CS 2210 — Data Structures and Algorithms
Assignment 2
Due Date: October 19 at 11:55 pm
Total marks: 20

1 Introduction

In this assignment we will consider a game played on a square board by two players. The board is divided into a grid of size $R \times R$ where players take turns placing tiles. The game is won by the player who is able to place k of their tiles in adjacent positions within the same row, column, or diagonal of the grid. However, there are several unavailable squares in the grid where tiles cannot be placed. If while playing the game there are no more empty squares in the grid to place tiles and no player has won the game, then the game ends in a draw.

For this assignment you will be tasked with writing several Java classes needed for a program to play the above game. The game will be played between a human player and the computer. The values of R and k are parameters of the program. You will be provided part of the code for this assignment, so you must follow the specifications below closely to ensure that your code will work correctly with the code provided.

Figure **??** shows a possible set of tiles on a board of size 4×4 where $k = 3$; in this figure the human player uses yellow tiles and the computer uses green tiles. Unavailable positions are dark blue squares and empty positions are light blue squares. In Figure **??**(a) the computer has placed three green tiles in adjacent positions of the same diagonal, so it has won the game. In Figure **??**(b) the game has ended in a draw.

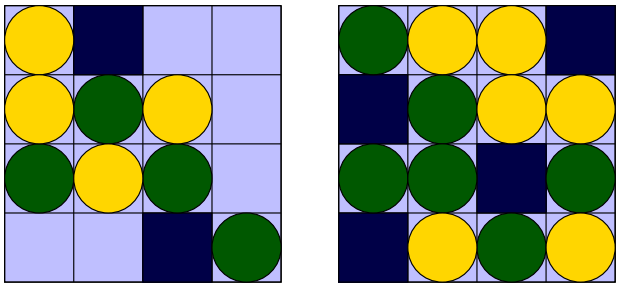


Figure 1: (a) The computer wins. (b) The game is a draw.

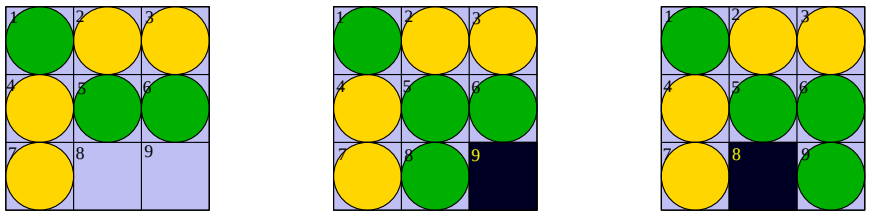
One of the Java classes provided to you will select the plays that the computer will make. If you are interested in knowing how the algorithm for playing the above game works additional information is in the file `OptionalInfo.pdf`.

2 Game States

A *game state* shows the position of every tile on the board. Your program will represent a game state as a string formed by the characters 'h', 'c', 'e', and 'u' as follows. A tile used by the computer is represented with the letter 'c'; a tile used by the human player is represented with the letter 'h'; an empty square in the board is represented with 'e', and an unavailable position in the board is represented with 'u'.

To form the string representation for a given game state we concatenate the characters corresponding to the tiles, empty, and unavailable positions on the board starting at the upper left position

and moving top to bottom and left to right. For example, for the game states in Figure 2, their string representations are "`chhhcehce`", "`chhhcchcu`", and "`chhhcuhcc`".



(a) (b) (c)

Figure 2: Game states.

Each game state is assigned a score. The higher the score is the better the current state of the game is for the computer. Conversely, the lower the score is the better the current state of the game is for the human player. The program will use the following four scores for game states:

- 0: for the game states where the human player wins
- 1: for the game states where no player has won and the game is not a draw
- 2: for the game states where the game is a draw
- 3: for the game states where the computer wins the game.

For example, if $k = 3$, the score for the board shown in Figure 2(a) is 1. The score for the board displayed in Figure 2(b) is 2 and for the board in Figure 2(c) is 3.

As mentioned above when playing the game the computer and human player will take turns placing tiles on the board. In each turn of the computer the program will consider all positions where it can place a tile and what the potential outcome is for putting it there; at the end it selects the best possible position to put its tile. Since there is a very large number of possibilities that the program needs to consider, to speed it up the program will use a hash table to store game states that it has already processed; this way the program will not have to consider the same game state multiple times.

3 Classes to Implement

You are to implement the following Java classes: `Record.java`, `Dictionary.java`, and `Evaluate.java`. You can implement more classes if you need to. You must write all the code yourself. You cannot use code from the textbook, the Internet, or any other sources. You **cannot** use Java's `Hashtable` class or `hashCode()` method.

3.1 Class `Record`

This class represents the records that will be stored in the hash table, implemented in the below class `Dictionary.java`. An object of this class stores a string and two integers; therefore there will be three instance variables in this class. The string stored in an object of this class will be used as its key attribute.

For this class, you must implement all and only the following `public` methods:

- `public Record(String key, int score, int level)`: The constructor for the class.
- `public String getKey()`: Returns the string stored in this `Record` object.

- `public int getScore()`: Returns the first integer stored in this `Record` object.
- `public int getLevel()`: Returns the second integer stored in this `Record` object.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods.

3.2 Class `Dictionary`

This class implements a dictionary using a hash table in which collisions are resolved using separate chaining. The hash table will store objects of the class `Record`.

You will decide on the size of the table, keeping in mind that the size of the table must be a prime number.

You must design your hash function so that it produces few collisions. A bad hash function that induces many collisions will result in a lower mark. You are **required to use a polynomial hash function**. As mentioned above, **you cannot use** Java's `hashCode()` method in your hash function.

For this class, you must implement all the `public` methods in the following interface:

```
public interface DictionaryADT {
    public int put(Record rec) throws DuplicatedKeyException;
    public void remove(String key) throws InexistentKeyException;
    public Record get(String key);
    public int numRecords();
}
```

The descriptions of these methods follows:

- `public int put(Record rec) throws DuplicatedKeyException`: Inserts the given `Record` object referenced by `rec` in the dictionary. This method must throw a `DuplicatedKeyException` (see below) if the string stored in the object referenced by `rec` is already in the dictionary.

  You are required to implement the dictionary using a hash table with separate chaining. To determine how good your design is, we will count the number of collisions produced by your hash function. Method `put` **must** return the value 1 if the insertion of the object referenced by `rec` into the hash table produces a collision, and it must return the value 0 otherwise. In other words, if for example your hash function is h(key) and the name of your table is T, this method must return the value 1 if the list in entry $T[h(\texttt{rec.getKey()})]$ of the table already stores at least one element; it must return 0 if that list was empty before adding the new record.

- `public void remove(String key) throws InexistentKeyException`: Removes the `Record` object containing string `key` from the dictionary. Must throw a `InexistentKeyException` if the hash table does not store any `Record` object with the given `key` value.
- `public Record get(String key)`: A method which returns the `Record` object stored in the hash table containing the given `key` value, or `null` if no `Record` object stored in the hash table contains the given `key` value.
- `public int numRecords()`: Returns the number of `Record` objects stored in the hash table.

Since your `Dictionary` class must implement all the methods of the `DictionaryADT` interface, the declaration of your method should be as follows:

```
public class Dictionary implements DictionaryADT
```

You can download the file `DictionaryADT.java` from OWL. The only other public method that you can implement in the `Dictionary` class is the constructor method, which must be declared as follows

```
public Dictionary(int size)
```

this initializes a dictionary with an empty hash table of the specified size.

You can implement any other methods that you want to in this class, but they must be declared as `private` methods (i.e. not accessible to other classes).

Hint. You might want to implement a class `Node` storing an object of the class `Record` to construct the linked lists associated to the entries of the hash table. You do not need to follow this suggestion. You can implement the lists associated with the entries of the table in any way you want.

3.3 Class `Evaluate`

This class implements all the auxiliary methods needed by the algorithm that plays the game. For details on the algorithm that plays the game, please read document `OptionalInfo.pdf` posted in OWL as an optional reading for this assignment.

The constructor for this class must be as follows

```
public Evaluate (int size, int tilesToWin, int maxLevels)
```

The first parameter specifies the size of the board, the second parameter is the number of adjacent tiles needed to win the game, and the last parameter specifies the playing quality of the program (the higher this value is the better the program will play, but the slower it will be; when you test your program use values between 3 and 5 for this parameter so the program plays OK but it is not too slow).

This class must have an instance variable called `gameBoard` of type `char[][]` to store the board. This variable is initialized inside the constructor so that every entry of `gameBoard` stores the character 'e' indicating that every position of the board is empty. As the game is played, every entry of `gameBoard` will store one of the characters 'c', 'h', 'e', or 'u'. This class must also implement the following public methods.

- `public Dictionary createDictionary()`: returns an empty `Dictionary` of the size that you have selected. Remember that the size of the dictionary must be a prime number.
- `public Record repeatedState(Dictionary dict)`: This method first represents the content of the two dimensional array `gameBoard` as a string as described in Section 2; then it checks whether there is a record in `dict` with this string as key attribute: If there is, this method returns the `Record` object that contains it; otherwise the method returns the value null.
- `public void insertState(Dictionary dict, int score, int level)`: This method first represents the content of `gameBoard` as a string as described in Section 2, then it creates an object of the class `Record` storing this string, `score`, and `level`; finally, this `Record` object is stored in `dict`. Remember that the hash table cannot store two records with the same key attribute.
- `public void storePlay(int row, int col, char symbol)`: This method stores `symbol` in `gameBoard[row][col]`.
- `public boolean squareIsEmpty (int row, int col)`: This method returns true if `gameBoard[row][col]` is 'e'; otherwise it returns false.
- `public boolean tileOfComputer (int row, int col)`: This method returns true if `gameBoard[row][col]` is 'c'; otherwise it returns false.

- `public boolean tileOfHuman (int row, int col)`: Returns true if `gameBoard[row][col]` is 'h'; otherwise it returns false.
- `public boolean wins (char symbol)`: Returns true if there are the required number of adjacent tiles of type `symbol` in the same row, column, or diagonal of `gameBoard`; otherwise it returns false.
- `public boolean isDraw()`: Returns true if there are no empty positions left in `gameBoard`; otherwise it returns false.
- `public int evalBoard()`: Returns one of the following values:
  - ▷ 3, if the computer has won, i.e. there are the required number of adjacent 'c's in the same row, column, or diagonal of `gameBoard`.
  - ▷ 0, if the human player has won, i.e. there are the required number of adjacent 'h's in the same row, column, or diagonal of `gameBoard`.
  - ▷ 2, if the game is a draw.
  - ▷ 1, if the game is still undecided, i.e. no player has won and the game is not a draw.

You can implement more methods in this class, if you want, but they must be declared as `private`.

4 Classes Provided

You can download classes `DictionaryADT.java`, `PosPlay.java`, `Play.java`, `InexistentKeyException` and `DuplicatedKeyException` from OWL. Class `PosPlay` is an auxiliary class used by `PlayGame` to represent plays. Class `Play` includes the `main` method for the program. The program will be executed by typing

```
java Play inputFile
```

where `inputFile` is the name of a text file containing the description of the game board. The format of this file is as follows:

- The first line is the size of the gameboard
- The second line is the number of tiles of the same player that need to appear in adjacent positions of the same row, column, or diagonal for that player to win the game.
- The third line is a parameter specifying the playing quality of the program. The larger this value is the better the program will play, but the slower it will be.
- The remaining lines of the file contain the characters 'e' and 'u'. Character 'e' indicates an empty position of the board, and 'u' indicates an unavailable position of the board.

The following sample input file specifies a board of size 4×4 in which 3 symbols in adjacent positions are needed to win, and the quality parameter is equal to 3. In the game board all positions except two positions in the center are empty.

```
4
3
3
eeee
eeue
euee
```

```
eeee
```

Class `Play` also contains methods for displaying the game board on the screen and for allowing the human player to place their tiles.

5 Testing your Program

We will perform two kinds of tests on your program: (1) tests for your implementation of the dictionary, and (2) tests for your implementation of class `Evaluate`. For testing the dictionary we will run a test program called `TestDict` which performs a few simple tests to check whether your dictionary works as specified. We supply you with a copy of `TestDict` so you can use it to test your implementation.

6 Coding Style

Your mark will be based partly on your coding style.

- Variable and method names should be meaningful and they must reflect their purpose in the program.
- Comments, indenting, and white spaces should be used to improve readability.
- You must only use instance variables for data which needs to be maintained throughout the life span of an object. In other words, variables which are needed only inside methods should be declared as local variables inside those methods.
- All instance variables should be declared `private` to maximize information hiding. Any outside access to these variables should be done with accessor methods (like `getScore()` for class `Record`).

7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `Dictionary` tests pass: 4 marks.
- `Evaluate` tests pass: 4 marks.
- Coding style: 2 marks.
- Hash table implementation: 4 marks.
- `Evaluate` program implementation: 4 marks.

8 Notes About Eclipse

If you are using Eclipse, please read the tutorials in the FAQ page of OWL to learn how to pass the name of the input file as parameter to your program. Depending on how you configured Eclipse, in most cases for Eclipse to find the input file you must place it in the project's root directory; this is the folder that contains the `src` and `bin` folders. The provided image files must also be placed in the project's root directory.

If Eclipse cannot find any of the above files, move them around within your project root's directory until Eclipse finds them. Alternatively you can place all java files, input files, and image files in the

same folder and compile your program by typing `javac Play.java` and then run the program by typing `java Play inputFile`, where `inputFile` is the name of the input file.

9 Handing In Your Program

You must submit an electronic copy of your program. To submit your program, login to OWL and submit **your java** files from there. **DO NOT** put your code in sub-directories. **DO NOT** compress your files or submit a .zip, .rar, .gzip, or any other compressed file. Only the .java files that you have written must be submitted. If you do not follow this guidelines you might lose marks.

Remember that the TA's will test your program on the computers of the Department.

When you submit your program, we will receive a copy of it with a datestamp and timestamp. If you submit your program more than once we will take the latest program submitted as the final version, and will deduct marks accordingly if it is late.