Introduction to Algorithms and Data Structures

Lecture 12: BuildHeap and HeapSort

Mary Cryan

School of Informatics
University of Edinburgh

29th October, 2024

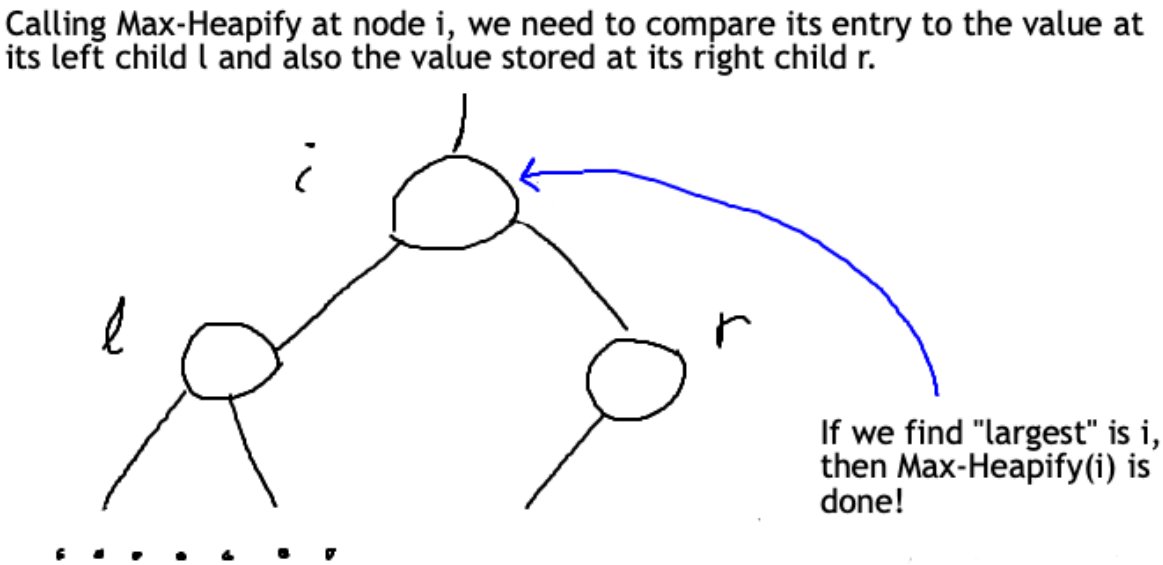
Heap operations and their asymptotic complexity

Parent(i)	$O(1)$
Left(i)	$O(1)$
Right(i)	$O(1)$
Heap-Maximum(A)	$O(1)$
Max-Heapify(A, i)	$\Theta(\lg(n))$
Heap-Extract-Max(A)	$\Theta(\lg(n))$
Max-Heap-Insert(A, k)	$\Theta(\lg(n))$
Heap-Increase-Key(A, i, k)	$\Theta(\lg(n))$
Build-Heap(A)	$\Theta(n)$

Max-heapify

The "top-level" work of Max-Heapify(i) depends on comparisons between the value at i and the values at i's two child nodes. $\Theta(1)$ time to do these comparisons.

The easy case:

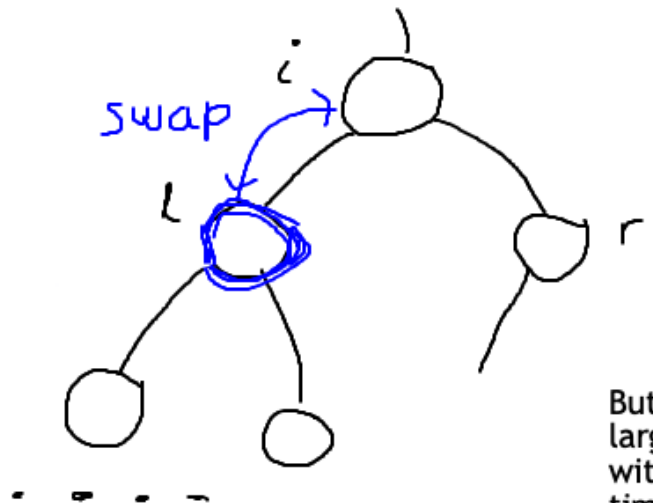


Max-heapify

The "top-level" work of Max-Heapify(i) depends on comparisons between the value at i and the values at i's two child nodes. $\Theta(1)$ time to do these comparisons.

The recursive case:

Calling Max-Heapify at node i, we need to compare its entry to the value at its left child l and also the value stored at its right child r.



But if one/more of i's child nodes have a larger value, then we must swap i's value with the "largest" (say l in this), in O(1) time .... then follow-up with a recursive call from "largest".

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.
- Then (if some change happened) we call Max-Heapify$(A, \mathit{largest})$ on the child node $\mathit{largest}$.

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.
- Then (if some change happened) we call Max-Heapify$(A, \mathit{largest})$ on the child node $\mathit{largest}$.

Let $T'_{\text{Max-Heapify}}(h)$ be running-time on a Heap of height h.

Clearly

$$T'_{\text{Max-Heapify}}(h) \leq \begin{cases} T'_{\text{Max-Heapify}}(h-1) + O(1) & h \geq 1 \\ O(1) & h = 0 \end{cases}$$

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.
- Then (if some change happened) we call Max-Heapify$(A, \mathit{largest})$ on the child node $\mathit{largest}$.

Let $T'_{\text{Max-Heapify}}(h)$ be running-time on a Heap of height h.

Clearly

$$T'_{\text{Max-Heapify}}(h) \leq \begin{cases} T'_{\text{Max-Heapify}}(h-1) + O(1) & h \geq 1 \\ O(1) & h = 0 \end{cases}$$

So $T'_{\text{Max-Heapify}}(h)$ is $(h+1) \cdot O(1)$.

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.
- Then (if some change happened) we call Max-Heapify$(A, \mathit{largest})$ on the child node $\mathit{largest}$.

Let $T'_{\text{Max-Heapify}}(h)$ be running-time on a Heap of height h.

Clearly

$$T'_{\text{Max-Heapify}}(h) \leq \begin{cases} T'_{\text{Max-Heapify}}(h-1) + O(1) & h \geq 1 \\ O(1) & h = 0 \end{cases}$$

So $T'_{\text{Max-Heapify}}(h)$ is $(h+1) \cdot O(1)$.

Hence $T_{\text{Max-Heapify}}(n)$ is $\lg(n) \cdot O(1)$, ie $O(\lg(n))$.

Running time of Max-Heapify

Max-Heapify(A, i):

- $\Theta(1)$ work to consider i and its two child nodes and to swap the maximum of those 3 values into node/index i.
- Then (if some change happened) we call Max-Heapify$(A, \mathit{largest})$ on the child node $\mathit{largest}$.

Let $T'_{\text{Max-Heapify}}(h)$ be running-time on a Heap of height h.

Clearly

$$T'_{\text{Max-Heapify}}(h) \leq \begin{cases} T'_{\text{Max-Heapify}}(h-1) + O(1) & h \geq 1 \\ O(1) & h = 0 \end{cases}$$

So $T'_{\text{Max-Heapify}}(h)$ is $(h+1) \cdot O(1)$.

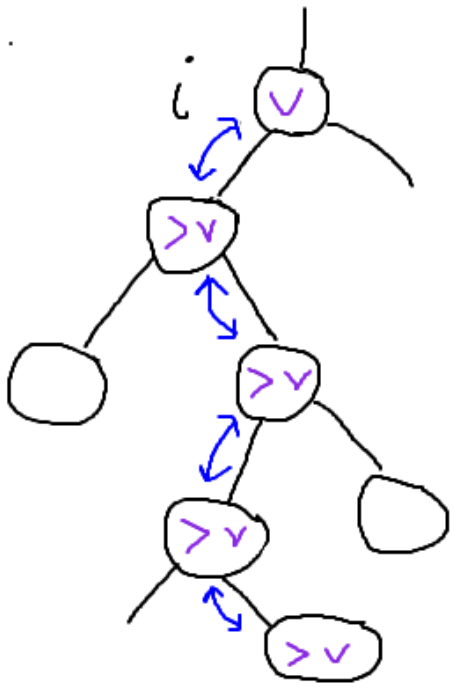
Hence $T_{\text{Max-Heapify}}(n)$ is $\lg(n) \cdot O(1)$, ie $O(\lg(n))$.

Note we don't need/use the Master theorem here ..., and it would not fit.

Running time of Max-Heapify

For the lower bound on $T_{\text{Max-Heapify}}(n)$, again work wrt height and $T'_{\text{Max-Heapify}}(h)$.

Note we may end up making all h recursive calls:



If there is a downwards path from i with values > v all the way (v the value at i) then Max-Heapify will get called recursively all the way down to the Leaf.

Are doing $\Omega(1)$ work each step, so get $h \cdot \Omega(1)$, ie $\Omega(h)$.
If i is the root, then $h \geq \lg(n) - 1$, so we have $T_{\text{Max-Heapify}}(n) = \Omega(\lg(n))$

Running times of related operations

The other $\Theta(\lg(n))$ operations ...

Heap-Extract-Max This is just a swap of the Maximum with the rightmost leaf position $\Theta(1)$, a decrement of the Heap size ($\Theta(1)$) and a call to Heapify ($\Theta(\lg(n))$).

Note $\Omega(\lg(n))$ can really happen (explain).

Max-Heap-Insert We "bubble-up" through the Heap. $\Theta(1)$ work done at each step, so $O(\lg(n))$ overall.

We have $\Omega(\lg(n))$ for the worst-case - say we insert a new item bigger than anything in the Heap.

Heap-Increase-Key This operation is similar to Max-Heap-Insert, and can use same arguments.

Build-Max-heap

Algorithm Build-Max-Heap(A)

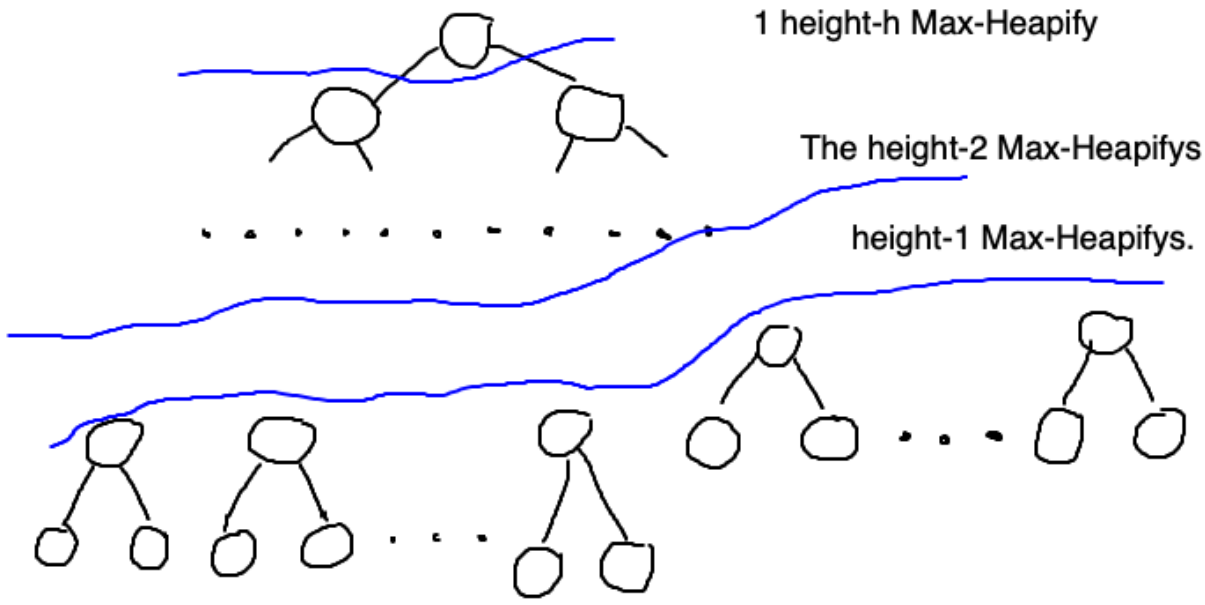
1. $A.heap\_size \leftarrow A.length$
2. **for** $i \leftarrow \lfloor A.heap\_size/2 \rfloor$ -1 **downto** 0
3. Max-Heapify(A, i)

$A.length$ is our usual "n".
Build-Max-Heap builds a Max Heap from an unsorted array in $O(n)$ time.
To show correctness, our invariant will be

> *(Inv) At step i, we are guaranteed that for every node/index j with $j > i$, the subtree-at-j is a legal Max Heap.*

At step i, Left(i) and Right(i) are guaranteed to be legal Max Heaps. So conditions for Max-Heapify(A, i) are satisfied. Hence after calling Max-Heapify(A, i) in line 3, *(Inv)* now holds for $i - 1$.
Base case: all indices $\lfloor A.length/2 \rfloor, \ldots A.length - 1$ are leaves, so trivially Heaps.

Build-Max-Heap



- We will start with the height-1 Heaps at the bottom first, then the heaps of height 2, ... finally the one Heap of height h
- So we always satisfy the "pre-condition" for Max-Heapify and its Left and Right sub-Heaps.

Analysis of Build-Max-Heap

We will do $\lfloor A.length/2 \rfloor$ calls to Max-Heapify.

But these are on Heaps of varying heights ...

1 call of height h (root) ...
2 calls of height $h-1$ ...
...
$\lceil n/2^{\ell+1} \rceil$ calls of height ℓ.
So

$$T_{Build-Max-Heap}(n) = \sum_{\ell=1}^{\lfloor \lg(n) \rfloor} \left\lceil \frac{n}{2^{\ell+1}} \right\rceil O(\ell)$$

$$= O\left(\sum_{\ell=1}^{\lfloor \lg(n) \rfloor} \frac{n}{2^{\ell+1}} \ell \right) = O\left(\frac{n}{2} \cdot \sum_{\ell=1}^{\lfloor \lg(n) \rfloor} \frac{\ell}{2^{\ell}} \right) = O(n).$$

(in the last step we use A.8 of the book, with $x = 1/2$, to show $\sum_{\ell=1}^{\lfloor \lg(n) \rfloor} \frac{\ell}{2^{\ell}} < 2$)

HeapSort

In a Heap, we are missing "total order" information about the items, but we do know where the maximum is, and can retrieve it efficiently. SO:

Heap-Extract-Max the max item (*heap gets rearranged*) ...
Heap-Extract-Max the next-max item (*heap gets rearranged again*) ...

Algorithm HeapSort(A)

1. $n \leftarrow A.length$
2. Build-Max-Heap(A)
3. **for** $i \leftarrow n-1$ **downto** 0
4. $\quad v \leftarrow$ Heap-Extract-Max(A)
5. $\quad A[A.heap\_size] \leftarrow v$

Each time Heap-Extract-Max is called (and the current maximum is deleted) $A.heap\_size$ gets decremented again. So we copy the extracted value into the index $A.heap\_size$ (this is where this "next max" belongs in the sorted order).

Analysis of HeapSort

(forget about the time for the Build-Max-Heap call as that is only $\Theta(n)$)
For the rest of the work, like Build-Max-Heap we are doing a linear number of Max-Heapify (well, Heap-Extract-Max) operations ...

But unlike Build-Max-Heap every single one of the Max-Heapify calls (from Heap-Extract-Max) is carried out from the root ...

Upper bound is therefore

$$\begin{aligned}
\sum_{m=n}^{2} O(\lg(m)) &= O\left(\sum_{m=1}^{n} \lg(m)\right) \\
&= O\left(\lg(\prod_{m=1}^{n} m)\right) \qquad \text{(by } \lg(a.b) = \lg(a) + \lg(b)\text{)} \\
&= O(\lg(n!))
\end{aligned}$$

Easy inequality $n^{n/2} \leq n! \leq n^n$ gives us $O(n \lg(n))$.

To show $\Omega(n \lg(n))$ direction consider an input array in reverse sorted order.

HeapSort: some properties

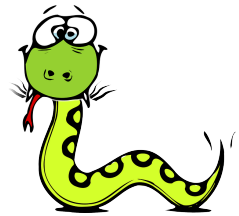
HeapSort (and the binary Heap itself) were invented by J.W.J. (Bill) Williams, in 1964.

HeapSort is an in-place Algorithm, as it can be implemented without any auxiliary "scratch" array.

A sorting algorithm is said to be stable if for any input array A, we are guaranteed that any two entries of A with the same key will appear in the same relative order after sorting.

- HeapSort is not stable.

Python and Heaps



Python provides the `heapq` Library:

- Their implementation is a Min Heap, rather than a Max heap
  - Not hard to re-work it to be a Max Heap, can store the items with negative values (if they are `int` or `float` or similar).
  - For more complicated types, can instead add a "wrapper" class wrapping them in a class with an inverted `__lt__` operator.
- arrays start at 0 (like us)
- Names of operations are different (their Heapify is really Build-Max-Heap, lots of other differences).

Reading Material

From [CLRS]:

- BuildHeap is pressented and analysed in 6.1-6.3 of [CLRS] (eds 3 and 4)
- Heapsort and its running time is 6.4 of [CLRS] (eds 3 and 4).

"Algorithms Illuminated" [Al] by Roughgarden:

- Sections 10.2, 10.3 ... plus Problem 10.5 (Heapify/BuildHeap)