Hanumanthu
4th sem
CSE

Module-03

Greedy Methods

chapter-01
Greedy Method

Topic-01

Greedy Method

Greedy Method

* In Greedy Technique, The solution is constructed through a sequence of steps each Expanding a partially constructed solution obtained so far, until a complete solution to the problem is Reached.

* while Making a choice there should be a Greed for the optimum solution.

Algorithm

Greedy(D, n)

// In Greedy approach D is a domain
// from which solution is to be obtained of size n

// Initially ~~Assume~~

solution ← 0

for i ← 1 to n do

{

S ← Select (D)

if (Feasible (solution, S)) then

solution ← Union (solution, S);

}

return solution.

Greedy Method following Activities are performed

1. First we select some solution from input domain.

2. Then we check whether the solution is feasible (or) not.

3. From the set of feasible solutions, particular solution that satisfies (or) nearly satisfies the objective of the function.
   Such a solution is called Optimal solution

4. As Greedy method works in stages.
   * At Each stage only one input is considered at each time.
   * Based on this input it is decided whether particular input gives the optimal solution (or) not.

Application of Greedy Method

1. Optimal Merge Patterns
2. Huffman coding.
3. Minimum Spanning Tree.
4. Knapsack Problem
5. Job Sequencing with deadlines
6. Single Source Shortest path Algorithm.

General Characteristics

1. **Greedy Choice Property**

* A Globally Optimal Solution can be Arrived at by making a locally optimal choice.
* That means for finding the Solution to the Problem.
* The Greedy choice property Brings

Efficiency in solving the problem with the help of subproblems.

2. Optimal substructure

* A Problem shows optimal substructure if An optimal solution to the problem contains optimal solution to the Sub- Problems.

Topic -02

Coin Change Problem

Coin change Problem

Statement :- " The Coin change Problem is a Problem in which there are some coin denominations Using which we have to make change for Amount S Using smallest number of Coins"

Algorithm

Input : 1) Coin denominations $d[1] > d[2] \ldots > d[m] = 1$

2) Amount S for obtaining change

Output : The optimal number of coins for change of S.

It is stored in coins [i].

Procedure: for i := 1 to m do

$$\{ \text{coins}[i] = \left\lfloor \frac{S}{d[i]} \right\rfloor$$

$$S = S \bmod d[i]$$

$$\}$$

Example:-

The Coin Denominations are 1, 5, 10, 20, 25, obtain the change for an Amount $S = 40$

Solution

1. we will start from largest coin denomination i.e 25

Then solution 1 = 25×1, 10×1, 5×1 = 3 coins

2. solution - 2 : 20×2 = 2 coin

3. solution 3 : 20×1, 10×2 = 3 coins

4. solution 4 : 10×4 = 4 coins

5. solutions : 5×8 = 8 coins

These are some feasible solution. out of which solution 2 is optimal solution.

Topic-03

Knapsack Problem

** The Knapsack Problem can be started as follows.

* Suppose there are n objects from i = 1, 2, 3 - - - n.

* Each object i has some positive weight w_i & some profit value i.e. Associated with Each object which is denoted as p_i.

* This Knapsack carry at the most weight W.

* while solving Above mentioned Knapsack problem we have the capacity Constraint.

when we try to solve ~~this~~ problem using Greedy Approach our Goal is,

1. Choose only those objects that give maximum profit.

2. The Total weight of selected objects should be $\leq W$.

And then we can obtain the set of feasible solutions.

$$\boxed{\text{Maximized } \sum_{n}^{1} p_i x_i \text{ subject to } \sum_{n}^{1} w_i x_i \leq W}$$

where the Knapsack can carry the fraction x_i of an object i such that $0 \leq x_i \leq 1$ and $1 \leq i \leq n$.

Example:- Consider that There are three items. Weight & profit value of each item is as given Below.

i	w_i	p_i
1	18	30
2	15	21
3	10	18

Also $W = 20$.

obtain the solution for the Above given ~~the~~ Knapsack problem.

Solution :- The feasible solutions are as

x_1	x_2	x_3
$1/2$	$1/3$	$1/4$
1	$2/15$	0
0	$2/3$	1
0	1	$1/2$

Let us compute $\sum w_i x_i$

1. $1/2 * 18 + 1/3 * 15 + 1/4 * 10$

$= \underline{16.5}$

2. $1*18 + 2/5*15 + 0*8$

$= \underline{20}$

3. $0*18 + 2/3*15 + 10$

$= \underline{\underline{20}}$

4. $0*18 + 1*15 + 1/2*10$

$= \underline{\underline{20}}$

Let us compute $\sum P_i x_i$

1. $1/2*30 + 1/3*21 + 1/4*18$

$= \underline{26.5}$

2. $1*30 + 2/15*21 + 0*18$

$= \underline{32.8}$

3. $0*30 + 2/3*21 + 18$

$= 32$

4. $0 \times 30 + 1 \times 21 + 1/2 \times 18$

$= 30$

To summerise this

$\sum w_i x_i$	$\sum p_i x_i$
16.5	26.5
20	32.8
20	32
20	30

The solution 2 gives the maximum profit & hence it turns out to be optimum solution.

Algorithm

The Algorithm for solving knapsack problem with Greed Approach is as given Below.

Algorithm Knapsack_Greedy(m, W, n)

{

// P[i] contains the profits of i items such that

$1 \leq i \leq n$

// w[i] contains weights of I items

// x[i] is the solution vector.

// W is the total size of knapsack

for i := 1 to n do

{

if (w[i] < W) then // capacity of knapsack is a constraint

{

x[i] := 1.0;

W = W - w[i];

}

if (i<=n) then x[i] := W/w[i];

}

Analysis :- The Basic Operation is Comparing Total weight of Knapsack objects with capacity.

* Hence only if loop is Applied.
* Thus Time Complexity O(n)

Topic-04

Solving job sequencing with deadlines Problems

* Consider that there are n jobs that are to be Executed.

* At Any Time $t = 1, 2, 3, \ldots$ only Exactly one job is to be Executed.

* The profit P_i are Given.

* These profits are Gained by Corresponding jobs.

* For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

Let n=4

n	P_i	d_i
1	70	2
2	12	1
3	18	2
4	35	1

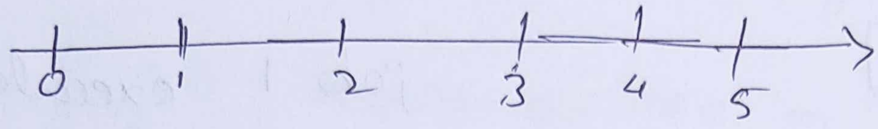
we will follow following Rules to obtain the feasible solution

* Each job takes one unit of time
* If job starts Before or at it's deadlines profit is obtained, otherwise no profit

* Goal is to schedule jobs to maximize the Total profit.

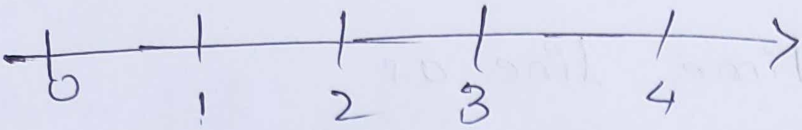
* Consider all possible schedules & compute the minimum Total time in the system.

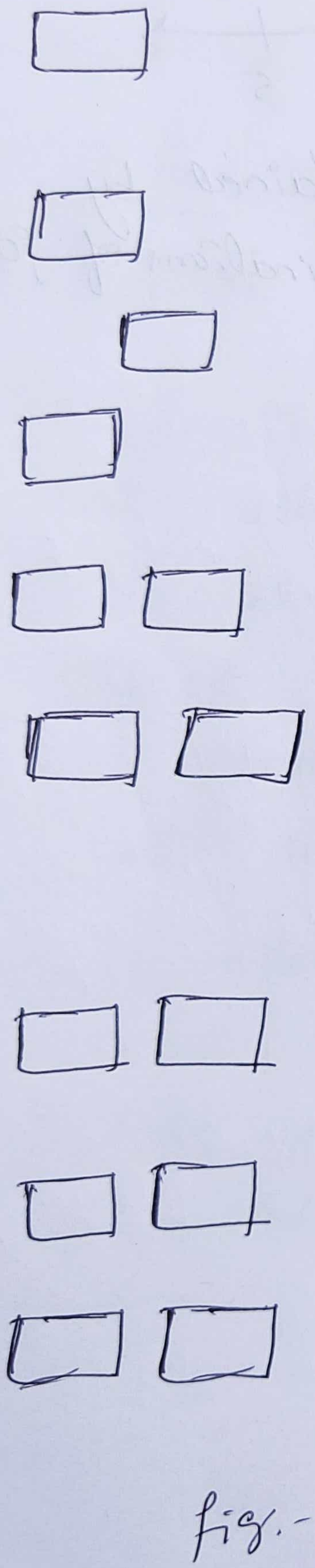
Consider the time line as



The feasible solutions are obtained by various permutations & combinations of jobs.

n	P_i
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105
4,3	53





job 1 executes

job 2 executes

job 3 executes

job 4 executes

job 1,3 or 3,1

job 2,1 executes.

we cannot select 1,2
Because di of job 2
is 1.

job 2,3 executes

job 4,1 executes

job 4,3 executes.

fig.- Job Sequencing with deadline

* Each job takes only one unit of time.

* Deadline of job means a time on which or Before which the job has to be Executed.

* The feasible sequence is a sequence that allows all jobs in a sequence to be Executed within their deadlines & highest profit can be gained.

1. The Optimal Solution is a feasible solution with maximum profit.

2. In Above Example sequence 3, 2 is not considered as $d_3 > d_2$ but we have considered the sequence 2, 3 as feasible solution Because $d_2 < d_3$

Algorithm

The Algorithm for job Sequencing is as given Below

Algorithm Job_seq(D, J, n)

{

//Problem Description : This Algorithm is for job Sequencing Using Greedy Method.

//D[i] denotes i^{th} deadlines where $1 \leq i \leq n$

//J[i] denotes i^{th} job.

// $D[J[i]] \leq D[J[i+1]]$

//initially

D[0] ← 0;

J[0] ← 0;

J[1] ← 1;

count ← 1;

for ← 2 to n do

{

t ← count

while (D[J[t]] > D[i] AND D[J[t]]

! = t)) do t ← t-1;

if (D[J[t]] ≤ D[i]) AND (D[i] > t))

then

{

// insertion of i^{th} feasible sequence into J array.

for s ← count to (t+1) step -1 do

J[s+1] ← J[s];

J[t+1] ← I;

count ← Count+1;

} // end of if

} // end of while

return count;

}

Analysis

* The sequence of J will be inserted if and only if D[J[t]] ! = t.

* This also means that the job J will be processed if it is in within the deadline.

* The computing time taken by above Job_Seq Algorithm is $O(n^2)$, Because the Basic operation of Computing Sequence in Array J is within two nested for loops.

Example

1. Using Greedy Algorithm find an optimal schedule for following jobs with $n=7$ profits: $(P1, P2, P3, P4, P5, P6, P7) = (3, 5, 18, 20, 6, 1, 38)$ and deadline $(d1, d2, d3, d4, d5, d6, d7) = (1, 3, 3, 4, 1, 2, 1)$.

Solution:-

Step - 1 :- We will Arrange the profits p_i in descending order.

* The Corresponding deadlines will Appear.

Profit	38	29	18	6	5	3	1
Job	P_7	P_4	P_3	P_5	P_2	P_1	P_6
Deadline	1	4	3	1	3	1	2

Step-2

Create an Array J[] which stores the jobs. Initially J[] will be

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

J[7]

Step-3

* Add i^{th} job in Array J[] at the index denoted by its deadline.
* D_i First job is P_7.
* The deadline for this job is 1.

* Hence insert P_7 in the array J[ ] at 1st index.

1	2	3	4	5	6	7
P_7						

Step-4

Next job is P_4. Insert it in array J[ ] at index 4.

1	2	3	4	5	6	7
P_7			P_4			

Step-5

Next job is P_3. Its has a deadline 3. Therefore insert it at index 3.

1	2	3	4	5	6	7
P_7		P_3	P_4			

Step-6 :- Next job is P_5, it has deadline 1.

* But as 1 is already occupied & there is no empty slot at Index < J[1].

* Hence just discard job P_5. Similarly job P_5 will get discarded.

<u>Step-7:</u> Next job is P_2.

* It has a deadlines 3.
* There is an empty slot at index < J(3)

Therefore insert it at Index 2.

1	2	3	4	5	6	7
P_7	P_2	P_3	P_4			

<u>Step-8:-</u> Thus we now obtain the job sequence as <u>7-2-3-4</u> with the profit of 81.

Chapter-02

Minimum Cost Spanning Trees

Topic-01

Prim's Algorithm

* In this method, we will consider all the vertices first.

* Then we will select an Edge with minimum weight.

* The Algorithm proceeds by selecting adjacent Edges with Minimum weight.

* Care should be taken for not forming Circuit

Algorithm

Prim's Algorithm

Algorithm

Prim (G [0 - - - size-1, 0 - - - size -1], nodes)

// Problem Description: This Algorithm is for implementing

// Prim's algorithm for finding spanning tree.

// Input: Weighted Graph G and Total Number of nodes

// output: Spanning Tree gets printed with total path Length.

total = 0;

// Initialize the selected vertices list.

for i ← 0 to nodes -1 do

tree [i] ← 0

tree [0] = 1; // take Initial vertex.

for k ← 1 to nodes do

{

```
for j ← 0 to nodes -1 do
{
 if (G [i, j] AND (tree [i] AND !tree [j])
     OR (!tree [i] AND tree [j]))) then
   {
    if (G [i, j] < min_dist) then
     {
       min_dist ← G [i, j]
           v1 ← i
           v2 ← j
     }
   }
  }
 }
write (v1, v2, min_dist);
tree [v1] ← tree [v2] ← 1
total ← total + min_dist
}
write ("Total Path Length is", total).
```

Analysis

* The Algorithm spends most of the Time in selecting the Edge with minimum length.

* Hence the Basic Operation of this Algorithm is to find the Edge with minimum path length.

* This is can be given by following formula

$$T(n) = \sum_{k=1}^{nodes-1} \left(\sum_{i=0}^{nodes-1} 1 + \sum_{j=0}^{nodes-1} \right)$$

Time taken by for k=1 to nodes-1 loop

Time taken by for i=0 to nodes-1 loop

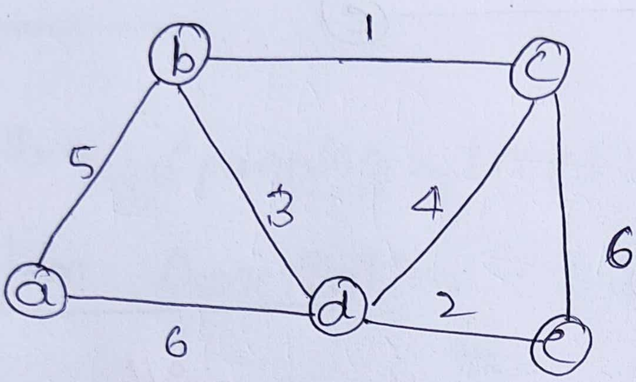
Time taken by for j=0 to nodes-1 loop.

Time complexity of Prim's Algorithm is

$\Theta(|V^2|)$.

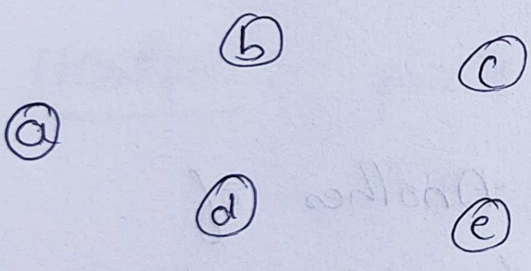
Example

Find the Minimum spanning tree prims Methode for the following graph.

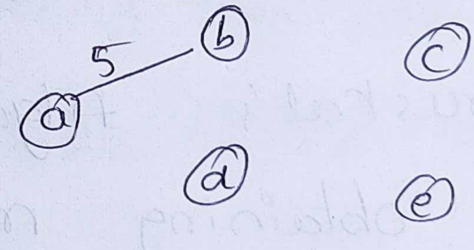


Solution:-

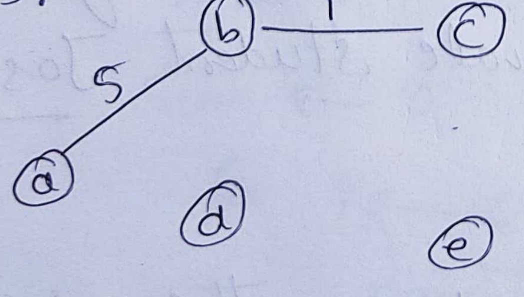
Step-1



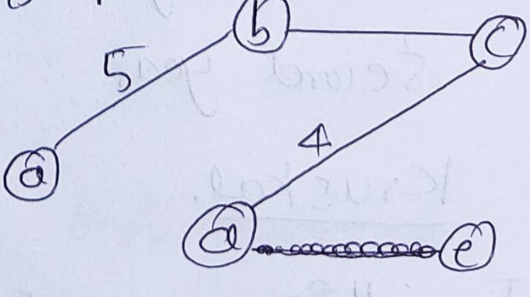
Step - 2:-



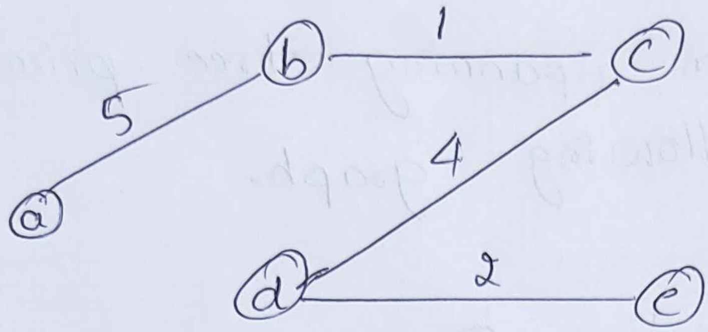
Step -3:-



Step -4



Step- 5 :-



Topic-02

Kruskal's Algorithm with Performance Analysis

Kruskal's Algorithm

* Kruskal's Algorithm is Another of obtaining minimum spanning tree.
* This Algorithm is discovered by a second year Graduate student Joseph Kruskal.
* In this Algorithm always the minimum cost Edge has to be selected.

* But it is not Neccessary that selected optimum Edge is adjacent.

Algorithm

Algorithm spanning_tree()

//Problem Description : This Algorithm finds the minimum.

//spanning tree Using Kruskal's Algorithm.

//Input :- The Adjacency matrix graph G containing cost.

//output :- prints the spanning tree with The total cost of spanning Tree.

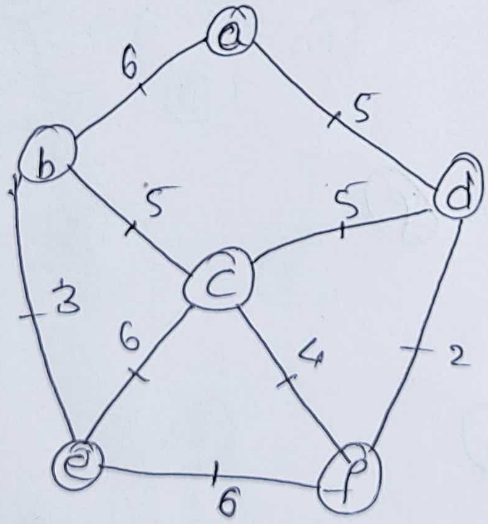
```
count ← 0
k ← 0
sum ← 0
  for i ← 0 to tot_nodes do
      parent[i] ← i
while(count != tot_nodes-1) do
{
```

```
pos ← minimum (tot_edges);
if( pos = -1) then
    break
  v1 ← G[Pos].v1;
  v2 ← G[Pos].v2.
  i ← find (v1, parent)
  j ← find (v2, parent)
  if (i != j) then
  {
    tree [k][0] ← v1

    tree [k][1] ← v2

      k++
      count++;
      sum + ← G[pos].cost
      Union (i, j, parent);
  }
  G[Pos].cost INFINITY
}
if (count == tot_nodes-1) then
```

{
for i ← 0 to tot_nodes − 1
{
write (tree[i][0], tree[i][1])
}
write("Cost of spanning Tree is", sum).
}.

Example:

Apply Kruskal's Algorithm to find Minimum spanning Tree of the graph shown in fig.



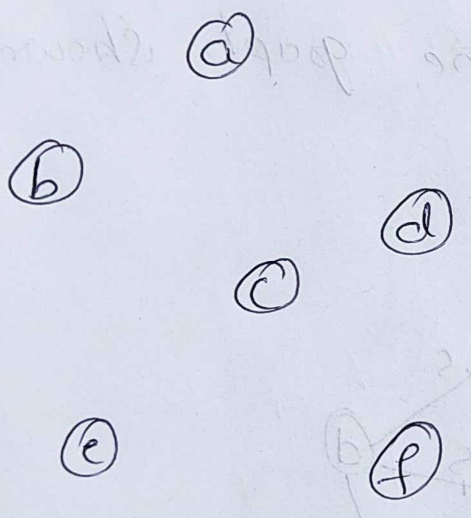
Solution:

* First we will select all the vertices.

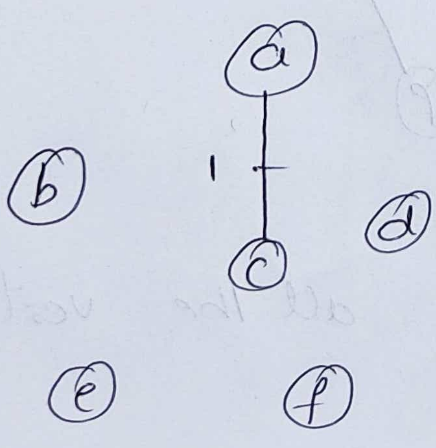
* Then an Edge with optimum weight is selected from heap. even though it is not adjacent to previously selected edge.

* Care should be taken for not forming Circuit.

Step-1:-



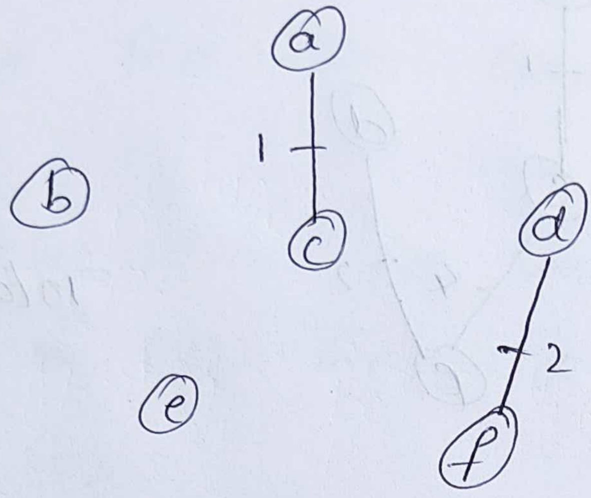
Step-2



Total weight

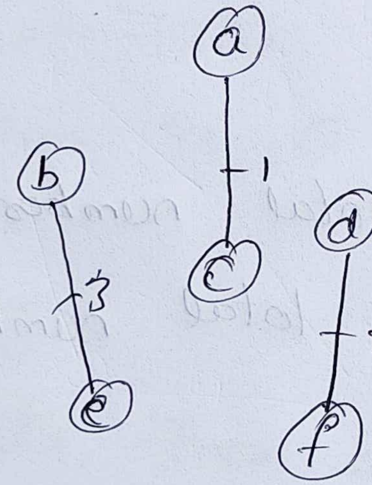
=1

Step-3 :-



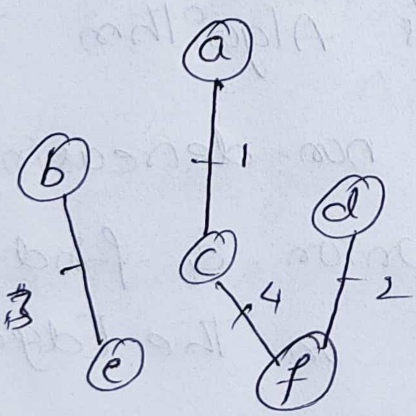
Total weight = 3

Step-4 :-



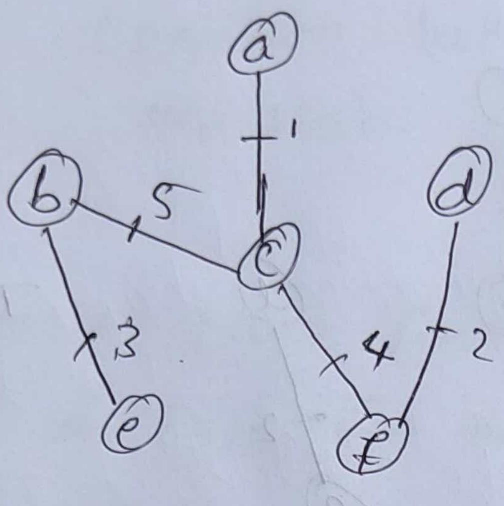
Total weight = 6

Step- 5



Total weight = 10

Step-6:



Total weight = 15

Thus total cost of Above minimum spanning tree is 15.

Analysis

* Let E be the Total number of Edges & V denotes the total number of vertices of the Graph.

* In the Kruskal's Algorithm the Edges are first sorted in non-decreasing order & then the Union & find operations are performed on The Edges.

* The Sorting of Edges takes $O(E \log E)$.

* The Find Operation takes $O(E)$ and whereas the Union Operation takes $O(V)$ time.

* The Total time for Union & find is $O(E\,\alpha(V))$ The $\alpha(V) = \log(V) = \log(E)$.

* Thus the total Time is dominated by the Sorting.

Hence the Time complexity of Kruskal's Algorithm is $\underline{O(|E| \log |E|)}$

Applications

1) Kruskal's tree are very Important in designing Efficient Routing Algorithms.
2) Network Design makes Use of Kruskal's Algorithm.

* The Sorting of Edges takes $O(E \log E)$.
* The Find Operation takes $O(E)$ and whereas the Union Operation takes $O(V)$ time.
* The Total time for Union & find is $O(E\,\alpha(V))$ The $\alpha(V) = \log(V) = \log(E)$.
* Thus The total Time is dominated by the Sorting.

Hence the Time complexity of Kruskal's Algorithm is $\underline{O(|E| \log |E|)}$

<u>Applications</u>

1) Kruskal's tree are very Important in designing Efficient Routing Algorithms.
2) Network Design Makes Use of Kruskal's Algorithm.

Chapter-03

Single Source Shortest paths

Topic-01

Dijkstra's Algorithm

* Many ~~cities~~ Times Graph is Used to Represent the distance B/w two cities.

* Everybody is often ~~Interste~~ Interested in Moving from one city to other as Quickly as possible.

* The single source shortest path is Based on this Interest.

* In single source shortest path problem the shortest distance from a single vertex called source is obtained.

* Let G (V, E) be a Graph, then in single source ~~shortest~~ shortest path the shortest paths from vertex V_0 to all Remaining vertex is determined.

* The vertex V_0 is then called as source

* The last vertex is called destination.

* It is Assumed that all the distances are positive

Algorithm

The Algorithm for single source shortest path is given as.

Algorithm single_short_path(p, Cost, Dist, n)
{
//Cost is an adjacency matrix ~~cost~~ storing cost of each Edge i.e. Cost[1:n, 1:n]. Given graph can be Represented by Cost.

//Dist is a set of that stores the shortest path from the source vertex 'p' to any other.

// Vertex in the graph.

//S stores all the visited vertexes of graph. It is of Boolean type array.

//Initially

for i ← 1 to n do

{

S[i] ← 0;

Dist ← cost[p, i];

}

S[p] ← 1

Dist [p] ← 0.0;

for val ← 2 to n-2 do

{

// obtain n-1 paths from p.

Choose q from the vertices that are not

visited (not in S) and with minimum distance

$Dist[q] = \min [Dist[i]]$ };

$S[q] \leftarrow 1$

/* Updates the ~~Dist~~ Distance values of the other nodes. */

for (all nodes r adjacent to q with

$S[r] = 0$) do

if ($Dist[r] > Dist[q] + cost[p,q]$))

then

$Dist[r] \leftarrow Dist[q] + Dist[p,q]$.

}

}

<u>Analysis</u>

Time complexity of the Algorithm

* The first for-loop clearly takes $O(n)$ time.

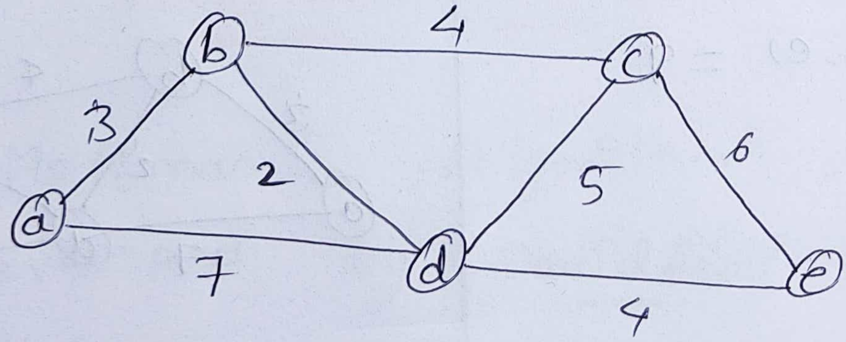
* Choosing q takes ~~at~~ O(n) time.

* The innermost for-loop for updating Dist iterates at most n times.

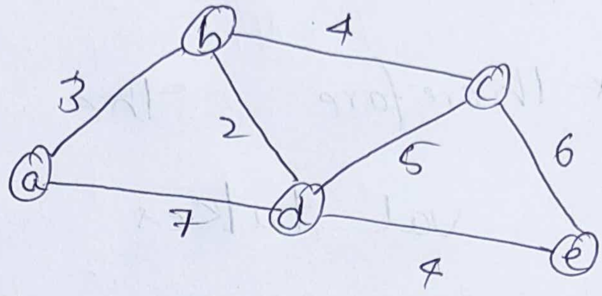
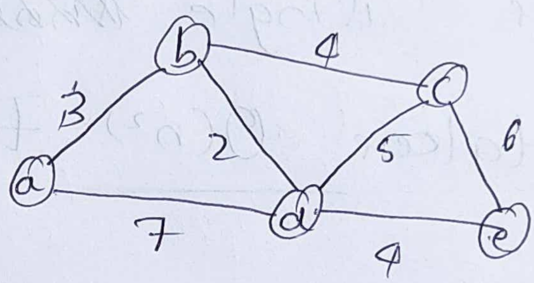
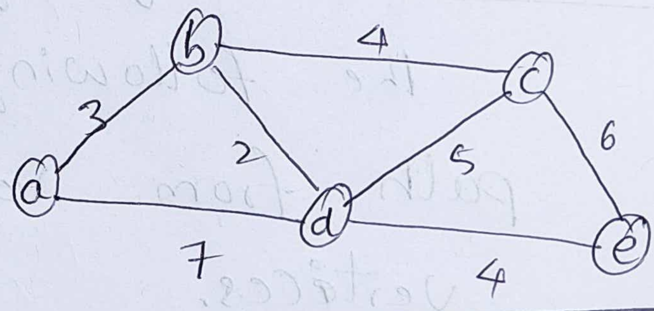
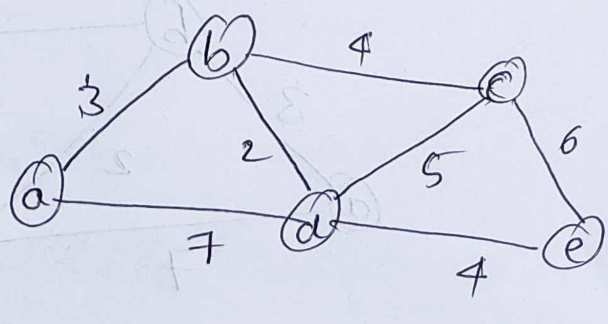
* Therefore the for-loop iterating over val takes O(n * n) time.

Thus single ~~short~~ source shortest path takes $\underline{O(n^2)}$ time.

<u>Example:-</u> Using greedy method trace the following graph to get shortest path from vertex a to all other vertices.



Solution :-

Selected Tree vertices	Distance towards other vertices	Path shown in graph
a-b	(a, b) = 3, (a,c) = ∞ (a,d) = 7, (a, e) = ∞	
a-b-d	(a,c) = 7, (a,d) = 5 (a, e) = ∞	
a-b-c	(a, c) = 7, (a, e) = 9	
a-b-d-e	(a, e) = 9	

Thus we obtain shotest paths from vertex a to all the remaining vertex ~~are~~. as.

From a to b : a-b Path length = 3

from a to d : a-b-d path length = 5

From a to c : a-b-c path length = 7

From a to e : a-b-d-e path length = 9

Chapter-04

Optimal Tree Problem

Topic-01

Huffman Trees & Codes

* The Huffman's Algorithm was developed by David ~~Huff~~ ~~Huffman~~
• Huffman when he was a ph.D

* The Algorithm is Basically a Coding Technique for ~~Coding~~ Encoding data.

* In Haffman's coding method, the data is inputted as a sequence of characters.

* From the table of Frequencies Haffman's tree is constructed.

* The Haffman's Tree is further Used for encoding each character, so that Binary encoding is obtained for given data.

* In Haffman's Tree is further is a specific method of Representing each symbol.

* This method produces a code in such a manner that no code word is prefix of some other code word.

Algorithm

The Greedy method is used to construct optimal prefix code called Huffman Code

* The Algorithm Builds a tree in Bottom Up manner.

* We can denote this tree By T.

Let, |C| be number of leaves

|C| -1 are number of Operations Required to Merge the nodes.

Q be the priority queue which can be used while constructing Binary heap.

Algorithm Haffman (C)
{

n = |C|

Q = C

for i ← 1 to n−1

do

{

temp ← get_node()

left[temp] Get_main (Q)

right[temp] = Get_ min (Q)

a = left[temp]

b = right[temp]

F[temp] ← f[a] + f[b]

insert (Q, temp)

}

return Get_min (Q)

Analysis

* The Haffman's Algorithm Requires O(log n) Time.

Example

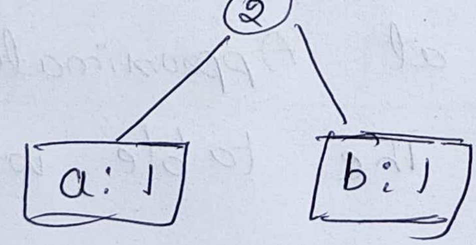
what is the optimal Huffman code for the following set of frequencies Based on first 8 fibonacci numbers.

a: 1, b: 1, c: 2, d: 3, e: 5, f: 8, g: 13, h: 21.

Solution :- we will Arrange the data in Ascending order of weight in a table.

a : 1	b : 1	c : 2	d : 3	e : 5	f : 8	g : 13	h : 21

Step-1 : we will combine first Two entries of the table and create a parent node.

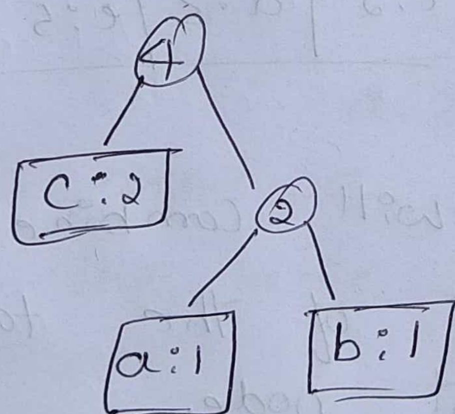


Remove the entries a & b from the Table & insert 2 at Appropriate position in the Table.

The Table will be as follows

c : 2	2	d : 3	e : 5	f : 8	g : 13	h : 21

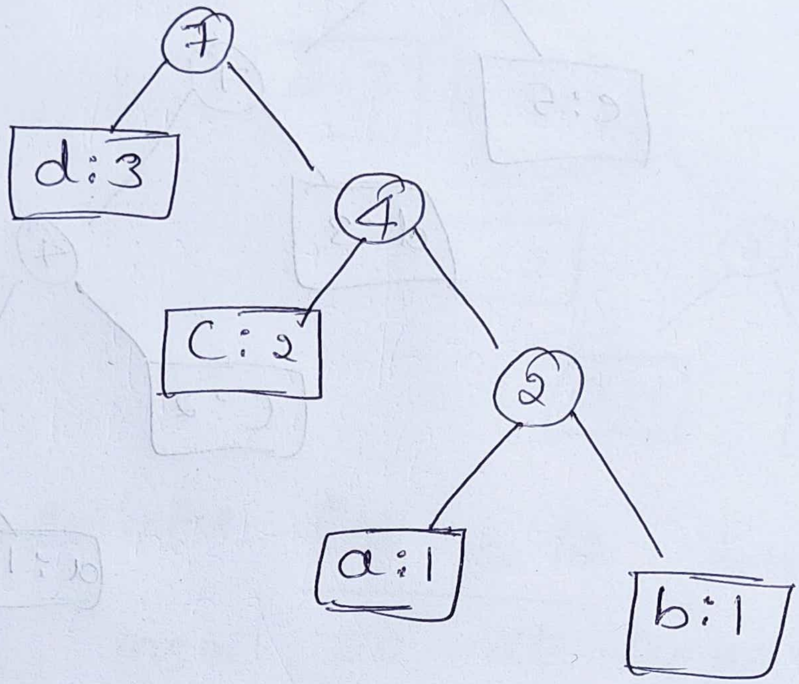
Step-2 :- we will combine first two entries of the table & Create parent node.



Remove the entries c : 2 & 2 from the table & insert 4 at Approximate position in the table. The table will be as follows

d : 3	4	e : 5	f : 8	g : 13	h : 21

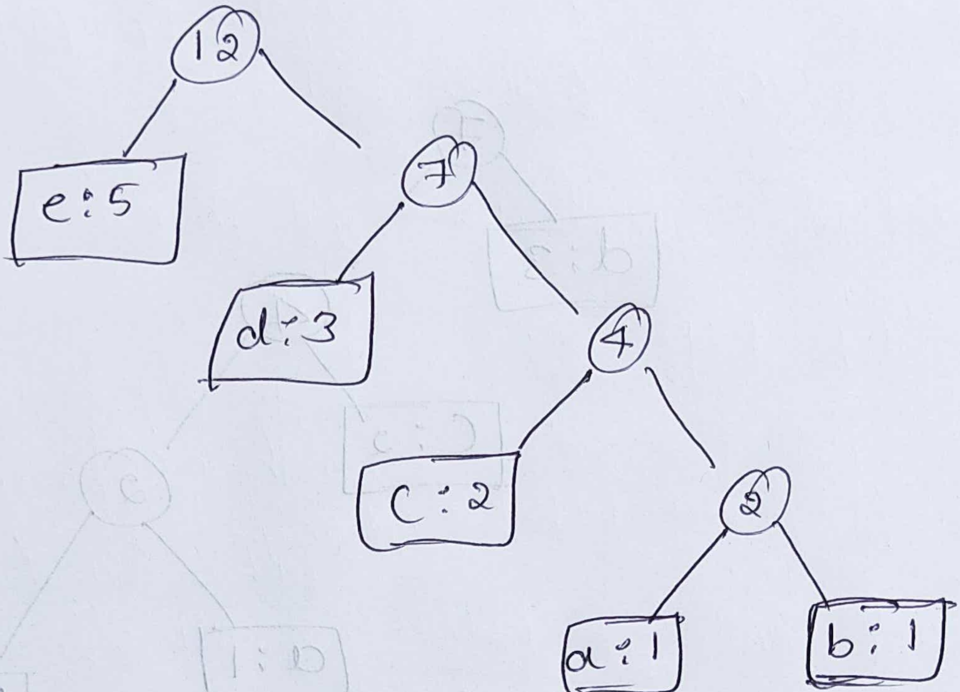
Step-3:- combine first two entries of the table & Create parent node.



Remove the Entries d:3 & 4 from the table & insert 7 at Appropriate position in the table. The table will be as follows.

e:5	7	f:8	g:13	h:21

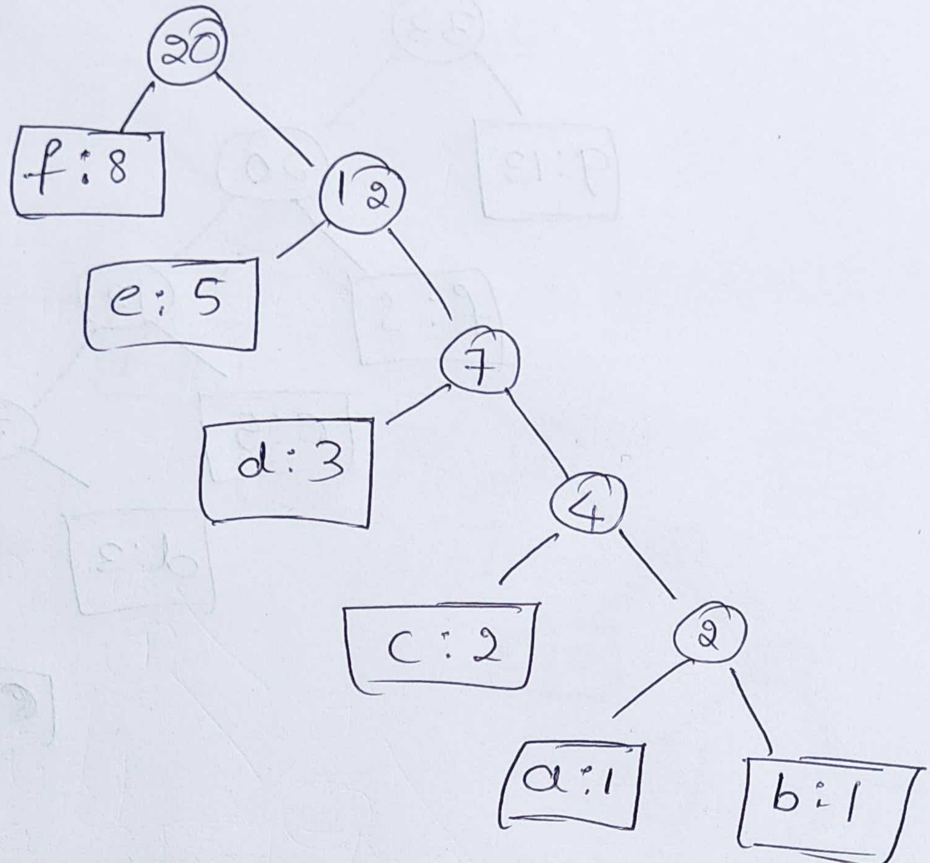
Step-4 :- Combine first two entries of the table & Create parent node.



Remove the Entries e:5 and 7 from the table & insert 12 at approximate position in table.

f:8	12	g:13	h:21

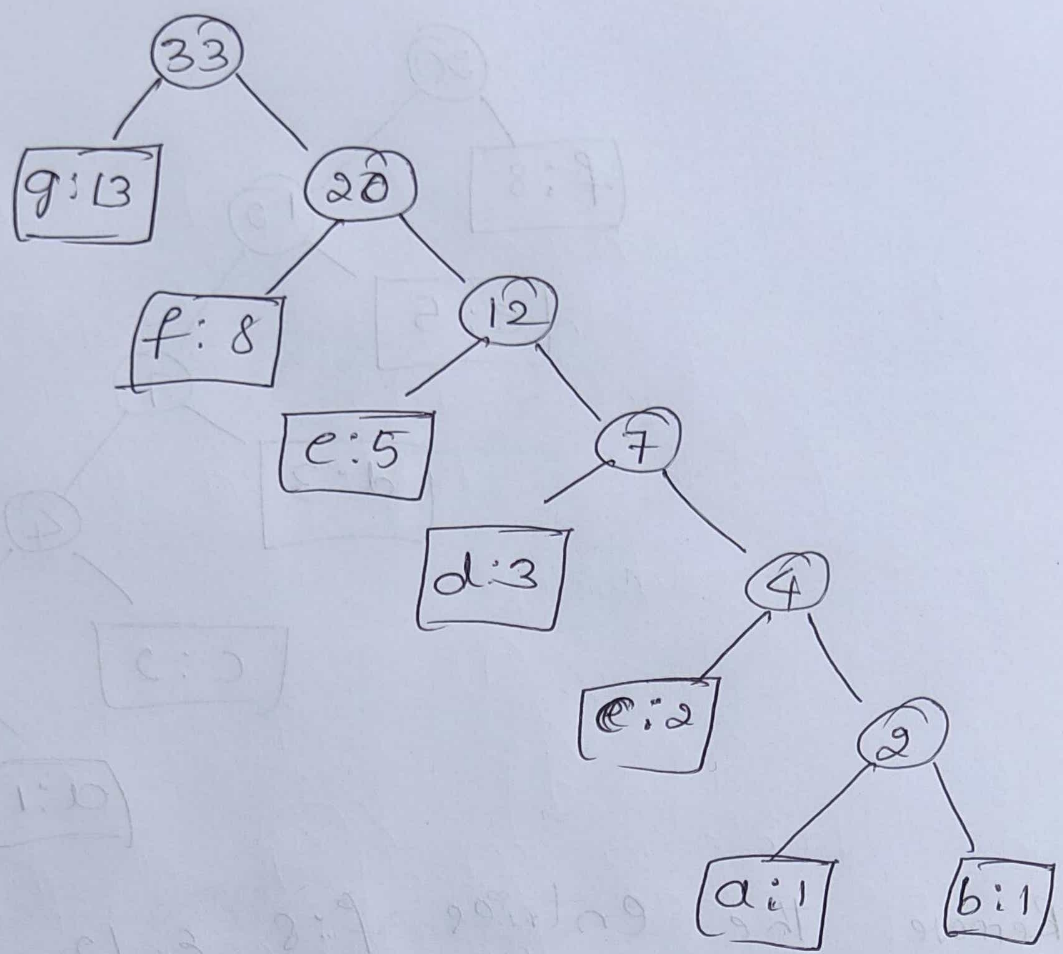
Step-5 :- Combine first two entries of the table & Create parent node.



Remove the entries f:8 & 12 from the table & insert 20 at Appropriate position in the table.

g: 13	20	h: 21

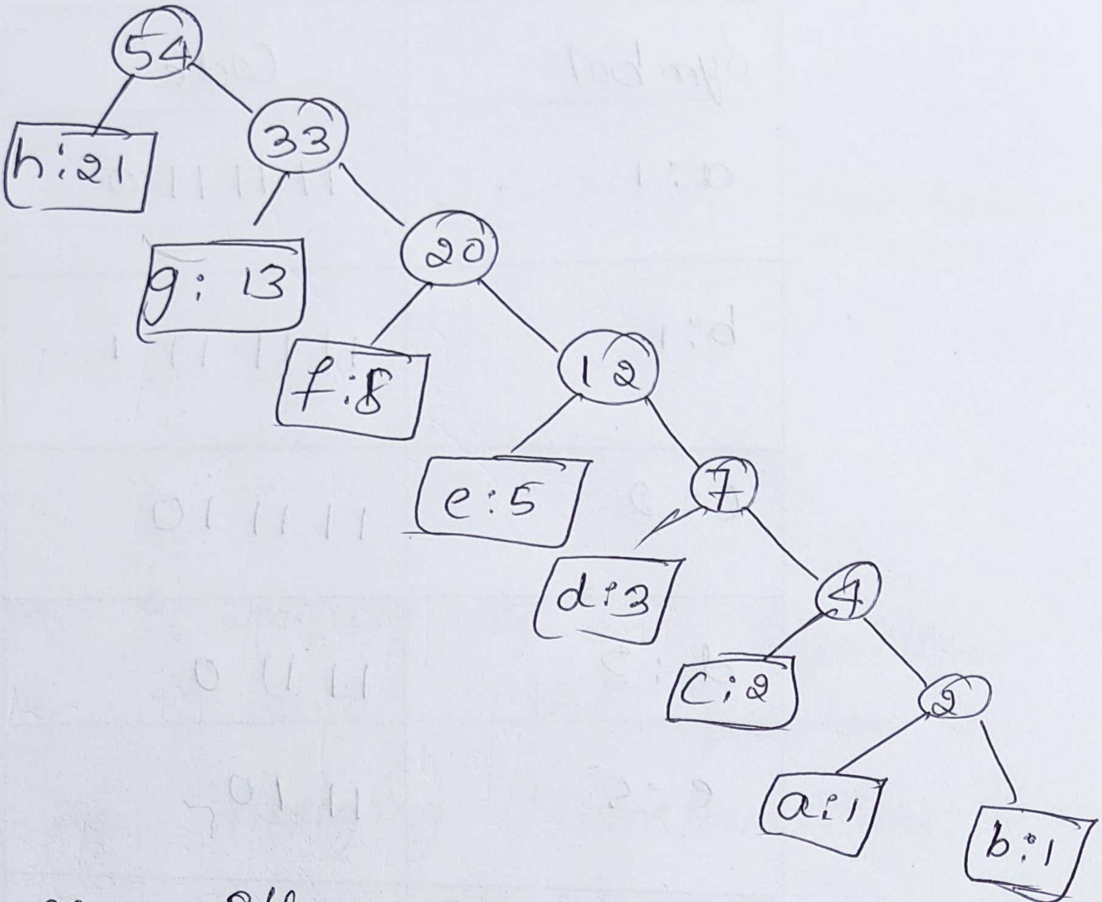
Step-6:- Combine first two entries of the table & Create parent node.



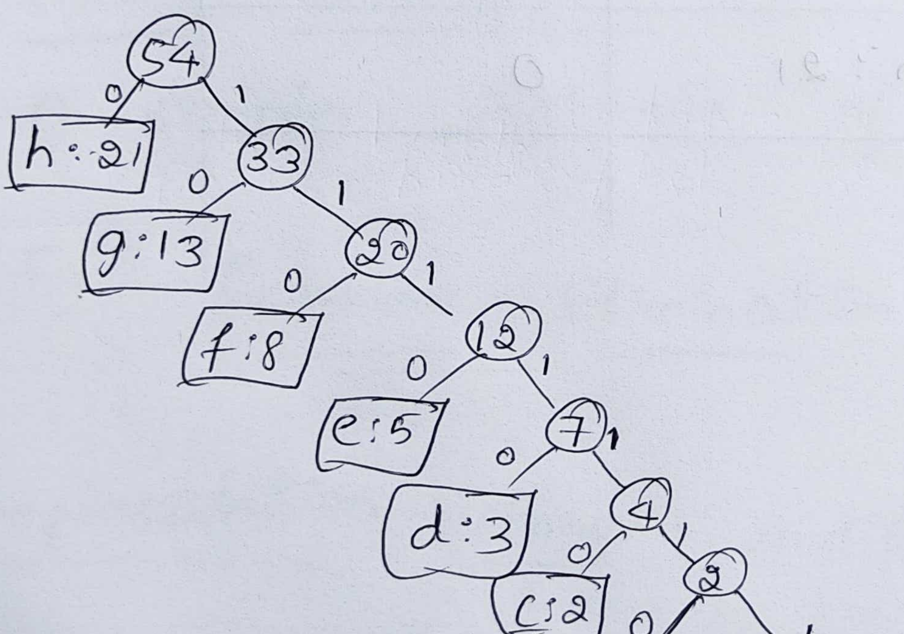
Remove the entries g:13 & 20 from the table & insert 33 at appropriate position in the table. The table will then be as follows.

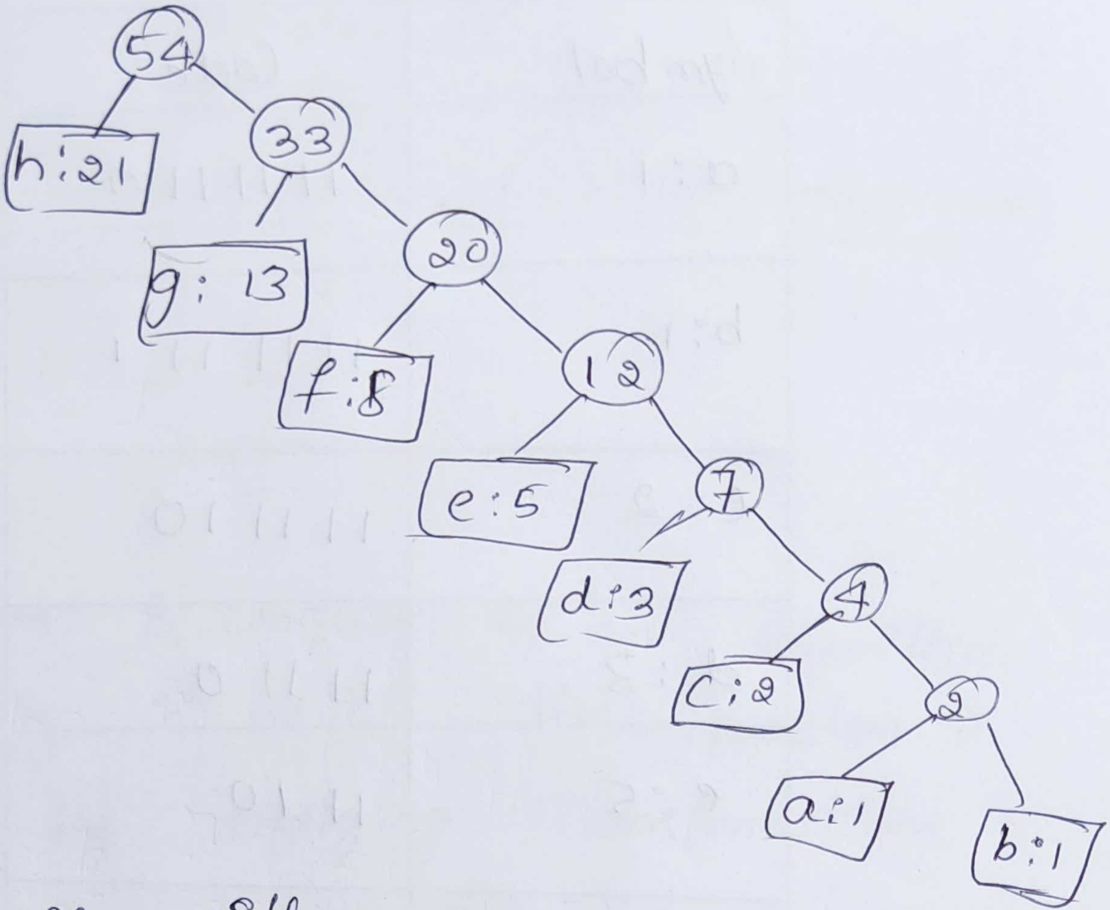
h:21	33

Step – 7 :- combine first two entries of the table of the table & create parent node.

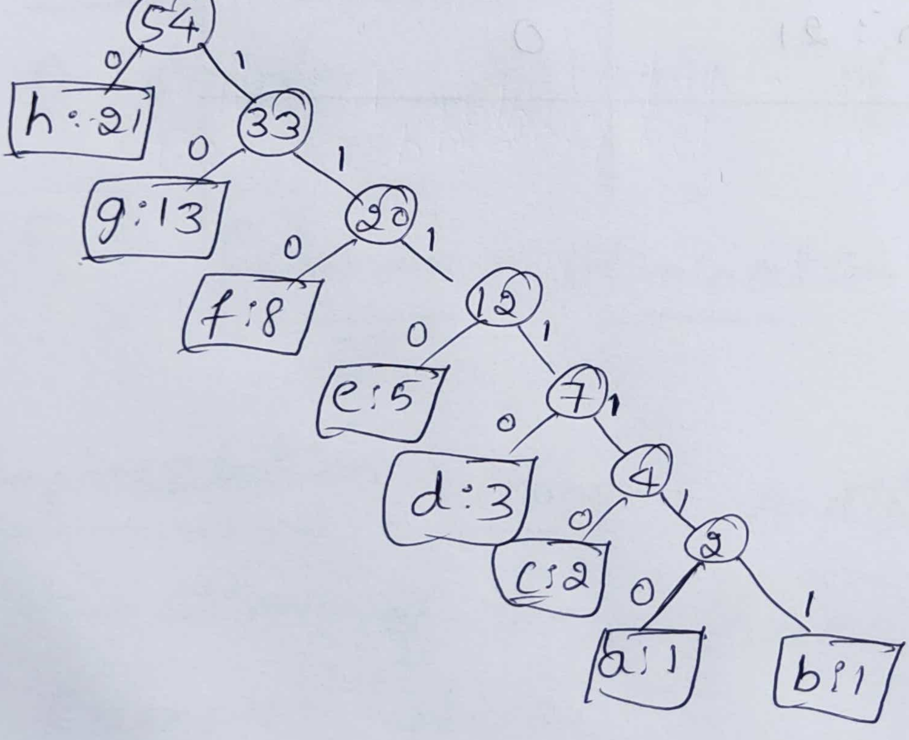


Step-8:- We will encode the Above Created Huffman's Code. The left Branch is numbered as 0 & right Branch is numbered as 1.





Step-8:- we will encode the Above Created Huffman's Code. The left Branch is numbered as 0 & right Branch is numbered as 1.



Symbol	Code
a:1	1111110
b:1	1111111
c: 2	111110
d : 3	11110
e :5	1110
f : 8	110
g : 13	10
h : 21	0

Chapter-05

Transform & Conquer Approach

Topic-01

Introduction

* Transform & Conquer is an Algorithm strategy in which the problem is solved by Applying Transformations.

There are three variations of this Approach

1. Instance simplification:

   A simpler instance of the same problem.

   Ex:- Gaussian Elimination method & AVL Tree

2. Representation Change: A different Representation of the same problem.

   Examples. Heaps.

3. Problem Reduction

An instance of the different problem.

Ex:- Counting ~~the~~ paths in a graph,

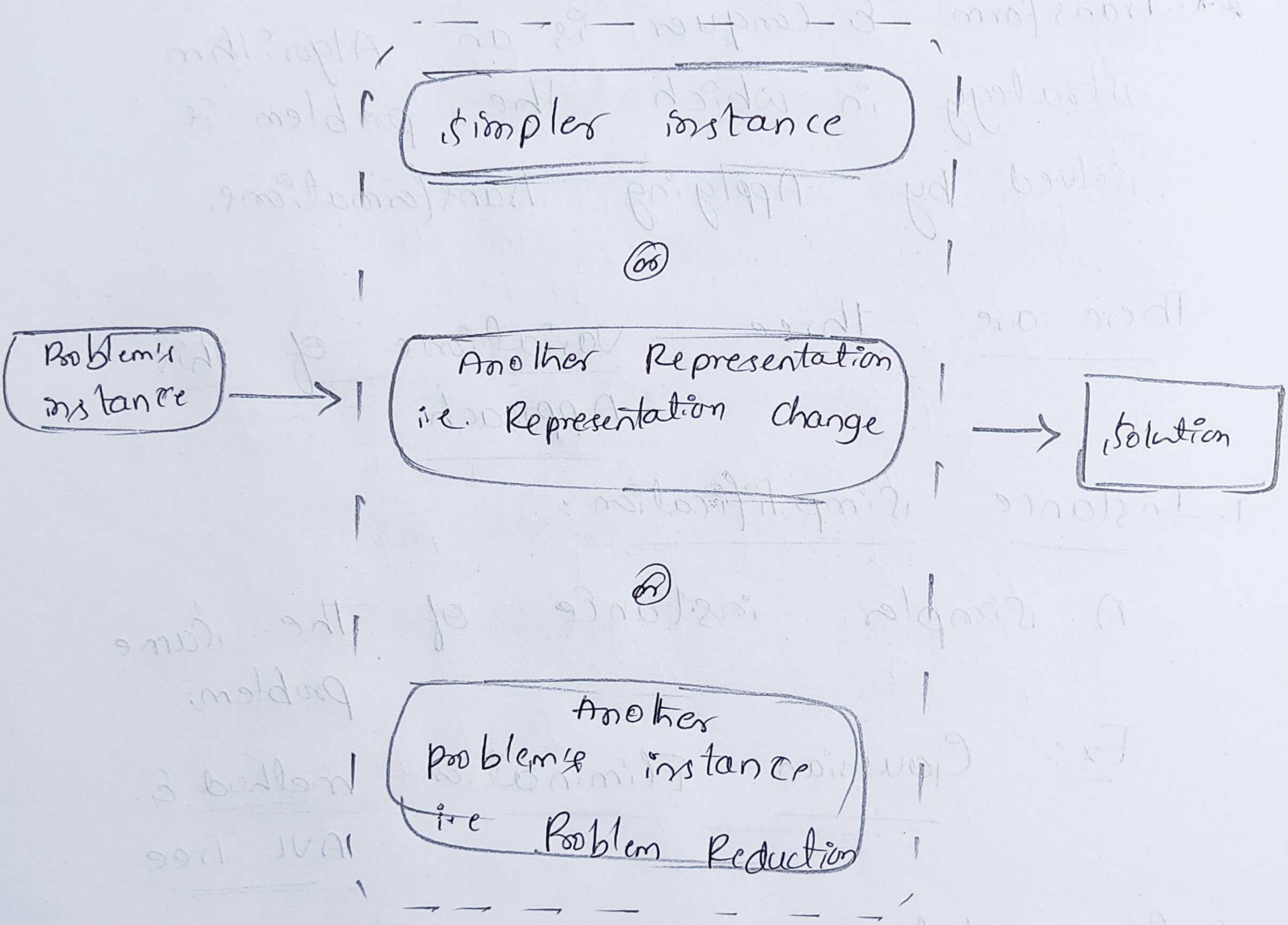


fig:- Transform & Conquer

Topic-02

Heaps and Heap Sort

Heap

Heap is a compute Binary tree (or) a almost complete Binary tree in which Every parent Node be either greater (or) lesser than its child nodes.

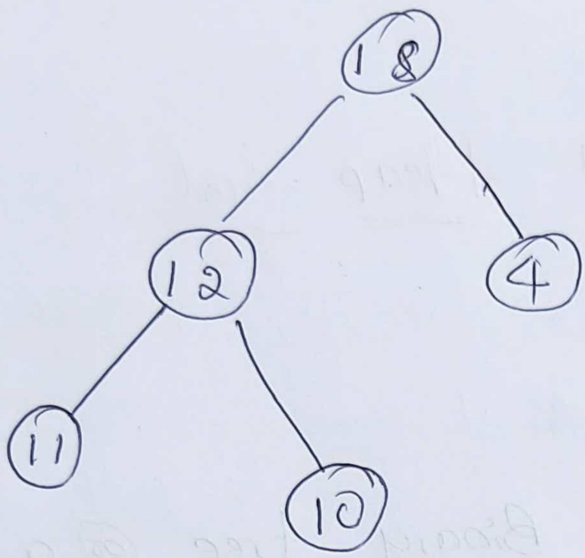
Types of Heap

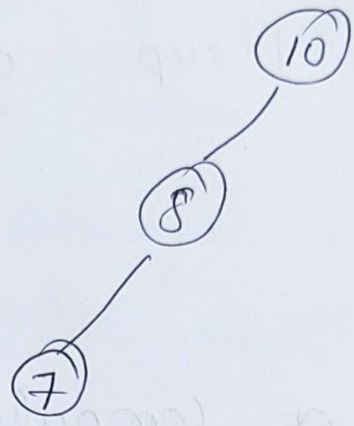
1. Max Heap
2. Min Heap.

1. 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 Nodes.

Ex:-

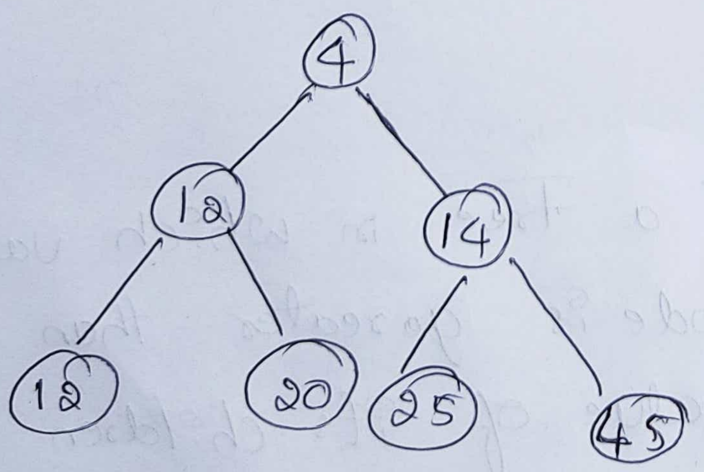


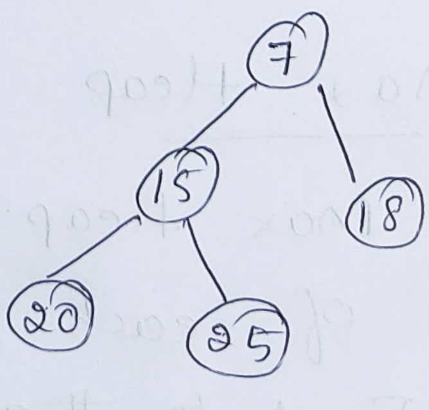


2. Min Heap

A Min Heap is a Tree in which value of each node is less than or Equal to value of its children Node.

Ex:-

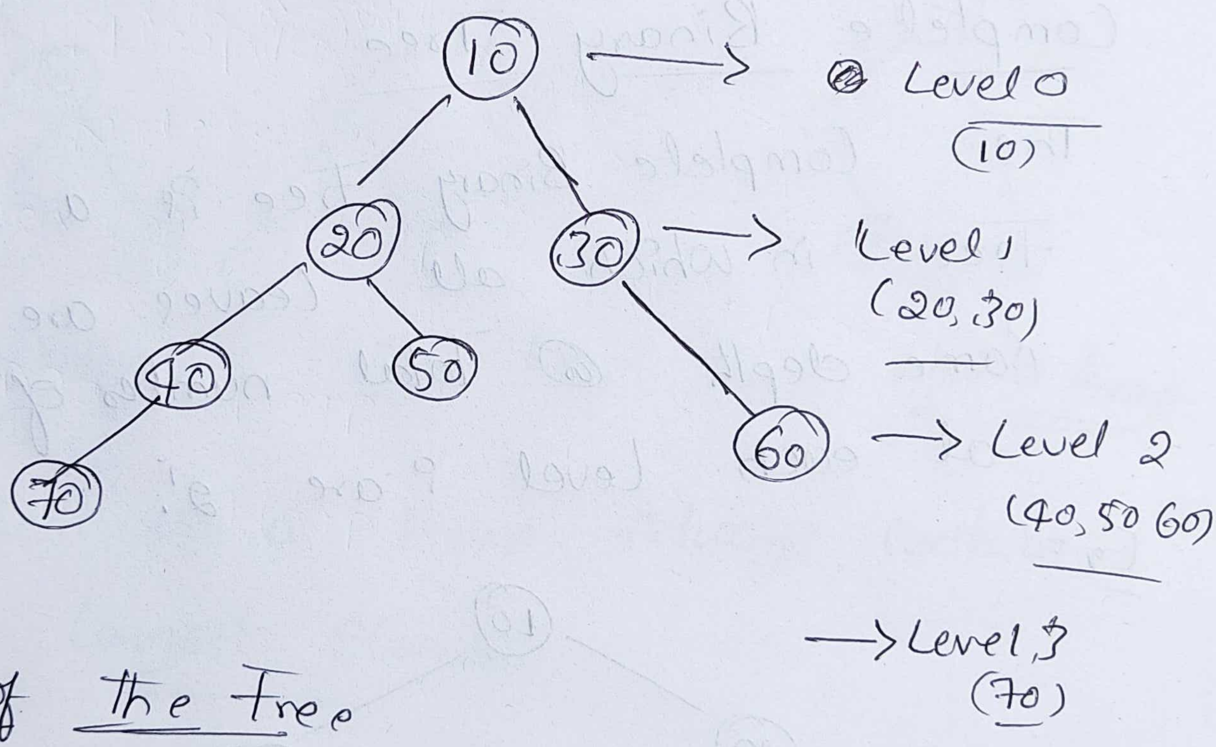




Level of Binary Tree

* The Root of Tree is always at Level 0.
* Any Node is Always at a Level one more than its parent nodes Level.

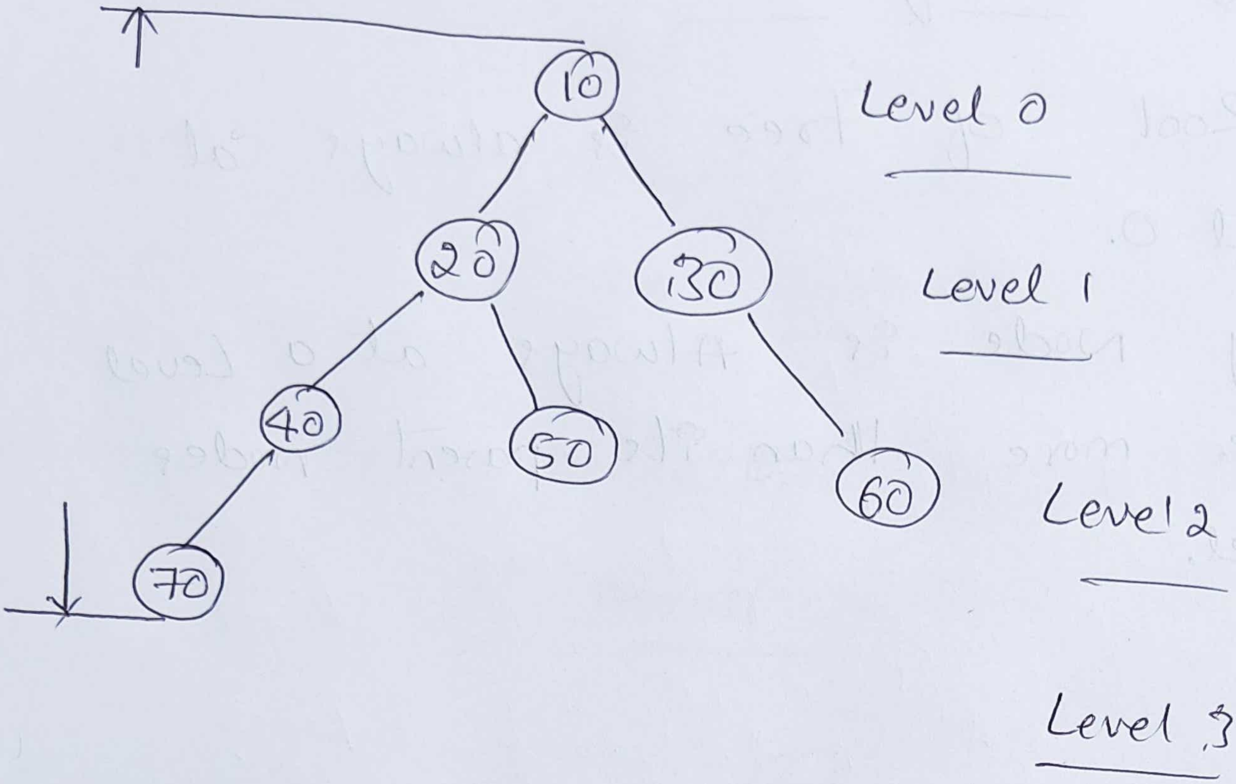
Ex:-



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.

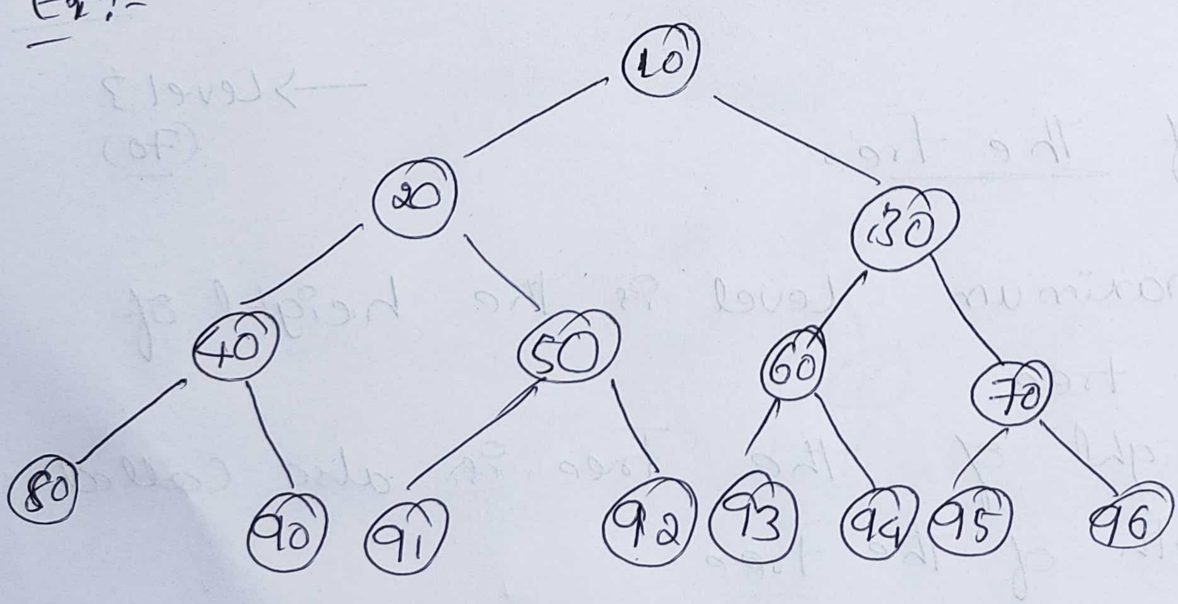
Ex:-



Complete Binary Tree

The Complete Binary Tree is a Binary Tree in which all Leaves are at the same depth & Total numbers of nodes at each Level i are 2^i.

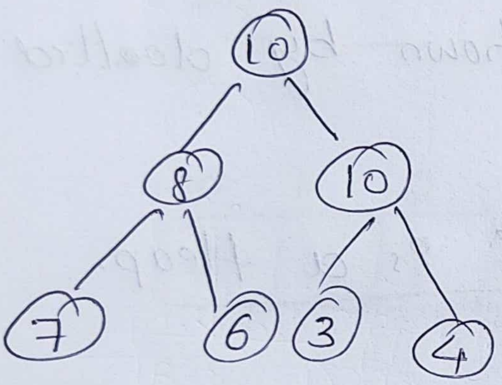
Ex:-



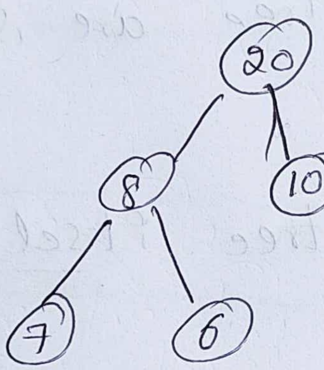
Properties of Heap

There are some Important Properties that should be followed by heap

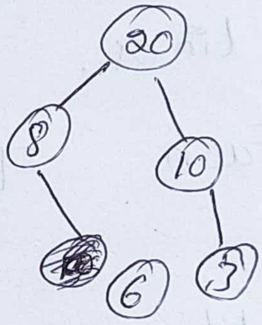
1. There should be either "Complete Binary Tree" or "Almost Complete Binary Tree"



is a heap

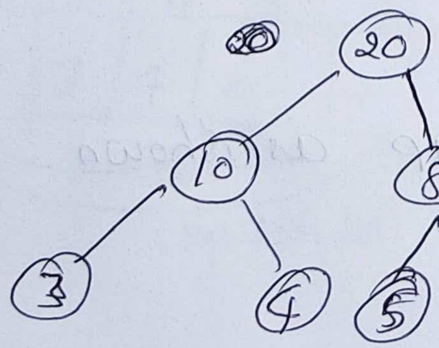


is a heap



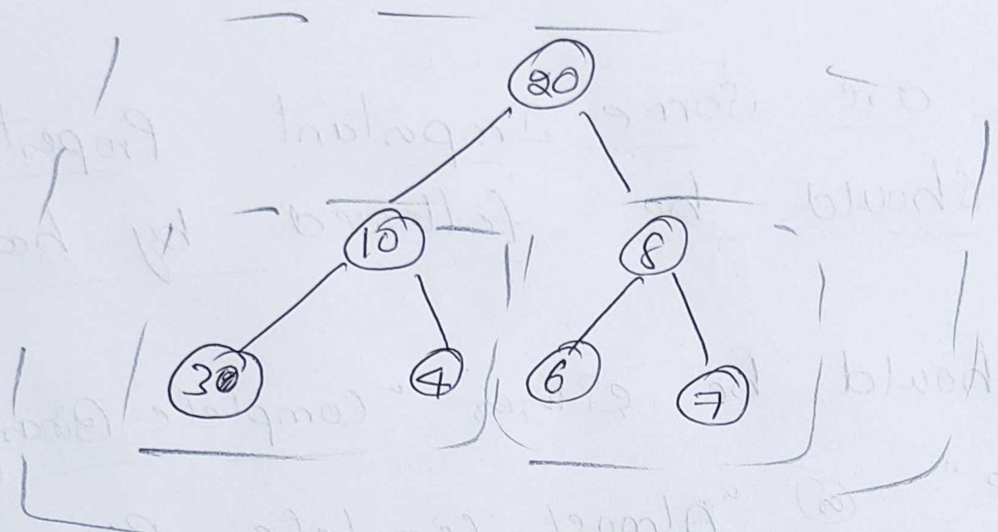
Not a heap

2. The Root of a heap Always Contains its Largest Element



This is heap Because 20 is largest Among all nodes values.

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



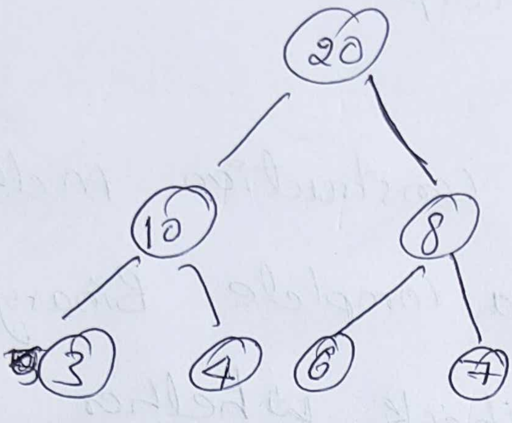
* The Tree & subtree are shown by dotted lines.

* Each Tree/subtree itself is a Heap.

4. Heap can be Implemented as Array by Recording its Elements in the top-down & left to right fashion.

Example

Consider Heap as shown Below.



⇓

0		
20		

Root is stored at A[0].

0	1	2	
20	10	8	

0	1	2	3	4	
20	10	8	3	4	

left child
right child

Left child of 10 is at A[3] & right child of 10 is at A[4]

0	1	2	3	4	5	6
20	10	8	3	4	6	7

left child
right child

Thus array can Represent Complete Heap

construction of Heap

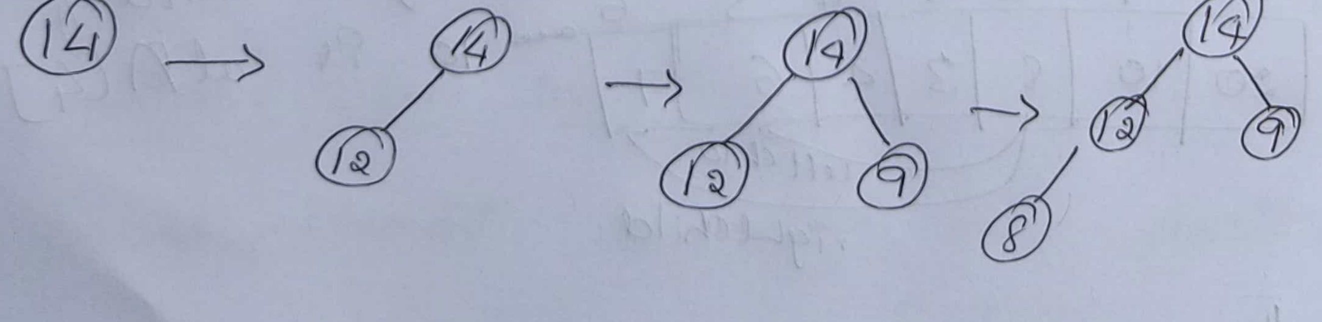
* In the heap. construction method we go on Building a complete Binary Tree & Each time check whether the parent is maximum or not.

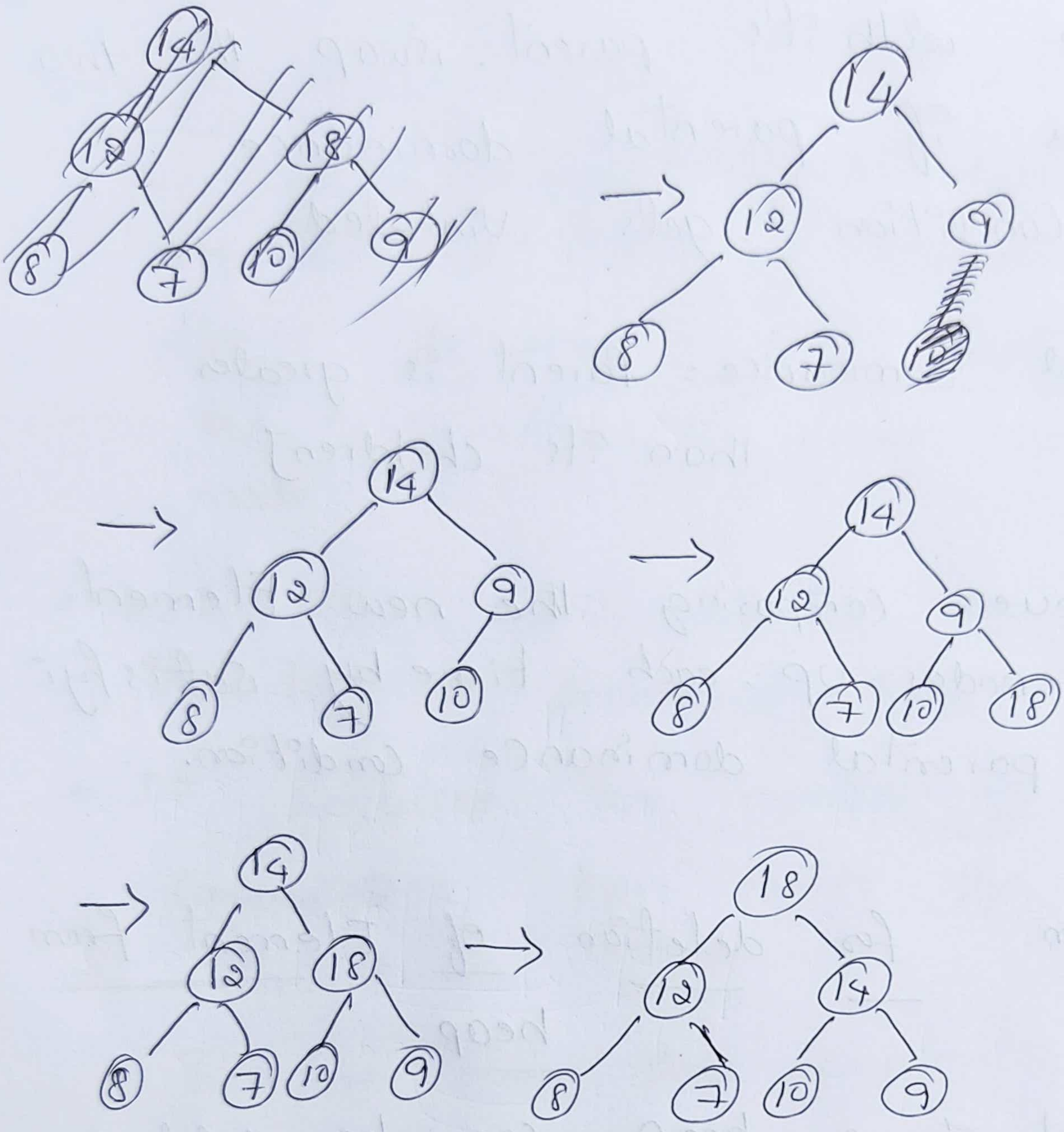
* If parent is less than any of it child node then swapping is performed to make the parent larger than its parent.

Example

construct heap for 14, 12, 9, 8, 7, 10, 18

solution:- we can construct heap with Repeated insertion of Element.





Heap is formed.

Insertion & Deletion Operations for Heap

Algorithm for insertion of Element

1. Insert Element at the last position in heap.

2. Compare with it's parent, swap the two nodes if parental dominance ~~cond~~ Condition gets violated

[Parental dominance = Parent is greater than its children]

3. Continue comparing the new Element with nodes up, each time by satisfying parental dominance condition.

Algorithm for deletion of Element from heap

The Root of a heap can be deleted & the fixed up can be as follows

1. Exchange the root with last leaf.

2. Decrease the heap's size by 1

3. Heapify the smaller tree Using Bottom up Construction Algorithm.

Analysis

* The Basic operation in deletion Algorithm is the key comparision that should be made to "heapify" the 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 ~~Com~~ Complexity of deletion of O(logn).

Heap Sort

* Heap Sort is a Sorting method ~~are~~ discovered by J. W. J Williams.

* It works in Two Stages

1. Heap Construction: first construct a heap for given numbers.

2. Deletion of Maximum Key:-

* Delete Root Key always for (n-1) times to Remaining heap.

* Hence we ~~can~~ will get the Elements in decreasing order.

For descending order max heap & for ascending order min heap is Created.

Example

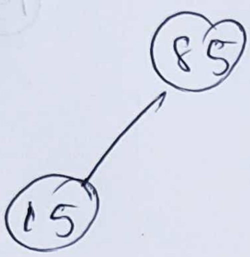
Sort the following data in descending order using heap sort 85, 15, 25, 95, 145, 55, 165, 75. Show all steps.

Solution :- To sort the Elements in descending order, we must Create max heap structure.

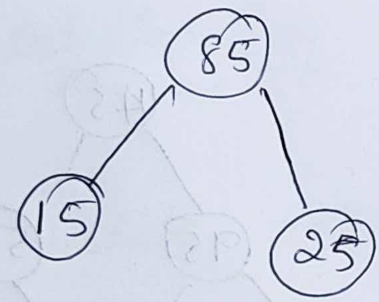
Stage I : Creation of Max heap.

Step-1 : Insert 85

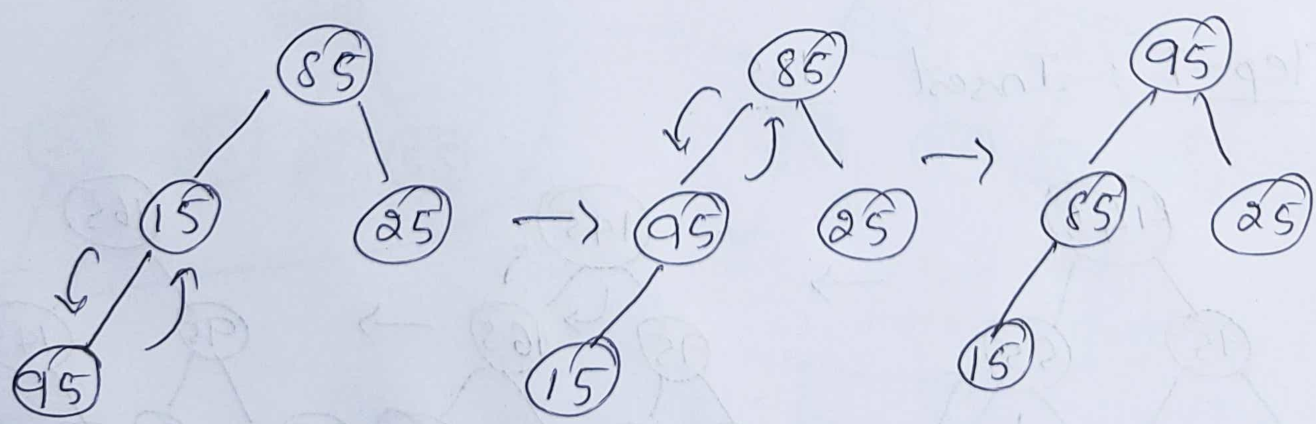
Step-2 :-



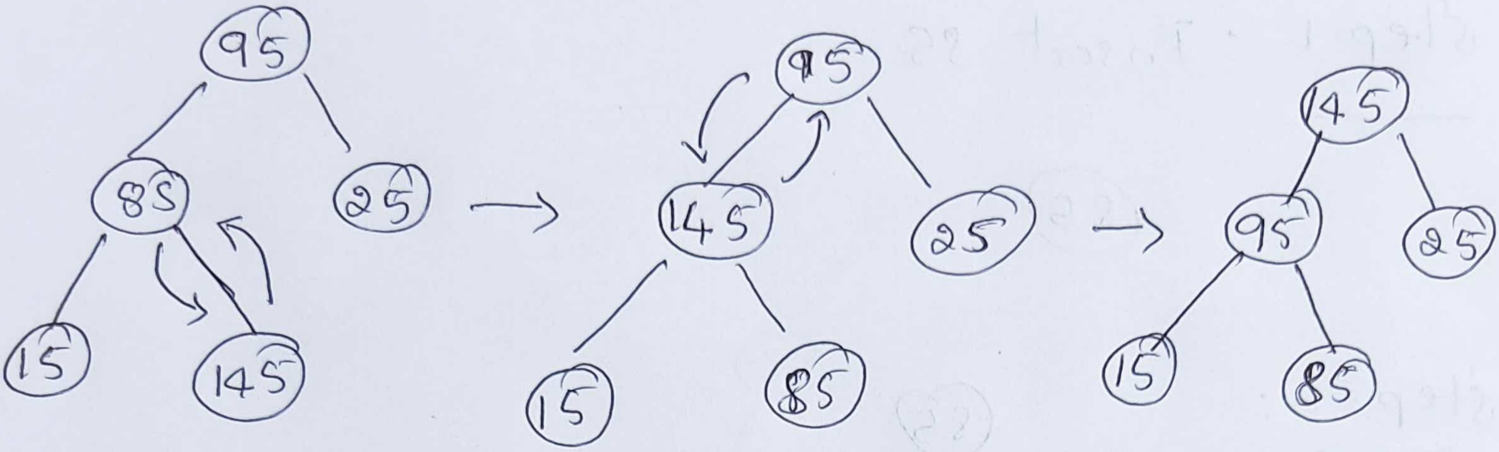
Step-3 :-



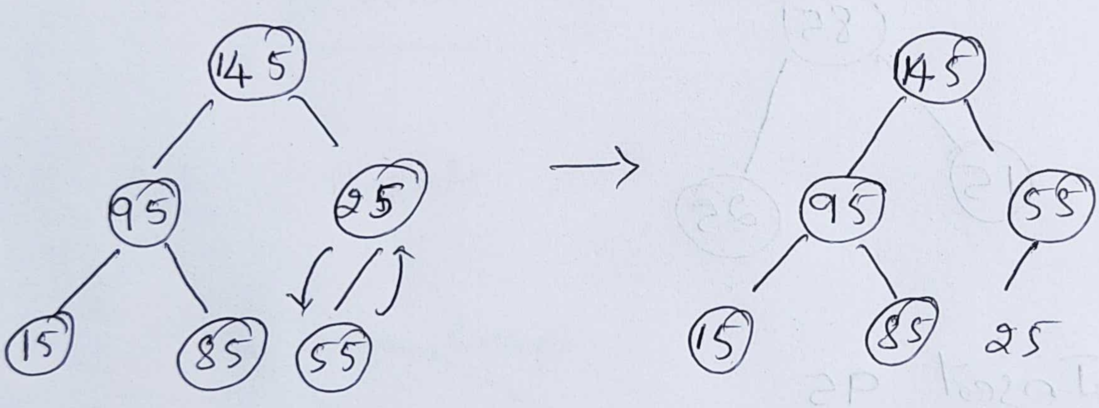
Step-4 :- Insert 95



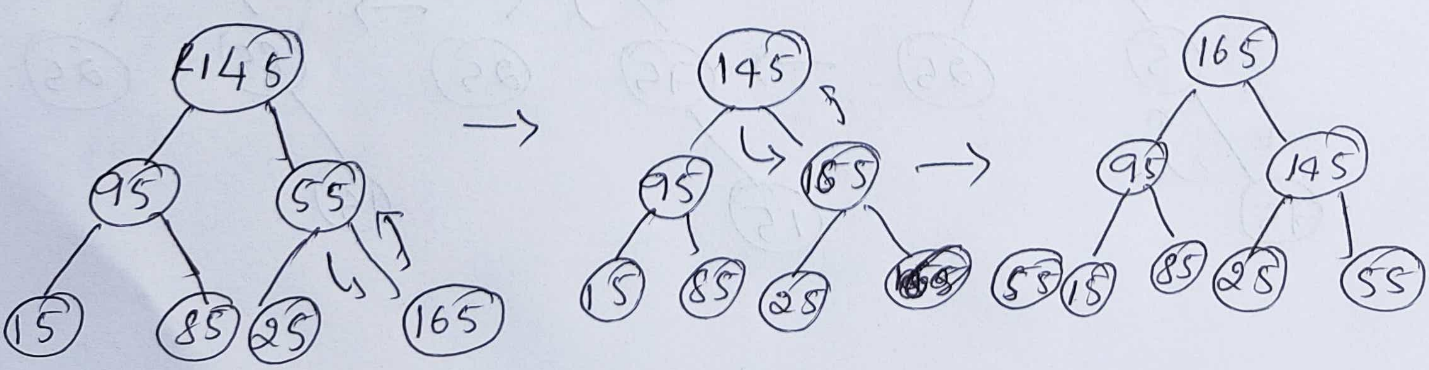
Step-5 : Insert 145



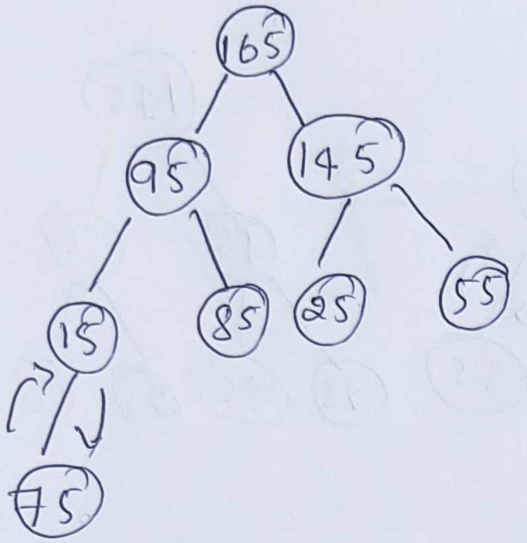
Step-6 : Insert 55

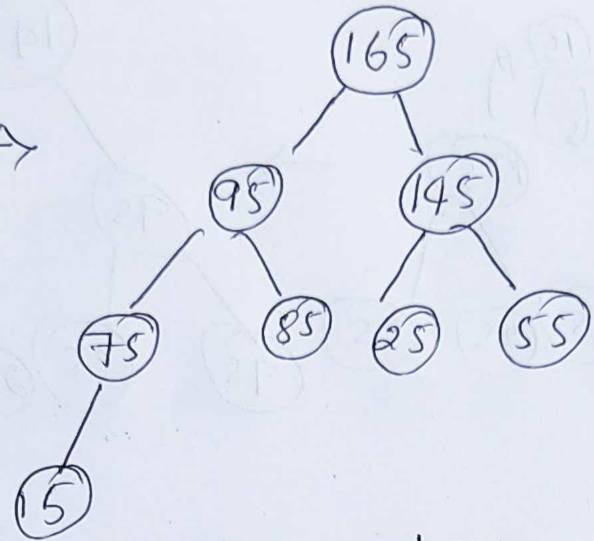


Step-7 : Insert 165



Step-8 : Insert 75

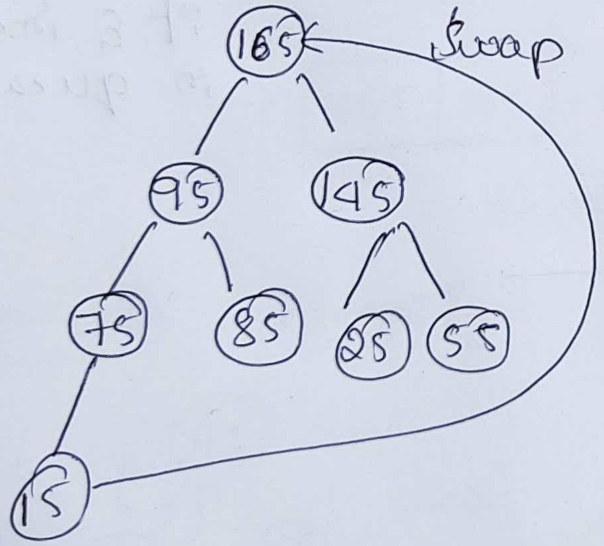


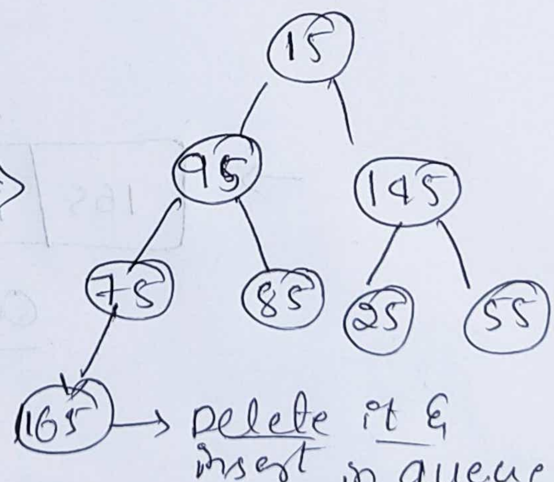


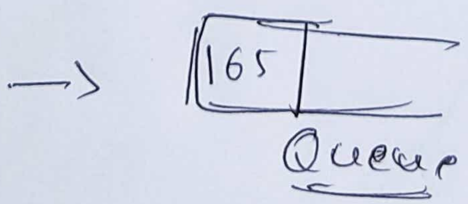
max heap

Stage-II : Deletion of Root

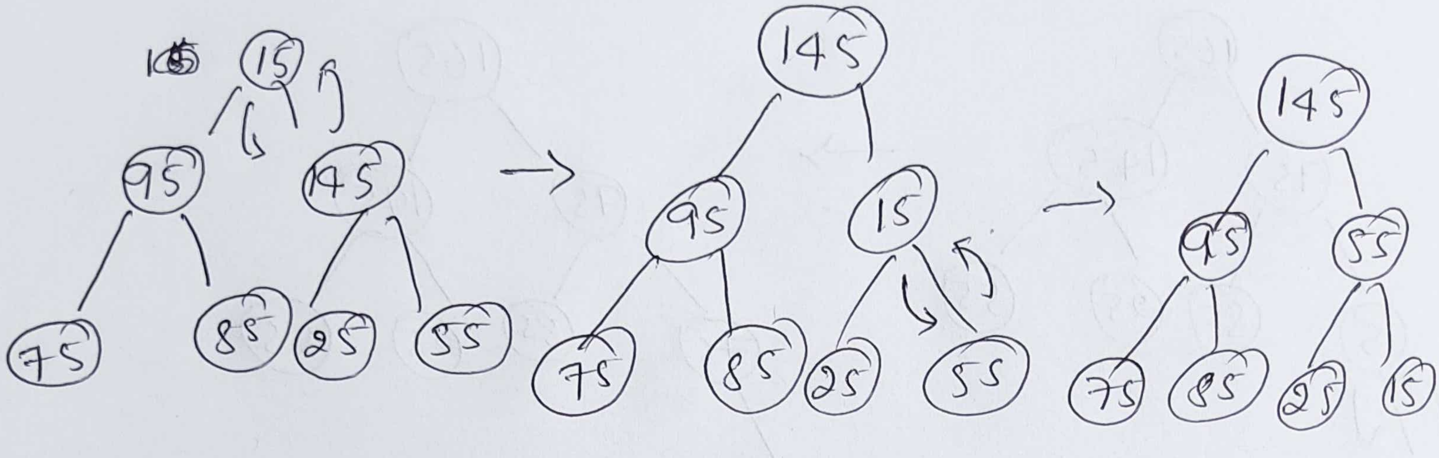
Step-1a :- Delete root node.



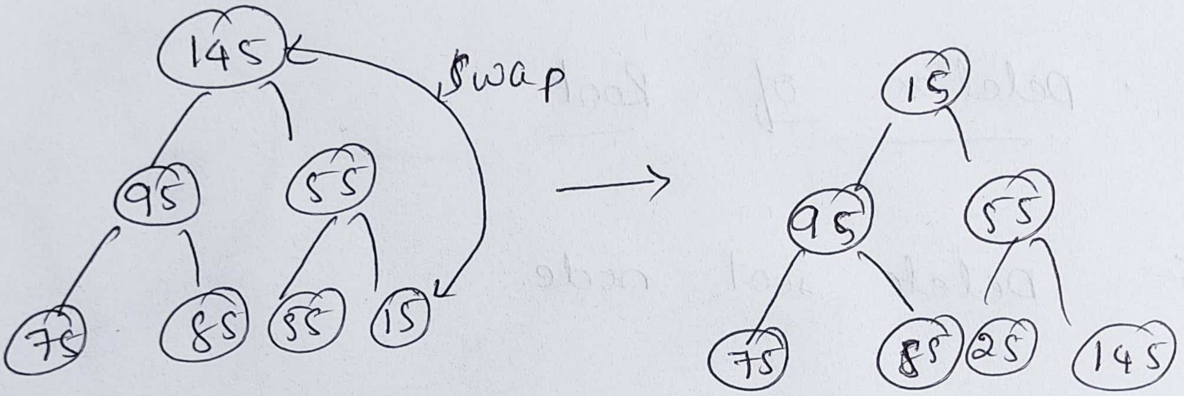




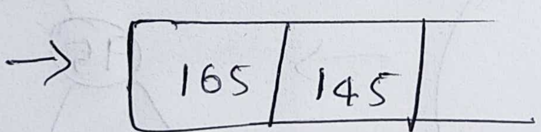
Step-1b : Heapify



Step-2a : Delete root node

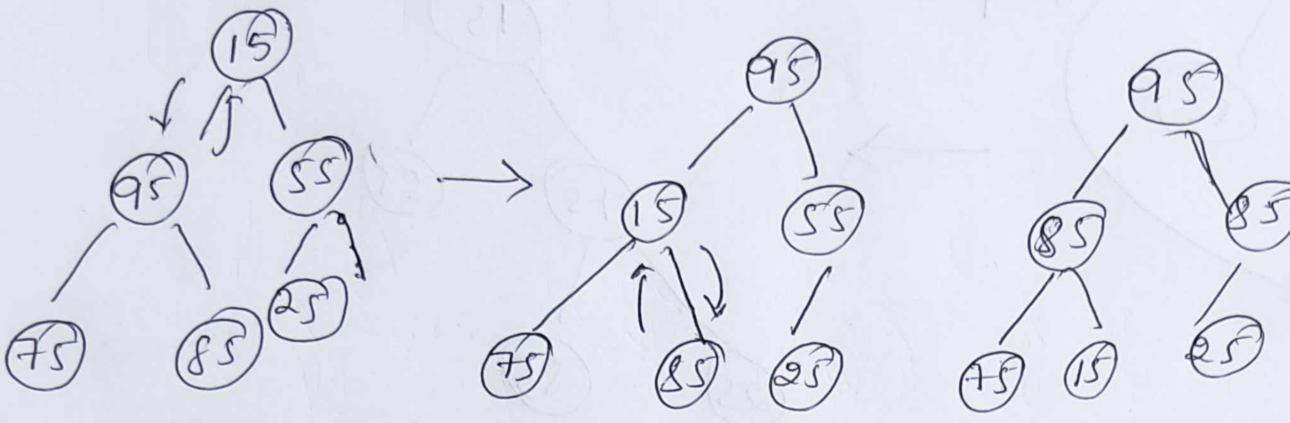


← Delete it & insert in queue.

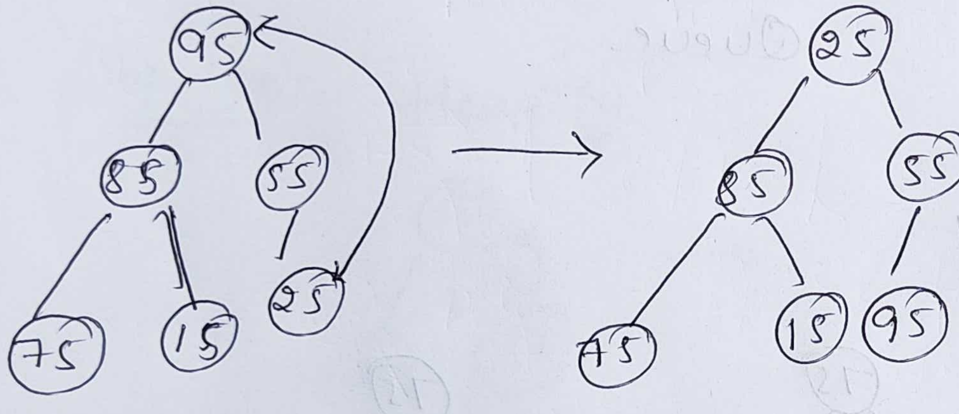


Queue

Step-2b: Heapify



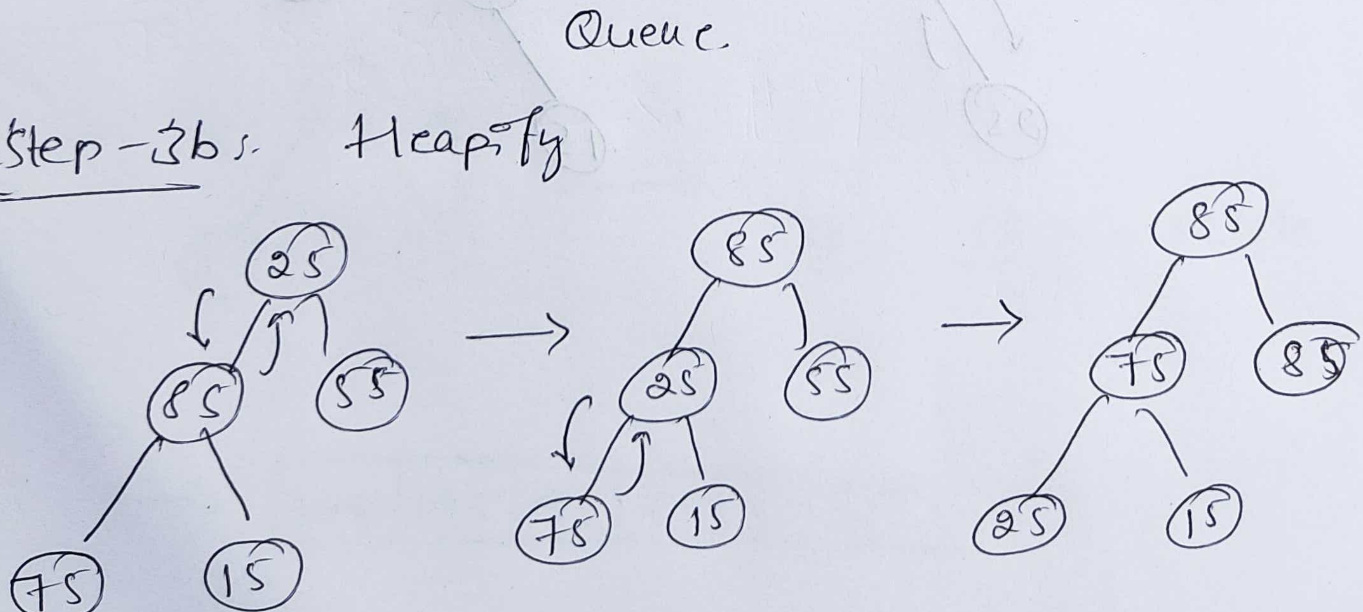
Step-3a: Delete of root node



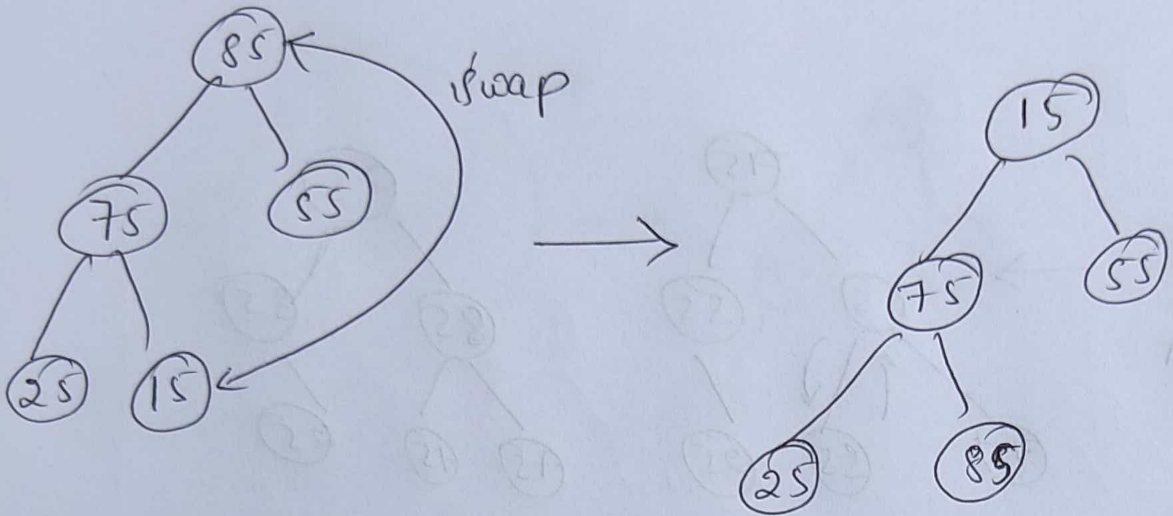
165	145	95	

Queue

Step-3b: Heapify



Step-4a:- Delete of root node

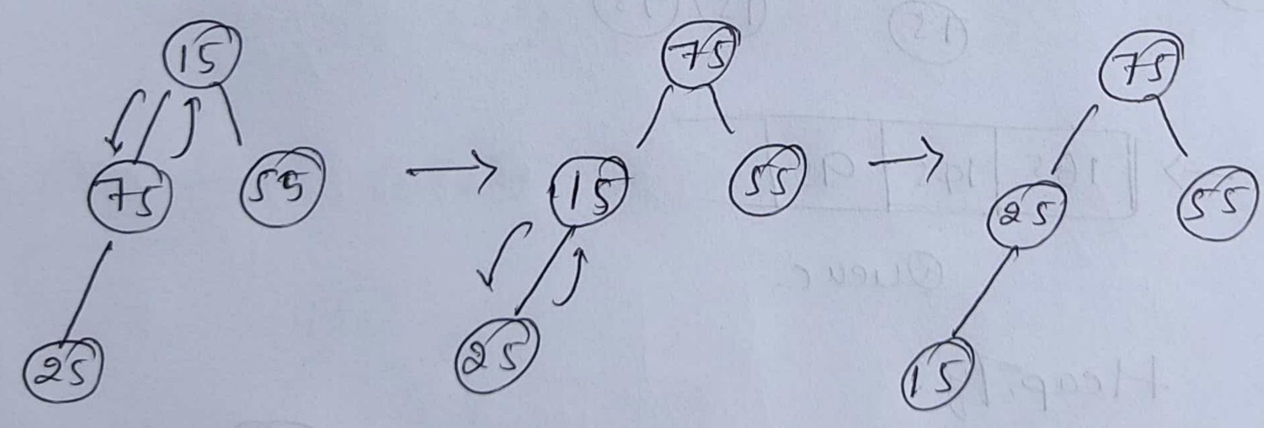


→

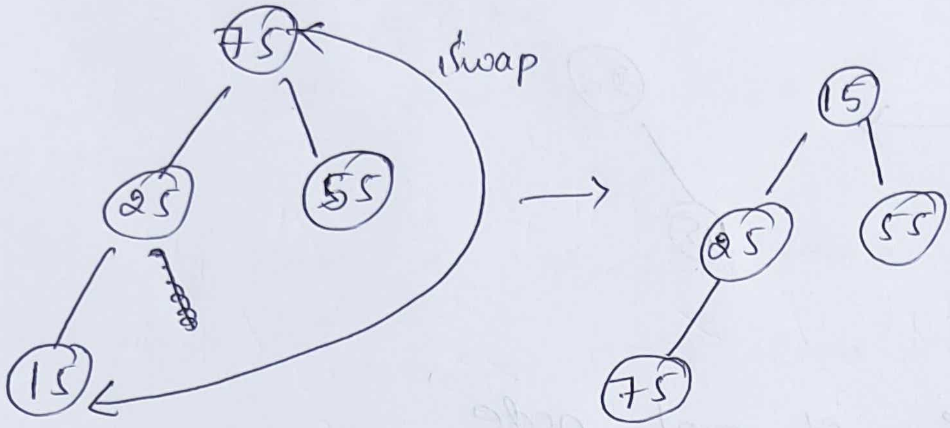
165	145	95	85	

Queue

Step-4b:- Heapify



Step-5a: Delete of root node

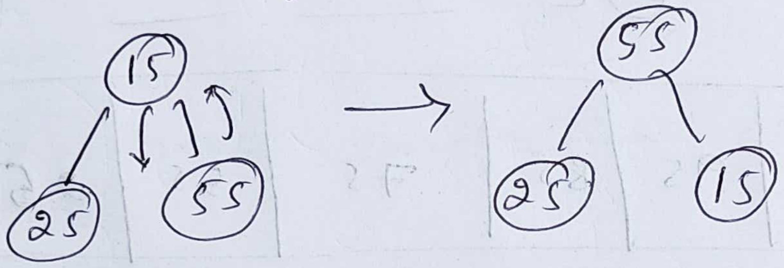


→

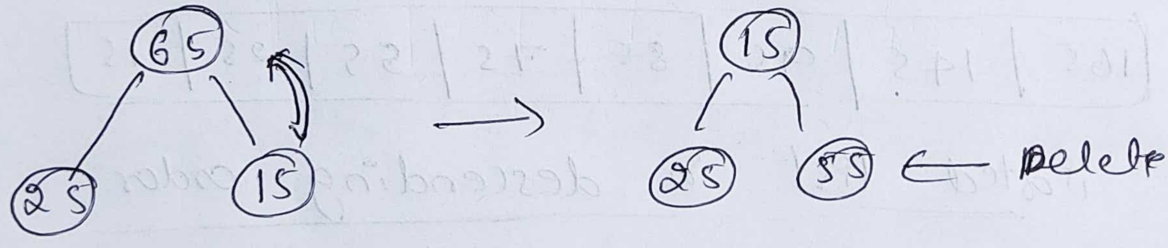
165	145	95	85	75

Queue

Step-5b: Heapify



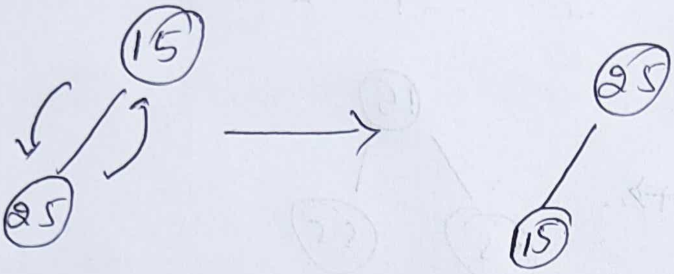
Step-6a: Delete of root node



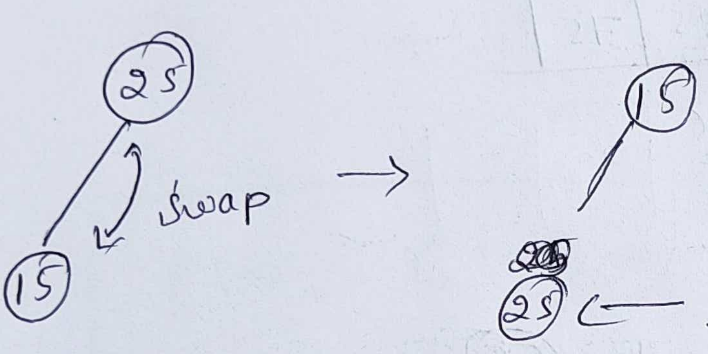
→

165	145	95	85	75	55

Step-6b:- Heapify



Step-7:- Delete of root node



Delete

165	145	95	85	75	55	25

Step-8:- Delete & insert in array

165	145	95	85	75	55	25	15

sorted list in descending order

Analysis

* As this Algorithm works in two stages we get Running time.

Running time for heap sort = Running time required by heap construction

+ Running time Required by deletion of the root key.

$$\boxed{C(n) = C_1(n) + C_2(n)}$$

As the heap construction Requires $O(n)$ time

$$\boxed{C_1(n) = O(n)}$$

Time complexity of heapsort is $\Theta(n \log n)$ in Both worst & Average case