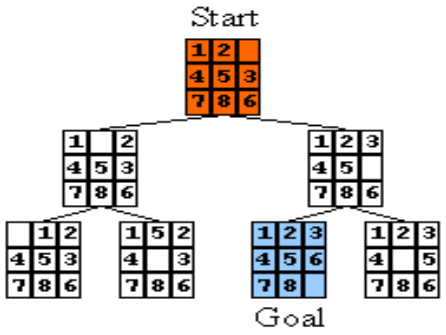
CS 302 – Assignment #08

Purpose: Learn concepts regarding priority queues and heaps.
Due: Tuesday (4/02) → Must be submitted on-line.
Points: Part A → 100 pts, Part B → 50 pts

Assignment:

Part A:



Design and implement a C++ class, ***priorityQueue***, to implement a priority queue[1] data structure. The ***priorityQueue*** class will implement a minimum Heap priority queue. The priority queue will be implemented in template.

Once the ***priorityQueue*** class is completed and fully tested, design and implement a **puzzle** class to solve a sliding puzzle[2] that consists of a frame of numbered square tiles in random order with one tile missing. The puzzle also exists in many sizes. If the size is 3×3 tiles, the puzzle is called the 8-puzzle. If 4×4 tiles, the puzzle is called the 15-puzzle.

Part B:

Create and submit a brief write-up including the following:

- Name, Assignment, Section.
- Summary of the ***priority queue*** data structure.
- Compare the priority queue data structure to using a balanced binary search tree. Include the associated trade-offs.
- The assignment requires using *buildHeap()* instead of calling the *insert()* function multiple times. Explain why and the consequences of doing it incorrectly.
- Big-O for the various priority queue operations (insert, buildHeap, deleteMin, reheapUp, reheapDown, and resize).

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:55.
- Submit a copy of the write-up (open document, word, or PDF format).

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

1 For more information, refer to: http://en.wikipedia.org/wiki/Priority_queue
2 For more information, refer to: http://en.wikipedia.org/wiki/15_puzzle

Make File:
You will need to develop a make file. You should be able to type:

```
make
```

Which should create the executable.

Class Descriptions

- Priority Queue Class
  The priority queue class will implement functions specified below.

priorityQueue<myType>
-heapNode: struct
-priority: int
-element: myType
-count: int
-heapSize: int
-myHeap: * heapNode
+priorityQueue(int=1000)
+~priorityQueue()
+entries() const: int
+insert(const myType, const int): void
+deleteMin(myType &, int &): bool
+isEmpty() const: bool
+printHeap() const: void
+readData(const string): bool
-reheapUp(int): void
-reheapDown(int): void
-buildHeap(): void
-resize(): void

Function Descriptions

- The *priorityQueue()* constructor should initialize the binary heap to an empty state. The parameter must be checked to ensure it is at least 1000 and, if not, the default value should be used.
- The *~priorityQueue()* destructor should delete the heap.
- The *entries()* function should return the total count of elements in the heap.
- The *insert()* function should insert an entry into the binary heap. If the count of heap entries exceeds the heap size, the heap must be expanded via the private *resize()* function. The heap properties must be maintained via the private *reheapUp()* function. The count should be updated.
- The private *buildHeap()* function should update the heap to apply the heap properties.
- The *deleteMin()* function should remove the minimum entry from the heap. The heap properties must be maintained via the private *reheapDown()* function. Additionally, the count should be updated. If the heap is already empty, the function should return false and otherwise return the minimum priority information (via reference) and return true.

- The *isEmpty()* function should return true if there are no elements in the heap and false otherwise.
- The *printHeap()* function should print the current heap in level order with a blank line between each level. Refer to the sample output for an example of the formatting.
- The *reheapUp()* function to recursively ensure the heap order property is maintained. Starts at tree leaf and works up to the root. Must be written recursively.
- The *readData()* function should read data from the passed file. The file format will be <dataItem> and then <priority>. If the ifle can ont be opened or read, the function should return false. If the file can be read, the data should be placed into the heap in the order read, the *buildHeap()* function called, and true returned.
- The *reheapDown()* function to recursively ensure the heap order property is maintained. Starts at tree root and works down to the applicable leaf. Must be written recursively.
- The *resize()* function should create a new heap array twice the size of the existing heap, copy all entries from the current heap into the new heap, and delete the old heap. The *heapSize* should be updated accordingly.

- Puzzle Class
  The puzzle class will implement functions specified below.

puzzle
-title: string
-order: int
-*initialState: unsigned int
-*goalState: unsigned int
-**states: unsigned int
-statesSize=80000000: static const int
-ORDER_MIN=3: static const int
-ORDER_MAX=25: static const int
+puzzle(int=0, string="Puzzle")
+~puzzle()
+setInitialState(unsigned int []): void
+setGoalState(unsigned int []): void
+printPuzzle(unsigned int []) const: void
+printPuzzle() const: void
+readPuzzle(string): bool
+findSolution(): bool
+manhattan(unsigned int []) const: int
+setTitle(string): void
+getTitle() const: string
-isValid(unsigned int []) const: bool
-isGoal(unsigned int []) const: bool
-isRepeat(unsigned int []) const: bool
-printSolution(int, int) const: void

Function Descriptions

- The *puzzle()* constructor should initialize the puzzle object variables as appropriate. If the order parameter is not zero, it must be checked to ensure it is between ORDER_MIN and ORDER_MAX and, if not, and error message should be displayed and the ORDER_MIN value should be used.
- The *~puzzle()* destructor should delete the dynamically allocated data.
- The *setInitialState()* function should verify that the passed array is valid by calling the *isValid()* Function. If valid, it should create and populate the initial state array.
- The *setGoalState()* function should verify that the passed array is valid by calling the *isValid()* Function. If valid, it should create and populate the goal state array.
- The *printPuzzle()* function should display the formatted puzzle of the passed puzzle state array. Refer to the sample output for formatting examples. The function should use the *isValid()* function to ensure that the passed state array is valid.
- The *printPuzzle()* function should, if it exists, print the initial state by using the *printPuzzle()* function called with the initial state.
- The *isValid()* function should verify that the passed array is valid by ensuring that the initial $\boldsymbol{order}^2$ entries are between 0 and $\boldsymbol{order}^2$-1 with each appearing only once.
- The *isRepeat()* function should compare the passed state to all of its parents (via the parent index pointer in the state arrays) and return true if a copy was found and return false otherwise. The function should check only the appropriate tile positions (i.e., 0-8 for the 8-puzzle with order 3).
- The *isGoal()* function should compare the passed state to the goal state and return true if they match and false otherwise. The function should check only the tile positions (i.e., 0-8 for the 8-puzzle with order 3).
- The *readPuzzle()* should read a puzzle from the passed file name. If the file can not be opened or there is an error in the file contents, or the contained puzzle is not valid, the function should return false. The function should use the *isValid()* function to ensure that the passed state array is valid. If the puzzle is valid, the contents should be placed in the initialState (after it is created).
- The *findSolution()* function should find a solution to the current puzzle (initial state to goal state). This includes ensuring that the order, initial state, and goal state array have been set and are valid. Refer to the Puzzle Algorithm section for additional information.
- The *manhattan()* function should compute the sum of the Manhattan Distance[3] between each tile in passed state and the goal state (excluding the blank tile). The distance between two points, (x_0, y_0) and (x_1, y_1), can be computed as follows:

$$\boldsymbol{distance} = \sum_{i=1}^{8} \left(|x_{i1} - x_{i0}| + |y_{i1} - y_{i0}| \right)$$

- The *printSolution()* function should recursively print the formatted solution using the parent states array index pointers and the *printPuzzle()* function.
- The *setTitle()* function should set the current title to the passed value.
- The *getTitle()* function should return the current puzzle title.

You may add additional private function as necessary.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 302 for additional information. ***Note, points will be deducted for especially poor style or inefficient coding.***

3 For more information, refer to: http://en.wikipedia.org/wiki/Taxicab_geometry

Puzzle Algorithm

The 8-puzzle is a small board game for a single player; it consists of 8 square tiles numbered 1 through 8 and one blank space on a 3 x 3 board. Moves of the puzzle are made by sliding an adjacent tile into the position occupied by the blank space, which has the effect of exchanging the positions of the tile and blank space. Only tiles that are horizontally or vertically adjacent (not diagonally adjacent) may be moved into the blank space. An initial configuration or state and a goal state must be provided.

Data Structures

A one-dimensional array will be used to represent the puzzle. Each tile is represented by its number using a 0 for the blank. For example, given the following initial and the goal states:

```
5  4  3          1  2  3
1  8  _          4  5  6
6  7  2          7  8  _
```

The initial representation would be: 5 4 3 1 8 0 6 7 2
The goal representation would be: 1 2 3 4 5 6 7 8 0

An additional two entries are needed for the parent state index and the move which are used to display the final solution. The blank tile may slide in one of four directions; up, down, left, or right. A valid move must be on the board. For example, given the initial state above, the blank tile can not be moved right. An array of pointers, states array, will be used to track the initial state and all the intermediate states generated. A priority queue will be used to track and select the intermediate states.

Heuristic Function

A heuristic function, ***h*(*x*)**, is used to help select the best path forward when selecting paths to try. The priority for the priority queue is set as ***p(currState) = h(currState) + level*** where level is the current level for that state in the tree. For example, the Manhattan priority of the initial state, relative to the goal state shown below is 10.

```
initial      goal
8 1 3        1 2 3          1 2 3 4 5 6 7 8
4 _ 2        4 5 6          ---------------
7 6 5        7 8 _          1 2 0 0 2 2 0 3 = 10
```

Main Loop

The main loop will perform the following operations

```
insert initial state into priority queue
while priority queue not empty and the array of states is not full
    deleteMin an index of a node from the priority queue
    if it is not the goal
        apply the four operations: move up, move down, move left, move right (in that
        order) for each valid operation
            if applicable, construct the new state
                set the parent point to the index of the parent node
                set the move direction
            if repeated state
                delete the new (duplicate) state
            if not repeated state
                store the address of the new state in the array of states
                compute the priority, p(state) = h(state) + level
                insert the new state index and its priority in the priority queue
```

If a solution is found, the *printSoluton()* function should recursively print the solution (see example output). The solution can be tracked to the initial state based on the parent index pointers.

Constructing a New State

If the move is valid, the new state should be constructed. Specifically, the a new state array should be created and the parent tiles copied into the new state array. The applicable move should be performed and the parent index and the move should be set into the last two array elements.

Checking for Repeated States

When employing the algorithm described above method you will discover that the same board configuration is repeated many times. A simple strategy is to avoid revisiting the board position that led you directly to the current board position. The boards below lead to a repetition that must be avoided. For example, given the initial state shown, the blank could be slide to the right. From that state, if the blank is slide back left, it would be a repeated state leading to a cycle.

```
_ 1 3      1 _ 3      _ 1 3
4 2 5      4 2 5      4 2 5
7 8 6      7 8 6      7 8 6
```

The repeat check consists of comparing the board of the current node with the board of each node along the path from the current node to the root. The *isRepeat()* function should perform this check.

Checking for a Goal State

The tiles from the current state should be compared to the tiles from the goal state by the *isGoal()* function. The move and parent nodes will not be the same. The algorithm includes checking for a goal state immediately after a node is taken out of the priority queue. A useful optimization is to check for the goal state before a node is inserted in the priority queue.

Example Execution:

Below is an example output for the test script and a program execution for the main.

```
ed-vm%
ed-vm% ./pqTest
****************************************************************
CS 302 - Assignment #9
Priority Queue Tester


================================================================
Test Set #0
----------------------------------------
PQ Heap (level order):
google  1

amazon  2
newegg  3

apple  4
dell  5
oracle  6
cisco  7

jupiter  8
belkin  9
ebay  10
```

```
----------------------------------------
PQ Heap Size: 10
Priority Order:
google  1
amazon  2
newegg  3
apple  4
dell  5
oracle  6
cisco  7
jupiter  8
belkin  9
ebay  10

================================================================
Test Set #1
----------------------------------------
PQ Heap (level order):
g  1

x  1
y  1

o  2
n  5
p  7
d  9

a  3
c  4
j  6
b  8
e  11
f  12
h  13
k  14

m  15

----------------------------------------
PQ Heap Size: 16
Priority Order:
g  1
x  1
y  1
o  2
a  3
c  4
n  5
j  6
p  7
b  8
d  9
e  11
f  12
h  13
k  14
m  15

================================================================
Test Set #2
----------------------------------------
PQ Heap (level order):
0x55e87fe1cf90  10

0x55e87fe1cfc0  11
0x55e87fe1cff0  12

0x55e87fe1d020  13
```

```
0x55e87fe1d050  14
0x55e87fe1d080  15
0x55e87fe1d0b0  16

0x55e87fe1d0e0  17
0x55e87fe1d110  18
0x55e87fe1d140  19

----------------------------------------
PQ Heap Size: 10
Priority Order:
0x55e87fe1cf90  10
0x55e87fe1cfc0  11
0x55e87fe1cff0  12
0x55e87fe1d020  13
0x55e87fe1d050  14
0x55e87fe1d080  15
0x55e87fe1d0b0  16
0x55e87fe1d0e0  17
0x55e87fe1d110  18
0x55e87fe1d140  19

======================================================================
Test Set #3
Large test successful.

======================================================================
Test Set #4
Email (1):  leo@MorbimetusVivamus.com
Email (2):  non.cursus@sapien.edu
Email (3):  ut.ipsum.ac@et.org
Email (4):  nec.ante@eteuismodet.edu
Email (5):  netus@nonummy.com
Email (6):  sit.amet.risus@tacitisociosquad.net
Email (7):  nascetur.ridiculus.mus@DonecestNunc.com
Email (8):  sed.dolor.Fusce@ornaretortorat.org
Email (9):  id.risus@non.co.uk
Email (10):  urna.justo@lectus.ca

**********************************************************************
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%
ed-vm% ./solver data/p1.txt
----------------------------------------------------------------------
CS 302 - Assignment #9

Puzzle: simple test (4 steps)

Solution in 4 steps.

Initial:
 ____ ____ ____
|    |    |    |
|    |  1 |  3 |
|____|____|____|
|    |    |    |
|  4 |  2 |  5 |
|____|____|____|
|    |    |    |
|  7 |  8 |  6 |
|____|____|____|     move: initial



 ____ ____ ____
|    |    |    |
|  1 |    |  3 |
|____|____|____|
```

```
 ____ ____ ____
|    |    |    |
|  4 |  2 |  5 |
|____|____|____|
|    |    |    |
|  7 |  8 |  6 |
|____|____|____|     move: right


 ____ ____ ____
|    |    |    |
|  1 |  2 |  3 |
|____|____|____|
|    |    |    |
|  4 |    |  5 |
|____|____|____|
|    |    |    |
|  7 |  8 |  6 |
|____|____|____|     move: down


 ____ ____ ____
|    |    |    |
|  1 |  2 |  3 |
|____|____|____|
|    |    |    |
|  4 |  5 |    |
|____|____|____|
|    |    |    |
|  7 |  8 |  6 |
|____|____|____|     move: right


 ____ ____ ____
|    |    |    |
|  1 |  2 |  3 |
|____|____|____|
|    |    |    |
|  4 |  5 |  6 |
|____|____|____|
|    |    |    |
|  7 |  8 |    |
|____|____|____|     move: down
```

****
Game Over, thank you for playing.
ed-vm%
ed-vm%
ed-vm%