**Assignment 8**

1. Add the following method to both classes *ArraySortedList* and *RefSortedList:*

***public int removeAll(T element)***

*Precondition: the list may or may not contain the "element"*

*Post-condition: all occurrences of "element" are removed from the list; the number of elements removed is returned*.

(Note that if you use the combination of methods ‘*contains’* and ‘*remove’*, then the implementation of the method can be trivial, and it applies to both cases (array-based and link-based); but the approach is inefficient. So, think about a different approach, especially one that doesn’t require temporary storage.)

1. A list application

You’d like to simulate a racing game among N players. The players move in sequence starting with player 1 and ending with player N, and the same cycle repeats as necessary. On each move, a player goes forward or backward depending on the player’s position change for the move. A positive position change for a move implies a forward move, whereas a negative position change results in a backward move. As players move in sequence, the first player whose total move exceeds certain number of units (say M units) wins, and the game is over. The position change a player receives on each move is determined by:

* The outcome of throwing a fair die (as needed, you may want to search Web a bit to learn how it is done in code)
* The relative position of the player that makes a move with respect to the other players.
* The move logic a player picks. (Move logic determines how the die-throw outcomes and the relative position of the other players may affect the given player’s position change and hence the move by the player.)

There are three distinct types of move logic. Each player is *randomly* select one of the three when making a move. The details related to the three types of move logic are given below.

Move Type 1

The change in position is computed as:

*die throw + ( position of leading player – player’s position ) / 2*

*If the die throw is 3, 4, 5, or 6, then the change in position is positive; otherwise it is negative.*

Move Type 2

The change in position is computed as:

*3 times the die throw if the die throw is an even number; otherwise the die throw if the die throw is an odd number.*

(Thus, the change in position is always positive.)

Move Type 3

The change in position is computed as:

*die throw + (player’s position – position of the trailing player) / 2*

*If the die throw is 1 or 2, then the change in position is positive; otherwise it is negative.*

**Program output:**

After each player’s move, a line of output should be sent to the console that identifies the player’s number and his/her current position. When the game is over, the program displays a table that identifies the players and their final positions (the format of the table is at your discretion).

*Here is an example with 4 players:*

At some point, the four players are at positions shown in the following table:

Player 1 2 3 4

Position 10 5 12 7

Suppose player 3 is making a move. She selects move type 2 (by luck). So she throws a die (assuming it’s a 2). Thus her position change would be 3\*2 = 6 (according to the type 2 move logic). So her new position would be 18. Now, game continues and it’s player 4’s turn….

**Note:** there are obviously different ways to implement the program. You may even wonder whether you might use the “chain of responsibility” pattern to do it (yes, it can work). But since we are learning sorted lists, why not to think of an implementation along that line? (Hint: if the list is sorted based on players’ positions, then if a player makes a move, it gets removed from the list, and added back in when the move is completed … in that way, you would always know who the leading player is and who the trailing player is without additional calculations?)