Contents

1 Introduction

This lab focuses on tree data structures, including Binary Search Trees (BST), AVL Trees, and 2-3-4 Trees. You will implement functions that manipulate these structures in various ways.

1.1 Prerequisites and Setup

Before starting this lab, you must download and extract the lab files. Follow these steps carefully:

1. **Download from YULearn:** Log in to YULearn and download the compressed lab archive (e.g., `lab5_1.tar.gz`) to your Windows Downloads folder
2. **Open Ubuntu WSL Terminal:**
   - Click the Windows Start menu
   - Type "Ubuntu" and select the Ubuntu WSL application
   - Wait for the terminal to open
3. **Navigate to Downloads Folder:**
   ```
   cd /mnt/c/Users/user/Downloads/
   ```
4. **Extract the Lab Files:**
   ```
   make uncompress ARCHIVE=lab5_1.tar.gz
   ```
5. **Navigate to Lab Directory:**
   ```
   cd lab5_1/
   ```
6. **Verify Setup:** Check that the extraction was successful:
   ```
   ls -la
   ```
   You should see directories: `src/`, `include/`, `examples/`, and files: `Makefile`, `questions.pdf`

1.1.1 Building and Running the Lab

Once setup is complete, you can build and run the demonstration programs:

- **Build the program:**
  ```
  make
  ```
  This compiles all source files and creates the executable in `bin/program`
- **Run all question demonstrations:**

```
make run
# or
./bin/program
```

This runs demonstrations for all questions

- **Run a specific question:**

```
./bin/program 1    # Run Question 1 only
./bin/program 2    # Run Question 2 only
./bin/program 3    # Run Question 3 only
./bin/program 4    # Run Question 4 only
```

Use these commands to test individual questions as you implement them

- **Clean build files:**

```
make clean
```

Removes compiled object files and executables

Tip: Use `./bin/program N` frequently while working on each question to verify your implementation matches the expected output.

1.2 Provided Tree Classes

This lab provides three fully implemented tree classes:

- **BST<T>** - Binary Search Tree
- **AVLTree<T>** - Self-balancing AVL Tree
- **Tree234<T>** - 2-3-4 Multi-way Search Tree

Each class provides the following common operations:

- `insert(value)` - Insert a value
- `remove(value)` - Remove a value
- `search(value)` - Check if value exists
- `getRoot()` - Get the root node pointer
- `nodeCount()` - Get number of nodes
- `height()` - Get tree height
- `inorderTraversal(count)` - Get sorted values array
- `print()` - Display tree structure

1.3 Provided Resources

The lab directory structure is organized as follows:

```
tree_lab/
├── include/ ..................................... Header files and templates
│   ├── Color.h ................................ Terminal color codes for visualization
│   ├── json.hpp ..................................... nlohmann/json library
│   ├── TreeNode.hpp ............... Node structures (BSTNode, AVLNode, Node234)
│   ├── BST.hpp / BST.tpp ..................... Binary Search Tree implementation
│   ├── AVLTree.hpp / AVLTree.tpp ..................... AVL Tree implementation
│   ├── Tree234.hpp / Tree234.tpp ..................... 2-3-4 Tree implementation
│   ├── TreeQuestions.h ............................ Question function declarations
│   └── Q1.tpp - Q4.tpp ............ Individual question implementations (your work)
├── src/ ..................................................... Source files
│   └── main.cpp ..................................... Demo runner program
├── examples/ ..................................... JSON test data for each question
└── Makefile ..................................................... Build automation
```

1.4 How to Use This Document

1. Read each question carefully and understand the requirements
2. Analyze the provided example to grasp expected behavior
3. Consider edge cases (empty tree, single node, etc.)
4. Design your solution before coding
5. Think about multiple solution approaches
6. Implement and test thoroughly with `./bin/program N`

1.5 Question Overview

This lab contains 4 questions, each focusing on a different tree operation:

- **Question 1:** BST to Circular Doubly Linked List (in-place conversion)
- **Question 2:** Reconstruct AVL Tree from Preorder and Heights
- **Question 3:** Find Distance Between Two Nodes with Path
- **Question 4:** Range Query with Statistics on 2-3-4 Tree

2 Questions

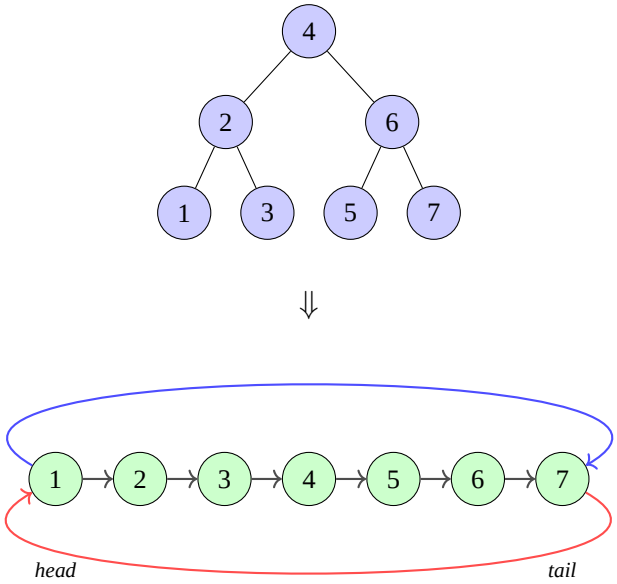
2.1 Question 1: BST to Circular Doubly Linked List

Convert a BST to a sorted circular doubly linked list **in-place**. Use `left` as `prev`, `right` as `next`. Return head (smallest element) or `nullptr` if empty.

Function Signature:

```
template <typename T>
BSTNode<T>* convertToCircularDLL(BST<T>& tree);
```

Example:




```
BST<int> bst;  // Contains: 4, 2, 6, 1, 3, 5, 7

BSTNode<int>* head = convertToCircularDLL(bst);
// head->data == 1
// head->left->data == 7 (circular prev)
// head->right->data == 2 (next)
```

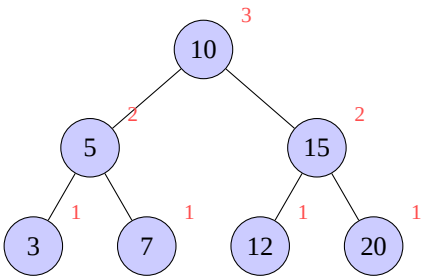
2.2 Question 2: Reconstruct AVL from Preorder and Heights

Given parallel arrays of preorder traversal and node heights, reconstruct the original AVL tree. Return `nullptr` if invalid. Caller must delete returned tree.

Function Signature:

```cpp
template <typename T>
AVLTree<T>* reconstructAVL(const T* preorder,
                           const int* heights,
                           size_t size);
```

Example:



preorder[]	10	5	3	7	15	12	20
heights[]	3	2	1	1	2	1	1

```cpp
int preorder[] = {10, 5, 3, 7, 15, 12, 20};
int heights[]  = {3,  2, 1, 1, 2,  1,  1};

AVLTree<int>* tree = reconstructAVL(preorder, heights, 7);
// tree->getRoot()->data == 10
// tree->height() == 3

delete tree;
```

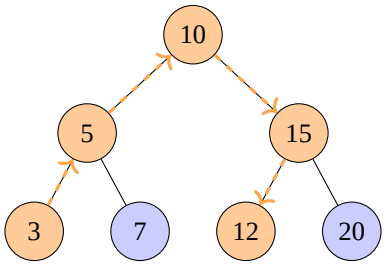
2.3 Question 3: Distance Between Nodes with Path

Find the distance (number of edges) between two nodes in a BST and return the path connecting them. Return `-1` if either node doesn't exist.

Function Signature:

```
template <typename T>
int findDistanceWithPath(const BST<T>& tree,
                         const T& val1,
                         const T& val2,
                         T* pathOut,
                         size_t& pathLength);
```

Example:



`path(3, 12) = [3, 5, 10, 15, 12]`, distance = 4

```
BST<int> bst;  // Contains: 10, 5, 15, 3, 7, 12, 20

int path[20];
size_t pathLen;

int dist = findDistanceWithPath(bst, 3, 12, path, pathLen);
// dist == 4, path == [3, 5, 10, 15, 12]

dist = findDistanceWithPath(bst, 7, 3, path, pathLen);
// dist == 2, path == [7, 5, 3]

dist = findDistanceWithPath(bst, 5, 5, path, pathLen);
// dist == 0, path == [5]
```

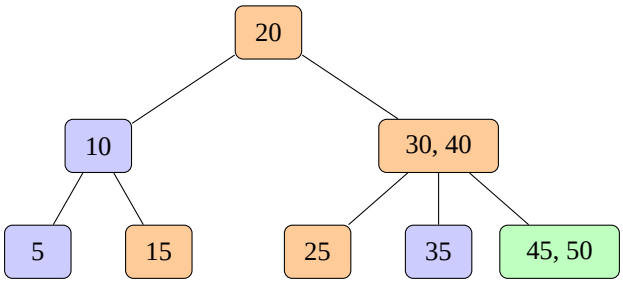
2.4 Question 4: Range Query on 2-3-4 Tree

Find all keys in a 2-3-4 tree within range $[minVal, maxVal]$ (inclusive). Return them in sorted order along with their sum, min, and max. Return `true` if any keys found.

Function Signature:

```
template <typename T>
bool rangeQuery(const Tree234<T>& tree,
                const T& minVal,
                const T& maxVal,
                T* resultOut,
                size_t& resultCount,
                T& sum,
                T& rangeMin,
                T& rangeMax);
```

Example:



Range [12, 32]: returns `[15, 20, 25, 30]` (highlighted nodes)

```
Tree234<int> tree;  // Contains: 5,10,15,20,25,30,35,40,45,50

int result[20];
size_t count;
int sum, min, max;

bool found = rangeQuery(tree, 12, 32, result, count, sum, min, max);
// found == true
// result == [15, 20, 25, 30], count == 4
// sum == 90, min == 15, max == 30

found = rangeQuery(tree, 100, 200, result, count, sum, min, max);
// found == false, count == 0
```

Node Structure:

```
template <typename T>
struct Node234 {
    T keys[3];              // Max 3 keys
    Node234* children[4];   // Max 4 children
    size_t keyCount;        // 1, 2, or 3
    bool isLeaf() const;
};
```