Project 7: Culinary Skill Trees – Implementing a Binary Recipe Book

---

Project Overview

Welcome to your final culinary challenge! In this project, you will implement a **binary search tree** to manage recipes and their cooking techniques. You will create a RecipeBook class that allows for efficient organization and retrieval of recipes based on their difficulty level.

As the head chef and manager, you need to efficiently manage recipes, understand the progression of cooking techniques, and plan training for your kitchen staff. This project is self-contained and does **not** require any of the classes or code from previous projects. It will test your understanding of **binary search trees**, **inheritance**, **operator overloading**, **tree traversal**, and **dynamic memory allocation** in C++.

Here is the link to accept this project on GitHub Classroom:
https://classroom.github.com/a/u5Gpp4t2

Documentation Requirements

As with all previous projects, documentation is crucial. You will receive **15%** of the points for proper documentation. Ensure that you:

1. **File-level Documentation**: Include a comment at the top of each file with your name, date, and a brief description of the code implemented in that file.
2. **Function-level Documentation**: Before each function declaration and implementation, include a comment that describes:

- **@pre**: Preconditions for the function.
- **@param**: Description of each parameter.
- **@return**: Description of the return value.
- **@post**: Postconditions after the function executes.

3. **Inline Comments**: Add comments within your functions to explain complex logic or important steps.

Remember: All files and all functions must be commented, including both `.hpp` and `.cpp` files.

Additional Resources

If you need to brush up on or learn new concepts, the following resources are recommended:

- **Binary Trees and Binary Search Trees**:
- Binary Trees (GeeksforGeeks)
- Binary Search Trees (GeeksforGeeks)
- **Inheritance and Templates**:
- Templates in C++ (cplusplus.com)
- **Operator Overloading**:
- Operator Overloading (GeeksforGeeks)
- **Dynamic Memory Allocation**:
- Pointers and Dynamic Memory (cplusplus.com)

Implementing the Recipe Book in the Virtual Bistro Simulation

You will implement the `RecipeBook` class as a subclass of a provided `BinarySearchTree` template class. The `RecipeBook` will manage `Recipe` objects, allowing you to organize and retrieve recipes based on their difficulty level efficiently.

Key Concepts

- **Binary Search Trees**: Using a binary search tree to store and manage recipes, enabling efficient search, insertion, and deletion operations.
- **Inheritance**: Extending the provided `BinarySearchTree` class to create a specialized `RecipeBook` class.
- **Operator Overloading**: Defining comparison operators for the `Recipe` struct to facilitate tree operations.
- **Tree Traversal**: Implementing preorder traversal to display the recipes.
- **Dynamic Memory Allocation**: Managing nodes and memory in the binary search tree.

Task 1: Implement the `Recipe` Struct

Define the `Recipe` struct in `RecipeBook.hpp`, but **outside** the `RecipeBook` class definition.

Data Members

- **name_**: `std::string`

The name of the recipe.

- **difficulty_level_**: `int`

An integer representing the difficulty level of the recipe (1-10).

- **description_**: `std::string`

A brief description of the recipe.

- **mastered_**: `bool`

Indicates whether the recipe has been mastered by the kitchen staff.

Constructors

1. **Default Constructor**

```
/**
* Default constructor.
* @post: Initializes name_ and description_ to empty strings,
difficulty_level_ to 0, and mastered_ to false.
*/
```

2. **Parameterized Constructor**

```
/**
* Parameterized Constructor.
* @param name The name of the recipe.
```

```
* @param difficulty_level The difficulty level of the recipe.

* @param description A brief description of the recipe.

* @param mastered Indicates whether the recipe has been mastered (default is
false).

* @post: Initializes the Recipe with the provided values.

*/
```

Operator Overloads

Implement the following operators based on `difficulty_level_`:

- **Operator==**

```
/**

* Equality operator.

* @param other A const reference to another Recipe.

* @return True if name_ is equal to other’s name_; false otherwise.

*/
```

- **Operator<**

```
/**

* Less-than operator.

* @param other A const reference to another Recipe.

* @return True if name_ is lexicographically less than other's name_; false
otherwise.
```

```
*/
```

- **Operator>**

```
/**
* Greater-than operator.
* @param other A const reference to another Recipe.
* @return True if name_ is lexicographically greater than other's name_;
false otherwise.
*/
```

Task 2: Implement the `RecipeBook` Class as a Subclass of `BinarySearchTree`

You are provided with `BinaryNode` and `BinarySearchTree` classes. Implement the `RecipeBook` class as a subclass of `BinarySearchTree<Recipe>`.

Constructors

1. **Default Constructor**

```
/**
* Default Constructor.
* @post: Initializes an empty RecipeBook.
*/
```

2. **Parameterized Constructor**

```
/**

* Parameterized Constructor.

* @param filename A const reference to a string representing the name of a
CSV file.

* @post: The RecipeBook is populated with Recipes from the CSV file.

* The file format is as follows:

* name,difficulty_level,description,mastered

* Ignore the first line. Each subsequent line represents a Recipe to be
added to the RecipeBook.

*/
```

Task 3: Implement the `findRecipe` Function

Implement the `findRecipe` function in `RecipeBook`:

```
/**

* Finds a Recipe in the tree by name.

* @param name A const reference to a string representing the name of the
Recipe.

* @return A pointer to the node containing the Recipe with the given
difficulty level, or nullptr if not found.

*/
```

Task 4: Implement the `addRecipe` Function

Implement the `addRecipe` function in `RecipeBook`:

```
/**
* Adds a Recipe to the tree.
* @param recipe A const reference to a Recipe object.
* @pre: The Recipe does not already exist in the tree (based on name).
* @post: The Recipe is added to the tree in BST order (based on name).
* @return: True if the Recipe was successfully added; false if a Recipe with
the same name already exists.
*/
```

Task 5: Implement the `removeRecipe` Function

Implement the `removeRecipe` function in `RecipeBook`:

```
/**
* Removes a Recipe from the tree by name.
* @param name A const reference to a string representing the name of the
Recipe.
* @post: If found, the Recipe is removed from the tree.
* @return: True if the Recipe was successfully removed; false otherwise.
*/
```

Task 6: Implement the `clear` Function

Implement the `clear` function in `RecipeBook`:

```
/**
* Clears all Recipes from the tree.
* @post: The tree is emptied, and all nodes are deallocated.
*/
```

Task 7: Implement the `calculateMasteryPoints` Function

Implement the `calculateMasteryPoints` function in `RecipeBook`:

```
/**
* Calculates the number of mastery points needed to master a Recipe.
* @param name A const reference to a string representing the name of the Recipe.
* @note: For a Recipe to be mastered, all Recipes with lower difficulty levels must also be mastered.
* @return: An integer representing the number of mastery points needed, or -1 if the Recipe is not found. If the recipe is already mastered, return 0.
* Note: Mastery points are calculated as the number of unmastered Recipes in the tree with a lower difficulty level than the given Recipe. Add one if the Recipe is not mastered.
*/
```

Task 8: Implement the `balance` Function

Implement the `balance` function in `RecipeBook`:

```
/**

* Balances the tree.

* @post: The tree is balanced such that for any node, the heights of its
left and right subtrees differ by no more than 1.

* @note: You may implement this by performing an inorder traversal to get
sorted Recipes and rebuilding the tree.

*/
```

Task 9: Implement the `preorderDisplay` Function

Implement the `preorderDisplay` function in `RecipeBook`:

```
/**

* Displays the tree in preorder traversal.

* @post: Outputs the Recipes in the tree in preorder, formatted as:

* Name: [name_]

* Difficulty Level: [difficulty_level_]

* Description: [description_]

* Mastered: [Yes/No]

* (Add an empty line between Recipes)

*/
```

Submission

You will submit your solution to Gradescope via GitHub Classroom. The autograder will grade the following files:

- `RecipeBook.hpp`
- `RecipeBook.cpp`

Submission Instructions:

- Ensure that your code compiles and runs correctly on the Linux machines in the labs at your institution.
- **Use the provided Makefile to compile your code.**
- You are expected to thoroughly test your code in a fashion similar to the previous projects. This project does not have a testing guide, as part of the requirement is for you to test it yourselves.
- Push the code you want to submit to the GitHub repository created by GitHub Classroom.
- Submit your assignment on Gradescope by linking it to your GitHub repository.

Grading Rubric

- **Correctness**: 80% (distributed across unit testing of your submission)
- **Documentation**: 15%
- **Style and Design**: 5% (proper naming, modularity, and organization)

Due Date

This project is **due on December 17th, 2024 at 11:00 PM EST**.

You can receive an extra 5/80 points on the project by submitting early and scoring above a certain amount of points (TBA on discord) on an early submission. That early submission deadline will be December 11 at 11:00 PM EST.

No late submissions will be accepted.

Tutoring may not be available during finals week.

Important Notes

- **Start Early**: Begin working on the project as soon as it is assigned to identify any issues and seek help if needed.
- **No Extensions**: There will be no extensions and no negotiation about project grades after the submission deadline.
- **Help Resources**: Help is available via drop-in tutoring (see Blackboard or Discord for the schedule). Starting early will allow you to get the help you need.

Authors: Michael Russo, Georgina Woo, Prof. Wole

Credit to Prof. Ligorio