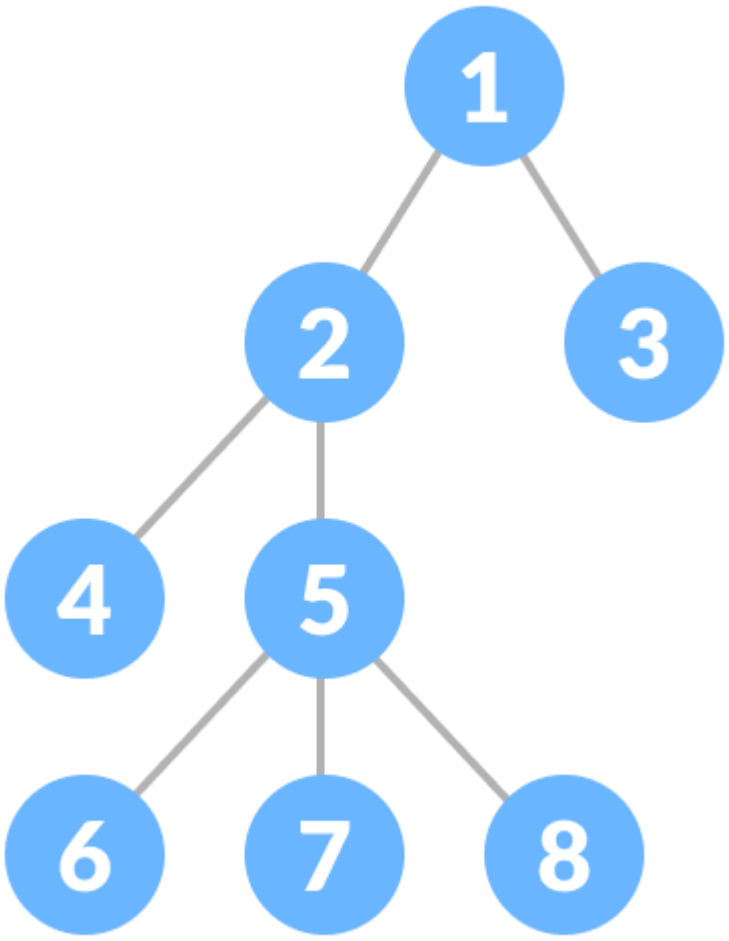
Tree Data Structure

In this tutorial, you will learn about tree data structure. Also, you will learn about different types of trees and the terminologies used in tree.

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



A Tree

Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

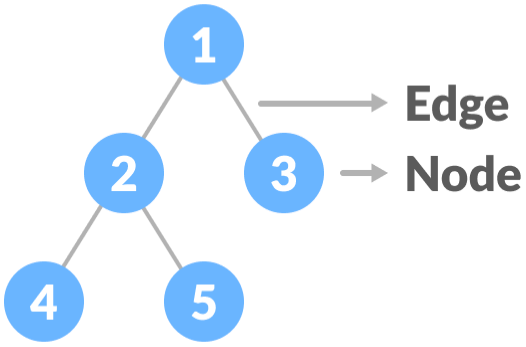
Node

A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes.



Nodes and edges of a tree

Root

It is the topmost node of a tree.

Height of a Node

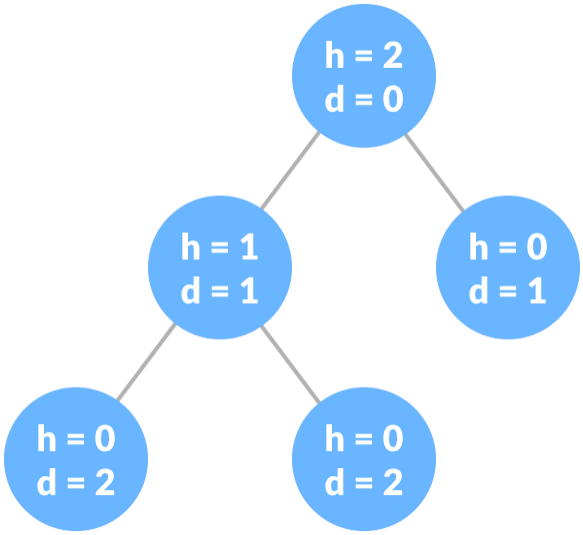
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.



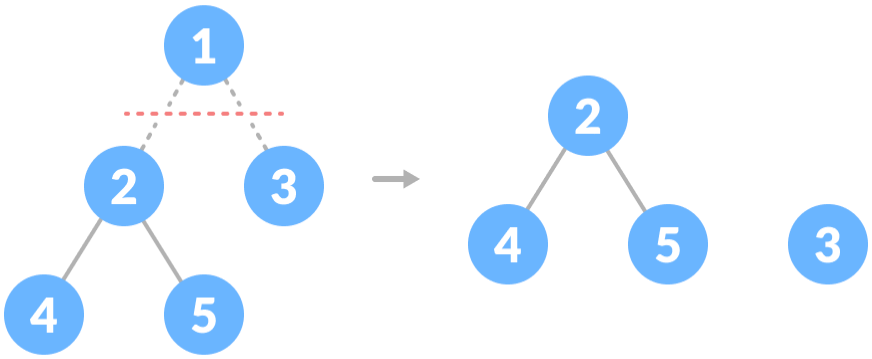
Height and depth of each node in a tree

Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



Creating forest from a tree

You can create a forest by cutting the root of a tree.

Tree Traversal

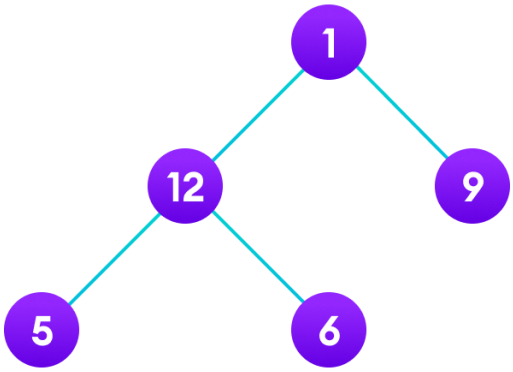
In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

To learn more, please visit

Tree Traversal - inorder, preorder and postorder

In this tutorial, you will learn about different tree traversal techniques. Also, you will find working examples of different tree traversal methods in C.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.



Tree traversal

Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

```
1 -> 12 -> 5 -> 6 -> 9
```

Starting from bottom, Left to right

```
5 -> 6 -> 12 -> 9 -> 1
```

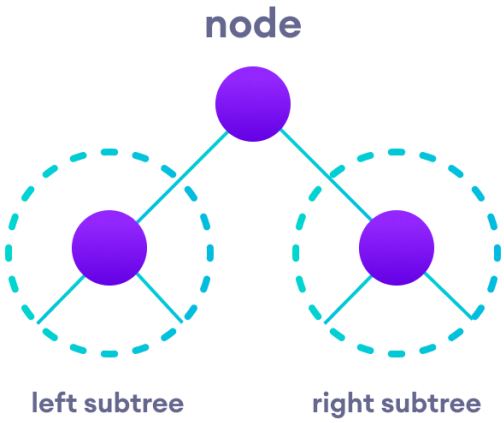
Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes.

Instead, we use traversal methods that take into account the basic structure of a tree i.e.

```
struct node {
   int data;
   struct node* left;
   struct node* right;
}
```

The struct node pointed to by `left` and `right` might have other left and right children so we should think of them as sub-trees instead of sub-nodes. According to this structure, every tree is a combination of

- A node carrying data
- Two subtrees



Left and Right Subtree

Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well.

Depending on the order in which we do this, there can be three types of traversal.

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

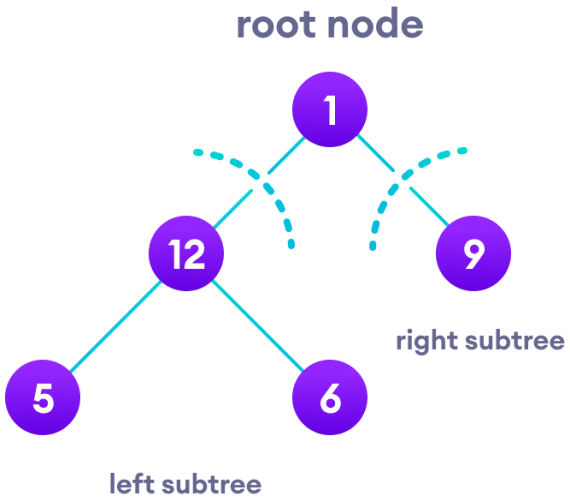
```
display(root->data)
preorder(root->left)
preorder(root->right)
```

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

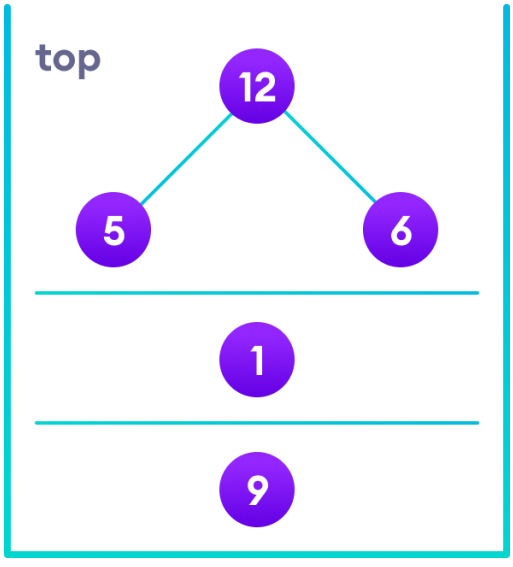
Let's visualize in-order traversal. We start from the root node.



Left and Right Subtree

We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.



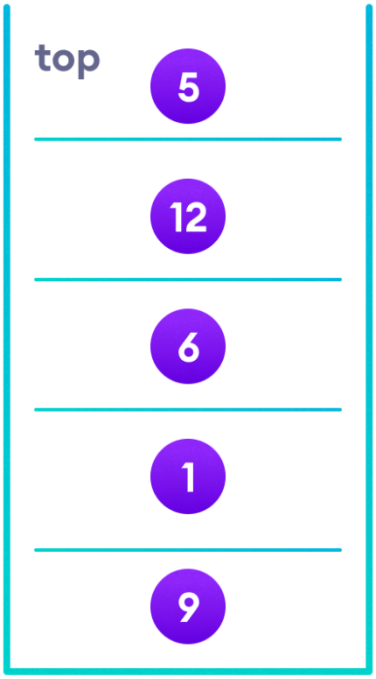
Stack

Now we traverse to the subtree pointed on the TOP of the stack.

Again, we follow the same rule of inorder

```
Left subtree -> root -> right subtree
```

After traversing the left subtree, we are left with



Final Stack

Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as

```
5 -> 12 -> 6 -> 1 -> 9
```

We don't have to create the stack ourselves because recursion maintains the correct order for us.

```
// Tree traversal in C

#include <stdio.h>
#include <stdlib.h>

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}
```

```
// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}

// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
```

```
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

Tree Applications

- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data

- Compilers use a syntax tree to validate the syntax of every program you write.

Types of Tree

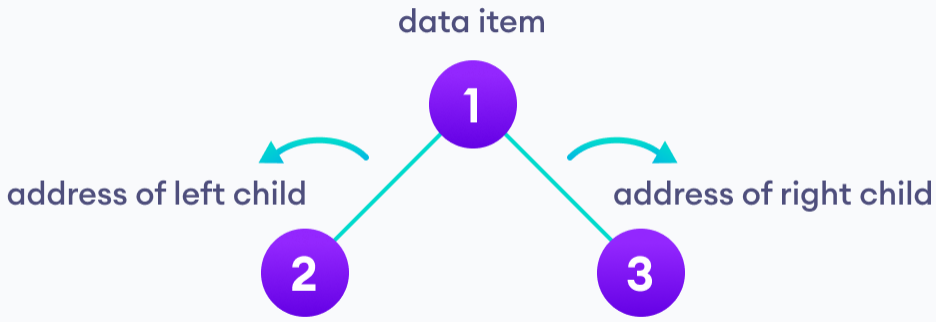
1. Binary Tree
2. Binary Search Tree
3. AVL Tree
4. B-Tree

Binary Tree

In this tutorial, you will learn about binary tree and its different types. Also, you will find working examples of binary tree in C.

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- data item
- address of left child
- address of right child

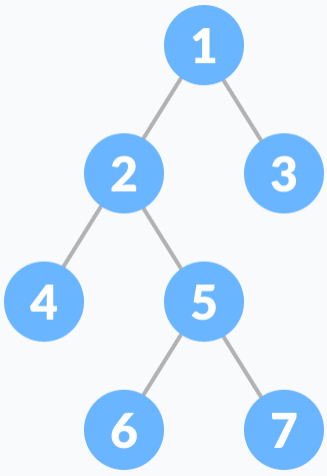


Binary Tree

Types of Binary Tree

1. Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



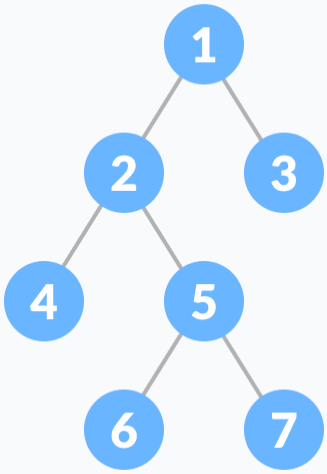
Full Binary Tree

Full Binary Tree

In this tutorial, you will learn about full binary tree and its different theorems. Also, you will find working examples to check full binary tree in C.

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

It is also known as **a proper binary tree**.



Full Binary Tree

Full Binary Tree Theorems

```
Let, i = the number of internal nodes
     n = be the total number of nodes
     l = number of leaves
     λ = number of levels
```

1. The number of leaves is $i + 1$.
2. The total number of nodes is $2i + 1$.
3. The number of internal nodes is $(n - 1) / 2$.
4. The number of leaves is $(n + 1) / 2$.

5. The total number of nodes is $2l - 1$.
6. The number of internal nodes is $l - 1$.
7. The number of leaves is at most $2^{\lambda - 1}$.

C Examples

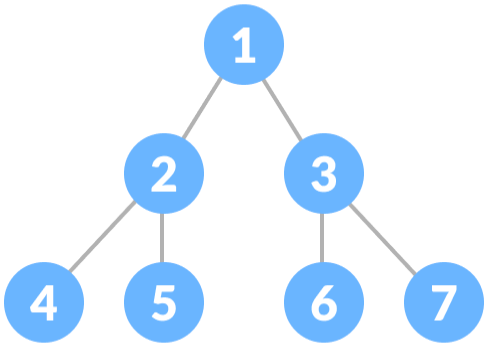
The following code is for checking if a tree is a full binary tree.

```
// Checking if a binary tree is a full binary tree in C


#include <stdbool.h>

#include <stdio.h>

#include <stdlib.h>


struct Node {

  int item;

  struct Node *left, *right;

};


// Creation of new Node

struct Node *createNewNode(char k) {

  struct Node *node = (struct Node *)malloc(sizeof(struct Node));

  node->item = k;

  node->right = node->left = NULL;

  return node;

}


bool isFullBinaryTree(struct Node *root) {

  // Checking tree emptiness
```

```
  if (root == NULL)
    return true;

  // Checking the presence of children
  if (root->left == NULL && root->right == NULL)
    return true;

  if ((root->left) && (root->right))
    return (isFullBinaryTree(root->left) && isFullBinaryTree(root->right));

  return false;
}

int main() {
  struct Node *root = NULL;
  root = createNewNode(1);
  root->left = createNewNode(2);
  root->right = createNewNode(3);

  root->left->left = createNewNode(4);
  root->left->right = createNewNode(5);
  root->left->right->left = createNewNode(6);
  root->left->right->right = createNewNode(7);

  if (isFullBinaryTree(root))
    printf("The tree is a full binary tree\n");
  else
    printf("The tree is not a full binary tree\n");                }
```

2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

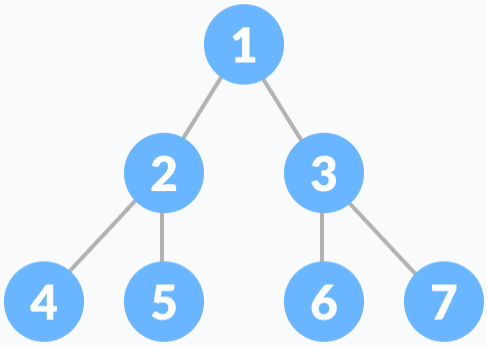


Perfect Binary Tree

Perfect Binary Tree

In this tutorial, you will learn about the perfect binary tree. Also, you will find working examples for checking a perfect binary tree in C.

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

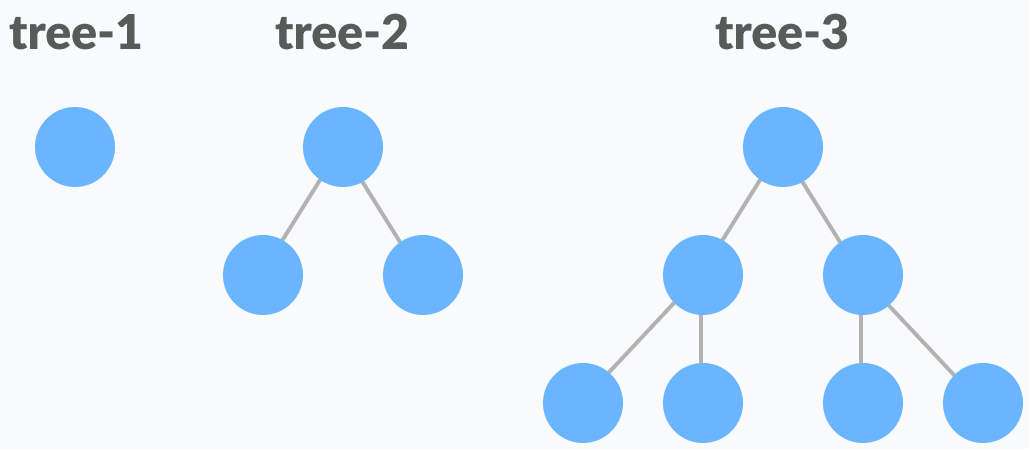
All the internal nodes have a degree of 2.

Perfect Binary Tree Theorems

1. A perfect binary tree of height h has $2^{h+1} - 1$ node.
2. A perfect binary tree with n nodes has height $\log(n + 1) - 1 = \Theta(\ln(n))$.
3. A perfect binary tree of height h has 2^h leaf nodes.
4. The average depth of a node in a perfect binary tree is $\Theta(\ln(n))$.

Recursively, a perfect binary tree can be defined as:

1. If a single node has no children, it is a perfect binary tree of height $h = 0$,
2. If a node has $h > 0$, it is a perfect binary tree if both of its subtrees are of height $h - 1$ and are non-overlapping.



Perfect Binary Tree (Recursive Representation)

C Examples

The following code is for checking whether a tree is a perfect binary tree.

```
// Checking if a binary tree is a perfect binary tree in C


#include <stdbool.h>

#include <stdio.h>
```

```
#include <stdlib.h>

struct node {
  int data;
  struct node *left;
  struct node *right;
};

// Creating a new node
struct node *newnode(int data) {
  struct node *node = (struct node *)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return (node);
}

// Calculate the depth
int depth(struct node *node) {
  int d = 0;
  while (node != NULL) {
    d++;
    node = node->left;
  }
  return d;
}

// Check if the tree is perfect
```

```
bool is_perfect(struct node *root, int d, int level) {
   // Check if the tree is empty
  if (root == NULL)
    return true;

  // Check the presence of children
  if (root->left == NULL && root->right == NULL)
    return (d == level + 1);

  if (root->left == NULL || root->right == NULL)
    return false;

  return is_perfect(root->left, d, level + 1) &&
       is_perfect(root->right, d, level + 1);
}

// Wrapper function
bool is_Perfect(struct node *root) {
  int d = depth(root);
  return is_perfect(root, d, 0);
}

int main() {
  struct node *root = NULL;
  root = newnode(1);
  root->left = newnode(2);
  root->right = newnode(3);
  root->left->left = newnode(4);
  root->left->right = newnode(5);
```

```
  root->right->left = newnode(6);


  if (is_Perfect(root))
    printf("The tree is a perfect binary tree\n");
  else
    printf("The tree is not a perfect binary tree\n");
}
```

AVL Tree

In this tutorial, you will learn what an avl tree is. Also, you will find working examples of various operations performed on an avl tree in C.

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

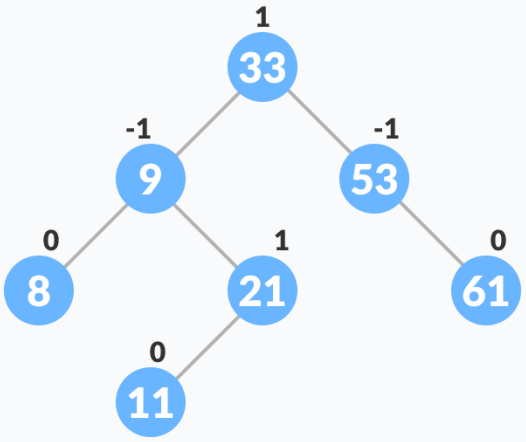
Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:



Avl tree

Operations on an AVL tree

Various operations that can be performed on an AVL tree are:

Rotating the subtrees in an AVL Tree

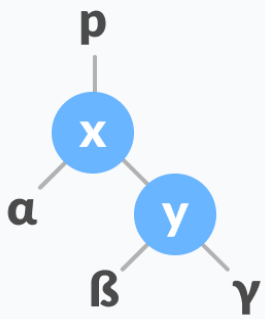
In rotation operation, the positions of the nodes of a subtree are interchanged.

There are two types of rotations:

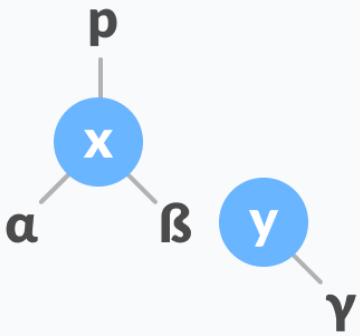
Left Rotate

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

Algorithm

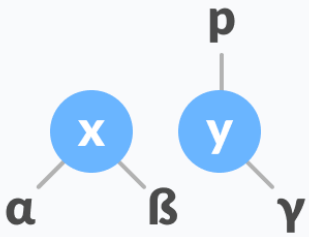


1. Let the initial tree be: Left rotate
2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.

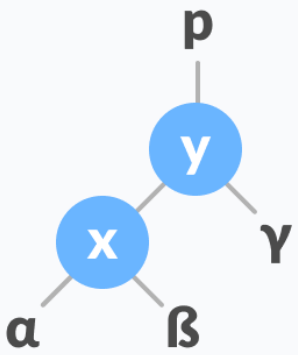


Assign x as the parent of the left subtree of y

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



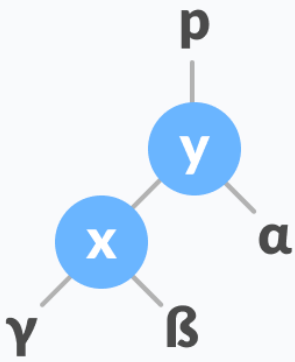
5. Else assign `y` as the right child of `p`. Change the parent of x to that of y



6. Make `y` as the parent of `x`. Assign y as the parent of x.

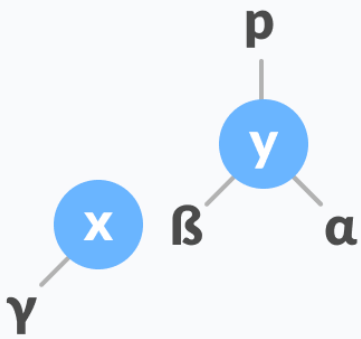
Right Rotate

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



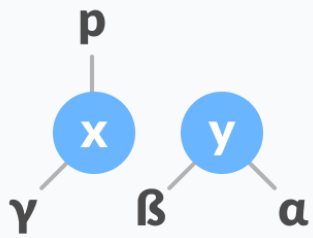
1. Let the initial tree be: Initial tree

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

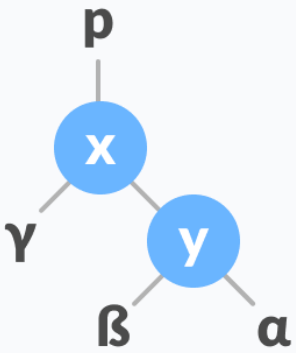


Assign y as the parent of the right subtree of x

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



5. Else assign `x` as the left child of `p`. Assign the parent of y as the parent of x.

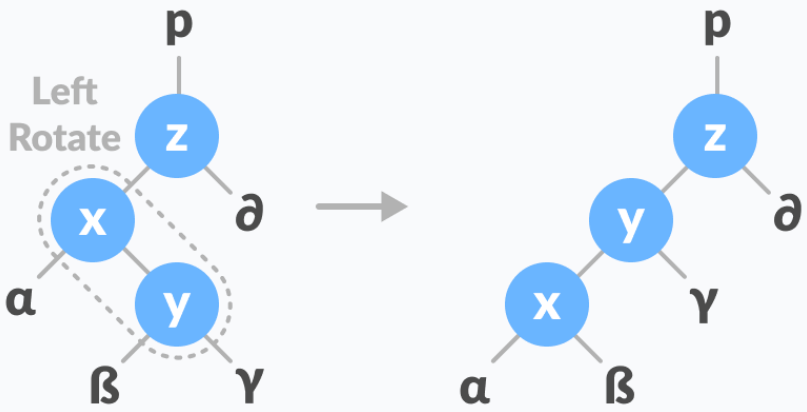


6. Make `x` as the parent of `y`. Assign x as the parent of y

Left-Right and Right-Left Rotate

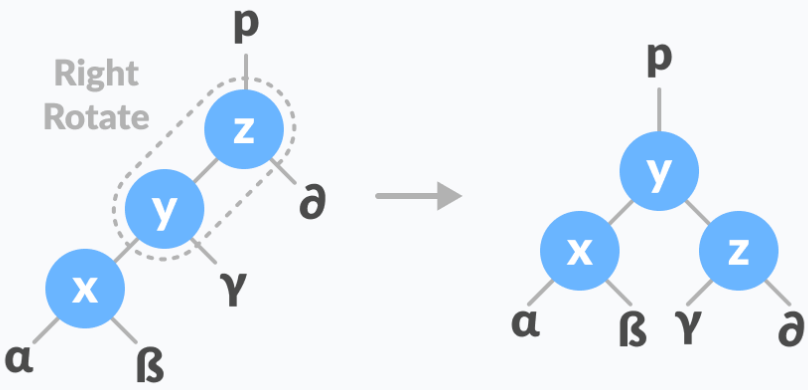
In left-right rotation, the arrangements are first shifted to the left and then to the right.

1. Do left rotation on x-y.



Left rotate x-y

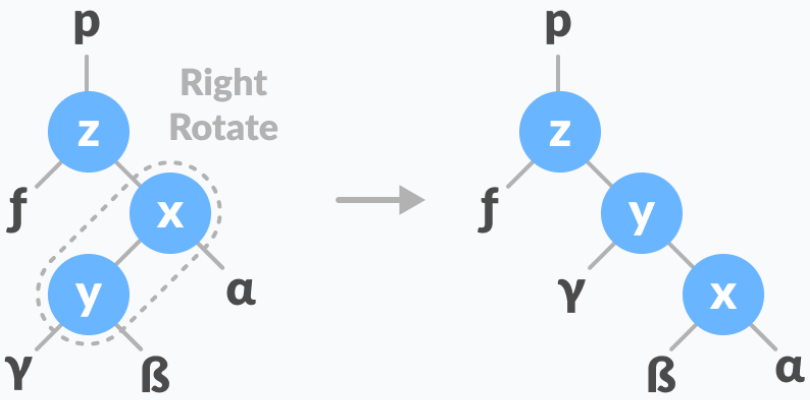
2. Do right rotation on y-z.



Right rotate z-y

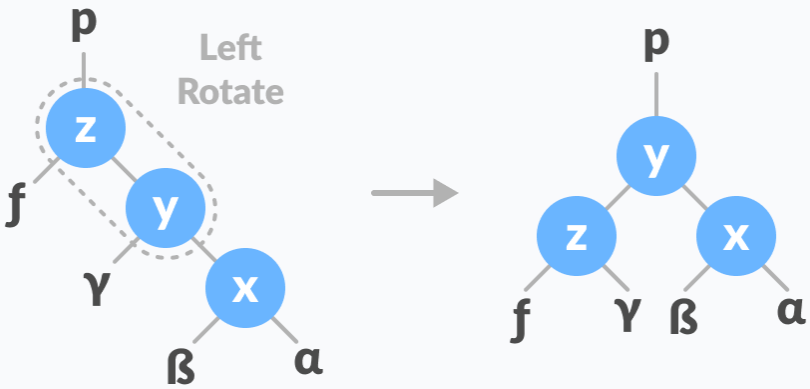
In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



Right rotate x-y

2. Do left rotation on z-y.

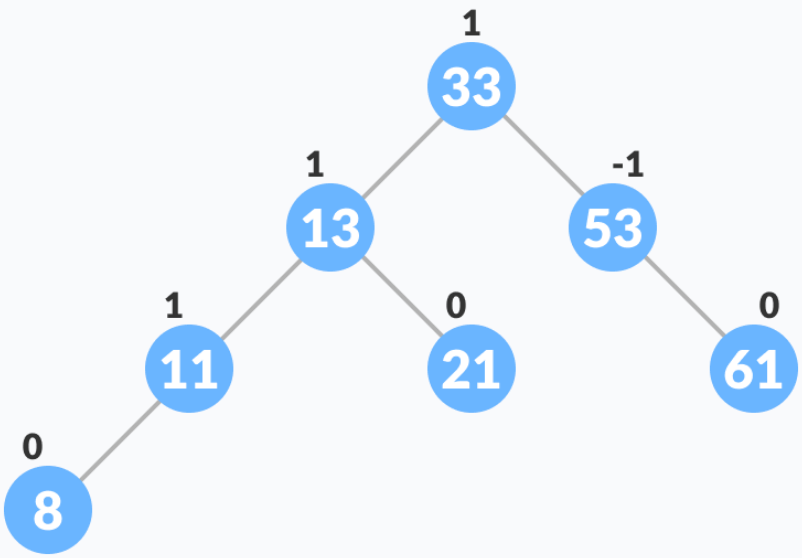


Left rotate z-y

Algorithm to insert a newNode

A `newNode` is always inserted as a leaf node with balance factor equal to 0.

1. Let the initial tree be:

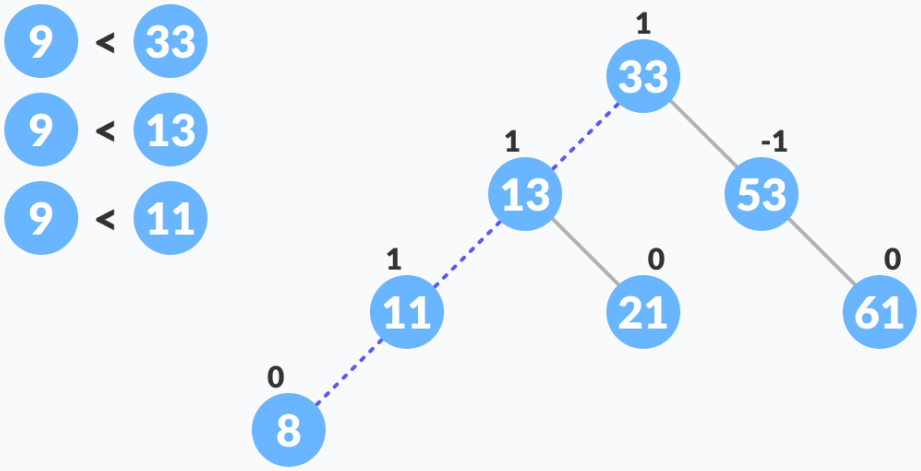


Initial tree for insertion

Let the node to be inserted be: New node

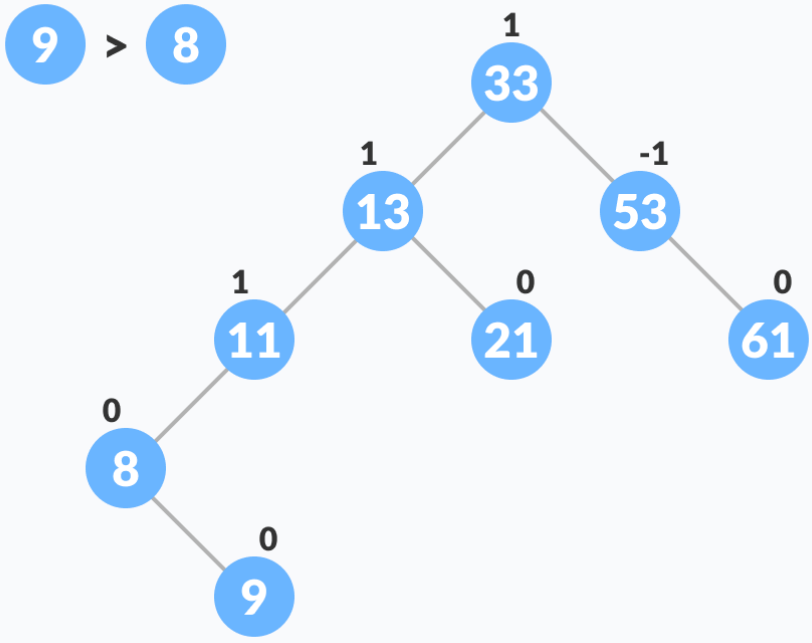
2. Go to the appropriate leaf node to insert a `newNode` using the following recursive steps. Compare `newKey` with `rootKey` of the current tree.
    a. If `newKey` < `rootKey`, call insertion algorithm on the left subtree of the current node until the leaf node is reached.
    b. Else if `newKey` >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

c. Else, return `leafNode`.



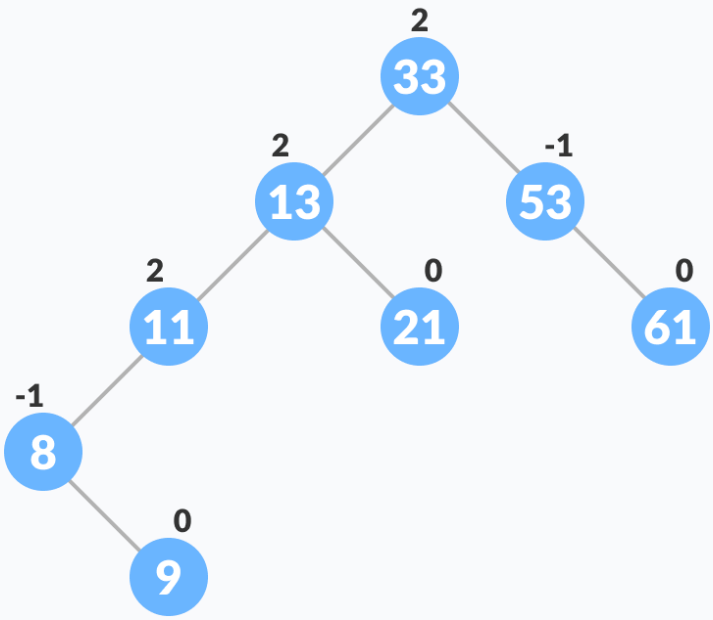
Finding the location to insert newNode

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



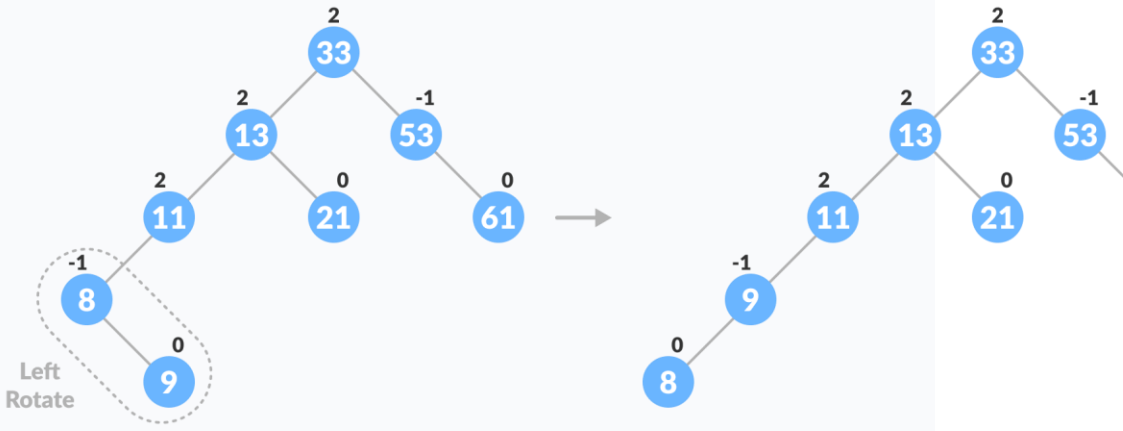
Inserting the new node

4. Update `balanceFactor` of the nodes.

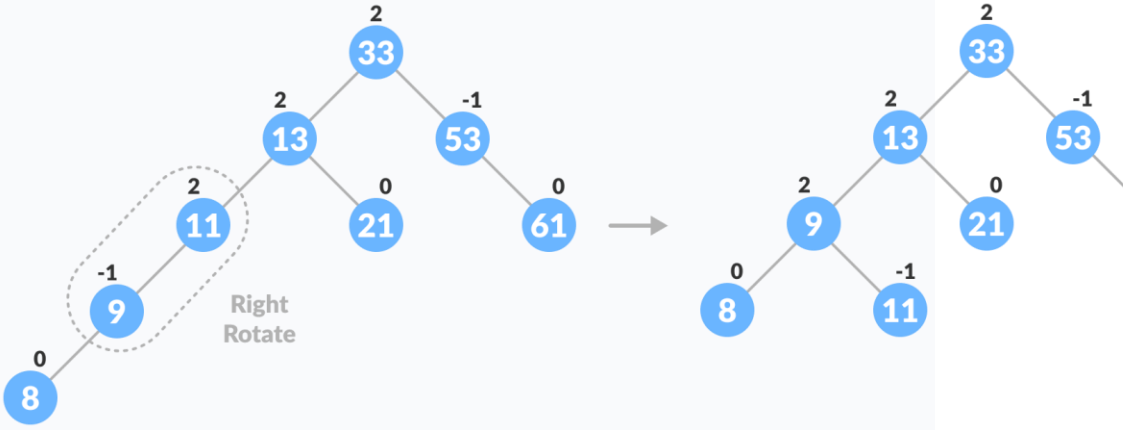


Updating the balance factor after insertion

5. If the nodes are unbalanced, then rebalance the node.
    a. If `balanceFactor` > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation
        a. If `newNodeKey` < `leftChildKey` do right rotation.
        b. Else, do left-right rotation.



Balancing the tree with rotation

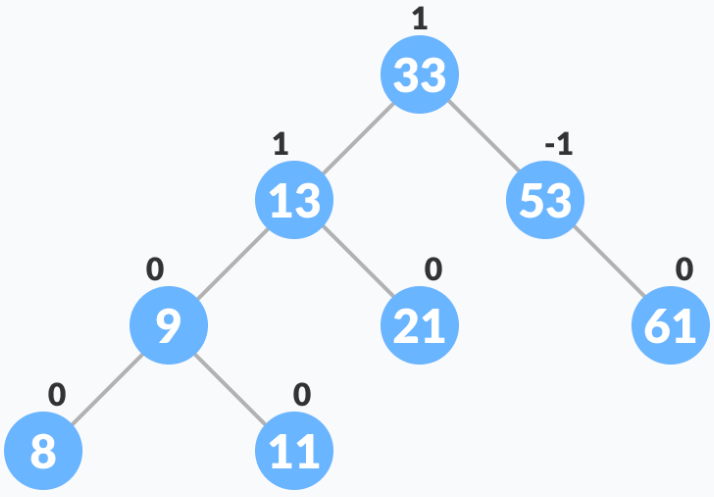


Balancing the tree with rotation

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

   a. If `newNodeKey` > `rightChildKey` do left rotation.

   b. Else, do right-left rotation

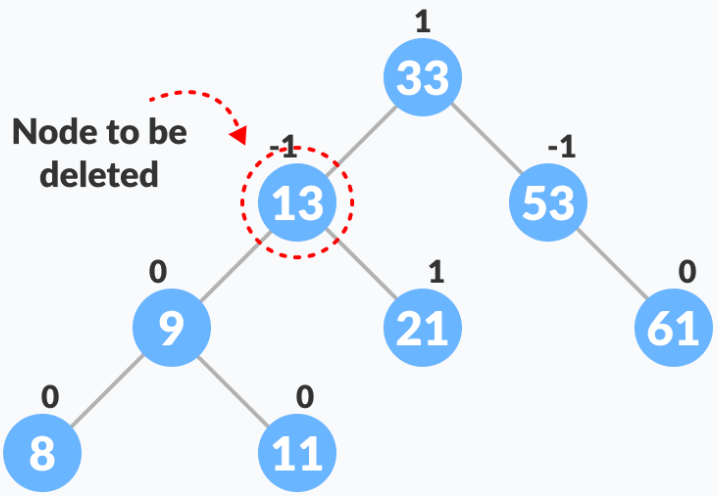


6. The final tree is: Final balanced tree

Algorithm to Delete a node

A node is always deleted as a leaf node. After deleting a node, the balance factors of the nodes get changed. In order to rebalance the balance factor, suitable rotations are performed.

1. Locate `nodeToBeDeleted` (recursion is used to find `nodeToBeDeleted` in the code used below). Locating the node to be deleted

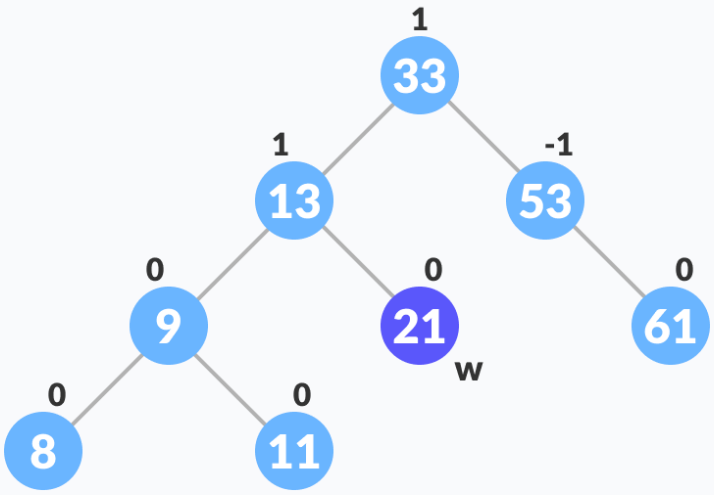


2. There are three cases for deleting a node:

    a. If `nodeToBeDeleted` is the leaf node (ie. does not have any child), then remove `nodeToBeDeleted`.

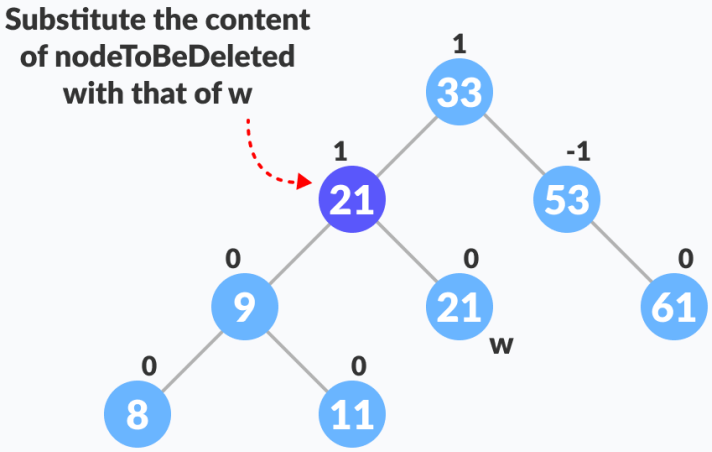
    b. If `nodeToBeDeleted` has one child, then substitute the contents of `nodeToBeDeleted` with that of the child. Remove the child.

    c. If `nodeToBeDeleted` has two children, find the inorder successor `w` of `nodeToBeDeleted` (ie. node with a minimum value of key in the right subtree).



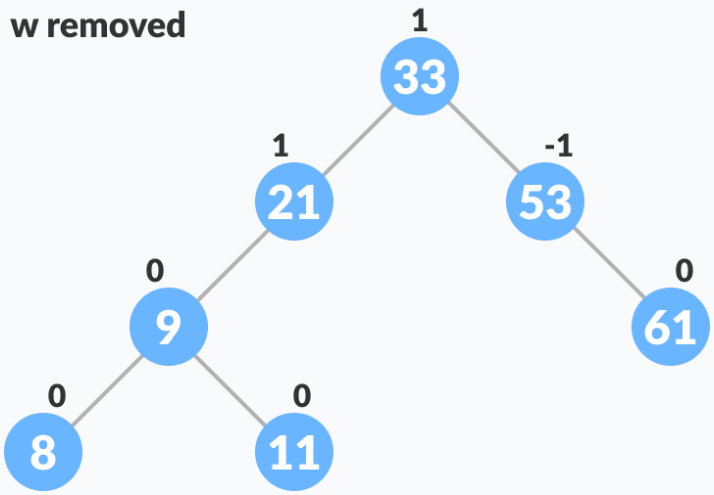
Finding the successor

a. Substitute the contents of `nodeToBeDeleted` with that of `w`.



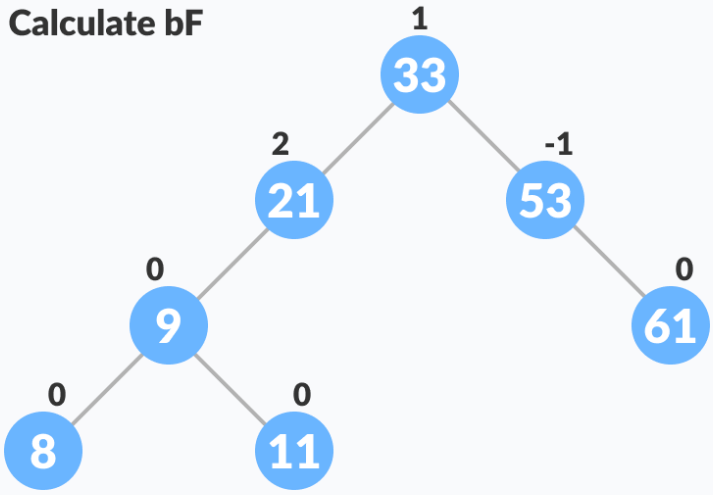
Substitute the node to be deleted

b. Remove the leaf node `w`.



Remove w

3. Update `balanceFactor` of the nodes.

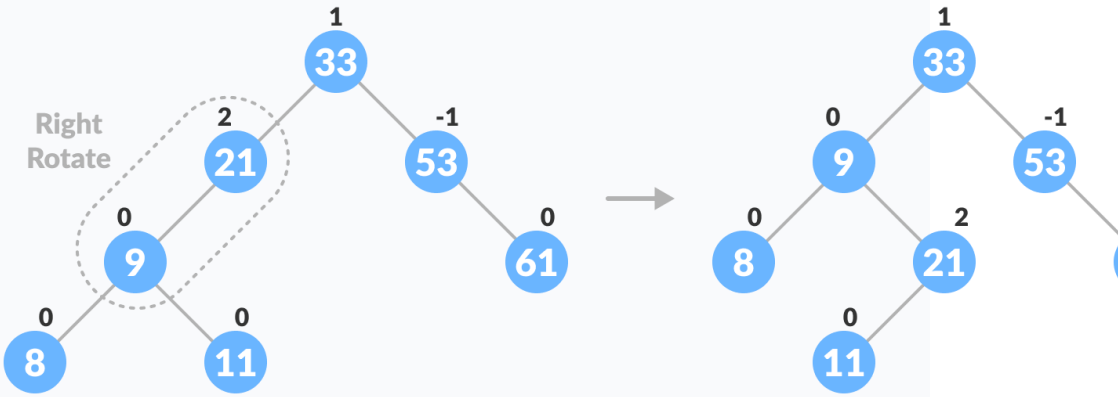


Update bf

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

   a. If `balanceFactor` of `currentNode` > 1,

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



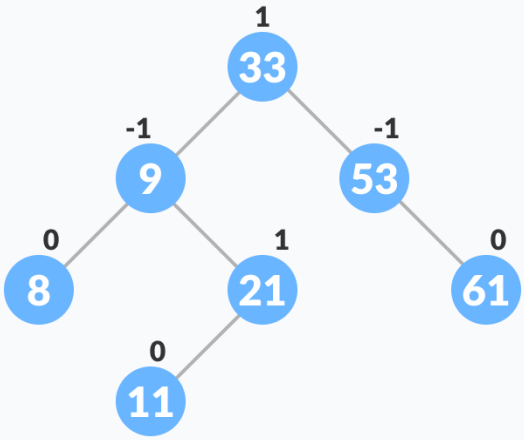
Right-rotate for balancing the tree

b. Else do left-right rotation.

b. If `balanceFactor` of `currentNode` < -1,

a. If `balanceFactor` of `rightChild` <= 0, do left rotation.

b. Else do right-left rotation.



5. The final tree is: Avl tree final

C Examples

// AVL tree implementation in C

#include <stdio.h>

```
#include <stdlib.h>

// Create Node
struct Node {
  int key;
  struct Node *left;
  struct Node *right;
  int height;
};

int max(int a, int b);

// Calculate height
int height(struct Node *N) {
  if (N == NULL)
    return 0;
  return N->height;
}

int max(int a, int b) {
  return (a > b) ? a : b;
}

// Create a node
struct Node *newNode(int key) {
  struct Node *node = (struct Node *)
    malloc(sizeof(struct Node));
  node->key = key;
  node->left = NULL;
```

```
  node->right = NULL;
  node->height = 1;
  return (node);
}

// Right rotate
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;

  x->right = y;
  y->left = T2;

  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;

  return x;
}

// Left rotate
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;
```

```
  return y;
}

// Get the balance factor
int getBalance(struct Node *N) {
  if (N == NULL)
    return 0;
  return height(N->left) - height(N->right);
}

// Insert node
struct Node *insertNode(struct Node *node, int key) {
  // Find the correct position to insertNode the node and insertNode it
  if (node == NULL)
    return (newNode(key));

  if (key < node->key)
    node->left = insertNode(node->left, key);
  else if (key > node->key)
    node->right = insertNode(node->right, key);
  else
    return node;

  // Update the balance factor of each node and
  // Balance the tree
  node->height = 1 + max(height(node->left),
               height(node->right));
```

```
  int balance = getBalance(node);
  if (balance > 1 && key < node->left->key)
    return rightRotate(node);

  if (balance < -1 && key > node->right->key)
    return leftRotate(node);

  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }

  if (balance < -1 && key < node->right->key) {
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }

  return node;
}

struct Node *minValueNode(struct Node *node) {
  struct Node *current = node;

  while (current->left != NULL)
    current = current->left;

  return current;
}
```

```
// Delete a nodes
struct Node *deleteNode(struct Node *root, int key) {
 // Find the node and delete it
 if (root == NULL)
  return root;

 if (key < root->key)
  root->left = deleteNode(root->left, key);

 else if (key > root->key)
  root->right = deleteNode(root->right, key);

 else {
  if ((root->left == NULL) || (root->right == NULL)) {
   struct Node *temp = root->left ? root->left : root->right;

   if (temp == NULL) {
    temp = root;
    root = NULL;
   } else
    *root = *temp;
   free(temp);
  } else {
   struct Node *temp = minValueNode(root->right);

   root->key = temp->key;

   root->right = deleteNode(root->right, temp->key);
  }
```

```
  }

  if (root == NULL)
    return root;

  // Update the balance factor of each node and
  // balance the tree
  root->height = 1 + max(height(root->left),
              height(root->right));

  int balance = getBalance(root);
  if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

  if (balance > 1 && getBalance(root->left) < 0) {
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

  if (balance < -1 && getBalance(root->right) > 0) {
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}
```

```
// Print the tree
void printPreOrder(struct Node *root) {
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  }
}

int main() {
  struct Node *root = NULL;

  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);

  printPreOrder(root);

  root = deleteNode(root, 3);

  printf("\nAfter deletion: ");
  printPreOrder(root);
  return 0;
}
```

Complexities of Different Operations on an AVL Tree

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

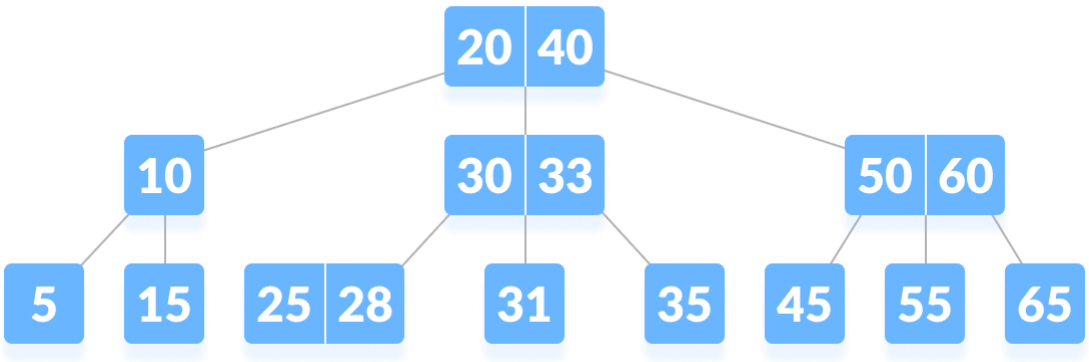
AVL Tree Applications

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

B-tree

In this tutorial, you will learn what a B-tree is. Also, you will find working examples of search operation on a B-tree in C.

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.
It is also known as a height-balanced m-way tree.



B-tree

Why do you need a B-tree data strcuture?

The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B-tree Properties

1. For each node x, the keys are stored in increasing order.
2. In each node, there is a boolean value $x.leaf$ which is true if x is a leaf.
3. If n is the order of the tree, each internal node can contain at most $n - 1$ keys along with a pointer to each child.
4. Each node except root can have at most n children and at least $n/2$ children.
5. All leaves have the same depth (i.e. height-h of the tree).
6. The root has at least 2 children and contains a minimum of 1 key.
7. If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

Operations on a B-tree

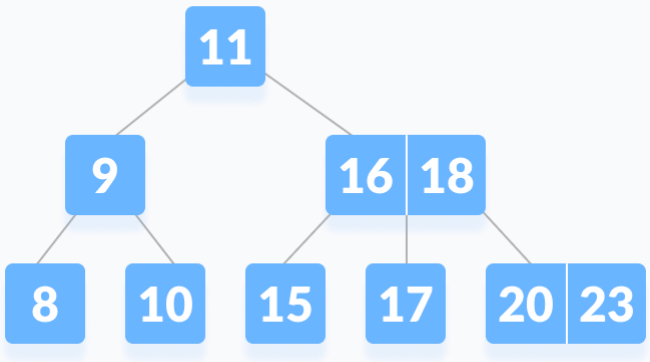
Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

1. Starting from the root node, compare k with the first key of the node.
   If `k = the first key of the node`, return the node and the index.
2. If `k.leaf = true`, return `NULL` (i.e. not found).
3. If `k < the first key of the root node`, search the left child of this key recursively.
4. If there is more than one key in the current node and `k > the first key`, compare k with the next key in the node.
   If `k < next key`, search the left child of this key (ie. k lies in between the first and the second keys).
   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

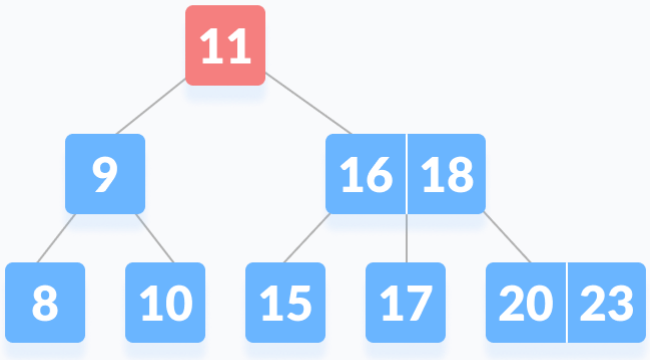
Searching Example

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



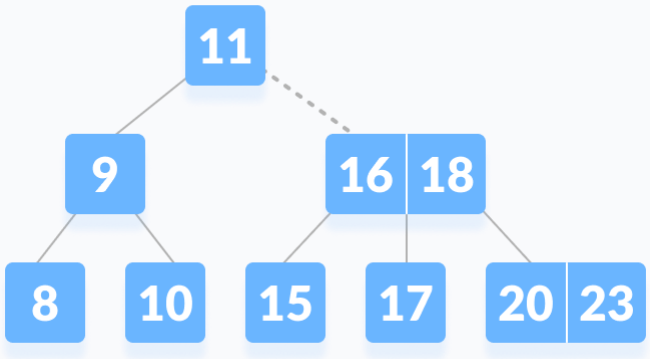
B-tree

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



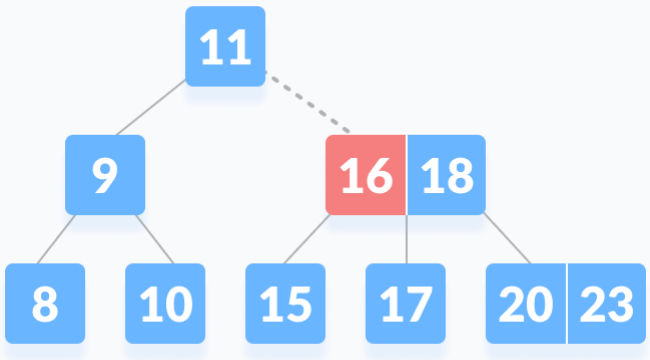
k is not found on the root node

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



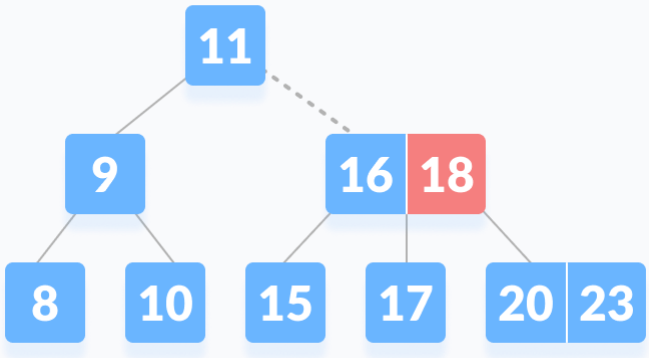
Go to the right subtree

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

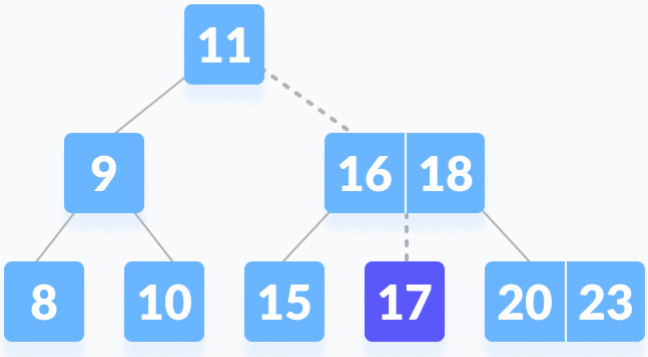


Compare with the keys from left to right

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



the left child of 18. k lies in between 16 and 18



6. k is found. k is found

Algorithm for Searching an Element

```
BtreeSearch(x, k)
 i = 1
 while i ≤ n[x] and k ≥ keyi[x]       // n[x] means number of keys in x node
   do i = i + 1
if i  n[x] and k = keyi[x]
   then return (x, i)
if leaf [x]
   then return NIL
else
```

```
  return BtreeSearch(ci[x], k)
```

B-tree operations code in C

```
// Searching a key on a B-tree in C

#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a node
struct BTreeNode *createNode(int val, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->val[1] = val;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}
```

```
// Insert node
void insertNode(int val, int pos, struct BTreeNode *node,
      struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->val[j + 1] = node->val[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->val[j + 1] = val;
  node->link[j + 1] = child;
  node->count++;
}

// Split node
void splitNode(int val, int *pval, int pos, struct BTreeNode *node,
       struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;

  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;

  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->val[j - median] = node->val[j];
```

```
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}

// Set the value
int setValue(int val, int *pval,
          struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
    *pval = val;
    *child = NULL;
    return 1;
  }

  if (val < node->val[1]) {
    pos = 0;
  } else {
```

```
    for (pos = node->count;
       (val < node->val[pos] && pos > 1); pos--)
      ;
    if (val == node->val[pos]) {
      printf("Duplicates are not permitted\n");
      return 0;
    }
  }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
void insert(int val) {
  int flag, i;
  struct BTreeNode *child;

  flag = setValue(val, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}
```

```
// Search node
void search(int val, int *pos, struct BTreeNode *myNode) {
  if (!myNode) {
    return;
  }

  if (val < myNode->val[1]) {
    *pos = 0;
  } else {
    for (*pos = myNode->count;
       (val < myNode->val[*pos] && *pos > 1); (*pos)--)
      ;
    if (val == myNode->val[*pos]) {
      printf("%d is found", val);
      return;
    }
  }
  search(val, pos, myNode->link[*pos]);

  return;
}

// Traverse then nodes
void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->val[i + 1]);
```

```
    }
    traversal(myNode->link[i]);
  }
}

int main() {
  int val, ch;

  insert(8);
  insert(9);
  insert(10);
  insert(11);
  insert(15);
  insert(16);
  insert(17);
  insert(18);
  insert(20);
  insert(23);

  traversal(root);

  printf("\n");
  search(11, &ch, root);
}
```

Searching Complexity on B Tree

Worst case Time complexity: $\Theta(\log n)$

Average case Time complexity: $\Theta(\log n)$

Best case Time complexity: $\Theta(\log n)$

Average case Space complexity: $\Theta(n)$

Worst case Space complexity: $\Theta(n)$

B Tree Applications

- databases and file systems
- to store blocks of data (secondary storage media)
- multilevel indexing

To learn more about different B-tree operations, please visit

1. Insertion on B-tree
2. Deletion on B-tree

Insertion into a B-tree

In this tutorial, you will learn how to insert a key into a btree. Also, you will find working examples of inserting keys into a B-tree in C.

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

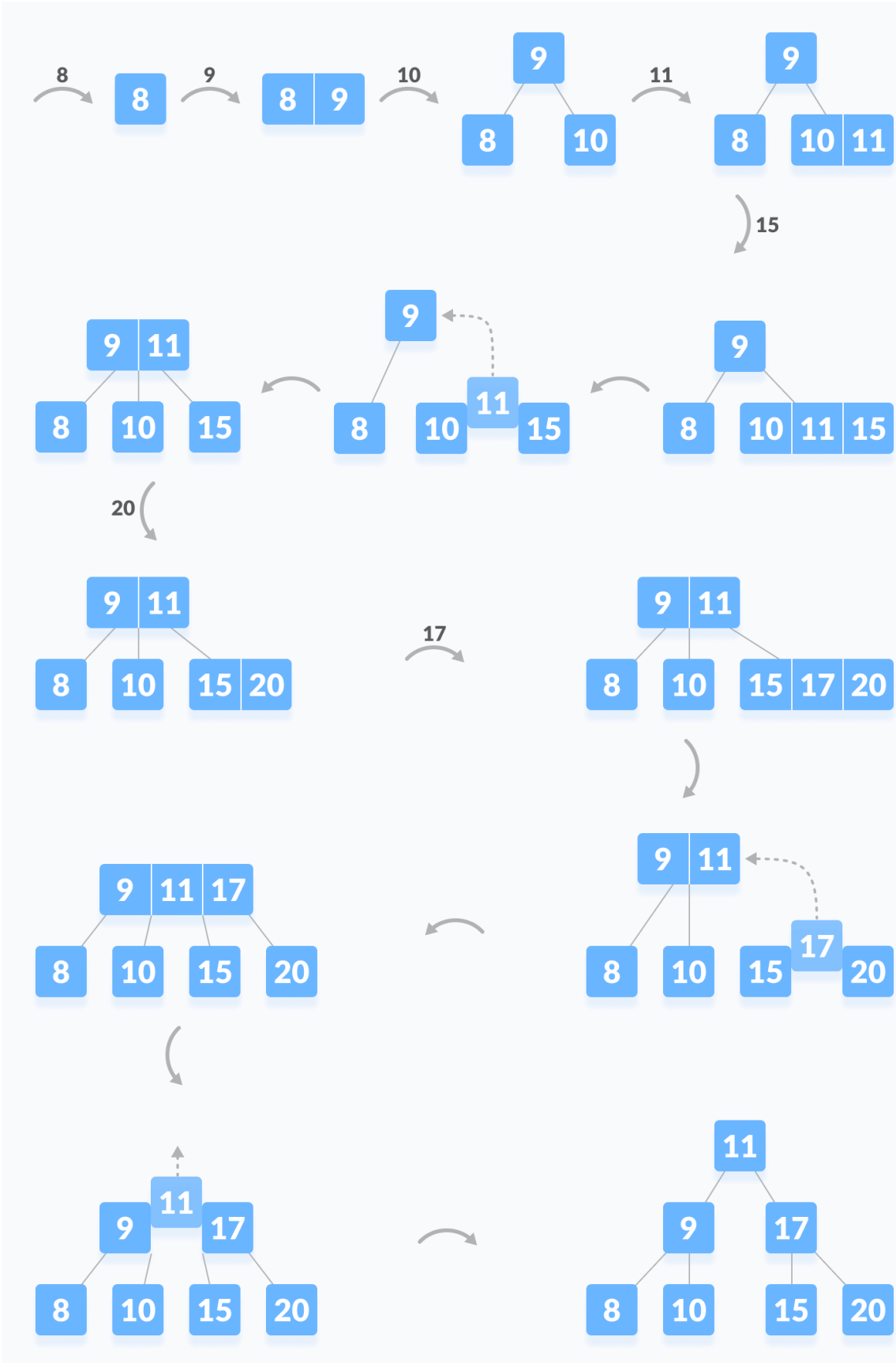
Insertion Operation

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Insertion Example

Let us understand the insertion operation with the illustrations below.

The elements to be inserted are 8, 9, 10, 11, 15, 20, 17.


8
8
9
8 9
10
9
8 10
11
9
8 10 11
15
9 11
8 10 15
9
8 10 11 15
9
8 10 11 15
20
9 11
8 10 15 20
17
9 11
8 10 15 17 20
9 11 17
8 10 15 20
9 11
8 10 15 17 20
11
9 17
8 10 15 20
11
9 17
8 10 15 20

Inserting elements into a B-tree

Algorithm for Inserting an Element

```
BreeInsertion(T, k)
r  root[T]
if n[r] = 2t - 1
   s = AllocateNode()
   root[T] = s
   leaf[s] = FALSE
   n[s] <- 0
   c1[s] <- r
   BtreeSplitChild(s, 1, r)
   BtreeInsertNonFull(s, k)
else BtreeInsertNonFull(r, k)
BtreeInsertNonFull(x, k)
i = n[x]
if leaf[x]
   while i ≥ 1 and k < keyi[x]
      keyi+1 [x] = keyi[x]
      i = i - 1
   keyi+1[x] = k
   n[x] = n[x] + 1
else while i ≥ 1 and k < keyi[x]
      i = i - 1
   i = i + 1
   if n[ci[x]] == 2t - 1
      BtreeSplitChild(x, i, ci[x])
      if k &rt; keyi[x]
         i = i + 1
   BtreeInsertNonFull(ci[x], k)
BtreeSplitChild(x, i)
BtreeSplitChild(x, i, y)
z = AllocateNode()
leaf[z] = leaf[y]
n[z] = t - 1
for j = 1 to t - 1
   keyj[z] = keyj+t[y]
```

```
if not leaf [y]
   for j = 1 to t
      cj[z] = cj + t[y]
n[y] = t - 1
for j = n[x] + 1 to i + 1
   cj+1[x] = cj[x]
ci+1[x] = z
for j = n[x] to i
   keyj+1[x] = keyj[x]
keyi[x] = keyt[y]
n[x] = n[x] + 1
```

C Examples

```
// insertioning a key on a B-tree in C

#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct btreeNode {
  int item[MAX + 1], count;
  struct btreeNode *link[MAX + 1];
};

struct btreeNode *root;

// Node creation
```

```
struct btreeNode *createNode(int item, struct btreeNode *child) {
  struct btreeNode *newNode;
  newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
  newNode->item[1] = item;
  newNode->count = 1;
  newNode->link[0] = root;
  newNode->link[1] = child;
  return newNode;
}

// Insert
void insertValue(int item, int pos, struct btreeNode *node,
         struct btreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->item[j + 1] = node->item[j];
    node->link[j + 1] = node->link[j];
    j--;
  }
  node->item[j + 1] = item;
  node->link[j + 1] = child;
  node->count++;
}

// Split node
void splitNode(int item, int *pval, int pos, struct btreeNode *node,
        struct btreeNode *child, struct btreeNode **newNode) {
  int median, j;
```

```
  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;

  *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->item[j - median] = node->item[j];
    (*newNode)->link[j - median] = node->link[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertValue(item, pos, node, child);
  } else {
    insertValue(item, pos - median, *newNode, child);
  }
  *pval = node->item[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
}

// Set the value of node
int setNodeValue(int item, int *pval,
         struct btreeNode *node, struct btreeNode **child) {
  int pos;
```

```
  if (!node) {
    *pval = item;
    *child = NULL;
    return 1;
  }

  if (item < node->item[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (item < node->item[pos] && pos > 1); pos--)
      ;
    if (item == node->item[pos]) {
      printf("Duplicates not allowed\n");
      return 0;
    }
  }
  if (setNodeValue(item, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertValue(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
```

```
void insertion(int item) {
  int flag, i;
  struct btreeNode *child;

  flag = setNodeValue(item, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Copy the successor
void copySuccessor(struct btreeNode *myNode, int pos) {
  struct btreeNode *dummy;
  dummy = myNode->link[pos];

  for (; dummy->link[0] != NULL;)
    dummy = dummy->link[0];
  myNode->item[pos] = dummy->item[1];
}

// Do rightshift
void rightShift(struct btreeNode *myNode, int pos) {
  struct btreeNode *x = myNode->link[pos];
  int j = x->count;

  while (j > 0) {
    x->item[j + 1] = x->item[j];
    x->link[j + 1] = x->link[j];
  }
  x->item[1] = myNode->item[pos];
```

```
  x->link[1] = x->link[0];
  x->count++;

  x = myNode->link[pos - 1];
  myNode->item[pos] = x->item[x->count];
  myNode->link[pos] = x->link[x->count];
  x->count--;
  return;
}

// Do leftshift
void leftShift(struct btreeNode *myNode, int pos) {
  int j = 1;
  struct btreeNode *x = myNode->link[pos - 1];

  x->count++;
  x->item[x->count] = myNode->item[pos];
  x->link[x->count] = myNode->link[pos]->link[0];

  x = myNode->link[pos];
  myNode->item[pos] = x->item[1];
  x->link[0] = x->link[1];
  x->count--;

  while (j <= x->count) {
    x->item[j] = x->item[j + 1];
    x->link[j] = x->link[j + 1];
    j++;
  }
```

```
  return;
}

// Merge the nodes
void mergeNodes(struct btreeNode *myNode, int pos) {
  int j = 1;
  struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];

  x2->count++;
  x2->item[x2->count] = myNode->item[pos];
  x2->link[x2->count] = myNode->link[0];

  while (j <= x1->count) {
    x2->count++;
    x2->item[x2->count] = x1->item[j];
    x2->link[x2->count] = x1->link[j];
    j++;
  }

  j = pos;
  while (j < myNode->count) {
    myNode->item[j] = myNode->item[j + 1];
    myNode->link[j] = myNode->link[j + 1];
    j++;
  }
  myNode->count--;
  free(x1);
}
```

```
// Adjust the node
void adjustNode(struct btreeNode *myNode, int pos) {
  if (!pos) {
    if (myNode->link[1]->count > MIN) {
      leftShift(myNode, 1);
    } else {
      mergeNodes(myNode, 1);
    }
  } else {
    if (myNode->count != pos) {
      if (myNode->link[pos - 1]->count > MIN) {
        rightShift(myNode, pos);
      } else {
        if (myNode->link[pos + 1]->count > MIN) {
          leftShift(myNode, pos + 1);
        } else {
          mergeNodes(myNode, pos);
        }
      }
    } else {
      if (myNode->link[pos - 1]->count > MIN)
        rightShift(myNode, pos);
      else
        mergeNodes(myNode, pos);
    }
  }
}

// Traverse the tree
```

```
void traversal(struct btreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->link[i]);
      printf("%d ", myNode->item[i + 1]);
    }
    traversal(myNode->link[i]);
  }
}

int main() {
  int item, ch;

  insertion(8);
  insertion(9);
  insertion(10);
  insertion(11);
  insertion(15);
  insertion(16);
  insertion(17);
  insertion(18);
  insertion(20);
  insertion(23);

  traversal(root);
}
```

Deletion from a B-tree

In this tutorial, you will learn how to delete a key from a b-tree. Also, you will find working examples of deleting keys from a B-tree in C.

Deleting an element on a B-tree consists of three main events: **searching the node where the key to be deleted exists**, deleting the key and balancing the tree if required.
While deleting a tree, a condition called **underflow** may occur. Underflow occurs when a node contains less than the minimum number of keys it should hold.
The terms to be understood before studying deletion operation are:

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

Deletion Operation

Before going through the steps below, one must know these facts about a B tree of degree **m**.

1. A node can have a maximum of m children. (i.e. 3)
2. A node can contain a maximum of $m - 1$ keys. (i.e. 2)
3. A node should have a minimum of $\lceil m/2 \rceil$ children. (i.e. 2)

4. A node (except root node) should contain a minimum of $\lceil m/2 \rceil - 1$ keys. (i.e. 1)

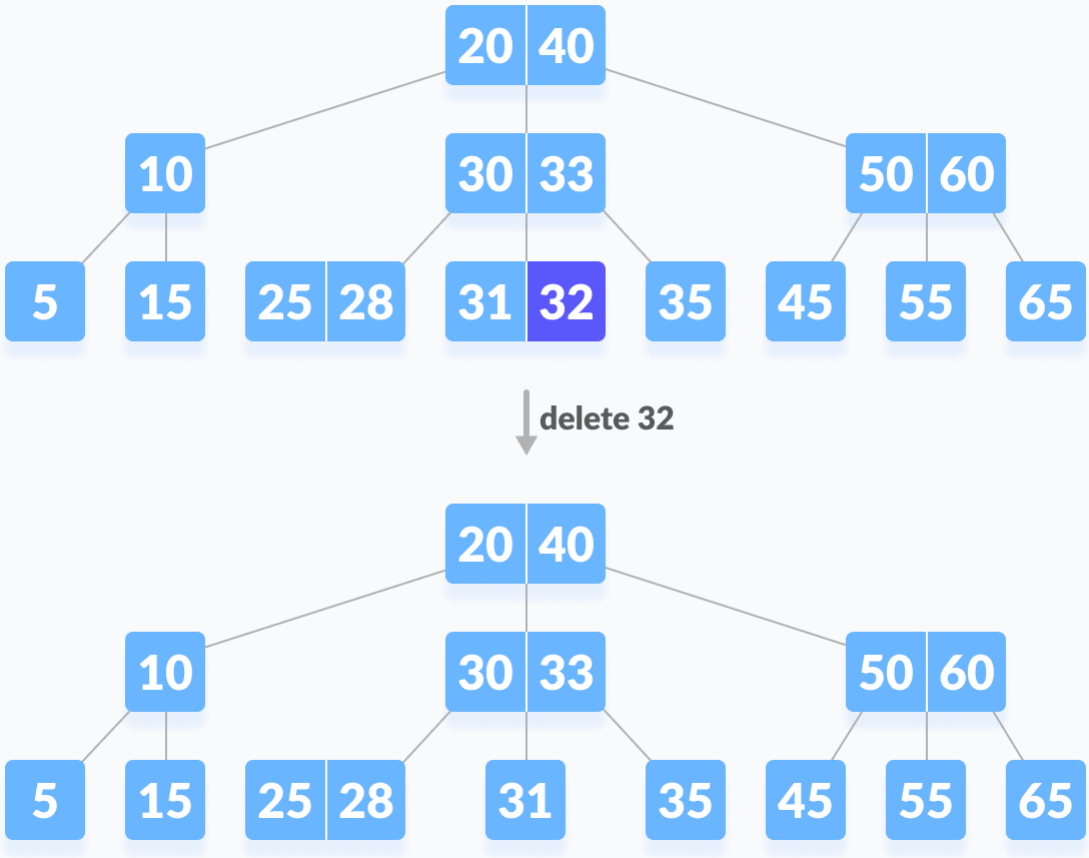
There are three main cases for deletion operation in a B tree.

Case I

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

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

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



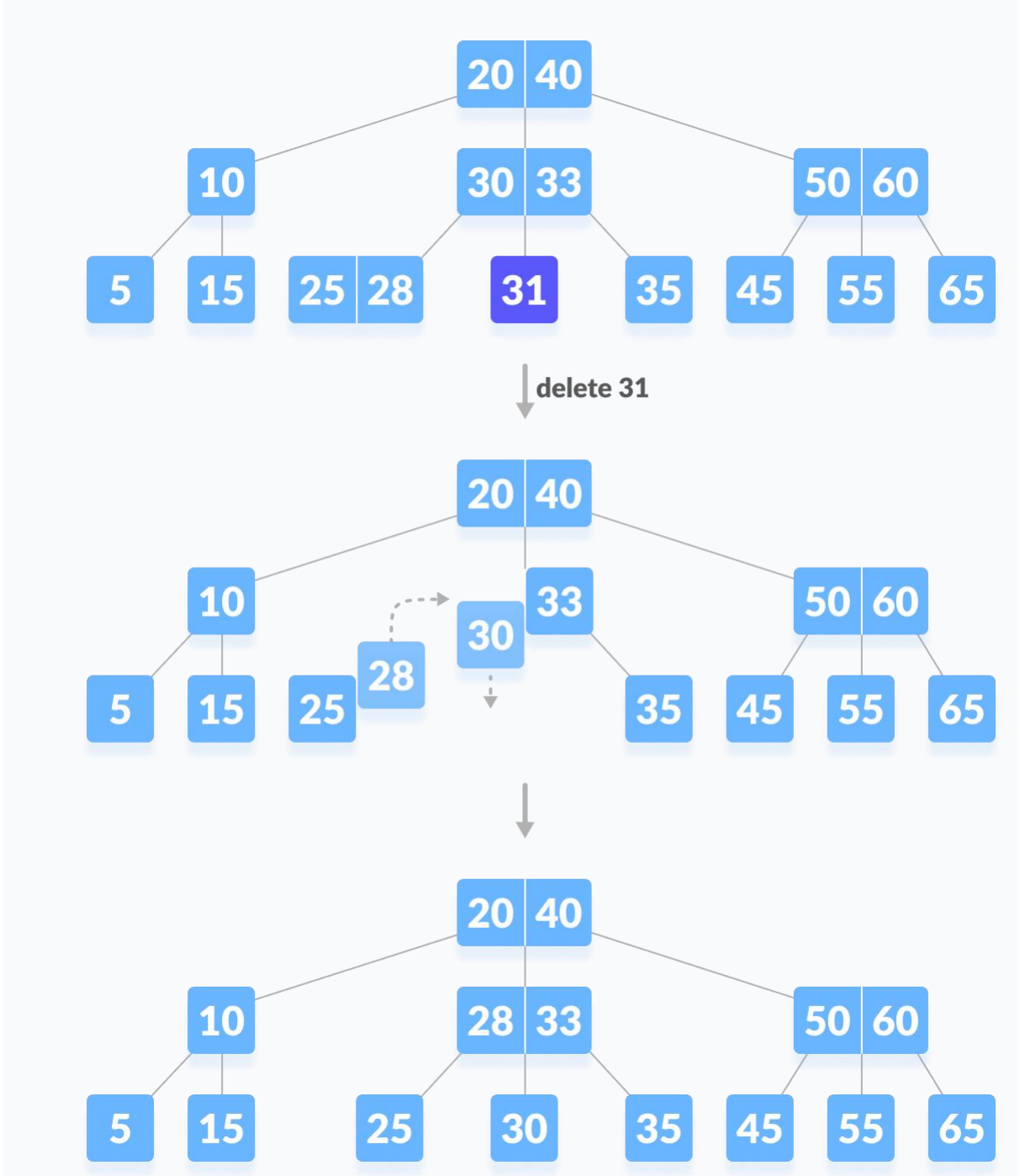
Deleting a leaf key (32) from B-tree

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

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

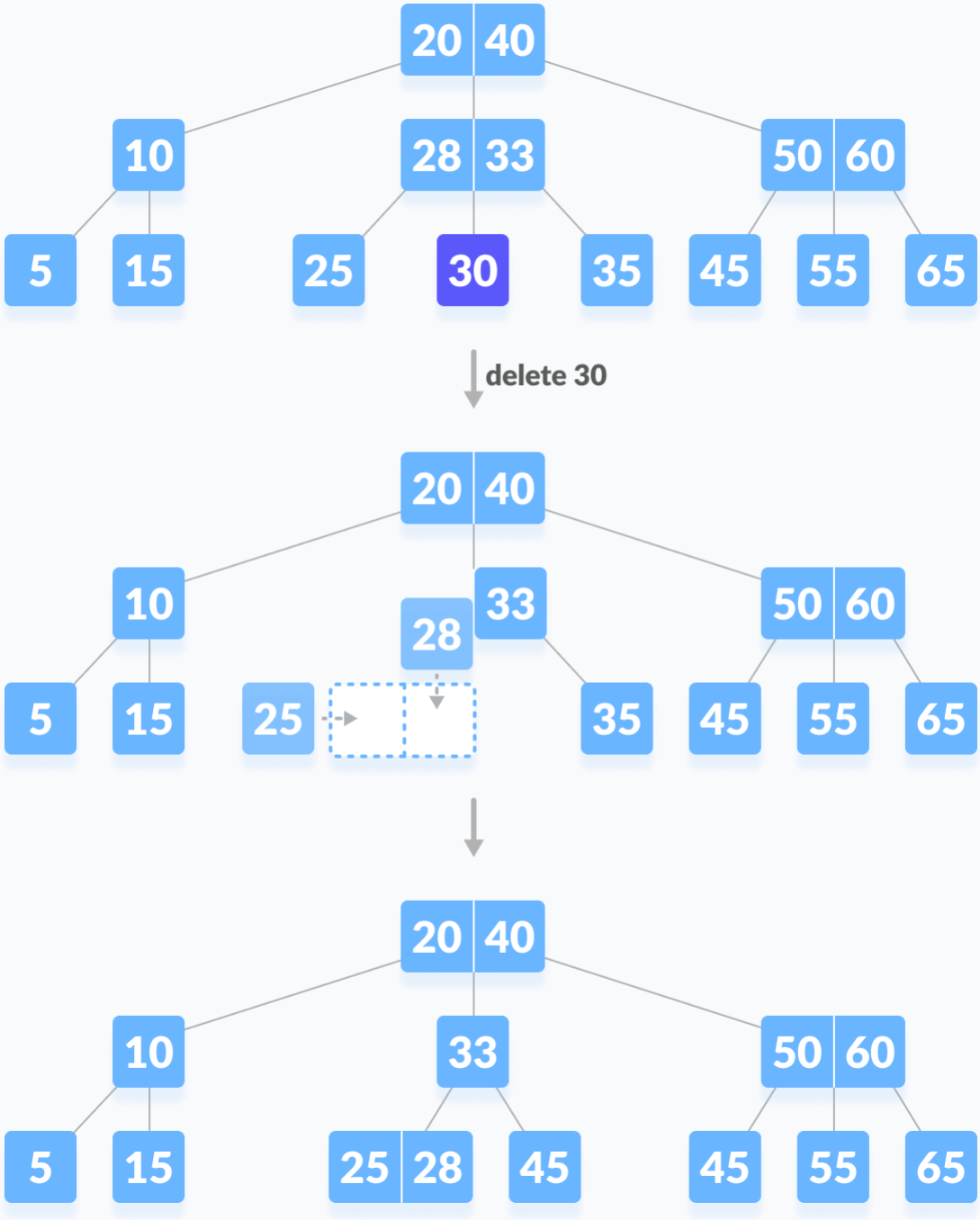
   Else, check to borrow from the immediate right sibling node.

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



Deleting a leaf key (31)If both the immediate sibling nodes already have a minimum number of keys, then merge the node with either the left sibling node or the right sibling node. **This merging is done through the parent node.**

Deleting 30 results in the above case.

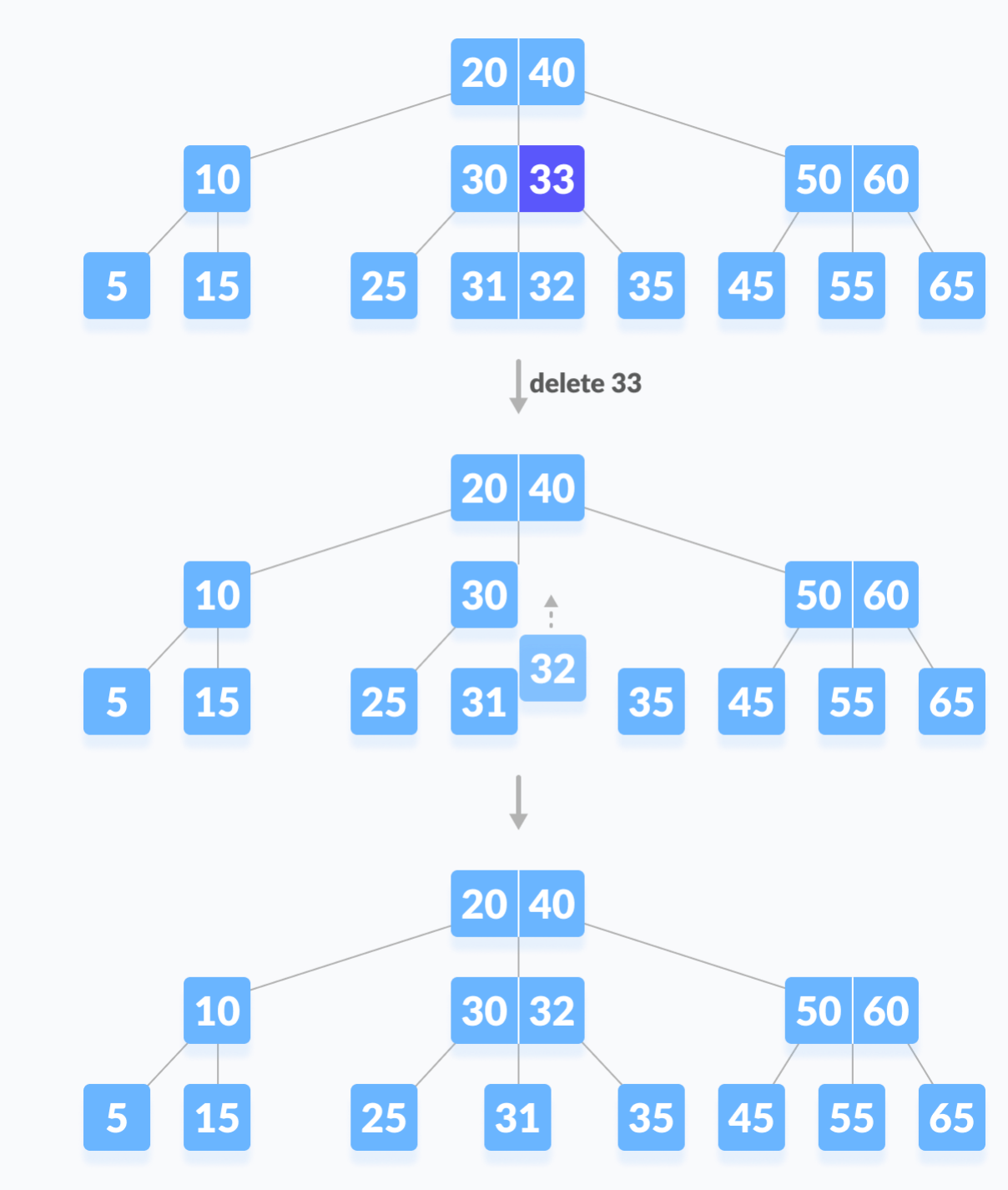


Delete a leaf key (30)

Case II

If the key to be deleted lies in the internal node, the following cases occur.

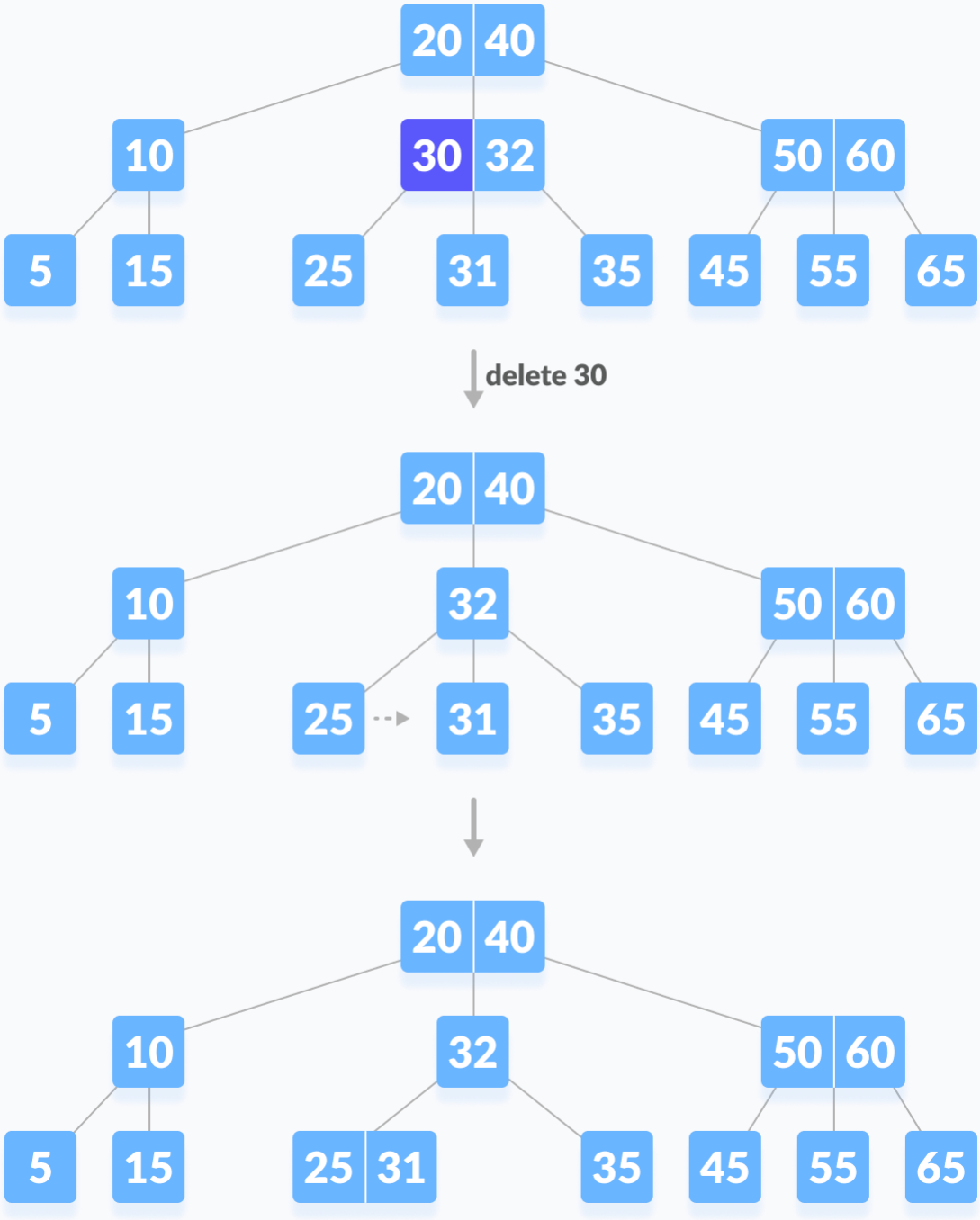
1. The internal node, which is deleted, is replaced by an inorder predecessor if the left child has more than the minimum number of keys.



Deleting an internal node (33)

2. The internal node, which is deleted, is replaced by an inorder successor if the right child has more than the minimum number of keys.

3. If either child has exactly a minimum number of keys then, merge the left and the right children.



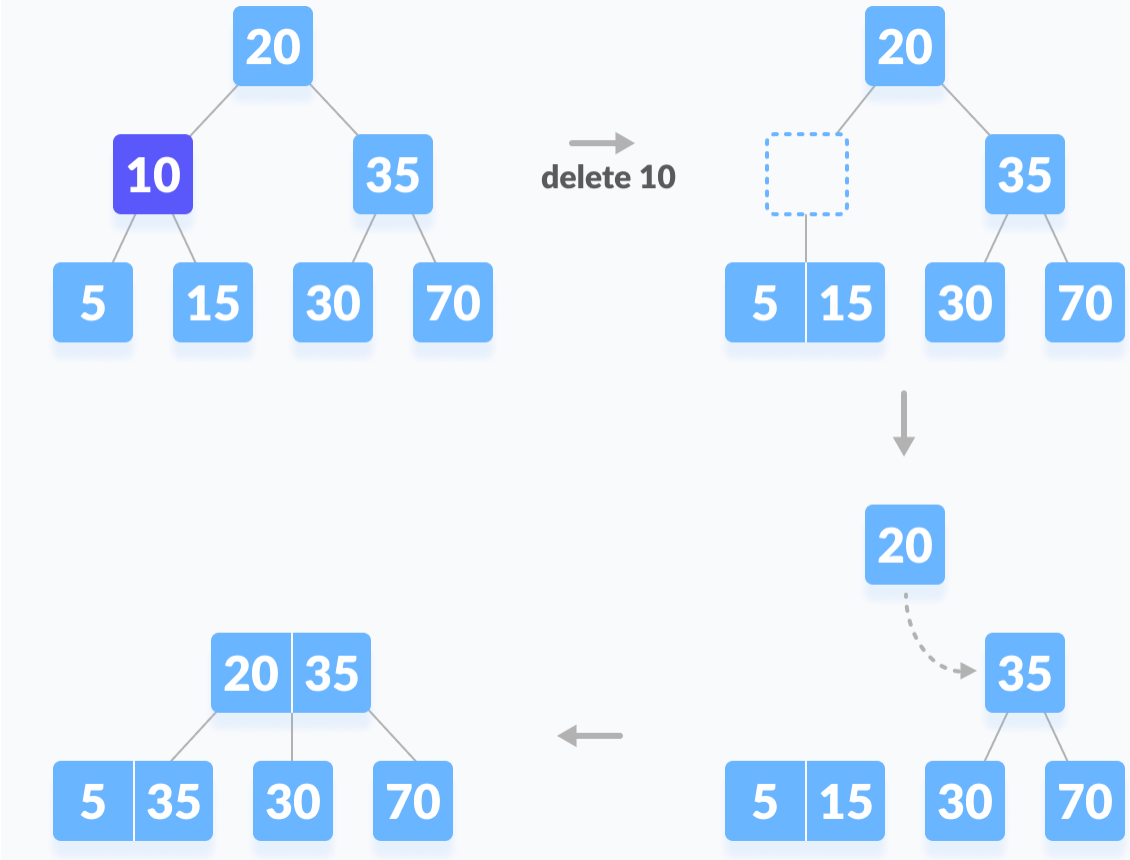
Deleting an internal node (30)After merging if the parent node has less

than the minimum number of keys then, look for the siblings as in Case I.

Case III

In this case, the height of the tree shrinks. If the target key lies in an internal node, and the deletion of the key leads to a fewer number of keys in the node (i.e. less than the minimum required), then look for the inorder predecessor and the inorder successor. If both the children contain a minimum number of keys then, borrowing cannot take place. This leads to Case II(3) i.e. merging the children.

Again, look for the sibling to borrow a key. But, if the sibling also has only a minimum number of keys then, merge the node with the sibling along with the parent. Arrange the children accordingly (increasing order).



Deleting an internal node (10)

C Examples

```
// Deleting a key from a B-tree in C


#include <stdio.h>

#include <stdlib.h>


#define MAX 3

#define MIN 2
```

```
struct BTreeNode {
  int item[MAX + 1], count;
  struct BTreeNode *linker[MAX + 1];
};

struct BTreeNode *root;

// Node creation
struct BTreeNode *createNode(int item, struct BTreeNode *child) {
  struct BTreeNode *newNode;
  newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  newNode->item[1] = item;
  newNode->count = 1;
  newNode->linker[0] = root;
  newNode->linker[1] = child;
  return newNode;
}

// Add value to the node
void addValToNode(int item, int pos, struct BTreeNode *node,
        struct BTreeNode *child) {
  int j = node->count;
  while (j > pos) {
    node->item[j + 1] = node->item[j];
    node->linker[j + 1] = node->linker[j];
    j--;
  }
  node->item[j + 1] = item;
  node->linker[j + 1] = child;
```

```
  node->count++;
}

// Split the node
void splitNode(int item, int *pval, int pos, struct BTreeNode *node,
        struct BTreeNode *child, struct BTreeNode **newNode) {
  int median, j;

  if (pos > MIN)
    median = MIN + 1;
  else
    median = MIN;

  *newNode = (struct BTreeNode *)malloc(sizeof(struct BTreeNode));
  j = median + 1;
  while (j <= MAX) {
    (*newNode)->item[j - median] = node->item[j];
    (*newNode)->linker[j - median] = node->linker[j];
    j++;
  }
  node->count = median;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    addValToNode(item, pos, node, child);
  } else {
    addValToNode(item, pos - median, *newNode, child);
  }
  *pval = node->item[node->count];
```

```
  (*newNode)->linker[0] = node->linker[node->count];
  node->count--;
}

// Set the value in the node
int setValueInNode(int item, int *pval,
          struct BTreeNode *node, struct BTreeNode **child) {
  int pos;
  if (!node) {
    *pval = item;
    *child = NULL;
    return 1;
  }

  if (item < node->item[1]) {
    pos = 0;
  } else {
    for (pos = node->count;
       (item < node->item[pos] && pos > 1); pos--)
      ;
    if (item == node->item[pos]) {
      printf("Duplicates not allowed\n");
      return 0;
    }
  }
  if (setValueInNode(item, pval, node->linker[pos], child)) {
    if (node->count < MAX) {
      addValToNode(*pval, pos, node, *child);
    } else {
```

```
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insertion operation
void insertion(int item) {
  int flag, i;
  struct BTreeNode *child;

  flag = setValueInNode(item, &i, root, &child);
  if (flag)
    root = createNode(i, child);
}

// Copy the successor
void copySuccessor(struct BTreeNode *myNode, int pos) {
  struct BTreeNode *dummy;
  dummy = myNode->linker[pos];

  for (; dummy->linker[0] != NULL;)
    dummy = dummy->linker[0];
  myNode->item[pos] = dummy->item[1];
}

// Remove the value
void removeVal(struct BTreeNode *myNode, int pos) {
```

```
  int i = pos + 1;
  while (i <= myNode->count) {
    myNode->item[i - 1] = myNode->item[i];
    myNode->linker[i - 1] = myNode->linker[i];
    i++;
  }
  myNode->count--;
}

// Do right shift
void rightShift(struct BTreeNode *myNode, int pos) {
  struct BTreeNode *x = myNode->linker[pos];
  int j = x->count;

  while (j > 0) {
    x->item[j + 1] = x->item[j];
    x->linker[j + 1] = x->linker[j];
  }
  x->item[1] = myNode->item[pos];
  x->linker[1] = x->linker[0];
  x->count++;

  x = myNode->linker[pos - 1];
  myNode->item[pos] = x->item[x->count];
  myNode->linker[pos] = x->linker[x->count];
  x->count--;
  return;
}
```

```
// Do left shift
void leftShift(struct BTreeNode *myNode, int pos) {
  int j = 1;
  struct BTreeNode *x = myNode->linker[pos - 1];

  x->count++;
  x->item[x->count] = myNode->item[pos];
  x->linker[x->count] = myNode->linker[pos]->linker[0];

  x = myNode->linker[pos];
  myNode->item[pos] = x->item[1];
  x->linker[0] = x->linker[1];
  x->count--;

  while (j <= x->count) {
    x->item[j] = x->item[j + 1];
    x->linker[j] = x->linker[j + 1];
    j++;
  }
  return;
}

// Merge the nodes
void mergeNodes(struct BTreeNode *myNode, int pos) {
  int j = 1;
  struct BTreeNode *x1 = myNode->linker[pos], *x2 = myNode->linker[pos - 1];

  x2->count++;
  x2->item[x2->count] = myNode->item[pos];
```

```
  x2->linker[x2->count] = myNode->linker[0];

  while (j <= x1->count) {
    x2->count++;
    x2->item[x2->count] = x1->item[j];
    x2->linker[x2->count] = x1->linker[j];
    j++;
  }

  j = pos;
  while (j < myNode->count) {
    myNode->item[j] = myNode->item[j + 1];
    myNode->linker[j] = myNode->linker[j + 1];
    j++;
  }
  myNode->count--;
  free(x1);
}

// Adjust the node
void adjustNode(struct BTreeNode *myNode, int pos) {
  if (!pos) {
    if (myNode->linker[1]->count > MIN) {
      leftShift(myNode, 1);
    } else {
      mergeNodes(myNode, 1);
    }
  } else {
    if (myNode->count != pos) {
```

```
      if (myNode->linker[pos - 1]->count > MIN) {
        rightShift(myNode, pos);
      } else {
        if (myNode->linker[pos + 1]->count > MIN) {
          leftShift(myNode, pos + 1);
        } else {
          mergeNodes(myNode, pos);
        }
      }
    } else {
      if (myNode->linker[pos - 1]->count > MIN)
        rightShift(myNode, pos);
      else
        mergeNodes(myNode, pos);
    }
  }
}

// Delete a value from the node
int delValFromNode(int item, struct BTreeNode *myNode) {
  int pos, flag = 0;
  if (myNode) {
    if (item < myNode->item[1]) {
      pos = 0;
      flag = 0;
    } else {
      for (pos = myNode->count; (item < myNode->item[pos] && pos > 1); pos--)
        ;
      if (item == myNode->item[pos]) {
```

```
      flag = 1;
    } else {
      flag = 0;
    }
  }
  if (flag) {
    if (myNode->linker[pos - 1]) {
      copySuccessor(myNode, pos);
      flag = delValFromNode(myNode->item[pos], myNode->linker[pos]);
      if (flag == 0) {
        printf("Given data is not present in B-Tree\n");
      }
    } else {
      removeVal(myNode, pos);
    }
  } else {
    flag = delValFromNode(item, myNode->linker[pos]);
  }
  if (myNode->linker[pos]) {
    if (myNode->linker[pos]->count < MIN)
      adjustNode(myNode, pos);
  }
 }
 return flag;
}

// Delete operaiton
void delete (int item, struct BTreeNode *myNode) {
 struct BTreeNode *tmp;
```

```
  if (!delValFromNode(item, myNode)) {
    printf("Not present\n");
    return;
  } else {
    if (myNode->count == 0) {
      tmp = myNode;
      myNode = myNode->linker[0];
      free(tmp);
    }
  }
  root = myNode;
  return;
}

void searching(int item, int *pos, struct BTreeNode *myNode) {
  if (!myNode) {
    return;
  }

  if (item < myNode->item[1]) {
    *pos = 0;
  } else {
    for (*pos = myNode->count;
       (item < myNode->item[*pos] && *pos > 1); (*pos)--)
      ;
    if (item == myNode->item[*pos]) {
      printf("%d present in B-tree", item);
      return;
    }
```

```
  }
  searching(item, pos, myNode->linker[*pos]);
  return;
}

void traversal(struct BTreeNode *myNode) {
  int i;
  if (myNode) {
    for (i = 0; i < myNode->count; i++) {
      traversal(myNode->linker[i]);
      printf("%d ", myNode->item[i + 1]);
    }
    traversal(myNode->linker[i]);
  }
}

int main() {
  int item, ch;

  insertion(8);
  insertion(9);
  insertion(10);
  insertion(11);
  insertion(15);
  insertion(16);
  insertion(17);
  insertion(18);
  insertion(20);
  insertion(23);
```

```
  traversal(root);


  delete (20, root);
  printf("\n");
  traversal(root);
}
```

Deletion Complexity

Best case Time complexity: $\Theta(\log n)$

Average case Space complexity: $\Theta(n)$

Worst case Space complexity: $\Theta(n)$

Red-Black Tree

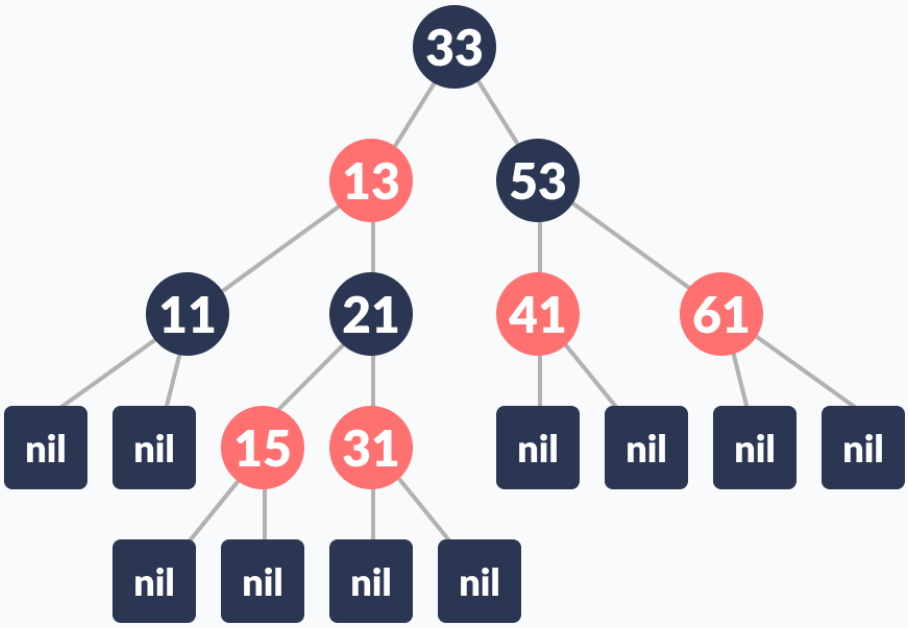
In this tutorial, you will learn what a red-black tree is. Also, you will find working examples of various operations performed on a red-black tree in C, C++, Java and Python.

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

An example of a red-black tree is:



Red Black Tree

Each node has the following attributes:

- color
- key
- leftChild
- rightChild
- parent (except root node)

How the red-black tree maintains the property of self-balancing?

The red-black color is meant for balancing the tree.

The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

Operations on a Red-Black Tree

Various operations that can be performed on a red-black tree are:

Rotating the subtrees in a Red-Black Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

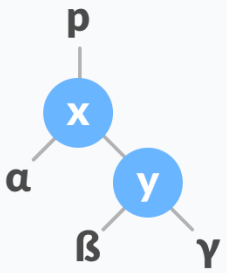
Rotation operation is used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

There are two types of rotations:

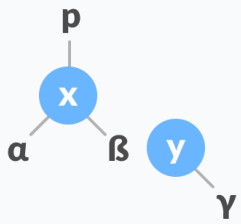
Left Rotate

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

Algorithm

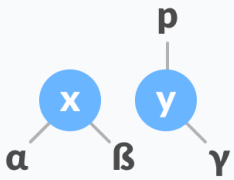


1. Let the initial tree be: Initial tree
2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.

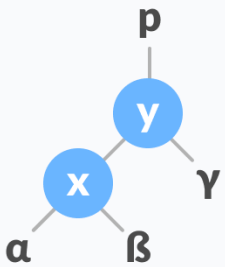


Assign x as the parent of the left subtree of y

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



5. Else assign `y` as the right child of `p`. Change the parent of x to that of y

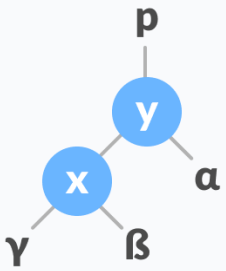


6. Make `y` as the parent of `x`. Assign y as the parent of x.

Right Rotate

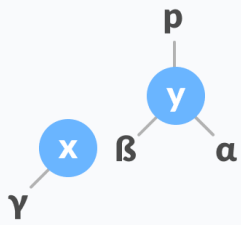
In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

1. Let the initial tree be:



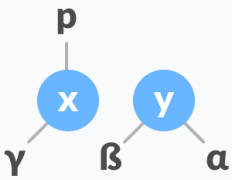
Initial Tree

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



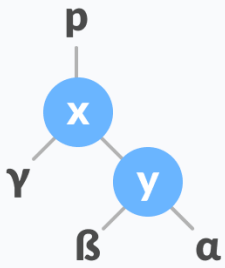
Assign y as the parent of the right subtree of x

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



5. Else assign `x` as the left child of `p`.

Assign the parent of y as the parent of x

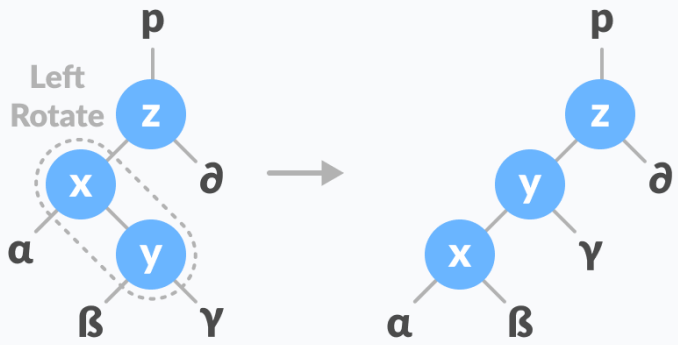


6. Make `x` as the parent of `y`.

Assign x as the parent of y

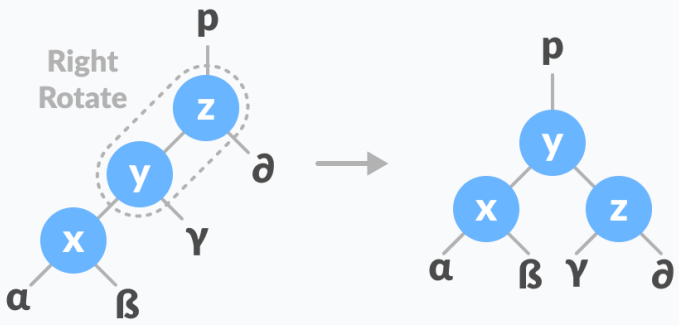
Left-Right and Right-Left Rotate

In left-right rotation, the arrangements are first shifted to the left and then to the right.



1. Do left rotation on x-y.

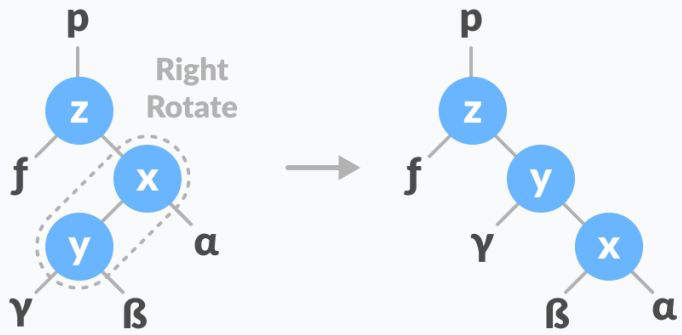
   Left rotate x-y



2. Do right rotation on y-z.

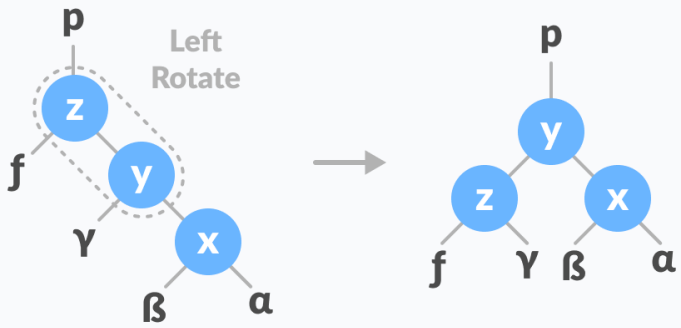
   Right rotate z-y

In right-left rotation, the arrangements are first shifted to the right and then to the left.



1. Do right rotation on x-y.

   Right rotate x-y



2. Do left rotation on z-y.

   Left rotate z-y

Inserting an element into a Red-Black Tree

While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

1. Recolor
2. Rotation

Algorithm to insert a node

Following steps are followed for inserting a new element into a red-black tree:

1. Let y be the leaf (ie. `NIL`) and x be the root of the tree.
2. Check if the tree is empty (ie. whether `x` is `NIL`). If yes, insert `newNode` as a root node and color it black.
3. Else, repeat steps following steps until leaf (`NIL`) is reached.
    a. Compare `newKey` with `rootKey`.
    b. If `newKey` is greater than rootKey, traverse through the right subtree.
    c. Else traverse through the left subtree.
4. Assign the parent of the leaf as a parent of `newNode`.
5. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.
6. Else, make `newNode` as `leftChild`.
7. Assign `NULL` to the left and `rightChild` of `newNode`.
8. Assign RED color to `newNode`.
9. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?
This is because inserting a red node does not violate the depth property of a red-black tree.

If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

Algorithm to maintain red-black property after insertion

This algorithm is used for maintaining the property of a red-black tree if the insertion of a newNode violates this property.

1. Do the following while the parent of `newNode` `p` is RED.
2. If `p` is the left child of `grandParent` `gP` of `z`, do the following.

   Case-I:

   a. If the color of the right child of `gP` of `z` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.
   b. Assign `gP` to `newNode`.

   Case-II:

   c. Else if `newNode` is the right child of `p` then, assign `p` to `newNode`.
   d. Left-Rotate `newNode`.

   Case-III:

   e. Set color of `p` as BLACK and color of `gP` as RED.
   f. Right-Rotate `gP`.
3. Else, do the following.

   a. If the color of the left child of `gP` of `z` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.
   b. Assign `gP` to `newNode`.
   c. Else if newNode is the left child of `p` then, assign `p` to `newNode` and Right-Rotate `newNode`.
   d. Set color of `p` as BLACK and color of `gP` as RED.
   e. Left-Rotate `gP`.
4. Set the root of the tree as BLACK.

Deleting an element from a Red-Black Tree

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.

Algorithm to delete a node

1. Save the color of `nodeToBeDeleted` in `origrinalColor`.
2. If the left child of `nodeToBeDeleted` is `NULL`
    a. Assign the right child of `nodeToBeDeleted` to x.
    b. Transplant `nodeToBeDeleted` with `x`.
3. Else if the right child of `nodeToBeDeleted` is `NULL`
    a. Assign the left child of `nodeToBeDeleted` into `x`.
    b. Transplant `nodeToBeDeleted` with `x`.
4. Else
    a. Assign the minimum of right subtree of `noteToBeDeleted` into `y`.
    b. Save the color of `y` in `originalColor`.
    c. Assign the `rightChild` of `y` into `x`.
    d. If `y` is a child of `nodeToBeDeleted`, then set the parent of `x` as `y`.
    e. Else, transplant `y` with `rightChild` of `y`.
    f. Transplant `nodeToBeDeleted` with `y`.
    g. Set the color of y with originalColor.
5. If the `originalColor` is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node `x` (which is occupying `y`'s original position) has an extra black. This makes node `x` neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.
However, the color attribute of `x` is not changed rather the extra black is represented in `x`'s pointing to the node.
The extra black can be removed if

1. It reaches the root node.
2. If `x` points to a red-black node. In this case, `x` is colored black.
3. Suitable rotations and recoloring are performed.

The following algorithm retains the properties of a red-black tree.

1. Do the following until the `x` is not the root of the tree and the color of `x` is BLACK
2. If `x` is the left child of its parent then,
    a. Assign w to the sibling of x.
    b. If the right child of parent of `x` is RED,
    Case-I:
        a. Set the color of the right child of the parent of `x` as BLACK.
        b. Set the color of the parent of `x` as RED.
        c. Left-Rotate the parent of `x`.
        d. Assign the `rightChild` of the parent of `x` to `w`.

c. If the color of both the right and the `leftChild` of `w` is BLACK,

Case-II:

a. Set the color of `w` as RED
b. Assign the parent of `x` to `x`.

d. Else if the color of the `rightChild` of `w` is BLACK

Case-III:

a. Set the color of the `leftChild` of `w` as BLACK
b. Set the color of `w` as RED
c. Right-Rotate `w`.
d. Assign the `rightChild` of the parent of `x` to `w`.

e. If any of the above cases do not occur, then do the following.

Case-IV:

a. Set the color of `w` as the color of the parent of `x`.
b. Set the color of the parent of `x` as BLACK.
c. Set the color of the right child of `w` as BLACK.
d. Left-Rotate the parent of `x`.
e. Set `x` as the root of the tree.

3. Else the same as above with right changed to left and vice versa.
4. Set the color of `x` as BLACK.

Please refer to insertion and deletion operations for more explanation with examples.

C Examples

```
// Implementing Red-Black Tree in C

#include <stdio.h>
#include <stdlib.h>

enum nodeColor {
  RED,
  BLACK
};

struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}
```

```
// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
      return;
    }
    index = (data - ptr->data) > 0 ? 1 : 0;
    stack[ht] = ptr;
    ptr = ptr->link[index];
    dir[ht++] = index;
  }
  stack[ht - 1]->link[index] = newnode = createNode(data);
  while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
    if (dir[ht - 2] == 0) {
      yPtr = stack[ht - 2]->link[1];
      if (yPtr != NULL && yPtr->color == RED) {
        stack[ht - 2]->color = RED;
        stack[ht - 1]->color = yPtr->color = BLACK;
```

```
    ht = ht - 2;
  } else {
    if (dir[ht - 1] == 0) {
      yPtr = stack[ht - 1];
    } else {
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[1];
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      stack[ht - 2]->link[0] = yPtr;
    }
    xPtr = stack[ht - 2];
    xPtr->color = RED;
    yPtr->color = BLACK;
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = xPtr;
    if (xPtr == root) {
      root = yPtr;
    } else {
      stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
  }
} else {
  yPtr = stack[ht - 2]->link[0];
  if ((yPtr != NULL) && (yPtr->color == RED)) {
    stack[ht - 2]->color = RED;
    stack[ht - 1]->color = yPtr->color = BLACK;
    ht = ht - 2;
```

```
      } else {
        if (dir[ht - 1] == 1) {
          yPtr = stack[ht - 1];
        } else {
          xPtr = stack[ht - 1];
          yPtr = xPtr->link[0];
          xPtr->link[0] = yPtr->link[1];
          yPtr->link[1] = xPtr;
          stack[ht - 2]->link[1] = yPtr;
        }
        xPtr = stack[ht - 2];
        yPtr->color = BLACK;
        xPtr->color = RED;
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
```

```
struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
struct rbNode *pPtr, *qPtr, *rPtr;
int dir[98], ht = 0, diff, i;
enum nodeColor color;

if (!root) {
  printf("Tree not available\n");
  return;
}

ptr = root;
while (ptr != NULL) {
  if ((data - ptr->data) == 0)
    break;
  diff = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  dir[ht++] = diff;
  ptr = ptr->link[diff];
}

if (ptr->link[1] == NULL) {
  if ((ptr == root) && (ptr->link[0] == NULL)) {
    free(ptr);
    root = NULL;
  } else if (ptr == root) {
    root = ptr->link[0];
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
```

```
  }
 } else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
   xPtr->link[0] = ptr->link[0];
   color = xPtr->color;
   xPtr->color = ptr->color;
   ptr->color = color;

   if (ptr == root) {
    root = xPtr;
   } else {
    stack[ht - 1]->link[dir[ht - 1]] = xPtr;
   }

   dir[ht] = 1;
   stack[ht++] = xPtr;
  } else {
   i = ht++;
   while (1) {
    dir[ht] = 0;
    stack[ht++] = xPtr;
    yPtr = xPtr->link[0];
    if (!yPtr->link[0])
     break;
    xPtr = yPtr;
   }

   dir[i] = 1;
```

```
    stack[i] = yPtr;
    if (i > 0)
      stack[i - 1]->link[dir[i - 1]] = yPtr;

    yPtr->link[0] = ptr->link[0];

    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = ptr->link[1];

    if (ptr == root) {
      root = yPtr;
    }

    color = yPtr->color;
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}

if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
      break;
    }
```

```
if (ht < 2)
  break;

if (dir[ht - 2] == 0) {
  rPtr = stack[ht - 1]->link[1];

  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[1] = rPtr->link[0];
    rPtr->link[0] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 0;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;

    rPtr = stack[ht - 1]->link[1];
  }
```

```
  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
   (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
   rPtr->color = RED;
  } else {
   if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
    qPtr = rPtr->link[0];
    rPtr->color = RED;
    qPtr->color = BLACK;
    rPtr->link[0] = qPtr->link[1];
    qPtr->link[1] = rPtr;
    rPtr = stack[ht - 1]->link[1] = qPtr;
   }
   rPtr->color = stack[ht - 1]->color;
   stack[ht - 1]->color = BLACK;
   rPtr->link[1]->color = BLACK;
   stack[ht - 1]->link[1] = rPtr->link[0];
   rPtr->link[0] = stack[ht - 1];
   if (stack[ht - 1] == root) {
    root = rPtr;
   } else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
   }
   break;
  }
 } else {
  rPtr = stack[ht - 1]->link[0];
  if (!rPtr)
   break;
```

```
if (rPtr->color == RED) {
  stack[ht - 1]->color = RED;
  rPtr->color = BLACK;
  stack[ht - 1]->link[0] = rPtr->link[1];
  rPtr->link[1] = stack[ht - 1];

  if (stack[ht - 1] == root) {
    root = rPtr;
  } else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
  }
  dir[ht] = 1;
  stack[ht] = stack[ht - 1];
  stack[ht - 1] = rPtr;
  ht++;

  rPtr = stack[ht - 1]->link[0];
}
if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
  (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
  rPtr->color = RED;
} else {
  if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
    qPtr = rPtr->link[1];
    rPtr->color = RED;
    qPtr->color = BLACK;
    rPtr->link[1] = qPtr->link[0];
    qPtr->link[0] = rPtr;
    rPtr = stack[ht - 1]->link[0] = qPtr;
```

```
        }
        rPtr->color = stack[ht - 1]->color;
        stack[ht - 1]->color = BLACK;
        rPtr->link[0]->color = BLACK;
        stack[ht - 1]->link[0] = rPtr->link[1];
        rPtr->link[1] = stack[ht - 1];
        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        break;
       }
      }
      ht--;
    }
  }
}

// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}
```

```
// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
        break;
      case 2:
        printf("Enter the element to delete:");
        scanf("%d", &data);
        deletion(data);
        break;
      case 3:
        inorderTraversal(root);
        printf("\n");
        break;
      case 4:
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
```

```
    printf("\n");
  }
  return 0;
}
```

Red-Black Tree Applications

1. To implement finite maps
2. To implement Java packages: `java.util.TreeMap` and `java.util.TreeSet`
3. To implement Standard Template Libraries (STL) in C++: multiset, map, multimap
4. In Linux Kernel

Insertion in a Red-Black Tree

In this tutorial, you will learn how a new node can be inserted into a red-black tree is. Also, you will find working examples of insertions performed on a red-black tree in C, C++, Java and Python.

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

Before reading this article, please refer to the article on red-black tree.
While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

1. Recolor
2. Rotation

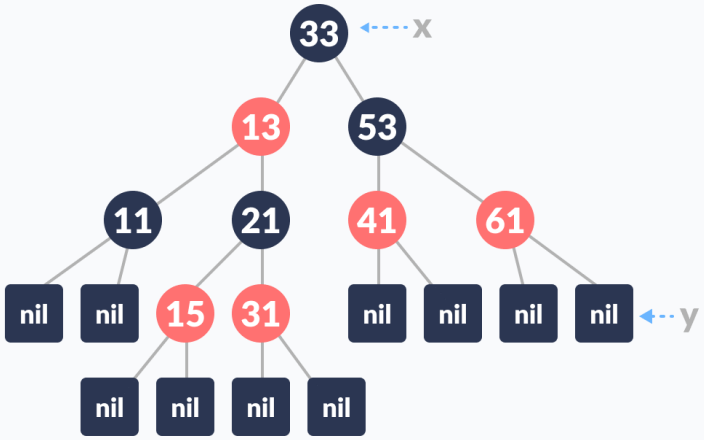
Algorithm to Insert a New Node

Following steps are followed for inserting a new element into a red-black tree:

1. The `newNode` be: 

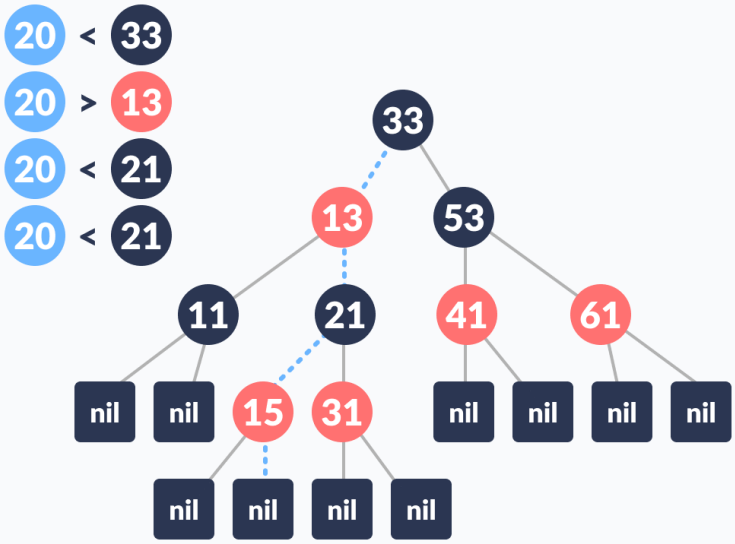
New node

2. Let y be the leaf (ie. `NIL`) and `x` be the root of the tree. The new node is inserted in the following tree.



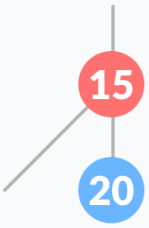
Initial tree

3. Check if the tree is empty (ie. whether `x` is `NIL`). If yes, insert `newNode` as a root node and color it black.
4. Else, repeat steps following steps until leaf (`NIL`) is reached.
    a. Compare `newKey` with `rootKey`.
    b. If `newKey` is greater than `rootKey`, traverse through the right subtree.
    c. Else traverse through the left subtree.

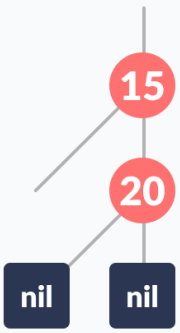


Path leading to the node where newNode is to be inserted

5. Assign the parent of the leaf as parent of `newNode`.
6. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.



7. Else, make `newNode` as `leftChild`. New node inserted
8. Assign `NULL` to the left and `rightChild` of `newNode`.



9. Assign RED color to `newNode`. Set the color of the newNode red and assign null to the children
10. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?

This is because inserting a red node does not violate the depth property of a red-black tree.

If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

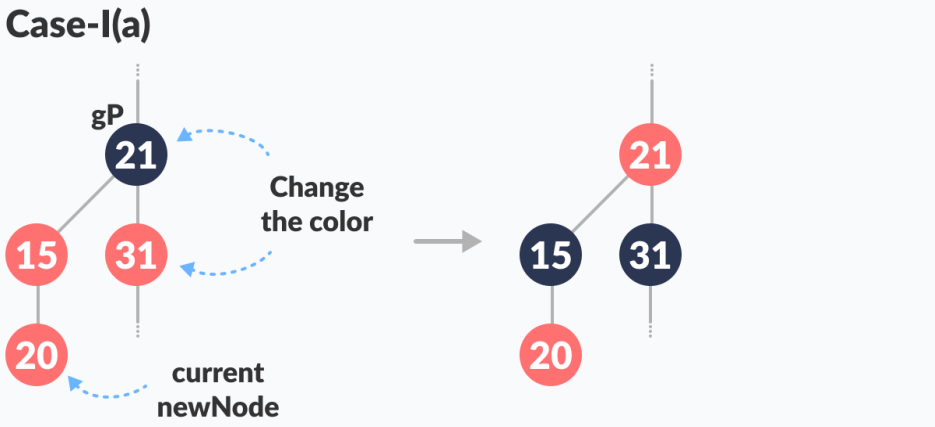
Algorithm to Maintain Red-Black Property After Insertion

This algorithm is used for maintaining the property of a red-black tree if insertion of a newNode violates this property.

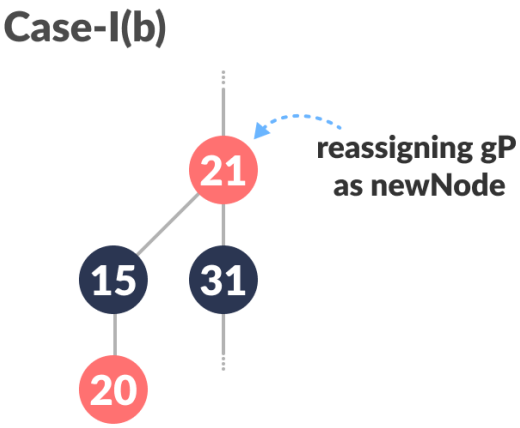
1. Do the following until the parent of `newNode` `p` is RED.
2. If `p` is the left child of `grandParent` `gP` of `newNode`, do the following.

   Case-I:

   a. If the color of the right child of `gP` of `newNode` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.



Color change

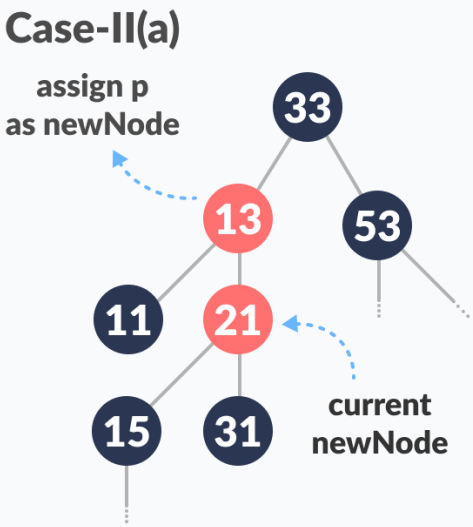


   b. Assign `gP` to `newNode`.

   Reassigning newNode

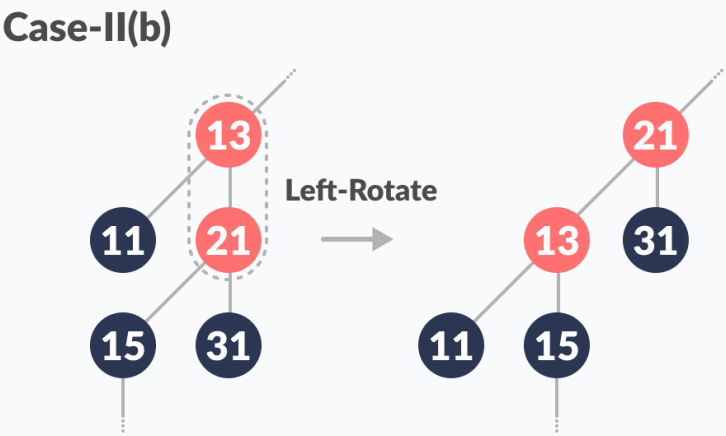
   Case-II:

c. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)
Else if `newNode` is the right child of `p` then, assign `p` to `newNode`.



Assigning parent of newNode as newNode

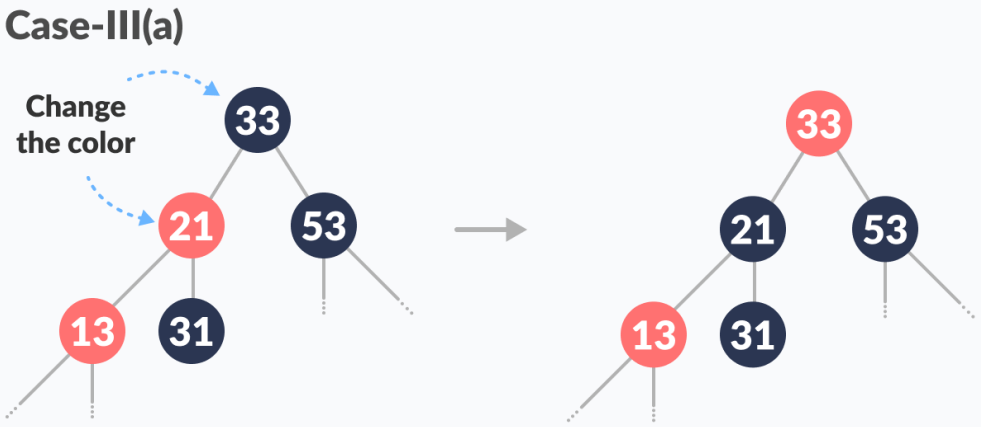
d. Left-Rotate `newNode`.



Left Rotate

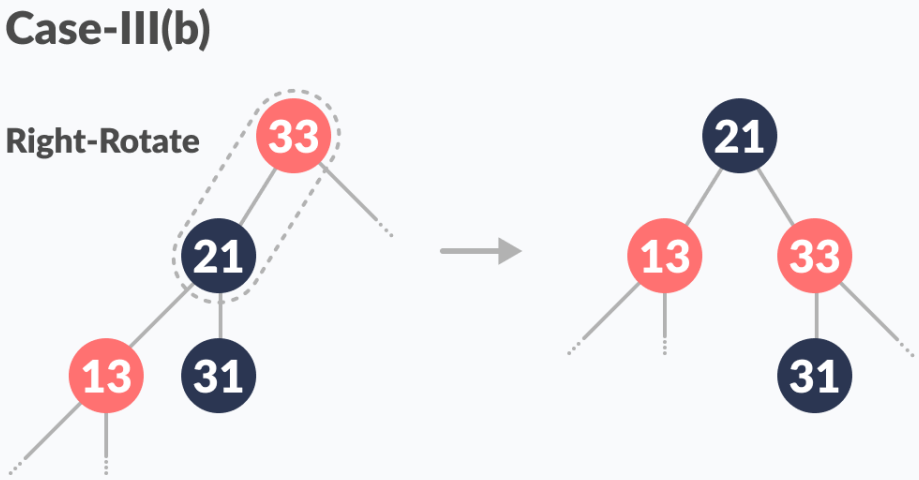
Case-III:

e. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)
Set color of `p` as BLACK and color of `gP` as RED.



Color change

f. Right-Rotate `gP`.



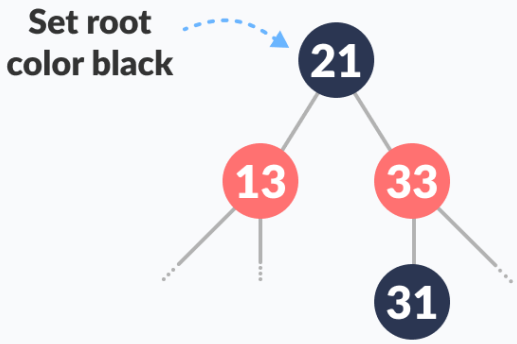
Right Rotate

3. Else, do the following.

a. If the color of the left child of `gP` of `z` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.
b. Assign `gP` to `newNode`.
c. Else if `newNode` is the left child of `p` then, assign `p` to `newNode` and Right-Rotate `newNode`.
d. Set color of `p` as BLACK and color of `gP` as RED.

e. Left-Rotate gP.

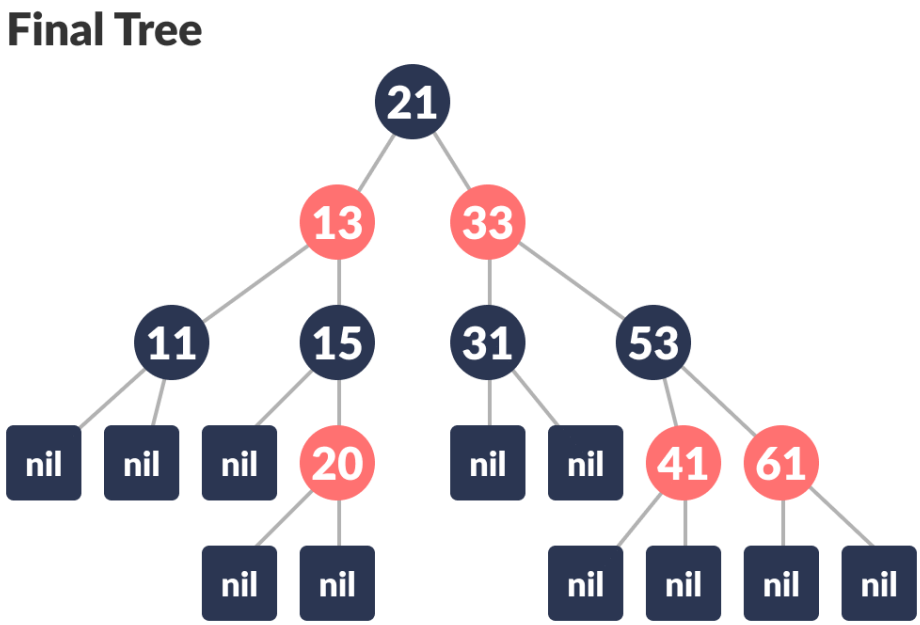
4. (This step is performed after coming out of the while loop.)



Set the root of the tree as BLACK.

Set root's color black

The final tree look like this:



Final tree

C Examples

```
// Implementing Red-Black Tree in C


#include <stdio.h>

#include <stdlib.h>


enum nodeColor {

  RED,

  BLACK

};


struct rbNode {

  int data, color;

  struct rbNode *link[2];

};


struct rbNode *root = NULL;
```

```
// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
```

```
}


stack[ht] = root;
dir[ht++] = 0;
while (ptr != NULL) {
  if (ptr->data == data) {
    printf("Duplicates Not Allowed!!\n");
    return;
  }
  index = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  ptr = ptr->link[index];
  dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
```

```
  if (yPtr != NULL && yPtr->color == RED) {
    stack[ht - 2]->color = RED;
    stack[ht - 1]->color = yPtr->color = BLACK;
    ht = ht - 2;
  } else {
    if (dir[ht - 1] == 0) {
      yPtr = stack[ht - 1];
    } else {
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[1];
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      stack[ht - 2]->link[0] = yPtr;
    }
    xPtr = stack[ht - 2];
    xPtr->color = RED;
    yPtr->color = BLACK;
    xPtr->link[0] = yPtr->link[1];
```

```
      yPtr->link[1] = xPtr;
      if (xPtr == root) {
        root = yPtr;
      } else {
        stack[ht - 3]->link[dir[ht - 3]] = yPtr;
      }
      break;
    }
  } else {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED)) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 1) {
        yPtr = stack[ht - 1];
      } else {
```

```
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[0];
      xPtr->link[0] = yPtr->link[1];
      yPtr->link[1] = xPtr;
      stack[ht - 2]->link[1] = yPtr;
    }
    xPtr = stack[ht - 2];
    yPtr->color = BLACK;
    xPtr->color = RED;
    xPtr->link[1] = yPtr->link[0];
    yPtr->link[0] = xPtr;
    if (xPtr == root) {
      root = yPtr;
    } else {
      stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
  }
```

```
    }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
  struct rbNode *stack[98], *ptr, *xPtr, *yPtr;
  struct rbNode *pPtr, *qPtr, *rPtr;
  int dir[98], ht = 0, diff, i;
  enum nodeColor color;

  if (!root) {
    printf("Tree not available\n");
    return;
  }

  ptr = root;
```

```
while (ptr != NULL) {
  if ((data - ptr->data) == 0)
    break;
  diff = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  dir[ht++] = diff;
  ptr = ptr->link[diff];
}


if (ptr->link[1] == NULL) {
  if ((ptr == root) && (ptr->link[0] == NULL)) {
    free(ptr);
    root = NULL;
  } else if (ptr == root) {
    root = ptr->link[0];
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
```

```
  }
} else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
    xPtr->color = ptr->color;
    ptr->color = color;

    if (ptr == root) {
      root = xPtr;
    } else {
      stack[ht - 1]->link[dir[ht - 1]] = xPtr;
    }

    dir[ht] = 1;
    stack[ht++] = xPtr;
  } else {
```

```
i = ht++;
while (1) {
  dir[ht] = 0;
  stack[ht++] = xPtr;
  yPtr = xPtr->link[0];
  if (!yPtr->link[0])
    break;
  xPtr = yPtr;
}

dir[i] = 1;
stack[i] = yPtr;
if (i > 0)
  stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];
```

```
    yPtr->link[1] = ptr->link[1];


    if (ptr == root) {
      root = yPtr;
    }


    color = yPtr->color;
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}


if (ht < 1)
  return;


if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
```

```
if (pPtr && pPtr->color == RED) {
  pPtr->color = BLACK;
  break;
}

if (ht < 2)
  break;

if (dir[ht - 2] == 0) {
  rPtr = stack[ht - 1]->link[1];

  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[1] = rPtr->link[0];
```

```
    rPtr->link[0] = stack[ht - 1];


    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
    dir[ht] = 0;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;


    rPtr = stack[ht - 1]->link[1];
  }


  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
```

```
    } else {
      if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
        qPtr = rPtr->link[0];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[0] = qPtr->link[1];
        qPtr->link[1] = rPtr;
        rPtr = stack[ht - 1]->link[1] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[1]->color = BLACK;
      stack[ht - 1]->link[1] = rPtr->link[0];
      rPtr->link[0] = stack[ht - 1];
      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
```

```
      }
      break;
    }
  } else {
    rPtr = stack[ht - 1]->link[0];
    if (!rPtr)
      break;

    if (rPtr->color == RED) {
      stack[ht - 1]->color = RED;
      rPtr->color = BLACK;
      stack[ht - 1]->link[0] = rPtr->link[1];
      rPtr->link[1] = stack[ht - 1];

      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
```

```
    }
    dir[ht] = 1;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;


    rPtr = stack[ht - 1]->link[0];
  }
  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
  } else {
    if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
      qPtr = rPtr->link[1];
      rPtr->color = RED;
      qPtr->color = BLACK;
      rPtr->link[1] = qPtr->link[0];
      qPtr->link[0] = rPtr;
```

```
        rPtr = stack[ht - 1]->link[0] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[0]->color = BLACK;
      stack[ht - 1]->link[0] = rPtr->link[1];
      rPtr->link[1] = stack[ht - 1];
      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      break;
    }
  }
  ht--;
 }
}
```

```
}


// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
  if (node) {
    inorderTraversal(node->link[0]);
    printf("%d  ", node->data);
    inorderTraversal(node->link[1]);
  }
  return;
}


// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
```

```
printf("\nEnter your choice:");
scanf("%d", &ch);
switch (ch) {
  case 1:
    printf("Enter the element to insert:");
    scanf("%d", &data);
    insertion(data);
    break;
  case 2:
    printf("Enter the element to delete:");
    scanf("%d", &data);
    deletion(data);
    break;
  case 3:
    inorderTraversal(root);
    printf("\n");
    break;
  case 4:
```

```
        exit(0);
      default:
        printf("Not available\n");
        break;
    }
    printf("\n");
  }
  return 0;
}
```

Deletion From a Red-Black Tree

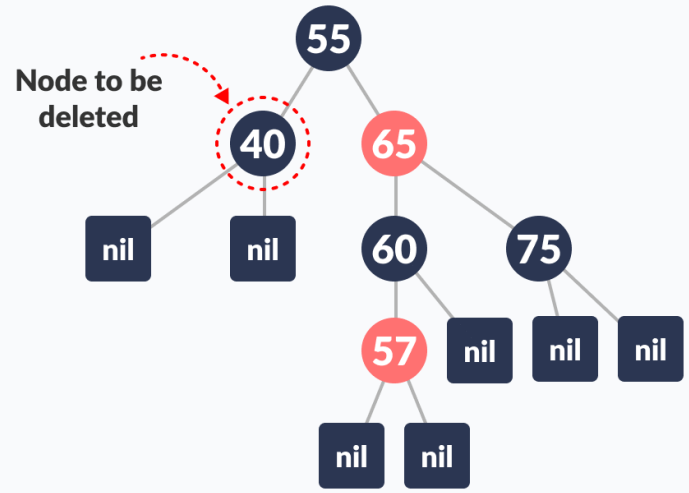
In this tutorial, you will learn how a node is deleted from a red-black tree is. Also, you will find working examples of deletions performed on a red-black tree in C, C++, Java and Python.

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

Before reading this article, please refer to the article on red-black tree. Deleting a node may or may not disrupt the red-black properties of a red-black tree. If this action violates the red-black properties, then a fixing algorithm is used to regain the red-black properties.

Deleting an element from a Red-Black Tree

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.



1. Let the `nodeToBeDeleted` be:

   Node to be deleted

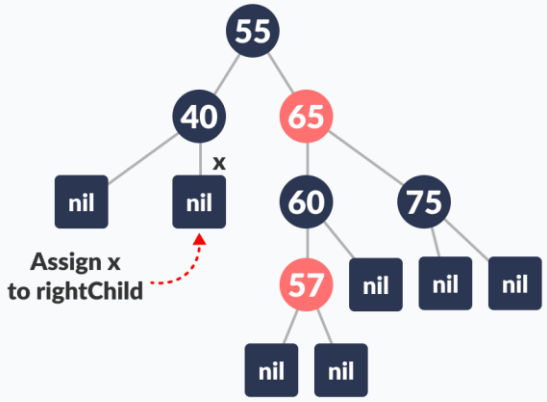
   originalColor = Black

2. Save the color of `nodeToBeDeleted` in `origrinalColor`.

   Saving original color

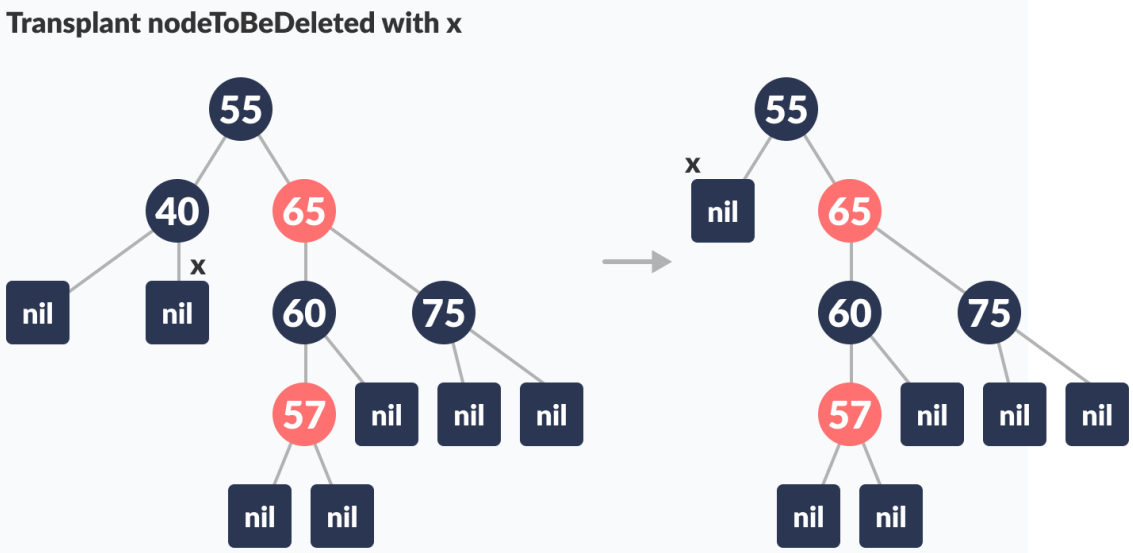
3. If the left child of `nodeToBeDeleted` is `NULL`

   a. Assign the right child of `nodeToBeDeleted` to x.



   Assign x to the rightChild

b. Transplant `nodeToBeDeleted` with `x`.



Transplant nodeToBeDeleted with x

4. Else if the right child of `nodeToBeDeleted` is `NULL`
    a. Assign the left child of `nodeToBeDeleted` into `x`.
    b. Transplant `nodeToBeDeleted` with `x`.
5. Else
    a. Assign the minimum of right subtree of `noteToBeDeleted` into `y`.
    b. Save the color of `y` in `originalColor`.
    c. Assign the `rightChild` of `y` into `x`.
    d. If `y` is a child of `nodeToBeDeleted`, then set the parent of `x` as `y`.
    e. Else, transplant `y` with `rightChild` of `y`.
    f. Transplant `nodeToBeDeleted` with `y`.
    g. Set the color of y with originalColor.
6. If the `originalColor` is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

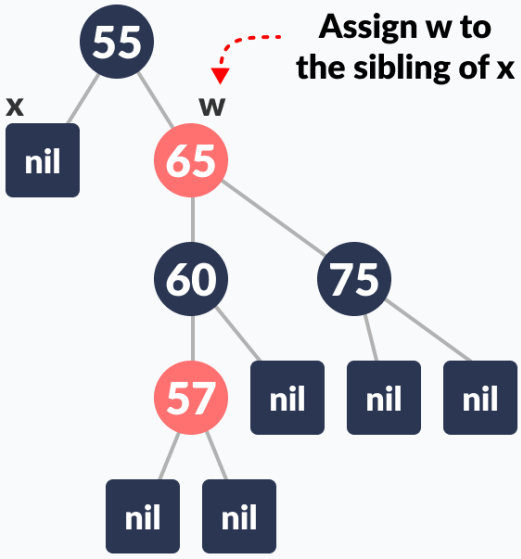
This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node `x` (which is occupying `y`'s original position) has an extra black. This makes node `x` neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.
However, the color attribute of `x` is not changed rather the extra black is represented in `x`'s pointing to the node.
The extra black can be removed if

1. It reaches the root node.
2. If `x` points to a red-black node. In this case, `x` is colored black.
3. Suitable rotations and recolorings are performed.

Following algorithm retains the properties of a red-black tree.

1. Do the following until the `x` is not the root of the tree and the color of `x` is BLACK
2. If `x` is the left child of its parent then,



a. Assign `w` to the sibling of `x`.

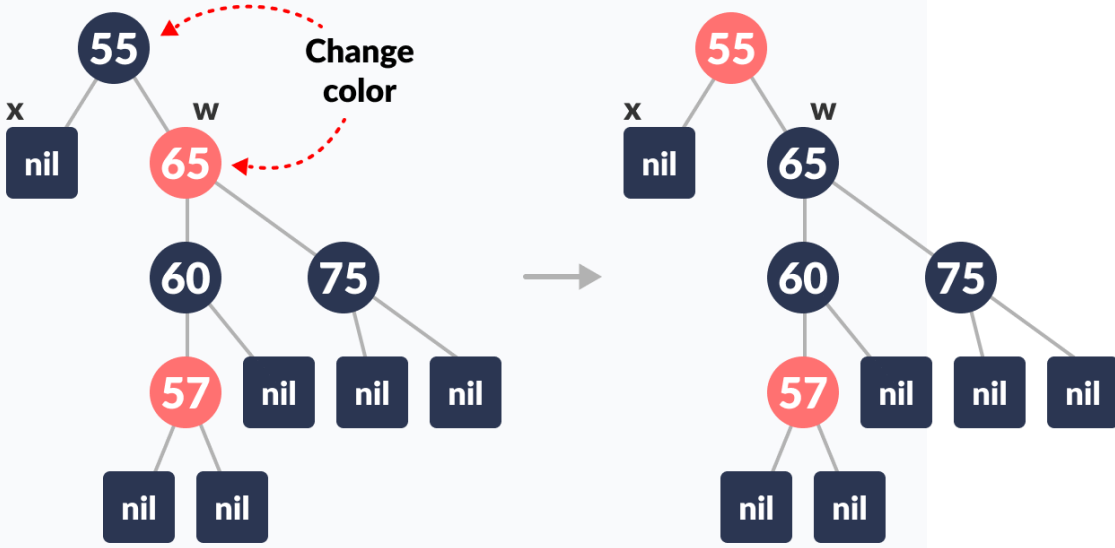
Assigning w

b. If the sibling of `x` is RED,

Case-I:

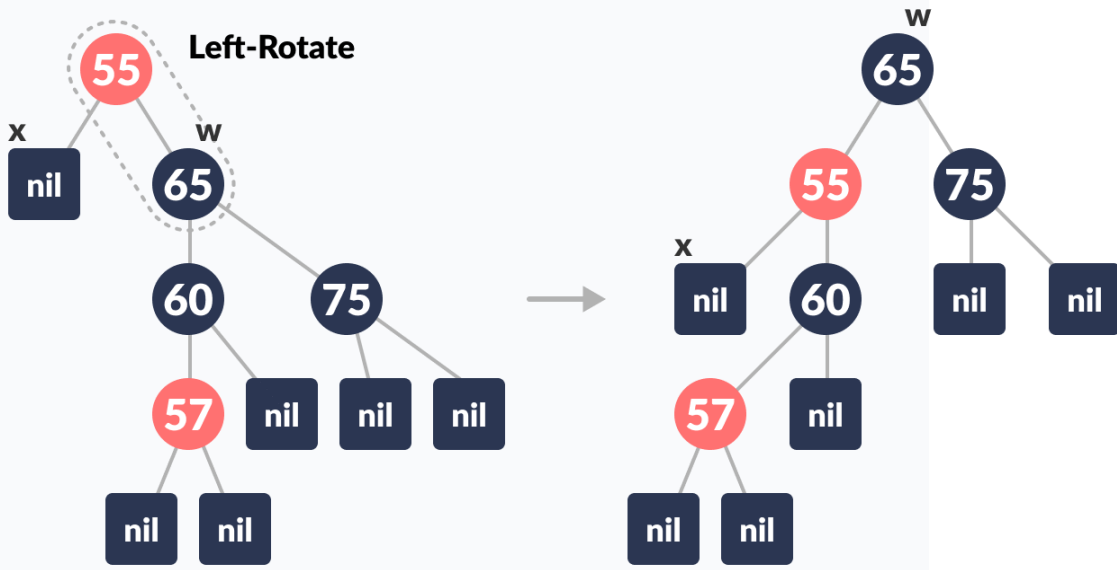
a. Set the color of the right child of the parent of `x` as BLACK.

b. Set the color of the parent of `x` as RED.



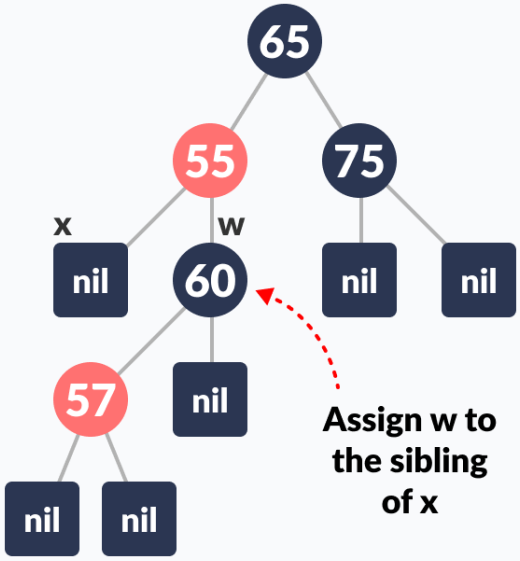
Color change

c. Left-Rotate the parent of `x`.



Left-rotate

d. Assign the `rightChild` of the parent of `x` to `w`.



Reassign w

c. If the color of both the right and the `leftChild` of `w` is BLACK,

Case-II:

a. Set the color of `w` as RED

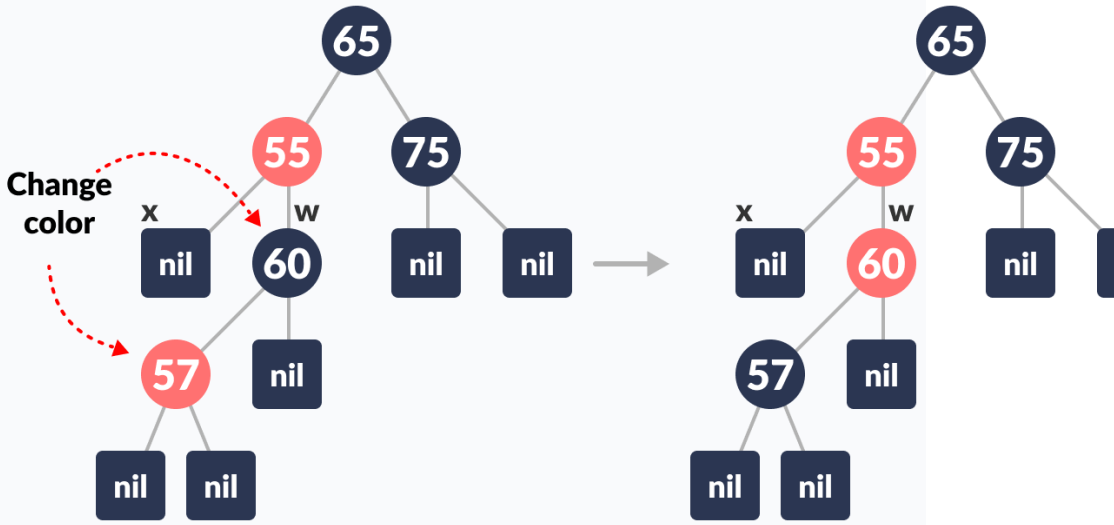
b. Assign the parent of `x` to `x`.

d. Else if the color of the `rightChild` of `w` is BLACK

Case-III:

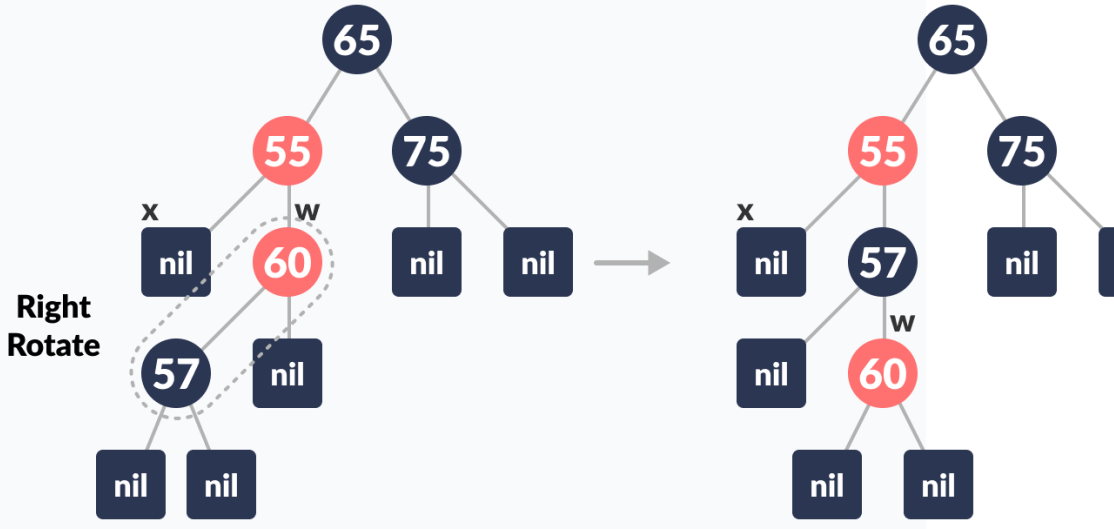
a. Set the color of the `leftChild` of `w` as BLACK

b. Set the color of `w` as RED



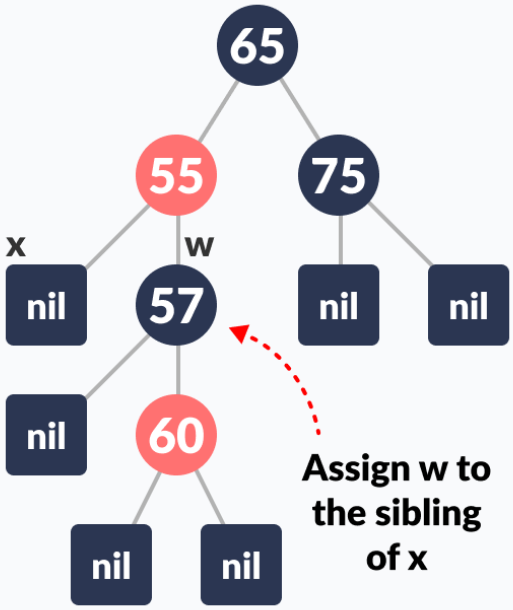
Color change

c. Right-Rotate `w`.



Right rotate

d. Assign the `rightChild` of the parent of `x` to `w`.

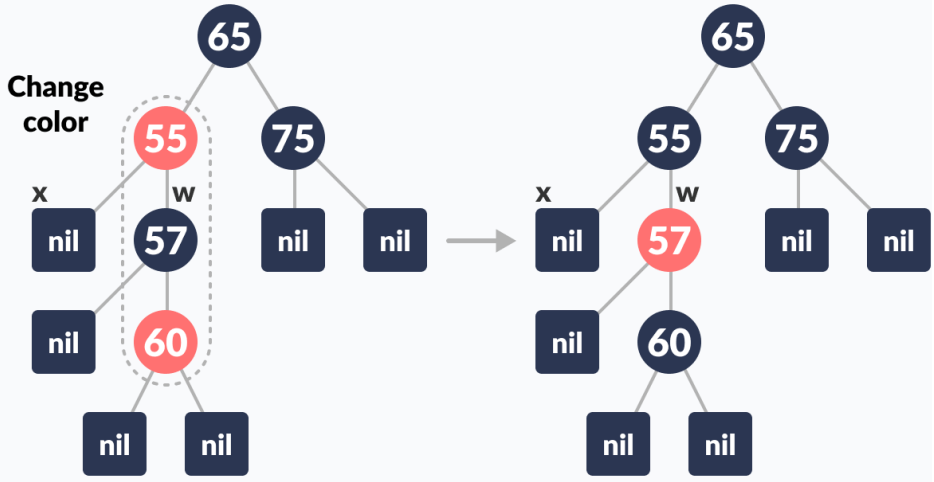


Reassign w

e. If any of the above cases do not occur, then do the following.

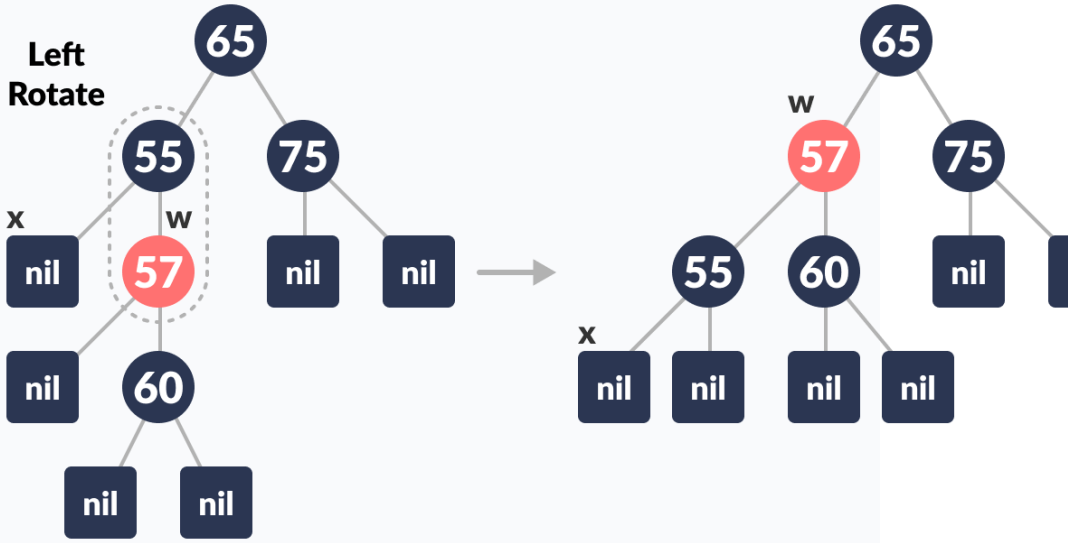
Case-IV:

a. Set the color of `w` as the color of the parent of `x`.
b. Set the color of the parent of parent of `x` as BLACK.
c. Set the color of the right child of `w` as BLACK.



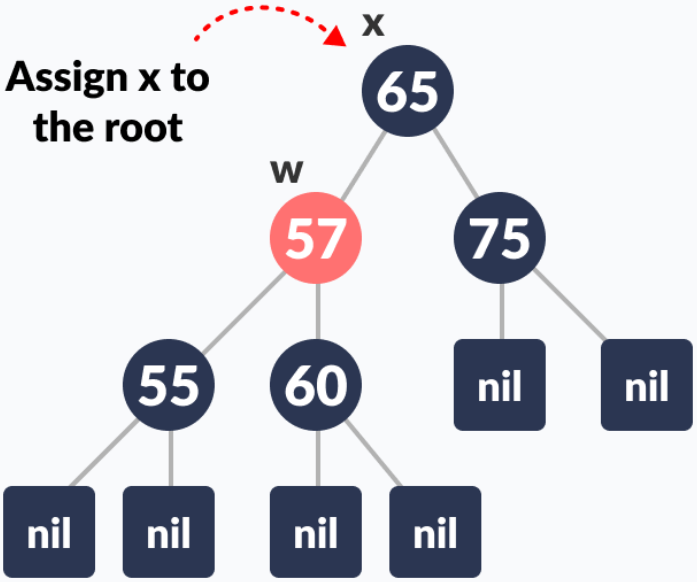
Color change

d. Left-Rotate the parent of `x`.



Left-rotate

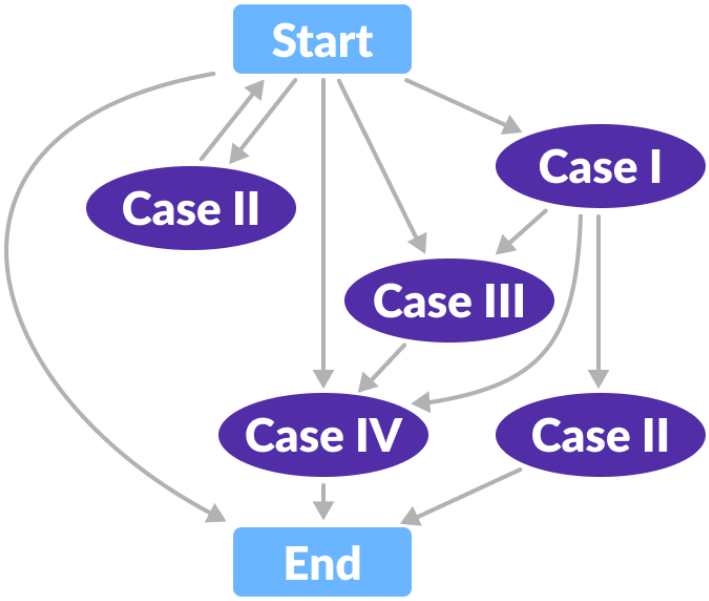
e. Set `x` as the root of the tree.



Set x as root

3. Else same as above with right changed to left and vice versa.
4. Set the color of `x` as BLACK.

The workflow of the above cases can be understood with the help of the flowchart below.



Flowchart for deletion operation

C Examples

```
// Implementing Red-Black Tree in C


#include <stdio.h>

#include <stdlib.h>


enum nodeColor {
```

```
  RED,
  BLACK
};

struct rbNode {
  int data, color;
  struct rbNode *link[2];
};

struct rbNode *root = NULL;

// Create a red-black tree
struct rbNode *createNode(int data) {
  struct rbNode *newnode;
  newnode = (struct rbNode *)malloc(sizeof(struct rbNode));
  newnode->data = data;
  newnode->color = RED;
  newnode->link[0] = newnode->link[1] = NULL;
```

```
  return newnode;
}

// Insert an node
void insertion(int data) {
  struct rbNode *stack[98], *ptr, *newnode, *xPtr, *yPtr;
  int dir[98], ht = 0, index;
  ptr = root;
  if (!root) {
    root = createNode(data);
    return;
  }

  stack[ht] = root;
  dir[ht++] = 0;
  while (ptr != NULL) {
    if (ptr->data == data) {
      printf("Duplicates Not Allowed!!\n");
```

```
    return;
  }
  index = (data - ptr->data) > 0 ? 1 : 0;
  stack[ht] = ptr;
  ptr = ptr->link[index];
  dir[ht++] = index;
}
stack[ht - 1]->link[index] = newnode = createNode(data);
while ((ht >= 3) && (stack[ht - 1]->color == RED)) {
  if (dir[ht - 2] == 0) {
    yPtr = stack[ht - 2]->link[1];
    if (yPtr != NULL && yPtr->color == RED) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 0) {
        yPtr = stack[ht - 1];
```

```
    } else {
      xPtr = stack[ht - 1];
      yPtr = xPtr->link[1];
      xPtr->link[1] = yPtr->link[0];
      yPtr->link[0] = xPtr;
      stack[ht - 2]->link[0] = yPtr;
    }
    xPtr = stack[ht - 2];
    xPtr->color = RED;
    yPtr->color = BLACK;
    xPtr->link[0] = yPtr->link[1];
    yPtr->link[1] = xPtr;
    if (xPtr == root) {
      root = yPtr;
    } else {
      stack[ht - 3]->link[dir[ht - 3]] = yPtr;
    }
    break;
```

```
    }
  } else {
    yPtr = stack[ht - 2]->link[0];
    if ((yPtr != NULL) && (yPtr->color == RED)) {
      stack[ht - 2]->color = RED;
      stack[ht - 1]->color = yPtr->color = BLACK;
      ht = ht - 2;
    } else {
      if (dir[ht - 1] == 1) {
        yPtr = stack[ht - 1];
      } else {
        xPtr = stack[ht - 1];
        yPtr = xPtr->link[0];
        xPtr->link[0] = yPtr->link[1];
        yPtr->link[1] = xPtr;
        stack[ht - 2]->link[1] = yPtr;
      }
      xPtr = stack[ht - 2];
```

```
        yPtr->color = BLACK;
        xPtr->color = RED;
        xPtr->link[1] = yPtr->link[0];
        yPtr->link[0] = xPtr;
        if (xPtr == root) {
          root = yPtr;
        } else {
          stack[ht - 3]->link[dir[ht - 3]] = yPtr;
        }
        break;
      }
    }
  }
  root->color = BLACK;
}

// Delete a node
void deletion(int data) {
```

```
struct rbNode *stack[98], *ptr, *xPtr, *yPtr;

struct rbNode *pPtr, *qPtr, *rPtr;

int dir[98], ht = 0, diff, i;

enum nodeColor color;



if (!root) {

  printf("Tree not available\n");

  return;

}



ptr = root;

while (ptr != NULL) {

  if ((data - ptr->data) == 0)

    break;

  diff = (data - ptr->data) > 0 ? 1 : 0;

  stack[ht] = ptr;

  dir[ht++] = diff;

  ptr = ptr->link[diff];
```

```
}


if (ptr->link[1] == NULL) {
  if ((ptr == root) && (ptr->link[0] == NULL)) {
    free(ptr);
    root = NULL;
  } else if (ptr == root) {
    root = ptr->link[0];
    free(ptr);
  } else {
    stack[ht - 1]->link[dir[ht - 1]] = ptr->link[0];
  }
} else {
  xPtr = ptr->link[1];
  if (xPtr->link[0] == NULL) {
    xPtr->link[0] = ptr->link[0];
    color = xPtr->color;
    xPtr->color = ptr->color;
```

```
    ptr->color = color;



    if (ptr == root) {
      root = xPtr;
    } else {
      stack[ht - 1]->link[dir[ht - 1]] = xPtr;
    }



    dir[ht] = 1;
    stack[ht++] = xPtr;
  } else {
    i = ht++;
    while (1) {
      dir[ht] = 0;
      stack[ht++] = xPtr;
      yPtr = xPtr->link[0];
      if (!yPtr->link[0])
        break;
```

```
  xPtr = yPtr;
}

dir[i] = 1;
stack[i] = yPtr;
if (i > 0)
  stack[i - 1]->link[dir[i - 1]] = yPtr;

yPtr->link[0] = ptr->link[0];

xPtr->link[0] = yPtr->link[1];
yPtr->link[1] = ptr->link[1];

if (ptr == root) {
  root = yPtr;
}

color = yPtr->color;
```

```
    yPtr->color = ptr->color;
    ptr->color = color;
  }
}

if (ht < 1)
  return;

if (ptr->color == BLACK) {
  while (1) {
    pPtr = stack[ht - 1]->link[dir[ht - 1]];
    if (pPtr && pPtr->color == RED) {
      pPtr->color = BLACK;
      break;
    }

    if (ht < 2)
      break;
```

```
if (dir[ht - 2] == 0) {
  rPtr = stack[ht - 1]->link[1];

  if (!rPtr)
    break;

  if (rPtr->color == RED) {
    stack[ht - 1]->color = RED;
    rPtr->color = BLACK;
    stack[ht - 1]->link[1] = rPtr->link[0];
    rPtr->link[0] = stack[ht - 1];

    if (stack[ht - 1] == root) {
      root = rPtr;
    } else {
      stack[ht - 2]->link[dir[ht - 2]] = rPtr;
    }
```

```
    dir[ht] = 0;
    stack[ht] = stack[ht - 1];
    stack[ht - 1] = rPtr;
    ht++;

    rPtr = stack[ht - 1]->link[1];
  }

  if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
    (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
    rPtr->color = RED;
  } else {
    if (!rPtr->link[1] || rPtr->link[1]->color == BLACK) {
      qPtr = rPtr->link[0];
      rPtr->color = RED;
      qPtr->color = BLACK;
      rPtr->link[0] = qPtr->link[1];
      qPtr->link[1] = rPtr;
```

```
        rPtr = stack[ht - 1]->link[1] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[1]->color = BLACK;
      stack[ht - 1]->link[1] = rPtr->link[0];
      rPtr->link[0] = stack[ht - 1];
      if (stack[ht - 1] == root) {
        root = rPtr;
      } else {
        stack[ht - 2]->link[dir[ht - 2]] = rPtr;
      }
      break;
    }
  } else {
    rPtr = stack[ht - 1]->link[0];
    if (!rPtr)
      break;
```

```
if (rPtr->color == RED) {
  stack[ht - 1]->color = RED;
  rPtr->color = BLACK;
  stack[ht - 1]->link[0] = rPtr->link[1];
  rPtr->link[1] = stack[ht - 1];

  if (stack[ht - 1] == root) {
    root = rPtr;
  } else {
    stack[ht - 2]->link[dir[ht - 2]] = rPtr;
  }
  dir[ht] = 1;
  stack[ht] = stack[ht - 1];
  stack[ht - 1] = rPtr;
  ht++;

  rPtr = stack[ht - 1]->link[0];
```

```
    }
    if ((!rPtr->link[0] || rPtr->link[0]->color == BLACK) &&
      (!rPtr->link[1] || rPtr->link[1]->color == BLACK)) {
      rPtr->color = RED;
    } else {
      if (!rPtr->link[0] || rPtr->link[0]->color == BLACK) {
        qPtr = rPtr->link[1];
        rPtr->color = RED;
        qPtr->color = BLACK;
        rPtr->link[1] = qPtr->link[0];
        qPtr->link[0] = rPtr;
        rPtr = stack[ht - 1]->link[0] = qPtr;
      }
      rPtr->color = stack[ht - 1]->color;
      stack[ht - 1]->color = BLACK;
      rPtr->link[0]->color = BLACK;
      stack[ht - 1]->link[0] = rPtr->link[1];
      rPtr->link[1] = stack[ht - 1];
```

```
        if (stack[ht - 1] == root) {
          root = rPtr;
        } else {
          stack[ht - 2]->link[dir[ht - 2]] = rPtr;
        }
        break;
      }
    }
    ht--;
  }
 }
}


// Print the inorder traversal of the tree
void inorderTraversal(struct rbNode *node) {
 if (node) {
   inorderTraversal(node->link[0]);
   printf("%d  ", node->data);
```

```
    inorderTraversal(node->link[1]);
  }
  return;
}

// Driver code
int main() {
  int ch, data;
  while (1) {
    printf("1. Insertion\t2. Deletion\n");
    printf("3. Traverse\t4. Exit");
    printf("\nEnter your choice:");
    scanf("%d", &ch);
    switch (ch) {
      case 1:
        printf("Enter the element to insert:");
        scanf("%d", &data);
        insertion(data);
```

```
      break;
    case 2:
      printf("Enter the element to delete:");
      scanf("%d", &data);
      deletion(data);
      break;
    case 3:
      inorderTraversal(root);
      printf("\n");
      break;
    case 4:
      exit(0);
    default:
      printf("Not available\n");
      break;
  }
  printf("\n");
}
```

```
  return 0;

}
```