Transform and Conquer Approach :

Transform and Conquer is an algorithmic strategy in which the problem is solved by applying transformations.

This method works as two-stage procedures.
→ First, in the transformation stage, the problem's instance is modified to be for one reason or another more amenable to solution. Then, in the second or conquering stage, it is solved.

How to apply these transformations ?

There are three major variations of this idea that differ by what we transform a given instance to

a) Instance Simplification : Transformation to a simpler or more convenient instance of the same problem.

Eg:- Consider the sequence of disks (shaded, plain and striped. ○ ● ● ● ⊕ ○ ● ○

To identify the uniqueness of disks, one can arrange the disks as shown below -

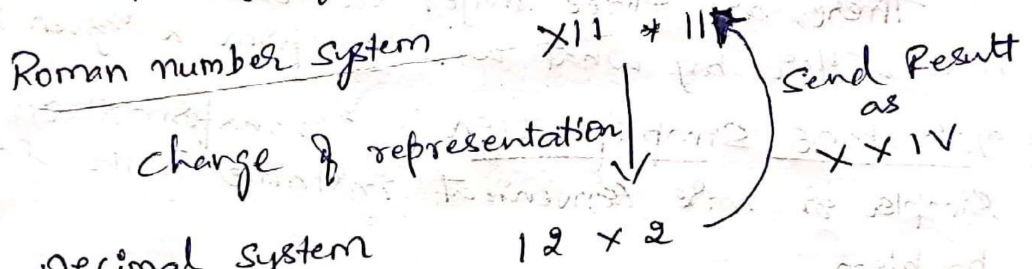
⊕ ● ● ● ● ○ ○ ○

Now, it is easier to evaluate the uniqueness of the disks, as only the adjacent disk needs to be checked. Thus, the striped disk can be found to be unique. It can be observed that problem complexity is not reduced, but the task of recognition is simplified.

b) Representation change : Transformation to a different representation of the same instance.

Eg. Multiplication of Roman numbers is difficult. Hence, the numbers are converted into another representation, that is, decimal Arabic number system, which is familiar and convenient to use. In this case, multiplication is carried out in the new representation. After multiplication, the result is converted back to the Roman number system.

The change of representation simplifies the complexity of problem solving.



Eg. Heap sort is also one good example for this. In heap sort, an array of numbers is transformed into a heap and the numbers are ~~stored~~ sorted using the heap property.

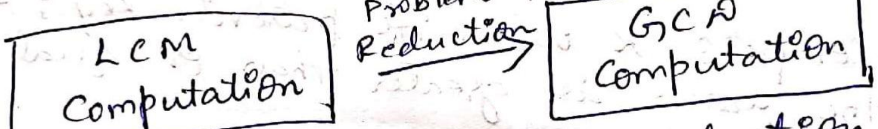
Eg. Polynomial Evaluation using the Horner's method.

Eg. Binary Exponentiation.

c) Problem reduction : Transformation to an instance of a different problem for which an algorithm is already available.

E.g. Computing the LCM from GCD.
Hypothetically, let us assume that an algorithm exists only for GCD. If so, then the problem of LCM can be converted to the problem of GCD using the following equation :

$$LCM(m,n) = \frac{m \times n}{GCD(m,n)}$$



The success of problem reduction lies in the identification of a way to reduce an unfamiliar problem into a problem of a familiar domain.

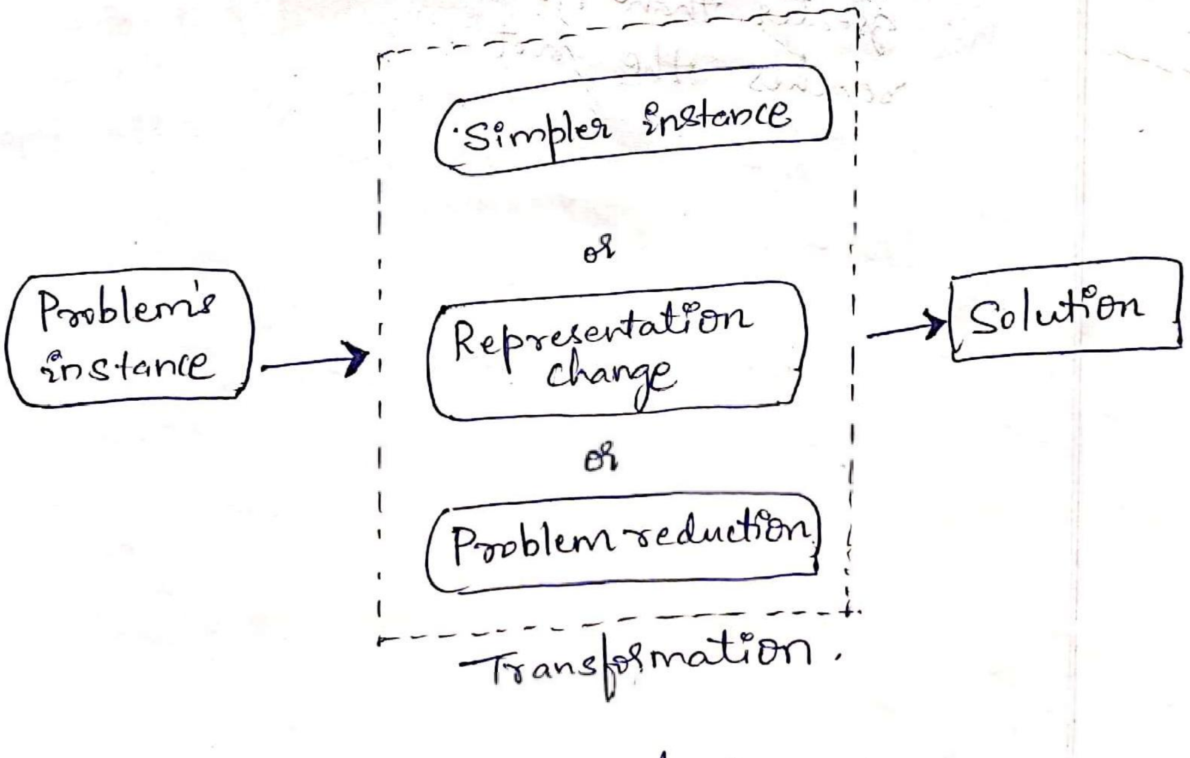


Fig. Transform and Conquer.

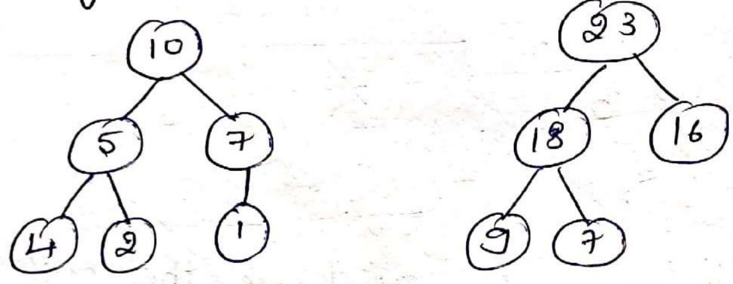
Transform and Conquer approach:

Heaps and Heap sort

Heap is a complete binary tree or a almost complete binary tree in which every parent node be either greater or lesser than its child nodes.

The binary tree is essentially complete, i.e all its levels are full except possibly the last level where only right most leaves may be missing.

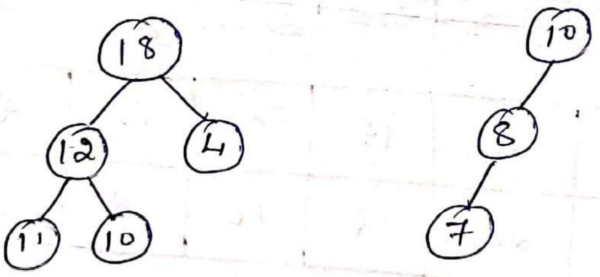
Eg



→ Heap can be min heap or max heap.

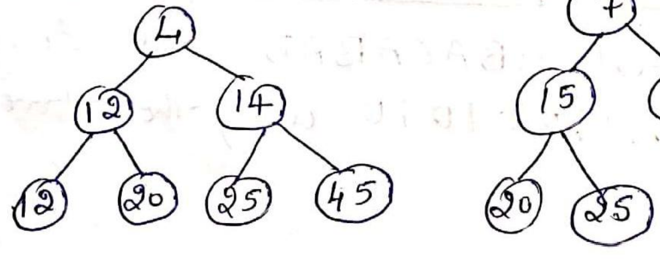
→ A Max heap is a tree in which value of each node is greater than or equal to the value of its children node.

Eg



→ A min heap is a tree in which value of each node is less than or equal to value of its children nodes.

Eg.

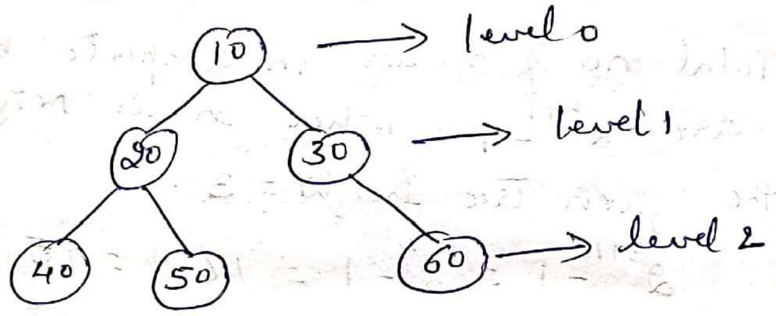


Scanned with CamScanner

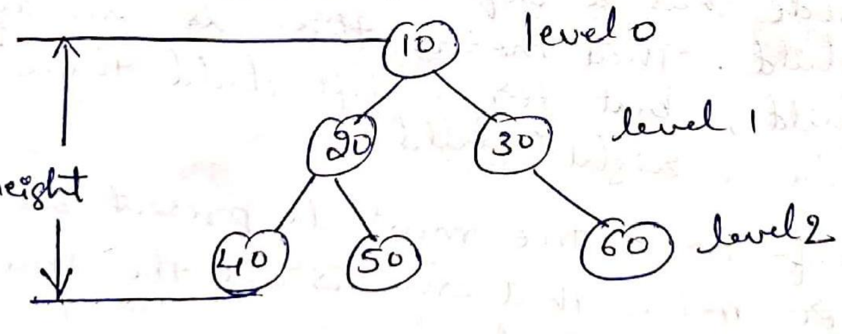
Parent being greater or lesser in heap is called Parental property. Thus heap has two important properties.

$$\begin{tikzcd}
 & \text{It should be a complete binary tree / Almost complete binary tree} \\
\text{Heap} \arrow[r] & {} \arrow[u] \arrow[d] \\
 & \text{It should satisfy parental property}
\end{tikzcd}$$

Level of binary tree :- The root of the tree is always at level 0. Any node is always at a level one more than its parent nodes level.



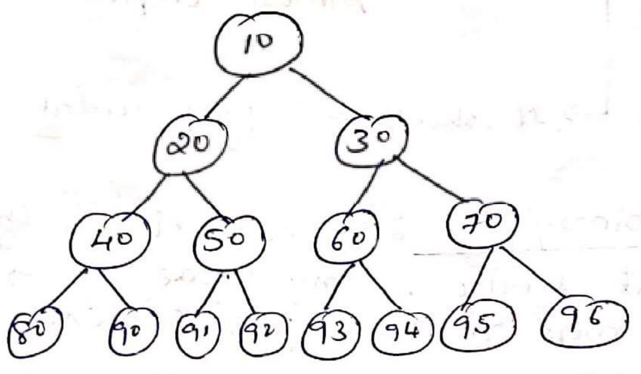
Height of the tree :- The maximum level is the height of the tree. The height of the tree is also called depth of the tree.



The max level of this tree is 2. Hence height of this tree is 2.

Complete binary tree : The complete binary tree is a binary tree in which all leaves are at the same depth or total number of nodes at each level i are 2^i.

Eg.



Note: Total no of nodes in complete binary tree are $2^{h+1} - 1$, where h is height of tree.

In the given tree height = 3.

$\therefore 2^{3+1} - 1 = 2^4 - 1 = 16 - 1 = 15$

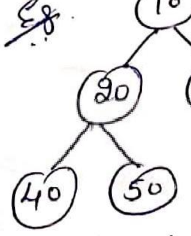
$\therefore$ Total 15 nodes in this complete binary tree.

Almost complete binary tree :- It is a tree in which -

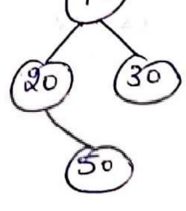
(i) Each node has a left child whenever it has a right child. That means there is always a left child, but for a left child there may not be a right child.

(ii) The leaf in a tree must be present at height h or h-1. That means all the leaves are on two adjacent levels.

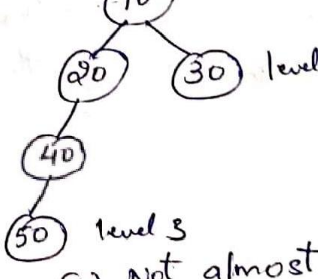
Eg.



(a) Almost complete binary tree



(b) Not almost binary tree ($\because$ 1st Property not satisfied)



(c) Not almost binary tree ($\because$ 2nd property not satisfied)

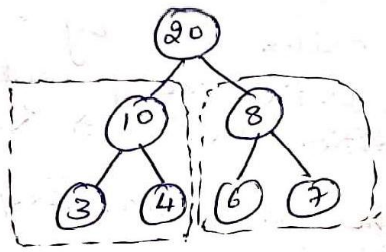
Properties of heap :

1) There should be either "complete binary tree" or "almost complete binary tree".

2) The root of a heap always contains its largest element.

3) Each subtree in a heap is also a heap.

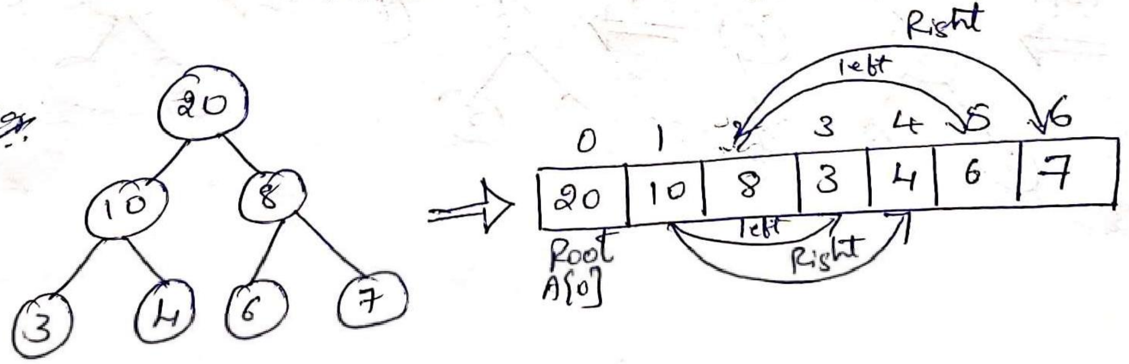
Eg.



4) Heap can be implemented as array by recoding its elements in the top-down and left to right fashion. i represents index

For array A[Size] —
- → Root node is stored at A[0].
- → left child is at 2i+1 position in array A
- → Right child is at 2i+2 position in array A.
- → Parent node is at $\left(\frac{i-1}{2}\right)$ position in Array A.

Eg.



Thus array can represent complete heap.

Construction of Heap : Two approaches are there : a.) Top down heap construction
b) Bottom up heap construction

a) Top down heap construction : This method constructs a heap by successive insertions of a new key into a previously constructed heap.

→ First, attach a new node with key K in it after the last leaf of the existing heap.

→ Then sift K up to its appropriate place in the new heap as follows -

* Compare K with its parent's key : If the latter is greater than or equal to K, stop.
* Otherwise, swap these two keys and compare K with its new parent.
* This swapping continues until K is not greater than its last parent or it reaches the root.

Bottom-up heap construction algorithm :

This algorithm initializes the essentially Complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows :

Starting with the last parental node, the algorithm checks whether the parental dominance holds for the key at this node. If it does not, the algorithm exchanges the node's key k with the larger key of its children and checks whether the parental dominance holds for k in ~~its~~ new position. This process continues until the parental dominance requirement for k is satisfied.

After completing the "heapification" of the subtree rooted at the current parental node, the algorithm proceeds to do the same for the node's immediate predecessor. The algorithm stops after this is done for the trees root.

```
ALGORITHM HeapBottomUp (H[1...n])
// Constructs a heap from the elements of a given
// array by the bottom-up algorithm
// Input : An array H[1..n] of orderable items
// output : A heap H[1...n]

  for i ← ⌊n/2⌋ downto 1 do
      K ← i ;
      v ← H[K]
      heap ← false
      while not heap and 2*K ≤ n do
          j ← 2*K
          if j < n   // there are two children
              if H[j] < H[j+1]
                  j ← j+1
          if v ≥ H[j]
              heap ← true
          else
              H[K] ← H[j] ;
              K ← j

      H[K] ← v
```

How To Construct a heap for a given list of keys?

There are two approaches for the Construction of heap:

(i) Top-down approach $O(n\log n)$

(ii) Bottomup approach $O(\log n)$

Note: Top-down approach is not economical interms of time complexity, so we are considering bottom up approach.

Scanned with CamScanner

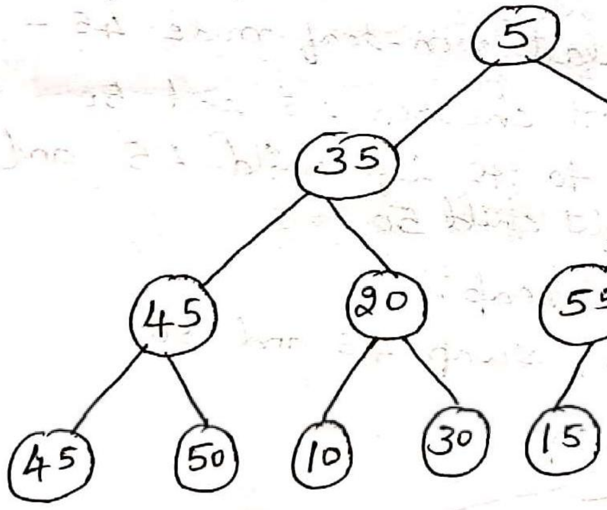
Create a Heap (Bottom-up Heap construction)

Create a heap for the elements 5, 35, 25, 45, 20, 55, 25, 45, 50, 10, 30 and 15 using bottom up approach.

Sol :- Even though tree structure is used, we use array to store the heap elements. The given elements are stored in an array as shown below:

0	1	2	3	4	5	6	7	8	9	10	11
5	35	25	45	20	55	25	45	50	10	30	15

The above array elements can be represented in the form of a tree as shown below:

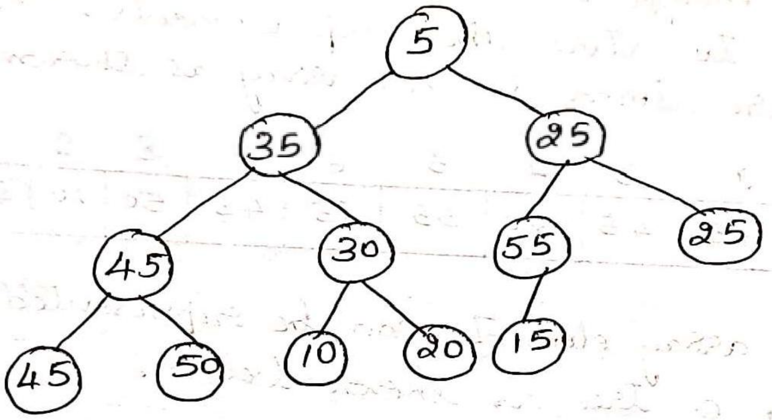


Step 1 :- Consider the last non-leaf node 55 - Compare 55 with its children. clearly, 55 is greater than its child 15.

Subtree satisfies the heap condition and no change is required.

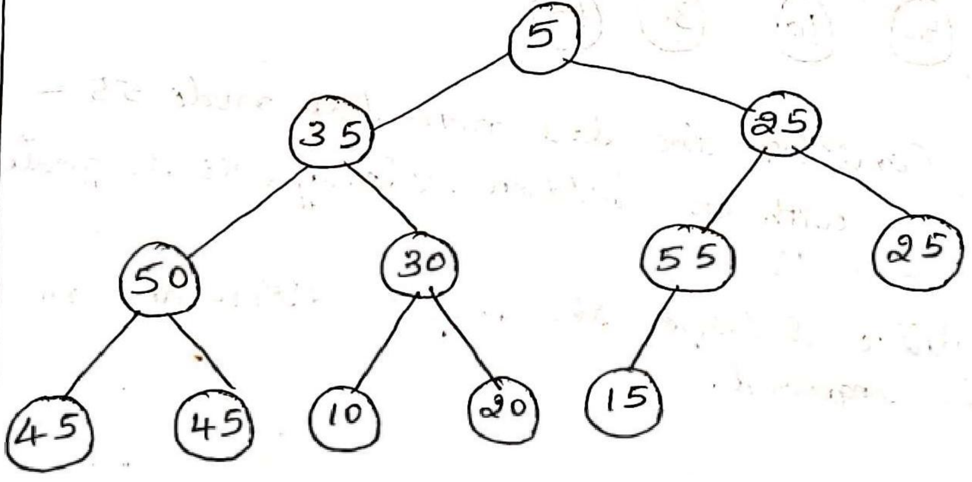
Step 2 : Consider the last non-leaf node 20 –

- Compare 20 with its children 10 and 30.
- Cleary 20 is greater than 10 and less than 30.
- Subtree is not a heap.
- we must swap 20 and 30 to make it a heap.



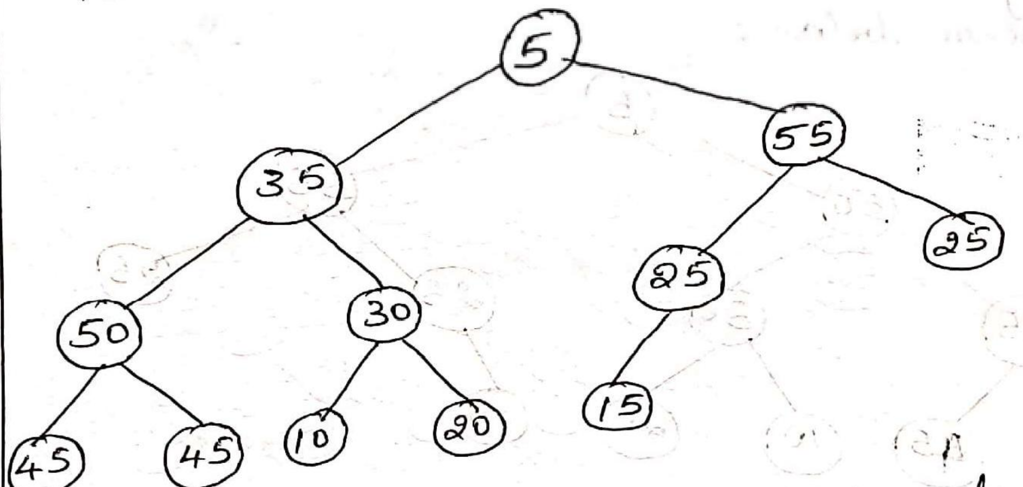
Step 3 :- Consider the last non-leaf node 45 –

- Compare 45 with its children 45 and 50.
- clearly 45 is equal to its left child 45 and less than to its right child 50.
- Subtree is not a heap.
- To make it a heap, swap 45 and 50.



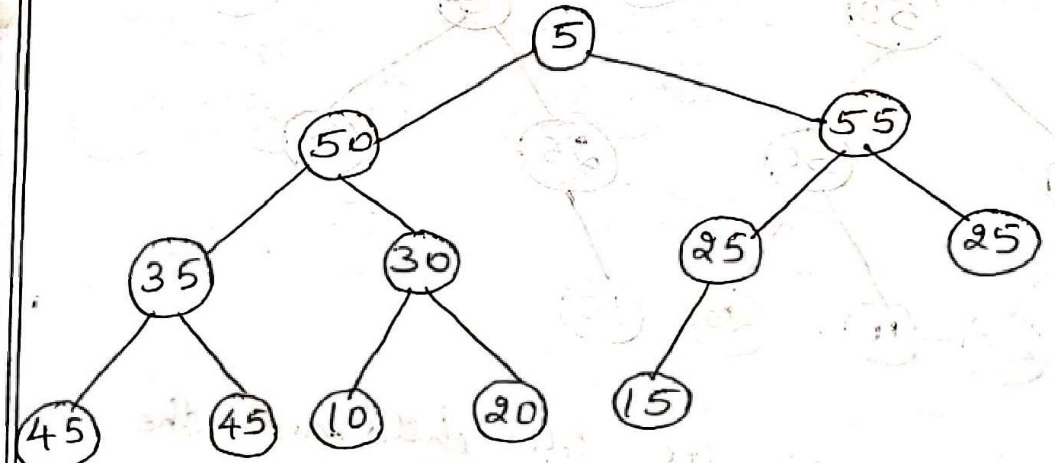
Step 4 :- Consider the last non leaf node 25 :

- Compare 25 with its children 55 and 25.
- clearly 25 is less than its left child 55.
- Subtree is not a heap.
- To make it a heap, swap 25 and 55.



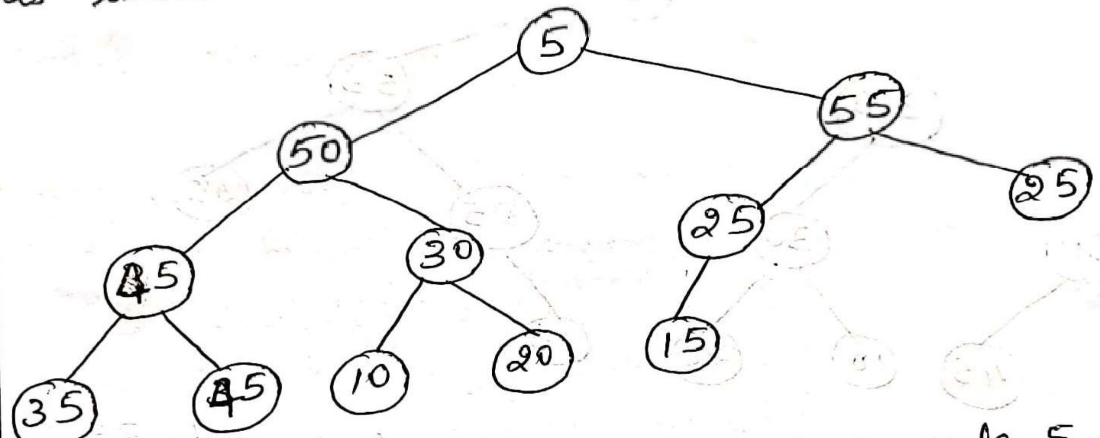
Step 5 :- Consider the last non leaf node 35 :
Consider the last non leaf node 35 with its children 50 and 30.

- Compare 35 with its children 50 and 30.
- clearly 35 is less than its left child 50.
- Subtree is not a heap.
- To make it a heap, swap 35 and 50

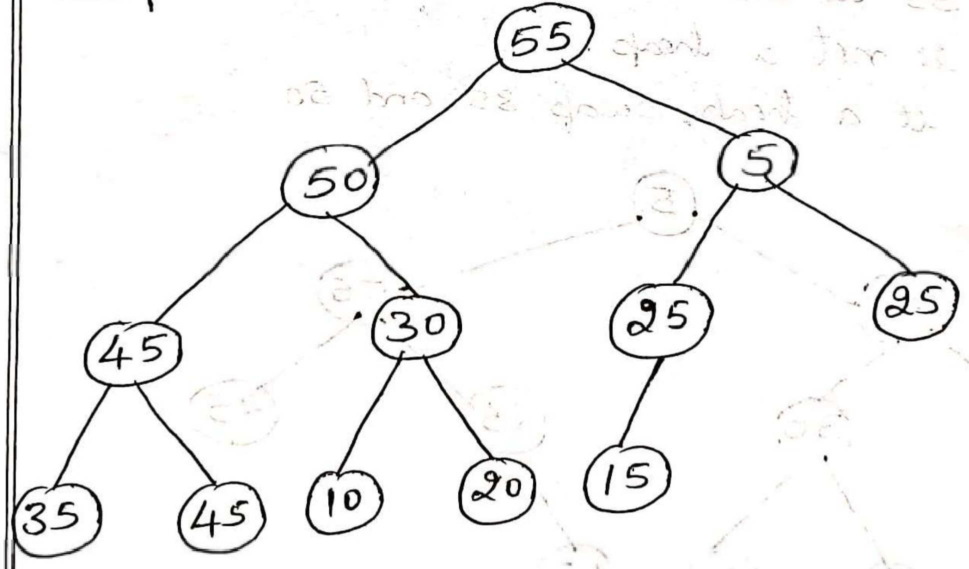


After swapping 35 and 50, check whether subtree identified by root node 35 forms a heap.

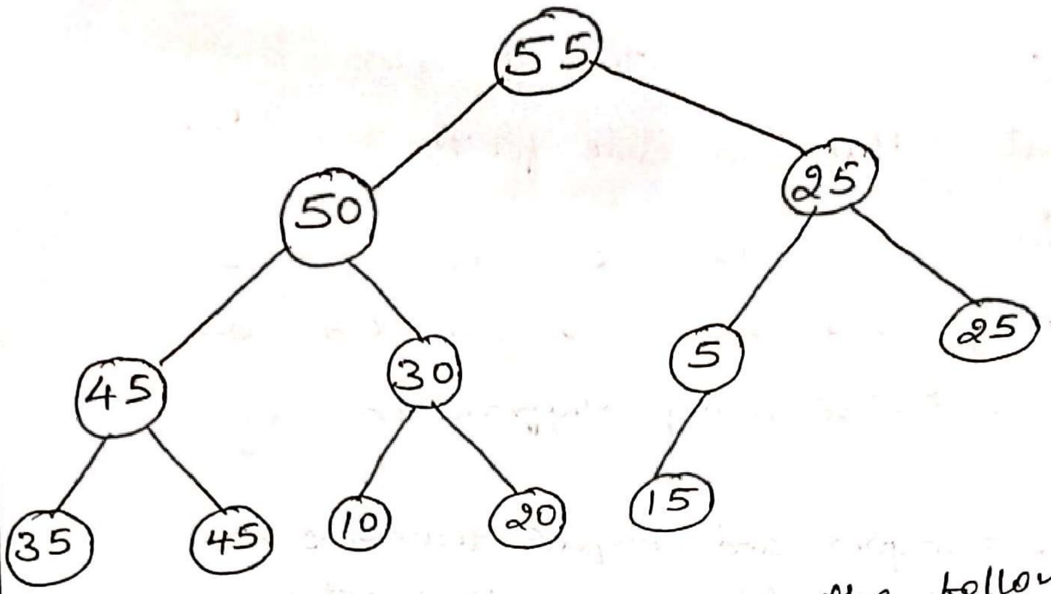
Comparing 35 with 45 and 45. clearly it is not a heap. So, 35 should be exchanged with largest of two children. Since left child and right child are same, we can exchange either with left child or right child. Let us exchange with left child and the resulting tree is shown below:



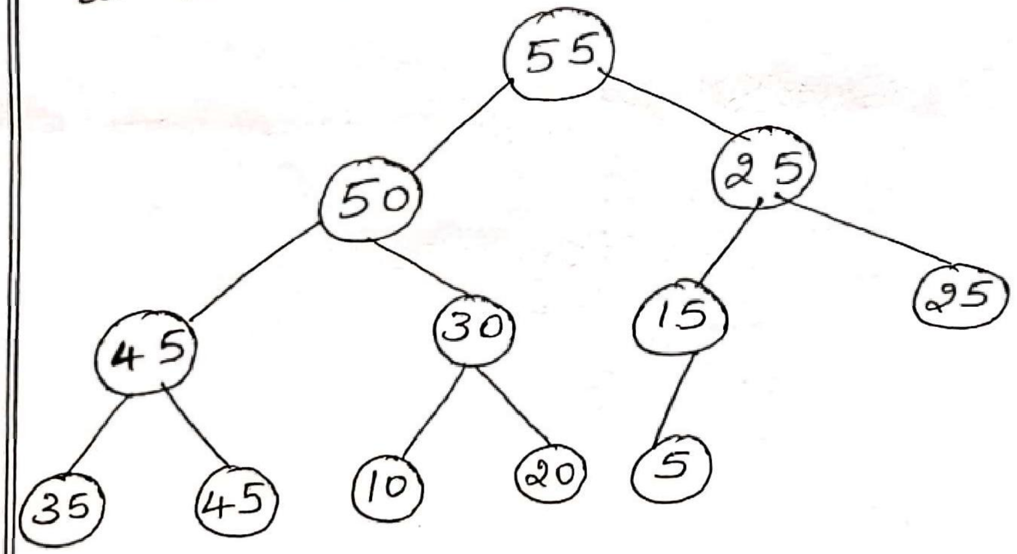
Step 6:- Consider the last non leaf node 5 - Swap 5 with largest of two children.



Swap 5 with its left child and the resulting tree is shown below:



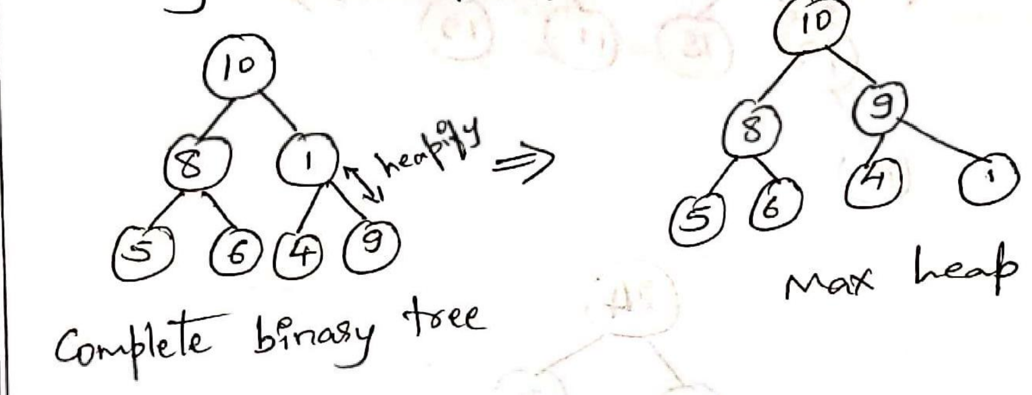
After swapping 5 and 15, the following tree is obtained.



Construct a heap for the list 10, 8, 1, 5, 6, 4, 9

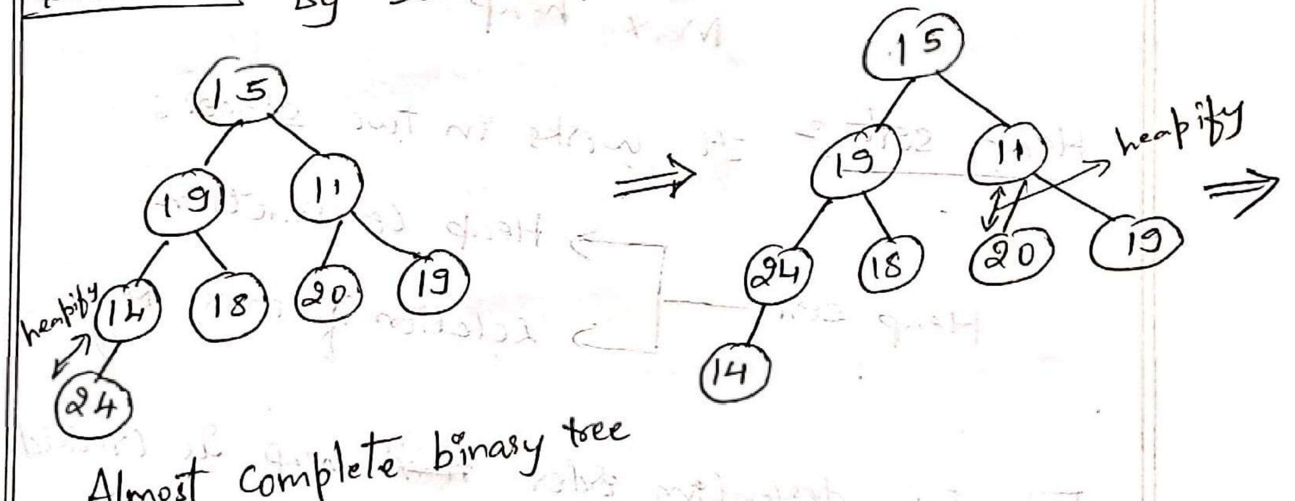
Solution :

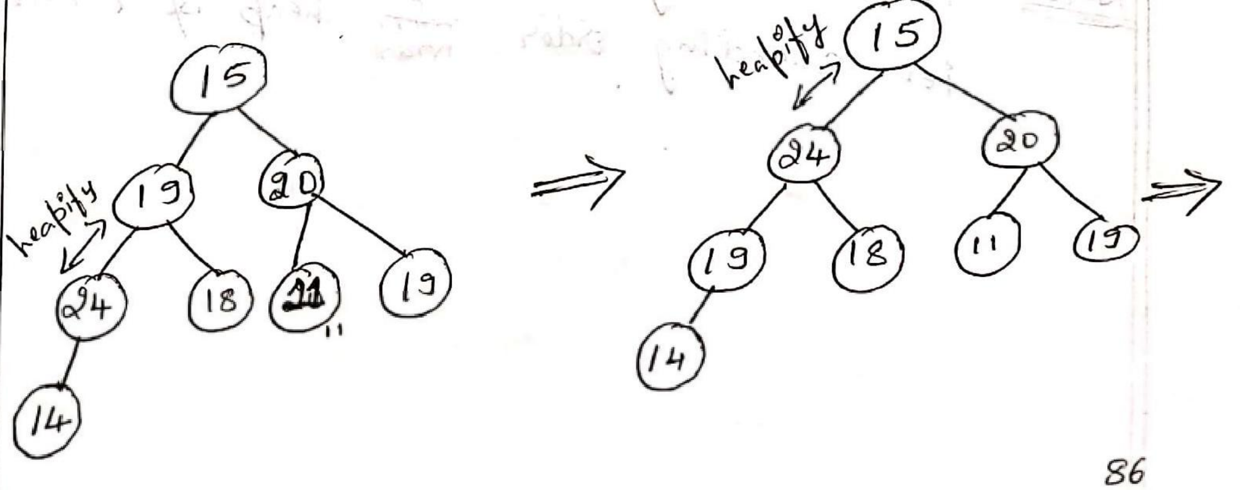
By bottom-up algorithm –

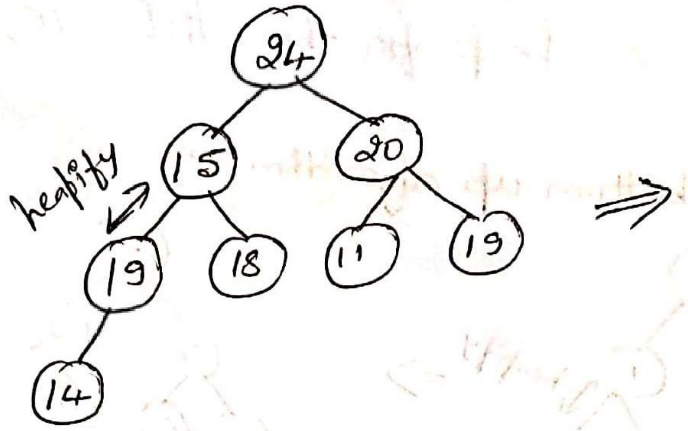


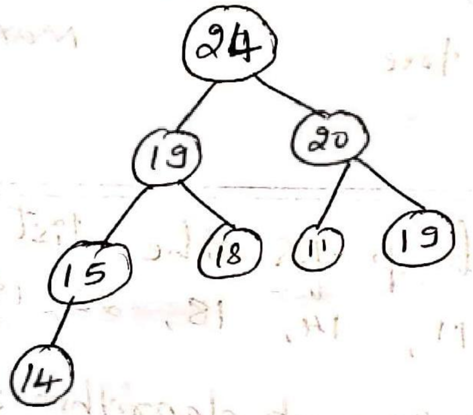
Construct a heap for the list –
15, 19, 11, 14, 18, 20, 19, 24

Solution :- By bottom-up algorithm :









Max heap.

Heap sort :- It works in two stages :

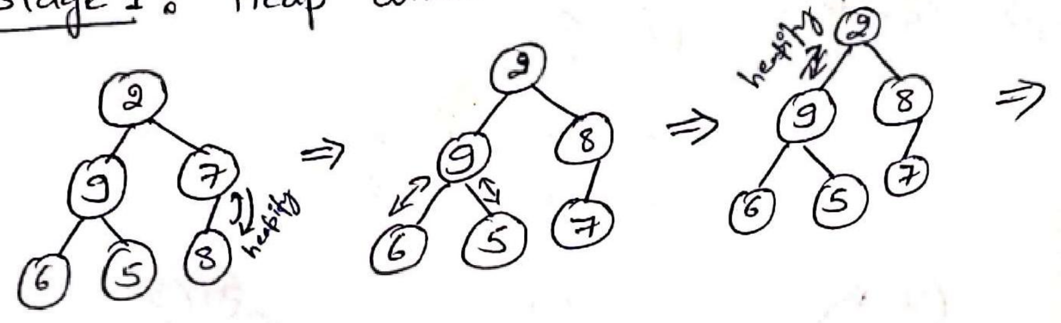
$$\text{Heap sort} \begin{cases} \to \text{Heap Construction} \\ \to \text{Deletion of max key} \end{cases}$$

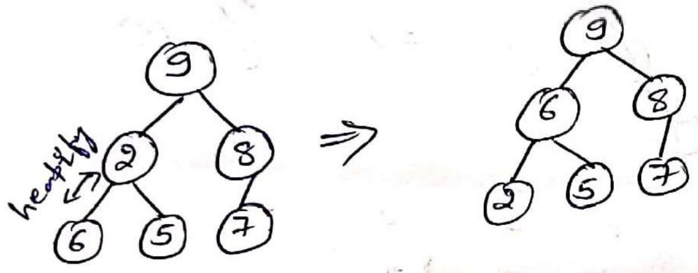
Note :- For descending order, Max heap is created.
For ascending order, Min heap is created.

Construct a heap for the list 2 9 7 6 5 8 Using bottom-up algorithm and sort them in order.

Sol:- Descending

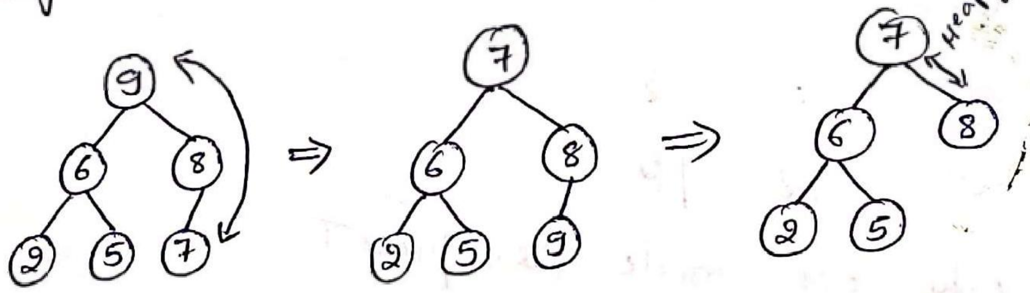
Stage 1: Heap construction



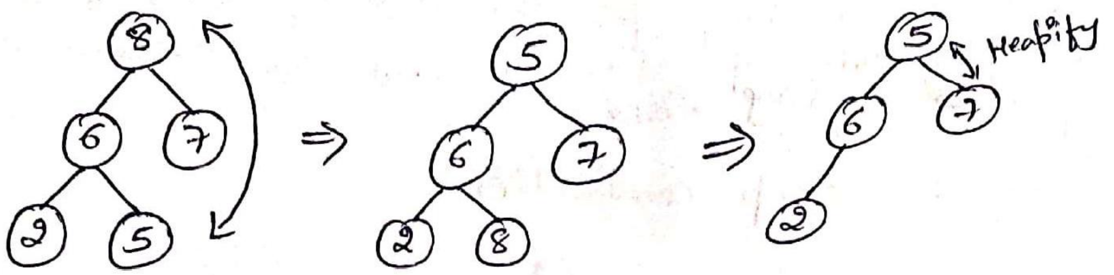


```
2   9   7   6   5   8
2   9   8   6   5   7
2   9   8   6   5   7
9   2   8   6   5   7
9   6   8   2   5   7
```

Stage 2: Maximum deletions

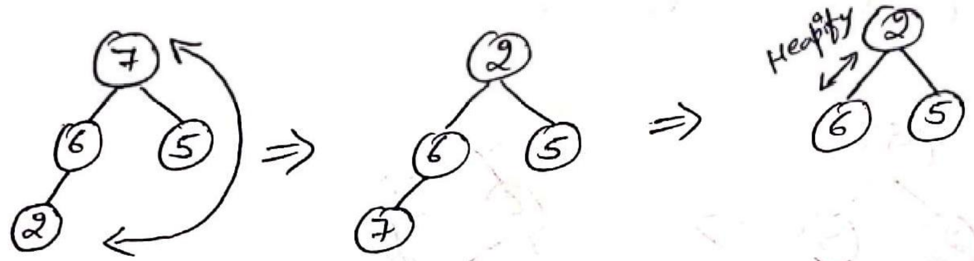


```
9   6   8   2   5   7
7   6   8   2   5 | 9
```



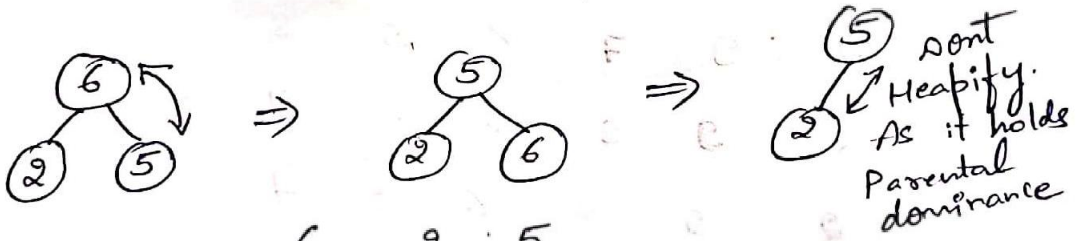
8 6 7 2 5

5 6 7 2 | 8



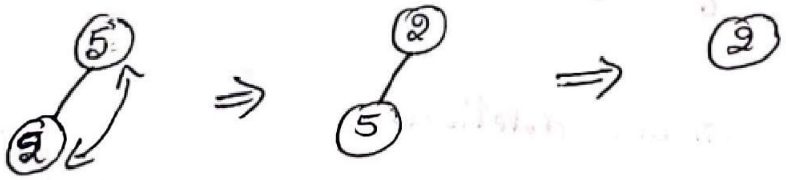
7 6 5 2

2 6 5 | 7



6 2 5

5 2 | 6



5 2

2 | 5

only one node is present.

2

9 6 8 2 5 7

7 6 8 2 5 | 9

8 6 7 2 5

5 6 7 2 | 8

7 6 5 2

2 6 5 | 7

6 2 5

5 2 | 6

5 2

2 | 5

2

<u>Assignment</u> :

1) Construct a heap for the list 1, 8, 6, 5, 3, 7, 4 by the bottom-up algorithm.

2) Sort the below given lists by heapsort using the array representation of heaps.
   a. 3, 2, 4, 1, 6, 5 (increasing order)
   b. 3, 4, 2, 1, 5, 6 (decreasing order)
   c. H, E, A, P, S, O, R, T (alphabetical order)

3) Write an efficient program for maximum key deletion from a given heap.

4) Design an efficient algorithm for finding and deleting an element of the smallest value in a heap and determine its time efficiency.

5) Implement three advanced sorting algorithms - mergesort, quicksort, and heapsort in the language of your choice and investigate their performance on arrays of sizes $n = 10^2, 10^3, 10^4$, 10^5 and 10^6. For each of these sizes, consider

a) randomly generated files of integers in the range [1...n]

b) increasing files of integers 1, 2, ...n

c) decreasing files of integers n, n-1, ...1

Features of heap sort :

1. The time complexity of heap sort is O(nlogn).
2. This is an in-place sorting algorithm. That means it does not require extra storage space while sorting the elements.
3. For random input it works slower than quick sort.
4. Heap sort is not a stable sorting method.
5. The space complexity of heap sort is O(1). As it does not require any extra storage to sort.

Note :- The heap can be represented by binary tree or array.

Why array based representation for binary heap?

Since a binary heap is a complete binary tree, it can be easily represented as array and array based representation is space efficient.

Maximum key Deletion from a heap :-

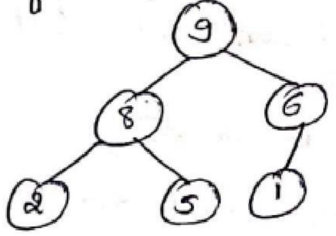
The procedure to be followed for deleting the maximum element from a given heap is as follows –

Step1 :- Exchange the root's key with the last key k of the heap.

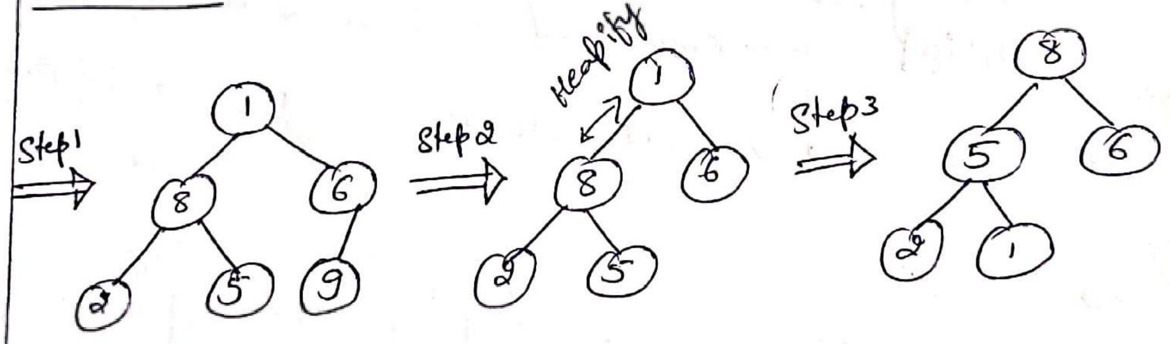
Step2 :- Decrease the heap's size by 1.

Step3 :- Heapify the smaller tree by shifting k down the tree, until the parental dominance condition holds for K.

Example : Demonstrate the maximum key deletion from the foll. given heap.



Solution :-



Note :- The data structure called the "<u>heap</u>" is a partially ordered data structure that is especially suitable for implementing <u>priority queues</u>.

Priority queue has the following operations :

1. Finding an item with the highest priority.
2. Deleting an item with the highest priority
3. Adding a new item to the multiset.

Priority queues finds its applications in scheduling job executions by computer operating systems and traffic management by communication networks.

<u>Assignment</u>

Construct a max heap for the foll. elements

10 9 6 5 7 8

<u>Time complexity analysis for heap sort</u> :

This algo works in two stages, we get the running time.

Running time for heap sort = Running time required by heap construction + Running time required by deletion of root key.

$$c(n) = c_1(n) + c_2(n) \quad \text{——} (1)$$

Heap construction requires $O(n)$ time.

$C_1(n) = O(n)$.

Now, let us compute time required by second stage.

Let no. of key comparisons needed for eliminating root keys from heap are for size from n to 2.

Then, we can establish the foll. relation :

$$C_2(n) \leq 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \dots + 2\lfloor \log_2 1 \rfloor$$

$$\leq 2 \sum_{i=1}^{n-1} \log_2 i$$

formula

$$\sum_{i=1}^{n} \log_2 i = n \log_2 n$$

similarly

$$\sum_{i=1}^{n-1} \log_2 i = (n-1)\log_2(n-1)$$

$$\leq 2(n-1)\log_2(n-1)$$

$$\leq (2n-2)\log_2(n-1)$$

$$\leq 2n\log_2(n-1) - 2\log_2(n-1)$$

$$\leq 2n\log_2 n$$

$$C_2(n) = O(n\log_2 n)$$

Substituting $C_1(n)$ and $C_2(n)$ in eq^n (1).

$$C(n) = O(n) + O(n\log_2 n)$$

$$\boxed{C(n) = O(n\log_2 n)}$$

<u>Note</u>:- <u>Analysis</u> : The basic operation in deletion algorithm is the key comparison that should be made to "heapify" the tree after the swap has been made. Each time, after deletion the size of the tree is decreased by 1. It requires less number of key comparisons than twice the heap's height. Therefore time complexity of deletion of O(log n).

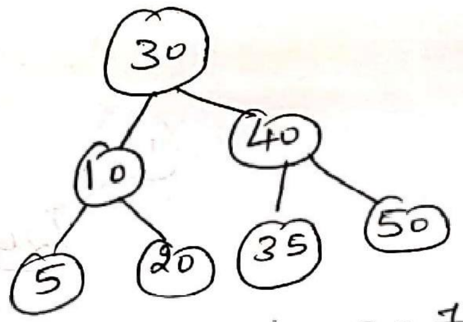
BALANCED SEARCH TREES

BINARY SEARCH TREE :-

It is a binary tree in which for each node say 'x' in the tree, elements in the left subtree are less than info(x) and elements in the right subtree are greater or equal to info(x).

Eg:- Given the list of elements, construct BST

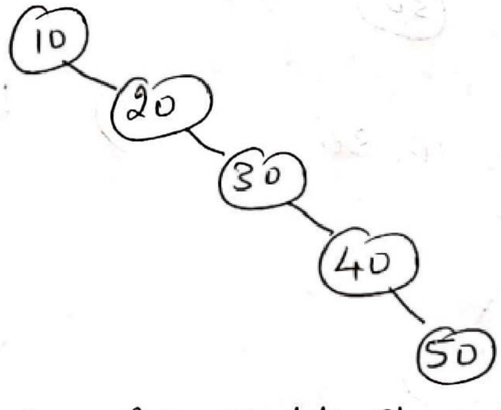
30, 40, 10, 50, 20, 5, 35



In the above tree, $n=7$ and $h=3$.

So, the operations like insertion, deletion and searching ~~takes~~ $O(\log n)$ time.

consider another set of elements like 10, 20, 30, 40, 50. and construct B.S.T.



$n=5$
$h=5$

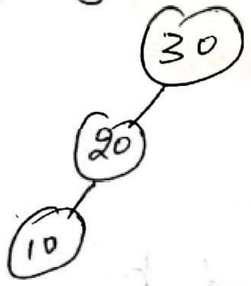
The above tree is Right skewed B.S.T. The operations like insertion, deletion, searching & traversing takes $O(n)$ time.

Even if the tree is left skewed B.S.T, operations takes $O(n)$ time.

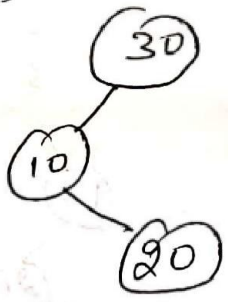
i.e if the tree is balanced, then the various operations takes less amount of time ($\log_2^n$) compared to imbalanced tree $O(n)$.

consider another set of three key elements 10, 20, 30 and construct B.S.T

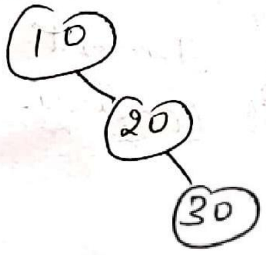
1) key = 30, 20, 10



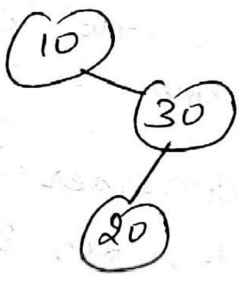
2) key = 30, 10, 20



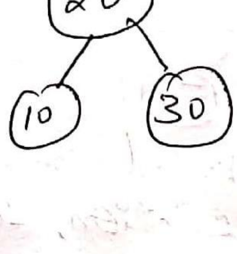
3) key = 10, 20, 30



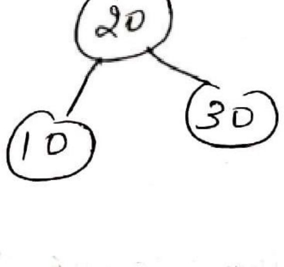
4) key = 10, 30, 20



5) key = 20, 10, 30



6) key = 20, 30, 10



Note1:- For same set of keys, if the order of insertion is different, then we get different shapes of binary search trees.

Note 2:- Given n keys, the total number of B.S.T possible is : n!

Eg:- n = 3, 3! = 6 B.S.T are possible.

Note3:- Min. height B.S.T is preferred, so that searching time reduces.

A binary search tree constructed may be a balanced tree or unbalanced tree.

To balance the B.S.T, we can use two approaches :

a) Instance simplification
b) Representation change

a) Instance simplification :- Here, the moment the tree is unbalanced, at that instant the tree is transformed so that tree is balanced tree

Eg. AVL trees, Red-black trees.

b) Representation change :- The B.S.T has one item in each node. Here, instead of only one item in each node, we allow more than one element in each node. This is representation change

Eg 2-3 trees, B-trees.

Scanned with CamScanner

AVL Trees :-

AVL tree was invented by two Russian Scientist G. M. Adelson - Velsky and E.M. Landis

Definition :-

"An AVL tree is an ordered binary search tree in which the height of the two subtrees of every node differ maximum by 1. i.e the height of the left subtree - the height of the Right subtree can be 0, 1 or -1".

If this condition is satisfied by each node in the binary tree then the tree is called AVL tree.

In an AVL tree each node is associated with balance factor, which is the height of the left subtree - height of the right subtree and is given by :-

$$\text{Balance factor} = \text{Height}\begin{pmatrix}\text{Left}\\ \text{subtree}\end{pmatrix} - \text{Height}\begin{pmatrix}\text{Right}\\ \text{subtree}\end{pmatrix}$$

Balance factor is 0 :

If the height of the left subtree and the height of the right subtree are same.

Balance factor is 1 :

If the height of the left subtree is 1 more than the height of the right subtree.

Balance factor is -1 :
If the height of the left subtree is 1 less than the height of the right subtree.

Example:-
Consider the following Binary Search Tree

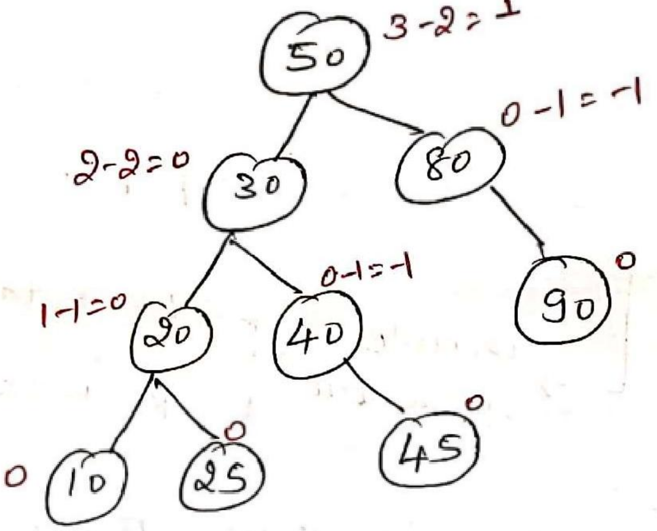


fig (a) AVL Tree

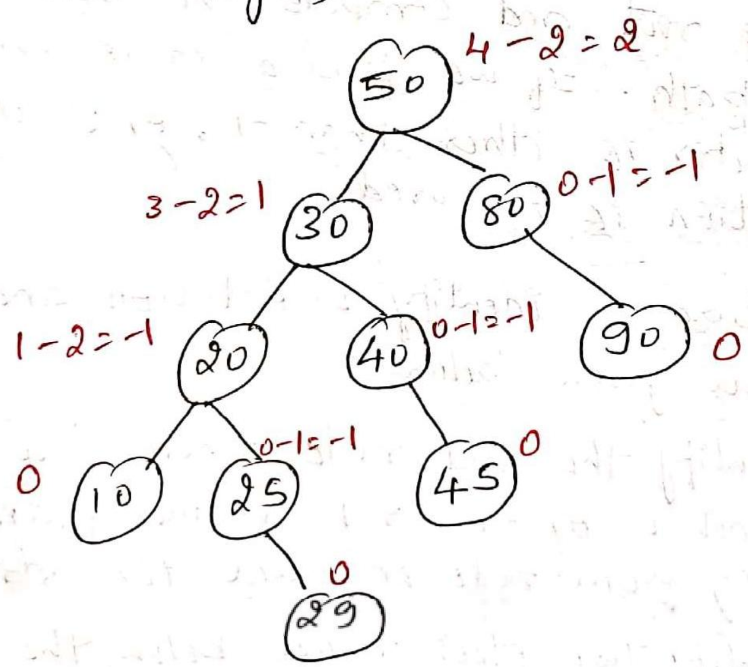


fig (b) Not an AVL tree because it is not balanced (since balanced factor of 50 should be 1 and not 2)

Creating an AVL Tree :-

Once an item is inserted into the tree, the tree may be unbalanced. In such case, it is necessary to balance the tree. Balancing of AVL tree is done using rotations. The various type of rotations are :

$$\begin{tikzcd}
 & \text{Single rotation} \arrow[r] \arrow[dr] & \text{L-Rotation (left)} \\
\text{Types of Rotations} \arrow[ur] \arrow[dr] & & \text{R-Rotation (Right)} \\
 & \text{Double rotation} \arrow[r] \arrow[dr] & \text{LR Rotation (Left Right)} \\
 & & \text{RL Rotation (Right left)}
\end{tikzcd}$$

After inserting a node, trace backward towards the root and compute the balance factor along the path. If we find a node whose balance factor is other than -1, or 0 or +1, then rotation is required.

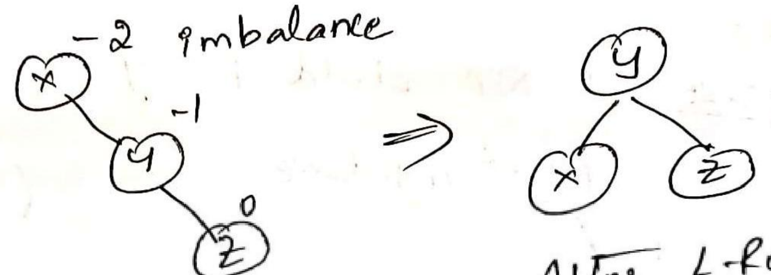
The procedure to identify L-Rotation and R-Rotation is given below :

Step 1 : Identify the first node where balance factor is not 0 or -1 or 1 in the path while moving from node inserted towards root.

Step 2 :- Identify two other nodes below the node where imbalance occurs in the path.

1) L-Rotation :- If the three nodes identified in step1 and step2 are in a straight line, single rotation is required. If the balance factor of a node, where imbalance occurs. is -2, then the tree is heavy towards right. so to balance it we rotate left. This is called left rotation.

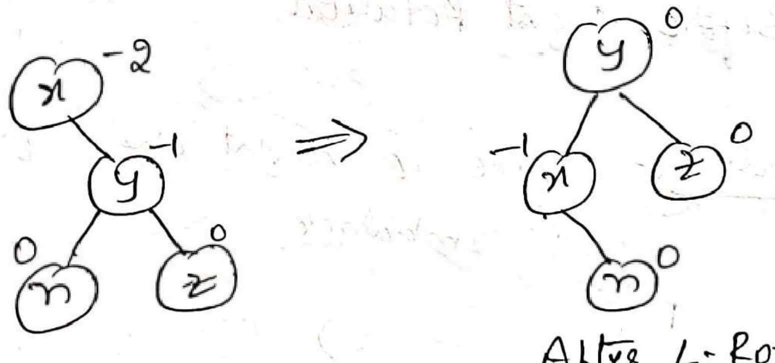
Case 1 :-



Before L-rotation

After L-Rotation

Case 2 :- There is a left child for y



Before L-Rotation

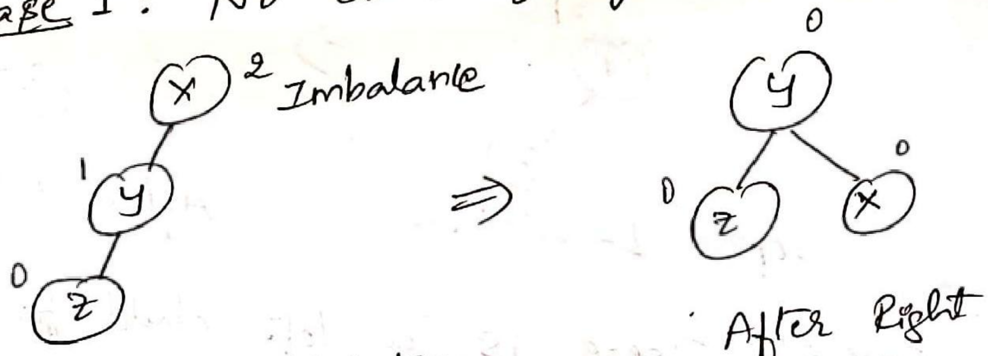
After L-Rotation

2) R-Rotation :-

If the three nodes identified in Step1 and Step2 are in a straight line, single rotation is required. If the balance factor of a node, where imbalance occurs is 2, then tree is heavy towards left. So to balance it we rotate right. This is called right rotation.

Eg:-

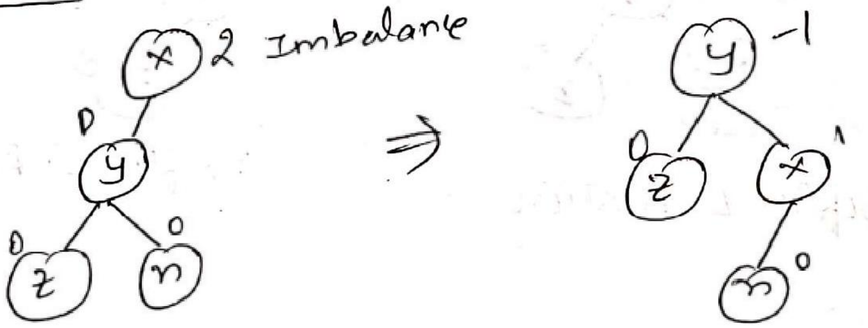
Case 1 : No child for y



Before Right Rotation

After Right Rotation

Case 2:- There is right child for y



Before Right Rotation

After Right Rotation

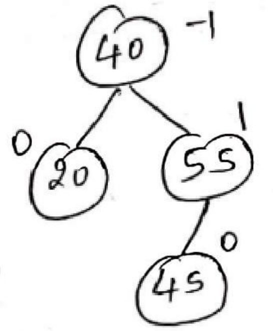
(ii) L-R Rotation :-

If the three nodes identified are not in a straight line, then double rotation is required. A double rotation is a combination of two single rotations as shown below:

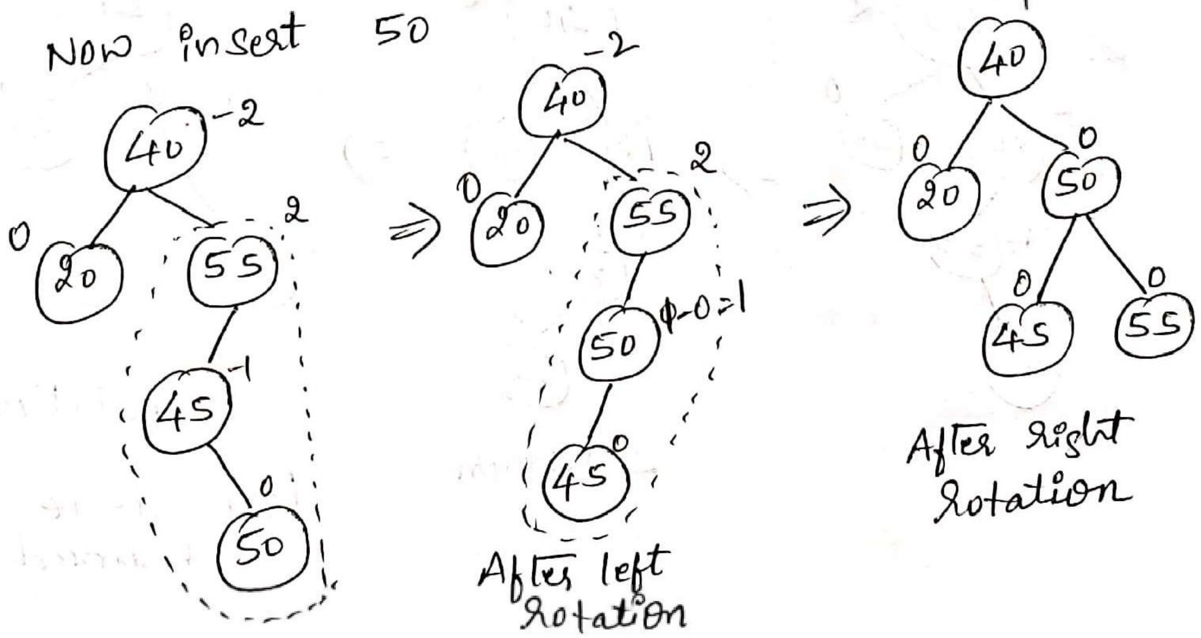
i) Assume 'X' is a node where imbalance occurs and y is its child in the path. If the balance factor of y is -1, this subtree is right heavy, so rotate left at y. Attach the parent of resulting subtree to X.

ii) Second rotation is required at X. Here right rotation is required.

Eg:- Consider the following tree



Now insert 50



After left rotation

After right rotation

(iv) R-L Rotation :-

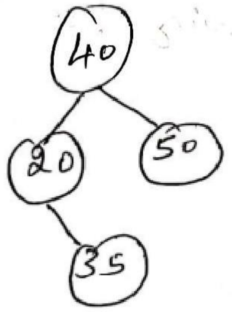
If the three nodes identified are not in a straight line, then double rotation is required.

A double rotation is the combination of two single rotations as shown below :

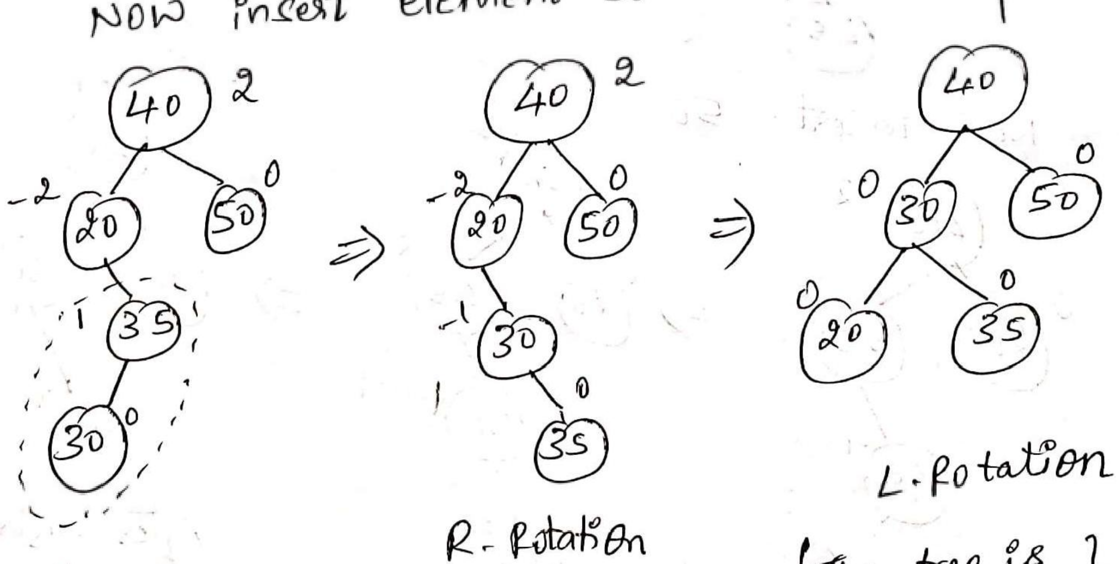
(i) Assume 'x' is a node where imbalance occurs and 'y' is its child in the path. If the balance factor of y is 1, this subtree is left heavy, so rotate right at y. Attach the parent of resulting subtree to x.

(ii) Second rotation is required at x. Here left rotation is required.

Eg:- Consider the following tree :



Now insert element 30



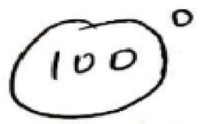
{The tree is balanced}

Problems :-

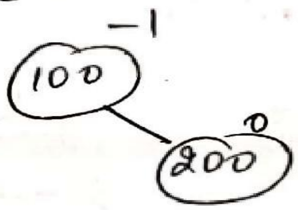
1) Construct an AVL tree by inserting the elements 100, 200, 300, 250, 270, 70 and 40. successively starting from an empty tree.

Solution:-

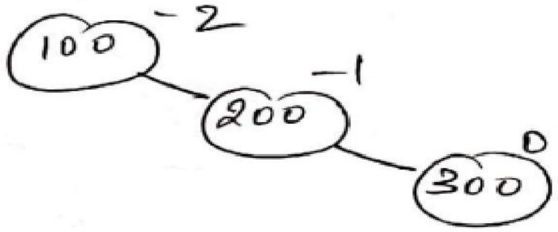
Step1 : Insert 100 to empty tree.



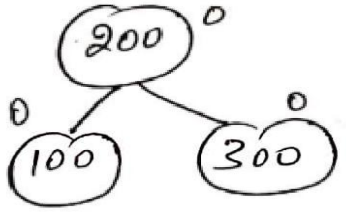
Step2:- Insert 200 to ~~empty tree~~. Step1.



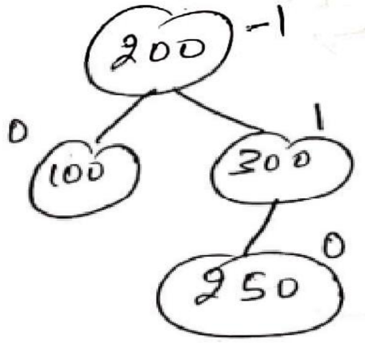
Step3:- Insert 300 to Step2.



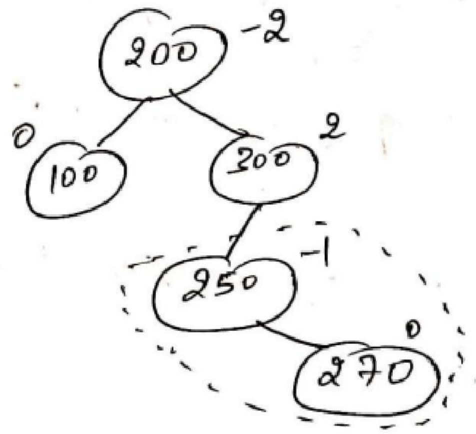
Step4:- Perform left rotation to step3 to make the tree balance



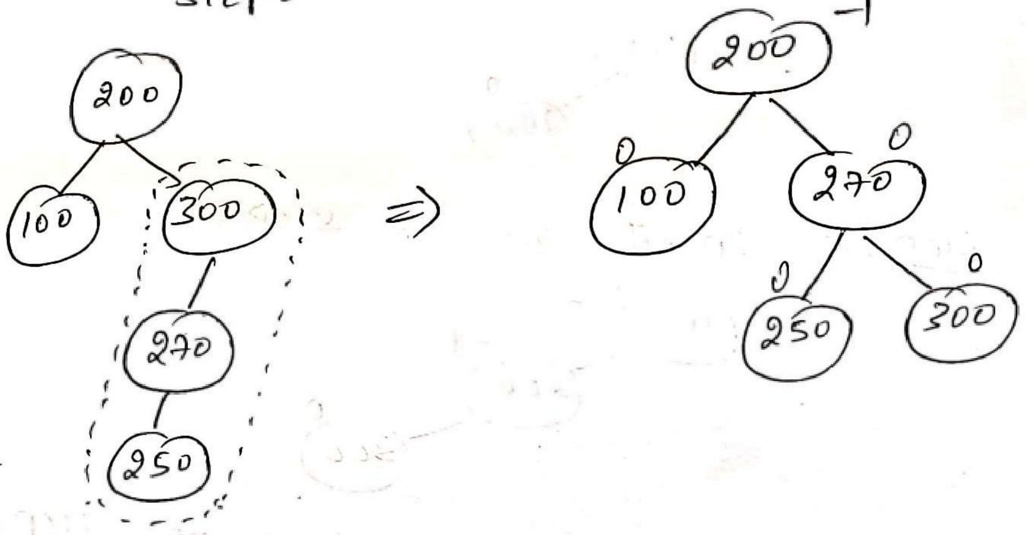
Steps:- Insert 250 to Step4.



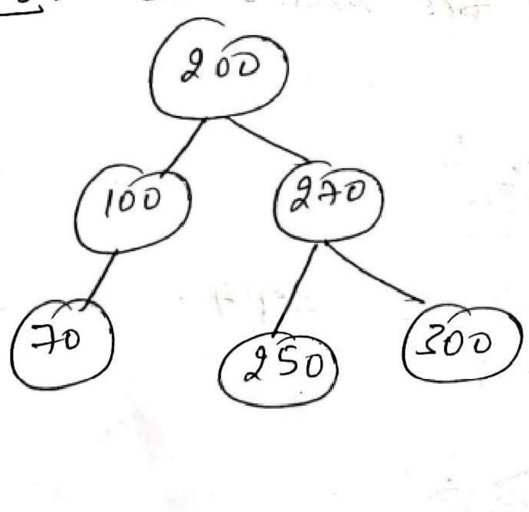
Step6 :- Insert 270 to Step 5



Step7 :- Perform Left-Right rotation to step6 to make it balance.

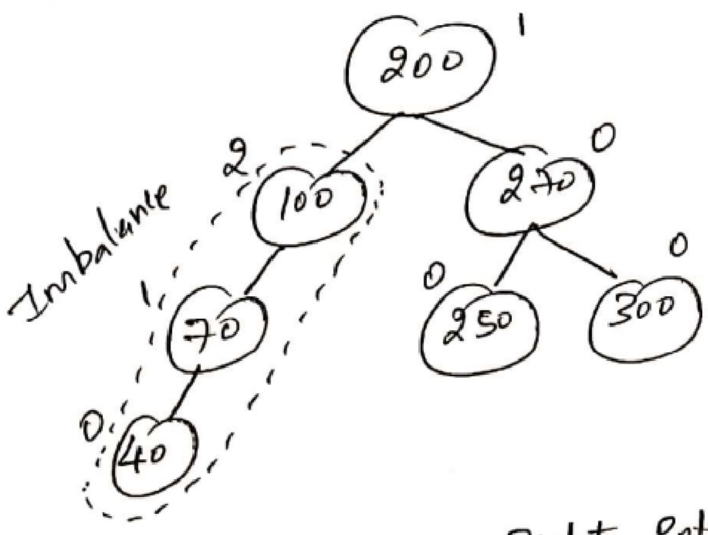


Step8 :- Insert 70 to Step 7

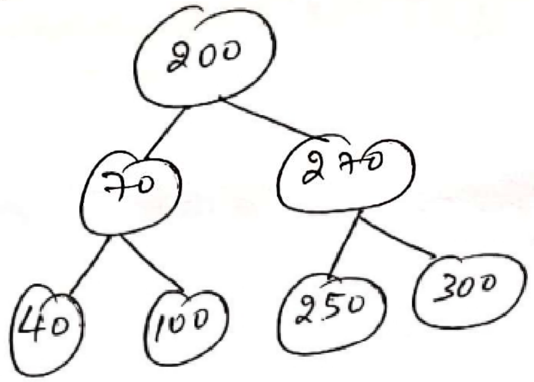


Scanned with CamScanner

Step 9:- Insert 40 to step 8.



Step 10:- Perform Right Rotation to step 9 to make it balance.



Problem 2:- Construct an AVL tree by inserting the items 25, 26, 28, 23, 22, 24 and 27 successively starting from an empty tree.

Problem 3:- Construct an AVL tree by inserting the items 1, 2, 3, 4, 5 and 6

Space-Time Tradeoffs - Sorting by Counting

Space- and time tradeoffs in algorithm design are a well known issue for both theoreticians and practitioners for computing

The idea is to process the problem's input, in whole or in part and store the additional information obtained to accelerate solving the problem afterward.

The approach is also called as input enhancement.

Based on that

1. counting methods for sorting
2. Boyer Moore algorithm for string matching suggested by Horspool.

Sorting by counting

Applying the input-enhancement technique, to the sorting problem.

Count for each element of a list to be sorted, the total no of elements smaller than this element and record the results in a table. The no will indicate the positions of the elements in the sorted list.

Also called as comparision counting sort

Example

	0	1	2	3	4	5
A[0...5]	62	31	84	96	19	47

Initially count[]	0	0	0	0	0	0
i= 0	3	0	1	1	0	0
		1	2	2	0	1
			4	3	0	1
				5	0	1
					0	2
	3	1	4	5	0	2

0	1	2	3	4	5
19	31	47	62	84	96.

```
Algorithm ComparisionCountingSort (A[0... n-1])
  // Sorts an array by comparision counting
  // Input: An array A[0.. n-1] of orderable elements
  // output: Array S[0.. n-1] A's elements in
  //                       non decreasing order.
   for i ← 0 to n-1 do count[i] ← 0
    for i ← 0 to n-2 do
         for j ← i+1 to n-1 do
              if A[i] < A[j]
                count[j] ← count[j] +1
              else
                count[i] ← count[i]+1
         for i ← 0 to n-1 do S[count[i]] ← A[i]
    return S.
```

Time efficiency:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}$$

$$\underline{O(n^2)}$$

Sorting by Counting - Distribution Counting

Consider the array

13	11	12	13	12	12

Array values {11, 12, 13}

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

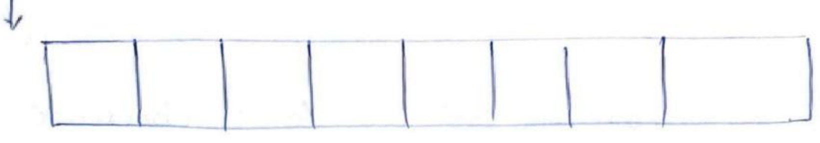
		D[0 ... 2]			S[0 .. 5]					
A[5] :	12	1	4	6				12		
A[4]	12	1	3	6			12			
A[3]	13	1	2	6						13
A[2]	12	1	2	5		12				
A[ϕ]	11	1	1	5	11					
A[0]	13	0	1	5					13	

```
Algorithm DistributionCoding(A[0 .. n-1], l, u)
// Sorts an array of integers from a limited range
                         by distribution counting
// Input: An array A[0 -- n-1] of integers b/w l & u
                                           l ≤ u
// Output: Array S[0 -- n-1] of A's elements sorted
                          in non decreasing order
    for j ← 0 to u-1 do D[j] ← 0
     for i ← 0 to n-1 do D[A[i]-l] ← D[A[i]-l] + 1
     for j ← 1 to u-1 do D[j] ← D[j-1] + D[j]
      for i ← n-1 down to 0 do
           j ← A[i] - l
           S[D[j] - 1] ← A[i]
            D[j] ← D[j] - 1
      return S
```

Horspool Algorithm

- used for pattern matthing
- to find a substring in a given string

Given string



pattern string

Shifts more than '1' position

To decide the positeon shift table is constructed.

① Consider the pattern String BANGALORE

Length of String is 9

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
~~7~~ 4	8					5					3		6	2			1									

Algorithm

P array stores the pattern
Length of the pattern is m.
pattern is stored in P from 0 to m-1
ST[0 to asize-1] is the shift table where asize is alphabet size.

asize = 256 for all the ASCII characters,
asize = 26

Input: pattern string P output : shift table ST.

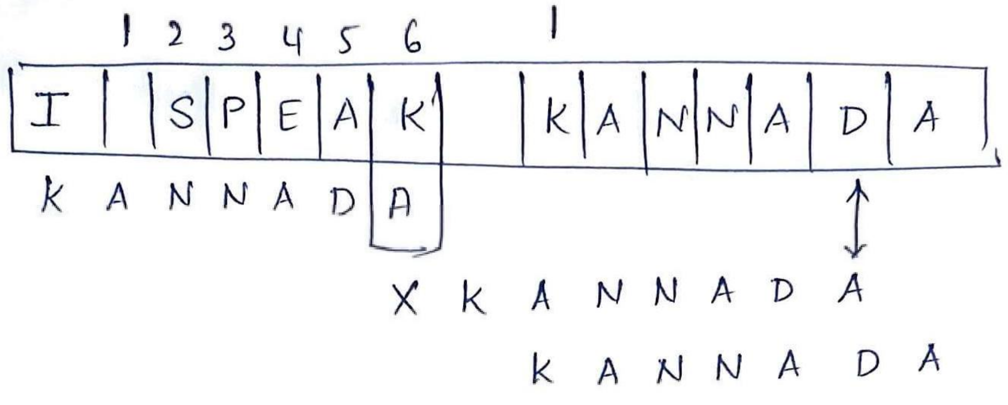
```
Algorithm Shift-table (P, ST)
{
    for i = 0 to asize - 1
        ST[i] = m;
    for j = 0 to m-2
    {
        index = ASCII value of P[j] - ASCII value of 'A'
        ST[index] = m-1-j;
    }
}
```

(2) Ex:- Pattern string "~~BA~~ KANNADA

0	1	2	3	4	5	6
K	A	N	N	A	D	A

size = 27
m = 7

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	
~~7~~ ~~5~~ 2	7	7	~~7~~ 1	7	7	7	7	7	7	~~7~~ 6.	7	7	~~7~~ ~~4~~ 3	7	7	7	7	7	7	7	7	7	7	7	7	7



Horspool algorithm

```
Algorithm Horspool (S, P)
{
    n= length(S);
    m= length (P);
    Shifttable (P, ST);
    i= m-1;          ———— Start the comparision from
                           last character
    while(i <=n-1)
    {
        K=0
        while( K<=m-1 and P[m-1-K]= S[i-K])
            K= K+1;
        if(K==m)
        then return i -m+1
        else i= i + ST[S[i]];
    }
    return -1;
}
```