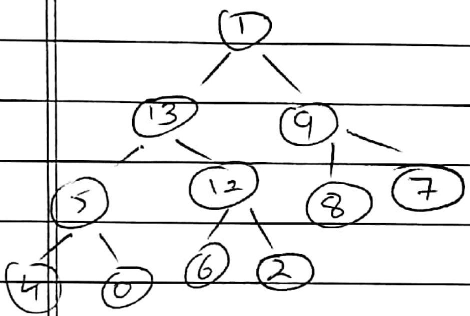
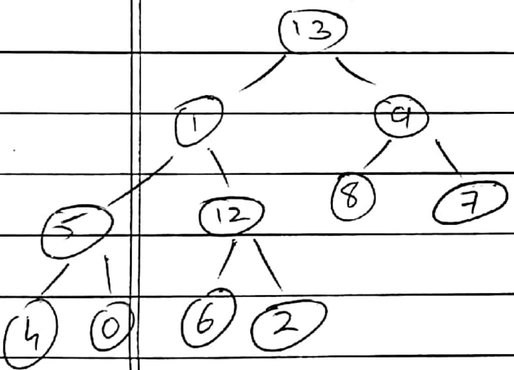
(1) Given Binary heap A = <15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1>.

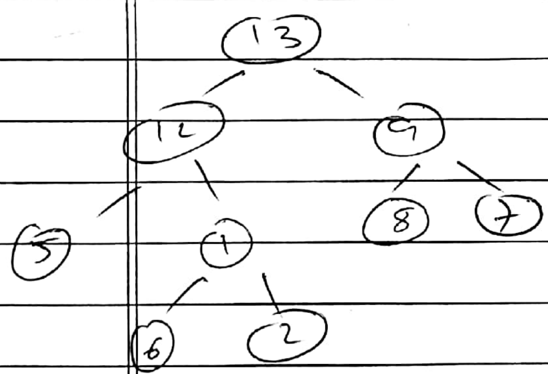
Using Max-heapify we set the first element as MAX = 15.
Now we construct a tree from the remaining we get:-



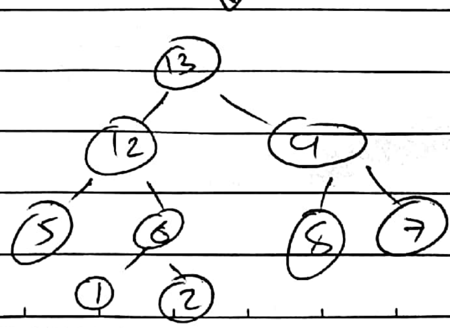
→ We do a shift down on this tree we get:-



→ Do one more shift down



→ One More shiftdown



Now since the root $= 13 < Max$

Max of the heap $= Max$

$= \underline{\underline{15}}$

Scanned by CamScanner

(2) To sort the array in nlogn time we use two steps:-

→ Sort the array elements using Merge sort in O(nlogn) time

→ Traverse the array by creating an auxillary array in O(n) time

∴ The total time = ~~k~~O(n) + O(nlogn)
= O(nlogn)

Algorithm :-

input : Array a of length 'n' A[n]

```
void Mergesort ( int ~~a~~ A [] )
{
    if ( length == 1) return A
    else
    {
        int a1[] = a[0] ... a[n/2]
        int a2[] = a[n/2+1] ... a[n] ;

        a1 = Mergesort (a1);
        a2 = MergeSort (a2);
        return(Merge (a1, a2));
    }
}
void Merge ( int a1[], int a2[])
{
    int arr[] = new int[];
    while ( a1[] != empty and a2[] != empty)
    {
        if (a1[0] > a2[0])
        {
            arr[Maxsize(arr)] = a2[0];
            del a2[0]; Maxsize(arr)--;
        }
        else { arr[Maxsize(arr)] = a1[0];
            del a1[0]; Maxsize(arr)--; }
```

```
~~Maxsize(arr)--;~~
while (a1[]!= empty)
{
        ~~a[0]  arr [Maxsi~~
          arr [Maxsize(arr)] = a1[0];
            del a1[0] ;
            Maxsize (arr)--;
}
while (a2[]!= empty)
 {
          arr [Maxsize(arr)] = a2[0];
            del a2[0].
            Maxsize(arr)--;
 }
 return   arr;

// Driver  code
  sorted_array[] = Mergesort (A[]);   ~~int temp[]~~
   int  index=0;
   if (len (~~A~~ sorted_array)==1)
  {
          print ("Only 1 element in array");
   }
  elif (len ( sorted_array) == 0)
  {
          print (" No element  array  empty");
  }
  else {
   int  auxillary [];
   for (i=0; i< ~~A~~ len (~~arr~~ A) -1  ; i++)
    {
            if (~~arr~~ sorted_array [i] != sorted_array [i+1]){
        auxillary ~~temp~~ [index++] = Sorted_array [i];
     }
   ~~tep~~ auxillary [~~tem~~ index ++] = sorted_array (len(arr) -1];
   print (" Sorted  array:" + auxillary);
```

(13)

(3) Now let's say we have n operations

operation 1 → cost = 1

operation 2 → cost = 2 — power of 2

operation 3 → cost = 1

operation 4 → cost = 4 — power of 2

⋮

operation 8 → cost = 8 — power of 2

⋮

let total cost $= \sum_{i=1}^{n} cost_i$

$$\therefore \sum_{i=1}^{n} cost_i \leq n + \text{powers of } 2 \quad \text{— (1)}$$

$$\text{powers of } 2 = \sum_{j=0}^{\log n} 2^j = n + \frac{2^{\log n+1} - 1}{2-1}$$

$$= (2n-1) \quad \text{put in eq (1)}$$

$$\therefore \sum_{i=1}^{n} cost_i \leq n + 2n - 1$$

$$\sum_{i=1}^{n} cost_i \leq 3n - 1 \sim 3n$$

Hence if $n \Rightarrow$ no. of operations

$$\frac{\text{Total cost}}{\text{Total operations}} < 3$$

hence amortized cost per operation $\Rightarrow O(1)$

(4) We have two operations here PUSH and POP
So we'll assign two credits say 2k to each operation PUSH and POP
- PUSH will cost one credit ie 'k' and the other will be stored in the stack.
- Similarly POP will cost one credit ie 'k' and the other operation will be stored in the ~~stack~~ stack.
- So by the time the stack finishes all the operations it has k credits
- So if there are ~~k~~n operations the stack has O(K) credits = O(n) credits)
- The amortized cost is O(1) because of which the cost never goes in negative
- Now after k operation multiple PUSH and POP operations occur between a copy function in the stack. This results in O(K) credits for k operations and O(n) for total of 'n' operations.

(5) Now the counter out here is an array of bits

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

- We define a variable var1 to hold the highest-order index of 1 in this case var1 = 7.
- Everytime the value of counter is changed the value of variable1 = var1 is updated
- ~~For this particular operation we assign 4 total credits to the whole operation~~

Increment:-

```
counter = 0
while counter < len (A) and A[I] == 1
{
      A[I] = 0          // A => array
      Counter = counter + 1
}
```

~~RESET:- while~~

```
If counter < len(A)
{   A[I] = 1
    if (I > var1)
      {
        var1 = I
      }
}
```

var 1 => global variable consisting the index of high-order of 1 of the counter.

RESET.

```
while ~~Arg~~ var1 > 0
{
    A[var1] = 0
    var1 = var1 - 1
}
var1 = 0.
```

The amortized cost is O(1) so cost never goes in negative

So whenever we set a bit to 1 we use a total of ~~two~~ 2K credits.

1K is the total cost of the operation

1K stays with the bit ~~to~~ for reset operation

So for example when all the bits are set each bit still has 1K credit to RESET themselves

Thus the total amortized cost = O(n), since the cost of every operation is constant

References:- http://www.columbia.edu/~cs2305/courses/csor4231.F15/amortized.pdf

⑥ let the operations have following credits for the following Operations

PUSH : 2k ——— 2 credits
POP : 0k ——— 0 credits
MULTPOP: 0k ——— 0 credits

The explaination to this is :-

- whenever we push something on the stack we push the object in the stack with 1k as the cost of operation and 1k as credit on the item.
- whenever we use POP or MULTIPOP operation we use the one credit from the ~~plate~~ variable and use it for the operation.
- Thus by giving some extra credits for the ~~POP~~ PUSH operation POP operation can be given 0 credits

Now the total cost of operation $\Rightarrow O(n)$
average amortized cost $\Rightarrow O(1)$

$$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} c_i + f(D_i) - f(D_{i-1})$$

$$= \sum_{i=1}^{n} c_i + f(D_i) - f(D_0)$$

$$= \sum_{i=1}^{n} c_i + S_0 - \cancel{S_0}^{S_n}$$

$$\leq 2n + \cancel{S_n}^{S_0} - \cancel{S_0}^{S_n}$$

Now when the stack is growing $S_0 \leq S_n$

$$\therefore \sum_{i=1}^{n} c_i' = O(2n + S_0 - S_n)$$

$$= O(n)$$

when stack is decreasing $S_0 \geq S_n$.

Reference:- http://www.columbia.edu/~cs2035/courses/csor4231.F15/amortized.pdf.

7) Given Keys A = < 5, 28, 19, 15, 20, 33, 12, 17, 10 >
where each element is a Key

∴

K		h(K) = K Mod 9
5	→	5
28	→	1
19	→	1
15	→	6
20	→	2
33	→	6
12	→	3
17	→	8
10	→	1

Hash table

Slots		Keys
0	→	Null
1	→	28 — 19 — 10
2	→	20
3	→	12
4	→	null
5	→	5
6	→	15 — 33
7	→	null
8	→	17

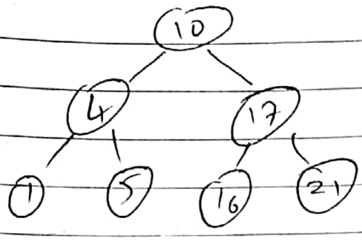
Algorithm.

```
class Data {
 int data;
 int key;
}
int funcMod(int Key){
 return key % 9;
}
void insert(int key, int data){
 ~~item~~ Data obj = new Data()
 obj.data = data;
 obj.Key = key;
 int hashindex = funcMod(Key).
 while(hashtablerow[hashindex] -> key == -
 and hashtablerow[hashindex] != NULL)
 {
   C++;
   hashindex = hashindex % total size;
 }
 hashtablerow[hashindex] = item;
}}.
```

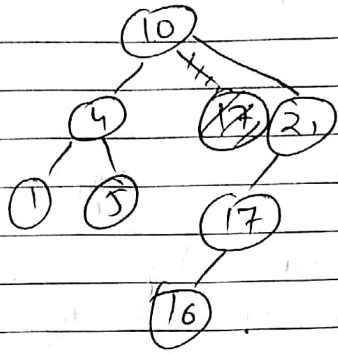
Scanned by CamScanner

⑧ Given set of keys :- $<1, 4, 5, 10, 16, 17, 21>$

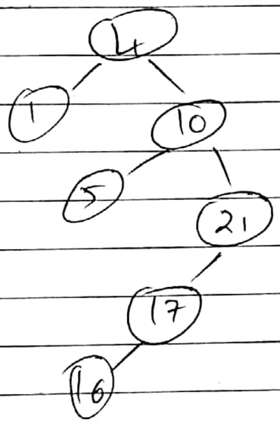
Height 2



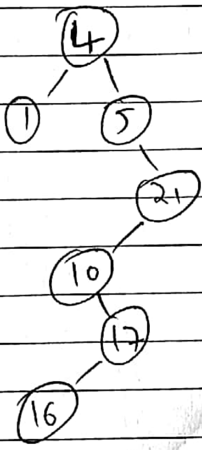
Height 3



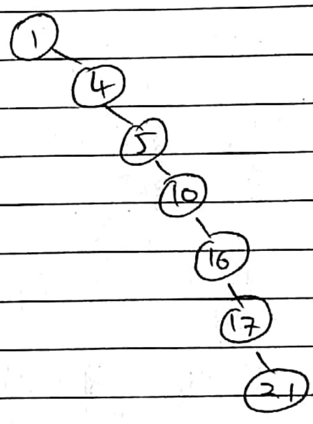
Height 4



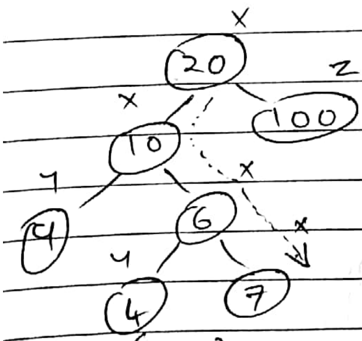
Height 5



Height 6



⑨ let $X \Rightarrow$ search path elements.
$Y \Rightarrow$ elements on left of the search path
$Z \Rightarrow$ elements on right of the search path



The claim is $a \in X$; $b \in Y$; $c \in Z$

$a \le b \le c$

~~Here $a \in \{20, 10, 6, 7\}$~~
~~$b \in \{9, 4\}$~~
~~$c \in \{100\}$~~
~~$a \le b \le c$~~

$a \in \{9, 4\}$
$b \in \{20, 10, 6, 7\}$
$c \in \{100\}$

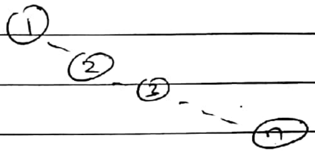
$9 \le 6 \le 100$ — Fails
$9 \le 7 \le 100$ — Fails

} 2 cases which fails the theory

(10) Algorithm to convert Array to Binary search tree :-

```
Node insert (node, value) {
  if node == null {
      return;
  }
    if (value > node.value) {
        node.right = insert (node.right, value);
    }
    if (value < node.value) {
        node.left = insert (node.left, value);
    }
}.
```

- The worst case would be when the array is sorted so its basically a linked list as all the nodes get added on the right.



$\therefore$ Time for Node 1 = 1

" Node 2 = 2

⋮

n

Total time = $1 + 2 + 3 + 4 + \ldots + n$

$= \frac{n(n+1)}{2} = O\left(n^2 + \frac{n}{2}\right) \approx O(n^2)$

- The Best case is when all the branches of the tree are of approximately the same length.

So the time for each insertion and traversal would be :-

total time = $n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} \ldots \frac{n}{n} = n \log n$

$= O(n \log n)$. The branch length is $\log n$