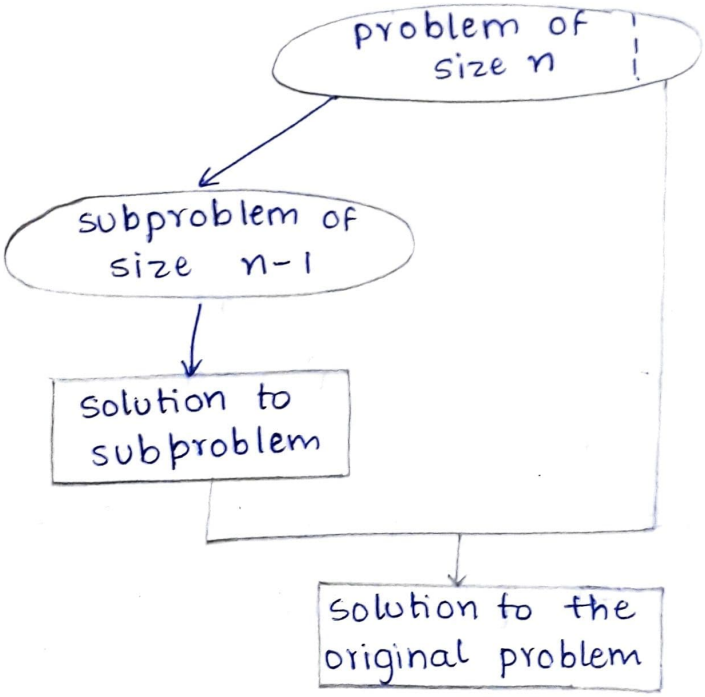
UNIT - III

DECREASE AND CONQUER

The decrease and conquer technique is based on exploiting the relationship between solution to a given instance of a problem and solution to smaller instance of the same problem. Once a relationship is established, it can be solved either by top down (recursively) or bottom up (non recursively) approach.

The 3 major variations of decrease and conquer:

i) Decrease by a constant (one):



Here, the size of an instance is reduced by some constant on each iteration of algorithm, typically by 1.

Eg: Consider the problem of finding a^n.

The solution can be obtained with an instance of size n-1 as.

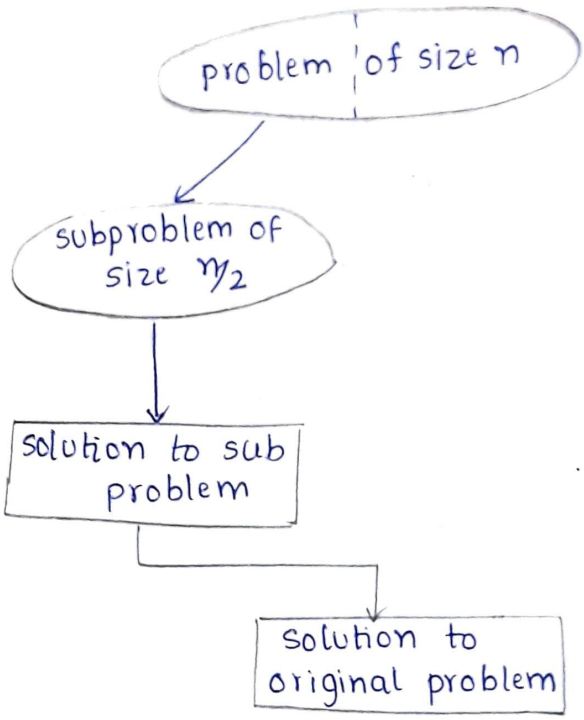
$$a^n = a^{n-1} \cdot a$$

This can be solved using recursion as.

$$F(n) = \begin{cases} a & \text{if } n = 1 \\ F(n-1) \cdot a & \text{if } n > 1 \end{cases}$$

2) Decrease-by-a-constant-factor

Here, problem is solved by reducing the problem size by same constant factor on each iteration of algorithm. In most applications, it is 2.



Eg: Consider example of computing a^n.

It can be computed as $a^n = (a^{n/2})^2$ if n is even and $(a^{(n-1)/2})^2 \cdot a$ when is odd.

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even \& +ve} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd \& } > 1 \\ a & \text{if } n = 1 \end{cases}$$

3) Variable size decrease:

Here reduction pattern varies from one iteration to another for an algorithm.

Eg: Euclid's algorithm to find GCD(m,n):

$$\gcd(m,n) = \gcd(n, m\%n)$$

Insertion Sort:

This sorting technique is an example for decrease by one technique, where we assume that smaller problem of sorting array A[0...n-2] has already been solved to give us sorted array A[0...n-1]

Here, we scan the sorted subarray from right to left until the first element smaller than or equal to A[n-1] is encountered and then insert A[n-1] right after that element.

Algorithm InsertionSort(A[0..n-1])

// Sorts a given array by insertion sort

// Input: Array A[0..n-1] of n orderable elements

// Output: Array A[0..n-1] sorted in ascending order

```
for i ← 1 to n-1 do
    v ← A[i]
    j ← i - 1
    while j ≥ 0 and A[j] > v do
        A[j+1] ← A[j]
        j ← j - 1
    A[j+1] ← v
```

The basic operation of the algorithm is comparison A[j]>v
The no. of key comparisons depends on nature of I/P given.

Sort 76 15 25 2 40 using Insertion Sort:

Pass 1:

76 (j)	15 (i)	25	2	40	v ← 15
	76	25	2	40	76 > 15
15	76	25	2	40	

Pass 2:

	15 (j)	76 (i)	25	2	40	v ← 25
	15 (j)		76 (i)	2	40	76 > 25
	15	25	76	2	40	15 < 25

Pass 3:

	15	25	76 (j)	2 (i)	40	v ← 2
	15	25 (j)		76	40	76 > 2
	15 (j)		25	76	40	25 > 2

j	15	25	76	40	15 > 2
2	15	25	76	40	

Pass 4:

			j	i	
2	15	25	76	40	v ← 40
2	15	25		76	76 > 40
2	15	25	40	76	25 < 40

Analysis:

Best Case:

If the array of input elements is already sorted, in each iteration, comparison is made only once and thus results in best case.

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

$$= n-1-1+1$$

$$= n-1$$

$$f(n) \in \Omega(n)$$

Worst Case: worst case occurs when all the elements in the array are in descending order.

$$f(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

$$= \sum_{i=1}^{n-1} i-1-0+1$$

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

$$= 1+2+3+\ldots+n-1$$

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

$$f(n) \in O(n^2)$$

Average Case:

This happens in randomly ordered arrays where

in every iteration of i, comparisons can range from 1 to i.

On an average, comparisons will be $\frac{1+2+3+\cdots+i}{i}$

$$f(n) = \sum_{i=1}^{n-1} \frac{1+2+3+\cdots+i}{i}$$

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

$$= \sum_{i=1}^{n-1} i/2 + \sum_{i=1}^{n-1} 1/2$$

$$= \frac{1}{2}\left(\frac{n(n-1)}{2}\right) + \frac{1}{2}[n-1-1+1]$$

$$= \frac{n(n-1)}{4} + \frac{n-1}{2}$$

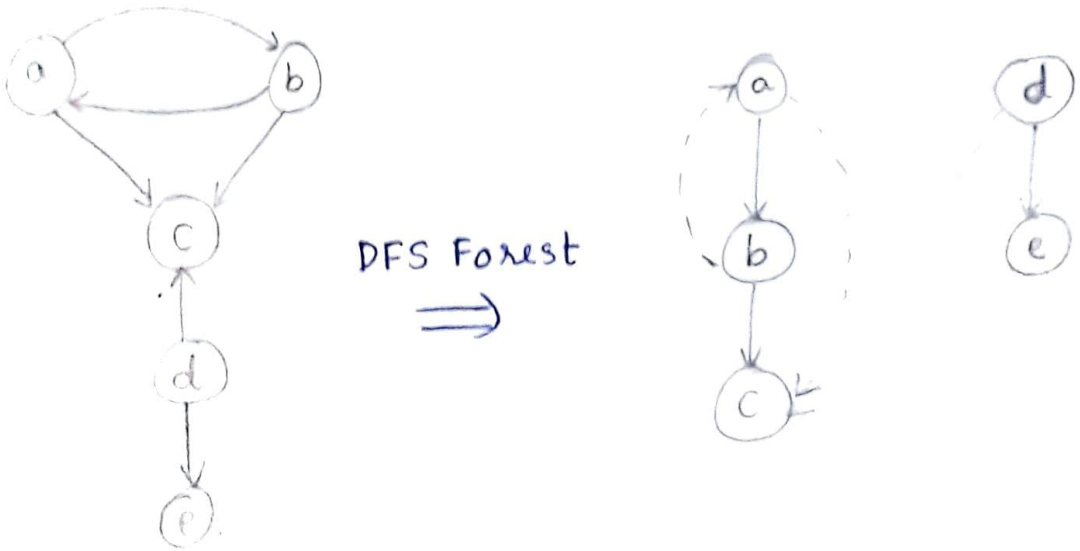
$$= \frac{1}{4}[n^2+n-2]$$

$$f(n) \in \Theta(n^2)$$

Topological Sorting:

Topological sorting is a problem that can be discussed for directed graphs. A directed graph or a digraph is a graph with directions specified for each edge. Hence the adjacency matrix will not be symmetric for directed graphs.

Consider the following directed graph:



The DFS forest consists of all types of edges: tree edges, forward edges, back edges and cross edges. We also, find that there is a directed cycle aba in the graph. However, a directed graph whose DFS forest does not have any back edges is called a dag (directed acyclic graph).

Topological sorting can be applied only to directed acyclic graphs. If a directed graph has cycles, the topological sorting will have no solution.

In a digraph, topological sorting requires us to list its vertices in such an order that for every edge in the graph, the vertex where the edge starts is listed before the vertex where the edge ends

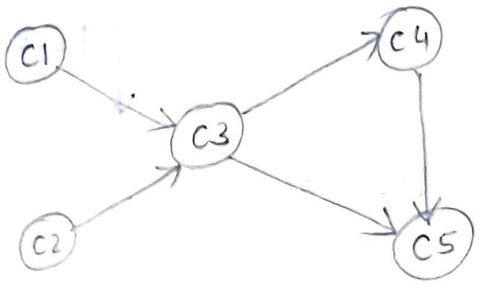
Two methods for Topological sorting:

1) DFS Method
2) Source Removal Method:

DFS Method:

1) Perform DFS traversal and note down the order in which vertices become dead ends (i.e the pop order of vertices)
2) Reversing the order yields a solution to topological sorting problem, provided, no back edges are encountered during traversal.

Eg:

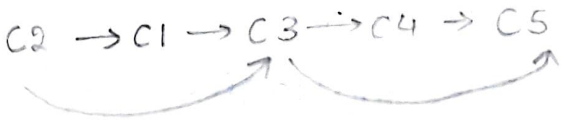


Stack:

$c5_{4,1}$
$c4_{3,2}$
$c3_{2,3}$
$c1_{1,4}$ $c2_{5,5}$

To popping order:

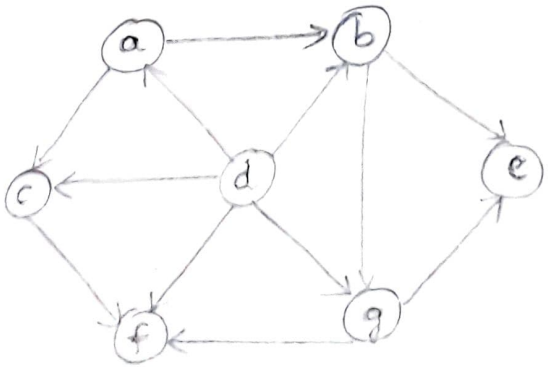
c5, c4, c3, c1, c2



∴ Topological sorting:

c2 → c1 → c3 → c4 → c5

Eg2:



Stack:

$f_{5,2}$
$e_{3,1}$ $g_{4,3}$
$b_{2,4}$ $c_{6,5}$
$a_{1,6}$ $d_{7,7}$

Popping order:

e, f, g, b, c, a, d

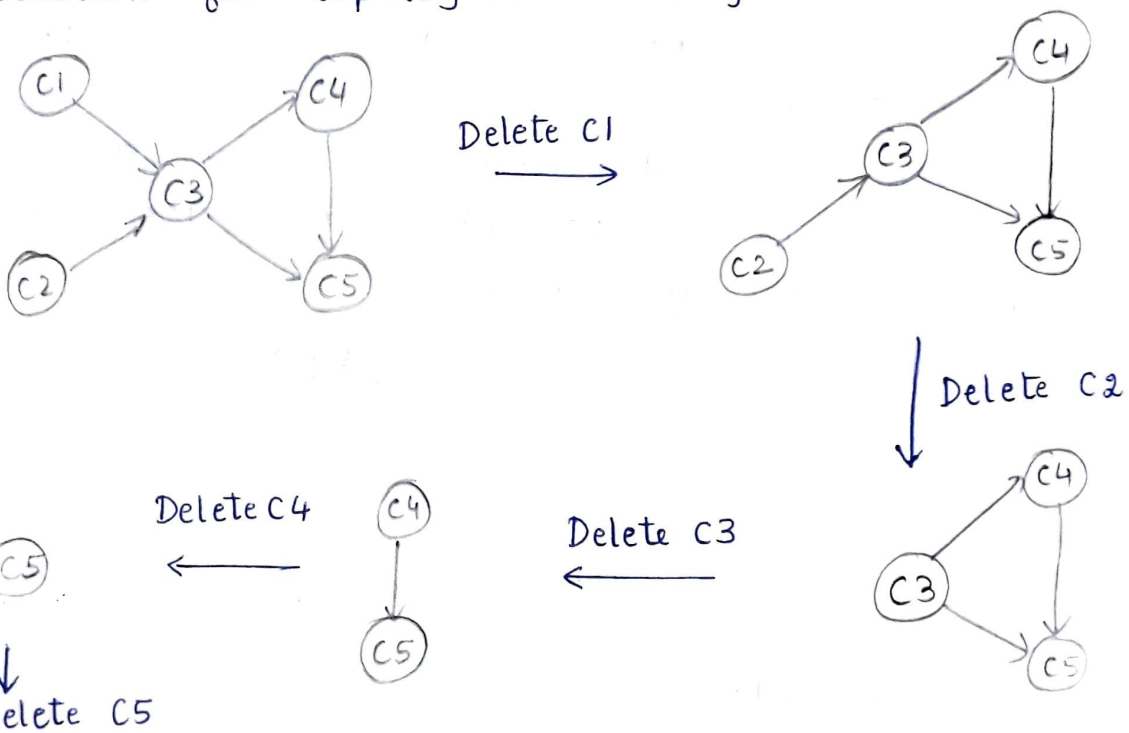
Topological sorting: d → a → c → b → g → f → e

Source Removal Method:

1) This is an algorithm based on decrease-(by-one) and conquer technique.

2) Repeatedly identify a source which is a vertex with no incoming edge and delete it along with all the edges outgoing from it. If there are several sources, break the tie arbitrarily. If there is none, the problem cannot be solved.

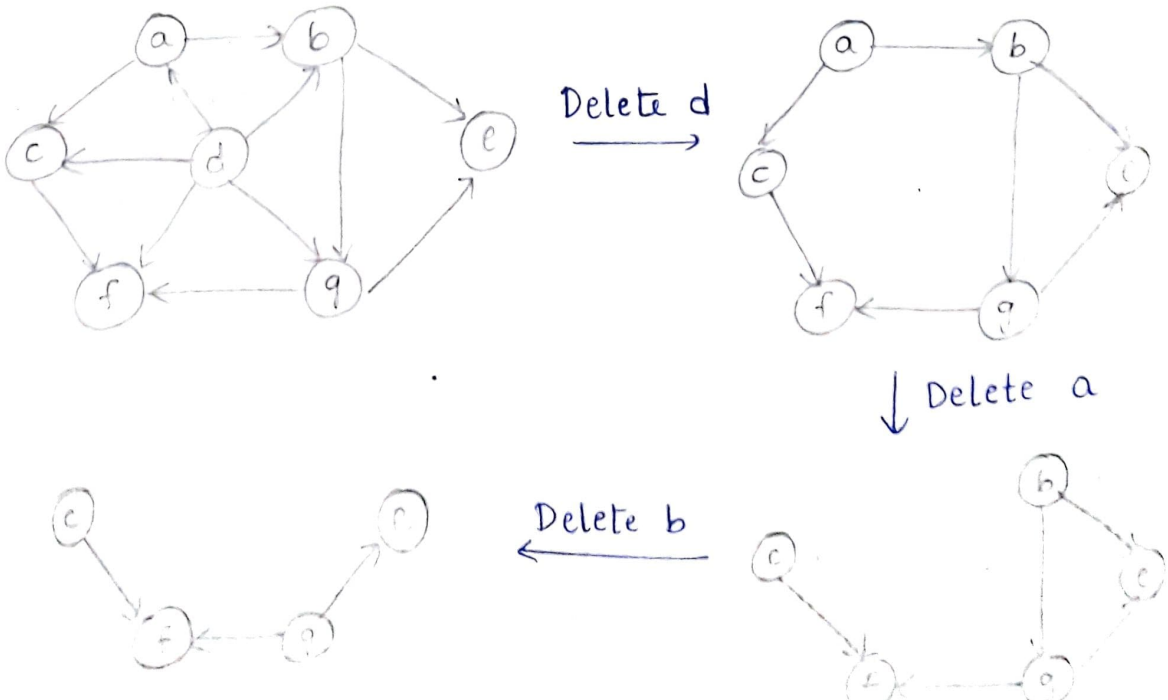
The order in which vertices are deleted will yield a solution for topological sorting.

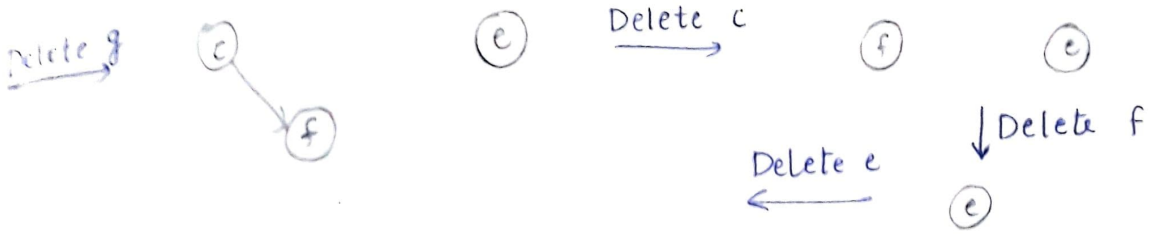


The topological sort order:

C1 → C2 → C3 → C4 → C5

Eg 2:





The topological order is:

$$d \rightarrow a \rightarrow b \rightarrow g \rightarrow c \rightarrow e \rightarrow f$$

Algorithms for generating combinatorial objects:

The most important type of combinatorial objects are permutations, combinations & subsets of a set.

Generating Permutations:

we generate permutations by assuming that the underlying set of elements are simply a set of integers from 1 to n. The solution to large problem with n elements is obtained by inserting n in various possible positions of n-1 elements. All permutations obtained will be distinct and total no. of permutations will be $n * (n-1)! = n!$

Bottom up minimal change algorithm:

Consider $n = 3$ $\{1, 2, 3\}$

Initial : 1

Insert 2 to 1 right to left : 12 21

Insert 3 to 12 right to left:

123 132 312

Insert 3 to 21 right to left:

213 231 321

Total permutations for $n = 3$ are:

123, 132, 312, 213, 231, 321

Johnson - Trotter Algorithm:

It is possible to get the permutations of n elements without explicitly generating permutations for smaller value of n. It can be done by associating direction with each element k in the permutation.

Eg: $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$

The element 'k' is said to be mobile in an arrow marked permutation if its arrow points to a smaller number adjacent to it. In the above permutation, 3 and 4 are mobile.

Algorithm JohnsonTrotter(n)

// Implements Johnson-Trotter Algorithm for //generating permutations.

// Input: A positive integer n

// Output: A list of all permutations of {1,2,..,n}

initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \ldots\ \overleftarrow{n}$

while the last permutation has a mobile element

do

find its largest mobile element k

swap k and the adjacent integer k's arrow points to

reverse the direction of all the elements that are larger than k

add the new permutation to the list.

Consider n = 3:

$\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$

$\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$

$\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$

$\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$

$\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$

$\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$

This algorithm can be implemented to run in time proportional to $\Theta(n!)$.

Lexicographic Order:

It is the order in which they would be listed in dictionary if numbers would be interpreted as letters of an alphabet.

Lexicographic ordering Example:

✓ 1 2 3

a_i a_{i+1}

a_j

✓ 1 3 2

a_i a_{i+1}

a_j

✓ 2 1 3

a_i a_{i+1}

a_j

✓ 2 3 1

a_i a_{i+1}

a_j

✓ 3 1 2

a_i a_{i+1}

a_j

✓ 3 2 1

Hence, the permutations are:

123, 132, 213, 231, 312

and 321

Consider $n = 4$. Generate permutations for $\{1, 2, 3, 4\}$ using Johnson Trotter Algorithm

$\leftarrow 1\ \leftarrow 2\ \leftarrow 3\ \leftarrow 4$
$\leftarrow 1\ \leftarrow 2\ \leftarrow 4\ \leftarrow 3$
$\leftarrow 1\ \leftarrow 4\ \leftarrow 2\ \leftarrow 3$
$\leftarrow 4\ \leftarrow 1\ \leftarrow 2\ \leftarrow 3$
$\rightarrow 4\ \leftarrow 1\ \leftarrow 3\ \leftarrow 2$
$\leftarrow 1\ \rightarrow 4\ \leftarrow 3\ \leftarrow 2$
$\leftarrow 1\ \leftarrow 3\ \rightarrow 4\ \leftarrow 2$
$\leftarrow 1\ \leftarrow 3\ \leftarrow 2\ \rightarrow 4$

$\leftarrow 3\ \leftarrow 1\ \leftarrow 2\ \leftarrow 4$
$\leftarrow 3\ \leftarrow 1\ \leftarrow 4\ \leftarrow 2$
$\leftarrow 3\ \leftarrow 4\ \leftarrow 1\ \leftarrow 2$
$\leftarrow 4\ \leftarrow 3\ \leftarrow 1\ \leftarrow 2$
$\rightarrow 4\ \rightarrow 3\ \leftarrow 2\ \leftarrow 1$
$\rightarrow 3\ \rightarrow 4\ \leftarrow 2\ \leftarrow 1$
$\rightarrow 3\ \leftarrow 2\ \rightarrow 4\ \leftarrow 1$
$\rightarrow 3\ \leftarrow 2\ \leftarrow 1\ \rightarrow 4$

$\leftarrow 2\ \rightarrow 3\ \leftarrow 1\ \leftarrow 4$
$\leftarrow 2\ \rightarrow 3\ \leftarrow 4\ \leftarrow 1$
$\leftarrow 2\ \leftarrow 4\ \rightarrow 3\ \leftarrow 1$
$\leftarrow 4\ \leftarrow 2\ \rightarrow 3\ \leftarrow 1$
$\rightarrow 4\ \leftarrow 2\ \leftarrow 1\ \rightarrow 3$
$\leftarrow 2\ \rightarrow 4\ \leftarrow 1\ \rightarrow 3$
$\leftarrow 2\ \leftarrow 1\ \rightarrow 4\ \rightarrow 3$
$\leftarrow 2\ \leftarrow 1\ \rightarrow 3\ \rightarrow 4$

Generating Subsets:

The decrease-by-one approach can be applied where all subsets of $A = \{a_1, a_2, \ldots, a_n\}$ can be divided in two groups: those containing a_n & those that do not. The latter group is the subsets of $\{a_1, a_2, \ldots, a_{n-1}\}$ and each element of power set of A can be obtained by adding a_n to subsets of $\{a_1, a_2 \ldots, a_{n-1}\}$.

Bottom up:

n	subsets
0	ϕ
1	ϕ $\{a_1\}$
2	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$
3	ϕ $\{a_1\}$ $\{a_2\}$ $\{a_1, a_2\}$ $\{a_3\}$ $\{a_1, a_3\}$ $\{a_2, a_3\}$, $\{a_1, a_2, a_3\}$

In order to avoid generation of power sets of smaller sets, we can generate 2^n subsets of n element set using bit string of length n.

Given a bit string, if $b_i = 1$, then a_i belongs to subset and if $b_i = 0$, then a_i does not belong to subset.

To generated subsets of $n = 3$,

Bit strings range from 0 to $2^n - 1 \Rightarrow 0$ to 7

000	001	010	011	100	101
ϕ	$\{a_3\}$	$\{a_2\}$	$\{a_2, a_3\}$	$\{a_1\}$	$\{a_1, a_3\}$

110	111
$\{a_1, a_2\}$	$\{a_1, a_2, a_3\}$

The minimal change algorithm for generating bit strings ensure the every bit string differs from its immediate predecessor by one single bit.

If $n = 3$,

000	001	011	010	110	111	101	100

This sequence is called binary reflected Gray code.

Binary Reflected Gray code generation for $n = 3$:

$n = 1$ 0 1

$n = 2$ 0 1 List of $n = 1$ (L1)

1 0 reversed list of $n = 1$ (L2)

Add 0 in front of each bit string in L1

00 10

Add 1 in front of each bit string in L2

11 01

00 10 11 01

$n = 3$ 00 10 11 01

01 11 10 00

⇓

000 100 110 010

011 111 101 001

⇓

000	100	110	010	011	111	101	001

TRANSFORM-AND CONQUER

Transform and conquer is a method of solving a problem by transforming the problem instance into another simpler instance using which solutions can be obtained. There are three variations in this idea which differ by what we transform at a given instance.

1) Instance Simplification:

Transformation to a simpler or more convenient instance of same problem.

Eg: Presorting.

2) Representation Change:

Transformation to different representation of same instance.

Eg: Heap sort

3) Problem Reduction:

Transformation to an instance of different problem for which an algorithm is already available.

problem's instance → simpler instance or another representation or another problem's instance → solution

Presorting:

Arranging the members in ascending or descending order before solving the actual problem is called presort.

Some problems whose efficiency can be improved using this technique are:

1) Checking element uniqueness in an array
2) Searching problem
3) Computing mode

Algorithm PresortElementUniqueness(A[0...n-1])

// Solves the element uniqueness problem by sorting
// the array first
// Input: An array A[0...n-1] of orderable elements
// Output: Returns true if A has no equal elements
// false otherwise

```
sort the array A
for i ← 0 to n - 2 do
    if A[i] = A[i+1]
        return false
return true
```

The running time of algorithm is sum of time spent on sorting and time spent on checking consecutive elements.

$$T(n) = T_{sort}(n) + T_{scan}(n)$$

$T_{sort}(n) \in \theta(n \log n)$ for worstcase of mergesort

$T_{scan}(n) = n-1$ in worst case $\in \theta(n)$

$$T(n) \in \theta(n \log n) + \theta(n)$$

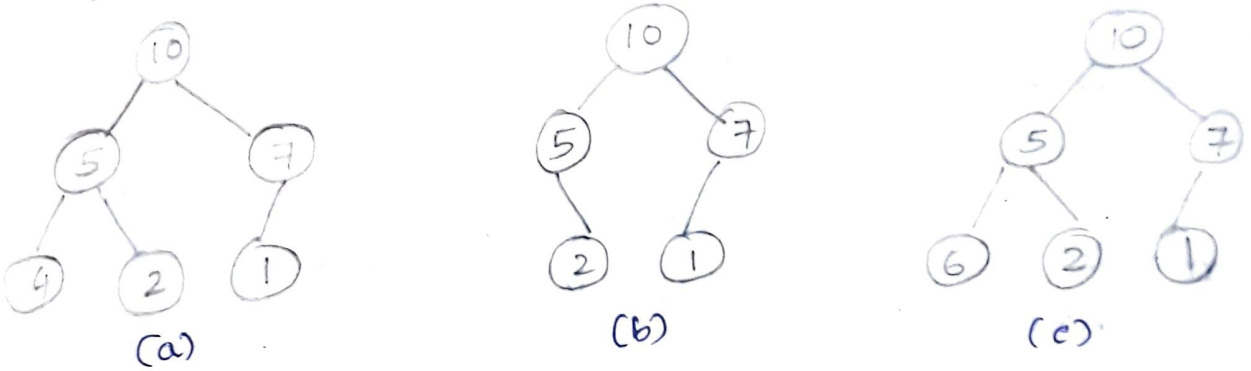
$$\in \theta(n \log n)$$ [using asymptotic notation property]

Heaps and Heapsort:

A heap can be defined as a binary tree with keys assigned to its nodes provided that the following two conditions are met:

1) The tree's shape requirement - the binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some right most leaves may be missing.

2) The parental dominance requirement - the key at each node is greater than or equal to the keys at its children.



(a) is a heap whereas (b) and (c) are not. (b) violates tree's shape requirement and (c) violates parental dominance requirement.

Properties of Heap:

1) The height of a heap with n nodes is $\lfloor \log_2 n \rfloor$

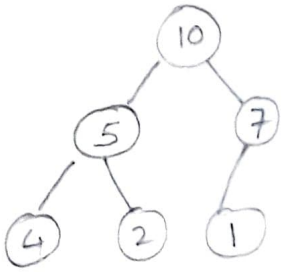
2) The root of the heap always contains the largest element.

3) A node of a heap considered along with its descendants is also a heap.

4) A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store heap's elements in position 1 to n in array. In this representation,

a) The parental node keys will be in first $\lfloor n/2 \rfloor$ positions and leaf node keys will be in last $\lceil n/2 \rceil$ positions

b) The children of a key in parental position i will be in position $2i$ and $2i+1$. Also for any key at the position i, the parent key will be in position $\lfloor i/2 \rfloor$.



$\Rightarrow$

Array Representation:

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

Bottom-up Heap Construction:

This algorithm initializes the essentially complete binary tree with n nodes by placing keys in the order given and then heapifies the tree to satisfy parental dominance requirement.

```
Algorithm BottomUpHeap(H[1...n])
// Constructs a heap from the elements of a given
//array
// Input: An array H[1..n] of orderable element
// 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
```

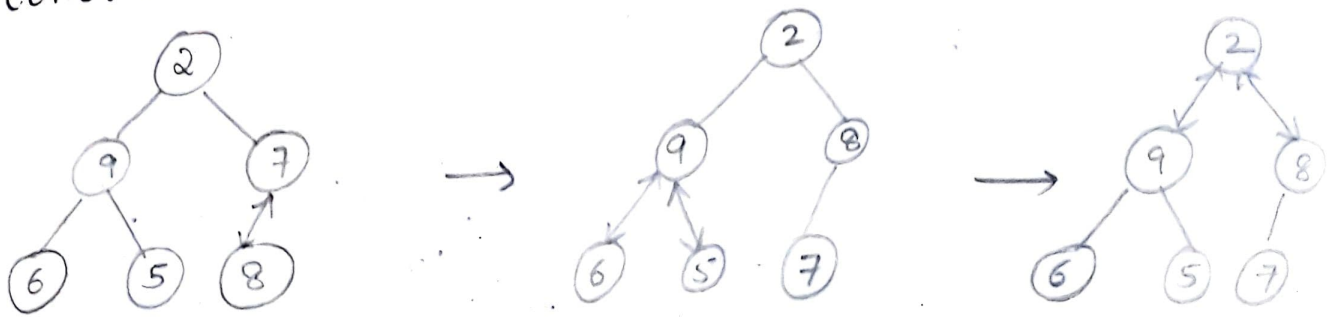
else

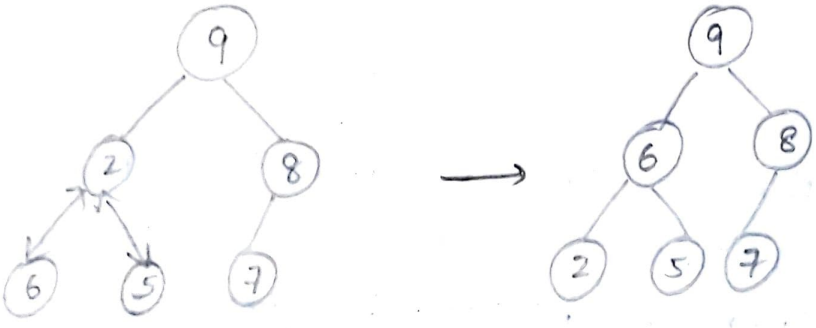
$H[k] \leftarrow H[j]$

$k \leftarrow j$

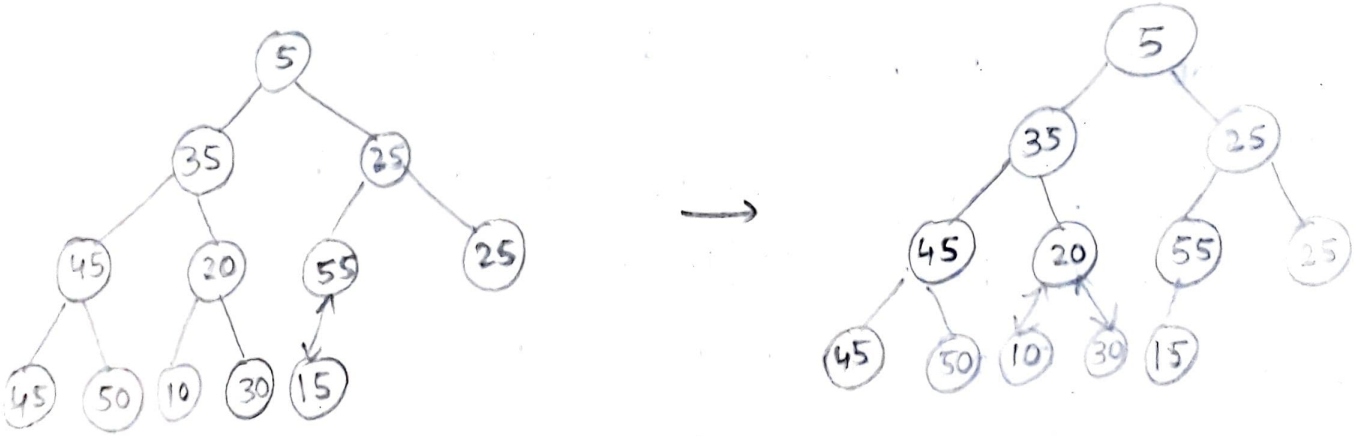
$H[k] \leftarrow v$

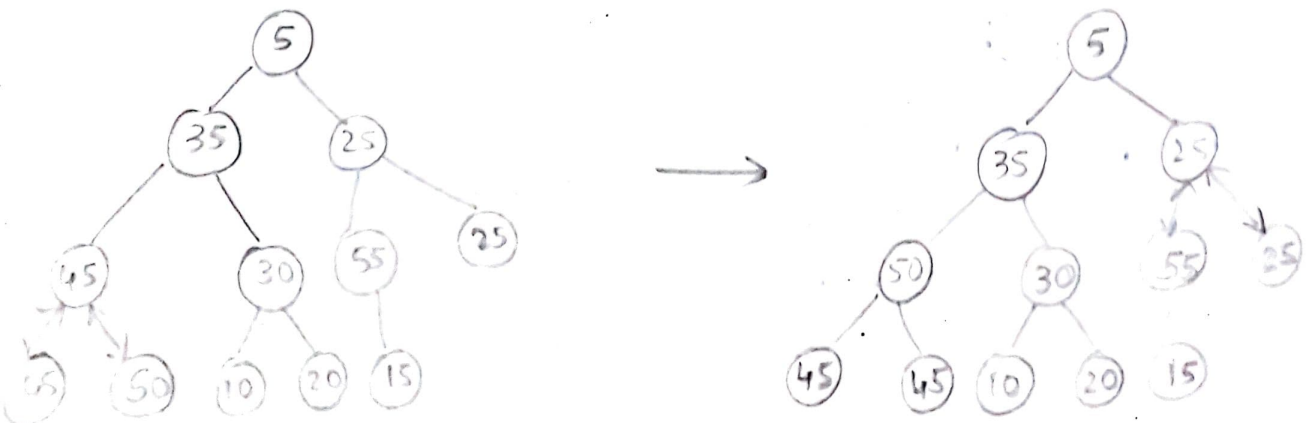
Construct a heap for list 2, 9, 7, 6, 5, 8

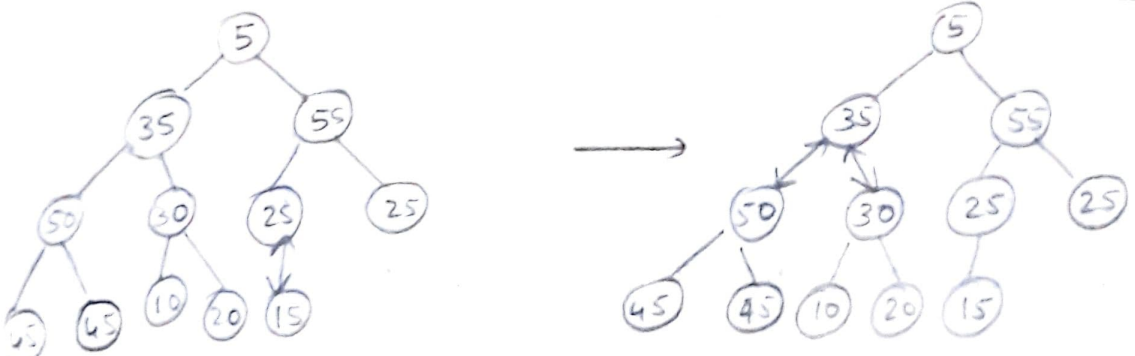


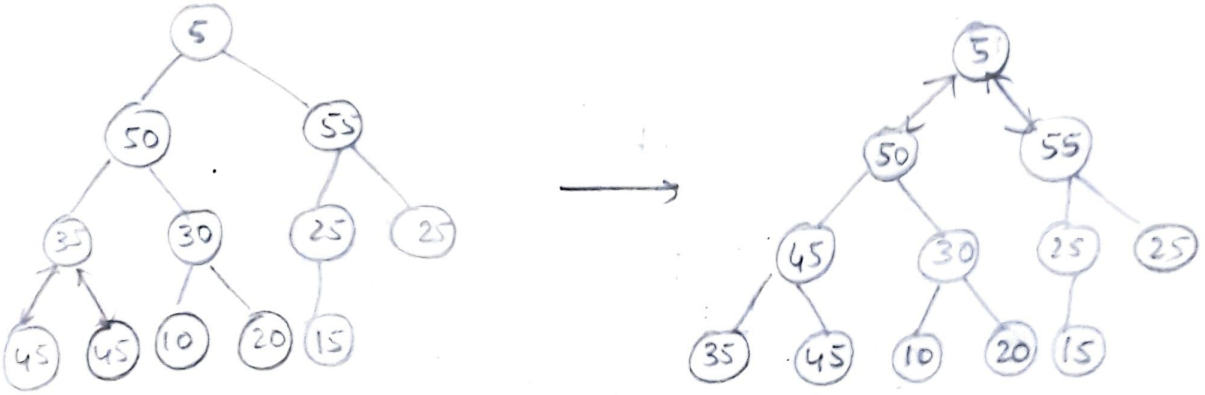


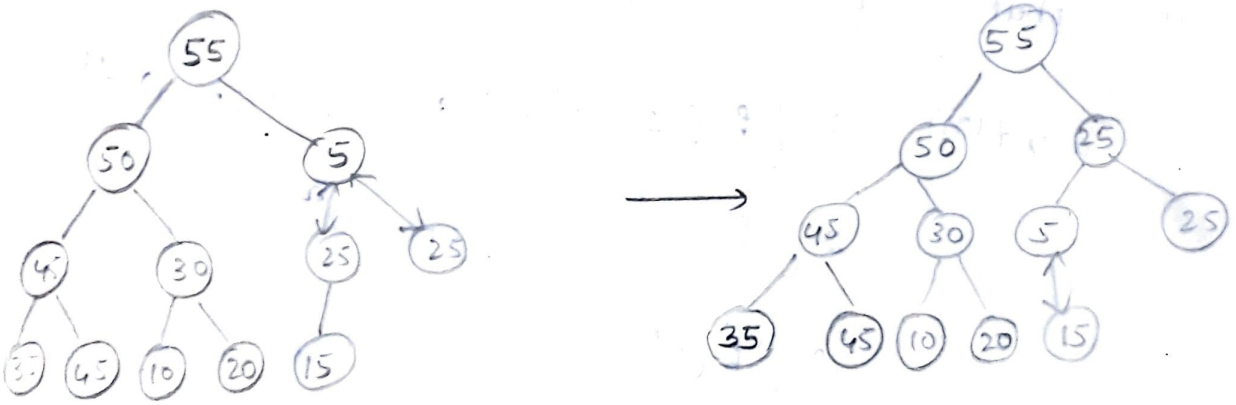
Construct a heap for list 5, 35, 25, 45, 20, 55, 25, 45, 50, 10, 30, 15

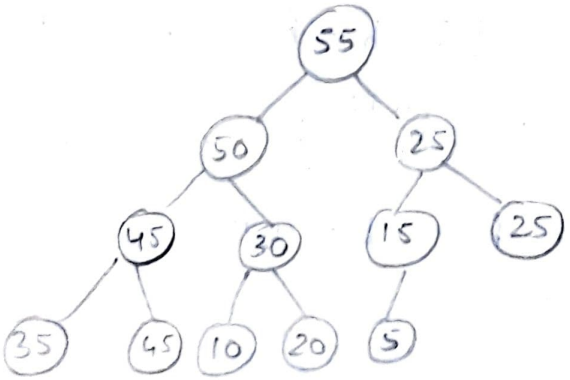












Analysis for creating a heap:

While creating a heap, the no. of comparisons required to move an item from parent position to child position is 2. First comparison is to find largest child and second comparison is to determine if parent should be exchanged with child.

∴ Total no. of comparison to move a child from level i to leaf level in a tree of height h is $2(h-i)$. in worst case.

The no. of nodes at each level $i = 2^i$

∴ No. of comparisons required to move all nodes from level i to leaf level is $= 2^i \cdot 2(h-i)$

The no. of levels range from 0 to $h-1$. The total no. of key comparison from level $h-1$ to level 0 is given by,

$$C_{worst}(h) = \sum_{i=0}^{h-1} 2(h-i)\, 2^i$$

$$= 2 \sum_{i=0}^{h-1} (h-i)\, 2^i$$

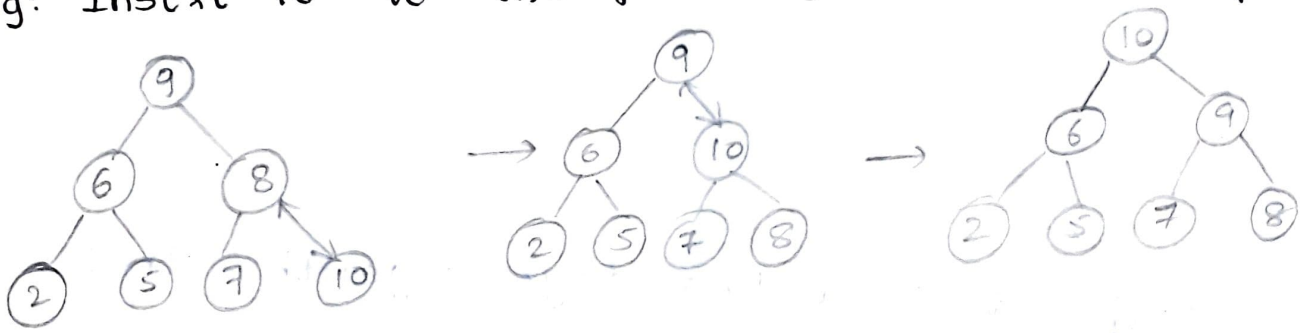
$C_{worst}(n) = 2[n - \log_2(n+1)]$

$\therefore C_{worst}(n) \in O(n)$

Top Down Heap Construction:

This algorithm constructs heap by successively inserting new key to previously constructed heap. First attach new key after the last leaf of existing heap. Then shift K to its appropriate position by heapification. For every item to be inserted, it may have to move through the height of the tree, i.e logn. Hence efficiency of insertion for one node is $O(\log_2 n)$

Eg: Insert 10 to the previously constructed heap



Maximum Key Deletion from a Heap:

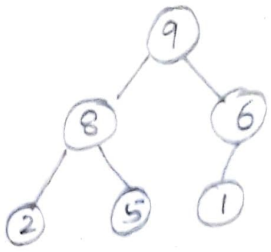
Step 1: Exchange the root's key with the last key K of the heap.

Step 2: Decrease the size of heap by 1

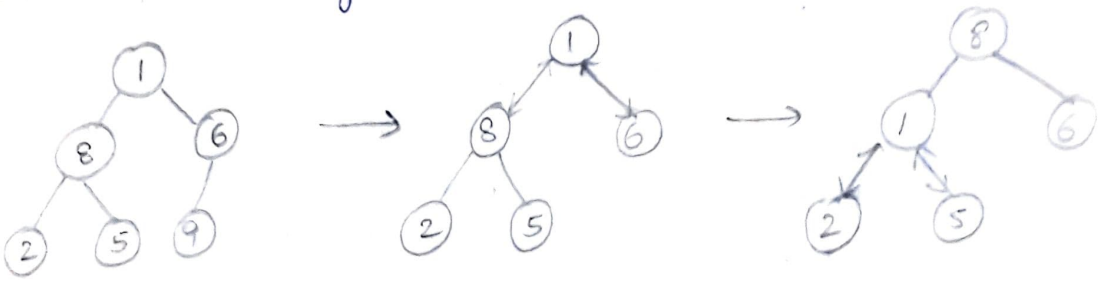
Step 3: "Heapify" the smaller tree by shifting K down the tree exactly in the same way we did for bottom up heap construction algorithm. That is, verify the parental dominance for K. If it holds, we are done; if not, swap K with the larger of its children and repeat this operation until parental dominance condition holds for K in its new position.

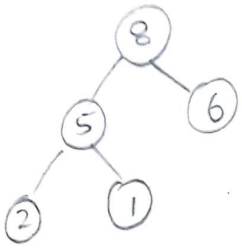
The efficiency of deletion is determined by no of key comparisons needed to heapify the tree after swap has been made and size of tree is reduced by 1.

Since, no of comparisons will not be more than twice the height of heap, the time efficiency for deletion is $O(\log n)$ as well.



Delete max key value - 9





HeapSort:

Heapsort is a two stage algorithm that works as follows:

Step 1) Heap construction: Construct a heap for a given array.

Step 2) (maximum deletions): Apply the root deletion operation $n-1$ times to remaining heap.

Analysis:

The heap construction stage of Heapsort is in $O(n)$, we need to find time efficiency for the second stage. The no. of comparisons $C(n)$ can be given by:

$$C(n) \leq 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \leq 2 \sum_{i=1}^{n-1} \log_2 i$$

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

Hence $C(n) \in O(n \log n)$ for second stage of heapsort.

Hence, the time efficiency for Heapsort can be given as follows:

Time efficiency of Heap Sort = Time efficiency to create heap + Time efficiency to swap and recreate heap

$$T(n) = O(n) + O(n \log n)$$

$$T(n) \in O(n \log n)$$

Problem: Sort the list 2, 9, 7, 6, 5, 8

1) i) Heap Construction (Already shown in example)

2 9 7 6 5 8

2 9 8 6 5 7

2 9 8 6 5 7

9 2 8 6 5 7

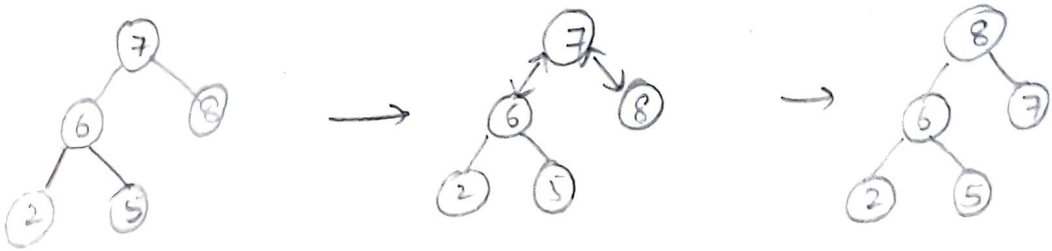
9 6 8 2 5 7

Stage 2) Maximum Deletion:

Exchange first and last elements

7 6 8 2 5 | 9

2) Recreate Heap

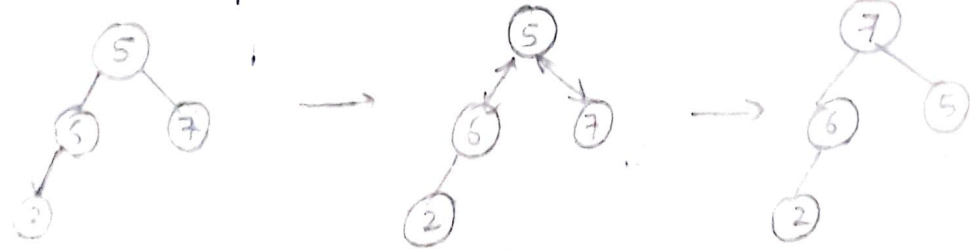


8 6 7 2 5

Delete Maximum element:

5 6 7 2 | 8

3) Recreate Heap:



7 6 5 2

Maximum Deletion:

2 6 5 | 7

4) Recreate Heap

6 2 5

Maximum Deletion:

5 2 | 6

5) Recreate Heap

5 2

Maximum Deletion:

2 | 5

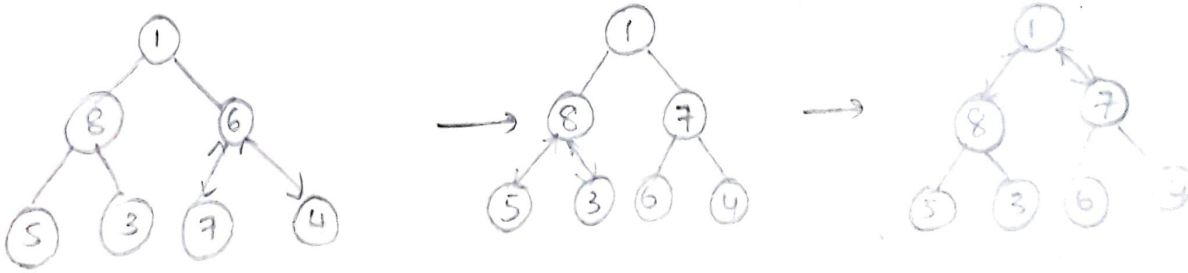
6) 2

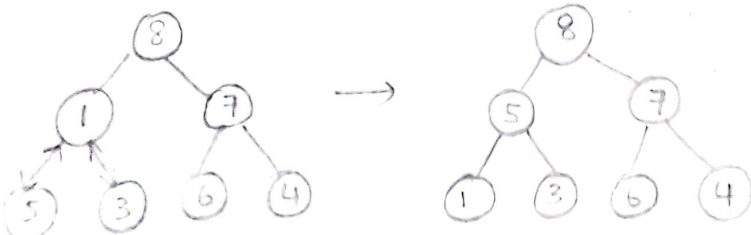
∴ The sorted list is: 2 5 6 7 8 9

HW:

a) Sort the elements {S, O, R, T, I, N, G} using Heapsort.

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





1, 8, 6, 5, 3, 7, 4

1, 8, 7, 5, 3, 6, 4

1, 8, 7, 5, 3, 6, 4

8, 1, 7, 5, 3, 6, 4

8, 5, 7, 1, 3, 6, 4

The resultant heap is:

8, 5, 7, 1, 3, 6, 4

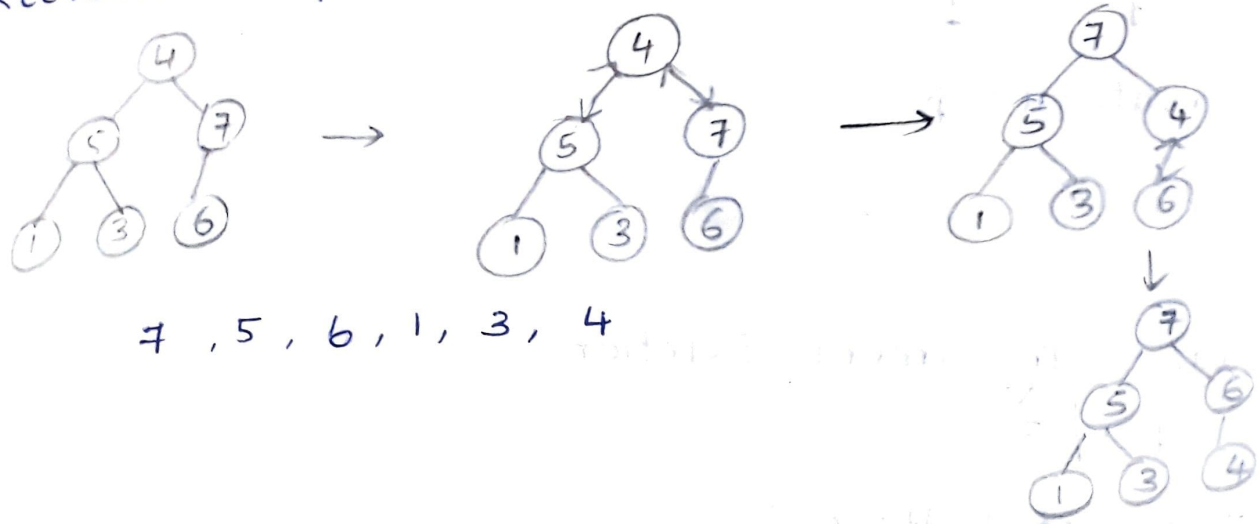
Heapsort

Stage 2: maximum Deletion

Exchange first and last element

4, 5, 7, 1, 3, 6 | 8

Recreate Heap:

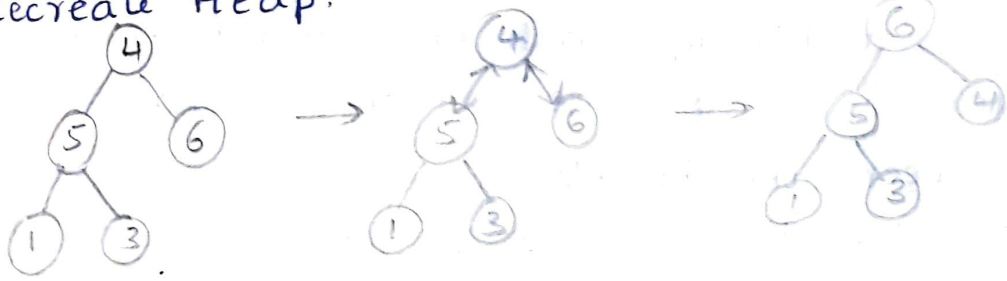


7, 5, 6, 1, 3, 4

Stage 2: Maximum Deletion:

4, 5, 6, 1, 3 | 7

Recreate Heap:

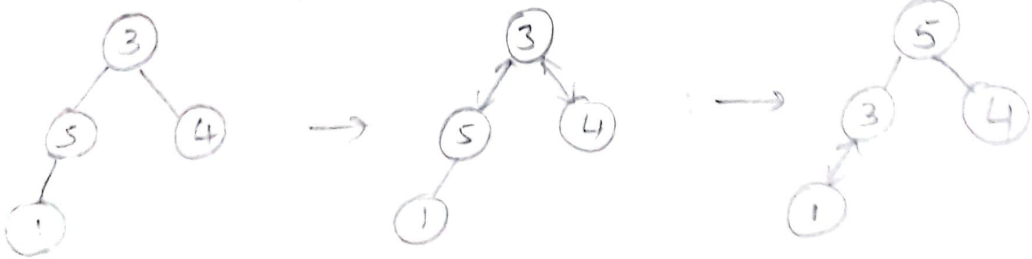


6, 5, 4, 1, 3

Stage 2: Maximum Deletion.

3, 5, 4, 1 | 6

Recreate Heap:



5, 3, 4, 1

Stage 2: Maximum Deletion

1, 3, 4 | 5

Recreate Heap

Stage 2 Maximum Deletion.

1 , 3 | 4 (4 ×)

Recreate Heap:

Stage 2: Maximum Deletion

1 | 3 (3 ×)

∴ The sorted Heap is:

1 , 3 , 4 , 5 , 6 , 7 , 8 .

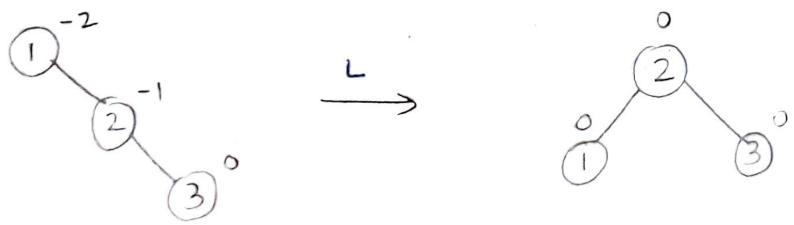
AVL Trees:

An AVL Tree is a binary search tree in which the balance factor of every node, which is defined as difference between the heights of the node's left and right subtrees is either 0, 1 or -1.

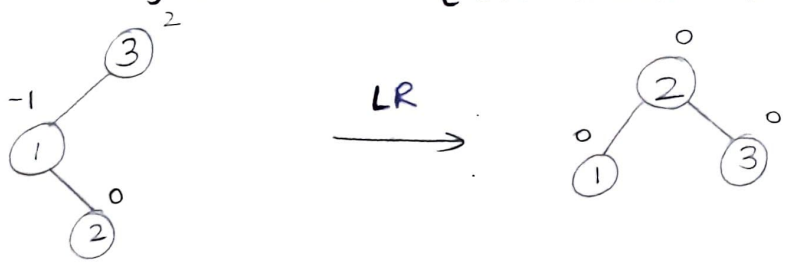
If an insertion of new node makes AVL Tree unbalanced we transform the tree by rotation. The rotation will be performed on subtree rooted at node whose balance has become -2 or +2. There are 4-types of rotation

1) R-Rotation (single right rotation):

2) L-Rotation (single left rotation):



3) Double Left Right Rotation (LR -Rotation):

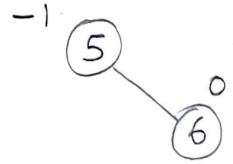


4) RL - rotation (Double Right Left Rotation):

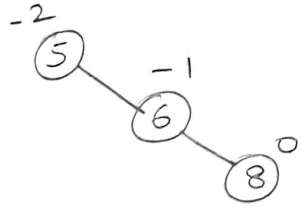
Construct an AVL Tree for the list 5, 6, 8, 3, 2, 4, 7.

1) Insert 5

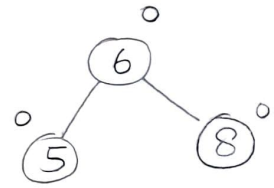
2) Insert 6



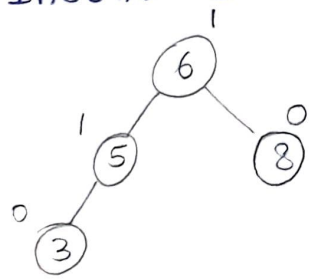
3) Insert 8



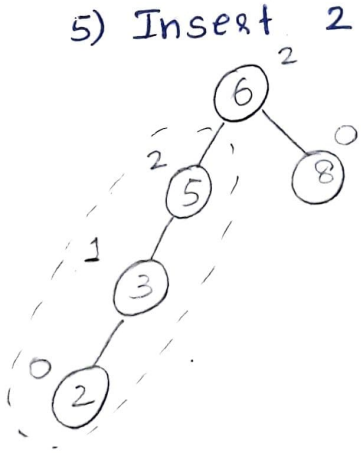
L(5)



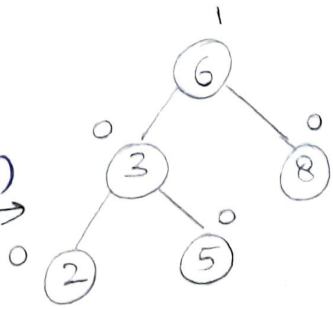
4) Insert 3



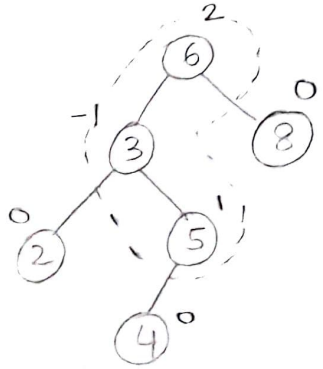
5) Insert 2



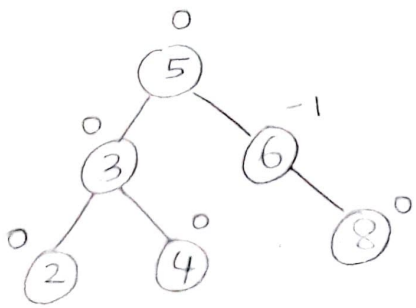
R(5)



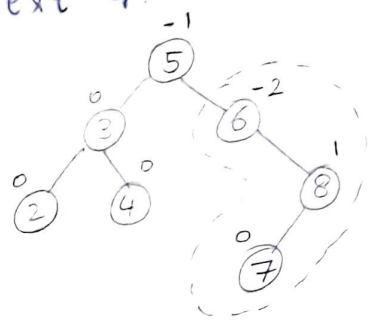
6) Insert 4



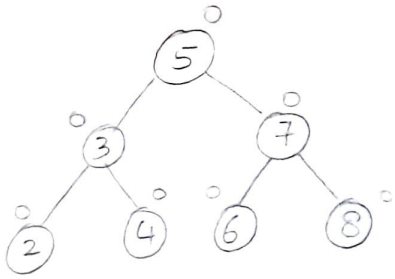
LR(6)



Insert 7:



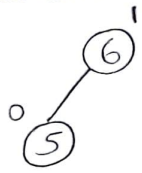
RL(6)



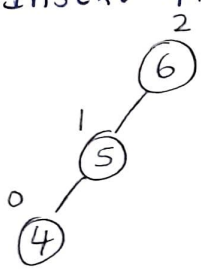
2) 6, 5, 4, 3, 2, 1

Insert 6:

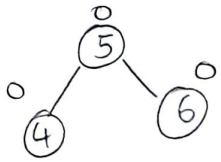
Insert 5:



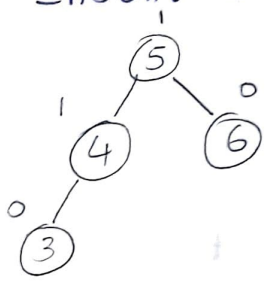
Insert 4:



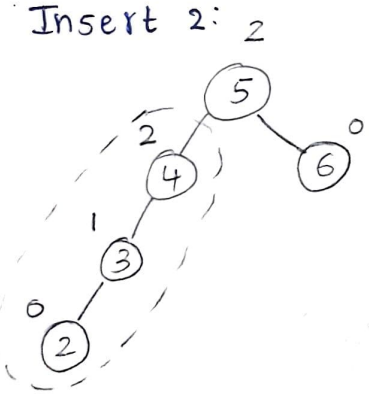
R(6)



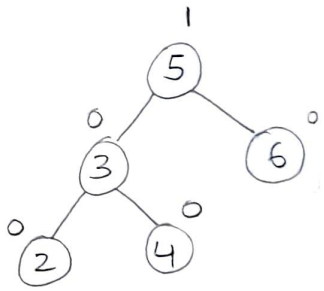
Insert 3:



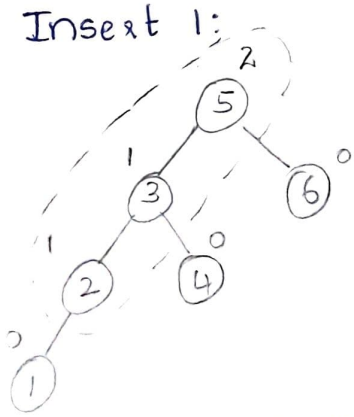
Insert 2:



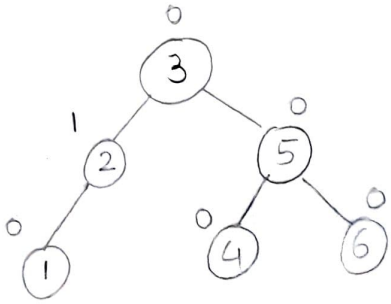
R(4)



Insert 1:



R(5)



HW: Construct AVL Tree for 3, 6, 5, 1, 2, 4

Analysis for AVL Trees:

For any search tree, the characteristic is the tree's height. AVL tree's height is bounded above and below by logarithmic functions.

$\lfloor \log_2 n \rfloor \leq h < 1.44 \log_2(n+2) - 1.3277$

Hence the searching and insertion operation are $\theta(\log n)$ in worst case.

2-3 Trees:

A 2-3 Tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes. A 2-node contains a single key K and has two children: the left child serves as root for subtree whose keys are less than K and right child serves as root for subtree whose keys are greater than K.

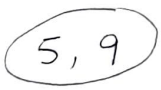
A 3-node contains two ordered keys K_1 and K_2 ($K_1 < K_2$) and has 3 children. The leftmost child serves as a root of subtree with keys less than K_1, the middle child serves as a root of subtree with keys between K_1 and K_2 and the rightmost child serves as the root of a subtree with keys greater than K_2.

The last requirement of 2-3 Tree is that all leaves must be on same level.

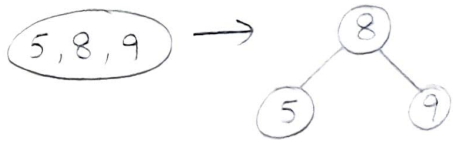
Construct a 2-3 Tree for list 9, 5, 8, 3, 2, 4, 7.

Insert 9:

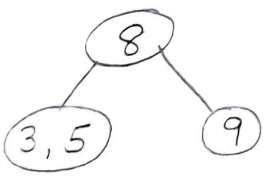
Insert 5:



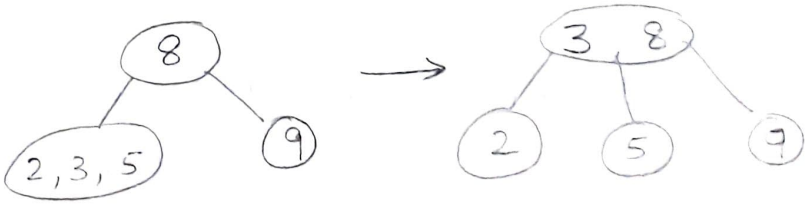
Insert 8:



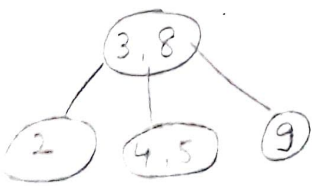
Insert 3:



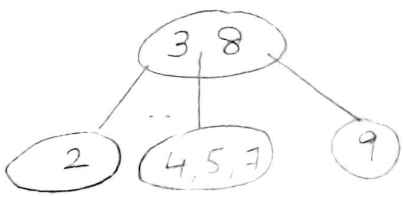
Insert 2:



Insert 4:



Insert 7:

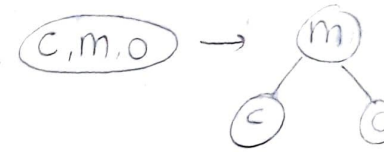


Construct a 2-3 Tree for list C, O, M, P, U, T, I, N, G

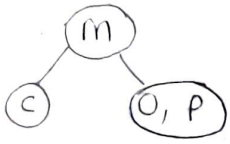
Insert C:

Insert O:

Insert M:

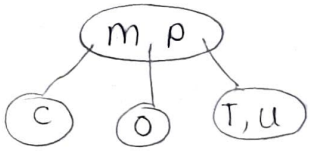


Insert P:

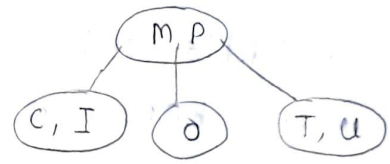


Insert U:

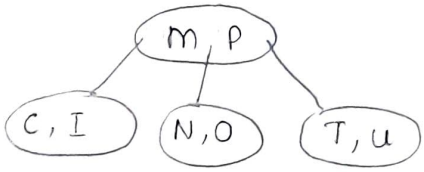
Insert T:



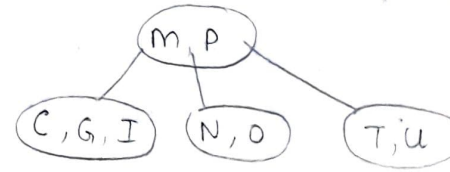
Insert I:

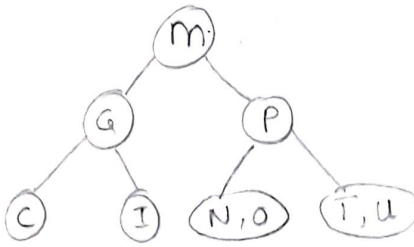


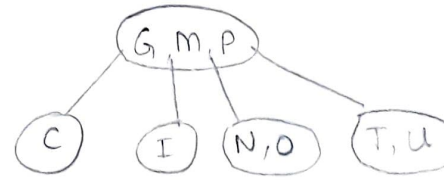
Insert N:



Insert G:







Analysis:

To find upper bound for 2-3 Tree:

For any 2-3 Tree of height h with n nodes, we get inequality.

$$n \geq 2^0 + 2^1 + 2^2 + \ldots + 2^h$$

$$n \geq 2^{h+1} - 1$$

$$n + 1 \geq 2^{h+1}$$

Taking $\log_2$ on both sides.

$$\log_2(n+1) \geq h+1$$

$$h \leq \log_2(n+1) - 1$$

To find lower bound for 2-3 Tree:

In a 2-3 tree of height h with largest no. of keys is a full tree of 3-nodes, each with 2 keys & 3 children. Therefore for any 2-3 Tree of n nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + 2 \cdot 3^2 + \ldots + 2 \cdot 3^h$$

$$n \leq \frac{2\left[3^{h+1} - 1\right]}{2}$$

$$n \leq 3^{h+1} - 1 \quad \Rightarrow \quad n+1 \leq 3^{h+1}$$

Taking $\log_3$ on both sides,

$$\log_3(n+1) \leq \log_3(3^{h+1})$$

$$\log_3(n+1) \leq h+1$$

$$h \geq \log_3(n+1) - 1$$

Hence, with lower bound and upper bound for h,

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1$$

Hence, insertion, deletion, searching are all in $\Theta(\log n)$ in both worst and average case.