Project 4: Dictionary Builder

CMSC 204

Fall 2025

Assignment Overview

In this project, you will implement a dictionary or concordance system using a hash table with chaining. The program will read words from a file and store each one with an occurrence count. You will also implement lookup and removal functionality.

Learning Objectives

- Implement a hash table with separate chaining using linked lists
- Parse and clean text from an input file
- Design a reusable class to represent dictionary entries
- Apply the concepts of hash codes, buckets, and collision handling

Project Requirements

Your program must:

- Read a text file and insert all **unique** words into a hash table. Each word should be:
  - Converted to lowercase
  - Have punctuation removed
  - Be stored with a frequency counter
- Use **separate chaining** (linked lists) to handle collisions.
- You must write your own linked list class to store words in each bucket. You *may* re-use the linked list implementation from previous class projects.
- Estimate the correct hash table size based on the input file size and a target load factor of 0.6.
- Create a DictionaryShell class (command line interface) that provides an interface to your Dictionary-Builder class. It should include:
  - Accepting a command line argument of a filename to load
  - Support the following commands: `search <word>`, `add <word>`, `delete <word>`, `list`, `stats`, `exit`
- **Important:** You may not use any built-in collection classes (such as `ArrayList`, `LinkedList`, `HashMap`, or `Set`) to implement the hash table or its chains. However, you may use collection classes for post-processing tasks (e.g., sorting output), as long as they are not part of your hash table structure.
- Standard utility classes (e.g., `Scanner`, `String`, `Arrays`) are allowed for parsing and I/O.

Assumptions and Notes

- Input may be arbitrary text, not one word per line. You must sanitize and split input text appropriately.
- Your dictionary should treat words case-insensitively and strip punctuation (e.g., "apple" and "Apple!" should be treated the same).
- Duplicates must not be inserted.
- Sorting for `list` may be done after collecting all words from the table.

You will implement the following classes:

- `DictionaryBuilder` – manages the hash table, implements methods:
  - `DictionaryBuilder(int estimatedEntries)` - Constructor that creates a hash table appropriate for the estimated entries. Must consider load factor and 4k+3 prime table size.
  - `DictionaryBuilder(String filename)` - Constructor that reads a file and adds all the words to the DictionaryBuilder
  - `addWord(String word)`
  - `getFrequency(String word)`
  - `removeWord(String word)`
  - `getAllWords()` – return sorted ArrayList of all the words in the dictionary
- `DictionaryShell` - contains a main that provides a command-line interface to your DictionaryBuilder Class. It should:
  - Accept a command line argument of a filename to load as the dictionary (see the section on Using Command Line Arguments)
  - Allow the user to interact with the dictionary using the following commands:
    - `search <word>` – report whether a word exists
    - `add <word>` – insert a new word into the table
    - `delete <word>` – remove a word from the table
    - `list` – print all stored words in alphabetical order
    - `stats` – print the following statistics about the dictionary
      - **total words** the total number of words inserted into the dictionary *including* duplicates
      - **unique words** the number of *different* words inserted into the dictionary.
      - **estimated load factor** the load factor for the underlying hash table.
    - `exit` - quit the program

Hash Table Design

- Use an array of linked lists for buckets (you may reuse your generic Linked List from the previous project)
- Use the word's hashCode (from Java String) to find a bucket
- Handle collisions using chaining
- Implement your own resizing method (optional challenge)

Choosing a Hash Table Size

Even though separate chaining handles collisions by design, a poor table size or high load factor can still result in long chains and degraded performance. This is why choosing a good table size is important.

Your hash table must be large enough to reduce collisions while remaining efficient in memory use. To help guide your design, we recommend aiming for a **load factor** of approximately 0.6. The load factor is defined as:

$$\text{Load Factor} = \frac{\text{Number of stored elements}}{\text{Table size}}$$

To estimate the number of unique words in the input file **without reading it twice**, use the following conservative rule of thumb:

$$\text{Estimated Unique Words} \approx \frac{\text{File size in bytes}}{100}$$

For a justification of this estimation method, see *Appendix A: Justification for Hash Table Size Heuristic* at the end of this document.

Then use this to compute a recommended table size:

$$\text{Table Size} = \left\lceil \frac{\text{Estimated Unique Words}}{0.6} \right\rceil = \left\lceil \frac{\text{File size}}{60} \right\rceil$$

Once you compute this value, select the smallest **prime number of the form** $4k + 3$ that is greater than or equal to your result. Use the provided `primes.txt` file to find a suitable value.

Example: If your input file is approximately `180,000 bytes`, you estimate:

- Unique words: `180000 / 100 = 1800`
- Table size: `1800 / 0.6 = 3000`
- Choose the next $4k + 3$ prime ≥ 3000

Include a brief comment or note in your code explaining:

- The file size used for your estimate
- How you calculated the number of unique words
- The final table size you selected

Exception Handling

Gracefully handle:

- File not found (FileNotFoundException)
- Empty dictionary
- Word not found during removal or lookup (DictionaryEntryNotFoundException)

Using Command Line Arguments

Your program's entry point is the `main(String[] args)` method. This method may receive command-line arguments when the program is launched from the terminal.

- If one argument is provided, treat it as the name of a file to load.
- If no arguments are provided, initialize an empty dictionary instead.

The following is an example of how you might check for and use a command-line argument:

```
public static void main(String[] args) {
    if (args.length == 0) {
        System.out.println("No command-line arguments provided.");
    } else {
        System.out.println("Command-line arguments:");
        for (int i = 0; i < args.length; i++) {
            System.out.println("args[" + i + "]: " + args[i]);
        }
    }
}
```

Make sure your program handles missing or unreadable files gracefully using exception handling.

Sample Input and Output

Note: the following example presumes the following input file. `sample.txt:`

```
The quick brown fox jumps over the lazy dog.
The quick red fox leapt over the sleepy cat.
```

Note on Input Files

To ensure your program can read the input file correctly, place `sample_input.txt` in your project's **root directory**, not inside the `src/` folder.

Eclipse users:

- Right-click your project in the **Package Explorer**.
- Choose `Show in` → `System Explorer` (or Finder on macOS).
- Move or copy `sample_input.txt` into that root project folder (the same level as `src`).
- When running your program, Java will look for the file relative to the working directory, which defaults to this root.

If you place the file inside `src/`, your program may fail to find it unless you adjust the path.

0.1 Console Interaction Example

```
java Main sample.txt

Welcome to the Dictionary Builder CLI.
Available commands: add <word>, delete <word>, search <word>, list, stats, exit
> search fox
2 instance(s) of "fox" found.
```

```
> search elephant
"elephant" not found.

> add elephant
"elephant" added.

> delete red
"red" deleted.

> list
brown
cat
dog
elephant
fox
jumps
lazy
leapt
over
quick
sleepy
the

> stats
Total words: 18
Total unique words: 12
Estimated load factor: 0.63

> exit
Quitting...
```

Testing Notes

The provided GFA (Good Faith Attempt) JUnit tests represent the minimum requirements to validate the basic functionality of your project. If the project due date has passed, your submission must pass these tests in order to be considered for course credit.

In addition to the GFA tests, any GUI code and public JUnit tests are provided to assist with basic verification of your implementation. However, these do not cover all functionality. You are expected to design and run additional tests to ensure the full correctness and robustness of your project.

Student Tests

You must write a suite of JUnit tests that thoroughly verify your implementation. Your tests should include:

- Adding words with varying cases and punctuation (e.g., "Apple" vs. "apple!")
- Searching for known words and verifying frequency counts
- Removing words and checking that the dictionary updates correctly
- Verifying that duplicate words are not re-added
- Handling edge cases, such as an empty dictionary or search/delete on nonexistent words

Running the Provided JUnit Tests

To verify that your implementation meets the project requirements, we have provided a set of automated JUnit test cases. These include:

- `DictionaryBuilderGFATests.java`
- `DictionaryBuilderPublicTests.java`
- `DictionaryCliIntegrationTests.java`

These tests cover both functional correctness and integration behavior. You are expected to pass all tests prior to submission.

Using Eclipse (Recommended)

To run the tests in Eclipse:

1. Make sure your project uses a Java JDK (not a JRE).
2. Right-click your project folder in the **Package Explorer**.
3. Choose **Build Path** → **Configure Build Path**, and ensure that the `junit-5` library is included under **Libraries**. If not, click **Add Library** → **JUnit** → **JUnit 5**.
4. Make sure the test files (`DictionaryBuilderPublicTests.java`, etc.) are in the `src` folder or a designated `test` folder on the build path.
5. To run a test class:
   (a) Right-click the file (e.g., `DictionaryBuilderPublicTests.java`) in the Package Explorer.
   (b) Select **Run As** →**JUnit Test**.
6. Use the **JUnit tab** to see which tests passed or failed. Double-click a failing test to jump to the relevant line.

Test Coverage Notes

- The `DictionaryBuilderGFATests` performs minimal testing to qualify the submission as a Good Faith Attempt.
- The `DictionaryBuilderPublicTests.` focuses on basic expected behavior.
- The `DictionaryCliIntegrationTests` simulate realistic user input and output flow.

If you are failing tests, read the output messages carefully and verify that your logic matches the specification exactly. You are encouraged to write additional tests as you develop your solution.

Deliverables

Before submitting your project, ensure your code is free of syntax errors. Submissions that do not compile will receive zero points.

Design Documents UML and/or Pseudo-Code

Implemention Only submit files that you have created or modified, do not submit unmodified files that were provided in the project download. Place all your .java files inside a src folder. Include the entire doc folder with Javadoc for your own classes.

Summary Write-Up A 2-3 paragraph write-up (eg. LearningExperience.doc)

Submission Packaging

You will submit two compressed `.zip` files:

Main Project Files All student created or modified project files and data
Filename: `LastNameFirstName_AssignmentX.zip`

- **src/** directory containing `.java` files created or modified by the student
- **doc/** directory containing student created Javadoc files
- **LearningExperience.doc** reflection and write-up
- **Design Documents** all design related documents

MOSS files Only the student created or modified source code files
Filename: `LastNameFirstName_AssignmentX_Moss.zip`

- **source files** only the `.java` files created or modified by the student

Grading Rubric

Criteria	Points
Hash Table Implementation	40%
All commands (search, add, delete, list) work as described	25%
Correct Statistics	20%
File parsing and word cleaning	15%
Student JUnit tests	-5%
Code Quality, Style	-5%
Doumentation (Javadoc, Design & Write-up)	-10%

Appendix A: Justification for Hash Table Size Heuristic

To estimate the number of unique words in the input file without reading it twice, this project recommends using the heuristic:

$$\text{Estimated Unique Words} \approx \frac{\text{File size (in bytes)}}{100}$$

This rule of thumb is grounded in both empirical analysis and logical reasoning:

- We analyzed multiple real-world English texts from Project Gutenberg, including *The Complete Works of Shakespeare*, *The Federalist Papers*, *How to Analyze People on Sight*, and *A Short History of the United States*.
- Across these texts, the ratio of file size to number of unique words ranged from 56 to 229, with an average around 132.
- The average length of a word (including a trailing newline character, assuming one word per line) was approximately 8–9 characters.

The 1:100 ratio is a conservative simplification:

- It is easy to apply mentally or programmatically.
- It errs slightly on the side of overestimating the number of unique words, reducing the chance of excessive collisions in the hash table.
- It accounts for variation in formatting, vocabulary richness, and document style.

This kind of estimation reflects a common task in real-world software development: designing efficient data structures when exact input characteristics are not known in advance.

Appendix B: Determining if an Integer is Prime

To determine whether a number is prime, you may use the following method as-is or adapt it. This method uses the 6k ± 1 optimization, which improves efficiency by skipping obvious non-prime candidates and reducing the number of checks:

```
static boolean isPrime(int n)
    {
        // Corner case
        if (n <= 1)
            return false;
        // For n=2 or n=3 it will check
        if (n == 2 || n == 3)
            return true;
        // For multiple of 2 or 3 This will check
        if (n % 2 == 0 || n % 3 == 0)
            return false;
        // Check for remaining possible factors
        for (int i = 5; i <= Math.sqrt(n); i = i + 6)
            if (n % i == 0 || n % (i + 2) == 0)
                return false;

        return true;
    }
```

You can use this method in conjunction with a simple loop to find the next prime number of the form $4k + 3$ that is greater than or equal to a given table size. Be sure to validate both the primality and the congruence condition ($p \bmod 4 = 3$).