Date .....
Page .....

B-Trees

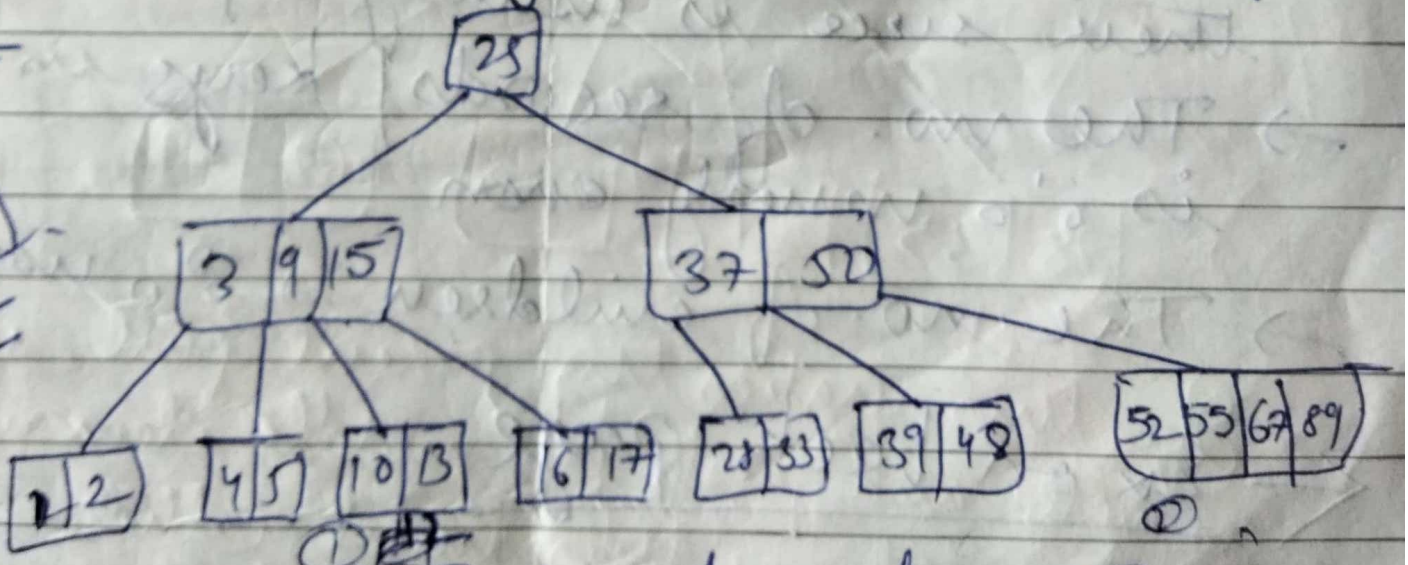
It is good example of a data structure for external memory. better than binary search trees.

→ Each node in a tree should correspond to a block of data Each node stores many data items & has many successors

→ The tree has fewer levels but search for an item involves more comparisons at each level.

→ For database search, it is more imp. to minimise page swaps than comparis

Ex



B-Tree of order 5

Definition of a B-tree of order m

→ There is a single root node, which may have only one record (key) & two children or none if the tree is empty

→ All the other non-leaf nodes must have between m/2 & m+1 ordered records (keys) and m/2 +1 & d^m children

→ All keys in the i^{th} subtree of a node are less than the i^{th} key (if exists) & greater than the i-1st key (if exists)

→ All leaves on the tree must be at the same level

Practical consideration

→ Since each node correspond to a block/page their size is usually a power of 2

→ The no. of records/keys in a node is usually even

→ The no. of children is usually odd.

← Ex on back page

Date
Page

Search in a B-Tree

→ It is same as is done in m-way search tree

Inserting in a B-Tree

→ Find the node where the item is to be inserted by following the search procedure
→ If the node is not full, insert the item into the node in order
→ If the node is full, it has to be split

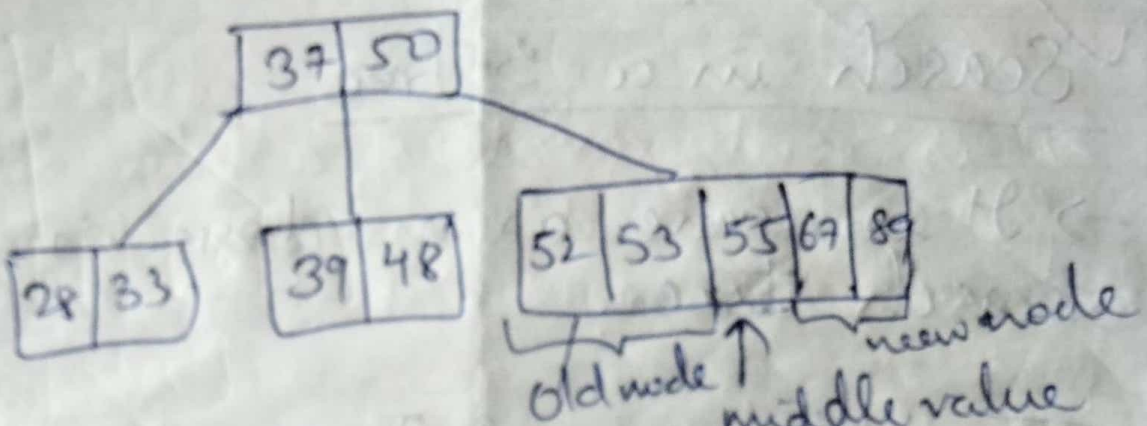
Insert 12 in fig 1

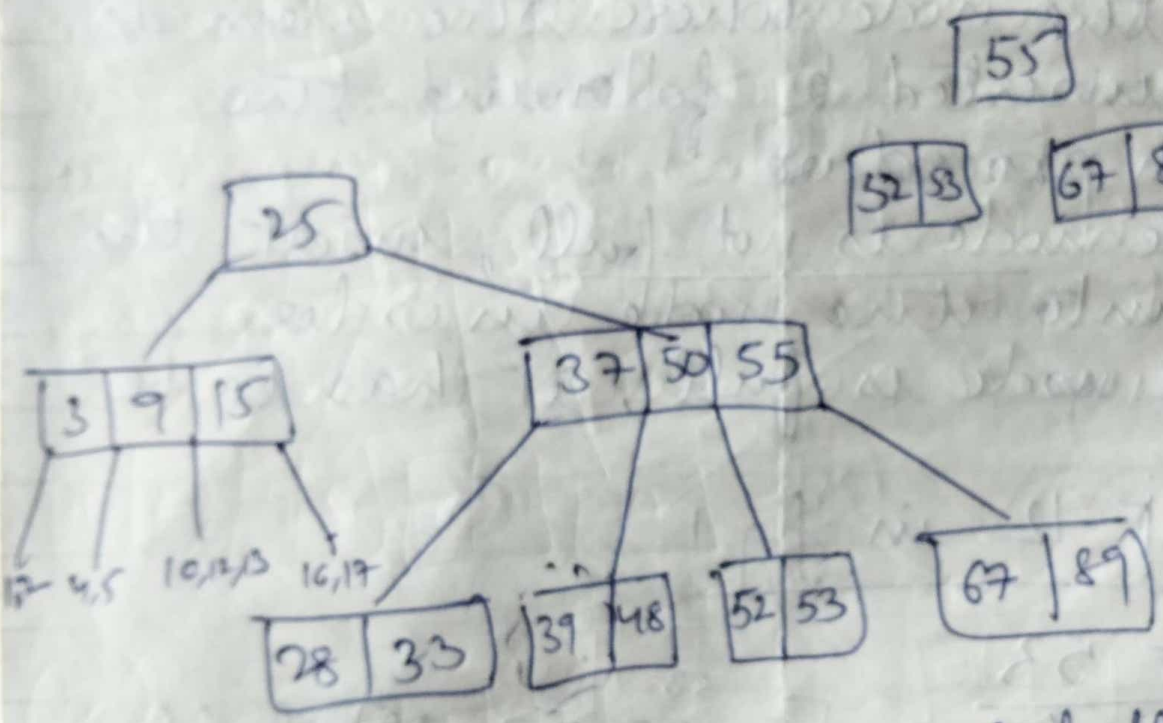
Insert 53

→ See node [52 55 67 89] full (53)

→ Splitting the node
- Find the middle value (including newvalue)
- Keep records with keys smaller than middle in the odd node
- Put records with keys greater than middle in the new node.

- Push middle record up into parent node.





→ If this makes the parent full, split it as well and push its middle item upwards
→ Continue doing this until either some space is found in an ancestor node or a new root node is created.

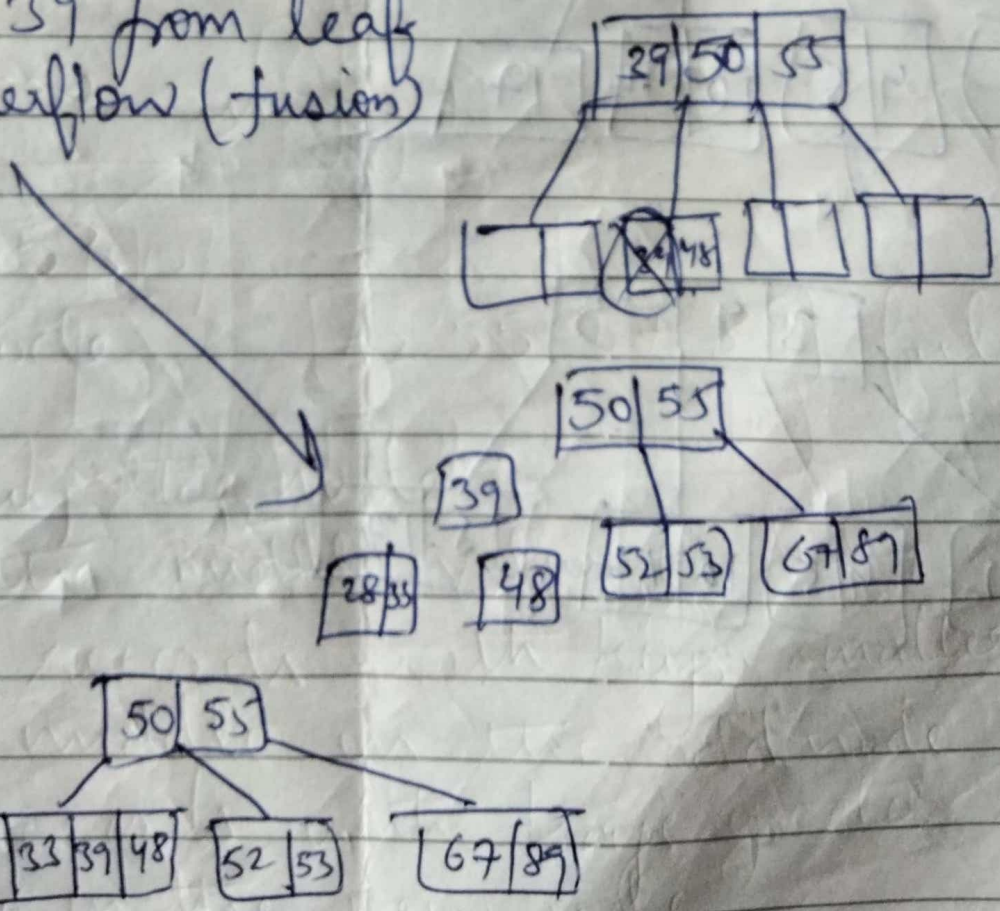
Date .....
Page .....

Deletion

→ As in (2,4) trees, replace the item to be removed by its in-order successor.
→ Remove successor from its leaf node.
→ This may cause underflow (i.e fewer than $m/2$ keys in the leaf node)
→ Depending on how many records the sibling of u has, this is fixed either by fusion or by transfer.

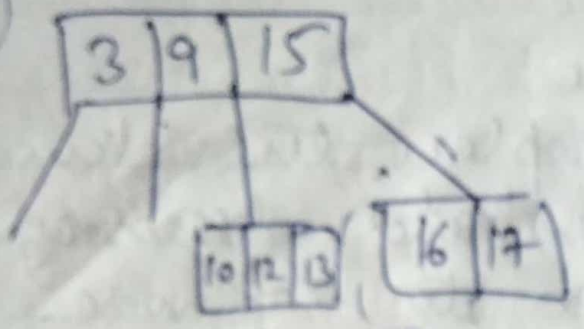
Delete 37

→ Find replacement (39)
→ Replace
→ Remove 39 from leaf
→ fix underflow (fusion)

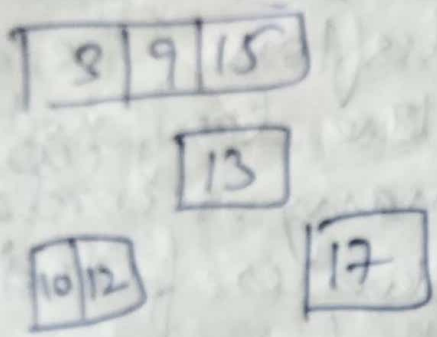


Delete 16 → No need to replace
→ Underflow (transfer)

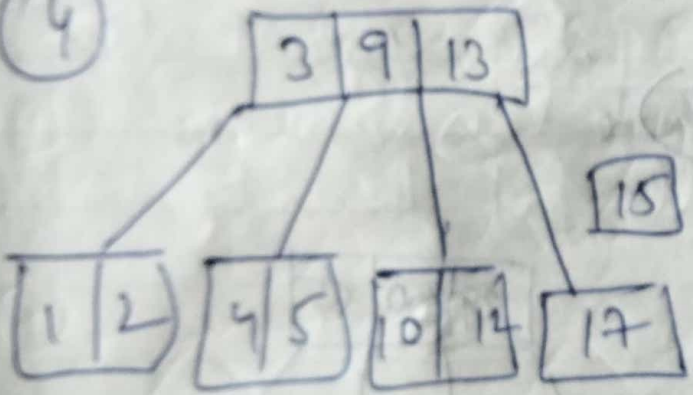
①
②



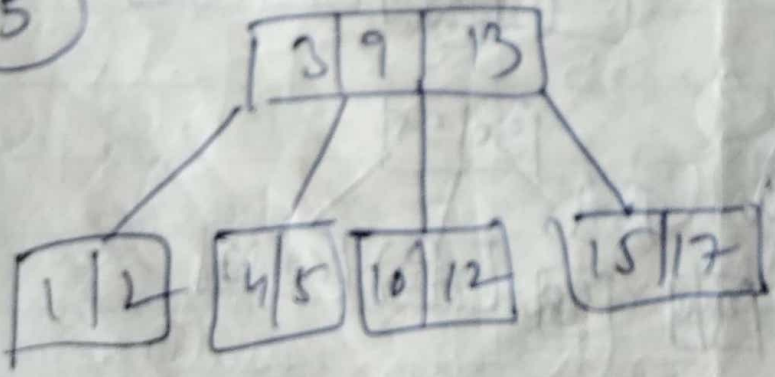
③



④



⑤



HEAP

A binary heap is a simple data structure that efficiently supports the priority queue operations.

Suppose H is a complete binary tree with n elements. Then H is called a heap or a maxheap if each node N of H has the following property

→ The value at N is greater than or equal to the value at each of the children of N

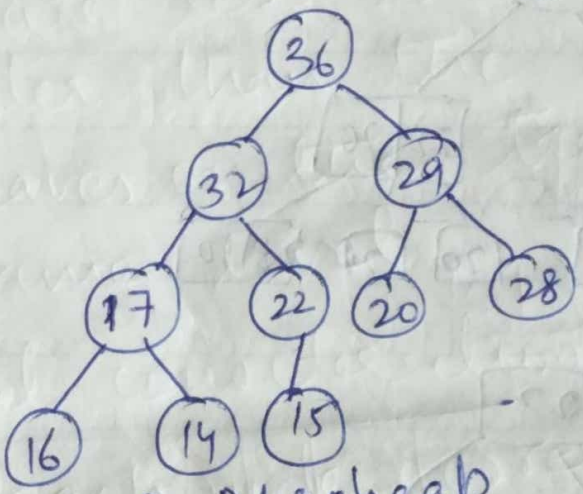


Fig: Maxheap

Inserting into a Heap

Suppose H is a heap with N elements & suppose an ITEM of info is given. We insert ITEM into the heap H as follows:

(1) First adjoin ITEM at the end of H so that H is still a complete tree, but not necessary heap.

(2) Then let ITEM rise to its appropriate place in H so that H is finally a heap.

after 3 page

Date .....
Page .....

Inserting into a Heap

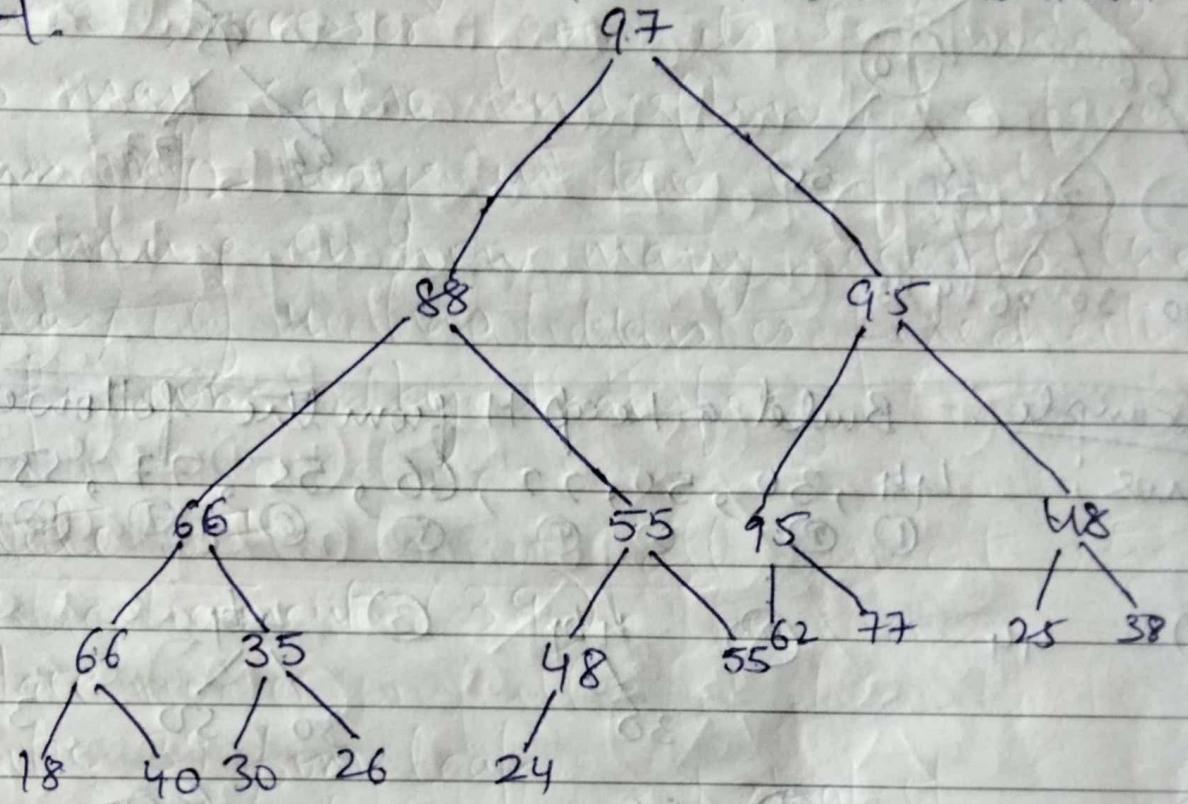
Efficiency of B-trees

Example

TREE

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
97	88	95	66	55	95	48	66	35	48	55	62	77	25	38	18	40	30	26	24

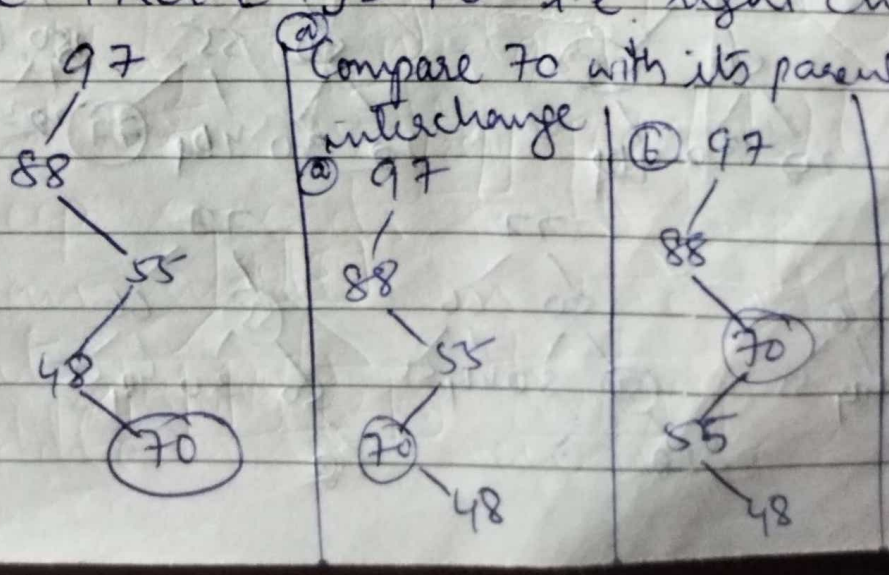
Given H.

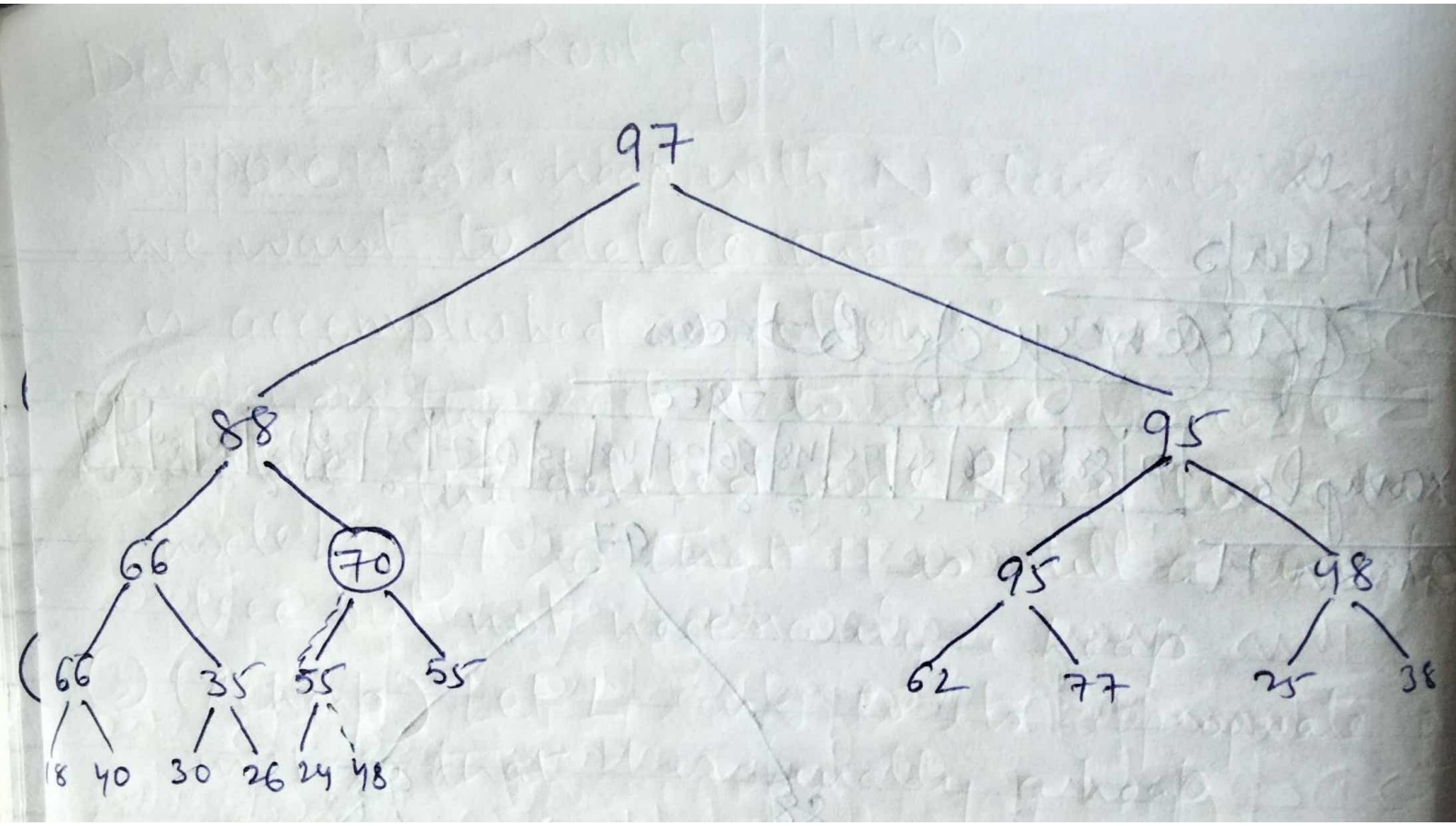


Heap

Add 70 to H

Adjoin 70 as the next element in the complete tree i.e TREE[21] = 70 i.e right child of TREE[10] = 48




97
88
95
66
70
95
48
66
35
55
55
62
77
25
38
18
40
30
26
24
48

Date .....
Page .....

Procedure 1° INSHEAP (TREE, N, ITEM)

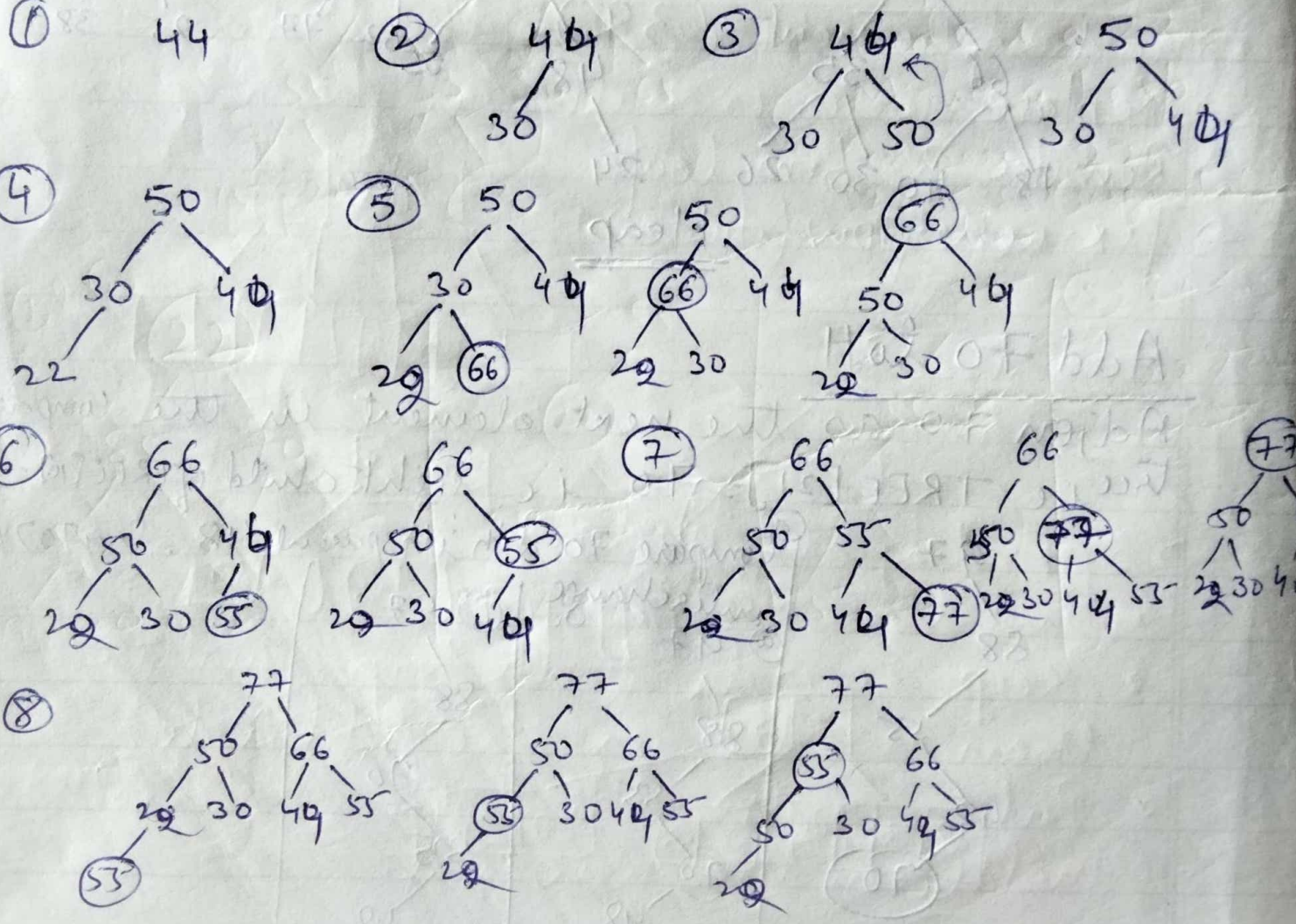
N → no. of elements in array.
TREE → array
PTR → location of ITEM as it rises in tree.
PAR → location of the parent of ITEM

1 [Add new node to H & initialize PTR]
Set N := N+1 & PTR := N.
2 [Find location to insert
Repeat steps 3 to 6 while PTR < 1
3. Set PAR := ⌊PTR/2⌋
4. If ITEM ≤ TREE [PAR] then
Set TREE [PTR] := ITEM & Return
[End of if structure]
5. Set TREE [PTR] := TREE [PAR]
6. Set PTR := PAR
[End of step 2 loop]
7. Set TREE [1] := ITEM
8 Return

Call INSHEAP (A, J, A[J+1])
for J = 1, 2, - - -, N-1

Example :- Build a heap H from the following list of nos :- 44, 30, 50, 22, 66, 55, 77, 55

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

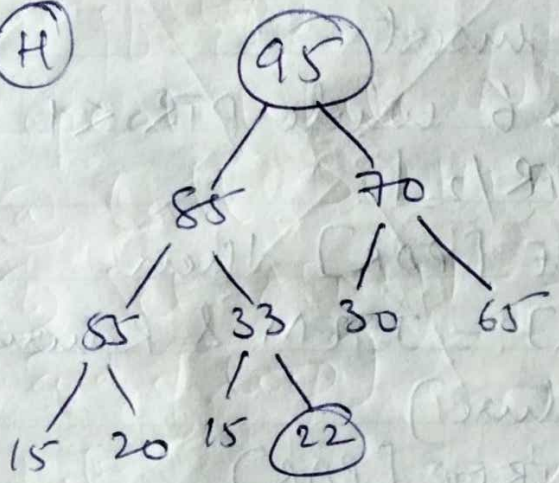


Deleting the Root of a Heap

Suppose H is a heap with N elements & suppose we want to delete the root R of H. This is accomplished as follows:

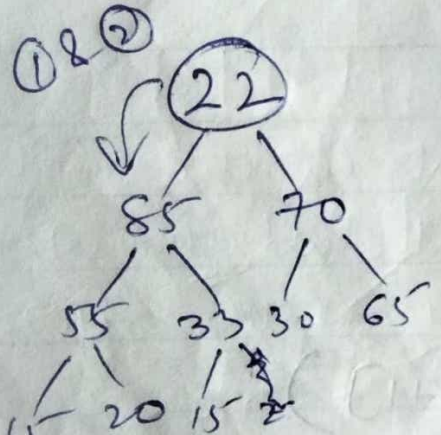
(1) Assign the root R to some variable ITEM
(2) Replace the deleted node R by the last node L of H so that H is still a complete tree but not necessarily a heap.
(3) (Reheap) Let L sink to its appropriate place in H so that H is finally a heap.

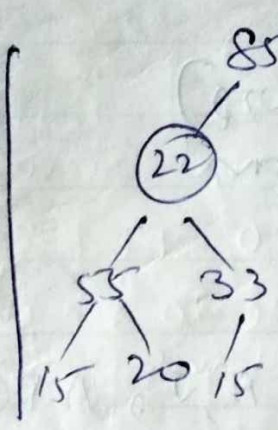
Example
Given (H)

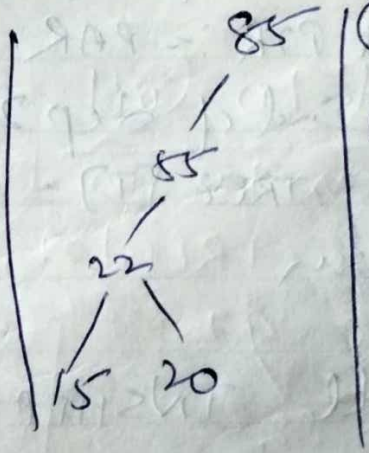


Delete 95

(1) Delete 95
(2) Replace 95 by last node i.e 22.
(3) Make a heap
(4) Compare 22 with 85 & 70
22 < 85 (large child)
Interchange







(5) 22 < 55
(6) 22 > 20, 15

Date .....
Page .....

Procedure 2 : DELHEAP (TREE, N, ITEM)

TREE → array of size N
LAST → saves last node

1. Set ITEM := TREE[1]
2. Set LAST := TREE[N] & N := N - 1
3. Set PTR := 1, LEFT := 2 & RIGHT := 3
4. Repeat Steps 5 to 7 while RIGHT ≤ N :
5. If LAST ≥ TREE [LEFT] & LAST ≥ TREE [RIGHT] then
   Set TREE [PTR] := LAST & Return
   [End of If structure]
6. If TREE [RIGHT] ≤ TREE [LEFT] then :
   Set TREE [PTR] := TREE [LEFT] & PTR := LEFT
   Else
   Set TREE [PTR] := TREE [RIGHT] & PTR := RIGHT
   [End of If structure]
7. Set LEFT := 2 * PTR & RIGHT := LEFT + 1
   [End of Step 4 loop]
8. If LEFT = N & if LAST < TREE [LEFT] then
   Set PTR := LEFT
9. Set TREE [PTR] := LAST
10. Return

Heap Sort

Phase A: Build a heap H out of the elements of A.
Phase B: Repeatedly delete the root element of H.
Since the root H always contains the largest node in H, Phase B deletes the elements of A in decreasing order

Algo HEAPSORT (A, N)

1. [Build a heap H]
   Repeat for J = 1 to N-1
      Call INSHEAP(A, J, A[J+1])
   [End of loop]
2. Repeat while N > 1
   (a) Call DELHEAP(A, N, ITEM)
   (b) Set A[N+1] := ITEM
   [End of loop]
3. Exit