S-Q 1997 - Assignment 4

Due 07/24/97 at 6:00pm

Last Date Accepted: 07/28/97 at 6:00pm

1 Getting Prepared

1. Copy over the files for this assignment. Copy all of the files from ~`libsq/assts/asst4` into your directory ~`/Q/asst4` (which should have been created when you ran `q-setup` for assignment 1).

2 Exercises (100 points)

All work should be done in directory ~`/Q/asst4`. The `Makefile` provided includes rules to build all the programs in this assignment; you shouldn't need to change it at all.

1. (20 points) Devise algorithms for writing and reading 2-3 trees in such a way that the reconstructed 2-3 tree has exactly the same structure as the original. (You may assume that the 2-3 trees are inorder search trees, if this is useful.) Describe how your algorithms could be extended to work with B-trees, and any pitfalls that might arise when doing so. Give the big-Oh of your write and read algorithms, in terms of how many keys are present.

   You do not need to implement your algorithms, but your description must be detailed enough so that it could be converted into code quite easily by anyone with an existing library of 2-3 tree code.

   Write your answers to this question in a file named `asst4.txt`. If you prefer to write out your algorithms by hand (or using a word processor), simply note this in your `asst4.txt` and turn in your hardcopy in class.

2. (10 points) Course Book, Monte Carlo chapter, problem 1. Write your answers to this question in a file named `asst4.txt`. If you prefer to write out your algorithms by hand (or using a word processor), simply note this in your `asst4.txt` and turn in your hardcopy in class.

3. (40 points) This problem explores the relative speed of separate chaining, open addressing, and the effectiveness of various hash functions.

   The file `dict.c` is a main driver for a program that reads the contents of a 25,000-word dictionary into a list, creates several hash tables based on this data, and computes some metrics for each table.

   (a) Complete the functions in file `open_hash.c`, to implement functions to insert, find, and delete keys from a hash table using open addressing with linear probing. (You may use a more complicated probing strategy, if you wish, but I suggest that you begin with something simple.)

   (b) Complete the functions in file `measure.c`, so that you can gather some basic statistics about the effectiveness of these hashing functions for this set of keys at several load factors. See `measure.c` for more information.

   (c) Devise at least two new hash functions, and add them to the functions in file `hash.c`. Modify the main function of `dict.c` to test these hashing functions. Comment your code to briefly explain how you devised these hashing functions, and whether they appear to work well (or if they seem plausible but do not work well).

   Note– your goal is *not* to invent the world's greatest hashing function, but simply to see if you can exploit any of the characteristics of this set of keys to devise a better hashing function– or to explain why a hash function that looks good is actually not appropriate for this data.

4. (30 points)

   Write your answers to this question in a file named `asst4.txt`. If you prefer to write out your algorithms by hand (or using a word processor), simply note this in your `asst4.txt` and turn in your hardcopy in class.

   (a) Given a (potentially) unfair coin, devise a procedure to simulate a fair coin toss. You may assume that even though the coin may be unfair, each toss is independent of the others.

   Your procedure may require tossing the unfair coin several times. Be sure to give an estimate of how many times you need to toss the unfair coin. For the sake of this problem, you may assume that the probability of the unfair coin coming up heads is always $0.25 < p < 0.75$ (no need to worry about two-headed or two-tailed coins).

   Hint– the first solution that many people immediately think of is wrong, so double-check your logic!

   (b) Given a fair coin, devise a procedure for simulating a coin toss of a coin that has probability p of coming up heads (and $1 - p$ of tails), where p is a positive rational number (i.e. the ratio of two positive integers).

   Once again, your procedure may require multiple coin tosses. Be sure to give an estimate of how many times you will need to toss the fair coin.

3 Graduate Problem (30 points)

Implement `skipListInsert` and `skipListLookup` to complete the implementation of `skiplist.c`. You may assume that the maximum level of the list is defined as `MAX_LEVELS` (which is defined in `skiplist.c`). Once again, be sure to document your code; a reasonable approach is worth many points, even if the code does not function perfectly. Skip lists are described in Weiss, section 10.4. You may find the `FairCoin` function (defined in `random.c`) to be useful.

Your insertion routines should be efficient– by keeping track of the last node visited at each level, you will know exactly which nodes you need to update with pointers to new node. Note that *only the most recently visited node at each level* (and only nodes to the left of the key) need to be updated during an insertion– therefore it is easy to keep track of all the nodes that have to be updated. Once again, you will find this much easier to understand if you draw a picture of what has to happen at each step.

4 Submit Your Work

1. Use the `submit` program to hand in your work. From your directory ~/Q, run the `submit` program. The name of the course is **cscisq**, and the name of the project is **asst4**, and the name of the file to submit is also **asst4**. (See the UNIX tutorial for more info about `submit`.)

2. Print out a copy of any of the files you created or modified as part of the assignment, and hand them in. This includes any hand-drawn proofs or drawings for the written parts of the assignment.