Week 3 - Lecture Notes

Topics: - Analysis of QuickSort
Randomized QuickSort
Heap
Heap Sort
Decision Tree

Pseudo-code for Quick Sort

QUICKSORT (A, p, r)

1. if $p < r$
2. then $q \leftarrow$ PARTITION (A, p, r)
3. QUICKSORT (A, p, q)
4. QUICKSORT $(A, q+1, r)$

Initial call: QUICKSORT $(A, 1, n)$

Analysis of Quick sort

- Assume all input elements are distinct
- In practice, there are better partitioning algorithms for when duplicate input may exist.
- Let $T(n)$ = worst case running time on an array of n elements

Worst case of Quick Sort

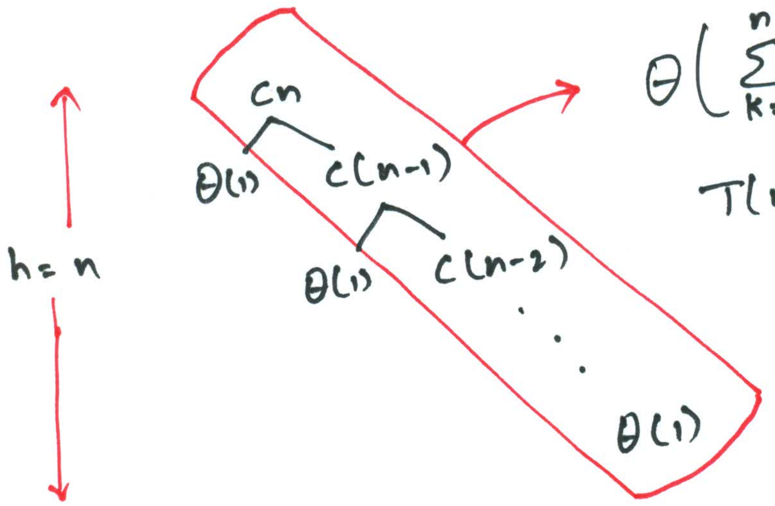
- Input is sorted or reverse sorted
- Partition around minimum or maximum element
- Split : $0 : n-1$,

  one side of the partition always has no element.

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= \Theta(n) + T(n-1) \\
&= \Theta(n^2)
\end{aligned}
$$

Worst Case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

$$
\begin{aligned}
T(n) &= \Theta(n) + \Theta(n^2) \\
&= \Theta(n^2)
\end{aligned}
$$

Best-Case Analysis

If we are lucky, PARTITION splits the array evenly $\left(\frac{1}{2}:\frac{1}{2}\right)$

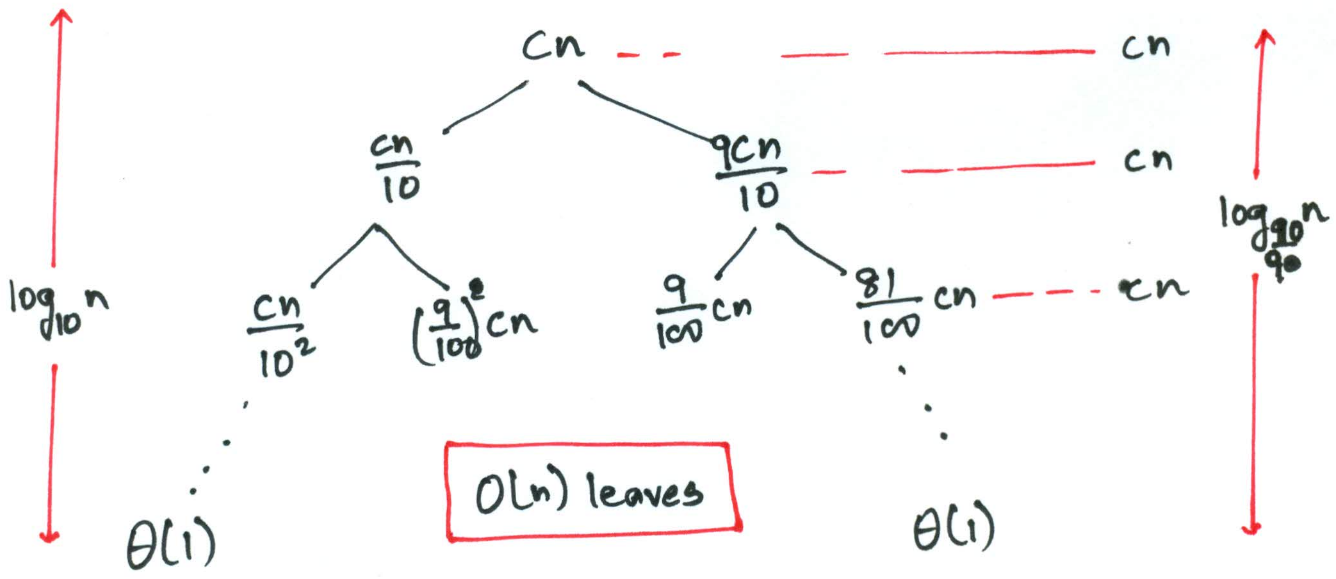
$$T(n) = 2T(n/2) + \Theta(n)$$

$$= \Theta(n \log n) \quad [\text{same as Merge Sort}]$$

Analysis of "almost-best" case

Consider the split is always $\frac{1}{10} : \frac{9}{10}$.

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$



$$cn\log_{10} n \le T(n) \le cn\log_{\frac{10}{9}} n + O(n)$$

Randomized QuickSort

Idea: Partition around a random element

- Running order is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst case behaviour.
- The worst case is determined only by the output of a random-number generator.

Randomized Quick Sort Analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an ipput of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the indicator random variable as:

$$X_k = \begin{cases} 1 & \text{if PARTITION generates } k : n-k-1 \text{ spit} \\ 0 & \text{otherwise} \end{cases}$$

$E[X_k] = Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0:n-1 \text{ split} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1:n-2 \text{ split} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1:0 \text{ split.} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)$$

Calculating expectation

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k]\, E\left[T(k) + T(n-k-1) + \Theta(n)\right]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)]$$

$$+ \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

The $k=0,1$ terms can be absorbed in the $\Theta(n)$

Prove: $E[T(n)] \leq an\lg n$ for constant $a > 0$

Choose 'a' large enough so that $an\lg n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

Use fact: $\sum_{k=2}^{n-1} k\lg k \leq \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$

$$
\begin{aligned}
E[T(n)] &\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\leq an\lg n
\end{aligned}
$$

if a is chosen large enough so than $\frac{an}{4}$ dominates $\Theta(n)$

QuickSort in Practice

- Quicksort is a great general-purpose sorting algorithm
- Quicksort can benefit substantially from code tuning
- Quicksort is typically over twice as fast as merge sort.

Priority Queue

A data structure implementing a set S of elements, each associated with a key, supporting the following operations.

insert(S, x) : insert element x into set S

max(S) : return element of S with largest key

extract_max(S): return element of S with largest key and remove it from S

increase_key(S, x, k): increase the value of element x's key to new value k.

Heap

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key ~~node~~ of a node is $\geq$ than the keys of its children

(Min Heap defined analogously)

Heap as a Tree

root of tree : first element in the array, corresponding $i=1$

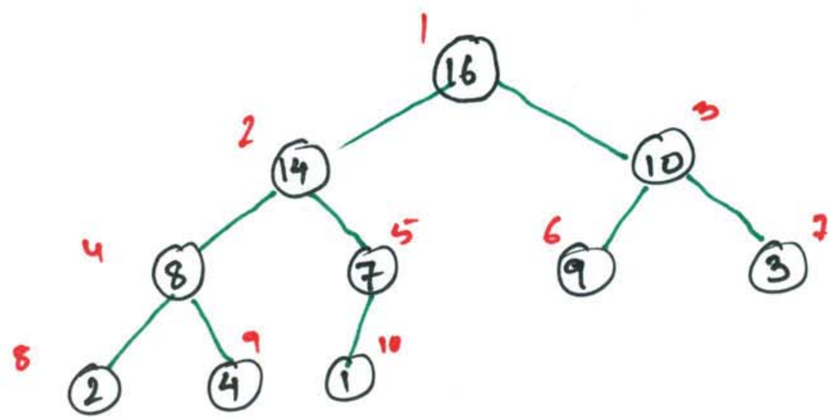
$parent(i) = i/2$: returns index of node's parent

$left(i) = 2i$: returns index of node's left child.

$right(i) = 2i+1$: returns index of node's right child.

Example:

16 14 10 8 7 9 3 2 4 1



No pointers required.

Height of a binary heap is $O(\lg n)$

Heap Operations:

max-heapify : correct a single violation of the heap property in a subtree at its root

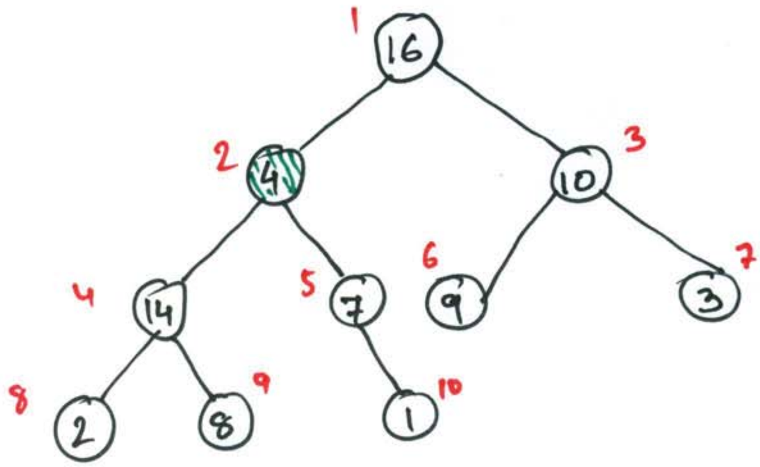
build-max-heap: produce a max-heap from an unordered array.

insert, extract_max, heapsort.

Max-heapify

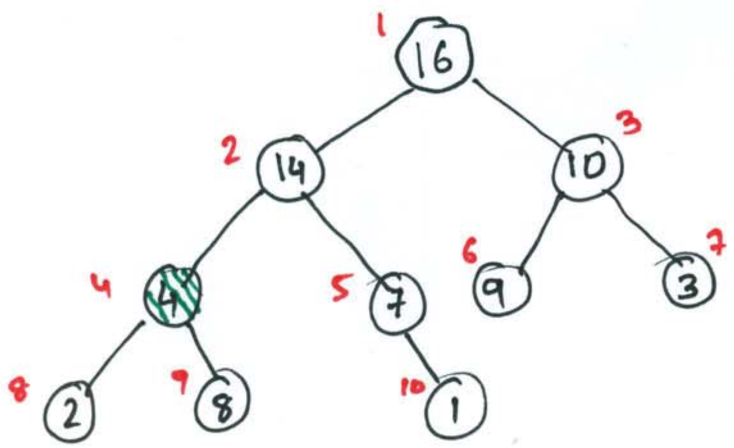
- Assume that the trees rooted at left(i) and right (i) are max-heaps
- If A[i] violates the max-heap property, correct violation by "tricking" element A[i] down the tree, making the subtree rooted at index i a max-heap.

Max-heapify (Example)

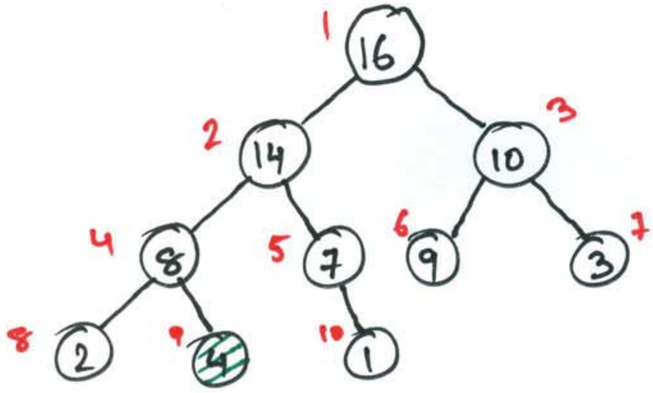


MAX-HEAPIFY (A,2)
heap-size [A] = 10

Node 10 is the left child of node 5 but is drawn to the right for convenience.



Exchange A[2] with A[4]
Call MAX-HEAPIFY (A,4)
because max-heap property is violated



Exchange A[4] with A[9]

No more calls.

Time = $O(\log n)$

Max. Heapify Pseudocode

```
l = left (i)
r = right (i)
if (l ≤ heap-size (A) and A[l] > A[i])
    then largest = l
    else largest = i
if (r ≤ heap-size (A) and A[r] > A[largest])
    then largest = r
if largest ≠ i
    then exchange A[i] and A[largest]
    Max-Heapify (A, largest)
```

Build_Max_Heap (A)

Converts $A[1, \ldots, n]$ to a max heap

Build_Max_Heap (A):

for $i = n/2$ down to 1

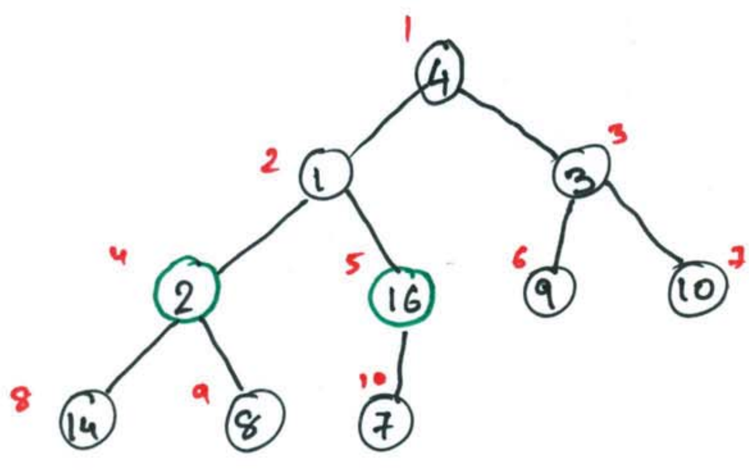
do Max_Heapify (A, i)

- We start at $i = n/2$ because elements $A[n/2+1, \ldots, n]$ are all leaves of the tree

$2i > n$, for $i > n/2 + 1$

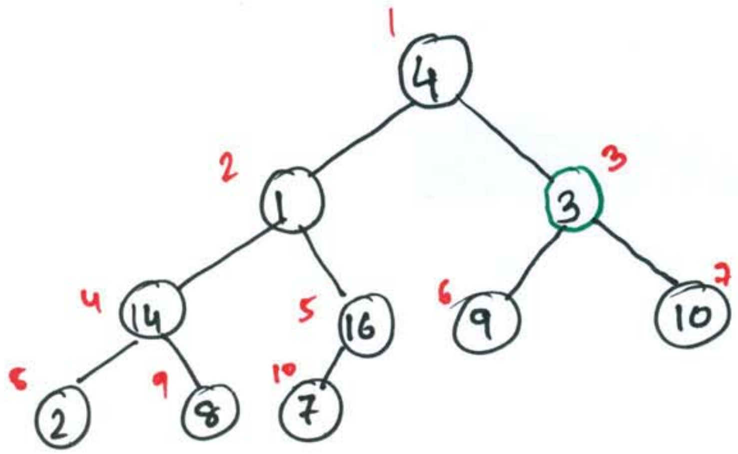
Build_Max_Heap Example

4 1 3 2 16 9 10 14 8 7



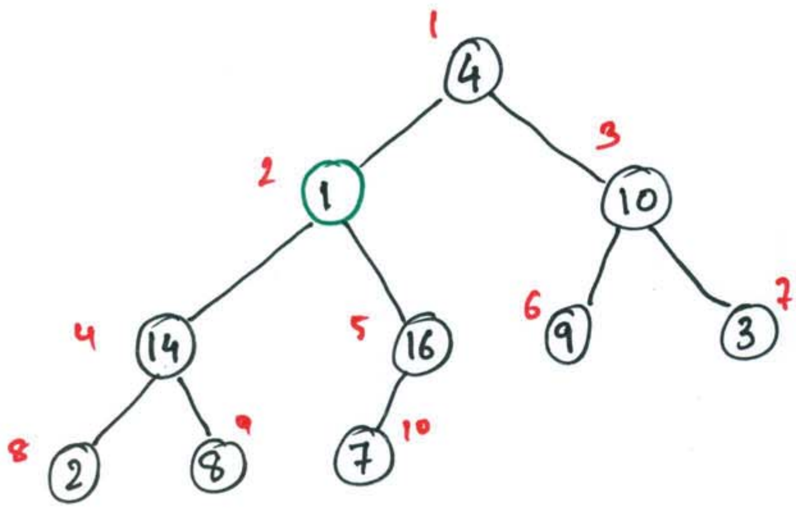
Max_Heapify (A, 5)
no change

Max_Heapify (A, 4)
Swap A[4] and A[8]



Max- Heapify (A,3)

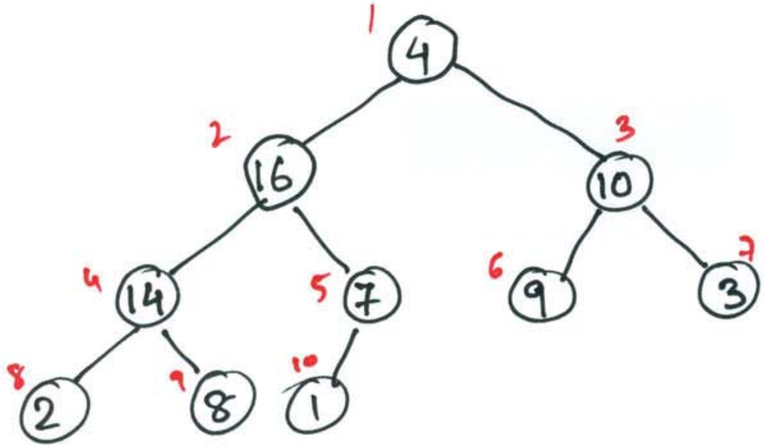
Swap A[3] and A[7]



Max- Heapify (A,2)

Swap A[2] and A[5]

Swap A[5] and A[10]



Max- Heapify (A,1)

Swap [A,1] with A[2]

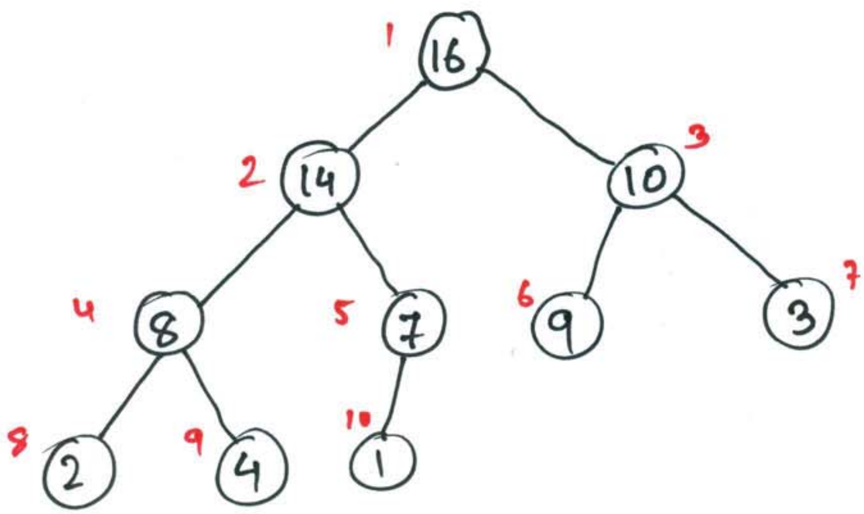
Swap [A,2] with A[4]

Swap [A,4] with A[9]

So,

A: 4 1 3 2 16 9 10 14 8 7

⇓



Build-Max-Heap (A) Analysis

We can observe that Max-Heapify takes $O(1)$ for nodes that are one level above the leaves, and in general, $O(l)$ for the nodes that are l levels above the leaves.

We have $n/4$ nodes with level 1, $n/8$ with level 2, and so on till we have one root node that is $\lg n$ levels above the leaves.

So, total amount of work in the for loop can be summed as:

$$n/4(1c) + n/8(2c) + n/16(3c) + \dots + 1(\lg c)$$

setting $n/4 = 2^k$ and simplyfying we get

$$c\,2^k\left(1/2^0 + 2/2^1 + 3/2^2 + \dots + (k+1)/2^k\right)$$

The term in brackets is bounded by a constant.

This means that Build-Max-Heap is $O(n)$

Heap - Sort

Sorting Strategy

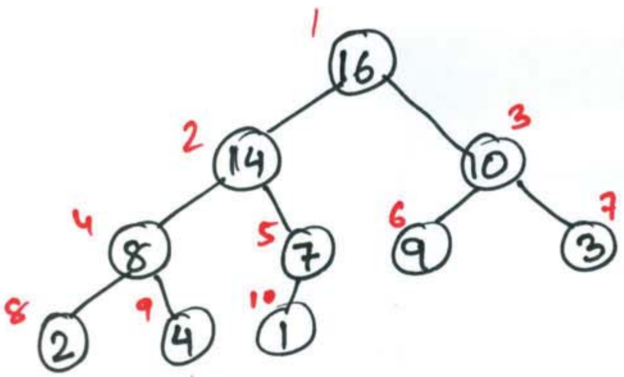
1. Buid Max Heap from unordered array;
2. Find maximum element A[i]
3. Swap elements A[n] and A[i]
   now max element is at the end of array.
4. Discard node n from heap
   (by decrementing heap-size variable)
5. New root may violate max heap property, but it's children are max heaps. Run max-heap to fix this.
6. Go to Step 2 unless heap is empty.

Heap Sort Running Time

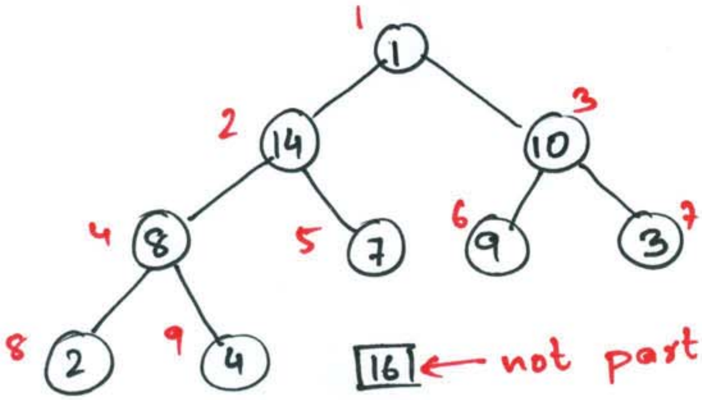
- after n iterations Heap is empty
- every iteration involves a swap and a max. heapify operation; O(logn) time.

Hence, overall : O(nlogn).

Heap-Sort Example



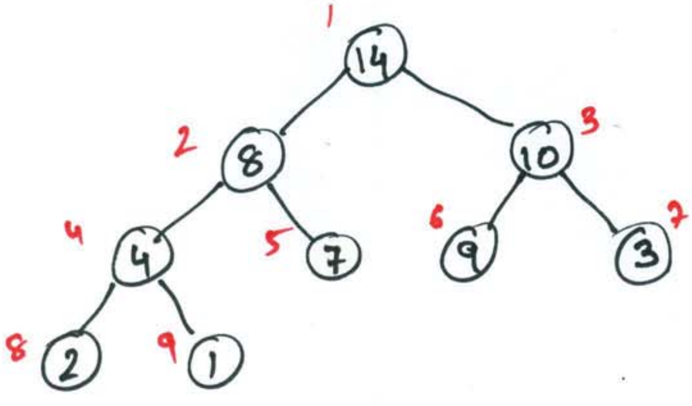
Swap A[10] and A[1]

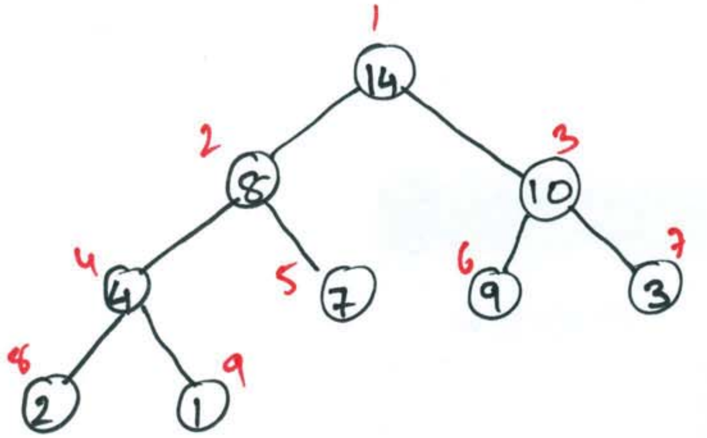


Heap size = 9

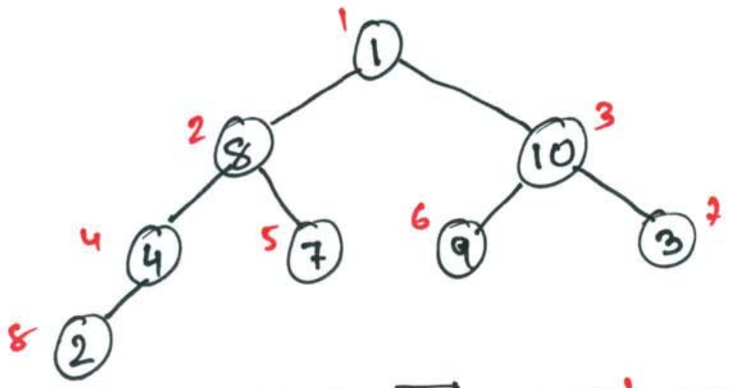
16 ← not part of heap

Max-heapify (A,1)





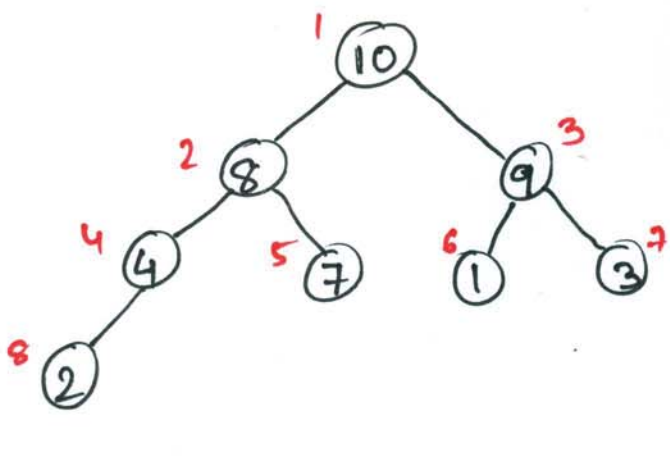
Swap A[9] and A[1]

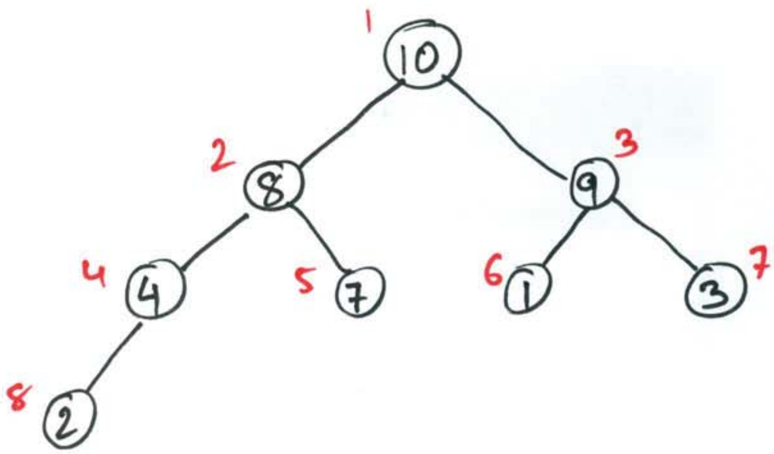


heap size = 8

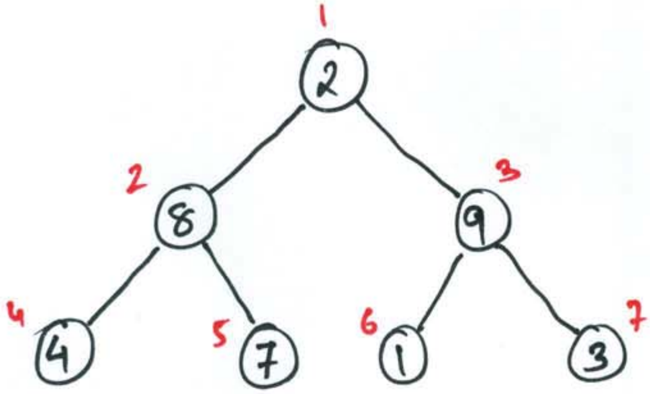
$\boxed{14}$ $\boxed{16}$ ← not part of heap

Max_heapify (A,1)





Swap A[8] and A[1]



8	9	10	
$\boxed{10}$	$\boxed{14}$	$\boxed{16}$	← not part of heap.

How fast can we Sort?

All the sorting algorithms we have seen so far are comparison sorts: only use comparisons to determine the relative order of elements.

- E.g.: insertion sort, mergesort, quicksort, heap sort.

The best worst case running time that we have seen for comparison sorting is $O(n \log n)$

Is $O(n \log n)$ the best we can do?

→ Decision Trees can help answer this question

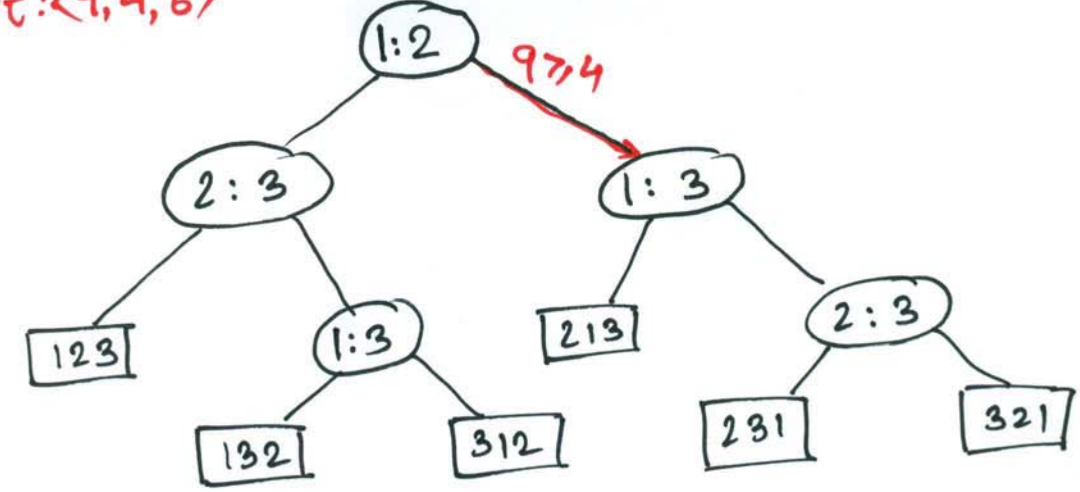
Decision Tree Model

A decision tree can model the execution of any comparison sort:

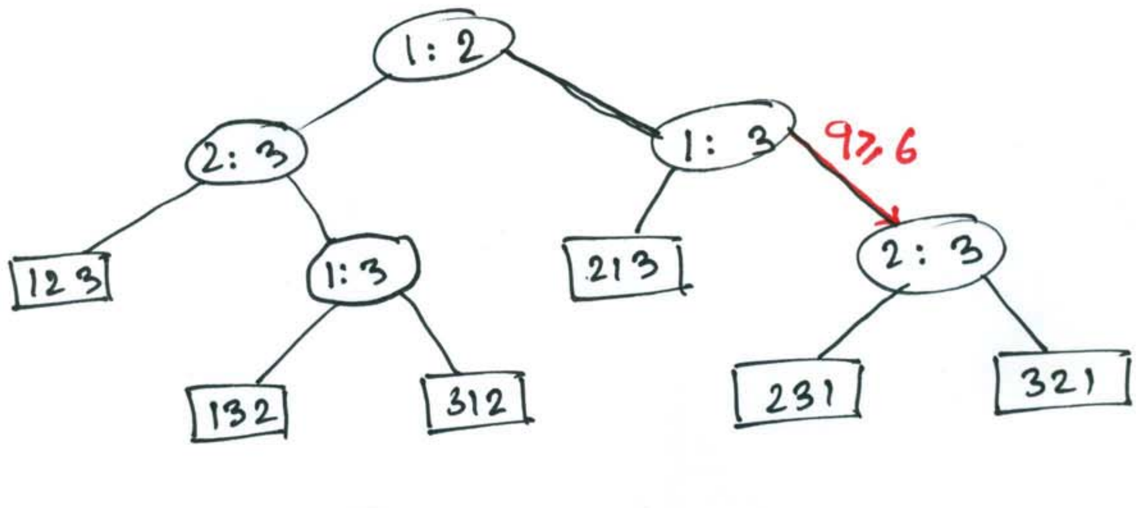
- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of algorithm = the length of the path taken.
- Worst case running time = height of the tree.

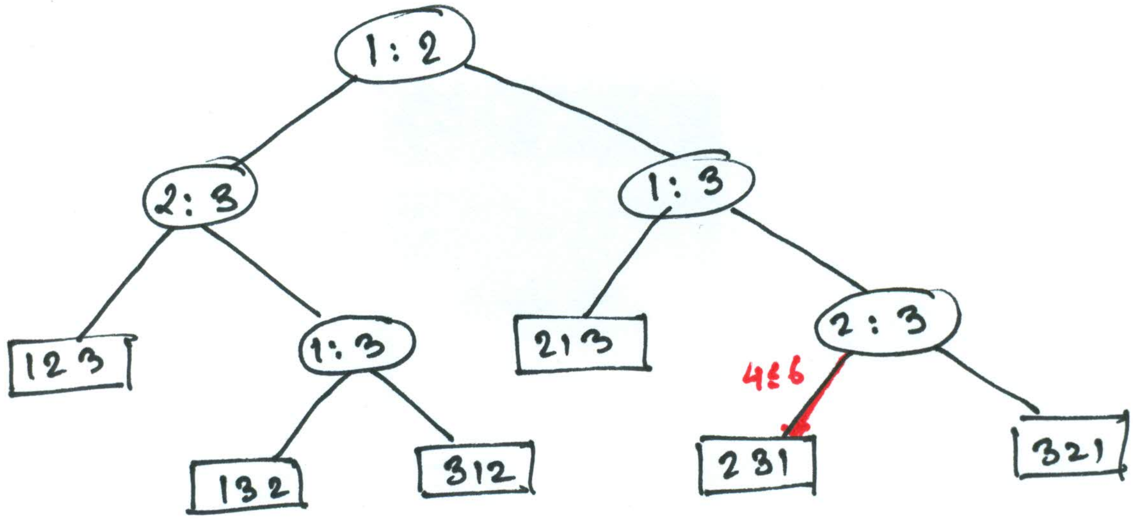
Decision Tree Example

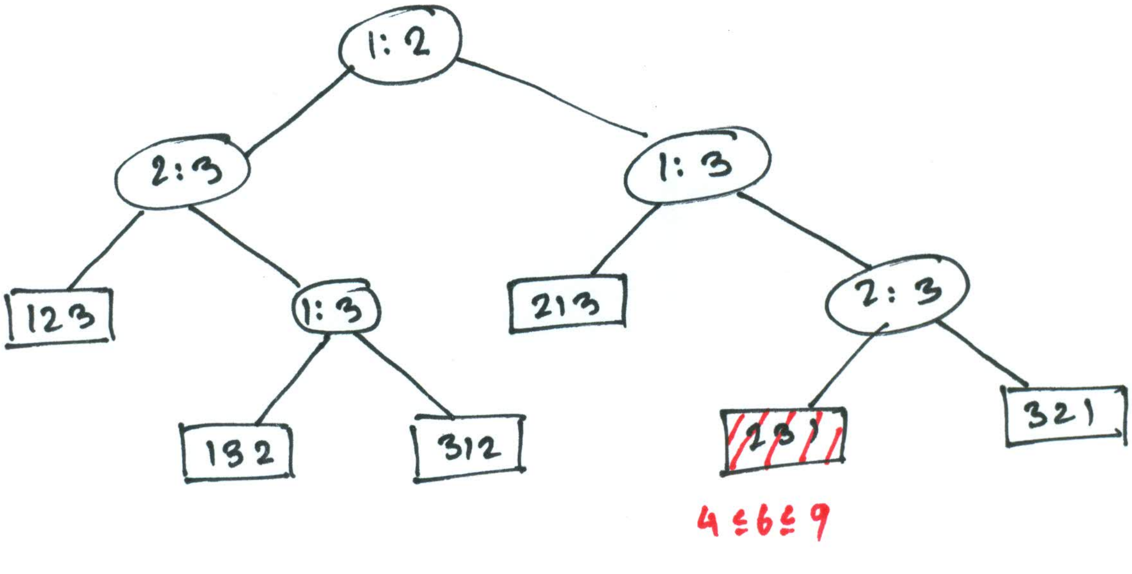
Sort: ⟨9, 4, 6⟩



- Each internal node is labelled $i:j$ for $i, j \in \{1, 2, \ldots, n\}$
- The left subtree shows subsequent comparisons if $a_i \le a_j$
- The right subtree shows subsequent comparisons if $a_i \geq a_j$







Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate the ordering $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ has been established.

Lower bound for decision tree Sorting

Theorem:

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$

Proof:

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations.
A height-h binary tree has $\leq 2^h$ leaves.
Thus $n! \leq 2^h$

$$\begin{aligned}
\therefore \quad h &\geq \log(n!) \\
&\geq \log\left((n/e)^n\right) \quad [\text{Stirling's formula}] \\
&= n\lg n - n\lg e \\
&= \Omega(n\lg n)
\end{aligned}$$

Corollary:

Heapsort and Merge sort are asymptotically optimal comparisons sorting algorithm.