Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 4: The Master Theorem and Heaps
SOLUTIONS

1. (a) *Write down and justify the recurrence relation satisfied by $T(n)$.*

   Choosing not to count the function header as a line to be executed, when $j - i = 1$ we execute exactly 3 lines (whether our search succeeds or fails) Otherwise, choosing to count the 'else' as a line to be executed, we will perform 5 line executions, of which will be the subcall **binarySearch**(A,key,i,k) or **binarySearch**(A,key,k,j) — plus of course all line executions performed by this subcall itself.[1] The latter are subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ respectively, so in the worst case the subproblem will have size $\lceil n/2 \rceil$.[2] This leads us to the recurrence:

   $$\begin{aligned} T(1) &= 3 \\ T(n) &= T(\lceil n/2 \rceil) + 5 \quad \text{when } n > 1 \end{aligned}$$

   (The numbers will be slightly different under other views of what counts as a line execution.)

   (b) *Simplify this down to an asymptotic recurrence relation, and solve it using the Master Theorem.*

   Simplifying down to asymptotics, and ignoring the ceiling, we get

   $$\begin{aligned} T(1) &= \Theta(1) \\ T(n) &= T(n/2) + \Theta(1) \quad \text{when } n > 1 \end{aligned}$$

   This is in the right form for the Master Theorem, with $a = 1$, $b = 2$, $k = 0$. Since $b^k = 2^0 = 1 = a$, we are in the 'middle case' of the theorem, and we conclude that $T(n) = \Theta(n^k \lg n) = \Theta(\lg n)$ (which agrees with our earlier conclusions).

   (c) Some easy exercises in plugging the relevant numbers into the Master Theorem:

   i. $T(n) = 2T(n/3) + \Theta(n)$: Here $a = 2, b = 3, k = 1$. So $a < b^k$, and we conclude $T(n) = \Theta(n^k) = \Theta(n)$.

   ii. $T(n) = 7T(n/2) + \Theta(n^2)$: Here $a = 7, b = 2, k = 2$. So $a > b^k$, and we conclude $T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807\ldots})$.

[1] Making the call should certainly be counted as a line execution, as there is some work to be done in pushing the relevant information onto the call stack.

[2] Strictly speaking, one should formally justify the implicit assumption here that $T(\lceil n/2 \rceil) \geq T(\lfloor n/2 \rfloor)$ always — but we'll take this as read.

Note: This is actually the recurrence relation arising from Strassen's amazing algorithm for multiplying two $n \times n$ matrices, which is covered in UG3 Algorithms and Data Structures, and which improves asymptotically on the $\Theta(n^3)$ runtime of the obvious method.

iii. $T(n) = 2T(n/4) + \Theta(\sqrt{n})$: Here $a = 2, b = 4, k = 1/2$. So $a = b^k$, and we conclude $T(n) = \Theta(\sqrt{n} \lg n)$.

2. *Draw the heap, and each intermediate state, which is created when we apply the* Max-Heap-Insert *algorithm to the following sequence of elements* {12, 5, 4, 8, 9, 1, 16, 20, 7, 6}. *At each step draw both the tree representation and the contents of the array.*

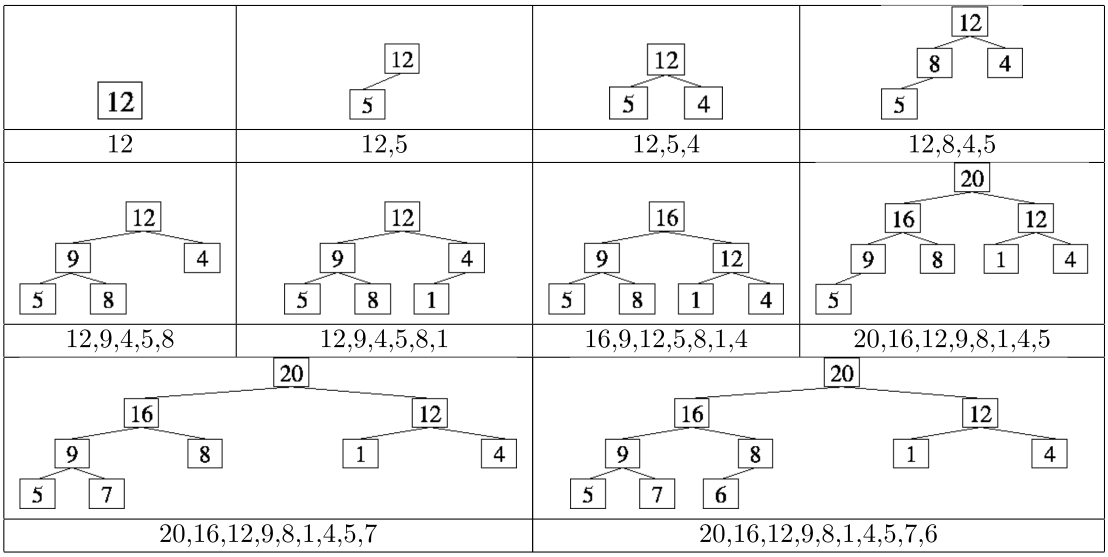

Figure 1: Tree and array representation of the heap from question 1

Answer: We work on the heap, taking each element in the sequence, and temporarily add it to the heap as the new *last node*, then may need to rearrange the tree.

In the figures you can see the situation after one of the Max-Heap-Insert calls. The solutions do not show how do that step-by-step, they just show the resulting heap (tree and array representation). When presenting the solution in class, the steps should be shown explicitly, i.e., how the new element is first placed as a new leaf (and appended to the last position of the array) and then how to rearrange up the heap. For example, when 8 goes in, it first is added as the left child of 5, which at that point is the first available leaf node, and then it gets swapped with its parent 5. The MAX-HEAP-INSERT operation is detailed in the slides for Lecture 11, which can be found here: `https://opencourse.inf.ed.ac.uk/inf2-iads/course-materials/semester-1/schedule`.

Under each figure is written the array representation of the heap. From the heap it is very easy to recover the array representation. In terms of presenting in class, it is probably better to update the array representation together with the tree representation. In particular, for every rearrangement step in the tree, it makes sense to show what happens correspondingly in the array.

3. *Show that when we consider a list of items in sorted order (smallest first) that it will take time $\Omega(n\log(n))$ to insert them into an initially empty heap. Give details of the running-time we will have for each of the individual* Max-Heap-Insert *operations (and why), and then show that the total running-time for this bad case satisfies $\Omega(n\log(n))$.*

   Why does this differ from the $\Theta(n)$ running-time for Build-Max-Heap *on the input array?*

   Answer: The key observation is that if the value of the key of the item being inserted into a heap exceeds the values of all the k items already stored in the heap, then Max-Heap-Insert will take time $\Theta(h) = \Theta(\lg(k))$, where h is the height of the heap (because the new item will need to be swapped all the way up to the root of the heap). So if we insert an increasing sequence of n items, then every item inserted will be larger than all the current heap items, and, for some sufficiently small constant $c > 0$, the total time will be at least

$$\begin{aligned}
\sum_{i=1}^{n} c \cdot \lg(i) &\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} c\lg(i) && \text{drop the smallest } \left\lfloor \frac{n}{2} \right\rfloor \text{terms} \\
&\geq \sum_{i=\lceil \frac{n}{2} \rceil}^{n} c\lg\left(\frac{n}{2}\right) && \text{remaining terms all at least } \frac{n}{2} \\
&\geq \frac{n}{2} c\lg\left(\frac{n}{2}\right)) \in \Omega(n\lg n)\,,
\end{aligned}$$

   so the time taken to insert the list $(1, 2, \ldots, n)$ into an initially empty heap is $\Omega(n\lg n)$.

   Note that this $\Omega(n\lg n)$ lower bound (for the case where items are added in non-decreasing order) is also a lower bound for *worst-case running time* (wrt input size n).

   Build-Max-Heap: Observe that both Max-Heapify and Max-Heap-Insert have $\Theta(h)$ running-time, where h is the height of the relevant heap (the *worst-case running time* of these operations is $\Omega(h)$ as well as $O(h)$). Hence, to contrast the running time of an algorithm that repeatedly calls Max-Heap-Insert against the Build-Max-Heap algorithm, we need to look at the number of times each is called for each heap size.

   Let $h = \lfloor \lg n \rfloor$ be the height of the heap with n elements. While using Build-Max-Heap we have 1 call on a heap of height h, two on heaps of height $h-1$, four on heaps of size $h-2$, and so on until we have $i \in \{1, \ldots, 2^{h-1}\}$ calls on heaps of height 1 (heaps of height 0 do not need Max-Heapify-ing). On the other hand, when inserting $\{1, \ldots, n\}$ we have one call on a heap of height 0, two on heaps of height 1, four on heaps of height 2, and so on until we have $i \in \{1, \ldots, 2^h\}$ calls on heaps of height h. From this we can see the Build-Max-Heap algorithm is organised to ensure that more calls are made on smaller heaps (because it knows all the input in advance of constructing the heap). However, when inserting the items one by one, it is possible that the input is presented in such a way that we do a large number of calls which depend on $\lg(n)$.

4. *This is a discussion question about* `heapq` *in Python and the differences from the classical Heap methods in the book/slides.*

   Answer: You may use `https://docs.python.org/3/library/heapq.html` as a reference. The methods in `heapq` are `heappush(heap, item)`, `heappop(heap)`, `heapify(x)` (to transform the list x into a heap), and `heapreplace(heap, item)`. They also have "private" methods `_siftdown` and `_siftup`.

Indexing is 0-based (so the relationship between the parent and child nodes is slightly different).

Their implementation is based on a min heap, so `heappop(heap)` and `heap[0]` give the minimum element not the largest.

They mention that `heap.sort()` (for their default sorting algorithm of Python)will result in a list/array satisfying the heap property - that is because this is a min heap.

Now the particular methods:

- `heappush` is essentially an implemention of Min-Heap-Insert. It does an append to the list/array and then a call to `_siftdown` with indices 0 and `len(heap)-1`. `siftdown` is the bubbling-up process done by Heap-Insert.

  The running time is $\Theta(1)$ plus the work done by the `_siftdown` call, which will be $O(h)$ for height h as with Min-Heap-Insert.

- `heappop` is an implementation of Heap-Extract-Min (for a Min Heap). It locally copies the min item `heap[0]`, then copies the final element of the heap into this 'top' position, and finally does a call to `_siftup` with index 0 to fix the heap property.

  The running time is $\Theta(1)$ plus the work done by the `_siftup` call, which will be $O(h)$ for height h. In fact `_siftup` works a bit differently to the classic (Min variant of) Heapify, but it will still have the same $\Theta(h)$ running-time for a sub-Heap of height h.

- `heapreplace` is a new method not in our classical set-up - if we want to *both* extract the min and also put in a new item, it makes sense to read `heap[0]` and then replace it with the new item, then a call to `_siftup`.

  This is just an 'optimisation' method for when we do the two operations in order, will do a bit less work overall but still have $\Theta(h)$ running-time.

- `heappushpop` where we plan to first pop and then push implements a shortcut and a call to `_siftup`

  Also an 'optimisation' method for when we do the two operations in order, will do a bit less work overall but still have $\Theta(h)$ running-time.

- `heapify(x)` called on the list x is really is our `Build-Heap`. It runs bottom-up from indices $\lfloor n/2 \rfloor$ down to 0, doing `_siftup(i)` on each such index.

  This will run in $\Theta(n)$ time overall as discussed in the comments in the source file, i.e., the same time as `Build-Heap`.

There are also `_max` variants of some of these methods, to operate on a max heap. However not all methods have a 'Max' variant, for example there is no `heappush_max`.

There are also two 'private' methods `_siftdown` and `_siftup`.

- `_siftdown` is a method which does the 'bubbling up" part of our Max-Heap-Insert. It is a bit more general than the bubbling-up of Max-Heap-Insert as it has an index parameter to mark the limit of the bubbling (don't necessarily) go all the way to the top.

- `_siftup` is a method which has the same effect as (a Min variant of) `Heapify`. It is called at a node/index *pos* whose two child sub-Heaps are true heaps, but where the item at *pos* breaks the rules, and then it fixes everything to satisfy the Heap property from *pos* down.

It works a bit different to Heapify, however the asymptotic running time is also $\Theta(h)$ for a sub-Heap of height h.