Chapter 6

Heaps.

Heap is the first data structure exhibiting a non-trivial algorithmic mechanism. It allows extracting the min (max) value at a logarithmic time cost. It plays a crucial role in finding shortest-path algorithms, which we will see at the end of the course. In this recitation we will:

1. Outline heaps in brief.
2. Practice correction proving by proving the correction of heap-sort and heapify-down.
3. Present median-heap, an application of heap to maintaining dynamically the median value.

6.1 Review.

Definition 6.1.1. Consider the sequence $H = H_1, H_2, \cdots, H_n \in \mathbb{R}$. We will say that H is a min-Heap if for every $i \in [n]$, we have that $H_i \le H_{2i}, H_{2i+1}$[1].

Max-Heap is defined the same way, but by flipping the directions of the inequalities.

That definition is equivalent to the following recursive definition: Consider an almost complete binary tree where each node is associated with a number. Then, we will say that this binary tree is a heap if the root's value is lower than its children's values, and each subtree defined by its children is also a heap. There is a one-to-one mapping between these definitions by setting the array elements on the tree in order.

Checking vital signs. Are the following sequences are heaps?

1. 1,2,3,4,5,6,7,8,9,10 (Y)
2. 1,1,1,1,1,1,1,1,1,1 (Y)
3. 1,4,3,2,7,8,9,10 (N)
4. 1,4,2,5,6,3 (Y)

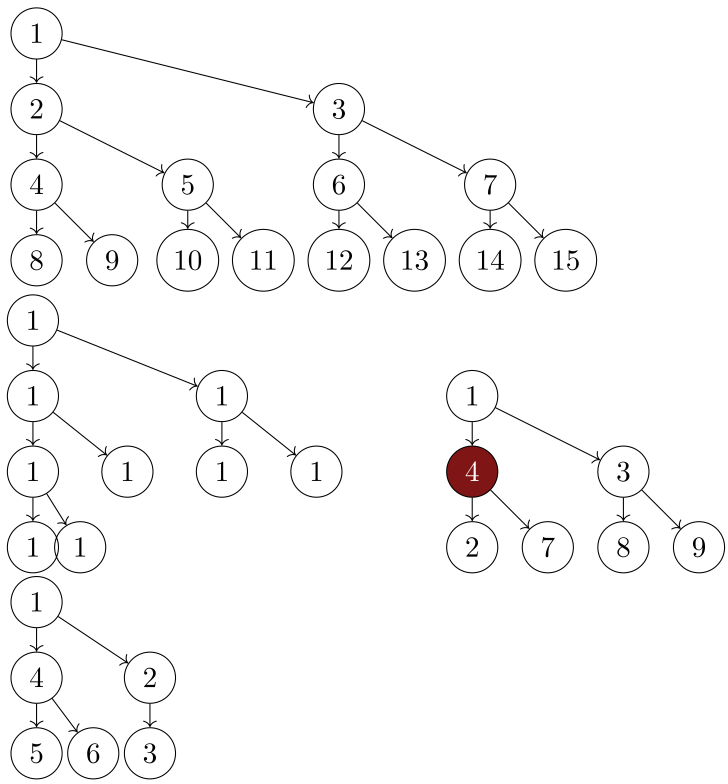

Figure 6.1: The trees representations of the heaps above. The node which fails to satisfy the Heap inequality is colorized.

Remark 6.1.1. Why do we[2] stick to the array representation of heaps when the binary tree representation might seem more convenient? The reason is that the array representation exhibits an in-place sorting algorithm, namely a sorting algorithm that does not allocate additional memory for the target input.

6.2 Heapsort.

We will start by introducing the heap-sort algorithm and providing a proof of its correctness, assuming the correctness of the heapify-down. The heapify-down assumes it is over the root of two heaps, the root itself might not satisfy the heap inequality, and in that case, it is bubbled down until the array becomes a heap. The heap-sort is given in Algorithm 1.

Correctness. We are going to prove the following statement. For convenience, we use the following notation: $A_i^{(t)}$ is the ith item of A at iteration t immediately after line (3), and $A_i'^{(t)}$ is the ith item of A at iteration t after line (5) (or equivalently at the beginning of the $t+1$ iteration). Observe that by the definition of the algorithms, line (3) touches only the 1st and the $(n-t-1)$th elements, so $A_j'^{(t-1)} = A_j^{(t)}$ for any $j \neq 1, n-t-1$.

---

[1] When we think of the values at indices greater than n as $H_{i>n} = -\infty$

[2] Here: we = any entity that teaches data structure, not the HUJI course in particular.

```
1 A ← Build-Heap(A)
2 for i ∈ [n] do
3     swap A_1 ↔ A_{n−i+1}
4     heapsize(A) ← n − i
5     heapify(A, 1)
6 end
7 return A
```

Algorithm 1: Heap-sort(A)

Claim 6.2.1. *At the end of the ith iteration:*

1. $A'^{(i)}_{n-i+1}, A'^{(i)}_{n-i+2}, ..A'^{(i)}_{n}$ *are the i largest elements of A placed in order*
2. $A'^{(i)}_{1}, A'^{(i)}_{2}, ..A'^{(i)}_{n-i}$ *is a maximum heap.*

Proof. By induction.

1. Base.
   - $A^{(1)}_n$ is set in line (3) to be the root of the heap and therefore is the maximum of A. Since, at line (4), we decrease the length size of the heap, and by the correctness of heapify, we have that $A^{(1)}_n$ is untouched when executing heapify-down, or in other words, $A'^{(1)}_n = A^{(1)}_n = A^{(0)}_1 = \max A$, so we have the first part of the claim.
   - Since earlier in line (5), any other element but $A^{(0)}_1, A^{(0)}_n$ was untapped, we have that for any $j \in [2, n-1]$, $A^{(1)}_j = A'^{,(0)}_j$. Thus, for any $j \in [2, n-1]$, A_j satisfies the heap inequality, or in other words, $A^{(1)}_1$ is a root of two heaps.

     Hence, by the correctness of the heapify-down algorithm and the decreasing of the heap length at line (4), we have that $A'^{(1)}_1, A'^{(1)}_2, \ldots, A'^{(1)}_{n-1}$ is a heap, and we have the correctness of the second part of the claim.
2. Assumption. Assume the correctness of the claim for any $i' < i$.
3. Step. Consider the ith iteration.
   - By the first part of the induction assumption, $A'^{(i-1)}_1$ is a root of the heap $A'^{(i-1)}_1, A'^{(i-1)}_2, ..A'^{(i-1)}_{n-i+1}$ and therefore is their maximum.

     So after the swapping in line (3), we get that $A^{(i)}_{n-i+1}$ is the element which is greater than $n - i$ elements in A.

     By using the second part of the induction assumption, we know that it is also less than $A'^{(i-1)}_{n-i+2}, A'^{(i-1)}_{n-i+3}, ..A'^{(i-1)}_{n}$, so after line (3) and by that $A'^{(i-1)}_{n-i+2}, A'^{(i-1)}_{n-i+3}, ..A'^{(i-1)}_{n}$ are being the $i-1$ largest elements placed in order, we have that:

     $$A^{(i)}_{n-i+1}, A^{(i)}_{n-i+2}, A^{(i)}_{n-i+3}, ..A^{(i)}_{n} = A^{(i)}_{n-i+1}, A'^{(i-1)}_{n-i+2}, A'^{(i-1)}_{n-i+3}, ..A'^{(i-1)}_{n}$$

     are the i largest elements placed in order.

- Similarly to the base case, line (4) confines the 'heap' into $A^{(i)}_1, \dots, A^{(i)}_{n-i}$. Any element $A'^{(i-1)}_j$, such that $j \neq 1, n-i+1$, is untouched by line (3), and therefore $A^{(i)}_j = A'^{(i-1)}_j$ for any $j \in [2, n-i]$. Thus, all the elements in the range satisfy the heap inequality, and by the correctness of heapify-down, we get that $A'^{(i)}_1, \dots, A'^{(i)}_{n-i}$ is a heap.

□

6.3 Heapify-Down.

We are now ready to prove correction of the bubbling routine.

input: Array $a_1, a_2, ..a_n$
next $\leftarrow i$
left $\leftarrow 2i$
right $\rightarrow 2i + 1$
if *left* $< n$ *and* $H_{left} < H_{next}$ **then**
    next $\leftarrow$ left
end
if *right* $< n$ *and* $H_{right} < H_{next}$ **then**
    next $\leftarrow$ right
end
if $i \neq$ *next* **then**
    $H_i \leftrightarrow H_{\text{next}}$
    Heapify-down(next)
end

Algorithm 2: Heapify-down

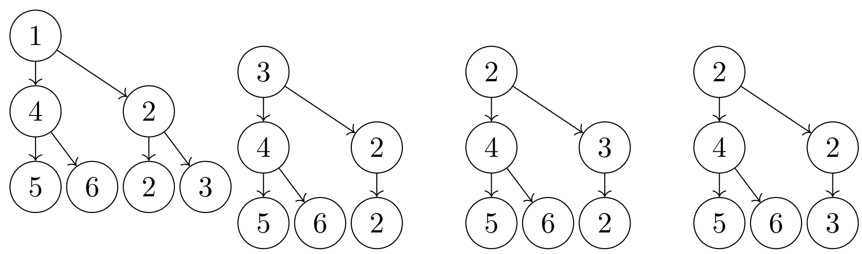

Figure 6.2: Running Example, Extract.

Claim 6.3.1. *Assume that H satisfies the Heap inequality for all the elements except the root. Namely for any $i \neq 1$ we have that $H_i \leq H_{2i}, H_{2i+1}$. Then applying Heapify-down on H at index 1 returns a heap.*

Proof. By induction on the heap size.

- Base, Consider a heap at the size at most three and prove for each by considering each case separately. (lefts as exercise).

- Assumption, assume the correctness of the Claim for any tree that satisfies the heap inequality except the root, at size $n' < n$.

- Induction step. Consider a tree at size n which and assume w.l.g (why could we?) that the right child of the root is the minimum between the triple. Then, by the definition of the algorithm, at line 9, the root exchanges its value with its right child. Given that before the swapping, all the elements of the heap, except the root, had satisfied the heap inequality, we have that after the exchange, all the right subtree's elements, except the root of that subtree (the original root's right child) still satisfy the inequality. As the size of the right subtree is at most $n-1$, we could use the assumption and have that after line (10), the right subtree is a heap.

  Now, as the left subtree remains the same (the values of the nodes of the left side didn't change), we have that this subtree is also a heap. So it's left to show that the new tree's root is smaller than its children's. Suppose that is not the case, then it's clear that the root of the right subtree (heap) is smaller than the new root. Combining the fact that its origin must be the right subtree, we have a contradiction to the fact that the original right subtree was a heap (as its root wasn't the minimum element in that subtle).

$\square$

6.4 Example, Median Heap

Task: Write a datastructure that support insertion and deltion at $O(\log n)$ time and in addition enable to extract the median in $O(\log n)$ time.

Solution. We will define two separate Heaps, the first will be a maximum heap and store the first $\lfloor n/2 \rfloor$ smallest elements, and the second will be a minimum heap and contain the $\lceil n/2 \rceil$ greatest elements. So, it's clear that the root of the maximum heap is the median of the elements. Therefore to guarantee correctness, we should maintain the balance between the heap's size. Namely, promising that after each insertion or extraction, the difference between the heap's size is either 0 or 1.

input: Array $H_1, H_2, ..H_n, v$

if $H_{\max,1} \leq v \leq H_{\min,1}$ **then**
    if *size* $(H_{\max})-$ *size*$(H_{\min}) = 1$ **then**
        Insert-key (H_{min}, v)
    end
    else
        Insert-key (H_{max}, v)
    end
end
else
    median $\leftarrow$ Median-Extract H
    if $v <$ *median* **then**
        Insert-key (H_{max}, v)
        Insert-key ($H_{min}, median$)
    end
    else
        Insert-key (H_{min}, v)
        Insert-key ($H_{max}, median$)
    end
end

Algorithm 3: Median Insert key.

input: Array $H_1, H_2, ..H_n$

median $\leftarrow$ extract H_{max}
if *size*(H_{min}) - *size*$(H_{max}) > 0$ **then**
    temp $\leftarrow$ extract $H_{\min}$
    Insert-key (H_{max}, temp)
end
return median

Algorithm 4: Median-Extract.

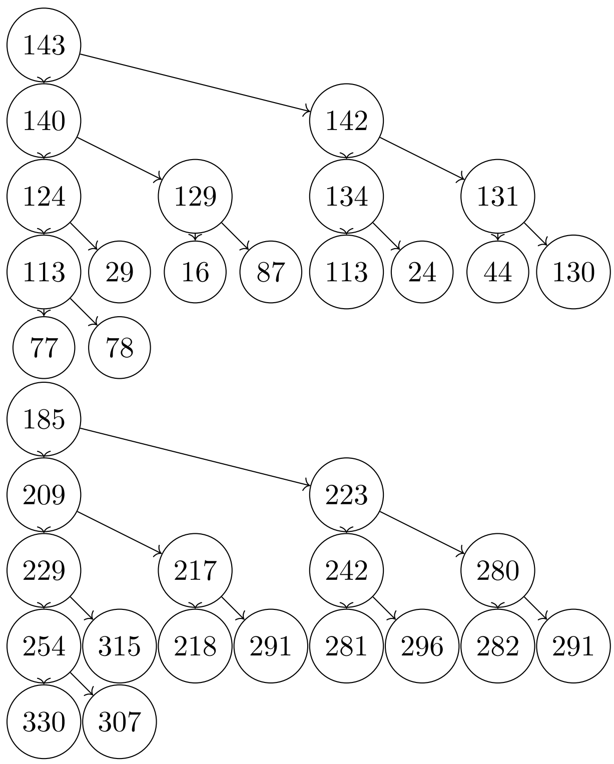

Figure 6.3: Example for Median-Heap, the left and right trees are maximum and minimum heaps.