CPSC 221: Programming Project 2

Due: Friday November 2, 2012 at 9pm via `handin`.

Out: October 15, 2012 Last Updated: October 15, 2012

FreqQ

A *frequency queue*, or *FreqQ* for short, is an abstract data type that supports four operations:

- **add**(string *word*): Increase by one the frequency of *word* in the FreqQ. If *word* is not in the FreqQ, add it to the FreqQ with frequency one.
- **sub**(string *word*): Decrease by one the frequency of *word* in the FreqQ. If the frequency of *word* becomes zero, remove it from the FreqQ.
- string **topWord**(): Return a word with the highest frequency in the FreqQ.
- int **topFreq**(): Return the frequency of the highest frequency word in the FreqQ.

A FreqQ combines operations of a priority queue (**topWord** and **topFreq**) with operations of a dictionary (both **add** and **sub** require finding *word*).

For this assignment, you should implement a FreqQ using a **heap** (a **max** heap) and a **hash table** that uses open addressing with linear probing.

The heap and the hash table work together to provide the FreqQ operations. For example, **add**(*word*) looks up *word* in the hash table. Here is my hash function:

```
int FreqQ::hash( string word ) {
  int h = word[0];
  for( unsigned int i=1; i<word.length(); ++i )
    h = (31 * h + word[i]) & (HASHTAB_SIZE - 1);
  return h;
}
```

If *word* is found in the hash table, the frequency of *word* is incremented and **swapUp** is called on its position in the heap to restore the heap's ordering property. But how do we know its position in the heap? This is the tricky part of the implementation. The hash table entry for *word* contains the position (or index) of *word* in the heap, which is where *word*'s frequency is kept. The **swap** procedure used by **swapUp** and **swapDown** must update the heap indices stored in the hash table entry. Here is my **swap** procedure:

```
void FreqQ::swap( int i, int j ) {
  hashTab[heap[i].hashIndex].heapIndex = j;
  hashTab[heap[j].hashIndex].heapIndex = i;
  HeapEntry tmp = heap[i];
  heap[i] = heap[j];
  heap[j] = tmp;
}
```

Notice that I also keep the hash table index in my heap entry so I can update the appropriate hash table entry.

If *word* is not in the FreqQ, it needs to be inserted into the hash table and the heap. Since it's frequency is as small as any frequency in the heap, it can simply be added to the end of the array representing the heap.

The operation **sub**(*word*) also looks up *word* in the hash table to find its index, call it j, in the heap. It decreases the frequency of *word* and calls **swapDown**(j) to restore the heap's ordering property. If the frequency becomes zero, the word is removed from the heap (by swapping the last heap entry into position j, calling **swapUp**(j), and decreasing the heap's size) and removed from the hash table (by replacing the hash table entry with a **tombstone**).

You may assume that no more than 32768 words will be in your FreqQ at any one time.

My FreqQ class has the following methods: hash, swap, swapDown, swapUp, hashFind, hashInsert, add, sub, topWord, topFreq.

Testing your implementation

To test your implementation, I have provided a file `topwords.cpp` that will print the most frequent word in every contiguous subsequence of k words from a text file. For example, given $k = 3$ and the text file:

```
a a a b b b c c c d d d
```

I get the output:

```
1-3 a(3)
2-4 a(2)
3-5 b(2)
4-6 b(3)
5-7 b(2)
6-8 c(2)
7-9 c(3)
8-10 c(2)
9-11 d(2)
10-12 d(3)
```

I encourage you to test your implementation in other ways as well.

Deliverables (to be submitted online, not on paper)

- Your source code (.cpp and .h files).
- A `Makefile` so that typing `make` in your handin directory on an undergrad Unix server will produce an executable called `topwords`.
- A `README` file containing:
    1. Your name or, if a team submission, both your names.
    2. Approximately how long the project took you to complete.

3. If you worked in a team, a breakdown of the work.
4. Acknowledgment of any assistance you received from anyone but your team members, the 221 staff, or the 221 textbooks, but please cite code quoted or adapted directly from the texts (per the course's Collaboration policy).
5. A list of the files in your submission with a brief description of each file.
6. Any special instructions for the marker.

- DO NOT HAND IN: .o files, executables, core dumps, irrelevant stuff.

How to `handin` this assignment

1. Create a directory called `~/cs221/proj2` (i.e., create directory `cs221` in your home directory, and then create a subdirectory within `cs221` called `proj2`).

2. Move or copy all of the files that you wish to hand in, to the `proj2` directory that you created in Step 1.

3. Before the deadline, hand in your directory electronically, as follows: `handin cs221 proj2`

   Note that you will receive a set of confirmation messages. If you don't get any kind of an acknowledgment, then something went wrong. Please re-read the instructions and try again.

4. You can overwrite an earlier submission by including the `-o` flag, and re-submitting, as follows: `handin -o cs221 proj2`

   You can hand in your files electronically as many times as you want, up to the deadline.

5. Additional instructions about `handin`, if you need them, are listed in the man pages (type: `man handin`). At any time, you can see what files you have already handed in (and their sizes) by typing the command: `handin -c cs221 proj2`

   If your files have zero bytes, then something went wrong and you should run the original `handin` command again.