CSCI 104 - Spring 2016 Data Structures and Object Oriented Design

Data Structures and Object Oriented Design

- Due: Tuesday, April 5 11:59 PM
- Directory name for this homework (case sensitive): `hw6`
  - This directory needs its own `README.md` file
  - You should provide/commit a Qt `.pro` file. We will then just run `qmake` and `make` to compile your `search` program.
- Start early!

Problem 1 (review material)

Carefully review Heaps, Exceptions, and A*.

Problem 2 (Heaps and Heap Operations, 15%)

As some of you might know, and the rest might enjoy to learn, Dr. Shindler has a favorite min-heap (yes, really): the word "chemistry." If you interpret the `string` as an array of `char`s, with `s[0] = 'c'`, `s[1] = 'h'` and so on, with alphabetical order as a comparator, you can verify that this is, in fact, a min-heap. Aaron doesn't have a favorite min-heap, and in fact thinks having one is silly, despite the fact that his name is one.

For each of the following words:

1. indicate if it is or is not a min-heap by drawing the tree representation of the heap.
2. if it is a min-heap, perform the remove-min operation and show the array that results from performing this operation.
3. if it is NOT a min-heap, insert the characters in the order they appear in the word into an initially empty min-heap, and show the resulting array.

Solve for the following words:

1. android
2. bedpost
3. biology
4. dentist
5. monsoon

6. tropical

You can choose to draw this by hand and scan it (and submit as a JPG or PDF), or to use some graphics software (and produce a JPG or PDF), or draw it using ASCII art (this may be a lot of work). As always, show your work.

Problem 3 (Create a d-ary Heap, 30%)

Build your own templated d-ary `MinHeap` class with the interface given below. Put your entire implementation in `heap.h` (because the class is templated). You learned in class how to build a binary `MinHeap`, where each node had 2 children. For a d-ary `MinHeap`, each node will have d children.

```
template <typename T>
class MinHeap {
   public:
     MinHeap (int d);
     /* Constructor that builds a d-ary Min Heap
        This should work for any d >= 2,
        but doesn't have to do anything for smaller d.*/

     ~MinHeap ();

     void add (T item, int priority);
       /* adds the item to the heap, with the given priority. */

     const T & peek () const;
       /* returns the element with smallest priority. */

     void remove ();
       /* removes the element with smallest priority. */

     bool isEmpty ();
       /* returns true iff there are no elements on the heap. */

 private:
    // whatever you need to naturally store things.
    // You may also add helper functions here.
};
```

In order to build it, you may use internally the vector container (you are not required to do so). You should of course not use the STL `priority_queue` class or `make_heap`, `push`, `pop` algorithms.

In order to guide you to the right solution, think first about the following questions. We strongly recommend that you start your array indexing at 0 (that will make the following calculations easier). In order to figure out the answers, we suggest that you create some examples and find a pattern.

1. If you put a complete d-ary tree in an array, what is the index of the parent of the node at position i?
2. In the same scenario as above, what are the indices of the children of the node at position i?
3. What changes in the heap functions you learned in class when you move to d-ary arrays?

Problem 4 (A-MAZE-ing Problem Solver, 55%)

For this portion of the assignment, you will be using various search techniques to solve a maze. You can find the (very extensive) skeleton code in `homework-resources`. Most of the code you write will be in `mazesolver.cpp` although you're free to declare other functions you may need in `mazesolver.h` as private functions.

I encourage you to read the following before you set to coding, so you know what is provided:

- `maze.h` -- this is the interface you have for the Maze. Note that you cannot (and should not) create one yourself, although you may call public functions on a Maze object during your search.
- `visitedtracker.h` -- you will be using this interface to track which spaces you have visited when exploring a maze. You will need to use this, as you will need to report the list of what you have (and have not) visited to the interface when your search is done. Fortunately for you, this functionality is provided to you.
- `mazesolver.cpp` -- in particular, look at `solveByBFS()`. Note how it uses the `maze` and `VisitedTracker` items to implement BFS. You may also want to compare that function's implementation to the pseudo-code provided to you in lecture. Note that the graph is **implicit** (that is, the nodes and edges are not explicit objects, but are determined implicitly from a more concise representation). A few good questions to ask yourself here is what are the implicit nodes/vertices? What are the implicit edges? Why don't we build the graph explicitly?

You will need to implement the following functions in `mazesolver.cpp`. Note that they're immediately tied in with the UI and you can observe them immediately. If you need to do debug output, use `std::cerr` -- it will still appear in the console (similarly to how your `PRINT` statements in assignment five did).

- Depth-First Search. We are requiring that your function be implemented recursively. There is iterative code provided that you can view and call in the UI, if you want to see how they compare.
- A* with a heuristic estimate of `0` for all nodes. How do you expect the behavior of this to compare to BFS?
- A* with a heuristic estimate of the euclidean distance to the goal location. Why is this a valid heuristic

for A*, even though you would, in most cases, need to be the Kool-Aid mascot in order to walk a straight line to the goal?

- A* with a heuristic estimate of the "Manhattan distance" to the goal location. This means the distance you'd have to walk if there were no walls, but you can only move UP DOWN LEFT or RIGHT.

As a hint, the various A* implementations are **much easier** if you write them once and abstract the heuristic function, and then call your one function differently from the three required portions. If you understand that last sentence, it means you have to write two functions for the above four requirements. If you don't understand that hint, estimate how much copy/paste of code you'd have to do if you finished the first A* requirement and wanted to implement the other two and recall that, when you want to copy/paste, there's a better way.

You should use your MinHeap from part 2 to implement A*. If you did not complete part 2, you may still be able to do this section. You may use the C++ STL's class for a priority queue, although doing so will cost you `5` points in this section. In fact, you might choose to do this part first, using the STL, and once it works, replace it with your own heap.

Note that your MinHeap does not have an update function. What should you do if you found a better path to a node than what is currently stored in the heap?

You can simply add the node again to the heap with the new priority. Note that the node will now appear *twice* in the heap. When you remove a node from the heap, you should verify that you haven't already explored this node. That is, the first time you remove a node from the heap you should explore it, and you should ignore that node from then on if you ever remove it from the heap again. This increases the number of items in your heap from `n` to `m`, but only doubles the worst-case runtime of your functions, because `log m < 2 log n`.

You do not need to create any new files for this problem.

Commit then Re-clone your Repository

Be sure to add, commit, and push your code in your `hw6` directory to your `hw_usc-username` repository. Now double-check what you've committed, by following the directions below (failure to do so may result in point deductions):

1. Go to your home directory: `$ cd ~`
2. Create a `verify` directory: `$ mkdir verify`
3. Go into that directory: `$ cd verify`
4. Clone your hw_username repo: `$ git clone git@github.com:usc-csci104-spring2016/hw_usc-username.git`
5. Go into your hw6 folder `$ cd hw_username/hw6`

6. Recompile and rerun your programs and tests to ensure that what you submitted works.