Find it Fast

Hash Tables and
Collision Resolution

Abe Karplus

Science Fair

11 February 2009

Table of Contents

Abstract

Hash tables are an efficient method for data storage and access. They allow accessing data without extensive search. When two keys hash to the same location, a collision resolution algorithm is used to determine where the keys should be placed. I studied three collision resolution algorithms (CRAs): linear probing, linked list, and cellar. My hypothesis was that all three hash table CRAs would perform about equally well, and all much better than a simpler linear search method.

I measured time, memory usage, and number of string comparisons (as a more precise proxy for time) on 50 different inputs for these three hash methods and for linear search. As expected, hashing was a hundred times faster than linear search, taking 0.35–0.5 seconds to look up the approximately one million words of the Bible, instead of 45 seconds for linear search.

The cellar and linked list CRAs always performed well. Contrary to my hypothesis, linear probing sometimes performed poorly when it didn't rehash until the table was full. Rehashing when the table was only 70% full allowed linear probing to perform on par with linked list and cellar.

Importance of Fast Retrieval

People use computers to store large quantities of information and access it quickly. To understand how this is done, consider the following analogy:

You are writing a dictionary and keeping the words you find on an unsorted set of index cards. You come across a word and need to look it up. You could look through from the beginning until you find the word or get to the end. This is called *linear searching*, and it could take a while, especially if the set is big. On a computer, the equivalent of this set of index cards is an array.

What if you put the index cards in pigeonholes and had a method of just looking at the word and having it tell you in which pigeonhole to look for the index card? This method is known as a *hash table*, and the function for converting a word into a pigeonhole address is known as a *hash function*. The disadvantage of this method is that the hash function takes time to compute.

Does the reduced search time offset the time taken to compute the hash function? How much time is saved? Does it depend on the size of the input?

What is a Hash Table?

A hash table consists of three parts: a hash function, a data structure that includes an array, and a collision resolution algorithm (CRA). Linear search includes only the data structure, not the hash function or CRA.

First let's look at the hash function. This function takes as input a *key*, usually a word, and outputs a numeric value, called the *hash value* of that key. The hash function is a true function in that the same key as input will always produce the same numeric output.

How does it do this? Figure 1 shows a sample hash function—the one I used in my code. To understand how it works, you must know how characters are converted into numbers by the ASCII standard. Each character (letter, number, or punctuation) has a specific numeric value.In ASCII, uppercase is different from lowercase (A=65, Z=90, a=97, z=122). I did not want words that were identical except for punctuation being treated as different words, and so I used the 'tolower' function, which converts uppercase letters into lowercase.

In the function, we have a variable called `val` that starts at 1. For each character in the word, that character's "lowered" value is added to the previous `val`, and then the whole value multiplied by a very large prime to become the new `val`. All computation is done in 32-bit words, which means that whenever `val` gets higher than 2^{32}, the remainder when divided by 2^{32} is taken. Figure 2 has an example computation.

```
#define START 1
#define PRIME 999999137

unsigned long hash1(char* str)
{           unsigned long val;

            for(val = START; *str != 0; str++)
            {           val = (val + tolower(*str)) * PRIME;
            }
            return(val);
}
```

Figure 1. This code in the programming language C for the hash function I used.See main body for explanation.

Hash("d") = 2215665029
Hash("do") = 2008927860
Hash("dog") = 85621179

Hash("god") = 1703519611

Figure 2. Here is an example computation for "dog", and the hash value for "god" to show that the order of the letters matters.

All the keys are stored in an array. You can think of this array as a set of pigeonholes. The array gets *allocated*—memory space is set aside for it—before entering the keys. The computer knows how large the array is because it has allocated it.

The remainder when dividing the hash value of a word by the size of the array specifies which pigeonhole, the *hash location*, to put the key in, together with whatever data is associated with it—for example, in a dictionary, the definition of the word. When you get the hash value for a word, you go to that location, and check if it's already there—after all, a word can appear more than once in a document. If the word is not already there, you put it in that location.

A hash function always gives you the same value for the same word, but it can return the same value for different words. The remainder can be the same, even for different hash values. When two different keys hash to the same location we call it a collision. Let's say you have key A and it hashes to position 79. It would be put there. Then you have key B and it also hashes to position 79. Since A is already there, you need to find somewhere else for B to go. To do this, you need a collision resolution algorithm—a method for determining where colliding keys should be stored. I will be studying three: linear probing, linked list, and cellar.

It is possible to write perfect hash functions, ones in which different words always produce different hash values, but to do so, you need to know what keys will be in the data set you will be hashing, a luxury not often afforded to programmers. I did not study perfect hash functions for this report.

Hypothesis

I hypothesize that, with any collision resolution algorithm, hashing will be significantly faster than linear search and that there will be little difference between different collision resolution algorithms.

Clustering

Collisions result in clustering. There are two types of clustering: *Primary clustering*, collisions between keys with the same hash value, and *secondary clustering*, collisions between keys with different hash values. Clustering greatly reduces the search efficiency. If clusters are big enough, then the algorithm can be as slow as linear search.

Primary clustering occurs in all methods and is handled by an algorithm like linear search, though often by following pointers, rather than stepping through an array. Pointers are addresses of locations in memory. To follow a pointer means to access the location it points to. A null pointer has the value zero, which means that it does not point anywhere. A linked list consists of several objects, each one containing the key, associated data, and a pointer to the next item in the list. The last item in the list has a null pointer, as there is no next item. An empty list is just a null pointer.

Secondary clustering occurs in CRAs that use cells in the table to store collinding keys, like linear probing and cellar, but not linked list. The linear probing example shows how how secondary clustering occurs.

Collision Resolution Algorithms

A collision resolution algorithm is a method for determining where colliding keys should be stored. There are several. The three I will be studying are: linear probing, linked list, and cellar.

Linear Probing

Linear probing is a lot like linear search, but, instead of starting at the beginning, you start at the hash location of the key and do a linear search from there, checking to see
- if the key is already there, then you can return it;
- if the space is empty, then you can insert the key; or
- if there's something else there, then you go on to the next space and check again.

In Figure 3, clusters are building up, and secondary clustering occurs.

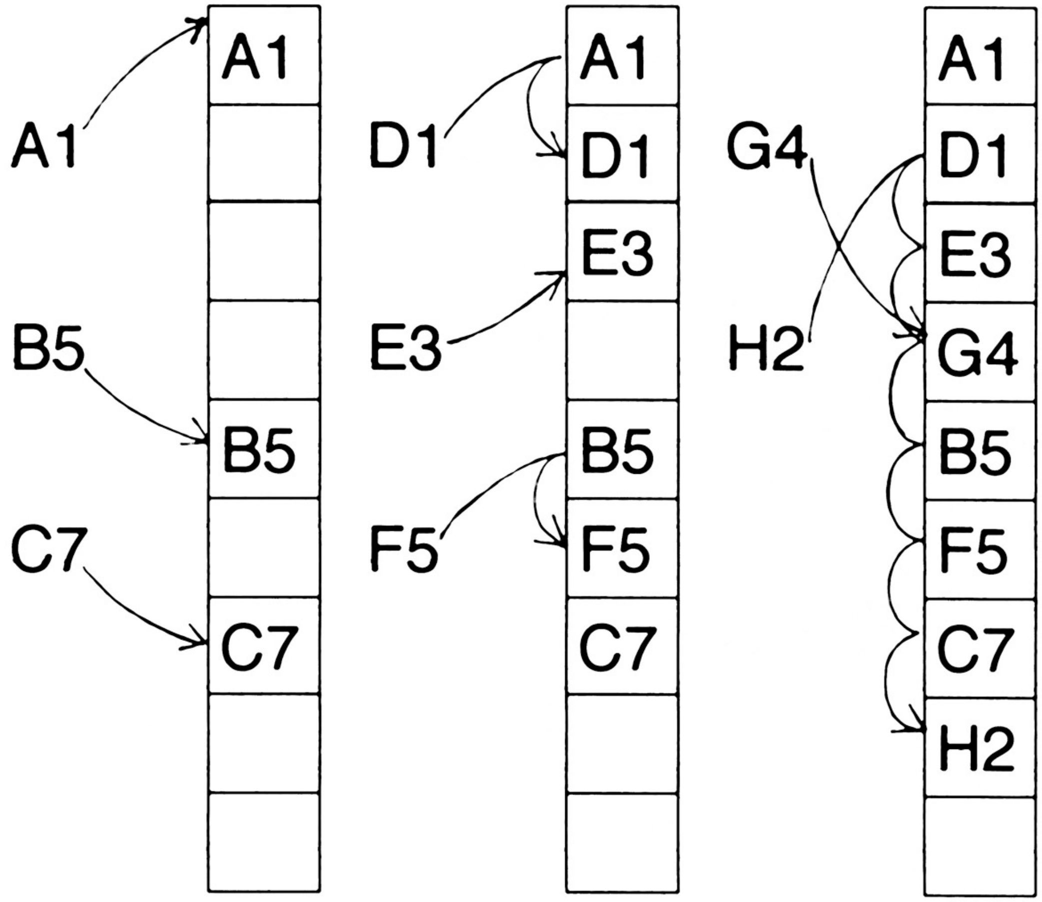

Figure 3. In the diagram, keys A1, B5, and so on have hash values of 1, 5, and so on. In the first column, A1, B5, and C7 have each been placed in their respective positions. In the second column, D1 would go in position 1, but A1 is already there, so it has to go in position 2. E3 goes in position 3, but F5 collides with B5 and has to go in position 6. In the third column, G4 goes in position 4, but when H2 attempts to enter, it can't go in positions 2 through 7 and so must go in position 8, even though it is the only key with a hash value of 2.

Linked List

The linked list method of resolving collisions places all the colliding keys for a given hash value in a sort of chain, or linked list, attached to the corresponding space in the hash table. The advantage of linked list hashing is that it completely removes *secondary clustering*—that is, collisions between keys with different hash values. *Primary clustering*, collisions between keys with the same hash value, is handled by something that amounts to a linear search, but by following pointers, rather than stepping through an array. Figure 4 shows how collisions are handled in the linked list method.

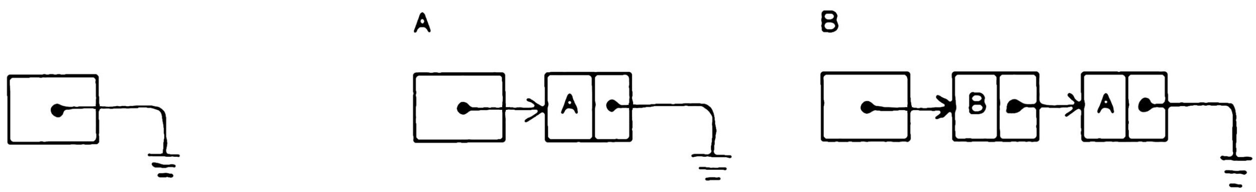

Figure 4. These pictures show one pigeonhole of a hash array. In the first picture, nothing has been hashed to that spot, so it has a null pointer. In the second one, "A" has been hashed there and added to the list. In the third one, "B" is hashed there. Note that it is added to the beginning of the list. Both "A" and "B" are added in the same way: a piece of space is allocated, the next pointer of the new thing is set to point to the beginning of the existing list, then the pigeonhole pointer is set to point at the newly allocated space.

Cellar

Like linked list, cellar uses chains of pointers for colliding keys. Like linear probing, however, the colliding results are stored in the table, not in newly allocated space. The cellar CRA has a portion of the table walled off (the *cellar*) and only used for collision resolution storage.

The advantage of cellar is that it greatly reduces secondary clustering compared to linear probing. Its advantage over linked list is that reduces the number of times the memory allocator is called.

If the hash position of a key is empty, you can put the key there. Otherwise look down the chain of pointers for the key. If it is not on the chain, put it in the lowest empty spot in the table, and set the pointer at the end of the old chain to point to it. Figure 5 shows what happens when seven keys are added to a hash table using the cellar CRA.

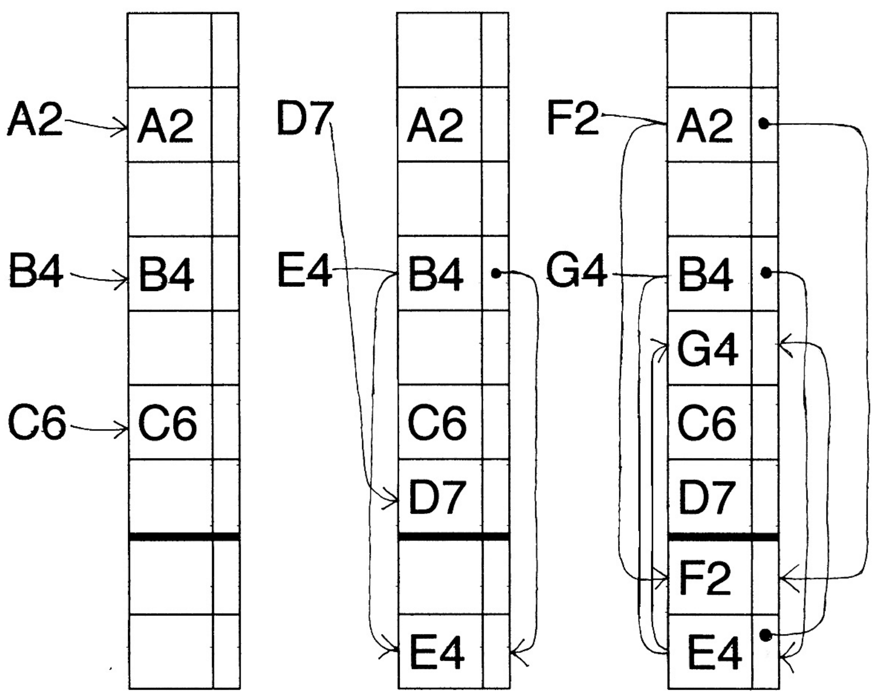

Figure 5. In the first column, A2, B4, and C6 are hashed to positions 2, 4, and 6, respectively. In the second column, D7 is hashed to position 7. E4 would go in position 4, but B4 is there, so it is placed in the lowest available slot (9). A pointer from 4 to 9 is then made. In the third column, F2 collides with A2, and so it is placed in the lowest available spot (8), and a pointer is made from 2 to 8. G4 collides with B4, so the pointer is followed to E4. Since this is not a match, and E4 has a null pointer, G4 is put in the lowest available position (5), and a pointer is made from 9 to 5. There is no secondary clustering in this example, but if H5 were added, then secondary clustering would occur.

Rehashing

What happens when the table gets too full—when there's no more space in the table to put in any more keys? Then you have to *rehash*. Rehashing consists of allocating space for a larger array. Then you have to transfer all the keys from the old array into the new array. Of course, since the table size matters in computing their positions, this requires going to each key, getting its hash value, and recomputing its hash location. After you've rehashed, new keys can just be hashed into the table as normal, using its new size to compute the hash location

How does the algorithm know when to rehash? It knows how much space is allocated for the table, and every time a new key gets added, it increments a counter, so it knows how much of the space allocated is actually being used.

Rehashing slows things down a bit, because it has to go through the entire table, to get all the keys to rehash. However, rehashing causes a subsequent speedup, because clustering has been reduced.

For linked lists, rehashing is not forced, but can improve efficiency. Inside the lists, no new allocation is required—pointers can simply be changed to make the new chains.

For linear probing, it is often a bad idea to wait until the table is completely full until rehashing, because mostly full tables have a very large amount of secondary clustering. Rehashing when the table is only 70–80% full actually speeds the program up on average, despite the additional time taken for the rehashes. To experiment with how full the table should be before rehashing, I introduced a parameter (FULLNESS) into the program, and I tested linear probing rehashing when 70%, 80%, 90%, or 100% full.

Cellar has very little secondary clustering, and so can be rehashed when completely full. One parameter for cellar that I did not experiment with was what fraction of the table is reserved for the cellar. I chose a constant 14% as recommended in the PDF of a lecture by Dr. Hussain.

You don't need to rehash for linear search when you need more space—you can just transfer the data as a block, because linear search has no order to the keys and no gaps.

Testing Procedures

First, I wrote a linear-search C program as a control to see how much hashing helped compared to the simpler algorithm. Also linear search allowed me to find several bugs n my hash code, comparing the buggy code's output to that of linear search. I used TextEdit v1.3 to create the files, and Gnu C Compiler v3.3 to compile them. All experiments were performed on a 1.8GHz PowerPC G5 with 512 MB DDR SDRAM, running Mac OS 10.3.9.

I wrote linear probing, linked list, and cellar collision resolution algorithms in C. I debugged them by comparing the number of distinct words with that output be the linear-search program. Most of the bugs I found had to do with rehashing moving things around or forgetting to adding one to the string length for memory allocation when copying strings.

Each run of a program outputted number of words, number of distinct words, time, number of string comparisons, and memory usage in bytes. To reduce errors in time measurement, all the words were read into an array before timing was started and the hashing routine called. I used getrusage to find the time used, but getrusage only returned time to the hundredth of a second, which is too large an increment for some of the smaller inputs. Time also had a lot of random variability, especially for the small measurements (see Figure 6), and so in the final analysis I used only string comparisons and memory usage.

To get more control over the manipulated variables (number of words and number of distinct words), I wrote a program whose output is all the distinct words in its input.

I used GNU Make version 3.79 to control the compiling and to run the experiment (see Appendix 2). I then graphed the results using Gnuplot v3.8j, as shown in the Results section.

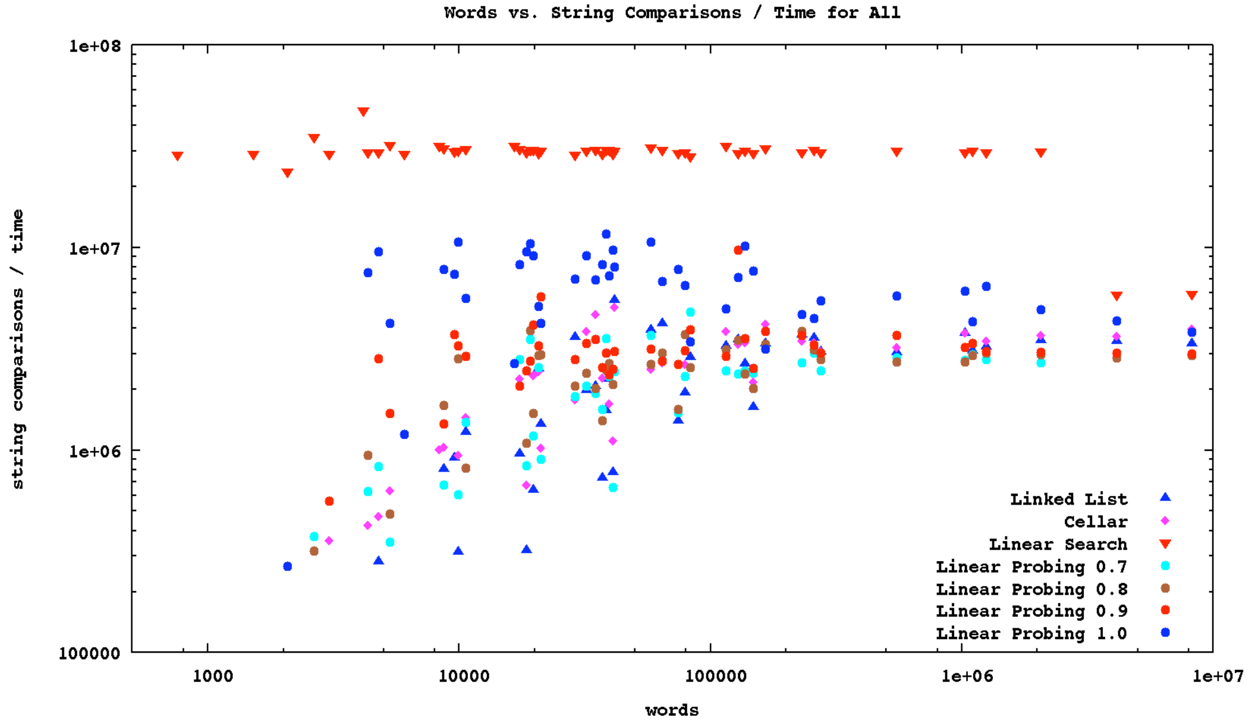

Figure 6. This graph shows the relationship between string comparisons divided by time on the y-axis and words on the x-axis. Linear search has the most string comparisons per second and the least variability, because it does not have the overhead of hashing. For inputs over 20,000 words, string comparisons per second is fairly constant. Under that, the time measurements are unreliable, and so the spread is bigger. Because the time measurements were so unreliable, I used string comparisons as a proxy in subsequent graphs.

Procedure

Here are the steps of my project in list form:

1) Write a linear-search program in C as a control.
2) Write linear probing, linked list, and cellar collision resolution algorithms in C. To reduce errors in time measurement, all the words are read into an array before timing is started and the hashing routine called.
3) Debug programs by comparing the number of distinct words with that output by the linear-search program.
4) Write a program which outputs all the distinct words in its input, to get more control over the manipulated variables (number of words and number of distinct words).
5) Select 6 input files from Project Gutenberg (see Input Files).
6) For each hashing method, do fifty tests: 8 for each of the six files and 2 extra for a concatenation of all the files. The 8 tests consist of two sets of four. One set consists of the file, a concatenation of the file with itself, four copies of the file, and eight copies of the file. The other set consists of all the distinct words in the file, as well as 2, 4, and 8 copies of all the distinct words. The two extra runs are a concatenation of all the files and a concatenation of all the distinct word files.
7) Plot results using gnuplot.

The programs for linear search and the three hashing methods are provided in Appendices 3 through 7.

Input Files

For each hashing method, I did fifty tests: 8 for each of the six files and 2 extra for a concatenation of all the files. The 8 tests consisted of two sets of four. One set consisted of the file, a concatenation of the file with itself, four copies of the file, and eight copies of the file. The other set consisted of all the distinct words in the file, as well as 2, 4, and 8 copies of all the distinct words. The two extra runs were on a concatenation of all the files and a concatenation of all the distinct word files.

The purpose of the self concatenations and of the distinct-word files was to get a better idea of whether it was the number of words or the number of distinct words that was affecting the results.

File	Author	Words	Distinct Words
The Gift of the Magi	O. Henry	2079	758
The Importance of Being Earnest	Oscar Wilde	20741	2649
A Christmas Carol	Charles Dickens	28856	4342
The Call of the Wild	Jack London	32122	4784
A Tale of Two Cities	Charles Dickens	137477	9882
Complete Douay-Rheims Bible	Anonymous	1029071	18556

Table 1. I used six different data files downloaded from Project Gutenberg. The reason I wanted several different sizes was to determine how the hashing was affected by different numbers of words and distinct words.

Results

Data

For each trial, I recorded words, distinct words, time (in seconds), number of string comparisons, and memory usage (in bytes). In Appendix 1, I have a complete list of all the data I collected.

Graphs

All of the graphs in this section use a logarithmic scale on both axes, because the data was spread over such a wide range.

Figure 7 compares linear search with four variants of linear probing. Figure 8 shows how the number of string comparions is affected by number of distinct words for all methods. Figure 9 makes the comparison clearer, by plotting string comparisons per word on the y axis. Figure 10 demonstrates how linear search is dependent on the product of the number of words and number of distinct words. Figure 11 shows that linked list and cellar are very similar in speed. Figure 12 shows that memory use, which is similar for all methods, depends only on distinct words.

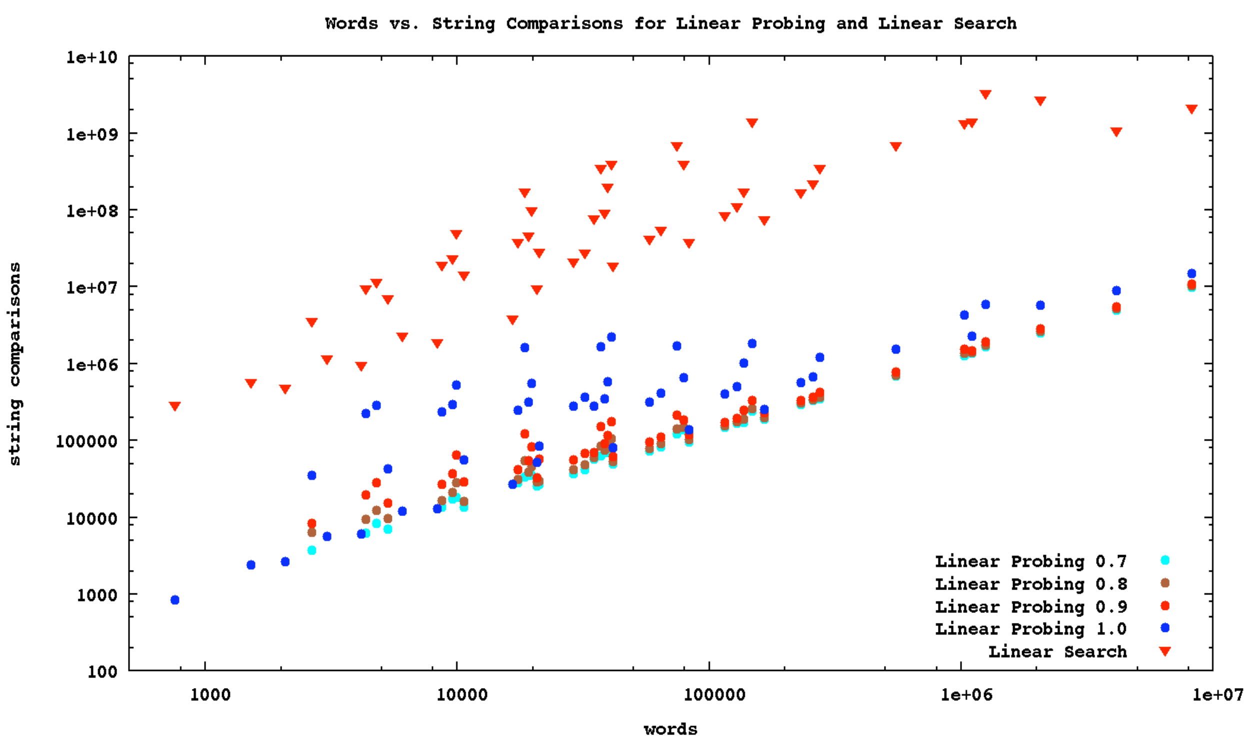

Figure 7. This figure demonstrates how lowering the FULLNESS parameter on linear probing causes it to approximate a straight line—a constant number of string comparisons per word looked up. Note that with FULLNESS=1 (that is, not rehashing until the table is completely full), linear probing is almost as slow as linear search.

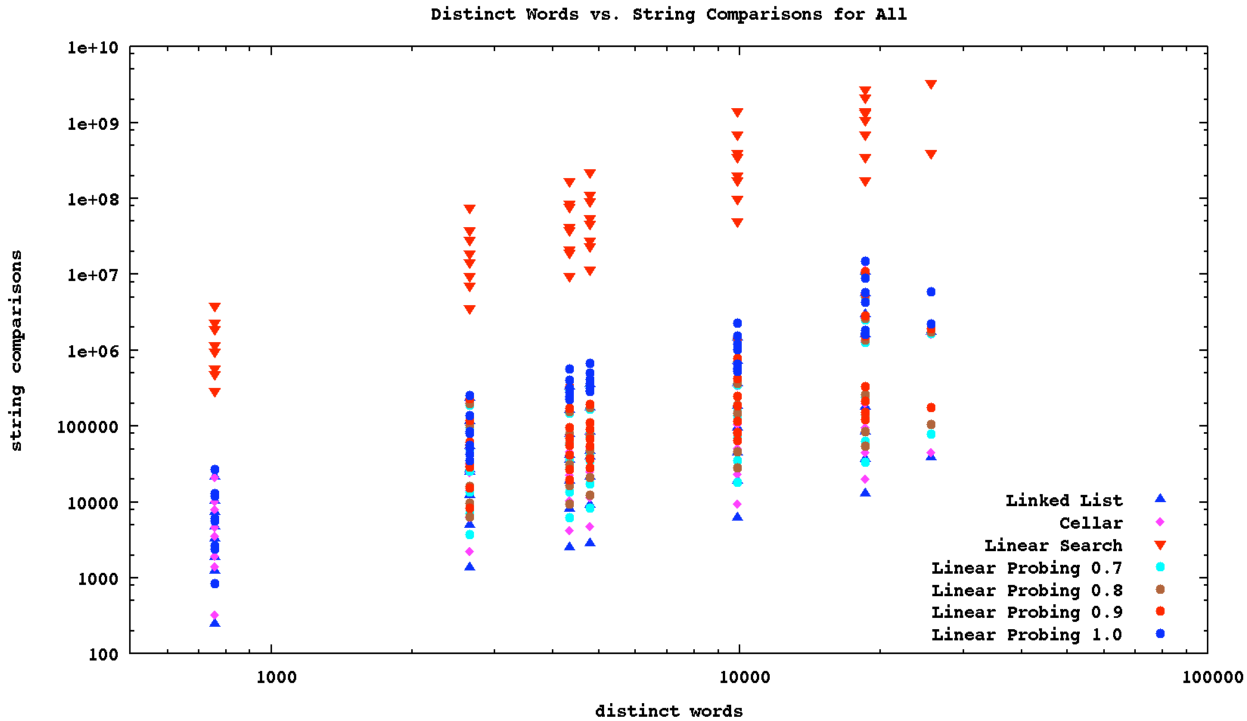

Figure 8. This graph shows that number of string comparisons does not depend very much on number of distinct words, except for linear probing at FULLNESS 1.0 and linear search.

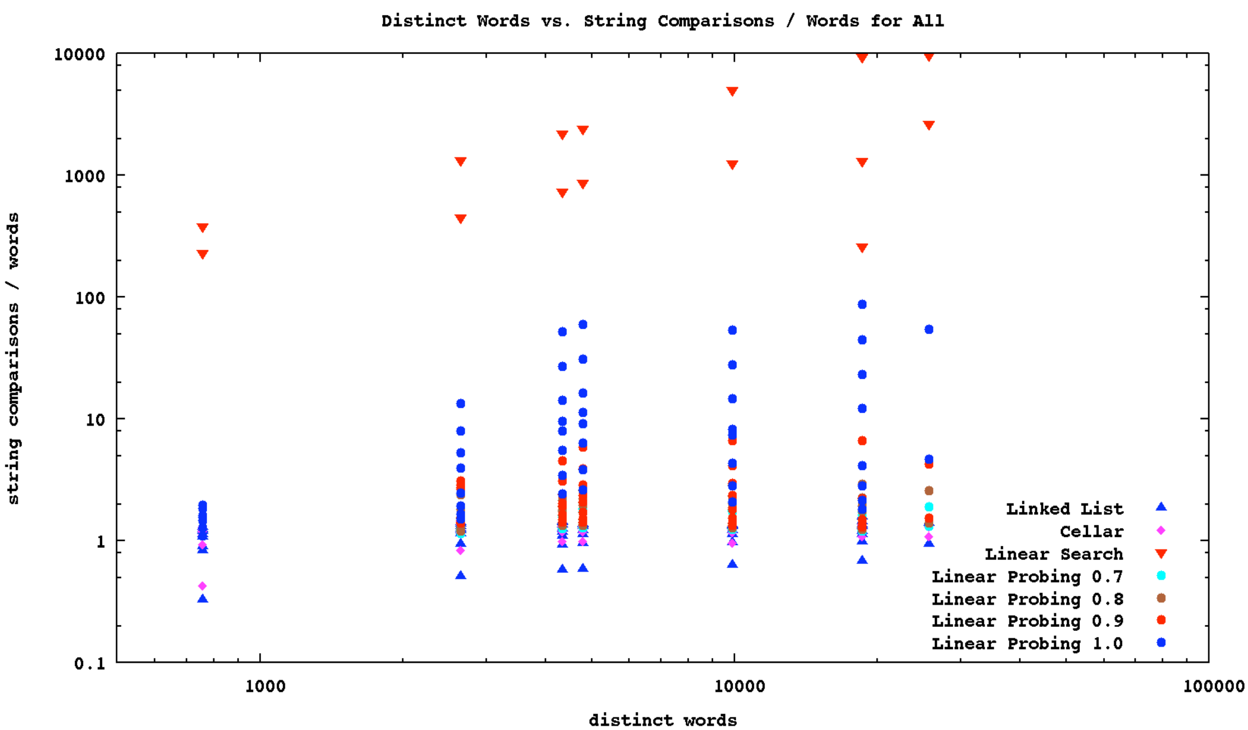

Figure 9.This graph shows that for all the CRAs, except linear probing 1.0 and linear search, string comparisons per word is a constant of about 2.

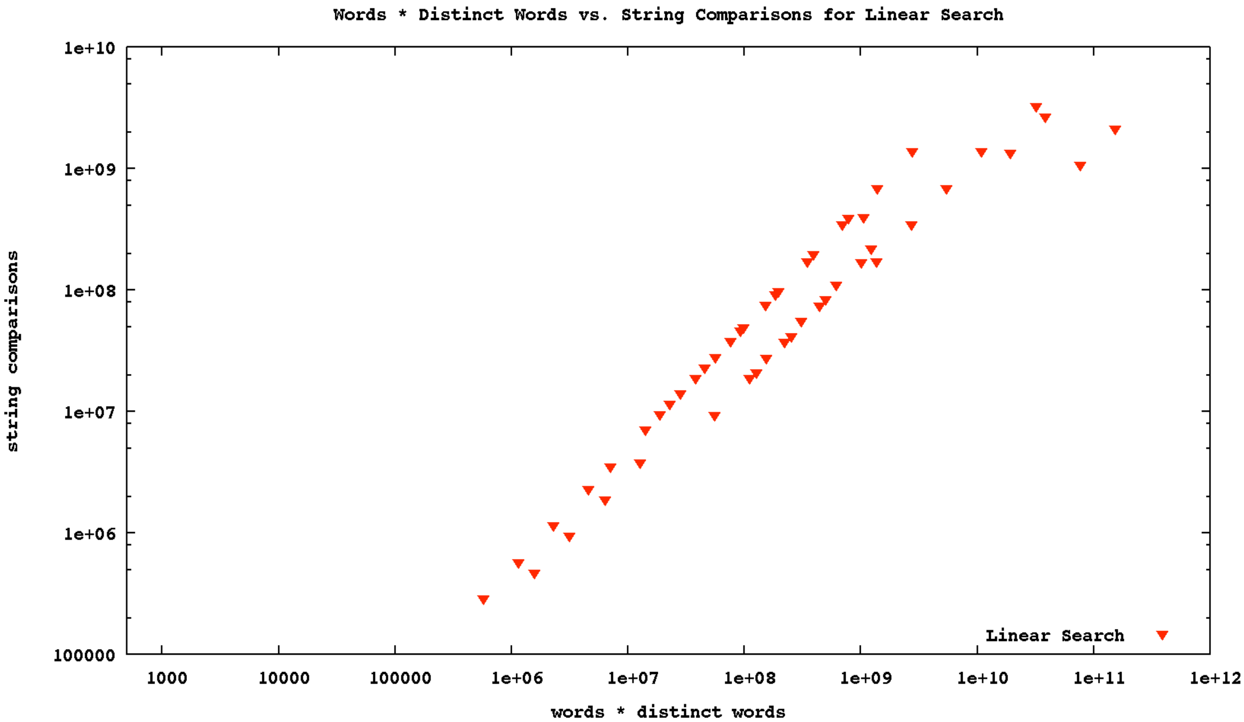

Figure 10. This graph shows that for linear search the number of string comparisons is linearly dependent on the number of words multiplied by number of distinct words.

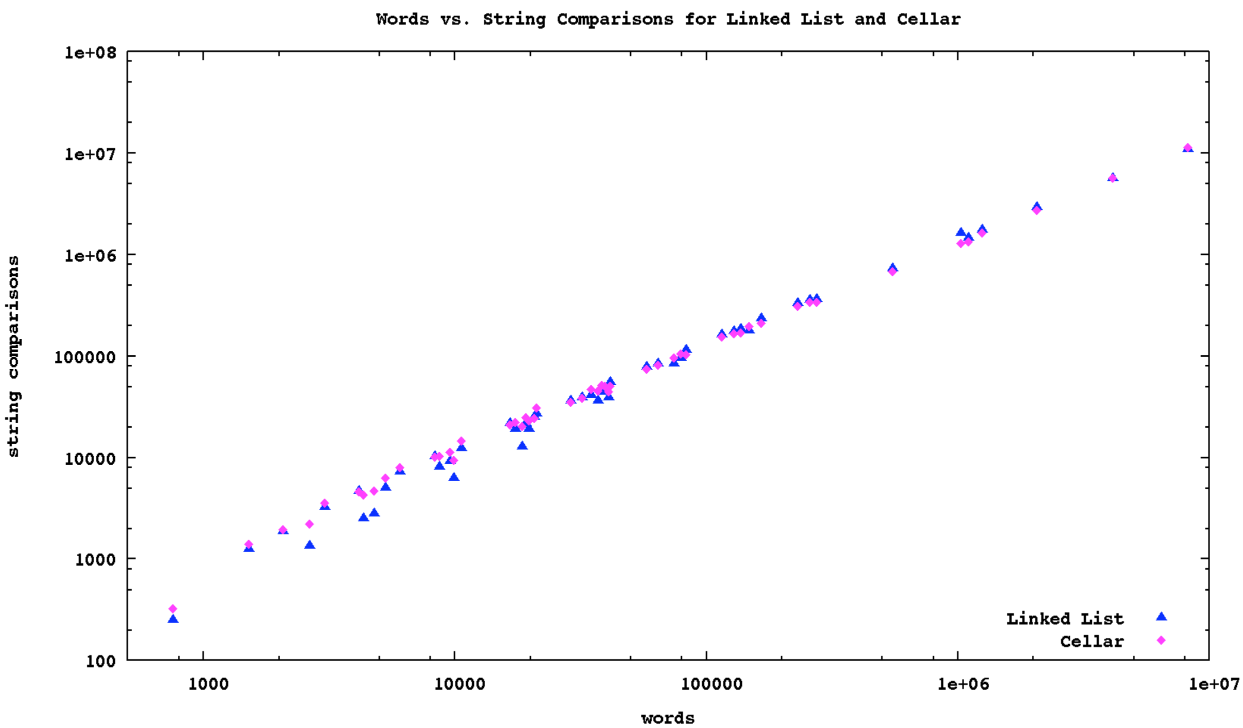

Figure 11. This graph shows that for linked list and cellar, there is a linear relationship between words and string comparisons.

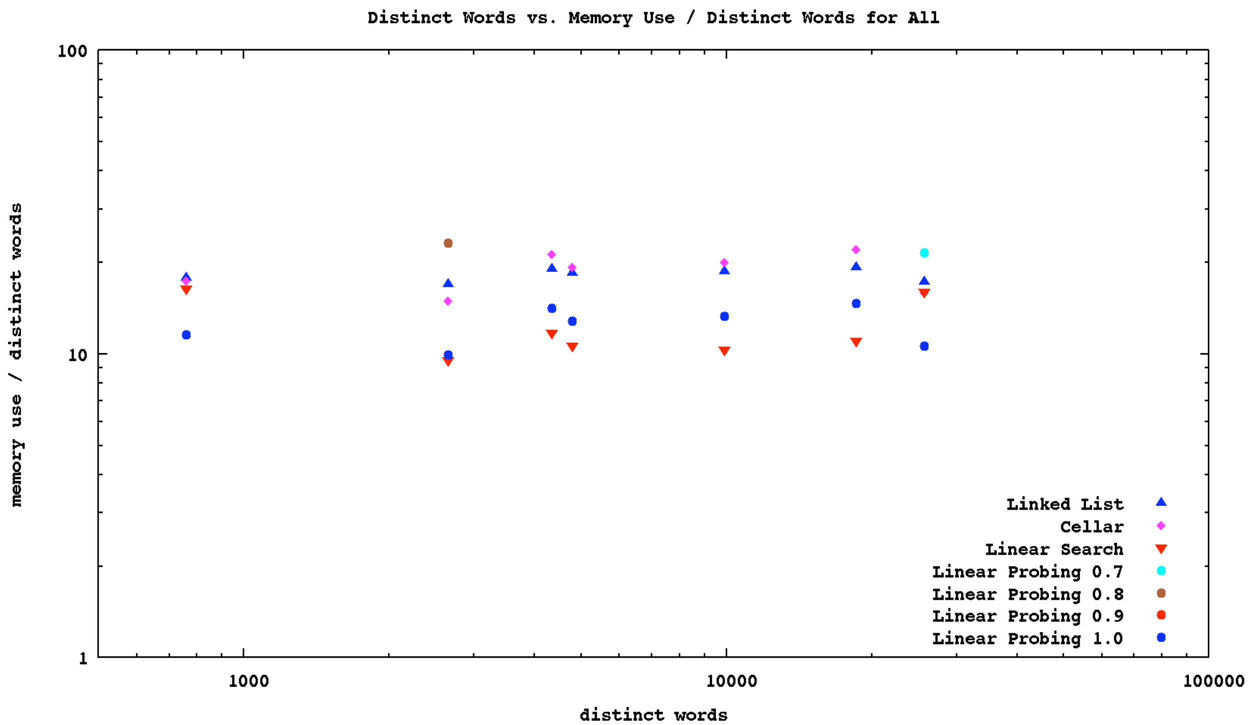

Figure 12. The graph shows that memory use divided by the number of distinct words is roughly constant, using about 10 to 20 bytes per distinct word. Linked list and cellar generally use the most memory per distinct word, due to the pointers.

Conclusions

As hypothesized, hashing was a hundred times faster than linear search, even for small data sets. The three collision resolution algorithms, linear probe, linked list, and cellar, were comparable in speed. Of them, linear probing generally took the longest, particularly if the table was allowed to get too full before rehashing, but also took the least memory. Cellar and linked list were very similar with respect to string comparisons and memory usage, but linked list usually used a little less memory.

What I Learned

For this project, I learned how to program in C, particularly concerning pointers, strings, and memory allocation. I learned how to use Make, a tiny bit of awk, and more about gnuplot. I learned a lot more on how to debug programs, and I learned how to better present data in gnuplot. I also learned again that science fair projects always take a lot longer than I expect.

Acknowledgments

To my father, Kevin Karplus, for helping me understand C and Make, as well as providing me assistance on the more confusing parts of hashing.
To my mother, Michele Hart, for drawing the arrows on the diagrams in this report.
To the Project Gutenberg website, from which my input files were downloaded.

Bibliography

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms* (Cambridge, Massachusetts: MIT Press) and (New York: McGraw-Hill Book Publishing Company) 1990.
Drozdek, Adam and Donald L. Simon. *Data Structures in C* (Boston: PWS Publishing Company) 1995.
Gookin, Dan. *C for Dummies* (Hoboken, NJ: Wiley Publications) 2004.
Hussain, Muhammad. *Hash Tables* http://faculty.ksu.edu.sa/mhussain/CSC212/Lecture%20-%20Hashing.pdf
Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language, Second Edition* (Englewood Cliffs, NJ: Prentice Hall) 1988.
Project Gutenberg. http://www.gutenberg.org, 2008.
Wikipedia. *Hash Function* http://en.wikipedia.org/wiki/Hash_function, 2008.

Mentor Statement

When Abe started this project, he knew how to program in a few simple programming languages, mainly drag-and-drop languages like Scratch and various Lego Mindstorms tools. His first task was to learn a programming language that allowed arrays and pointers, so that he could implement hash tables. I chose C for him (rather than Java), since he had some exposure to NQC for Lego robotics, C is a smaller language to learn, and I did not want Java garbage collection to add noise to his timing measurements. The explicit memory allocation in C also makes it easier to see how much memory is used for different data structures.

He did some initial programming exercises to learn about memory allocation and strings represented as arrays of characters, then dove into the writing of hash table algorithms. He chose the collision resolution algorithms himself from his reading, and he did not need explanations from me. He wrote all the programs himself, but he did need more debugging help than I would have given to a college student, but not too much more. The expected problems with C programming (allocating string arrays with one too few bytes and off-by-one errors in array subscripts) were indeed the most common problems. I did teach him about using printf statements to determine where things were failing, but did not provide him with a modern debugging environment.

In addition to teaching him C, I also taught Abe how to use gnu make for controlling compilation and running his experiments. Much of the makefile he ended up with is my design, but the amount of help I gave him was comparable to what I give to graduate students, as the sophisticated use of makefiles is not often taught to undergrads. I believe that he understands what everything in the makefile does, and he has modified the makefile as needed, but I don't think he could create a similarly complex makefile from scratch.

For the experimental design, I suggested the strategy of reading his entire input file into an array, so that the timing of the hash tables could be separated from the I/O time, and using the getrusage function to measure user time. It turned out that getrusage on the Max OS X system he was using did not have good resolution and was not very repeatable for the small times he needed to measure. As a proxy for time, I suggested using the number of string comparisons done by the algorithm (a standard substitution used in theoretical computer science). I also suggested that he measure memory usage as well.

Some other aspects of the experimental design that I suggested: measuring both the number of words and the number of distinct words, checking each algorithm to make sure it got the same results as the simple linear search, getting text files from the Gutenberg project to use as inputs, and repeating the same input text multiple times to get files with the same number of distinct words but different numbers of total words.

I also helped Abe learn to use gnuplot for displaying his results, though he has had some experience with it from previous projects.

Appendix 1: Output Data

Complete set of results.The first column is words, the second is distinct words, the third is time in seconds, the fourth is string comparisons, the fifth is memory usage in bytes.

Linear Search

758	758	0.010000	286903	12400
1516	758	0.020000	574564	12400
3032	758	0.040000	1149886	12400
6064	758	0.080000	2300530	12400
2649	2649	0.100000	3507276	25200
5298	2649	0.220000	7017201	25200
10596	2649	0.460000	14037051	25200
21192	2649	0.940000	28076751	25200
4342	4342	0.320000	9424311	50800
8684	4342	0.610000	18852964	50800
17368	4342	1.240000	37710270	50800
34736	4342	2.490000	75424882	50800
4784	4784	0.390000	11440936	50800
9568	4784	0.770000	22886656	50800
19136	4784	1.510000	45778096	50800
38272	4784	3.020000	91560976	50800
9882	9882	1.630000	48822021	102000
19764	9882	3.230000	97653924	102000
39528	9882	6.440000	195317730	102000
79056	9882	13.290000	390645342	102000
18556	18556	5.880000	172153290	204400
37112	18556	11.930000	344325136	204400
74224	18556	23.550000	688668828	204400
148448	18556	47.260000	1377356212	204400
2079	758	0.020000	472308	12400
4158	758	0.020000	945374	12400
8316	758	0.060000	1891506	12400
16632	758	0.120000	3783770	12400
20741	2649	0.320000	9306567	25200
41482	2649	0.620000	18615783	25200
82964	2649	1.330000	37234215	25200
165928	2649	2.420000	74471079	25200
28856	4342	0.730000	20872314	50800
57712	4342	1.340000	41748970	50800
115424	4342	2.640000	83502282	50800
230848	4342	5.680000	167008906	50800
32122	4784	0.920000	27447538	50800
64244	4784	1.810000	54899860	50800
128488	4784	3.780000	109804504	50800
256976	4784	7.240000	219613792	50800

137477	9882	5.740000	171494208	102000
274954	9882	11.710000	342998298	102000
549908	9882	23.000000	686006478	102000
1099816	9882	45.920000	1372022838	102000
1029071	18556	45.420000	1337866642	204400
2058142	18556	89.890000	2675751840	204400
4116284	18556	180.750000	1056554940	204400
8232568	18556	360.040000	2113128436	204400
40971	25673	13.590000	393428946	409200
1250346	25673	111.310000	3258786725	409200

Linked list

758	758	0.000000	251	13484
1516	758	0.000000	1260	13484
3032	758	0.000000	3278	13484
6064	758	0.000000	7314	13484
2649	2649	0.000000	1360	44952
5298	2649	0.000000	5015	44952
10596	2649	0.010000	12325	44952
21192	2649	0.020000	26945	44952
4342	4342	0.000000	2510	82820
8684	4342	0.010000	8084	82820
17368	4342	0.020000	19232	82820
34736	4342	0.020000	41528	82820
4784	4784	0.010000	2822	88124
9568	4784	0.010000	9157	88124
19136	4784	0.000000	21827	88124
38272	4784	0.030000	47167	88124
9882	9882	0.020000	6304	184404
19764	9882	0.030000	19148	184404
39528	9882	0.020000	44836	184404
79056	9882	0.050000	96212	184404
18556	18556	0.040000	12796	358700
37112	18556	0.050000	36554	358700
74224	18556	0.060000	84070	358700
148448	18556	0.110000	179102	358700
2079	758	0.000000	1883	13484
4158	758	0.000000	4705	13484
8316	758	0.000000	10349	13484
16632	758	0.000000	21637	13484
20741	2649	0.010000	25270	44952
41482	2649	0.010000	55217	44952
82964	2649	0.040000	115111	44952
165928	2649	0.070000	234899	44952
28856	4342	0.010000	36216	82820
57712	4342	0.020000	78757	82820

115424	4342	0.050000	163839	82820
230848	4342	0.090000	334003	82820
32122	4784	0.020000	39434	88124
64244	4784	0.020000	84794	88124
128488	4784	0.050000	175514	88124
256976	4784	0.100000	356954	88124
137477	9882	0.070000	186692	184404
274954	9882	0.120000	368377	184404
549908	9882	0.240000	731747	184404
1099816	9882	0.470000	1458487	184404
1029071	18556	0.430000	1634367	358700
2058142	18556	0.850000	2965369	358700
4116284	18556	1.630000	5627373	358700
8232568	18556	3.270000	10951381	358700
40971	25673	0.050000	38950	444104
1250346	25673	0.540000	1751895	444104

Cellar

758	758	0.000000	326	13164
1516	758	0.000000	1410	13164
3032	758	0.010000	3578	13164
6064	758	0.000000	7914	13164
2649	2649	0.000000	2217	39492
5298	2649	0.010000	6306	39492
10596	2649	0.010000	14484	39492
21192	2649	0.030000	30840	39492
4342	4342	0.010000	4272	92148
8684	4342	0.010000	10326	92148
17368	4342	0.010000	22434	92148
34736	4342	0.010000	46650	92148
4784	4784	0.010000	4714	92148
9568	4784	0.000000	11404	92148
19136	4784	0.000000	24784	92148
38272	4784	0.020000	51544	92148
9882	9882	0.010000	9479	197460
19764	9882	0.010000	23289	197460
39528	9882	0.030000	50909	197460
79056	9882	0.040000	106149	197460
18556	18556	0.030000	20190	408084
37112	18556	0.020000	45327	408084
74224	18556	0.060000	95601	408084
148448	18556	0.090000	196149	408084
2079	758	0.000000	1945	13164
4158	758	0.000000	4648	13164
8316	758	0.010000	10054	13164
16632	758	0.000000	20866	13164

20741	2649	0.010000	24345	39492
41482	2649	0.010000	50693	39492
82964	2649	0.030000	103389	39492
165928	2649	0.050000	208781	39492
28856	4342	0.020000	35456	92148
57712	4342	0.030000	74993	92148
115424	4342	0.040000	154067	92148
230848	4342	0.090000	312215	92148
32122	4784	0.010000	38475	92148
64244	4784	0.030000	81328	92148
128488	4784	0.050000	167034	92148
256976	4784	0.100000	338446	92148
137477	9882	0.050000	170015	197460
274954	9882	0.110000	338127	197460
549908	9882	0.210000	674351	197460
1099816	9882	0.400000	1346799	197460
1029071	18556	0.340000	1284897	408084
2058142	18556	0.740000	2738593	408084
4116284	18556	1.540000	5645985	408084
8232568	18556	2.880000	11460769	408084
40971	25673	0.040000	44679	408084
1250346	25673	0.470000	1625990	408084

Linear Probing (0.7)

758	758	0.000000	836	8776
1516	758	0.000000	2430	8776
3032	758	0.000000	5618	8776
6064	758	0.000000	11994	8776
2649	2649	0.010000	3737	61432
5298	2649	0.020000	7051	61432
10596	2649	0.010000	13679	61432
21192	2649	0.030000	26935	61432
4342	4342	0.010000	6228	61432
8684	4342	0.020000	13490	61432
17368	4342	0.010000	28014	61432
34736	4342	0.030000	57062	61432
4784	4784	0.010000	8343	61432
9568	4784	0.000000	17290	61432
19136	4784	0.010000	35184	61432
38272	4784	0.020000	70972	61432
9882	9882	0.030000	18082	131640
19764	9882	0.030000	35288	131640
39528	9882	0.030000	69700	131640
79056	9882	0.060000	138524	131640
18556	18556	0.040000	33639	272056
37112	18556	0.040000	63267	272056

74224	18556	0.080000	122523	272056
148448	18556	0.100000	241035	272056
2079	758	0.000000	2679	8776
4158	758	0.000000	6116	8776
8316	758	0.000000	12990	8776
16632	758	0.010000	26738	8776
20741	2649	0.010000	25462	61432
41482	2649	0.020000	48990	61432
82964	2649	0.020000	96046	61432
165928	2649	0.060000	190158	61432
28856	4342	0.020000	36664	61432
57712	4342	0.020000	73743	61432
115424	4342	0.060000	147901	61432
230848	4342	0.110000	296217	61432
32122	4784	0.020000	41318	61432
64244	4784	0.030000	83035	61432
128488	4784	0.070000	166469	61432
256976	4784	0.110000	333337	61432
137477	9882	0.070000	173255	131640
274954	9882	0.140000	343973	131640
549908	9882	0.240000	685409	131640
1099816	9882	0.460000	1368281	131640
1029071	18556	0.460000	1282081	272056
2058142	18556	0.930000	2522118	272056
4116284	18556	1.750000	5002192	272056
8232568	18556	3.360000	9962340	272056
40971	25673	0.120000	78526	552888
1250346	25673	0.590000	1660013	552888

Linear Probing (0.8)

758	758	0.000000	836	8776
1516	758	0.000000	2430	8776
3032	758	0.000000	5618	8776
6064	758	0.010000	11994	8776
2649	2649	0.020000	6361	61432
5298	2649	0.020000	9675	61432
10596	2649	0.020000	16303	61432
21192	2649	0.010000	29559	61432
4342	4342	0.010000	9415	61432
8684	4342	0.010000	16677	61432
17368	4342	0.000000	31201	61432
34736	4342	0.030000	60249	61432
4784	4784	0.000000	12281	61432
9568	4784	0.000000	21228	61432
19136	4784	0.010000	39122	61432
38272	4784	0.030000	74910	61432

9882	9882	0.010000	28349	131640
19764	9882	0.030000	45555	131640
39528	9882	0.030000	79967	131640
79056	9882	0.040000	148791	131640
18556	18556	0.050000	54298	272056
37112	18556	0.060000	83926	272056
74224	18556	0.090000	143182	272056
148448	18556	0.130000	261694	272056
2079	758	0.010000	2679	8776
4158	758	0.000000	6116	8776
8316	758	0.000000	12990	8776
16632	758	0.000000	26738	8776
20741	2649	0.010000	29255	61432
41482	2649	0.000000	53675	61432
82964	2649	0.040000	102515	61432
165928	2649	0.060000	200195	61432
28856	4342	0.020000	41271	61432
57712	4342	0.030000	79496	61432
115424	4342	0.050000	155946	61432
230848	4342	0.080000	308846	61432
32122	4784	0.020000	47784	61432
64244	4784	0.030000	90014	61432
128488	4784	0.050000	174474	61432
256976	4784	0.110000	343394	61432
137477	9882	0.080000	190733	131640
274954	9882	0.130000	364345	131640
549908	9882	0.260000	711569	131640
1099816	9882	0.480000	1406017	131640
1029071	18556	0.500000	1357820	272056
2058142	18556	0.890000	2631869	272056
4116284	18556	1.810000	5179967	272056
8232568	18556	3.500000	10276163	272056
40971	25673	0.050000	105566	272056
1250346	25673	0.580000	1754548	272056

Linear Probing (0.9)

758	758	0.000000	836	8776
1516	758	0.000000	2430	8776
3032	758	0.010000	5618	8776
6064	758	0.000000	11994	8776
2649	2649	0.000000	8262	26328
5298	2649	0.010000	15218	26328
10596	2649	0.010000	29130	26328
21192	2649	0.010000	56954	26328
4342	4342	0.000000	19620	61432
8684	4342	0.020000	26882	61432

17368	4342	0.020000	41406	61432
34736	4342	0.020000	70454	61432
4784	4784	0.010000	28291	61432
9568	4784	0.010000	37238	61432
19136	4784	0.020000	55132	61432
38272	4784	0.030000	90920	61432
9882	9882	0.020000	65549	131640
19764	9882	0.020000	82755	131640
39528	9882	0.050000	117167	131640
79056	9882	0.060000	185991	131640
18556	18556	0.050000	123210	272056
37112	18556	0.060000	152838	272056
74224	18556	0.080000	212094	272056
148448	18556	0.130000	330606	272056
2079	758	0.000000	2679	8776
4158	758	0.000000	6116	8776
8316	758	0.000000	12990	8776
16632	758	0.010000	26738	8776
20741	2649	0.010000	32636	26328
41482	2649	0.020000	61146	26328
82964	2649	0.030000	118166	26328
165928	2649	0.060000	232206	26328
28856	4342	0.020000	55842	61432
57712	4342	0.030000	95084	61432
115424	4342	0.060000	173568	61432
230848	4342	0.090000	330536	61432
32122	4784	0.020000	67436	61432
64244	4784	0.040000	109959	61432
128488	4784	0.020000	195005	61432
256976	4784	0.110000	365097	61432
137477	9882	0.070000	248697	131640
274954	9882	0.140000	424364	131640
549908	9882	0.210000	775698	131640
1099816	9882	0.440000	1478366	131640
1029071	18556	0.480000	1538098	272056
2058142	18556	0.940000	2851647	272056
4116284	18556	1.810000	5478745	272056
8232568	18556	3.590000	10732941	272056
40971	25673	0.070000	176360	272056
1250346	25673	0.630000	1934315	272056

Linear Probing (1.0)

758	758	0.000000	836	8776
1516	758	0.000000	2430	8776
3032	758	0.000000	5618	8776
6064	758	0.010000	11994	8776

2649	2649	0.000000	35477	26328
5298	2649	0.010000	42433	26328
10596	2649	0.010000	56345	26328
21192	2649	0.020000	84169	26328
4342	4342	0.030000	226520	61432
8684	4342	0.030000	233782	61432
17368	4342	0.030000	248306	61432
34736	4342	0.040000	277354	61432
4784	4784	0.030000	286756	61432
9568	4784	0.040000	295703	61432
19136	4784	0.030000	313597	61432
38272	4784	0.030000	349385	61432
9882	9882	0.050000	530966	131640
19764	9882	0.060000	548172	131640
39528	9882	0.080000	582584	131640
79056	9882	0.100000	651408	131640
18556	18556	0.170000	1626892	272056
37112	18556	0.200000	1656520	272056
74224	18556	0.220000	1715776	272056
148448	18556	0.240000	1834288	272056
2079	758	0.010000	2679	8776
4158	758	0.000000	6116	8776
8316	758	0.000000	12990	8776
16632	758	0.010000	26738	8776
20741	2649	0.010000	51447	26328
41482	2649	0.010000	80003	26328
82964	2649	0.040000	137115	26328
165928	2649	0.080000	251339	26328
28856	4342	0.040000	278809	61432
57712	4342	0.030000	319364	61432
115424	4342	0.080000	400474	61432
230848	4342	0.120000	562694	61432
32122	4784	0.040000	365951	61432
64244	4784	0.060000	409512	61432
128488	4784	0.070000	496634	61432
256976	4784	0.150000	670878	61432
137477	9882	0.100000	1020312	131640
274954	9882	0.220000	1201862	131640
549908	9882	0.270000	1564962	131640
1099816	9882	0.530000	2291162	131640
1029071	18556	0.700000	4269098	272056
2058142	18556	1.170000	5793292	272056
4116284	18556	2.030000	8841680	272056
8232568	18556	3.890000	14938456	272056
40971	25673	0.230000	2228186	272056
1250346	25673	0.910000	5853193	272056

Appendix 2: Makefile

Each line in the makefile with a colon is a rule, which gives instructions on how to make a specific file (the final name is given before the colon) from other files (listed after the colon). The indented lines below the rule provide the actual instructions for making the file. The shell command "make out_linked" checks to make sure that the executable program "test_linked" is up-to-date, then runs it for all 50 data sets.

```
# Makefile: Abe Karplus 1-29-2009

test_l_probe_%:         c17+_hasher1_2_v2_%.o readword.o hash1_2.o
            gcc -o $@ $^
c17+_hasher1_2_v2_%.o: c17+_hasher1_2_v2.c
            gcc -o $@ -c -D FULLNESS=$* $^
test_linear:            c15_word_store_search_v2.o readword.o linear_search.o
            gcc -o $@ $^
test_linked:            c18_hasher2_v2.o readword.o hash2.o
            gcc -o $@ $^
test_cellar:            c19_hasher3_v2.o readword.o hash3.o
            gcc -o $@ $^
test_distinct:          c20_distinct_words.o hash3.o readword.o
            gcc -o $@ $^
all.% : test_%
            cat input/magi.txt input/cities.txt input/carol.txt input/
wild.txt\
                        input/bible.txt | test_$* >> output/all_$*.txt
%.l_probe: test_l_probe
            test_l_probe < input/$*.txt >> output/$*_l_probe.txt
%.linear: test_linear
            test_linear < input/$*.txt >> output/$*_linear.txt
%.linked: test_linked
            test_linked < input/$*.txt >> output/$*_linked.txt
%.cellar: test_cellar
            test_cellar < input/$*.txt >> output/$*_cellar.txt

INPUTS = magi earnest carol wild cities bible

# run needs two arguments: the first is the CRA type,
# the second is an input file.
# Note: this macro designed by Kevin Karplus.
define run
            test_$(1) < $(2) >> output/$(1).txt
            cat $(foreach x,1 2, $(2) ) | test_$(1) >> output/$(1).txt
            cat $(foreach x,1 2 3 4, $(2) ) | test_$(1) >> output/$(1).txt
            cat $(foreach x,1 2 3 4 5 6 7 8, $(2) ) | test_$(1) >> output/
```

```
$(1).txt

endef

out_%:
	make output/$*.txt output/$*_grid.txt
output/%.txt: test_%
	echo '# $*' > $@
	$(foreach x,${INPUTS}, $(call run,$*,input/${x}_distinct.txt))
	$(foreach x,${INPUTS}, $(call run,$*,input/${x}.txt))
	cat $(foreach x,${INPUTS}, input/${x}_distinct.txt) | test_$* >>
$@
	cat $(foreach x,${INPUTS}, input/${x}.txt) | test_$* >> $@
%_distinct.txt: %.txt test_distinct
	test_distinct < $*.txt > $@
%_grid.txt: %.txt
	sort -n -k 2 -k 1 < $^ \
	| awk -f grid.awk \
	> $@
```

Appendix 3: Readword

The readword routine is used in all the programs.

readword.h

```
/*Abe Karplus 12-21-2008*/

/*readword.h*/

/*****************************
Readword gets the next word from stdin.
(A word is any sequence of characters containing
no spaces or other punctuation than apostrophes.)
The function returns 0 for EOF or a pointer to a
newly allocated string containing the word if one exists.
*****************************/

#ifndef READWORD_H
#define READWORD_H
char* readword();
#endif
```

readword.c

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*Abe Karplus 12-21-2008*/

/*****************************
Readword gets the next word from stdin.
(A word is any sequence of characters containing
no spaces or other punctuation than apostrophes.)
The function returns 0 for EOF or a pointer to a
newly allocated string containing the word if one exists.
*****************************/

char* readword()
{
#define ISFRWO (isalpha(nexc) || (nexc == '\''))
            char nexc;
            char buffer[50];
            short i;
            char* word;

            for(nexc = getc(stdin); !ISFRWO; nexc = getc(stdin))
            {
```

```
                if(nexc == EOF)
                {
                        return(0);
                }
        }
        for(i = 0; ISFRWO; nexc = getc(stdin))
        {
                buffer[i] = nexc;
                i++;
        }
        buffer[i] = 0;
        word = calloc(strlen(buffer) + 1, 1);
        strcpy(word, buffer);
        return(word);
}
```

Appendix 4: Linear Search

.h file

```
/*Abe Karplus 12-23-2008*/

/*linear_search.h*/

/*********************
Linear Search. Given a pointer to a string, this function
will check its dynamically allocated array for the string.
If the array contains the string, the function returns a
pointer to a struct containing: a) a pointer to the string,
and b) an int called occur. If the string is not in the
table it will be added and a similar struct pointer returned.
*********************/

#ifndef LINEAR_H
#define LINEAR_H
struct data_pair
{
            char* word;
            int occur;
};
struct dyn_table
{
            struct data_pair* words;
            int n_used;
            int n_alloc;
};

struct data_pair* find_create(char* data);

extern struct dyn_table freq_words;
extern unsigned long comps;
#endif
```

.c file

```
#include <string.h>
#include <stdlib.h>
#include "linear_search.h"

/*Abe Karplus 12-23-2008*/

/*********************
Linear Search. Given a pointer to a string, this function will check its
dynamically allocated array
```

```
for the string. If the array contains the string, the function returns a
pointer to a struct
containing: a) a pointer to the string, and b) an int called occur. If the
string is not in the
table it will be added and a similar struct pointer returned.
*********************/


struct dyn_table freq_words = {0, 0, 0};
unsigned long comps = 0;

struct data_pair* find_create(char* data)
{
	int i;

	for(i = 0; i < freq_words.n_used; i++)
	{
		if(comps++, !strcasecmp(freq_words.words[i].word,
data))
		{
			return &(freq_words.words[i]);
		}
	}
	if(freq_words.n_used >= freq_words.n_alloc)
	{
		freq_words.n_alloc *= 2;
		freq_words.n_alloc += 50;
		freq_words.words = realloc(freq_words.words,
freq_words.n_alloc * sizeof(struct data_pair));
	}
	freq_words.words[freq_words.n_used].occur = 0;
	freq_words.words[freq_words.n_used].word = calloc(strlen(data) +
1, 1);
	strcpy(freq_words.words[freq_words.n_used].word, data);
	return &(freq_words.words[freq_words.n_used++]);
}
```

main program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "readword.h"
#include "linear_search.h"
```

```
/*Abe Karplus 12-24-2008*/

struct dyn_array
{
            char** words;
            int n_used;
            int n_alloc;
};

double gettime();

main()
{
            char* word;
            struct dyn_array all_words;
            int i;
            struct data_pair* found;
            double old_time;
            double new_time;

            all_words.n_alloc = 5;
            all_words.words = calloc(all_words.n_alloc, sizeof(char*));
            all_words.n_used = 0;
            while(1)
            {
                        word = readword(); /*Initial input*/
                        if(word == 0)
                                    break;
                        if(all_words.n_used >= all_words.n_alloc)
                        {
                                    all_words.n_alloc *= 2;
                                    all_words.words =
realloc(all_words.words, all_words.n_alloc * sizeof(char*));
                        }
                        all_words.words[all_words.n_used++] = word;
            }
            printf("%d\t", all_words.n_used); /*words*/
            old_time = gettime(); /*timing*/
            for(i = 0; i < all_words.n_used; i++)
            {
                        find_create(all_words.words[i]) -> occur++; /*linear
search*/
            }
            new_time = gettime(); /*timing*/
            printf("%d\t", freq_words.n_used); /*distinct*/
            /*
```

```
				found = find_create("the");
				printf("The word \"the\" occured %d times.\n", found
-> occur);
		*/
		printf("%f\t", (new_time - old_time)); /*time*/
		printf("%lu\t", comps); /*strcmps*/
		printf("%u\n", freq_words.n_alloc * sizeof(struct data_pair)); /
*memuse*/
		return(0);
}

double gettime()
{
		struct rusage elap;

		getrusage(0, &elap);
		return((elap.ru_utime.tv_sec)+1.e-06*(elap.ru_utime.tv_usec));
}
```

Appendix 5: Linear Probing

.h file

```
/*Abe Karplus 12-31-2008*/

/*hash1_2.h*/

#ifndef HASH1_2_H
#define HASH1_2_H
struct data_pair
{
            char* word;
            int occur;
};
struct dyn_table
{
            struct data_pair* words;
            int n_used;
            int n_alloc;
};

struct data_pair* find_create(char* data, double fullness);

extern struct dyn_table freq_words;
extern long comps;
#endif
```

.c file

```
#include <string.h>
#include <stdlib.h>
#include "hash1_2.h"

#define START 1
#define PRIME 999999137

/*Abe Karplus 12-31-2008*/

struct dyn_table freq_words = {0, 0, 0};
long comps = 0;

void rehash();
struct data_pair* find_create_nocpy(char* data);
unsigned long hash1(char* str);

struct data_pair* find_create(char* data, double fullness)
{
```

```
            int i;
        int at;
            unsigned long start_at = hash1(data);
            char* temp_data;

            /*printf("DEBUG: Looking for %s.\n", data);*/
            for(i = 0; i < freq_words.n_alloc; i++)
            {
                        at = (start_at + i) % freq_words.n_alloc;
                        if(freq_words.words[at].word == 0)
                        {
                                    break;
                        }
                        if(comps++, !strcasecmp(freq_words.words[at].word,
data))
                        {
                                    /*printf("DEBUG: Found %s.\n", data);*/
                                    return &(freq_words.words[at]);
                        }
            }
            if(freq_words.n_used >= freq_words.n_alloc * fullness)
            {
                        rehash();
                        temp_data = data; /*because at doesn't work after
rehash*/
                        return(find_create_nocpy(temp_data)); /*ditto*/
            }
            freq_words.n_used++;
            /*printf("DEBUG: at == %d.\n", at);*/
            freq_words.words[at].occur = 0;
            freq_words.words[at].word = calloc(strlen(data) + 1, 1);
            strcpy(freq_words.words[at].word, data);
            /*printf("DEBUG: About to return %s.\n",
freq_words.words[at].word);*/
            return &(freq_words.words[at]);
}

struct data_pair* find_create_nocpy(char* data)
{
            int i;
        int at;
            unsigned long start_at = hash1(data);

            /*printf("DEBUG: Looking for %s.\n", data);*/
            for(i = 0; i < freq_words.n_alloc; i++)
            {
```

```
                        at = (start_at + i) % freq_words.n_alloc;
                        if(freq_words.words[at].word == 0)
                        {
                                    break;
                        }
/**********************************************
DON'T NEED STRCASECMP, SINCE data ALWAYS NEW */
                        if(comps++, !strcasecmp(freq_words.words[at].word,
data))
                        {
                                    /*printf("DEBUG: Found %s.\n", data);*/
                                    return &(freq_words.words[at]);
                        }
            }
            freq_words.n_used++;
            /*printf("DEBUG: at == %d.\n", at);*/
            freq_words.words[at].occur = 0;
            freq_words.words[at].word = calloc(strlen(data) + 1, 1);
            freq_words.words[at].word = data;
            /*printf("DEBUG: About to return %s.\n",
freq_words.words[at].word);*/
            return &(freq_words.words[at]);
}

unsigned long hash1(char* str)
{
            unsigned long val;

            for(val = START; *str != 0; str++)
            {
                        val = (val + tolower(*str)) * PRIME;
            }
            return(val);
}

void rehash()
{
            int i;
            struct dyn_table old_table;

            old_table = freq_words;
            freq_words.n_alloc *= 2;
            freq_words.n_alloc += 1097;
            freq_words.words = calloc(freq_words.n_alloc, sizeof(struct
data_pair));
            freq_words.n_used = 0;
```

```
            for(i = 0; i < old_table.n_alloc; i++)
            {
                        if(old_table.words[i].word != 0)
                        {

            find_create_nocpy(old_table.words[i].word) -> occur =
old_table.words[i].occur;
                        }
            }
            if (old_table.words)
            {
                        free(old_table.words);
            }
            /*printf("DEBUG: Rehashing done.\n");*/
}
```

main program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "readword.h"
#include "hash1_2.h"

/*Abe Karplus 12-31-2008*/

struct dyn_array
{
            char** words;
            int n_used;
            int n_alloc;
};

double gettime();

main()
{
            char* word;
            struct dyn_array all_words;
            int i;
            struct data_pair* found;
            double old_time;
            double new_time;

            all_words.n_alloc = 5;
```

```
            all_words.words = calloc(all_words.n_alloc, sizeof(char*));
            all_words.n_used = 0;
            while(1)
            {
                        word = readword();
                        if(word == 0)
                                    break;
                        if(all_words.n_used >= all_words.n_alloc)
                        {
                                    all_words.n_alloc *= 2;
                                    all_words.words =
realloc(all_words.words, all_words.n_alloc * sizeof(char*));
                        }
                        all_words.words[all_words.n_used++] = word;
            }
            printf("%d\t", all_words.n_used); /*words*/
            old_time = gettime();
            for(i = 0; i < all_words.n_used; i++)
            {
                        found = find_create(all_words.words[i], FULLNESS);
                        /*printf("DEBUG: found -> word = %s.\n", found ->
word);*/
                        found -> occur++;
            }
            new_time = gettime();
            printf("%d\t", freq_words.n_used); /*distinct*/
            printf("%f\t", (new_time - old_time)); /*time*/
            printf("%ld\t", comps); /*strcmps*/
            printf("%u\n", freq_words.n_alloc * sizeof(struct data_pair)); /
*memuse*/
            /*printf("FULLNESS = %lf\n", FULLNESS);*/
            return(0);
}

double gettime()
{
            struct rusage elap;

            getrusage(0, &elap);
            return((elap.ru_utime.tv_sec)+1.e-06*(elap.ru_utime.tv_usec));
}
```

Appendix 6: Linked List

.h file

```
/*Abe Karplus 12-29-2008*/

/*hash2.h*/

#ifndef HASH2_H
#define HASH2_H
struct data_pair
{
            char* word;
            int occur;
            struct data_pair* next;
};
struct dyn_table
{
            struct data_pair** words;
            int n_used;
            int n_alloc;
};

struct data_pair* find_create(char* data);

extern struct dyn_table freq_words;
extern long comps;
#endif
```

.c file

```
#include <string.h>
#include <stdlib.h>
#include "hash2.h"

#define START 1
#define PRIME 999999137

/*Abe Karplus 12-30-2008*/

struct dyn_table freq_words = {0, 0, 0};
long comps = 0;

void rehash();
unsigned long hash1(char* str);

struct data_pair* find_create(char* data) /*15*/
{
```

```
            int i;
        unsigned long at;
            struct data_pair* cur_ptr;

            /*printf("DEBUG: Looking for %s.\n", data);*/
            if(freq_words.n_alloc)
            {
                        at = hash1(data) % freq_words.n_alloc;
                        for(cur_ptr = freq_words.words[at]; cur_ptr; cur_ptr
= cur_ptr -> next)
                        {
                                    if(comps++, !strcasecmp(cur_ptr -> word,
data))
                                    {
                                                /*printf("DEBUG: Found
%s.\n", data);*/
                                                return (cur_ptr);
                                    }
                        }
            }
            if(freq_words.n_used >= freq_words.n_alloc)
            {
                        rehash();
                        at = hash1(data) % freq_words.n_alloc; /*because at
doesn't work after rehash*/
            }
            freq_words.n_used++;
            /*printf("DEBUG: at == %d.\n", at);*/
            cur_ptr = calloc(1, sizeof(struct data_pair));
            cur_ptr -> next = freq_words.words[at];
            freq_words.words[at] = cur_ptr;
            cur_ptr -> word = calloc(strlen(data) + 1, 1);
            strcpy(cur_ptr -> word, data);
            /*printf("DEBUG: About to return %s.\n", data);*/
            return (cur_ptr);
}

unsigned long hash1(char* str)
{
            unsigned long val;

            for(val = START; *str != 0; str++)
            {
                        val = (val + tolower(*str)) * PRIME;
            }
            return(val);
```

```
}

void rehash()
{
            int i;
            unsigned long hashed;
            struct data_pair* current;
            struct data_pair* save;
            struct dyn_table old_table;

            old_table = freq_words;
            freq_words.n_alloc *= 2;
            freq_words.n_alloc += 1097;
            freq_words.words = calloc(freq_words.n_alloc, sizeof(struct
data_pair*));
            for(i = 0; i < old_table.n_alloc; i++)
            {
                        for(current = old_table.words[i]; current; current =
save)
                        {
                                    save = current -> next;
                                    hashed = hash1(current -> word) %
freq_words.n_alloc;
                                    current -> next =
freq_words.words[hashed];
                                    freq_words.words[hashed] = current;
                        }
            }
            free(old_table.words);
}
```

main program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "readword.h"
#include "hash2.h"

/*Abe Karplus 12-30-2008*/

struct dyn_array
{
            char** words;
            int n_used;
```

```
            int n_alloc;
};

double gettime();

main()
{
            char* word;
            struct dyn_array all_words;
            int i;
            struct data_pair* found;
            double old_time;
            double new_time;

            all_words.n_alloc = 5;
            all_words.words = calloc(all_words.n_alloc, sizeof(char*));
            all_words.n_used = 0;
            while(1)
            {
                        word = readword();
                        if(word == 0)
                                    break;
                        if(all_words.n_used >= all_words.n_alloc)
                        {
                                    all_words.n_alloc *= 2;
                                    all_words.words =
realloc(all_words.words, all_words.n_alloc * sizeof(char*));
                        }
                        all_words.words[all_words.n_used++] = word;
            }
            printf("%d\t", all_words.n_used); /*words*/
            old_time = gettime();
            for(i = 0; i < all_words.n_used; i++)
            {
                        found = find_create(all_words.words[i]);
                        /*printf("DEBUG: found -> word = %s.\n", found ->
word);*/
                        found -> occur++;
            }
            new_time = gettime();
            printf("%d\t", freq_words.n_used); /*distinct*/
            /*
                        found = find_create("the");
                        printf("The word \"the\" occured %d times.\n", found
-> occur);
            */
```

```
            printf("%f\t", (new_time - old_time)); /*time*/
            printf("%ld\t", comps); /*strcmps*/
            printf("%u\n", sizeof(struct data_pair*) * freq_words.n_alloc +
sizeof(struct data_pair) * freq_words.n_used); /*memuse*/
            return(0);
}

double gettime()
{
            struct rusage elap;

            getrusage(0, &elap);
            return((elap.ru_utime.tv_sec)+1.e-06*(elap.ru_utime.tv_usec));
}
```

Appendix 7: Cellar

.h file

```
/*Abe Karplus 1-9-2009*/

/*hash3.h*/

#ifndef HASH3_H
#define HASH3_H
struct dyn_house
{
            struct data_pair* every;
            int n_used;
            int n_alloc;
            struct data_pair* try_here;
            unsigned mod_by;
};
struct data_pair
{
            char* word;
            int occur;
            struct data_pair* follow;
};

struct data_pair* find_create(char* data);

extern struct dyn_house freq_words;
extern long comps;
#endif
```

.c file

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "hash3.h"

/*Abe Karplus 1-9-2009*/

#define START 1
#define PRIME 999999137

struct dyn_house freq_words = {0, 0, 0, 0, 0};
long comps = 0;

unsigned long hash1(char* str);
void rehash();
```

```
struct data_pair* find_create(char* data) /* 16 */
{
            struct data_pair* next;
            struct data_pair* prev;

            /* printf("DEBUG: Looking for %s.\n", data); */
            /* fflush(stdout); */
            if(!freq_words.n_alloc) rehash();
            /* printf("DEBUG: hash1(data)=%lu freq_words.mod_by=%u.\n",
hash1(data),freq_words.mod_by); */
            /* fflush(stdout); */
            for(next = prev = &freq_words.every[(hash1(data) %
freq_words.mod_by)]; next; prev = next, next = prev -> follow)
            {
                        if(!next -> word)
                        {
                                    freq_words.n_used++;
                                    next -> word = calloc(strlen(data) + 1,
1);

                                    strcpy(next -> word, data);
                                    return(next);
                        }
                        comps++;
                        if(!strcasecmp(data, next -> word))
                        {
                                    /* printf("DEBUG: Found %s.\n", data); */
                                    return(next);
                        }
            }
            if(freq_words.n_used >= freq_words.n_alloc)
            {
                        rehash();
                        return(find_create(data));
            }
            freq_words.n_used++;
            /* printf("DEBUG: entering try_here loop, n_used=%d
n_alloc=%d\n", freq_words.n_used, freq_words.n_alloc); */
            /* fflush(stdout); */

            for(; freq_words.try_here -> word; freq_words.try_here--);
            prev -> follow = freq_words.try_here--;
            prev -> follow -> word = calloc(strlen(data) + 1, 1);
            strcpy(prev -> follow -> word, data);
            /* printf("DEBUG: About to return %s.\n", prev -> follow ->
word); */
```

```
            return(prev -> follow);
}

struct data_pair* find_create_nocpy(char* data)
{
            struct data_pair* next;
            struct data_pair* prev;

            if(!freq_words.n_alloc) rehash();
            for(next = prev = &freq_words.every[(hash1(data) %
freq_words.mod_by)]; next; prev = next, next = prev -> follow)
            {
                        if(!next -> word)
                        {
                                    freq_words.n_used++;
                                    next -> word = data;
                                    return(next);
                        }
            }
            freq_words.n_used++;
            for(; freq_words.try_here -> word; freq_words.try_here--);
            prev -> follow = freq_words.try_here--;
            prev -> follow -> word = data;
            return(prev -> follow);
}

void rehash()
{
            int i;
            struct dyn_house old_table;

            /*printf("DEBUG: starting rehash\n");*/
            /*fflush(stdout);*/
            old_table = freq_words;
            freq_words.n_alloc *= 2;
            freq_words.n_alloc += 1097;
            freq_words.every = calloc(freq_words.n_alloc, sizeof(struct
data_pair));
            freq_words.n_used = 0;
            /* printf("DEBUG: new calloc done\n"); */
            /* fflush(stdout); */

            freq_words.try_here = &freq_words.every[freq_words.n_alloc - 1];
            freq_words.mod_by = (76 * freq_words.n_alloc) / 100;
            /* printf("DEBUG: before for loop in rehash\n"); */
            /* fflush(stdout); */
```

```
            if (old_table.every)
            {           for(i = 0; i < old_table.n_alloc; i++)
                        {
                                    if(old_table.every[i].word != 0)
                                    {

            find_create_nocpy(old_table.every[i].word) -> occur =
old_table.every[i].occur;
                                                /* printf("DEBUG: In for loop
of rehash. Word is %s\n", old_table.every[i].word); */
                                    }
                        }
                        free(old_table.every);
                        /*printf("DEBUG: old n_used=%d, new n_used=%d\n",
old_table.n_used, freq_words.n_used);*/
                        /*fflush(stdout);*/
            }
}

unsigned long hash1(char* str)
{
            unsigned long val;

            for(val = START; *str != 0; str++)
            {
                        val = (val + tolower(*str)) * PRIME;
            }
            return(val);
}
```

main program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include "readword.h"
#include "hash3.h"

/*Abe Karplus 1-8-2009*/

struct dyn_array
{
            char** words;
            int n_used;
```

```
            int n_alloc;
};

double gettime();

main()
{
            char* word;
            struct dyn_array all_words;
            int i;
            struct data_pair* found;
            double old_time;
            double new_time;

            all_words.n_alloc = 5;
            all_words.words = calloc(all_words.n_alloc, sizeof(char*));
            all_words.n_used = 0;
            while(1)
            {
                        word = readword();
                        if(word == 0)
                                    break;
                        if(all_words.n_used >= all_words.n_alloc)
                        {
                                    all_words.n_alloc *= 2;
                                    all_words.words =
realloc(all_words.words, all_words.n_alloc * sizeof(char*));
                        }
                        all_words.words[all_words.n_used++] = word;
            }
            printf("%d\t", all_words.n_used); /*words*/
            fflush(stdout);
            old_time = gettime();
            for(i = 0; i < all_words.n_used; i++)
            {
                        found = find_create(all_words.words[i]);
                        /* printf("DEBUG: found -> word = %s.\n", found ->
word); */
                        /* fflush(stdout); */
                        found -> occur++;
            }
            new_time = gettime();
            printf("%d\t", freq_words.n_used); /*distinct*/
            /*
                        found = find_create("the");
                        printf("The word \"the\" occured %d times.\n", found
```

```
-> occur);
            */
            printf("%f\t", (new_time - old_time)); /*time*/
            printf("%ld\t", comps); /*strcmps*/
            printf("%u\n", freq_words.n_alloc * sizeof(struct data_pair)); /
*memuse*/
            return(0);
}

double gettime()
{
            struct rusage elap;

            getrusage(0, &elap);
            return((elap.ru_utime.tv_sec)+1.e-06*(elap.ru_utime.tv_usec));
}
```