TCSS 342 - Data Structures
Assignment 6 - Binary Search Tree

Version 1.1.2 (April 27, 2022)

Time Estimate: 6-8 hours

Learning Outcomes

The purpose of this assignment is to:

- Build a binary search tree data structure.
- Apply the binary search tree to a problem well suited to it.
- Gain experience testing and debugging data structures.
- Build data structures exactly to an API's specifications.
- Read a text file and extract the unique words.

Overview

A binary search tree is a common tool for storing data that we want to retrieve quickly. In this assignment we will build a naive binary search tree. We call it 'naive' because it will not try to keep itself balanced. Binary search trees are an upgrade to all lists because under most conditions they can maintain a logarithmic run time for all three primary functions: insert, delete, and search.

While you should test your data structure yourself, we will put the binary search tree to the task from Assignment 2 - Unique Words. So for this assignment we will upgrade the UniqueWords class.

To complete this you will upgrade one class and implement one public class and one private class:

- MyBinarySearchTree<Type> (with a private Node)
- UniqueWords

Formal Specifications

`MyBinarySearchTree<Type extends Comparable<Type>>`

`-root : Node`
`-size : int`
`+comparisons : long`

`+add(item : Type)`
`-add(item : Type, subTree : Node) : Node`

```
+remove(item : Type)
-remove(item : Type, subTree : Node) : Node
+find(item : Type) : Type
-find(item : Type, subTree : Node) : Type
+height() : int
+size() : int
+isEmpty() : boolean
-updateHeight(node : Node)
+toString() : String
```

Field summary

- `root` - Stores the root node of the binary search tree.
- `size` - Stores the number of items stored in the binary search tree.
- `comparisons` - Stores the number of comparisons made in the find method (one per recursive call).

Method summary

- `add` - Adds the `item` into the binary search tree where it belongs. The public method should call the private recursive method on the `root`. The private method adds the `item` to the sub-tree (recursively) and returns the root of the new sub-tree. This method should run in O(d) time where d is the depth the `item` added.
- `remove` - Removes the `item` from the binary search tree. The public method should call the private recursive method on the `root`. The private method removes the `item` from the sub-tree (recursively) and returns the root of the new sub-tree. This method should run in O(d) time where d is the depth of the `item` removed.
- `find` - Returns the item found if the `item` is in the binary search tree and null otherwise. The public method should call the private recursive method on the `root`. The private method searches the appropriate sub-tree recursively for the `item`.This method should run in O(d) time where d is the depth of the `item` found.
- `height` - Returns the height of the binary search tree. This method should run in O(1) time.
- `size` - Returns the number of elements in the binary search tree. This method should run in O(1) time.
- `isEmpty` - Returns true if the trie is empty and false otherwise. This method should run in O(1) time.

- `updateHeight` - Updates the height of the node. This method should run in O(1) time.
- `toString` - Returns the contents of the binary search tree, in ascending order, as a string. This method should run in O(n) time where n is the number of items stored in the trie.

Node
+item : Type +left : Node +right : Node +height : int
+toString() : String

Method summary

- `item` - The item stored in the node.
- `left` - The left subtree.
- `right` - The right subtree.
- `height` - The height of the node (the distance to the leaf nodes). We will count edges so leaves have height 0.

Method summary

- `toString` - Returns the contents of the node in this format:

<item>:H<height>

UniqueWords
…
… +addUniqueWordsToBST()

Method summary

- `addUniqueWordsToBST` - Adds the unique words to a MyBinarySearchTree.
    a. For each word in the list stored in book:
        - Check to see if the word is stored in the binary search tree of unique words.
            - If it is not, add it to the binary search tree.
            - Otherwise, skip this word.
    b. Calls toString from the binary search tree to extract the words in order.

This method should time both steps and output each runtime to the console as well as the number of unique words, height of the binary search tree and the number of comparisons made. This method should run in time O(nlogn) where n is the number of words in the book.

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is the output from my solution:

```
Reading input file "./WarAndPeace.txt"... 3291642 characters in 83 milliseconds.

Finding words and adding them to a linked list...  in 336 milliseconds.
The linked list has a length of 570240.

Adding unique words to a binary search tree...  in 184 milliseconds.
20228 unique words
The binary search tree had a height of 40 and made 6914511 comparisons.
Traversing the binary search tree...  in 33 milliseconds.
```

Submission

You will submit a .zip file containing:

- MyBinarySeachTree.java
- MyLinkedList.java
- BookReader.java
- UniqueWords.java

Grading Rubric

In order to count as complete your submission must:

1. Implement the binary search tree efficiently and without errors.
2. Follow the API exactly so that your classes will work with any testing code.
3. Pass all tests in the testing code. If you fail any tests, you will be told which test failed.
4. Get the numbers right:
    a. Find all 570240 words in the book.
    b. Find the 20228 unique words in the book.
    c. The binary search tree should have depth 40.
    d. Make 6914511 comparisons to find the words in the binary search tree.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.