The University of Faisalabad

Name: Hafiza Tehreem Fatima

Registration: 2023-bs-ai-026

Submitted to: Miss Irsha Qureshi

Department: Artificial Intelligence

Course Code: CS - 216

Table of Contents

¦LAB -【Introduction】¦

VARIABLES :

Variables are fundamental building blocks in C++ programming, used to store data that can be modified and accessed throughout a program.

Types of variables :

C++ supports various data types for variables, each serving a specific purpose:

1. **int**: Used for integers (whole numbers).
2. **float**: Used for floating-point numbers (numbers with decimals).
3. **double**: Like float but with double precision.
4. **char**: Used for single characters.
5. **string**: Used for text (requires the #include <string> header).

- **Declaration:** To declare a variable in C++, you need to specify the type of the variable followed by its name. The type determines the kind of data the variable can hold.
- **Initialization:** Variables can be initialized at the time of declaration or later in the code. Initialization can be done using the assignment operator " = ".

Syntax:

```cpp
int age = 25; // Integer variable storing 25
```

FUNCTIONS :

Functions are blocks of code that perform a specific task and can be reused throughout a program. They help in organizing code, making it more readable, and reducing redundancy.

- ***Declaraton:***

A function declaration (or prototype) tells the compiler about the function's name, return type, and parameters. It does not contain the actual body of the function.

Syntax:

return_type function_name(parameter_list);

- **Defination:**

A function definition contains the actual body of the function, which includes the statements that perform the task.

Syntax:

```
return_type function_name(parameter_list)
{
// Function's body…
}
```

POINTERS :

Pointers are variables that store the memory address of another variable. They are powerful tools that allow for direct memory access and manipulation, which can lead to more efficient code.

Key Concepts:

- **Declaration :** A pointer is declared by placing an asterisk (*) before the pointer variable's name.

Syntax:

```cpp
int* ptr; // Pointer to an integer
```

- **Initialization :** A pointer is initialized by assigning it the address of a variable using the address-of operator (&).

```cpp
int num = 10;
int* ptr = &num; // ptr now stores the address of num
```

- **Dereferencing:** Dereferencing a pointer means accessing the value stored at the memory address the pointer is pointing to, using the asterisk (*).

```cpp
cout << *ptr; // Prints the value at the address stored in ptr,
```

- **Pointer Arithmetic:** You can perform arithmetic operations on pointers, such as incrementing or decrementing to move to the next or previous memory location.

```cpp
ptr++; // Moves the pointer to the next memory address (if it's an array)
```

- **Null Pointer:** A null pointer does not point to any valid memory location. It is often initialized as nullptr in modern C++.

```cpp
int* ptr = nullptr; // A pointer that doesn't point to any valid memory
```

Why Use Pointers?

- **Dynamic Memory Allocation:** Using new and delete to allocate and free memory during runtime.
- **Efficient Data Handling:** Passing large data structures (like arrays, objects) to functions without copying the entire structure.
- **Low-Level Memory Management:** Allows direct manipulation of memory locations.

¦LAB - 【Arrays】¦

DESCRIPTION :

*Array **->** Same data type **->** Contiguous memory.*

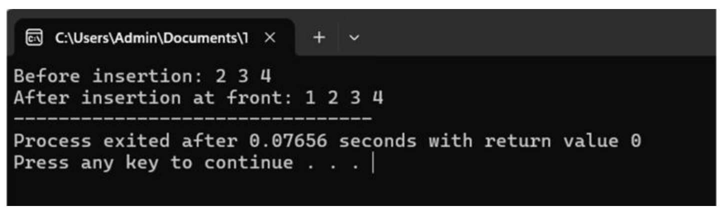
1. Insertion (Front, Mid, End)
2. Deletion (Front, Mid, End)
3. Traversing
4. Searching

INSERTION :

Statement: Write a C++ program to insert an element at the front of an array

```
#include <iostream>

using namespace std;


void insertAtFront(int arr[], int& size, int element) {

   for (int i = size; i > 0; i--) {

      arr[i] = arr[i - 1];

   }

   arr[0] = element;

   size++;

}

int main() {

   int arr[10] = {2, 3, 4};

   int size = 3;

   insertAtFront(arr, size, 1);

   for (int i = 0; i < size; i++) cout << arr[i] << " ";

   return 0;

}
```

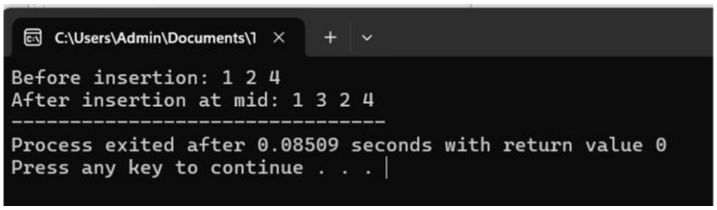
OUTPUT



Statement: Write a C++ program to insert an element in the middle of an array:

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

void insertAtMid(int arr[], int& size, int element) {
    int mid = size / 2;
    for (int i = size; i > mid; i--) {
        arr[i] = arr[i - 1];
    }
    arr[mid] = element;
    size++;
}

int main() {
    int arr[10] = {1, 2, 4};
    int size = 3;

    cout << "Before insertion: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    cout << endl;

    insertAtMid(arr, size, 3);

    cout << "After insertion at mid: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    return 0;
}
```

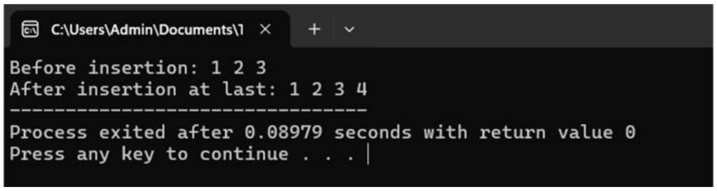
OUTPUT



Statement: Write a C++ program to insert an element at the end of an array

```
#include <iostream>

using namespace std;


void insertAtLast(int arr[], int& size, int element) {

    arr[size] = element;

    size++;

}


int main() {

    int arr[10] = {1, 2, 3};

    int size = 3;


    cout << "Before insertion: ";

    for (int i = 0; i < size; i++) cout << arr[i] << " ";

    cout << endl;


    insertAtLast(arr, size, 4);


    cout << "After insertion at last: ";

    for (int i = 0; i < size; i++) cout << arr[i] << " ";

    return 0;

}
```

OUTPUT



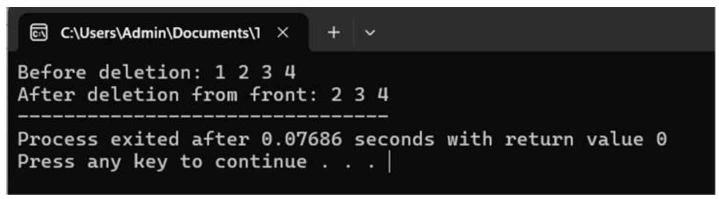
DELETION :

Statement: Write a C++ program to delete an element from the front of an array

```
#include <iostream>
```

```
using namespace std;

void deleteFromFront(int arr[], int& size) {
    for (int i = 0; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}
int main() {
    int arr[10] = {1, 2, 3, 4};
    int size = 4;

    cout << "Before deletion: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    cout << endl;

    deleteFromFront(arr, size);

    cout << "After deletion from front: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    return 0;
}
```

OUTPUT

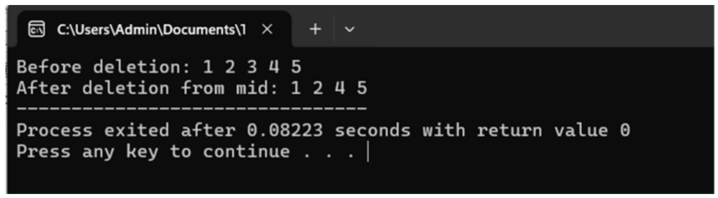


Statement: Write a C++ program to delete an element from the mid of an array

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

```
void deleteFromMid(int arr[], int& size) {
    int mid = size / 2;
    for (int i = mid; i < size - 1; i++) {
        arr[i] = arr[i + 1];
    }
    size--;
}

int main() {
    int arr[10] = {1, 2, 3, 4, 5};
    int size = 5;

    cout << "Before deletion: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    cout << endl;

    deleteFromMid(arr, size);

    cout << "After deletion from mid: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    return 0;
}
```

OUTPUT

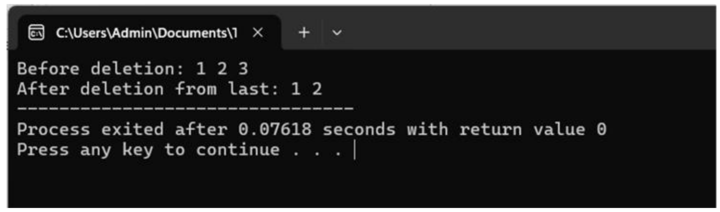


Statement: Write a C++ program to delete an element from the end of an array

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

void deleteFromLast(int& size) {
```

```
    size--;
}

int main() {
    int arr[10] = {1, 2, 3};
    int size = 3;

    cout << "Before deletion: ";
    for (int i = 0; i < size; i++) cout << arr[i]
<< " ";
    cout << endl;

    deleteFromLast(size);
    cout << "After deletion from last: ";
    for (int i = 0; i < size; i++) cout << arr[i] << " ";
    return 0;
}
```

OUTPUT

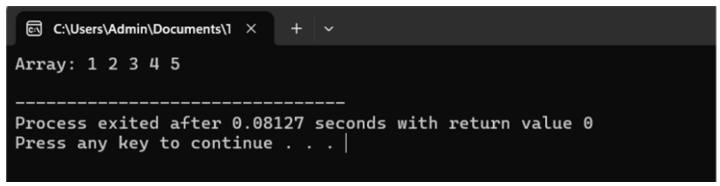


TRAVERSING :

Statement: Write a C++ program to traverse an array and display all its elements sequentially.

```
#include <iostream>
using namespace std;
void traverse(int arr[], int size) {
    cout << "Array: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
```

OUTPUT



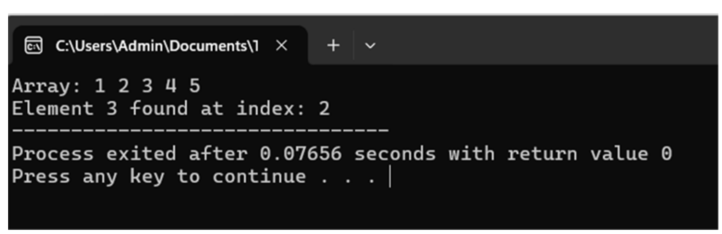
```
    cout << endl;
}

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int size = 5;
    traverse(arr, size);
}
```

SEARCHING:

Statement: Write a C++ program to search for an element in an array

```
#include <iostream>
using namespace std;
int search(int arr[], int size, int element) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == element) return i; // return index if found
    }
    return -1; // not found
}
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int size = 5;
    int element = 3;

    cout << "Array: ";
    for (int i = 0; i < size; i++) cout << arr[i] <<
" ";
    cout << endl;
```

OUTPUT



```
    int result = search(arr, size, element);
    if (result != -1) cout << "Element " << element << " found at index: " << result;
    else cout << "Element " << element << " not found";
    return 0;
}
```

✄---

¦LAB -【Stack】¦

DESCRIPTION :

*Stack **->** Linear Data Structure (DS) **->** LIFO (Last In, First Out)*

INFIX TO POSTFIX :

Statement: Convert an infix expression to postfix using a stack in C++.

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

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

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

// Function to convert infix to postfix
string infixToPostfix(string infix) {
    stack<char> s;
```

```
string postfix = "";

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

    // If the character is an operand, add it to the result
    if (isalnum(c)) {
        postfix += c;
    }
    // If the character is '(', push it to the stack
    else if (c == '(') {
        s.push(c);
    }
    // If the character is ')', pop from the stack until '(' is found
    else if (c == ')') {
        while (!s.empty() && s.top() != '(') {
            postfix += s.top();
            s.pop();
        }
        s.pop(); // Pop '(' from the stack
    }
    // If the character is an operator
    else if (isOperator(c)) {
        while (!s.empty() && precedence(s.top()) >= precedence(c)) {
            postfix += s.top();
            s.pop();
        }
        s.push(c);
    }
```

```
    }

    // Pop all the remaining operators from the stack
    while (!s.empty()) {
        postfix += s.top();
        s.pop();
    }

    return postfix;
}

int main() {
    string infix = "A*(B+C)-D/E";
    cout << "Infix Expression: " << infix << endl;

    string postfix = infixToPostfix(infix);
    cout << "Postfix Expression: " << postfix << endl;

    return 0;
}
```

OUTPUT

✂---

¦LAB - 【Queue】¦

DESCRIPTION :

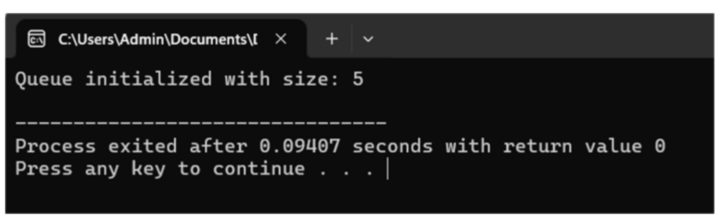
*Queue **->** Linear Data Structure (DS) **->** FIFO (First In, First Out)*

INITIALIZATION :

Statement: Write a C++ program to initialize a queue using a class. Define a constructor that accepts the size of the queue..

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

class Queue {
private:
    int front, rear, size;
    int* queue;

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

int main() {
    Queue q(5); // Create a queue of size 5
```

OUTPUT



```
    return 0;
}
```

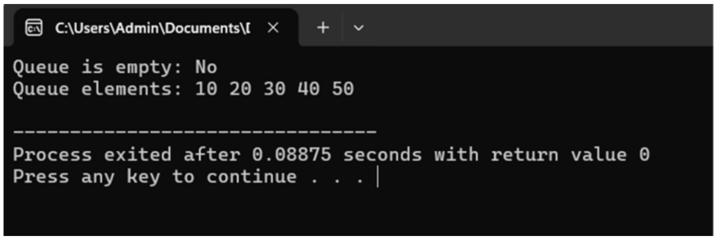
TO CHECK :

Statement: Write a C++ program to initialize a queue using a class. Define a constructor that accepts the size of the queue..

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

class Queue {
private:
    int front, rear, size; // Front and rear pointers and queue size
    int* queue;           // Dynamic array to hold queue elements

public:
    // Constructor to initialize the queue and populate it with elements
    Queue(int n) {
        size = n;
        queue = new int[size];
        front = 0;      // Initialize front to 0
        rear = size - 1; // Initialize rear to the last index
        for (int i = 0; i < size; i++) {
            queue[i] = (i + 1) * 10; // Populate the queue with values (10, 20, 30, ...)
        }
    }

    // Function to check if the queue is full
    bool isFull() {
```

```
        return rear == size - 1;
    }

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

    // Function to display the elements of the queue
    void display() {
        if (isEmpty()) {
            cout << "Queue is empty!" << endl;
        } else {
            cout << "Queue elements: ";
            for (int i = front; i <= rear; i++) {
                cout << queue[i] << " ";
            }
            cout << endl;
        }
    }
};
int main() {
    Queue q(5); // Create a queue of size 5 with default values
    // Check if the queue is empty
    cout << "Queue is empty: " << (q.isEmpty() ? "Yes" : "No") << endl;
    // Display the queue before checking its state
    q.display();
    return 0;
}
```

OUTPUT

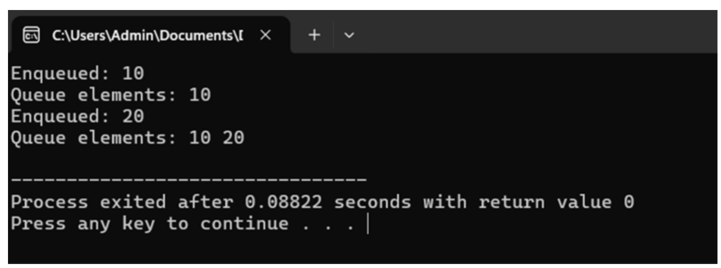


ENQUEUE :

Statement: Write a program to add an element to a queue using an enqueue function. Ensure that you check if the queue is full before adding an element.

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

class Queue {
private:
    int front, rear, size; // Front and rear pointers and queue size
    int* queue;           // Dynamic array to hold queue elements

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

    // Function to check if the queue is full
    bool isFull() {
        return rear == size - 1;
    }

    // Function to add an element to the queue
    void enqueue(int value) {
        if (isFull()) {
```

OUTPUT



```
            cout << "Queue is full, cannot enqueue!" << endl;
        } else {
            if (front == -1) front = 0; // Set front to 0 if it's the first element
            rear++;
            queue[rear] = value;
            cout << "Enqueued: " << value << endl;
            display(); // Show the list after enqueueing
        }
    }

    // Function to display the elements of the queue
    void display() {
        if (front == -1 || front > rear) {
            cout << "Queue is empty!" << endl;
        } else {
            cout << "Queue elements: ";
            for (int i = front; i <= rear; i++) {
                cout << queue[i] << " ";
            }
            cout << endl;
        }
    }
};

int main() {
    Queue q(5); // Create a queue of size 5

    q.enqueue(10); // Add element 10
    q.enqueue(20); // Add element 20
```

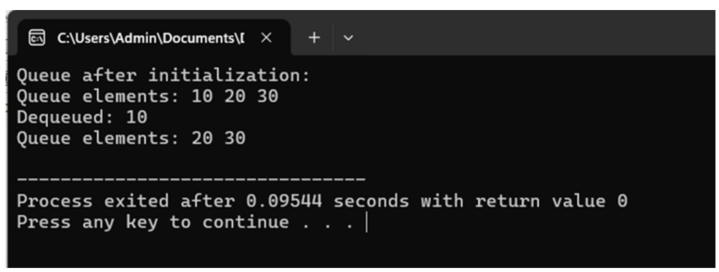
```
    return 0;
}
```

DEQUEUE :

Statement: Write a program to add an element to a queue using an enqueue function. Ensure that you check if the queue is full before adding an element.

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

class Queue {
private:
    int front, rear, size;
    int* queue;

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

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == -1 || front > rear;
    }
```

OUTPUT



```
// Function to remove an element from the queue
void dequeue() {
    if (isEmpty()) {
        cout << "Queue is empty, cannot dequeue!" << endl;
    } else {
        cout << "Dequeued: " << queue[front] << endl;
        front++;
    }
    display(); // Show the list after dequeuing
}

// Function to add an element to the queue
void enqueue(int value) {
    if (rear == size - 1) {
        cout << "Queue is full, cannot enqueue!" << endl;
    } else {
        if (front == -1) front = 0;
        rear++;
        queue[rear] = value;
    }
}

// Function to display the elements of the queue
void display() {
    if (isEmpty()) {
        cout << "Queue is empty!" << endl;
    } else {
        cout << "Queue elements: ";
        for (int i = front; i <= rear; i++) {
```

```
            cout << queue[i] << " ";
          }
          cout << endl;
        }
    }
};

int main() {
    Queue q(5); // Create a queue of size 5
    // Enqueue some elements
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    // Display the queue after initialization
    cout << "Queue after initialization:" << endl;
    q.display();
    // Perform dequeue and show the list after it
    q.dequeue(); // Attempt to dequeue from the queue
    return 0;
}
```

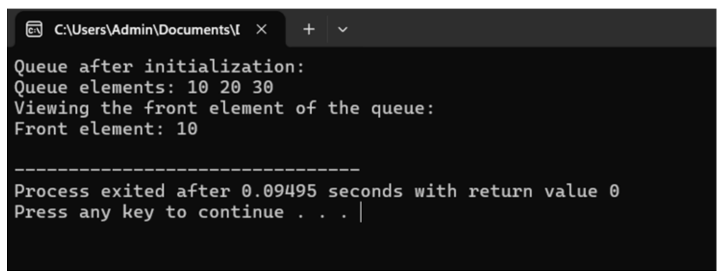
PEEK :

Statement: Write a program to add an element to a queue using an enqueue function. Ensure that you check if the queue is full before adding an element.

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

class Queue {
```

```
private:
    int front, rear, size;
    int* queue;

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

    // Function to check if the queue is empty
    bool isEmpty() {
        return front == -1 || front > rear;
    }

    // Function to add an element to the queue
    void enqueue(int value) {
        if (rear == size - 1) {
            cout << "Queue is full, cannot enqueue!" << endl;
        } else {
            if (front == -1) front = 0;
            rear++;
            queue[rear] = value;
        }
    }
```

OUTPUT



```
// Function to view the front element of the queue
void peek() {
   if (isEmpty()) {
      cout << "Queue is empty, no front element!" << endl;
   } else {
      cout << "Front element: " << queue[front] << endl;
   }
}

// Function to remove an element from the queue
void dequeue() {
   if (isEmpty()) {
      cout << "Queue is empty, cannot dequeue!" << endl;
   } else {
      cout << "Dequeued: " << queue[front] << endl;
      front++;
   }
}

// Function to display the elements of the queue
void display() {
   if (isEmpty()) {
      cout << "Queue is empty!" << endl;
   } else {
      cout << "Queue elements: ";
      for (int i = front; i <= rear; i++) {
         cout << queue[i] << " ";
      }
      cout << endl;
```

```
        }
    }
};

int main() {
    Queue q(5); // Create a queue of size 5

    // Enqueue some elements
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    // Display the queue after initialization
    cout << "Queue after initialization:" << endl;
    q.display();

    // View the front element of the queue
    cout << "Viewing the front element of the queue:" << endl;
    q.peek();

    return 0;
}
```

✂---

¦LAB -【Linked List】¦

DESCRIPTION :

Linked List **->** *Linear Data Structure (DS)* **->** *Non-contiguous memory with nodes linked by references*

NODE STRUCTURE AND LINKEDLIST CLASS :

This will be the base class and node structure for the linked list

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

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to display the list (traverse)
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

❝ ꒰ SINGLE ꒱ ❞

1. ***Insertion (Front, Mid, End)***
2. ***Deletion (Front, Mid, End)***

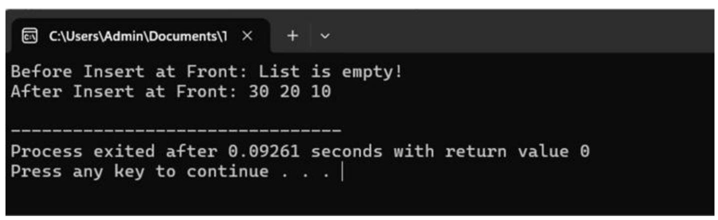
INSERTION :

Statement: Write a C++ program to insert an element at the front using single linked list

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

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }
```

```
    // Function to insert a new node at the front
    void insertAtFront(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Function to display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;

    // Before insertion
    cout << "Before Insert at Front: ";
    list.traverse();  // Output: List is empty!
```

OUTPUT

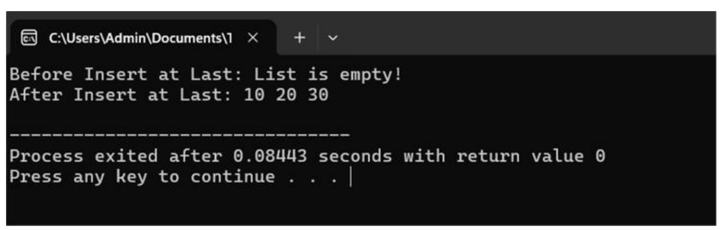


```
    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);

    // After insertion at front
    cout << "After Insert at Front: ";
    list.traverse();  // Output: 30 20 10

    return 0;
}
```

Statement: Write a C++ program to insert an element at the end using single linked list

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

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;
```

OUTPUT



```
LinkedList() {
    head = nullptr;
}

// Function to insert a new node at the last
void insertAtLast(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
```

```
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;

    // Before insertion
    cout << "Before Insert at Last: ";
    list.traverse();  // Output: List is empty!

    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);

    // After insertion at last
    cout << "After Insert at Last: ";
    list.traverse();  // Output: 10 20 30

    return 0;
}
```

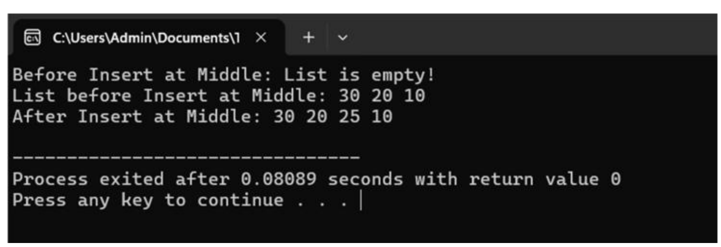
Statement: Write a C++ program to insert an element at the mid using single linked list

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

struct Node {
    int data;
```

```
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to insert a new node at the middle
    void insertAtMid(int value) {
        if (head == nullptr) {
            insertAtFront(value); // If list is empty, insert at front
            return;
        }

        Node* slow = head;
        Node* fast = head;

        // Move slow pointer to the middle and fast pointer to the end
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
```

```
        fast = fast->next->next;
    }

    // Insert new node after the slow pointer
    Node* newNode = new Node(value);
    newNode->next = slow->next;
    slow->next = newNode;
}

// Function to insert a node at the front (used when the list is empty)
void insertAtFront(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

OUTPUT



```
};

int main() {
    LinkedList list;

    // Before insertion at middle
    cout << "Before Insert at Middle: ";
    list.traverse();  // Output: List is empty!

    // Inserting nodes
    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);
    cout << "List before Insert at Middle: ";
    list.traverse();  // Output: 30 20 10
    // Insert at middle
    list.insertAtMid(25);
    // After insertion at middle
    cout << "After Insert at Middle: ";
    list.traverse();  // Output: 30 20 25 10
    return 0;
}
```

DELETION :

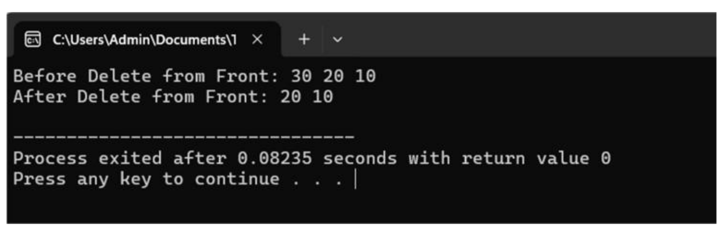
Statement: Write a C++ program to del an element from front using single linked list

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

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

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to insert a new node at the front
    void insertAtFront(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Function to delete a node from the front
    void deleteFromFront() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
```

```
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
    }

    // Function to display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;
    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);

    // Before deletion from front
```

OUTPUT



```
    cout << "Before Delete from Front: ";
    list.traverse();  // Output: 30 20 10

    list.deleteFromFront();

    // After deletion from front
    cout << "After Delete from Front: ";
    list.traverse();  // Output: 20 10

    return 0;
}
```

Statement: Write a C++ program to del an element from mid using single linked list

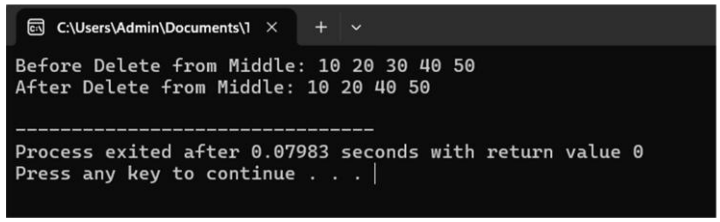
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;
```

```
LinkedList() {
    head = nullptr;
}

// Function to insert a new node at the end
void insertAtLast(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to delete a node from the middle
void deleteFromMid() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
```

```
    }

    Node* slow = head;
    Node* fast = head;
    Node* prev = nullptr;

    // Move slow pointer to middle and fast pointer to the end
    while (fast != nullptr && fast->next != nullptr) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // Delete the middle node
    prev->next = slow->next;
    delete slow;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
```

```
        cout << endl;
    }
};

int main() {
    LinkedList list;
    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);
    list.insertAtLast(40);
    list.insertAtLast(50);

    // Before deletion from middle
    cout << "Before Delete from Middle: ";
    list.traverse();  // Output: 10 20 30 40 50

    list.deleteFromMid();

    // After deletion from middle
    cout << "After Delete from Middle: ";
    list.traverse();  // Output: 10 20 40 50

    return 0;
}
```

OUTPUT



Statement: Write a C++ program to del an element from end using single linked list

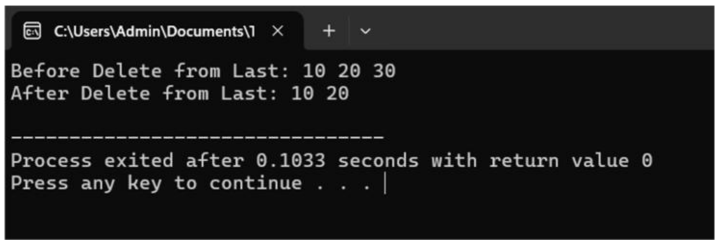
```
#include <iostream>
using namespace std;
```

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

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to insert a new node at the end
    void insertAtLast(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
```

```
    temp->next = newNode;
}

// Function to delete a node from the last
void deleteFromLast() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr && temp->next->next != nullptr) {
        temp = temp->next;
    }
    delete temp->next;
    temp->next = nullptr;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
```

```
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;
    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);

    // Before deletion from last
    cout << "Before Delete from Last: ";
    list.traverse();  // Output: 10 20 30

    list.deleteFromLast();

    // After deletion from last
    cout << "After Delete from Last: ";
    list.traverse();  // Output: 10 20

    return 0;
}
```

OUTPUT



“꒰DOUBLY꒱”

1. ***Insertion (Front, Mid, End)***
2. ***Deletion (Front, Mid, End)***

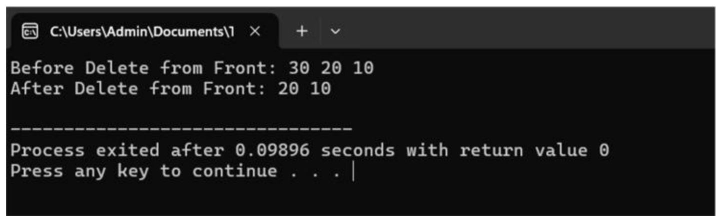
DELETION :

Statement: Write a C++ program to del an element at from front using double linked list

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

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }
```

```
// Function to insert a new node at the front
void insertAtFront(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;
    head = newNode;
}

// Function to delete a node from the front
void deleteFromFront() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
```

OUTPUT



```
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;
    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);

    // Before deletion from front
    cout << "Before Delete from Front: ";
    list.traverse();  // Output: 30 20 10

    list.deleteFromFront();

    // After deletion from front
    cout << "After Delete from Front: ";
    list.traverse();  // Output: 20 10

    return 0;
}
```

Statement: Write a C++ program to del an element from mid using double linked list

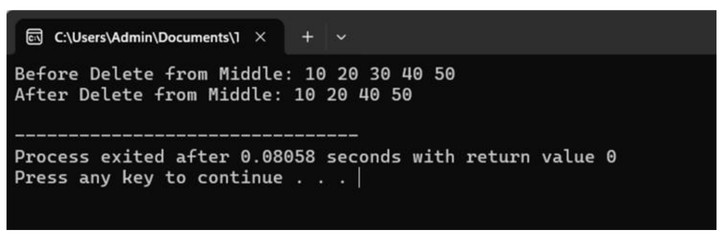
```
#include <iostream>
using namespace std;

struct Node {
```

```
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }

    // Function to insert a new node at the end
    void insertAtLast(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
```

```
}

// Function to delete a node from the middle
void deleteFromMid() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }

    Node* slow = head;
    Node* fast = head;
    Node* prev = nullptr;

    // Move slow pointer to middle and fast pointer to the end
    while (fast != nullptr && fast->next != nullptr) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // Delete the middle node
    prev->next = slow->next;
    delete slow;
}
```

```
    // Function to display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

OUTPUT




```
int main() {
    LinkedList list;
    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);
    list.insertAtLast(40);
    list.insertAtLast(50);

    // Before deletion from middle
    cout << "Before Delete from Middle: ";
    list.traverse();  // Output: 10 20 30 40 50

    list.deleteFromMid();
```

```
    // After deletion from middle
    cout << "After Delete from Middle: ";
    list.traverse();  // Output: 10 20 40 50

    return 0;
}
```

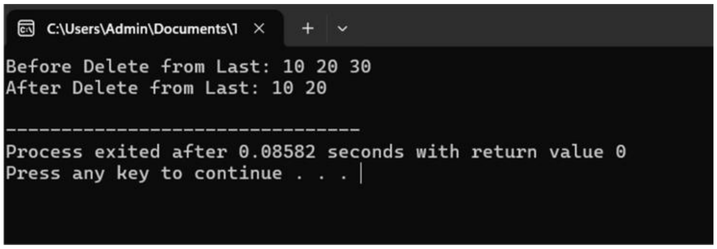
Statement: Write a C++ program to del an element from end using double linked list

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

struct Node {
    int data;
    Node* next;

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

class LinkedList {
public:
    Node* head;

    LinkedList() {
        head = nullptr;
    }
```

```
// Function to insert a new node at the end
void insertAtLast(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to delete a node from the last
void deleteFromLast() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr && temp->next->next != nullptr) {
        temp = temp->next;
```

```
        }
        delete temp->next;
        temp->next = nullptr;
    }

    // Function to display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList list;
    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);

    // Before deletion from last
    cout << "Before Delete from Last: ";
    list.traverse();  // Output: 10 20 30
```

OUTPUT



```
    list.deleteFromLast();

    // After deletion from last
    cout << "After Delete from Last: ";
    list.traverse();  // Output: 10 20

    return 0;
}
```

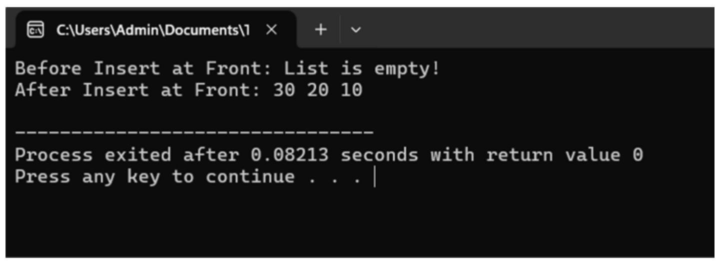
INSERTION :

Statement: Write a C++ program to insert an element at the front using double linked list

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

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

    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class DoublyLinkedList {
```

```
public:
    Node* head;

    DoublyLinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the front
    void insertAtFront(int value) {
        Node* newNode = new Node(value);
        if (head != nullptr) {
            head->prev = newNode;
            newNode->next = head;
        }
        head = newNode;
    }

    // Function to display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
```

OUTPUT



```
    }
};

int main() {
    DoublyLinkedList list;

    // Before insertion at front
    cout << "Before Insert at Front: ";
    list.traverse();  // Output: List is empty!

    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);

    // After insertion at front
    cout << "After Insert at Front: ";
    list.traverse();  // Output: 30 20 10

    return 0;
}
```

Statement: Write a C++ program to insert an element at the mid using double linked list

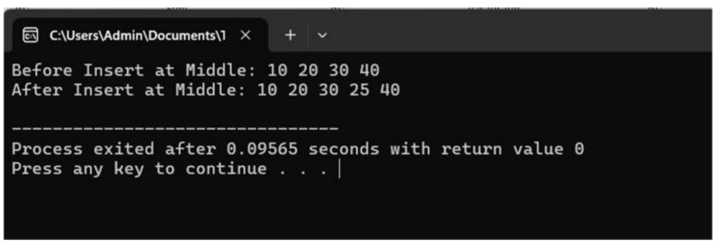
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* prev;
    Node* next;
```

```
    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class DoublyLinkedList {
public:
    Node* head;

    DoublyLinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the end
    void insertAtLast(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
```

```
}

// Function to insert a node at the middle
void insertAtMid(int value) {
    if (head == nullptr || head->next == nullptr) { // Insert as the first or second element
        insertAtLast(value);
        return;
    }

    Node* slow = head;
    Node* fast = head;

    // Find the middle using two pointers
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Insert new node after 'slow'
    Node* newNode = new Node(value);
    newNode->next = slow->next;
    if (slow->next != nullptr) {
        slow->next->prev = newNode;
    }
    slow->next = newNode;
    newNode->prev = slow;
}

// Function to display the list
```

```
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};

int main() {
    DoublyLinkedList list;

    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);
    list.insertAtLast(40);

    // Before insertion at middle
    cout << "Before Insert at Middle: ";
    list.traverse();  // Output: 10 20 30 40

    list.insertAtMid(25);

    // After insertion at middle
```

OUTPUT



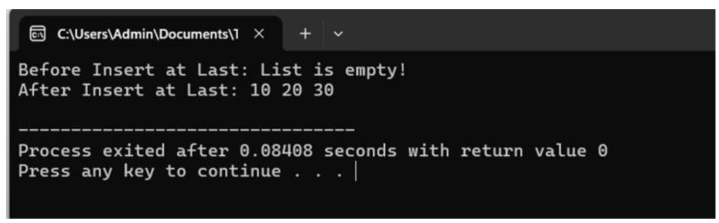
```
    cout << "After Insert at Middle: ";
    list.traverse();  // Output: 10 20 25 30 40

    return 0;
}
```

Statement: Write a C++ program to insert an element at the end using double linked list

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

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

    Node(int value) {
        data = value;
        prev = nullptr;
        next = nullptr;
    }
};

class DoublyLinkedList {
public:
    Node* head;

    DoublyLinkedList() {
        head = nullptr;
    }
```

```
// Function to insert a node at the end
void insertAtLast(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

// Function to display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
```

OUTPUT



```
};

int main() {
    DoublyLinkedList list;

    // Before insertion at last
    cout << "Before Insert at Last: ";
    list.traverse();  // Output: List is empty!

    list.insertAtLast(10);
    list.insertAtLast(20);
    list.insertAtLast(30);

    // After insertion at last
    cout << "After Insert at Last: ";
    list.traverse();  // Output: 10 20 30

    return 0;
}
```

“꒰CIRCULAR꒱”

1. Insertion (Front, Mid, End)

*2. **Deletion (Front, Mid, End)***

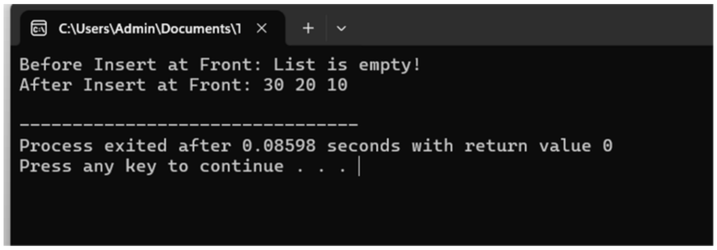
INSERTION :

Statement: Write a C++ program to insert an element at the front using circular linked list

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

struct Node {
    int data;
    Node* next;

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }
```

```
// Function to insert a node at the front
void insertAtFront(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
        head = newNode;
    }
}

// Function to traverse and display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
```

OUTPUT



```
        cout << endl;
    }
};

int main() {
    CircularLinkedList list;

    // Before insertion at front
    cout << "Before Insert at Front: ";
    list.traverse();  // Output: List is empty!

    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);

    // After insertion at front
    cout << "After Insert at Front: ";
    list.traverse();  // Output: 30 20 10

    return 0;
}
```

Statement: Write a C++ program to insert an element at the mid using circular linked list

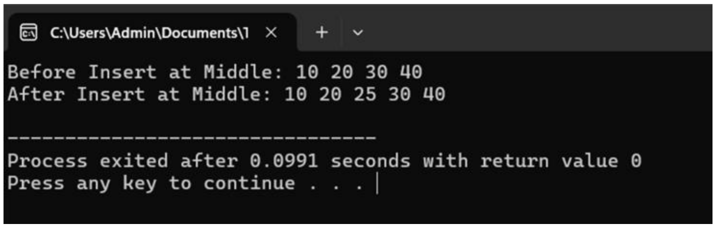
```
#include <iostream>
using namespace std;

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

```
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
```

```
}

// Function to insert a node at the middle
void insertAtMid(int value) {
    if (head == nullptr || head->next == head) { // If list has 0 or 1 nodes
        insertAtEnd(value);
        return;
    }

    Node* slow = head;
    Node* fast = head;

    // Use two pointers to find the middle
    while (fast->next != head && fast->next->next != head) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Insert new node after slow
    Node* newNode = new Node(value);
    newNode->next = slow->next;
    slow->next = newNode;
}

// Function to traverse and display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
```

```
        }
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};

int main() {
    CircularLinkedList list;

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.insertAtEnd(40);

    // Before insertion at middle
    cout << "Before Insert at Middle: ";
    list.traverse();  // Output: 10 20 30 40

    list.insertAtMid(25);

    // After insertion at middle
    cout << "After Insert at Middle: ";
    list.traverse();  // Output: 10 20 25 30 40

    return 0;
```

OUTPUT



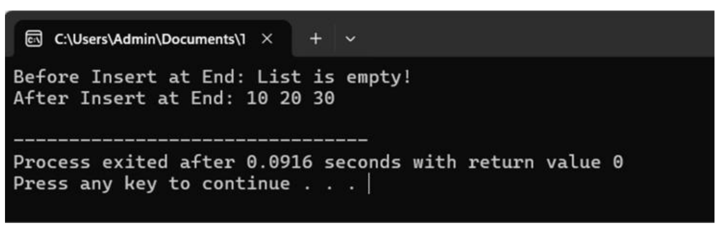
```
}
```

Statement: Write a C++ program to insert an element at the end using circular linked list

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

struct Node {
    int data;
    Node* next;

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }

    // Function to insert a node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
```

```
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
    }

    // Function to traverse and display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};

int main() {
    CircularLinkedList list;
```

OUTPUT



```
    // Before insertion at end
    cout << "Before Insert at End: ";
    list.traverse();  // Output: List is empty!

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);

    // After insertion at end
    cout << "After Insert at End: ";
    list.traverse();  // Output: 10 20 30

    return 0;
}
```

DELETION :

Statement: Write a C++ program to del an element from front using circular linked list

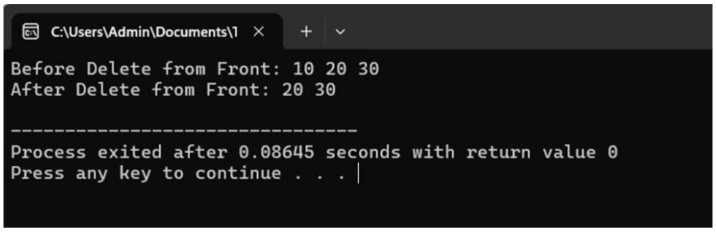
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

    Node(int value) {
        data = value;
        next = nullptr;
    }
```

```
};

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }

    // Insert a node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
            newNode->next = head;
        }
    }

    // Delete a node from the front
    void deleteFromFront() {
        if (head == nullptr) {
```

```
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == head) { // Only one node
        delete head;
        head = nullptr;
        return;
    }
    Node* temp = head;
    Node* last = head;
    while (last->next != head) { // Find the last node
        last = last->next;
    }
    head = head->next;
    last->next = head;
    delete temp;
}

// Display the list
void traverse() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
```

```
        cout << endl;
    }
};

int main() {
    CircularLinkedList list;

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);

    // Before deletion from front
    cout << "Before Delete from Front: ";
    list.traverse(); // Output: 10 20 30

    list.deleteFromFront();

    // After deletion from front
    cout << "After Delete from Front: ";
    list.traverse(); // Output: 20 30

    return 0;
}
```

OUTPUT



Statement: Write a C++ program to del an element from mid using circular linked list

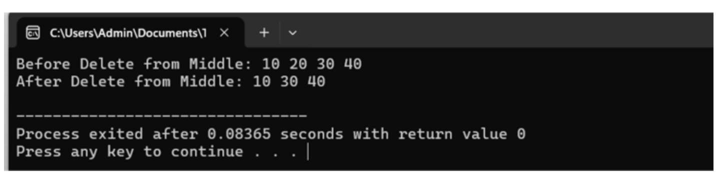
```
#include <iostream>
using namespace std;

struct Node {
```

```
    int data;
    Node* next;

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
        head = nullptr;
    }

    // Insert a node at the end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            newNode->next = head;
        } else {
            Node* temp = head;
            while (temp->next != head) {
                temp = temp->next;
            }
            temp->next = newNode;
```

```
        newNode->next = head;
    }
}

// Delete a node from the middle
void deleteFromMid() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == head) { // Only one node
        delete head;
        head = nullptr;
        return;
    }

    Node* slow = head;
    Node* fast = head;
    Node* prev = nullptr;

    // Use two pointers to find the middle
    while (fast->next != head && fast->next->next != head) {
        prev = slow;
        slow = slow->next;
        fast = fast->next->next;
    }

    // Remove the middle node
    prev->next = slow->next;
```

```
            delete slow;
        }

        // Display the list
        void traverse() {
            if (head == nullptr) {
                cout << "List is empty!" << endl;
                return;
            }
            Node* temp = head;
            do {
                cout << temp->data << " ";
                temp = temp->next;
            } while (temp != head);
            cout << endl;
        }
};

int main() {
    CircularLinkedList list;

    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.insertAtEnd(40);

    // Before deletion from middle
    cout << "Before Delete from Middle: ";
    list.traverse(); // Output: 10 20 30 40
```

OUTPUT



```
    list.deleteFromMid();

    // After deletion from middle
    cout << "After Delete from Middle: ";
    list.traverse(); // Output: 10 20 40

    return 0;
}
```

Statement: Write a C++ program to del an element from end using circular linked list

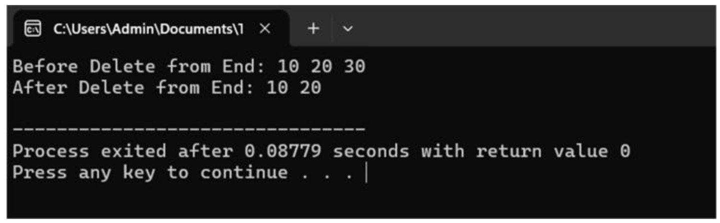
```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;

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

class CircularLinkedList {
public:
    Node* head;

    CircularLinkedList() {
```

```
    head = nullptr;
}

// Insert a node at the end
void insertAtEnd(int value) {
    Node* newNode = new Node(value);
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = head;
    }
}

// Delete a node from the end
void deleteFromEnd() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }
    if (head->next == head) { // Only one node
        delete head;
        head = nullptr;
        return;
```

```
        }
        Node* temp = head;
        Node* prev = nullptr;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head;
        delete temp;
    }

    // Display the list
    void traverse() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        do {
            cout << temp->data << " ";
            temp = temp->next;
        } while (temp != head);
        cout << endl;
    }
};

int main() {
    CircularLinkedList list;
```

```
    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);

    // Before deletion from end
    cout << "Before Delete from End: ";
    list.traverse(); // Output: 10 20 30

    list.deleteFromEnd();

    // After deletion from end
    cout << "After Delete from End: ";
    list.traverse(); // Output: 10 20

    return 0;
}
```

OUTPUT



¦LAB -【BST】¦

DESCRIPTION :

Binary Search Tree (BST) -> Hierarchical Data Structure (DS) -> Left < Root < Right

1. **Insertion**
2. **Deletion**
3. **Searching**
4. **Traversal (In-order, Pre-order, Post-order)**

TRAVERSING:

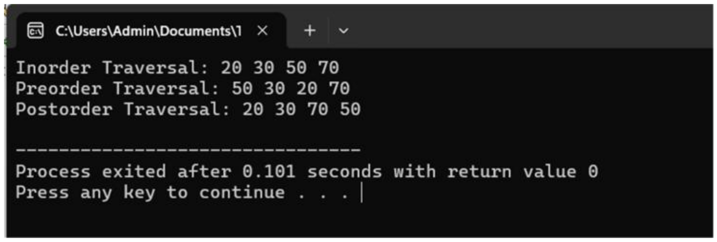
Statement: Implement a Binary Search Tree (BST) in C++ with insertion and tree traversals (in-order, pre-order, post-order).

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

// Define the structure of a tree node
struct Node {
    int data; // Value of the node
    Node* left; // Pointer to the left child
    Node* right; // Pointer to the right child

    // Constructor to initialize a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Inorder Traversal: Left -> Root -> Right
void inorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    inorder(root->left);        // Recur on the left subtree
```

```
    cout << root->data << " ";   // Print the root's data
    inorder(root->right);       // Recur on the right subtree
}

// Preorder Traversal: Root -> Left -> Right
void preorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    cout << root->data << " ";   // Print the root's data
    preorder(root->left);       // Recur on the left subtree
    preorder(root->right);      // Recur on the right subtree
}

// Postorder Traversal: Left -> Right -> Root
void postorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    postorder(root->left);      // Recur on the left subtree
    postorder(root->right);     // Recur on the right subtree
    cout << root->data << " ";   // Print the root's data
}

// Insert a new node into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) return new Node(value); // Create a new node if root is null

    // Recur down the tree to find the correct position
    if (value < root->data)
        root->left = insert(root->left, value); // Insert in the left subtree
    else
        root->right = insert(root->right, value); // Insert in the right subtree

    return root; // Return the unchanged root
}

int main() {
    Node* root = nullptr; // Initialize the root of the BST
```

```
    // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);

    // Perform different traversals
    cout << "Inorder Traversal: ";
    inorder(root); // Call inorder traversal
    cout << endl;

    cout << "Preorder Traversal: ";
    preorder(root); // Call preorder traversal
    cout << endl;

    cout << "Postorder Traversal: ";
    postorder(root); // Call postorder traversal
    cout << endl;

    return 0;
}
```

OUTPUT



SEARCHING:

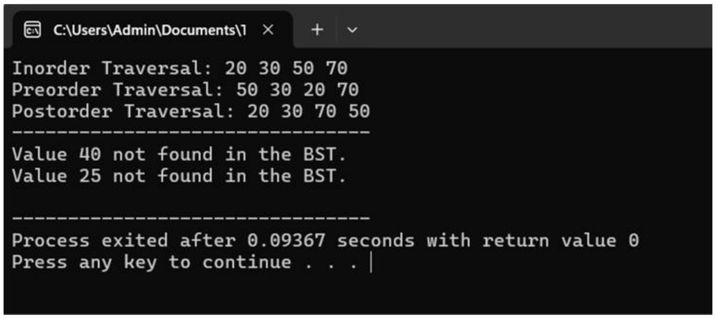
Statement: Implement a Binary Search Tree (BST) in C++ with insertion, tree traversal (in-order, pre-order, post-order), and search functionality.

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

// Define the structure of a tree node
struct Node {
    int data; // Value of the node
    Node* left; // Pointer to the left child
```

```
    Node* right; // Pointer to the right child

    // Constructor to initialize a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Inorder Traversal: Left -> Root -> Right
void inorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    inorder(root->left);        // Recur on the left subtree
    cout << root->data << " ";   // Print the root's data
    inorder(root->right);       // Recur on the right subtree
}

// Preorder Traversal: Root -> Left -> Right
void preorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    cout << root->data << " ";   // Print the root's data
    preorder(root->left);       // Recur on the left subtree
    preorder(root->right);      // Recur on the right subtree
}

// Postorder Traversal: Left -> Right -> Root
void postorder(Node* root) {
    if (root == nullptr) return; // Base case: If node is null, stop
    postorder(root->left);      // Recur on the left subtree
    postorder(root->right);     // Recur on the right subtree
    cout << root->data << " ";   // Print the root's data
}

// Insert a new node into the BST
Node* insert(Node* root, int value) {
```

```
    if (root == nullptr) return new Node(value); // Create a new node if root is null

    // Recur down the tree to find the correct position
    if (value < root->data)
        root->left = insert(root->left, value); // Insert in the left subtree
    else
        root->right = insert(root->right, value); // Insert in the right subtree

    return root; // Return the unchanged root
}

// Search for a value in the BST using recursion (short version)
bool search(Node* root, int target) {
    return root && (root->data == target || search(target < root->data ? root->left : root->right, target));
}

int main() {
    Node* root = nullptr; // Initialize the root of the BST

    // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);

    // Perform different traversals
    cout << "Inorder Traversal: ";
    inorder(root); // Call inorder traversal
    cout << endl;

    cout << "Preorder Traversal: ";
    preorder(root); // Call preorder traversal
    cout << endl;

    cout << "Postorder Traversal: ";
```

OUTPUT



```
    postorder(root); // Call postorder traversal
    cout << endl << "--------------------------------" << endl;

    // Search for values in the BST
    int target1 = 40;
    int target2 = 25;

    // Check if target1 exists in the BST
    if (search(root, target1))
        cout << "Value " << target1 << " found in the BST." << endl;
    else
        cout << "Value " << target1 << " not found in the BST." << endl;

    // Check if target2 exists in the BST
    if (search(root, target2))
        cout << "Value " << target2 << " found in the BST." << endl;
    else
        cout << "Value " << target2 << " not found in the BST." << endl;
    return 0;
}
```

DELETION:

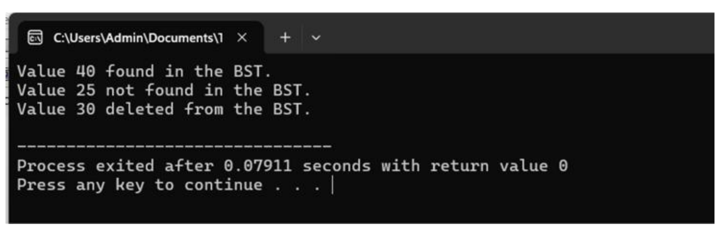
Statement: Implement a C++ program to create a Binary Search Tree (BST) that supports node insertion, searching for a value, and deleting a node using recursion.

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

// Define the structure of a tree node
struct Node {
    int data; // Value of the node
    Node* left; // Pointer to the left child
    Node* right; // Pointer to the right child

    // Constructor to initialize a new node
```

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

// Insert a new node into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) return new Node(value); // Create a new node if root is null

    // Recur down the tree to find the correct position
    if (value < root->data)
        root->left = insert(root->left, value); // Insert in the left subtree
    else
        root->right = insert(root->right, value); // Insert in the right subtree

    return root; // Return the unchanged root
}

// Search for a value in the BST using recursion
bool search(Node* root, int target) {
    // If root is null, value is not found. Otherwise, check current node or recur to left/right subtree
    return root && (root->data == target || search(target < root->data ? root->left : root->right, target));
}

// Delete a node from the BST using recursion
Node* deleteNode(Node* root, int key) {
    if (root == nullptr) return root; // If the tree is empty, return null

    // Recur down the tree to find the node to be deleted
    if (key < root->data)
        root->left = deleteNode(root->left, key); // Search in the left subtree
    else if (key > root->data)
        root->right = deleteNode(root->right, key); // Search in the right subtree
    else {
```

```
        // Node to be deleted is found
        // Case 1: Node has no child or only one child
        if (root->left == nullptr) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (root->right == nullptr) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        // Case 2: Node has two children
        // Find the smallest value in the right subtree (inorder successor)
        Node* temp = root->right;
        while (temp && temp->left != nullptr)
            temp = temp->left;

        // Replace the node's value with the inorder successor's value
        root->data = temp->data;

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

    return root; // Return the updated root
}
int main() {
    Node* root = nullptr; // Initialize the root of the BST

   // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
```

OUTPUT



```
// Search for values in the BST
  int target1 = 40;
  int target2 = 25;

  // Check if target1 exists in the BST
  if (search(root, target1))
     cout << "Value " << target1 << " found in the BST." << endl;
  else
     cout << "Value " << target1 << " not found in the BST." << endl;

  // Check if target2 exists in the BST
  if (search(root, target2))
     cout << "Value " << target2 << " found in the BST." << endl;
  else
     cout << "Value " << target2 << " not found in the BST." << endl;

  // Delete a node from the BST
  int deleteKey = 30;
  root = deleteNode(root, deleteKey);
  cout << "Value " << deleteKey << " deleted from the BST." << endl;
  return 0;
}
```

INSERTION:

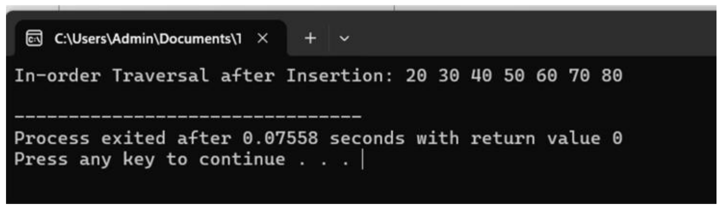
Statement: Implement a C++ program to create a Binary Search Tree (BST) with node insertion and in-order traversal to display the tree's elements.

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

// Define the structure of a tree node
struct Node {
   int data; // Value of the node
```

```
    Node* left; // Pointer to the left child
    Node* right; // Pointer to the right child

    // Constructor to initialize a new node
    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};

// Insert a new node into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) return new Node(value); // Create a new node if root is null

    // Recur down the tree to find the correct position
    if (value < root->data)
        root->left = insert(root->left, value); // Insert in the left subtree
    else
        root->right = insert(root->right, value); // Insert in the right subtree

    return root; // Return the unchanged root
}

// In-order traversal to display the BST
void inOrderTraversal(Node* root) {
    if (root == nullptr) return; // Base case: if tree is empty
    inOrderTraversal(root->left); // Visit left subtree
    cout << root->data << " "; // Visit node
    inOrderTraversal(root->right); // Visit right subtree
```

```
}

int main() {
    Node* root = nullptr; // Initialize the root of the BST

    // Insert nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 70);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 60);
    root = insert(root, 80);

    // Display the BST using in-order traversal
    cout << "In-order Traversal after Insertion: ";
    inOrderTraversal(root);
    cout << endl;

    return 0;
}
```

OUTPUT



✂---