UNIT-III

Decrease and Conquer:

The decrease-and-conquer technique is based on exploiting the relationship between a solution to given instance of a problem and a solution to a smaller instance of the same problem. Once such a relationship is established, it can be exploited either top down (recursively) or bottom up (without recursion)

There are three major variations of decrease & conquer.

→ decrease by a constant
→ decrease by a constant factor
→ variable size decrease.

1) Decrease by a constant

In the decrease-by-a-constant variation the size of an instance is reduce by the same constant on each iteration of the algorithm. Typically this constant is equal to one.

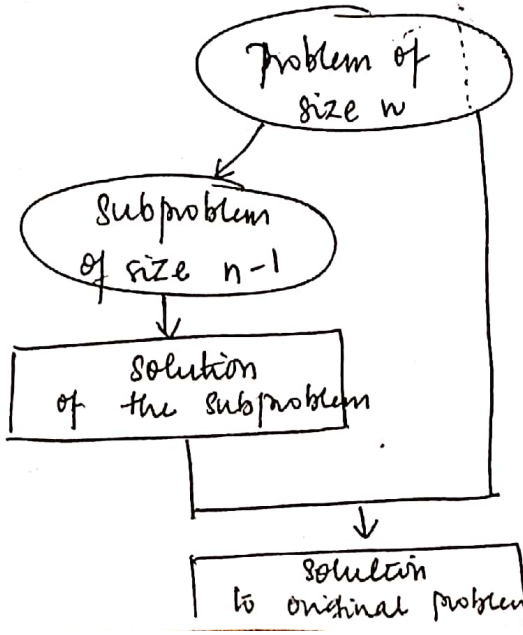


Fig:- Decrease (by one) -and conquer technique.

eg Exponentiation problem of computing a^n for positive integer exponents.

$a^n = ?$

$2^4 = ?$

$a^{n-1} = ?$

$2^3 = ?$

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

$2^4 = 2^3 \cdot 2 = 8 \cdot 2 = 16$.

So the function $f(n) = a^n$ can be computed either "top down" by using its recursive definition

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

2] Decrease by a constant factor

This technique suggests reducing a problem's instance by the same factor on each iteration of the algorithm. (usually by 2).

eg $a^n = 1$ This is true if n is 0 eg $3^0 = 1$

$a^n = a$ This is true if n is 1 eg $3^1 = 3$

$a^n = a^{n/2} \cdot a^{n/2} = (a^{n/2})^2$ This is true if n is even eg $3^8 = 3^4 \cdot 3^4 = (3^4)^2$

$a^n = a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a = a^{((n-1)/2)^2} \cdot a$ This is true if n is odd

eg $3^9 = 3^4 \cdot 3^4 \cdot 3 = (3^4)^2 \cdot 3$

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even \& positive} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd \& greater than 1} \\ a & \text{if } n = 1. \end{cases}$$

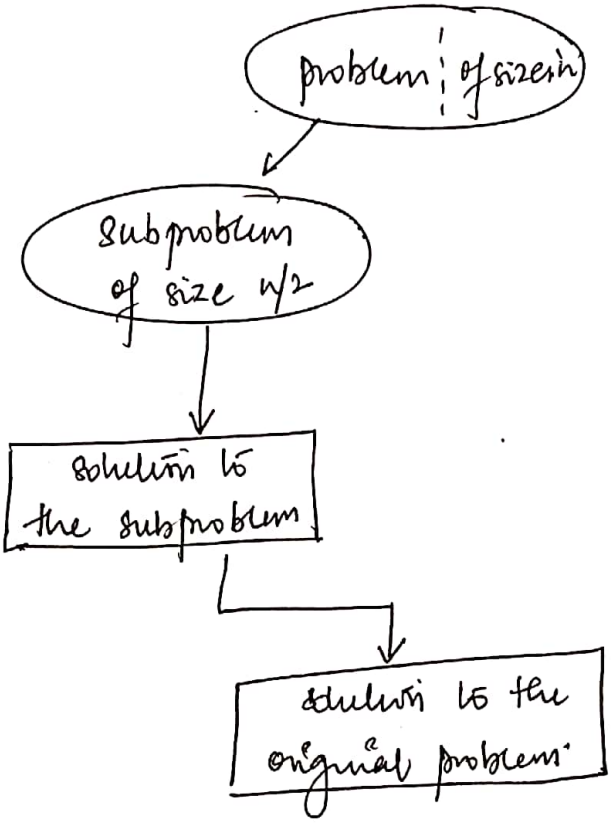


Fig: Decrease (by half)- and conquer technique.

3) Variable-size-decrease

In variable-size-decrease, in each iteration of the loop the size reduction pattern varies from one iteration of the algorithm to another iteration.

Eg GCD of two numbers m & n using Euclid's algorithm.

$$gcd(m, n) = gcd(n, m \bmod n).$$

$A[0] \leq \dots \leq A[j] < A[j+1] \leq \dots \leq A[i-1] \mid A[i] \dots A[n-1]$

The algorithm works as follows:-
Starting with A[1] and ending with A[n-1], A[i] is inserted in its appropriate place among the first i elements of the array that have been already sorted (but, unlike selection sort, are generally not in their final positions).

Algorithm InsertionSort (A[0...n-1])
// Sorts a given array by insertion sort.
// Input: An array A[0...n-1] of n orderable elements
// Output: Array A[0...n-1] sorted in nondecreasing 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
    end while
    A[j+1] ← v
end for.
```

An application of the decrease-by-one technique is sorting array A[0...n-1]

⇒ we need to find an appropriate position for A[n-1] among the sorted elements and insert it there.

⇒ There are 3 reasonable alternatives for doing this :-

1] we scan the sorted subarray from left to right until the first element greater than or equal to A[n-1] is encountered and then insert A[n-1] right before that element

2] Second 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

These two are essentially equivalent. usually it is the second one that is implemented in practice because it is better for sorted and almost sorted arrays. The resulting algorithm is called straight insertion-sort or simply insertion sort

(b) The third alternative is to use binary search to find an appropriate position for A[n-1] in the sorted position of the array. The resulting algorithm is called binary insertion sort

89 | 45 68 90 29 34 17 $n=7$

$v = A[i]$

$= 45$

$j = 0$

$89 > 45$

$A[j+1] = A[1] = 89$

$j = -1 \rightarrow$ out

$A[j+1] = v$

$A[0] = 45$

---

45 89 | (68) 90 29 34 17 (i=2, j=1)

v

$A[2] = A[1]$

$j = 0$

→ 45 89 89 90 29 17 (j=0)

---

45 68 89 | (90) 29 34 17 → (i=3, j=2)

v

---

45 68 89 90 | (29) 34 17 (i=4, j=3)

v

$A[4] = A[3]$ → 89 90 90 34 17

$j = 2$.

$A[3] = A[2]$

~~45 68 89 89 29 34 17~~

$j = 1$

A[

---

29 45 68 89 90 | (34) 17 (i=5, j=4)

v

29 34 45 68 89 90 | (17) (i=6, j=5)

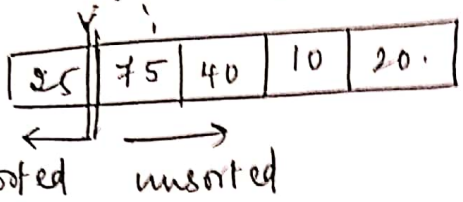
v

17 29 34 45 68 89 90

eg 25 75 40 10 20

Step 1 Item to be inserted = 75

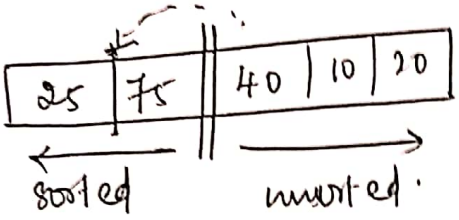
item = a[1]



75 is inserted after 25

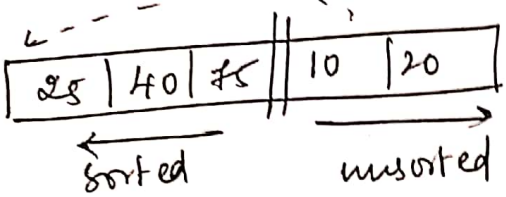
~~item~~

Step 2 Item to be inserted = 40 = a[2]



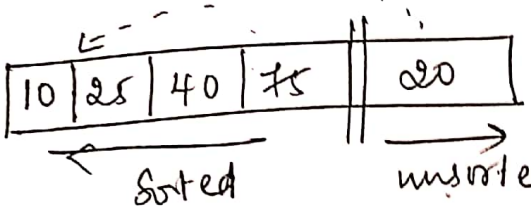
40 inserted ~~after~~ between 25 & 75

Step 3 Item to be inserted = 10 = a[3]

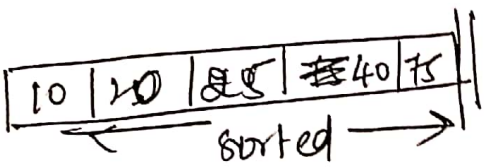


10 is inserted before 25

Step 4 Item to be inserted = 20 = a[4]



O/P



① item = a[1] to a[4]

item = a[i] where i = 1 to n-1

② item is compared with a[j] as long as item < a[j] and j >= 0 with initial value of i = i - 1

→ copy a[j] to a[j+1]

→ decrement j by 1

Analysis

The best case occurs when the items in the list are partially or nearly sorted. Whenever for loop is executed, the expression "item < a[j]" is executed. Since the loop is executed (n-1) times, the expression "item < a[j]" is also executed (n-1) times. So the time complexity is given by

$$C(n) = \sum_{i=1}^{n-1} 1 = (n-1) - 1 + 1 = n - 1 = \Omega(n)$$

worst case

The worst case occurs when the expression "item < a[j]" is executed maximum no. of times. ie when the elements are sorted in decreasing order.

```
for i ← 1 to n-1 do
    v ← a[i]
    j ← i-1
    while (j ≥ 0 and a[j] > v)
```

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

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

$$= \frac{n^2}{2} - \frac{n}{2}$$

$\in \Theta(n^2)$

Average case :-

On randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays ie

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2) .$$

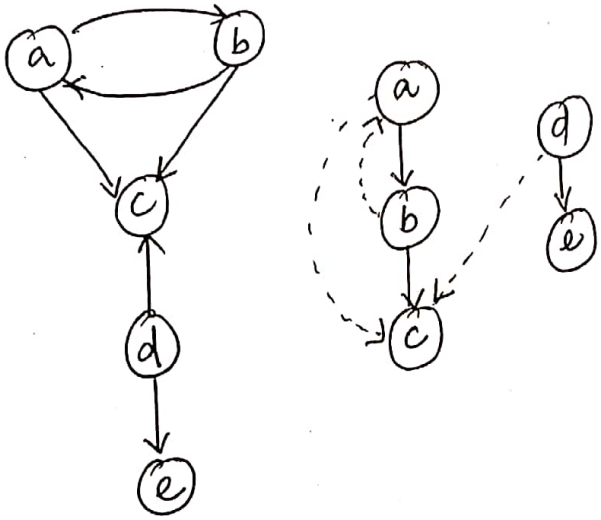
Topological Sorting

Definitions

A directed graph, or digraph for short is a graph with direction specified for all its edges.

The adjacency matrix and adjacency list are still two principal means of representing a digraph.

eg



The depth-first search forest exhibits all four types of edges possible in a DFS forest of a directed graph: tree edges (ab, bc, de), back edges (ba) from vertices to their ancestors, forward edges (ac) from vertices to their descendants in the tree other than their children & cross edge (dc).

The presence of a back edge indicates that the digraph has a directed cycle → a directed cycle in a digraph is a sequence of three

or more of its vertices that starts & ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor)

- If a DFS forest of a digraph has no back edges, the digraph is a dag.

Topological sorting

→ The topological sort of a directed acyclic graph (DAG) is a linear ordering of all the vertices such that for every edge (u,v) in graph G, the vertex u appears before the vertex v in the ordering

→ If A depends on B and B depends on A, then it is cyclic. A graph which is cyclic does not have topological sequence.

→ Thus for topological sorting to be possible, a digraph must be a dag.

→ It turns out that being a dag is not only necessary but also sufficient for topological sorting to be possible. ie if a digraph has no cycles the topological sorting problem for it has a solution

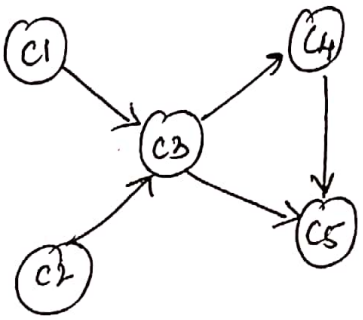
There are two efficient algorithms that both verify whether a digraph is a dag and if it is produce an ordering of vertices that solves the topological sorting problem.

Topological sorting methods
- DFS method
- Source removal method

DFS method

1) Perform DFS traversal and note the order in which vertices become dead ends (ie are popped off the traversal stack)
2) Reversing this order yields a solution to the topological sorting problem, provided of course no back edge has been encountered during the traversal.
3) If back edge has been encountered, the digraph is not a dag, and topological sorting of its vertices is impossible.

eg



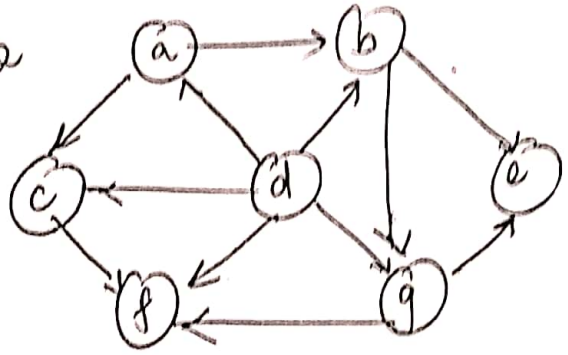
C_{51}
C_{42}
C_{33}
C_{14} C_{25}

The popping - off order :-

C_5, C_4, C_3, C_1, C_2

The topological sorted list

eg-2



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

Popping order

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

Topological order

d → a → c → b → g → f → e

Source removal method

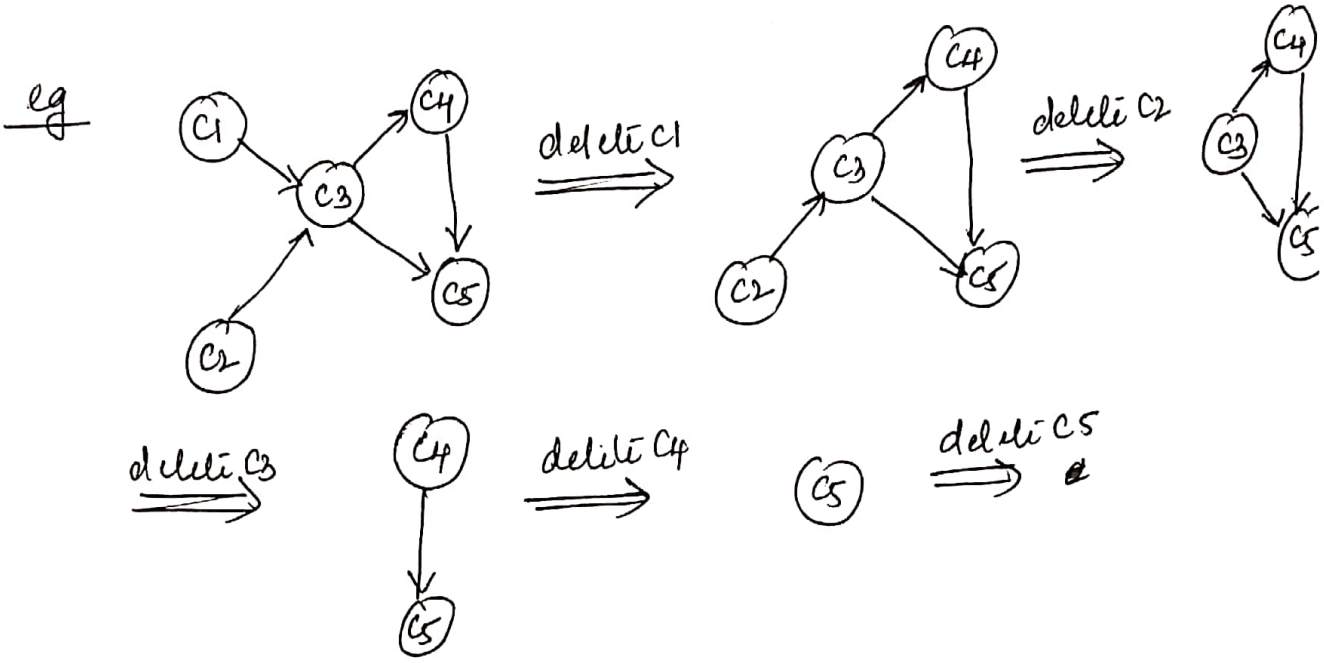
→ The second algorithm is based on direct implementation of the decrease (by one) - and conquer technique.

→ · Repeatedly identify in a remaining digraph a source which is a vertex with no incoming edges and delete it along with all the edges outgoing from it.

→ If there are several sources, break the tie arbitrarily.
If there is none, stop because the problem cannot be solved.

→ The order in which the vertices are deleted yields a solution to the topological sorting problem.

eg



The solution is C1, C2, C3, C4, C5

Algorithms for Generating Combinatorial Objects

- In this approach, we first assume that we have a solution to a given smaller problem. For eg, assume that we have solution for n-1 elements. The solution to the larger problem with n elements is obtained by inserting n in various possible positions of n-1 elements.
- All the permutations obtained in this fashion are distinct and the total number obtained will be $n*(n-1)! = n!$

Generate permutations of {1, 2, 3, 4} using bottom-up minimal-change algorithm.

Initial: 1

Insert 2: right to left → 12 21

Insert 3 to 12 right to left

123
132
312

Insert 3 to 21 left to right

321
231
213

Insert 4 :-

Insert 4 from right to left to 123

1234
1243
1423
4123

Insert 4 from left to right to 132

4132
1432
1342
1324

Insert 4 from right to left to 312

3124
3142
3412
4312

Insert 4 from left to right 321

4321
3421
3241
3214

Insert 4 from right to left 231

2314
2341
2431
4231

Insert 4 from left to right to 213

4213
2413
2143
2134

Advantages

- Each permutation can be obtained from its immediate predecessor by exchanging just two elements.
- The minimal-change requirement is beneficial for algorithm speed and for the application using the permutation

Disadvantages

- To get the permutation for n elements we need to know the permutation for (n-1) elements

Johnson-Trotter algorithm

- Using this algorithm we can generate permutation of n elements without knowing the permutation of n-1 elements

Algorithm JohnsonTrotter(n)

// Implements Johnson-Trotter algorithm for generating permutation
// Input: A positive integer n
// Output: A list of all permutations of {1, ..., n}
initialize the first permutation with $\overleftarrow{1}\ \overleftarrow{2}\ \ldots\ \overleftarrow{n}$
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

Johnson Trotter n=3

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

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

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

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

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

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

Generate permutations of {1, 2, 3, 4} using Johnson Trotter algorithm

1. $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overleftarrow{4}$
2. $\overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}\ \overleftarrow{3}$
3. $\overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}\ \overleftarrow{3}$
4. $\overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}$
5. $\overrightarrow{4}\ \overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}$
6. $\overleftarrow{1}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{2}$
7. $\overleftarrow{1}\ \overleftarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}$
8. $\overleftarrow{1}\ \overleftarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}$
9. $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}\ \overleftarrow{4}$
10. $\overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}\ \overleftarrow{2}$
11. $\overleftarrow{3}\ \overleftarrow{4}\ \overleftarrow{1}\ \overleftarrow{2}$
12. $\overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{1}\ \overleftarrow{2}$
13. $\overrightarrow{4}\ \overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}$
14. $\overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}$
15. $\overrightarrow{3}\ \overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}$
16. $\overrightarrow{3}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}$
17. $\overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{4}$
18. $\overleftarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overleftarrow{1}$
19. $\overleftarrow{2}\ \overleftarrow{4}\ \overrightarrow{3}\ \overleftarrow{1}$
20. $\overleftarrow{4}\ \overleftarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}$
21. $\overrightarrow{4}\ \overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}$
22. $\overleftarrow{2}\ \overrightarrow{4}\ \overleftarrow{1}\ \overrightarrow{3}$
23. $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{4}\ \overrightarrow{3}$
24. $\overleftarrow{2}\ \overleftarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}$ → No mobile integer. Algorithm stops

Time complexity of Johnson is $O(n!)$.

Lexicographic order

If the permutation is listed in the increasing order, then it is called lexicographic order

Algorithm

Step1: [Generate the sequence in increasing order]
[1,2,3,4, ... n] which are denoted by $a_1, a_2, a_3 \ldots a_n$

Step2: [Scan the symbols from right to left and generate the lexicographic sequence]

if $a_i < a_{i+1}$ exchange a_i and a_{i+1}

if $a_i > a_{i+1}$, consider a_{i+1}

if $a_{i-1} < a_i$, replace a_{i-1} by next higher number on the right hand side and rearrange the remaining elements in the permutation in increasing order

Step-3: [Repeat through step 2 by considering the item a_{i-2}, a_{i-4} and so on]

Step 4: [Finished]
return.

eg

a_{i-1}	a_i	a_{i+1}
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

$\because (2<3)$ swap.

$\because (1<3)$ Replace 1 by next higher order term ie 2 & rearranging in ascending order

	a_1	a_2	a_3	a_4
1.	1	2	3	4
2.	1	2	4	3
3.	1	3	2	4
4.	1	3	4	2
5.	1	4	2	3
6.	1	4	3	2
7.	2	1	3	4
8.	2	1	4	3
9.	2	3	1	4
10.	2	3	4	1
11.	2	4	1	3
12.	2	4	3	1
13.	3	1	2	4
14.	3	1	4	2
15.	3	2	1	4
16.	3	2	4	1
17.	3	4	1	2
18.	3	4	2	1
19.	4	1	2	3
20.	4	1	3	2
21.	4	2	1	3
22.	4	2	3	1
23.	4	3	1	2
24.	4	3	2	1

Generating Subsets

→ The decrease-by-one idea can be applied to the knapsack problem.

→ All subsets of $A = \{a_1, \ldots, a_n\}$ can be divided into two groups: those that do not contain a_n and those that do.

→ The former group is nothing but all the subsets of $\{a_1, \ldots, a_{n-1}\}$ while each & every element of the latter can be obtained by adding a_n to a subset of $\{a_1, \ldots, a_{n-1}\}$

eg $\{a_1, a_2, a_3\}$

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

→ For generating permutations →

A convenient way of solving the problem is directly based on a one-to-one correspondence between all 2^n subsets of an n element set $A = \{a_1 \ldots a_n\}$ and all 2^n bit strings $b_1 \ldots, b_n$ of length n.

→ The easiest way to establish such a correspondence is to assign to a subset the bit string in which

$b_i = 1$ if a_i belongs to the subset and $b_i = 0$ if a_i does not belong to it.

eg bit strings 000 001 010 011 100 101 110 111

subsets ϕ $\{a_3\}$ $\{a_2\}$ $\{a_2 a_3\}$ $\{a_1\}$ $\{a_1, a_3\}$ $\{a_1, a_2\}$ $\{a_1, a_2 a_3\}$

There exists a minimal-change algorithm for generating bit strings so that every one of them differs from its immediate predecessor by only a single bit.

for eg $n = 3$

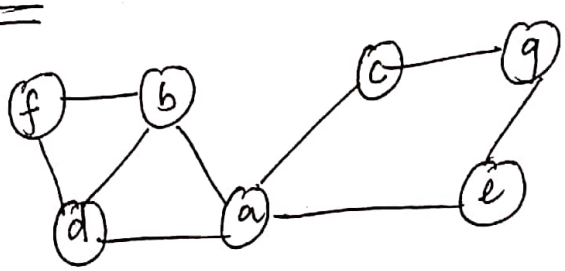
000 001 011 010 110 111 101 100

Such a sequence of bit strings is called the binary reflected Gray code.

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
edge types (undirected graphs)	tree & back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	minimum-edge paths, connectivity, acyclicity
Efficiency for adjacent matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
efficiency for adjacent lists	$\Theta(\|V\|+\|E\|)$	$\Theta(\|V\|+\|E\|)$

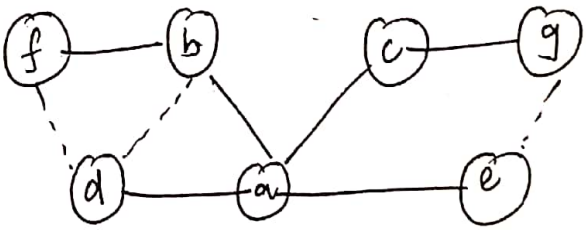
Eg

BFS



ordering

$a_1 b_2 c_3 d_4 e_5 f_6 g_7$.



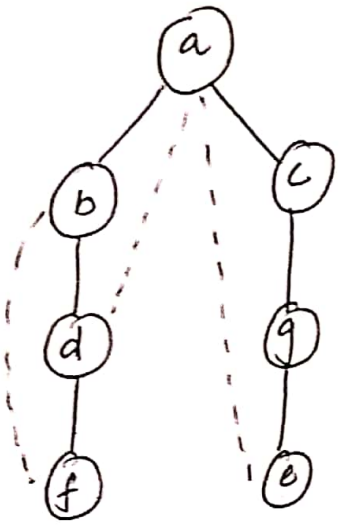
DFS

$f_{4,1}$ $e_{7,4}$

$d_{3,2}$ $g_{6,5}$

$b_{2,3}$ $c_{5,6}$

$a_{1,7}$



UNIT-III

Transform and Conquer

It is a method of solving a problem, by transforming problem instance into another simpler instance, using which solutions can be obtained. There are three major variations of this idea that differ by, what we transform at a given instance

1) Instance simplification

Here, transformation to a simpler or more convenient instance of the same problem.

eg AVL Trees, Presorting.

2) Representation change

In this strategy, one representation of a problem is changed into another representation of same instance

eg Heap Sort.

3) Problem reduction

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

eg finding the maximum of 'n' number using minimization function, finding the LCM of two numbers using GCD.

Presorting

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

Some problems whose efficiency can be improved using this technique.

1) Checking an element uniqueness in an array
2) Searching problem
3) Computing a 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 element
// 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
```

Analysis

Basic operation is comparison. → $T(n) = \sum_{i=0}^{n-2} 1$

$= n-2-0+1$

$= n-1$

$\therefore T_{scan}(n) = \Theta(n)$.

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

$\in \Theta(n \log n) + \Theta(n)$

$\in \Theta(n \log n)$

$T_{sort}(n) = \Theta(n \log n)$

↓

Mergesort

using property of asymptotic notation

$t(n) = \max\{\Theta(n), \Theta(n \log n)\}$

Note

Mergesort, time complexity is $\Theta(n \log n)$ in worst, Average & Best cases. Even though Quicksort has same time complexity in Best case & Average case, but in worst case it is $\Theta(n^2)$. So mergesort can be better choice

Representation change

Definition :-

A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met :-

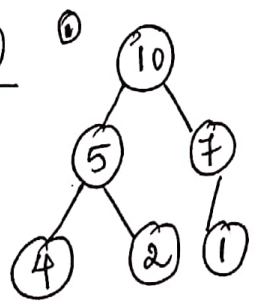
1] The tree's shape requirement :-

The binary tree is essentially complete (or simply complete) that is all its levels are full except possibly the last level, where only some rightmost leaves may be missing

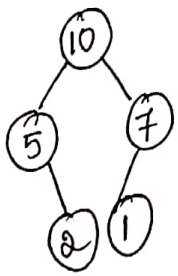
2] Parental Dominance requirement :-

The key at each node is greater than or equal to the key at its children.

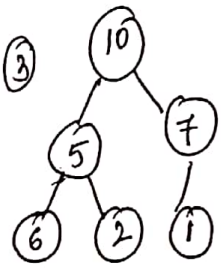
eg (1)



(2)



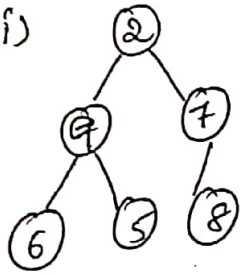
(3)

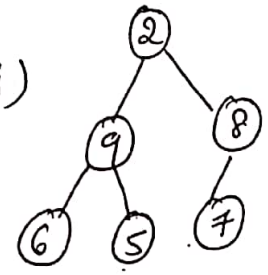


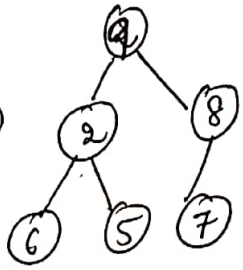
Here ① is a heap, because it satisfies both condition.
② is not a heap, because "Tree's shape requirement" is failed
and ③ is also not a heap because it satisfies first condition but violates "parental dominance requirement".

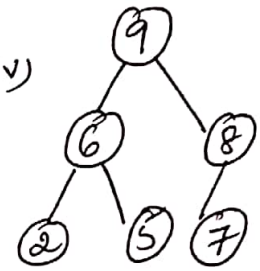
Bottom-up construction of heap :-

① 2, 9, 7, 6, 5, 8.

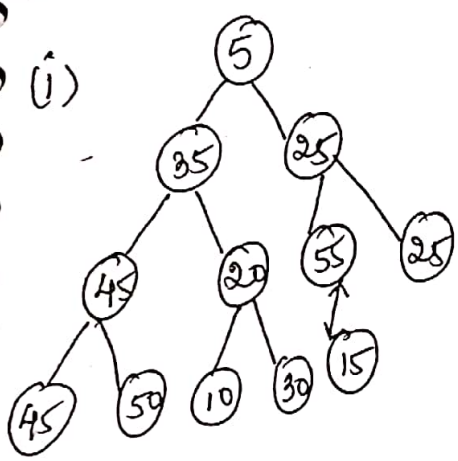
(i) 

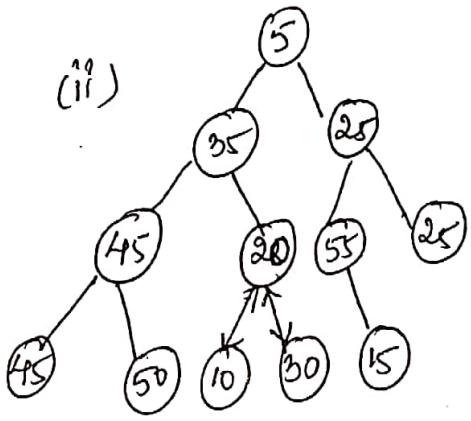
(ii) 

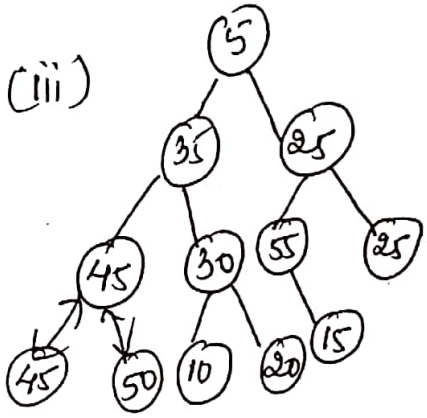
(iii) 

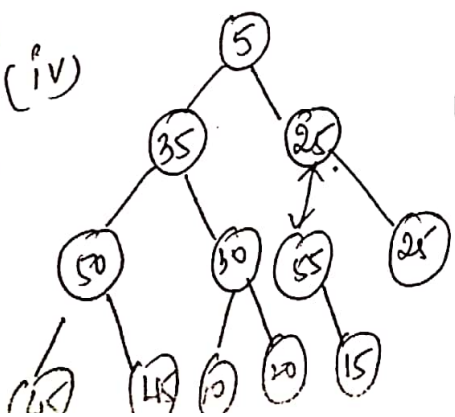
(iv) 

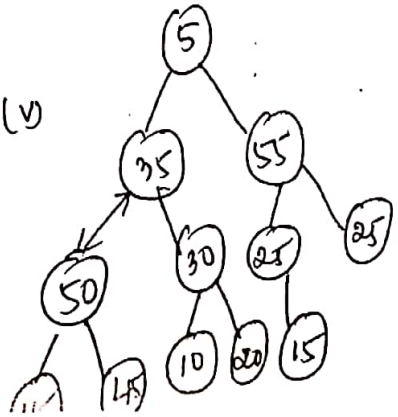
② 5, 35, 25, 45, 20, 55, 25, 45, 50, 10, 30, 15

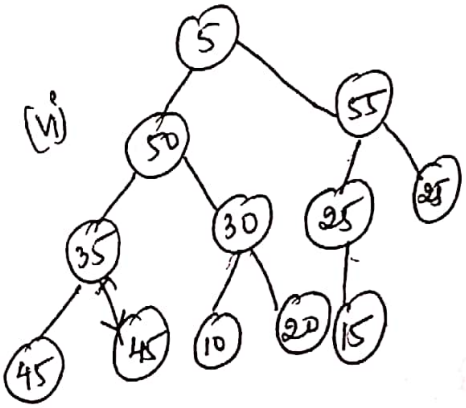
(i) 

(ii) 

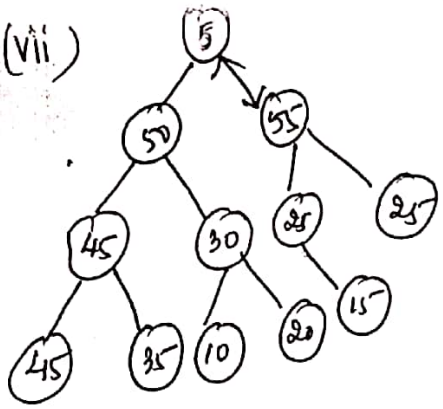
(iii) 

(iv) 

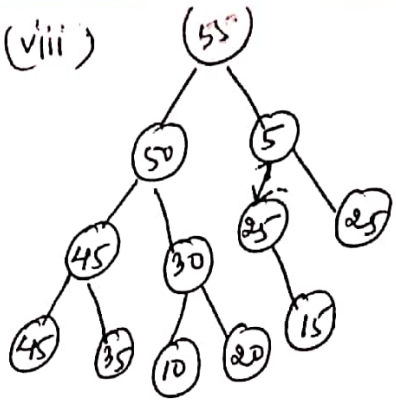
(v) 

(vi) 

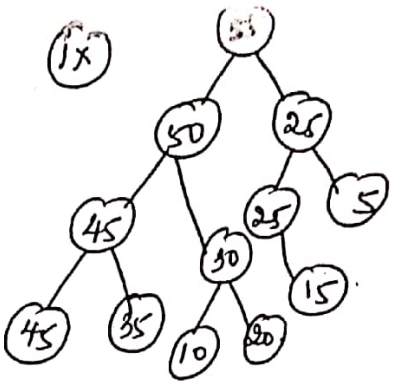
(vii)



(viii)



(ix)



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
```

Alternative Approach

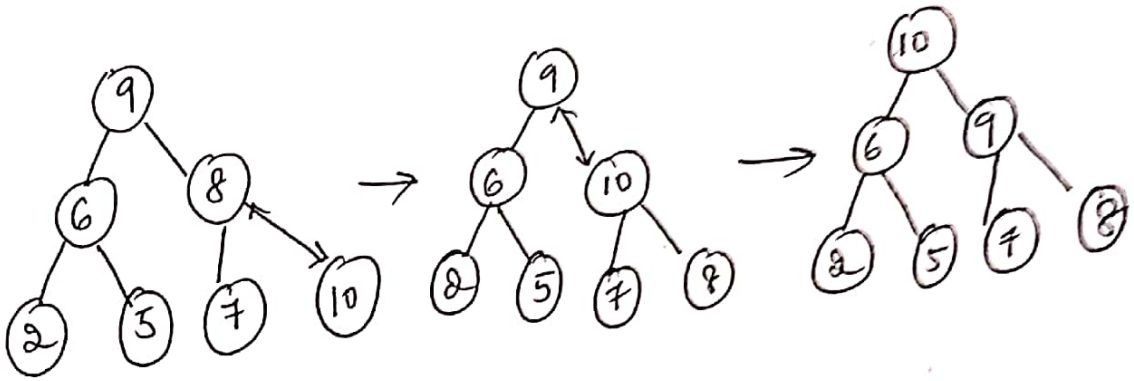
Top down heap construction

→ First attach a new node with key K in it after the last leaf of the existing heap. Then shift K up to its appropriate place in the new heap as follows:-

→ Compare K with its parent's key : if parent's is greater than or equal to K, stop otherwise swap these two keys & compare K with its new parent.

→ This swapping continues until K is not greater than its last parent or it reaches the root.

→ In this algorithm we can shift up an empty node, until it reaches its proper position, where it will get K's value.



Analysis

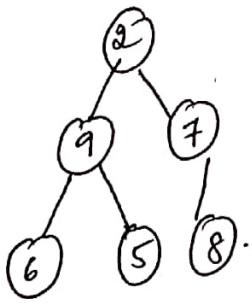
Obviously this insertion operation cannot require more key comparisons than heap's height.

Since height of a heap with n nodes is $\log_2 n$, the time efficiency of insertion is $O(\log n)$

2, 9, 7, 6, 5, 8

2	9	7	6	5	8

(i)



1	2	3	4	5	6
	9	8			7

$i \leftarrow 1$

same process

for $i \leftarrow 3$ to 1

$K \leftarrow 3$, $v \leftarrow H[3] \leftarrow 7$

heap = F

while ✗ heap & $2*3 \leq 6$ do

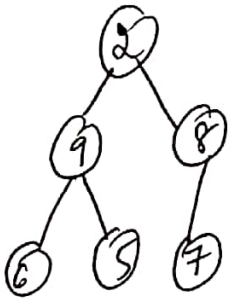
$j \leftarrow 2*3$

if $j < n$ ie ✗ there are no two children.

if $v \geq H[6]$ ie $7 \geq 8$ ✗

else $H[3] \leftarrow H[6]$; $K \leftarrow 6$

$H[6] \leftarrow 7$



parent

$i \leftarrow 2$ to 1

$K \leftarrow 2$, $v \leftarrow H[2] = 9$

heap ← false

while ✗ heap & $2*2 \leq n$

$j \leftarrow 2*2 = 4 < 6$ ✓ there are two children

children

if $H[4] < H[5]$ ✗

if $v \geq H[4]$ $9 > 6$ ✓

heap ← true ✓

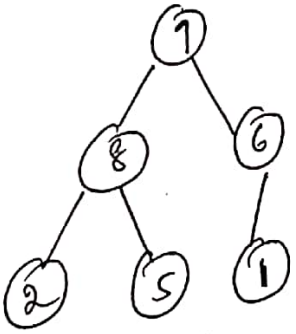
$H[2] \leftarrow 9$

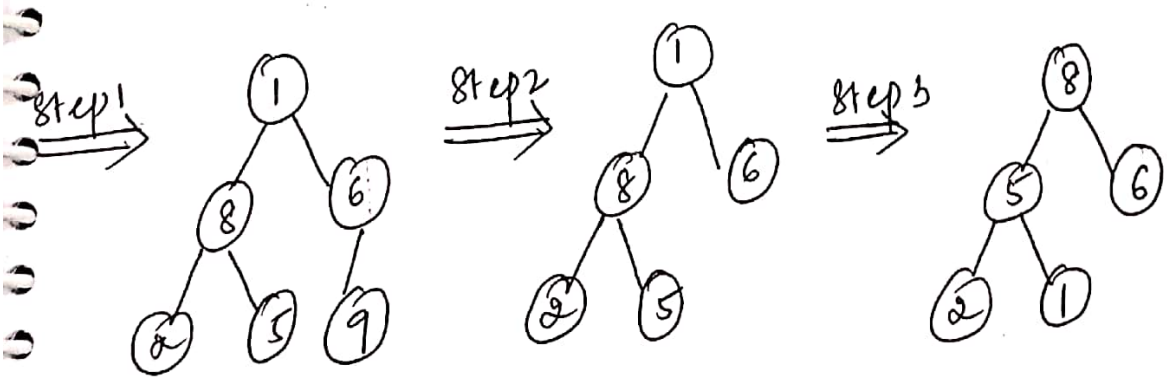
Maximum Key Deletion from a heap

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

Step 2: Decrease the heap's size by 1

Step 3: "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in bottom up heap construction. 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 the parental dominance condition holds for K in its new position.





Efficiency

The efficiency of deletion is determined by the no. of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Since it cannot require more key comparisons than twice the heap's height, the time efficiency of deletion is O(log n) as well

Heapsort

Heapsort is a two stage algorithm that works as follows :-

Stage 1 (heap construction): Construct a heap for a given array

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

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

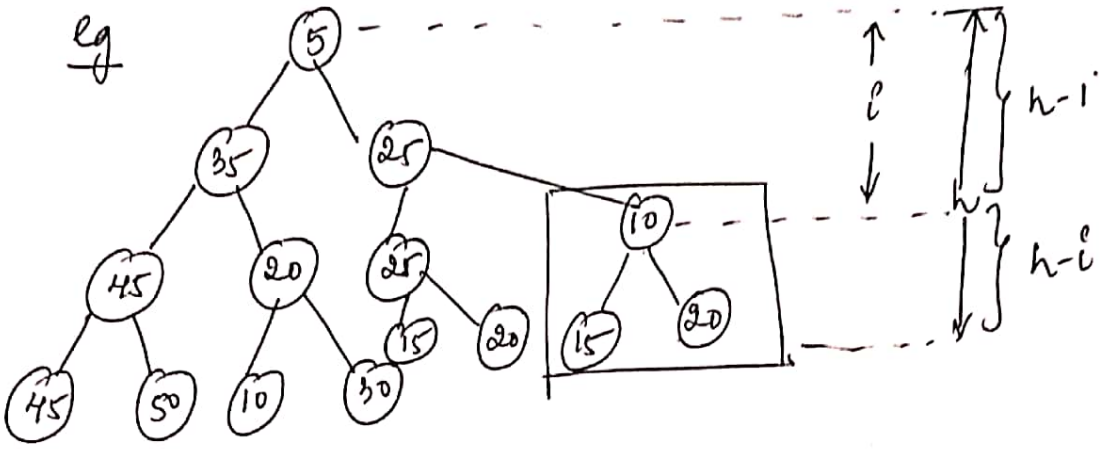
2 | 5

Analysis of heap sort using bottom-up approach.

Time efficiency of Heapsort = Time efficiency to create heap + Time efficiency to swap & recreate heap.

$$f(n) = f_1(n) + f_2(n).$$

eg



While creating a heap the no. of comparisons required to move an item from parent position to child position is 2 ie

→ First comparison to find largest child (ie 15 or 20)

→ Second comparison to determine whether parent should exchange with child node. (ie 10 and 20).

So total no. of comparisons required to move a node from level i to leaf ~~level~~ level in worst case is $2(h-i)$

Since the no. of nodes at level i is 2^i

∴ The total no. of comparisons required to move all these nodes from level i to leaf level is given by

$$2(h-i) * 2^i$$

- Starting from last non-leaf node at level i which is at a height of h-1 we move backwards till we get the root node which is at level 0. ∴ The total no. of key comparisons from level (h-1) to level 0 is given by,

$$f(n) = \sum_{i=0}^{h-1} 2(h-i) * 2^i$$

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

$$= 2 \left[\sum_{i=0}^{h-1} h * 2^i - \sum_{i=0}^{h-1} i * 2^i \right]$$

$$= 2 \left[h(2^0 + 2^1 + 2^2 + \ldots 2^{h-1}) - \left((h-2)2^h + 2 \right) \right]$$

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

$$= 2 \left[h+1 (2^h - 1) - h * 2^h + 2 * 2^h - 2 \right]$$

$$= 2 \left[h * 2^h - h - h * 2^h + 2 * 2^h - 2 \right]$$

$$= 2 \left[2 * 2^h - h - 2 \right] \quad \text{---(1)}$$

See appendix for $\sum_{i=1}^{n} i 2^i = (n-1)2^{n+1} + 2$

we know that if h is the maximum level of a complete binary tree, the relation between n and the no. of nodes in each level is given by the inequality :-

$n = 2^0 + 2^1 + 2^2 + \dots 2^h = 1\left(\frac{2^{h+1}-1}{2-1}\right)$ using $S = a\left[\frac{r^n - 1}{r-1}\right]$

$\therefore n = 2^{h+1} - 1$

$\therefore n+1 = 2^{h+1}$

Taking log on both sides

$\log_2 n+1 = h+1 \log_2 2 = h+1$

$\therefore h = \log_2(n+1) - 1$

Substituting in (1)

$f(n) = 2(2 * 2^h - h - 2]$

$= 2\left(2 * 2^{\log_2(n+1)-1} - \log_2(n+1) - 1\right) - 2\Big]$

$= 2\left[2 * \left(\frac{2^{\log_2(n+1)}}{2}\right) - \log_2(n+1) - 1\right]$

$= 2\left[2^{\log_2(n+1)} - \log_2(n+1) - 1\right]$

$= 2\left[(n+1)^{\log_2 2} - \log_2(n+1) - 1\right]$

$= 2\left[n+1 - \log_2(n+1) - 1\right]$

$= 2\left(n - \log_2(n+1)\right) \approx O(n)$

Time efficiency to swap & recreate heap

→ The relation between n & height is given by $h = \log_2 n$.

→ No. of comparisons required to move an item from parent to child position is 2

→ The re-creation of heap depends on height of the tree. So time complexity to swap & recreate the heap from n-1 down to 0 can be obtained as :-

Time efficiency to recreate the heap for (n-1) elements $= 2 \log_2 (n-1)$

Time efficiency to recreate the heap for (n-2) elements $= 2 \log_2 (n-2)$

Time efficiency to recreate the heap for $(n-3) = 2 \log_2 (n-3)$

⋮

for 3 elements $= 2 \log_2 3$

for 2 elements $= 2 \log_2 2$

for 1 element $= 2 \log_2 1$

$\therefore T(n) \leq 2 \log_2 1 + 2 \log_2 2 + \cdots 2 \log_2 (n-3) + 2 \log_2 (n-2) + 2 \log_2 (n-1)$

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

$\sum_{i=1}^{n} \log i \approx n \log n$

$\approx n \log_2 n$ for large value of n.

$\therefore T(n) = \theta(n) + \theta(n \log_2 n)$

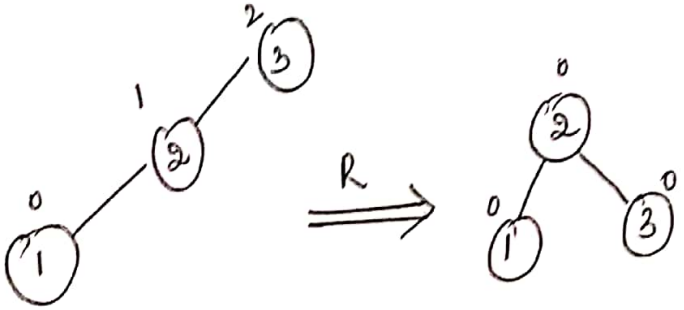
$= \theta(\max\{n, n\log_2 n\})$ using property of asymptotic notation

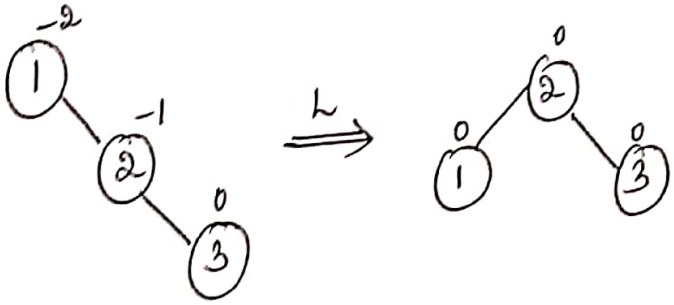
$$\boxed{T(n) = \theta(n \log_2 n).}$$

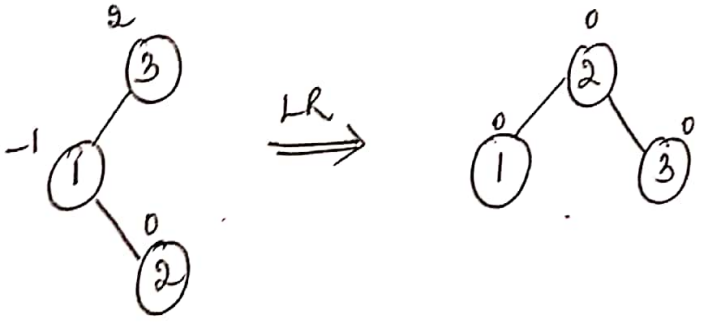
AVL Trees

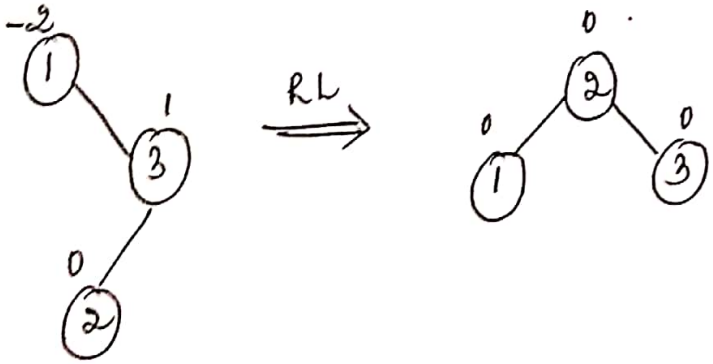
An AVL tree is a binary search tree in which the balance factor of every node which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or -1.

- The height of empty tree is defined as -1

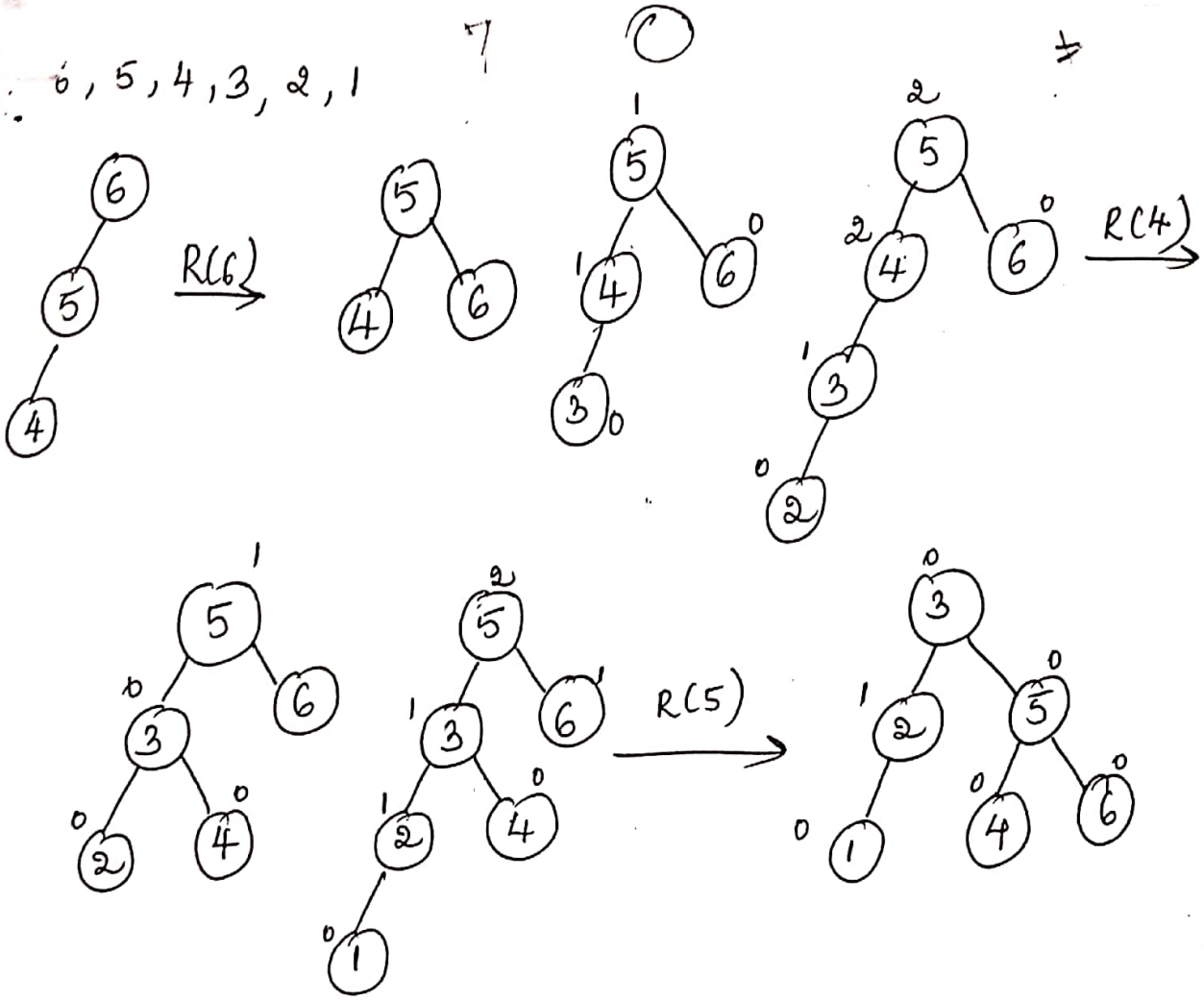




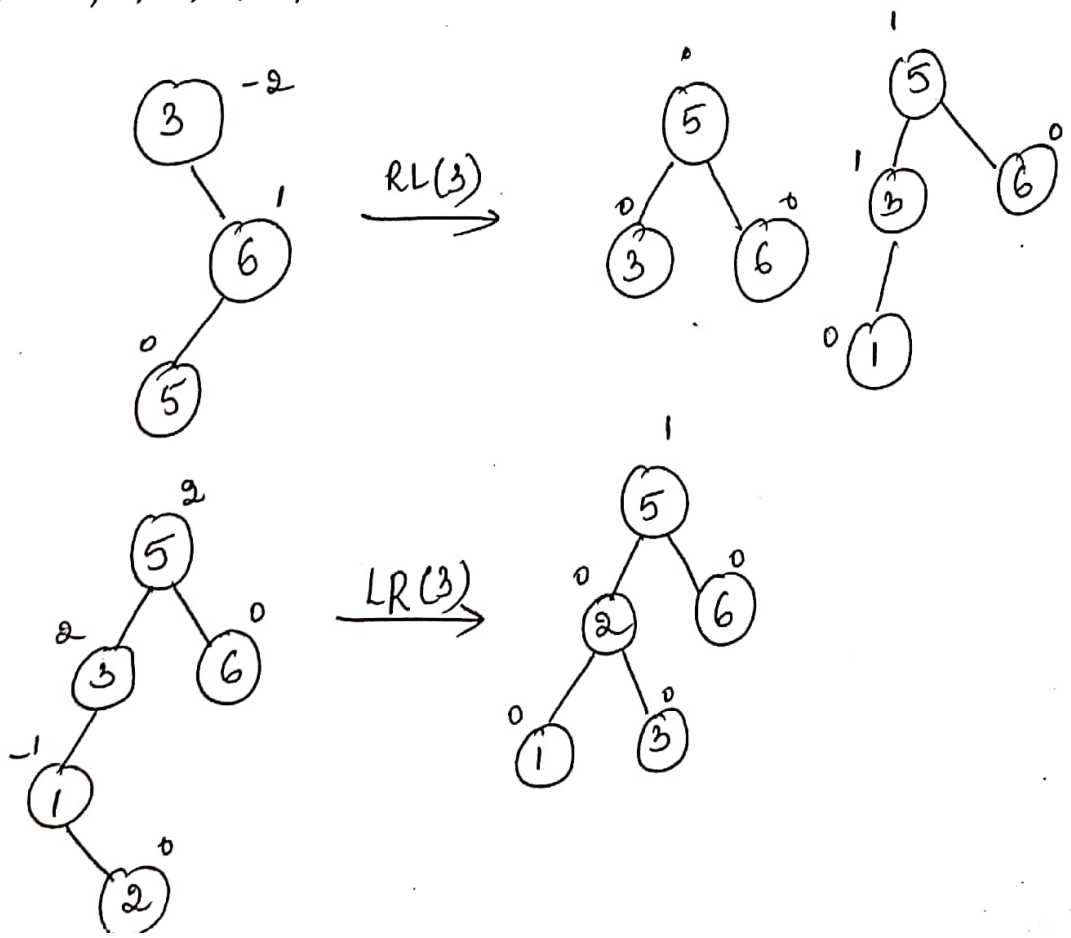


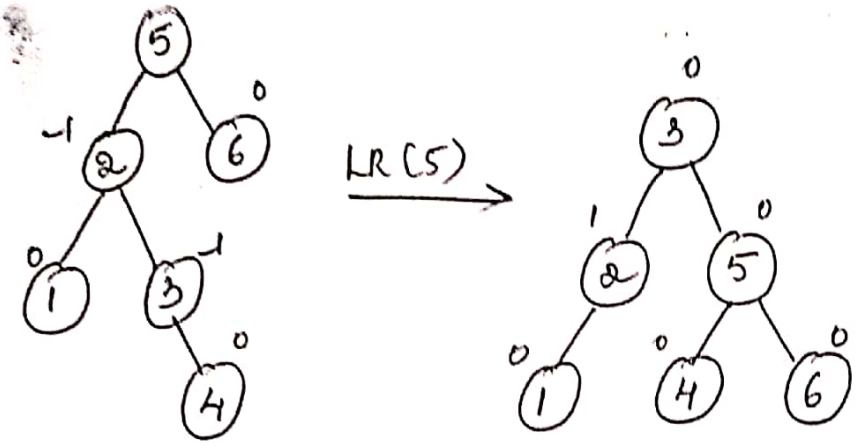


(2) 6, 5, 4, 3, 2, 1



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





Efficiency of AVL trees

The height 'h' of any AVL tree with 'n' nodes satisfies the inequalities

$$\lfloor \log_2 n \rfloor \leq h < 1.4405 \log_2 (n+2) - 1.3277$$

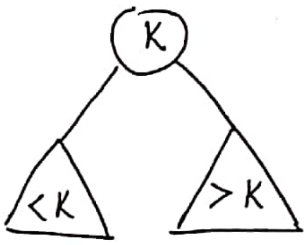
The inequalities immediately imply that the operations of searching & insertion are $\Theta(\log n)$ in the worst case.

2-3 Trees

→ A 2-3 tree is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

→ **2-node**

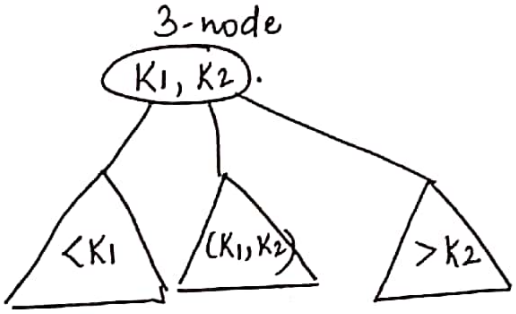
2-node.



A 2-node contains a single key K and has two children: the left child serves as the root of a subtree whose keys are ~~greater~~ less than K and the right child serves as the root of a subtree whose keys are less ~~greater~~ than K.

3-node

3-node



A 3-node contains two ordered keys K1 & K2 (K1 < K2) and has three children. The leftmost child serves as the root of a subtree with keys between K1 & K2 and the rightmost child serves as the root of a subtree with keys greater than K2.

→ For the 2-3 tree ~~that~~ all its leaves must be on the same level i.e a 2-3 tree is always perfectly

height balanced: the length of a path from the root of the tree to a leaf must be same for every leaf.

Searching for a given key K in 2-3 tree

→ We start at the root.

→ If the root is 2-node, we act as if it were a binary search tree: we either stop if K is equal to the root's key or continue the search in the left or right subtree if K is, respectively, smaller or larger than the root's key.

→ If root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if K is equal to one of the root's keys) or in which of the root's three subtrees it needs to be continued.

Inserting a new key in 2-3 tree :-

1) We always insert a new key K in a leaf, except for the empty tree.

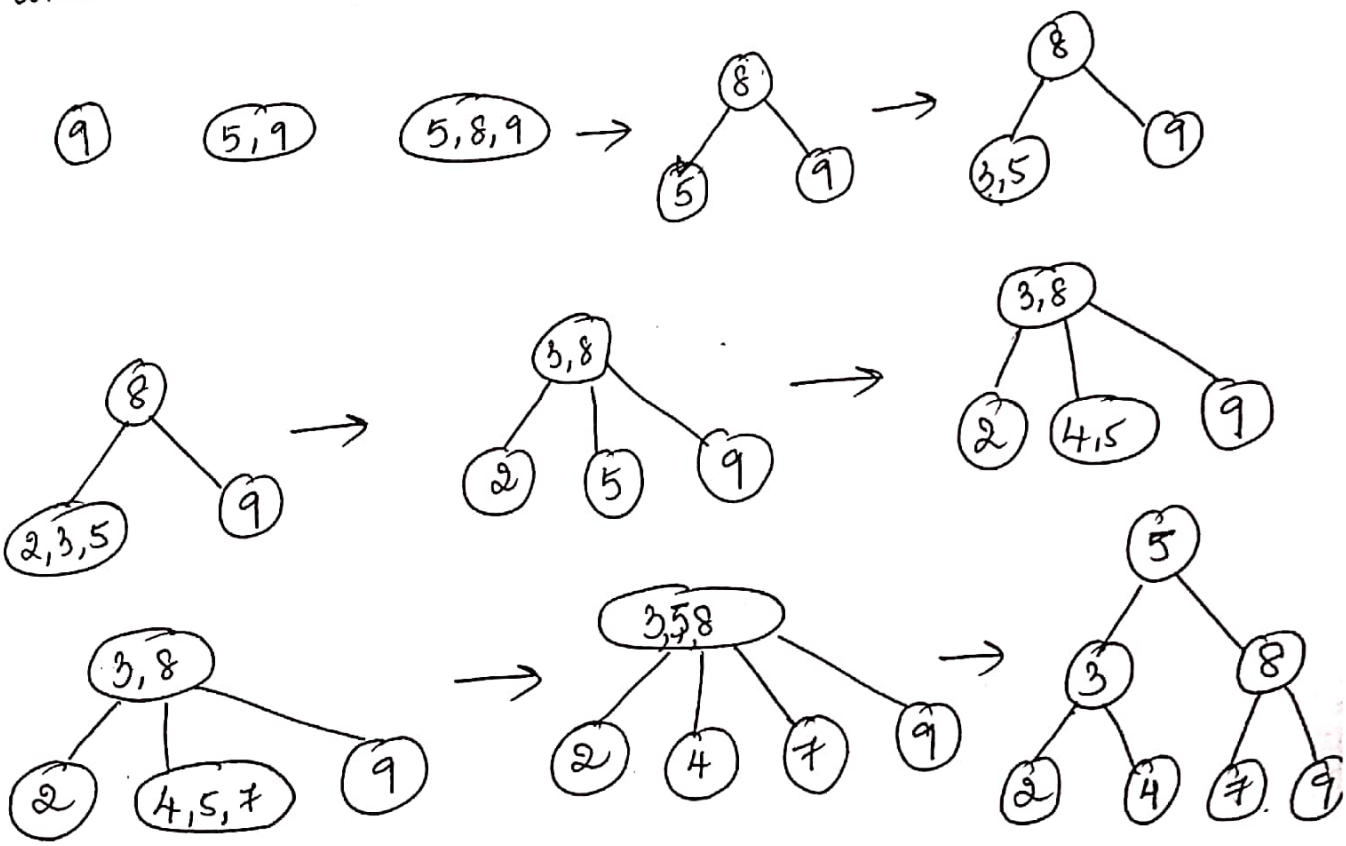
2) The appropriate leaf is found by performing a search for K.

3) If the leaf in question is a 2-node, we insert K there as either first or second key depending on whether K is smaller or larger than node's old key.

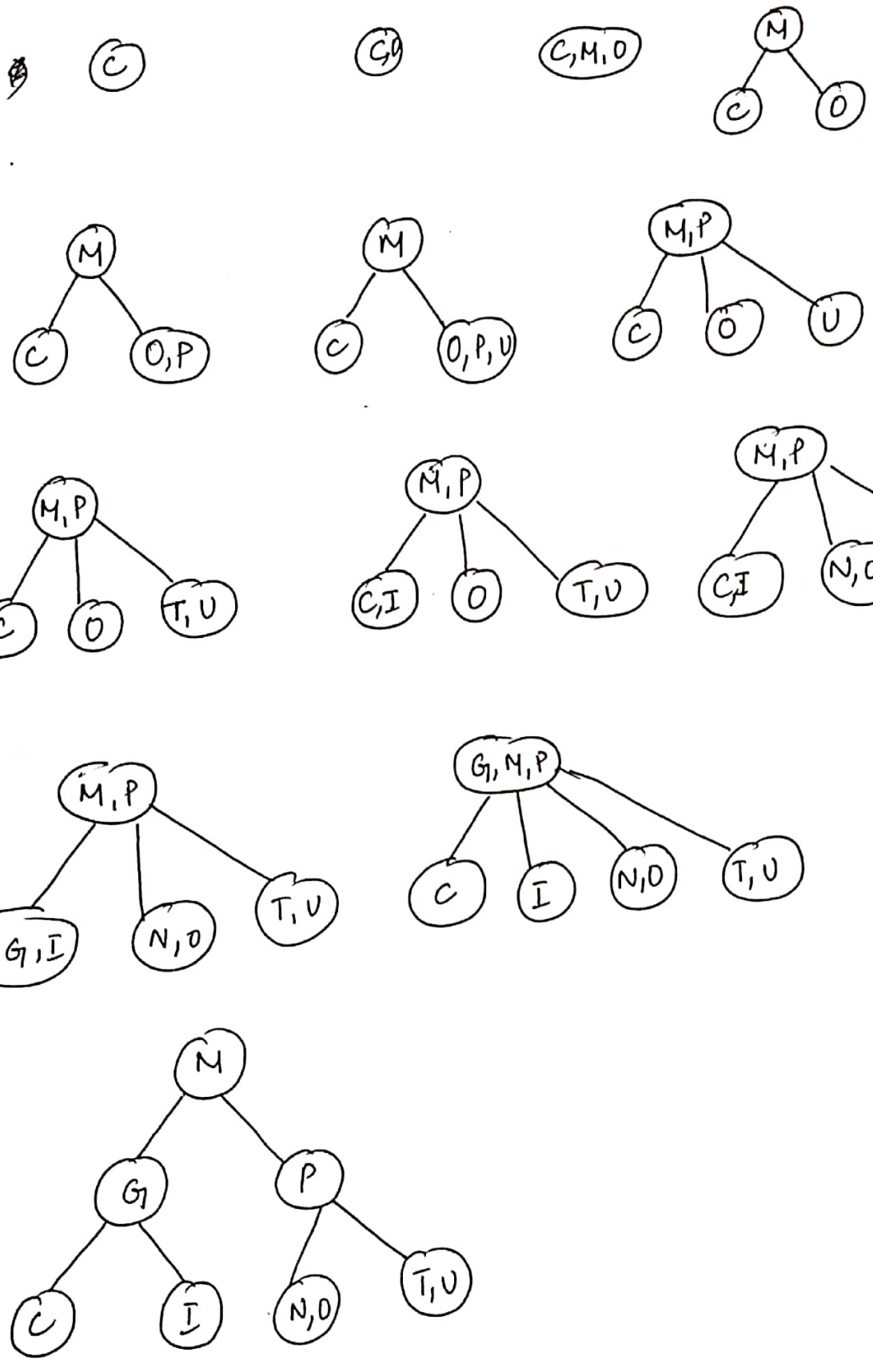
- If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, while the middle key is promoted to the old leaf's parent. (If the leaf happens to be the tree's root, a new root is created to accept the middle key).

- Note that promotion of a middle key to its parent can cause the parent's overflow (if it was 3-node) and hence can lead to several node splits along the chain of the leaf's ancestors.

Q Construct a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.



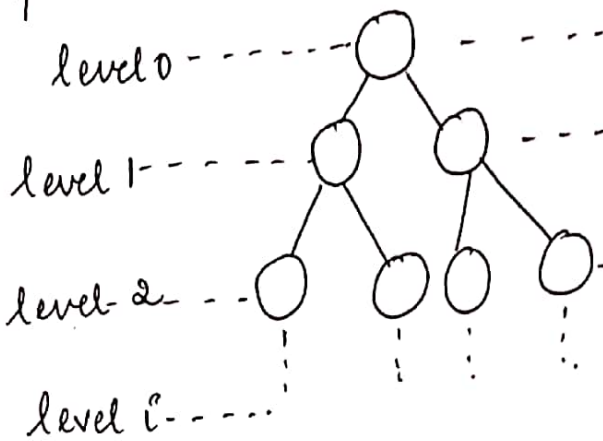
Q Construct 2-3 tree for the list C, O, M, P, U, T, I, N, G



Analysis

To find the upper bound for 2-3 tree :-

Consider 2-3 tree with each node exhibiting the property of 2-nodes.



Let us assume that each node has only one item
Now the relation between the total no. of items n and no. of nodes occuring in each level from level 0 to level i is given by :-

$$n = 2^0 + 2^1 + 2^2 + \dots 2^i$$

But the leaf nodes may have one or two items and the height still remains same.
In such situation, the relation between the no. of items & no. of nodes is given by the inequality :-

$$n \geq 2^0 + 2^1 + 2^2 + \dots 2^i = 1\left(\frac{2^{i+1}-1}{2-1}\right)$$ using formula $S = a\left(\frac{r^n - 1}{r-1}\right)$

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

(where $h = i+1$ ie last level + 1 = height of the tree)

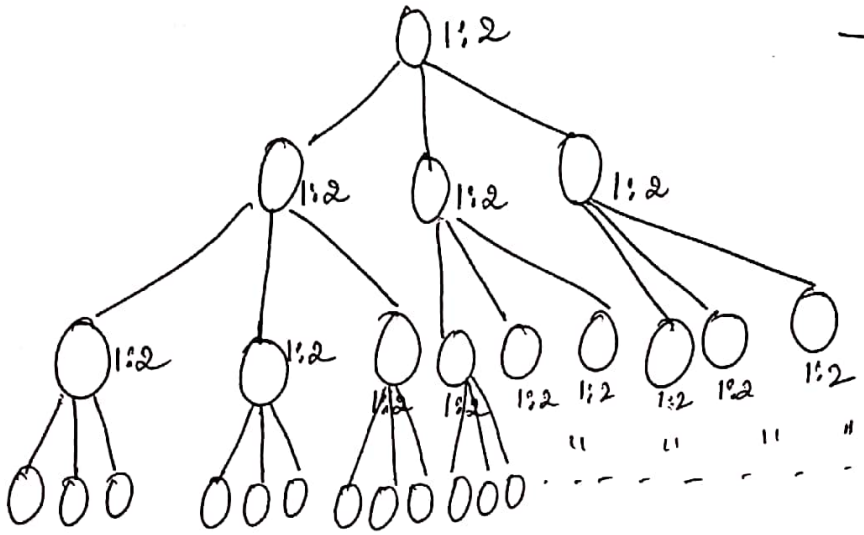
$\therefore n+1 \geq 2^h$

So taking log on both sides we get $\log_2 n + 1 \geq h$

So the relation between height of the tree and the no. of nodes n in a tree is given by :-

$$\boxed{h \leq \log_2 n + 1} \quad \text{---------- (1)}$$

Case2: To find lower bound for 2-3 tree



level	no. of items	items in terms of level
0	2	$2*3^0$
1	6	$2*3^1$
2	18	$2*3^2$
3	54	$2*3^3$
i		$2*3^i$

The total no. of items in each level by assuming each node has 2 items is given by the relation :-

$$n = 2*3^0 + 2*3^1 + 2*3^2 + 2*3^3 + \cdots 2*3^i$$

In case of some node having less than 2-items then the relation is given by :-

$$n \leq 2*3^0 + 2*3^1 + 2*3^2 + 2*3^3 + \cdots 2*3^i$$

$$n \leq 2(3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^i)$$

$$n \leq 2*1\left(\frac{3^{i+1}-1}{3-1}\right)$$ using the formula $S = a\left(\frac{r^n - 1}{r-1}\right)$

$n \leq 3^{i+1} - 1$

$n+1 \leq 3^{i+1}$

$n+1 \leq 3^h$. h = total no. of levels + 1

Taking log on both sides.

$\log_3(n+1) \leq h * \log_3 3$.

$$\boxed{h \geq \log_3(n+1)} \quad \text{——} \quad (2)$$

From equation (1) and (2).

$$\boxed{\log_3(n+1) \leq h \leq \log_2(n+1)}$$

So the Time complexity for 2-3 tree is $\underline{\theta(\log n)}$.