

Data Structures Semester Project, Spring 2020 (v. 1.7)

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 your 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. Large real-world software is built using build systems, not manually invoking `javac` at the command line.
 - b. First exposure to independently learning about and reusing third party libraries, specifically [Apache PDFBox - A Java PDF Library](#)
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 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
- If it's a question about a given open source library or tool (e.g. Maven or Apache PDFBox), 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 and/or Lewis & Chase

Important Advice Getting Credit for Your Work

1. **Code that does not compile will get a zero unless the professor is overly kind and fixes it for you. Even in that overly kind situation, you will lose a large number of points.**
2. Make sure your personal computer has [Oracle JDK 8](#), not any other version of Java, since that is the version I will use when grading
3. 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, by following these steps:
 - i. Download [VirtualBox](#) and create a Linux (Ubuntu) virtual machine. [Follow these instructions](#), but note that there are newer versions of VirtualBox and Ubuntu than those mentioned in the video.

- ii. Install [Oracle JDK 8](#), [Maven](#), and [Git](#), AND NOTHING ELSE, inside that virtual machine if they are not there already.
- iii. `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 user 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. 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.
6. Each stage of the project must compile and run entirely independently of the other stages; in other words, each stage must be submitted 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 of each stage.

All interfaces and starter code for all stages of the project have been posted at <https://github.com/Yeshiva-University-CS/DataStructuresProjectSpring2020>

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 PDFBox or Junit. 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.**
-

Stage 1: Build an In-Memory Document Store

Due: Tuesday, February 25, 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. Documents can be plain text, a.k.a. a `String`, or they can be PDFs.

You will use the open source library [Apache PDFBox - A Java PDF Library](#) to extract text from PDF files, and generate PDF files from text. Some additional links for working with PDFBox:

- [Maven coordinates to download PDFBox](#)
- [Code example of extracting text from a PDF file](#)
- [Code example of creating a new PDF file from text](#)

Logic Requirements

1. *Implement a HashTable from Scratch*

- Implement a hash table which uses separate chaining to deal with collisions.
- 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 should be 5.
- 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 (i.e. [generic](#)) class, i.e. not be limited to specific key-value types.
- It must implement the `edu.yu.cs.com1320.project.HashTable` interface that has been provided to you.
- It must be called `edu.yu.cs.com1320.project.impl.HashTableImpl`

2. *Implement a Document Store*

- Implement the `edu.yu.cs.com1320.project.stage1.DocumentStore` interface, which specifies an API to:
 - put a document in the store
 - get a document as a `String` or as a `byte[]` that makeup the PDF version of the document
 - delete a document
- It must be called `edu.yu.cs.com1320.project.stage1.impl.DocumentStoreImpl`
- Your document store must use an instance of `HashTableImpl` to store documents in memory.
- If a user calls `deleteDocument`, or `putDocument` 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.
- 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:
 - Read the entire contents of the document from the `InputStream` into a `byte[]`
 - If the document format is PDF, extract the text of the PDF and hold onto it as a `String`. If the document format is TXT, just hold on to it as a `String` – no need to create a PDF at this point.
 - Get the hash code of the `String` by calling [String.hashCode\(\)](#)

- If a document is already present in the hash table with this URI and the same hash code, return from your method (i.e. stop here), since the document you are trying to put is IDENTICAL to the one already stored (hence them having the same hash code). Otherwise, continue to the next step.
- Create a `Document` object that holds the following:
 - The hash code of the text of the document
 - The URI
 - The document text, and the `byte[]` PDF content if it was initially passed to you as a PDF
- Your document object must be an instance of a class that implements the `edu.yu.cs.com1320.project.stage1.Document` interface.
- Document **MUST NOT** implement `java.lang.Comparable`
- Insert document object into the hash table with URI as the key and document object as the value

3. Other requirements on your implementation

- You must fully implement the interfaces defined in the code provided to you under the `stage1` folder. Other than constructors, you may not have any public or protected methods in your implementations of these interfaces that are not specified in the interfaces. Any additional methods you add must be private.
- 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, at a minimum, have the following classes as your solution:
 - `edu.yu.cs.com1320.project.stage1.impl.DocumentImpl`
 - `edu.yu.cs.com1320.project.stage1.impl.DocumentStoreImpl`
 - `edu.yu.cs.com1320.project.impl.HashTableImpl`
- Make sure the name of your `impl` package starts with a lower case “i”, not an upper case “I”
- DO NOT move the interfaces – they must remain in the packages in which they were placed in the starter code.
- `HashTableImpl` and `DocumentStoreImpl` must both have no-argument constructors. `DocumentImpl` must provide the following two constructors:

```
public DocumentImpl(URI uri, String txt, int txtHash){
}
public DocumentImpl(URI uri, String txt, int txtHash, byte[] pdfBytes){
}
```

Stage 2: Add Undo Support to the Document Store Using a Stack. Your First Exposure to Functional Programming in Java.

Due: Tuesday, March 17, 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 to 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. 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.putDocument` and `DocumentStore.deleteDocument` 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 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

- There are two cases you must deal with to undo a call to `DocumentStore.putDocument`:
 - `putDocument` added a brand new `Document` to the `DocumentStore`
 - `putDocument` resulted in overwriting an existing `Document` with the same `URI` in the `DocumentStore`
- To undo a call to `DocumentStore.deleteDocument`, you must put whatever was deleted back into the `DocumentStore` exactly as it was before
- **DO NOT add any new commands to the command stack in your undo logic**

4. Functional Implementations for Undo

- 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 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 as lambda functions.

5. Protected method in document store for my testing purposes:

- Add the following method to your `DocumentStoreImpl`:

```
/**
 * @return the Document object stored at that URI, or null if there is no such
 * Document
 */
protected Document getDocument(URI uri) {
}
```

Methods Added to Preexisting Classes or Interfaces For Stage 2

- `DocumentStore.undo()`
 - `DocumentStore.undo (URI)`
 - `HashTableImpl` **must** have a constructor that takes no arguments
 - `DocumentStoreImpl.getDocument (URI uri)`
-

Stage 3: Keyword Search Using a Trie.

Due: Friday, April 17, 10:00 AM 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. 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 there for your convenience/reference.

2. Miscellaneous Points:

- Searching and word counting are CASE INSENSITIVE. That means that in both the keyword and the document, “THE”, “the”, “ThE”, “tHe”, etc. are all considered to be the same 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 **MUST NOT** implement `java.lang.Comparable`
- `TrieImpl` **must** use `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.
- `TrieImpl` **must** have a constructor that takes no arguments
- 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

- 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.
 - Be sure to ignore all characters that are not a letter or number!
 - This will help you both for implementing `Document.wordCount` and also for its interactions with the Trie
- For each word that appears in the `Document`, add the `Document` to the `Value` collection at the appropriate Node in the Trie
 - 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 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

All Undo logic must now also deal with updating the Trie appropriately. Because undo must now deal with undo 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. When a `CommandSet` has no commands left in it due to `undo(uri)` being called on the URIs of all the commands in the command set, the `CommandSet` should be removed from the command stack.

Methods Added to Preexisting Interfaces For Stage 3

- `edu.yu.cs.com1320.project.stage3.Document.wordCount (word)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.search (keyword)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.searchPDFs (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.searchPDFsByPrefix (prefix)`
- `edu.yu.cs.com1320.project.stage3.DocumentStore.deleteAllWithPrefix (prefix)`

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

Due: Sunday, April 26, 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 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

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 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”, or returned in any form as the result of any “get” or “search” request, or an action on it is undone via any call to either of the `DocumentStore.undo` methods.

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 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.

For purposes of this project, the memory usage of a document defined as the total number of bytes in both of the document’s two representations (TXT/`String.getBytes()` and PDF/`byte[]`) added together, i.e. you add the length of those two byte arrays.

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 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.

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: Sunday, May 24, 11:59 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 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 extend this class if you wish, and/or cut and paste whatever parts of it you wish into your code.

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 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 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.
3. If `BTree.get` or `BTree.put` is called with 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 `lastUseTime` or the `PDF` `byte[]`. You must serialize/deserialize:

1. the text/String version of the contents of the document
2. the URI/key
3. the document contents hashcode
4. the wordcount map.

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

```
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 Document (De)Serialization

- `BTreeImpl` MUST NOT implement (de)serialization itself. `DocumentStoreImpl` calls `BTreeImpl.setPersistenceManager`, passing 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.
- You must complete the `DocumentPersistenceManager` class, whose skeleton has been given to you.
- 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` when creating the `DocumentPersistenceManager`, then all serialization/deserialization occurs from that directory.
- `DocumentPersistenceManager` should use instances of `com.google.gson.JsonSerializer<Document>` and `com.google.gson.JsonDeserializer<Document>` to (de)serialize from/to disk.
- You also must add another constructor to `DocumentStoreImpl` that accept `File baseDir` as an argument.
- 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 `JsonDeserializer` / `JsonSerializer` for Documents
2. You must add the maven dependencies for GSON.

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.6</version>
</dependency>
```

3.3 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

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