- # Linked List

A linked list is a linear data structure where elements, known as nodes, are not stored in contiguous memory locations. Each node contains two main components: data and a reference (or pointer) to the next node in the sequence. Linked lists are dynamic in nature, allowing efficient insertion and deletion of elements.

Types of Linked Lists

1. **Singly Linked List**: Each node points to the next node in the sequence.
2. **Doubly Linked List**: Each node contains references to both the next and the previous nodes.
3. **Circular Linked List**: The last node points back to the first node, forming a circle.

Basic Operations and Time Complexities

Singly Linked List

1. **Insertion**:
   - **At the beginning**: (O(1))
   - **At the end**: (O(n)) (requires traversal to the end)
   - **After a specific node**: (O(1)) (if the node is given)
2. **Deletion**:
   - **From the beginning**: (O(1))
   - **From the end**: (O(n)) (requires traversal to the node before the last one)
   - **A specific node**: (O(n)) (if the node is not given, needs traversal to find it)
3. **Traversal**: (O(n))
4. **Searching**: (O(n))

Doubly Linked List

1. **Insertion**:
   - **At the beginning**: (O(1))
   - **At the end**: (O(1)) (if there is a tail pointer, otherwise (O(n)))
   - **After a specific node**: (O(1)) (if the node is given)
2. **Deletion**:
   - **From the beginning**: (O(1))
   - **From the end**: (O(1)) (if there is a tail pointer, otherwise (O(n)))
   - **A specific node**: (O(n)) (if the node is not given, needs traversal to find it)
3. **Traversal**: (O(n))
4. **Searching**: (O(n))

Circular Linked List

1. **Insertion**:
   - **At the beginning**: (O(1))
   - **At the end**: (O(1)) (if there is a tail pointer, otherwise (O(n)))
   - **After a specific node**: (O(1)) (if the node is given)
2. **Deletion**:
   - **From the beginning**: (O(1))
   - **From the end**: (O(n)) (requires traversal to the node before the last one, unless there's a tail pointer)
   - **A specific node**: (O(n)) (if the node is not given, needs traversal to find it)
3. **Traversal**: (O(n))
4. **Searching**: (O(n))

Example in C++

Here is an example of a basic singly linked list in C++ with comments indicating time complexities:

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

class Node {
public:
    int data;
    Node* next;
    Node(int value) {
        data = value;
        next = nullptr;
    }
};

class LinkedList {
public:
    Node* head;
    LinkedList() {
        head = nullptr;
    }

    // Time Complexity: O(1)
    void insertAtBeginning(int value) {
        Node* newNode = new Node(value);
        newNode->next = head;
        head = newNode;
    }

    // Time Complexity: O(n)
    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;
    }

    // Time Complexity: O(n)
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }

    // Time Complexity: O(n)
    Node* search(int value) {
        Node* temp = head;
        while (temp != nullptr) {
            if (temp->data == value) {
                return temp;
            }
            temp = temp->next;
        }
        return nullptr;
    }

    // Time Complexity: O(n)
    void deleteNode(int value) {
        if (head == nullptr) return;
        if (head->data == value) {
            Node* toDelete = head;
            head = head->next;
            delete toDelete;
            return;
        }
        Node* temp = head;
        while (temp->next != nullptr && temp->next->data != value) {
            temp = temp->next;
        }
        if (temp->next == nullptr) return;
        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        delete toDelete;
    }
};

int main() {
    LinkedList list;
    list.insertAtBeginning(10); // O(1)
    list.insertAtEnd(20);       // O(n)
    list.insertAtEnd(30);       // O(n)
    list.display();             // O(n)
    Node* found = list.search(20); // O(n)
    if (found) {
        cout << "Found: " << found->data << endl;
    }
    list.deleteNode(20);        // O(n)
    list.display();             // O(n)
    return 0;
}
```

Applications

- **Dynamic Memory Allocation**: Linked lists can easily grow and shrink in size.
- **Implementation of Stacks and Queues**: Using linked lists provides flexibility.
- **Graphs**: Adjacency lists in graph representations are implemented using linked lists.
- **Navigation Systems**: Used in applications like browser history where forward and backward navigation is required.

Advantages

- **Dynamic Size**: Can grow or shrink as needed.
- **Efficient Insertions/Deletions**: Easier to insert or delete elements compared to arrays.

Disadvantages

- **Memory Overhead**: Extra memory required for storing pointers.
- **Sequential Access**: Nodes need to be accessed sequentially from the head, making it slower compared to arrays for certain operations.

Summary of Key Differences:

Lists, typically implemented as doubly-linked lists

Feature	Array	List (std::list)
Size	Fixed	Dynamic
Memory Allocation	Contiguous	Non-contiguous (nodes)
Access Time	O(1)	O(n)
Insertion/Deletion	Expensive (O(n) worst case)	Efficient (O(1) if position known)
Initialization	Can be initialized inline	Can be initialized inline
Flexibility	Less flexible	More flexible
Cache Performance	Better	Worse

Feature	Array	List (std::list)
Use Case	When size is known and fixed	When size is unknown or frequently changes

Choosing between an array and a list depends on the specific needs of your application. Arrays are preferred for scenarios requiring fast element access and where the size is fixed, while lists are better suited for scenarios where the size of the collection changes frequently and element insertion/deletion operations are common.