CS 104 (Fall 2013) — Assignment 10
Due: 12/03/2013, 11:59am

BitBucket directory name for this homework (case sensitive): HW10

(1) Review Chapters 19, 18.4 and the additional handout on hashing.

(2) Implement a template hash table. The public parts of your node and hash table signatures should be the following (you are of course welcome to add data fields and private methods):

```
template <class KeyType, class ValueType>
struct Node {
     KeyType key;
     ValueType value;
};

template <class KeyType, class ValueType>
class Hashtable {
  public:
     add (const KeyType & key, const ValueType & value);
     remove (const KeyType & key);
     ValueType & get (const KeyType & key) const;

     Hashtable (int initialSize);
};
```

You should implement the Hashtable either "as-A" (recommended) or "has-A" (also accepted) `List<Node>`. More specifically, the `List` can be either just a regular array, or your own `List` implementation from earlier homeworks, or a C++ STL `vector` or similar.

Your Hashtable should handle collisions using chaining, i.e., each element of your array should be either a pointer to the first element of a linked list, or again some kind of a `List` or `vector`. For full credit, you should keep track of your hash table's load factor, and if it goes beyond 0.5, you should double the size of your table and rehash all elements. But we recommend implementing all other functionality first, before worrying about this one.

You should assume that the class `KeyType` contains a method `int hash() const`, which gives you a hash value for the object.

(3) In order to test your Hashtable implementation from the previous problem, write a simple interface in which a user can enter new student/GPA pairs, delete a student from the hash table, and look up a student's GPA based on the student's name. You are welcome to reuse and adapt your code from Homework 9 here. In order to make this work, you should define a new

```
class StudentName : public string {
  public:
    int hash ();
};
```

and implement your own hash function. You are encouraged to read through the textbook and look at Google and Wikipedia to learn about some hash functions to choose from.

(4) Thanksgiving is coming up! Imagine that you are throwing a potluck dinner for your $n \leq 50000$ best friends. Each friend brings a dish to the Thanksgiving potluck. Since people aren't coordinating, you may end up with duplicates of dishes, and being the charitable person that you are, you want to donate some of the extra unneeded food to those more in need than you and your friends. You decide to donate one each of each food that is *the most duplicated*. In other words, you are looking for those dishes that maximize the number of times that they occur among what your friends brought, and donate one of each of them.[1] Write a program that figures out what food you are donating.

The input is stored in a file of strings. The first line contains the number n of guests, and the next n lines n strings, the names of the foods. To keep things simple, let's assume that "turkey" and "Turkey" are two different foods.

This problem may look somewhat familiar to you. This time, your solution has to run in $O(n)$ (typically). Slower solutions will not be worth any credit. Again, include with your solution a high-level description of your solution approach. As a hint, using your solution to other problems on this set may be helpful.

Sample Input:

```
8
turkey
yams
yams
gravy
turkey
stuffing
yams
Turkey
```

Sample Output:

```
yams
```

(5) Now suppose that at your Thanksgiving potluck, you've carved the turkey into $n \leq 50000$ pieces, of different sizes, which are sitting on a nice platter. You've also already prepared $m \leq 50000$ plates with side dishes; each plate contains a nice mix of yams, stuffing, potatoes, and cranberries.

For each plate, you know the calories (some integer between 0 and $(2^{31} - 1)$; let's just say that you and your friends like to gorge yourselves on Thanksgiving); similarly, for each piece of turkey, you know its calories (same range as for the side dishes). Now, some friends of yours have very precise diet requirements — they will need to achieve a precise number of calories in their dinner. Your goal is to find whether there is a combination of a plate of sides and a piece of turkey that will exactly give them their desired calories. Since this is getting tough, you want to write a program to solve this problem.

The input to your program is a file. The first line of the file contains integers n, m and C (the number of turkey pieces, the number of plates of side dishes, and the calorie target). The next n lines each contain the calories for one turkey piece; let's call those $t_1, t_2, \ldots, t_n$. They are given in no particular order. The next m lines contain, in no particular order, the number of calories for the m plates of side dishes; let's call those $s_1, s_2, \ldots, s_m$. Your goal is to find out whether there are indices i, j such that $t_i + s_j = C$.

This problem, too, may look somewhat familiar. Again, now, your solution has to run in $O(n)$ (typically). Slower solutions will not be worth any credit. Include with your solution a high-level description of your solution approach. Using your solution to other problems on this set may again be helpful.

Sample Input:

[1] When you have exactly one copy of each of your food items, instead of giving away all your food, you'll keep all of it. So you'll only ever donate if there are duplicates.

```
// The comments in this file are only for illustration.
// You don't need to parse comments
4 3 8   // 4 pieces of turkey, 3 side plates, 8 calorie target
4       // calories for turkey pieces start here
7
2
4
9       // calories for side dish plates start here
6
3
```

Sample Output:

```
8 = 2 + 6
```

Note that in the given example, $8 = 4+4$ is not a solution, as you cannot combine two pieces of turkey to form a Thanksgiving dinner.

(6) You've been implementing Facebook all semester long, so why not implement Google, too? Well, not quite all of Google, but one of the key pieces. As we talked about in class, the way Google answers queries is basically by pre-computing a dictionary that maps search queries to search results. Here, we will focus just on single-word queries, and use a very simple format for "web pages" and search results.

Your input will be a file in the following format. The first line will contain an integer $n \leq 200000$, which is the number of lines/pages in the file. This is followed by n lines. Think of each line as a (small) web page. Each line consists of up to 30 words, each word consists of up to 8 lowercase characters, and words are separated by one or more white spaces.

Your program should read such a file and first build from it a Dictionary, using either your own 2-3 tree implementation from Homework 9 or your own Hashtable implementation from Homework 10. This pre-computation is allowed to take a little while. Subsequently, your program should allow a user to pose single-word queries. For each query, it should (blazingly fast) print all lines from the input file in which this word occurs.

We will soon post a couple of sample input files for you to test your program on, so that you don't need to generate your own; we will also use these (and perhaps additional files in the same format) to grade your solution. Of course, if you want, you can also produce your own input files in addition.

(7) Chocolate Problem: Same rules as in the past; 2 chocolate bars.

The lookup table for queries is only half the battle for a search engine. If you search for "turkey," Google not only has quick access to all pages that contain the word, it also is able to rank them pretty well, so that the "best" pages appear at the top of the search results. Many many clever techniques go into it, but at the very heart (at least for Google's initial success compared to other search engines in the mid 1990s) is a link analysis technique called PageRank.

The rough idea is to interpret the links between web pages (directed edges in the web graph) as endorsements. If page A links to page B, then it is not unreasonable to assume that the author of page A thought that page B had some useful information, so if page B contains the word "turkey," perhaps it should be displayed higher than other pages which also contain the word, but do not receive any links from other pages.

This would suggest sorting the pages containing "turkey" by the number of incoming edges. But a refinement would be to notice that not all incoming links are equal. If page C receives only one link, but that one link comes from a really important page (which itself has many incoming links), perhaps that link should count for a lot. Another thing to notice is that if page A links to many many pages,

then perhaps the fact that it endorses B does not mean as much as an endorsement from D, which has few outgoing links.

An eventual solution to this is the following: each page i has an *authority* (or PageRank) a_i. Then, each page i confers authority $a_i/\text{outdegree}(i)$ to each page it points to, meaning that the authorities should satisfy $a_i = \sum_{j \to i} a_j/\text{outdegree}(j)$. This almost works, but for technical reasons (feel free to talk to me why), one needs a little bit of a fix, and instead writes: $a_i = \frac{1}{7n} + \frac{6}{7} \cdot \sum_{j \to i} a_j/\text{outdegree}(j)$. Notice that this is a system of linear equations for the a_i, so one can solve it and obtain nice a_i values. (You need one more equation to make the solution unique, so let's either say that $a_1 = 1$, or that $\sum_i a_i = 1$.) To learn more about PageRank vectors, and see a nice visualization, check out `https://googledrive.com/host/0B2GQktu-wcTiaWw5OFVqT1k3bDA/`

Now we have a solution: display all pages i containing the query term, and display them in order of decreasing a_i values. Notice that this raises two different approaches:

(a) Pre-compute the a_i values for all pages based on the entire web graph, and sort pages once and for all. Then, just output all relevant pages (containing the query term) sorted by these pre-computed a_i values.

(b) When the query comes in, focus attention only on pages containing the query. That gives you a smaller graph. Compute the a_i values for this smaller graph at query time, and sort by them.

The second approach probably gives the better results, but it's much slower. You should feel free to implement either (or both) of these approaches.

For concreteness of format, let's describe hyperlinks in your input file as follows. If a word is a hyperlink, then instead of `turkey`, your text file contains `[turkey,17]` to denote that the keyword `turkey` contains a hyperlink from the current page to page 17.

Write an extension to the previous problem in which links are also parsed, and the final output is sorted by decreasing PageRank values.