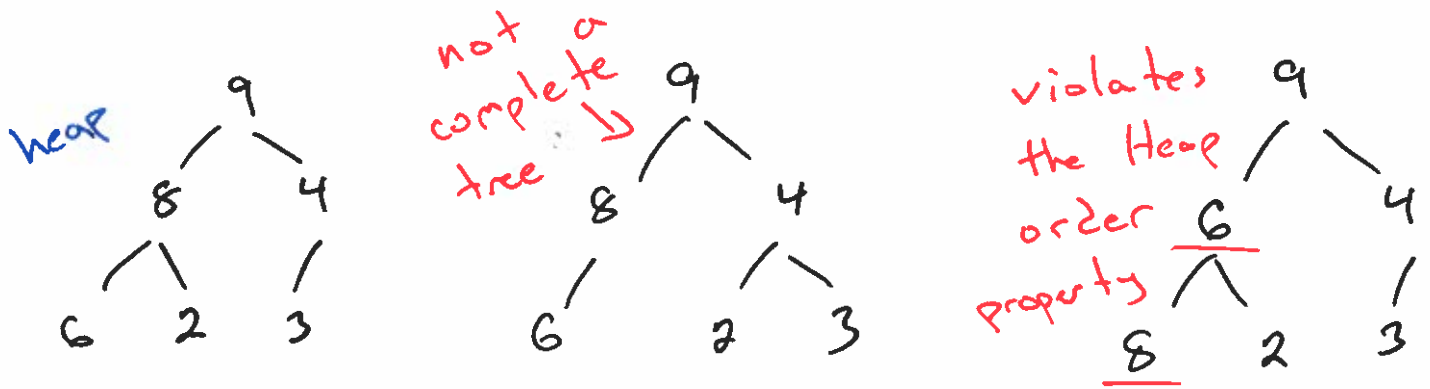
Heaps

Priority Queue

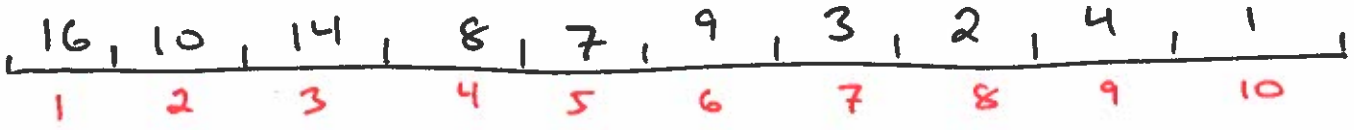
- implements a set S where each element has some key
- operations
  - Insert(S, x) add x to S
  - Max(S) returns the element in S with largest key
  - ExtractMax(S) Find Max(S) and remove it
  - IncreaseKey(S, x, k) increase the key of x to a new value k

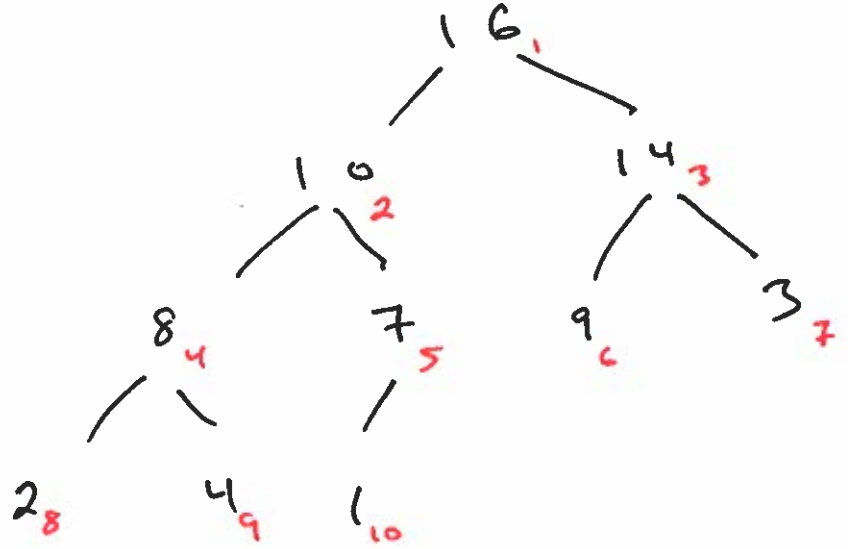
- A Heap is a binary tree with the following properties
  - It is a complete binary tree
  - Satisfy the Heap order property
    - for a Max Heap

      the key of a node is ≥ the keys of its children



Heap as Array





root: $i = 1$

parent(i): $\lfloor \frac{i}{2} \rfloor$

left(i): $2i$

right(i): $2i + 1$

Build Max Heap

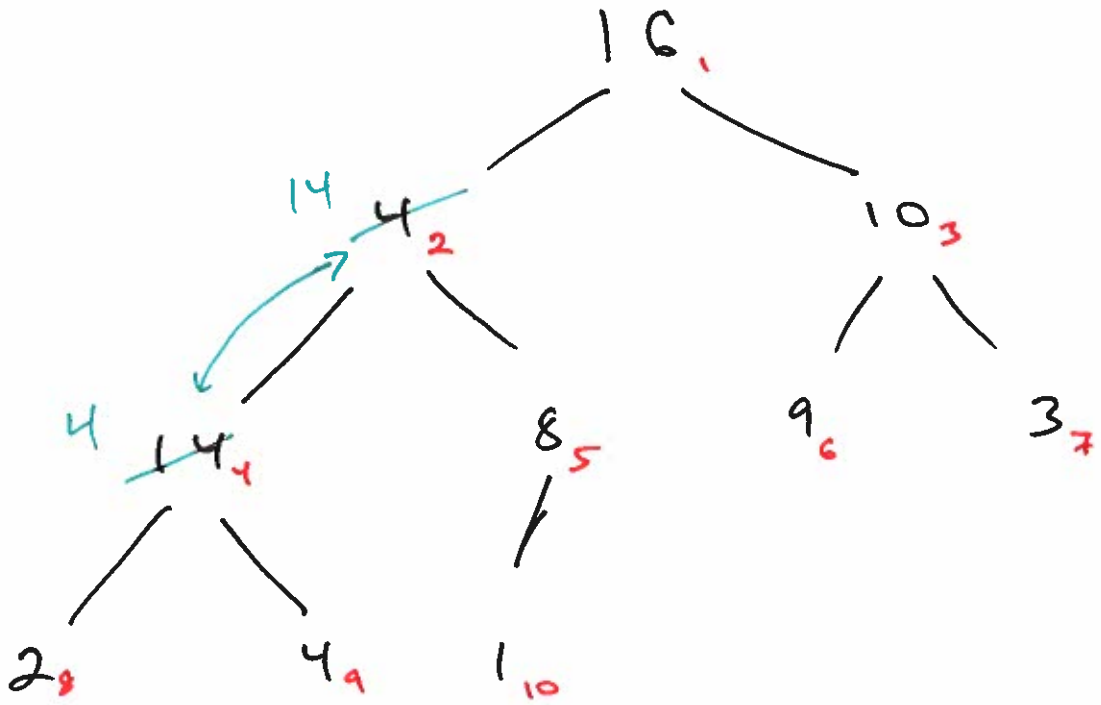
- produce a max heap from an unordered array

MaxHeapify(A, i)

Input: an array A where the left and right children of i roots heaps

i: an array index

Output: A modified so that i roots a heap



- Max Heapify(A, 2)
- Swap A[2] with the larger of its children (A[4])
- recursively call Max Heapify(A, 4)

```
MaxHeapify(A, i)
1   l = Left(i)
2   r = Right(i)
3   if l ≤ A.heapsize and A[l] > A[i]
4       largest = l
5   else
6       largest = i
7   if r ≤ A.heapsize and A[r] > A[largest]
8       largest = r
9   if largest ≠ i
10      Exchange A[i] and A[largest]
11      MaxHeapify(A, largest)
```

Input size: A.heapsize

n = array length

Basic op: comparison

Best case: $A[i] \geq A[r]$ and $A[i] \geq A[l]$

$O(1)$

Worst case: $i = 1$ and $A[i]$ is the smallest element

$$T(n) = T\left(\frac{n}{2}\right) + a \in \Theta(\log n)$$

```
BUILDMAXHEAP(A)
1  A.heapsize = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAXHEAPIFY(A, i)
```

Input size: n = A.length

Basic op: comparison inside MaxHeapify

Best case: any max heap or reverse sorted

$\Theta(n)$

Worst case: sorted array or any min heap

Naive Analysis: $O(n \log n)$

Build Max Heap Complexity

- assume $n = 2^{h+1} - 1$
- level 0 has 1 node
- level 1 has 2 nodes
- level 2 has 4 nodes
- level h has 2^h nodes with 0 work
- level $h-1$ has 2^{h-1} nodes with 1 work
- level $h-2$ has 2^{h-2} nodes with 2 work

In general at level $h-j$ has 2^{h-j} nodes that might shift downwards j levels

$$C(n) = \sum_{j=0}^{h} j\, 2^{h-j}$$

$$= \sum_{j=0}^{h} j \frac{2^h}{2^j}$$

$$= 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

Infinite geometric series $x < 1$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{(1-x)}$$

Take the derivitive of both sides with respect to x

$$\sum_{j=0}^{\infty} j x^j = \frac{x}{(1-x)^2}$$

So if $x = \frac{1}{2}$

$$\frac{x}{(1-x)^2} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

$$C(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j}$$

$$\leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j}$$

$$\leq 2^h \cdot 2$$

$$\leq 2^{h+1}$$

$$\leq n+1 \in \Theta(n)$$

$n = 2^{h+1} - 1$

```
HeapSort(A)
1  BuildMaxHeap(A)                          Θ(n)
2  for i = A.length downto 2
3      exchange A[1] and A[i]
4      A.heapsize = A.heapsize − 1
5      MaxHeapify(A, 1)                     Θ(log n)
```

Input size: $n = A.length$

Basic op: comparison in maxHeapify

Complexity

$$\Theta(n) + \Theta(n \log n) \in \Theta(n \log n)$$

Adaptive? No

In-place? yes

Stable? No

Build and analyse ExtractMax(A)

```
HeapExtractMax(A)
1  max = A[1]
2  A[1] = A[A.heapsize]
3  A.heapsize = A.heapsize − 1
4  MaxHeapify(A, 1)
5  return max
```

```
HEAPINCREASEKEY(A, i, key)
    // Assume A[i] ≤ key
1   A[i] = key
2   while i > 1 and A[PARENT(i)] < A[i]
3         exchange A[i] and A[PARENT(i)]
4         i = PARENT(i)
```