CS 2210 — Data Structures and Algorithms
Assignment 2: Achi Program with Hash Tables

Due Date: October 19 at 11:59 pm

Total marks: 20

1 Overview

For this assignment you are required to write a Java program that plays achi. In achi two players take turns putting tiles on a square game board of size $n \times n$. The goal is for a player to place n of their tiles in the same row, column, or diagonal of the board, similarly as in the game of tic tac toe. In achi, however, at most $n^2 - 1$ tiles can be placed on the board and if by then no player has won then the players take turns sliding onto the empty position of the board one of their tiles adjacent to that position. If on their turn a player does not have any tiles adjacent to the empty position of the board the game ends in a draw.

Figure 1 (a) shows a possible set of tiles on a board of size 3×3. If the next player to move is the one with the 'X' tiles then they can move any of the tiles in positions 2, 3, or 5 into position 6. If, for example, the tile in position 3 is moved to position 6 the board will look as shown in Figure 1 (b). Since now it is the turn of the player with the 'O' tiles, the game ends in a draw as there is no 'O' tile adjacent to the empty position.

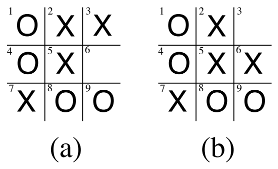

Figure 1: Game of achi.

2 The Algorithm for Playing Achi

You will be given code for displaying the game board on the screen. Your program will play against a human opponent. The human player uses 'X' tiles and always starts the game. The computer uses 'O' tiles. You can ignore the rest of Section 2, which explains how the algorithm to play achi works. You will be given java code with this algorithm, but there are several java classes that you need to implement as described in Section 3.

In each turn the computer examines all possible moves and selects the best one; to do this, each possible move is assigned a score. Your program will use the following 4 scores for moves:

- 0: if a move ensures that the human player wins
- 1: if after a move is selected it is not clear who will win
- 2: if a move leads to a draw
- 3: if a move guarantees that the computer wins the game.

For example, suppose that the game board looks like the one in Figure 2(a). If the computer plays in position 8, the game will end in a draw (see Figure 1(b)), so the score for the computer playing in position 8 is 2. We say that the score for the board configuration in Figure 1(b) is 2, where a *configuration* is simply the positioning of the tiles on the game board. Similarly, the score for the configuration in Figure 1(c) is 3, since in this case the computer wins the game.

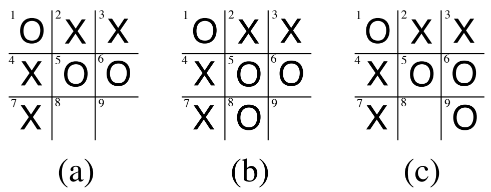

Figure 2: Board configurations.

To compute scores, your program will use a recursive algorithm that will repeatedly simulate a move from the computer followed by a move from the human, until an outcome for the game has been decided. This recursive algorithm will implicitly create a tree formed by all the moves that the players can make starting at the current board configuration. This tree is called a *game tree.* An example of a game tree is shown in Figure 3.

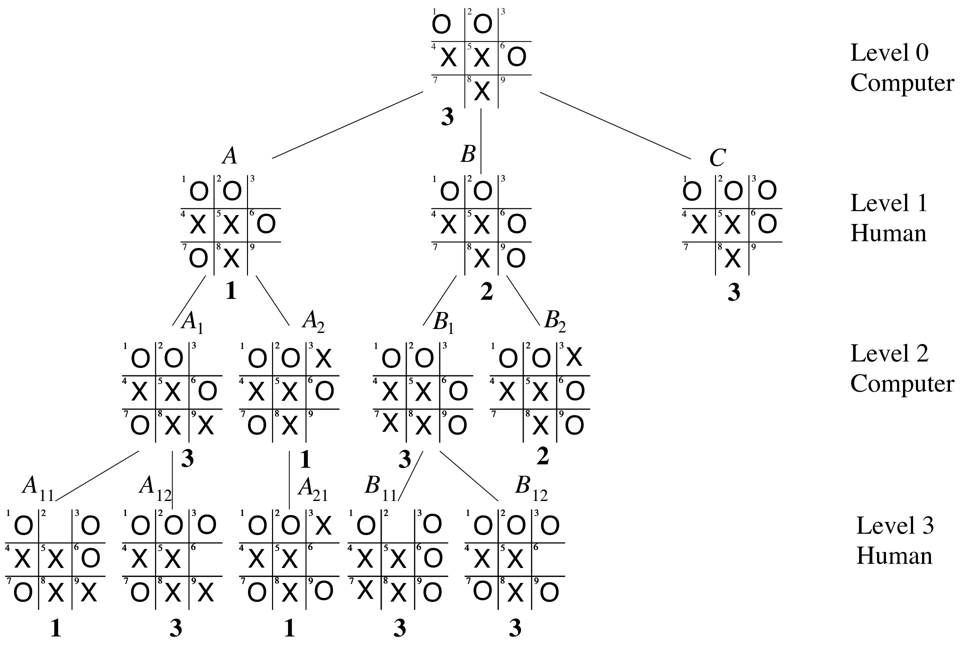

Figure 3: A game tree.

Assume that after 6 moves the game board is as the one shown at the top of Figure 3 and suppose that it is the computer's turn to move. The algorithm for computing scores will first try all possible moves that the computer can make: A, B, and C. For each one of them, the algorithm will then consider all possible moves by the human player: A_1, A_2, B_1, and B_2. Then, all possible responses by the computer are attempted, and so on. Note that a game tree might have many levels as players might need many moves to decide the outcome of the game. In order for your program to be responsive, we will limit the maximum number of levels of the game tree that the program will consider (see Section 4). In Figure 3 we limited the number of levels to 3.

In Figure 3 each level of the tree is labelled by the player whose turn is next. So levels 0 and 2 are labelled "computer" and the other 2 levels are labelled "human". After reaching configurations A_{11}, A_{12}, B_{11}, and B_{12} in level 3, the algorithm computes a score for each one of them depending on whether the computer wins, the human wins, the game is a draw, or no decision has been reached. Scores are the numbers in bold below the configurations. The scores are propagated upwards as follows:

- For a configuration b on a level labelled "computer", the highest score of the configurations in the next level is selected as the score for b. This is because we have chosen the highest score (`COMPUTER_WINS` = 3) for a play that makes the computer win.
- For a configuration b on a level labelled "human", the score of b is equal to the minimum score in the next levels, because the lowest score (`HUMAN_WINS`=0) has been given to any play that guarantees a win by the human player.

Since for the board at the top of Figure 3, putting an 'O' in position 3 yields the configuration with the highest score, then the computer will choose to play in position 3. We give below the algorithm for computing scores and for selecting the best available move. The algorithm is given in Java, but we have omitted variable declarations and some initialization steps. A full version of the algorithm can be found inside class `PlayAchi.java`, which can be downloaded from the course's website.

```
private PosPlay computerPlay(char symbol, int highestScore, int lowestScore, int level) {
   if (level == 0) configurations = t.createDictionary();
   if (symbol == 'X') opponent = 'O'; else opponent = 'X';

   for(int row = 0; row < board_size; row++)
     for(int column = 0; column < board_size; column++)
       if(t.tileIsEmpty(row,column)) { // Empty position found
         t.storePlay(row,column,symbol);
         if (t.wins(symbol)||t.isDraw(symbol)||(level == max_level))
           reply = new PosPlay(t.evalBoard(symbol),row,column);
         else {
         lookupVal = t.repeatedConfig(configurations);
(*)         if (lookupVal != -1)
             reply = new PosPlay(lookupVal,row,column);
           }
           else {
             reply = computerPlay(opponent, highestScore, lowestScore, level + 1);
             t.insertConfig(configurations,reply.getScore());
           }
         }

         t.storePlay(row,column,' ');
         if((symbol == COMPUTER && reply.getVal() > value) || // A better play was found
           (symbol == HUMAN && reply.getVal() < value)) {
           bestRow = row; bestColumn = column;
           value = reply.getVal();
           if (symbol == COMPUTER && value > highestScore) highestScore = value;
           else if (symbol == HUMAN && value < lowestScore) lowestScore = value;
           if (highestScore >= lowestScore) /* alpha/beta cut */
             return new PosPlay(value, bestRow, bestColumn);
         }
       }
   return new PosPlay(value, bestRow, bestColumn);
}
```

The fist parameter of the algorithm is the symbol (either 'X' or 'O') of the player whose turn is next. The second and third parameters are the highest and lowest scores for the board positions that have been examined so far. The last parameter is used to bound the maximum number of levels of the game tree that the algorithm will consider.

2.1 Speeding-up the Algorithm with a Dictionary

The above algorithm includes several tests that allow it to reduce the number of configurations that need to be examined in the game tree. For this assignment, the most important test used to speed-up the program is the one marked (*). Every time that the score of a board configuration is computed, the configuration and its score are stored in a dictionary, that you will implement using a hash table. Then, when algorithm `computerPlay` is exploring the game tree trying to determine the computer's best move, before it expands a configuration b it will look it up in the dictionary. If b is in the dictionary then its score is simply extracted from the dictionary instead of exploring the part of the game tree below b.

For example, consider the game tree in Figure 3. The algorithm examines first the left branch of the game tree, including configuration D and all the configurations that appear below it. After exploring the configurations below D, the algorithm computes the score for D and then in stores D and its score in the dictionary. When later the algorithm explores the right branch of the game tree, configuration D will be found again, but this time its score is simply obtained from the dictionary instead of exploring all configurations below D, thus reducing the running time of the algorithm.

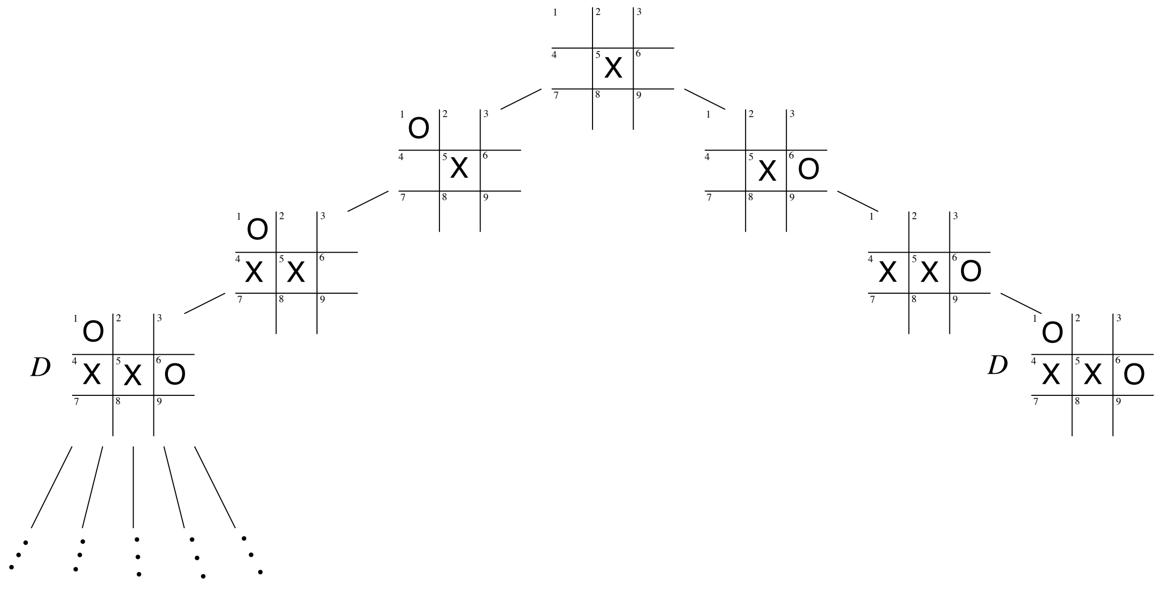

Figure 4: Detecting repeated configurations.

3 Classes to Implement

You are to implement at least 4 Java classes: `ConfigData.java`, `Dictionary.java`, `DictionaryException.java`, and `Achi.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 the standard Java `Hashtable` class.

3.1 ConfigData

This class represents the data stored in one entry of the dictionary. An object of this class associates a configuration with its integer score. Each board configuration will be represented as a string as follows: Concatenate all characters ('X', 'O', or ' ') placed on the board starting at the upper left position and moving left to right and top to bottom. For example, for the configurations in Figure 2, their string representations are "`OXXXOOX␣␣`" (note that there are two spaces at the end of the string representing the two empty places in the board of Figure 2(a)), "`OXXXOOXO␣`", and "`OXXXOOX␣O`".

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

- `public ConfigData(String config, int score)`: A constructor which returns a new `ConfigData` with the specified configuration and score. The string `config` will be used as the key attribute for every `ConfigData` object.
- `public String getConfig()`: Returns the configuration stored in a `ConfigData` object.
- `public int getScore()`: Returns the score in a `ConfigData` object.

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).

3.2 Dictionary

This class implements a dictionary using a hash table with separate chaining. You will decide on the size of the table, keeping in mind that the size of the table must be a prime number. A table of size between 5000-10000, should work well.

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 cannot use** the built-in java `hashCode()` method as your hash function.

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

```
public interface DictionaryADT {
    public int insert (ConfigData pair) throws DictionaryException;
    public void remove (String config) throws DictionaryException;
    public int find (String config);
}
```

The description of these methods follows:

- `public int insert(ConfigData pair) throws DictionaryException`: Inserts the given `ConfigData` object referenced by `pair` in the dictionary. This method must throw a `DictionaryException` (see below) if the configuration in `pair`, `pair.getConfig()`, 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 `insert` will return the value 1 if the insertion of the object referenced by `pair` into the hash table produces a collision, and it will return the value 0 otherwise. In other words, if for example your hash function is $h(\text{key})$ and the name of your table is T, this method will return the value 1 if the list in $T[h(\texttt{pair.getConfig()})]$ already stores at least one element; it will return 0 if the list in $T[h(\texttt{pair.getConfig()})]$ was empty before the insertion.
- `public void remove(String config) throws DictionaryException`: Removes the entry with configuration `config` from the dictionary. Must throw a `DictionaryException` (see below) if the configuration is not in the dictionary.

- `public int find(String config)`: A method which returns the score stored in the dictionary for the given configuration, or -1 if the configuration is not in the dictionary.

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 the course's website. 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 returns an empty dictionary 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 type `ConfigData` and a link object of type `Node` to construct the linked list associated to an entry 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 DictionaryException

This is just the class implementing the class of exceptions thrown by the `insert` and `remove` methods of `Dictionary`. See the class notes on exceptions.

3.4 Achi

This class implements all the support methods needed by algorithm `computerPlay`, described in Section 2. The constructor for this class must be as follows

```
public Achi (int board_size, int max_levels)
```

The first parameter specifies the size of the game board, and the second one is the maximum level of the game tree that will be explored by the program.

This class must have an instance variable called `gameBoard` of type `char[][]` to store the game board. This variable is initialized inside the above constructor method so that every entry of `gameBoard` stores a space ' '. As the game is played, every entry of `gameBoard` will store one of the characters 'X', 'O', or ' '. This class must also implement the following public methods.

- `public Dictionary createDictionary()`: returns an empty `Dictionary` of the size that you have selected.

- `public int repeatedConfig(Dictionary configurations)`: This method first represents the content of `gameBoard` as a string as described above; then it checks whether the string representing the `gameBoard` is in the `configurations` dictionary: If it is in the dictionary this method returns its associated score, otherwise it returns the value -1.

- `public void insertConfig(Dictionary configurations, int score)`: This method first represents the content of `gameBoard` as a string as described above; then it inserts this string and `score` in the `configurations` dictionary.

- `public void storePlay(int row, int col, char symbol)`: Stores `symbol` in `gameBoard[row][col]`.

- `public boolean tileIsEmpty (int row, int col)`: Returns true if `gameBoard[row][col]` is ' '; otherwise it returns false.
- `public boolean tileIsComputer (int row, int col)`: Returns true if `gameBoard[row][col]` is 'O'; otherwise it returns false.
- `public boolean tileIsHuman (int row, int col)`: Returns true if `gameBoard[row][col]` is 'X'; otherwise it returns false.
- `public boolean wins (char symbol)`: Returns true if there are n adjacent tiles of type `symbol` in the same row, column, or diagonal of `gameBoard`, where n is the size of the game board.
- `public boolean isDraw(char symbol)`: Returns true if the game configuration corresponding to `gameBoard` is a draw assuming that the player that will perform the next move uses symbols of the type specified by `symbol`.
- `public int evalBoard(char symbol)`: Returns one of the following values:
  - ▷ 3, if the computer has won, i.e. there are n adjacent 'O's in the same row, column, or diagonal of `gameBoard`.
  - ▷ 0, if the human player has won.
  - ▷ 2, if the game is a draw when the player that needs to make the next move uses tiles of the type specified by `symbol`.
  - ▷ 1, if the game is still undecided, i.e. no player has won and the next player can move a tile.

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` and `PlayAchi.java` from the course's website. Class `PosPlay` is an auxiliary class used by `PlayAchi` to represent plays. Class `PlayAchi` includes the main method for your program, so the program will be executed by typing

```
java PlayAchi size max_levels
```

where `size` is the size of the game board and `max_levels` is the maximum number of levels of the game tree that the program will explore. Remember that the larger the value of `max_levels` is the better the program will play, but the slower it will be.

Class `PlayAchi` also contains methods for displaying the game board on the screen and for reading the moves of the human player.

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 the program to play achi. 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 will 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.
- No variable declarations should appear outside methods ("instance variables") unless they contain data which needs to be maintained in an object from call to call. In other words, variables which are needed only inside methods should be declared inside those methods.
- All variables declared outside methods ("instance variables") should be declared `private` (not `protected`), to maximize information hiding. Any outside access to these variables should be done with accessor methods (like `getScore()` for `ConfigData`).

7 Marking

Your mark will be computed as follows.

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

8 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. Please **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 your .java files should be submitted.

Read the tutorials posted in the course's website to learn how to copy your program onto your GAUL account and how to test it there. Remember that the TA's will test your program on Gaul. Please also read the tutorials about how to configure Eclipse to read command line arguments.

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 please send me an email message to let me know. We will take the latest program submitted as the final version, and will deduct marks accordingly if it is late.