CSE 326
Autumn 2005
Assignment 8
Due Friday 12/2/05

For all algorithm and data structure design problems please provide elegant pseudocode and an adequate explanation of your methods. If is often helpful to include small examples demonstrating the method. Put your name at the top of each sheet of paper that you turn in.

1. In this problem we see how to use hashing to do fast string search. Suppose we have a string $A = a_1 a_2 \cdots a_m$ that we would like to find the first occurrence of in a longer target string $T = t_1 t_2 \cdots t_n$. Assume that we use a large prime p for hashing strings as described on slide 7 of lecture 15 is given by the function h. The idea is to first compute $h(A)$ then compute $h(t_i \cdots t_{i+m-1})$ for $i = 1, 2, 3, \ldots$ until this value equals $h(A)$. The string $t_i \cdots t_{i+m-1}$ can then be checked to see if it equals A. If so, we're done, if not we have a false positive and we continue the search.

   (a) Show that $h(t_{i+1} \cdots t_{i+m})$ can be computed in constant time given $h(t_i \cdots t_{i+m-1})$.

   (b) Show that the time to do the search is $O(m + n)$ time plus the time to check false positives.

   (c) Compute the probability of a false positive as a function of p.

2. Consider the following possible implementations of a spell checking program that uses hash tables. For each, assume that: the dictionary contains 10,000 words, the average length of a word is 7 characters, a pointer requires 4 bytes, a character string of length k requires k+1 bytes, and there are 8 bits to a byte. For quadratic probing, the size of the hash table must be a prime number; for the the other methods, it need not be prime. A table of prime numbers can be found at http://primes.utm.edu/lists/small/10000.txt.

   (A) An open-addressing hash table using quadratic probing and a load factor of 0.5.

   (B) A separate-chaining hash table using a load factor of 1.0.

   (C) A new strategy, called "approximate hashing", that works as follows. For each entry in the hash table we keep a single bit value, that is set if some key has been inserted that hashes to that location. We do not store the key itself! To do a find, we do a single probe and see if the corresponding bit is set; if so, we return "yes", otherwise, return "no". We create such a hash table with a load factor of (approximately) 0.10, and insert the contents of the dictionary.

Answer the following questions:

(a) How much memory is required is each implementation (A), (B), and (C) in bytes? Make clear how you arrive at your answer.

(b) What is the probability that (C) will flag as misspelled a word that is actually in the dictionary?

(c) What is the probability that (C) will fail to flag as misspelled a word that is not in the dictionary?

(d) How big a hash table (in bytes) would (C) require so that the probability of failing to flag a misspelled word is no more than 1%?