Crib prepared by A C Norman
ENGINEERING TRIPOS PART IIA

Wednesday 9 May 2012 9:00 – 10:30

Module 3I1

DATA STRUCTURES AND ALGORITHMSCRIB

*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.*

There are no attachments.

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

Version: 1

SECTION A

1 (a) Give one set of circumstances in which you might prefer Quicksort to Heapsort, and one in which you might prefer a simple selection sort to Mergesort. Briefly justify your answer . [10%]

If I was merely concerned with average (not worst case time) or if memory locality had a strong impact on performance then Quicksort will beat Heapsort. Although Mergesort has guaranteed $n\log(n)$ time costs and simple selection sort is quadratic, selection sort does not need any extra memory, while mergesort does – so unavailability of extra memory forces your hand.

(b) Lempel-Ziv style data compression involves detecting when a pair of symbols has arisen before and then replacing them with a single symbol. What technology would be liable to make maintaining the table of symbol-pairs efficient? [10%]

A hash table. And if you think that is an obvious answer to a straightforward question I note that the use of hash tables in this sort of context was subject to a patent, so it was not obvious to the patent examiner who accepted that!

(c) If you have a limited total amount of memory and your program will fill about 40% of it with active data, would you expect a stop-and-copy or a mark-and-sweep garbage collection strategy to be most suitable? [10%]

A stop-and-copy scheme has to split the raw memory into two half-spaces so if 40% of the whole is in use then 80% of a half space is full. And all that will need copying on each GC. So stop-and-copy will do rather badly, leaving mark-and-sweep the winner.

(d) A team of programmers have a graph and want a minimum spanning subtree. They are worried because the graph has some edges with negative weights. So they increase the weight on each edge by an amount W such that all weights become positive. They can then find a spanning subtree in the adjusted graph. Will it be minimal in the original graph? [10%]

In the graph has n vertices the MST will have $n-1$ edges and so in the adjusted tree will have $(n-1)W$ excess weight, but because the number of edges in the MST is known and fixed and definite the weight adjustment does not hurt the search, and you do find a MST for the original.

(e) Why is it generally said that sorting is an $n\log(n)$ process? Give and explain a case where sorting can be performed more rapidly than this. [10%]

If there are n items there are (in general) $n!$ possible orderings, and you need $\log_2(n!)$ bits of information to identify one. This is around $n\log(n)$. If you have a case with significantly less than $n!$ possible orderings (eg if the items in the list to be sorted contain very many repeats, or something about the order is already known) you can do better.

(f) When you create a splay tree by adding in all the items in ascending order the "tree" ends up totally imbalanced. If n items have been added it has depth n. You then wish to look up the item you had inserted first and find that the cost if severe. In the light of this worst case why would anybody ever use splay trees? [10%]

The key to understanding is the term "amortised". Splay trees guarantee that over a (long) sequence of operations the worst case cost averaged over the sequence is logarithmic. So in this case the linear cost of the costly lookup will be balanced against the fact that each of the inserts that preceded it had unit cost. And after the (costly) lookup the tree will have been rearranged in such a way that the next operation can not be as expensive.

(g) If algorithm A has cost $O(n^2)$ and algorithm B has cost $O(n^3)$ is it possible that algorithm B always runs faster in every practical case? Is it possible that algorithm B is always faster even in hypothetical or impractical cases? [10%]

Because big-O notation only specifies an upper bound it could be that as well as being $O(n^3)$ algorithm B was in fact linear and always beat algorithm A. However proper convention for the use of big-O notation would discourage people from characterising it with a non-sharp upper bound, so this issue should only arise when somebody is trying to be "clever". Also you could have $1000000n^2$ vs. $0.0001n^3$ where the higher growth rate method could win in all practical cases.

(h) Why can you expect that Arithmetic Coding can be capable of better data compression that use of Huffman Codes? [10%]

Huffman codes use a whole number of bits for each symbol in the original text, and this necessarily leads to a waste whenever the probability associated with a symbol is not suitably related to a power of two. One can informally estimate that Huffman may lose about half a bit per symbol to this. Arithmetic coding avoids artefacts of this sort and can

hence do better.

(i) A team of programmers have a graph and want to find the shortest path through in it between two particular vertices A and B. They are worried because the graph has some edges with negative weights. So they increase the weight on each edge by an amount W such that all weights become positive. They can then find a shortest path in the adjusted graph. Will it be minimal in the original graph? [10%]

No it might not be, mainly because the lengths of paths in the adjusted graph depend on the number of edges traversed, so in the adjusted graph there can be an undue penalty on paths that involve many short steps. This section is intended to mirror an earlier one about spanning subtrees.

(j) Why is the guaranteed linear cost median finding algorithm hardly ever used? What is the worst case cost of the Quicksort-based scheme that usually (but not always) delivers linear cost? [10%]

While linear, the elaborate algorithm (that involves grouping the initial data into fives etc) has a large constant of proportionality that tends to render it unattractive. The expected linear cost version will be a *lot* quicker (as well as easier to code) most of the time. However as with full Quicksort its worst case cost is quadratic. But since the course teaches that its average time is linear the quadratic worst case must be amazingly uncommon!

2 In an experiment aimed at generating a sequence of pseudo-random integers a programmer generates a sequence x_1, x_2 ... where each value is derived just from the previous one. In other words the basis of the generator is a function f such that $x_{i+i} = f(x_i)$.

Of course once any value recurs in the sequence everything after that will be a repeat. In evaluating different functions f it is important to detect when this happens and how long the repeating loop of values is. Note that a sequence need not return to its very first value – a loop can arise later on as in the example 1,2,3,4,5,6,4,5,6,4,5,6,....

(a) A simple algorithm for detecting loops stores all values in the sequence, and when a new item x_n is generated it will compare against everything, thus detecting a cycle at the very earliest possibly stage. Using big-O notation characterise the time and store costs of this in terms of when the sequence under test starts to repeat. [15%]

If the first repeat happens after n items this will have done about $n^2/2$ comparisons. This is $O(n^2)$.

(b) An alternative scheme inserts each value x_i into a hash table as it is generated. It is not know how long it will be before a cycle is detected, and so initially a hash table with 256 entries is used. If the table becomes more than 80% full it is abandoned and the entire process is re-started beginning from x_1 again but using a new hash table twice the size of the previous one. What are the costs in this case? You may assume that the hash tables behave well without an excessive number of collisions. [25%]

There are two components to the analysis involved here. The first is the cost of detecting loops using the hash-table method, the second is the consequence of re-starting on hash table overflow. When hash table overflow does not happen this involved n hash table operations, so has linear cost. The effect of restarting leads to a cost that grows like $2+4+8+16+\ldots$ where the final term in the series is the cost of the trial without hash overflow. This at worst doubles your total overall cost! So the total cost remains linear.

(c) As a third attempt the programmer uses the hash table scheme as above, and if the hash table becomes too full they abandon it and create a new empty hash table of twice the size. However unlike the previous version they do not go back and re-start the sequence from x_1, but merely continue checking values $x_k, x_{k+1}\ldots$ where x_k was the value under consideration when the hash table became full. Will this still even detect loops? If so how promptly and at what cost in time and memory? [25%]

This can fail to spot a loop promptly, but once the hash table can hold as many values as the length of the loop it will detect one. And it will identify the loop within a factor of 2 of the perfect moment, so if you are mostly concerned with the length of the repeating cycle this is still a good method and may be at least as fast as the previous one.

(d) What are the consequences for the hash-table-based methods if unfortunately the particular sequence used interacts badly with the hash function and there are worst-case hash collisions? [10%]

Costs may be $O(n^2)$ rather than linear.

(e) To avoid the need for hash functions and all that uncertainty, but building on the methods above, the programmer records x_2 and compares x_3 against it. Then they record x_4 and that is used as a basis for comparison with $x_5 \ldots x_7$. Doubling the index again, x_8 is then saved so that the sequence as far as x_{15} can be checked against it. In general at each stage a value of x_n is stored where n is the most recent power of two passed. This can be seen as related to the idea of doubling the size of the hash table that has previously been used, but now there is no hash table at all, merely a single saved value. In what circumstances can this final scheme detect cycles? In a case when it can, the first repeat occurs after N items in the sequence and the loop is then of length M. Find a bound on how long it may take the algorithm to notice the cycle. [25%]

Look up Pollard Rho for the inspiration of this! The scheme always detects loops. It spots them within a factor of two of the earliest possible moment. There is no extra storage required, and low computational overhead.

3 (a) Standard Red-Black trees can be viewed as an interpretation of 2-3-4 trees and they require just one extra bit in each (binary) node to mark that node as Red or Black. What is the greatest possible ratio between the height of the left and right subtrees of a Red-Black tree? What is the greatest depth of a Red-Black tree that has N items stored in it? [15%]

Viewed as a 2-3-4 tree the depth of all leaves is the same, But as a red-black tree some paths might have NO red nodes while others might have every other node on the path red. So there can be a factor of 2 difference in height. To make a tree as deep as possible consider a 2-3-4 tree that consists of only 2-nodes. Its height is $\log_n(N)$. Now to make it worse as a red-black tree make ONE path through it consists of 3-nodes. This only adds $\log(N)$ values to the tree which is unimportant, but it can cause that path to have alternating red-black nodes all the way down, so you get a worst depth of around $2\log_2(N)$.

(b) B-trees generalise 2-3-4 trees by allowing a node to store a larger number of values. Explain the structure of a B-tree where each node can have between 4 and 8 descendants, and explain both how existing keys can be searched for and new data added. Are there any special cases that arise for either very large or very small trees? [25%]

A 4-8 tree will have each node having at least 4 and at most 8 children, and hence storing from 3 to 7 keys that partition them. If that situation is adhered to strictly then the tree *must* have at least 3 keys in it, so it becomes impossible to have a tree with only 1 or 2. So one might reasonably make a special exemption and allow the root node to be a 2-8 node. Searching for a key is trivial. The key is not present in an empty tree. If the key is one of the values in a node it has been found. Otherwise continue searching the sub-tree based on comparisons with the keys in the node. To add a new key start by searching for it, and identity the final non-empty node inspected. If that is anything other than an 8-node it can be expanded. If it is an 8-node split it into a pair of 4-nodes and its parent needs to be expanded.

(c) A engineer decides to design a family of binary almost-balanced trees, starting from 4-8 trees using the same ideas that led to the derivation of Red-Black trees from 2-3-4 trees. Explain the structure that they will use to represent for each sort of node. Can they still use just two colours or will they require more? [20%]

The nodes can be stored as binary trees as follows: 4: ((. r .) B (. r .)),
5: (((. r .) r .) B (. r .)),

6: (((. r .) r .) B ((. r .) r .)),
7: (((. r .) r (. r .)) B ((. r .) r .)),
8: (((. r .) r (. r .)) B ((. r .) r (. r .))),
where (. B .) denotes a black node with offspring where the dots are, and (. r .) a red node. Observe that black marks the start of a new logical 4-8 node and it is still OK to have just two colours.

(d) Explain how to look up an item in the new style of tree. How will its worst-case compare with the worst case that can arise using standard Red-Black trees? [20%]

Search is exactly as for a standrad binary search tree and does not need to be aware of balancing, colour etc. In a standard red-black tree a traversing a 2-3-4 node involved either 1 or 2 binary nodes. Here a single 4-8 node corresponds to 2 or 3 binary nodes. Repeatint my previous worst case analysis, a 4-8 tree that is all 4 nodes has depth $\log_4(N)$, and adjusting the tree so that there is a path of 5-nodes gives max depth $3\log_4(N)$. This is better than standard red-black trees!

(e) The most complicated transformation on a 2-3-4 tree is when a 4-node has to be split into a pair of 2-nodes. Correspondingly on a 4-8 tree it will be when an 8-node is split into two 4-nodes. Compare the frequency with which these operations arise when each sort of tree is built with N items. [20%]

Consider merely splitting of leaf nodes. With a 2-3-4 tree the steps are 2:3, 3:4 and 4:2+2 so one could expect that one insert in 3 will involve a splitting operation. Cascade splits of internal nodes are less frequent so I will ignore them! With the new version I have 4:5, 5:6, 6:7, 7:8 and 8:4+4 so maybe only one insert in five leads to the need for a split. If I wanted to allow for the work splitting internal nodes I would sum a geometric progression (1/3 + 1/9 + 1/27 + ...) and (1/5 + 1/25 + ...) and still find the new scheme doing usefully fewer splitting operations.

4 A programmer wishes to implement a "best fit" storage allocation scheme. It must provide for allocating a new block of memory of requested size n, and for returning blocks once they are not needed. On allocation the smallest block of free memory large enough to fir n words in must be used. When a block is returned it must be consolidated with any free block that is adjacent to it.

(a) The programmer's plan starts with an idea that each block of free or in-use memory will be preceded and also followed by extra words. Each of these will contain the length of the block plus a flag saying whether it is in use. Why might this information useful? [10%]

So that when a block is being freed the size and status of its two neighbours are easy to find.

(b) All free blocks of memory will be such that in addition to the header and trailer words, there is space to hold two pointers, and these will be used to arrange free blocks into a binary search tree. The ordering used in this tree will be such that if two blocks have different lengths the shorter one will be considered smaller. If two blocks have the same length then they will be ordered based on their relative positions in memory. Using this search tree explain how to find a free block suitable either for direct use or for use after splitting to provide n words of memory for the user. Explain all tree operations you need in detail. [30%]

First let me imagine that I know how to search a binary tree for an exact match. Do that for the exact size of block required. If found I will want to use it. So I now need to remind myself of the algorithm for deleting a node from a binary search tree (identify next-highest key, exchange values...). If an exact fit is not possible I need to find a block that I can split. This must be big enough that the residue is a valid size, so I will search for a block of size $n+4$ words. If I find one I can use it. If not I know where it would have been and I use the "find next largest key" procedure starting from there – that gives me the block I will use. The overall cost is proportional to the depth of the binary tree. I have not documented tree insert, delete and find-next here since those are standard bookwork.

(c) Develop and present an algorithm that exploits the given structure to allow freeing of memory blocks, including the step of consolidating adjacent free blocks. Characterise its costs. Is allocation or freeing of memory more expensive or do they cost about the same? [25%]

If you have described allocation well the freeing memory is not too bad. For the block you are freeing check each neighbour (using the header words). If either is free you need to consolidate. So you need to delete the neighbour from the tree (look it up based on its size and address so you know where it is in the tree and can remove it). Note that you should not need to repeat the "free neighbour" check since things already recorded as free will already have consolidated. Form a new free block and insert it in the tree.

(d) What effects would arise if instead of using a simple binary search tree use was made of splay trees? [15%]

With regular binary search trees the trees could become ill-balanced and costs would then escalate. With splay trees the cost of every tree operation would be logarithmic (in an amortised sense). Furthermore with splay trees any blocks of memory that were allocated but then not freed would count as items not inspected, and so their presence would not (in the long term) impact costs. Which could be a very good thing!

(e) Make an informal comparison between the allocation scheme considered here and one based on a Buddy system, considering both time and space costs. [20%]

A (binary) buddy system forces each allocation to be a power of two in size, so it potentially "wastes" half your space. The scheme here has a two-word per block overhead and insists that every block is of size at least 2 words. So for usage when many very small blocks (eg just 1 or 2 words) are required buddy might win on this basis, but as larger block are used the scheme here will have significantly lower waste.

It is hard to predict fragmentation, but at least both schemes consolidate adjacent free blocks and take some steps intended to prevent gratuitous fragmentation. I do not believe one can judge what will happen in that respect without knowing a lot more about the pattern of use.

Wrt time, the Buddy system has a worst case allocate or free cost that is logarithmic in the heap size. But very often its cost is close to constant. The new scheme (with splay trees) has a worst case amortised time logarithmic in the number of block free (which is bounded by the heap size). In cases where splay-tree arrangements work well it will have a cost logarithmic in the number of blocks involved in heavy memory turn-over. I expect that mostly buddy will win, and the splay tree code is certainly going to be longer and messier.

END OF PAPER

Version: 1