CS 202 – Data Structures – Fall 2025

Programming Assignment 2

(Lead TAs: Areeba Fatima, Muhammad Ali Wajid, Saif, Sarfraz)
DEADLINE: 27th October 11:55 pm

Plagiarism Policy

1. Students must not share/copy the actual program code or logic with other students. Anybody who gives or takes code will be held equally guilty for plagiarism by the DC.
2. Students must be prepared to explain any program code they submit.
3. Students cannot copy code from the Internet. All sources of assistance must be clearly cited.
4. All submissions are subject to automated plagiarism detection.
5. Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Submission Guidelines

- You must ONLY include the `solution` folder in your submission. Do not include binaries, build files, or any other directories like .exe files.
- Use the following naming convention for your zip file: `PA2_RollNumber.zip`.
- You are not allowed to modify any header files. Your implementation must work with the provided headers as-is.
- All submissions must be uploaded on LMS before the deadline.
- You are allowed 5 "free" late days during the semester (which can be applied to one or more assignments).

Important Note Before You Start

Please note that the final two components(given below) of PA2 depend on files from PA1. For now, please use your own PA1 files. However, the official PA1 solution will be released on October 5th for your use.

- **Category Tree:** Imports **post file**
- **User Search Engine**: Uses **User Manager** that in turn imports **user**, **follow_list** and **post_pool**.

Contents

1 Introduction

Programming Assignment-1 focused on **linear data structures** (linked lists, memory pools, manager classes), emphasizing sequential data storage, allocation, and access. These structures work well for ordered traversal but scale poorly for search, range queries, and random access. This assignment focuses on **hierarchical search structures** that overcome these limits and allows you to design an efficient user search engine. In this assignment, you will be working on the following components:

- **BST**: logarithmic search via key ordering.
- **AVL Tree**: self-balancing to maintain performance under all inputs.
- **Category Tree**: models hierarchical parent–child organization.
- **User Search Engine**: combines trees and hashing for fast lookups.

2 Binary Search Tree (BST)

2.1 Understanding the Provided Skeleton

The skeleton defines a generic `BST` class. The header `BST.h` gives the node structure and method declarations; the source file `BST.cpp` is where you implement them. Each node stores a key, a value, links to its children, a weak parent pointer, and a height field (reserved for AVL use later).

2.2 Core Methods and Signatures

2.2.1 Insert

Inserts a key-value pair into the tree. Nodes are arranged by key (i.e., the smallest node in the tree is the node with the smallest key).

```
// Protected helper
shared_ptr<BSTNode> insertHelper(shared_ptr<BSTNode> node, const K& key, const V& value);

// Public API
virtual bool insert(const K& key, const V& value);
```

2.2.2 Remove

Removes a node by key while preserving BST property.

```
// Protected helper
shared_ptr<BSTNode> removeHelper(shared_ptr<BSTNode> node, const K& key);

// Public API
virtual bool remove(const K& key);
```

2.2.3 Find

Locate a node by key and return a pointer to its value.

```
// Protected helper
shared_ptr<BSTNode> findHelper(shared_ptr<BSTNode> node, const K& key) const;

// Public API
V* find(const K& key);
const V* find(const K& key) const;
```

2.2.4 Min and Max

Returns the smallest or largest key-value pair.

```
// Protected helpers
shared_ptr<BSTNode> findMinHelper(shared_ptr<BSTNode> node) const;
shared_ptr<BSTNode> findMaxHelper(shared_ptr<BSTNode> node) const;

// Public API
pair<K, V> min() const;
pair<K, V> max() const;
```

2.2.5 Range Queries

Collects all key-value pairs within an inclusive key range in the result vector using the helper function. The main findRange function returns this result vector.

```
// Protected helper
void rangeHelper(shared_ptr<BSTNode> node, const K& minKey, const K& maxKey,
    vector<pair<K, V>>& result) const;

// Public API
vector<pair<K, V>> findRange(const K& minKey, const K& maxKey) const;
```

2.2.6 In-order Traversal

Produce an ordered list of all key-value pairs (in-order) which are stored by the helper in the result vector. The main inOrderTraversal function returns this vector.

```
// Protected helper
void inOrderHelper(shared_ptr<BSTNode> node, vector<pair<K, V>>& result) const;

// Public API
vector<pair<K, V>> inOrderTraversal() const;
```

2.2.7 Validation

Each node must satisfy the rule that its key lies strictly between the current bounds. If any node violates this property, the tree is not a proper BST. This method enforces the invariant that defines binary search trees.

```
// Protected helper
bool isValidBSTHelper(shared_ptr<BSTNode> node, const K* minVal, const K* maxVal
    ) const;
```

```cpp

// Public API
bool isValidBST() const;
```

2.2.8 Auxiliary / Utilities

Small utility methods exposed in the header. Their names are self-explanatory and hence we will not dedicate independent sub-sections for these.

```cpp
// Height / AVL support (protected)
void updateHeight(shared_ptr<BSTNode> node);
int getHeight(shared_ptr<BSTNode> node) const;

// Constructors / destructor / comparator (public)
BST();
BST(function<bool(const K&, const K&)> comp);
virtual ~BST() = default;

// Misc public helpers
size_t size() const;
bool empty() const;
int getTreeHeight() const;
void displayTree() const;
void displayHelper(shared_ptr<BSTNode> node, int depth) const;

// Root access (testing)
shared_ptr<BSTNode> getRoot();
void setRoot(shared_ptr<BSTNode> ptr);
```

3 AVL Tree (Self-Balancing BST)

Introduction

The AVL Tree is a Binary Search Tree that restores balance after insertions and removals by ensuring subtree height differences never exceed one.

3.1 Insert

Inserts a key-value pair while maintaining AVL balance.

```cpp
shared_ptr<BSTNode> insertAVL(shared_ptr<BSTNode> node, const K& key, const V&
    value);
bool insert(const K& key, const V& value) override;
```

3.2 Remove

Removes a key while preserving the AVL property.

```cpp
shared_ptr<BSTNode> removeAVL(shared_ptr<BSTNode> node, const K& key);
bool remove(const K& key) override;
```

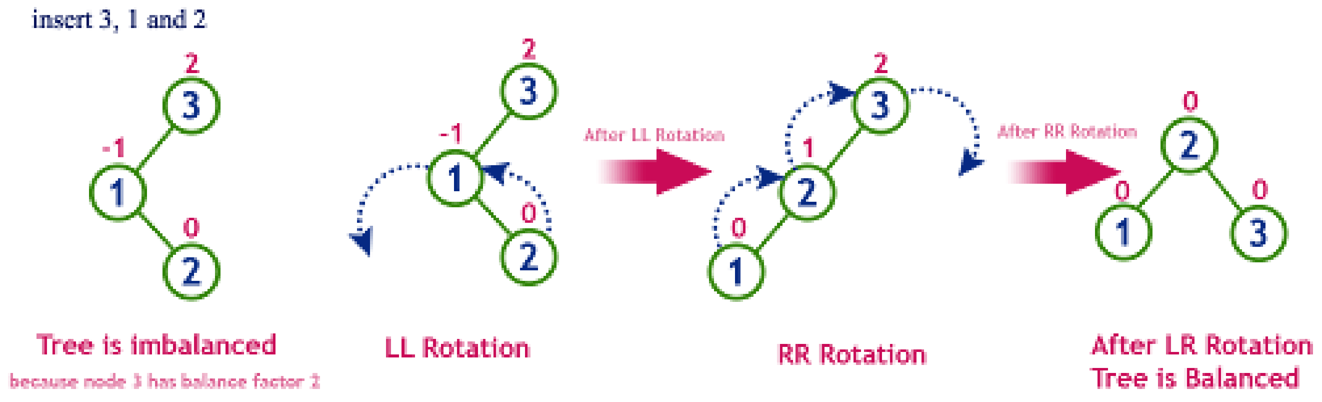


Figure 1: Example of Left-Right rotation.

3.3 Rotations

Performs the standard AVL rotations discussed in class to restore balance. Rotations are triggered only in the case of imbalance which only occur upon insertion or removal of a node from the AVL-Tree.

```
shared_ptr<BSTNode> rotateLeft(shared_ptr<BSTNode> node);
shared_ptr<BSTNode> rotateRight(shared_ptr<BSTNode> node);
shared_ptr<BSTNode> rotateLeftRight(shared_ptr<BSTNode> node);
shared_ptr<BSTNode> rotateRightLeft(shared_ptr<BSTNode> node);
```

3.4 Balance Helpers

Computes balance factor and trigger re-balancing.

```
int getBalanceFactor(shared_ptr<BSTNode> node) const;
shared_ptr<BSTNode> rebalance(shared_ptr<BSTNode> node);
```

3.5 Validation and Metrics

Check AVL invariants and measure depth.

```
bool isBalanced() const;
int getMaxDepth() const;
double getAverageDepth() const;
bool isValidAVL() const;
bool isValidAVLHelper(shared_ptr<BSTNode> node) const;
void calculateDepthStats(shared_ptr<BSTNode> node, int depth, int& totalDepth,
    int& nodeCount, int& maxDepth) const;
```

3.6 Constructors

Initialize AVLTree with default or custom comparator.

```
AVLTree();
AVLTree(function<bool(const K&, const K&)> comp);
```

4 Category Tree

Introduction

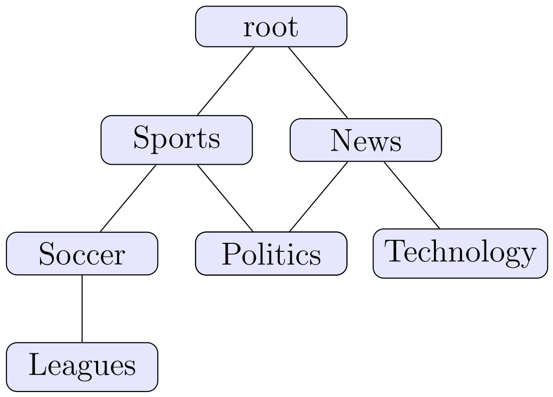


Figure 2: Example of a category hierarchy built from paths like `Sports_Soccer_Leagues`.

While BSTs and AVL trees focus on ordering by keys, the **Category Tree** organizes data into a hierarchy. This mirrors real-world systems such as forums, blogs, or file systems, where content is grouped under categories and subcategories. Each node represents one category and maintains posts directly attached to it and an aggregated count of all posts across its descendants. Unlike binary search trees, here each node may have many children, so you must think in terms of *parent–child graphs* where children are stored in a vector rather than left–right pointers in a Binary Tree.

The path parser is already provided:

```
vector<string> parseCategoryPath(const string &path) const
```

It splits an input like `"Sports_Soccer_Leagues"` on the underscore character and returns the sequence {`"Sports"`,`"Soccer"`,`"Leagues"`}, so you can assume callers hand you a vector of components rather than raw strings.

4.1 CategoryNode

We recommend you start off by implementing the category node instead of the category tree. A category node is easier to implement since it only deals with its parent node, children vector and post vector.

4.1.1 addChild

Create a new child node and attach it to this node, ensuring the child's `parent` weak pointer references you; this is a simple pointer/link operation and should not touch post counts.

```
void CategoryNode::addChild(shared_ptr<CategoryNode> child);
```

4.1.2 findChild

Search only among this node's direct children for a child with a matching name and return it (or `nullptr` if none); this is used by higher-level path-walking code.

```
shared_ptr<CategoryNode> CategoryNode::findChild(const string &childName) const;
```

4.1.3 removeChild

Detach and remove a direct child by name, making sure to clear the removed node's parent pointer; do not free the child's posts here. Return `true` if removal succeeded.

```
bool CategoryNode::removeChild(const string &childName);
```

4.1.4 addPost

Attach a `Post*` to this node's `posts` list and increment `totalPostCount` for this node. Call `updatePostCounts` (discussed below) function to ripple the addition to post count of this node up the category hierarchy.

```
void CategoryNode::addPost(Post *post);
```

4.1.5 removePost

Remove a `Post*` from this node's list if present and decrement `totalPostCount` accordingly; return `true` when the post was found and removed. Again call `updatePostCounts` (discussed below) function to ripple the affects of the post deletion up the category hierarchy.

```
bool CategoryNode::removePost(Post *post);
```

4.1.6 updatePostCounts

Recompute `totalPostCount` for this node by summing its own `posts.size()` plus the `totalPostCount` of each child. ***Don't forget to call this function*** whenever a post is added or removed.

```
void CategoryNode::updatePostCounts();
```

4.2 Category Tree

4.2.1 addCategory

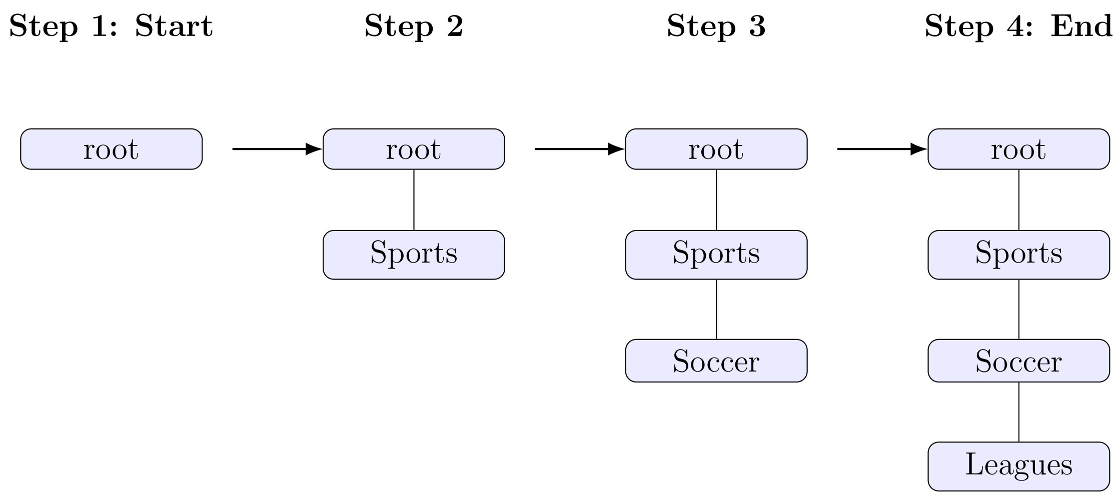


Figure 3: Path creation: split `Sports_Soccer_Leagues` and materialize each segment from the root.

Ensure a path exists by walking components (use the provided parser), creating intermediate nodes on demand; this returns `true` on success and guarantees the final node exists for later operations. The work is mostly repeated calls to `findChild` and `addChild`.

```
bool CategoryTree::addCategory(const string &categoryPath);
```

4.2.2 findCategory

Resolve a path into a node by parsing and descending child-by-child; if any component is missing return `nullptr`. This is the natural, read-only counterpart to `addCategory`.

```
shared_ptr<CategoryNode> CategoryTree::findCategory(const string &categoryPath)
    const;
```

4.2.3 addPost

Use `addCategory` to ensure the category exists, attach the post to the final node via `CategoryNode::addPost`, then walk parents (using `parent.lock()`) and increment each ancestor's `totalPostCount` so the cached totals remain correct. The key idea is: create-or-find, insert, then ripple counts up.

```
void CategoryTree::addPost(Post *post);
```

4.2.4 removePost

Locate the post's category node, remove the post from the node via `CategoryNode::removePost`, and then walk upward decrementing ancestor totals; return `false` if the post wasn't present. Keep structural removals (category deletion) separate from content removals.

```
bool CategoryTree::removePost(Post *post);
```

4.2.5 getPostsInCategory

Given a path, locate the node and either return its posts or traverse the subtree collecting posts (a simple queue-based breadth-first collection works). This function is about choosing narrow vs. wide collection and performing a traversal that accumulates `Post*`.

```
vector<Post *> CategoryTree::getPostsInCategory(const string &categoryPath, bool
    includeSubcategories = true) const;
```

4.2.6 removeCategory

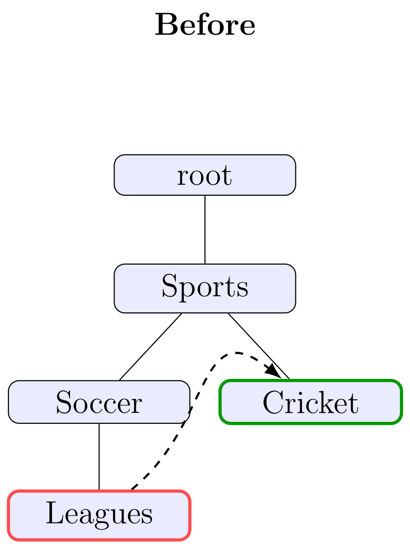
Find and detach the target subtree (the root cannot be removed), subtract the subtree's `totalPostCount` from all ancestors, and ensure the removed node's parent pointer is cleared; callers should then let the shared pointer ownership drop the subtree. Return `true` when deletion succeeded.

```
bool CategoryTree::removeCategory(const string &categoryPath);
```

4.2.7 moveCategory

Resolve the source node, detach it from its parent (updating ancestor counts), ensure the destination path exists (creating nodes if needed), then reattach the source subtree and update counts along the new ancestor chain. Think of this as cut + ensure-destination + paste + fix-counts.

```
bool CategoryTree::moveCategory(const string &fromPath, const string &toPath);
```



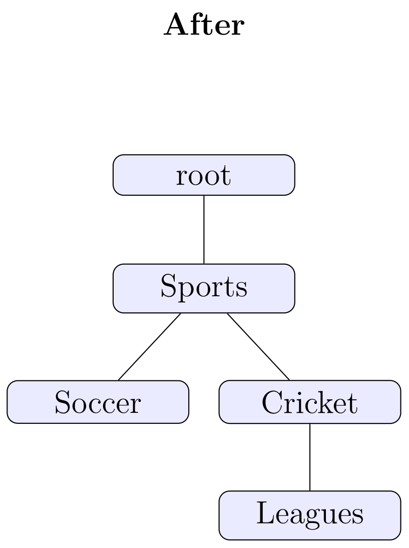


Figure 4: Moving a category: detach the subtree at `Sports_Soccer_Leagues` and reattach under `Sports_Cricket`.

4.2.8 Traversals

Traversal iterators The three iterator classes (`PreOrderIterator`, `PostOrderIterator`, `BreadthFirstIte` each encapsulate a lightweight container (stack or queue) and expose `operator*`, `operator++` and equality checks; implement them so the tree can be walked without exposing internal vectors. `displayTree()` can then use any of these iterators to render a simple textual view.

```cpp
PreOrderIterator CategoryTree::preOrderBegin() const;
PreOrderIterator CategoryTree::preOrderEnd() const;
PostOrderIterator CategoryTree::postOrderBegin() const;
PostOrderIterator CategoryTree::postOrderEnd() const;
BreadthFirstIterator CategoryTree::breadthFirstBegin() const;
BreadthFirstIterator CategoryTree::breadthFirstEnd() const;
void CategoryTree::displayTree() const;
```

Iterator operator signatures Each iterator exposes the usual operators; implement them to keep state in a small vector and return shared pointers to nodes so users can inspect `categoryName` and `totalPostCount`.

```cpp
// Example operator signatures to implement inside iterator classes
shared_ptr<CategoryNode> CategoryTree::PreOrderIterator::operator*() const;
PreOrderIterator &CategoryTree::PreOrderIterator::operator++();
bool CategoryTree::PreOrderIterator::operator!=(const PreOrderIterator &other)
    const;
```

5 User Search Engine

Introduction

The User Search Engine combines your balanced AVL structures into a practical indexing system. Two indices are maintained in parallel: one by `userID`, the other by `username`. These guarantee logarithmic lookups while supporting both numeric and lexical queries. Consistency across both indices is the core invariant: a user must appear in both or in neither. On top of these, the engine layers prefix queries and fuzzy matching to support richer functionality.

5.1 Migration from PA1

5.1.1 migrateFromLinkedList

```cpp
void migrateFromLinkedList(const LinkedList<User>& userList);
```

This function transfers all users from the PA1 linked list structure into the new AVL-based search engine. Begin by obtaining the head of the linked list. Walk through each node sequentially, accessing the user data stored within. For each user encountered, call the `addUser` method to insert it into both AVL indices (i.e., userID AVL Tree and userName AVL Tree).

5.2 User Management Operations

5.2.1 addUser

```cpp
bool addUser(User* user);
```

Adding a user requires maintaining consistency across both indices. Keep in mind that no duplicate users are allowed. Attempt to insert the user into the ID-based AVL tree using the `userID` as the key. Then insert into the username-based tree using `userName` as the key. If both insertions succeed, the operation is complete. However, if either insertion fails, you must rollback the successful one to maintain consistency. Return true only when both indices have been successfully updated.

5.2.2 removeUser

```
bool removeUser(int userID);
```

You must remove the user from both userID and username AVL indexes. The function returns true only when both data structures have been updated consistently.

5.2.3 removeUser

```
bool removeUser(const string& username);
```

Overloaded version of the same function as above. This gives the end user the flexibility to either delete using userID or username. This is just meant to serve as a little exercise in overloading.

5.3 Search Operations

5.3.1 searchByID

```
User* searchByID(int userID) const;
```

Use the userID keyed AVL tree. This method returns a pointer to the stored value, which is itself a pointer to a user. If the search succeeds, dereference this double pointer to obtain the user pointer.

5.3.2 searchByUsername

```
User* searchByUsername(const string& username) const;
```

Username lookup mirrors the ID search but operates on the string-keyed tree. This demonstrates how the dual-index design provides efficient access through either key type.

5.3.3 getUsersInIDRange

```
vector<User*> getUsersInIDRange(int minID, int maxID) const;
```

Range queries leverage the ordered nature of the ID tree. Be smart and use the AVL tree's methods to do this!

5.3.4 searchByUsernamePrefix

```
vector<User*> searchByUsernamePrefix(const string& prefix) const;
```

You can use the username keyed AVL for this. Collect the members of that tree and see in how many of them does the prefix appear (just a reminder, a prefix is the starting few characters of a string). Throw those users into a result vector and return.

5.4 Advanced Search: Fuzzy Matching

Fuzzy search tolerates small differences between the query and stored usernames, making the system robust to typos and variations. This is achieved through edit distance, a measure of similarity between strings.

Understanding Edit Distance

Edit distance counts the minimum number of single-character changes needed to transform one string into another. Three operations are allowed: inserting a character, deleting a character, or substituting one character for another. Each operation costs 1.

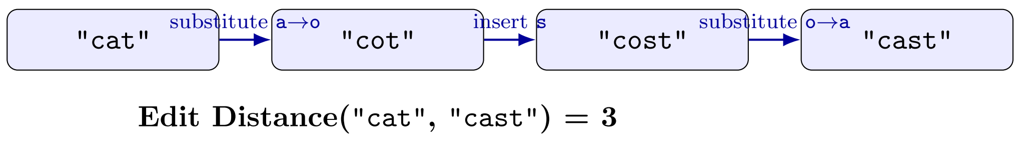


Figure 5: Example: transforming `"cat"` into `"cast"` requires 3 operations.

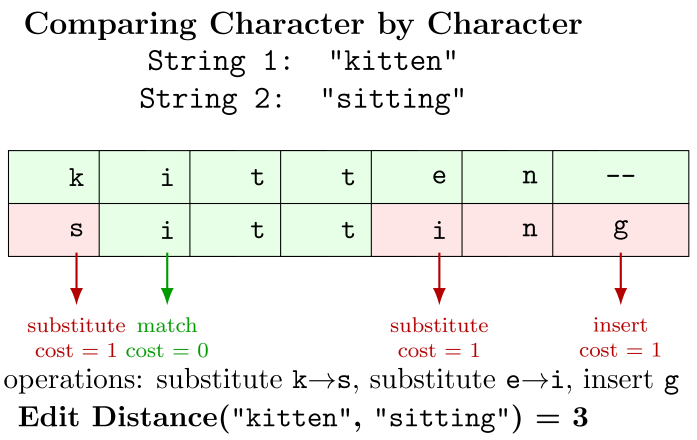

Comparing Character by Character
String 1: "kitten"
String 2: "sitting"
k i t t e n --
s i t t i n g
substitute cost = 1
match cost = 0
substitute cost = 1
insert cost = 1
Total operations: substitute k→s, substitute e→i, insert g
Edit Distance("kitten", "sitting") = 3

Figure 6: Character-by-character comparison reveals where strings differ and what operations are needed.

5.4.1 calculateEditDistance

```
int calculateEditDistance(const string& str1, const string& str2) const;
```

This function computes the minimum number of operations needed to transform one string into another. Use a recursive approach with a helper function that tracks positions in both strings. At each step, compare the current characters. If they match, advance both positions without cost. If they differ, try three possibilities: delete from the first string and advance its position, insert into the first string and advance the second position, or substitute the current character and advance both. Recursively compute the cost for each option and return the minimum plus one for the operation. Base cases occur when either string is exhausted: the cost is simply the number of remaining characters in the other string. To avoid exponential recursion, create a 2D vector to store already-computed results for each position pair.

5.4.2 fuzzyUsernameSearch

```
vector<User*> fuzzyUsernameSearch(const string& username,
                                   int maxEditDistance = 2) const;
```

Fuzzy search combines edit distance with tree traversal. Obtain all users from the username index through in-order traversal. For each stored username, calculate its edit distance from the query string. If this distance is less than or equal to the specified threshold, add the user to the results.

5.5 Validation and Statistics

Function: `bool isConsistent() const`

```
bool isConsistent() const;
```

Consistency validation ensures the two indices remain synchronized. Begin by comparing the sizes of both trees. If they differ, the system is inconsistent and the function returns false immediately. If sizes match, perform a deeper check by traversing the ID index and verifying that each user can also be found in the username index. For each user retrieved by ID, perform a username lookup and confirm that the returned pointer matches. Any mismatch indicates corruption. Return true only when all checks pass.

Function: `size_t getTotalUsers() const`

```
size_t getTotalUsers() const;
```

Simply return the size of the ID tree, which reflects the total number of indexed users.

Function: `void displaySearchStats() const`

```
void displaySearchStats() const;
```

This diagnostic function prints summary statistics about the search engine. Output the total user count and the heights of both AVL trees. This information helps verify that the trees remain balanced.

Function: `vector<User*> getAllUsersSorted(bool byID) const`

```
vector<User*> getAllUsersSorted(bool byID) const;
```

Return all indexed users in a stable order determined by `byID`. When `byID` is `true`, perform an in-order traversal of the ID AVL and collect `User*` in ascending numeric order. When `byID` is `false`, traverse the name AVL to produce users in lexicographic username order. The implementation simply maps the tree traversal result (pairs of key and value) to a flat `vector<User*>` for callers to consume.

6 Testing your Implementation

Follow these steps to compile and run the assignment:

1. Open a terminal and navigate to the root directory of the assignment (the folder containing the `Makefile`).

2. Build the test runner by typing:

```
    make

```

   This will automatically compile `test_runner` using the provided `Makefile`.

3. Once started, the test runner will display a menu of available tests along with their point distribution. For example:

```
1. BST Tester (20 points)
2. AVL Tester (25 points)
3. Category Tree Tester (30 points)
4. User Search Engine Tester (25 points)
0. Exit
```

4. Enter the number of the test you want to compile and execute (e.g., type `2` for the AVL Tester). The program will:

   (a) Compile the chosen test together with your solution files.

   (b) Run the compiled executable and show its output in the terminal.

5. You may rerun the test runner and select different tests until you exit by typing `0`.

6. To remove compiled executables and start fresh, run:

```
        make clean

```

By following these instructions, you can build and test each component of the assignment while also viewing the associated scoring breakdown.

Good Luck!