Homework 8

GWU CSCI 1112 - Fall 2019

December 3, 2019

1 Introduction

This homework is intended to reinforce your understanding of working with maps in general. Your task is to implement and test both a treemap and a hashmap and to compare the performance of the two data structures using a large set of data.

1.1 Deadline

December 9, 2019 at 11:59pm

1.2 Submission

You must create a `.zip` file containing all of the files that you develop and work with and you must submit only the `.zip` file to blackboard before the deadline.

Your `.zip` file must be named using the following naming convention: `<yourNetId>-hw-08.zip`
For example, my NetId is `jrt`. If I submitted the homework, I would name my `.zip` file: `jrt-hw-08.zip`

Do not submit the compiled `.class` files. You must submit any `.java` files that support the application.

1.3 Grading Rubric

- 30% For successful compilation
- 15% For sufficient comments (file headers, function headers, line comments)
- 10% For consistent coding style
- 20% For base implementation
- 10% For extension implementations
- 15% For unit testing

1.4 Multiple Programs

Like the previous homework, you will develop a number of programs with different roles. You will develop independent unit tests for each map data structure in the base implementation. You will also develop an application for the extension that uses both of the map data structures.

1.5 Unit Testing

In this homework, you will take responsibility for unit testing your data structure and you must implement your own unit test file modeled on those provided in the previous homeworks. Your unit tests will be evaluated in terms of comprehensiveness and correctness. We are also withholding a set of unit tests that will be used to independently evaluate your implementation.

Your class implementations must therefore fulfill the prescribed interfaces so that our unit tests can use your classes. You may add additional functions as needed; however, for each new function you add, you should provide a unit test and evaluate it in your test application. You must unit test all of the prescribed interfaces.

You are explicitly required to unit test the interfaces to the `BinaryTree` class and the `HashTable` class. You do not need to unit test the application that you will develop for the extension.

1.6 Comments

You must provide a file header for any files that you submit that documents the author and what is defined in the file. You must also document all functions that you implement with a function block header which must include information on what the function does, what the inputs to the function represent, and what the function returns. Finally, you must also provide relevant line comments inside any functions.

1.7 Plagarism

We will use a set of automated tools specifically designed to analyze code for plagarism. If you copy code from another source, classmate or website, there is a very high probability that these tools will flag your work as plagarized. You are permitted to discuss the problems at a high level; however, you must code your own solution. If you do not share code or outright borrow code from a website, you will have no problem with the plagarism filter.

2 Maps

As we discussing during class, a map is an abstract data type that associates, *i.e.* maps, keys to values. A map allows programmers to insert and search for values in the data structure quickly by key. The key can also be a much more complex representation that is defined by the data in the value object rather than a simple, arbitrary integer value. To give a little more context, let's consider two examples: the english dictionary and the Dewey Decimal Classification.

A dictionary such as the english dictionary is a map where the key is an english word and the value is the definition(s) of the word. Word definitions are actually complex, multi-value data records that might include enumerated definitions subject to usage, phonetic information, and possibly historical context. Our ability to efficiently look up a definition is facilitated by the ordering of the dictionary by key. The key (word) dictates where in the dictionary the value (definition) is found. One of the inefficiencies of a physical dictionary (a book) is that it is a flat file that requires a bit of searching in order to locate the key. Granted, because a dictionary is ordered by key we can apply novel strategies to minimize the number of comparisons needed to find a key and its associated value; however, there are other organizational approaches that we might use to locate a value given a key.

Libraries traditionally index books within their collections using a proprietary organizational model called Dewey Decimal Classification. This classification system associates a key, *i.e.* an index consisting of numbers and letters, with a book to dictate where in a library that book is stored. This system basically defines how to generate a key in order to prioritize library organization by topic and attempts to ensure that books with similar subject matter are more tightly clustered within the library over a more simple organizational model such as author and/or title. Consider how difficult it would be to use a library to cross-reference computer science topics if books by Eberly and Sipser were stored on different floors or even different buildings rather than within several shelves of one another. The key generated by the Dewey classification directs library users directly to a floor, aisle, and shelf within the library and allows library users to quickly filter out thousands or even millions of non-relevant books from a search.

Both of these organizational frameworks represent maps that allow quick access to information. Programmers need to be able to apply similar systems to ensure that search efficiency is minimized. We have also noted that there is close relationship between search efficiency and sorting where our most efficient search algorithm in "flat" data, *i.e.* binary search, requires an expensive sort operation to impose the organization necessary to gain the advantage of a fast search. To make our systems as efficient as possible, we need to reduce the cost of insertion and maintain order so that the combination of sort and search is efficient. In order to achieve this requirement, we have looked at binary trees and hash tables.

As we noted in class, the binary tree is a hierarchical data structure whose organizational structure produces the same search efficiency as binary search as long as the tree is balanced. The expensive aspect of maintaining the binary tree is actually due to rebalancing the tree which is performed in response to insertion. If rebalancing is as efficient as optimal quicksort, then a binary tree is no less efficient than a combination of quicksort and binary search and if rebalancing is more efficient than optimal quicksort then we can use a binary search approach that is more efficient overall than quicksort.

We also looked at hash tables and noticed that insertion into the hash table occurs in constant time; however, the efficiency of search in a hash table is subject to a number of factors, primarily the bias of the hashing algorithm and the ratio between the number of elements stored to the number of buckets. If the hash algorithm results in the same index being selected, then we are effectively storing data in a single linked-list and our search efficiency will be no better than searching a linked-list, and even if the hash algorithm is unbiased, if the ratio of elements to buckets is large then every bucket contains a lengthy linked-list.

In this homework, we will implement maps using a binary tree and a hash table. We will then run a few experiments involving the same data in a binary tree and a set of hash tables with varying number of buckets to examine the efficiency of search in these data structures.

2.1 Framework

You will note that a number of classes have been provided as a framework. One change from previous homeworks that you should note is that there is only a single folder provided. For the base implementation, you will focus on developing `BinaryTree.java`, `BinaryTreeUnitTests.java`, `HashTable.java`, and `HashTableUnitTests.java`. Your binary tree will use the class provided in `TreeNode.java` to maintain elements in the tree, and your hash table will use the class provided in `ListNode.java` to maintain elements in a bucket. For the extension, you will use the classes that you developed in the base implementation in `Extension.java` to compare the performance of these data structures and you will use the helper classes provided in `Person.java` and `PersonLoader.java` to load and interact with the data provided in the `people.txt` data file.

For the base implementation, you must implement and test both a `BinaryTree` class and a `HashTable` class. Each of these classes will be considered to contribute half of the credit for the base implementation.

For the extension implementation, you will focus on implementing the application in `Extension.java` that is described below.

2.2 Interfaces

Below, you should note that the functions that we are implementing have the same signatures in both the `BinaryTree` and `HashTable` classes. We touched on the idea that we could use a java interface to build an inheritence hierarchy. We have already discussed how both the binary tree and hash table are both maps, so it make sense that both classes could inherit a single interface. Our extension implementation could benefit from the commonality shared between these two classes and help simplify the logic; however, inheritence is beyond the scope of this course, and because it would raise significant questions that we cannot address in the remainder of the course, we have opted not to use a java interface. We recommend that you reconsider this particular assignment after you have had sufficient instruction in inheritence and polymorphism as you should see the commonality between these classes allows for code reduction and simplification. In essence, we could develop one single unit test that could be used on both classes because they

would have the same interface which suggests that **you can reuse the same unit tests developed for one map with another map with very minor changes**.

While it may appear that the functions you need to implement are somewhat complex, they are actually rather simple and you have already applied much of the knowledge necessary in earlier assignments. For example, after determining which bucket in the hash table you are operating on, searching a bucket is basically traversing a linked list. These algorithms can also be implemented using either an iterative or recursive approach where the recursive approaches will require less code but will be difficult to debug and the iterative approaches will likely demand more complex branches that can hide bugs if you are not careful. So focus on the concepts and do not overthink the problems as you have done most of this before.

2.2.1 The `profile` field

As in some earlier assignments, the `search` functions accept a `profile` integer array as a parameter. The `profile` array must consist of at least one field (index 0) in which the number of comparisons made during search is recorded. You must pass a `profile` array dedicated to the specific search function when it is called and inside that search function increment the comparisons field each time a comparison is performed inside the search function. The `profile` field will be used in the extension in order for us to quantify and assess the cost of search in these data structures. This type of profiling is purely an academic exercise designed so that you may quantify the overall number of comparisons made inside the search operations. Search operations would not normally require this field in the interface.

2.3 Base - Binary Tree

You must implement the following `BinaryTree` class members using the following function signatures:

```
public boolean insert(String key, int value)
public int search(String key, int[] profile)
```

The `BinaryTree` class also provides a number of helper functions for rebalancing the tree that you will not need to implement; however, **you will need to explicitly call `balanceTree` after each insertion**. The rebalancing algorithm is brute force and could be made more efficient by checking the depth against the number of elements to determine whether balancing is necessary; however, this would require you to track depth as well and we did not want to overcomplicate the implementation of `insert`. As a result insertion will be somewhat slow, so you might consider using a print statement when calling `insert` (not in `insert` itself) to gain confidence that the tree is performing inserts if your program does not appear respond.

2.3.1 Base Functions

This section gives a general explanation of the requirements for each function that must be implemented to satisfy the base requirements.

```
boolean insert(String key, int value)
```

`insert` takes a `key` and a `value` as input and returns a truth value indicating whether or not the insertion operation was successfully performed. `insert` is successful if it either inserts the key-value pair into the binary tree subject to the `key` if the `key` does not exist in the tree or updates the associated `value` for the given `key` if the `key` already exists in the tree. If the `key` is either `null` or the empty string, the key is invalid and `insert` fails to insert the value into the tree. If the key-value pair is successfully inserted and before returning from `insert`, you must rebalance the tree to ensure that the search efficiency is optimal.

```
int search(String key, int[] profile)
```

`search` returns the `value` stored in the tree for the associated `key` if the `key` is valid and is found; otherwise, `search` returns `-1`. You can minimize the comparisons for a single node by making the comparison once, storing the result, and then branching on the stored result. For information on the `profile` parameter, see Section 2.2.1.

2.4 Base - Hash Table

You must implement the following `HashTable` class members using the following function signatures:

```
public boolean insert(String key, int value)
public int search(String key, int[] profile)
```

The `HashTable` class also provides its own helper function for hashing a String to an integer that you will not need to implement; however, **you will need to explicitly call `hash` whenever you need to locate a bucket given a key**. The hash function simply uses the built in hash operation in the java `String` class combined with a modulo operation.

2.4.1 Base Functions

This section gives a general explanation of the requirements for each function that must be implemented to satisfy the base requirements.

```
boolean insert(String key, int value)
```

`insert` takes a `key` and a `value` as input and returns a truth value indicating whether or not the insertion operation was successfully performed. `insert` is successful if it either inserts the key-value pair into the hash table subject to the `key` if the `key` does not exist in the hash table or updates the associated `value` for the given `key` if the `key` already exists in the hash table. If the `key` is either `null` or the empty string, the key is invalid and `insert` fails to insert the value into the hash table.

```
int search(String key, int[] profile)
```

`search` returns the `value` stored in the tree for the associated `key` if the `key` is valid and is found; otherwise, `search` returns `-1`. Consider the hash operation to account for one comparison and any subsequent comparison of an element in a linked-list bucket to also account for another comparison. For information on the `profile` parameter, see Section 2.2.1.

2.5 Extension

This section details the requirements for the extension.

For the extension, you will create multiple data structures and load those data structures with personal information derived from the data stored in the `person.txt` data set. There are roughly 85,000 records in the person data set, so this exercise is intended to examine the performance of search using a binary tree and a hash table with a reasonably large data set.

You will note that in the `Extension.java` `main` function that the data is loaded for you into an array containing `Person` instances. You will insert this information into an instance of a binary tree and several instances of a hash map with varying numbers of buckets. The key that you will insert will be the person's name and the associated value

that you will insert will be the person's age which can be accessed from the `Person` class.

Create a binary tree, a hash table consisting of 1,000 buckets, a hash table consisting of 20,000 buckets, a hash table consisting of 100,000 buckets, and a hash table consisting of 100,003 buckets. Insert all of the peoples' information into each of these data structures. Once you have inserted the data, search for the following keys in each of the data structures and report the number of comparisons that are made when searching in each data structure as recorded by the associated `profile` argument. Search for these keys:

```
GERALD WORKMAN
JAMES TAYLOR
JEFFERY DORSEY
```

In addition to your code, submit a short `pdf` document that summarizes the number of comparisons for each of the above keys searched on in each data structure. Explain why each data structure exhibits the number of comparisons it does for each key and compare the performance between hash table and the binary tree. Specifically, identify which hash tables exhibit worse, equivalent, or better search performance than the binary tree. Also, explain what the relationship is between the number of buckets and the efficiency for the hash tables. Finally, make an argument as to why we might choose 100,003 buckets.