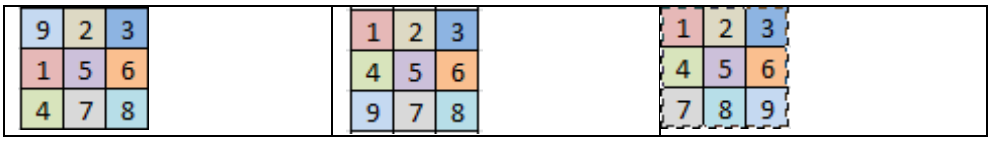
CS 2420 –Sprint 2018 – Check Canvas for due date
Program 1 – 20 points
Rotation Puzzle

The problem. The rotation puzzle is played on a 3-by-3 grid with 9 square blocks labeled 1 through 9. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically with wrapping. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right) by rotating north on column 0 and then west on row 2.



initial goal

Write a program to solve the rotation puzzle problem in the minimal number of moves using a brute force technique. You will use a queue to store all the possibilities you want to explore. **In order to refresh pointer skills, you must use a linked list representation of a queue. This needs to be code that you have written (possibly last semester).** Starter code for the board is provided. Feel free to modify it (or discard it) as you see fit.

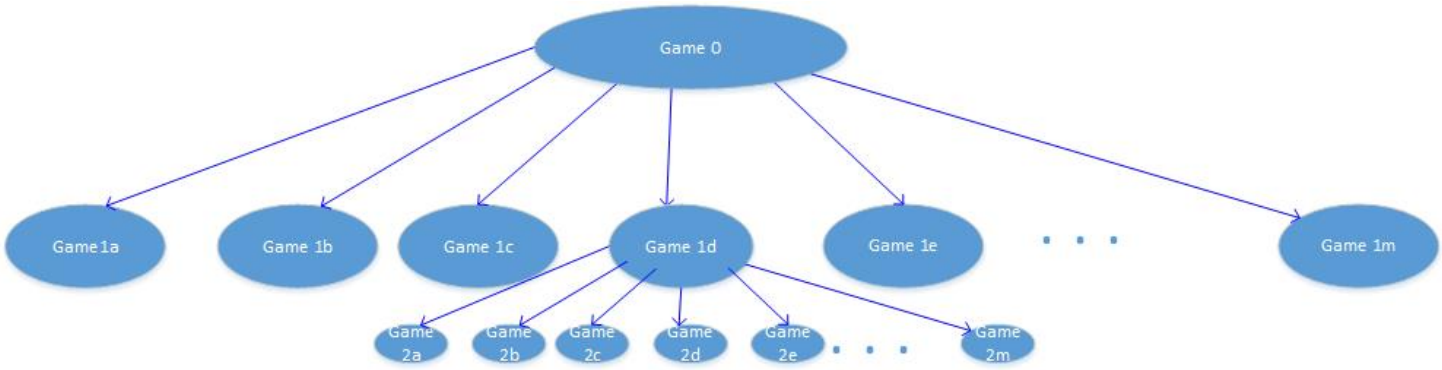
One way: You could just randomly try moves, each time checking if you have found a solution. While this would likely work eventually, you wouldn't find the minimal sequence of moves.

Our way: We are going to be more methodical. Instead of picking a particular move, we say, "I have twelve possible moves. I don't know which one to try. Let me try each of them and then (later) pick the one that was the best." We try all choices from each board we reach. We use a queue to remember all of the partial games we are considering. We call a partial game a GameState.

A logical tree helps us consider the possibilities. This is a LOGICAL tree, not a coded tree. Starting at "Game 0" (the initial game board) you can make one of twelve moves (One of four directions (North, South, East West) and one of three rows or columns). From each of those moves, you can make twelve moves. [We would say this is a tree with a twelve way branching factor.] We can keep expanding nodes until a "leaf" is the desired game. That represents an exhaustive search. However, we haven't said which order we expand the nodes. Are we going to keep going deeper in the tree [a depth first search]? This has the advantage of being easy to do (as recursion takes care of the returning to a previous node), but has some disadvantages.

We are going to approach it slightly differently. Instead of doing one move after another – we are going to consider all boards reachable with one move, then all boards reachable with two moves, etc. So in the picture below, we consider Game 0, then Game 1a, Game 1b, Game 1c, Game 1d, Game 1e, … Game1m, and then all the games at the next level. This is like traversing the logical game tree "by level". This is a called a breadth-first traversal. We examine all one move positions, then all two move positions, etc.

The bad news is recursion can't help us with this kind of traversal.



Because we try lots of possibilities, we need to record the state of the game we are working on. We define a *state* of the game to be the board and the history of how we got to the board. In our case, state is (1) the board, (2) the stateID (a sequential number we assign), (3) the previous stateID, (4) all the moves to get from the initial game to this board. First, insert the initial state into a queue. Then, delete from the queue a state, and insert onto the queue all neighboring states (those that can be reached in one move) of the removed state. Repeat this procedure until you reach the goal state.

Optimization: You will notice some inefficiency to this procedure. In a readme.txt file submitted with your assignment, (1) Point out the inefficiencies (2) Describe what you would do to try to improve the algorithm

Your task. Using the following boards as input, print out the series of board positions you consider to solve each of the games.

Input 1
1 2 3
6 4 5
9 7 8

Input 2:
4 2 9
7 5 3
1 8 6

Input 3:
6 7 2
1 5 9
3 4 8

Input 4: Create a jumbled board [using makeBoard] and solve it.

Sample output (your output may vary):

I used >, <,^,v to represent rotations East,West,North,South, followed by the row or column number. Use whatever notation seems best to you.

```
State 0 From State -1 History
7 8 3
1 2 6
```

```
4 5 9
State 1 From State 0 History :^0
1 8 3
4 2 6
7 5 9
State 2 From State 0 History :v0
4 8 3
7 2 6
1 5 9
State 3 From State 0 History :>0
3 7 8
1 2 6
4 5 9
State 4 From State 0 History :<0
8 3 7
1 2 6
4 5 9
State 5 From State 0 History :^1
7 2 3
1 5 6
4 8 9
State 6 From State 0 History :v1
7 5 3
1 8 6
4 2 9
State 7 From State 0 History :>1
7 8 3
6 1 2
4 5 9
State 8 From State 0 History :<1
7 8 3
2 6 1
4 5 9
State 9 From State 0 History :^2
7 8 6
1 2 9
4 5 3
State 10 From State 0 History :v2
7 8 9
1 2 3
4 5 6
State 11 From State 0 History :>2
7 8 3
1 2 6
9 4 5
State 12 From State 0 History :<2
7 8 3
1 2 6
5 9 4
State 13 From State 1 History :^0:^0
4 8 3
7 2 6
1 5 9
State 14 From State 1 History :^0:v0
7 8 3
1 2 6
4 5 9
State 15 From State 1 History :^0:>0
3 1 8
4 2 6
7 5 9
State 16 From State 1 History :^0:<0
8 3 1
4 2 6
```

```
7 5 9
State 17 From State 1 History :^0:^1
1 2 3
4 5 6
7 8 9
YOU WIN!!! OriginalBoard
7 8 3
1 2 6
4 5 9
```

Hints

Use of assert

One of the biggest problems we have with pointers is following NULL pointers. You will be amazed at how many problems are solved by being cautious. Before you follow a pointer, always check to see if it is NULL. (In fact, many of the base cases in recursion are simply checking for a NULL pointer.) If you are absolutely sure the pointer cannot be NULL, I would still recommend checking it via an assert statement.

To use assert:

#include <cassert>

If you think something is true, you simply state

assert(statement);

If the statement evaluates to true, nothing happens. If it doesn't, you abort in a dramatic way (that can't be missed). Some have suggested that you really need to throw an exception that could be caught and dealt with properly. That is true, but this is a simple way of locating errors you don't expect to happen.

Use of stringstream

For each data structure, I create a "toString" function which gives a printable version of the data structure. This is better than just using cout, as I can easily write to a different location. Since the only way I print the contents of a data structure is by calling toString, it is easy to modify the view I want to see.

A stringstream allows you to use the power of input/output operators to create strings for you to use.
The following example shows how I create a string version of the Board.

```
#include <sstream>
// Create a printable representation of the Board class
string Board::toString() {
	stringstream ss;
	for (int i=0; i < SIZE; i++)
	{
		for (int j = 0; j < SIZE; j++)
			ss << board[i][j] << " ";
		ss << endl;
	}
	return ss.str();
};
```

My GameState looked like:

```
class GameState
{
   public:
```

```
        Board current;   // Contents of current board
        int stateID;     // State ID of the current board
        int prevID;      // State ID of the previous board
        string allMoves; // What are all the moves you made?
        string toString();  // Convert GameState to printable form
        GameState(Board curr, int prev, int currID, string all)
        {       current = curr;
                stateID = currID;
                prevID = prev;
                allMoves = all;
        }
        GameState()
        { current.makeBoard();
        }
};
```

What to Turn in:

Submit a zip file containing your software project and your readme.txt file. Read the document "Submitting Assignments" to make sure you have the needed files. It is included below for your convenience.

Submitting Assignments

How to Submit To Canvas

In an attempt to help avoid problems with incorrect project submissions, please follow these steps when submitting a project to Canvas.

1. Create a .zip file (if asked) of all the source and project files necessary to build the project. *Do not ever submit executable code.*
2. Upload the .zip (or other appropriate) file to Canvas.
3. Log out of Canvas.
4. (Preferably on another computer) Log into Canvas.
5. Go to the project submission and download the assignment you just submitted.
6. Extract files to a location different from where they were prepared for submission (if doing this on the same computer as the submission).
7. Verify the project correctly builds and runs
8. In the comments section of the submission, enter the word "verified", indicating that you have gone through these steps to ensure you have submitted the correct project, and that the project you have submitted builds and runs correctly.

What to Submit - Visual Studio

Create a zip file of your project. Never move any files around before making the zip file, maintain the folder/file structure. You will, however, need to selectively remove some files and folders before making the zip file.

1. All of the follow files types should be included:
    1. All .cpp files
    2. All .h or .hpp files
    3. The .sln file

    4. The .vcxproj file
    5. The .vcxproj.filters file
2. None of the following files or folders should be included:
    1. *.db
    2. *.exe
    3. *.obj
    4. *.ilk
    5. *.pdb
    6. /x64 folders
    7. /Debug folders
    8. /Release folders

What to Submit - Linux/macOS

- Submit all .cpp and .h or .hpp files. No need to include the projects with them.