1 Overview

In this assignment you will implement an efficient word search puzzle solver based on hash table data structure. Word search puzzle is a word game that consists of letters placed in a rectangular grid. The objective is to find all valid English words hidden inside the grid. The words may be placed horizontally, vertically, or diagonally. The direction of the word can be both left-to-right and right-to-left. A small word search puzzle is in Fig. 1(a).

Your program should be named WordSearch and it should take as command line arguments two file names. The first file stores the puzzle, each puzzle row on a separate line, with no white space between the characters. The second file stores English words, one per line. Example program invocation:

```
WordSearch puzzle.txt dictionary.txt
```

Your program should find all hidden words that exist in the provided dictionary and that are at least four characters long. The words found should be saved to file output.txt, one per line, in any order. For the puzzle in Fig. 1(a), the output file contents should be:

```
NOSE
BODY
HEAD
```

Please do not put anything extra in the output, as we will use an automatic program to check your output. The list of output words should be duplicate-free.

2 Implementation

You are to implement the program using a hash-table based dictionary. First you should read all dictionary words from file dictionary.txt and store them in the hash table, named, say, Dict. The word itself should be used as a key, thus the key type should be String. In the simplest implementation, you do not need any value associated with the key. Therefore your value field can be of any type, and you can simply store null in the value field for each dictionary entry.

![Figure 1: (a) Word puzzle example. Hidden words are highlighted: nose, body, and head. (b) Eight search directions from letter E highlighted in red.](https://en.wikipedia.org/wiki/Word_search)
Then read the puzzle from puzzle.txt and store it in a two-dimensional array of type char. Let us call it the puzzle array. There are 8 possible orientations words can be hidden at: horizontal left-to-right, horizontal right-to-left, vertical top-to-bottom, vertical bottom-to-top, etc., see Fig. 1(b). Loop through each letter in the puzzle array. For each letter, loop through all eight orientations, searching for words that start with that letter and follow in the current orientation. For example, for the letter E highlighted in Fig. 1(b) and the vertical upward orientation, you should check whether strings E, EY, EYG are valid words by searching for them in dictionary Dict. Ignore all words that are shorter than 4 characters, even if they are in the dictionary Dict.

You need to store the words found in the puzzle array in a data structure that checks for presence of duplicates efficiently. For this purpose, you can use another instance of hash dictionary, let us call it puzzleWords. Before inserting a newly found word in puzzleWords, check if it is already there. Note that after you find some valid English word, you should continue with the loop in the current direction until the puzzle array ends. Otherwise you will never find the word FISHERMAN, as you will stop the loop after finding the word FISH.

2.1 Hints
To loop over all orientations, you can create a two dimensional array of size 2 by 8, storing orientations, say named o[2][8]. Each orientation is given by vertical and horizontal components. Orientations are (-1,1), (-1,0), (-1,1), (0,1), (0,1), (1,1), (1,0), (1,1). Orientation (-1,0), for example, means that the horizontal component needs to be changed by -1, and vertical by 0. This corresponds to searching at a horizontal orientation in right-to-left (backwards) manner, or the green arrow in Fig. 1(b).

You can initialize orientation array as: o[0][0] = -1, o[0][1] = 1, o[1][0] = -1, o[1][1] = 0, etc. To loop through orientations, loop through the orientation array along the second dimension (of size 8). Suppose current iteration is i = 5. You should use o[0][5] for the horizontal, and o[1][5] for the vertical increment to add to the current puzzle array indexes in order to follow orientation i = 5. Make sure to stay within the array bounds.

2.2 Optional Faster Implementation based on Prefixes
To have an even faster implementation of WordSearch, in addition to inserting all English words, you can also insert their prefixes. For example, for word FISHERMAN, also insert into Dict prefixes FISHERMA, FISHERM, FISHER, FISHE, etc. To distinguish prefixes from valid English words, for each English word, insert, say Integer with value 0 in the value field of the corresponding dictionary entry, and for a prefix, insert Integer with value 1. Suppose you are reading the word puzzle starting with some letter, say E, and currently you have read EOG. If the dictionary Dict does not contain either EOG as an English word, or EOG as a prefix of an English word, then there is no English word that starts with EOG. Therefore you can stop the loop that is reading in the current direction, saving computations.

In this implementation, it may happen that you insert a prefix FISH first before you insert an actual word FISH. Since no duplicates are allowed, you should delete prefix FISH and insert word FISH in such circumstances. Otherwise the word FISH will never be found in the puzzle.

3 Classes Provided
3.1 Dictionary (K,V)
The interface your dictionary should implement.

3.2 HashCode
Interface for the class implementing the HashCode object.
3.3 DefaultHashCode

You do not have to implement the default constructor for the HashDictionary, but if you wish to do so, the DefaultHashCode class is useful.

3.4 TestHashDictionary

This is a program which the TA will use to test your hash table implementation. Compile and run it after you have implemented your HashDictionary class. It will run some tests and will let you know which tests are passed/failed. Each test is worth 5 marks.

4 Classes Partially Implemented

4.1 HashDictionary< K,V >

This class should implement hash-table based dictionary. It must implement the provided Dictionary interface. You have to implement open addressing with double hashing strategy. I recommend starting with a hash array of size 7. When the load factor gets larger than the maximum allowed load factor, rehash. When rehashing, increase the hash array size to the next prime number at least twice larger than the current hash array size. You must design your hash function so that it produces few collisions. A bad hash function causing many collisions will result in lowering of your mark.

The hash table uses an external object to compute the hash code. I put some code in to illustrate how this is done. The constructor gets passed an object inputCode of class HashCode as an input. Class HashDictionary has a private variable hCode of type HashCode. In the constructor, assign hCode = inputCode. Now the object hCode has the method for computing the hash code. Inside the methods of HashDictionary, to compute a hash code for a key, use int hC= hCode.giveCode(key). See partially implemented method find(K key) for an illustration.

You must implement the following public methods, and all the other methods which you might implement for this class must be private. Also any member variables must be private.

- public void insert(K key, V value) throws DictionaryException. Inserts a new entry (key, value) in the Dictionary if key is not present in the dictionary. Otherwise, throws an exception.
- public Entry <K,V> find(K key). If there is an entry with key in the dictionary, returns this entry. Otherwise returns null.
- Iterator <Entry(K,V)> elements(). Returns an Iterator over all dictionary entries. The iteration is over objects of class Entry(K,V). You can use java.util.Iterator which gives the Iterator interface. You can create a LinkedList l, insert all entries of the hash table into l, and then return l.listIterator().

You have to be careful when using iterators. Say you created an iterator it over some object myDict. While you are using this iterator, you cannot modify myDict as then the iterator it would become invalid. You can modify myDict only after you stopped using iterator it.
- void remove(K key) throws DictionaryException. Removes entry with specified key. Throws exception if no such entry with this key exists.
- public float averNumProbes(). Returns an average number of probes performed so far. Count and store the total number of operations performed by the hash table in variable, say, named numOps. Each invocation of find, insert, remove should increase numOps by 1. Also count and store the total number of probes performed so far in a variable, say numProbes. When averNumProbes()
is called, it should return \((float) \text{numProbes})/\text{numOps}. As you decrease the maximum allowed load factor, the average number of probes should go down. When you run TestHashDictionary program, it will run your hash table at different load factors and will print out the average probe numbers versus running time. The average probe number should go up slightly as the maximum load factor increases. With a good implementation, the average number of probes should be between 3 and 5.

5 Classes to Implement Fully

5.1 DictionaryException

This exception should be thrown by your dictionary in case of unexpected conditions, see section 5 for cases in which to throw this exception.

5.2 StringHashCode implements HashCode\langle String\rangle

This class computes hash code for strings using the polynomial accumulation method. It should implement HashCode. Only the following public method is allowed:

\[
\text{public int giveCode(String key)}
\]

5.3 Entry \langle K, V \rangle

Objects of this class are stored in the dictionary. Only the following public methods are allowed:

- \[
\text{public Entry(K k, V v). Constructor that takes as an input initializations for key k and value v.}
\]
- \[
\text{public K Key(). Returns key.}
\]
- \[
\text{public V Value(). Returns value.}
\]
- \[
\text{public void modifyValue(V v). Changes value to v.}
\]

5.4 WordPuzzle

Program for solving a word puzzle. Before you instantiate an object of type HashDictionary, instantiate an object of type StringHashCode with \(hC = \text{new StringHashCode()}\) and pass it to the constructor of HashDictionary.

6 Other Files Provided

- \text{dictionary.txt}: valid English words.
- \text{puzzleSmall.txt}: small size word puzzle.
- \text{puzzleSmall.out.txt}: correct output on puzzleSmall.
- \text{puzzleMedium.out.txt}: correct output on puzzleMedium.

7 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be chosen to reflect their purpose in the program.
- Comments, indenting, and whitespace should be used to improve readability.
• No variable declarations should appear outside methods (“instance variables”) unless they contain data which is to be maintained in the object from call to call. In other words, variables which are needed only inside methods, whose value does not have to be remembered until the next method call, should be declared inside those methods.

• All variables declared outside methods (“instance variables”) should be declared private (not protected) to maximize information hiding. Any access to the variables should be done with accessor methods (like key() and value()).

8 Built-in Java Classes

You are allowed to use the following built-in Java classes and interfaces: LinkedList, Iterator, String, Integer, ArrayList, any file input/output libraries such as BufferedReader. Contact me before using any other Java built-in class.

8.1 Grading

Your grade will be computed as follows.

• Program compiles, produces a meaningful output: 15 marks
• HashDictionaryTest pass: 45 marks, 9 tests, each test is worth 5 marks.
• Coding Style: 10 marks
• HashDictionary implementation: 20 marks. If you do not implement double hashing, you will lose at least 15 marks.
• WordSearch program implementation: 10 marks.

9 Handing In Your Program

Submit your assignment via OWL.