**CS 2210b — Data Structures and Algorithms**
**Assignment 2: Blocked Tic-Tac-Toe**
Due Date: February 14 at 11:59 pm
Total marks: 20

# 1 Overview

For this assignment you are required to write a Java program that plays *blocked* $(n, k)$*-tic-tac-toe*. Blocked $(n, k)$-tic-tac-toe is played on a board of size $n \times n$ where some positions are blocked or unavailable to the players, and to win the game a player needs to put $k$ symbols in-line, i.e. on adjacent positions of the same row, column, or diagonal. The program will play against a human opponent. You will be given code for displaying the gameboard on the screen.

# 2 Blocked Tic-Tac-Toe

The program will randomly decide whether the human player or the computer starts the game. The human uses 'x's and the computer uses 'o's. We will use 'b' to denote a blocked position. 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: This score is assigned to a move that will ensure that the human player will win.
- 1: This score is assigned to a move that will lead to a draw or a tie.
- 2: This score is assigned to a move that does not allow us to determine which player will win. This score will be explained in more detail later in this section.
- 3: This score is assigned to a move that guarantees that the computer will win the game.

For example, suppose that the gameboard looks like the one in Figure 1(a). Position 3 of the board (the positions are indicated by the small numbers in the figure) is blocked so no player can move there. 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 1. Instead of saying that "the score for the computer playing in position 8 is 1", we will say that "the score for the configuration in Figure 1(b) is 1", where a *configuration* is simply the positioning of the symbols on the gameboard. Similarly, the score for the configuration in Figure 1(c) is 0, since in this case the human player wins the game.

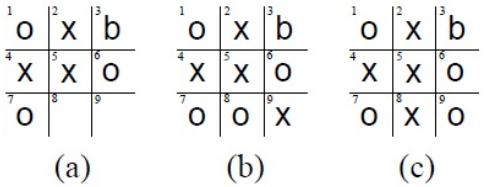


Figure 1: Board configurations.

The rest of this section explains how the algorithm to play blocked tic-tac-toe works. You do not need to understand this part to complete the assignment. To compute scores, your program will use a recursive algorithm that repeatedly simulates 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 structure formed by all the moves that the players can make starting at the current configuration. This tree structure is called a *game tree*. An example of a game tree is shown in Figure 2.

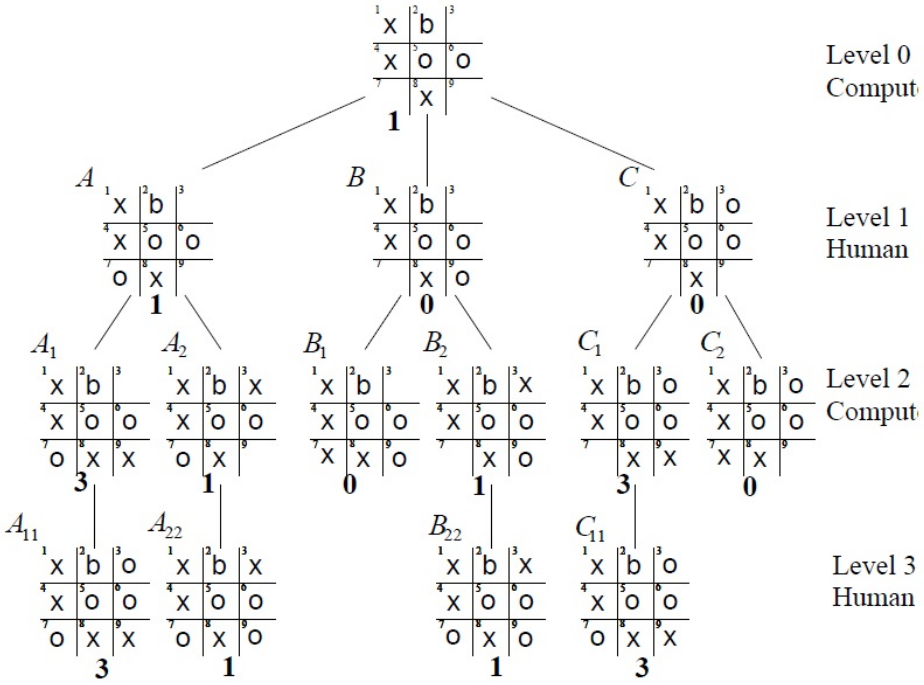


Figure 2: A game tree.

Assume that after several moves the gameboard is as the one shown at the top of Figure 2 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: $A_1$, $A_2$, $B_1$, $B_2$, $C_1$, and $C_2$. 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 2 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 $A_{11}$, $A_{22}$, $B_1$, $B_{22}$, $C_{11}$, and $C_2$, 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 upward as follows:

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

The scores for the configurations in Figure 2 are the numbers in boldface. Since for the board at the top of Figure 2, putting an 'o' in position 7 yields the configuration with the highest score, then the computer will choose to play in position 7. 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 `PlayTTT.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.squareIsEmpty(row,column)) { // Empty position found
          t.storePlay(row,column,symbol);
          if (t.wins(symbol)||t.isDraw()||(level == max_level))
            reply = new PosPlay(t.evalBoard(),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. 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 2 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 $B_1$ 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 $A_1$, $A_2$, $B_2$, and $C_1$ will receive a score of 2.

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

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 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.

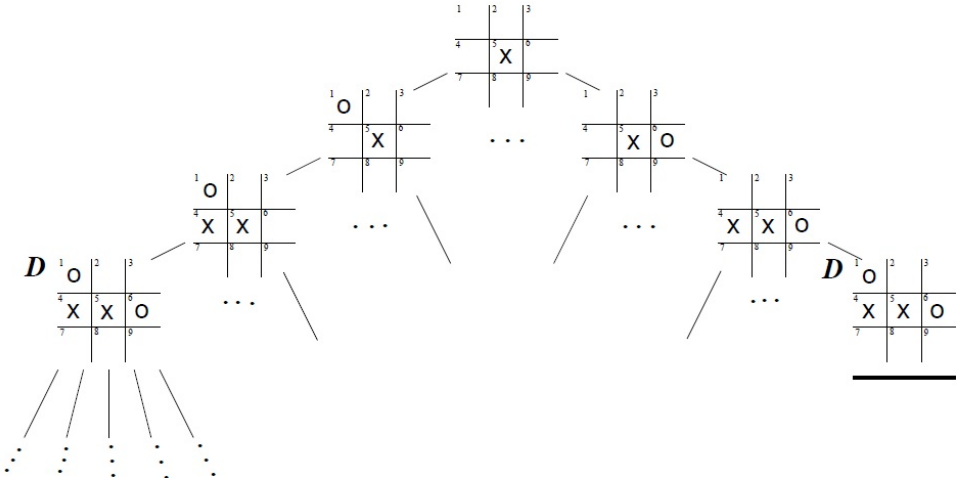


Figure 3: Detecting repeated configurations..

## 3 Classes to Implement

You are to implement at least 5 Java classes: `TTTRecord.java`, `TTTDictionary.java`, `DuplicatedKeyException.java`, `InexistentKeyException.java`, and `BlockedTicTacToe.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 or Java's `hashCode` method.

### 3.1 TTTRecord

This class represents one of the configurations that will be stored in the dictionary along with its integer score and level. Each board configuration will be represented as a string obtained by concatenating all characters placed on the board starting at the top left position and moving from top to bottom and left to right. For example, for the configurations in Figure 1, their string representations

are "oxoxx␣bo␣"[1], "oxoxxobox", and "oxoxxxboo".

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

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

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 TTTDictionary

This class implements a dictionary. You must implement the 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. The size of your table **must** be between 4000-6000.

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.) Collisions must be resolved using separate chaining. You cannot use Java's class `HashTable` or method `hashCode`.

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

```
public interface TTTDictionaryADT {
    public int put (TTTRecord record) throws DuplicatedKeyException;
    public void remove (String config) throws InexistentKeyException;
    public TTTRecord get (String config);
    public int numElements();
}
```

The description of these methods follows:

- `public int put(TTTRecord record) throws DuplicatedKeyException`:

  Inserts the given `TTTRecord record` in the dictionary. This method must throw a `DuplicatedKeyException` if `record.getConfiguration()` 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 `record` 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 hash 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 before the insertion.

- `public void remove(String config) throws InexistentKeyException`:

  Removes the record with the given key `config` from the dictionary. Must throw an `InexistentKeyException` if the configuration is not in the dictionary.

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

- `public TTTRecord get(String config)`:

  A method which returns the TTTRecord stored in the dictionary for the given configuration, or null if the configuration is not in the dictionary.

- `public int numElements()`:

  This method returns the number of `TTTRecord` objects stored in the dictionary.

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

```
public class TTTDictionary implements TTTDictionaryADT
```

You can download the file `TTTDictionaryADT.java` from the course's website. The only other public method that you can implement in the `TTTDictionary` class is the constructor method, which must be declared as follows

```
public TTTDictionary(int size)
```

this returns an empty TTTDictionary 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).

### 3.3 DuplicatedKeyException and InexistentKeyException

This are just the classes implementing the classes of exceptions thrown by the `put` and `remove` methods of `TTTDictionary`.

### 3.4 BlockedTicTacToe

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

```
public BlockedTicTacToe (int board_size, int inline, int max_levels)
```

The first parameter specifies the size of the board, the second is the number of symbols in-line needed to win the game, and the third is the maximum level of the game tree that will be explored by the program. So, for example, to play the usual $(3, 3)$-tic-tac-toe, the first two parameters will have value 3.

This class must have an instance variable `gameBoard` of type `char[][]` to store the gameboard. This variable is initialized inside the above constructor method so that every entry of `gameBoard` stores a space ' '. Every entry of `gameBoard` stores one of the characters 'x', 'o', 'b', or ' '. This class must also implement the following public methods.

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

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

- `public void storePlay(int row, int col, char symbol)`: Stores the character `symbol` in `gameBoard[row][col]`.
- `public boolean squareIsEmpty (int row, int col)`: This method returns true if `gameBoard[row][col]` is ' '; otherwise it returns false.
- `public boolean wins (char symbol)`: Returns true if there are $k$ adjacent occurrences of `symbol` in the same row, column, or diagonal of `gameBoard`, where $k$ is the number of required symbols in-line needed to win the game.
- `public boolean isDraw()`: Returns true if `gameBoard` 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 are $k$ adjacent 'o's in the same row, column, or diagonal of `gameBoard`;
  - 0, if the human player has won.
  - 1, if the game is a draw, i.e. there are no empty positions in `gameBoard` and no player has won.
  - 2, if the game is still undecided, i.e. there are still empty positions in `gameBoard` and no player has won.

# 4 Classes Provided

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

```
java PlayTTT size in_line max_levels list_of_blocked_positions
```

where `size` is the size of the gameboard, `in_line` is the number of symbols that need to be placed in-line to win the game, `max_levels` is the maximum number of levels of the game tree that the program will explore, and `list_of_blocked_positions` is a list of positions of the game board that are unavailable to the players (this list consists of a set of numbers separated by spaces). For example, to play a usual tic-tac-toe game with the same blocked positions as in Figure 1(a), the program could be executed as

```
java PlayTTT 3 3 4 3
```

Class `PlayTTT` also contains methods for displaying the gameboard 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 blocked tic-tac-toe. For testing the dictionary we will run a test program called `TestDict` which verifies 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 chosen to 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 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 variables declared outside methods ("instance variables") should be declared `private` (not `protected`), to maximize information hiding. Any access to these variables should be done with accessor methods (like `getScore()` for `TTTRecord`).

## 7 Marking

Your mark will be computed as follows.

- Program compiles, produces meaningful output: 2 marks.
- `TTTDictionary` tests: 4 marks.
- `BlockedTicTacToe` tests: 4 marks.
- Coding style: 2 marks.
- Hash table implementation: 4 marks.
- `BlockedTicTacToe` 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 read the tutorials on how to configure Eclipse to read command line arguments. Please **DO NOT** put your code in sub-directories; you will be penalized if you do this, as sub-directories make it harder for the TA's to mark your assignments.

When you submit your program, we will receive a copy of it with a datestamp and timestamp. 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.