G
O
O
D
O
O
O
D

Week 9 ① Sorting Algo

an array of n elements

$A[1]$ $A[n]$

& put these in sorted order

Merge Sort Quick Sort

Main computational operation? Worst case Average case

$\Theta(n \log n)$

Comparison Complexity

→ * Can we do better than $\Theta(n \log n)$ in the worst case?

Comparison based sorting ⟹ Sorting order is determined based on comparison between i/p elements

Can I do better than $\Theta(n \log n)$ in the worst case?

If it is not possible, can I prove $\Omega(n \log n)$ as a lower bound?

Comparison Model

→ All i/p items are black boxes (ADTs)

→ Only operations allowed are comparisons ($<, \leq, >, \geq, =$)

binary yes/no answer

- We only charge for comparisons
- You can't use anything related to one single value
  - → range, # of digits, ...

Searching — $\Omega(\log n)$ [Compare x with any element]

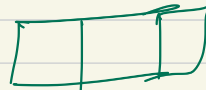
Sorting — $\Omega(n \log n)$ [Comparing two elements]

Binary Search

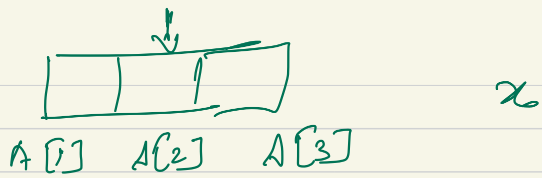
x to search

A[1] ... A[n]

$n = 3$



x



Decision Tree	Algorithm
internal nodes	Binary decision (comp.)
leaf	Answer found
root-to-leaf path	algorithm execution
length of a path (root to a leaf)	running time

Worst case run time ?

⇒ height of the tree

- $A[2] < x$?
  - No: $A[1] < x$?
    - No: $x \le A[1]$
    - Yes: $A[1] < x \le A[2]$
  - Yes: $A[3] < x$?
    - No: $A[2] < x \le A[3]$
    - Yes: $x > A[3]$

↙ exhaustive options

Binary Tree

* if I can figure out the height of the tree, we know the worst-case run-time complexity.

Searching lower bound - $\Omega(\log n)$ in the worst case

x   n items

Proof :→ To prove that the height of decision tree is atleast $\log n$ $\Omega(\log n)$

Binary Tree with $\geq n$ leaves. (one for each answer)

Height of the tree?

→ $\Omega(\log n)$

[Recap: Binary Trees: for n items, the tree height is $\Omega(\log n)$]

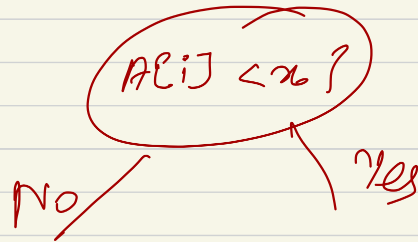
worst-case time complexity

Hence proved

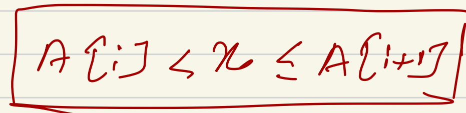
Sorting lower bound

Searching

internal nodes

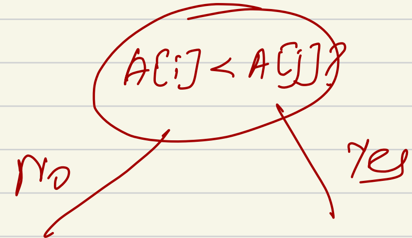


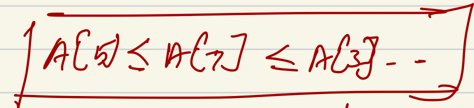
leaf nodes



an answer

Sorting





Sorted order

Goal: Make enough comparisons such that by the time you reach leaf, you know final sorted order

→ The lower bound for comparison-based sorting is

$\Omega(n \log n)$

Proof: → Decision Tree is binary

Goal: Estimate the height

↳ Estimate # of leaf nodes

Searching: How many possible answers? $\geq n$

Sorting: __________ ? Each possible sorted order will denote a leaf node.

leaves $\geq$ # of possible answers

$\geq n!$

Height $= \Omega(\log(n!))$

$$\log n! = \sum_{i=1}^{n} \log i$$

$$\geq \sum_{i=n/2+1}^{n} \log i$$

$$\geq \sum_{i=n/2+1}^{n} \log \frac{n}{2}$$

$$= \frac{n}{2} \log\left(\frac{n}{2}\right)$$

worst-case time complexity

Height $\geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$

proof

Are these examples of non-comparison based sorting?

→ Integer Sorting.

under certain restrictions

Can they achieve better than $\Omega(n \log n)$?

→ linear-time sorting

- You're sorting integers
- Assume that the n items that you're sorting are in range $\{0, \ldots, K-1\}$

I can do other things, compare, add, subtract, ...

$K = O(n)$, we can sort in $O(n)$-time

A[ ]   0 1 1 2 2 0 0 3 4 1 5 2 9

13 items ($n = 13$)

What'll you do?

0 → 9 ($K = 10$)

$K = O(n)$ range 0 ... 9

Find out # of 0's, 1's ... 9's

Counting Sort

declare an array of size K   C[ ]

0 0 0 1 1 1 2 2 2 3 4 5 9

C[ ]

3	3	3	1	1	1	0	0	0	1
0	1	2	3	4	5	6	7	8	9

Output
for i = 0 → K-1
  if C[i] > 0
    print i
    C[i] times

for i = 0 → K-1
  C[i] = 0

Traverse i/p & fill this
for i = 1 → n
  C[A[i]]++

Complexity $O(n+k)$

$k = O(n)$ linear-time sorting

Subtle point :- Generally when you are sorting each item has a key (for sorting), but it can have other things as well.

Sorting on an spreadsheet

{ Name, Roll No, ... Marks }

Student 1

(Marks →) key for sorting

When two or more students have the same marks, which one'd appear before?

→ Preserve the initial order

Abrahim	35
Bishal	27
Chirag	27
⋮	
Mohan	23

sort →

Mohan	23
Bishal	27
Chirag	27
Abrahim	35

Stable Sorting :→ A sorting that ensures that for keys with the same value, the order of the key in o/p is the same as in the i/p array, is called stable sorting.

[* Important while sorting on multiple fields/key
roll no, age, . . . marks]

How do we use this concept in counting sort?

A: an array of n elements, which needs to be sorted wrt a field, key. We impose a restriction that this key value lies between 0 to $k-1$ range.

Use an array C of size K. [integers]

Suppose I want to write output in array B.

C[i] stores the # of elements of A with the key value i

→ Modify C such that while copying A to B, C[i] stores the location where the next element of A with key value i is to be copied in B.

Key for sorting

1	A
2	B
0	B
1	B
0	C
0	D
0	K
0	M
0	P
2	R
2	S
1	S
0	S

$K = O(n)$

A[n] ⟶ B[n] (Sorted array)

make use of C[K]

C[i] tells where the next val i in A would be placed (in B)

1. Create C

0	1	2
7	3	3

C[i] : # times i occurs in A

2. Right shift ←

C

0	7	3

$C[i] = C[i-1]$

$C[0] = 0$

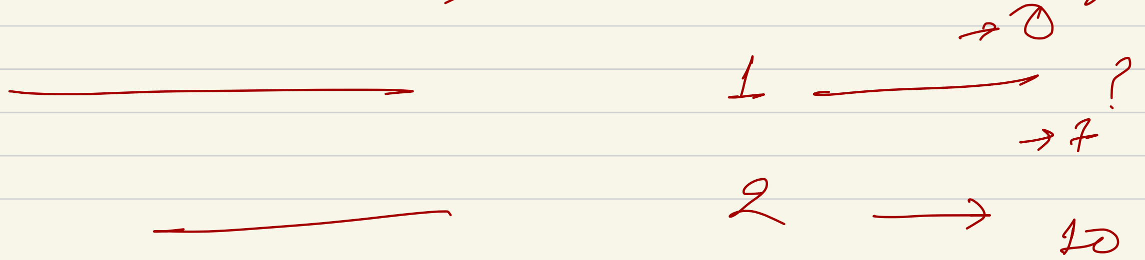
3. Add the values

C

0	7	10

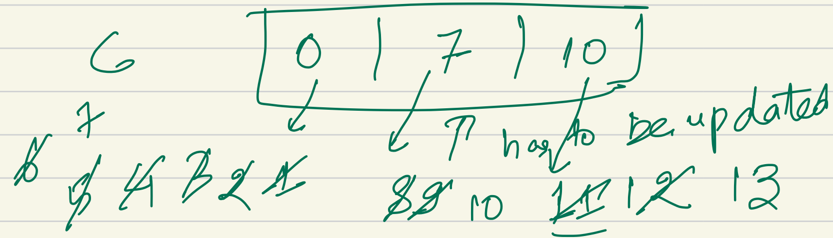
$C[i] += C[i-1]$ →

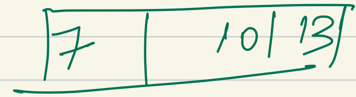
At what location (in B), the next 0 in A would go?



A
1 A →
2 B →
0 B →
1 B →
0 C →
0 D →
0 K →
0 M →
0 P →
2 R →
2 S →
1 S →
0 S →

Modify C[ ] such that C[i] stores location (in B) where the next element of A with key i to be copied





B	0B	0C	0D	0K	0M	0P	0S	1A	1B	1S	2B	2R	2S
	0	1	2	3	4	5	6	7	8	9	10	11	12

* Increment C[i] when you copy a record with key i

Verify that it gives a stable sorting

Complexity? $O(n) + O(k) = O(n+k)$

if $k = O(n)$ → linear time sorting

$K = O(n^2)$

9 keys $\qquad$ 0 to 80

$K = O(n^2)$

array of size 81 $\qquad$ $O(n^2)$ not good.

But you can modify that idea to give a linear time algo for $O(n^2)$ range - - -

$O(n^d)$ range.

Radix Sort

$K = O(n^2)$ $\qquad$ Can we still sort in linear time?

Let A be an array of n integers in the range 0 to $n^2 - 1$ $(k = n^2)$

Leverage on counting sort, stable sort

An integer a in this range can be written as

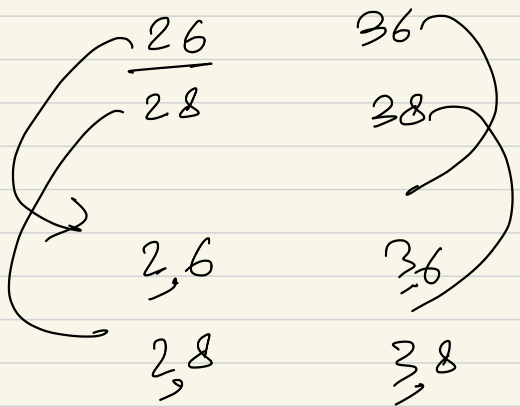
$$a = a_1 n + a_0$$

$a_1 = \lfloor a/n \rfloor$ $\quad$ $a_0 = a \bmod n$

$a_0, a_1 \in \{0, \ldots n-1\}$ — range is $O(n)$

$a = (a_1, a_0)$

counting sort individually!!



10 integers

0 to 99

a_1 a_0

2 6

3 6

2 8

3 8

Stable sort wrt a_1

MSD LSD

MSD → Most significant digit

LSD → Least significant digit

- 26
- 28
- 36
- 38

1. Sort wrt a_1 → stable sort wrt a_0

26
28
36
38

→

26
36
28
38

2. Sort wrt a_0

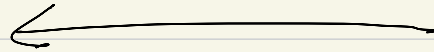
- We first sort the array of such pairs wrt a_0 (LSD)
- Stable sorting wrt a_1 (MSD)

a_1 a_0				
52		30		07
33		41		17
07		52		27
41	sort wrt a_0 →	62	sort wrt a_1 →	30
62		33		33
27		75		41
75		07		47
17		27		52
47		17		62
30		47		75
range 100		LSD		MSD

$K = O(n) \rightarrow K = O(n^d)$

d is a constant

$a = (a_1, a_2, \ldots a_d)$



Counting sort d-times

Radix Sort

$O(nd)$ d: # of digits

$K = O(n^2) \rightarrow O(n)$

Counting Sort

$K = O(n)$

$O(n+K)$

Radix Sort

$K = O(n^d)$

$O(nd)$

Bucket Sort ⟹ Expected $O(n)$

(even if underlying data is not integers)

* Put the elements of the i/p array A of size n into n buckets (or bins).
  ↪ The buckets should be so chosen so that each bucket receive $O(1)$ elements on on average.

* bucketing criterion is very crucial.

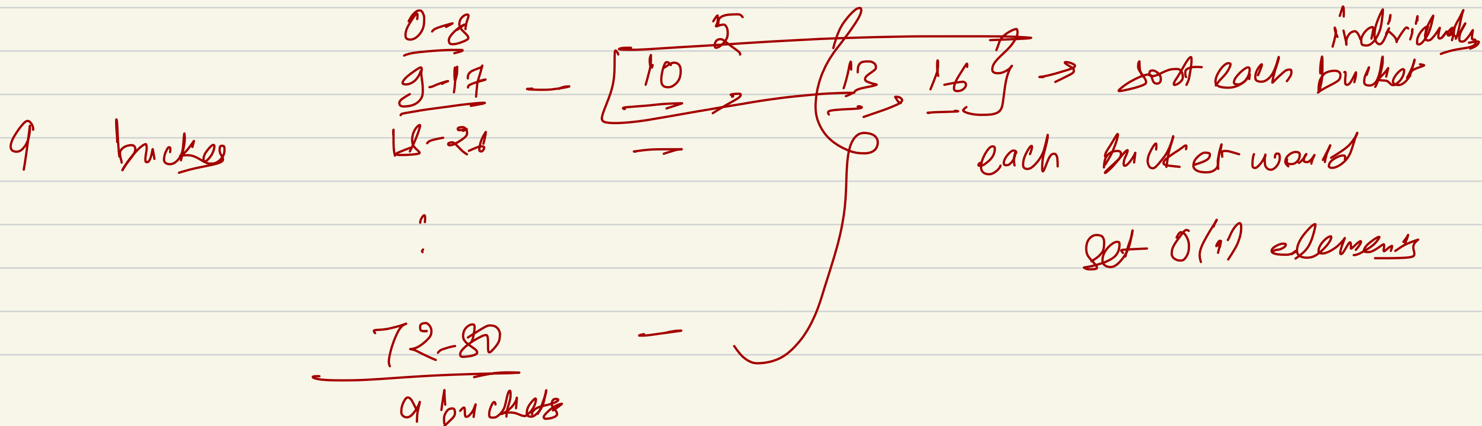
which element goes in which bucket.

What will this depend on?

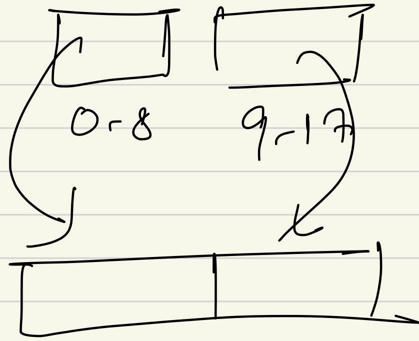
→ depends on the assumption on i/p array

Ex: A of size n = 9 keys. in the range 0 to 80.

uniformly distributed.



- Sort each bucket individually
- Concatenate the sorted buckets



Bucket Sort

Complexity

Expected case

$\Theta(n)$

Worst case :→ all elements in the same bucket

→ Insertion sort ($\Theta(n^2)$)

Merge sort $\Theta(n\log n)$

Counting sort (if integers)

Heaps (Priority Queue ADT)

A heap is a binary tree with the following property

- **Structure**

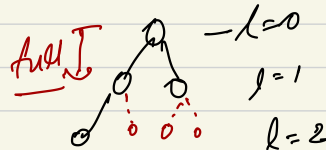
It is an almost complete binary tree. Thus all the levels but last must be completely full, and in the last level, all nodes must appear to the left of the empty positions

- **value**

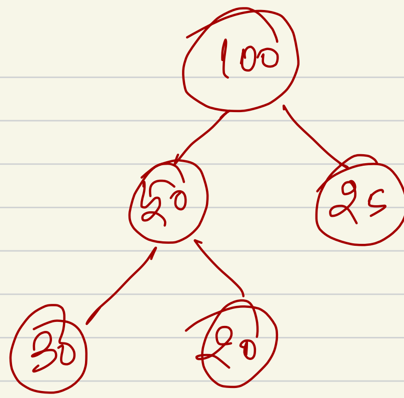
for any node u and its parent p, val(p) ≥ val(u) [Max-heap]

⇒ for any node, the value is larger than any value stored in the subtree.

[similarly for min-heap]



Ex.

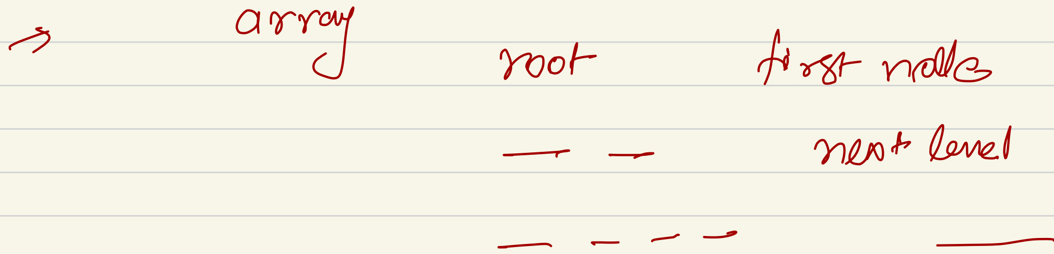


0 → full
1 → full
2 - not full

A heap of max level l contains $2^0, 2^1, \ldots, 2^{l-1}$ nodes at previous levels $(0, \ldots, l-1)$

between $1 \to 2^l$ nodes at level l.

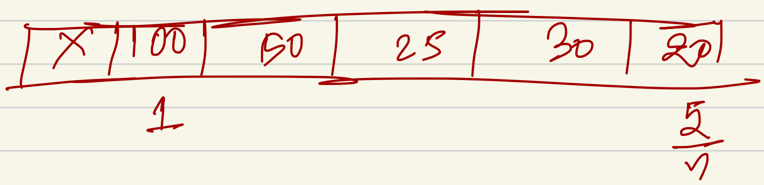
Because of this property, I can represent heap using an array, filling it using nodes from top to bottom (left to right at each level)



array start from 1

n elements

1 n positions



A[1] root

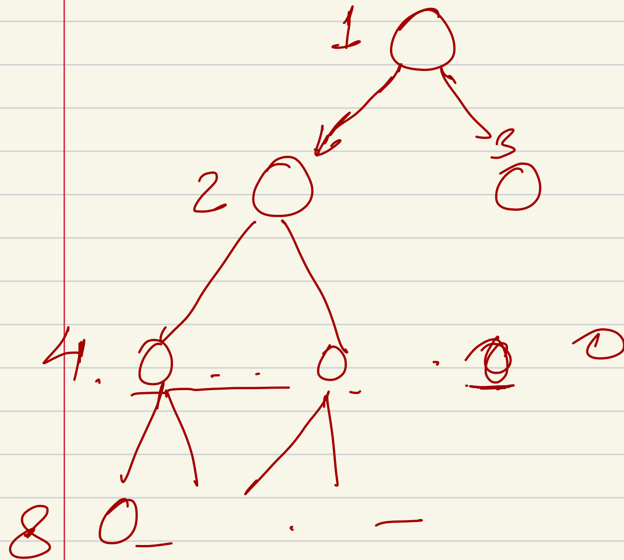
parent → child child → parent

node at index i

Where will the left & right children be located?

left → $2i$

right → $2i+1$



Similarly parent for a node at index i → $\lfloor \frac{i}{2} \rfloor$

- Heap Property
- Represent using arrays

Priority Queue ADT

Receiving some jobs each job comes with a priority

(Maintain a queue for these items.

— Removal according to priority)

Find Jobs with the max priority

Operations ⇒

1. Find Max :→ return the max value in the list
2. Delete Max :- (Return the max value). delete the max value
3. Insert

Priority Queue ADT

	Unsorted Array	Sorted Array	Heaps
Find Max	$O(n)$	$O(1)$	$O(1)$
Delete Max	$O(n)$	$O(1)$	$O(\log n)$
Insert	$O(1)$	$O(n)$	$O(\log n)$

4. Build Heap [You're given n elements & you want to create a heap]

→ ?? better than $O(n \log n)$?

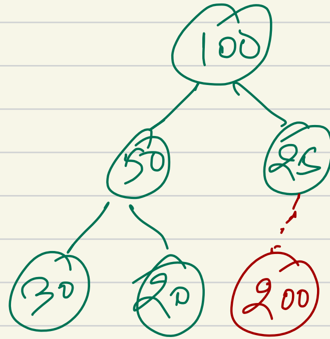
Find Max

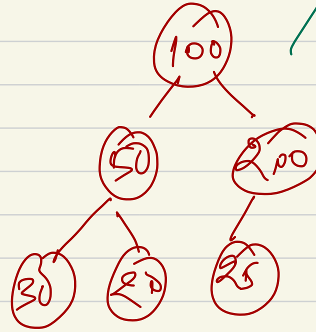
A[1]

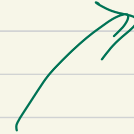
Insert & DeleteMax

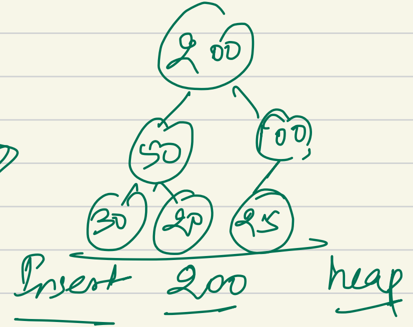
- First ensure structure property
- Then ensure value property

Insert









Insert 200 heap

insert (H, n, newval)

Percolate the value up :- Swap this value with the parent if heap property is violated. Continue this until heap property is restored or the new value reaches the root.

Time Complexity $\Leftarrow$ $\Rightarrow$ Height of the tree

Almost Complete

l levels $\geq 2^l$ children

$h = O(\log n)$ n nodes

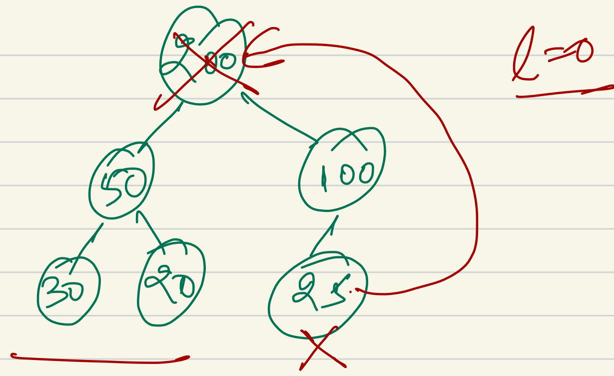
$O(\log n)$

Delete Max :→

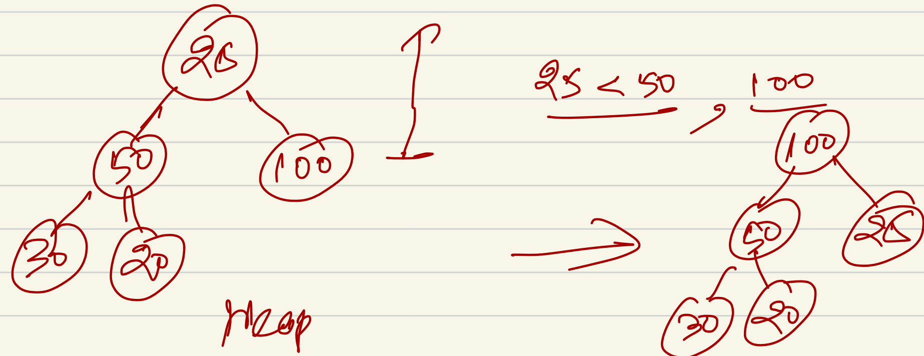
1. Maintain structure property
2. Value property

$l = 0$

DeleteMax



Element at the root is the max value stored in the heap, we delete it by copying the last element of the array to root & decrementing the size by 1



Placement at the root may violate the heap property. In that case, we swap this value with the child having a larger value. Continue until the heap property is restored or you reach a leaf.

→ $O(\log n)$ complexity

Build Heap

→ If elements come one by one

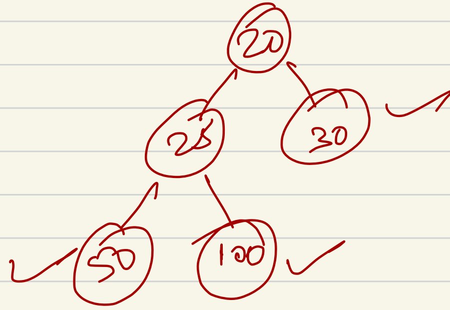
→ Insert one by one

→ $O(n \log n)$

→ If all elements are present together, we can do better.

⇝ bottom up

20 25 30 50 100



bottom-up

Not a Max-Heap

leaf nodes

$n \Rightarrow \lfloor \frac{n}{2} \rfloor$

$\lfloor \frac{n}{2} \rfloor + 1$ to n

leaf nodes already heaps

from $\frac{n}{2}$ to 1, ensure that the subtree rooted at node i is a Heap

When you are an node

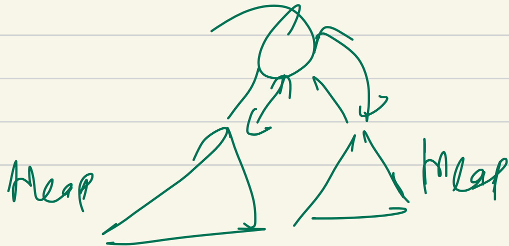
for($i = \frac{n}{2}$; $i > 0$; i--)

MaxHeapify (H, n, i)

Analogous to Delete Max

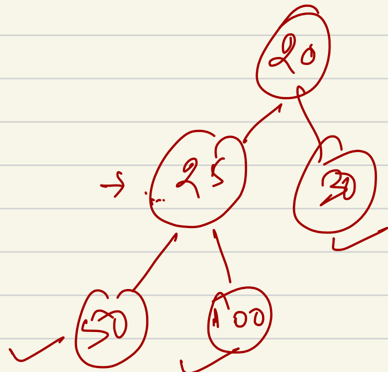
→ Find the max value stored in the children $(2i, 2i+1)$

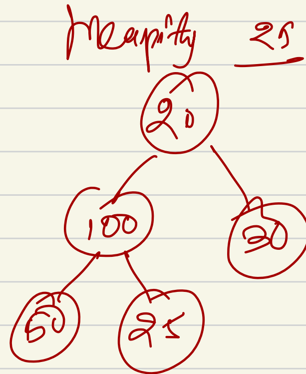
Compare with $A[i]$

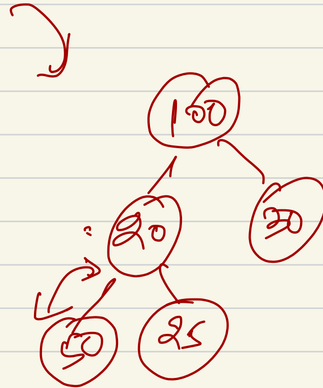


if A[i] < max, swap with max
& call this recursively

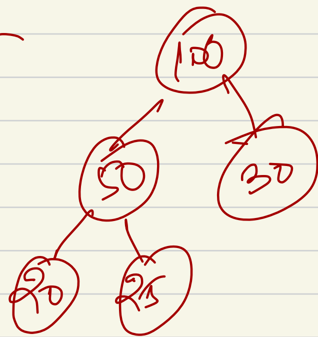
Heapify 25





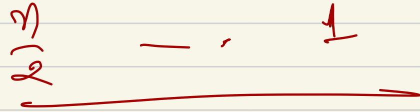


Given a set of integers, heap is not unique



Max Heap

Time Complexity



max Heapify
$\Downarrow$
$\equiv$ deleteMax
~~2~~
~~$\frac{n}{2}\log n$~~

n → nodes

l → max level

$$2^l \le n \le 2^{l+1} - 1$$

at a given level K → atmost 2^K nodes

Heapify on a node at level K → $O(l-K)$ time

$$T(\text{Heapify}) \le \sum_{K=0}^{l-1} (l-K) \cdot 2^K \le n \sum_{K=0}^{l-1} \frac{l-K}{2^{l-K}}$$

K
l
$l-2$
$l-2$

$$n \sum_{k=0}^{l-1} \frac{l-k}{2^{l-k}}$$

$$h = \underline{l-k}$$

$$= n \cdot \sum_{h=1}^{l} \frac{h}{2^h}$$

$$\leq n \cdot \sum_{h=1}^{\infty} \frac{h}{2^h} \qquad = \underline{2n}$$

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \cdots$$

$$-\ \frac{S}{2} = \frac{1}{2^2} + \frac{2}{2^3} + \cdots$$

$$\Rightarrow \frac{S}{2} = \frac{1}{2} + \frac{1}{2^2} + \cdots$$

$$= \frac{\frac{1}{2}}{1 - 1/2} = 1 \Rightarrow \boxed{S = 2}$$

⇒ Build Heap is of complexity $O(n)$

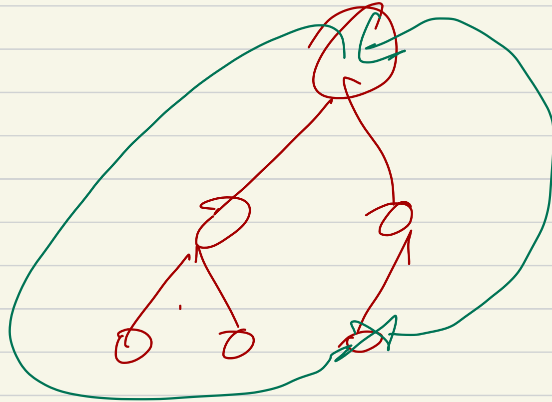
n elements → create a heap in $O(n)$ time

Insert	Delete Max	Find Max	Build Heap
$O(\log n)$	$O(\log n)$	$O(1)$	$O(n)$

Heap Sort → first we build heap from the array

Then we keep on deleting the max element from the array & storing this element in the vacant

post created at the end of the current heap



Heap Sort

Make Heap $\rightarrow O(n)$

Delete Max $\rightarrow O(\log m)$

Heap of size m

$$n + \sum_{m=1}^{n} \log m = n + \log n!$$

$$\leq n + \log n^n$$

$$= O(n \log n)$$

$$\boxed{\Theta(n \log n)}$$

AVL-sort

Heap sort

insert Heap (H, n, newVal) {

}

Heapify (H, n, i)

DeleteMax ( )

MinHeap