Unit 5: Application of Trees

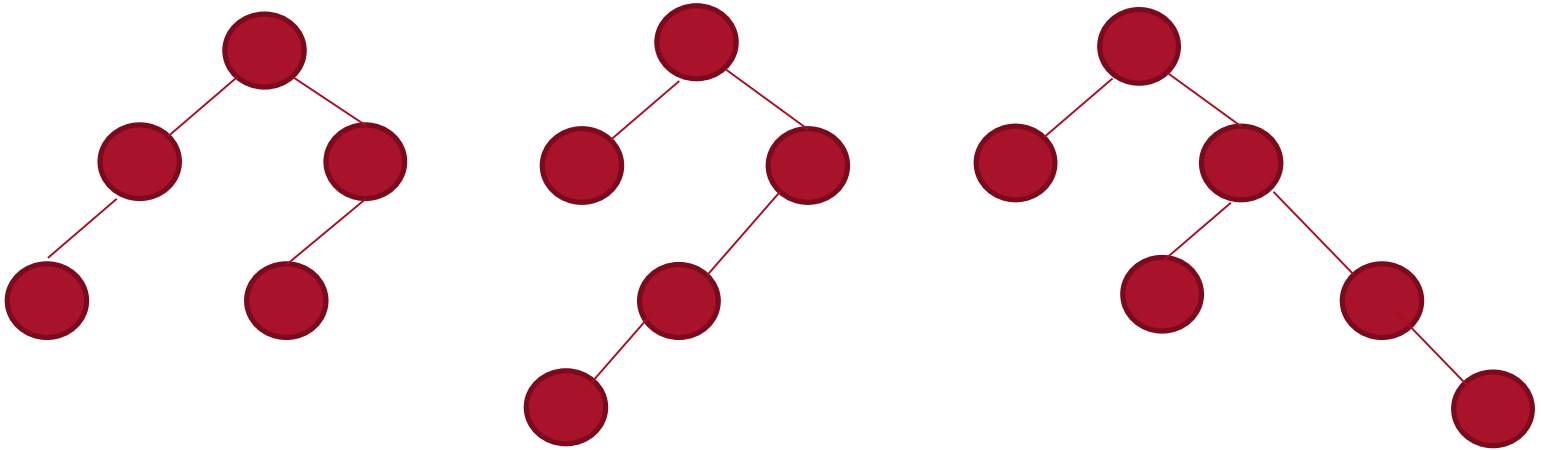
Prepared By:
Tanvi Patel
Asst. Prof. (CSE)

Contents

- Some balanced tree mechanism;
- e.g. Heap, AVL trees;
- 2-3 trees;
- Height Balanced, Weight Balance, Red black tree;
- Multi-way search tree: B and B+ tree;

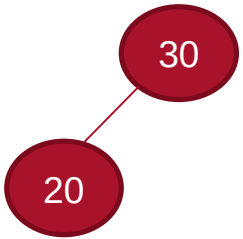
AVL Trees

- AVL Trees are height balanced BST.
- Balance Factor = height of left subtree – height of right subtree
- bf=hl-hr = {-1,0,1}
- |bf| = |hl-hr| <=1

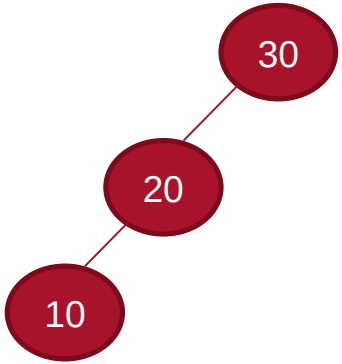


AVL Trees

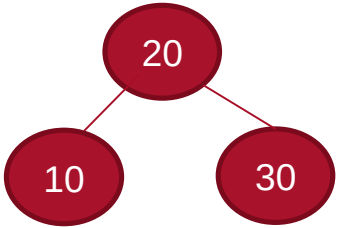
Initially



Insert 10



After Rotation

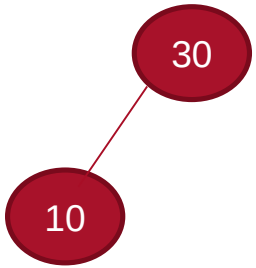


Why imbalance? Because we inserted left of left. So, it is called LL-imbalance.

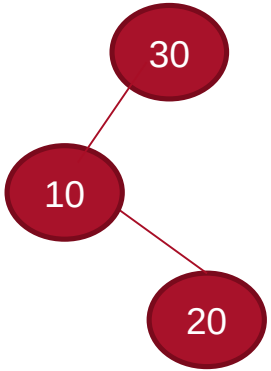
This rotation is called as LL- Rotation

AVL Trees

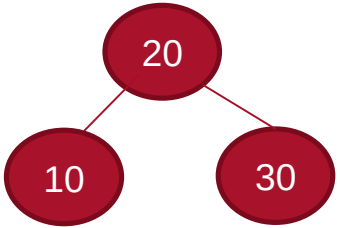
Initially



Insert 20



After Rotation

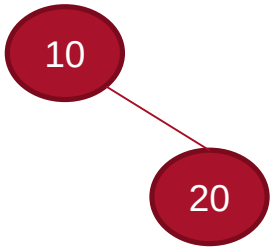


Why imbalance? Because we inserted left and then right. So, it is called LR-imbalance.

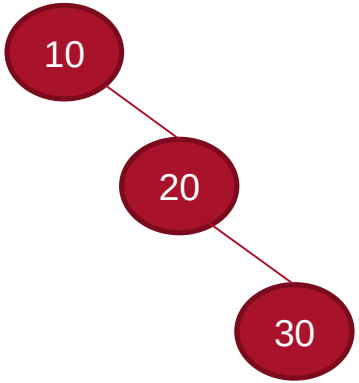
This rotation is called as LR- Rotation

AVL Trees

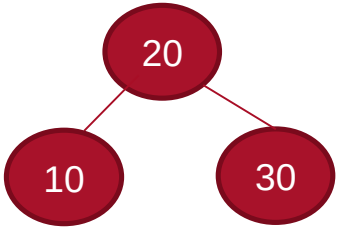
Initially



Insert 30



After Rotation

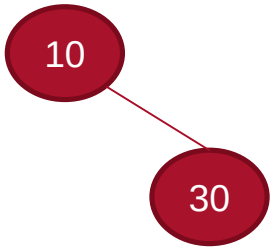


Why imbalance? Because we inserted right of right. So, it is called RR-imbalance.

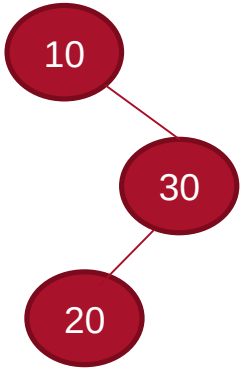
This rotation is called as RR- Rotation

AVL Trees

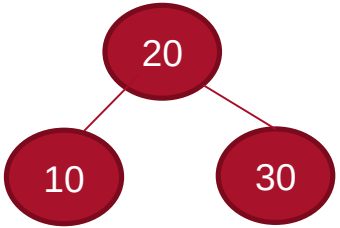
Initially



Insert 20



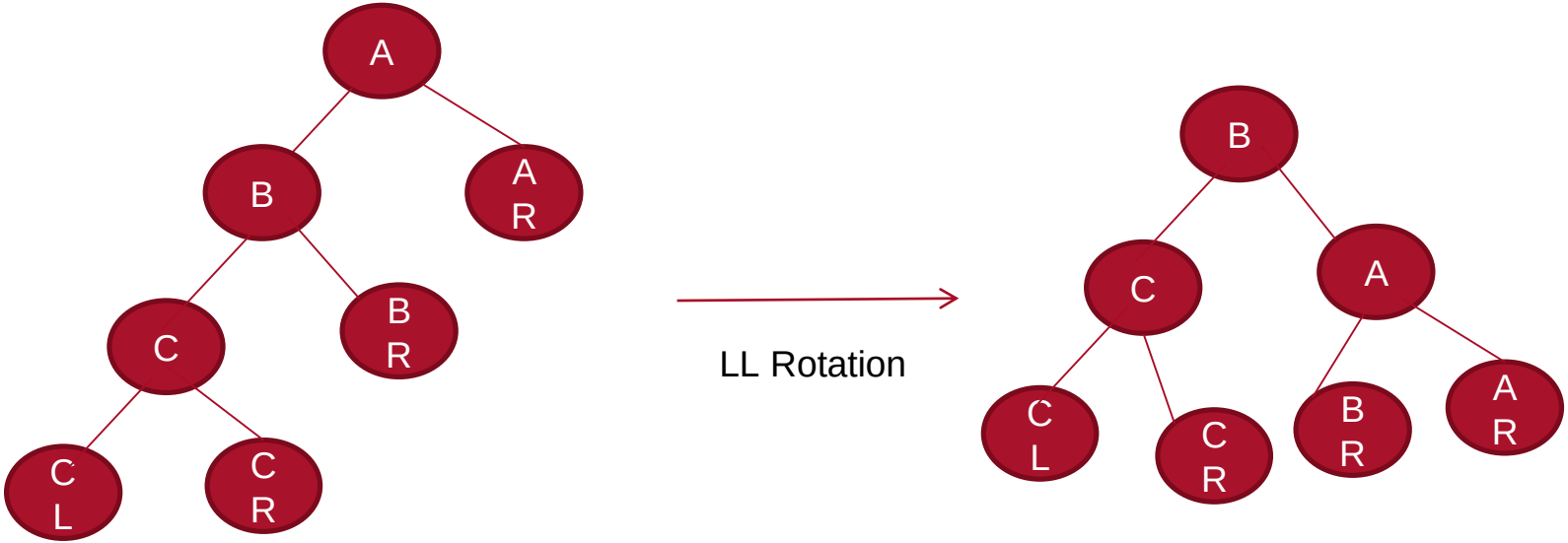
After Rotation



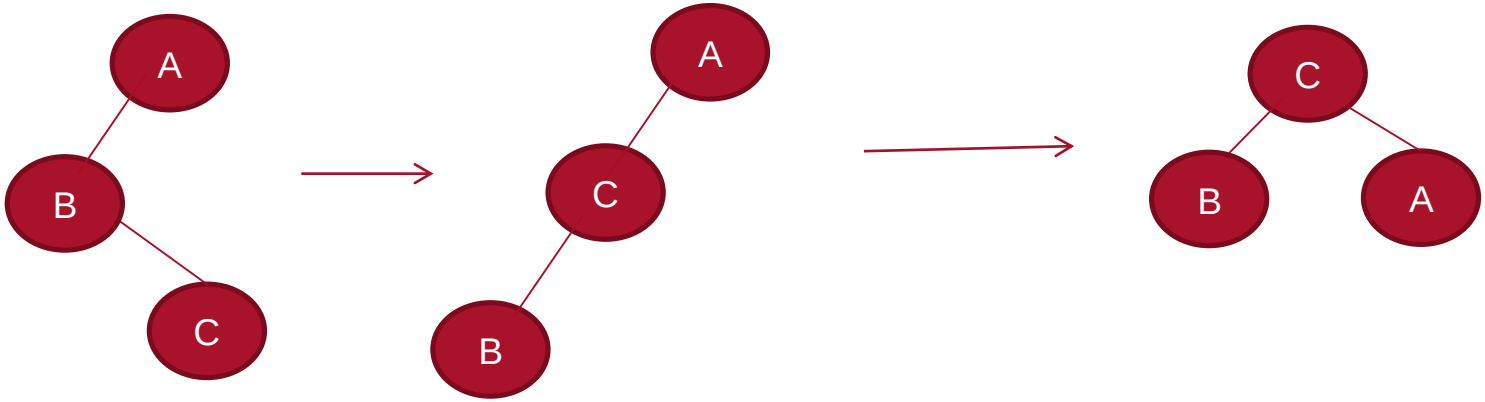
Why imbalance? Because we inserted right of left. So, it is called RL-imbalance.

This rotation is called as RL- Rotation

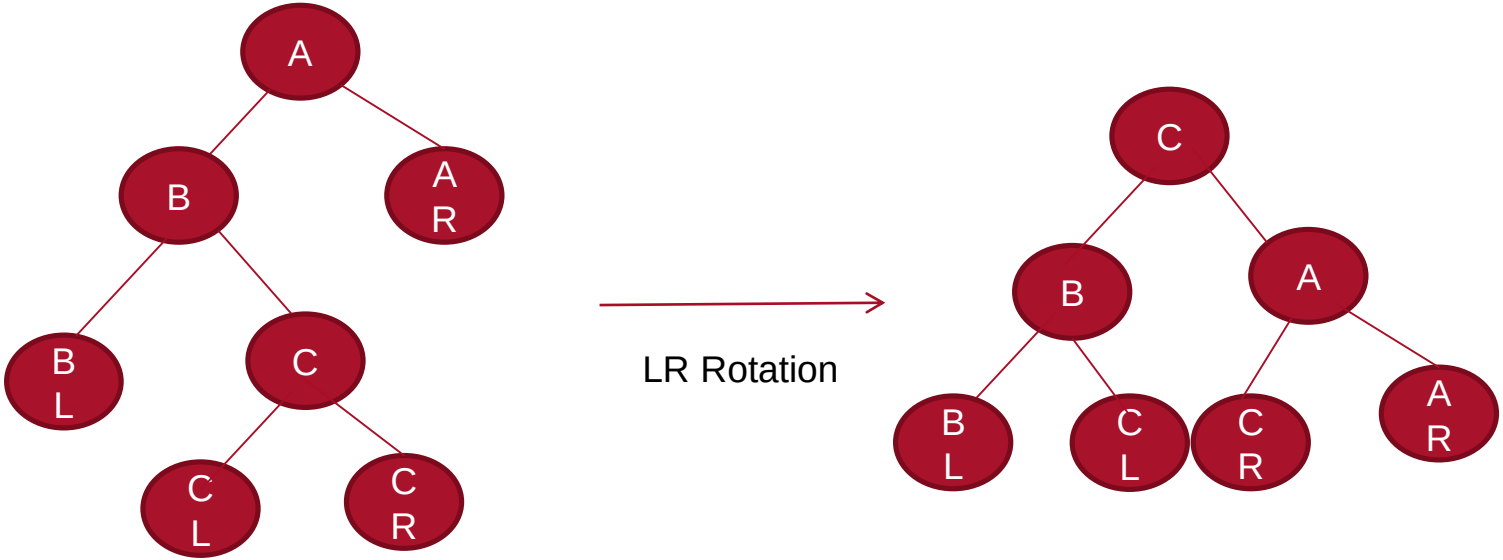
AVL Trees (Some Special case)



AVL Trees (Some Special case)



AVL Trees (Some Special case)



AVL Tree Example

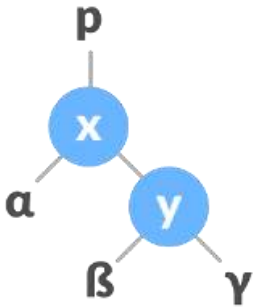
Rotating the subtrees in an AVL Tree

- In rotation operation, the positions of the nodes of a subtree are interchanged.
- There are two types of rotations:

Left Rotate

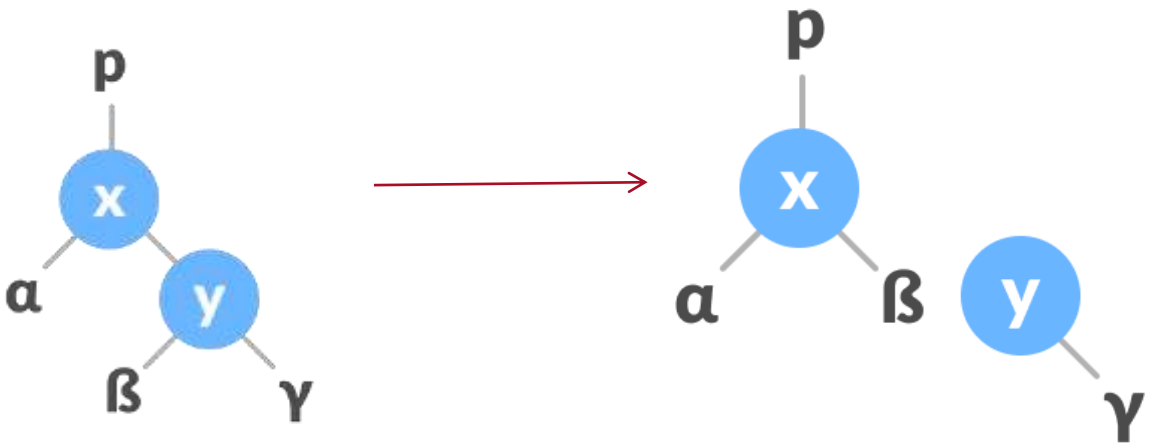
- In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.
- Algorithm

1. Let the initial tree be



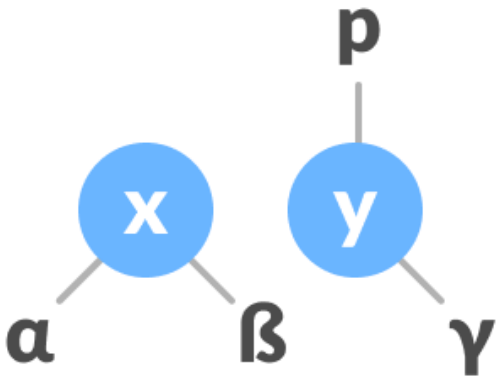
AVL Tree Example

2. If y has a left subtree, assign x as the parent of the left subtree of y



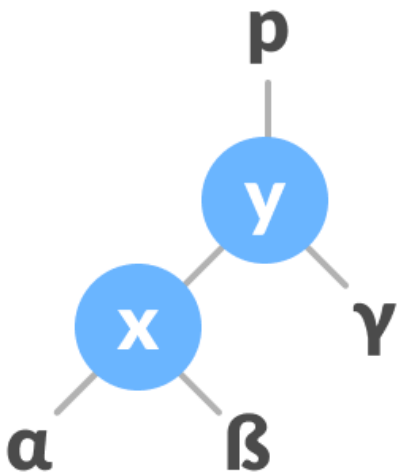
AVL Tree Example

3. If the parent of x is NULL, make y as the root of the tree.
4. Else if x is the left child of p, make y as the left child of p.
5. Else assign y as the right child of p



AVL Tree Example

6. Make y as the parent of x.

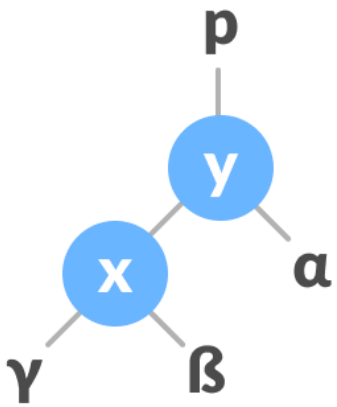


AVL Tree Example

Right Rotate

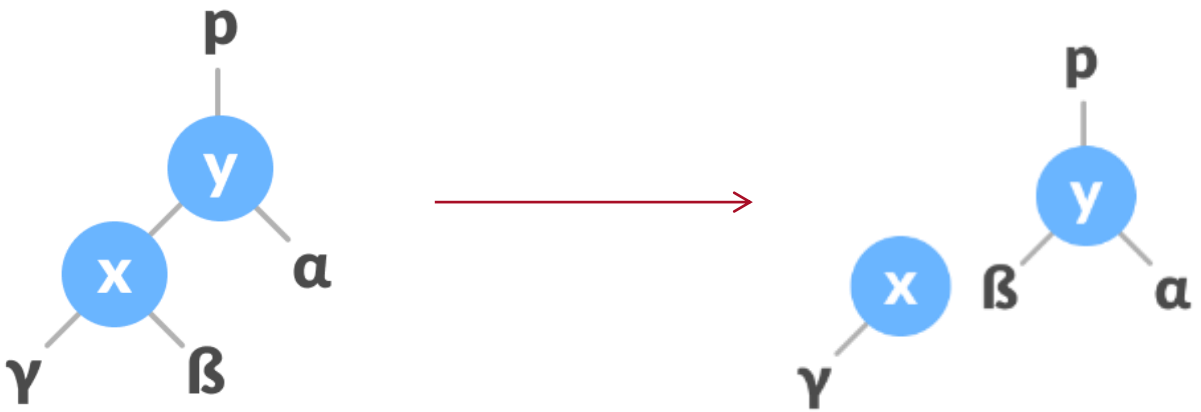
- In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



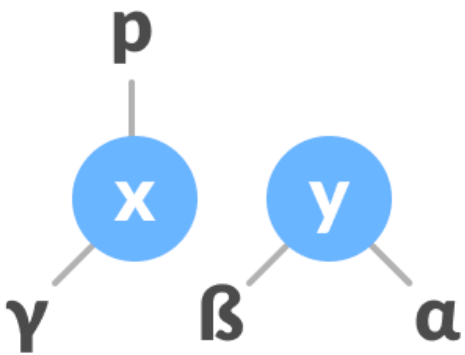
AVL Tree Example

- If x has a right subtree, assign y as the parent of the right subtree of x.



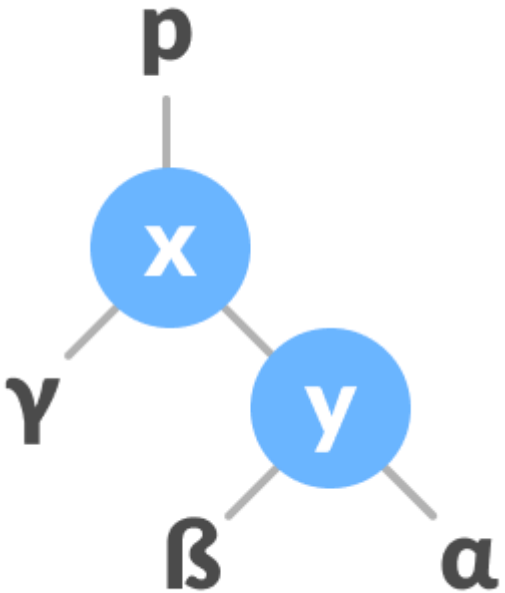
AVL Tree Example

- If the parent of y is NULL, make x as the root of the tree.
- Else if y is the right child of its parent p, make x as the right child of p.
- Else assign x as the left child of p.



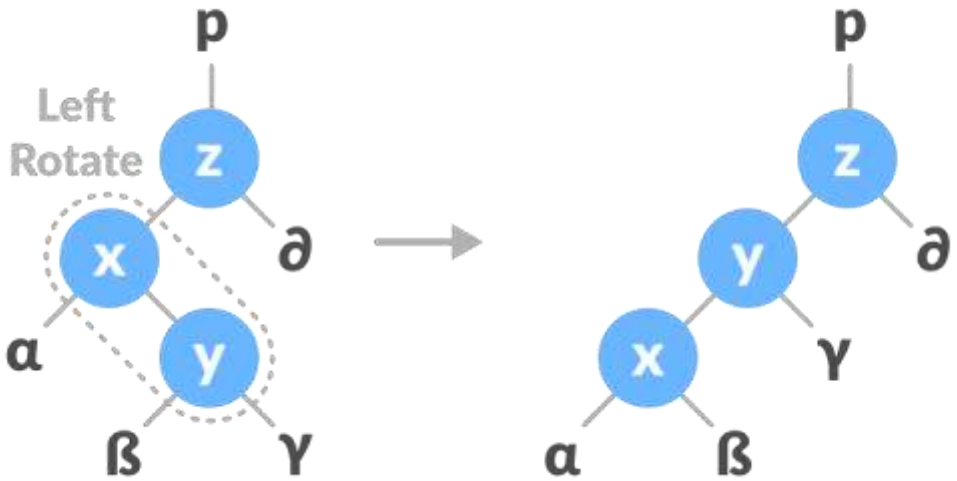
AVL Tree Example

- Make x as the parent of y



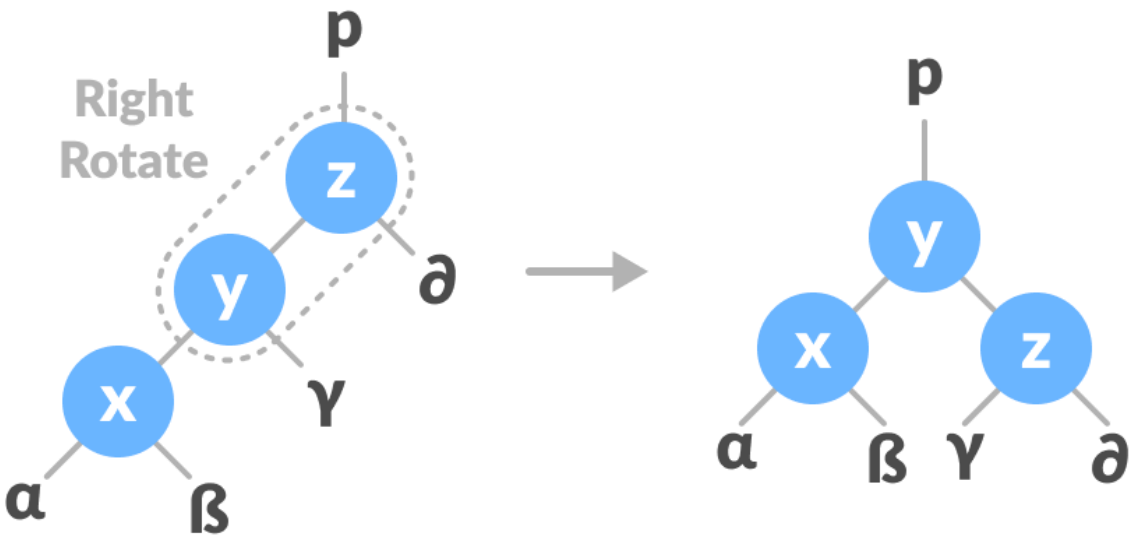
AVL Tree Example

- **Left-Right and Right-Left Rotate**
- In left-right rotation, the arrangements are first shifted to the left and then to the right.
- Do left rotation on x-y.



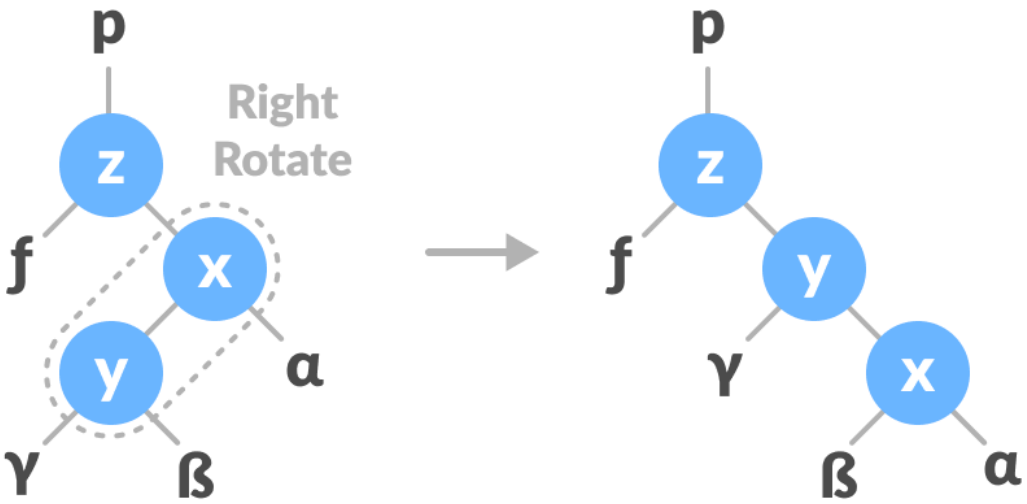
AVL Tree Example

- Do right rotation on y-z



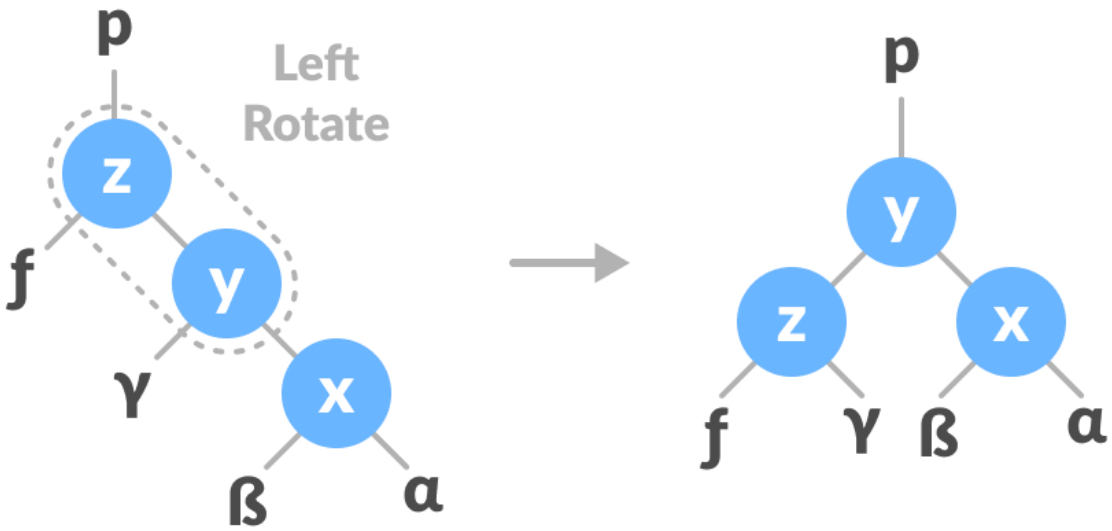
AVL Tree Example

- In right-left rotation, the arrangements are first shifted to the right and then to the left.
- Do right rotation on x-y.



AVL Tree Example

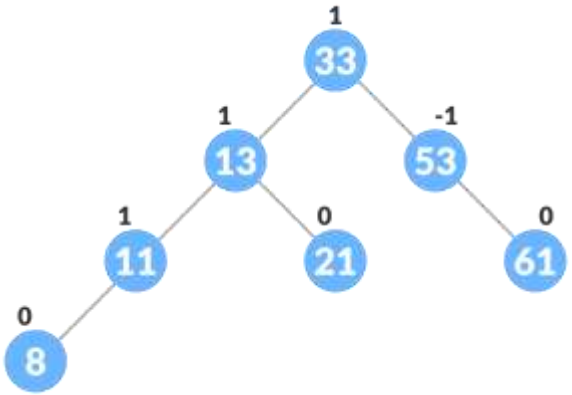
- Do left rotation on z-y.



AVL Tree Example

Algorithm to insert a newNode

- A newNode is always inserted as a leaf node with balance factor equal to 0.
- Let the initial tree be:

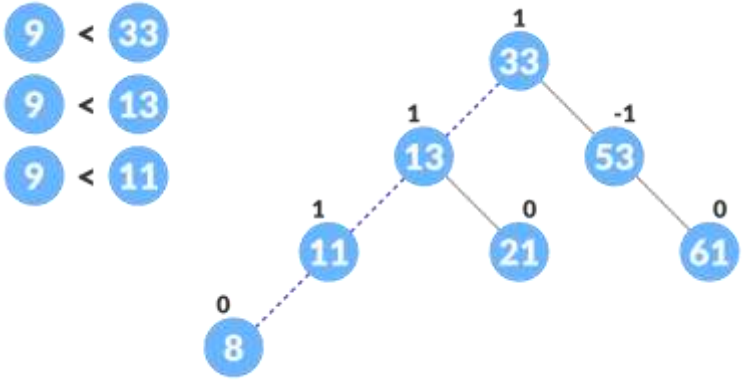


AVL Tree Example

- Let the node to be inserted be:

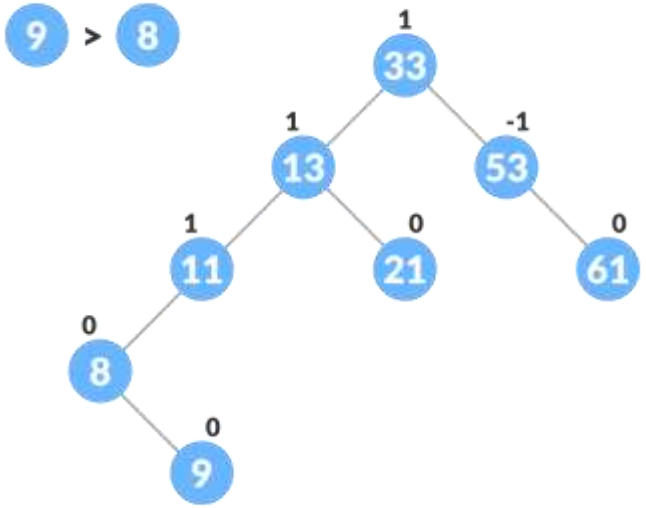
AVL Tree Example

- Go to the appropriate leaf node to insert a newNode using the following recursive steps.
  Compare newKey with rootKey of the current tree.
  a. If newKey < rootKey, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
  b. Else if newKey > rootKey, call insertion algorithm on the right subtree of current node until the leaf node is reached.
  c. Else, return leafNode.



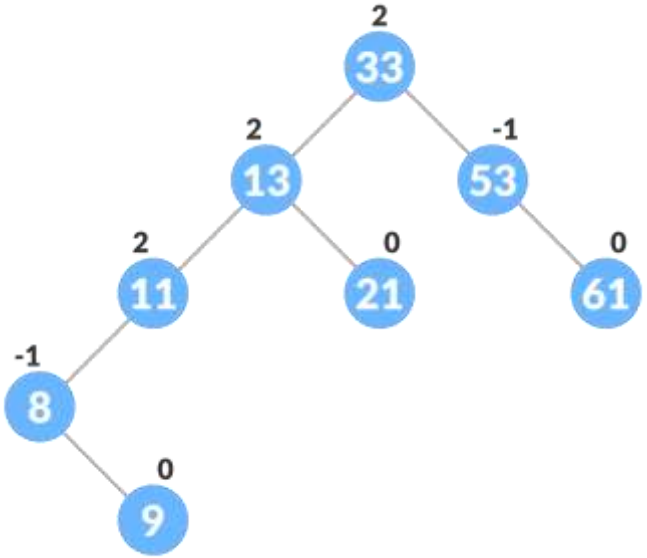
AVL Tree Example

- Compare leafKey obtained from the above steps with newKey:
  a. If newKey < leafKey, make newNode as the leftChild of leafNode.
  b. Else, make newNode as rightChild of leafNode



AVL Tree Example

- Update balanceFactor of the nodes.



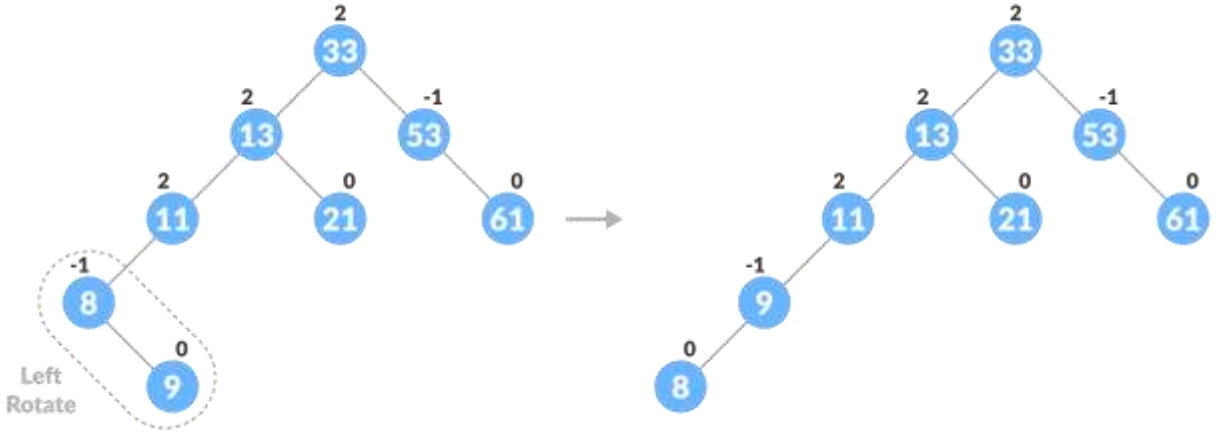
AVL Tree Example

- If the nodes are unbalanced, then rebalance the node.

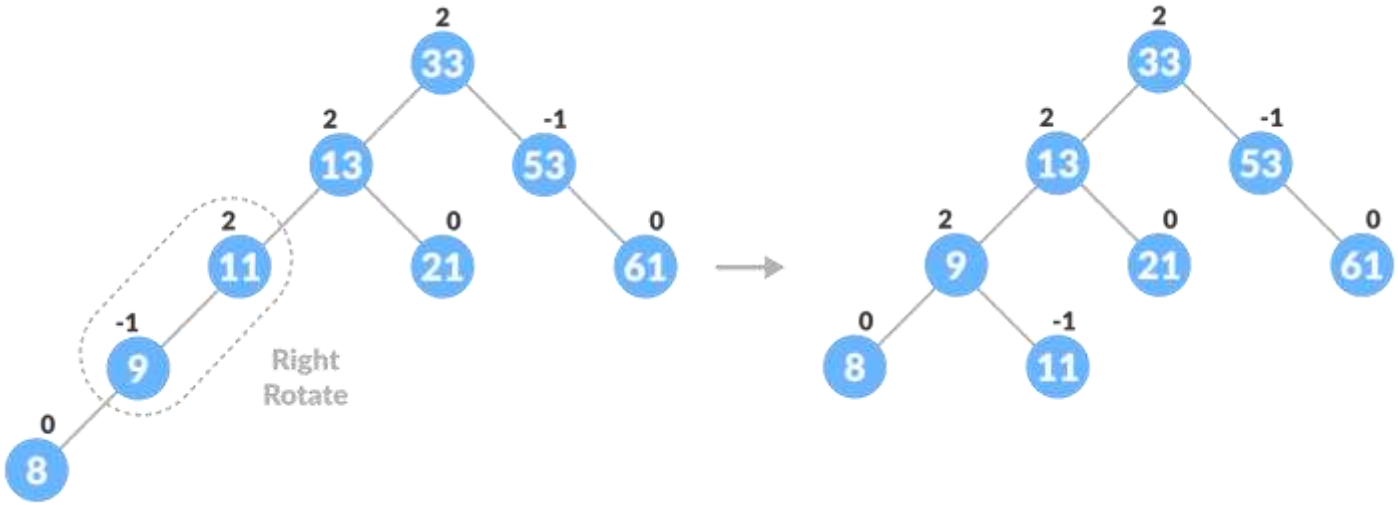
  a. If balanceFactor > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

    i. If newNodeKey < leftChildKey do right rotation.

    ii. Else, do left-right rotation.



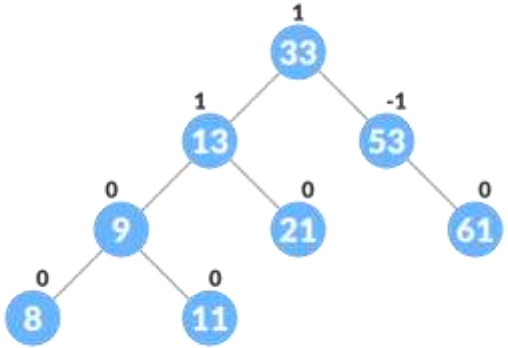
AVL Tree Example



AVL Tree Example

b. If balanceFactor < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation
   i. If newNodeKey > rightChildKey do left rotation.
   ii. Else, do right-left rotation

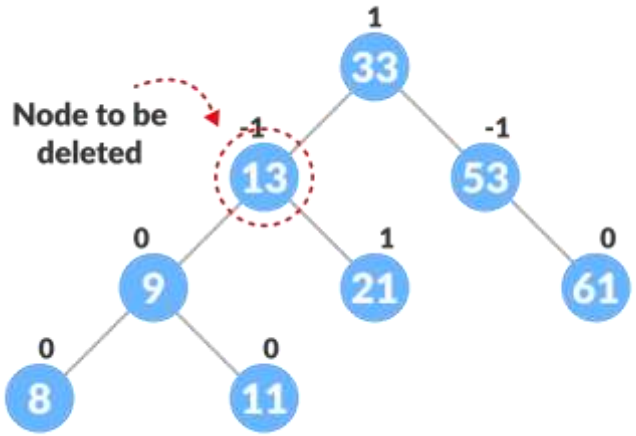
- The final tree is



AVL Tree Example

Algorithm to Delete a node

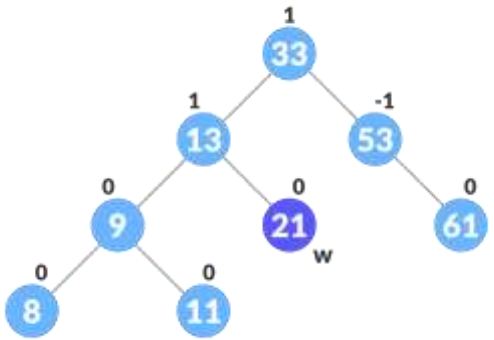
- A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.
- Locate nodeToBeDeleted (recursion is used to find nodeToBeDeleted).



AVL Tree Example

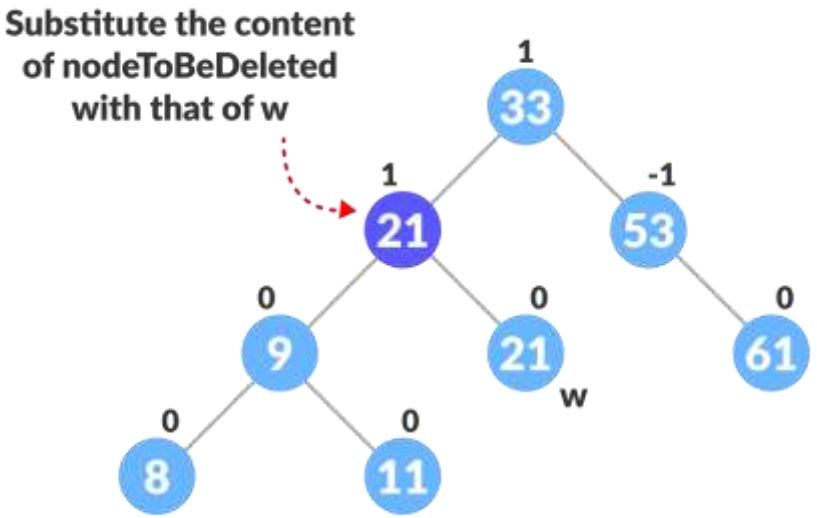
There are three cases for deleting a node:

- If nodeToBeDeleted is the leaf node (ie. does not have any child), then remove nodeToBeDeleted.
- If nodeToBeDeleted has one child, then substitute the contents of nodeToBeDeleted with that of the child. Remove the child.
- If nodeToBeDeleted has two children, find the inorder successor w of nodeToBeDeleted (ie. node with a minimum value of key in the right subtree).



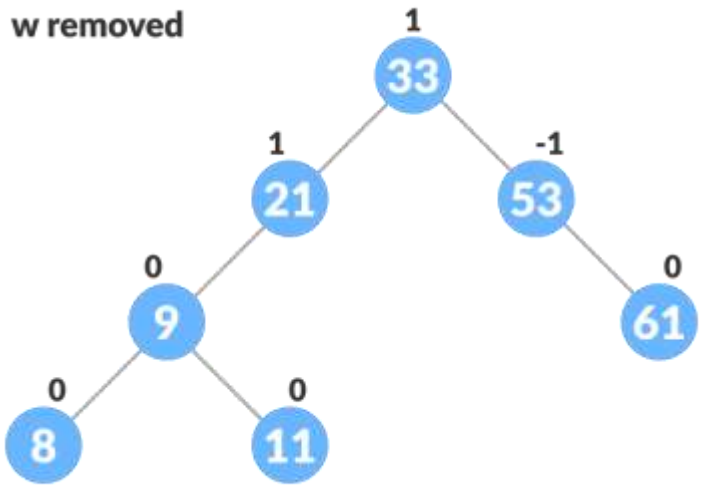
AVL Tree Example

a. Substitute the contents of nodeToBeDeleted with that of w



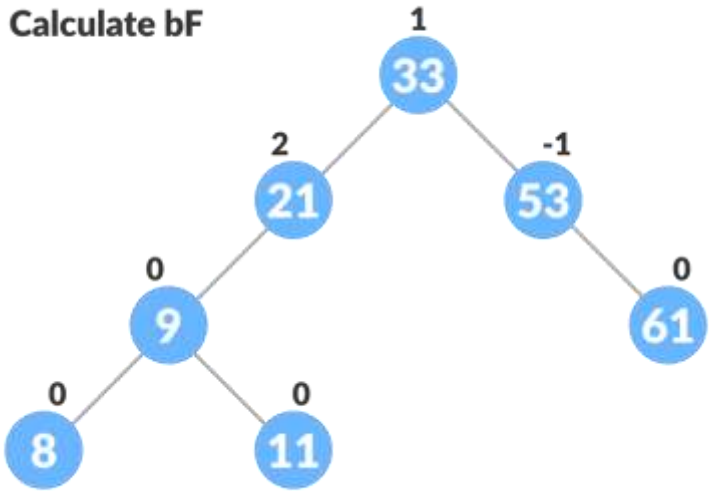
AVL Tree Example

b. Remove the leaf node w



AVL Tree Example

Update balanceFactor of the nodes.



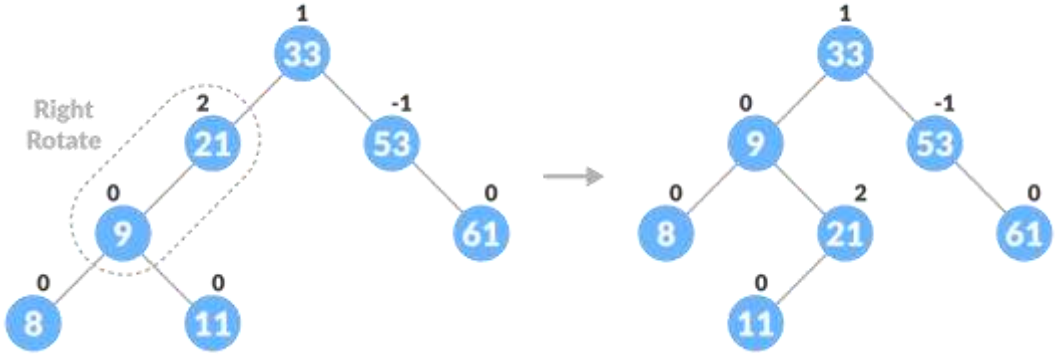
AVL Tree Example

- Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.

  a. If balanceFactor of currentNode > 1,

  i. If balanceFactor of leftChild >= 0, do right rotation

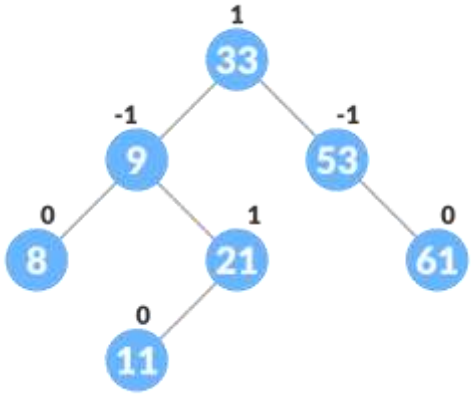
  ii. Else do left-right rotation.



AVL Tree Example

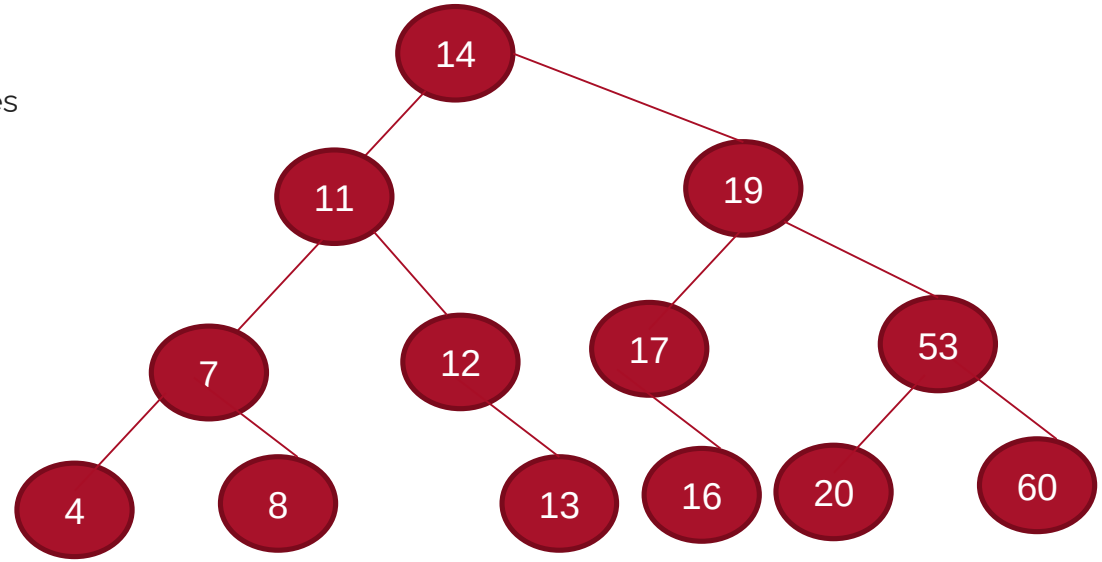
b. If balanceFactor of currentNode < -1,
    i. If balanceFactor of rightChild <= 0, do left rotation.
    ii. Else do right-left rotation.

The final tree is



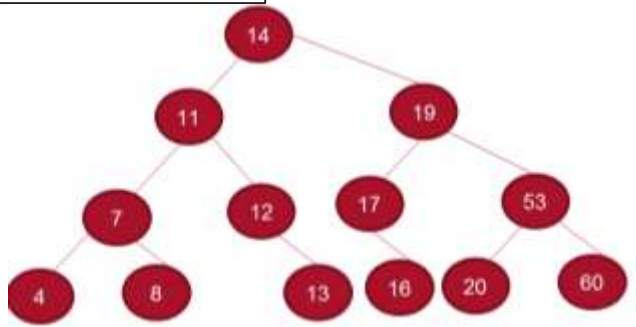
AVL Tree Example

- Deletion in AVL Trees



AVL Tree Example

- Keys to be deleted: 8,7,11,14,17



AVL Tree Example

Complexities of Different Operations on an AVL Tree

Insertion	Deletion	Search
O(log n)	O(log n)	O(log n)

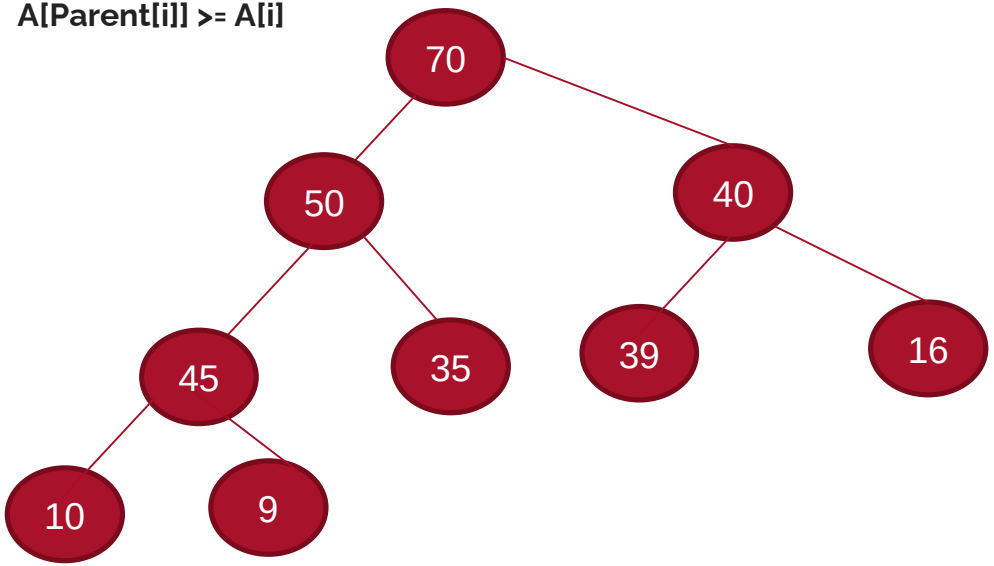
AVL Tree Applications

- For indexing large records in databases
- For searching in large databases

Heap

Max Heap: For every node i, the value of node is less than or equal to its parent value.

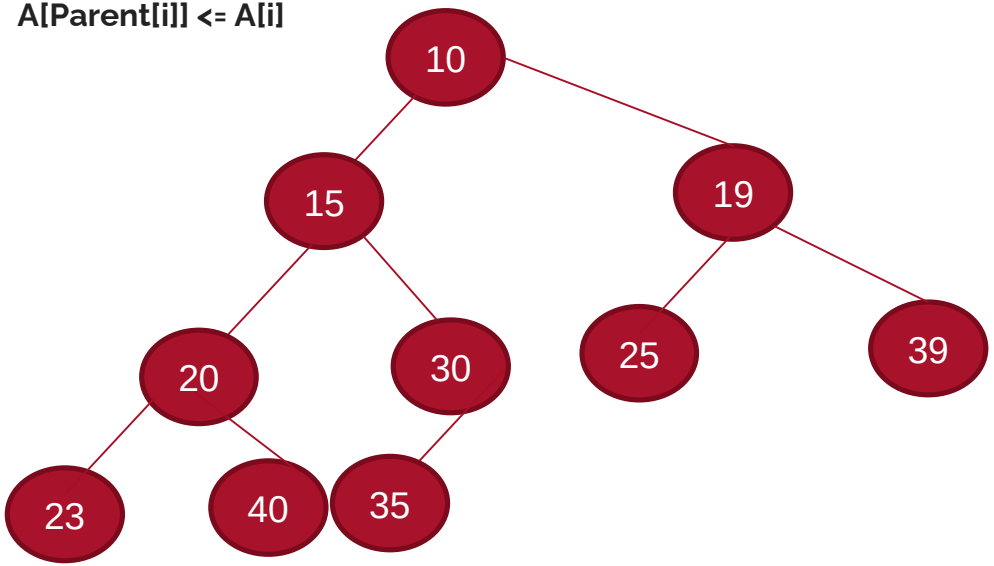
A[Parent[i]] >= A[i]



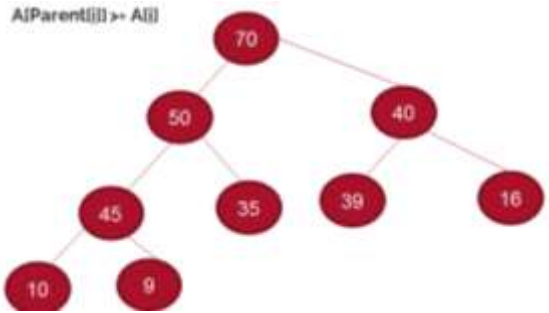
Heap

Min Heap: For every node i, the value of node is greater than or equal to its parent value.

A[Parent[i]] <= A[i]



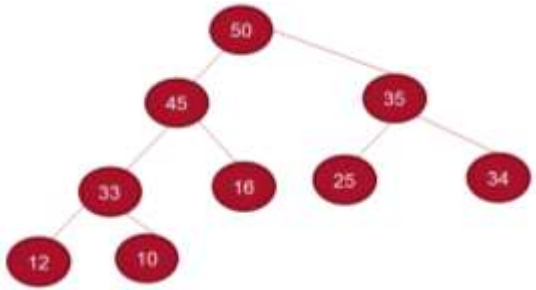
Heap Insertion



70	50	40	45	35	39	6	10	9
0	1	2	3	4	5	6	7	8

- Parent node=floor((i-1)/2)

Heap Deletion

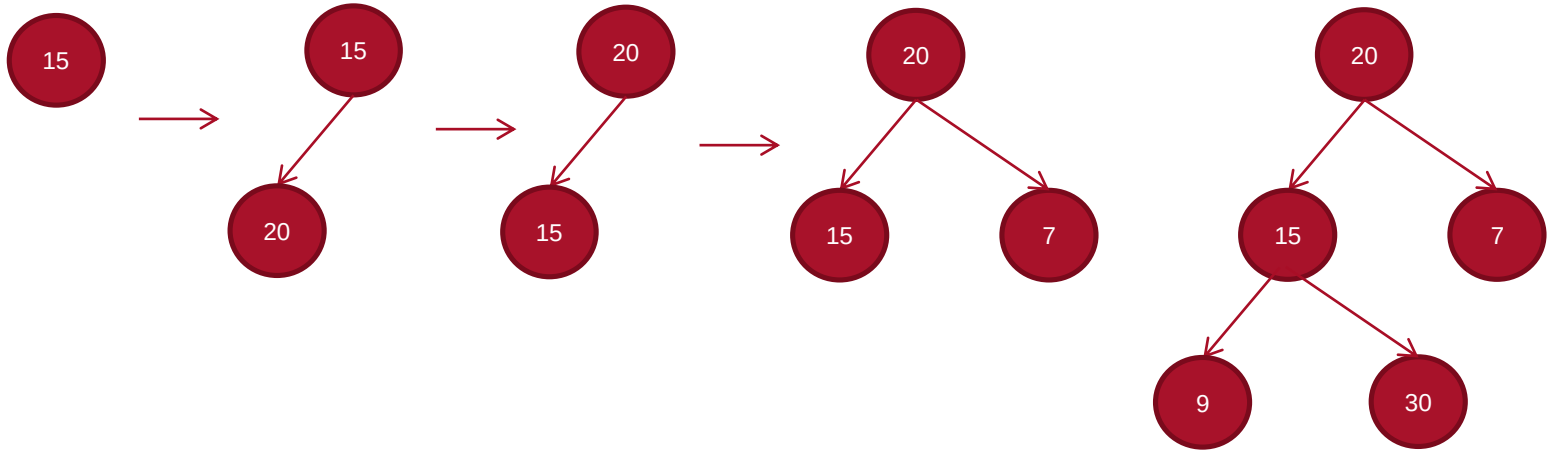


50	45	35	33	16	25	34	12	10
0	1	2	3	4	5	6	7	8

- Left child = 2i+1
- Right chid = 2i+2

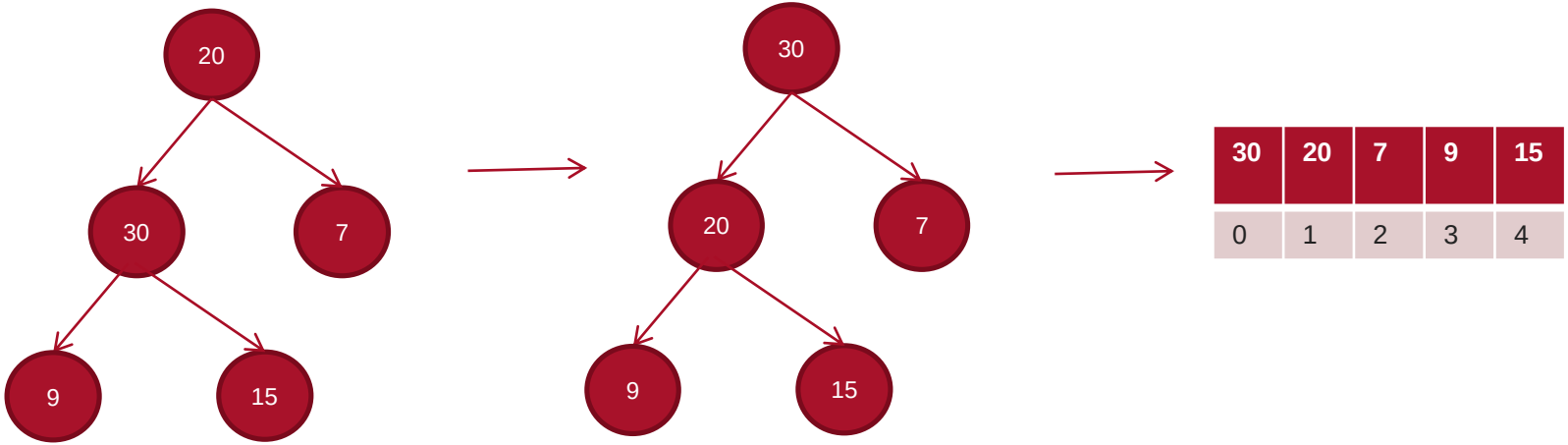
Building a Heap

15	20	7	9	30



Building a Heap

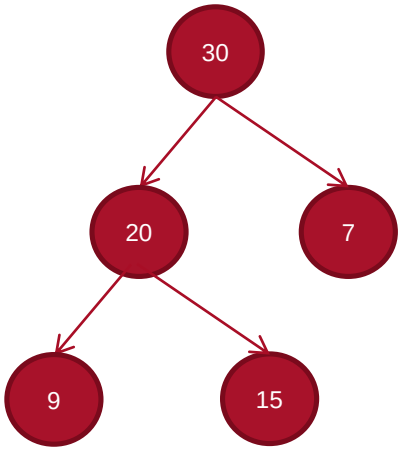
15	20	7	9	30



Heap sort

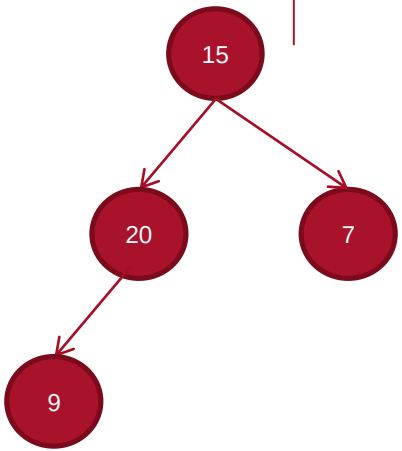
- Build the max heap and delete the node one by one. So, it build heap sort.
-

30	20	7	9	15
0	1	2	3	4

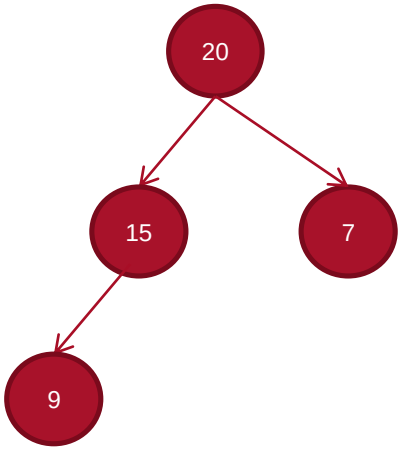


Delete 30

15	20	7	9	30
0	1	2	3	4

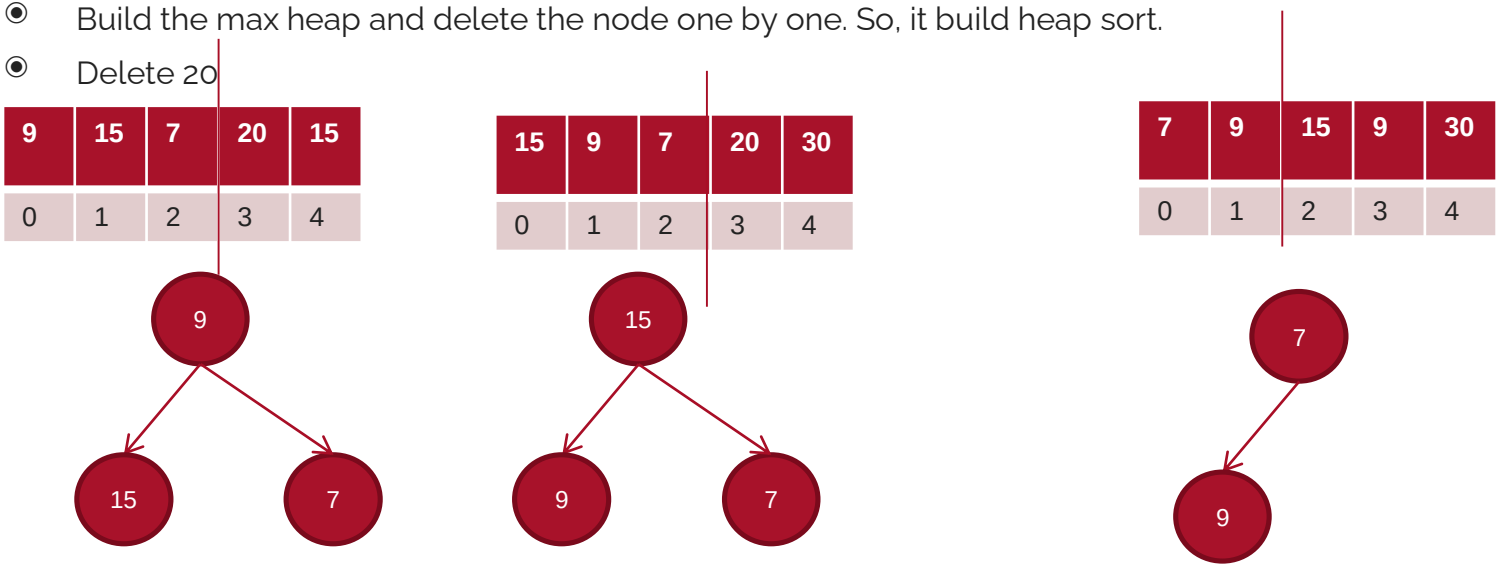


20	15	7	9	30
0	1	2	3	4



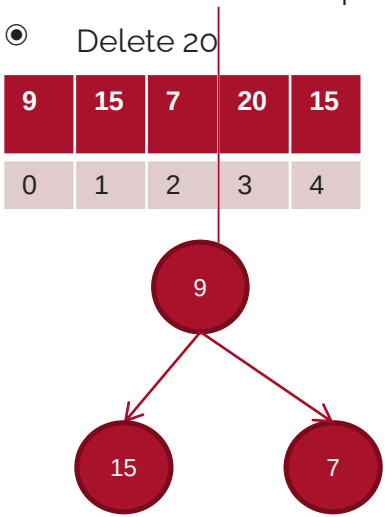
Heap sort

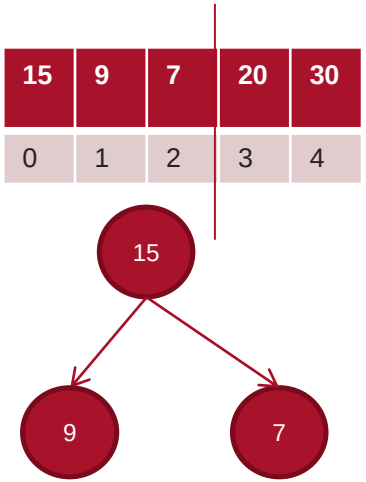
- Build the max heap and delete the node one by one. So, it build heap sort.
- Delete 20

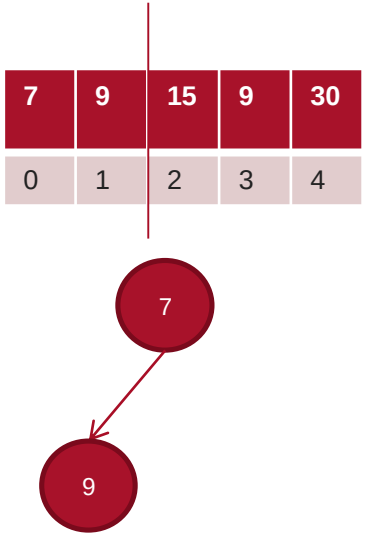


Heap sort

- Build the max heap and delete the node one by one. So, it build heap sort.
- Delete 20

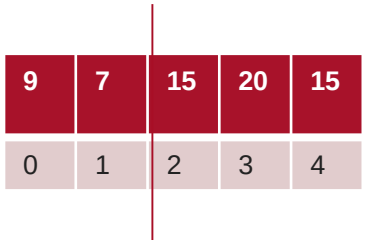


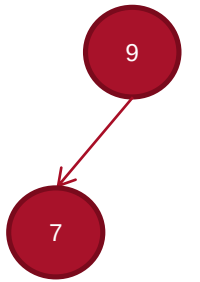


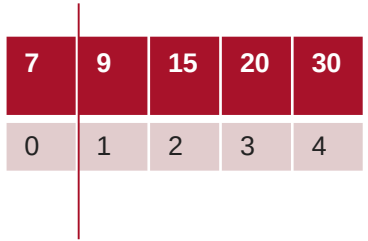


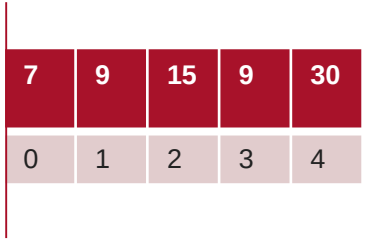
Heap sort

- Build the max heap and delete the node one by one. So, it build heap sort.









Red Black Tree

- A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the colour (red or black).
- These colours are used to ensure that the tree remains balanced during insertions and deletions.
- Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree. This tree was invented in 1972 by Rudolf Bayer.
- It must be noted that as each node requires only 1 bit of space to store the colour information.

Red Black Tree

Why Red-Black Trees?

- Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST.
- The cost of these operations may become O(n) for a skewed Binary tree.
- If we make sure that the height of the tree remains O(log n) after every insertion and deletion, then we can guarantee an upper bound of O(log n) for all these operations.
- The height of a Red-Black tree is always O(log n) where n is the number of nodes in the tree.

Sr. No.	Algorithm	Time Complexity
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Red Black Tree

Comparison with AVL Tree:

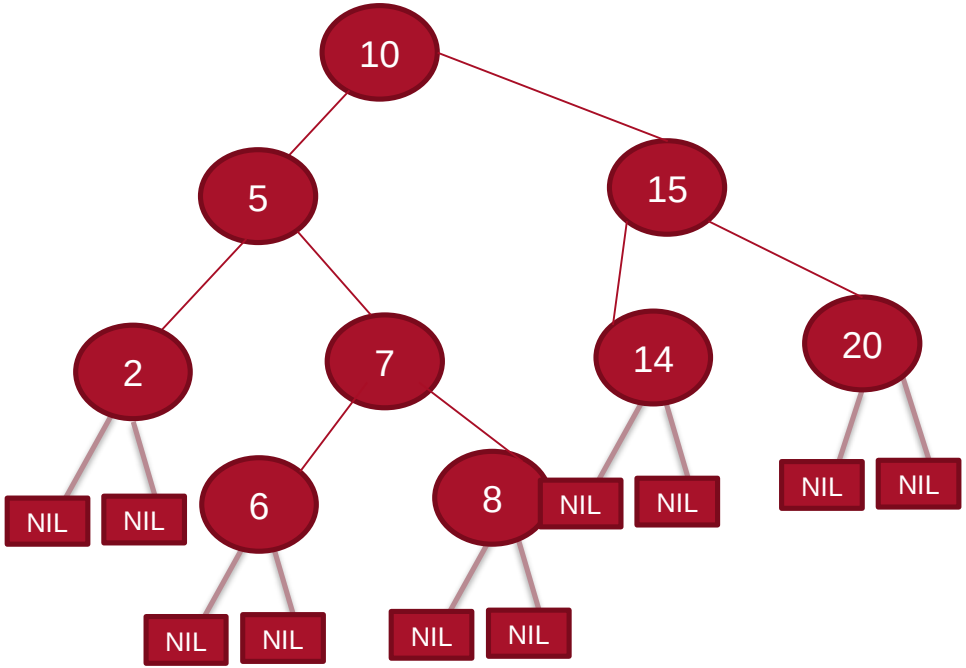
- The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion.
- So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred. And if the insertions and deletions are less frequent and search is a more frequent operation, then AVL tree should be preferred over Red-Black Tree.

Red Black Tree

Properties of Red Black Tree

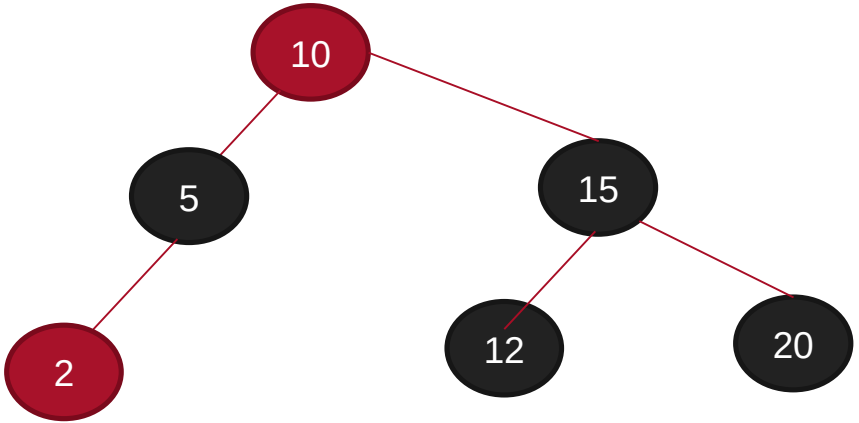
- It is a self balancing BST.
- Every node is either Black or Red.
- Root is always Black.
- Every leaf which is NIL is black.
- If node is Red then its children are Black.
- Ever path from a node to any of its nodes descendent NIL node has same no. of black nodes.

Red Black Tree



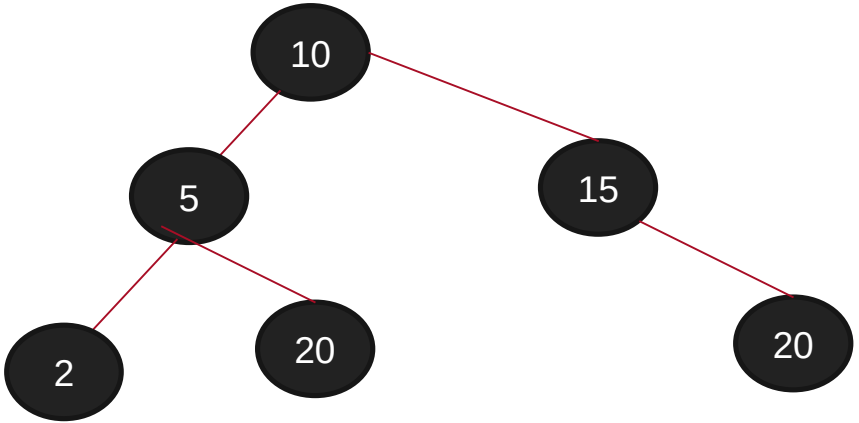
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



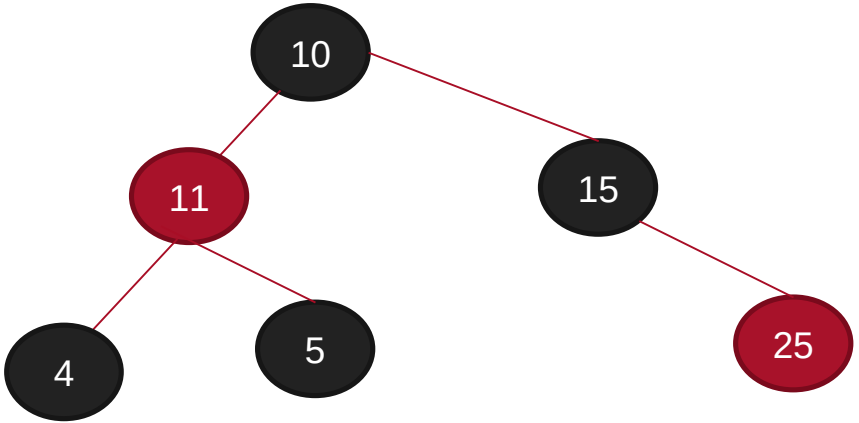
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



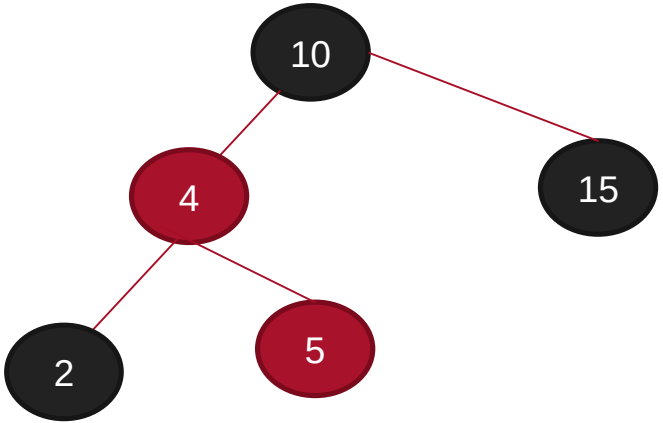
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



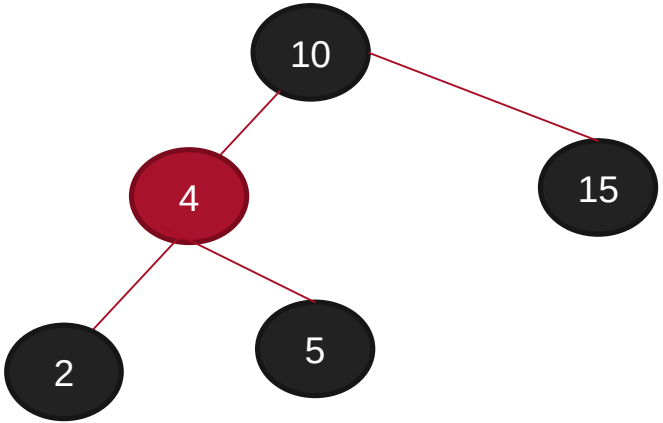
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



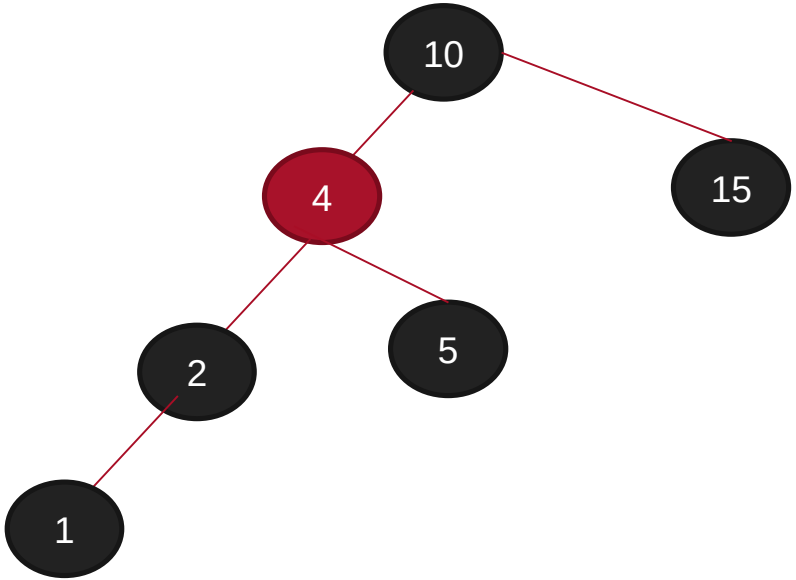
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



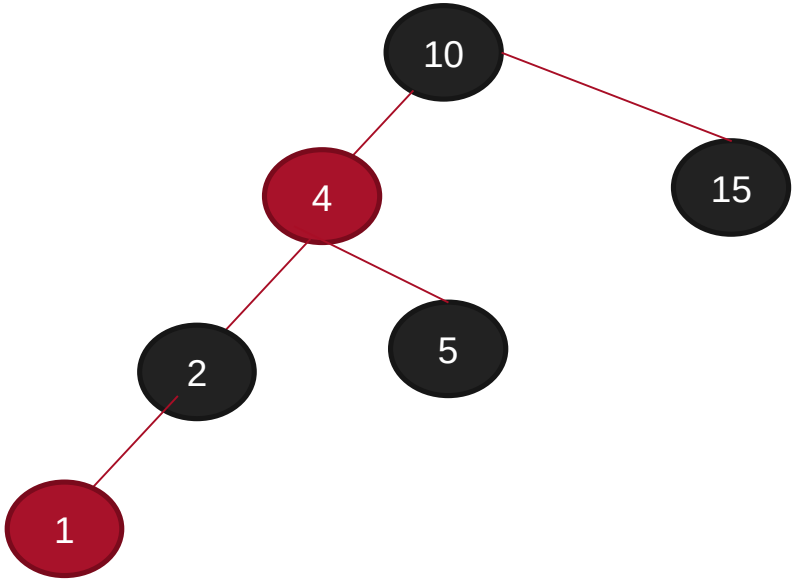
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



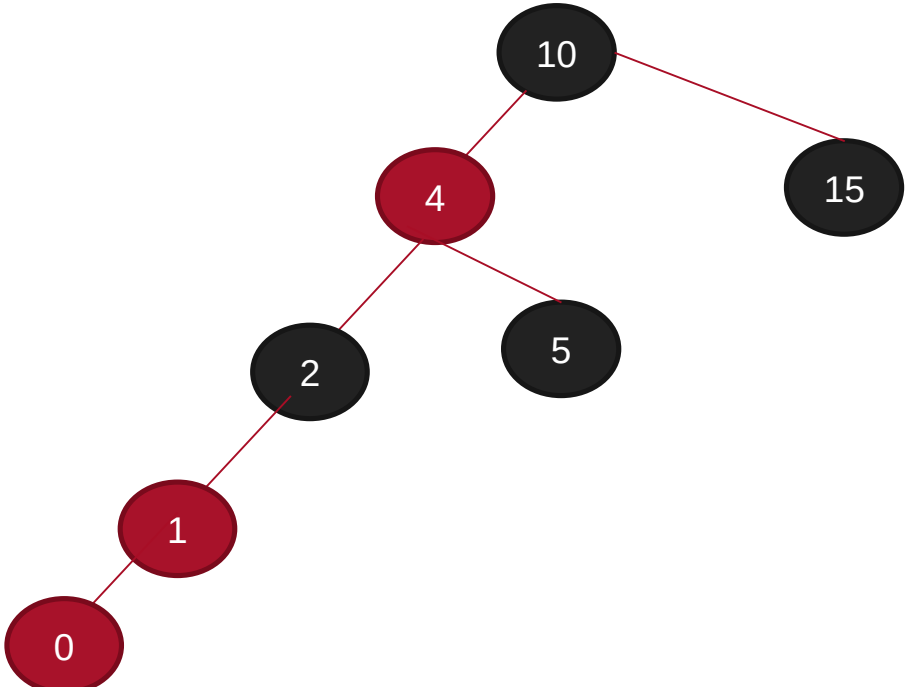
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



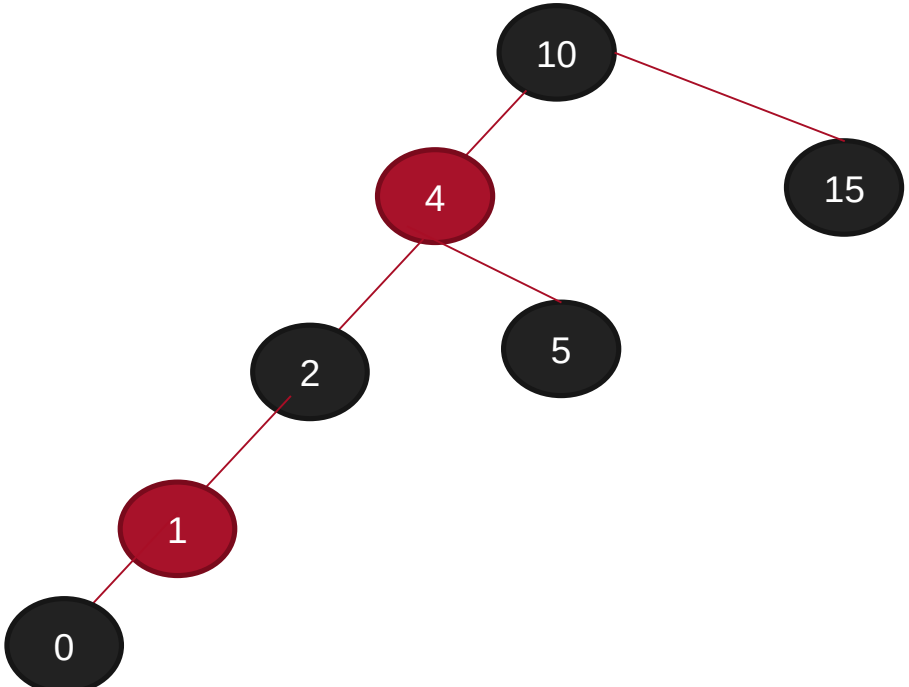
Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



Red Black Tree

Based on last property,

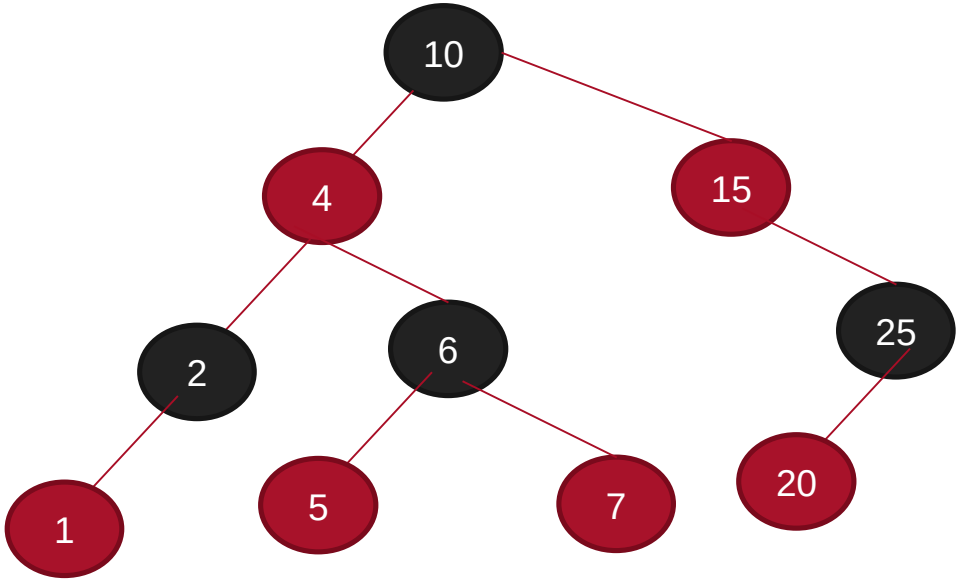
The longest path from the root is no more than twice the length of the shortest path.

OR

The path from the root to its farthest leaf node or extreme leaf node is no more than twice the length of root to its nearest node.

Red Black Tree

Check whether it is Red Black tree or not? Which property it is violating



Red Black Tree

Rules for Insertion in Red Black tree

1. If tree is empty, create a new node as root node with color black.
2. If tree is not empty, create new node as leaf node with color red.
3. If parent of a new node is black then exit.
4. If parent of a new node is red then check the color of parents sibling of new node
    1. If color is black or null then do suitable rotation and recolor.
    2. If color is red then recolor and also check if parent's parent of new node is not root node then recolor it and recheck.

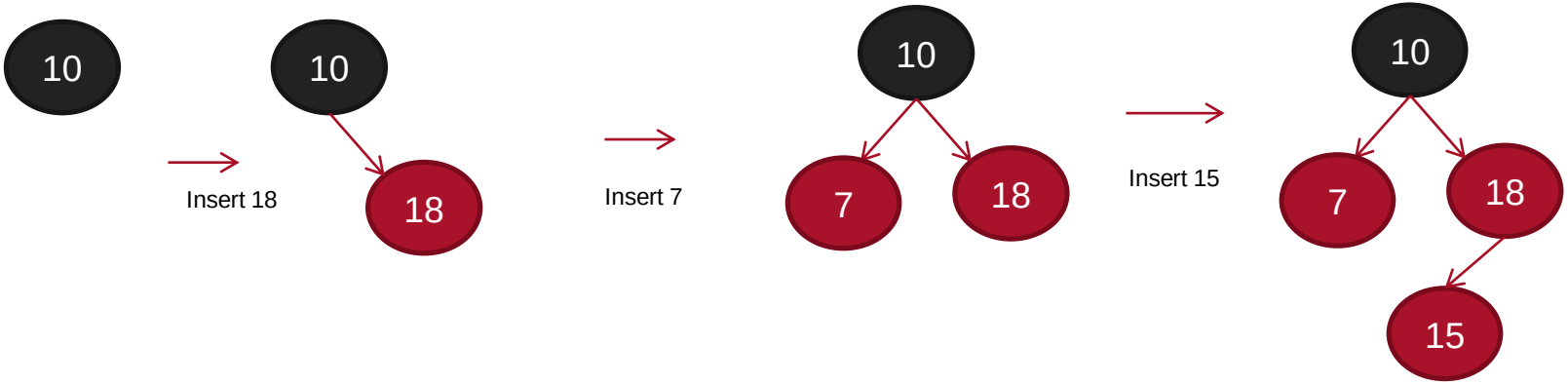
Properties to be Noted:

- Root = Black
- No two adjacent red nodes (If parent red then children black)
- Count the no. of black nodes in each path

Red Black Tree

Insertion in Red Black tree

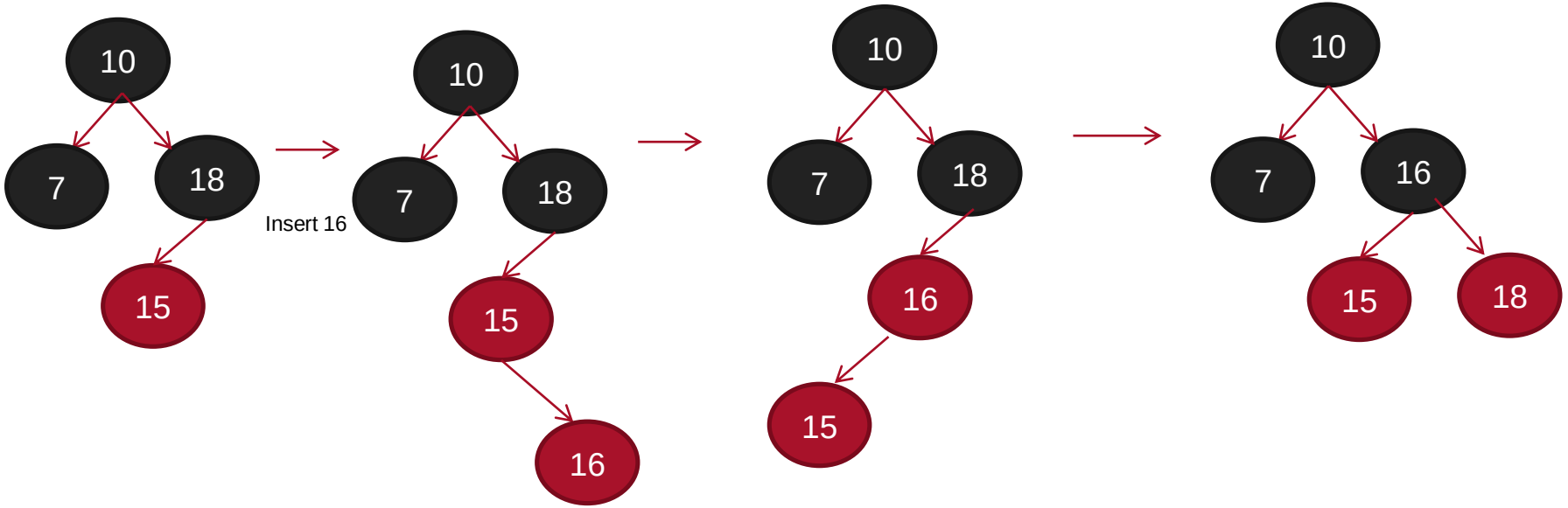
10,18,7,15,16,30,25,40,60,2,1,70



Red Black Tree

Insertion in Red Black tree

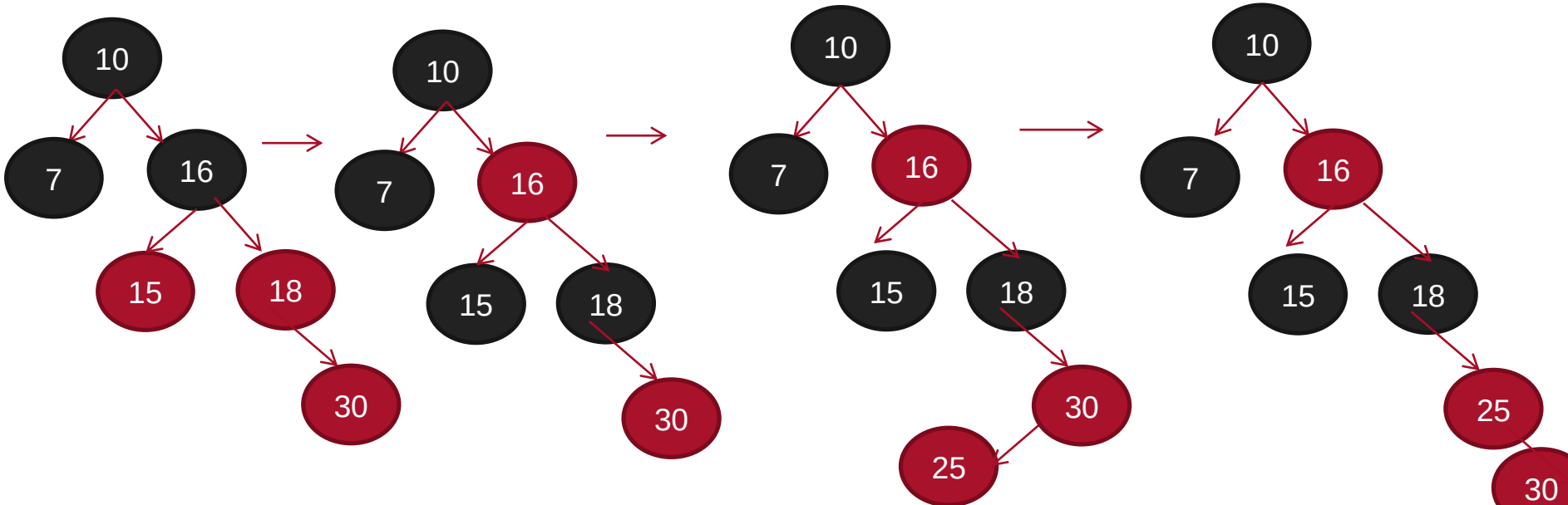
10,18,7,15,16,30,25,40,60,2,1,70



Red Black Tree

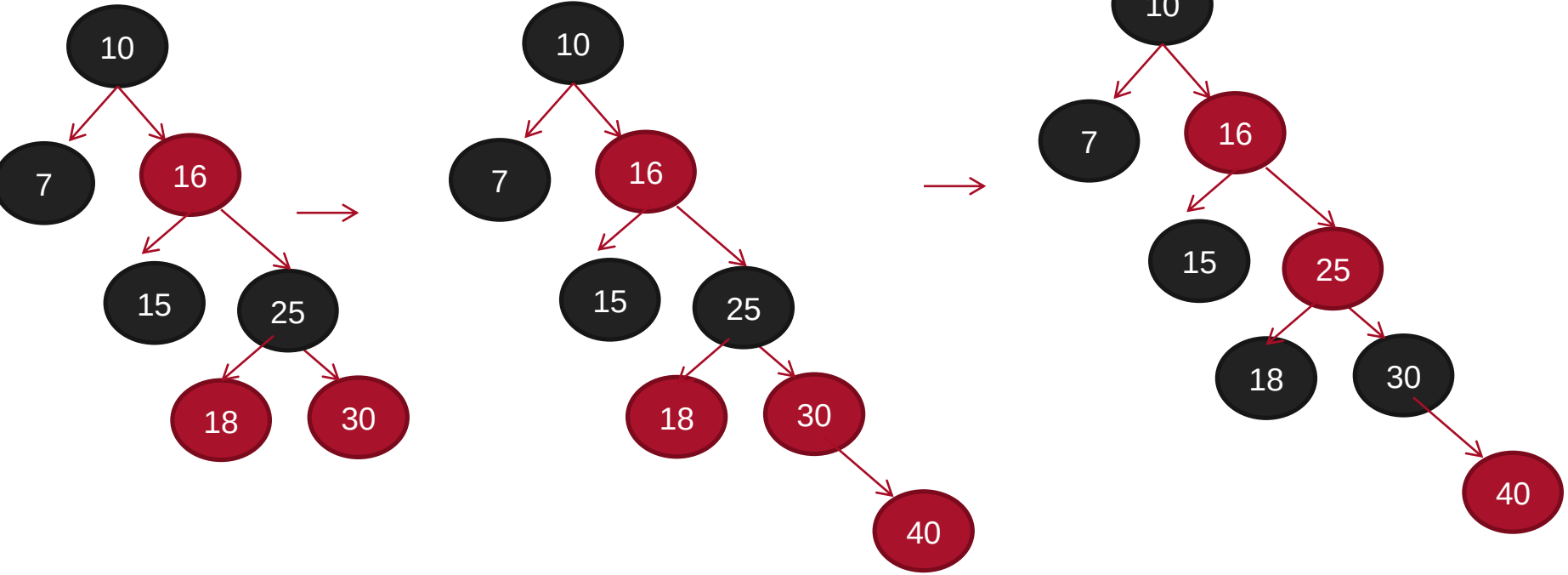
Insertion in Red Black tree

10,18,7,15,16,30,25,40,60,2,1,70

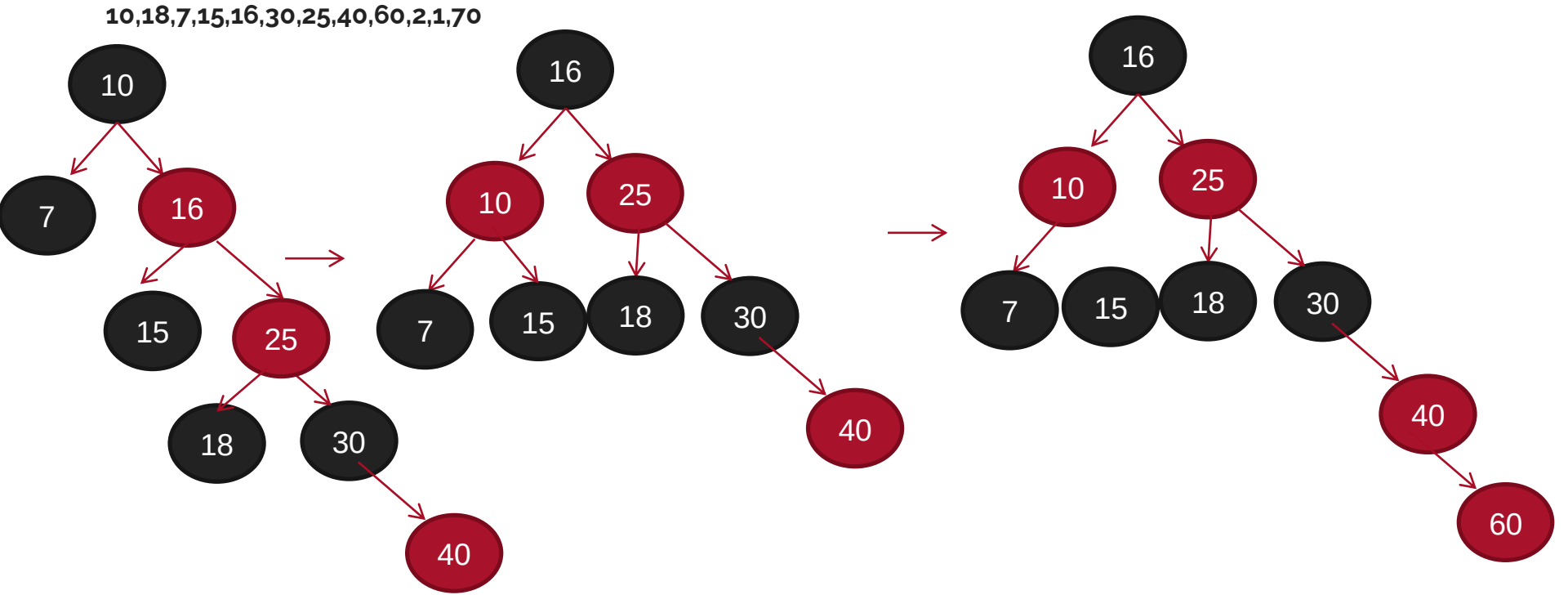


Red Black Tree

10,18,7,15,16,30,25,40,60,2,1,70

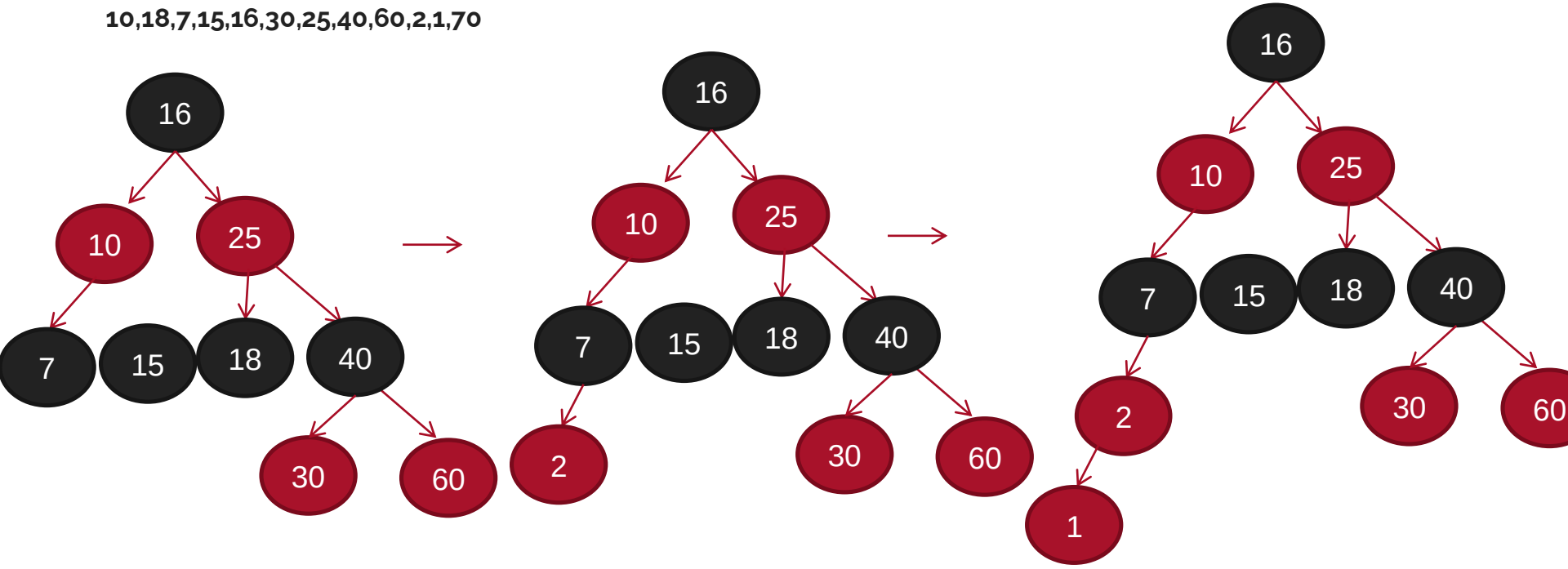


Red Black Tree



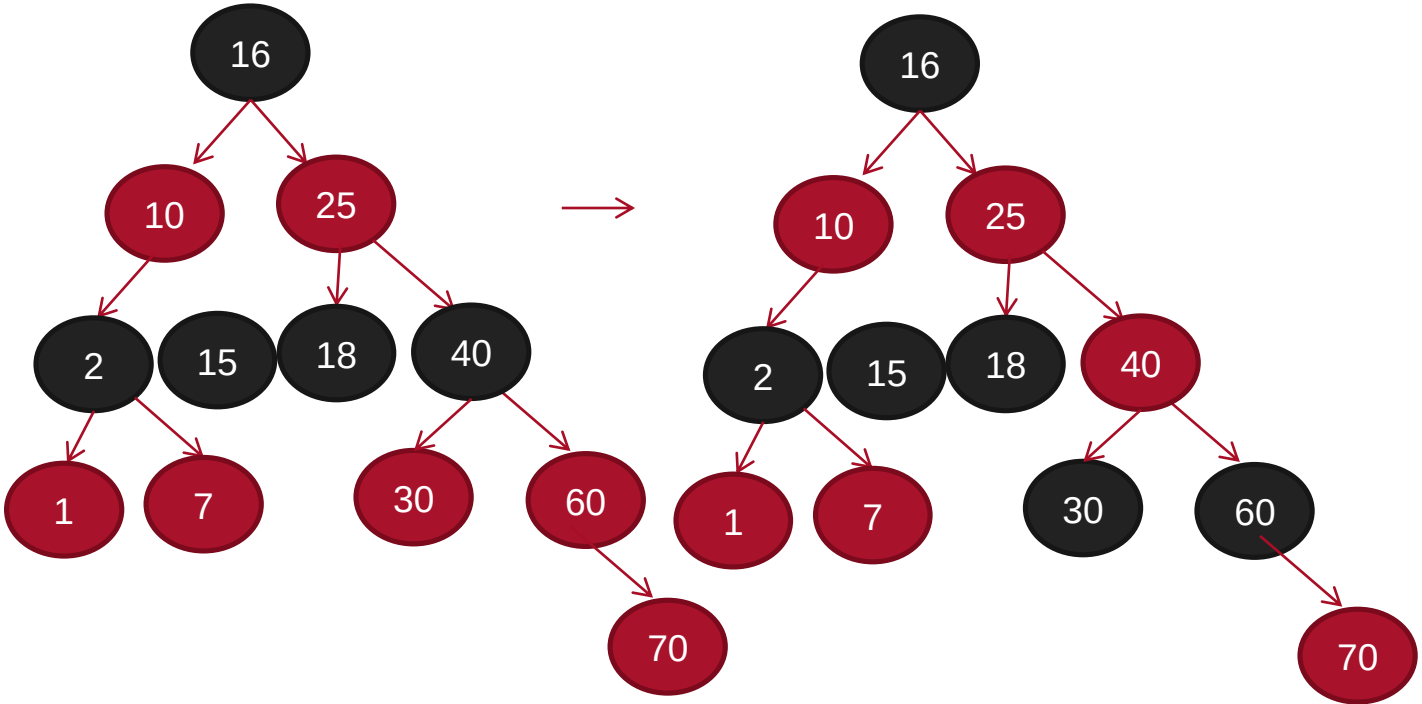
Red Black Tree

10,18,7,15,16,30,25,40,60,2,1,70



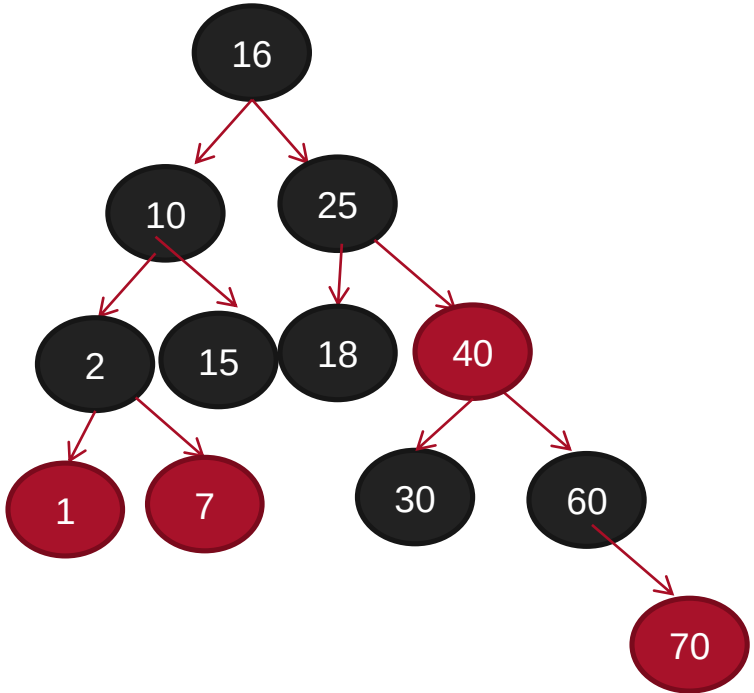
Red Black Tree

10,18,7,15,16,30,25,40,60,2,1,70



Red Black Tree

10,18,7,15,16,30,25,40,60,2,1,70



Red Black Tree

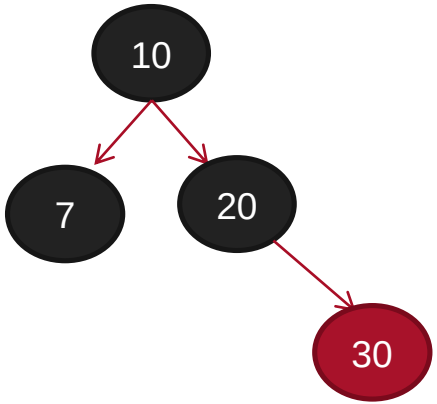
Deletion in RED BLACK Tree

Step 1: Perform BST Deletion

Step 2:

Case 1: If a node to be deleted is red, just delete it.

Delete 30



Red Black Tree

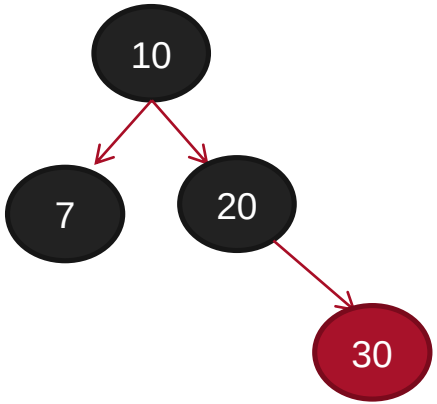
Deletion in RED BLACK Tree

Step 1: Perform BST Deletion

Step 2:

Case 1: If a node to be deleted is red, just delete it.

Delete 20



Red Black Tree

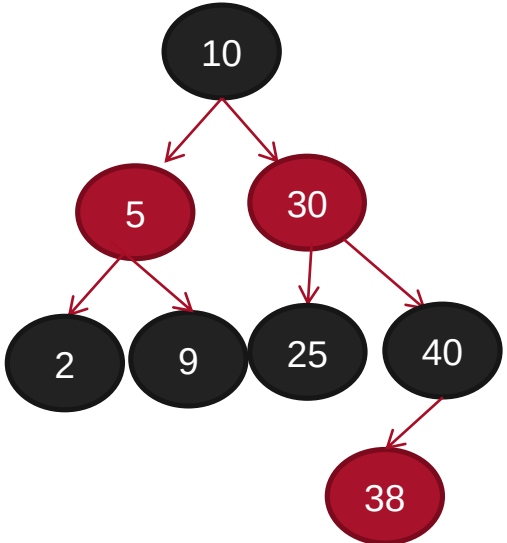
Deletion in RED BLACK Tree

Step 1: Perform BST Deletion

Step 2:

Case 1: If a node to be deleted is red, just delete it.

Delete 30



Red Black Tree

Step 1: Perform BST Deletion

Step 2:

Case 1: If a node to be deleted is red, just delete it.

Case 2: If root is DB, just remove DB

Case 3: If DB's sibling is black and both its children are black

3.1 Remove DB

3.2 Add Black to its parent (P)

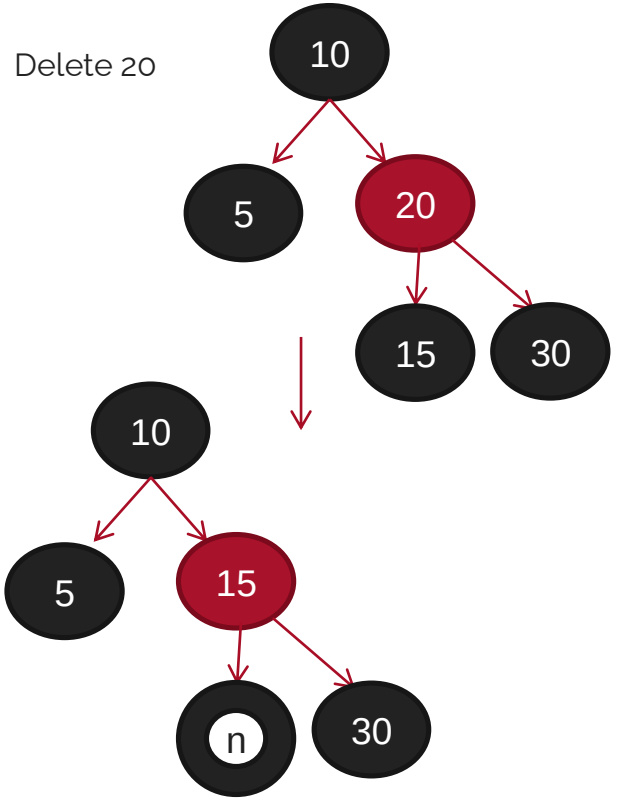
3.2.1 If P is red it becomes black

3.2.2 If P is black it becomes DB.

3.3 Add Red to its sibling

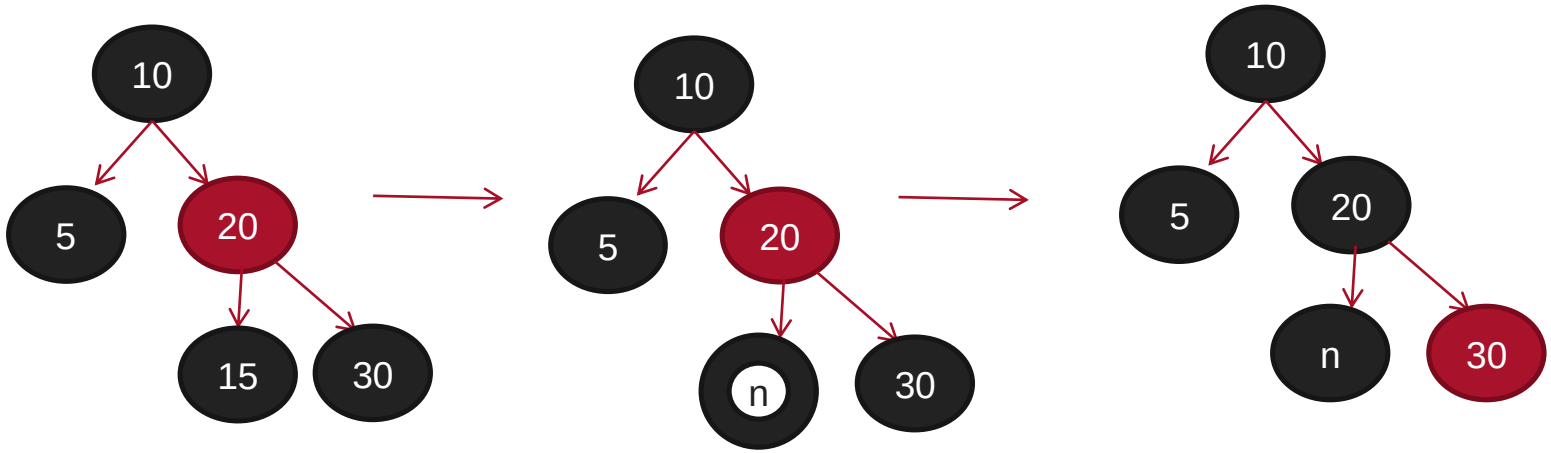
If still DB exists we will apply other cases.

Delete 20



Red Black Tree

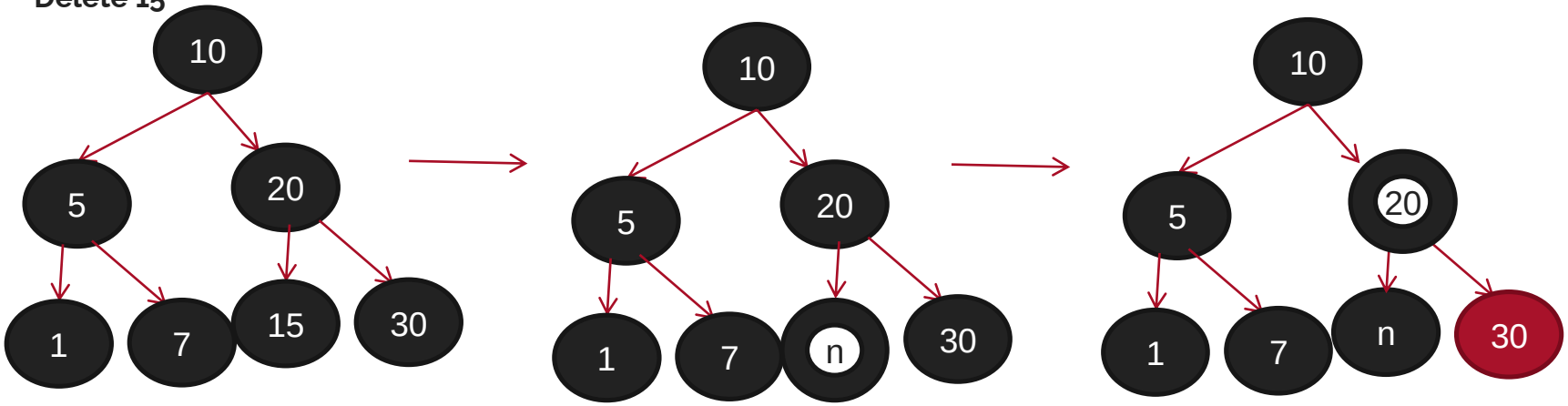
Delete 15



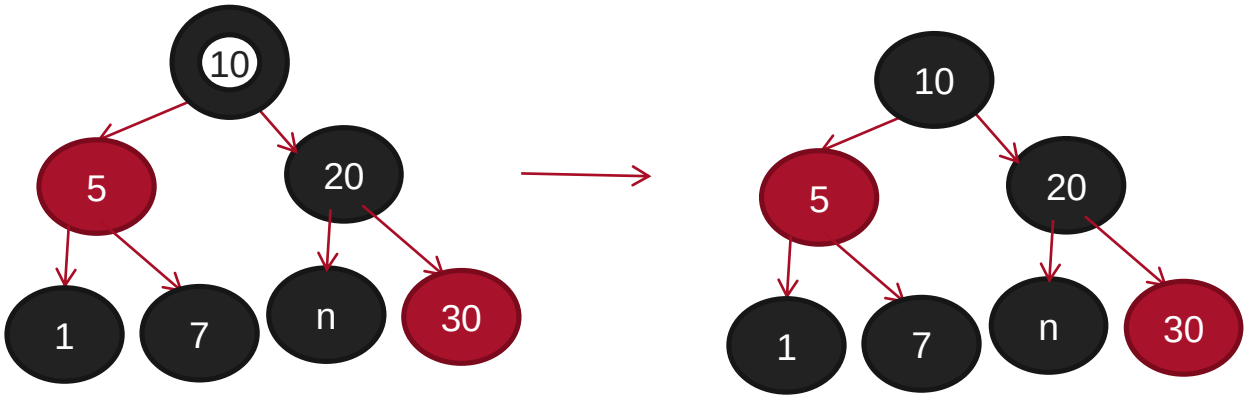
Red Black Tree

If parent is already black then check this case

Delete 15



Red Black Tree



Red Black Tree

Check the case if sibling is red in color.

Step 1: Perform BST Deletion

Step 2:

Case 1: If a node to be deleted is red, just delete it.

Case 2: If root is DB, just remove DB

Case 3: If DB's sibling is black and both its children are black

3.1 Remove DB

3.2 Add Black to its parent (P)

3.2.1 If P is red it becomes black

3.2.2 If P is black it becomes DB.

3.3 Add Red to its sibling

If still DB exists we will apply other cases.

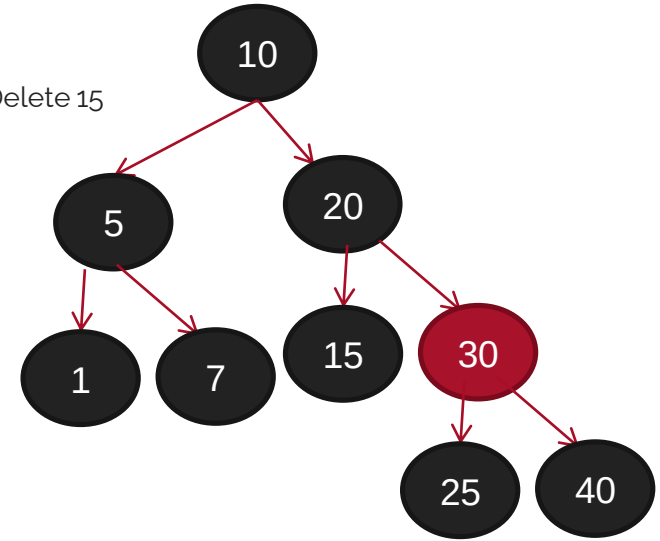
Case 4: If DB sibling is red

4.1 Swap the colors of Parent and sibiling

4.2 Rotate parent in DB direction

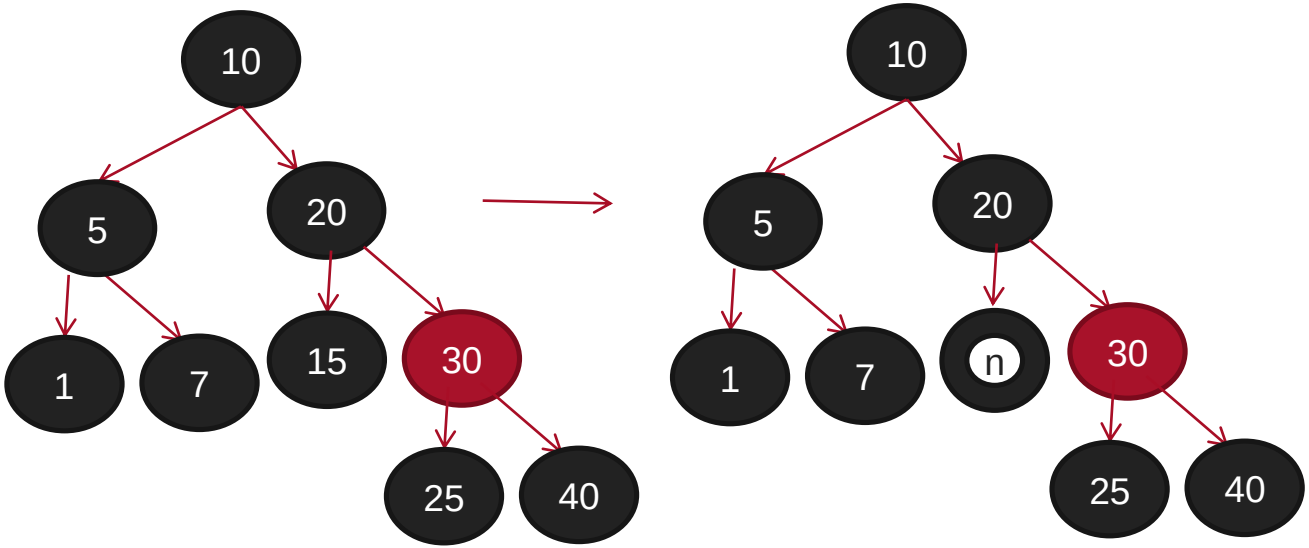
4.3 Reapply cases

Delete 15

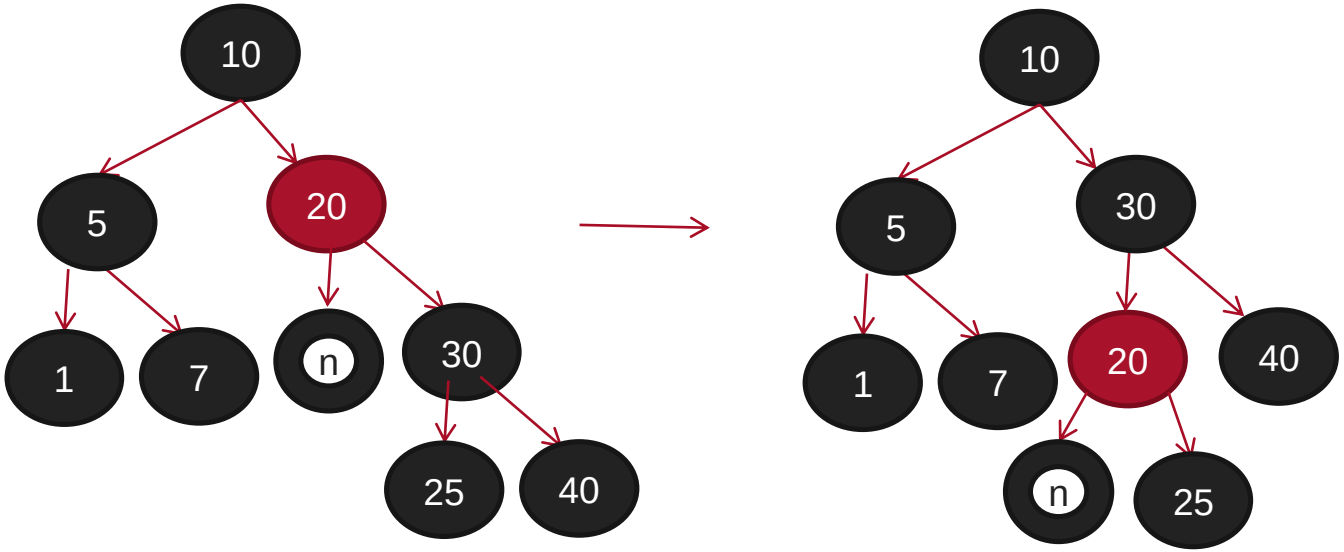


Red Black Tree

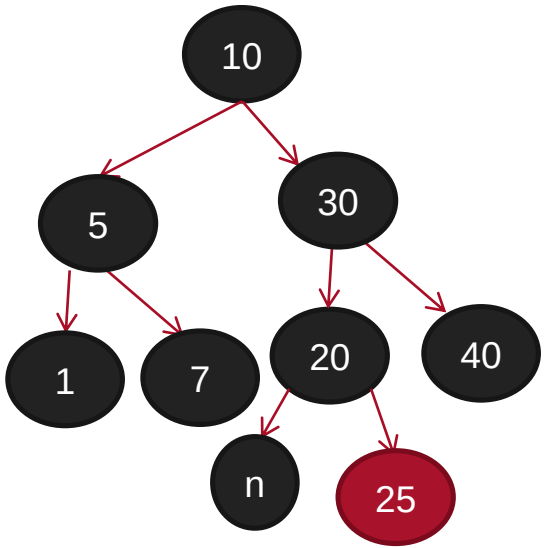
Delete 15



Red Black Tree



Red Black Tree



Red Black Tree

Case 5: DB's sibling is black, sibling's child who is far from DB is black but near child to DB is red.

4.1 Swap the colors of DB's sibiling and its child who is near to DB

4.2 Rotate sibling in opposite direction to DB

4.3 Apply case 6

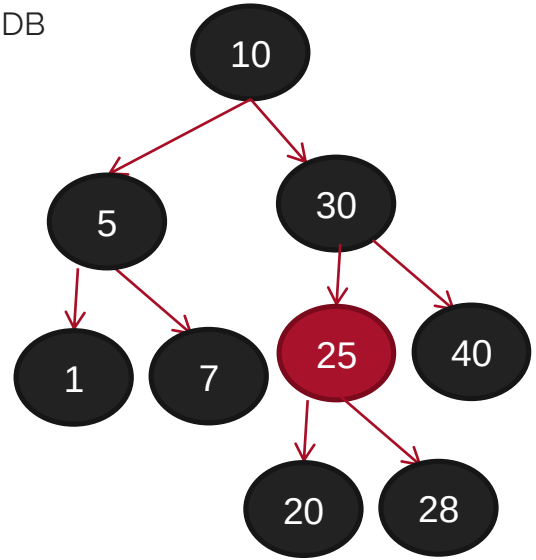
Case 6: DB's sibling is black, far child is red

6.1 Swap the colors of Parent and sibiling

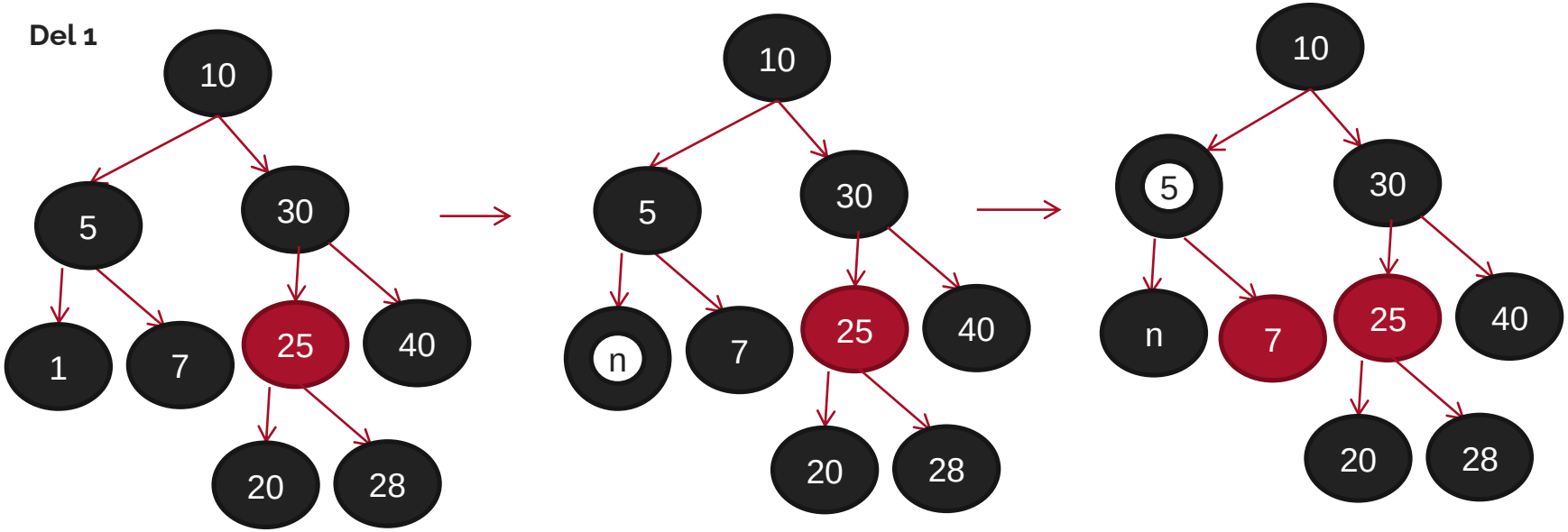
6.2 Rotate parent in DB's direction

6.3 Remove DB

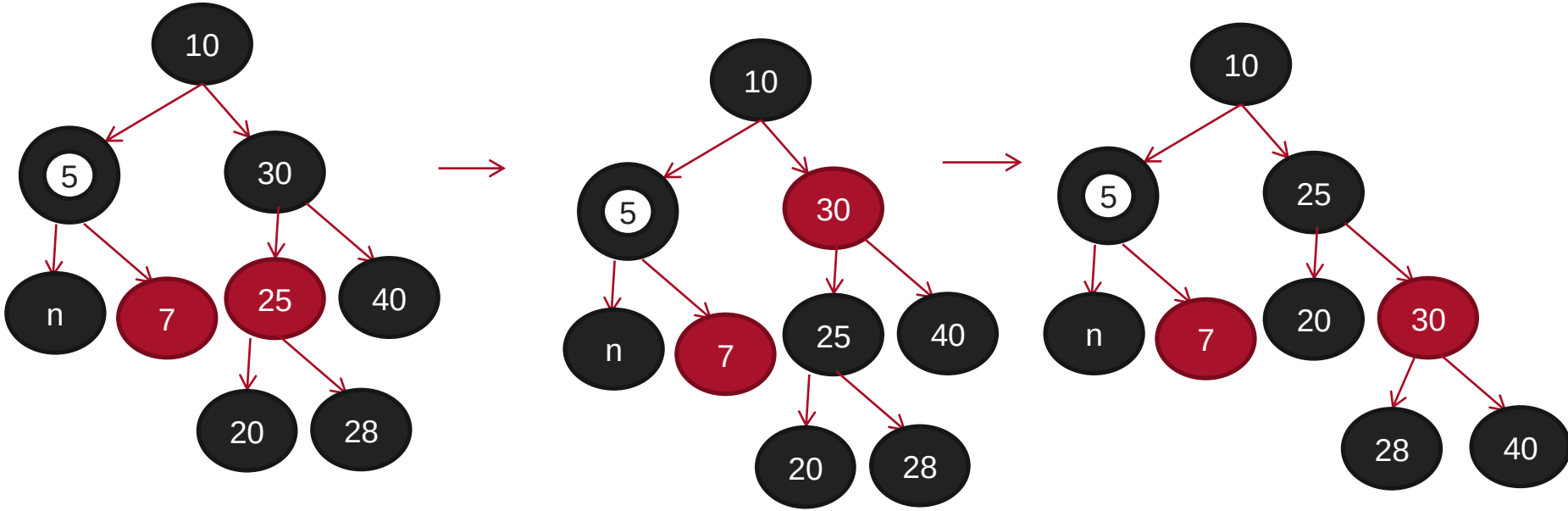
6.4 Change the color of Red child to black.



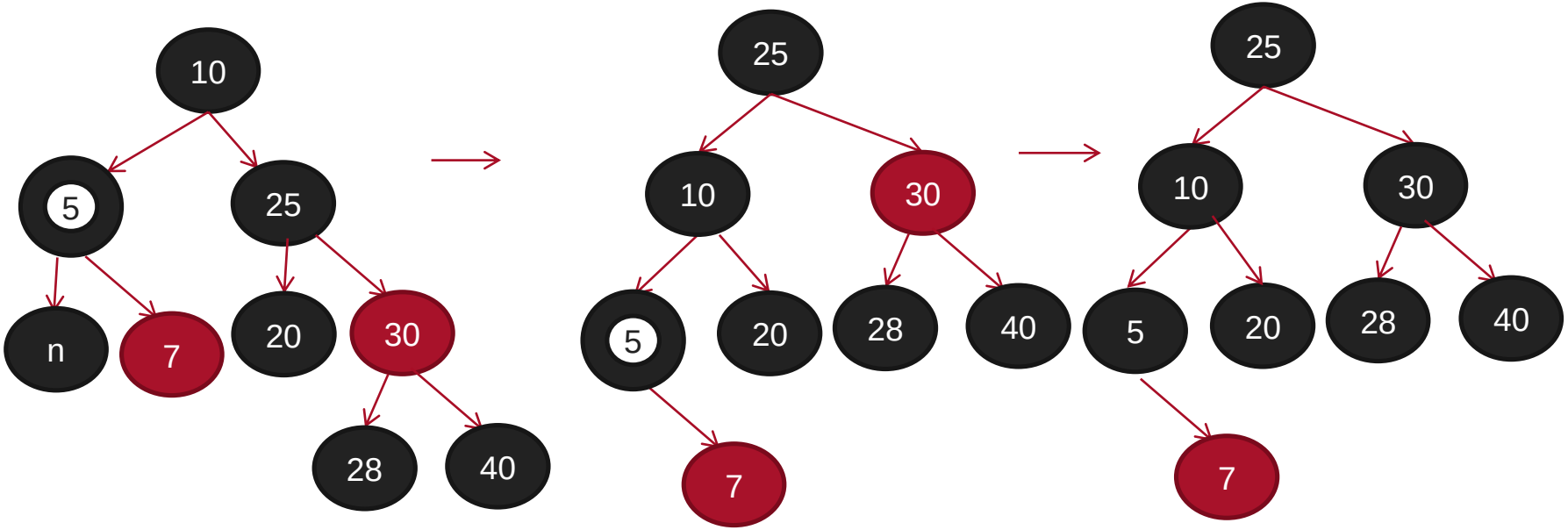
Red Black Tree



Red Black Tree



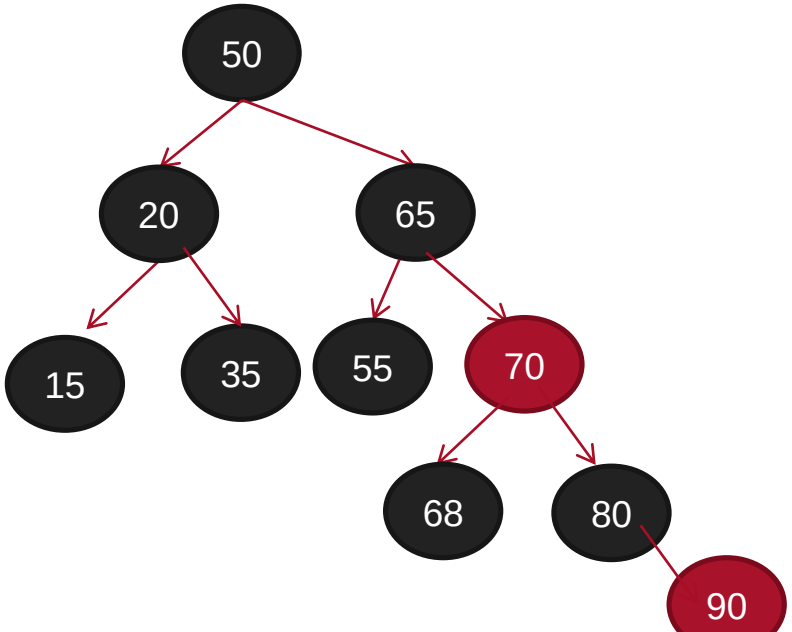
Red Black Tree



Red Black Tree

Exercise:

Delete the nodes: 55,30,90,80,50,35,15,65,68



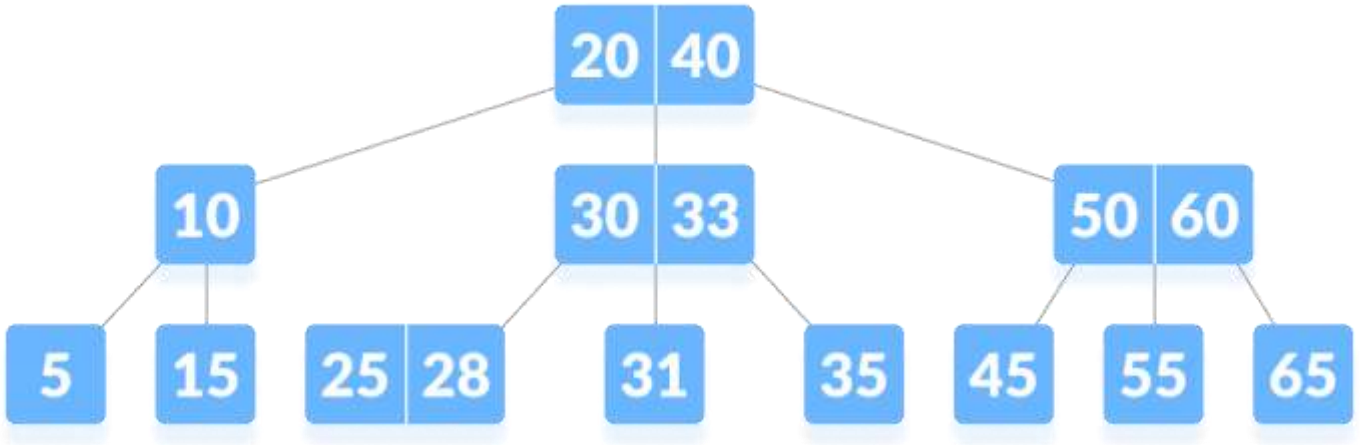
B Tree

- Balanced m-way tree.
- Generalization of BST in which a node can have more than one key and more than two children.
- Maintains sorted data.
- All leaf node 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 => ceil(m/2)

  Every node has max (m-1) keys.
  Min key => root => 1
  all other node => (ceil(m/2)) - 1

B Tree

- B-tree is 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 a generalized form of the binary search tree.
- It is also known as a height-balanced m-way tree.



B Tree

Why do you need a B-tree data structure?

- The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses.
- Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large and the access time increases.
- However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B Tree

Operations on a B-tree

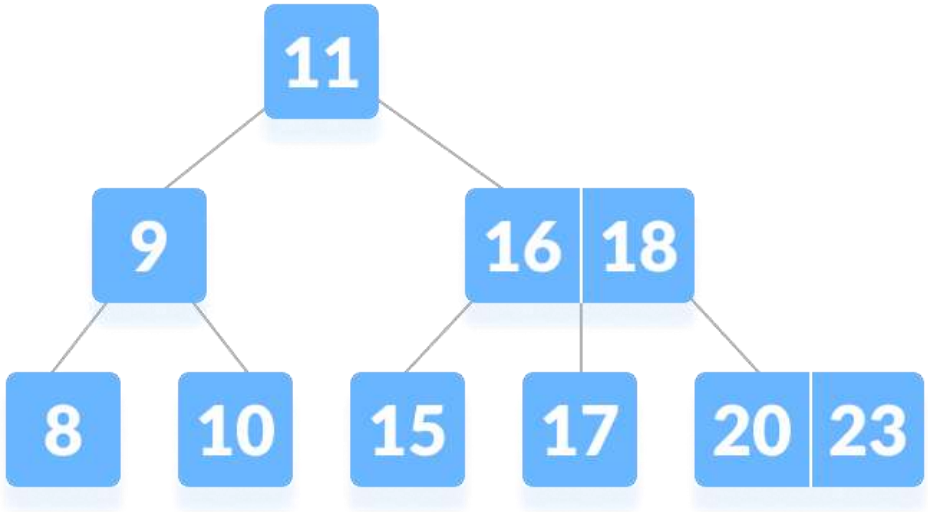
Searching an element in a B-tree

- Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.
- Starting from the root node, compare k with the first key of the node.
  If k = the first key of the node, return the node and the index.
- If k.leaf = true, return NULL (i.e. not found).
- If k < the first key of the root node, search the left child of this key recursively.
- If there is more than one key in the current node and k > the first key, compare k with the next key in the node.
  If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).
  Else, search the right child of the key.
- Repeat steps 1 to 4 until the leaf is reached.

B Tree

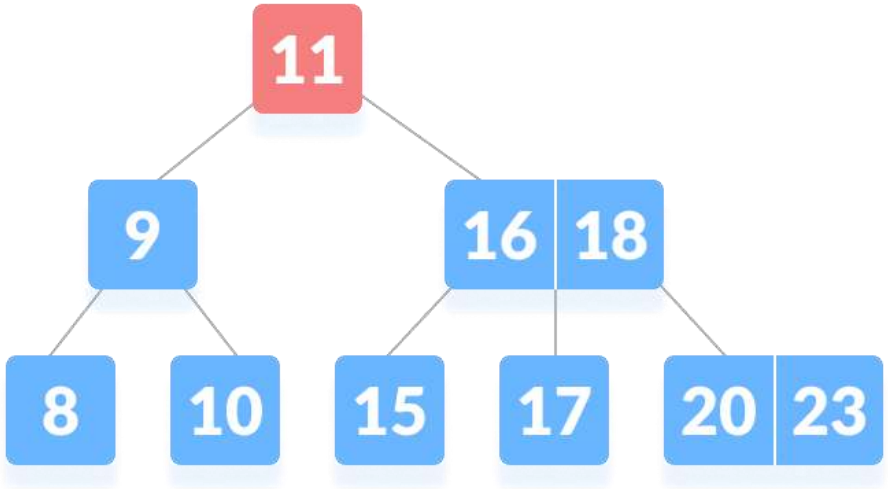
Searching Example

- Let us search key k = 17 in the tree below of degree 3.



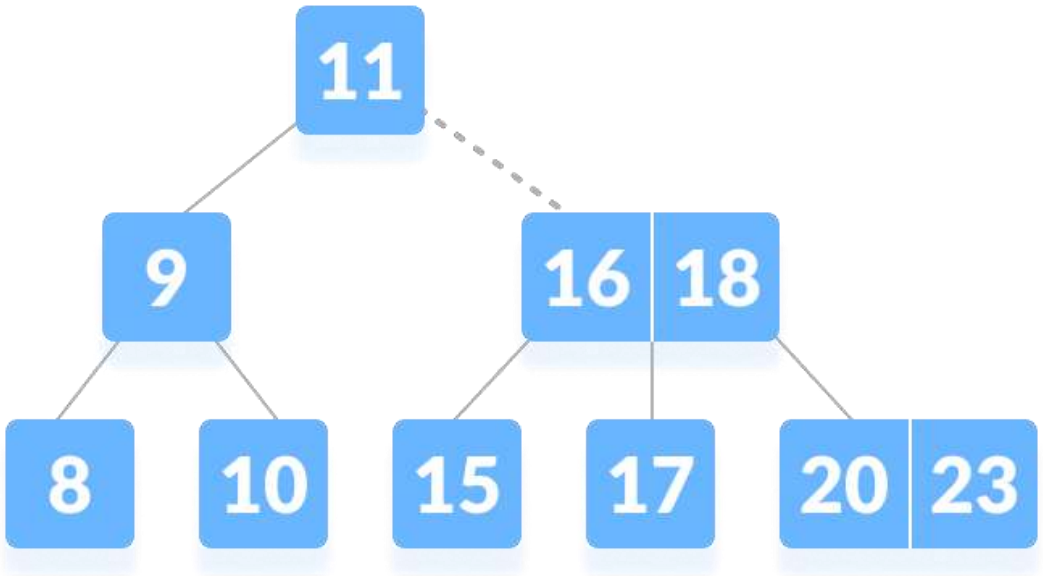
B Tree

k is not found in the root so, compare it with the root key.



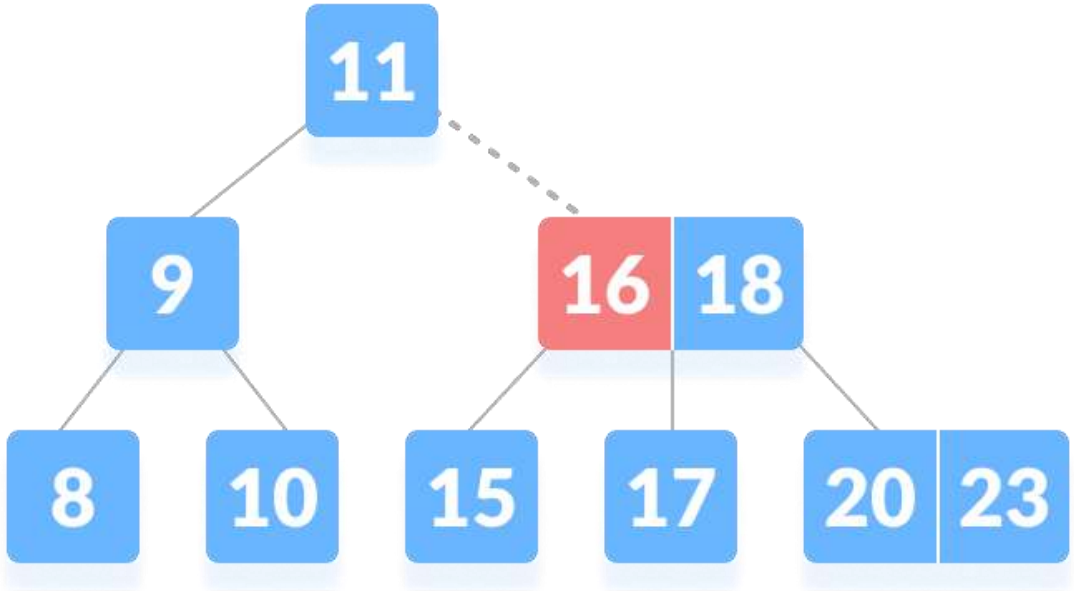
B Tree

Since k > 11, go to the right child of the root node.



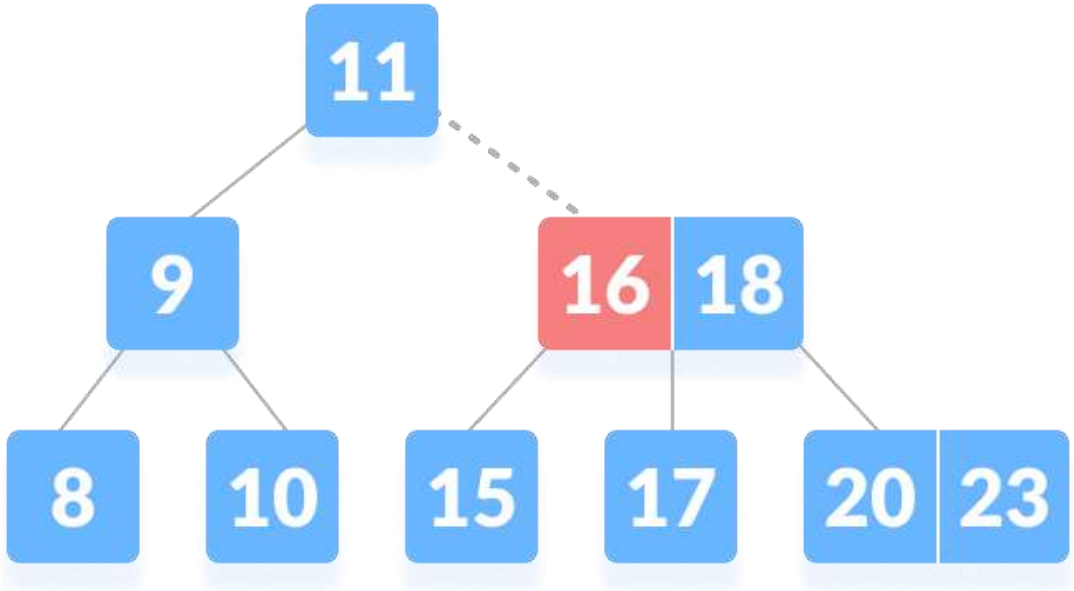
B Tree

Compare k with 16. Since k > 16, compare k with the next key 18.



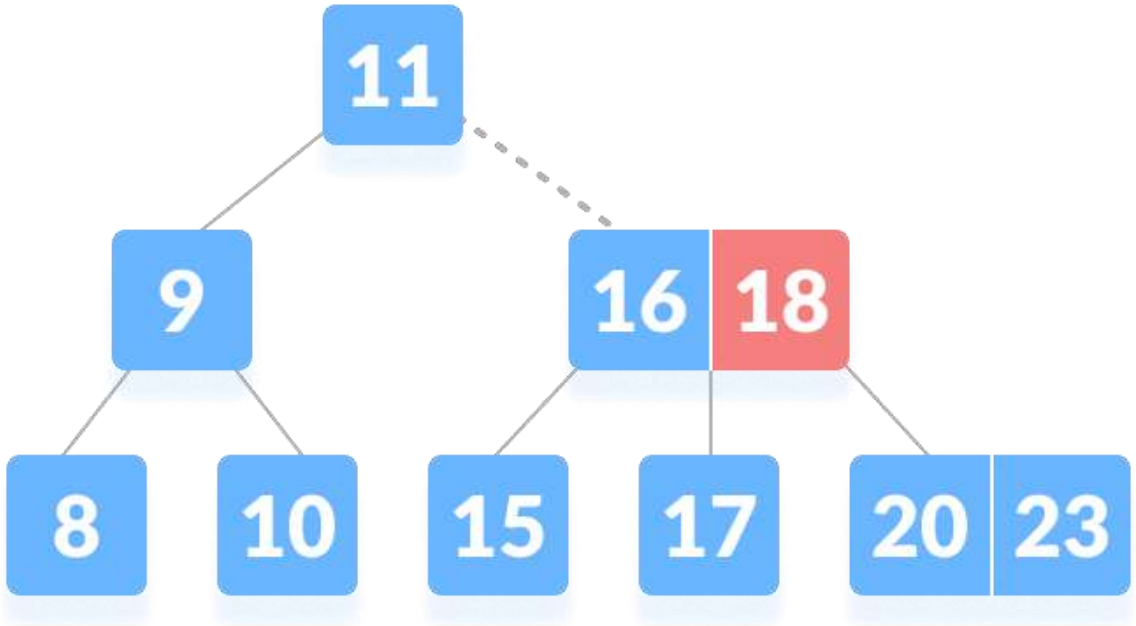
B Tree

Compare k with 16. Since k > 16, compare k with the next key 18.



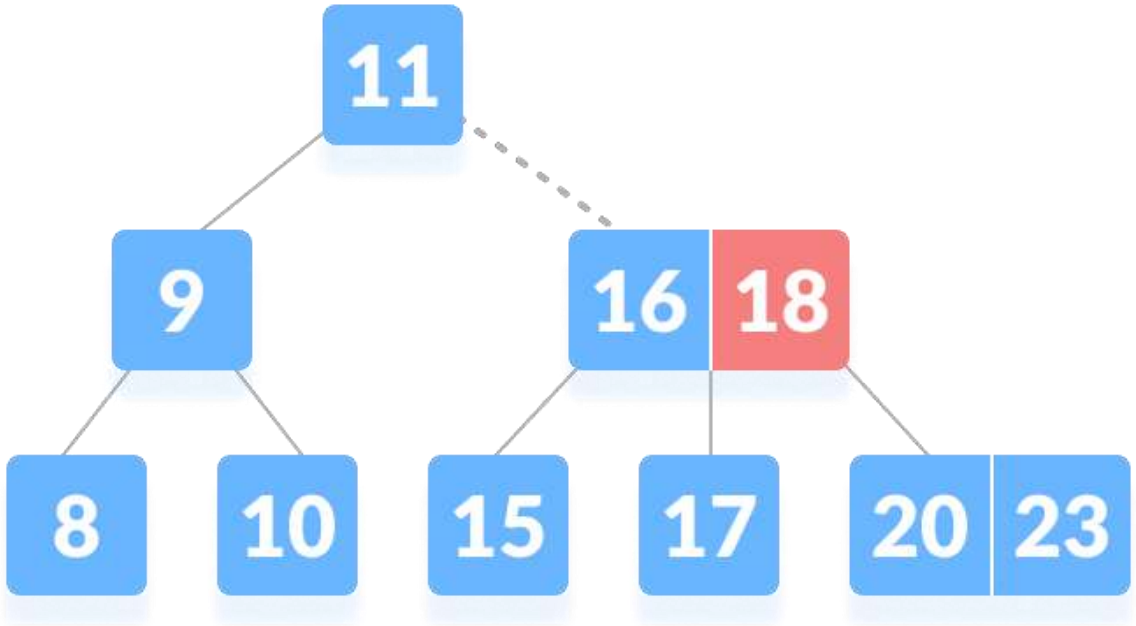
B Tree

Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



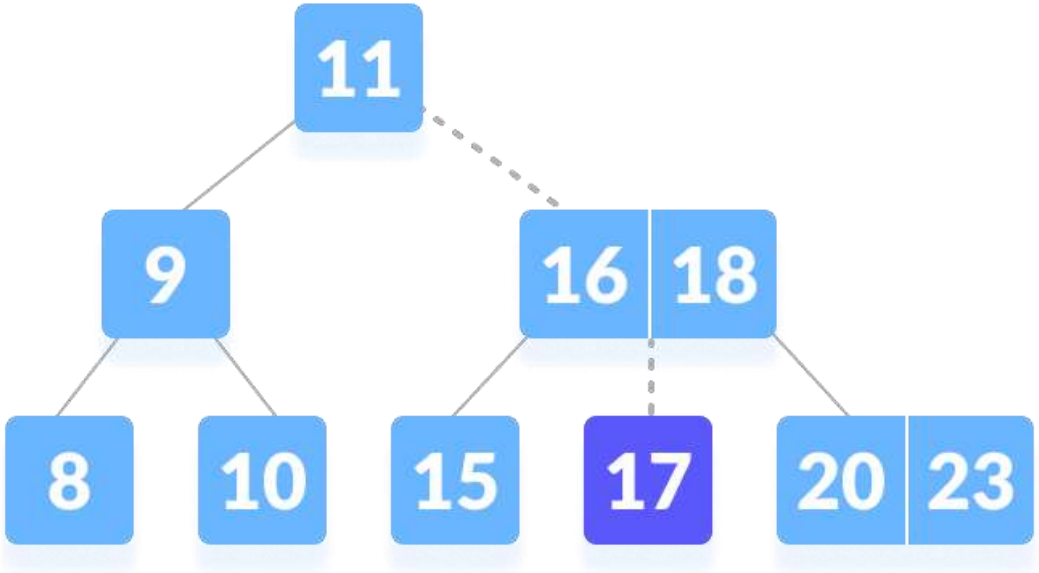
B Tree

Since k < 18, k lies between 16 and 18. Search in the right child of 16 or the left child of 18.



B Tree

k is found.



B Tree

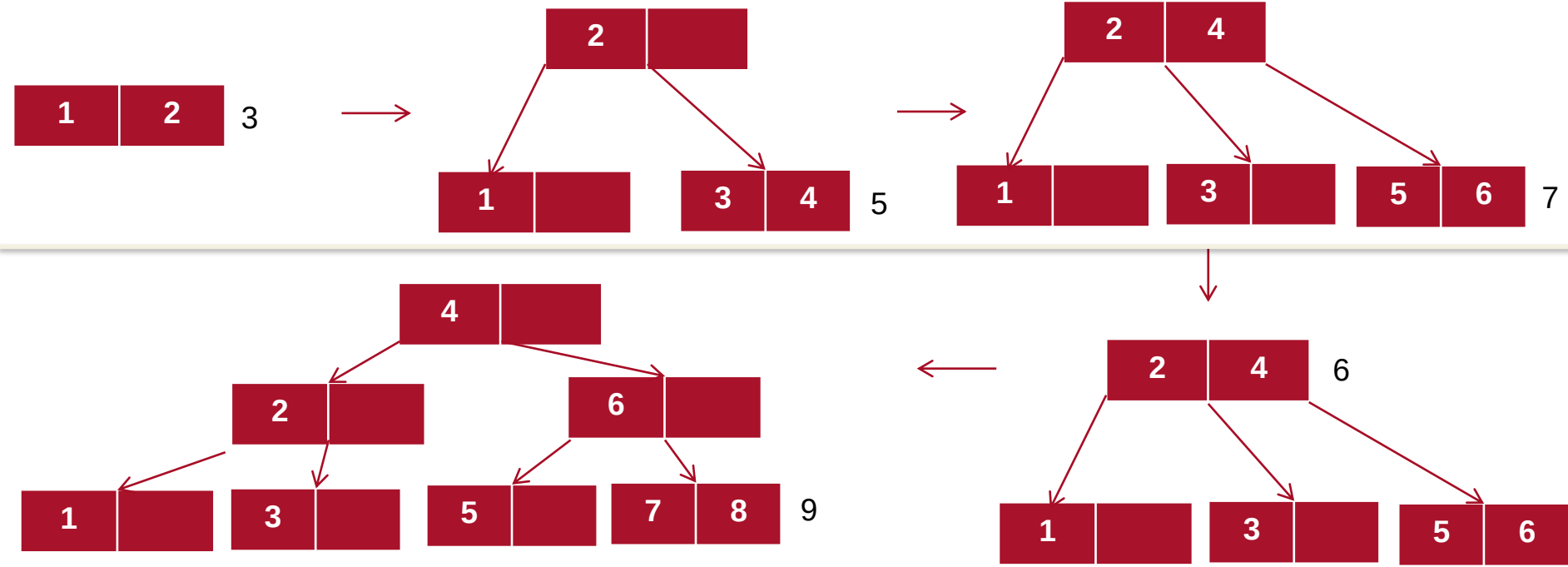
Insertion into a B-tree

Inserting an element on a B-tree consists of two events: **searching the appropriate node** to insert the element and **splitting the node** if required.Insertion operation always takes place in the bottom-up approach.

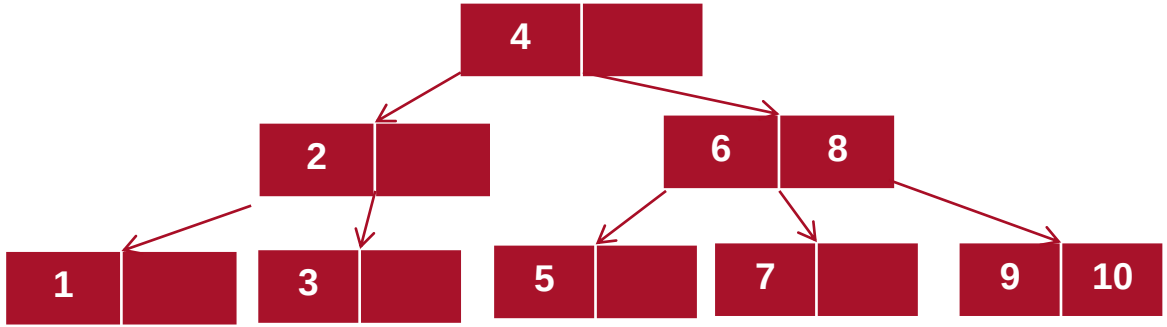
- **Insertion Operation**
- If the tree is empty, allocate a root node and insert the key.
- Update the allowed number of keys in the node.
- Search the appropriate node for insertion.
- If the node is full, follow the steps below.
- Insert the elements in increasing order.
- Now, there are elements greater than its limit. So, split at the median.
- Push the median key upwards and make the left keys as a left child and the right keys as a right child.
- If the node is not full, follow the steps below.
- Insert the node in increasing order.

B Tree

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



B Tree



B Tree

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

B Tree

- Create a B-tree of order 5 with following data.

D,H,Z,K,B,P,Q,E,A,S,W,T,C,L,N,Y,M

B Tree

- Create a B-tree of order 4 with following data.

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

B Tree

- **Deletion from a B-tree**
- Deleting an element on a B-tree consists of three main events: **searching the node where the key to be deleted exists**, deleting the key and balancing the tree if required.
- While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.
- The terms to be understood before studying deletion operation are:
- **Inorder Predecessor**
  The largest key on the left child of a node is called its inorder predecessor.
- **Inorder Successor**
  The smallest key on the right child of a node is called its inorder successor.

B Tree

Deletion Operation

- Before going through the steps below, one must know these facts about a B tree of degree **m**.
- A node can have a maximum of m children. (i.e. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
- A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)

There are three main cases for deletion operation in a B tree.

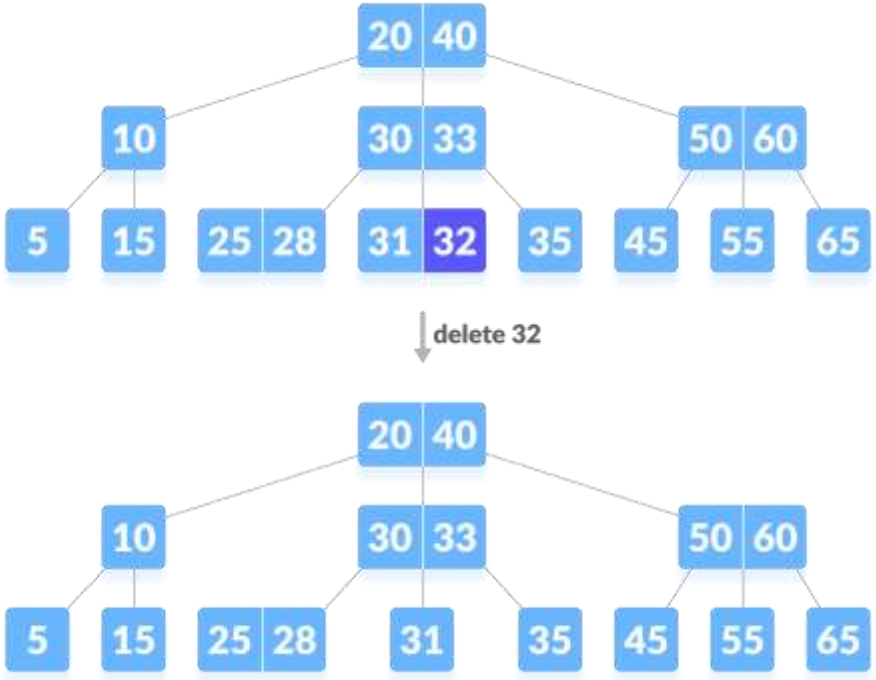
- **Case I**

1. The key to be deleted lies in the leaf. There are two cases for it.

   The deletion of the key does not violate the property of the minimum number of keys a node should hold.

B Tree

In the tree below, deleting 32 does not violate the above properties.



B Tree

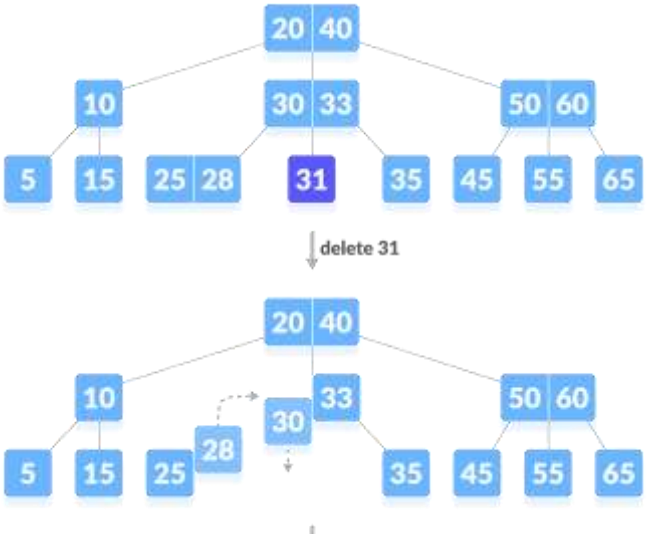
2. The deletion of the key violates the property of the minimum number of keys a node should hold. In this case, we borrow a key from its immediate neighboring sibling node in the order of left to right.

First, visit the immediate left sibling. If the left sibling node has more than a minimum number of keys, then borrow a key from this node.

Else, check to borrow from the immediate right sibling node.

B Tree

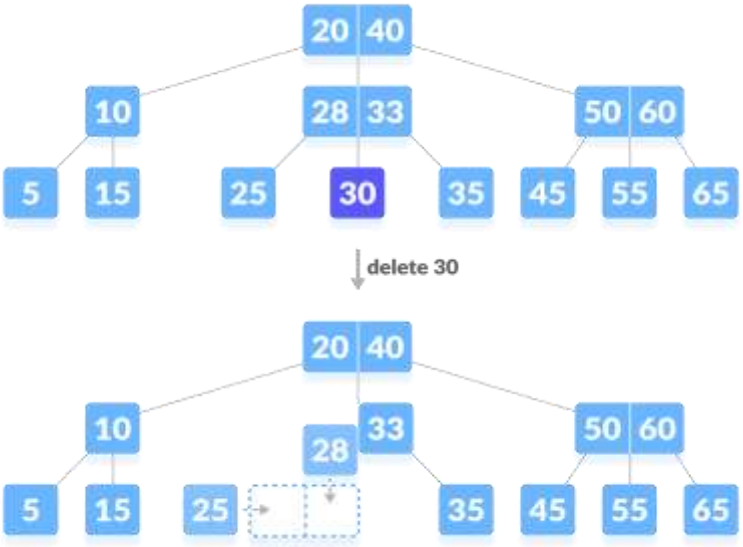
In the tree below, deleting 31 results in the above condition. Let us borrow a key from the left sibling node.

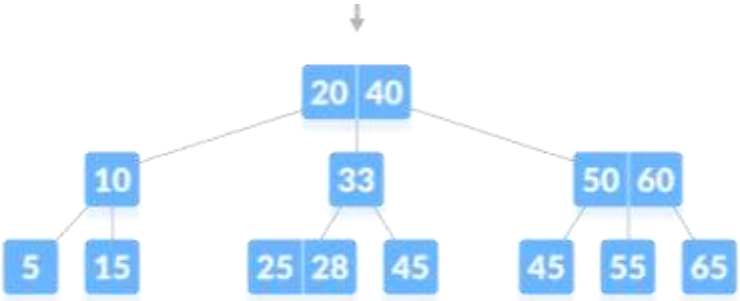


B Tree

If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

Deleting 30 results in the above case.

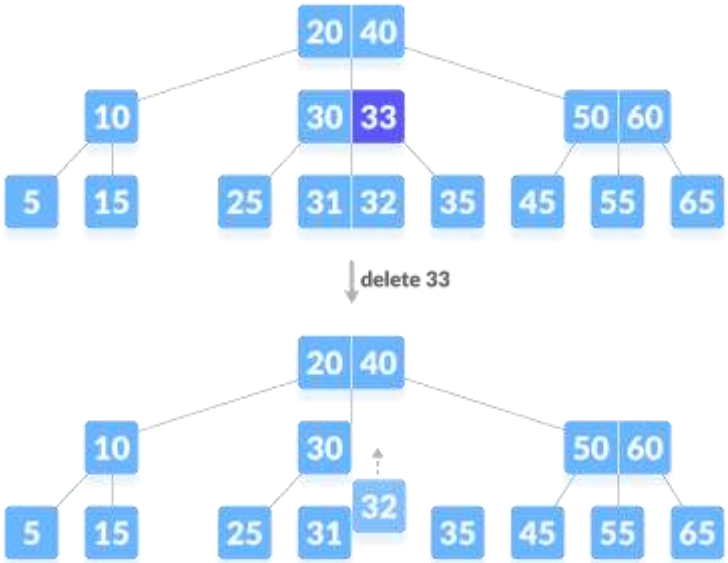


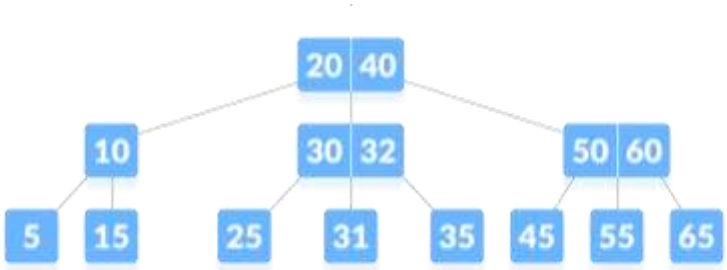


B Tree

Case II

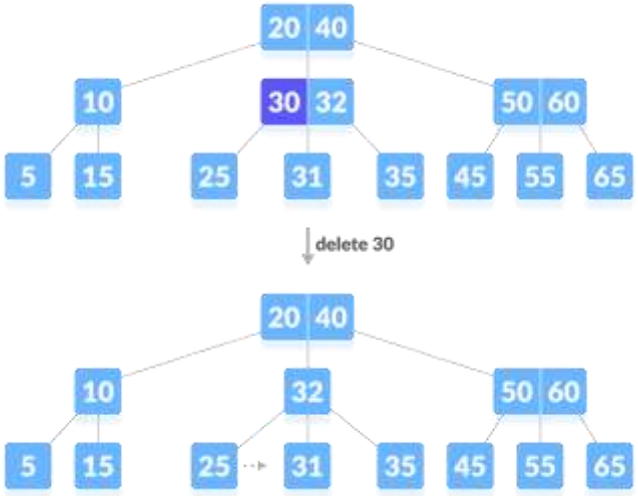
- If the key to be deleted lies in the internal node, the following cases occur.
- The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.





B Tree

- The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.
- If either child has exactly a minimum number of keys then, merge the left and the right children.



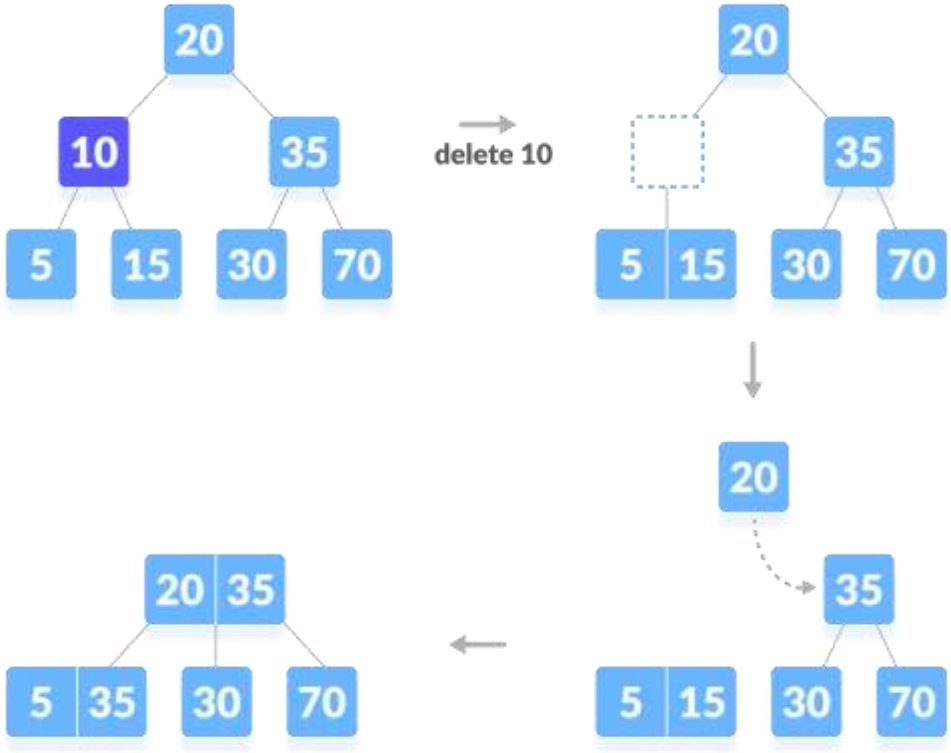
- After merging if the parent node has less than the minimum number of keys then, look for the siblings as in Case I.

B Tree

Case III

- In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.
- Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).

B Tree


20
10
35
5
15
30
70
delete 10
20
35
5
15
30
70
20
35
5
15
30
70
20 35
5
35
30
70

B+ Tree

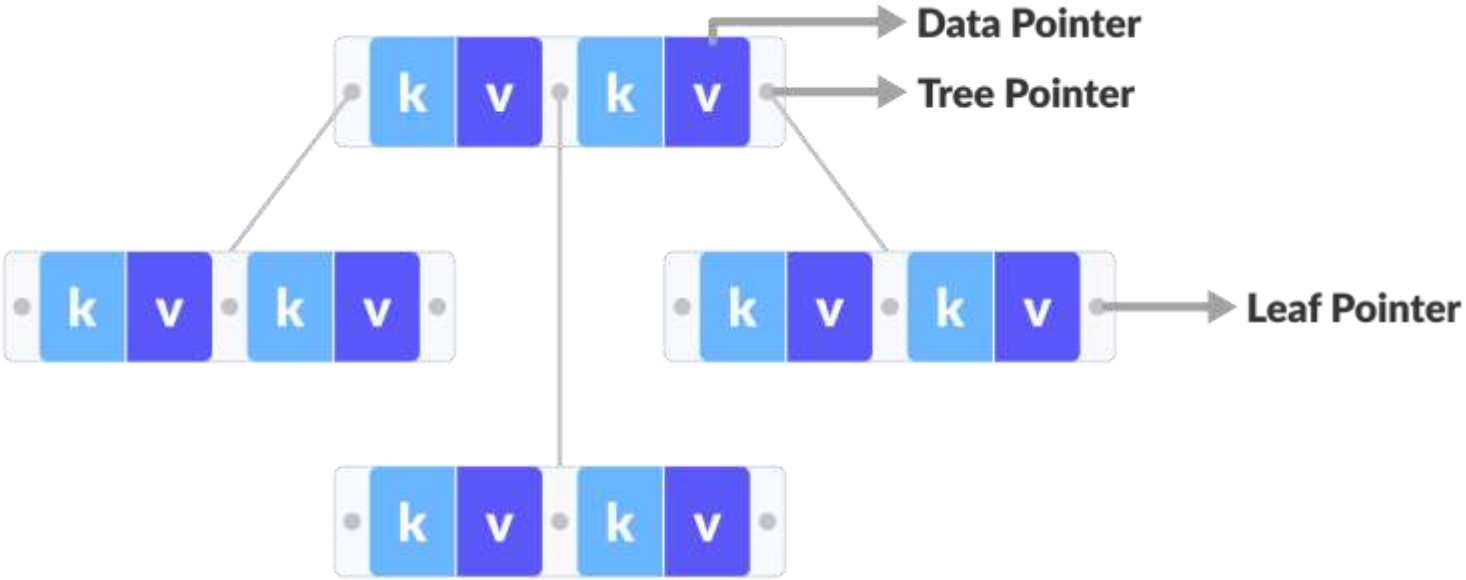
- A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.
- An important concept to be understood before learning B+ tree is multilevel indexing.

Properties of a B+ Tree

- All leaves are at the same level.
- The root has at least two children.
- Each node except root can have a maximum of m children and at least m/2 children.
- Each node can contain a maximum of m - 1 keys and a minimum of [m/2] - 1 keys.

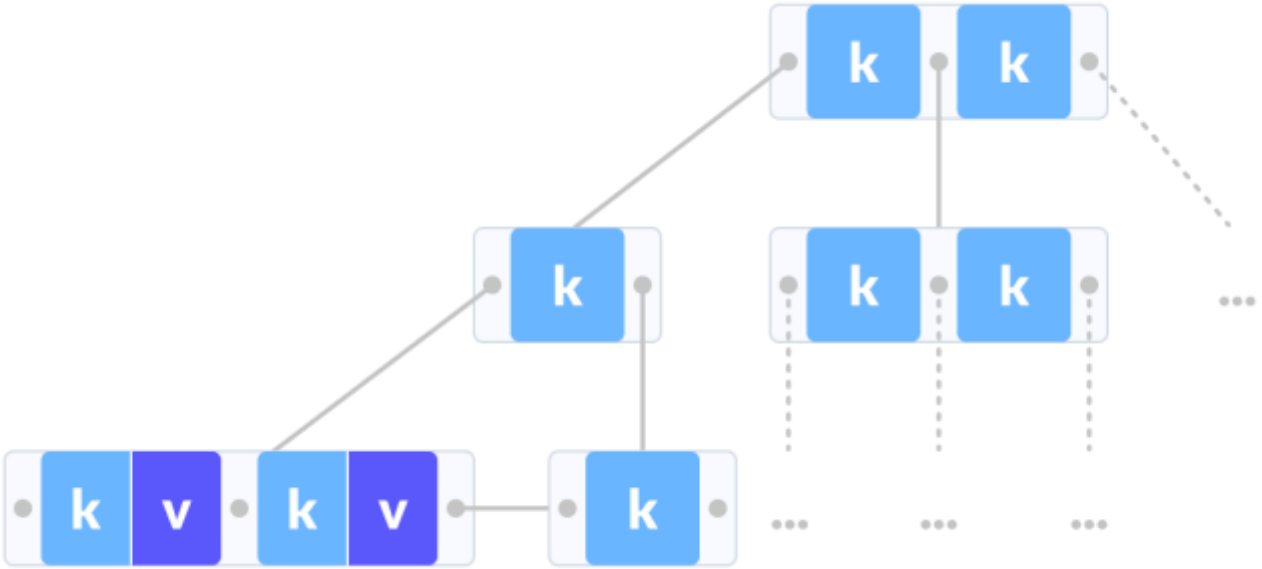
Comparison between a B-tree and a B+ Tree

- B Tree



Comparison between a B-tree and a B+ Tree

- B+ tree



Comparison between a B-tree and a B+ Tree

- The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.
- The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.
- Operations on a B+ tree are faster than on a B-tree.

Searching on a B+ Tree

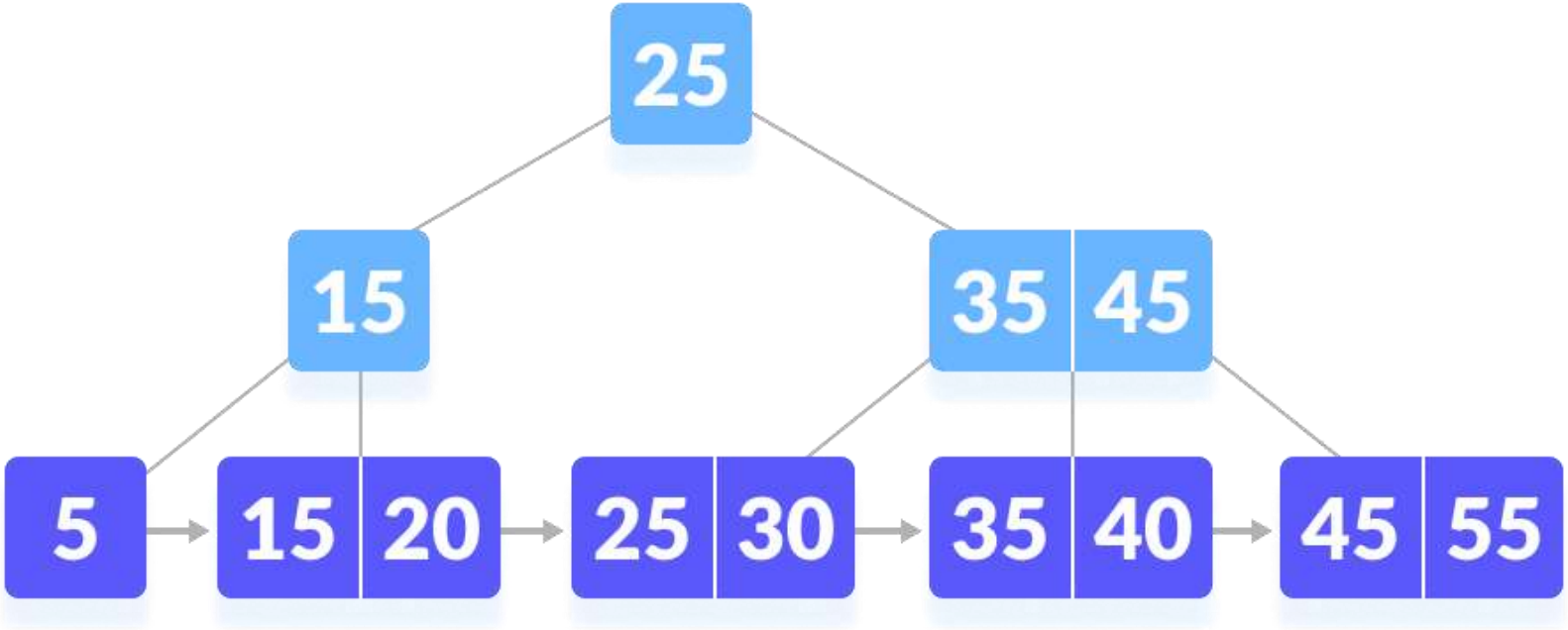
The following steps are followed to search for data in a B+ Tree of order m. Let the data to be searched be k.

- Start from the root node. Compare k with the keys at the root node [k1, k2, k3,.....km - 1].
- If k < k1, go to the left child of the root node.
- Else if k == k1, compare k2. If k < k2, k lies between k1 and k2. So, search in the left child of k2.
- If k > k2, go for k3, k4,...km-1 as in steps 2 and 3.
- Repeat the above steps until a leaf node is reached.
- If k exists in the leaf node, return true else return false.

Searching on a B+ Tree

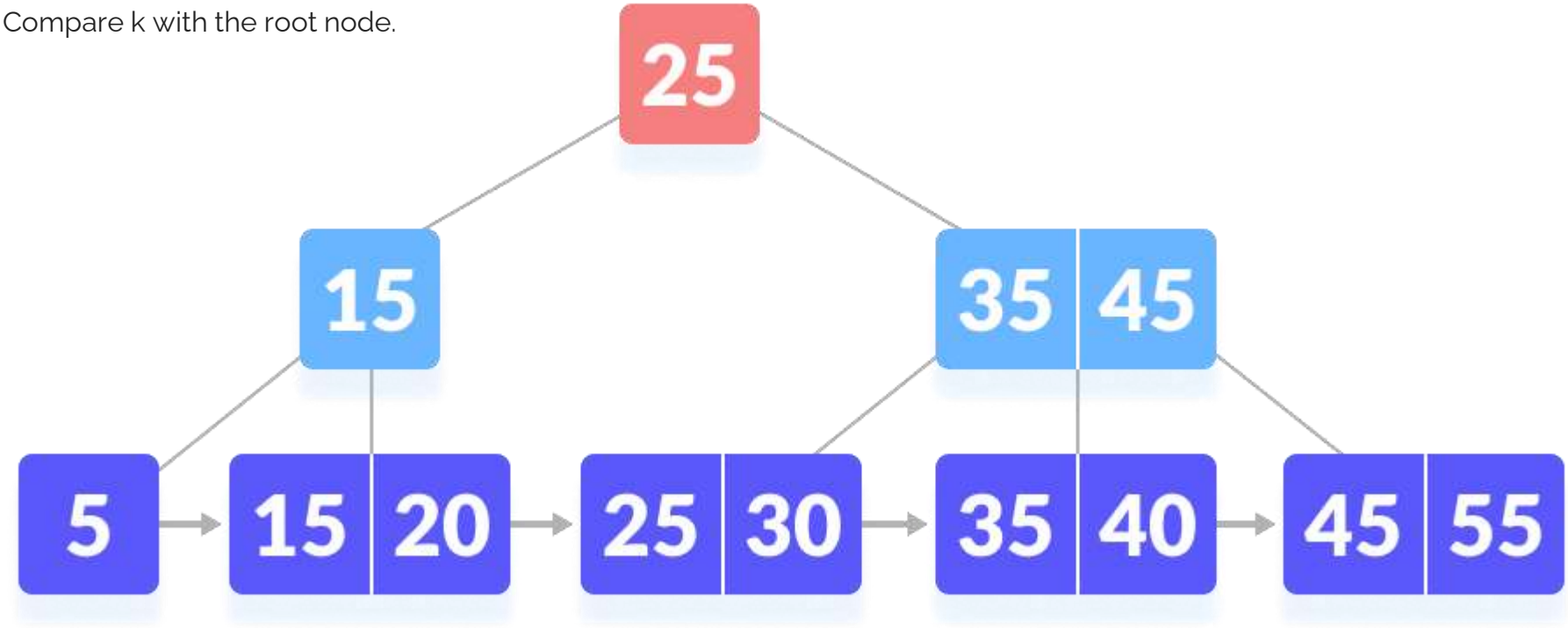
Searching Example on a B+ Tree

- Let us search k = 45 on the following B+ tree.



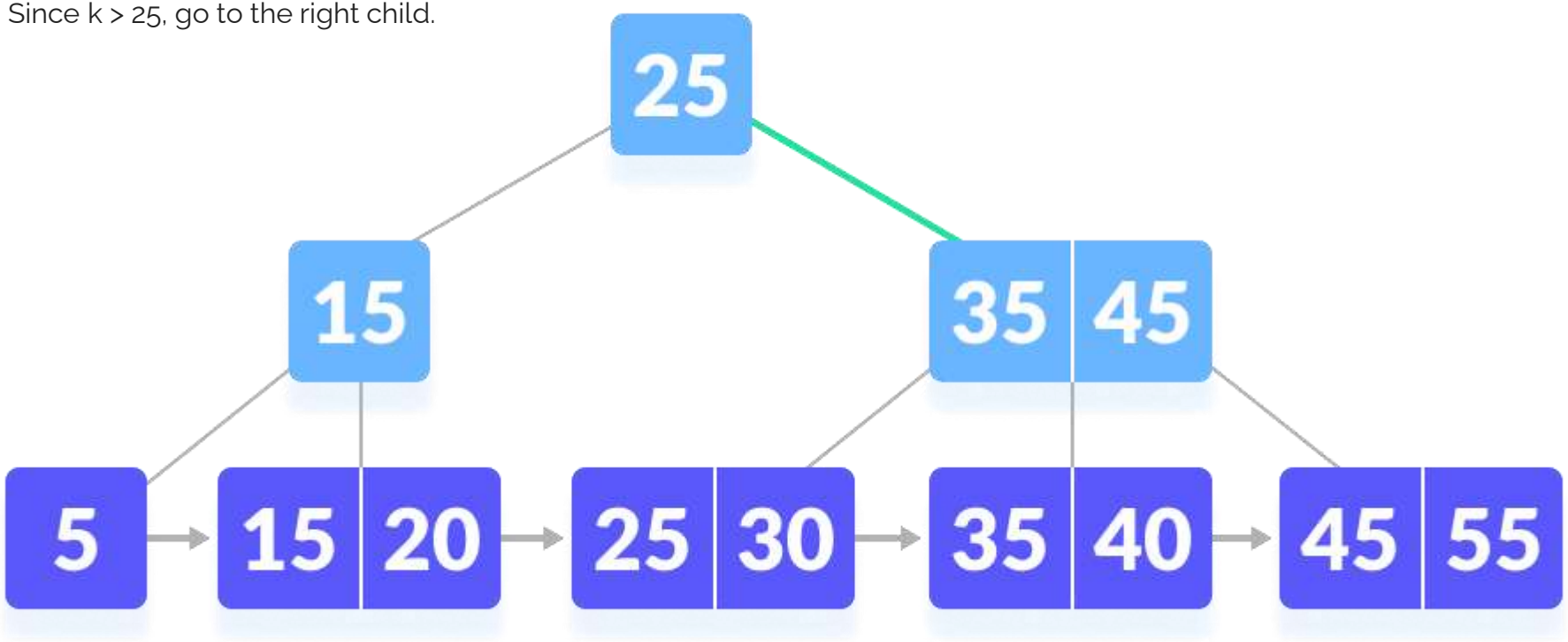
Searching on a B+ Tree

Compare k with the root node.



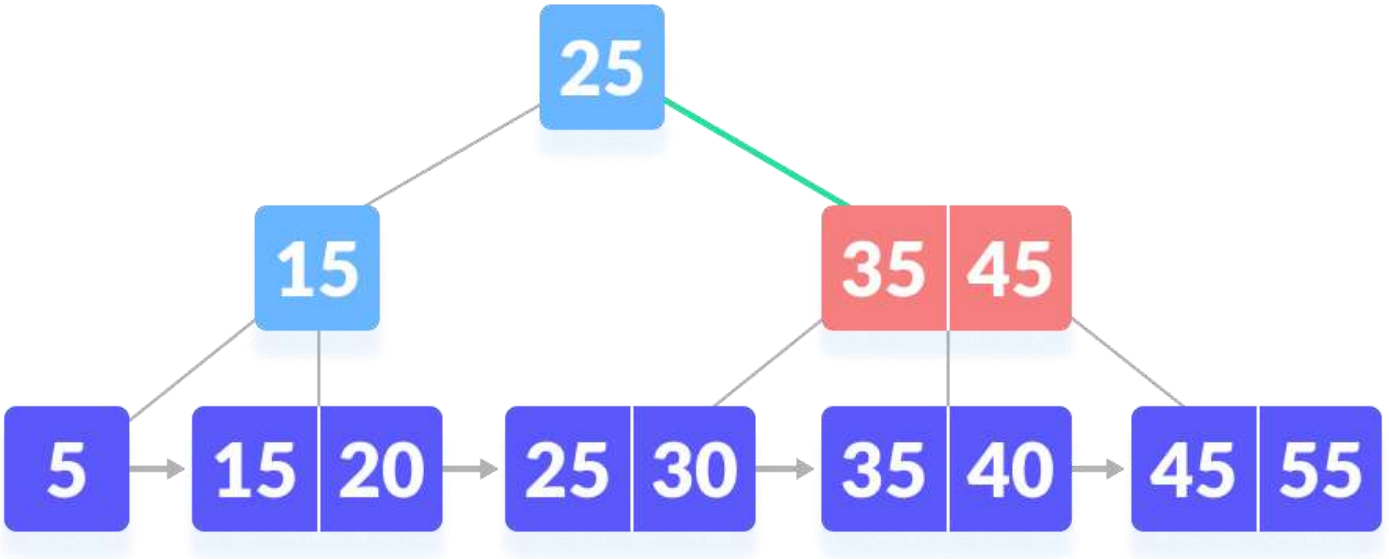
Searching on a B+ Tree

Since k > 25, go to the right child.



Searching on a B+ Tree

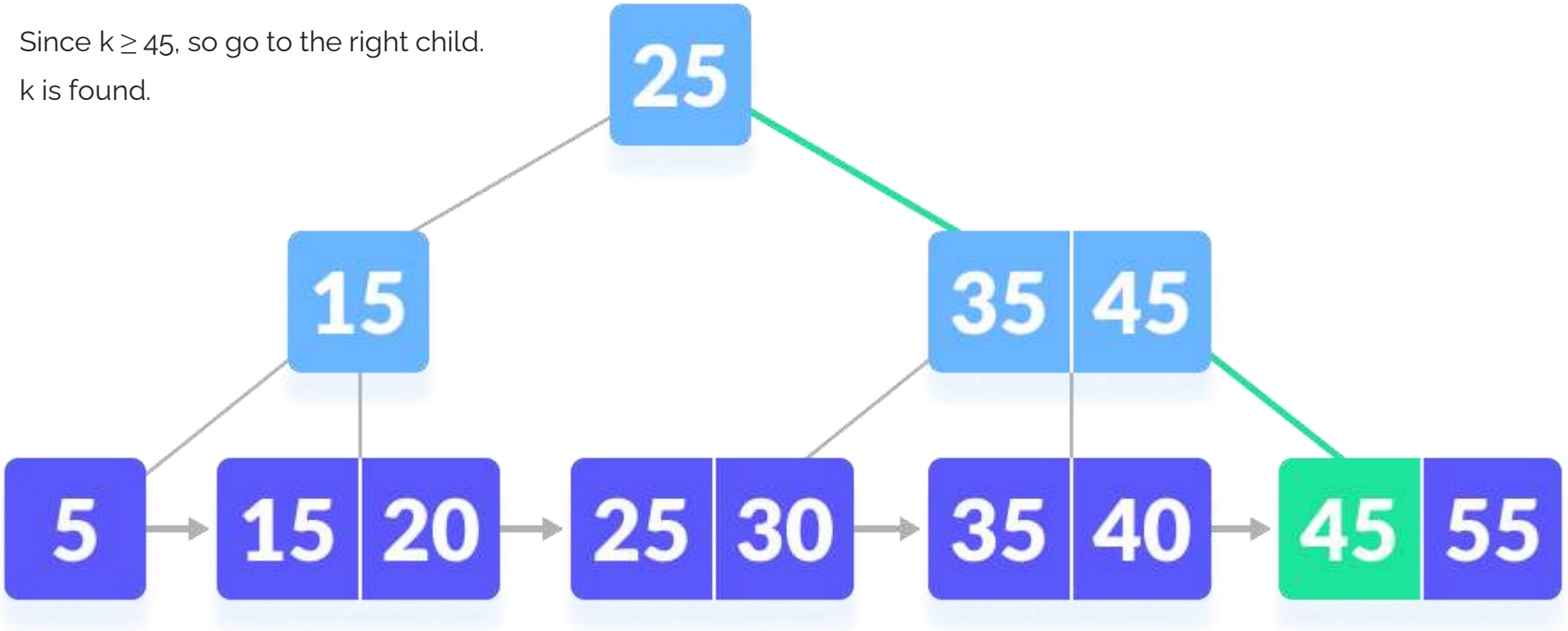
Compare k with 35. Since k > 35, compare k with 45.



Searching on a B+ Tree

Since k ≥ 45, so go to the right child.

k is found.



Insertion on a B+ Tree

- Inserting an element into a B+ tree consists of three main events: **searching the appropriate leaf**, **inserting** the element and **balancing/splitting** the tree.

Insertion Operation

- Before inserting an element into a B+ tree, these properties must be kept in mind.
- The root has at least two children.
- Each node except root can have a maximum of m children and at least m/2 children.
- Each node can contain a maximum of m - 1 keys and a minimum of [m/2] - 1 keys.

The following steps are followed for inserting an element.

- Since every element is inserted into the leaf node, go to the appropriate leaf node.
- Insert the key into the leaf node.

Insertion on a B+ Tree

Case I

- If the leaf is not full, insert the key into the leaf node in increasing order.

Case II

- If the leaf is full, insert the key into the leaf node in increasing order and balance the tree in the following way.
- Break the node at m/2th position.
- Add m/2th key to the parent node as well.
- If the parent node is already full, follow steps 2 to 3.

Insertion on a B+ Tree

- **Insertion Example:** The elements to be inserted are 5,15, 25, 35, 45.
- Insert 5.

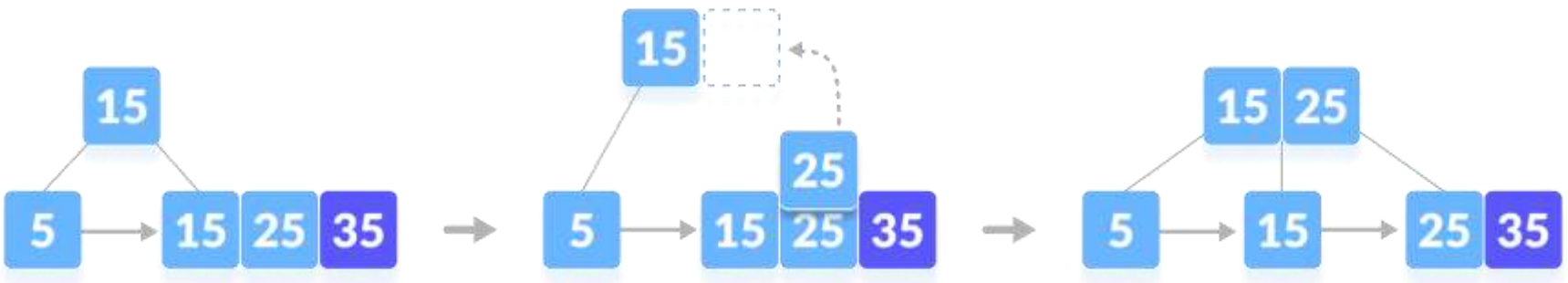
- Insert 15.

Insertion on a B+ Tree

- Insert 25.

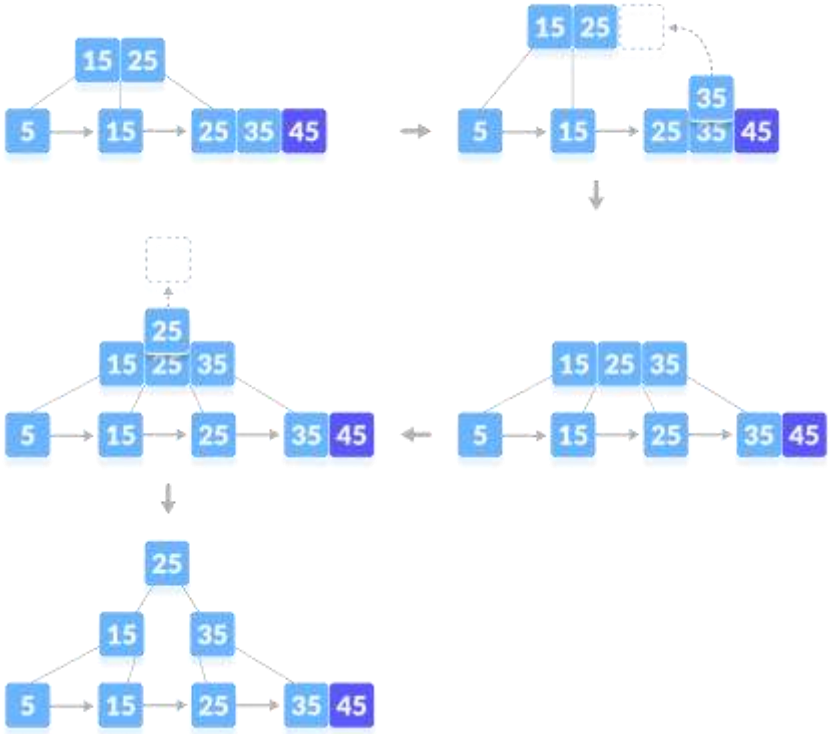
Insertion on a B+ Tree

- Insert 35.



Insertion on a B+ Tree

- Insert 45.



Insertion on a B+ Tree

- Insert keys: **1,4,7,10,17,21,31,25,19,20,28,42**
- **Order (m) = 4**

Insertion on a B+ Tree

- Insert keys: **7,10,1,23,5,15,17,9,11,39,35,8,40,25**
- **Order (m) = 5**

Deletion on a B+ Tree

- Deleting an element on a B+ tree consists of three main events: **searching** the node where the key to be deleted exists, deleting the key and balancing the tree if required. **Underflow** is a situation when there is less number of keys in a node than the minimum number of keys it should hold.

Deletion Operation

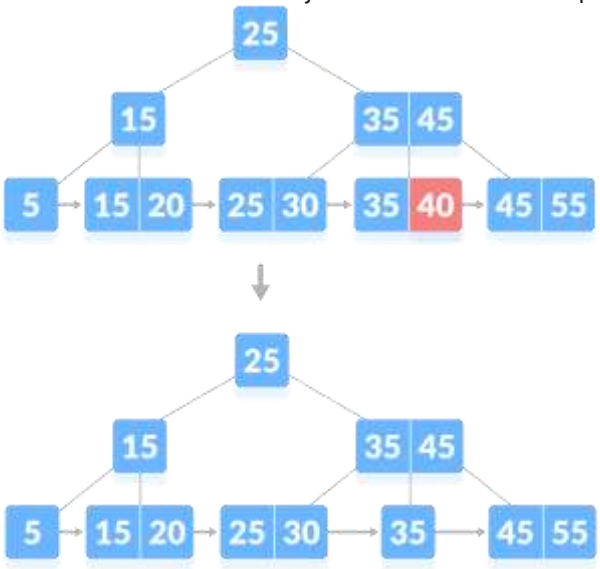
- Before going through the steps below, one must know these facts about a B+ tree of degree **m**.
- A node can have a maximum of m children. (i.e. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of ⌈m/2⌉ children. (i.e. 2)
- A node (except root node) should contain a minimum of ⌈m/2⌉ - 1 keys. (i.e. 1)
- While deleting a key, we have to take care of the keys present in the internal nodes (i.e. indexes) as well because the values are redundant in a B+ tree.

Deletion on a B+ Tree

Case I

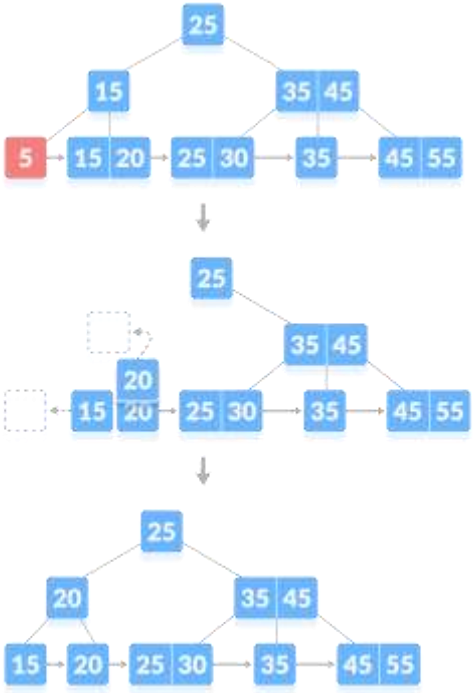
- The key to be deleted is present only at the leaf node not in the indexes (or internal nodes).
- There are two cases for it:

1. There is more than the minimum number of keys in the node. Simply delete the key



Deletion on a B+ Tree

2. There is an exact minimum number of keys in the node. Delete the key and borrow a key from the immediate sibling. Add the median key of the sibling node to the parent.



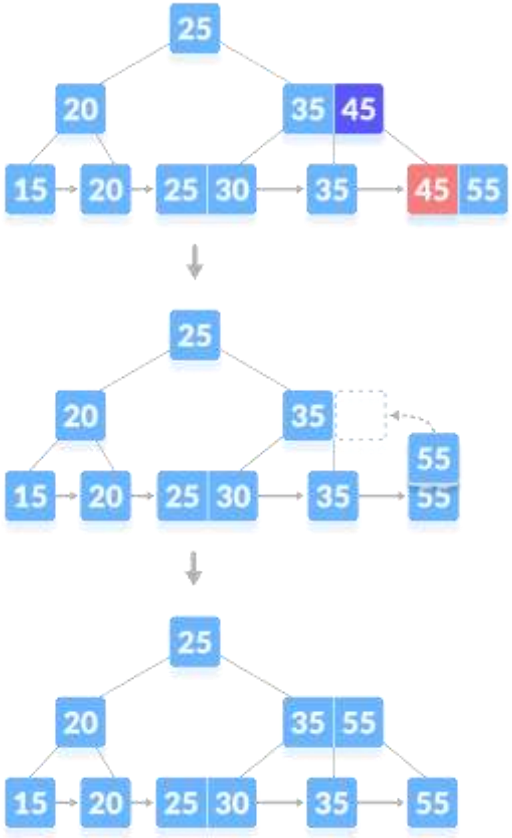
Deletion on a B+ Tree

Case II

- The key to be deleted is present in the internal nodes as well. Then we have to remove them from the internal nodes as well. There are the following cases for this situation.

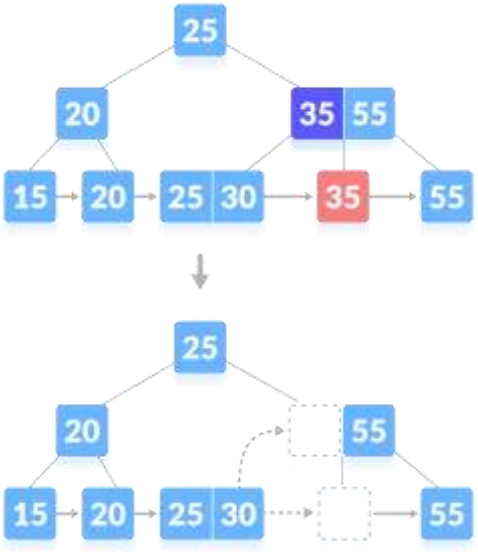
1. If there is more than the minimum number of keys in the node, simply delete the key from the leaf node and delete the key from the internal node as well.
Fill the empty space in the internal node with the inorder successor.

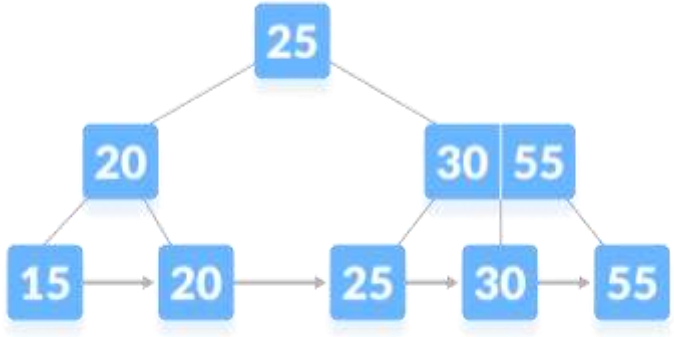
Deletion on a B+ Tree



Deletion on a B+ Tree

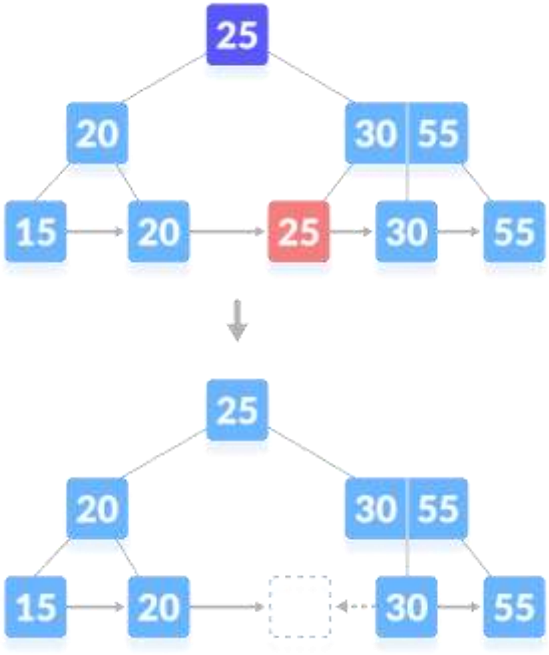
2. If there is an exact minimum number of keys in the node, then delete the key and borrow a key from its immediate sibling (through the parent).
Fill the empty space created in the index (internal node) with the borrowed key.

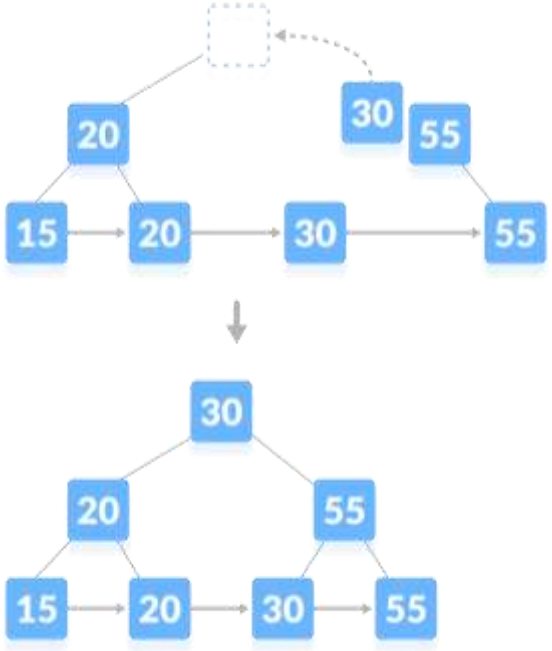




Deletion on a B+ Tree

3. This case is similar to Case II(1) but here, empty space is generated above the immediate parent node. After deleting the key, merge the empty space with its sibling.
Fill the empty space in the grandparent node with the inorder successor.

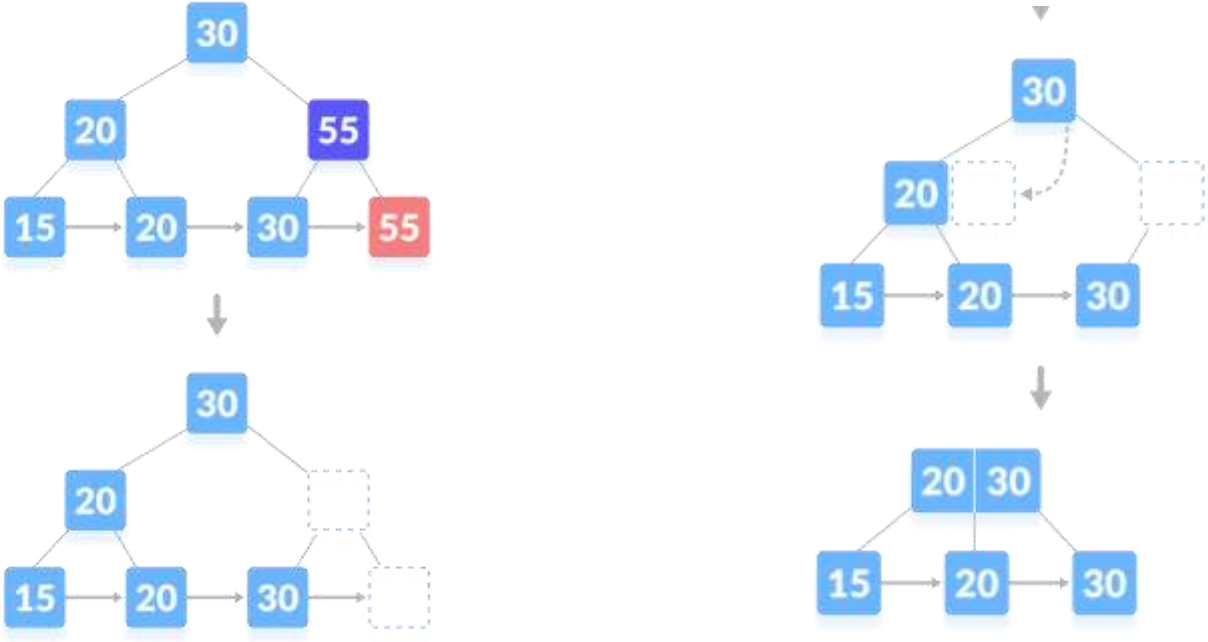




Deletion on a B+ Tree

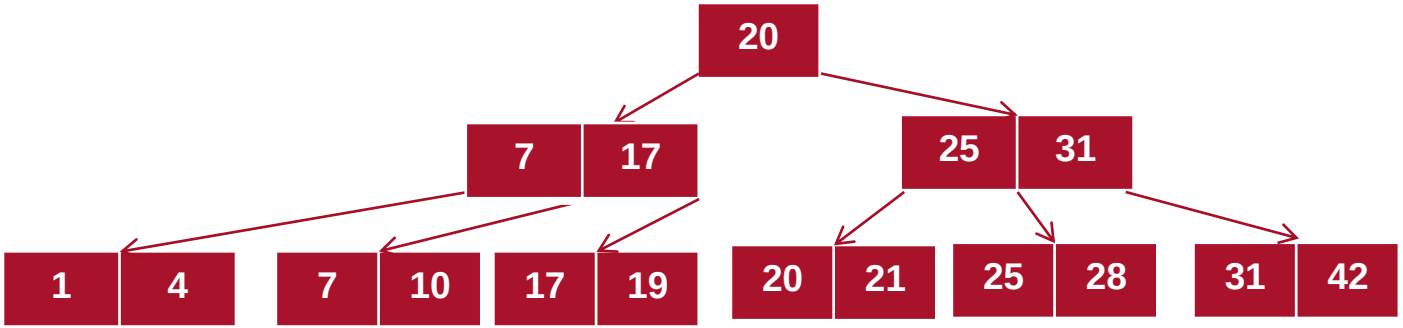
Case III

- In this case, the height of the tree gets shrinked. It is a little complicated.Deleting 55 from the tree below leads to this condition. It can be understood in the illustrations below.



Deletion on a B+ Tree

Delete keys = 21,31,20,10,7,25,42,4



Thank You