SECTION 8. BINARY TREE ALGORITHMS

8.1. Representation of binary trees

Binary trees as an abstract data type were discussed in the first section, here we will talk about the computer implementation of this type of data structures using the visual algorithmic language DRAKON and the programming language Golang. Recall the basic terminology of binary trees (Fig 8.1):

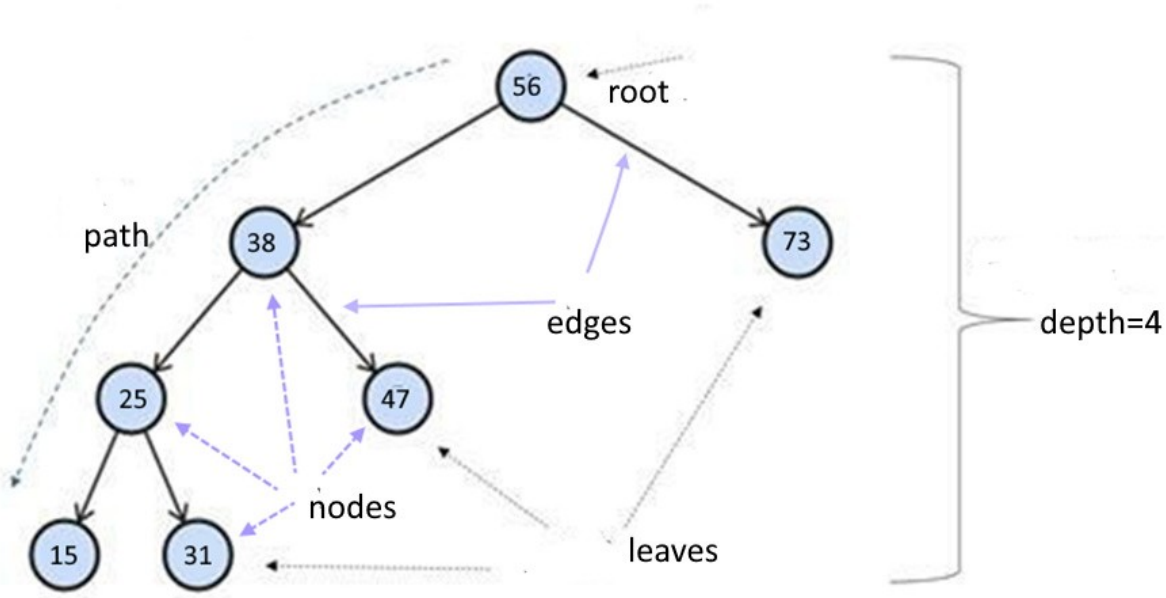


Figure 8.1. Basic terminology of a tree

Root: The root of a tree is the only node with no incoming edges. It is the top node in the tree;

Node: This is the basic element of a tree. Each node has data and two references that can point to zero or its descendants; Edge: This is also the fundamental part of a tree and is used to connect between two nodes.

Path: A path is an ordered list of nodes connected by edges.

Leaf: A leaf node is a node that has no descendants.

Tree height: The height of a tree is the number of edges on the longest path between the root and a leaf.

Node level: Node level is the number of edges on the path from the root node of this node.

The information structure of the binary tree is organised as follows (Figure 8.2):

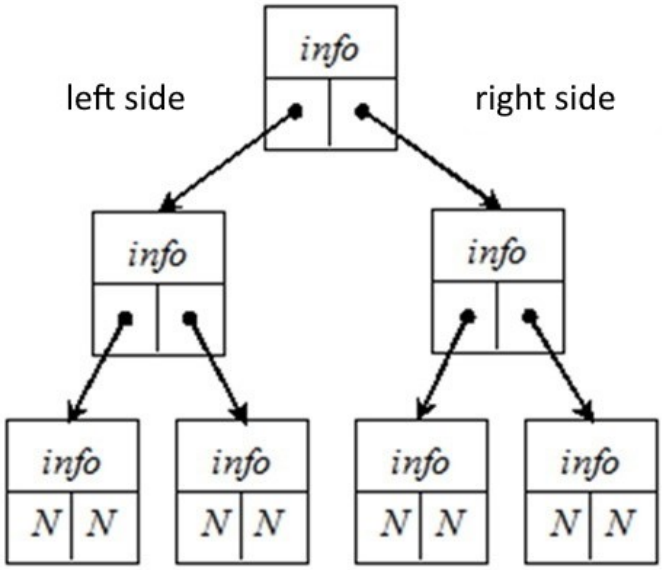


Figure8.2 Binary tree structure (*info* - value (key), (N - NULL))

Several types of binary trees are discussed in the training literature, the most important of which is classification based on node values:

- a binary search tree;
- AVL-Tree;
- Red-Black Tree.

8.2. Binary search tree

8.2.1. Building a binary tree

If the tree is organised in such a way that, for each node, all the node values of its left subtree are less than this node and all the values of its right subtree are greater, it is called a binary search tree. A BST is a recursive data structure because each subtree is also a tree. A BST has the following properties the tree consists of nodes that retain unique values;

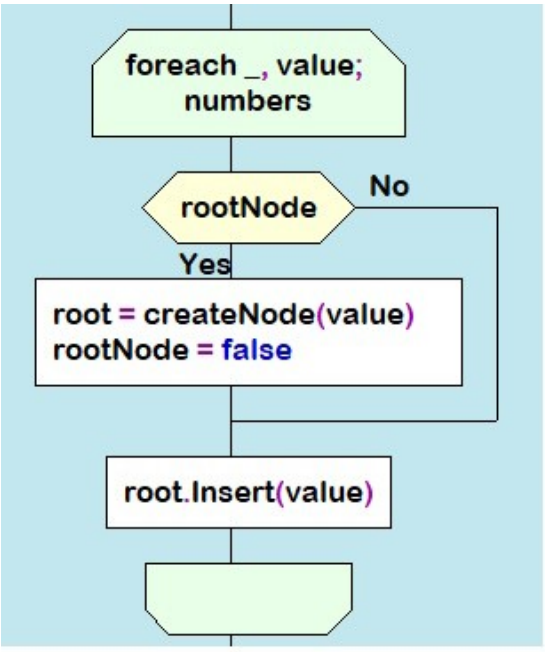
- each node has zero, one or two child nodes;

- one of the nodes is referred to as the root node, which is at the top of the tree structure;

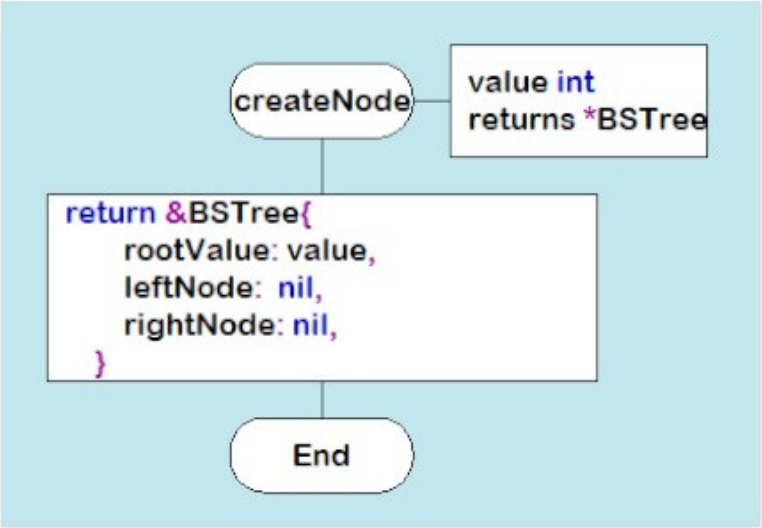
- each node has only one parent node, except for the root node which has no parent node;

- the value of each node is greater than that of its left child but less than that of its right child;

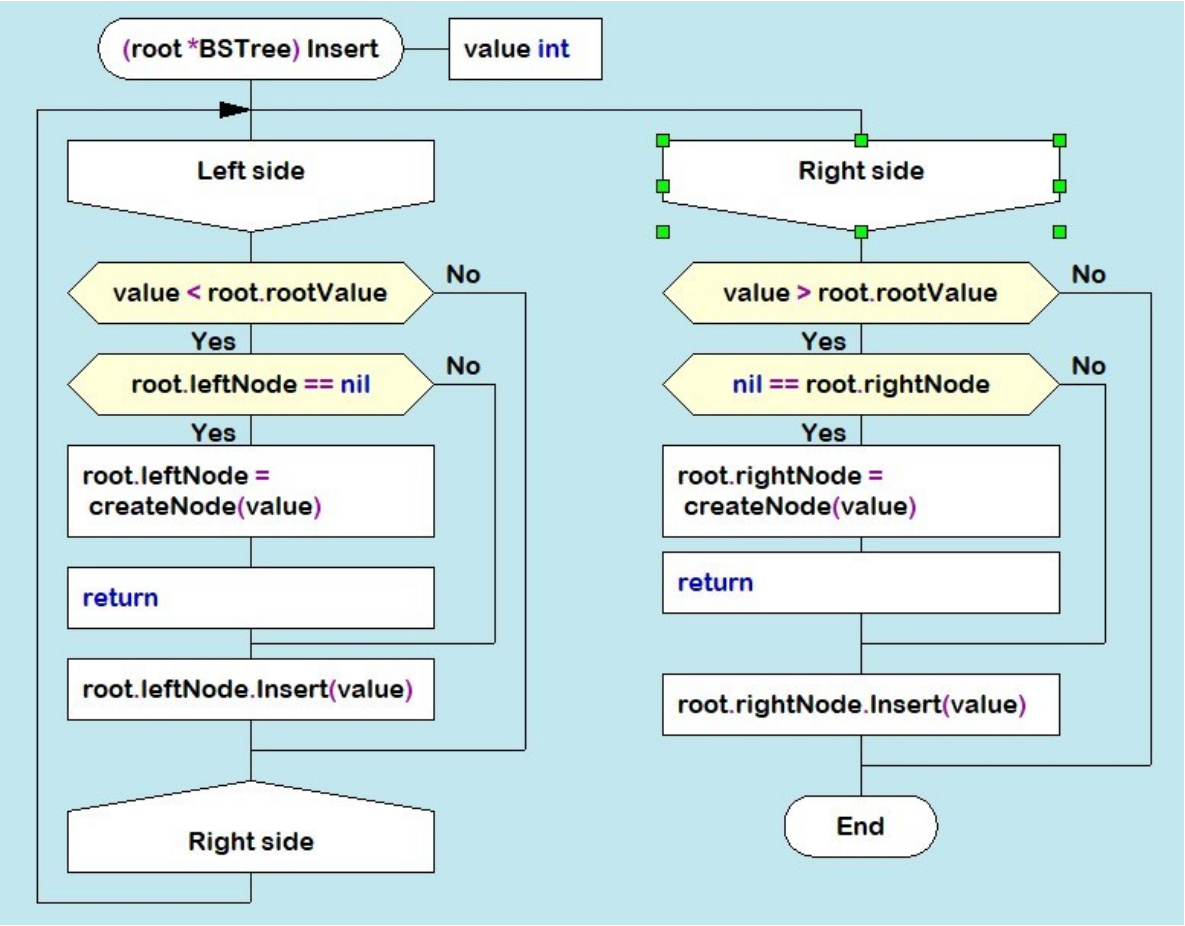
1. A binary search tree is built according to a certain rule (algorithm). Consider the sequence of integers {11, 5, 17, 15, 1, 8, 19, 13, 21}, represented as a slice of `numbers`. First, the root node {11} is formed. Then in the loop for each node recursively the method *`insert(value)`* is called (Figure 8.2 a), which in turn calls the method *`createNode(value)`*, which creates a new node (Figure 8.2 b) and places it in the left or right subtree depending on the value (Figure 8.2 c).



a) building a tree;



b) creating a node



c) node insertion

Figure 8.3. Drakon-diagram of tree building methods

The complete process of creating a binary tree is shown in Figure 8.4. The first number 11 is written to the root of the tree. The second number 5 is less than the value in the tree root, so it is written to the left subtree. The next number 17 is greater than the root number, so it is written to the right subtree. Then the number 15 is greater than the value at the root of the tree, so it is written to the right subtree, but the right subtree is already built. The number 15 is compared to the number 17 at the root of the right subtree. Since the value to be added is less than the value in the root of the right subtree, we add the left subtree to this node. The final result is a binary search tree with three variants: a node 5 is the parent of two children (1.8), a node 15 has only a left child, and a node 19 has only a right child. This arrangement of nodes is chosen to demonstrate the function of removing nodes, which will be discussed later.

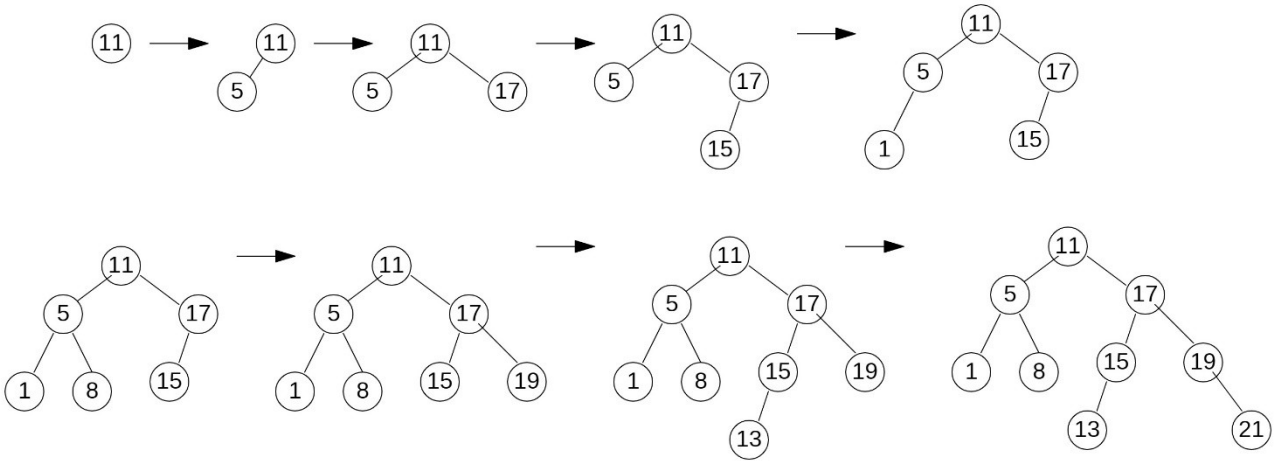


Figure 8.4. Building a binary tree

8.2.2 Finding a node based on a set value

Another basic operation is the *findNode(value)* function. This function uses a Golang construct, Select, which recursively compares the value of val with the values of other nodes as it traverses the tree. Note that the Select statement is represented by if-else operators in the generated code. If a match is found, the result "Node exists" is printed. The absence of a node with this value is detected by the equality *<nil>* of the addresses *root.leftNode* and *root.rightNode* in the variable root of type BSTree: *BSTree struct {*

```
    rootValue int
    leftNode *BSTree
    rightNode *BSTree
}
```

The Drakon-diagram of the *findNode(root *BSTree, val int)* function is shown in Figure 8.5. The search is performed over the whole tree, the end nodes are determined when the condition (*root.leftNode == nil && root.rightNode == nil)* is satisfied.

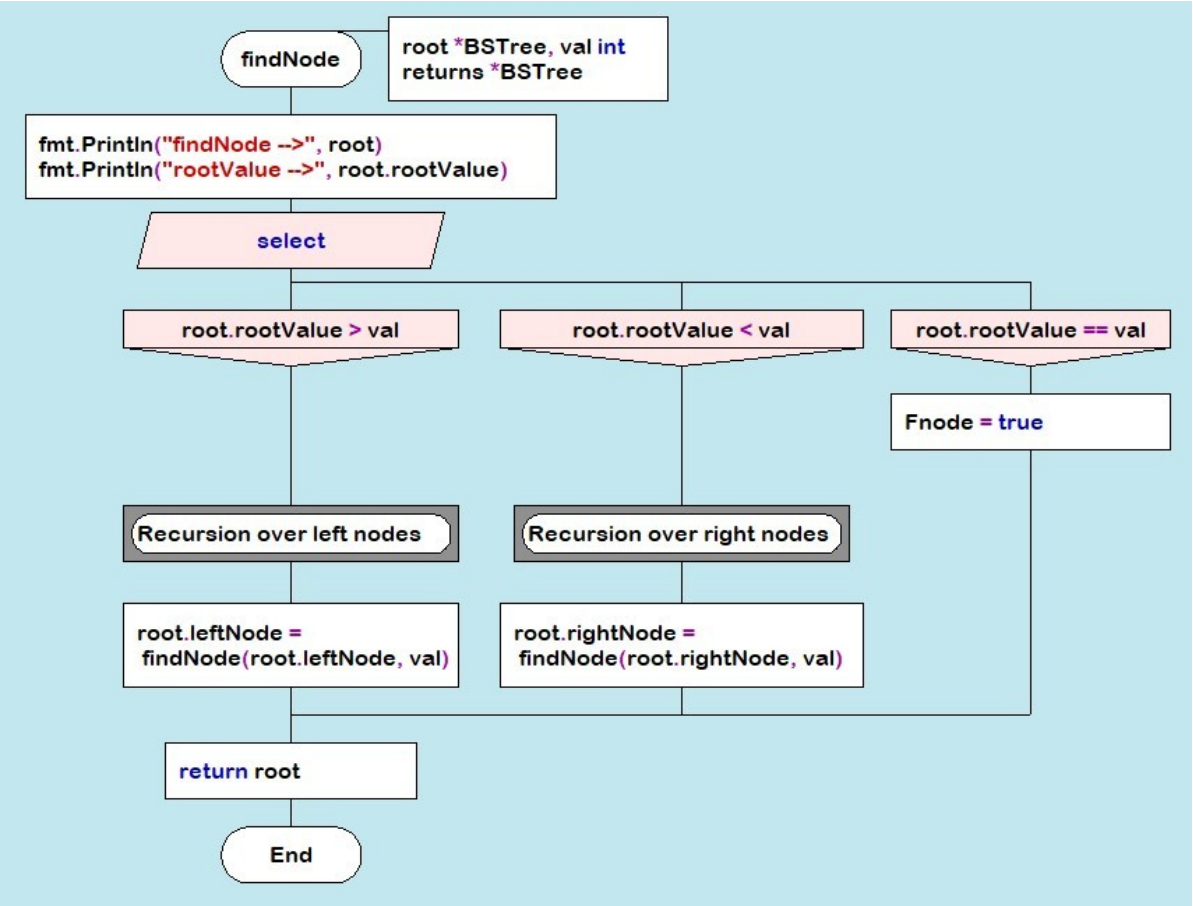


Figure 8.5. Drakon-diagram of *findNode(root *BSTree, val int)* method

8.2.3. Deleting a node with a set value

The next basic operation is to delete a node with the given value. The function *deleteNode (root *BSTree, val int)* is used recursively. The algorithm of this function is complicated by the fact that the following descendant arrangements are possible, as shown in Figure 8.3:

(a) Node 21 to be deleted has no descendants;

b) the node 19 to be deleted has a right-hand descendant;

c) the node 15 to be deleted has a left-hand descendant;

d) the node 5 to be deleted has a right-hand descendant.

Let's analyse in more detail the order in which the nodes are moved in these variants.

Option a): Node (21) has no descendants (Figure 8.6.).

In this case, this node is removed by changing the value of `root = nil` in the parent node. Figure 8.6. shows the process of deleting node (21) and the corresponding Drakon-diagram fragment, where L1 := root.leftNode == nil and R1 := root.rightNode == nil.

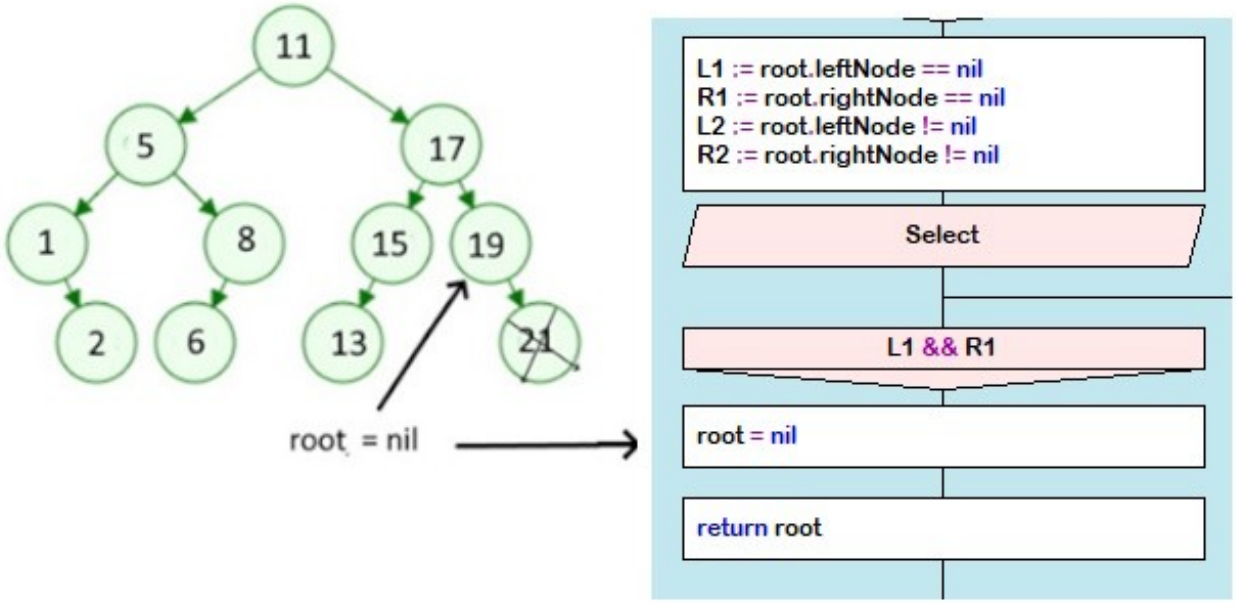


Figure 8.6. Deleting a node without childs

Option b): Node (19) has a right-hand descendant (Figure 8.7).

In this case, node (19) is removed from the tree by replacing its address in the parent node (17) with the address of its only descendant node (21). Figure 8.7. shows the process of removing node (19) and the corresponding Drakon-diagramfragment, where `L1 := root.leftNode == nil` and `R2 := root.rightNode != nil`.

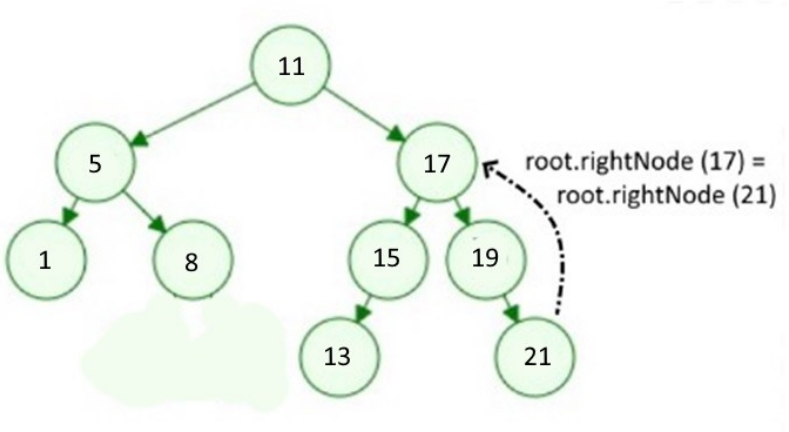


Figure 8.7. Deleting node (19) with right-hand descendant

Variant c): Node (15) has a left-hand child (Figure 8.8.).

In this case, node (15) is removed from the tree by replacing its address in parent node (17) with the address of its only child node (13). Figure 8.8. shows the process of removing node (15) and the corresponding Drakon-diagramfragment, where L1 := root.leftNode == nil and R2 := root.rightNode != nil.

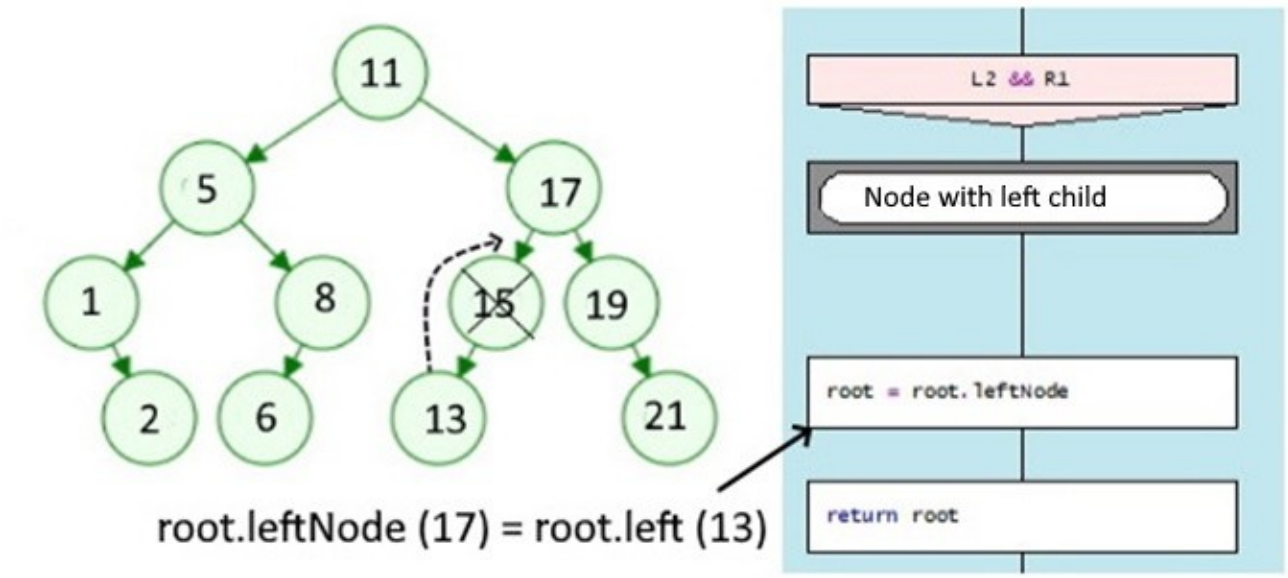


Figure 8.8. Deleting node (15) with left-hand child

Variant d): Node (5) has two descendants (Figure 8.9.). In this case the binary search tree is rearranged: node (2) moves to the place of node (5):

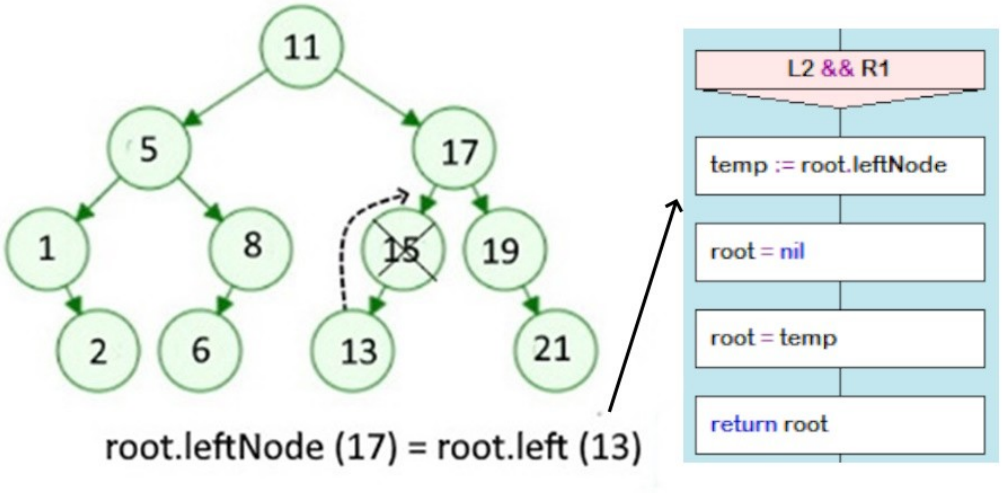


Figure 8.9. Deleting a node (5) with two childs

The algorithmic representation of the process for removing a node from a binary search tree in Drakon-diagram form is shown on Figure.8.10:

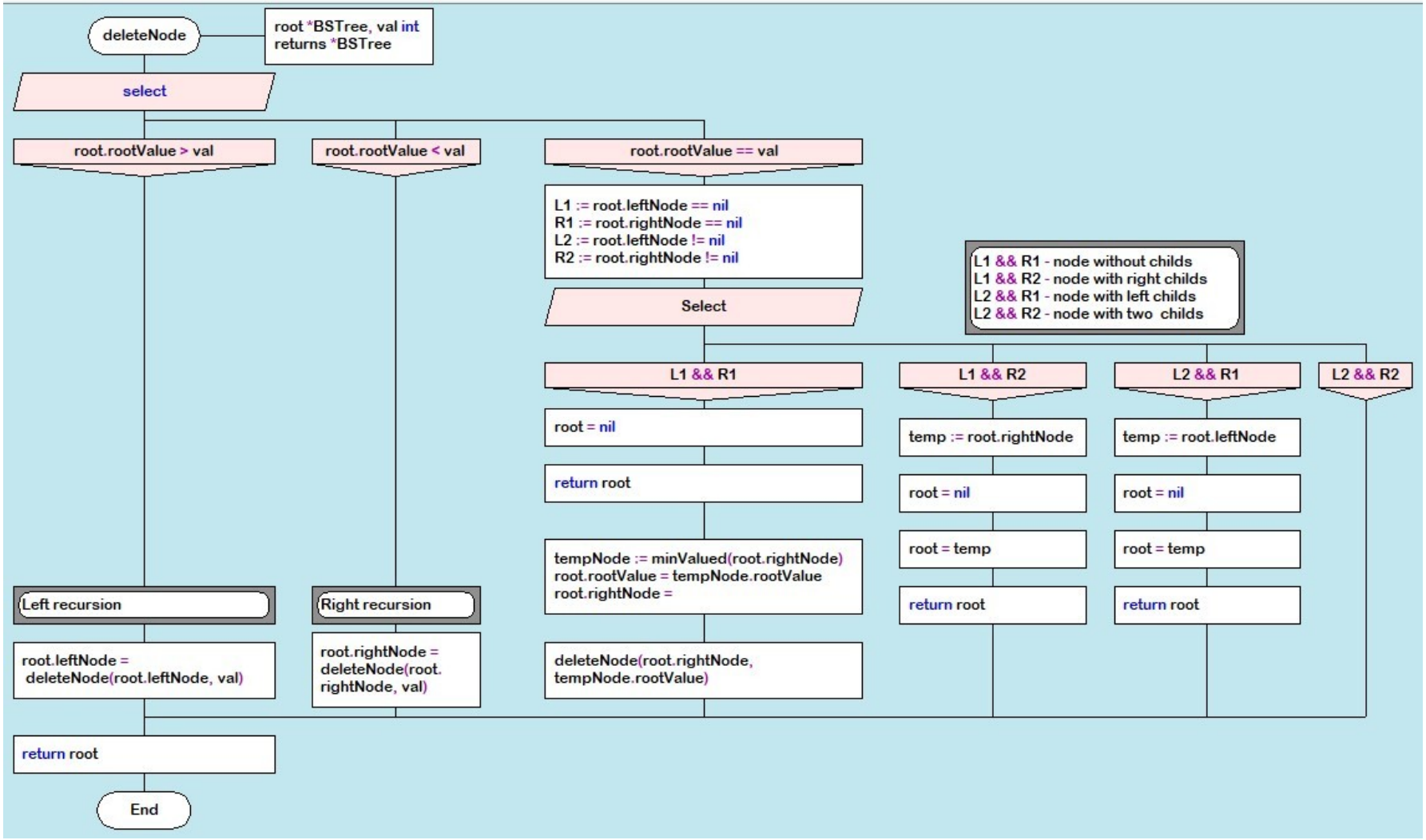


Figure 8.10. Drakon-diagram depicting the process of constructing a tree

8.2.4 Traversing the binary search tree

One of the basic tree operations is to traverse tree nodes. Unlike linear data structures, where elements are traversed in a linear order, tree nodes can be traversed in a variety of ways. A traversal in which each ancestor node is traversed before its childs is called a pre-order traversal or a forward traversal. A traversal where each ancestor node is traversed before its ancestors is called a post-order traversal. There is also an in-order traversal where the left subtree is visited from bottom to top, then the root node, then the right subtree (Figures 8.11, 8.12, 8.13).

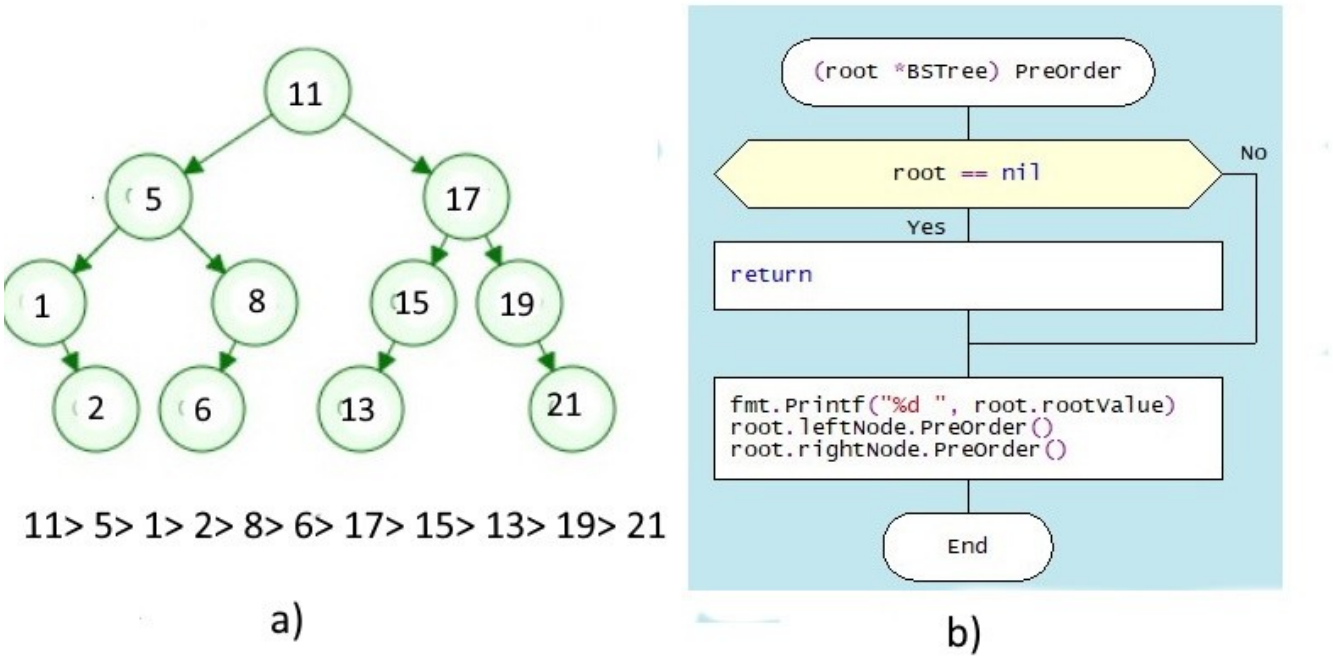


Figure 8.11. a) Direct tree traversal; b) PreOrder() Drakon-diagram

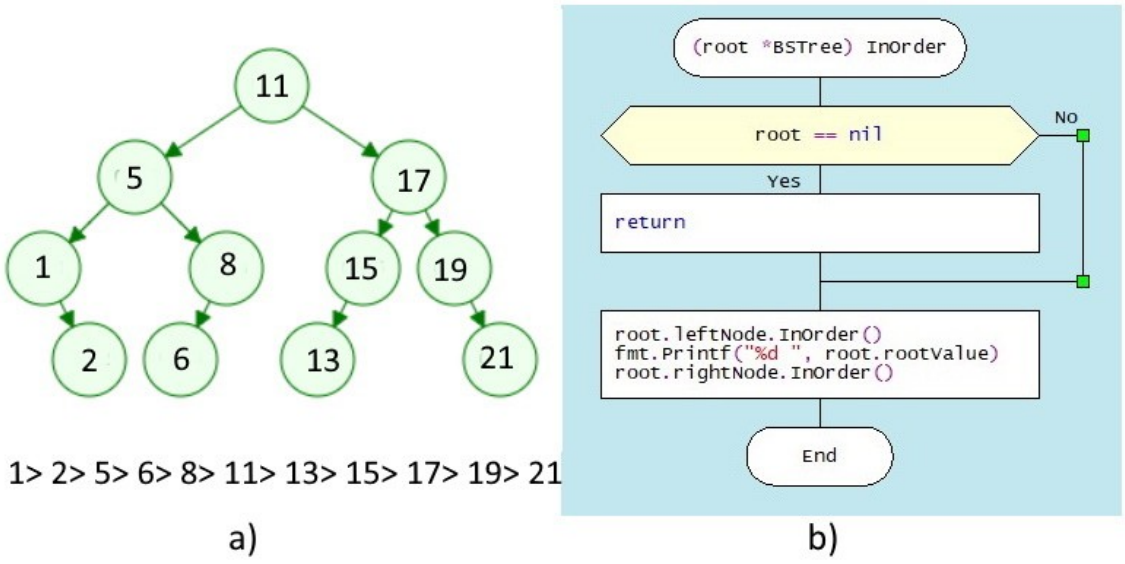


Figure 8.12. a) Centred traversal; b) *InOrder()* Drakon_diagram

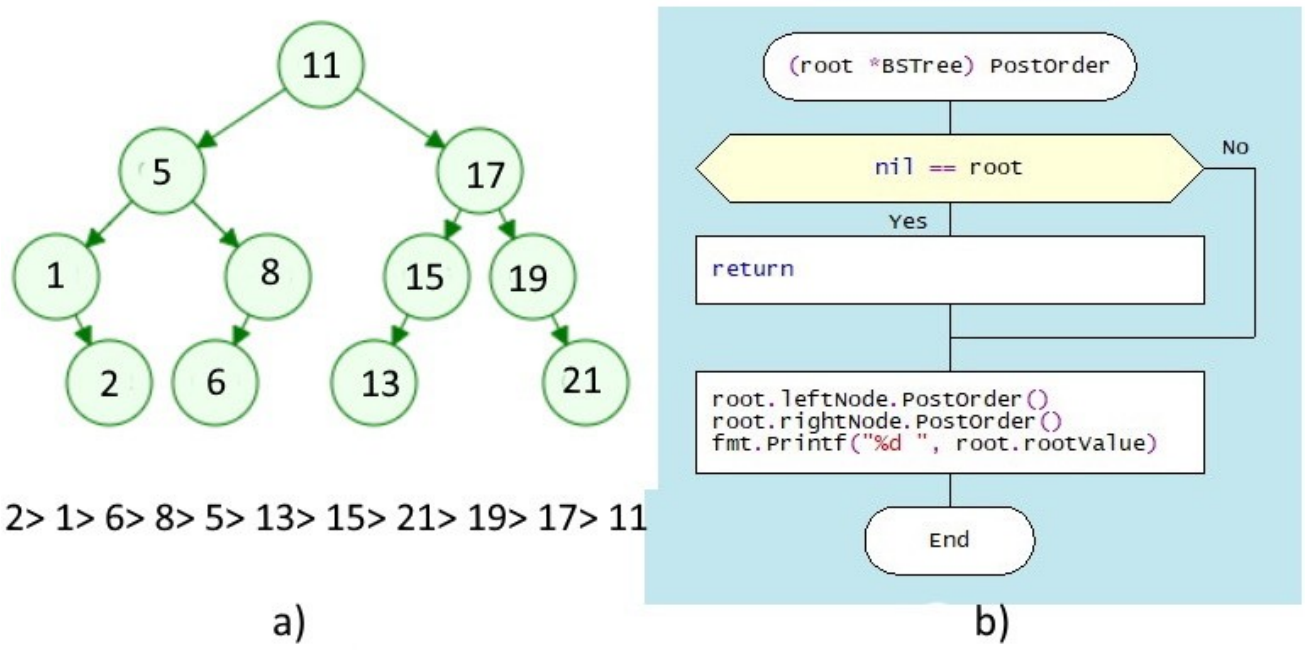


Figure 8.13. a) Reverse traversal; b) Drakon-diagram of the PostOrder() function

8.3. Self-balancing binary trees (AVL-trees)

8.3.1. The need to balance trees

The efficiency of any tree operation depends substantially on the order in which the input data is received. For example, if an incoming sequence of numbers is partially

sorted in ascending or descending order, this structure will no longer look like a tree (Figure 8.14).

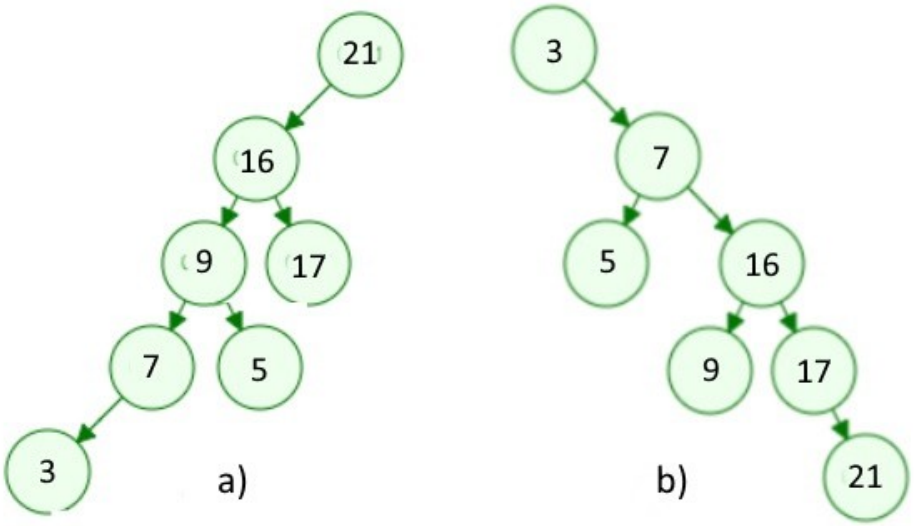


Figure 8.14. Partially sorted input data

In such practically "degenerate" trees, the complexity of the operations is determined by the number of nodes, i.e. it is almost linear - O(n). The left and right subtrees are unbalanced, which can be estimated by the balance coefficient (kb), which is equal to the difference between the heights of the left and right subtrees. Recall that the height of a tree is defined as the length of the longest branch in a subtree (sum of edges). For an ideal binary search tree (a tree where the number of nodes in the left subtree is equal to the number of nodes in the right subtree), this factor is 0 (Figure 8.15).

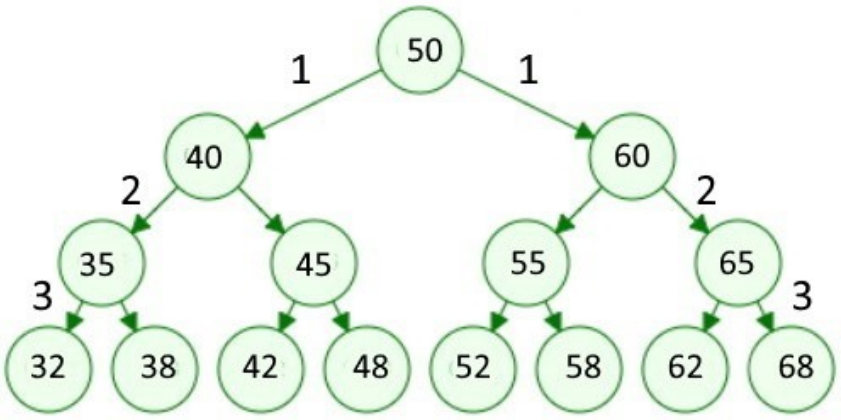


Figure 8.15. An ideal binary tree

In real life, ideal binary trees are almost never achieved; programmers often try to build a tree where the height of the left subtree differs from the height of the right subtree by no more than 1. Such trees are called AVL-trees; for such trees the complexity of operations is defined as O(logn), i.e. the execution time of basic operations (search, delete) is significantly less than for BST-trees.

8.3.2. Tree balancing principle

The algorithms for such trees are based on the process of balancing the tree when a new node is inserted or an existing node is removed. The purpose of balancing is to reconstruct the tree so that the heights of the left and right subtrees do not differ by more than 1. The balancing factor must satisfy the following conditions allowable values `kb` = -1, 0 and +1;

- The value `kb` = -1 indicates that the right subtree "outweighs" the left subtree;
- The value kb= +1 indicates that the left subtree "outweighs" the right subtree;
- a value of kb = 0 indicates that the tree contains an equal number of nodes on each side, i.e. the tree is perfectly balanced.

The balancing technique boils down to making circular movements of the knots in four variants:

- right turn (RR);
- Left turn (LL);
- Right - left turn (RL);
- Left to right turn (LR).

The right-hand turn is performed when the root node has a balance factor kb = +2 and its left-hand child has a balance factor kb = +1 (Figure 8.16):

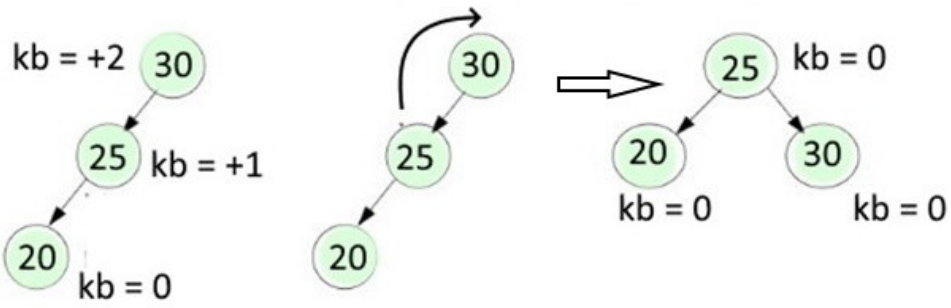


Figure 8.16. Right turn

The left-hand turn is performed when the root node has a balanced kb = - 2 and its right-hand descendant has a balanced kb = - 1 (Figure 8.17):

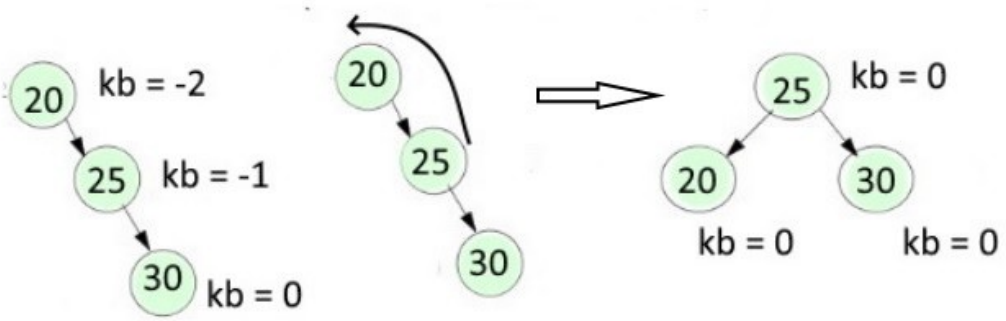


Figure 8.17. Left turn

A right-left turn is performed when the root node has a balance factor kb = -2 and its right-hand child has a balance factor kb = +1 (Figure 8.18):

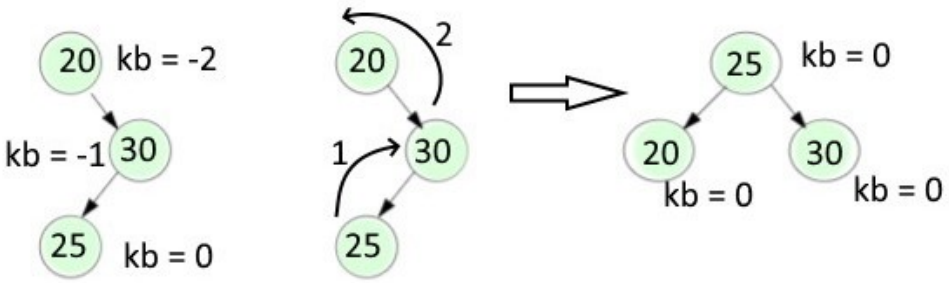


Figure 8.18. Right-left turn

Left-right rotation is performed when a node has a balance factor of kb = +2 and its left child has a balance factor of kb = -1 (Figure 8.19):

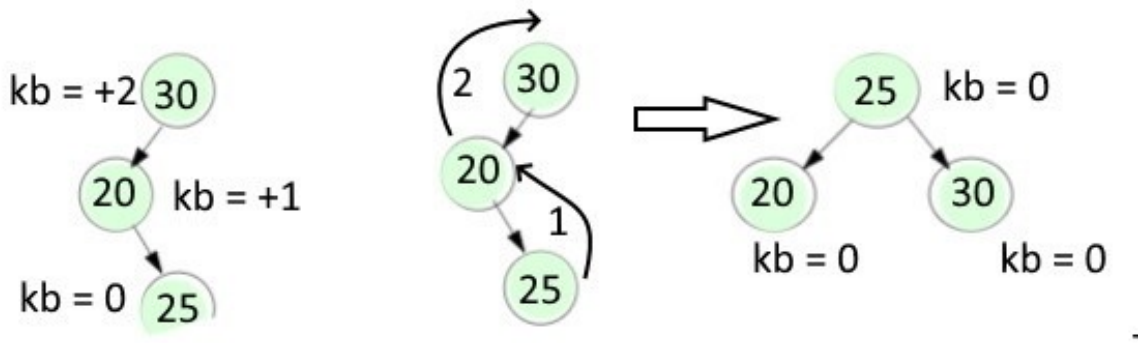


Figure 8.19. Left-right turn

Consider the balancing process in more detail with an example of tree formation when input data is received in this order: 21, 17, 16, 11, 9, 7, 5, 3. Such a tree is unbalanced, or almost a "twig" (Figure 8.20):

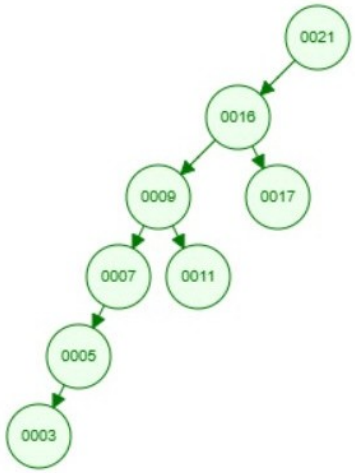


Figure 8.20. Unbalanced tree

8.3.3. Balancing the tree when introducing a new node

Balancing checks are started on the arrival of each new node using the *InsertNode(node *node, value int)* method, which determines the position of the new node in the left or right subtree relative to the root node. The Drakon-diagram of that algorithm is shown in Figure 8.21. The choice of one of the possible balancing paths (rotations) is determined by the *rotateInsert (node *node, val int)* method (Figure 8.22).

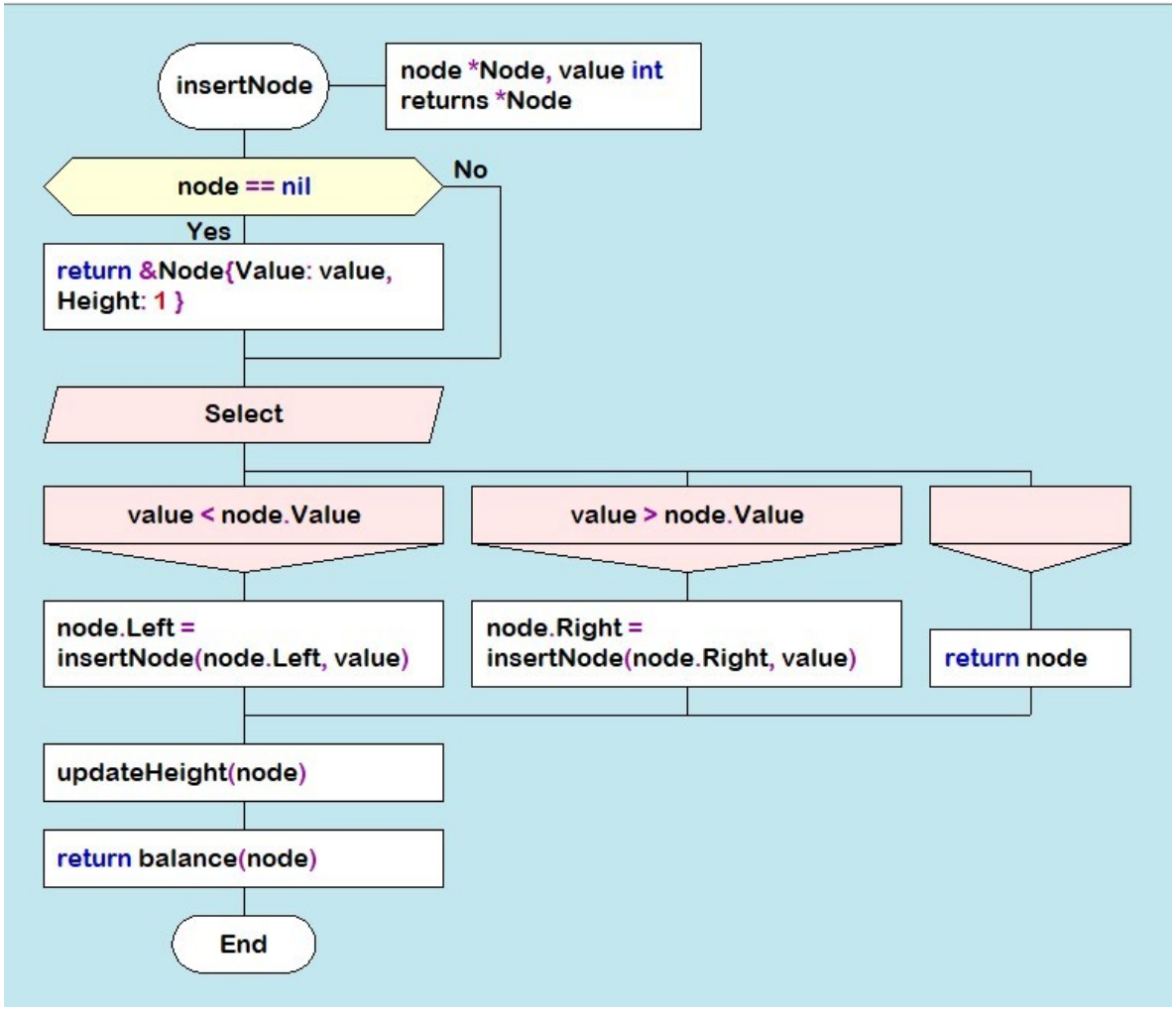


Figure 8.21. Drakon-diagram depicting the method of inserting node

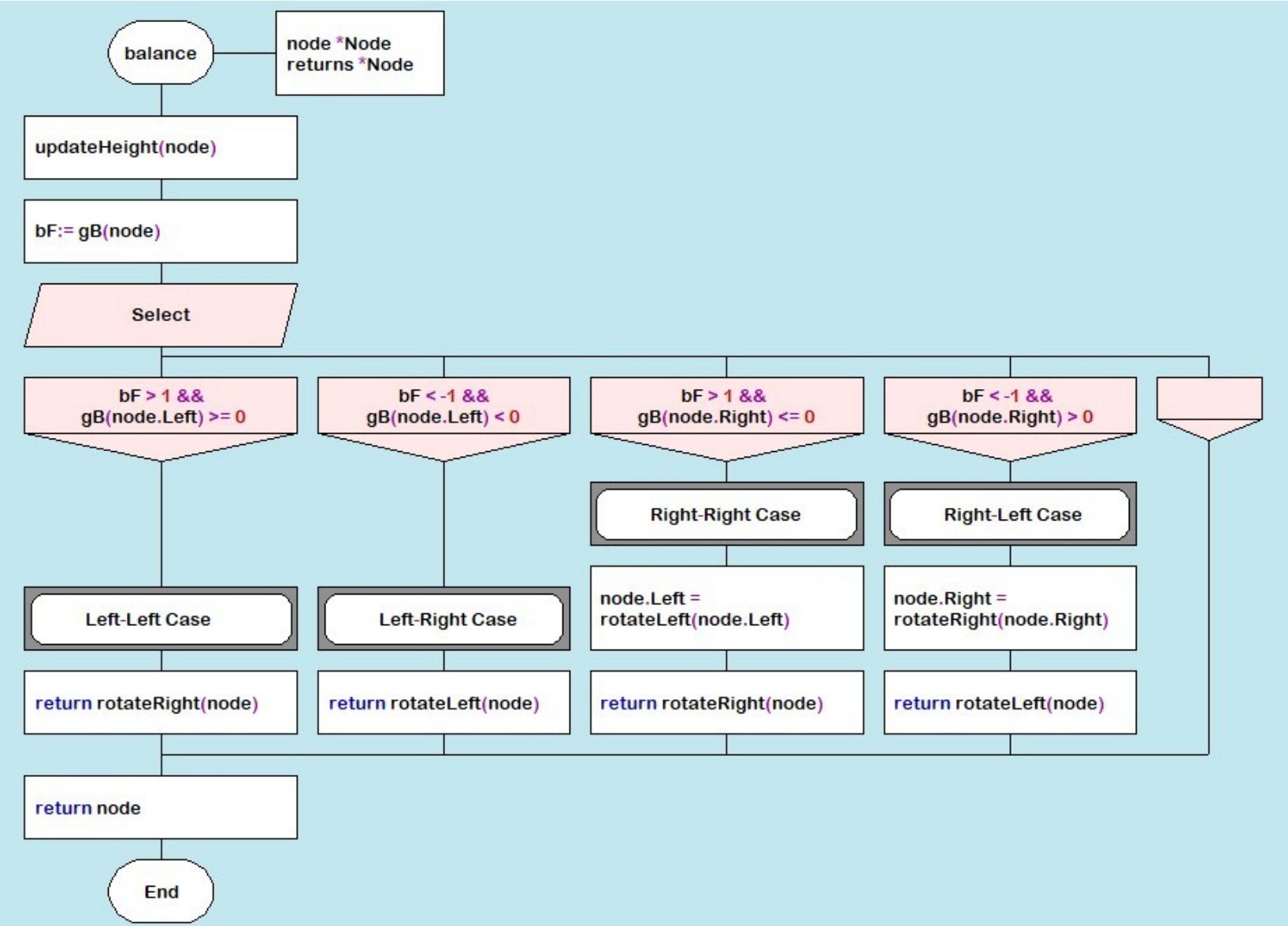


Figure 8.22. Drakon-diagram depicting the rotation method

Depending on the location of the new node relative to the parent node and the balance factor values, the left-hand or right-hand rotation methods are invoked (Figure 8.23):

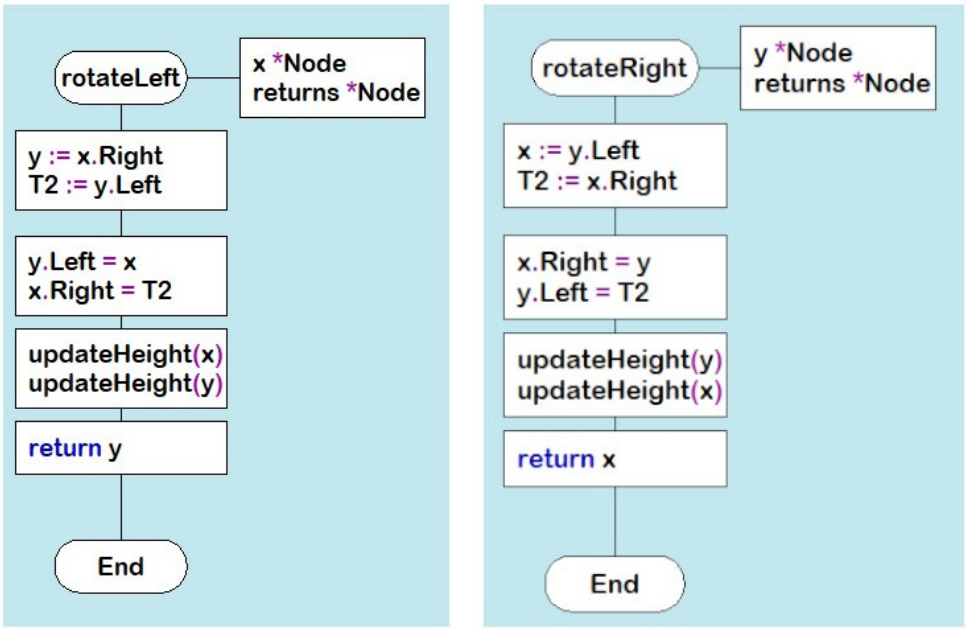


Figure 8.23. Node rotation programme codes

Let's look at the tree rebuilding process in detail as a result of using the rotateInsert(node *node, val int) method on the example of entering the values of three nodes (21,17,16). After receiving the value of one node (16), a "twig" appears instead of the tree that needs to be balanced. In this case the condition in the rotateInsert(node *node, val int) method is met: the balancing factor is 2 and the value of node (16) is less than the parent node (17) and the rightRotate(node) method is called. The balancing process is shown in Figure 8.24:

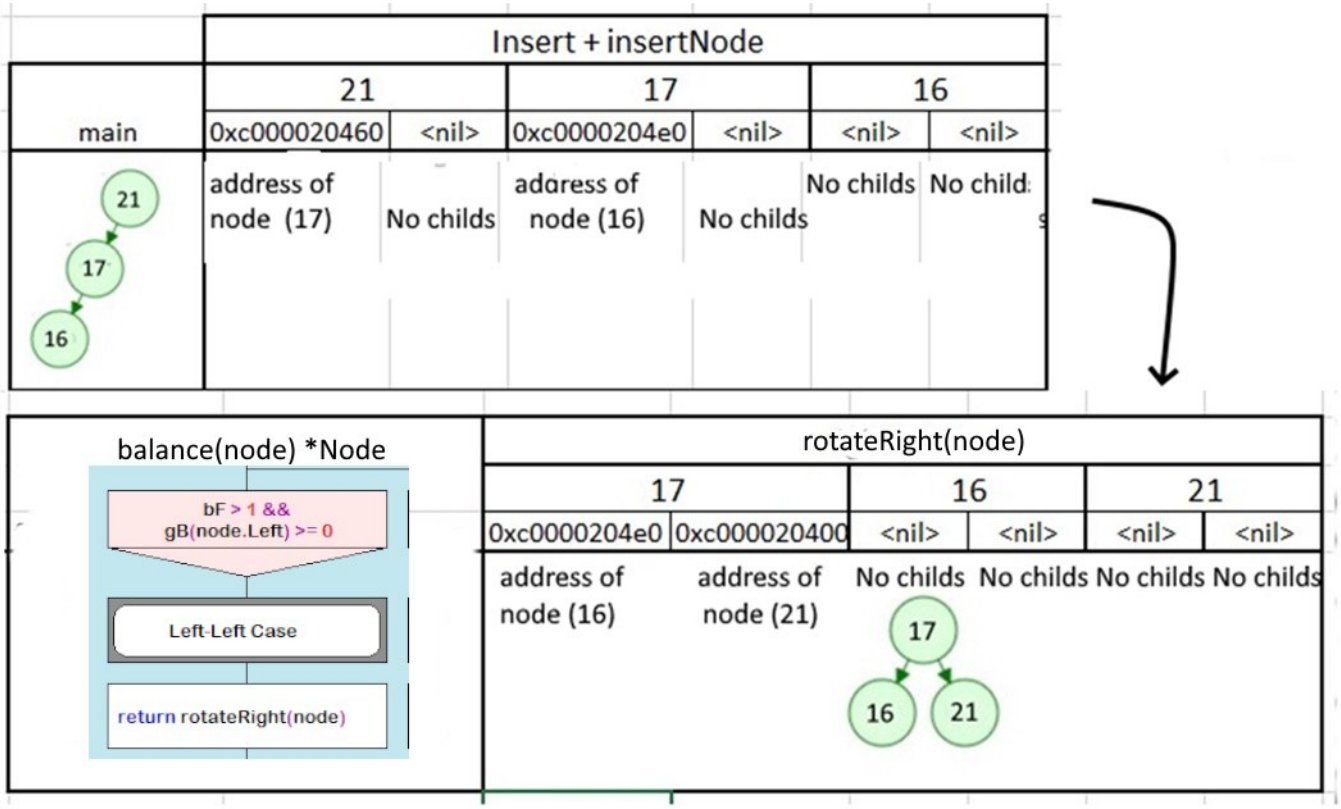


Figure 8.24. Process of rotating assemblies for balancing

As you can see from this figure, the result of the *rotateRight (node)* method is the address fields of elements of the node structure. Recall that this structure consists of four fields: tree height, node value, and addresses of left and right descendant nodes. Figure 8.18 shows the addresses of the descendant nodes below the node value. After rearranging the structure fields node.Left and node.Right, nodes (17) and (21) change and node (17) becomes the root node, node (16) remains unchanged and the former root node (21) becomes the right descendant node (17). As a result of the entry of node (16) the tree becomes balanced.

The process of new nodes arriving and rebuilding the resulting tree by balancing for the input set (21,17,16,11,9,7,5,3) is shown in Figure 8.25.

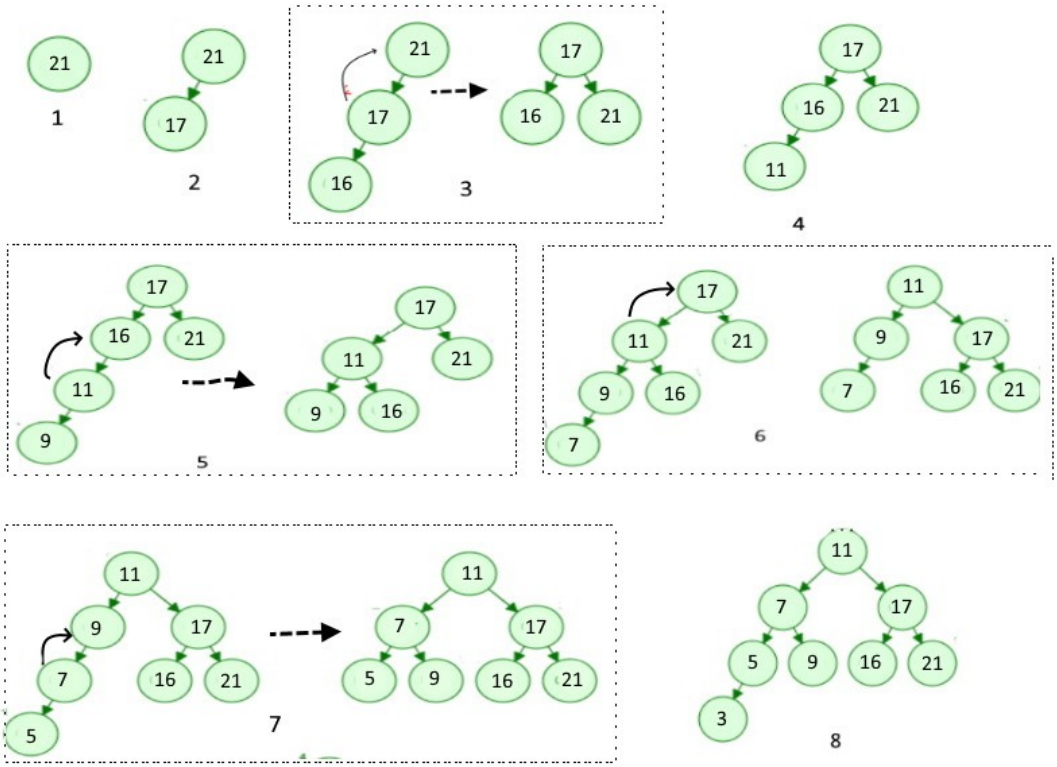


Figure 8.25. Process of building a balanced AVL-tree

The basic operations with AVL-trees are the *findNode(node *Node, value)* and *deleteNode(node * Node, value)* operations. Drakon-diagram of *findNode (node *Node, value)* method is shown in Figure 8.26:

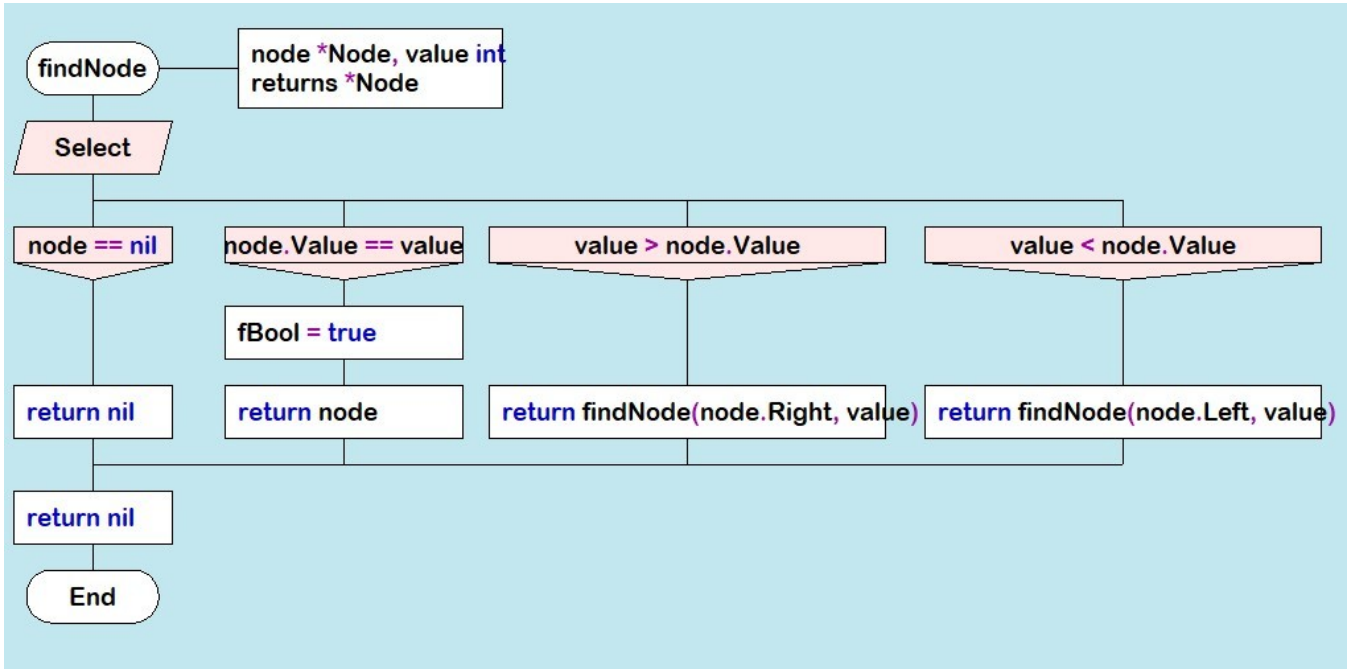


Figure 8.26. Drakon-diagram depicting the finding a node method

8.3.4. Balancing a tree when removing a node

Another basic operation, deleting a node with a given value, consists of the following steps. The node is searched from the root down through the branches to the node to be deleted. The following situations may occur (Figure 8.27):

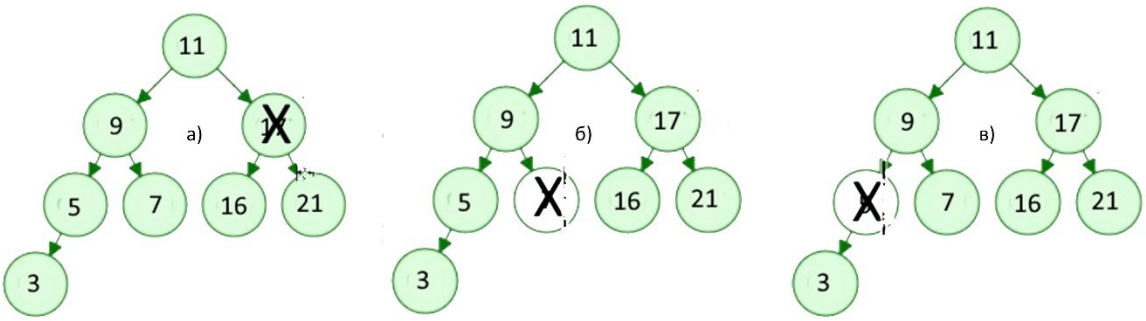


Figure 8.27. Options for nodes to be removed

a) The node to be deleted has two non-empty descendants;

b) The node to be deleted has no descendants;

c) The node to be deleted has one descendant (left or right).

As with the other methods, a node with a specified value is first recursively identified and then one of the options presented is selected. A Drakon-diagram of the algorithm for removing a node with the given value is shown in Figure 8.28.

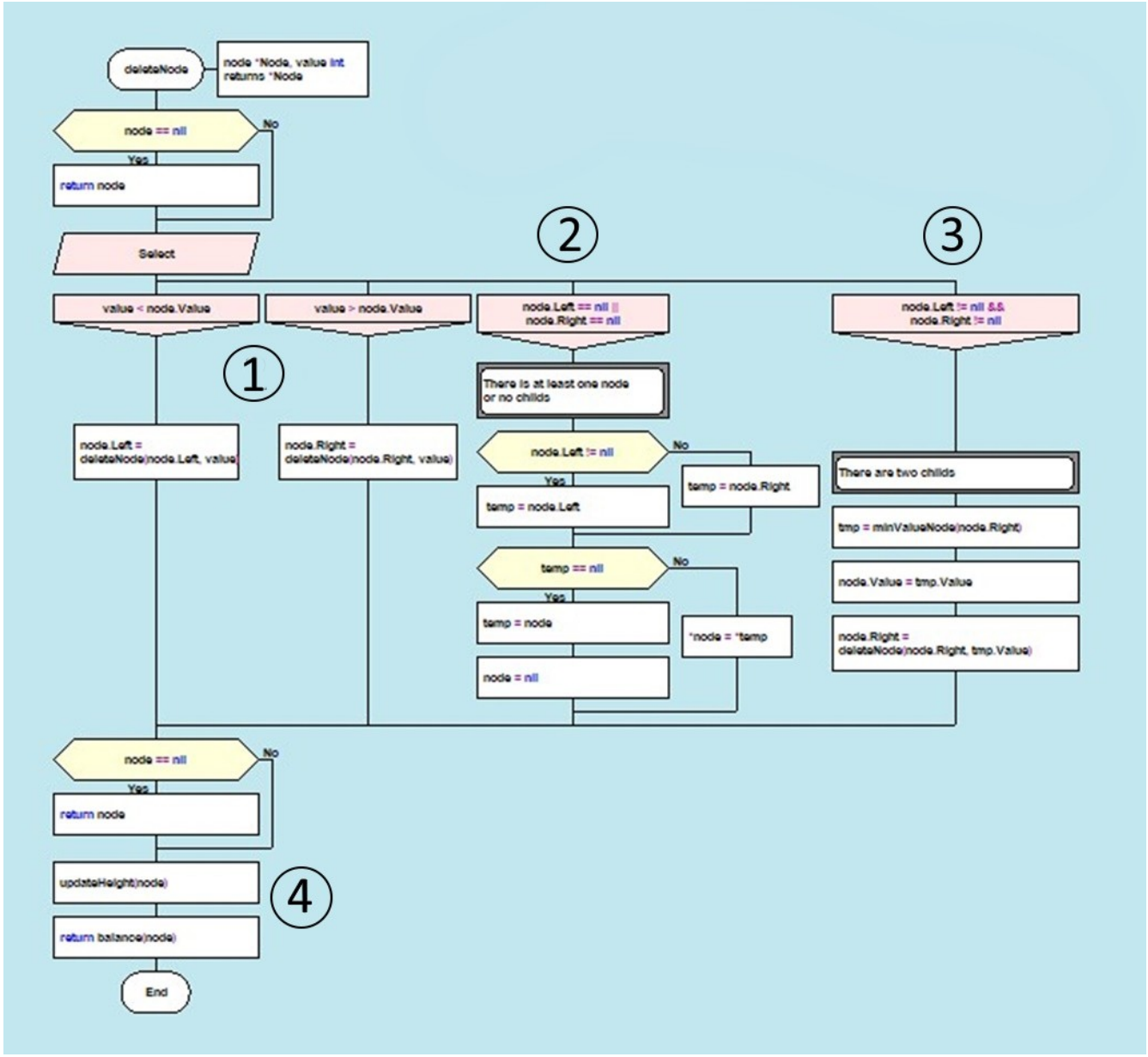


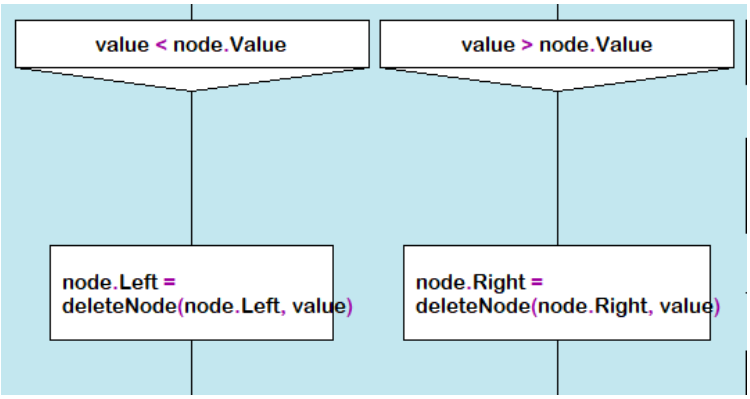
Figure 8.28. "Drakon-diagram depicting an algorithm for removing a node with a given value

Verbal description of the algorithm for deleting a avl-tree node with a specific value.

Fragment 1: Traversing the Tree:

If the value to be deleted is less than the value of the current node, the algorithm recursively calls the `deleteNode` function on the left subtree.

If the value to be deleted is greater than the value of the current node, the algorithm recursively calls the `deleteNode` function on the right subtree.

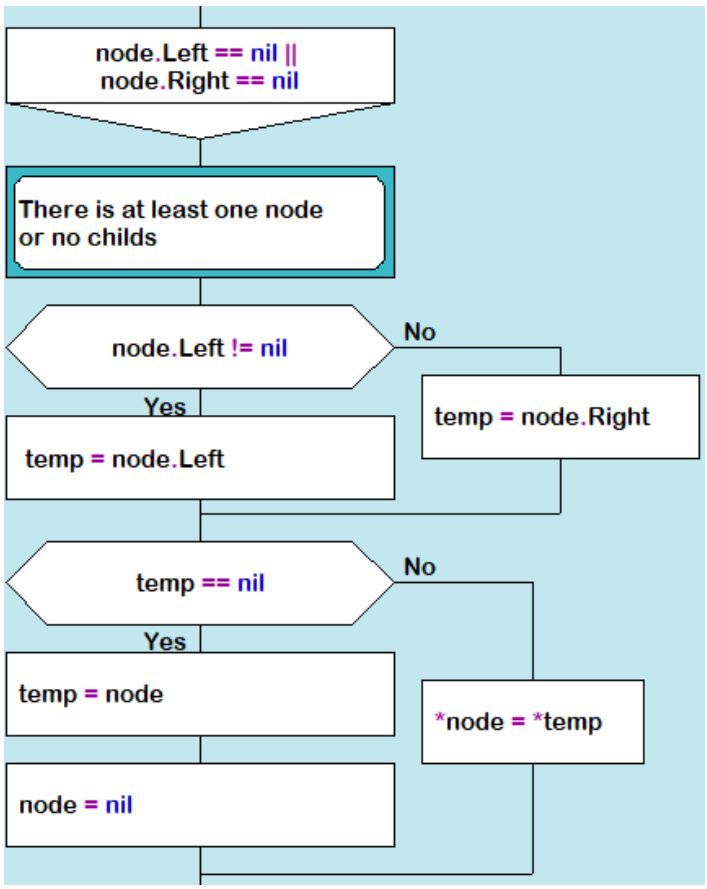


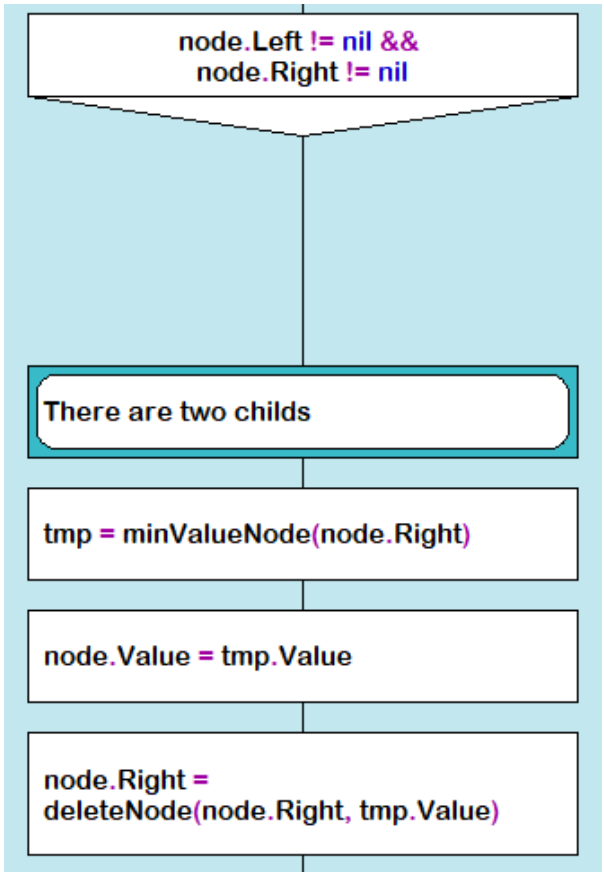
Fragment 2 and 3: Handling the Node to be Deleted:

If the value to be deleted is equal to the value of the current node, the algorithm checks for the following cases:

If the node has no children or only one child, it directly removes the node by adjusting the pointers.

If the node has two children, it finds the minimum value node in the right subtree, replaces the value of the current node with the value of the minimum node, and then recursively deletes the minimum node from the right subtree.

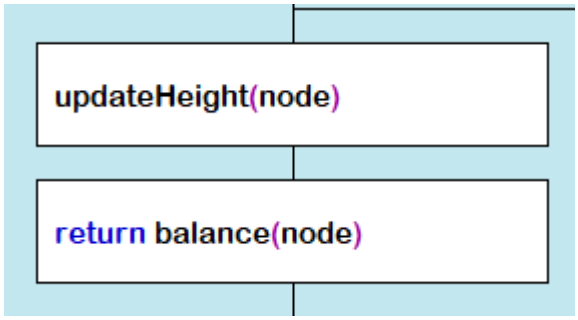

node.Left == nil ||
node.Right == nil
There is at least one node
or no childs
node.Left != nil
No
Yes
temp = node.Right
temp = node.Left
temp == nil
No
Yes
temp = node
*node = *temp
node = nil


node.Left != nil &&
node.Right != nil
There are two childs
tmp = minValueNode(node.Right)
node.Value = tmp.Value
node.Right =
deleteNode(node.Right, tmp.Value)

Fragment 4: Updating the Tree:

After the deletion operation, the algorithm updates the height of the nodes and balances the tree if necessary (*updateHeight(node)*).

The algorithm uses recursion to traverse and modify the tree structure. It returns the modified node after the deletion operation (*return balance(node)*).



The main() program inputs an array of data, searches for a node with a given value, and deletes a node with a given value. The result of implementing the generated code is shown below:

```
Traversing the tree prior to removal:
 preOrder 11 7 5 3 9 17 16 2 1
inOrder 3 5 7 9 11 16 17 21
 postOrder 3 5 9 7 16 21 17 11
The number 3 is present
Number 6 is missing
Removing the node 7
Traversing the tree after deleting number 7
  preOrder 11 5 3 9 17 16 21
```

```
inOrde 3 5 9 11 16 17 21

postOrder 3 9 5 16 21 17 11
```

8.4. Red-black trees

8.4.1. Properties of red-black trees

A red-black tree is a variant of a self-balancing binary search tree in which the nodes are placed according to a certain rule and coloured red or black (Figure 8.29)

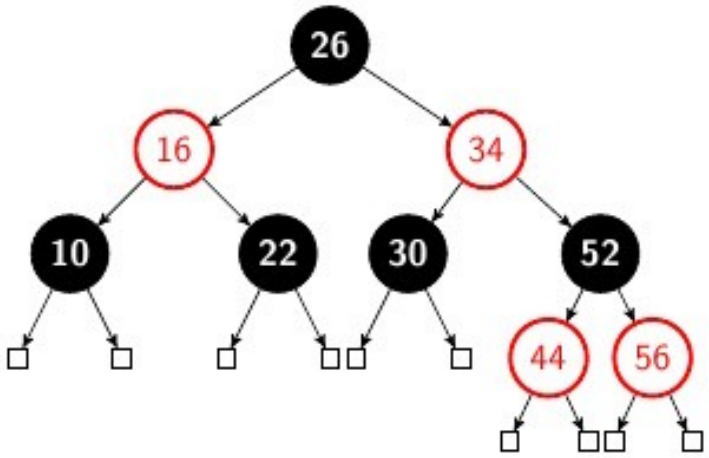


Figure 8.29. Red-black tree

The nodes containing data (in this case, integers) are internal. In addition, red-black trees contain imaginary, "null" nodes associated with tree leaves (Null - in Figure 8.29). Red-black trees satisfy all the properties of a binary search tree and must have the following properties:

1. Each node is colored red or black.

2. The root of the tree is always black.

3. All leaves are black (Null).

4. Both descendants of the red node are black, i.e. there cannot be consecutive red nodes.

5. All simple paths from the node to the descending leaves contain the same number of black nodes.

Unlike AVL trees, where balance is achieved by balancing the heights of the left and right subtrees, red-black balance is achieved by the properties mentioned above. Adding or removing a node from the red-black tree can disrupt the red-black tree properties, and restoring balance is achieved by two operations: repainting the nodes and/or rebuilding the whole tree or its subtrees using specific rotations.The most important of these properties are properties 4 and 5.

A consequence of property 4 is that it requires the same number of red and black nodes in the path between two nodes. Property 5 requires that regardless of the choice of path between two nodes the number of black nodes must be the same. In other words, in the worst case, the height of the tree must not be more than twice the height of the shortest path. In this case the red-black tree becomes fairly balanced.

8.4.2. Inserting a new node and the balancing process

To achieve a balanced red-black tree, the process of inserting a new node is accompanied by a check to ensure that the above properties are satisfied. As red-black is a binary search tree (BST), it first determines where on the tree the node should be placed, colouring it red (Figure 8.30):

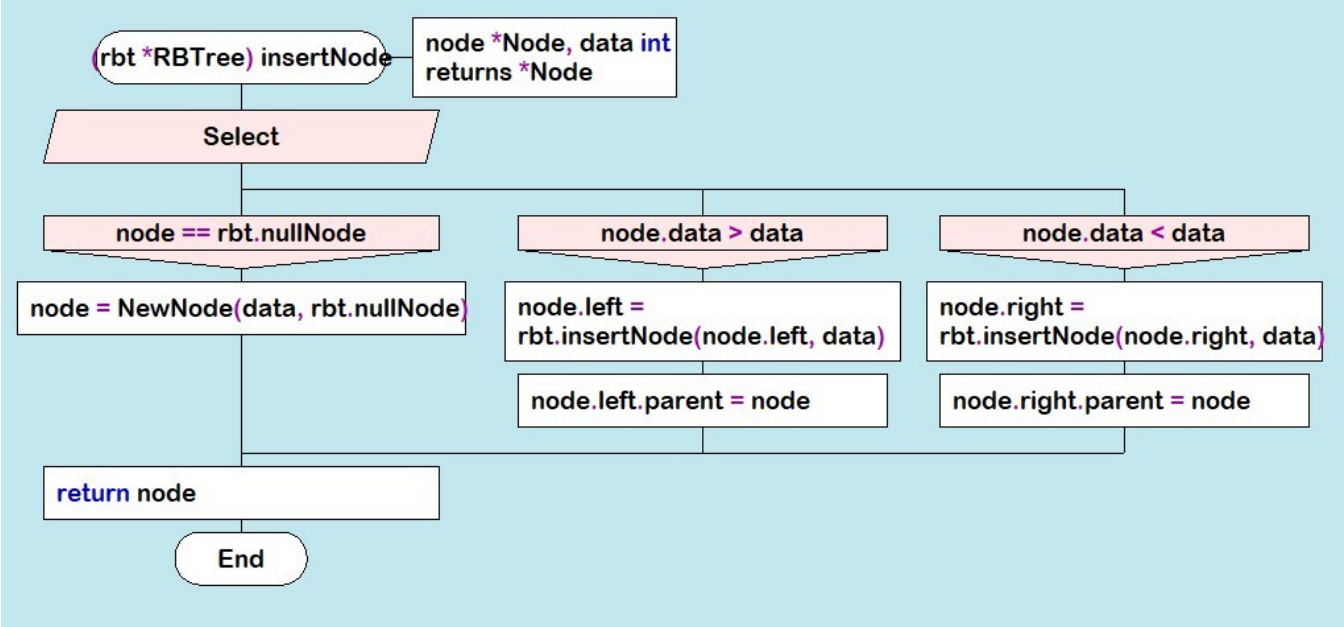
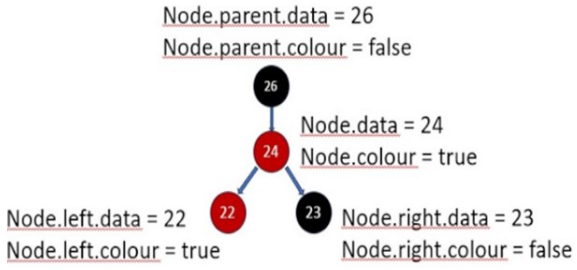


Figure 8.30. Drakon-diagram depicting an algorithm for inserting a node

This method calls the *NewNode(d,nullNode)* function, which returns full information about the new node, with the colour field of type bool initially taking the value red:

```
func NewNode(d int, nullNode *Node) (nd *Node) {
nd = &Node{}
nd.data = d
nd.left = nullNode
nd.right = nullNode
nd.parent = nullNode
nd.colour = true
// new node - red
return
}
```

```
type Node struct {
data int
colour bool
left, right, parent *Node
}
```



Next, consider the node positions in the tree as a result of each new node insertion and determine which properties of the red-black tree are violated. To understand this

process, we introduce the following notation: x - new node, P (Parent) - parent of node x, G (Grandparent) - ancestor (parent of parent), U (Uncle) - uncle of node x (Figure 8.31):

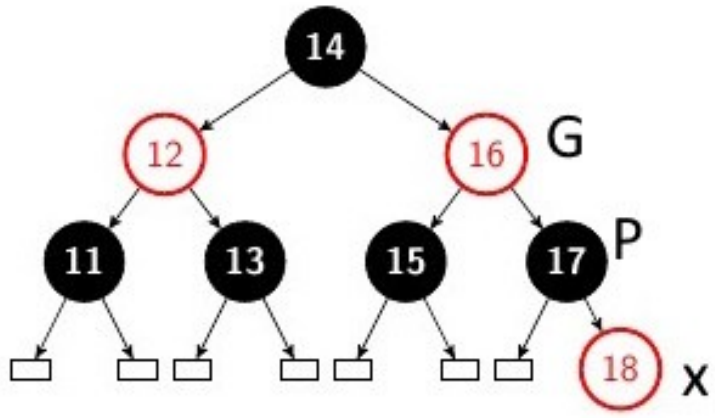


Figure 8.31. Depiction of associated nodes x, P, G, U

When a new element is inserted, it is assigned a red color. To satisfy the first two rules, it is sufficient to simply repaint the new vertices in the desired colour. After each insertion, all the properties of the red-black tree must be checked. If at least one property is not satisfactory, the rotation and colour change operations are performed. Figure 8.32. shows all cases of mutual arrangement of related nodes and fragments of the corresponding Drakon-diagram.

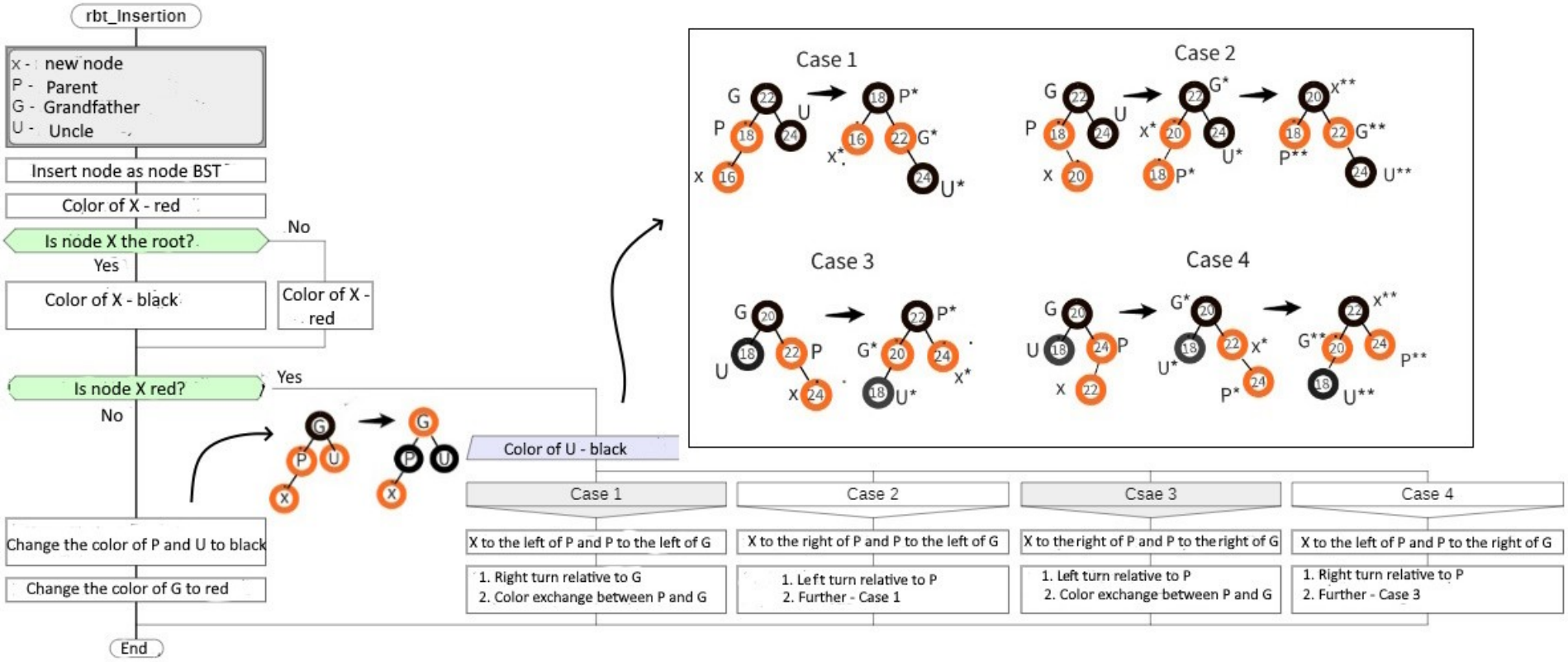


Figure 8.32. Illustration of Red-Black Tree Balancing Algorithms

Consider the balancing process using the example of building a red-black tree from sets of integers that are the keys of the input nodes: {11,12,13,14,15,16,17,18,19} (Figure 8.33):

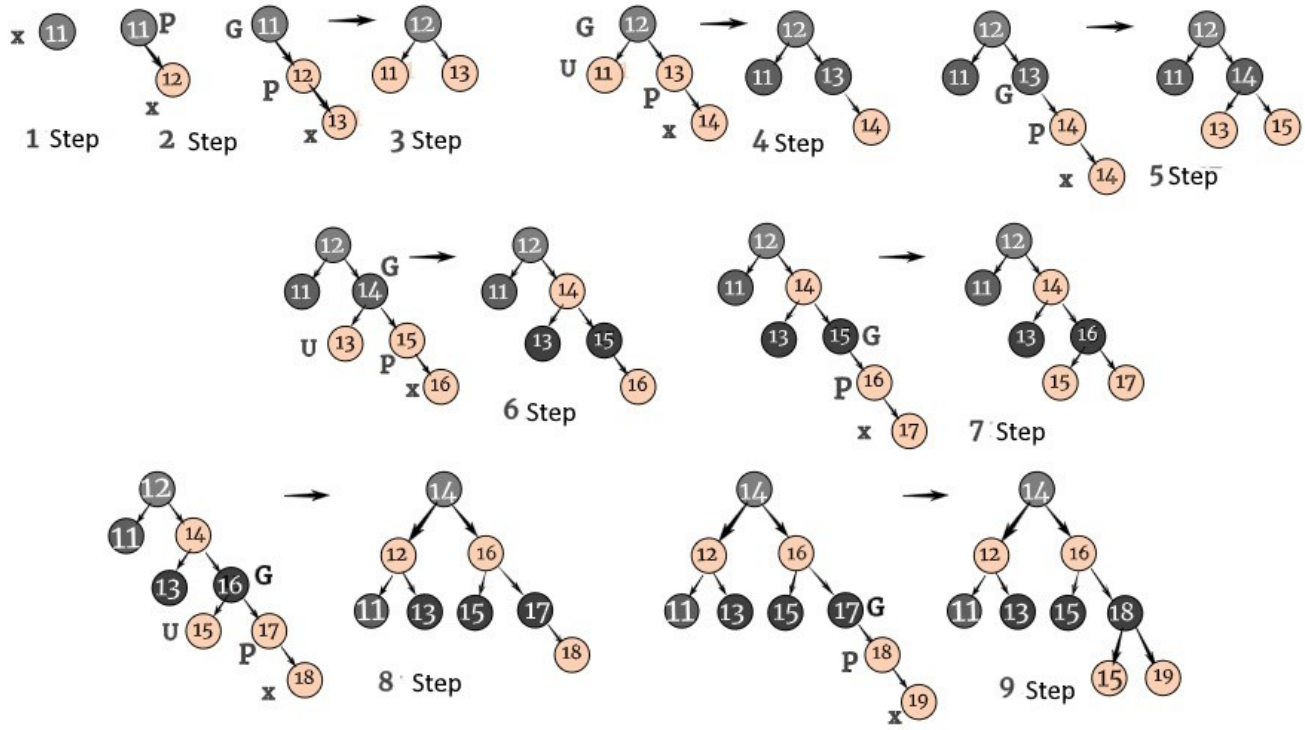


Рис. 8.33. Процесс балансировки красно-черного дерева

The first two steps do not lead to a violation of the properties. Entering a node with key (13) results in a violation of property #4. To restore balance in the third step, a left counterclockwise turn is performed, causing the node (12) to become the root node to black, and the former root node (11) to become the left child. In the next step, there is a situation where three nodes turn red: the new node x(14), the parent P(13) and the uncle U(11), here the balance can be achieved by changing the color of nodes (11) and (13).

Other sets of keys produce other combinations of related nodes (x, P, G, U) of the tree, which are balanced by node rotations and colour changes, as illustrated by the Drakon-diagram (Figure 8.34). This diagram is created in silhouette mode and consists of four branches. In the first branch the root node is colored black. The second branch defines the hierarchy of related nodes (x, P, G, U), the third branch colours these nodes according to the hierarchy. The fourth branch calls the appropriate rotation methods.

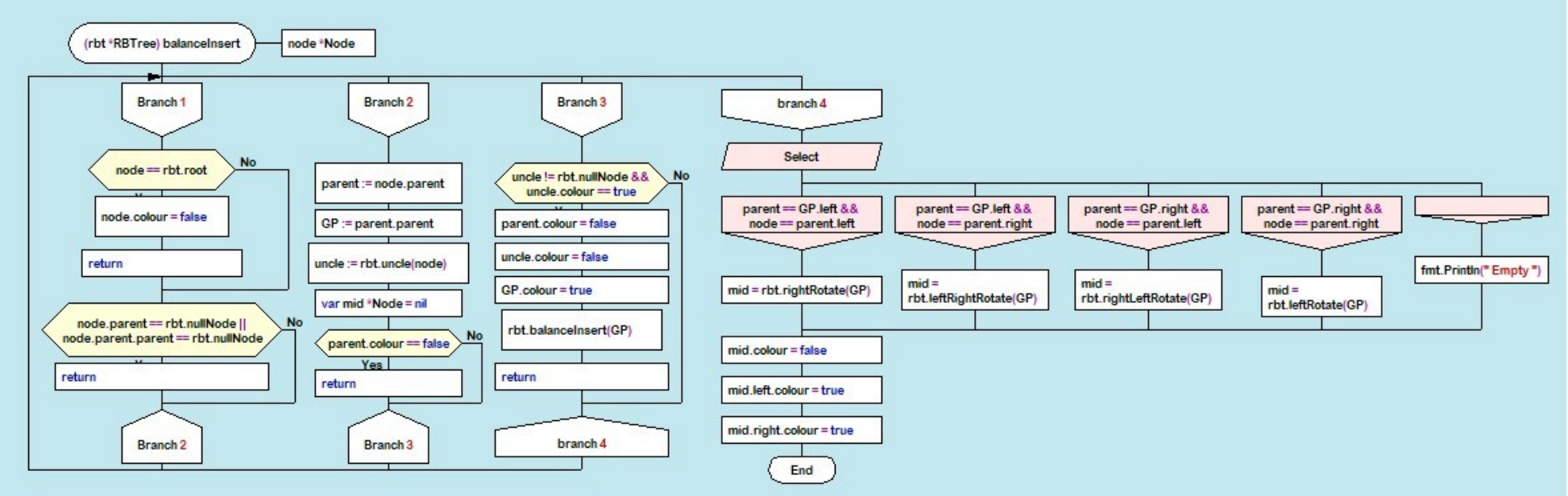


Figure 8.34. Red-black tree balancing Drakon-diagram

parent == GP.left && node == parent.left	Left child of the left child — right rotation.
parent == GP.left && node == parent.right	Right child of the left child — left-right rotation.
parent == GP.right && node == parent.left	Left child of the right child — right-left rotation.
parent == GP.right && node == parent.right	Right child of the right child — left rotation.

Drakon-diagrams of node rotations during the red-black tree balancing when a new node is inserted are shown in Figure 8.35.

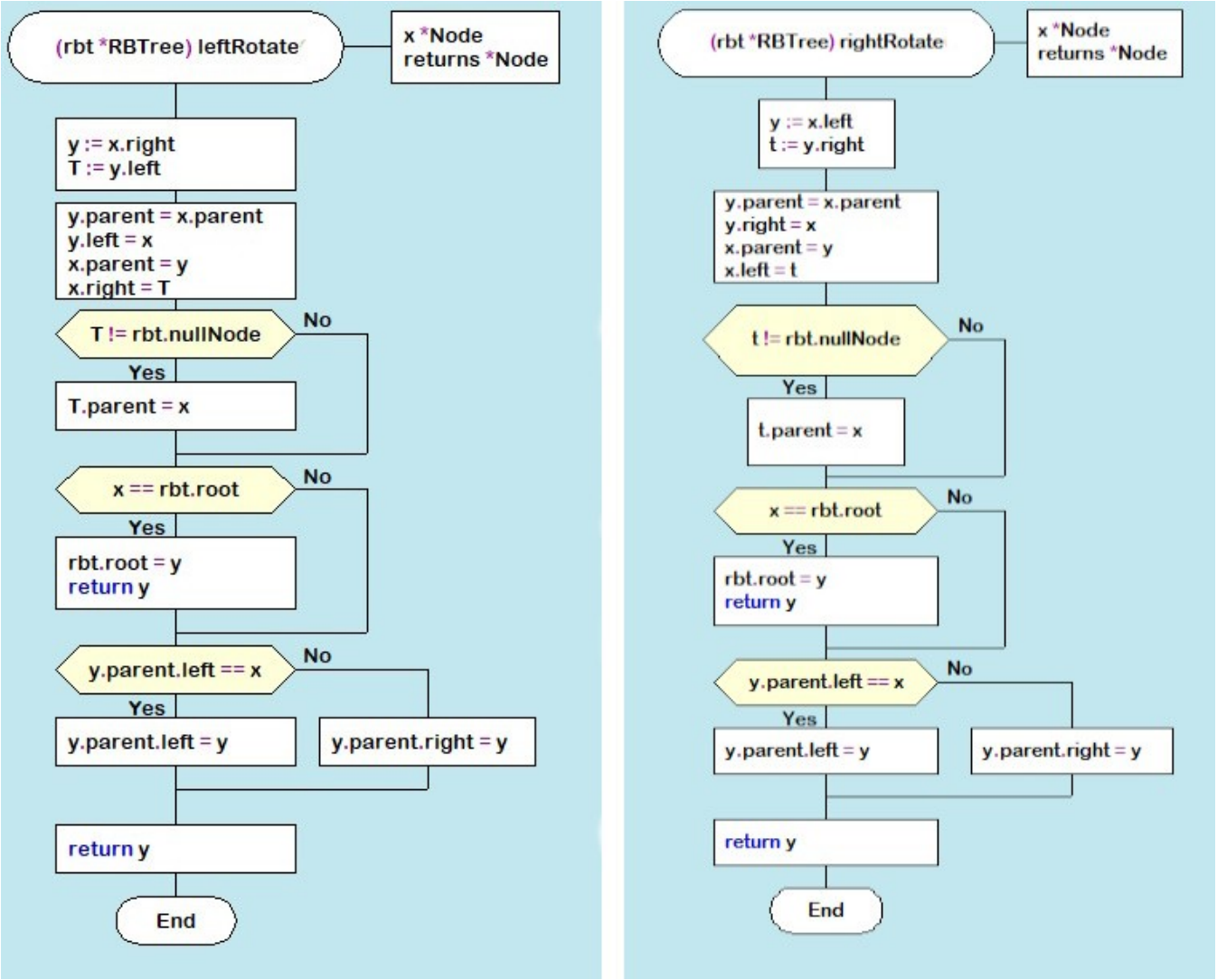


Figure 8.35. Rotation methods in balancing leftRotate and rightRotate

The *rightRotation* algorithm during the balancing process when entering a new node is similar to the *LeftRotate* algorithm.

It's important to note that the structure of a red-black tree can vary based not only on the sequence in which elements are inserted, but also on the specific implementation algorithm utilized. Despite potential changes in the tree's structure, all red-black trees constructed from the same data set will maintain the same "black height" (the count of black nodes from the root to any leaf), a fundamental characteristic of red-black trees. This property ensures the tree's balance, thereby optimizing the efficiency of search, insertion, and deletion operations (Figure 8.36):

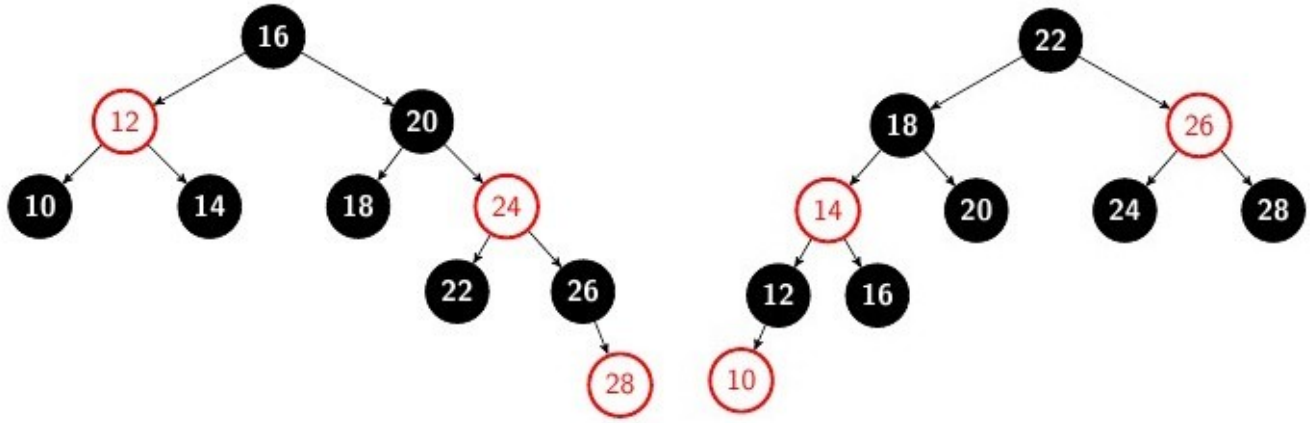


Figure 8.36. Red and black trees built on the same array

a) [10,12,14,16,18,20,22,24,26]; b) [26,24,22,20,18,16,14,12,10]

8.4.3. Deleting a Node from a red-black tree

Removing a node from a red-and-black tree is a complex process because it depends on the location of the node, the presence of children, and the color of the nodes. It is important to remember that all transformations of the tree structure must respect its properties. The Drakon-diagram of the node removal algorithm is shown in Figure 8.36. This algorithm can be divided into three stages.

The first step is to delete the node. The algorithm traverses the nodes of the red-black tree, moving left or right, depending on whether the key of the current node is larger or smaller than the specified value. When a node with the key key is found, it is stored in the variable z. The specified node is then removed from its original location in the tree.

The second step is to fix the double black knot**.** If the node to be removed was black, it may disturb the properties of the red-black tree, since all paths from the root to the leaves must contain the same number of black nodes. To remedy this, the concept of a "double black" node is introduced. If the node x *that replaced the deleted node is black, it becomes a "double black" node.*

The third step is the correction of the double black knot. To fix the "double black" node, the function *balanceDeleteNode (*node * *Node) is called* (Figure 8.37). This feature fixes the "double black knot" by using a series of rotations and repaints. It considers several cases, depending on the colour of the brother of the "double black" node and its descendants. *The balanceDeleteNode (node *Node)* handles all of these cases to ensure that the red-black tree remains balanced after each deletion operation.

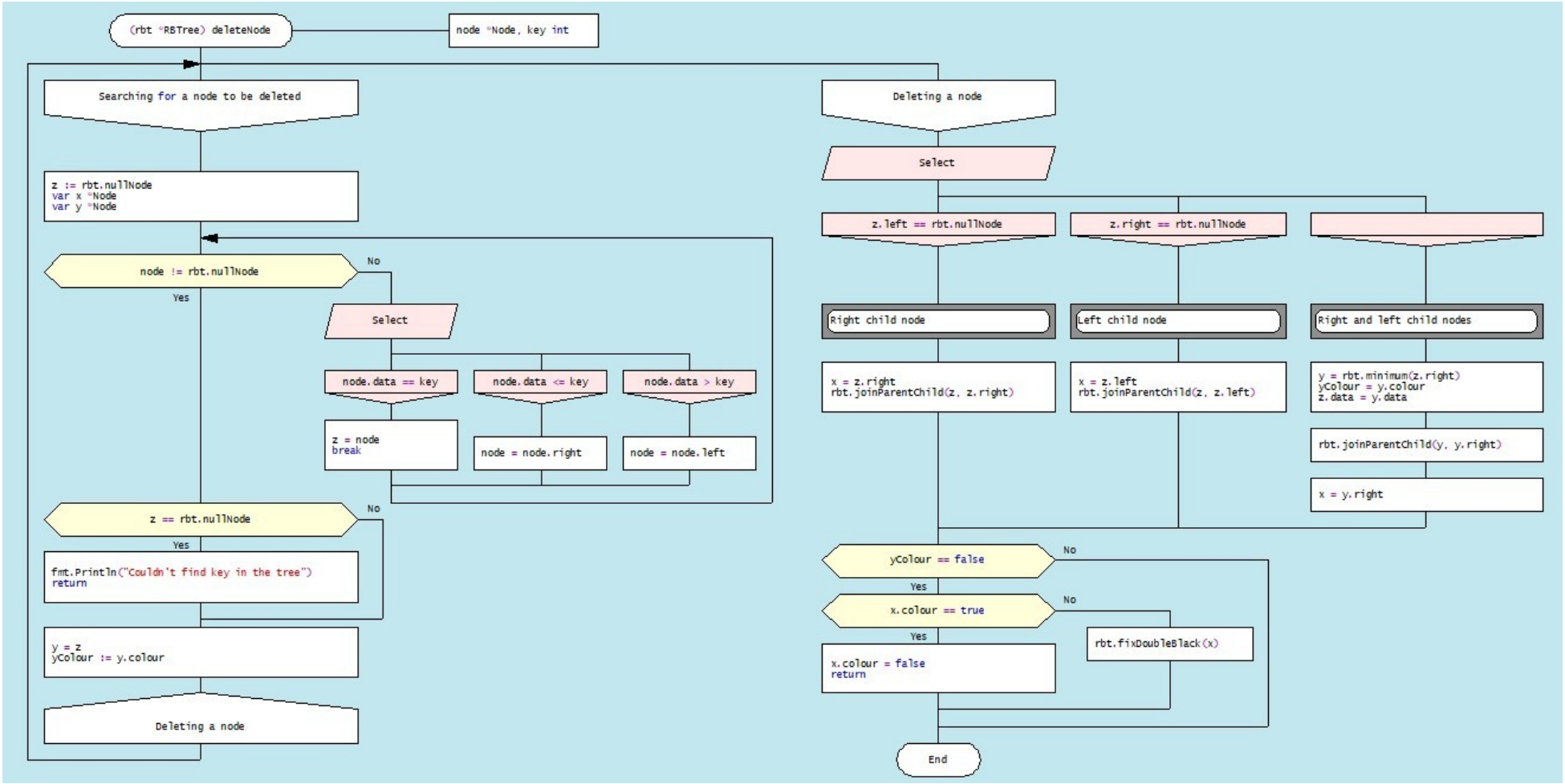


Figure 8.37. Drakon-diagram of algorithm for removing a node from a red-black tree

Consider in more detail the different options of the removal node in the structure of the red-black tree. In the first step, the node is removed simply as a node in the binary search tree. If the replacement node is a "red" node, or if the removed node is a "red" node, the removed node is replaced by another node. There is no need to make any further changes to the tree structure (Figure 8.38).

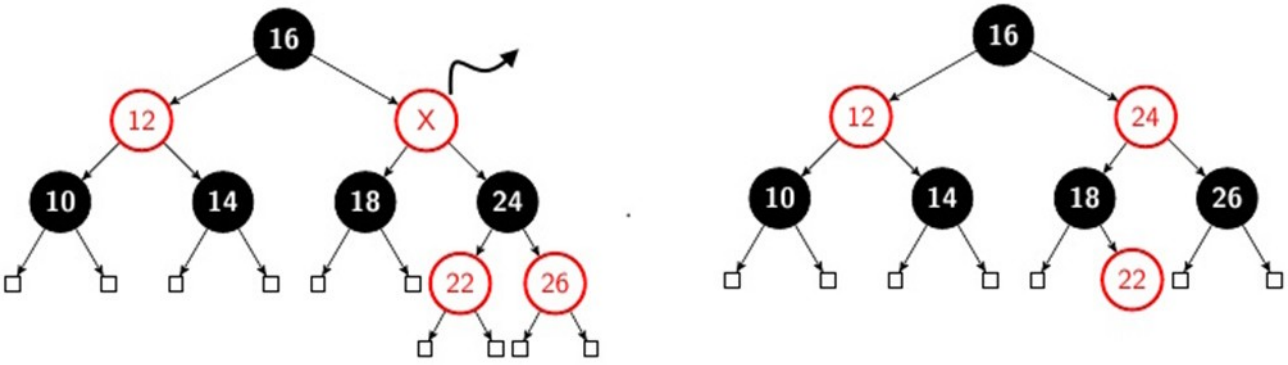
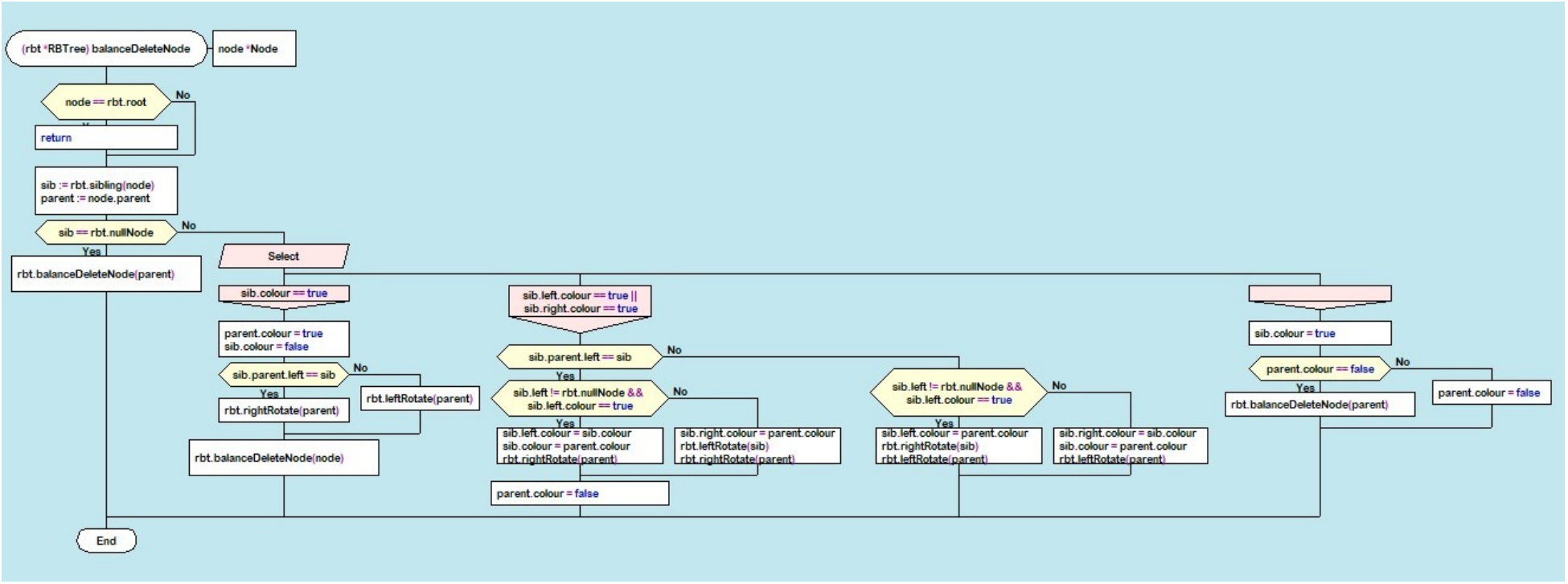


Figure 3.38. Deletion of red node (20)

If the node to be deleted and the node that replaces it are black nodes, a situation called a "double black node" occurs. In this case, some additional operations must be performed to ensure that the properties of the red-black tree are preserved. The Drakon-diagram of this algorithm is shown in Figure 8.39. , 8.40.

If the replacement node is a leaf node (i.e., has no child nodes), we replace the node with a "zero" leaf node and color it black. If the replacement node has one child node, we replace the node with its child node and color it black. If the replacement node has two child nodes, we replace the node with its subsequent node in order, and then delete the next node in order (which is no more than a node with one child node) using the method described above.



=

Figure 8.39. Drakon-diagram of the node structure correction algorithm when removing *balanceDeleteNode*(node)

Let's consider options for restoring the properties of red-black tree. Enter the following notations: the node to be deleted is "x", the child node of the deleted node is "y", and the sibling of the node to be deleted is "s" (Figure 8.40).

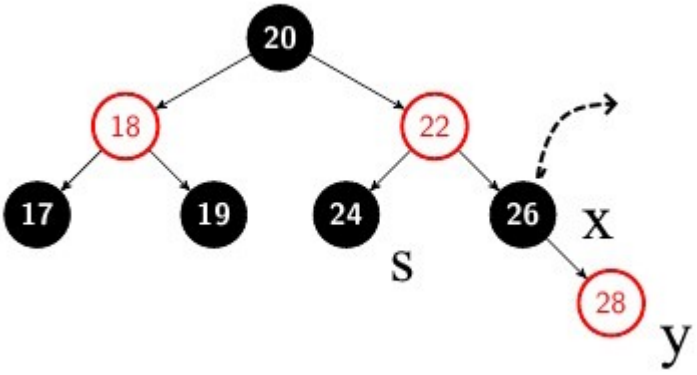


Figure 8.40. Tree node relationship

Option 1: If the child element (y) of the node to be deleted (x) is colored red, then after deletion it should be recolored black, as a result of which the number of black nodes will be restored (Figure 8.41).

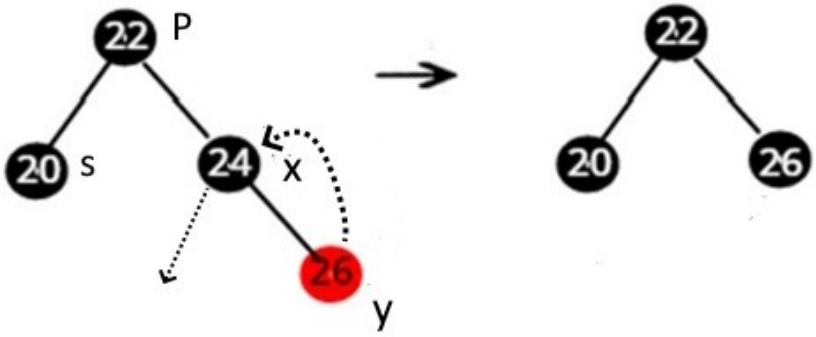


Figure 8.41. Deleting and repainting the child of the node to be removed (y)

Option 2: When the sib(s) of the node to be deleted (x) is colored black, and at least one of the descendants is red, then four different combinations are possible:

Option 2(a): The brother .(s) of the node to be deleted (x) is black. In this case, the brother (s) is the left descendant of the parent (P), and the left child of the brother is coloured red. This is the so-called "left-left *configuration"*, in which the balance is achieved by the right rotation, after which the color of the child should be replaced by black (Figure 8.42).

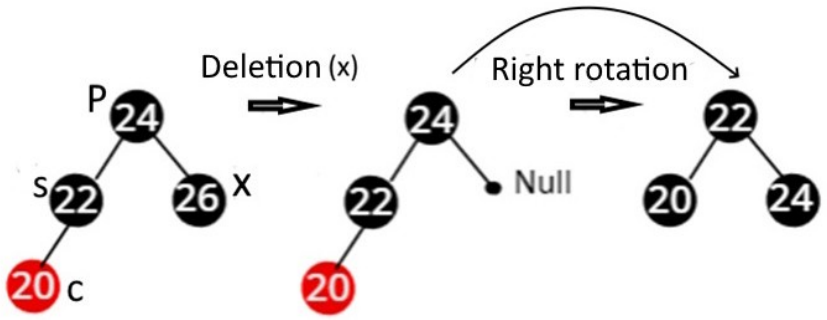


Figure 8.42. Left-Left node delete configuration

Option 2(b): Mirror image of case 2(a). The "s" brother is his parent's right child, and his right child is the red child. The right-right configuration performs a left-hand rotation. The child node is then changed to black. As a result, the black node counter property is restored (Figure 8.43).

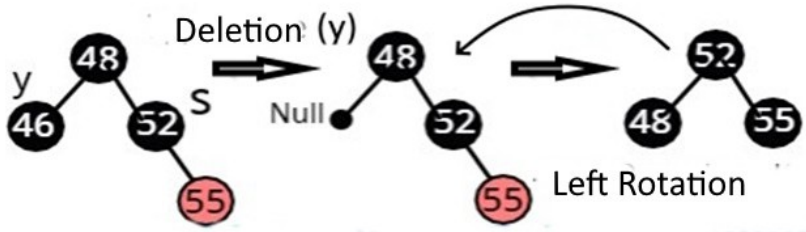


Figure 8.43. Right-Right configuration of node deletion

Option 2 (c). The brother (s) of the remote node (y) is black. The brother(s) is the left offspring of his parent, and his right child is red. This is a left-to-right configuration, so a left rotation is performed followed by a right rotation. The child node is colored black (Figure 8.44):

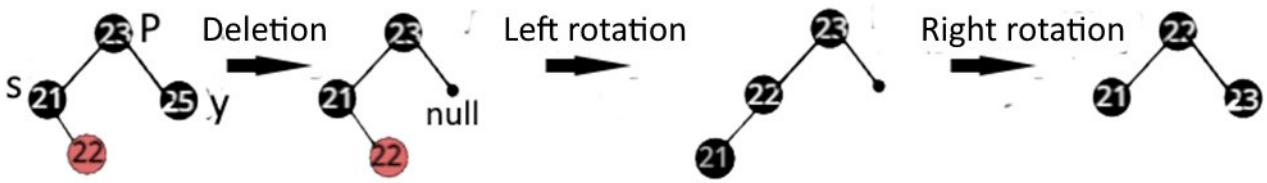


Figure 8.44. Left-Right node delete configuration

Option 2(d): The brother (s) of the remote node (y) is colored black. The brother (s) is the right child of his parent (P) and his left child is the red child. A right-left rotation is then performed, followed by a left-hand rotation. The color of the child node changes to black (Figure 8.45).

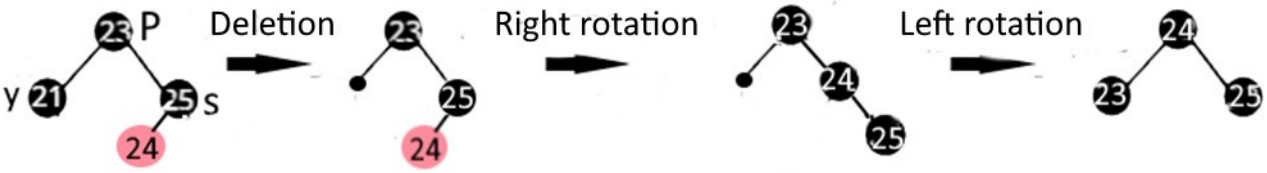


Figure 8.45. Right-Left node delete configuration

Option 3: If the node (y) to be removed and its brother(s) are colored black and both of its descendants are missing, i.e. black by definition, then you need to recolor the children red and recursively add black to the parent. If the parent was red, then it became black. If the parent was black, it would become double black. If the parent is the root, then it remains black. For example, below is a case where the node (y) to be deleted and its brother (s) are black (Figure 8.46):

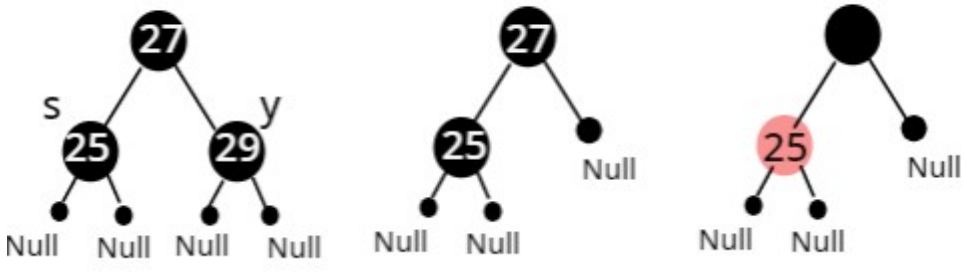


Figure 8.46 The process of rebuilding a tree with two black nodes (y) and (s)

Option 4: If the sib(s) of the node to be removed (y) is colored red, then a rotation is performed to achieve the balance of the tree. The sister nodes are then repainted. For example, consider the following option (Figure 8.47):

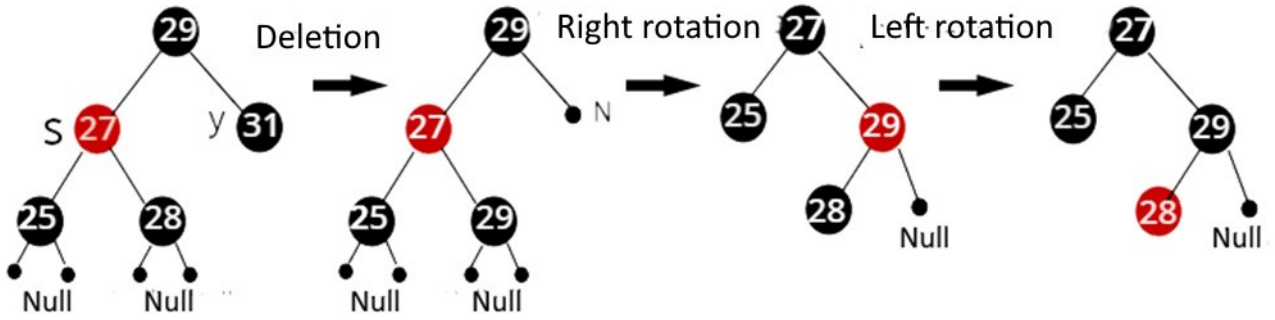


Figure 8.47. The process of rebuilding a tree with two nodes (y) and (s) of different colors

A general view of the program, including all the Drakon-diagrams of the implementation of the algorithms of basic and auxiliary functions, is shown in Figure 8.48:

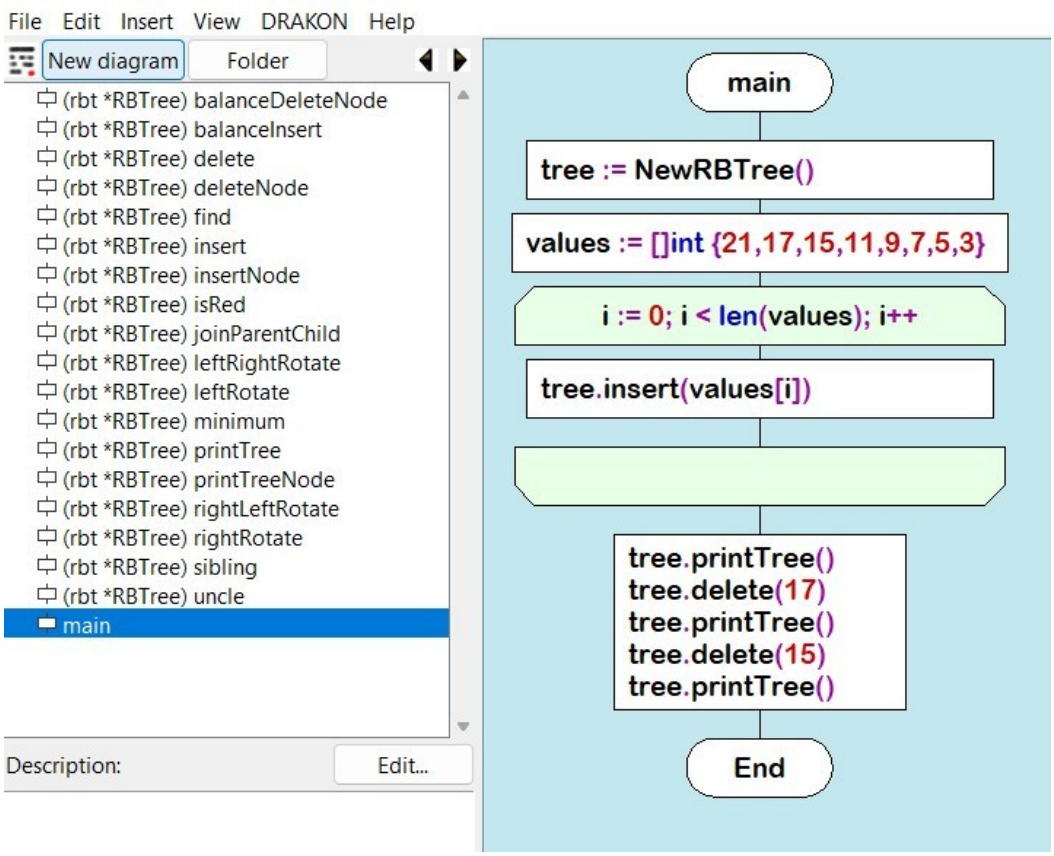


Figure 8.48. General view of the program for implementing the basic functions of a red-black tree

Descriptions of the types used in the program are shown in Figure 8.49:

```
=== header ===
package main

import "fmt"

type RBTree struct {
        root     *Node
        nullNode *Node
}

func NewRBTree() (rbt *RBTree) {
        rbt = &RBTree{}
        rbt.nullNode = NewNode(0, nil)
        rbt.nullNode.colour = false
        rbt.root = rbt.nullNode
        return
}

type Node struct {
        data                int
        colour              bool
        left, right, parent *Node
}
=== footer ===
```

Figure 8.49. Description of types of variables in program *rbtAll.drn*

Drakon-diagrams of the accompanying algorithms for implementing the basic functions of red-black trees are presented below:

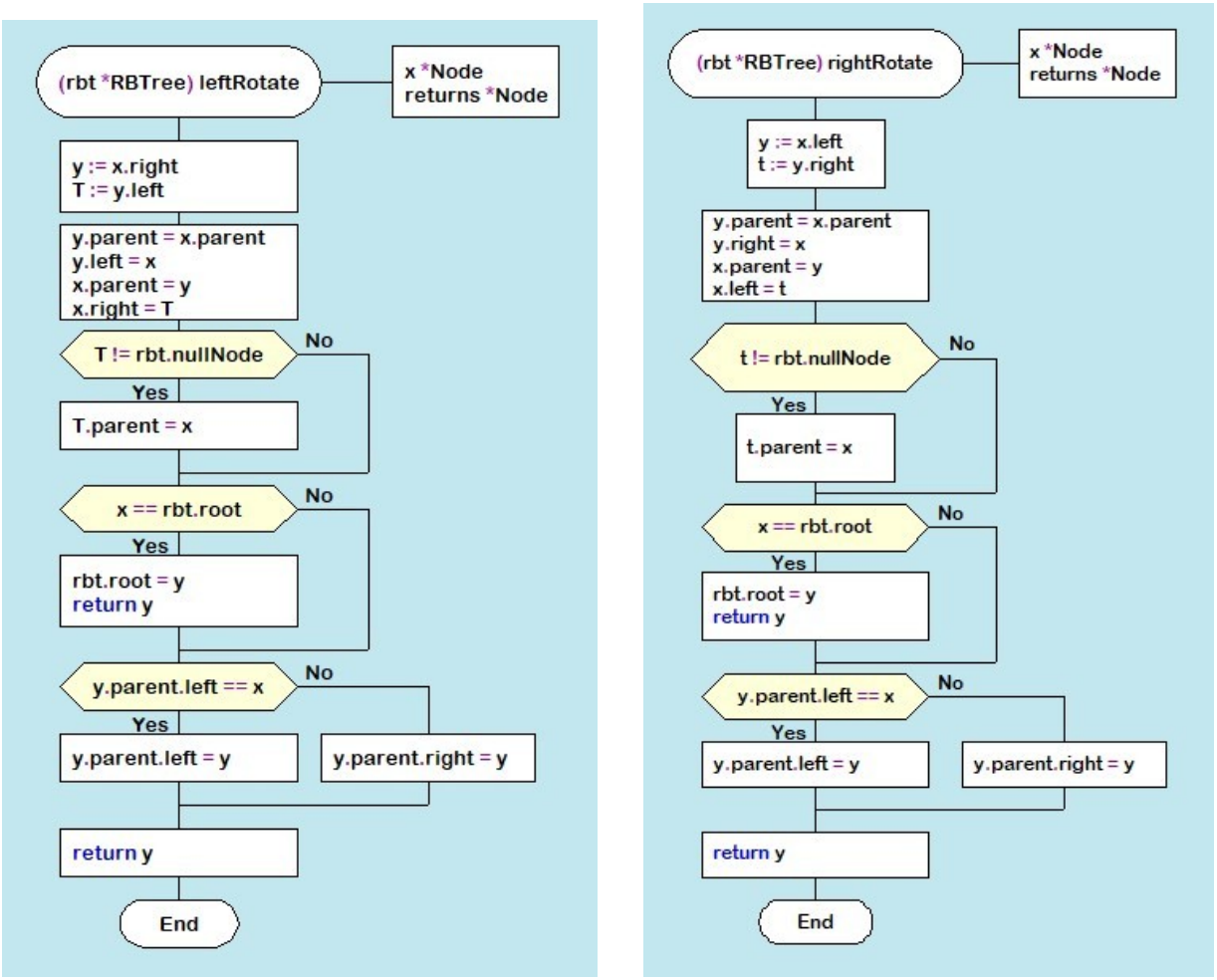


Figure 8.50. Дралон-diagrams of red-black tree node rotation algorithms

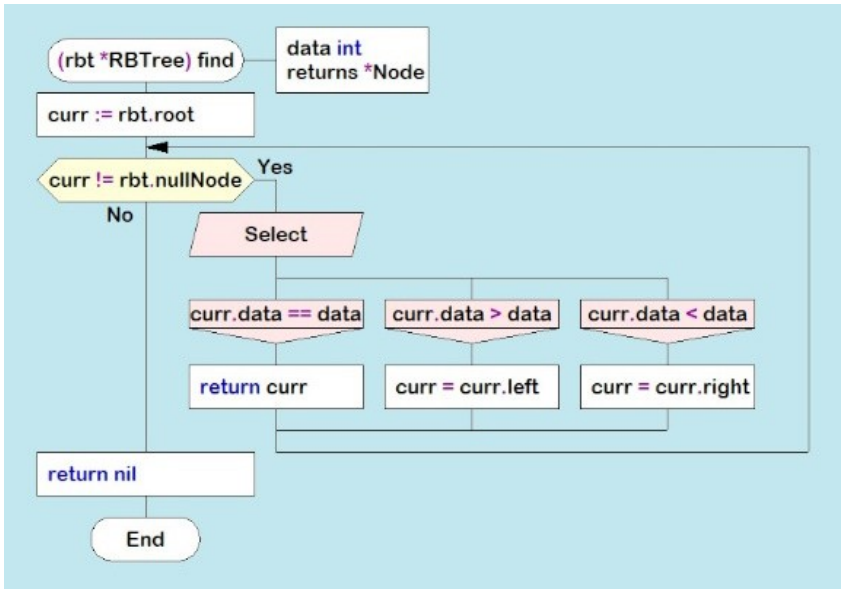


Figure 8.51. Drakon-diagram of node finding algorithm *find(node)*

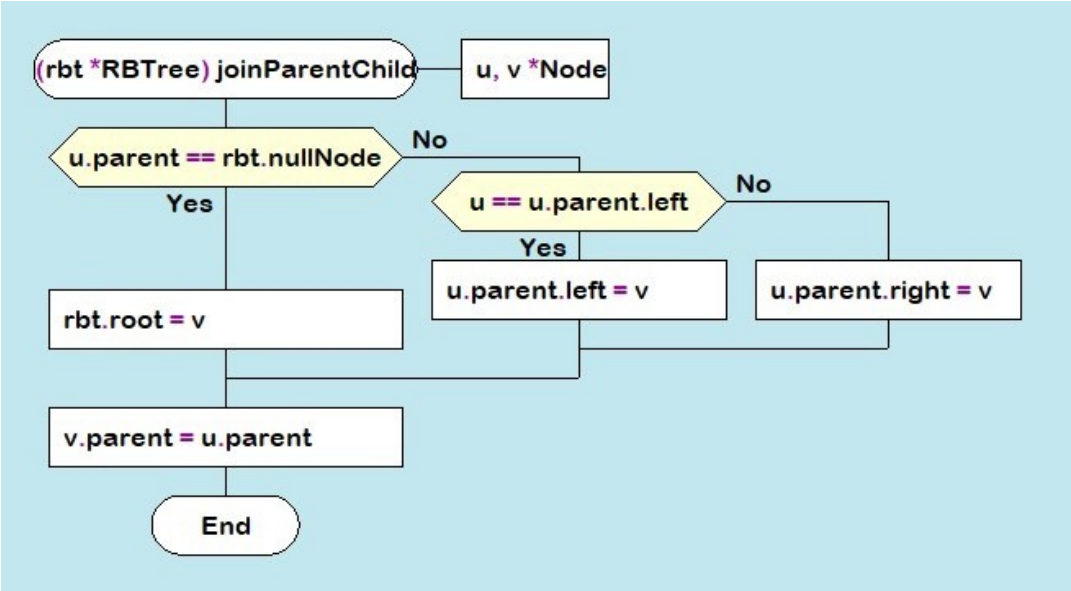


Figure 8.52. Drakon-diagram of algorithm of correct merging of two nodes

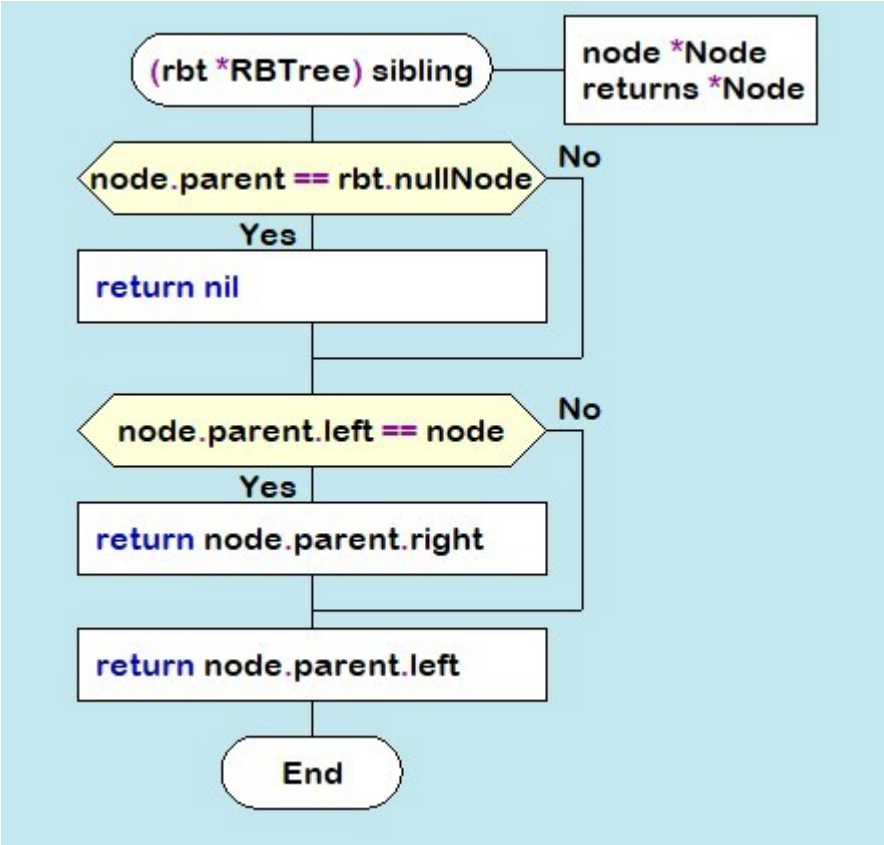


Figure 8.53. Drakon-diagram of finding a sibling node in a red-black tree

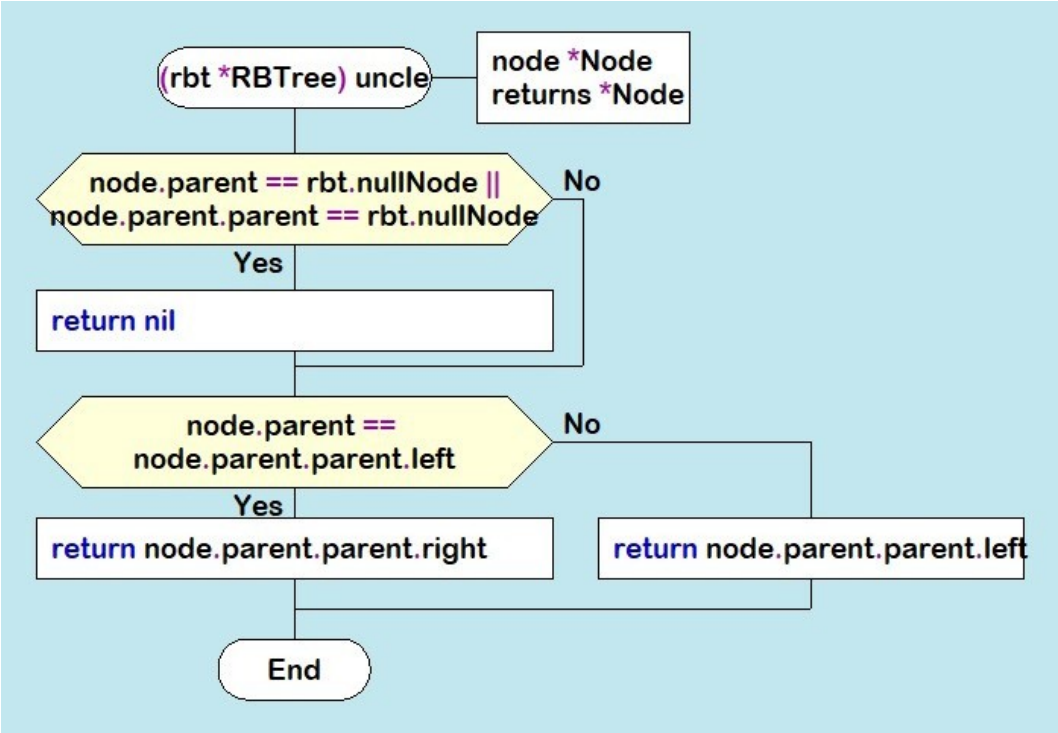


Figure 8.54. Drakon-diagram of finding the uncle node algorithm of a given node

8.5. Space and Time Complexity of different tree species

The space and time complexity of search, insertion, and deletion operations in a BSTree depends on the level to which the tree is balanced. Let's consider estimates of the complexity of basic search, insertion and deletion operations for various types of trees (BSTree, AVL-tree, red-black tree).

BSTree

The space and time complexity of search, insertion, and deletion operations in these trees largely depends on their balance level. For a BST, the best-case space complexity for search, insertion, and deletion operations is O(1), as these operations do not require additional space that scales with the number of nodes in the tree. The worst-case space complexity is O(n), which occurs when the tree is highly unbalanced, leading to a linear structure similar to a singly linked list. The average-case space complexity for a balanced BST is O(log n), where n represents the number of nodes in the tree.

The time complexity for search, insertion, and deletion operations in a BST can be summarized as follows: The best-case time complexity for a balanced BST is O(log n). The worst-case time complexity for an unbalanced BST is O(n). The average-case time complexity for search, insertion, and deletion operations in a balanced BST is also O(log n).

AVL-tree

For an AVL Tree can be summarized as follows:

the best-case space complexity for search, insertion, and deletion operations is O(log n), as these operations do not require additional space that scales with the number of nodes in the tree. The worst-case space complexity is O(n), which occurs when the tree is highly unbalanced, leading to a linear structure similar to a singly linked list. The average-case space complexity for a balanced AVL-tree is O(log n), where n represents the number of nodes in the tree.

The time complexity for search, insertion, and deletion operations in a AVL-tree can be summarized as follows: the best-case time complexity for a balanced AVL-tree is O(log n). The worst-case time complexity for an unbalanced BST is O(n). The average-case time complexity for search, insertion, and deletion operations in a balanced AVL-tree is also O(log n)

Red-Black Tree

For an AVL Tree can be summarized as follows:

The best-case space complexity for search, insertion, and deletion operations is O(1), as these operations do not require additional space that scales with the number of nodes in the tree. The worst-case space complexity is O(log n), which occurs when the tree is highly unbalanced, leading to a linear structure similar to a singly linked list. The average-case space complexity for a balanced red-black tree is O(log n).

The time complexity for search, insertion, and deletion operations in a red-black tree can be summarized as follows: the best-case time complexity for a balanced red-

black tree is O(log n). The worst-case time complexity for an unbalanced BST is O(n). The average-case time complexity for search, insertion, and deletion operations in a balanced AVL-tree is also O(log n).