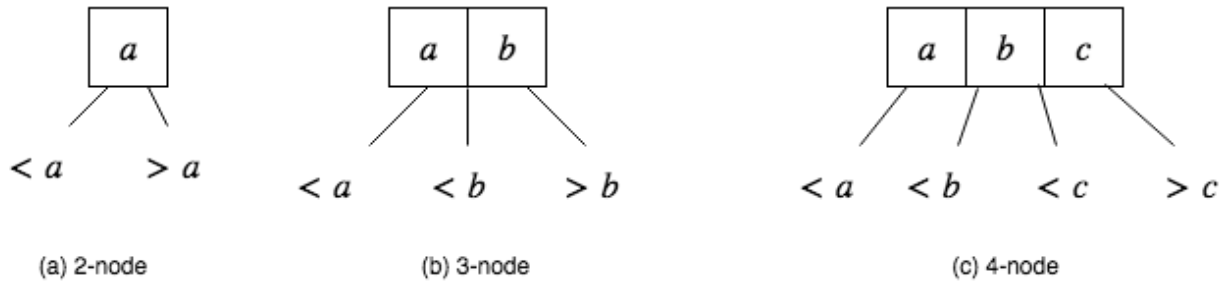
2-3-4 Trees

Introduction

A 2-3-4 tree is a balanced search tree having following three types of nodes.

1. **2-node** has one key and two child nodes (just like binary search tree node).
2. **3-node** has two keys and three child nodes.
3. **4-node** has three keys and four child nodes.

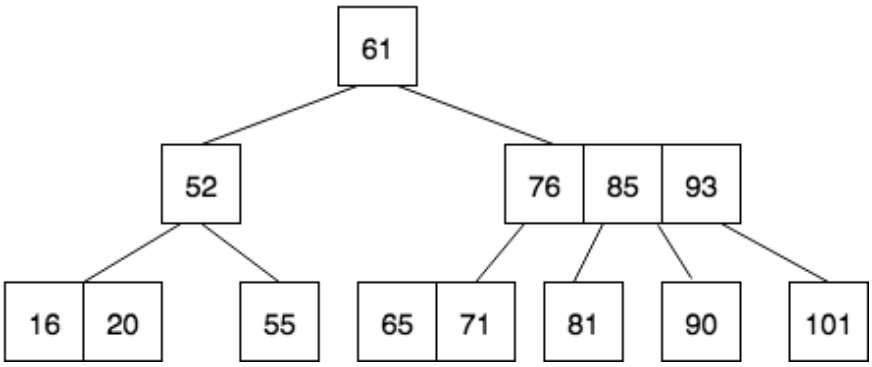
The reason behind the existence of three types is to make the tree perfectly balanced (all the leaf nodes are on the same level) after each insertion and deletion operation. Figure 1 illustrates these node types graphically.



(

Figure 1: Illustrating node types

If a node has more than one keys (3-node and 4-node), the keys must be in the sorted order. This makes sure that the in-order traversal always yields the keys in sorted order. An example of a 2-3-4 tree is given in Figure 2.



(

Figure 2: An example of a 2-3-4 tree

Operations on 2-3-4 Trees

We discuss three major operations on the 2-3-4 tree. The first one and the most straightforward is the search, the next is insert and the last one is delete. All the operations take $O(\log n)$ time since the height of the tree is in the logarithmic order.

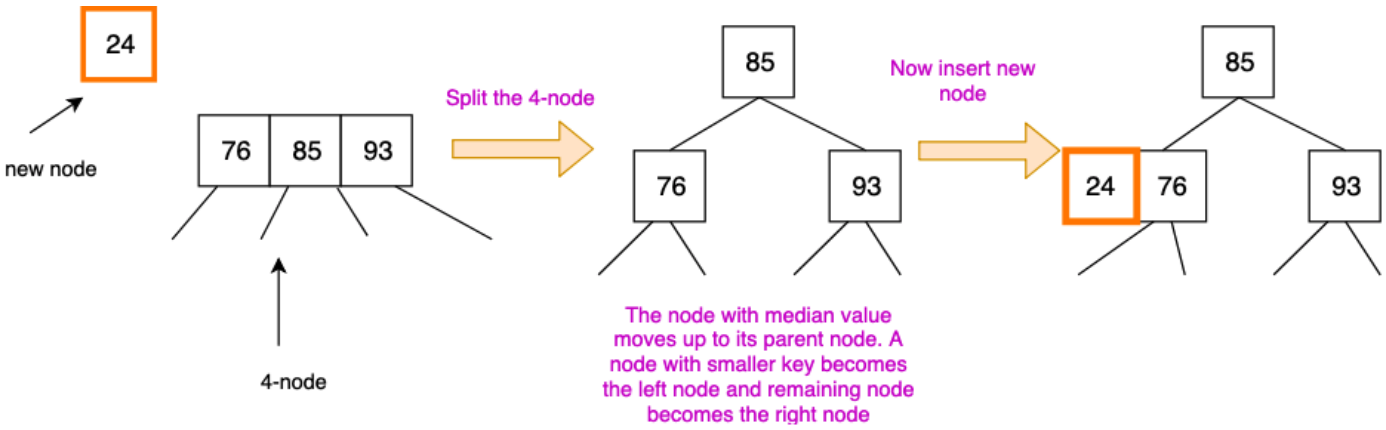
Search Operation

Compare the item to be searched with the keys of the node and move to the appropriate direction. Unlike BST where we move either to the left child or to the right child, we need to make choice among three or four different paths. Since the keys are sorted, it is obvious to choose the path following the rule given in Figure 1.

Insert operation

> A node cannot hold more than three keys. If a node is full before insertion, we split the node so that the new node can be inserted.

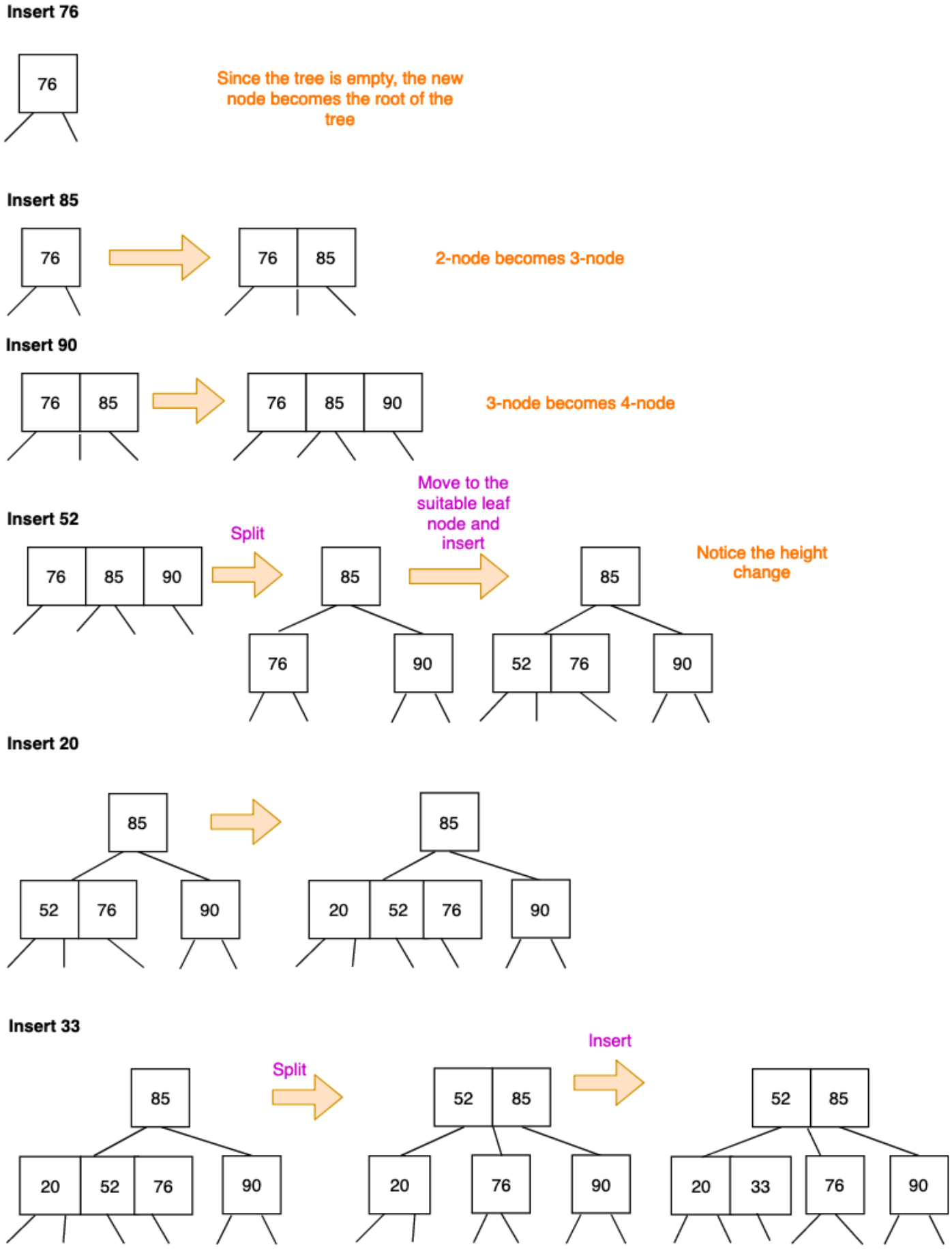
The insertion takes place always on the leaf nodes. I repeat again, *we never insert a new node on the internal nodes even if they have room to accommodate*. Therefore, we perform the search operation on the tree until we reach the leaf node. If the leaf node is a 2-node, we insert the item and make it a 3-node. Similarly, if the leaf node is a 3-node, we make it a 4-node. But what if the node is a 4-node? We can not insert a new node in this node right? The node is already full. In this case, we split the node that splits into nodes with a smaller number of keys and inserts the new node in the appropriate child node. This is illustrated in figure 3.



(

Figure 3: Splitting of a 4-node before inserting a new node.

The splitting process can, sometimes, go up to the root node. This happens when all the nodes on the path from the root to the leaf node are 4-nodes. When the root node splits, it increases the height of the tree by 1. Figure 4 illustrates the insertion operation.



(

Figure 4: illustrating insertion operation

Delete Operation

Delete is a bit tricker than insert operation. Depending upon the location of the node containing the target (x) to be deleted, we need to consider several cases. I am going to explain each of the cases one by one.

Case 1: If x is in a leaf node

This has further two cases.

Case 1.1: If x is either in a 3-node or 4-node

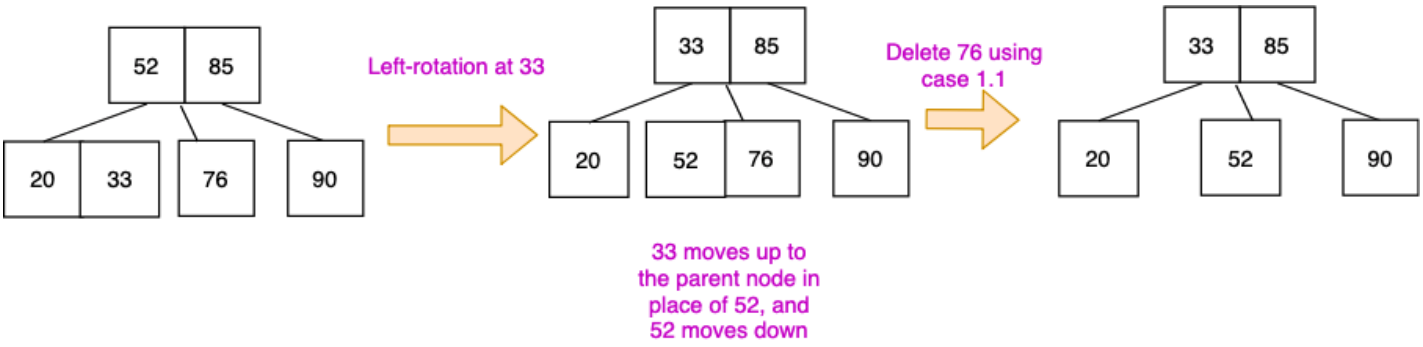
Delete x. If the node is a 3-node, it becomes 2-node and if the node is a 4-node, it becomes 3-node.

Case 1.2: If x is in a 2-node

This is called *underflow*. To resolve this, we need to consider further three cases.

Case 1.2.1: If the node containing x has 3-node or 4-node siblings

Convert the 2-node into a 3-node by stealing the key from the sibling. This can be done by left or right rotation. If the left sibling is a 3-node (or 4-node), do the left rotation otherwise do the right rotation. The left rotation is illustrated in figure 5. Here node with key 76 is being deleted.



(

Figure 5: Illustrating the left rotation

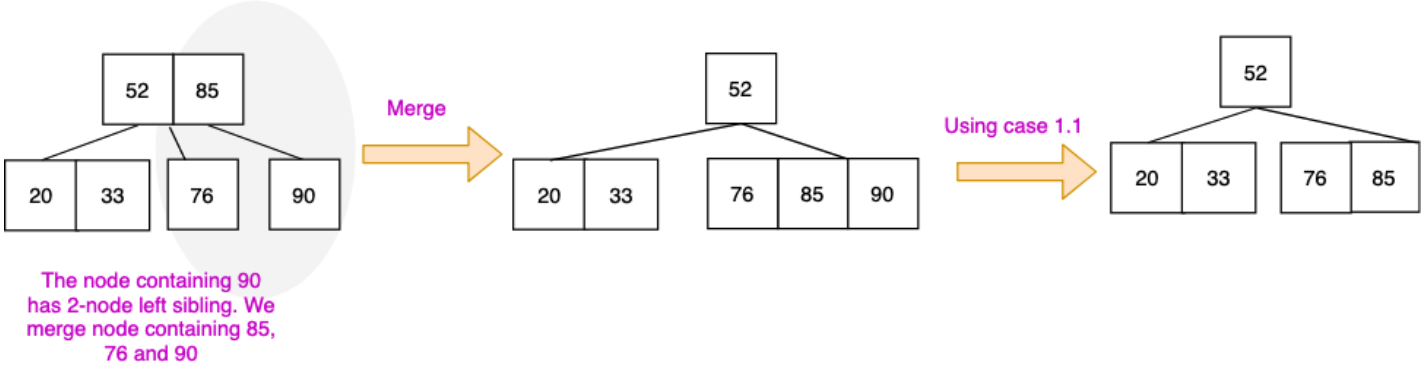
The right rotation is symmetric. After the rotation, use case 1.1 to delete x.

Case 1.2.2: If both the siblings are 2-node but the parent node is either a 3-node or a 4-node

In this case, we convert the 2-node into a 4-node using the merge (or fusion) operation. We merge the following three nodes.

1. The current 2-node containing x.
2. The left or right sibling node (which is also a 2-node).
3. The parent node corresponding to these two nodes.

After the fusion, the keys in the parent node decreases by 1. This is illustrated in figure 6 where a 2-node containing 90 is being deleted.



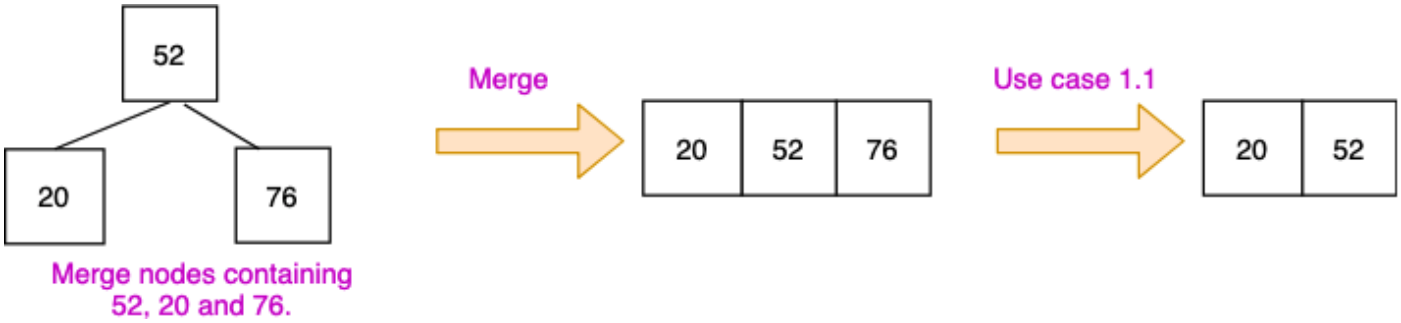
(

Figure 6: The merge operation

After the merge, we use case 1.1 to delete x.

Case 1.2.3: If both siblings and the parent node are a 2-node

We encounter this scenario rarely. In this particular case, the parent node must be a root. Just like fusion, we combine both the siblings and the parent node to make it a 4-node. This process shrinks the height of the tree by 1. Figure 7 illustrates this. The node containing 76 is being deleted.



(

Figure 7: The shrink operation

Case 2: If x is in an internal node

In this case, we do the following

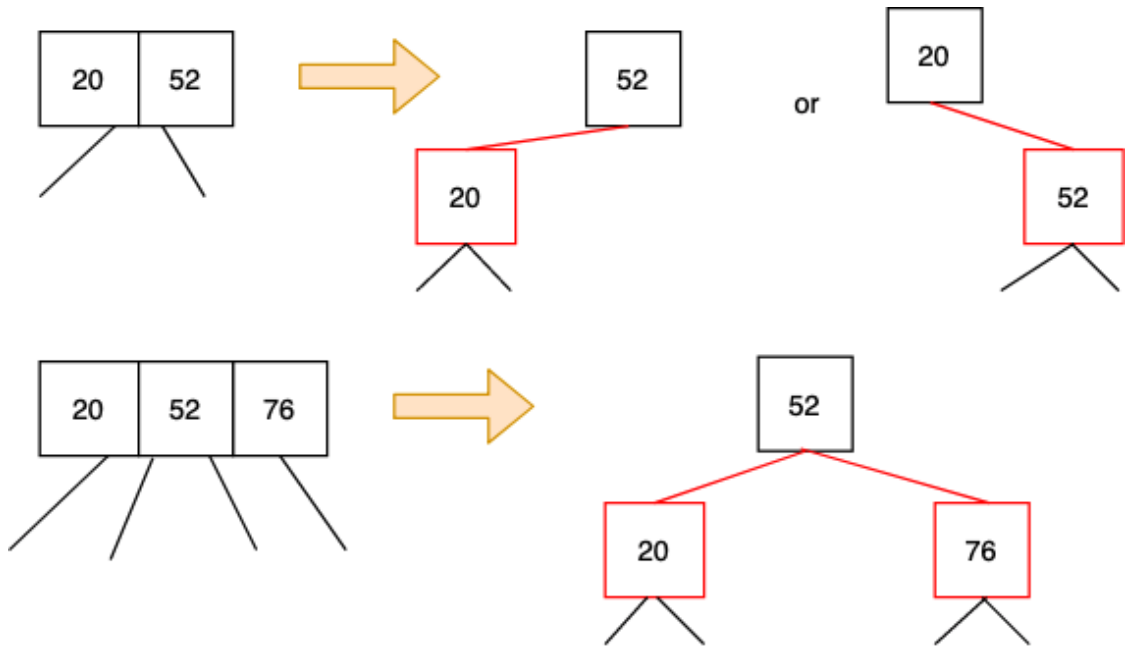
1. Find the predecessor of the node containing x. The predecessor is always a leaf node.
2. Exchange the node containing x with its predecessor node. This moves the node containing x to the leaf node.
3. Since x is in a leaf node, this is case 1. Use the rules given in case 1 to delete it.

Implementation

The implementation of a 2-3-4 tree is not straightforward. Three types of node and frequent switching of the node to different type make the implementation difficult. Therefore, we do not implement a 2-3-4 tree rather we study it from a theoretical viewpoint. An equivalent data structure of 2-3-4 trees is called a Red-Black tree. Being the binary search tree, Red Black trees are much easier to implement. In the next section, we discuss the mapping of a 2-3-4 tree to a red-black tree.

Mapping a 2-3-4 tree into a red-black tree

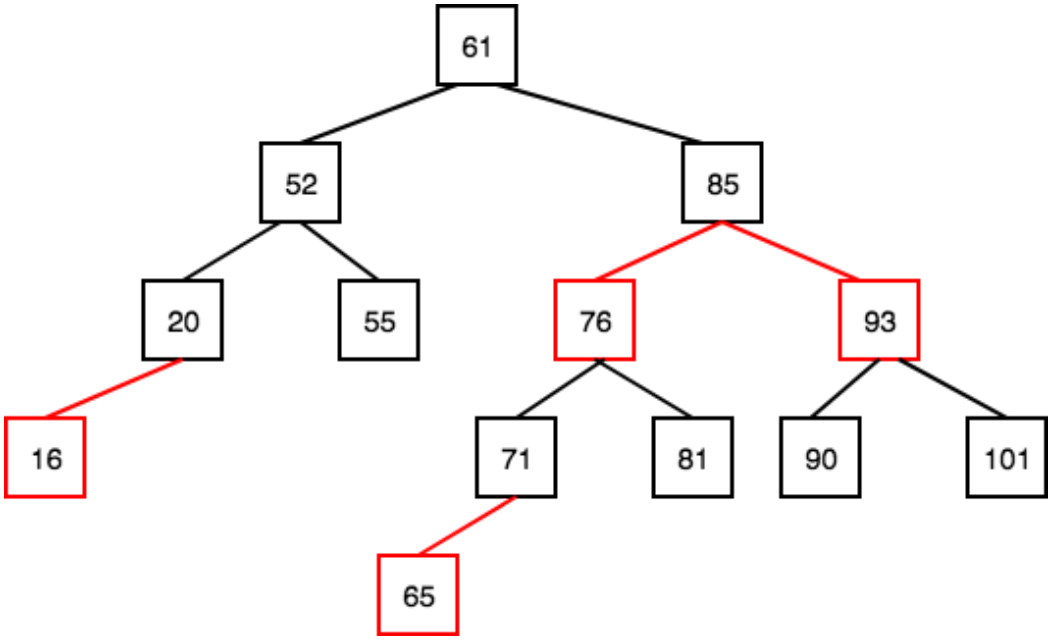
A 2-node in a 2-3-4 tree becomes a black node in a red-black tree. The mapping of a 3-node and a 4-node is illustrated in Figure 8.



(

Figure 8: Mapping a 3-node and a 4-node into the red-black tree nodes

When the above rules are applied to every node of the 2-3-4 tree, it becomes a red-black tree. The red black tree in figure 9 is an isometry of a tree given in figure 2.



(

Figure 9: An equivalent red-black tree of the 2-3-4 tree given in figure 2

While mapping, we make the following observations.

1. Every node of a red-black tree is either red or a black.
2. The root is always a black node.
3. All the paths from the root the leaf nodes contain the same number of black nodes. This is because the 2-3-4 tree is perfectly balanced.
4. We can not have two consecutive red nodes along the same path.
5. If a node is red, both of its child nodes must be black.

These are the five properties of a red-black node. These properties make it a balanced search tree.

2-3 Trees (/Data-Structures/Tree/2-3-Trees/)

Red Black Trees (/Data-Structures/Tree/Red-Black-Trees/)