Data Structures (2928) -- Fall 2012

Project 5

Hash Table Performance

Due in class Dec. 5, 2012 9:05 am

Each student should submit a tar.gz file containing all sourcecode including Makefile, Hash.cpp, Hash.h, BSTree.cpp, BSTree.h, (or LList.cpp & LList.h), spellcheck.cpp, any scripts you have written, and an analysis document (described below) to Blackboard. Hand in printed copies of all files.

This project will tie together some of the data structures we’ve talked about in this course. The end goal is to create an analysis document showing the relative performance between different hash table collision techniques to implement a spell check program.

We’ll break this project down into 3 parts:

1. Hash table (Weekly Assignment due Dec 2nd, W14) Implement a hash table class (Hash.cpp, Hash.h) which allows storage and lookup of strings (no deletion). Implement 3 collision resolution techniques. First, use open addressing and linear probing, next use another open addressing scheme (Double Hashing, Quadratic probing, or any other open addressing scheme). Lastly implement separate chaining using a linked list or BST. I HIGHLY recommend you reuse your BSTree class or LList class from earlier. For 5% off your project grade I can provide working implementations of BSTree or LList if you ask by email.

You may use whatever hash function you like, just be sure you mention which in your analysis document.

Rather than implement a dynamically sized hash table, we’ll set the size during object construction, as described below. Implement the following public functions for the Hash class:

a. Hash() – Default constructor which uses linear probing and hash table size of 100.
b. Hash(int mode, int size) – Creates a new hashtable using a specific collision mode (1 = linear, 2 = other open addressing, 3 = separate chaining) and a hash table of size size.
c. ~Hash() – Object destructor
d. bool insert(string thing) – Inserts the string thing into the hash table object. Returns true if successful, false otherwise. Can’t insert into a full table!
e. bool isFull() – Returns true if the hash table can not store any more elements.
2. Spellcheck Program – create a spellcheck.cpp program, which will output the number of correct & incorrectly spelled words for a given file. Ignore case but **not** punctuation. More specifically, it should accept 5 command line arguments

- **a.** `--d <filename>` - Dictionary file. scrabble.txt for example.
- **b.** `--f <filename>` - ASCII file to spellcheck.
- **c.** `--m <int>` - Collision mode (1, 2, or 3) to use in the hash table.
- **d.** `--s <int>` - Size hash table to use.
- **e.** `--v` – Verbose output, see below.

See the screenshot below an example of the non-verbose output. When the verbose flag is set, it needs to output more information. The exact format of the information is up to you, but minimally you’ll need to output the following:

1. hash table size
2. load factor
3. number of bytes
4. Average number of comparisons per successful isStored lookup.
5. Average number of comparisons per unsuccessful isStored lookup.

3. Analysis - You’ll be analyzing the efficiency of all 3 of your implemented collision-avoidance schemes, just like the two plots on page 704 of our text. Additionally draw separate (or over top) plots of the number of bytes needed for each scheme. The load factor from 0 to 1 is the x axis, and the average number of string comparisons is the y axis (max 20). Separate the successful searches from the unsuccessful. For the bytes used plot, the y axis is bytes or some reasonable unit of bytes.
Unlike (I suspect) the plots on page 704, the number of comparisons needed for separate chaining should include the number of comparisons in the linked list or BST structure.

To generate the data for this analysis use your spellcheck program above and output data using the verbose flag. It’s a good idea to output a single line of numbers interspersed with commas (CSV). Thus, you can use a separate program to run your spellcheck program multiple times with different command line inputs. The run.pl script provided gives an example of how this can be done. UCFilespace and most linux distros come with Perl. Use and manipulate the script as you wish, you do not need to hand it in. Then, copy and paste the results into your favorite spreadsheet program and plot. Any other data analysis workflow is fine.

You should hand in an analysis document with 3 plots (or 2 overlaid with size) showing how the number of comparisons changes as the load factor increases, or in other words as the hash table size decreases, for both successful and unsuccessful searches. Limit your y axis to 20 avg comparisons. The remaining plot should show space usage as a function of load factor.

Your completed project should contain a Makefile. When ‘make’ is run in the folder a ‘spellcheck’ executable should be made.

You’ll be graded on Hash.cpp/.h, spellcheck.cpp, Makefile, and the analysis document.

Your grade will consist of the following:

1. 15% - Operation of spellcheck.cpp, including command line parameter handling.
2. 10% - Function headers for public functions, files, classes, comments, code indentation.
3. 60% - Proper operation of Hash class functions. 6% per function above, broken down into each of 3 collision avoidance techniques.
4. 15% - Analysis document

Possible deductions:

1. -70 points – Fails to compile.

Extra Credit:

1. +5 points: Plot the time to spellcheck the document as the load factor changes as well. Do not include the time to fill the hash or analyze the data.
2. +5 points: Implement a bloom filter as the 4th mode in the hash class (http://en.wikipedia.org/wiki/Bloom_filter). Use a k value of 5 (number of hash functions). What is the lowest load factor (hash table size) which still outputs the correct number of correct and incorrect words? Plot the % correct as the load factor varies from 0 to 1. Include the bloom filter in your space usage plot.

Example Output: