CS 2210a — Data Structures and Algorithms
Assignment 2: The X+Game

Due Date: October 18 at 11:55 pm

Total marks: 20

1 The X+Game

In this assignment you will write part of a Java program to play the X+Game. In this game two players (a human and the computer) take turns putting tiles on a board having n rows and n columns, and to win the game a player needs to put at least k tiles on board positions forming an X-shape or a +shape. When running the program to play the game we will specify the value of n and the value of k.

An X-shape is formed as follows:

- There is a center tile and four tiles in adjacent board positions along the four corners of the center tile (see Figure 1(a)).
- There might be additional tiles in adjacent positions to the four corners of the center tile along its diagonals (see Figure 1(b)).

A +shape is formed as follows:

- There is a center tile and four tiles in adjacent positions above, below, to the right, and to the left of the center tile (see Figure 1(b)).
- There might be additional tiles in adjacent positions in the same row or in the same column of the center tile (see Figure 1(d)).

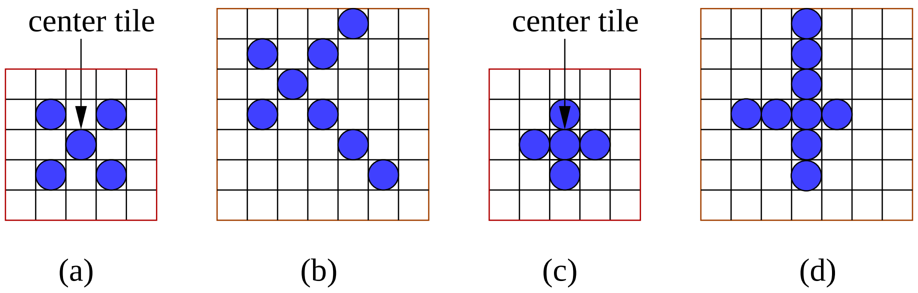

Figure 1: (a) X-shape of length 5. (b) X-shape of length 8, (c) +shape of length 5. (d) +shape of length 9.

The *length* of an X-shape or a +shape is the number of tiles in it. Figure 1 shows several X-shapes and +shapes, along with their lengths. As mentioned above, to win the game a player needs to form with their tiles an X-shape or a +shape (either one) of length at least k.

You will be given code for displaying the board on the screen and some of the code for playing the game, as explained below.

2 The Algorithm for Playing the X+Game

The human player always starts the game and uses blue tiles; the computer uses red tiles. In each turn the computer examines all free board positions and selects the best one to place a tile there;

to do this, each possible play is assigned a score. In this assignment we will use the following four scores for plays:

- `HUMAN_WINS` = 0: this score is given to a play that makes the human player win
- `UNDECIDED` = 1: this score is given to a play for which it is not clear which player will win
- `DRAW` = 2: this score is given to a play that will lead to a draw (i.e. no player wins the game)
- `COMPUTER_WINS` = 3: this score is given to a play that will ensure that the computer wins.

For example, suppose that the board looks like the one in Figure 2(a) and the length of the X-shape of +shape needed to win is 5. If the computer plays in position 8, the game will end in a draw (see Figure 2(b)), so the score for the computer playing in position 8 is 2 (`DRAW`). Similarly, the score for the computer playing in position 9 is 0 (`HUMAN_WINS`) as then the human player will win.

We define a *configuration* as the way in which the tiles are positioned on the board. For example, the configuration shown in Figure 2(b) corresponds to a game that ends up in a draw. The configuration in Figure 2(c) corresponds to a game won by the human player, as there are 5 blue tiles forking a +shape of length 5. Each configuration is also assigned one of the 4 above scores. So, the configuration in Figure 2(b) gets a score of 2 and the configuration in Figure 2(c) gets a score of 0.

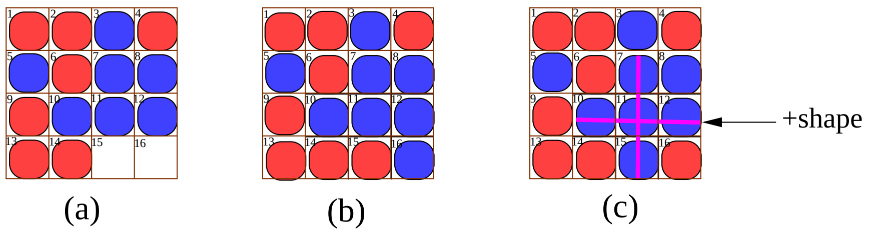

Figure 2: Board configurations.

Note. You can skip the rest of this section and Section 2.1, as you do not need them to complete the assignment. However, if you are interested in knowing how a computer program that plays a 2-player game works, then keep on reading.

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

Assume that after several moves the board is as the one shown at the top of Figure 3 and suppose that it is the computer's turn to play. The algorithm for computing scores will first try all possible plays that the computer can make: play A: play at position 2, play B: play at position 10, and play C: play at position 16. For each one of these plays, the algorithm will then consider all possible plays by the human player: A_1 and A_2 (if the computer plays as in configuration A), B_1 and B_2 (if the computer plays as in configuration B), and C_1 and C_2 (if the computer plays as in configuration C). Then, all possible responses by the computer are attempted, and so on until the outcome of each possible sequence of plays is determined.

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 final configurations A_{11}, A_{22}, B_{11}, B_{22}, C_{11}, and C_{22}, the algorithm computes a score for each one of them depending on whether the computer wins, the human wins, or the game is a draw. These scores are propagated upwards as follows:

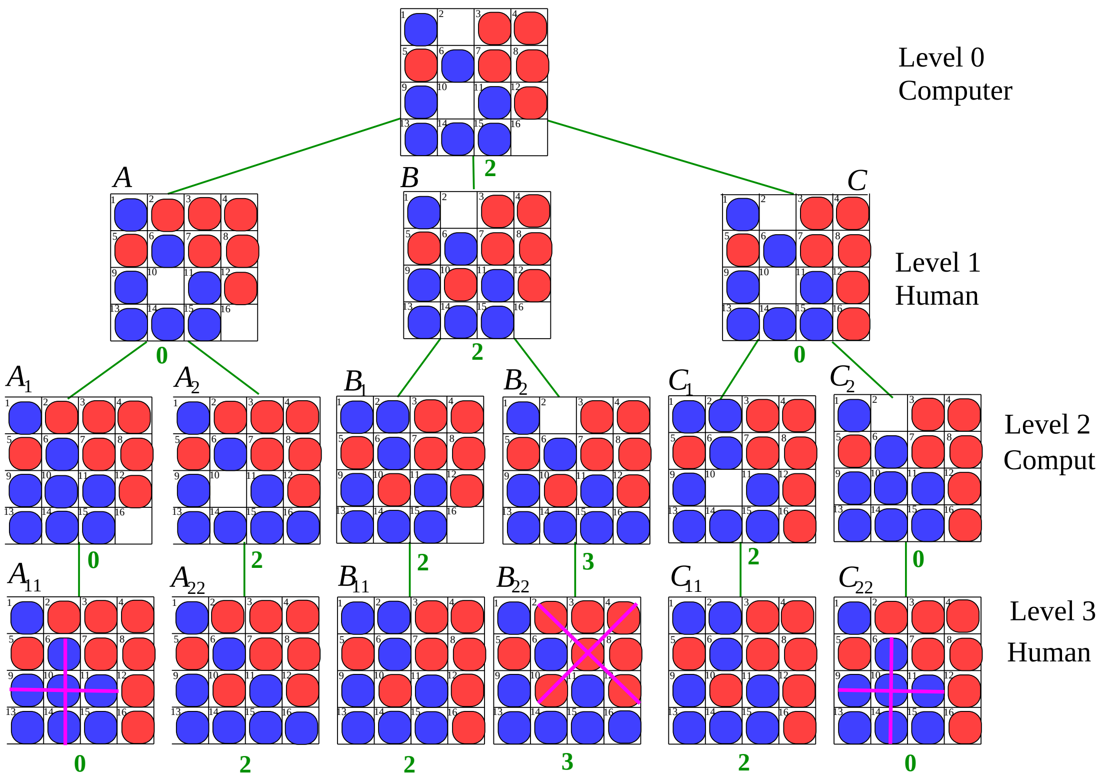

Figure 3: A game tree.

- For a configuration c on a level labelled "Computer", the highest score of the adjacent configurations in the next level is selected as the score for c. This is because the higher the score is, the better the outcome is for the computer.
- For a configuration c on a level labelled "Human", the score of c is equal to the minimum score of the adjacent configurations in the next level, because the lower the score is, the better the outcome is for the human player.

The scores for the configurations in Figure 3 are the numbers in green. For example, for the configuration at the top of Figure 3, putting a red tile in position 10 yields the configuration with the highest score (2), hence the computer will choose to play in position 10. Similarly, for configuration A in Figure 3, placing a blue tile in position 10 yields the configuration with the smallest score (0), so the human would choose to play in position 10.

We give below the algorithm for computing scores and for selecting the best available play. 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 `Play.java`, which can be downloaded from OWL.

```
private PosPlay computerPlay(char symbol, int highestScore, int lowestScore, int level) {
   if (level == 0) configurations = t.createDictionary();
   if (symbol == HUMAN) opponent = COMPUTER; else opponent = HUMAN;

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

         t.savePlay(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 first parameter of the algorithm is the symbol (either HUMAN or COMPUTER) representing the tiles 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. Since the number of configurations in the game tree could be very large, to speed up the algorithm the value of the last parameter specifies the highest level of the game tree that will be explored. Note that the smaller the value of this parameter is, the faster the algorithm will be, but the worse it will play.

Also note that if we bound the number of levels of the game tree, it might not be possible to determine the outcome of the game for some of the configurations in the lowest level of the tree. For example, if in the game tree of Figure 3 we set the maximum level to 2, then the algorithm will explore only levels 0, 1, and 2. At the bottom of the tree will appear configurations A_1, A_2, B_1, B_2, C_1, and C_2. Among these configurations, the scores for A_1 and C_2 are 0, as the human player wins in those cases; however, the scores for the remaining configurations are not known as in none of these configurations any player has won, and the configurations still include empty positions, so they do not denote game draws. In this case, configurations A_2, B_1, B_2, and C_1 will receive a score of `UNDECIDED` = 1.

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 c it will look it up in the dictionary. If c is in the dictionary, then its score is simply extracted from the dictionary, instead of exploring the part of the game tree below c.

For example, consider the game tree in Figure 4. 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 it 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.

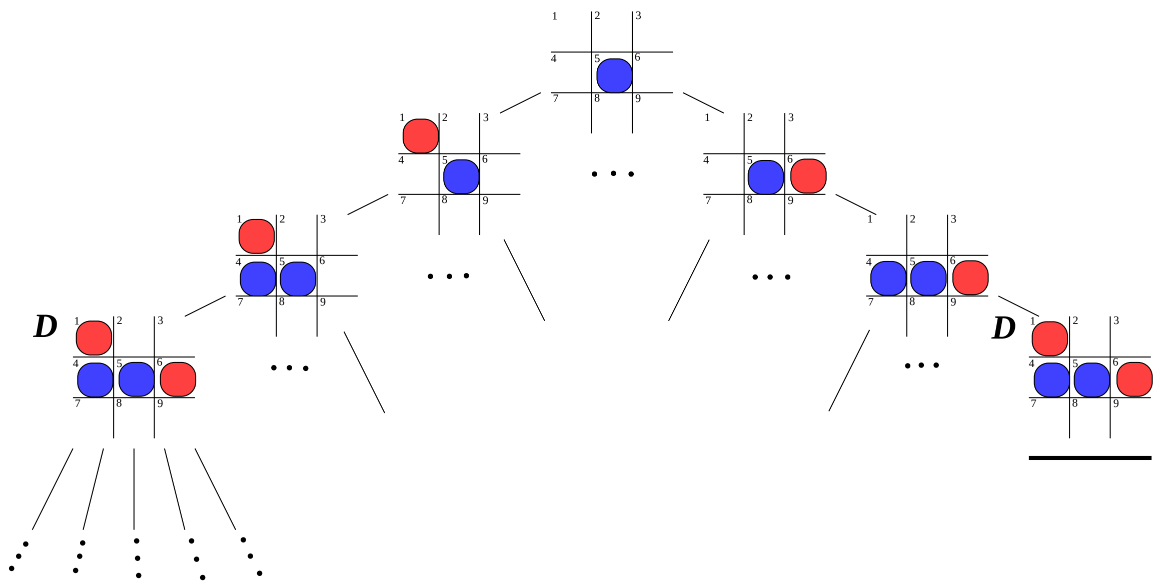

Figure 4: Detecting repeated configurations.

3 Classes that You Need to Implement

You are to implement 3 Java classes: `HashDictionary.java`, `Configurations.java`, and `Data.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 classes `Hashtable`, `HashMap`, `HashSet` or any other Java provided class that implements a hash table. You **cannot** use the `hashCode()` method. However, you can use the LinkedList Java class.

3.1 Class Data

This class represents the records that will be stored in the HashDictionary. Each record stored in the dictionary will consists of two parts: a configuration and an integer score.

Each board configuration will be represented as a string as follows. A blue tile will be represented with the character `'X'` and a red tile will be represented with the character `'O'`. We will concatenate all the characters corresponding to the tiles placed on the board starting at the top left position and moving from left to right and from top to bottom. For example, for the configurations in Figure 2, their string representations are "`OOXOXOXXOXXXOO␣␣`"[1], "`OOXOXOXXOXXXOOOX`", and "`OOXOXOXXOXXXOOXO`".

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

- `public Data(String config, int score)`: A constructor which initializes a new `Data` object with the specified configuration and score. The string `config` will be used as the key attribute for every `Data` object.

- `public String getConfiguration()`: Returns the configuration stored in `this Data` object.

- `public int getScore()`: Returns the score in `this Data`.

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

3.2 Class HashDictionary

This class implements the Dictionary ADT using a hash table. You must implement the dictionary using a hash table with separate chaining. 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 encouraged to use the polynomial hash function discussed in class. As mentioned above, collisions must be resolved using separate chaining.

Note. Your hash function will receive as input a String and the size of the table and it will map the String into a position of the table. The String received as the first parameter can be of any length and can contain any characters.

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

```
public interface DictionaryADT {
    public int put (Data record) throws DictionaryException;
    public void remove (String config) throws DictionaryException;
    public int get (String config);
    public int numRecords();
}
```

The description of these methods follows:

- `public int put(Data record) throws DictionaryException`: Adds `record` to the dictionary. This method must throw a `DictionaryException` if `record.getConfiguration()` is already in the dictionary.

  To determine how good your design is, we will count the number of collisions produced by your hash function. To be able to do this, method `put` must return the value 1 if adding `record` to the table produces a collision, and it will 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 will return the value 1 if $T[h($`record.getConfiguration()`$)]$ already stores at least one element; it will return 0 if $T[h($`record.getConfiguration()`$)]$ was empty.

[1]Note that there are two blank spaces at the end of the string representing the two empty places in the board of Figure 1(a)

- `public void remove(String config) throws DictionaryException`: Removes the record with the given `config` from the dictionary. Must throw a `DictionaryException` if no record in the hash table stores `config`.

- `public int get(String config)`: Returns the score stored in the record of the dictionary with key `config`, or -1 if `config` is not in the dictionary.

- `public int numRecords()`: Returns the number of `Data` objects stored in the dictionary.

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

```
public class HashDictionary implements DictionaryADT
```

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

```
public HashDictionary(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.

3.3 Class Configurations

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

```
public Configurations (int board_size, int lengthToWin, int max_levels)
```

The first parameter specifies the size of the board, the second is the length of the X-shape or +shape needed to win the game, and the third is the maximum level of the game tree that will be explored by the program.

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

- `public HashDictionary createDictionary()`: returns an empty `HashDictionary`. You will select the size of the hash table, keeping in mind that it must be a prime number. A table of size between 6 000-10 000, should work well for this assignment.

- `public int repeatedConfiguration(HashDictionary hashTable)`: This method first stores the characters of `board` in a String as described above; then it checks whether the String representing the `board` is in `hashTable`: If the String is in `hashTable` this method returns its associated score, otherwise it returns the value -1.

- `public void addConfiguration(HashDictionary hashDictionary, int score)`: This method first represents the content of `board` as a String as described above; then it inserts this String and `score` in `hashDictionary`.

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

- `public boolean squareIsEmpty (int row, int col)`: Returns true if `board[row][col]` is ' '; otherwise it returns false.

- `public boolean wins (char symbol)`: Returns true if there is an X-shape or a +shape of length at least k formed by tiles of the kind `symbol` on the `board`, where k is the length of the X-shape or +shape needed to win the game.
- `public boolean isDraw()`: Returns true if `board` has no empty positions left and no player has won the game.
- `public int evalBoard()`: Returns one of the following values:
  - 3, if the computer has won, i.e. there is an X-shape or a +shape formed by tiles of type 'O' on the board
  - 0, if the human player has won
  - 2, if the game is a draw, i.e. there are no empty positions in `board` and no player has won
  - 1, if the game is still undecided, i.e. there are still empty positions in `board` and no player has won yet.

4 Classes Provided

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

```
java Play size length max_levels
```

where `size` is the size of the board, `length` is the length of the X-shape or +shape needed to win the game, and `max_levels` is the maximum number of levels of the game tree that the program will explore. If you use Eclipse you need to configure it so it passes these parameters to the `main` method. Look at the FAQ tab in OWL if you need help to configure Eclipse.

Class `Play` also contains methods for displaying the board on the screen and for capturing the plays 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 using a hash table, and (2) tests for your implementation of the program to play the X+Game. For testing the dictionary we will run a test program called `TestDict`. We will supply you with a copy of `TestDict` so you can use it to test your implementation. We will perform additional tests on your code that we will not provide to you to encourage you to test your program thoroughly.

6 Coding Style

Your mark will be based partly on your coding style.

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

- All instance variables should be declared `private` to maximize information hiding. Any access to these variables should be done with accessor methods .

7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `HashDictionary` tests: 4 marks.
- `Configurations` tests: 4 marks.
- Coding style: 2 marks.
- `Data` and `HashDictionary` implementation: 4 marks.
- `Configurations` 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**. Remember that the TA's will test your program on the computers of the Department.

If you re-submit your program we will take the last program submitted as the final version, and will deduct marks accordingly if it is late.

It is your responsibility to ensure that your assignment was received by the system.