Heaps - Recitation 4

Correctness, Loop Invariant And Heaps.

November 14, 2022

Apart from quantifying the resource requirement of our algorithms, we are also interested in proving that they indeed work. In this Recitation, we will demonstrate how to prove correctness via the notation of loop invariant. In addition, we will present the first (non-trivial) data structure in the course and prove that it allows us to compute the maximum efficiently.

Correctness And Loop Invariant.

In this course, any algorithm is defined relative to a task/problem/function, And it will be said correctly if, for any input, it computes desirable output. For example, suppose that our task is to extract the maximum element from a given array. So the input space is all the arrays of numbers, and proving that a given algorithm is correct requires proving that the algorithm's output is the maximal number for an arbitrary array. Formally:

Correctness.

We will say that an algorithm $\mathcal{A}$ (an ordered set of operations) computes $f : D_1 \to D_2$ if for every $x \in D_1 \Rightarrow f(x) = \mathcal{A}(x)$. Sometimes when it's obvious what is the goal function f, we will abbreviate and say that $\mathcal{A}$ is correct.

Other functions f might be including any computation task: file saving, summing numbers, posting a message in the forum, etc. Let's dive back into the maximum extraction problem and see how one should prove correctness in practice.

Task: Maximum Finding. *Given $n \in \mathbb{N}$ numbers $a_1, a_2, \cdots a_n \in \mathbb{R}$ write an Algorithm that returns their maximum.*

Consider the following suggestion. How would you prove it correct?

Maximum finding.

input: Array $a_1, a_2, ..a_n$
1 let $b \leftarrow a_1$
2 **for** $i \in [2, n]$ **do**
3 $\quad b \leftarrow \max(b, a_i)$
4 return b

Usually, it will be convenient to divide the algorithms into subsections and then characterize and prove their correctness separately. One primary technique is using the notation of Loop Invariant. Loop Invariant is a property that is characteristic of a loop segment code and satisfies the following conditions:

Loop Invariant.

1. Initialization. The property holds (even) before the first iteration of the loop.
2. Conservation. As long as one performs the loop iterations, the property still holds.
3. Termination. Exiting from the loop carrying information.

Let's denote by $b^{(i)}$ the value of b at line number 2 at the ith iteration for $i \geq 2$ and define $b^{(1)}$ to be its value in its initialization. What is the Loop Invariant here? **Claim.** *"at the i-th iteration,* $b^{(i)} = \max\{a_1...a_i\}$*"*.

Proof. Initialization, clearly, $b^{(1)} = a_1 = \max\{a_1\}$. Conservation, by induction, we have the base case from the initialization part, assume the correctness of the claim for any $i' < i$, and consider the ith iteration (of course, assume that $i < n$). Then:

$$b^{(i)} = \max\{b^{(i-1)}, a_i\} = \max\{\max\{a_1, ..a_{i-2}, a_{i-1}\}, a_i\} = \max\{a_1, ..a_i\}$$

And that completes the Conservation part. Termination, followed by the conservation, at the n iteration, $b^{(i)}$ is seted to $\max\{a_1, a_2..a_n\}$.

Task: Element finding. *Given* $n \in \mathbb{N}$ *numbers* $a_1, a_2, \cdots a_n \in \mathbb{R}$ *and additional number* $x \in \mathbb{R}$ *write an Algorithm that returns* i *s.t* $a_i = x$ *if there exists such* i *and* False *otherwise.*

Element finding.

```
   input: Array a_1, a_2, ..a_n
1  for i ∈ [n] do
2      if a_i = x then
3          return i, a_i
4  return False
```

Correctness Proof. First, let's prove the following loop invariant.

Claim *Suppose that the running of the algorithm reached the i-th iteration, then* $x \notin \{a_1..a_{i-1}\}$. **Proof.** Initialization, for $i = 1$ the claim is trivial. Let's use that as the induction base case for proving Conservation. Assume the correctness of the claim for any $i' < i$. And consider the ith iteration. By the induction assumption, we have that $x \notin \{a_1..a_{i-2}\}$, and by the fact that we reached the ith iteration, we have that in the $i-1$ iteration, at the line (2) the conditional weren't satisfied (otherwise, the function would return at line (3) namely $x \neq a_{i-1}$. Hence, it follows that $x \notin \{a_1, a_2..a_{i-2}, a_{i-1}\}$.

Separate to cases. First, consider the case that given the input $a_1..a_n$, the algorithm return False. In this case, we have by the termination property that $x \notin \{a_1..a_n\}$. Now, Suppose that the algorithm returns the pair (i, x), then it means that the conditional at the line (2) was satisfied at the ith iteration. So, indeed $a_i = x$, and the algorithm returns the expected output.

Heaps.

Task: The Superpharm Problem. (Motivation for Heaps) *You are requested to maintain a pharmacy line. In each turn, you get one of the following queries, either a new customer enters the shop, or the pharmacist requests the next person to stand in front. In addition, different customers have different priorities, So you are asked to guarantee that in each turn, the person with the height priority will be at the front.*

Before we consider a sophisticated solution, What is the running time for the naive solution? (maintaining the line as a linear array) $(\sim O\left(n^2\right))$.

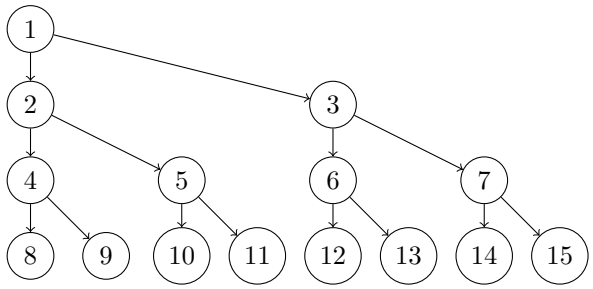
Heaps are structures that enable computing the maximum efficiency in addition to supporting adding and removing elements. We have seen in the Lecture that no Algorithm can compute the max function with less than $n-1$ comparisons. So our solution above is indeed the best we could expect for. The same is true for the element-finding problem. Yet, we saw that if we are interested in storing the numbers, then, by keeping them according to sorted order, we could compute each query in logarithmic time via binary search. That raises the question, is it possible to have a similar result regarding the max problem?

Heap

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 a binary tree in that we associate a number for each node. Then, we will say that this binary tree is a heap if the root's value is lower than its sons' values, and each subtree defined by its children is also a heap.



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)

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 $\{H_2, H_3\}$.)

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:**

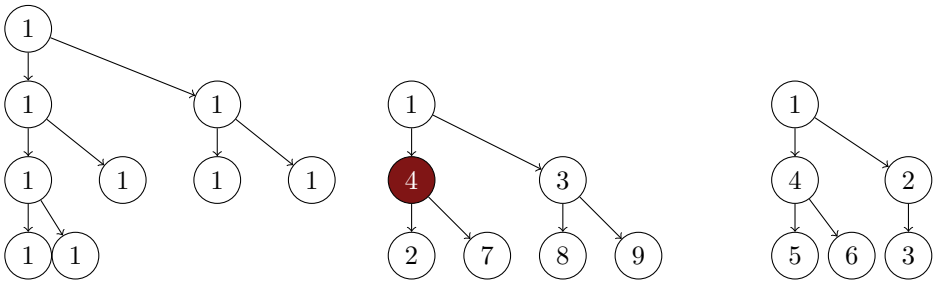


Figure 1: The trees representations of the heaps above. The node which fails to satisfy the Heap inequality is colorized.

Extract-min.

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

Heapify-down.

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 **if** $right < n$ *and* $H_{right} < H_{next}$ **then**
7 next $\leftarrow$ right
8 **if** $i \neq next$ **then**
9 $H_i \leftrightarrow H_{\text{next}}$
10 Heapify-down(next)

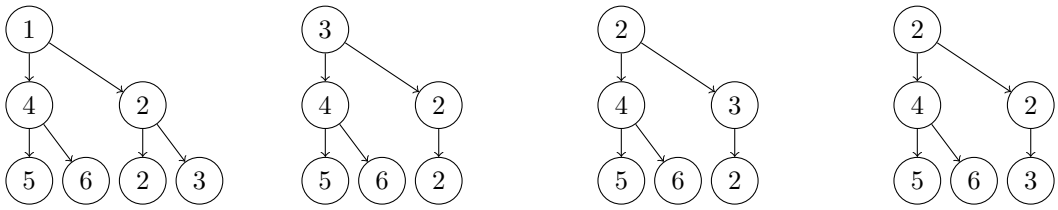


Figure 2: Running Example, Extract.

Claim. 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).

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

Build.

```
   input: Array H = H_1..H_n
1  i ← ⌊½n⌋
2  while i > 1 do
3  |  Heapify-down (H, i)
4  |  i ← i − 1
5  return H_1..H_n
```

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:

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

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.

Heap-Sort.

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
return ret.

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

Heapify-up.

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)

Insert-key.

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

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.

Soultlion. 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.

Median-Insert-key.

input: Array $H_1, H_2, ..H_n$
median $\leftarrow$ extract H_{max}
if $v <$*median* **then**
    Insert-key (H_{max}, v)
    Insert-key (H_{min}, median)
else
    Insert-key (H_{min}, v)
    Insert-key (H_{max}, median)

Median-Extract.

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)
return median

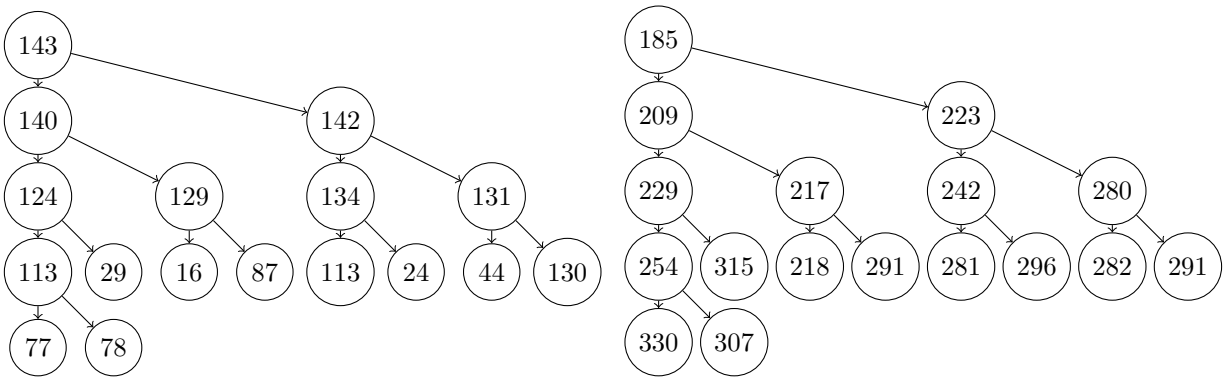


Figure 3: Example for Median-Heap, the left and right trees are maximum and minimum heaps.