Algorithms

DATE: 29/03/2022

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)
```

L: The function HEAPIFY-DOWN(H, i) fixes the heap property of H, provided that H has the ALMOST-HEAP-WITH-H[i]-TOO-LARGE property, in time $O(\log n)$

P: We prove the claim by reverse induction (we start from $i = n$, and we move from i to $i-1$)

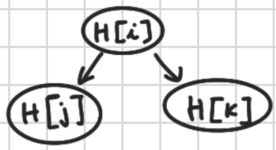
In general, if $i > 2n$, then i has no children, thus the A-H-W-H[i]-T-L property implies the heap property. Thus, the claim holds for $i > 2n$.

Otherwise, $i \leq 2n$. If the algorithm swaps $H[i]$ with $H[j]$, then - at the outset - $KEY[H[j]] < KEY[H[i]]$ 1

(If the algorithm does not swap, then the heap property holds.)

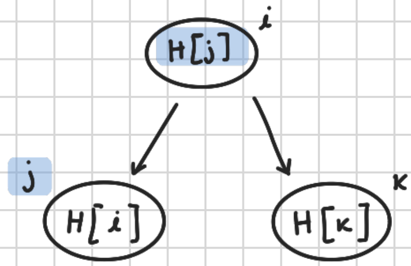
If the algorithm decided to swap $H[j]$ with $H[i]$, then the algorithm chose j as the child of i to consider

Thus, if k is the sibling of j, then $KEY[H[j]] \leq KEY[H[i]]$

IF we start from 

, then $KEY[H[j]] \leq \min(KEY[H[i]], KEY[H[k]])$

and the algorithm will swap $H[i]$ and $H[j]$, obtaining



Thus, the heap property now holds at i.

After the swap, the heap satisfies the ALMOST-HEAP-WITH-H[i]-TOO-LARGE property.

Since the number of levels is $O(\log n)$, and since each call takes $O(1)$ time, plus possibly the time for a new call at a lower level, the runtime is $O(\log n)$ ■

REMOVE (H, i):

LET n be the current number of items in H
SWAP H[i] WITH H[n] $O(1)$
H[n] = NONE $O(1)$
n -= 1 $O(1)$
IF KEY[H[i]] < KEY[H[parent(i)]]:
    // ALMOST-HEAP-WITH-H[i]-TOO-SMALL
    HEAPIFY-UP(H, i) $O(\log n)$

ELIF KEY[H[i]] > min(KEY[H[left(i)]], KEY[H[right(i)]]): $O(1)$

// ALMOST-HEAP-WITH-H[i]-TOO-LARGE:

HEAPIFY-DOWN(H, i) $O(\log n)$

THM: REMOVE(H, i) removes the item in position i, while keeping the heap property of H, in $O(\log n)$ time.

HEAPS:

- ADD(H, v) takes $O(\log n)$ TIME;
- FINDMIN(H) = $O(1)$ = ; // the minimum will be in position 1 because of the heap property.

```
DEF FINDMIN(H):
    RETURN H[1]
```

- REMOVE(H, i) = $O(\log n)$ = ;
- EXTRACT MIN(H) removes and returns the minimum of the heap

```
DEF EXTRACT MIN(H):
    v = FINDMIN(H)
    REMOVE(H, n)
    RETURN v
```

WITH THESE FUNCTIONS, we can already sort an array in $O(n \log n)$ time.

```
DEF HEAP SORT(V):

    LET V BE AN ARRAY OF N ELEMENTS
    INITIALISE A HEAP WITH N POSITIONS
    FOR i = 0, 1, .., N-1
        LET x BE S.T. KEY[x] = V[i] AND VALUE[x] = V[i]
        ADD(H, V[i])
```

FOR $i = 0, 1, \dots, N-1$

  $V[i] = \text{KEY}[\text{EXTRAMIN}(H)]$

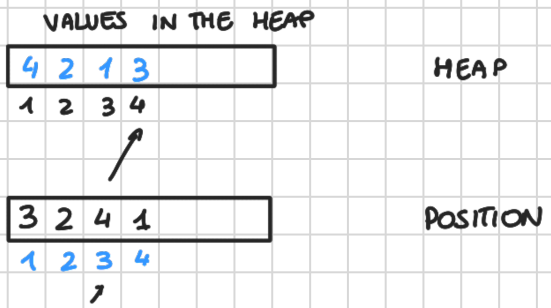
RETURN V

THM: HEAPSORT sorts an array with N items in time $O(N \log N)$

One could want to be able to access items in the heap by value. In order to do so, values should be unique.

Suppose that the values form a subset of $\{1, 2, \dots, N\}$.

IF we use a vector/array V (POSITION) that assigns to the generic value v the position of v in the heap, one gets a good solution.



THUS, $\text{POSITION}[\text{VALUE}[v]] = i$ IF and only if $\text{VALUE}[H[i]] = v$.

- $\text{REMOVEV}(H, \text{value})$ REMOVES the (UNIQUE) ELEMENT of VALUE "value" in time $O(\log n)$

  DEF $\text{REMOVEV}(H, v)$:

    $\text{REMOVE}(H, \text{POSITION}[v])$
    $\text{POSITION}[v] \cdot \text{NONE}$

Observe that, for everything to work out correctly, "Position" should be updated whenever we modify the heap.

We modify the heap in 3 ways:

- ADD AN ELEMENT;
- REMOVE " " ;
- SWAP TWO " " ;

Each of these operations requires no more than updating 2 entries of "Position". Thus, the extra work is only $O(1)$ - TIME - The runtime of each of the functions remain same.