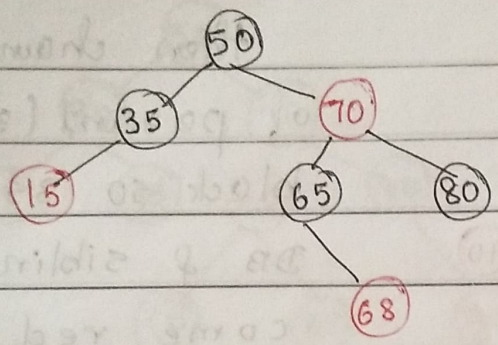
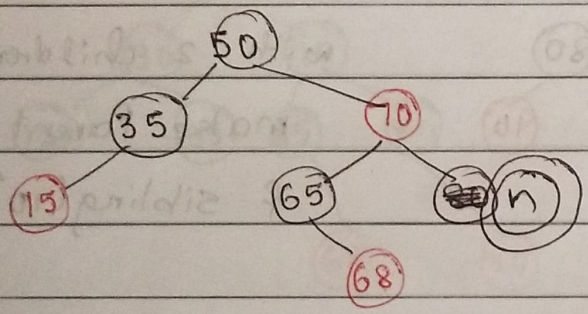
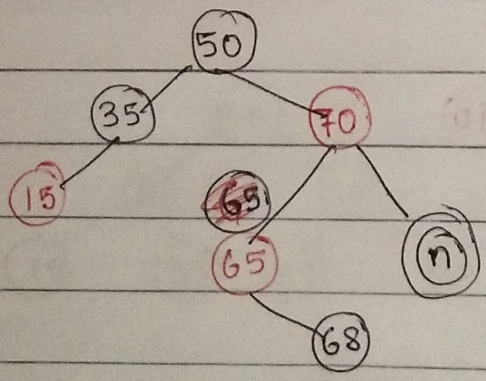


90 is red. So Delete 90.

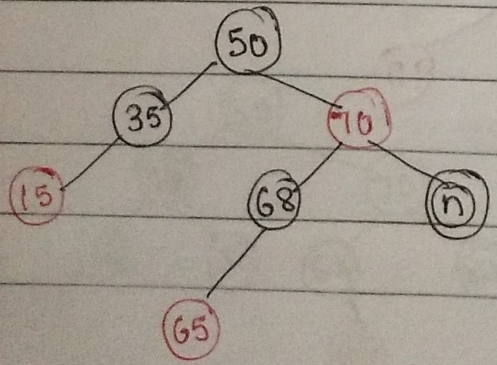
4) Delete 80



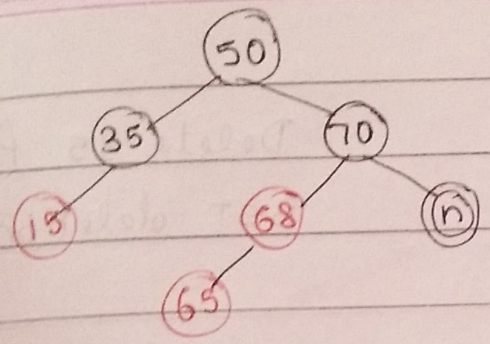
80 leaf Node, black DB arises. Sibling (65) is black. Near child (68) is Red. case 5



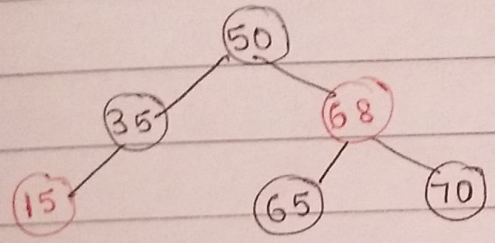
swap color of DB's sibling (65) & Sibling's child (68).



Rotate sibling (65) in opp. direction to DB

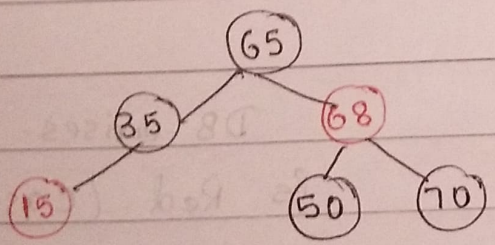


Apply case 6
DB's parent (70) & sibling (68) swap color

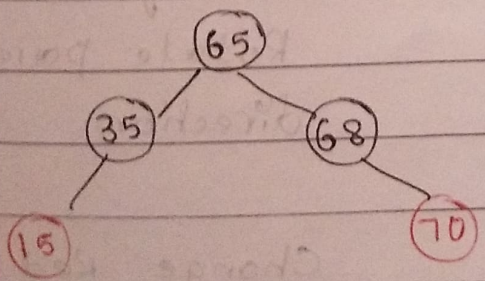


Rotate parent (70) in DB's direction.
Change red child of Sibling (65) to Black

5) Delete 50

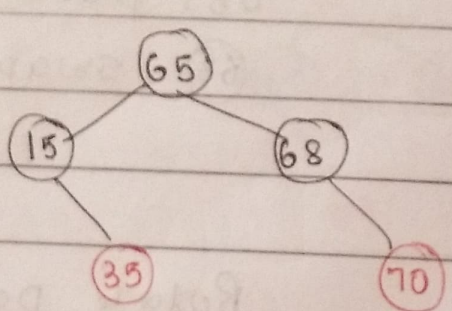


50 is a Root Node
2 child - BST deletion

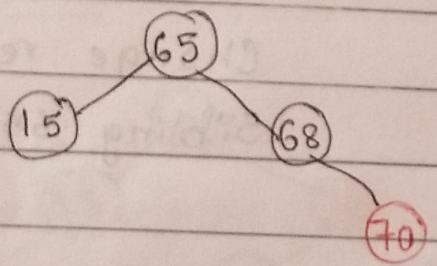


DB arised - check Sibling (70) is black with black child (NULL)
case 3: 68 become black & 70 become Red

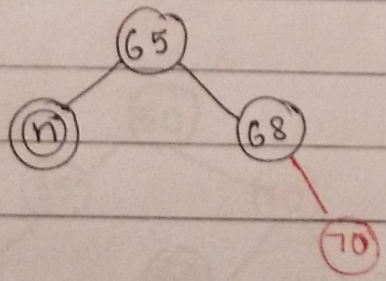
6) Delete 35



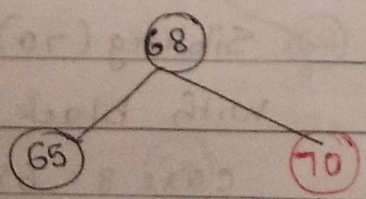
Delete 35. Perform BST deletion



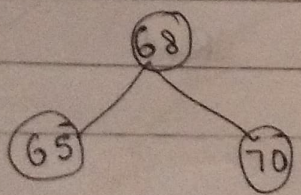
7) Delete 15



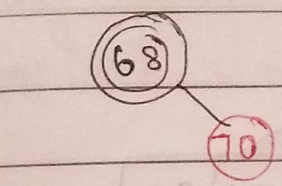
DB arises. Far child is Red (Case 6)
DB's parent (65) & sibling 68 — swap colors
Rotate parent in DB's direction



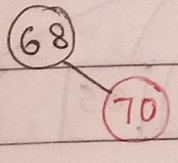
Change Red child to Black



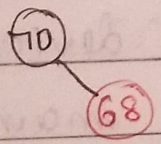
8) Delete 65



65 is black. DB. sibling is black(70) case 3.
Parent(68) color DB
Sibling(70) Red.



9) Delete 68



68 have one child
BST deletion

23/1/2020

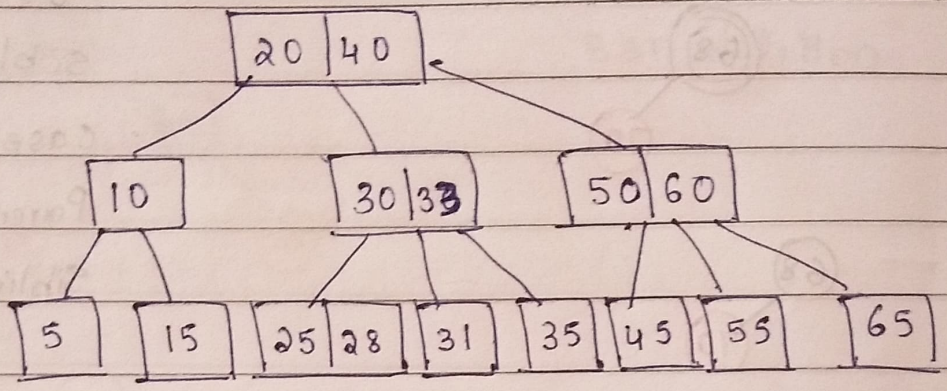
B-tree (SLIDE CLASS ROOM)

- B-tree is a special type of self-balancing search tree in which each node can contain more than one key & can have more than ~~one~~ two children.
- It is a generalised form of BST
- It is also known as a height-balanced

Secondary Storage

If n keys, n+1 children

m way tree.



- B-trees are balanced search trees designed to work well on magnetic disks or other direct access secondary storage devices
- B-tree nodes may have many children. This is called 'branching factor' of B-tree.
- Every n-node B tree has height O(log n), therefore, B-trees can be used to implement many dynamic-set operations in time O(log n).

B-Tree properties

1) Every node x has the following fields:

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

b) the n[x] keys themselves, stored in nondecreasing order $key_1[x] \leq key_2[x] \ldots$

c) leaf[x], → TRUE if x is a leaf
→ FALSE if x is a internal node.

2) if x is an internal node, it also contain n[x]+1 pointers $c_1[x], c_2[x] \ldots$ to its children.

3) the keys $key_i[x]$ separate the ranges of keys stored in each subtree with root $c_i[x]$, then

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

4) Every leaf has 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. These bounds can be expressed in terms of a fixed integer $t \geq 2$ called minimum degree of B-tree.

a) Every node other than the root must have atleast t-1 keys. Every internal node other than the root thus has atleast t children.

b) Every node can contain at most 2t-1

keys. ∴ an internal node can have at most 21 children.

Searching a B-tree

B-TREE-SEARCH (x, k)

1. i ← 1
2. while i ≤ n[x] and k ≥ key_i[x]
3. do i ← i + 1
4. if i ≤ n[x] and k = key_i[x]
5. then return (x, i)
6. if leaf[x]
7. then return NIL
8. else DISK-READ(c_i[x])
9. return B-TREE-SEARCH (c_i[x], k)

Search is similar to the search in BST. Let the key to be searched be k. We start from the root & recursively traverse down.

For every visited non-leaf node, if the node has the key, return the node.

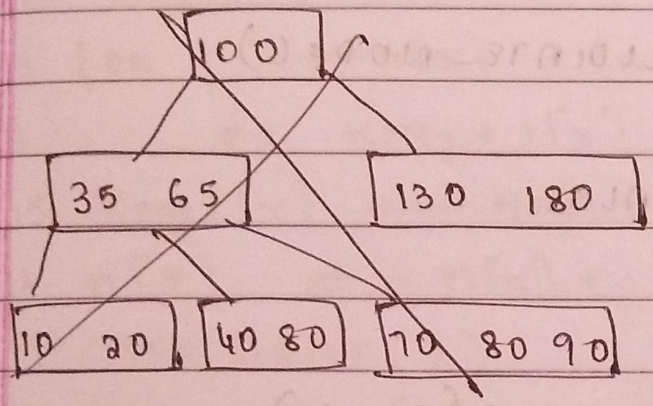
else, we recurse down to the app-

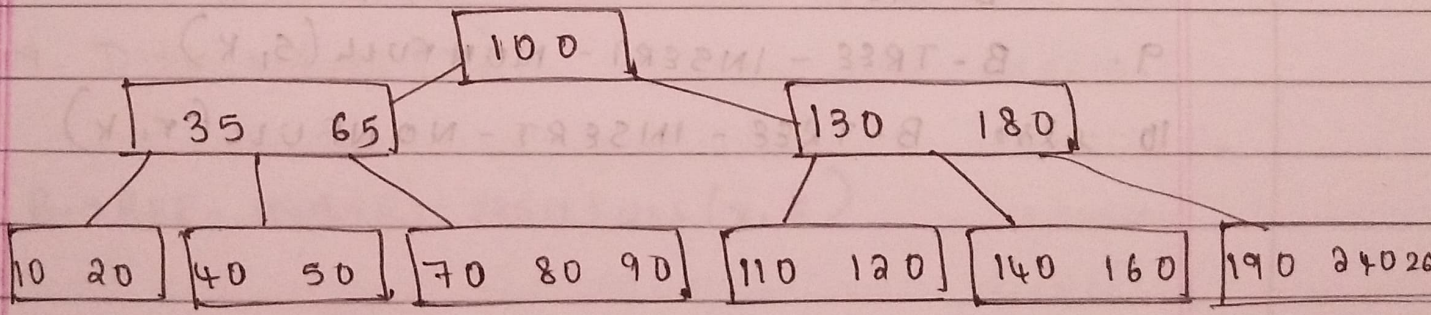
Page No :
Date :

ropriat child of the node.

At each level, the search is optimised as if the key is present in another branch.

Eg: Searching 120 in the given B-Tree





Step 1 : First search will start at Root Node.

Step 2 : • check if 120 ≥ 100 and ~~120~~ 120 < 130. So it has to be in left branch of B-Tree.

• control goes to [130 180]

Step 3 : • So 120 < 130, so it has to be in the rightmost child node of current parent

• control ⟶ [110 120]

• No get the value

Page No :
Date :

25/1/2021

Inserting into B-tree

```
B-TREE-INSERT(T, k)

1. r ← root(T)
2. if n[r] = 2t - 1
3.    then s ← ALLOCATE-NODE()
4.         root(T)  s
5.         leaf[s]  FALSE
6.         n[s] ← 0
7.         c1[s] ← r
8.         B-TREE-SPLIT-CHILD(s, 1, r)
9.         B-TREE-INSERT-NONFULL(s, k)
10. else B-TREE-INSERT-NONFULL(r, k)
```

```
B-TREE-SPLIT-CHILD(x, i, y)

1. z ← ALLOCATE-NODE()
2. leaf[z] ← leaf[y]
3. n[z] ← t-1
4. for j ← 1 to t-1
5.    do keyj[z] ← keyj+t[y]
6. if not leaf[y]
```

```
7.  then for j ← 1 to t
8.       do c_j[z] ← c_j + t[y]
9.  n[y] ← t - 1
10. for j ← n[x] + 1 downto i + 1
11.       do c_j+1[x] ← c_j[x]
12. C_i+1[x] ← 2
13. for j ← n[x] downto i.
14.       do Key_j+1[x] ← Key_j[x]
15. Key_i[x] ← Key_t[y]
16. n[x] ← n[x] + 1
17. DISK_WRITE(y)
18. DISK_WRITE(z)
19. DISK_WRITE(x)
```

B-TREE-INSERT-NONFULL(x, k)

```
1.  i ← n[x]
2.  if leaf[x]
3.       then while i ≥ 1 and k < Key_i[x]
4.            do Key_i+1[x] ← Key_i[x]
5.               i ← i - 1
6.       Key_i+1[x] ← k
7.       n[x] ← n[x] + 1
8.       DISK_WRITE(x)
```

```
9 else while i ≥ 1 and k < key i[x]
10.        do i ← i-1
11.  i ← i+1
12.  DISK-READ (ci[x])
13.  if n[ci[x]) = 2t - 1
14.        then B-TREE-SPLIT-CHILD (x, i, ci[x])
15.           if k > key i[x]
16.              then i ← i+1
17.     B-TREE-INSERT-NONFULL (ci[x], k)
```

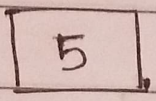
1) If a tree is empty, allocate a root node & insert the key.
2) Update the allowed number of keys in the node.
3) Search the appropriate node for insertion
4) If the node is full, follow the steps below
5) Insert the elements in increasing order.
6) Now, there are elements greater than its limit. So, split at the median.
7) Push the median key upwards & make the left keys as left child & right keys as right child.
8) If node is not full, insert the node in the increasing node.

Page No :
Date :

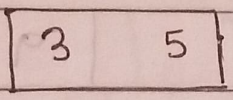
Eg: Construct a BTree of order 4 with the following values.

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

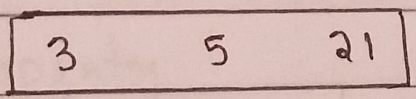
Insert 5



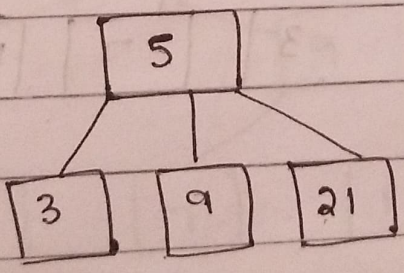
Insert 3



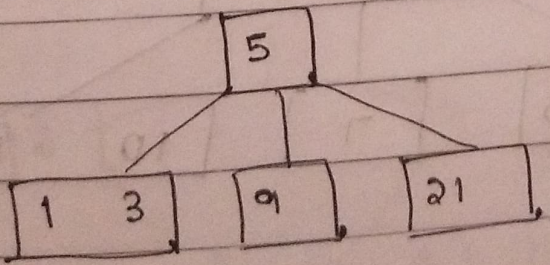
Insert 21



Insert 9



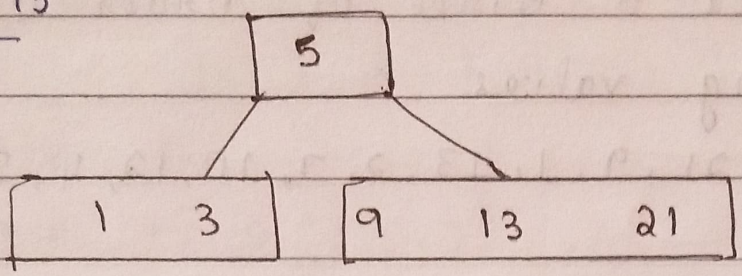
Insert 1



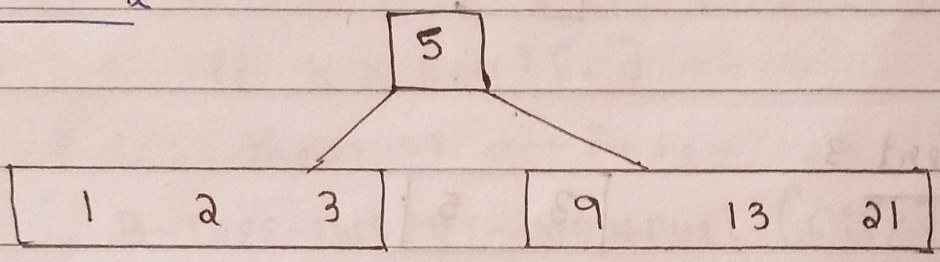
Page No :

Date :

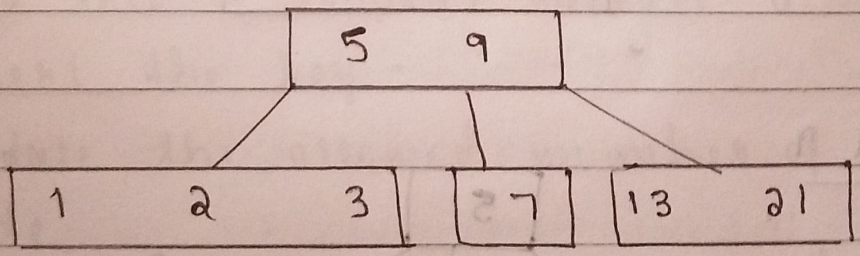
Insert 13



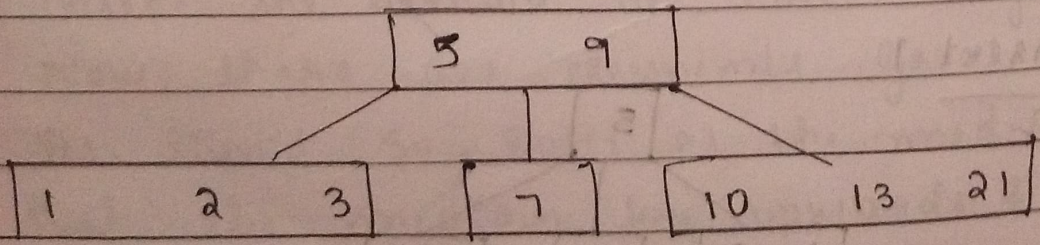
Insert 2



Insert 7

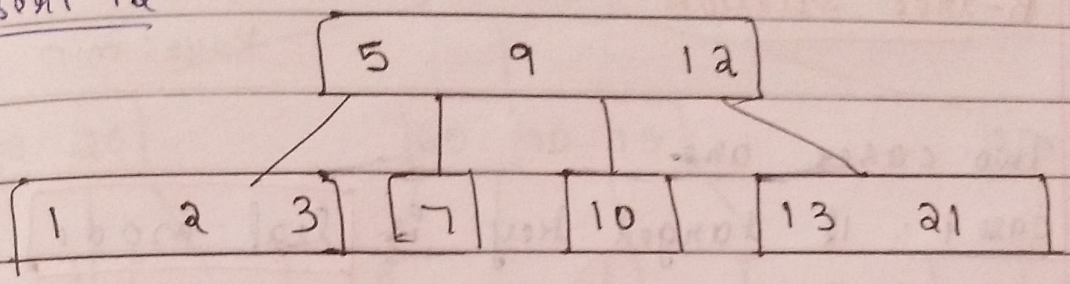


Insert 10

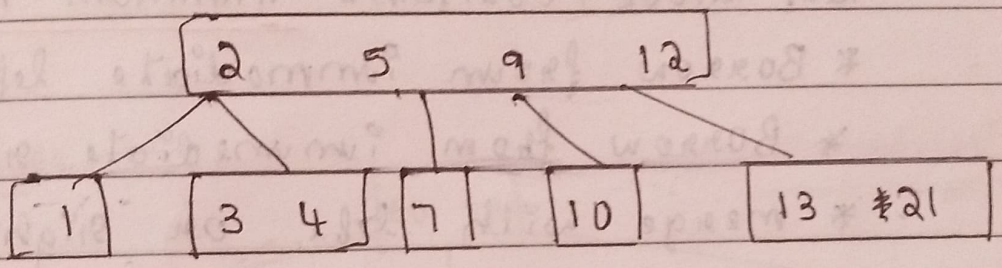


Page No :
Date :

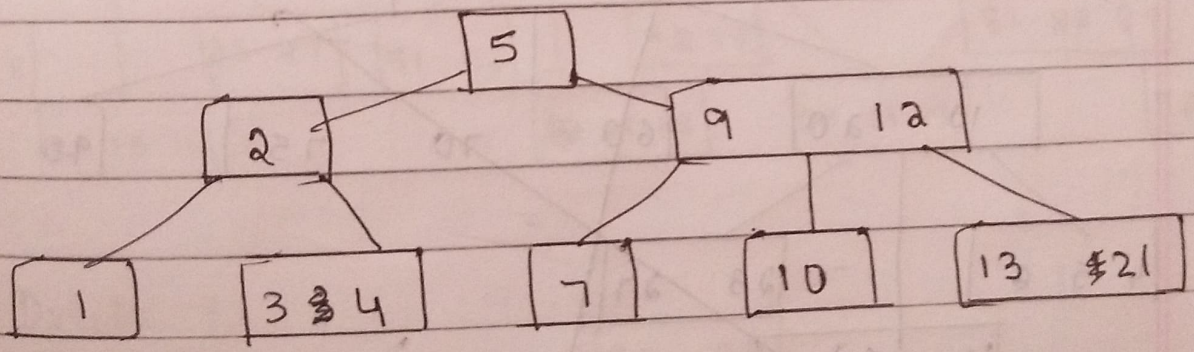
Insert 12



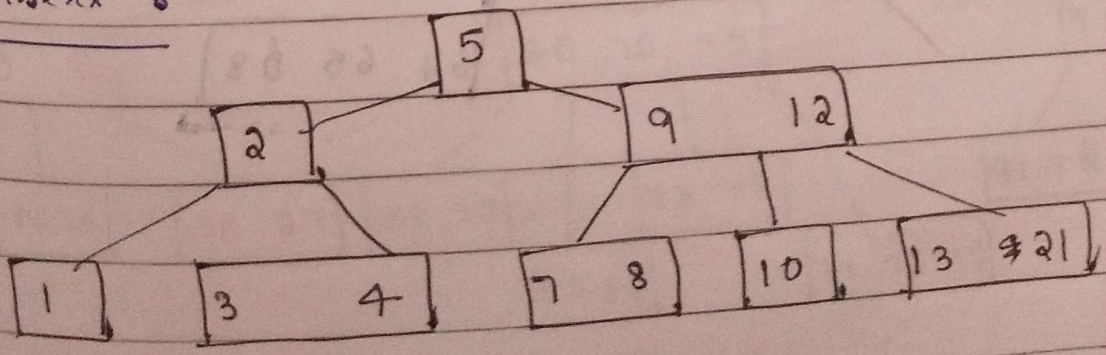
Insert 4



Violates Property :



Insert 8



order

Keys = min $-(\lceil m/2 \rceil - 1)$

max $-(m-1)$

children = min $-\lceil m/2 \rceil$

max $- m$

eg: order = 5

Keys: min $(\lceil 5/2 \rceil - 1)(3-1)$

Page No :

Date :

10/1/2021

B-tree Deletion

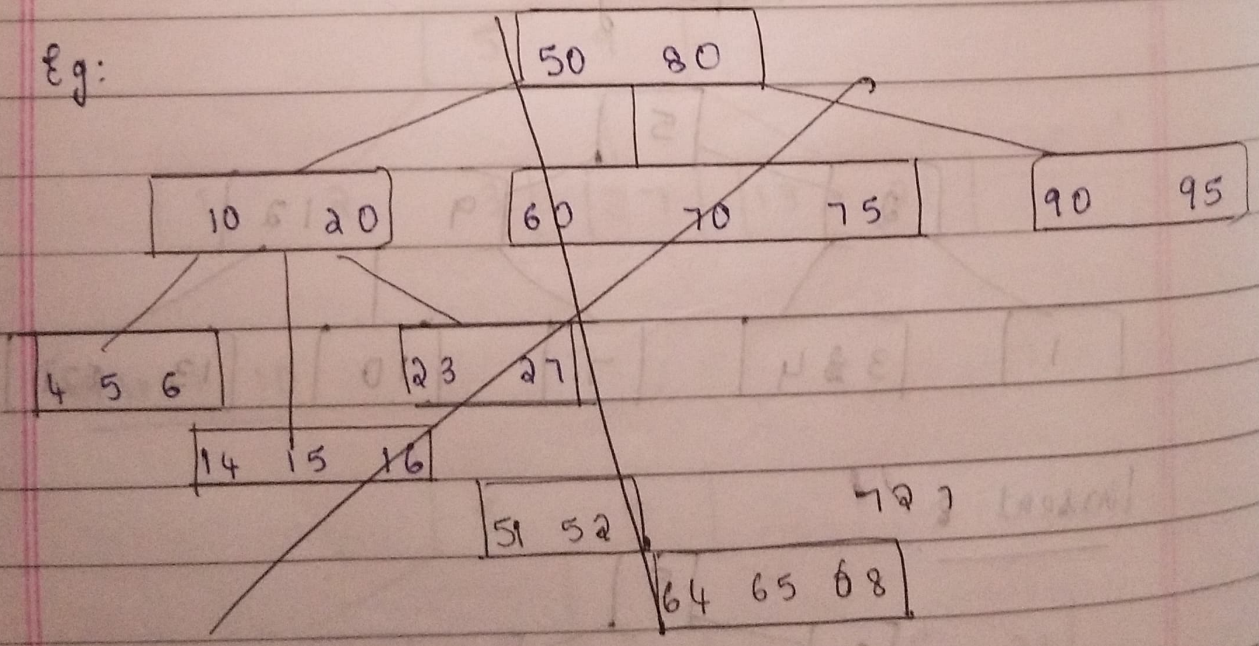
Two cases are

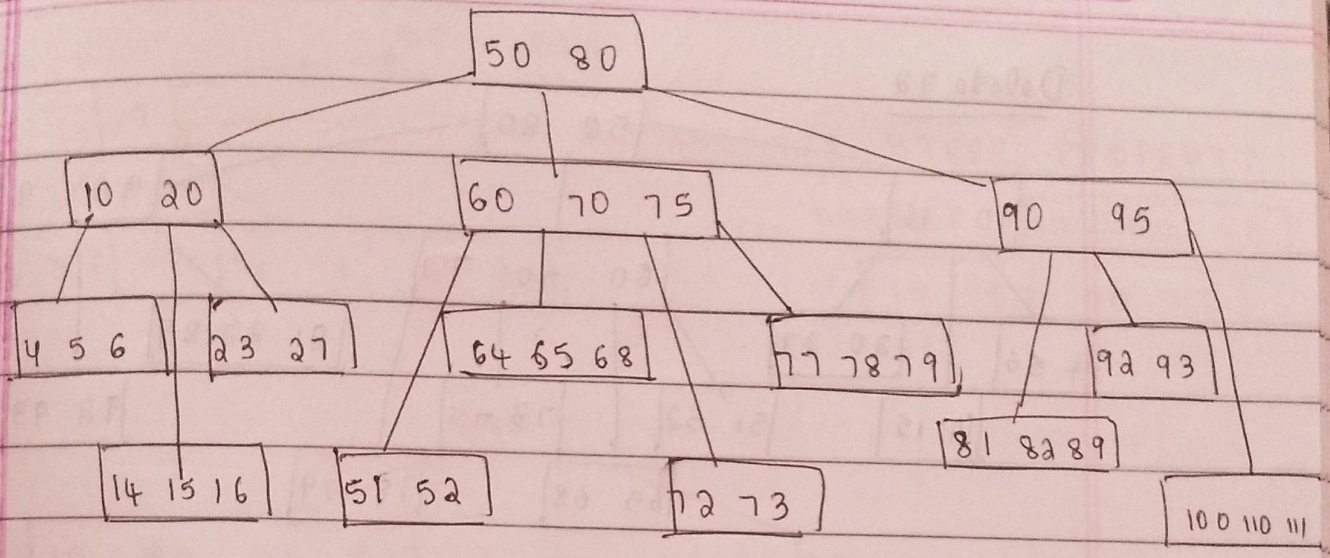
case 1: If target key is **leaf node**

- Leaf node contains more than minimum no. of keys.
- Leaf node contains minimum no. of keys
  - * Borrow from immediate left Node
  - * Borrow from immediate right Node
  - * merge with left or right child

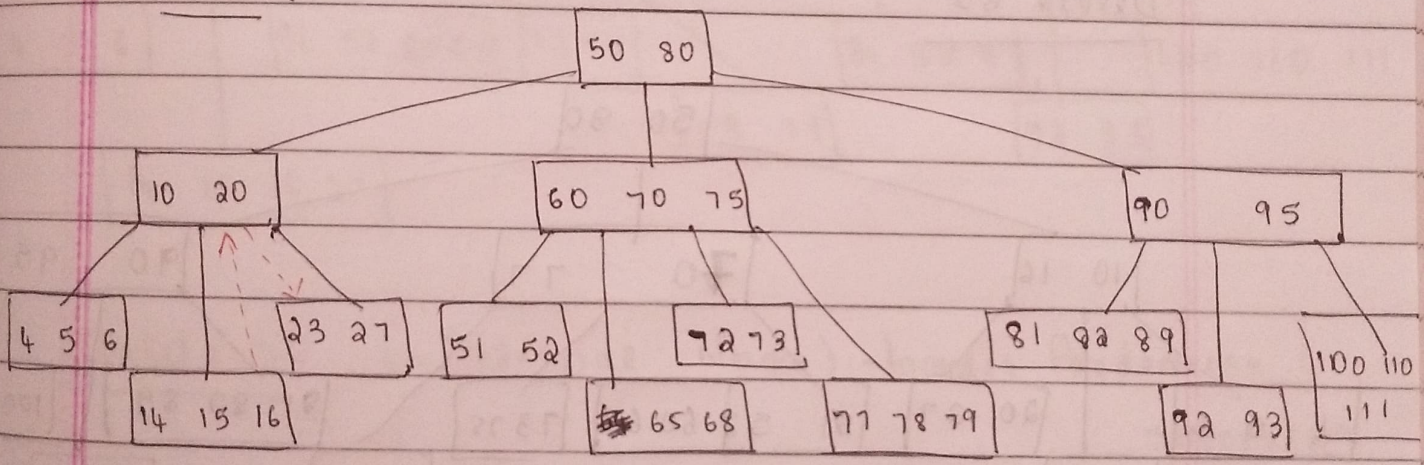
case 2: If target key is **internal Node**

eg:

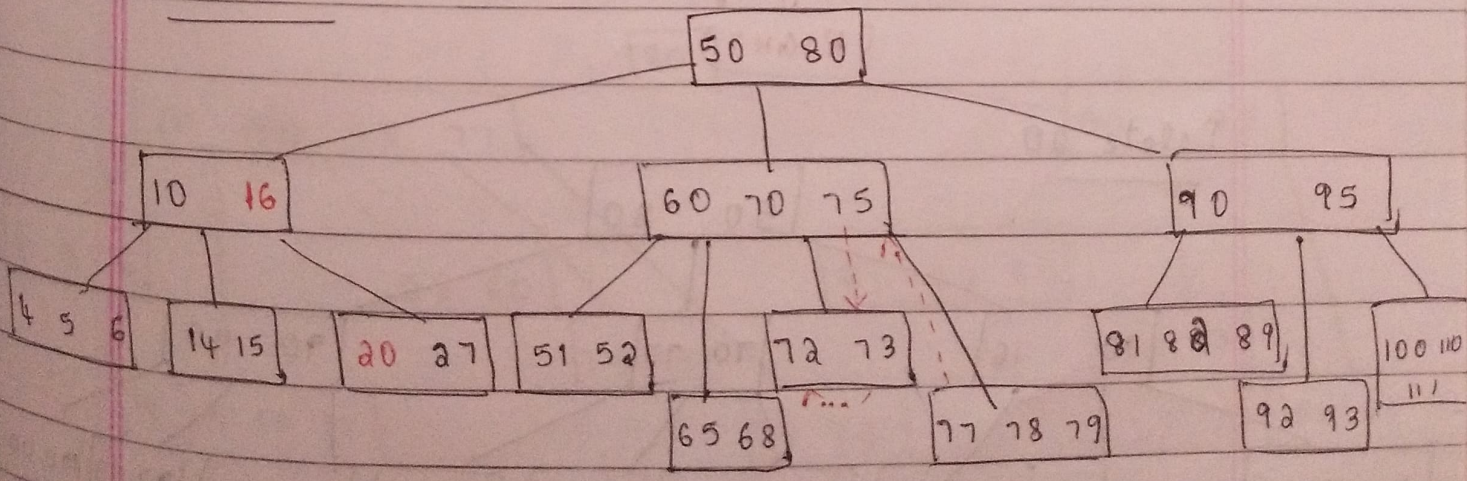




Delete 64

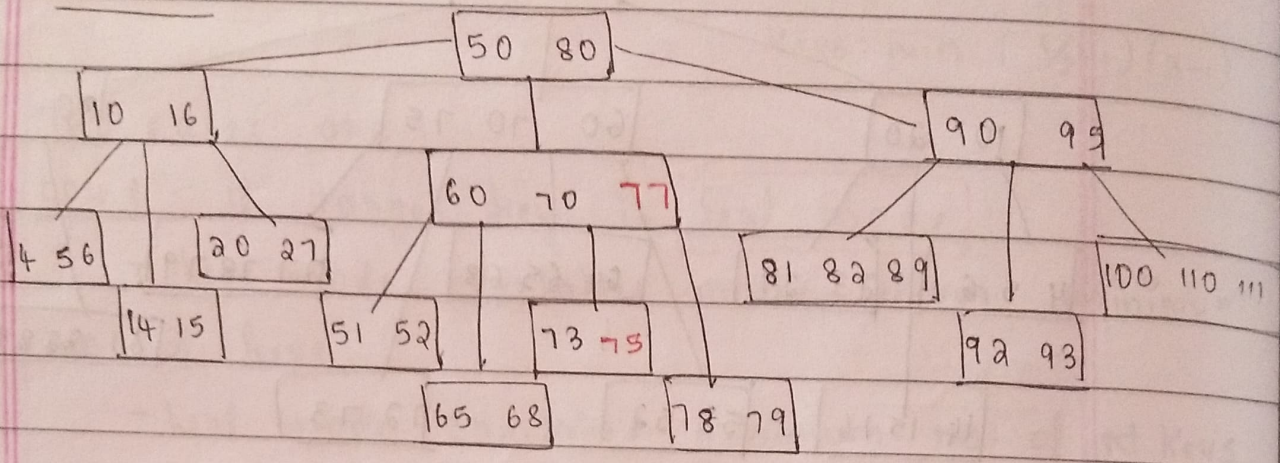


Delete 23

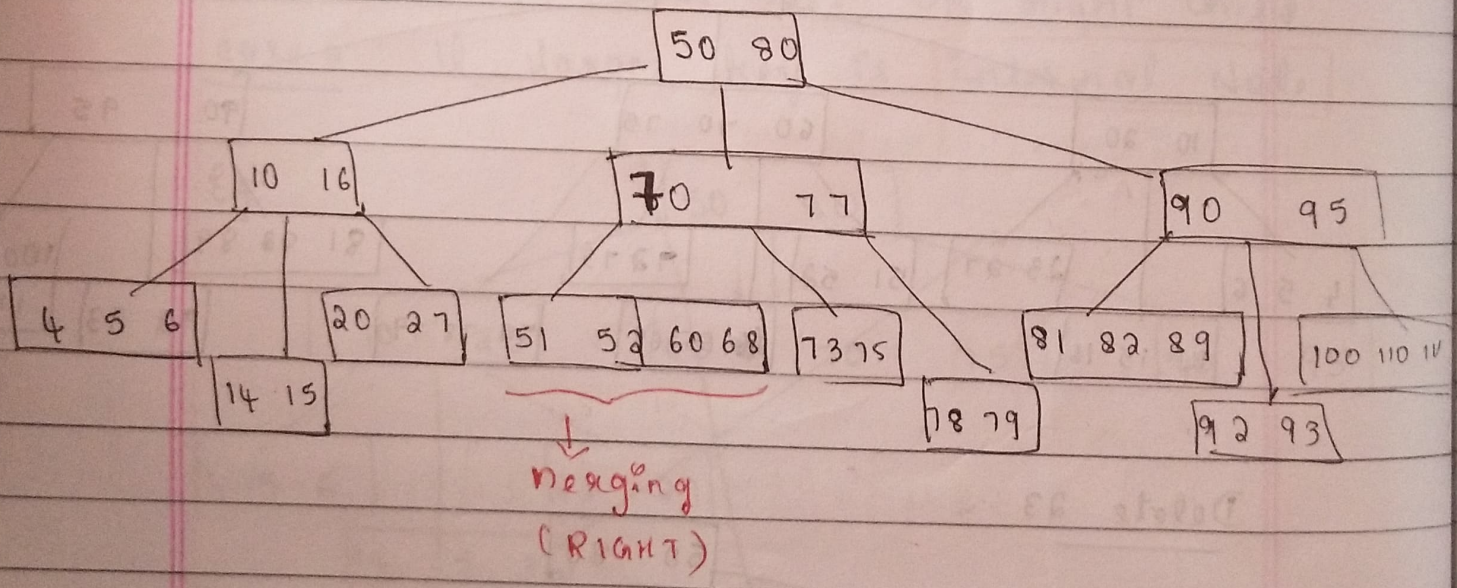


Page No :
Date :

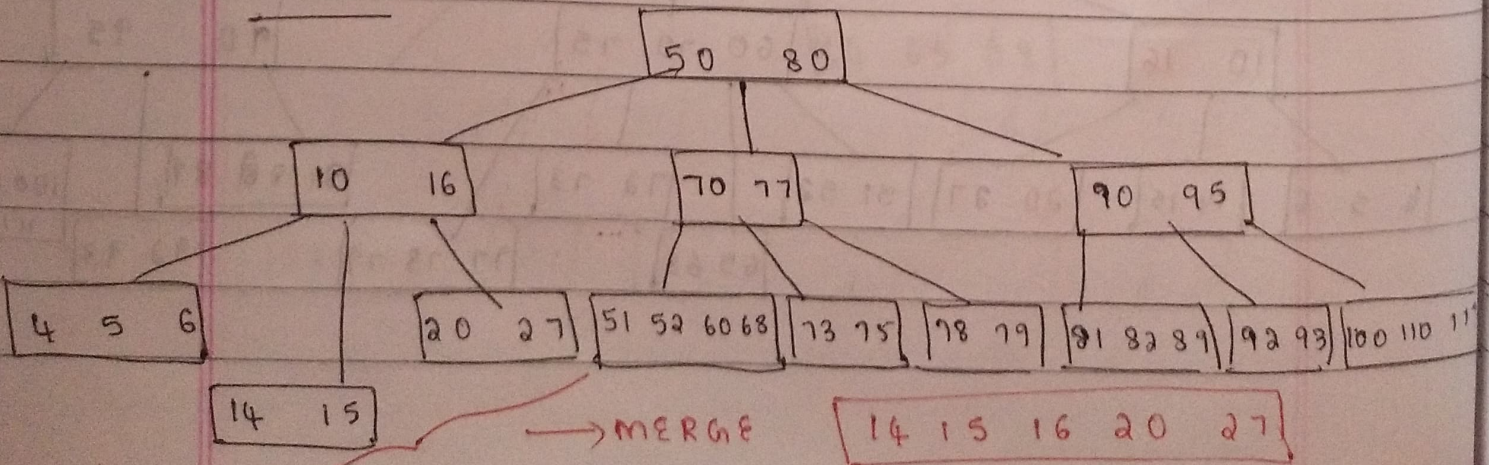
Delete 72

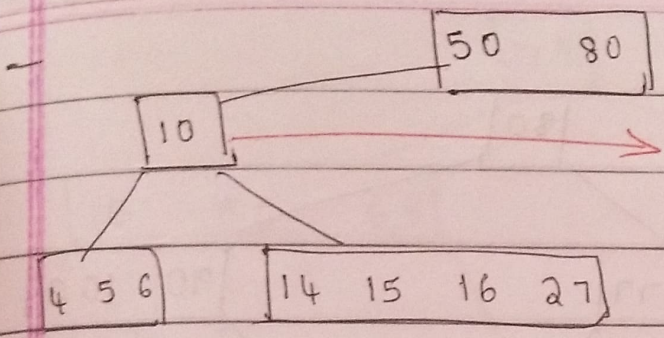


Delete 65



Delete 20

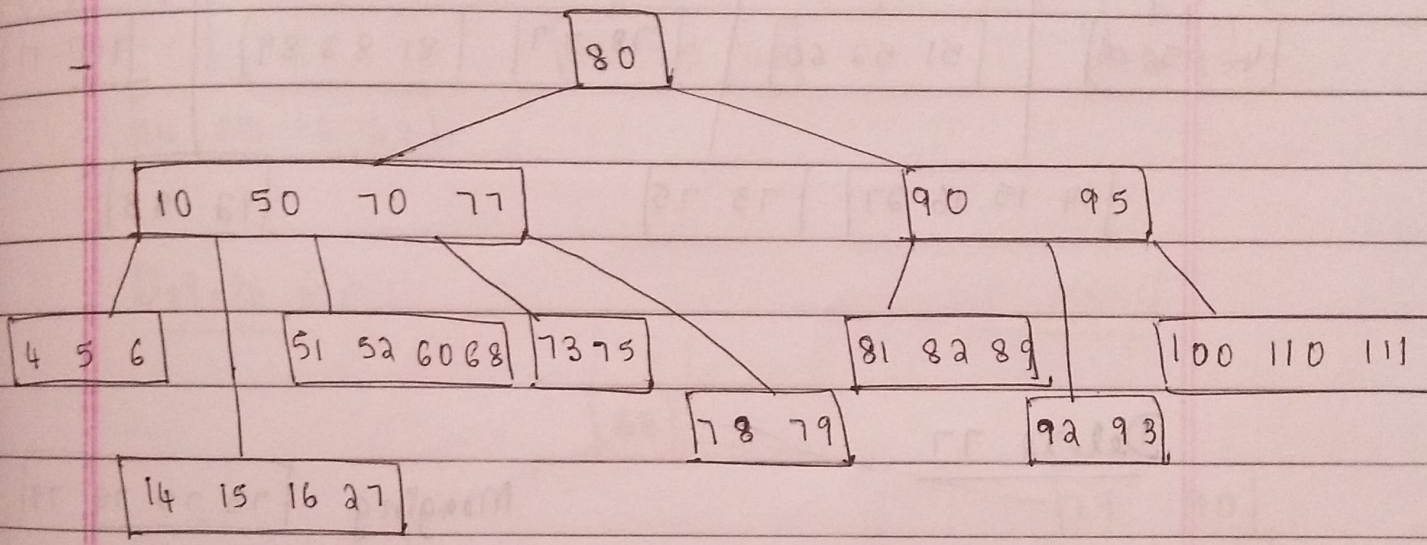




VIOLATES BTREE PROPERTY

MERGE ~~10 50 80~~

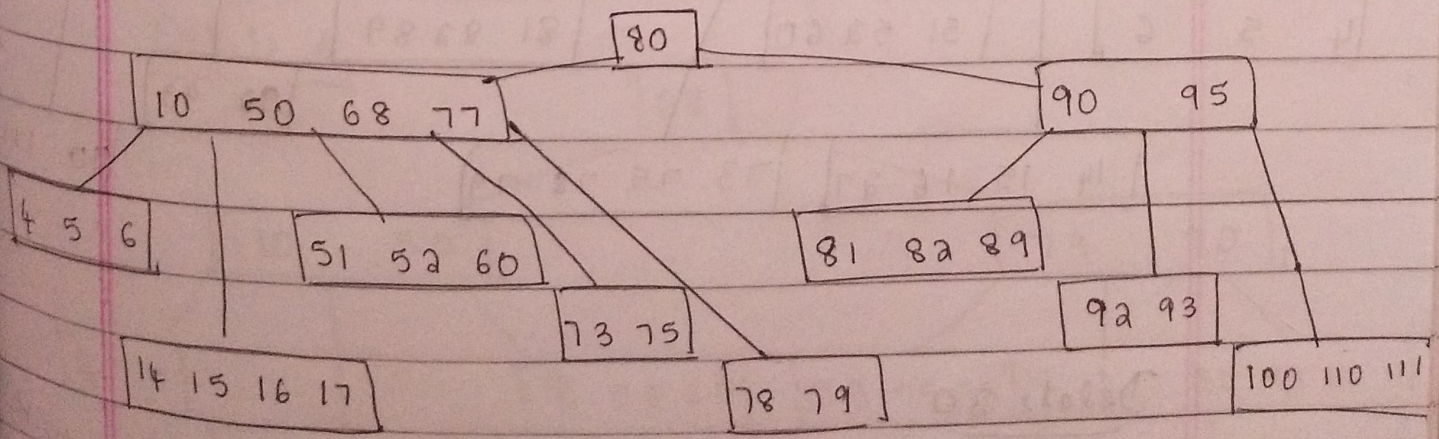
10 50 70 77



Delete 70 (internal Node) - Inorder Predecessor / Successor

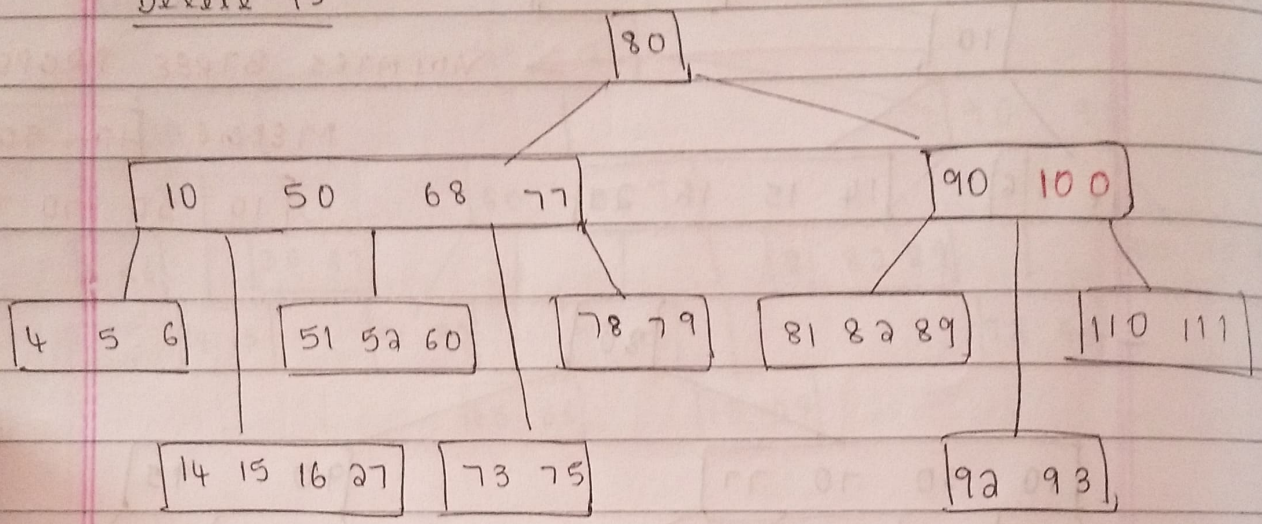
Maximum value of ~~so~~ left SubTree

Minimum value of Right SubTree



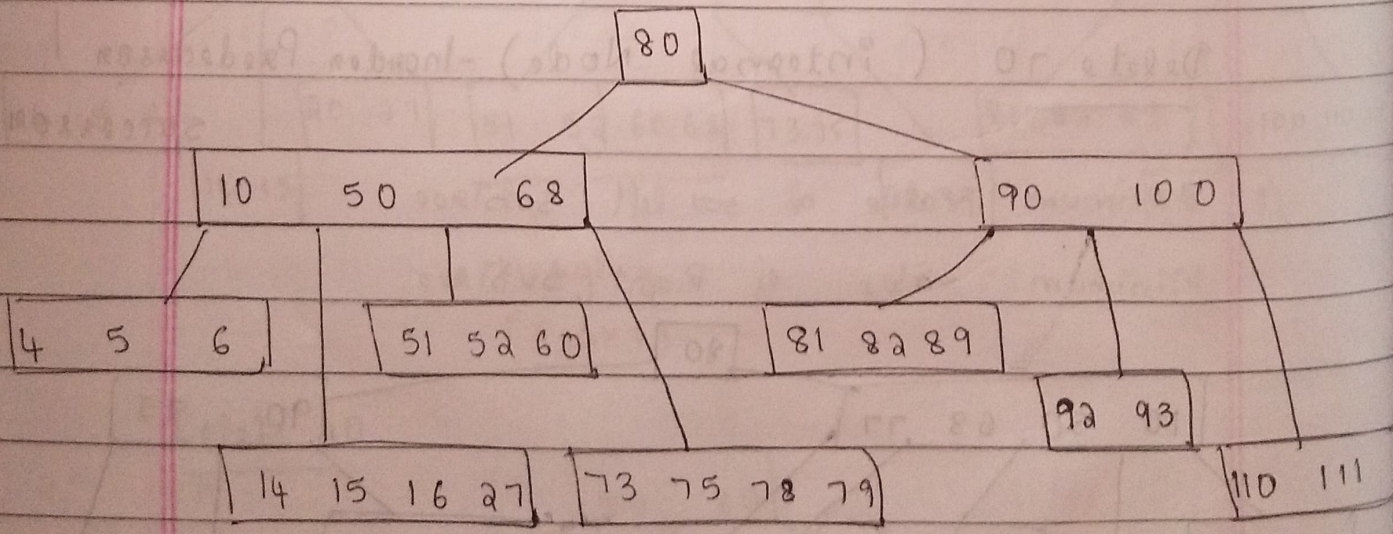
Page No :
Date :

Delete 95



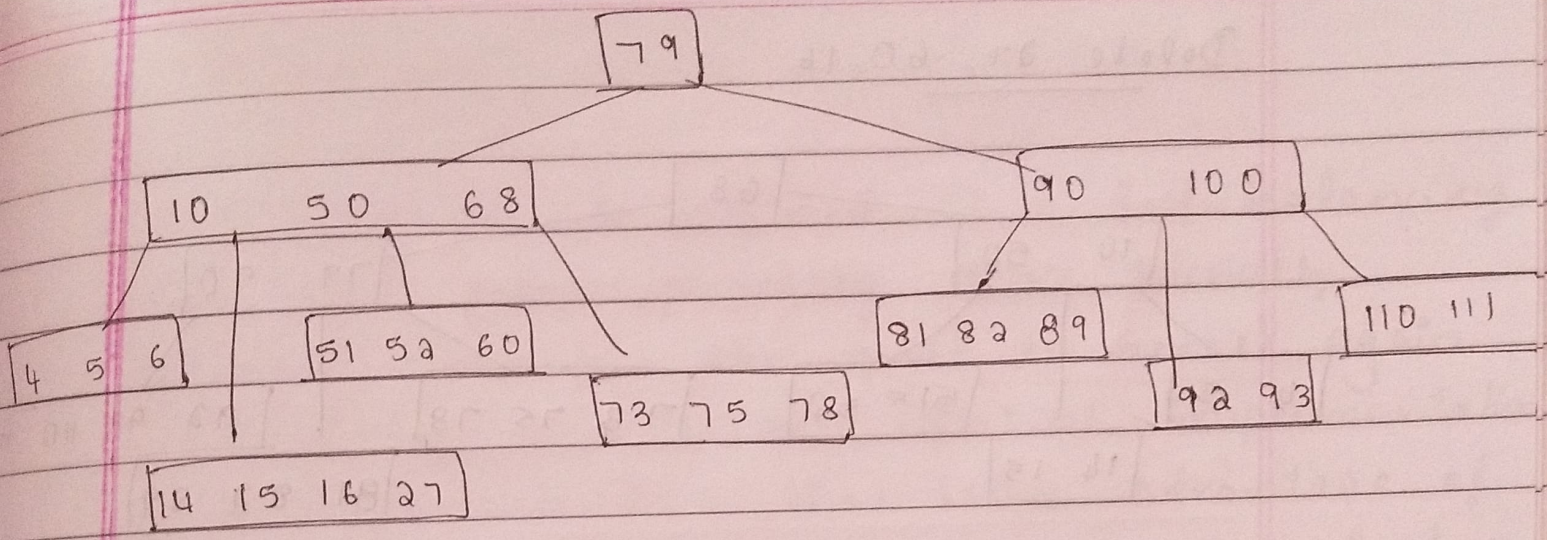
Delete 77

Merging 73 75 78 79

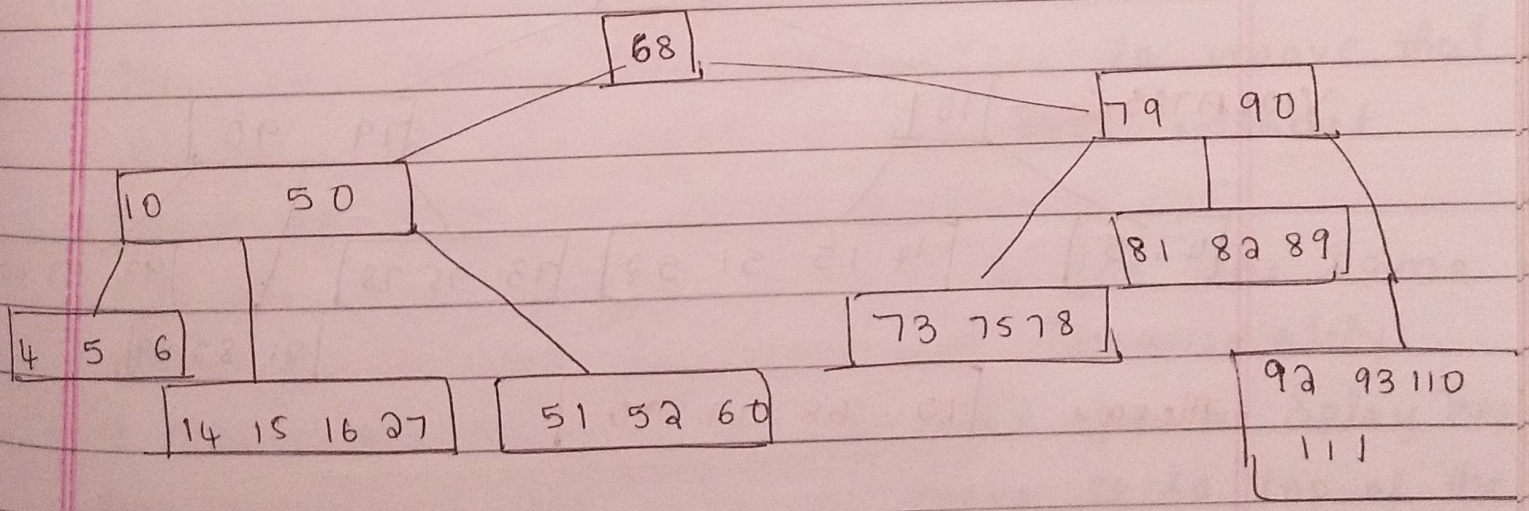


Delete 80

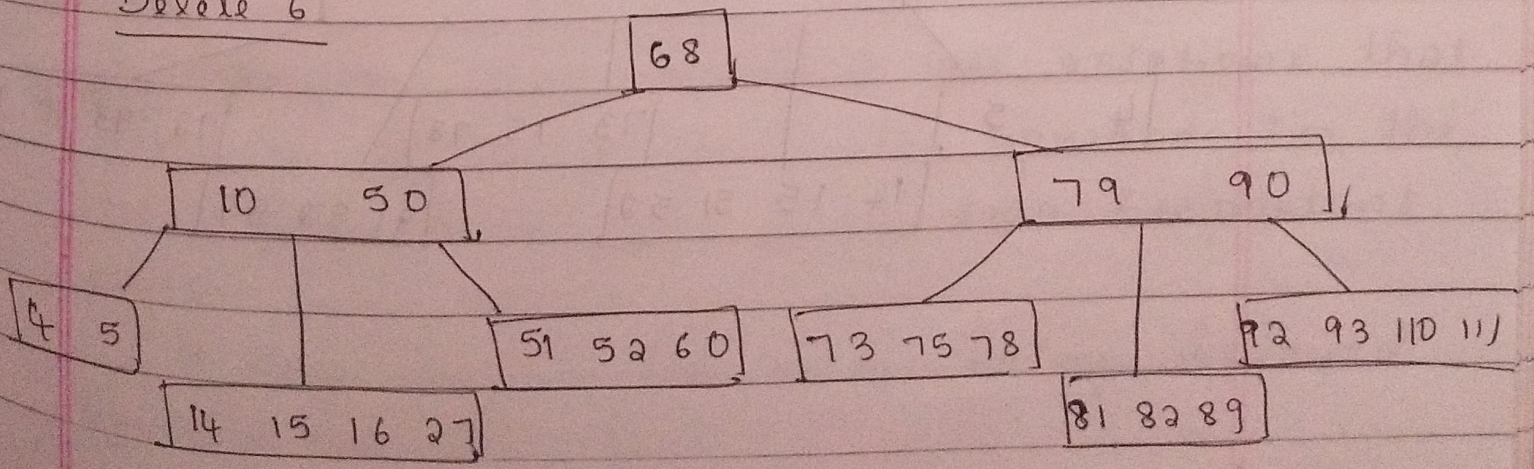
80 is a Root Node. Replace with Inorder predecessor



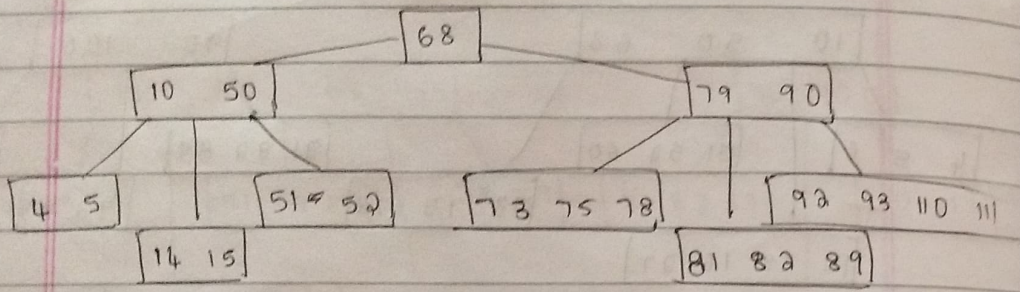
Delete 100



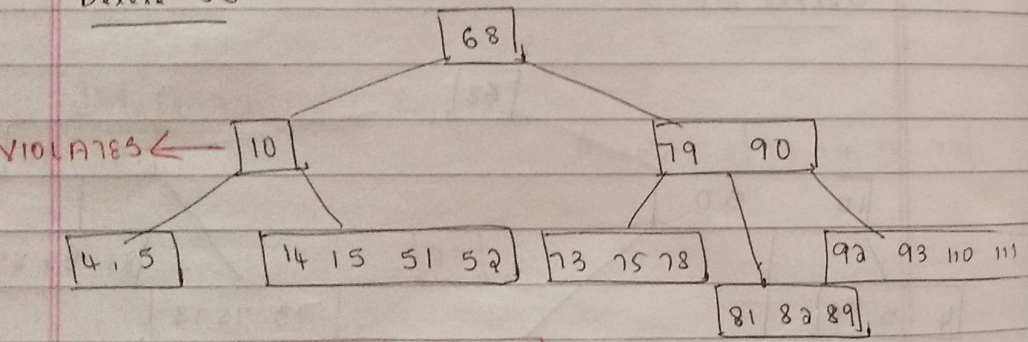
Delete 6



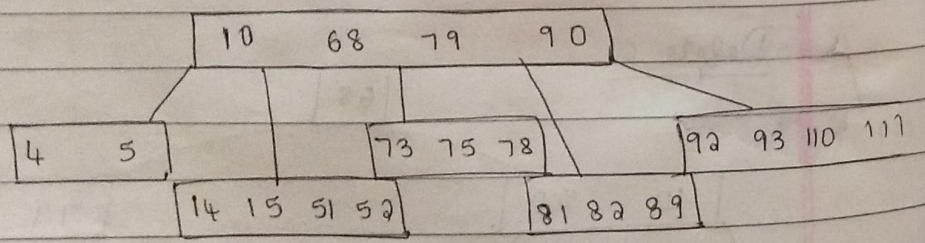
Delete 27, 60, 16



Delete 50



Merge 10 68 79 90



28/11/2021

SPLAY TREE cache

- Splay tree is defined as a self-balancing BST with extra property that recently accessed elements are quick to access again.
- A Splay tree is an efficient implementation of a balanced BST that takes advantage of locality in the keys used in incoming lookup requests.
- Whenever an element is looked up in the tree, the Splay tree recognizes to move that element to the root of the tree, without breaking the BST invariant.
- If the next lookup request is for the same element, it can be returned immediately.
- When an element is accessed in a Splay Tree, tree rotations are used to move it to top of the tree.
- There are three kinds of tree rotations that are used to move elements upward in the tree. These rotations have two important effects.

  * They move the node being Splayed upward in the tree

* They also shorten the path to any nodes along the path to the Sployed Node

• In splay tree, Splaying an element re-arranges all the elements in the tree.

Rotations in Splay Tree

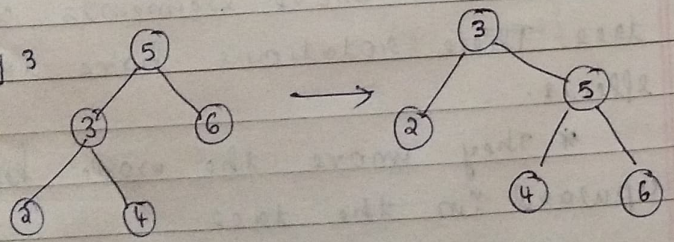
1) Zig Rotation
2) Zig - Zig Rotation
3) Zig - Zag Rotation
4) ZAG Rotation
5) Zig - zag Rotation
6) Zag - Zig Rotation

ZIG ROTATION:

• The Zig Rotation in Splay tree is similar to the Single right rotation in ALT Tree Rotation.

• Here, every node moves one position to the right from its current position.

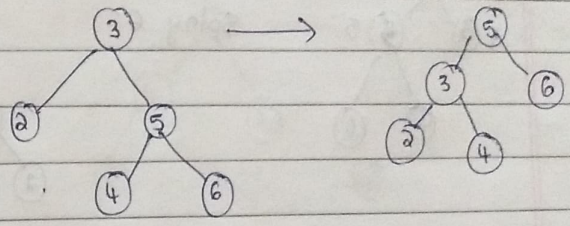
Eg: Splay 3



ZAG ROTATION

• It is similar to Single left Rotation in AVL tree rotations.

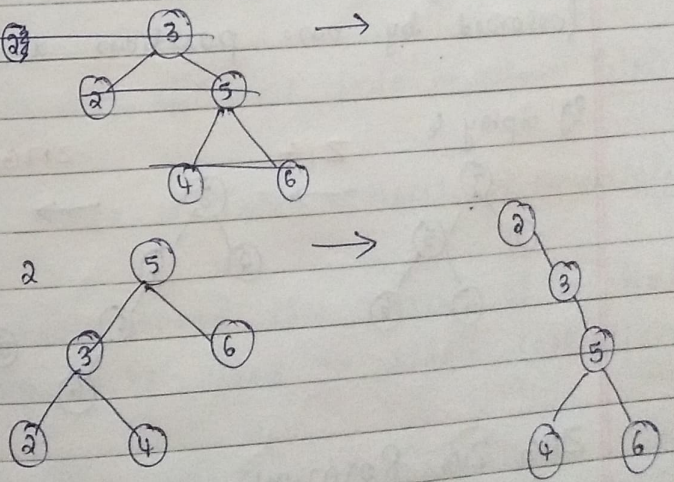
Eg: Splay 5



ZIG ZIG ROTATION

• It is a double Zig Rotation.

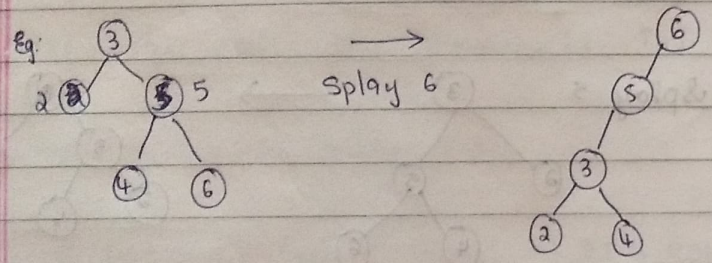
Eg: Splay 2



ZAG ZAG ROTATION

- Double Zag Rotation (double left)

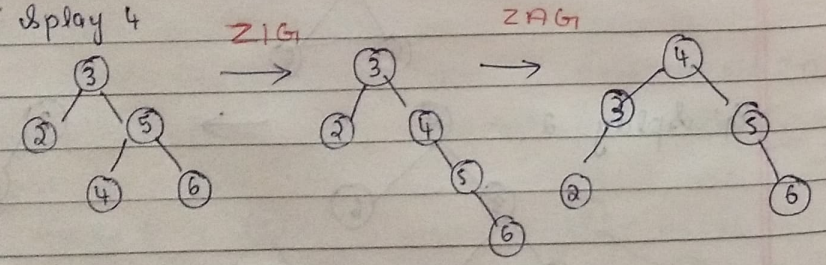
Eg:



ZIG ZAG ROTATION

- Zig followed by Zag Rotation
- every node moves one position to right followed by one position to the left.

Eg: Splay 4

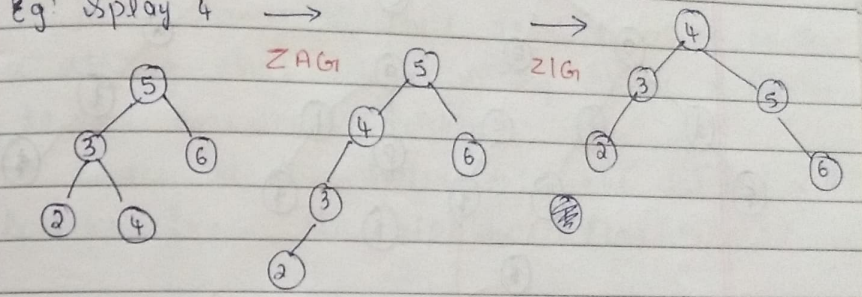


ZAG ZIG ROTATION

- ~~Zag - Zig Rotation~~

- It is a sequence of zag followed by zig.

Eg: Splay 4



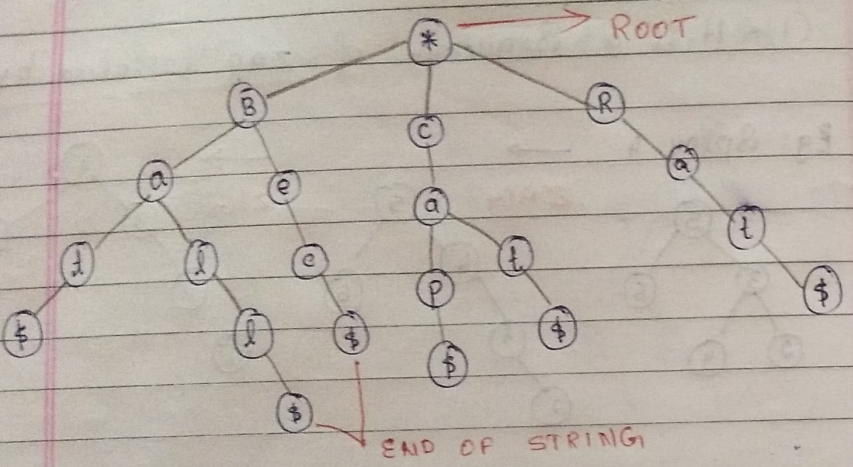
29/11/2021

SUFFIX TREE

Trie

- Trie is an efficient Information Retrieval data structure.
- Every node of Trie consists of multiple branches.
- Its nodes stores the letters of alphabets
- data structure used to store collection of strings.
- Trie is an efficient information storage and Retrieval data structure.

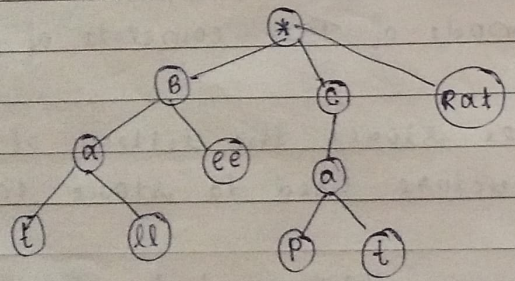
Construct trie using cat, Bat, Ball, Rat, Cap, Bee



Compressed trie :

- To achieve Space Optimisation.
- Trie with nodes of degree at least 2

Eg:



Suffix tree :

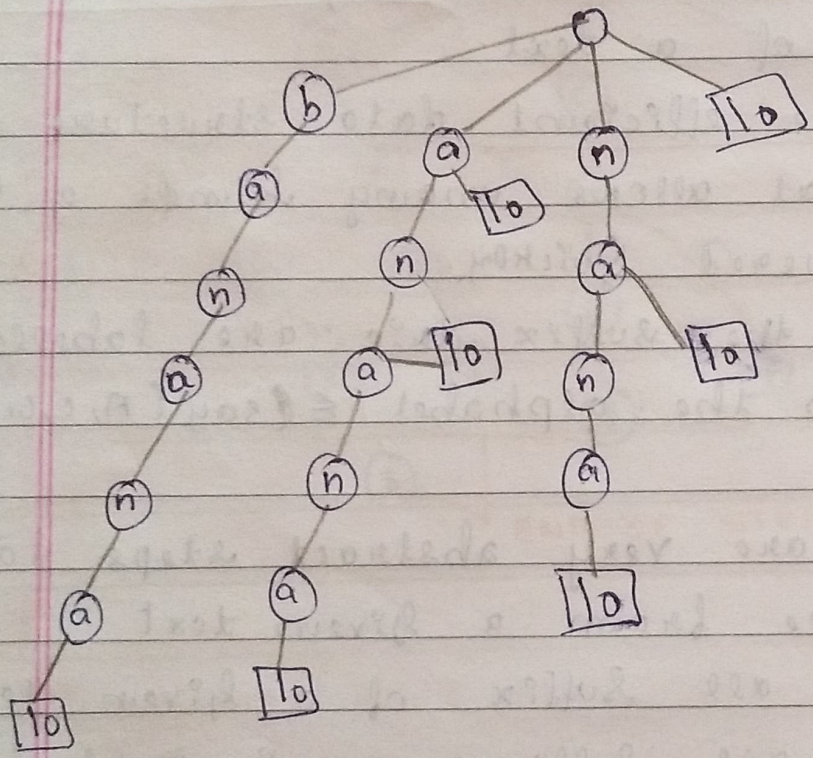
→ A Suffix tree is a compressed trie for all the suffixes of a text.
→ It is Space efficient data structure to store a string that allows many kinds of queries to be answered quickly.
→ Edges of the suffix trie are labelled with letters from the alphabet Σ (say (A, C, G, T)

Following are very abstract steps to build a Suffix tree from a given text
* Generate all Suffix of a given text
* Consider all Suffixes as individual words & build a compressed tree

Eg: banana\0

→ banana\0
→ anana\0
→ nana\0
→ ana\0
→ na\0
→ a\0
→ \0

If we consider all of the above suffixes a individual words & build a trie, then



If we join chains of Single Node, we get the compressed trie, given as