CS 2124: DATA STRUCTURES
Spring 2024

Topics: **B-Trees**

Topics

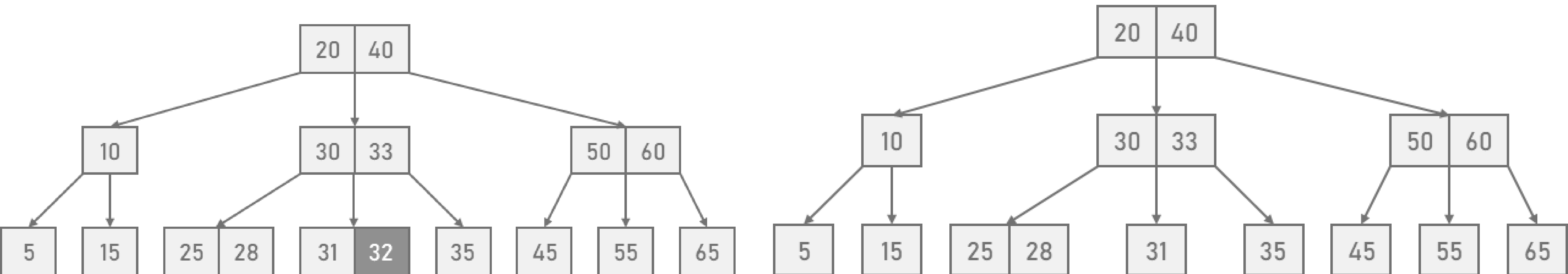
- Multiway Tree (M-way tree)
    - Multiway Search Tree
- B-Tree (Height-balanced m-way tree/Balanced Tree)
    - Application of B-tree
    - B-Tree Properties
    - B-Tree Order
    - Creating a B-Tree
    - B-Tree Operations
    - Advantages and Disadvantages of B-tree
- RB Tree (Red Black Tree)
    - RB Tree Properties
    - Creating RB Tree
    - RB Tree Operations
    - RB Tree Rotations

B-Tree (Deletion)

The following are **three prominent cases** of the deletion operation in a B Tree:

Case 1: The Deletion/Removal of the key lies in the Leaf node - This case is further divided into two different cases:

1. The deletion/removal of the key does not violate the property of the minimum number of keys a node should hold.
   - Let us visualize this case using an example where we will delete key 32 from the following B Tree.



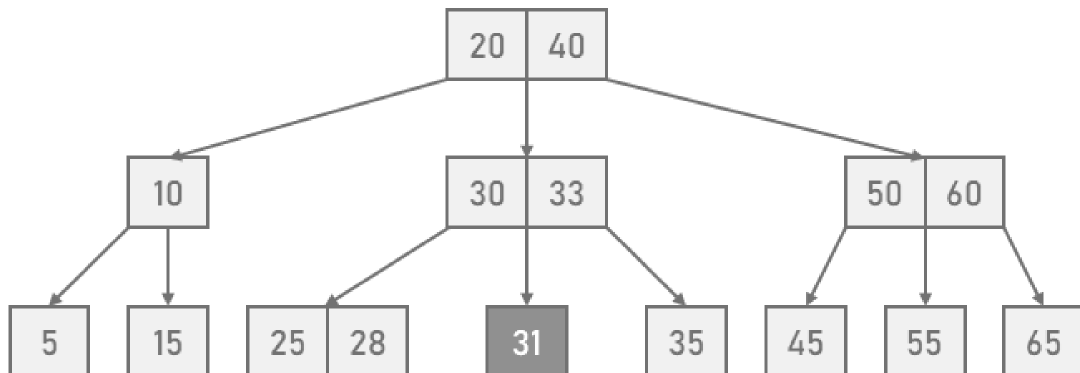
B-Tree (Deletion)

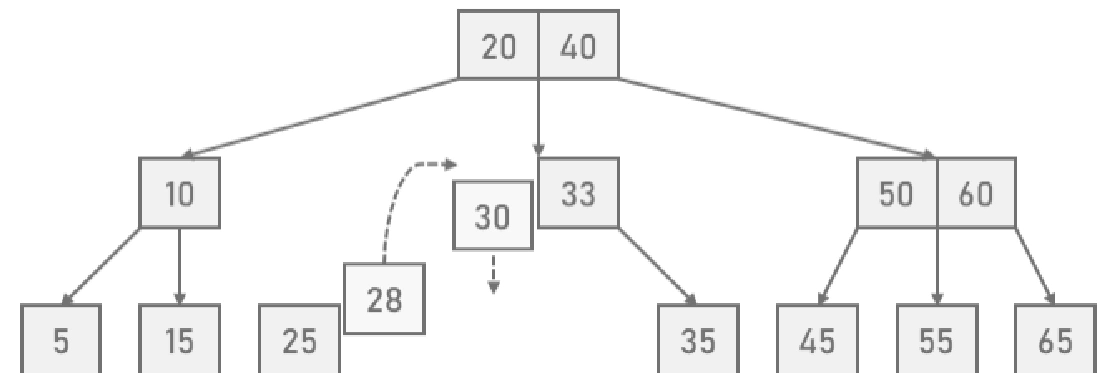
The following are **three prominent cases** of the deletion operation in a B Tree:

Case 1: The Deletion/Removal of the key lies in the Leaf node - This case is further divided into two different cases:

2. The deletion/removal of the key violates the property of the minimum number of keys a node should hold. In this case, we must borrow a key from its proximate sibling node in the order of Left to Right.

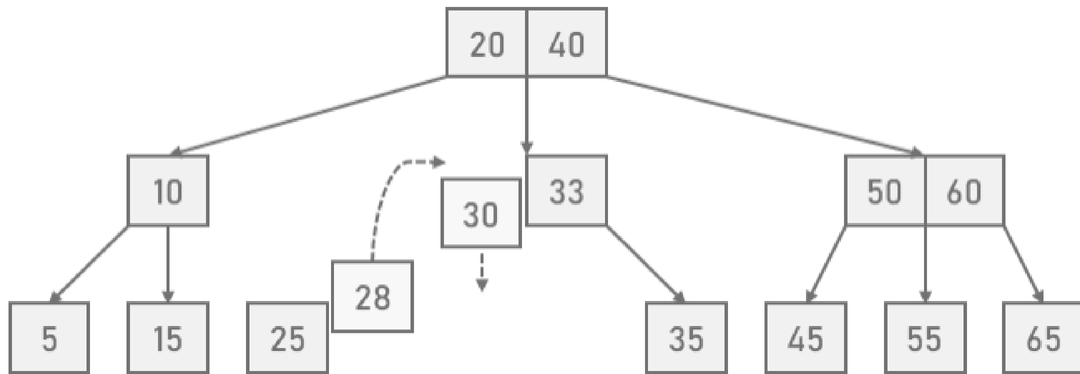
- Firstly, we will visit the proximate Left sibling. If the Left sibling node has more than a minimum number of keys, it will borrow a key from this node.
- Else, we will check to borrow from the proximate Right sibling node.
- Let us now visualize this case with the help of an example where we will delete 31 from the above B Tree. We will borrow a key from the left sibling node in this case.

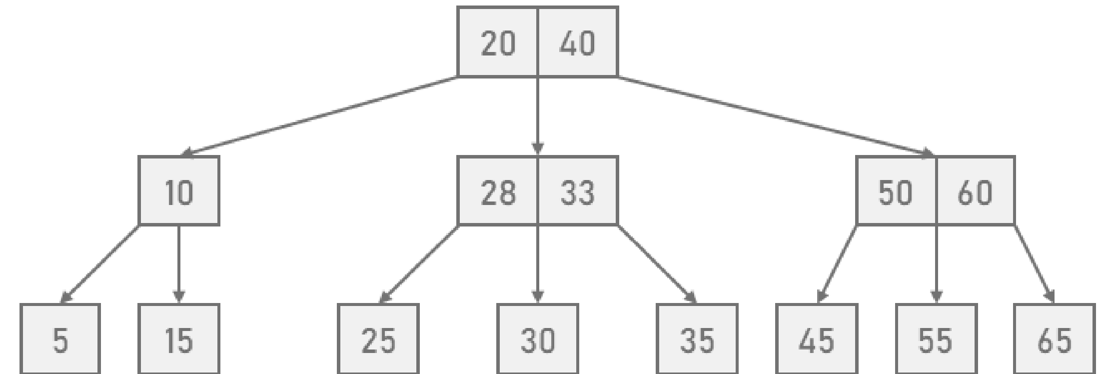




B-Tree (Deletion)

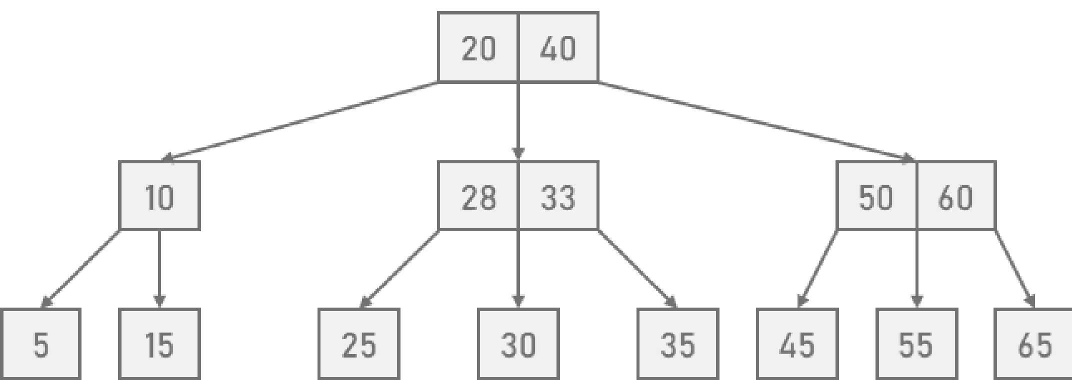
Case 1.2:

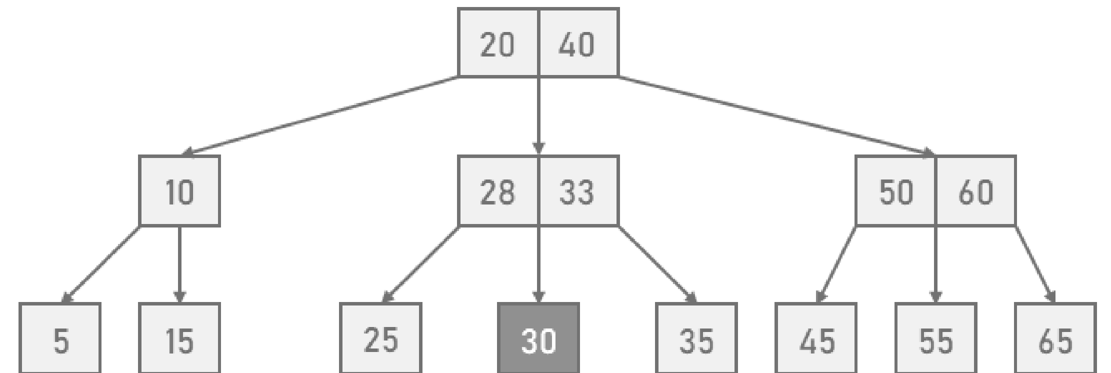




B-Tree (Deletion)

Case 1.2:

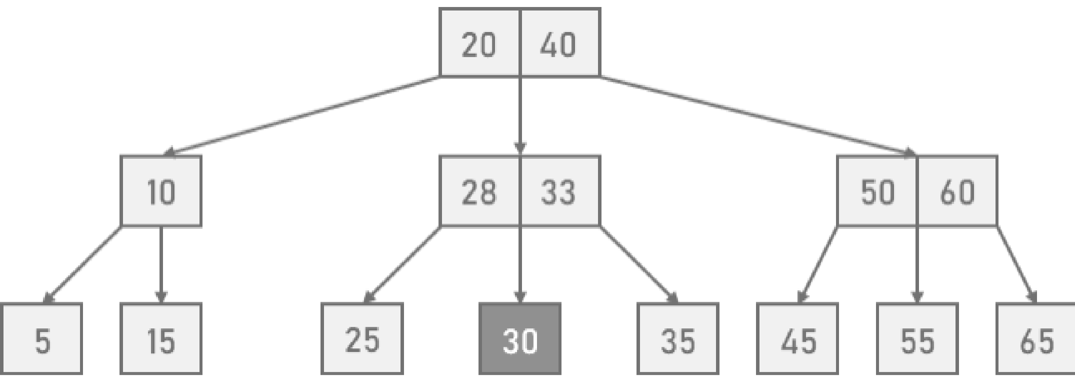


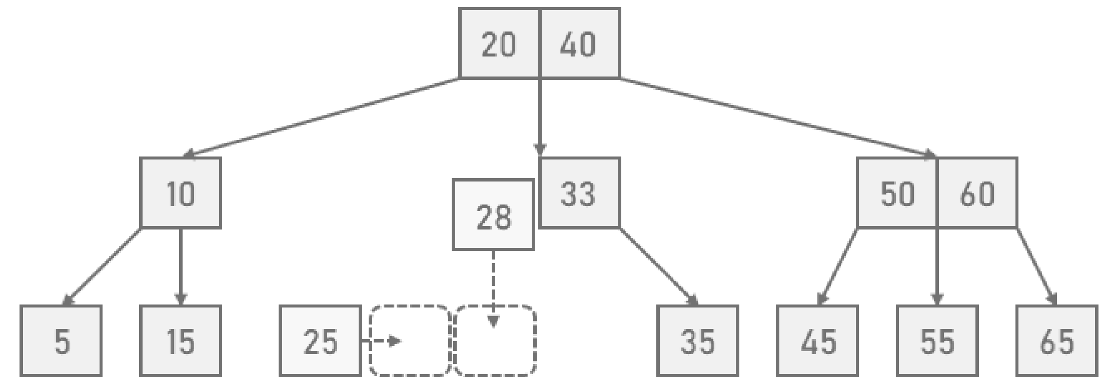


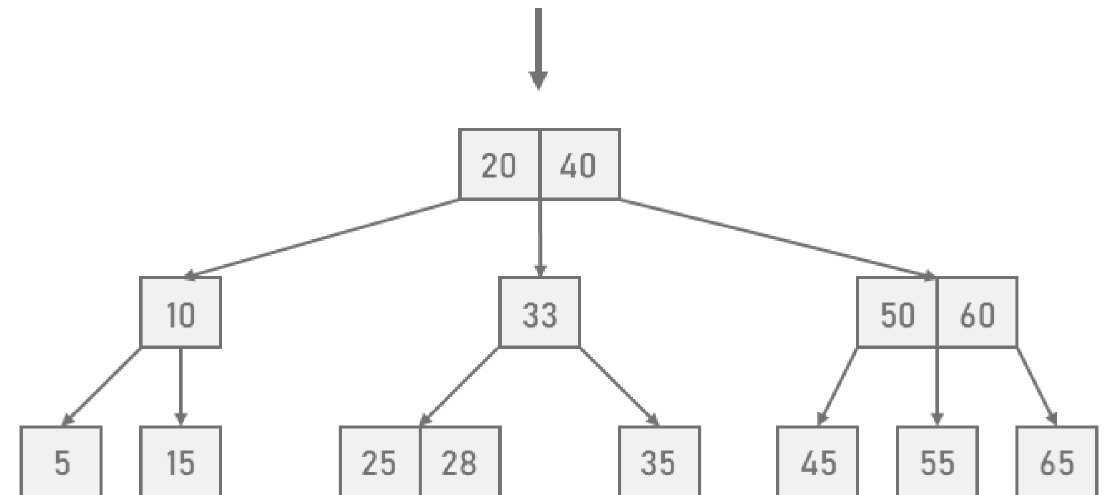
- If both the proximate sibling nodes already consist of a minimum number of keys, then we will merge the node with either the left sibling node or the right one.
- This process of merging is done through the parent node.

B-Tree (Deletion)

Case 1.2:



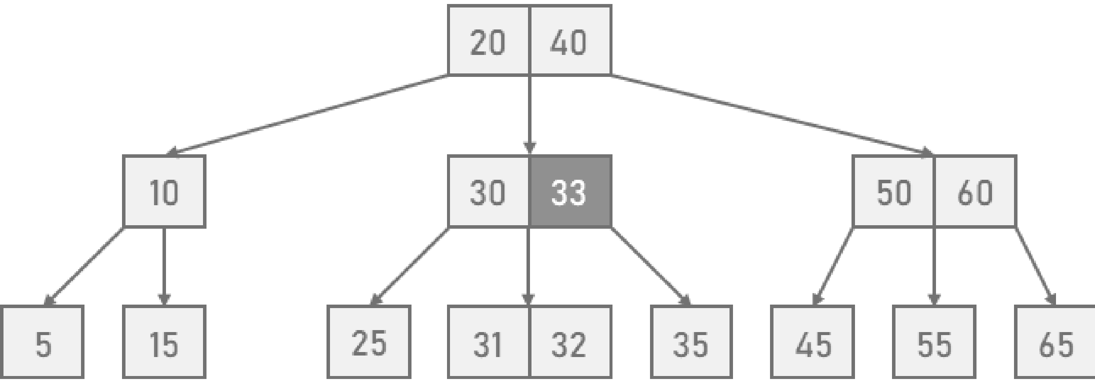


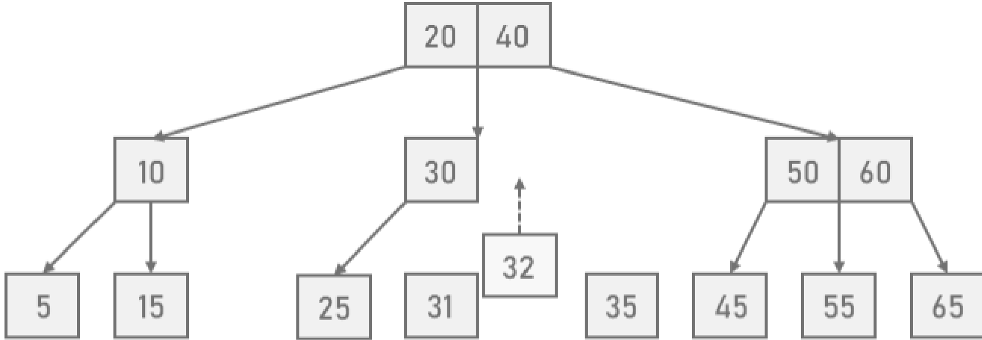


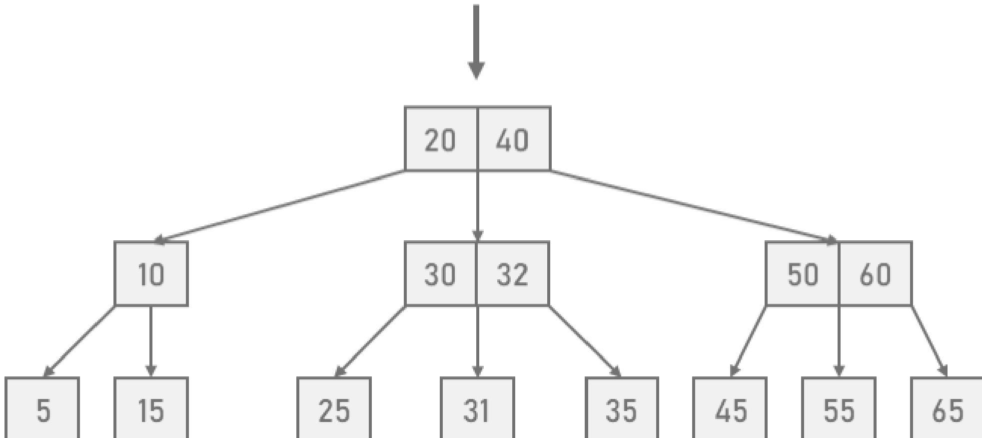
B-Tree (Deletion)

- **Case 2:** The Deleting/Removal of the key lies in the non-Leaf node - This case is further divided into different cases:

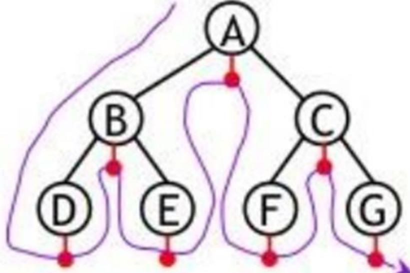
1. The non-Leaf/Internal node, which is removed, is replaced by an in-order predecessor if the Left child node has more than the minimum number of keys.







In-order
1. Left Subtree,
2. Root,
3. Right Subtree

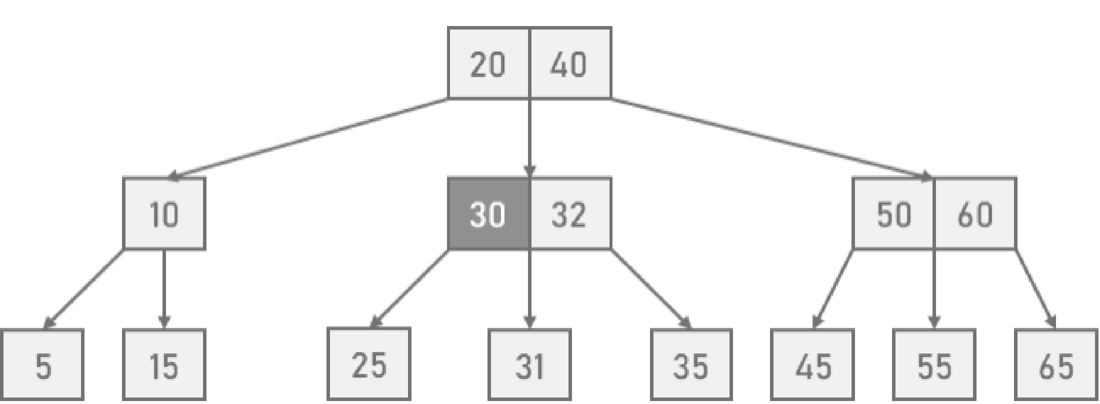


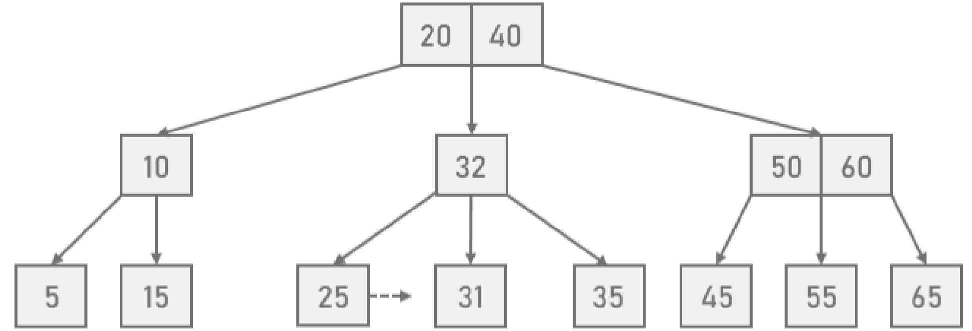
B-Tree (Deletion)

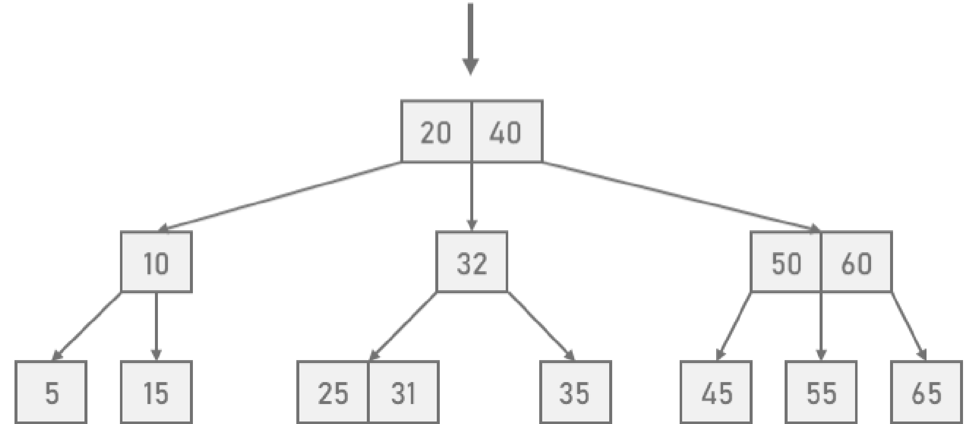
- **Case 2:** The Deleting/Removal of the key lies in the non-Leaf node - This case is further divided into different cases:

2. The non-Leaf/Internal node, which is removed, is replaced by an in-order successor if the Right child node has more than the minimum number of keys.

3. If either child has a minimum number of keys, then we will merge the Left and the Right child nodes.

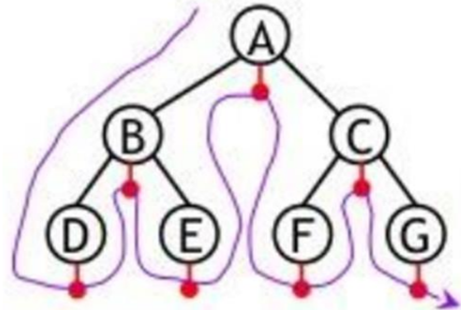






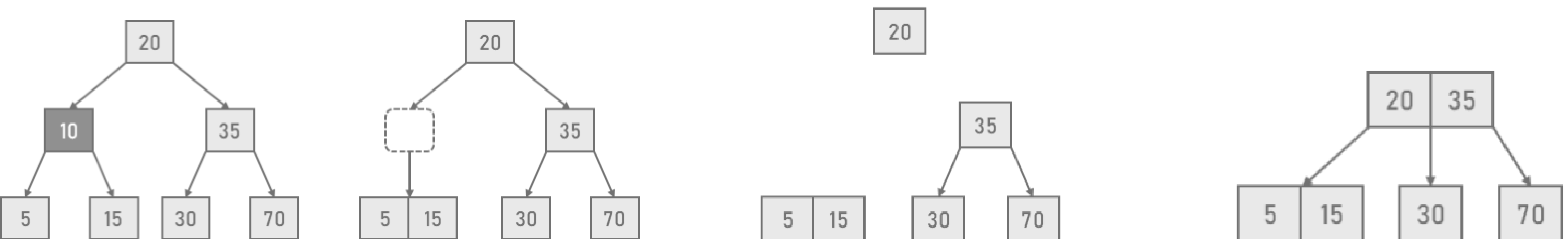
In-order

1. Left Subtree,
2. Root,
3. Right Subtree



B-Tree (Deletion)

- **Case 3:** In the following case, the tree's height shrinks.
- If the target key lies in an Internal node, and the removal of the key leads to fewer keys in the node (which is less than the minimum necessitated), then look for the in-order predecessor and the in-order successor.
- If both the children have a minimum number of keys, then borrowing can't occur.
  - This leads to Case 2(3), i.e., merging the child nodes.
- We will again look for the sibling to borrow a key.
- However, if the sibling also consists of a minimum number of keys, then we will merge the node with the sibling along with the parent node and arrange the child nodes as per the requirements (ascending order).



Search Operation in B-Tree

- **Step 1** - Read the search element from the user.
- **Step 2** - Compare the search element with first key value of root node in the tree.
- **Step 3** - If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4** - If both are not matched, then check whether search element is smaller or larger than that key value.
- **Step 5** - If search element is smaller, then continue the search process in left subtree.
- **Step 6** - If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7** - If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

Search Operation in B-Tree

```
111 // Search node
112 void search(int val, int *pos, struct BTreeNode *myNode) {
113   if (!myNode) {
114     return;
115   }
116   if (val < myNode->val[1]) {
117     *pos = 0;
118   } else {
119     for (*pos = myNode->count;
120         (val < myNode->val[*pos] && *pos > 1); (*pos)--)
121       ;
122     if (val == myNode->val[*pos]) {
123       printf("%d is found", val);
124       return;
125     }
126   }
127   search(val, pos, myNode->link[*pos]);
128   return;
129 }
```

Note: This Code is not part of quiz/exam

Advantages and Disadvantages of B-Trees

- Advantages of B-Trees:
    - B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
    - B-Trees are self-balancing.
    - High-concurrency and high-throughput.
    - Efficient storage utilization.
- Disadvantages of B-Trees:
    - B-Trees are based on disk-based data structures and can have a high disk usage.
    - Not the best for all cases.
    - Slow in comparison to other data structures.

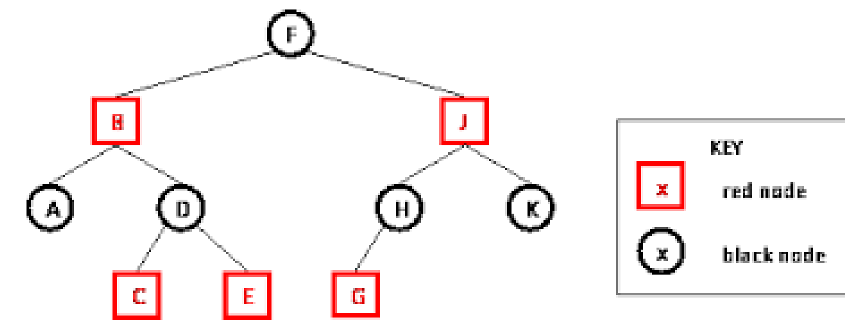
Red - Black (RB) Tree

Topics:

RB Tree (Red Black Tree)

- RB Tree Properties
- RB Tree Applications
- RB Tree Rotations
- Creating RB Tree
- RB Tree Operations (Case 1 & 2 only for deletion)

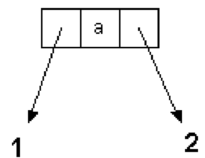
Red - Black (RB) Tree

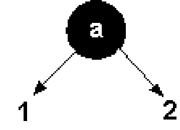


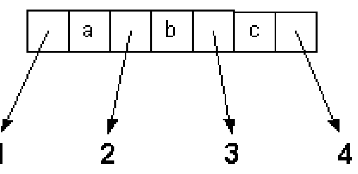
- Red - Black Tree is another variant of Binary Search Tree in which every node is colored either RED or BLACK.
- We can define a Red Black Tree as “Red Black Tree is a Binary Search Tree in which every node is colored either RED or BLACK.”
- Compared to other self-balancing binary search trees, the nodes in a red-black tree hold an extra bit called "color" representing "red" and "black" which is used when re-organising the tree to ensure that it is always approximately balanced.

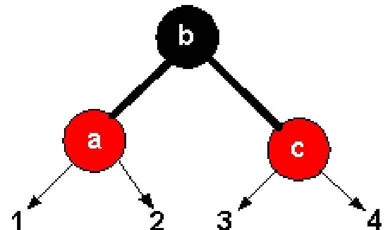
2-3-4 Nodes

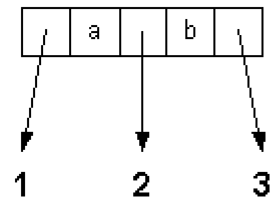
Red-Black Nodes

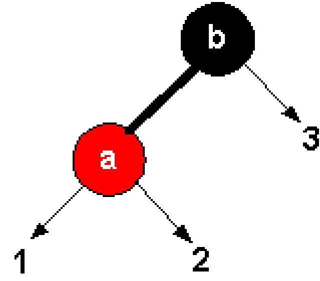




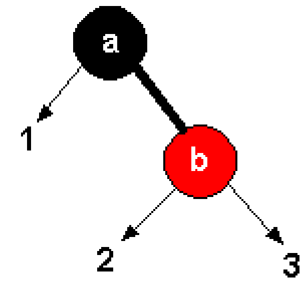






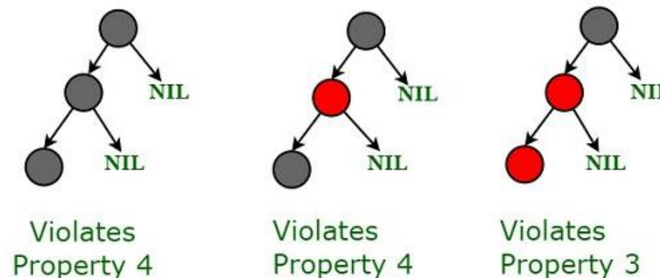


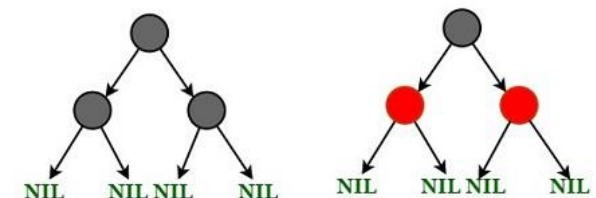
or



Red - Black (RB) Tree Propoerties

- In Red Black Tree, the color of a node is decided based on the properties of Red-Black Tree. Every Red Black Tree has the following properties.
    1. Property #1: Red - Black Tree must be a Binary Search Tree.
    2. Property #2: The ROOT node must be colored BLACK.
    3. Property #3: The children of Red colored node must be colored BLACK. (There should not be two consecutive RED nodes).
    4. Property #4: In all the paths of the tree, there should be same number of BLACK colored nodes.
    5. Property #5: Every new node must be inserted with RED color.
    6. Property #6: Every leaf (i.e. NULL node) must be colored BLACK.





Red - Black (RB) Tree Applications

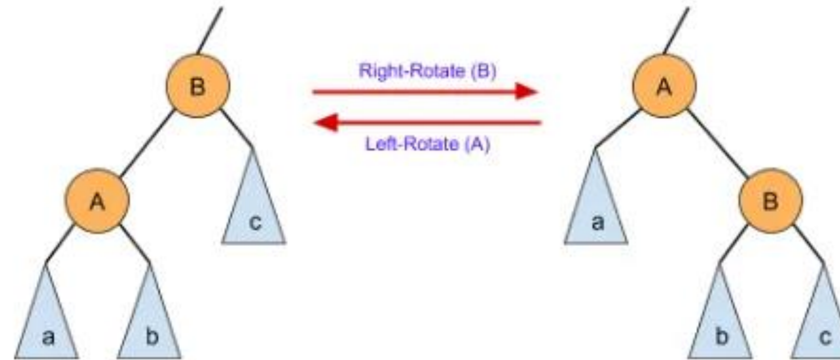
- Red-Black trees can be used to efficiently index data in databases, allowing for fast search and retrieval of data.
- Red-Black trees can be used to efficiently implement graph algorithms such as Dijkstra's shortest path algorithm
- Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing. AVL trees provide complex insertion and removal operations as more rotations are done due to relatively strict balancing.

- The RB tree is use in Linux Kernel scheduling
- To keep track of virtual memory segments for a process. The start address of the range serves as the key.
- RB-trees are particularly valuable in functional programming*

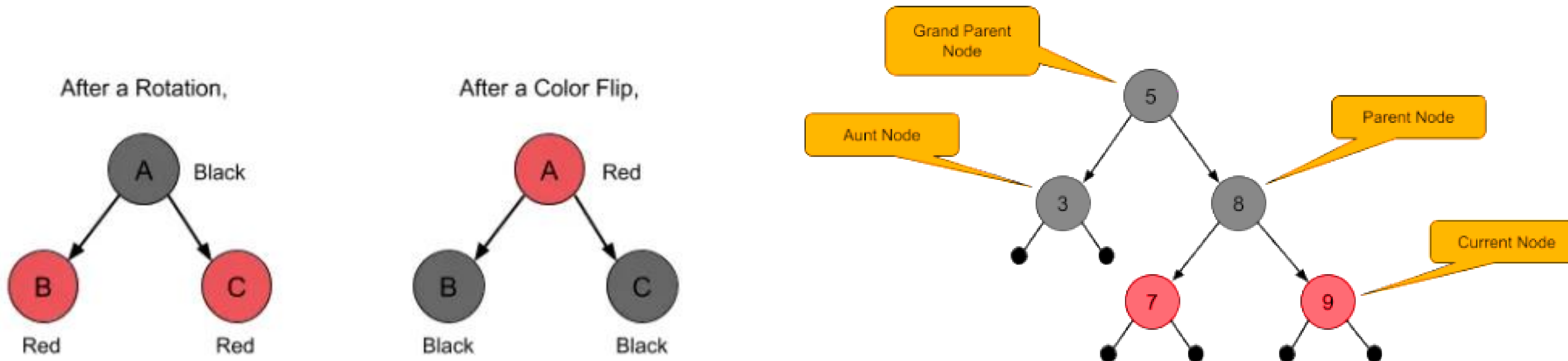
**Functional programming* is a paradigm of building computer programs using expressions and functions without mutating state and data.*

Rotation Red-Black (RB) Tree

- A rotation is a special operation designed for self-balancing Binary Search Trees which can be performed in O(1) time. An interesting property on rotations is that it preserves the in-order traversing of the keys (ordering of the keys).



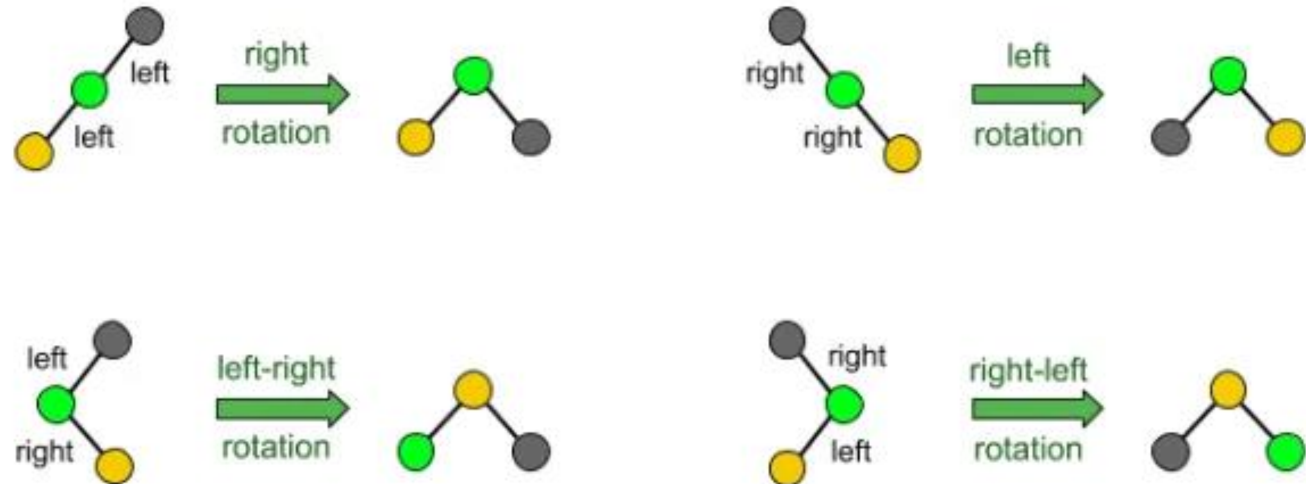
- The decision to perform a rotation or a color change is based on the aunt of the considered node (the current node). If the node has a Black Aunt, we do a rotation. If the node has a Red Aunt, we do a color flip. After performing a rotation, we should color fix the tree. After performing those operations, the tree should be ended like below.



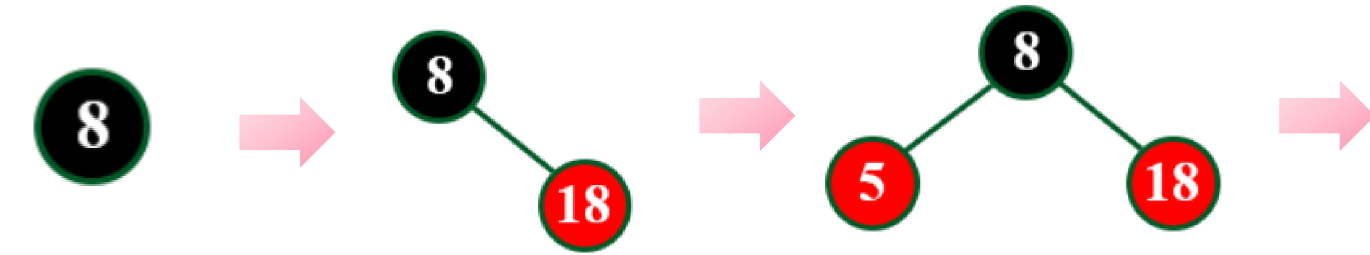
Source

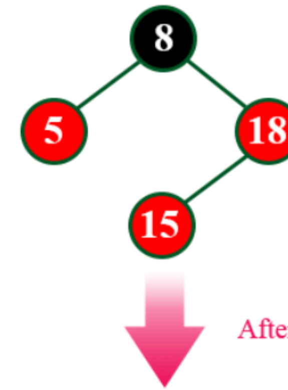
Rotation Red-Black (RB) Tree

1. If the current node is the left child of its grandparent's left child, we right rotate the grandparent around the parent.
2. If the current node is the right child of its grandparent's right child, we left rotate the grandparent around the parent.
3. If the current node is the right child of its grandparent's left child, we perform a left-right rotation where we rotate the grandparent and the child around the parent.
4. If the current node is the left child of its grandparent's right child, we perform a right-left rotation where we rotate the grandparent and the child around the parent.



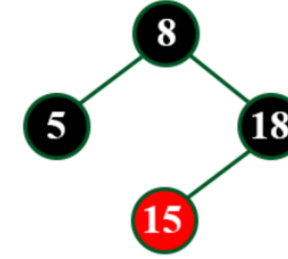
Create a RED BLACK Tree by inserting following sequence of number
8, 18, 5, 15, 17, 25, 40 & 80.



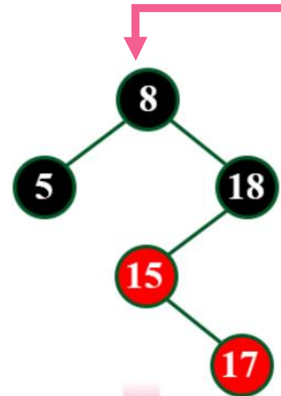


Here there are two consecutive Red nodes (18 & 15).
The newnode's parent sibling color is Red
and parent's parent is root node.
So we use RECOLOR to make it Red Black Tree.

After RECOLOR

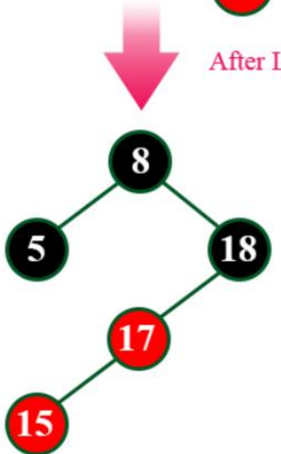


After Recolor operation, the tree is satisfying all Red Black Tree properties.

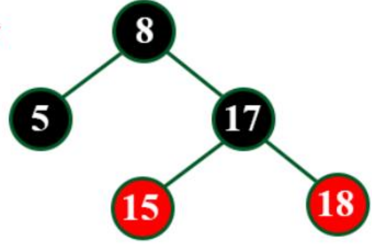


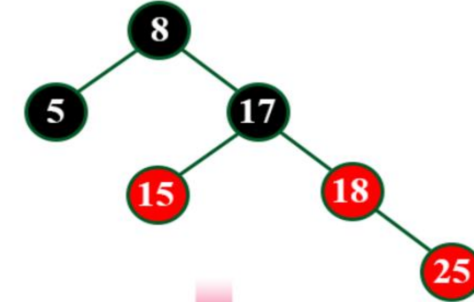
Here there are two consecutive Red nodes (15 & 17).
The newnode's parent sibling is NULL. So we need rotation.
Here, we need LR Rotation & Recolor.

After Left Rotation



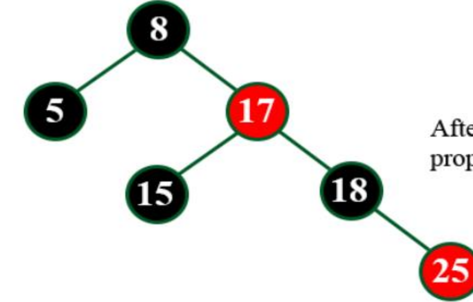
After Right Rotation & Recolor



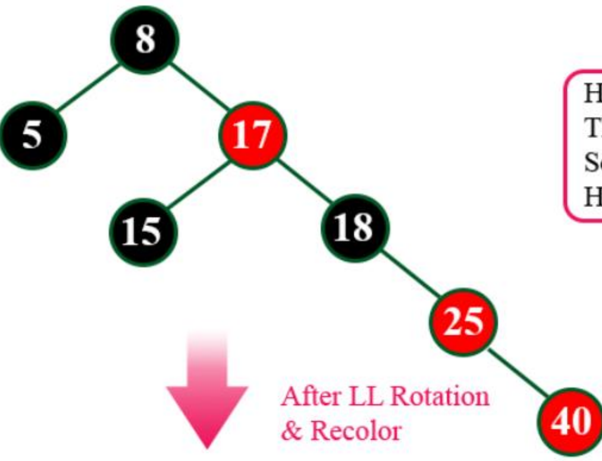


Here there are two consecutive Red nodes (18 & 25).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor

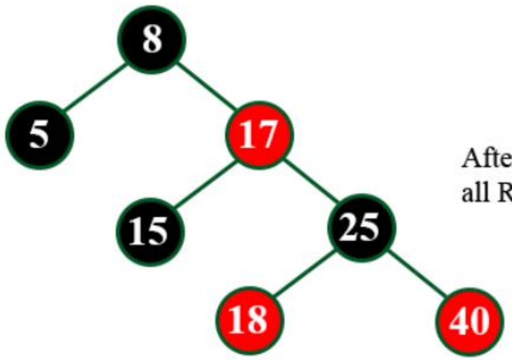


After Recolor operation, the tree is satisfying all Red Black Tree properties.

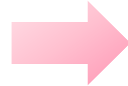


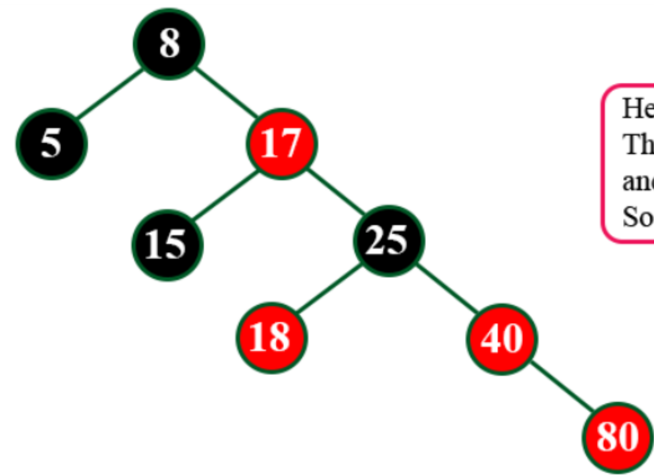
Here there are two consecutive Red nodes (25 & 40).
The newnode's parent sibling is NULL
So we need a Rotation & Recolor.
Here, we use LL Rotation and Recheck.

After LL Rotation & Recolor



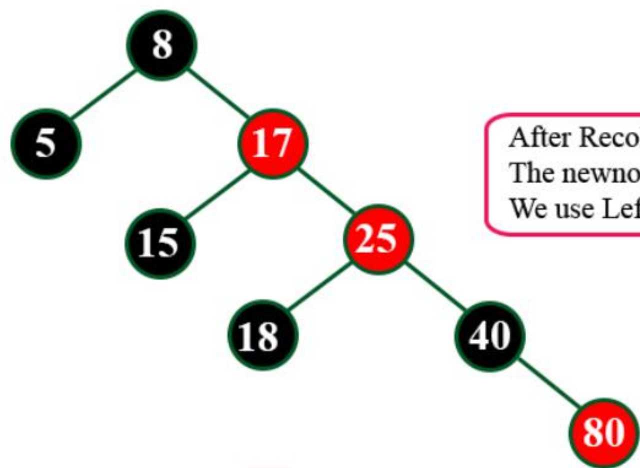
After LL Rotation & Recolor operation, the tree is satisfying all Red Black Tree properties.





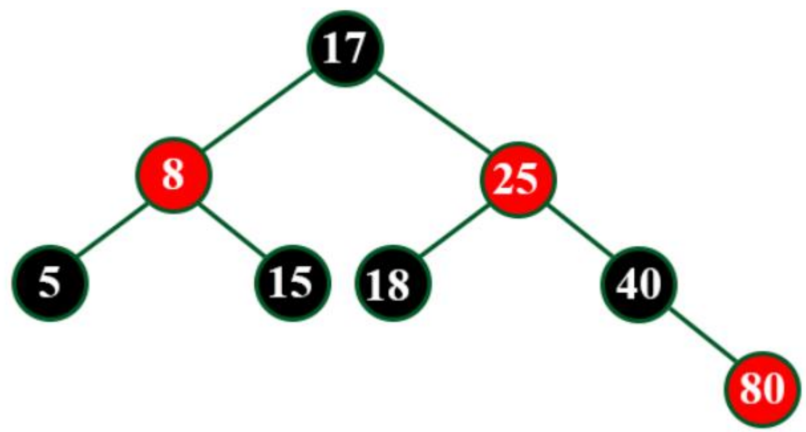
Here there are two consecutive Red nodes (40 & 80).
The newnode's parent sibling color is Red
and parent's parent is not root node.
So we use RECOLOR and Recheck.

After Recolor



After Recolor again there are two consecutive Red nodes (17 & 25).
The newnode's parent sibling color is Black. So we need Rotation.
We use Left Rotation & Recolor.

After Left Rotation & Recolor



Insertion into RED BLACK (RB) Tree

- In a Red-Black Tree, every new node must be inserted with the color RED. The insertion operation in Red Black Tree is similar to insertion operation in Binary Search Tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of Red-Black Tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it Red Black Tree.
    1. Recolor
    2. Rotation
    3. Rotation followed by Recolor
- The insertion operation in Red Black tree is performed using the following steps...
    1. Step 1 - Check whether tree is Empty.
    2. Step 2 - If tree is Empty then insert the new Node as Root node with color Black and exit from the operation.
    3. Step 3 - If tree is not Empty then insert the new Node as leaf node with color Red.
    4. Step 4 - If the parent of new Node is Black then exit from the operation.
    5. Step 5 - If the parent of new Node is Red then check the color of parent node's sibling of new Node.
    6. Step 6 - If it is colored Black or NULL then make suitable Rotation and Recolor it.
    7. Step 7 - If it is colored Red then perform Recolor. Repeat the same until tree becomes Red Black Tree.

Insertion in Red-Black (RB) tree

- Now let's construct a simple RB-Tree by inserting below set of keys. Note that for simplicity, leaf nodes (NULL's) are ignored in below diagrams. Elements: 5, 8, 1, 10, 9, 15, 20

We should insert each key as a red node.

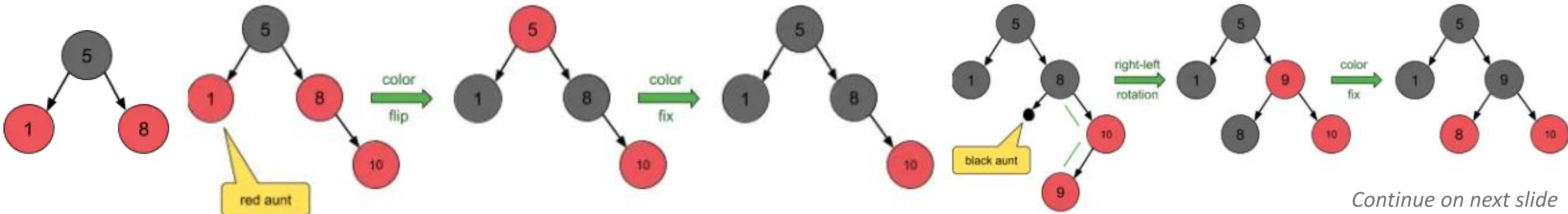
Therefore we insert 5 as a red node and immediately color it back to black as it's the root node.

Then we insert 8 as node 5's right child.
We follow the basic BST property when inserting keys.
Inserting 1 is the same as inserting 8.

Insert 10, but now both 8 and 10 are red so fix color of node 8.

Root node cannot be RED, so color fix at node 5

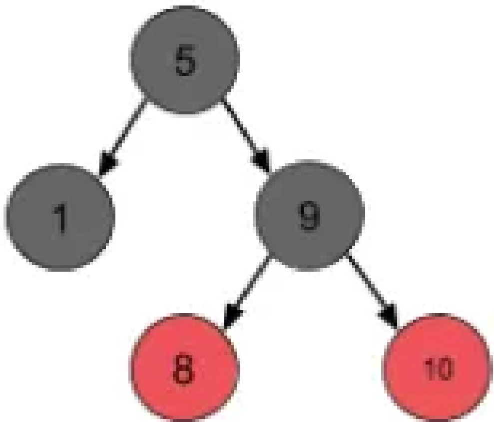
Add 9 as per BST property, but that makes the tree imbalanced.
Left rotation on node 8 and then color fix at node 9

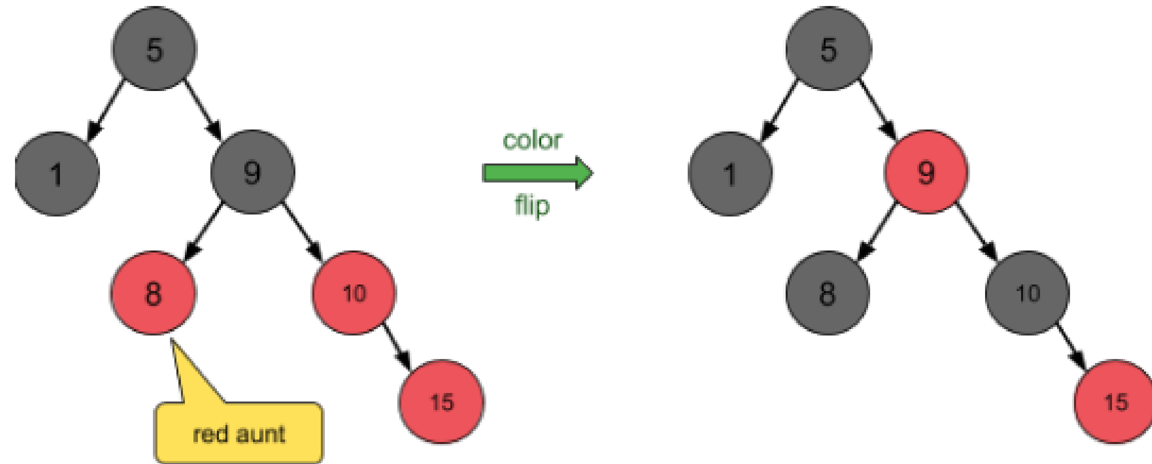


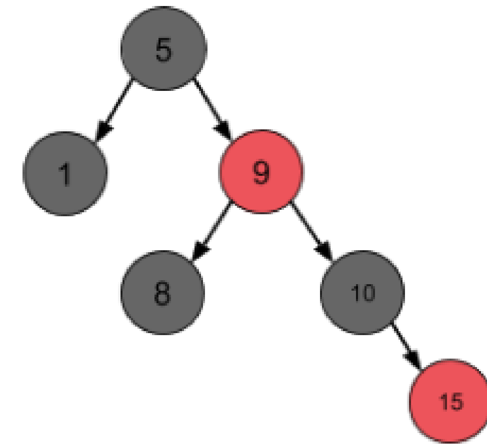
Continue on next slide

Insertion in RB tree

Elements: 5, 8, 1, 10, 9, 15, 20



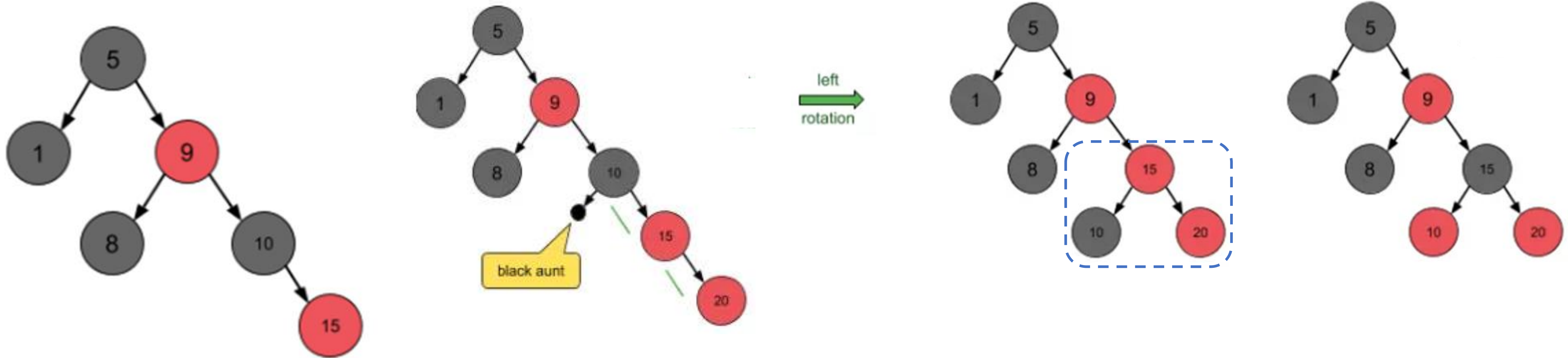




- Insert node 15
- But both nodes 10 & 15 are red
- Color flip

Insertion in RB tree

Elements: 5, 8, 1, 10, 9, 15, 20



- Insert node 20
- Left rotate at node 10 (right child of its grandparent’s right child, we left rotate the grandparent around the parent)
- Nodes 9, 15 & 20 are red
- Color fix at node 15

Deletion Operation in Red Black (RB) Tree

- The deletion operation in Red-Black Tree is similar to deletion operation in BST.
- But after every deletion operation, we need to check with the Red-Black Tree properties.
- If any of the properties are violated then make suitable operations like Recolor, Rotation and Rotation followed by Recolor to make it Red-Black Tree.

Steps for Deletion

1. If the node to be deleted has no children, simply remove it and update the parent node.
2. If the node to be deleted has only one child, replace the node with its child.
3. If the node to be deleted has two children, then replace the node with its in-order successor, which is the leftmost node in the right subtree. Then delete the in-order successor node as if it has at most one child.
4. After the node is deleted, the red-black properties might be violated. To restore these properties, some color changes and rotations are performed on the nodes in the tree. The changes are similar to those performed during insertion, but with different conditions.
5. The deletion operation in a red-black tree takes O(log n) time on average, making it a good choice for searching and deleting elements in large data sets.

Deletion in RB tree
(Total cases – 3)

- **Case 1:** Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child

- **Case 2: If either u or v is red,** we mark the replaced child as black (No change in black height).
  - Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.

- **Case 3: If Both u and v are Black**.
- **(3.1)**
  1. Color **u** as double black.
  2. Now our task reduces to convert this double black to single black.
  3. Note that If v is leaf, then u is NULL and color of NULL is considered black.
  4. So the deletion of a black leaf also causes a double black.

Deletion in RB tree

- **(3.2a)** Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.
  - **If sibling s is black and at least one of sibling's children is red**, perform rotation(s).
  - Let the red child of s be **r**.
  - This case can be divided in four subcases depending upon positions of s and r.

i. Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

ii. Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

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

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

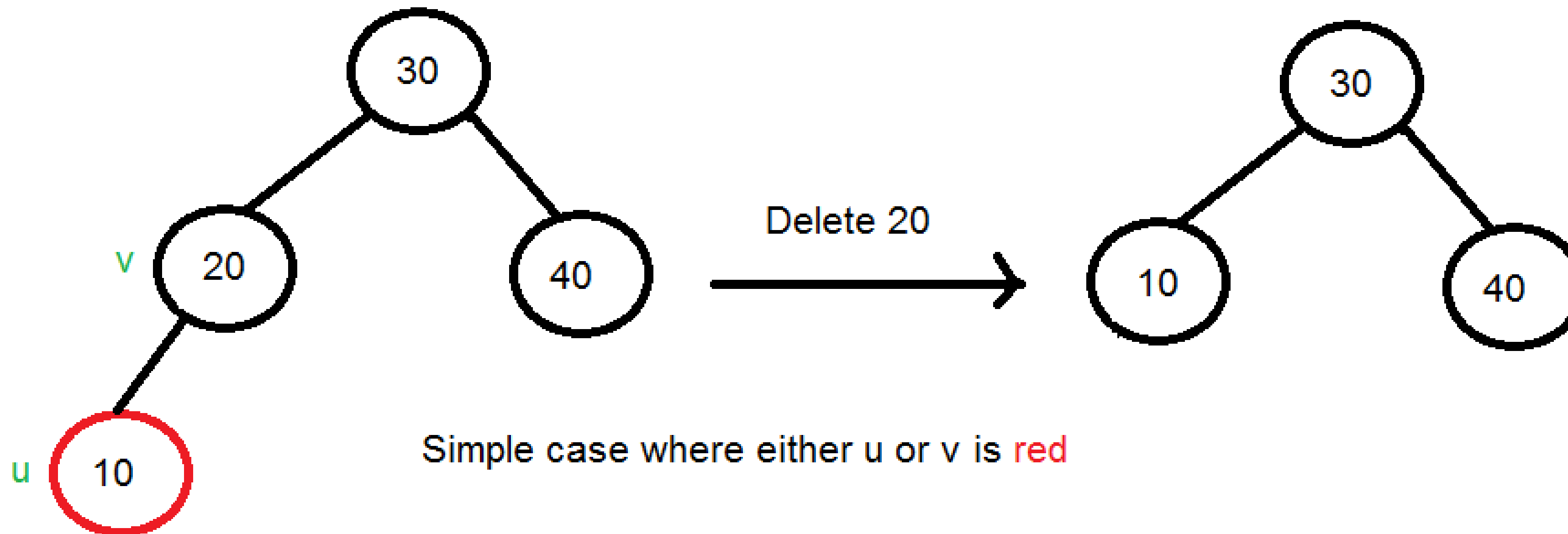
Deletion in RB tree

- **(3.2b):** If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.
- **(3.2c):** If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b).
- This case can be divided in two subcases.
    i. Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.
    ii. Right Case (s is 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 reduces by 1).

Deletion in RB tree

- **Case 1:** Perform standard BST delete. When we perform standard delete operation in BST, we always end up deleting a node which is an either leaf or has only one child
- **Case 2: If either u (10) or v (20) is red,** we mark the replaced child as black (No change in black height).
  - Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



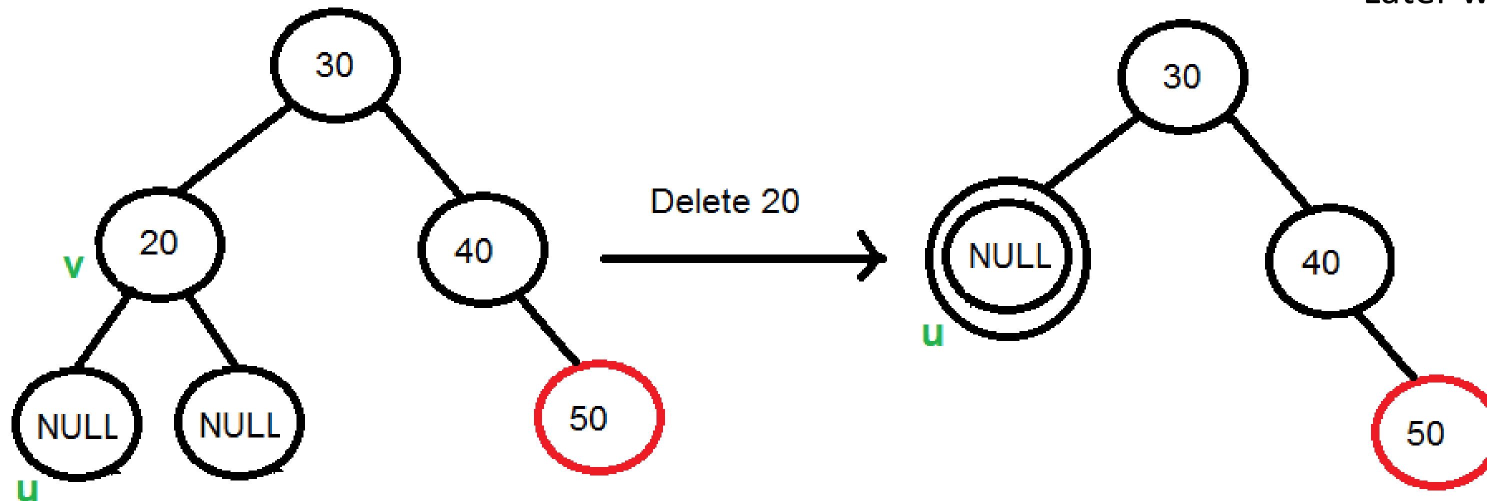
Source

Deletion in RB tree

- **Case 3: If Both u and v are Black**.
- **(3.1)**

1. Color **u** as double black.
2. Now our task reduces to convert this double black to single black.
3. Note that If v is leaf, then u is NULL and color of NULL is considered black.
4. So the deletion of a black leaf also causes a double black.

Later we will see about Null Nodes and their purpose



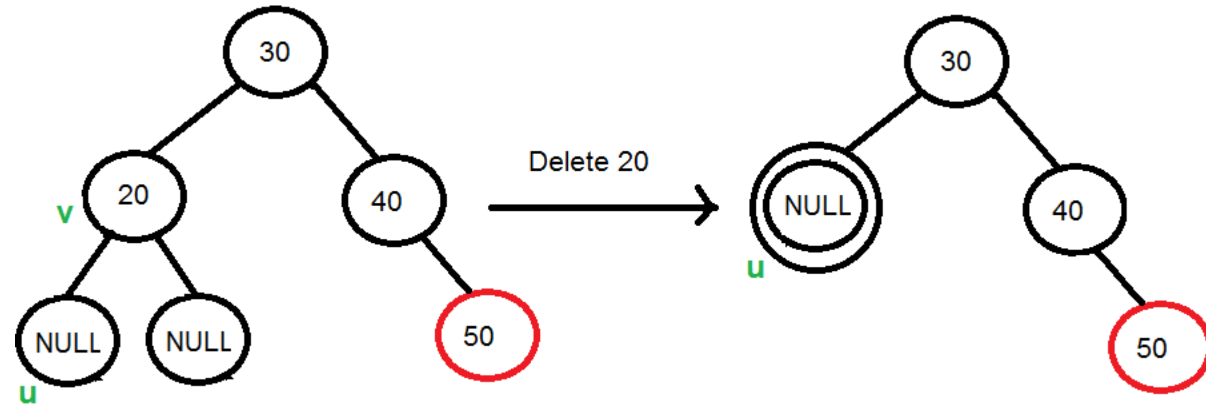
When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

Source

Deletion in RB tree

- **Case 3: If Both u and v are Black**.
- **(3.1)**

1. Color **u** as double black.
2. Now our task reduces to convert this double black to single black.
3. Note that If v is leaf, then u is NULL and color of NULL is considered black.
4. So the deletion of a black leaf also causes a double black.



When 20 is deleted, it is replaced by a NULL, so the NULL becomes double black.
Note that deletion is not done yet, this double black must become single black

(3.2) Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.
(a): If sibling s is black and at least one of sibling's children is red, perform rotation(s).
Let the red child of s be **r**.
This case can be divided in four subcases depending upon positions of s and r.

Source

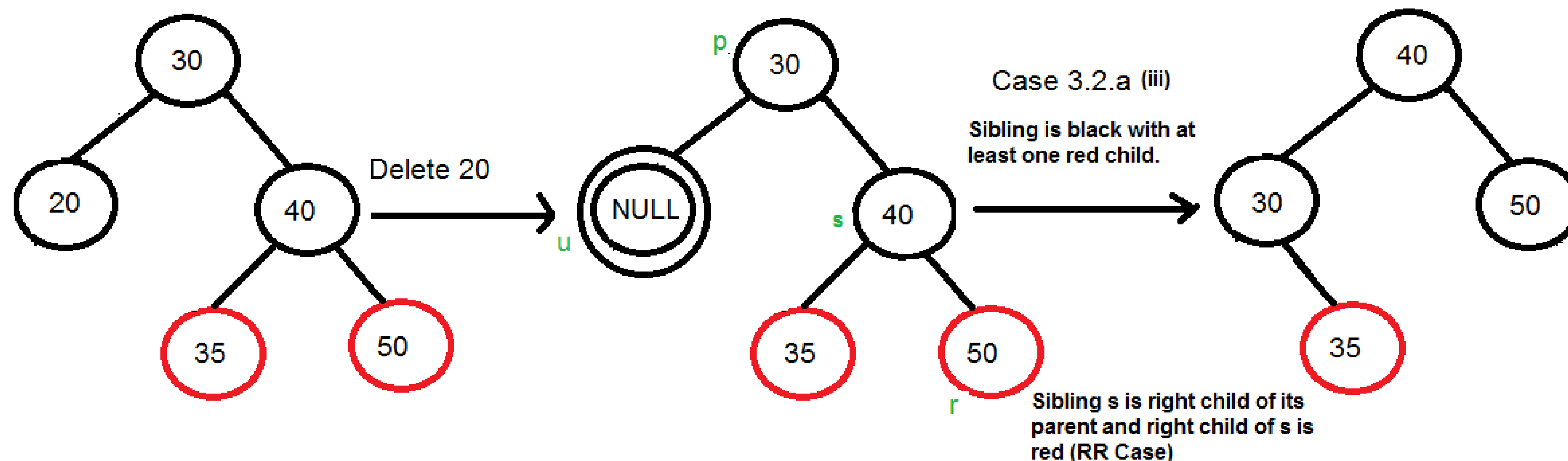
Deletion in RB tree

Do following while the current node u is double black, and it is not the root. Let sibling of node be **s**.
(3.2a): If sibling s is black and at least one of sibling's children is red, perform rotation(s). Let the red child of s be **r**. This case can be divided in four subcases depending upon positions of s and r.

(i) Left Left Case (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

(ii) Left Right Case (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

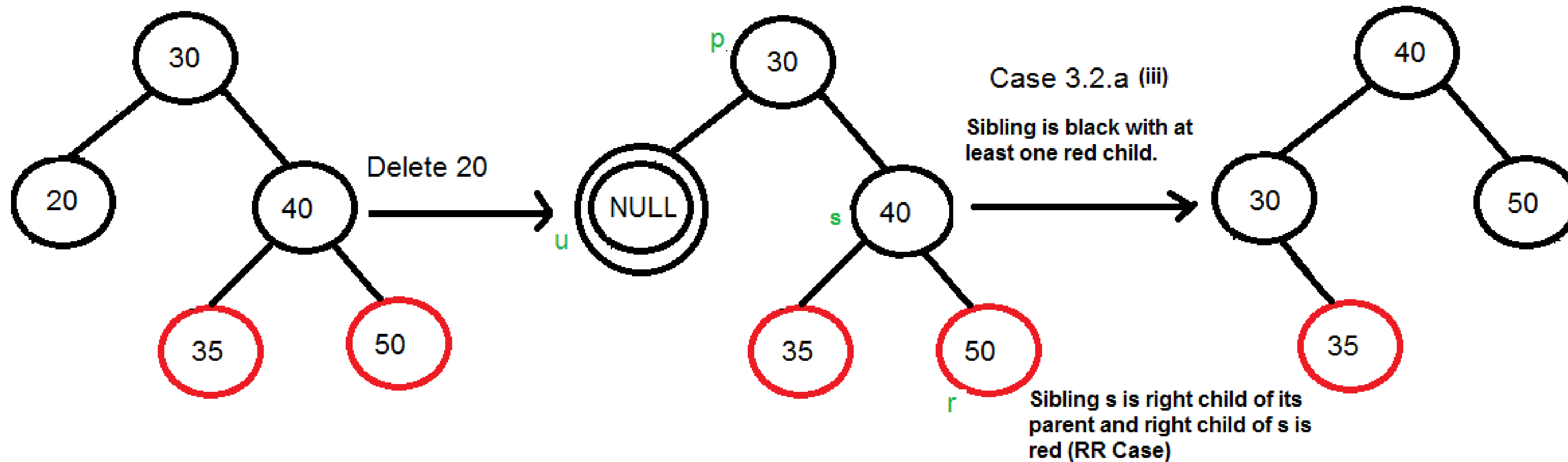
(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



Source

Deletion in RB tree

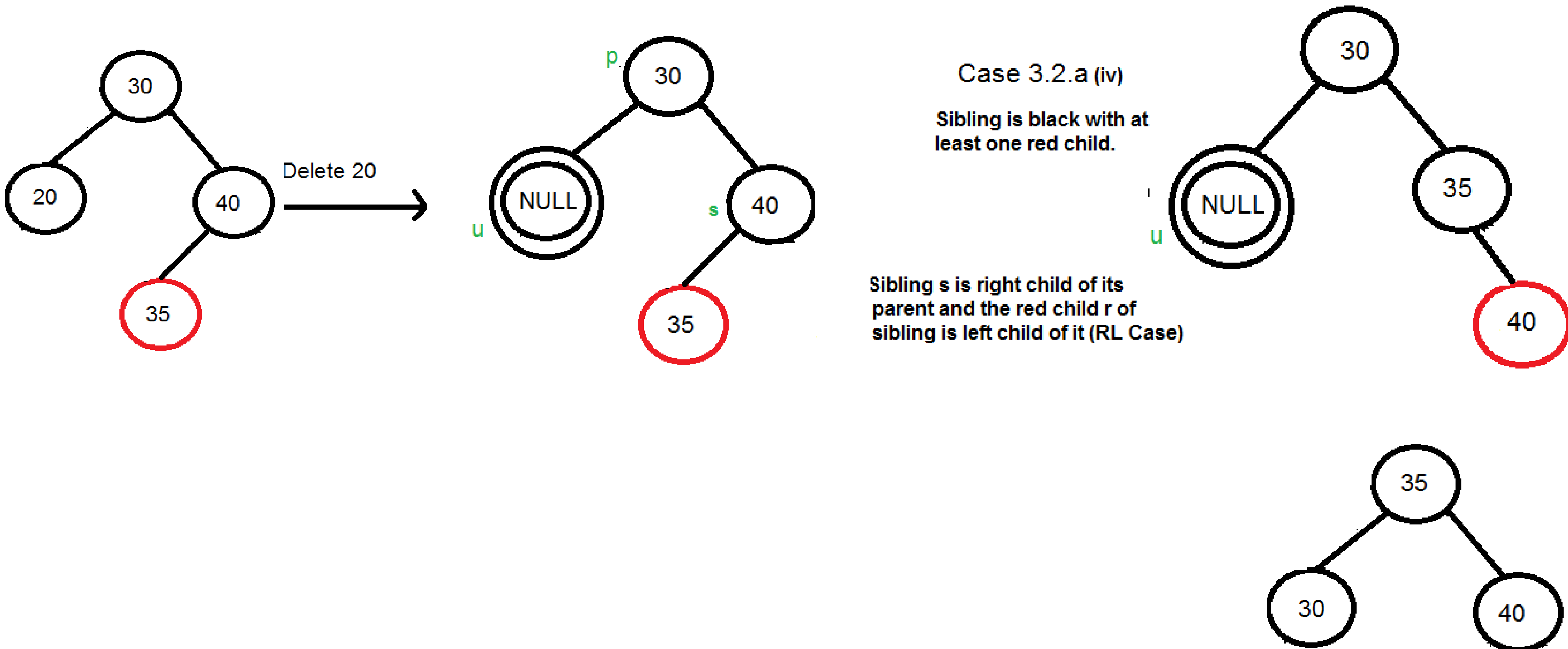
(iii) Right Right Case (s is right child of its parent and r is right child of s or both children of s are red)



Source

Deletion in RB tree

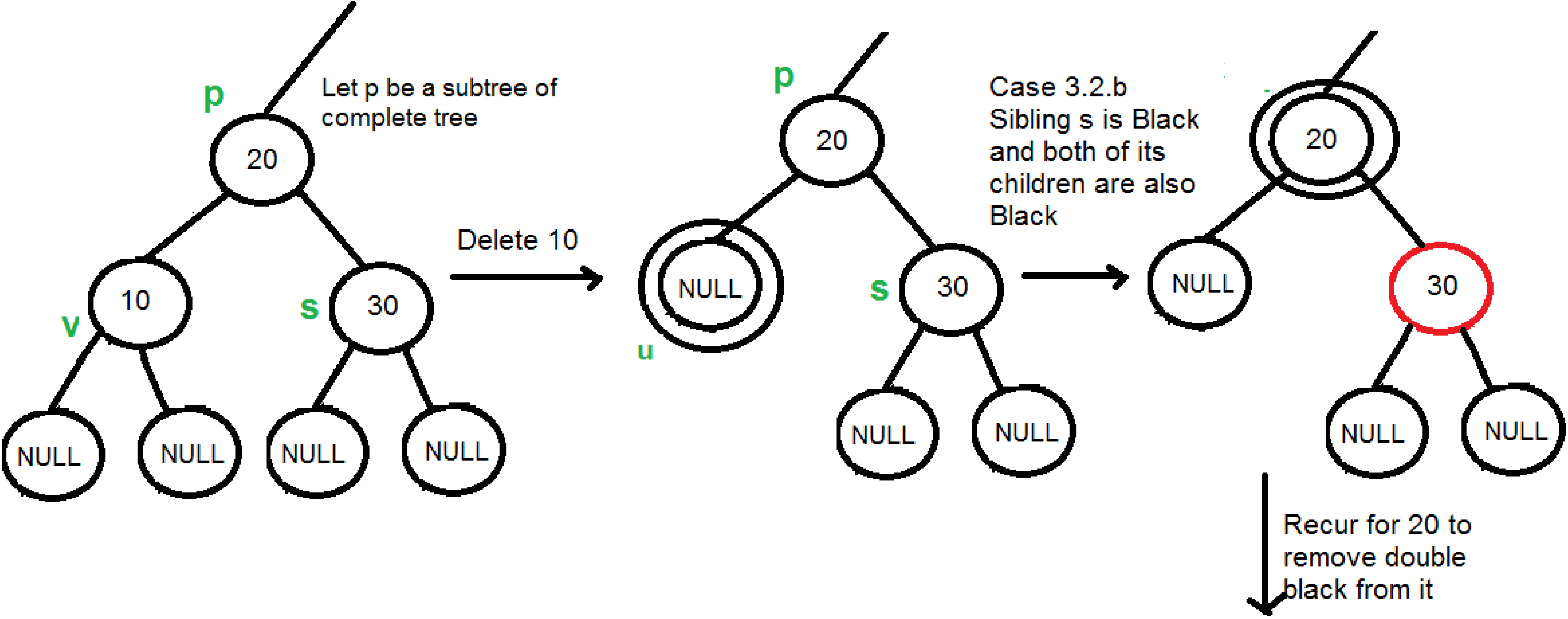
(iv) Right Left Case (s is right child of its parent and r is left child of s)



Source

Deletion in RB tree

(3.2b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

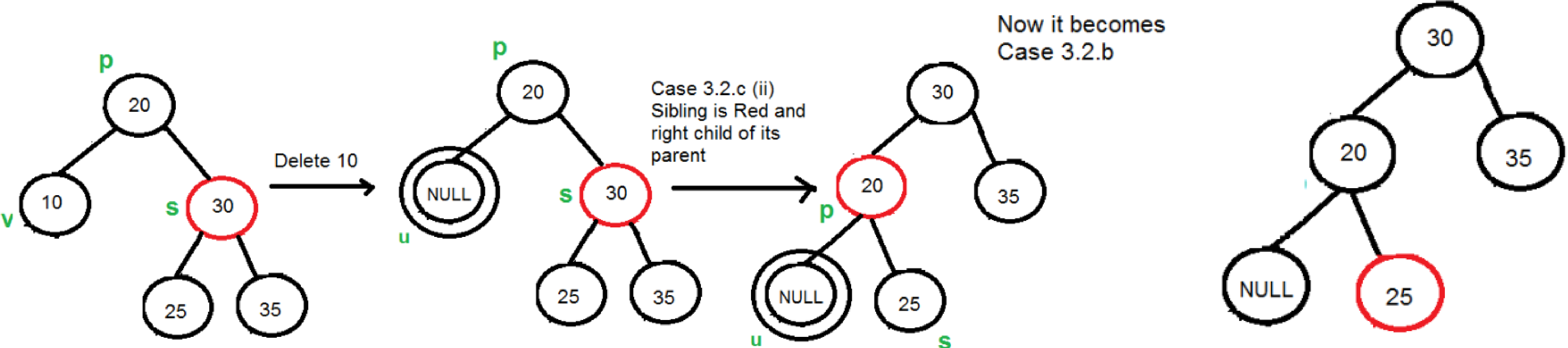


Deletion in RB tree

- In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)
- **(3.2c):** If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b).
- This case can be divided in two subcases.

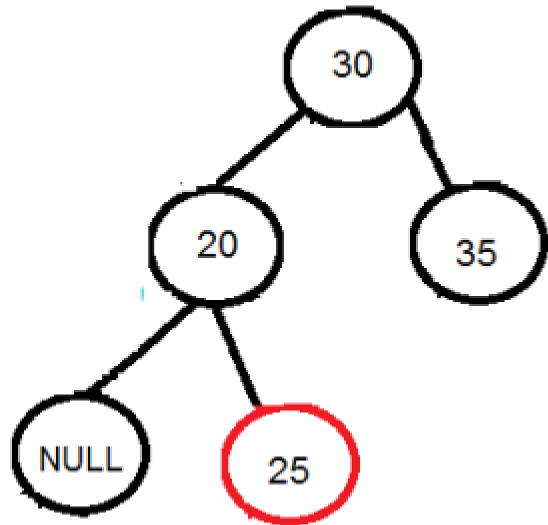
(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

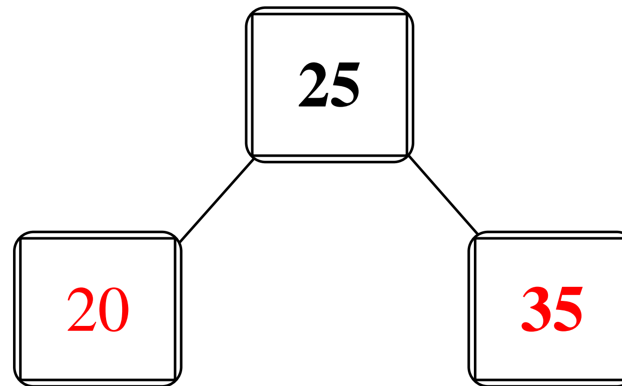
(ii) Right Case (s is right child of its parent). We left rotate the parent p.



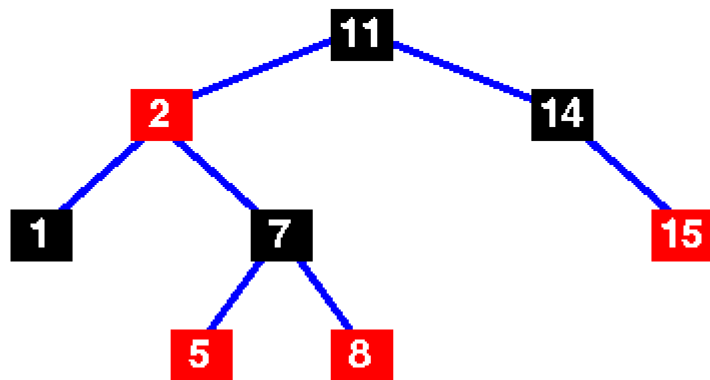
Deletion in RB tree

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

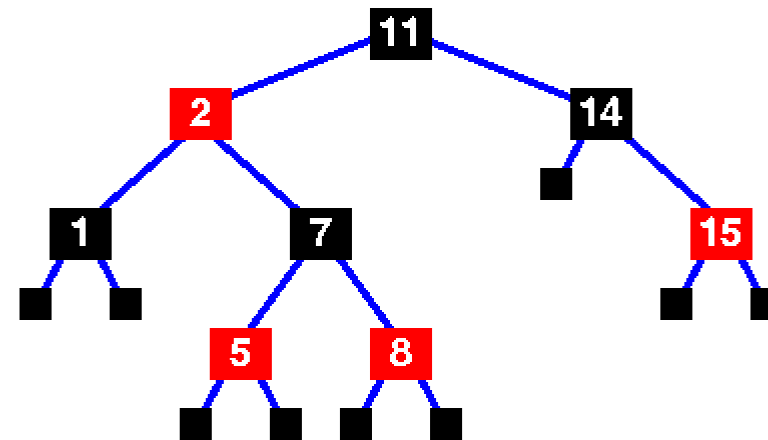




Deletion in RB tree



A basic red-black tree



- Basic red-black tree with the sentinel nodes added.
- Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node.

Searching RB tree

Search Key = 11
Works as BST