Project 2

Bulls and Cows Game

CIS 18C Section 42030

Name: Marlo Zeroth

Date: 06/10/2015

**Introduction**

Title: Bulls and Cows Game

Bulls and Cows is a code breaking game similar to Mastermind, but was developed earlier. When Bulls and Cows begins the player is presented the option of choosing among easy, medium and hard difficulty. The difficulty determines playing with or without duplicates letters in the puzzle and the length of puzzle word. The puzzle words are isograms. Isograms are words where each letter is repeated only once or if a letter is repeated more than once all letters must repeat by the same amount. A valid puzzle word, for example, is anna which has the letters ‘a’ and ‘n’ repeated twice and is an isogram. Cows are letters that are in the puzzle, but not in the right spot. Bulls are letters that are in the puzzle in the right spot. The player can enter letters from A to Z without regard for case. After the player enters his/her choices the program creates a random code for the player to break. The player can enter up to 8 guesses to win the game. If a letter of the guess is in the puzzle, but not in the right spot the program prints to the screen the number of cows. If the player entered a letter that is in the puzzle in the right position, the number of bulls is printed to the screen. If no letter of the guess are in the puzzle 0 is printed for cows and also 0 for bulls. If the players runs out of turns before the game ends, the player looses. The player is given the choice of starting another game after he/she breaks the code or runs out of guesses. The game is fun to play and I wanted to see if I could create a program that could take words and take the user input and check it against the code the computer chose from a dictionary.

**Summary**

Project size:

Number of variables: About 47 major variables. The most important where in my classes. I used a lot of MyMap<String> types to calculate whether a word is an isogram and calculate whether a guess and the puzzle had similar letters and if they were in the correct position. Trees were used to create the dictionary and to get the puzzle for each game. The most important class was MyMap class and the most important object is the mainDictionary instantiated from the MyTree<String> class. Storing all the words in one dictionary really simplified my code in main for the Game class and allowed me more flexibility. The only trade of from using MyTree object instead of MyMap object as the dictionary is that the game takes a few tenths of a second to actually start. The only things changed from project one were the dictionary and the actual fillDictionary method.

Number of methods: 69

I implemented most of the concepts we have learned up to hashing from the Carrano book and the lectures. The game still needs to be completed. It still needs the ability to save and load games from binary files, to play at the last save point, and a graphical user interface instead of the console but with the use of the generic classes it will make it much easier to implement a two player version with a graphical user interface.

The program itself heavily uses a lot of the concepts for linked lists and generics. Particularly useful was the chapter on generics and linked lists. However, it was very frustrating to create an array of generic objects. This is apparently a limitation of java and some answers on stackoverflow.com suggested to use a linked list instead. Dr. Lehr also suggested I create an array of Strings using the toString() method that every object inherits. This version does not have an implemented write to file function. However, with the toString methods it should be much easier to implement. It does, however, use files to input a dictionary to generate puzzle codes. It was an added benefit that should help when I create a gui for the game.

**Description**

The main point of the game is how it functions when a player enters a guess and how the program delivers output the player can use to base their next guess.

**Pseudo Code**

Initialize

Get user input for game play choices for difficulty.

Set size of puzzle and guess to size specified by the difficulty given by player.

If the player chooses medium

Generate a puzzle word that is a length of 4 letters with no duplicate letters in the answer

Else if the player choose hard

Generate a puzzle word with duplicates and length of 6 words.

Else

Generate a puzzle word with 3 letters only and no duplicate letters.

Begin game loop

Get the users guess.

Compare guess to puzzle word.

If guess matches puzzle word exactly

Show the puzzle to the screen, the number of guesses, and a message that player won.

Else if guess does not match answer and guesses are less than 8

Print out dummy puzzle answer made up of ?s, print out cows and bulls for guesses,

and print out guess

Go next loop iteration for next guess

Else

Print out puzzle word, print out cows and bulls for guesses, print guesses,

and message for better luck next time.

End game loop and ask user if they want to start a new game.

If user chooses to play again

Start new game

Else

Exit the game loop.

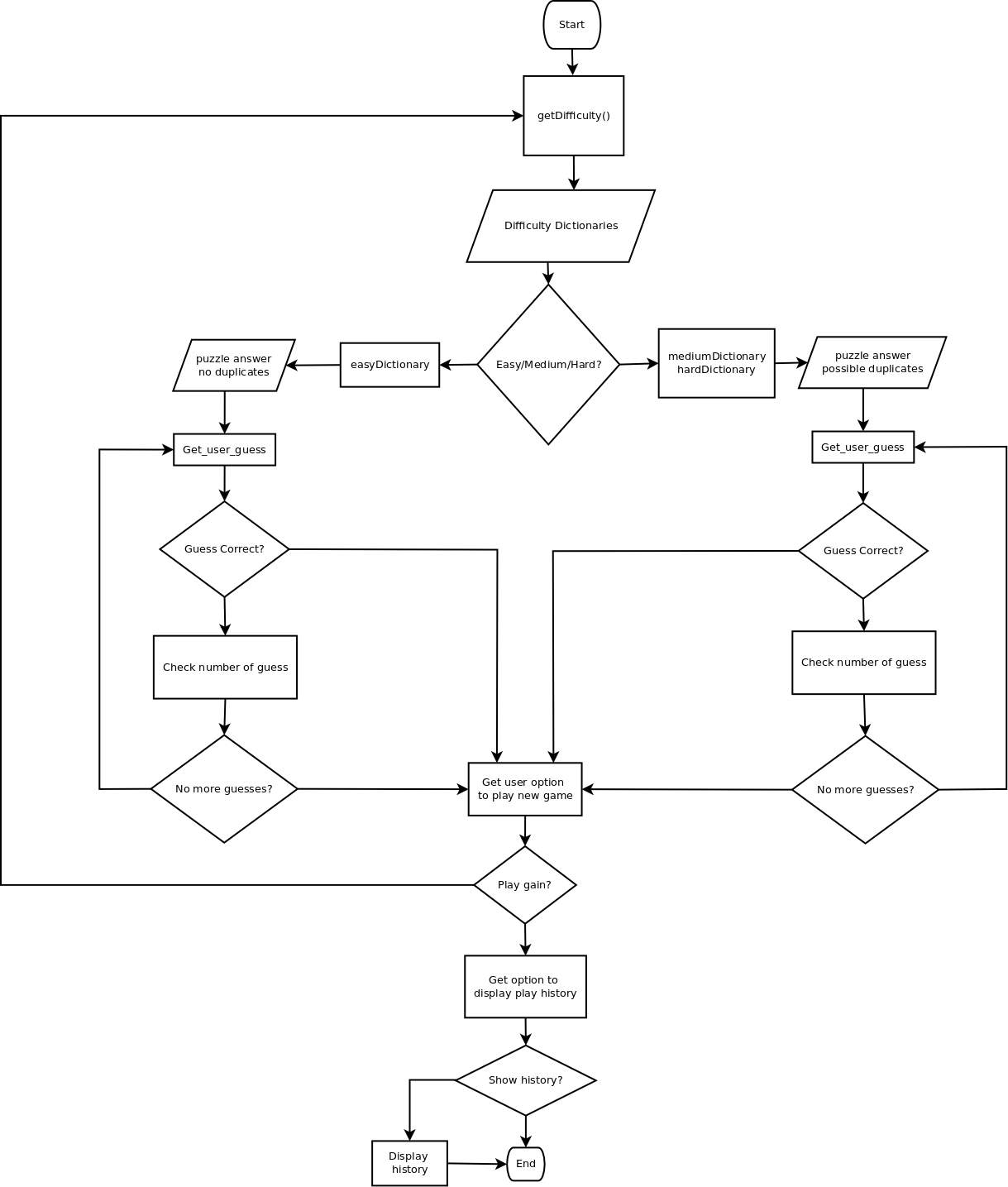
If user chooses to show play history

display play history

else

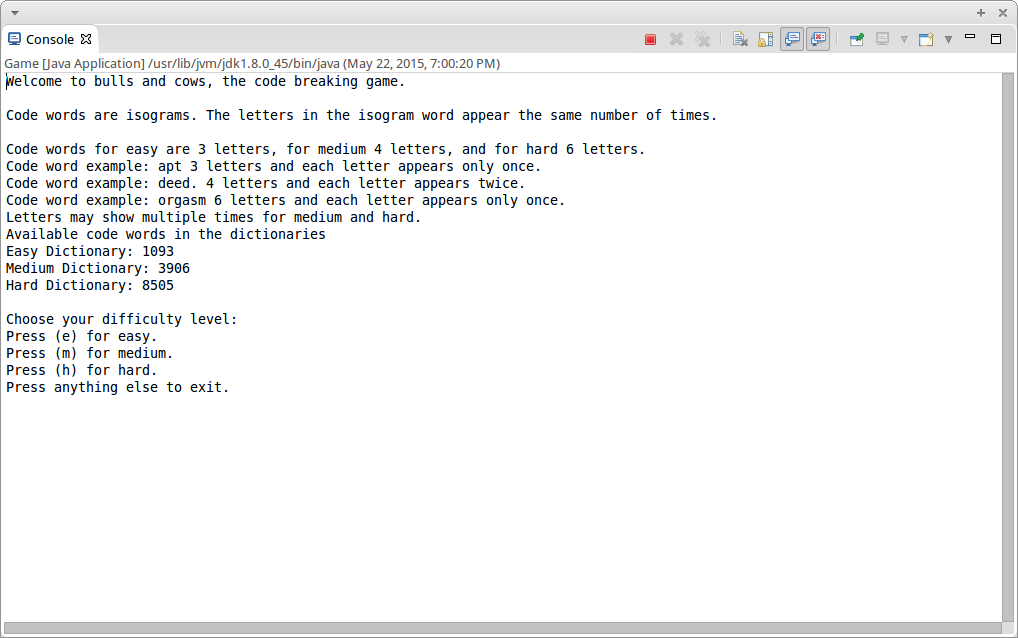
Exit program

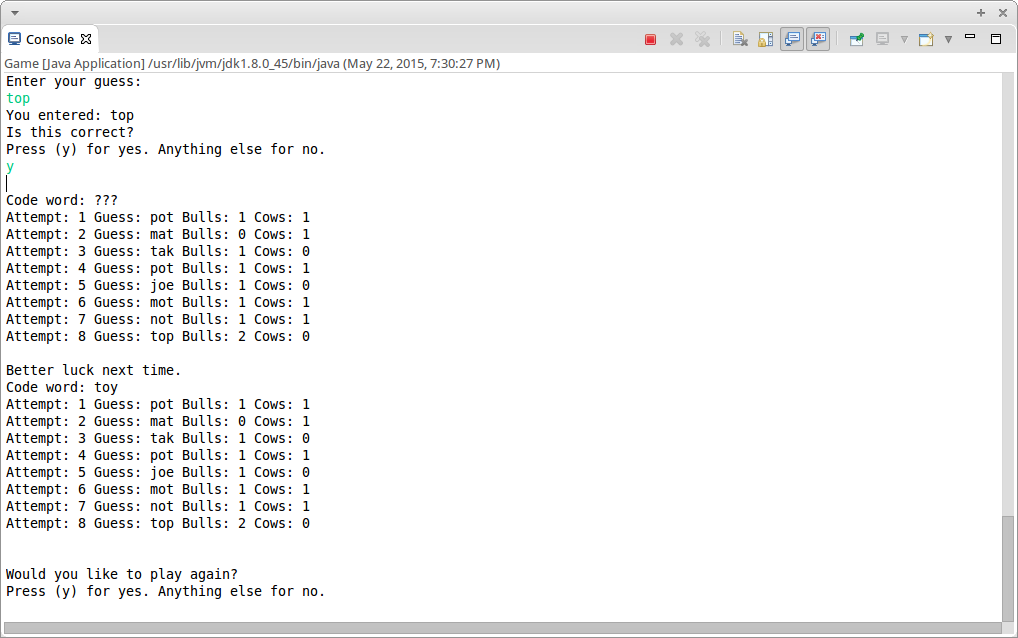
**Flow chart**

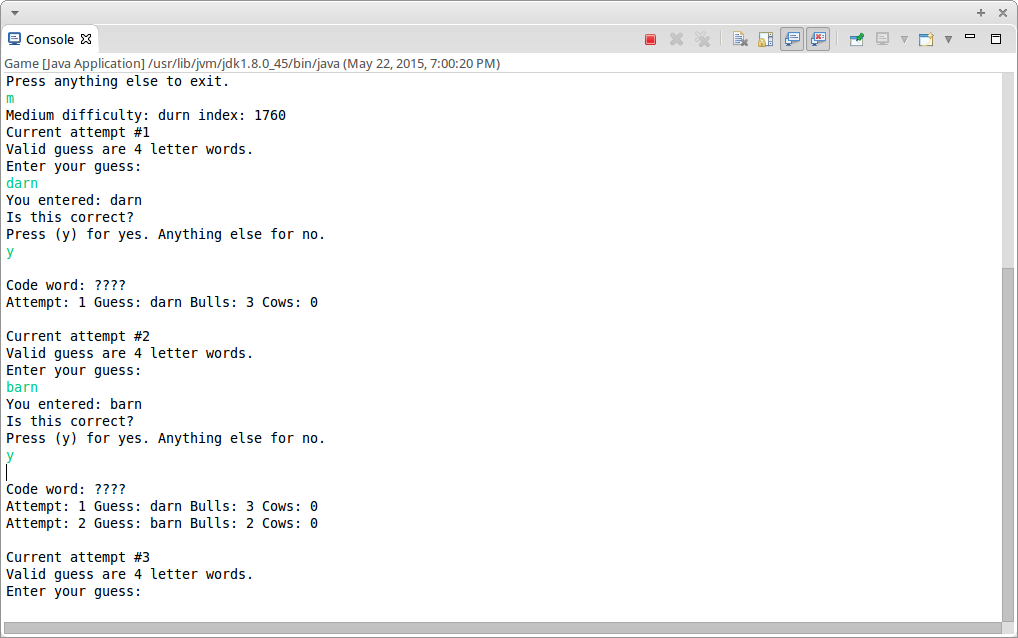


**Screen Shots**

Regular program flow.







**Major Variables**

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Variable Name | Description | Location |
| MyMap<Key,Value> | puzzleMap  guessMap | Holds unique characters in a puzzle word. Used to calculate bulls and cows in guess. It is also underlying structure for MySet class | GamePlay.java under method cows(String, String) and in MySet.java. Has its own class. |
| MySet<Key> | intersection | Used to get the number of bulls and cows in a guess compared with a code word | GamePlay.java file under method cows(String, String). Has its own class |
| MyTree<E,Difficulty> | mainDictionary | Holds all the words available for the game. | GamePlay.java file under main and in MyFileReader.java under fillDictionary() |
| MyStack<E> | playHistory | Holds the player history | In main in the GamePlay.java file. Has its own class |
| MyQueue<E> | guessQueue | Holds the user inputed guesses | In main in the GamePlay.java file passed to guessLoop(int, String, MyQueue<E>). Has its own class |
| MyLinkedList<E> | easySetting  mediumSetting  hardSetting | Hold the puzzle answer randomly generated using numbers. Also used to export keys for MySet and MyMap classes | In main in the GamePlay.java file. Has its own class. |
| Game<> | currentGame | Holds the current game contents | In main in the GamePlay.java file |
| Guess | newGuess | Holds a new guess and the numbers for bulls and cows | In main in the GamePlay.java file passed to guessQueue.enqueue(newGuess) |
| LinkedListMap | hashTable [] | Used as underlying bucket for hash table in MyHashMap | MyHashMap.java |
| String | puzzle | Temporarily holds the code word | In GamePlay.java under main, getGuess(), validGuess, cows() |
| String | guess | Temporarily holds the users guess | In GamePlay.java under main, getGuess(), validGuess, cows() |
| Int | size | Holds the size of a given datastructure | Most data structure classes use it e.g. MyLinkedList, MySet, MyMap |

**Java Data Structures**

|  |  |  |
| --- | --- | --- |
| Data Structure | New syntax and keywords | Location |
| Linked Lists | MyLinkedList<E> list = new MyLinkedList<E>(); | Underlying structure for MyMap.java and used for word list in GamePlay.java |
| Recursion | getGuess(guess, puzzle); | getGuess(String,String) in GamePlay.java |
| Hashing | index = hash(key); | Underlying method for indexing hash table in MyMap.java |
| Queue |  |  |
| Set | MySet<Character> intersection = puzzleMap.set().retainAll(guessMap.set()); | intersection variable in GamePlay.java |
| Map | MyMap<Key,Value> map = new MyMap<Key,Value>(); | puzzleMap, guessMap in GamePlay.java  keySet in MySet.java |
| Generics | Guess<E>,Game<Guess<E>> | Used in all classes |
| Stacks | MyStack<Game<String>> playHistory = new MyStack<Game<String>>(); | playHistory variable in GamePlay.java |
| Trees | MyTree<E,Difficulty> tree = new MyTree() | Underlying structure for the dictionary used to get code words for the player to break in GamePlay.java |

**Reference**

1. Textbook
2. stackoverflow.com

**Program**

/\*\*

\* @File GamePlay.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

import java.io.File;

import java.util.Random;

import java.util.Scanner;

public class GamePlay {

public static final Scanner input = new Scanner(System.in);

/\*\*

\* Returns the chosen difficulty for the game.

\*

\* @return

\*/

public static int getDifficulty() {

// Print out a menu for the difficulty

System.out.println("Choose your difficulty level: ");

System.out.println("Press (e) for easy. ");

System.out.println("Press (m) for medium. ");

System.out.println("Press (h) for hard. ");

System.out.println("Press anything else to exit. ");

// Get the difficulty

String choice = input.nextLine();

choice = choice.toLowerCase();

switch (choice) {

case "e":

return 3;

case "m":

return 4;

case "h":

return 6;

default:

System.out.println("You entered: " + choice + " exiting now.");

return -1;

}

}

/\*\*

\* Returns the users input for an option

\*

\* @return option

\*/

public static String getOption() {

// Prompt for input

System.out.println("Press (y) for yes. Anything else for no.");

// Get the user's option

String option = input.nextLine();

// Change it to lower case

option = option.toLowerCase();

return option;

}

/\*\*

\* Gets user input recursively for a guess to solve the puzzle

\*

\* @param puzzle

\* @return guess

\*/

public static String getGuess(String puzzle, String option) {

String guess = "";

boolean valid = false;

// Keep getting user input until it is valid

while (!valid) {

guess = "";

System.out.println("Valid guess are " + puzzle.length()

+ " letter words." + " \nEnter your guess: ");

guess = input.nextLine();

valid = validGuess(guess, puzzle);

guess = guess.toLowerCase();

}

System.out.println("You entered: " + guess + "\nIs this correct?");

option = getOption();

if (option.equalsIgnoreCase("y")) {

return guess;

} else {

return getGuess(puzzle, option);

}

}

/\*\*

\* Prints a history of the guess made

\*

\* @param guesses

\* @param bulls

\*/

public static void printGuesses(MyQueue<Guess<String>> guesses) {

// Iterate through each guess and print the guess, the bulls, and cows

for (Guess<String> g : guesses) {

System.out.println(g);

}

}

public static void guessLoop(int tries, String puzzle,

MyQueue<Guess<String>> guessQueue, boolean won) {

String guess = "";

String option = "";

int guesses = 0;

// String to use as a dummy until all guesses are used or puzzle solved

String dummy = "";

for (int i = 0; i < puzzle.length(); i++) {

dummy = dummy + "?";

}

boolean solved = false;

MyMap<String, Integer> bullcows = null;

while (guesses < tries && !solved) {

// Reset the bulls and cows

bullcows = null;

// Get the user's guess

System.out.println("Attempt #" + (guesses + 1));

// Get the users guess

guess = getGuess(puzzle, option);

// Get the number of bulls and cows for the guess

bullcows = cows(guess, puzzle);

// Create a new guess to put in the guessQueue

Guess<String> newGuess = new Guess<String>(guess, guesses + 1,

bullcows.getValue("cows"), bullcows.getValue("bulls"));

guessQueue.enqueue(newGuess);

// Print the guess history with bulls and cows for each guess

System.out.println();

System.out.println("Code word: " + dummy);

printGuesses(guessQueue);

System.out.println();

if (guess.equals(puzzle)) {

solved = true;

}

// If guess was wrong go to the next attempt.

guesses++;

}

if (solved) {

System.out.println("You win!!");

System.out.println("Code word: " + puzzle);

printGuesses(guessQueue);

System.out.println();

} else {

System.out.println("Better luck next time.");

System.out.println("Code word: " + puzzle);

printGuesses(guessQueue);

System.out.println();

}

// Set whether the game was won or not.

won = solved;

}

/\*\*

\* Validates an user input for a guess. If input is valid returns true.

\*

\* @param guess

\* @param puzzle

\* @return boolean

\*/

public static boolean validGuess(String guess, String puzzle) {

if (guess.length() < puzzle.length()

|| guess.length() > puzzle.length()) {

return false;

}

int i = 0;

// Go through the guess and make sure it is all letters

while (i < guess.length()) {

// If any element is not a letter the guess is not valid

if (!Character.isLetter(guess.charAt(i))) {

return false;

}

i++;

}

// The guess is valid

return true;

}

/\*\*

\* Returns the total bulls and cows in a guess from passed puzzle

\*

\* @param guess

\* @param puzzle

\* @return MyMap<String, Integer>

\*/

public static MyMap<String, Integer> cows(String guess, String puzzle) {

// Map to return with bulls and cows

MyMap<String, Integer> bullcow = new MyMap<String, Integer>();

int cows = 0; // If letters are in the puzzle, but wrong spot

int bulls = 0; // If letters are in the puzzle and right spot

if (validGuess(guess, puzzle)) {

// Maps for the guess and puzzle

MyMap<Character, Integer> puzzleMap = new MyMap<Character, Integer>();

MyMap<Character, Integer> guessMap = new MyMap<Character, Integer>();

for (int i = 0; i < puzzle.length(); i++) {

if (guess.charAt(i) == puzzle.charAt(i)) {

bulls++;

}

// Fill both maps with the frequency value for each letter key

if (puzzleMap.contains(puzzle.charAt(i))) {

int count = puzzleMap.getValue(puzzle.charAt(i));

puzzleMap.insert(puzzle.charAt(i), count + 1);

} else {

puzzleMap.insert(puzzle.charAt(i), 1);

}

if (guessMap.contains(guess.charAt(i))) {

int count = guessMap.getValue(guess.charAt(i));

guessMap.insert(guess.charAt(i), count + 1);

} else {

guessMap.insert(guess.charAt(i), 1);

}

}

// Now get the intersection of the guess and puzzle sets

MySet<Character> intersection = puzzleMap.set().retainAll(

guessMap.set());

// The cows are those keys that are not in the right position

cows = intersection.size() - bulls;

}

bullcow.insert("bulls", bulls);

bullcow.insert("cows", cows);

return bullcow;

}

public static void main(String[] args) {

// TODO Auto-generated method stub

// Text file with words from american-english dictionary

// Found at http://www.andrew.cmu.edu/course/15-121/dictionary.txt

File file = new File("american-english.txt");

// Dictionaries with isograms ranging from easy to hard

MyMap<String, Integer> easyDictionary = new MyMap<String, Integer>();

MyMap<String, Integer> mediumDictionary = new MyMap<String, Integer>();

MyMap<String, Integer> hardDictionary = new MyMap<String, Integer>();

// Fill the three dictionaries

MyFileReader.fillDictionary(file, easyDictionary, 3);

MyFileReader.fillDictionary(file, mediumDictionary, 4);

MyFileReader.fillDictionary(file, hardDictionary, 6);

// Lists of puzzle words ranging from easy to hard

// Easy each letter in a word appears only once

MyLinkedList<String> easySetting = easyDictionary.keysTolist();

// Medium setting: each letter can appear once or twice in the word

MyLinkedList<String> mediumSetting = mediumDictionary.keysTolist();

// This is only for first order isogram (each letter appears only once)

MyLinkedList<String> hardSetting = hardDictionary.keysToList(1);

// For user option to continue game and get difficulty

String option = "";

// For guesses and puzzle

String puzzle = "";

// For difficulty

int difficult = 0;

// Index to randomly choose a letter from a dictionary

int index = 0;

// Default number of tries. Could be adapted to difficulty later

// More tries for more difficult puzzles?

int tries = 8;

int game = 0;

// Stack to keep track of played games

MyStack<Game<String>> playHistory = new MyStack<Game<String>>();

boolean won = false;

System.out.println("Welcome to bulls and cows,"

+ " the code breaking game.\n");

System.out

.println("Code words are isograms."

+ " The letters in the isogram word appear the same number of times.\n");

System.out.println("Code words for easy are 3 letters,"

+ " for medium 4 letters, and for hard 6 letters.");

System.out

.println("Code word example: apt 3 letters and each letter appears only once.");

System.out

.println("Code word example: deed. 4 letters and each letter appears twice.");

System.out

.println("Code word example: orgasm 6 letters and each letter appears only once.");

System.out

.println("Letters may show multiple times for medium and hard.");

System.out.println("Available code words in the dictionaries");

System.out.println("Easy Dictionary: " + easyDictionary.size());

// gameDictionary.print();

System.out.println("Medium Dictionary: " + mediumDictionary.size());

System.out.println("Hard Dictionary: " + hardDictionary.size());

System.out.println();

Random randomNumber = new Random(System.currentTimeMillis());

do {

won = false;

// Guess queue to be used later in the play history

MyQueue<Guess<String>> guessQueue = new MyQueue<Guess<String>>();

difficult = getDifficulty();

switch (difficult) {

case (3):

// Increase the number of games

game++;

// Get the index for the puzzle to use

index = randomNumber.nextInt((easySetting.size()) + 1);

// Set the puzzle from the easy dictionary

puzzle = easySetting.get(index).toLowerCase();

// System.out.println("Easy difficulty: " + puzzle + " index: "

// + index);

// Begin the guess loop for this game

guessLoop(tries, puzzle, guessQueue, won);

break;

case (4):

// Increase the number of games

game++;

// Get the index for the puzzle to use

index = randomNumber.nextInt((mediumSetting.size()) + 1);

// Set the puzzle from the medium dictionary

puzzle = mediumSetting.get(index).toLowerCase();

// System.out.println("Medium difficulty: " + puzzle +

// " index: "

// + index);

// Begin the guess loop for this game

guessLoop(tries, puzzle, guessQueue, won);

break;

case (6):

// Increase the number of games

game++;

// Get the index for the puzzle to use

index = randomNumber.nextInt((hardSetting.size()) + 1);

// Set the puzzle from the hard dictionary

puzzle = hardSetting.get(index).toLowerCase();

// System.out.println("Hard difficulty: " + puzzle + " index: "

// + index);

// Begin the guess loop for this game

guessLoop(tries, puzzle, guessQueue, won);

break;

default:

System.out.println("You didn't choose a valid option.");

break;

}

Game<String> currentGame = new Game<String>(game, won, puzzle, guessQueue);

playHistory.push(currentGame);

System.out.println("\nWould you like to play again?");

option = getOption();

} while (option.equalsIgnoreCase("y"));

System.out.println("Would you like to see your play history?");

option = getOption();

if (option.equalsIgnoreCase("y")) {

System.out.println("Play history\n");

while (!playHistory.isEmpty()) {

System.out.println(playHistory.pop().toString());

}

}

input.close();

}

}

/\*\*

\* @File Game.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

public class Game<E> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

//Default constructor

Game() {

this(0, false, null, null);

}

// Main constructor

Game(int game, boolean won, E puzzle, MyQueue<Guess<E>> guess) {

this.game = game;

this.won = won;

this.puzzle = puzzle;

if (guess != null) {

this.guesses = new MyQueue<Guess<E>>();

for (Guess<E> g : guess) {

this.guesses.enqueue(g);

}

} else {

this.guesses = guess;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Will set the guesses of a game

\*

\* @param guess

\*/

public void setGuesses(MyQueue<Guess<E>> guess) {

if (guess != null) {

// If the internal guesses are not empty

if (!guesses.isEmpty()) {

this.guesses = new MyQueue<Guess<E>>();

for (Guess<E> g : guess) {

this.guesses.enqueue(g);

}

}

// If they are, clear them then add the new guesses

else {

this.guesses.clear();

for (Guess<E> g : guess) {

this.guesses.enqueue(g);

}

}

}

else{

this.guesses = guess;

}

}

// Default setters and getters

public void setGameNumber(int game) {

this.game = game;

}

public void setWon(boolean won) {

this.won = won;

}

public void setPuzzle(E puzzle) {

this.puzzle = puzzle;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Returns a copy of the guesses in the game. Returns null if there are no guesses

\* or the guess queue is empty.

\* @return

\*/

public MyQueue<Guess<E>> getGuesses(){

if(!this.guesses.isEmpty()){

MyQueue<Guess<E>> myGuesses = new MyQueue<Guess<E>>();

for (Guess<E> g : this.guesses) {

myGuesses.enqueue(g);

}

return myGuesses;

}

return null;

}

/\* IDE created functions \*/

public int getGameNumber() {

return game;

}

public boolean isWon() {

return won;

}

public E getPuzzle() {

return puzzle;

}

/\* IDE created functions \*/

/\*\*

\* Returns a string with the play history of the saved game

\*/

public String toString(){

StringBuilder sb = new StringBuilder();

String s = "Game # "+game + " Puzzle: " + puzzle + " Won: "+ won + "\r\n";

sb.append(s);

for(Guess<E> g : this.guesses){

sb.append(g.toString()+"\r\n");

}

return sb.toString();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Puzzle that was issued for the game

private E puzzle;

// Save for mage number

private int game;

// Was the game won

private boolean won;

// History of the guesses made by the player

private MyQueue<Guess<E>> guesses;

}

/\*\*

\* @File Guess.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

public class Guess <E>{

// Default constructor

Guess(){

data = null;

bulls = 0;

cows = 0;

}

// Main constructor

Guess(E data,int attempt, int cows, int bulls){

this.data = data;

this.attempt = attempt;

this.cows = cows;

this.bulls = bulls;

}

/\*\*

\* Return the number of cows associate with the guess

\* @return

\*/

public int getCows(){

return cows;

}

/\*\*

\* return the number of bulls associated with this guess

\* @return bulls

\*/

public int getBulls(){

return bulls;

}

/\*\*

\* Return the guess data

\* @return data

\*/

// Return the guess data

public E getGuess(){

return data;

}

public String toString(){

String s = "Attempt: " + attempt+" Guess: " + data.toString() + " Bulls: "+bulls + " Cows: "+cows;

return s;

}

private E data;

private int attempt;

private int bulls;

private int cows;

}

/\*\*

\* **@File** MySet.java

\* **@author** Marlo Zeroth

\* **@date** May 18, 2015

\* **@course** CSC18C DataStructures

\*/

**package** edu.rcc.datastructures;

**public** **class** MySet<Key **extends** Comparable<? **super** Key>> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Main Constructor

@SuppressWarnings({ "unchecked", "rawtypes" })

MySet() {

keySet = **new** MyMap();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Inserts a key into the set

\*

\* **@param** key

\*/

**public** **void** insert(Key key) {

keySet.insert(key, **null**);

}

/\*\*

\* Adds a set to the existing set if the passed set is not already present

\* in the existing set.

\*

\* **@param** set

\*/

**public** **void** addSet(MySet<Key> set) {

MyLinkedList<Key> setsKeys = set.toList();

**for** (Key key : setsKeys) {

keySet.insert(key, **null**);

}

}

/\*\*

\* Removes a key from the set

\*

\* **@param** key

\* **@return**

\*/

**public** **boolean** delete(Key key) {

**return** keySet.deleteKey(key);

}

/\*\*

\* Clears the set of all keys

\*/

**public** **void** clear() {

keySet.clear();

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Returns if a key is in the set

\*

\* **@param** key

\* **@return**

\*/

**public** **boolean** contains(Key key) {

**return** keySet.contains(key);

}

/\*\*

\* Returns the set that is intersection of this object's set and the passed

\* set.

\*

\* **@param** testSet

\* **@return**

\*/

**public** MySet<Key> retainAll(MySet<Key> testSet) {

// Lists to iterate through the keys in the sets

MyLinkedList<Key> setTest = testSet.toList();

MyLinkedList<Key> thisSet = keySet.keysTolist();

MySet<Key> intersect = **new** MySet<Key>();

// Iterate through the keys in both sets and test if the keys are in the

// passed set.

**for** (Key key : setTest) {

**if** (thisSet.contains(key))

intersect.insert(key);

}

// Iterate through this object's set and test if its keys are in the

// passed set

**for** (Key key : thisSet) {

**if** (setTest.contains(key))

intersect.insert(key);

}

**return** intersect;

}

**public** MyLinkedList<Key> toList() {

**return** keySet.keysTolist();

}

/\*\*

\* Provides a deep copy of the keys in the set

\*

\* **@return**

\*/

**public** String[] toArray() {

String[] elements = **new** String[keySet.size()];

String[] mapElements = keySet.keysToArray();

**for** (**int** i = 0; i < keySet.size(); i++) {

elements[i] = mapElements[i];

}

**return** elements;

}

/\*\*

\* Returns whether the set is empty

\*

\* **@return** boolean

\*/

**public** **boolean** isEmpty() {

**return** keySet.isEmpty();

}

/\*\*

\* Returns the total number of keys in the set

\*

\* **@return** int

\*/

**public** **int** size() {

**return** keySet.size();

}

/\*\*

\* Print the keys

\*/

**public** **void** print() {

String[] array = keySet.keysToArray();

**for** (String s : array) {

System.***out***.println(s + " ");

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Uses a PuzzleMap as the underlying data structures.

// Similar to what the text book suggests

MyMap<Key, ?> keySet;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Nested Classes \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**public** **static** **void** main(String[] args) {

// Testing the insert function

System.***out***.println("Testing the insert function");

System.***out***.println("adding 1000 string entries of apple+i");

MySet<String> searchTable = **new** MySet<String>();

MySet<String> searchTable2 = **new** MySet<String>();

String s;

**for** (**int** i = 1; i < 5; i++) {

s = "apple" + i;

searchTable.insert(s);

}

**for** (**int** i = 3; i < 7; i++) {

s = "apple" + i;

searchTable2.insert(s);

}

MySet<String> intersection = searchTable.retainAll(searchTable2);

System.***out***.println("Is searchTable empty: " + searchTable.isEmpty());

System.***out***.println("Total entries: " + searchTable.size());

searchTable.print();

System.***out***.println();

System.***out***.println("Is searchTable2 empty: " + searchTable2.isEmpty());

System.***out***.println("Total entries: " + searchTable2.size());

searchTable2.print();

System.***out***.println("Is intersection empty: " + intersection.isEmpty());

System.***out***.println("Total entries: " + intersection.size());

intersection.print();

}

}

/\*\*

\* @File MyMap.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

//Used Key and Value instead of E or T for generic implementation. It is easier to follow.

public class MyMap<Key extends Comparable<? super Key>, Value extends Comparable<? super Value>> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Default constructor

public MyMap() {

// Call the main constructor

this(MINIMUM);

}

// Main constructor

@SuppressWarnings("unchecked")

public MyMap(int size) {

// Set the minimum size and allocate memory for hash table

this.tableSize = size;

hashTable = (LinkedListMap<Key, Value>[]) new LinkedListMap[size];

// Now create a bucked for each index

for (int i = 0; i < size; i++) {

// Now add a linked list as the bucket

hashTable[i] = new LinkedListMap<Key, Value>();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Takes a key value pair to add to the hash table. It checks the current

\* load factor and resizes the hash table if the current load factor is

\* greater than 1.0

\*

\* @param key

\* @param value

\*/

public void insert(Key key, Value value) {

// Before adding we need to check if the current load factor is greater

// than the preset load factor of 1.0

double currentLoad = ((double) this.keyPairs / (double) this.hashTable.length);

if (LOADFACTOR < currentLoad) {

int newSize = PRIMES[0];

int index = 1;

double newLoadFactor = this.keyPairs / (double) newSize;

// Find the next biggest prime that will restore the

// load factor back to less than 1.0

while (newLoadFactor > LOADFACTOR && index < PRIMES.length) {

// the next size up (size being a prime number)

if (newLoadFactor > LOADFACTOR) {

newSize = PRIMES[index++];

}

// Update the load factor

newLoadFactor = (double) this.keyPairs / newSize;

}

// Resize the hash table with the new size

resize(newSize);

}

// Find the index of the new entry

int index = hash(key);

// If the key is not in the hash table insert it at the appropriate

// index

if (!hashTable[index].contains(key)) {

keyPairs++;

hashTable[index].insert(key, value);

}

// Otherwise just update the value for the key

else {

hashTable[index].updateValue(key, value);

}

}

/\*\*

\* Deletes a key from the hash table. Returns the value of the deleted key

\* or null if the key was not found in the hash table. It will resize the

\* hash table if the load factor is less than 0.25 even if an entry is not

\* found in the hash table

\*

\* @param key

\* @return value

\*/

public Value delete(Key key) {

Value keyValue = null;

// Find the index of the new entry

int index = hash(key);

// If the key is in the hash table delete it

if (hashTable[index].contains(key)) {

keyValue = hashTable[index].delete(key);

// Decrease the number of keys stored in hash table

keyPairs--;

}

// Test if we need to resize.

// Only resize if the load factor is less than 0.25

double currentLoad = ((double) this.keyPairs / (double) this.hashTable.length);

if (this.hashTable.length > MINIMUM && currentLoad < 0.25) {

int newSize = PRIMES[PRIMES.length - 1];

int i = PRIMES.length - 1;

double newLoadFactor = this.keyPairs / (double) newSize;

// Find the next biggest prime that will restore the

// load factor back to less than 1.0 and greater than 0.75

while (newLoadFactor < 0.75 && i > 0) {

// the next size up (size being a prime number)

if (newLoadFactor < LOADFACTOR) {

newSize = PRIMES[i--];

}

// Update the load factor

newLoadFactor = (double) this.keyPairs / newSize;

}

// Resize the hash table with the new size

resize(newSize);

}

return keyValue;

}

public boolean deleteKey(Key key) {

boolean deleted = false;

// Find the index of the new entry

int index = hash(key);

// If the key is in the hash table delete it

if (hashTable[index].contains(key)) {

hashTable[index].delete(key);

deleted = true;

// Decrease the number of keys stored in hash table

keyPairs--;

}

// Test if we need to resize.

// Only resize if the load factor is less than 0.25

double currentLoad = ((double) this.keyPairs / (double) this.hashTable.length);

if (this.hashTable.length > MINIMUM && currentLoad < 0.25) {

int newSize = PRIMES[PRIMES.length - 1];

int i = PRIMES.length - 1;

double newLoadFactor = this.keyPairs / (double) newSize;

// Find the next biggest prime that will restore the

// load factor back to less than 1.0 and greater than 0.75

while (newLoadFactor < 0.75 && i > 0) {

// the next size up (size being a prime number)

if (newLoadFactor < LOADFACTOR) {

newSize = PRIMES[i--];

}

// Update the load factor

newLoadFactor = (double) this.keyPairs / newSize;

}

// Resize the hash table with the new size

resize(newSize);

}

return deleted;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* User Interface\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Clears the hash table of all key value pair entries. It does not resize

\* the hash table

\*/

public void clear() {

// Go through each linked list and clear it

for (int i = 0; i < hashTable.length; i++) {

hashTable[i].clear();

}

// Set the number of key pairs to zero

keyPairs = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Will return the value associated with the parameter key. Returns null if

\* the value is not in the hash table.

\*

\* @param key

\* @return

\*/

public Value getValue(Key key) {

// Get the location of the key in the hashTable

int index = hash(key);

// Return the value pair of the key.

// will return null if not in the hash table.

return hashTable[index].getValue(key);

}

/\*\*

\* Searches for a specified key in the hash table. Returns true if found,

\* false otherwise.

\*

\* @param key

\* @return

\*/

public boolean contains(Key key) {

// Find the index of the new entry

int index = hash(key);

// Find if it is in the linked list at this index

return hashTable[index].contains(key);

}

/\*\*

\* Returns true if the hash table is empty or false otherwise

\*

\* @return boolean

\*/

public boolean isEmpty() {

// If the tableSize is 0, the table is empty

return keyPairs == 0;

}

/\*\*

\* Returns the internal size of the hash table

\*

\* @return int

\*/

public int tableSize() {

return tableSize;

}

/\*\*

\* Returns the total number of keys in the hash table

\*

\* @return int

\*/

public int size() {

return keyPairs;

}

/\*\*

\* Returns an list of keys associated with a given value

\* @param value

\* @return MyLinkedList<String>

\*/

public MyLinkedList<Key> keysToList(Value value){

MyLinkedList<Key> elements = new MyLinkedList<Key>();

// Index to add elements to elements array

int index = 0;

// Iterate through the hash table to get the keys

for (int i = 0; i < hashTable.length; i++) {

// Only look at those buckets that are not empty

// Do not exceed the number of key pairs in the hash map

if (hashTable[i] != null && index < keyPairs) {

// Iterate through the keys at this bucket

for (Key key : hashTable[i]) {

if(getValue(key).equals(value))

elements.insert(key);;

}

}

}

return elements;

}

/\*\*

\* Returns a list of keys in the dictionary

\* @return MyLinkedList<Key>

\*/

public MyLinkedList<Key> keysTolist(){

MyLinkedList<Key> elements = new MyLinkedList<Key>();

// Index to add elements to elements array

int index = 0;

// Iterate through the hash table to get the keys

for (int i = 0; i < hashTable.length; i++) {

// Only look at those buckets that are not empty

// Do not exceed the number of key pairs in the hash map

if (hashTable[i] != null && index < keyPairs) {

// Iterate through the keys at this bucket

for (Key key : hashTable[i]) {

elements.insert(key);;

}

}

}

return elements;

}

/\*\*

\* Returns a string array of the values in the hash table

\* @return

\*/

// Work around until I figure out how to get the values

// in an array for generics

public String[] valuesToArray(){

String[] elements = new String[keyPairs];

// Index to add elements to elements array

int index = 0;

// Iterate through the hash table to get the keys

for (int i = 0; i < hashTable.length; i++) {

// Only look at those buckets that are not empty

// Do not exceed the number of key pairs in the hash map

if (hashTable[i] != null && index < keyPairs) {

// Iterate through the keys at this bucket

for (Key key : hashTable[i]) {

elements[index++] = getValue(key).toString();

}

}

}

return elements;

}

/\*\*

\* Returns an array of strings of the hash map keys

\*

\* @return

\*/

// Work around until I figure out how to get the values

// in an array for generics

public String[] keysToArray() {

String[] elements = new String[keyPairs];

// Index to add elements to elements array

int index = 0;

// Iterate through the hash table to get the keys

for (int i = 0; i < hashTable.length; i++) {

// Only look at those buckets that are not empty

// Do not exceed the number of key pairs in the hash map

if (hashTable[i] != null && index < keyPairs) {

// Iterate through the keys at this bucket

for (Key key : hashTable[i]) {

elements[index++] = key.toString();

}

}

}

return elements;

}

/\*\*

\* Returns a set of keys in the hash table

\* @return

\*/

public MySet<Key> set(){

MySet<Key> elements = new MySet<Key>();

// Index to add elements to elements array

int index = 0;

// Iterate through the hash table to get the keys

for (int i = 0; i < hashTable.length; i++) {

// Only look at those buckets that are not empty

// Do not exceed the number of key pairs in the hash map

if (hashTable[i] != null && index < keyPairs) {

// Iterate through the keys at this bucket

for (Key key : hashTable[i]) {

elements.insert(key);

}

}

}

return elements;

}

public void print() {

for (int i = 0; i < tableSize; i++) {

hashTable[i].displayList();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constants\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Minimum table size

private static final int MINIMUM = 5;

// Range of keys to create hash code by

private static final int RANGE = 31;

// Load factor to decide when to resize the hash table

private static final double LOADFACTOR = 1.0;

// Large prime numbers to be used as a table size

private static final int[] PRIMES = { 7, 13, 31, 61, 127, 251, 509, 1021,

2039, 4093, 8191, 16381, 32749, 65521, 131071, 262139, 524287,

1048573, 2097143, 4194301, 8388593, 16777213, 33554393, 67108859,

134217689, 268435399, 536870909, 1073741789, 2147483647 };

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private int keyPairs;// Total number of key pairs in the table

private int tableSize;// Size of the hash table

// Array to store linked list of entries

LinkedListMap<Key, Value>[] hashTable;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Utility Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Hash value between 0 and tableSize

private int hash(Key key) {

int hash = hashFunction(key, hashTable.length);

hash = hash % hashTable.length;

if (hash < 0)

hash = hash + hashTable.length;

// while (hash > hashTable.length) {

// hash = hash / 10;

// System.out.println("Dividing hash by 10 hash is now: " + hash);

// }

// System.out.println("Hash is:" +hash);

return hash;

}

// Resize the table when the number of entries exceed the size

private void resize(int newSize) {

// Resize only if the new size is greater than the required minimum

if (newSize > MINIMUM) {

MyMap<Key, Value> tempTable = new MyMap<Key, Value>(newSize);

for (int i = 0; i < hashTable.length; i++) {

{

// Only add those keys where the chains that are not empty

if (!hashTable[i].isEmpty()) {

for (Key key : hashTable[i]) {

// Put the key in the existing objects hash table in

// the

// new

// hash table

tempTable.insert(key, hashTable[i].getValue(key));

}

}

}

}

// Update the size of the current object table size

// and number of key pairs

this.tableSize = tempTable.tableSize;

this.keyPairs = tempTable.keyPairs;

// Change the reference to point to the new hash table

this.hashTable = tempTable.hashTable;

}

}

// Returning a double only to process hash in hash table

private int hashFunction(Key key, int size) {

double temp = 0;

String s = key.toString();

for (int i = 0; i < s.length(); i++) {

temp = RANGE \* temp + s.charAt(i);

// System.out.println("charAt(" + i + ") is " + s.charAt(i));

// System.out.println("Hascode is now: " + hash);

}

// Code from textbook. We assign the bits from temp to

// bits then shift the bits 32 bits to the right and cast to an int

long bits = Double.doubleToLongBits(temp);

int hash = (int) (bits ^ (bits >> 32));

return hash;

}

}

/\*\*

\* @File LinkedListMap.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\* The class is used to as hash table for HashMaps

\*/

package edu.rcc.datastructures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class LinkedListMap<Key extends Comparable<? super Key>, Value extends Comparable<? super Value>>

implements Iterable<Key> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Default Constructor \*/

public LinkedListMap() {

head = null;

listSize = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Adds an entry to the front of the list.

\*

\* @param entry

\*/

public void insert(Key key, Value entry) {

// Create a new node to the list

Node<Key, Value> node = new Node<Key, Value>(key, entry, head);

// node.next = head;

head = node;// The new node is the head now.

listSize++;

}

/\*\*

\* Deletes an entry from the list. It maintains the chronological order. if

\* the key is found it returns the value of the key, otherwise returns null

\*

\* @param entry

\* @return Value Returns the value if the key was deleted, null otherwise

\*/

public Value delete(Key entry) {

// To return if the entry was found and deleted.

boolean deleted = false;

Value found = null;

if (!isEmpty()) {

// Cursor node to traverse the list

Node<Key, Value> cursor = head;

Node<Key, Value> previousNode = null;

// Test the first node to see if the entry is there.

if (head.key.compareTo(entry) == 0) {

// Set the next node to be the new head/first node

head = head.next;

// Set the value to be returned

found = cursor.value;

// Delete the node with the entry

cursor = null;

deleted = true;

listSize--;

}

// If not then go through the rest of the nodes and delete the node

// that contains the key if it exists in the list.

else {

// Stop when we have we reached the end of the list

while (!deleted && cursor != null) {

if (cursor.key.compareTo(entry) == 0) {

// We found it set the boolean and the return found

// value

deleted = true;

found = cursor.value;

// Delete the node

previousNode.next = cursor.next;

// cursor = null;

listSize--;

} else {

previousNode = cursor;

cursor = cursor.next; // Go to the next node

}

}

}

}

return found;

}

/\*\*

\*

\* @param key

\* @param value

\*/

public void updateValue(Key key, Value newValue) {

// Cursor to traverse the list

Node<Key, Value> cursor = head;

// Boolean to break from while loop once we find the key

boolean found = false;

// Iterate through the list until we find the key or we are at the end

while (cursor != null && !found) {

// If we found the key update its value pair

if (cursor.key.equals(key)) {

cursor.value = newValue;

found = true;

}

// Search the next node

cursor = cursor.next;

}

}

/\*\*

\* Clears the linked list of all entries

\*/

public void clear() {

// If the list is not empty, then empty it.

if (!isEmpty()) {

head = null;

listSize = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Iterator to go through a linked list

\*/

public Iterator<Key> iterator() {

return new MyListIterator();

}

/\*\*

\* Returns a value associated with a key

\*

\* @param key

\* @return value

\*/

public Value getValue(Key key) {

// Cursor to traverse the list

Node<Key, Value> cursor = head;

while (cursor != null) {

// If we found the key return the value pair

if (cursor.key.equals(key)) {

return cursor.value;

}

// Search the next node

cursor = cursor.next;

}

// If not return that we don't have it

return null;

}

/\*\*

\* Get the data at a given position

\* @param position

\* @return

\*/

public Key get(int position) {

int index = 0;

Node<Key, Value> cursor = head;

Key key = null;

while (index < listSize && cursor != null) {

if (index == position) {

return (key = cursor.key);

}

index++;

cursor = cursor.next;

}

return key;

}

/\*\*

\* Returns a key associate with a particular key

\*

\* @param key

\* @return

\*/

public Key get(Key key) {

// Cursor to traverse the list

Node<Key, Value> cursor = head;

while (cursor != null) {

// If we found the key return the value pair

if (cursor.key.equals(key)) {

return cursor.key;

}

// Search the next node

cursor = cursor.next;

}

// If not return that we don't have it

return null;

}

/\*\*

\* Returns true if a key is in the list or false otherwise

\*

\* @param E

\* @param size

\* @return

\*/

public boolean contains(Key key) {

// If there is nothing in the list we return false

// If there is something we return true

return get(key) != null;

}

/\*\*

\* Returns whether there are any key pairs in the list

\*

\* @return

\*/

public boolean isEmpty() {

return (listSize == 0 && head == null);

}

/\*\*

\* Returns the total number of key pairs in the linked list

\*

\* @return

\*/

public int size() {

return listSize;

}

/\*\*

\* Returns an array of strings of the linked list

\*

\* @return String

\*/

public String[] toArray() {

// Allocate memory for the array

String[] result = new String[listSize];

// Node to traverse the list

Node<Key, Value> currentNode = head;

int index = 0;

// Traverse the list until we reach the end

while (index < listSize && currentNode != null) {

// Set array element value equal to list node value

result[index++] = currentNode.key.toString();

currentNode = currentNode.next;

}

return result;

}

/\*\*

\* Print the contents of the list in order

\*/

public void displayList() {

int index = 0;// So we don't exceed the number of entries in the list

Node<Key, Value> tempNode = new Node<Key, Value>();// To traverse the

// list

tempNode = head;

while (tempNode != null && index < listSize) {

System.out.println(tempNode.key + " " + tempNode.value);

tempNode = tempNode.next;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Total number key pairs in the linked list

private int listSize;

// Node to point to the beginning of the list

private Node<Key, Value> head;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Nested Classes\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Node nested class

private static class Node<Key, Value> {

/\* Node Constructors \*/

public Node() {

this(null, null, null);

}

public Node(Key searchKey, Value dataValue, Node<Key, Value> next) {

// Set the key and value variables equal to the input.

this.key = searchKey;

this.value = dataValue;

// Set the reference nodes to null. These will be changed in the

// list

// depending on where the node is inserted

this.next = next;

}

/\* Node member variables \*/

private Key key;

private Value value;

// The head will always have the previous node member variable null

// and the tail will always have its next node member variable null

private Node<Key, Value> next;

}

// Nested Iterator class

private class MyListIterator implements Iterator<Key> {

// Node to traverse the list

private Node<Key, Value> nextNode;

// Index to keep track of iterated node

private int index = 0;

// Default constructor

public MyListIterator() {

nextNode = head;

}

@Override

public Key next() {

// If there is not a next node throw an error

if (!hasNext()) {

throw new NoSuchElementException("Illegal call to next(); "

+ "iterator is after end of list");

}

// Otherwise return the current key and advance to the next node

Key returnKey = nextNode.key;

nextNode = nextNode.next;

index++;

return returnKey;

}

@Override

public boolean hasNext() {

// If the index is less than the total nodes in

// the list we have a next node

return index < listSize;

}

@Override

public void remove() {

throw new UnsupportedOperationException("remove() is not "

+ "supported by this iterator");

}

}

}

/\*\*

\* @File MyLinkedList.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class MyLinkedList<Key extends Comparable<? super Key>>

implements Iterable<Key> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Default Constructor \*/

public MyLinkedList() {

head = null;

listSize = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Adds an entry to the front of the list.

\*

\* @param entry

\*/

public void insert(Key key) {

// Create a new node to the list

Node<Key> node = new Node<Key>(key, head);

// node.next = head;

head = node;// The new node is the head now.

listSize++;

}

/\*\*

\* Deletes an entry from the list. It maintains the chronological order. if

\* the key is found it returns the value of the key, otherwise returns null

\*

\* @param entry

\* @return Value Returns the value if the key was deleted, null otherwise

\*/

public Key delete(Key entry) {

// To return if the entry was found and deleted.

Key found = null;

boolean deleted = false;

if (!isEmpty()) {

// Cursor node to traverse the list

Node<Key> cursor = head;

Node<Key> previousNode = null;

// Test the first node to see if the entry is there.

if (head.key.compareTo(entry) == 0) {

// Set the next node to be the new head/first node

head = head.next;

// Set the value to be returned

found = cursor.key;

// Delete the node with the entry

cursor = null;

deleted = true;

listSize--;

}

// If not then go through the rest of the nodes and delete the node

// that contains the key if it exists in the list.

else {

// Stop when we have we reached the end of the list

while (!deleted && cursor != null) {

if (cursor.key.compareTo(entry) == 0) {

// We found it set the boolean and the return found

// value

deleted = true;

found = cursor.key;

// Delete the node

previousNode.next = cursor.next;

// cursor = null;

listSize--;

} else {

previousNode = cursor;

cursor = cursor.next; // Go to the next node

}

}

}

}

return found;

}

/\*\*

\* Clears the linked list of all entries

\*/

public void clear() {

// If the list is not empty, then empty it.

if (!isEmpty()) {

head = null;

listSize = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Iterator to go through a linked list

\*/

public Iterator<Key> iterator() {

return new MyListIterator();

}

/\*\*

\* Returns a key associate with a particular key

\*

\* @param key

\* @return

\*/

public Key get(Key key) {

// Cursor to traverse the list

Node<Key> cursor = head;

while (cursor != null) {

// If we found the key return the value pair

if (cursor.key.equals(key)) {

return cursor.key;

}

// Search the next node

cursor = cursor.next;

}

// If not return that we don't have it

return null;

}

/\*\*

\* Get the data at a given position

\* @param position

\* @return

\*/

public Key get(int position) {

int index = 0;

Node<Key> cursor = head;

Key key = null;

while (index < listSize && cursor != null) {

if (index == position) {

return (key = cursor.key);

}

index++;

cursor = cursor.next;

}

return key;

}

/\*\*

\* Returns true if a key is in the list or false otherwise

\*

\* @param E

\* @param size

\* @return

\*/

public boolean contains(Key key) {

// If there is nothing in the list we return false

// If there is something we return true

return get(key) != null;

}

/\*\*

\* Returns whether there are any key pairs in the list

\*

\* @return

\*/

public boolean isEmpty() {

return (listSize == 0 && head == null);

}

/\*\*

\* Returns the total number of key pairs in the linked list

\*

\* @return

\*/

public int size() {

return listSize;

}

/\*\*

\* Returns an array of strings of the linked list

\*

\* @return String

\*/

public String[] toArray() {

// Allocate memory for the array

String[] result = new String[listSize];

// Node to traverse the list

Node<Key> currentNode = head;

int index = 0;

// Traverse the list until we reach the end

while (index < listSize && currentNode != null) {

// Set array element value equal to list node value

result[index++] = currentNode.key.toString();

currentNode = currentNode.next;

}

return result;

}

/\*\*

\* Print the contents of the list in order

\*/

public void displayList() {

int index = 0;// So we don't exceed the number of entries in the list

Node<Key> tempNode = new Node<Key>();// To traverse the

// list

tempNode = head;

while (tempNode != null && index < listSize) {

System.out.println(tempNode.key + " ");

tempNode = tempNode.next;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Total number key pairs in the linked list

private int listSize;

// Node to point to the beginning of the list

private Node<Key> head;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Nested Classes\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Node nested class

private static class Node<Key> {

/\* Node Constructors \*/

public Node() {

this(null, null);

}

public Node(Key searchKey,Node<Key> next) {

// Set the key and value variables equal to the input.

this.key = searchKey;

// Set the reference nodes to null. These will be changed in the

// list

// depending on where the node is inserted

this.next = next;

}

/\* Node member variables \*/

private Key key;

// The head will always have the previous node member variable null

// and the tail will always have its next node member variable null

private Node<Key> next;

}

// Nested Iterator class

private class MyListIterator implements Iterator<Key> {

// Node to traverse the list

private Node<Key> nextNode;

// Index to keep track of iterated node

private int index = 0;

// Default constructor

public MyListIterator() {

nextNode = head;

}

@Override

public Key next() {

// If there is not a next node throw an error

if (!hasNext()) {

throw new NoSuchElementException("Illegal call to next(); "

+ "iterator is after end of list");

}

// Otherwise return the current key and advance to the next node

Key returnKey = nextNode.key;

nextNode = nextNode.next;

index++;

return returnKey;

}

@Override

public boolean hasNext() {

// If the index is less than the total nodes in

// the list we have a next node

return index < listSize;

}

@Override

public void remove() {

throw new UnsupportedOperationException("remove() is not "

+ "supported by this iterator");

}

}

}

/\*\*

\* @File MyStack.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\*/

package edu.rcc.datastructures;

import java.util.EmptyStackException;

public class MyStack<E> {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Constructors\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Default Constructor \*/

public MyStack() {

top = null;

size = 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Mutator Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Adds an entry to the top of the stack.

\*

\* @param entry

\*/

public void push(E entry) {

// Create a new node to the list

Node<E> node = new Node<E>(entry, top);

top = node;// The new node is the head now.

size++;

}

/\*\*

\* Deletes and returns an entry from the top of the stack.

\*

\* @param entry

\* @return Value Returns the value if the key was deleted, null otherwise

\*/

public E pop() {

if(top!=null){

// Set the top node to return

//System.out.println("Is data null " + (top.data==null?true:false));

E topItem = top.data;

// Delete the top node

top = top.next;

size--;

return topItem;

}else{

throw new EmptyStackException();

}

}

/\*\*

\* Peek at the top item without removing it from the stack

\*

\* @return

\*/

public E peek() {

if (isEmpty()) {

throw new EmptyStackException();

} else {

//System.out.println("Is data null: " + top.data);

return top.data;

}

}

/\*\*

\* Clears the linked list of all entries

\*/

public void clear() {

// If the list is not empty, then empty it.

if (!isEmpty()) {

top = null;

size = 0;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Accessor Functions\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* Returns whether there are any entries in the stack

\*

\* @return

\*/

public boolean isEmpty() {

return (size == 0);

}

/\*\*

\* Returns the total number of entries in the stack

\*

\* @return

\*/

public int size() {

return size;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Member Variables\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Total number key pairs in the linked list

private int size;

// Node to point to the beginning of the list

private Node<E> top;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Nested Classes\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Node nested class

private static class Node<E> {

/\* Node Constructors \*/

public Node() {

this.data = null;

this.next = null;

}

public Node(E entry, Node<E> next) {

// Set the entry and reference for the next node

this.data = entry;

this.next = next;

}

/\* Node member variables \*/

private E data;

// The head will always have the previous node member variable null

// and the tail will always have its next node member variable null

private Node<E> next;

}

}

/\*\*

\* @File MyFileReader.java

\* @author Marlo Zeroth

\* @date May 18, 2015

\* @course CSC18C DataStructures

\* Class is for helper functions and for input output of files

\*/

package edu.rcc.datastructures;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

public class MyFileReader {

/\*\*

\* Returns true if a word is an isogram. False otherwise. It looks at the

\* frequency of each letter in the word to determine if it is an isogram

\*

\* @param word

\* @return boolean

\*/

public static boolean isogram(String word) {

MyMap<Character, Integer> isogramWord = new MyMap<Character, Integer>();

// Get the frequencies of the letters in the words

for (int i = 0; i < word.length(); i++) {

if (isogramWord.contains(word.charAt(i))) {

int count = isogramWord.getValue(word.charAt(i));

isogramWord.insert(word.charAt(i), count + 1);

} else {

isogramWord.insert(word.charAt(i), 1);

}

}

// Test to see if the frequencies for all the letters is equal

String[] value = isogramWord.valuesToArray();

for (int i = 0; i < value.length - 1; i++) {

// If they differ we don't have an isogram

if (!value[i].equals(value[i + 1])) {

return false;

}

}

// If we get this far we have an isogram

return true;

}

/\*\*

\* Takes in a word returns the score based on the frequency of the letters

\* in the word. If the frequency is not the same it returns -1.

\*

\* @param word

\* @return int

\*/

public static int wordScore(String word) {

if (isogram(word)) {

MyMap<Character, Integer> isogramWord = new MyMap<Character, Integer>();

// Get the frequencies of the letters in the words

for (int i = 0; i < word.length(); i++) {

if (isogramWord.contains(word.charAt(i))) {

int count = isogramWord.getValue(word.charAt(i));

isogramWord.insert(word.charAt(i), count + 1);

} else {

isogramWord.insert(word.charAt(i), 1);

}

}

// Test to see if the frequencies for all the letters is equal

String[] value = isogramWord.valuesToArray();

return Integer.parseInt(value[0]);

}

return -1;

}

/\*\*

\* Takes a map fills it with words according the passed length of the word with

\* isograms from a text file.

\*

\* @param file

\* @param map

\* @param wordList

\* @param wordLength

\*/

public static void fillDictionary(File file, MyMap<String, Integer> map, int wordLength) {

try {

BufferedReader fromFile = new BufferedReader(new FileReader(file));

String line = null;

// Go through each line it the file and process it

do {

line = fromFile.readLine();

// In case the first line is null

if (line == null)

break;

// Fill the word dictionary to be used in the puzzles

if (line.length() == wordLength

&& line.charAt(wordLength - 2) != '\'' && isogram(line)) {

map.insert(line, wordScore(line));

}

} while (line != null);

fromFile.close();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

System.out.println("PrintWriter error opening " + file);

System.out.println(e.getMessage());

} catch (IOException e) {

e.printStackTrace();

}

}

}