INDEX

S.NO	PARTICULAR	SIGN
1	Write a program using array of c++ for the following A. Find the max and min for a given set of numbers in a single dimensional array B. Make a 3x3 matrix as 4x4 matrix in two dimensional array	
2	Write a program using array for the concept of stack.	
3	Write a program using linked list for first in first out	
4	Write a program using linked list for stack concept	
5	Postfix to infix using stack concept	
6	Evaluation of expression using stack concept	
7	Pointers	
8	Queues	
9	Sparse matrix using array	
10	Sparse matrix using linked list	
11	Double linked traverse	
12	Double linked list- prime and perfect	
13	Polynomial addition using linked list	
14	Polynomial multiplication using linked list	
15	Recursion - factorial and ackermann function	
16	Binary tree traversal	
17	Binary search tree- sorting	
18	Sorting-bubble	
19	Sorting- selection	
20	Sorting- insertion	
21	Sorting - quick	
22	Sorting - merge	

23	Threaded binary tree	
24	Graph	
25	Symmetric or non symmetric	
26	Weighted graph	
27	Path from source to destination using d/t/l/r	
28	Find the number of available path from source to destination using 0 as source,1 s destination,-1 as obstacle	
29	Find the number of paths if vector matrix is provided with vertex and edge-mazing problem	
30	Find the shortest path if matrix is provided with distance values	
31	Sequential file organisation	
32	Indexed sequential file organisation	
33	Random file organisation	
34	Linked list file organisation	
35	Cellular partition	
36	Mazing problem	

1. Write a program using array of c++

Aim

To create a C++ program that takes either an array of integers or a 3x3 matrix from the user, then finds and displays the maximum and minimum values in the array, or calculates and displays row-wise and column-wise sums in an extended 4x4 matrix for the matrix input.

Algorithm

Step 1: Start the program by defining the necessary classes and including any required libraries.

Step 2: Define a class `Array` with an integer array `a` of fixed size, and variables `i`, `min`, and `max` to assist in finding minimum and maximum values.

Step 3: Define the `input()` method in `Array` to prompt the user to enter five integers, using a loop to store each input in the array `a`.

Step 4: Define the `output()` method in `Array` to initialize `max` and `min` with the first array element, then loop through the array to find and display the maximum and minimum values.

Step 5: Define another class `Array1` that contains a 2D integer array `a[4][4]` to store a 3x3 matrix and room for row, column, and diagonal sums.

Step 6: Define the `input()` method in `Array1` to prompt the user for a 3x3 matrix, using nested loops to store the inputs in the first three rows and columns of `a`.

Step 7: Define the `output()` method in `Array1` to calculate row sums, column sums, and diagonal sums, storing them in the fourth row, fourth column, and bottom-right corner of the matrix.

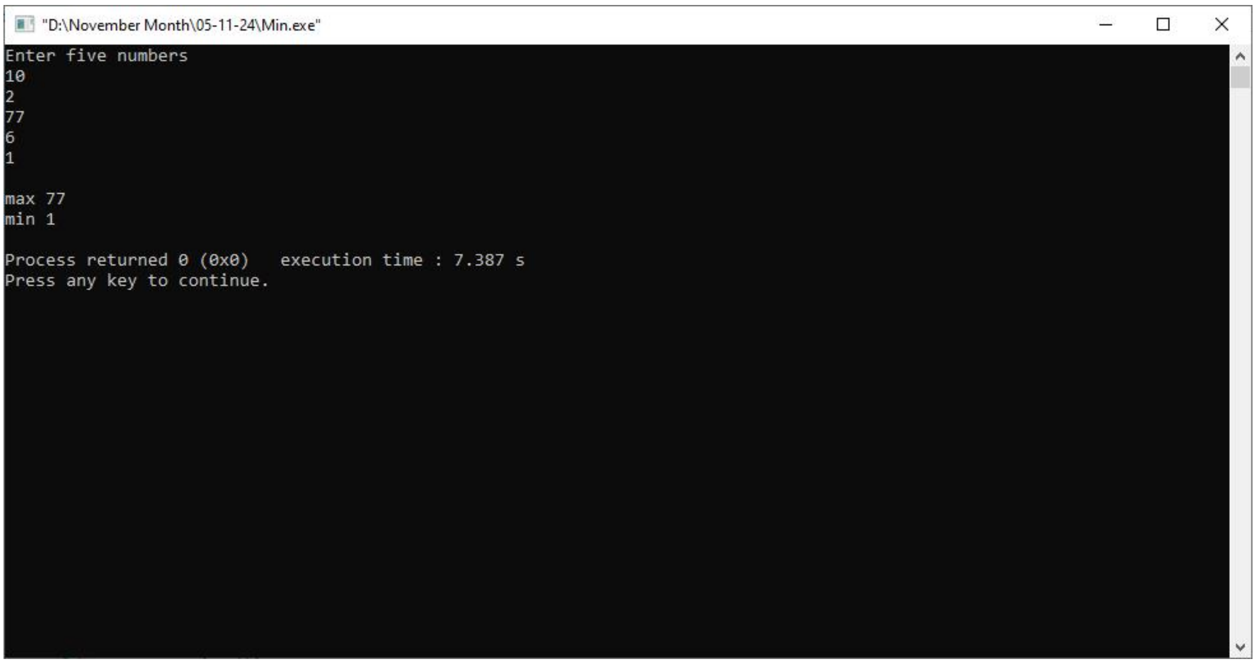
Step 8: In `main()`, create objects of `Array` and `Array1`, calling their `input()` and `output()` methods to gather inputs and display results based on whether an array or matrix is used.

Step 9: Stop the program

A) Find the max and min for a given set of numbers in a single dimensional array:

```
#include <iostream>
using namespace std;
class Array {
public:
   int a[5];
   int i, min, max;
   void input() {
      cout << "Enter five numbers" << endl;
      for (i = 0; i < 5; i++) {
         cin >> a[i];
      }
      cout << endl;
   }
   void output() {
      max = a[0];
      min = a[0];
      for (i = 0; i < 5; i++) {
         if (a[i] > max)
            max = a[i];
         if (a[i] <= min)
            min = a[i];
      }
      cout << "max " << max << endl;
      cout << "min " << min << endl;
   }
};
int main()
{
   Array arr;
   arr.input();
   arr.output();
   return 0;
}
```

Output:



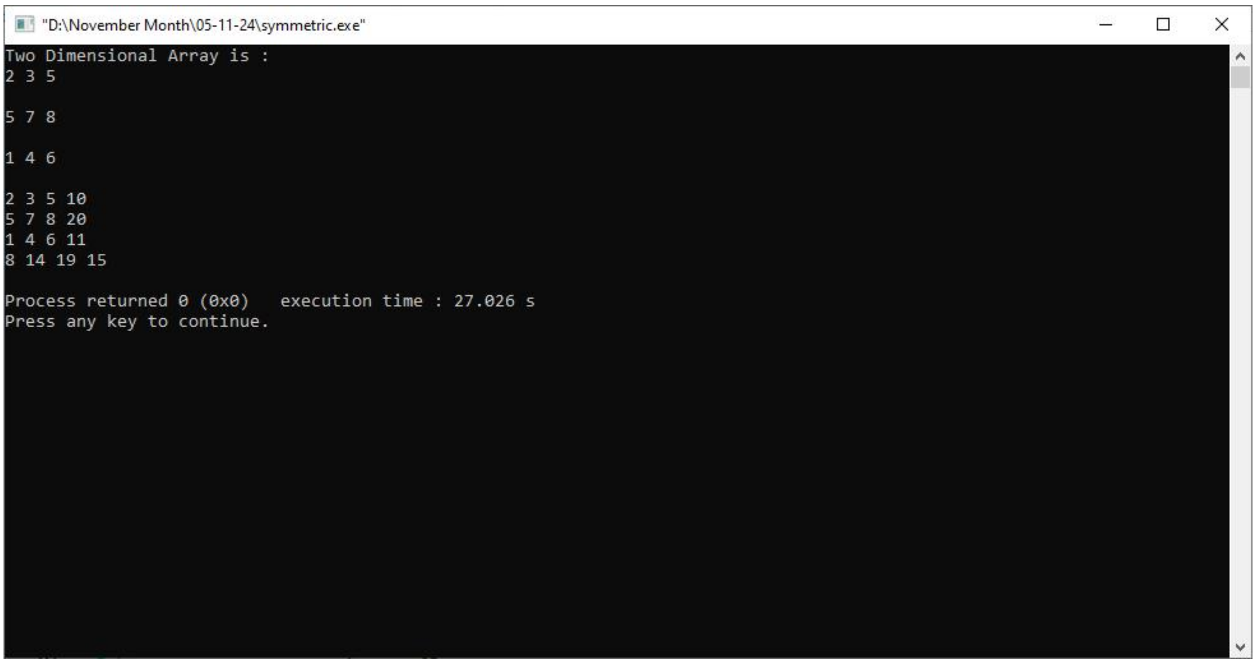
```
Enter five numbers
10
2
77
6
1

max 77
min 1

Process returned 0 (0x0)   execution time : 7.387 s
Press any key to continue.
```

B.Make a 3x3 matrix as 4x4 matrix in two dimensional array:

```
#include<iostream>
using namespace std;
class array1{
public:
   int a[10][10];

   void output(){
      for(int i=0;i<=2;i++){
         a[i][3]=0;
         for(int j=0;j<=2;j++){
            a[i][3]=a[i][3]+a[i][j];
         }
      }

      for(int i=0;i<=2;i++){
         a[3][i]=0;
         for(int j=0;j<=2;j++){
            a[3][i]=a[3][i]+a[j][i];
         }
      }

      a[3][3]=0;
      for(int i=0;i<=2;i++){
         for(int j=0;j<=2;j++){
            if(i==j){
                a[3][3]=a[3][3]+a[i][j];
            }
         }
      }

      for(int i=0;i<=3;i++){
         for(int j=0;j<=3;j++){
            cout<<a[i][j]<<" ";
         }
```

```
            cout<<endl;
        }
    }
    void input(){
        cout<<"Two Dimensional Array is : "<<endl;
        for(int i=0;i<=2;i++){
            for(int j=0;j<=2;j++){
                cin>>a[i][j];
            }
            cout<<endl;
        }
    }
};
int main()
{
    array1 arr;
    arr.input();
    arr.output();
    return 0;
}
```

Output:



```
Two Dimensional Array is :
2 3 5

5 7 8

1 4 6

2 3 5 10
5 7 8 20
1 4 6 11
8 14 19 15

Process returned 0 (0x0)   execution time : 27.026 s
Press any key to continue.
```

2. <u>Write a program using array for the concept of stack.</u>

Aim:

To write a C++ program to implement array for the concept of stack.

Algorithm:

Step 1: Start the process.

Step 2: Initialize the stack with a maximum size and set the \`top\` pointer to -1.

Step 3: Define the \`push\` operation to add an element if the stack is not full; otherwise, display "Stack Overflow".

Step 4: Define the \`pop\` operation to remove and return the top element if the stack is not empty; otherwise, display "Stack Underflow".

Step 5: Define the \`peek\` operation to view the top element if the stack is not empty; otherwise, display "Stack is Empty".

Step 6: Define the \`isEmpty\` operation to check if the stack is empty.

Step 7: In the main function, initialize a \`Stack\` object and use the \`push\` operation to add elements to the stack.

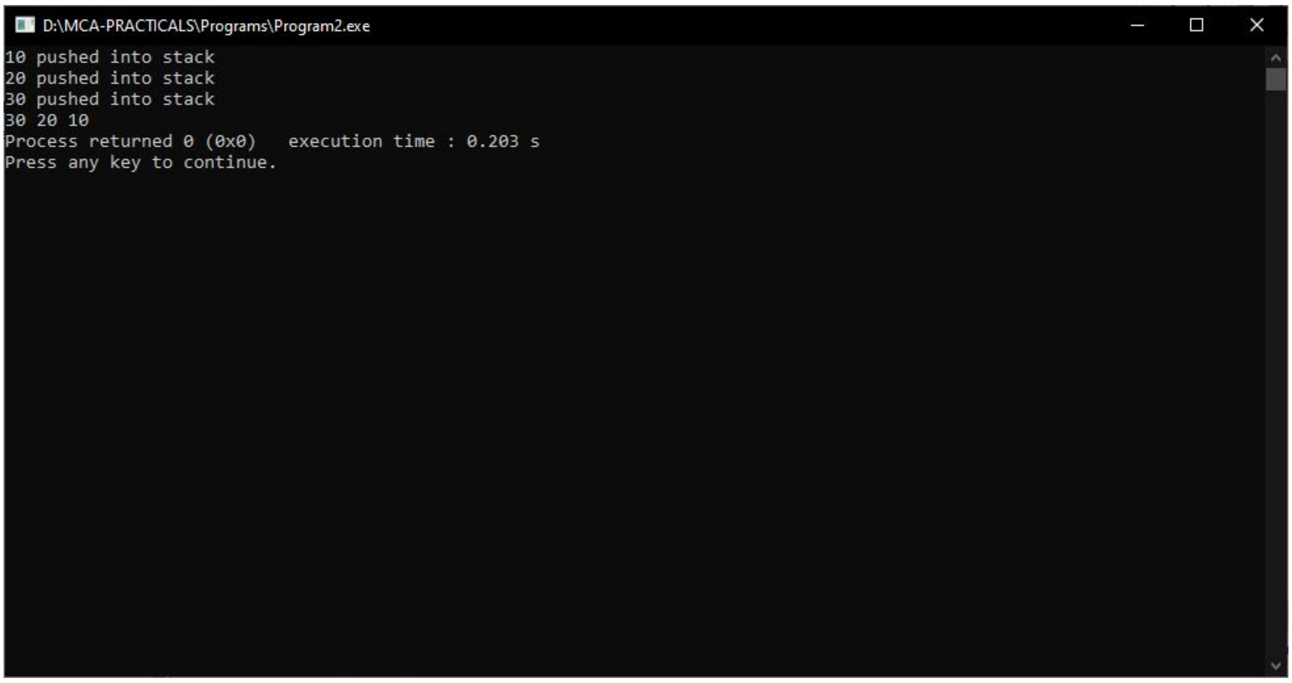
Step 8: Use a loop to display each element and remove it using the \`pop\` operation until the stack is empty.

Step 9: Stop the process.

Program:

```
#include <bits/stdc++.h>
#include<iostream>
using namespace std;
#define MAX 1000

class Stack {
   int top;
public:
   int a[MAX];
   Stack() { top = -1; }
   bool push(int x);
   int pop();
   int peek();
   bool isEmpty();
};

bool Stack::push(int x) {
   if (top >= (MAX - 1)) {
      cout << "Stack Overflow";
      return false;
   } else {
      a[++top] = x;
      cout << x << " pushed into stack\n";
      return true;
   }
}

int Stack::pop() {
   if (top < 0) {
      cout << "Stack Underflow";
      return 0;
   } else {
      return a[top--];
   }
}

int Stack::peek() {
    if (top < 0) {
        cout << "Stack is Empty";
        return 0;
    } else {
        return a[top];
    }
}

bool Stack::isEmpty() {
    return (top < 0);
}


int main() {
    Stack s;
    s.push(10);
    s.push(20);
    s.push(30);
    while(!s.isEmpty()) {
        cout << s.peek() << " ";
        s.pop();
    }
    return 0;
}
```

Output:



```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 20 10
Process returned 0 (0x0)   execution time : 0.203 s
Press any key to continue.
```

3. Write a program using linked list for first in first out

Aim:

To write a C++ program that implements linked list for first in first out.

Algorithm:

Step 1: Start the process.

Step 2: Define a Node structure with `data` to store the value and `next` as a pointer to the next node.

Step 3: Define a Queue class with `front` and `rear` pointers to the front and rear nodes.

Step 4: Initialize the Queue by setting `front` and `rear` to `NULL` to indicate it's empty.

Step 5: Define the enqueue operation to create a new node, and if the queue is empty, set both `front` and `rear` to this new node; otherwise, add it to the end and update `rear`.

Step 6: Define the dequeue operation by checking if the queue is not empty, removing the front node, and updating `front`; if empty after dequeue, set `rear` to `NULL`, otherwise display "Queue Underflow".

Step 7: Define the peek operation to return the front node's value if not empty; otherwise, display "Queue is Empty".

Step 8: Define the isEmpty operation to check if `front` is `NULL`.

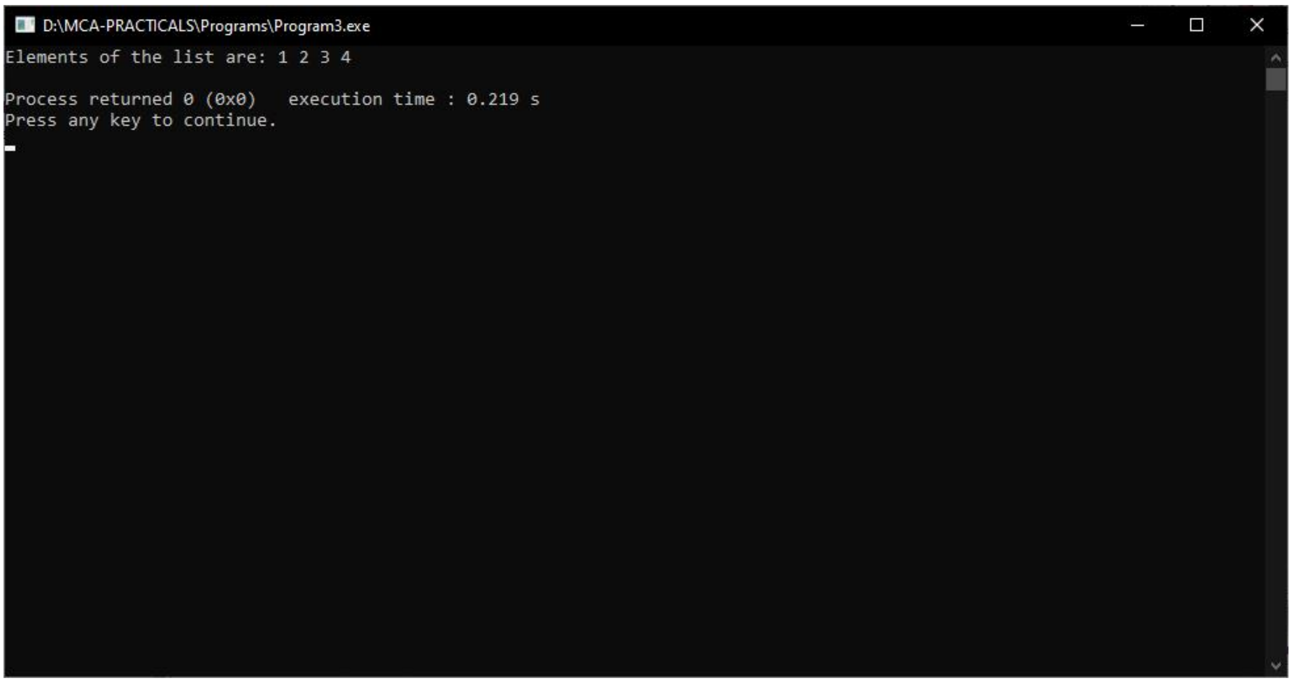
Step 9: In the main function, create a Queue object, enqueue elements, display each element with peek, and dequeue until empty.

Step 10: Stop the process.

Program:

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

class Node {
  public:
    int data;
    Node *next;
    Node(int data) {
        this->data = data;
        this->next = NULL;
    }
};

class Linkedlist {
    Node *head;
  public:
    Linkedlist() {
        head = NULL;
    }
    void insertAtHead(int data) {
        Node *newNode = new Node(data);
        newNode->next = head;
        head = newNode;
    }
    void print() {
        Node *temp = head;
        while (temp != NULL) {
            cout << temp->data << " ";
            temp = temp->next;
        }
    }
};

int main() {
    Linkedlist list;
    list.insertAtHead(4);
    list.insertAtHead(3);
    list.insertAtHead(2);
    list.insertAtHead(1);
    cout << "Elements of the list are: ";
    list.print();
    cout << endl;
    return 0;
}
```

Output:



```
Elements of the list are: 1 2 3 4

Process returned 0 (0x0)   execution time : 0.219 s
Press any key to continue.
```

4. Write a program using linked list for stack concept

Aim:

To write a C++ program that implements basic stack operations using a linked list.

Algorithm:

Step 1: Start the process.

Step 2: Define a StackNode structure to represent each element in the stack with members: `data` to store the value and `next` as a pointer to the next node.

Step 3: Create a function `newNode` to allocate a new StackNode and initialize it with the given data.

Step 4: Define the `isEmpty` function to check if the stack is empty by verifying if the root pointer is `NULL`.

Step 5: Define the `push` function to create a new node; link it to the current top of the stack, and update the root pointer to point to this new node.

Step 6: Define the `pop` function to check if the stack is not empty; if not, remove the top node, update the root pointer to the next node, and free the memory of the popped node.

Step 7: Define the `peek` function to return the value of the top node if the stack is not empty; otherwise, return an error value.

Step 8: In the main function, create a StackNode pointer as the root and use the `push` function to add elements to the stack.

Step 9: Display the result of the `pop` operation and the top element using the `peek` function.

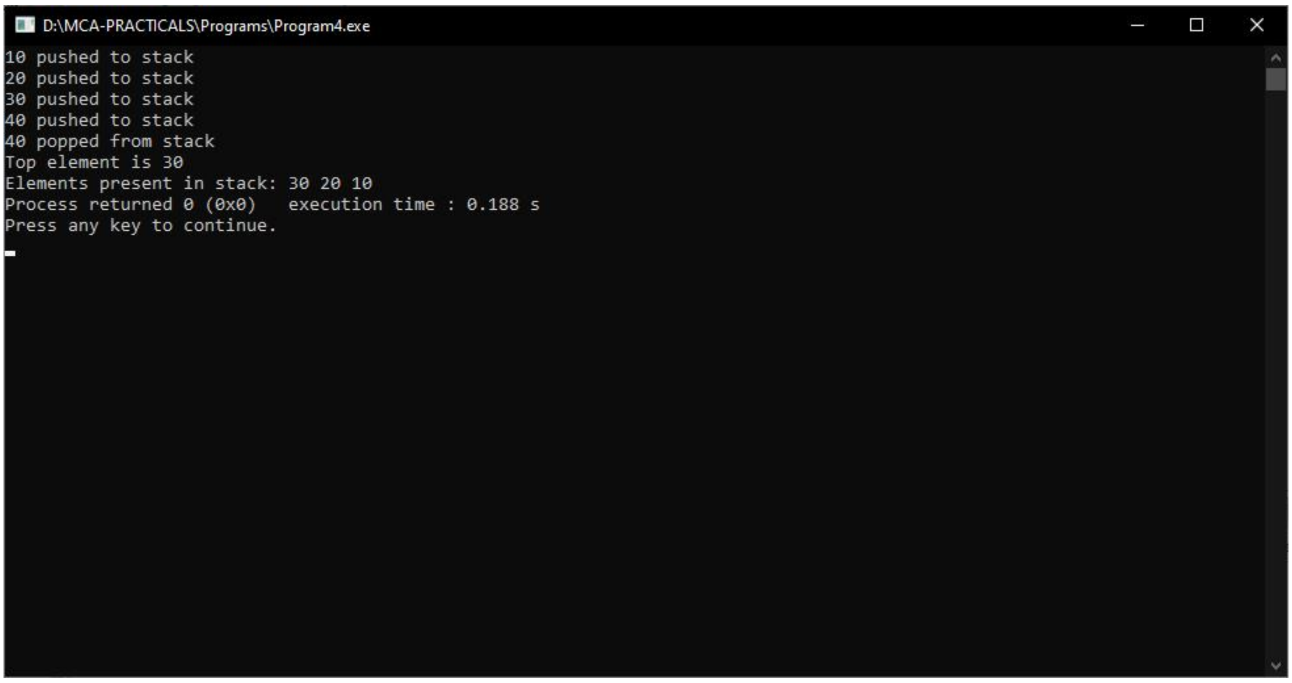
Step 10: Use a loop to display and remove all elements in the stack using the `peek` and `pop` functions until the stack is empty.

Step 11: Stop the process.

Program:

```
#include <bits/stdc++.h>
using namespace std;

class StackNode {
public:
    int data;
    StackNode* next;
};

StackNode* newNode(int data) {
    StackNode* stackNode = new StackNode();
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}

int isEmpty(StackNode* root) {
    return !root;
}

void push(StackNode** root, int data) {
    StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    cout << data << " pushed to stack\n";
}

int pop(StackNode** root) {
    if (isEmpty(*root))
        return INT_MIN;
    StackNode* temp = *root;
    int popped = (*root)->data;
    *root = (*root)->next;
    free(temp);
    return popped;
}

int peek(StackNode* root) {
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}

int main() {
    StackNode* root = NULL;

    push(&root, 10);
    push(&root, 20);
    push(&root, 30);
    push(&root, 40);

    cout << pop(&root) << " popped from stack\n";
    cout << "Top element is " << peek(root) << endl;

    cout << "Elements present in stack: ";
    while(!isEmpty(root)) {
        cout << peek(root) << " ";
        pop(&root);
    }

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program4.exe

```
10 pushed to stack
20 pushed to stack
30 pushed to stack
40 pushed to stack
40 popped from stack
Top element is 30
Elements present in stack: 30 20 10
Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

5. Postfix to infix using stack concept

Aim:

To write a C++ program that converts a postfix expression to an infix expression using stack concepts.

Algorithm:

Step 1: Start the process.

Step 2: Define a function `prec` to return the precedence of operators, where higher values indicate higher precedence.

Step 3: Define a function `associativity` to determine the associativity of operators, indicating whether they are left or right associative.

Step 4: In the `postfixToInfix` function, initialize an empty stack to hold operands and an empty string to build the infix expression.

Step 5: Iterate through each character in the postfix expression string, if the character is an operand (either a letter or a digit), push it onto the stack; if the character is an operator, pop the top two operands from the stack, combine them with the operator in infix format (adding parentheses as needed), and push the resulting string back onto the stack.

Step 6: After processing all characters, the top of the stack will contain the final infix expression.

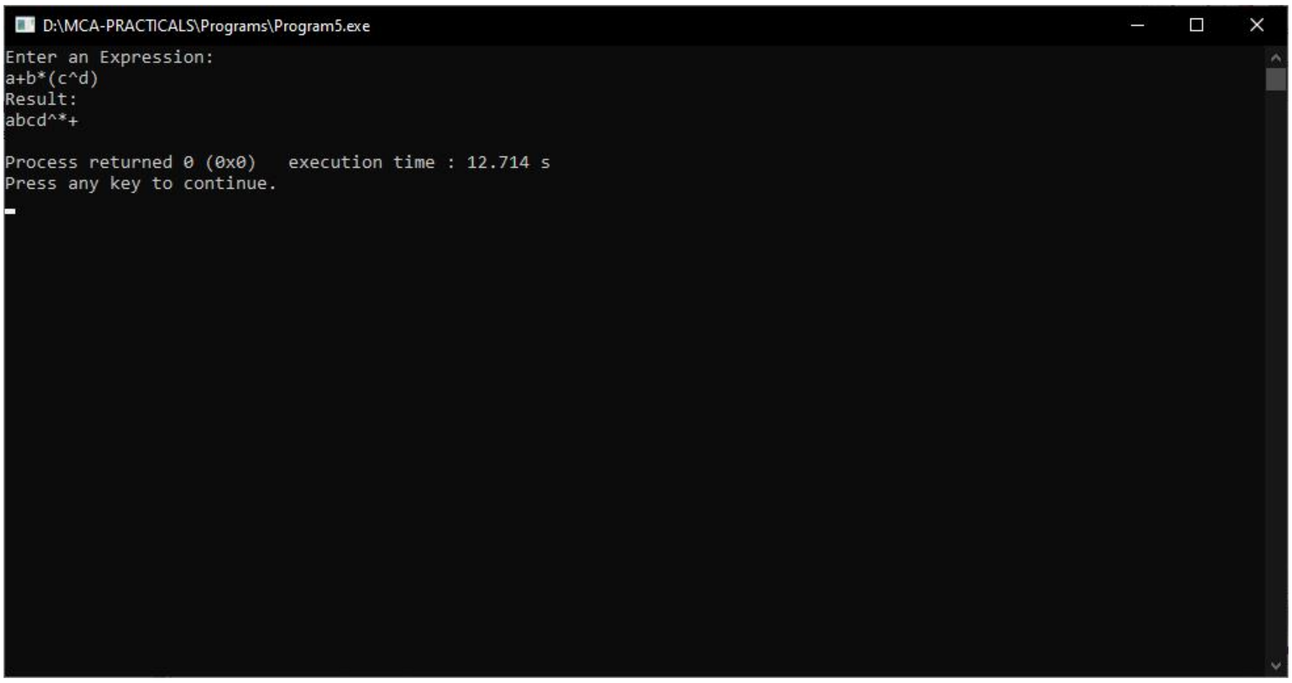
Step 7: Output the final infix expression.

Step 8: Stop the process.

Program:

```
#include <bits/stdc++.h>
using namespace std;

int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1;
}

char associativity(char c) {
    return (c == '^') ? 'R' : 'L';
}

void infixToPostfix(string s) {
    stack<char> st;
    string result;

    for (int i = 0; i < s.length(); i++) {
        char c = s[i];

        if (isalnum(c))
            result += c;
        else if (c == '(')
            st.push(c);
        else if (c == ')') {
            while (st.top() != '(') {
                result += st.top();
                st.pop();
            }
            st.pop();
        }
        else {
            while (!st.empty() && (prec(c) < prec(st.top()) ||
                    (prec(c) == prec(st.top()) && associativity(c) == 'L'))) {
                result += st.top();
                st.pop();
            }
            st.push(c);
        }
    }

    while (!st.empty()) {
        result += st.top();
        st.pop();
    }

    cout << result << endl;
}

int main() {
    string exp = "a+b*(c^d)";
    infixToPostfix(exp);
    return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program5.exe

```
Enter an Expression:
a+b*(c^d)
Result:
abcd^*+

Process returned 0 (0x0)   execution time : 12.714 s
Press any key to continue.
```

6. Evaluation of expression using stack concept

Aim:

To write a C++ program that evaluates an arithmetic expression with operators and parentheses using stacks for operators and values.

Algorithm:

Step 1: Start the process.

Step 2: Define a function `precedence` to return the precedence of operators, where `*` and `/` have higher precedence than `+` and `-`.

Step 3: Define a function `applyOp` to perform arithmetic operations based on the given operator and operands.

Step 4: Define the `evaluate` function, which initializes two stacks: one for values and one for operators.

Step 5: Traverse each character in the expression, pushing numbers and operators to their stacks and evaluating subexpressions as needed.

Step 6: After processing all characters in the expression, apply any remaining operators in the operators stack to the values in the values stack.

Step 7: The final value in the values stack is the result of the expression.

Step 8: Output the final result.

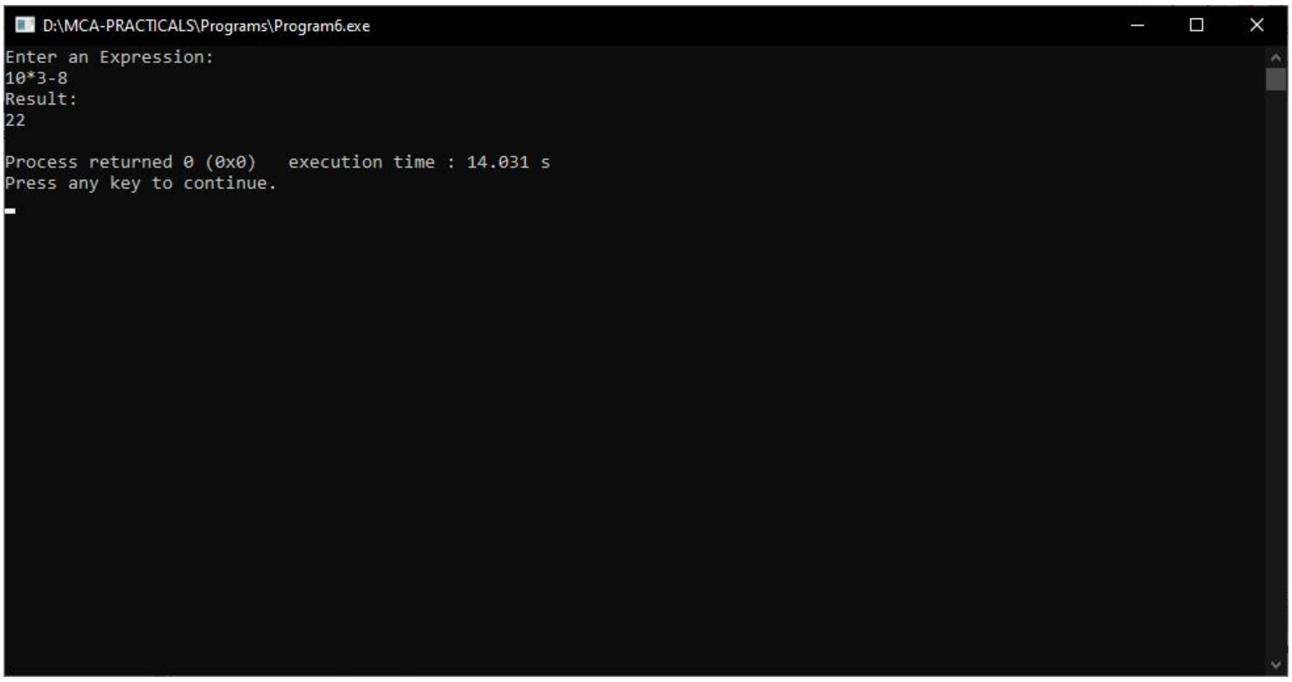
Step 9: Stop the process.

Program:

```
#include <bits/stdc++.h>
using namespace std;

int precedence(char op) {
   if(op == '+' || op == '-') return 1;
   if(op == '*' || op == '/') return 2;
   return 0;
}

int applyOp(int a, int b, char op) {
   switch(op) {
      case '+': return a + b;
      case '-': return a - b;
      case '*': return a * b;
      case '/': return a / b;
   }
   return 0;
}
int evaluate(string tokens) {
   stack<int> values;
   stack<char> ops;

   for(int i = 0; i < tokens.length(); i++) {
      if(tokens[i] == ' ')
         continue;
      else if(tokens[i] == '(') {
         ops.push(tokens[i]);
      }
      else if(isdigit(tokens[i])) {
         int val = 0;
         while(i < tokens.length() && isdigit(tokens[i])) {
            val = (val * 10) + (tokens[i] - '0');
            i++;
         }
```

```
            values.push(val);
            i--;
        }
        else if(tokens[i] == ')') {
            while(!ops.empty() && ops.top() != '(') {
                int val2 = values.top(); values.pop();
                int val1 = values.top(); values.pop();
                char op = ops.top(); ops.pop();
                values.push(applyOp(val1, val2, op));
            }
            if(!ops.empty()) ops.pop();
        }
        else {
            while(!ops.empty() && precedence(ops.top()) >= precedence(tokens[i])) {
                int val2 = values.top(); values.pop();
                int val1 = values.top(); values.pop();
                char op = ops.top(); ops.pop();
                values.push(applyOp(val1, val2, op));
            }
            ops.push(tokens[i]);
        }
    }
    while(!ops.empty()) {
        int val2 = values.top(); values.pop();
        int val1 = values.top(); values.pop();
        char op = ops.top(); ops.pop();
        values.push(applyOp(val1, val2, op));
    }

    return values.top();
}
int main() {
    cout << evaluate("10 + 2 * 6") << "\n";
    return 0;
}
```

Output:


```
D:\MCA-PRACTICALS\Programs\Program6.exe
Enter an Expression:
10*3-8
Result:
22

Process returned 0 (0x0)   execution time : 14.031 s
Press any key to continue.
```

7. Pointers

Aim:

To demonstrate various pointer concepts in C++ using a class with multiple methods.

Algorithm:

Step 1: Start the process.

Step 2: Define a class `pointerexample` to include methods for different pointer operations.

Step 3: Implement `pointermeaning` to show basic pointer usage, including accessing a variable's address and value.

Step 4: Implement `nullpointer` to initialize a null pointer and print its value and address.

Step 5: Implement `incrementingpointer` and `decrementingpointer` to show pointer arithmetic through an array by incrementing and decrementing a pointer.

Step 6: Implement `pointertopointer`, `passingpointertofunctions`, and `returnpointertofunction` to show multi-level pointers, passing pointers to functions, and returning pointers.

Step 7: In `main`, create an object of `pointerexample` and call each method

Step 8: Stop the process

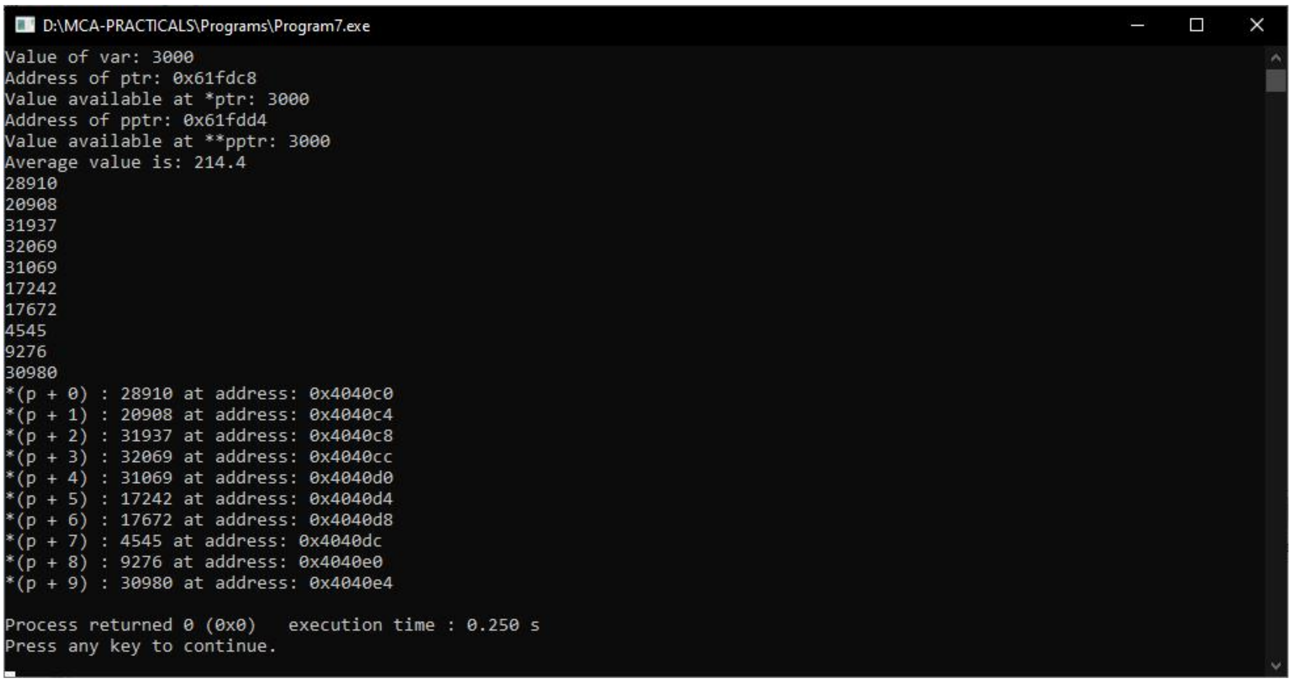
Program:

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

class PointerExample {
public:
   void pointerMeaning() {
      int var = 20;
      int *ip = &var;

      cout << "Value of var variable: " << var << endl;
      cout << "Address stored in ip variable: " << ip << endl;
      cout << "Value of *ip variable: " << *ip << endl;
   }

   void nullPointer() {
      int *ptr = NULL;
      cout << "The value of ptr is " << ptr << endl;
      cout << "The address of ptr is " << &ptr << endl;
   }

   void incrementingPointer() {
      const int MAX = 3;
      int var[MAX] = {10, 100, 200};
      int *ptr = var;

      for (int i = 0; i < MAX; i++) {
         cout << "Address of var[" << i << "] = " << ptr << endl;
         cout << "Value of var[" << i << "] = " << *ptr << endl;
         ptr++;
         cout << "Jump to address " << &(*ptr) << endl;
      }
      cout << "Address of var[1] = " << &var[1] << endl;
   }
```

```
void decrementingPointer() {
    const int MAX = 3;
    int var[MAX] = {10, 100, 200};
    int *ptr = &var[MAX - 1];

    for (int i = MAX - 1; i >= 0; i--) {
        cout << "Address of var[" << i << "] = " << ptr << endl;
        cout << "Value of var[" << i << "] = " << *ptr << endl;
        ptr--;
    }
}

void pointerToPointer() {
    int var = 3000;
    int *ptr = &var;
    int **pptr = &ptr;

    cout << "Value of var: " << var << endl;
    cout << "Address of ptr: " << &ptr << endl;
    cout << "Value available at *ptr: " << *ptr << endl;
    cout << "Address of pptr: " << &(*ptr) << endl;
    cout << "Value available at **pptr: " << **pptr << endl;
}

double getAverage(int *arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; ++i) {
        sum += arr[i];
    }
    return double(sum) / size;
}

void passingPointerToFunctions() {
    int balance[5] = {1000, 2, 3, 17, 50};
```

```
        double avg = getAverage(balance, 5);
        cout << "Average value is: " << avg << endl;
    }

    int* getRandom() {
        static int r[10];
        srand((unsigned)time(NULL));

        for (int i = 0; i < 10; ++i) {
            r[i] = rand();
            cout << r[i] << endl;
        }
        return r;
    }

    void returnPointerToFunction() {
        int *p = getRandom();
        for (int i = 0; i < 10; i++) {
            cout << "*(p + " << i << ") : " << *(p + i) << " at address: " << (p + i) << endl;
        }
    }
};

int main() {
    PointerExample p1;
    p1.pointerMeaning();
    p1.nullPointer();
    p1.incrementingPointer();
    p1.decrementingPointer();
    p1.pointerToPointer();
    p1.passingPointerToFunctions();
    p1.returnPointerToFunction();
    return 0;
}
```

Output:



```
Value of var: 3000
Address of ptr: 0x61fdc8
Value available at *ptr: 3000
Address of pptr: 0x61fdd4
Value available at **pptr: 3000
Average value is: 214.4
28910
20908
31937
32069
31069
17242
17672
4545
9276
30980
*(p + 0) : 28910 at address: 0x4040c0
*(p + 1) : 20908 at address: 0x4040c4
*(p + 2) : 31937 at address: 0x4040c8
*(p + 3) : 32069 at address: 0x4040cc
*(p + 4) : 31069 at address: 0x4040d0
*(p + 5) : 17242 at address: 0x4040d4
*(p + 6) : 17672 at address: 0x4040d8
*(p + 7) : 4545 at address: 0x4040dc
*(p + 8) : 9276 at address: 0x4040e0
*(p + 9) : 30980 at address: 0x4040e4

Process returned 0 (0x0)   execution time : 0.250 s
Press any key to continue.
```

8. Queues

Aim:

To implement a basic queue data structure using an array with enqueue, dequeue, and front/rear element retrieval functions.

Algorithm:

Step 1: Start the process.

Step 2: Define a `Queue` class with private variables `front`, `rear`, `size`, and a dynamic array `queue` to store elements.

Step 3: In the constructor, initialize `front` and `rear` to -1, set the `size` to the given value, and allocate memory for the `queue` array.

Step 4: Define the `isFull` function to check if the queue has reached its maximum capacity, and the `isEmpty` function to check if the queue has no elements.

Step 5: Implement `enqueue` to add elements to the queue by incrementing `rear`, handling full queue cases, and initializing `front` to 0 if the queue was previously empty.

Step 6: Implement `dequeue` to remove the front element by incrementing `front`, handle empty queue cases, and reset `front` and `rear` when the queue is empty.

Step 7: Define `getFront` and `getRear` to return the first and last elements of the queue, respectively, handling empty queue cases.

Step 8: In `main`, create a `Queue` object, perform enqueue operations, display front and rear elements, and carry out multiple dequeue operations.

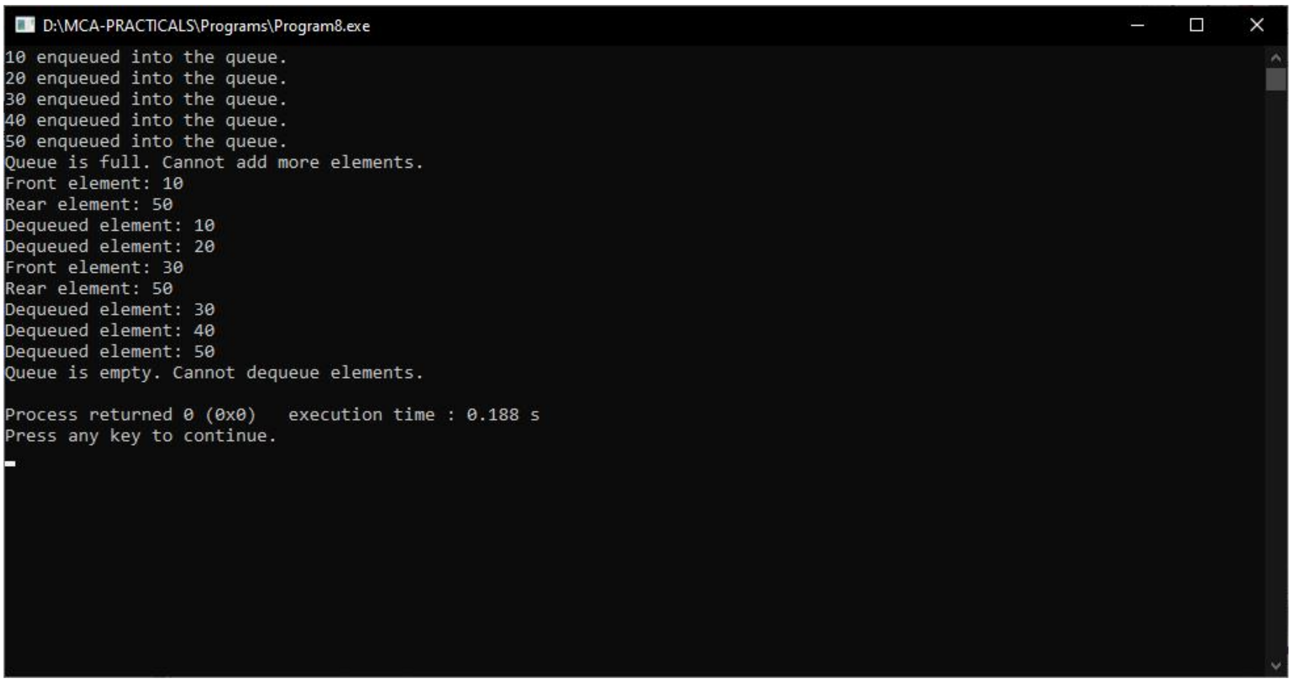
Step 9: Stop the process.

Program:

```
#include <iostream>
using namespace std;
class Queue {
private:
   int front, rear, size;
   int* queue;
public:
   Queue(int s) {
      front = -1;
      rear = -1;
      size = s;
      queue = new int[size];
   }
   ~Queue() {
      delete[] queue;
   }
   bool isFull() {
      return (rear == size - 1);
   }
   bool isEmpty() {
      return (front == -1 || front > rear);
   }
   void enqueue(int value) {
      if (isFull()) {
         cout << "Queue is full. Cannot add more elements.\n";
         return;
      }
      if (isEmpty()) {
         front = 0;
      }
      rear++;
      queue[rear] = value;
      cout << value << " enqueued into the queue.\n";
   }
```

```
    void dequeue() {
        if (isEmpty()) {
            cout << "Queue is empty. Cannot dequeue elements.\n";
            return;
        }
        cout << "Dequeued element: " << queue[front] << endl;
        front++;
        if (front > rear) {
            front = rear = -1;
        }
    }
    int getFront() {
        if (!isEmpty()) {
            return queue[front];
        } else {
            cout << "Queue is empty.\n";
            return -1;
        }
    }
    int getRear() {
        if (!isEmpty()) {
            return queue[rear];
        } else {
            cout << "Queue is empty.\n";
            return -1;
        }
    }
};
int main() {
    Queue q(5);
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.enqueue(40);
    q.enqueue(50);
    q.enqueue(60);
    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;
    q.dequeue();
    q.dequeue();
    cout << "Front element: " << q.getFront() << endl;
    cout << "Rear element: " << q.getRear() << endl;
    q.dequeue();
    q.dequeue();
    q.dequeue();
    q.dequeue();
    return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program8.exe

```
10 enqueued into the queue.
20 enqueued into the queue.
30 enqueued into the queue.
40 enqueued into the queue.
50 enqueued into the queue.
Queue is full. Cannot add more elements.
Front element: 10
Rear element: 50
Dequeued element: 10
Dequeued element: 20
Front element: 30
Rear element: 50
Dequeued element: 30
Dequeued element: 40
Dequeued element: 50
Queue is empty. Cannot dequeue elements.

Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

9. Sparse matrix using array

Aim

To represent a sparse matrix using a compact format by storing only non-zero elements and their positions in an array.

Algorithm

Step 1: Start the process.

Step 2: Define the `sparsematrix` class with a 4x5 2D array `sparseMatrix` to represent the original sparse matrix and an integer variable `size` to count non-zero elements.

Step 3: Initialize `sparseMatrix` with given values, and in the `sparsematrixformat` method, iterate over each element to calculate the count of non-zero elements, updating `size`.

Step 4: Define a 3-row `compactMatrix` array, where the number of columns is `size`, to store the row index, column index, and value of each non-zero element.

Step 5: Iterate through `sparseMatrix`, and for each non-zero element, store its row index, column index, and value in `compactMatrix`, incrementing the column counter `k`.

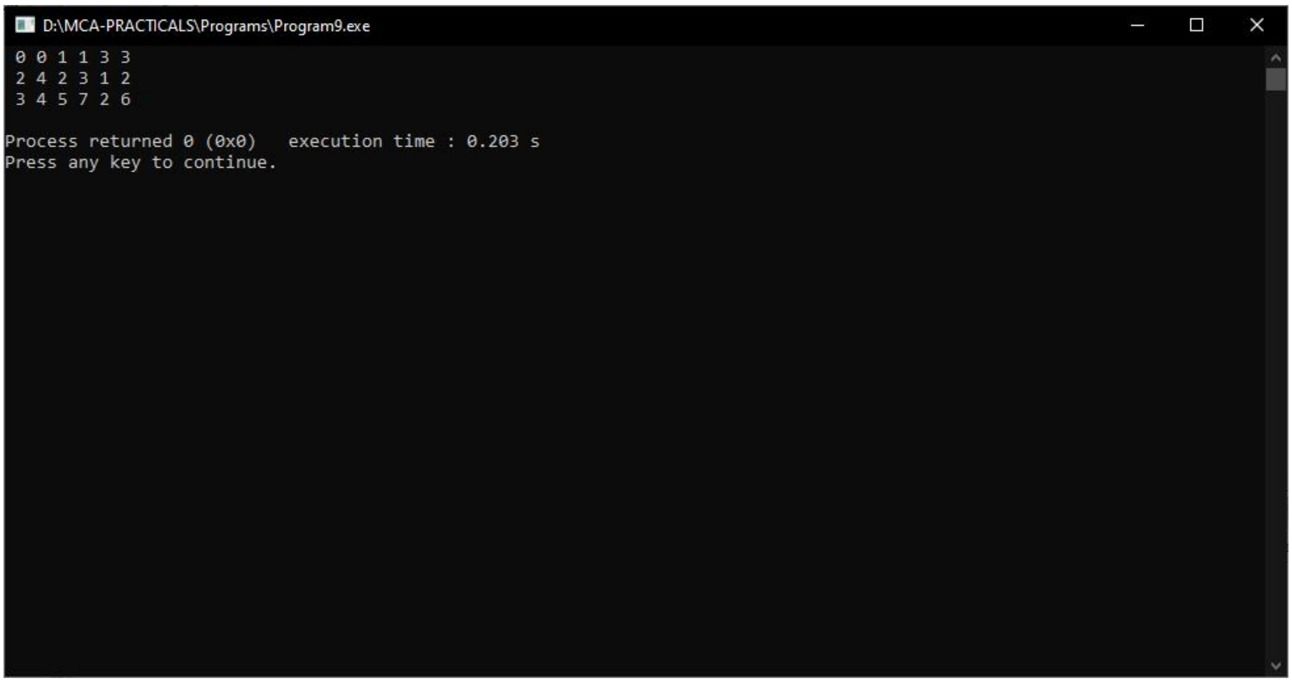
Step 6: Print the `compactMatrix` by iterating over its rows and columns.

Step 7: Stop the process.

Program:

```
#include <iostream>
using namespace std;
class sparsematrix {
public:
        int sparseMatrix[4][5] = {
                {0, 0, 3, 0, 4},
                {0, 0, 5, 7, 0},
                {0, 0, 0, 0, 0},
                {0, 2, 6, 0, 0}
        };
        int size = 0;
        void sparsematrixformat() {
                for (int i = 0; i < 4; i++)
                        for (int j = 0; j < 5; j++)
                                if (sparseMatrix[i][j] != 0)
                                        size++;
                int compactMatrix[3][size];
                int k = 0;
                for (int i = 0; i < 4; i++)
                        for (int j = 0; j < 5; j++)
                                if (sparseMatrix[i][j] != 0) {
                                        compactMatrix[0][k] = i;
                                        compactMatrix[1][k] = j;
                                        compactMatrix[2][k] = sparseMatrix[i][j];
                                        k++;
                                }
                for (int i = 0; i < 3; i++) {
                        for (int j = 0; j < size; j++)
                                cout << " " << compactMatrix[i][j];
                        cout << "\n";
                }
        }
};
int main() {
        sparsematrix s1;
        s1.sparsematrixformat();
        return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program9.exe

```
 0 0 1 1 3 3
 2 4 2 3 1 2
 3 4 5 7 2 6

Process returned 0 (0x0)   execution time : 0.203 s
Press any key to continue.
```

10. Sparse matrix using linked list

Aim:

To represent a sparse matrix using a linked list, storing only the non-zero elements along with their row and column indices.

Algorithm:

Step 1: Start the process.

Step 2: Define a `Node` class with integer members for `row`, `col`, `data`, and a pointer `next` to the next node.

Step 3: Create the function `create_new_node` to allocate memory for a new node, initialize it with the given row index, column index, and data, and append it to the linked list.

Step 4: In the `printList` function, iterate through the linked list starting from the head, printing the `row`, `col`, and `data` of each node.

Step 5: In the `main` function, define a 4x5 array to represent the sparse matrix, initializing it with specified values.

Step 6: Loop through the array and call `create_new_node` for each non-zero element to build the linked list.

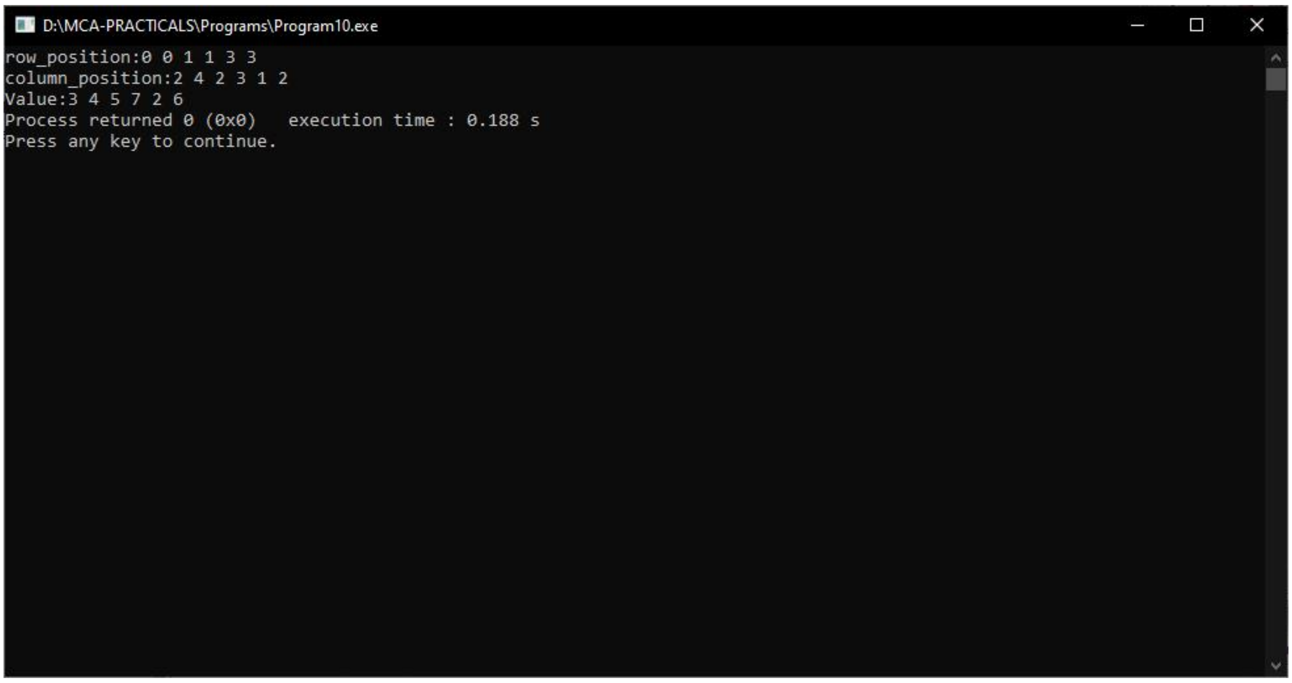
Step 7: Call `printList` to display the contents of the linked list representing the sparse matrix.

Step 8: Stop the process.

Program:

```
#include<iostream>
using namespace std;
class Node {
public:
        int row;
        int col;
        int data;
        Node *next;
};
void create_new_node(Node **p, int row_index, int col_index, int x) {
        Node *temp = *p;
        Node *r;
        if (temp == NULL) {
                temp = new Node();
                temp->row = row_index;
                temp->col = col_index;
                temp->data = x;
                temp->next = NULL;
                *p = temp;
        } else {
                while (temp->next != NULL)
                        temp = temp->next;
                r = new Node();
                r->row = row_index;
                r->col = col_index;
                r->data = x;
                r->next = NULL;
                temp->next = r;
        }
}

void printList(Node *start) {
        Node *ptr = start;
        cout << "row_position:";
        while (ptr != NULL) {
                cout << ptr->row << " ";
                ptr = ptr->next;
        }
        cout << endl << "column_position:";
        ptr = start;
        while (ptr != NULL) {
                cout << ptr->col << " ";
                ptr = ptr->next;
        }
        cout << endl << "Value:";
        ptr = start;
        while (ptr != NULL) {
                cout << ptr->data << " ";
                ptr = ptr->next;
        }
}
int main() {
        int sparseMatrix[4][5] = {
                {0, 0, 3, 0, 4},
                {0, 0, 5, 7, 0},
                {0, 0, 0, 0, 0},
                {0, 2, 6, 0, 0}
        };
        Node *first = NULL;
        for(int i = 0; i < 4; i++) {
                for(int j = 0; j < 5; j++) {
                        if (sparseMatrix[i][j] != 0)
                                create_new_node(&first, i, j, sparseMatrix[i][j]);
                }
        }
        printList(first);
        return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program10.exe

```
row_position:0 0 1 1 3 3
column_position:2 4 2 3 1 2
Value:3 4 5 7 2 6
Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

11. Double linked traverse

Aim:

To create and manage a doubly linked list in C++, allowing for insertion of nodes and traversal in both forward and backward directions.

Algorithm:

Step 1: Start the process.

Step 2: Define a `LinkedList` class with integer member `data` and pointers `front` and `back` to represent the next and previous nodes.

Step 3: Initialize the linked list by creating the head node with a predefined value.

Step 4: In a loop (for five iterations), prompt the user to input data for new nodes, creating a new node each time, and link it to the existing list.

Step 5: Update pointers to maintain the links: set the `front` of the current node to the newly created node and the `back` of the new node to the current node.

Step 6: After inserting all nodes, set up two pointers: one (`t2`) to traverse from the head to the end (front to back) and another (`t3`) to traverse from the last node back to the head (back to front).

Step 7: Print the values of the nodes from front to back using the `front` pointer.

Step 8: Print the values of the nodes from back to front using the `back` pointer.

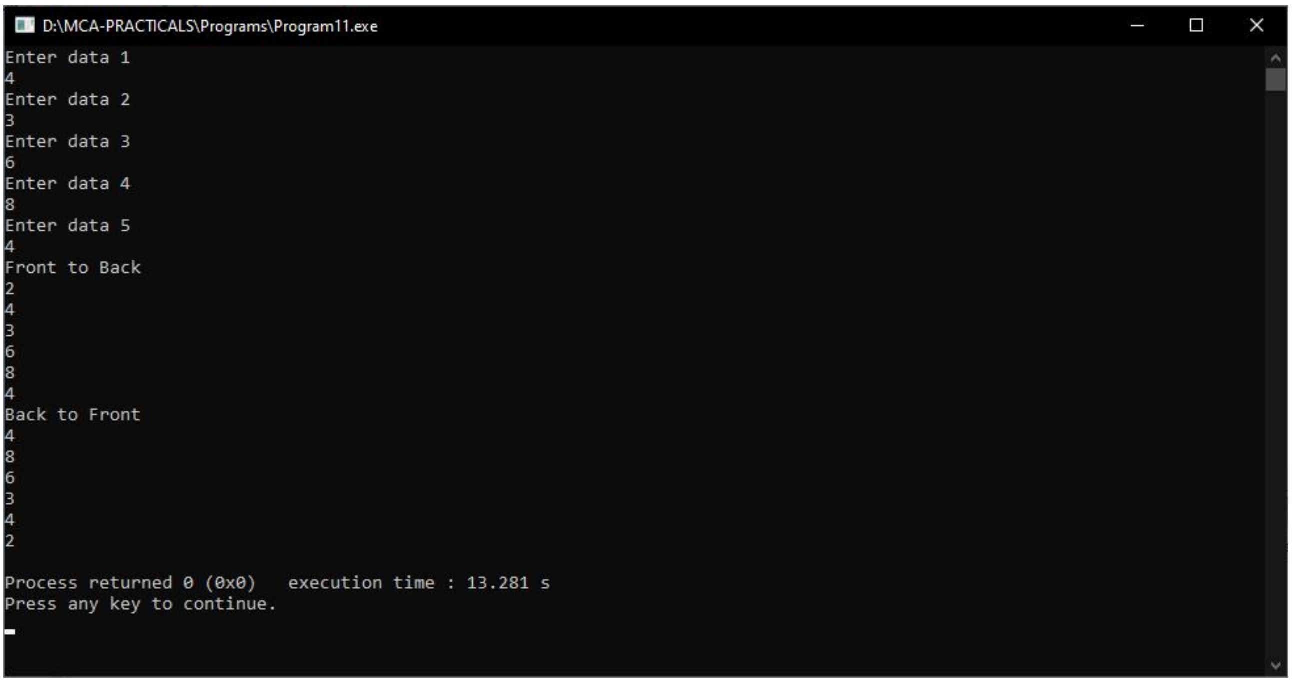
Step 9: Stop the process.

Program:

```
#include <bits/stdc++.h>
#include<iostream>
using namespace std;

class LinkedList {
public:
   int data;
   LinkedList *front;
   LinkedList *back;
};

int main() {
   int i, data;
   LinkedList* t;
   LinkedList* t1;
   LinkedList* t2;
   LinkedList* t3;
   t = new LinkedList;
   t->data = 2;
   t->front = NULL;
   t->back = NULL;
   t2 = t;
   for (i = 1; i <= 5; i++) {
      cout << "Enter data " << i << endl;
      cin >> data;
      t1 = new LinkedList;
      t1->data = data;
      t1->front = NULL;
      t1->back = NULL;
      t->front = t1;
      t1->back = t;
      t = t1;
   }
   t3 = t;
```

```
    cout << "Front to Back" << endl;
    while (t2 != NULL) {
        cout << t2->data << endl;
        t2 = t2->front;
    }
    cout << "Back to Front" << endl;
    while (t3 != NULL) {
        cout << t3->data << endl;
        t3 = t3->back;
    }
    return 0;
}
```

Output:



```
Enter data 1
4
Enter data 2
3
Enter data 3
6
Enter data 4
8
Enter data 5
4
Front to Back
2
4
3
6
8
4
Back to Front
4
8
6
3
4
2

Process returned 0 (0x0)   execution time : 13.281 s
Press any key to continue.
```

12. Double linked list- prime and perfect

Aim

To implement a doubly linked list in C++ and check if the given numbers in the list are prime and perfect numbers.

Algorithm

Step 1: Start the program.

Step 2: Define the `LinkedList` class with members `data`, `front`, and `back`. `data` stores the node value, while `front` and `back` are pointers for forward and backward traversal, respectively.

Step 3: In the `main()` function, initialize pointers `t`, `t1`, `t2`, and `t3` of type `LinkedList*` to manage and traverse the linked list.

Step 4: Create the first node by allocating memory for a new node `t`, setting its data to 2, and initializing both `front` and `back` pointers to `NULL`. Set `t2` to `t` as the head of the list.

Step 5: Use a loop to add 5 new nodes based on user input: prompt the user to enter a value, create a new node `t1`, and link it to the current node `t` by updating the `front` and `back` pointers.

Step 6: Traverse the list from front to back by setting `t3` to `t` to mark the end. While `t2` is not `NULL`, print `t2->data` and check if it is prime by checking divisibility from 2 to `n-1`. Print whether it is prime or not.

Step 7: Traverse the list from back to front by printing "Back to Front". While `t3` is not `NULL`, check if `t3->data` is a perfect number by checking divisibility from 2 to `n-1` and summing divisors. Print whether it is perfect or not.

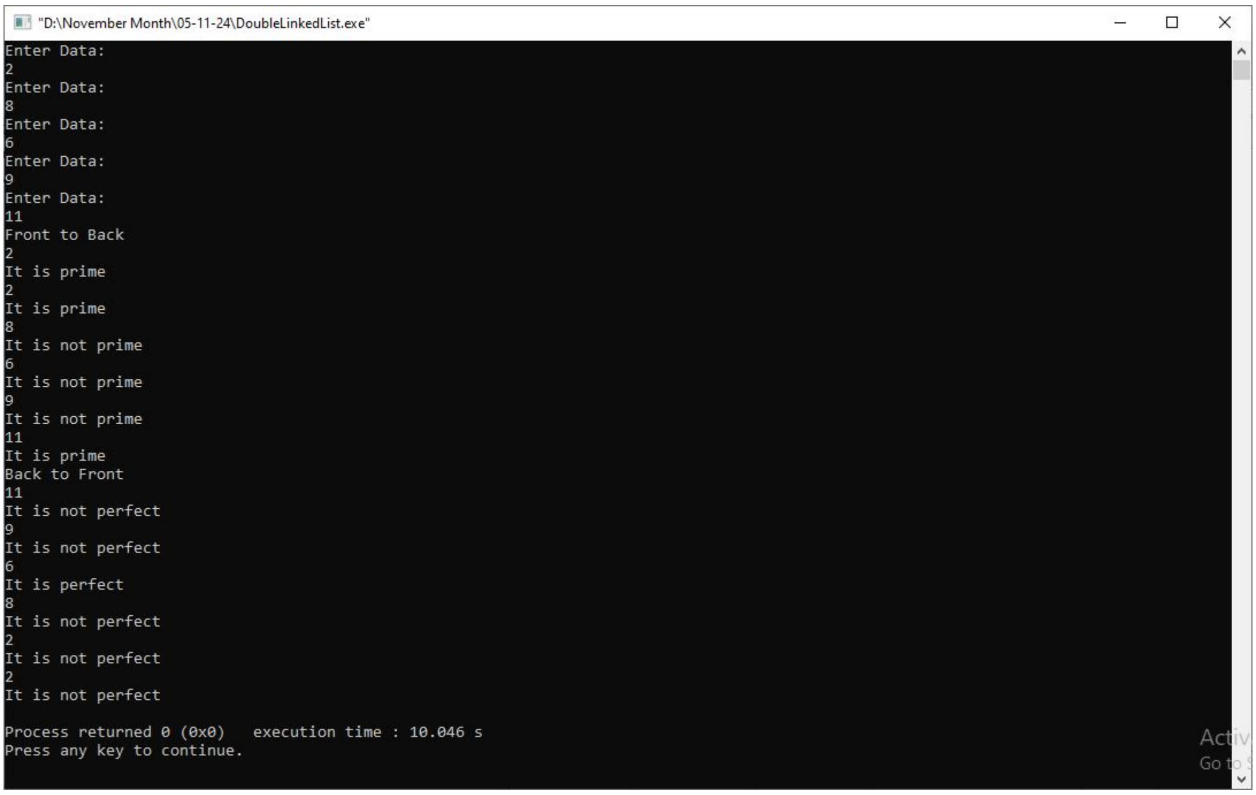
Step 8: Stop the program.

Program:

```
#include<iostream>
#include<bits/stdc++.h>
using namespace std;

class LinkedList
{
public:
    int data;
    LinkedList *front;
    LinkedList *back;
};

int main()
{
    int i, data;
    LinkedList *t;
    LinkedList *t1;
    LinkedList *t2;
    LinkedList *t3;

    t = new LinkedList();
    t->data = 2;
    t->front = NULL;
    t->back = NULL;
    t2 = t;

    for(i = 1; i <= 5; i++)
    {
        cout << "Enter Data: " << endl;
        cin >> data;
        t1 = new LinkedList();
        t1->data = data;
        t1->front = NULL;
        t1->back = NULL;
        t->front = t1;
        t1->back = t;
        t = t1;
    }
    t3 = t;
    cout << "Front to Back " << endl;
    int j;
    while (t2 != NULL)
    {
        cout << t2->data << endl;
        int flag = 1;
        int n = t2->data;
        for(j = 2; j < n; j++)
        {
            if (t2->data % j == 0)
            {
                flag = 0;
            }
        }
        if (flag == 1)
            cout << "It is prime" << endl;
        else
            cout << "It is not prime" << endl;

        t2 = t2->front;
    }
    cout << "Back to Front" << endl;
    while (t3 != NULL)
    {
        int sum = 1;
        cout << t3->data << endl;
        int n = t3->data;
        for(j = 2; j < n; j++)
        {
            if (t3->data % j == 0)
```

```
            sum += j;
        }
        if (sum == t3->data)
            cout << "It is perfect" << endl;
        else
            cout << "It is not perfect" << endl;

        t3 = t3->back;
    }
    return 0;
}
```

Output:



"D:\November Month\05-11-24\DoubleLinkedList.exe"

```
Enter Data:
2
Enter Data:
8
Enter Data:
6
Enter Data:
9
Enter Data:
11
Front to Back
2
It is prime
2
It is prime
8
It is not prime
6
It is not prime
9
It is not prime
11
It is prime
Back to Front
11
It is not perfect
9
It is not perfect
6
It is perfect
8
It is not perfect
2
It is not perfect
2
It is not perfect

Process returned 0 (0x0)   execution time : 10.046 s
Press any key to continue.
```

13. Polynomial addition using linked list

Aim:

To implement a linked list structure for polynomials in C++, allowing for the addition of two polynomials and displaying the resulting polynomial.

Algorithm:

Step 1: Start the process.

Step 2: Define a `Node` class to represent each term of a polynomial, including its coefficient, power, and a pointer to the next term.

Step 3: Implement the `create_node` function to allocate a new node, set its coefficient and power, link it to the existing polynomial, and update the head of the polynomial list.

Step 4: Implement `addPolynomials` to traverse both polynomials, summing coefficients for equal powers and adding remaining terms as needed.

Step 5: Implement the `displayPolynomial` function to traverse the linked list representing a polynomial and print its terms in a readable format.

Step 6: In the `main` function, create two polynomials by calling the `create_node` function with the desired coefficients and powers.

Step 7: Display the two polynomials using the `displayPolynomial` function.

Step 8: Call the `addPolynomials` function to compute the sum of the two polynomials and store the result.

Step 9: Display the sum of the polynomials.

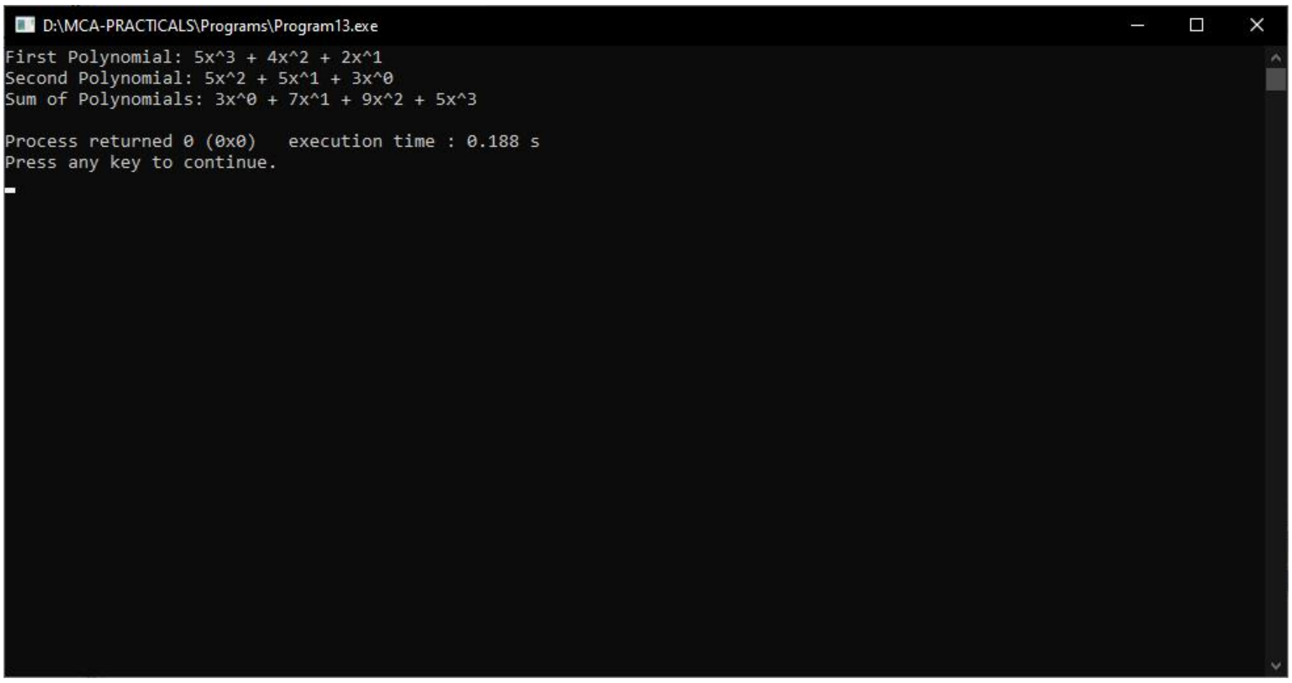
Step 10: Stop the process.

Program:

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

class Node {
public:
   int coeff;
   int pow;
   Node* next;
};

void create_node(int coeff, int pow, Node** poly) {
   Node* newNode = new Node();
   newNode->coeff = coeff;
   newNode->pow = pow;
   newNode->next = *poly;
   *poly = newNode;
}

Node* addPolynomials(Node* poly1, Node* poly2) {
   Node* result = NULL;
   while (poly1 != NULL && poly2 != NULL) {
      if (poly1->pow == poly2->pow) {
         create_node(poly1->coeff + poly2->coeff, poly1->pow, &result);
         poly1 = poly1->next;
         poly2 = poly2->next;
      }
      else if (poly1->pow > poly2->pow) {
         create_node(poly1->coeff, poly1->pow, &result);
         poly1 = poly1->next;
      }
      else {
         create_node(poly2->coeff, poly2->pow, &result);
         poly2 = poly2->next;
      }
```

```
    }
    while (poly1 != NULL) {
        create_node(poly1->coeff, poly1->pow, &result);
        poly1 = poly1->next;
    }
    while (poly2 != NULL) {
        create_node(poly2->coeff, poly2->pow, &result);
        poly2 = poly2->next;
    }
    return result;
}

void displayPolynomial(Node* poly) {
    while (poly != NULL) {
        cout << poly->coeff << "x^" << poly->pow;
        poly = poly->next;
        if (poly != NULL)
            cout << " + ";
    }
    cout << endl;
}

int main() {
    Node* poly1 = NULL;
    Node* poly2 = NULL;
    Node* sum = NULL;
    create_node(2, 1, &poly1);
    create_node(4, 2, &poly1);
    create_node(5, 3, &poly1);

    create_node(3, 0, &poly2);
    create_node(5, 1, &poly2);
    create_node(5, 2, &poly2);
    cout << "First Polynomial: ";
    displayPolynomial(poly1);
    cout << "Second Polynomial: ";
    displayPolynomial(poly2);
    sum = addPolynomials(poly1, poly2);
    cout << "Sum of Polynomials: ";
    displayPolynomial(sum);

    return 0;
}
```

Output:



```
First Polynomial: 5x^3 + 4x^2 + 2x^1
Second Polynomial: 5x^2 + 5x^1 + 3x^0
Sum of Polynomials: 3x^0 + 7x^1 + 9x^2 + 5x^3

Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

14. Polynomial multiplication using linked list

Aim:

To implement polynomial multiplication using a linked list in C++, enabling the creation, multiplication, and display of polynomials.

Algorithm:

Step 1: Start the process.

Step 2: Define a `Node` class to represent each term of the polynomial, including the coefficient, power, and a pointer to the next term.

Step 3: Implement the `create_node` function, which creates a new node for a polynomial term and inserts it at the head of the linked list.

Step 4: Implement the `addTermToPolynomial` function to add or combine terms in the resultant polynomial based on their powers.

Step 5: Implement the `multiplyPolynomials` function to multiply each term of the first polynomial by every term of the second polynomial and add the resulting terms to the product polynomial.

Step 6: Implement the `displayPolynomial` function to traverse the linked list and print the polynomial in a readable format.

Step 7: In the `main` function, create two polynomials by calling the `create_node` function with the desired coefficients and powers.

Step 8: Display both polynomials using the `displayPolynomial` function.

Step 9: Call the `multiplyPolynomials` function to compute the product of the two polynomials and store the result.

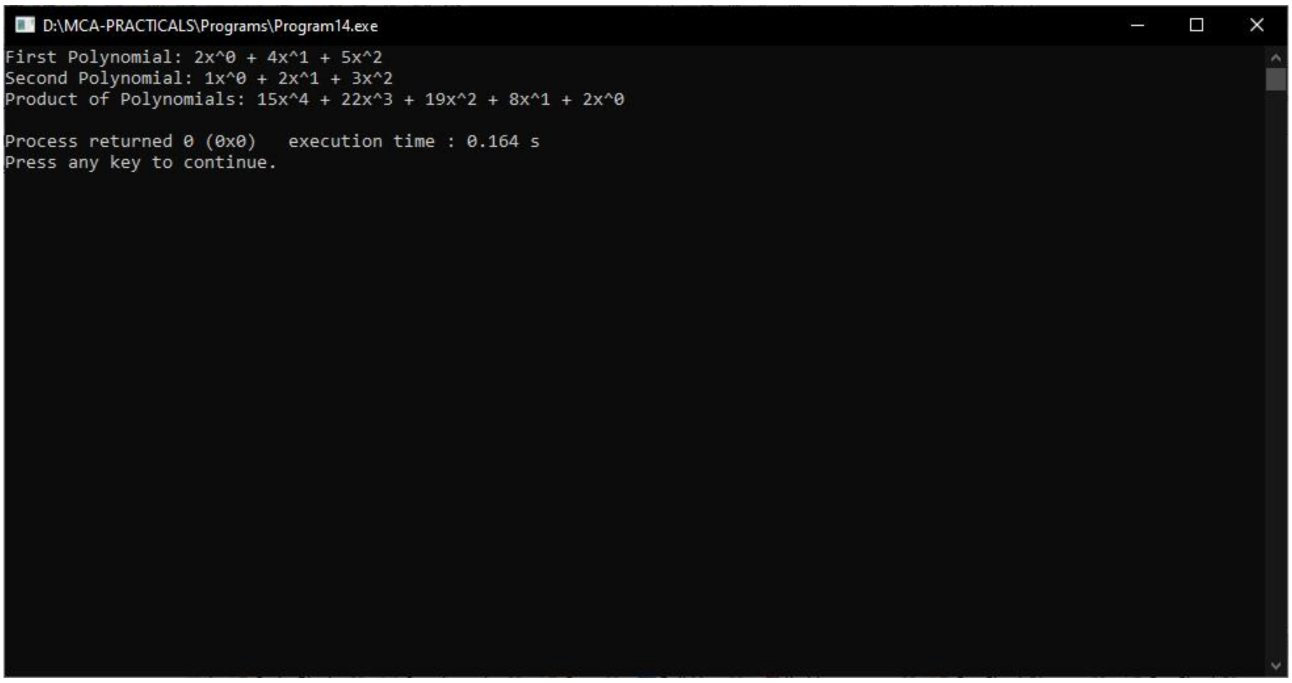
Step 10: Display the product of the polynomials.

Step 11: Stop the process.

Program:

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

class Node {
public:
   int coeff;
   int pow;
   Node* next;
};

void create_node(int coeff, int pow, Node** poly) {
   Node* newNode = new Node();
   newNode->coeff = coeff;
   newNode->pow = pow;
   newNode->next = *poly;
   *poly = newNode;
}

Node* addTermToPolynomial(Node* result, int coeff, int pow) {
   Node* temp = result;
   Node* prev = NULL;
   while (temp != NULL && temp->pow > pow) {
      prev = temp;
      temp = temp->next;
   }
   if (temp != NULL && temp->pow == pow) {
      temp->coeff += coeff;
   }
   else {
      Node* newNode = new Node();
      newNode->coeff = coeff;
      newNode->pow = pow;
      newNode->next = temp;
      if (prev == NULL) {
            result = newNode;
        } else {
            prev->next = newNode;
        }
    }
    return result;
}

Node* multiplyPolynomials(Node* poly1, Node* poly2) {
    Node* result = NULL;
    for (Node* ptr1 = poly1; ptr1 != NULL; ptr1 = ptr1->next) {
        for (Node* ptr2 = poly2; ptr2 != NULL; ptr2 = ptr2->next) {
            int coeff = ptr1->coeff * ptr2->coeff;
            int pow = ptr1->pow + ptr2->pow;
            result = addTermToPolynomial(result, coeff, pow);
        }
    }
    return result;
}

void displayPolynomial(Node* poly) {
    while (poly != NULL) {
        cout << poly->coeff << "x^" << poly->pow;
        poly = poly->next;
        if (poly != NULL)
            cout << " + ";
    }
    cout << endl;
}

int main() {
    Node* poly1 = NULL;
    Node* poly2 = NULL;
    Node* product = NULL;
    create_node(5, 2, &poly1);
    create_node(4, 1, &poly1);
    create_node(2, 0, &poly1);

    create_node(3, 2, &poly2);
    create_node(2, 1, &poly2);
    create_node(1, 0, &poly2);

    cout << "First Polynomial: ";
    displayPolynomial(poly1);

    cout << "Second Polynomial: ";
    displayPolynomial(poly2);

    product = multiplyPolynomials(poly1, poly2);

    cout << "Product of Polynomials: ";
    displayPolynomial(product);

    return 0;
}
```

Output:



```
First Polynomial: 2x^0 + 4x^1 + 5x^2
Second Polynomial: 1x^0 + 2x^1 + 3x^2
Product of Polynomials: 15x^4 + 22x^3 + 19x^2 + 8x^1 + 2x^0

Process returned 0 (0x0)   execution time : 0.164 s
Press any key to continue.
```

15. Recursion - factorial and ackermann function

Aim:

To demonstrate the implementation of fundamental algorithms in C++, including generating the Fibonacci series, calculating factorials, and computing the Ackermann function.

Algorithm:

Step 1: Define a class `simpledsa` for generating the Fibonacci series with both iterative and recursive methods.

Step 2: In the iterative Fibonacci method, initialize the first two Fibonacci numbers and print them.

Step 3: Use a loop to compute and print the next 20 Fibonacci numbers by updating the previous two numbers iteratively.

Step 4: In the recursive Fibonacci method, use recursion to print Fibonacci numbers until the count reaches 20.

Step 5: Define a class `factorial` for calculating the factorial of a number using both iterative and recursive methods.

Step 6: In the iterative method, hardcode an input value, compute its factorial using a loop, and print the result.

Step 7: Implement a recursive method to compute the factorial of a given input, returning the result.

Step 8: Define a function to compute the Ackermann function with two integer parameters mmm and nnn.

Step 9: In the main function, prompt the user for input values, call the Ackermann function, and display the result.

Step 10: Stop the process.

Program:

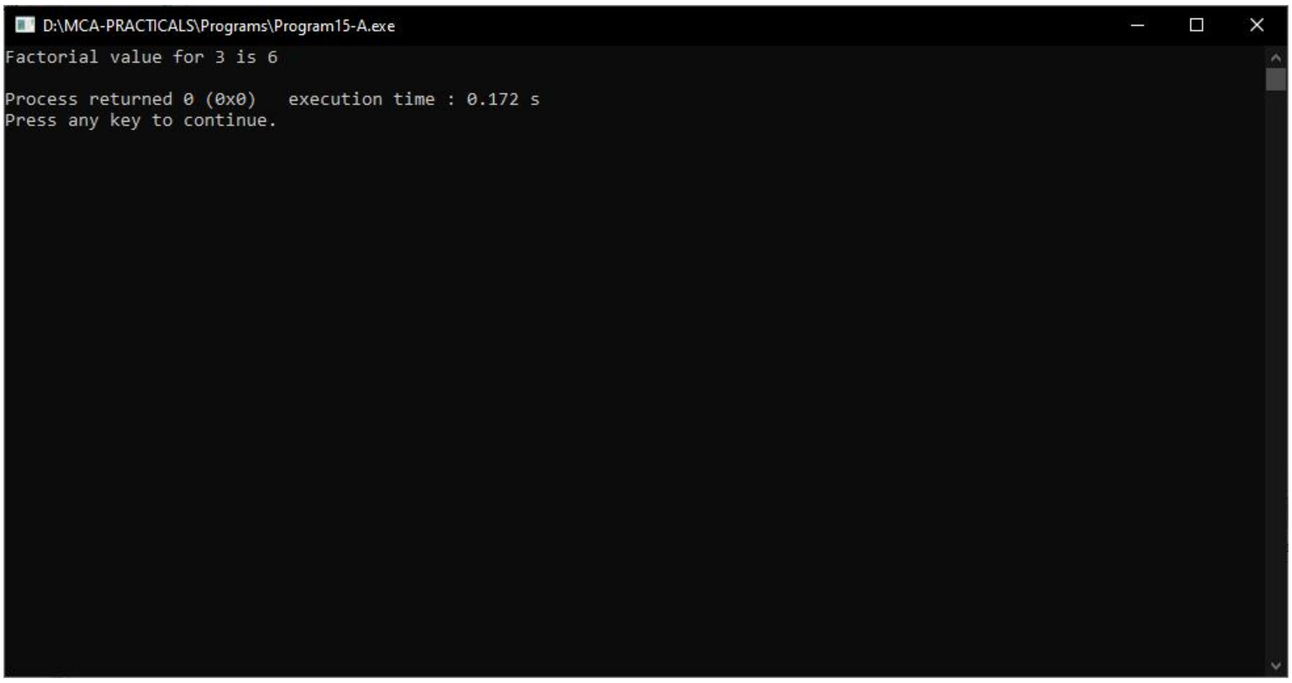
a) Factorial

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

class simpledsa {
public:
   void fibnnaci() {
      int a, b, c, i;
      a = 1;
      b = 1;
      cout << a << endl;
      cout << b << endl;
      for (i = 1; i <= 20; i++) {
         c = a + b;
         cout << c << endl;
         a = b;
         b = c;
      }
   }

   void recursion(int a, int b, int i) {
      int c;
      if (i <= 20) {
         c = a + b;
         cout << c << endl;
         a = b;
         b = c;
         i = i + 1;
         recursion(a, b, i);
      }
   }
};

class factorial {
```

```
private:
   int a;
   void input() {
      cout << "Enter the number for factorial" << endl;
      cin >> a;
   }

public:
   int f = 1, i = 1;
   void factorial1() {
      int i1;
      a = 3;
      for (i1 = 1; i1 <= a; i1++) {
         f = f * i1;
      }
      cout << "Factorial value for " << a << " is " << f << endl;
   }

   int resfactorial(int a) {
      if (a >= 1) {
         f = a * resfactorial(a - 1);
         return f;
      } else {
         return 1;
      }
   }
};

int main() {
   simpledsa s1;
   factorial f1;
   int k;
   s1.fibnnaci();
   s1.recursion(1, 1, 1);
   f1.factorial1();
```

```
    k = f1.resfactorial(3);
    cout << k;
    return 0;
}
```

Output:

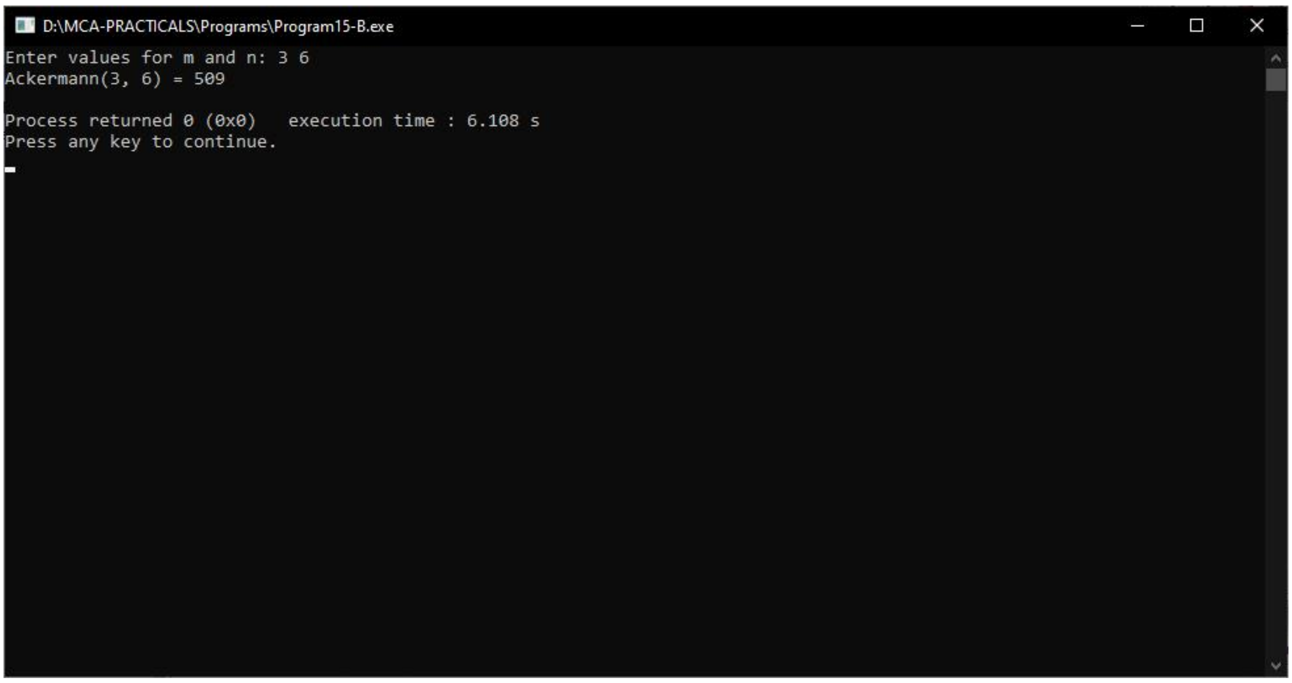

D:\MCA-PRACTICALS\Programs\Program15-A.exe

```
Factorial value for 3 is 6

Process returned 0 (0x0)   execution time : 0.172 s
Press any key to continue.
```

b) Ackermann function

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

int ackermann(int m, int n) {
   if (m == 0) {
      return n + 1;
   } else if (m > 0 && n == 0) {
      return ackermann(m - 1, 1);
   } else {
      return ackermann(m - 1, ackermann(m, n - 1));
   }
}

int main() {
   int m, n;
   cout << "Enter values for m and n: ";
   cin >> m >> n;
   int result = ackermann(m, n);
   cout << "Ackermann(" << m << ", " << n << ") = " << result << endl;
   return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program15-B.exe

```
Enter values for m and n: 3 6
Ackermann(3, 6) = 509

Process returned 0 (0x0)   execution time : 6.108 s
Press any key to continue.
```

16. Binary tree traversal

Aim

To implement a binary tree in C++ and demonstrate in-order, pre-order, and post-order traversals.

Algorithm

Step 1: Start the process.

Step 2: Define a `Node` class that represents a node in the binary tree, storing an integer `data`, a left child pointer `left`, and a right child pointer `right`.

Step 3: Create a function `createNode` that takes an integer value, initializes a `Node` with this value, and returns a pointer to the newly created node.

Step 4: Implement the `inOrderTraversal` function for in-order traversal, where the left subtree is traversed first, then the root node, followed by the right subtree.

Step 5: Implement the `preOrderTraversal` function for pre-order traversal, visiting the root node first, followed by the left and right subtrees.

Step 6: Implement the `postOrderTraversal` function for post-order traversal, traversing the left and right subtrees first and visiting the root node last.

Step 7: In the `main` function, create the root node and its child nodes to form a binary tree with specified values.

Step 8: Call `inOrderTraversal` on the root node and display the sequence of visited nodes as the in-order traversal output.

Step 9: Call `preOrderTraversal` on the root node and display the sequence of visited nodes as the pre-order traversal output.

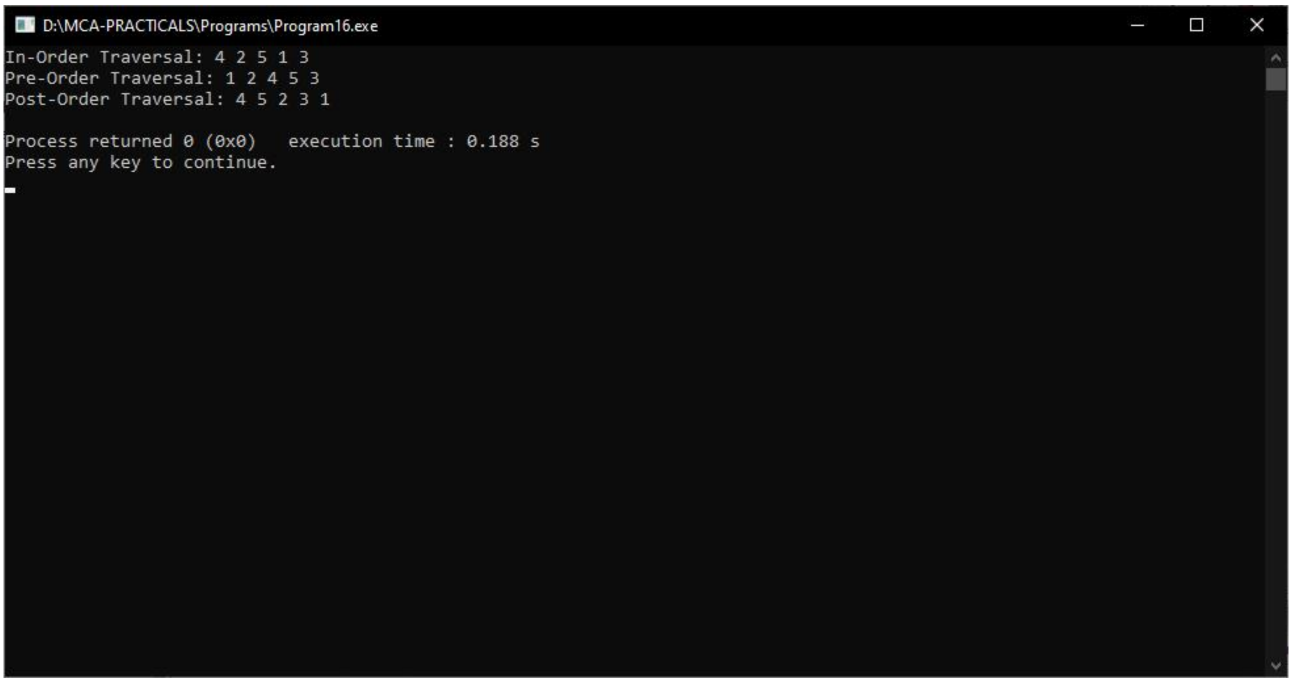
Step 10: Call `postOrderTraversal` on the root node and display the sequence of visited nodes as the post-order traversal output.

Step 11: Stop the process.

Program:

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

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = right = NULL;
    }
};

Node* createNode(int data) {
    return new Node(data);
}

void inOrderTraversal(Node* node) {
    if (node == NULL) return;
    inOrderTraversal(node->left);
    cout << node->data << " ";
    inOrderTraversal(node->right);
}

void preOrderTraversal(Node* node) {
    if (node == NULL) return;
    cout << node->data << " ";
    preOrderTraversal(node->left);
    preOrderTraversal(node->right);
}

void postOrderTraversal(Node* node) {
    if (node == NULL) return;
    postOrderTraversal(node->left);
    postOrderTraversal(node->right);
    cout << node->data << " ";
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    cout << "In-Order Traversal: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    preOrderTraversal(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    postOrderTraversal(root);
    cout << endl;

    return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program16.exe

```
In-Order Traversal: 4 2 5 1 3
Pre-Order Traversal: 1 2 4 5 3
Post-Order Traversal: 4 5 2 3 1

Process returned 0 (0x0)   execution time : 0.188 s
Press any key to continue.
```

17. Binary search tree- sorting

Aim

To implement a Binary Search Tree (BST) in C++ that allows insertion of nodes and displays the tree's contents in sorted order through in-order traversal.

Algorithm

Step 1: Start the process.

Step 2: Define a `Node` class that represents a BST node, holding an integer `data` and two pointers: `left` and `right`, which point to the left and right child nodes, respectively.

Step 3: Create a constructor for the `Node` class to initialize `data` with a given value and set both `left` and `right` pointers to `NULL`.

Step 4: Define the `insert` function that takes a `root` node and a `value` as inputs. If the `root` is `NULL`, create and return a new `Node` with the given `value`.

Step 5: In the `insert` function, check if `value` is smaller than `root->data`. If true, recursively call `insert` on the left subtree. If `value` is greater, recursively call `insert` on the right subtree.

Step 6: Return the root node from the `insert` function after inserting the new node in the correct position.

Step 7: Define the `inOrderTraversal` function to perform an in-order traversal on the tree, which visits nodes in sorted order. Traverse the left subtree, print `root->data`, and then traverse the right subtree.

Step 8: In the `main` function, initialize an empty `root` and an array of integers to insert into the BST.

Step 9: Loop through the array, calling the `insert` function for each integer to build the BST.

Step 10: Print the numbers in sorted order by calling `inOrderTraversal` on the root node.

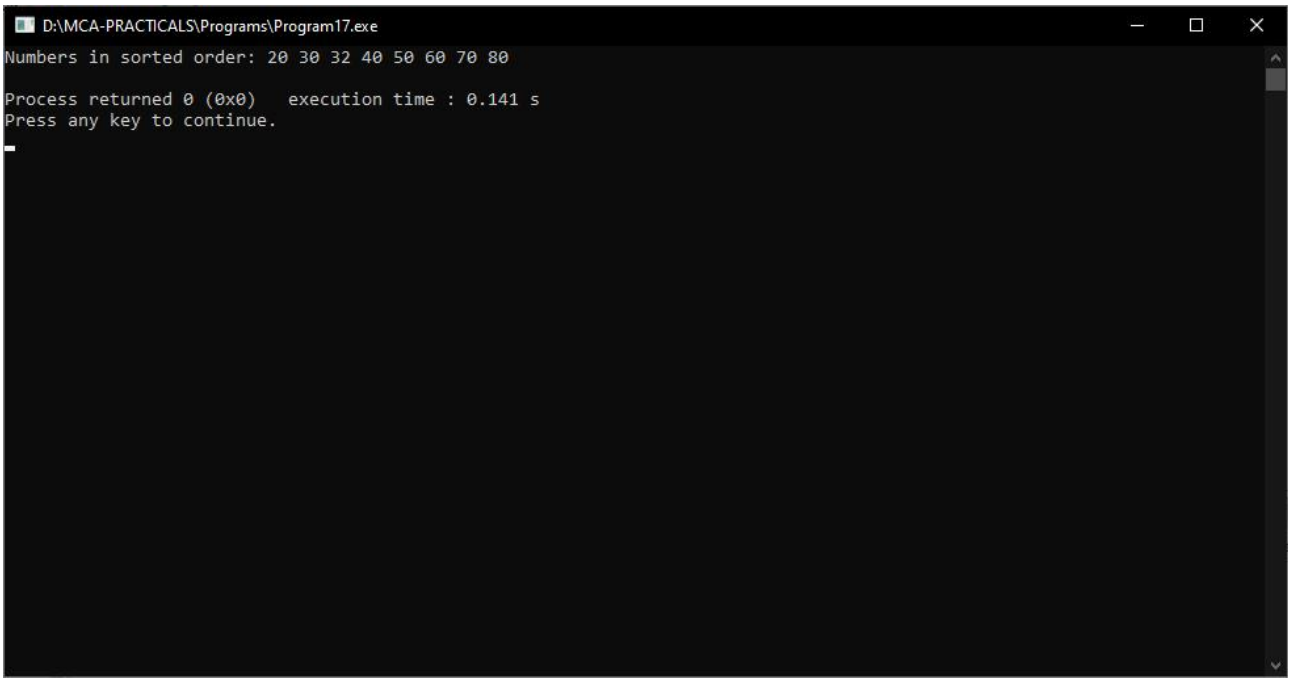
Step 11: Stop the process

Program:

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

class Node {
public:
   int data;
   Node* left;
   Node* right;

   Node(int value) {
      data = value;
      left = right = NULL;
   }
};

Node* insert(Node* root, int value) {
   if (root == NULL) {
      return new Node(value);
   }
   if (value < root->data) {
      root->left = insert(root->left, value);
   } else if (value > root->data) {
      root->right = insert(root->right, value);
   }
   return root;
}

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

```
int main() {
    Node* root = NULL;
    int numbers[] = {50, 30, 20, 40, 70, 60, 80, 32};
    int n = sizeof(numbers) / sizeof(numbers[0]);

    for (int i = 0; i < n; i++) {
        root = insert(root, numbers[i]);
    }

    cout << "Numbers in sorted order: ";
    inOrderTraversal(root);
    cout << endl;
    return 0;
}
```

Output:



```
Numbers in sorted order: 20 30 32 40 50 60 70 80

Process returned 0 (0x0)   execution time : 0.141 s
Press any key to continue.
```

18. Sorting-bubble

Aim

To implement a Bubble Sort algorithm in C++ that sorts a given array of integers in ascending order.

Algorithm

Step 1: Start the process.

Step 2: Define a `bubblesorting` class with an integer array `arr` and integer variable `n` to store the size of the array.

Step 3: Define the `bubbleSort` function, which takes an array and its size as parameters. This function sorts the array using the Bubble Sort algorithm.

Step 4: In `bubbleSort`, iterate over the array with two nested loops. The outer loop runs from `0` to `n-1`, while the inner loop compares adjacent elements from `0` to `n-i-1`.

Step 5: If any element is greater than the next, swap them. Repeat this process to "bubble up" the largest unsorted element to its correct position by the end of each pass.

Step 6: Define an `input` function to prompt the user to enter `n` (5 in this case) elements and store them in `arr`. Call `bubbleSort` to sort `arr`.

Step 7: Define an `output` function that iterates through `arr` to display each sorted element.

Step 8: In the `main` function, create an object of the `bubblesorting` class.

Step 9: Call `input` to read the array elements and sort them, and then call `output` to display the sorted array.

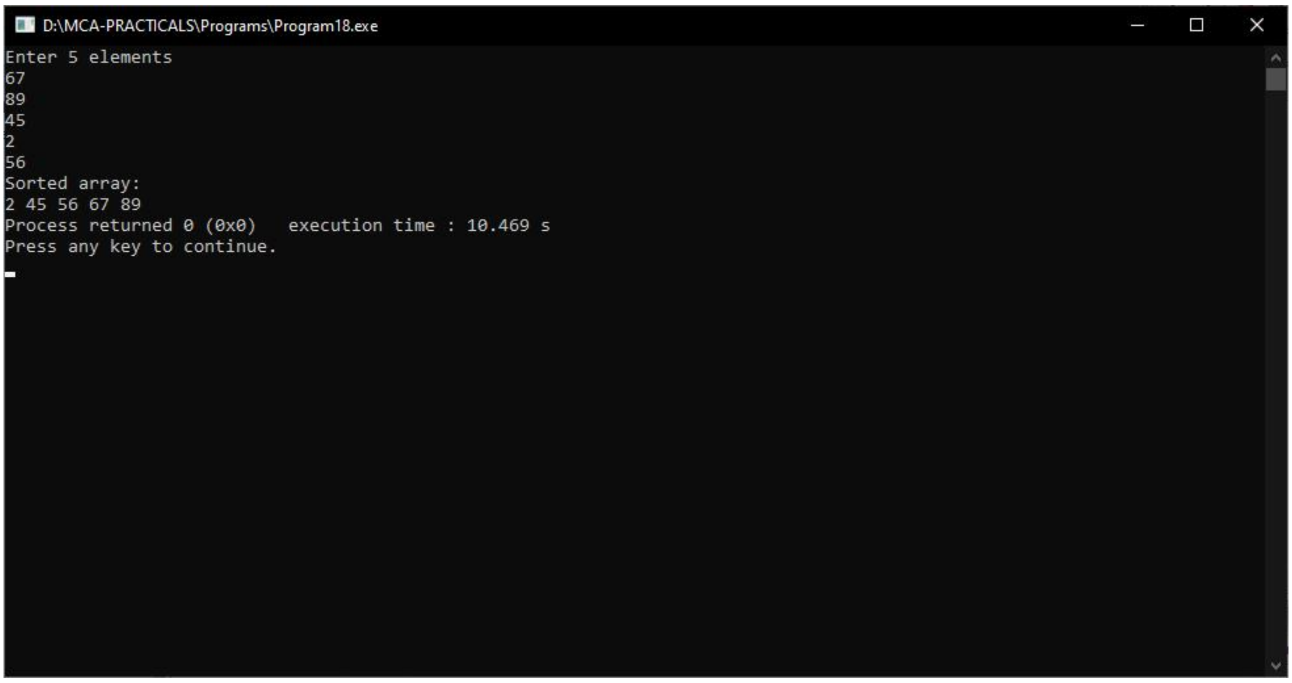
Step 10: Stop the process.

Program:

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

class bubblesorting {
public:
  int arr[10];
  int n;

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

  void input() {
    n = 5;
    cout << "Enter 5 elements" << endl;
    for (int i = 0; i < 5; i++) {
      cin >> arr[i];
    }
    bubbleSort(arr, n);
  }
  void output() {
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
      cout << arr[i] << " ";
  }
};
```

```
int main() {
   bubblesorting b1;
   b1.input();
   b1.output();
   return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program18.exe

```
Enter 5 elements
67
89
45
2
56
Sorted array:
2 45 56 67 89
Process returned 0 (0x0)   execution time : 10.469 s
Press any key to continue.
```

19. Sorting- selection

Aim

To implement the Selection Sort algorithm in C++ to sort an array of integers in ascending order.

Algorithm

Step 1: Start the process.

Step 2: Define a `selectionsorting` class with an integer array `arr` and integer variable `n` to store the size of the array.

Step 3: Define the `selectionSort` function, which takes an array and its size as parameters and sorts the array using the Selection Sort algorithm.

Step 4: In `selectionSort`, iterate over the array with a loop, selecting the minimum element from the unsorted part of the array in each pass.

Step 5: For each element at position `i`, set `minIndex` to `i` and compare it with each element in the rest of the array. If a smaller element is found, update `minIndex`.

Step 6: Once the minimum element is found, swap it with the element at position `i`.

Step 7: Define an `input` function that prompts the user to enter `n` (5 in this case) elements, stores them in `arr`, and calls `selectionSort` to sort `arr`.

Step 8: Define an `output` function that iterates through `arr` to display each sorted element.

Step 9: In the `main` function, create an object of the `selectionsorting` class.

Step 10: Call `input` to read and sort the array elements, then call `output` to display the sorted array.

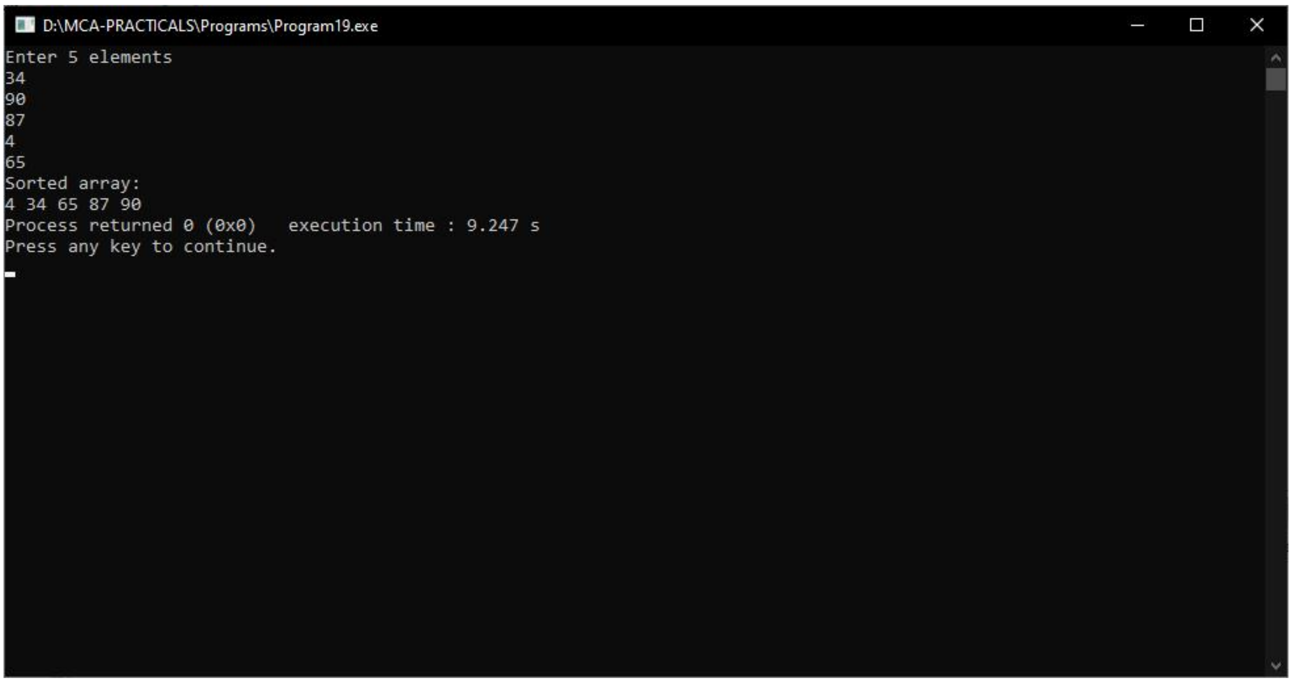
Step 11: Stop the process.

Program:

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

class selectionsorting {
public:
   int arr[10];
   int n;

   void selectionSort(int arr[], int n) {
      for (int i = 0; i < n-1; i++) {
         int minIndex = i;
         for (int j = i+1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
               minIndex = j;
            }
         }
         int temp = arr[minIndex];
         arr[minIndex] = arr[i];
         arr[i] = temp;
      }
   }

   void input() {
      n = 5;
      cout << "Enter 5 elements" << endl;
      for (int i = 0; i < 5; i++) {
         cin >> arr[i];
      }
      selectionSort(arr, n);
   }

   void output() {
      cout << "Sorted array: \n";
      for (int i = 0; i < n; i++)
```

```
            cout << arr[i] << " ";
    }
};

int main() {
    selectionsorting s1;
    s1.input();
    s1.output();
    return 0;
}
```

Output:



```
Enter 5 elements
34
90
87
4
65
Sorted array:
4 34 65 87 90
Process returned 0 (0x0)   execution time : 9.247 s
Press any key to continue.
```

20. Sorting- insertion

Aim

To implement the Insertion Sort algorithm in C++ to sort an array of integers in ascending order.

Algorithm

Step 1: Start the process.

Step 2: Define a class `insertionsorting` with an integer array `arr` and a variable `n` for the array size.

Step 3: Create the `insertionSort` function, which sorts `arr` by moving each `key` element to its correct position within the sorted part of the array.

Step 4: In `insertionSort`, iterate through `arr`, setting each element as `key` and shifting larger elements one position forward to make room for the `key`.

Step 5: Insert `key` in its correct position within the sorted portion of `arr`.

Step 6: Define an `input` function to read `n` elements from the user into `arr` and call `insertionSort` to sort the array.

Step 7: Define an `output` function that displays each element in the sorted `arr`.

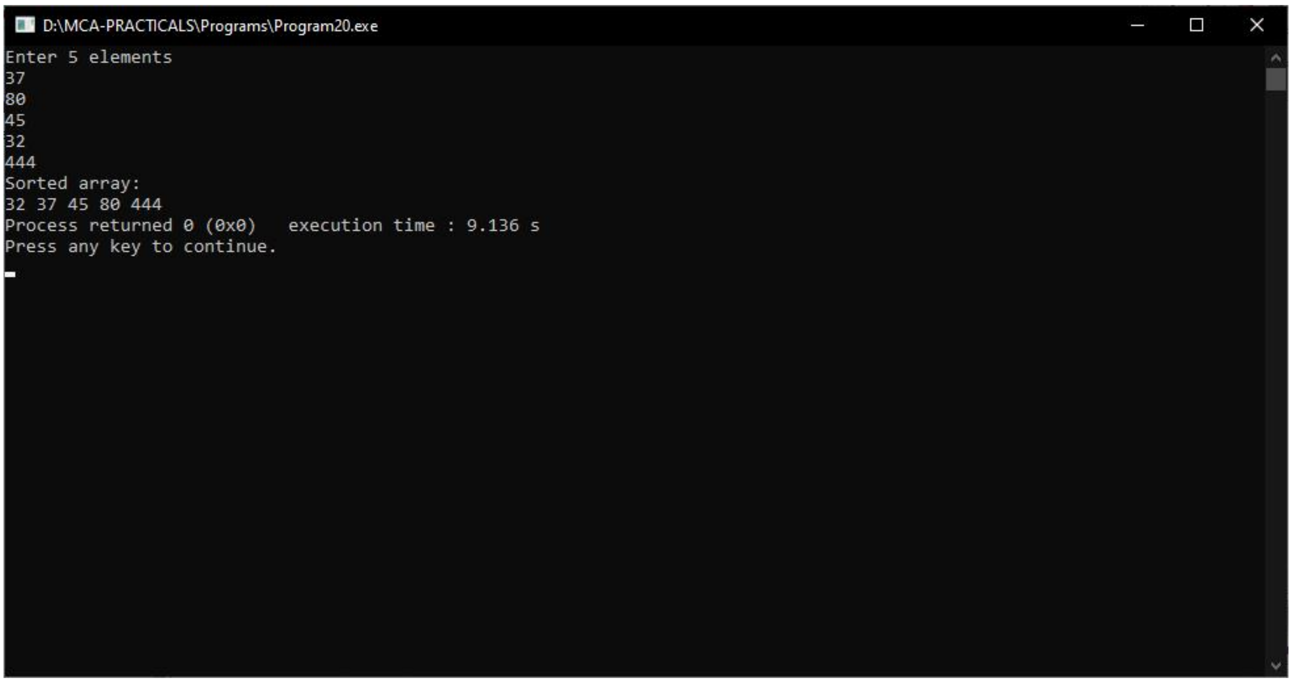
Step 8: In `main`, create an object of the `insertionsorting` class, then call `input` to sort the array and `output` to display it.

Step 9: Stop the process.

Program:

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

class insertionsorting {
public:
   int arr[10];
   int n;

   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 input() {
      n = 5;
      cout << "Enter 5 elements" << endl;
      for (int i = 0; i < 5; i++) {
         cin >> arr[i];
      }
      insertionSort(arr, n);
   }

   void output() {
      cout << "Sorted array: \n";
      for (int i = 0; i < n; i++)
         cout << arr[i] << " ";
   }
};

int main() {
    insertionsorting i;
    i.input();
    i.output();
    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program20.exe

```
Enter 5 elements
37
80
45
32
444
Sorted array:
32 37 45 80 444
Process returned 0 (0x0)   execution time : 9.136 s
Press any key to continue.
```

21. Sorting - quick

Aim

To implement the Quick Sort algorithm in C++ to sort an array of integers in ascending order.

Algorithm

Step 1: Start the process.

Step 2: Define the `partition` function, where the last element of the array is chosen as the pivot.

Step 3: Initialize `i` as the index of the smaller element, and iterate through the array comparing each element with the pivot.

Step 4: For each element less than the pivot, increment `i` and swap the element at `j` with the element at `i`.

Step 5: After the loop, place the pivot in its correct position by swapping it with the element at `i + 1`.

Step 6: Define the `quickSort` function, which calls `partition` and recursively sorts the subarrays to the left and right of the pivot.

Step 7: In `main`, define an array `arr` and calculate its size `n`.

Step 8: Call `quickSort` on `arr` with `0` as the starting index and `n-1` as the ending index.

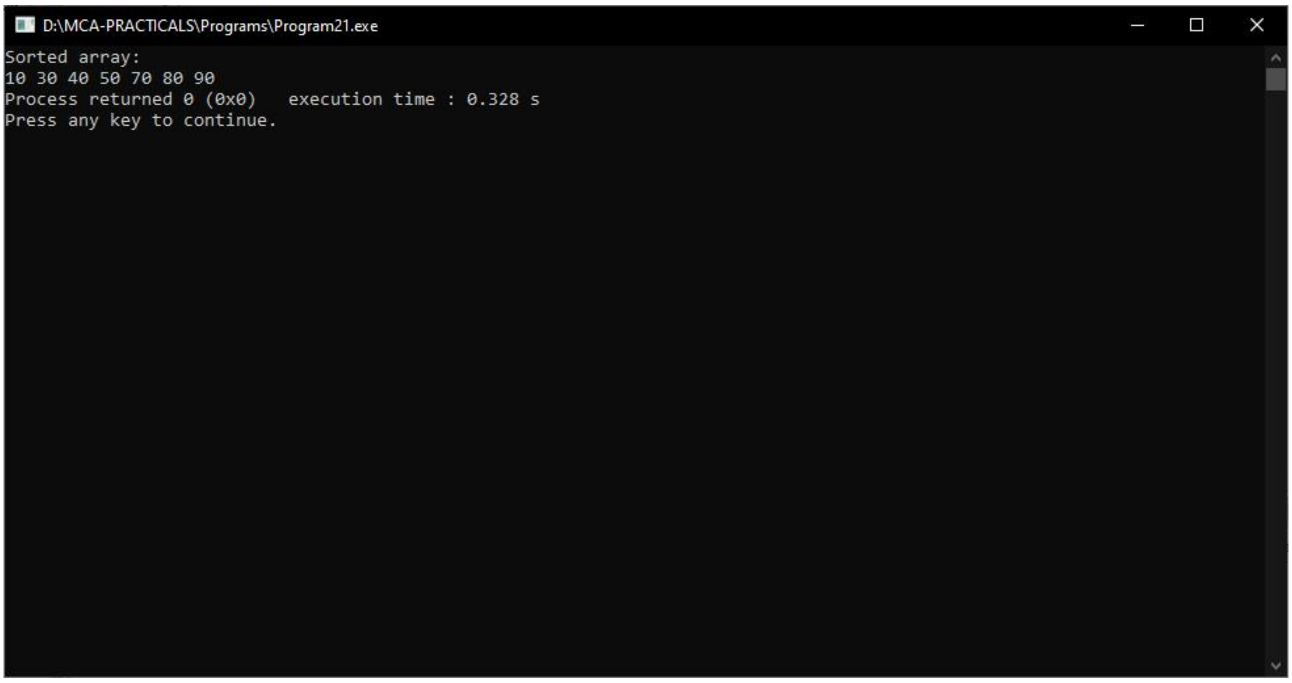
Step 9: Output the sorted array elements.

Step 10: Stop the process.

Program:

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

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; 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[] = {10, 80, 30, 90, 40, 50, 70};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Output:



```
Sorted array:
10 30 40 50 70 80 90
Process returned 0 (0x0)   execution time : 0.328 s
Press any key to continue.
```

22. Sorting - merge

Aim

To implement the Merge Sort algorithm in C++ for sorting an array of integers in ascending order.

Algorithm

Step 1: Start the process.

Step 2: Define the `merge` function to merge two halves of the array, creating temporary arrays for the left and right segments. Stop the process.

Step 3: Copy data from the original array to the temporary arrays `L` and `R`. Stop the process.

Step 4: Compare elements from the temporary arrays and copy the smaller element back into the original array. Stop the process.

Step 5: Copy the remaining elements of the other temporary array into the original array if one array is exhausted. Stop the process.

Step 6: Define the `mergeSort` function to recursively sort the array by checking if the left index is less than the right index. Stop the process.

Step 7: Calculate the middle index `m`, and recursively call `mergeSort` on the left half and right half of the array. Stop the process.

Step 8: Call the `merge` function to combine the sorted halves. Stop the process.

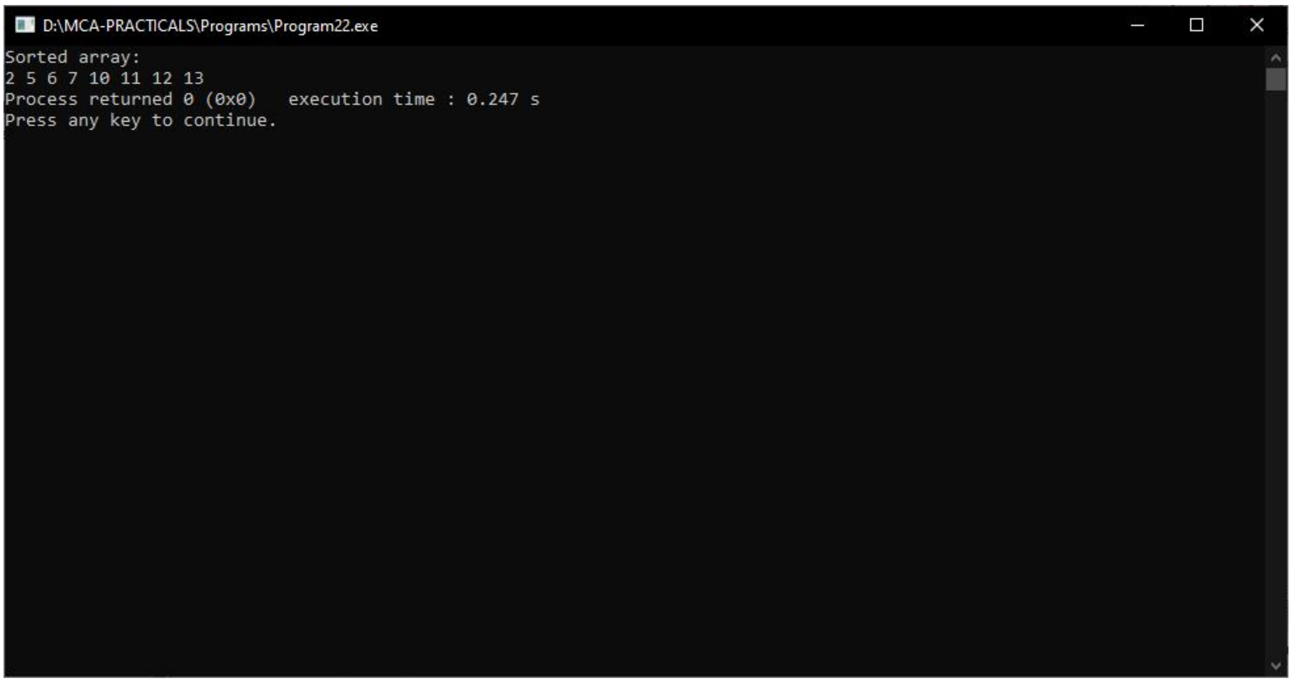
Step 9: In `main`, define an array `arr`, calculate its size, call `mergeSort` on the entire array, and output the sorted array elements.

Step 10: Stop the process.

Program:

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

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1, n2 = r - m, L[n1], R[n2];
    for (int i = 0; i < n1; i++) L[i] = arr[l + i];
    for (int i = 0; i < n2; i++) R[i] = arr[m + 1 + i];
    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 - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

int main() {
    int arr[] = {12, 11, 13, 5, 6, 7, 10, 2};
    int arr_size = sizeof(arr) / sizeof(arr[0]);
    mergeSort(arr, 0, arr_size - 1);
    cout << "Sorted array: \n";
    for (int i = 0; i < arr_size; i++) cout << arr[i] << " ";
    return 0;
}
```

Output:


D:\MCA-PRACTICALS\Programs\Program22.exe

```
Sorted array:
2 5 6 7 10 11 12 13
Process returned 0 (0x0)   execution time : 0.247 s
Press any key to continue.
```

23. Threaded binary tree

Aim

To implement a Threaded Binary Tree in C++ and perform in-order traversal to efficiently visit all nodes.

Algorithm

Step 1: Start the process.

Step 2: Define the `Node` class, including data, left and right pointers, and a boolean `rightThreaded` to indicate if the right pointer is a thread. Stop the process.

Step 3: Create the `inOrderTraversal` function to traverse the threaded binary tree in in-order. Start at the leftmost node, printing node values, and use threading to move to the next node. Stop the process.

Step 4: Define the `insert` function to find the correct position for a new node based on its value. Stop the process.

Step 5: In the `insert` function, handle the insertion logic for the new node, ensuring to set the appropriate threads for in-order traversal. Stop the process.

Step 6: In `main`, initialize the root of the tree as NULL and insert nodes into the threaded binary tree. Stop the process.

Step 7: Call the `inOrderTraversal` function to print the values of the nodes in sorted order. Stop the process.

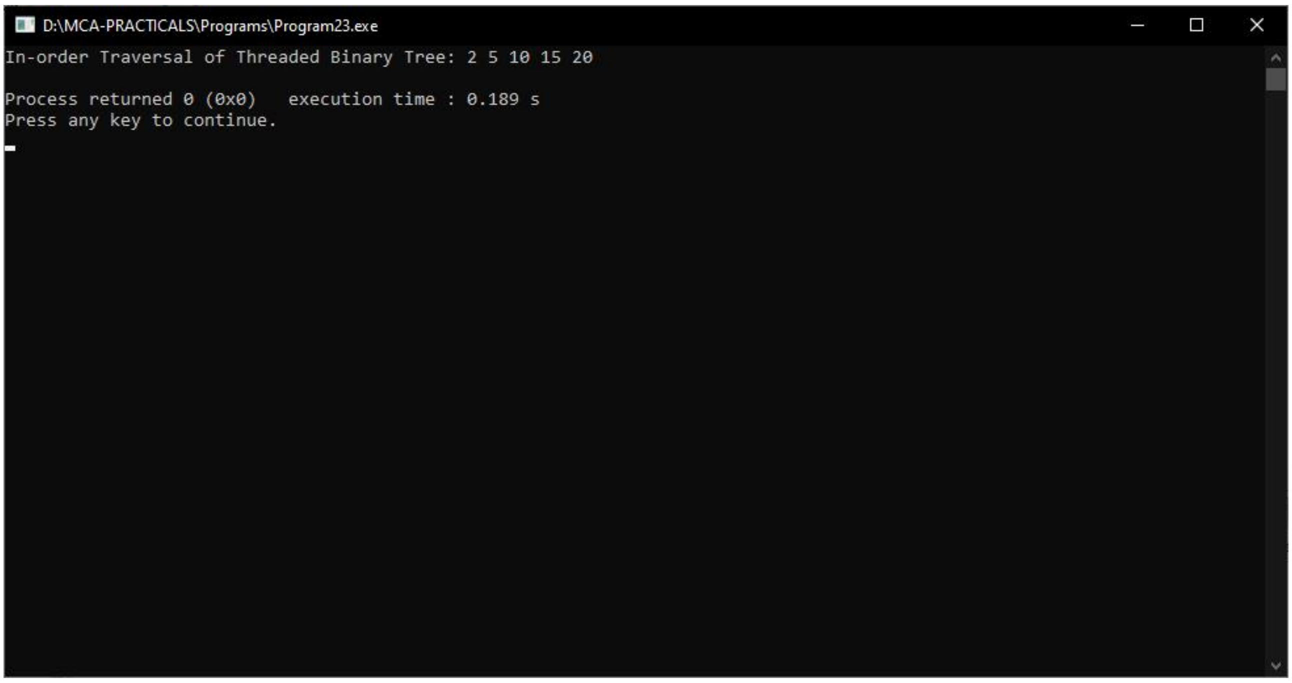
Step 8: Print the result of the in-order traversal. Stop the process.

Step 9: Stop the process.

Program:

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

class Node {
public:
    int data;
    Node* left;
    Node* right;
    bool rightThreaded;
    Node(int value) {
        data = value;
        left = right = NULL;
        rightThreaded = false;
    }
};

void inOrderTraversal(Node* root) {
    Node* current = root;
    while (current->left != NULL) current = current->left;
    while (current != NULL) {
        cout << current->data << " ";
        if (current->rightThreaded) current = current->right;
        else {
            current = current->right;
            while (current != NULL && current->left != NULL) current = current->left;
        }
    }
}

Node* insert(Node* root, int value) {
    Node* parent = NULL;
    Node* current = root;
    while (current != NULL) {
        parent = current;
        if (value < current->data) {
            if (current->left == NULL) break;
            current = current->left;
        } else {
            if (!current->rightThreaded && current->right != NULL) break;
            current = current->right;
        }
    }
    Node* newNode = new Node(value);
    if (parent == NULL) return newNode;
    if (value < parent->data) {
        parent->left = newNode;
        newNode->right = parent;
        newNode->rightThreaded = true;
    } else {
        newNode->right = parent->right;
        newNode->rightThreaded = parent->rightThreaded;
        parent->right = newNode;
        parent->rightThreaded = false;
    }
    return root;
}

int main() {
    Node* root = NULL;
    root = insert(root, 10);
    root = insert(root, 5);
    root = insert(root, 2);
    root = insert(root, 20);
    root = insert(root, 15);
    cout << "In-order Traversal of Threaded Binary Tree: ";
    inOrderTraversal(root);
    cout << endl;
    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program23.exe

```
In-order Traversal of Threaded Binary Tree: 2 5 10 15 20

Process returned 0 (0x0)   execution time : 0.189 s
Press any key to continue.
```

24. Graph

Aim

To implement an undirected graph using an adjacency matrix in C++ and demonstrate how to add edges and display the adjacency matrix.

Algorithm

Step 1: Start the process.

Step 2: Define the `Graph` class that contains the number of vertices and a 2D array for the adjacency matrix. Stop the process.

Step 3: Implement the constructor of the `Graph` class to initialize the number of vertices and allocate memory for the adjacency matrix, setting all values to 0. Stop the process.

Step 4: Create the `addEdge` method to add an edge between two vertices by updating the adjacency matrix to indicate the presence of an edge. Stop the process.

Step 5: Implement the `displayMatrix` method to iterate through the adjacency matrix and print its contents. Stop the process.

Step 6: Implement the destructor of the `Graph` class to free the allocated memory for the adjacency matrix. Stop the process.

Step 7: In the `main` function, create an instance of the `Graph` class with a specified number of vertices. Stop the process.

Step 8: Use the `addEdge` method to add edges between specified vertices. Stop the process.

Step 9: Call the `displayMatrix` method to output the adjacency matrix representation of the graph. Stop the process.

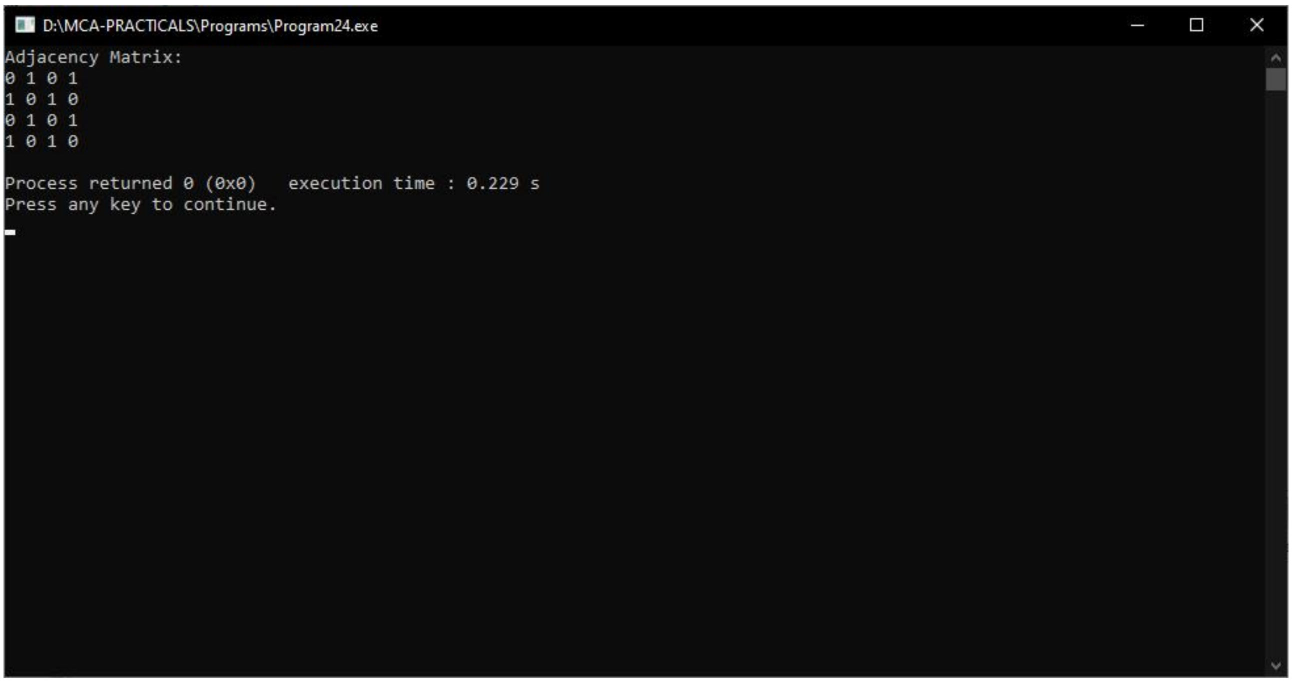
Step 10: Stop the process.

Program:

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

class Graph {
    int numVertices;
    int** adjMatrix;

public:
    Graph(int vertices) {
        numVertices = vertices;
        adjMatrix = new int*[vertices];
        for (int i = 0; i < vertices; i++) {
            adjMatrix[i] = new int[vertices];
            for (int j = 0; j < vertices; j++)
                adjMatrix[i][j] = 0;
        }
    }

    void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1;
    }

    void displayMatrix() {
        for (int i = 0; i < numVertices; i++) {
            for (int j = 0; j < numVertices; j++) {
                cout << adjMatrix[i][j] << " ";
            }
            cout << endl;
        }
    }

    ~Graph() {
        for (int i = 0; i < numVertices; i++)
```

```
            delete[] adjMatrix[i];
        delete[] adjMatrix;
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(2, 3);

    cout << "Adjacency Matrix:" << endl;
    g.displayMatrix();

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program24.exe

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

Process returned 0 (0x0)   execution time : 0.229 s
Press any key to continue.
```

25. Symmetric or non symmetric

Aim

To create a C++ program that checks if a graph is symmetric (undirected) or non-symmetric (directed) by analyzing its adjacency matrix.

Algorithm

Step 1: Start the process.

Step 2: Define the `Graph` class with members `numVertices` for vertex count and `adjMatrix`, a 2D integer array for the adjacency matrix.

Step 3: Define the constructor `Graph(int vertices)` to initialize `numVertices`, dynamically allocate `adjMatrix`, and set all elements to 0 (indicating no edges).

Step 4: Define the `addEdge(int u, int v)` function to set `adjMatrix[u][v]` to 1, representing a directed edge from vertex `u` to vertex `v`.

Step 5: Define the `displayMatrix()` function to print the adjacency matrix, showing connections between vertices.

Step 6: Define the `checkDirectedOrNot()` function, initializing a flag variable `flag` to 1, assuming the graph is symmetric.

Step 7: Use a loop to check each vertex pair `(i, j)`; if `adjMatrix[i][j]` is 1 and `adjMatrix[j][i]` is not, set `flag` to 0.

Step 8: After the loop, if `flag` is 1, print "Symmetric Graph"; otherwise, print "Non-Symmetric Graph."

Step 9: Define the destructor `~Graph()` to deallocate `adjMatrix` memory.

Step 10: In `main()`, create a `Graph` object, add edges with `addEdge()`, display the matrix with `displayMatrix()`, and call `checkDirectedOrNot()`.

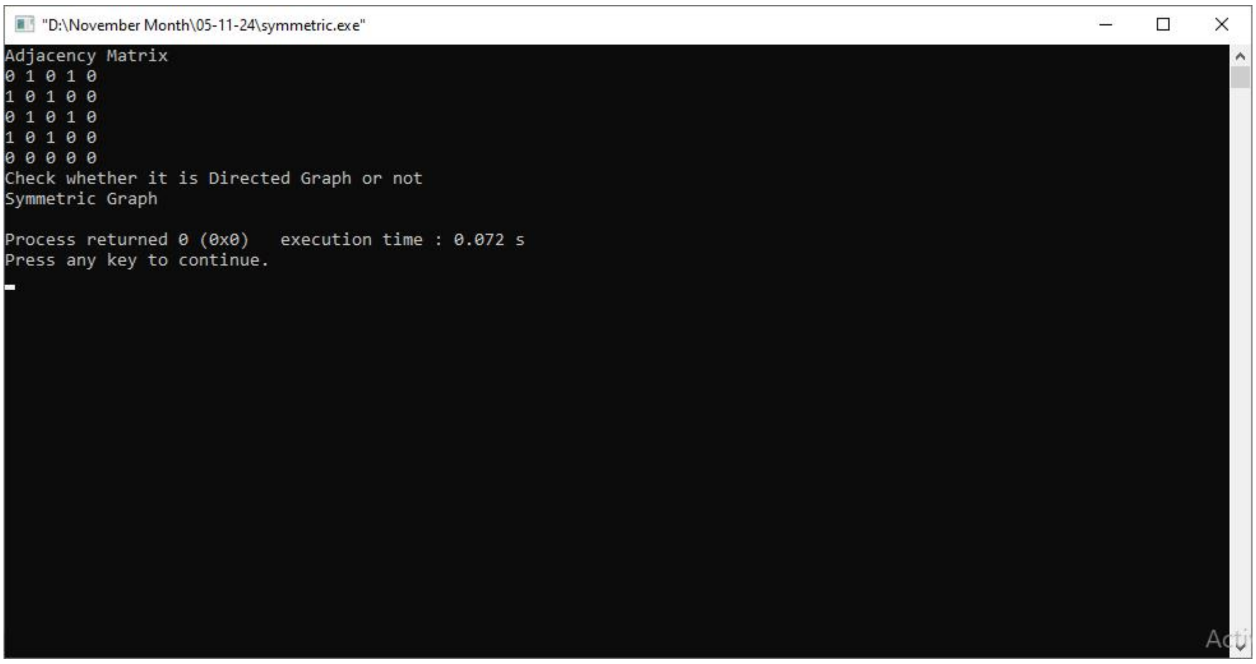
Step 11: Stop the process

Program:

```
#include <iostream>
using namespace std;
class Graph {
   int numvertices;
   int** adjmatrix;
public:
   Graph(int vertices) {
      numvertices = vertices;
      adjmatrix = new int*[vertices];
      for (int i = 0; i < vertices; i++) {
         adjmatrix[i] = new int[vertices];
         for (int j = 0; j < vertices; j++) {
            adjmatrix[i][j] = 0;
         }
      }
   }
   void addEdge(int u,int v) {
      adjmatrix[u][v] = 1;
   }
   void displayMatrix() {
      for (int i = 0; i < numvertices; i++) {
         for (int j = 0; j < numvertices; j++) {
            cout << adjmatrix[i][j] << " ";
         }
         cout << endl;
      }
   }
   void checkDirectedOrNot() {
      int flag = 1;
      for (int i = 0; i < numvertices; i++) {
         for (int j = 0; j < numvertices; j++) {
            if (i != j) {
               if (adjmatrix[i][j] != 0) {
                  if (adjmatrix[j][i] != 1) {
```

```
                            flag = 0;
                        }
                    }
                }
            }
        }
        if (flag == 1) {
                cout << "Symmetric Graph" << endl;
            } else {
                cout << "Non-Symmetric Graph" << endl;
            }
        }
    ~Graph() {
        for (int i = 0; i < numvertices; i++) {
            delete[] adjmatrix[i];
        }
        delete[] adjmatrix;
    }
};
int main() {
    Graph g(5);
    g.addEdge(0, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(3, 2);
    g.addEdge(2, 1);
    g.addEdge(2, 3);
    g.addEdge(1, 0);
    g.addEdge(3, 0);
    cout << "Adjacency Matrix" << endl;
    g.displayMatrix();
    cout << "Check whether it is Directed Graph or not" << endl;
    g.checkDirectedOrNot();
    return 0;
}
```

Output:



"D:\November Month\05-11-24\symmetric.exe"

```
Adjacency Matrix
0 1 0 1 0
1 0 1 0 0
0 1 0 1 0
1 0 1 0 0
0 0 0 0 0
Check whether it is Directed Graph or not
Symmetric Graph

Process returned 0 (0x0)   execution time : 0.072 s
Press any key to continue.
```

26. Weighted graph

Aim

Implement an undirected graph using an adjacency matrix with weights, allowing the addition of edges, displaying matrices, and calculating the sum of weights.

Algorithm

Step 1: Start the process. Define the `Graph` class with vertices and two 2D arrays for adjacency and weighted matrices.

Step 2: Implement the constructor to initialize the number of vertices and allocate memory for matrices, setting values to 0.

Step 3: Create the `addEdge` method to add an edge between two vertices in the adjacency matrix.

Step 4: Create the `weightededge` method to assign a weight to the edge between two vertices in the weighted matrix.

Step 5: Implement the `displayMatrix` method to print the adjacency matrix contents.

Step 6: Implement the `displayweightedMatrix` method to print the weighted matrix contents.

Step 7: Implement the `sumweight` method to calculate and display the sum of weights for edges present in the adjacency matrix.

Step 8: In the `main` function, create a `Graph` instance, add edges, assign weights, and display both matrices.

Step 9: Stop the process.

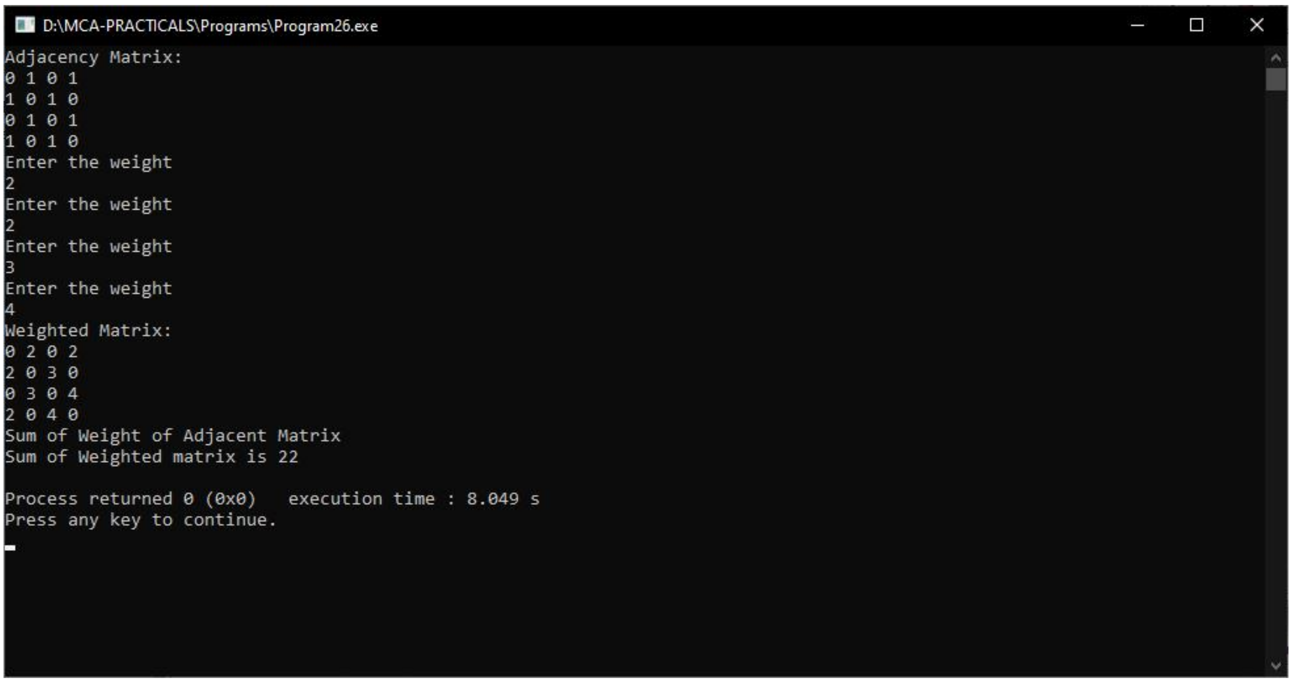
Program:

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

class Graph {
   int numVertices;
   int** adjMatrix;
   int** weightedMatrix;
   int i;
   int j;

public:
   Graph(int vertices) {
      numVertices = vertices;
      adjMatrix = new int*[vertices];
      for (int i = 0; i < vertices; i++) {
         adjMatrix[i] = new int[vertices];
         for (int j = 0; j < vertices; j++)
            adjMatrix[i][j] = 0;
      }
      weightedMatrix = new int*[vertices];
      for (int i = 0; i < vertices; i++) {
         weightedMatrix[i] = new int[vertices];
         for (int j = 0; j < vertices; j++)
            weightedMatrix[i][j] = 0;
      }
   }

   void addEdge(int u, int v) {
      adjMatrix[u][v] = 1;
      adjMatrix[v][u] = 1;
   }

   void weightededge(int u, int v) {
      int a;
```

```
    cout << "Enter the weight" << endl;
    cin >> a;
    weightedMatrix[u][v] = a;
    weightedMatrix[v][u] = a;
}

void displayMatrix() {
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

void displayweightedMatrix() {
    for (int i = 0; i < numVertices; i++) {
        for (int j = 0; j < numVertices; j++) {
            cout << weightedMatrix[i][j] << " ";
        }
        cout << endl;
    }
}

void sumweight(int vertices) {
    int sum = 0;
    for (i = 0; i < vertices; i++) {
        for (j = 0; j < vertices; j++) {
            if (adjMatrix[i][j] == 1) {
                sum = sum + weightedMatrix[i][j];
            }
        }
    }
    cout << "Sum of Weighted matrix is " << sum << endl;
}
```

```
    ~Graph() {
        for (int i = 0; i < numVertices; i++)
            delete[] adjMatrix[i];
        delete[] adjMatrix;
    }
};

int main() {
    Graph g(4);
    g.addEdge(0, 1);
    g.addEdge(0, 3);
    g.addEdge(1, 2);
    g.addEdge(2, 3);
    cout << "Adjacency Matrix:" << endl;
    g.displayMatrix();

    g.weightededge(0, 1);
    g.weightededge(0, 3);
    g.weightededge(1, 2);
    g.weightededge(2, 3);

    cout << "Weighted Matrix:" << endl;
    g.displayweightedMatrix();

    cout << "Sum of Weight of Adjacent Matrix" << endl;
    g.sumweight(4);

    return 0;
}
```

Output:



```
Adjacency Matrix:
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
Enter the weight
2
Enter the weight
2
Enter the weight
3
Enter the weight
4
Weighted Matrix:
0 2 0 2
2 0 3 0
0 3 0 4
2 0 4 0
Sum of Weight of Adjacent Matrix
Sum of Weighted matrix is 22

Process returned 0 (0x0)   execution time : 8.049 s
Press any key to continue.
```

27. Path from source to destination using d/t/l/r

Aim

To navigate through a 3x3 grid using directional commands (U, R, L, D) while calculating the sum of the visited cells' values until reaching the destination at the bottom-right corner.

Algorithm

Step 1: Initialize a 3x3 grid `a[N][N]` with predefined floating-point values.

Step 2: Set starting coordinates `row = 0` and `col = 0` and initialize `sum1 = 0.0` to accumulate the values of visited cells.

Step 3: Display the grid values for reference.

Step 4: Prompt the user to enter a direction command: "U" for up, "R" for right, "L" for left, "D" for down, or "Q" to quit.

Step 5: Check the command and update `row` and `col` accordingly while ensuring boundaries are not crossed:

- Adjust `row` for up and down movements and `col` for left and right.

Step 6: Add the value at the current position `a[row][col]` to `sum1`.

Step 7: Print the current position and cumulative sum after each move.

Step 8: Stop if the destination `(2, 2)` is reached or if the user enters `Q`.

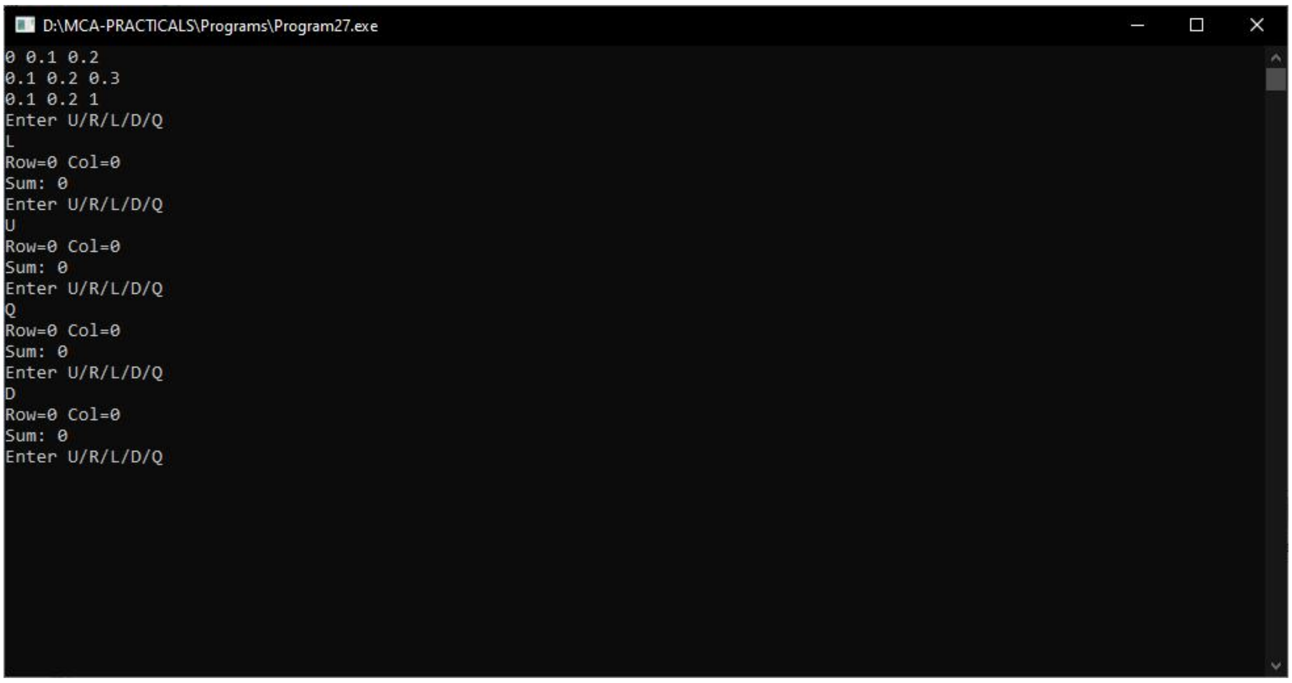
Step 9: Display the final sum and end the program.

Program:

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

class findpath {
public:
    void path() {
        int N = 3;
        float a[N][N] = {
            {0, 0.1, 0.2},
            {0.1, 0.2, 0.3},
            {0.1, 0.2, 1}
        };
        int row = 0, col = 0, i = 1;
        float sum1 = 0.0;
        char s;

        for (int m = 0; m <= 2; m++) {
            for (int b = 0; b <= 2; b++) {
                cout << a[m][b] << " ";
            }
            cout << endl;
        }

        while (i != 0) {
            cout << "Enter U/R/L/D/Q" << endl;
            cin >> s;

            if (s == 'u') {
                row = max(0, row - 1);
            } else if (s == 'r') {
                col = min(2, col + 1);
            } else if (s == 'l') {
                col = max(0, col - 1);
            } else if (s == 'd') {
```

```
            row = min(2, row + 1);
        } else if (s == 'q') {
            i = 0;
        }

        cout << "Row=" << row << " Col=" << col << endl;
        sum1 += a[row][col];
        cout << "Sum: " << sum1 << endl;

        if (row == 2 && col == 2) {
            cout << "Total Sum: " << sum1 << endl;
            i = 0;
        }
      }
    }
};

int main() {
    findpath path1;
    path1.path();
    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program27.exe

```
0 0.1 0.2
0.1 0.2 0.3
0.1 0.2 1
Enter U/R/L/D/Q
L
Row=0 Col=0
Sum: 0
Enter U/R/L/D/Q
U
Row=0 Col=0
Sum: 0
Enter U/R/L/D/Q
Q
Row=0 Col=0
Sum: 0
Enter U/R/L/D/Q
D
Row=0 Col=0
Sum: 0
Enter U/R/L/D/Q
```

28. Find the number of available path from source to destination using 0 as source,1 s destination,-1 as obstacle

Aim

To find and print all possible paths between a source and destination node in an undirected graph using Depth-First Search (DFS).

Algorithm

Step 1: Initialize the graph as an adjacency matrix and specify the source and destination nodes.

Step 2: Define the `findAllPaths` function for DFS, marking nodes as visited and storing the current path.

Step 3: Within `findAllPaths`, mark the source node as visited and add it to the path.

Step 4: If the current source node matches the destination, print the current path as one complete path.

Step 5: If not, loop through adjacent nodes; for each unvisited adjacent node, recursively call `findAllPaths`.

Step 6: After exploring each node, backtrack by marking the node as unvisited and removing it from the path.

Step 7: In the `main` function, initialize a visited array and path vector, then call `findAllPaths` with source and destination.

Step 8: Print all possible paths found between the source and destination.

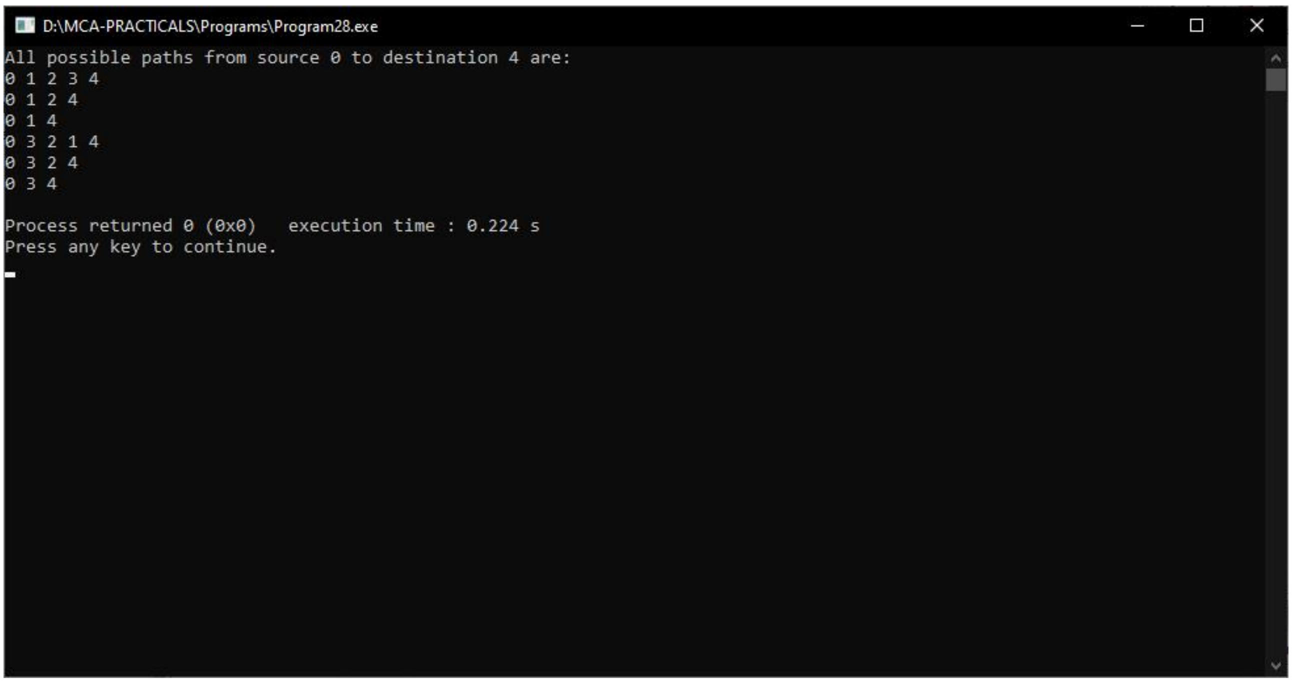
Step 9: End the program once all paths are displayed

Program:

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

void findAllPaths(vector<vector<int>>& graph, int source, int destination, vector<bool>&
visited, vector<int>& path) {
   visited[source] = true;
   path.push_back(source);

   if (source == destination) {
      for (int node : path) cout << node << " ";
      cout << endl;
   } else {
      for (int i = 0; i < graph[source].size(); i++) {
         if (graph[source][i] != 0 && !visited[i]) findAllPaths(graph, i, destination, visited,
path);
      }
   }

   path.pop_back();
   visited[source] = false;
}

int main() {
   int V = 5;
   vector<vector<int>> graph = {
      {0, 1, 0, 1, 0},
      {1, 0, 1, 0, 1},
      {0, 1, 0, 1, 1},
      {1, 0, 1, 0, 1},
      {0, 1, 1, 1, 0}
   };

   int source = 0, destination = 4;
```

```
    vector<bool> visited(V, false);
    vector<int> path;

    cout << "All possible paths from source " << source << " to destination " << destination
<< " are:" << endl;
    findAllPaths(graph, source, destination, visited, path);

    return 0;
}
```

Output:



```
All possible paths from source 0 to destination 4 are:
0 1 2 3 4
0 1 2 4
0 1 4
0 3 2 1 4
0 3 2 4
0 3 4

Process returned 0 (0x0)   execution time : 0.224 s
Press any key to continue.
```

29. Find the number of paths if vector matrix is provided with vertex and edge-mazing problem

Aim

To find all possible paths from the source cell (0, 0) to the destination cell (3, 3) in a given 4x4 matrix, where cells with -1 represent obstacles.

Algorithm

Step 1: Start the process.

Step 2: Define a function `isValid()` to check if a cell is within matrix bounds, is not an obstacle (value -1), and has not been visited.

Step 3: Define a recursive function `findPaths()` that takes the matrix, visited array, current row and column indices, and a reference to `pathCount` to track valid paths.

Step 4: In `findPaths()`, check if the current cell is the destination (3, 3). If it is, increment `pathCount` and return.

Step 5: Mark the current cell as visited in the visited array.

Step 6: Define possible movement directions (down, up, right, left) using arrays for row and column adjustments.

Step 7: Loop through each direction, calculating new row and column indices.

Step 8: For each new position, call `isValid()` to check if moving there is allowed. If valid, make a recursive call to `findPaths()` for the new position.

Step 9: After exploring moves, backtrack by unmarking the current cell in the visited array to allow for alternative paths.

Step 10: In `main()`, initialize the 4x4 matrix and visited array, then call `findPaths()` starting from (0, 0).

Step 11: Print the total number of valid paths from source to destination after all recursive calls complete.

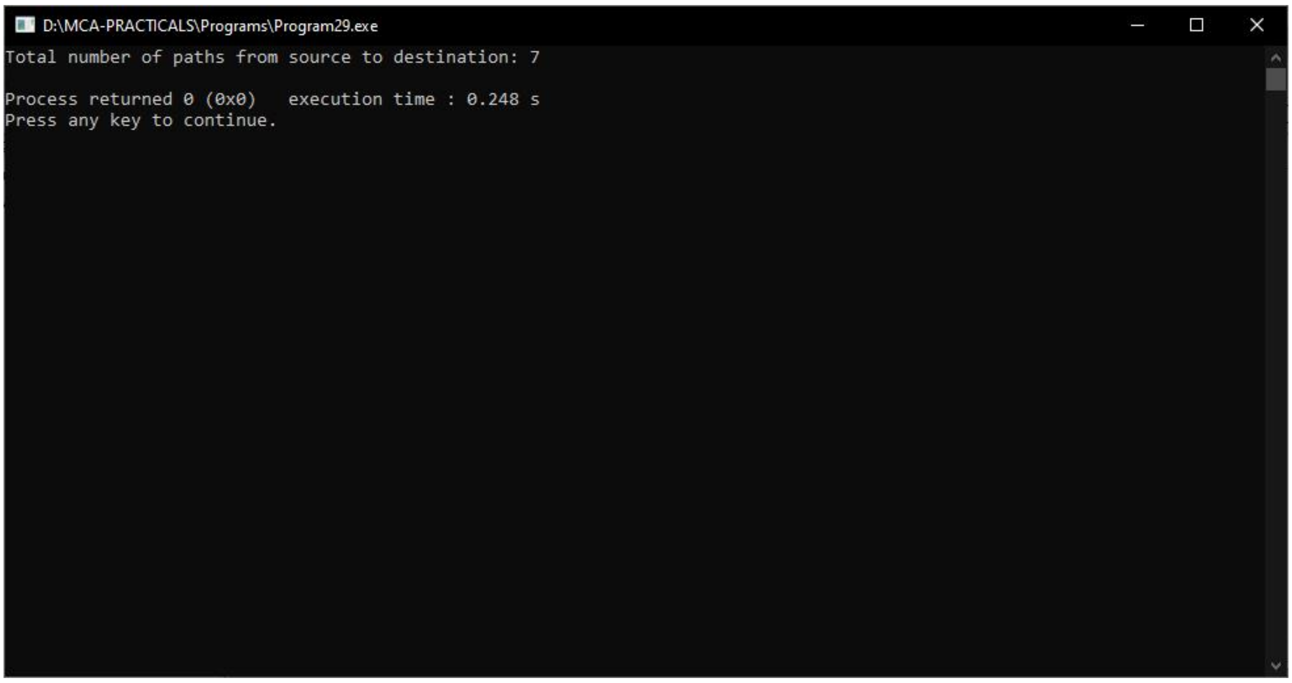
Step 12: Stop the process.

Program:

```
#include <iostream>
using namespace std;
#define N 4

bool isValid(int matrix[N][N], int visited[N][N], int row, int col) {
   return (row >= 0 && row < N && col >= 0 && col < N && matrix[row][col] != -1
&& !visited[row][col]);
}

void findPaths(int matrix[N][N], int visited[N][N], int row, int col, int& pathCount) {
   if (row == N-1 && col == N-1) {
      pathCount++;
      return;
   }
   visited[row][col] = 1;
   int rowDir[] = {1, -1, 0, 0};
   int colDir[] = {0, 0, 1, -1};

   for (int i = 0; i < 4; i++) {
      int newRow = row + rowDir[i];
      int newCol = col + colDir[i];
      if (isValid(matrix, visited, newRow, newCol)) {
         findPaths(matrix, visited, newRow, newCol, pathCount);
      }
   }
   visited[row][col] = 0;
}

int main() {
   int matrix[N][N] = {
      {0, 0, 0, 0},
      {0, -1, -1, 0},
      {0, 0, 0, 0},
      {-1, 0, 0, 1}
```

```
    };
    int visited[N][N] = {0};
    int pathCount = 0;

    findPaths(matrix, visited, 0, 0, pathCount);

    cout << "Total number of paths from source to destination: " << pathCount << endl;
    return 0;

}
```

Output:



D:\MCA-PRACTICALS\Programs\Program29.exe

```
Total number of paths from source to destination: 7

Process returned 0 (0x0)   execution time : 0.248 s
Press any key to continue.
```

30. Find the shortest path if matrix is provided with distance values

Aim

To find the shortest path from a source vertex to all other vertices in a weighted graph using Dijkstra's algorithm.

Algorithm

Step 1: Initialize an array `dist[]` to store the shortest path estimates from the source, setting each to infinity except the source itself, which is set to 0.

Step 2: Maintain a boolean array `sptSet[]` to track vertices that are included in the shortest path tree, initializing all to false.

Step 3: Iterate over all vertices to build the shortest path tree.

Step 4: Use `minDistance()` to select the vertex `u` with the minimum distance estimate that is not yet processed.

Step 5: Mark `u` as processed by setting `sptSet[u] = true`.

Step 6: Update the distance estimates of adjacent vertices to `u`. For each adjacent vertex `v`, if `v` is not in `sptSet` and there's a shorter path from `src` to `v` via `u`, update `dist[v]`.

Step 7: Repeat steps 4–6 until all vertices are processed.

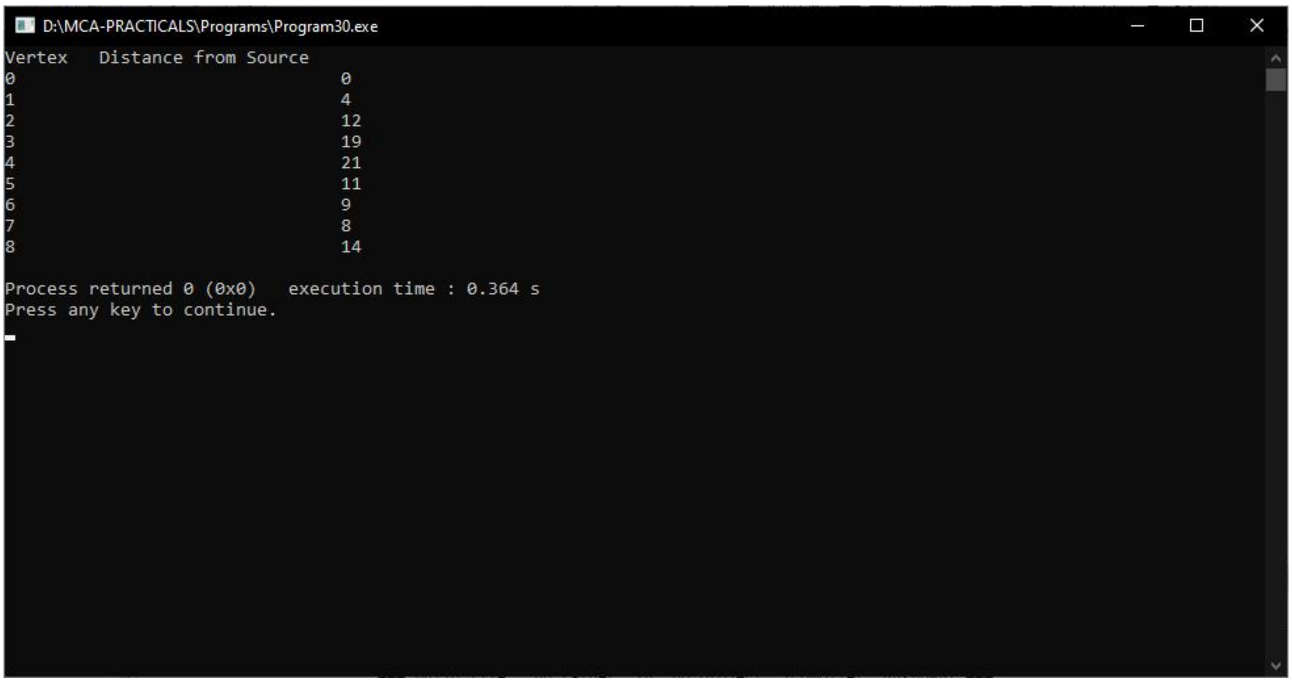
Step 8: Print the distances from the source to all vertices in the graph.

Step 9: End the program after displaying the output.

Program:

```
#include <iostream>
#include <limits.h>
using namespace std;
#define V 9

int minDistance(int dist[], bool sptSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (!sptSet[v] && dist[v] <= min)
         min = dist[v], min_index = v;
   return min_index;
}

void printSolution(int dist[]) {
   cout << "Vertex \t Distance from Source" << endl;
   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);
}

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:



D:\MCA-PRACTICALS\Programs\Program30.exe

```
Vertex   Distance from Source
0                                0
1                                4
2                                12
3                                19
4                                21
5                                11
6                                9
7                                8
8                                14

Process returned 0 (0x0)   execution time : 0.364 s
Press any key to continue.
```

31. Sequential file organisation

Aim

To create a simple record management system in C++ that allows users to insert, delete, modify, and display records consisting of an ID and a name.

Algorithm

Step 1: Start the process.

Step 2: Define a `Record` structure to store the ID and name of each record.

Step 3: Implement the `displayRecords()` function to show all existing records in a tabular format, checking if the list is empty and notifying the user if no records exist.

Step 4: Implement the `insertRecord()` function to add a new record by taking an ID and a name as parameters, creating a new `Record`, and adding it to the records vector.

Step 5: Implement the `deleteRecord()` function to remove a record by ID by iterating through the vector, finding a matching ID, and erasing the record if found.

Step 6: Implement the `modifyRecord()` function to update the name of a record based on its ID, locating the record, and modifying its name if the ID matches.

Step 7: In `main()`, initialize a vector of `Record` to store records, and set up a menu using a do-while loop to allow user interaction.

Step 8: Use a switch-case statement to handle user input for different operations (insert, delete, modify, display, exit), prompting for input and calling the relevant function for each case.

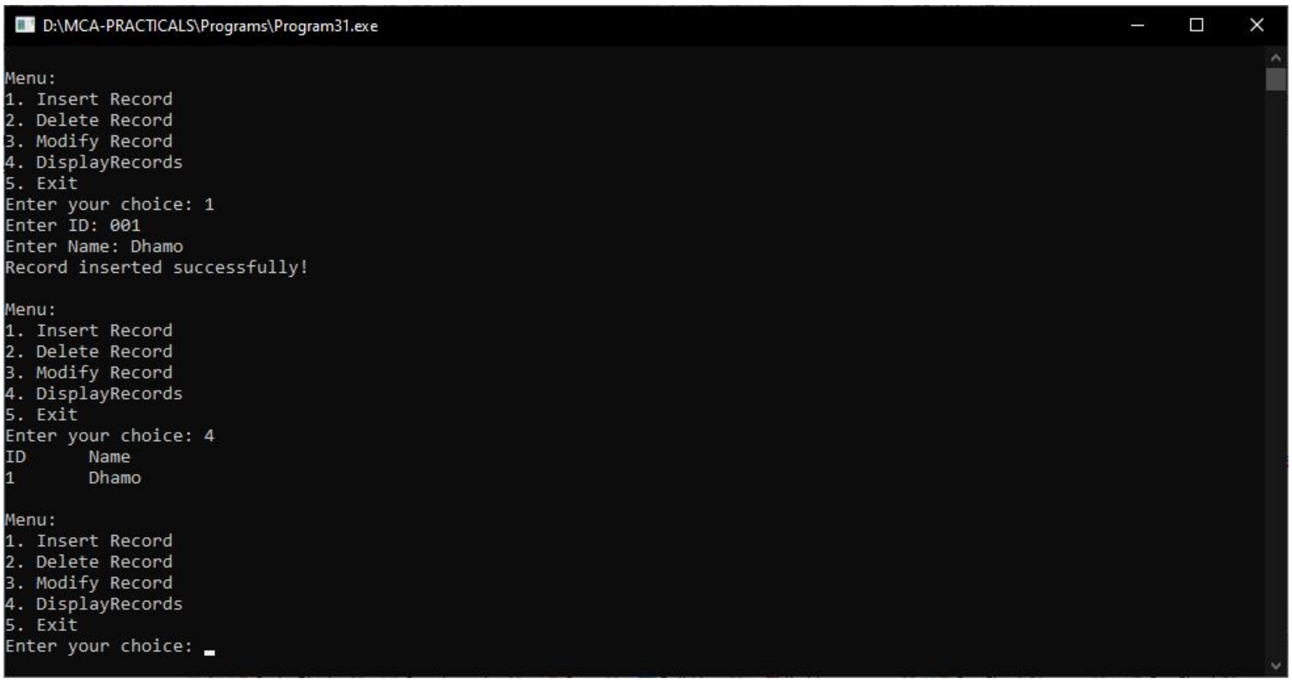
Step 9: Continue the loop until the user selects exit, at which point display an exit message.

Step 10: Stop the process.

Program:

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

struct Record {
    int id;
    string name;
};

void displayRecords(const vector<Record>& records) {
    if (records.empty()) {
        cout << "No records to display!" << endl;
        return;
    }
    cout << "ID\tName" << endl;
    for (const auto& record : records) {
        cout << record.id << "\t" << record.name << endl;
    }
}

void insertRecord(vector<Record>& records, int id, const string& name) {
    records.push_back({id, name});
    cout << "Record inserted successfully!" << endl;
}

void deleteRecord(vector<Record>& records, int id) {
    bool found = false;
    for (auto it = records.begin(); it != records.end(); ++it) {
        if (it->id == id) {
            records.erase(it);
            found = true;
            cout << "Record with ID " << id << " deleted successfully!" << endl;
            break;
        }
    }
    if (!found) cout << "Record with ID " << id << " not found!" << endl;
}

void modifyRecord(vector<Record>& records, int id, const string& newName) {
    bool found = false;
    for (auto& record : records) {
        if (record.id == id) {
            record.name = newName;
            found = true;
            cout << "Record with ID " << id << " modified successfully!" << endl;
            break;
        }
    }
    if (!found) cout << "Record with ID " << id << " not found!" << endl;
}

int main() {
    vector<Record> records;
    int choice, id;
    string name;

    do {
        cout << "\nMenu:\n1. Insert Record\n2. Delete Record\n3. Modify Record\n4. Display
Records\n5. Exit\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter Name: ";
                cin.ignore();
                getline(cin, name);
                insertRecord(records, id, name);
                break;
            case 2:
                cout << "Enter ID to delete: ";
                cin >> id;
                deleteRecord(records, id);
                break;
            case 3:
                cout << "Enter ID to modify: ";
                cin >> id;
                cout << "Enter new name: ";
                cin.ignore();
                getline(cin, name);
                modifyRecord(records, id, name);
                break;
            case 4:
                displayRecords(records);
                break;
            case 5:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 5);

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program31.exe

```
Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 1
Enter ID: 001
Enter Name: Dhamo
Record inserted successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 4
ID      Name
1       Dhamo

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: _
```

32. Indexed sequential file organisation

Aim

To develop an indexed sequential data management system allowing users to insert, delete, modify, and display records efficiently by using an unordered map for fast indexing.

Algorithm

Step 1: Start the process.

Step 2: Define a `Record` structure with fields for `id` and `name`.

Step 3: Create a class `IndexedSequentialData` to manage records using a vector for storage and an unordered map as an index for efficient lookup.

Step 4: Implement the `insertRecord()` function to add a new record to the vector and update the index map. If the record ID exists, display an error message.

Step 5: Implement the `deleteRecord()` function to remove a record by ID. If the ID exists, delete the record from the vector and update the index map to reflect new positions.

Step 6: Implement the `modifyRecord()` function to update a record's name by ID. If the ID exists, update the name; otherwise, display an error message.

Step 7: Implement `displayRecords()` to output all records or notify the user if the list is empty.

Step 8: In `main()`, initialize `IndexedSequentialData` and prompt the user with a menu for inserting, deleting, modifying, displaying records, or exiting.

Step 9: Handle user input using a `do-while` loop with a switch-case statement to call the appropriate function based on the user's choice. Prompt the user for ID and name where applicable.

Step 10: Continue until the user selects the exit option, then display an exit message.

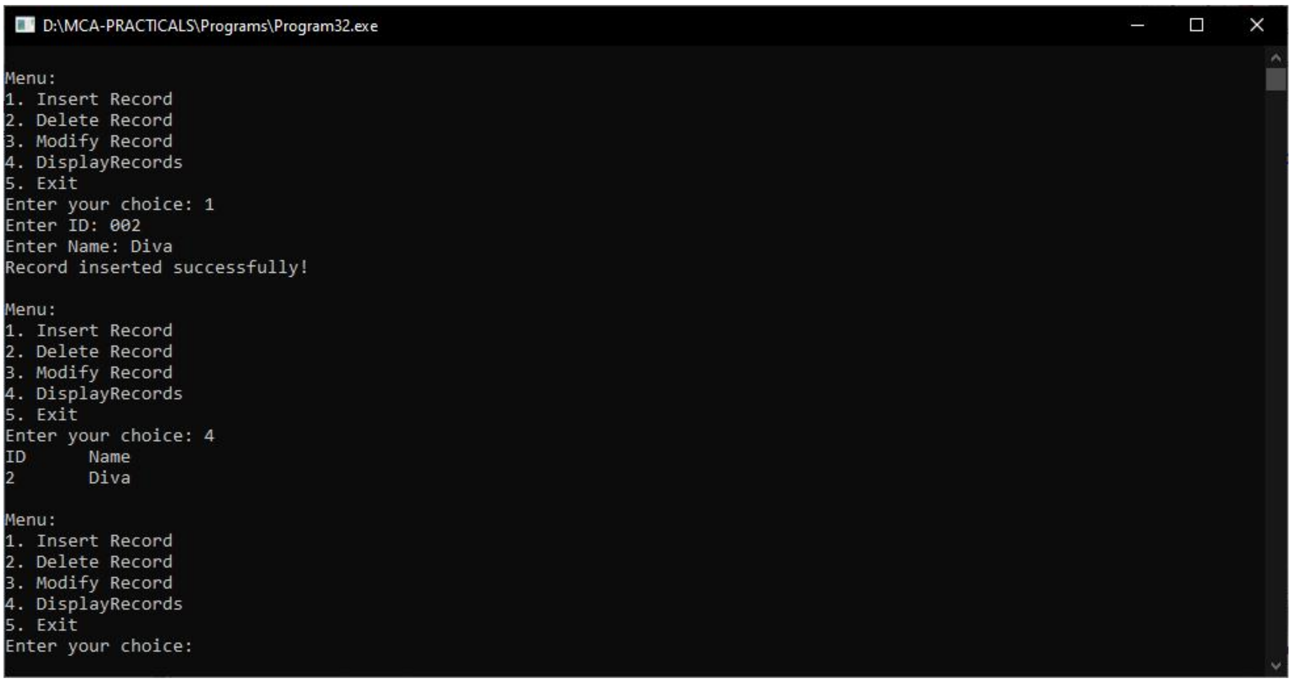
Step 11: Stop the process.

Program:

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

struct Record {
   int id;
   string name;
};

class IndexedSequentialData {
   vector<Record> records;
   unordered_map<int, int> index;

public:
   void displayRecords() {
      if (records.empty()) {
         cout << "No records to display!" << endl;
         return;
      }
      cout << "ID\tName" << endl;
      for (const auto& record : records) {
         cout << record.id << "\t" << record.name << endl;
      }
   }

   void insertRecord(int id, const string& name) {
      if (index.find(id) != index.end()) {
         cout << "Record with ID " << id << " already exists!" << endl;
         return;
      }
      records.push_back({id, name});
      index[id] = records.size() - 1;
```

```
        cout << "Record inserted successfully!" << endl;
    }

    void deleteRecord(int id) {
        if (index.find(id) == index.end()) {
            cout << "Record with ID " << id << " not found!" << endl;
            return;
        }
        int pos = index[id];
        records.erase(records.begin() + pos);
        index.erase(id);
        for (int i = pos; i < records.size(); i++) {
            index[records[i].id] = i;
        }
        cout << "Record with ID " << id << " deleted successfully!" << endl;
    }
    void modifyRecord(int id, const string& newName) {
        if (index.find(id) == index.end()) {
            cout << "Record with ID " << id << " not found!" << endl;
            return;
        }
        records[index[id]].name = newName;
        cout << "Record with ID " << id << " modified successfully!" << endl;
    }
};
int main() {
    IndexedSequentialData data;
    int choice, id;
    string name;

    do {
        cout << "\nMenu:\n1. Insert Record\n2. Delete Record\n3. Modify Record\n4. Display
Records\n5. Exit\nEnter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter Name: ";
                cin.ignore();
                getline(cin, name);
                data.insertRecord(id, name);
                break;
            case 2:
                cout << "Enter ID to delete: ";
                cin >> id;
                data.deleteRecord(id);
                break;
            case 3:
                cout << "Enter ID to modify: ";
                cin >> id;
                cout << "Enter new name: ";
                cin.ignore();
                getline(cin, name);
                data.modifyRecord(id, name);
                break;
            case 4:
                data.displayRecords();
                break;
            case 5:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 5);

    return 0;
}
```

Output:



```
Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 1
Enter ID: 002
Enter Name: Diva
Record inserted successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 4
ID      Name
2       Diva

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice:
```

33. Random file organisation

Aim

To simulate a random file organization system using a hash table, allowing for efficient insertion, deletion, modification, and display of records based on unique record IDs.

Algorithm

Step 1: Start the process.

Step 2: Define a `Record` structure with fields for `id` and `name`.

Step 3: Create a class `RandomFileOrganization` that uses an unordered map (`unordered_map<int, Record>`) to simulate direct file access with fast lookups by `id`.

Step 4: Implement the `insertRecord()` function to add a new record to the map. If the `id` already exists, display an error message.

Step 5: Implement the `deleteRecord()` function to remove a record by `id`. If the `id` is found, delete the record from the map and display a success message. Otherwise, display an error message.

Step 6: Implement the `modifyRecord()` function to update a record's `name` by `id`. If the `id` is found, update the `name`; otherwise, display an error message.

Step 7: Implement `displayRecords()` to output all records if the map is not empty. If there are no records, display a message indicating so.

Step 8: In `main()`, initialize `RandomFileOrganization` and prompt the user with a menu for inserting, deleting, modifying, displaying records, or exiting.

Step 9: Handle user input using a `do-while` loop with a switch-case statement to call the appropriate function based on the user's choice. Prompt the user for `id` and `name` where applicable.

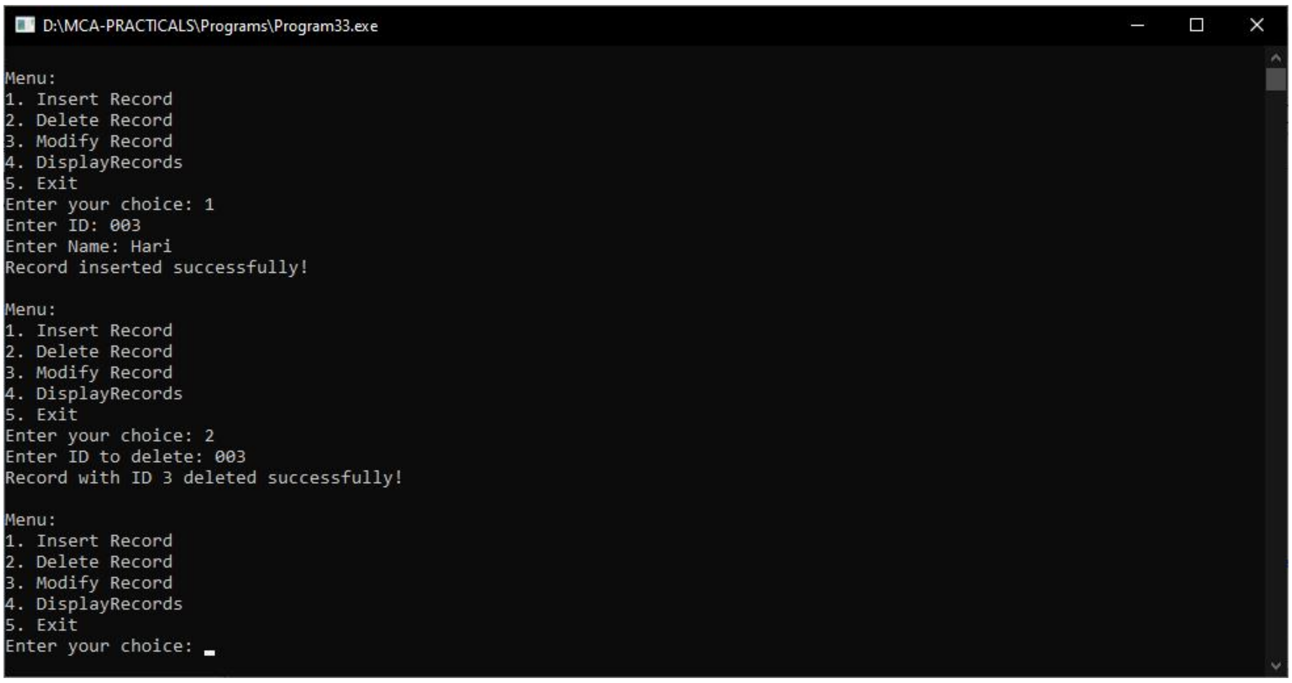
Step 10: Continue until the user selects the exit option, then display an exit message.

Step 11: Stop the process.

Program:

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

struct Record {
    int id;
    string name;
};

class RandomFileOrganization {
    unordered_map<int, Record> records;

public:
    void insertRecord(int id, const string& name) {
        if (records.find(id) != records.end()) {
            cout << "Record with ID " << id << " already exists!" << endl;
            return;
        }
        records[id] = {id, name};
        cout << "Record inserted successfully!" << endl;
    }

    void deleteRecord(int id) {
        if (records.find(id) == records.end()) {
            cout << "Record with ID " << id << " not found!" << endl;
            return;
        }
        records.erase(id);
        cout << "Record with ID " << id << " deleted successfully!" << endl;
    }

    void modifyRecord(int id, const string& newName) {
        if (records.find(id) == records.end()) {
            cout << "Record with ID " << id << " not found!" << endl;
            return;
        }
        records[id].name = newName;
        cout << "Record with ID " << id << " modified successfully!" << endl;
    }

    void displayRecords() {
        if (records.empty()) {
            cout << "No records to display!" << endl;
            return;
        }
        cout << "ID\tName" << endl;
        for (const auto& record : records) {
            cout << record.second.id << "\t" << record.second.name << endl;
        }
    }
};

int main() {
    RandomFileOrganization data;
    int choice, id;
    string name;

    do {
        cout << "\nMenu:\n1. Insert Record\n2. Delete Record\n3. Modify Record\n4. Display
Records\n5. Exit\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter Name: ";
                cin.ignore();
                getline(cin, name);
                data.insertRecord(id, name);
                break;
            case 2:
                cout << "Enter ID to delete: ";
                cin >> id;
                data.deleteRecord(id);
                break;
            case 3:
                cout << "Enter ID to modify: ";
                cin >> id;
                cout << "Enter new name: ";
                cin.ignore();
                getline(cin, name);
                data.modifyRecord(id, name);
                break;
            case 4:
                data.displayRecords();
                break;
            case 5:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 5);

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program33.exe

```

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 1
Enter ID: 003
Enter Name: Hari
Record inserted successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 2
Enter ID to delete: 003
Record with ID 3 deleted successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: _
```

34. Linked list file organisation

Aim

To implement a linked list to manage records using a linked organization, allowing sequential storage of records with insertion and display operations.

Algorithm

Step 1: Start the process.

Step 2: Define a `Record` structure with fields `id`, `name`, and a pointer `next` pointing to the next record in the list.

Step 3: Define the `insertRecord` function to insert a new record at the end of the list. Create a new node, set its `id` and `name` fields, and if the list is empty, make this new record the head. Otherwise, traverse to the last node and set its `next` pointer to the new record.

Step 4: Define the `displayRecords` function to traverse the list from head to end, displaying each record's `id` and `name`.

Step 5: In `main()`, initialize the head of the list to `nullptr`.

Step 6: Use `insertRecord` to add records with unique IDs and names.

Step 7: Display all records in the list by calling `displayRecords`.

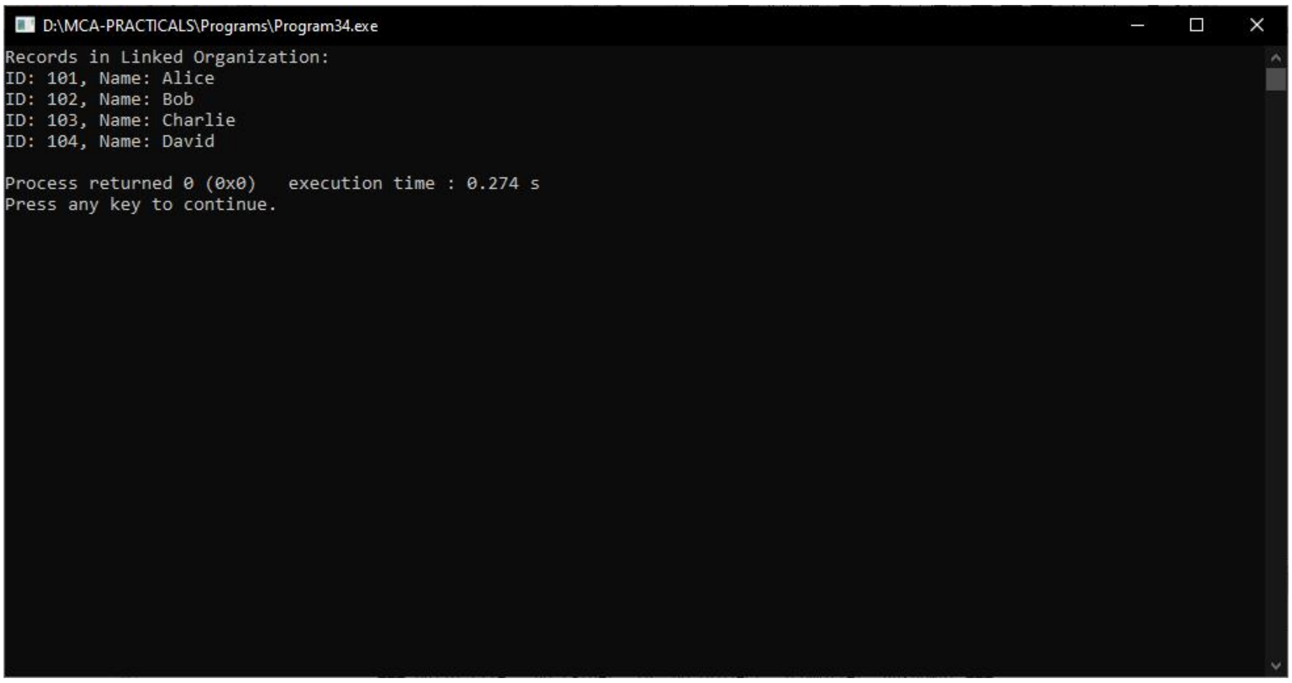
Step 8: Stop the process.

Program:

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

struct Record {
    int id;
    string name;
    Record* next;
};

void insertRecord(Record*& head, int id, string name) {
    Record* newRecord = new Record();
    newRecord->id = id;
    newRecord->name = name;
    newRecord->next = nullptr;

    if (head == nullptr) {
        head = newRecord;
    } else {
        Record* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newRecord;
    }
}

void displayRecords(Record* head) {
    Record* temp = head;
    while (temp != nullptr) {
        cout << "ID: " << temp->id << ", Name: " << temp->name << endl;
        temp = temp->next;
    }
}
```

```
int main() {
    Record* head = nullptr;

    insertRecord(head, 101, "Alice");
    insertRecord(head, 102, "Bob");
    insertRecord(head, 103, "Charlie");
    insertRecord(head, 104, "David");

    cout << "Records in Linked Organization:" << endl;
    displayRecords(head);

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program34.exe

```
Records in Linked Organization:
ID: 101, Name: Alice
ID: 102, Name: Bob
ID: 103, Name: Charlie
ID: 104, Name: David

Process returned 0 (0x0)   execution time : 0.274 s
Press any key to continue.
```

35. Cellular partition

Aim

To implement a cellular partition data organization technique that divides records into multiple partitions (cells) for efficient operations based on record IDs.

Algorithm

Step 1: Start the process.

Step 2: Initialize the partition count and create storage for each partition.

Step 3: Use a hash function to map a record's ID to its partition.

Step 4: Insert records by calculating their partition using the hash function and storing them in the corresponding partition.

Step 5: Delete records by locating their partition, finding the record by ID, and removing it if found.

Step 6: Modify records by locating their partition, finding the record by ID, and updating its name if found.

Step 7: Display records by iterating through each partition and printing the stored records.

Step 8: Prompt the user for operations (insert, delete, modify, display) and execute until the user chooses to exit.

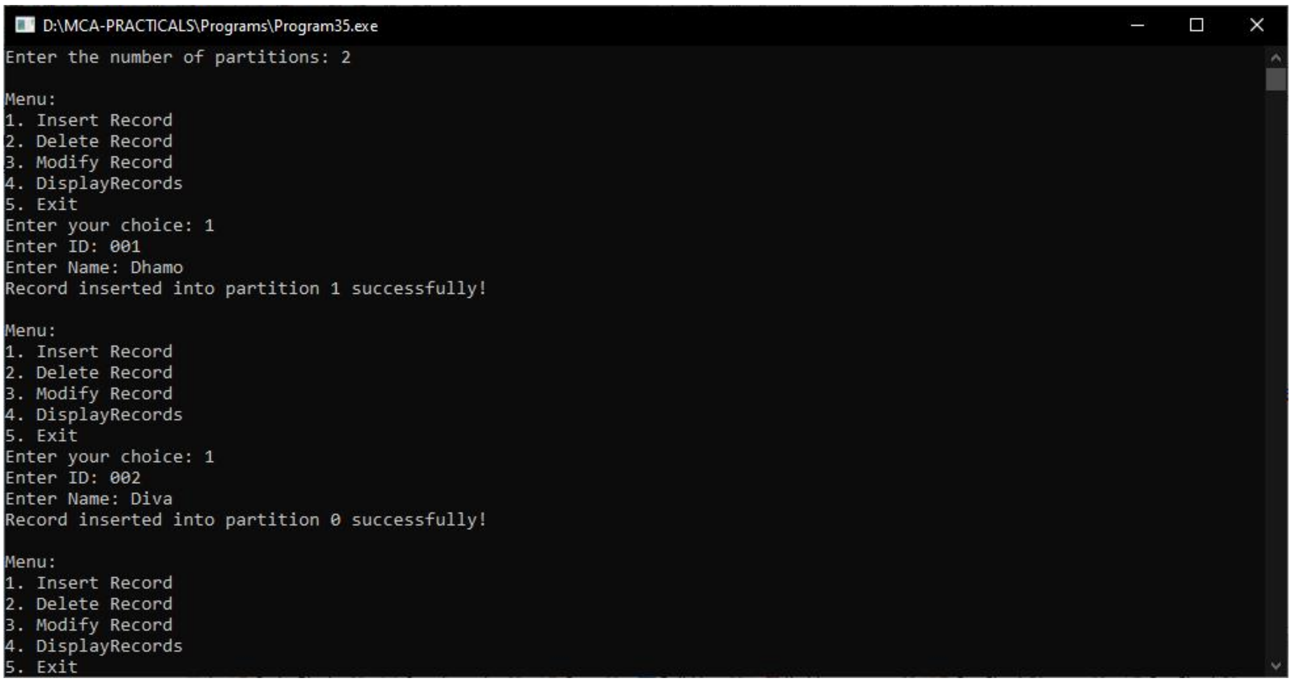
Step 9: Stop the process.

Program:

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

struct Record {
    int id;
    string name;
};

class CellularPartition {
    vector<vector<Record>> partitions;
    int partitionCount;

public:
    CellularPartition(int partitionCount) : partitionCount(partitionCount) {
        partitions.resize(partitionCount);
    }

    int hashFunction(int id) {
        return id % partitionCount;
    }

    void insertRecord(int id, const string& name) {
        int partitionIndex = hashFunction(id);
        Record newRecord = {id, name};
        partitions[partitionIndex].push_back(newRecord);
        cout << "Record inserted into partition " << partitionIndex << " successfully!" << endl;
    }

    void deleteRecord(int id) {
        int partitionIndex = hashFunction(id);
        bool found = false;
        for (auto it = partitions[partitionIndex].begin(); it != partitions[partitionIndex].end();
++it) {
            if (it->id == id) {
                partitions[partitionIndex].erase(it);
                cout << "Record with ID " << id << " deleted from partition " << partitionIndex
<< " successfully!" << endl;
                found = true;
                break;
            }
        }
        if (!found) cout << "Record with ID " << id << " not found in partition " <<
partitionIndex << "!" << endl;
    }

    void modifyRecord(int id, const string& newName) {
        int partitionIndex = hashFunction(id);
        bool found = false;
        for (auto& record : partitions[partitionIndex]) {
            if (record.id == id) {
                record.name = newName;
                cout << "Record with ID " << id << " modified in partition " << partitionIndex <<
" successfully!" << endl;
                found = true;
                break;
            }
        }
        if (!found) cout << "Record with ID " << id << " not found in partition " <<
partitionIndex << "!" << endl;
    }

    void displayRecords() {
        for (int i = 0; i < partitionCount; ++i) {
            if (!partitions[i].empty()) {
                cout << "Partition " << i << " records:\nID\tName" << endl;
                for (const auto& record : partitions[i]) {
```

```
                cout << record.id << "\t" << record.name << endl;
            }
        } else {
            cout << "Partition " << i << " is empty." << endl;
        }
    }
  }
};

int main() {
    int partitionCount;
    cout << "Enter the number of partitions: ";
    cin >> partitionCount;

    CellularPartition data(partitionCount);
    int choice, id;
    string name;

    do {
        cout << "\nMenu:\n1. Insert Record\n2. Delete Record\n3. Modify Record\n4. Display
Records\n5. Exit\nEnter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter ID: ";
                cin >> id;
                cout << "Enter Name: ";
                cin.ignore();
                getline(cin, name);
                data.insertRecord(id, name);
                break;
            case 2:
                cout << "Enter ID to delete: ";
                cin >> id;
                data.deleteRecord(id);
                break;
            case 3:
                cout << "Enter ID to modify: ";
                cin >> id;
                cout << "Enter new name: ";
                cin.ignore();
                getline(cin, name);
                data.modifyRecord(id, name);
                break;
            case 4:
                data.displayRecords();
                break;
            case 5:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice != 5);

    return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program35.exe

```
Enter the number of partitions: 2

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 1
Enter ID: 001
Enter Name: Dhamo
Record inserted into partition 1 successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
Enter your choice: 1
Enter ID: 002
Enter Name: Diva
Record inserted into partition 0 successfully!

Menu:
1. Insert Record
2. Delete Record
3. Modify Record
4. DisplayRecords
5. Exit
```

36. Mazing problem

Aim

To find all possible paths in a maze represented as a grid, from the top-left corner to the bottom-right corner using Depth-First Search (DFS) while avoiding obstacles and ensuring valid moves.

Algorithm

Step 1: Start the process.

Step 2: Define the maze as a 2D grid where `1` represents a valid cell and `0` represents an obstacle.

Step 3: Create a function `isValid` to check if a cell is within the maze boundaries and is valid for movement.

Step 4: Define the movement directions (down, left, right, up) using two arrays for row and column changes.

Step 5: Implement the `findPath` function to recursively explore valid moves and track paths; if reaching the destination, save the path.

Step 6: In the `main` function, initialize the maze and call `findPath` if the start and end cells are valid.

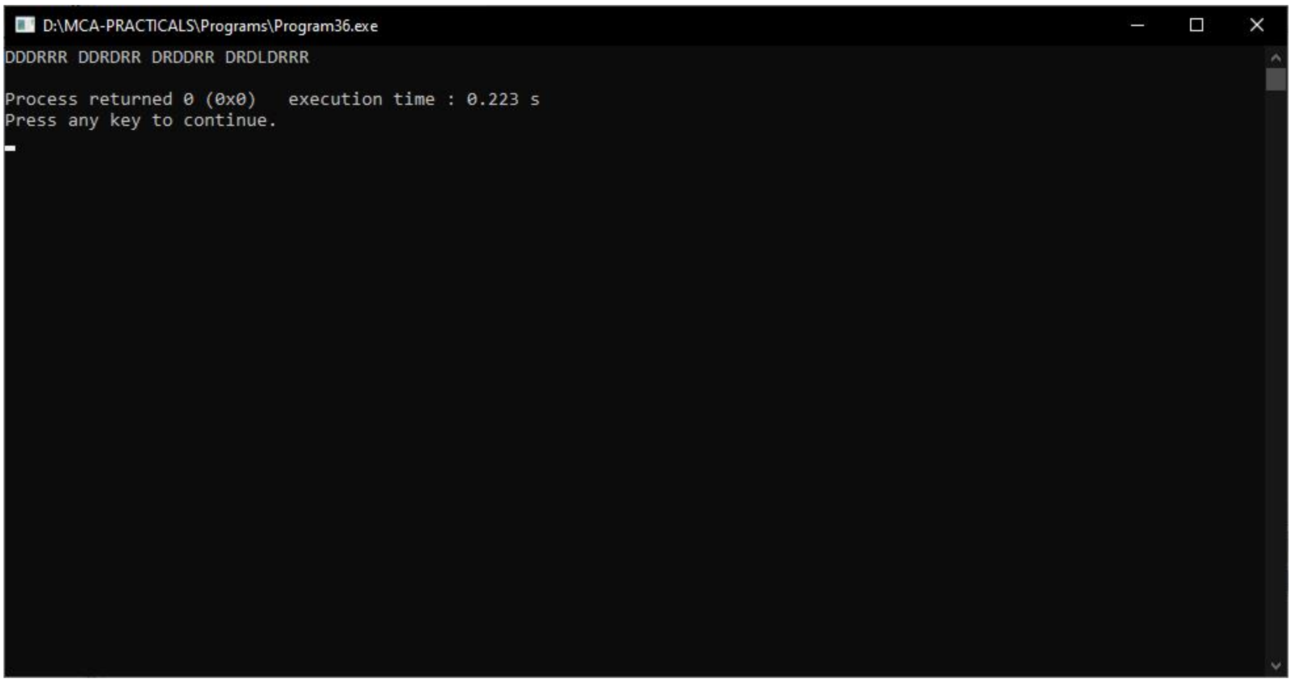
Step 7: If no paths are found, output `-1`; otherwise, print all the found paths.

Step 8: Stop the process.

Program:

```
#include <bits/stdc++.h>
using namespace std;

string direction = "DLRU";
int dr[4] = {1, 0, 0, -1};
int dc[4] = {0, -1, 1, 0};

bool isValid(int row, int col, int n, vector<vector<int>>& maze) {
   return row >= 0 && col >= 0 && row < n && col < n && maze[row][col];
}

void findPath(int row, int col, vector<vector<int>>& maze, int n, vector<string>& ans,
string& currentPath) {
   if (row == n - 1 && col == n - 1) {
      ans.push_back(currentPath);
      return;
   }
   maze[row][col] = 0;
   for (int i = 0; i < 4; i++) {
      int nextrow = row + dr[i];
      int nextcol = col + dc[i];
      if (isValid(nextrow, nextcol, n, maze)) {
         currentPath += direction[i];
         findPath(nextrow, nextcol, maze, n, ans, currentPath);
         currentPath.pop_back();
      }
   }
   maze[row][col] = 1;
}

int main() {
   vector<vector<int>> maze = {
      {1, 0, 0, 0},
      {1, 1, 0, 1},
       {1, 1, 0, 0},
       {1, 1, 1, 1}
   };
   int n = maze.size();
   vector<string> result;
   string currentPath = "";

   if (maze[0][0] && maze[n - 1][n - 1]) {
      findPath(0, 0, maze, n, result, currentPath);
   }

   if (result.empty()) cout << -1;
   else for (const auto& path : result) cout << path << " ";
   cout << endl;

   return 0;
}
```

Output:



D:\MCA-PRACTICALS\Programs\Program36.exe

```
DDDRRR DDRDRR DRDDRR DRDLDRRR

Process returned 0 (0x0)   execution time : 0.223 s
Press any key to continue.
```