Project 1 Inverted Index

For this project, you will write a Java program that recursively processes all text files in a directory and builds an inverted index to store the mapping from words to the documents (and position within those documents) where those words were found. For example, suppose we have the following mapping stored in our inverted index:

```
{
  "capybara": {
    "mammals.txt": [
      11
    ]
  },
  "platypus": {
    "mammals.txt": [
      3,
      8
    ],
    "venomous.txt": [
      2
    ]
  }
}
```

This indicates that the word `capybara` is found in the file `mammals.txt` in position 11. The word `platypus` is found in two files, `mammals.txt` and `venomous.txt`. In the file `mammals.txt`, the word `platypus` appears twice in positions 3 and 8. In file `venomous.txt`, the word `platypus` is in position 2 in the file.

Functionality

For this project, your code must traverse a directory and process all text files found in that directory (including all subdirectories). For each text file, you must parse each line into words. For each word, you must store a mapping of the word to the file and position that word was found in a custom inverted index data structure.

Specifically, the core functionality of your project must satisfy the following requirements:

- Create an custom **inverted index** data structure that stores a mapping from a word to the file(s) the word was found, and the position(s) in that file the word is located. *This will require nesting multiple built-in data structures.*
- Store the normalized relative path for file locations as a `String` object. Do not convert the file paths to absolute paths!

- Positions stored in your inverted index should start at 1. For example, if a file has words "ant bat cat", then "ant" is in position 1, "bat" is in position 2, and "cat" is in position 3.
- Traverse a directory and its subdirectories and parse all of the text files found. Any file ending in the `.txt` extension (case-insensitive) should be considered a text file.
- Use the `UTF-8` character encoding for all file processing (including reading and writing).
- Replace all characters except letters and digits with an empty string. This includes the `_` underscore character as well. For example, the word `age_long` should be seen as `agelong` since the `_` underscore will be replaced with an empty string.
- Separate text into words by any whitespace character, including spaces, tabs, and newline characters.
- Words must be case-insensitive. For example, the words APPLE, Apple, apple, and aPpLe should all be seen as the same word.
- Process command-line parameters to determine the directory to parse and output to produce. See the **Execution** section below for specifics.
- Output the inverted index to a text file if the proper command-line parameters are provided. See the **Output** section below for specifics.

You must satisfy the core functionality prior to submitting your project for code review. See the **Project README** for specifics.

Design

In addition to the functionality requirements, you should consider the following when designing your project:

- Your program should support large text files. As a result, you should **not** read the entire file into memory at once. Instead, read a single line from the file into memory at a time.
- Your program should be object-oriented. For example, you should separate directory traversing, file parsing, and data maintenance into different classes.
- You program should protect the integrity of your inverted index, making sure all private data members are properly encapsulated. For example, do not return a reference to a private mutable data member in a public method.

These requirements are **not** necessary to be eligible for code review. You should only worry about these requirements once your core functionality is complete.

Execution

Your `main` method must be placed in a class named `Driver`. The `Driver` class should accept the following command-line arguments:

- `-input directory` where `-input` indicates the next argument is a directory, and `directory` is the input directory of text files that must be processed
- `-index filepath` where `-index` is an *optional* flag that indicates the next argument is a file path, and `filepath` is the path to the file to use for the inverted index output file. If the `filepath` argument is not provided, use `index.json` as the default output filename. If the `-index` flag is not provided, do not produce an output file.

The command-line flag/value pairs may be provided in any order. Your code should support all of the command-line arguments from the previous project as well.

⚠️ The `Driver` class should be the only file that is not generalized and specific to the project. If the proper command-line arguments are not provided, the `Driver` class should output a user-friendly error message to the console and exit gracefully.

Examples

The following are a few examples (non-comprehensive) to illustrate the usage of the command-line arguments. Consider the following example:

```
java Driver -input input/index/simple
            -index index-simple.json
```

In this case your program should build an inverted index from all of the text files in the `input/index/simple` directory. The inverted index should be output to the file `index-simple.json` in the current working directory.

Next, consider the following example:

```
java Driver -index -input input/index/simple
```

Your program should execute the same as before, except that it will output the inverted index to the default file `index.json` instead. Notice how the arguments may appear in any order.

Next, consider the following example:

```
java Driver -input input/index/simple
```

In this case, your program still executes as before but produces no output files. **Your code must still, however, build the inverted index!** This is useful later for testing the efficiency of your code.

Output

The output of your program should be a file that contains the contents of your inverted index in alphabetically sorted order as a nested JSON object using a "pretty" format using 2 spaces for indentation. For example:

```
{
  "capybara": {
    "mammals.txt": [
      11
    ]
  },
  "platypus": {
    "mammals.txt": [
      3,
      8
    ],
    "venomous.txt": [
      2
    ]
  }
}
```

Each path should be a normalized relative path. Make sure there are no trailing commas after the last element. See the JSON files in the `expected` directory for more examples.

Hints

It is important to develop the project **iteratively**. In fact, you may already have certain components complete thanks to the homework assignments. One possible breakdown of tasks are:

- Create code that handles parsing command-line arguments, so that you can easily retrieve the directory containing the text files to parse. Test your code.
- Create code that is able to traverse a directory and return a list of all the text files found within that directory. Test your code.
- Create code that handles parsing a single text file into words. Test your code.
- Create code that handles storing a word, file path, and location into an inverted index data structure. Test your code.
- Integrate your code.
    - Make your code that parses a text file into words store words in an inverted index data structure.
    - Make your code that traverses a directory work with the code that parses a text file and stores words into an inverted index data structure.
    - Make your code that parses command-line arguments work with the code that traverses a directory, and so on.

The important part will be to **test your code as you go**. The tests provided only test the entire project as a whole, not the individual parts. You are responsible for testing the individual parts themselves.

💡 These hints may or may *not* be useful depending on your approach. Do not be overly concerned if you do not find these hints helpful for your approach for this project.

Submission

💡 See the **Project README** for additional details on testing, submission, and code review.

Project 2 Partial Search

For this project, you will extend your previous project to support partial search. In addition to meeting the previous project requirements, your code must return a *sorted* list of results from your inverted index that start with the provided query word(s).

For example, suppose your inverted index contains the words: `after`, `apple`, `application`, and `happen`. If the query word is `app`, your code should return results for both `apple` and `application` but *not* `happen`.

Your search results must be sorted such that the most relevant search result is listed first, and the least relevant search result is listed last. You will determine relevance based on both the frequency of the query word and position.

Functionality

For this project, your code must pass all of the previous project requirements and support the following additional functionality:

- Process additional command-line parameters to determine the file to parse for queries. See the **Execution** section below for specifics.
- Efficiently return partial search results from your inverted index, such that any word in your inverted index that *starts with* a query word is returned.
- Sort the search results using a simple metric for relevance (see details below).
- Support multiple word search queries, where search results from individual query words are combined together. For example, a file should only appear *once* in the search results even if it contains multiple words from a query.
- Output the sorted search results to a text file if the proper command-line parameters are provided. See the **Output** section below for specifics.

More details are provided next for how to read and parse queries and sort the search results.

Query File Parsing

Queries will be provided in a text file. The input query file will consist of one multi-word search query per line. The query file may contain symbols and mixed-case words. You will need to perform the necessary transformations to match the queries with your inverted index. A sample query file may look like:

```
two
chicka-dee
elephANT & ANTelope
```

There are three different search queries in this example. The first line `two` consists of only one query word. The second line `chicka-dee` should have the special character replaced with an empty string to form a one word query `chickadee`. The last line `elephANT & ANTelope` should be made all lowercase and have the special characters removed, resulting in the two word query `elephant` and `antelope` instead.

Result Sorting

To rank your search results, use the following criteria:

- **Frequency:** Locations where the query word(s) are more frequent should be ranked above others. For example, suppose `a.txt` has 5 partial matches to the query and `b.txt` has 10 partial matches. In that case, `b.txt` should appear in the results before `a.txt`.
- **Position:** For locations that have the same frequency of query word(s), locations where the words appear in earlier positions should be ranked above others. For example, suppose `a.txt` and `b.txt` both have the same number of partial matches. If `a.txt` has the first partial match in position 10 and `b.txt` has the first partial match in position 5, then `b.txt` should appear in the results before `a.txt`.
- **Location:** For locations that have the same frequency and position, the results should be sorted by path in case-insensitive order. For example, suppose `a.txt` and `b.txt` both have the same number of partial matches and positions. If `a.txt` and `b.txt` are in the same directory, then `a.txt` should appear in the results before `b.txt`.

You can use the `String.CASE_INSENSITIVE_ORDER` comparator for comparing absolute paths and `Integer.compare(int, int)` or `Integer.compareTo(Integer)` to compare frequencies and positions.

Multiple Word Queries

For multiple-word queries, you should add the count for each word together and use the earliest position any of the query words appear.

For example, suppose the query words are `apple banana`. Let `file1.txt` have 15 results for `apple` and let `file2.txt` have 5 results for `apple` and 5 results for `banana`. Then, `file1.txt` will be ranked higher than `file2.txt` since the total count of 15 for `file1.txt` is greater than that of 10 for `file2.txt` (even though `file2.txt` has both of the search terms). If `apple` first appears in position 3 in `file2.txt` and `banana` first appears in position 14, then 3 should be used as the position for sorting.

Design

In addition to the functionality requirements, you should consider the following when designing your project:

- Your program should support large text files. As a result, you should **not** read the entire file into memory at once. Instead, read a single line from the file into memory at a time.
- Your program should be object-oriented. For example, you should separate directory traversing, file parsing, and data maintenance into different classes.
- Your program should protect the integrity of your inverted index, making sure all private data members are properly encapsulated. For example, do not return a reference to a private mutable data member in a public method.
- Your program should search efficiently; iterating through the smallest number of elements as possible to create the partial search results.

These requirements are **not** necessary to be eligible for code review. You should only worry about these requirements

once your core functionality is complete.

Execution

Your `main` method must be placed in a class named `Driver`. The `Driver` class should accept the following **additional** command-line arguments:

- `-query filepath` where `-query` indicates the next argument is a path to a text file of queries. If this flag is not provided, then no search should be performed. In this case, your code should behave exactly the same as the previous project.
- `-results filepath` where `-results` is an *optional* flag that indicates the next argument is a file path, and `filepath` is the path to the file to use for the search results output file. If the `filepath` argument is not provided, use `results.json` as the default output filename. If the `-results` flag is not provided, do not produce an output file of search results.

The command-line flag/value pairs may be provided in any order. **Your code should support all of the command-line arguments from the previous projects as well.**

⚠️ The `Driver` class should be the only file that is not generalized and specific to the project. If the proper command-line arguments are not provided, the `Driver` class should output a user-friendly error message to the console and exit gracefully.

Examples

The following are a few examples (non-comprehensive) to illustrate the usage of the command-line arguments. Consider the following example:

```
java Driver -input input/index/simple
            -query input/query/simple.txt
            -index index-simple.json
            -results results-simple.json
```

In this case your program should build an inverted index from all of the text files in the `input/index/simple` directory. It should then build search results for all the queries located in the file `input/query/simple.txt`. The inverted index should be output to the file `index-simple.json` in the current working directory, and the search results should be output to a file `results-simple.json`.

Next, consider the following example:

```
java Driver -input input/index/simple
            -query input/query/simple.txt
            -results
```

Your program should execute the same as before, except that it will not output the inverted index to file and will output the search results to the default file `results.json` instead. Note that in this case, you must still build the inverted index even though you do not output it to a file.

Next, consider the following example:

```
java Driver -input input/index/simple
            -query input/query/simple.txt
```

In this case, your program still executes as before but produces no output files. In general, if the flags `-index` or `-results` are not provided, then your program should not produce any output files. **Your code must still, however, build**

the inverted index and perform the partial search! This is useful later for testing the efficiency of your code.

Output

The output of your inverted index should be the same from the previous project. The output of your search results should be sorted by the order the queries were found in the original query file, and use "pretty" JSON with 2 spaces for each indent level. Each search result should be considered an object with "where" (for the quoted path), "count", and "index" (for position) keys. Consider the following example:

```
{
  "two": [
    {
      "where": "input/index/simple/position.txt",
      "count": 1,
      "index": 2
    }
  ],
  "chicka-dee": [
  ],
  "elephANT & ANTelope": [
    {
      "where": "input/index/simple/symbols.txt",
      "count": 9,
      "index": 1
    },
    {
      "where": "input/index/simple/capitals.txt",
      "count": 4,
      "index": 1
    }
  ]
}
```

The example above demonstrates the case where there is only 1 search result, 0 search results, and 2 search results (note where the commas are included). The **original** query (including all words if it is a multiple word query) is listed in the order they appeared in the query file as keys to the outer object. The value is an sorted array of search results, where each search result is an object with the keys "where", "count", and "index". The value of the "where" key is a quoted normalized relative path, and the search result count and index/position is given (without quotes) as the values of the other keys.

Hints

It is important to develop the project **iteratively**. One possible breakdown of tasks are:

- Create a class that stores a single search result and implements the `Comparable` interface. You will need data members for each of the sort criteria (frequency, initial position, and path).
- Add a partial search method to your inverted index data structure that takes already parsed words from a single query, and returns a sorted list of search results.
- Use lists and `Collections.sort(List)` to sort search results, not a [`TreeSet`] data structure. *Custom mutable objects do not behave well when used in sets or as keys in maps.*
- Create a query helper class, that is able to parse the query file into lines and lines into either an array or list of cleaned words. For each query parsed, use the search method for your inverted index to get the results and save them to be output to a file later.
- Do not worry about efficient partial search until *after* you are getting correct results.

- Check out the code demos from the Data Structures lectures for hints on how to efficiently iterate and search through different types of data structures.

The important part will be to **test your code as you go**. The JUnit tests provided only test the entire project as a whole, not the individual parts. You are responsible for testing the individual parts themselves.

💡 These hints may or may *not* be useful depending on your approach. Do not be overly concerned if you do not find these hints helpful for your approach for this project.

Submission

💡 See the **Project README** for additional details on testing, submission, and code review.