Dr A Norman.

ENGINEERING TRIPOS PART IIA

Monday 28 April 2008 CRIB 2.30 to 4

Module 3I1

DATA STRUCTURES AND ALGORITHMS CRIB

*Answer **all** of Section A (which consists of short questions), and **two** questions from Section B.*

All questions carry the same number of marks.

*The **approximate** percentage of marks allocated to each part of a question is indicated in the right margin.*

STATIONERY REQUIREMENTS	SPECIAL REQUIREMENTS
Single-sided script paper	none

You may not start to read the questions printed on the subsequent pages of this question paper until instructed that you may do so by the Invigilator

SECTION A

Answer all parts of this question. The question in this section will be marked out of the same total as each question in section B, and each part carries the same weight.

1 (a) A simple hash table of size N has k values stored in it. You are provided with a function f that could take one of the values as stored and map it onto an integer. Using big-O or big-Theta notation as relevant explain how long it would take to discover whether there are two values, say x and y in the table such that $f(x) = f(y)$. [10%]

The key problem here is that it is liable to take N steps to scan the hash table and find the k items. If $N >> k$ this is a substantial overhead. A simple way to find matches would then be pairwais comparisons at N^2. set of values was extracted from the hash table then sorted ($N^2 + k \log k$) duplicates would become adjacent. Or of the values of fx were hashed the total clsot could be expected to be linear in N. A nice explanation of any of the above will suffice

(b) It is well known that the ratio between the maximum and minimum possible depth for a 2-3-4-tree holding N items is close to 2. What is the corresponding ratio for a form of B-tree that where each node can have a fan-out somewhere between 256 and 512? [10%]

The is $\log_{512} N / \log_{256} N$ which is liable to be around $9/8$. Ie the ratio is a lot closer than the factor of 2 for 2-3-4 trees.

(c) A set of items are held in a linked list. The list is (already) sorted, and there are n items. What are the best, worst and average costs of looking up an item that is already present in the list, and what are the costs of verifying that an item is not present? [10%]

Already present: best:1, worst:N, average:N/2. When the item is not present the costs are the same!

(d) Give and explain an example of an algorithms where the use of randomness provides a benefit. [10%]

I will list a couple here even though the answer only needs one. Quicksort there is a benefit in using a randomly selected pivot in that that guarantees average costs

(cont.

regardless of input data. Miller-Rabin testing to see if a number os prime relies on having independent random probes.

(e) Explain precisely the meaning of the big-O and big-Theta notations. [10%]

There exist N and K_1, K_2 such that when $n > N$ we have $K_1 g(n) < f(n) < K_2 g(n)$. That is for big-Theta, and big-O has just one half of it.

(f) Construct and explain a small example sequence of allocate and free operations where best fit would lead to noticably less fragmentation than first fit. [10%]

We want to leave two gaps such that the second is a perfet fit and the first is too big, leaving a messy left-over bit. So how about allocate(10), allocate(1), allocate(7), allocate(1), free the 10 and the 7 and then ask for another 7. First fit will split the 10 up leaving a fragment of size 3. Best fit fills in the space previously used by a 7.

(g) Explain what is meant by the amortised cost of operations on a data-structure. Contrast it with other ways of predicting performance, showing one case where it is useful and one where it is not. [10%]

This applies when a sequence of operations are performed on the same data-structure. It is the average cost per operation but averaged over long sequences, Its purpose is to "allow" for occasional expensive tidy-up steps. A case where it is good is eg Splay Trees where it gives a guarantee of good long-term cost. A case where it would be bad would be almost any real-time application where worst case cost per individual operation matters.

(h) In bad cases Quicksort can be very much more costly than Heapsort. Why, then, is it still very often used? [10%]

The bad cases for quicksort are fairly rare, and with a slightly careful implementation they will not include the especially common cases of data that is almost sorted already. And in average and good cases Quicksort is typically noticably faster, so across a series of uses it is liable to win even if a bad case does happen occasionally.

(i) What expectation motivates the concept of Ephemeral Garbage Collection? Describe a scenario where it will not deliver benefits. [10%]

Ephemeral Garbage collection works on the basis (observed very well in most real life) that data tends to either die young or live for quite a while. So if works to recycle the

(TURN OVER for continuation of Question 1

data that dies young especially smoothly. So to make it misbehave how about giving all data the same life-span.

(j) Explain how to reinstate the heap property if the one of the values in the heap is changed. What is the cost? Does it matter whether the value is increased or decreased?

[10%]

If an item decreases (in a heap with small at the top) it can only move up, and you only need to look at "triangles" from it upwards to the root. If it increases it may move down, but then you only need to look at "triangles" drom where it was down the the bottom of the heap. In either case the cost os bounded by $\log(heapsize)$.

SECTION B

*Attempt **two** questions from this section. Each question has the same weight in marks.*

2 A monitoring device needs to transmit an unending series of status messages to home base. Each message is either an X or a Y, and there is strong reason to expect both that all messages are independent and that in the long term X will occur twice as often as Y. Information is to be transmitted as a stream of bits, and data compression is to be used to reduce the amount of transmission necessary.

(a) Why would one normally expect Arthmetic Coding to be better than Huffman Coding in this situation? [10%]

Because Huffman must send a whole number of bits per symbol, and for symbols that are already bits it gives no compression at all!

(b) The design engineer responsible for this project is really keen on Huffman Coding, and so wants to use it. The engineer arranges to group the status messages into clumps of three, eg such as XYX. Huffman Coding is now applied using as its alphabet all the possible strings of 3 original symbols. Does this improve things? If so predict the extent to which it compresses data, and if not explain why. Would using groups of more than three change your analysis, so for instance would using clumps of 20 or 50 letters make sense? [40%]

List the eight 3-bit sequences, and note that they have associated probabilities/frequencies from 1/27 to 8/27. Build the Huffman tree and evaluate the result and you find that (because you now have an alphabet with some non-uniformity n frequencies) Huffman helps a bit.

If you used longer runs you woudl go further in avoiding the bit-granularity of Huffman but note there are around a million 20-bit runs (so managable but a bit big) while there are too many 50-bit runs for any sense at all.

(c) A second engineer looks at the problem and adapts Lempel Zif to the case where its initial alphabet consists of just the two letters X and Y. Explain how Lempel Zif

(TURN OVER for continuation of Question 2

works, illustrating its behaviour using the string XYXXXXXXYXYXYXXX. Explain whether it is liable to give reasonable compression. [25%]

Need to run through showing what width of output symbol is generated each time and how the dictionary build up. The initial X and Y will be sent plain, but then XY will be in the dictionary and the next few syms will need to be sent as 2-bit codes, then 3-bit codes etc. Expect dictionary to tend to build up strings with more X than Y in and on that basis it has scope to do good.

(d) Both coding and decoding Lempel Zif are made more complicated by the need to update internal tables as you go. To remove this overhead one could try running the full process over an arbitrary sample of a list of X/Y symbols of length say 8000, then freezing the coding tables and using them as static tables permenantly built into the monitoring device. Explain roughly what collection of entries you would expect to find in the table used by Lempel Zif in such a case, and discuss whether you could propose hand designed fixed coding that would do well. [25%]

Recall that Huffman above used all strings of constant length, and exploited their varied frequencies. Well here I think we should try to set up a catalogue of strings so that the list is comprehensive of all ones down to some cut-off probability. The result will be that there are only short strings made up mostly of Y, but longer ones mostly of X. I think it is in many respects a sort of dual of the Huffman result.

3 A hash table is built using an array that has N slots, and each slot has a sub-table that stores all information that hashes to that location. If just 2 values are put into the table there is only one way a collision can occur, and so (if the hash functions used are good) there is a chance of $1/N$ of a collision, and putting that another way the expected number of pairwise collisions is $1/N$.

(a) If k items are inserted how many ways are there in which collisions can arise? Hence predict the expected number of collisions. [15%]

$k(k-1)/2$ is a sufficient answer here

(b) Based on the result above, suggest how N must grow as a function of k so that you have a reasonable chance, say around $\frac{1}{2}$, of having no collisions at all. [25%]

When $N >= k^2$ you have a decent chance of having no collisions. For N smaller than that the chances of that sort of good fortune decrease rather rapidly.

(c) Explain what a Universal Hash function is, especially as it can be applied to constructing hash values for strings. [25%]

If the string is c1, c2, c3 ... map each character onto a numeric code (ie use its character coding, so you have n1, n2, ... Then have a sequence of arbitrary (perhaps randonly generated) constants k1, k2, ... and compute n1k1 + n2 k2 + ...

The idea of this is that if you have a set of kets (even if the keys are not random) the output results have a probability distribution determined by the statistics of the weights k. So you get to be safe from having any pathological input date bu way of keys.

(d) A variant on the idea of hashing is set up as follows, using two levels of hash table:

(i) When a known set of k keys are to be used a master hash table is set up with a size that is close to k. Universal hashing is used, and half a dozen different sets of numbers are used with the universal hashing method, with the one of these that gives the most even distribution of keys to table entries being the one that is used;

(ii) Each entry in the table may need to store several values (when the first hash table suffered from collisions). But typically not too many. Each entry then contains a set of parameters for a second universal hash function,

(TURN OVER for continuation of Question 3

together with a small hash table based on this. By choice of the hash function and table size it is arranged that this second hash table does not have any collisions at all.

Discuss the performance and limitations of this scheme. Can you explain why the designer of the method allowed each sub-table to define its own universal hash function rather than just selecting a single one for use with all of them? [35%]

This is a scheme for Perfect Hashing. It always costs exactly two hash computations to access anything, with no need for collision worry or uncertainty or cost variance. But it is only applicable if the set of keys is fixed and known in advance. This may be true for detecting keywords or for data frozen into an un-updatable program.

Each sub table is (by an earlier part of this question) liable to have a roughly 50:50 chance of no collisions if you use a given hash function with it. That makes it easy to find a set of universal hashing constants that work. But only around in 1 in 2^k hash functions would be good for all the (about) k sub-tables, and for other than rather small k that can mean there are not enough universal hash functions available.

Note that this procedure is documented in Cormen et al and that any student who has enthusiastically read distinctly beyond the course will thereby be better prepared than one who has not. I am not worried about this for the exam, but when this crib is viewed by students in any future year it may help them to know that they can find a comprehensive discussion of the issue in the standard textbook.

4 (a) Explain the structure of a Binary Search tree. If such a tree contains n nodes what are the best and worst cases for its height? Explain how you would look up a key to find it in such a tree. [25%]

Height can be between $\log n$ *and* n. *Look up by checking against a node and going found, left or right.*

(b) Given a tree and an item in it show how to find the key that has the next higher value. What are the best and worst-case costs involved? If you start from a key a and finding its successor b has almost worst case cost is it possible for finding the successor of b to be a bad case as well? [25%]

If node has a right child then successor is smallest item in that. Ie first node with no left child found by traversing left in that. If node does not have a right child it must be maximal in some sub-tree, so its succssor is found by going up the tree so long as that is unwinding a right edge, then up one level more. The cheapest case will be a node with a right child that is a leaf. The worst case has cost that is the height of the tree. If you are the rightmost node in the left sub-tree then your successor is the root node and was a long way up, and its successor could be a leaf a long way down again.

(c) Given a binary seach tree and a key that is present in it explain an efficient procedure that updates the tree so as to delete the given key from it. Indicate expected and worst-case costs for your procedure. [25%]

If item to delete has no children or just one then deletion is easy. If it has 2 children then swap its value with its successor, and note that in this case its successor could at worst have had one child, so now the node is easy to delete. Cost bounded by tree height, whihc is expected to be $\log n$.

(d) Given a binary search tree and a key that is present in it describe a procedure that rearranges the tree so that the specified key is in the root of the updated tree. Is this always possible? Explain the costs involved. [25%]

Explain a rotation that preserves the ordering propery of the tree but moves an item upone level. By applying it repeatedly the problem can be solved in cost proportional to the duatance upwards that the node must move. There are more complicated dual rotations that could move you up two levels at a time that woudl save time but leave assymptotic estimates unaltered.

END OF PAPER