The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, AUTUMN SEMESTER 2012-2013

ALGORITHMS AND DATA STRUCTURES

PARTIAL ANSWERS & HINTS

Time allowed TWO Hours

The answers are NOT “model answers” but are only given in the form of notes and hints. This should be enough to check whether your ideas are correct, but do not treat these answers here as a template to be memorised!

This version is from Moodle for G52ADS 2013-13, and may be updated at any time – so always return to the original.

The general structure of questions is that roughly the first 20 marks are associated with bookwork or standard computation. Roughly the last 5 marks require deeper insight to be able to answer – but conversely the marking takes account of them being harder, and so requires less precision, and giving more credit for having a rough idea.

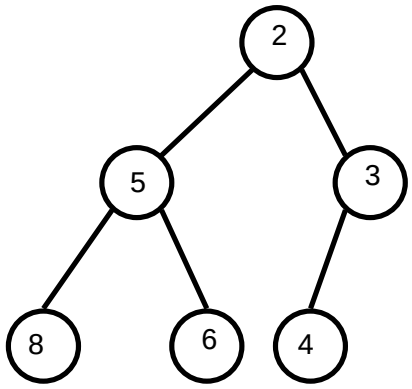
NOTE:

You may notice that in the 2012-13 exam there is no large “in-depth” question on advanced sorting algorithms (merge or quick sort) just “little questions” e.g. in the first compulsory question. This reflects the general guidelines to module convenors that topics (in this case, advanced sorting) are not assessed in depth multiple times during a session. (Though, testing in slightly different ways via ‘small’ questions is reasonable, so that the topics cannot be entirely ignored.)

In the 2012-13 session the C/W 3, the main programming project, was "Implementation of selection sort and merge sort.” So in that year, the detailed working of the merge sort was assessed in depth by the C/W 3 and so was not covered in depth again by the exam. However, you should not take this as an indication that the (advanced) sorting algorithms would not be tested in depth in other years.

1.

Part	Answer	Comment
a	D	Mostly just need to know to ignore the smaller n log n term
b	C (or E)	insertion is straightforward and bookwork – the harder "deletion" is tested in Q6. (C and E being identical was an error in exam preparation)
c	E	need to know that the height of a balanced tree is bounded above by log n growth
d	E	A and B are inappropriate. Only E expresses that after the first pass, the first two entries are sorted
e	A	standard – note that the distorted way of drawing should not distract from C being an immediate neighbour of A
f	C	final state is shown below



QUESTION 2.

a) [Bookwork, but with extensions to ask for understanding]
The definitions are from lectures, e.g.
f(n) is O(g(n)) if and only if there exists c > 0 and n0 such that
f(n) <= c g(n) for all n >= n0

It means that for large enough n – corresponding to the n >= n0, that f is less than some constant times g, and so the growth of f is limited by that of g.
The point value of the constant does not matter, so a “there exists” is enough, but the c must be chosen before n, hence the order is “exists c … for all n”

f(n) is Omega(g(n)) if and only if there exists c > 0 and n0 such that
f(n) >= c g(n) for all n >= n0
with similar explanation as for big-Oh

theta is both big-Oh and big-Omega
and combines both, but the corresponding constants ‘c’ need not be the same for each part

f(n) is o(g(n) iff for all c > 0 there exists n0 such that f(n) <= c g(n) for all n >= n0

b) [Computation, based on the definitions] statement is true: need to show O and Omega
for big Oh:
2 n^3 + n <= c n^3
e.g. take c = 3 then need
n <= n^3 so n >= 1, c=3 n0=1 will suffice
for big Omega
2 n^3 + n >= c n^3
c = 1 and n0=1 will suffice

c) [Computation, based on the definitions]

need

2 n log n <= c n (log n)^2
2 <= c (log n)
2/c <= log n divide by c - which is possible as c > 0
2^(2/c) <= n

so n0 = 2^(2/c) suffices for any c > 0.

Note how n0 needs to depend on c – hence, in the definition, the order “forall c … there exists n0” is important

Hence the statement is proved.

QUESTION 3.

a)

The Vector is similar to an Array mixed with a Stack in that it allows direct access to all elements using their index, or rank (the number of elements preceding the required entry).
Its main difference is that it automatically resizes whenever an element is pushed onto the end.
A common usage would be reading lines from a file of unknown size, or from standard input, where they need to be stored, but one does not know in advance the number that need to be stored.

b)

Suppose some individual operation (such as 'push') takes time T in the worst-case
Suppose do k such operations taking time T_k
Then kT is an upper-bound for the total time; however, such an upper-bound might not ever occur.
The time T_k might well be o(kT) even in the worst-case
the average time per operation, T_k /k, is sometimes the most relevant quantity in practice

c)

i) We replace the array k = n/c times
Each "replace" costs the current size
The total time T(n) of a series of n push operations is proportional to
n + c + 2c + 3c + 4c + … + kc =
n + c(1 + 2 + 3 + … + k) =
n + ck(k + 1)/2
Since c is a constant, T(n) is O(n + k2), i.e., O(n2)
The amortized time of a push operation is O(n)

ii) We replace the array $\boldsymbol{k} = \log_2 \boldsymbol{n}$ times
The total time $\boldsymbol{T}(\boldsymbol{n})$ of a series of $\boldsymbol{n}$ push operations is proportional to
$\boldsymbol{n} + 1 + 2 + 4 + 8 + \ldots + 2^{\boldsymbol{k-1}} =$
$\boldsymbol{n} + 2^{\boldsymbol{k}} - 1 = 2\boldsymbol{n} - 1$
$\boldsymbol{T}(\boldsymbol{n})$ is $\boldsymbol{O}(\boldsymbol{n})$
The amortized time of a push operation is $\boldsymbol{O}(1)$
That is, no worse than if all the needed memory was pre-assigned!

d) The fact that the Heap is used as a vector does not affect the key point which is that adding a new entry has amortised complexity that is still O(1). This is better than the O(log n) of the other parts of the heap operations
The worst case becomes O(n) because might need to resize the Vector, But amortised stays the same at O(log n) because the cost of the underlying resize is amortised as normal.

QUESTION 4.

a) Standard explanation of Dijkstras algorithm: maintain a priority queue of nodes with the length of the currently best know path to that node. Also, keep a closed list of nodes. Initialise with the start node.
Pop the first element of the PQ and expend to find unvisited neighbours, compute new lengths and add them to the PQ, add the popped node to the closed list.

PQ	Closed
[A(0)]	{ }
[B(10), C(11)]	{ A }
[C(11), D(18)]	{A,B}
[D(18), F(27)]	{A,B,C}
[E(20), F(27)]	{A,B,C,D}
[F(24)]	{A,B,C,D,E}

so shortest path is length 24 A-B-D-E-F

b) A spanning tree is a subset of the edges of the graph, which forms a tree and that includes all the nodes of the graph. The weight of the spanning tree is the sum of its edge weights. The MST is a spanning tree for which there does not exist another tree with smaller weight.

Prim's algorithm (bookwork): is a simple greedy algorithm that starts from a single node and builds a spanning tree one node and edge at a time. At each iteration it adds the edge with the minimum weight over all edges that connect a node in the tree with one not in the tree.
A simple star graph is an example as needed, because the graph itself is already its own MST.

c. In modifying Dijkstra we merely need to include the width when computing the new distance. There are variations possible on where the widths are counted, but one option is to not count them until expansion, and not counting the start and finish nodes

PQ	Closed	
[A(0)]	{ }	
[B(10), C(11)]	{ A }	expand B but use w=3
[C(11), D(21)]	{A,B}	expand C
[D(21), F(30)]	{A,B,C}	expand D
[E(25), F(30)]	{A,B,C,D}	expand E
[F(30), F(32)]	{A,B,C,D,E}	

pop F, to get a path of length 30 A – C – F = 11+3+16

QUESTION 5. This question concerns Maps and hash tables. …

a. standard bookwork from lecture notes.
Need to define the table and the has functions, and that it allows a key to be found quickly (usually)

b.

Collisions are when two keys map to the same hash value.

One option is linear probing which means that the next available free slot is used instead.

On looking for an entry we will scan along contiguous sequences until hitting a blank.

The main catch with this is that when it is used, if an entry is deleted from the original sequence of values, then we risk losing access because we stop prematurely. A simple fix to this would be to also remove all entries not at their natural home and then re-insert them afterwards.

```
severn:~$ awk 'BEGIN{for(x=1;x<10;x++){print x,(3*x+5)%7}}'
1 1
2 4
3 0
4 3
5 6
6 2
7 5
8 1
9 4
```

insert 6
[# # 6 # # # #]
insert 4
[# # 6 4 # # #]
insert 8
[# 8 6 4 # # #]
insert 1
[# 8 6 4 1 # #] by linear probing
delete 6
[# 8 1 4 # # #] ←- KEY STEP as need to remove and reinsert 1 so that do not lose it

delete 1

[# 8 # 4 # # #]

c: The main thing is that we need to create a larger table, and so need to change the hash function so everything will have a new location.

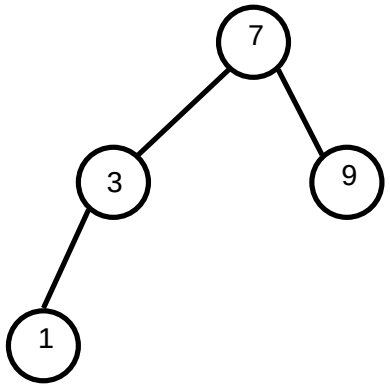
We could rehash everything immediately but this is expensive as will be O(n) – especially if in a real-time system. Another option is to simply maintain two separate tables and look in both. New entries are added to the larger new table.
Entries might be moved from one table to the next.

The typical complexity will then still be O(1) but with the worst case still being O(n) as usual.

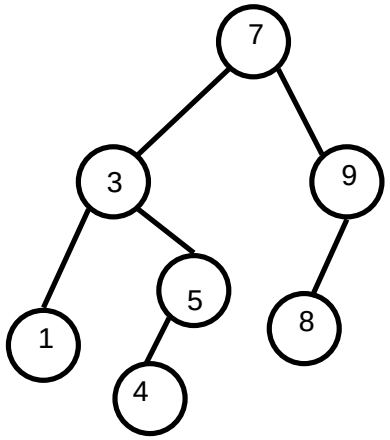
QUESTION 6. This question concerns Maps and Binary Search Trees.

Standard bookwork.
Map has the (methods as in lecture notes)
find: find a key and return its value if present
insert a key with its value, maybe return the old value if it was already there
remove a key, optionally return the value
size, isEmpty
BST has the numeric ordering left-tree <= parent <= right-tree
but heaps have the property that both sub-trees are >= the parent.

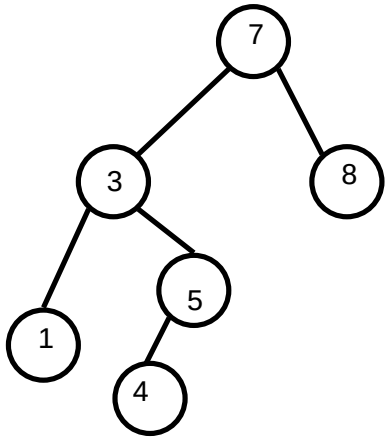
b)



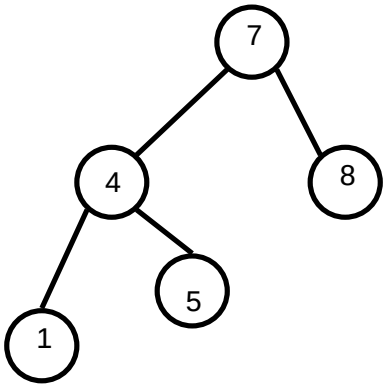
insert(5) , insert(4) insert (8) is straightforward and gives



delete(9) is the case with one child and gives



delete(3) is the case with two children, and the standard would be to replace with the next numeric entry to the right giving



another possibility is to replace the 3 with the 1, but if done then it ought to be explained why.

c.

i. The problem of doing it directly is to find the key that needs to be replaced, as in a heap there is no way to do this directly in O(log n) rather than just doing an exhaustive search taking O(n)

(3 marks)

ii. If we maintain a separate BST then we can at least use the BST to keep the location (index) of each key within the heap, and so find it quickly. To do the finding in O(log n) we need to use a version of a BST that maintains the balance.

Once we have found the key, then if we change its value we will need to do an up or down heap operation. This means that the locations of keys within the heap will also change, and so each time they change we will need to do an update to the BST. We could do this be just removing the old key and then re-inserting with the new key and location. These operations will still be O(log n) formally. However, it could well be that constant factors make it quite slow, and not worthwhile unless the heap is very large.