Program 3 Fall 2015
20 Points
Sliding Puzzle Revisited AVL Trees

PART 1

You have been given the AVL tree code from your author as a starting point. Reading code is a valuable experience. On the job, you are often asked to modify/maintain existing code. You can't start over and do it your way. You must incorporate your changes into the exisiting code. **You are expected to understand the code that has been given to you.** Make the following changes:

1. Change all variable names to be meaningful names.
2. Make any changes needed to conform to our style guidelines.
3. Write a toString function which creates an indented version of the tree (as in program 2).
4. Implement the removeMin function. Remove the smallest node in the tree and rebalance. This function is to be your own work, not copied from ANY other source.

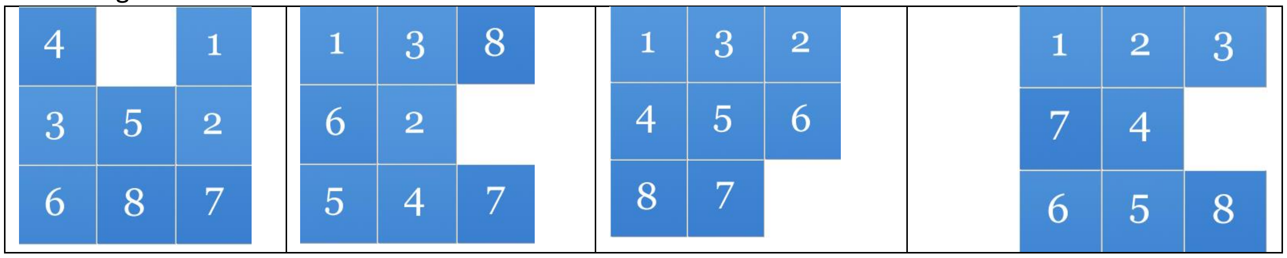
The data to be stored in the AVL tree is to be generic and work for either an integer or a GameState (of the slider puzzle variety).

Illustrate that the AVL tree is working properly by printing the tree (using toString) after each of the following groups of operations:

- Add: 1 3 5 7 9 11 2 4 8 (now print tree)
- Remove 7 9(now print tree)
- Add 50 30 15 18 (now print tree)
- Remove Min (now print tree)
- Remove Min (now print tree)
- Remove Min (now print tree)
- Add 17(now print tree)

PART 2

Use the AVL tree as a priority queue to improve your solution to the slider puzzle problem (of program 1). This is the idea. Suppose instead of looking at solutions in the regular order (which we used in program 1), what if you considered partial solutions which look closer? For example, which of the following is the closest?



A priority queue is a queue such that insertions are made in any order, but when you remove a node from the queue, it is the one with the best priority. Our priority will be an estimation of the total amount of work needed to solve the problem – so a lower score is preferred. Thus, we will first consider boards that "look better" to us.

In our previous solution, (program 1) , we considered all one-move solutions before considering all two move solutions, and so on. If we are smarter in which solutions we consider, we can improve the functioning of the solution. In this assignment, you will compare your previous solution to this new technique.

We define a *state* of the game to contain at least:

- board
- cost so far: number of moves taken from initial state to reach current board
- estimated cost to reach a solution
- priority (cost so far + estimated cost to reach a solution)
- String of moves indicating how the current board was achieved.

Best-first search. We describe a solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm.

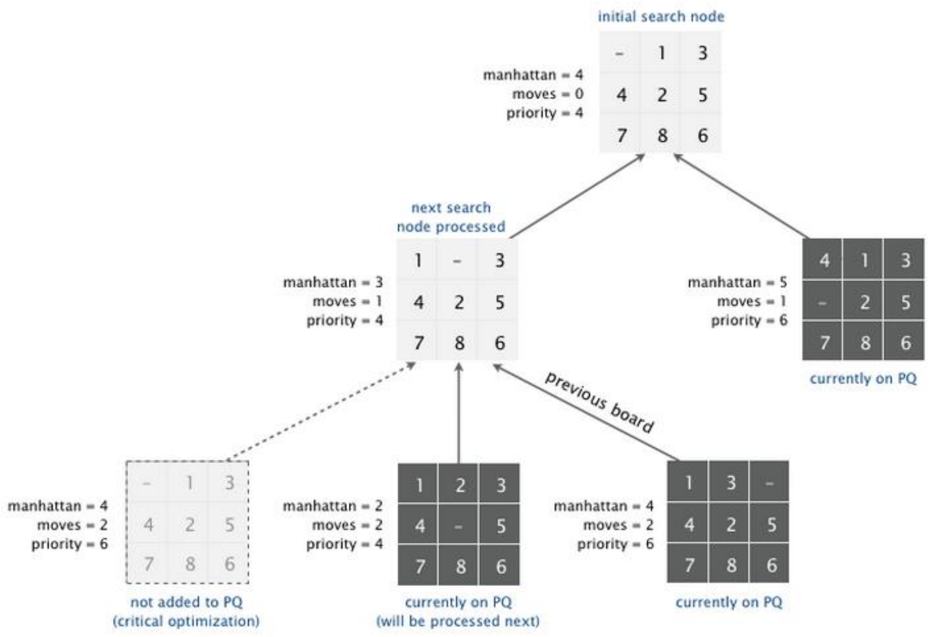
First, insert the initial state into a priority queue. Then, delete from the priority queue the state with the smallest priority, and insert onto the priority queue all neighboring states (those that can be reached in one more move) using an appropriate estimated cost for each. Repeat this procedure until the state removed from the priority queue is the goal state. The success of this approach hinges on the choice of estimated cost or *priority function*.

You can compute your "expected work function" any way you want as long it is reasonable and underestimates the real cost. Be creative. Here are two choices.

Manhattan distance: For each number (other than the blank), give each entry a score showing how far it needs to be slid to get in the right location. (This is termed a Manhattan distance after the gridlike street geography of Manhattan.)

Hamming distance: The number of tiles in the wrong position. Intuitively, a search node with a small number of tiles in the wrong position is close to the goal.

The diagram below shows how the Manhattan distance could be used to aid the search.



We make a key observation: to solve the puzzle from a given state on the priority queue, the total number of moves required (including those already made) is at least its priority, using our distance function.

Consequently, as soon as we dequeue a board which has the goal state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves. When we dequeue, we know everything else we will enqueue in the future will take at least as many moves to reach the goal. This is true because our distance is an UNDER-estimate of the number of moves required.

Output:

Since we want to compare this version with our brute force solution in program 1, you will need to have both methods working. Create a class "SliderGame" which allows you to call either "bruteForceSolve" or "aStarSolve" on the same board. Each method should keep track of the total number of states that were pulled off the queue.

1. Show the output for the four boards shown above under the "Part 2" heading using both methods (brute force and A*). Your output should include
   a. Which method was used
   b. the initial board
   c. the sequences of moves used to solve the board (i.e., URRDLL)
   d. the "show me" sequence which shows what the board looks like after each move.
   e. total number of boards dequeued from the queue.

2. Find an example for which your "more intelligent" method saves significant time over the brute force method of program 1. Show the same output for this board.

Hints

During debug, when using the A* search, show each board as you pull it off the queue. Be sure to print all the data in the board state (history, priority, expected moves remaining).

Print output both to the console and to a file. Then, you can easily remove the printing to the console (to make it run faster and easier to read).

Getting the same code to work for ints and GameStates forces us to be more methodical in our approach. For example:
The toString function, it looks something like:

```
string toString( AvlNode *t, string indent) const
{
stringstream ss;
if( t != nullptr )
{
  ss << toString(t->right, indent + " ");
  ss << indent <<t->element << endl;
  ss << toString(t->left, indent + " ");
}
return ss.str();
}
```

This works fine if element is an int, but not so great if element is a GameState. How do you get this to work? Ints don't have an element component. GameStates don't know how to print themselves.

AHHHHH. You need to overload << for GameStates.
It is done like:

```
ostream&  operator<<(ostream& ss, const GameState& gs) {
       ss << gs.toString() << endl;
       return ss;
}
```

Note however, that this cannot be a member function of GameState because the first parameter of << needs to be an ss. When you make something a member function of GameState, the first (understood) parameter is GameState. Right?

As an example: To overload == for the class MyClass:

```
  bool MyClass::operator==(const MyClass &other) const {
    ...  // Compare the values, and return a bool result.
  }
```

This allows us to do a==b, where both a and b are members of MyClass. == is a binary operator, but only the second argument is listed as a parameter. The "understood" (first) argument is a member of the class.

Thus, if we need to overload << and use it as ss << gameState, the << would HAVE to be a member function of ostream not of gameState. Since we have no access to ostream, we are stuck making it a non-member function.

What to Turn in:

Submit a zip file containing your software project . The zip file should contained a readme file which tells how to run the program (what IDE you used).