Data Structures Semester Project, Spring 2019

Over the course of the semester, students 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 built using build systems, not javac at the command line.
    b. First exposure to independently learning about and reusing third party libraries, specifically Apache Commons Compress
4. Build your first real piece of software (a search engine)
5. 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 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 IntelliJ, start by looking in IntelliJ’s “Help” menu and read the relevant part of IntelliJ’s documentation
- If it’s a question about a given open source library or tool (e.g. Maven or Apache Commons Compress), 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 Deitel

Important Advice Getting Full Credit for Your Work

1. Update your personal computer to JDK 11, since that is the version I will use when grading
2. Before you submit your final 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. Download VirtualBox and create a Linux (Ubuntu) virtual machine. Install Oracle JDK 11, Maven, and Git, AND NOTHING ELSE, inside that virtual machine if they are not there already. Then `git clone` your project from github 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, at my discretion, in you getting a zero.

1. Under the root of your repository there MUST be a directory named "DataStructures", NO SPACES.
    a. For example, if your repository 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 kin 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 project code - src/main/java/
    c. Your project Junit test code - src/test/java
5. 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.

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 Apache Commons Compress or Junit. Do not manually add them to your classpath. Only declare them as a Maven dependency, and Maven will download them for you.

Stage 1: Build an In-Memory Document Store

Due: Sunday, March 17, 11:59 PM EST

Description

In stage 1 you are building a very simple storage mechanism with no search capability, only "get" and "put." Documents are stored in memory in a HashTable, and can only be retrieved using the key with which they are stored.

Logic Requirements

1. Implement a HashTable from Scratch

- Implement a hash table, with separate chaining to deal with collisions.
- You may not use any Java collection classes, rather you must implement the hash table yourself.
- Your hash table class must be a general purpose class, i.e. not specific to any application or key-value types

2. Implement a Document Store

- As specified in the API, your DocumentStore provides a public API to:
    - put documents in the store
    - get documents out of the store, either as a plain String or as a byte[] which is the compressed version of the String
    - set the default compression format/algorithm to use if one is not specified when putting a document
    - delete a document
- Your document store will use an instance of your HashTable implementation to store documents in memory
- Your code will receive documents as an InputStream, and the document's key as an instance of java.net.URI. When a document is added to your document store, you must do the following:
    - Read in the entire contents of the document from the InputStream as a byte[], and then create a String from the byte[] which you read.
    - Create a hash code of the string.
    - If a doc already is present in the HashTable with this URI and the same hash code, return from your method (i.e. stop here), since this is the same key-value pair you already have stored and nothing more needs to be done.
    - Compress the document using the format specified by the caller
        - Must support zip, 7z, gzip, jar, bzip2, all using the Apache Commons Compress library
    - If the caller does not specify a compression format, use the default compression format
    - Create a document object that holds the following:
        - Compressed file as a byte[]
        - Hash code
        - URI
        - Compression format (one of the values of DocumentStore.CompressionFormat)
    - Insert document object into the hash table with URI as the key and document object as the value
- The default compression format is ZIP unless setDefaultCompressionFormat is called to set it to something else
- For further information about the requirements for put, get, and delete, see the comments on the Java interface

3. General requirements on your implementation

- You must implement the interfaces defined in the code posted on Piazza. Other than constructors, you may not have any public methods in your implementations of these interfaces that are not specified in the interfaces. Any additional methods you add must be private.
- Your implementation classes must be located in the same Java package as the interfaces
- The name of your implementation classes should be the name of the interface with "Impl" (short for "implementation") added to the end. So, for example, "HashTableImpl". That means that you must, at a minimum, have the following classes as your solution:
    - edu.yu.cs.com1320.project.DocumentImpl.java
    - edu.yu.cs.com1320.project.DocumentStoreImpl.java
    - edu.yu.cs.com1320.project.HashTableImpl.java

Stage 2: Refactor. Add Undo Support to the Document Store Using a Stack

Due: Sunday, March 31, 11:59 PM EST

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 on 2) undo the last action on a specific document.

You will also get your first experience with functional programming in Java, as well as

Logic Requirements

1. Move your Impl Classes

Move all of your implementation classes to a subpackage called "edu.yu.cs.com1320.project.Impl". The original package – "edu.yu.cs.com1320.project" – should contain nothing other than my interfaces.

2. Update HashTableImpl to Support Unlimited Entries, i.e. Array Doubling

Implement array doubling on the array used in your HashTableImpl. Don't forget to re-hash all your entries after doubling the array!

3. Update HashTableImpl to Completely Delete an Entry When null is Put

If a user calls "deleteDocument" on a URI, or "putDocument" with null as the value, you must completely remove any/all vestiges of that entry from your HashTable, including any objects created to house it. In other words, it should no longer exist in the array, or chained lists, of your HashTable.

4. Add Undo via a Command Stack

1. Every put and delete done on your `DocumentStore` must result in the adding of a new `Command` instance to a single `Stack` which serves as your **command stack**
    a. No other class besides your document store may have any direct references to the stack; it should be a private field within the `DocumentStore`
    b. You must use the supplied `Command` class to model commands. You may not alter or subclass `Command`.
2. If a user calls `undo()`, then your `DocumentStore` must undo the last command on the stack
3. If a user calls `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. In order to undo and/or redo, the `DocumentStore` must use `Command.undo` and `Command.redo` methods; it may not implement the actual undo or redo logic, although it must manage the command stack itself and determine which undo or redo on which Command to call.
5. Your `Stack` class should implement the `Stack` interface which is in the code I shared with you, be named `StackImpl`, and should be in its own Java file in the edu.yu.cs.com1320.project.Impl package

5. Undo/Redo Logic

- There are two cases you must deal with to undo a put:
    - The put added a brand new `Document` to the `DocumentStore`
    - The put resulted in overwriting an existing `Document` with the same URI in the `DocumentStore`
- To undo a delete, you put whatever was deleted back into the `DocumentStore`
- **DO NOT add any commands to the command stack in your undo/redo logic**
- **Redo IS NOT part of the public API of DocumentStore – it is only there to assist in your undoing a command which is not at the top of the command stack.**

Requirements Version: May 17, 2019

6. Functional Implementations for Undo and Redo

- As stated above, every put and delete done on your `DocumentStore` must result in the adding of a new `Command` onto your command stack.
- Undo and redo must be defined as lambda functions that are passed in to the `Command` constructor.
- You must read Chapter 17 in "Java How to Program" (by Deitel) to learn about lambdas, closures, and functions. (Alternatively, you can read Modern Java Recipes.) Once you learn about lambdas and closures, you should understand how to implement your undo and redo as lambda functions.

Stage 3: Keyword Search Using a Trie

Due: Sunday, April 14, 11:59 PM EST

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 you document store that contain the given keyword. The data structure used for searching is a Trie.

Logic Requirements

1. Implement New Interfaces and Methods

The following methods/interface have been added that must be implemented. See the comments on each of those methods/interface in the newly posted code for more details on what they return.

- edu.yu.cs.com1320.project.Document.wordCount(word)
- edu.yu.cs.com1320.project.DocumentStore.search(keyword)
- edu.yu.cs.com1320.project.DocumentStore.searchCompressed(keyword)
- edu.yu.cs.com1320.project.Trie

You must create a class edu.yu.cs.com1320.project.Impl.TrieImpl in which you implement the Trie. An Abstract class has been provided - edu.yu.cs.com1320.project.Impl.TooSimpleTrie – which includes the Trie logic that we saw in class. It does NOT do all that is needed for this stage; it is just there for your convenience/reference.

Miscellaneous Points:

- Searching and word counting is CASE INSENSITIVE. That means that in both the keyword and the document, "THE", "the", "ThE", "tHe", etc. are all considered to be the same/matching word.
- Search results are returned in descending order. That means that 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.
- Document DOES NOT implement java.lang.Comparable
- The Trie must create a java.util.Comparator<Document> which can be used to sort a collection of documents by how many times a given word appears in them. This is to be used when implementing Trie.getAllSorted.

2. When a Document is Added to the DocumentStore…

- you must go through the document and create a 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 documents.
    - Be sure to ignore all punctuation!
    - This will help you both for implementing Document.wordCount and also for its interactions with the Trie
- For each word that appears, add it to the Value collection at the appropriate Node in the Trie
    - Trie Values are collections, not individual Documents!

4. When a Document is Deleted From DocumentStore…

1. You must delete all references to it from the Trie.
2. As we discussed in class, if the Document being removed is that 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.

5. Undo/Redo

All Undo/Redo logic must now also deal with updating the Trie appropriately.

Stage 4: Memory Management, Part 1: Tracking Document Usage via a Heap

Due: Sunday, May 12, 11:59 PM EST

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.

Logic Requirements

1. Queue Documents by Usage Time via a MinHeap

You are given (posted on Piazza) 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

2. Track Document Usage Time

You must add 2 new methods to the Document interface:

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

Every time a document is used, its last use time should be updated to the current system time, as measured in milliseconds (see java.lang.System.currentTimeMillis.) 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", or returned in any form as the result of a "get" or "search" request, or an action on it is undone via any call to "undo."

Document must also now extend Comparable<Document>.

3. Enforce Memory Limits

You must add 2 new methods to the DocumentStore interface:

```
/**
 * 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 compressed
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.

When carrying out a “put” or an “undo” will push the DocumentStore above either 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 compressed size is 5MB. In this case you have to delete the four least recently used document in memory in order to be able to put the new 5MB document.

Stage 5: Memory Management, Part 2: Two Tier Storage (RAM and Disk) Using a Btree

Due: Friday, May 31, 5:00 PM EST

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 be able to 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. The HashMap, 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. The API that your BTree must implement is copied below. You must copy this interface into your **`edu.yu.cs.com1320.project`** package, and your implementation of the BTree must be **`edu.yu.cs.com1320.project.Impl.BTreeImpl`**. I have posted on Piazza a reference implementation of a BTree. 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 extend this class if you wish, and/or reuse whatever parts of it you wish.

```
package edu.yu.cs.com1320.project;

public interface BTree<Key extends Comparable<Key>, Value>
{
    Value get(Key k);
    Value put(Key k, Value v);
    void moveToDisk(Key k) throws Exception;
}
```

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 pointer to a Document object in memory OR a reference to where the document is stored on disk (if 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 identify the least recently used doc using your MinHeap, 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 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.
   - 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 have 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.
2. If get or put is called on a 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

3. Document Serialization and Deserialization

3.1 What to Serialize

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

1. the byte[] which has the contents of the document
2. the compression format
3. the URI/key
4. the document hashcode
5. the wordcount map.

You must add the following to the Document interface and DocumentImpl class:

```
Map<String,Integer> getWordMap();
```
This must return a copy of the word→count map so it can be serialized

```
void setWordMap(Map<String,Integer> wordMap);
```
This must set the word→count map during deserialization

3.2 Library for serialization

Documents must be written to disk as JSON documents. You must use the GSON library for this. I strongly recommend you go through this tutorial about using GSON. Some details about your use of GSON:

1. you must write a custom serializer/deserializer for Documents
2. because we have a byte[] for document contents, you will have to use a byte[] encoder/decoder for this field. See this post for an example that uses the org.apache.commons.codec.binary.Base64 class for this.
3. You must declare a maven dependency on GSON in your project, and on Apache Commons Codec to use that encoder/decoder

3.3 Serialization Interface

Your BTreeImpl class MUST NOT have any serialize/deserialize logic in it. Below is an abstract class for document I/O. You must copy this abstract class into your **`edu.yu.cs.com1320.project`** package, and you must extend this abstract class in a class called **`edu.yu.cs.com1320.project.Impl.DocumentIOImpl`**. Your BTreeImpl must use this class for all disk I/O.

By default, your DocumentIOImpl will serialize/deserialize to/from the working directory of your application (see user.dir property here.) However, if the passes in a baseDir when creating the Document store, then all serialization/deserialization occurs from that directory.

```
package edu.yu.cs.com1320.project;

import java.io.File;
import java.net.URI;

public abstract class DocumentIO
{
    protected File baseDir;
    public DocumentIO(File baseDir)
    {
      this.baseDir = baseDir;
    }
    public DocumentIO()
    {
    }
    /**
     * @param doc to serialize
     * @return File object representing file on disk to which document was serialized
     */
    public File serialize(Document doc)
    {
      return null;
    }

    /**
```

```
     * @param uri of doc to deserialize
     * @return deserialized document object
     */
    public Document deserialize(URI uri)
    {
      return null;
    }
}
```

You also must add a constructor to DocumentStoreImpl that accept File baseDir and pass it to the constructor of DocumentIO.

3.4 Converting URIs to location 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/Redo

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

WARNINGS

1. Make sure the name of your Impl package starts with a capital "I", not lower case "i"
2. Make sure your project is under the correct directory, e.g. https://github.com/Yeshiva-University-CS/UncleMoishe/DataStructures/project/stage5