14

Bonus Content: Balanced Trees

We saw in *Chapter 6*, *Trees*, that if nodes are inserted into a binary search tree in sequential order, it becomes slow and behaves more or less like a list; that is, each node has exactly one child node. The time complexity of searching, insertion, and deletion is, on average, $O(\log_2 n)$, and in the worst case, it is `O(n)` in a binary search tree. If we want to perform any operation (searching, insertion, deletion) in a binary search tree, it depends upon the height of the tree, where the height of the tree (h) is $O(\log_2 n)$. We would want to have the binary search tree balanced in order to have a time complexity of $O(\log_2 n)$ for the average and worst-case scenarios. Therefore, to improve the performance of the tree's data structure, we generally reduce the height of the tree as much as possible to balance the tree by filling up each row instead. This process is called **balancing the tree**. A self-balancing binary search tree that automatically keeps its height at a minimum typically has a time complexity of $\log_2 n$ and also ensures that the operation in the tree maintains a worst-case time complexity of $O(\log_2 n)$.

There are different types of self-balancing trees, such as red-black trees, AA trees, and scapegoat trees. They balance the tree during each operation that modifies the tree, such as an insertion or deletion operation. There are also external algorithms that balance a tree. The benefits of these are that we do not need to balance the tree on every single operation and can leave balancing so that we do not need it.

In this chapter, we will cover the following topics:

- Introduction to balanced search trees
- Self-balancing binary search trees, such as the AVL tree
- Red-black trees
- B-trees

Let us discuss few of the self-balancing trees first, starting with AVL trees.

AVL trees

The AVL tree is named after its inventors: Adelson, Velski, and Landis. An AVL tree is a binary search tree that has a balanced height in which the difference between the height of the left subtree and right subtree of every node in the tree is not more than `1`. This difference is called a balanced factor. The balancing factor for any node (*X*) can be computed using the following formula:

Balancing Factor (X) = Height(left(X)) - Height(Right(X))

Here, *Height(left(X))* means the height of the left subtree with respect to node *X* and *Height(right(X))* is the height of the right subtree in respect to node *X*. An example of a balanced AVL tree is shown in *Figure 14.1*, in which each node has a balancing factor in the range of `-1` to `1`.

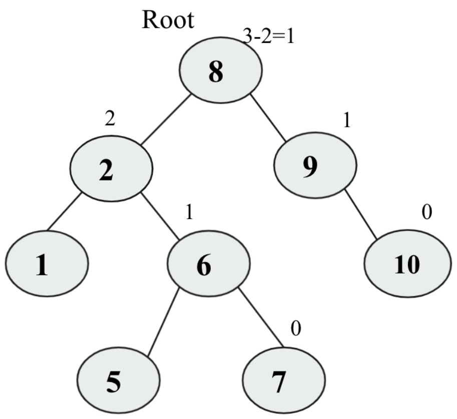

Figure 14.1: An example of a balanced AVL tree

The height of the AVL tree is balanced, and the tree is not skewed, since the time taken in different operations has a worst-case scenario of `O(n)` if the binary search tree is skewed.

In the next section, we discuss different operations in an AVL tree.

Operations in an AVL tree

All the operations in an AVL tree, such as traversing, searching, insertion, and deletion operations are very similar to a binary search tree, since an AVL tree is a binary search tree. It is noteworthy that searching and traversing operations do not violate the condition of the AVL tree because, in such operations, we do not change the structure of the tree, so these operations are exactly the same as those of a binary search tree.

However, in the case of insertion and deletion operations, we need to check if the balancing factor of all nodes satisfies the conditions of an AVL tree.

The insertion and deletion operations in an AVL tree are performed as follows:

1. We insert or delete any data item, as it is what we do in a binary search tree
2. After the insertion or deletion operation, we check the balancing factor of all the nodes from the insertion/deletion point to the root of the tree
3. If the balancing factor of all the nodes is in the range of `-1` and `1`, then we use other operations if required
4. If the balancing factor of any node violates the condition, then we use any of the above rotations to make the tree balanced

Let's consider the insertion operation first.

Inserting nodes

After the insertion of a node in an AVL tree, if the tree becomes unbalanced, it will only impact the nodes that are in the path from the newly added node to the root node, as only subtrees of these nodes will be updated. So, it is important to note that when any insertion operation is performed in an AVL tree, the balancing factor of the nodes, which is within the path from the inserted node to the root node, will be changed. So, in order to keep the AVL property intact, such as the balancing factor of each node not being more than `1`, we have to check all the nodes from the insertion point to the root node. Once we find a node violating this condition, we use a rotation to balance that node. After fixing the balancing property for this node, there is no need to further check any other node from that node to the root node, as the issue will automatically be fixed.

An AVL tree performs rotations after every operation, such as insertion or deletion, to ensure that the tree remains balanced and that every node adheres to the AVL tree's properties. Whenever any node (*X*) violates the balancing condition of an AVL tree, there will be one of the following four cases:

1. An insertion in the left subtree of the left child node of node `X`
2. An insertion in the right subtree of the right child node of node `X`
3. An insertion in the right subtree of the left child node of node `X`
4. An insertion in the left subtree of the right child node of node `X`

Left, left variant

If a node violates the AVL property, as mentioned in the first case, in which a new element is added to the left subtree of the left child of a node (say X), then that node X becomes unbalanced. For example, after adding node 4 in the AVL tree on the left subtree of left child 5, node 6 becomes unbalanced (since the height difference for node 6 becomes two). This is shown in *Figure 14.2*:

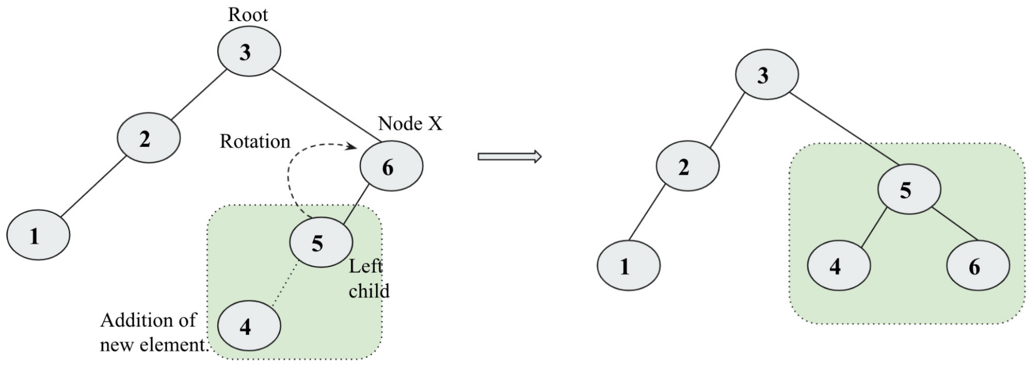

Figure 14.2: Demonstration of the insertion operation for case 1 of the left, left variant

A single right rotation at node X will balance the tree, as shown in the above figure.

Right, right variant

If a node violates the AVL tree's property in which a new element is added to the right subtree of the right child of a node (say X), then node X becomes unbalanced. For example, after adding node 9 into the AVL tree in the right subtree of right child 8 of node X, the tree becomes unbalanced (since the height difference for node 6 becomes two). This is shown in *Figure 14.3*:

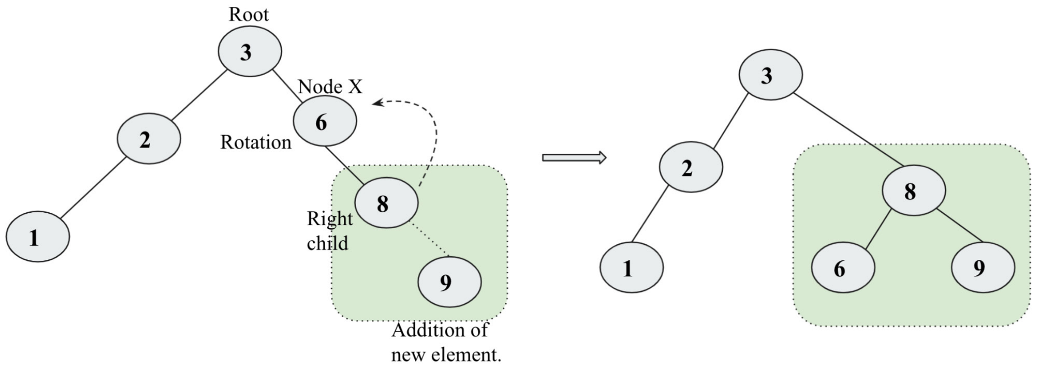

Figure 14.3: Demonstration of the insertion operation for case 2 of the right, right variant

A single left rotation at node X will make the tree balanced as shown in the above figure.

Left, right variant

If a node violates the AVL property in which a new element is added to the right subtree of the left child node, then that node becomes unbalanced. For example, after adding node 4 in the right subtree of left child node 2, which is the left child of node 9, the balancing factor of node X (node 9 in this example) violates the property of the AVL tree. This is shown in *Figure 14.4*:

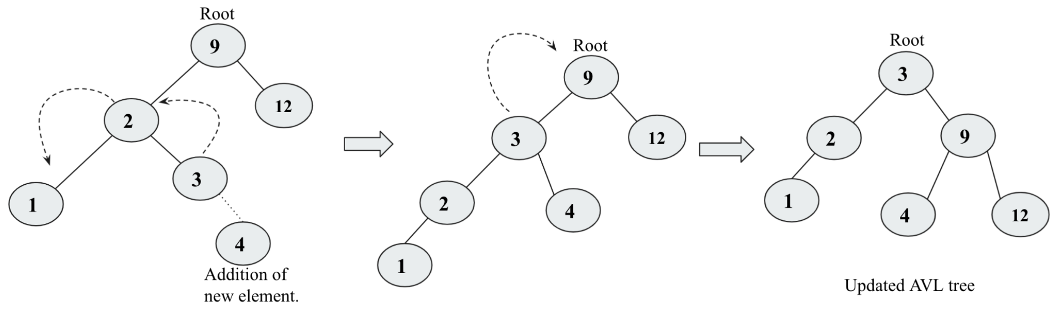

Figure 14.4: Demonstration of the insertion operation for case 3 of the left, right variant

One left rotation and one right rotation (double rotation) will balance the tree, as shown in the figure.

Right, left variant

If a node violates the AVL tree's property in which a new element is added to the left subtree of the right child node, then that node becomes unbalanced. For example, after adding node 6 into the left subtree of node 8, which is the right child of node 5, the balancing factor of node 5 becomes two, which violates the property of the AVL tree. This is shown in *Figure 14.5*:

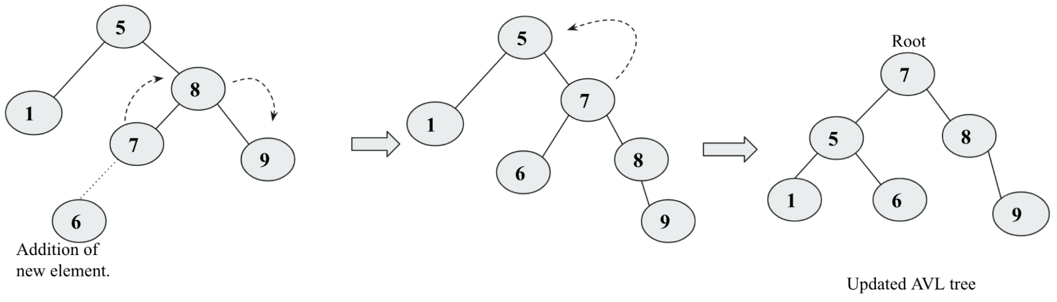

Figure 14.5: Demonstration of the insertion operation for case 4 of the right, left variant

One right rotation and one left rotation (double rotation) will balance the tree, as shown in *Figure 14.5*. Here, in the first right rotation, node `7` becomes the parent of node `8`. Furthermore, in the second rotation, which is the left rotation, node `7` becomes the parent node of node `5`, and then node `6` becomes the right child of node `5` since node `7` already has two children, nodes `5` and `8`.

Next, let us discuss the process of deleting a node in an AVL tree.

Deleting nodes

The deletion operation in an AVL tree is as follows:

1. Firstly, we search for the node that is to be deleted in the tree. Let's say that node is `X`.
2. We delete the contents of the node as described in *Deleting nodes* of *Chapter 6*, *Trees*.
3. After deleting a node in the AVL tree, we check the balancing factor of the nodes of the tree. If the balancing factor remains in the range of `–1` to `1`, it means the deletion of the node is as per the AVL tree properties. If the balancing factor becomes `+2` or `–2`, it means the tree becomes unbalanced, and there is a need to balance the tree using rotation methods, as discussed in the previous subsection.

The worst-case time complexity of the insert, search, and delete operations in an AVL tree is `O(log n)`. Therefore, AVL trees perform better than binary search trees, which have a worst-case time complexity of `O(n)`. The binary search tree works worst when the tree becomes skewed and, in such cases, the AVL tree can be used.

Next, we will discuss another one of the popular balanced binary search tree algorithms.

Red-black trees

A red-black tree is one of the most popular data structures to be used for the Linux kernel scheduler. A red-black tree is a kind of self-balancing binary search tree in which each node has extra information to store, which is the color of the node as red or black. This extra color information is used to ensure that the tree remains balanced after performing any operation, such as insertion or deletion.

Red-black trees ensure that no simple path from the root to a leaf node is more than twice as long as any other path by restricting the node colors, ensuring that the tree is roughly balanced. When the tree is updated, the new tree is rearranged and "recolored" in order to restore the coloring properties that limit how unbalanced the tree can become in the worst-case scenario.

The following attributes are the properties of a red-black tree:

- Each node should be either red or black
- A root node should always be black
- Every leaf node, which is an empty node (`nil`), should be black
- Children of a red node should be black
- If a node is red, then its parent should be black
- For each node, all pathways from the node to any of its descendent leaves (`nil` nodes) should contain the same number of black nodes

To demonstrate the above characteristics of a red-black tree, we can refer to the example shown in *Figure 14.6*:

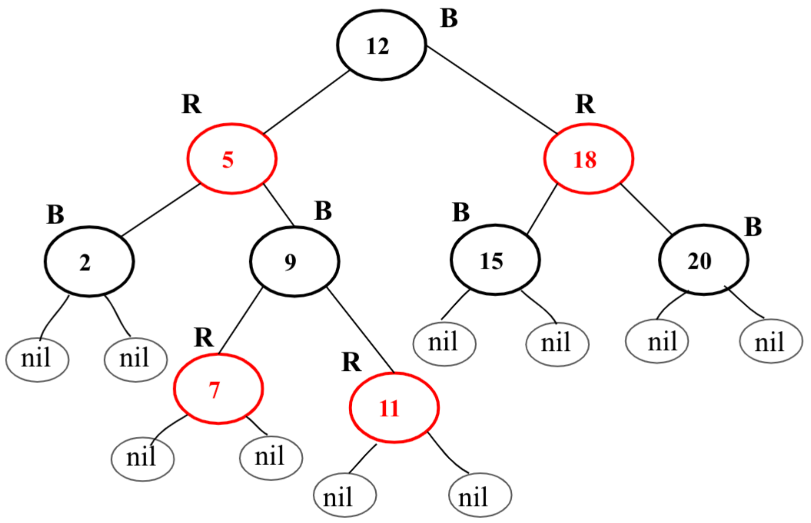

Figure 14.6: An example of a red-black tree

Insertion in a red-black tree

In order to insert any element in a red-black tree, we should first ensure that the tree follows the property of the binary search tree; for example, every node in the tree should have a higher value as compared to the left subtree, and a lower value as compared to the right subtree. Furthermore, whenever we insert any node into the tree, we ensure that all the nodes in the tree follow the properties of the red-black tree, and if the tree violates the attributes of the red-black tree after the insertion of a new node, we perform recoloring or rotation.

Here, recoloring means we may need to change the color of nodes to keep the properties of the red-black tree according to the rules of inserting a new node in the tree. How to perform recoloring and rotation is discussed in the following rules, which depend on how we insert a new element in the tree.

The following rules ensure that the tree is following the red-black tree properties after inserting a new element into a red-black tree:

1. Firstly, check if the tree is empty. If it is empty, then we create a new node as the root node of the tree, and we color it black.
2. If the tree is not empty, then we create a new node as a leaf node with the color red.
3. We traverse the red-black tree to reach the appropriate node according to the binary search tree property when inserting a new node.
4. If the parent of the new node is black, then we exit.
5. If the parent of the new node is red, then we check the color of the parent's sibling of the new node:

    a. If the color of the parent's sibling of the new node is black, then we perform suitable rotation and recolor.
    b. If the color of the parent's sibling of the new node is red, then we recolor that node and also check if the parent's parent of the new node is not a root node. If not, then we recolor it and recheck the property of the red-black tree.

Let us understand the insertion operation in the red-black tree by creating and inserting the following elements {12, 18, 6, 14, 16, 32}.

At the start, the tree is empty so we add a node with the value 12 and color it black. Next, we add another node, 18. Since it is greater than 12, we make it a right child and color it red, since a new node will always be red. Next, as per the properties of the red-black tree, we check if the parent of the new node is black, and then we exit. This is a red-black tree since it is has all the properties of a red-black tree. Next, we add a new node with the value 6. Since it is less than the root value, we go to the left subtree, and then we add the new node in the color red. We again check the parent of the new node, and since it is a black node, we exit. The process of adding these three nodes is shown in *Figure 14.7*:

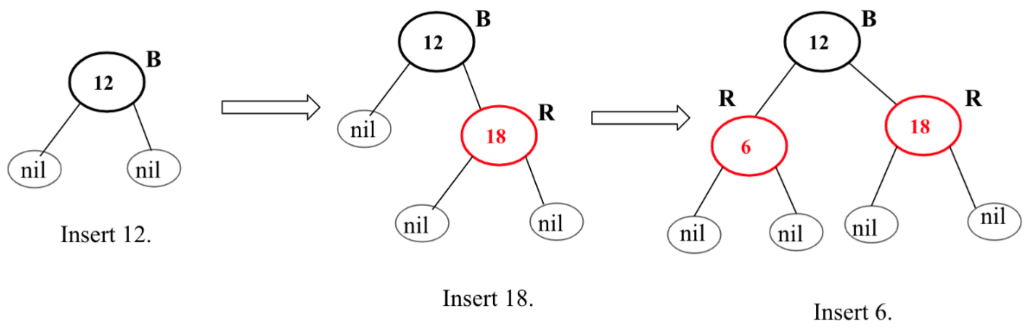

Figure 14.7: Insertion of elements 12, 18, and 6 in the red-black tree

Next, we add a new element, 15. To do this, we must first reach an appropriate node where we can add it, according to the binary search tree property. So, a new node is created as the left child of node 18, which is colored red. Next, we check the parent of the new node, and since it is red (node 18 in this example), it violates the condition of the red-black tree that the child of a red node should be black. In this case, we have to recolor or rotate the tree to ensure that the tree is following the properties of the red-black tree.

So, as per rule five, if the parent of the new node is red, then we check the color of the parent's sibling (which is red, as it is node 6 in this example). If the color of the parent's sibling is red, then we recolor it as suitable, and we check that the parent's parent of the new node is not a root node. Then we recolor it and recheck the properties of the red-black tree as per rule five (*B*) again. So, for this example, since the color of the parent's sibling is red, and the parent's parent is a root node, we just recolor the parent's sibling nodes of the new node. Now, we can check that all the properties of the red-black tree are being fulfilled. This process is shown in *Figure 14.8*:

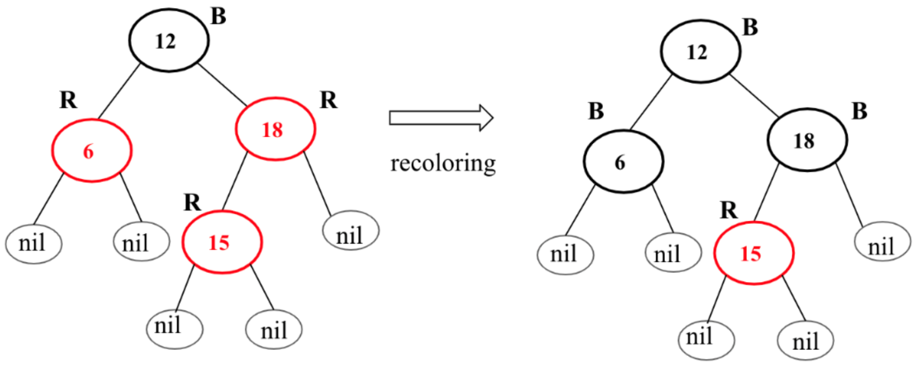

Figure 14.8: Insertion of element 15 in the red-black tree

Next, we insert a new element, 16, into the red-black tree. Firstly, we traverse the tree and add the new node, colored red, at the appropriate position, which is as the right child of node 15. Here, we can see that there is a violation of the rule that the child of a red node cannot be red. So, we have to rearrange or recolor the tree.

We check as per rule number five that if the parent of the new node is red, then we check the color of the parent's sibling of the new node, and if it is black or `null`, then we perform the suitable rotation and recolor. In this example case, we can see that the parent's sibling is `null`, so we will have to perform some rotations. We follow the same rules for rotations as we have already discussed in *Inserting nodes*, so for this situation, in order to make the tree balanced, we need to make two rotations, a left rotation and then a right rotation, which is shown in *Figure 14.4*. After the rotations, we have to recolor nodes 16 and 18, so node 16 becomes black and node 18 becomes red. Furthermore, we check all the properties of the red-black tree, since in this example, it is following all the rules so there is no need to recolor any nodes. The process of adding element 15 is shown in *Figure 14.9*:

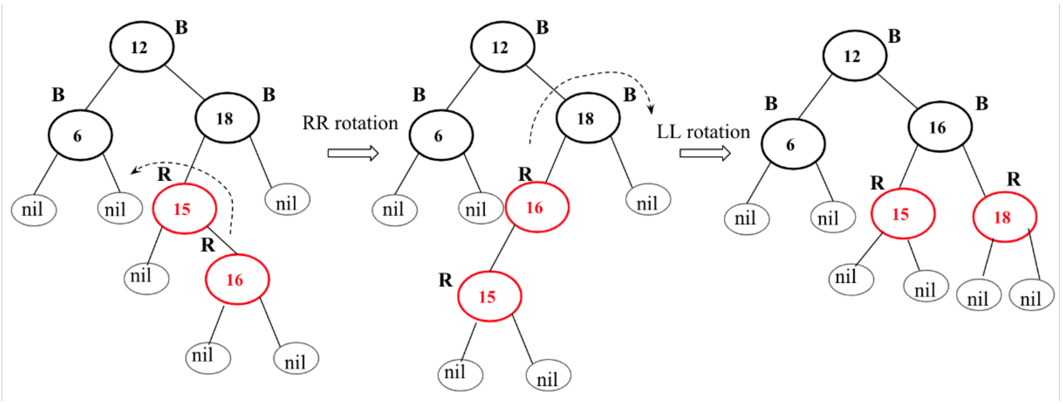

Figure 14.9: Insertion of elements 15 in the red-black tree

Next, we insert a node with the value 32 in the red-black tree. We traverse the tree to add the new node at the appropriate position according to the binary search tree property. For the given tree, we add the new node with a value of 32 in red, and as the right child of node 18. Next, we check the parent of the new node, which is red in this case, so there is a violation of the rule in which children of a red node should not be red. Next, we check the parent of the new node's sibling, which is red, so as per rule five (*B*), we recolor the parent and parent's sibling, which are nodes 15 and 18 in this example.

Next, we also check whether the parent's parent is a root node or not, and if it is not a root node, we also recolor that node (node 16 in this example). Finally, we check if the tree is following the red-black tree properties or not. The complete process of this is shown in *Figure 14.10*:

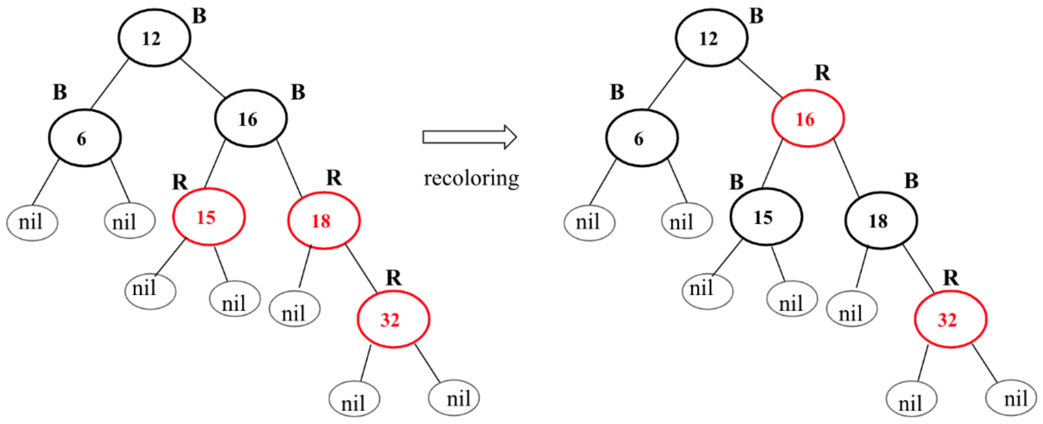

Figure 14.10: Insertion of element 32 in the red-black tree

In a nutshell, whenever we try to insert a new node that is colored red into the red-black tree, the property that may be violated is that the children of the red node should not also be red. We can solve this conflict by recoloring and rotating the tree. The time complexity to insert a node into a red-black tree is `O(log n)`. Next, let us discuss the deletion operation in red-black trees.

Deletion in the red-black tree

There are two steps for deleting any node from the red-black tree:

1. Firstly, we follow the same process that we do in binary search trees to delete a node, which has already been discussed in *Deleting nodes* in *Chapter 6, Trees*. We search for the element to be deleted in the tree. Then, we have three cases in which to perform the deletion operation:

    a. When the node to be deleted is a leaf node, we simply delete it.

    b. When the node to be deleted has only one child, then we replace it with its child node.

    c. When the node to be deleted has two children, then we replace it with either an inorder predecessor or an inorder successor. It is important to note that while applying the deletion operation in a binary search tree or red-black tree, we do not delete any internal nodes; we always replace the internal node with the value of the leaf node.

2. After identifying the node to be deleted, we have rules to ensure that all the properties of the red-black tree are intact. In general, whenever we delete any node from the tree that is a red node, no rules of the red-black tree will be violated. When we delete a black node from the tree, then there is a possibility that the rule that we should have equal numbers of black nodes within all the paths from the node to any of the descendent `nil` nodes will be violated. To resolve this, we need to follow some specific rules of the deletion operation from the red-black tree, which are discussed below with examples.

Case: If the node to be deleted is red, we can simply delete that node, similar to a binary search tree, and exit

To understand this, let us take an example of deleting node 18 from the red-black tree shown in Figure *14.11*. Firstly, we search for node 18 in the tree, since 18 is greater than the root node 12, so we search in the right subtree. Again, 18 is greater than node 16, so we search in the right subtree.

Once we find the desired node, in this example, it is the leaf node that is colored red, so we can simply delete it from the tree or replace its value with a `nil` value. We can check that all the properties of the red-black tree are fulfilled. We can directly delete the red node from the red-black tree because it will not violate any properties of the red-black tree. This process is shown in *Figure 14.11*:

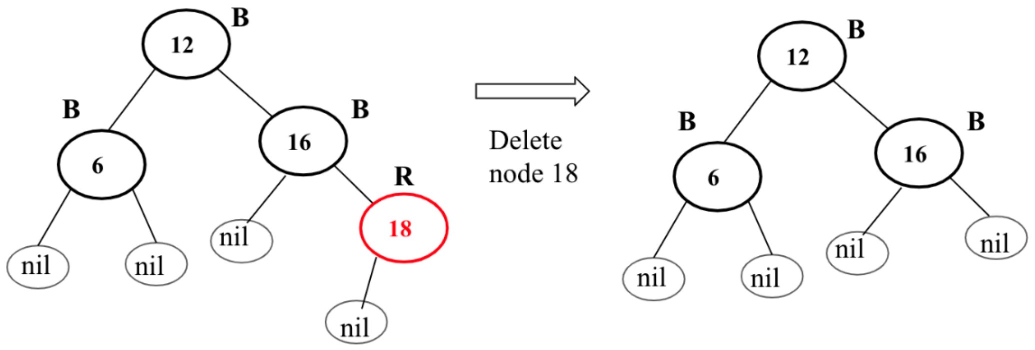

Figure 14.11: Deletion of node 18 from the red-black tree

Let us take another example in which we have two children from the same parent node, node 16, that we want to delete. We first search the node, then check its color since it is a red-colored node. The node is an internal node with two children.

To delete such a node, we can replace it with either an inorder successor or an inorder predecessor. The inorder successor is the smallest element in the right subtree, and the inorder predecessor is the largest element in the left subtree.

In this example, we use an inorder successor, and the inorder successor of the node `16` is node `32`. So, we replace the value of the node to be deleted with the value of the inorder successor, `32`. We can then delete leaf node `32` since it will not violate any of the properties of the red-black tree. This process is shown in *Figure 14.12*:

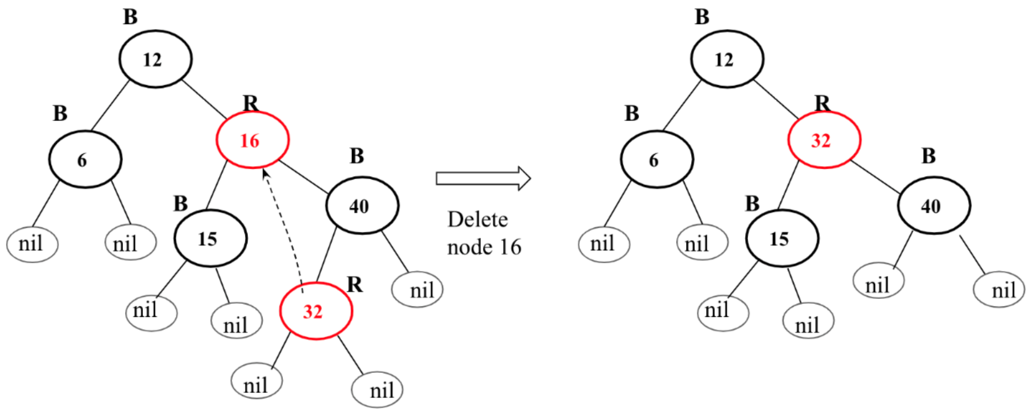

Figure 14.12: Deletion of node 16 from the red-black tree

It is important to note that when we replace the value of the node to be deleted with one of its children's values, we do not change the color of the node. For example, the color of node `16` will be red when we replace its value with its appropriate child.

Let us take another example, deleting node `20` from the red-black tree shown in *Figure 14.13*. Firstly, we search for the element to be deleted, since it is an internal node with two children (node `15` and node `40`), so we replace its value with an inorder successor, which is `32` (the smallest element in the right subtree of node `20`). We replace the value `20` with the value `32`. We recursively find the inorder successor until we reach the leaf node. In order to remove node `32`, we again search for the inorder successor of node `32`, which is node `36`.

We replace the value of node 32 with 36. It is noteworthy that we do not change the color of node 32 at the time of the replacement. Again, we recursively search for the next inorder successor, which is 36 in this example. Since it is a leaf node and colored red, we can simply delete it. This process is shown in *Figure 14.13*:

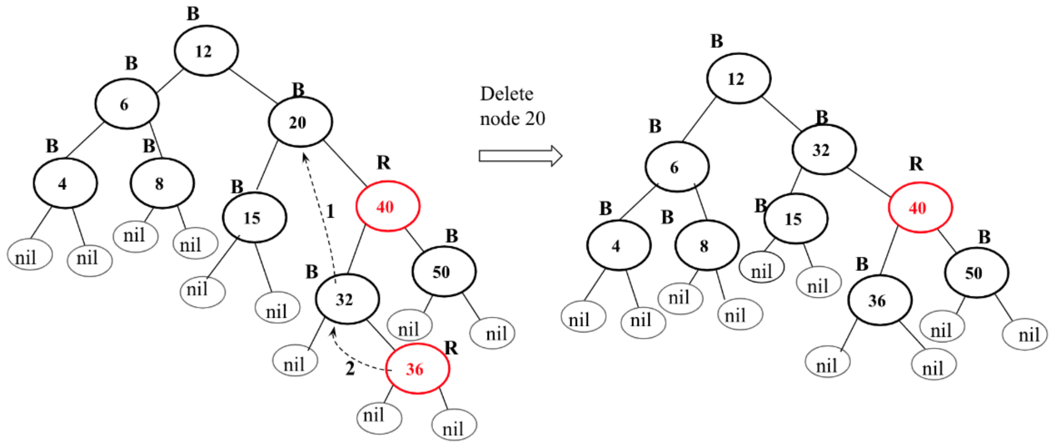

Figure 14.13: Deletion of node 20 from the red-black tree

Case: If the node to be deleted is black

When deleting a black node in a red-black tree, the main property that is violated is the change in the number of black nodes in the path that we delete the black node from, as one black node will be removed from that path. To understand the deletion of the black node, we use the concept of **double black**, which is important to understand in the deletion process. This is when we delete a black node, and that node will have black children. This is a double black situation, and that node is marked (or treated as) as a double black node. Now the main task is how we can convert this double black node into a single black node. There can be several different possible cases in which we want to delete a black node from a red-black tree, as shown in *Table 14.1*. We will discuss them with examples.

Case	Check condition	Action
1	If the node to be deleted is a leaf node in the color red.	We can simply delete it.
2	If the double black node is a root node.	We remove the double black from the root and make it into a single black node.

3	If the color of the sibling of a double black node is black, and the children of the double black's siblings are also black.	We apply the following three steps one by one: • We convert the double black into a single black node. • If the parent of the double black is also black, make it double black DB, and if the parent of the double black is red, make it black. • We color the sibling of the double black red. If the double black problem still exists, then we reapply the appropriate cases.
4	If the color of the sibling of the double black node is red.	• We swap the color of the parent and the sibling of the double black node. • We rotate the parent node in the direction of the double black node. We recheck and reapply the appropriate case.
5	If the sibling of the double black node is black, the sibling's furthest child is black, and the nearest child is red.	• We swap the color of the sibling and the sibling's child that is nearest to the double black node. • We perform the rotation operation on the sibling node in the opposite direction of the double black node. • Apply case 6.
6	If the sibling of the double black node is black and the sibling's child that is furthest away is red.	• We swap the color of the parent and sibling of the double black node. • We perform the rotation of the parent of the double black node in the direction of the DB node. • We convert the double black node into a single black node. • We change the color of the red sibling of the double black node, which is further from the black.

Table 14.1: Different cases for the deletion of a node from the red-black tree

Consider the red-black tree shown in *Figure 14.14*. Let us delete node 17 from this tree. Firstly, we search for the desired node in the tree, which is a black node, so we cannot directly delete it. Since node 17 is a black node, and its `nil` children are also black, we make it a double black node, and then as per case 3, as defined in *Table 14.1*, we check its sibling and if that is black and both of its children are also black, then perform the following steps to ensure that the properties of the red-black are not violated.

We convert the double black node into a single black node, then we color the parent black (node `20` in this example) since it is already red. Finally, we make the sibling of the double black node red (node `40` in this example). We can see that the final tree is fulfilling all the properties of the red-black tree. The process of deleting node 18 is shown in *Figure 14.14*:

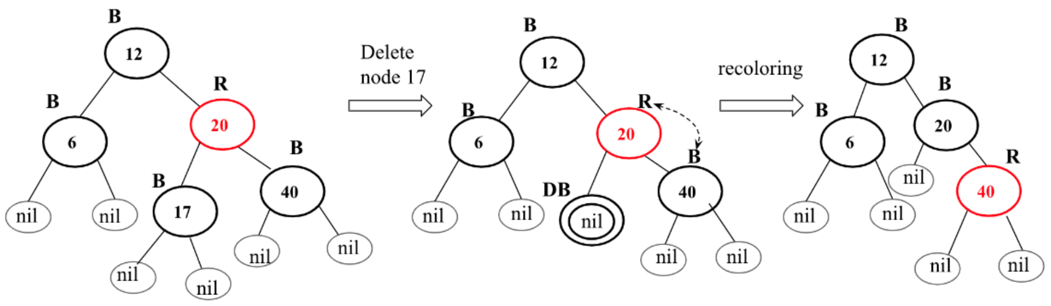

Figure 14.14: Deletion of node 18 from the red-black tree

Next, let us take another example of deleting node 18 from the red-black tree, as shown in *Figure 14.15*. Firstly, we find node 18 in the tree. Node 18 is a black node, so we cannot directly delete it. Since it is a black node, and its `nil` children are also black, we make it a double black node. Furthermore, since its sibling and their children are also black, so as per case 3, we remove the double black from the node and make the parent of it a double black (since it is already black), and make the sibling red.

We again check the tree, and we see whether there is still a double black node. If there is any double black node in the tree, we iteratively process it to convert the double black node into a single black node. We check whether the color of the siblings and their children is black. Here, in this example, they are node 6 and nodes 4 and 8 respectively. So as per case 3, we remove the double black, make the parent into a double black, and then color the sibling red. We again check the tree, and we see that there is still a double black node. As per case 2 from *Table 14.1*, we can simply remove the double black from node 12, since it is a root node.

The complete step-by-step process to delete node 18 from the red-black tree is shown in *Figure 14.15*:

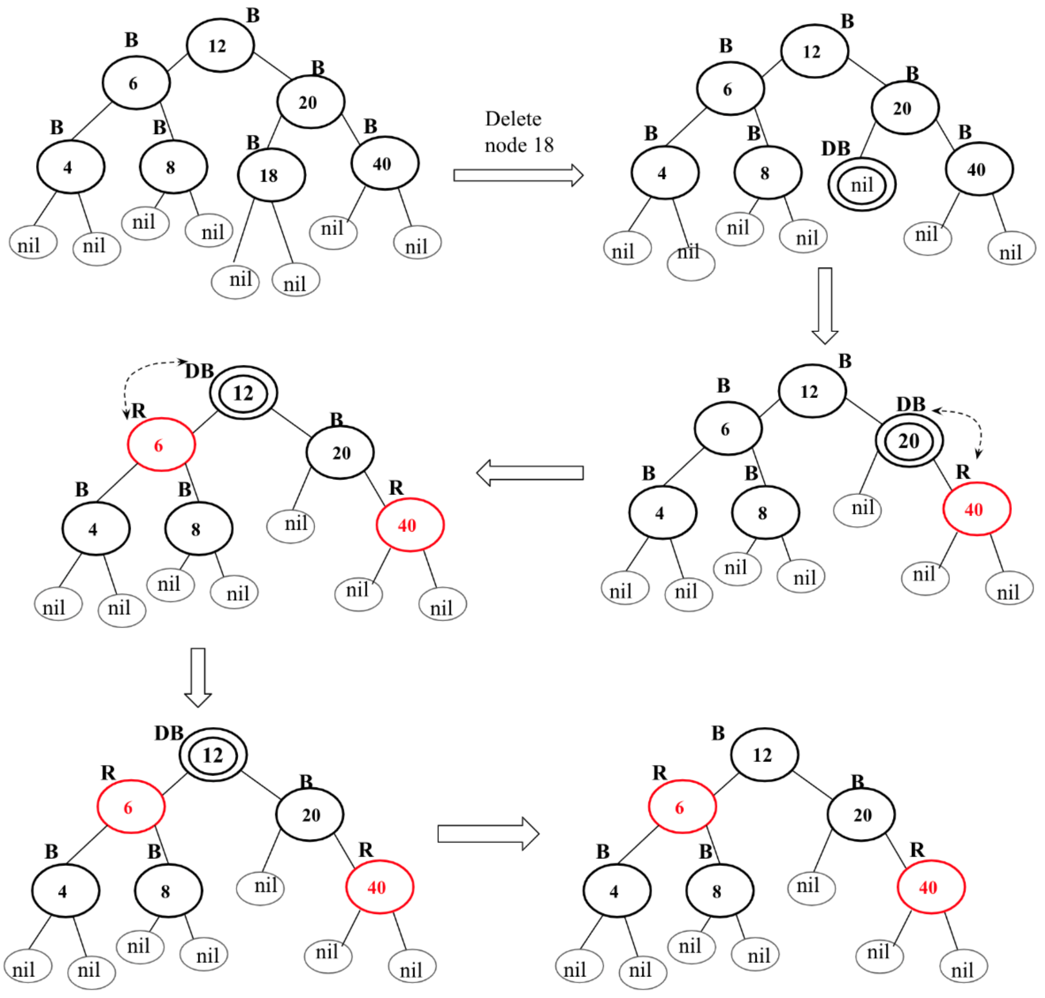

Figure 14.15: Deletion of node 18 from the red-black tree

To understand case 4 from *Table 7.1*, we take an example of deleting node 19 from the red-black tree, as shown in *Figure 14.15*.

First, we search for node 19 in the tree. It is a black node, and its `nil` children are also black, so we delete the element and make it into a double black node. To convert the double black node into a single black node, we check the color of its sibling, which is red (node `40` in this example). As per case 4 of *Table 6.2*, we swap the colors of the parent and the sibling of the double black node (which are nodes `20` and `40`).

Still, there is a double black node in the tree, so the next step is to rotate the parent node in the direction of the double black node. In this direction, it is a left rotation. After the rotation, the tree can be seen in the third diagram of *Figure 14.15*.

We again check if the tree is a red-black tree. However, there is still a double black node. We check the condition, and it is case 3, in which the color of the sibling of the double black node is black, and the children of the double black's siblings are also black. So, we convert the double black node into a single black node, then we color the parent black and the sibling red. Furthermore, we recheck the properties of the tree, and it is fulfilling all properties, so we stop. The step-by-step process to delete node 19 is shown in *Figure 14.16*:

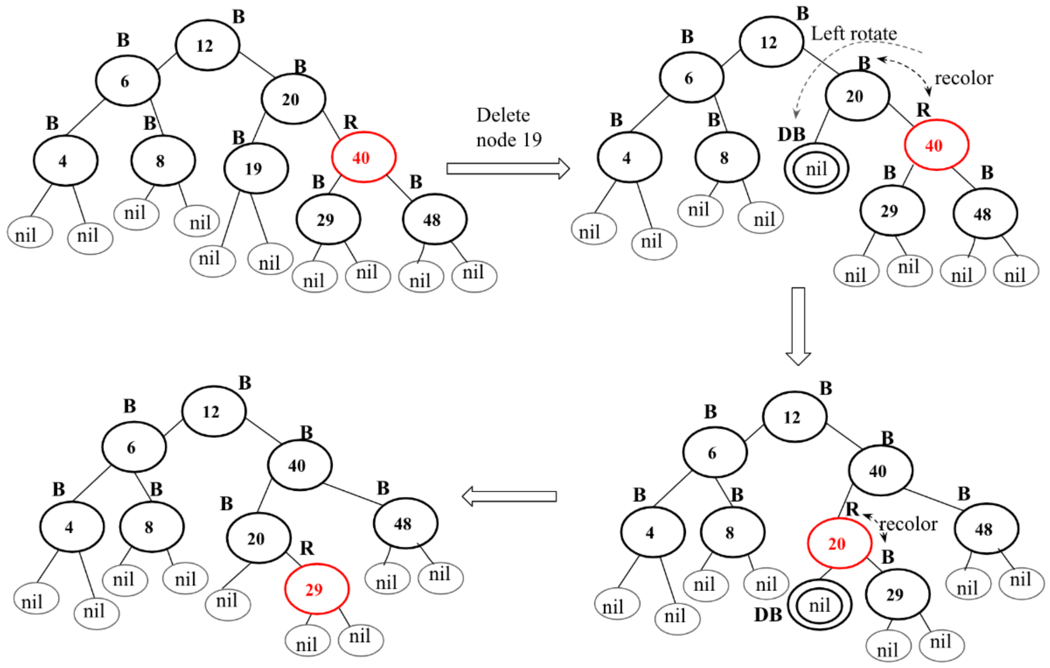

Figure 14.16: Deletion of node 18 from the red-black tree

To understand cases 5 and 6 of *Table 14.1*, we take another example of deleting node 4 from the red-black tree, as shown in *Figure 14.17*.

First, we search for node `4` in the given red-black tree. Since node `4` is a black node, and its `nil` children are also black, we delete the element and make it into a double black node. To convert the double black node into a single black node, we check the color of its sibling, which is black (node `8` in this example). We check the condition, which fits case `3`, in which the color of the sibling of the double black node is black, and the children of the double black's siblings are also black. So, we convert the double black node into a single black node, then we make the parent into double black, and the sibling red.

We check the tree again, since there is still a double black node, so that we know which condition should be applied. Here, the sibling of the double black node is black, and it's one of the children that is far from the double black node (node `48` in this example), and the nearer child is red (which is node `30` in this example).

We apply case `5`, given in *Table 14.1*. So, we swap the colors of the sibling and the sibling's child, which are near the double black node (in this example, node `30` and node `40`, so node `30` becomes black and node `40` becomes black), and the next step is that we perform the rotation operation on the sibling node in the opposite direction of the double black node. After applying case `5`, we can see the result in the tree in *Figure 14.17* (*d*). Still, the tree has the double black node, so we again check which case we can apply to convert this tree into a red-black tree.

After applying case `5`, we will always have to apply case `6`. We check the double black node's sibling, which is black, and the sibling's child, which is further away, is red (node `40` in this example). Since it is case `6`, given in *Table 14.1*, accordingly, we firstly swap the color of the parent and the sibling of the double black node (in this example, node `12` and node `30` are both black, so there will be no impact). Next, we perform the rotation of the parent node (node `12` in this example). The next step is to convert the double black node into a single black node. Next, we change the color of the red sibling of the double black node, which is further away from the black node (`40` in this example). Finally, we recheck the tree to see if it is following the properties of the red-black tree.

Here, it is following all the properties, so we stop here. This step-by-step process is shown in *Figure 14.17*:

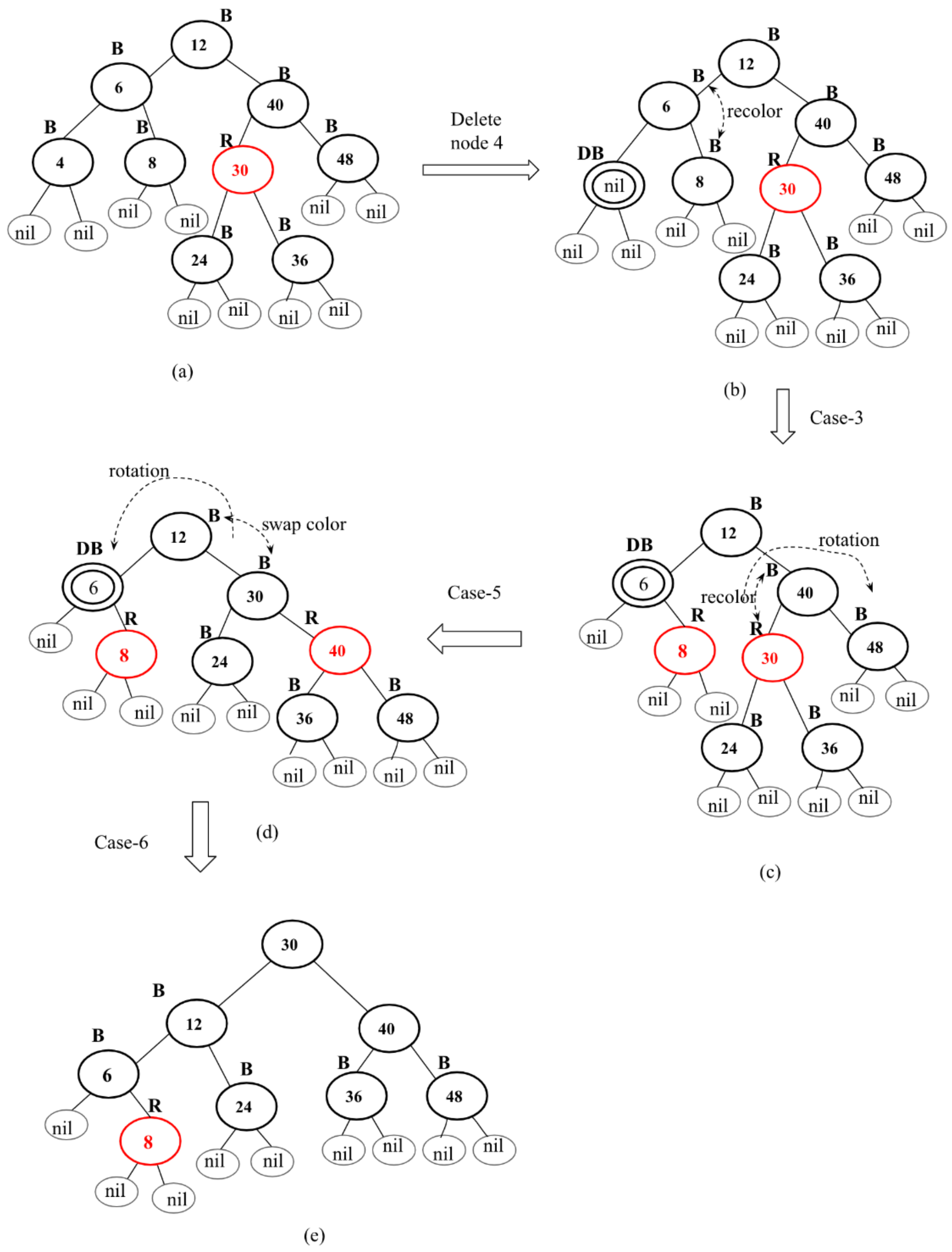

Figure 14.17: Deletion of node 4 from the red-black tree

To summarize, a red-black tree is a binary search tree with additional storage of color information per node, which is the color of the node. This can be black or red.

Whenever we modify the tree, the tree is rearranged so that the tree follows the properties of the red-black tree, which are designed in such a way that the tree remains balanced and guarantees the worst-case performance of `O(logn)`. The worst-case, best-case, and average-case cost of insert, delete, and search operations in the red-black tree will be bound by `O(logn)`. Next, let us discuss another variant of binary search tree, a B-tree.

The B-tree

A B-tree is a tree-type data structure that keeps all the data sorted, and also different operations such as searching, insertions, and deletions can be performed in logarithmic time. It is also known as a **height-balanced m-way tree**. A B-tree is a sort of self-balancing search tree in which each node can have more than two child nodes and more than one key. Here, the term *key* means the data values are stored in a node. Whereas in binary trees, each node can have a maximum of two children and exactly one key. It's a generalized version of a binary search tree.

When we need to store a large number of keys, the height of the tree will grow significantly in the case of a binary search tree, an AVL tree, etc., and the time complexity of the search, insert, or delete operations will also grow. Whereas in the case of B-trees, we may store many keys in a single node and have several child nodes. This considerably reduces the height.

Since a B-tree is a specific type of tree that can have a maximum number of children per node, the order of the B-tree specifies that maximum limit. A B-tree of the order *m* should have the following properties:

1. Every node can have a maximum of *m* children.
2. The minimum number of children of leaf nodes, root nodes, and internal nodes should be `0`, 2, and *m/2* respectively. Here, $\left\lceil \frac{m}{2} \right\rceil$ is the ceiling of *m/2* that is computed as the upper closest integer value, e.g., $\left\lceil \frac{5}{2} \right\rceil = \lceil 2.5 \rceil = 3$.
3. Every node should have a maximum of (*m-1*) keys.
4. The minimum number of keys belonging to the root node and internal nodes should be 1 and $\left\lceil \frac{m}{2} \right\rceil - 1$, respectively.
5. All leaf nodes should be at the same level.

6. Keys are stored in the sorted order in the B-tree. This means that different values stored in one node of the tree are sorted. For example, in *Figure 14.18*, the root node has two keys, `20` and `50`. Both keys are stored in sorted order.
7. Insertion of new nodes in the B-tree is bottom up, which means insertion will always be done at the leaf node.

Let's consider an example of a B-tree of order 5, as shown in *Figure 14.18*:

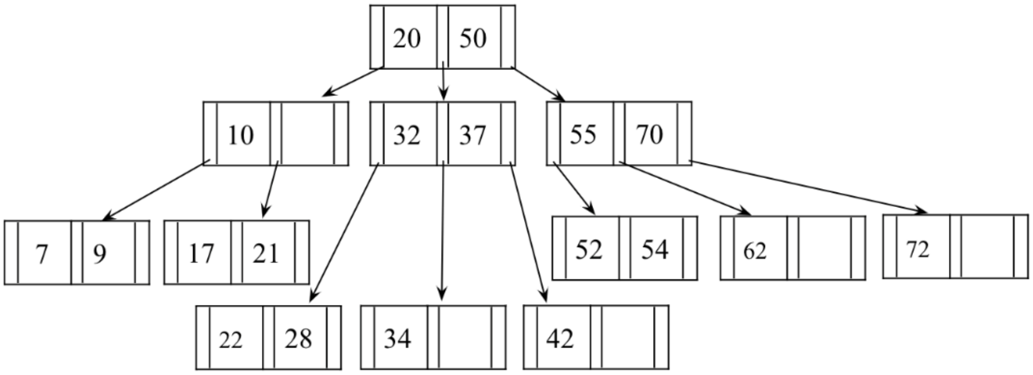

Figure 14.18: An example of a B-tree

In the above figure, we can see that every node has a maximum of 3 children. Internal nodes have at least `[3/2] = [1.5] = 2`. Next, each node has a maximum of 2 keys, and a minimum of 1 key in root and internal nodes. Further, all the leaf nodes are at the same level. We can also observe that all the keys in each node are sorted since *Figure 14.18* fulfills all the properties of the B-tree.

Next, let us discuss how to create a B-tree by inserting elements one by one into the tree.

Inserting elements into a B-tree

On a B-tree, inserting an element entails two steps: finding the right node to insert the element and, if necessary, splitting the node. The bottom-up technique is always used for insertion operations. The following steps are for the insertion operation in a B-tree:

1. Create a root node and insert the key if the tree is empty.
2. Change the node's maximum number of keys.
3. For insertion, look for the relevant node.
4. Follow the procedures below if the node is full.
5. Place the components in ascending order.

6. There are now elements that exceed its limit. As a result, divide by the median.
7. Make the left keys a left child and the right keys a right child by pushing the median key upwards.
8. Follow the procedures below if the node is not filled.
9. Place the nodes in ascending order.

Let us understand the creation of a B-tree of order 3 by inserting the elements `10`, `20`, `30`, `40`, `50`, `60`, `70`, `80`, `90`, and `100`. The order of the B-tree can be decided by the user. In this example, the order of the B-tree (*m*) is 3, so the maximum number of children can be 3 and the maximum number of keys can be (*m*-1)2.

Initially, we insert element `10` in the B-tree and ensure that all the properties of the B-tree are fulfilled. Next, we insert the key element `20` in the B-tree in the root node without any problem since the maximum number of key elements in the root is two. We also ensure that the key elements in a node are sorted. This process is shown in *Figure 14.19*:

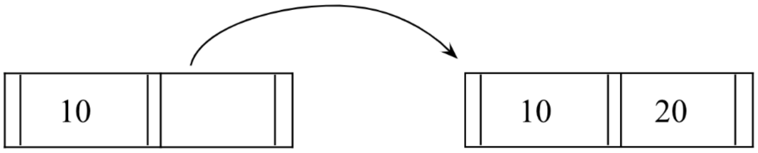

Figure 14.19: Insertion of key elements 10 and 20 in the B-tree

Next, we insert `30` in the B-tree. When we try to insert key element `30` into the root node, we will have to split the node since the maximum number of key elements in any node can be, at a maximum, two. The node is split from the middle point in such a way that the middle element is moved up in the tree so that the tree is growing in a bottom-up fashion. In this example, when we try to add key element `30`, the middle element is `20` for the key elements {`10`, `20`, `30`}, so the key element `20` is moved up in the B-tree and made into a new node. This process is shown in *Figure 14.20*:

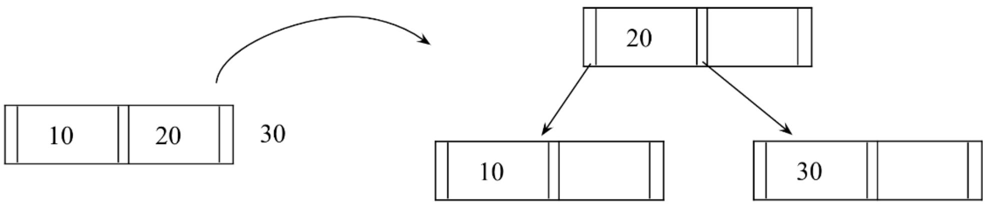

Figure 14.20: Insertion of key element 30 in the B-tree

Next, let us add key element `40` into the B-tree. As per the properties of a B-tree, we can only add new nodes at the leaf node, so we visit the tree to reach the leaf node. While traversing to reach the leaf node, we follow the binary search tree property (i.e., the key value of every node should be higher than all the key values in the left subtree and lower than all the key values in the right subtree). So, we reach the leaf node and, having key element `30`, we insert the new key element here in such a way that key elements are sorted. We insert the new element `40` into the leaf node as shown in *Figure 14.21*:

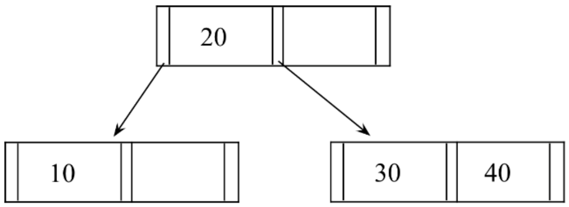

Figure 14.21: Insertion of key element 40 into the B-tree

Next, we add key element `50` into the tree. For this, we traverse the tree to reach the leaf node, and then we try to add `50` into the leaf node, having key elements {`30`, `40`}. Since this node cannot have more than two key elements, we have to split the node from the middle point, i.e., `40`. The middle key element (i.e., `40`) is moved up in the tree and added with the node of key element `20` in such a way that key elements are sorted. Furthermore, the remaining two key elements, `30` and `50`, are stored in two nodes in such a way that they follow the binary search tree property. This process is shown in *Figure 14.22*:

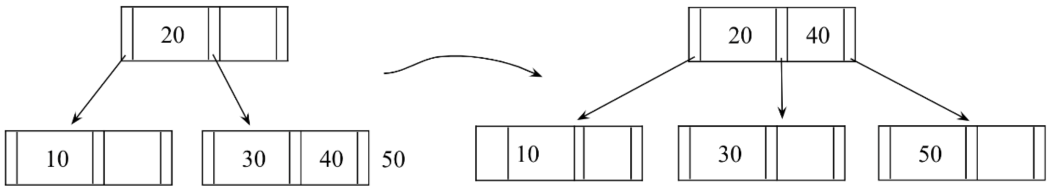

Figure 14.22: Insertion of key element 50 into the B-tree

Next, we add `60` into the B-tree. We reach the leaf node according to the binary search tree property having key element 50.

Here, when we try to insert key element 60, we can insert it since this node has 1 key element, and we can store up to 2 key elements. This process is shown in *Figure 14.23*:

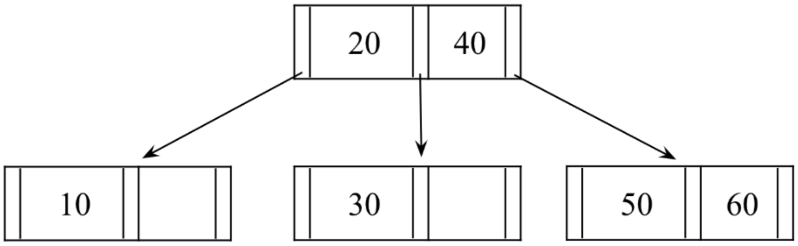

Figure 14.23: Insertion of key element 60 into the B-tree

Now we add the key element `70` into the B-tree. We start with reaching the leaf node where we can add key element `70`, which is the node that has the key elements {`50`, `60`}. Here, we cannot add key element `70` since the maximum number of key elements is two, hence we split the node with the middle element. The middle key element is `60`, which is moved up since we moved the key element `60` to the upward node.

In the upward node, we already have two key elements {`20`, `40`}, hence, we will again have to split this node from the middle key element. The middle key element is `40`, which should be moved up in the B-tree. So, we split the node and make a new node with key element `40`. We can observe here that the tree is growing in a bottom-up fashion. The process of adding key element `70` is shown in *Figure 14.24*:

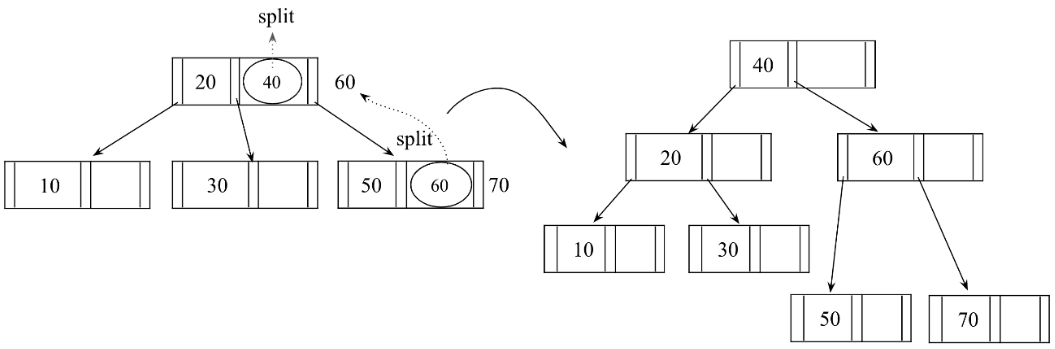

Figure 14.24: Insertion of key element 70 into the B-tree

Next, we add key element `80` into the B-tree. We traverse the tree to reach the appropriate leaf node. To add key element `80`, we reach the node that has key element `70`. Here, we can insert key element `80` in such a way that all the keys in the node are sorted, as shown in *Figure 14.25*:

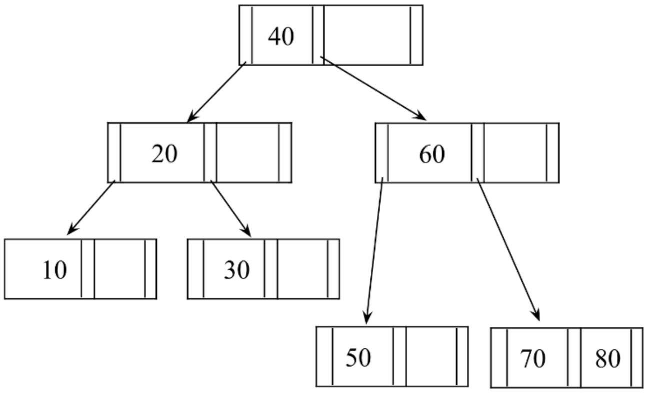

Figure 14.25: Insertion of key element 80 into the B-tree

Next, we add key element `90` into the B-tree. Firstly, we reach the appropriate leaf node, i.e., the node that has key elements {`70`, `80`}, then we split from the middle key element, i.e., key element `80` is moved up with the node having key element `60`, as shown in *Figure 14.26*:

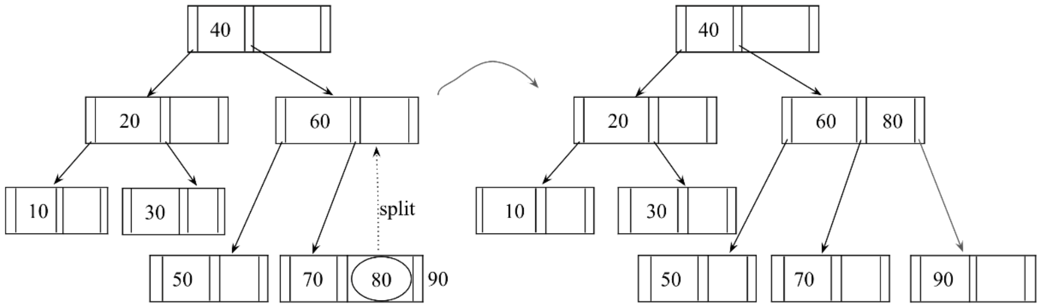

Figure 14.26: Insertion of key element 90 into the B-tree

Next, we insert key element `100` into the B-tree and traverse the tree following the binary search tree property, reaching the appropriate leaf position where we can insert key element `100`. Accordingly, we reach the leaf node that has key element `90`, so we can add key element `100` in such a way that the keys are sorted.

This process is shown in *Figure 14.27*:

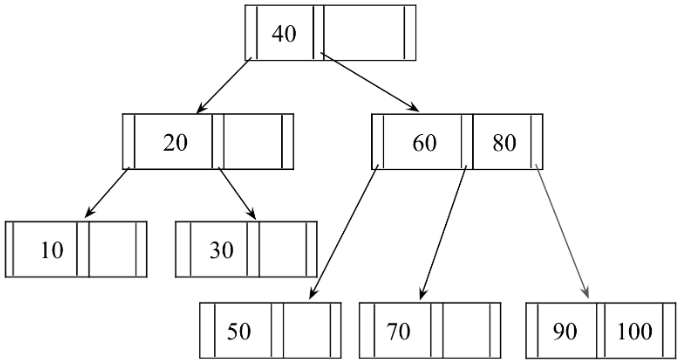

Figure 14.27: Insertion of key element 100 into the B-tree

The figure shows the final B-tree. Next, let us understand how we can delete a key element from any node of the B-tree.

Deletion operation in a B-tree

A deletion operation in a B-tree can be performed in the following steps:

1. Search the node where the key element is present
2. Delete the key element
3. Balance the tree in order to keep the properties of the B-tree intact if required

While deleting a key element from the B-tree, we have to ensure that the properties of the B-tree are intact, so that each node can have a maximum of *m* children and *m* – 1 number of keys. And each node (except for the root node) should have a minimum of $\left[\frac{m}{2}\right]$ children, and $\left[\frac{m}{2}\right] - 1$ number of keys.

When we wish to delete any key element from the B-tree, there can be the following cases:

1. The key element that we want to delete (we call it a **target key**) can be presented in the leaf node
2. The target key element is present in the internal node

Let us discuss and understand these cases with examples.

Case 1: Target key is present in the leaf node

When the target key to be deleted is present in the leaf node, we will have one of two cases:

- The node from which the target key is to be deleted contains more than the minimum number of keys.
- The node from which the target key is to be deleted contains just the minimum number of keys.

In the first case, it is straightforward to delete the node since directly deleting the key element from such a leaf node that has more than the minimum number of keys does not violate the properties of the B-tree. Let us take an example of deleting key element 37 from the B-tree shown in *Figure 14.28*. Key element 37 is deleted directly because deleting this does not violate the properties of the required minimum number of keys in the B-tree. The target key element and node are shown within the dotted line in *Figure 14.28*:

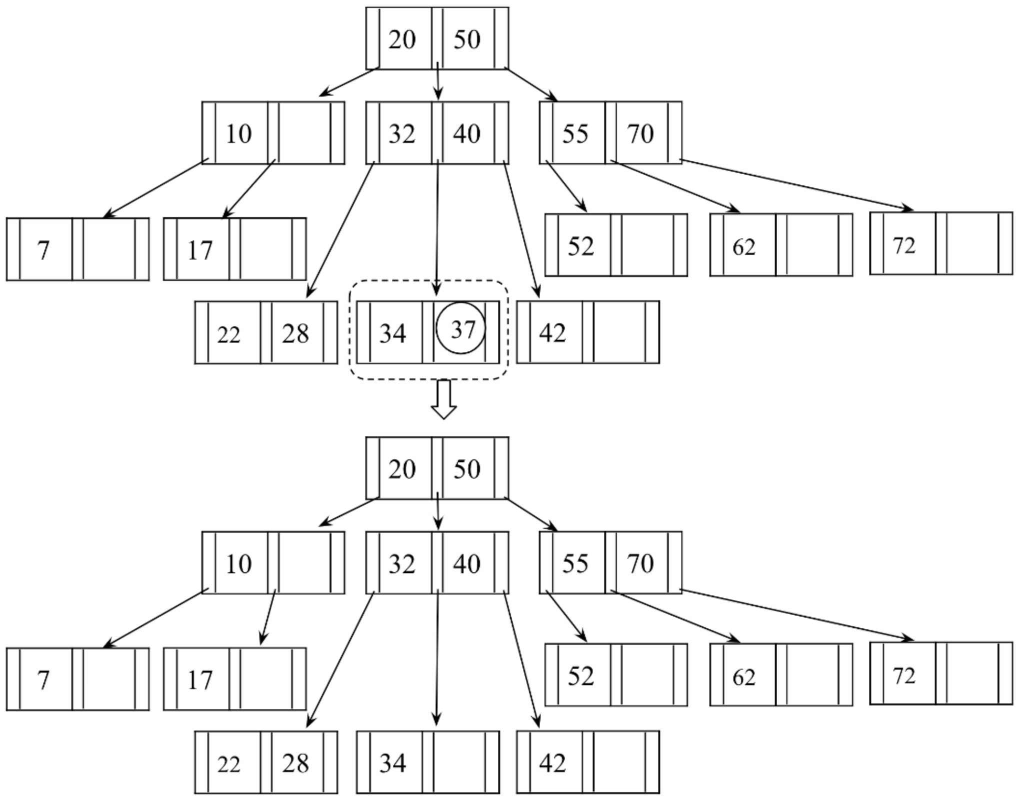

Figure 14.28: Deletion of key element 37 from the B-tree

In the second case, when we have to delete the key element from a node that has the minimum number of keys, then deleting one key element will violate the properties of the B-tree. In such a case, we can borrow a key element from its immediate neighboring sibling node.

It works as follows: firstly, we visit the immediate left sibling. If the left sibling node has more than the required minimum number of keys, then we borrow one key. Otherwise, we visit the right sibling node, and if it has more than the minimum keys, we can borrow from there.

Let us understand this with an example in which we want to delete key element 34 from the given B-tree shown in *Figure 14.29*. In this example, we can borrow a key from the left neighbor sibling, which has two key elements {22, 28}, which is more than the minimum required number of keys. So, in this example, we can borrow a key from this node. Borrowing a key element will be done through the parent node. For this, we transfer the maximum value in the sibling node to the parent node (the maximum key element in this case is 28), then the key element from the parent node (i.e., 32 in this example) will be transferred to the node from which we are deleting the target key (i.e., 34 in this example). Finally, the target key can be deleted. This process is shown in *Figure 14.29*:

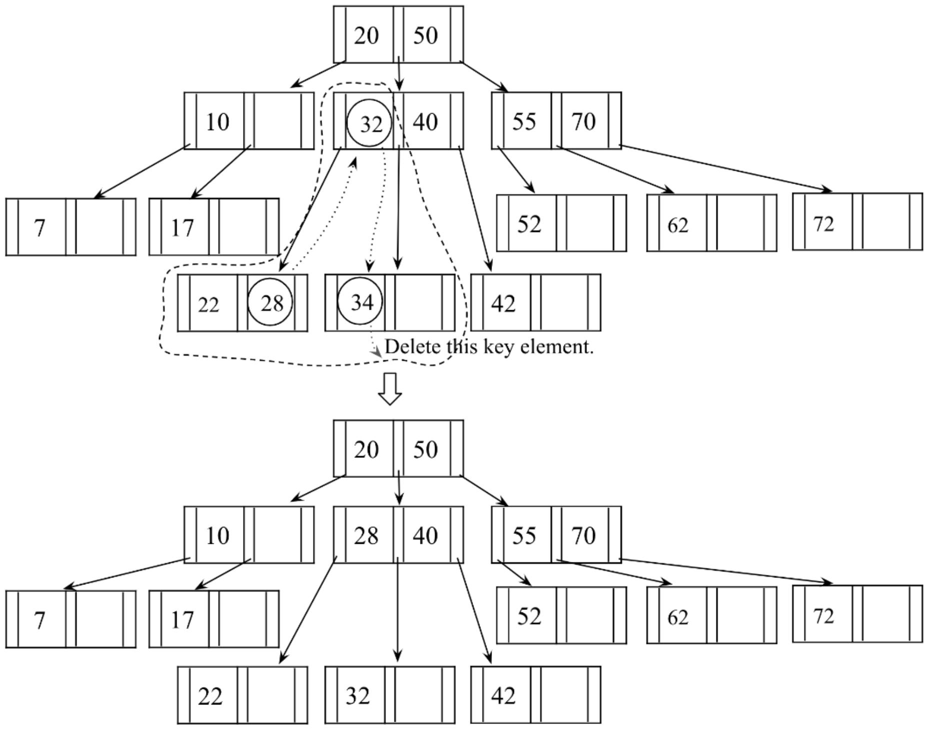

Figure 14.29: Deletion of key element 34 from the B-tree

Similarly, a key element can be borrowed from the right immediate sibling if the left sibling does not have more than the required minimum number of keys through the parent node. However, it is possible that both the immediate siblings do not have the minimum number of keys. In that case, we merge the node with either the left immediate sibling or the right sibling node, also through the parent node.

Let us understand this concept with an example, as shown in *Figure 14.30*. In this B-tree, let us delete key element 32. We can see that in this case, the left and right immediate siblings already have the minimum number of keys. We will have to merge the nodes either with the left or right immediate sibling, so we will merge with the left sibling. Merging is always performed through the parent node. For this, we consider the key element of the parent, which is in between two nodes, i.e., the left sibling node and the node that we are deleting (the nodes with keys 22 and 32 in this example). We merge by creating a node with the key element from the left sibling, and the parent node, and then the target key is deleted. This process is shown in *Figure 14.30*:

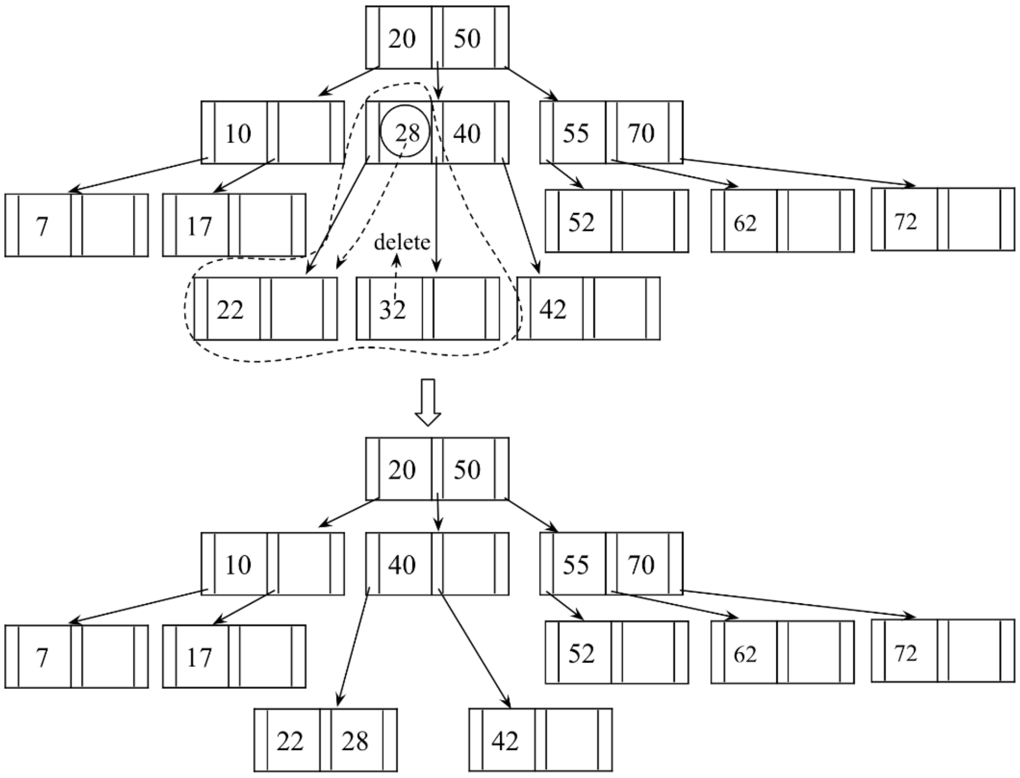

Figure 14.30: Deletion of key element 32 from the B-tree

Now, let us understand how we can delete a key element when the target key is present in the internal node.

Case 2: Target key is present in the internal node

When the target key to be deleted is present in the internal node of the tree, the following cases can occur:

1. The internal node, which is to be deleted, is replaced by an inorder predecessor if the left child of the target key has more than the minimum number of required keys as per the properties of the `B`-tree. The inorder predecessor is the largest key element of the left subtree of the node.
2. The internal node, which is to be deleted, is replaced by an inorder successor, which is the smallest key element of the right subtree of the node, if the right child has more than the minimum required number of keys.
3. If both the left and right child of the target key only have the minimum number of required keys, then we delete the target key, and then we merge the left and right child, but only if the node having the target key element has more than the minimum number of required key elements. That means the deletion of the target key does not lead to less than the minimum number of required keys in the node.
4. If both the left and right child of the target key have just the minimum number of required keys and the node having the target key element also has just the minimum number of required elements, then we merge the left and right child. Then, a key element is borrowed from its neighboring node through the parent node.

Firstly, let us discuss case 1 with an example in which we delete key element `40` in the `B`-tree, as shown in *Figure 14.31*. We can see in the `B`-tree shown in *Figure 14.31* that the target key element `40` is present in the internal node. The left child of the target key has more than the minimum number of required keys, i.e. two `{34, 37}`, so that we can replace the target key element with the inorder predecessor (which is `37` in this example).

This is shown within the dotted lines in *Figure 14.31*:

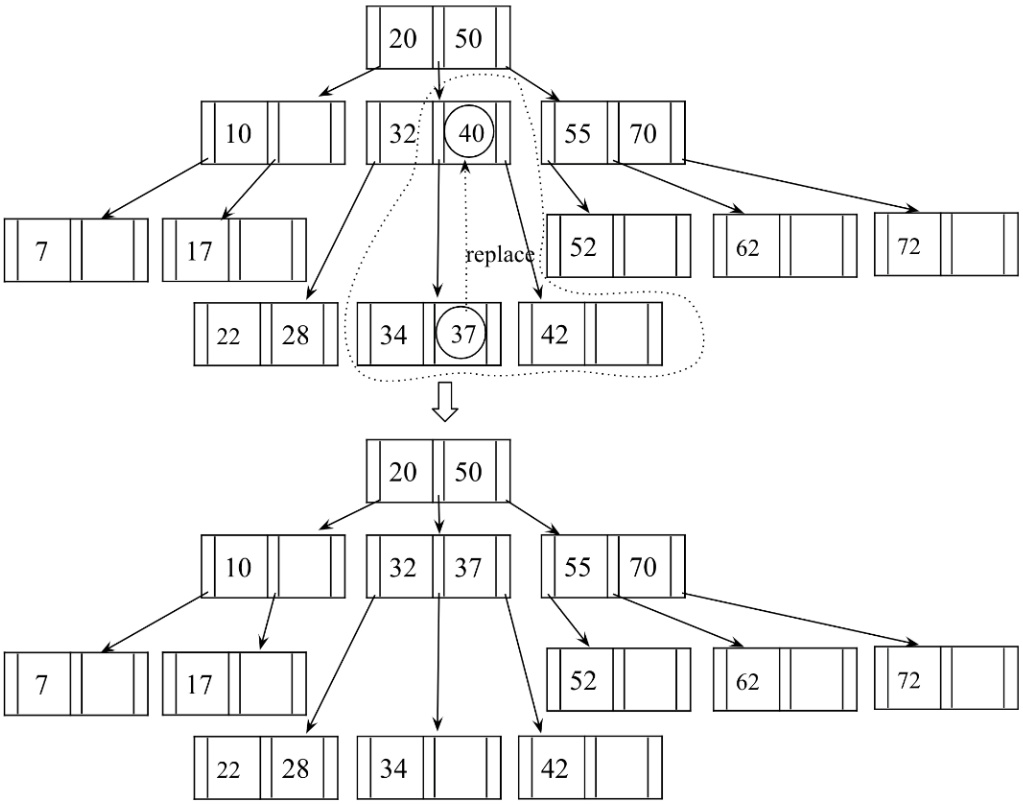

Figure 14.31: Deletion of key element 40 from the B-tree

Next, let us discuss case 2 with an example in which we delete key element 37 from the B-tree, as shown in *Figure 14.32*. We can see in the B-tree that the target key element 37 is present in the internal node, so we first check if its left child has more than the minimum number of required keys, which is one in this case, so we cannot replace the target element with it.

Next, we check if the right child of the target key has more than the minimum number of required keys, which is two, {i.e., 42, 47}. So, we can replace the target key element with the inorder successor (which is 42 in this example). This process is shown within the dotted lines in *Figure 14.32*:

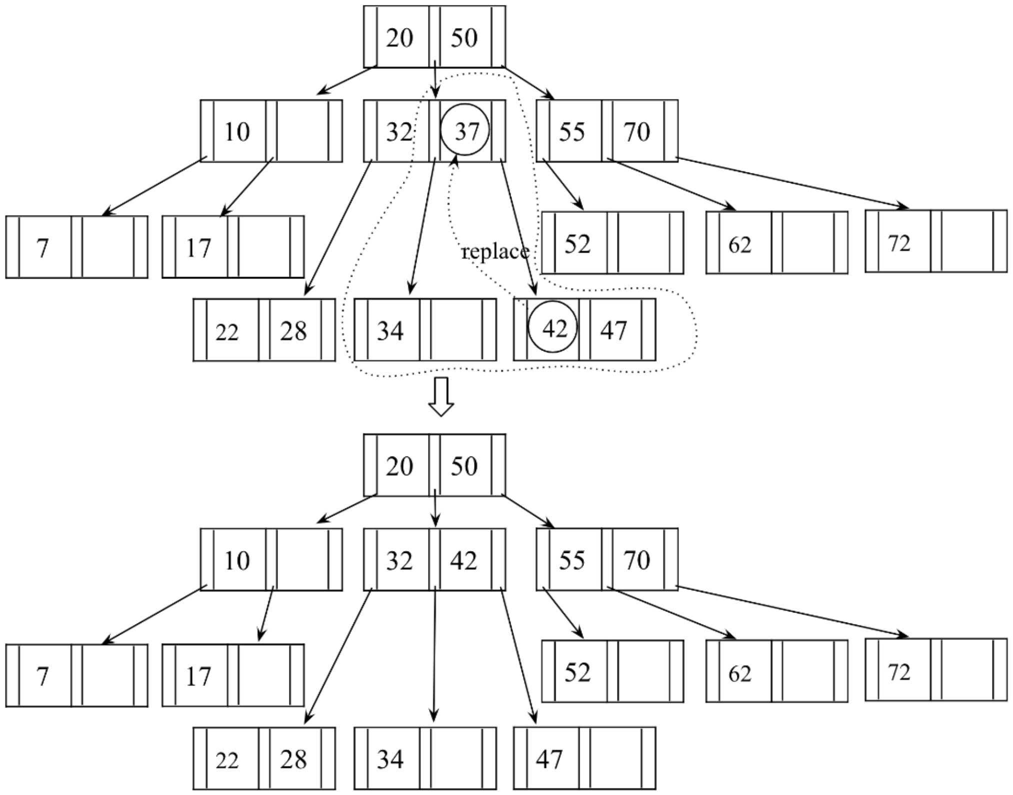

Figure 14.32: Deletion of key element 37 from the B-tree

Next, we can have case 3, which is to delete key element 42 in which both the children of the target key elements have exactly the minimum number of required key elements. In that case, we merge the left and right child nodes. Here, we can see that the node has a target key element of more than the minimum number of required keys, i.e. {32, 42}. So, we can merge the left and right children and delete the target key.

Let us discuss this with an example of deleting key element 42, both the left and right child nodes of which have the exact minimum number of required keys, so we can delete the target key element and merge both the child nodes, as shown in *Figure 14.33*:

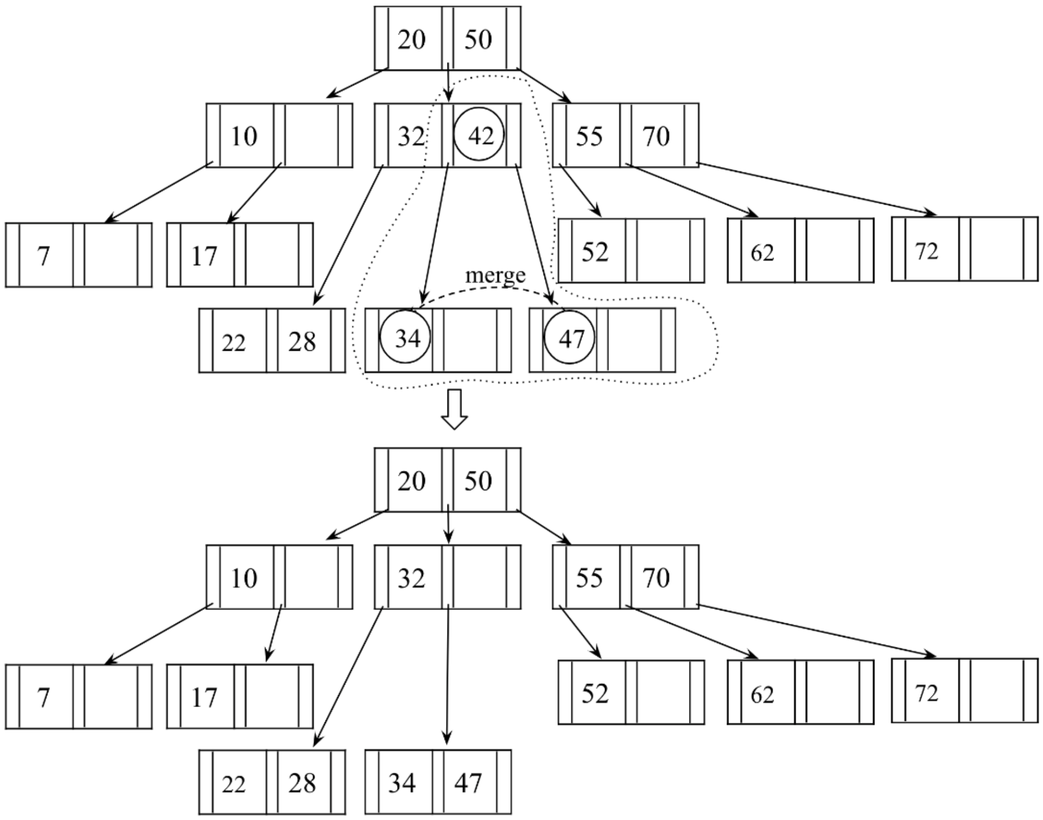

Figure 14.33: Deletion of key element 42 from the B-tree

Next, let us understand case 4 with an example in which we delete key element `10`, where both the left and right child of the target key elements have exactly the minimum number of required key elements, but deleting the target key will make the node violate the condition of the minimum number of keys. So, we merge the left and right child nodes (keys `7` and `17` in this example), and then we borrow a key element from the neighboring key element through the parent node. The process is shown in *Figure 14.34*:

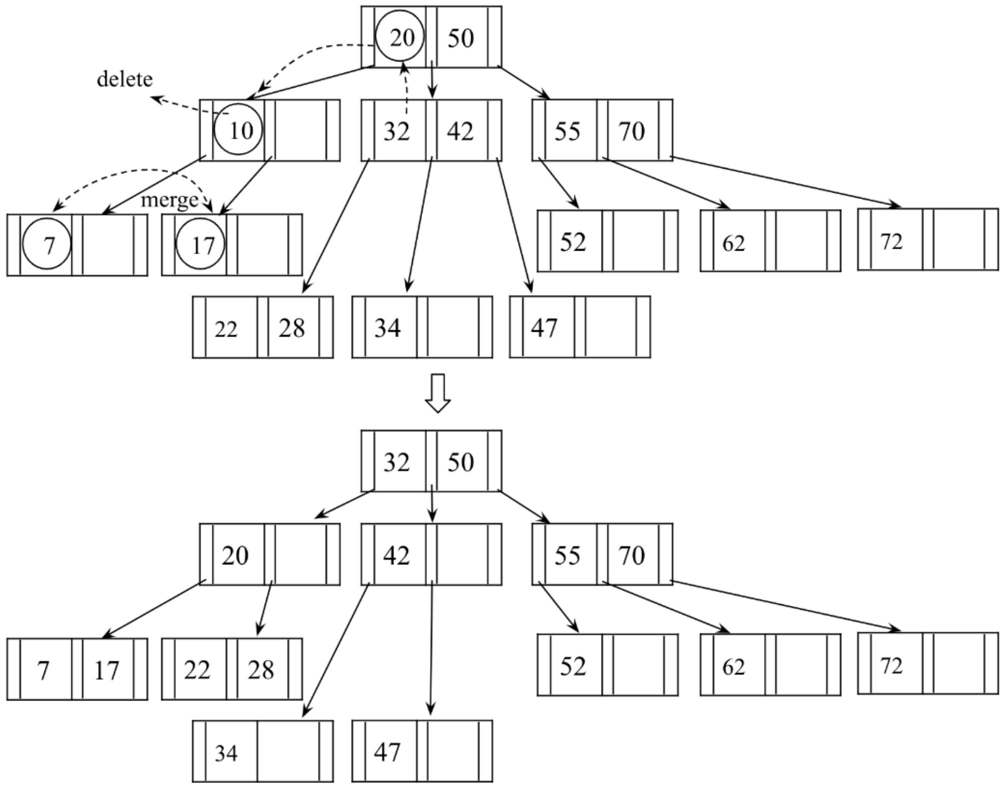

Figure 14.34: Deletion of key element 42 from the B-tree

Finally, we shall look into how the B-tree can shrink with an example of the deletion of key element `10` in the B-tree, as shown in *Figure 14.35*. Here, the target key element is present in the internal node, and its left and right child nodes have exactly the minimum number of elements (i.e., `7` and `17`). Deleting the target key can make the node violate the condition of the minimum number of key elements. So, we merge the left and right child nodes, and then we look for the sibling of the node having the target key so that we can borrow a key. But in this case, the sibling itself has exactly the minimum number of keys, so we merge the node with the sibling along with the parent. Finally, we arrange the child nodes accordingly in increasing order. This process is shown in *Figure 14.35*:

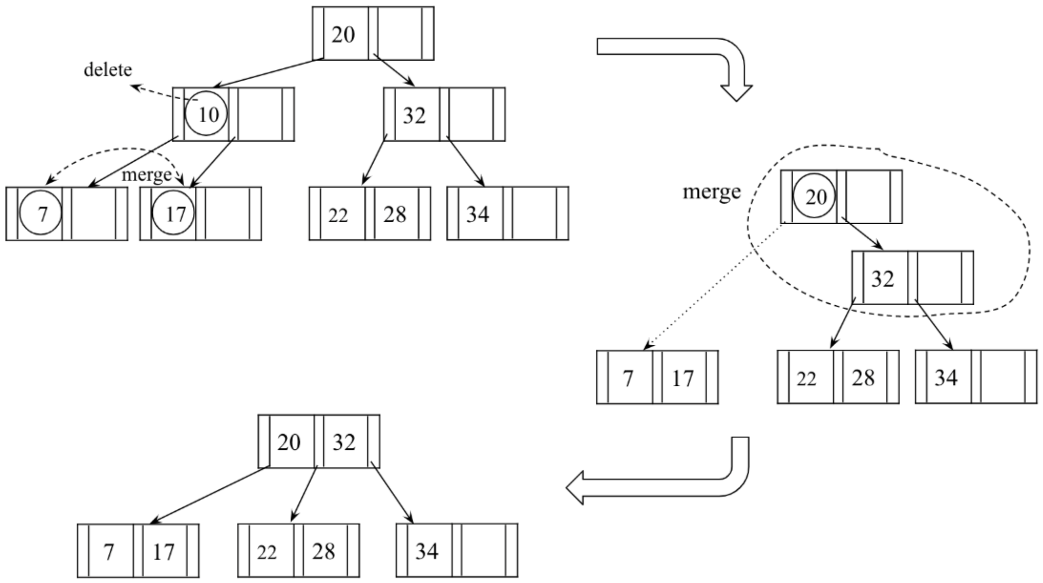

Figure 14.35: Deletion of key element 10 from the B-tree

The average-case, best-case, and worst-case complexity of the insert, search, and delete operations in the B-tree is `O(log n)`.

The AVL, red-black, and B-tree data structures are commonly used to balance search trees, providing the insertion, deletion, and search operations in guaranteed `O(log n)` time. AVL trees are more strictly balanced and hence provide faster lookups. So, AVL trees are useful when we have a lookup-intensive task. When we have an insertion- or deletion-intensive application, we use a red-black tree since it requires very few rotations as compared to AVL trees and B-trees. B-trees are used for indexing data and they provide fast access to data stored on disks. The searching of data is very efficient using a B-tree data structure.

Summary

In this chapter, we discussed how we can ensure that a binary search tree is always balanced. It is important because a balanced tree gives us a guaranteed performance of `O(logn)` for insertion, deletion, and traversal operations. We have discussed AVL trees, red-black trees, and `B`-trees in detail.