Columbia River Watershed Binary Tree Project Report

1. Introduction

The Columbia River, one of the most significant natural features of the Pacific Northwest, is supported by a vast network of tributaries and controlled by numerous dams. This project models the Columbia River and its watershed using a binary tree structure in C++. It enables traversal of river features, addition of new tributaries and dams, and outputs key information. A special emphasis is placed on the inclusion of the Spokane River's Monroe Street Dam.

2. Initial Design and Changes

Initial Design Choices:

- Binary Tree Structure: Designed with each node representing either a tributary or a dam.

- Pointers: `left` is primarily used for tributaries and `right` for dams.

- Data Encapsulation:

- `Tributary` and `Dam` structures encapsulate relevant attributes.

- `RiverNode` objects store one of the two, never both, and point to children.

- Root Node: Represents the Columbia River's mouth and serves as the tree's entry point.

Design Adjustments:

- Separate Constructors for Dams and Tributaries in `RiverNode`.

- Manual vs. Dynamic Input: Originally intended user input for node creation; final implementation shifted toward hardcoded data for testing.

- Binary File Planning: Considered using char arrays for binary file compatibility but left serialization as a future improvement.

3. How to Use the Program

High-Level Overview:

The `RiverTree` class provides methods to:

- Add tributaries and dams using either direct input or parameterized objects.

- Print all tributaries and dams in the watershed.

- Traverse the river network (currently in pre-order logic).

- Traverse specifically to Monroe Street Dam (via naming convention or traversal logic).

Components:

- `RiverNode`: Stores data for a dam or tributary and tree structure.

- `Dam`, `Tributary`: Structures containing relevant hydrological or infrastructure data.

- `RiverTree`: Manages the root, insertion, traversal, and output functions.

4. Ease of Use

The codebase is modular and intuitive:

- Clear naming conventions (`add_dam`, `add_trib`, `print_tribs`, etc.).

- Logical class separation and encapsulated data.

- Interactive input or batch initialization options.

- Traverse methods provide useful debugging and verification tools.

5. Program Features – Formal Specifications

Feature: `add_trib()`

- Purpose: Adds a new tributary to the tree.

- Assumptions: Valid `Tributary` object passed.

- Inputs: `Tributary* trib`

- Outputs: None directly, node is inserted.

- State Changes: Tree structure updated with new left child.

- Expected Behavior: Tributary appears on the leftmost available node.

Feature: `add_dam()`

- Purpose: Adds a new dam to the river tree.

- Assumptions: Valid `Dam` object passed.

- Inputs: `Dam* dam`

- Outputs: None directly.

- State Changes: Tree updated with new dam as a right child.

- Expected Behavior: Dam node added to the rightmost available node.

Feature: `traverse()`

- Purpose: Walks through the river system from the mouth onward.

- Assumptions: Tree has been populated.

- Inputs: None.

- Outputs: Console output.

- State Changes: None.

- Expected Behavior: Outputs river flow including tributaries and dams.

6. Class Features – Formal Specifications

Class: `RiverTree`

- Purpose: Encapsulates the entire watershed system using a binary tree.

- Assumptions: Constructed with the Columbia River as root.

- Inputs: Optional pre-built tributary/dam data.

- Outputs: Console logs or future file exports.

- State Changes: Nodes added to tree.

- Expected Behavior: Accurate modeling of the river system.

Class: `RiverNode`

- Purpose: Represents a single node in the river system.

- Assumptions: Node is either a dam or a tributary.

- Inputs: Constructors for tributary/dam input.

- Outputs: N/A

- State Changes: Memory allocation for `Tributary` or `Dam`.

- Expected Behavior: Stores and links correct information.