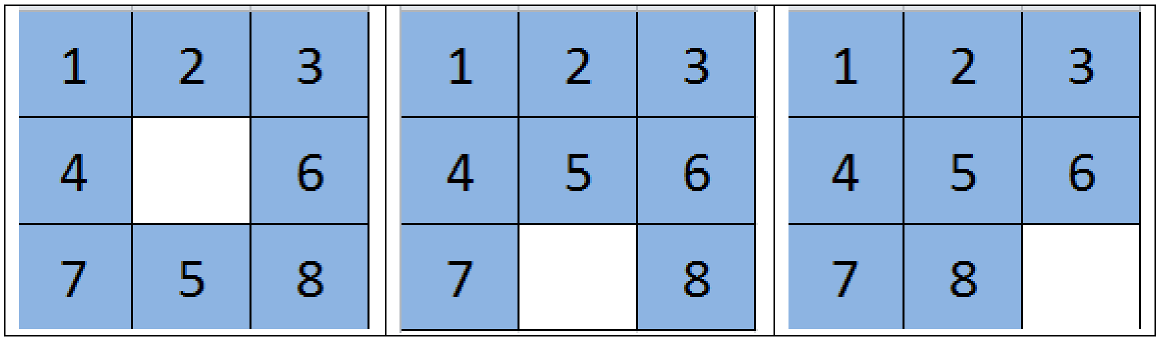
CS 2420 – Fall 2015 – See Canvas for Due Date
Program 1 – 20 points
Slider Puzzle

The problem. The slider puzzle is played on a 3-by-3 grid with 9 square blocks labeled 1 through 8 (with one blank). Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank spot. The following shows a sequence of legal moves from an initial board position (left) to the goal position (right) by first moving 5 UP into the blank spot and then moving 8 LEFT into the blank spot.



`initial` `goal`

If you want to try your skills, go to: http://mypuzzle.org/sliding.

Part 1 How to Make Moves

Use the code for the board which is provided (or code of your own). Write the "ShowMe" method which takes a board and a sequence of moves and shows the boards that result from those moves.

Sample Output:

```
OriginalBoard  Moves LDRUL
1  2  3
4  6  8
7  0  5
L==>
1  2  3
4  6  8
7  5  0
D==>
1  2  3
4  6  0
7  5  8
R==>
1  2  3
4  0  6
7  5  8
U==>
1  2  3
4  5  6
7  0  8
L==>
1  2  3
4  5  6
7  8  0
```

Part 2 Solving the Problem

For this problem, you are writing a program to SOLVE the puzzle, not to play the puzzle. **From a specific board, you are to illustrate an efficient solution. The file Prog1OUT.txt shows you an example output.**

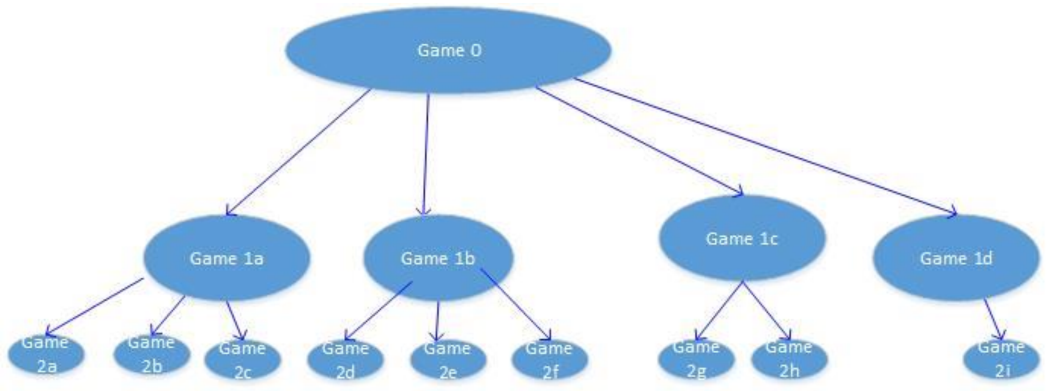
The Intelligence

One way: You could just randomly try moves, each time checking if you have found a solution. While this would likely work eventually, it could take a while.

Our way: We are going to be more methodical. Instead of randomly picking a particular move, we say, "I have four (or fewer) possible moves. I don't know which one to try. I want to try ALL of them. 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.

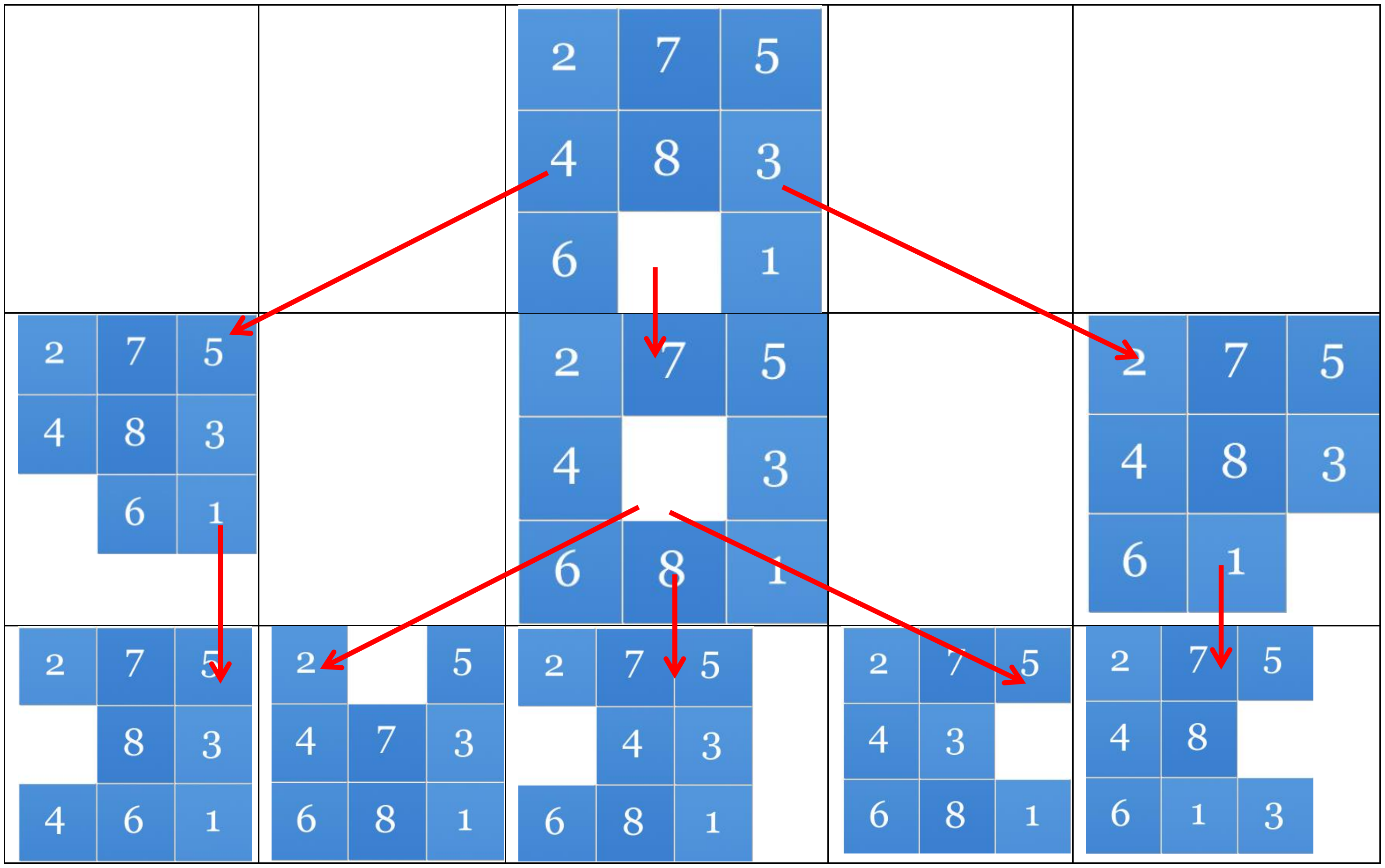
A logical tree helps us consider the possibilities. This is a LOGICAL tree, not a coded tree. Starting at "Game 0" you can make one of four moves (UP, DOWN, LEFT, RIGHT). [We would say this is a tree with a four way branching factor.] We can keep expanding nodes until a "leaf" is the desired (goal) game. 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, and then all the games at the next level. While some original games have four "next games", other games have fewer (due to the position of the blank and because we don't want to undo the previous move). This is like traversing the logical game tree "by level". This is a called a breadth-first traversal. We examine all one move states, then all two move states, etc.



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

Below is an example of the boards I would consider. The original game is the root. Notice that I eliminate moves which take me immediately back to the previous position. Can you explain why? Study the example below, by marking each arrow by the move that was taken to get to the next board.



Because we try lots of possibilities, we need keep all 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. First, insert the initial state into a queue. Then, delete the state from the queue, 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.

Input

Allow the user to either

a. Create a random board. The user will select the number of times to jumble a board (as this controls how difficult the board will be to solve). Make sure the game is legal.

b. Solve a specific board.

Your code should work correctly for arbitrary *N*-by-*N* boards (for any $1 \leq N \leq 100$), even if it is too slow to solve some of them in a reasonable amount of time.

Output

Use a file similar to SliderProject.cpp to generate test cases. The File prog1OUT.txt illustrates the format of the output

For each puzzle

1. Show the series of moves needed to solve the puzzle.
2. Show the number of boards placed on the queue and the number of boards removed from the queue.

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 <assert.h>
```

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).

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.

```
// 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();
};
```

What to Turn in:

Submit a zip file containing your software project and your readme.txt file.