CS 171: Introduction to Computer Science II
Assignment #5: Binary Search Tree and HashTable

Due: Monday May 3 at 11:59 PM EDT [Late submission: see syllabus for policy].

Part 1: BST for Movie Search [60 points]

Congratulations! You've been hired by IMDB to implement a Binary Search Tree (BST) based index for indexing/searching movies efficiently in their IMDB dataset.

Requirements: Our goal is to quickly find the information associated with movies, sometimes based on a movie name (e.g. Gangster Squad), in other times based on the *prefix* of a name (e.g. Gangster*). In terms of key-value notations, your search **key** is the short or simplified title of a movie, and the associated **value** or data is the entire object of type `MovieInfo`, which contains more information on the movie and is defined in the following class:

```
class MovieInfo {
  public String shortTitle; //simplified movie title, e.g. "Gangster Squad"
  public String fullTitle; //full title including year, e.g. "Gangster Squad (2012)"
  public int ID;           //integer ID
}
```

The BST index is implemented as a stand-alone class `BSTIndex`, with an inner class `Node`. Each node within a BSTIndex tree contains 4 fields: key (of type String), data (of type MovieInfo), and left and right links to the children nodes. The BST index tree should never contain duplicate nodes; i.e. each node *key* needs to be unique across the tree.

You will complete the implementation of the following methods and operations in class `BSTIndex`. All BST operations should be implemented using recursion.

- **[15 points]** `public int calcHeight()`
  This method should calculate and return the *height* of the current BSTIndex tree. We define the height of a tree (similar to our lecture slides) as the maximum depth in the tree plus 1. The height of a tree with only one node is 1, and the height of an empty tree is 0.

  Important: Notice that this is a public method that calls another *private* method *calcNodeHeight(Node t)* and passes it a reference to the root. This is a common way of structuring BST methods such that external client code will not have direct access to the tree root. You will thus be implementing the code in the private `calcNodeHeight(Node t)` method. This applies to the rest of the methods in this assignment as well.

- **[15 points]** `public void insertMovie (MovieInfo data)`
  This method should insert the given data element into the proper place in the BST structure. If the input file turned out to have duplicate keys (i.e. repeated short movie titles), then your insertion method should detect that and *not* add the new duplicated movie to the tree.

- **[15 points]** `public MovieInfo findMovie (String key)`
  This method should return the data element (i.e. the MovieInfo object) of the node where the movie's `shortTitle` matches the given key. Return `null` if the movie is not found.

- **[15 points]** `public void printMoviesPrefix (String prefixStr)`
  This method should print out all movies in the tree whose `shortTitle` begins with the passed prefix string. If no movies match the given prefix, nothing will be printed. The order of printing

does not matter, but make sure to print each match in a separate line.

The IndexTester class is provided which will test your BSTIndex class. The IndexTester creates an empty BSTIndex object then reads the data from an input movie file, builds a MovieInfo object for each row, and inserts it into the BST index (using the insert method you implemented in BSTIndex class). At this point, the BST index will be built for all the movie entries in the file. Then, the program will ask the user to enter a string. The code for parsing the input string and deciding which operation to invoke is part of the starter code.

Here's an example of how a program run might look like. Note that the data file name is passed as an argument to the program's main() method here using the command line:

```
> java IndexTester data/movies100K.txt
Inserted 10000 records.
Inserted 20000 records.
Inserted 30000 records.
Inserted 40000 records.
Inserted 50000 records.
Inserted 60000 records.
Inserted 70000 records.
Inserted 80000 records.
Inserted 90000 records.
Inserted 100000 records.
Index building complete. Inserted 100000 records. Elapsed time =  <your_result_
here> seconds.

Welcome to the IMDB movie search engine!
- Enter search string to find a specific movie.
- Enter search string ending with '*' to print all movies that match this prefix
- Enter 'h' to print tree height.
- Enter 'q' to quit.
Enter Option: h
Height of BSTIndex tree = <your_result_here>
Enter Option: Gang
Movie Gang Not Found
Enter Option: Gang*
[Finding all movies that start with Gang..]
Gang tai Gang
Gangster Squad
Gang Wars
Gangotri
...
<rest_of_matches_here>
Enter Option: q
```

Getting started:

1. Download and understand the starter code IndexTester.java, MovieInfo.java, and BSTIndex.java.
2. The input data is available in the data folder on Canvas. Note that the full data file (movies.txt) is relatively large. You can copy the smaller extract (movies100K.txt) for debugging. They are all in the same format: one movie per line, with fields ID, shortTitle, and fullTitle, separated by a tab character '\t'. Read IndexTester code to remind yourself how to read this data.

3. Implement all missing code in `BSTIndex.java`. This is the only file you are required to submit on Canvas for Part 1 of this assignment.

4. Test your program with IndexTester on the available datasets, starting from the smaller one.

Part 2: HashTable for Word Search [40 points]

In this part of the assignment you will complete the implementation of a hash table with separate chaining. Then, you will test the hash table by completing a simple program that reads a text file and generates a *word frequency* table for the file.

Requirements:

1. **[30 points]** Make the size of the hash table's array dynamic.

   The starter code in the file `HashSeparateChaining.java` contains a functional implementation of a hash table where collisions are resolved by separate chaining. Each key-value pair in the table is contained in an `Entry` object, defined in the `Entry.java` file and has the following properties:

   - The key is a `String`.
   - The value is an `Integer`.

   However, the size of the array that is used to store the entries is static (fixed) in the starter code. From lecture, we know that this is not ideal because as more and more entries are added to the table, the average length of the chains will become longer and longer. Long chains destroy the efficiency of searching in the hash table, so we would like to avoid this by resizing the hash table's array when appropriate.

   Your first task is to implement the `resize` method. You must also modify the `put` and `delete` methods to call the resize method at the approprate times! Read through each method carefully before you make the modifications so that you understand exactly what it is doing and how it is doing it. Your implementation should result in an average length of the chains in the hash table between 3 and 5 nodes at all times (similar to our lecture discussion).

2. **[10 points]** Complete the method `getMap` in class `HashTableApp.java`. The main method of this program will take in one parameter, the name of a text file. It will read in the contents of the text file one word at a time and maintains an entry in the hash table for each word where:

   - The key is the word from the text.
   - The value is the number of times that the word has appeared in the text so far.

   To test your program, you should start by generating a few small text files where you can directly control the frequencies. The A5 folder has a small test file `hamlet.txt` and a large file `WarAndPeace.txt`, which contains the full text of the novel War and Peace.

Tip: You may want to start by implementing `getMap` in class `HashTableApp.java` and testing the hash table on a small text file **without** resizing. Then, implement `resize` in class `HashSeparateChaining.java` and make sure `put` and `delete` are calling it at proper times. Now, re-run the tester class `HashTableApp.java` and observe the changes to the hash table's size and chain lengths.

Note: There is a `toString` method included in `HashSeparateChaining` that will create a text representation of the hash table. You can use this to print out your hash table and look at its structure. The output will look bad if the chains get too long ;-).

Implement all missing code in `BSTIndex.java`. This is the only file you are required to submit on Canvas for Part 1 of this assignment.

Submission: For this part of the assignment you need to submit both `HashSeparateChaining.java` and `HashTableApp.java` after completing all required modifications.

Grading:

- Correctness and robustness of the hash table: your hash table resizes correctly and at the right times to maintain the average length of chains, the entries are never lost (i.e. If a key is added to a table, it should be searchable until it is removed with the `delete` method.) (30pts)
- Correctness and robustness of the application: the hash table should have the correct frequencies for the words in the text file read by the application, and should never crash. (10pts)

Honor Code

The assignment is governed by the College Honor Code and Departmental Policy. Remember, any code you submit must be your own; otherwise you risk being investigated by the Honor Council and facing the consequences of that. We do check for code plagiarism (across student groups and against online resources). Please remember to have the following comment included at the top of the file.

```
/*
THIS CODE WAS MY OWN WORK, IT WAS WRITTEN WITHOUT CONSULTING
CODE WRITTEN BY OTHER STUDENTS OR COPIED FROM ONLINE RESOURCES.
 _Student_Name_Here_
*/
```