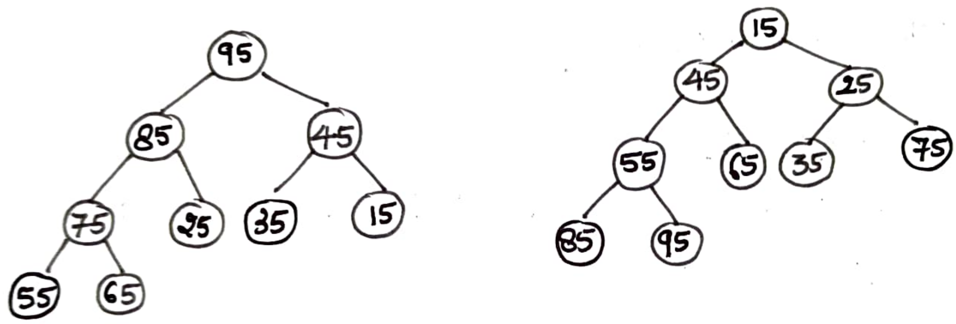
HEAP TREES

Suppose H is a complete Binary Tree. It will be termed as heap tree, if it full fill following properties :-

(i) For each node N in H, the value of N is greater than or equal to the value of each of the children of N.

(ii) Or, in other words, N has a value which is greater than or equal to the value of every successor of N.

Such a heap tree is k/a MAX HEAP.



(a) MAX HEAP

it contain Largest element at the ROOT.

(b) MIN HEAP

it contain Smallest Element at the ROOT.

Representation of a Heap Tree

→ A heap can be rep. using a LINKED STRUCTURE. But a single array rep. has certain advantages for a heap tree over its linked represent.

A heap tree is a COMPLETE BINARY TREE. → thus THERE IS NO WASTAGE OF ARRAY SPACE B/W THE TWO NON-NULL ENTRIES; IF THERE ARE NULL ENTRIES, THEY ARE ONLY AT THE TAIL OF THE ARRAY.

→ Another Advantage → is that we do not have to maintain any link of descendants (child); here, these are automatically implied.

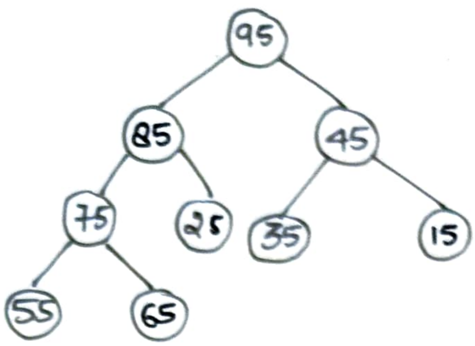
→ Major advantages with this representation is that from a node we can go in both direction ie, towards its ANCESTOR & SUCCESSORS AS WELL. This although possible in a linked structure is a matter of maintenance of an extra link field.

OPERATIONS ON A HEAP TREE

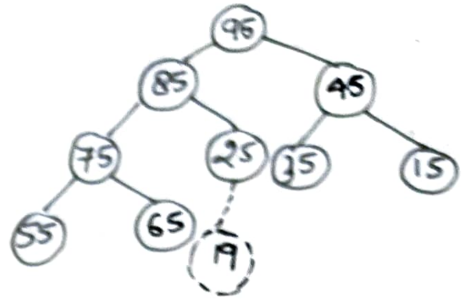
(a) Insertion into a heap tree

- used to insert a node into an existing heap tree satisfying the properties of a heap tree.

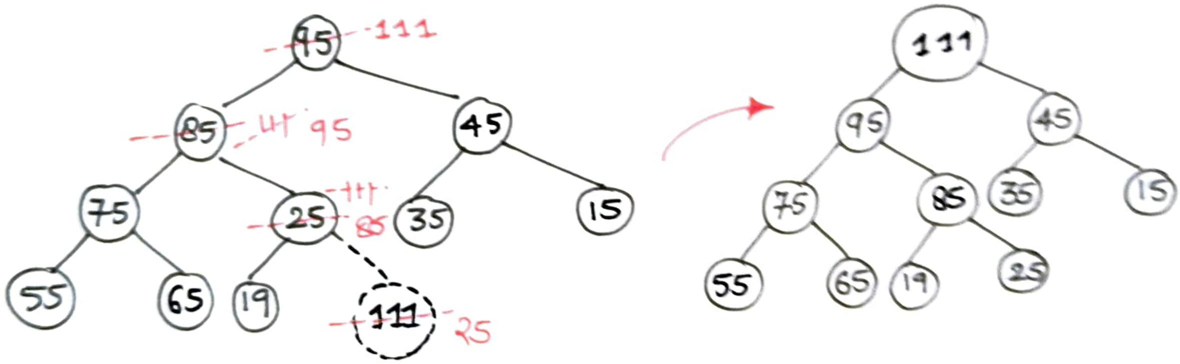
- PRINCIPLE OF INSERTION :-
  - We have to adjoin the data in the complete Binary Tree.
  - Next, compare data with its parent; if the value is greater than that at parent then we interchange the values. This can continue b/w two nodes on paths from the newly inserted node to the root node till we get a parent whose value is greater than its child. or we reach till the root.



(a) MAX HEAP



(b) Inclusion of 19 in the fashion of complete Binary Tree & it satisfy the property of heap.



(c) When 111 is inserted into the heap tree

ALGORITHM INSERTMAXHEAP

Input: ITEM, the data to be Inserted; N, the strength of nodes.

Output: ITEM, is inserted into the heap tree.

Data Structure: Array A [1.... SIZE] stores the heap tree; N being the no of nodes in the tree.

Steps:-

1. If (N ≥ SIZE) then
2. PRINT " Heap Tree is saterated: Insertion is void".
3. EXIT.
4. ELSE
5. N=N+1
6. A[N] = ITEM
7. i=N
8. p= i div 2.
9. While (p>0) and (A[p] < A[i]) do
10. temp= A[i]
11. A[i] = A[p]
12. A[p] = temp
13. i=p
14. p=p div 2
15. EndWhile
16. EndIf
17. Stop.

Deletion of a node from a heap tree

- Any node can be deleted from a heap tree.
- Deleting the Root node has some special importance.
- This principle can be stated as follows:—

  - Read the Root Node into a Temporary storage say, ITEM.
  - Replace the Root node by the last node in the heap tree. Then reheap the tree as stated below:-

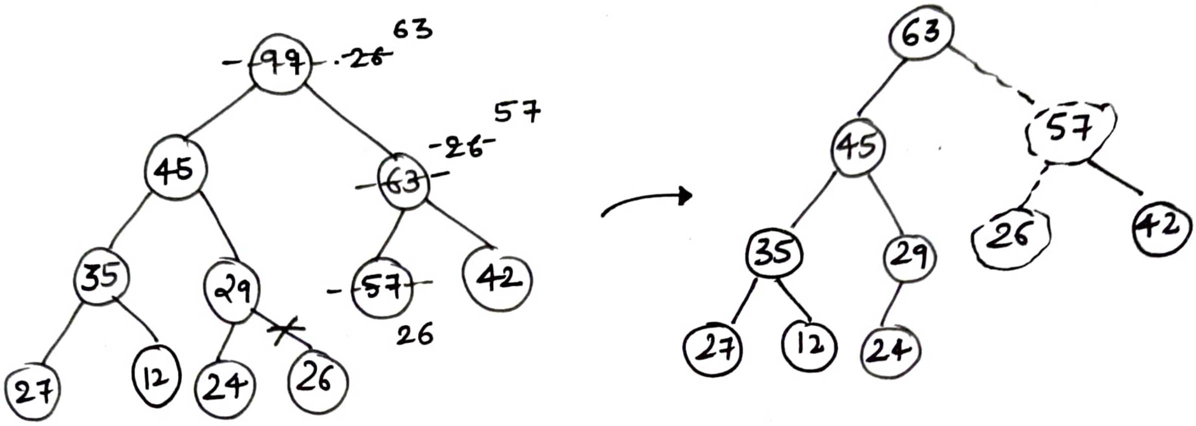
    ↳ Let the newly modified root node be the current node. Compare its values with the values of its two children.
    Let X be the child whose value is the largest.
    Interchange the value of X with the value of the current node.

    ↳ Make X as the current node.

    ↳ Continue reheap if the current node is not an empty node.

In this figure, the Root node is 99. The last node is 26, and it is level 3. So 99, is replaced by 26 & this node with data 26 is removed from the tree.

Next 26 at the Root Node is compared with its two children 45 and 63. As 63 is greater, so they are interchanged. Now, 26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged.



deleting the node with 99

ALGORITHM DeleteMaxHeap

Steps

1. If (N=0) then
2. Print " Heap tree is exhausted : Deletion is not possible"
3. Exit
4. EndIf
5. ITEM = A[1] ⟶ Value at the root node
6. A[1] = A[N] ⟶ Replace the value at the Root node by its counterpart at the last node on the last level.
7. N = N-1 ⟶ size of heap reduced by 1.
8. flag = FALSE , i=1
9. While (flag = FALSE) and (i<N) do ⟶ Rebuild the heap tree
10. lchild = 2*i , rchild = 2*i+1 ⟶ // Address of the left and right children of the current node.
11. if (lchild ≤ N) then
12. x = A[lchild]
13. Else
14. x = -∞
15. EndIf

```
16. If (rchild ≤ N) then
17.     y = A[rchild]
18. Else
19.     y = -∞
20. EndIf
21. If (A[i] > x) and (A[i] > y) then      // If the parent is larger than its child
22.         flag = TRUE                     // Reheap is over
23.   Else
24.       If (x > y) and (A[i] < x)   // If the left child is larger than right child
25.           swap (A[i], A[lchild]) // Interchange the data b/w parent and left child
26.           i = lchild  →   // left child becomes the current node
27.         Else
28.           if (y > x) and (A[i] < y)  // If the right child is larger than the left
                                            child
29.             swap(A[i], A[rchild]) // Interchange the data b/w the parent &
                                            the right child.
30.             i = rchild  →  // Right child becomes the current node.
31.           EndIf
32.         EndIf
33.   EndIf
34. Endwhile
35. Stop.
```

APPLICATION OF HEAP TREES

There are two known main application of heap trees

(a). Sorting (b) Priority Queue.

SORTING USING A HEAP TREE.

- Any kind of data can be sorted either in ascending order or in descending order using a heap tree. This actually consists of the following steps:→

Step 1: Build a heap tree with the given set of data.

Step 2: (a) Delete the Root Node from the heap.

(b) Rebuild the heap after the deletion.

(c) Place the deleted node in the output.

Step 3: Continue Step 2 until the heap tree is empty.

→ The heap sort uses heap tree as an underlying data structure to sort an array of elements.

→ The Heap Sort, unlike the TREE SORT, is an INPLACE SORTING METHOD, because it doesn't require any extra storage space other than the input storage list.

We assume that the heap tree satisfies the property of max heap unless otherwise stated. The Basic steps in the heap sort are listed below :→

1. CREATE HEAP : Create the INITIAL HEAP TREE n elements stored in the array A.

2. REMOVE MAX: Select the value in the root node. Swap the value (that is A[1]) with the value at the i^{th} location in A.

3. REBUILD HEAP: Rebuild the heap tree for elements A[1, 2, 3, ..., i-1].

CREATE HEAP

A

1	2	3	4	5	6	7	8	9	10
15	35	55	75	05	95	85	65	45	25

Create heap (Pass 1)

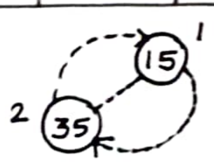
B

1	2	3	4	5	6	7	8	9	10
15									

(a) Initially, 15 is inserted into the empty heap.

A

15	35	55	75	05	95	85	65	45	25

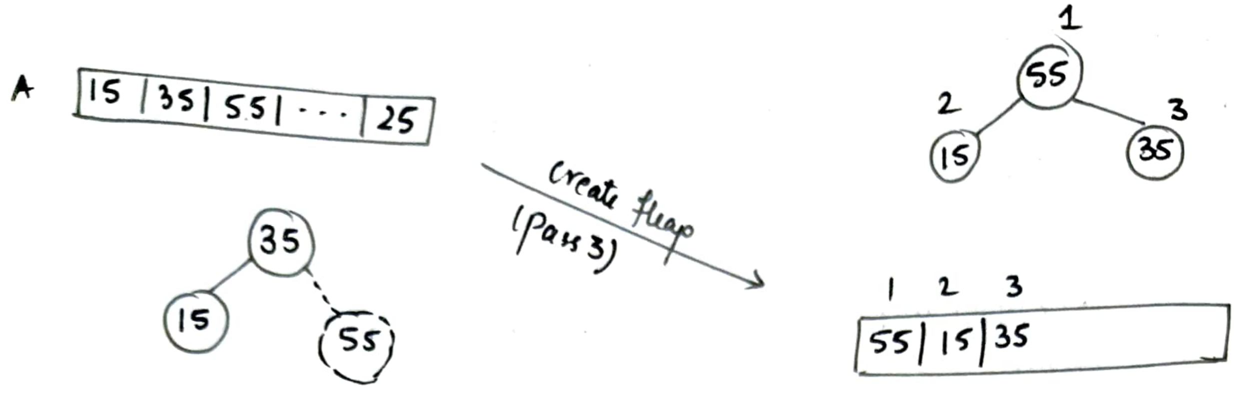


Create heap (Pass 2)

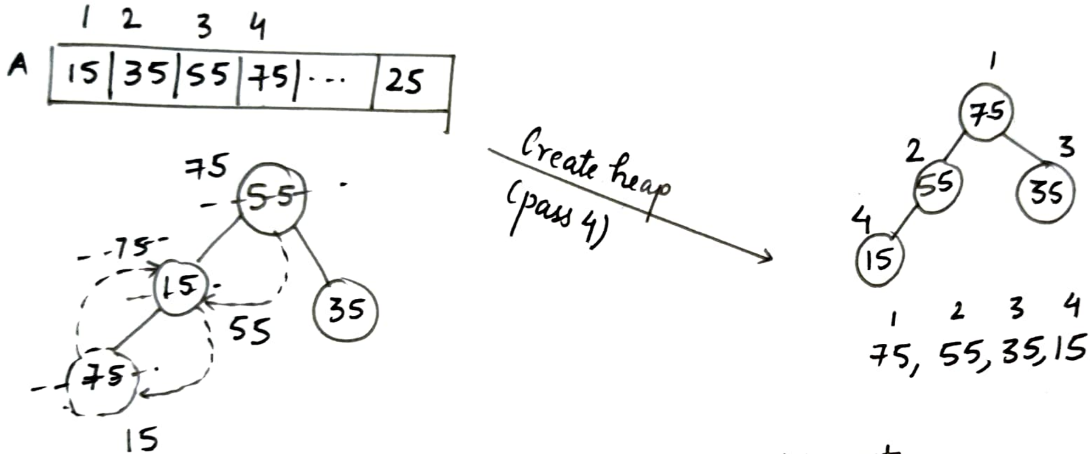
B

1	2	 10
35	15	

(b) 35 inserted.

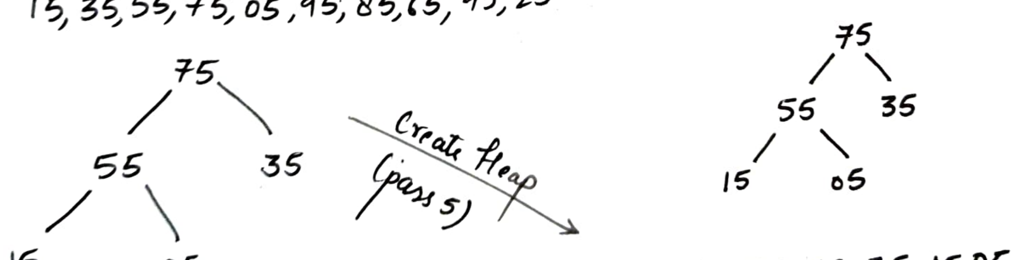


(c) 55 is inserted.



(d) 75 is inserted and moved to root

A :- 15, 35, 55, 75, 05, 95, 85, 65, 45, 25



B: 75, 55, 35, 15, 05

(e) 05 is inserted and remains there as its satifies the heap property.

A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 25

B: 95, 55, 75, 15, 05, 35

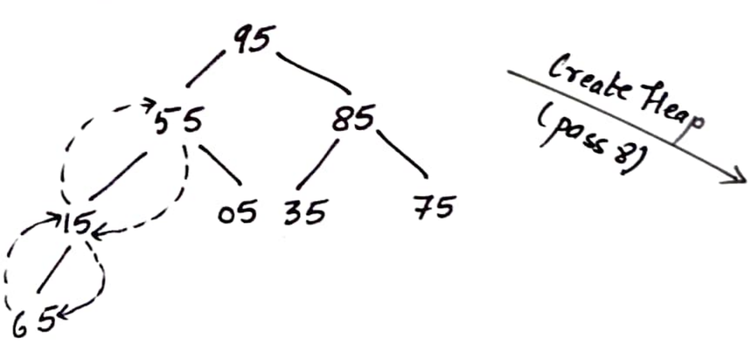
(f) 95 is inserted & moved to the Root.

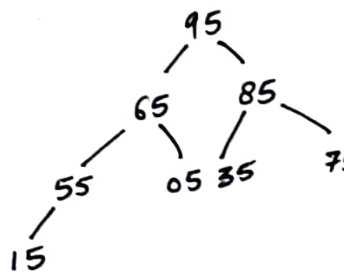
A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 25

B: 95, 55, 85, 15, 05, 35, 75

(g) 85 is inserted and moved to location 3

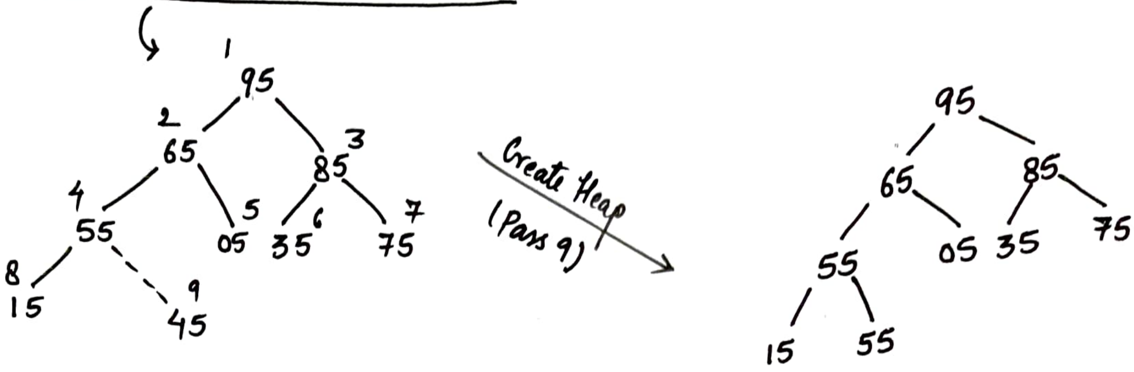
A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 25





B: 95, 65, 85, 55, 05, 35, 75, 15

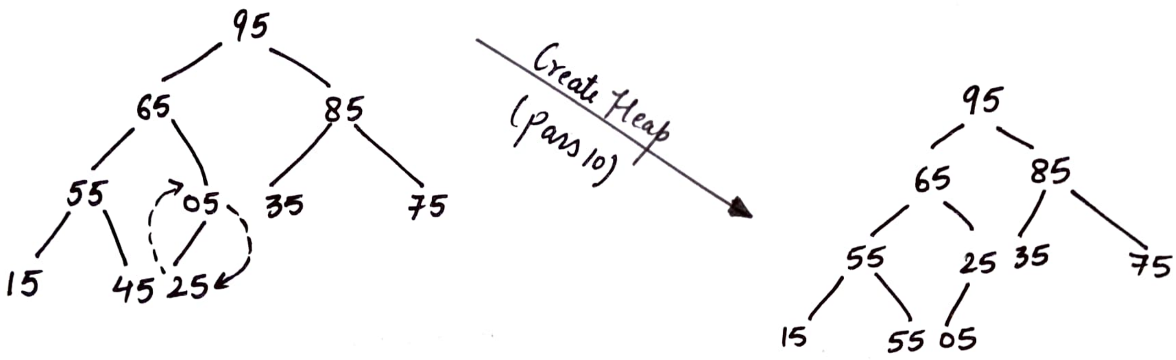
A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 42



B: 95, 65, 85, 55, 05, 35, 75, 15, 45

(i) 45 is inserted and remain there as it satisfies the heap property

A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 42



B: 95, 65, 85, 55, 25, 35, 75, 15, 45, 05

ALGORITHM CreateHeap

Input: A[1, 2, ... n] an array of n items.

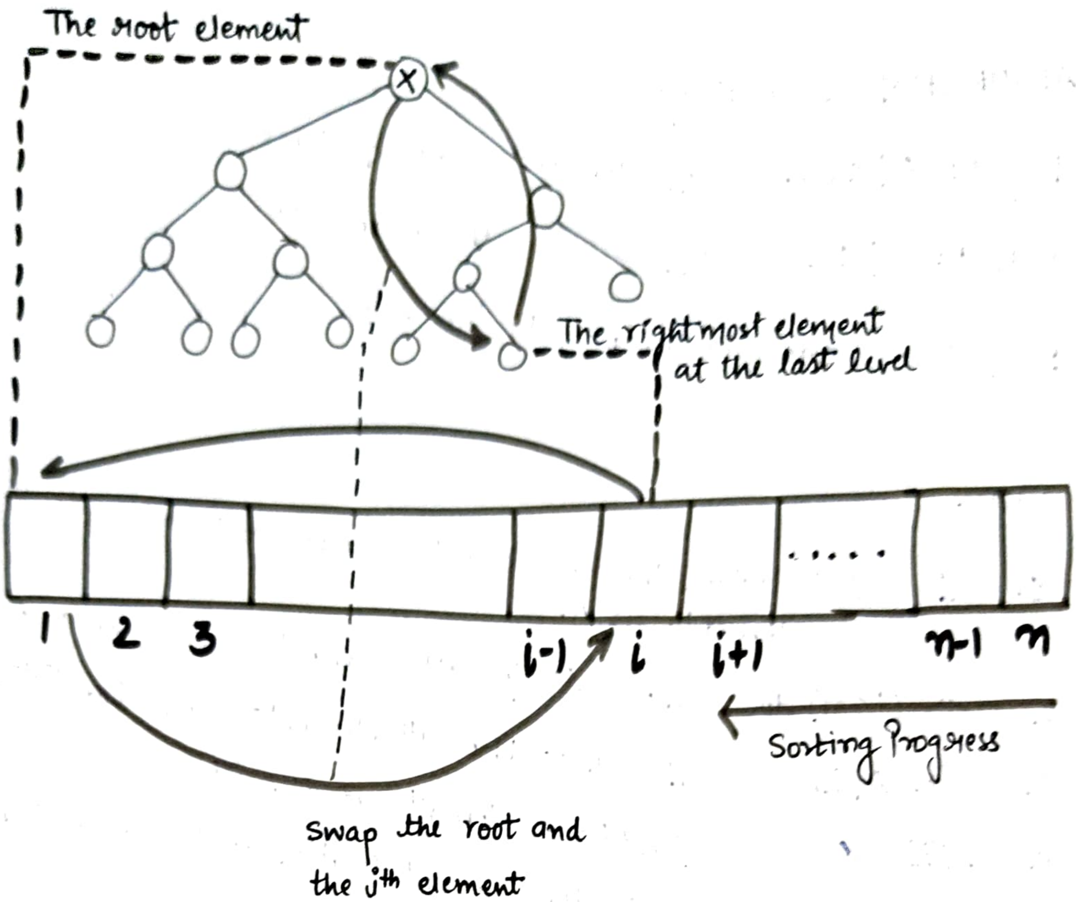
Output: B[1, 2, ... n] stores the heap tree.

Remarks: Creates the heap with the max heap property.

Steps:

```
1.  i=1                      // Initially, the heap tree B is empty and starts with the first
                                elements in A.
2.  While (i ≤ n) do         // Repeat for all elements in the array A
3.      x = A[i]             // Select the ith element from the list A
4.      B[i] = x             // Add the element at the ith
5.      j=i                  // j is the current location of the element in B.
6.      While (j>1) do       // Continue until the root is checked
7.          If B[j] > B[j/2] then    // It violates the heap (max) property
8.              temp = B[j]          // Swap the element
9.              B[j] = B[j/2]        //
10.             B[j/2] = temp
11.             j = j/2              // Go to the Parent node
12.         Else
13.             j=1                  // Satisfies the heap property, terminates this inner loop
14.         EndIf
15.     EndWhile
        i=i+1                // Select the next element from the input list.
16. EndWhile
17. Stop
```

Remove Max.



In i^{th} iteration the heap is confined within this part.

Algorithm Remove Max.

Input: $B[1,2,...n]$ an array of n items and the last element in the heap is at i.

Output: The first element and the i^{th} element get interchanged

Steps:

1. temp = B[i]   // Swap the element
2. B[i] = B[1]
3. B[1] = temp
4. Stop

Algorithm Rebuild Heap

Steps

1. If (i=1) then
2. Exit → no rebuild with single element in the list.
3. j=1 → // else start with the Root Node
4. flag = TRUE → // Rebuild is required.
5. while (flag = TRUE) do
6. left Child = 2*j, rightChild = 2*j+1

/* Check if the right child is within the range of heap or not */

// Note: If the right child is within the range then also left child.

7. If (rightChild ≤ i) then

8. /* Compare whether the left child or the right child will move to up or not */

9. If (B[j] ≤ B[leftChild]) AND B[leftChild] ≥ B[rightChild] then

// Parent & left child violate the heap property

10. Swap(B[j], B[leftChild]) // Swap the parent & the left child

11. j= left child // Move down to node at the next level.

12. Else

13. If (B[j] ≤ B[rightChild]) AND B[rightChild] ≥ B[leftChild] then

// Parent & the right child violate the heap property

14. Swap (B[j], B[right Child]) // Swap the parent & the right child

15. j= right Child // Move down to node at the next level

16. Else

17. flag= FALSE

18. EndIf

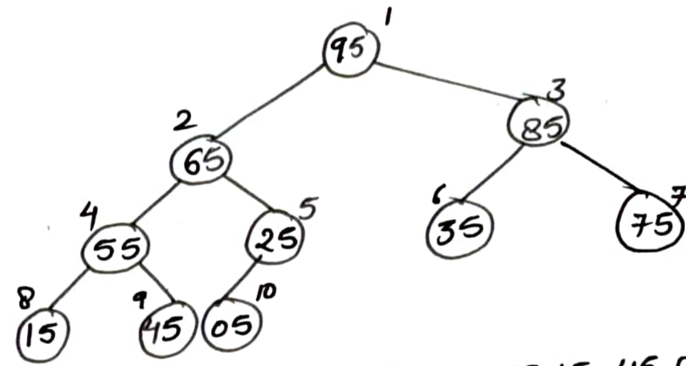
19. EndIf

```
Else
    If (leftChild ≤ i) then
        If (B[j] ≤ B[leftChild]) then
                        // Parent & left child violate the heap property
        Swap (B[j], B[leftChild]) // Swap the parent & the left child.
        j = leftChild      // Move down to node at the next level
        Else      // heap property is not violated.
            flag = FALSE
        EndIf
    EndIf
EndIf
EndWhile
Stop.
```

REBUILD HEAP.

A: 15, 35, 55, 75, 05, 95, 85, 65, 45, 25

create heap →

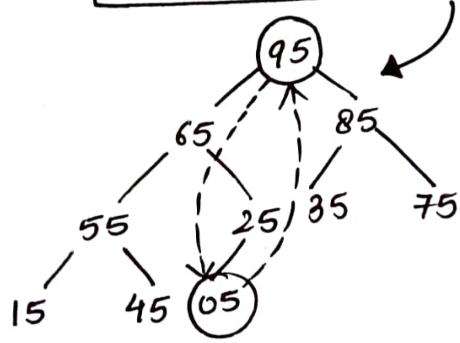


B: 95, 65, 85, 55, 25, 35, 75, 15, 45, 05

(a) Create heap

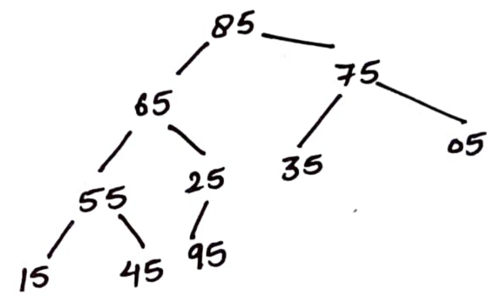
B: 95, 65, 85, 55, 25, 35, 75, 15, 45, 05

Rebuild heap



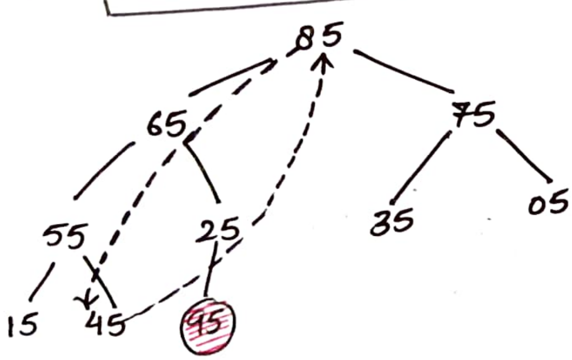
1. Remove Max
2. Rebuild heap

B: 85, 65, 75, 55, 25, 35, 05, 15, 45, <u>95</u>



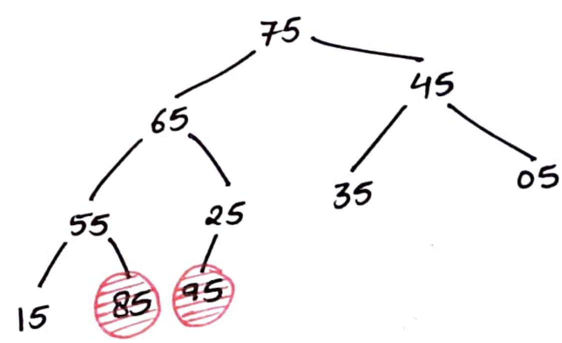
(b) Iteration 1 with $i = 10$

B: 85, 65, 75, 55, 25, 35, 05, 15, 45, $\boxed{95}$



1. Remove Max
2. Rebuild heap

B: 75, 65, 45, 55, 25, 35, 05, 15, <u>85, 95</u>



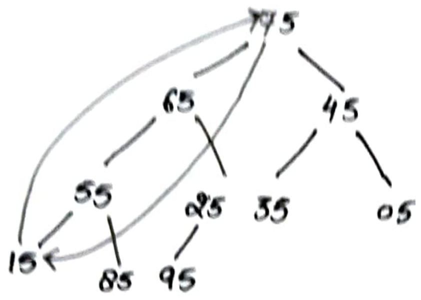
(c) Iteration 2 with $i = 9$.

1. Remove Max

B: 75, 65, 45, 55, 25, 35, 05, 15, 85, 95

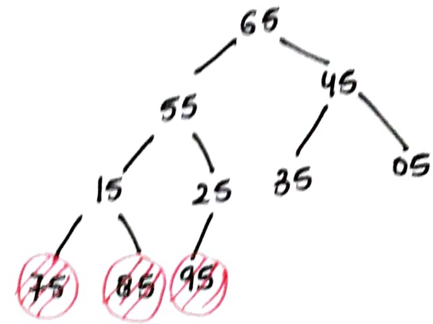
2. Rebuild Heap

B: 65, 55, 45, 15, 25, 35, 05, 75...



1. Remove Max

2. Rebuild heap



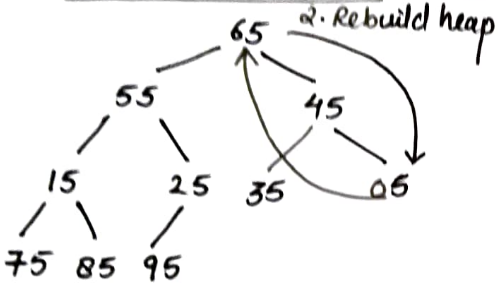
(d) Iteration 3 with i=8

1. Remove Max

B: 65, 55, 45, 15, 25, 35, 05, 75, 85, 95

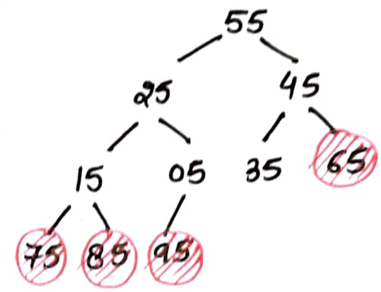
2. Rebuild heap

B: 55, 25, 45, 15, 05, 35, 65, 75, 85, 95



1. Remove Max

2. Rebuild Heap



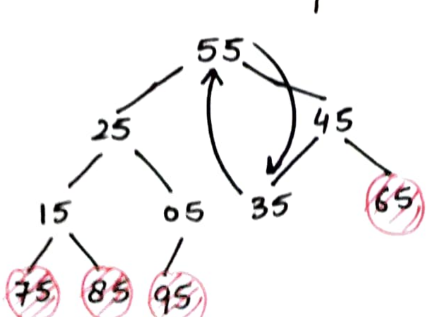
(e) Iteration 4 with i=7.

1. Remove Max

B: 55, 25, 45, 15, 05, 35...

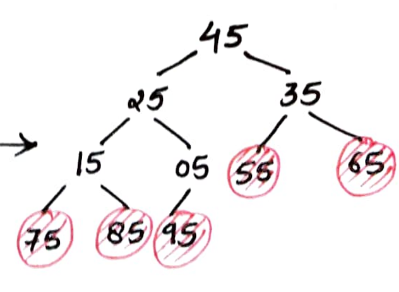
2. Rebuild Heap

B: 45, 25, 35, 15, 05, ...



1. Remove max

2. Rebuild Heap



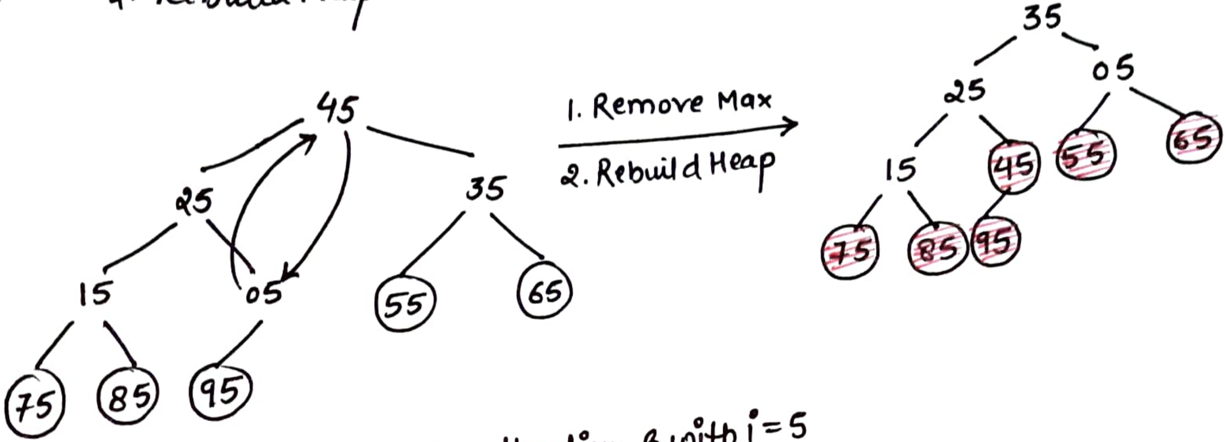
(f) Iteration 5 with i=6

1. Remove MAX

B: 45, 25, 35, 15, 05, 55, ..., 95

2. Rebuild Heap

B: 35, 25, 05, 15, 45, 55, 65, 75, 85, 95



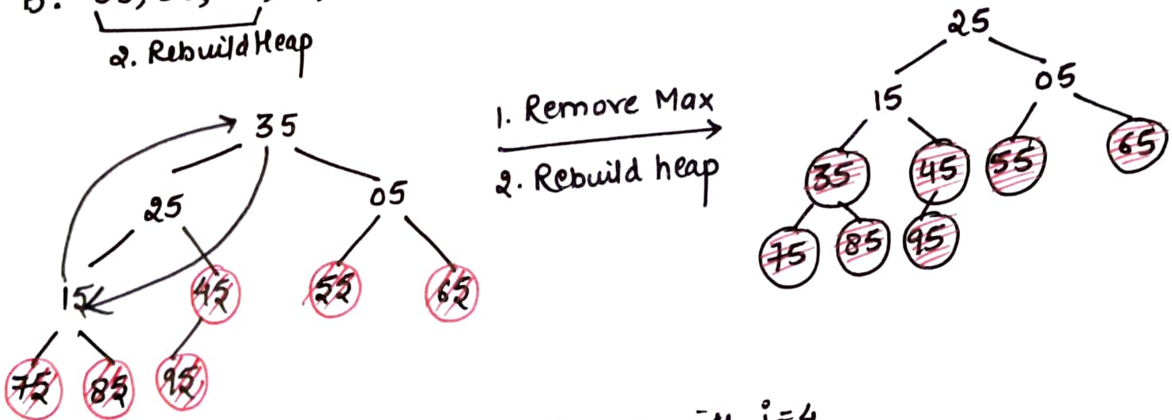
(g) Iteration 6 with i=5

1. Remove Max

B: 35, 25, 05, 15, 45, ..., 95

2. Rebuild Heap

B: 25, 15, 05, 35, 45, ..., 95

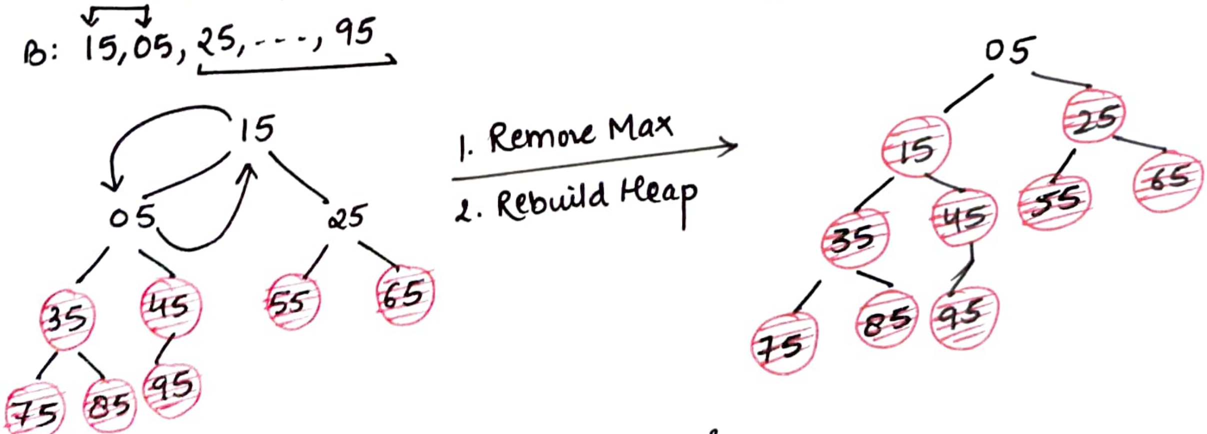


(h) Iteration 7 with i=4.

1. Remove Max

B: 15, 05, 25, ..., 95

B: 05, 15, 25, ... 95



(i) Iteration 9 with i=2.