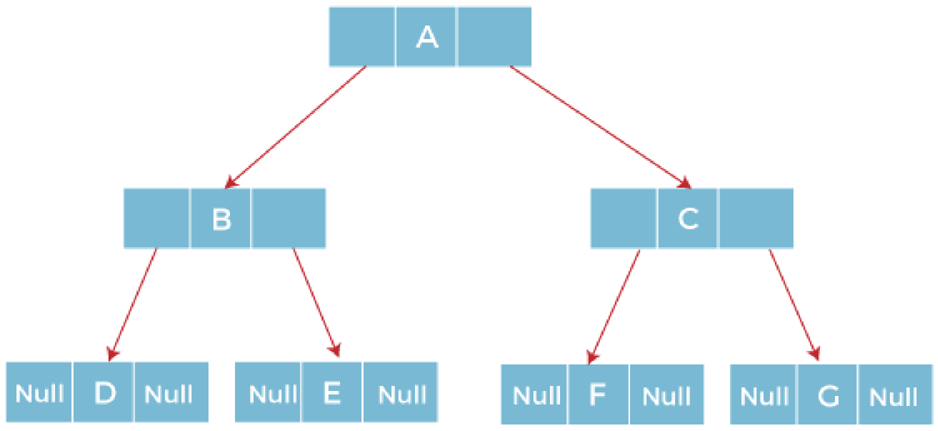
ADVANCE DATA STRUCTURE NOTES
UNIT-II

Threaded Binary Tree

In this article, we will understand about the threaded binary tree in detail.

What do you mean by Threaded Binary Tree?

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.



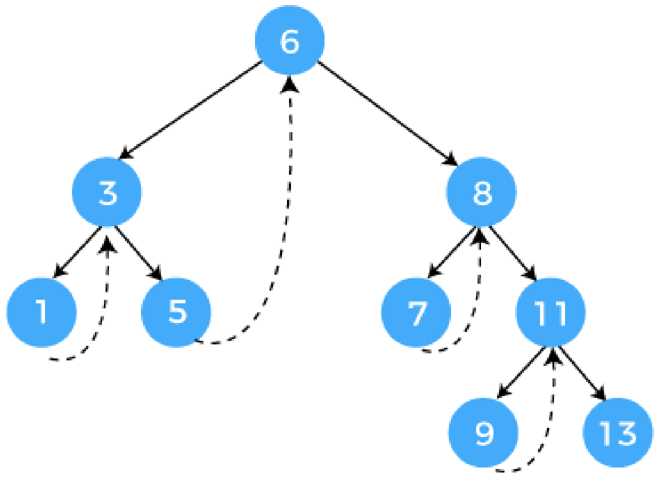
Threaded binary tree

Types of Threaded Binary Tree

There are two types of threaded Binary Tree:

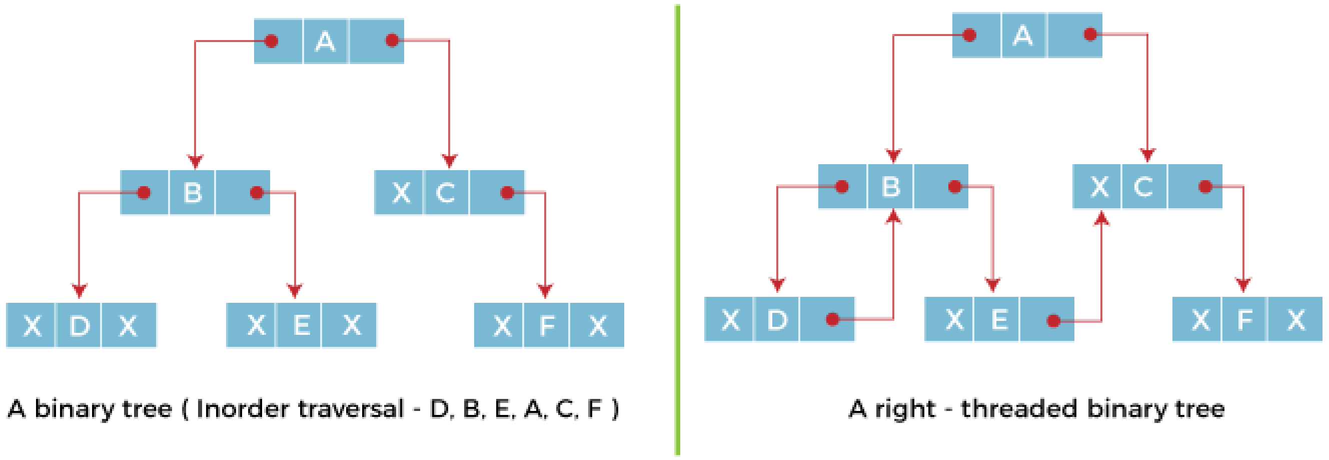
- One-way threaded Binary Tree
- Two-way threaded Binary Tree

One-way threaded Binary trees:



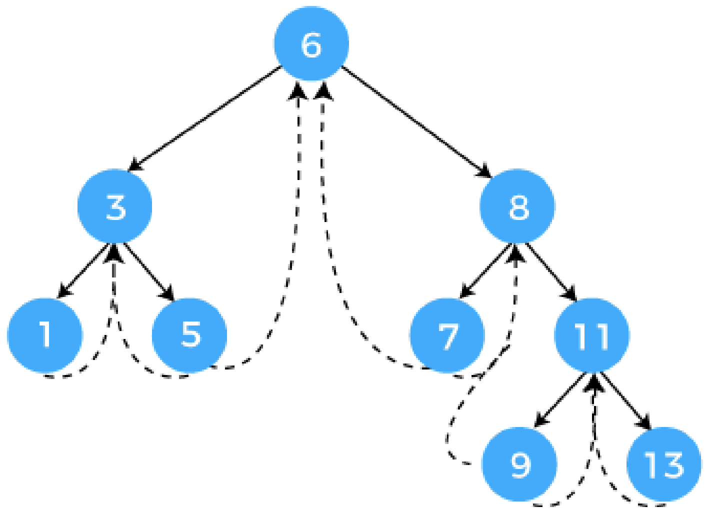
Single Threaded Binary Tree

In one-way threaded binary trees, a thread will appear either in the right or left link field of a node. If it appears in the right link field of a node then it will point to the next node that will appear on performing in order traversal. Such trees are called **Right threaded binary trees**. If thread appears in the left field of a node then it will point to the nodes inorder predecessor. Such trees are called **Left threaded binary trees.** Left threaded binary trees are used less often as they don't yield the last advantages of right threaded binary trees. In one-way threaded binary trees, the right link field of last node and left link field of first node contains a NULL. In order to distinguish threads from normal links they are represented by dotted lines.


A
B
X C
X D X
X E X
X F X
A binary tree (Inorder traversal - D, B, E, A, C, F)
A
B
X C
X D
X E
X F X
A right - threaded binary tree

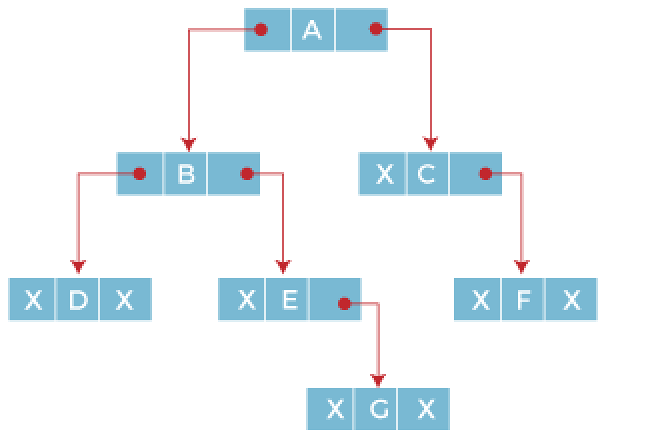
The above figure shows the inorder traversal of this binary tree yields D, B, E, A, C, F. When this tree is represented as a right threaded binary tree, the right link field of leaf node D which contains a NULL value is replaced with a thread that points to node B which is the inorder successor of a node D. In the same way other nodes containing values in the right link field will contain NULL value.

Two-way threaded Binary Trees:

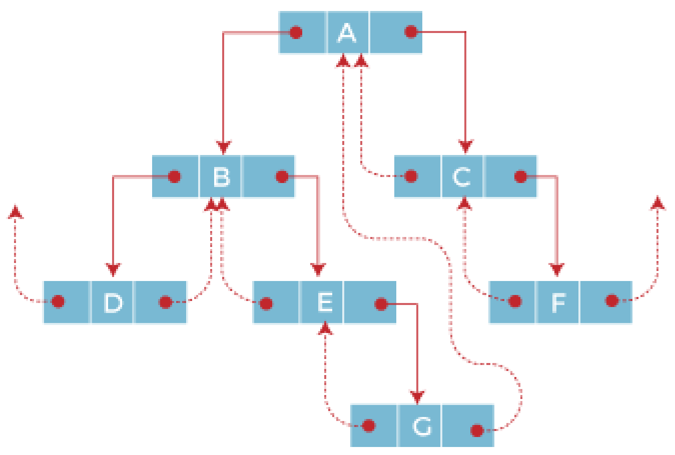


Double Threaded Binary Tree

In two-way threaded Binary trees, the right link field of a node containing NULL values is replaced by a thread that points to nodes inorder successor and left field of a node containing NULL values is replaced by a thread that points to nodes inorder predecessor.

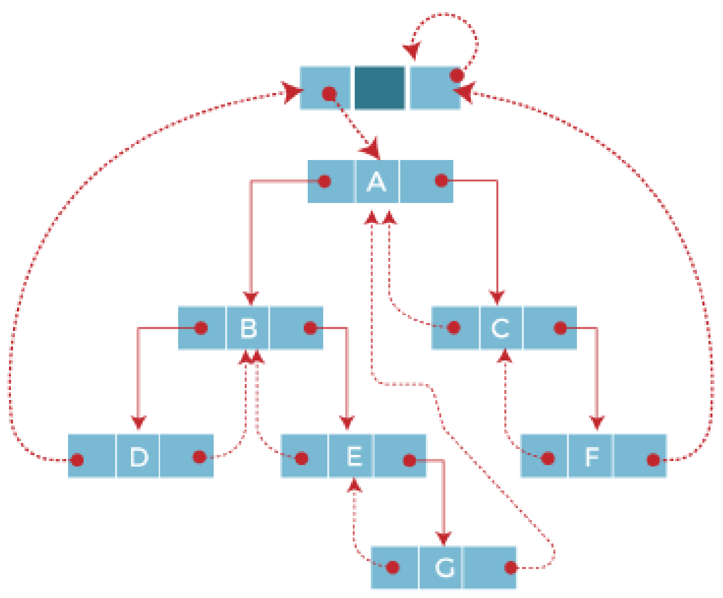


A binary tree (Inorder traversal - D, B, E, G, A, C, F)



A two - way threaded binary tree

The above figure shows the inorder traversal of this binary tree yields D, B, E, G, A, C, F. If we consider the two-way threaded Binary tree, the node E whose left field contains NULL is replaced by a thread pointing to its inorder predecessor i.e. node B. Similarly, for node G whose right and left linked fields contain NULL values are replaced by threads such that right link field points to its inorder successor and left link field points to its inorder predecessor. In the same way, other nodes containing NULL values in their link fields are filled with threads.



Two-way threaded - tree with header node

In the above figure of two-way threaded Binary tree, we noticed that no left thread is possible for the first node and no right thread is possible for the last node. This is because they don't have any inorder predecessor and successor respectively. This is indicated by threads pointing nowhere. So in order to maintain the uniformity of threads, we maintain a special node called the **header node**. The header node does not contain any data part and its left link field points to the root node and its right link field points to itself. If this header node is included in the two-way threaded Binary tree then this node becomes the inorder predecessor of the first node and inorder successor of the last node. Now threads of left link fields of the first node and right link fields of the last node will point to the header node.

Advantages of Threaded Binary Tree:

- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

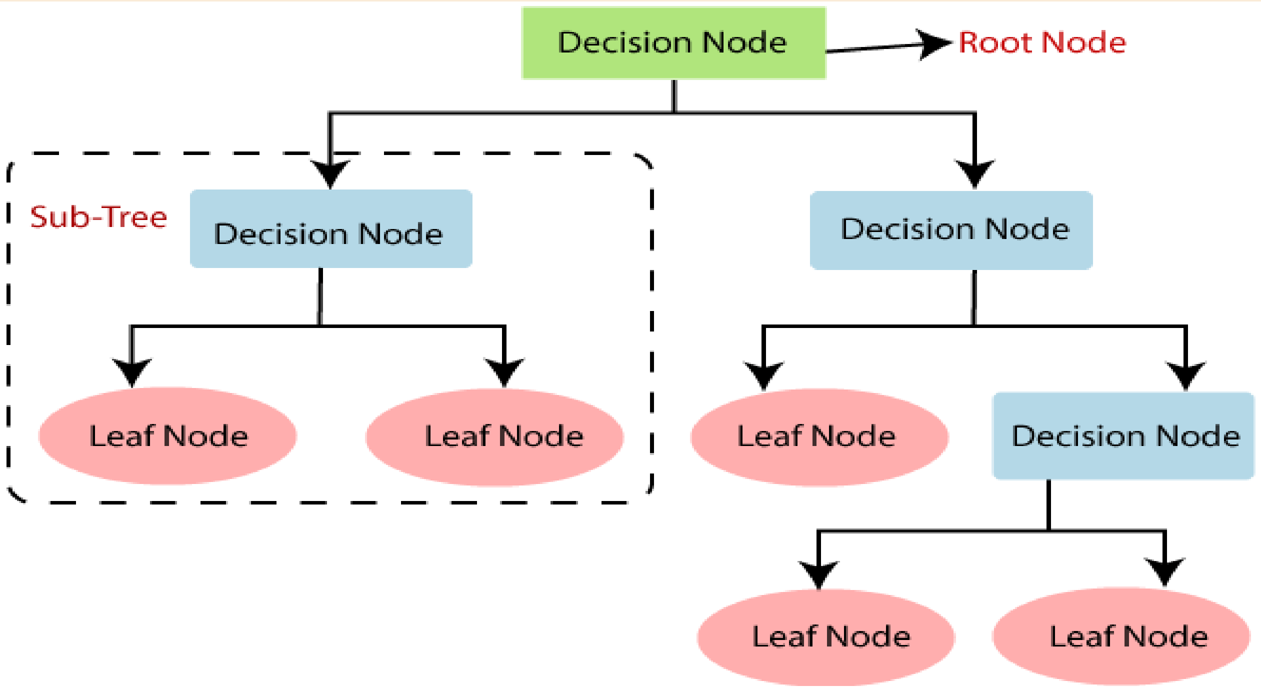
Disadvantages of Threaded Binary Tree:

- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

Decision Tree

- Decision Tree is a **Supervised learning technique** that can be used for both classification and Regression problems, but mostly it is preferred for solving Classification problems. It is a tree-structured classifier, where **internal nodes represent the features of a dataset, branches represent the decision rules** and **each leaf node represents the outcome.**
- In a Decision tree, there are two nodes, which are the **Decision Node** and **Leaf Node.** Decision nodes are used to make any decision and have multiple branches, whereas Leaf nodes are the output of those decisions and do not contain any further branches.
- The decisions or the test are performed on the basis of features of the given dataset.
- ***It is a graphical representation for getting all the possible solutions to a problem/decision based on given conditions.***
- It is called a decision tree because, similar to a tree, it starts with the root node, which expands on further branches and constructs a tree-like structure.
- In order to build a tree, we use the **CART algorithm,** which stands for **Classification and Regression Tree algorithm.**
- A decision tree simply asks a question, and based on the answer (Yes/No), it further split the tree into subtrees.
- Below diagram explains the general structure of a decision tree:

Note: A decision tree can contain categorical data (YES/NO) as well as numeric data.



Why use Decision Trees?

There are various algorithms in Machine learning, so choosing the best algorithm for the given dataset and problem is the main point to remember while creating a machine learning model. Below are the two reasons for using the Decision tree:

- Decision Trees usually mimic human thinking ability while making a decision, so it is easy to understand.
- The logic behind the decision tree can be easily understood because it shows a tree-like structure.

Decision Tree Terminologies

 Root Node: Root node is from where the decision tree starts. It represents the entire dataset, which further gets divided into two or more homogeneous sets.

 Leaf Node: Leaf nodes are the final output node, and the tree cannot be segregated further after getting a leaf node.

 Splitting: Splitting is the process of dividing the decision node/root node into sub-nodes according to the given conditions.

 Branch/Sub Tree: A tree formed by splitting the tree.

 Pruning: Pruning is the process of removing the unwanted branches from the tree.

 Parent/Child node: The root node of the tree is called the parent node, and other nodes are called the child nodes.

How does the Decision Tree algorithm Work?

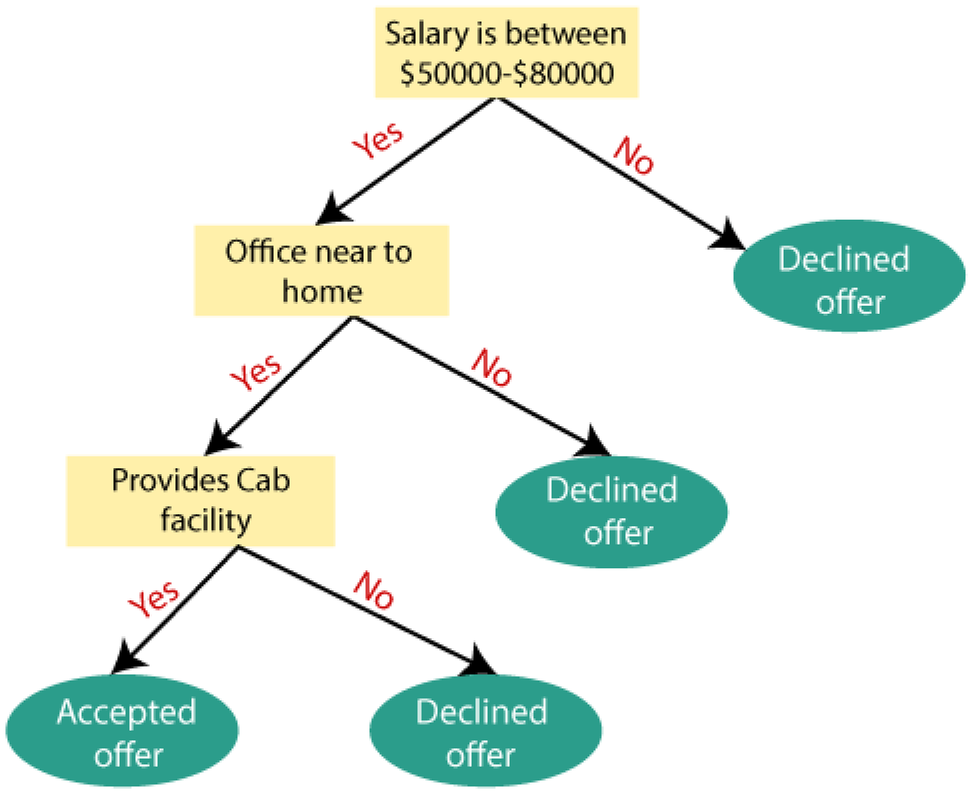
In a decision tree, for predicting the class of the given dataset, the algorithm starts from the root node of the tree. This algorithm compares the values of root attribute with the record (real dataset) attribute and, based on the comparison, follows the branch and jumps to the next node.

For the next node, the algorithm again compares the attribute value with the other sub-nodes and move further. It continues the process until it reaches the leaf node of the tree. The complete process can be better understood using the below algorithm:

- **Step-1:** Begin the tree with the root node, says S, which contains the complete dataset.

- **Step-2:** Find the best attribute in the dataset using **Attribute Selection Measure (ASM).**
- **Step-3:** Divide the S into subsets that contains possible values for the best attributes.
- **Step-4:** Generate the decision tree node, which contains the best attribute.
- **Step-5:** Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you cannot further classify the nodes and called the final node as a leaf node.

Example: Suppose there is a candidate who has a job offer and wants to decide whether he should accept the offer or Not. So, to solve this problem, the decision tree starts with the root node (Salary attribute by ASM). The root node splits further into the next decision node (distance from the office) and one leaf node based on the corresponding labels. The next decision node further gets split into one decision node (Cab facility) and one leaf node. Finally, the decision node splits into two leaf nodes (Accepted offers and Declined offer). Consider the below diagram:



Attribute Selection Measures

While implementing a Decision tree, the main issue arises that how to select the best attribute for the root node and for sub-nodes. So, to solve such problems there is a technique which is called as **Attribute selection measure or ASM.** By this measurement, we can easily select the best attribute for the nodes of the tree. There are two popular techniques for ASM, which are:

- **Information Gain**
- **Gini Index**

1. Information Gain:

- Information gain is the measurement of changes in entropy after the segmentation of a dataset based on an attribute.
- It calculates how much information a feature provides us about a class.
- According to the value of information gain, we split the node and build the decision tree.
- A decision tree algorithm always tries to maximize the value of information gain, and a node/attribute having the highest information gain is split first. It can be calculated using the below formula:

1. Information Gain= Entropy(S)- [(Weighted Avg) *Entropy(each feature)

Entropy: Entropy is a metric to measure the impurity in a given attribute. It specifies randomness in data. Entropy can be calculated as:

Entropy(s)= -P(yes)log2 P(yes)- P(no) log2 P(no)

Where,

- **S= Total number of samples**
- **P(yes)= probability of yes**
- **P(no)= probability of no**

2. Gini Index:

- Gini index is a measure of impurity or purity used while creating a decision tree in the CART(Classification and Regression Tree) algorithm.
- An attribute with the low Gini index should be preferred as compared to the high Gini index.
- It only creates binary splits, and the CART algorithm uses the Gini index to create binary splits.
- Gini index can be calculated using the below formula:

Gini Index= $1- \sum_j P_j^2$

Pruning: Getting an Optimal Decision tree

Pruning is a process of deleting the unnecessary nodes from a tree in order to get the optimal decision tree.

A too-large tree increases the risk of overfitting, and a small tree may not capture all the important features of the dataset. Therefore, a technique that decreases the size of the learning tree without reducing accuracy is known as Pruning. There are mainly two types of tree **pruning** technology used:

- **Cost Complexity Pruning**
- **Reduced Error Pruning.**

Advantages of the Decision Tree

- It is simple to understand as it follows the same process which a human follow while making any decision in real-life.
- It can be very useful for solving decision-related problems.
- It helps to think about all the possible outcomes for a problem.
- There is less requirement of data cleaning compared to other algorithms.

Disadvantages of the Decision Tree

- The decision tree contains lots of layers, which makes it complex.
- It may have an overfitting issue, which can be resolved using the **Random Forest algorithm.**
- For more class labels, the computational complexity of the decision tree may increase.

Expression Tree

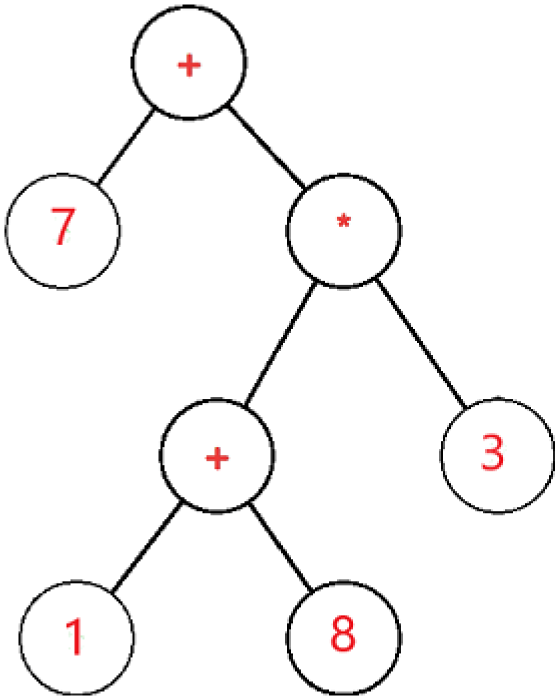
The expression tree is a tree used to represent the various expressions. The tree data structure is used to represent the expressional statements. In this tree, the internal node always denotes the operators.

- The leaf nodes always denote the operands.
- The operations are always performed on these operands.
- The operator present in the depth of the tree is always at the highest priority.
- The operator, which is not much at the depth in the tree, is always at the lowest priority compared to the operators lying at the depth.

- The operand will always present at a depth of the tree; hence it is considered the **highest priority** among all the operators.
- In short, we can summarize it as the value present at a depth of the tree is at the highest priority compared with the other operators present at the top of the tree.
- The main use of these expression trees is that it is used to **evaluate, analyze** and **modify** the various expressions.
- It is also used to find out the associativity of each operator in the expression.
- For example, the + operator is the left-associative and / is the right-associative.
- The dilemma of this associativity has been cleared by using the expression trees.
- These expression trees are formed by using a context-free grammar.
- We have associated a rule in context-free grammars in front of each grammar production.
- These rules are also known as semantic rules, and by using these semantic rules, we can be easily able to construct the expression trees.
- It is one of the major parts of compiler design and belongs to the semantic analysis phase.
- In this semantic analysis, we will use the syntax-directed translations, and in the form of output, we will produce the annotated parse tree.
- An annotated parse tree is nothing but the simple parse associated with the type attribute and each production rule.
- The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees.
- It is immutable, and once we have created an expression tree, we can not change it or modify it further.
- To make more modifications, it is required to construct the new expression tree wholly.
- It is also used to solve the postfix, prefix, and infix expression evaluation.

Expression trees play a very important role in representing the **language-level** code in the form of the data, which is mainly stored in the tree-like structure. It is also used in the memory representation of the **lambda** expression. Using the tree data structure, we can express the lambda expression more transparently and explicitly. It is first created to convert the code segment onto the data segment so that the expression can easily be evaluated.

The expression tree is a binary tree in which each external or leaf node corresponds to the operand and each internal or parent node corresponds to the operators so for example expression tree for 7 + ((1+8)*3) would be:



Let S be the expression tree

```
If S is not null, then

If S.value is an operand, then

Return S.value

x = solve(S.left)

y = solve(S.right)

Return calculate(x, y, S.value)
```

Here in the above example, the expression tree used context-free grammar.

We have some productions associated with some production rules in this grammar, mainly known as **semantic rules**. We can define the result-producing from the corresponding production rules using these semantic rules. Here we have used the value parameter, which will calculate the result and return it to the grammar's start symbol. S.left will calculate the left child of the node, and similarly, the right child of the node can be calculated using the S.right parameter.

Use of Expression tree

1. The main objective of using the expression trees is to make complex expressions and can be easily be evaluated using these expression trees.
2. It is also used to find out the associativity of each operator in the expression.
3. It is also used to solve the postfix, prefix, and infix expression evaluation.

Implementation of an Expression tree

To implement the expression tree and write its program, we will be required to use a stack data structure.

As we know that the stack is based on the last in first out LIFO principle, the data element pushed recently into the stack has been popped out whenever required. For its implementation, the main two operations of the stack, push and pop, are used. Using the push operation, we will push the data element into the stack, and by using the pop operation, we will remove the data element from the stack.

Main functions of the stack in the expression tree implementation:

First of all, we will do scanning of the given expression into left to the right manner, then one by one check the identified character,

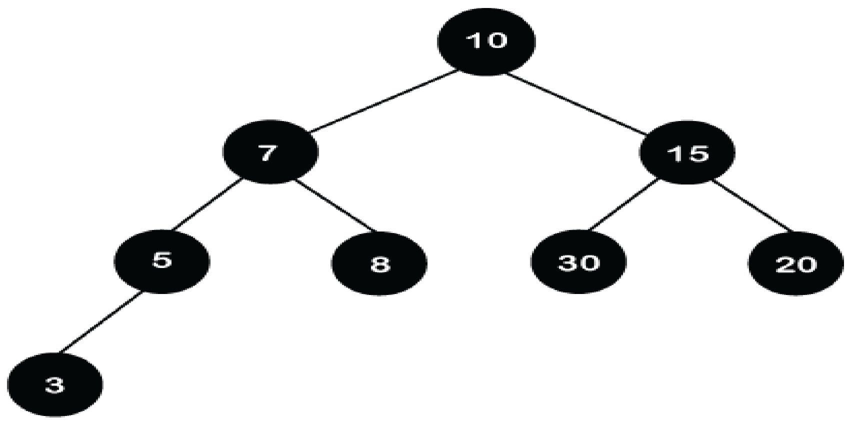
1. If a scanned character is an operand, we will apply the push operation and push it into the stack.
2. If a scanned character is an operator, we will apply the pop operation into it to remove the two values from the stack to make them its child, and after then we will push back the current parent node into the stack.

Red-black tree in Data Structure

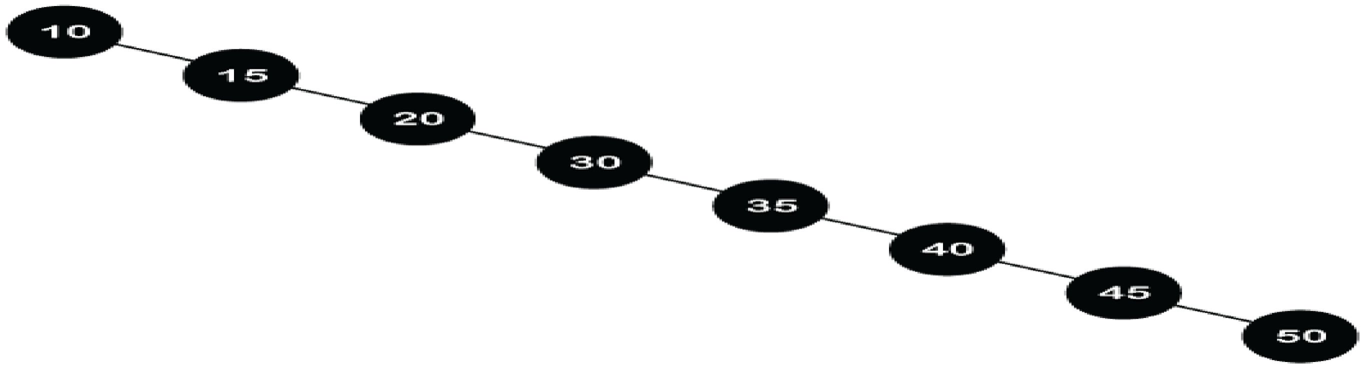
The red-Black tree is a binary search tree. The prerequisite of the red-black tree is that we should know about the binary search tree. In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

Each node in the Red-black tree contains an extra bit that represents a color to ensure that the tree is balanced during any operations performed on the tree like insertion, deletion, etc. In a binary search tree, the searching, insertion and deletion take **O(log2n)** time in the average case, **O(1)** in the best case and **O(n)** in the worst case.

Let's understand the different scenarios of a binary search tree.



In the above tree, if we want to search the 80. We will first compare 80 with the root node. 80 is greater than the root node, i.e., 10, so searching will be performed on the right subtree. Again, 80 is compared with 15; 80 is greater than 15, so we move to the right of the 15, i.e., 20. Now, we reach the leaf node 20, and 20 is not equal to 80. Therefore, it will show that the element is not found in the tree. After each operation, the search is divided into half. The above BST will take O(logn) time to search the element.



The above tree shows the right-skewed BST. If we want to search the 80 in the tree, we will compare 80 with all the nodes until we find the element or reach the leaf node. So, the above right-skewed BST will take **O(N)** time to search the element.

In the above BST, the first one is the balanced BST, whereas the second one is the unbalanced BST. We conclude from the above two binary search trees that a balanced tree takes less time than an unbalanced tree for performing any operation on the tree.

Therefore, we need a balanced tree, and the Red-Black tree is a self-balanced binary search tree. Now, the question arises that ***why do we require a Red-Black tree*** if AVL is also a height-balanced tree. The Red-Black tree is used because the AVL tree requires many rotations when the tree is large, whereas the Red-Black tree requires a maximum of two rotations to balance the tree. The main difference between the AVL tree and the Red-Black tree is that the AVL tree is strictly balanced, while the Red-Black tree is not completely height-balanced. So, the AVL tree is more balanced than the Red-Black tree, but the Red-Black tree guarantees O(log2n) time for all operations like insertion, deletion, and searching.

Insertion is easier in the AVL tree as the AVL tree is strictly balanced, whereas deletion and searching are easier in the Red-Black tree as the Red-Black tree requires fewer rotations.

As the name suggests that the node is either colored in **Red** or **Black** color. Sometimes no rotation is required, and only recoloring is needed to balance the tree.

Properties of Red-Black tree

- It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.
- This tree data structure is named as a Red-Black tree as each node is either Red or Black in color. Every node stores one extra information known as a bit that represents the color of the node. For example, 0 bit denotes the black color while 1 bit denotes the red color of the node. Other information stored by the node is similar to the binary tree, i.e., data part, left pointer and right pointer.
- In the Red-Black tree, the root node is always black in color.
- In a binary tree, we consider those nodes as the leaf which have no child. In contrast, in the Red-Black tree, the nodes that have no child are considered the internal nodes and these nodes are connected to the NIL nodes that are always black in color. The NIL nodes are the leaf nodes in the Red-Black tree.
- If the node is Red, then its children should be in Black color. In other words, we can say that there should be no red-red parent-child relationship.
- Every path from a node to any of its descendant's NIL node should have same number of black nodes.

Is every AVL tree can be a Red-Black tree?

Yes, every AVL tree can be a Red-Black tree if we color each node either by Red or Black color. But every Red-Black tree is not an AVL because the AVL tree is strictly height-balanced while the Red-Black tree is not completely height-balanced.

Insertion in Red Black tree

The following are some rules used to create the Red-Black tree:

1. If the tree is empty, then we create a new node as a root node with the color black.
2. If the tree is not empty, then we create a new node as a leaf node with a color red.
3. If the parent of a new node is black, then exit.
4. If the parent of a new node is Red, then we have to check the color of the parent's sibling of a new node.

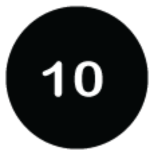
4a) If the color is Black, then we perform rotations and recoloring.

4b) If the color is Red then we recolor the node. We will also check whether the parents' parent of a new node is the root node or not; if it is not a root node, we will recolor and recheck the node.

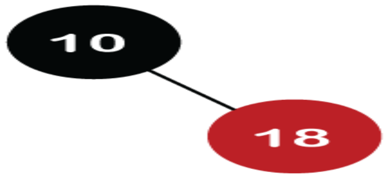
Let's understand the insertion in the Red-Black tree.

10, 18, 7, 15, 16, 30, 25, 40, 60

Step 1: Initially, the tree is empty, so we create a new node having value 10. This is the first node of the tree, so it would be the root node of the tree. As we already discussed, that root node must be black in color, which is shown below:



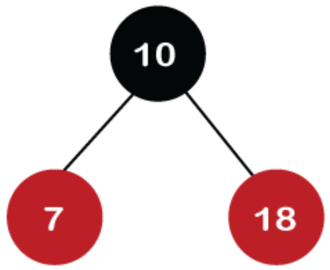
Step 2: The next node is 18. As 18 is greater than 10 so it will come at the right of 10 as shown below.



We know the second rule of the Red Black tree that if the tree is not empty then the newly created node will have the **Red** color. Therefore, node 18 has a Red color, as shown in the below figure:

Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. In the above figure, the parent of the node is black in color; therefore, it is a Red-Black tree.

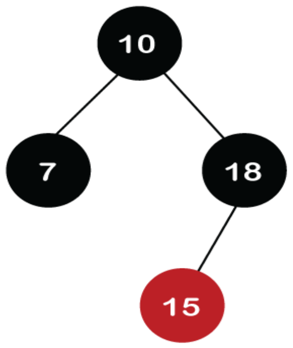
Step 3: Now, we create the new node having value 7 with Red color. As 7 is less than 10, so it will come at the left of 10 as shown below.



Now we verify the third rule of the Red-Black tree, i.e., the parent of the new node is black or not. As we can observe, the parent of the node 7 is black in color, and it obeys the Red-Black tree's properties.

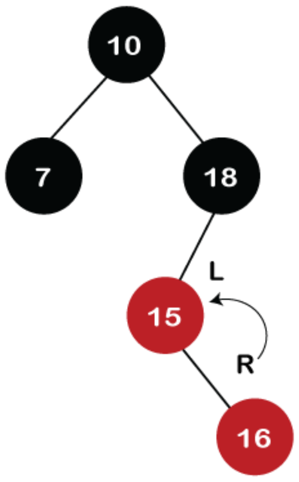
Step 4: The next element is 15, and 15 is greater than 10, but less than 18, so the new node will be created at the left of node 18. The node 15 would be Red in color as the tree is not empty.

The above tree violates the property of the Red-Black tree as it has Red-red parent-child relationship. Now we have to apply some rule to make a Red-Black tree. The rule 4 says that ***if the new node's parent is Red, then we have to check the color of the parent's sibling of a new node.*** The new node is node 15; the parent of the new node is node 18 and the sibling of the parent node is node 7. As the color of the parent's sibling is Red in color, so we apply the rule 4a. The rule 4a says that we have to recolor both the parent and parent's sibling node. So, both the nodes, i.e., 7 and 18, would be recolored as shown in the below figure.



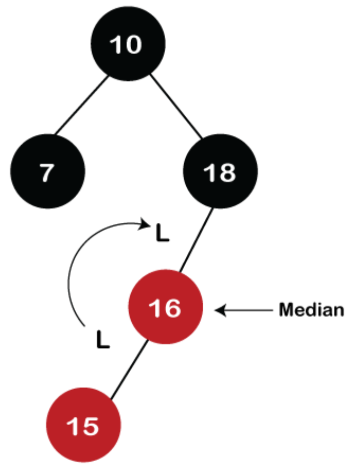
We also have to check whether the parent's parent of the new node is the root node or not. As we can observe in the above figure, the parent's parent of a new node is the root node, so we do not need to recolor it.

Step 5: The next element is 16. As 16 is greater than 10 but less than 18 and greater than 15, so node 16 will come at the right of node 15. The tree is not empty; node 16 would be Red in color, as shown in the below figure:

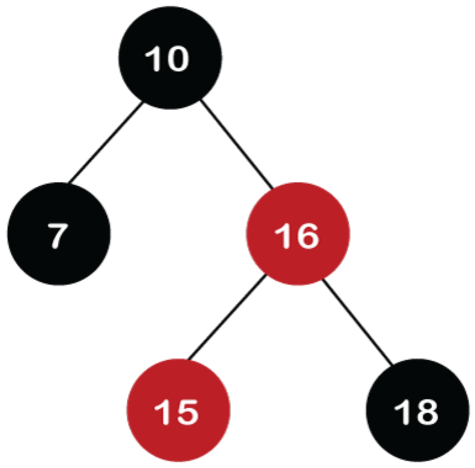


In the above figure, we can observe that it violates the property of the parent-child relationship as it has a red-red parent-child relationship. We have to apply some rules to make a Red-Black tree. Since the new node's parent is Red color, and the parent of the new node has no sibling, so rule **4a** will be applied. The rule **4a** says that some rotations and recoloring would be performed on the tree.

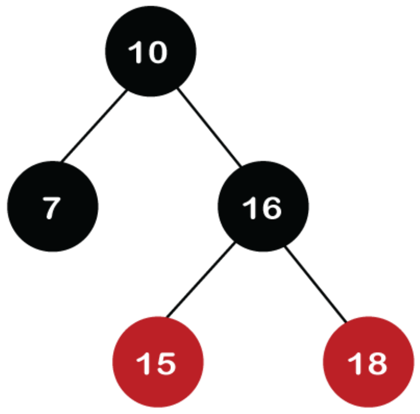
Since node 16 is right of node 15 and the parent of node 15 is node 18. Node 15 is the left of node 18. Here we have **an LR** relationship, so we require to perform two rotations. First, we will perform left, and then we will perform the right rotation. The left rotation would be performed on nodes 15 and 16, where node 16 will move upward, and node 15 will move downward. Once the left rotation is performed, the tree looks like as shown in the below figure:



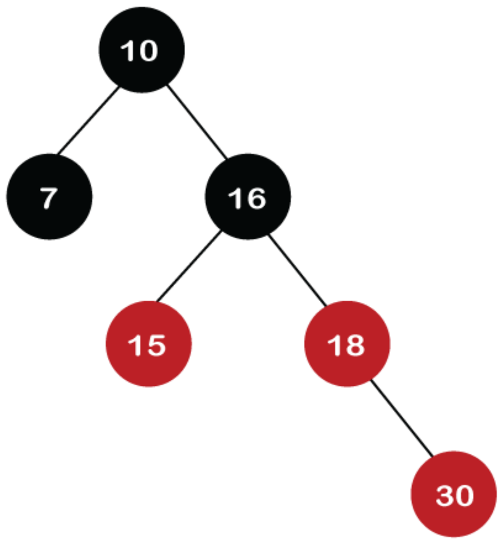
In the above figure, we can observe that there is **an LL** relationship. The above tree has a Red-red conflict, so we perform the right rotation. When we perform the right rotation, the median element would be the root node. Once the right rotation is performed, node 16 would become the root node, and nodes 15 and 18 would be the left child and right child, respectively, as shown in the below figure.



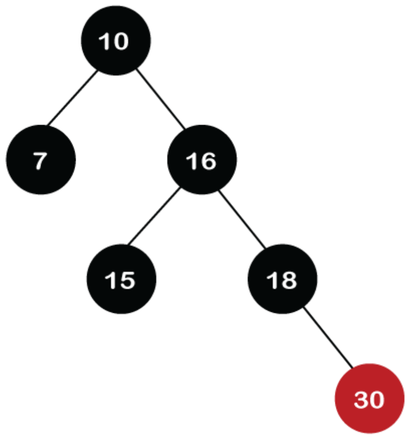
After rotation, node 16 and node 18 would be recolored; the color of node 16 is red, so it will change to black, and the color of node 18 is black, so it will change to a red color as shown in the below figure:



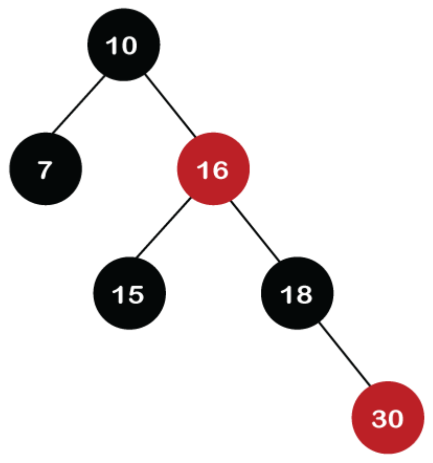
Step 6: The next element is 30. Node 30 is inserted at the right of node 18. As the tree is not empty, so the color of node 30 would be red.



The color of the parent and parent's sibling of a new node is Red, so rule 4b is applied. In rule 4b, we have to do only recoloring, i.e., no rotations are required. The color of both the parent (node 18) and parent's sibling (node 15) would become black, as shown in the below image.

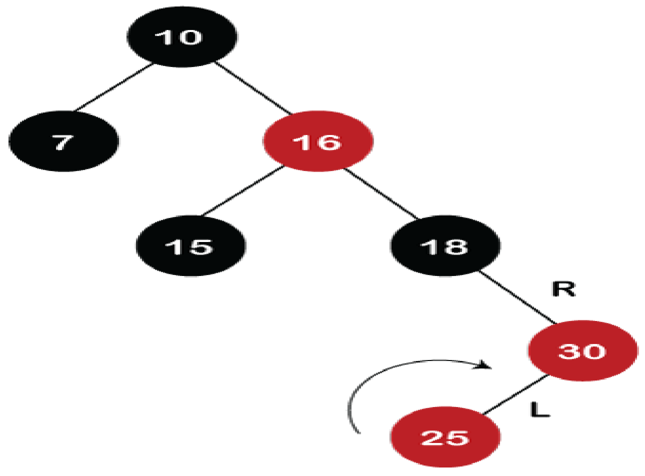


We also have to check the parent's parent of the new node, whether it is a root node or not. The parent's parent of the new node, i.e., node 30 is node 16 and node 16 is not a root node, so we will recolor the node 16 and changes to the Red color. The parent of node 16 is node 10, and it is not in Red color, so there is no Red-red conflict.

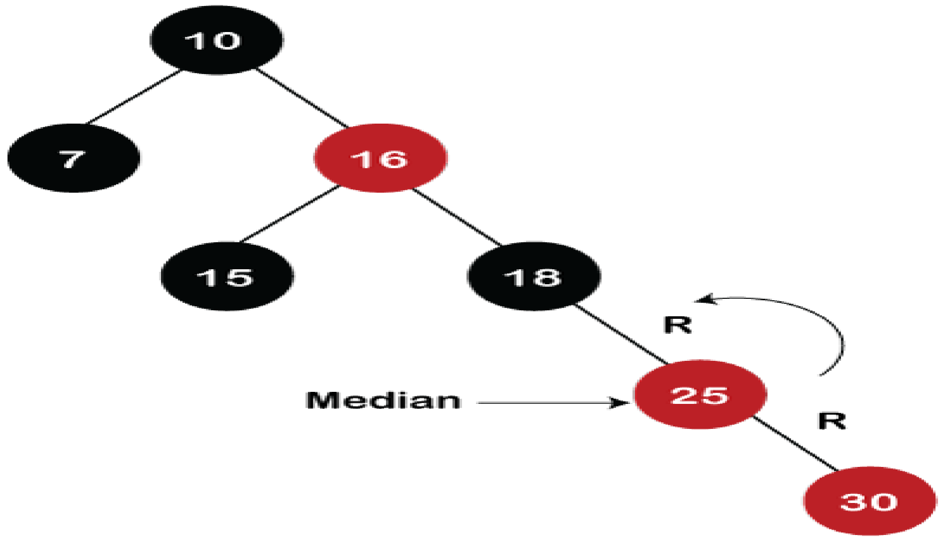


Step 7: The next element is 25, which we have to insert in a tree. Since 25 is greater than 10, 16, 18 but less than 30; so, it will come at the left of node 30. As the tree is not empty, node 25 would be in Red color. Here Red-red conflict occurs as the parent of the newly created is Red color.

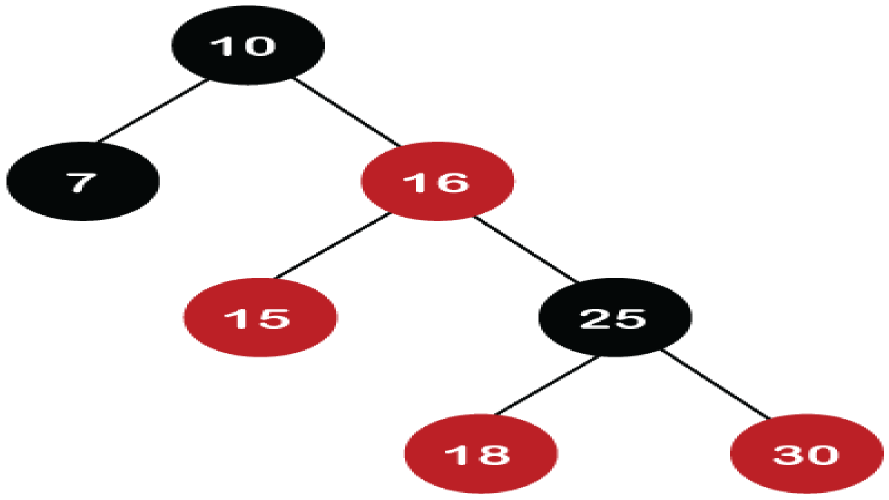
Since there is no parent's sibling, so rule 4a is applied in which rotation, as well as recoloring, are performed. First, we will perform rotations. As the newly created node is at the left of its parent and the parent node is at the right of its parent, so the RL relationship is formed. Firstly, the right rotation is performed in which node 25 goes upwards, whereas node 30 goes downwards, as shown in the below figure.



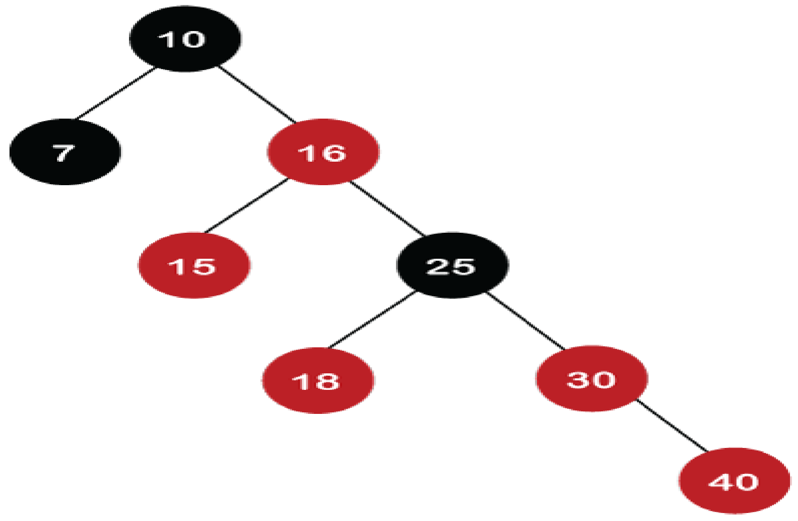
After the first rotation, there is an RR relationship, so left rotation is performed. After right rotation, the median element, i.e., 25 would be the root node; node 30 would be at the right of 25 and node 18 would be at the left of node 25.



Now recoloring would be performed on nodes 25 and 18; node 25 becomes black in color, and node 18 becomes red in color.



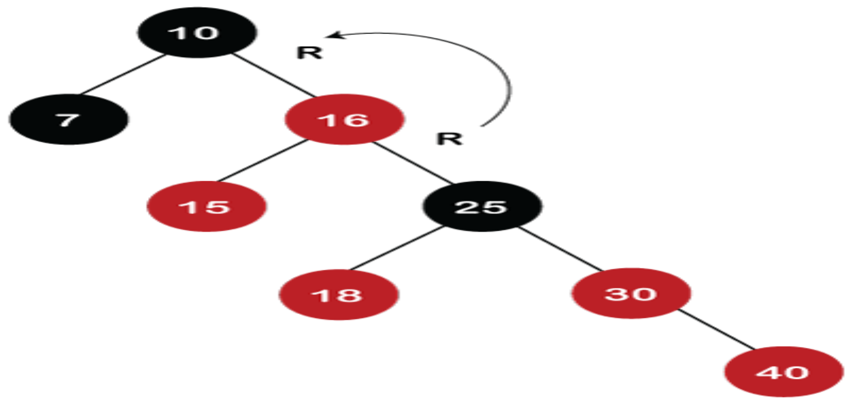
Step 8: The next element is 40. Since 40 is greater than 10, 16, 18, 25, and 30, so node 40 will come at the right of node 30. As the tree is not empty, node 40 would be Red in color. There is a Red-red conflict between nodes 40 and 30, so rule 4b will be applied.



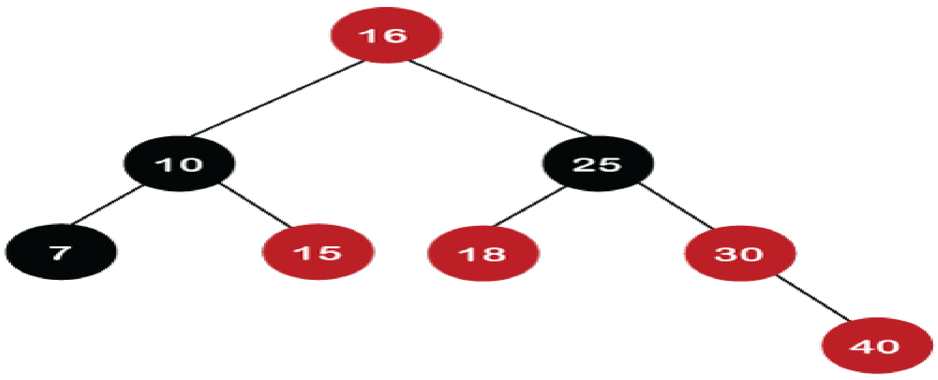
As the color of parent and parent's sibling node of a new node is Red so recoloring would be performed. The color of both the nodes would become black, as shown in the below image.

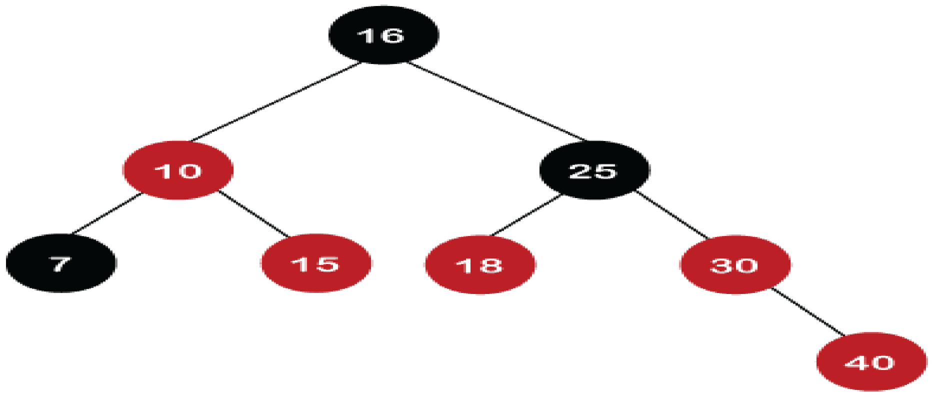
After recoloring, we also have to check the parent's parent of a new node, i.e., 25, which is not a root node, so recoloring would be performed, and the color of node 25 changes to Red.

After recoloring, red-red conflict occurs between nodes 25 and 16. Now node 25 would be considered as the new node. Since the parent of node 25 is red in color, and the parent's sibling is black in color, rule 4a would be applied. Since 25 is at the right of the node 16 and 16 is at the right of its parent, so there is an RR relationship. In the RR relationship, left rotation is performed. After left rotation, the median element 16 would be the root node, as shown in the below figure.



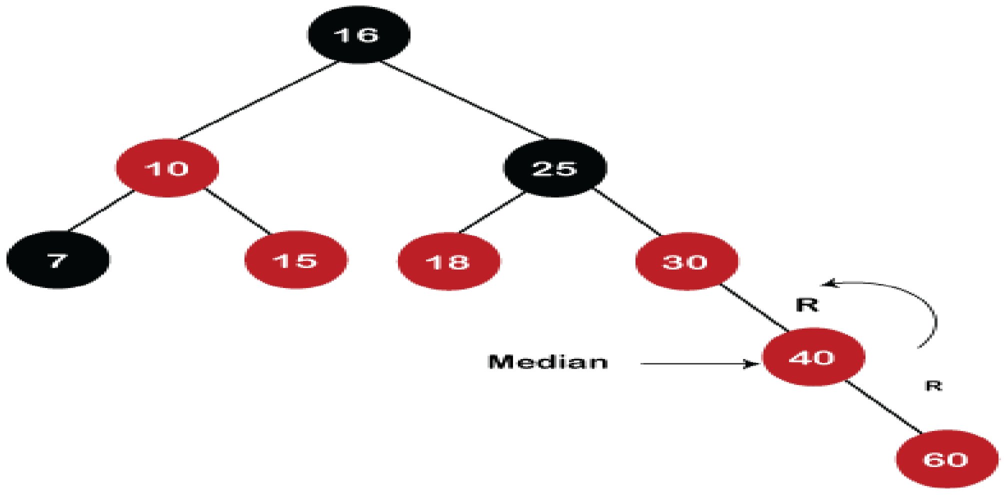
After rotation, recoloring is performed on nodes 16 and 10. The color of node 10 and node 16 changes to Red and Black, respectively as shown in the below figure.



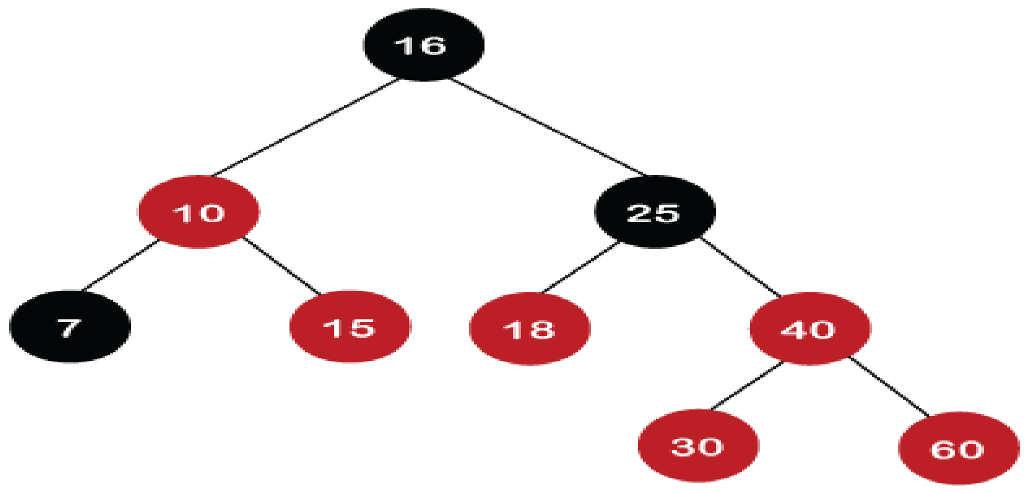


Step 9: The next element is 60. Since 60 is greater than 16, 25, 30, and 40, so node 60 will come at the right of node 40. As the tree is not empty, the color of node 60 would be Red.

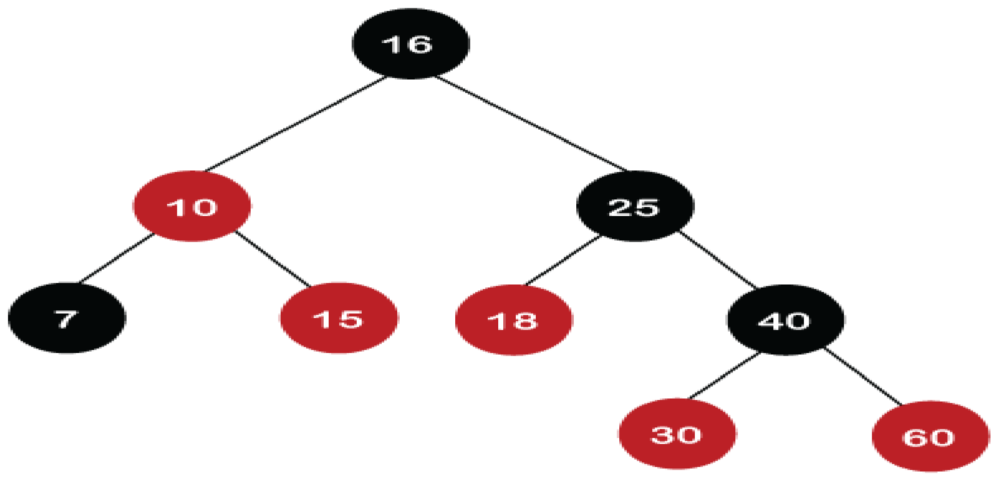
As we can observe in the above tree that there is a Red-red conflict occurs. The parent node is Red in color, and there is no parent's sibling exists in the tree, so rule 4a would be applied. The first rotation would be performed. The RR relationship exists between the nodes, so left rotation would be performed.



When left rotation is performed, node 40 will come upwards, and node 30 will come downwards, as shown in the below figure:



After rotation, the recoloring is performed on nodes 30 and 40. The color of node 30 would become Red, while the color of node 40 would become black.



The above tree is a Red-Black tree as it follows all the Red-Black tree properties.

Deletion in Red Back tree

Let's understand how we can delete the particular node from the Red-Black tree. The following are the rules used to delete the particular node from the tree:

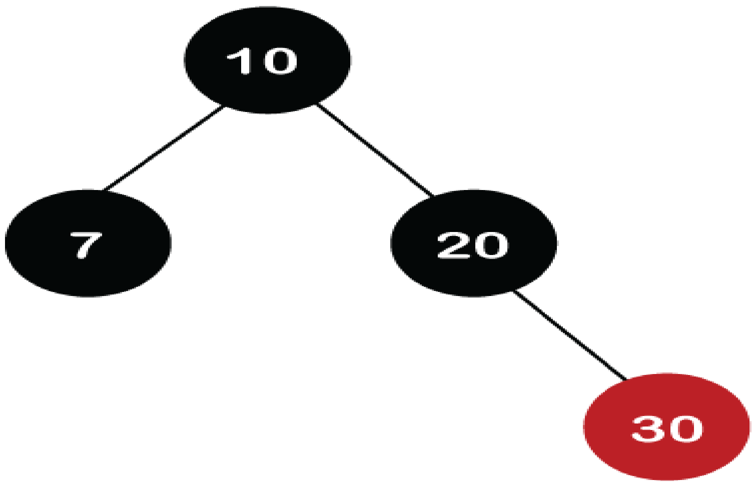
Step 1: First, we perform BST rules for the deletion.

Step 2:

Case 1: if the node is Red, which is to be deleted, we simply delete it.

Let's understand case 1 through an example.

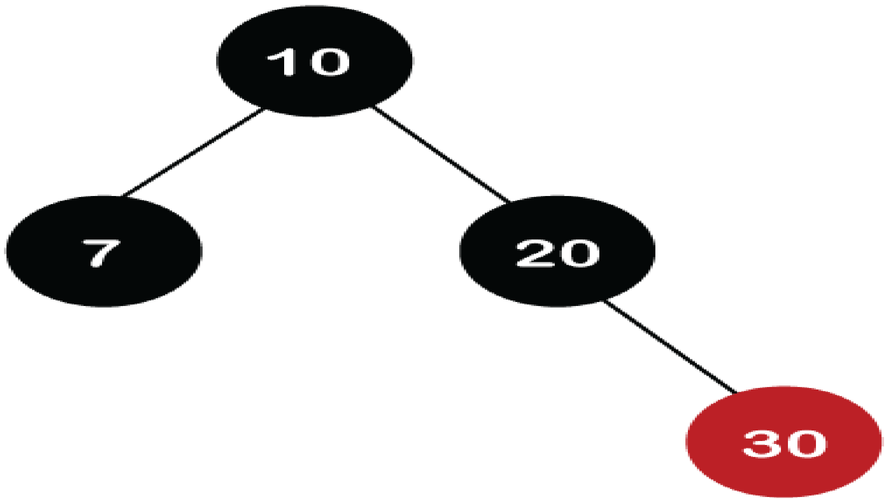
Suppose we want to delete node 30 from the tree, which is given below.



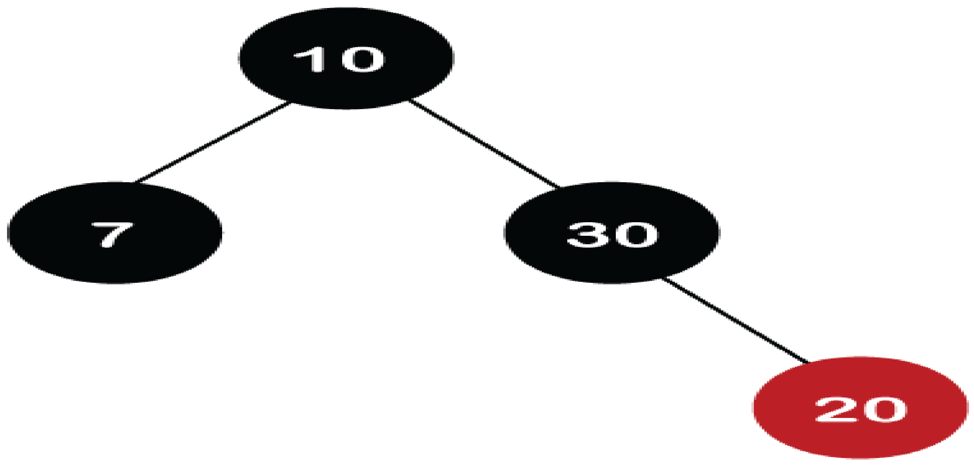
Initially, we are having the address of the root node. First, we will apply BST to search the node. Since 30 is greater than 10 and 20, which means that 30 is the right child of node 20. Node 30 is a leaf node and Red in color, so it is simply deleted from the tree.

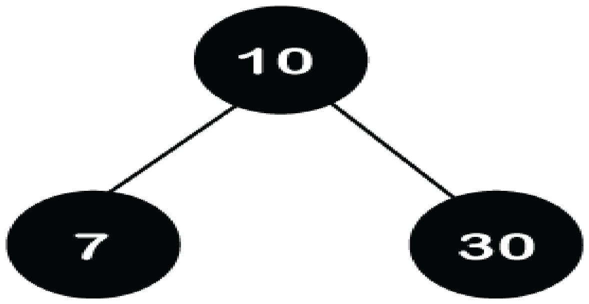
If we want to delete the internal node that has one child. First, replace the value of the internal node with the value of the child node and then simply delete the child node.

Let's take another example in which we want to delete the internal node, i.e., node 20.



We cannot delete the internal node; we can only replace the value of that node with another value. Node 20 is at the right of the root node, and it is having only one child, node 30. So, node 20 is replaced with a value 30, but the color of the node would remain the same, i.e., Black. In the end, node 20 (leaf node) is deleted from the tree.

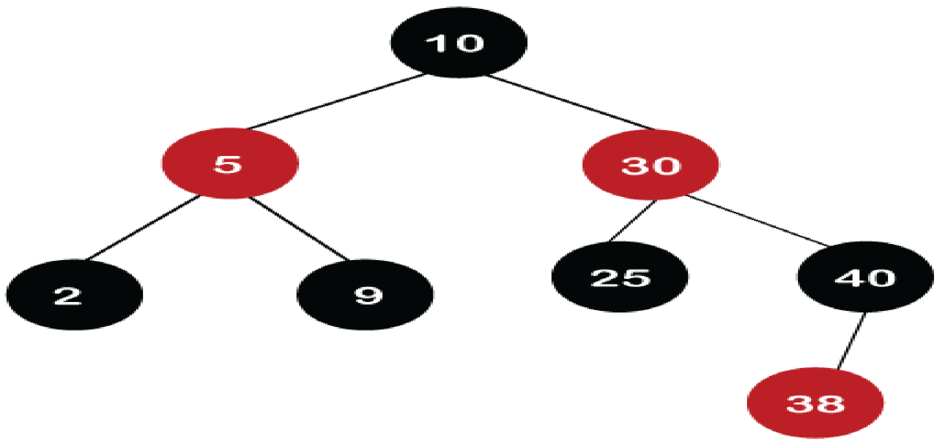




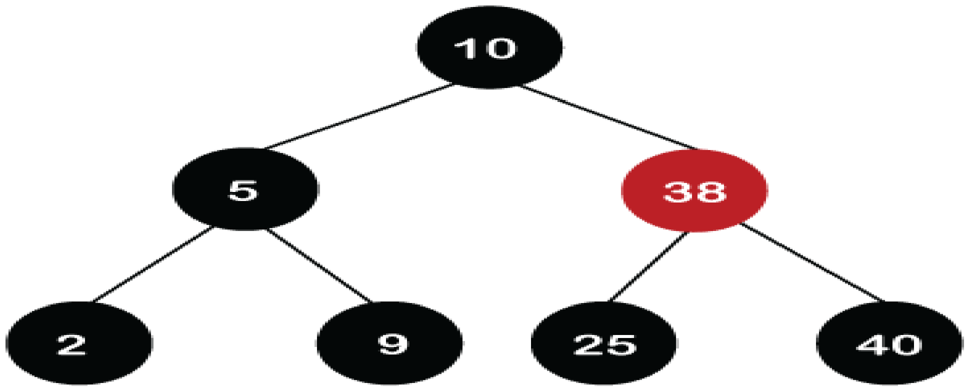
If we want to delete the internal node that has two child nodes. In this case, we have to decide from which we have to replace the value of the internal node (either left subtree or right subtree). We have two ways:

- **Inorder predecessor:** We will replace with the largest value that exists in the left subtree.
- **Inorder successor:** We will replace with the smallest value that exists in the right subtree.

Suppose we want to delete node 30 from the tree, which is shown below:



Node 30 is at the right of the root node. In this case, we will use **the inorder successor**. The value 38 is the smallest value in the right subtree, so we will replace the value 30 with 38, but the node would remain the same, i.e., Red. After replacement, the leaf node, i.e., 30, would be deleted from the tree. Since node 30 is a leaf node and Red in color, we need to delete it (we do not have to perform any rotations or any recoloring).



Case 2: If the root node is also double black, then simply remove the double black and make it a single black.

Case 3: If the double black's sibling is black and both its children are black.

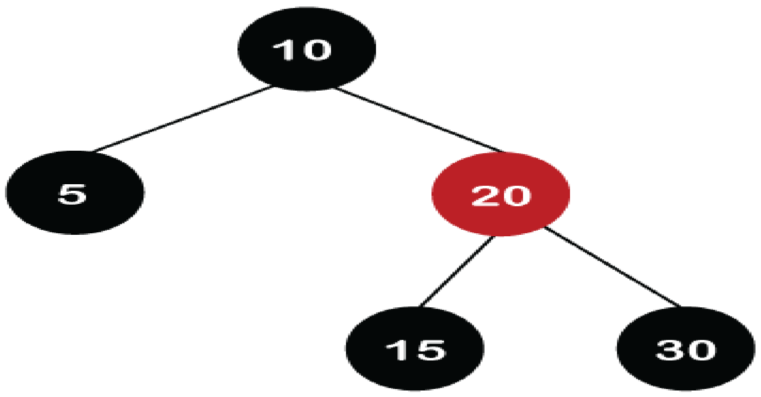
- Remove the double black node.
- Add the color of the node to the parent (P) node.

1. If the color of P is red then it becomes black.
2. If the color of P is black, then it becomes double black.

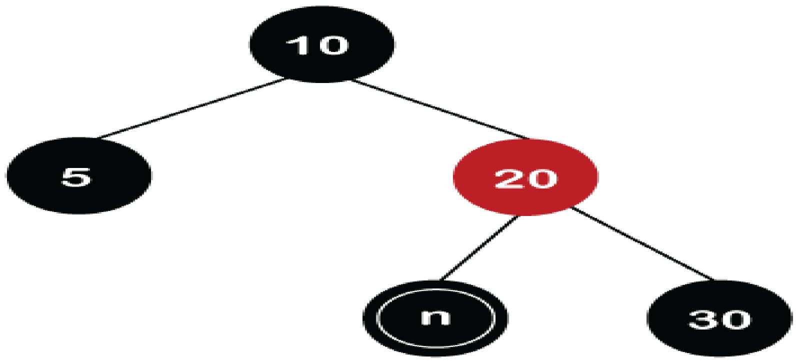
- The color of double black's sibling changes to red.
- If still double black situation arises, then we will apply other cases.

Let's understand this case through an example.

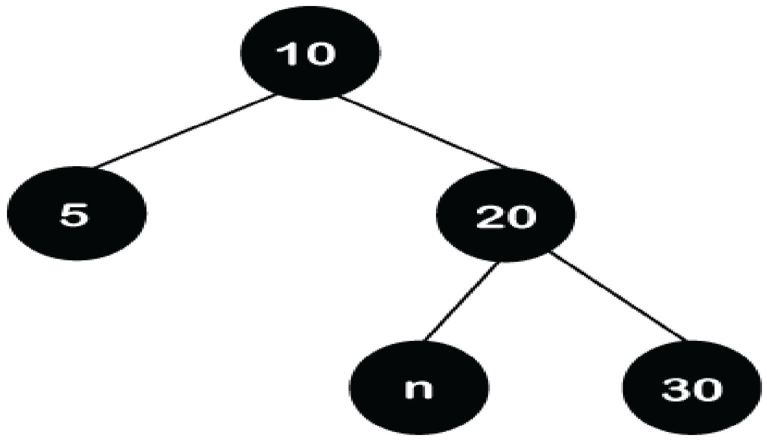
Suppose we want to delete node 15 in the below tree.



We cannot simply delete node 15 from the tree as node 15 is Black in color. Node 15 has two children, which are nil. So, we replace the 15 value with a nil value. As node 15 and nil node are black in color, the node becomes double black after replacement, as shown in the below figure.



In the above tree, we can observe that the double black's sibling is black in color and its children are nil, which are also black. As the double black's sibling and its children have black so it cannot give its black color to neither of these. Now, the double black's parent node is Red so double black's node add its black color to its parent node. The color of the node 20 changes to black while the color of the nil node changes to a single black as shown in the below figure.



After adding the color to its parent node, the color of the double black's sibling, i.e., node 30 changes to red as shown in the below figure.

In the above tree, we can observe that there is no longer double black's problem exists, and it is also a Red-Black tree.

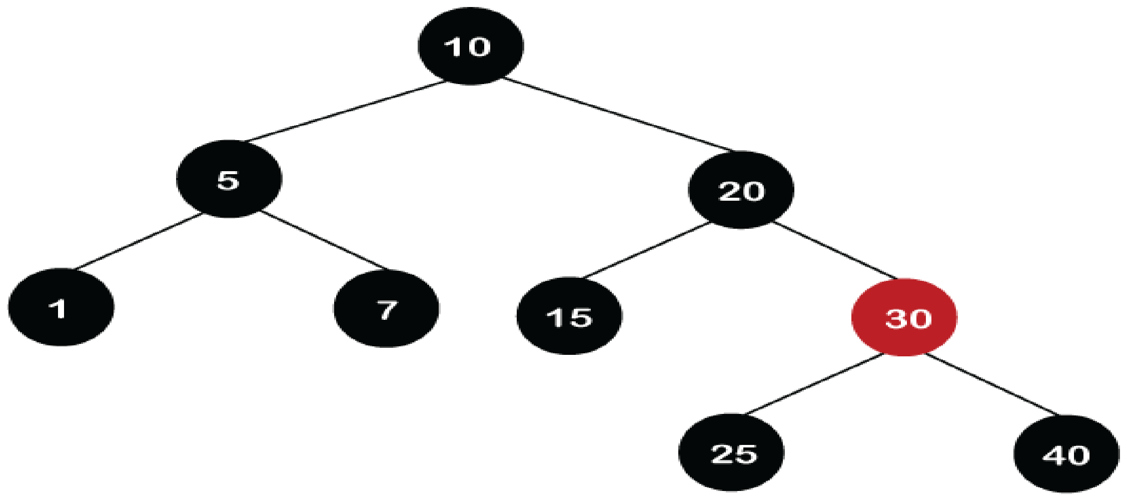
Case 4: If double black's sibling is Red.

- Swap the color of its parent and its sibling.
- Rotate the parent node in the double black's direction.

- Reapply cases.

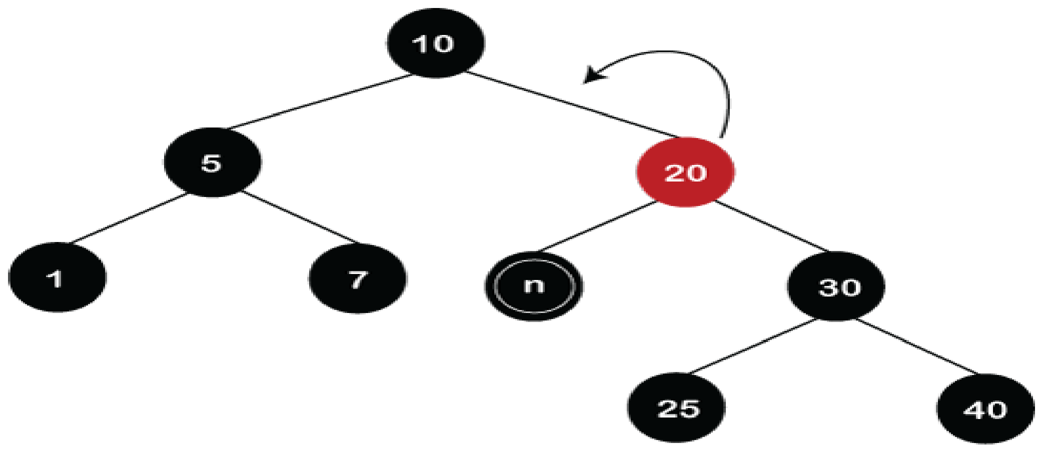
Let's understand this case through an example.

Suppose we want to delete node 15.

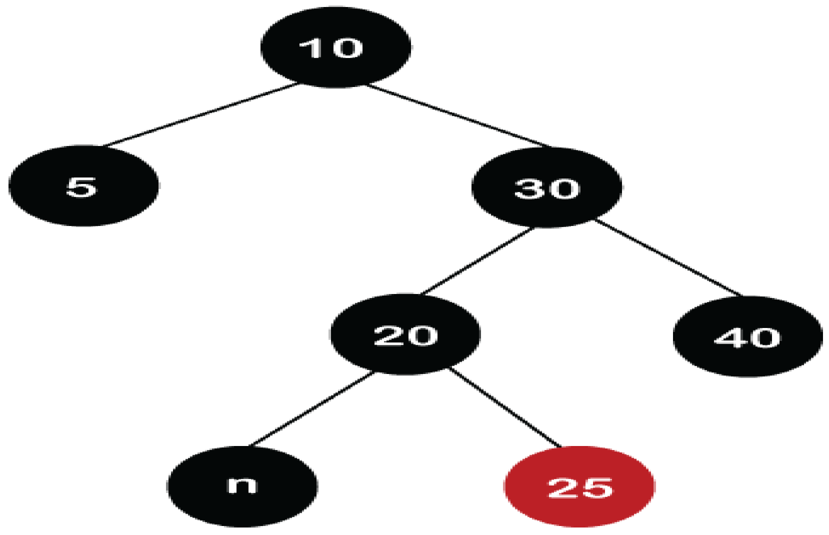


Initially, the 15 is replaced with a nil value. After replacement, the node becomes double black. Since double black's sibling is Red so color of the node 20 changes to Red and the color of the node 30 changes to Black.

Once the swapping of the color is completed, the rotation towards the double black would be performed. The node 30 will move upwards and the node 20 will move downwards as shown in the below figure.



In the above tree, we can observe that double black situation still exists in the tree. It satisfies the case 3 in which double black's sibling is black as well as both its children are black. First, we remove the double black from the node and add the black color to its parent node. At the end, the color of the double black's sibling, i.e., node 25 changes to Red as shown in the below figure.

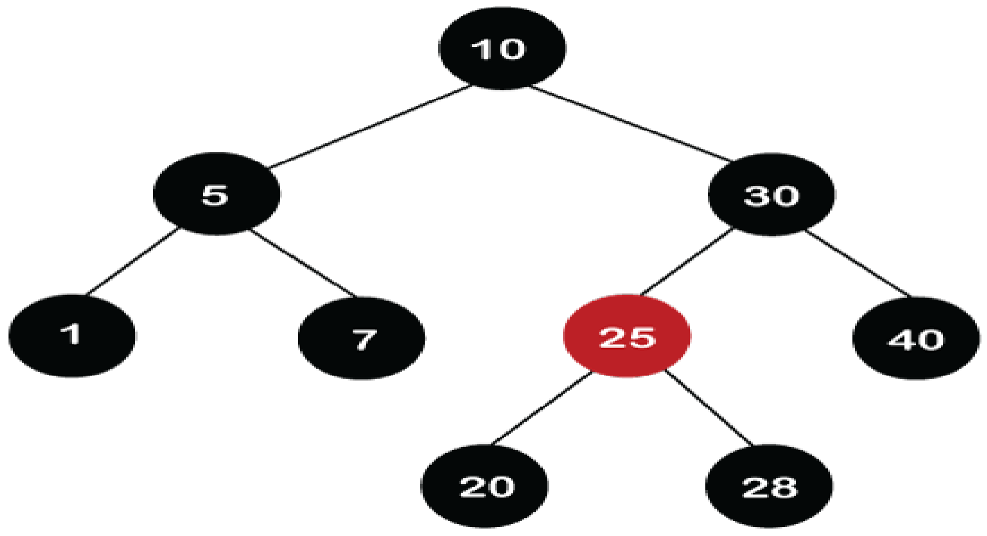


In the above tree, we can observe that the double black situation has been resolved. It also satisfies the properties of the Red Black tree.

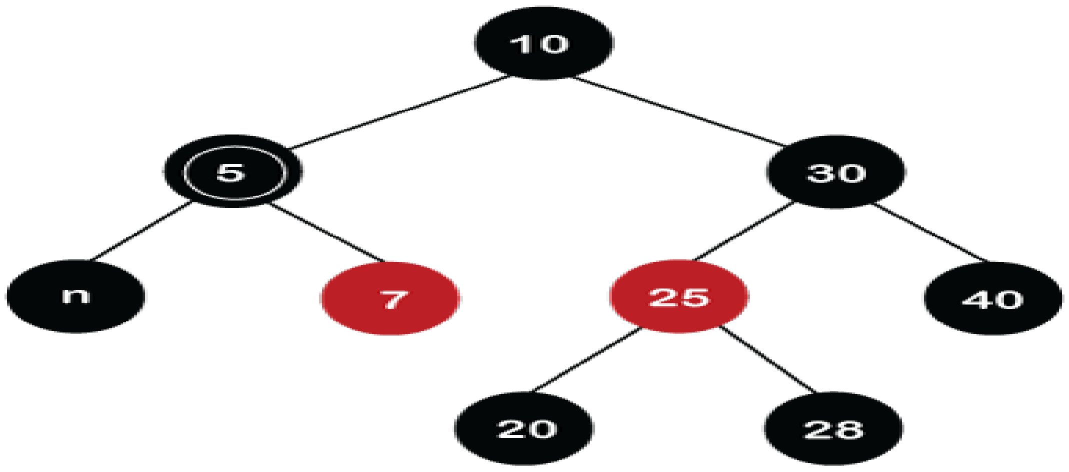
Case 5: If double black's sibling is black, sibling's child who is far from the double black is black, but near child to double black is red.

- Swap the color of double black's sibling and the sibling child which is nearer to the double black node.
- Rotate the sibling in the opposite direction of the double black.
- Apply case 6

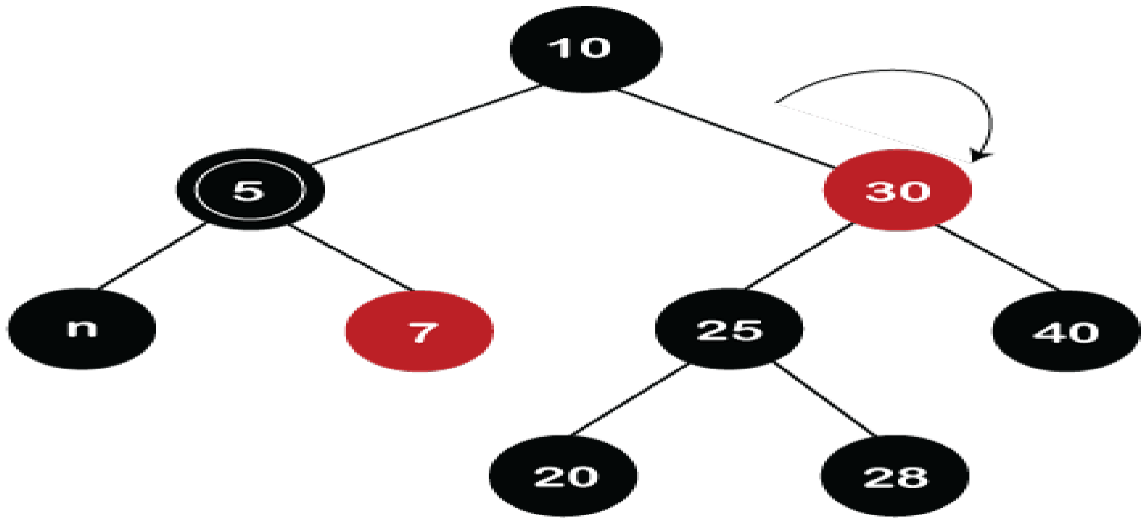
Suppose we want to delete the node 1 in the below tree.



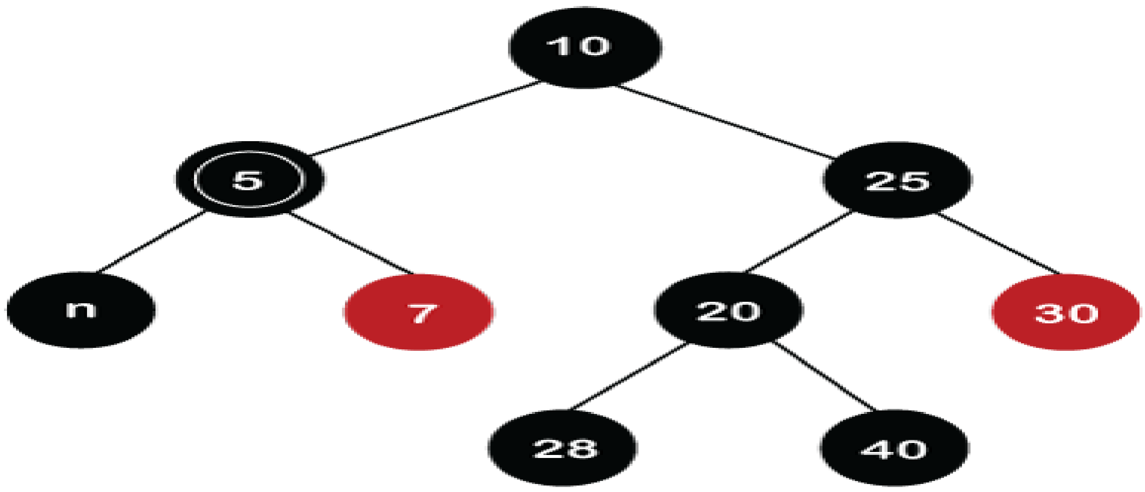
First, we replace the value 1 with the nil value. The node becomes double black as both the nodes, i.e., 1 and nil are black. It satisfies the case 3 that implies ***if DB's sibling is black and both its children are black.*** First, we remove the double black of the nil node. Since the parent of DB is Black, so when the black color is added to the parent node then it becomes double black. After adding the color, the double black's sibling color changes to Red as shown below.



We can observe in the above screenshot that the double black problem still exists in the tree. So, we will reapply the cases. We will apply case 5 because the sibling of node 5 is node 30, which is black in color, the child of node 30, which is far from node 5 is black, and the child of the node 30 which is near to node 5 is Red. In this case, first we will swap the color of node 30 and node 25 so the color of node 30 changes to Red and the color of node 25 changes to Black as shown below.



Once the swapping of the color between the nodes is completed, we need to rotate the sibling in the opposite direction of the double black node. In this rotation, the node 30 moves downwards while the node 25 moves upwards as shown below.



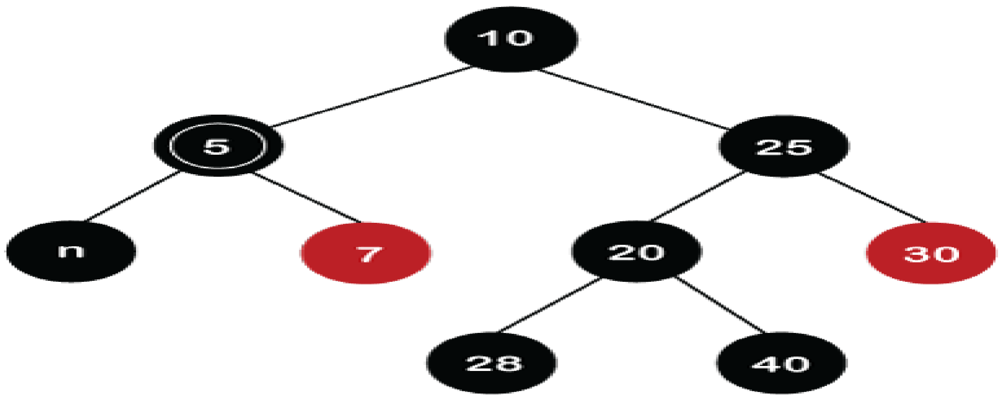
As we can observe in the above tree that double black situation still exists. So, we need to case 6. Let's first see what is case 6.

Case 6: If double black's sibling is black, far child is Red

- Swap the color of Parent and its sibling node.
- Rotate the parent towards the Double black's direction
- Remove Double black
- Change the Red color to black.

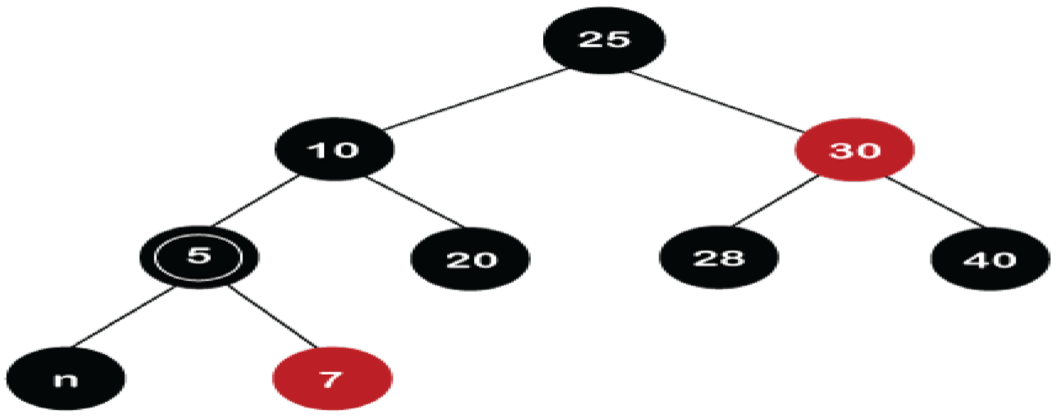
Now we will apply case 6 in the above example to solve the double black's situation.

In the above example, the double black is node 5, and the sibling of node 5 is node 25, which is black in color. The far child of the double black node is node 30, which is Red in color as shown in the below figure:

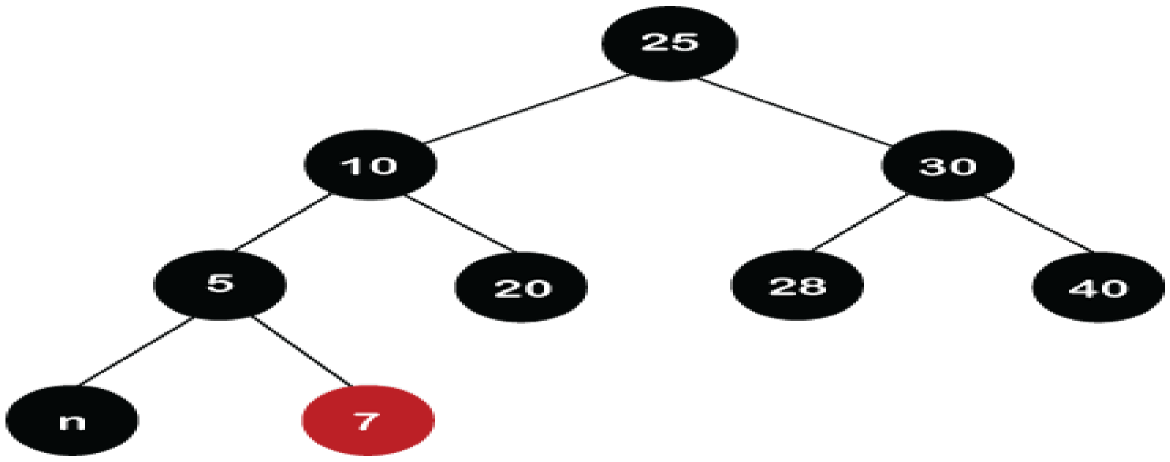


First, we will swap the colors of Parent and its sibling. The parent of node 5 is node 10, and the sibling node is node 25. The colors of both the nodes are black, so there is no swapping would occur.

In the second step, we need to rotate the parent in the double black's direction. After rotation, node 25 will move upwards, whereas node 10 will move downwards. Once the rotation is performed, the tree would like, as shown in the below figure:



In the next step, we will remove double black from node 5 and node 5 will give its black color to the far child, i.e., node 30. Therefore, the color of node 30 changes to black as shown in the below figure.



2-3 Trees | (Search, Insert and Deletion)

In binary search trees we have seen the average-case time for operations like search/insert/delete is O(log N) and the worst-case time is O(N) where N is the number of nodes in the tree.

Like other Trees include AVL trees, Red Black Tree, B tree, **2-3 Tree is also a height balanced tree**.
The time complexity of search/insert/delete is O(log N) .

A 2-3 tree is a *B-tree of order 3.*
Properties of 2-3 tree:

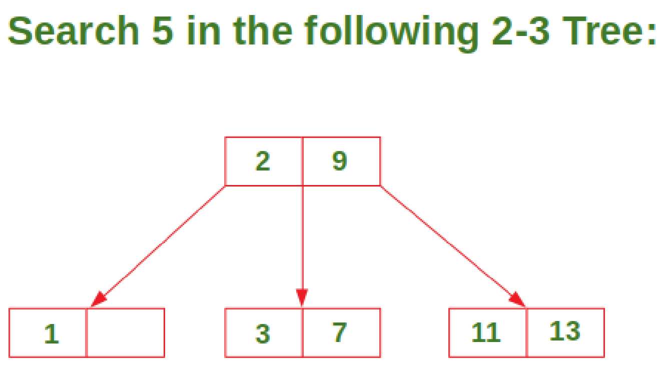
Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children.
Data is stored in sorted order.
It is a balanced tree.
All the leaf nodes are at same level.
Each node can either be leaf, 2 node, or 3 node.
Always insertion is done at leaf.
Search: To search a key **K** in given 2-3 tree **T**, we follow the following procedure:
Base cases:

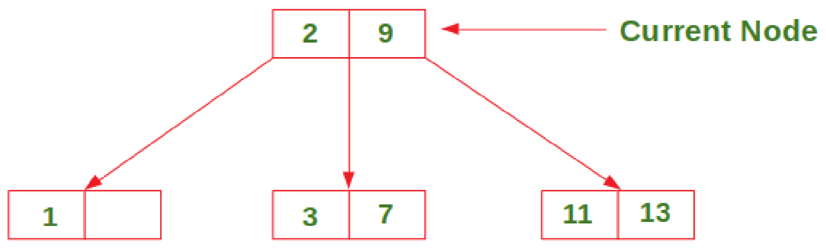
1. If **T** is empty, return False (key cannot be found in the tree).
2. If current node contains data value which is equal to **K**, return True.
3. If we reach the leaf-node and it doesn't contain the required key value **K**, return False.

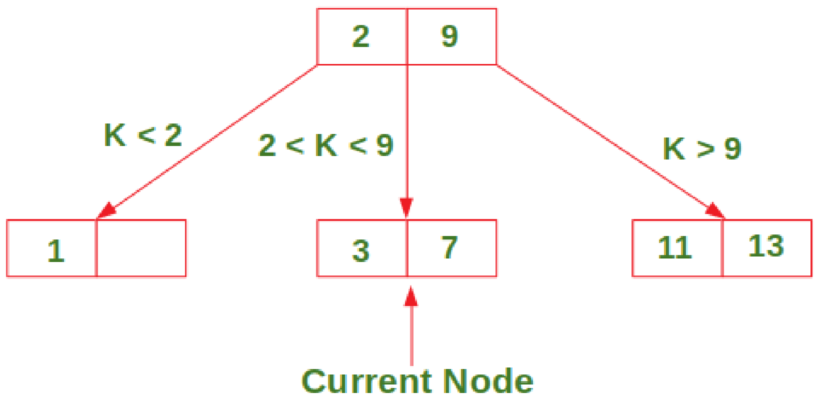
Recursive Calls:

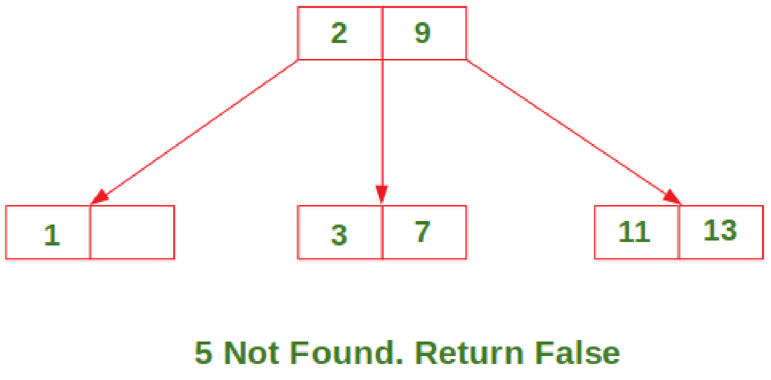
1. If **K** < currentNode.leftVal, we explore the left subtree of the current node.
2. Else if currentNode.leftVal < **K** < currentNode.rightVal, we explore the middle subtree of the current node.
3. Else if **K** > currentNode.rightVal, we explore the right subtree of the current node.

Consider the following example:

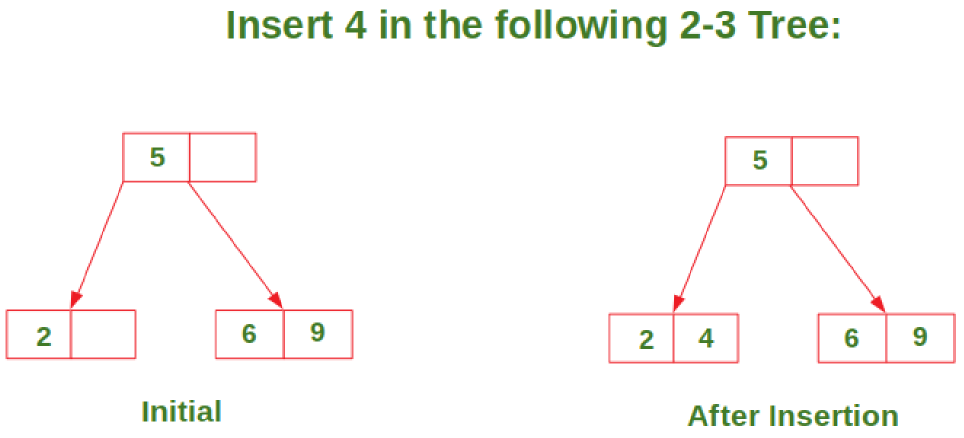






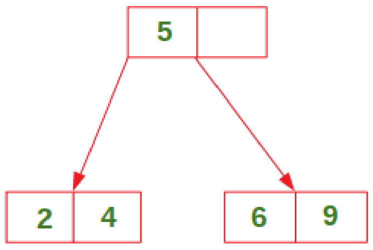


Insertion: There are 3 possible cases in insertion which have been discussed below:
Case 1: Insert in a node with only one data element

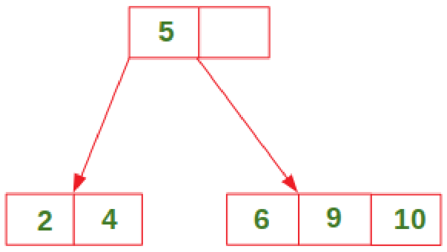


Case 2: Insert in a node with two data elements whose parent contains only one data element.

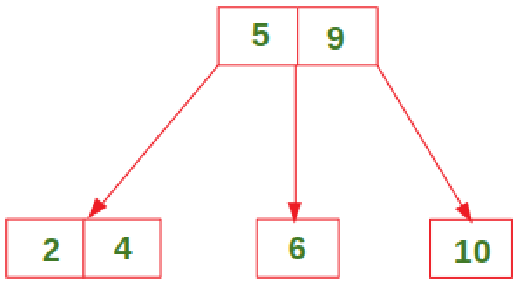
Insert 10 in the following 2-3 Tree:



Initial



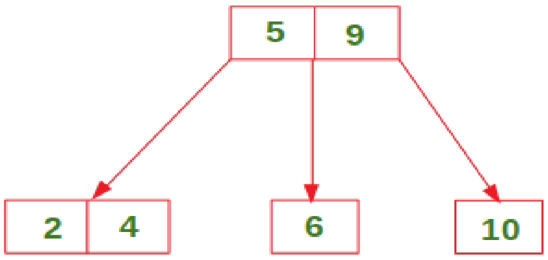
Temporary Node with 3 data elements



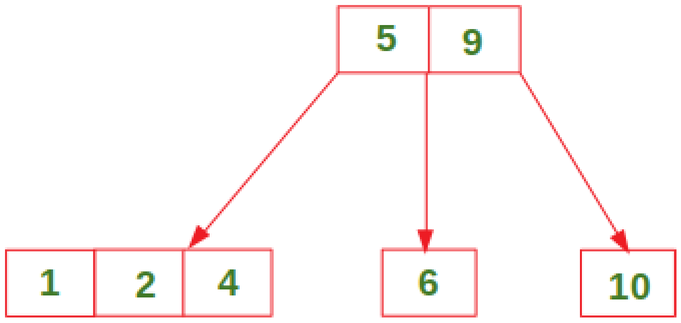
Move the middle element to
parent and split the current Node

Case 3: Insert in a node with two data elements whose parent also contains two data elements.

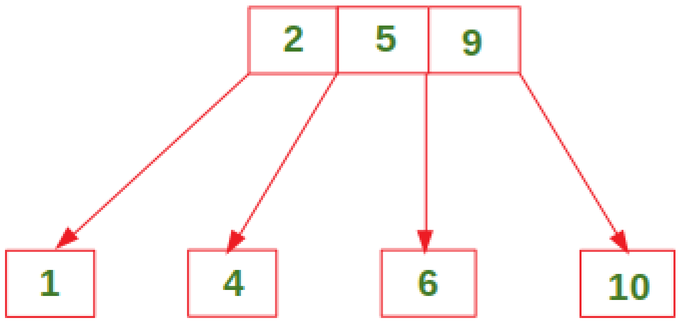
Insert 1 in the following 2-3 Tree:



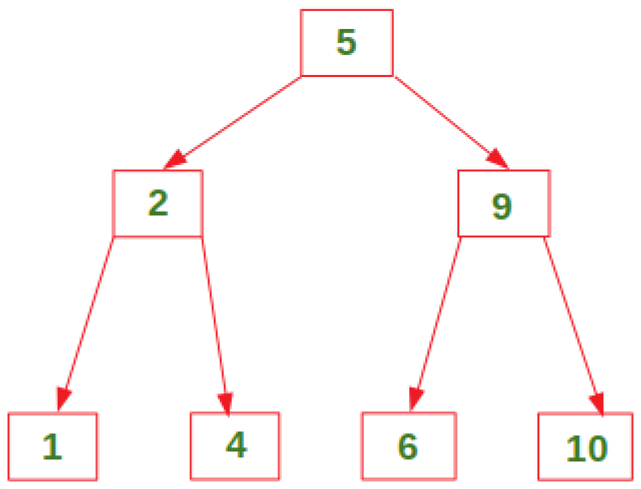
Initial



Temporary Node with 3 data elements



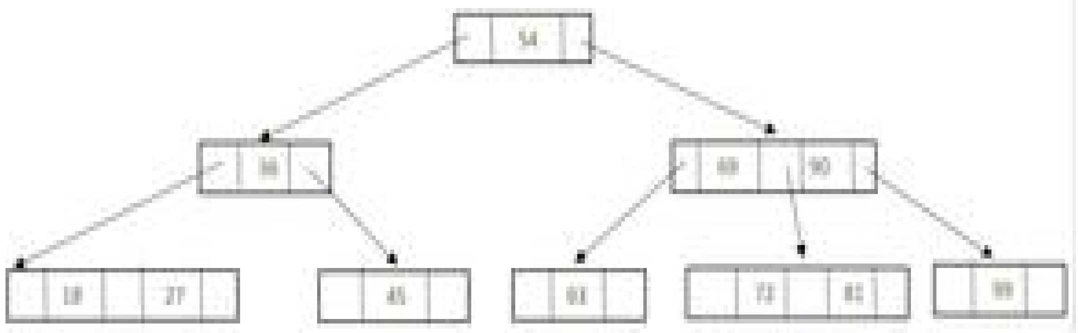
Move the middle element to the parent and split the current Node



Move the middle element to the parent
and split the current Node

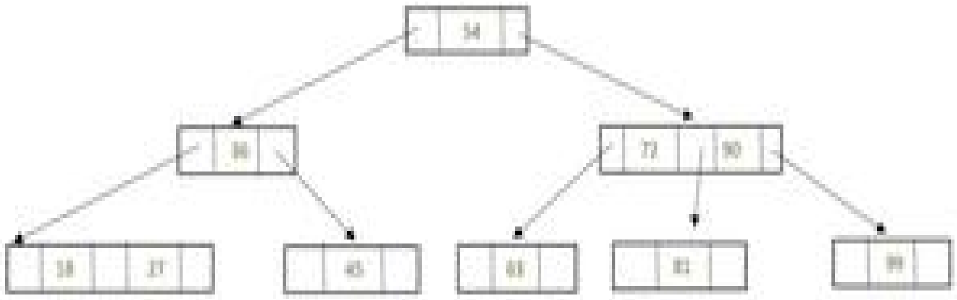
In Deletion Process for a specific value:
To delete a value, it is replaced by its in-order successor and then removed.
If a node is left with less than one data value then two nodes must be merged together.
If a node becomes empty after deleting a value, it is then merged with another node.
To Understand the deletion process-

Consider the 2-3 tree given below



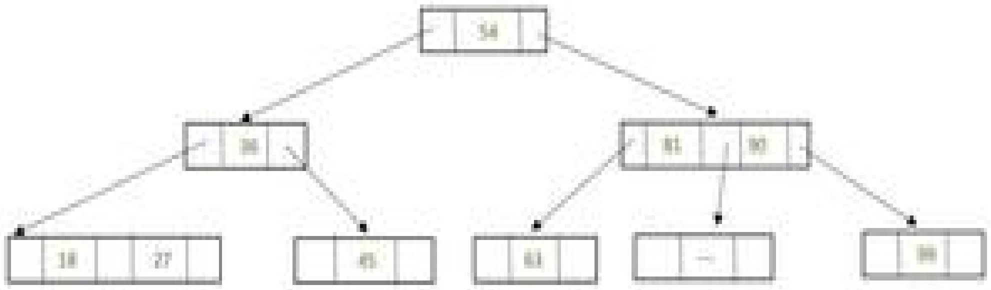
Given 2-3 Tree

delete the following values from it: 69,72, 99, 81.
To delete 69, swap it with its in-order successor, that is, 72. 69 now comes in the leaf node. Remove the value 69 from the leaf node.



After deletion 69

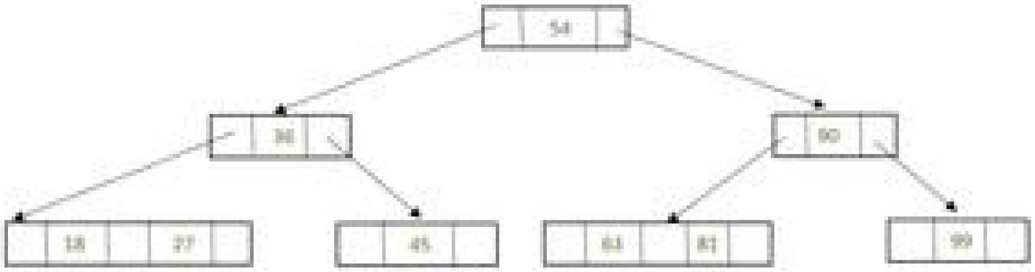
To delete 72, 72 is an internal node. To delete this value swap 72 with its in-order successor 81 so that 72 now becomes a leaf node. Remove the value 72 from the leaf node.



After deletion 72

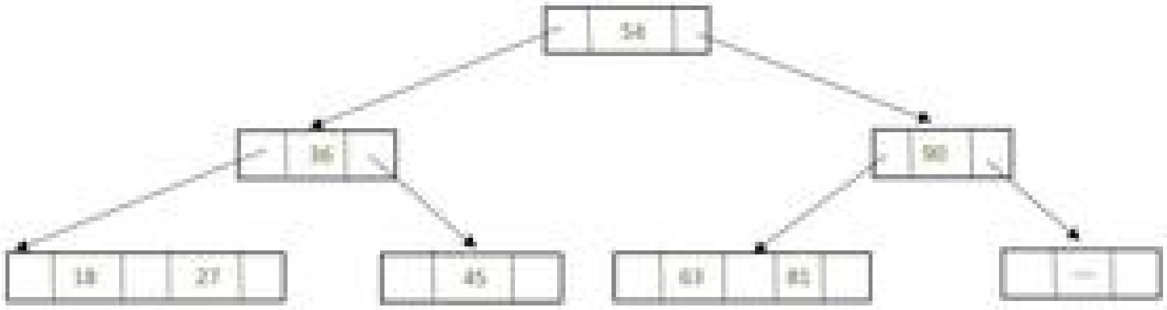
Now there is a leaf node that has less than 1 data value thereby violating the property of a 2-3 tree. So the node must be merged.

To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.



Rebalancing to Satisfy 23 Tree property

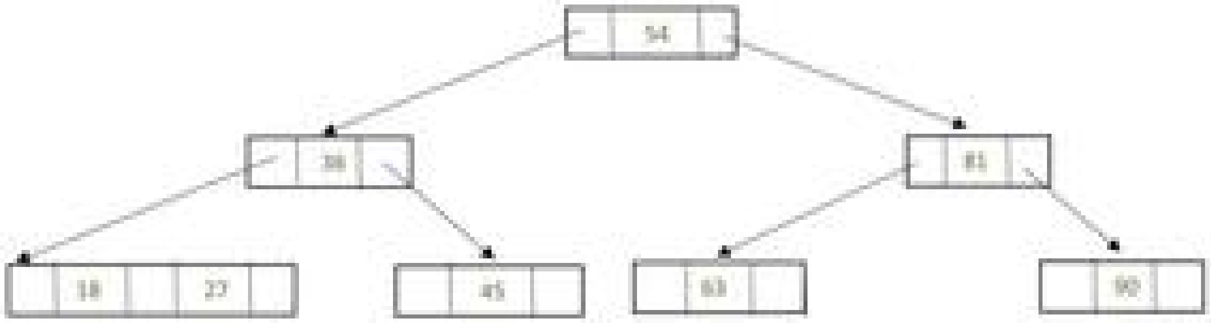
To delete 99, 99 is present in a leaf node, so the data value can be easily removed.



After deletion 99

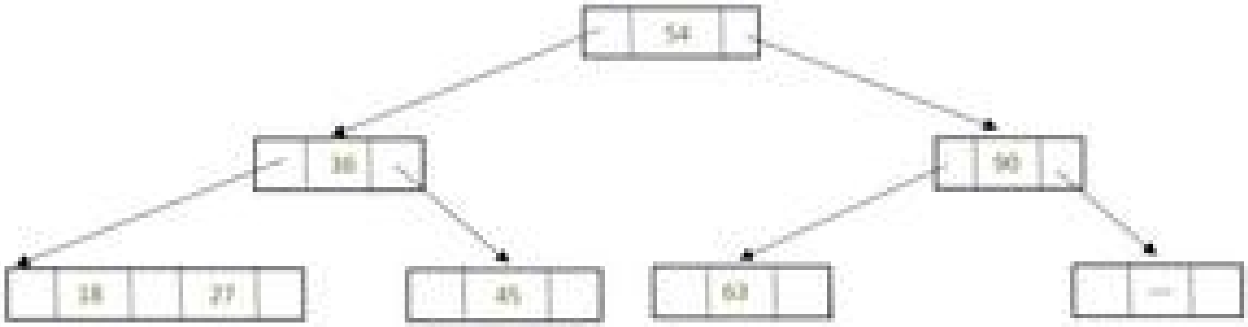
Now there is a leaf node that has less than 1 data value, thereby violating the property of a 2-3 tree.

So the node must be merged. To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.



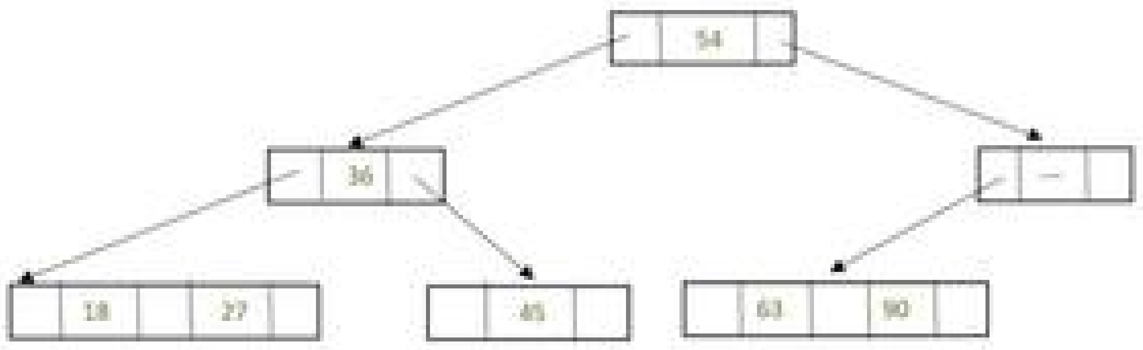
Rebalancing to Satisfy 2-3 Tree Property

To delete 81, 81 is an internal node. To delete this value swap 81 with its in-order successor 90 so that 81 now becomes a leaf node. Remove the value 81 from the leaf node.



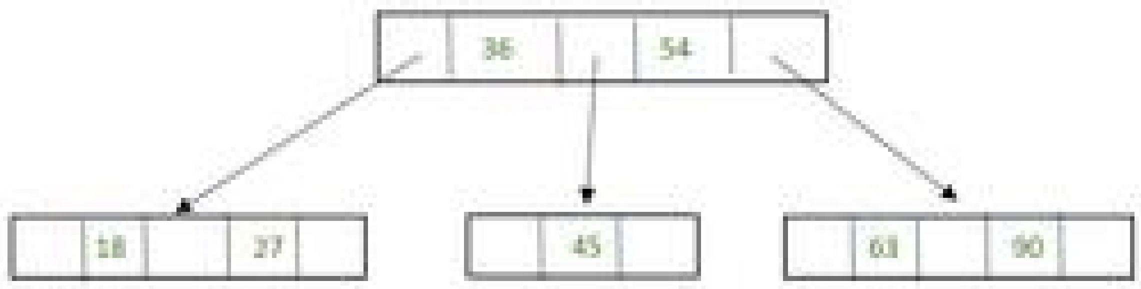
After deletion 81

Now there is a leaf node that has less than 1 data value, thereby violating the property of a 2-3 tree. So the node must be merged. To merge the node, pull down the lowest data value in the parent's node and merge it with its left sibling.



Rebalancing to Satisfy 2-3 Tree property

As internal node cannot be empty. So now pull down the lowest data value from the parent's node and merge the empty node with its left sibling



Rebalancing to Satisfy 2-3 Tree Property

NOTE: In a 2-3 tree, each interior node has either two or three children. This means that a **2-3 tree is not a binary tree.**