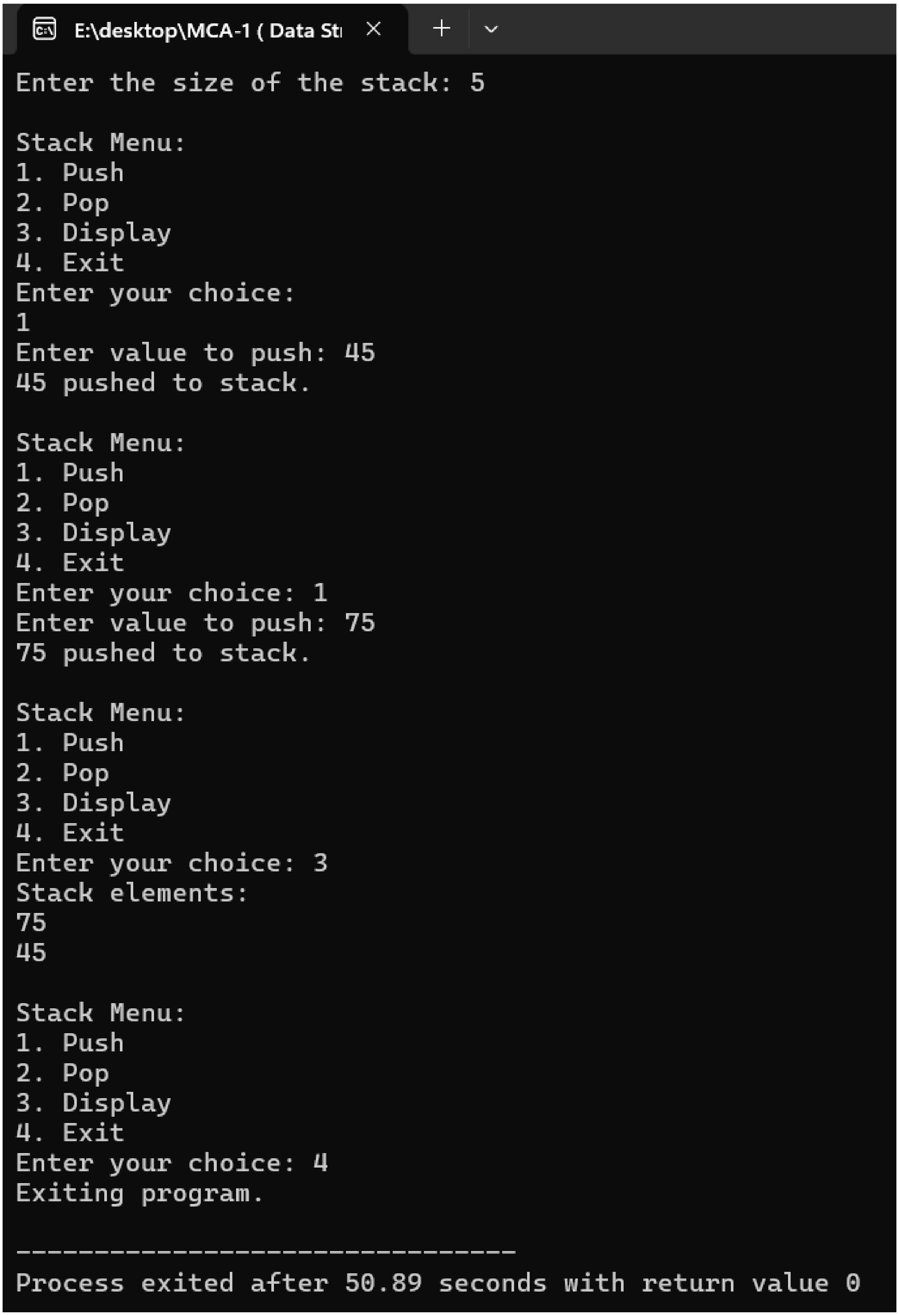
1. Write a C++ program to implement the Stack ADT using an Array.

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

// Define a Stack class
class Stack {
private:
    int top;    // Index of the top element in the stack
    int *arr;   // Array to store stack elements

public:
    // Constructor to initialize the stack with a given size
    Stack(int size) {
        top = -1;                   // Initialize top to -1 (empty stack)
        arr = new int[size];        // Allocate memory for the stack array
    }

    // Destructor to free the memory allocated for the stack array
    ~Stack() {
        delete[] arr;
    }

    // Check if the stack is empty
    bool isEmpty() {
        return (top == -1);
    }

    // Check if the stack is full
    bool isFull(int size) {
        return (top == size - 1);
    }

    // Push an element onto the stack
    void push(int value, int size) {
        if (isFull(size)) {
            cout << "Stack Overflow! Cannot push more elements.\n";
            return;
        }
        arr[++top] = value;
        cout << value << " pushed to stack.\n";
    }

    // Pop the top element from the stack
    void pop() {
        if (isEmpty()) {
            cout << "Stack Underflow! Cannot pop from an empty stack.\n";
            return;
        }
        cout << arr[top--] << " popped from stack.\n";
    }

    // Display all elements in the stack
    void display() {
        if (isEmpty()) {
            cout << "Stack is empty.\n";
```

```cpp
            return;
        }
        cout << "Stack elements:\n";
        for (int i = top; i >= 0; i--) {
            cout << arr[i] << endl;
        }
    }
};

// Main function to test the Stack class
int main() {
    int size;
    cout << "Enter the size of the stack: ";
    cin >> size;

    // Create a stack object with the specified size
    Stack stack(size);
    int choice, value;

    // Menu-driven program to interact with the stack
    do {
        cout << "\nStack Menu:\n";
        cout << "1. Push\n";
        cout << "2. Pop\n";
        cout << "3. Display\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1:
            cout << "Enter value to push: ";
            cin >> value;
            stack.push(value, size);
            break;
        case 2:
            stack.pop();
            break;
        case 3:
            stack.display();
            break;
        case 4:
            cout << "Exiting program.\n";
            break;
        default:
            cout << "Invalid choice. Please try again.\n";
            break;
        }
    } while (choice != 4);

    return 0;
}
```

Output:

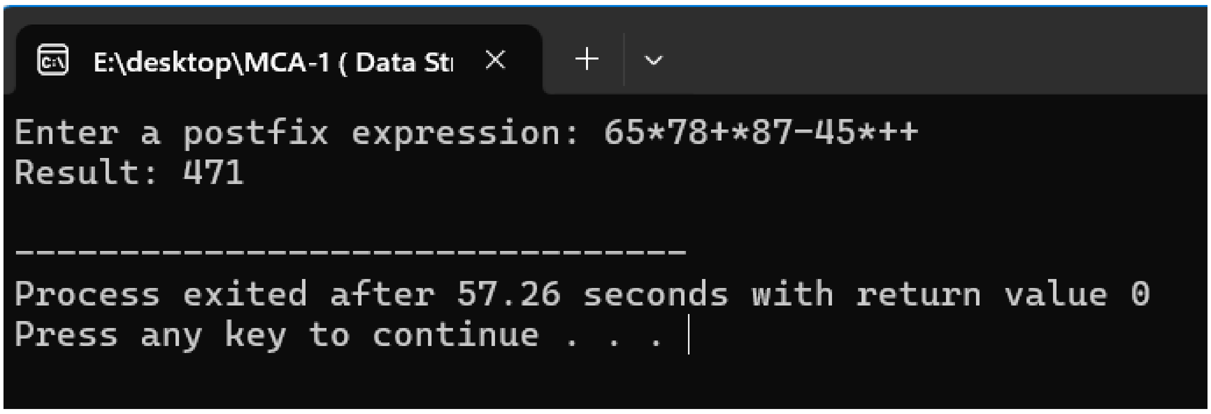


```
Enter the size of the stack: 5

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice:
1
Enter value to push: 45
45 pushed to stack.

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 75
75 pushed to stack.

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements:
75
45

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting program.

--------------------------------
Process exited after 50.89 seconds with return value 0
```

2. Write a C++ program to evaluate postfix evaluation.

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

// Function to evaluate a postfix expression
int evaluatePostfix(string expr) {
    stack<int> stk;

    // Iterate over the expression
    for (size_t i = 0; i < expr.size(); i++) {
        char c = expr[i];
        // If the character is a digit, push it onto the stack
        if (isdigit(c)) {
            stk.push(c - '0');
        } else {            // If the character is an operator, pop two operands from the stack,perform
the operation, and push the result back onto the stack
            int operand2 = stk.top();
            stk.pop();
            int operand1 = stk.top();
            stk.pop();
            switch (c) {
                case '+':
                    stk.push(operand1 + operand2);
                    break;
                case '-':
                    stk.push(operand1 - operand2);
                    break;
                case '*':
                    stk.push(operand1 * operand2);
                    break;
                case '/':
                    stk.push(operand1 / operand2);
                    break;
                case '^':
                    stk.push(pow(operand1, operand2));
                    break;
                default:
                    cout << "Invalid operator\n";
                    return -1;
            }
        }
    }

    // The final result will be at the top of the stack
    return stk.top();
}

int main() {
    string expr;
    cout << "Enter a postfix expression: ";
    getline(cin, expr);

    int result = evaluatePostfix(expr);
    if (result != -1) {
        cout << "Result: " << result << endl;
    }

    return 0;
}
```

Output:



E:\desktop\MCA-1 (Data St

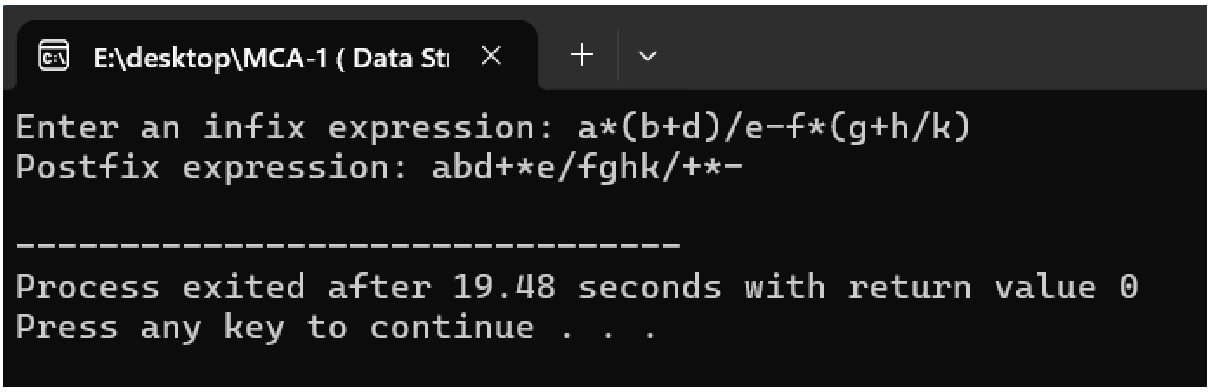
```
Enter a postfix expression: 65*78+*87-45*++
Result: 471

--------------------------------
Process exited after 57.26 seconds with return value 0
Press any key to continue . . .
```

3. Write a C++ program to convert an expression from Infix to Postfix.

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
using namespace std;

// Function to check if a character is an operator
bool isOperator(char c) {
   return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}

// Function to get the precedence of an operator
int precedence(char c) {
   if (c == '^')
      return 3;
   else if (c == '*' || c == '/')
      return 2;
   else if (c == '+' || c == '-')
      return 1;
   else
      return -1; // for '('
}

// Function to convert an infix expression to a postfix expression
string infixToPostfix(string infix) {
   stack<char> stk;
   string postfix = "";

   // Iterate over each character in the infix expression
   for (size_t i = 0; i < infix.size(); i++) {
      char c = infix[i];
      // If the character is an operand (digit or letter), append it to the postfix expression
      if (isalnum(c)) {
         postfix += c;
      } else if (c == '(') { // If the character is '(', push it onto the stack
         stk.push(c);
      } else if (c == ')') { // If the character is ')', pop and append operators from the stack until
'(' is encountered
         while (!stk.empty() && stk.top() != '(') {
            postfix += stk.top();
            stk.pop();
         }
         if (!stk.empty())
            stk.pop(); // Remove '(' from the stack
      } else { // If the character is an operator
         // Pop and append operators from the stack with higher or equal precedence
         while (!stk.empty() && precedence(c) <= precedence(stk.top())) {
            postfix += stk.top();
            stk.pop();
         }
         stk.push(c); // Push the current operator onto the stack
      }
   }
```

```
    // Append any remaining operators from the stack to the postfix expression
    while (!stk.empty()) {
        postfix += stk.top();
        stk.pop();
    }

    return postfix;
}

int main() {
    string infix;
    cout << "Enter an infix expression: ";
    getline(cin, infix);

    // Convert the infix expression to postfix
    string postfix = infixToPostfix(infix);
    cout << "Postfix expression: " << postfix << endl;

    return 0;
}
```

Output:



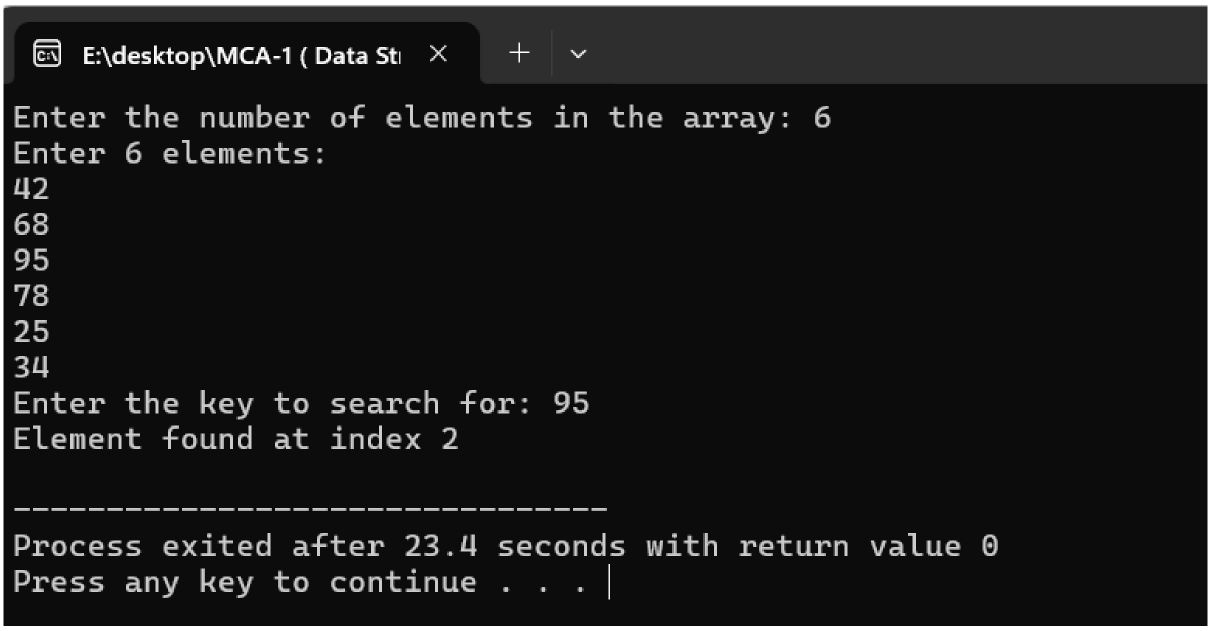
```
Enter an infix expression: a*(b+d)/e-f*(g+h/k)
Postfix expression: abd+*e/fghk/+*-

--------------------------------
Process exited after 19.48 seconds with return value 0
Press any key to continue . . .
```

4. Write a C++ program to implement using recursive functions.

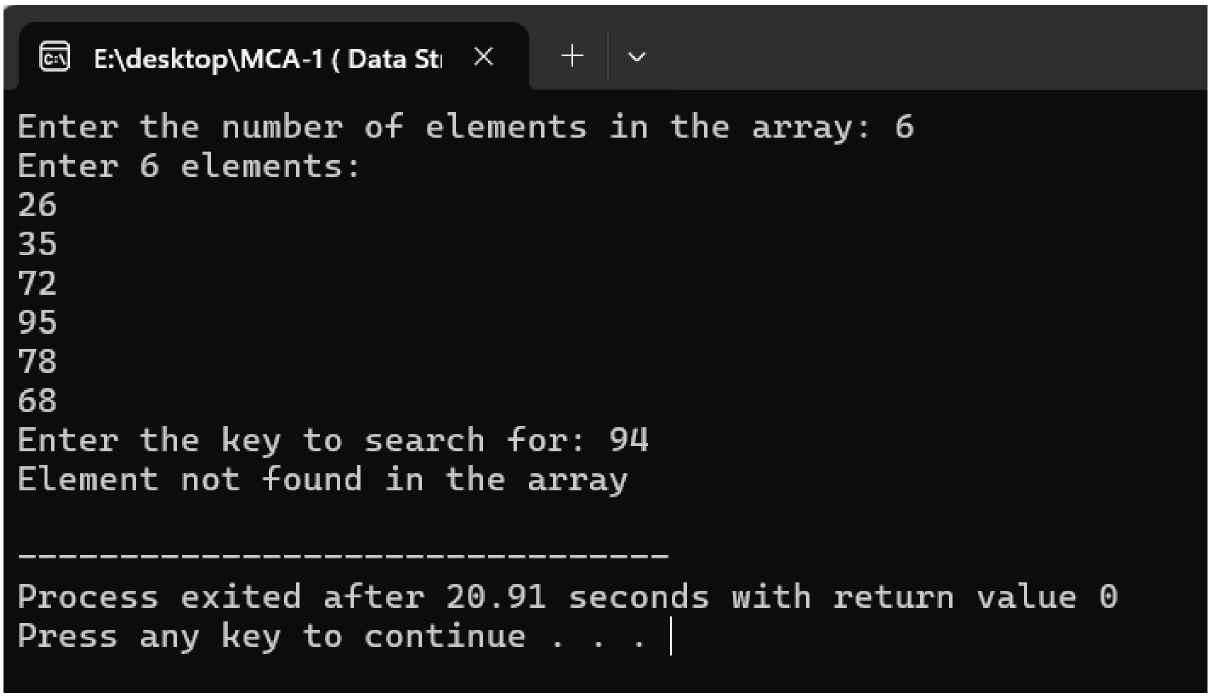
a. Linear Search

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

// Recursive function for linear search
int linearSearchRecursive(int arr[], int start, int end, int key) {
   // Base case: If start index is greater than end index, key is not found
   if (start > end) {
      return -1; // Key not found
   }

   // If the key is found at start index, return the index
   if (arr[start] == key) {
      return start;
   }

   // Recursive case: Search in the remaining array (start + 1 to end)
   return linearSearchRecursive(arr, start + 1, end, key);
}

int main() {
   int n;
   cout << "Enter the number of elements in the array: ";
   cin >> n;

   // Declare an array of size n
   int arr[n];

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

   // Input the key to search for
   int key;
   cout << "Enter the key to search for: ";
   cin >> key;

   // Perform linear search using recursive function
   int result = linearSearchRecursive(arr, 0, n - 1, key);

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

   return 0;
}
```

Output:



```
Enter the number of elements in the array: 6
Enter 6 elements:
42
68
95
78
25
34
Enter the key to search for: 95
Element found at index 2

--------------------------------
Process exited after 23.4 seconds with return value 0
Press any key to continue . . .
```

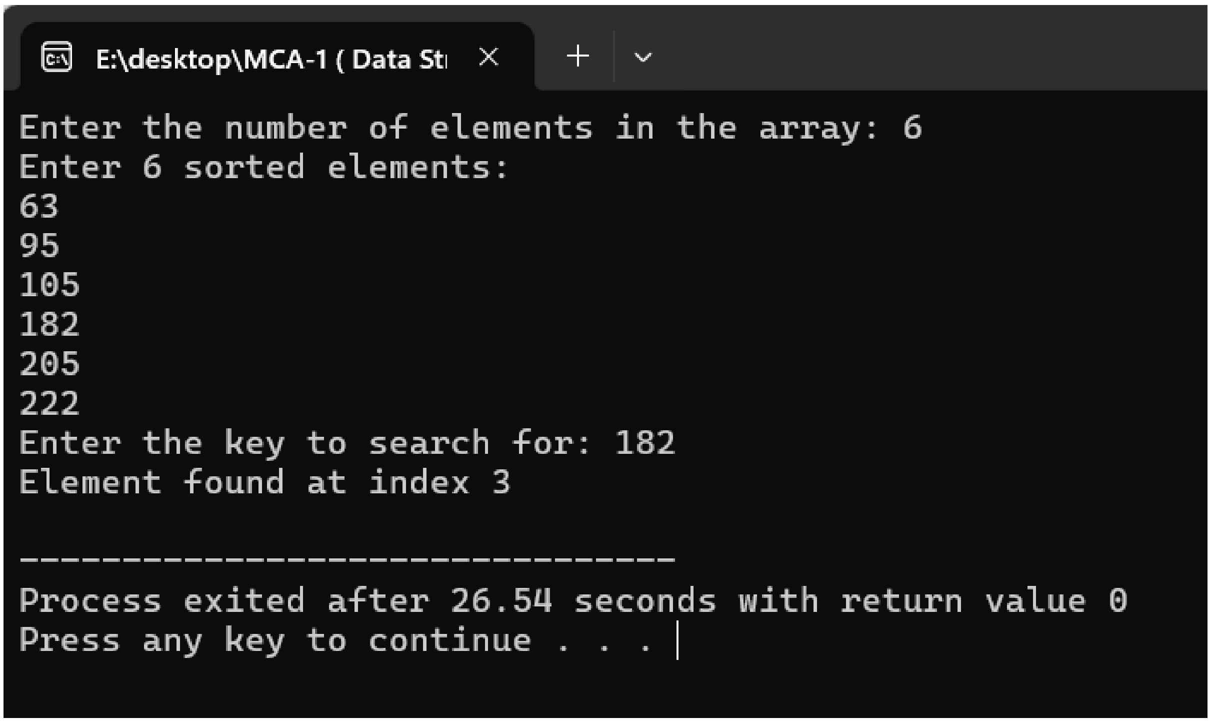


```
Enter the number of elements in the array: 6
Enter 6 elements:
26
35
72
95
78
68
Enter the key to search for: 94
Element not found in the array

--------------------------------
Process exited after 20.91 seconds with return value 0
Press any key to continue . . .
```

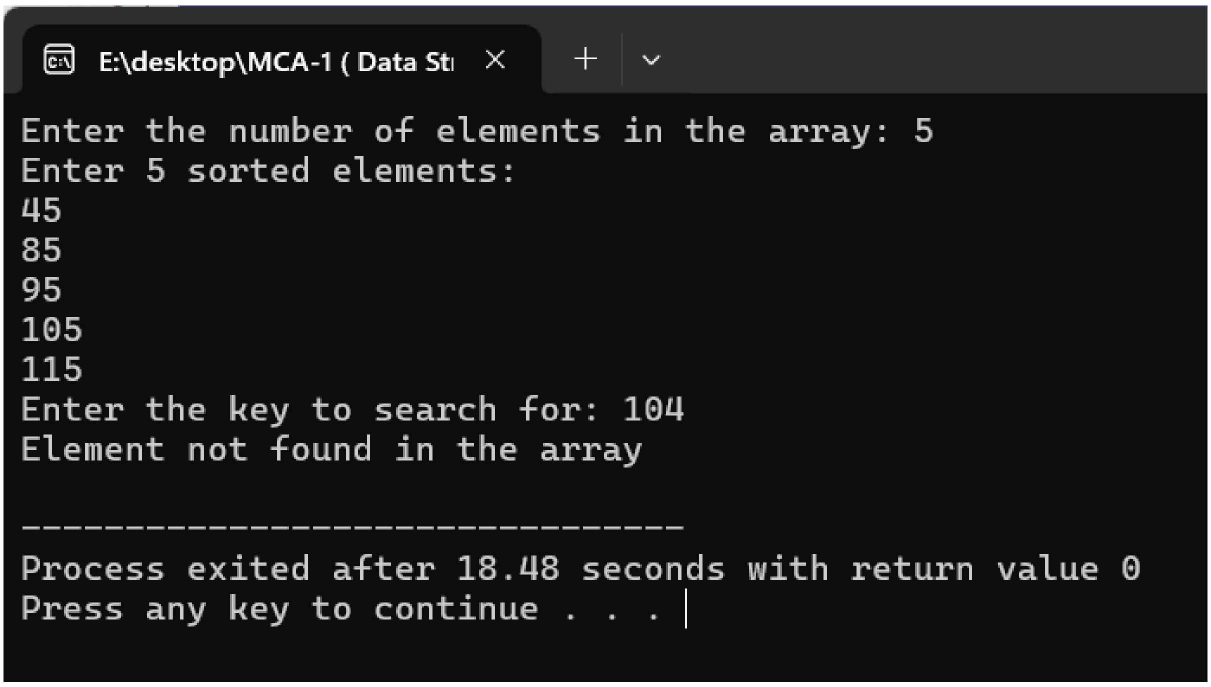
b. Binary Search

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

// Recursive function for binary search
int binarySearchRecursive(int arr[], int low, int high, int key) {
    if (low > high) {
        return -1; // Key not found
    }

    int mid = low + (high - low) / 2;
    if (arr[mid] == key) {
        return mid; // Key found at index mid
    } else if (arr[mid] < key) {
        return binarySearchRecursive(arr, mid + 1, high, key); // Search in the right half
    } else {
        return binarySearchRecursive(arr, low, mid - 1, key); // Search in the left half
    }
}

int main() {
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;

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

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

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

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

    return 0;
}
```

Output:



```
Enter the number of elements in the array: 6
Enter 6 sorted elements:
63
95
105
182
205
222
Enter the key to search for: 182
Element found at index 3

--------------------------------
Process exited after 26.54 seconds with return value 0
Press any key to continue . . .
```



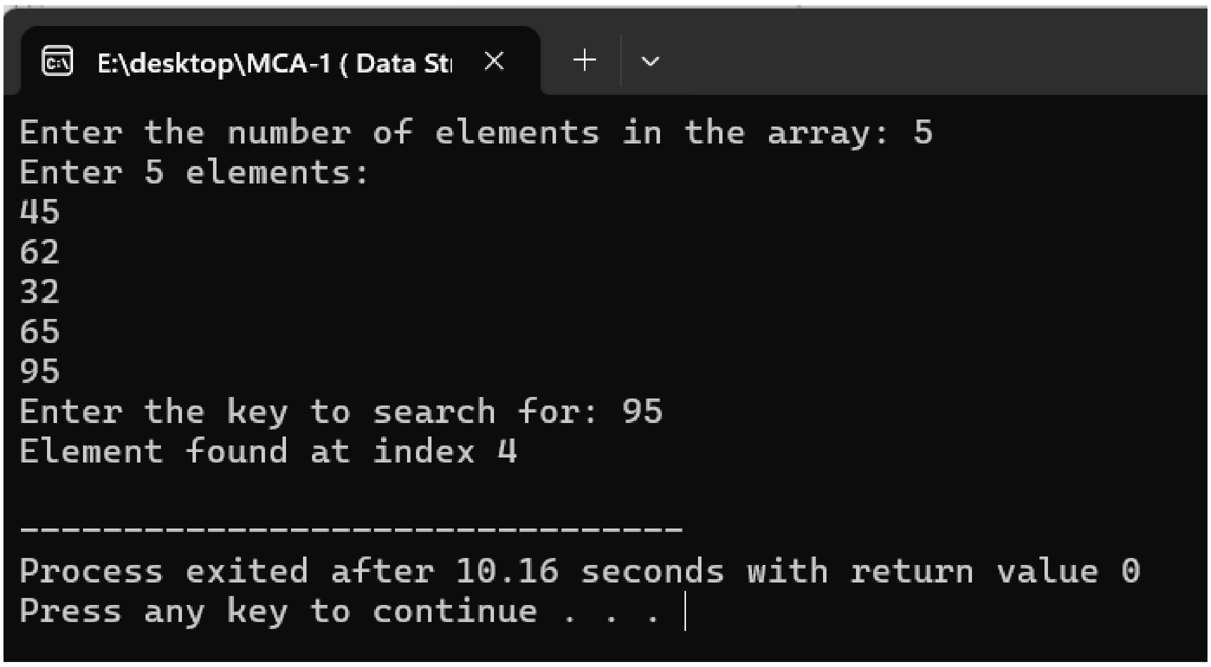
```
Enter the number of elements in the array: 5
Enter 5 sorted elements:
45
85
95
105
115
Enter the key to search for: 104
Element not found in the array

--------------------------------
Process exited after 18.48 seconds with return value 0
Press any key to continue . . .
```

5. Write a C++ program to implement using non-recursive functions.

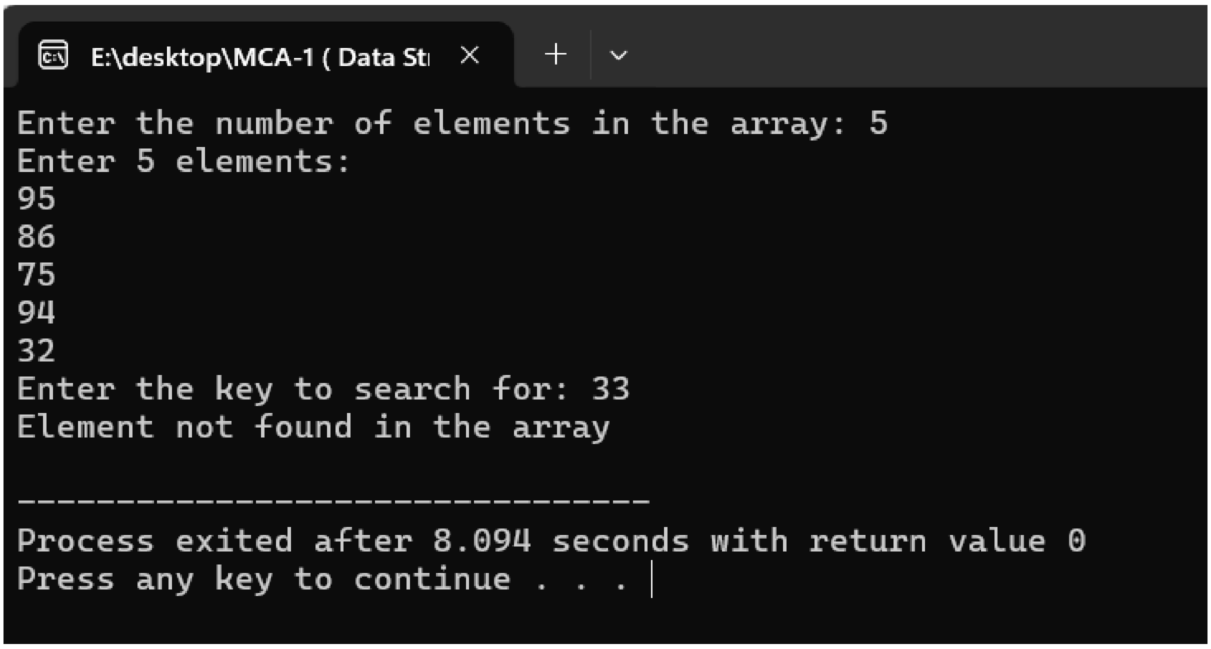
a. Linear Search

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

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

int main() {
   int n;
   cout << "Enter the number of elements in the array: ";
   cin >> n;

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

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

   int result = linearSearch(arr, n, key);

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

   return 0;
}
```

Output:



```
Enter the number of elements in the array: 5
Enter 5 elements:
45
62
32
65
95
Enter the key to search for: 95
Element found at index 4

--------------------------------
Process exited after 10.16 seconds with return value 0
Press any key to continue . . .
```

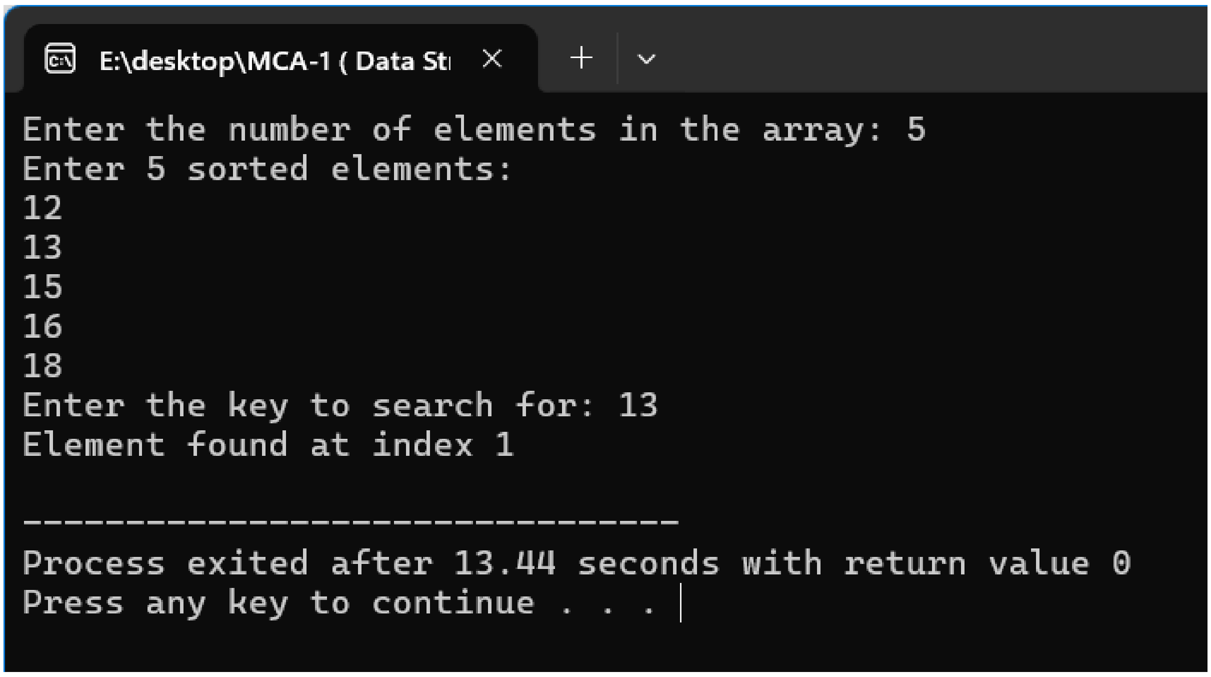


```
Enter the number of elements in the array: 5
Enter 5 elements:
95
86
75
94
32
Enter the key to search for: 33
Element not found in the array

--------------------------------
Process exited after 8.094 seconds with return value 0
Press any key to continue . . .
```

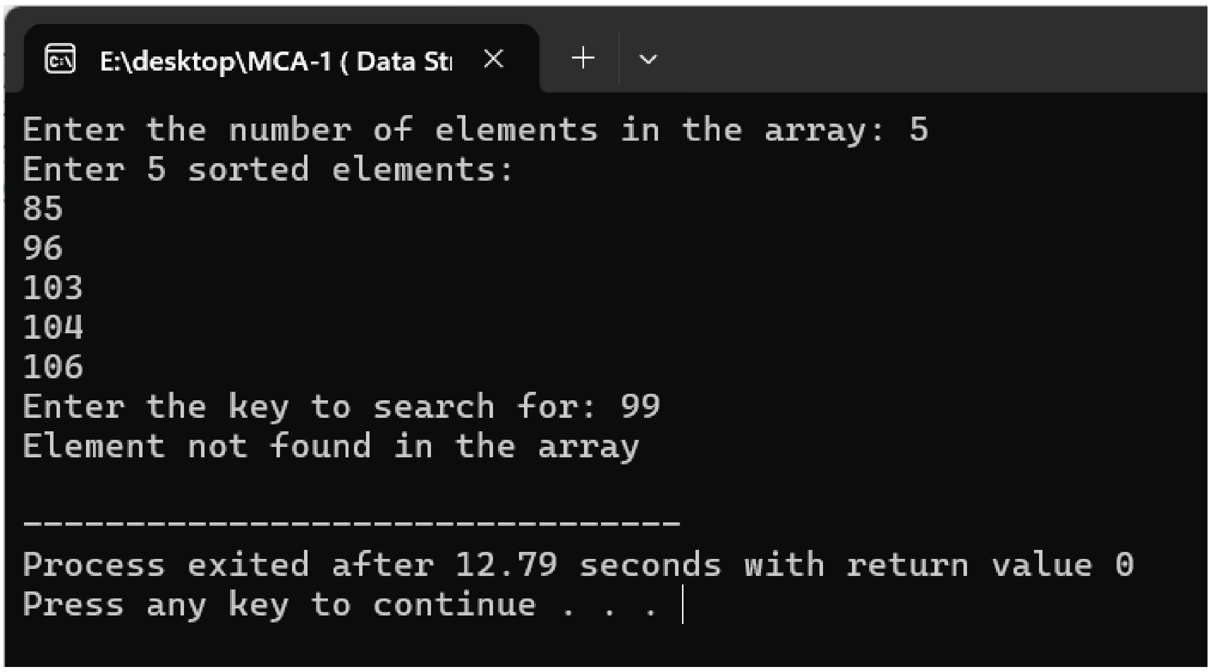
b. Binary Search

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

int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;

    while (low <= high) {
        int mid = low + (high - low) / 2;

        if (arr[mid] == key) {
            return mid; // Return the index if key is found
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    return -1; // Return -1 if key is not found
}

int main() {
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;

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

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

    int result = binarySearch(arr, n, key);

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

    return 0;
}
```

Output:



```
Enter the number of elements in the array: 5
Enter 5 sorted elements:
12
13
15
16
18
Enter the key to search for: 13
Element found at index 1

--------------------------------
Process exited after 13.44 seconds with return value 0
Press any key to continue . . .
```

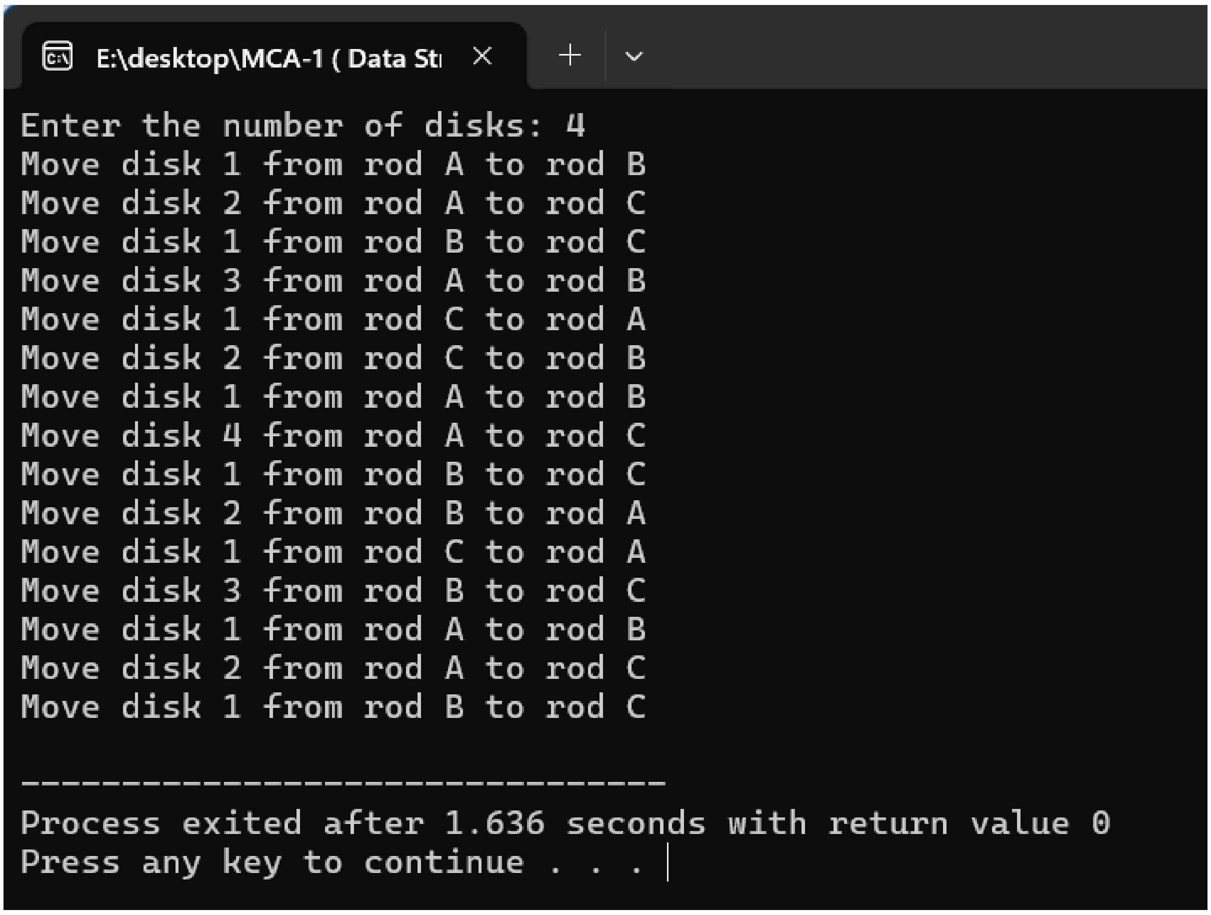


```
Enter the number of elements in the array: 5
Enter 5 sorted elements:
85
96
103
104
106
Enter the key to search for: 99
Element not found in the array

--------------------------------
Process exited after 12.79 seconds with return value 0
Press any key to continue . . .
```

6. Write a C++ program of Tower of Hanoi.

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

// Function to solve Tower of Hanoi puzzle
void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        cout << "Move disk 1 from rod " << source << " to rod " << destination << endl;
        return;
    }

    towerOfHanoi(n - 1, source, destination, auxiliary);
    cout << "Move disk " << n << " from rod " << source << " to rod " << destination << endl;
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

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

    towerOfHanoi(n, 'A', 'B', 'C');

    return 0;
}
```

Output:

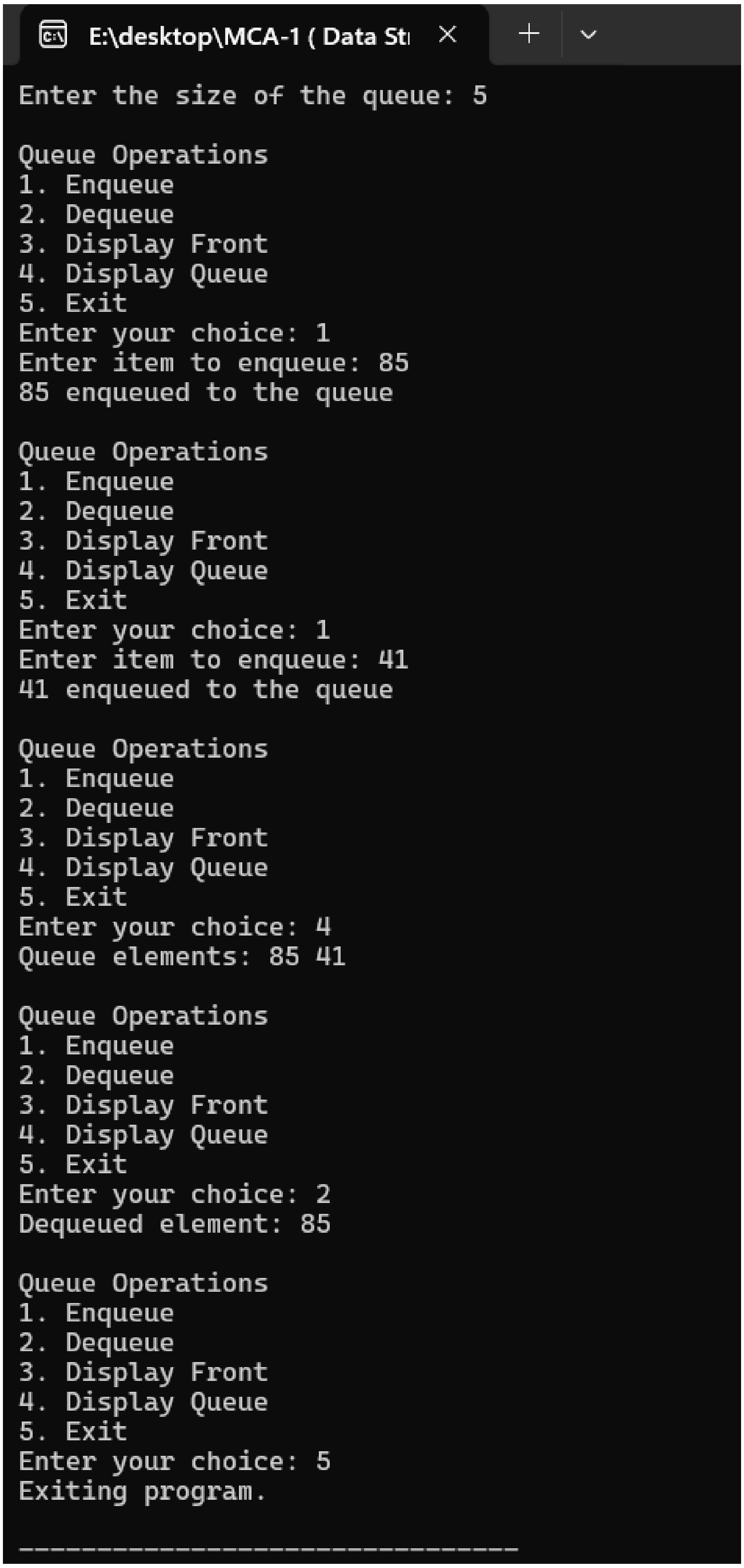


```
Enter the number of disks: 4
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C

--------------------------------
Process exited after 1.636 seconds with return value 0
Press any key to continue . . .
```

7. Write a C++ program to implement the Queue ADT using an Array.

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

class Queue {
private:
    int front, rear;
    int* arr;
    int maxSize;

public:
    // Constructor to initialize the queue with a given size
    Queue(int size) {
        front = rear = -1;
        maxSize = size;
        arr = new int[maxSize];
    }

    // Destructor to free the memory allocated for the queue array
    ~Queue() {
        delete[] arr;
    }

    // Check if the queue is empty
    bool isEmpty() {
        return (front == -1 && rear == -1);
    }

    // Check if the queue is full
    bool isFull() {
        return (rear + 1) % maxSize == front ? true : false;
    }

    // Enqueue an element into the queue
    void enqueue(int x) {
        if (isFull()) {
            cout << "Error: Queue is full" << endl;
            return;
        } else if (isEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % maxSize;
        }
        arr[rear] = x;
        cout << x << " enqueued to the queue" << endl;
    }

    // Dequeue an element from the queue and return the dequeued element
    int dequeue() {
        if (isEmpty()) {
            cout << "Error: Queue is empty" << endl;
            return -1;
        } else if (front == rear) {
            int dequeued = arr[front];
            front = rear = -1;
            return dequeued;
        } else {
            int dequeued = arr[front];
            front = (front + 1) % maxSize;
            return dequeued;
        }
    }

    // Get the front element of the queue
    int getFront() {
        if (isEmpty()) {
            cout << "Error: Queue is empty" << endl;
            return -1;
        }
        return arr[front];
    }

    // Display all elements in the queue
    void display() {
        if (isEmpty()) {
            cout << "Error: Queue is empty" << endl;
            return;
        }
        cout << "Queue elements: ";
        int i = front;
        while (i != rear) {
            cout << arr[i] << " ";
            i = (i + 1) % maxSize;
        }
        cout << arr[rear] << endl;
    }
};

int main() {
    int size;
    cout << "Enter the size of the queue: ";
    cin >> size;

    // Create a queue object with the specified size
    Queue q(size);

    int choice, item;

    do {
        cout << "\nQueue Operations\n";
        cout << "1. Enqueue\n";
        cout << "2. Dequeue\n";
        cout << "3. Display Front\n";
        cout << "4. Display Queue\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter item to enqueue: ";
                cin >> item;
                q.enqueue(item);
                break;
            case 2:
                item = q.dequeue();
                if (item != -1) {
                    cout << "Dequeued element: " << item << endl;
                }
                break;
            case 3:
                cout << "Front element: " << q.getFront() << endl;
                break;
            case 4:
                q.display();
                break;
            case 5:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 5);

    return 0;
}
```

Output:



E:\desktop\MCA-1 (Data St

```
Enter the size of the queue: 5

Queue Operations
1. Enqueue
2. Dequeue
3. Display Front
4. Display Queue
5. Exit
Enter your choice: 1
Enter item to enqueue: 85
85 enqueued to the queue

Queue Operations
1. Enqueue
2. Dequeue
3. Display Front
4. Display Queue
5. Exit
Enter your choice: 1
Enter item to enqueue: 41
41 enqueued to the queue

Queue Operations
1. Enqueue
2. Dequeue
3. Display Front
4. Display Queue
5. Exit
Enter your choice: 4
Queue elements: 85 41

Queue Operations
1. Enqueue
2. Dequeue
3. Display Front
4. Display Queue
5. Exit
Enter your choice: 2
Dequeued element: 85

Queue Operations
1. Enqueue
2. Dequeue
3. Display Front
4. Display Queue
5. Exit
Enter your choice: 5
Exiting program.

--------------------------------
```

8. Write a C++ program to implement the Linked List editing to perform following operations.

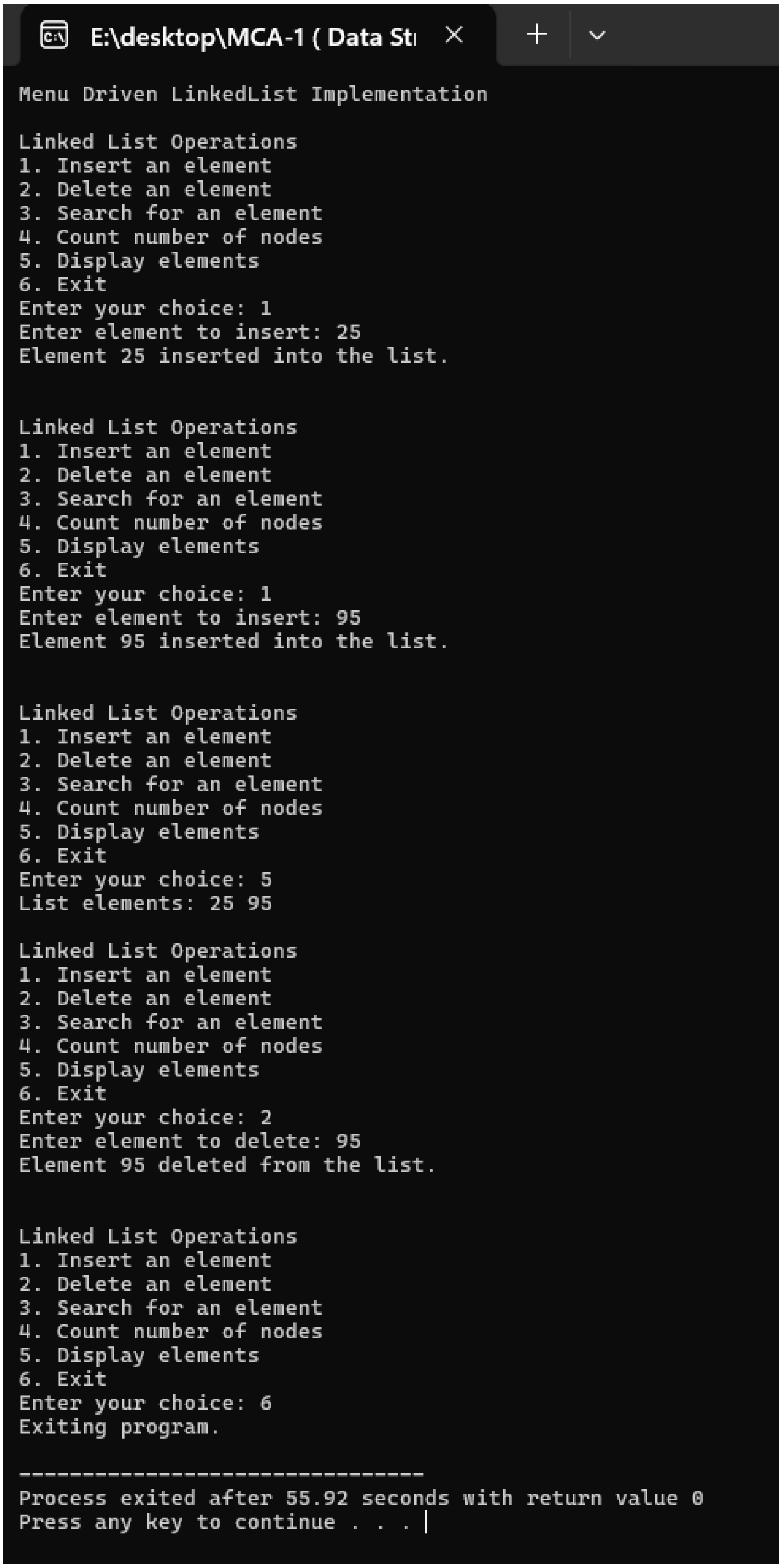
   a. Insert an element into a list.
   b. Delete an element from the list.
   c. Search for a key element in the list.
   d. Count number of nodes in the list.

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

class Node {
public:
   int data;
   Node* next;

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

class LinkedList {
private:
   Node* head;

public:
   LinkedList() {
      head = NULL;
   }

   // Insert an element into the list
   void insert(int value) {
      Node* newNode = new Node(value);
      if (head == NULL) {
         head = newNode;
      } else {
         Node* temp = head;
         while (temp->next != NULL) {
            temp = temp->next;
         }
         temp->next = newNode;
      }
      cout << "Element " << value << " inserted into the list. \n" << endl;
   }

   // Delete an element from the list
   void remove(int value) {
      if (head == NULL) {
         cout << "List is empty. Cannot delete. \n" << endl;
         return;
      }

      if (head->data == value) {
         Node* temp = head;
```

```
        head = head->next;
        delete temp;
        cout << "Element " << value << " deleted from the list. \n" << endl;
        return;
    }

    Node* prev = NULL;
    Node* curr = head;
    while (curr != NULL && curr->data != value) {
        prev = curr;
        curr = curr->next;
    }

    if (curr == NULL) {
        cout << "Element " << value << " not found in the list. \n" << endl;
    } else {
        prev->next = curr->next;
        delete curr;
        cout << "Element " << value << " deleted from the list. \n" << endl;
    }
}

// Search for a key element in the list
void search(int key) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == key) {
            cout << "Element " << key << " found in the list. \n" << endl;
            return;
        }
        temp = temp->next;
    }
    cout << "Element " << key << " not found in the list. \n" << endl;
}

// Count number of nodes in the list
int countNodes() {
    int count = 0;
    Node* temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }
    return count;
}

// Display the elements in the list
void display() {
    if (head == NULL) {
        cout << "List is empty." << endl;
        return;
    }
    cout << "List elements: ";
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " ";
```

```
            temp = temp->next;
        }
        cout << endl;
    }

    // Destructor to free memory allocated for nodes
    ~LinkedList() {
        Node* temp = head;
        while (temp != NULL) {
            Node* next = temp->next;
            delete temp;
            temp = next;
        }
    }
};

int main() {
    LinkedList list;
    int choice, value;

    cout<<"Menu Driven LinkedList Implementation \n";
    do {

        cout << "\nLinked List Operations\n";
        cout << "1. Insert an element\n";
        cout << "2. Delete an element\n";
        cout << "3. Search for an element\n";
        cout << "4. Count number of nodes\n";
        cout << "5. Display elements\n";
        cout << "6. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter element to insert: ";
                cin >> value;
                list.insert(value);
                break;
            case 2:
                cout << "Enter element to delete: ";
                cin >> value;
                list.remove(value);
                break;
            case 3:
                cout << "Enter element to search: ";
                cin >> value;
                list.search(value);
                break;
            case 4:
                cout << "Number of nodes: " << list.countNodes() << endl;
                break;
            case 5:
                list.display();
                break;
            case 6:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 6);

    return 0;
}
```

Output:



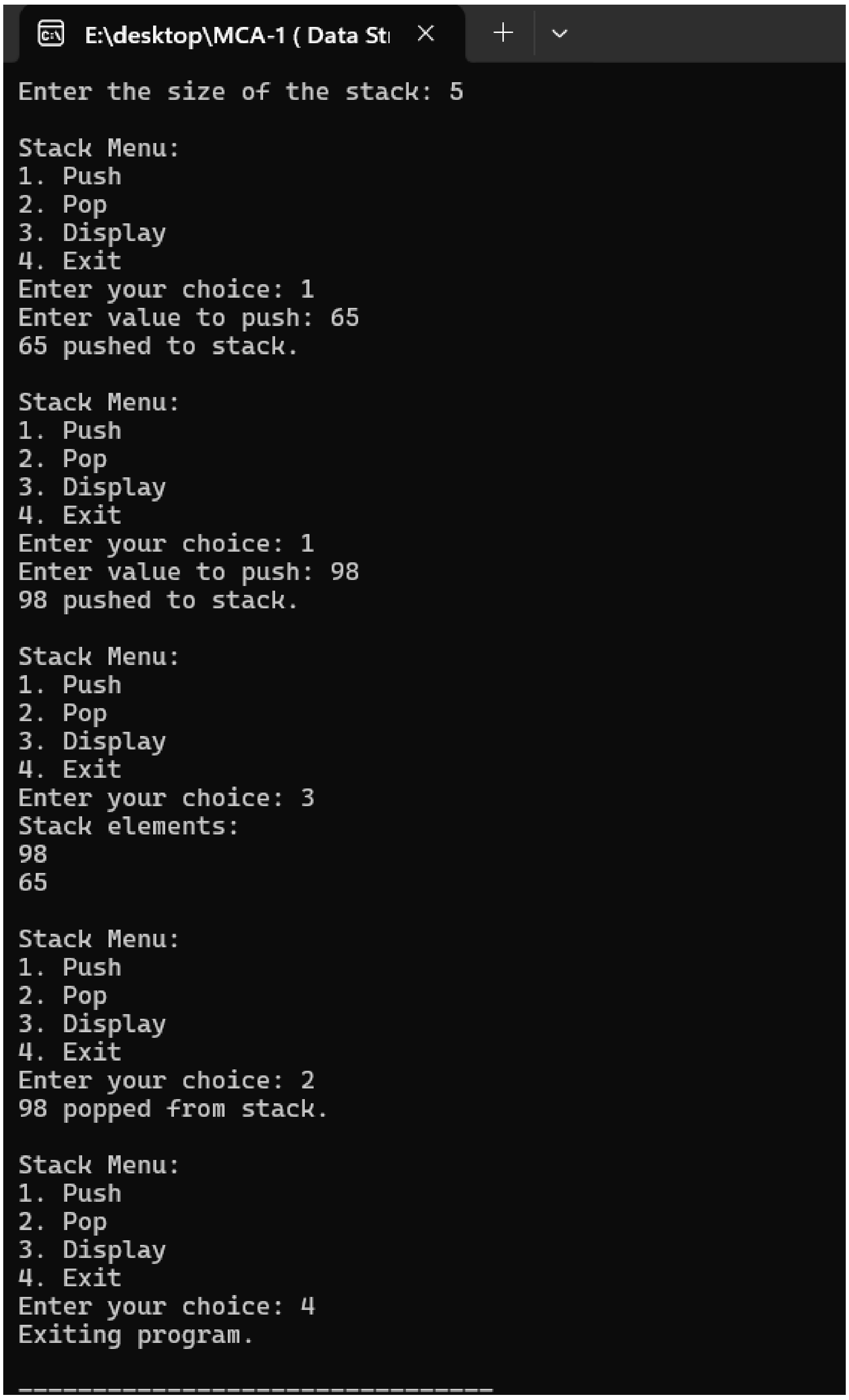
```
Menu Driven LinkedList Implementation

Linked List Operations
1. Insert an element
2. Delete an element
3. Search for an element
4. Count number of nodes
5. Display elements
6. Exit
Enter your choice: 1
Enter element to insert: 25
Element 25 inserted into the list.


Linked List Operations
1. Insert an element
2. Delete an element
3. Search for an element
4. Count number of nodes
5. Display elements
6. Exit
Enter your choice: 1
Enter element to insert: 95
Element 95 inserted into the list.


Linked List Operations
1. Insert an element
2. Delete an element
3. Search for an element
4. Count number of nodes
5. Display elements
6. Exit
Enter your choice: 5
List elements: 25 95

Linked List Operations
1. Insert an element
2. Delete an element
3. Search for an element
4. Count number of nodes
5. Display elements
6. Exit
Enter your choice: 2
Enter element to delete: 95
Element 95 deleted from the list.


Linked List Operations
1. Insert an element
2. Delete an element
3. Search for an element
4. Count number of nodes
5. Display elements
6. Exit
Enter your choice: 6
Exiting program.

--------------------------------
Process exited after 55.92 seconds with return value 0
Press any key to continue . . .
```

9. Write a C++ program to implement the following using the Single Linked List.

a. Stack ADT

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

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

    // Constructor to initialize a node with given data
    Node(int value) {
        data = value;
        next = NULL;
    }
};

// Stack class implementing the Stack ADT using a linked list
class Stack {
private:
    Node* top; // Pointer to the top of the stack

public:
    // Constructor to initialize an empty stack
    Stack() {
        top = NULL;
    }

    // Push an element onto the stack
    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = top;
        top = newNode;
        cout << value << " pushed to stack.\n";
    }

    // Pop the top element from the stack
    void pop() {
        if (isEmpty()) {
            cout << "Stack Underflow! Cannot pop from an empty stack.\n";
            return;
        }
        Node* temp = top;
        top = top->next;
        cout << temp->data << " popped from stack.\n";
        delete temp;
    }

    // Check if the stack is empty
    bool isEmpty() {
        return top == NULL;
    }
    // Display the elements in the stack
    void display() {
        if (isEmpty()) {
            cout << "Stack is empty.\n";
            return;
        }
        cout << "Stack elements:\n";
        Node* temp = top;
        while (temp != NULL) {
            cout << temp->data << endl;
            temp = temp->next;
        }
    }
};

int main() {
    Stack stack; // Create a stack object
    int choice, value, size;

    // Ask user for the size of the stack
    cout << "Enter the size of the stack: ";
    cin >> size;

    // Menu-driven program to interact with the stack
    do {
        cout << "\nStack Menu:\n";
        cout << "1. Push\n";
        cout << "2. Pop\n";
        cout << "3. Display\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                // Push an element onto the stack if not full
                if (size > 0) {
                    cout << "Enter value to push: ";
                    cin >> value;
                    stack.push(value);
                    size--;
                } else {
                    cout << "Stack is full. Cannot push more elements.\n";
                }
                break;
            case 2:
                // Pop an element from the stack if not empty
                stack.pop();
                size++;
                break;
            case 3:
                // Display the elements in the stack
                stack.display();
                break;
            case 4:
                // Exit the program
                cout << "Exiting program.\n";
                break;
            default:
                // Invalid choice
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 4);

    return 0;
}
```

Output:


E:\desktop\MCA-1 (Data St

```
Enter the size of the stack: 5

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 65
65 pushed to stack.

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 1
Enter value to push: 98
98 pushed to stack.

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 3
Stack elements:
98
65

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 2
98 popped from stack.

Stack Menu:
1. Push
2. Pop
3. Display
4. Exit
Enter your choice: 4
Exiting program.

--------------------------------
```

10.Write a C++ program to perform the following operations.

a. Insert an element into a binary search tree.

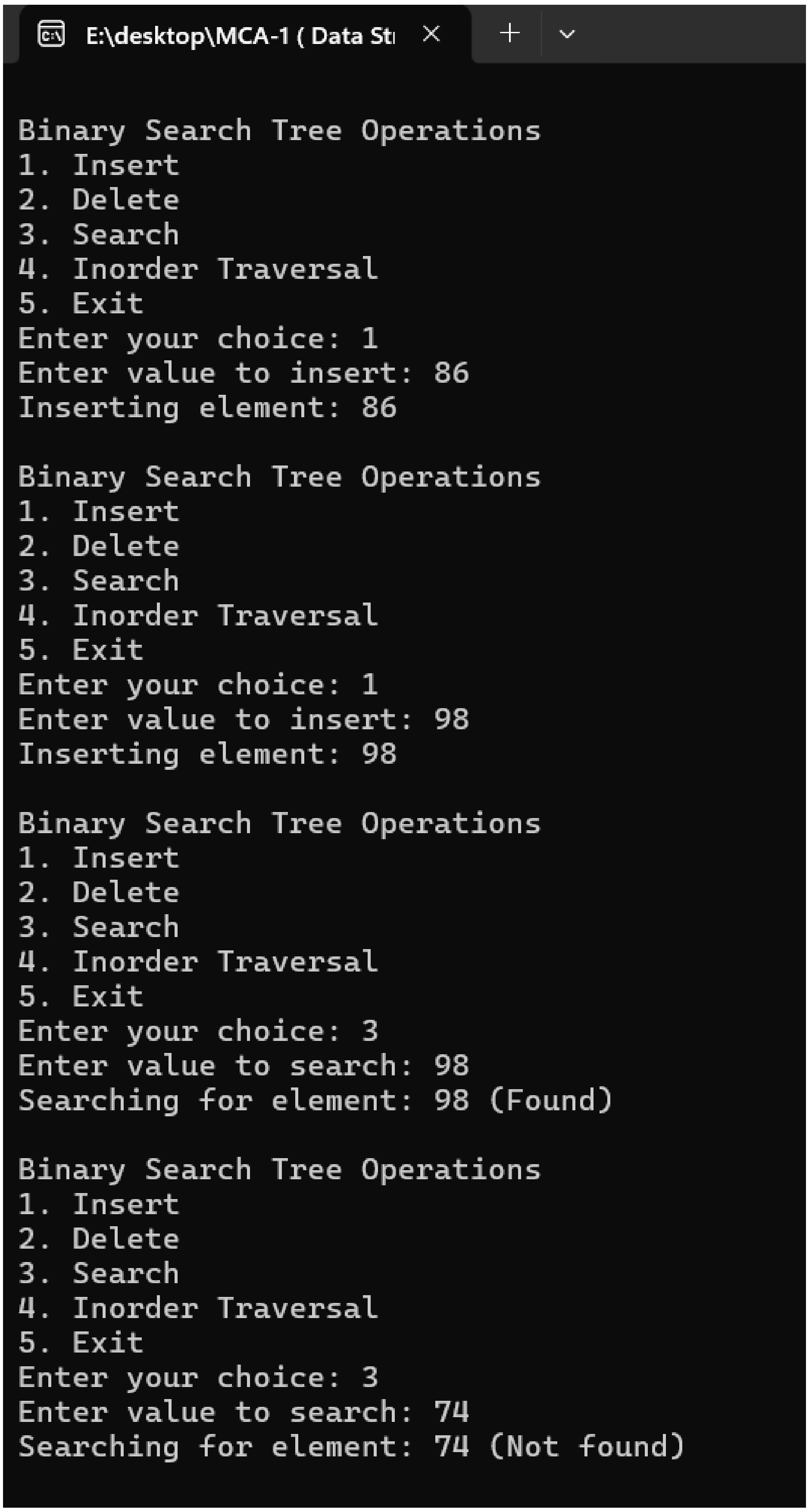
b. Delete an element from a binary search tree.

c. Search for a key element in a binary search tree.

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

// Node class for BST
class Node {
public:
   int data;
   Node* left;
   Node* right;

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

// Insert a new node with the given value into the BST
Node* insert(Node* root, int value) {
   if (root == NULL) {
      cout << "Inserting element: " << value << endl;
      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;
}

// Find the minimum value node in a BST (used in deleteNode function)
Node* minValueNode(Node* node) {
   Node* current = node;
   while (current && current->left != NULL) {
      current = current->left;
   }
   return current;
}

// Delete a node with the given value from the BST
Node* deleteNode(Node* root, int value) {
   if (root == NULL) {
      cout << "Element " << value << " not found in the tree.\n";
      return root;
   }

   if (value < root->data) {
      root->left = deleteNode(root->left, value);
   } else if (value > root->data) {
      root->right = deleteNode(root->right, value);
   } else {
      cout << "Deleting element: " << value << endl;
      if (root->left == NULL) {
          Node* temp = root->right;
          delete root;
          return temp;
      } else if (root->right == NULL) {
          Node* temp = root->left;
          delete root;
          return temp;
      }

      Node* temp = minValueNode(root->right);
      root->data = temp->data;
      root->right = deleteNode(root->right, temp->data);
   }

   return root;
}

// Search for a node with the given value in the BST
bool search(Node* root, int value) {
   if (root == NULL) {
      cout << "Searching for element: " << value << " (Not found)\n";
      return false;
   }

   if (root->data == value) {
      cout << "Searching for element: " << value << " (Found)\n";
      return true;
   }

   if (value < root->data) {
      return search(root->left, value);
   } else {
      return search(root->right, value);
   }
}

// Inorder traversal of the BST
void inorder(Node* root) {
   if (root != NULL) {
      inorder(root->left);
      cout << root->data << " ";
      inorder(root->right);
   }
}
```

```
int main() {
    Node* root = NULL;
    int choice, value;

    do {
        cout << "\nBinary Search Tree Operations\n";
        cout << "1. Insert\n";
        cout << "2. Delete\n";
        cout << "3. Search\n";
        cout << "4. Inorder Traversal\n";
        cout << "5. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
            case 1:
                cout << "Enter value to insert: ";
                cin >> value;
                root = insert(root, value);
                break;
            case 2:
                cout << "Enter value to delete: ";
                cin >> value;
                root = deleteNode(root, value);
                break;
            case 3:
                cout << "Enter value to search: ";
                cin >> value;
                search(root, value);
                break;
            case 4:
                cout << "Inorder Traversal: ";
                inorder(root);
                cout << endl;
                break;
            case 5:
                cout << "Exiting program.\n";
                break;
            default:
                cout << "Invalid choice. Please try again.\n";
                break;
        }
    } while (choice != 5);

    return 0;
}
```

Output:

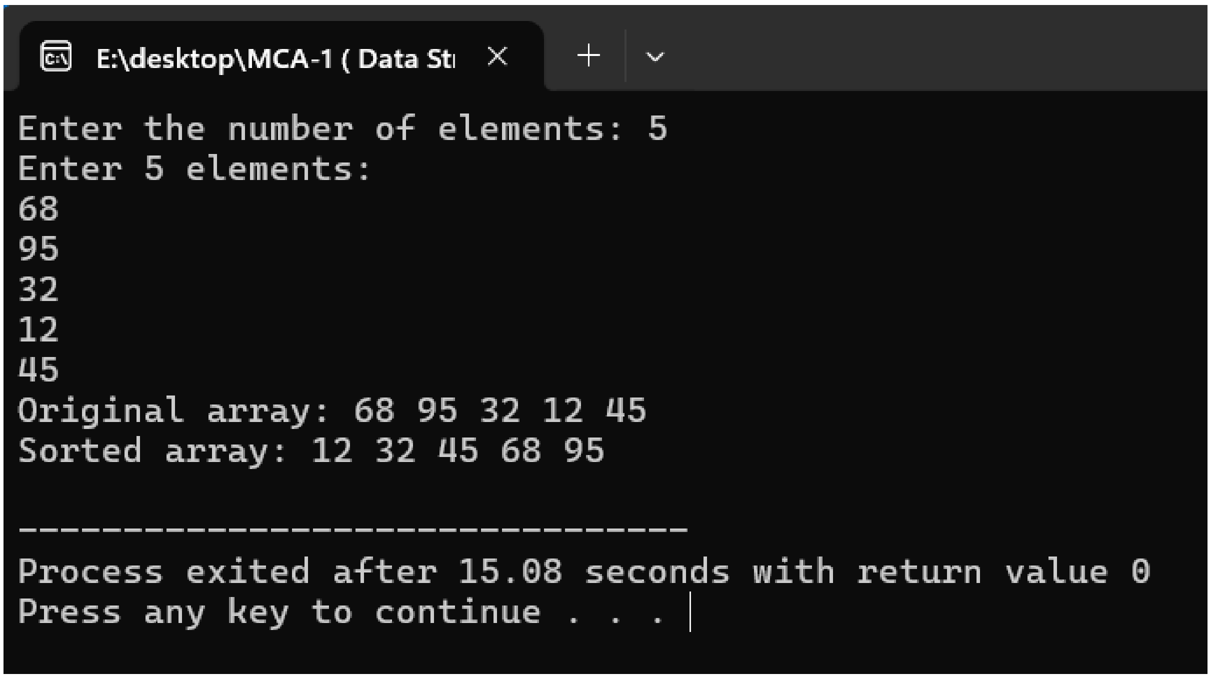


```
Binary Search Tree Operations
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 1
Enter value to insert: 86
Inserting element: 86

Binary Search Tree Operations
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 1
Enter value to insert: 98
Inserting element: 98

Binary Search Tree Operations
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 3
Enter value to search: 98
Searching for element: 98 (Found)

Binary Search Tree Operations
1. Insert
2. Delete
3. Search
4. Inorder Traversal
5. Exit
Enter your choice: 3
Enter value to search: 74
Searching for element: 74 (Not found)
```

11.Write a C++ program to implement Bubble Sort.

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

// Function to perform bubble sort
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]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original array: ";
    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:

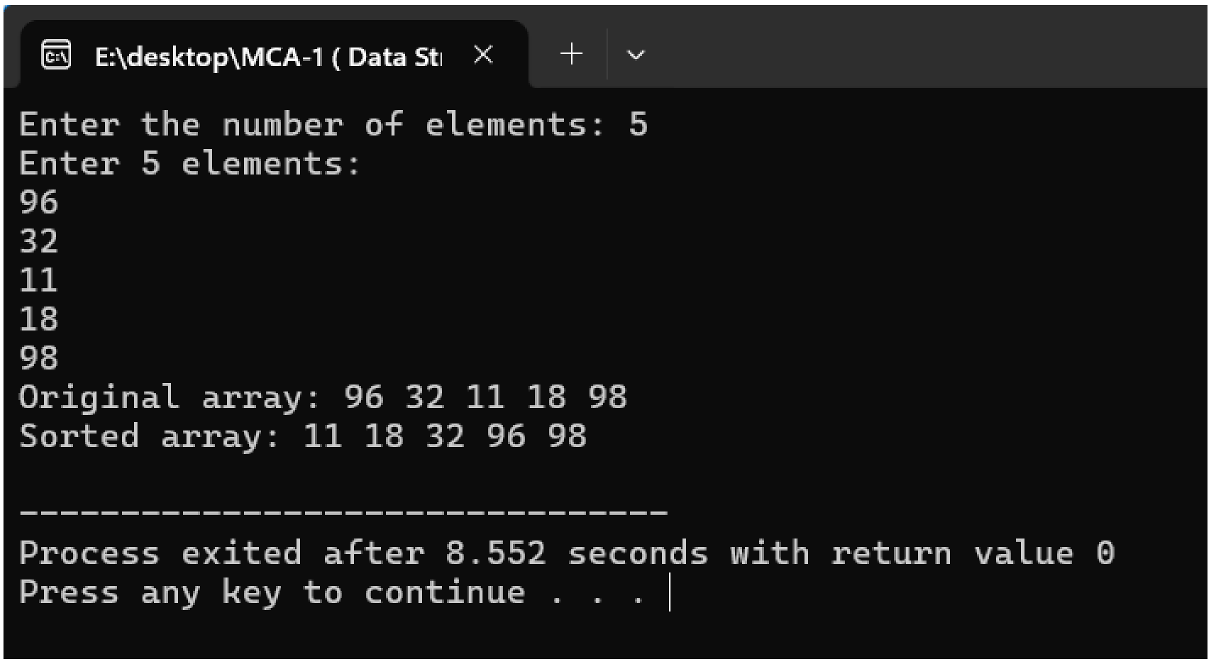


```
Enter the number of elements: 5
Enter 5 elements:
68
95
32
12
45
Original array: 68 95 32 12 45
Sorted array: 12 32 45 68 95

--------------------------------
Process exited after 15.08 seconds with return value 0
Press any key to continue . . .
```

12.Write a C++ program to implement Selection Sort.

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

// Function to perform selection sort
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;
            }
        }
        // Swap arr[i] and arr[minIndex]
        int temp = arr[i];
        arr[i] = arr[minIndex];
        arr[minIndex] = temp;
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original array: ";
    printArray(arr, n);

    selectionSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:

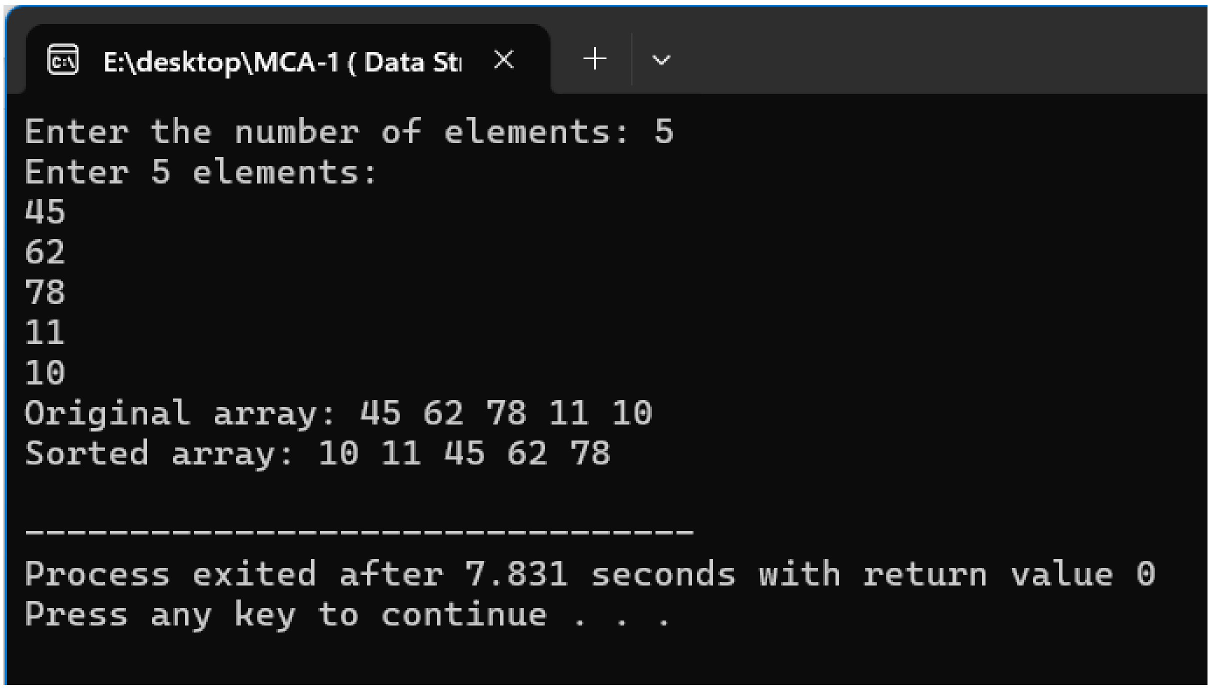


```
Enter the number of elements: 5
Enter 5 elements:
96
32
11
18
98
Original array: 96 32 11 18 98
Sorted array: 11 18 32 96 98

--------------------------------
Process exited after 8.552 seconds with return value 0
Press any key to continue . . .
```

13.Write a C++ program to implement Quick Sort.

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

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Select the rightmost element as pivot
    int i = low - 1; // Index of smaller element

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

// Function to perform Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // pi is partitioning index, arr[pi] is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original array: ";
    printArray(arr, n);

    quickSort(arr, 0, n - 1);
    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:

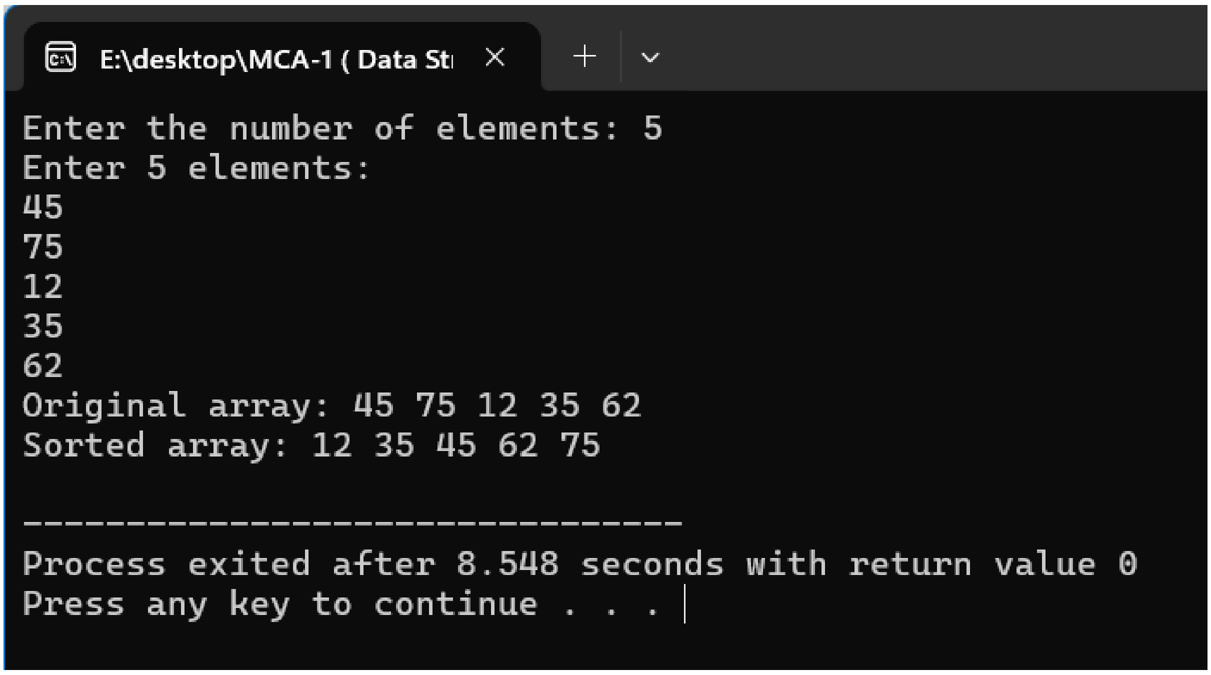


```
Enter the number of elements: 5
Enter 5 elements:
45
62
78
11
10
Original array: 45 62 78 11 10
Sorted array: 10 11 45 62 78

--------------------------------
Process exited after 7.831 seconds with return value 0
Press any key to continue . . .
```

14.Write a C++ program to implement Insertion Sort.

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

// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements of arr[0..i-1], that are greater than key, to one position ahead of their
current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original array: ";
    printArray(arr, n);

    insertionSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:



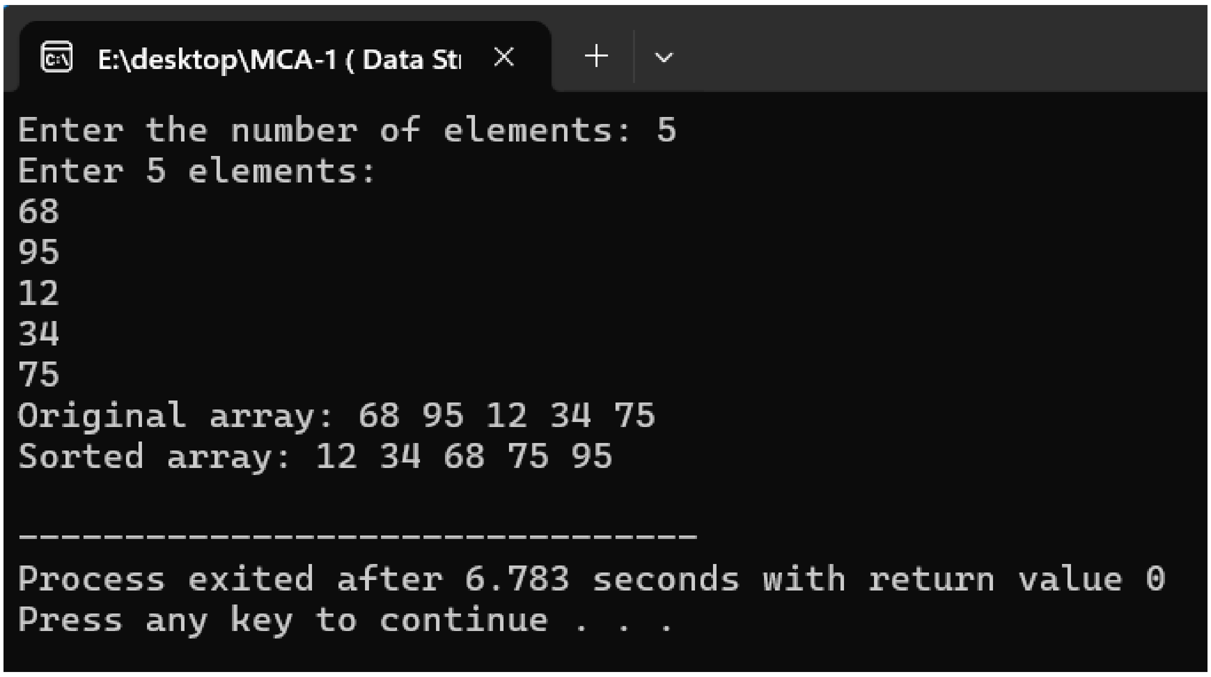
```
Enter the number of elements: 5
Enter 5 elements:
45
75
12
35
62
Original array: 45 75 12 35 62
Sorted array: 12 35 45 62 75

--------------------------------
Process exited after 8.548 seconds with return value 0
Press any key to continue . . .
```

15.Write a C++ program for implementing the Merge Sort.

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

// Function to merge two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r) {
   int n1 = m - l + 1;
   int n2 = r - m;

   // Create temporary arrays
   int L[n1], R[n2];

   // Copy data to temporary arrays L[] and R[]
   for (int i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (int j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   // Merge the temporary arrays back into arr[l..r]
   int i = 0;
   int j = 0;
   int k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      } else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }

   // Copy the remaining elements of L[], if there are any
   while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }

   // Copy the remaining elements of R[], if there are any
   while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}

// l is for left index and r is right index of the sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r) {
   if (l < r) {
      // Same as (l+r)/2, but avoids overflow for large l and h
      int m = l + (r - l) / 2;
```

```
        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

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

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

    cout << "Original array: ";
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:

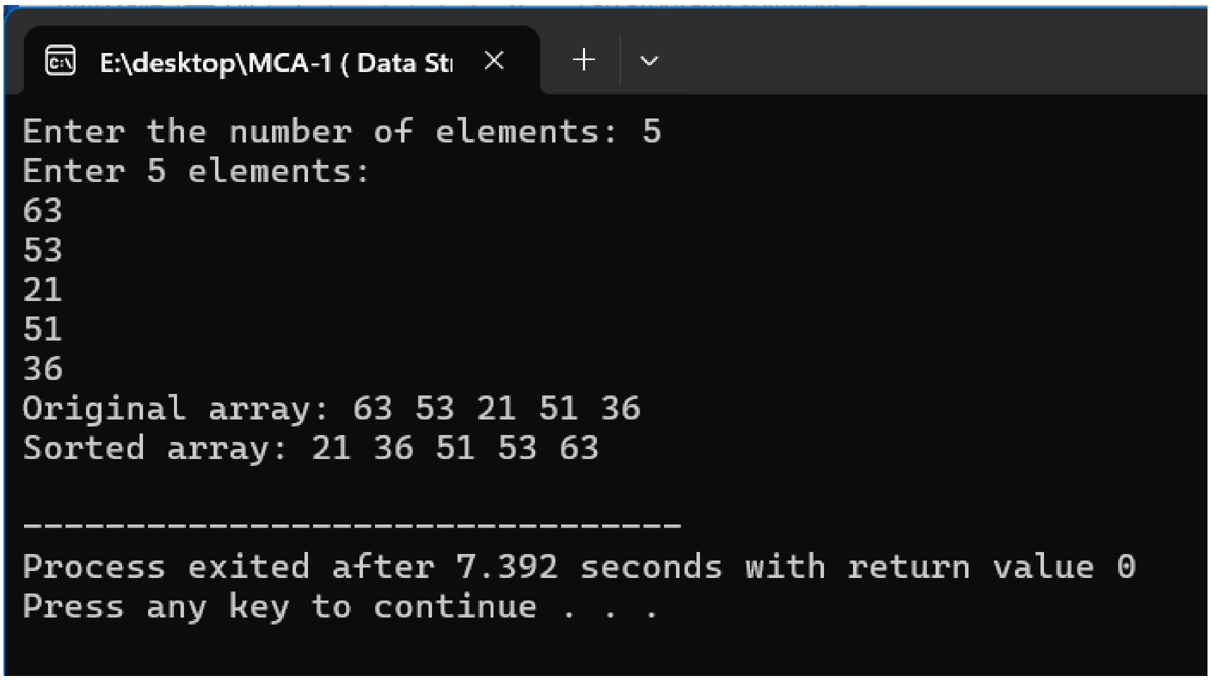


```
Enter the number of elements: 5
Enter 5 elements:
68
95
12
34
75
Original array: 68 95 12 34 75
Sorted array: 12 34 68 75 95

--------------------------------
Process exited after 6.783 seconds with return value 0
Press any key to continue . . .
```

16.Write a C++ program for implementing the Heap Sort.

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

// Function to heapify a subtree rooted with node i which is an index in arr[].
// n is the size of the heap
void heapify(int arr[], int n, int i) {
    int largest = i;    // Initialize largest as root
    int left = 2 * i + 1;    // Left child
    int right = 2 * i + 2;    // Right child

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

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

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Function to perform heap sort
void heapSort(int arr[], int n) {
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter " << n << " elements:" << endl;
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Original array: ";
    printArray(arr, n);

    heapSort(arr, n);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Output:



```
Enter the number of elements: 5
Enter 5 elements:
63
53
21
51
36
Original array: 63 53 21 51 36
Sorted array: 21 36 51 53 63

--------------------------------
Process exited after 7.392 seconds with return value 0
Press any key to continue . . .
```

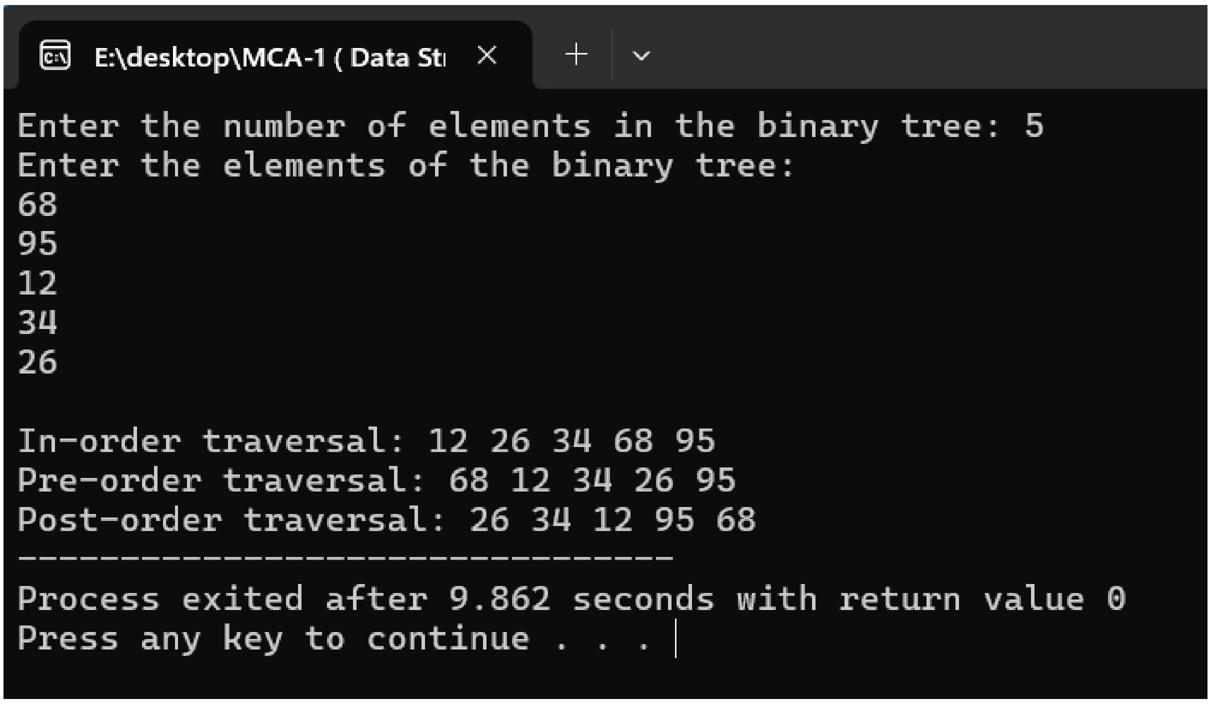
17. Write a C++ program that use recursive function to traverse the given binary tree in.

a. Pre-order, b. In order, c. post-order

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

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

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

// Function to insert a new node with the given key
Node* insert(Node* root, int key) {
   if (root == NULL) {
      return newNode(key);
   }

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

   return root;
}

// Function to perform in-order traversal of the tree
void inOrder(Node* root) {
   if (root == NULL) {
      return;
   }
   inOrder(root->left);
   cout << root->data << " ";
   inOrder(root->right);
}

// Function to perform pre-order traversal of the tree
void preOrder(Node* root) {
   if (root == NULL) {
      return;
   }
   cout << root->data << " ";
   preOrder(root->left);
```

```
    preOrder(root->right);
}

// Function to perform post-order traversal of the tree
void postOrder(Node* root) {
    if (root == NULL) {
        return;
    }
    postOrder(root->left);
    postOrder(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = NULL;

    int n, data;
    cout << "Enter the number of elements in the binary tree: ";
    cin >> n;

    cout << "Enter the elements of the binary tree:" << endl;
    for (int i = 0; i < n; i++) {
        cin >> data;
        root = insert(root, data);
    }

    cout << "\nIn-order traversal: ";
    inOrder(root);

    cout << "\nPre-order traversal: ";
    preOrder(root);

    cout << "\nPost-order traversal: ";
    postOrder(root);

    return 0;
}
```

Output:



```
Enter the number of elements in the binary tree: 5
Enter the elements of the binary tree:
68
95
12
34
26

In-order traversal: 12 26 34 68 95
Pre-order traversal: 68 12 34 26 95
Post-order traversal: 26 34 12 95 68
--------------------------------
Process exited after 9.862 seconds with return value 0
Press any key to continue . . .
```

18.Write a C++ program to perform the following operations.

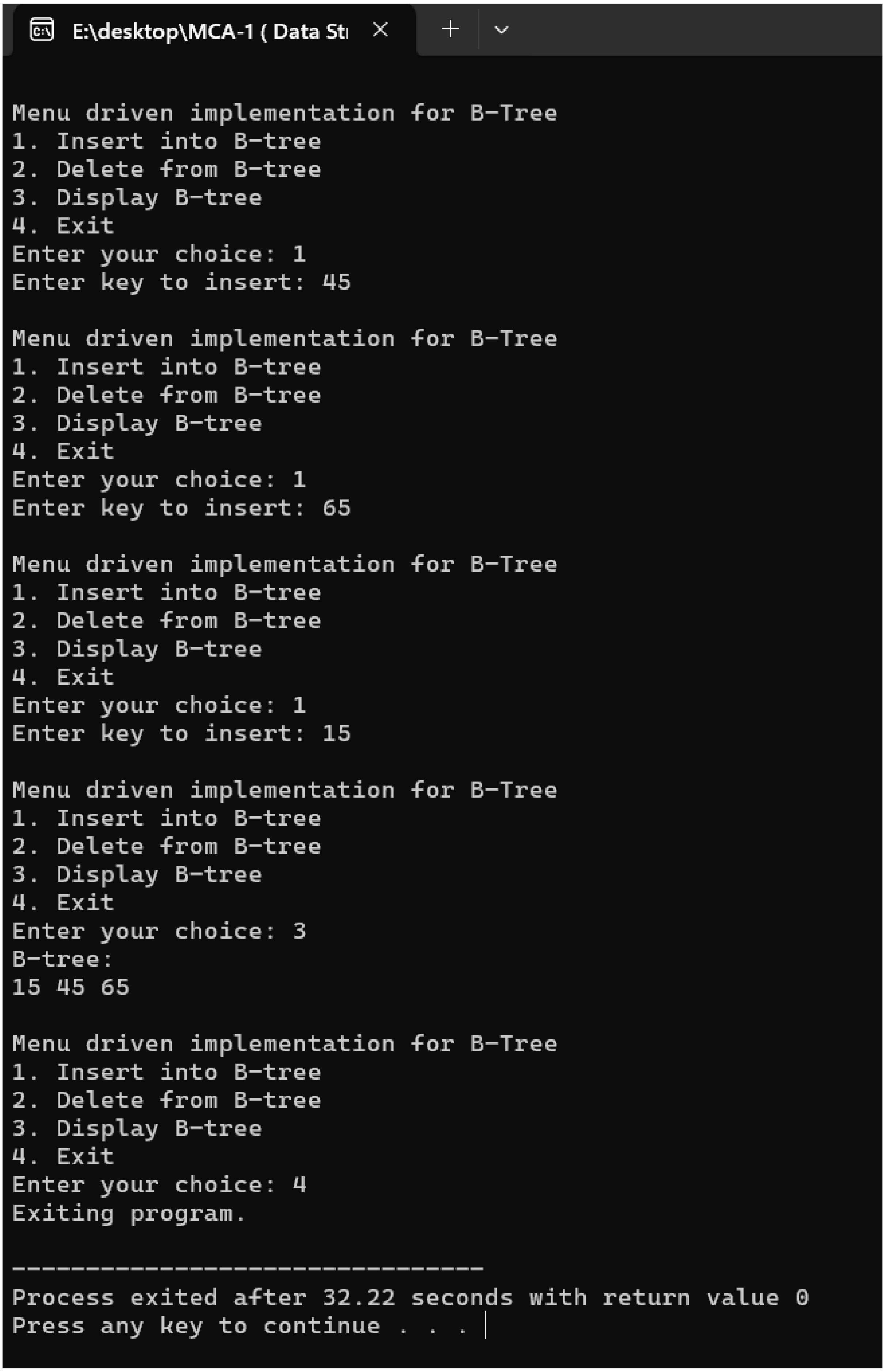
a. Insertion into a B-tree.

b. Deletion into a B-tree.

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

const int MAX_KEYS = 3; // Maximum number of keys in a node
const int MAX_CHILDREN = MAX_KEYS + 1; // Maximum number of children in a node

// Node structure for the B-tree
struct Node {
   vector<int> keys; // Vector to store keys
   vector<Node*> children; // Vector to store pointers to child nodes
   bool isLeaf; // Flag to indicate if the node is a leaf

   Node(bool leaf) : isLeaf(leaf) {} // Constructor to initialize a node
};

class BTree {
private:
   Node* root; // Pointer to the root node of the tree

         // Function to create a new node
   Node* createNode(bool leaf) {
      return new Node(leaf);
   }
         // Function to split a child node
   void splitChild(Node* parent, int index, Node* child) {
      Node* newChild = createNode(child->isLeaf);
      newChild->keys.assign(child->keys.begin() + MAX_KEYS / 2 + 1, child->keys.end());
      child->keys.erase(child->keys.begin() + MAX_KEYS / 2, child->keys.end());

      if (!child->isLeaf) {
        newChild->children.assign(child->children.begin() + MAX_CHILDREN / 2, child->children.end());
        child->children.erase(child->children.begin() + MAX_CHILDREN / 2, child->children.end());
     }

      parent->keys.insert(parent->keys.begin() + index, child->keys[MAX_KEYS / 2]);
      parent->children.insert(parent->children.begin() + index + 1, newChild);
   }
 // Function to insert a key into a non-full node
   void insertNonFull(Node* node, int key) {
      int i = node->keys.size() - 1;

      if (node->isLeaf) {
         node->keys.push_back(0);
         while (i >= 0 && key < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
         }
         node->keys[i + 1] = key;
      } else {
         while (i >= 0 && key < node->keys[i]) {
            i--;
        }

        i++;
        if (node->children[i]->keys.size() == MAX_KEYS) {
            splitChild(node, i, node->children[i]);
            if (key > node->keys[i]) {
                i++;
            }
        }
        insertNonFull(node->children[i], key);
    }
}

// Function to merge a child node with its sibling
void merge(Node* parent, int index) {
    Node* child = parent->children[index];
    Node* sibling = parent->children[index + 1];

    child->keys.push_back(parent->keys[index]);
    child->keys.insert(child->keys.end(), sibling->keys.begin(), sibling->keys.end());
    parent->keys.erase(parent->keys.begin() + index);
    parent->children.erase(parent->children.begin() + index + 1);

    if (!child->isLeaf) {
        child->children.insert(child->children.end(),     sibling->children.begin(),     sibling-
>children.end());
    }

    delete sibling;
}

// Function to borrow a key from the previous sibling
void borrowFromPrev(Node* parent, int index) {
    Node* child = parent->children[index];
    Node* sibling = parent->children[index - 1];

    child->keys.insert(child->keys.begin(), parent->keys[index - 1]);
    parent->keys[index - 1] = sibling->keys.back();
    sibling->keys.pop_back();

    if (!child->isLeaf) {
        child->children.insert(child->children.begin(), sibling->children.back());
        sibling->children.pop_back();
    }
}

// Function to borrow a key from the next sibling
void borrowFromNext(Node* parent, int index) {
    Node* child = parent->children[index];
    Node* sibling = parent->children[index + 1];
    child->keys.push_back(parent->keys[index]);
    parent->keys[index] = sibling->keys[0];
    sibling->keys.erase(sibling->keys.begin());
```

```
        if (!child->isLeaf) {
            child->children.push_back(sibling->children.front());
            sibling->children.erase(sibling->children.begin());
        }
    }

// Function to remove a key from a leaf node
    void removeFromLeaf(Node* node, int index) {
        node->keys.erase(node->keys.begin() + index);
    }

// Function to remove a key from a non-leaf node
    void removeFromNonLeaf(Node* node, int index) {
        int key = node->keys[index];

        if (node->children[index]->keys.size() >= MAX_KEYS) {
            Node* pred = node->children[index];
            while (!pred->isLeaf) {
                pred = pred->children.back();
            }
            node->keys[index] = pred->keys.back();
            pred->keys.pop_back();
        } else if (node->children[index + 1]->keys.size() >= MAX_KEYS) {
            Node* succ = node->children[index + 1];
            while (!succ->isLeaf) {
                succ = succ->children.front();
            }
            node->keys[index] = succ->keys.front();
            succ->keys.erase(succ->keys.begin());
        } else {
            merge(node, index);
        }
    }

 // Helper function to remove a key from the tree
    void removeHelper(Node* node, int key) {
        int index = 0;
        while (index < node->keys.size() && key > node->keys[index]) {
            index++;
        }

        if (index < node->keys.size() && key == node->keys[index]) {
            if (node->isLeaf) {
                removeFromLeaf(node, index);
            } else {
                removeFromNonLeaf(node, index);
            }
        } else {
            Node* child = node->children[index];
            if (child->keys.size() < MAX_KEYS) {
                int siblingIndex = (index == node->keys.size()) ? index - 1 : index + 1;
                Node* sibling = node->children[siblingIndex];

                if (sibling->keys.size() >= MAX_KEYS) {
                    if (siblingIndex < index) {
                        borrowFromPrev(node, index);
                    } else {
                        borrowFromNext(node, index);
                    }
                } else {
                    if (siblingIndex < index) {
                        merge(node, index - 1);
                    } else {
                        merge(node, index);
                    }
                }
            }
            removeHelper(child, key);
        }
    }

public:
    BTree() : root(NULL) {} // Constructor to initialize an empty tree

// Function to insert a key into the tree
    void insert(int key) {
        if (root == NULL) {
            root = createNode(true);
            root->keys.push_back(key);
        } else {
            if (root->keys.size() == MAX_KEYS) {
                Node* newRoot = createNode(false);
                newRoot->children.push_back(root);
                splitChild(newRoot, 0, root);
                root = newRoot;
            }
            insertNonFull(root, key);
        }
    }

// Function to display the tree
    void display(Node* node) {
        if (node != NULL) {
            for (int i = 0; i < node->keys.size(); i++) {
                cout << node->keys[i] << " ";
            }
            cout << endl;
            if (!node->isLeaf) {
                for (int i = 0; i < node->children.size(); i++) {
                    display(node->children[i]);
                }
            }
        }
    }
// Function to display the tree (wrapper function)
    void display() {
        if (root != NULL) {
            display(root);
        } else {
            cout << "B-tree is empty." << endl;
        }
    }
    // Function to remove a key from the tree
    void remove(int key) {
        if (root == NULL) {
            cout << "B-tree is empty." << endl;
            return;
        }
        removeHelper(root, key);
        if (root->keys.empty()) {
            Node* tmp = root;
            if (root->isLeaf) {
                root = NULL;
            } else {
                root = root->children[0];
            }
            delete tmp;
        }
    }
};
int main() {
    BTree btree; // Create a B-tree object
    int choice, key;
// Menu-driven interface for interacting with the B-tree
    do {
        cout << "\nMenu driven implementation for B-Tree\n";
        cout << "1. Insert into B-tree\n";
        cout << "2. Delete from B-tree\n";
        cout << "3. Display B-tree\n";
        cout << "4. Exit\n";
        cout << "Enter your choice: ";
        cin >> choice;
        switch (choice) {
            case 1:
                cout << "Enter key to insert: ";
                cin >> key;
                btree.insert(key); // Insert key into the B-tree
                break;
            case 2:
                cout << "Enter key to delete: ";
                cin >> key;
                btree.remove(key); // Remove key from the B-tree
                break;
            case 3:
                cout << "B-tree:" << endl;
                btree.display(); // Display the B-tree
                break;
            case 4:
                cout << "Exiting program." << endl;
                break;
            default:
                cout << "Invalid choice. Please try again." << endl;
                break;
        }
    } while (choice != 4);

    return 0;}
```

Output:



```

Menu driven implementation for B-Tree
1. Insert into B-tree
2. Delete from B-tree
3. Display B-tree
4. Exit
Enter your choice: 1
Enter key to insert: 45

Menu driven implementation for B-Tree
1. Insert into B-tree
2. Delete from B-tree
3. Display B-tree
4. Exit
Enter your choice: 1
Enter key to insert: 65

Menu driven implementation for B-Tree
1. Insert into B-tree
2. Delete from B-tree
3. Display B-tree
4. Exit
Enter your choice: 1
Enter key to insert: 15

Menu driven implementation for B-Tree
1. Insert into B-tree
2. Delete from B-tree
3. Display B-tree
4. Exit
Enter your choice: 3
B-tree:
15 45 65

Menu driven implementation for B-Tree
1. Insert into B-tree
2. Delete from B-tree
3. Display B-tree
4. Exit
Enter your choice: 4
Exiting program.

--------------------------------
Process exited after 32.22 seconds with return value 0
Press any key to continue . . .
```

19.Write a C++ program to perform the following operations.

a. Insertion into AVL tree.

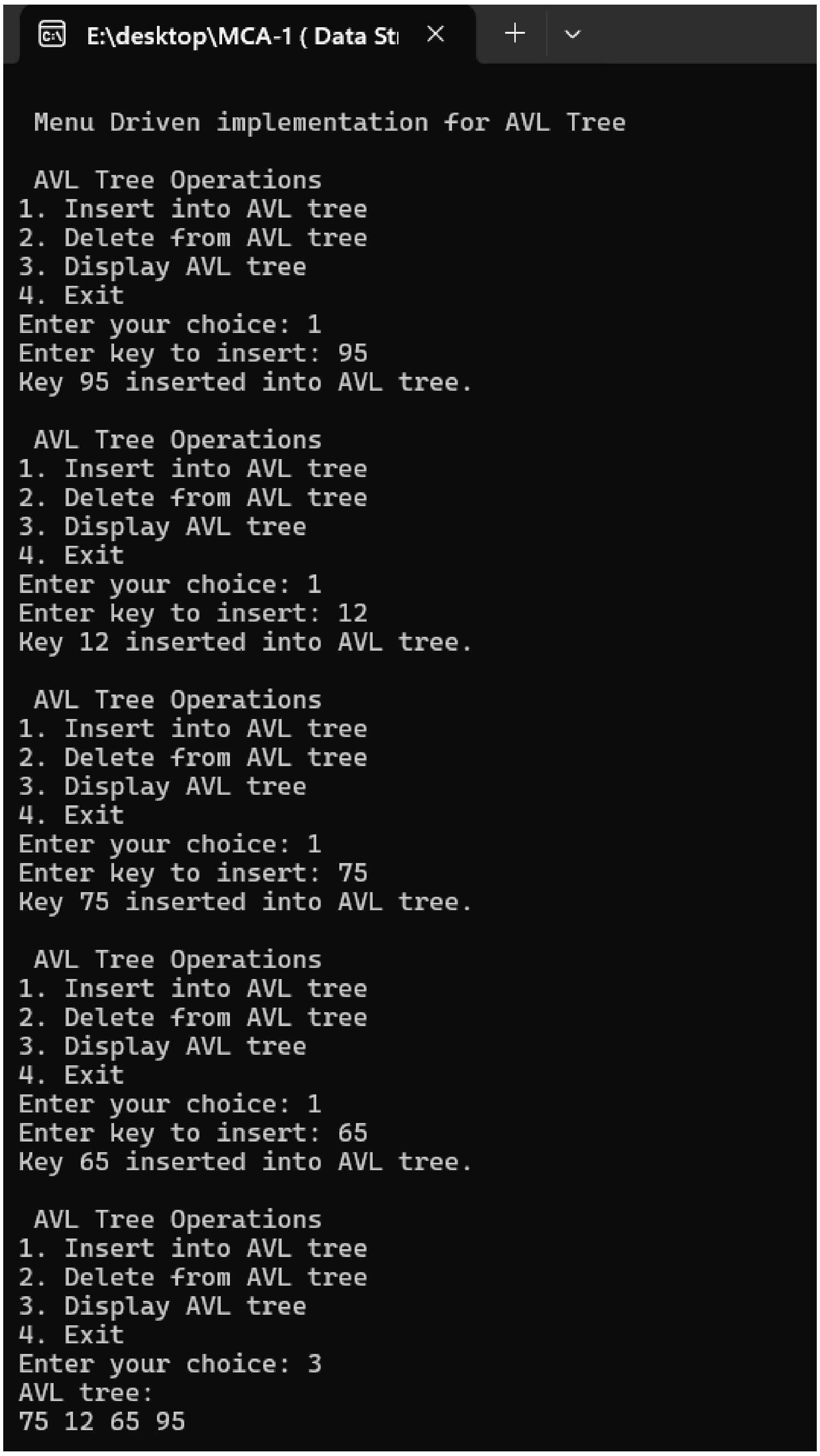
b. Deletion from a AVL tree.

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

// Node structure for AVL tree
struct Node {
    int key;
    Node* left;
    Node* right;
    int height;
};

// Function to calculate maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to get height of a node
int height(Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to create a new node with given key
Node* newNode(int key) {
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return node;
}

// Function to right rotate subtree rooted with y
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    // Return new root
    return x;
}
```

```
// Function to left rotate subtree rooted with x
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to insert a key into AVL tree
Node* insert(Node* node, int key) {
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);
    else // Equal keys are not allowed in BST
        return node;

    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left), height(node->right));

    /* 3. Get the balance factor of this ancestor
          node to check whether this node became
          unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there
    // are 4 cases
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

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

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

    /* return the (unchanged) node pointer */
    return node;
}

// Function to find the node with minimum key value
Node* minValueNode(Node* node) {
    Node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

// Function to delete a key from AVL tree
Node* deleteNode(Node* root, int key) {
    // STEP 1: PERFORM STANDARD BST DELETE
    if (root == NULL)
        return root;

    // If the key to be deleted is smaller than the
    // root's key, then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the
    // root's key, then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    // if key is same as root's key, then this is the node
    // to be deleted
    else {
        // node with only one child or no child
        if ((root->left == NULL) || (root->right == NULL)) {
            Node* temp = root->left ? root->left : root->right;

            // No child case
            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else // One child case
                *root = *temp; // Copy the contents of the non-empty child
            delete temp;
        } else {
            // node with two children: Get the inorder
            // successor (smallest in the right subtree)
            Node* temp = minValueNode(root->right);

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

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;

    // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
    root->height = 1 + max(height(root->left), height(root->right));

    // STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
    // this node became unbalanced)
    int balance = getBalance(root);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);

    // Left Right Case
    if (balance > 1 && getBalance(root->left) < 0) {
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    // Right Right Case
    if (balance < -1 && getBalance(root->right) <= 0)
        return leftRotate(root);

    // Right Left Case
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

// Function to traverse the AVL tree in preorder
void preOrder(Node* root) {
    if (root != NULL) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

```
// Main function
int main() {
   Node* root = NULL;
   int choice, key;
cout << "\n Menu Driven implementation for AVL Tree\n";
   do {
      cout << "\n AVL Tree Operations\n";
      cout << "1. Insert into AVL tree\n";
      cout << "2. Delete from AVL tree\n";
      cout << "3. Display AVL tree\n";
      cout << "4. Exit\n";
      cout << "Enter your choice: ";
      cin >> choice;

      switch (choice) {
         case 1:
            cout << "Enter key to insert: ";
            cin >> key;
            root = insert(root, key);
            cout << "Key " << key << " inserted into AVL tree." << endl;
            break;
         case 2:
            cout << "Enter key to delete: ";
            cin >> key;
            root = deleteNode(root, key);
            cout << "Key " << key << " deleted from AVL tree." << endl;
            break;
         case 3:
            cout << "AVL tree:" << endl;
            preOrder(root);
            cout << endl;
            break;
         case 4:
            cout << "Exiting program." << endl;
            break;
         default:
            cout << "Invalid choice. Please try again." << endl;
            break;
      }
   } while (choice != 4);

   return 0;
}
```

Output:



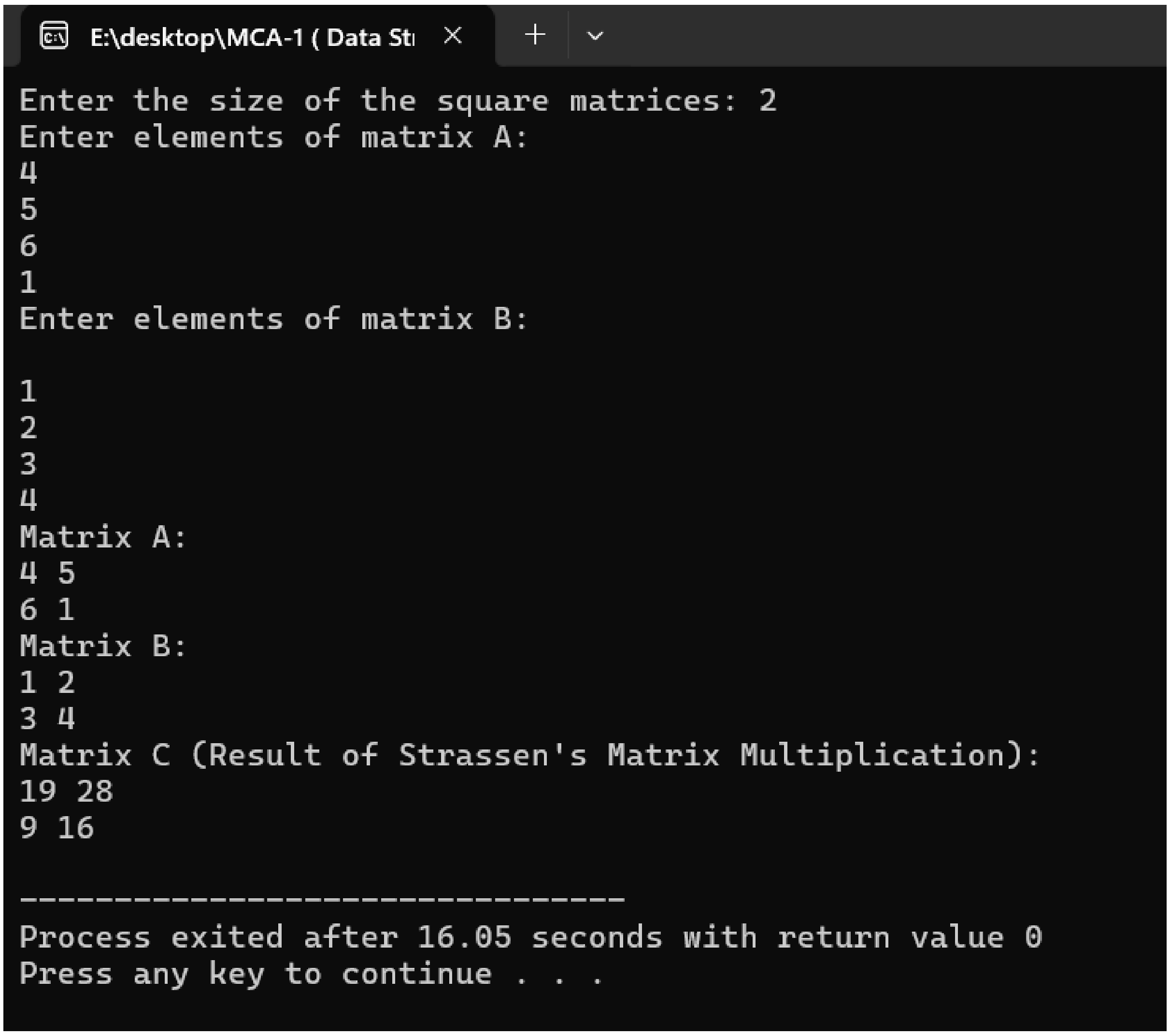
```
 Menu Driven implementation for AVL Tree

 AVL Tree Operations
1. Insert into AVL tree
2. Delete from AVL tree
3. Display AVL tree
4. Exit
Enter your choice: 1
Enter key to insert: 95
Key 95 inserted into AVL tree.

 AVL Tree Operations
1. Insert into AVL tree
2. Delete from AVL tree
3. Display AVL tree
4. Exit
Enter your choice: 1
Enter key to insert: 12
Key 12 inserted into AVL tree.

 AVL Tree Operations
1. Insert into AVL tree
2. Delete from AVL tree
3. Display AVL tree
4. Exit
Enter your choice: 1
Enter key to insert: 75
Key 75 inserted into AVL tree.

 AVL Tree Operations
1. Insert into AVL tree
2. Delete from AVL tree
3. Display AVL tree
4. Exit
Enter your choice: 1
Enter key to insert: 65
Key 65 inserted into AVL tree.

 AVL Tree Operations
1. Insert into AVL tree
2. Delete from AVL tree
3. Display AVL tree
4. Exit
Enter your choice: 3
AVL tree:
75 12 65 95
```

20.Write a C++ program to implement Strassen’s matrix multiplication.

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

// Function to add two matrices
vector<vector<int> > matrixAddition(const vector<vector<int> >& A, const
vector<vector<int> >& B) {
   int n = A.size();
   vector<vector<int> > C(n, vector<int>(n, 0));
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
   return C;
}

// Function to subtract two matrices
vector<vector<int> > matrixSubtraction(const vector<vector<int> >& A, const
vector<vector<int> >& B) {
   int n = A.size();
   vector<vector<int> > C(n, vector<int>(n, 0));

   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         C[i][j] = A[i][j] - B[i][j];
      }
   }

   return C;
}

// Function to multiply two matrices using Strassen's algorithm
vector<vector<int> > strassenMatrixMultiplication(const vector<vector<int> >& A, const
vector<vector<int> >& B) {
   int n = A.size();

   if (n == 1) {
      vector<vector<int> > C(1, vector<int>(1, 0));
      C[0][0] = A[0][0] * B[0][0];
      return C;
   }

   int newSize = n / 2;
   vector<vector<int> > A11(newSize, vector<int>(newSize));
   vector<vector<int> > A12(newSize, vector<int>(newSize));
   vector<vector<int> > A21(newSize, vector<int>(newSize));
   vector<vector<int> > A22(newSize, vector<int>(newSize));
   vector<vector<int> > B11(newSize, vector<int>(newSize));
   vector<vector<int> > B12(newSize, vector<int>(newSize));
   vector<vector<int> > B21(newSize, vector<int>(newSize));
   vector<vector<int> > B22(newSize, vector<int>(newSize));
```

```
    // Dividing matrices into submatrices
    for (int i = 0; i < newSize; ++i) {
        for (int j = 0; j < newSize; ++j) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + newSize];
            A21[i][j] = A[i + newSize][j];
            A22[i][j] = A[i + newSize][j + newSize];

            B11[i][j] = B[i][j];
            B12[i][j] = B[i][j + newSize];
            B21[i][j] = B[i + newSize][j];
            B22[i][j] = B[i + newSize][j + newSize];
        }
    }

    // Calculating the 10 required products
    vector<vector<int> > P1 = strassenMatrixMultiplication(matrixAddition(A11, A22),
matrixAddition(B11, B22));
    vector<vector<int> > P2 = strassenMatrixMultiplication(matrixAddition(A21, A22), B11);
    vector<vector<int> > P3 = strassenMatrixMultiplication(A11, matrixSubtraction(B12,
B22));
    vector<vector<int> > P4 = strassenMatrixMultiplication(A22, matrixSubtraction(B21,
B11));
    vector<vector<int> > P5 = strassenMatrixMultiplication(matrixAddition(A11, A12), B22);
    vector<vector<int> > P6 = strassenMatrixMultiplication(matrixSubtraction(A21, A11),
matrixAddition(B11, B12));
    vector<vector<int> > P7 = strassenMatrixMultiplication(matrixSubtraction(A12, A22),
matrixAddition(B21, B22));

    // Calculating the result submatrices
    vector<vector<int> > C11 = matrixAddition(matrixSubtraction(matrixAddition(P1, P4),
P5), P7);
    vector<vector<int> > C12 = matrixAddition(P3, P5);
    vector<vector<int> > C21 = matrixAddition(P2, P4);
    vector<vector<int> > C22 = matrixAddition(matrixAddition(matrixSubtraction(P1, P2),
P3), P6);

    // Combining the result submatrices into the final result matrix
    vector<vector<int> > C(n, vector<int>(n, 0));
    for (int i = 0; i < newSize; ++i) {
        for (int j = 0; j < newSize; ++j) {
            C[i][j] = C11[i][j];
            C[i][j + newSize] = C12[i][j];
            C[i + newSize][j] = C21[i][j];
            C[i + newSize][j + newSize] = C22[i][j];
        }
    }

    return C;
}

// Function to print a matrix
void printMatrix(const vector<vector<int> >& matrix) {
    int n = matrix.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    int n;
    cout << "Enter the size of the square matrices: ";
    cin >> n;

    vector<vector<int> > A(n, vector<int>(n));
    vector<vector<int> > B(n, vector<int>(n));

    cout << "Enter elements of matrix A:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> A[i][j];
        }
    }

    cout << "Enter elements of matrix B:" << endl;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            cin >> B[i][j];
        }
    }

    cout << "Matrix A:" << endl;
    printMatrix(A);

    cout << "Matrix B:" << endl;
    printMatrix(B);

    vector<vector<int> > C = strassenMatrixMultiplication(A, B);

    cout << "Matrix C (Result of Strassen's Matrix Multiplication):" << endl;
    printMatrix(C);

    return 0;
}
```

Output:

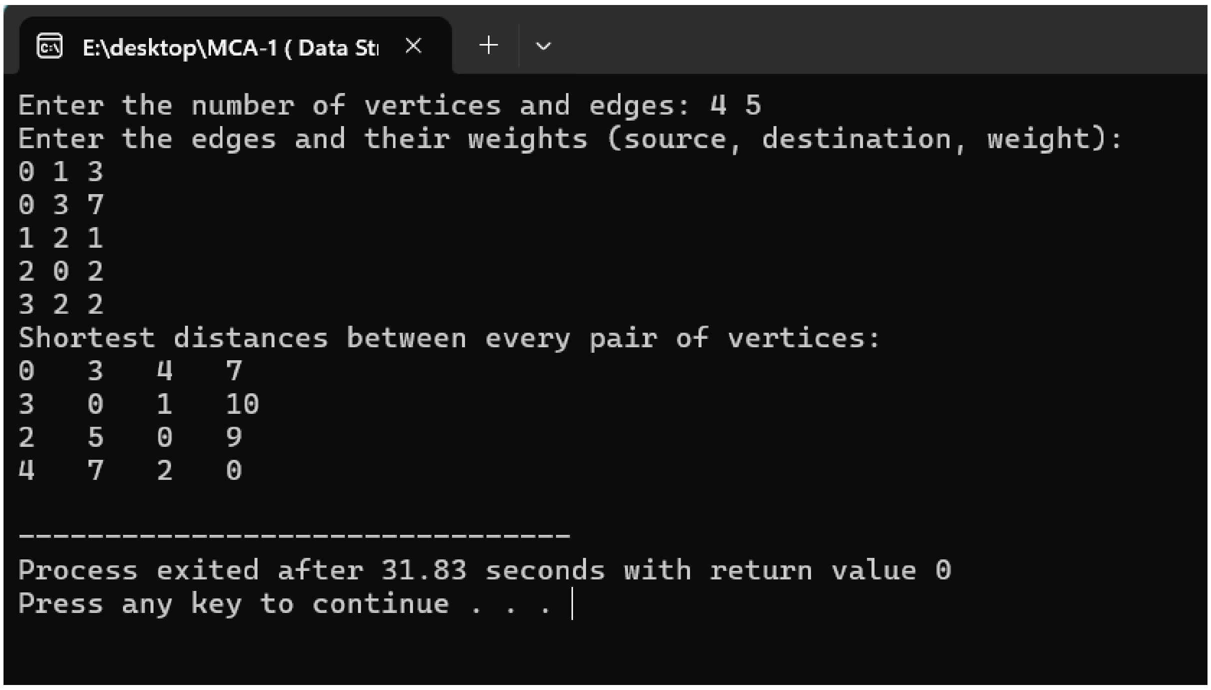


```
Enter the size of the square matrices: 2
Enter elements of matrix A:
4
5
6
1
Enter elements of matrix B:

1
2
3
4
Matrix A:
4 5
6 1
Matrix B:
1 2
3 4
Matrix C (Result of Strassen's Matrix Multiplication):
19 28
9 16

--------------------------------
Process exited after 16.05 seconds with return value 0
Press any key to continue . . .
```

21.Write a C++ program to implement Floyd’s Warshall’s Algorithm.

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
// Function to print the distance matrix
void printMatrix(const vector<vector<int> >& dist) {
    int V = dist.size();
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (dist[i][j] == INT_MAX)
                cout << "INF ";
            else
                cout << dist[i][j] << "   ";
        }
        cout << endl;
    }
}

// Function to implement Floyd-Warshall's algorithm
void floydWarshall(const vector<vector<int> >& graph) {
    int V = graph.size();
    vector<vector<int> > dist(graph);

    // Apply Floyd-Warshall's algorithm
    for (int k = 0; k < V; ++k) {
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
  if (dist[i][k] != INT_MAX && dist[k][j] != INT_MAX && dist[i][k] + dist[k][j] <
dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Check for negative cycles
    for (int i = 0; i < V; ++i) {
        if (dist[i][i] < 0) {
            cout << "Graph contains negative weight cycle" << endl;
            return;
        }
    }

    // Print the shortest distances
    cout << "Shortest distances between every pair of vertices:" << endl;
    printMatrix(dist);
}

int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;
    vector<vector<int> > graph(V, vector<int>(V, INT_MAX));
    cout << "Enter the edges and their weights (source, destination, weight):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v, w;
        cin >> u >> v >> w;
        if (u >= V || v >= V || u < 0 || v < 0) {

            cout << "Invalid edge input. Please enter valid vertices between 0 and " << V-1 << "."
            << endl;
            --i; // decrement i to re-enter this edge
            continue;
        }
        graph[u][v] = w;
    }
    // Setting the diagonal elements to 0
    for (int i = 0; i < V; ++i) {
        graph[i][i] = 0;
    }

    floydWarshall(graph);

    return 0;
}
```

Output:



E:\desktop\MCA-1 (Data St

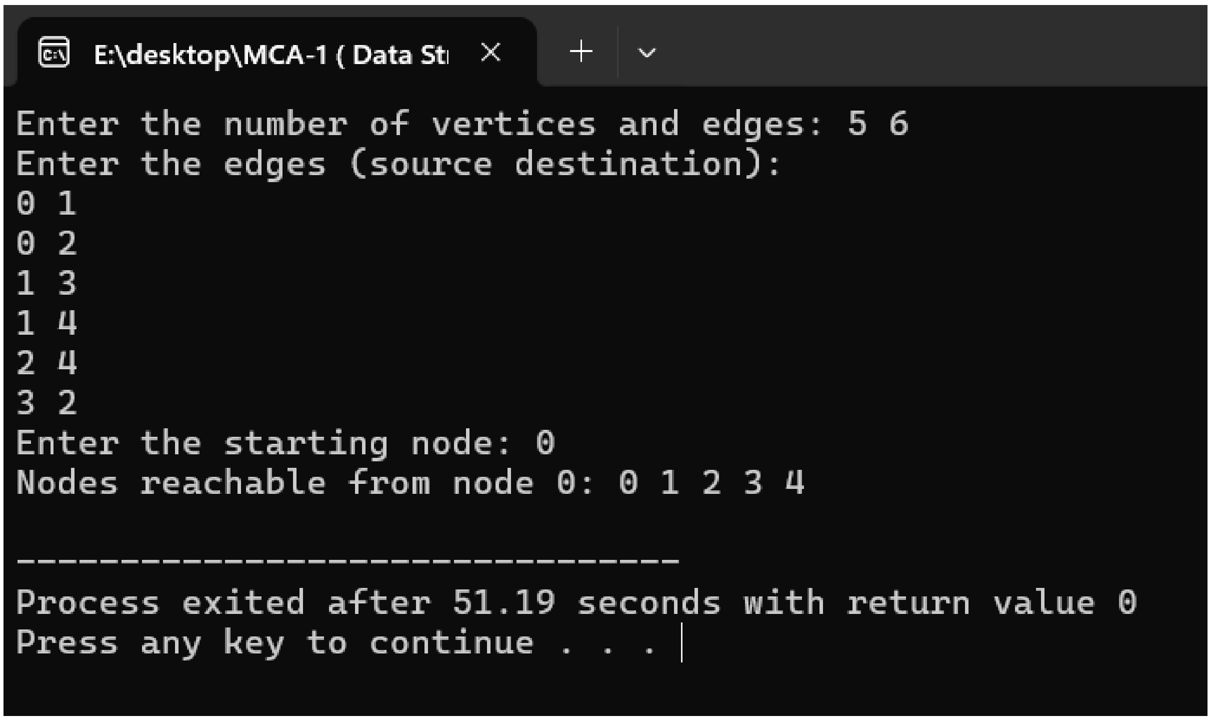
```
Enter the number of vertices and edges: 4 5
Enter the edges and their weights (source, destination, weight):
0 1 3
0 3 7
1 2 1
2 0 2
3 2 2
Shortest distances between every pair of vertices:
0   3   4   7
3   0   1   10
2   5   0   9
4   7   2   0

--------------------------------
Process exited after 31.83 seconds with return value 0
Press any key to continue . . .
```

22. Write a C++ program to print all the nodes reachable from a given starting node in a diagram using BFS method.

```
#include <iostream>

#include <vector>

#include <queue>

using namespace std;

// Function to perform BFS and print reachable nodes from the given start node

void BFS(const vector<vector<int> >& adjList, int startNode) {

    int n = adjList.size();

    vector<bool> visited(n, false);

    queue<int> q;


    // Start BFS from the startNode

    visited[startNode] = true;

    q.push(startNode);

    cout << "Nodes reachable from node " << startNode << ": ";

     while (!q.empty()) {

        int node = q.front();

        q.pop();

        cout << node << " ";


        // Traverse all adjacent nodes

        for (size_t i = 0; i < adjList[node].size(); ++i) {

            int neighbor = adjList[node][i];

            if (!visited[neighbor]) {

                visited[neighbor] = true;

                q.push(neighbor);

            }

        }

    }

    cout << endl;

}
```

```
int main() {
    int V, E;
    cout << "Enter the number of vertices and edges: ";
    cin >> V >> E;
    vector<vector<int> > adjList(V);
    cout << "Enter the edges (source destination):" << endl;
    for (int i = 0; i < E; ++i) {
        int u, v;
        cin >> u >> v;
        adjList[u].push_back(v);
        // For an undirected graph, also add the reverse edge:
        // adjList[v].push_back(u);
    }
    int startNode;
    cout << "Enter the starting node: ";
    cin >> startNode;
    if (startNode >= 0 && startNode < V) {
        BFS(adjList, startNode);
    } else {
        cout << "Invalid starting node." << endl;
    }
    return 0;
}
```

Output:



E:\desktop\MCA-1 (Data St

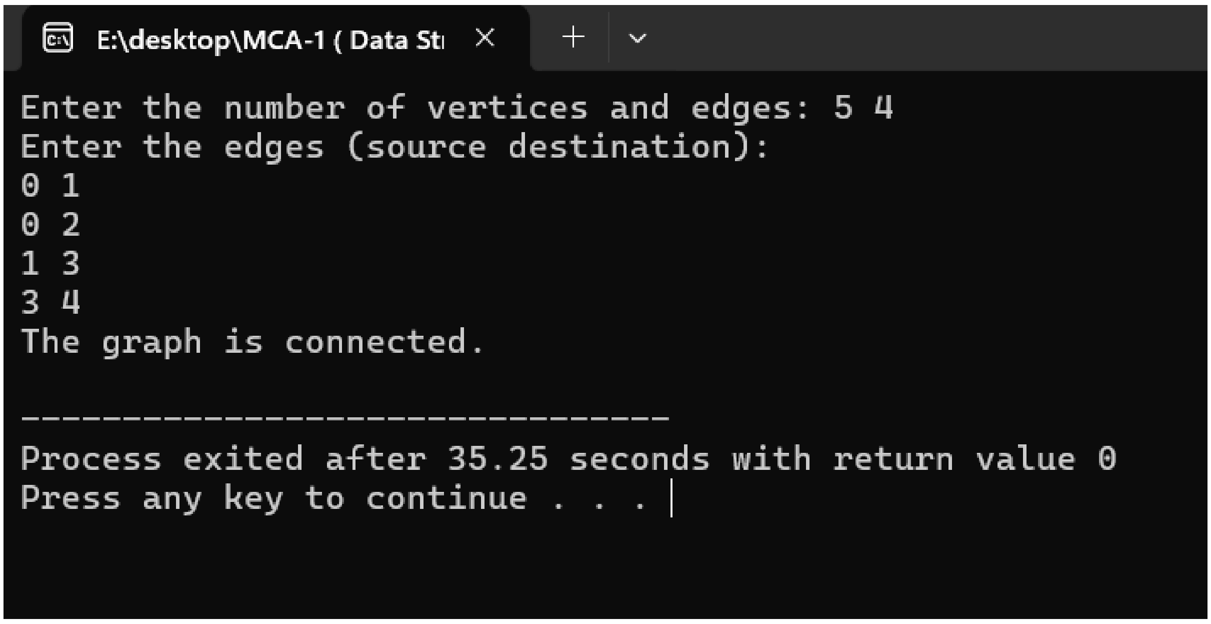
```
Enter the number of vertices and edges: 5 6
Enter the edges (source destination):
0 1
0 2
1 3
1 4
2 4
3 2
Enter the starting node: 0
Nodes reachable from node 0: 0 1 2 3 4

--------------------------------
Process exited after 51.19 seconds with return value 0
Press any key to continue . . .
```

23. Write a C++ program to check whether a given graph is connected or not using DFS method.

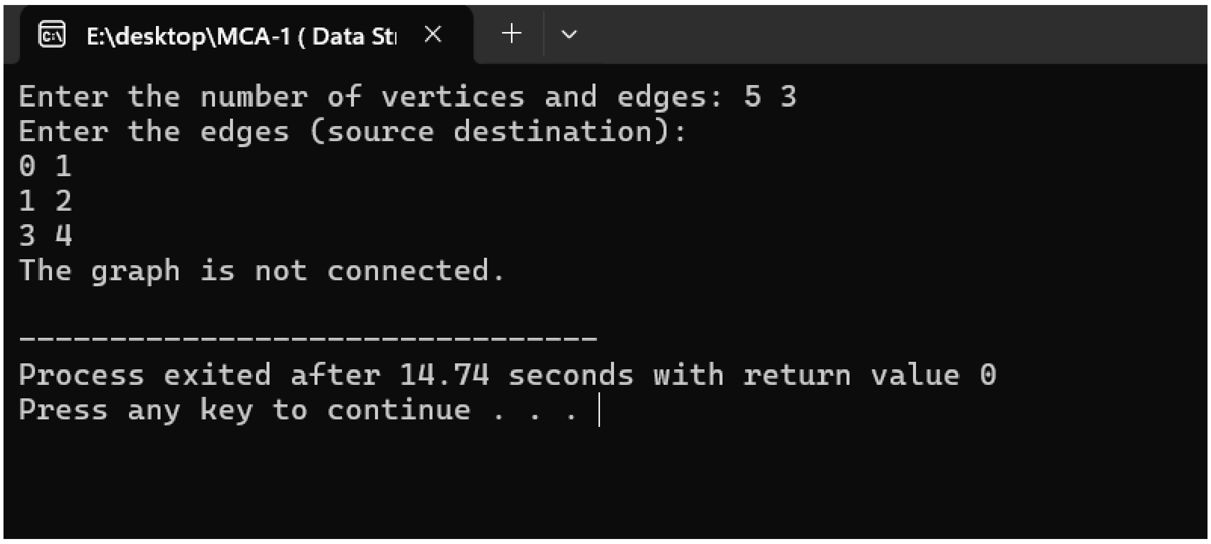
```
#include <iostream>

#include <vector>

using namespace std;


// Function to perform DFS

void DFS(const vector<vector<int> >& adjList, vector<bool>& visited, int node) {

    visited[node] = true;

    for (size_t i = 0; i < adjList[node].size(); ++i) {

        int neighbor = adjList[node][i];

        if (!visited[neighbor]) {

            DFS(adjList, visited, neighbor);

        }

    }

}


// Function to check if the graph is connected

bool isConnected(const vector<vector<int> >& adjList) {

    int V = adjList.size();

    vector<bool> visited(V, false);


    // Start DFS from the first node (assuming the graph is 0-indexed)

    DFS(adjList, visited, 0);


    // Check if all vertices are visited

    for (int i = 0; i < V; ++i) {

        if (!visited[i]) {

            return false;

        }

    }

    return true;

}
```

```
int main() {
   int V, E;
   cout << "Enter the number of vertices and edges: ";
   cin >> V >> E;
   vector<vector<int> > adjList(V);
   cout << "Enter the edges (source destination):" << endl;
   for (int i = 0; i < E; ++i) {
      int u, v;
      cin >> u >> v;
      adjList[u].push_back(v);
      adjList[v].push_back(u); // For an undirected graph
   }

   if (isConnected(adjList)) {
      cout << "The graph is connected." << endl;
   } else {
      cout << "The graph is not connected." << endl;
   }
   return 0;
}
```

Output:



```
Enter the number of vertices and edges: 5 4
Enter the edges (source destination):
0 1
0 2
1 3
3 4
The graph is connected.

--------------------------------
Process exited after 35.25 seconds with return value 0
Press any key to continue . . .
```

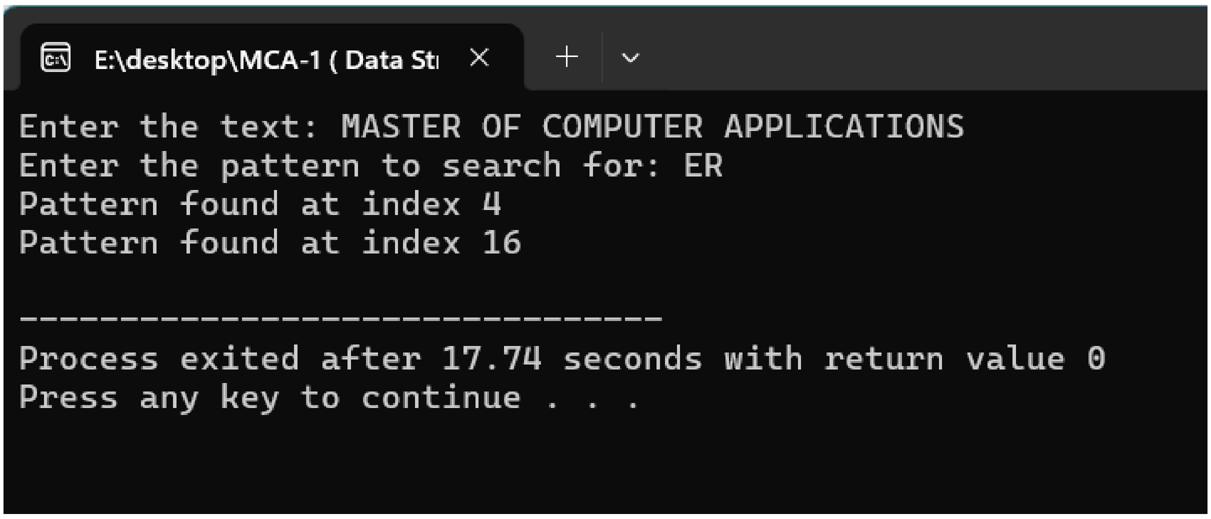


```
Enter the number of vertices and edges: 5 3
Enter the edges (source destination):
0 1
1 2
3 4
The graph is not connected.

--------------------------------
Process exited after 14.74 seconds with return value 0
Press any key to continue . . .
```

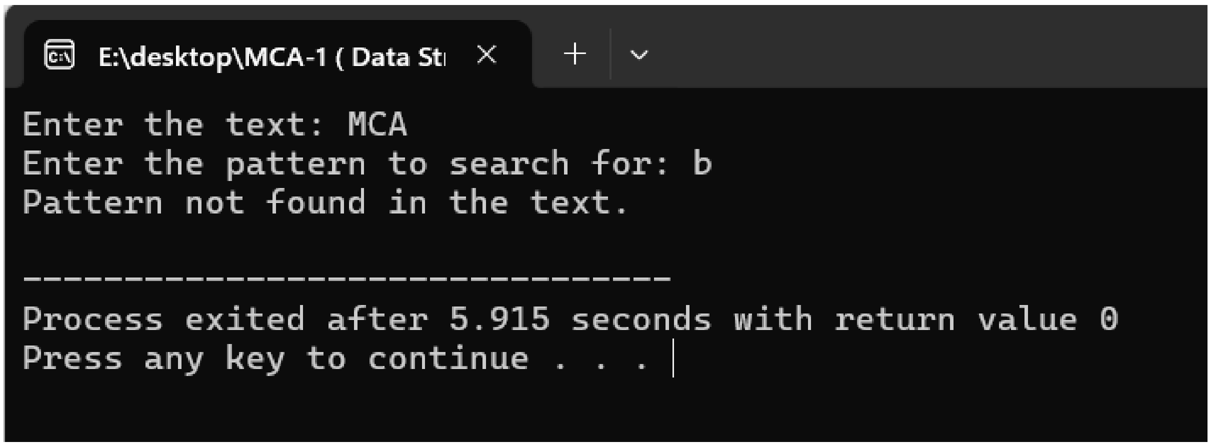
24. Write a C++ program to implement Brute Force String Matching algorithm.

```cpp
#include <iostream>

#include <string>

using namespace std;

// Function to perform Brute Force String Matching

void bruteForceStringMatch(const string& text, const string& pattern) {

    int n = text.length();

    int m = pattern.length();

    int count = 0;

    for (int i = 0; i <= n - m; ++i) {

        int j;

        for (j = 0; j < m; ++j) {

            if (text[i + j] != pattern[j])

                break;

        }

        if (j == m) {

            count++;

            cout << "Pattern found at index " << i << endl;

        }

    }

    if (count == 0) {

        cout << "Pattern not found in the text." << endl;

    } else {

        cout << "Pattern found " << count << " time(s) in the text." << endl;

    }

}


int main() {

    string text, pattern;

    cout << "Enter the text: ";

    getline(cin, text);
    cout << "Enter the pattern to search for: ";
    getline(cin, pattern);
    bruteForceStringMatch(text, pattern);
    return 0;
}
```

Output:



```
Enter the text: MASTER OF COMPUTER APPLICATIONS
Enter the pattern to search for: ER
Pattern found at index 4
Pattern found at index 16

--------------------------------
Process exited after 17.74 seconds with return value 0
Press any key to continue . . .
```



```
Enter the text: MCA
Enter the pattern to search for: b
Pattern not found in the text.

--------------------------------
Process exited after 5.915 seconds with return value 0
Press any key to continue . . .
```