UNIT-5

B-Tree (Balanced Search Tree)

height = $O(\log(n))$

A B-Tree is a rooted tree (whose root is root[T]) having the following properties.

1. Every node x has the following fields.
   (a) n[x] the number of keys currently stored in node x
   (b) the n[x] keys themselves stored in non-decreasing order, so that $key_1[x] \le key_2[x] \le \dots \le key_{n[x]}[x]$
   (c) leaf[x], a boolean value is True if x is a leaf and false if x is an internal node.
2. Each internal node x also contains n[x]+1 pointers $c_1[x], c_2[x] \dots c_{n[x]+1}[x]$ to its children. Leaf nodes have no children, so their c_i fields are undefined.
3. The keys $key_i[x]$ separate the range of keys stored in each subtree; if k_i is any key stored in the subtree with root $c_i[x]$ then.

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

4. All leaves have the same depth, which is the tree's height h.
5. There are lower & upper bounds on the number of keys a node can contain. These bounds are expressed in terms of a fixed integer $t \ge 2$ called the minimum degree of the B-tree.—

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

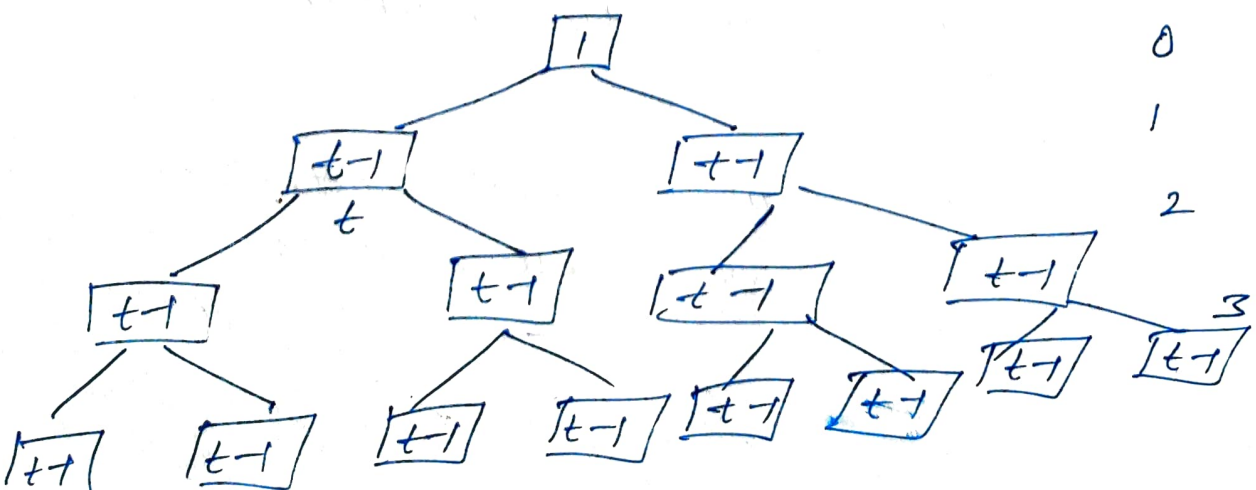
(b) Every node can contain at most $2t-1$ key. Therefore an internal node other than the root has at least t children. We say that a node is full if it contains exactly $2t-1$ keys.

Theorem: If $n \geq 1$, then for any n-keys B-tree T of height h and minimum degree $t \geq 2$, then

$$h \leq \log_t \frac{n+1}{2}$$

The root contains at least one key. All other nodes contains at least $t-1$ keys. There are at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^{i-1}$ nodes at depth i and $2t^{h-1}$ nodes at depth h.

Depth	No of nodes
0	1
1	2
2	$2t$
3	$2t^2$



$$n \geq 1 + (t-1)(2 + 2t + 2t^2 + 2t^3 + \ldots + 2t^{h-1})$$

$$n = 1 + 2(t-1)(1 + t + t^2 + t^3 + \ldots + t^{h-1})$$

$$n = 1 + 2(t-1)\left[\frac{(t^h - 1)}{(t-1)}\right]$$

$$n \geq 1 + 2t^h - 2$$

$$n \geq 1 + 2t^h - 2 \Rightarrow n \geq 2t^h - 1$$

$$n+1 \geq 2t^h$$

$$\frac{n+1}{2} \geq t^h$$

$$t^h \leq \frac{(n+1)}{2}$$

Taking base-t logarithms of both sides, we get $h \leq \log_t \frac{(n+1)}{2}$

Insertion in B-Tree

The insertion of a key in a B-tree requires first traversal in B-tree. Though traversal it will find that key to be inserted is already existing or not. Suppose key does not exist in the tree then through traversal it will reach leaf node.

Two cases for inserting the key :-

(a) Node is not full

(b) Node is already full.

(1) If the leaf node in which the key to be inserted is not full, then the insertion is done in the node. A

node. A node is said to be full if it contains $m-1$ maximum of $m-1$ keys, given the order of the B-tree to be m.

② If the node were to be full, then insert the key in order into the existing set of keys in the node. Split the node at its median into two nodes at the same level, pushing the median element up by one level.

Note that the split nodes are only half full. Accomodate the median element in the parent node if it not full. otherwise repeat the same procedure and this may even call for rearrangement of the keys in the root node or the formation of a new root itself.

Thus, a major observation pertaining to insertion in a B-tree is that, since the leaf nodes are all at the same level, unlike an m-way search tree, the tree grows upwards.

Eg
10 20 50 60 40 80, 100, 70, 130, 90, 30, 120, 140, 25
35, 160, 180.

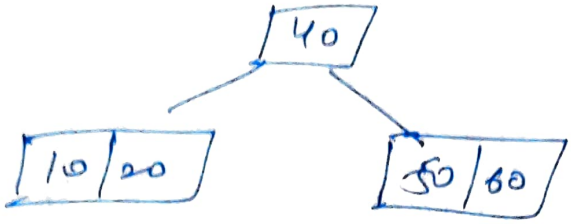
Insert 10 — [10]

Insert 20 — [10 | 20]

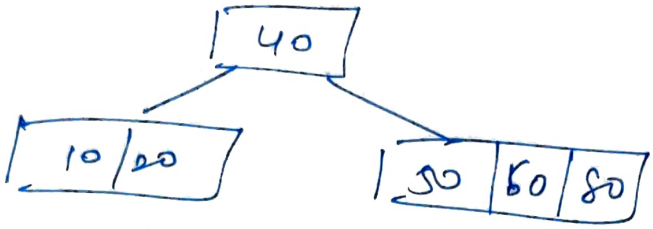
Insert 50 — [10 | 20 | 50]

Insert 60 — [10 | 20 | 50 | 60]

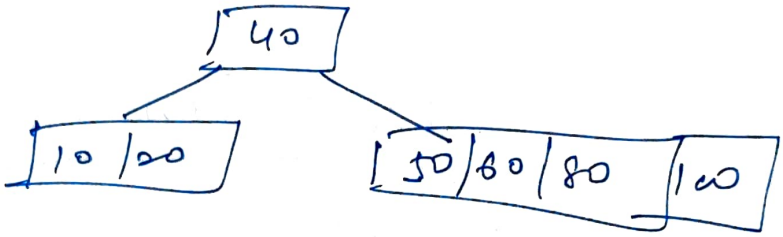
Insert 40



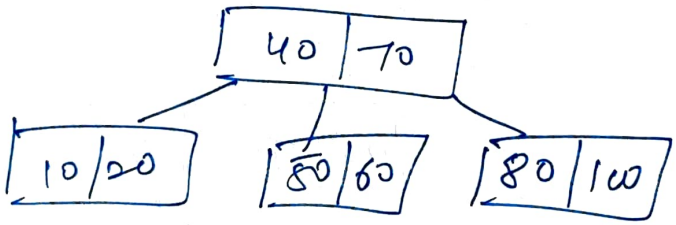
Insert 80



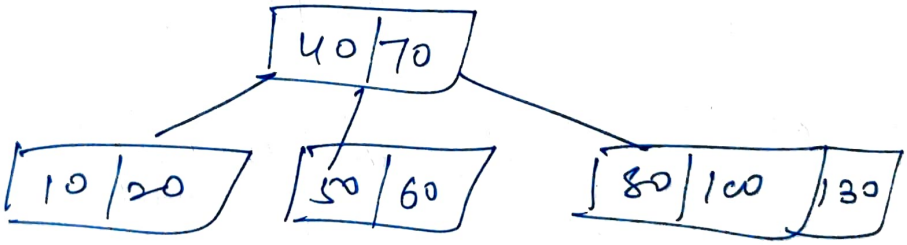
Insert 100



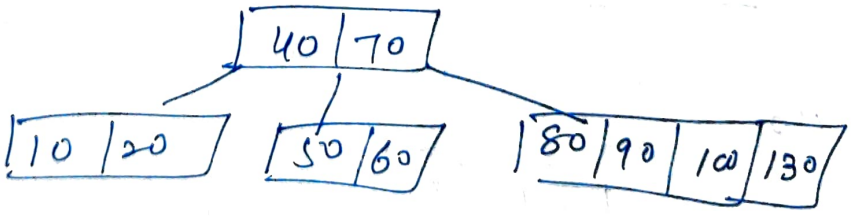
Insert 70



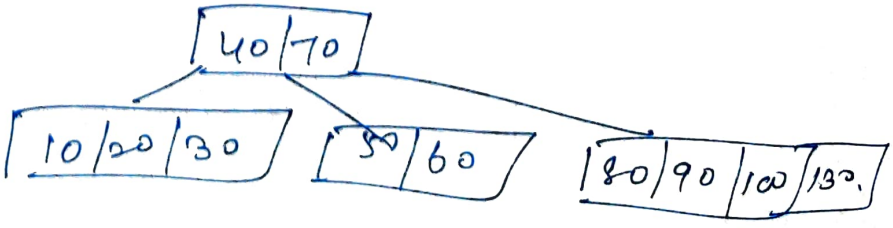
Insert 130



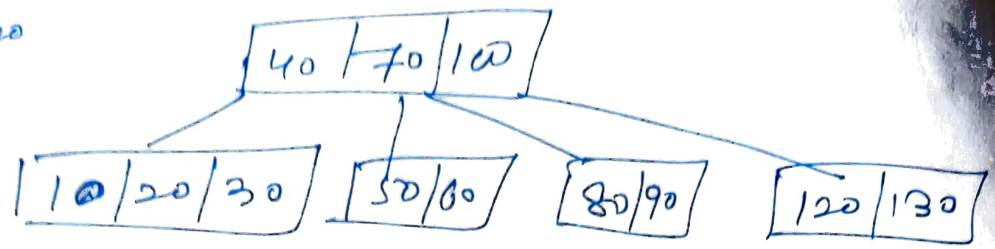
Insert 90



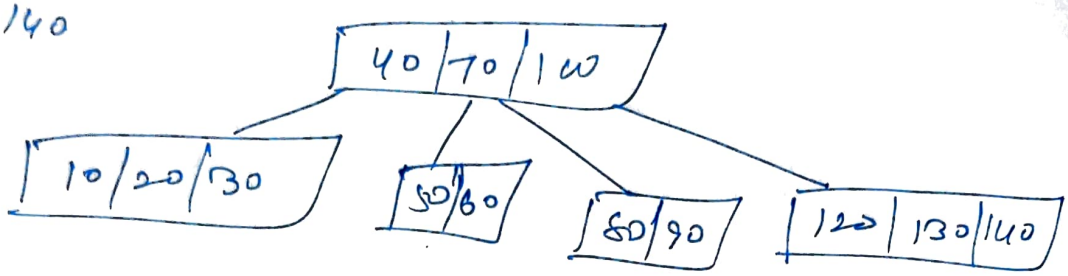
Insert 30



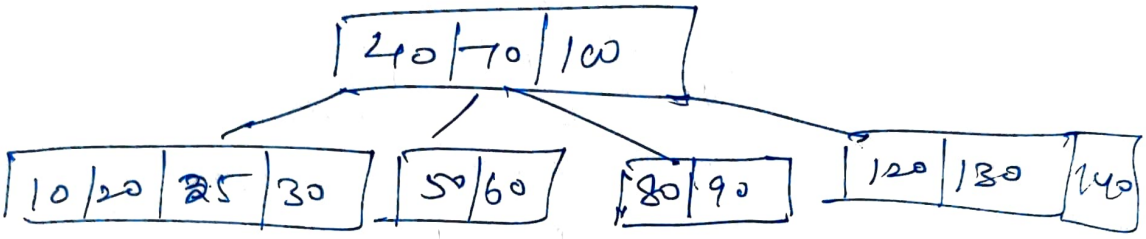
Insert 120



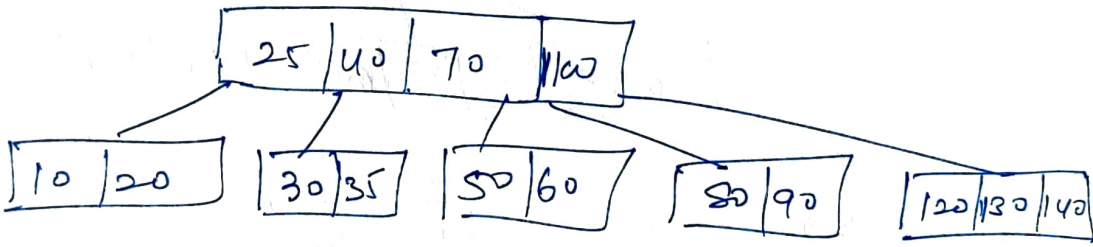
Insert 140



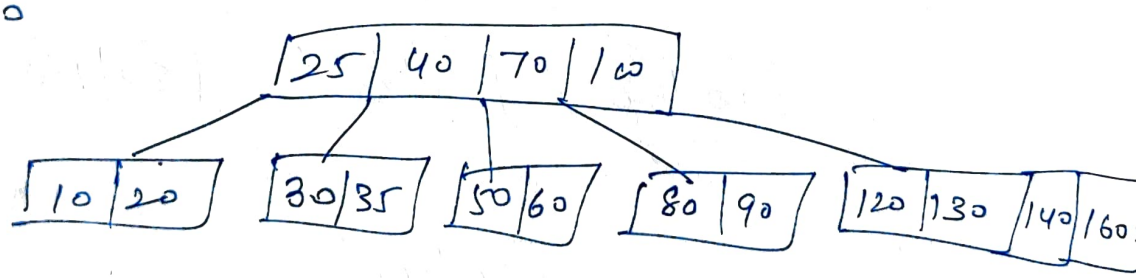
Insert 25



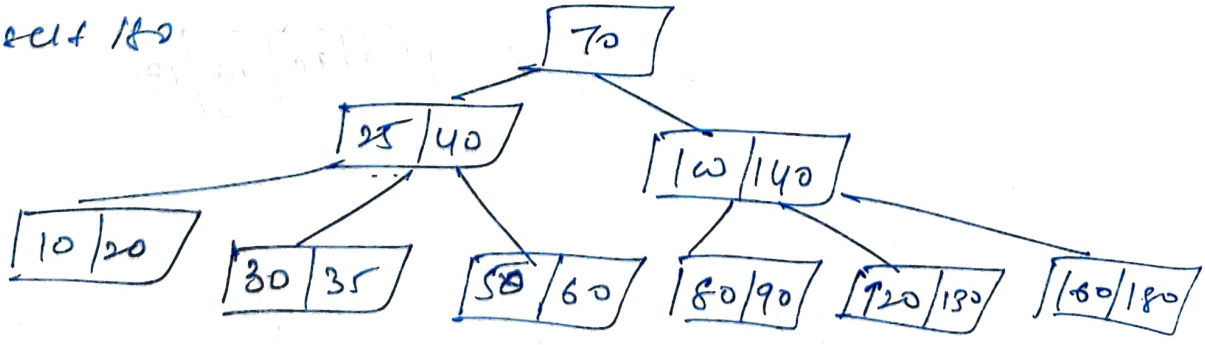
Insert 35



Insert 160



Insert 180



Deletion in B-tree

Deletion of key also requires first traversal in B-tree. After reaching on particular node, two cases may occur.

(1) Node is leaf node
(2) Node is non-leaf node.

(1) First case :- Suppose node has more than minimum number of keys then it can be easily deleted. But suppose it has only minimum number of keys then first will see number of keys in adjacent leaf node if it has more than minimum number of keys then first key of the adjacent node will go to the parent node and key in parent node which is partitioning will be combined together in one node. Suppose now parent has also less than the minimum number of keys then same thing will be repeated until it will get the node which has more than the minimum number of keys.

(2) Second case :- key will be deleted and it's predecessor or successor key will come on its place. Suppose both nodes of predecessor & successor key have minimum number of keys then the nodes of predecessor and successor keys will be combined.

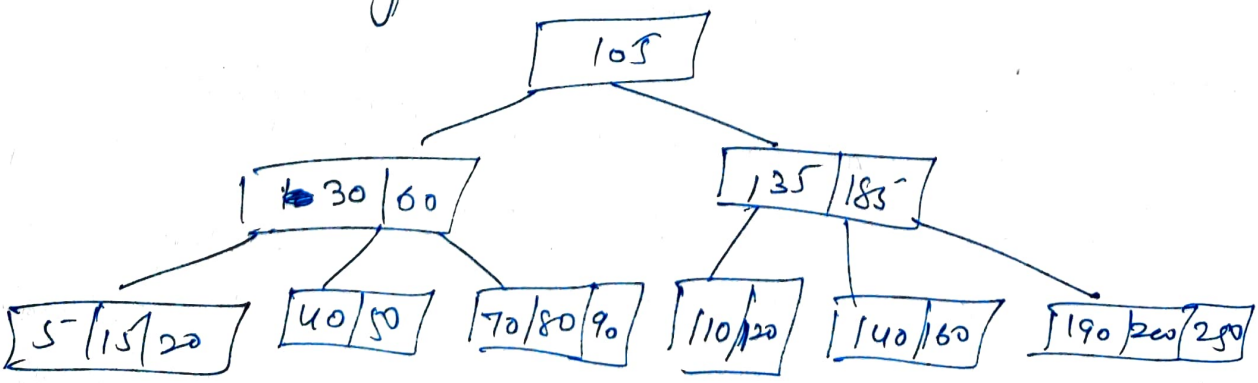
Thus, while removing a key from a leaf node, if the node contains more than the minimum number of elements, then keys can be easily removed. However if the leaf node contains just number of elements then question rises for an element to fill the vacancy choose from left sibling node or from

Right Sibling. If the left Sibling node has more than the minimum number of keys, pull the largest key up into the parent node and move down the intervening entry from the parent node to the leaf node where the key is to be deleted. Otherwise pull the smallest key of the right sibling node to the parent node and move down the intervening parent element to the leaf.

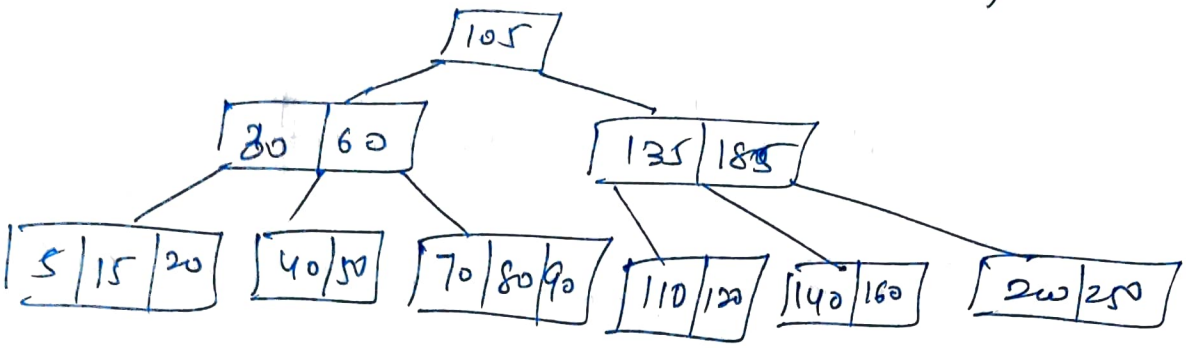
(3) If both the Sibling nodes have only minimum number of entries, then create a new leaf node out of the two leaf nodes and the intervening element of the parent node, ensuring that the total number does not exceed the maximum limit for a node.

If while borrowing the intervening element from the parent node, it leaves the number of keys in the parent node to be below the minimum number, then we propagate the process upwards ultimately resulting in a reduction of height of the B-tree.

eg B-tree of order 5

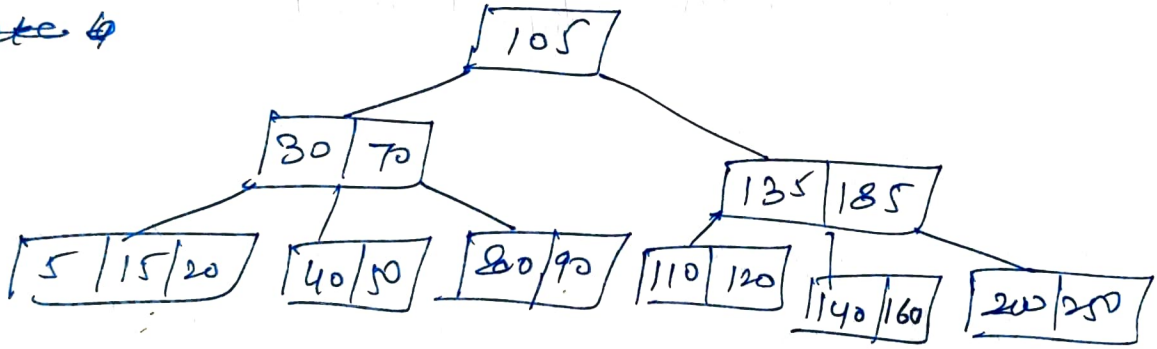


delete 190 → it is a leaf node easily deleted,



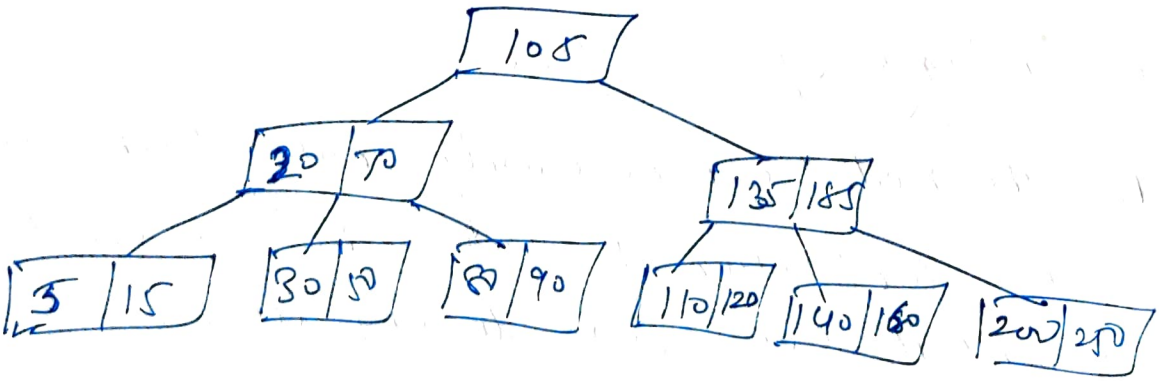
Delete 30 → non-leaf node, so first it will be deleted from the node and then the element of the right child will come in that node.

Delete 30

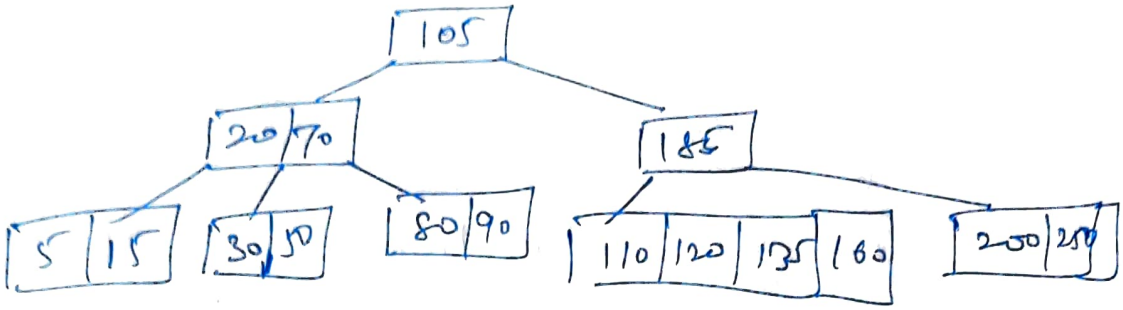


Delete 40 → will be deleted from the leaf node then left side element in the parent node will come in leaf node and then last element of the left side node of the parent node will come in parent node

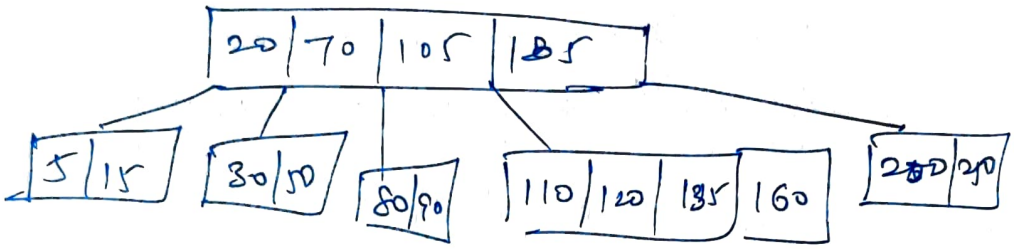
Delete



Delete 140



But at least two elements should be in child node so it will go in left node.



Application of B-Tree

The main Application of B-tree is the organisation of huge collection of records into a file structure. The organisation should be in such way that any record in it can be searched very efficiently, ie insertion, deletion and modification operations can be carried out perfectly and efficiently.

Advantages of B-Tree

(1) The nodes can be easily inserted and deleted.

(2) The declaration of size is not required in advance and it can grow to any limit.

(3) The in-order traversal of a tree always produces the sorted list so it is easy to keep the sorted data.