Algorithms

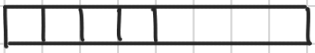
DATE: 24/03/2022

PRIORITY QUEUES / HEAPS

This data structure can make the Dijkstra's algorithm run in linear time.

A heap makes it possible for you to store N (key, value) pairs, where N is to be fixed a priori, in such a way that:

- The item with the lowest key can be accessed in O(1) time; (constant)
- The item in position "i" can be removed from the heap in O(log n) time;
- A new item can be inserted into the heap in O(log n) time.

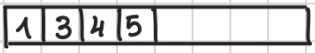


* ARRAY

- Removing and adding takes O(1) time
- Finding the minimum takes Ω(n) time

---

constant time = constantly many operations.

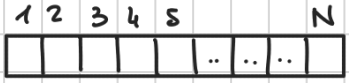


* SORTED ARRAY

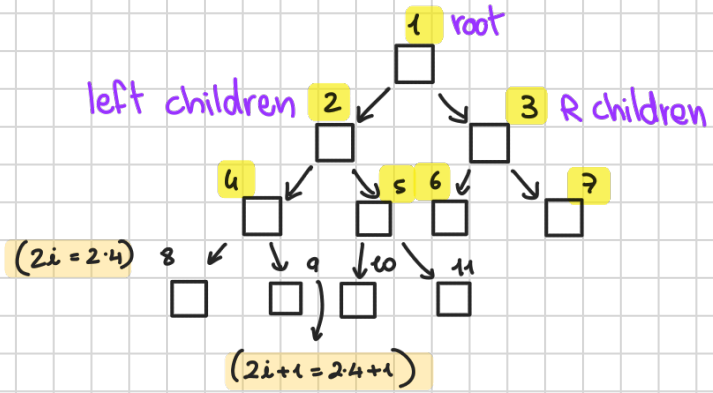
- Finding the minimum takes O(1) time. (constant)
- Remove and adding takes Ω(n) time. (linear)

If we want the array to be sorted.

Heaps "interpolate" between the two extremes *



Heaps, similar to arrays, are pre-allocated data structures. So, if we want to add more than N items, then we have to create a new heap.

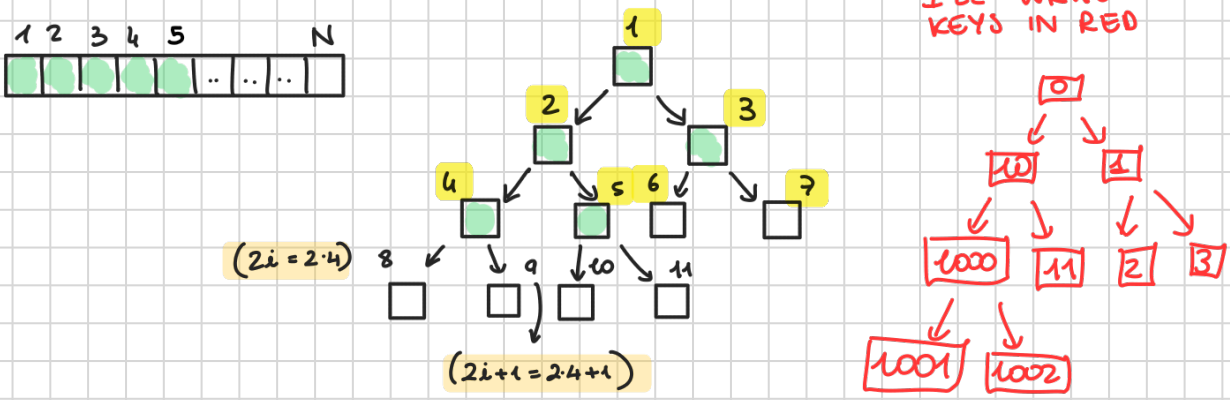


This "logical" representation of the array has a number of useful properties: # Ofc there are others...

- The left child of node "i" has index/position $2i \triangleq left(i)$
- The right child of node i has index "$2i+1$" $\triangleq$ right(i)
- The parent of node i has index $\lfloor \frac{i}{2} \rfloor \triangleq parent(i)$

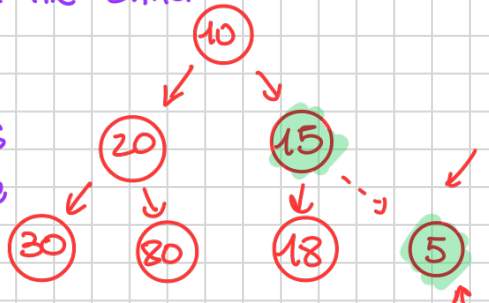
We will be representing this binary tree with an array.

If at some point, the heap only contains $n \leq N$ items then only the first "n" positions of the array will be filled.



HEAP PROPERTY: # We want this "invariant" to always hold.

A heap satisfies the HEAP PROPERTY if, for each filled position i in the heap, if "v" is the item in position i, and if w is the item in position parent(i) ($= \lfloor \frac{i}{2} \rfloor$), then, the KEY [w] $\leq$ KEY [v] # The key of the parent should be smaller or equal than the key of the child.



HEAD PROPERTY ERROR

HEAPIFY-UP

Remember that heaps are made by key-value pairs.

The root has the smallest key and can be accessed in one step.

Here we're trying to add item 5 at the next position available in the array.

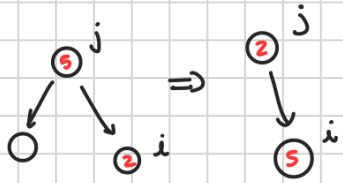
While we go down in the tree, the keys can either remain the same or increase.

Usually we add items to the right* of the last item of the tree, but this usually causes issues that we'll solve with heapify-up.

HEAPIFY-UP(H, i) # H = heap i = position i *n+1 (small n+1)

// This Function will try to "push up" the error
// in position i

IF i ≥ 2: # If i is less than 2, then it's the root, so no issues. :)



// i has a parent

We use heapify-up to guarantee the heap property after adding an item.

$j = \lfloor \frac{i}{2} \rfloor$ # How to find the parent

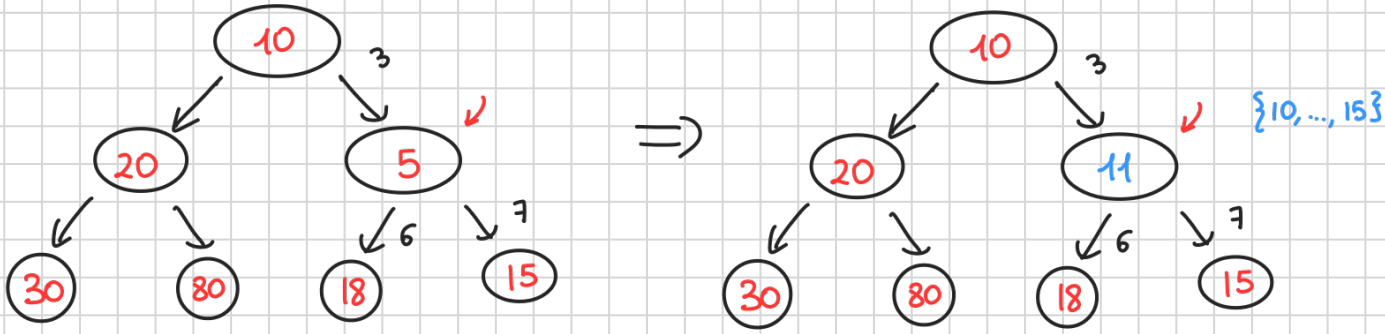
IF KEY[H[i]] < KEY[H[j]]:

// the heap property fails at i

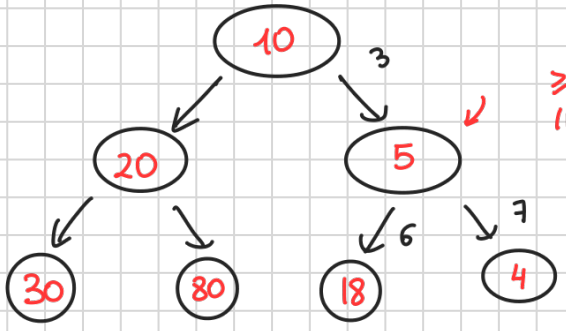
SWAP H[i] with H[j] # This procedure could become recursive if possibly needed to fix also the position of J.

Once we're at the root, we're done. HEAPIFY-UP(H, J)

DEF: We say that H is an almost-heap-with-H[i]-too-small if it is possible to increase the key of H[i] to a value $\alpha \geq$ KEY[H[i]], so that the resulting structure has the heap property.



This H has the almost-heap-with-H[3]-too small prop.



≥10 AND ≤4 IMPOSSIBLE TO FIX

THIS HEAP DOES NOT HAVE THE ALMOST-HEAP-WITH-H[3]-TOO-SMALL PROPERTY

If the heap is in a really bad shape, we "can't" fix it with heapify-up, because it would take too much...

L: The function HEAPIFY-UP(H, i) fixes the heap property of H (provided it needs fixing), if H has the ALMOST-HEAP-WITH-H[i]-TOO-SMALL PROPERTY.

The runtime of HEAPIFY-UP(H, i) is $O(\log n)$.

P: We prove the claim by induction on i.

If $i=1$, the claim is true since H already has the HEAP-PROPERTY (i has no parent that needs to be fixed).

If $i \geq 2$, then the algorithm swaps $H[i]$ with $H[j]$ where $j = \text{parent}(i) = \lfloor \frac{i}{2} \rfloor$ if $\text{KEY}[H[i]] < \text{KEY}[H[j]]$.

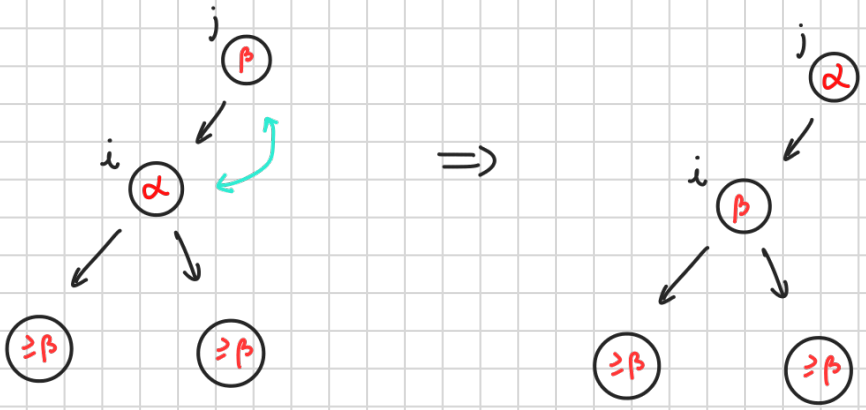
Let β be equal to $\beta = \text{KEY}[H[j]]$, before the swap

Because of the A-H-W-H[i]-T-S property * # there must exist some α
* $\exists\, \alpha \geq \text{KEY}[H[i]]$ such that if the key of $H[i]$ becomes α, the heap is fixed.

But, then, such a α has to satisfy $\alpha \geq \beta$, i.e., α cannot be smaller than the key of the parent of i, and, moreover, α has to satisfy

$$\alpha \leq \text{KEY}[H[\text{left}(i)]] \text{ AND } \alpha \leq \text{KEY}[H[\text{right}(i)]]$$

Thus, $\beta \leq \alpha \leq \min(\text{KEY}[H[\text{left}(i)]], \text{KEY}[H[\text{right}(i)]])$ ✓



Thus, after the swap, left(i) will have a key not smaller than that of i. Same is true for right(i). Since the algorithm made the swap, β was larger than the key of $H[i]$. After the swap, then, the key of i will be larger than the key of $j = \text{parent}(i)$.

Then, after the swap, the heap is going to have the A-H-W-H[j]-T-S PROPERTY.

↑ parent

Since $j = \lfloor \frac{i}{2} \rfloor$, we have $j < i$. And the claim for $j < 1$ is true by induction.

As for the runtime, each call to HEAPIFY-UP takes $O(1)$ time plus, possibly, an extra call to HEAPIFY-UP.
Since the number of calls is bounded by $O(\log n)$, the height of the tree, the runtime is $O(\log n)$ ■

Using HEAPIFY-UP, we can easily add/insert items to the heap. #Heapify-up passes through each node at most once.

ADD (h, v, n) # h = heap v = key-value pair

// n is the number of items currently in the heap

IF $n \leq N - 1$: // then the HEAP has enough space for v :)

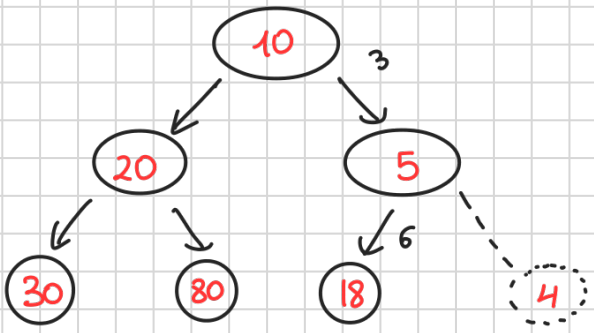
$n += 1$ #Since we're about to increase the number of items of the heap, we increase n.

$H[n] = v$ ← #The tail of the heap, where we'll insert v.

:(// now the HEAP might not satisfy the HEAP PROPERTY

✓ // But the HEAP satisfies the ALMOST-HEAP-WITH-H[n]-TOO-SMALL PROP.

HEAPIFY-UP(H, n)



ADD 4 #When we want to add a new item, the algorithm creates a new node, and the only possible issue is with a parent, but we can fix it by adding to the key-value of the new item whatever is needed for that key to be larger of that of the parent.

THM: ADD adds an item to the HEAP, while keeping the HEAP Property, in time $O(\log n)$

DEF: We say that H is an almost-heap with H[i] too large if $\exists\ \alpha \leq$ KEY[H[i]] s.t. if we decrease KEY[H[i]] to α, then H satisfies the HEAP property.

HEAPIFY-DOWN(H, i):

```
# i is a position in the heap, 1 ≤ i ≤ n
LET n BE THE CURRENT SIZE OF THE HEAP
IF 2i > n:
    # i has no left, and no right, child
    RETURN
ELIF 2i == n:
    # i has only the left child
    j = 2i
ELSE:
    # i has both children
    IF KEY[H[right(i)]] < KEY[H[left(i)]]:
        j = right(i)
    ELSE:
        j = left(i)

IF KEY[H[j]] < KEY[H[i]]:
    SWAP H[i] AND H[j]
    HEAPIFY-DOWN(H, j)
```

Removing items from the heap becomes slightly more complicated, because if we remove an item in the middle of the heap, this may cause troubles.