Tutorial 5: AVL-trees, Heaps & the Master Theorem Example Solutions

(1) See Figure 4.1.

(2) (a) Copying A and then B into an array C of size $2n$ takes time $\Theta(n)+\Theta(n) = \Theta(n)$. Now we call buildHeap on C. From the notes we know that buildHeap takes time $\Theta(2n) = \Theta(n)$. Thus the total time is $\Theta(n) + \Theta(n) = \Theta(n)$.

(b) Note that if k is a key that is larger than the root of A and the root of B then we can build a new heap with k at its root and A, B as the left and right subtrees respectively. (We need A to be complete for this to work otherwise the resulting tree is not almost complete. The fact that B is complete is not needed but as it has the same number of elements as A it is necessarily complete.) Now we delete the root of this heap to obtain just the elements of A and B in a single heap. We delete by copying the last item of a heap into the root and then calling heapify; after this we know that we have our two heaps turned into one. Given this observation, we do not need to create the artificial root with k; we get the same effect by using the last entry of B as a root (we delete it from B) and then use A and the resulting tree from B as left and right subtrees (the order is important). Note however that this observation (i.e., avoiding the introduction of a root vertex with a new key) does not affect the runtime growth rate.

The cost is $\Theta(\lg n)$ for copying and deleting the last element of B. We then have a $\Theta(1)$ cost for joining the two trees with a new root vertex. Finally the call to heapify takes time $\Theta(\lg(2n)) = \Theta(\lg n)$ since the tree has $2n$ vertices and so has height $\Theta(\lg(2n))$. Thus the overall cost is $\Theta(\lg n) + \Theta(1) + \Theta(\lg n) = \Theta(\lg n)$.

(3) This sample answer is given in considerable detail as regards the analysis and correctness since it illustrates some subtle points that could not be covered in lectures. You were not expected to go into so much detail in your own answer, the shorter version of the answer for each part given below would have been sufficient.

We create an empty AVL tree T and then insert all the elements of A into T one at a time. After this we put the elements back into A in increasing order by calling preOrder(T, A) defined as follows:

```
$i \leftarrow 0$
preOrder$(T, A)$
  if $T$ is not empty then
    preOrder$(T.left, A)$
    copy element at root of $T$ into $A[i]$
    $i \leftarrow i + 1$
    preOrder$(T.right, A)$
```

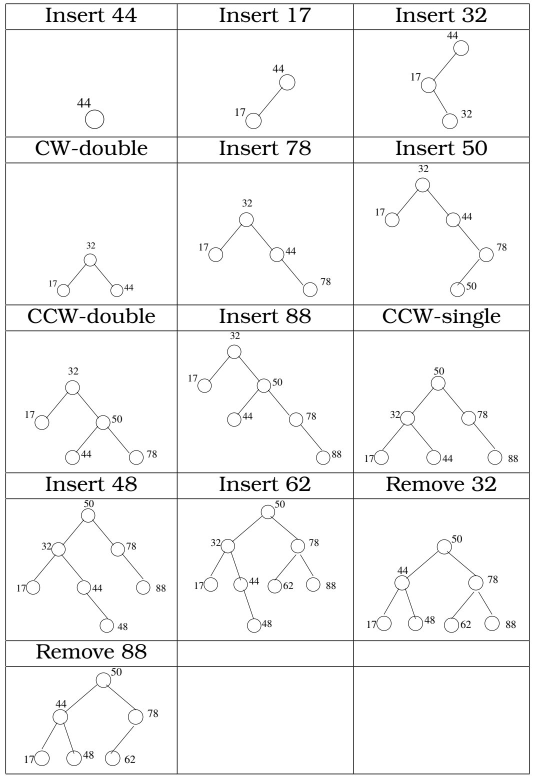

Figure 4.1. Evolution of the AVL-tree of question 2

Here we have used a variable i that is global to preOrder to keep track of how far the array has been copied. To be precise the value of i gives us the next available location of A to copy the next element from the tree if there is one, otherwise i is just out of range of the array.

Correctness is intuitively clear from the ordering property of binary search trees: we put things on the left subtree in the array then the root and then things on the right subtree and this is done recursively. We can justify this intuition with a more formal proof by induction on n; we claim the following:

> Once preOrder(T, A) is finished, the array A contains all the elements of T in sorted order starting with the position given by the value a that i had before the call, moreover once the call is finished we have $i = a + n$ where n is the number of elements in T.

If $n = 0$ the claim is clearly true since there are no elements in T and indeed the assignment $i = 0$ is not changed so $i = 0 + 0$ as required, this takes care of the basis of the induction[1]. Now for the induction step; we assume that the inductive claim is correct for all values up to n. Consider now the case $n+1$, i.e. we call preOrder(T, A) where T has $n+1$ entries (obtained by inserting the entries of the array A). Let L denote the left subtree of the root of T and R the right subtree. Since L has at most n entries the induction hypothesis applies and so we are assured that L is copied into the the array in sorted order starting at $A[0]$ and leaving i with the value $0+|n_L|$ where $|n_L|$ denotes the number of entries in L. The method then copies the entry in the root at the position i of the array which is indeed the next unused position thus far (in the copying process) and the value of i is set to be $1 + n_L$. Since everything in L is no larger than the entry at the root of the tree it follows that the array remains sorted. Now again the number of entries in the right subtree R is no more than n so the induction hypothesis applies. Thus the call preOrder(R, A) places the entries in R into A starting at position $1+|n_L|$ and finishing with $i = 1 + |n_L| + |n_R|$, i.e., $i = n + 1$. Since everything in R is at least as large as the entry at the root it follows that the array is indeed sorted. This completes the induction.

Finally, since the AVL tree contains exactly the entries of the original input array, it follows that the final version of this array is just the original one but with the entries sorted. Before leaving this proof it is worth your while looking again at the induction hypothesis, why did we not just say that the value of i starts off as 0?

Now for the runtime analysis. Inserting the array entries into the AVL tree costs $O(n \lg n)$ since there are n items to insert and each one is inserted into a tree of size no larger than n (so the cost is $O(\lg n)$ each time). It remains to account for the cost of copying the AVL tree back into the array. We might be tempted to write a recurrence for this recursive algorithm but we face the

[1]One might object that arrays of size 0 do not make sense in the context of software. This is not really a problem we can certainly assign a meaning to arrays of size 0 and declare them in the usual way with all operations having their usual meaning. Alternatively we can start the induction at $n = 1$ but this seems a bit humdrum.

problem that we do not know the exact sizes of the left and right subtrees at any stage (they can vary within certain limits). Intuitively the cost is linear because for each vertex of the tree we just do a constant amount of work.

Once again we can justify our intuition by an induction argument. We claim that the cost of the procedure for a tree with n entries is at most $an + b$, for all n, for some constants $a, b > 0$. The basis of the induction is $n = 0$ and here we can follow the algorithm to see that it does a constant amount of work (so we just need b to be large enough to account for this). This time we'll assume the claim is true for all values strictly less than n and prove it true for n (this is just a notational change, we could just as well assume it true for values up to n and prove it true for $n+1$ as before). Suppose now that the left subtree has size n_L and the right subtree has size n_R; note that $n = 1 + n_L + n_R$. The method does a test then recurses on the left subtree, copies the root and recurses on the right subtree. In addition it increments the variable i. So the total cost is at most

$$(an_L + b) + (an_r + b) + c.$$

where c accounts for the copying of the root element, the test and the increment to i. Now we need to have

$$(an_L + b) + (an_r + b) + c \leq an + b,$$

in order to complete the induction. Since $n = 1 + n_L + n_R$ this is equivalent to

$$(an_L + b) + (an_r + b) + c \leq a(1 + n_L + n_R) + b,$$

i.e.,

$$b + c \leq a.$$

So provided we choose a to satisfy the inequality the induction step goes through. (Recall that the only other condition on the constants was that b should be large enough to account for the runtime when $n = 0$, while c is as described above. Thus we certainly can choose a to be large enough as there is no other constraint on it. Since b, c are constant we can also take a to be constant.)

It is worth discussing a common error in connection with the above. We might be tempted to use as induction hypothesis the claim that the runtime is $O(n)$. The basis is of course correct. This time we will revert to going from case n to case $n+1$ as the induction step, simply for notational reasons. For the induction step we argue that the amount of work we do is

$$O(1) + O(n-1) + O(1) + O(1) + O(n-1),$$

just following the code and applying the induction hypothesis to the calls on the left and right subtrees (using the fact that their sizes are at most $n-1$). The expression then simplifies, by the usual laws, to $O(n)$. Since this is certainly $O(n+1)$ we are done.

The preceding 'proof' is incorrect, in fact it stinks. To see why let us spell out our claims. We begin by claiming that the runtime of preOrder is $O(n)$.

What does this mean? We are saying that there is a *fixed* function $f \in O(n)$ such that the runtime of preOrder for all large enough inputs of size n is at most $f(n)$. Proving this claim by induction entails two things:

- Check that the base case holds. But what is the base case if our claim is that the runtime holds for all large enough n? OK we can fix this by claiming the bound holds for all $n \geq n_0$ for some specified n_0 and then we check the claim for $n = n_0$. This is no problem as n_0 is a constant which we can assume is bigger than 0 (we get to choose n_0 remember) and the runtime of the algorithm for a given fixed size input is a constant hence $O(n_0)$ (which is the same as $O(1)$).
- For the induction step we assume that the runtime is at most $f(m)$ for all inputs if size m with $n_0 \leq m \leq n$ and must then show that the same holds for inputs of size $n+1$. That is, the runtime on inputs of size $n+1$ is at most $f(n+1)$.

Here is the crucial observation: throughout the above we are making assertions about the *same* function f. In the supposed 'proof' above the induction step took the form

$$O(1) + O(n-1) + O(1) + O(1) + O(n-1) = O(n) = O(n+1).$$

But remember that in asserting an equality such as $O(f) + O(g) = O(h)$ we are saying that for any functions $f_1 \in O(f)$ and $g_1 \in O(g)$ there is a function $h_1 \in O(h)$ such that $f_1(n) + g_1(n) = h_1(n)$, we are not necessarily keeping to the same function throughout.

Note that the problem just described does not occur when we do an asymptotic runtime analysis of an algorithm. There we are not using induction; we find the upper bound by accounting for the costs of various parts of the algorithm for all inputs of size n. So our functions do change as we add up the costs but at the end we have a single fixed function (which we only specify up to growth rate).

(4) (a)

$$s(n) = \begin{cases} \Theta(1), & \text{if } n \leq 3; \\ 2s(n/2) + \Theta(1), & \text{if } n > 3. \end{cases}$$

Our "extra work" exponent k is $k = 0$ (because the extra term is $\Theta(1) = \Theta(n^0)$). The reduction in size for the subproblems is $b = 2$. The number of small subproblems is $a = 2$. Hence the critical exponent is $\log_b(a) = \log_2(2) = 1$. Thus $k < \log_b(a)$. We are in case (I) of the master theorem, so $s(n) = \Theta(n^{\log_b(a)}) = \Theta(n)$.

(b)

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 2; \\ 2T(\lfloor n/3 \rfloor) + T(\lceil n/3 \rceil) + \Theta(n), & \text{if } n > 2. \end{cases}$$

Remember that floors and ceilings don't matter for the Master theorem. Our "extra work" exponent k is $k = 1$ (because the extra work term is $\Theta(n)$). The reduction in size for the subproblems is $b = 3$. The number

of small subproblems is $a = 3$. Hence the critical exponent is $\log_b(a) = \log_3(3) = 1$. Now $k = \log_b(a)$. We are in case (II) (the "middle case") of the master theorem, so $s(n) = \Theta(n \lg n)$.

(5) (a) If $n = 1$ then we call algB(A) twice and this costs $2\Theta(1) = \Theta(1)$. We also execute line 3 (but not the body) and this costs $\Theta(1)$ so the overall cost is $\Theta(1) + \Theta(1) = \Theta(1)$. Otherwise the two calls to algB(A) cost $2T_B(n)$. Lines 2, 3, 4 cost $\Theta(1)$ each and so cost a total of $\Theta(1)$. The loop on lines 5-6 is executed $\lfloor n/2 \rfloor$ times and each time costs $\Theta(1)$. Since $n/2 - 1 \leq \lfloor n/2 \rfloor \leq n/2$ we have $\lfloor n/2 \rfloor = \Theta(n)$ and the cost of the loop is $\Theta(n)\Theta(1) = \Theta(n)$. Similarly the loops on lines 7-8 and 11-15 each cost $\Theta(n)$. Thus the three loops incur a total cost of $\Theta(n)$. Line 9 costs $T_A(\lfloor n/2 \rfloor)$ while line 10 costs $T_A(\lceil n/2 \rceil)$. Thus the total cost is $T_A(\lfloor n/2 \rfloor) + T_A(\lceil n/2 \rceil) + 2T_B(n) + \Theta(1) + \Theta(n)$. It follows that the recurrence is

$$T_A(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1; \\ T_A(\lfloor n/2 \rfloor) + T_A(\lceil n/2 \rceil) + 2T_B(n) + \Theta(n), & \text{if } n > 1. \end{cases}$$

(b) Observe that $2T_B(n) + \Theta(n) = 2\Theta(n^\alpha) + \Theta(n) = \Theta(n^\alpha + n)$. If $\alpha \leq 1$ then $\Theta(n^\alpha + n) = \Theta(n)$ otherwise $\Theta(n^\alpha + n) = \Theta(n^\alpha)$. Applying the Master Theorem we have $e = \log_2(2) = 1$. So if $\alpha \leq 1$ then $k = 1 = e$ and the middle case applies giving us $T_A(n) = \Theta(n \lg n)$. If $\alpha > 1$ then $k = \alpha > e$ and the third case applies giving us $T_A(n) = \Theta(n^\alpha)$.

In simplifying $\Theta(n^\alpha + n)$ it is tempting just to quote items 5 of Theorems 2.5 and 2.8 of Note 2. However these cases deal with *polynomials* and by definition the powers that occur in a polynomial are all natural numbers whereas α is some non-negative real number. Luckily in this case the proofs go through even if we allow exponents to be real numbers, the statements and proofs need a bit of extra care. In our case we can just observe that if $\alpha \leq 1$ then $n^\alpha \leq n$ and so $n \leq n^\alpha + n \leq 2n$. On the other hand if $\alpha > 1$ then $n^\alpha \geq n$ and so $n^\alpha \leq n^\alpha + n \leq 2n^\alpha$.