CS 2420 Program 6 – 20 points  
Hex Union Find

**Part 1: Union Find** Write the code to perform union find (using path compression) on a set of 121 (or more) elements. Use a smart union (by size or height, your choice). Create a series of carefully constructed tests so that you can verify union/find is working properly and that path compression works. Print out the contents of your array.

Everyone’s tests will be different. You will be graded on how well your tests demonstrate proper functioning. This ability to test your code (and dig into the details) is critical.

The Union/Find you create should work for any problem with integer items, and not be coded to only work for this assignment. In other words, union/find doesn’t care who is using it. If you wanted a union/find with non-integer values, you would have to do convert between the values and integers – as union find only works for integers.

Operations on this data structure would include:

1. UnionFind(int size) Constructor: create union find instance of a given size. n
2. Int find(int item) Return the group containing item
3. void union(int item1, int item2): union item1 and item2

**Part 2: Hex** The game of Hex is played on a grid of hexagons. Hex is a strategy board game for two players played on a grid, theoretically of any size and several possible shapes, but traditionally as an 11×11 rhombus. Players (red and blue) alternate placing markers or stones on unoccupied spaces in an attempt to link their opposite sides of the board in an unbroken chain. In the figure below, the blue player is attempting to link the two blue edges, while the red player is attempting to link the red edges. Write the code to detect when red or blue has won the game.

While the logical view of the data is a rhombus, it can actually be stored as a single dimensioned array. This makes using the union-find simple (as union find works with integers). Thus, we need to convert our logical rhombus-based view of the data to be a simple array representation. To do that we need a formula to convert from our model to the physical storage. In fact, we have two arrays to represent the cells. One to store the color stored at each location and the second to be used by union find. We can think of this storage as parallel arrays. Recall, parallel arrays are multiple arrays of the same size such that i-th element of each array is closely related and all i-th elements together represent an object or entity. Logically, we could have an array of classes, but since union find wants integers, parallel arrays are more convenient.

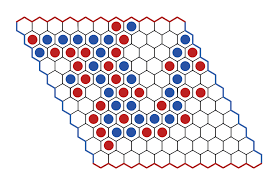
Table

Description automatically generatedcA picture containing vector graphics

Description automatically generated

Use the union-find data structure to determine when opposite edges have been connected. We will assume that the hexagons are numbered by rows, with the first row being 1-11, the next row being 12-22 and so on. With this numbering system, the neighbors of a non-edge node x are items x-1, x+1,x-11, x-10, x+10, x+11.

1. Keep track of which cells have been selected by each player. My cells were coded BLUE, RED, or NONE.
2. Use the smart union find (with path compression) to keep track of the set of cells that are connected.
3. You will need a plan to decide how you can tell if the edges are linked. Our goal with union/find is to have an almost constant time operation. Thus, anything you add has to maintain this goal.
   1. One way to tell is edges have been connected would be to have extra array element associated with the TOP, BOTTOM, LEFT, and RIGHT. In my case, RIGHT=122 LEFT=123, TOP=124, BOTTOM = 125, (so I could treat them just like the hexagons). The color specified for TOP and BOTTOM is RED. The color specified for LEFT and RIGHT is BLUE. For the red player, when any item on the top row is selected, union it with item TOP. When any item on the bottom row is selected, union it with item BOTTOM. The game is won for the red player when find(TOP)=find(BOTTOM). Blue wins in the game below.



* 1. Do not check to see if the boundaries are reached by seeing if any red item in row 1 is in the same group as any red item in row 11 as this would be very slow.

Input:

The input is a series of moves indicated by the board location selected. Assume that the blue player always goes first. Indicate who wins and how many moves were made. If a player attempts to select a cell that has already been played, flag this as an error and continue. Test your code with the supplied input files.

**Output**

Print out the board, who wins, and the number of moves required. This will make debugging much easier. You can print the board in color by

public static final String ANSI\_RESET = "\u001B[0m";

public static final String ANSI\_BLACK = "\u001B[30m";

public static final String ANSI\_RED = "\u001B[31m";

public static final String ANSI\_GREEN = "\u001B[32m";

public static final String ANSI\_YELLOW = "\u001B[33m";

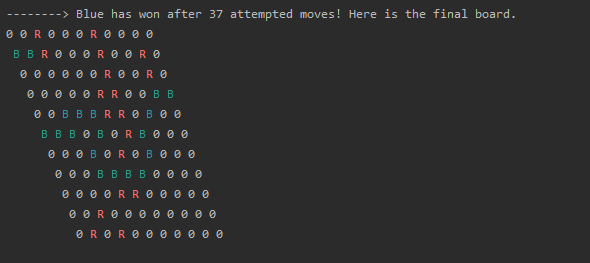
public static final String ANSI\_BLUE = "\u001B[34m";

public static final String ANSI\_PURPLE = "\u001B[35m";

public static final String ANSI\_CYAN = "\u001B[36m";

public static final String ANSI\_WHITE = "\u001B[37m";

System.out.println(ANSI\_RED + "This text is red!" + ANSI\_RESET);



**Hint:** The awkward part is deciding who the neighbors of a node are. If a cell is in the center, the neighbors are found by adding the following offsets to item: int[] inc= {-11, -10, -1, 1, 10, 11}; These offsets correspond to neighbors in the following directions: NW,NE,W,E,SW,SE.

The tricky part is that the corners only have neighbors (not counting LEFT, RIGHT, TOP, BOTTOM) in two of the six directions. All other edges have neighbors in four directions, but each edge is different. It is nice if you have a separate method to find neighbors as it simplifies the code.

One choice is to create a getNeighbors method which returns only the legal neighbors. If the neighbor was at a border, I returned the border (LEFT, RIGHT, TOP, BOTTOM) as a neighbor. The border numbers know what color they are associated with, so a blue move will not union with TOP or BOTTOM (as those “cells” aren’t associated with BLUE).

int RIGHT=122;

int LEFT = 123

int TOP=124;

int BOTTOM=125;

**private int[] g**etNeighbors( **int** item)

Output from this routine may look like:

Neighbors of 83 [72,73,82,84,93,94]

Neighbors of 121 [110,120,122,125]

Neighbors of 62 [51,52,61,63,72,73]

Neighbors of 84 [73,74,83,85,94,95]

Neighbors of 112 [101,102,111,125]

**Bonus (2 points)**

Allow the game to be parameterized so that the number of rows and columns can be specified by the user.