Chapter 6

Heaps.

Leading Problem. find k largest elements. **[COMMENT]** Next year present the finding k-largest elements in $\Theta(n + k \log k)$ time as the goal of the recitation.

Definition 6.0.1. Let $n \in \mathbb{N}$ and consider the sequence $H = H_1, H_2 \cdots H_n \in \mathbb{R}(*)$. we will say that H is a Heap if for every $i \in [n]$ we have that: $H_i \leq H_{2i}, H_{2i+1}$ when we think of the value at indices greater than n as $H_{i>n} = -\infty$.

$\Leftrightarrow$

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)

6.1 Heapify-Down.

How much is the cost (running time) to compute the min of H? (without changing the heap). ($O(1)$). 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 $\{H_2, H_3\}$.)

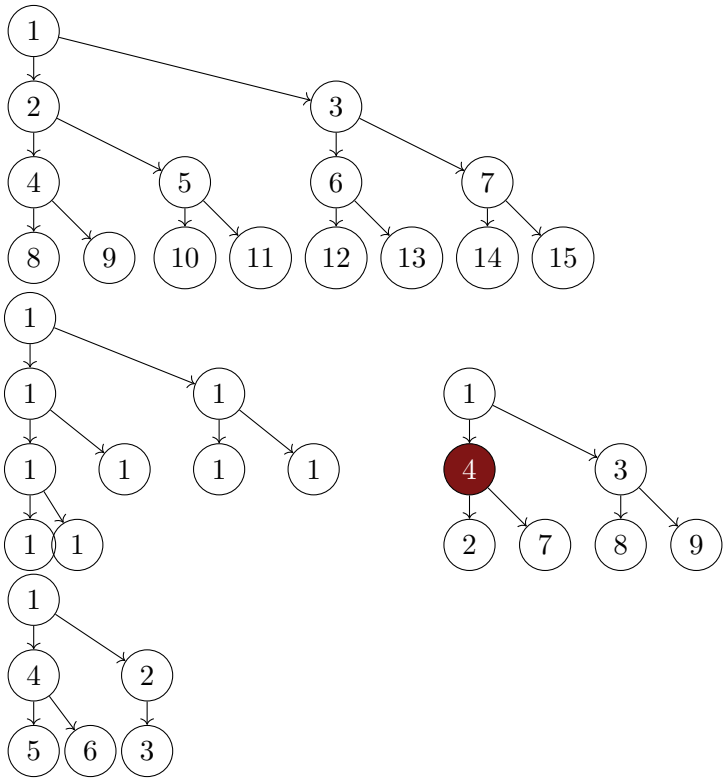

Figure 6.1: The trees representations of the heaps above. The node which fails to satisfy the Heap inequality is colorized.

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**

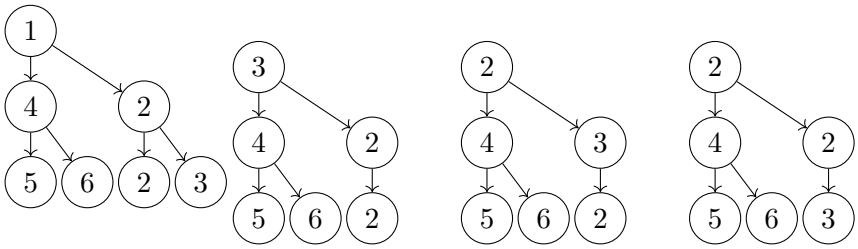

Figure 6.2: Running Example, Extract.

Claim 6.1.1. *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).

□

6.2 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 Heapify-down (H, i)
4 $i \leftarrow i - 1$
5 **end**
6 return $H_1..H_n$

Algorithm 1: 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:

$$\begin{aligned} T(n) \leq & \sum_{h_i=1}^{\log n} c \cdot (\log n - h_i) \frac{n}{2^{logn-h_i}} = cn \sum_{j=1}^{\log n} 2^{-j} \cdot j \\ \leq & cn \sum_{j=1}^{\infty} 2^{-j} \cdot j \end{aligned}$$

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: $T(n) = \Theta(n)$.

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 $H_1, H_2, ..H_n$
$H \leftarrow$ build $(x_1, x_2..x_n)$
ret $\leftarrow$ Array $\{\}$
for $i \in [n]$ **do**
  ret$_i \leftarrow$ extract H
end
return ret.

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

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

Algorithm 2: Heapify-up.

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

Algorithm 3: Insert-key

6.2.1 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
 1 if H_{max,1} ≤ v ≤ 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 ← 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 4: Median Insert key.

```
   input: Array H_1, H_2, ..H_n
 1 median ← extract H_max
 2 if size(H_min) - size(H_max) > 0 then
 3 |   temp ← extract H_min
 4 |   Insert-key ( H_max, temp )
 5 end
 6 return median
```

Algorithm 5: Median-Extract.

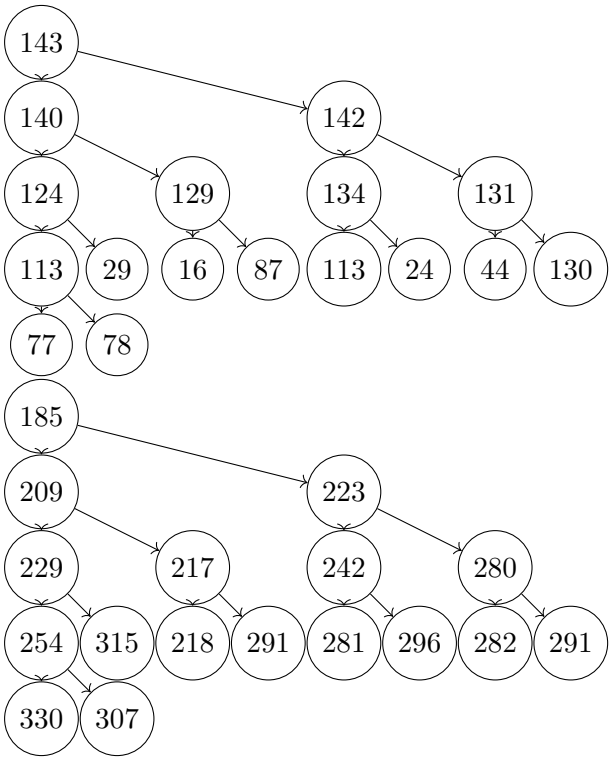

Figure 6.3: Example for Median-Heap, the left and right trees are maximum and minimum heaps.

6.3 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 ← build (X)
2  H_Y ← build (Y)
3  S_X ← extract-k (H_X)
4  S_Y ← extract-k (H_Y)
5  H_XY ← Heap ({})
6  for i ∈ [k] do
7  |   for j ∈ [k/i] do
8  |   |   Heappush( H_XY, S_{X,i} + S_{Y,j} )
9  |   end
10 end
11 return extract-k ( H_XY )
```

6.4 Heapsort (David's group).

We will start by introducing the heap-sort algorithm and providing a proof of its correctness.

```
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 6: Heap-sort(A)

Correctness. We are going to prove the following statement.

Claim 6.4.1. *At the end of the ith iteration, $A_{n-i+1}, A_{n-i+2}, ..A_n$ are the i largest elements of A placed in order, and $A_1, A_2, ..A_{n-i}$ is a maximum heap.*

Proof. By induction.

1. Base. A_n is set in line (3) to be the root of the heap, and therefore is the maximum of A. After line (4), A_1 is the parent of the heap's roots as line (4) excludes A_n from the heap (So the heap inequality holds for any $j \in [2, (n-1)/2]$). Therefore, by the correctness of heapify, we get that after line (5), $A_1, A_2, ..A_{n-1}$ is a heap.

2. Assumption. Assume the correctness of the claim for any $i' < i$.

3. Step. Consider the ith iteration. By the induction assumption, A_1 is a root of the heap $A_1, A_2, ..A_{n-i+1}$ and therefore is their maximum. So after the swapping in line (3), we get that A_{n-i+1} is the element which is greater than $n-i$ elements in A. By using the induction assumption again, we know that it is also less than $A_{n-i+2}, A_{n-i+3}, ..A_n$, so after line (3) and by the fact that $A_{n-i+2}, A_{n-i+3}, ..A_n$ are the $i-1$ largest elements placed in order, we have that $A_{n-i+1}, A_{n-i+2}, A_{n-i+3}, ..A_n$ are the i largest elements placed in order.

By repeating the same arguments in the base case, we can conclude, based on the correctness of heapify, that after line (5), A_1 is either the root of a heap or the heap inequality did not hold for some $i \in [2, (n-i)/2]$. In the latter case, this would contradict the induction assumption (since before line (3), $A_1..A_{n-i+1}$ were heaps).

□