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.

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 $H_i \leq H_{2i}, H_{2i+1}$[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)

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.

---

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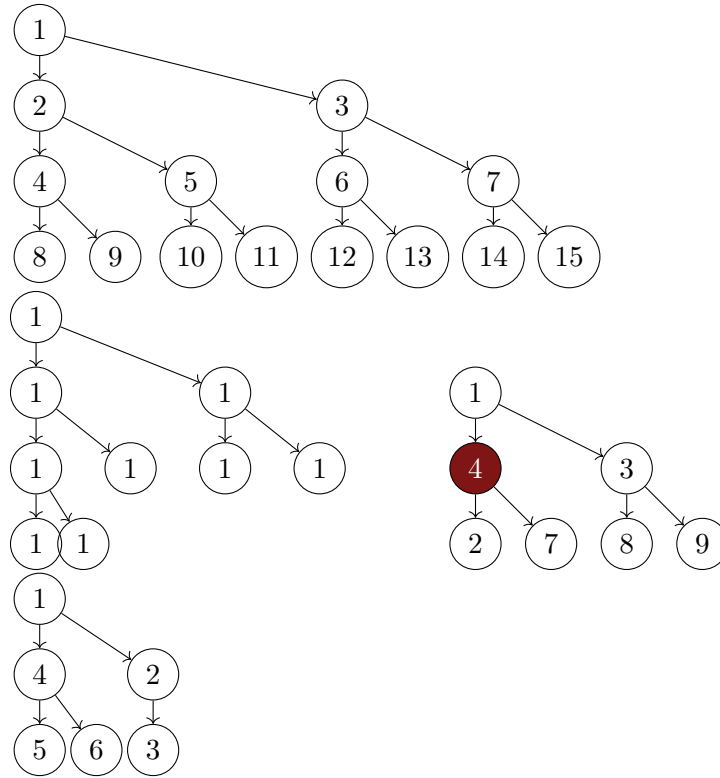
Figure 6.1: The trees representations of the heaps above. The node which fails to satisfy the Heap inequality is colorized.

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.

1 $A \leftarrow \text{Build-Heap}(A)$
2 **for** $i \in [n]$ **do**
3 | swap $A_1 \leftrightarrow A_{n-i+1}$
4 | heapsize(A) $\leftarrow n - i$
5 | heapify($A, 1$)
6 **end**
7 return A

Algorithm 1: Heap-sort(A)

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 $j \neq 1, n - t - 1$.

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

1. *$A_{n-i+1}^{\prime(i)}, A_{n-i+2}^{\prime(i)}, ..A_n^{\prime(i)}$ are the i largest elements of A placed in order*

2. *$A_1^{\prime(i)}, A_2^{\prime(i)}, ..A_{n-i}^{\prime(i)}$ is a maximum heap.*

Proof. By induction.

1. Base.

   - $A_n^{(1)}$ 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 $A_n^{(1)}$ is untapped, or in other words, $A_n^{\prime(1)} = A_n^{(1)} = A_1^{(0)} = \max A$, so we have the first part of the claim.

   - Since earlier in line (5), any other element but $A_1^{(0)}, A_n^{(0)}$ was untapped, we have that for any $j \in [2, n-1]$, $A_j^{(1)} = A_j^{\prime;(0)}$. 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 $A_1^{\prime(1)}, A_2^{\prime(1)}, \ldots, A_{n-1}^{\prime(1)}$ 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, $A_1^{\prime(i-1)}$ is a root of the heap $A_1^{\prime(i-1)}, A_2^{\prime(i-1)}, ..A_{n-i+1}^{\prime(i-1)}$ and therefore is their maximum.

     So after the swapping in line (3), we get that $A_{n-i+1}^{(i)}$ 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 $A_{n-i+2}, A_{n-i+3}, ..A_n$, so after line (3) and by that $A_{n-i+2}^{prime(i-1)}, A_{n-i+3}^{prime(i-1)}, ..A_n^{prime(i-1)}$ are the $i - 1$ largest elements placed in order, we have that:

     $$A_{n-i+1}^{(i)}, A_{n-i+2}^{(i)}, A_{n-i+3}^{(i)}, ..A_n^{(i)} = A_{n-i+1}^{(i)}, A_{n-i+2}^{\prime(i-1)}, A_{n-i+3}^{\prime(i-1)}, ..A_n^{\prime(i-1)}$$

     are the i largest elements placed in order.

   - Similarly to the base case, line (4) confines the 'heap' into $A_1^{(i)}, \ldots, A_{n-i}^{(i)}$. Any element $A_j^{\prime(i-1)}$, such that $j \neq 1, n - i + 1$, is untouched by line (3), and therefore $A_j^{(i)} = A_j^{\prime(i-1)}$ 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 $A_1^{\prime(i)}, \ldots, A_{n-i}^{\prime(i)}$ is a heap.

   $\square$

6.3   Heapify-Down.

How much is the cost (running time) to compute the min of H? (without changing the heap). ($O\left(1\right)$). Assume that option 4 is our Superpharm Line. Let's try to imagine how we should maintain the line. After serving the customer at the top, what can be said on $\{H_2, H_3\}$? or $\{H_{i>3}\}$? (the second highest value is in $\{H_2, H_3\}$.)

Subtask: Extracting Heap's Minimum.   *Let H be an Heap at size n, Write algorithm which return H_1, erase it and returns H', an Heap which contain all the remain elements.* **Solution:**

    input: Heap $H_1, H_2, ..H_n$
1 ret $\leftarrow H_1$
2 $H_1 \leftarrow H_n$
3 $n \leftarrow n - 1$
4 Heapify-down(1)
5 return ret

    input: Array $a_1, a_2, ..a_n$
1 next $\leftarrow i$
2 left $\leftarrow 2i$
3 right $\rightarrow 2i + 1$
4 **if** *left $< n$ and $H_{left} < H_{next}$* **then**
5   │   next $\leftarrow$ left
6 **end**
7 **if** *right $< n$ and $H_{right} < H_{next}$* **then**
8   │   next $\leftarrow$ right
9 **end**
10 **if** *$i \neq$ next* **then**
11   │ $H_i \leftrightarrow H_{\text{next}}$
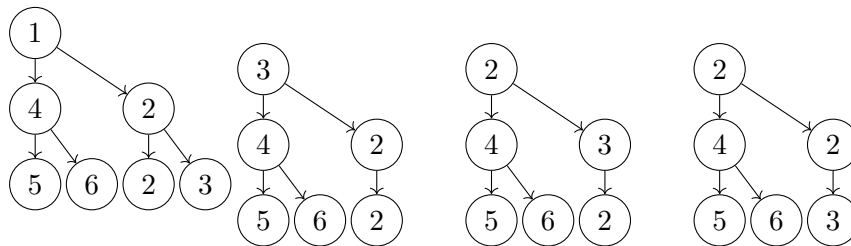12   │   Heapify-down(next)
13 **end**

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).

  □

6.4   Heap Construction.

How to construct a heap? And how much time will it take?

> **input:** Array $H = H_1..H_n$
>
> **1** $i \leftarrow \lfloor \frac{1}{2}n \rfloor$
> **2** **while** $i > 1$ **do**
> **3** $\quad$ Heapify-down (H, i)
> **4** $\quad$ $i \leftarrow i - 1$
> **5** **end**
> **6** return $H_1..H_n$

Algorithm 2: Build a heap.

We can compute a simple upper bound on the running time of Build as follows. Each call to Heapify costs $O(\log n)$ time, and Build makes $O(n)$ such calls. Thus, the running time is $O(n \log n)$. This upper bound, though correct, is not as tight as it can be.

Let's compute the tight bound. Denote by U_h the subset of vertices in a heap at height h_i. Also, let $c > 0$ be the constant quantify the work that is done in each recursive step, then we can express the total cost as being bonded from above by:

$$T(n) \leq \sum_{h_i=1}^{\log n} c \cdot (\log n - h_i) \, |U_{h_i}|$$

By counting arguments, we have the inequality $2^{\log n - h_i}|U_i| \leq n$ (Proving this argument is left as an exercise). We thus obtain:

$$T\left(n\right) \leq \sum_{h_i=1}^{\log n} c \cdot \left(\log n - h_i\right) \frac{n}{2^{logn-h_i}} = cn \sum_{j=1}^{\log n} 2^{-j} \cdot j$$

$$\leq cn \sum_{j=1}^{\infty} 2^{-j} \cdot j$$

And by the Ratio test for inifinte serires $\lim_{j \to \infty} \frac{(j+1)2^{-j-1}}{j2^{-j}} \to \frac{1}{2}$ we have that the serire covergence to constant. Hence: $T\left(n\right) = \Theta\left(n\right)$.

Heap Sort.   Combining the above, we obtain a new sorting algorithm. Given an array, we could first Heapify it (build a heap from it) and then extract the elements by their order. As we saw, the construction requires linear time, and then each extraction costs $\log n$ time. So, the overall cost is $O(n \log n)$. Correction follows immediately from Build and Extract correction.

    input: Array $H_1, H_2, ..H_n$
1   $H \leftarrow$ build $(x_1, x_2..x_n)$
2   ret $\leftarrow$ Array $\{\}$
3   **for** $i \in [n]$ **do**
4 | $\text{ret}_i \leftarrow$ extract H
5   **end**
6   return ret.

Adding Elements Into The Heap.   (Skip if there is no time).

    input: Heap $H_1, H_2, ..H_n$
1   parent $\leftarrow \lfloor i/2 \rfloor$
2   **if** *parent* > 0 *and* $H_{parent} \leq H_i$ **then**
3 | $H_i \leftrightarrow H_{\text{parent}}$
4 | Heapify-up(parent)
5   **end**

Algorithm 3: Heapify-up.

    input: Heap $H_1, H_2, ..H_n$
1   $H_n \leftarrow v$
2   Heapify-up(n)
3   $n \leftarrow n + 1$

Algorithm 4: Insert-key

6.4.1   Example, Median Heap

Task: Write a datastructure that support insertion and deltion at $O\left(\log n\right)$ time and in addition enable to extract the median in $O\left(\log n\right)$ 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 $H_1, H_2, ..H_n, v$
> 1 **if** $H_{\max,1} \leq v \leq H_{\min,1}$ **then**
> 2 | **if** *size* $(H_{\max}) -$ *size*$(H_{\min}) = 1$ **then**
> 3 | | Insert-key (H_{min}, v)
> 4 | **end**
> 5 | **else**
> 6 | | Insert-key (H_{max}, v)
> 7 | **end**
> 8 **end**
> 9 **else**
> 10 | median $\leftarrow$ Median-Extract H
> 11 | **if** $v < median$ **then**
> 12 | | Insert-key (H_{max}, v)
> 13 | | Insert-key ($H_{min}, median$)
> 14 | **end**
> 15 | **else**
> 16 | | Insert-key (H_{min}, v)
> 17 | | Insert-key ($H_{max}, median$)
> 18 | **end**
> 19 **end**

Algorithm 5: Median Insert key.

> **input:** Array $H_1, H_2, ..H_n$
> 1 median $\leftarrow$ extract H_{max}
> 2 **if** *size(H_{min}) - size(H_{max})* > 0 **then**
> 3 | temp $\leftarrow$ extract $H_{\min}$
> 4 | Insert-key (H_{max}, temp)
> 5 **end**
> 6 return median
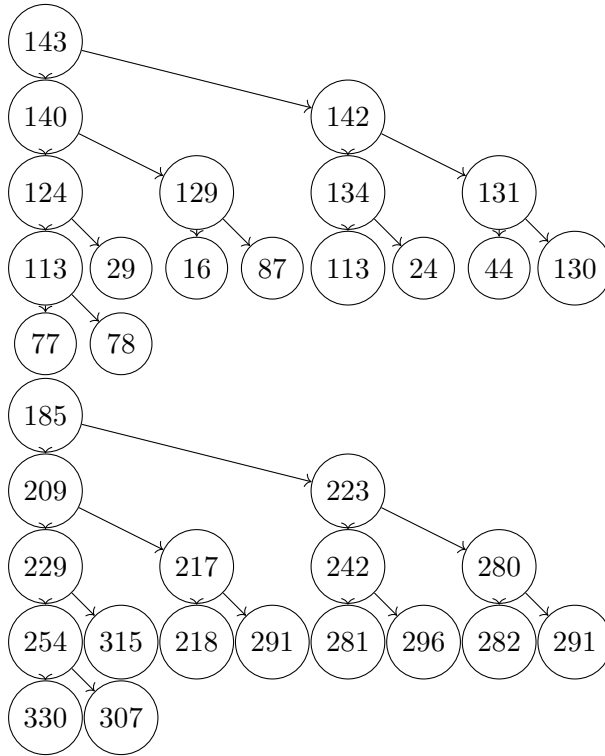
Algorithm 6: Median-Extract.

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

6.5   Appendix. Exercise from last year

Question.   Consider the sets $X = \{x_1, x_2..x_n\}, Y = \{y_1, y_2..y_n\}$. Assume that each of the values x_i, y_i is unique. Write an Algorithm which compute the k most small items in $X \oplus Y = \{x_i + y_j : x_i \in X, y_j \in Y\}$ at $O(n + k \log k)$ time.

   Solution. Notice that If $a \in X$ is greater than i elements of X and $b \in Y$ greater than j elements of Y. Then, $a + b$ greater than $i \cdot j$ elements of $X \oplus Y$. Denote by $X' = \{x'_1..x'_n$ (Y') The elements of X in sorted order. So it's clear that if $x_i + y_j = x'_{i'} + y'_{j'}$ is one of the k smallest elements of $X \oplus Y$ then $i'j' \geq k$. So we will create a heap of elements that respect that inequality and then query that heap.

1 $H_X \leftarrow \text{build}\,(X)$

2 $H_Y \leftarrow \text{build}\,(Y)$

3 $S_X \leftarrow \text{extract-}k\,(H_X)$

4 $S_Y \leftarrow \text{extract-}k\,(H_Y)$

5 $H_{XY} \leftarrow \text{Heap}\,(\{\})$

6   **for** $i \in [k]$ **do**

7   **for** $j \in [k/i]$ **do**

8   Heappush($H_{XY}, S_{X,i} + S_{Y,j}$)

9   **end**

10 **end**

11 return extract-k (H_{XY})