02. Data Structures. Binary Heap. Heap Sort.

Data Structures

Def. **Data Structure:** A Structure that contains some data.

Why do we need a data structure instead of putting it in one big array? Because we want to access it and do operations on it.
These operations might be different, and decide what data structures should be used.

eg. Arrays
Operations are:

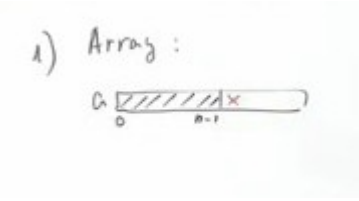
1. get(i) // return a[i]
2. put(i, v) // a[i] = v
   Time complexity = O(1) for both

Binary Heap

Def. **Heap (Priority Queue):** Class of data structures that contains set of elements on which following operations are defined:

1. insert(x)
2. remove_min()

#1 First we try to do the above operations using a simple array:



Here, the shaded region contains our elements and the array is big enough.

```
insert(x)
    a[n]=x //Insert at the end
    n++ //Increase counter
```

```
remove_min()
    j=0
    for i=1..n-1 //For every element
```

```
    if a[i]<a[j] //Find minimum element
        j=i
    swap(a[j], a[n-1]) //Swap minimum element with last element
    n-- //Decrease the counter
    return a[n] //Return the minimum element
```

Time complexities:
insert(x) - O(1)
remove_min() - O(n)

How to improve upon the linear complexity? Keep the array sorted.

#2 Sorted Array (in decreasing order)

```
remove_min()
    n-- //Decrease the counter
    return a[n]
```

```
insert(x)
    a[n]=x //Insert at the end
    n++ //Increase the counter
    i=n-1
    while i>0 and a[i]>a[i-1]: //Like in insertion sort, move the element to front
until it's at the right(sorted) position
        swap(a[i], a[i-1])
        i--
```

Time complexities:
insert(x) - O(n)
remove_min() - O(1)

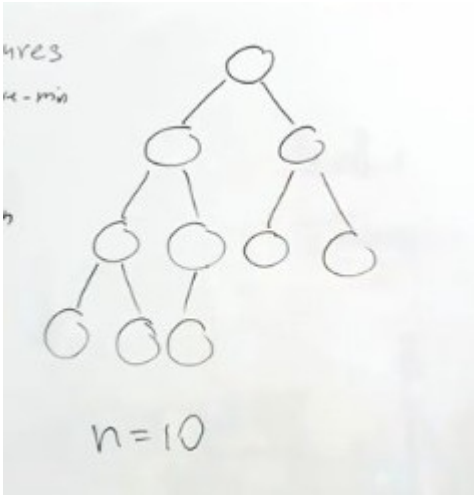
We saw that either one of the operations was O(n)
We can further improve by changing the data structure to binary heap and getting both `insert()` and `remove_min` as O(logn)

Why is it better?
Suppose you have n insertions and n removals

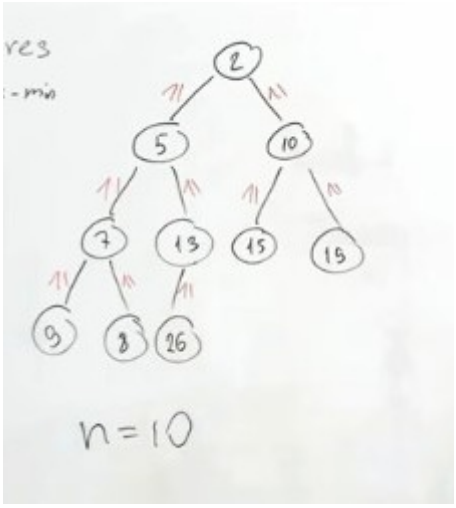
1. By simple array: n + n^2 (n*O(1) insertions + n*O(n) removals)
2. By sorted arrat: n^2 + n (n*O(n) insertions + n*O(1) removals)
3. By binary heap nlogn (n*O(logn) insertions + n*O(logn) removals)

Binary Heap

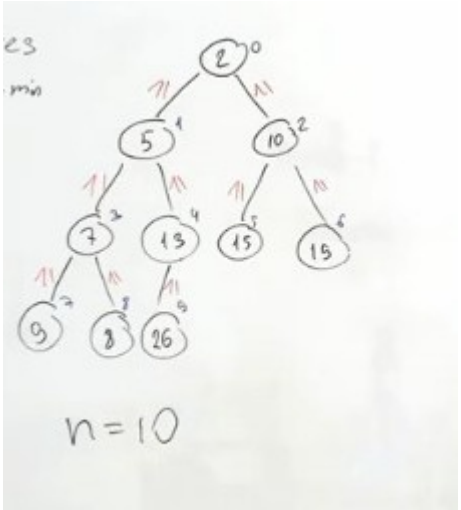
You have a binary tree. Tree with at most two children for every node



Heap property: Every parent should be <= its children

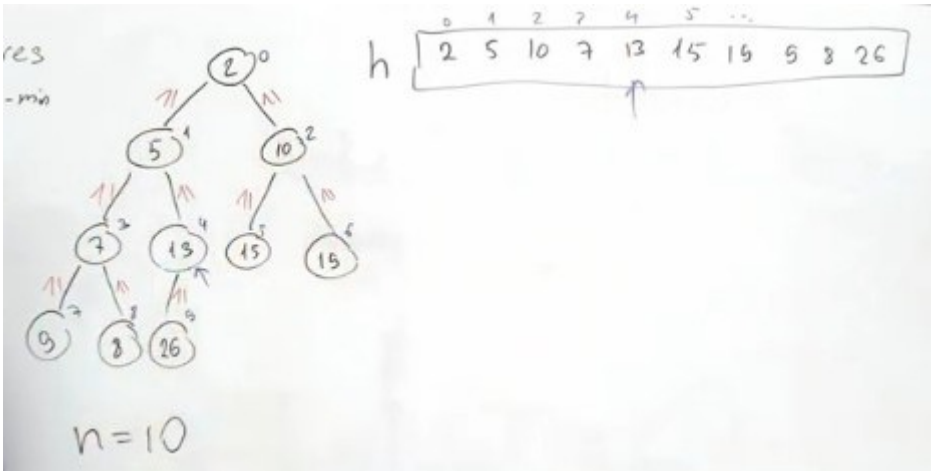


The nodes are indexed as follows:



Hence, a binary heap of size n will always have the same structures.
Instead of making nodes and setting pointers, we can now store the elements in an array,

indexed properly.

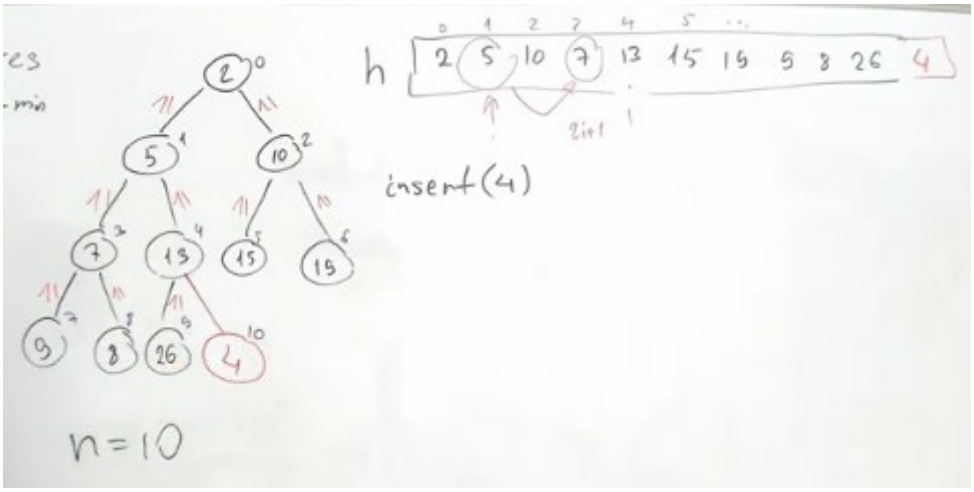


Suppose we are at node indexed i (0-based).
Then,
Index of left child = 2*i + 1*
Index of right child = 2i + 2
Index of the parent = floor((i-1)/2)

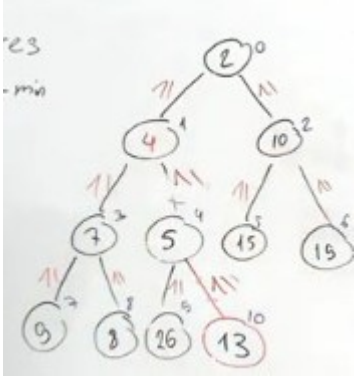
Operations:
insert(x): We insert the element at the very end. Then we swap it with its parent(sift_up) until the heap property is satisfied (i.e children>=parent).

```
insert(x)
    h[n]=x //insert x at position n
    n++ //increase n (size of heap)
    i=n-1 //index of newly inserted element is now n-1
    while i>0 and a[i]<a[(i-1)/2]: //while index in heap and child is smaller than
its parent
        swap(a[i], a[(i-1)/2]) //swap parent and child
    i=(i-1)/2 //new index of newly inserted element
```

Element '4' inserted at the end:



Element '4' after making the necessary swaps until the heap property is satisfied:



Time complexity:
Number of iterations of the while loop. At most logn iterations (height of tree)
∴O(logn)

remove_min(x):
Because of the heap property, the minimum element of the binary heap will be the root and can be found in O(1) time. We remove the root by swapping with last node and then swap the newly replaced root element with its smallest children(sift_down) till the heap property is maintained again.

```
remove_min()
    swap(h[0], h[n-1]) //swap root with last element
    n-- //remove the node
    i=0 //index of root
    while 2i+1 < n: //While we have at least 1 child
        j=2i+1 //j=left child
        if (2i+2<n) and h[2i-2]<h[j] //if we have second child and it's smaller
            j=2i+2 //j=right child
            //j is now the smallest child of the parent
        if h[j]>=h[i] //If the smallest child is greater than parent
            break //nothing else to do
        swap(h[i],h[j]) //swap child and parent
        i=j //repeat for next
    return h[n] //return the removed element
```

Time complexity: O(logn)

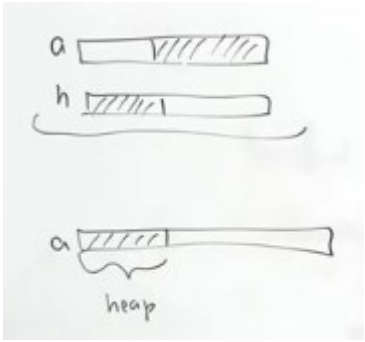
Heap Sort

1. Insert all elements of the array into a heap
2. Remove the last element one by one

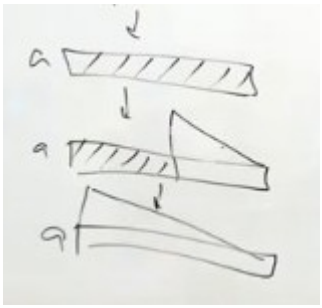
```
sort(a)
    for i=0..n-1 //n
        insert(a[i]) //logn
    for i=0..n-1 //n
        a[i]=remove_min() //logn
```

Time complexity: O(nlogn)

We need two arrays. The source array(a), and another for the heap(h).
We can improve upon this.
While inserting



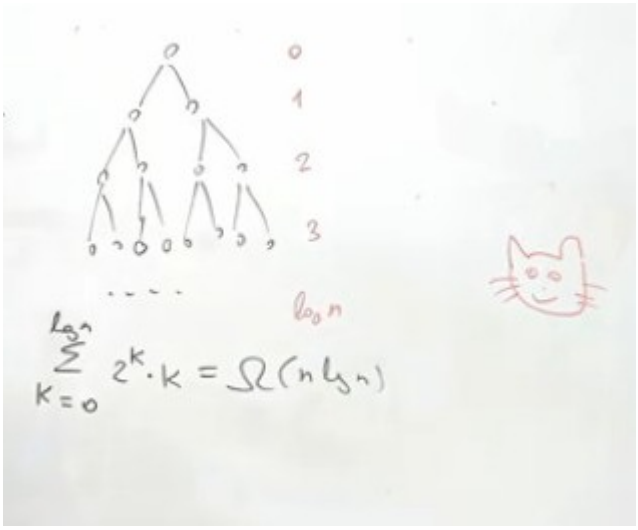
Here, left part of the array(a) is already traversed, and only the left part of the heap(h) is used.
So we can use the left part of the array as binary heap.
While removing



Removed elements go to the back of the heap, sorting the array in **decreasing order**

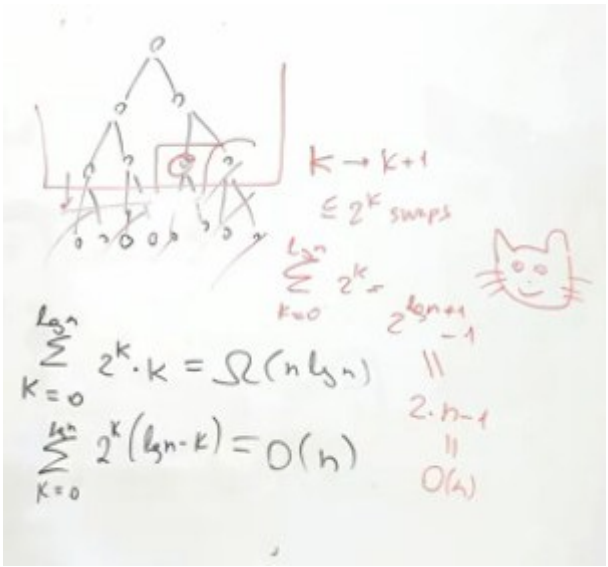
```
sort(a)
    for i=0..n-1
        sift_up(a[i])
    for i=n-1..0
        swap(a[0], a[i]) //Swap the last element with the first(minimum) element
        sift_down(a[0]) //Sift down
```

Time complexity to build a heap:
We traverse from the beginning, and sift up to the root.



Can be improved.

Instead traverse from below, and sift down to the bottom.



This gives O(n) complexity.

In essence, the top of the tree have lesser elements, and the bottom of the tree have more elements. Beginning from the bottom ensures faster operations on more elements.