The University of Faisalabad

The image part with relationship ID rId5 was not found in the file.

DATA STRUCTURE 's

LAB MANUAL

Name: AYESHA IMRAN

Student ID: 2023-BSAI-011

Semester : 3RD SEMESTER

Submitted to: MAM IRSHA QURESHI

Section: A SECTION

LAB NO 1

Introduction to c++ and reviewing websites such as w3school also memorizing syntax of basic c++

LAB NO 2

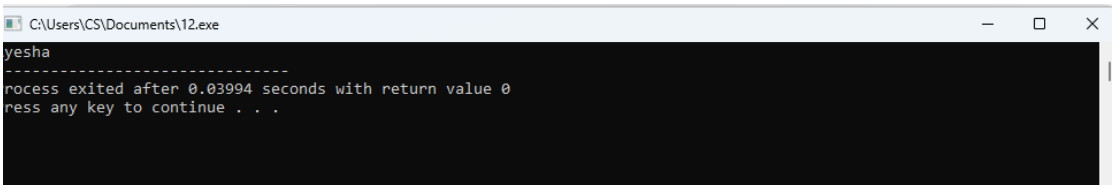
What is Array?

An array is a data structure that stores a fixed-size sequential collection of elements of the same type. In other words, it's a collection of variables (called elements), all of the same type, stored under a single variable name.

PROGRAM NO 1:

```
#include <iostream>
#include <string>
using namespace std;
int main () {
string names [4] = {"Ayesha", "Ali", "Hassan", "Amina"};
cout << names [0];
return 0;
}
```

OUTPUT:



PROGRAM NO 2:

```
#include <iostream>
#include <string>
using namespace std;
int main() {
string names[5] = {"Ayesha", "Ali", "Hassan", "Amina", "Arsh"};
for (int i = 0; i < 5; i++) {
cout << names[i] << "\n";
}
return 0;
}
```

OUTPUT:

```
C:\Users\CS\Documents\12.exe
Ayesha
Ali
Hassan
Amina
Arsh

--------------------------------
Process exited after 0.06048 seconds with return value 0
Press any key to continue . . .
```

PROGRAM NO 3:

```
#include <iostream>
using namespace std;
int main() {
int myNumbers[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
cout << myNumbers[i] << "\n";
}
return 0;
}
```

OUTPUT:

```
C:\Users\CS\Documents\12.exe
10
20
30
40
50

--------------------------------
Process exited after 0.05792 seconds with return value 0
Press any key to continue . . .
```

PROGRAM NO 4:

```
#include <iostream>
using namespace std;
int main() {
int myNumbers[5] = {10, 20, 30, 40, 50};
cout << sizeof(myNumbers);
return 0;
}
```

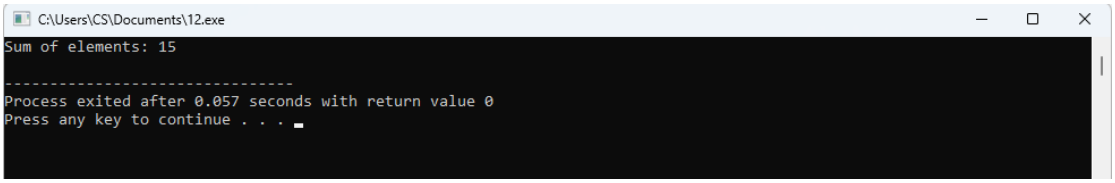
OUTPUT:

```
C:\Users\CS\Documents\12.exe
20
--------------------------------
Process exited after 0.05354 seconds with return value 0
Press any key to continue . . .
```

PROGRAM NO 5:

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < 5; i++) {
    sum += arr[i];
    }
    cout << "Sum of elements: " << sum << endl;
    return 0;
}
```

OUTPUT:



PROGRAM NO 6:

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

int main() {
    int arr[5] = {10, 25, 7, 33, 15};
    int max = arr[0];
    for (int i = 1; i < 5; i++) {
    if (arr[i] > max) {
     max = arr[i];
    }
    }
    cout << "Largest element: " << max << endl;
    return 0;
}
```

OUTPUT:

```
C:\Users\CS\Documents\12.exe
Largest element: 33

--------------------------------
Process exited after 0.05419 seconds with return value 0
Press any key to continue . . .
```

PROGRAM NO 7:

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    cout << "Reversed Array: ";
    for (int i = 4; i >= 0; i--) {
    cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```

OUTPUT:

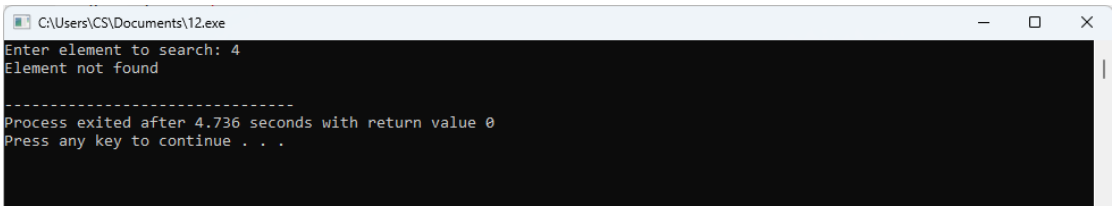
```
C:\Users\CS\Documents\12.exe
Reversed Array: 50 40 30 20 10

--------------------------------
Process exited after 0.06195 seconds with return value 0
Press any key to continue . . .
```

PROGRAM NO 8:

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

int main() {
    int arr[5] = {5, 8, 12, 20, 25};
    int search, found = -1;

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

    for (int i = 0; i < 5; i++) {
    if (arr[i] == search) {
    found = i;
    break;
     }
```

```
    }
    if (found != -1)
    cout << "Element found at index: " << found << endl;
    else
     cout << "Element not found" << endl;
    return 0;
}
```

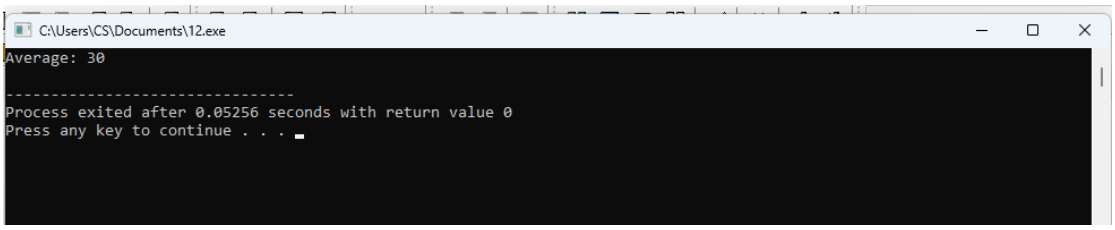
OUTPUT:



PROGRAM NO 9:

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

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int sum = 0;
    float average;
    for (int i = 0; i < 5; i++) {
    sum += arr[i];
    }
    average = sum / 5.0;
    cout << "Average: " << average << endl;
    return 0;
}
```

OUTPUT:

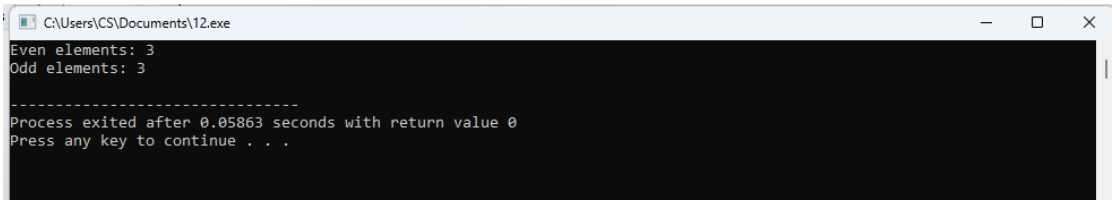


PROGRAM NO 10:

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

```
int main() {
    int arr[6] = {1, 2, 3, 4, 5, 6};
    int evenCount = 0, oddCount = 0;
    for (int i = 0; i < 6; i++) {
        if (arr[i] % 2 == 0)
            evenCount++;
        else
            oddCount++;
    }

    cout << "Even elements: " << evenCount << endl;
    cout << "Odd elements: " << oddCount << endl;
    return 0;
}
```

OUTPUT:



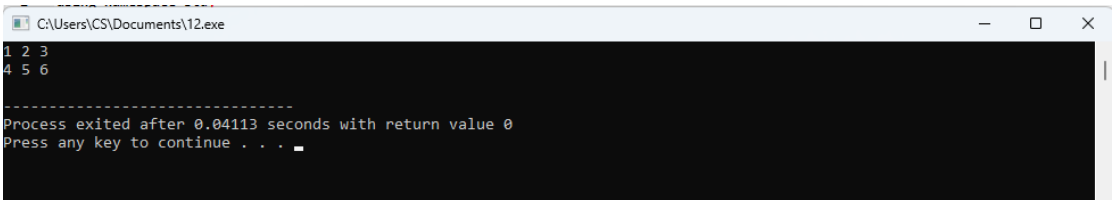
LAB NO 3

A **multidimensional array** is an array of arrays, where each element is itself an array. In a **2D array**, elements are arranged in rows and columns, forming a matrix-like structure. This allows you to store data in a tabular form, making it ideal for scenarios like storing matrices, tables, or grids.

PROGRAM NO 1:

```
#include <iostream>
using namespace std;
int main() {
    // Declare and initialize a 2x3 array (2 rows, 3 columns)
    int matrix[2][3] = {
        {1, 2, 3}, // First row
        {4, 5, 6}  // Second row
    };
    // Print the 2D array
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";  // Access elements using row and column indices
        }
        cout << endl;  // Newline after each row
    }
    return 0;
}
```

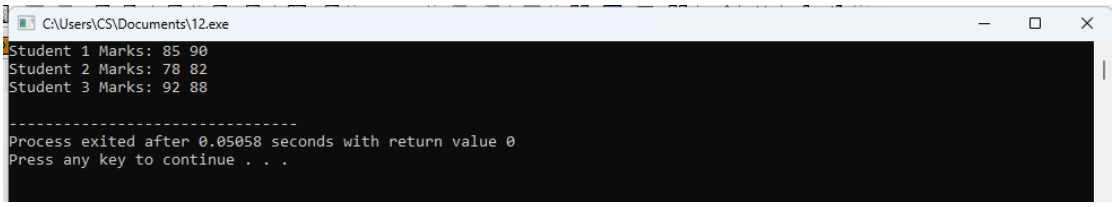
OUTPUT:



PROGRAM NO 2:

```
#include <iostream>
using namespace std;
int main() {
    // Declare and initialize a 3x2 array to store marks for 3 students in 2 subjects
    int marks[3][2] = {
        {85, 90},  // Marks for Student 1 in Subject 1 and Subject 2
```

```
        {78, 82},  // Marks for Student 2 in Subject 1 and Subject 2
        {92, 88}   // Marks for Student 3 in Subject 1 and Subject 2
    };
    // Display the marks
    for (int i = 0; i < 3; i++) {
        cout << "Student " << i+1 << " Marks: ";
        for (int j = 0; j < 2; j++) {
            cout << marks[i][j] << " ";  // Print each student's marks
        }
        cout << endl;
    }
    return 0;
}
```

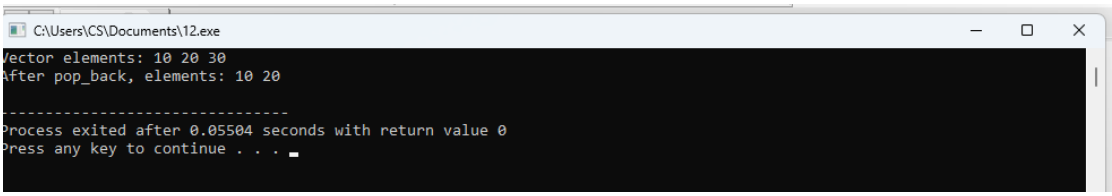


PROGRAM NO 3:

```
#include <iostream>
#include <vector> // Include the vector header
using namespace std;
int main() {
    // Create a vector to store integers
    vector<int> numbers;
    // Add elements to the vector
    numbers.push_back(10); // Add 10
    numbers.push_back(20); // Add 20
    numbers.push_back(30); // Add 30
    // Display the elements of the vector
    cout << "Vector elements: ";
    for (int i = 0; i < numbers.size(); i++) { // Use size() to get the number of elements
        cout << numbers[i] << " "; // Access elements using the index
    }
    cout << endl;
    // Remove the last element
    numbers.pop_back(); // Removes 30
    // Display the updated vector
    cout << "After pop_back, elements: ";
```

```
    for (int i = 0; i < numbers.size(); i++) {
        cout << numbers[i] << " "; // Print updated vector
    }
    cout << endl;
    return 0;
}
```

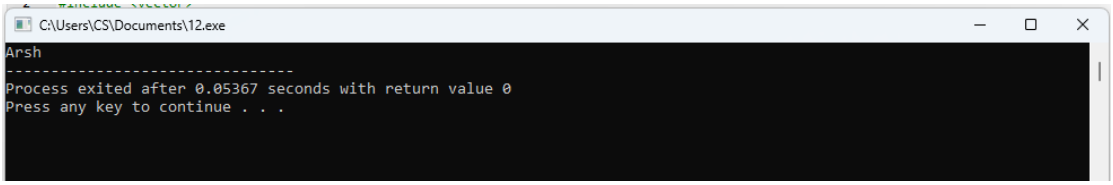
OUTPUT:



PROGRAM NO 4:

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

int main() {
  vector<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Change the value of the first element
  names[0] = "Arsh";

  cout << names[0];
  return 0;
}
```

OUTPUT:

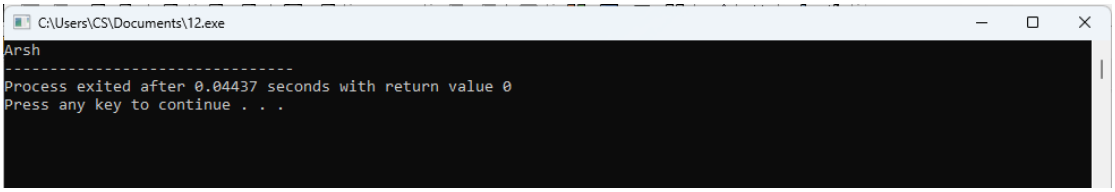


PROGRAM NO 5:

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

int main() {
```

```
  vector<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Change the value of the first element
  names.at(0) = "Arsh";

  cout << names.at(0);
  return 0;
}
```

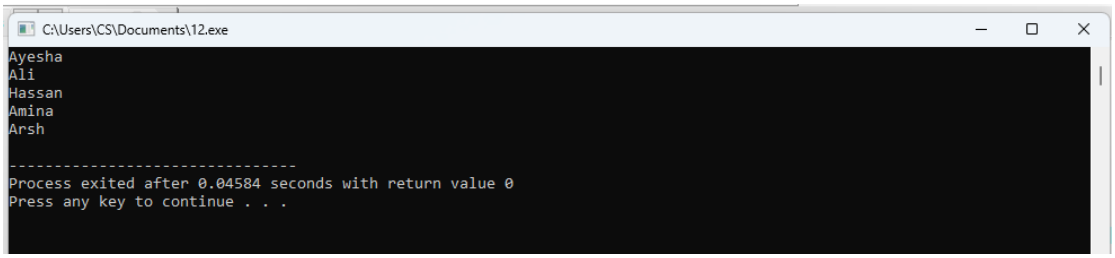
OUTPUT:



PROGRAM NO 6:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};
  names.push_back("Arsh");
  for (string name : names) {
    cout << name << "\n";
  }
  return 0;
}
```

OUTPUT:

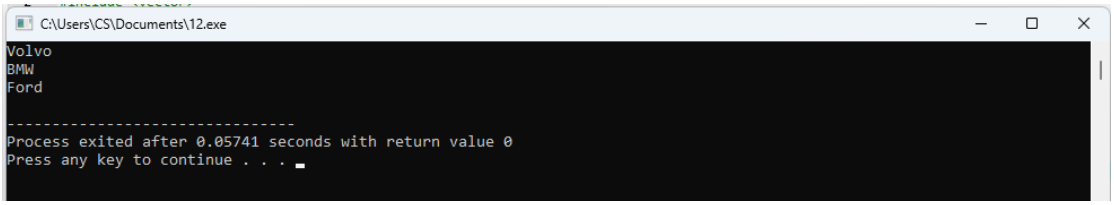


PROGRAM NO 7:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
```

```
  vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};
  cars.pop_back();
  for (string car : cars) {
    cout << car << "\n";
  }
  return 0;
}
```

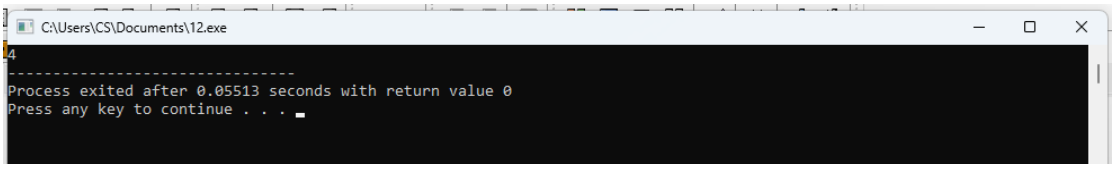
OUTPUT:



PROGRAM NO 8:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
 {
   vector<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};
   cout << names.size();
    return 0;
    }
```

OUTPUT:

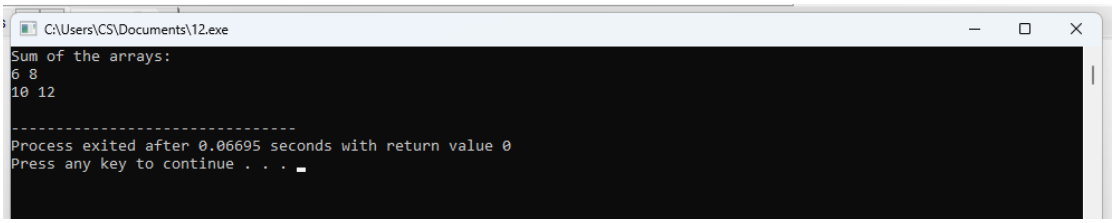


PROGRAM NO 9:

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

int main() {
   int arr1[2][2] = {{1, 2}, {3, 4}};
   int arr2[2][2] = {{5, 6}, {7, 8}};
   int sum[2][2];
```

```
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sum[i][j] = arr1[i][j] + arr2[i][j];
        }
    }

    cout << "Sum of the arrays:" << endl;
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << sum[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

OUTPUT:

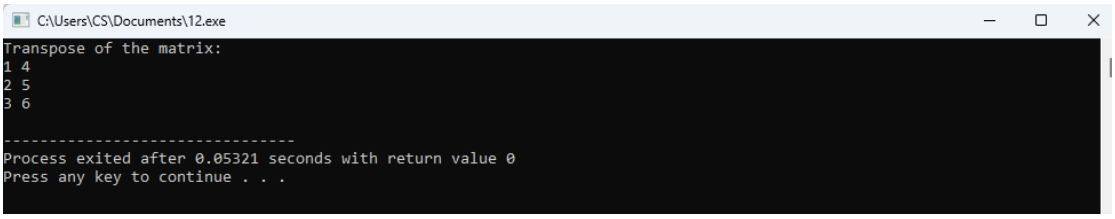


PROGRAM NO 10:

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

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int transpose[3][2];

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

    cout << "Transpose of the matrix:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 2; j++) {
```

```
            cout << transpose[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

OUTPUT:



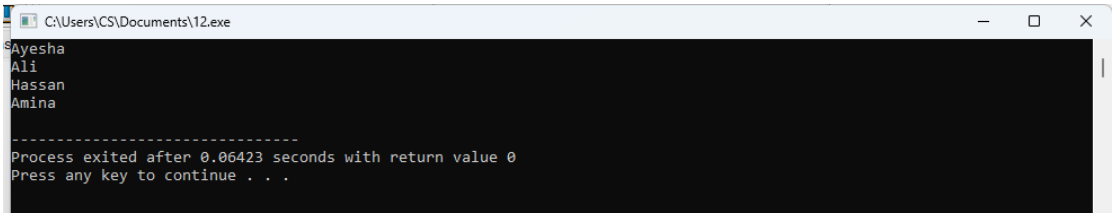
LAB NO 4

A list is similar to a vector in that it can store multiple elements of the same type and dynamically grow in size.

PROGRAM NO 1:

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

int main() {
  // Create a list called cars that will store strings
  list<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Print list elements
  for (string name: names) {
    cout << name << "\n";
  }
  return 0;
}
```

OUTPUT:

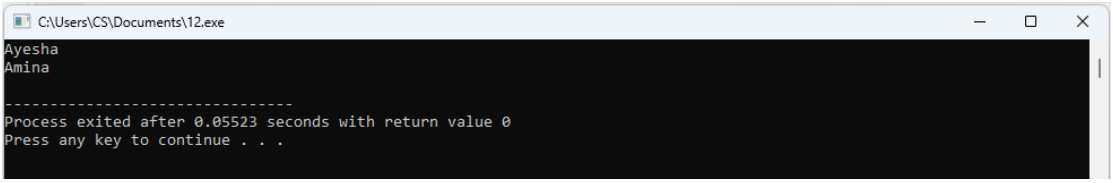


PROGRAM NO 2:

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

int main() {
  // Create a list called cars that will store strings
  list<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Get the first element
```

```
  cout << names.front() << "\n";

  // Get the last element
  cout << names.back() << "\n";
  return 0;
}
```

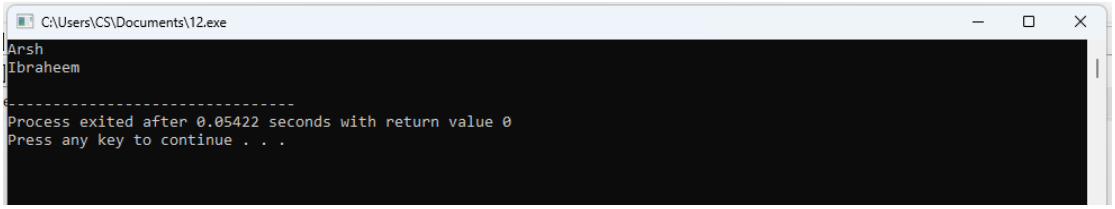
OUTPUT:



PROGRAM NO 3:

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

int main() {
  list<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Change the value of the first element
  names.front() = “Arsh ";

  // Change the value of the last element
  names.back() = "Ibraheem";

  cout << names.front() << "\n";
  cout << names.back() << "\n";
  return 0;
}
```

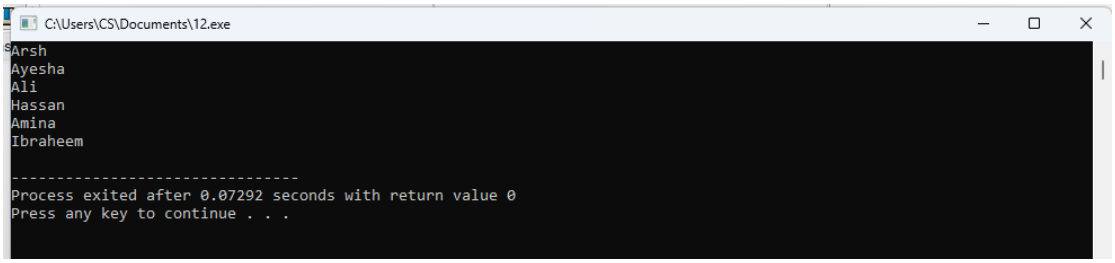
OUTPUT:



PROGRAM NO 4:

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

int main() {
  list<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Add an element at the beginning
  names.push_front("Arsh");

  // Add an element at the end
  names.push_back("Ibraheem");

  // Print list elements
  for (string name : names) {
    cout << name << "\n";
  }

  return 0;
}
```

OUTPUT:

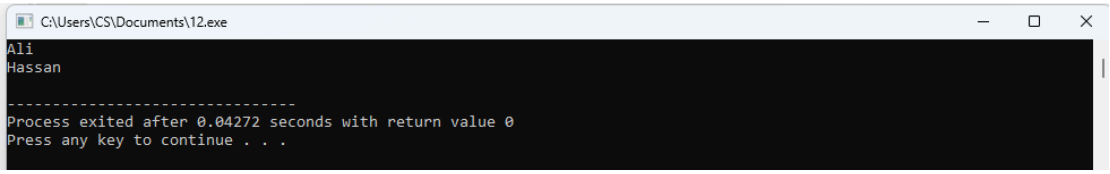


PROGRAM NO 5:

```
#include <iostream>
#include <list>
using namespace std;
int main() {
  list<string> names = {"Ayesha", "Ali", "Hassan", "Amina"};

  // Remove the first element
  names.pop_front();
```

```
  // Remove the last element
  names.pop_back();

  // Print list elements
  for (string name : names) {
    cout << name << "\n";
  }

  return 0;
}
```

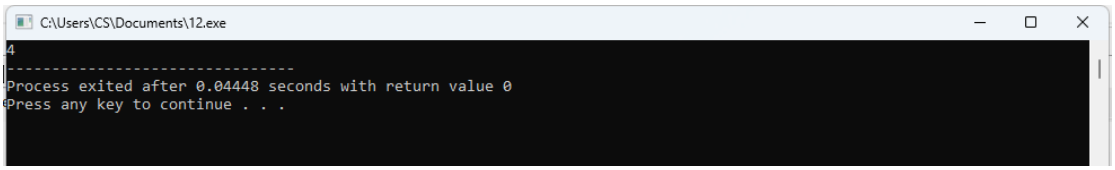
OUTPUT:



PROGRAM NO 6:

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

int main() {
  list<string> names = {"ayesha", "ali", "hassan", "amina"};
  cout << names.size();
  return 0;
}
```

OUTPUT:



PROGRAM NO 7:

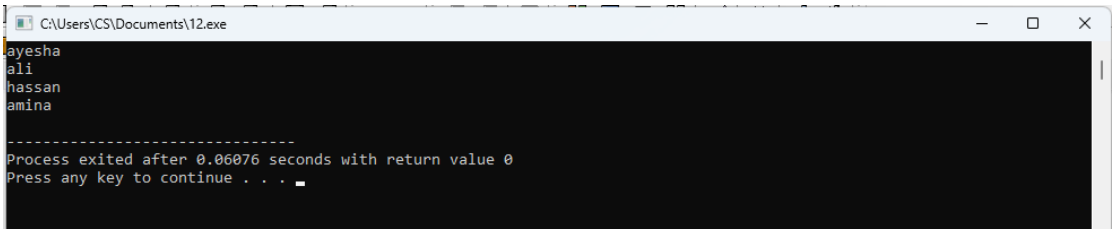
```
#include <iostream>
#include <list>
using namespace std;

int main() {
```

```
  list<string> names = {"ayesha", "ali", "hassan", "amina"};
  cout << names.empty();  // Outputs 0 (not empty)
  return 0;
}
```

PROGRAM NO 8:

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

int main() {
  // Create a list called cars that will store strings
  list<string> names = {"ayesha", "ali", "hassan", "amina"};

  // Print list elements
  for (string name : names) {
    cout << name << "\n";
  }
  return 0;
}
```

OUTPUT:



LAB NO 5

A stack stores multiple elements in a specific order, called LIFO.
LIFO stands for Last in, First Out. To vizualise LIFO, think of a pile of pancakes, where pancakes are both added and removed from the top. So when removing a pancake, it will always be the last one you added. This way of organizing elements is called LIFO in computer science and programming.

PROGRAM NO 1:

```
#include <stack>
#include<iostream>
Using namespace std;
stack<string> names;
stack<string> names = {"ayesha", "ali", "hassan", "amina"};
  cout << names.top();
  return 0;
}
```

OUTPUT:

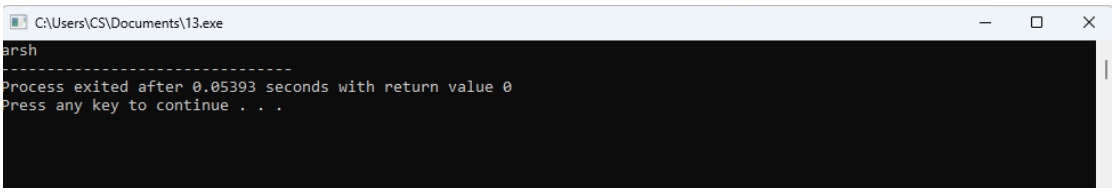
Ayesha

PROGRAM NO 2:

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

int main() {
  // Create a stack of strings

  stack<string> names;

  // Add elements to the stack
  names.push("ayesha");
  names.push("ali");
  names.push("hassan");
  names.push("amina");

  // Change the value of the top element
  names.top() = "arsh";
```

```
  // Access the top element
  cout << names.top();
  return 0;
}
```

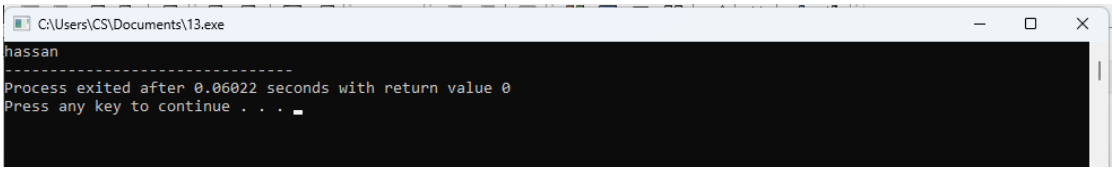
OUTPUT:



PROGRAM NO 3

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

int main() {
  // Create a stack of strings called cars
  stack<string> names;

  // Add elements to the stack
  names.push("ayesha");
  names.push("ali");
  names.push("hassan");
  names.push("amina");

  // Remove the last/latest added element
  names.pop();

  // Access the top element
  cout << names.top();
  return 0;
}
```

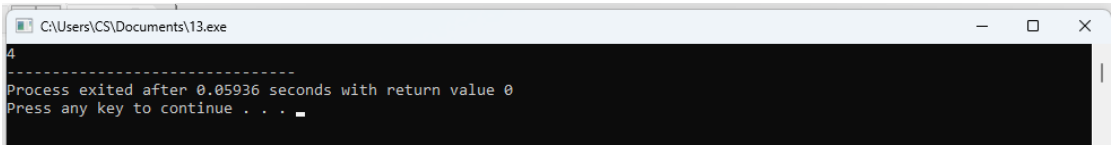
OUTPUT:



PROGRAM NO 4:

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

int main() {
  // Create a stack of strings called cars
  stack<string> names;

  // Add elements to the stack
  names.push("ayesha");
  names.push("ali");
  names.push("hassan");
  names.push("amina");
    // Get the size of the stack
  cout << names.size();
  return 0;
}
```

OUTPUT:

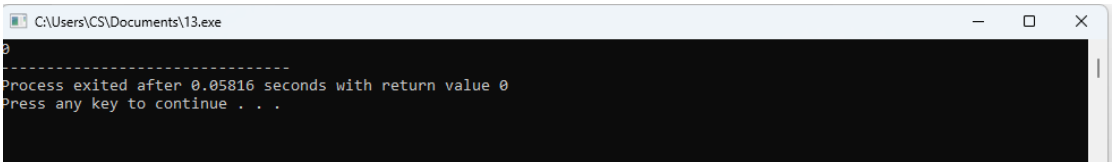


PROGRAM NO 5:

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

int main() {
  // Create a stack of strings called cars
  stack<string> names;

  // Add elements to the stack
  names.push("ayesha");
  names.push("ali");
  names.push("hassan");
  names.push("amina");
    // Get the size of the stack
  cout << names.empty();
```

```
  return 0;
}
```

OUTPUT:



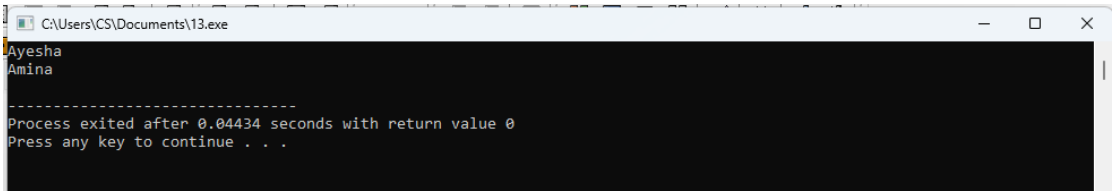
LAB NO 6

A queue stores multiple elements in a specific order, called **FIFO**.
FIFO stands for **First in, First Out**. To visualize FIFO, think of a queue as people standing in line in a supermarket. The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO in computer science and programming.

PROGRAM NO 1:

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

int main() {
  // Create a queue of strings
  queue<string> names;

  // Add elements to the queue
  names.push("Ayesha");
  names.push("Ali");
  names.push("Hassan");
  names.push("Amina");

  // Access the front element (first and oldest)
  cout << names.front() << "\n";

  // Access the back element (last and newest)
  cout << names.back() << "\n";
  return 0;
}
```

OUTPUT:

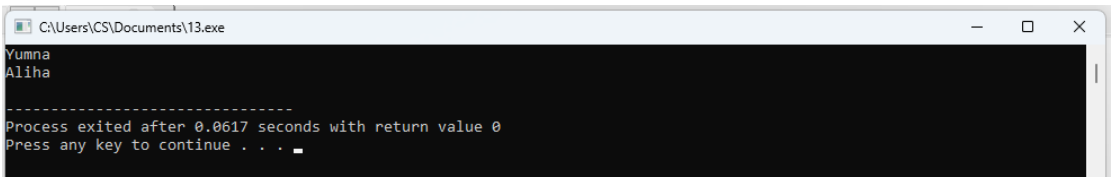


PROGRAM NO 2:

```
#include <iostream>
```

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

int main() {
  // Create a queue of strings
  queue<string> names;

  // Add elements to the queue
  names.push("Ayesha");
  names.push("Ali");
  names.push("Hassan");
  names.push("Amina");

// Change the value of the front element
  names.front() = "Yumna";

  // Change the value of the back element
  names.back() = "Aliha";

  // Access the front element (first and oldest)
  cout << names.front() << "\n";

  // Access the back element (last and newest)
  cout << names.back() << "\n";
  return 0;
}
```

OUTPUT:

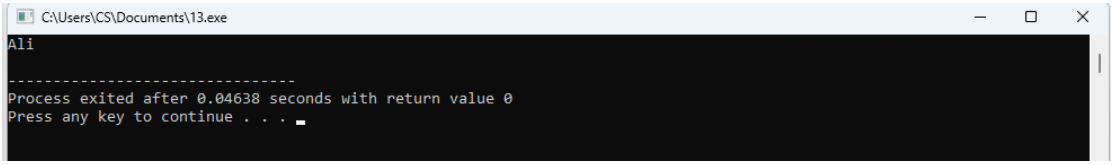


PROGRAM NO 3:

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

int main() {
  // Create a queue of strings
```

```
  queue<string> names;

  // Add elements to the queue
  names.push("Ayesha");
  names.push("Ali");
  names.push("Hassan");
  names.push("Amina");

  // Remove the front element
  names.pop();

  // Access the front element (first and oldest)
  cout << names.front() << "\n";

  return 0;
}
```

OUTPUT:

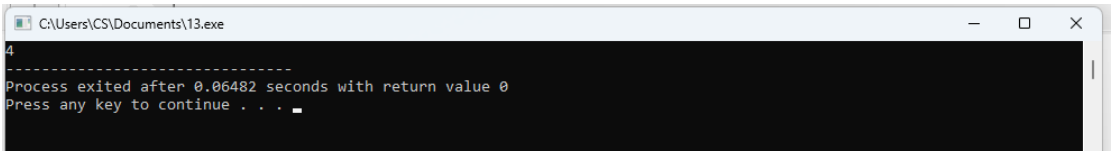


PROGRAM NO 4:

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

int main() {
  // Create a queue of strings
  queue<string> names;

  // Add elements to the queue
  names.push("Ayesha");
  names.push("Ali");
  names.push("Hassan");
  names.push("Amina");

  // Get the size of the queue
  cout << names.size();
```

```
  return 0;
}
```

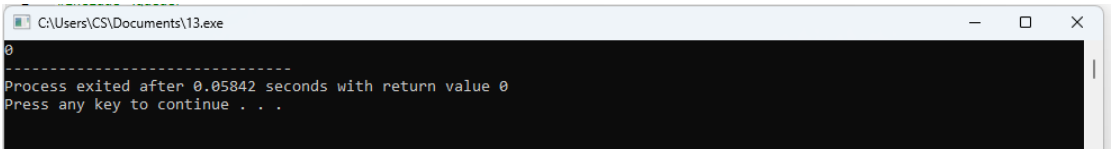
OUTPUT:



PROGRAM NO 5:

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

int main() {
  // Create a queue of strings
  queue<string> names;

  // Add elements to the queue
  names.push("Ayesha");
  names.push("Ali");
  names.push("Hassan");
  names.push("Amina");

// Check if the queue is empty
  cout << names.empty();
  return 0;
}
```

OUTPUT:



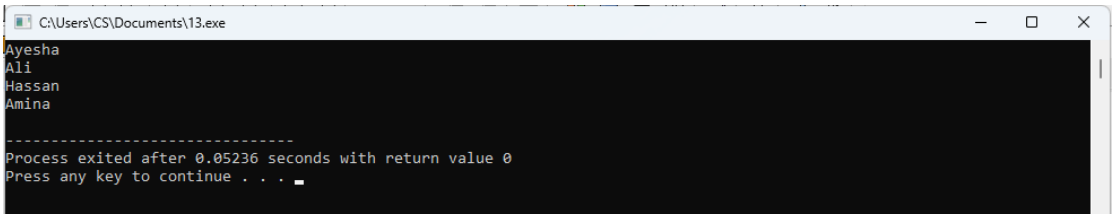
LAB NO 7

A deque (stands for **d**ouble-**e**nded **queue**) however, is more flexible, as elements can be added and removed from both ends (at the front and the back). You can also access elements by index numbers.

PROGRAM NO 1:

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

int main() {
  // Create a deque called names that will store strings
  deque<string> names = {"Ayesha","Ali", "Hassan", "Amina"};

  // Print deque elements
  for (string name : names) {
    cout << name << "\n";
  }
  return 0;
}
```

OUTPUT:

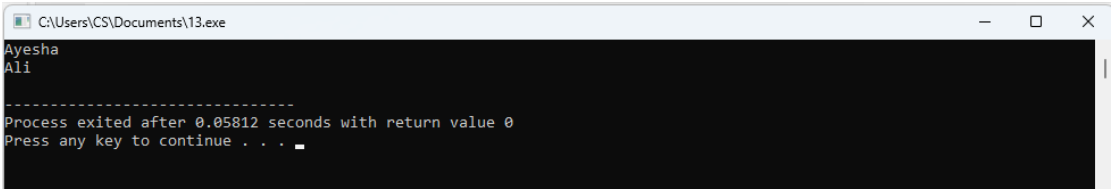


PROGRAM NO 2:

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

int main() {
  // Create a deque called names that will store strings
  deque<string> names = {"Ayesha","Ali", "Hassan", "Amina"};
```

```
  // Get the first element
 cout << names[0] << "\n";

 // Get the second element
 cout << names[1] << "\n"
 return 0;
}
```

OUTPUT:



PROGRAM NO 3:

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

int main() {
 // Create a deque called names that will store strings
 deque<string> names = {"Ayesha","Ali", "Hassan", "Amina"};


  // Get the first element
 cout << names.front<< "\n";

 // Get the second element
 cout << names.back << "\n"
 return 0;
}
```

OUTPUT:

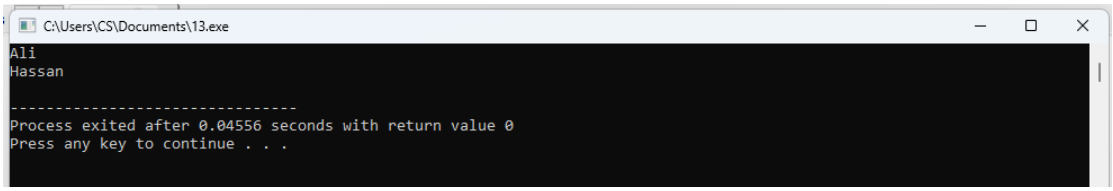
Ayesha
Amina

PROGRAM NO 4:

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

```
int main() {
  // Create a deque called names that will store strings
  deque<string> names = {"Ayesha","Ali", "Hassan", "Amina"};

  // Get the second element
  cout << names.at(1) << "\n";

  // Get the third element
  cout << names.at(2) << "\n";"
  return 0;
}
```

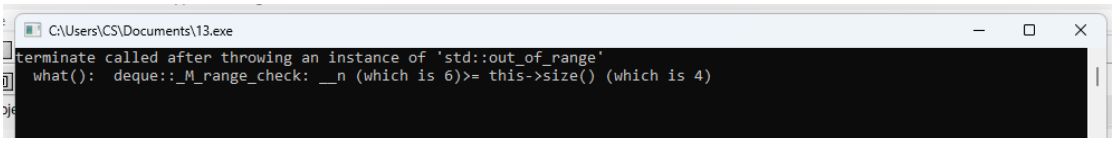
OUTPUT:



PROGRAM NO 5:

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

int main() {
  // Create a deque called names that will store strings
  deque<string> names = {"Ayesha","Ali", "Hassan", "Amina"};

  // Try to access an element that does not exist (will throw an exception)
  cout << cars.at(6)
  return 0;
}
```

OUTPUT:



LAB NO 8

LINK LIST :

PROGRAM 1 : INSERTION AT FIRST:

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = head;
   head = newNode;
}
void traverse() {
   Node* temp = head;
   if (temp == NULL) {
      cout << "List is empty.\n";
      return;
   }

   while (temp != NULL) {
      cout << temp->data << " -> ";
      temp = temp->next;
   }
   cout << "NULL\n";
}
int main() {
   insertFront(10);
   insertFront(20);
   insertFront(30);
```

```
    insertFront(40);

    cout << "List after insertions:\n";
    traverse();}
```

OUTPUT:

PROGRAM 2:INSERTION AT END:

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertEnd(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
        return;
    }

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

```
void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertEnd(10);
    insertEnd(20);
    insertEnd(30);
    insertEnd(40);



     cout << "List after insertions:\n";
     traverse();}
```

OUTPUT:

PROGRAM 3: INSERTION AT ANY POINT

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

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

```
Node* head = NULL; // Global pointer for the head of the list
void insertAfter(int afterValue, int value) {
    Node* temp = head;
    while (temp != NULL && temp->data != afterValue) {
        temp = temp->next;
    }

    if (temp == NULL) {
        cout << "Value " << afterValue << " not found in the list.\n";
        return;
    }

    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = temp->next;
    temp->next = newNode;
}

void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertAfter(10,5);
            insertAfter(5,6);
    insertAfter(6,2);
 cout << "List after insertions:\n";
    traverse();}
```

OUTPUT:

```
C:\Users\Ayaan\Desktop\dssss\Untitled3.exe
Value 10 not found in the list.
Value 5 not found in the list.
Value 6 not found in the list.
List after insertions:
List is empty.

--------------------------------
Process exited after 0.1375 seconds with return value 0
Press any key to continue . . .
```

PROGRAM 4: DELETION AT FIRST:

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
void deleteFront() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }

    Node* temp = head;
    head = head->next;
    delete temp;
}
void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }
```

```
    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);
    deleteFront();

     cout << "List after deletion at first:\n";
traverse();}
```

OUTPUT:

PROGRAM NO 5: DELETION AT END

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
    Node* newNode = new Node();
```

```
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
void deleteEnd() {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }

    if (head->next == NULL) {
        delete head;
        head = NULL;
        return;
    }

    Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }

    delete temp->next;
    temp->next = NULL;
}

void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertFront(10);
```

```
    insertFront(20);
    insertFront(30);
    insertFront(40);
   deleteEnd();

     cout << "List after deletion at end:\n";
traverse();}
```

OUTPUT:

PROGRAM NO 6: DELETON AT ANY POINT:

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
void deleteValue(int value) {
    if (head == NULL) {
        cout << "List is empty.\n";
        return;
    }
```

```
    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

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

    if (temp->next == NULL) {
        cout << "Value " << value << " not found in the list.\n";
        return;
    }

    Node* nodeToDelete = temp->next;
    temp->next = temp->next->next;
    delete nodeToDelete;
}

void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);
```

```
    deleteValue(30);


     cout << "List after deletion at any point:\n";
traverse();}
```

OUTPUT:

PROGRAM NO 7: SEARCHING

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
bool search(int value) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == value) {
            return true;
        }
        temp = temp->next;
    }
```

```
    return false;
}
void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);


     cout << "List after deletion at any point:\n";
traverse();
cout << "Searching 30: " << (search(30) ? "Found" : "Not Found") << "\n";
}
```

OUTPUT:

PROGRAM NO 8: SEARCH INDEX:

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

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

Node* head = NULL; // Global pointer for the head of the list
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    newNode->next = head;
    head = newNode;
}
int findIndex(int value) {
    Node* temp = head;
    int index = 0;
    while (temp != NULL) {
        if (temp->data == value) {
            return index;
        }
        temp = temp->next;
        index++;
    }
    return -1; // Value not found
}

void traverse() {
    Node* temp = head;
    if (temp == NULL) {
        cout << "List is empty.\n";
        return;
    }

    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
int main() {
```

```
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);


     cout << "List after deletion at any point:\n";
traverse();
cout << "Index of 30: " << findIndex(30) << "\n";
}
```

OUTPUT:

LAB NO 9

DOUBLE LINK LIST :

PROGRAM NO 1: INSERTION AT START END AND AT ANY POINT:

```
// Node structure
#include <iostream>
using namespace std;

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

Node *head = nullptr; // Global head pointer

// Insert at the front
void insertFront(int value) {
    Node *newNode = new Node;
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = head;
    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}

// Insert at the end
void insertLast(int value) {
    Node *newNode = new Node;
    newNode->data = value;
    newNode->next = nullptr;
    if (head == nullptr) {
        newNode->prev = nullptr;
```

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

// Insert at a specific position
void insertMid(int value, int position) {
    if (position == 0) {
        insertFront(value);
        return;
    }
    Node *newNode = new Node;
    newNode->data = value;
    Node *temp = head;
    for (int i = 0; temp != nullptr && i < position - 1; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Position out of range" << endl;
        return;
    }
    newNode->next = temp->next;
    newNode->prev = temp;
    if (temp->next != nullptr) {
        temp->next->prev = newNode;
    }
    temp->next = newNode;
}
// Traverse and display the list
void traverse() {
    Node *temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
```

```
    }
    cout << endl;
}
int main() {
    // Separate tests for each function

    // Test Insertions
    insertFront(10);
    insertFront(20);
    traverse(); // 20 10

    insertLast(30);
    traverse(); // 20 10 30

    insertMid(25, 2);
    traverse(); // 20 10 25 30
 return 0;
}
```

OUTPUT:

PROGRAM NO 2: DELETION AT START, END AND AT ANY POINT:

```
// Node structure
#include <iostream>
using namespace std;

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

Node *head = nullptr; // Global head pointer
```

```
// Insert at the front
void insertFront(int value) {
    Node *newNode = new Node;
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = head;
    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}

// Delete from the front
void deleteFront() {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
    Node *temp = head;
    head = head->next;
    if (head != nullptr) {
        head->prev = nullptr;
    }
    delete temp;
}

// Delete from the end
void deleteLast() {
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
    Node *temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    if (temp->prev != nullptr) {
        temp->prev->next = nullptr;
    } else {
        head = nullptr;
    }
    delete temp;
}

// Delete from a specific position
void deleteMid(int position) {
```

```
    if (head == nullptr) {
        cout << "List is empty" << endl;
        return;
    }
    if (position == 0) {
        deleteFront();
        return;
    }
    Node *temp = head;
    for (int i = 0; temp != nullptr && i < position; i++) {
        temp = temp->next;
    }
    if (temp == nullptr) {
        cout << "Position out of range" << endl;
        return;
    }
    if (temp->next != nullptr) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != nullptr) {
        temp->prev->next = temp->next;
    }
    delete temp;
}
// Traverse and display the list
void traverse() {
    Node *temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
int main() {
    // Separate tests for each function

    // Test Insertions
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);
    insertFront(50);
    traverse();
    // Test Deletions
    deleteFront();
    traverse();
```

```
    deleteLast();
    traverse();
    deleteMid(1);
    traverse();

    return 0;
}
```

OUTPUT:

PROGRAM N 3: SEARCHING AND FINDING INDEX:

```
// Node structure
#include <iostream>
using namespace std;

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

Node *head = nullptr; // Global head pointer
// Insert at the front
void insertFront(int value) {
    Node *newNode = new Node;
    newNode->data = value;
    newNode->prev = nullptr;
    newNode->next = head;
    if (head != nullptr) {
        head->prev = newNode;
    }
    head = newNode;
}

// Search for an element
int search(int value) {
```

```
    Node *temp = head;
    int index = 0;
    while (temp != nullptr) {
        if (temp->data == value) {
            return index;
        }
        temp = temp->next;
        index++;
    }
    return -1;
}
// Find the index of an element
int findIndex(int value) {
    return search(value);
}

// Traverse and display the list
void traverse() {
    Node *temp = head;
    while (temp != nullptr) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}
int main() {
    // Separate tests for each function

    // Test Insertions
    insertFront(10);
    insertFront(20);
    traverse(); // 20 10
  // Test Search and Edit
    cout << "Index of 10: " << search(10) << endl; // 0
     // Test Finding Index
    cout << "Index of 15: " << findIndex(15) << endl; // 0



    return 0;
}
```

OUTPUT:

C:\Users\Ayaan\Desktop\ds sem 4\duble link list searching and index.exe

```
20 10
Index of 10: 1
Index of 15: -1

--------------------------------
Process exited after 0.146 seconds with return value 0
Press any key to continue . . .
```

LAB NO 10

CIRCULAR LINK LIST :

PROGRAM NO 1: INSERTION AT START , END AND AT ANY POINT:

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

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

Node* head = NULL;

// Insert at the Front
void insertFront(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   if (head == NULL) {
      newNode->next = newNode;
      head = newNode;
   } else {
      Node* temp = head;
      while (temp->next != head) {
         temp = temp->next;
      }
      newNode->next = head;
      temp->next = newNode;
      head = newNode;
   }
}

// Insert at the Last
void insertLast(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   if (head == NULL) {
```

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

// Insert in the Middle
void insertMiddle(int value, int position) {
    Node* newNode = new Node();
    newNode->data = value;
    if (position == 1) {
        insertFront(value);
        return;
    }
    Node* temp = head;
    for (int i = 1; i < position - 1 && temp->next != head; i++) {
        temp = temp->next;
    }
    newNode->next = temp->next;
    temp->next = newNode;
}
// Traverse the Circular Linked List
void traverse() {
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
```

```
int main() {
    insertFront(10);
    insertLast(20);
    insertLast(30);
    insertMiddle(15, 2);

    cout << "List after insertions: ";
    traverse();
     return 0;
}
```

OUTPUT:

PROGRAM NO 2: DELETION AT FRONT,LAST AND MID:

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

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

Node* head = NULL;
// Insert at the Front
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    if (head == NULL) {
        newNode->next = newNode;
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
```

```
        newNode->next = head;
        temp->next = newNode;
        head = newNode;
    }
}
// Delete from the Front
void deleteFront() {
    if (head == NULL) return;
    Node* temp = head;
    if (head->next == head) {
        head = NULL;
        delete temp;
    } else {
        Node* last = head;
        while (last->next != head) {
            last = last->next;
        }
        head = head->next;
        last->next = head;
        delete temp;
    }
}

// Delete from the Last
void deleteLast() {
    if (head == NULL) return;
    Node* temp = head;
    if (head->next == head) {
        head = NULL;
        delete temp;
    } else {
        Node* prev = NULL;
        while (temp->next != head) {
            prev = temp;
            temp = temp->next;
        }
        prev->next = head;
        delete temp;
    }
}

// Delete from the Middle
void deleteMiddle(int position) {
    if (head == NULL) return;
    if (position == 1) {
        deleteFront();
```

```
        return;
    }
    Node* temp = head;
    Node* prev = NULL;
    for (int i = 1; i < position && temp->next != head; i++) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == head) return;
    prev->next = temp->next;
    delete temp;
}
// Traverse the Circular Linked List
void traverse() {
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}

int main() {
    insertFront(10);
    insertFront(20);
    insertFront(30);
    insertFront(40);
     cout << "List after insertions: ";
    traverse();
     deleteFront();
    cout << "List after deleting front: ";
    traverse();

    deleteLast();
    cout << "List after deleting last: ";
    traverse();

    deleteMiddle(2);
    cout << "List after deleting middle: ";
    traverse();

    return 0;
```

```
}
```

OUTPUT:

PROGRAM NO 3: SEARCHING AND FINDING INDEX

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

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

Node* head = NULL;

// Insert at the Front
void insertFront(int value) {
    Node* newNode = new Node();
    newNode->data = value;
    if (head == NULL) {
        newNode->next = newNode;
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != head) {
            temp = temp->next;
        }
        newNode->next = head;
        temp->next = newNode;
        head = newNode;
    }
}
// Search for an Element
bool search(int value) {
    if (head == NULL) return false;
```

```
    Node* temp = head;
    do {
        if (temp->data == value) return true;
        temp = temp->next;
    } while (temp != head);
    return false;
}
// Find the Index of an Element
int findIndex(int value) {
    if (head == NULL) return -1;
    Node* temp = head;
    int index = 0;
    do {
        if (temp->data == value) return index;
        temp = temp->next;
        index++;
    } while (temp != head);
    return -1;
}

// Traverse the Circular Linked List
void traverse() {
    if (head == NULL) {
        cout << "List is empty" << endl;
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
    cout << endl;
}
int main() {
    insertFront(10);
    insertFront(20);
    insertFront(25);
    insertFront(30);
    cout << "List after insertions: ";
    traverse();
    cout << "Searching for 20: " << (search(20) ? "Found" : "Not Found") << endl;
    cout << "Index of 25: " << findIndex(25) << endl;

    return 0;
}
```

OUTPUT:

C:\Users\Ayaan\Desktop\ds sem 4\circular link list seraching.exe

```
List after insertions: 30 25 20 10
Searching for 20: Found
Index of 25: 1

--------------------------------
Process exited after 0.1473 seconds with return value 0
Press any key to continue . . .
```

LAB NO 11

BINARY SEARCH TREE :

PROGRAM NO 1: INSERTION AND TRAVERSING(IN,PRE AND POST ORDER)

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

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

// Function to insert a value into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}
// In-order traversal (Left, Root, Right)
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
```

```
        cout << root->data << " ";
        inOrder(root->right);
    }
}

// Pre-order traversal (Root, Left, Right)
void preOrder(Node* root) {
    if (root != nullptr) {
        cout << root->data << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

// Post-order traversal (Left, Right, Root)
void postOrder(Node* root) {
    if (root != nullptr) {
        postOrder(root->left);
        postOrder(root->right);
        cout << root->data << " ";
    }
}
int main() {
    Node* root = nullptr;

    // Insert nodes
    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);

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Pre-order Traversal: ";
    preOrder(root);
```

```
    cout << endl;

    cout << "Post-order Traversal: ";
    postOrder(root);
    cout << endl;
     return 0;
}
```

OUTPUT:

PROGRAM NO 2: DELETION:

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

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

// Function to insert a value into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }
    if (value < root->data) {
```

```
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}
// Function to find the minimum value in a BST
Node* findMin(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}
// Function to delete a value from the BST
Node* deleteNode(Node* root, int value) {
    if (root == nullptr) {
        return root;
    }
    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
        root->right = deleteNode(root->right, value);
    } else {
        // Node with only one child or no 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;
        }
        // Node with two children: Get the inorder successor
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
// In-order traversal (Left, Root, Right)
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
```

```
    }
}
int main() {
    Node* root = nullptr;

    // Insert nodes
    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);

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;
// Delete a node
    root = deleteNode(root, 50);
    cout << "In-order Traversal after deletion: ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

OUTPUT:

PROGRAM NO 3: SEARCHING

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

struct Node {
    int data;
    Node* left;
    Node* right;
};
```

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

// Function to insert a value into the BST
Node* insert(Node* root, int value) {
    if (root == nullptr) {
        return createNode(value);
    }
    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }
    return root;
}
// Function to search for a value in the BST
bool search(Node* root, int value) {
    if (root == nullptr) {
        return false;
    }
    if (root->data == value) {
        return true;
    } else if (value < root->data) {
        return search(root->left, value);
    } else {
        return search(root->right, value);
    }
}

// In-order traversal (Left, Root, Right)
void inOrder(Node* root) {
    if (root != nullptr) {
        inOrder(root->left);
        cout << root->data << " ";
        inOrder(root->right);
    }
}
int main() {
    Node* root = nullptr;
```

```
    // Insert nodes
    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);

    cout << "In-order Traversal: ";
    inOrder(root);
    cout << endl;
     // Search for a value
    int key = 40;
    if (search(root, key)) {
        cout << key << " found in the BST." << endl;
    } else {
        cout << key << " not found in the BST." << endl;
    }

    return 0;
}
```

OUTPUT: