S. No.	Date	Title	Page No.	Teacher's Sign / Remarks

classmate
Date
Page

www.ankurgupta.net

Design and Analysis of Algorithms + Data Structures

☆ These notes have been prepared from the following two books :-

(1) Introduction to Algorithms by Cosman, Leiserson, Rivest, Stein

(2) Data Structures by Seymour Lipschutz

Please read the above books for more details.

Growth of Functions :-

(1) Θ-notation :- (Asymptotic tight bound)

$\Theta(g(n)) = \{f(n)$: there exist positive consts. c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$

(2) O-notation :- (Asymptotic upper bound)

$O(g(n)) = \{f(n)$: there exist +ve const c & n_0 such that $0 \le f(n) \le c g(n)$ for all $n \ge n_0\}$

Big O

$$f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$$
and $$\Theta(g(n)) \subseteq O(g(n))$$

$f(n) = O(n^2)$, it means the worst case running time is $O(n^2)$.

(3) Ω-notation :- (Asymptotic lower Bound)

$\Omega(g(n)) = \{f(n)$: there exist +ve const c & n_0 such that $0 \le c g(n) \le f(n)$ $\forall\ n \ge n_0\}$

Big-omega

It means that for all values of n greater than or equal to n_0, the value $f(n)$ is on or above $c g(n)$.

$f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

$O(g(n)) \cap \Omega(g(n)) = \theta(g(n))$

(4) o-notation :- (Upper bound that is not asymptotic tight.)

$o(g(n)) = \{f(n)$: for any +ve const. $c > 0$, there exist a const $n_0 > 0$ such that $0 \le f(n) < c(g(n))$ $\forall\ n \ge n_0\}$

E.g. :- $2(n) = O(n^2)$; but $2n^2 \ne o(n^2)$
(small-o) (not small o)

The definitions of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $0 \le f(n) \le cg(n)$ holds for some const $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \le f(n) < cg(n)$ holds for all constants $c > 0$.

(5) ω-notation :- (Lower bound that is not asymptotic tight.)

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

$\omega(g(n)) = \{f(n)$: for any +ve const $c > 0$, there exists a const $n_0 > 0$ such that $0 \le cg(n) < f(n)$ for all $n \ge n_0\}$

$o(g(n)) \cap \omega(g(n)) = \phi$

Some important results :-

Transitivity :-

$f(n) = \theta(g(n))$ and $g(n) = \theta(h(n))$ imply $f(n) = \theta(h(n))$
→ for all $\theta, O, \Omega, o, \omega$

Reflexivity :-

$f(n) = \theta(f(n))$
$f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

Symmetry :-

$f(n) = \theta(g(n))$ iff $g(n) = \theta(f(n))$

Transpose Symmetry :-

$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

$f(n) = O(g(n)) \approx a \leq b$

$f(n) = \Omega(g(n)) \approx a \geq b$

$f(n) = \Theta(g(n)) \approx a = b$

$f(n) = o(g(n)) \approx a < b$

$f(n) = \omega(g(n)) \approx a > b$

Important Examples:-

(1) $(n+a)^b = \Theta(n^b)$

a & b are const and $b > 0$

(2) $2^{n+1} = O(2^n)$

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

(3) Logarithms:-

$\lg^k n = (\lg n)^k$

$\lg \lg n = \lg(\lg(n))$

$\log_b(1/a) = -\log_b a$

$\log_b a = \frac{1}{\log_a b}$

$a^{\log_b c} = c^{\log_b a}$

Functional Iteration:-

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n.

Iterated Logarithm Function:-

It is defined as:-

$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$

Examples:-

$\lg^* 2 = 1$

$\lg^* 4 = 2$

$\lg^* 16 = 3$

$\lg^* 65536 = 4$

$\lg^*(2^{65536}) = 5$

Recurrences :-

Master Method :-

Used for solving recurrences of the form :-

$$T(n) = a\,T(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

Solution :-

(1) If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then, $T(n) = \theta(n^{\log_b a})$

(2) If $f(n) = \theta(n^{\log_b a})$, then,

$$T(n) = \theta(f(n) \lg n)$$

(3) If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a f(n/b) \leq c f(n)$ for $c < 1$, and all sufficiently large n, then :-

$$T(n) = \theta(f(n))$$

Here $f(n)$ must be polynomially smaller or greater than $n^{\log_b a}$ in case (1) & (3).

Changing Variables :-

$$T(n) = 2T(\sqrt{n}) + \lg n$$

Put $n = 2^m$

$$\Rightarrow T(2^m) = 2T(2^{m/2}) + m$$

Put $T(2^m) = S(m)$

$$\Rightarrow S(m) = 2S(m/2) + m$$

Using Master's Method :-

$$S(m) = \theta(m \lg m)$$

Putting $m = \lg n$

$$\Rightarrow T(2^m) = \theta(m \lg m)$$

$$\Rightarrow T(n) = \theta(\lg n \lg \lg n)$$

Sorting

Insertion Sort :-

Best Case Running Time = $O(n)$

Worst Case Running Time = $O(n^2)$

Average Case Running Time = $O(n^2)$

Maximum no. of swaps required = $O(n^2)$

Merge Sort :-

$T(n) = 2\,T(n/2) + \Theta(n)$ (MERGE-SORT / MERGE)

Worst & Average Case Complexity = $\Theta(n \lg n)$

Bubble Sort :-

Average & Worst Case Complexity = $O(n^2)$

Selection Sort :-

Worst Case Complexity = $O(n^2)$

Average Case Complexity = $O(n^2)$

Max swaps required = $O(n)$

HEAP SORT :-

MAX-HEAPIFY(A, i) :-

Used for maintaining the heap property. Its input are an array A and an index i into the array. When MAX-HEAPIFY is called, it is assumed that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] may be smaller than its children, thus violating the max-heap property. The function of MAX-HEAPIFY is to let the value at A[i] "float down" in the max heap so that the subtree rooted at index i becomes a max heap.

Running time = $O(\lg n)$

for a node of height h :-

Running time = $O(h)$.

BUILD-MAX-HEAP :-

Used for building a max heap from an array. It applies MAX-HEAPIFY on all internal nodes from last to root node.

Running time = $O(n)$

HEAPSORT :-

Used for sorting the elements of the heap. The resultant array is in non-decreasing order.

Running time = $O(n \lg n)$

HEAP-EXTRACTS-MAX :-

Removes the maximum element of the heap and returns it.

Running time = $O(\lg n)$

HEAP-INCREASE-KEY(A, i, key) :-

It increases the value at location i by the value key and then traverses the heap upward to maintain heap property.

Running Time = $O(\lg n)$

MAX-HEAP-INSERT :-

Inserts the element at the end of the heap and then traverses upward to maintain heap property.

Running Time = $O(\lg n)$.

Don't confuse in Quick & radix sort

Quick Sort Algorithm :-

It uses divide and conquer approach. It uses PARTITION algorithm to partition the array into two sub arrays. The PARTITION algo selects the last element of the input array as a pivot element and partition the array such that the elements to the left are smaller than or equal to the pivot element and the elements to the right of pivot element are greater than it.

Performance :-

The running time of quicksort depends on whether the partitioning is balanced or unbalanced.

Worst Case Partitioning :-

The worst case occurs when the partitioning routine produces one subproblem with n-1 elements and one with 0 elements.

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$
$$= O(n^2)$$

Worst Case occurs when the i/p array is already sorted. or all the elements are same.

Best Case Partitioning :-

$$T(n) \leq 2T(n/2) + \theta(n)$$

$$T(n) = O(n \lg n)$$

Balanced Partitioning :-

* The average-case running time of quicksort is much closer to the best case than to the worst case.

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

Running Time = $O(n \lg n)$.

Counting Sort :-

It assumes that each of n input elements is an integer in the range 0 to k, for some integer k, where $k = O(n)$.

Running Time = $\theta(n+k)$

$= \theta(n)$

* It is a stable sorting algorithm. This property is generally used for sorting satalite data.

Radix Sort :-

It uses stable sorting algorithm.

Running Time = $\theta(d(n+k))$

where d = no. of digits

$k = 10$ [0, 1, ... 9]

If d is constant and $k = O(n)$

Running time = $O(n)$ → if d is constant

Bucket Sort :-

Divide the interval [0, 1) into n equal-sized subintervals or buckets and then distribute the n i/p no. in the buckets. Here i/p no. are in range [0, 1).

Running Time = $\theta(n)$

Graph Algorithms

Representation of Graphs :- G(V, E)

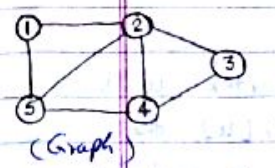
(1) Adjacency list representation :-
Used to represent sparse graphs, i.e. those for which, $|E| << |V|^2$

(2) Adjacency Matrix Representation :-
Used for representing dense graphs, i.e. $|E| \approx |V|^2$.

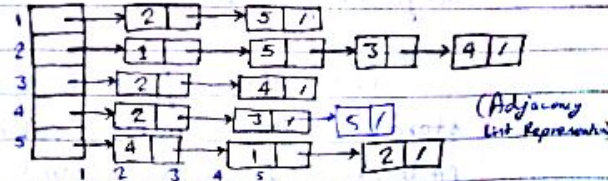
For Adjacency List Representation, if G is a directed graph, the sum of all the adjacency lists is |E| and if G is a undirected graph, the sum is 2|E|.

Amount of memory required for Adjacency List Representation = $\theta(V+E)$.

Transpose of a directed graph G = (V, E) is the graph $G^T = (V, E^T)$.



(Graph)


(Adjacency List Representation)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(Adjacency Matrix Representation)

Breadth First Search :-

Used for searching a graph.

⇒ Prim's minimum-spanning tree algorithms and Dijkstra's single source shortest paths algorithm use ideas similar to those in BFS.

⇒ Produces a breadth-first tree with root s that contains all reachable vertices.

⇒ It works on both, directed and undirected graphs.

⇒ To keep track of progress, breadth first search colors each vertex white, gray or black. All vertices start out white and may later become gray and then black.

⇒ It assumes that the input graph $G = (V, E)$ is represented using adjacency lists.

⇒ The color of each vertex u is stored in the variable color[u], the predecessor of u is stored in variable $\pi[u]$. The distance from the source s to vertex u is stored in d[u]. The algo also uses a FIFO queue Q to manage the set of gray vertices.

Running Time of BFS = O(V+E)

Depth First Search :-

⇒ Produces a depth-first forest composed of several depth first trees.

⇒ Each vertex is initally white, is grayed when it is discovered in the search, and is blackend when it is finished.

⇒ It timestamps each vertex. Each vertex v has two timestamps: the first timestamp d[v] records when v is first discovered (and grayed), and the second timestamp f[v] records when the search finishes examining v's adjacency list (and blackens v).

⇒ Vertex u is white before d[u], gray b/w d[u] and f[u], and black thereafter.

⇒ Contains two methods DFS (G) and DFS-VISIT (u).

A node in a graph is called articulation point, if removal of this node disconnects the graph.

Running Time :-

Running time of DFS = $\Theta(V+E)$

Topological Sorting :-

→ Only for DAG.

Running Time = $\Theta(V+E)$.

Possible only if the graph is acyclic. Put vertices at the head of linked list in order of increasing finishing

Strongly Connected Components :-

Running Time = $\Theta(V+E)$.

→ A graph is strongly connected if for every pair of vertices u, v, $u \rightsquigarrow v$ and $v \rightsquigarrow u$, i.e. u & v are reachable from each other.

STRONGLY-CONNECTED-COMPONENTS(G)

(1) Call DFS(G) to compute finishing times $f[u]$ for each vertex u.
(2) Compute G^T
(3) Call DFS(G^T), but in main loop of DFS, consider the vertices in order of decreasing $f[u]$.
(4) Off the vertices of each tree in the depth first forest formed in line 3 as a strongly connected component.

Minimum Spanning Trees :-

Kruskal's and Prim's Algorithms are used to find minimum spanning Trees from a weighted graph.
⇒ Both are greedy algorithms.

Kruskal's Algorithm :-

It is a greedy algorithm, because at each step it adds to the forest an edge of least possible weight

⇒ It uses a disjoint set data structure to maintain several disjoint sets of elements.

⇒ Running time of Kruskal's algorithm for a graph $G = (V, E)$ depends on the implementation of the disjoint set data structure.

Running time = $O(E \lg E)$
Since $|E| < |V|^2$
$\Rightarrow \lg |E| = O(\lg V)$.

⇒ Running time = $O(E \lg V)$

Prim's Algorithm :-

⇒ It operates much like Dijkstra's algorithm for finding shortest paths in a graph.

⇒ The edges in the set A always form a single tree. The tree starts from an arbitrary vertex and grows until the tree spans all the vertices in V.

Use it to make spanning tree

⇒ At each step, a light edge is added to the tree A, that connects A to an isolated vertex of $G_A = (V, A)$.

⇒ This strategy is greedy since the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree's weight.

⇒ The performance of Prim's algorithm depends on how we implement the min-priority queue Q.

If we use binary min heap then :-

Running Time = $O(V \lg V + E \lg V)$

= $O(E \lg V)$

If we use Fibonacci heap then :-

Running Time = $O(E + V \lg V)$

Single Source Shortest Paths :-

The breadth first search is a shortest-paths algorithm that works on unweighted graphs.

Shortest-paths algorithms typically rely on the property that a shortest path between two vertices contains other shortest paths within it.

Dijkstra's algorithm is a greedy algorithm, while Floyd-Warshall algorithm is a dynamic programming algorithm.

If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$

Dijkstra's algorithm assume that all edge weights in the i/p graph are non-negative.

Bellman Ford algorithm, allow negative-weight edges in the i/p graph and produce a correct answer as long as no-negative-weight cycles are reachable from the source If there is a negative weight cycle, the algo can detect and report its existence.

Bellman-Ford Algorithm :-

Solves the single source shortest paths problem in which edge weights may be negative.

It uses relaxation to find shortest path. It returns true iff the graph contains no-negative-weight cycles that are reachable from source.

Running time = $O(VE)$

(This algo does not find any negative weight cycle, it only finds whether a -ve weighted cycle is reachable from the source or not.)

Single Source Shortest Paths in directed acyclic graphs :-

By relaxing the edges of a weighted DAG according to a topological sort of its vertices, we can compute shortest paths from a single source in $\theta(V+E)$ time.

Dijkstra's Algorithm :-

Solves the single source shortest paths problem in which all edge weights are non-negative.

It maintains a set S of vertices whose shortest-path weights from the source s have been determined. The algo repeatedly selects the vertex $u \in V-S$ with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.

The running time of Dijkstra's algorithm depends on how the min-priority queue is implemented.

Running Time :-

(i) Using Binary Min-heap :-

$$O((V+E)\lg V) \equiv O(E \lg V)$$

(ii) Using Fibonacci Min-heap :-

$$O(E + V \lg V)$$

Data Structures

Arrays :-

A linear array is a list of a finite number n of homogeneous data elements such that :-

(a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.

(b) The elements of the array are stored respectively in successive memory locations.

Length of the array :-

$$\text{Length} = UB - LB + 1$$

Representation of Linear Array in Memory :-

$$LOC(LA[K]) = Base(LA) + w(K - LB)$$

where, w → no. of words/memory cell
K → Index of element in array
LOC → Location of an element

Representation of 2-D Arrays in memory :-

(a) Row Major Order :-

$$LOC(A[J,K]) = Base(A) + w[N(J-1) + (K-1)]$$

(b) Column Major Order :-

$$LOC(A[J,K]) = Base(A) + w[M(K-1) + (J-1)]$$

where, A is M × N array
Base(A) is base address of A.

Stacks :-

In a stack, the element removed from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO policy.

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP.

We can implement a stack of at most n elements with an array S[1,..n]. The array has an attribute top[S] that indexes the most recently inserted element. When top[S] = 0, the stack contains no elements and is empty.

Arithmetic Expressions :-

(a) Infix Notation :-

A+B, C-D, E*F, G/H

(b) Polish Notation or Prefix Notation :-

+AB, -CD, *EF, /GH

(c) Reverse Polish Notation or Postfix Notation :-

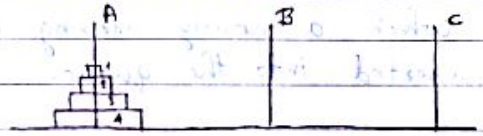
AB+, CD-, EF*, GH/

Tower of Hanoi Problem :-

Suppose three pegs are labeled A, B and C and suppose on peg A, there are placed a finite no. n of disks with decreasing size. The object of the game is to move the disks from peg A to peg C using peg B as an auxilary. The rules of the game are as follows :-

(a) Only one disk may be moved at a time, i.e. the top disk on any peg.

(b) At no time a larger disk be placed on a smaller disk.



Algorithm :-

TOWER(N, BEG, AUX, END)

1. If N=1, then:
   (a) Write BEG → END
   (b) Return.
2. Call TOWER(N-1, BEG, END, AUX)
3. Write BEG → END
4. Call TOWER(N-1, AUX, BEG, END)
5. Return.

Complexity :-

$$T(n) = 2T(n-1) + \theta(1)$$
$$= \theta(2^n)$$

Queue :-

In a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO policy.

The insert operation of a queue is called ENQUEUE and the delete operation is called DEQUEUE.

The queue has an attribute head[Q] that indexes, or points to its head. The attribute tail[Q] indexes the next location at which a newly arriving element will be inserted into the queue.

The elements in the queue are in locations head[Q], head[Q]+1, ..., tail[Q]-1, where we "wrap around" in the sense that location 1 immediately follows location n in a circular order.

When head[Q] = tail[Q], the queue is empty.

Initially, we have head[Q] = tail[Q] = 1.

When head[Q] = tail[Q]+1, the queue is full.

Linked Lists :-

A linked list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

A doubly linked list L is an object with a key field and two other pointer fields: next and prev. Given an element x in the list, next[x] points to its successor in the linked list, and prev[x] points to its predecessor. If prev[x] = NIL, the element x has no predecessor and is therefore the first element, or head, of the list. If next[x] = NIL, the element x has no successor and is therefore the last element, or tail, of the list. An attribute head[L] points to the first element of the list. If head[L] = NIL, the list is empty.

If a list is singly linked, we omit the prev pointer in each element.

If a list is sorted, the linear order of the list corresponds to the linear order of keys stored in elements of the list; the minimum element is the head of the list, and the maximum element is tail.

In a circular list, the prev pointer of the head of the list points to tail, and the next pointer of the tail of the list points to the head.

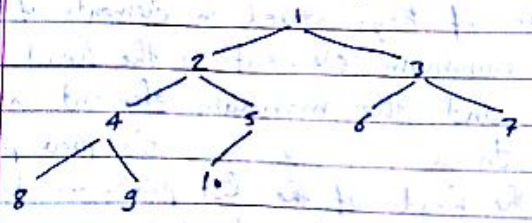
Binary Trees :-

Binary trees T and T' are said to be similiar if they have the same structure or, in other words, if they have the same shape. The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.

A node L is called a descendant of a node N (and N is called an ancestor of L) if there is a succession of children from N to L. In particular, L is called a left or right descendant of N according to whether L belongs to the left or right subtree of N.

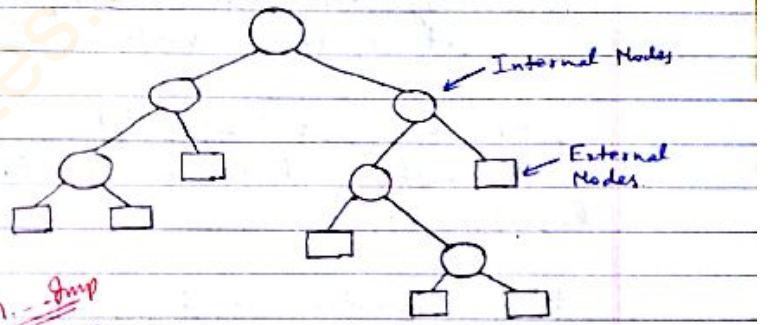
Complete Binary Trees :-

The tree T is said to be complete if all its levels, except possibly the last, have the maximum no of possible nodes, and if all the nodes at the last level appear as far left as possible.



Extended Binary Tree :- (2-Tree)

A binary tree T is said to be a 2-tree or an extended binary tree if each node N has either 0 or 2 children. In such a case, the nodes with 2 children are called internal nodes, and the nodes with 0 children are called external nodes.



M.m.n. - Imp

In an extended k-ary tree, the no. of leaves with n-internal nodes is :-

$$l = n(k-1)+1$$

Binary Search Tree :-

A BST is organized in a binary tree. In addition to a key field, each node contains fields left, right and p that points to the nodes corresponding to its left child, right child and its parent.

The keys in BST always satisfy the Binary-Search-Tree-Property :-

Let x be a node in BST. If y is a node in the left subtree of x, then key[y] ≤ key[x]. If y is a node in the right subtree of x then key[x] ≤ key[y].

The in-order walk of BST prints the keys in non-decreasing order.

It takes $\theta(n)$ time to walk an n-node BST, since after the initial call the procedure is called recursively exactly twice for each node in the tree - once for its left child and once for its right child.

Operations on BST :-

(1) Searching :-

Running Time = $O(h)$

where h → height of the tree.

(2) Minimum and Maximum :-

To find minimum, follow left child pointers and to find maximum follow right child pointers from the root until a NIL is encountered.

Running Time = $O(h)$.

MMM-Imp

Number of different binary search trees :-

Let b_n denote the no. of different binary trees with n nodes, then

$$b_n = \sum_{k=0}^{n-1} b_k \, b_{n-1-k}$$

where, $b_0 = 1$ & $b_1 = 1$

& $n \geq 2$

AVL Search Trees :-

An AVL tree is a binary search tree that is height balanced :- for each node x, the heights of the left and right subtrees of x differ by at most 1.

$$B.F. = h(T^L) - h(T^R)$$

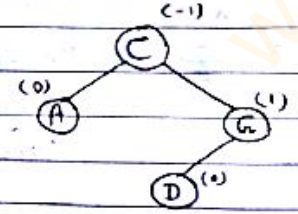
where, B.F = Balance Factor
T^L & T^R = left & right subtrees
$h(T)$ = height of tree T

for AVL tree :-

$$|h(T^L) - h(T^R)| \leq 1$$

Representation :-

AVL search trees like BST are represented using a linked representation. However, every node registers its balance factor.



Insertion in AVL Search Tree :-

Inserting an element into an AVL tree in its first phase is similar to that of the one used in a binary search tree. However, if after insertion of the element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, we resort to techniques called Rotations to restore the balance of the tree.

To perform rotations, it is necessary to identify a specific node A whose BF(A) is neither 0, 1 or -1 & which is the nearest ancestor to the inserted node on the path from the inserted node to the root.

Types of Rotations :-

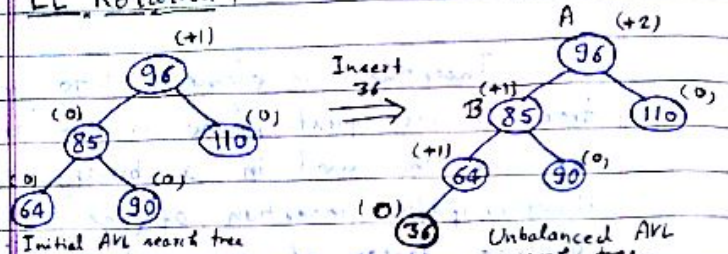
LL Rotation :- Inserted node is in the left subtree of left subtree of node A.

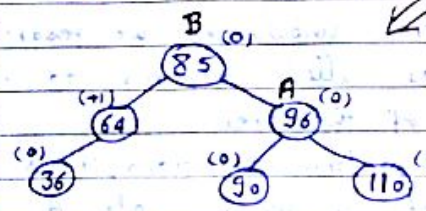
RR Rotation :- Inserted node is in the right subtree of right subtree of node B.

LR Rotation :- Inserted node is in the right subtree of left subtree of node A.

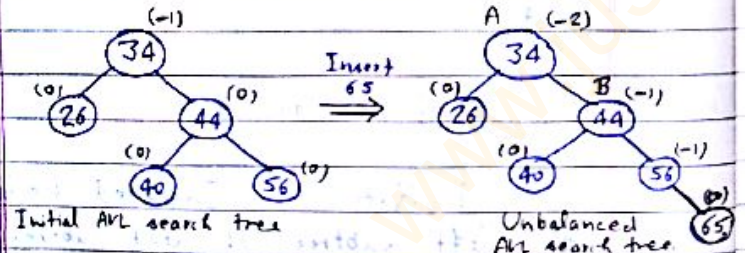
RL Rotation :- Inserted node in in the left subtree of right subtree of node A.

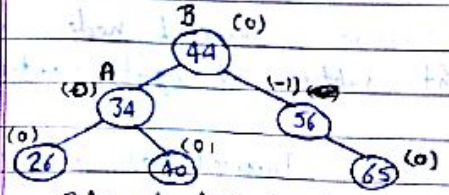
LL Rotation :-





RR-Rotation :-

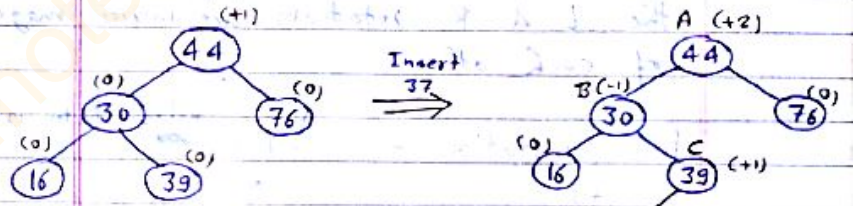


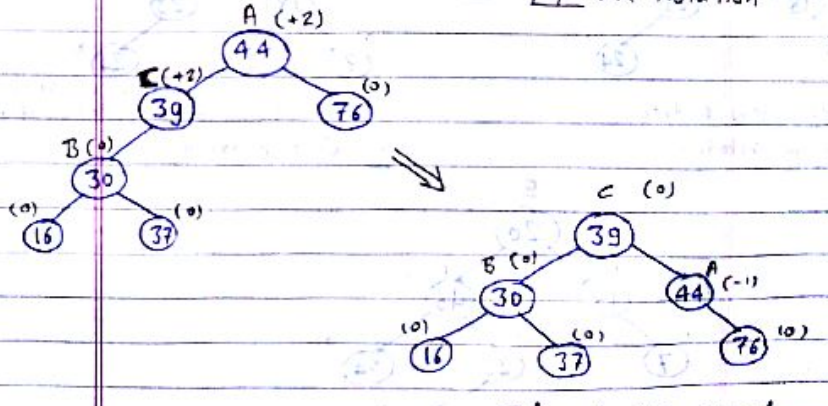


LR and RL Rotations :-

The balancing methodology of LR and RL rotations are similar in nature but are mirror images of one another.

LR can be accomplished by RR followed by LL rotation and RL can be accomplished by LL followed by RR rotation



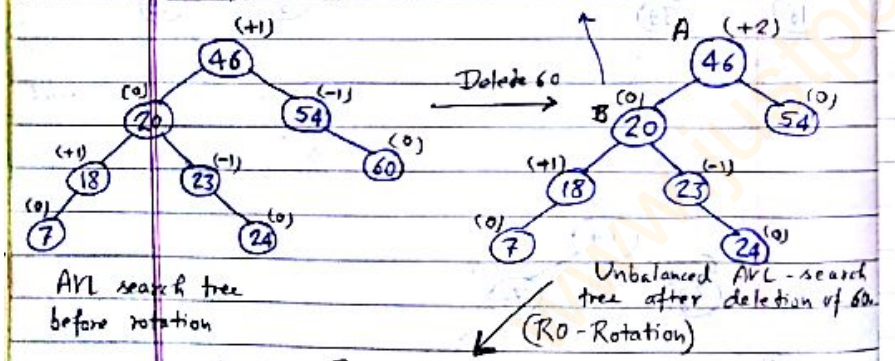


Deletion in AVL Search Tree :-

On deletion of a node X from the AVL tree, let A be the closest ancestor node on the path from X to the root node, with a balance factor of +2 or -2. If the deletion has occured in left subtree of A, then we apply L-rotations and if the deletion has occured in right subtree of A then we apply R-rotations. The L & R rotations are mirror images of each other.

R0 Rotation :-

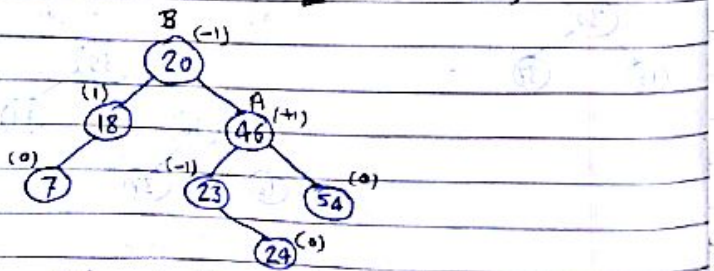
B's balance factor is 0



AVL search tree before rotation

Unbalanced AVL - search tree after deletion of 60.

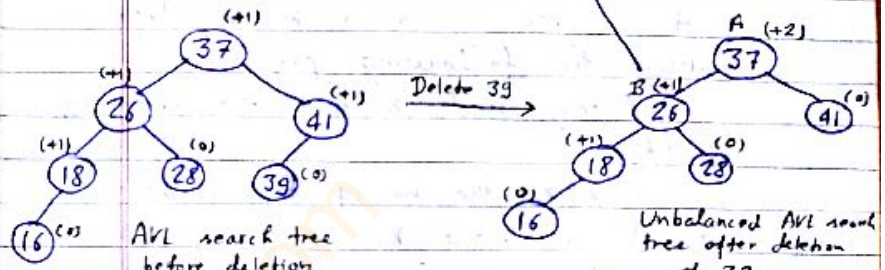
(R0 - Rotation)



Balanced AVL search tree after R0 Rotation

R1 Rotation :-

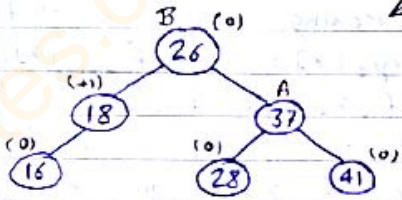
B's balance factor is +1



AVL search tree before deletion

Unbalanced AVL search tree after deletion of 39

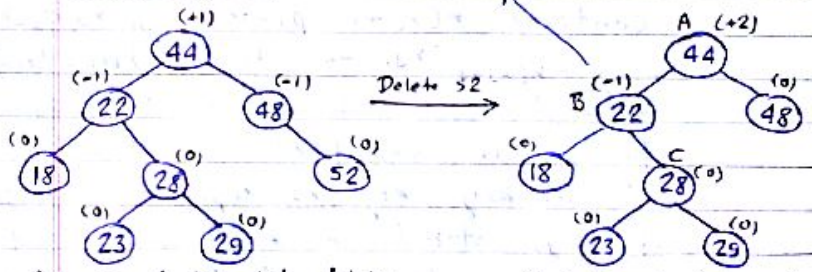
R1 Rotation



Balanced AVL search tree after R1 Rotation.

R -1 Rotation :-

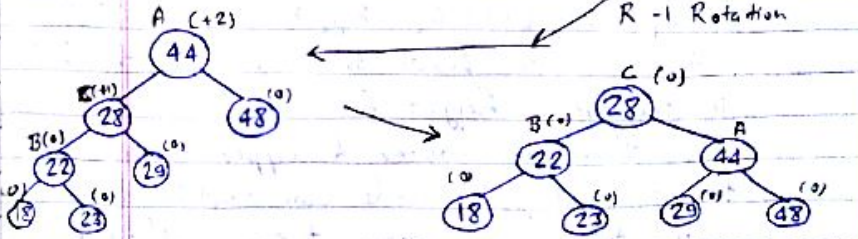
B's BF is -1



AVL - search tree before deletion

Unbalanced AVL search tree after deletion

R -1 Rotation



Balanced AVL Search tree after R -1 Rotation.

B-Trees :→

A B-tree T is a rooted tree having the following properties :—

(1) Every node x has the following fields :—

(a) n[x], the no. of keys currently stored in node x.

(b) the n[x] keys themselves, stored in non-decreasing order, so that $key_1[x] \le key_2[x] \le \cdots \le key_{n[x]}[x]$.

(c) leaf[x], a boolean value that is true if x is a leaf and false if x is an internal node.

(2) Each internal node x also contains n[x]+1 pointers $c_1[x], c_2[x], \ldots, c_{n[x]+1}[x]$, to its children. Leaf nodes have no children, so their c_i fields are undefined.

(3) The keys $key_i[x]$ separate the ranges of keys stored in each subtree: if k_i is any key stored in the subtree with root $c_i[x]$, then :—

$$k_1 \le key_1[x] \le k_2 \le key_2[x] \le \cdots \le key_{n[x]}[x] \le k_{n[x]+1}$$

(4) All leaves have the same depth, which is the tree's height h.

(5) There are lower & upper bounds on the no. of keys a node can contain. It is represented by fix integer $t \ge 2$ called the minimum degree of the B-tree :—

(a) Every node other than the root must have at least t-1 keys. Every internal node other than the root thus has at least t children. If the tree is non-empty, the root must have at least one key.

(b) Every node can contain at most 2t-1 keys. Therefore, an internal node can have at most 2t children. A node is full if it contains (2t-1) keys.

Height of a B-tree :—

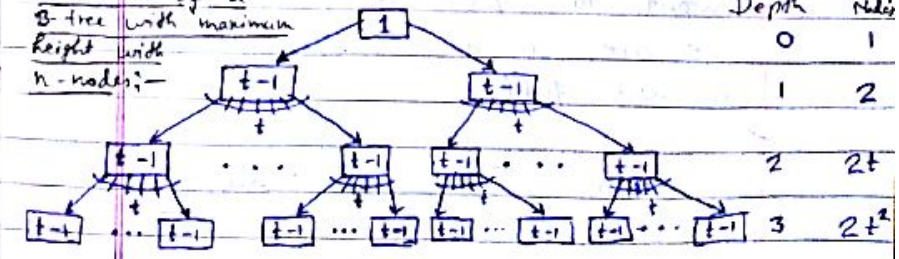
If $n \ge 1$, then for any n-key B-tree T of height h and minimum degree $t \ge 2$:—

$$h \le \log_t\left(\frac{n+1}{2}\right)$$

or

$$n \ge 2t^h - 1$$

Structure of a B-tree with maximum height with n-nodes :—



No. of keys in a node = t-1

Searching a B-Tree :-

Running time :-

The no of disk pages accessed by B-Tree-Search $= O(\log_t n)$

and the no. of keys in a page $< 2t$
$= O(t)$

So, total search time $= O(t \cdot \log_t n)$

Inserting a key into a B-Tree :-

To insert a key into a B-Tree, we travel down the tree searching for the position where the new key belongs, we split each full node, we come to along the way (including the leaf itself). Thus whenever we want to split a full node y, we are assured that its parent is not full.

Running Time $= O(t \log_t n)$

Deleting a key from a B-tree :-

Here, we must insure that a node doesn't get too small during deletion (except root).

Procedure :-

(1) If the key k is in node x and x is a leaf, delete the key k from x.

(2) If the key k is in node x and x is an internal node, do the following :-

(a) If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k' and replace k by k' in x.

(b) Symmetrically, if the child z that follows k in node x has at least t keys, then find the successor k' of k in the subtree rooted at z. Recursively delete k' and replace k by k' in x.

(c) Otherwise if both y and z have only $t-1$ keys, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains $2t-1$ keys. Then free z and recursively delete k from y.

(3) If the key k is not present in internal node x, determine the root $c_i[x]$ of the appropriate subtree that must contain k. If $c_i[x]$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

(a) If $c_i[x]$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $c_i[x]$ an extra key by moving a key from x down into $c_i[x]$, moving a key from $c_i[x]$'s immediate left or right up into x, and moving the appropriate child pointer from the sibling into $c_i[x]$.

(b) If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys, merge $c_i[x]$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Running time = $O(t \log_t n)$

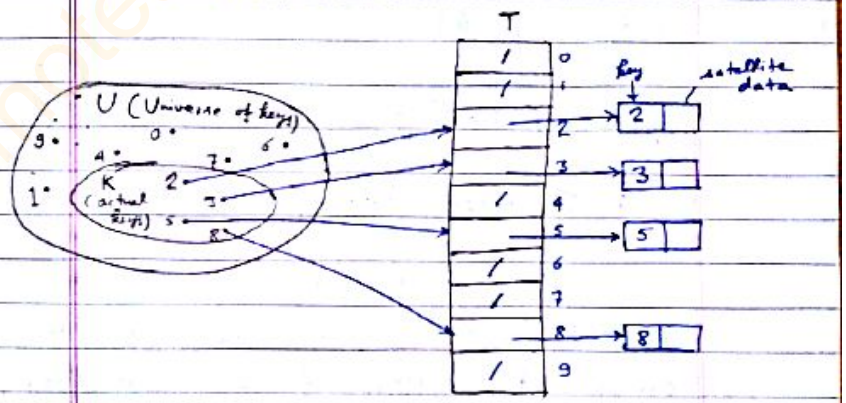
Hash Tables

Direct Address tables:-

It works well when the universe U of keys is reasonably small.

Let universe $U = \{0, 1, \_\_, m-1\}$, where m is not too large.

To represent the dynamic set, we use an array, or Direct-address table, denoted by $T[0, 1 \_\_ m-1]$, in which each position, or slot, corresponds to a key in the universe U.



Hash Tables :-

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$, i.e. we use a hash function h to compute the slot from the key k.

Here :-

$h : U \longrightarrow \{0, 1, \_\_, m-1\}$

Hash Functions :-

A good hash function satisfies the assumption of simple uniform hashing : each key is equally likely to hash to any of the m slots, independently to where any other key has hashed to.

A hash function h must be deterministic in that a given input k should always produce the same output h(k).

Types of hashing schemes :-

(a) Division Method :-

$$h(k) = k \pmod m$$

A prime not too close to an exact power of 2 is often a good choice for m.

(b) Multiplication Method :-

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where :- $0 < A < 1$,

$kA \bmod 1$ is fractional part of kA

(c) Universal hashing :-

Here we choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach is called Universal hashing.

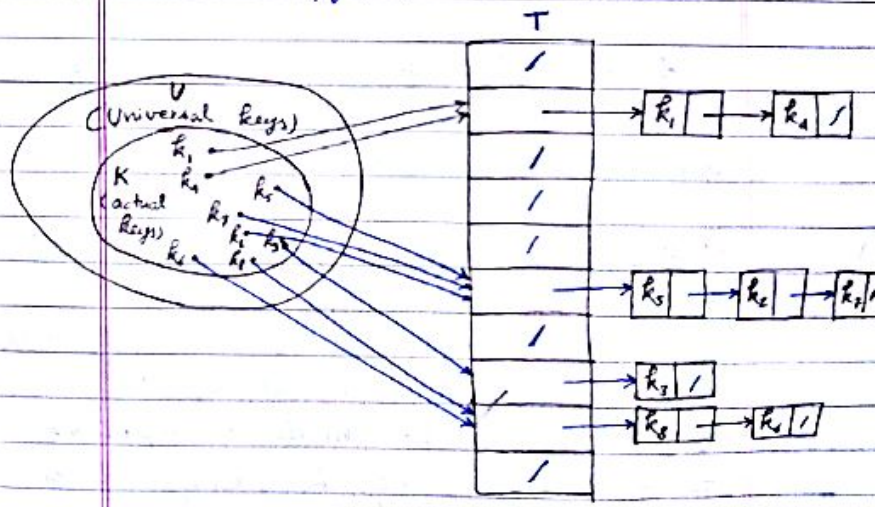
Collision Resolution :-

When two keys hash to the same slot, this situation is called collision.

It is also called as open hashing

(1) Collision Resolution by Chaining :-

In chaining, we put all the elements that hash to the same slot in a linked list as shown in figure :-



It is also called as closed hashing

(2) Open Addressing :-

In open addressing, all elements are stored in the hash table itself.

To perform insertion using open addressing, we successively examine, or probe, the hash table until we find an empty slot to put the key.

To determine which slots to probe, we extend the hash function to include the **probe number** (starting from 0) as a second input. Thus, the hash function becomes :-

$$h : U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$$

The algo for searching for key k probes the same sequence of slots, used during insertion.

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we can't simply mark that slot as empty by storing NIL in it. Doing so might make it impossible to retrieve any key k during whose insertion, we had probed slot i and found it occupied.

To solve this problem, we mark the ~~solution~~ slot by storing in it the special value DELETED instead of NIL.

Types of Probing :→

(a) Linear probing :-

Given an ordinary hash function $h' : U \rightarrow \{0, 1, \ldots, m-1\}$, which we refer to as an auxiliary hash function, ~~the linear of does~~ the method of linear probing uses the hash function :-

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$.

↳ Probe no.

(b) Quadratic Probing :-

It uses a hash function of the form :-

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where, h' is an auxiliary function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m-1$.

(c) Double hashing :-

It uses the hash function of the form :-

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

where h_1 and h_2 are auxiliary functions.

Perfect hashing :-

We call a hashing technique perfect hashing if the worst case no. of memory accesses to perform a search is $O(1)$.

Instead of making a list of the keys hashing to slot j, however we use a small secondary ~~table~~ hash table S_j with an associated hash function h_j.

Matrix Chain Multiplication :-

$M_1 = (10 \times 100)$, $M_2 = (100 \times 20)$,
$M_3 = (20 \times 5)$, $M_4 = (5 \times 80)$

Solution :-

M_1	M_2	M_3	M_4
(10×100)	(100×20)	(20×5)	(5×80)
$M_{1,2}$ (10×20) $(20,000)$	$M_{2,3}$ (100×5) $(10,000)$	$M_{3,4}$ (20×80) $(8,000)$	
$M_{1,3}$ (10×5) $(15,000)$	$M_{2,4}$ (100×80) $(50,000)$		
$M_{1,4}$ (10×80) $(19,000)$			

No. of multiplications required = 19,000

Inversions :-

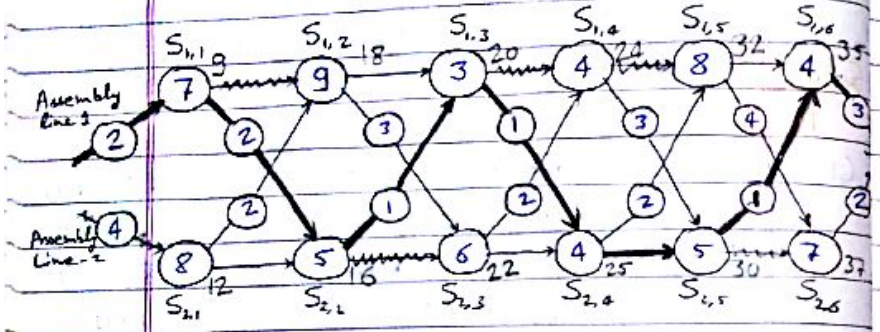
Let $A[1 \ldots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion of A.

Maximum no of inversions in an array with n distinct elements $= {}^nC_2 = \frac{n(n-1)}{2}$
(when its sorted in decreasing order).

As the no. of inversions increase, the running time of insertion sort also increases.

Algorithm to determine the number of inversions in any permutation on n-elements takes $\theta(n \lg n)$ in worst case (Using modified merge sort).

Assembly-Line Scheduling :-



j	1	2	3	4	5	6
$f_1[j]$	9	18	20	24	32	35
$f_2[j]$	12	16	22	25	30	37

$f^* = 38$

j	2	3	4	5	6
$l_1[j]$	1	(2)	1	1	(2)
$l_2[j]$	(1)	2	(1)	(2)	2

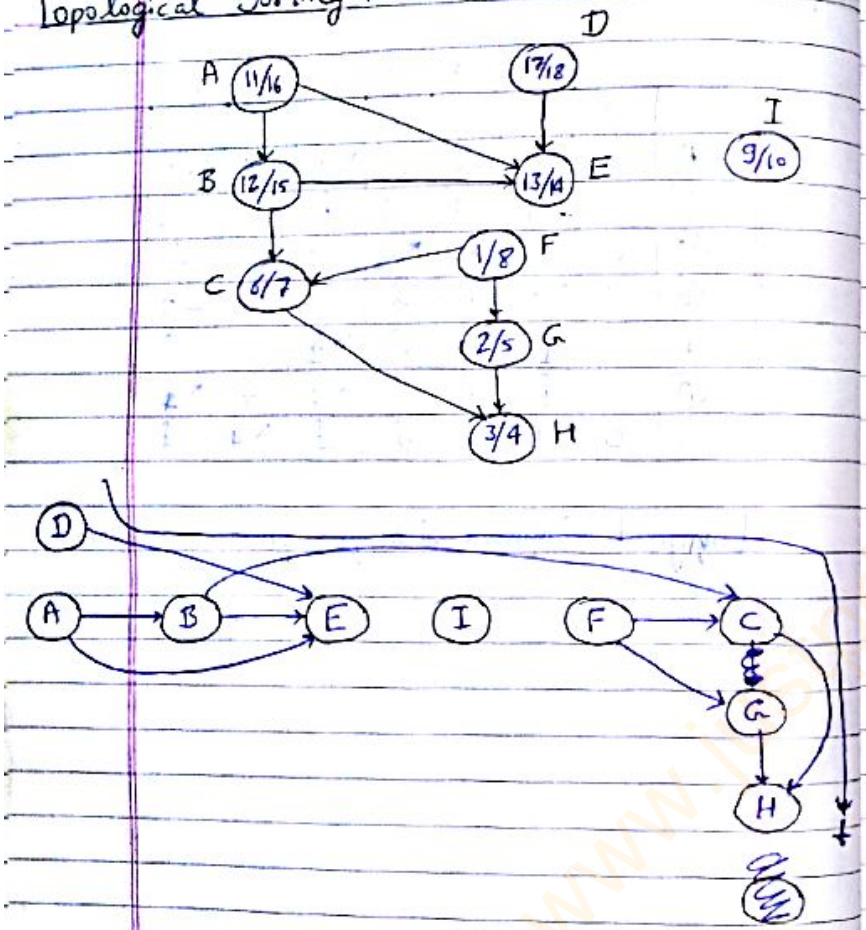
$l^* = 1$

Longest Common Subsequence :-

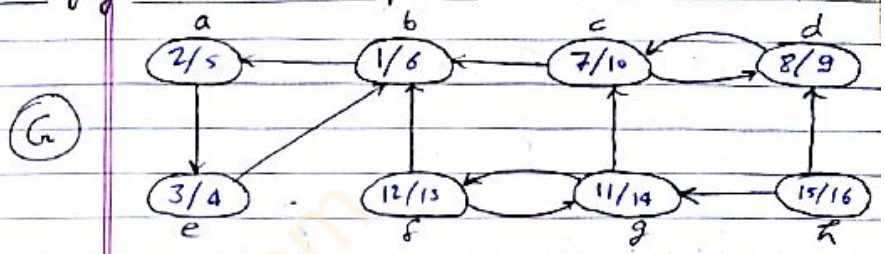
		(B)	D	(C)	A	(B)	(A)
	0	0	0	0	0	0	0
A	(0)	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
(B)	0	(↖ 1)	(← 1)	← 1	↑ 1	↖ 2	← 2
(C)	0	↑ 1	↑ 1	(↖ 2)	(← 2)	↑ 2	↑ 2
(B)	0	↖ 1	↑ 1	↑ 2	↑ 2	(↖ 3)	← 3
D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
(A)	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	(↖ 4)
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	(↑ 4)

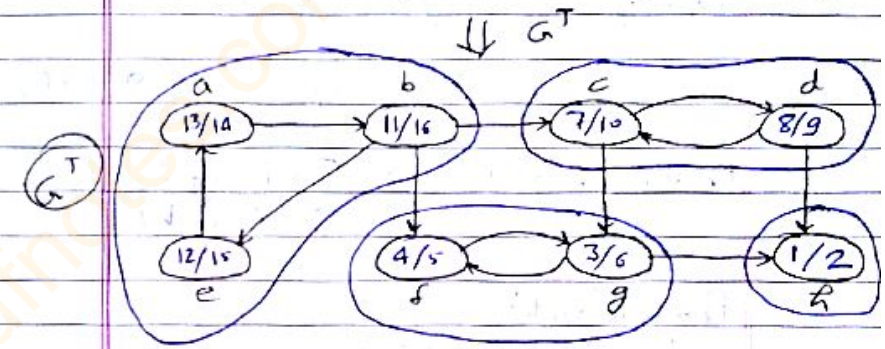
Longest Common Subsequence = BCBA

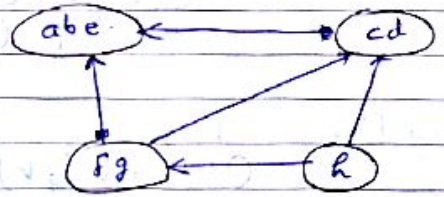
Topological Sorting :-



Strongly Connected Components :-







Complexities of Graph Algorithms :→

(1) BFS ⇒ $O(V+E)$

(2) DFS ⇒ $\Theta(V+E)$

(3) Topological Sorting ⇒ $\Theta(V+E)$

(4) Strongly Connected Components ⇒ $\Theta(V+E)$

(5) Kruskal's Algorithm ⇒ $O(E \lg E)$
$= O(E \lg V)$

(6) Prim's Algorithm:-

(i) Using binary min-heap:-
$O(V \lg V + E \lg V)$
$= O(E \lg V)$

(ii) Using Fibonacci heap:-
$O(E + V \lg V)$

(7) Bellman Ford Algorithm:-
$O(VE)$

(8) Single-Source Shortest path in Directed Acyclic Graph:-
$\Theta(V+E)$

(9) Dijkstra's Algorithm:-

(i) Using binary min-heap:-
$O((V+E) \lg V)$
$= O(E \lg V)$

(ii) Using fibonacci min-heap:-
$O(E + V \lg V)$