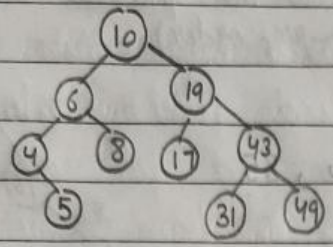
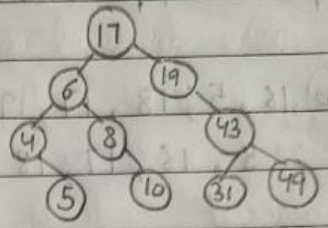
delete (11) inorder : 4, 5, 6, 8, 10, (11), 17, 19, 31, 43, 49



or

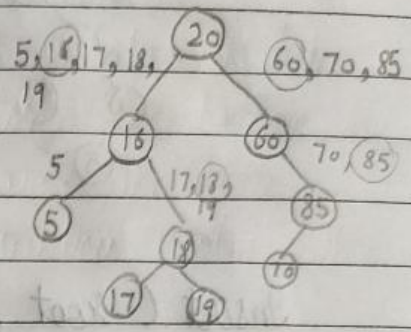


O(log n) best/avg
O(n) worst

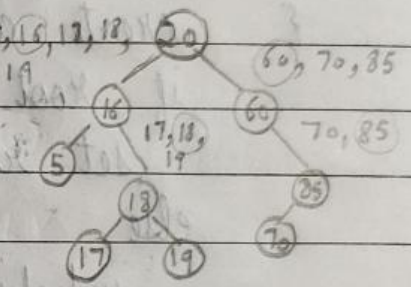
```
Insert ( root , value)
{   create a newNode
    if root is NULL then,
      set root = newNode
    else,
      initialise a temp node as root node
      while temp is not null do,
          set prev = temp
          if temp → data = value
          return root
          else if temp → data < value
          set temp = temp → right
          else
            set temp = temp → left
      // end of while loop
      if prev → data > value
       set prev → left = newNode
      else
       set prev → right = newNode
    //end
    return root
}
```

Construction of BST
(from preorder / postorder) Bcz we know the inorder of BST (in asc. order)

Pre : 20, 16, 5, 18, 17, 19, 60, 85, 70
inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85



Post : 5, 17, 19, 18, 16, 70, 85, 60, 20
inorder : 5, 16, 17, 18, 19, 20, 60, 70, 85



AVL Trees - BST

- ~~diff b/w~~ Ht of left subtree - Ht of rt subtree = {-1, 0, 1}

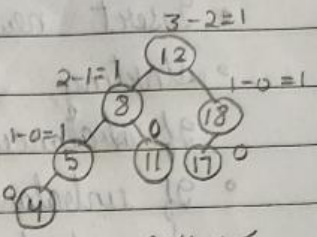
this diff - balance factor

ALGORITHMS

AVL Tree

(Named after Adelson, Velski & Landis)

(height balancing BST / self balancing BST)

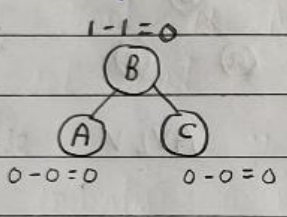


conditions :

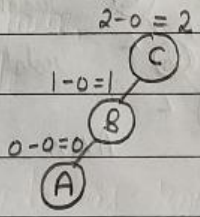
- It must be BST.
- Balance factor = {-1, 0, 1}

(Balance factor is the difference between height of left subtree and right subtree of a node)

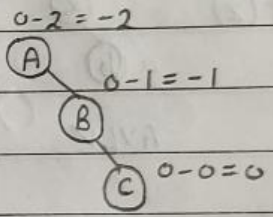
eg :



BST ✓ AVL tree ✓



BST ✓ AVL tree ✗



BST ✓ AVL tree ✗

Advantage : reduces the skewness

BST operations may take O(n) for skewed binary trees.

But AVL tree is balanced ∴ takes O(log n)

n = nodes in the tree

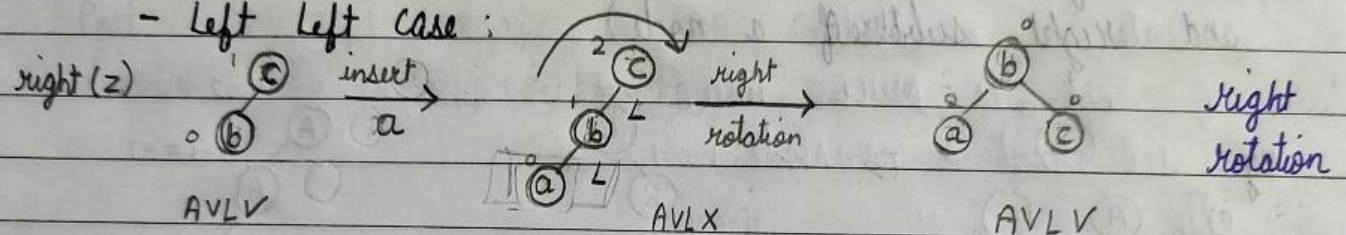
Imp pts : • After insertion, only those nodes are affected that lie on the path from w to root
• After insertion if tree becomes imbalanced, only one node is responsible & it should be balanced to re-balance the tree.

Dt. Page

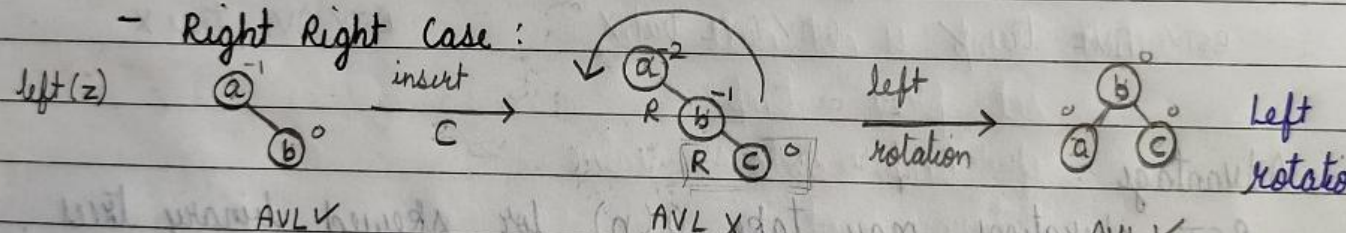
Insertion :

- Insert new node (w) following standard BST operation.
- Starting from w, travel up and find first unbalanced node. If unbalanced node is not present, no need for re-balancing.
- If unbalanced node is found (let z), re-balance the tree by performing appropriate rotations on the subtree rooted with z. four rotations can be possible :

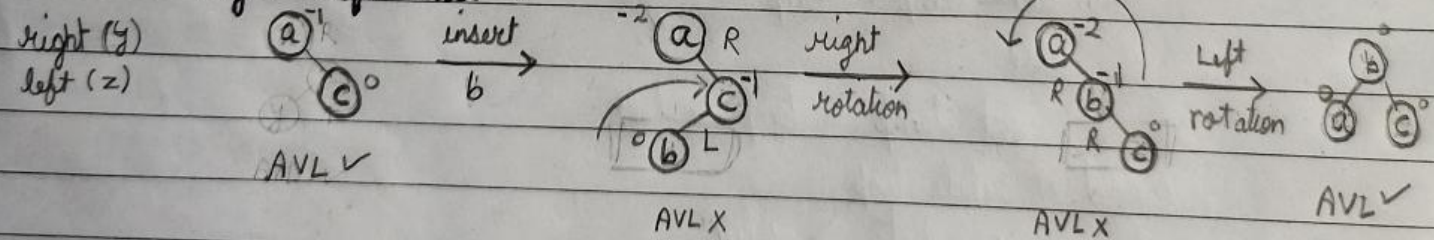
- Left Left case :



- Right Right Case :

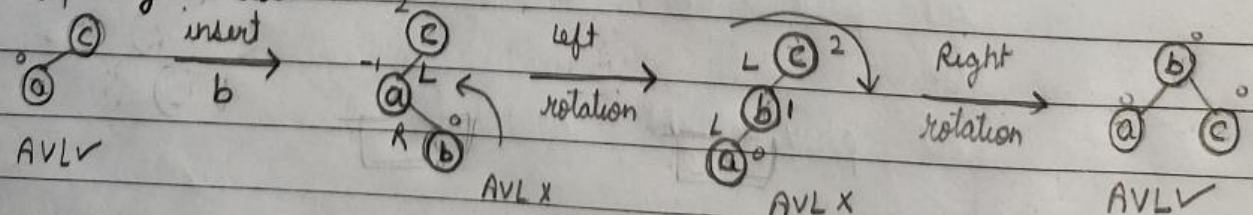


- Right left case :



Left Right case :

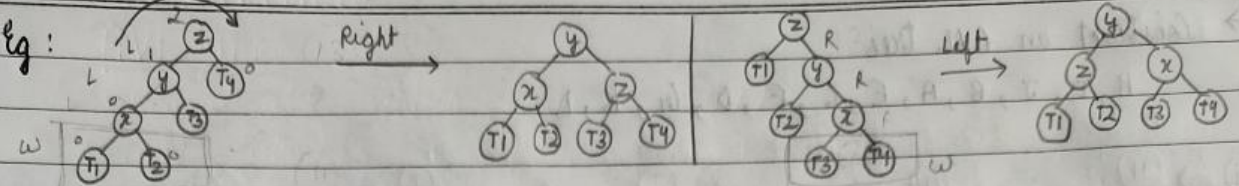
left (y) –
right (z)

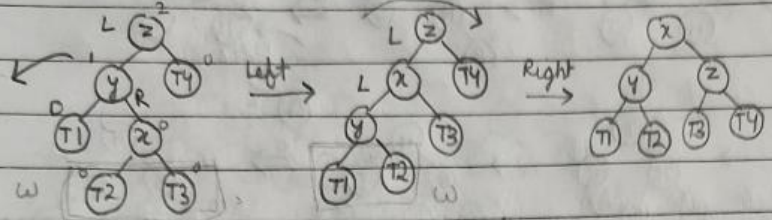


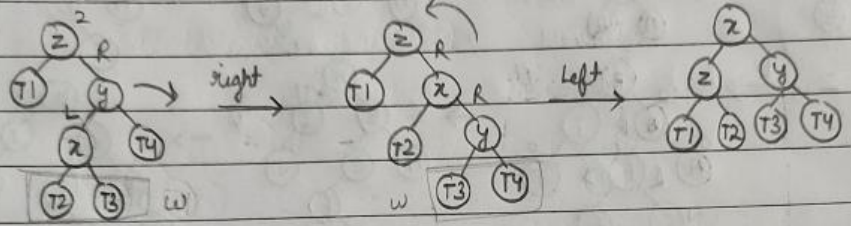
y = child of z x = grandchild of z
(in the path from w to z)

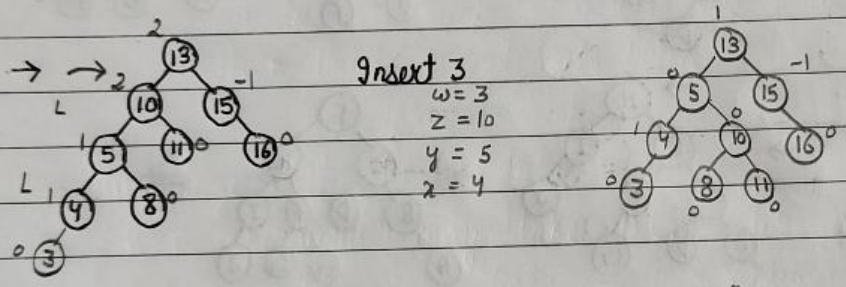
3 child nodes ✗ so keep middle one as it is

Eg :



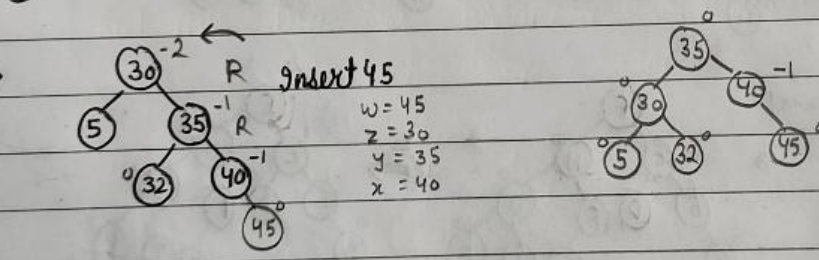






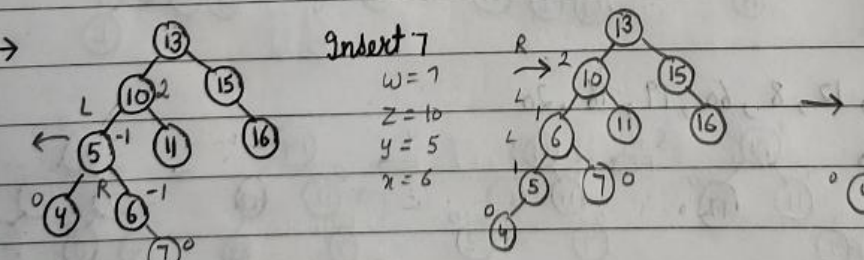
Insert 3

$w = 3$
$z = 10$
$y = 5$
$x = 4$



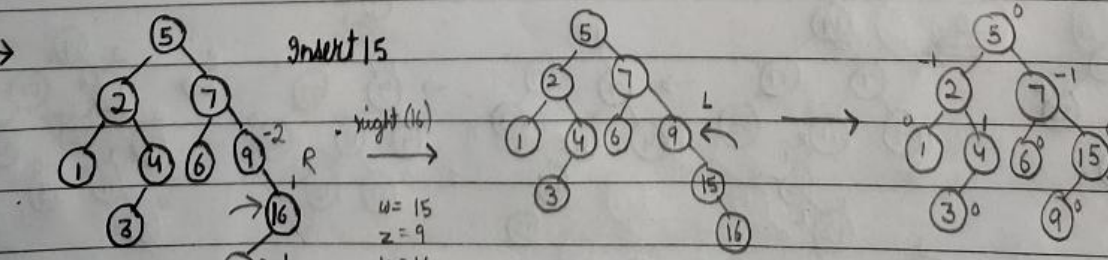
Insert 45

$w = 45$
$z = 30$
$y = 35$
$x = 40$

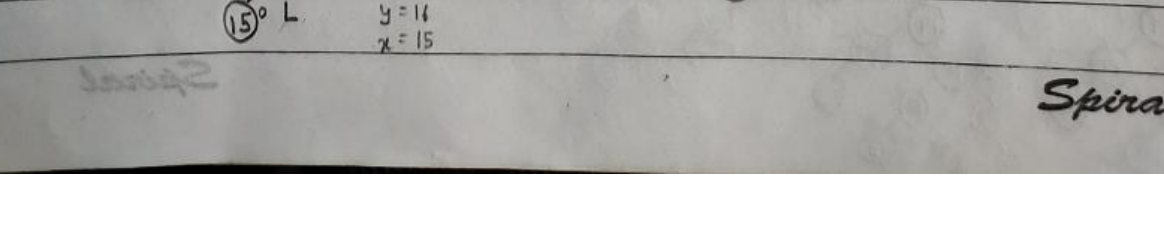


Insert 7

$w = 7$
$z = 10$
$y = 5$
$x = 6$



Insert 15



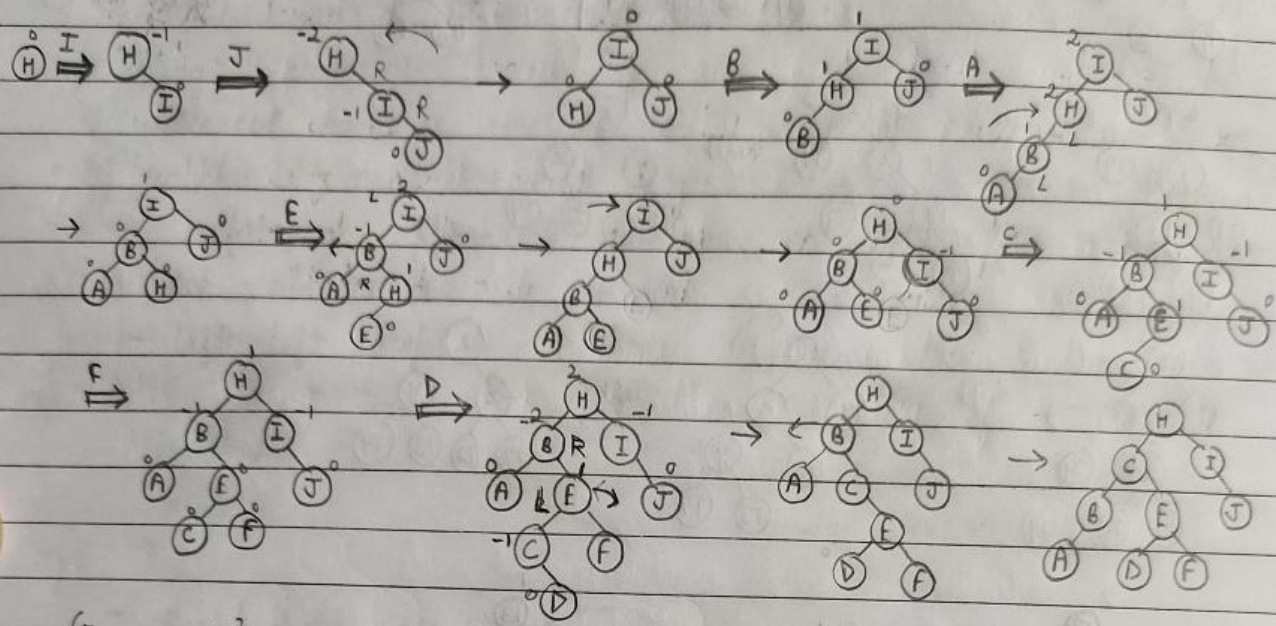
$w = 15$
$z = 9$
$y = 16$
$x = 15$

Spiral

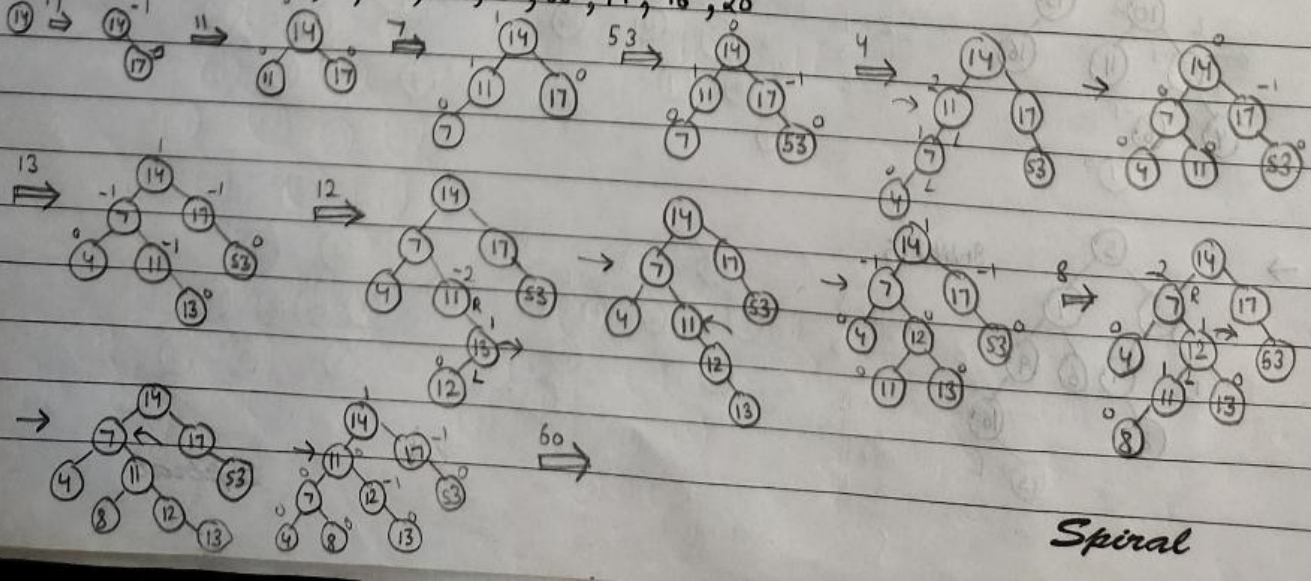
! Adjust child of shifted root properly !

→ Construct an AVL tree

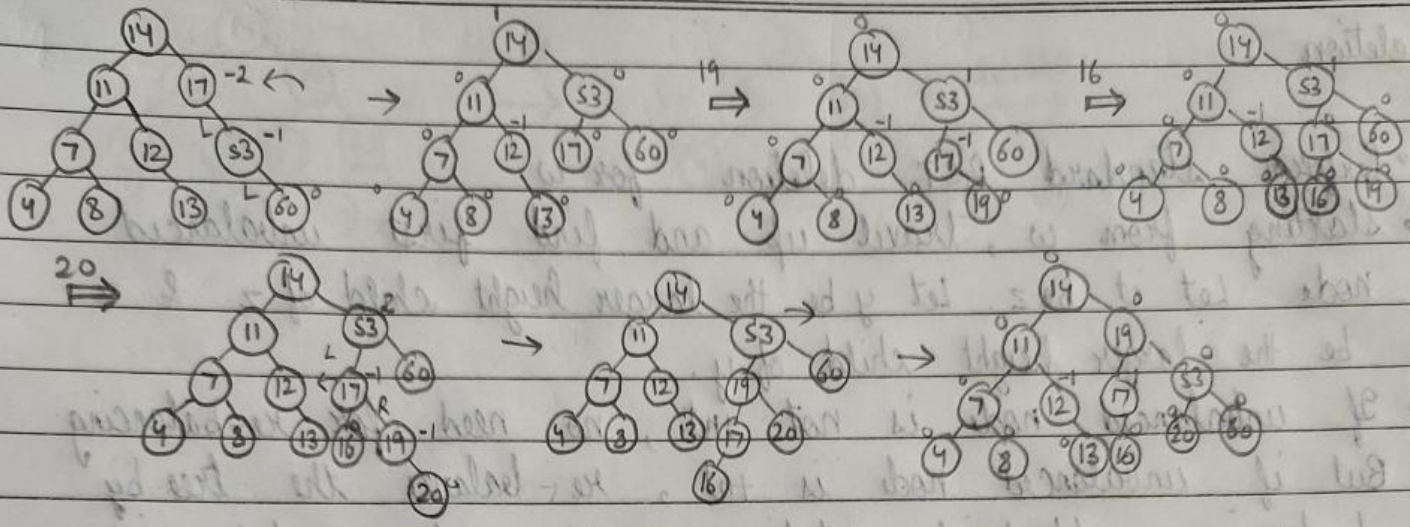
H, I, J, B, A, E, C, F, D, G, K, L



→ 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20



Spiral

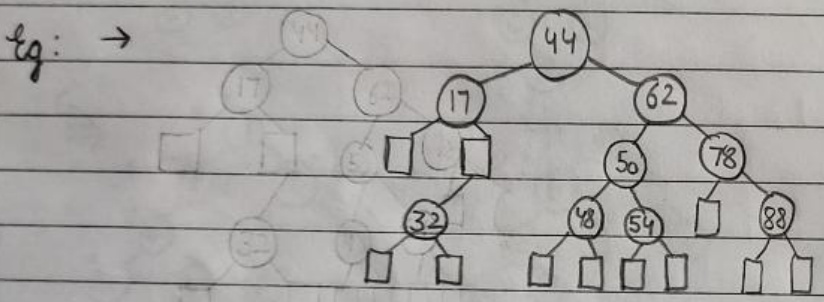

19
16
20

Imp pts: • nodes tht b/w w & root are affected
• may need to balance more than 1 node

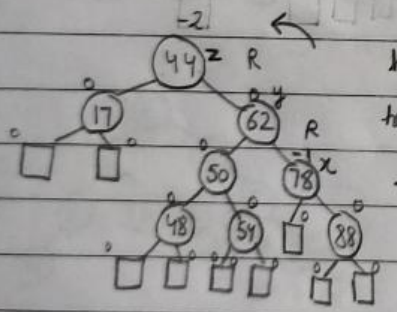
Deletion :

- Perform standard BST deletion for w
- Starting from w, travel up and find first unbalanced node. Let it be z. Let y be the larger height child of z & x be the larger height child of y.
- If unbalanced node is not tht, no need for re-balancing
- But if unbalanced node is tht, re-balance the tree by performing appropriate rotations on the subtree rooted with z.
- After fixing z, check for the ancestors of z as well, till root.

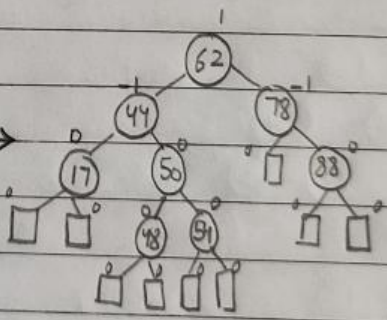
Eg: →



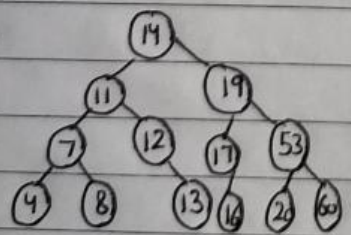
delete 32



height of 17 = 1
height of 62 = 3
y = 62
height of 50 = 2
height of 78 = 2
x can be either
x = 78

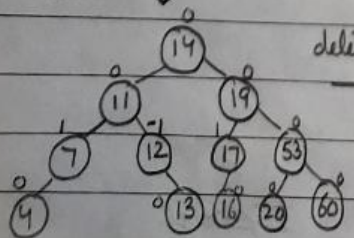


→

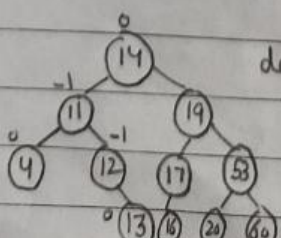


delete 8, 7, 11, 14, 17

delete 8 ↓



delete 7 →

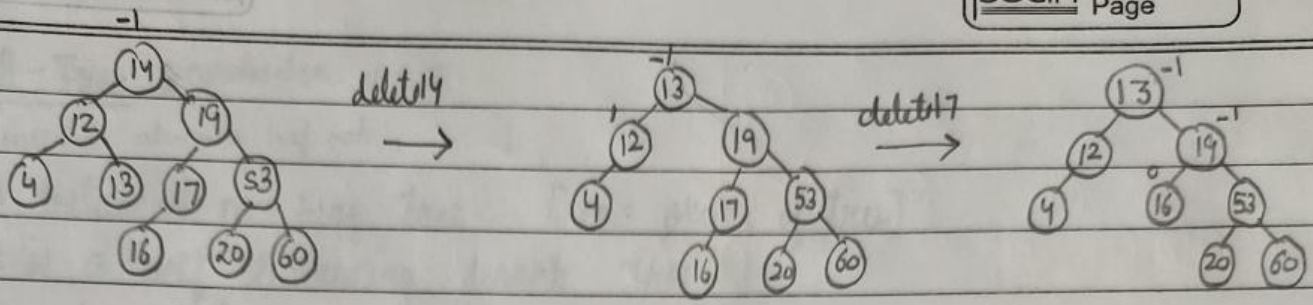


no need to check ht of right subtree

delete 11 →

If we replace 11 by 12, no need to re-balance

Spiral


-1
14
12
19
4
13
17
53
16
20
60
delete 14
-1
13
12
19
4
17
53
16
20
60
delete 17
-1
13
12
-1
14
4
16
53
20
60

(2) node
key

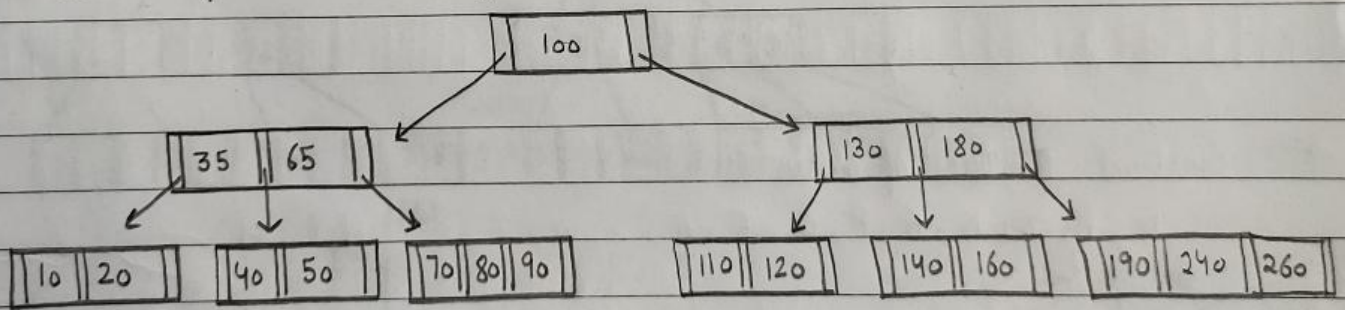
Dt.
Page

B-Tree generalisation of BST

insertion always in leaf node

- A balanced m-way tree [m = order of tree]
- It is a self balancing search Tree
- A node can have more than 2 children
- A node can have more than one key.
- Maintains data in sorted order (asc)
- All leaf nodes must be at same level.
- B tree of order m has following properties:
  - → Every node has max = m children
  - → Min children: leaf = 0
    root = 2
    internal nodes = $\lceil \frac{m}{2} \rceil$
  - → Every node has max = (m-1) keys
  - → Min keys: root = 1
    all other = $\lceil \frac{m}{2} \rceil - 1$

values in different nodes are arranged as in BST and in each node in asc order



Main application: in databases - indexing

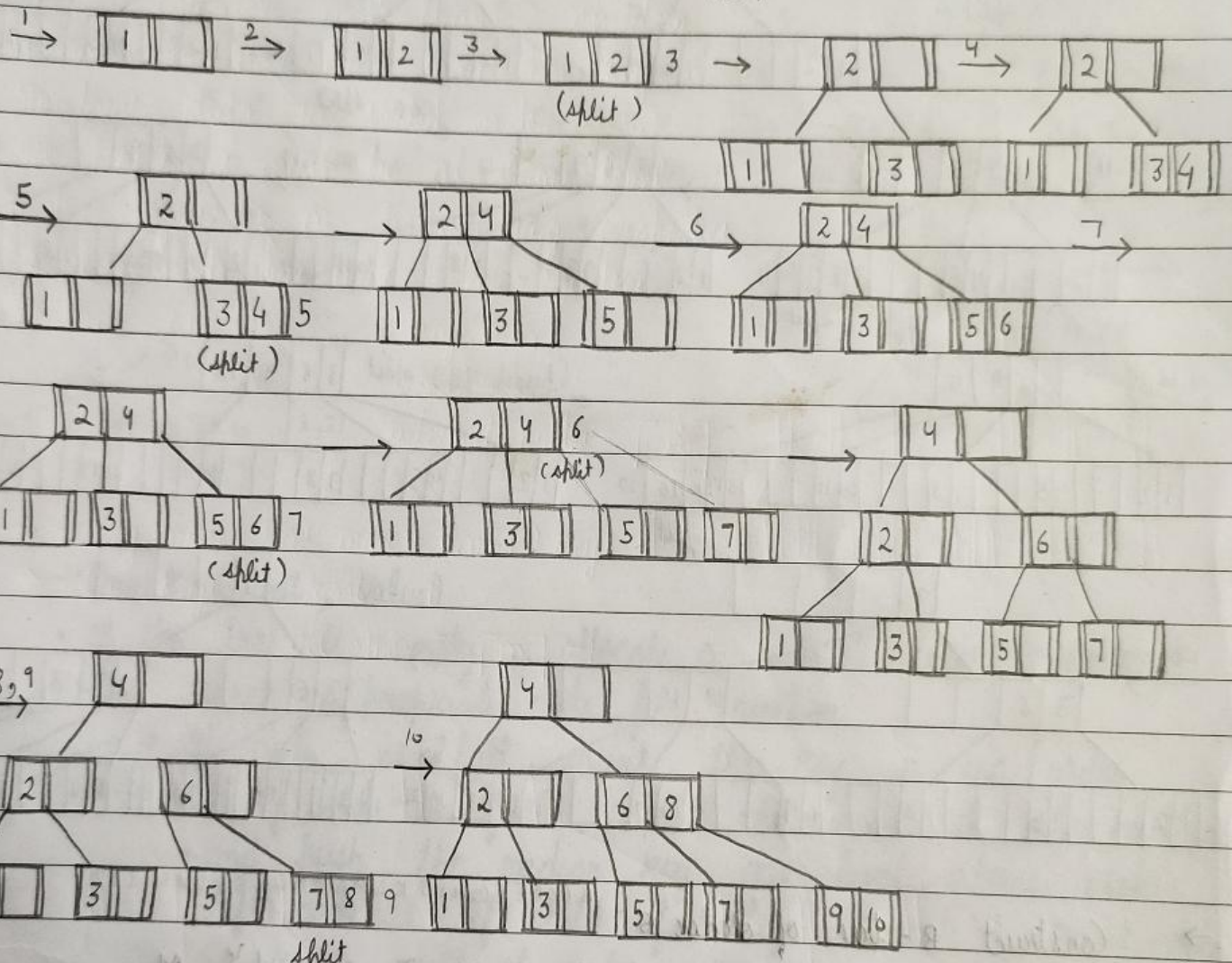
- → used to organise data in secondary memory (in blocks)
- → disk access time is very high compared to main memory
  B-Trees reduce the time → O(h)
  [here h is height of B-Tree which is kept min by putting max possible keys in the node
  ∴ it is a fat tree
  generally node size ≈ block size]

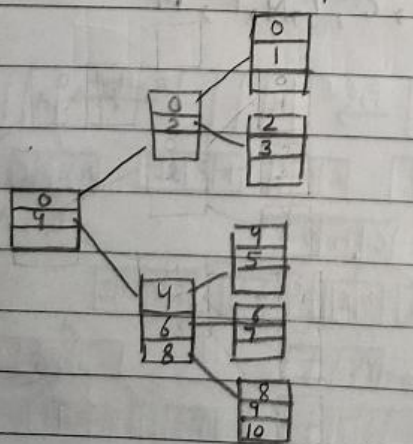
Spiral

Insertion :

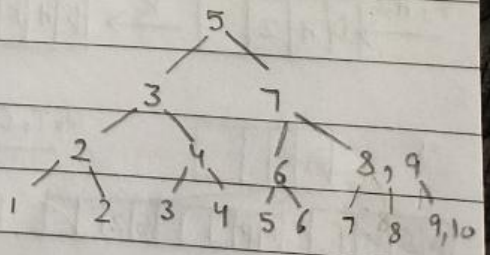
→ m = 3 insert values from 1 to 10

max keys = 3-1 = 2 min $\lceil \frac{3}{2} \rceil - 1 = 1$



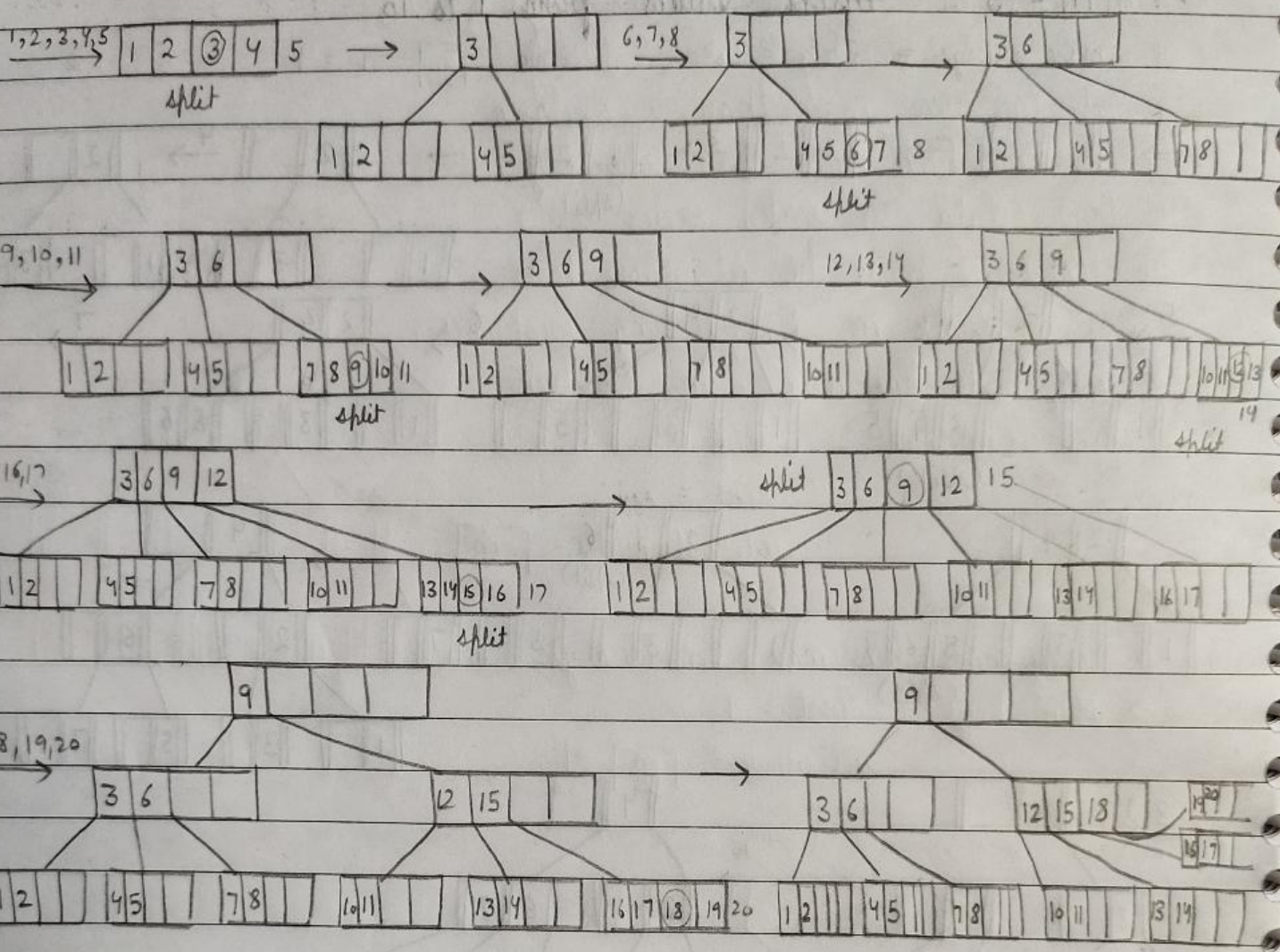


In B+ tree



→ $m = 5$ insert values 1 to 20

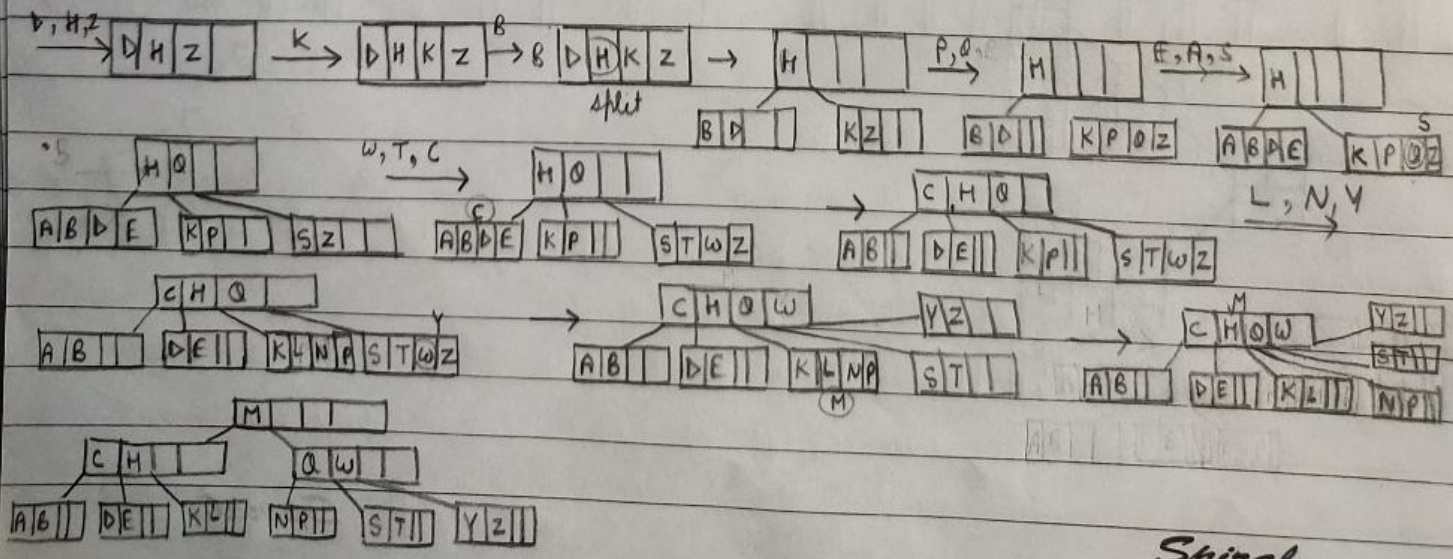
max keys = 4 min keys $\lceil \frac{5}{2} \rceil - 1 = 2$



→ Construct B-tree of order 5:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

D, H, Z, K, B, P, Q, E, A, S, W, T, C, L, N, Y, M

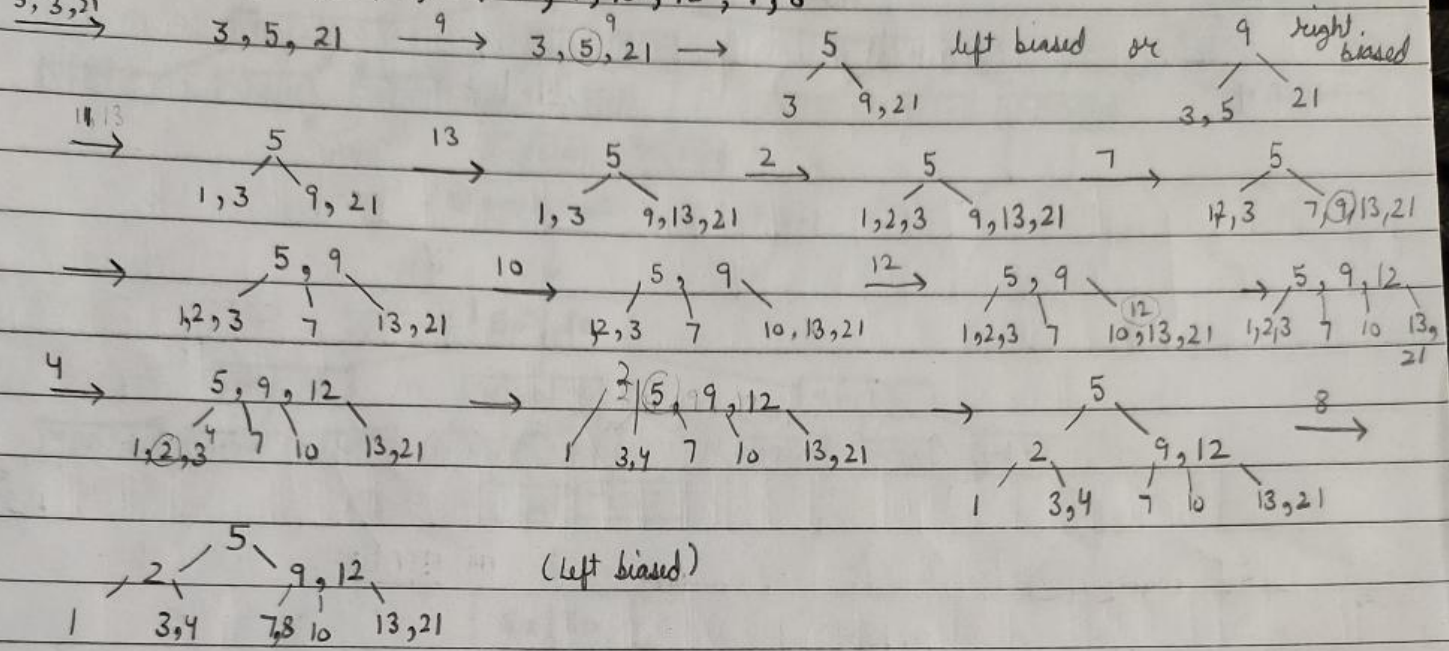


Spiral

for even order – left & Right biased trees are possible.

→ m = 4 max keys = 3

5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8



→ Insert (root , Value)

- If the tree is empty, allocate a node & insert the key.
- Else, search appropriate node for insertion.
  - → If node is not full, insert the key in inc. order.
  - → else if node is full, split the node at the median and push the median key one level above. Make left node as left child of median element & right node as right child of median element.

If target node is leaf node

① contains more than min keys

② contains min no of keys

② borrow - left ③ borrow - right ④ merge ← L / R

If target node is internal node

⑤ inorder Pred. ⑥ inorder succ. ⑦ Merge

Deletion

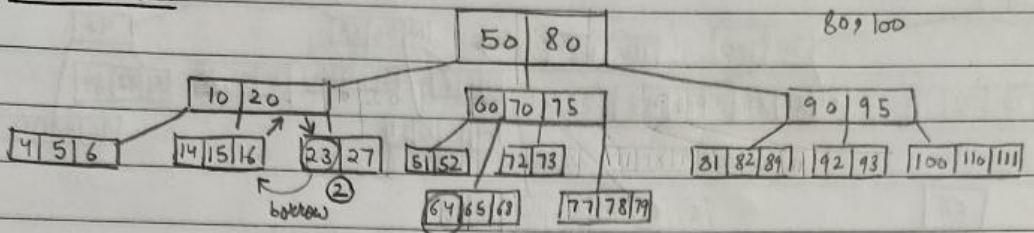
64, 23, 72, 65, 20, 70, 95, 77

80, 100

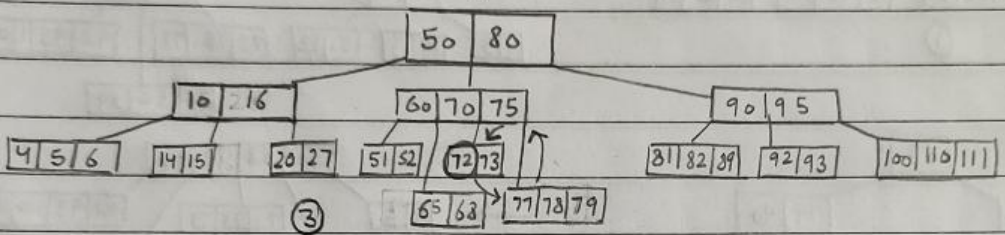
min keys = $\lceil \frac{5}{2} \rceil - 1$

= 2

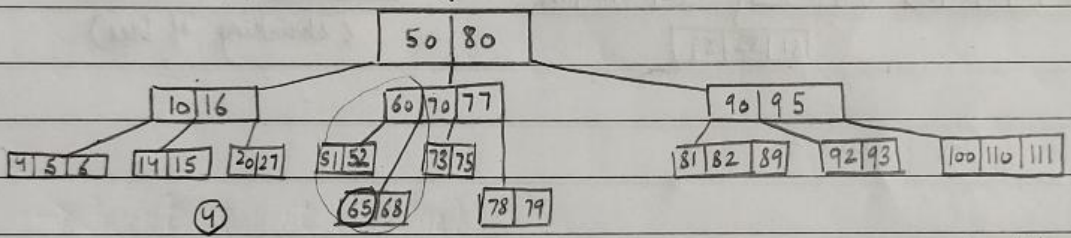
max keys = 4



① simply delete



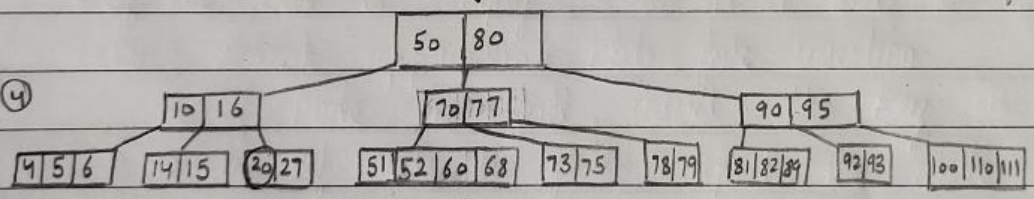
③



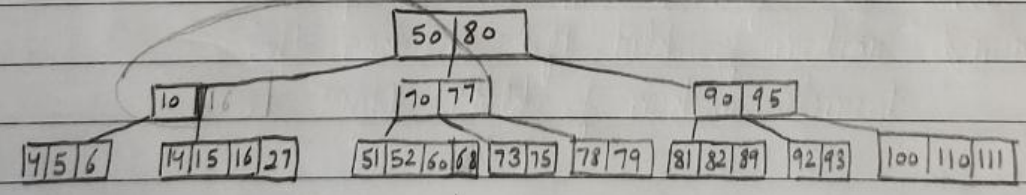
④

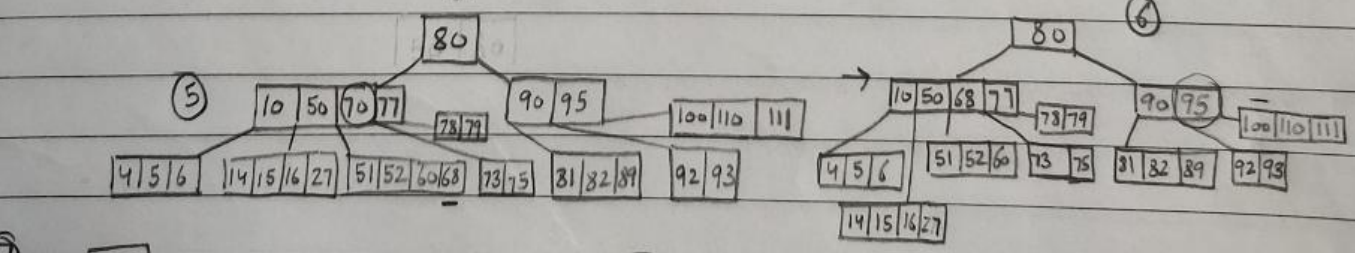
if left merge

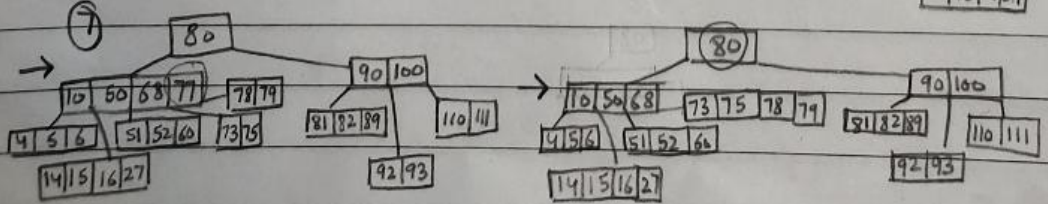
68, 70, 73, 75



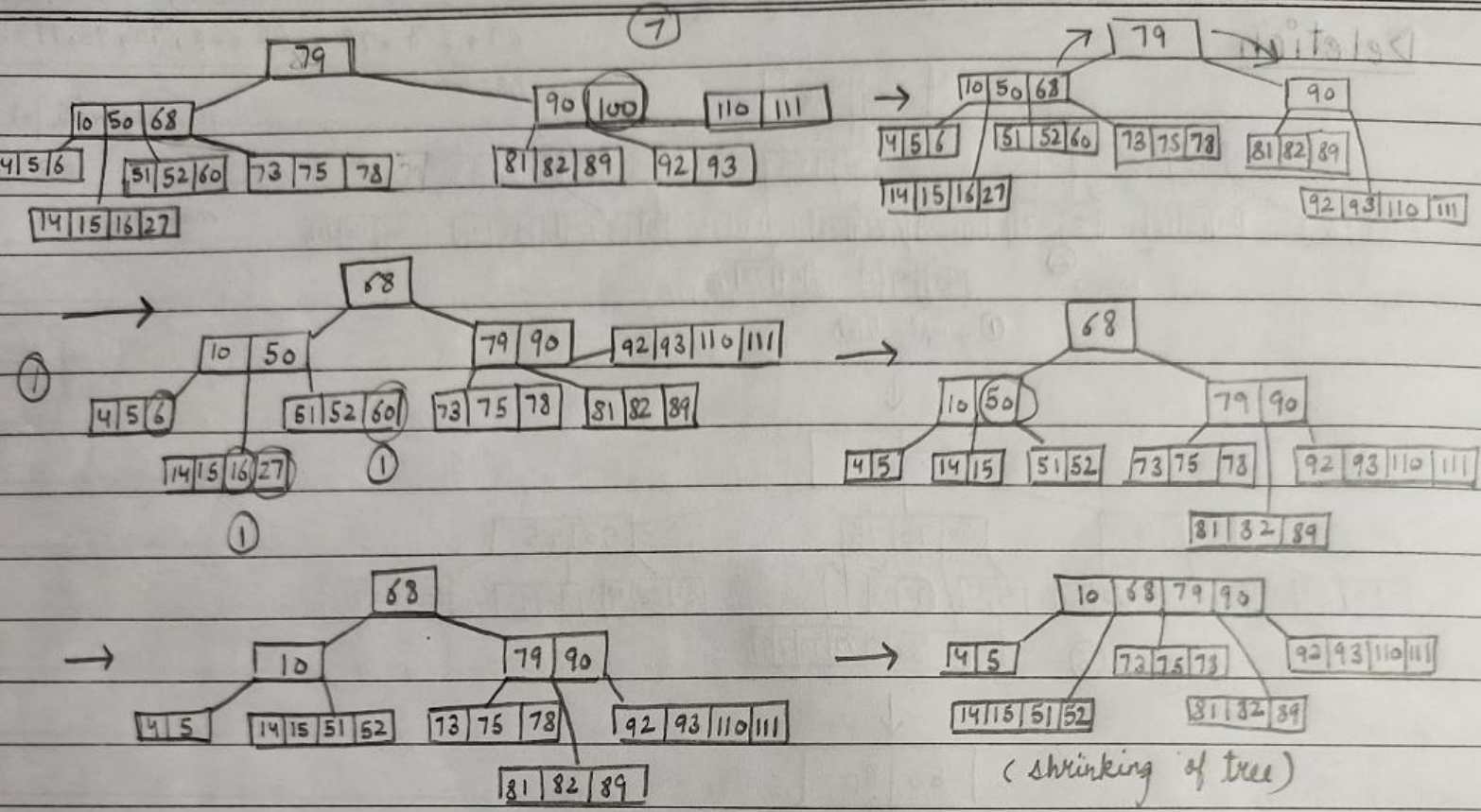
④







If while merging parent node gets short of keys → sibling otherwise merge



(shrinking of tree)

B^+ TREE

B tree → height is more
[bcz keys are also stored in internal nodes]

- Extension of B tree.
- ~~Only leaf nodes store the key values along with data pointers to disk file block~~
- Data is stored only in leaf nodes.
- Internal nodes consist of pointers and search keys.
- All the leaf nodes are linked together to form singly linked list.
- Height remains balanced & less as compared to B tree
- Data can be accessed sequentially as well as directly.
- Searching is fast because data is stored in leaf nodes only.
- Redundant search keys are present.
  internal nodes form the indices & leaf nodes (tht in secondary memory) constitute actual data
- search keys / data to the left of a node is strictly less than it and to the right, it is either equal to or greater than the node value.
- structure of internal nodes & leaf nodes is different.

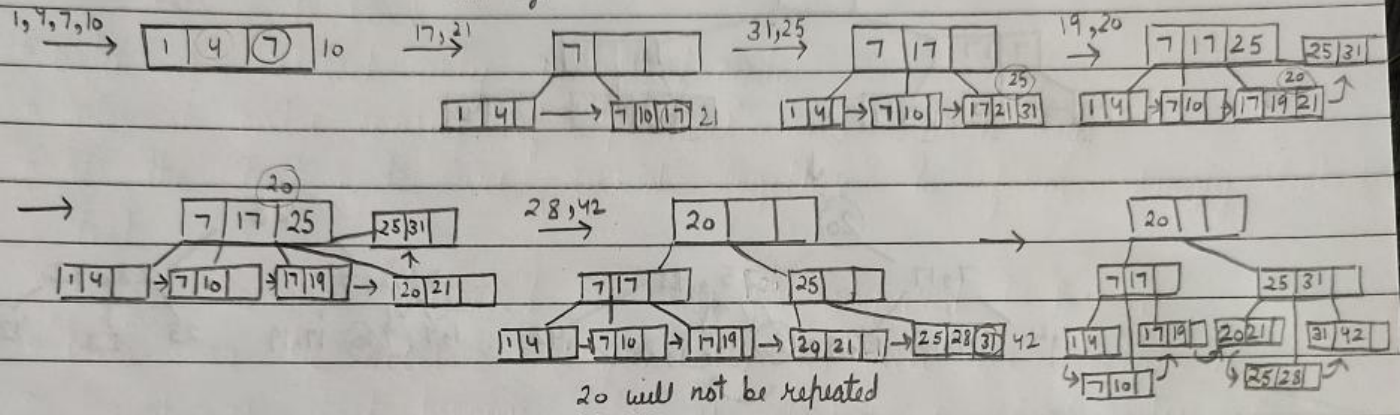
left strictly less
right = or greater
all leaf nodes are linked like LL

Insertion

→ m = 4 Insert 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42

max links = 4
Min links = 2

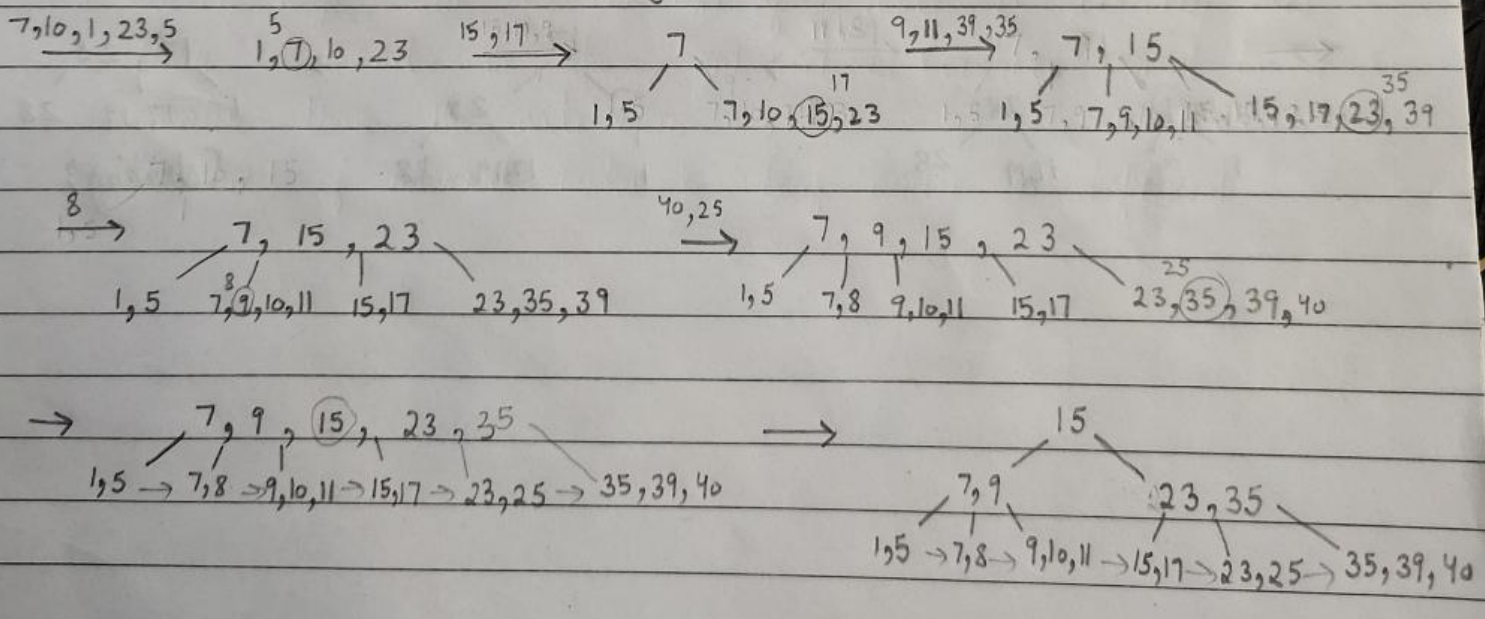
max keys = 3
min keys = 1



→ m = 5 Insert 7, 10, 1, 23, 5, 15, 17, 9, 11, 39, 35, 8, 40, 25

max children = 5
min children = 3

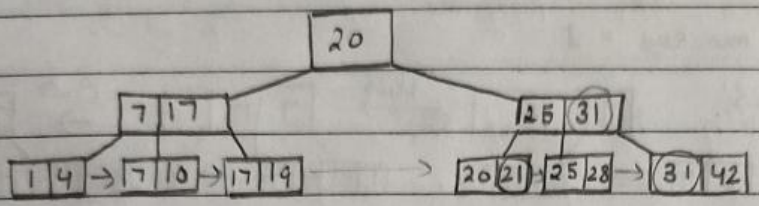
max keys = 4
min key = 2

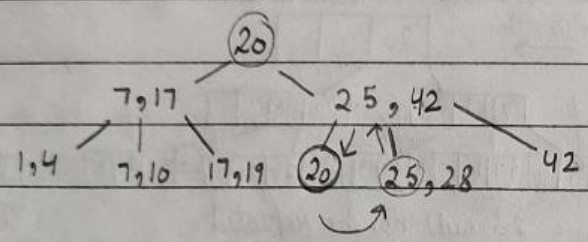


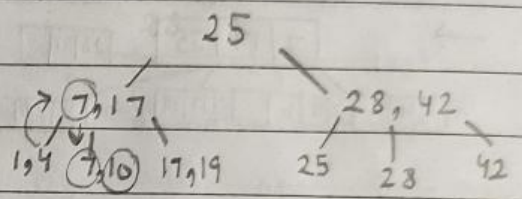
Deletion

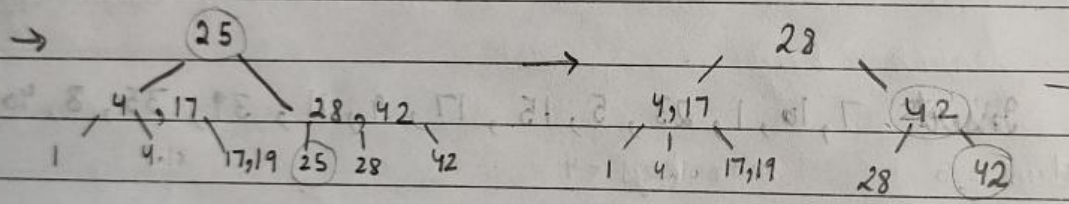
Max keys = 3
Min keys = 1
max links = 4
min links = 2

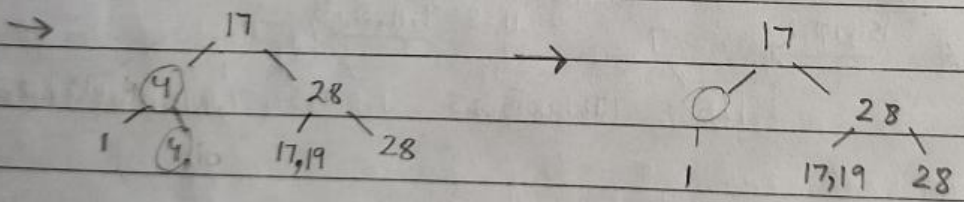
→ 21, 31, 20, 10, 7, 25, 42, 4







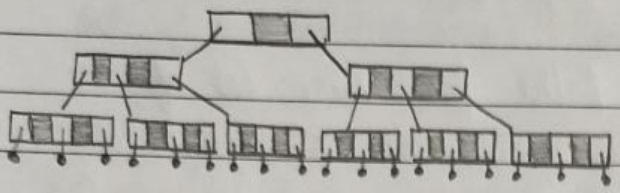




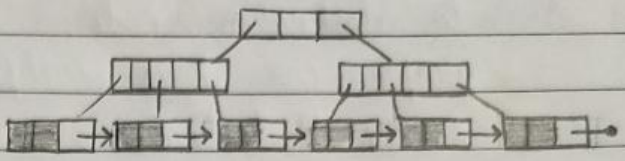
(shrinking)

B Tree vs B^+ Tree

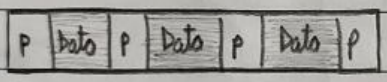
B Tree



B^+ Tree

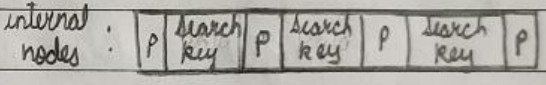


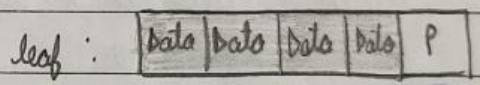
- data can be stored in internal as well as leaf nodes.
- leaf nodes are not connected to each other.
- structure of internal & leaf nodes is same.



- values can not be repeated in nodes.
- data can be found at any level ∴ searching is slow.
- Deletion can be complicated
- It usually has greater height / depth.

- data is only stored in leaf nodes.
- Leaf nodes are connected to each other forming a linked list.
- structure of internal & leaf nodes is different.





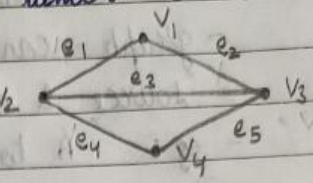
- Values can be repeated in nodes.
- data is present at the last level in leaf nodes ∴ searching is fast.
- Deletion involves removal of data from leaf nodes directly.
- It is usually smaller in height greater in width ← →

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

complete graph — weighted graph — weight on edges

GRAPH

A graph is a non-linear data structure which consists of set of nodes (vertices) and links (edges) which connects the vertices.

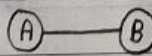


- $G = \{E, V\}$
  E = edges set V = Vertices set
- Here, E consists of edges e_i which is a unique (unordered) pair of nodes in V
  $e = (u, v)$
  $u, v \in V$

$V = \{v_1, v_2, v_3, v_4\}$
$E = \{e_1, e_2, e_3, e_4, e_5\}$
$e_1 = (v_2, v_1)$ or (v_1, v_2)
$e_2 = (v_1, v_3)$ or (v_3, v_1)
$e_3 = (v_2, v_3)$ or (v_3, v_2)
$e_4 = (v_2, v_4)$ or (v_4, v_2)
$e_5 = (v_4, v_3)$ or (v_3, v_4)

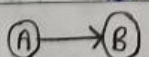
Terms:

- Vertex (node): individual data element in a graph.
  - isolated node : degree = 0
- edge (link): It is the link between two edges.
  - undirected edge: edge in which direction is not mentioned (bidirectional)



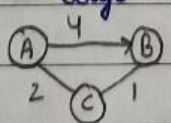
edge = (A, B) or (B, A)

  - directed edge: edge in which particular direction is specified (unidirectional)

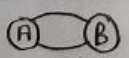


edge = (A, B) ≠ (B, A)

  - weighted edge: edge in which weight / cost is specified



  - parallel edges (multiple edges): two edges with same end vertices (undirected) / two edges with same origin & destination (directed)

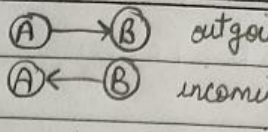


undirected

directed

Spiral

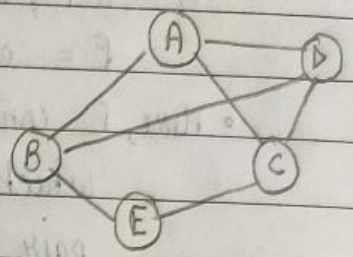
- outgoing edge : edge from origin
- incoming edge : edge to destination



outgoing for A
incoming for A

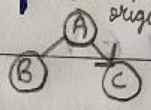
- path : sequence of continuous edges/nodes from source node to destination node.
  - graph can have multiple paths for same set of source & destination.
  - eg : A to E

    $A \to B \to E$, $A \to C \to E$, $A \to D \to C \to E$ etc.



  - simple path : path in which nodes don't repeat

- origin : If an edge is directed, its first end point is called its origin



- degree of node : Total no. of edges connected to a node is its degree.
  - indegree : total no. of incoming edges
  - outdegree : total no. of outgoing edges

  degree = indegree + outdegree

- size of graph : total no. of edges in the graph

  size = n(E)

- order of graph : no of vertices in the graph = n(V)
- adjacent vertices : nodes connected to the same edge.

  two nodes are called <u>end vertices</u> of the edge.

  eg: A & C , A & B , B & E etc (in above eg)

- cycle / loop : path which starts & ends at the same node is called a cycle

  eg : $A \to B \to E \to C \to A$ (closed path)

Types Of Graphs

- Directed graph : graph which contains directed edges.
  edges → arc
- undirected graph : graph which contains undirected edges.
- Mixed graph : graph which contains directed as well as undirected edges.
- weighted graph : graph which contains weighted edges.
  (labelled graph)
- simple graph : graph which do not contain loops & cycles.
- Complete graph : graph in which each node is connected to every other node in the graph

  no. of edges $= \frac{n(n-1)}{2}$

  (edges for first node) + (edges for 2nd node -1) + ---

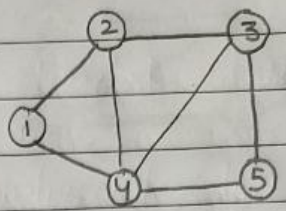
  $(n-1) + (n-2) + (n-3) + \cdots = n(n-1)/2$
- Trivial graph graph with single vertex & no edge
- null graph multiple vertices but no edges
- Connected graph : If there is a path from a node to every other node in the graph
- sub graph : A graph $G' = (V', E')$ is a sub-graph of $G = (V, E)$ if

  V' is a subset of V

  E' is a subset of E

Representation

→ Adjacency Matrix



n = 5

It is a matrix A[n][n]

n = no. of vertices (order)

& a[i][j] = 1 if i & j are adjacent

= 0 otherwise

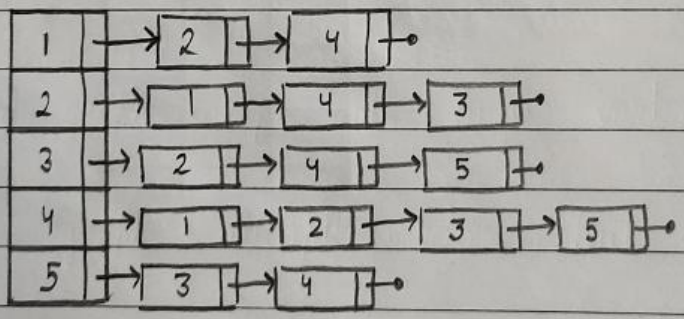
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	1	1
4	1	1	1	0	1
5	0	0	1	1	0

5×5

useful when graph is dense
≈ complete

space complexity = $O(n^2)$

→ Adjacency list



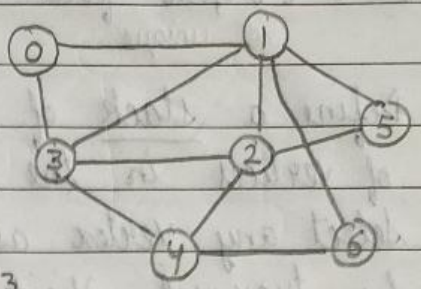
$O(n+2e)$

useful when graph is sparse

Traversals

BFS (Breadth First Search)

not unique

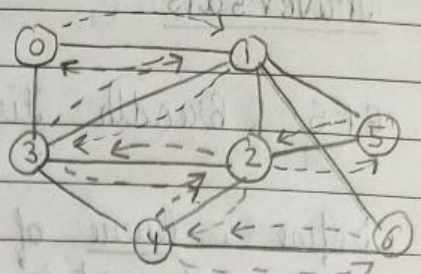


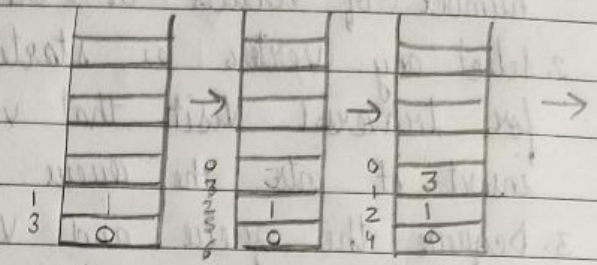
1. Define a Queue of size of total number of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
3. Dequeue the queue and visit (exploration) all the non-visited adjacent vertices of the dequeued vertex and insert them into the queue. Print the dequeued vertex.
4. When there is no new vertex to be visited from the dequeued vertex, dequeue the queue again
5. Repeat steps 3 and 4 until queue becomes empty.

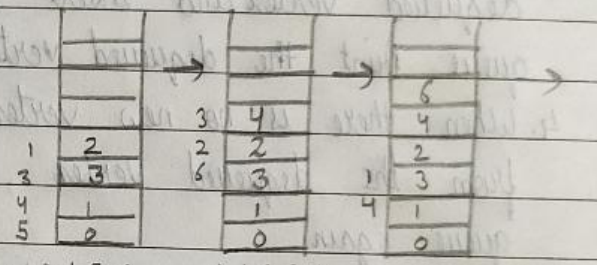
1, 3	~~0~~ 1 3	0
0, 3, 2, 6, 5	~~0~~ ~~1~~ 3 2 5 6	0, 1
0, 1, 2, 4	~~0~~ ~~1~~ ~~3~~ 2 5 6 4	0, 1, 3
1, 3, 4, 5	~~0~~ ~~1~~ ~~3~~ ~~2~~ 5 6 4	0, 1, 3, 2
1, 2	~~0~~ ~~1~~ ~~3~~ ~~2~~ ~~5~~ 6 4	0, 1, 3, 2, 5
1, 4	~~0~~ ~~1~~ ~~3~~ ~~2~~ ~~5~~ ~~6~~ 4	0, 1, 3, 2, 5, 6
3, 2, 6	~~0~~ ~~1~~ ~~3~~ ~~2~~ ~~5~~ ~~6~~ ~~4~~	Result: 0, 1, 3, 2, 5, 6, 4

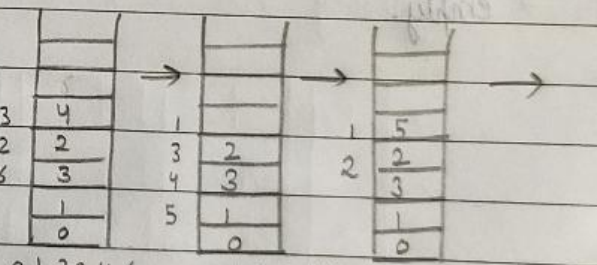
DFS (Depth first Search)
not unique

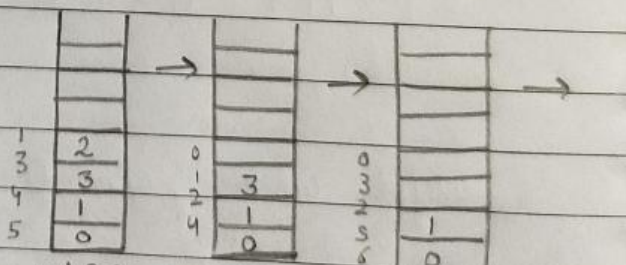
1. Define a stack of size total no. of vertices in the graph.
2. Select any vertex as starting point for traversal. Visit that vertex and push it on to the stack ~~Visit~~ Also print the vertex.
3. Visit any one of the non-visited adjacent vertices of the vertex which is at the top of stack
4. Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top.
5. When there is no new vertex to visit, use back tracking & pop one vertex from the stack.
6. Repeat steps 3,4,5 until stack becomes empty.

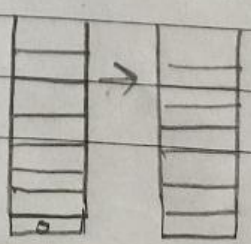












result = 0,1,3,2,
4,6,5

Spiral

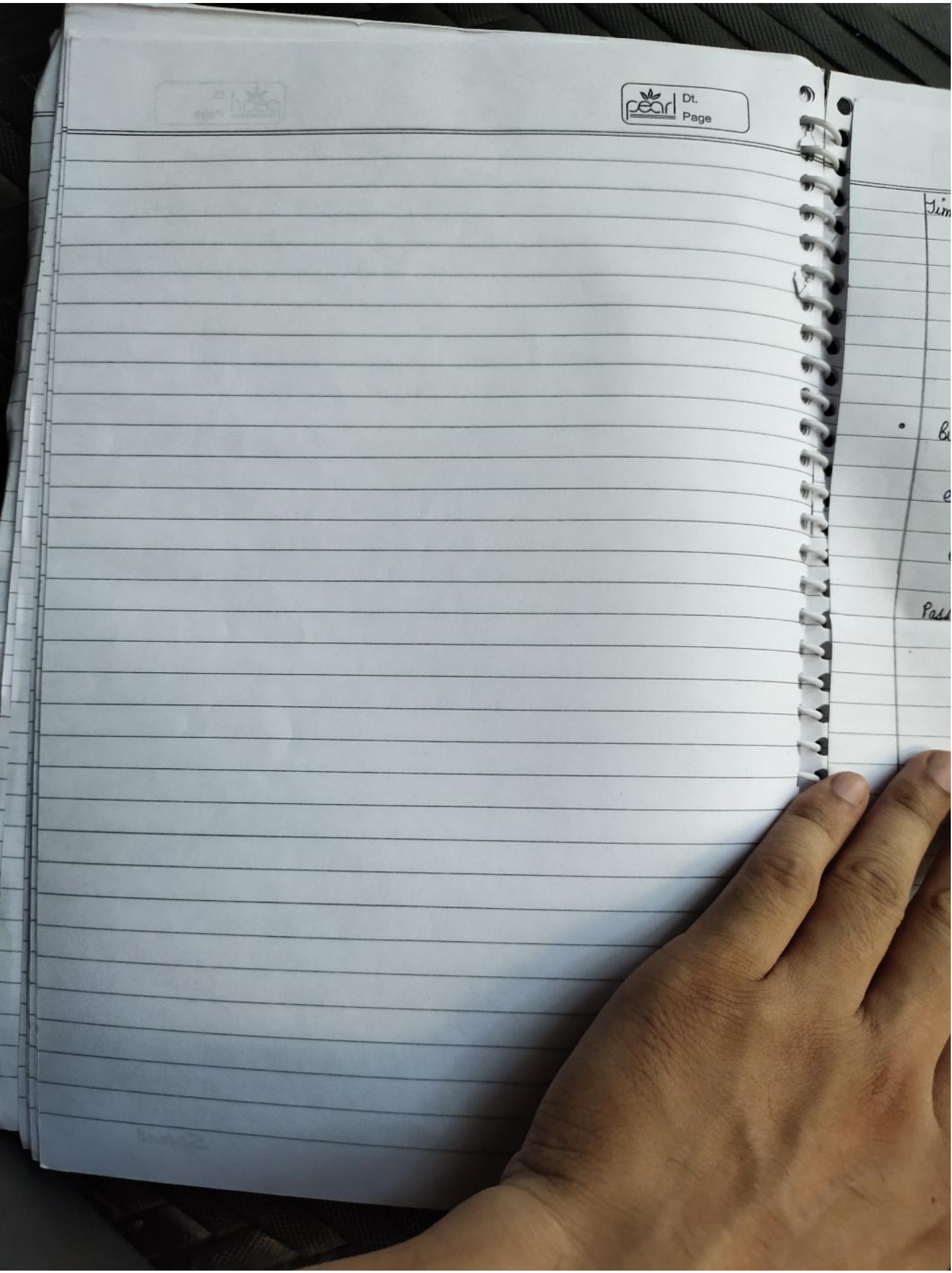
remove loops

remove parallel edges with comp greater cost

choose any vertex

check outgoing / incident edges. Choose min wt edge

Arrange in inc order of wts


Dt.
Page

Time Complexity :

- Bubble sort : Each element is compared to its succeeding element (in multiple passes) positioning the greatest element at the end.

eg: 33 11 22 55 44

Pass : 1 33 11 22 55 44

11 33 22 55 44

11 22 33 55 44

11 22 33 44 55

9 5 10 2 1 4

5 9 10 2 1 4

5 9 2 10 1 4

5 9 2 1 10 4

5 9 2 1 4 (10)

5 2 9 1 4 10

5 2 1 9 4 10

5 2 1 4 (9) 10

2 5 1 4

2 1 5 4

2 1 4 (5)

1 2 (4)

1 2

Spiral

→ `i = 0` — 1 unit

`while ( i < n)` — n+1

`{ stmt ;` — n units

`i++ ;` — n

`}`

$3n+2 = O(n)$

→ For finding hcf

```
while ( m != n)
  { if (m > n)
      m = m - n;
    else
      n = n - m ;
  }
```

m	n	no. of times	
5	5	0	≈ 5/5
3	6	1	≈ 6/3
16	2	7	≈ 16/2

as if - else conditional statements comes into action, we get best / avg / worst cases

Best case for this algo is $O(1)$ $\frac{m}{n} \approx 1$

worst case $O(n)$ $m >> n$ so $\frac{m}{n} \approx m$

→

```
p = 0
for (i=1; i<n; i = i*2)          → log2 n
  { p++;
  }
for (j=1; j<p; j = j*2)          → log2 (log2 n)
  { -
  }
```

$\log n + \log(\log n)$

$O(\log(\log n))$

->

```
for (i=0; i<n; i++)              → n+1
  { for (j=1; j<n; j = j*2)      → n log n
    { -                          → n (log n)
    }
  }
```

R	i
1	0
2	1
3	2
:	:
R	R-1

$R-1 < n \quad R < n+1$

$2 < 3 \quad 3 \quad 3+1$

$O(n \log n)$

R=3

Conclusions :

- for (i=0; i<n; i++) n
- for (i=0; i<n; i = i+2) n/2 } $O(n)$
- for (i=n; i>1; i--) n

- for (i=1; i<n; i = i*2) } $O(\log_2 n)$
- for (i=n; i>1; i = i/2

- for (i=1; i<n; i = i*3) } $O(\log_3 n)$

$\frac{\log_x a}{\log_x b} = \log_b a$

→ for(i=1; i<n; i=i*2)
{ — ;
}

i	2^k
1	2^0
2	2^1
4	2^2
8	2^3
:	:
	2^k

n=8	n=10
i	i
1	1
2	2
4	4
8 X	8
	16 X
③	④

$2^k < n \Rightarrow k \log 2 < \log n$

$\Rightarrow k < \frac{\log n}{\log 2} = \log_2 n$

$e^x = b$

$x = \log_e b$

$2^k = n$

$k = \log_2 n$

$O(\lceil \log_2 n \rceil)$
ceil of log

for $i * x \rightarrow \log_x n$

→ for(i=n; i>=1; i=i/2)
{ — ;
}

i
n
n/2
n/4
n/8

$\frac{n}{2^k} >= 1 \Rightarrow 2^k < n$

$k < \log_2 n$ $\quad O(\log_2 n)$

→ for(i=0; i*i<n; i++)
{ —
}

i	i*i
0	0
1	1
2	4
3	9
:	:
k	k^2

$k^2 < n$

$k < \sqrt{n}$

$O(\sqrt{n})$

→ for(i=0; i<n; i++) — n
{ —
}

for(j=0; j<n; j++) — n
{ —
}

$n + n = 2n$ $\quad O(n)$

Page

• Multiplication of 2 matrices

```
Multiply ( A, B, n)
{
    for( i = 0 ; i < n; i ++)                  → n+1
    {
        for ( j =0 ; j < n ; j++)              n (n+1)
        {   C[i, j] = 0;                       n x n
            for (k=0 ; k < n ; k++)            n x n x (n+1)
            {
                C[i, j] = C[i, j] + A[i, k] * B[k, j]
                                               n x n x n
            }
        }
    }
}
```

A → n^2
B → n^2
n → 1
i → 1
j → 1
C → n^2
k → 1
$3n^2+4$

$2n^3+3n^2+2n+1$
$O(n^3)$

T.C. : $O(n^3)$
S.C. : $O(n^2)$

∴ → for(i =0 ; i<n , i ++)   skip n+1 not req
{   sout _   — n $O(n)$
}

i=0 - - - - n-1

i	
1	0
2	0+2
3	~~2+2+2~~
4	0+2+2+2
k	0+2+2+ - - (k-1)

0+(k-1)2
2k-2 < n
k < $\frac{n+2}{2}$

→ for (i=0; i < n , i=i+2)
{   1 statement $\frac{n}{2}$ $O(n)$
}

1	0
2	0+2 =2
3	0+2+2 =4

→ for (i =0; i <=n ; i ++)   - n+1
{   for (j=0 ; j<i ; j++)   1+2+3 - - +n = $\frac{n(n+1)}{2}$
    {   —   }
    }
}
$O(n^2)$

i=0; i<5; i++ →

```
p=0;
for ( i=1 ; p <= n , i ++)
{   p = p + i ;
}
```

4<5
i<4

i	p
1	0+1
2	1+2
3	1+2+3
4	1+2+3+4
⋮	⋮
k	1+2+3+4+ - - +k

$O(\sqrt{n})$

when p <= n   (stops if k > √n)
$\frac{k(k+1)}{2}$ <= n ⇒ k^2 <= n ⇒ k <= √n

Frequency Count method

[for time complexity & space]

```
sum(A, n)
{   S = 0                          → 1
    for(i = 0; i < n; i++)         → n+1      (n+1 : n)
    {   S = S + A[i];              → n
    }
    return S;                      → 1
}
```

$f(n) = 2n + 3$

$O(n)$ (order of n)

we consider only middle one

- time complexity = $O(n)$
- space —— = $S(n) = n + 3$ ≈ $O(n)$

  A → n , S → 1 , i → 1 , n → 1

- sum of 2 matrices

```
Add(A, B, n)
{
    for(i = 0; i < n; i++)                    → n+1
    {
        for(j = 0; j < n; j++)                → n(n+1)
        {   C[i,j] = A[i,j] + B[i,j];         → n^2
        }
    }
}
```

A → n^2
B → n^2
n → 1
C → n^2
i → 1
j → 1

$3n^2 + 3$

$f(n) = 2n^2 + 2n + 1$

$O(n^2)$

time complexity : $O(n^2)$

space —— : $O(n^2)$

Analysis of an Algorithm

1) Time - taken by algo to execute
2) Space - to used by an ——
3) data transferred
4) Power consumption
5) CPU registers consumed

1) Time Analysis we assume every single statement takes 1 unit of time

eg: • Swap (a, b)

start

temp = a — 1 unit
a = b — 1 unit
b = temp — 1 unit

3 units O(3)

end

Space analysis :
3 var = a, b, temp
O(3)

x = 5 * a + 6 * b

at Comp. side 4 step will be executed but we take it as a single statement & time taken → 1 units

{ bcz we do only brief analysis

→ We can do detailed analysis in case it is required But generally we focus on brief analysis only

2) Space to Analysis

above eg use 3 variables
so space complexity = O(3)

Algorithm	Program
- Design	- Implementation
- Domain knowledge req.	- Programmers
- any lang. (eng)	- Prog. lang
- H/w & OS ~~no~~ doesn't matter	- H/w & O/s matters
- Analyse	- Testing

Priori Analysis	Posteriori Testing
- Algo	- Program
- Indep. of lang.	- lang. dep.
- OS/ Hardware ~~ds~~ ind.	- OS/ Hardware dep.
- Time & space function	- watch time & bytes

Characteristics of algos
- input - takes some inputs
- generates desired output (atleast 1)
- Definiteness - statement must be unambigious (must not be invalid)
- Finiteness - must terminate
- Effectiveness - must not write unnecessary statements
- better if lang independent

There is no specific syntax to write an algo

- multi int
- pass arrays
- eval ^

algos (english) → program (programming lang)

Dt. Page

Data Structures

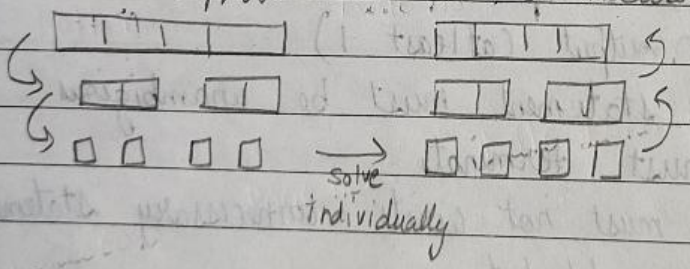
A - concept of Data Structures
- choice of right DS
- Algos - how to design & develop algo
- Complexity of algos
- operations - insertion, deletion, traversal etc.
- Analysis of algos
- Searching - linear / Binary techniques & their complexity analysis
- ~~Techniques &~~

Algorithm

It is a set of finite steps to solve a given computational problem

Design
- Incremental approach : start → inc → end / to a certain pt
  sub & conquer
  eg : selection, insertion, sorts

- Divide & conquer approach : we divide a large problem into sub problems & solved recursively



★ Steps to design an algorithm
- understand the problem
- Identify the output of the problem
- Identify the inputs required by the problem & choose the data structures
- Design a logic that will produce the desired output from the given inputs.
- Test the algo for diff sets of input