Creative Project 3: B(e)ST of the B(e)ST

Specification

Learning Objectives

By completing this assignment, students will demonstrate their ability to:

- Implement a custom class that implements the `Comparable` interface.
- Implement a functionally correct Java class to represent a binary search tree structure.
- Write tests to verify correctness and functionality.
- Design classes that are readable and maintainable, and that conform to provided guidelines for style, implementation, and performance.
- Produce clear and effective documentation to improve comprehension and maintainability of programs, methods, and classes.

Assignment Details

Program Overview

In this assignment, you will implement a system to represent a real-world collection that interests you. You will represent your collection using a binary search tree (BST) to enable fast (O(log n)) operations and maintain the collection in sorted order according to a comparison operation that you will define.

At a high level, you will:

- **Design a custom class** to represent items in your collection.
- **Implement core BST operations**, ensuring the structure maintains sorted order.
- **Implement a file format** to provide file-based input/output functionality.
- **Develop a unique tree method** that either traverses or modifies the tree, using recursion.
- **Write test cases** to validate your tree's behavior.

Example Themes

- A music enthusiast might create a **Song** class storing song titles, artists, and durations.
- A sneakerhead could implement a **Shoe** class categorizing shoes by brand, size, and rarity.
- A card collector might create a **TradingCard** class storing trading cards with attributes like rarity and value.

Have fun with this! Make something that aligns with your hobbies and interests!

We have provided you with a sample implementation of a `YarnExample` class, its (unfinished) corresponding `CollectionManagerExample`, and an example `ClientExample` file. We encourage you to look over these files to see some examples of how one might implement an item and collection manager. **You may not implement Yarn as your item.**

> **NOTE:** The term "Collection" in this assignment refers to a collection of things in the real world and is not at all related to the Java Collections Framework.

System Structure

The program consists of 3 primary components:

1. Item class

> For generalizability, we will be referring to the class you implement in this section as "Item". However, you should not name your class Item, and instead, should choose a class name that accurately describes the items items you are implementing.

You should define a class in its own file to represent the items (songs, shoes, cards, etc.) in your collection. You may choose whatever fields you believe best represent your class, but you must have *at least two* characteristics of your items to represent as fields. Your class should also implement the `Comparable` interface to provide a way to compare items in your collection to each other.

At minimum, your item class **must** implement the following methods, but feel free to add any other methods you deem necessary:

```
public String toString()
```

- Produces a readable string representation of your item. You can choose the format, but **it should be clear and include relevant details.**

```
public boolean equals(Object o)
```

- This method compares two objects, returning `true` if they are equal and `false` if not. **Your implementation should compare all key fields in your item class** and return `true` only if *all* those fields contain the same values in both objects.

```
public int hashCode()
```

- This method generates a unique integer (hash) for each object. Hash codes must not involve any randomness, and objects that are equal (as determined by the `equals` method) *must* have the same hash code. **You should use the `hashcode()` algorithm presented in lecture for this method.**

> **NOTE:** `hashCode()` is not actually used anywhere in this assignment. However, it is general convention to

implement `hashCode()` when you implement `equals()`.

```
public static Item parse(Scanner input)
```

- This method prompts the user for item attributes via the provided `Scanner`, then constructs and returns a new instance of the item class

Note that this method is `static`! Expand below to read more about why:

▼ Expand

If this `parse()` method were not static, then we would need to create an Item instance before calling it, which would lead to some extra, useless `Item` objects when used in our `Client.java` class! Making this method static allows us to call it without an instance of Item, instead directly use the class name. For example we can do:
`Item item = Item.parse(scanner);`

instead of:

`Item item = new Item().parse(scanner);`

Notice that in the non-static version a new item object is created but immediately discarded.

```
public int compareTo()
```

- Finally, your item class should implement the `Comparable` interface , and you should implement *a comparison algorithm of your choice*. **Your comparison must make use of *multiple* fields and/or methods of your class.**
- Note that your `compareTo()` method should be consistent with `equals()` — if two objects are considered equal, `compareTo()` should return 0 and vice versa.

2. Node Class

i For generalizability, we will be referring to the class you implement in this section as "Node". However, you should not name your class Node, and instead, should choose a class name that accurately reflects the items being stored.

You should define an inner node class *within* `CollectionManager.java` to represent your Item within a tree. Your class should store:

- A single Item field that represents the item.
- References to left and right child nodes

3. CollectionManager.java

⚠ **NOTE:** To earn a grade higher than N on this assignment, **your *core* algorithms for each method in this class must be implemented *recursively*. You will want to utilize the *public-private pair* technique discussed in class.** You are free to create any helper methods you like, but the core of your implementations must be recursive.

This class represents a full collection of Items, implemented as a binary search tree, and provides the required BST operations. Your `CollectionManager` class **must** implement the following methods:

```
public CollectionManager()
```

- Initializes an empty collection.

```
public CollectionManager(Scanner input)
```

- Loads a collection using the provided `Scanner`. Nodes should be added to the collection (and the underlying binary tree) in the same order in which they appear within the `Scanner`.
- The content of `input` should match the output content of the `save` method.
- You may assume that the format of `input` is always valid.

```
public void add(Item item)
```

- Adds the provided item to the collection.
  - This operation *must* maintain the binary search tree property.
- **Your implementation should take advantage of the ordering properties of the binary search tree and traverse as few nodes as possible.**

```
public boolean contains(Item item)
```

- Returns `true` if the collection contains the specified item, and `false` otherwise.
- **Your implementation should take advantage of the ordering properties of the binary search tree and traverse as few nodes as possible.**

```
public String toString()
```

- Returns a readable string representation of the collection. You can choose the format, but it should be **clear and include relevant details**.
  - This also means that the content of `toString` does not need to be the exact same as the `Scanner` constructor and `save` methods!
- The items in the collection must appear **in sorted order** (as defined by the `compareTo` method in your Item class) in the resulting string.

```
public void save(PrintStream output)
```

- Saves this current collection to the given `PrintStream`. The items in your collection should be printed to the file following a *pre-order traversal*.

- The content of the output file must be consistent with the content of the input file that can be passed to `CollectionManager(Scanner input)` constructor.

Creative Extension

To earn an **E** on this assignment, you must implement one of the following extensions in your `CollectionManager` class:

Filter

Write a method that returns a `List` of items in your collection that match a property of your choice. You get the decide the property! You may choose to filter by an exact match, a range, containing a keyword, or any other property or properties you like!

For example:

- In a Yarn collection, calling `filter("a", pink)` might return all yarns with names that start with "a" and that are pink.
- In a Sneaker collection, calling `filter(10)` might return all shoes of size 10.

You can define what parameters your filter method takes, but it must return a `List<Item>`. This method must be implemented *recursively*.

Remove

⚠ **NOTE:** This extension may be more complex than `filter`. Do this if you're up for a challenge!

Write a method that removes all items from your collection that match a condition of your choice. You get to decide the condition! You may choose to remove if an item is an exact match, within a range, contains a keyword, or any other condition you like! **The collection must preserve the BST property after each removal**, though the tree does *not* have to be balanced.

For example:

- In a Yarn collection, calling `remove(4)` might remove all yarns of weight 4
- In a Sneaker collection, `remove(2017, 2019)` might remove all shoes released between 2017 and 2019.

You can define what parameters your remove method takes. This method must be implemented *recursively*.

Custom Extension

If you would like, you may propose a different extension of your choice. Your proposed extension must be roughly similar in complexity to the above options. If you would like to propose a custom extension, you must post it in this Ed thread and receive approval. Requests for a custom extension must be made by **11:59pm on Tuesday, December 2** to allow enough time for review and approval before the deadline. (We will monitor the thread and approve on a rolling basis.)

Testing Requirement

Since this project is open-ended, **there are limited EdStem tests.** It is your responsibility to make sure your code compiles and behaves as expected. Additionally, we are asking you to provide the following:

- Three non-empty sample input files that can construct a tree using the `Scanner` constructor
    - In order for us to test that your `Scanner` constructor is consistent with your `save` method, your files should follow this naming convention:
        - `input1.txt`, `input2.txt`, `input3.txt`, etc...
        - You should **not** include any duplicate items in the input.
- A non-trivial test for *every* public method (including constructors) that you write in your `CollectionManager` class. These tests must be in a single file called `Testing.java`.
    - This means that you do **not** need to write tests for your Item class.

Using the Client File

We have provided a `Client.java` file for you to run your collection manager! Note that there are 3 TODOs indicated by `// TODO:` in the code for you to modify such that it runs your collection. Take a look at the `ClientExample.java` for an example of how these TODOs can be resolved.

Implementation Guidelines

For this assignment, you should follow the Code Quality guide when writing your code to ensure it is readable and maintainable. In particular, you should focus on the following requirements:

- **Node Design:**
    - The `compareTo()` method must establish a consistent sorting order.
- **Tree Behavior:**
    - The BST should remain valid (left < root < right).
- **`x = change(x)`:**
    - Similar to with linked lists, do not "morph" a node by directly modifying fields (especially when replacing an intermediary node with a leaf node or vice versa). Existing nodes can be rearranged in the tree, but adding a new value should always be done by creating and inserting a new node, not by modifying an existing one.
    - An important concept introduced in lecture was called `x = change(x)`. This idea is related to the proper design of recursive methods that manipulate the structure of a binary tree. **You should follow this pattern where necessary when modifying your trees**.

- **Avoid redundancy:**
  - If you find that multiple methods in your class do similar things, you should create helper method(s) to capture the common code. As long as all extra methods you create are private (so outside code cannot call them), you can have additional methods in your class beyond those specified here.
  - Look out for including additional base or recursive cases when writing recursive code. While multiple calls may be necessary, you should avoid having more cases than you need. Try to see if there are any redundant checks that can be combined!
- **Data Fields:**
  - Properly encapsulate your objects by making data fields in your `CollectionManager` class private. (Fields in your node class should be public following the pattern from class.)
  - Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used in one place.
  - Fields should always be initialized inside a constructor or method, never at declaration.
- **Commenting**
  - Each method should have a comment including all necessary information as described in the Commenting Guide. Comments should be written in your own words (i.e. not copied and pasted from this spec).
  - Make sure to avoid including *implementation details* in your comments. In particular, for your object class, a *client* should be able to understand how to use your object effectively by only reading your class and method comments, but your comments should maintain *abstraction* by avoiding implementation details.
  - Continuing with the previous point, keep in mind that the client should **not** be aware of what implementation strategy your class/methods utilize.

Assignment Deliverables Checklist

Here's a summary of what you need to implement for this assignment. The steps do not necessarily need to be completed in the order presented. See the spec for details of the expectations for each task.

1. Decide what type of item you would like to collect. Then create your own class from scratch for items of that type. This class is required to:
    - Override `.toString()`
    - Override `.equals()`
    - Override `.hashcode()`
    - Implement the `Comparable` interface, and therefore the `compareTo` method
    - Implement the `parse(Scanner input)` static method
2. Create an inner node class inside of `CollectionManager`.
    - This class must have the following fields:
        - A reference to your item class from task 1 above
        - A reference to a left node
        - A reference to a right node
    - It must also have any constructors necessary for your assignment
3. Complete the `CollectionManager` class, which is primarily a BST with some additional features. You need to do the following:
    - Implement a 0-argument constructor
    - Implement a `Scanner` constructor
    - Implement an `add` method, which inserts a given instance of your item class from task 1 above into the BST
    - Implement a `contains` method, which indicates whether or not a given instance of your item class is within the BST
    - Override `.toString`
    - implement a `save` method, which prints a pre-order traversal of the BST to the given `PrintStream`.
4. Creative extension: Implement a new BST method of your choice. You should implement one of the following:
    - `filter`
    - `remove`
    - Something else of your choice, so long as you receive approval in Ed.

Coding Workspace

Download Starter Code

C3_B(e)ST_of_the_B(e)ST.zip

Remember, in addition to implementing your item class and CollectionManager class, you must also satisfy the testing requirements listed below and update the client file. These are copied from the specification.

Testing Requirement

Since this project is open-ended, **there are limited EdStem tests.** It is your responsibility to make sure your code compiles and behaves as expected. Additionally, we are asking you to provide the following:

- Three non-empty sample input files that can construct a tree using the `Scanner` constructor
  - In order for us to test that your `Scanner` constructor is consistent with your `save` method, your files should follow this naming convention:
    - `input1.txt`, `input2.txt`, `input3.txt`, etc...
    - You should **not** include any duplicate items in the input.
- A non-trivial test for *every* public method (including constructors) that you write in your `CollectionManager` class. These tests must be in a single file called `Testing.java`.
  - This means that you do **not** need to write tests for your Item class.

Using the Client File

We have provided a `Client.java` file for you to run your collection manager! Note that there are 3 TODOs indicated by `// TODO:` in the code for you to modify such that it runs your collection. Take a look at the `ClientExample.java` for an example of how these TODOs can be resolved.

Reflection

Question 1

Your Item class represents a unique collection that you have chosen. Proper documentation is essential for ensuring that your class is understandable and maintainable.

1. Describe Your Item Class:

- What real-world collectible does your class represent? Why did you choose it?
- What attributes have you chosen to store? Why?
- What attributes did you decide to use for comparison? Why?

2. Formatting & Representation:

- How should a user expect to format input when using your `Scanner` constructor?
- What details are included in your `toString()` output? How do they contribute to readability?
- How does `save()` format your data for later retrieval?

For full credit, <u>all</u> questions must be clearly answered.

No response

Question 2

Java's `TreeSet` is an implementation of a Binary Search Tree that automatically keeps elements sorted and allows for fast lookups. But why does Java use a BST for this? And what are the trade-offs?

Step 1.

Before answering the reflection questions, explore Java's `TreeSet` documentation:
Java TreeSet Documentation

Step 2.

Write a response to **one** of the following prompts.

Prompt 1: Why do you think Java chooses a BST for `TreeSet` instead of an `ArrayList` or some other implementation? What advantages does this give when adding, removing, or searching for elements? Describe a situation where an `ArrayList` might actually be a better choice?

Prompt 2: When you add elements to a `TreeSet`, they are automatically sorted because of the underlying BST. What do you think "sorted order" means for different types of data? What happens if you add objects that don't have a natural ordering (like custom classes that you wrote, but without `Comparable`)? Provide a real-world example where automatic sorting could be useful.

Prompt 3: BSTs provide fast searching, inserting, and deleting—but only if they stay balanced. What happens if a BST becomes unbalanced (e.g., due to inserting elements in sorted order)? How do you think Java prevents this issue in `TreeSet`? *(Hint: It doesn't use a plain BST—look up "Red-Black Tree" if you're curious!).* Have you ever used a program where speed slowed down because of bad data organization?

Please label which prompt you are answering. For full credit, all questions within a prompt must be clearly answered. In particular, we suggest using the ACE (answer, cite, and explain) format. For a meaningful response, it may be helpful to pose some counterexamples, connect in terms of your own experience, add additional support from course materials, and be as specific as possible in your own reasoning

No response

Question 3

You've now designed and implemented a program that reflects something meaningful to you—a collection from your own interests, hobbies, or passions!

How did it feel to create something personally relevant using code? Looking ahead, what other programs might you build that connect to your real life, experiences, or interests? How do you think building meaningful programs can shape the way you approach coding in the future?

No response

Question 4

What skills did you learn and/or practice with working on this assignment?

No response

Question 5

What did you struggle with most on this assignment?

No response

Question 6

What questions do you still have about the concepts and skills you used in this assignment?

No response

Question 7

About how long (in hours) did you spend on this assignment? (Feel free to estimate, but try to be close.)

No response

Question 8

Was any part of the specification or requirements unclear? If so, which part(s), how was it unclear, and how could it have been made more clear?

No response

Question 9

[OPTIONAL] Do you have any other feedback, questions, or comments about this assignment?

(Note that we may not be able to respond to questions here, so please post on the message board if you would like a response!)

No response