B Trees :-

Disk Structure
How data is stored in disk
What is indexing
What is multilevel Indexing
M-way Search Trees
B-Trees
Insertion, Deletion in B-Trees
B+ Trees!

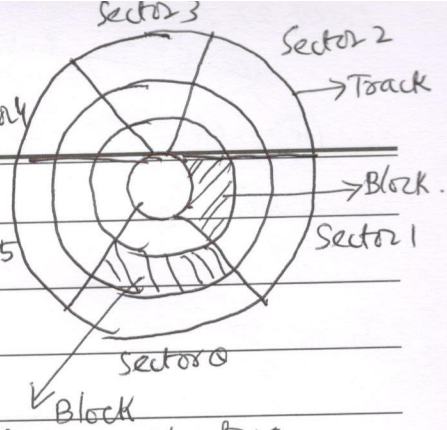
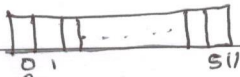


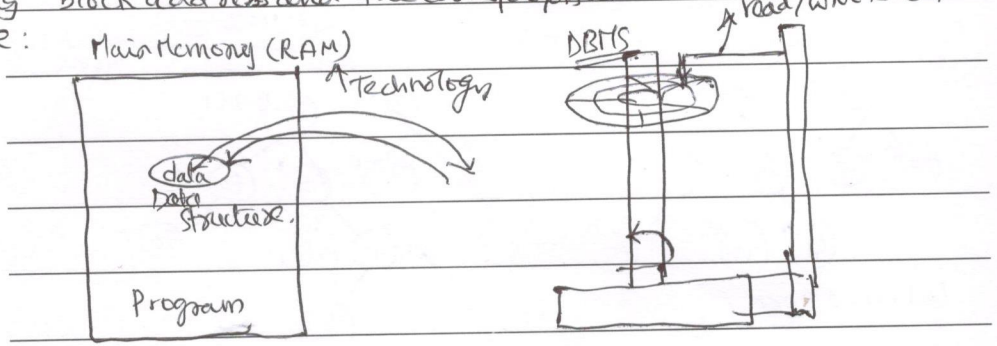
fig: Disk Structure.

1. Disk Structure) -

Block size : 512 Bytes (May vary based on manufacturer)
each block is identified with ⇒ (Track no, Sector no)
Data is stored in disk in the form of blocks. Each block is identified with the address called "offset"



we can refer any data in teh block using block address and the corresponding offset
byte:



Data is brought into RAM from the disk, Program can access the data using the data structure", Processed and the results were sent back to the disk. We can't process the data when it is in the disk.

Organization of data in the Main memory is directly useful for the program which is called as "Data structure"
Similarly organization of data efficiently in the disc that results efficient access is done using "DBMS"

Employee Table consists
EID — 10
Name — 50
Dept — 10
Section — 8
address — 50
Size = 128 Bytes

Each record takes 128 Bytes

Block size = 512 Bytes
each Block can store = $\frac{512}{128} = 4$
4 Records can be stored in each Block.

Employee Table

Eid	name	Dept	...	
1	John			
2	Tom			
3				
.				
100				

100 records

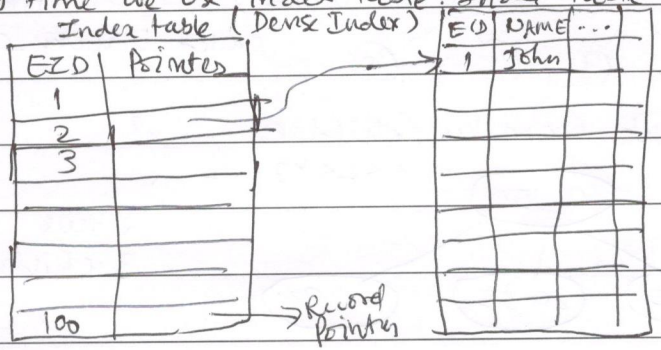
No of Blocks required : $\frac{100}{4} = 25$

∴ 25 Blocks required for storage of 100 Records.

∴ Access time to locate a byte of data, System takes lot of time.

Time depends on the numbers of blocks used to Storage.

To reduce this time we use index table. Index table looks intere following:



EID: 10 Bytes
Pointer: 6 Bytes
16 Bytes

No of entries per block = $\frac{512}{16}$

$= 32$

for 100 records = $\frac{100}{32} = \simeq 3.2$

= Maximum of 4 Blocks

∴ 4+1 Blocks are required to store the index for Index table Storage.

∴ Just 5 Blocks are required to access the data

Multi-level Indexing:-

1000 Records → 1000 entries in Index table → 40 blocks required.

Construct another level of indexing to the primary index table (Sparce Indexing)

Secondary Index Table

EID	Points
1	
33	

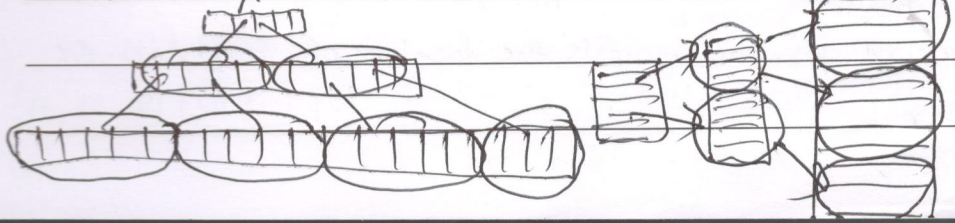
for each block in the primary index table make one entry in the secondary index table.

:

1.2 blocks requires
≃ 2 blocks required

} Search in 2 blocks much easier and reduce the time of search.

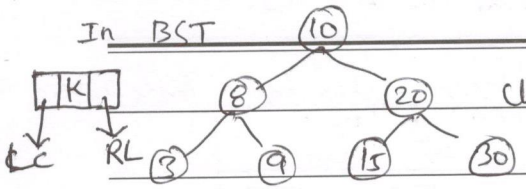
This is the original idea for the B-Trees and B^+ Trees



We want to construct self managed indexing called as B-tree.

M-way Search Tree :-

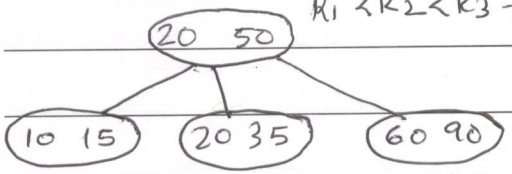
In BST each node can hold a single key each node can have Max two child nodes



Key : 1 (Per node)
Children : 2 (Per node) That is why it is called "Binary Search Tree"

If we want more key/children - - - ?

$K_1 < K_2 < K_3 \cdots$

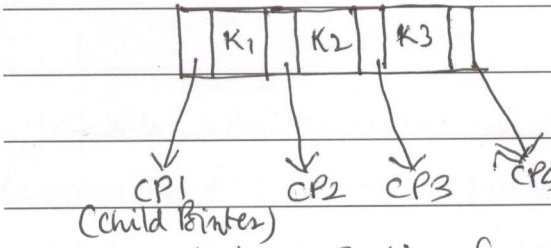


2 Keys
3 Children

3 way Search tree

We can have "m-way search tree". Here we can have (m-1) keys with 'm children'

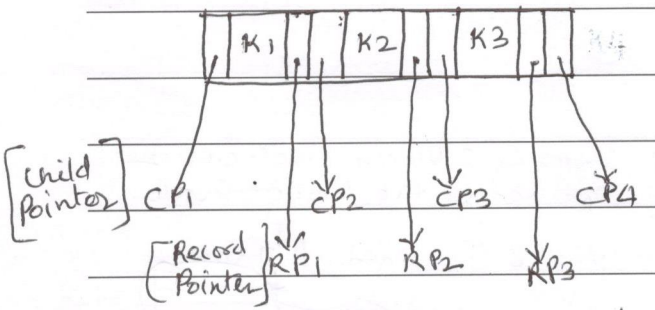
4-way Search Tree



3 Keys
4 child nodes / Pointer

Can we use this notation for creation of index tables?

Yes. This can be done using the following node structure



The record pointer connects the location of records of the database

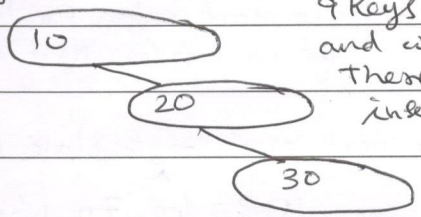
B-Tree:-

Problem in M-way SearchTree:

Ex:- Let us take 10-way SearchTree:

9 Keys
10 children

Input Keys: 10, 20, 30



Even though first node can accommodate 9 keys, there is no strict rule for this and we can insert as we wish. There is no control. If we want to insert 10 keys we get tree of level-n

There must be some rules/guidelines during the creation of M-way Search tree which are all mentioned in B-Tree creation.

B-Tree is a M-way Search tree with some guidelines.

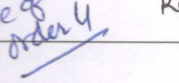
Guidelines :-

1. In every node we must fill at least half nodes/keys. $\left[\frac{M}{2}\right]$ children must be there.
2. Root can have a minimum of 2 children.
3. All leaf nodes must be at same level.
4. The creation process is "bottom-up Approach".

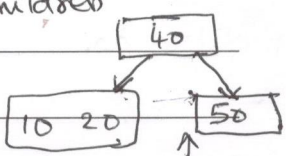
Example:- M = 4 [4-way SearchTree] { 3 Keys / 4 children
Keys: 10, 20, 40, 50

BTree of order 4

Step1



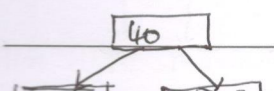
⇒ Step 2 - (Insert 50)
no space in the node
∴ it can hold 3 keys



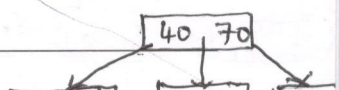
10, 20, 40, 50
node1 node2

i.e The tree is growing like bottom-up approach.

Insert 60, 70, 80

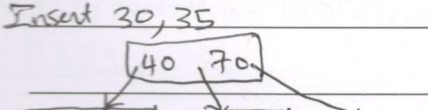


Since no space for insertion of key 80 ⇒

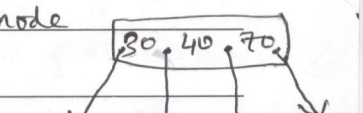


50, 60, 70, 80
node2 into new node

Insert 30, 35

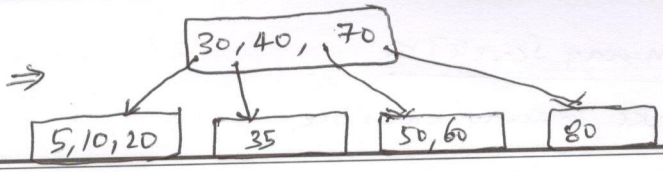


Since no space for insert of key 35



10, 20, 30, 35
node1 new node

Insert 5



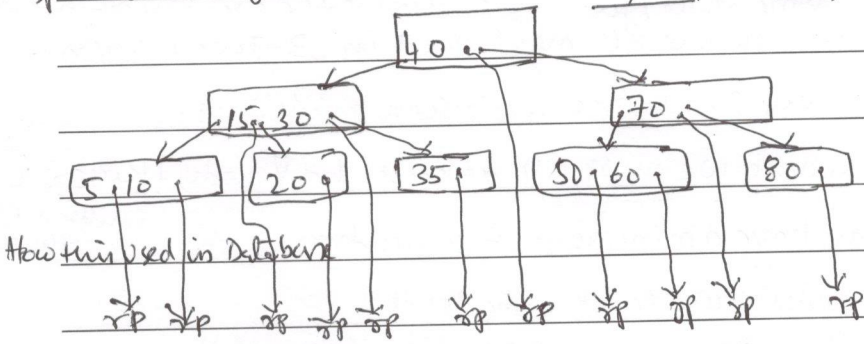
Insert 15 ⇒ no space in node1 to place key = 15 then split

5, 10, 15, 20

[Left Biased Split
i.e More keys in the left side
we can follow right biased too]

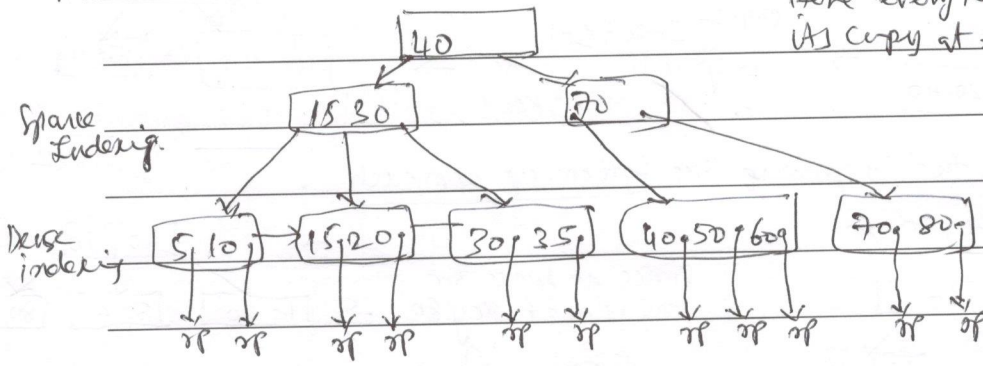
∴ 5, 10 | 20

As there is no space in the next level too to place key=15 again split it into further level as 15, 30, 40, 70 which results to:



How this used in Database

B+ Tree : In B+ Tree we don't have rp (record Pointer) from each node / inner rp are maintained at leaf nodes only. this is modified as



This is looking like a dense indexing / multilevel indexing : Used for maintenance of database records.

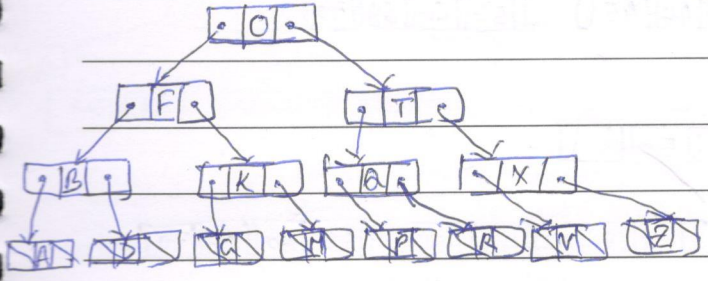
Insert operation in B-Tree:-

1. Initialize 'x' as root [x is an element to insert into tree]
2. While 'x' is not a leaf, do the 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 one of the two parts of y. If x is smaller than mid key in 'y' then set x as the first part of y.
   Else second part of 'y'. When we split 'y' 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.

B-Tree : Extension of BST.

B-Tree of order 'm' satisfies the following properties:
1. Each node has atmost m children.
2. Each internal node has atleast $\lceil \frac{m}{2} \rceil$ children.
3. Root has atleast 2 children if it is not a leaf
4. A non-leaf node with K children has (K-1) keys
5. All leaves appear in same level.

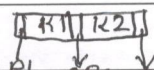
m=3 B-Tree with order=3 is also a BST.



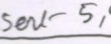
Create a B-Tree of order 3 with the following key elements:
-5, 9, 3, 7, 1, 2, 8, 6, 0, 4

∵ order of B-Tree is 3
it can have Max: 3 child nodes for each node
and each node can have 2 keys

The node structure



Insert 5,9 ①

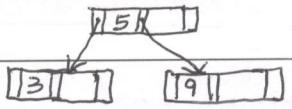


Insert 3 ②

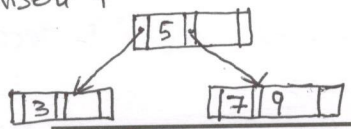
∵ no placement for 3 in node
(∵ Node can hold Max 2 Keys)

apply split

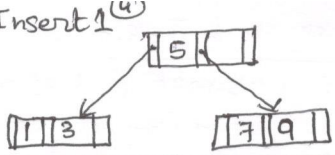
3, 5, 9



Insert 7 (3)

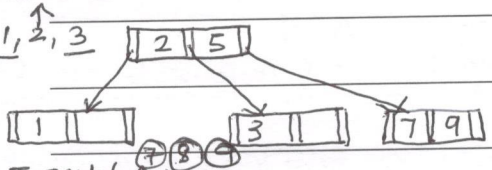


Insert 1 (4)



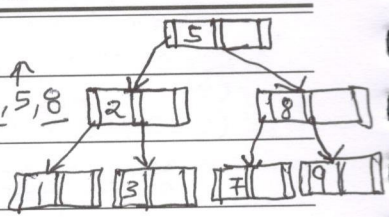
Insert 2 (5)

1, 2, 3

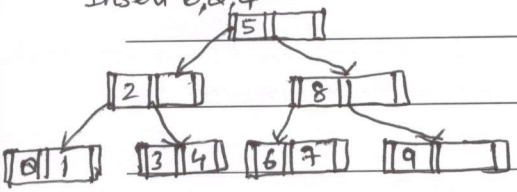


Insert 8 (6)

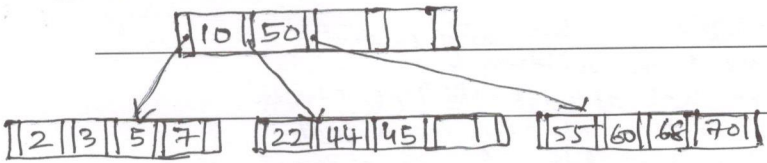
7, 8, 9 ⇒ 2, 5, 8



Insert 6, 0, 4 (7) (8) (9)

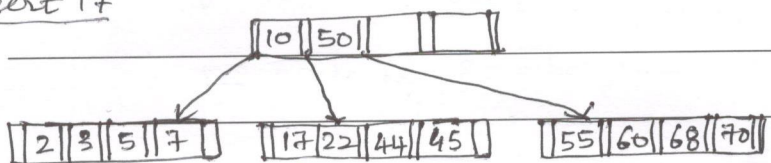


other cases Consider the following B-Tree with order = 5

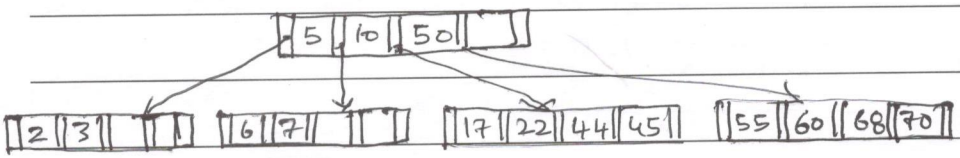


Insert 17

⇒



Insert 6 2, 3, 5, 6, 7



B-Tree: Deletion of a 'key element' from B-Tree

Consider the following B-Tree with order (m) = 5

i.e minimum children in each "internal node" is $\lceil \frac{m}{2} \rceil = 3$

maximum children for each node : 5

minimum key (elements) in each leaf node = 2 $\because \lceil \frac{m}{2} \rceil - 1 = 3 - 1 = 2$

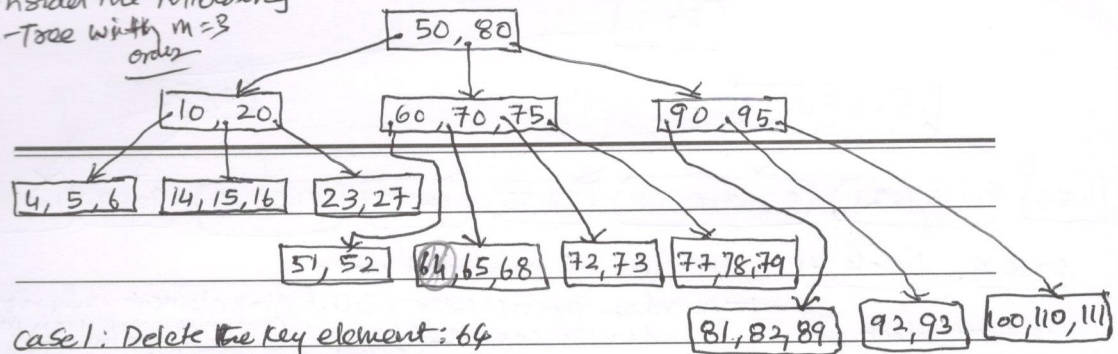
maximum number of keys in each node : 4

cases to be considered

1) If target key (key to be deleted) is in leaf node?

- That leaf node contains more than minimum no of keys
- That leaf node contains minimum no of keys
  1) Borrow from immediate left node (siblings)
  2) Borrow from immediate right
  3) can't borrow from left/right

Consider the following B-Tree with m=3 order

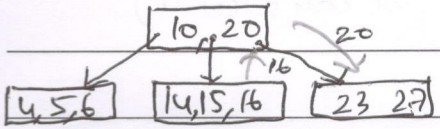


Case 1: Delete the key element: 64

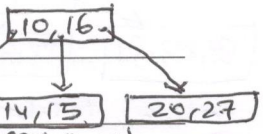
64 appears in node (leaf node) contains more than minimum no of keys.
∴ Considered as simple delete operation. Just delete it from the node
After deletion observe that all nodes has min of 2 elements (B Tree Property - except root node.

Case 2: Delete 23:

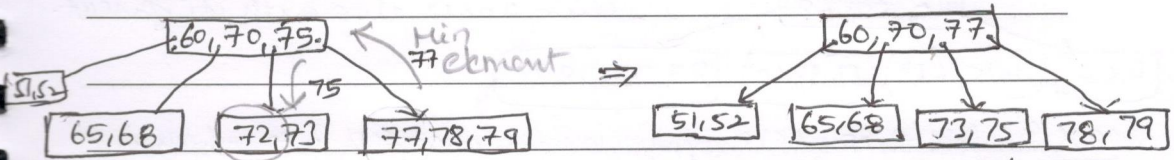
It is at the node which contain only 2 elements. So we can't delete it directly [∵ every node should contain min no of keys = 2]
This node can take/borrow the elements from its immediate left/right siblings) The node can borrow the keys only if the other node contains more than minimum no of elements.



The element can be transferred using its parent. Then delete key = 23

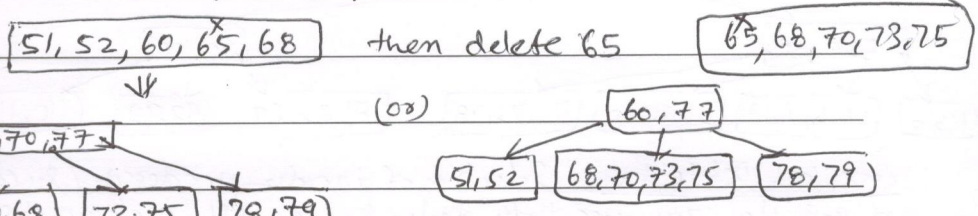


Case 3: Delete 72:- (Borrow element from its right (Siblings)

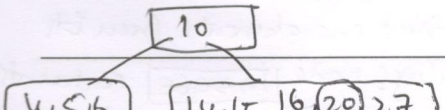


Case 4: Delete 65 (Can't borrow element either from left/right)

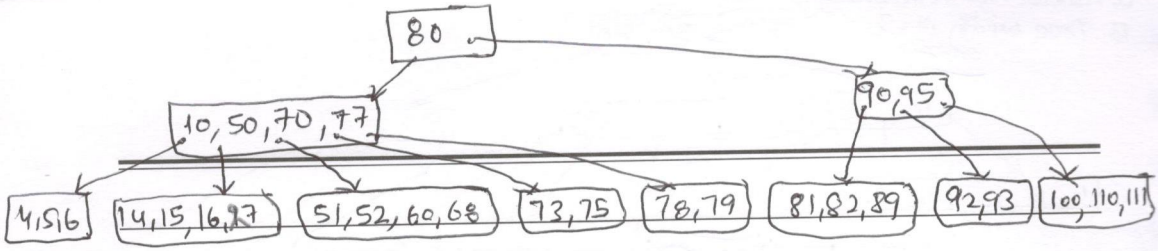
In this case merge the node with either its left/right sibling node along with its corresponding parent node element.



Case 5: Delete 20: (Can't borrow from its left/right siblings then the parent node merges its siblings along with the parent node)

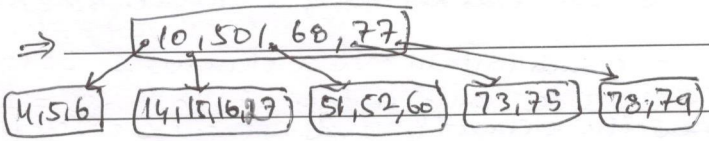


Not Possible since parent include only one element [∵ voilates B-Tree Property
∴ Merge with its parent node

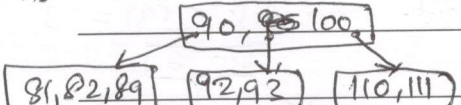


case 6: Delete 70: Deletion of internal node

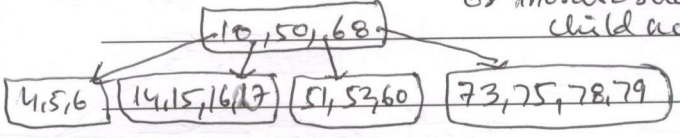
We can select — inorder predecessor (Select Max element only if it contains more than min elements)
inorder successor (select min element ouly if it contains more than min elements)



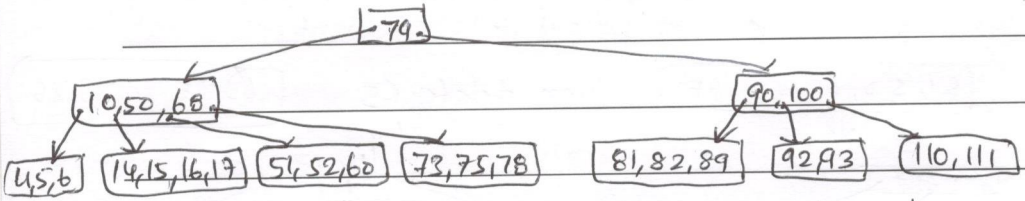
case 7: Delete 95 Now we can't use inorder predecessor because left sibling node has only 2 elements. Then select inorder successor of right sibling (Select minimum key from its right) As it include more than 2 elements we can select it)



case 8: Delete 77 :- Here we can't apply either inorder predecessor or inorder successor. Those merge its child nodes along with its parent.

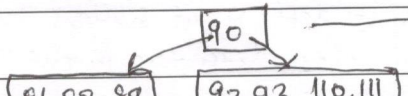


case 9: Delete 80: Here apply inorder predecessor of 80 which is 79 (Select maximum element from all of its nodes). As that node contains more than minimum elements, replace 80 with 79.



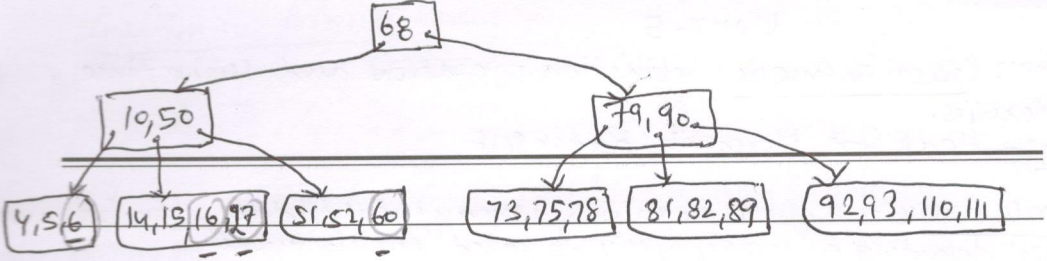
delete 100

case 10: Delete 100: Selection of inorder predecessor / successor is not possible. Because both nodes has exactly 2 elements. Apply merge



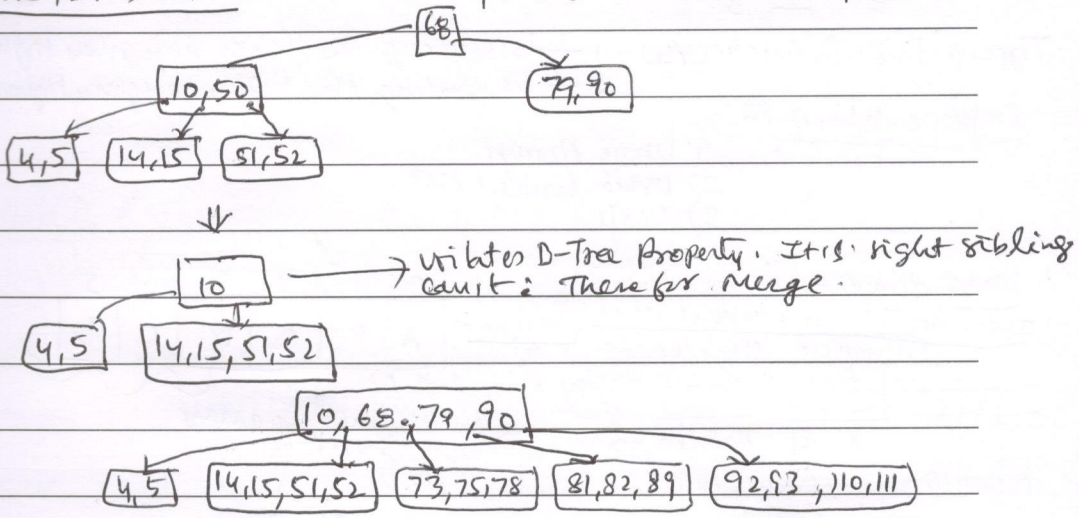
This node contains only one (violate the property of B-Tree)

∴ Borrow one elements from its sibilings node 10,50,68 contains more elements. ∴ Possible



Case 11: Delete 6, 27, 60, 16 is possible directly from the respective nodes as they we were satisfying B-Tree property (>=2)

Case 12: Delete 50 : we can't replace it with inorder predesso/successo.



Now we observe that B-Tree is shrinking upon deletion of elements. The Height of the tree is less than the original height of the B-Tree.