THE UNIVERSITY OF FAISALABAD

LAB-Manual

1-14

PREPARED BY

M. Faisal Kamran BS-AI-025

PREPARED FOR

Mam Irsha

Table of Contents

Lab 1: Introduction to DSA

What are data structures

- A data structure is a way to store data.
- We structure data in different ways depending on what data we have, and what we want to do with it

What are algorithms

- An algorithm is a set of step-by-step instructions to solve a given problem or achieve a specific goal
- Algorithms are fundamental to computer programming as they provide step-by-step instructions for executing tasks

Types of data structures

- There are two types of data structures
    1. **Primitive Data Structures:** Basic data structures provided by programming languages to represent single values, such as integers, floating-point numbers, characters, and Booleans.
    2. **Abstract Data Structures**: Higher-level data structures that are built using primitive data types and provide more complex and specialized operations. They are further divided into two types
        - **Linear:** Which have fixed size like arrays, list, stack and queue
        - **Non-Linear:** Which do not have fixed size like trees and graphs

Where are data structures used

- Operating Systems
- Database Systems
- Web Applications
- Machine Learning
- Video Games

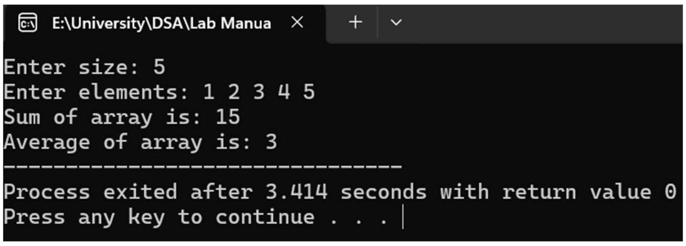
Lab 2: Array

- **Definition:** 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
- **Syntax:** data-type variable name [size] = {elements}
- **Key Characteristics of an Array:**
  1. Fixed Size
  2. Same Type
  3. Indexing
- **Example Programs**

1. Find the sum and average of an array

```
#include<iostream>
using namespace std;
int main() {
    int n,sum=0,average;
    cout << "Enter size: ";
    cin >> n;
    int arr[n];
    cout << "Enter elements: ";
    for (int i = 0; i < n; i++) {
    cin >> arr[i];
    }
    for (int i = 0; i < n; i++) {
    sum +=arr[i];
    }
    average = sum/n;
    cout<<"Sum of array is: "<<sum<<endl;
    cout<<"Average of array is: "<<average;
}
```

Output

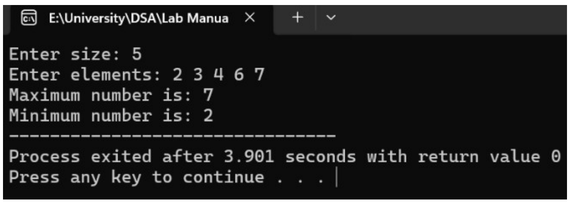


```
E:\University\DSA\Lab Manua
Enter size: 5
Enter elements: 1 2 3 4 5
Sum of array is: 15
Average of array is: 3
--------------------------------
Process exited after 3.414 seconds with return value 0
Press any key to continue . . .
```

2. Find the Maximum and Minimum of an array

```
#include<iostream>
using namespace std;
int main() {
    int mx = INT_MIN;
    int mn = INT_MAX;
    int n;
    cout << "Enter size: ";
```

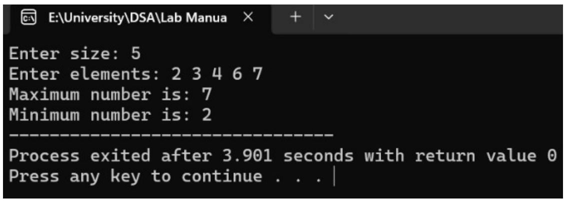
Output



```
E:\University\DSA\Lab Manua
Enter size: 5
Enter elements: 2 3 4 6 7
Maximum number is: 7
Minimum number is: 2
--------------------------------
Process exited after 3.901 seconds with return value 0
Press any key to continue . . .
```

```
    cin >> n;
    int arr[n];

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

    for (int i = 0; i < n; i++) {
    mx = max(mx, arr[i]);
    }
    cout << "Maximum number is: " << mx << endl;

    for (int i = 0; i < n; i++) {
        mn = min(mn, arr[i]);
    }
    cout << "Minimum number is: " << mn;
    return 0;
}
```

Output



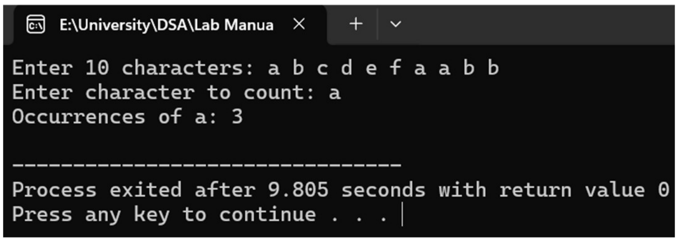
3. Reverse an Array

```
#include <iostream>
using namespace std;
int main() {
    int n=8;
    int originalArray[n]={1,2,3,4,5,6,7,8};
    int reversedArray[n];
    for (int i = 0; i < n; ++i) {
        reversedArray[i] = originalArray[n - 1 - i];
    }
    cout << "Original Array: ";
    for (int i = 0; i < n; ++i) {
        cout << originalArray[i] << " ";
    }
    cout << endl;
    cout << "Reversed Array: ";
    for (int i = 0; i < n; ++i) {
    cout << reversedArray[i] << " ";
    }
    cout << endl;
    return 0;}
```

4. Count characters in an array

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

int main() {
    char arr[10];
    char check;
    cout<<"Enter 10 characters: ";
    for(int i = 0; i < 10; i++) {
        cin >> arr[i];
    }
    cout<<"Enter character to count: ";
    cin>>check;
    int count = 0;
    for(int i = 0; i < 10; i++) {
        if(arr[i] == check) count++;
    }
    cout << "Occurrences of " << check << ": " << count << endl;
    return 0;
}
```

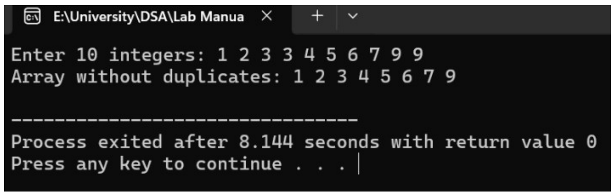
Output



5. Find duplicates in an array

```
#include <iostream>
using namespace std;
int main() {
    int arr[10], NewArr[10], Count = 0;
    cout << "Enter 10 integers: ";
    for (int i = 0; i < 10; i++) {
        cin >> arr[i];
        bool Duplicate = false;
        // Check if arr[i] is already in NewArr
        for (int j = 0; j < Count; j++) {
            if (arr[i] == NewArr[j]) {
                Duplicate = true;
                break;
            }
        }
        // If not a duplicate, add to NewArr
        if (!Duplicate) {

NewArr[Count] = arr[i];
```

Output

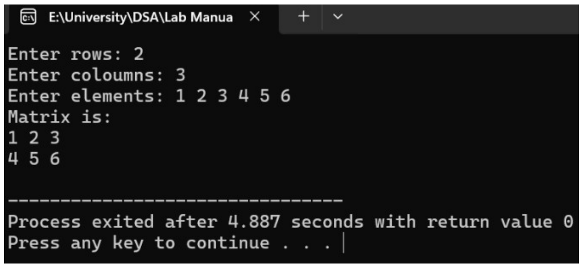


```
            Count++;
        }
    }
    cout << "Array without duplicates: ";
    for (int i = 0; i < Count; i++) {
        cout << NewArr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Lab 3: 2d Array

- **Definition:** A 2D array is a data structure that stores a fixed-size table-like collection of elements of the same type, organized in rows and columns. It is essentially an array of arrays, where each element is accessed using two indices: one for the row and one for the column.
- **Syntax:** data-type variable_name [rows][columns] = {{row1_elements}, {row2_elements}, ...};
- **Key Characteristics of a 2d Array:**
  1. Fixed Size
  2. Same Type
  3. Indexing
- **Example Programs**

Output



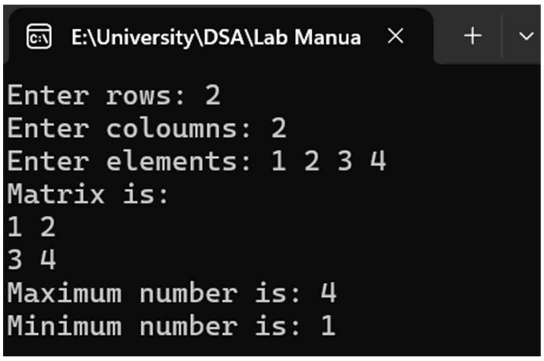
1. **Write Basic Implementation of 2d array**

```
#include<iostream>
using namespace std;
int main(){
        int n,m;
        cout<<"Enter rows: ";
        cin>>n;
        cout<<"Enter coloumns: ";
        cin>>m;
        int arr[n] [m];
        cout<<"Enter elements: ";
        for(int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
        cin>> arr[i][j];
}
}
        cout<<"Matrix is: "<<endl;
        for(int i=0;i<n;i++) {
        for (int j=0; j<m;j++) {
        cout<<arr[i][j]<<" ";
}
        cout<<"\n";
}
}
```

2. Find the Maximum and Minimum of an array

```
#include<iostream>
using namespace std;
int main(){
        int n,m;
        int mx=INT_MIN;
        int mn=INT_MAX;
        cout<<"Enter rows: ";
        cin>>n;
        cout<<"Enter coloumns: ";
        cin>>m;
        int arr[n] [m];
        cout<<"Enter elements: ";
        for(int i=0;i<n;i++) {
        for (int j=0;j<m;j++) {
        cin>> arr[i][j];}}
        cout<<"Matrix is: "<<endl;
        for(int i=0;i<n;i++) {
        for (int j=0; j<m;j++) {
        cout<<arr[i][j]<<" ";}
        cout<<"\n";}
        for(int i=0;i<n;i++) {
        for (int j=0; j<m;j++) {
        mx=max(mx,arr[i][j]);}        }
        cout << "Maximum number is: " << mx << endl;
        for(int i=0;i<n;i++) {
        for (int j=0; j<m;j++) {
        mn=min(mn,arr[i][j]);}        }
        cout << "Minimum number is: " << mn << endl;}
```

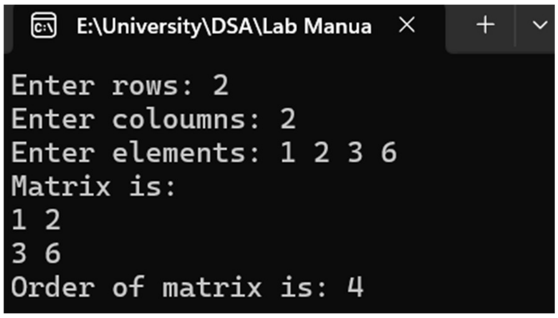
Output



3. Find the order of matrix in a 2d array

```
#include<iostream>
using namespace std;
int main(){
        int n,m,order;
        cout<<"Enter rows: ";
        cin>>n;
        cout<<"Enter coloumns: ";
        cin>>m;
        int arr[n] [m];
```

Output



```
            cout<<"Enter elements: ";
            for(int i=0;i<n;i++) {
            for (int j=0;j<m;j++) {
            cin>> arr[i][j];
    }
    }
            cout<<"Matrix is: "<<endl;
            for(int i=0;i<n;i++) {
            for (int j=0; j<m;j++) {
            cout<<arr[i][j]<<" ";
    }
            cout<<"\n";
    }
            order= n*m;
            cout<<"Order of matrix is: "<<order;
    }
```

4. Find the sum of matrices

```
#include<iostream>
using namespace std;
int main() {
   int n, m;
   cout << "Enter rows: ";
   cin >> n;
   cout << "Enter columns: ";
   cin >> m;
   int a[n][m]; int b[n][m]; int c[n][m];
         cout << "Enter elements of 1st matrix: ";
   for(int i = 0; i < n; i++) {
   for (int j = 0; j < m; j++) {
   cin >> a[i][j];}}
   cout << "1st Matrix is: " << endl;
   for(int i = 0; i < n; i++) {
   for (int j = 0; j < m; j++) {
   cout << a[i][j] << " ";}
   cout << "\n";}
   cout << "Enter elements of 2nd matrix: ";
   for(int i = 0; i < n; i++) {
   for (int j = 0; j < m; j++) {
   cin >> b[i][j];}}
   cout << "2nd Matrix is: " << endl;
```

Output

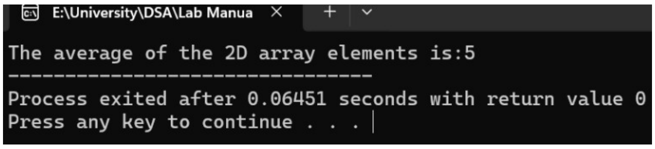
```
E:\University\DSA\Lab Manua
Enter rows: 2
Enter columns: 2
Enter elements of 1st matrix: 1 2 3 4
1st Matrix is:
1 2
3 4
Enter elements of 2nd matrix: 1 2 3 4
2nd Matrix is:
1 2
3 4
Total of Matrices is:
2 4
6 8
```

```
    for(int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
    cout << b[i][j] << " ";}
    cout << "\n";}
    cout << "Total of Matrices is: " << endl;
    for(int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
    c[i][j] = a[i][j] + b[i][j];
    cout << c[i][j] << " ";}
    cout << "\n";}
    return 0;
}
```

5. Find Average in a 2d array

```
#include <iostream>
using namespace std;
int main() {
    int array[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    int rows = 3, cols = 3;
    int sum = 0;
    int count = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            sum += array[i][j];
            count++;
        }
    }
    float average = (float)sum / count;
    cout<<"The average of the 2D array elements is:"<<average;
    return 0;
}
```

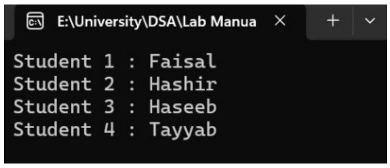
Output



Lab 4: Vectors

- **Definition:** A vector is a dynamic array provided by the Standard Template Library (STL) in C++. It stores a collection of elements of the same type in a contiguous memory block, and its size can grow or shrink dynamically as needed
- **Syntax:** vector<data-type> variable_name;
- **Key Characteristics of a Vector:**
    1. **Dynamic Size**: Vectors can grow or shrink as needed.
    2. **Ordered**: Elements are stored in a defined order.
    3. **Indexing**: Elements can be accessed using zero-based indices.
- **Example Programs**

Output

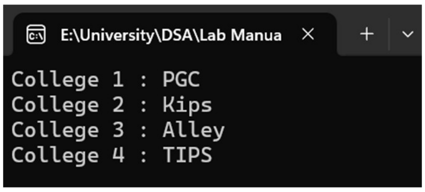


1. Create a student vector list

```
#include<iostream>
#include<vector>
using namespace std;
int main(){
vector <string>students={"Faisal","Hashir","Haseeb","Tayyab","Hanzla"};
for(int i=0;i<4;i++){
			cout<<"Student "<<i+1<<" : "<<students[i]<<endl;
		}
}
```

2. Create a college vector list

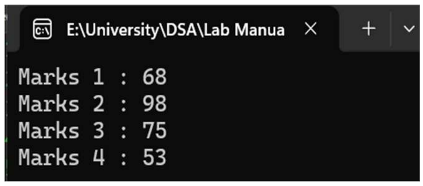
Output




```
#include<iostream>
#include<vector>
using namespace std;
int main(){
		vector <string>colleges={"PGC","Kips","Alley","TIPS"};
		for(int i=0;i<4;i++){
			cout<<"College "<<i+1<<" : "<<colleges[i]<<endl;
		}
}
```

3. Create a marks vector list

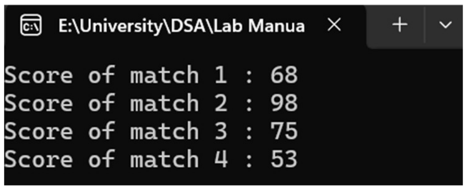
Output




```
#include<iostream>
#include<vector>
using namespace std;
int main(){
        vector <int>marks={68,98,75,53,62};
        for(int i=0;i<4;i++){
                cout<<"Marks "<<i+1<<" : "<<marks[i]<<endl;
        }
}
```

4. Create a match scores vector list

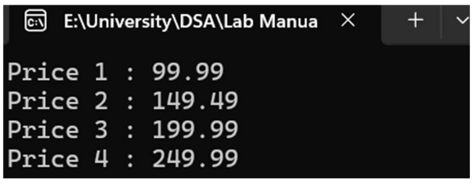
Output




```
#include<iostream>
#include<vector>
using namespace std;
int main(){
        vector <int>scores={68,98,75,53,62};
        for(int i=0;i<4;i++){
                cout<<"Score of match "<<i+1<<" : "<<scores[i]<<endl;
        }
}
```

5. Create a price vector list

Output




```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<double> prices = {99.99, 149.49, 199.99, 249.99, 299.99};
    for (int i = 0; i < 4; i++) {
        cout << "Price " << i + 1 << " : " << prices[i] << endl;
    }
    return 0;
}
```

Lab 5: List

- **Definition:** A list is a data structure that stores a dynamic collection of ordered elements, where each element can be of the same or different types. Lists allow easy insertion, deletion, and access of elements based on their position.
- **Syntax:** list<data-type> variable_name;
- **Key Characteristics of a List:**
  1. **Dynamic Size**: List can grow or shrink as needed.
  2. **Ordered**: Elements are stored in a defined order.
  3. **Indexing**: Elements can be accessed using zero-based indices.
- **Example Program**

1. **Write the basic implementation of a list**

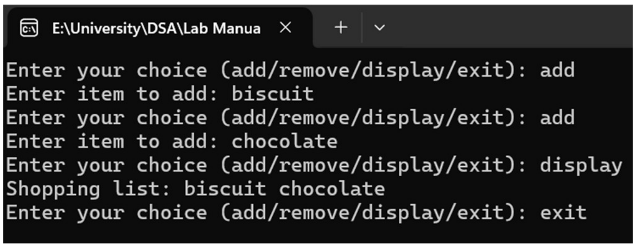
```
#include <iostream>
#include <list>
using namespace std;
int main() {
   list<int> numbers;
   numbers.push_back(10);
   numbers.push_front(20);
   numbers.push_back(30);
         cout << "List elements: ";
   for (int num : numbers) {
      cout << num << " ";
   }
   cout << std::endl;
   numbers.pop_front();
   cout << "After removing front element: ";
   for (int num : numbers) {
      cout << num << " ";
   }
   cout << std::endl;

   return 0;
}
```

Output

```
E:\University\DSA\Lab Manua
List elements: 20 10 30
After removing front element: 10 30
```

2. Create a shopping cart list

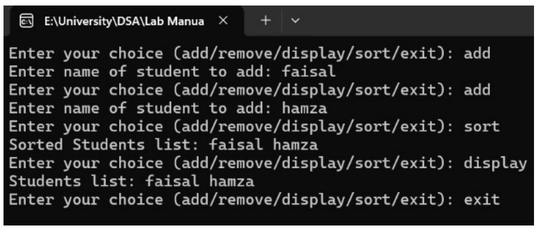
Output




```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<string> shopping;
    string choice,item;
    while(true) {
        cout << "Enter your choice (add/remove/display/exit): ";
        cin >> choice;
        if(choice == "add") {
            cout << "Enter item to add: ";
            cin >> item;
            shopping.push_back(item);
        } else if(choice == "remove") {
            string item;
            cout << "Enter item to remove: ";
            cin >> item;
            shopping.remove(item);
        } else if(choice == "display") {
            cout << "Shopping list: ";
            for(auto &item : shopping) cout << item << " ";
            cout << endl;
        } else if(choice == "exit") {
            break;
        }
    }
    return 0;
}
```

3. Create a student List

Output

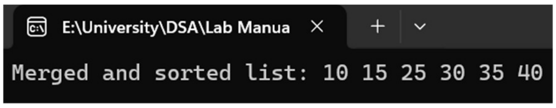



```
#include <iostream>
#include <list>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    list<string> student;
    string choice, name;
    while (true) {
        cout << "Enter your choice (add/remove/display/sort/exit): ";
```

```
    cin >> choice;
    if (choice == "add") {
      cout << "Enter name of student to add: ";
      cin >> name;
      student.push_back(name);
    } else if (choice == "remove") {
      cout << "Enter name of student to remove: ";
      cin >> name;
      student.remove(name);}
                    else if (choice == "display") {
      cout << "Students list: ";
      for (auto &name : student) cout << name << " ";
      cout << endl;
    } else if (choice == "sort") {
      vector<string> temp(student.begin(), student.end());
      sort(temp.begin(), temp.end());
      cout << "Sorted Students list: ";
      for (auto &name : temp) cout << name << " ";
      cout << endl;
    } else if (choice == "exit") {
      break;
    } else {
      cout << "Invalid choice. Please try again." << endl;}}
  return 0;
}
```

4. **Merge two lists**

Output

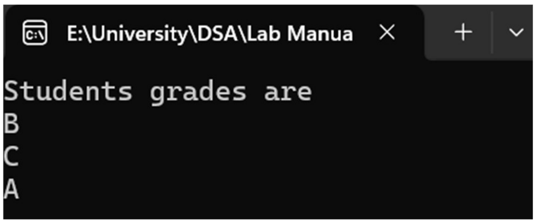



```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;
int main() {
  list<int> list1 = {40, 10, 30};
  list<int> list2 = {25, 15, 35};
  list1.sort();
  list2.sort();
  list1.merge(list2);
  cout << "Merged and sorted list: ";
  for (int num : list1) {
    cout << num << " ";}
  cout << std::endl;}
```

5. Create a grades List

```
#include<iostream>
#include<list>
using namespace std;
int main(){
	list <float>grades={'A','B','C','A','E'};
	cout<<"Students grades are"<<endl;
	grades.pop_front();
	grades.pop_back();
	for(char grades: grades){
		cout<<grades<<endl;
	}
}
```

Output



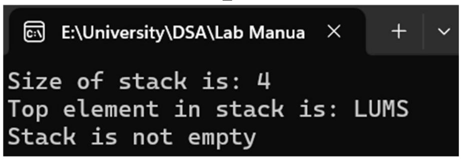
Lab 6: Stack

- **Definition:** A stack is a data structure that stores a collection of elements following the Last In, First Out (LIFO) principle. Elements can only be added or removed from the top of the stack. Stacks allow efficient insertion and deletion operations but restrict access to elements based on their position in the stack.
- **Syntax:** stack<data-type> variable_name;
- **Key Characteristics of a stack:**
  1. **LIFO (Last In, First Out)**: Elements follow the last-in, first-out principle, where the last element added is the first to be removed.
  2. **Restricted Access**: Elements can only be added or removed from the top of the stack.
  3. **No Indexing**: Elements are accessed based on their position in the stack, not via indexing
- **Example Programs**

1. **Write the basic Implementation of stack**

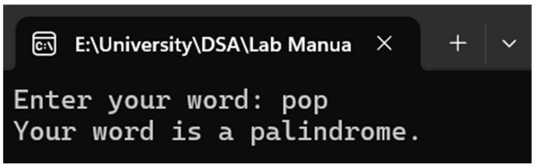
```
#include<iostream>
#include<string>
#include<stack>
using namespace std;
int main(){
        stack<string>universities;
        universities.push("TUF");
        universities.push("FAST");
        universities.push("NUST");
        universities.push("LUMS");
        universities.push("NTU");
        universities.push("Riphah");
        universities.push("BNU");
        universities.pop();
        universities.pop();
        universities.pop();
        cout<<"Size of stack is: "<<universities.size()<<"\n";
        cout<<"Top element in stack is: "<<universities.top();
        if(universities.empty()==true){
                cout<<"\nStack is empty";
         }
        else
                cout<<"\nStack is not empty";}
```

Output



2. Find if the stack is palindrome

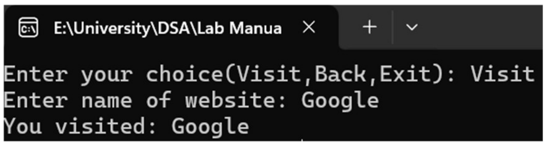
Output




```
#include<iostream>
#include<stack>
using namespace std;
int main(){
        string word;
        cout<<"Enter your word: ";
        cin>>word;
        stack<char> x;
        for(char c : word) x.push(c);
    string rev;
    while(!x.empty()) {
        rev += x.top();
        x.pop();
    }
    if(word == rev)
        cout <<"Your word is a palindrome."<<endl;
    else
        cout<<"Your word is not a palindrome."<<endl;
}
```

3. Create a website URL visiting code

Output

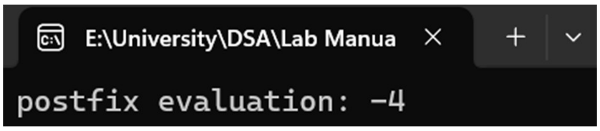



```
#include<iostream>
#include<stack>
using namespace std;
int main(){
        stack<string> browser;
        string choice;
        cout<<"Enter your choice(Visit,Back,Exit): ";
        cin>>choice;
        while (true) {
        if(choice=="Visit"){
                string name;
                cout<<"Enter name of website: ";
                cin>>name;
                browser.push(name);
                 cout << "You visited: " << name << endl;
        }
        else if (choice == "Back") {
        if (!browser.empty()) {
            cout << "Going back from: " << browser.top() << endl;
```

```
            browser.pop();
            if (!browser.empty()) {
                cout << "Current page: " << browser.top() << endl;
            } else {
                cout << "No more history. You're on a blank page.\n";
            }
        } else {
            cout << "No history to go back to.\n";
        }

    } else if (choice == "Exit") {
        break;

    } else {
        cout << "Invalid command. Please enter 'visit', 'back', or 'exit'.\n";
    }
}

return 0;
}
```

4. Find the Postfix Evaluation

Output



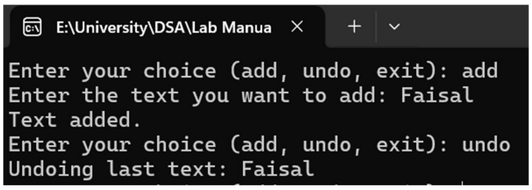

```
#include <iostream>
#include <stack>
using namespace std;
int evaluatePostfix(string exp)
{
 stack<int> st;
 for (int i = 0; i < exp.size(); ++i) {
  if (isdigit(exp[i]))
   st.push(exp[i] - '0');
  else {
   int val1 = st.top();
   st.pop();
   int val2 = st.top();
   st.pop();
   switch (exp[i]) {
   case '+':
    st.push(val2 + val1);
    break;
   case '-':
```

```
  st.push(val2 - val1);
  break;
 case '*':
  st.push(val2 * val1);
  break;
 case '/':
  st.push(val2 / val1);
  break;
 }
 }
}
return st.top();
}
    int main()
{
 string exp = "231*+9-";
 cout << "postfix evaluation: " << evaluatePostfix(exp);
 return 0;
}
```

5. **Create a undo text software**

```
#include <iostream>
#include <stack>
using namespace std;
int main() {
   stack<string> commandStack;
   string choice;

   while (true) {
      cout << "Enter your choice (add, undo, exit): ";
      cin >> choice;

      if (choice == "add") {
         string addText;
         cout << "Enter the text you want to add: ";
         cin.ignore();
         cin>>addText;
         commandStack.push(addText);
         cout << "Text added.\n";

      } else if (choice == "undo") {
```

Output



```
            if (!commandStack.empty()) {
                cout << "Undoing last text: " << commandStack.top() << endl;
                commandStack.pop();
            } else {
                cout << "No text to undo.\n";
            }

        } else if (choice == "exit") {
            break;

        } else {
            cout << "Invalid choice. Please enter 'add', 'undo', or 'exit'.\n";
        }
    }
}
```

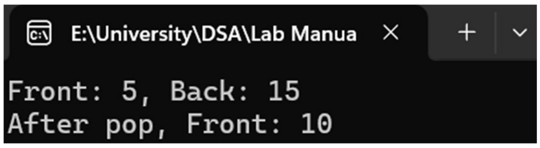
Lab 7: Queue

- **Definition:** A queue is a data structure that stores a collection of elements following the **First In, First Out (FIFO)** principle. Elements are added at the back enqueue) and removed from the front (dequeue). Queues allow efficient insertion and deletion operations while maintaining the order in which elements were added
- **Syntax:** queue<data-type> variable_name;
- **Key Characteristics of a Queue:**
    1. **FIFO (First In, First Out)**: Elements follow the first-in, first-out principle, where the first element added is the first to be removed.
    2. **Sequential Processing**: Ideal for scenarios requiring sequential order of processing.
    3. **No Indexing**: Elements are accessed in a linear fashion, not via indexing
- **Example Programs**

1. Write the basic Implementation of queue

```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<int> q;
    q.push(5);
    q.push(10);
    q.push(15);
    cout << "Front: " << q.front() << ", Back: " << q.back() << endl;
    q.pop();
    cout << "After pop, Front: " << q.front() << endl;
    return 0;
}
```

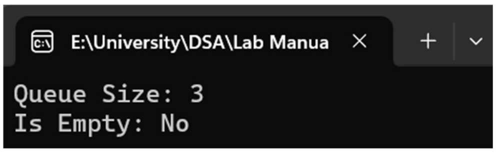
Output



2. Check the size and emptiness of queue

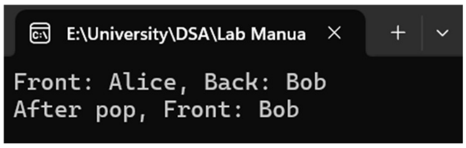
```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<int> q;
    q.push(1); q.push(2); q.push(3);
    cout << "Queue Size: " << q.size() << endl;
    cout << "Is Empty: " << (q.empty() ? "Yes" : "No") << endl;
    return 0;
}
```

Output



3. How queue works with strings

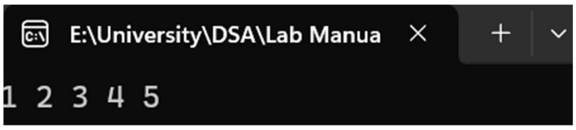
Output




```
#include <iostream>
#include <queue>
using namespace std;
int main() {
    queue<string> q;
    q.push("Alice");
    q.push("Bob");
    cout << "Front: " << q.front() << ", Back: " << q.back() << endl;
    q.pop();
    cout << "After pop, Front: " << q.front() << endl;
    return 0;
}
```

4. Iterating over queue

Output




```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;
int main() {
        queue<int> q; stack<int> s;
        for (int i = 1; i <= 5; ++i) q.push(i);
         while (!q.empty()) { s.push(q.front()); q.pop(); }
         while (!s.empty()) { cout << s.top() << " "; s.pop(); }
        return 0;
}
```

5. Merging two queues

Output


```
#include <iostream>
#include <queue>
using namespace std;
int main() {
         queue<int> q1, q2;
```

```
q1.push(1); q1.push(2);
q2.push(3); q2.push(4);
while (!q1.empty()) { cout << q1.front() << " "; q1.pop(); }
while (!q2.empty()) { cout << q2.front() << " "; q2.pop(); }
 return 0;
}
```

Lab 8: MCQ Quiz

- In this lab we did a MCQ quiz of the topics till mid term

1. **Which of the following data structures is a collection of elements of the same type, stored at contiguous memory locations?**

   - A) Stack

   - B) Queue

   - C) Array (Correct)

   - D) List

2. **In which data structure is the last element added the first to be removed?**

   - A) Queue

   - B) Stack (Correct)

   - C) List

   - D) Array

3. **What is the term for accessing elements in an array by their index?**

   - A) Random Access (Correct)

   - B) Sequential Access

   - C) Linear Access

   - D) Binary Access

4. **Which of the following allows dynamic resizing in C++?**

   - A) Array

   - B) Vector (Correct)

   - C) Stack

   - D) Queue

5. **Which operation removes the first element in a queue?**

- A) Push
- B) Pop
- C) Enqueue
- D) Dequeue **(**Correct**)**

6. **What is the default access order of elements in a stack?**

- A) First-In, First-Out (FIFO)
- B) Last-In, First-Out (LIFO) **(**Correct**)**
- C) Random
- D) Sequential

7. **Which data structure allows both insertion and deletion at both ends?**

- A) Array
- B) Stack
- C) List
- D) Deque **(**Correct**)**

8. **In an array of size `n`, what is the index of the last element?**

- A) `n`
- B) `n-1` **(**Correct**)**
- C) `n+1`
- D) `0`

9. What is the primary difference between an array and a vector in C++?

- A) Arrays can grow dynamically
- B) Vectors can grow dynamically (Correct)
- C) Vectors store elements in random order
- D) Arrays store elements in linked order

10. Which data structure would be most efficient for implementing an undo feature?

- A) Queue
- B) Stack (Correct)
- C) Array
- D) List

11. What is the primary operation for adding an element to a stack?

- A) Insert
- B) Add
- C) Push (Correct)
- D) Append

12. Which data structure follows the First-In, First-Out (FIFO) principle?

- A) Stack
- B) Queue (Correct)
- C) Array
- D) Vector

13. In a vector, what function is used to add elements at the end?

- A) insert()
- B) append()
- C) push_back() **(**Correct**)**
- D) add()

14. Which of the following is used to retrieve the number of elements in a vector?

- A) length()
- B) count()
- C) size() **(**Correct**)**
- D) capacity()

15. What type of data structure is suitable for organizing tasks in the order they should be executed?

- A) Stack
- B) Queue **(**Correct**)**
- C) Array
- D) List

16. Which data structure supports efficient access to any element by its index?

- A) Array **(**Correct**)**
- B) Stack
- C) Queue
- D) Linked List

17. What operation is used to add elements to the front of a list?

- A) push_back()
- B) insert()
- C) push_front() **(**Correct**)**
- D) append()

18. In a circular queue, if the front pointer is at the last position, where will it move after a dequeue operation?

- A) Remains at the same position
- B) Moves to the beginning **(**Correct**)**
- C) Moves to the end
- D) Moves randomly

19. Which of the following data structures is a sequence container with dynamic size, allowing elements to be accessed by their position?

- A) Queue
- B) Stack
- C) Vector **(**Correct**)**
- D) Linked List

20. What is the time complexity of accessing an element in an array by its index?

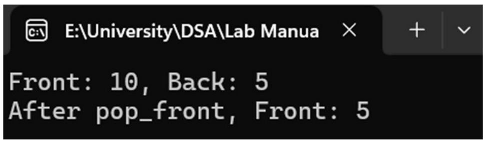
- A) O(n)
- B) O(log n)
- C) O(1) **(**Correct**)**
- D) O(n^2)

Lab 9: Deque

- **Definition:** A deque is a data structure that stores a collection of elements and allows insertion and deletion from both ends (front and back). Deques provide flexibility for accessing elements while maintaining an ordered sequence. They are ideal for scenarios requiring dynamic resizing and dual-end operations.
- **Syntax:** deque<data-type> variable_name;
- **Key Characteristics of a Queue:**
    1. **Flexible Access**: Elements can be added or removed from both ends (front and back).
    2. **Dynamic Size**: Deques can grow or shrink as needed.
- **Example Programs**

1. Write basic implementation of deque

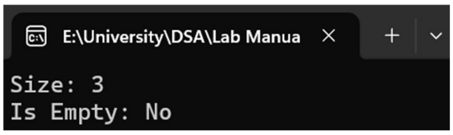
Output




```
#include <iostream>

#include <deque>

using namespace std;

int main() {

    deque<int> dq;

    dq.push_back(5);

    dq.push_front(10);

    cout << "Front: " << dq.front() << ", Back: " << dq.back() << endl;

    dq.pop_front();

    cout << "After pop_front, Front: " << dq.front() << endl;

    return 0;

}
```

2. Size and Empty Check of deque

Output



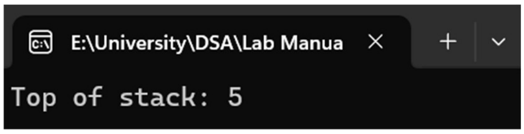

```
#include <iostream>

#include <deque>

using namespace std;

int main() {
```

```
    deque<int> dq = {1, 2, 3};
    cout << "Size: " << dq.size() << endl;
    cout << "Is Empty: " << (dq.empty() ? "Yes" : "No") << endl;
    return 0;
}
```

3. Deque used as Stack

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<int> dq;
    dq.push_back(5);
    dq.push_back(10);
    dq.pop_back();
    cout << "Top of stack: " << dq.back() << endl;
    return 0;
}
```

Output



4. Iterating our deque

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<int> dq = {1, 2, 3, 4};
    for (int x : dq) cout << x << " ";
    return 0;
}
```

Output

5. Reversing our deque

```
#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main() {
    deque<int> dq = {1, 2, 3, 4};
    reverse(dq.begin(), dq.end());
    for (int x : dq) cout << x << " ";
    return 0;
}
```

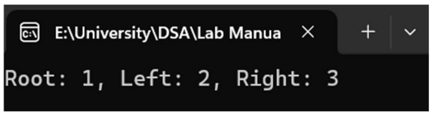
Output

Lab 10: Trees

- **Definition:** A tree is a hierarchical data structure that organizes elements in a parent-child relationship. It starts with a single root node and branches into sub-nodes (children), forming levels. Each node can have zero or more children, and there is no limit to the depth of the hierarchy. Trees are widely used for efficient searching, sorting, and hierarchical data representation
- **Key Characteristics of a Tree:**
  1. **Hierarchical Structure**: Organized in levels, with a root node and child nodes forming a hierarchy.
  2. **Parent-Child Relationship**: Each node (except the root) has one parent and may have multiple children.
  3. **Recursive Representation**: Trees are naturally represented and traversed using recursion (e.g., in-order, pre-order, post-order).
- **Example Programs**

1. Implement simple binary tree

Output




```
#include <iostream>

using namespace std;

struct Node {

    int data;

    Node *left, *right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}

};


int main() {

    Node* root = new Node(1);

    root->left = new Node(2);

    root->right = new Node(3);

    cout << "Root: " << root->data << ", Left: " << root->left->data << ", Right: " << root->right->data << endl;

    return 0;}
```

2. Find Preorder Traversal

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node *left, *right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};
void preorder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    preorder(root);
    return 0;
}
```

Output

3. Find In-order Traversal

```
#include <iostream>

using namespace std;


struct Node {

    int data;

    Node *left, *right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

void inorder(Node* root) {

    if (!root) return;

    inorder(root->left);

    cout << root->data << " ";

    inorder(root->right);

}

int main() {

    Node* root = new Node(1);

    root->left = new Node(2);

    root->right = new Node(3);

    inorder(root);

    return 0;

}
```

Output

4. Find Post-order Traversal

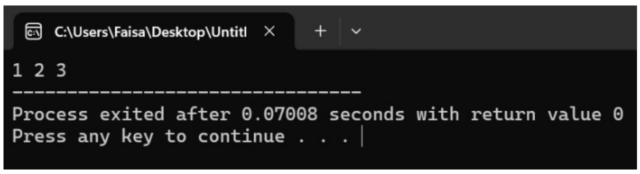
```
#include <iostream>

using namespace std;

struct Node {

    int data;
```

Output

```
    Node *left, *right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};
void postorder(Node* root) {
    if (!root) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    postorder(root);
    return 0;
}
```

5. Find Level Order Traversal

Output




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

struct Node {
    int data;
```

```
    Node *left, *right;
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

void levelOrder(Node* root) {
    if (!root) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        cout << curr->data << " ";
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
    }
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    levelOrder(root);
    return 0;
}
```

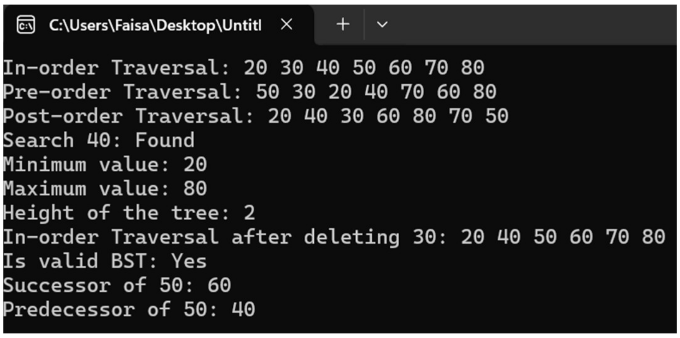
Lab 11: Binary Search Trees

- **Definition:** A Binary Search Tree (BST) is a type of binary tree in which each node follows a specific ordering property:
    1. **Left Subtree**: Contains nodes with keys less than the parent node's key.
    2. **Right Subtree**: Contains nodes with keys greater than the parent node's key. This property makes BSTs highly efficient for searching, insertion, and deletion operations.
- **Key Characteristics of a Tree:**
    1. **Ordered Structure**: Nodes are arranged in a way that enables efficient operations.
    2. **Recursive Representation**: BST operations like traversal and manipulation are naturally implemented using recursion.
    3. **No Duplicates**: In a standard BST, duplicate elements are not allowed.
    4. **Traversals**: Common traversal methods include in-order, pre-order, and post-order, where in-order traversal yields a sorted sequence of elements.
- **Example Programs**

1. Insertion in BST

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

Output



2. Deletion in BST

```
Node* findMin(Node* root) {
    while (root->left != nullptr) root = root->left;
    return root;
}
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 {
        if (root->left == nullptr) return root->right;
        if (root->right == nullptr) return root->left;
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

Output

```
In-order Traversal after deleting 30: 20 40 50 60 70 80
Is valid BST: Yes
Successor of 50: 60
Predecessor of 50: 40
```

3. Searching in BST

```
bool search(Node* root, int value) {
    if (root == nullptr) return false;
    if (root->data == value) return true;
    if (value < root->data)
        return search(root->left, value);
    return search(root->right, value);}
```

Output

```
Search 40: Found
Minimum value: 20
Maximum value: 80
Height of the tree: 2
In-order Traversal after deleting 30: 20 40 50 60 70 80
Is valid BST: Yes
Successor of 50: 60
Predecessor of 50: 40
```

4. Traversing in BST(in-order, pre-order, post-order)

```
void inOrder(Node* root) {
    if (root == nullptr) return;
    inOrder(root->left);
    std::cout << root->data << " ";
    inOrder(root->right);
}

void preOrder(Node* root) {
    if (root == nullptr) return;
    std::cout << root->data << " ";
    preOrder(root->left);
    preOrder(root->right);
}
void postOrder(Node* root) {
    if (root == nullptr) return;
    postOrder(root->left);
    postOrder(root->right);
    std::cout << root->data << " ";
}
```

Output

```
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
```

5. Minimum and Maximum

```
int findMinValue(Node* root) {
    while (root->left != nullptr) root = root->left;
    return root->data;
}

int findMaxValue(Node* root) {
    while (root->right != nullptr) root = root->right;
    return root->data;
}
```

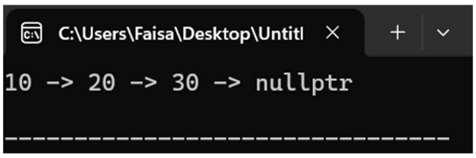
Output

```
Minimum value: 20
Maximum value: 80
```

Lab 12: Singly Link List

- **Definition:** A Linked List is a linear data structure where elements, called nodes, are connected using pointers. Each node contains two parts.
- **Singly link list:** A singly link list is the link where each node has a single pointer to the next node. It Traverses is one-directional and the last node's pointer is nullptr.
- **Example Programs**

Output

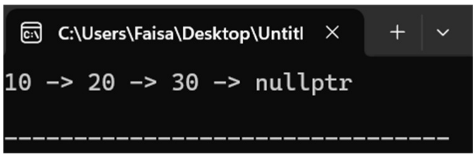


1. Node Structure

```
#include <iostream>

using namespace std;

struct Node {

    int data;

    Node* next;

    Node(int val) : data(val), next(nullptr) {}

};
```

2. Insertion at end

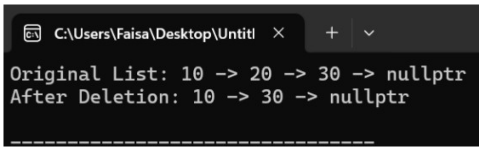
Output




```
void insertEnd(Node*& head, int val) {

    Node* newNode = new Node(val);

    if (head == nullptr) {

        head = newNode;

        return;

    }

    Node* temp = head;

    while (temp->next != nullptr) {

        temp = temp->next;

    }

    temp->next = newNode;

}
```

3. Deletion by value

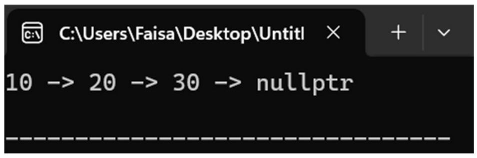
Output




```
void deleteNode(Node*& head, int val) {
    if (head == nullptr) return;

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

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

    if (temp->next == nullptr) return;

    Node* toDelete = temp->next;
    temp->next = toDelete->next;
    delete toDelete;
}
```

4. Insertion at begin

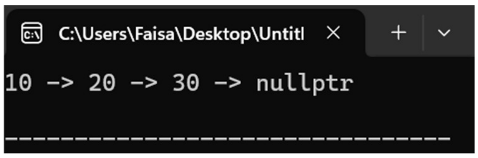
Output




```
void insertAtBegin(int data) {
        node *n = new node();
        n->data = data;
        n->link = head;
        head = n;
    }
```

5. Display the list

Output




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

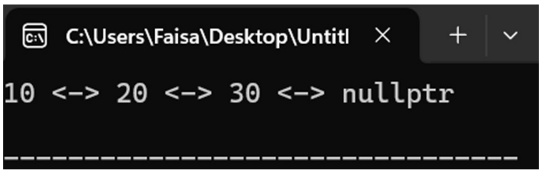
Lab 13: Doubly Link List

- **Characteristics**
  1. Each node has pointers to both the previous and next nodes.
  2. Traversal is bidirectional.
  3. The head's prev pointer and the last node's next pointer are nullptr.
- **Example Programs**

1. Node Structure

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

    DoublyNode(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

Output



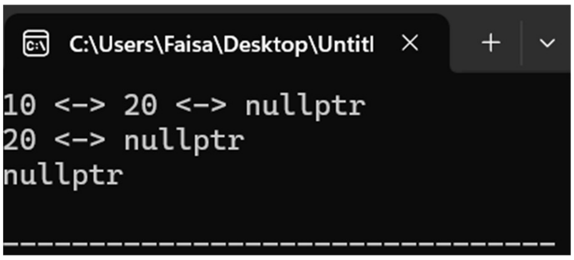
2. Insertion at end

```
void insertEnd(DoublyNode*& head, int val) {
    DoublyNode* newNode = new DoublyNode(val);
    if (head == nullptr) {
        head = newNode;
        return;
    }
    DoublyNode* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;}
```

Output

3. Deletion at value

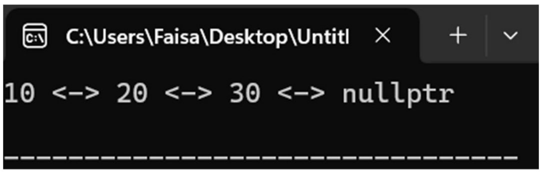
Output




```
void deleteNode(DoublyNode*& head, int val) {
    if (head == nullptr) return;
    if (head->data == val) {
        DoublyNode* temp = head;
        head = head->next;
        if (head != nullptr) head->prev = nullptr;
        delete temp;
        return;
    }
    DoublyNode* temp = head;
    while (temp != nullptr && temp->data != val) {
        temp = temp->next;
    }
    if (temp == nullptr) return;
    if (temp->next != nullptr) temp->next->prev = temp->prev;
    if (temp->prev != nullptr) temp->prev->next = temp->next;
    delete temp;
}
```

4. Display the list

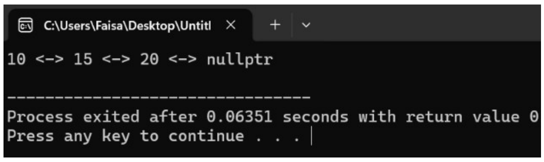
Output




```
void display(DoublyNode* head) {
DoublyNode* temp = head;
while (temp != nullptr) {
  cout << temp->data << " <-> ";
  temp = temp->next;
}
cout << "nullptr" << endl;}
```

5. Insert at postion

Output




```
void insertatposition(int position,int data){
        node *n = new node();
        n->data=data;
        if(position==1){
            n->next=head;
            if(head!=NULL){
                head->prev=n;}
            head = n;
            if(tail==NULL){
                tail=n;}}
        else{
            current = head;
            int count = 1;
            while(current!=nullptr && count<position-1){
                current = current->next;
                count++;}
            if(current==NULL){
                cout<<"position out of bounds";
                delete n;
                return;}
            n->next=current->next;
            n->prev=current;
            if (current->next != nullptr) {
    current->next->prev = n;} else { // If inserting at the tail
    tail = n;}
    current->next = n;}}
```

Lab 14: Circular Link List

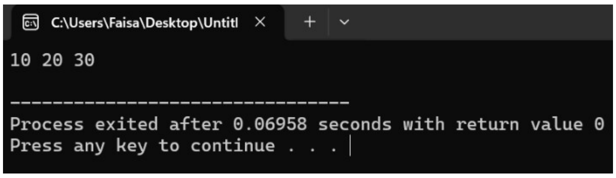
- **Characteristics**
  - The last node points to the first node.
  - Can be singly or doubly linked.
  - Enables circular traversal.

- **Example Programs**

1. Node Structure

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

    CNode(int val) : data(val), next(nullptr) {}
};
```

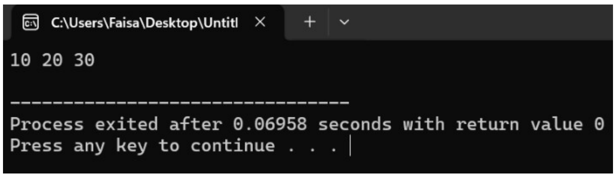
Output



2. Insertion at end

```
void insertEnd(CNode*& head, int val) {
    CNode* newNode = new CNode(val);
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    CNode* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
```

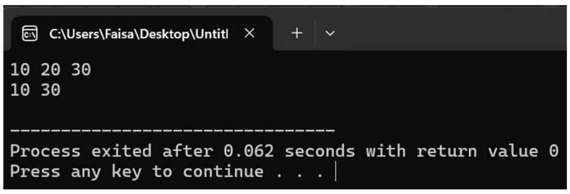
Output



3. Deletion at value

```
void deleteNode(CNode*& head, int val) {

    if (head == nullptr) return;


    if (head->data == val && head->next == head) {

        delete head;
```

Output

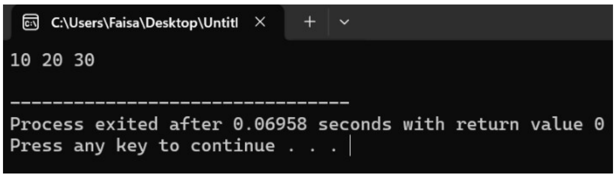


```
        head = nullptr;
        return;
    }

    CNode* temp = head;
    CNode* prev = nullptr;
    do {
        if (temp->data == val) break;
        prev = temp;
        temp = temp->next;
    } while (temp != head);

    if (temp == head && temp->data != val) return;

    if (temp == head) {
        prev = head;
        while (prev->next != head) prev = prev->next;
        head = head->next;
        prev->next = head;
    } else {
        prev->next = temp->next;
    }
    delete temp;
}   DoublyNode* temp = head;
    while (temp != nullptr && temp->data != val) {
        temp = temp->next;
    }
```

```
    if (temp == nullptr) return;
    if (temp->next != nullptr) temp->next->prev = temp->prev;
    if (temp->prev != nullptr) temp->prev->next = temp->next;
    delete temp;
}
```

4. **Display the list**

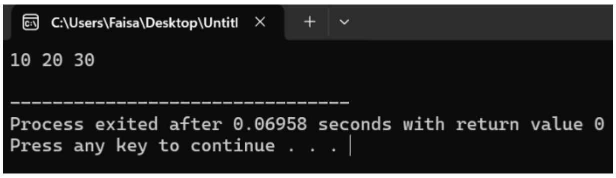
Output




```
void display(CNode* head) {
    if (head == nullptr) return;
    CNode* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(head)" << endl;
}
```

5. **Insert at position**

Output




```
void insertAtPosition(int data, int position) {
        node *n = new node();
        n->data = data;

        if (position == 0) { // Insert at the beginning
            n->link = head;
            head = n;
            if (tail == nullptr) {
                tail = n;
            }
        } else {
            current = head;
            for (int i = 0; i < position - 1 && current != nullptr; ++i) {
                current = current->link;
            }
```

```
        if (current != nullptr) {
            n->link = current->link;
            current->link = n;
            if (n->link == nullptr) {
                tail = n;
            }
        } else {
            cout << "Position out of bounds" << endl;
        }
    }
}
```