CSE 214 - Data Structures
Fall 1997
Program #4

Due: Monday, November 17, 1997

This semester we have seen a variety of different data structures for retrieval by key -- sequential search, sorting plus binary search, and hashing. While the analysis of algorithms provides considerable insight into their behavior, empirical studies are important to choose between algorithms of the same asymptotic complexity or select the best algorithm for small problems.

For this assignment, you are to implement several of the algorithms we have discussed in class and experiment with them to determine their performance. The algorithms you will implement will be:

(1) Quicksort-insertion sort

(2) Heapsort

(3) Binary search

(4) Hashing via chaining

To provide test data, the file ~cse214/program4/words contains the entire Unix dictionary, arranged in a random order, comprising almost 25,000 words. The words are given, one per line, and each word contains at most 15 characters. It is best to represent each word as a TEXT variable. A suitable hash function for a word is:

$$\sum_{i=1}^{m}(ord(word[3i-2]) + ord(word[3i-1]) + ord(word[3i])) \times 26^{i-1} \quad (mod\ tablesize)$$

where m is the number of characters we want to hash on. It is a constant, for example 1 to 5.

The results of your investigations will be written up in a three page paper. Your report will answer the following questions:

(1) What is the optimal switch point where a hybrid Quicksort-Insertion sort becomes fastest? This issue is discussed within Sedgewick's chapter on Quicksort.

(2) How does the run time of Quicksort compare to Heapsort? Which is faster on unsorted data? Sorted data?

(3) How does the value of m in the hash function affect performance? If $m = 1$, it is hashing on the three characters only. If $m = 5$, it would hash on all characters, including padded blanks. Maybe an intermediate value is best? What happens when you hash only on the first character?

(4) How does the size of the hash table effect the running time? Try hash tables of 1000, 5000, 20000, 25000, and 50000 buckets. Does making the hash table a prime number make a difference?

(5) How does the best hashing scheme compare with binary search and the best sorting algorithm, for both preprocessing (ie. building the hash table or sorting) and query times?

To measure the performance, your program can either explicitly count the number of element comparisons made and/or be broken into units which the unix program *time* can be applied to. To compute the average search cost, your program should search for each word in the data file, count the total time or comparisons it takes, and divide by the number of words. The best way to do this probably is to *close* the input file and read it again.

In addition the paper analyzing your results, you must handin the programs and a printout of a short run to show what your programs do.

Challenge Problems

Each assignment will contain one or more extra credit problems for students who want a challenge, or to pursue the subject in greater depth. NOTE THAT THESE PROBLEMS ARE NOT REQUIRED. For this assignment, you are encouraged to conduct experiments of your own choosing, but here are some ideas.

(1) What effect does the quality of the hash function have? Can you find a better hash function?

(2) Implement hashing by open addressing. How does it compare to chaining for similar-sized hash tables? (remember to factor in the costs of pointers in measuring similar size)

Rules of the Game

(1) This program is not conceptually difficult, in my opinion, since the algorithms are completely described and at least partially implemented in the text. However, it contains several different parts which require intimately understanding distinct algorithms, so it may require a significant amount of time to complete.

(2) You may copy to your heart's content from the texts, but not from other students. Copying functions from the book will be a frustrating experience if you do not understand the algorithms - trust me.

(3) Since the algorithms required for this project are all fair game for the midterm, use this programming assignment as an excuse to study the topics you do not understand well.

(4) Like a simulation, it is important to confirm that your program is working correctly before trusting the results. Thus it is important to build in a 'debug' mode so you (and we) can verify that your program is working correctly. You will want to debug your program on a file with only 50 words, which can be obtained with the command *head -50 <~cse214/program4/words >testdata*.

(5) There is absolutely no reason to copy the full words file into your directory, since it will always be there to be read. %unless you want to prevent other students %from working.

(6) Do not run your programs on the entire data file until you are 100% sure your program works correctly, since it is likely to run slowly. If your program cannot complete an experiment in two minutes or less, there is no shame in making a smaller data file using *head*. Just describe what you did in your report.

(7) Your report should honestly describe your results. There are no right or wrong results, since everyone's programs will differ. However, if any of your timings are very surprising, figure out why - you may have a bug in your implementation.

(8) Your report must be typed and printed using a text formatting program. You may use whatever program you have access to, including Microsoft Word or HTML (don't post the result). If you prefer, latex, my favorite UNIX text formatting program is invoked *latex filename*, and a sample latex file can be obtained from *~cse214/program4/sample.tex*. Further documentation can be found in the book 'Latex: a document preparation system' by Leslie Lamport, on reserve in the CS Library. *You do not have to use latex if you have access to some other text processing system.*

(9) Don't get too fancy with formatting. You can waste an incredible amount of time making documents pretty - don't for the purposes of this class. This is *especially* true if you use latex.

(10) One program you should definitely know about is the spelling checker *ispell*, which you can learn about by using *man*. Spelling checkers are a painless and useful tool. To get you in the habit of using one, the TA's have been instructed to deduct 20% from your grade for any obvious spelling errors.

(11) Unix time gives somewhat unpredicatable results. If you can find a better way to do it, go ahead.

(12) Don't forget to write out and sign the pledge - "On my honor as a student I have neither given nor received aid on this programrq. Your paper will not be graded without it.