[Deletion in binary Search tree examples(BST) using Golang (golangdev.in)](https://www.golangdev.in/2021/09/delete-node-from-binary-search-treebst.html)

[AVL Trees: Rotations, Insertion, Deletion with C++ Example (guru99.com)](https://www.guru99.com/avl-tree.html)

[**https://appliedgo.net/balancedtree/**](https://appliedgo.net/balancedtree/)

**РАЗДЕЛ 8. АЛГОРИТМЫ БИНАРНЫХ ДЕРЕВЬЕВ**

**8.1. Представление бинарных деревьев**

Двоичные деревья как абстрактный тип данных были рассмотрены в первом разделе, здесь речь пойдет о компьютерной реализации этого вида структур данных с помощью визуального алгоритмического языка ДРАКОН и языка программирования Golang. Напомним основную терминологию в отношении двоичных деревьев:

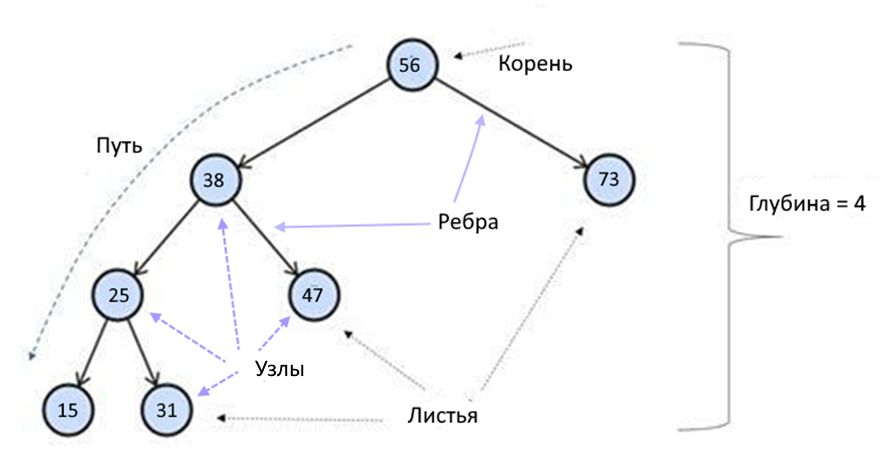


Рис. 8.1. Основная терминология дерева

*Корень*: корень дерева является единственным узлом без каких-либо входящих рёбер. Это верхний узел дерева;

*Узел*: это фундаментальный элемент дерева. Каждый узел имеет данные и две ссылки, которые могут указывать на нуль или его потомков;

*Ребро*: Это также фундаментальная часть дерева, которая используется для соединения двух узловые точки.

*Путь*: Путь - это упорядоченный список узлов, соединенных рёбрами.

*Лист*: Листовой узел - это узел, не имеющий потомков.

*Высота дерева*: Высота дерева - число рёбер на самом длинном пути между корнем и листом.

*Уровень узла*: Уровень узла - число рёбер на пути от корневой узел этого узла

Информационная структура двоичного дерева организуется следующим образом (рис.8.2):

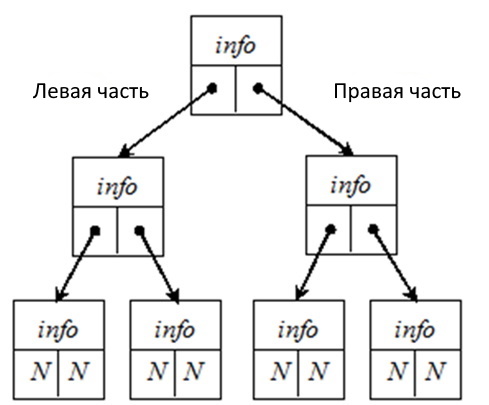


Рис.8.2. Структура бинарного дерева (*info* – значение (ключ), (N – NULL))

## В учебной литературе рассматриваются различные виды бинарных деревьев, среди которых наиболее значимой является классификация на **основе значений узлов:**

* двоичное дерево поиска;
* дерево АВЛ (AVL);
* красное черное дерево.

**8.2. Двоичное дерево поиска**

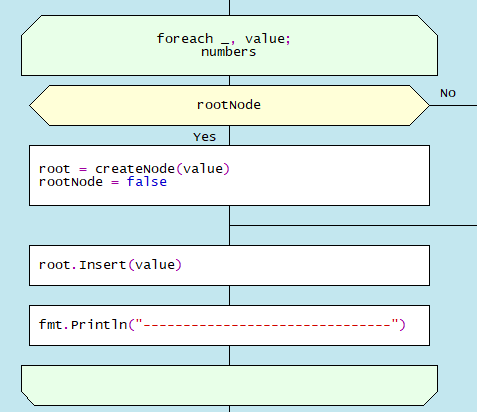
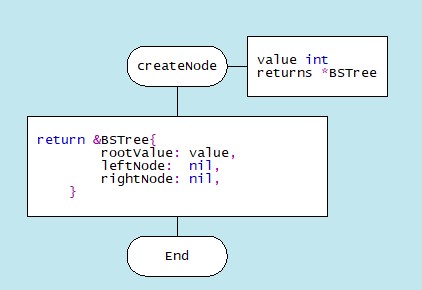
|  |  |
| --- | --- |
| Пути доступа к программным файлам проекта bstAll | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

8.2.1. Построение бинарного дерева

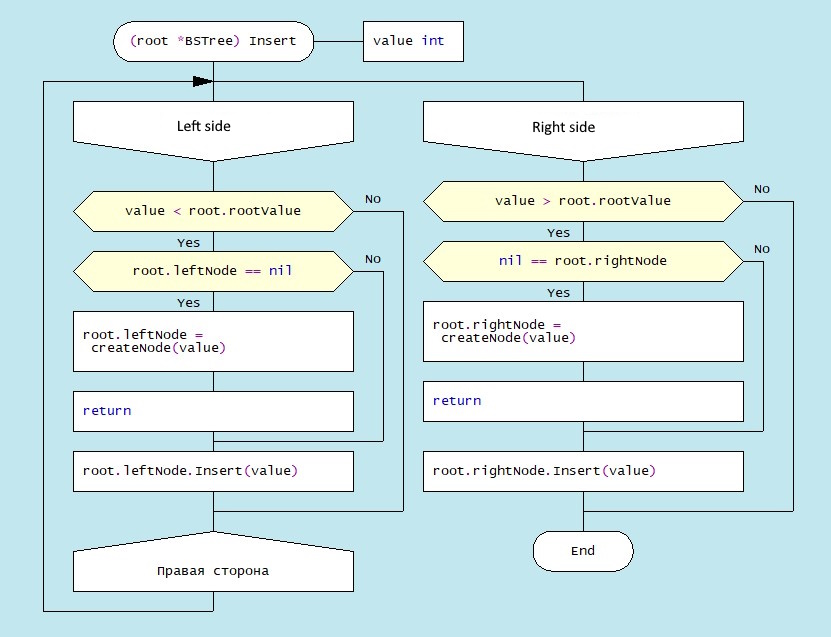
Если дерево организовано таким образом, что для каждого узла все значения узлов его левого поддерева меньше значений этого узла, а все значения его правого поддерева – больше, оно называется *бинарным деревом поиска* (Binary Search Tree). Двоичное дерево поиска по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. Двоичное дерево поиска обладает следующими свойствами:

* дерево состоит из узлов, которые сохраняют уникальные значения;
* каждый узел имеет ноль, один или два дочерних узла;
* один из узлов обозначается как корневой узел, который находится в верхней части структуры дерева;
* каждый узел имеет только один родительский узел, за исключением корневого узла, который не имеет родительского;
* значение каждого узла больше, чем значение его левого потомка, но меньше, чем значение его правого потомка;

Построение бинарного дерева поиска осуществляется по определенному правилу (алгоритму). Пусть дана последовательность целых чисел {11, 5, 17, 15, 1, 8, 19, 13, 21}, представленная как срез *numbers[]*. Вначале формируется корневой узел {11}, затем в цикле для каждого узла рекурсивно вызывается метод *insertion (value)* (рис. 8.2 а), который в свою очередь вызывает метод *createNod*e(*value*), создающий новый узел (рис. 8.2 б) и размещающий его в левом или правом поддереве в зависимости от значения *value* (рис. 8.2 в).

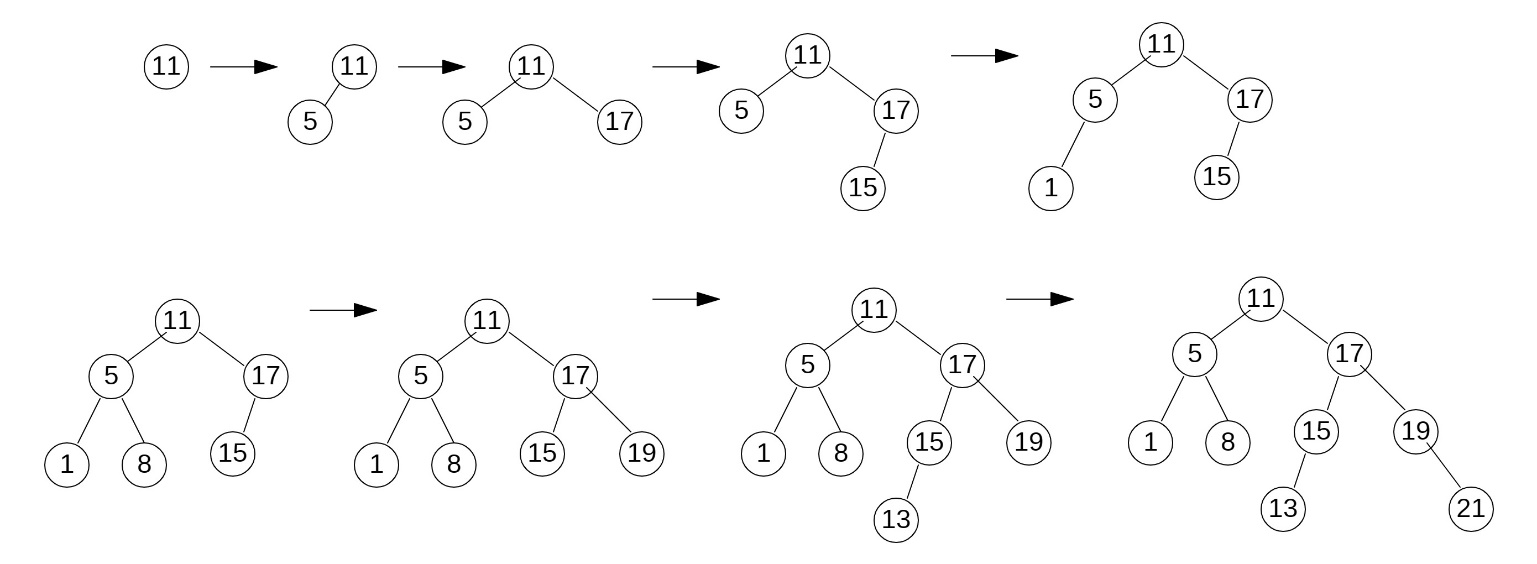
а) построение дерева б) создание узла



в) метод *Insert (value)*

Рис. 8.3. Дракон-диаграммы методов построения дерева

Полный процесс формирования бинарного дерева показан на рис. 8.4*.* Первое число 11 записывается в корень дерева*.* Второе число 5 меньше значения в корне дерева, следовательно, это число записывается в левое поддерево*.* Следующее число 17 больше значения в корне, соответственно оно записывается в правое поддерево*.* Далее число 15 больше, чем значение в корне дерева, значит, оно записывается в правое поддерево, но правое поддерево уже построено. Число 15 сравнивается со значением в корне правого поддерева — числом 17. Так как добавляемое значение меньше значения в корне правого поддерева, то добавляем левое поддерево уже к этому узлу. В конечном результате формируются бинарное дерево поиска, в котором представлены три варианта: узел 5 является родителем двух потомков (1,8), узел 15 имеет только одного левого потомка, а узел 19 — одного правого потомка. Такое расположение узлов выбрано для демонстрации работы функции удаления узлов, что будет рассмотрено позже.

 Рис. 8.4. Построение бинарного дерева

8.2.3. Поиск узла по заданному значению

Другой базовой операцией является функция поиск узла по его значению *findNode(value)*. В этой функции используется конструкция языка Golang – *Select*, где искомое значение *val* рекурсивно сравнивается со значениями других узлов в процессе обхода дерева. Заметим, что оператор *Select* в сгенерированном коде представляется операторами *if-else*. Если совпадение найдено, печатается результат “Узел существует”. Отсутствие узла с таким значением обнаруживается по признаку равенства <nil> адресов *root.leftNode* и *root.rightNode* в переменной *root* типаBSTree*:*

*BSTree struct {*

*rootValue int*

*leftNode \*BSTree*

*rightNode \*BSTree*

*}*

Дракон-диаграмма функции *findNode(root \*BSTree, val int)* представлена на рис. 8.5. Поиск осуществляется по всему дереву, конечные узлы определяются по выполнению условия (*root.leftNode == nil && root.rightNode == nil).*

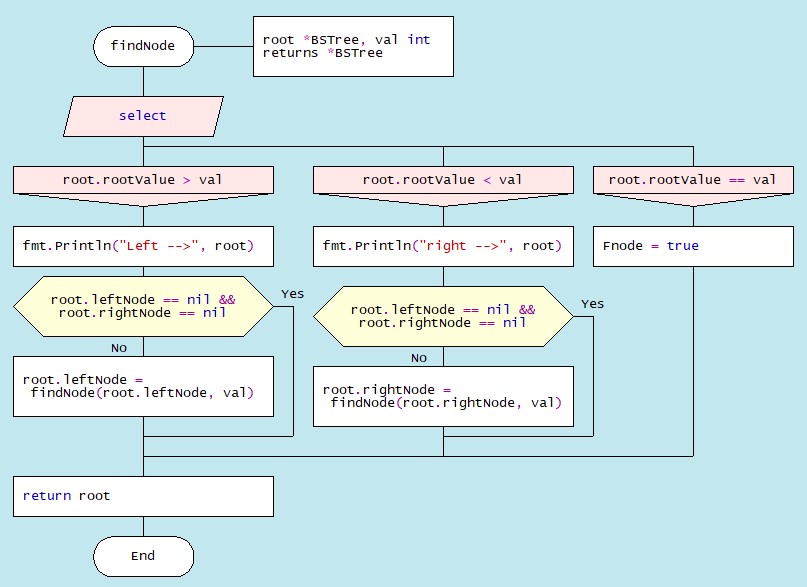


Рис. 8.5. Дракон-диаграмма метода *findNode(root \*BSTree, val int)*

8.2.3. Удаление узла с заданным значением

Следующая базовая операция - удаление узла с указанным значением. Здесь рекурсивно используется функция *deleteNode (root \*BSTree, val int)*. Алгоритм этой функции осложняется тем, что возможны следующие варианты расположения потомков, как, например, на рис. 8.3:

а) удаляемый узел 21 не имеет потомков;

б) удаляемый узел 19 имеет правого потомка;

в) удаляемый узел 15 имеет левого потомка;

г) удаляемый узел 5 имеет правого потомка.

Рассмотрим более подробно порядок перемещения узлов в указанных вариантах:.

*Вариант а):* Узел (21) не имеет потомков (рис. 8.6.).

В этом случае этот узел удаляется путем изменения в родительском узле значения *root = nil*. На рис.8.6. показан процесс удаления узла (21) и соответствующий фрагмент дракон-диаграммы, где L1 := root.leftNode == nil и R1 := root.rightNode == nil.

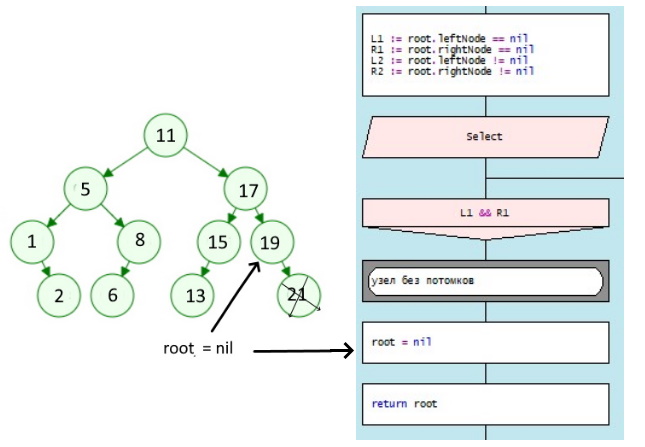


Рис. 8.4. Удаление узла без потомков

*Вариант б):* Узел (19) (рис. 8.7.) имеет правого потомка.

В этом случае узел (19) удаляется из дерева путем замены его адреса в родительском узле (17) на адрес единственного потомка – узла (21). На рис. 8.7. показан процесс удаления узла (19) и соответствующий фрагмент дракон-диаграммы, где L1 := root.leftNode == nil и R2 := root.rightNode != nil.

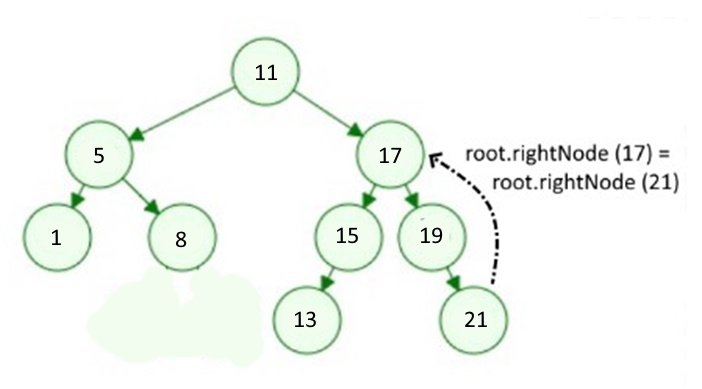


Рис. 8.7. Удаление узла (19) c правым потомком

*Вариант в):* Узел (15) имеет левого потомка (рис. 8.8.).

В этом случае узел (15) удаляется из дерева путем замены его адреса в родительском узле (17) на адрес единственного потомка – узла (13). На рис. 8.8. показан процесс удаления узла (15) и соответствующий фрагмент дракон-диаграммы, где L1 := root.leftNode == nil и R2 := root.rightNode != nil.

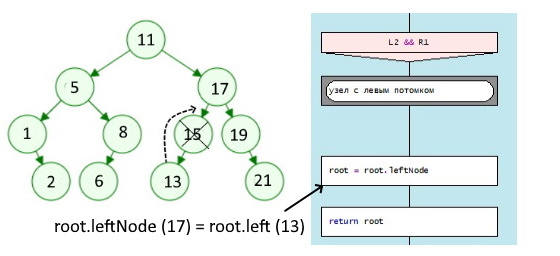


Рис. 8.6. Уд Рис. 8.8. Удаление узла (15) c левым потомком

*Вариант г):* Узел (5) имеет двух потомков (рис. 8.9.). В этом случае двоичное дерево поиска перестраивается: узел (2) переходит на место узла (5):

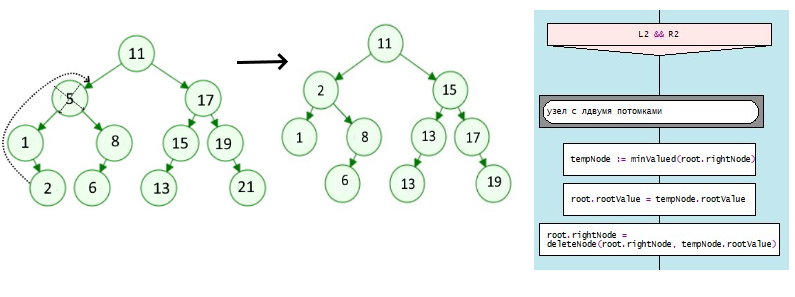


Рис. 8.9. Удаление узла (5) c двумя потомками

Полностью дракон-диаграмма метода удаления узла из двоичного дерева поиска имеет вид (рис. 8.10.):

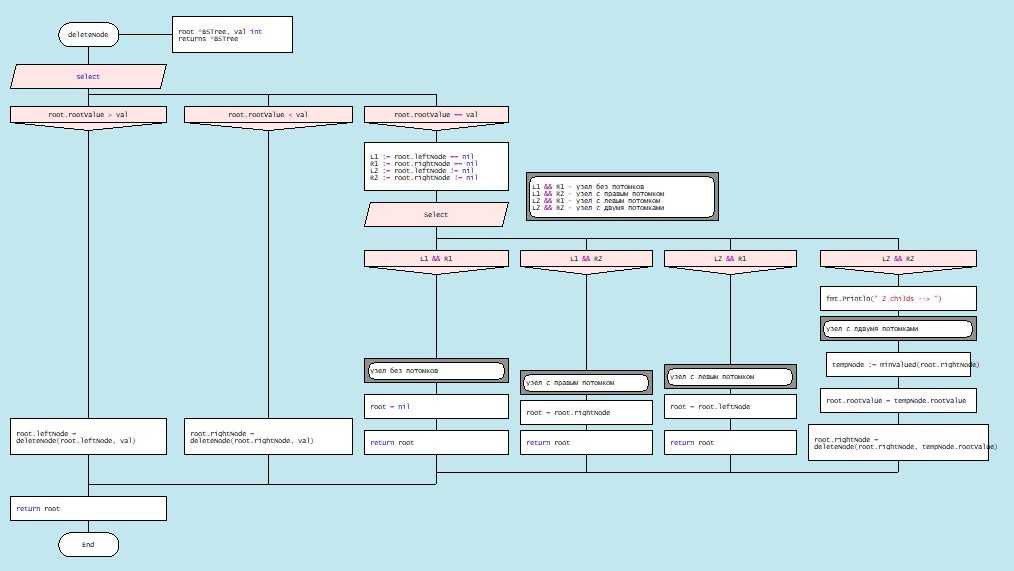


Рис. 8.10. Дракон-диаграмма функции удаления отдельного узла (4 варианта)

8.2.4. Обход двоичного дерева поиска

К базовым операциям с деревьями относится операция обхода дерева по узлам. В отличие от линейных структур данных, в которых обход элементов выполняется в линейном порядке, узлы деревьев можно обходить различными путями. Обход, при котором каждый узел-предок просматривается прежде его потомков, называется *предупорядоченным* обходом или обходом в прямом порядке (pre-order traversal). Обход, при котором вначале просматриваются потомки, а потом предки, называется *поступорядоченным* или обходом в обратном порядке (post-order traversal). Существует также центрированный обход (in-order traversal), при котором посещается сначала левое поддерево снизу-вверх, затем корневой узел, затем — правое поддерево (рис. 8.11, 8.12, 8.13) [[<https://intellect.icu/derevo-kak-struktury-dannykh-dvoichnye-binarnye-derevya-9856>].

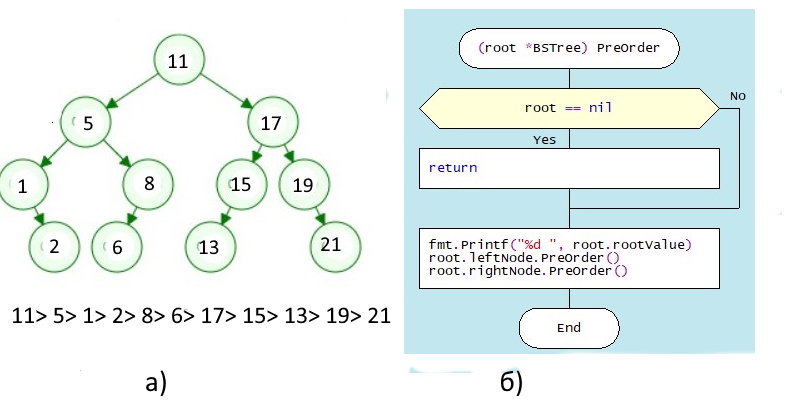


Рис. 8.11. a) Прямой обход дерева ;б) дракон-диаграмма функции PreOrder()

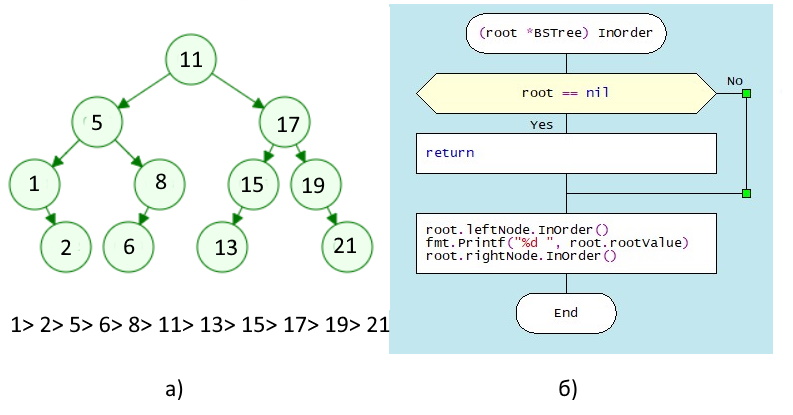


Рис. 8.12. a) Центрированный обход; б) дракон-диаграмма функции InOrder()

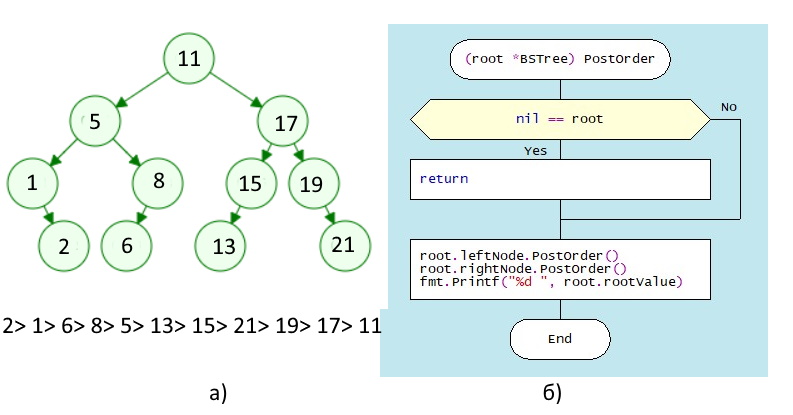


Рис. 8.13. a) Обратный обход; б) Дракон-диаграмма функции PostOrder()

**8.3. Самобалансирующиеся двоичные деревья (АVL-деревья)**

|  |  |
| --- | --- |
| Пути доступа к программным файлам проекта avl | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

Эффективность выполнения любых операций с деревьями существенно зависит от порядка поступления входных данных. Например, если поступающая последовательность чисел частично отсортирована по возрастанию или убыванию, то внешний вид такой структуры уже не будет похож на дерево (рис. 8.14).

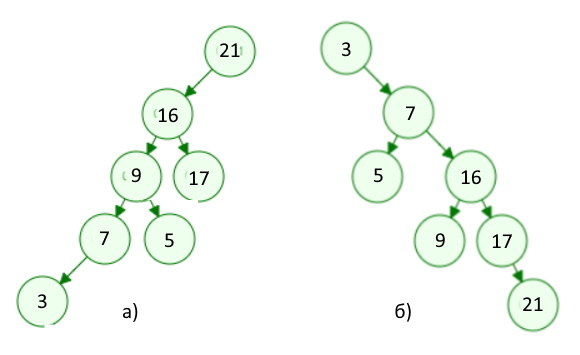


Рис. 8.14. Частично отсортированные входные данные

В таких практически «вырожденных» деревьях сложность операций определяется числом узлов, то есть, близка к линейной - Ο(n). При этом левое и правое поддерево неуравновешены (несбалансированы), что можно оценить коэффициентом баланса (kb), равным разности высот левого и правого поддеревьев. Напомним, что высота дерева определяется как длина самой длинной ветви в поддереве (сумма ребер). Для идеального двоичного дерева поиска (дерево, в котором число узлов в левом поддереве равно числу узлов в правом поддереве) этот коэффициент равен 0 (рис. 8.15).

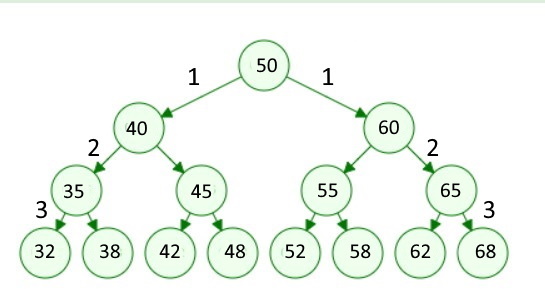


Рис. 8.15. Идеальное двоичное дерево

В реальной жизни идеальные двоичные деревья практически не встречаются, чаще программисты стремятся построить дерево, в котором высота левого поддерева отличается от высоты правого не более чем на 1. Такие деревья получили название AVL-деревьев; для таких деревьев сложность операций определяется как О(logn), то есть время выполнения базовых операций (поиск, удаление) существенно меньше чем BST-деревья.

Алгоритмы построения таких деревьев основаны на процессе балансировки дерева при вставке нового узла или удаления какого-либо существующего узла. Целью балансировки является перестройка дерева таким образом, чтобы высóты левого и правого поддеревьев не отличались более чем на 1. При этом коэффициент балансировки должен удовлетворять следующим условиям:

* допустимые значения kb = -1, 0 и +1;
* значение kb= -1 указывает на то, что правое поддерево «перевешивает» левое поддерево;
* значение kb= +1 указывает, что левое поддерево «перевешивает» правое поддерево;
* значение kb = 0 показывает, что дерево содержит равное число узлов с каждой стороны, то есть дерево идеально сбалансировано.

Технология балансировки сводится к выполнению круговых перемещений узлов в четырех вариантах:

* правый поворот (RR);
* Левый поворот (LL);
* Право - левый поворот (RL);
* Лево - правый поворот (LR).

Правый поворот выполняется, когда корневой узел имеет коэффициент баланса kb = +2, а его левый потомок имеет коэффициент баланса kb = +1 (рис. 8.16):

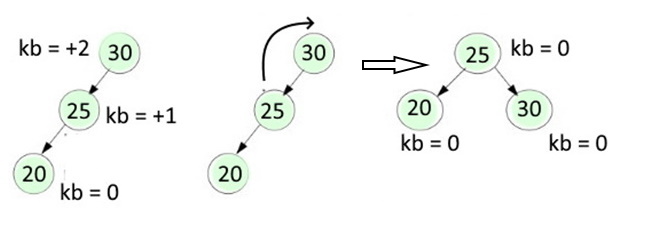


Рис. 8.16. Правый поворот

Левый поворот выполняется, когда корневой узел имеет сбалансированный коэффициент kb = - 2, а его правый потомок имеет коэффициент баланса kb = - 1 (8.17):

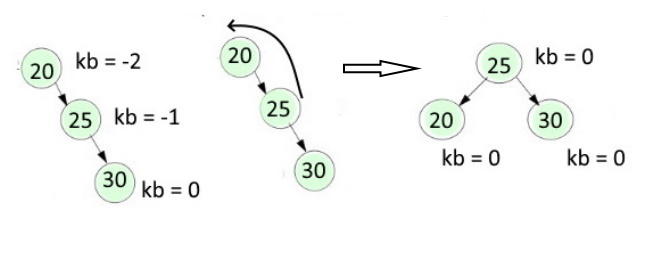


Рис. 8.17. Левый поворот

Право-левый поворот выполняется, когда корневой узел имеет коэффициент баланса kb = -2, а его правый потомок имеет коэффициент баланса kb = +1 (рис.8.18):

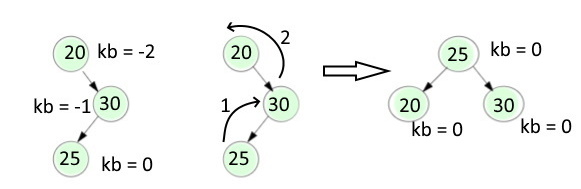


Рис. 8.18. Право-левый поворот

Лево-правое вращение выполняется, когда у узла коэффициент баланса равен kb = +2, а у его левого потомка коэффициент баланса равен kb = -1 (рис.8.19):

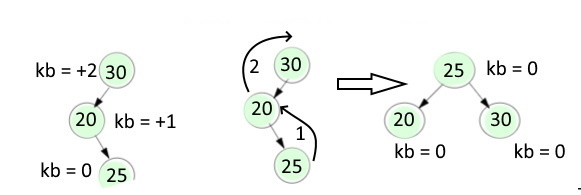


Рис. 8.19. Лево-правый поворот

Рассмотрим процесс балансировки более подробно на примере формирования дерева при поступлении входных данных в таком порядке: 21, 17, 16, 11, 9, 7, 5, 3. Такое дерево несбалансированно, точнее даже представляет собой практически «хворостину» (рис. 8.14):

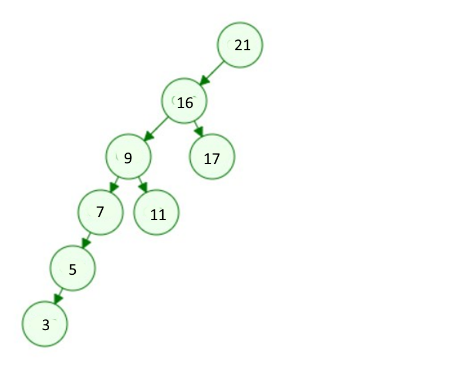
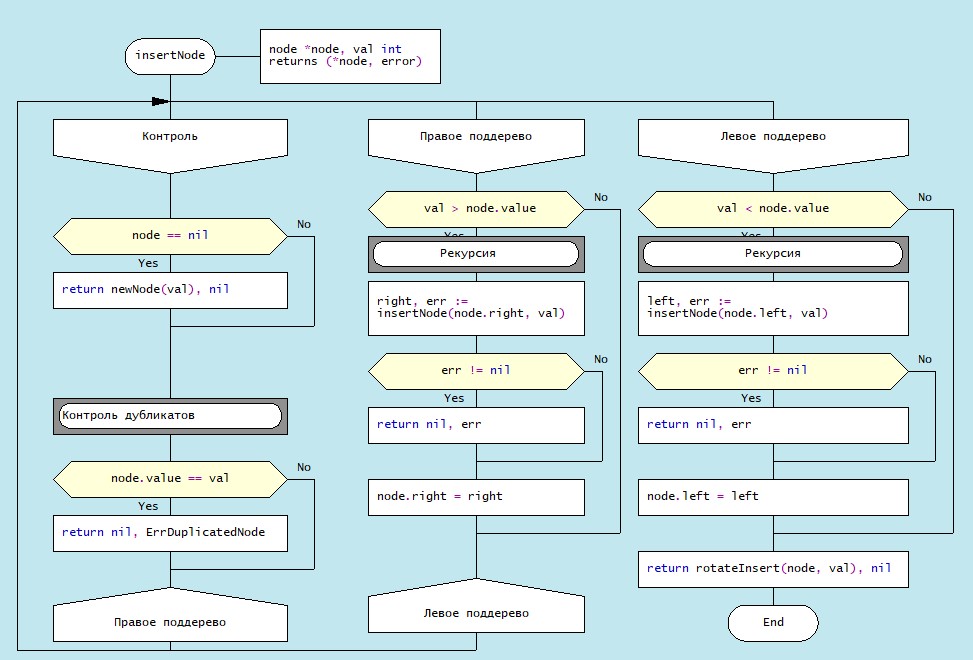


Рис. 8.20. Несбалансированное дерево

Проверка необходимости балансировки начинается при поступлении каждого нового узла с помощью метода *InsertNode(node \*node, value int)*, определяющего расположение нового узла в левом или правом поддереве относительно корневого узла. Выбор пути балансировки определяется методом *rotateInsert(node \*node, val int).* Дракон-диаграммыалгоритмов указанных методов представлены нарис. 8.21*.*



Риc. 8.21. Дракон-диаграммы метода вставки нового узла insertNode

Выбор одного из возможных путей балансировки (поворотов) определяется методом *rotateInsert(node \*node, val int)* (рис. 8.22)*.*

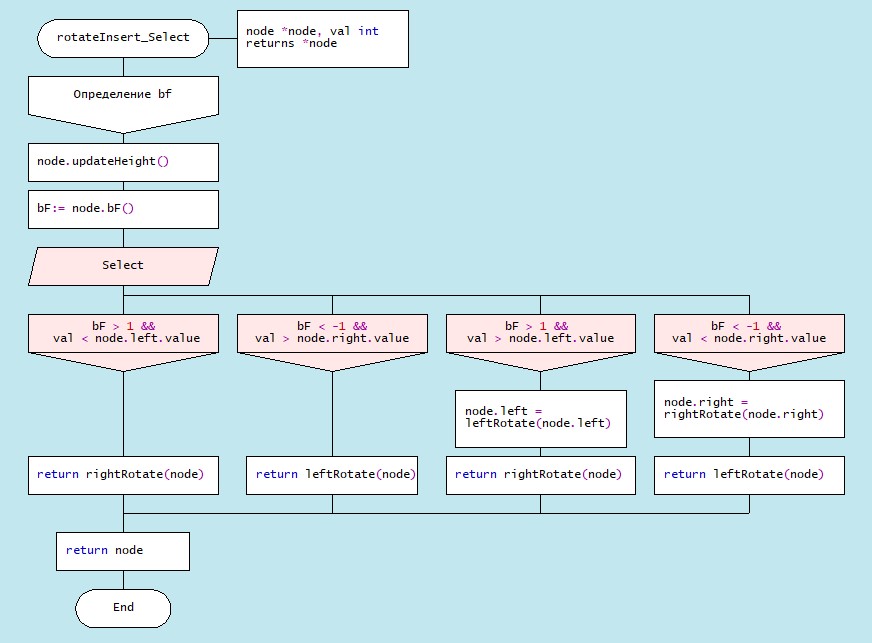
**

Рис. 8.22. Дракон-диаграмма метода *rotateInsert(node \*node, val int).*

В зависимости от расположения нового узла относительно родительского и от значений коэффициента балансировки выполняетсся вызов методов левого или правового вращений (рис. 8.23):

|  |  |
| --- | --- |
| func leftRotate(x \*node) \*node {  y := x.right  t := y.left  y.left = x  x.right = t  x.updateHeight()  y.updateHeight()  return y  } | func rightRotate(x \*node) \*node {  y := x.left  t := y.right  y.right = x  x.left = t  x.updateHeight()  y.updateHeight()  return y  } |

Рис. 8.23. Программные коды вращения узлов

Рассмотрим более подробно процесс перестройки дерева как результат применения метода *rotateInsert(node \*node, val int)* на примере поступления значений трех узлов (21,17,16). После введения значения узла (16) вместо дерева появляется «хворостина», требующая балансировки. В данном случае в методе *rotateInsert(node \*node, val int)* выполняется условие: коэффициент балансировки равен 2, и значение узла (16) меньше родительского узла (17) и происходит обращение к методу *rightRotate(node).* Процесс балансировки показан на рис.8.24:

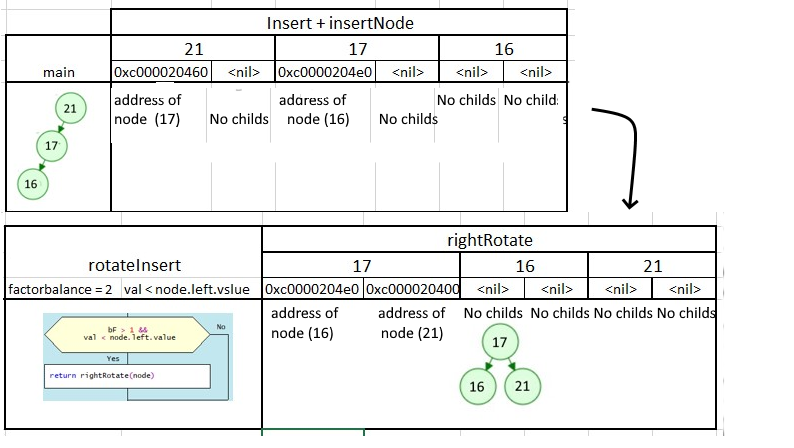


Рис. 8.24. Процесс вращения узлов для балансировки

Как видно из этого рисунка, результатом работы метода *rightRotate(node)* является значений полей адресов элементов структуры *node*. Нопомним, что эта структура состоит из четырех полей: высоты дерева, значения узла и адресов левого и правого узлов потомков. На рис.8.18. под значением узла показаны адреса узлов-потомков. После перестройки поля структуры *node.left* и *node.right* узлов (17) и (21) меняются и узел (17) становится корневым, узел (16) остается без изменений, а бывший корневой узел (21) становится правым потомком узла (17). В результате поступления узла (16) дерево становится сбалансированным.

Процесс поступления новых узлов и перестройка возникающего дерева путем балансирования для набора входных данных (21,17,16,11,9,7,5,3) представлен на рис. 8.25.

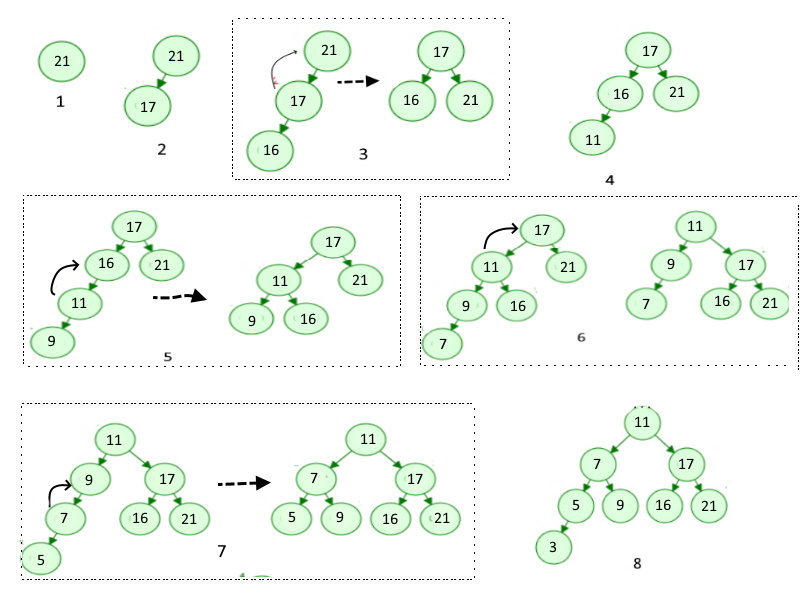


Рис. 8.25. Процесс построения сбалансированного AVL-дерева

Основными операциям с AVL-деревьями являются операция нахождения узла по заданному значению (*findNode(node \* node, val)* и операция удаления (removeNode(node \*node, val). Дракон-диаграмма метода *findNode(node \* node, val)* представлена на рис*.* 8.26:

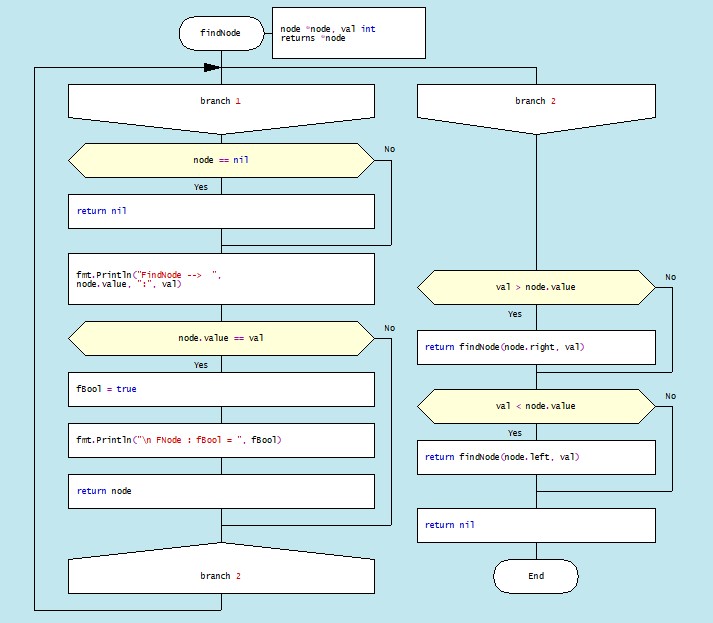
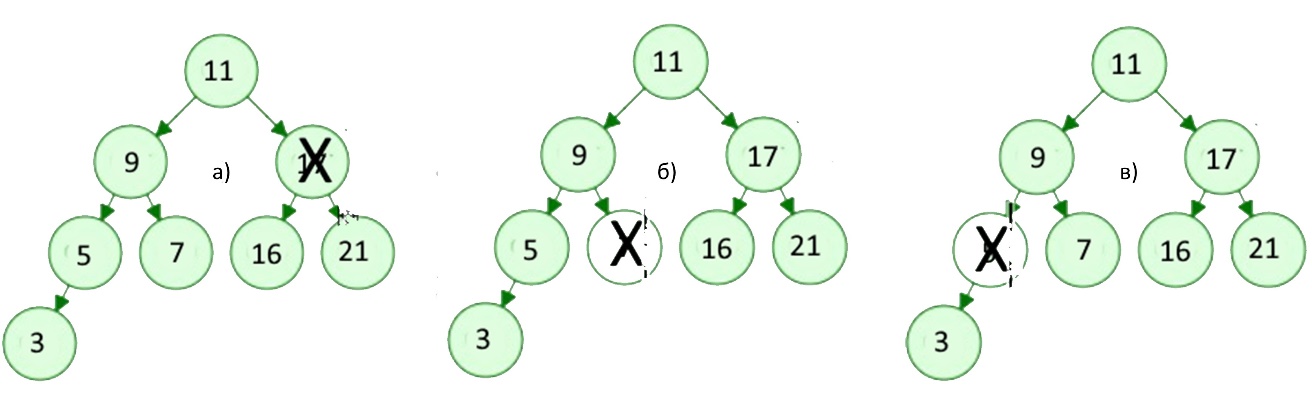


Рис. 8.26. Дракон-диаграмма метода нахождения узла с искомым значением

Другая базовая операция – удаление узла с указанным значением – состоит из следующих этапов. Поиск узла осуществляется с корня вниз по ветвям до удаляемого узла. При этом могут возникнуть следующие ситуации (рис. 8.27):

 Рис. 8.27. Варианты удаляемых узлов

а) Удаляемый узел имеет два непустых потомка;

б) Удаляемый узел не имеет потомков;

в) Удаляемый узел имеет один потомок (левый или правый).

Как и в других методах вначале рекурсивно определяется узел с заданным значением, а затем выбирается один из представленных вариантов. Дракон-диаграмма алгоритма удаления узла с указанным значением показан на рис. 8.28.

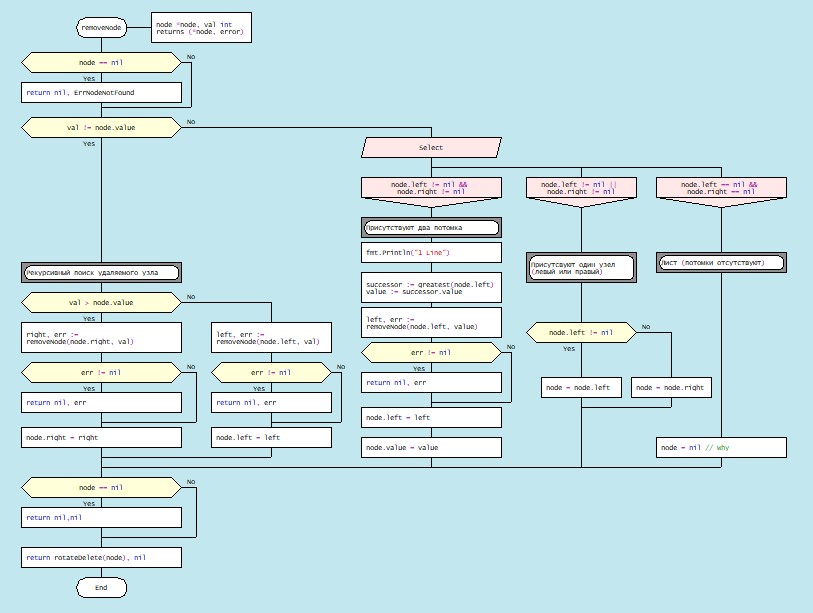


Рис. 8.28. Дракон-диаграмма алгоритма удаления узла с заданным значением

В головной программе *main()* вводится массив данных, выполняется поиск узла с заданным ззначением и удаление узла с заданным значением. Результат реализации сгенерированного кода представлен ниже:

*Обход дерева до удаления:*

*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*

*Число 3 присутствует*

*Число 6 отсутствует*

*Удаление узла 7*

*Обход дерева после удаления числа 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. Красно-черные деревья (Red-Black Tree)**

|  |  |
| --- | --- |
| Пути доступа к программным файлам проекта rbt | |
| Дракон-диаграмма | <https://github.com/ISA-victory/dsa-dg.git>, |
| Сгенерированный код | https://github.com/ISA-victory/dsa-dg.git |

Красно-черное дерево представляет собой разновидность самобалансирующегося двоичного дерева поиска, в котором узлы размещены по определенному правилу и окрашены в красный или черный цвета (рис. 8.29)

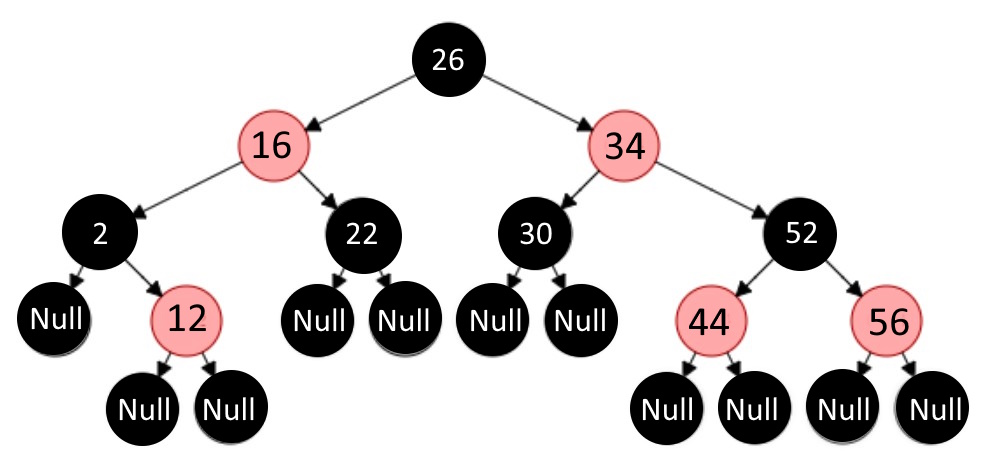


Рис. 8.29. Красно-черное дерево

Узлы, содержащие данные (в данном случае – целые числа), являются внутренними. Кроме того, красно-черные деревья содержит мнимые, «нулевые» узлы, связанные с листьями дерева (Null – на рис. 8.29). Красно-черные деревья удовлетворяют всем свойствам двоичного дерева поиска и должны обладать следующими свойствами:

#1. Каждый узел окрашен в красный или черный цвет.

#2. Корень дерева всегда черный.

#3. Все листья черные (Null).

#4. Оба потомка красного узла чёрные, то есть не может быть последовательных красных узлов.

#5. Все простые пути от узла к нисходящим листьям содержат одинаковое число чёрных узлов.

В отличие от AVL-деревьев, в которых баланс достигается соотношением высот левого и правого поддеревьев, баланс красно-черного дерева достигается свойствами, указанными выше. Добавление или удаление узла из красно-черного дерева может нарушить свойства красно-черного дерева и восстановление баланса достигается путем двух операций: перекрашивания узлов и (или) перестройки всего дерева или его поддеревьев с помощью определенных вращений.

Наиболее важными из перечисленных свойств являются свойства 4 и 5. Следствием свойства 4 является требование одинакового количества красных и чёрных узлов на пути между двумя узлами. Свойство 5 требует, чтобы независимо от выбора пути между двумя узлами число чёрных узлов должно быть одинаковым. Иными словами, в худшем случае высота дерева не должна быть более чем в два раза превосходить высоту кратчайшего пути. В этом случае красно-черное дерево становится достаточно сбалансированным.

Чёрная высота - важный термин, используемый для красно-чёрных деревьев. Это число чёрных узлов на любом простом пути от узла х (не включая его) к листу. Черная высота любого узла х представлена как логарифм от числа узлов в дереве. В частности, если в каждой ветви от корня к листьям одинаковое количество черных узлов, то высота дерева будет равна логарифму от числа узлов. [Это свойство позволяет быстро выполнять основные операции дерева поиска, такие как *добавление, удаление и поиск* узла](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE).

Для реализации указанных операций на программном уровне вначале создается красно-черное дерево, для чего объявляются два пользовательских типа – узeл *Node* и дерево *RBTree*:

Узел Node включает следующие поля: data – числовое значение узла, colour – цвет узла типа bool: красный узел – TRUE; черный узел – BLACK), левый (left), правый (right) и родительский (parent) узлы (Рис. 8.30):

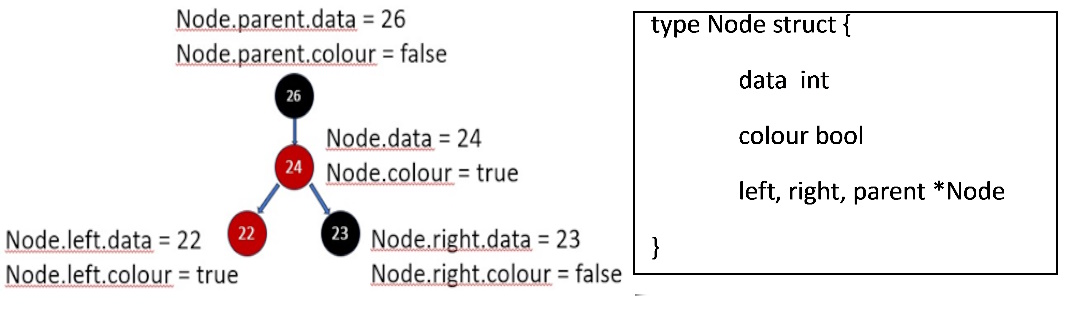


Рис. 8. 30 Создание структуры узла Node (data, colour, left, right, parent)

Формирование узлов дерева для заданного набора значений {v1, v2, …vn} с помощью метода NewNode(d int, nullNode \*Node) (nd \*Node), где d – числовое значение узла, nullNode \*Node – пустой узел, (nd \*Node) – возвращаемое значение функции:

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 // Новые узлы имеют красный цвет.

return

}

Дерево RBTreeвключает два поля: root и *nullNode*. Поле root - это корень дерева, используемый для доступа к другим узлам. Поле *nullNode* - специальный узел, который используется в красно-черных деревьях для представления пустых ссылок:

*type RBTree struct {*

*root \*Node*

*nullNode \*Node*

*}*

Для достижения баланса красно-черного дерева процесс вставки нового узла сопровождается проверкой на удовлетворение указанным выше свойствам. Поскольку красно-черное является двоичным деревом поиска (BST), вначале определяется местоположение на дереве, куда нужно ставить узел, одновременно окрашивая его в красный цвет (рис.8.31):

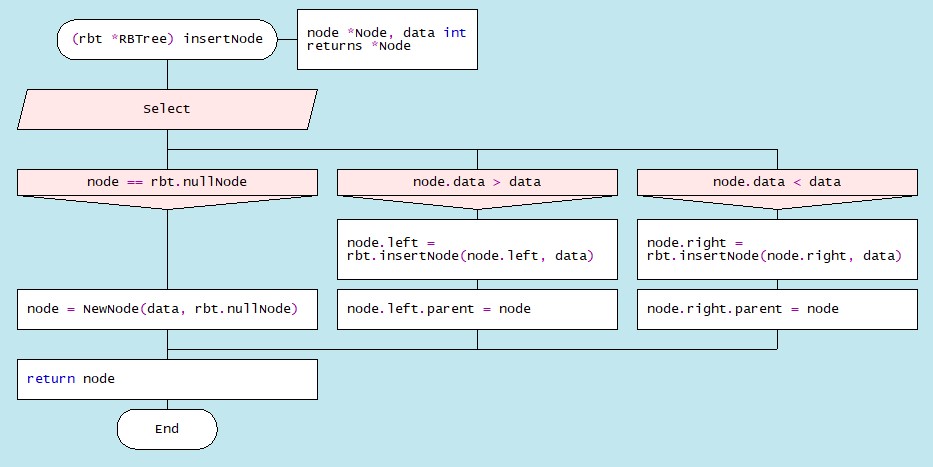


Рис.8.31. Дракон-диаграмма метода вставки insertNode(node,data)

В этом методе вызывается функция NewNode(d,nullNode), которая возвращает полную информацию о новом узле Node, при этом поле colour типа bool первоначально принимает значение red.

Далее рассмотрим варианты расположения узлов на дереве в результате вставки каждого нового узла и определим какие свойства красно-черного дерева нарушены. Для понимания этого процесса введем следующие обозначения: x - новый узел, P (Parent) – родитель узла х, G (Grandparent) – предок (родитель родителя), U (Uncle) – дядя узла х (рис. 8.32):

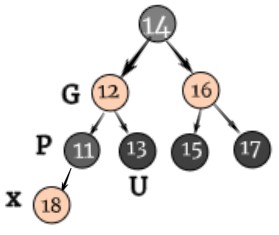


Рис. 8.32. Обозначения родственных узлов x, P,G,U

Процесс балансировки рассмотрим на примере построения красно-черного дерева из наборов целых числе, являющихся ключами вводимых узлов: {11,12,13,14,15,16,17,18,19} (рис.8.33):

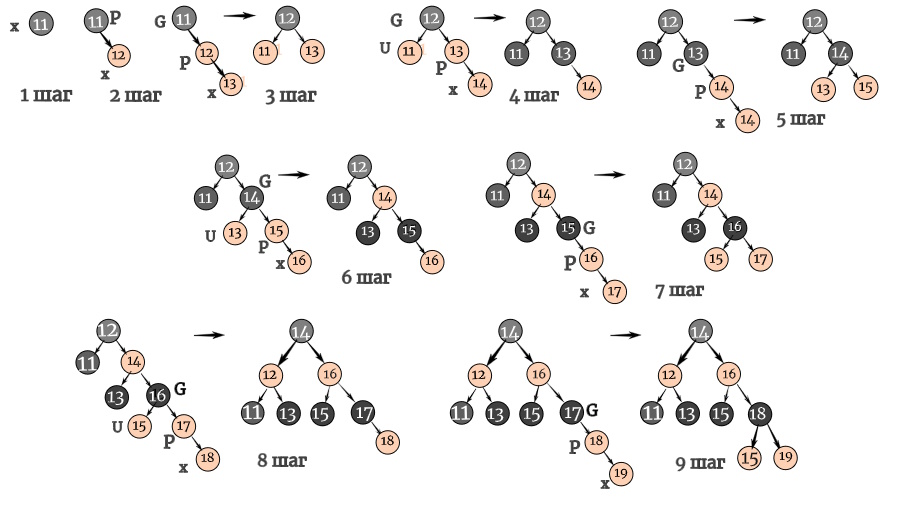


Рис. 8.33. Процесс балансировки красно-черного дерева

Первые два шага к нарушению свойств не приводят. Ввод узла с ключом (13) приводит к нарушению свойства #4. Для восстановления баланса на третьем шаге выполняется левый поворот против часовой стрелки, вследствие чего узел (12) становится корневым, меняя цвет на черный, а бывший корневой узел (11) становится левым потомком. На следующем шаге возникает ситуация, когда три узла становятся красными: новый узел x(14), родитель P(13) и дядя U(11), здесь баланс можно достичь изменением цвета узлов (11) и (13).

С другими наборами ключей возникают другие сочетания родственных узлов (x, P, G, U) (рис. 8.34):

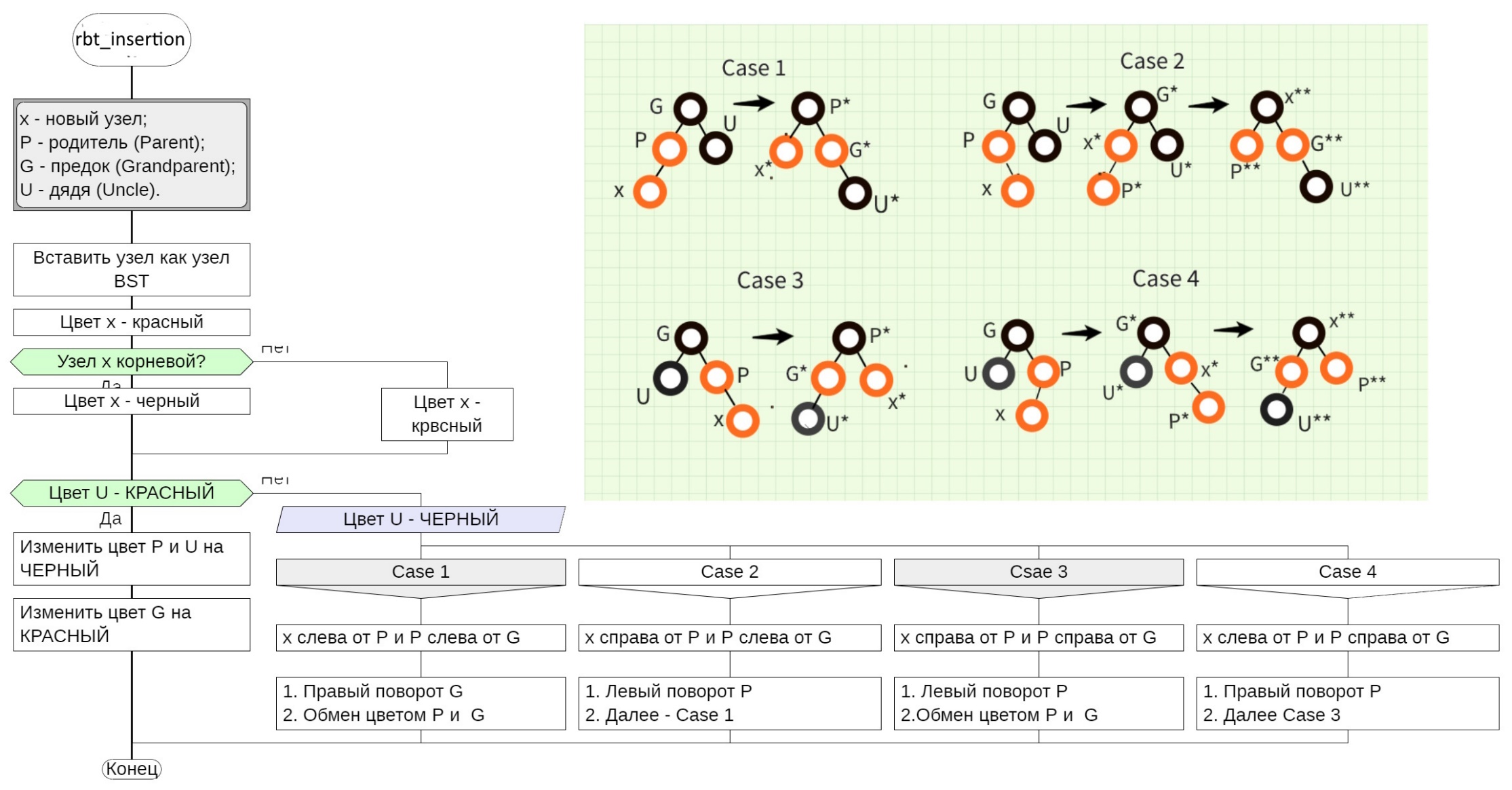


Рис.8.34. Фрагменты дракон-диаграммы вариантов взаимного расположения родственных узлов (x, P, G, U0

Удаление узла в красно-черном дереве

На первом шаге узел просто удаляется как узел в двоичном дереве поиска. После удаления узла и замены его другим узлом свойства красно-черного дерева могут быть нарушены. Для их восстановления необходимо выполнить следующие операции:

1. Если удаляемый узел красного цвета, то свойства красно-черного дерева не нарушаются. Если замещающий узел красный, то его следует перекрасить в черный.

2. Поворачивайте, перекрашивайте и/или переносите \"blackness\" (количество черных узлов по отношению к красным) вверх по дереву до тех пор, пока дерево не уравновесится.

3. При этом нужно следить за тем, чтобы корень дерева был черным.

Рассмотрим способы восстановления свойств красно-черного дерева. Мы вводим следующие имена: удаляемый узел — "x", дочерний узел удаленного узла - "y", одноуровневый узел удаляемого узла - "s" (рис. 8.35).

Рассмотрим балансировочный процесс построения красно-черного дерева из наборов целых чисел, являющихся ключами входных узлов: {11,12,13,14,15,16,17,18,19} (рис. 8.35):

![](../Engl\_images/E\_Im8/Fig8\_35\_???.jpg)

<figcaption>Figure 8.35. Red-black tree balancing process</figcaption>

The first two steps do not violate any properties. Entering a node with key (13) causes property #4 to be disturbed. To restore the balance in the third step, a counterclockwise left rotation is performed, causing node (12) to become the root node, changing its colour to black, and the former root node (11) to become the left descendant. In the next step a situation arises where three nodes become red: new node x(14), parent P(13) and uncle U(11), here the balance can be achieved by changing the colour of nodes (11) and (13).

With other sets of keys, other combinations of related nodes (x, P, G, U) of the tree arise, balancing which is achieved by colour change operations and node rotations, as illustrated by the drakon-diagram (Figure 8.33). The red-black tree balancing Drakon diagram  is shown in Figure 8.34. This diagram is made in Silhouette mode, consisting of four branches. In the first branch, the root node is coloured black. The second branch defines a hierarchy of related nodes (x, P, G, U), the third branch colours these nodes with respect to the hierarchy. The fourth branch calls the corresponding rotation methods:

Removing a node in the red-black tree

In the first step a node is simply removed as a binary search tree node (see BST).

[If the replacement node is a \'red\' node or if the node to be removed is a \'red\' node, we can simply delete the node and replace it with another node. There is no need to make any additional changes to the tree structure.]{.mark}

[З. If the replacing node is a \'black\' node and the node to be removed

is a \'black\' node, we need to perform some additional operations to ensure that the red-black tree properties are maintained.]{.mark}

[4. If the replacement node is a leaf node (i.e. has no child nodes), we replace the node with a \'null\' leaf node and colour it black.]{.mark}

[5. If the replacing node has one child, we replace the node with its

child and colour it black.]{.mark}

[6. If the replacing node has two child nodes, we replace the node with

its successor node in order and then remove the successor node in order

(which is no more than a node with one child node) using the above

method\....]{.mark}

After removing a node and replacing it with a new node, the properties of the red-black tree may be compromised. The following operations must be carried out to restore them:

If the node to be removed is red, the properties of the red-black tree are not affected. If the replacement node is red, it must be repainted black.

Next, rotate, recolour the nodes and/or move the \'blackness\' (the

number of black nodes relative to red nodes) up the tree until the tree is balanced.

Make sure that the root of the tree is black.

Consider the options for restoring the red-blackwood properties.

Introduce the following notations: node to be deleted is \"x\", child

node of deleted node is \"y\", sibling of deleted node is \"s\" (Figure

8.34).

2\. Case 1: If the child node of the deleted node is red. If you make

the descendant colour black, the number of black nodes will be restored.

![](media/image33.jpg){width="10.118055555555555in" height="4.3125in"}

*\*Figure 8.33. Drakon diagram  of variants of mutual arrangement of related nodes (x, P, G, U)\**

![](media/image36.jpg){width="10.118055555555555in"

height="3.8652777777777776in"}

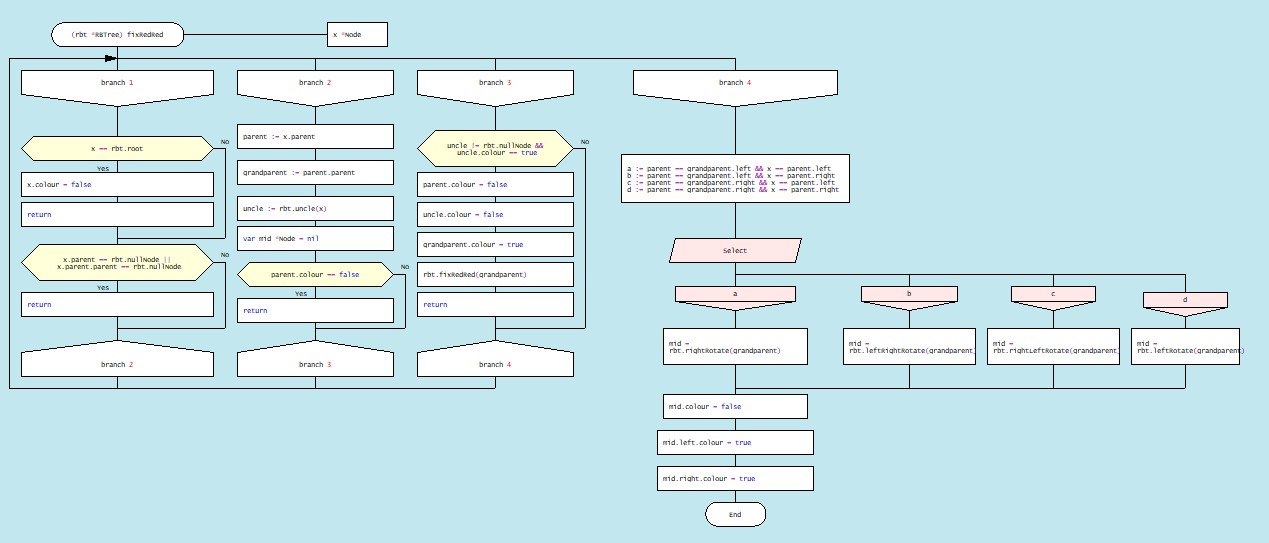
<figcaption>Figure 8.34. Drakon diagram  of the red-black tree node balancing method fixTreeBalance(G)\*   <figcaption>

![](media/image37.png){width="3.7916666666666665in"

height="1.7291666666666667in"}

\*\

\*



Литература

<https://slideplayer.com/slide/16885602/>

[C Program for Red Black Tree Insertion - GeeksforGeeks](https://www.geeksforgeeks.org/c-program-red-black-tree-insertion/)