

# CS 2210 — Data Structures and Algorithms

## Assignment 2

Due Date: October 18 at 11:55 pm

Total marks: 20

## 1 Overview

Consider the following game. Two players take turns placing tiles on a 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. However, at least  $k$  positions on the board need to be empty, where  $k$  is a value specified before the game starts. After the players have placed  $n - k$  tiles on the board, so that there are exactly  $k$  empty positions on the board, and no player has won then the players take turns sliding one of their tiles into an adjacent empty position; tiles can be slid horizontally, vertically, or diagonally.

If the number of empty positions is  $k$  and on their turn a player does not have any tiles adjacent to the empty positions of the board the game ends in a draw.

For this assignment you need to write a Java program that plays the above board game. Your program will play against a human opponent. The human player uses blue tiles and always starts the game. The computer uses red tiles.

Figure 1 (a) shows a possible set of tiles on a board of size  $3 \times 3$  where  $k = 1$ . If the next player to move is the human then any of the tiles in positions 2, 3, or 9 can be shifted to 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).

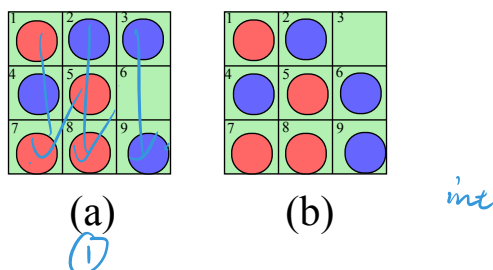


Figure 1: Board game.

You will be given code for displaying the game-board on the screen and part of the code for playing the game. There are several Java classes that you need to implement as described in the next sections. If you are interested in knowing how the algorithm for playing the above game works a document explaining this will be posted in OWL.

## 2 Board Layouts

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 2(b)), so the score for the computer

playing in position 8 is 2. Note that the board displayed in Figure 2(b) is a draw because now it is the turn of the human player, and no blue tile can be moved to the empty spot. We say that the *score for the board layout* in Figure 1(b) is 2, where a *board layout* is simply the positioning of the tiles on the game-board. Similarly, the score for the board layout in Figure 1(c) is 3, since in this case the computer wins the game.

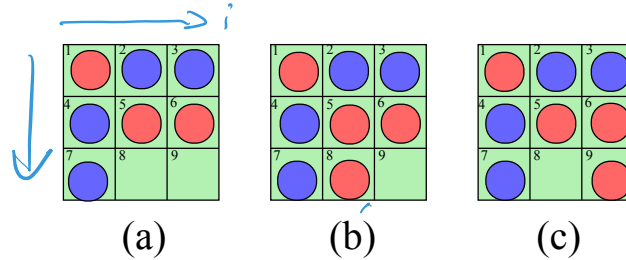


Figure 2: Board layouts.

### 3 Classes to Implement

You are to implement at least 4 Java classes: `Layout.java`, `Dictionary.java`, `DictionaryException.java`, and `Board.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 method `hashCode()` or Java's classes `Hashtable` or `HashMap`.

#### 3.1 Class Layout

An object of this class stores in two instance variables a board layout and its associated integer score, as explained above. Each board layout will be represented as a String as follows. A blue human's tile is represented with the character 'h', a red computer's tile with 'c', and a green empty space with 'e'. To form a String for a given board layout we concatenate the characters corresponding to the tiles and empty positions currently on the board starting at the upper left position and moving top to bottom and left to right. For example, for the layouts in Figure 2, their String representations are "chhhcchce", "chhhcchce", and "chhhcehcc".

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

- `public Layout(String boardLayout, int score)`: A constructor which initializes a new `Layout` object with the specified attributes `boardLayout` and `score`. The string `boardLayout` will be used as the key attribute for every `Layout` object.
- `public String getBoardLayout()`: Returns the `boardLayout` key attribute stored in a `Layout` object.
- `public int getScore()`: Returns the score stored in a `Layout` 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 Class Dictionary

This class implements a dictionary using a hash table with separate chaining. You will select the size of the table, keeping in mind that the size of the table must be a prime number and that it should

not be larger than 10,000. This size will be specified in method `makeDictionary` of class `Board` (see Section 3.4).

You must design your hash function so that it produces few collisions. A bad hash function that induces many collisions will result in a low mark. 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(Layout data) throws DictionaryException;
    public void remove(String boardLayout) throws DictionaryException;
    public int getScore(String boardLayout);
}
```

The descriptions of these methods follows:

- `public int put(Layout data) throws DictionaryException`: Inserts the `Layout` object referenced by `data` in the dictionary if the dictionary does not contain any object with the same key attribute as `data`; otherwise this method must throw a `DictionaryException`.

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 `data` 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  $h(\text{key})$  and the name of your table is  $T$ , this method will return the value 1 if the list stored in  $T[h(\text{data.getBoardLayout}())]$  already stores at least one element; it will return 0 if  $T[h(\text{data.getBoardLayout}())]$  is null or an empty list.

- `public void remove(String boardLayout) throws DictionaryException`: Removes the object with key `boardLayout` from the dictionary; must throw a `DictionaryException` if there is no data item in the dictionary with this key.

- `public int getScore(String boardLayout)`: A method which returns the score stored in the object in the dictionary with key `boardLayout`, or -1 if there no object in the dictionary with that key.

Since your `Dictionary` class must implement all the methods of the `DictionaryADT` interface, the declaration of this class 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 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 type `Layout` and a reference or pointer to an 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 Class DictionaryException

This is just the class implementing the class of exceptions thrown by the `put` and `remove` methods of `Dictionary`.

### 3.4 Class Board

This class stores information about the tiles placed on the game-board and implements support methods needed by the algorithm that plays the game. The constructor for this class must be as follows

✓ `public Board (int board_size, int empty_positions, int max_levels)`

The first parameter specifies the size of the game-board, the second one is the number of positions on the board that must remain empty, and the third one 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 so the program plays OK and it is not too slow).

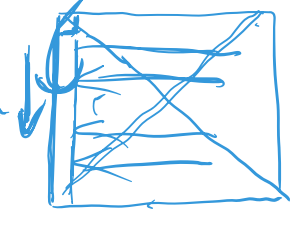
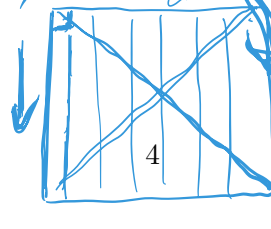
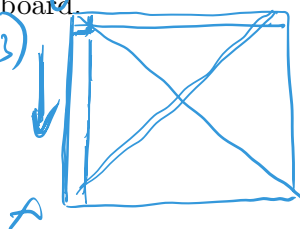
This class must have an instance variable called `theBoard` of type `char[] []` to store the game-board. This variable is initialized inside the above constructor method so that every entry of `theBoard` stores 'e'. As the game is played, every entry of `theBoard` will store one of the characters 'h', 'c', or 'e'. This class must implement the methods from the interface `BoardADT.java`, so the declaration of this class should be as follows

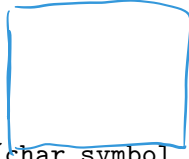
`public class Board implements BoardADT`

The methods that you must implement in this class are the following.

- ✓ `public Dictionary makeDictionary():` returns an empty `Dictionary` of the size that you have selected.
- ✓ `public int repeatedLayout(Dictionary dict):` This method first represents the content of the 2-dimensional array `theBoard` as a String `s` as described in Section 3.1; then it checks whether there is a data item in the dictionary referenced by `dict` with key `s`: If there is such a data item, this method returns the associated score; otherwise it returns the value -1. *boardLayout;*
- ✓ `public void storeLayout(Dictionary dict, int score):` This method first represents the content of `theBoard` as a String `s` as described in Section 3.1; then it creates a Layout object storing `s` and `score` and stores this object in `dict`.
- ✓ `public void saveTile(int row, int col, char symbol):` This method stores `symbol` in `theBoard[row][col]`.
- ✓ `public boolean positionIsEmpty (int row, int col):` This method returns true if `theBoard[row][col]` is 'e'; otherwise it returns false.
- ✓ `public boolean isComputerTile (int row, int col):` This method returns true if `theBoard[row][col]` is 'c'; otherwise it returns false.
- ✓ `public boolean isHumanTile (int row, int col):` Returns true if `theBoard[row][col]` is 'h'; otherwise it returns false.
- ✓ `public boolean winner (char symbol):` Returns true if there are `n` adjacent tiles of type `symbol` in the same row, column, or diagonal of `theBoard`, where `n` is the size of the game-board.

*4 = 4, 2, 2, 2, 2  
=> 10 = 4, 1, 1, 1, 1  
true/false  
=> (1, 2, 2, 2, 2, 2, 2, 2, 2, 2)*



- 
- `public boolean isDraw(char symbol, int empty_positions)`: Returns true if the game layout corresponding to `theBoard` is a draw assuming that the player that must perform the next move uses tiles of the type specified by `symbol`. The second parameter is the number of positions of the game-board that must remain empty.

Remember that a game is a draw if no player has won and either:

- `empty_positions = 0` and there are no empty positions left on the game board, or
- `empty_positions > 0` and none of the empty positions on the game-board has a tile of the type specified by `symbol` adjacent to it.

- `public int evaluate(char symbol, int empty_positions)`: Returns one of the following values:

- ▷ 3, if the computer has won, i.e. there are  $n$  adjacent 'c's in the same row, column, or diagonal of `theBoard`.
- ▷ 0, if the human player has won, i.e. there are  $n$  adjacent 'h's in the same row, column, or diagonal of `theBoard`.
- ▷ 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`. The second parameter is the number of positions of the game-board that must remain empty.
- ▷ 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`, `BoardADT`, `PosPlay.java` and `PlayGame.java` from OWL. Class `PosPlay` is an auxiliary class used by `PlayGame` to represent plays. Class `PlayGame` includes the main method for your program. The program will be executed by typing

```
java PlayGame size empty_positions max_levels
```

where `size` is the size of the game-board, `empty_positions` is the number of positions on the game-board that must remain empty, and `max_levels` is a parameter specifying the playing quality of the program. Remember that the larger the value of `max_levels` is the better the program will play, but the slower it will be.

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

**Note.** To place a tile on the board just click on desired position of the board. To slide a tile first click on the tile to select it and the click on an adjacent empty position to move it.

## 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 `Board`. 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 instance variable must be used 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 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 class `Layout`).

## 7 Marking

Your mark will be computed as follows.

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

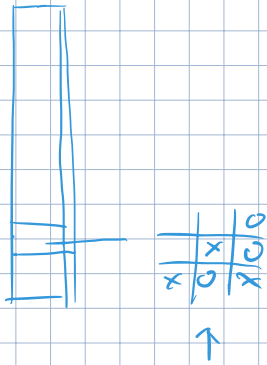
## 8 Handing In Your Program

You must submit an electronic copy of your program. To submit your program, login to OWL and submit **your java** files from there. **Make sure you do not just submit you .class files.** 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 use Eclipse, please read the tutorials about how to configure Eclipse to read command line arguments (select the tab FAQ in OWL).



Hash Table



The board is represented  
as a string and stored in  
a node of this hash table.