Crib prepared by A C Norman
ENGINEERING TRIPOS PART IIA

Wednesday 6 May 2009 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

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) Larsen's dynamic hashing is sometimes used when data lives on disc, and it manages to guarantee that retrieving data only ever uses a single disc access. Suppose my data is kept in memory not on disc, but I arrange that memory in blocks so that Larsen's method means I only access my target memory block just once. Is using Larsen in this way liable to be competitive with other in-store options? Is the cost of accessing information $O(1)$? [10%]

Larsen is willing to do lots of extra computation to achieve the goal of just one disc read, and that leads to overhead that will feel severe in the in-store case. But perhaps if memory was actually virtual memory the issue of what is memory and what disc starts to blur and the idea becomes less silly. With regard to $O(1)$, Larsen's method is $O(1)$ in the probabalistic sense that other hashing methods are, but has almost no meaningful bound in the very worst case, because it may have to compute a sequence of different hash and signature pairs until it finds one that gives it access to the data.

(b) Imagine a variation of quicksort that selects two pivots, say p and q, at random with $p < q$. It then partitions the data into values less than p, between p and q and a third block for data greater than q. It recurses on each sub-block. Give a recurrence formula for the cost in the ideal case where the partition splits the data into three equal-sized regions. Is the ideal cost still proportional to $n\log(n)$? [10%]

$c(n) = 3c(n/3) + n$ provided you can do the 3-way partition in linear time. This still gives $n\log(n)$ growth rate.

(c) Give a very brief sketch of an algorithm to find a minimum spanning tree for a graph. You do not need to include either a proof of its correctness or an analysis of its costs. [10%]

Kruskall or Prim! Eg start with a arbitrary single selected vertex that is a sub-tree of the graph with no edges. Then if the graph had n vertices you will do $n-1$ stages. At each stage identify the shortest edge that has not been used before, that has one end part of the existing sub-tree but the other end not. Add it to the tree. Done!

(cont.

(d) What statistical expectation about text-files does the Lempel Zif compression method (as used in "zip") rely on to allow it to compress data? How might you construct data that did not satisfy this expectation? [10%]

That if some pair or string of characters in the text occur once then that greatly increases the chances that the same sequence will occur again. Random data, or data that had been encrypted should not have this property and so should not be compressed by Lempel Zif.

(e) Explain what is meant by "amortised" cost analysis, giving an example from the realm of data structures and algorithms where it is helpful. [10%]

Average cost over a succession of operations on a single structure. This is mainly used when some individual operations may be expensive but the expensive case can not recur too often. Perhaps good examples would be splay trees and garbage collection.

(f) Explain how you could add a single new item into a heap (that is "heap" as in "heapsort"). [10%]

Place the item in the array at the position just beyond active data. Now you need to correct the heap property for the enlarged heap, and that can involve bubbling the new item up, potentially $\log(n)$ levels.

(g) Explain what aspects of computing costs are *not* captured when you specify costs using big-O notation. Why is this often found useful and how does big-Theta notation provide a variation? [10%]

Big O misses out constant factors, it is only an upper bound (and there is not guarantee it is sharp, and it may not apply at all in small cases. These limitations make it independent of technological advances, cope with cases where sometimes a procedure may be accidentally fast and simplify analysis by looking only at assymptotic behaviour. Big Theta provides a lower as well as an upper bound and thus is always sharp to within a constant factor, but is inapplicable when the cost of an algorithm may vary wildly depending on input, while big-O still gives you a worst case cost, which is often what anybody concerned with resource allocation and who has a conservative mind set will need.

(h) What problem does Ephemeral Garbage Collection (sometimes known as Generation-based collection) seek to address as a potential problem in other garbage

(TURN OVER for continuation of Question 1

collection strategies? What observation or assumption does it make about the patterns of storage allocation and release in order to achieve a benefit? [10%]

The intrusive pause that can make simple garbage collection unsuitable for use in real time contexts. It works on the basis that most allocated memory will be discarded early so memory that has already lasted a while is liable to last yet longer. It can therefore put most of its energy into trying to collect garbage from amongst recently allocated memory blocks.

(i) A simple implementation of quicksort is given data where all the values to be sorted happen to be identical. Explain how quicksort behaves in this case, giving enough details of the variant you are describing to back up your explanation, and predict the costs. [10%]

Any simple partitioning scheme is liable to end up here with the pivot at an extremal position, and the cost will grow as n^2 even though one might tend to feel that this ought to be an easy rather than a hard case!

(j) Are the following statements true, and comment on the line of reasoning: (a) Quicksort runs in time $O(n^3)$. (b) MergeSort runs in time $\Theta(n\log(n))$. (c) Big-O and Big-Θ analysis almost always gives predictions that are relevant in the real world. (d) So you should almost always use merge-sort in preference to quicksort. [10%]

(a) is true but is not sharp, so risks being misleading. I.e. it is $O(n^3$ because it is in fact $O(n^2)$ and big-O notation is just an upper bound! (b) is true. (c) is probably true. (d) mainly fails to note that mergesort has memory needs that make it less suitable than quicksort, and the average time taken by quicksort is so good that its infrequent worst-case cost is often not a worry.

SECTION B

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

2 Explain the facilities needed in a free-store management system, and comment on the different trade-offs offered by *first fit*, *buddy* and *garbage collection* strategies. [35%]

Two particular garbage collection methods are *mark and sweep* and *stop and copy*. For each of these explain the stages and steps involved, constraints that they impose on the programming system that uses them. Identify the parameters that determine the cost of a single garbage collection. Explain a case where *stop and copy* would complete its collection significantly faster than *mark and sweep*. [30%]

Suppose your application has about N units of data alive at any one stage, and that the computer on which it is run has M units of memory available to be used (it had better be the case that $N < M$). If some aspect of the growth-rate of the cost of one of the garbage collection procedures is $O(n)$ then use a specific symbol, say K_1 for the associated constant of proportionality.

In terms of these constants of proportionality and the ratio between M and N analyse which method is likely to be more economical. [35%]

A free store system needs to provide a way for the user to request a new block of memory and some scheme that allows unwanted memory block to be recycled. First-fit and Buddy systems place responsibility for recycling on the user, so user errors may cause trouble and user code has to track when memory can be freed. Garbage collection relieves the user of that job but the run-time system it lives in must allow the garbage collector to identify all places where pointers to memory blocks are kept, and that is only feasible when using disciplined programming languages. A serious problem for storage management is fragmentation and the impact of that can be balanced against both speed of allocation (and deallocation) and waste of space for overheads. First Fit does not have much overhead, but allocation and deallocation can be costly and fragmentation can be bad in many circumstances. The Buddy system is always reasonably fast and addresses fragmentation somewhat, but wastes space. Garbage collection can impose occasional long delays making it unsuitable (in its simple forms) for real-time use.

Mark and sweep: First start from all roots and mark all reachable data as live. Then sweep the heap area: reset the mark bit on live data and gather the dead data for recycling.

(TURN OVER for continuation of Question 2

Cost of mark phase may be proportional to amount of live data, of sweep phase to heap size.

Stop and Copy: Arrange heap in two half-spaces. Copy live data from existing space into the alternate one (cost is linear in amount of live data). Update all pointers to refer to relocated data (can be done as you go). Then merely discard the old space wholesale. Cost proportional to amount of live data.

For small amount of live data in a large heap the cost component of the sweep phase of mark and sweep can dominate, and stop and copy will be faster. For large amounts of data one should note that with stop and copy you have to reserve half your memory for the alternate heap-half, so if I imagine $N/M = 0.25$ then stop and copy will be invoked 3 times as often as mark and sweep, but cost of stop and copy will be around N while that of stop and copy has a term M – if you imagine the various constants all around equal you find this is about the cross-over point.

3 Dijkstra's algorithm for the single-source shortest path problem is often used in the context of routing in computer networks. Give an outline of just what problem it solves, how it works and any constraints on the graph that may need to apply. [30%]

A the graph has V vertices and E edges. The vertex to which you are trying to find path turns out to be as remote from the source as possible. Consider an implementation of the algorithm that identifies each next vertex to process by merely scanning through all the vertices. Do not use any form of heap or priority queue, but you may store and update information in the vertices themselves. In terms of big-O notation how does your cost depend on V and E. A graph is called "dense" if it has (almost) as many edges as any graph with that many vertices can have. How many edges can a graph with V vertices have? Hence express your cost prediction merely in terms of V. [30%]

Now suppose that you have a "sparse" graph (one with many fewer edges than a dense graph would have). Re-analyse your algorithm now supposing that you keep all vertices in a normal Heap, and use the heap as a priority queue to track which vertex to process next. You may assume without further discussion that of a heap contains N items at some moment that the item with smallest value is instantly available, and that new items can be added, old ones removed or the numeric weight associated with an item in the heap updated in time proportional to $\log(N)$. For dense graphs is your new method an improvement on the previous one? A Fibonacci Heap is a data structure whose details you are not required to understand, but which can be used to create a priority queue where removing the top item from the queue has logarithmic cost and all other operations have unit cost (in the amortised sense). Would its use change matters? [40%]

Dijkstra's algorithm finds the shortest distance from a given source vertex A to other vertices within a (weighted) graph. It can either run until it has found distances from A to all other vertices or be terminated early to find the shortest distance to a nominated destination B. It is a greedy breadth-first search. At each stage some vertices are "known". Initially only vertex A is "known" with distance 0. At each step the unknown vertex that will be next closest to A is identified. When it is make "known" its neighbours may be inserted into the priority queue used to help make finding the next vertex to process fast. The graph must not have any negative weights.

If you do not use priority queues then there are V steps (one per vertex). Each one traverses all the edges that exit from that vertex, but each edge is only traversed once, so the edge-related cost is E. For each step we scan V vertices to find the nearest. So in all we have a $O(E+V^2)$ cost. A dense graph with V vertices has around V^2 edges so this is just $O(V^2)$.

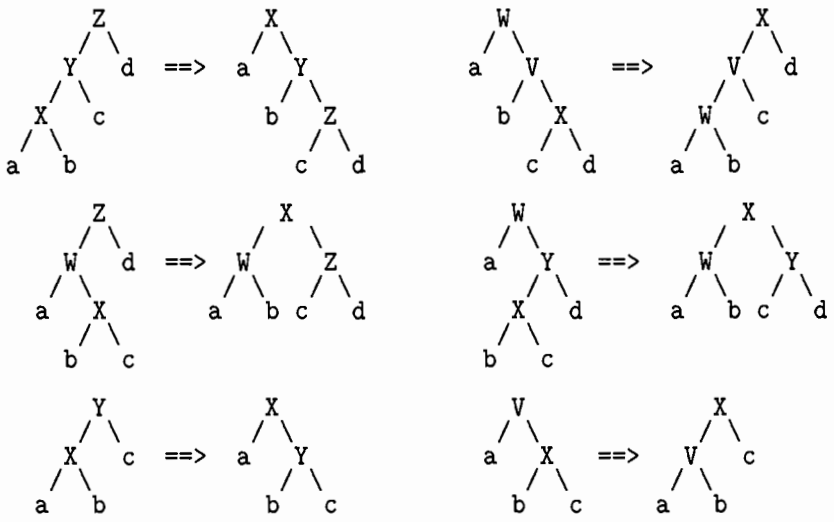
(TURN OVER for continuation of Question 3

With a (normal) heap as a priority queue the heap size is V so each heap operation costs $\log(V)$. There is an operation for each edge (updating the distance of the vertex at its end) and an operation for each vertex (when it is marked as known) so the total is $(E+V)\log(V)$.

If the graph is dense $E = V^2$ and this ends up as $V^2\log(V)$ which is actually worse than the previous apparently crude method!

With a Fibonacci heap we have $V\log(V)$ from when we get to "know" vertices and just a unit cost times E and this is as good as the simple method for dense graphs and as good as the ordinary heap one for totally sparse graphs (eg when E and V are about equal).

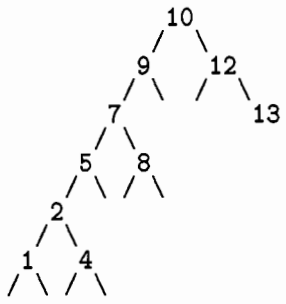
4 The lecture notes contained the following set of diagrams relating to operations on Splay Trees:



Explain when and how these transformations are used and how the choice of which of them to use is made. [25%]

Explain the key properties that Splay Trees then exhibit and comment when they may be useful and when some other structure (which you should also describe at least briefly) would be preferable. [25%]

Starting with the following seriously unbalanced tree, show what happens when you use Splay Tree procedure and access the node containing the value 1. [25%]



Show what the result would have been if you only used the simpler rotations from the bottom row of the original table of transformations. [25%]

When any item is added to or accessed in a splay tree the given rotations are used to bring the item concerned to the root of the tree. Where possible the rotations that move it up 2 places are used, while the very last step can use the final move that lifts it one place.

(TURN OVER for continuation of Question 4

Splay trees provide a guaranteed amortised cost of $\log(N)$ for adding and accessing data in a tree, and furthermore the value N relates to the set of values used (so ones that are added to the tree and then not accessed again do not significantly add to the cost). The main alternative I think I would compare against would be red-black trees. Red-black guaranteed log time on every single access, so would be preferable for any real time use. Splay trees are at their strongest when the distribution of accesses is severely non-uniform because then their self-optimising properties can really help. Splay trees do not need any extra data in the tree, while red-black trees need a bit to indicate red or black.

The working through a set of rotations on a sample tree is in fact pretty similar to an example given in the lecture notes and so I will not draw out all the resulting trees here. When you use only the single rotations in general and for large cases the tree does not get rebalanced.

END OF PAPER