🌳 Non-linear Data Structures in Java

1. Introduction to Trees

1.1 What is a Tree?

A tree is a hierarchical data structure that consists of nodes connected by edges. It is a collection of nodes where each node has a value, and possibly children, which are other nodes. The topmost node is called the **root**, and the nodes at the bottom without children are called **leaves**.

1.2 Types of Trees

- **Binary Trees**: A tree where each node has at most two children, referred to as the left child and the right child.
- **Binary Search Trees (BST)**: A binary tree with the additional property that for any node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater.
- **AVL Trees**: A self-balancing binary search tree where the difference in heights between the left and right subtrees cannot be more than one.
- **Red-Black Trees**: A balanced binary search tree where each node has an extra bit for denoting the color of the node, either red or black, which helps in balancing the tree during insertions and deletions.

---

2. Binary Trees

2.1 Structure of a Binary Tree

A binary tree is a tree in which each node has at most two children. These children are referred to as the left child and the right child. A binary tree is defined by:

```java
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}
```

2.2 Traversal Methods

Traversal of a tree means visiting each node exactly once. There are three types of traversal:

- **Inorder Traversal**: Visit left subtree, node, right subtree.
- **Preorder Traversal**: Visit node, left subtree, right subtree.
- **Postorder Traversal**: Visit left subtree, right subtree, node.

Example: Inorder Traversal

```
void inorder(Node node) {
    if (node == null)
        return;

    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
}
```

Output:

```
Given the tree:
    1
   / \
  2   3
 / \
4   5

The Inorder traversal is: 4 2 5 1 3
```

---

3. Binary Search Trees (BST)

3.1 Properties of BST

- **Left Subtree**: Contains nodes with values less than the root.
- **Right Subtree**: Contains nodes with values greater than the root.
- **No Duplicate Nodes**: All nodes are distinct.

3.2 Insertion in BST

To insert a node in BST, we start at the root and compare the node's value with the root's value. If the value is less, we go to the left subtree; if it's more, we go to the right subtree.

Example: Insert in BST

```
Node insert(Node root, int key) {
    if (root == null) {
        root = new Node(key);
        return root;
    }
    if (key < root.data)
        root.left = insert(root.left, key);
    else if (key > root.data)
        root.right = insert(root.right, key);
    return root;
}
```

Output:

```
Insert 6 into the BST:
    5
   / \
  3   7

After insertion:
    5
   / \
  3   7
     /
    6
```

---

4. AVL Trees

4.1 Introduction to Balanced Trees

AVL trees are self-balancing binary search trees. The height of two child subtrees of any node differs by at most one.

4.2 Rotations in AVL Trees

Rotations are used to balance the tree when nodes are inserted or deleted. There are four types of rotations:

- **Left Rotation**
- **Right Rotation**
- **Left-Right Rotation**
- **Right-Left Rotation**

4.3 Insertion in AVL Trees

Insertion may cause the tree to become unbalanced, requiring rotations to balance it.

Example: Right Rotation

```
Node rightRotate(Node y) {
    Node x = y.left;
    Node T2 = x.right;

    x.right = y;
    y.left = T2;

    return x;
}
```

Output:

```
Before Rotation:
    30
   /
  20
 /
10

After Right Rotation:
    20
   / \
  10  30
```

---

5. Red-Black Trees

5.1 Properties of Red-Black Trees

- Each node is either red or black.
- The root is always black.
- Red nodes cannot have red children (no two red nodes can be adjacent).
- Every path from a node to its descendant leaves must have the same number of black nodes.

5.2 Insertion and Balancing in Red-Black Trees

During insertion, we follow the properties of the Red-Black tree and adjust colors and perform rotations if necessary to maintain the balance.

---

6. Graphs

6.1 What is a Graph?

A graph is a data structure that consists of a set of vertices (nodes) and a set of edges connecting pairs of vertices. Graphs can represent many real-world problems like social networks, maps, etc.

6.2 Types of Graphs

- **Directed Graph**: Edges have a direction.
- **Undirected Graph**: Edges have no direction.
- **Weighted Graph**: Edges have weights (or costs).
- **Unweighted Graph**: Edges have no weights.

6.3 Graph Representation

Adjacency Matrix	Adjacency List

Adjacency Matrix	Adjacency List
Uses a 2D array to represent the graph	Uses an array of lists
Suitable for dense graphs	Suitable for sparse graphs
Easy to check if two nodes are connected	Efficient in terms of space
Takes up more memory	Takes up less memory

Example: Adjacency Matrix

```
int graph[][] = new int[][] { { 0, 1, 0, 0 },
                              { 1, 0, 1, 0 },
                              { 0, 1, 0, 1 },
                              { 0, 0, 1, 0 } };
```

Example: Adjacency List

```
class Graph {
    private LinkedList<Integer> adjLists[];

    Graph(int vertices) {
        adjLists = new LinkedList[vertices];
        for (int i = 0; i < vertices; i++)
            adjLists[i] = new LinkedList();
    }

    void addEdge(int src, int dest) {
        adjLists[src].add(dest);
        adjLists[dest].add(src);
    }
}
```

6.4 Graph Traversal

6.4.1 Breadth-First Search (BFS)

BFS starts at a selected node and explores all its neighbors before moving to the next level.

Example: BFS Implementation

```
void BFS(int start) {
    boolean visited[] = new boolean[V];
    LinkedList<Integer> queue = new LinkedList<Integer>();

    visited[start] = true;
    queue.add(start);
    while (queue.size() != 0) {
        start = queue.poll();
        System.out.print(start + " ");

        Iterator<Integer> i = adjLists[start].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}
```

Output:

```
Given Graph:
0-1-2
|   |
3---4

BFS starting from node 0: 0 1 3 2 4
```

6.4.2 Depth-First Search (DFS)

DFS explores as far as possible along each branch before backtracking.

Example: DFS Implementation

```
void DFS(int v) {
    boolean visited[] = new boolean[V];
    DFSUtil(v, visited);
}

void DFSUtil(int v, boolean visited[]) {
    visited[v] = true;
    System.out.print(v + " ");

    Iterator<Integer> i = adjLists[v].listIterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}
```

Output:

```
Given Graph:
0-1-2
|   |
3---4

DFS starting from node 0: 0 1 2 4 3
```

---

7. Hash Tables

7.1 Introduction to Hash Tables

A hash table is a data structure that stores key-value pairs. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

7.2 Implementation of a Hash Table

A basic hash table implementation involves storing data in an array, with each index corresponding to a key.

Example: Simple Hash Table

```
class HashTable {
    private int arr[];
    private int size;

    public HashTable(int size) {
        this.size = size;
        arr = new int[size];
        Arrays.fill(arr, -1);  // -1 indicates an empty slot
    }

    int hashFunction(int key) {
        return key % size;
    }

    void insert(int key) {
        int index = hashFunction(key);
        while (arr[index] != -1) {
            index = (index + 1) % size; // Linear probing for collision resolution
        }
        arr[index] = key;
    }

    void display() {
        for (int i = 0; i < size; i++) {
            System.out.println(i + " -> " + arr[i]);
        }
    }
}
```

Output:

```
Insert keys: 12, 22, 42 into a hash table of size 10

Hash Table:
0 -> -1
1 -> -1
2 -> 12
3 -> 22
4 -> 42
5 -> -1
6 -> -1
7 -> -1
8 -> -1
9 -> -1
```

7.3 Collision Handling

When two keys hash to the same index, a collision occurs. There are several ways to handle collisions:

7.3.1 Chaining

In chaining, each bucket points to a linked list of entries that map to the same bucket. When a collision occurs, the new entry is simply added to the list.

Example: Chaining

```
class HashTableChaining {
    private LinkedList<Integer>[] table;

    public HashTableChaining(int size) {
        table = new LinkedList[size];
        for (int i = 0; i < size; i++) {
            table[i] = new LinkedList<>();
        }
    }

    int hashFunction(int key) {
        return key % table.length;
    }

    void insert(int key) {
        int index = hashFunction(key);
        table[index].add(key);
    }

    void display() {
        for (int i = 0; i < table.length; i++) {
            System.out.print(i + " -> ");
            for (int key : table[i]) {
                System.out.print(key + " ");
            }
            System.out.println();
        }
    }
}
```

Output:

```
Insert keys: 12, 22, 32 into a hash table of size 10

Hash Table with Chaining:
0 ->
1 ->
2 -> 12 22 32
3 ->
4 ->
5 ->
6 ->
7 ->
8 ->
9 ->
```

7.3.2 Open Addressing

Open addressing is another method to handle collisions by finding another empty slot within the hash table array itself. There are several strategies for open addressing:

- **Linear Probing**: If a collision occurs, check the next slot in sequence.
- **Quadratic Probing**: If a collision occurs, check the slots in a quadratic sequence.
- **Double Hashing**: Use a second hash function to determine the step size when a collision occurs.

Difference Between Linear Probing, Quadratic Probing, and Double Hashing

Linear Probing	Quadratic Probing	Double Hashing
Searches for the next available slot sequentially.	Searches for the next available slot using a quadratic function.	Uses a second hash function to determine the next slot.
Simple and easy to implement.	Reduces clustering compared to linear probing.	Further reduces clustering and offers better performance.
May cause primary clustering.	May cause secondary clustering.	Minimizes clustering.
Example: `index = (hash + i) % size`	Example: `index = (hash + i^2) % size`	Example: `index = (hash + i * hash2(key)) % size`

---

8. Practice

8.1 Implement Tree Traversal Algorithms

Inorder Traversal

```
void inorder(Node node) {
    if (node == null)
        return;

    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
}
```

Example Output:

```
Inorder Traversal of the tree:
    1
   / \
  2   3
 / \
4   5

Output: 4 2 5 1 3
```

Preorder Traversal

```
void preorder(Node node) {
    if (node == null)
        return;

    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
}
```

Example Output:

```
Preorder Traversal of the tree:
    1
   / \
  2   3
 / \
4   5

Output: 1 2 4 5 3
```

Postorder Traversal

```
void postorder(Node node) {
    if (node == null)
        return;

    postorder(node.left);
    postorder(node.right);
    System.out.print(node.data + " ");
}
```

Example Output:

```
Postorder Traversal of the tree:
    1
   / \
  2   3
 / \
4   5

Output: 4 5 2 3 1
```

8.2 Implement Graph Traversal Algorithms

Breadth-First Search (BFS)

```
void BFS(int start) {
    boolean visited[] = new boolean[V];
    LinkedList<Integer> queue = new LinkedList<Integer>();

    visited[start] = true;
    queue.add(start);

    while (queue.size() != 0) {
        start = queue.poll();
        System.out.print(start + " ");

        Iterator<Integer> i = adjLists[start].listIterator();
        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}
```

Example Output:

```
BFS traversal starting from vertex 0:
Given Graph:
0-1-2
|   |
3---4

Output: 0 1 3 2 4
```

Depth-First Search (DFS)

```java
void DFS(int v) {
    boolean visited[] = new boolean[V];
    DFSUtil(v, visited);
}

void DFSUtil(int v, boolean visited[]) {
    visited[v] = true;
    System.out.print(v + " ");

    Iterator<Integer> i = adjLists[v].listIterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}
```

Example Output:

```
DFS traversal starting from vertex 0:
Given Graph:
0-1-2
|   |
3---4

Output: 0 1 2 4 3
```

8.3 Solve Problems Using Hash Tables

Problem 1: Detecting Duplicates

Given an array of integers, determine if the array contains any duplicates.

Example Solution:

```java
boolean containsDuplicate(int[] nums) {
    HashSet<Integer> set = new HashSet<>();
    for (int num : nums) {
        if (set.contains(num))
            return true;
        set.add(num);
    }
    return false;
}
```

Example Output:

```
Input: [1, 2, 3, 1]
Output: true

Input: [1, 2, 3, 4]
Output: false
```

Problem 2: Grouping Anagrams

Given an array of strings, group anagrams together.

Example Solution:

```java
List<List<String>> groupAnagrams(String[] strs) {
    if (strs.length == 0) return new ArrayList<>();
    Map<String, List<String>> map = new HashMap<>();
    for (String s : strs) {
        char[] ca = s.toCharArray();
        Arrays.sort(ca);
        String key = String.valueOf(ca);
        if (!map.containsKey(key)) map.put(key, new ArrayList<>());
        map.get(key).add(s);
    }
    return new ArrayList<>(map.values());
}
```

Example Output:

```
Input: ["eat","tea","tan","ate","nat","bat"]
Output: [["eat","tea","ate"],["tan","nat"],["bat"]]
```

Conclusion

This detailed guide covers all the essential concepts, differences, and practical implementations of Non-linear Data Structures in Java. By following these explanations and code examples, beginners will be able to grasp the fundamentals and start applying them in real-world scenarios.