CSCI 104 - Fall 2015 Data Structures and Object Oriented Design

Homework 8

- Due: Friday, December 4th, 11:59pm (PST)
- Directory name for this homework (case sensitive): `hw8`
  - This directory should be in your `hw_username` repository
  - This directory needs its own `README.md` file
  - You should provide a `Makefile` to compile your code.
  - This homework counts as a project in terms of grade calculation, but it is not an extension of your google project.

Problem 1 (Build a hashtable, 20%)

Build a `hashtable` class with the interface given below. You should keep track of the load factor (n/m, where n is the number of elements and m is the number of indices): if inserting an item would cause the load factor to be a value greater than 1, you should increase the size of your hashtable to 2m+1. You may include extra functions if they are appropriate (such as operator[]). You are specifically making a hashtable with `string` keys and `int` values.

```
class HashTable {
   public:
     HashTable ();
     /* Constructor that builds a hash table with 31 indices. */

     ~HashTable ();

     void add (std::pair<std::string, int> new_item);
       /* adds new_item to the hashtable.
          Conflicts should be resolved via chaining.
          You may use the STL list to accomplish this.
          Throws an exception if the key is already
          in the hashtable.*/

     const int& find(std::string key) const;
       /* returns the value with the associated key.
          Throws an exception if the key is not in the
          hashtable. */

     void remove (std::string key);
       /* removes the element with the associated key.
```

```
            Throws an exception if the key is not in the
            hashtable. */

      void update (std::pair<std::string, int> new_value);
        /* finds the item with the associated key,
            and updates its value accordingly.
            Throws an exception if the key is not in the
            hashtable. */

   private:
      // whatever you need to naturally store things.
      // You may also add helper functions here,
      // including your hash function and resize function.
};
```

Building a Hash Function

For the sake of simplicity, we will not construct a particularly good hash function. First translate the `string` key to an `long long`:

29n−1·pn−1+…+292·p2+29·p1+p0,

where p0 is the last letter of the string, and p(n-1) is the first letter of the string. Each letter should evaluate to an integer between 0 and 25, where a=0, b=1, ..., z=25. We are using 29 in the hash function because, in general, choosing a prime number for this task is superior.

Finally, mod the result by the size of your hashtable.

Translating the key into a `long long` is capable of handling keys of up to 12 letters. You may assume that you will never receive keys of length longer than this.

Problem 2 (Build a d-ary heap, 30%)

Build your own d-ary `MinHeap` class with the interface given below. 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.

```
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 (std:string item, int priority);
       /* adds the item to the heap, with the given priority. */

     const std:string & peek () const;
```

```
      /* returns the element with smallest priority.  If
         multiple items have the smallest priority, it returns
         the string which comes first alphabetically.
         Throws an exception if the heap is empty. */

    void remove ();
      /* removes the element with smallest priority.  If
         multiple items have the smallest priority, it removes
         the string which comes first alphabetically.
         Throws an exception if the heap is empty. */

    void update (std::string item, int priority);
      /* updates the priority for the specified element.
         You may want this function to do nothing if the new
         priority is higher than the old one.
         Throws an exception if the item is not in the heap. */

    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.

You will need to make use of the update function in your heap, so that you can change the priority of a string. To do this in O(log n) time, you need to know what index the word is stored at in the heap. You will make use of your hashtable implementation to do this. You should store a hashtable as a data member of your heap, which maps a string to the index it is currently located at in the heap.

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 3 (A* Puzzle Solver, 50%)

The word game "Doublet" was invented by Lewis Carroll, and is a word transformation puzzle. Two words of identical length are given. The objective is to transform the first word into the second word by forming successive words of the same length, changing only one letter at a time. Here is an example from HEAD to TAIL:

```
HEAD
```

```
HEAL
TEAL
TELL
TALL
TAIL
```

The challenge is to do the transformation in the least number of words.

Your program should be called `doublet`, which takes three command line parameters. The first indicates the starting word, the second indicates the ending word, and the third is a file which contains a list of valid words. So you might run the program as follows:

```
./doublet head tail words.txt
```

Everything should be case-insensitive, so there is no difference if the starting word is `HEad` or `heAD`.

Your program should output the following:

```
word1 word2 ... wordM
expansions = X
```

Where word1 is the starting word, wordM is the ending word, and X is the number of expansions (defined below) it took to find this solution.

The word file will be formatted as follows:

```
7
head
heAl
hem
Tail
tell
taLL
teal
```

The first row contains a number `n`, indicating the number of words in the file. There will be `n` more rows, each with a single word, possibly followed by some whitespace. There may be blank lines after the words. You may assume the file is formatted properly. We may give your program very large word files (around 1 million words).

You will implement the A* search algorithm to quickly find the shortest transformation. You can think of each word in the word file as a node, and there is an edge between two words of the same length if they differ by exactly one letter.

Your A* search algorithm **must** use your heap implementation from problem 2 to store the nodes and to figure out which node to explore next.

Review the A* algorithm

Recall that A* makes the move with smallest f-value. f = g + h where g = distance (number of moves made) from the start state while h is a score produced by a heuristic evaluation of the move. Please take some time to review the algorithm presented in class (slides, notes, etc.). We will use the following heuristic:

Incorrect Letters: counts the number of letters in the current word which do not match the letter in the same position in the ending word. So if you are currently at `DATA` and your final word is `SALT`, then your heuristic will evaluate to `3`.

Implementation Details

To add consistency to your solution, you should break ties in the following manner:

1. Always make the move with smallest f-value.
2. If multiple words have the smallest f-value, choose the one with the smallest h-value.
3. If multiple words have the smallest f and h-value, choose the string which comes first alphabetically.

To accomplish item `2`, you will want to calculate the priority as `f*(n+1)+h`, where `n` is the length of the word you are transforming. Since `h` can never be larger than `n`, this properly chooses the smallest f-value while breaking ties according to h-value.

To track how well your algorithm is performing, you should keep track of the number of **expansions**, that is, the number of words your algorithm considers. Every time you remove the min-value word from your MinHeap, you are considering that word, and you should increment the number of expansions. The starting word should increment the number of expansions (from 0 to 1), but the ending word should not.