DSA Assignment_4

1) AVL Trees :- (Adelson - Velsky and Landis)

The first type of self-balancing binary search tree to be invented is the AVL tree. The tree AVL tree is coined after its inventor's names - Adelson - Velsky and Landis

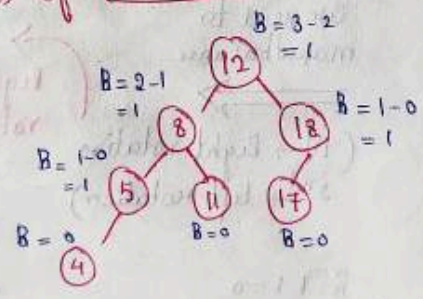
⇒ An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one

1) It must be a binary search tree (BST)
2) |The height of left subtree| - The height of right subtree| = {0, -1, 1}

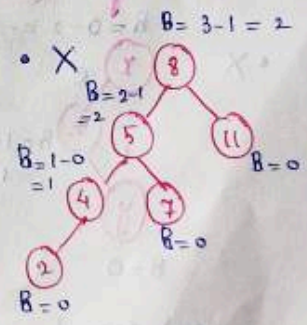
⇒ Balance factor : The difference between the height of left subtree and right subtree for any node is known as balance factor.

Examples of AVL Trees:-

• ✓



• ✗

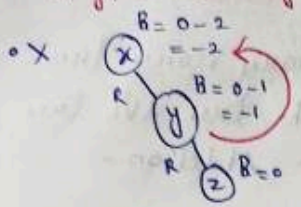


Operations on AVL Tree:-

⇒ Insertion
⇒ Deletion
⇒ Searching
⇒ height

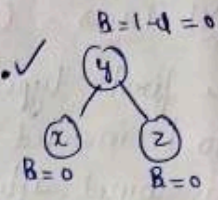
Rotations :-

1) x, y, z (x<y<z) (Right - Right)



• ✗ Unbalanced

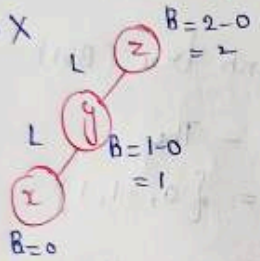
We need to make it balance ⟹ (left rotation)



• ✓ balanced

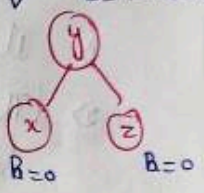
2) Left Left Rotation

z, y, x (x<y<z)



• ✗ Unbalanced

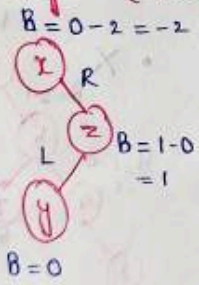
We need to make it balance ⟹ (Right rotation)



• ✓ balanced

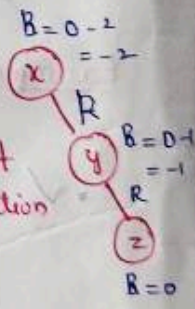
3) Right - Left rotation

x, z, y (x<y<z)

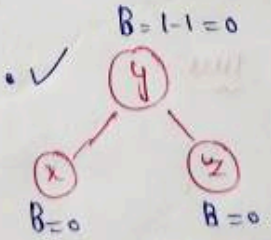


• ✗ Unbalanced

We need to make it balance ⟹ (1st → Right rotation, 2nd → Left rotation)



• ✗ Unbalanced



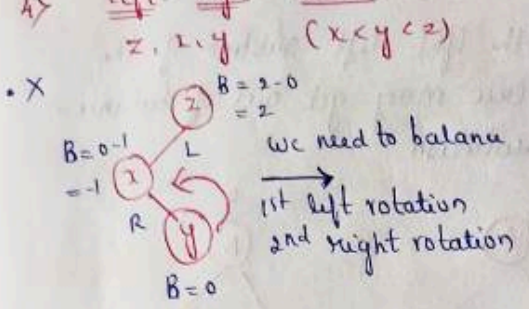
• ✓ balanced

Left rotation

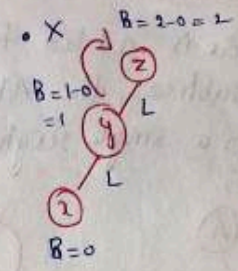
We need to make it balance (As it is a right-right form, we follow left rotation)

4) left-right rotation:

z, x, y (x < y < z)

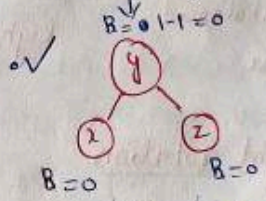


• Unbalanced



• Unbalanced

right rotation



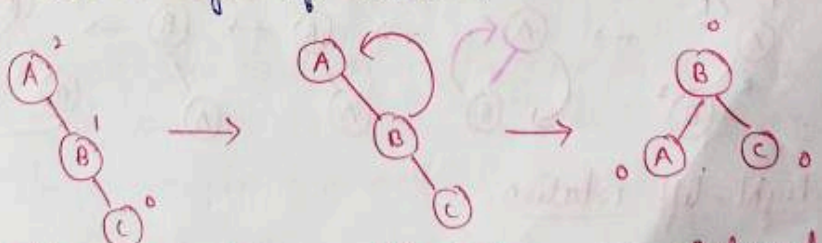
• balanced

Rotating the subtrees in an AVL Tree:

An AVL tree may rotate in one of the following four ways to keep itself balanced.

Left rotation:-

When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we need a single left rotation.

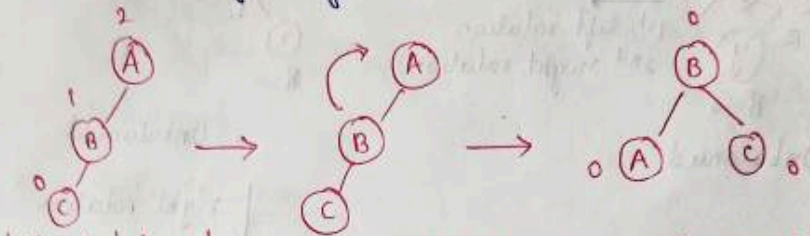


Right Unbalanced tree → Left rotation → Balanced

* Based on the new node insertion we need to take the 3 node (subtree) and then perform rotation operation

Right rotation :

If a node is added to the left left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.

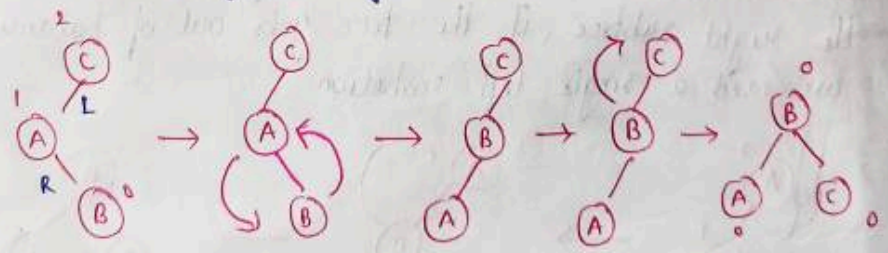


Left - Unbalanced tree → Right Rotation → Balanced

Left - right rotation :-

A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.

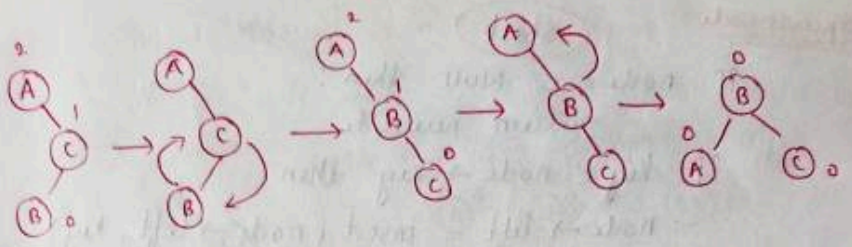
⇒ If a node is added to left subtree of the right subtree, the AVL tree may get out of balanced, firstly we do left rotation & then follow right rotation.



Right - left rotation:

A right-left rotation is a combination in

If a node is added to right subtree of the left subtree, the AVL tree may get out of balanced, firstly we do right rotation & follow left rotation



Operations:-

⇒ Insertion:

The data is inserted into the AVL tree by following the binary search tree property of insertion. i.e left subtree must contain elements less than root value & right subtree must contain greater value than root value and the balance factor ~~must~~ for every node must be either 0, -1, or 1.

Algorithm:

1. Create a node
2. Check if the tree is empty
3. If there the tree is empty, the new node created will become the root node of the AVL tree.
4. If the tree is not empty, we perform the binary search tree insertion operation and check the balancing factor of the node in the tree
5. Suppose the balancing factor exceeds ±1, we apply suitable rotations on said (that) node based on the requirement and resume the insertion from step 4.

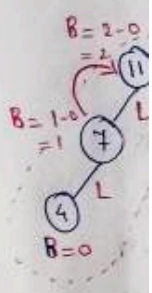
Pseudocode

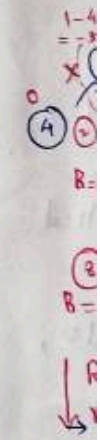
Start

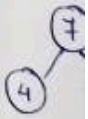
```
if node == Null then:
    return newnode
if key < node → key then:
    node → left = insert (node → left, key)
else if (key > node → key) then:
    node → right = insert (node → right, key)
else
    return node.
node → height = 1+ max( height(noode → left),
    height ( node → right)
    balance = getbalance (node)
if balance > 1 and key < node → left → key then:
    rightrotate
if balance < -1 and key > node → right → key then:
    left rotate
if balance > 1 and key > node → left → key then:
    node → left = leftrotate(node. left)
    rightrotate.
if balance < -1 and key < node → right → key then:
    node → right = right rotate (node → right)
    left rotate (node)
    return node
```

End

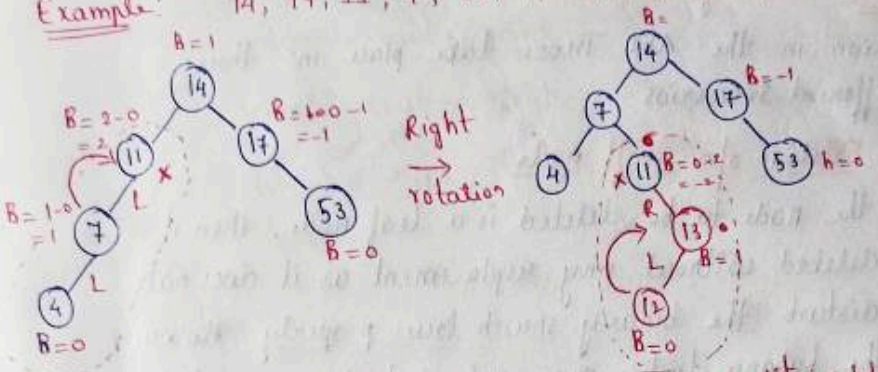
Example



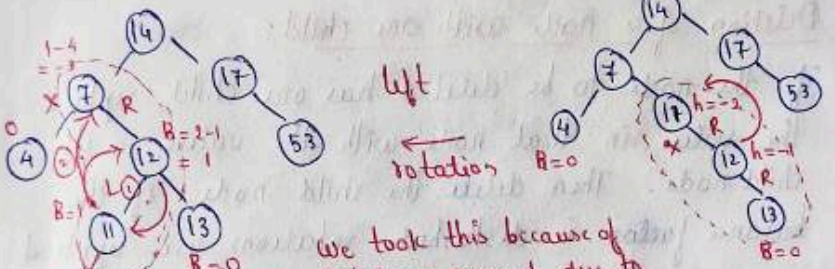


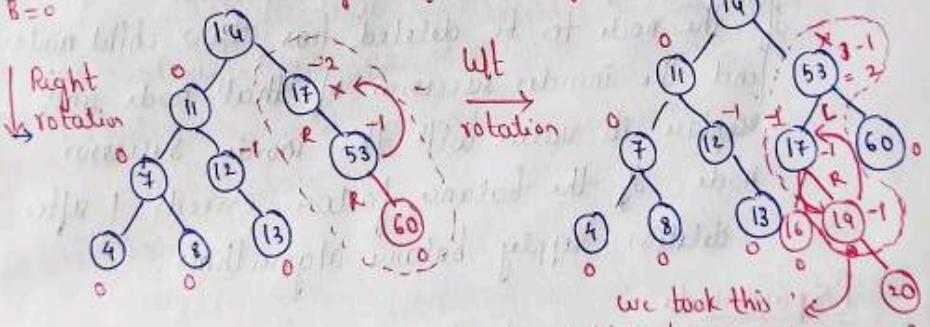


Example: 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20



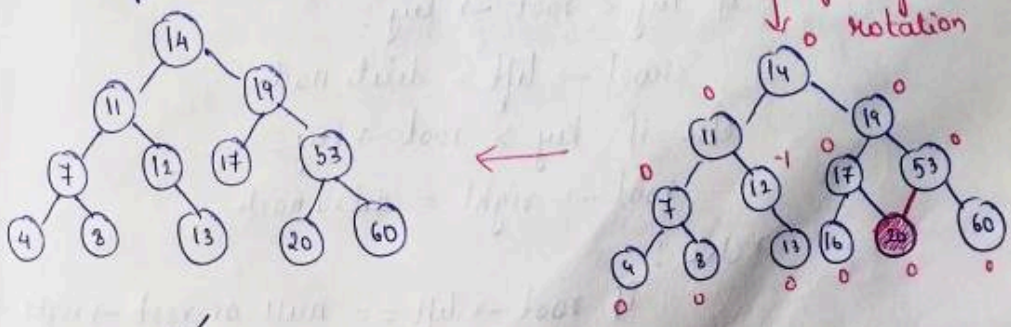
1st right rotation





Left right rotation

∴ Balanced tree after insertion



Deletion : (Algorithm)

Deletion in the AVL Trees take place in three different scenarios -

→ Deletion of a leaf node :-

If the node to be deleted is a leaf node, then it is deleted without any replacement as it does not disturb the binary search tree property. However the balance factor may get disturbed, so rotations are applied to restore it.

→ Deletion of a node with one child:

If the node to be deleted has one child, replace the value in that node with the value in its child node. Then delete the child node. If the balance factor is disturbed, rotations are applied

→ Deletion of a node with two child nodes.

If the node to be deleted has two child nodes, find the inorder successor of that node and replace its value with the inorder successor node. If the balance factor exceeds 1 after deletion apply balance algorithms.

Pseudocode :

start

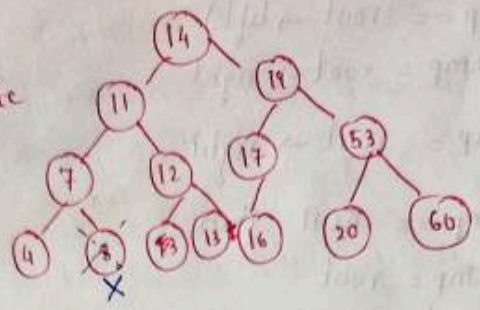
```
if root == NULL:
    return root
if key < root → key:
    root → left = delete node
else if key > root → key:
    root → right = delete node
else:
    if root → left == null or root → right = null;
    node temp = null
```

```
if ( temp == root → left)
        temp = root → right
else temp = root → right
if if temp == null then
        temp = root
        root = null
    else
        root = temp
    el
else :
    temp = minimum valued node
    root → key = temp → key
    root → right = delete node
If ( root == null ) then :
    return root
    root → height = (max ( height (root → left) ,
                           height (root → right)) + 1 )
    balance = getbalance
if (balance > 1 and getbalance (root → left ) >= 0)
    right rotate
if (balance > 1 and getbalance (root → left) >= 0)
    root → left = left rotate (root → left) :
    right rotate
if ( balance < -1 and getbalance (root → right) <= 0)
    left rotate
if (balance < -1 and getbalance (root → right) > 0)
    root → right = right rotate (root → right)
    left rotate
    return root
```

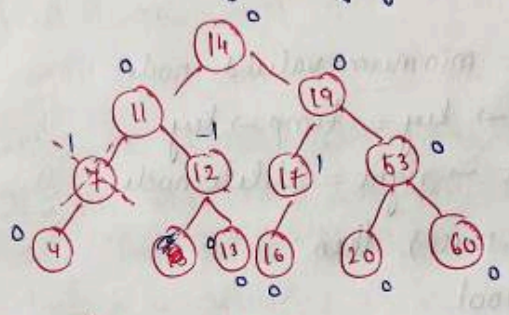
End

Example:

We need to delete
8, 7, 11, 14, 17

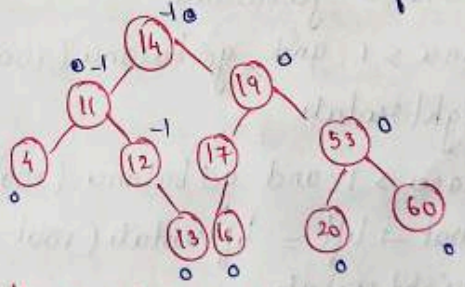


① 8 → leaf node (direct deletion)
and then for balancing factor. if unbalanced, balance it by applying proper rotations

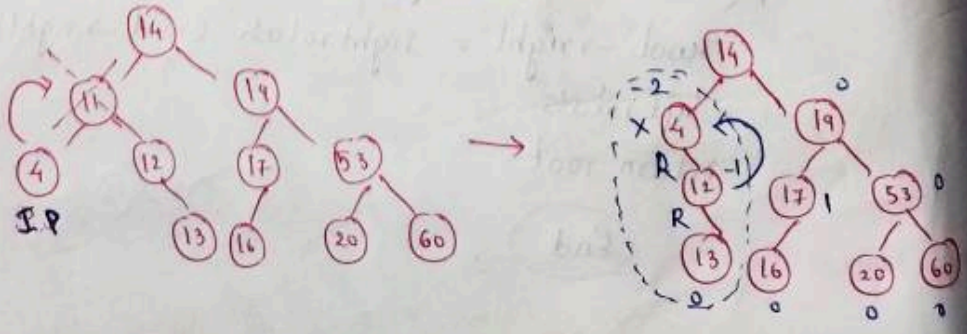


balanced

② 7 → (node with one child)
→ replace that deleted node with its child and then delete child node and check for balancing.

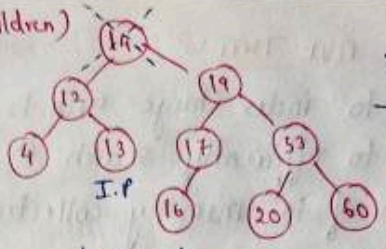


③ 11 → (node with 2 children)
→ replace that node (want to be deleted) with its inorder successor or inorder predecessor. and check for balanced condition.



④ 14 → (node with 2 children)

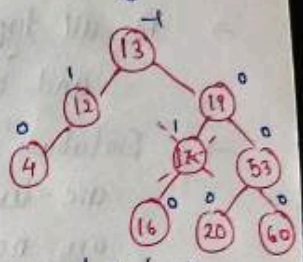
Left rotation



I.P

balanced

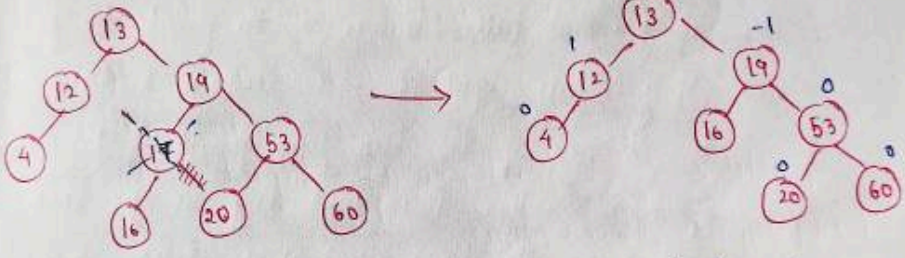
replacing 14 with inorder predecessor



· balanced

5) 17 → (node with one child)

→ replace that node with its childnode and delete childnode and balance it.



· balanced

Height of AVL tree:

→ If there are n nodes in AVL tree, minimum height of AVL tree is $\text{floor}(\log_2 n)$.

→ If there are n nodes in AVL tree, maximum height can't exceed $1.44 * \log_2 n$.

→ If height of AVL tree is h, maximum no. of nodes can be $2^{h+1} - 1$,

→ The minimum no. of nodes in a tree with height h can be represented as : $N(h) = N(h-1) + N(h-2) + 1$ for $n > 2$ where $N(0) = 1$ & $N(1) = 2$.

0

60

0

Applications of AVL Tree :-

→ It is used to index huge records in a data base and also to efficiently search in that

→ For all types of in-memory collections, including sets and dictionaries, AVL trees are used

→ Database applications, where insertions and deletions are less common but frequent data lookups are necessary.

→ software that needs optimized search

→ It is applied in corporate areas and storyline games.

B - Tree (Balanced - Tree)

Properties:

→ balanced (m) way tree (m: order)

→ Generalization of BST in which a node can have more than one key and more than 2 children

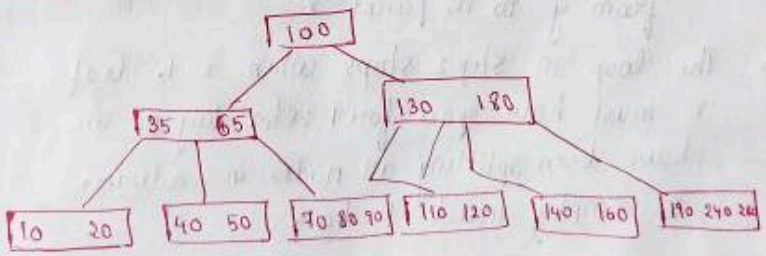
→ maintain sorted data.

→ 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 node = $\lceil \frac{m}{2} \rceil$
- every node has max. (m-1) keys
- Min. keys: rootnode → 1
  all other nodes → $\lceil \frac{m}{2} \rceil - 1$

Example:-



B-tree of order 3

Operations:

→ Insertion

→ Deletion

→ Searching

1) Insertion:

In B-Tree, A new key or node is always inserted at the leaf node. Let the key to be inserted be k, like BST, we start from the root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node.

Algorithm:

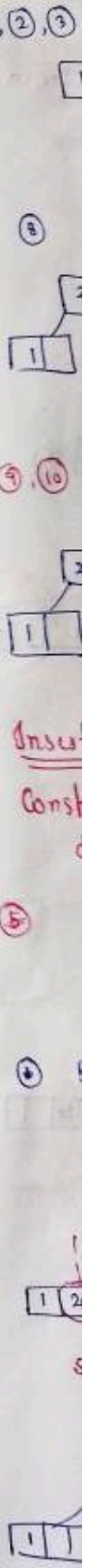
1) Initialize x as root
2) While x is not leaf, do following
   a) Find the child of x that is going to be traversed next. Let the child be y.
   b) If y is not full, change x to point to y
   c) If y is full, split it and change x to point to ~~of~~ one of the ~~two~~ two parts of y.
      If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.

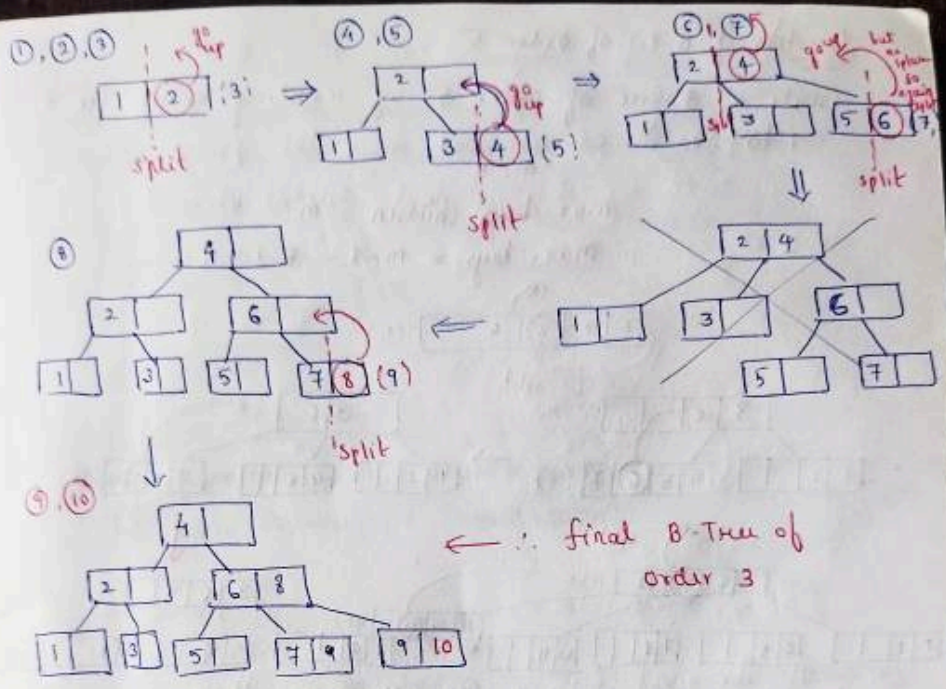
Eg: Insertion in B-Tree of Order 3

$m = 3$

Max key $= (m-1) = 3-1 = 2$

Create a B-Tree of order 3 by inserting values from 1 to 10.





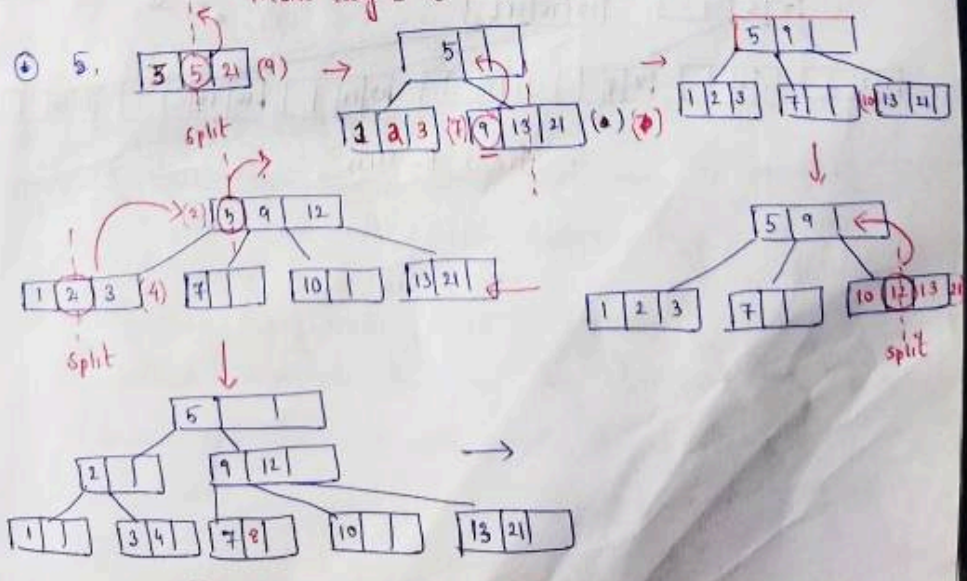
Insertion of B-Tree of order 4

Construct a B-Tree of order 4 with following set of data 5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

⑤ order = 4

m = 4

max key = (m-1) = 4-1 = 3



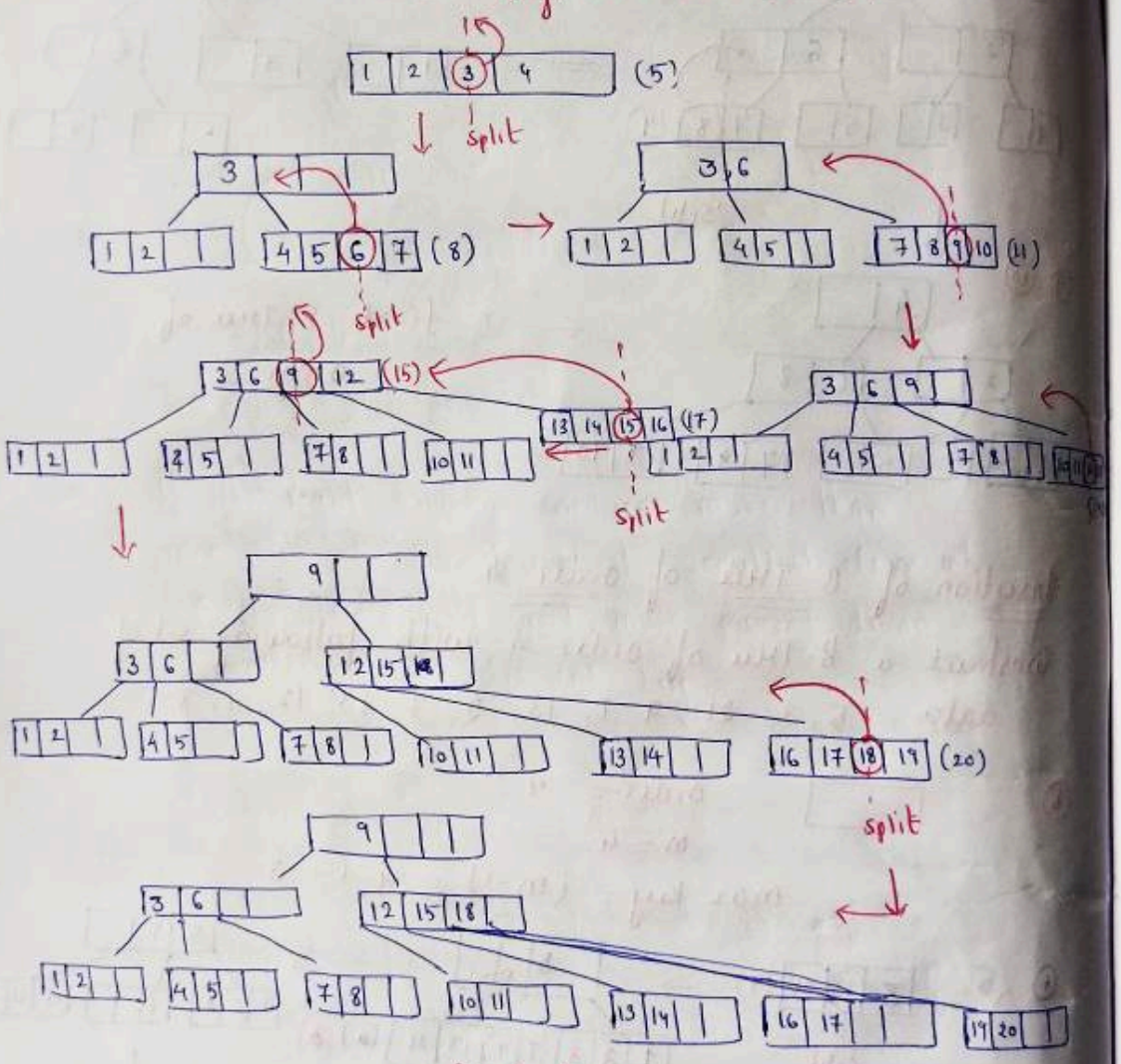
Insertion of B-T of Order 5

Create a B-tree of order 5 by inserting values from 1 to 20.

$m = 5$

max ~~keys~~ children $= m = 5$

max keys $= m - 1 = 5 - 1 = 4$



• final B+ Tree

Deletion

Deletion is can be performed in B-Tree. The node which is to be deleted can either be a leaf node or internal node.

i) leaf node —
- that leaf node contains more than min no. of keys
- that leaf node contain min no. of keys
  - borrow from immediate left node (sibling)
  - borrow from immediate right node
  - neither borrowing from left or right siblings

ii) in internal node
- inorder predecessor
- inorder successor
- neither inorder predecessor or successor.

Deletion of order 5 (m = 5)

min children = $\lceil \frac{m}{2} \rceil = \lceil \frac{5}{2} \rceil = 3$

max children = 5

min keys = $\lceil \frac{m}{2} \rceil - 1 = \lceil \frac{5}{2} \rceil - 1 = 3 - 1 = 2$

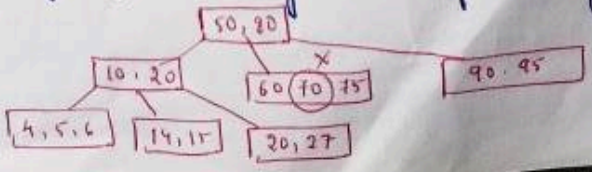
max keys = 4

Algorithm -

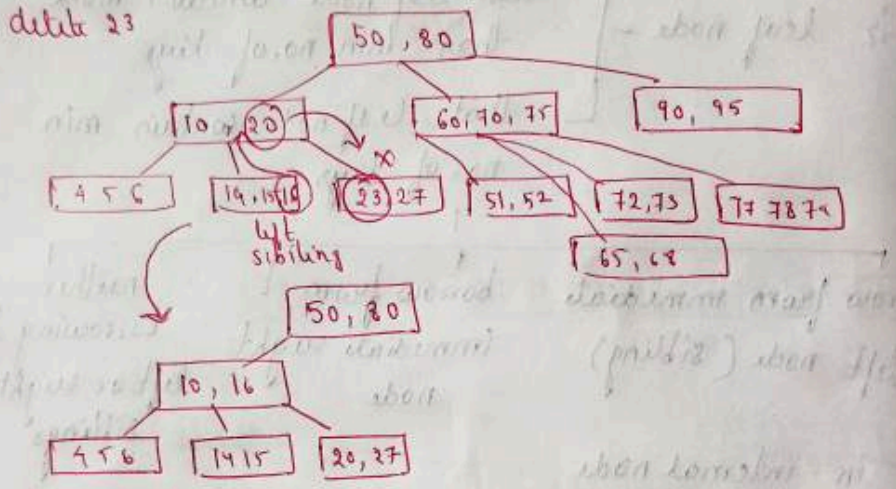
→ locate the leaf node

→ If there are more than min keys in the leaf node then delete the desired key from the node

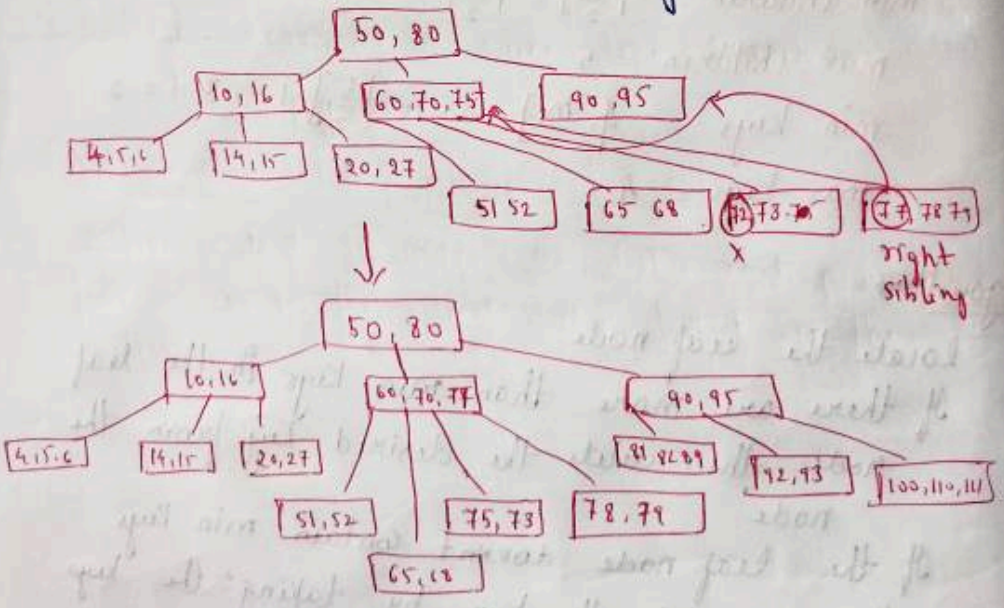
→ If the leaf node doesn't contain min keys then complete the keys by taking the keys (borrowing) from a right or left sibling



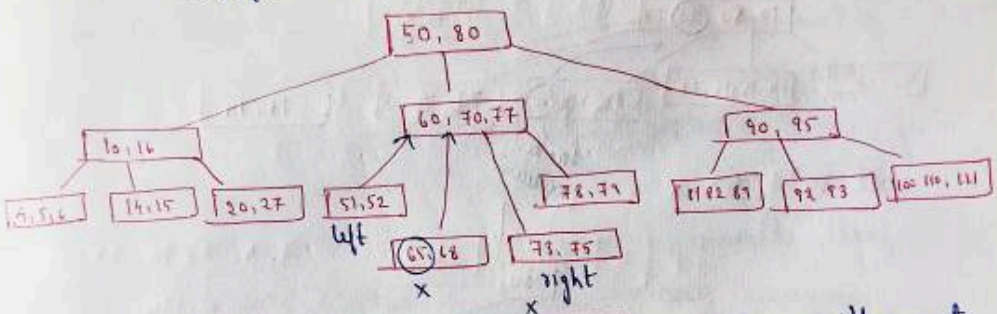
→ If the left sibling contains more than min keys then push its largest element up to its parent and move the parent down by deleting that desired element

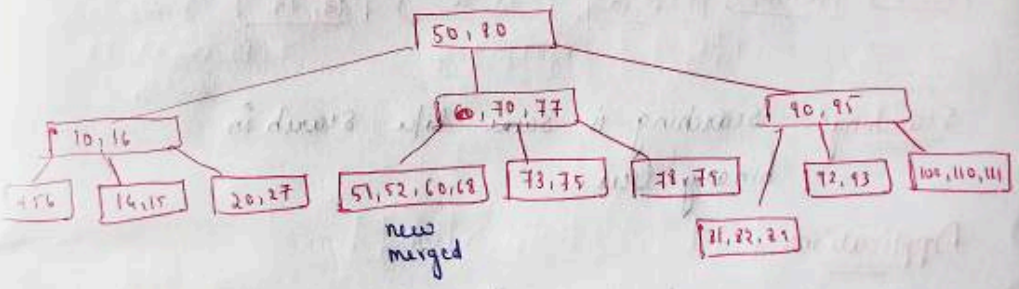


→ If the right sibling contains more than min elements then push its smallest element up to the parent and move parent down by deleting the desired key

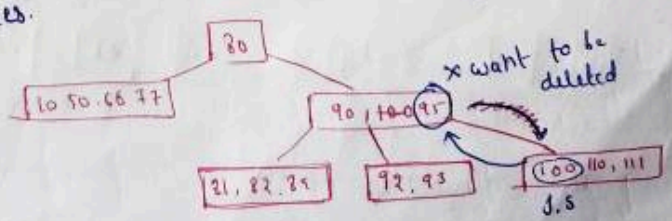


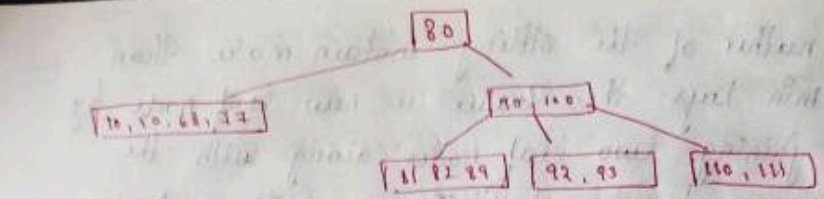
→ If neither of the sibling contain more than min keys then create a new leaf node by joining two leaf nodes along with the parent element. (The node with delete key and with either left or right sibling) along with the parent node between those two merged nodes.



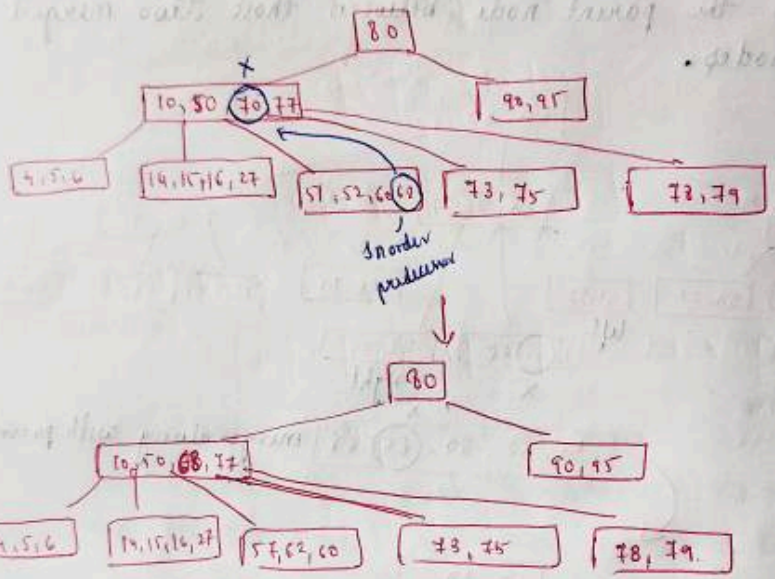


→ If the node which is to be deleted is an internal node, then replace the node with its inorder successor or predecessor by following the previous rules.





in Inorder predecessor



Searching:- Searching is same like search in binary tree.

Applications:-

→ B tree is used to index the data and provide fast access to the actual data stored on the disks.

→

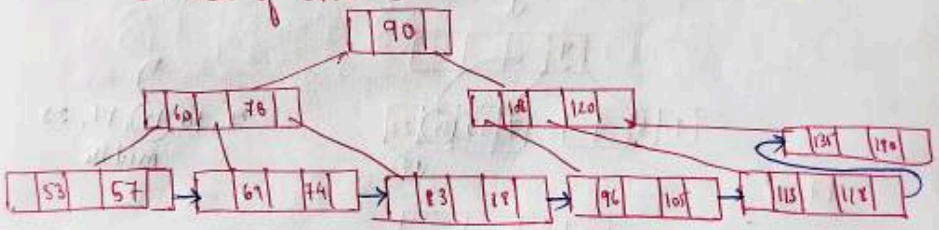
:- B+ Tree :-

B+ Tree is an extention of B Tree which allows efficient insertion, deletion and search operations.

⇒ The leaf nodes of a B+ Tree are linked together in the form of a singly linked list to make the search queries more efficient

⇒ B+ tree are used to store the large amount of data which can't be stored in the main memory. due to that fact, size of main memory is always limited, the internal nodes of B+ Tree are stored in the main memory, whereas, leaf nods are stored in the secondary memory

B+ tree of order '3'



Operations:

i) Insertion:

⇒ Insert the new node as a leaf node

→ If the leaf doesn't have required space, split the node and copy the middle node to its parent node

→ If the leaf node doesn't have required space after inserting then find the middle element then & slipt it by sending the middle element up. Slipting of nodes is done in such a way that middle element should also present in leaf node

Eg:- 7, 10, 1, 23, 5, 15, 17, 9, 11, 39, 35, 8, 40, 25

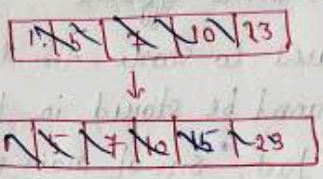
max B+ tree of order 5

max children = 5

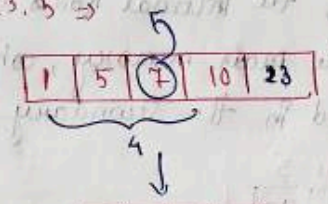
min children = $\lceil \frac{5}{2} \rceil = 3$

max keys = 4

min keys = 2

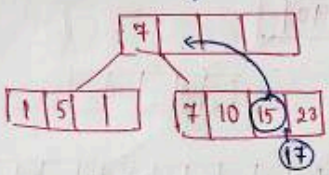


7, 10, 1, 23, 5 ⇒



1, 5, 7, 10, 23

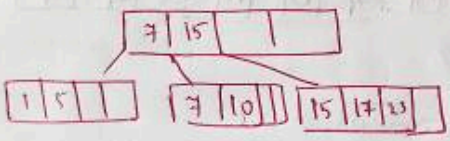
↓ middle



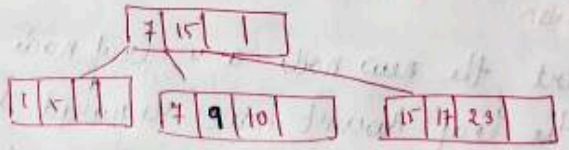
7, 10, 15, 17, 23

middle

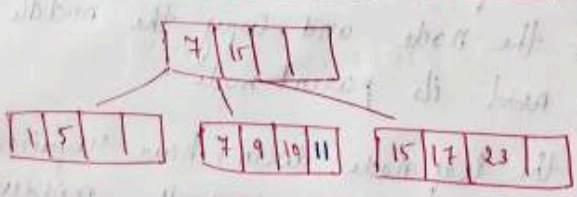
17 ⇒



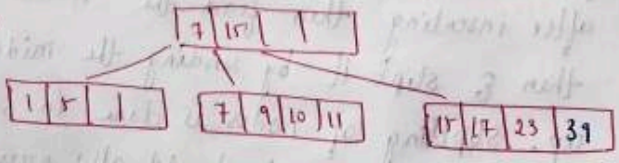
9 ⇒



11 ⇒



39 ⇒



35 ⇒

8 ⇒

40 ⇒

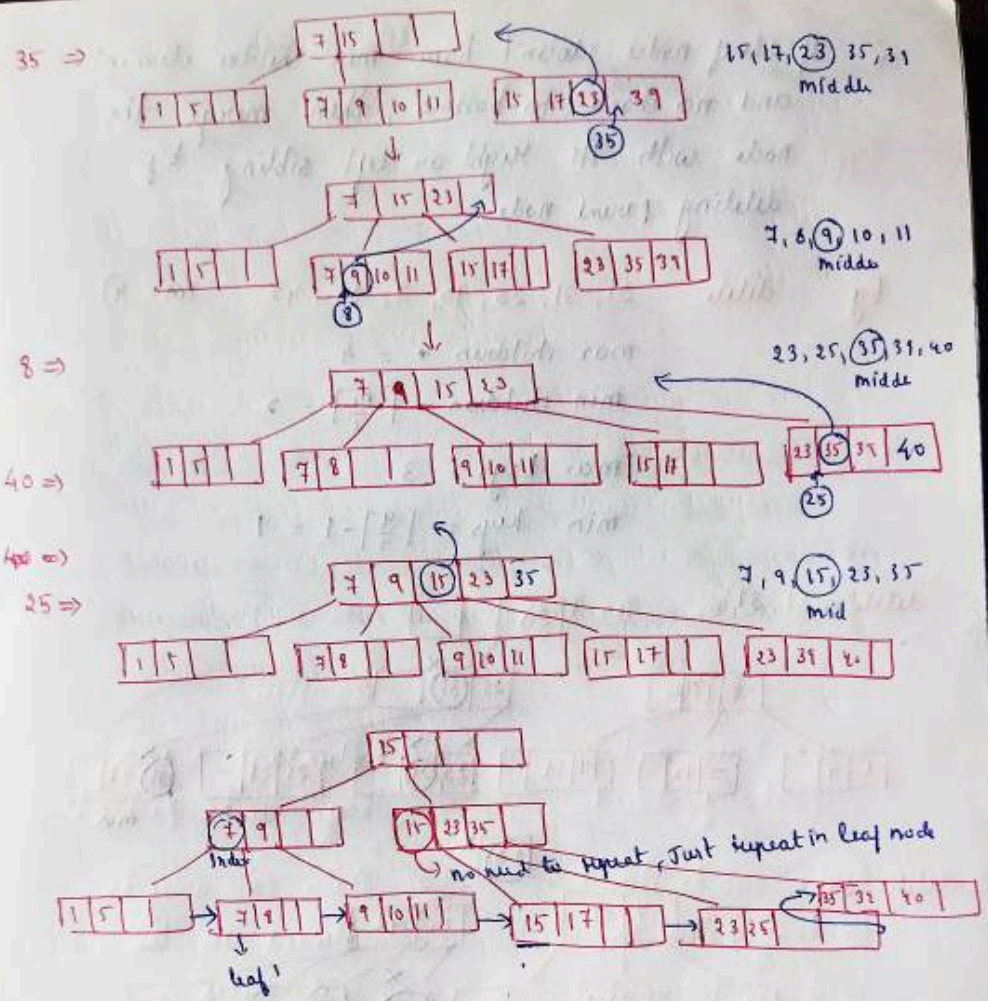
25 ⇒

Deletion

⇒ Dele

⇒

⇒



Deletion

⇒ Delete the key (data) from leaf node as well

⇒ as from where it already present (index)

⇒ If the leaf node contain less than minimum no. of elements (keys) ~~merge~~ borrow the node with it right or left siblings by deleting the parent node.

from right sibling → borrowed element must be minimum

from left sibling → borrowed element must be maximum

⇒ If leaf nodes doesn't have min children element and no chance to borrow, then merge the node with its right or left sibling by deleting parent node

Eg: delete 21, 31, 20, 10, 7, 25, 42 (m = 4)

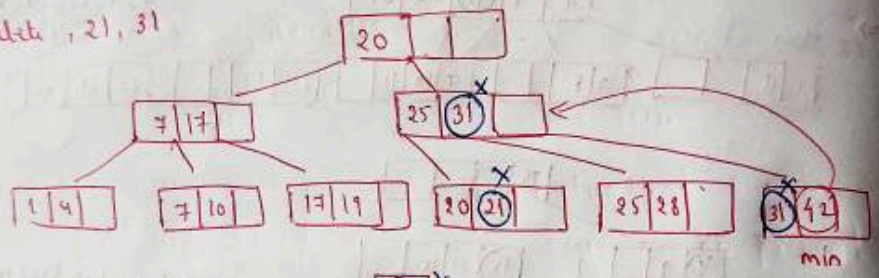
max children = 4

min children = $\lceil \frac{m}{2} \rceil = 2$

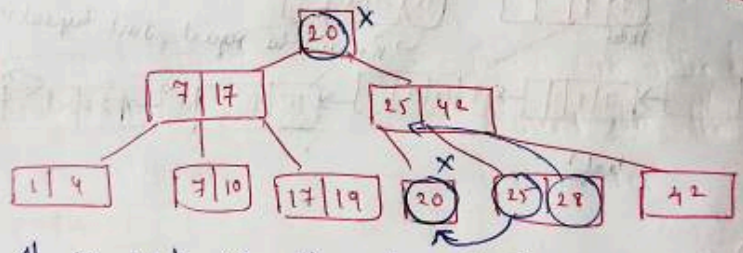
max keys = 3

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

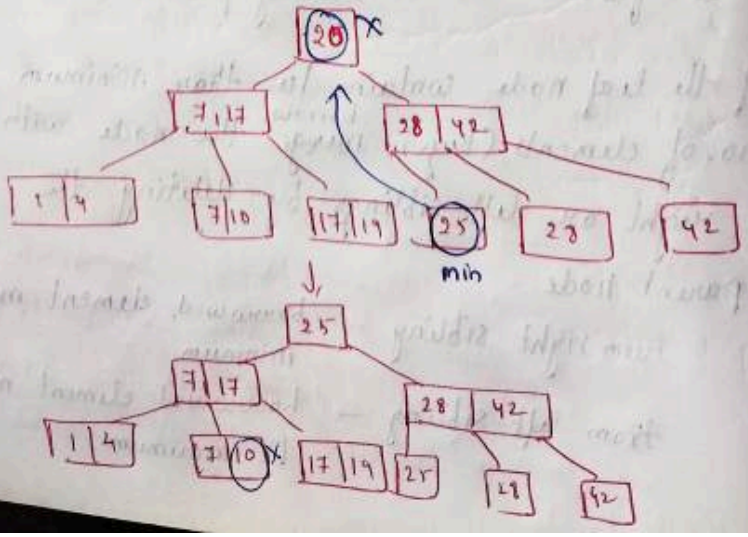
delete : 21, 31



delete : 20

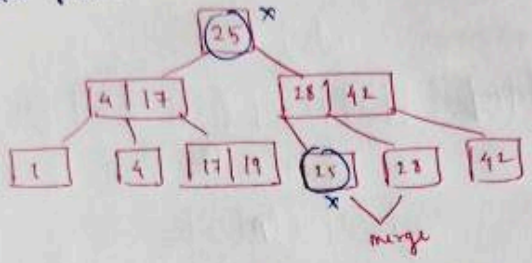


→ If we delete 20 then it will be empty, therefore borrowing takes takes.



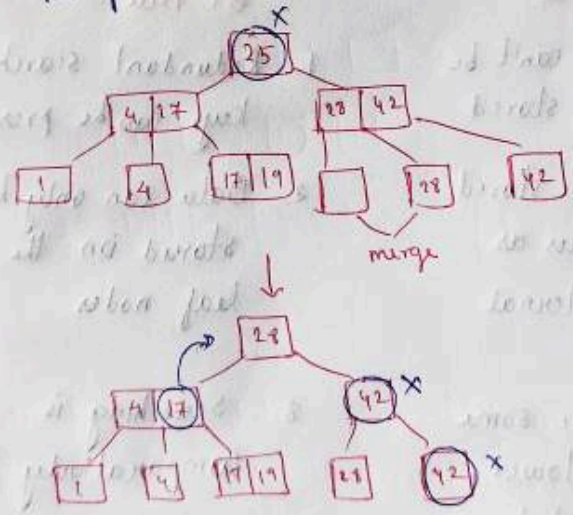
delete 10: 7

If we delete '7', it becomes empty, therefore borrowing takes place

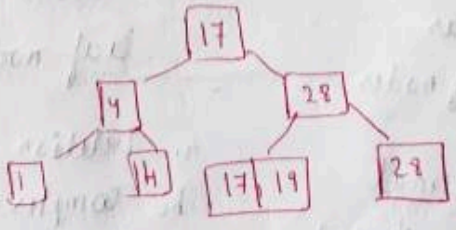


delete 25.

If we delete '25', it becomes empty (ϵ > min keys) then borrowing should takes place, but still borrowing is not possible as it violets B+Tree property. (i.e min keys = 1). Therefore merging takes place



delete 42:- If we delete '42' the same issue occur.



Searching: Searching is same like as searching in Binary tree.

Applications :-

B Tree Vs B+ Tree

	BTree	B+ Tree
1.	Search Keys can't be repeatedly stored	Redundant search keys can be present
2.	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes
3.	Searching for some data is slower process since data can be found on internal nodes as well as on leaf nodes	Searching is comparatively faster as data can only be found on the leaf node
4.	Deletion of internal nodes are so complicated and time consuming	Deletion will never be complexed process since element will always be deleted from the leaf node

5. leaf
be

5. leaf nodes can't be linked together	5. leaf nodes are linked together to make the search operation more efficient.