# Benchmarks

The benchmarks below are for an 8-way search on the 6x6 grid using a 400,000-word list I made using the Scowl word lists.

|  |  |
| --- | --- |
| Data Structure | Run Time (seconds) |
| Hash Set | 68.329825042 |
| Tree Set | 2.24389249 |
| Array List | 1.876302259 |
| Linked List | 5.915604884 |
| Linked Hash Set | 14.931852127 |

Full results are available in benchmarkdata.txt in the Git repo. You can also generate the data yourself using the Benchmark class’s main method, generation takes about 2.5 minutes on my laptop.

Array List is the winner because of its O(1) access, followed by Tree Set needing only O(log(n)). Linked List runs a bit slower since access is O(n). Due to the way the code is optimized, Hash Set and Linked Hash Set need an extra O(n) pass over all their contents to check for a base case in the recursion. Linked Hash Set is faster at this pass than Hash Set because it maintains an extra Linked List for fast iteration.

# Code Summaries

## Boggle Board

Implements a game of Boggle. The GameBoard class represents the game board and provides various methods for loading the dictionary and the Boggle grid, and finding valid words on the grid.

The class has a constructor that takes a collection as the dictionary, and the dictionary can be loaded using the loadDictionary method, which reads a file containing words and adds them to the dictionary collection. The longest word length in the dictionary is also determined.

The Boggle grid can be loaded using the loadGrid method, which reads a file and creates a two-dimensional array of Cell objects. Each Cell represents a letter on the Boggle grid and contains the row, column, and letter information.

The main functionality of finding valid words on the grid is implemented in the private method recursiveSearch. This method recursively explores the grid to construct words and checks if the words are valid by comparing them with the dictionary. The search can be performed with or without considering diagonal movements based on the isFourWay parameter.

The findWords method initiates the search process by iterating over each cell in the grid and calling recursiveSearch to find words starting from that cell. The found words are stored in a synchronized set and returned as the result.

## Word Search CLI

Program for searching a Boggle board for possible words and benchmarking different types of collections. The main method takes four arguments: the ruleset (4-way or 8-way), the file containing the Boggle board grid, the file containing the word list, and the type of collection to use for storing the found words (ArrayList, LinkedList, HashSet, TreeSet, or LinkedHashSet). If a fifth argument is provided and is equal to "q", then the program will not print the total number of words and words found, only the run time.

The program first creates a GameBoard object using the collection type specified in the arguments, and loads the dictionary and the grid from the corresponding files. It then calls the findWords method of the GameBoard object with the ruleset specified in the arguments, which returns a set of all the words found on the grid. The program then prints the words found and the total number of words found (unless the fifth argument is "q"), as well as the run time of the program.

## Benchmark

A utility for running a sequence of benchmarks. It performs benchmarking on different combinations of rules, grid files, dictionary files, and collection types.

The program contains four arrays: RULES, GRIDS, DICTIONARIES, and COLLECTION\_TYPES. These arrays store the possible values for rules (4way, 8way), grid files, dictionary files, and collection types, respectively.

The main method of the Benchmark class executes the benchmark. It starts by recording the current time using the Instant class. Then, it iterates over each combination of rules, grid, dictionary, and collection type using nested loops. For each combination, it calls the main method of the WordSearchCLI class with the appropriate arguments, including the "q" argument to suppress word printing. After each benchmark run, it prints a newline character.

Once all the benchmarks are completed, the program calculates the total run time by subtracting the start time from the current time using the Duration class. Finally, it prints the total run time in the format minutes:seconds.nanoseconds.

## Lab Exercise Tests

The class contains three test methods: loadDictionary(), loadGrid(), and cellMethods().

The loadDictionary() method tests that the GameBoard class can correctly load a dictionary file into a collection of strings. It does this by calling the doLoadDictionary() method four times, each with a different collection implementation, and checks that all the words in the dictionary file are present in the collection.

The loadGrid() method tests that the GameBoard class can correctly load a grid file into a 2D array of GameBoard.Cell objects. It does this by calling the doLoadGrid() method five times, each with a different grid file, and checks that the resulting 2D arrays match the expected ones.

The cellMethods() method tests various methods of the GameBoard.Cell class. It first loads a grid file using the doLoadGrid() method and retrieves the 2D array of cells using reflection. It then iterates over all the cells in the array and asserts that their default hash code is not equal to their identity hash code. It then loads the same grid file again into a separate GameBoard object and retrieves its 2D array of cells. It then iterates over both arrays and asserts that each pair of cells at the same position are equal.

## Search Tests

This is another JUnit test class for testing the search functionality of the game board implementation. The class contains a single test method called findWords(), which tests the findWords() method of the GameBoard class.

The findWords() method performs various assertions to check if the search functionality is working correctly. It tests both the 4-way and 8-way search modes for different grid sizes (2x2, 2x3, 3x3, 4x4, and 6x6) and different collection types (TreeSet, ArrayList, LinkedList, and LinkedHashSet).

For each combination of search mode, grid size, and collection type, the method compares the set of words found on the game board with the expected set of words read from a corresponding test output file. If the sets match, the search is considered successful. Otherwise, an assertion failure is thrown with a message indicating the failed search scenario.

The test method uses the doFindWords() helper method to set up a GameBoard object, load a grid file, load a dictionary, and perform the word search. The collection type is determined based on the input parameter and is used to instantiate the appropriate collection implementation for the game board.

# Learning

* Learned how to use the Set and Navigable Set interfaces.
* Learned how to optimize an algorithm based on the type of data structure being used.
* Got to test multiple revisions of an algorithm until I eventually got to something that runs quickly for everything.
* Got some difficult practice with determining all the possible base cases in a recursive algorithm.

# Feedback

* My Git repo has an extra “scowl.txt” file in the data directory. That has a much better list than words.txt.
* The lab instructions say to hardcode a word size limit of 15; that should instead be calculated when the dictionary is loaded.
* The Cell class should be a record.
* More time should have been spent on the Set and Navigable Set interfaces in class. All the binary search trees we have done are Navigable Sets and could have been good examples.
* If you expect to see improvement over the term, giving us feedback on what we do would be helpful. This is Lab 9, and I still have no idea how I did on Lab 3.