LinkedList and Binary Tee

Introduction to Linked Lists

What is a Linked List?

- A linear data structure where elements are not stored in contiguous memory locations.
- Each element, called a *node*, contains:
  - **Data:** The value being stored.
  - **Next Pointer:** A reference to the next node in the sequence.
- The first node is the **head**, and the last node's next pointer is null.
- **Analogy:** Think of a treasure hunt where each clue points to the location of the next clue.

Types of Linked Lists

- **Singly Linked List:** Nodes have a pointer only to the next node. (The one we've been discussing)
- **Doubly Linked List:** Nodes have pointers to both the next and the previous nodes, allowing for bidirectional traversal.
- **Circular Linked List:** The last node's next pointer points back to the head, forming a loop.

The LinkedList<T> Class in C#

- C# provides the generic LinkedList<T> class in the System.Collections.Generic namespace.
- It's a **doubly linked list**, offering efficient insertion and deletion.
- Key advantage: Dynamic size, can grow or shrink as needed.

LinkedList<T>: Core Operations

- **Adding Nodes:**
  - AddFirst(T value): Adds at the beginning (head).
  - AddLast(T value): Adds at the end (tail).
  - AddBefore(LinkedListNode<T> node, T value): Inserts before a specific node.
  - AddAfter(LinkedListNode<T> node, T value): Inserts after a specific node.

- **Removing Nodes:**
  - Remove(T value): Removes the first occurrence of a value.
  - Remove(LinkedListNode<T> node): Removes a specific node.
  - RemoveFirst(): Removes the first node.
  - RemoveLast(): Removes the last node.

- **Finding Nodes:**
  - Find(T value): Returns the first node containing the value.
  - Contains(T value): Checks if the list contains a value (returns bool).

- **Other:**
  - Clear(): Removes all nodes.
  - Count: Returns the number of nodes.
  - First: Returns the first node.
  - Last: Returns the last node.

LinkedList<T> Example 1: Creating and Adding

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

public class LinkedListExample1
{
    public static void Main(string[] args)
    {
        // Create a linked list of integers.
        LinkedList<int> numbers = new LinkedList<int>();

        // Add elements to the end.
        numbers.AddLast(1);
        numbers.AddLast(2);
        numbers.AddLast(3);  // numbers: 1 -> 2 -> 3

        // Add an element to the beginning.
```

```
        numbers.AddFirst(0); // numbers: 0 -> 1 -> 2 -> 3

        // Add an element before a specific node.
        LinkedListNode<int> node2 = numbers.Find(2); // Find the node containing 2
        numbers.AddBefore(node2, 15); // numbers: 0 -> 1 -> 15 -> 2 -> 3

        // Add an element after a specific node.
        LinkedListNode<int> node1 = numbers.Find(1);
        numbers.AddAfter(node1, 7); // numbers: 0 -> 1 -> 7 -> 15 -> 2 -> 3

        Console.WriteLine("List after adding:");
        foreach (int number in numbers)
        {
            Console.Write(number + " -> ");
        }
        Console.WriteLine("null");
    }
}
```

LinkedList<T> Example 2: Removing and Finding

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

public class LinkedListExample2
{
    public static void Main(string[] args)
    {
        // Continue with the list from the previous example
        LinkedList<int> numbers = new LinkedList<int>();
        numbers.AddLast(1);
        numbers.AddLast(2);
        numbers.AddLast(3);
        numbers.AddFirst(0);
        LinkedListNode<int> node2 = numbers.Find(2);
        numbers.AddBefore(node2, 15);
        LinkedListNode<int> node1 = numbers.Find(1);
        numbers.AddAfter(node1, 7);
        // Remove the first occurrence of a value.
        numbers.Remove(15); // numbers: 0 -> 1 -> 7 -> 2 -> 3

        // Remove a specific node.
        LinkedListNode<int> nodeToRemove = numbers.Find(7);
        numbers.Remove(nodeToRemove); // numbers: 0 -> 1 -> 2 -> 3

        // Remove the first element.
```

```
        numbers.RemoveFirst(); // numbers: 1 -> 2 -> 3

        // Remove the last element.
        numbers.RemoveLast(); // numbers: 1 -> 2

        // Find a node.
        LinkedListNode<int> foundNode = numbers.Find(2);
        if (foundNode != null)
        {
            Console.WriteLine($"\nFound node with value: {foundNode.Value}"); // Output: 
Found node with value: 2
        }

        // Check if the list contains a value
        bool containsTwo = numbers.Contains(2);  //true
        bool containsFive = numbers.Contains(5); //false
        Console.WriteLine($"Contains 2: {containsTwo}, Contains 5: {containsFive}");

        Console.WriteLine("\nList after removing:");
        foreach (int number in numbers)
        {
            Console.Write(number + " -> ");
        }
        Console.WriteLine("null");
        Console.WriteLine($"Count: {numbers.Count}"); // Output: Count: 2
    }
}
```

==

Introduction to Binary Trees

- A hierarchical data structure where each node has at most two children:
  - **Left child**
  - **Right child**
- The top node is the **root**.
- Nodes without children are called **leaves**.
- Useful for representing hierarchical relationships and efficient searching/sorting.
- **Analogy**: Family tree, organizational chart.

Usage Examples:

- **Efficient searching (Binary Search Trees)**
- **Hierarchical data representation (File Systems)**
- **Expression parsing (Compilers)**

Binary Tree Terminology

- **Root:** The topmost node in the tree.
- **Parent:** A node that has one or two children.
- **Child:** A node directly below a parent node.
- **Sibling:** Nodes that share the same parent.
- **Leaf:** A node with no children (left or right).
- **Edge:** The connection between two nodes.
- **Path:** A sequence of nodes and edges connecting a node to a descendant.
- **Depth of a Node:** The number of edges from the root to the node.
- **Height of a Tree:** The maximum depth of any node in the tree.
- **Subtree:** A tree formed by a node and its descendants.

Types of Binary Trees

Title: Types of Binary Trees

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are filled except possibly the last, and nodes are filled from left to right.
- **Perfect Binary Tree:** All levels are completely filled.
- **Balanced Binary Tree:** The height of the left and right subtrees of every node differ by at most 1.

Representing a Binary Tree Node in C#

```
public class TreeNode<T>
{
   public T Data { get; set; }       // The data stored in the node
   public TreeNode<T> Left { get; set; }  // Reference to the left child
   public TreeNode<T> Right { get; set; } // Reference to the right child

   public TreeNode(T data)
   {
      this.Data = data;
      this.Left = null;  // Initialize left child to null
      this.Right = null; // Initialize right child to null
   }
}
```

- This defines a generic node class. T can be any data type (e.g., int, string, custom objects).
- The Left and Right properties are also TreeNode<T> objects, forming the tree structure.

Representing a Binary Tree in C#

```
public class BinaryTree<T>
{
   public TreeNode<T> Root { get; set; } // Reference to the root node

   public BinaryTree()
   {
      this.Root = null; // Initially, the tree is empty
   }

   // Methods for operations like insertion, traversal, etc., would go here.
}
```

- The BinaryTree<T> class holds the Root and would contain methods to manipulate the tree.

Binary Tree Traversal

- The process of visiting (processing) each node in the tree exactly once.
- Common traversal methods:
  - **In-Order Traversal:** Left -> Root -> Right
  - **Pre-Order Traversal:** Root -> Left -> Right
  - **Post-Order Traversal:** Left -> Right -> Root
- **Level-Order Traversal:** Visit nodes level by level.

[Image: Diagrams illustrating the order of node visitation for in-order, pre-order, post-order, and level-order traversal.]

In-Order Traversal Example in C#

```
using System;

public class BinaryTreeTraversal
{
   // Recursive function for in-order traversal
   public static void InOrderTraversal(TreeNode<int> root)
   {
      if (root != null)
      {
         // 1. Traverse the left subtree
         InOrderTraversal(root.Left);

         // 2. Process the current node (e.g., print its data)
         Console.Write(root.Data + " ");
```

```
            // 3. Traverse the right subtree
            InOrderTraversal(root.Right);
        }
    }

    public static void Main(string[] args)
    {
        // Create a sample binary tree
        TreeNode<int> root = new TreeNode<int>(4);
        root.Left = new TreeNode<int>(2);
        root.Right = new TreeNode<int>(6);
        root.Left.Left = new TreeNode<int>(1);
        root.Left.Right = new TreeNode<int>(3);
        root.Right.Left = new TreeNode<int>(5);
        root.Right.Right = new TreeNode<int>(7);
        //      4
        //     / \
        //    2   6
        //   / \ / \
        //  1  3 5  7

        Console.WriteLine("In-Order Traversal:");
        InOrderTraversal(root); // Output: 1 2 3 4 5 6 7
    }
}
```

Pre-Order Traversal Example in C#

```
using System;
public class BinaryTreeTraversal
{
  public static void PreOrderTraversal(TreeNode<int> root)
  {
    if(root != null)
    {
      Console.Write(root.Data + " ");
      PreOrderTraversal(root.Left);
      PreOrderTraversal(root.Right);
    }
  }
  public static void Main(string[] args)
  {
     TreeNode<int> root = new TreeNode<int>(4);
        root.Left = new TreeNode<int>(2);
        root.Right = new TreeNode<int>(6);
        root.Left.Left = new TreeNode<int>(1);
```

```
        root.Left.Right = new TreeNode<int>(3);
        root.Right.Left = new TreeNode<int>(5);
        root.Right.Right = new TreeNode<int>(7);

        Console.WriteLine("Pre-Order Traversal:");
        PreOrderTraversal(root);
  }
}
```

Post-Order Traversal Example in C#

```
using System;
public class BinaryTreeTraversal
{
  public static void PostOrderTraversal(TreeNode<int> root)
  {
    if(root != null)
    {
      PostOrderTraversal(root.Left);
      PostOrderTraversal(root.Right);
      Console.Write(root.Data + " ");
    }
  }
  public static void Main(string[] args)
  {
    TreeNode<int> root = new TreeNode<int>(4);
        root.Left = new TreeNode<int>(2);
        root.Right = new TreeNode<int>(6);
        root.Left.Left = new TreeNode<int>(1);
        root.Left.Right = new TreeNode<int>(3);
        root.Right.Left = new TreeNode<int>(5);
        root.Right.Right = new TreeNode<int>(7);

        Console.WriteLine("Post-Order Traversal:");
        PostOrderTraversal(root);
  }
}
```

Conclusion

- Linked lists offer efficient insertion and deletion, but accessing elements by index can be slower than arrays. C#'s LinkedList<T> class provides a robust implementation.
- Binary trees are versatile hierarchical structures used in various applications, including searching, sorting, and representing relationships. Understanding tree traversals is crucial for working with binary trees.
- C# allows you to create and manipulate both LinkedList<T> and custom binary tree implementations.