Introduction to Data Structures

A **Data Structure** in C++ is a method to store and organize data so that operations like searching, insertion, deletion, and updating can be performed efficiently.
C++ provides built-in data structures (arrays, structures) and supports advanced data structures using pointers, classes, and the STL (Standard Template Library).

Classification of Data Structures in C++

Data structures in C++ are mainly classified into **two categories**:

1. Linear Data Structures (in C++)

A **linear data structure** stores data sequentially.
Each element points to the next element in a straight line.

Key Features

- Elements are arranged one after another.
- Traversal is simple (start → end).
- Implemented using **arrays, pointers, classes, STL containers**.

Examples of Linear Data Structures in C++

1. Array

- Fixed size.
- Stores elements of the same data type.
- Memory is continuous.
- Declared like:
- `int arr[5] = {1, 2, 3, 4, 5};`

2. Linked List

- Dynamic in size.
- Nodes are connected using pointers.
- A node looks like:

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

- Types: **Singly**, **Doubly**, **Circular** Linked Lists.

3. Stack (LIFO – Last In First Out)

- Implemented using **arrays**, **linked lists**, or **STL stack**.

```
#include <stack>
stack<int> s;
```

4. Queue (FIFO – First In First Out)

- Implemented using array, linked lists, or **STL queue**:
- `#include <queue>`
- `queue<int> q;`

5. Deque (Double-Ended Queue)

- Insert & delete from both ends.
- `#include <deque>`
- `deque<int> d;`

2. Non-Linear Data Structures (in C++)

A **non-linear data structure** does not store data sequentially.
Elements are arranged hierarchically or in connections like networks.

Key Features

- Data forms hierarchies or interconnections.
- Complex traversal (DFS, BFS).
- Implemented using **pointers**, **classes**, and **STL containers**.

Examples of Non-Linear Data Structures in C++

1. Tree

- Hierarchical structure of nodes.
- Each node may have multiple children.
- Common types:
  - **Binary Tree**
  - **Binary Search Tree (BST)**
  - **AVL Tree**
  - **Heap**
- Basic node structure:
- `struct Node {`
- `    int data;`
- `    Node* left;`
- `    Node* right;`
- `};`

2. Graph

- Collection of nodes (vertices) connected by edges.
- Could be:
  - Directed / Undirected
  - Weighted / Unweighted
- Represented in C++ using:
  - Adjacency List
  - `vector<int> adj[100];`

- Adjacency Matrix
- `int graph[100][100];`

Time and Space Complexity

When we write a C++ program, the **efficiency** of an algorithm depends on:

1. **Time Complexity** → How much time an algorithm takes
2. **Space Complexity** → How much memory an algorithm uses

We do not measure time using seconds because it varies from computer to computer.
So, we measure it using **number of operations** → using **Asymptotic Notations**.

Asymptotic Notations

Asymptotic Notations describe the **growth rate** of an algorithm when input size **n** becomes very large.

Main three notations:

1. **Big O (O)** → Worst Case
2. **Big Omega (Ω)** → Best Case
3. **Big Theta (Θ)** → Average/Tight Bound

Big O Notation (O) – Worst Case

Big O represents the **maximum time** an algorithm can take.
It gives an **upper bound** on time complexity.

Why Needed?

- To know how slow the algorithm can get.
- Helps in guaranteeing correctness for all inputs.

Example in C++

```
// Linear search
int search(int arr[], int n, int key) {
    for(int i = 0; i < n; i++) {
        if(arr[i] == key)
            return i;
    }
    return -1;
}
```

Worst Case

Key is not present → loop runs **n times**
Time Complexity = O(n)

Big Omega Notation (Ω) – Best Case

Big Omega gives the **minimum time** an algorithm will take.
It is the **lower bound**.

Example

In the same linear search:

Best Case

Key is present at index 0 → only 1 comparison
Time Complexity = $\Omega(1)$

Big Theta Notation (Θ) – Average / Tight Bound

Theta gives the **exact (tight) time complexity**, meaning it covers both:

- upper bound (like Big O)
- lower bound (like Omega)

When best case and worst case behaviours are similar, we use Θ.

Example

If we consider average-case behaviour of linear search:

On average, key will be found at middle position → $n/2$ comparisons
Time Complexity = $\Theta(n)$

Summary Table

Notation	Meaning	Represents	Example (Linear Search)
Big O (O)	Upper bound	Worst case	$O(n)$
Big Omega (Ω)	Lower bound	Best case	$\Omega(1)$
Big Theta (Θ)	Tight bound	Average case	$\Theta(n)$

Common Time Complexities

Complexity	Name	Example
$O(1)$	Constant	Accessing array element: `arr[i]`
$O(\log n)$	Logarithmic	Binary Search
$O(n)$	Linear	Traversing array/loop
$O(n \log n)$	Log-linear	Merge Sort, Quick Sort(avg)
$O(n^2)$	Quadratic	Nested loops (Bubble Sort)
$O(2^n)$	Exponential	Subset generation

Complexity	Name	Example
O(n!)	Factorial	Permutations

Space Complexity (C++ Example)

Space used =

- input variables
- extra variables
- recursion stack
- dynamic memory

Example

```
int sum(int arr[], int n) {
    int s = 0;   // takes constant space
    for(int i = 0; i < n; i++) {
        s += arr[i];
    }
    return s;
}
```

⇒ No extra space based on input size
Space Complexity = O(1)

Array in C++

An **array** stores multiple values of the **same data type** in continuous memory.

Example

```
int marks[5] = {90, 80, 70, 85, 88};
```

Key Points

- Fixed size
- Same data type only
- Located in contiguous memory
- Easy to iterate

Array Class With All Methods

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

class Array {
private:
    int *arr;
    int capacity;
    int length;
```

```
public:
    // Constructor
    Array(int size) {
        capacity = size;
        arr = new int[capacity];
        length = 0;
    }

    // Insert at position
    void insert(int pos, int value) {
        if (length == capacity) {
            cout << "Array is full!" << endl;
            return;
        }
        if (pos < 0 || pos > length) {
            cout << "Invalid position!" << endl;
            return;
        }

        for (int i = length; i > pos; i--) {
            arr[i] = arr[i - 1];
        }
        arr[pos] = value;
        length++;
    }

    // Remove element
    void remove(int pos) {
        if (pos < 0 || pos >= length) {
            cout << "Invalid position!" << endl;
            return;
        }

        for (int i = pos; i < length - 1; i++) {
            arr[i] = arr[i + 1];
        }
        length--;
    }

    // Search value
    int search(int value) {
        for (int i = 0; i < length; i++) {
            if (arr[i] == value)
                return i;
        }
        return -1;
    }

    // Get element at position
    int get(int pos) {
        if (pos < 0 || pos >= length) {
            cout << "Invalid index!" << endl;
            return -1;
        }
        return arr[pos];
    }

    // Update element
    void set(int pos, int value) {
        if (pos < 0 || pos >= length) {
            cout << "Invalid index!" << endl;
```

```
            return;
        }
        arr[pos] = value;
    }

    // Return size
    int size() {
        return length;
    }

    // Print array
    void print() {
        cout << "Array: ";
        for (int i = 0; i < length; i++)
            cout << arr[i] << " ";
        cout << endl;
    }

    // Destructor
    ~Array() {
        delete[] arr;
    }
};

int main() {
    Array a(10);

    a.insert(0, 10);
    a.insert(1, 20);
    a.insert(2, 30);
    a.print();

    a.remove(1);
    a.print();

    cout << "Search 30: " << a.search(30) << endl;

    a.set(1, 99);
    a.print();

    cout << "Get index 1: " << a.get(1) << endl;

    cout << "Size: " << a.size() << endl;
}
```

What is a Linked List?

A **Linked List** is a **linear data structure** where elements (nodes) are stored **non-contiguously** in memory.
Each node contains:

✓ **Data**
✓ **Pointer to next node**

Why Linked List? (Advantages)

- Dynamic size (grows/shrinks at runtime)

- Efficient insertion/deletion (O(1)) when pointer known
- No memory wastage

3. Disadvantages

- No random access (unlike arrays)
- More memory (because of pointers)
- Slower traversal due to scattered memory

4. Types of Linked Lists

- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**
- **Circular Doubly Linked List**

Most Used Linked List Operations

Method	Description
insertAtBeginning(x)	Insert node at start
insertAtEnd(x)	Insert node at end
insertAtPosition(pos,x)	Insert at specific index
deleteAtBeginning()	Delete first node
deleteAtEnd()	Delete last node
deleteAtPosition(pos)	Delete at index
search(x)	Find element
display()	Print all nodes

Linked List Node Structure

```
data → value
```

Linked List with All Important Methods

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

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

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

class LinkedList {
private:
```

```
    Node* head;

public:
    // Constructor
    LinkedList() {
        head = nullptr;
    }

    // Insert at beginning
    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Insert at end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
            return;
        }

        Node* temp = head;
        while (temp->next != nullptr)
            temp = temp->next;

        temp->next = newNode;
    }

    // Insert at position
    void insertAtPosition(int pos, int value) {
        if (pos == 0) {
            insertAtBeginning(value);
            return;
        }

        Node* newNode = new Node(value);
        Node* temp = head;

        for (int i = 1; i < pos && temp != nullptr; i++)
            temp = temp->next;

        if (temp == nullptr) {
            cout << "Invalid position!" << endl;
            return;
        }

        newNode->next = temp->next;
        temp->next = newNode;
    }

    // Delete at beginning
    void deleteAtBeginning() {
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }
        Node* temp = head;
        head = head->next;
        delete temp;
```

```
}

// Delete at end
void deleteAtEnd() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

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

    Node* temp = head;
    while (temp->next->next != nullptr)
        temp = temp->next;

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

// Delete at position
void deleteAtPosition(int pos) {
    if (pos == 0) {
        deleteAtBeginning();
        return;
    }

    Node* temp = head;

    for (int i = 1; i < pos && temp != nullptr; i++)
        temp = temp->next;

    if (temp == nullptr || temp->next == nullptr) {
        cout << "Invalid position!" << endl;
        return;
    }

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

// Search element
int search(int key) {
    Node* temp = head;
    int index = 0;

    while (temp != nullptr) {
        if (temp->data == key)
            return index;
        temp = temp->next;
        index++;
    }
    return -1;
}

// Display list
void display() {
```

```
        if (head == nullptr) {
            cout << "List is empty!" << endl;
            return;
        }

        Node* temp = head;
        cout << "Linked List: ";
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }
};

int main() {
    LinkedList list;

    list.insertAtBeginning(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.insertAtPosition(1, 15);

    list.display();

    cout << "Search 20: Index = " << list.search(20) << endl;

    list.deleteAtBeginning();
    list.deleteAtEnd();
    list.display();

    return 0;
}
```

Time Complexity

Operation	Time
Insert at beginning	O(1)
Insert at end	O(n)
Delete at beginning	O(1)
Delete at end	O(n)
Search	O(n)
Traverse	O(n)

Important Interview Questions

1. Reverse a linked list
2. Detect a cycle (Floyd's algorithm)
3. Middle element of a linked list
4. Merge two sorted lists
5. Delete loop in linked list

Doubly Linked List

A **Doubly Linked List (DLL)** is a linked list where each node has:

- **data**
- **next pointer** → points to next node
- **prev pointer** → points to previous node

This allows **forward and backward traversal**.

Advantages

- Bidirectional traversal (forward + backward)
- Easy deletion with direct access to a node
- Insertions/deletions are easier than singly linked list

3. Disadvantages

- Extra memory for `prev` pointer
- More complex implementation

Most Used Operations

Operation	Description
insertAtBeginning(x)	Insert at start
insertAtEnd(x)	Insert at end
insertAtPosition(pos, x)	Insert at index
deleteAtBeginning()	Delete first node
deleteAtEnd()	Delete last node
deleteAtPosition(pos)	Delete at index
search(x)	Find value
displayForward()	Print from head to tail
displayBackward()	Print from tail to head

Doubly Linked List All important methods included

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

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

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

```
class DoublyLinkedList {
private:
    Node* head;

public:
    DoublyLinkedList() {
        head = nullptr;
    }

    // Insert at beginning
    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);

        newNode->next = head;
        if (head != nullptr)
            head->prev = newNode;

        head = newNode;
    }

    // Insert at end
    void insertAtEnd(int value) {
        Node* newNode = new Node(value);

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

        Node* temp = head;
        while (temp->next != nullptr)
            temp = temp->next;

        temp->next = newNode;
        newNode->prev = temp;
    }

    // Insert at position
    void insertAtPosition(int pos, int value) {
        if (pos == 0) {
            insertAtBeginning(value);
            return;
        }

        Node* temp = head;
        for (int i = 1; i < pos && temp != nullptr; i++)
            temp = temp->next;

        if (temp == nullptr) {
            cout << "Invalid position!" << endl;
            return;
        }

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

        if (temp->next != nullptr)
            temp->next->prev = newNode;
```

```
    temp->next = newNode;
}

// Delete at beginning
void deleteAtBeginning() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

    Node* temp = head;
    head = head->next;

    if (head != nullptr)
        head->prev = nullptr;

    delete temp;
}

// Delete at end
void deleteAtEnd() {
    if (head == nullptr) {
        cout << "List is empty!" << endl;
        return;
    }

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

    Node* temp = head;
    while (temp->next != nullptr)
        temp = temp->next;

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

// Delete at position
void deleteAtPosition(int pos) {
    if (pos == 0) {
        deleteAtBeginning();
        return;
    }

    Node* temp = head;
    for (int i = 1; i < pos && temp != nullptr; i++)
        temp = temp->next;

    if (temp == nullptr) {
        cout << "Invalid position!" << endl;
        return;
    }

    if (temp->next != nullptr)
        temp->next->prev = temp->prev;

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

```
        delete temp;
    }

    // Search element
    int search(int key) {
        Node* temp = head;
        int index = 0;

        while (temp != nullptr) {
            if (temp->data == key)
                return index;
            temp = temp->next;
            index++;
        }
        return -1;
    }

    // Display forward
    void displayForward() {
        Node* temp = head;
        cout << "Forward: ";
        while (temp != nullptr) {
            cout << temp->data << " <-> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }

    // Display backward
    void displayBackward() {
        if (head == nullptr) {
            cout << "Backward: NULL" << endl;
            return;
        }

        Node* temp = head;
        while (temp->next != nullptr)
            temp = temp->next;

        cout << "Backward: ";
        while (temp != nullptr) {
            cout << temp->data << " <-> ";
            temp = temp->prev;
        }
        cout << "NULL" << endl;
    }
};

int main() {
    DoublyLinkedList dll;

    dll.insertAtBeginning(10);
    dll.insertAtEnd(20);
    dll.insertAtEnd(30);
    dll.insertAtPosition(1, 15);

    dll.displayForward();
    dll.displayBackward();

    cout << "Search 20 at index: " << dll.search(20) << endl;
```

```
    dll.deleteAtBeginning();
    dll.deleteAtEnd();
    dll.displayForward();

    return 0;
}
```

Time Complexity

Operation	Time
Insert at beginning	O(1)
Insert at end	O(n)
Delete at beginning	O(1)
Delete at end	O(n)
Insert/Delete at position	O(n)
Search	O(n)

Important Interview Questions

1. Reverse a doubly linked list
2. Insert in sorted doubly linked list
3. Convert DLL to circular DLL
4. Delete duplicates in a doubly linked list
5. Implement browser navigation using DLL