = newNode->right = NULL;
            return newNode;
        }

        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        }

        return node;
    }

    void insert(int value) {
        root = insert(root, value);
cout<< "Element inserted: " << value <<endl;
    }

    // Function to find the node with the minimum value in a binary search tree
TreeNode* findMin(TreeNode* node) {
        while (node->left != NULL) {
            node = node->left;
        }
        return node;
    }

    // Function to delete an element from the binary search tree
TreeNode* remove(TreeNode* node, int value) {
        if (node == NULL) {
            return NULL;
        }

        if (value < node->data) {
            node->left = remove(node->left, value);
        } else if (value > node->data) {
            node->right = remove(node->right, value);
        } else {
            // Node with the value to be deleted found

            // Case 1: Node with only one child or no child
            if (node->left == NULL) {
TreeNode* temp = node->right;
```

```
                delete node;
                return temp;
            } else if (node->right == NULL) {
TreeNode* temp = node->left;
                delete node;
                return temp;
            }

            // Case 2: Node with two children, get the inorder successor (smallest in the right
subtree)
TreeNode* temp = findMin(node->right);

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

            // Delete the inorder successor
            node->right = remove(node->right, temp->data);
        }

        return node;
    }

    void remove(int value) {
        if (search(root, value)) {
            root = remove(root, value);
cout<< "Element deleted: " << value <<endl;
        } else {
cout<< "Element not found. Cannot delete." <<endl;
        }
    }

    // Function to search for a key element in a binary search tree
    bool search(TreeNode* node, int key) {
        if (node == NULL) {
            return false;
        }

        if (key == node->data) {
            return true;
        } else if (key < node->data) {
            return search(node->left, key);
        } else {
            return search(node->right, key);
        }
    }
```

```
   bool search(int key) {
      return search(root, key);
   }

   // Function to display the binary search tree using in-order traversal
   void displayInOrder(TreeNode* node) {
      if (node != NULL) {
displayInOrder(node->left);
cout<< node->data << " ";
displayInOrder(node->right);
      }
   }

   void displayInOrder() {
cout<< "In-Order Traversal: ";
displayInOrder(root);
cout<<endl;
   }
};

int main() {
BinarySearchTreebst;
   int choice, value;

   do {
cout<< "Binary Search Tree Operations Menu:" <<endl;
cout<< "1. Insert" <<endl;
cout<< "2. Delete" <<endl;
cout<< "3. Search" <<endl;
cout<< "4. Display In-Order" <<endl;
cout<< "5. Exit" <<endl;
cout<< "Enter your choice: ";
cin>> choice;

      switch (choice) {
         case 1:
cout<< "Enter the value to insert: ";
cin>> value;
bst.insert(value);
            break;
         case 2:
cout<< "Enter the value to delete: ";
cin>> value;
bst.remove(value);
```

```
                break;
            case 3:
cout<< "Enter the value to search: ";
cin>> value;
cout<< (bst.search(value) ? "Element found." : "Element not found.") <<endl;
                break;
            case 4:
bst.displayInOrder();
                break;
            case 5:
cout<< "Exiting program." <<endl;
                break;
            default:
cout<< "Invalid choice. Please try again." <<endl;
        }

    } while (choice != 5);

    return 0;
}
```

OUTPUT:

INFERENCE:

A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. In this program we learnt to implement binary search tree operations.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 8	BINARY TREE TRAVERSAL
DATE:	

AIM:

To implement C++ program that use recursive functions to traverse the given binary tree.

ALGORITHM:

Step 1: Start the program.
Step 2: Node structure for binary tree.
Step 3: Create a structure for binary tree.
Step 4: Create a binary tree.
Step 5: Do the tree traversal-inorder.
Step 6: Do the tree traversal-preorder.
Step 7: Do the tree traversal-postorder.
Step 8: Print the results.
Step 9: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;

// Node class for binary tree
struct TreeNode {
   public:
   int data;
TreeNode* left;
TreeNode* right;
};

// BinaryTree class
class BinaryTree {
private:
TreeNode* root;

public:
BinaryTree(){
      root=NULL;
   }
   // Function to create a new node
TreeNode* createNode(int value) {
TreeNode* newNode = new TreeNode;
newNode->data = value;
newNode->left = newNode->right = NULL;
       return newNode;
   }

   // Function to insert an element into the binary tree
TreeNode* insert(TreeNode* node, int value) {
       if (node == NULL) {
          return createNode(value);
       }

       if (value < node->data) {
          node->left = insert(node->left, value);
       } else if (value > node->data) {
          node->right = insert(node->right, value);
       }

       return node;
   }

  void insert(int value) {
       root = insert(root, value);
cout<< "Element inserted: " << value <<endl;
   }

   // Function for recursive preorder traversal
   void preorderTraversal(TreeNode* node) {
       if (node != NULL) {
cout<< node->data << " ";
preorderTraversal(node->left);
preorderTraversal(node->right);
       }
   }

   // Function to initiate preorder traversal
   void preorderTraversal() {
cout<< "Preorder Traversal: ";
preorderTraversal(root);
cout<<endl;
   }

   // Function for recursive inorder traversal
   void inorderTraversal(TreeNode* node) {
      if (node != NULL) {
inorderTraversal(node->left);
cout<< node->data << " ";
inorderTraversal(node->right);
      }
   }

   // Function to initiate inorder traversal
   void inorderTraversal() {
cout<< "Inorder Traversal: ";
inorderTraversal(root);
cout<<endl;
   }

   // Function for recursive postorder traversal
   void postorderTraversal(TreeNode* node) {
      if (node != NULL) {
postorderTraversal(node->left);
postorderTraversal(node->right);
cout<< node->data << " ";
      }
   }

   // Function to initiate postorder traversal
   void postorderTraversal() {
cout<< "Postorder Traversal: ";
postorderTraversal(root);
cout<<endl;
   }
};

int main() {
BinaryTreebinaryTree;
   int choice, value;

   do {
cout<< "Binary Tree Traversal Menu:" <<endl;
cout<< "1. Insert Element" <<endl;
cout<< "2. Preorder Traversal" <<endl;
cout<< "3. Inorder Traversal" <<endl;
cout<< "4. Postorder Traversal" <<endl;
cout<< "5. Exit" <<endl;
cout<< "Enter your choice: ";
cin>> choice;
```

```
        switch (choice) {
            case 1:
cout<< "Enter the value to insert: ";
cin>> value;
binaryTree.insert(value);
                break;
            case 2:
binaryTree.preorderTraversal();
                break;
            case 3:
binaryTree.inorderTraversal();
                break;
            case 4:
binaryTree.postorderTraversal();
                break;
            case 5:
cout<< "Exiting program." <<endl;
                break;
            default:
cout<< "Invalid choice. Please try again." <<endl;
        }

    } while (choice != 5);

    return 0;
}
```

OUTPUT:

INFERENCE:
A binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree data structure with the key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree. In this program we learnt to implement binary tree traversal.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 9	IMPLEMENT AN AVL TREE
DATE:	

AIM:

Write C++ programs to implement an AVL Tree

ALGORITHM:

Step 1: Start the program.
Step 2: Create a Node for AVL Tree node.
Step 3: Create a class for AVL tree.
Step 4: Create a new node.
Step 5: Get the height of the tree.
Step 6: Do the left rotation.
Step 7: Do the right rotation.
Step 8: Get the balance factor of a node.
Step 9: Balance the tree through rotations of left and right.
Step 10: Do the tree traversals.
Step 11: Print the results.
Step 12: Stop the program.

PROGRAM:

```
#include <iostream>
#include <algorithm>

using namespace std;

// Node class for AVL Tree
class AVLNode {
public:
   int data;
AVLNode* left;
AVLNode* right;
   int height;
};

// AVLTree class
class AVLTree {
private:
AVLNode* root;
```

```
public:
AVLTree(){
        root=NULL;
    }

    // Function to create a new node
AVLNode* createNode(int value) {
AVLNode* newNode = new AVLNode;
newNode->data = value;
newNode->left = newNode->right = NULL;
newNode->height = 1; // New node is at height 1
        return newNode;
    }

    // Function to get the height of a node
    int getHeight(AVLNode* node) {
        return (node == NULL) ?0 : node->height;
    }

    // Function to update the height of a node
    void updateHeight(AVLNode* node) {
        node->height = 1 + max(getHeight(node->left), getHeight(node->right));
    }

    // Function to perform a right rotation
AVLNode* rightRotate(AVLNode* y) {
AVLNode* x = y->left;
AVLNode* T2 = x->right;

        x->right = y;
        y->left = T2;

updateHeight(y);
updateHeight(x);

        return x;
    }

    // Function to perform a left rotation
AVLNode* leftRotate(AVLNode* x) {
AVLNode* y = x->right;
AVLNode* T2 = y->left;

        y->left = x;
        x->right = T2;
```

```
updateHeight(x);
updateHeight(y);

        return y;
    }

    // Function to get the balance factor of a node
    int getBalance(AVLNode* node) {
        return (node == NULL) ?0 :getHeight(node->left) - getHeight(node->right);
    }

    // Function to insert an element into the AVL Tree
AVLNode* insert(AVLNode* node, int value) {
        if (node == NULL) {
            return createNode(value);
        }

        if (value < node->data) {
            node->left = insert(node->left, value);
        } else if (value > node->data) {
            node->right = insert(node->right, value);
        } else {
            // Duplicate values are not allowed in AVL Tree
            return node;
        }

        // Update height of the current node
updateHeight(node);

        // Get the balance factor to check if the node became unbalanced
        int balance = getBalance(node);

        // Left Left Case
        if (balance > 1 && value < node->left->data) {
            return rightRotate(node);
        }

        // Right Right Case
        if (balance < -1 && value > node->right->data) {
            return leftRotate(node);
        }

        // Left Right Case
        if (balance > 1 && value > node->left->data) {
            node->left = leftRotate(node->left);
            return rightRotate(node);
        }

        // Right Left Case
        if (balance < -1 && value < node->right->data) {
            node->right = rightRotate(node->right);
            return leftRotate(node);
        }

        return node;
    }

    void insert(int value) {
        root = insert(root, value);
cout<< "Element inserted: " << value <<endl;
    }

    // Function to perform an in-order traversal
    void inOrderTraversal(AVLNode* node) {
        if (node != NULL) {
inOrderTraversal(node->left);
cout<< node->data << " ";
inOrderTraversal(node->right);
        }
    }

    // Function to initiate in-order traversal
    void inOrderTraversal() {
cout<< "In-Order Traversal: ";
inOrderTraversal(root);
cout<<endl;
    }
};

int main() {
AVLTreeavlTree;
    int choice, value;

    do {
cout<< "AVL Tree Operations Menu:" <<endl;
cout<< "1. Insert Element" <<endl;
cout<< "2. In-Order Traversal" <<endl;
cout<< "3. Exit" <<endl;
cout<< "Enter your choice: ";
```

```
cin>> choice;

        switch (choice) {
            case 1:
cout<< "Enter the value to insert: ";
cin>> value;
avlTree.insert(value);
                break;
            case 2:
avlTree.inOrderTraversal();
                break;
            case 3:
cout<< "Exiting program." <<endl;
                break;
            default:
cout<< "Invalid choice. Please try again." <<endl;
        }

    } while (choice != 3);

    return 0;
}
```

OUTPUT:

INFERENCE:
AVL tree in data structures is a popular self-balancing binary search tree where the difference between the heights of left and right subtrees for any node does not exceed unity. In this program we learnt to implement AVL trees.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 10	**C++ PROGRAM THAT DEMONSTRATES B-TREE OPERATION**
DATE:	

AIM:

Write a C++ program that demonstrates B-Tree operation insertion, search, and display.

ALGORITHM:

Step 1: Start the program.
Step 2: Create a btree node.
Step 3: Create a function for btree node.
Step 4: Create a function to split the current node.
Step 5: Create a function to insert a key into Btree.
Step 6: Create a function to search a key in btree.
Step 7: Create a function to display the display the tree.
Step 8: Stop the program.

PROGRAM:

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

// B-Tree node class
class BTreeNode {
public:
   int *keys;
BTreeNode **children;
   bool leaf;
   int n; // Current number of keys
};

// Function to create a new B-Tree node
BTreeNode* createNode(bool leaf, int degree) {
BTreeNode* newNode = new BTreeNode;
newNode->leaf = leaf;
newNode->n = 0;
newNode->keys = new int[2 * degree - 1];
newNode->children = new BTreeNode *[2 * degree];
   return newNode;
}

// Function to split a child of the current node
void splitChild(BTreeNode* parentNode, int index, BTreeNode* child, int degree) {
```

```
BTreeNode* newNode = createNode(child->leaf, degree);
newNode->n = degree - 1;

   for (int i = 0; i< degree - 1; i++)
newNode->keys[i] = child->keys[i + degree];

   if (!child->leaf) {
      for (int i = 0; i< degree; i++)
newNode->children[i] = child->children[i + degree];
   }

   child->n = degree - 1;

   for (int i = parentNode->n; i> index; i--)
parentNode->children[i + 1] = parentNode->children[i];

parentNode->children[index + 1] = newNode;

   for (int i = parentNode->n - 1; i>= index; i--)
parentNode->keys[i + 1] = parentNode->keys[i];

parentNode->keys[index] = child->keys[degree - 1];
parentNode->n++;
}

// Function to insert a key into the B-Tree
void insert(BTreeNode*& root, int key, int degree) {
   if (!root) {
      root = createNode(true, degree);
      root->keys[0] = key;
      root->n = 1;
   } else {
      if (root->n == 2 * degree - 1) {
BTreeNode* newNode = createNode(false, degree);
newNode->children[0] = root;
splitChild(newNode, 0, root, degree);
         int i = 0;
         if (newNode->keys[0] < key)
i++;
         insert(newNode->children[i], key, degree);
         root = newNode;
      } else {
         int i = root->n - 1;
         while (i>= 0 && key < root->keys[i]) {
            root->keys[i + 1] = root->keys[i];
i--;
         }
         root->keys[i + 1] = key;
```

```
            root->n++;
        }
    }
}

// Function to search a key in the B-Tree
bool search(BTreeNode* root, int key) {
    if (!root)
        return false;

    int i = 0;
    while (i< root->n && key > root->keys[i])
i++;

    if (i< root->n && key == root->keys[i])
        return true;

    if (root->leaf)
        return false;

    return search(root->children[i], key);
}

// Function to display the B-Tree
void display(BTreeNode* root, int level) {
    if (root) {
        for (int i = root->n - 1; i>= 0; i--) {
            display(root->children[i + 1], level + 1);
            for (int j = 0; j < level; j++)
cout<< "    ";
cout<< root->keys[i] <<endl;
        }
        display(root->children[0], level + 1);
    }
}

int main() {
BTreeNode* root = nullptr;
    int degree;

cout<< "Enter the degree of B-Tree: ";
cin>> degree;

    while (true) {
cout<< "\nB-Tree Operations:\n";
cout<< "1. Insert Key\n";
cout<< "2. Search Key\n";
cout<< "3. Display B-Tree\n";
```

```
cout<< "4. Exit\n";

        int choice;
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 1: {
                int key;
cout<< "Enter key to insert: ";
cin>> key;
insert(root, key, degree);
cout<< "Key inserted successfully!\n";
                break;
            }
            case 2: {
                int key;
cout<< "Enter key to search: ";
cin>> key;
                if (search(root, key))
cout<< "Key found in the B-Tree.\n";
                else
cout<< "Key not found in the B-Tree.\n";
                break;
            }
            case 3:
cout<< "B-Tree:\n";
display(root, 0);
                break;
            case 4:
cout<< "Exiting program.\n";
                return 0;
            default:
cout<< "Invalid choice. Try again.\n";
        }
    }

    return 0;
}
```

OUTPUT:

INFERENCE:

B-tree is a self-balancing data structure commonly used in computer science for efficient storage and retrieval of large amounts of data. In this program we learnt to implement B-Tree operation insertion, search, and display.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 11	DIJKSTRA'S SINGLE SOURCE SHORTEST PATH ALGORITHM
DATE:	

AIM:

Write a C++ program for Dijkstra's single source shortest path algorithm.

ALGORITHM:

Step 1: Start the program.
Step 2: Set the number of vertices to 9.
Step 3: Find the vertex with minimum distance value, from the set of vertices not yet included in shortest path tree.
Step 4: Create a function to print the constructed distance.
Step5: Function that implements Dijkstra's single source shortest path algorithm for a graph represented using adjacency matrix representation.
Step 6: The output array. dist[i] will hold the shortest distance from src to i.
Step 7: sptSet[i] will be true if vertex i is included in shortest path tree or shortest distance from src to i is finalized.
Step 8: Initialize all distances as INFINITE and stpSet[] as false.
Step 9: Find shortest path for all vertices.
Step 10: Pick the minimum distance vertex from the set of vertices not yet processed. u is always equal to src in the first iteration.
Step 11: Mark the picked vertex as processed
Step 12: Update dist value of the adjacent vertices of the picked vertex.
Step 13: Update dist[v] only if is not in sptSet, there is an edge from u to v, and total weight of path from src to v through u is smaller than current value of dist[v]
Step 14: Print the constructed distance array
Step 15: Stop the program.

PROGRAM:

```
#include <iostream>
#include <limits.h>

#define V 9

using namespace std;

int minDistance(int dist[], bool sptSet[]) {
   int min = INT_MAX, min_index;

   for (int v = 0; v < V; v++)
      if (sptSet[v] == false &&dist[v] <= min)
         min = dist[v], min_index = v;
   return min_index;
}

void printSolution(int dist[], int n) {
cout<< "Vertex Distance from Source\n";
   for (int i = 0; i< V; i++)
cout<<i<< "\t\t\t\t" <<dist[i] <<endl;
}

void dijkstra(int graph[V][V], int src) {
   int dist[V];
   bool sptSet[V];

   for (int i = 0; i< V; i++)
dist[i] = INT_MAX, sptSet[i] = false;

dist[src] = 0;

   for (int count = 0; count < V - 1; count++) {
      int u = minDistance(dist, sptSet);

sptSet[u] = true;

      for (int v = 0; v < V; 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];
   }

printSolution(dist, V);
}

int main() {
   int graph[V][V] = {{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);

   return 0;
}
```

OUTPUT:

INFERENCE:
The Dijkstra Shortest Path algorithm computes the shortest path between nodes. The algorithm supports weighted graphs with positive relationship weights. In this program we learnt how to implement Dijkstra Shortest Path algorithm.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 12	PRIM'S MINIMUM SPANNING TREE
DATE:	

AIM:

To Write a C++ program for Prim's Minimum Spanning Tree (MST) algorithm.

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of nodes. Here it is 6.
Step 3: Create a node.
Step 4: Get the source node.
Step 5: Get the destination node.
Step 6: Get the weight of the node.
Step 7: Get the minimum for each time.
Step 8: Stop the program.

PROGRAM:

```
#include <iostream>

using namespace std;
class node
{
   public:
   int fr, to, cost;
}p[6];
int c = 0, temp1 = 0, temp = 0;
void prims(int *a, int b[][7], int i, int j)
{
   a[i] = 1;
   while (c < 6)
   {
      int min = 999;
      for (int i = 0; i< 7; i++)
      {
         if (a[i] == 1)
         {
            for (int j = 0; j < 7; )
            {
               if (b[i][j] >= min || b[i][j] == 0)
               {
j++;
               }
```

```
                    else if (b[i][j] < min)
                    {
                        min = b[i][j];
                        temp = i;
                        temp1 = j;
                    }
                }
            }
        }
        a[temp1] = 1;
        p[c].fr = temp;
        p[c].to = temp1;
        p[c].cost = min;
c++;
        b[temp][temp1] = b[temp1][temp]=1000;
    }
    for (int k = 0; k < 6; k++)
    {
cout<<"source node:"<<p[k].fr<<endl;
cout<<"destination node:"<<p[k].to<<endl;
cout<<"weight of node"<<p[k].cost<<endl;
    }
}
int main()
{
    int a[7];
    for (int i = 0; i< 7; i++)
    {
        a[i] = 0;
    }
    int b[7][7];
    for (int i = 0; i< 7; i++)
    {
cout<<"enter values for "<<(i+1)<<" row"<<endl;
        for (int j = 0; j < 7; j++)
        {
cin>>b[i][j];
        }
    }
prims(a, b, 0, 0);

}
```

OUTPUT:

INFERENCE:
The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. Through this program we have learnt how to implement Prim's Minimum Spanning Tree (MST) algorithm.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 13	BINARY SEARCH USING A RECURSIVE FUNCTION
DATE:	

AIM:

To Write a C++ programs to implement binary search using a recursive function.

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of elements in the array.
Step 3: Get the elements in the array.
Step 4: Get the element to be searched.
Step 5: Sort the array.
Step 6: Do the binary search.
Step 7: Binary search is done in recursion.
Step 8: Print the results.
Step 9: Stop the program.

PROGRAM:

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

// Function to perform binary search recursively
int binarySearchRecursive(int arr[], int low, int high, int key) {
   if (low <= high) {
      int mid = low + (high - low) / 2;

      if (arr[mid] == key)
         return mid;
      else if (arr[mid] > key)
         return binarySearchRecursive(arr, low, mid - 1, key);
      else
         return binarySearchRecursive(arr, mid + 1, high, key);
   }

   return -1; // Element not found
}

int main() {
const int MAX_SIZE = 100;
```

```
    int arr[MAX_SIZE];
    int n, key, result;

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

cout<< "Enter sorted elements of the array:\n";
    for (int i = 0; i< n; ++i) {
cout<< "Element " << (i + 1) << ": ";
cin>>arr[i];
    }

cout<< "Enter the element to search: ";
cin>> key;

    result = binarySearchRecursive(arr, 0, n - 1, key);

    if (result != -1)
cout<< "Element found at index " << result <<endl;
    else
cout<< "Element not found in the array." <<endl;

    return 0;
}
```

OUTPUT:

INFERENCE:

Binary Search is defined as a searching algorithm used in a sorted array by repeatedly dividing the search interval in half. The idea of binary search is to use the information that the array is sorted. In this program we learnt how to implement binary search using recursion.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 14	INSERTION SORT ALGORITHM
DATE:	

AIM:

To Write a C++ program to implement the Insertion Sort algorithm.

ALGORITHM:

Step 1: Start the program.
Step 2: Get the number of elements in the array.
Step 3: Get the elements in the array.
Step 4: Sort the array using insertion sort.
Step 5: Display the array after sorting.
Step 6: Stop the program.

PROGRAM:

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

void insertionSort(int arr[], int n) {
   for (int i = 1; i< n; ++i) {
      int key = arr[i];
      int j = i - 1;

      while (j >= 0 &&arr[j] > key) {
arr[j + 1] = arr[j];
         j = j - 1;
      }

arr[j + 1] = key;
   }
}

void displayArray(int arr[], int n) {
cout<< "Sorted Array: ";
   for (int i = 0; i< n; ++i) {
cout<<arr[i] << " ";
   }
cout<<endl;
}
```

```
int main() {
const int MAX_SIZE = 100;
   int arr[MAX_SIZE];
   int n;

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

cout<< "Enter the elements of the array:\n";
   for (int i = 0; i< n; ++i) {
cout<< "Element " << (i + 1) << ": ";
cin>>arr[i];
   }

insertionSort(arr, n);

displayArray(arr, n);

   return 0;
}
```

OUTPUT:

INFERENCE:
Insertion sort is a simple sorting algorithm that works similar to the way we sort playing cards in our hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part. In this program we learnt to implement the Insertion Sort algorithm.

RESULT:

The program was successfully executed and the output was got.

EX.NO: 15	SEPARATE CHAINING TECHNIQUE IN HASHING
DATE:	

AIM:

To Write a C++ program to implement the separate chaining technique in hashing

ALGORITHM:

Step 1: Start the program.
Step 2: Set Hash table size to 10.
Step 3: Insert the key in the hash table.
Step 4: Search a key in the hash table.
Step 5: Display the hash table.
Step 6: Stop the program.

PROGRAM:

```
#include <iostream>
#include <list>
#include <iterator>

using namespace std;

class HashTable {
private:
  static const int tableSize = 10;
  list<int> table[tableSize];

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

public:
  void insertKey(int key) {
    int index = hashFunction(key);
    table[index].push_back(key);
  }

  void displayTable() {
    for (int i = 0; i<tableSize; ++i) {
cout<< "Bucket " <<i<< ": ";
      copy(table[i].begin(), table[i].end(), ostream_iterator<int>(cout, " "));
cout<<endl;
    }
    }

    bool searchKey(int key) {
        int index = hashFunction(key);

        for (auto it = table[index].begin(); it != table[index].end(); ++it) {
            if (*it == key)
                return true; // Key found
        }

        return false; // Key not found
    }
};

int main() {
HashTablehashTable;
    int choice, key;

    do {
cout<< "\n1. Insert Key\n";
cout<< "2. Display Hash Table\n";
cout<< "3. Search Key\n";
cout<< "4. Exit\n";
cout<< "Enter your choice: ";
cin>> choice;

        switch (choice) {
            case 1:
cout<< "Enter key to insert: ";
cin>> key;
hashTable.insertKey(key);
                break;

            case 2:
cout<< "Hash Table:\n";
hashTable.displayTable();
                break;

            case 3:
cout<< "Enter key to search: ";
cin>> key;
                if (hashTable.searchKey(key))
cout<< "Key " << key << " found in the hash table.\n";
                else
cout<< "Key " << key << " not found in the hash table.\n";
                break;

            case 4:
```

```
cout<< "Exiting the program.\n";
        break;

      default:
cout<< "Invalid choice! Please enter a valid option.\n";
    }

  } while (choice != 4);

  return 0;
}
```

OUTPUT:

INFERENCE:

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used. In this program we learnt to implement separate chaining technique in hashing.

RESULT:

The program was successfully executed and the output was got.