Binary Heaps (1): Homework Solutions

Emanuele Ballarin

July 14, 2020

Exercises 1, 2, 3

Text:
Implement the array-based representation of binary heap together with the functions `HEAP_MIN`, `REMOVE_MIN`, `HEAPIFY`, `BUILD_HEAP`, `DECREASE_KEY` *and* `INSERT_VALUE`.
Implement an iterative version of `HEAPIFY`.
Test the implementation on a set of instances of the problem and evaluate the execution time.

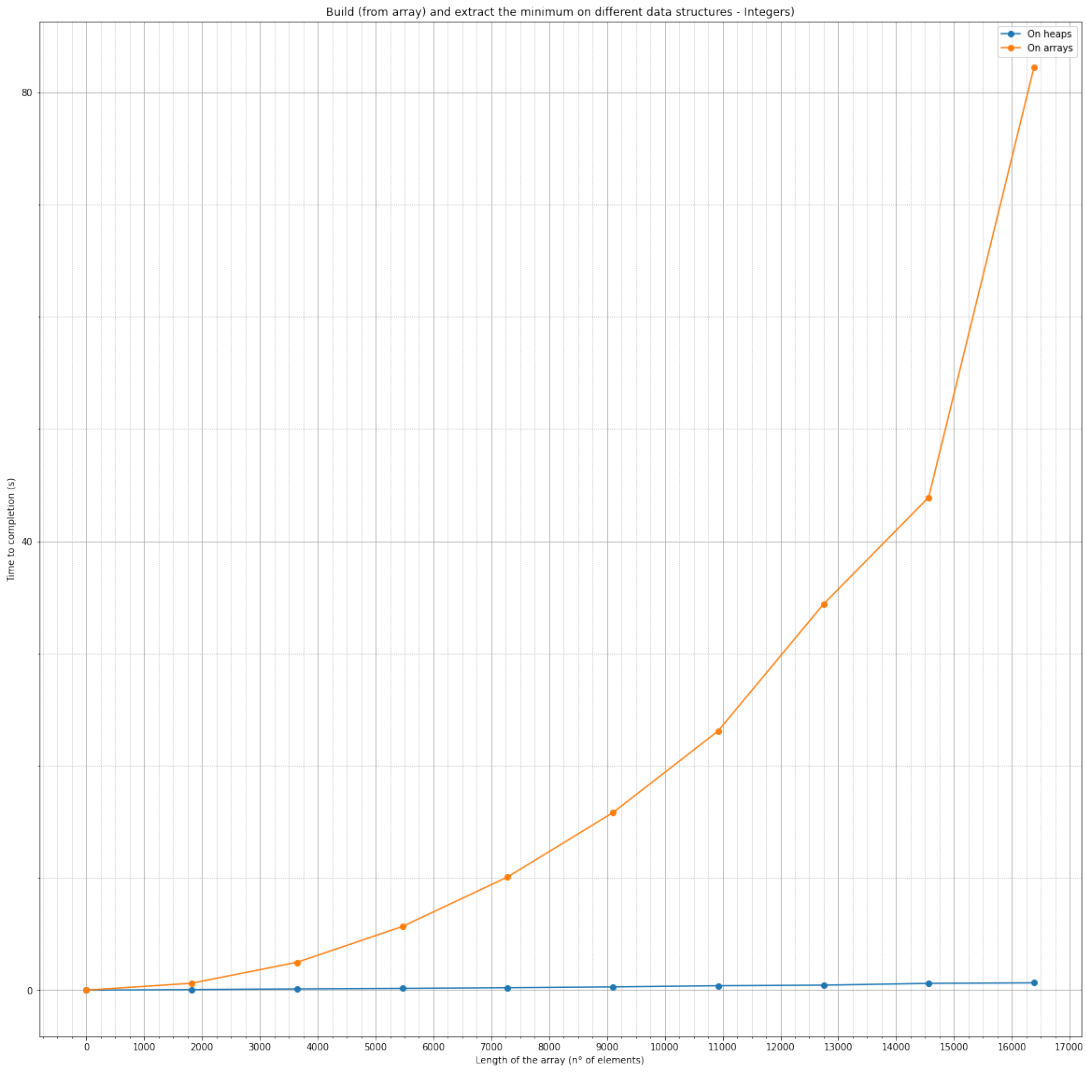
Solution:
These exercises have been originally tackled during some lectures in the form of *live-coding* sessions. Very minor modifications may be present w.r.t. the original implementation, but no major architectural change has been performed.

The testing tools provided have been used in order to ensure the correctness of the final implementation, and to benchmark the resulting data structure (*an array-based binary heap* indeed) against a simple vector managed via subsequent linear scans.

In particular, a version of the `test_insert` binary which exposes also the underlying array structure has been used to interactively check that every function implemented worked as expected. The `extract_min` binary, instead, has been used for bench-marking, as a representative of some of the most popular uses of a heap: *priority queue* emption (as used e.g. in some optimized implementations of Dijkstra's algorithm) and *heapsorting*.

Benchmarks:

As we can see from the graph, even at smaller sizes, the *heap-based* implementation appears to be much faster than the *array-based* one. Indeed, this strongly highlights the effectiveness of using heaps for tasks that require iterated extraction of the minimum (or maximum) element w.r.t. a *total order* from such data structure.



Exercise 4

Text:
Show that, with the array representation, the leaves of a binary heap containing n nodes are indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Solution:
To begin our proof, let us recall that – for any *heap*, in general – a *leaf* is a node which has no children. In addition – specifically for *heaps* represented in accordance to the *array-based representation* – given a node whose key is stored in the i-th position of the array, its childred (if any) will have keys stored in the $2i$-th (left) and $(2i + 1)$-th (right) position of the array.
As a direct consequence of the latter, in *array-represented heaps*, nodes belonging to the same level of the *heap* will have keys stored contiguously in the array.

Since *heaps* are – by definition – *trees complete at least up to the second-last level*, and *leaves* must occupy the last level, our proof can be recast into proving that:

1. Node indexed by $\lfloor n/2 \rfloor + 1$ is a leaf;
2. Node indexed by n is a leaf;
3. Node indexed by $\lfloor n/2 \rfloor$ is not a leaf.

To prove points 1 and 2 it is sufficient to show that children of node indexed by $\lfloor n/2 \rfloor + 1$ will have (if any) index $\geq 2(\lfloor n/2 \rfloor + 1) > n$, and those of node indexed by n (if any) will have index $\geq 2n > n$.

In both cases, children (if any) would have indexes which are greater than the size of the heap, and for that reason should not exist. This proves that nodes indexed by $\lfloor n/2 \rfloor + 1$ and n are leaves.

As far as point 3 is concerned, it is sufficient to notice that node indexed by $\lfloor n/2 \rfloor$ should necessarily have at least one children. Indeed it will have:

- the node indexed by n as a left child (if n even);
- the node indexed by $n - 1$ as a left child and that indexed by n as a right child (if n odd).

This concludes the proof.

Exercise 5

Text:
Show that the worst-case running time of `MAX_HEAPIFY` *on a heap of size n is $\Omega(\log_2 n)$. (Hint: For a heap with n nodes, give node values that cause*

`MAX_HEAPIFY` *to be called recursively at every node on a simple path from the root down to a leaf).*

Solution:
As suggested, we will start with an example. Let us consider – before the call to `MAX_HEAPIFY` – the *heap* (though the *heap property* still needs to be enforced) having the smallest (and only such) key at the root-node.

In order to enforce the *heap property*, `MAX_HEAPIFY` needs to swap keys as many times as levels in the *heap*, in order to place the smallest key in the leftmost leaf. Evidently, no other situation can produce a higher number of swaps.

Since comparisons and swaps have constant time-complexity ($\Theta(1)$), by calling h the number of levels in the heap (or, its *height*), the overall `MAX_HEAPIFY` time-complexity will be $\Theta(h)$.

Since the number of levels in a *heap* made up of n nodes is $log_2(n)$, the overall `MAX_HEAPIFY` time-complexity will be $\Theta(log_2(n)) = O(log_2(n)) \cap \Omega(log_2(n))$. This concludes our proof.

Exercise 6

Text:
Show that there are at most $\lceil n/2^{h+1} \rceil$ *nodes of height* h *in any* n*-element binary heap.*

Solution:
Let us recall that – as shown above – in any *binary heap* containing exactly n nodes there are $\lceil n/2 \rceil$ leaves.

We then proceed by induction.

As the *base case*, we show that – for $h = 0$, i.e. the leaf-nodes level – there are indeed $\lceil n/2 \rceil = \lceil n/2^{0+1} \rceil = \lceil n/2^{h+1} \rceil$ nodes.

If we remove the leaf-nodes level from a *heap* of n total elements, we obtain a new tree with $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ elements. Furthermore, all levels of such resulting tree correspond to the next level in the original tree.

If we apply the inductive hypothesis to such resulting tree – at level $h - 1$, i.e. level h of the original tree –, we obtain that the number of nodes in the h-level of the original tree is at most $\lceil \frac{\lfloor n/2 \rfloor}{2^h} \rceil < \lceil \frac{n/2}{2^h} \rceil = \lceil \frac{n}{2^{h+1}} \rceil$. This concludes our proof.