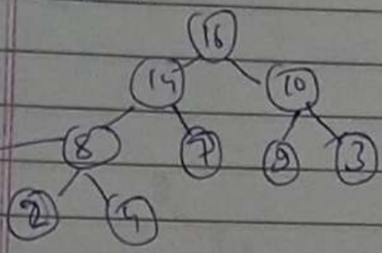
Heaps

Suppose tum esa data structure banana chahte ho jo bahut saari values store kare & jab koi value chaiye ho toh hamesha largest vaali return kare in O(1).

Heap internally work arrays pr krta h, but represent use tree se krte h.



16	14	10	8	7	9	3	2	4

↳ iske first index pr hamesha largest element hoga & baaki part may/may not sorted.

root = i
left child of i = 2i
Parent of i = i/2
right ——→ 2i + 1

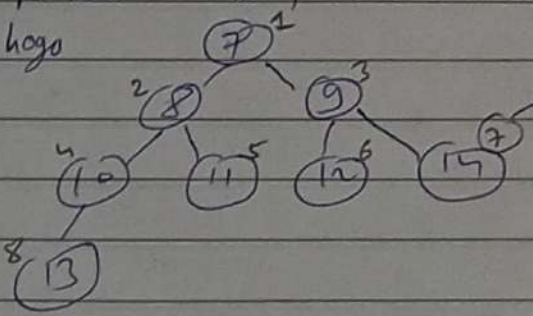
Conditions :- (1) Complete binary tree
(2) Every node value ≥ All of its children

→ Insertion (minHeap → minimum element at first index)

Suppose ye array h →

7	8	9	10	11	12	14	13
1	2	3	4	5	6	7	8

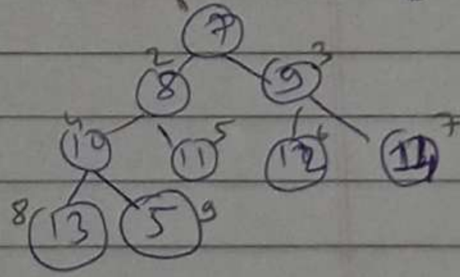
0th index pr kuch nhi h, 1 se chalu kiya array banana & isko BT esa hogo



→ this means ki array ke 7th index pr 14 hai

Now agar 5 ko insert krta toh pehle vo 9th (last) index pr gya

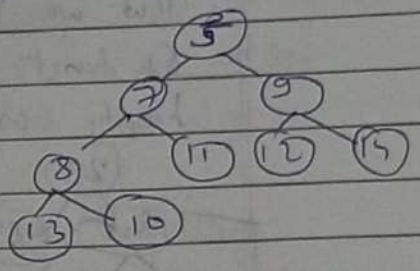
7	8	9	10	11	12	14	13	5
1	2	3	4	5	6	7	8	9



- ab ye dekho ki 5 apne parent se chota h ya nhi if yes than swap coz neeche bade nodes aane chaiye. & parent ka index 1/2 hoga. toh (5) 4th index pr jaaega & (10) 9th pr
for (5) 2nd index pr jaaega fir (1) st index pr & so on toh apun esa hi har level pr check krte rahege ki agar parent bada h toh swap krdo toh TC = O(logn) coz har level pr jaa rhe h check krne

→ Deletion (minHeap)

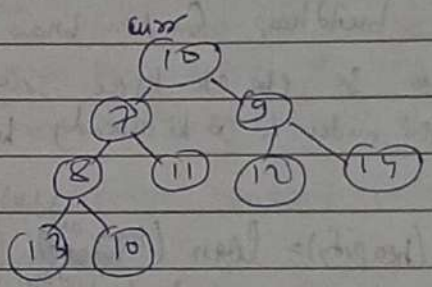
1	2	3	4	5	6	7	8	9
5	7	9	8	11	12	14	13	10



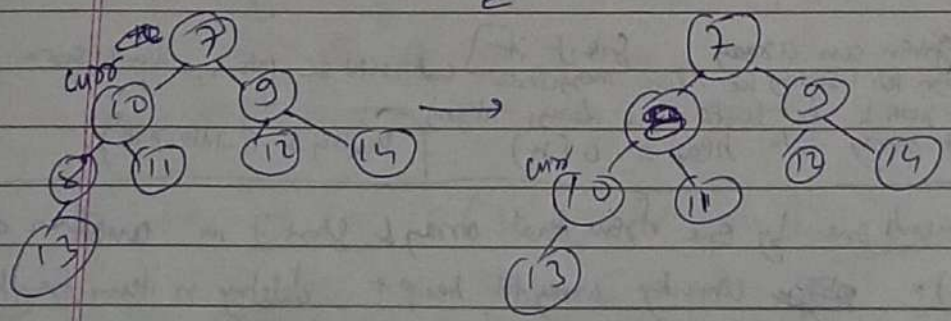
min heap mai top most element ki delete krte hai.

Suppose 5 ko delete kr diya & 5 ko hi krege agar kisi aur ko krna hota toh normally arrays use kr lete, toh jo bhi operation hoge first index pr hi hoge toh last index pr jo h usko 1st index pr dooldo.

10	7	9	8	11	12	14	13



First index se chalu kro agar left/right child use chota h toh swap (left child = 2i, right child = 2i+1)



right h nhi & left wala bada h toh swap h no need to swap. stop the function

TC = O(logn)

• Some more operations: Heap extract max = delete karna max element (root) = logN
Heap increase key = logN

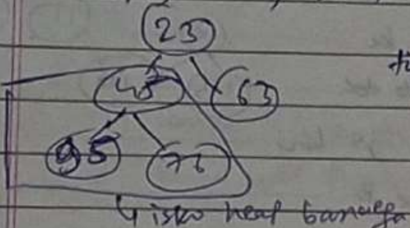
Heapify & Build Heap (O(n))

(Heapify)

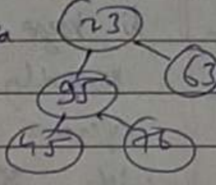
- Suppose you r given an array, and tumhe use max heap mai convert karna h. toh agar n elements h array mai toh agar insert vale code se n elements ko insert karoge toh since TC(insert) = O(logn) & n ... hai toh total n logn ho jaagi.

But we have better way in O(n) to convert it into heap.

Ek funct^n bnao heapify jo ki delete (top down) jaisa hi h 90%, heapify arr, index lega ki kis index se heap banani h



toh index = 3 doge toh 45 ke children se heap bana dega



Ek buildheap function bnao jo ki $\frac{n}{2}^{th}$ index se 1^{st} (bottom up) index se ek ek karke saare index se heapify call karaega. n/2 leaf nodes h jo ki already heap hogi toh $\frac{n}{2}$ se start karege rather than n.

TC (heapify) = logn (same as delete) (top down approach)

TC(Build Heap) = O(n) [bottom up approach from $\frac{n}{2} \to 1$]

Heapsort [given an array, sort it]

(There r 3 algorithm jo nlogn leti h sorting ke, like mergesort = but O(n) SC leti h, Quick sort = worst case m O(n²) ho jaati h but heapsort is always nlogn)

① Convert that array into heap = O(n) (Heapify & BuildHeap se)

② Delete elements one by one from that array & store it in another array.

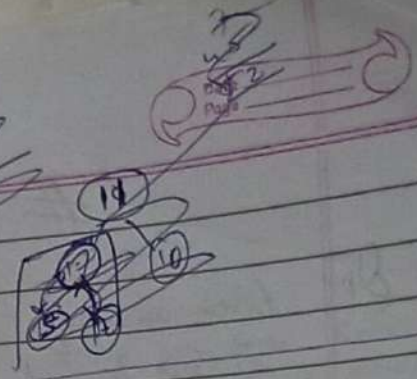
TC (heapsort) = converting array to heap + deleting n items from heap

O(n) + (n)(deleting one item)

= O(n) + nlogn

= nlogn

heapify, build, insert, extractMin
heap.c → insert
heapify.c → heapify, buildheap



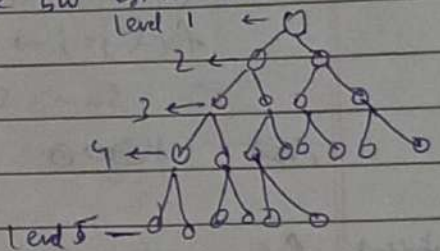
10	15	25	36

Priority queue ⇒ Priority wise elements bahar aana. Suppose priority is ki min element bahar aaye. so you can do it using →

(i) Linked list ⇒ 1 → 2 → 5 → 8 As year 7 insert karna h toh 5, 8 ke beech karlo. ll mai insert krna easy hota h, but it takes O(n) time.

ii) MinHeap ⇒ it will take log n time to insert.

Build heap index 0 to $\frac{n}{2}-1$ tak chalta h means $O(\frac{n}{2})$ & har iteration mai voh heapify call krta h jo ki logN time leta h toh apka according buildheap ki TC = $O(\frac{n}{2}\log N)$ ho jaiye but esi nhi hoti



Reason ⇒ level 5 ko toh consider hi nhi krege coz leaf node h. level 4 mai heapify chalaye toh max-heap mai 1 swap huga apne neeche vaale se

level 5 ⇒ 0 swaps
level 4 ⇒ 1 swap
level 3 ⇒ 2 swap

generalised way →

k^{th} level = 0 swaps = $\frac{n}{2}$ nodes
$(k-1)^{th}$ " ⇒ 1 swap = $\frac{n}{4}$ nodes
$(k-2)^{th}$ " ⇒ 2 " = $\frac{n}{8}$ "

TC ⇒ assuming 1 swap takes C time

1st level = logN swap (jitne level utne swap coz child, root swap hoskte h) ≈ 1 node

$$\left[\left(\frac{n}{2}\right)0 + \frac{n}{4}(C) + \frac{n}{8}(2C) - \cdots\right]$$

it will form AGP whose sum = O(n)

Hashing

Hashcode → given any object, return a unique number for
ex: String s = "shrey"; its generated hashcode is 6
so we can store it in 6th index of array.

0 1 2 3 … 6 (shrey) … 100

but suppose String s = "shrey2" & its hashcode came out to be 734985, then we have to create array of size 734986, so instead we will use hashing function i.e. % 10 or % 100, but then many strings will can have same index so it is called collision

(Margin notes: direct addressing table h & search(T,k) = return T[k]; insert(T,x): T[x.key] = x; delete(T,x) = T[key(x)] = null; no allocation, O(1) mai hoge. So we use hashtable where we use a hash function h(k) to find slot for each key. toh hash function universal set of keys ko m sized array (table) me store karta h.)

→ 2 ways to reduce collision → chaining → open addressing

(1) Chaining

array ke index of linked list bna do toh jitne bhi hashvalue 3 hoga sab ll mai add hote jayega

3 → kunal → shrey → Ashutosh

toh is case mai TC ÷
(i) search depend krega ki us slot mai kitne keys h.
ii) delete = depend krega us slot mai kitni keys h
(ii) insert O(1) coz us slot mai index only value dalna h, toh us slot ko iterate nhi krna padega.

Since most of operation r depending on size of slot toh avg case calculate krte h by using load factor

Worst case is when all are on same index toh O(n) lag jayega time so we do this →

n = no. of keys in table
m = size of table

load factor = $\alpha = \frac{n}{m}$ = expected key per slot

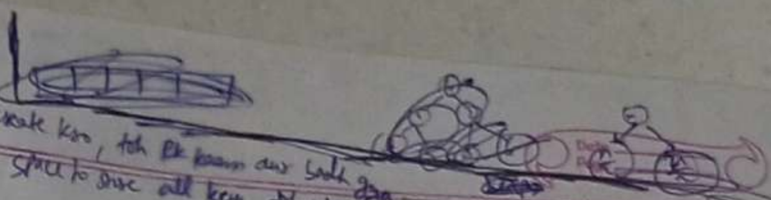
toh TC = $O(1+\alpha)$ ⇒ 1 to find hashcode & us particular index pr α items hoge toh $O(\alpha)$ search krne ke liye

So TC analysis show ki hashtable banate time hume size of table (m) par focus na krke α par focus krna chaiye.

→ Division method, multiplication method, Universal hashing are all just mathematical formulas to generate a hashcode.

Select a hash function at random, from a designed class of functions at the beginning. Probability of collision = $\frac{1}{m \text{ (size of table)}}$. function = $h_{a,b}(k) = ((ak+b) \% p) \% m$, p = select a prime no., a, b = select from {0, 1, 2 …… p-1}

but chaining mai ek disadvantage hai ki usme space use hoti h toh ek naya data structure LL create karo, toh ek kaam aur badh gaya.



So if we have enough space to store all keys toh chaining na use karke open addressing use karo.

(2) **Open Addressing** (table mai hi jayega data & use nhi karte)

→ One item per slot ⇒ $m \geq n$ ⇒ α should be lesser than 0.5

→ probe = try

→ Probing strategies

(1) Linear probing = $h(key, i) = (h(key) + i) \% m$ (h = hash function, i = trial number, m = size of table)

a good function should be able to produce m! probe sequences

0	
1	
2	38
3	
4	46
5	
6	97
7	

Suppose key = 34 as $h(34, 1) = 2$ aaya but 2 par already kuch h toh second try karo $h(34, 2) = 6$ but 6 mai bhi kuch h toh 3rd try karo $h(34, 3) = 3$ as 3 par 34 daal do.

But isse clustering ki problem hoti hai so use double hashing, dikkat linear probing m ye thi ki ek linear factor se increase ho rhi thi value isliye cluster form ho rha tha but double hashing mai, value kitne se increase hogi wo linear nhi ek function h, toh value jyada increase hogi toh cluster form nhi hoga.

(2) **Double Hashing** (can generate m^2 probe sequence maximum)

$$h(k, i) = (h_1(k) + i * h_2(k)) \% m$$

→ search time depend on length of probe sequence.

→ The avg. no. of cells that r examined in an insertion using linear probing = $\left(1 + \frac{1}{(1-\alpha)^2}\right) / 2$

→ avg no. of cells examined in unsuccessful search in linear probing = $\left(1 + \frac{1}{(1-\alpha)^2}\right) / 2$

avg ——— successful search ——→ $\left(1 + \frac{1}{1-\alpha}\right) / 2$

★ for chaining $\alpha = 1$
for open addressing = $\alpha > 0.5$