Deletion in Red-Black Tree

Like insertion, recoloring and restructuring (rotation) will be required to maintain the Red Black properties

In Insert, we used to check color of uncle to decide the type of correction needed.

In delete, we look at colour of sibling to decide the type of correction needed.

→ In insertion, the main property that gets violated is occurrence two consecutive red nodes.
While, in delete, the main property that gets disturbed is the change of black height in subtrees as deletion of

of a black node may cause reduce black height in one of the root to leaf path.

Deletion is relatively complex operation, a notion of double black is used in implementing the required correction. When a black node is deleted and is replaced by a black child, the child is marked double black. The main task is now to convert this double black to a single black.

Steps Involved in deletion

1) Perform the standard BST delete. When we perform standard delete in BST, we always end up deleting a

node which is either leaf node or has only one child
(For an internal node, we copy the successor/predecessor and then recursively call delete for successor/predecessor, finally, node getting deleted would be just a leaf node or a node with one child)
So, we need to handle only a case of a node being leaf or having one child.

Let v be the node to be deleted
u is the child replaces v

(u would be NULL when v is leaf, color of NULL is considered as black).

(2) Simple case — If either u or v is red — we mark

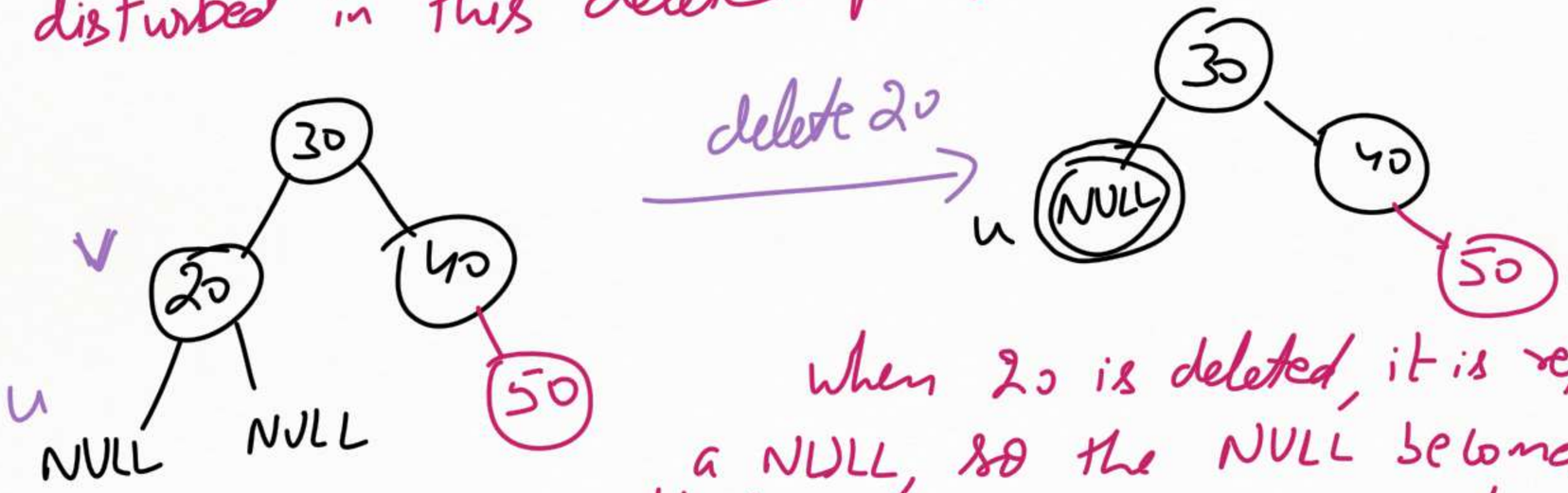
the replaced child as black (no change in black height)
(Please note:- both u and v cannot be red because v is a parent of u and two consecutive reds can not be present in a valid red-black tree.

Example

(3) If Both u and v are black

3.1) Color u as double black. Now, this double black is needed to be converted to single black.

Note that if v is leaf, the u is NULL and color of NULL is assumed to be black. So, the deletion of a black leaf also causes a double black (black height is disturbed in this deleted path)



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black. Deletion process is not done yet, the double black of NULL is to be adjusted to single black.

(3.2) Do the following, while the current node u is double black and it is not root.

Let the sibling of node be s.

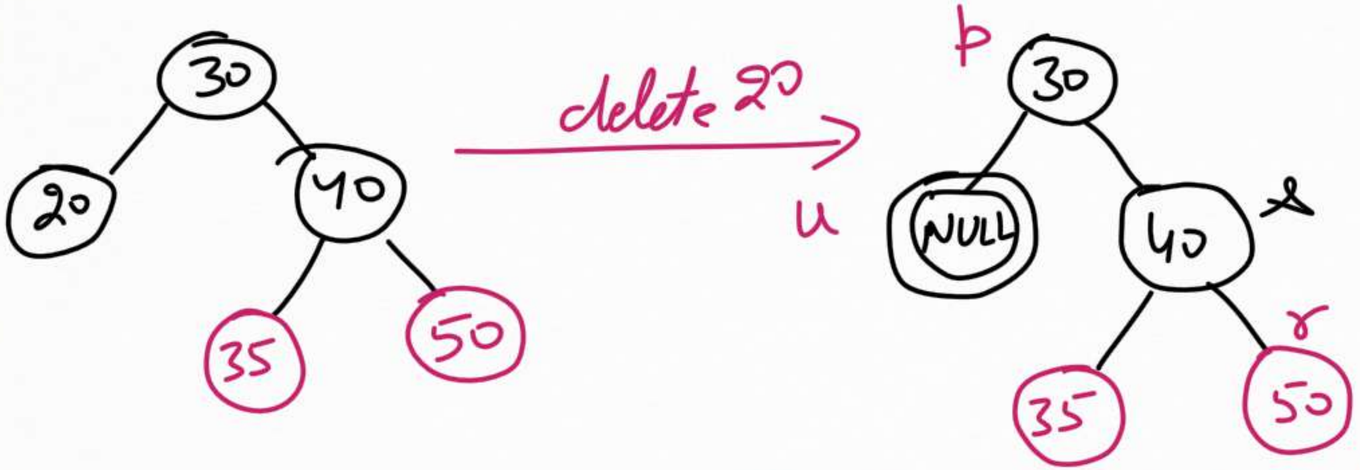
(a) If sibling s is black and atleast one of sibling's children is red, perform rotation. Let the red child of s be r. This case can be divided into four subcases depending upon positions of s and r

(i) Left Left Case: s is the left child of its parent and r is the left child of s or both children of s are red.

(ii) Left-Right Case: s is the left child of its parent and r is the right child.

(iii) Right Right Case: s is the right child of its parent and r is the right child of s or both children of s are red.

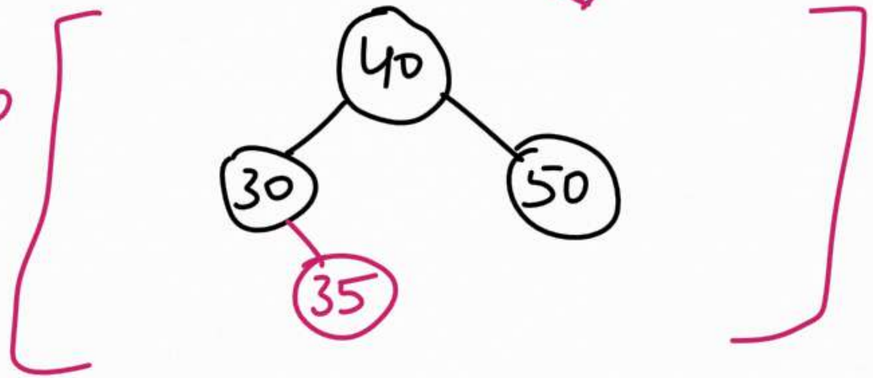
(iv) Right-Left Case: s is the right child of its parent and r is the left child



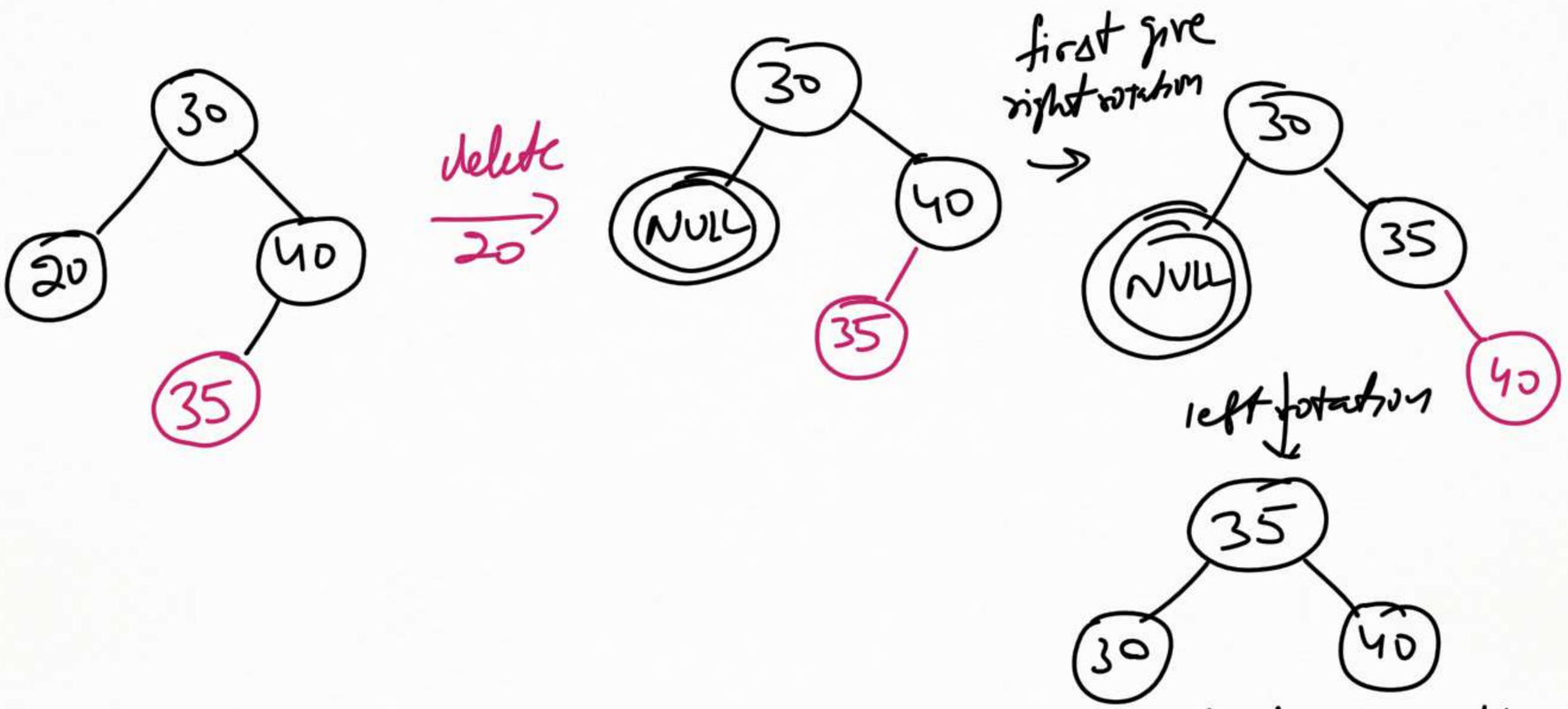
Case — 3.2.a.iii

Sibling is black and its right child (actually here both) is red

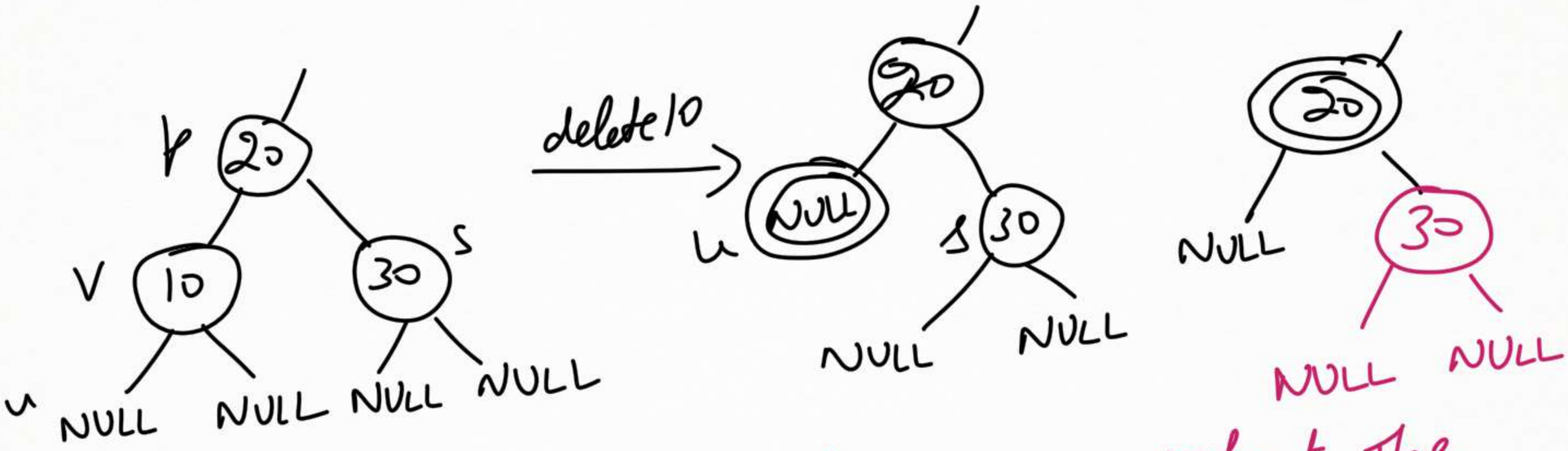
Apply rotation (Single left rotation)



Node Colouring need to looked into and explained



(b) If sibling s is black and its both children are black Perform recoloring and recursively do it if parent is black.

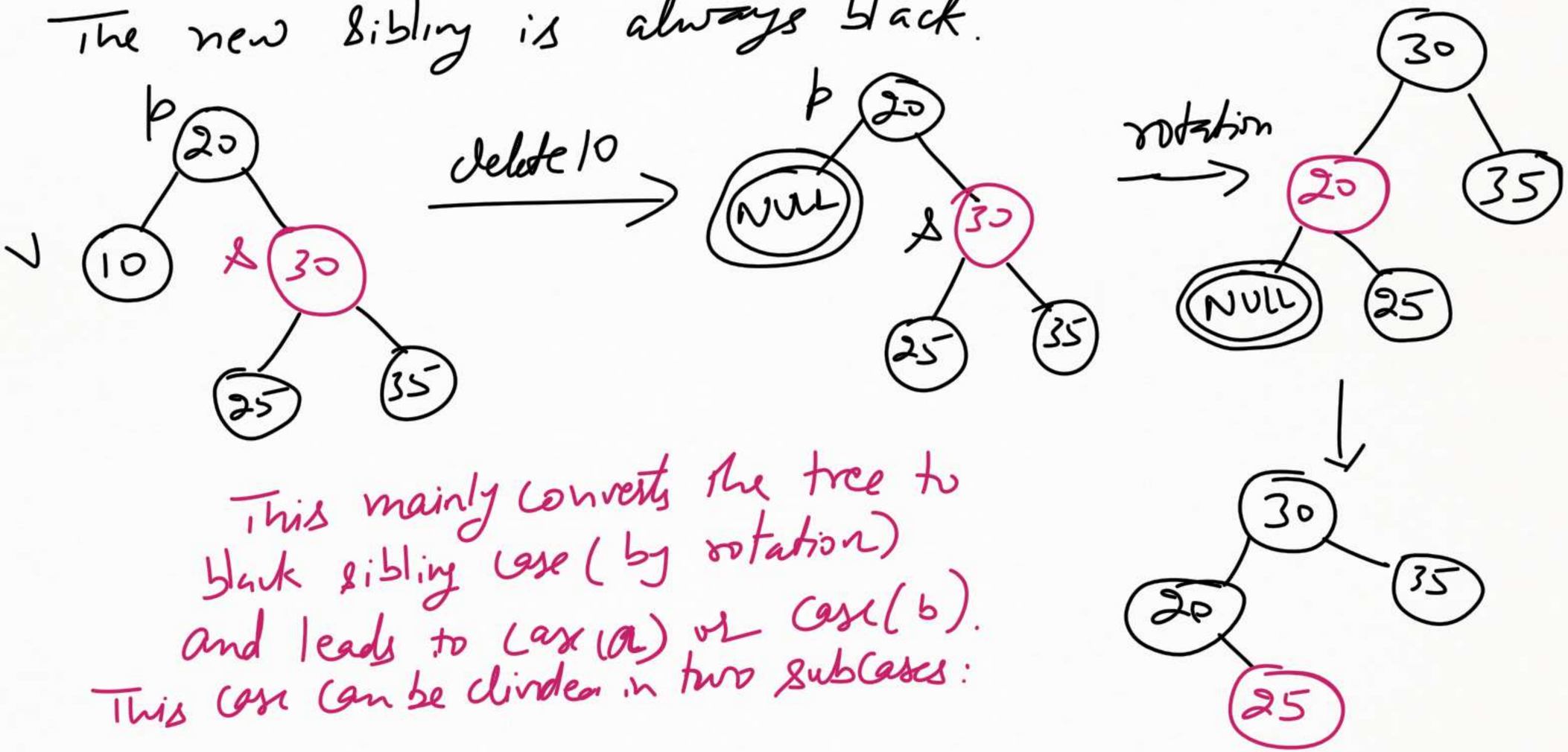


Here, if the parent p would have been red, we did not need to recur for parent because we can simply set it to black colour.

red + double black = single black

repeat the recoloring for 20 (p) to remove double black from it

(c) If sibling is <u>red</u>, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black.



This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or case (b).
This case can be divided in two subcases:

(i) left case : s is left child of its parent. We right rotate parent p

(ii) right case : s is the right child of its parent. we left rotate the parent p

3.3) If u is root, make it single black and return (black height of complete tree is reduced by 1)

B-Tree

B.S.T, AVL Tree, Red Black → each node used to have only one key : are not suitable if the application require lot of data/keys place in a localized area. In such a case, another tree structure called B-Tree is better choice

B-Tree a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children

It is also called height-balanced m-way tree

Why do we need B-Tree :- Their need arises for lesser time in accessing the physical storage media like a harddisk. The secondary storage devices are slower with a larger capacity and for such devices, we need data structures that minimize the disk accesses.

In comparison to B-Tree, AVL and Red-black can store only one key per node. If you are required to store a large number of keys, then the height of such trees becomes very large and access time increases.

In B-Tree, one node can have many keys and can have multiple child nodes. This decreases the height significantly that allows faster disk access. These were originally invented for storing data structures on disk, where locality is very crucial.

B-Trees have a high branching factor, much larger than 2, which ensures that few disk reads are needed to navigate to the place where data is stored.. B-trees may also be useful for in-memory data structures.

B-Tree Properties

1) For each node x, the keys are stored in increasing order
2) In each node, there is a boolean value $x.leaf$ which is true if x is a leaf node
3) If $\underline{\underline{m}}$ is the order of the tree, each internal node can contain at most $\underline{\underline{m-1}}$ keys along with a pointer to each child (m child subtrees)

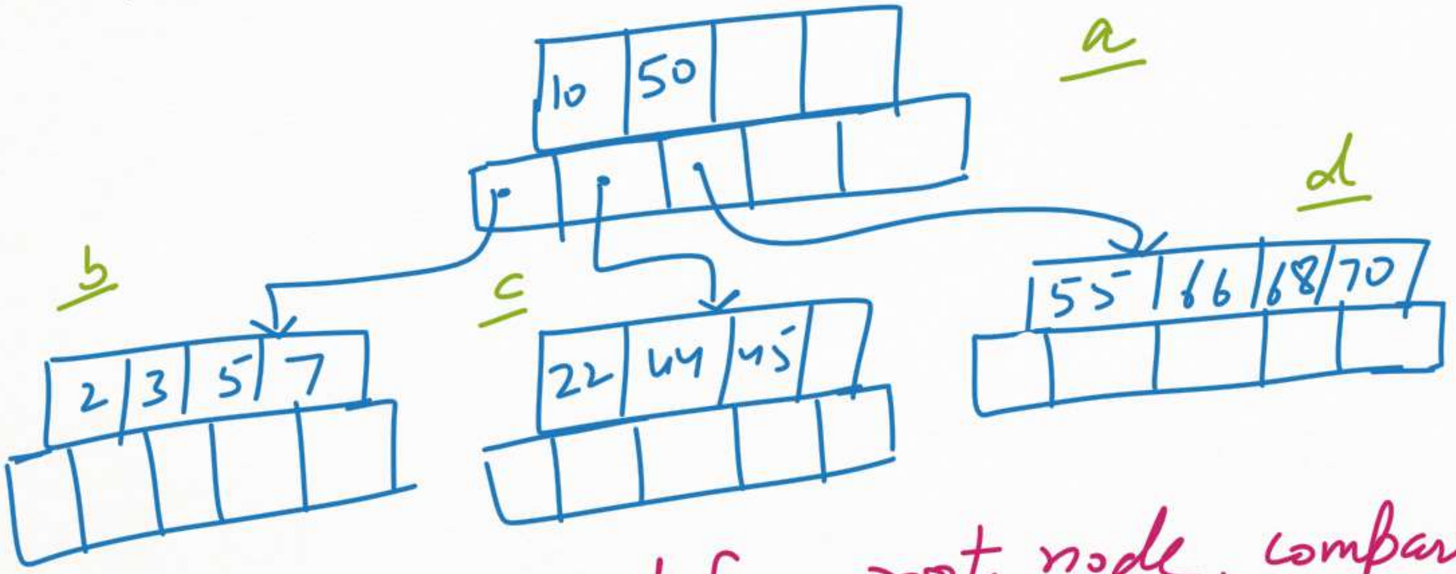
4) Each node except root can have atmost $\underline{\underline{m}}$ children and at least $m/2$ children

5) All leaf nodes have the same depth (i.e. height h of the tree)

6) The root has at least 2 children and contains a minimum of 1 key

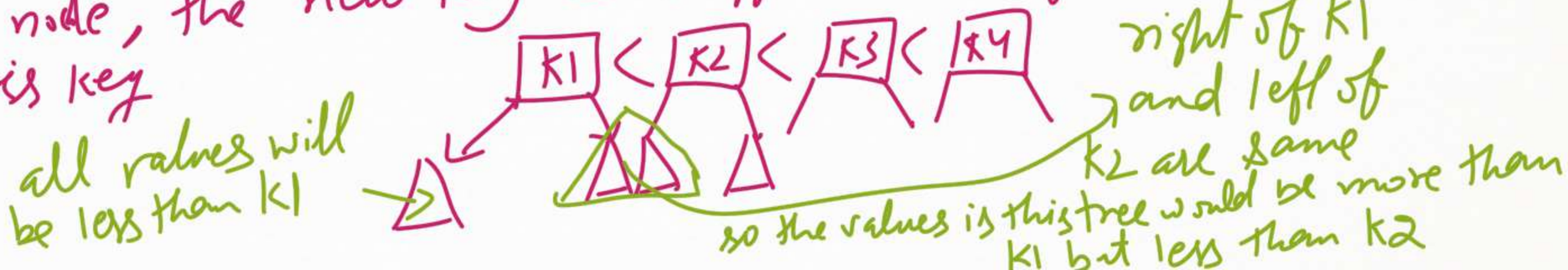
Let us look at an example of construction of a B-Tree by inserting some data

We have a B-Tree with $m/M = 5$



insert 17 :- Start from root node, compare the keys to be inserted with already present keys

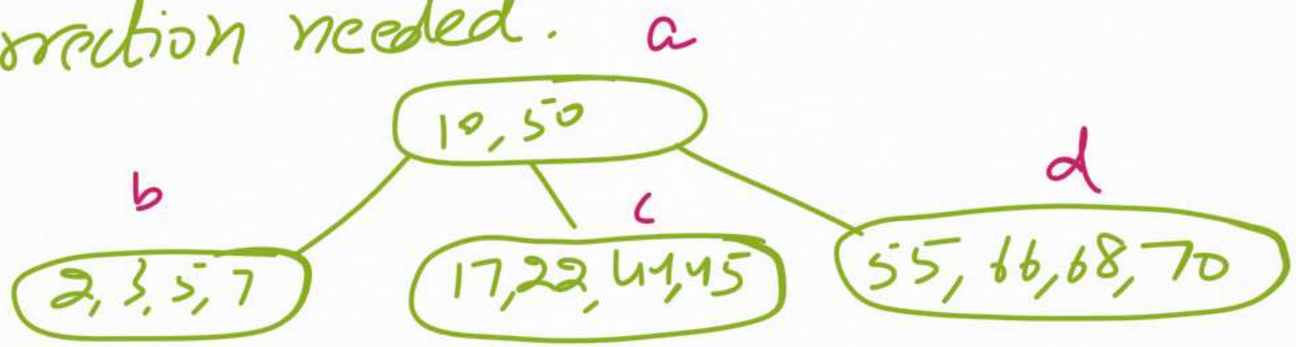
If the key to be inserted in less than first key present in the node, the new key can appear in left subtree of this key



17 is more than 10 but less than 50
So, 17 will appear in right subtree of 10 (which would be logically left subtree of 50) i.e. here 17 will be inserted in node C node C will become

17	22	44	45

After insertion of 17, total no. of keys in node C is less than M so there is no overflow, there is no correction needed.



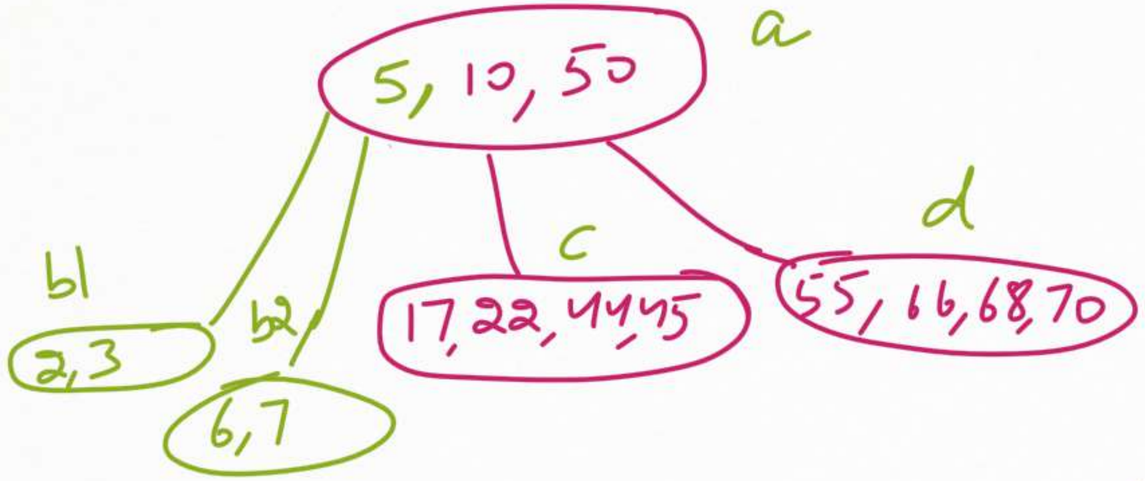
Insert 6 : 6 is less than 10, go to left pointer of 10
there node is b, lets try to insert it in b

2, 3, 5, 6, 7 } Total count of key is equal to M → overflowed

This node has to be split

left [2, 3]
Middle 5
Right [6, 7]

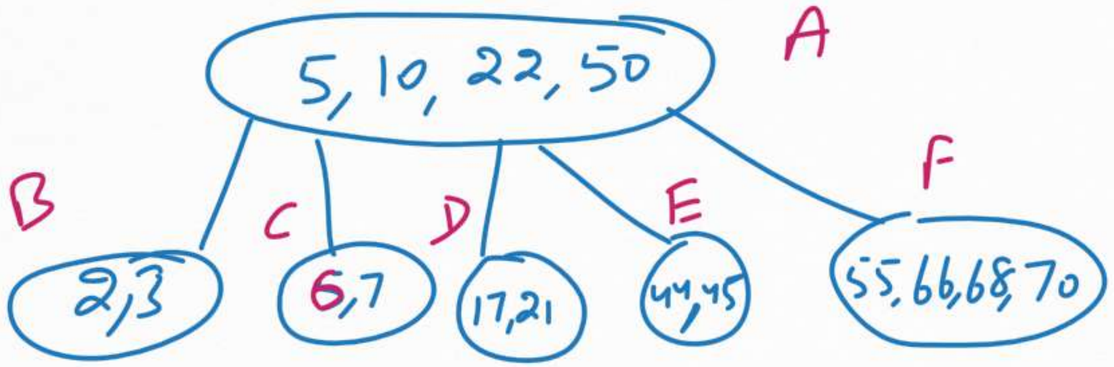
⇒ left and right becomes nodes; Middle is added to the node above with left and Right as its children



lets Insert 21 :- 21 is more than 5
21 is more than 10
21 is less than 50

So, 21 will be added to right of 10 and left of 50

i.e node C, adding to node C
17,21,22,44,45 > require splitting
22 will upside



(New label used for nodes)

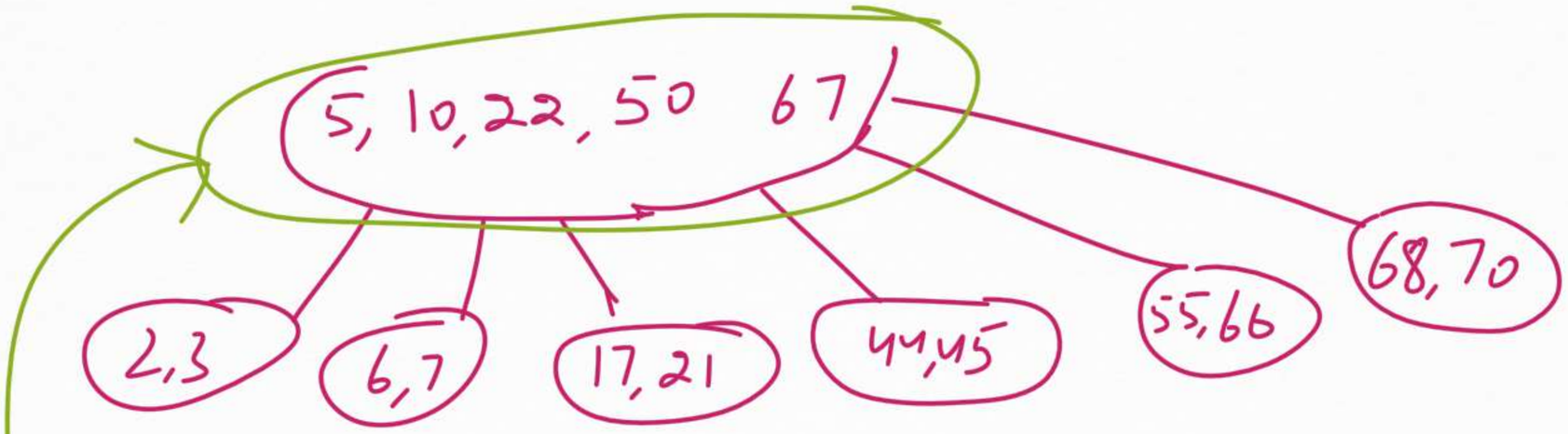
Insert 67 → is more than 5, 10, 22, 50
So 67 will appear in the right subtree of 50.

lets try to add it to node F

55, 66, 67, 68, 70 → it require splitting

left 55, 66
middle 67
right 68, 70

67 will move to higher (upper level node)

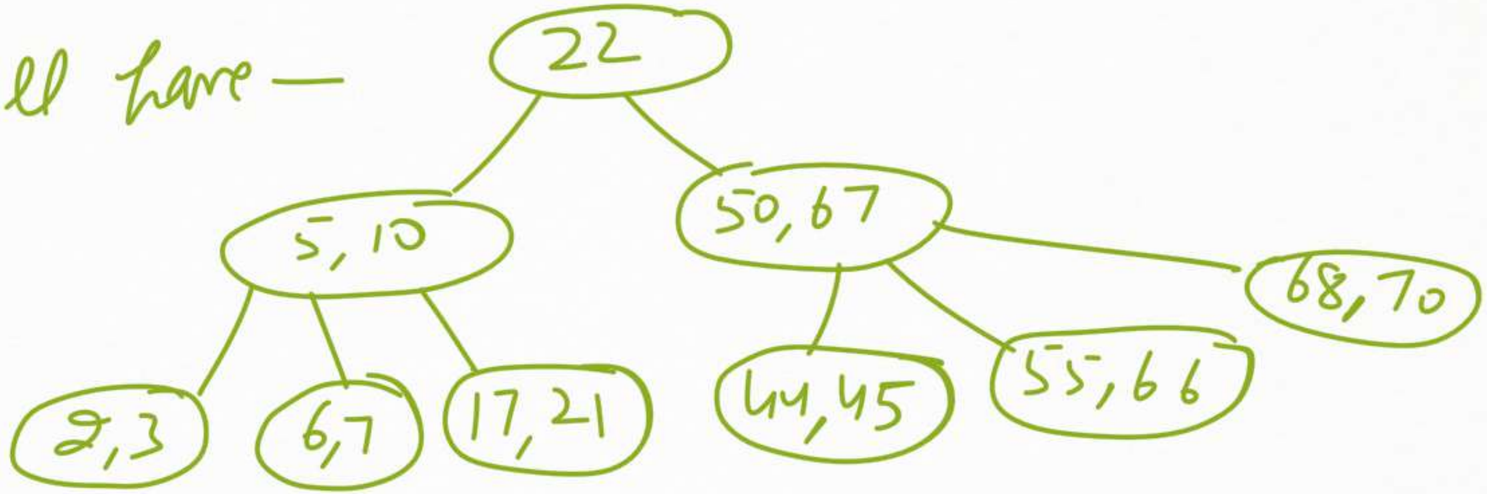


root node has overflowed, here also splitting would be required

5, 10, 22, 50, 67

On the basis of this split, the subtrees would be distributed

Finally we'll have —



Algorithms :— If we are searching for value X and are currently at node consisting of values $V_1 \ldots\ldots V_k$, there are four possible cases —

1. If $X < V_1$, recursively search for X in V_1's left subtree
2. If $X > V_k$, recursively search for X in V_k's right subtree

3) If $X = V_i$, for some i, then we are done. (X has been found) {(i is 1 to k)}

4) the only remaining possibility is that for some i, $V_i < X < V_{i+1}$. Here, recursively search for X in the subtree which is in between V_i and V_{i+1} {right of V_i which would be same as left of V_{i+1}}

Algorithm for Insertion into B-Tree

To insert X into B-Tree, follow the following steps:

1. Using the Search Procedure, find the leaf node to which X would be added
2. Add X to this node in the appropriate place (values within a node are in sorted order). Being a leaf node, there are no subtrees to worry about
3. If there are <u>M-1</u> or fewer values in the node after inserting X, then we are done. However, if there are M nodes after adding X, we say that node has overflowed. To correct, we split the node into three parts:

left : first $(M-1)/2$ values

Middle : the middle value position $1 + \frac{(M-1)}{2}$

Right : last $(M-1)/2$ values

left and right have just enough values to be made into inividual nodes. They become left and right child of middle., middle is added in the appropriate place in this node's parent.

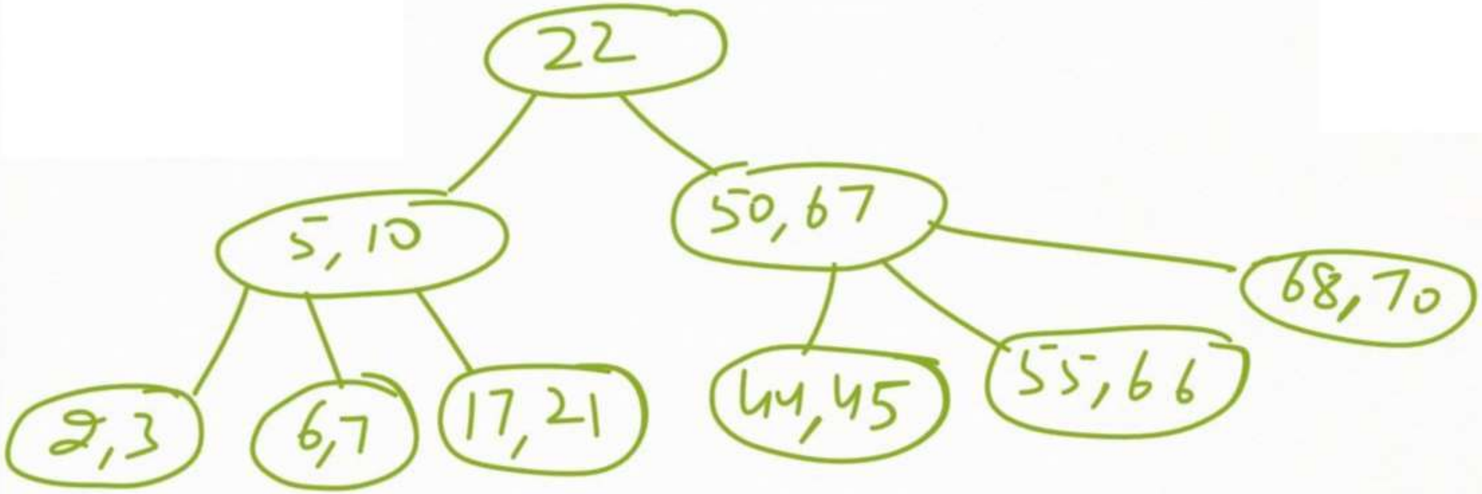
But, if there is no room in the parent? it will overflow & we apply splitting there also. We continue doing until no overflow occurs.

Deleting a value from a B-Tree

deleting algorithm for B.S.T. will be used — if the value to be deleted is in a node having two subtrees, we would replace the value with the largest value in its left subtree and then delete the node in the left subtree that has contained the largest value (this node would be easy to delete)

Here also, similar strategy will be used.

→



Lets delete 67

→ find the largest value in 67's left subtree, it is 66

→ Replace 67 with 66 and then delete the occurrence of 66 in left subtree

⇒ In a B-tree, the largest value in any value's left subtree is guaranteed to be in leaf. Therefore, whenever the value is to be deleted, the following algorithm always begins at a leaf

→ To delete value X from a B-tree, starting at a leaf node, there are 2 steps

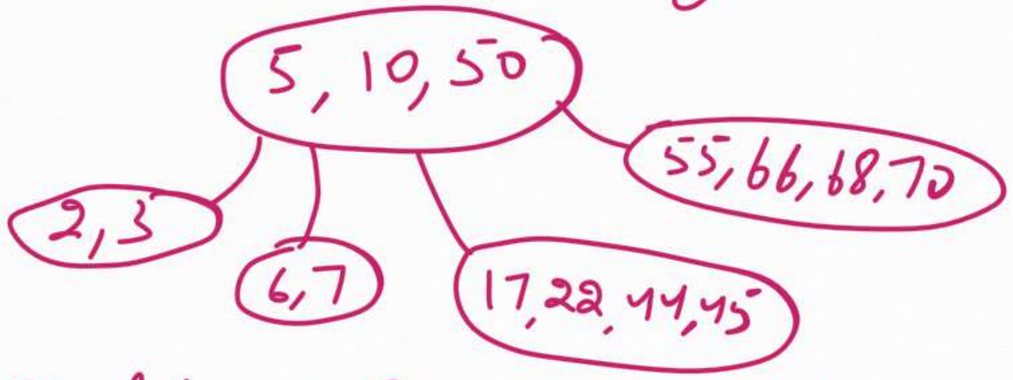
1.) Remove X from the current node. Being a leaf node, there are no subtrees to worry about.

2) Removing X might cause the node containing it to have too few values.

In B-tree, the root need to have atleast 1 value in it and all other nodes to have atleast $(M-1)/2$ values in them. less than these value, we say it has underflowed.

If there was no underflow, we are finished with deletion process. If it does occur, it has to be fixed (corrected).

Say following is some B-tree



B-tree
5-way tree
$M = 5$
→ 5 pointers for child nodes
→ Max. 4 values in each node
(Min each node must have $\frac{M-1}{2} = \frac{5-1}{2} = 2$

Delete 6 from this B-tree

Deleting 6 will result in underflow as it now contains just 1 value.

Strategy for fixing — try to borrow values from neighboring nodes. We join together the current node and its more populous neighbour to form a combined node, and we must also include in the combined node, the value in the parent node that is in between these two nodes

Here, we join node (7) with its more populous neighbour (17, 22, 44, 45) and put '10' in between them to create a combined node

7, 10, 17, 22, 44, 45

How many values might be there in this combined node?

- parent node contributes 1 value
- node that underflowed contributes exactly $\frac{(M-1)}{2} - 1$
- neighbouring node contributes somewhere between $\left(\frac{M-1}{2}\right)$ and $(M-1)$ values.

Treatment of the combined node is different depending on whether the neighbour contributes exactly $\frac{M-1}{2}$ values or more than this number

Case 1 :- Suppose that the neighbouring node contains more than $(M-1)/2$ values. In this case, total number of values in the combined node is strictly greater than $1 + \left(\frac{M-1}{2} - 1 + \frac{M-1}{2} \right)$ i.e. it is greater than $(M-1)$. So it must contain M values or more

We split the node into three pieces

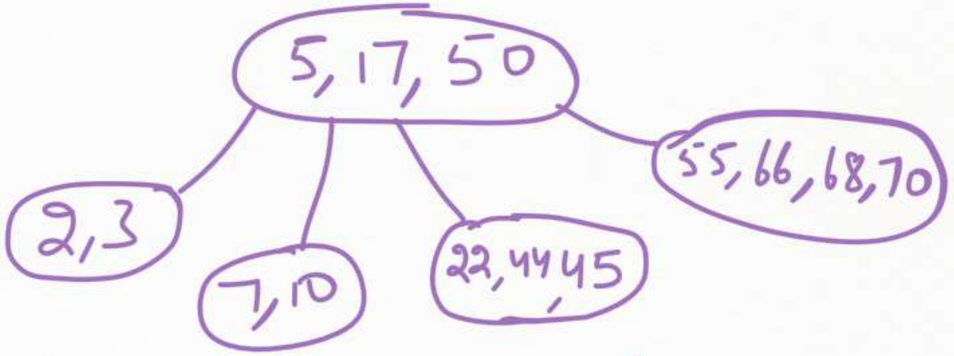
- left
- middle
- right

where middle is a single value in the very middle of the combined node. Because combined node has M values or more, left and right are guaranteed to have $(M-1)/2$ values each and therefore are valid nodes

We replace the value, we borrowed from the parent with the Middle and we use left & right as its two children. In this case, the parent size does not change. With this, the work required in this case will be finished

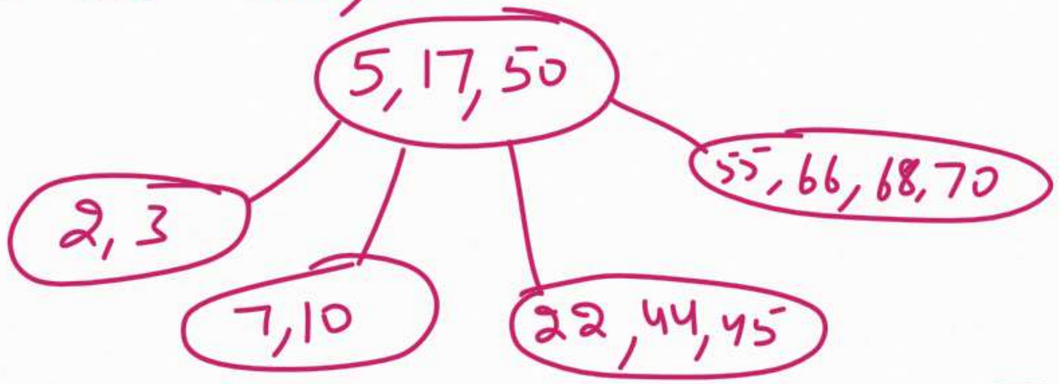
7, 10, 17, 22, 44, 45

left - 7, 10
Middle - 17
Right → 22, 44, 45



<u>Case 2</u> → Suppose, on the other hand, the neighbouring node contains exactly $\frac{M-1}{2}$ values. Then the total values in the combined node would be $1+\left(\frac{M-1}{2}-1\right)+\left(\frac{M-1}{2}\right)=(M-1)$

Here, the combined node contains the right number of values for a valid node. So, we make it into a node and remove from the parent node the value that has been used into the new, combined node. For example

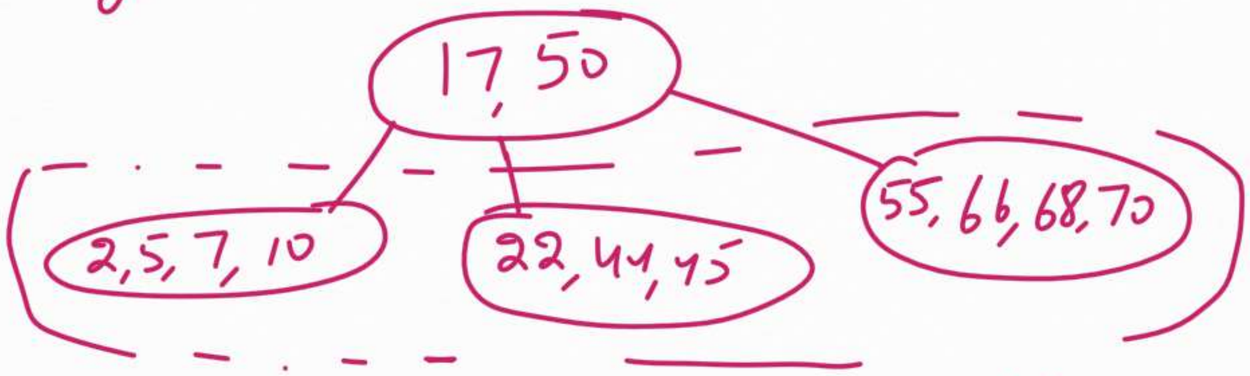


let's delete 3 → the node (2, ~~3~~) will underflow

we make a combined node

2, 5, 7, 10 → the combined contains 4 nodes (i.e. 5-1) so this node is a valid node

of 5-way tree (B-tree)



Now parent has one node less than earlier state.

This might cause it to underflow.

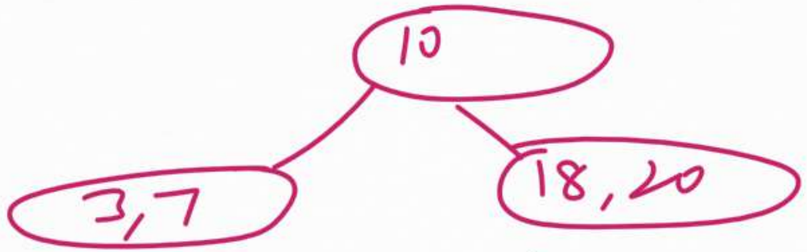
→ If the parent node underflows, it would be treated in exactly the same manner — combine with its more populous neighbour and so one. —

→ The underflow processing repeats at successive levels until no underflow occurs or untill the root underflow.

Lets' look at the root case

Processing at root :- combined node is constructed using two children of root along with the root's only value to form a single node. If the root's two children are now a single node, then that node will become the new root and the current root (underflowed one) can simply be deleted

Example -> for some (5-way Tree)



Say we want to delete 7.

combined

3, 10, 18, 20

it is having (M-1) values

this can itself be a node resulting in deletion of original root node

That's all about B-Tree

HASHING

Problem – We want to design a system for storing student record of all those students who use vehicle in campus and we need to do the following operations efficiently:

- insert a vehicle number and related information
- search a vehicle number and fetch the information
- delete a vehicle number and related information

The option available are →

1) An array of vehicle number and record
2) Balanced Binary Search Tree to the vehicle number as key
3) Direct Access Table

Array:- Searching in linear way will be highly inefficient
→ we can store data in sorted order, $O(\log n)$ will be needed for Searching using Binary Search but insertion and deletion would be costly as sorted order is to be maintained.

Balanced Binary Search Tree:- Best we can do is $O(\log n)$ operation

Direct Access Table :- We make a big array and use the vehicle number as an index in the array. An entry is NIL if there is no record for a particular vehicle number else it refers to the record of the student.

Best Part → Searching, Insertion, Deletion : $O(1)$

Problem → requirement of lot of memory
→ most of it will not have any information.

Solution → Instead of opting for a very large table array, choose a limited size array which is close to the requirement and then employ a method called Hashing to index into the small limited size array. In Hashing, we use a hash function which operates on the key (vehicle number) to generate a small number which can be used as an index in the limited size table array, which is known as hashtable. With hashing we get $O(1)$ search time on average and $O(n)$ in worst case

Hashing – It is a method of converting some kind of data into a relatively smaller number that may serve as a digital "fingerprint" of the data.

Hashing Algorithm (that implements the hash function) processes the data to create such fingerprints, called as hash values. One popular is MD5 hash (takes a string of any length and encodes it into a 128-bit fingerprint). MD5 hashes are used for storing passwords

Hash Values are used as indices into an array known as hash table.

Indexing into an array require only O(1) time, thus if we have a fast hashing algorithm to generate the index corresponding to data (keys + values), then we can have constant time for the basic operations in the table.

Hashtable consist of two major components
- Bucket Array
- Hashfunction

Bucket array is used for storing the data (key-value entries) according to their computed indices

Hashfunction maps keys to integers in a fixed interval $[0, N-1]$. Example of a very basic hashfunction is <u>modulo</u> operator

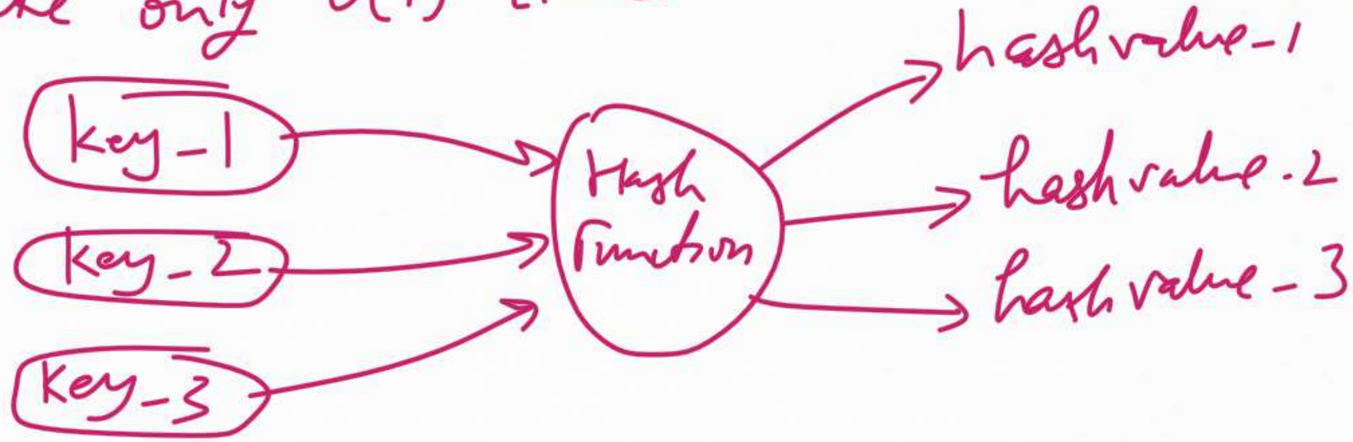
$$h(x) = x \bmod N$$

- $h(x)$: an integer called hashvalue of key x
- x: for integer keys
- mod: hashfunction
- N: size of hashtable

An entry e with a key k is simply inserted into the bucket array $A[h(k)]$

If each $h(k)$ returns a unique integer in the range $[0, N-1]$, then bucket hold at most one entry

In such a case, insertion, searching and deletion would take only $O(1)$ time.



(mod is giving as remainder)

Say we want to use a Table (array) of size 20.

Key	Hash value	Array Index
1	1 mod 20 = 1	1
	(1 % 20)	
2	2 mod 20 = 2	2 ←
4	4 mod 20 = 4	4
42	42 mod 20 = 2	2 ←

Known as Conflict Situation

Hash function → Polynomial hash codes
→ Summation Hash codes
→ Cyclic Shift hash codes

MD5 — was used earlier

SHA256 — now it is more popular

used for verifying the correctness of downloaded files

generate checksum of the downloaded ISO file and verify that it matches with the checksum provided by the organization which is distributing the ISO files.