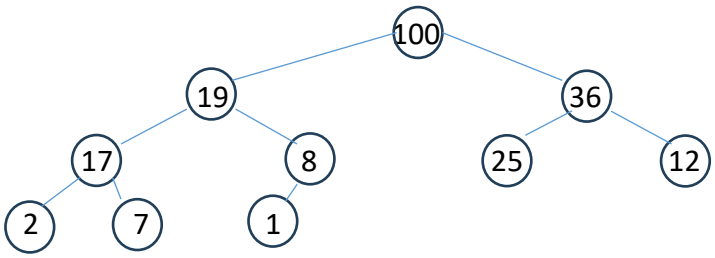
CS 315 - Day 16, Heapsort

Today we'll discuss Heap-Sort. First the heap data structure, then how to construct a heap, and finally how to use the hea[to sort. Later we will discuss other applications for heaps.



What do you notice about this image?

- Structure
  - Binary tree.
  - Is this a ***full binary tree***, where every node must have either zero or two children? ***Note that this is not required in a binary tree representation of a heap***.
  - Is this a ***complete binary tree***, where every internal node must have exactly two children, except for the last full level of internal nodes, where nodes can have between zero and two children. The nodes in the last level must also be left-filled (filled from left to right). ***This IS required in a binary tree representation of a heap***.
- Ordering
  - The key value of the parent is greater than (or equal to) the key values of each child.
  - The key values of the children nodes are unrelated to each other.

This structure follows the ***Heap property***, $A[\, parent(i)] \geq A[\, i\,]$.

"In a *max heap*, for any given node C, if P is the parent node of C, then the *key* (the *value*) of P is greater than or equal to the key of C."
https://en.wikipedia.org/wiki/Heap_(data_structure)#:~:text=In%20computer%20science%2C%20a%20heap,to%20the%20key%20of%20C.

So conceptually, we represent a heap as a complete binary tree satisfying the heap property.

In practice, a heap is stored in an array (or list):

A:	100	19	36	17	8	25	12	2	7	1
	1	2	3	4	5	6	7	8	9	10

The array containing the heap has some nice properties with factors of 2 (which can be implemented efficiently with binary shifts):

```
Parent(i):
   return i//2
```

```
Left(i):
   return 2*i
```

```
Right(i):
   return 2*i+1
```

In order to sort using a heap, we need to be able to:

- Build a heap
- Maintain a heap

Max-Heapify is called when the key value of A[i] maybe less that its children, but the subheaps rooted by each child are valid heaps (satisfy the heap property).

```
Max-Heapify(A,i)

  l = Left(i)
  r = Right(i)

  if l <= A.heap-size and A[l] > A[i]
     then largest = l
     else largest = i

  if r <= A.heap-size and A[r] > A[largest]
     then largest = r

  if largest != i
     then exchange A[i] <--> A[largest]
          Max-Heapify(A,largest)
```

Note this is recursive, and the related method discussed in CS 240 – Computer Science 2 is appropriately named ***ReHeapDown***.

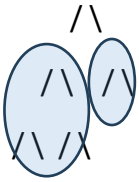
What is Max-Heapify's asymptotic running time?

All the lines are constant time, except for the recursive call. So we'll represent with a recurrence relation.

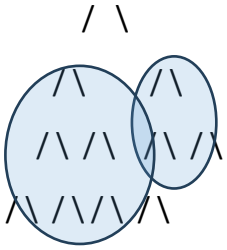
$$T(n) = T\left(\frac{2n}{3}\right) + \theta(1)$$

The $\frac{2}{3}$ is from the possible worst-case split of a complete tree where half the last level is filled, and always going to the largest subheap.

3 to 1, $\frac{3}{4} = 0.75$

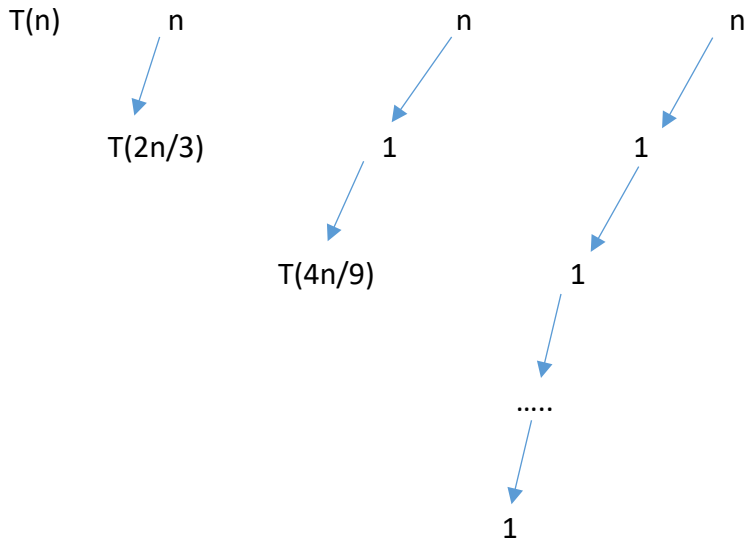


7 to 3, $\frac{7}{10} = 0.70$



15 to 7, $\frac{15}{22} = 0.682$, and so on → $\frac{2}{3}$

If we build the recurrence tree, we can guess a solution:



And so on… We recall the same argument we made that $T(n) = T\left(\frac{n}{2}\right) + \theta(1)$ → $O(\lg n)$, we have a logarithmic number of levels, each with a constant amount of work. Since we're only reducing the problem size by $\frac{1}{3}$ there will be more levels, but still asymptotically $O(\lg n)$.

Now that we have `Max-Heapify` we can use is to build a Max-Heap. Note that a subheap of size one element automatically satisfies the heap property, so we just need to start halfway up the heap. `Max-Heapify` requires the subheaps rooted at each child to be valid.

```
Build-Max-Heap(A)
A.heap-size = A.length

for i = floor( A.length/2 ) downto 1 do
       Max-Heapify(A,i)
```

An upper, "not very tight" bound on `Build-Max-Heap` is that there are $O(n)$ calls to a routine (`Max-Heapify`) that is $O(\lg n)$, which would give $O(\mathrm{n} \cdot \lg n)$ for `Build-Max-Heap`. However, the text gives a derivation of a tighter bound, and shows `Build-Max-Heap` is actually $O(n)$. Basically the idea is that the calls to `Max-Heapify` are mostly for small subheaps.

So, finally we are able to consider Heapsort:

```
Heapsort(A)
Build-Max-Heap(A)                          → O(n)
for i = A.length downto 2                  → this loops runs O(n) times
    exchange A[1] <--> A[i] // that's a one
    A.heap-size = A.heap-size - 1
    Max-Heapify(A,1) // that's a one → inside the loop, each call to Max-Heapify is O(lg n)
```

So, $O(n)$ loops with $O(\lg n)$ work at each level, gives Heapsort as $O(\mathrm{n} \cdot \lg n)$. Since it's recursive, intuitively you might already have deduced that.

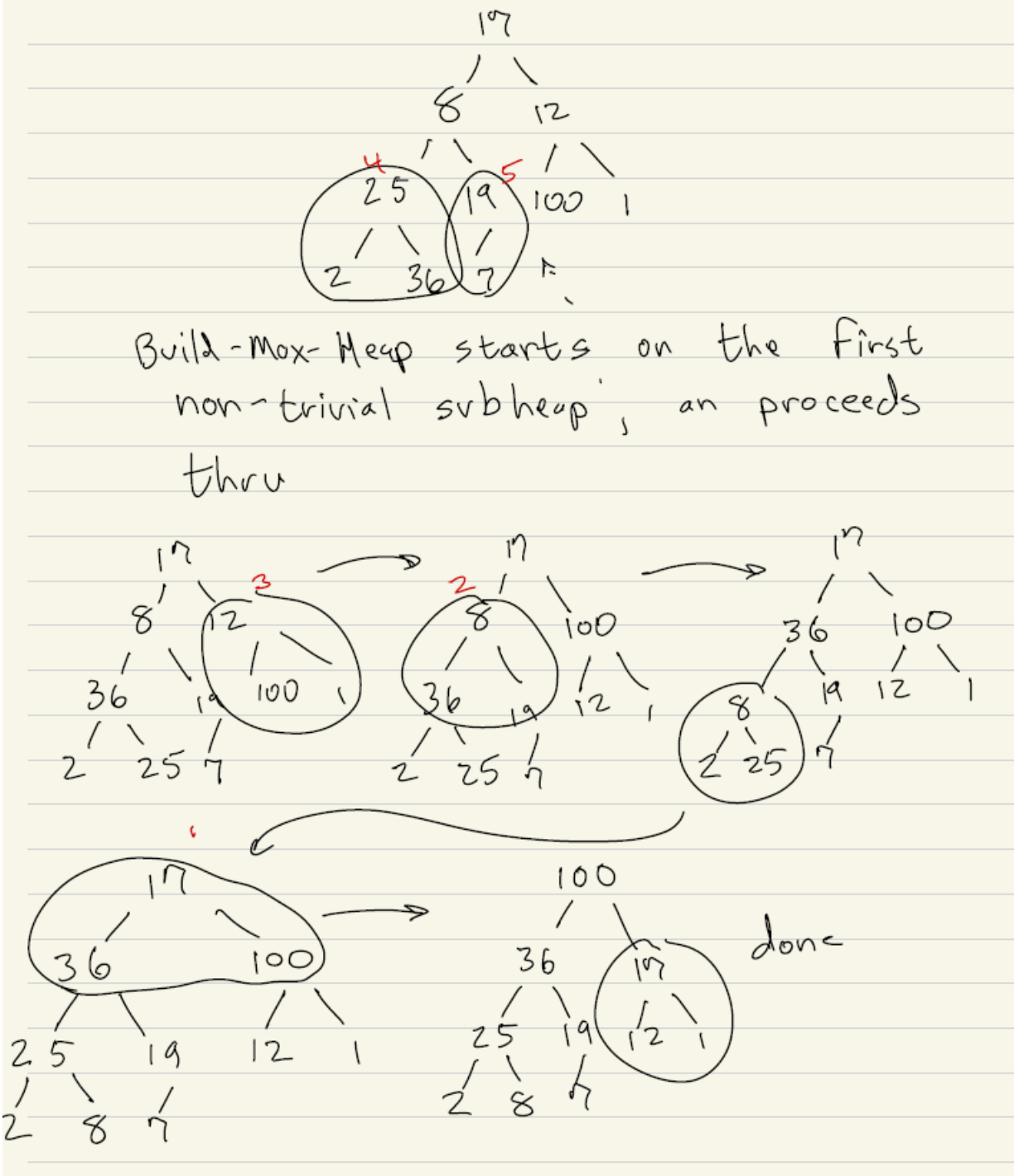
So the Max-Heap is built, the current max element is removed from the heap (the max element is swapped with the last element and the heapsize is reduced by one).

Let's try an example.

Here's the initial list:

A:	17	8	12	25	19	100	1	2	36	7
	1	2	3	4	5	6	7	8	9	10

Step through `Build-Max-Heap(A)`



Which gives the resulting heap:

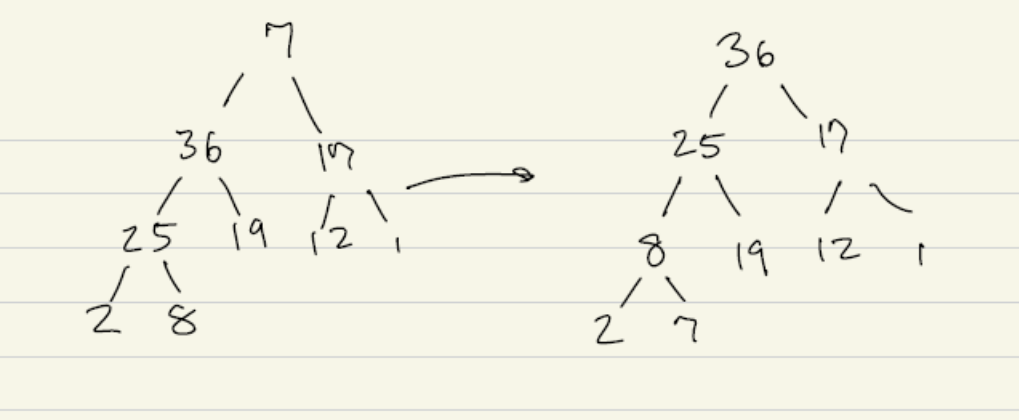
A:	100	36	17	25	19	12	1	2	8	7
	1	2	3	4	5	6	7	8	9	10

Now, in `Heapsort` the max element is swapped out of the heap and to the end of the list, and the heap is rebuilt with `Max-Heapify`.

A:	7	36	17	25	19	12	1	2	8	100
	1	2	3	4	5	6	7	8	9	

Gets “fixed” to

A:	36	25	17	8	19	12	1	2	7	100
	1	2	3	4	5	6	7	8	9	

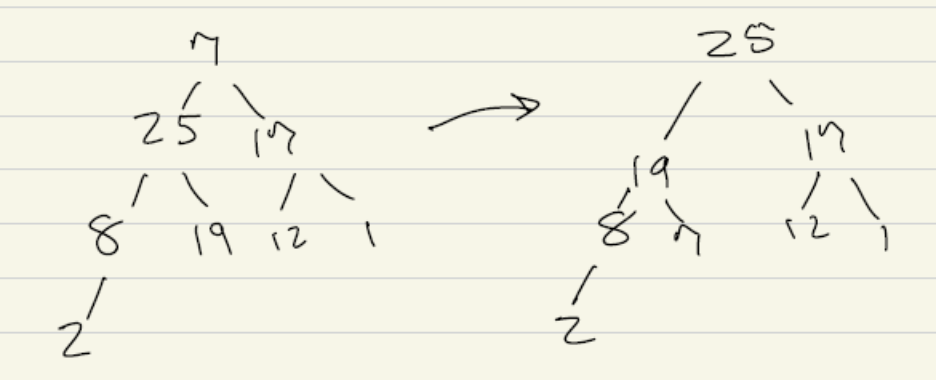


Then again, the max element is swapped out of the heap and to the end of the list, and the heap is rebuilt with `Max-Heapify`.

A:	7	25	17	8	19	12	1	2	36	100
	1	2	3	4	5	6	7	8		

Gets “fixed” to

A:	25	19	17	8	7	12	1	2	36	100
	1	2	3	4	5	6	7	8		



And this continues until all N elements have been removed from the heap.