Crib prepared by A C Norman

ENGINEERING TRIPOS PART IIA

Wednesday 5 May 2010 9:00 – 10:30

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

Version: final

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) The expected cost C of accessing an item in a hash table can be expressed as

$$C = \frac{1}{2}1 + \frac{1}{4}2 + \frac{1}{8}3 + \ldots$$

What is the value of this series? [10%]

It is $(1 + \frac{1}{2} + \frac{1}{4} + \ldots)^2/2$ and so is easily seen to have the value 2. This example series as the cost arose in the course not just when looking at hash table costs but also for the first part of Heapsort, so candidates have been shown applications of it at least twice!

(b) Two bad people are constructing data intended to give pain to your implementation of Larsen's Dynamic Hashing. One of them has found a set of keys so that when you apply any of your first series of hash functions (controlling which disc block will be used) only a small number of distinct hashes arise. The other has selected keys so that you only ever get a small number of distinct signature or priority values. Which will disrupt you more rapidly? [10%]

If the hash values are limited in effect you will only be using a small fraction of your big disc, and that will fill up rapidly in a way that can not be recovered from. If the signatures are limited you may still fit in quite a lot of data before any disc block becomes full.

(c) How do you expect the best, worst and typical costs of allocation under first-fit and best-fit to compare? [10%]

For first fit the best cost is when the first free block is big enough, so the cost there is O(1). The worst cost would be when the memory is fragmented and one needs to scan over many many fragments before finding a big enough block, so I might suggest O(memory size). Typical is very very dependent on the pattern of use, but one hopes to find a block at least reasonably soon.

Best fit could have a best case of O(1) if the first block considered was a perfect fit. Its worst case will be as for First Fit. Its typical case will be worse than First-Fit because merely finding a block that is big enough doe snot let it declare success. It must either find an exact fit or scan all free blocks to find the best one. So one could fear that typically it may search almost as much as its worst case. But "typical" is very very application sensitive, which is why the question used this general word rather than asking about an "average".

(d) The course covered 2-3-4-trees. One thing that it explained was that when necessary a 4-node could be split into two 2-nodes. One could perform that split every time a 4-node was about to arise, ending up with a tree consisting of only 2-nodes and 3-nodes. Using big-O notation explain what the cost of looking up data and inserting new data is liable to be. You do not need to give full details of algorithms — just enough overview to make it clear that you could construct them if necessary. [10%]

The tree has height between $\log_2(n)$ and $\log_3(n)$ and so lookup is clearly log-cost. When you insert you can always add a new leaf in a way that at worst *temporarily* introduces a 4-node. If you do then you split that and need to add an item to its parent. This mat again temporarily become a 4-node and need splitting. You at worst do that all the way to the top and the overall cost of logarithmic.

(e) It is commonly said that Huffman codes can waste half a bit for each symbol transmitted. Why is this? Give a small example to illustrate it and name a compression scheme that does not suffer from this waste. [10%]

The ideal length for a code is a logarithm of its probability, but actual symbols are always encoded as a whole number of bits. Hence the waste. A code-to-worst example is if you have an alphabet of 2 symbols, A and B, where A is 100 times more common than B. Huffman can only allocate 0 and 1 to these two, and so can not exploit the statistical imbalance at all.

(f) What is a minimum spanning tree? If all the edges in a graph have different lengths will any minimum spanning tree be unique? [10%]

Shortest sub-graph of a weighted graph such that it connects all the vertices. If all edges have different weights then Kruskal or Prim will both have deterministic behaviour and deliver a unique result.

(g) Why might one modify a simple implementation of Quicksort by selecting a random element of the array at each stage to be the pivot? Why might one terminate Quicksort's processing before it manages to split data into partitions of size at most 2? [10%]

So that the average performance is determined by the random generator used to make the random selection, and can not depend at all on the particular input data. Because practical overhead in quicksort may mean that a final pass of simple insertion sort will end up faster than pushing quicksort recursion to the ultimate limit.

(h) You have a binary search tree, and a pointer to one particular item stored in it. Explain how you would find the item in the tree that comes just before your selected one in terms of the sorting order, or determine that there is no such element. [10%]

If the item has a left child then the prior item will be the rightmost item in that left child. Otherwise the item concerned is itself the leftmost item in the tree or some sub-tree. Navigate right and up as far as possible. Note that in many cases a tree does not have upwards pointers so this may be a challenge. When you have traveresed as far as possible this way then if you are at the top of the tree there is no predecessor. Otherwise the prior node is left and up one.

Lectures covered "next key" not "prior key" but the process is the same apart from a reflection!

(i) What operations must be specified to describe an Abstract Data Type for a "stack". What merit is there in introducing abstract data types rather than just using concrete data structures? [10%]

make-empty(), is-empty(), push(), pop(), top() with the relevant identities linking their behaviour. Use of an ADT avoids specifying incidental aspects of behaviour accidentally and encourages or enables the consideration of several concrete implementations.

(j) On current real computers memory access times depend strongly on the extent to which an algorithm has a pattern of memory use that exhibits locality. Quote examples of algorithms that do this and hence run well, and ones that do not and hence behave less well than one might have hoped based on just an asymptotic analysis. [10%]

Quicksort as a good example and heapsort as a bad one are probably as good cases as can be found.

SECTION B

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

2 Suppose that a positive integer is represented in binary, with the binary number then stored as a list with the least significant bit as the first item in the list. Thus the number ten is `1010` in binary and so would become the list `[0, 1, 0, 1]`. Find the maximum and minimum costs of incrementing a number that is represented this way. If m and n are two integers that are roughly the same size what is the cost (in big-O notation) of adding them? [10%]

Start with an integer 1 represented as above (i.e. as a list) perform a long sequence of increment operations. Find a formula for the average cost of the increments. [20%]

A special variant on a tree data-structure has the following properties:

(a) There are trees that can hold 1, 2, 4, 8, ... 2^n items, but no other size.

(b) In any tree the root element will hold the smallest key.

(c) It is possible, in O(1) time, to take two trees each of size 2^i and combine them to form a tree holding 2^{i+1} items.

If you have k items to store you can represent k in binary and thus store your data in a list of these special trees. Explain how to add a single item to a list, preserving the property that it never contains two trees of the same size. Give the amortised cost of your method. [25%]

Show how to merge a list containing m keys and another with n to make a single list of size $m+n$. What is the cost? [25%]

Suppose that you can take one of the special trees containing 2^n keys and in O(1) cost you can remove the smallest key (i.e. the one at the top). You have $2^n - 1$ keys left and these will be returned to you as a list of trees. Given this operation, what algorithm can be used and at what cost if you start off with a list of trees containing k keys in all and you want to remove the smallest key? [20%]

This question builds on material covered in the lectures and derives (most of) Binomial Heaps. Binomial Heaps were mentioned several times in lectures but explicitly not covered! Thus their introduction here is self-contained.

If you have an integer n then in binary it used $\log(n)$ bits. If you increment it then the work you do is all to do with how far carries propagate. So the best cost is $O(1)$ and the worst cost is $O(\log(n))$. When you add two numbers the need to process all bits of the smaller number and may then need to propagate a carry up to the top of the larger so the cost is $O(\log(m))$ where m is the larger number.

When you perform a series of increments every other one is of an even number so costs 1. Of those that are left every other one costs just 2. And so on. The end result is that you can write the average cost as

$$(1/2)1+(1/4)2+(1/8)3+(1/16)4+\dots$$

and this is a series that has arisen at several stages in the lectures as having sum 2.

Now adding a single item to a list of trees is analogous to incrementing an integer, provided you ensure that you know the size that the trees are. So it has amortised cost $O(1)$.

Merging two lists can correspond to adding integers so its cost is logarithmic in the size of the trees.

Finally if you need to remove the smallest item, then it will be at the top of one of the trees in your list, and there are $\log(n)$ trees so there is that cost to find it. Then you pick out that tree and remove its top element to leave you with an "add" operation to put back the items you have left, giving a further logarithmic cost. So the total cost is $\log(n)$.

Candidates should recognise the "remove smallest" task as characteristic of heaps, and this variant has a performance comparable to standard heaps for creation an removal, but has a much better cost for merge. Good students should notice and comment on that. Excellent ones will have read around the course and may know about binomial heaps so recognising what this called is a sign of one who works hard.

3 Why is it normally said that the minimal cost of sorting n items is proportional to $n\log(n)$? [10%]

There are two main suppositions behind this. This first is that all $n!$ (that is factorial) different possible arrangements of the input data must be catered for and one may not rely on serious deviation from uniform distributions over this, and the second is that the sorting must be based on binary choices made by the sorting program. On that basis one can have a tree representing possible computations, with 2-way branching at each place the code makes a choice. The heigh of the tree is then the cost. If the cost is h the tree can have 2^h leaves, ie paths through the computation. If there are $n!$ input orderings the code must have the capacity to do $n!$ different things, hence we deduce $2^h > n!$ from which we extract cost being at least $n\log(n)$.

Give *two* special cases where you could propose an algorithm that would sort in linear time supposing your particular condition applied, Explain why these examples do not undermine the usual $n\log(n)$ prediction. [20%]

I will give *three* cases here to show that candidates have a choice!

(a) If we have a special case where we know that the data is almost in order then simple insertion sort runs in linear time. This does not conflict because it drops the "all orderings likely" assumption.

(b) If we have a small number of distinct values in the data to be sorted then we may use counting sort, or if the values are all known to be in a small numeric range maybe a radix sort. These two methods both rely on the fact that with a small range of values there will be many repeats in the data so the $n!$ raw orderings turn into a much smaller number of distinguishable orderings. They may also use index operations on the data that do not count as binary choices but as multi-way ones.

(c) A case notes in lectures was sorting a deck of cards. There are indeed 52! initial orderings, but by inspecting a card you can do an index operation to place it exactly where it will need to end up. This might be called distribution-sort. It breaches the "binary choice" assumption.

Now suppose you have an ideal parallel computer that can perform several comparisons and associated bits of data movement at once. But if the selection of

(TURN OVER for continuation of Question 3

what data is to be compared depends on the result of another comparison then the two comparisons must happen one after the other rather than at the same time.

Describe Heapsort and analyse its performance if adapted to exploit parallel operations. You may assume that administration and synchronisation of the various parallel operations is not a problem, and that having multiple operations all accessing the array that holds the data does not lead to any slow-down. What performance can be achieved and about how many operations must happen at once? How does your result relate to the "minimum cost of sorting"? [45%]

Heapsort has two phases. The first takes arbitrary data and rearranges it to ensure that the heap property applies. This takes linear cost, and so until and unless I might overall get a sub-linear total cost there is no need to try to exploit parallelism while performing it. But if anybody wanted to I could say fix all the "triangles" at one level of the heap in parallel. The bottom layer has the most of these — $n/4$ — so I may use that many computational units (that is rather a lot) and it is then easy to achieve a total cost of $\log(n)^2$ and with a bit of pipelining a cost of only $\log(n)$. But this will be dominated by the cost of the second stage and so is not liable to be a sensible thing to do!

The second part of heapsort performs one step for each item it removes from the heap. There are n items to remove, and each in turn is swapped to the end of the array. One now need to repair the heap property. Well there are $\log(n)$ layers of heap involved, but of one performs each of thse in turn the heap operations concerned happen one at a time in a way that predictably drops down through the layers of the heap. And the top parts of the heap have been mended very early on.' So one can start the second "remove least" step after the very first clock tick using a new comparator. A little thought shows that if you have one comparison unit for each heap layer (ie $\log(n)$ parallelism) you end up completing the second phase of heapsort in linear time. Hah! Linear time and logarithmic comparisons amount to $n\log(n)$ total resource so this seems very reasonable.

An alternative parallel sort scheme, potentially suitable for use in dedicated hardware, is to bring in each value to be sorted on its own wire. The basic hardware unit has two wires that go in (say a and b), and it transmits the two input values on a pair of output wires (A and B) but now with their positions swapped if necessary so that the larger value is sent on output A.

If the items to be sorted are $(v_1, v_2, v_3, \ldots)$ then a first row of comparison units processes the pairs (v_1, v_2), (v_3, v_4) and so on and a second row follows up on (v_2, v_3), $(v_4, v_5), \ldots$

If rows like this are merely stacked to get a grid of hardware that guarantees to fully sort its input how many comparators are needed in all? [10%]

Explain how ideas from Shell's software sorting method might improve on this scheme by both reducing the amount of hardware needed and shortening the delay between when data is initially presented and the final sorted data becomes available. [15%]

Firstly the two rows are needed because it can not make sense to send the same data into two swapper units at once. One dounle-row of swappers can move any item in the array either one step left or one step right by exchanging it with a neighbour. Well actually in some cases you may get a swap in each row so some items can move two places. If the largest value in the array starts off in the first position it has to move n places to reach the top. Thus we certainly require $n/2$ row-pairs and possiblt n. This represents n^2 hardware budget and n delay.

One could route the output of a double-row back to its input and so manage with linear amounts of hardware and also linear total delay.

Now what about Shell's sort? In a first course like this we do not document its worst-case cost because analysing it is hard and the exact sequence of increments to use is a delicate choice. But what its idea is is to take a sort that is a bit like simple insertion (and the hardware described here has much in common with that!) and enhance it by using it with larger strides. Thus it will do the simple parallel-sort as described above but on a gradually decreasing stride. For a candidate to explain this they need to show that they understand Shell's sort as described in the lectures!

As a note for external examiners and for the future when this crib becomes available to students, there are stride-sequences for Shellsort that lead it to having a guaranteed cost bounded by $n\log(n)^2$ and these lead to hardware sorting networks that are among the best that are available. The introduction of parallelism here related to the start of the question in that it is a further (increasingly important) way in which sorting may be done in time growing less fast than $n\log(n)$.

4 Define the height of a tree as the maximum distance from its root to a leaf. Define the balance of a tree as the difference between the height of its left and right sub-trees. Thus a tree where the tree itself and all its subtrees all have a balance of zero is a perfect binary tree.

If a tree where the balance is always zero has height h how many items are there in the tree? [5%]

Suppose that every position in a tree has a balance that is -1, and that its height is h. Find a recurrence formula that gives the number of items, $N(h)$, in the tree. [10%]

For large h this recurrence has a solution that grows as $N(h) \approx \lambda^h$ for some value of λ. Substitute that approximation into the recurrence and hence find the value of λ. [10%]

$$v_h = v_{h-1} + v_{h-2}$$

Substituting in give $lambda^2 = lambda + 1$ and hence λ is the solution of this quadratic, which can be written in terms of $\sqrt{5}$ or evaluated to get 1.618 (the golden ratio). This is not a difficult sum!

The course material on splay-trees included the following simple rotation that can be applied to a binary tree:

```
      Y             X
     / \           / \
    X   c   ==>   a   Y
   / \               / \
  a   b             b   c
```

Suppose that the node Y starts off so it has a balance of 2 (ie its left sub-tree is 2 taller than its right sub-tree) but all other nodes (including X) have a balance of $+1$, 0 or -1. Find the balance of the tree after the rotation for the three possible values of the balance at X. Identify cases when this does or does not improve the overall balance of the tree, allowing for the possibility that node Y might not be the root of the whole tree. [15%]

This is 15% because it asked 3 little bits, for the balance at X being 1, 0 and -1. These correspond to

```
      Y              X             Y              X             Y              X
     / \            / \           / \            / \           / \            / \
    X   1   ==>    2   Y         X   1   ==>    2   Y         X   1   ==>    1   Y
   / \                / \       / \                / \       / \                / \
  2   1              1   1     2   2              2   1     1   2              2   1
```

where here the numbers mark the height of the sub-tree involved. Note that the tree at X must be (2,2), (2,2) or (1,2) so that the balance at Y is 2.

So in these 3 cases the balance at X ends up as 0, 1 and 2 respectively. In the first case this portion of the tree sees its height decrease by 1, in the other two it stays the same height. Thus in the first case this rotation could have an effect on the balance at higher points in the tree.

In each case new node Y has a balance that is 1 or 0 and in 2 of the 3 cases it brings the balance at X down.

In any case where the balance would remain worse than 1 check the effect of performing a rotation on the tree that starts at X before doing the one at Y and document what can be achieved. [30%]

By now we know that the only case we need to consider is (1,(2,1)). Another way of expressing this is that Y has balance 2 and X has balance -1. All trees below X have +1,0 or -1.

So really there are only two things one could try, and only one makes even slight sense - a leftward rotation at X.

```
    Y           Y           Y           Y            Y           Y
   / \         / \         / \         / \          / \         / \
  X   1  ==>  Z   1       X   1  ==>  Z   1        X   1  ==>  Z   1
 / \         / \         / \         / \          / \         / \
1   Z       X   0       1   Z       X   1        1   Z       X
   / \     / \             / \     / \              / \     / \
  1   0   1   1           1   1   1   1            0   1   1   0
```

These change the node X from (1,2) to (2,0), (2,1) and (2,1). To follow through consequence on these if the rotation at Y is then performed involved checking one case (the (2,0) one) that was not looked at before, but lo and behold in the end all levels end up as balance 1, 0 or -1.

You have a tree that starts off with every node having balance +1, 0 or −1. Show that on adding a new item to it you can use rotations to restore its balance to the same range. How many rotations might you need to make? [20%]

If you add an item as per an ordinary binary search tree then much of the time you will not hurt the balance. But sometimes you will cause something to become deeper and that can leave a tree with balance 2 (or -2) at some point along the path to where you actually inserted. Find the lowest point where it is that far out of balance. Perform rotations as derived here (and the mirror image of them if the tree was right-heavy rather than left-heavy) and you certainly get the balance back to 1,0 or -1 at the point that you act.

Could there be really bad balance left higher up? No, by looking at the fact that what you had just inserted must have changed the height of a sub-tree at least and you end up shrinking it again!

Does this scheme lead to better or worse worst-case imbalance then Red-Black trees? [10%]

And easy rider - the scheme described here does better!

END OF PAPER

Version: final