Zakho Technical College

Department of Computer Information System

Data Structures and Algorithms

4. Analysis (Singly Linked List and Doubly Linked List)

Lecturers:

Sipan M. Hameed

Ahmed Jamil

www.sipan.dev

2024-2025

1. Table of Contents

4. Singly Linked List and Doubly Linked List

A **linked list** is a linear data structure consisting of nodes where each node contains data and a reference (pointer) to the next node in the sequence. Unlike arrays, linked lists allow efficient insertions and deletions without reallocation or reorganization of the entire structure.

4.1 Types of Linked Lists

1. **Singly Linked List**: Each node points only to the next node.
2. **Doubly Linked List**: Each node has pointers to both the next and previous nodes, enabling bidirectional traversal.

4.2 Singly Linked List Implementation

Node Class

```
public class Node<T>
{
    public T Data { get; set; }
    public Node<T> Next { get; set; }

    public Node(T data)
    {
        Data = data;
        Next = null;
    }
}
```

Singly Linked List Class

```
public class SinglyLinkedList<T>
{
    private Node<T> head;

    // Add to the head of the list
    public void AddFirst(T data)
    {
        Node<T> newNode = new Node<T>(data);
        newNode.Next = head;
        head = newNode;
    }

    // Add to the tail of the list
    public void AddLast(T data)
    {
        Node<T> newNode = new Node<T>(data);
        if (head == null)
        {
```

```
            head = newNode;
            return;
        }
        Node<T> current = head;
        while (current.Next != null)
        {
            current = current.Next;
        }
        current.Next = newNode;
    }

    // Remove the first node
    public void RemoveFirst()
    {
        if (head != null)
        {
            head = head.Next;
        }
    }

    // Remove the last node
    public void RemoveLast()
    {
        if (head == null) return;
        if (head.Next == null)
        {
            head = null;
            return;
        }
        Node<T> current = head;
        while (current.Next.Next != null)
        {
            current = current.Next;
        }
        current.Next = null;
    }

    // Check if the list contains a value
    public bool Contains(T data)
    {
        Node<T> current = head;
        while (current != null)
        {
            if (EqualityComparer<T>.Default.Equals(current.Data, data))
                return true;
            current = current.Next;
        }
        return false;
    }
```

```
    // Get the number of nodes
    public int Count()
    {
        int count = 0;
        Node<T> current = head;
        while (current != null)
        {
            count++;
            current = current.Next;
        }
        return count;
    }
}
```

4.3 Time/Space Complexity Analysis

Operation	Time Complexity	Space Complexity	Description
AddFirst()	O(1)	O(1)	Adds a node to the head.
AddLast()	O(n)	O(1)	Adds a node to the tail.
RemoveFirst()	O(1)	O(1)	Removes the head node.
RemoveLast()	O(n)	O(1)	Removes the tail node.
Contains()	O(n)	O(1)	Checks if a value exists.
Count()	O(n)	O(1)	Returns the number of nodes.

Space Complexity: O(n) (stores n nodes, each with data and a pointer).

4.4 Built-in LinkedList<T> (Doubly Linked List)

C# provides a generic LinkedList<T> class in the System.Collections.Generic namespace, which implements a doubly linked list.

4.5 Example Code(Doubly Linked List)

```
using System;
using System.Collections.Generic;

LinkedList<int> dll = new LinkedList<int>();

// Add to head
dll.AddFirst(10); // List: 10
dll.AddFirst(5);  // List: 5 -> 10

// Add to tail
dll.AddLast(20);  // List: 5 -> 10 -> 20

// Remove from head
dll.RemoveFirst(); // List: 10 -> 20

// Remove from tail
dll.RemoveLast();  // List: 10

// Insert after a node
LinkedListNode<int> node = dll.Find(10); // O(n)
if (node != null) dll.AddAfter(node, 15); // List: 10 -> 15

// Check if a value exists
bool has15 = dll.Contains(15); // True

// Get count
Console.WriteLine(dll.Count); // Output: 2
```

4.6 Time/Space Complexity Analysis(Doubly Linked List)

Operation	Time Complexity	Space Complexity	Description
AddFirst()	O(1)	O(1)	Adds a node to the head.
AddLast()	O(1)	O(1)	Adds a node to the tail.
RemoveFirst()	O(1)	O(1)	Removes the head node.
RemoveLast()	O(1)	O(1)	Removes the tail node.
AddAfter()	O(1)	O(1)	Inserts after a known node.
Contains()	O(n)	O(1)	Checks if a value exists.
Count	O(1)	-	Returns the number of nodes.

Space Complexity: O(n) (each node has two pointers: Next and Previous).

4.7 Key Differences: Singly vs. Doubly Linked List

Feature	Singly Linked List	Doubly Linked List
Pointers	Next only	Next and Previous
Insert/Delete Tail	O(n) (without tail pointer)	O(1) (built-in tail pointer)
Traversal	Forward-only	Bidirectional
Memory Overhead	Lower (1 pointer/node)	Higher (2 pointers/node)
Use Case	Simple FIFO operations	Frequent head/tail modifications

---

4.8 Advantages and Disadvantages (Doubly Linked List)

- **Advantages**:
  - **Dynamic Size**: No need to preallocate memory.
  - **Efficient Insertions/Deletions**: O(1) for head/tail in doubly linked lists.
- **Disadvantages**:
  - **No Random Access**: Accessing elements by index is O(n).
  - **Memory Overhead**: Extra pointers consume memory.
  - **Cache Inefficiency**: Nodes may be scattered in memory.

4.9 Use Cases

- **Singly Linked List**: Queues, stacks, or scenarios requiring forward-only traversal.
- **Doubly Linked List**: Undo/redo functionality, browser history, or applications needing bidirectional navigation.

4.10 Summary

- **Singly Linked List**: Simple and memory-efficient but limited to forward operations.
- **Doubly Linked List**: Flexible with bidirectional traversal but uses more memory.
- **Built-in LinkedList<T>**: Use for efficient head/tail operations and bidirectional traversal.

Choose a linked list when frequent insertions/deletions are required, and order matters. Use arrays or lists (List<T>) for scenarios needing fast random access.