YESHIVA COLLEGE
COMPUTER SCIENCE

Document Store and Search Engine.

Semester Project for Data Structures (COM 1320), Spring 2023

Table of Contents

How to Use This Document

1) The first time you are reading this, you should:
    a. make sure there are no distractions – turn your phone off, turn off all notifications on your laptop.
    b. read everything that comes before “Stage #1”, as well as the “Stage #1” section. As you read, take notes on any questions you have.
    c. If you have questions about any of the technologies or libraries this project requires, follow the links in this document to read more about them
    d. If after reading up on them you still have questions, feel free to post questions to Piazza
2) As you work on the project, be sure to refer back to this document to double check requirements as you go.
3) If new versions of the document are created in response to Piazza questions etc., make sure to download the updated version and use that as your reference.

Introduction to the Project

Over the course of the semester, you will build a very simple search engine. The project is split into multiple stages, each stage due at a different point in the semester. Each stage depends on the successful completion of the stages that came before it, so do not fall behind!

In addition to the primary focus, i.e. understanding and using data structures, the project also will teach some basic software engineering skills.

Learning Goals

1. Implement your first data structures, thus beginning down the path of more sophisticated design and software engineering.
2. Implement a general-purpose class and then using it for a specific application. This is a small step from a code perspective, but a giant leap in terms of how you design your code. This is you first foray into modular design.
3. Experience with, and competency in, professional software engineering skills and tools:
    a. Create your first Maven project for building your system and for dependency management. Real-world software is compiled and packaged using build systems (maven, gradle, esbuild, etc.), not by manually invoking a compiler at the command line.
4. Reuse an open source project – details to come as part of your project.
5. Build your first real piece of software - a search engine.
6. Teach yourself, and use, more advanced features of the Java language.

Important Advice on Looking for Answers

Before you google to find an answer to a technical question/issue, **first** look in the official documentation and the assigned book. There is a lot of good information on sites like StackOverflow, etc., **but there is also a lot of garbage on it**. Official documentation and books recommended by your professors, however, is high quality content.

Specifically:

- If it's a question about using an IDE of your choice, start by looking in the IDE's "Help" menu and read the relevant part of the IDE's documentation, and/or official online tutorials and documentation from the company that makes the IDE
- If it's a question about a given open source library or tool (e.g. Maven, GSON), the first place to look is the documentation and JavaDocs of those projects
- If it's a question about Java itself, look in the official JavaDoc and/or the book

Important Advice Getting Credit for Your Work

1. **Code that does not compile will get a zero.**
2. Make sure your laptop has Oracle JDK 17, not any other version of Java, since that is the version I will use when grading
3. Before you submit code for this project (or any other computer science homework for any class) TEST YOUR CODE ON A MACHINE OTHER THAN YOUR OWN LAPTOP. This will help avoid issues where a difference between your machine configuration and the professor's causing you to lose all or partial credit.
    a. The easiest way to do this is to install a virtual machine on your laptop, by following these steps:
        i. Download VirtualBox and create a Linux (Ubuntu) virtual machine. Follow these instructions, but note that there are newer version of VirtualBox and Ubuntu than those mentioned in the video.
        ii. Install Oracle JDK 17, Maven, and Git, AND NOTHING ELSE, inside that virtual machine if they are not there already.
        iii. `git clone` your project from your YU github.co repository into that virtual machine, and run it there at the command line. If it works there, you are likely in good shape to be graded without losing credit because of configuration issues.

Requirements for Your GitHub Repository Structure and Project Submissions

You must layout the folders of your YU GitHub repository as described below. Failure to do so may result in you getting a zero when our build and tests scripts can't find your code where it expects it to be.

1. Under the root of your repository there MUST be a directory named "DataStructures", NO SPACES.

   a. For example, if your repository name is "UncleMoishe", then the following directory must exist: `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures`
2. Under DataStructures, you will create a subdirectory called "project", like this: `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project`
3. Under project, you will create separate folders for each stage of the project. So, assuming we have 5 stages in this project, you will create the following subdirectories:
   a. `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage1`
   b. `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage2`
   c. `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage3`
   d. `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage4`
   e. `https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage5`
4. When you check your code in for each stage, you will check in the pom.xml file and the java code in the standard maven project layout. That means that under each "stage" directory listed above, the following will exist:
   a. Your Maven build file - `pom.xml`
   b. Your main code under - `src/main/java/`
   c. Your JUnit test code under - `src/test/java/`
5. As you know from intro, the directory structure a Java file is stored in must correspond to the name of the package the Java class is declared as being part of. The root of that directory structure are the two `java/`directories listed above below `main` and `test`. In other words, a class called `Hello` in the package `edu.yu.cs` for stage 1 would be stored, in `DataStructures/project/stage1/src/main/java/edu/yu/cs/Hello.java` in your GitHub repository.
6. Even when/if I do not require any specific tests as part of the project, **I urge you** to create unit tests to test your code and check them in. This will both help you do a better job, and also make it more likely that you get credit for parts that work even if you don't get a given stage 100% correct.
7. **Each stage of the project must compile and run entirely independently of the other stages, as its own independent maven project with no dependencies at all on other stages. If there are interfaces or classes that you will use in multiple stages, copy them into the project/directory of each stage.**

General Points About Your Code

The following rules apply to all stages of the semester project:

1. **You may not use any static methods** in your code besides `public static void main`. Every other method must be an instance method. You are writing object oriented code, not old-fashioned procedural code.
2. **Your code may not have any "monster" methods;** no method in your code may be longer than 30 lines (not counting comments.) Get used to breaking logic down into smaller chunks, i.e. methods that you call from within another method.
3. **You must use Maven** for building your code and for dependency declaration and resolution. Do not manually download JAR files for any opensource libraries we are using, and do not manually add them to your classpath. Only declare them as a Maven dependency, and Maven will download them for you.
4. **No other libraries: you may not use any libraries other than the JDK and whatever other libraries I explicitly specify that you should use.**
5. **You MUST submit JUnit test code for each stage. I urge you to write a separate unit test for every requirement of every stage to prove to yourself that you have met the requirement. You can have multiple tests in one test class, but they should function entirely independent of one another.**

It would be wise to print out the relevant pages of this document when working on a stage of the project, and check off each requirement after you have implemented and successfully tested it.

Stage 1: Build an In-Memory Document Store (`HashTable`) – Due Mar. 3 2:00pm

Description

In stage 1 you are building a very simple storage mechanism which supports "getting" and "putting" data. Documents are stored in memory in a hash table and can only be retrieved using the key (URI) with which they are stored. Documents can be plain text, a.k.a. a String, or binary data, e.g. images.

Logic Requirements

1. *Implement a `HashTable` from Scratch*

1. Implement a hash table which uses separate chaining to deal with collisions.
2. For stage 1, the size of the array that underlies your hash table is fixed in size and does not grow or shrink at all. The length of the array must be 5.
3. You may not use any Java collection classes or any other data structures (besides an array) provided in the JDK, rather you must implement the hash table yourself.
4. Your hash table class must be a general purpose (generic) class, i.e. not be limited to specific key-value types.
5. It must implement the `edu.yu.cs.com1320.project.HashTable` interface that has been provided to you.
6. It must be called `edu.yu.cs.com1320.project.impl.HashTableImpl`

2. *Implement the Document Interface*

1. A Document is made up of a unique identifier (`java.net.URI`) and the data that comprises the content of the document, either a java.lang.String for text or a byte[] for binary data.
2. You must write a class called `edu.yu.cs.com1320.project.stage1.impl.DocumentImpl` that implements the `edu.yu.cs.com1320.project.stage1.Document` interface.
3. `DocumentImpl` **MUST NOT** implement `java.lang.Comparable`
4. The hashCode for a `Document` is calculated as follows:
   a.
   ```
   @Override
   public int hashCode() {
       int result = uri.hashCode();
       result = 31 * result + (text != null ? text.hashCode() : 0);
       result = 31 * result + Arrays.hashCode(binaryData);
       return result;
   }
   ```
5. `DocumentImpl` must override the default `equals` and `hashCode` methods. Two documents are considered equal if they have the same hashCode.
6. `DocumentImpl` must provide the following two constructors, which should throw an `java.lang.IllegalArgumentException` if either argument is null or empty/blank:
   a. `public DocumentImpl(URI uri, String txt)`
   b. `public DocumentImpl(URI uri, byte[] binaryData)`

3. *Implement a Document Store*

1. Implement the `edu.yu.cs.com1320.project.stage1.DocumentStore` interface, which specifies an API to:
   a. put a `Document` into the store
   b. get a `Document` from the store
   c. delete a document
2. It must be called `edu.yu.cs.com1320.project.stage1.impl.DocumentStoreImpl`
3. Your document store must use an instance of `HashTableImpl` to store documents in memory.

4. If a user calls `delete,` or `put` with `null` as the value, you must completely remove any/all vestiges of the `Document` with the given `URI` from your hash table, including any objects created to house it. In other words, it should no longer exist anywhere in the array, or chained lists, of your hash table.
5. Your code will receive documents as an `InputStream` and the document's key as an instance of `URI`. When a document is added to your document store, you must do the following:
    a. Read the entire contents of the document from the `InputStream` into a `byte[]`
    b. Create an instance of `DocumentImpl` with the `URI` and the String or `byte[]` that was passed to you.
    c. Insert the `Document` object into the hash table with `URI` as the key and the `Document` object as the value
    d. Return the hashCode of the previous document that was stored in the hashTable at that `URI`, or zero if there was none

4. Other requirements on your implementation

1. You must fully implement the interfaces defined in the code provided to you under the stage1 folder. Other than constructors, you may not add any public or protected methods in your implementations of these interfaces. **Any additional methods you add must be private.**
2. The name of your implementation classes should be the name of the interface with "Impl" (short for "implementation") added to the end, and be located in a sub-package called "impl". That means that you must submit the following classes in your solution:
    a. `edu.yu.cs.com1320.project.stage1.impl.DocumentImpl`
    b. `edu.yu.cs.com1320.project.stage1.impl.DocumentStoreImpl`
    c. `edu.yu.cs.com1320.project.impl.HashTableImpl`
3. Make sure the name of your `impl` **package** starts with a lower case "i", not an upper case "I". The "I" in your class names, on the other hand, e.g. `DocumentImpl`, must be upper case.
4. DO NOT move the interfaces to a different package – they must remain in the packages in which they were placed in the code you are given.
5. `HashTableImpl` and `DocumentStoreImpl` must both have no-argument constructors.

Stage 2: Add Undo Support to the Document Store Using a **Stack**. Using Functional Programing in Java. **Due Mar. 22, 11:59pm**

Description

In this stage you add support for two different types of undo 1) undo the last action, no matter what document it was done to 2) undo the last action on a specific `Document`.

You will also get your first coding experience with functional programming in Java.

Logic Requirements

1. Update `HashTableImpl`

1. Implement array doubling on the array used in your `HashTableImpl` to support unlimited entries. Don't forget to re-hash all your entries after doubling the array!
2. Add a no-arguments constructor to `HashTableImpl` if you didn't already have one.

2. Add Undo Support via a Command Stack

1. Every call to `DocumentStore.put` and `DocumentStore.delete` must result in the adding of a new instance of `edu.yu.cs.com1320.project.Command` to a single `Stack` which serves as your **command stack**
    a. No other class besides your document store may have any access/references to the command stack; it must be a private field encapsulated within `DocumentStoreImpl`
    b. You must use the `Command` class given to you to model commands. You may not alter in any way, or subclass, `Command`.
    c. You must write a class called `edu.yu.cs.com1320.project.impl.StackImpl` which is found in its own .java file and implements the interface provided to you called `edu.yu.cs.com1320.project.Stack`, and your command stack must be an instance of `StackImpl`.
    d. `StackImpl` must have a constructor that takes no arguments.
2. If a user calls `DocumentStore.undo()`, then your `DocumentStore` must undo the last command on the stack
3. If a user calls `DocumentStore.undo(URI)`, then your `DocumentStore` must undo the last command on the stack that was done on the `Document` whose key is the given `URI`, without having any **permanent** effects on any commands that are on top of it in the command stack.
4. Undo must be achieved by `DocumentStore` calling the `Command.undo` method on the `Command` that represents the action to be undone. `DocumentStore` **may not** implement the actual undo logic itself, although it must manage the command stack and determine which undo to call on which `Command`.

3. Undo Logic Requirements

1. There are two cases you must deal with to undo a call to `DocumentStore.put`:
    1. The call to `put` which is being undone added a brand new `Document` to the `DocumentStore`
    2. The call to `put` which is being undone resulted in overwriting an existing `Document` with the same URI in the `DocumentStore`
2. To undo a call to `DocumentStore.delete`, you must put whatever was deleted back into the `DocumentStore` exactly as it was before
3. **DO NOT add any <u>new</u> commands to the command stack in your undo logic. In other words, the undo itself is not "recorded" as a command on the command stack, rather it simply causes the undoing of some pre-existing command. Once the undo process is completely done, there is no record at all of the fact that an undo took place.**

4. Functional Implementations for Undo

1. As stated above, every put and delete done on your `DocumentStore` must result in the adding of a new `Command` onto your command stack.
2. Undo must be defined as lambda functions that are passed as arguments to the `Command` constructor. You can read Chapter 19 in "Building Java Programs" (Reges) to learn about lambdas, closures, and functions. Alternatively, you can read Modern Java Recipes. Once you understand lambdas and closures, you should understand how to implement your undo as lambda functions.

Methods Added to Preexisting Classes or Interfaces for Stage 2

- `DocumentStore.undo()`
- `DocumentStore.undo(URI)`

Stage 3: Keyword Search Using a Trie. **Due April 17, 11:59pm**

Description

In this stage you will add key word search capability to your document store. That means a user can call `DocumentStore.search(keyword)` to get a list of documents in your document store that contain the given keyword. The data structure used for searching is a Trie.

Logic Requirements

1. Create a Trie Which Will Be Used for Searching Your Document Store:

You must create a class `edu.yu.cs.com1320.project.impl.TrieImpl` **in which you implement the** `edu.yu.cs.com1302.project.Trie` **interface.** An Abstract class has been provided - `edu.yu.cs.com1320.project.impl.TooSimpleTrie` – which includes the Trie logic and API that we saw in class. It does NOT do all that is needed for this stage NOR is its API exactly like the one for this project; it is just an example. Do not extend `TooSimpleTrie`, but feel free to cut-and-paste any of its code into yours.

2. Miscellaneous Points:

1. Searching and word counting are CASE **SENSITIVE**. That means that in both the keyword being searched for and in all documents, “THE”, “the”, “ThE”, “tHe”, etc. are all considered to be **DIFFERENT** words.
2. Search results are returned in **descending** order. That means that the document in which a word appears the most times is first in the returned list, the document with the second most matches is second, etc.
3. Document **MUST NOT** implement `java.lang.Comparable`
4. `TrieImpl` **must** use a `java.util.Comparator<Document>` to sort collections of documents by how many times a given word appears in them, when implementing `Trie.getAllSorted` and any other `Trie` methods that return a sorted collection.
5. `TrieImpl` **must** have a constructor that takes no arguments
6. Any search method in `TrieImpl` or `DocumentStoreImpl` that returns a collection **must** return an empty collection, **not null**, if there are no matches.

3. When a `Document` is Added to the `DocumentStore`

1. You must go through the document and create a `java.util.HashMap` that will be stored in the `Document` object that maps all the words in the document to the number of times the word appears in the document.
   a. Be sure to ignore all characters that are not a letter or a number!
   b. This will help you both for implementing `Document.wordCount` and also for its interactions with the Trie
2. For each word that appears in the `Document`, add the `Document` to the Value collection at the appropriate Node in the Trie
   a. The Trie stores **collections** of `Documents` at each node, not **individual** documents!

4. When a `Document` is Deleted from `DocumentStore`

1. You must delete **all** references to it within **all** parts of the Trie.
2. If the `Document` being removed is the last one at that node in the Trie, you must delete it and all ancestors between it and the closest ancestor that has at least one document in its Value collection.

REVISAR

5. Undo

All Undo logic must now also deal with updating the Trie appropriately. Because undo must now deal with an undo action affecting multiple documents at once, the command API has been changed to include the classes listed below. Please see the comments on those classes.

- `edu.yu.cs.com1320.project.Undoable`
- `edu.yu.cs.com1320.project.GenericCommand`
- `edu.yu.cs.com1320.project.CommandSet`

The command stack must be an instance of `StackImpl<Undoable>`. If a command involves a single document, you will create and push an instance of `GenericCommand` onto the command stack. If, however, the command involves multiple documents / URIs, you will use an instance of `CommandSet` to capture the information about the changes to each document. **The `CommandSet` should only be removed from the command stack once the `CommandSet` has no commands left in it due to undo(uri) being called on the URIs of all the commands in the command set.**

Methods Added to Preexisting Interfaces for Stage 3

- `edu.yu.cs.com1320.project.stage3.Document.wordCount(word)`
- `edu.yu.cs.com1320.project.stage3.Document.getWords()`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.search(keyword)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.deleteAll(keyword)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.searchByPrefix(prefix)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.deleteAllWithPrefix(prefix)`

Stage 4: Memory Management, Part 1: Tracking Document Usage via a Heap. **Due May 1, 11:59pm**

Description

In this stage you will use a **min** Heap to track the usage of documents in the document store. Only a fixed number of documents are allowed in memory at once, and when that limit is reached, adding an additional document must result in the least recently used document being deleted from memory (i.e. **all references** to it are removed, thus allowing Java to garbage collect it.)

Logic Requirements

1. Queue Documents by Usage Time via a `MinHeap`

You are given an abstract class called `edu.yu.cs.com1320.project.MinHeap`. You must extend and complete this abstract class as a new class called `edu.yu.cs.com1320.project.impl.MinHeapImpl`. After a `Document` is used and its `lastUsedTime` is updated, that `Document` may now be in the wrong place in the Heap, therefore you must call `MinHeapImpl.reHeapify`. The job of `reHeapify` is to determine whether the `Document` whose time was updated should stay where it is, move up in the heap, or move down in the heap, and then carry out any move that should occur.

`MinHeapImpl` must have a constructor that takes no arguments. `Document` must now extend `Comparable<Document>`, and the comparison must be made based on the last use time (see next point) of each document.

2. Track Document Usage Time

The `Document` interface has 2 new methods:

```
long getLastUseTime();
void setLastUseTime(long timeInNanoseconds);
```

Every time a document is used, its last use time should be updated to the relative JVM time, as measured in nanoseconds (see `java.lang.System.nanoTime()`.) A `Document` is considered to be "used" whenever it is accessed as a result of a call to any part of `DocumentStore`'s **public** API. In other words, if it is "**put**", if it is returned in any form as the result of **any** "get" or "search" request, or if an action on it is **undone** via any call to either of the `DocumentStore.undo` methods.

3. Enforce Memory Limits

The `DocumentStore` interface has 2 new methods:

```
/**
 * set maximum number of documents that may be stored
 * @param limit
 */
void setMaxDocumentCount(int limit);

/**
 * set maximum number of bytes of memory that may be used by all the documents in
memory combined
 * @param limit
 */
void setMaxDocumentBytes(int limit);
```

When your program first starts, there are no memory limits. However, the user may call either (or both) of the methods shown above on your `DocumentStore` to set limits on the storage used by documents. If both setters have been called by the user, then memory is considered to be full if **either** limit is reached.

If the user makes a call to either of the above methods and thereby sets a limit which is lower than the current usage of memory, the least recently used documents should be removed until the use of memory complies with the limits.

For purposes of this project, the memory usage of a document is defined as the total number of bytes in the document's in-memory representation. For text, that's the length of the array returned by String.getBytes(), and for a binary document it is the length of the binary data, i.e. the byte[].

When carrying out a "put" or an "undo" will push the `DocumentStore` above either memory limit, the document store must get the least recently used `Document` from the `MinHeap`, and then erase **all traces** of that document from the `DocumentStore`; it should no longer exist in the Trie, in any Undo commands, or anywhere else in memory. This must be done for as many least-recently-used documents as necessary until there is enough memory below the limit to carry out the "put" or "undo".

For example, assume that 1) the `MaxDocumentBytes` has been set to 10MB, 2) there are nine 1MB files already in memory and 3) the user calls `DocumentStore.put` with a document whose size is 5MB. In this case you have to delete the four least-recently-used documents in memory in order to be able to put the new 5MB document.

Methods Added to Preexisting Interfaces for Stage 4

See the interfaces supplied to you and their comments for details

- `Document.getLastUseTime()`
- `Document.setLastUseTime(timeInNanoseconds)`
- `DocumentStore.setMaxDocumentCount(limit)`
- `DocumentStore.setMaxDocumentBytes(limit)`

Stage 5: Memory Management, Part 2: Two Tier Storage (RAM and Disk) Using a BTree. Due May 22, 11:59pm

Description

In stage #4 a document that had to be removed from memory due to memory limits was simply erased from existence. In this stage, we will write it to disk and bring it back into memory if it is needed. You will continue to use a MinHeap to track the usage of documents in the document store, and you will continue to use the Trie for keyword search. `HashTableImpl`, however, is completely removed from your system, and replaced with a BTree for storing your documents. While the BTree itself will stay in memory, the documents it stores can move back and forth between disk and memory, as dictated by memory usage limits.

Logic Requirements

1. Replace HashTable with BTree

The primary storage structure for documents will now be a BTree instead of a HashTable. All traces of the HashTable you have used until now must be deleted from your code. You must implement `edu.yu.cs.com1320.project.BTree` and your implementation of the BTree must be called `edu.yu.cs.com1320.project.impl.BTreeImpl`. You have been given `edu.yu.cs.com1320.project.impl.WrongBTree`; this implementation is NOT exactly what you need – it has some features that you DO NOT need, and it does not deal with moving things to/from to disk. You may cut and paste whatever parts of it you wish into your code, but DO NOT extend this class and DO NOT submit it as part of your project – it is merely an example and should NOT be used as-is.

An entry in the BTree can have 3 different things as its Value:

1. If the entry is in an internal BTree node, the value must be a link to another node in the BTree
2. If the entry is in a leaf/external BTree node, the value can be either:
   a. a pointer to a Document object in memory OR…
   b. a reference to where the document is stored on disk (IFF it has been written out to disk.)

2. Memory Management

You will continue to track memory usage against limits, and when a limit is exceeded you will use your MinHeap to identify the least recently used doc, as you did in stage #4. **HOWEVER**:

1. When a document has to be kicked out of memory, instead of it being deleted completely it will be written to disk via a call to BTree.moveToDisk. When a document is moved to disk, the entry in the BTree has a reference to the file on disk as its value instead of a reference to the document in memory. When a document is written out to disk, it is removed from the MinHeap which is managing memory.
2. No data structure in your document store other than the BTree should have a direct reference to the Document object. Other data structures should only store the document URI, and call BTree.get whenever they need any piece of information from the document, e.g. it’s lastUseTime, its byte[], etc.
   - Even though this means the MinHeap will have to call the BTree every time it wants to compare the lastUseTime of two documents, and this is very inefficient, we neither want to complicate this project more by dealing with mechanisms to synch the two, nor do we want to allow the MinHeap to directly reference document objects, so we will just accept this inefficiency.
3. If BTree.get is called with a key/URI whose document has been written to disk, that document must be brought back into memory. If bringing it into memory causes memory limits to be exceeded, other documents must be written out to disk until the memory limit is conformed with. When a document is brought back into memory from disk:
   - its lastUseTime must be set to the current time
   - its file on disk must be deleted

4. If BTree.put is called with a key/URI that already has an entry in the BTree but its document has been written to disk, i.e. the document on disk is going to be deleted, `DocumentPersistenceManager.delete` must be called within the BTree.put logic in order to delete the old document from disk. The new version of the document must be in memory after the put operation has completed.

3. Document Serialization and Deserialization

3.1 What to Serialize

You do not serialize the lasUseTime. You must serialize/deserialize:

1. the contents of the document (String or binary)
2. the URI
3. the wordcount map. **You may not recalculate the word map when bring a document into memory from disk; it must be stored on disk and loaded from disk.**

The following has been added to the `Document` interface:

```
void setWordMap(Map<String,Integer> wordMap);
```

```
Map<String,Integer> getWordMap();
```

This must return a copy of the word→count map so it can be serialized

`DocumentImpl`'s constructor must now have the following signature:

```
public DocumentImpl(URI uri, String text, Map<String, Integer> wordCountMap)
```

If the map is null, you build a map in the constructor. If it is not null (e.g. this document was just deserialized), you use the map that has been passed in.

3.2 Document (De)Serialization

1. `BTreeImpl` MUST NOT implement (de)serialization itself. When `DocumentStoreImpl` is initialing itself, it must call `BTreeImpl.setPersistenceManager` and pass it an instance of `edu.yu.cs.com1320.project.stage5.impl.DocumentPersistenceManager` which will do all the disk I/O for the BTree. BTreeImpl uses the `DocumentPersistenceManager` for all disk I/O.
2. You must complete the `DocumentPersistenceManager` class, whose skeleton has been given to you.
3. By default, your `DocumentPersistenceManager` will serialize/deserialize to/from the working directory of your application (see user.dir property here.) However, if the caller passes in a non-null baseDir as a constructor argument when creating the `DocumentPersistenceManager`, then all serialization/deserialization occurs from that directory instead.
4. `DocumentPersistenceManager` should use instances of `com.google.gson.JsonSerializer<Document>` and `com.google.gson.JsonDeserializer<Document>` to (de)serialize from/to disk.
5. You also must add another constructor to `DocumentStoreImpl` that accept `File baseDir` as an argument, and it must pass that `baseDir` to the `DocumentPersistenceManager` constructor.
6. Documents must be written to disk as JSON documents. You must use the GSON library for this.

Some details about your use of GSON:

1. You must add the maven dependencies for GSON to your pom.xml file.
2. **VERY IMPORTANT LINKS FOR YOU REGARDING HOW TO (DE)SERIALIZE YOUR DOCUMENTS:**
    a. **GSON user's guide**
    b. **How to Encode and Decode JSON Byte Array** - you will need this info if your document is a byte[], not text
        i. Also see this and this

```
<dependency>
```

```
  <groupId>com.google.code.gson</groupId>

  <artifactId>gson</artifactId>

  <version>2.9.0</version>

</dependency>
```

3.3 Converting URIs to a location on disk for Serialized files

Let’s assume the user gives your doc a URI of “`http://www.yu.edu/documents/doc1`”. The JSON file for that document should be stored under `[base  directory]/www.yu.edu/documents/doc1.json`. In other words, remove the “`http://`”, and then convert the remaining path of the URI to a file path under your base directory. Each path segment represents a directory, and the namepart of the URI represents the name of your file. You must add “`.json`” to the end of the file name.

4. Undo

All Undo logic must now also deal with moving things to/from disk in the BTree.