Сбалансированные бинарные деревья поиска

Бинарное дерево поиска BTS

Ключи в дереве упорядочены по правилу:

• в левом поддереве – ключи со значениями меньшими чем в корне;

• в правом поддереве со значениями большими чем в корне .

При построении BTS : Если ключи: 1 2 3 4 5 6 7 – то только правое поддерево T1

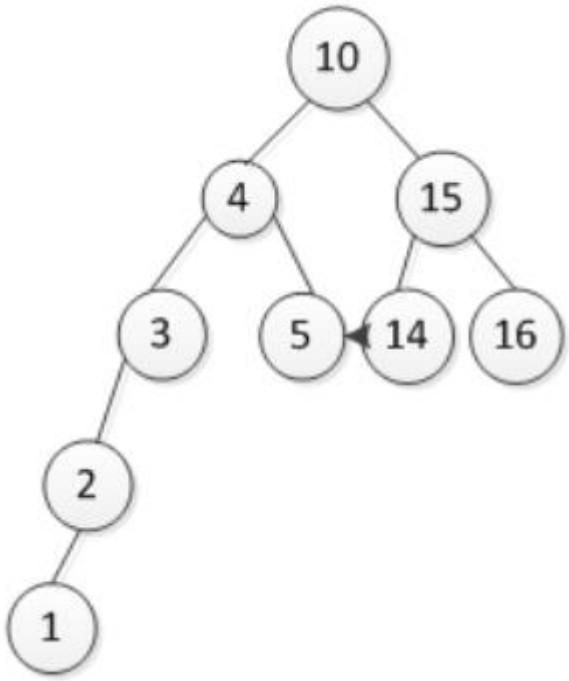
Если ключи: 7 6 5 4 3 2 1 – то только левое поддерево T2

Если ключи случайные : 10 3 18 4 15 2 1 19 два поддерева и левое и правое

Зачем балансировать дерево?

При построении бинарного дерева поиска мы лишь можем предположить, что оно не будет идеально сбалансированным.

При поиске ключа в ИСД с n узлами среднее число сравнений равно h=logn. Длина поиска в BTS определяется его высотой и зависит от количества узлов (n) и от порядка поступления ключей при построении дерева. Длина поиска может колебаться в пределах от logn – при получении идеально сбалансированного дерева до n – при получении вырожденного дерева. Тогда эффективность алгоритмов поиска и вставки элемента в дерево BTS будут иметь порядок от O(logn) до O(n). В книге Д. Кнут «Сортировка и поиск» доказано что при случайном порядке включения элементов в дерево поиска средние затраты на поиск элемента будут 1,386*O(logn).



При построении BTS дерева с ключами 1,2,3 ….16 высота такого дерева максимальна (вырожденное) для этого набора 16 ключей, следовательно, и время выполнения операций над ним также будет максимальным. Поэтому при добавлении очередного узла, возможно, дерево понадобится перестраивать, чтобы уменьшить его высоту, сохраняя тот же набор узлов. Если при добавлении в BTS очередного узла количество узлов в левом и правом поддеревьях какого - либо узла дерева станет различаться более, чем на 1, то дерево не будет являться идеально сбалансированным, и его надо будет перестраивать, чтобы восстановить свойства идеально сбалансированного дерева поиска. Идеальную балансировку поддерживать сложно. Поэтому обычно требования к сбалансированности дерева менее строгие.

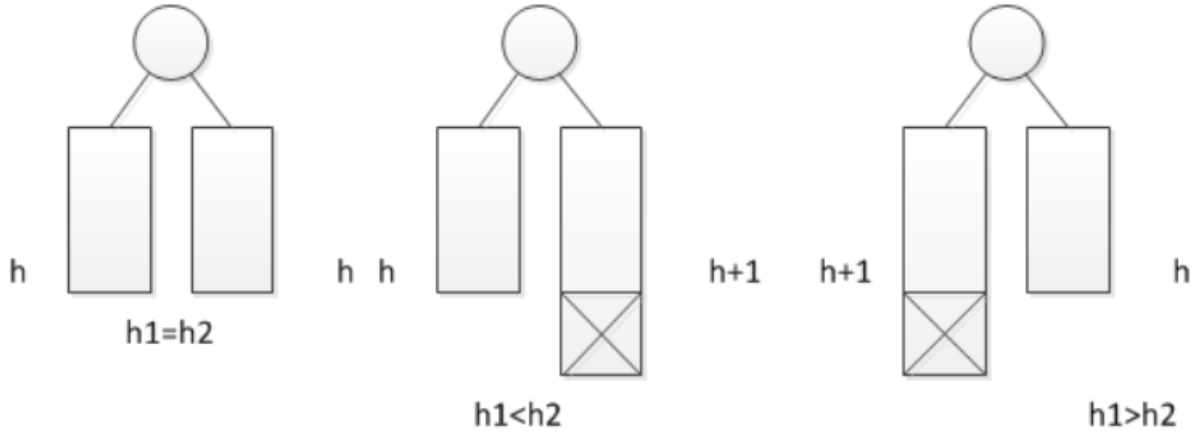
Сбалансированное дерево

Можно упростить операцию перестройки, если дать менее строгое определение сбалансированности (не по узлам).
Бинарное дерево является сбалансированным тогда и только тогда, когда для каждого узла ВЫСОТА его двух поддеревьев различается не более чем на 1.
Деревья, удовлетворяющие этим условиям стали называть АВЛ деревьями.

Операции над АВЛ деревьями имеют вычислительную сложность O(logn):
1. Найти узел сданным ключом
2. Включить узел сданным ключом
3. Удалить узел с данным ключом
Теорема Адельсон- Вильского и Ландиса утверждает:
максимальная высота сбалансированного дерева $\log(n+1) \leq h(n) \leq 1$,
$4404*\log(n+2) - 0{,}328$
Все ИСД являются АВЛ деревьями

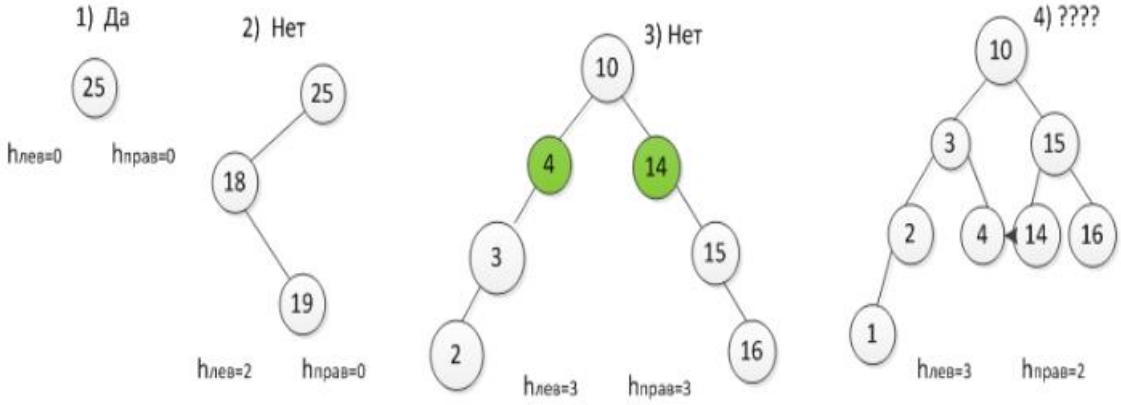
Сбалансированные двоичные деревья поиска
• АВЛ дерево
• Красно-черное дерево
• Самоперестраивающееся дерево(splay tree)
(Косое дерево)
• Рандомизированное дерево поиска

Показатели сбалансированности бинарного дерева



АВЛ дерево

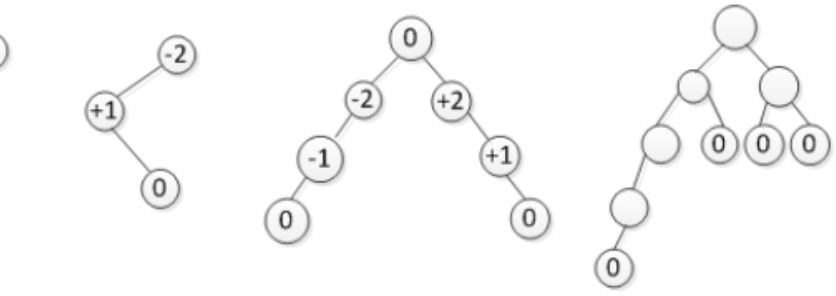
Это бинарное дерево поиска, которое:
- либо пусто,
- либо состоит из одного узла,
- либо имеет два поддерева, высота которых отличается не более чем на 1 и левое и правое поддеревья так же АВЛ.



Показатель сбалансированности и высота дерева

В каждом узле АВЛ-дерева, помимо ключа, данных и указателей на левое и правое поддеревья (левого и правого сыновей), родительский узел, хранится показатель баланса – разность высот правого и левого поддеревьев. В некоторых реализациях этот показатель может вычисляться отдельно в процессе обработки дерева тогда, когда это необходимо.

```
struct AVLNode{
 Tkey key;
 AVLNode *parent;
 AVLNode  *left;
 AVLNode *right;
 int balance;
};
```

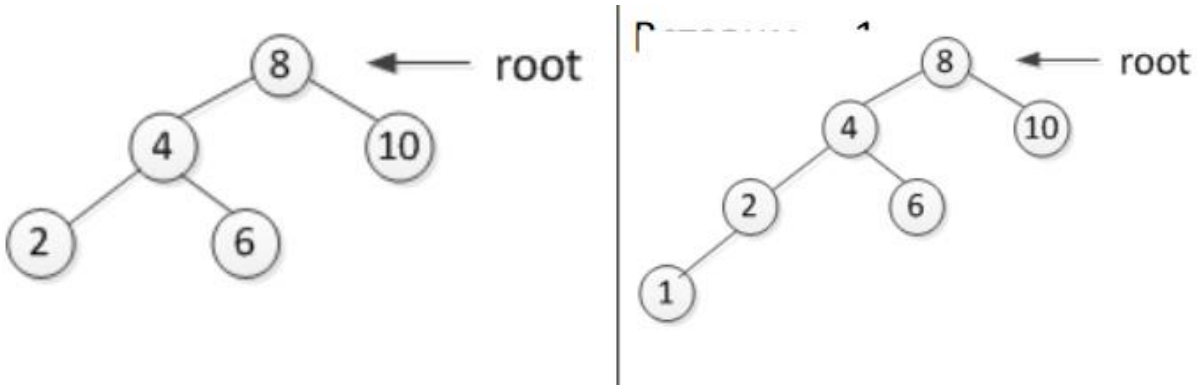


Таким образом, получается, что в АВЛ-дереве показатель баланса balance для каждого узла, включая корень, по модулю не превосходит 1 |balance|≤1. Если это условие нарушается, |balance|>1 то дерево не является АВЛ.

Определите по показателю, какие деревья на рис. выше являются АВЛ?

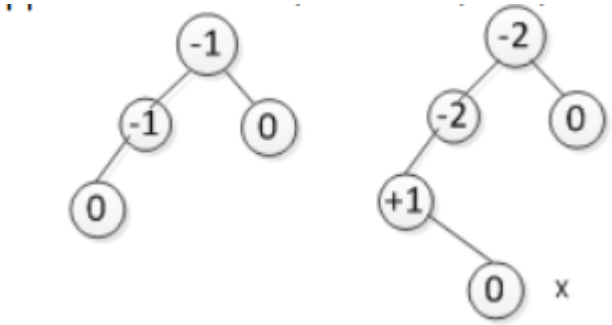
Включение узла в дерево может нарушить балансировку. Поэтому после включения узла дерево может быть перестроено, если оно стало не сбалансированным.

Root – корень дерева, L –левое поддерево, R –правое. Пусть в L включается новый узел . Происходит увеличение высоты левого поддерева на 1.



Вставим 1 - Балансировка нарушилась для узла с ключом 8. Требуется перестройка дерева.

Алгоритм: Вставка узла в сбалансированное дерево
1. Узел добавляется в дерево с помощью стандартного алгоритма вставки в двоичное дерево поиска. При вставке осуществляется корректировка показателя сбалансированности узлов на пути продвижения вставляемого узла до места вставки
2. Балансировка дерева
Показатели баланса в ряде узлов при вставке нового узла изменятся, и сбалансированность может нарушиться. Сбалансированность считается нарушенной, если показатель баланса по модулю превысил 1 в одном или нескольких узлах. При добавлении нового узла X разбалансировка может произойти сразу в нескольких узлах, но все они будут лежать на пути от этого добавленного узла к корню, как показано на Рис



Перестраиваться будет поддерево с корнем в том из этих узлов, который является ближайшим к добавленному.

2.1 Найти корень поддерева, которое подлежит перестроить, для этого надо подниматься вверх по дереву от вновь добавленного узла до тех пор, пока не найдется первый узел, в котором нарушена сбалансированность. Назовем его опорным узлом. Это общее правило для всех добавляемых узлов, приводящих к разбалансировке.

2.2. После того как опорный узел будет найден, необходимо выполнить перестройку поддерева с корнем в этом узле с целью восстановления его сбалансированности, алгоритмами поворотов. Остальная часть дерева в перестройке не участвует .

После перестройки поддерева все дерево также станет сбалансированным — показатель баланса не будет превышать 1 по модулю во всех узлах дерева.

При вставке узла в L (левое поддерево) возможны три случая:

1. Было hL= hR: после выполнения включения узла L и R имеют разную высоту, но показатель сбалансированности не нарушается и перестройка поддерева не требуется.

2. Было hL< hR: после выполнения включения узла L и R имеют равную высоту, сбалансированность даже улучшается перестройка поддерева не требуется.

3. Было hL>hR: после выполнения включения L и R сбалансированность нарушается и дерево нужно перестраивать.

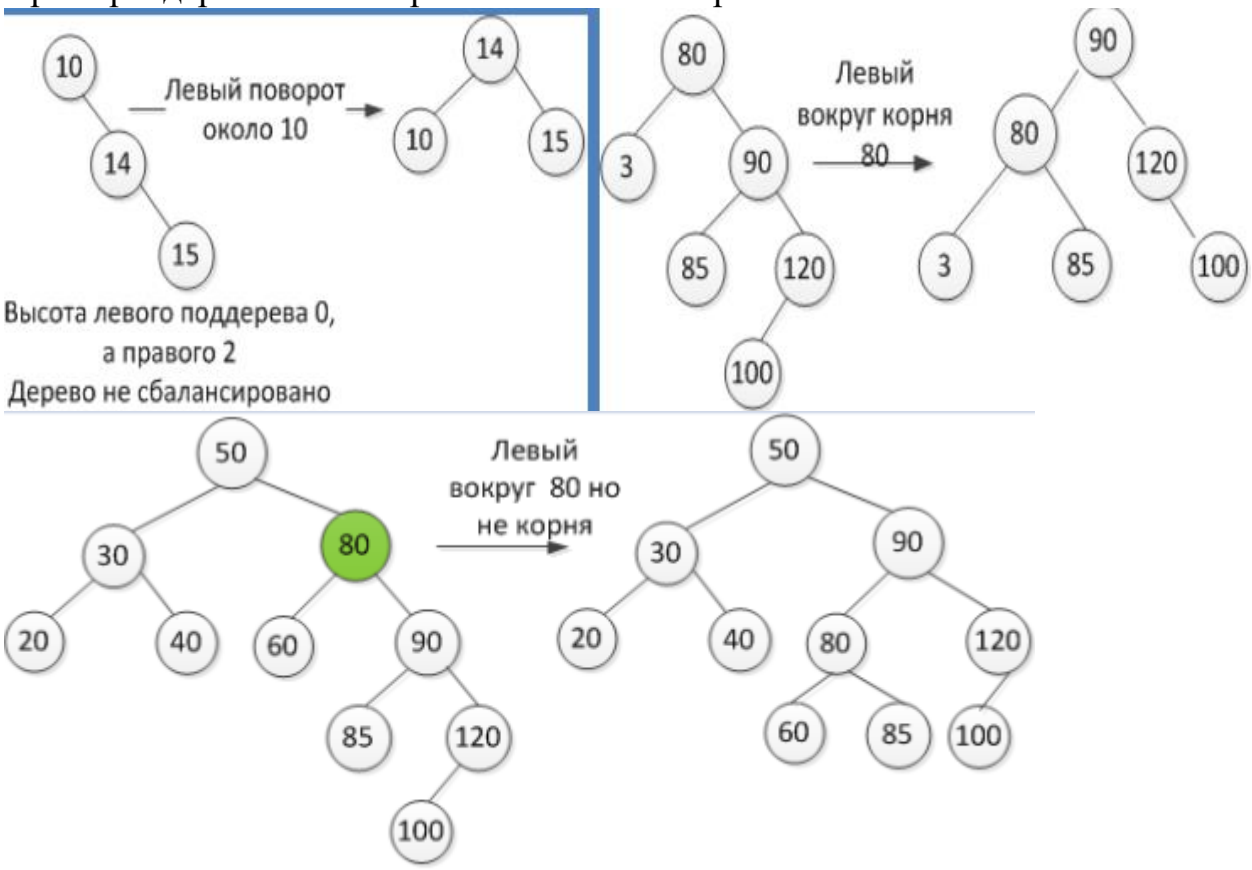
Примечание. Аналогичные случаи и для правого поддерева R.

Балансировка АВЛ – 4 случая

В зависимости от того, в какое поддерево опорного узла был добавлен новый узел, рассматривается четыре случая, которые можно разбить на две пары симметричных друг другу случаев. В каждом из них баланс восстанавливается с помощью одного или двух поворотов.

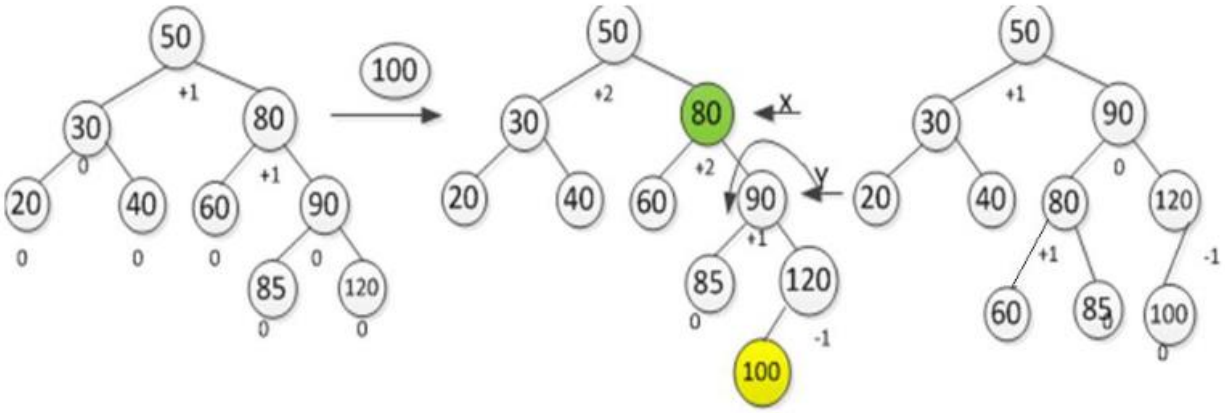
Поворот: алгоритм перестройки дерева вокруг X-опорного элемента, который определяет требуемый порядок элементов.
• Добавление в правое поддерево правого сына опорного узла (левый поворот L) – превышение высоты правого поддерева.
• Добавление в левое поддерево левого сына опорного узла (правый поворот R) – превышение высоты левого поддерева.
• Добавление в правое поддерево левого сына опорного узла.
• Добавление в левое поддерево правого сына опорного узла.
Все узлы, которые не принадлежат поддеревьям узла X при повороте не затрагиваются.
Основные алгоритмы поворотов:
• Левый поворот L
• Правый поворот R
• Двойной поворот:
    • Сначала левый, затем правый LR
    • Сначала правый, затем левый RL.

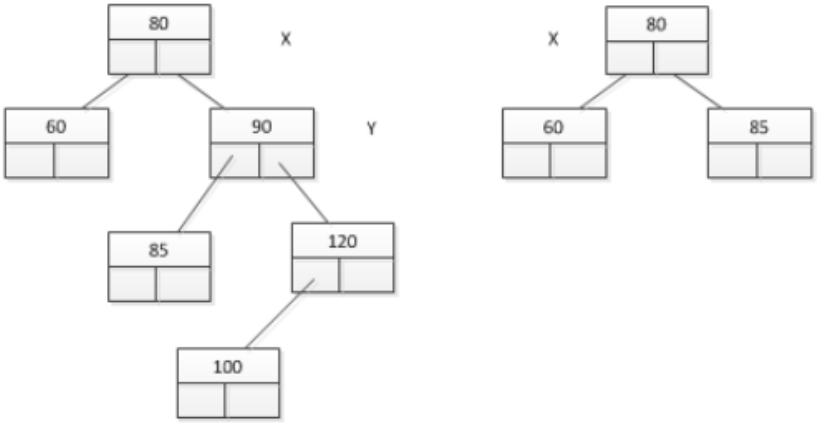
Примеры деревьев и алгоритмы левого поворота.



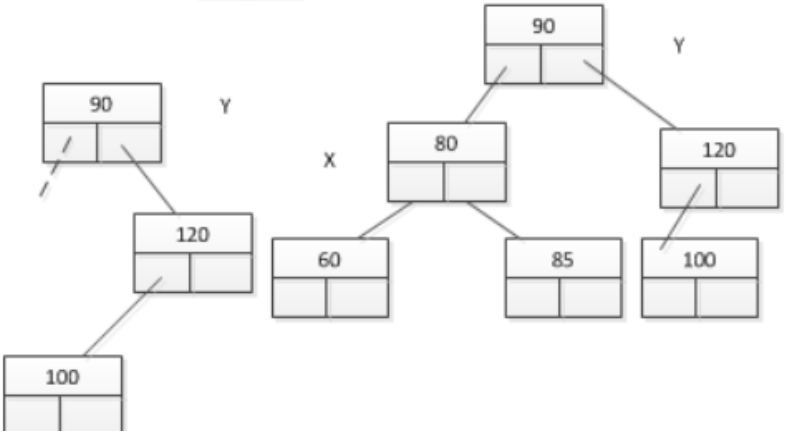
1 случай. Добавление в правое поддерево правого сына опорного узла (левый поворот L)

Необходимо произвести левый поворот (L): опорный узел (X) поворачивается налево относительно своего правого сына(Y).





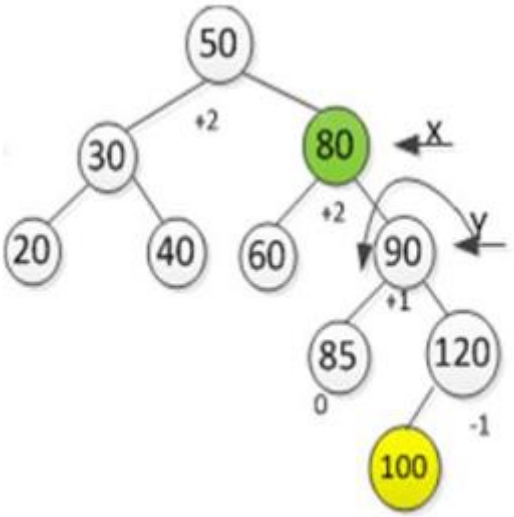
X->rigxt =Y->left
Y->left=X



Описание Алгоритма левого поворота АВЛ дерева вокруг узла X.
X – указатель на узел, у которого высота правого поддерева нарушает правило балансировки. В алгоритмах поворотов сами узлы не перемещаются, а изменяются значения ссылок на дочерние узлы и возможно родительские.
1. Сохраняем ссылку на правое поддерево узла X в указателе Y
2. Левое поддерево правого дочернего узла X (а это Y->left) перемещается на место правого поддерева узла X
3. Узел X перемещается на место левого поддерева узла Y Этот алгоритм в коде реализуют всего два оператора
X->right=Y->left;
Y->left = x;
Примечание. Но иногда смещается и корень, поэтому в узел дерева включена ссылка на родителя – поле parent. Использование родительских узлов увеличивает код, но не увеличивает время выполнения алгоритма.
Алгоритмы поворотов не перемещают узлы, а манипулируют указателями на узлы.

```
//постусловие: выполняет левый поворот вокруг
узла на который указывает x
rotateLeft(AVLNode* x){
AVL Node *y←right(x)
right(x) ←left(y)
If(left(y) not is Null)//есть левое п/д
parent(left(y)) ←x
parent(y) ←parent(x)
endif
If(x=root).//если x корень
root ← y;
Else
if(x=left(parent(x))//если x левый дочерний
элемент
left(parent(x)) ← y
else
right(parent(x)) ←y
Endif


left(y) ← x
parent(x)=y;
}
```

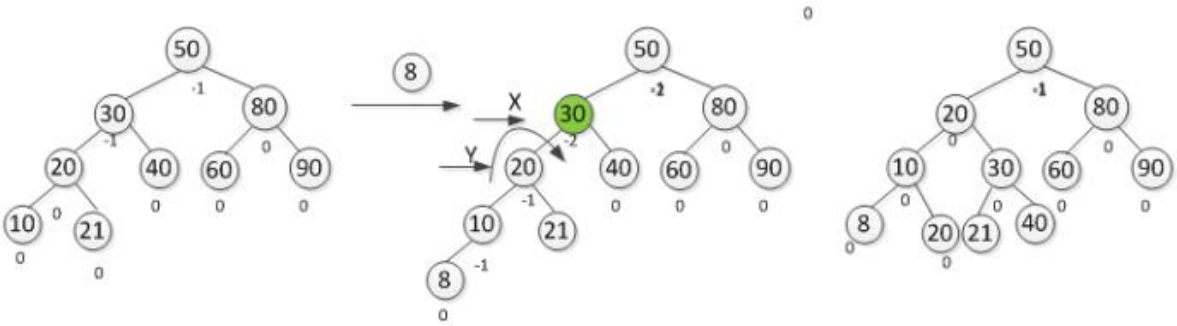


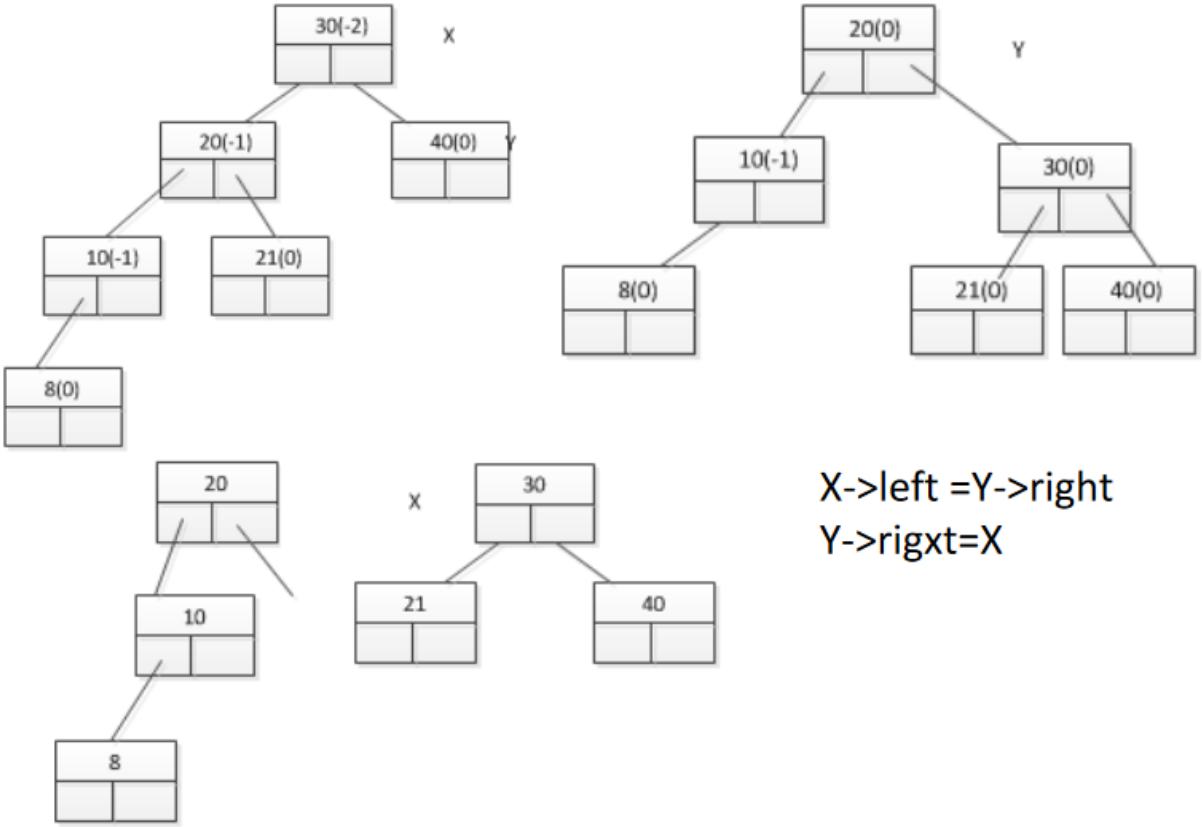

```
Struct AVLNode{
int key;
AVLNode *left,
*riht,*parent;
int balance;
};
left(A) – A->left
right(A) – A->right
parent(A) – A->parent
```

2 случай. Добавление в левое поддерево левого сына опорного узла (правый поворот R)

Алгоритм правого поворота аналогичен алгоритму левого поворота, только ссылку left надо заменить на right.

1. Элемент X перемещается на место правого дочернего узла У
2. Левый дочерний узел узла X перемещается на место самого узла X
3. Правое поддерево левого дочернего узла X становится левым поддеревом узла X





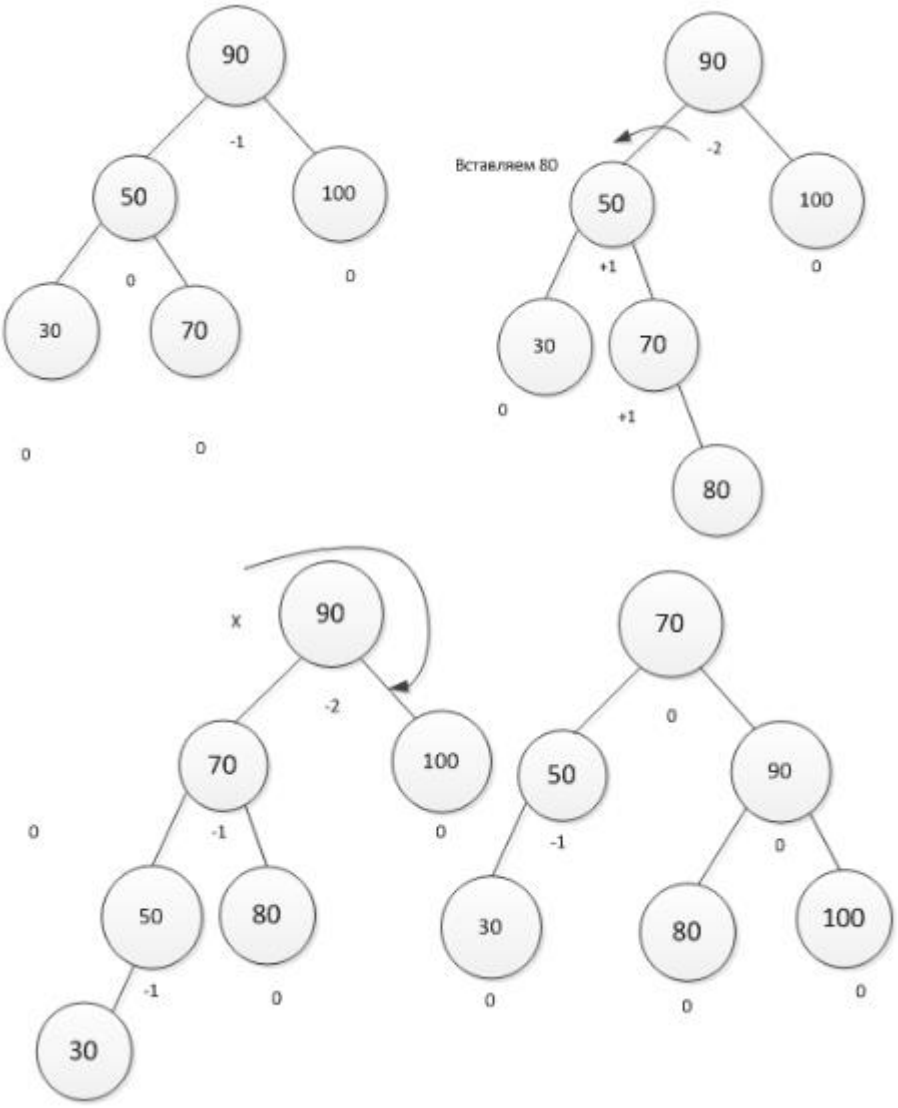
rotateLeft(T,x)
/* На вход подается дерево T и опорный узел x. */
y ← left(x) /
left(x) ← right(y)
If right(y) not is null then
parent(right(y)) ← x /* отсоединяем x от его родителя
и присоединяем y вместо x. */
parent(y) ← parent(x)
If (parent(x) is null then /* x-корень*/
root(T) ← y /* то корень y*/
else if (x = right(parent(x)) then /*x-правая ветвь родителя*/
right(parent(x)) ← y то привешиваем y по пр.в
else
left(parent(x)) ← y /* Соединяем x и y. */
right(y) ← x
parent(x) ← y}

3 случай. Добавление нового узла в правое поддерево левого сына опорного узла.

Если узел вставляется в правое поддерево левого поддерева опорного узла X то выполняется двойной поворот LR Необходимо произвести двойной поворот — налево, потом направо(LR):

• сначала левый сын опорного узла (X) поворачивается налево относительно своего правого сына (Y),

• а затем опорный узел (Z) поворачивается направо относительно своего нового левого сына (Y).



Алгоритм поворота LR

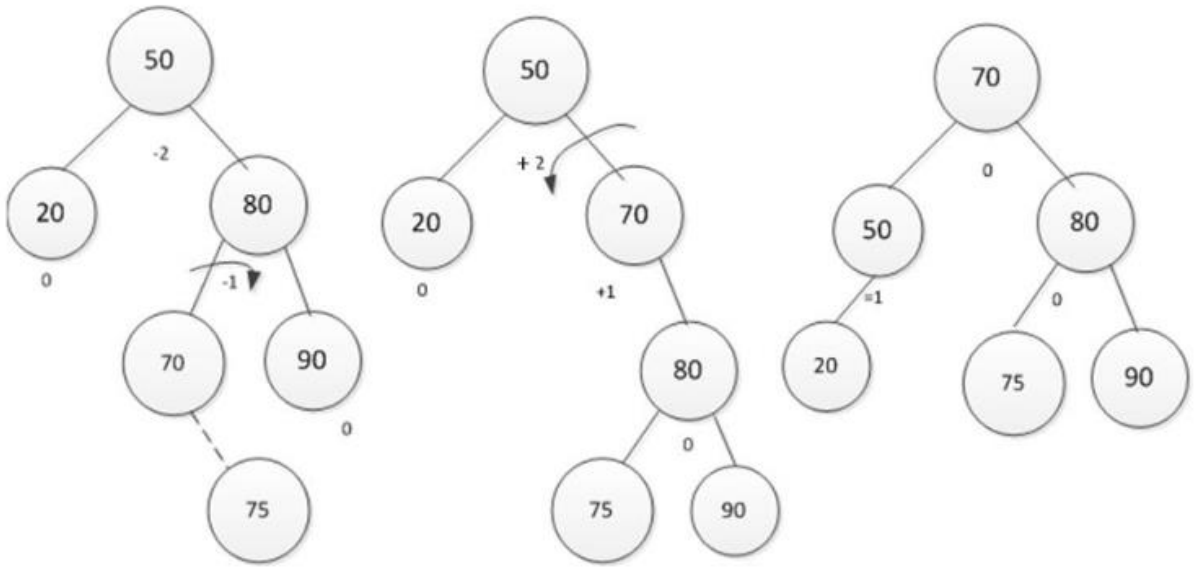
```
rotateLR(T,x){
rotateLeft(T,x->left)
rotateRight(T,x)
}
```

4 Случай. Добавление нового узла в левое поддерево правого сына опорного узла

Он симметричен алгоритму LR Если узел вставляется в левое поддерево правого поддерева опорного узла X то выполняется двойной поворот RL

• сначала правый сын опорного узла (X) поворачивается направо относительно своего левого сына (Y),

• а затем опорный узел (Z) поворачивается налево относительно своего нового правого сына (Y).



Общее правило поворотов

Если добавление нового узла, приводящее к разбалансировке, происходит в левое поддерево левого сына опорного узла или в правое поддерево правого сына опорного узла, т .е. если стороны сына и внука опорного узла одноименны, то необходимо произвести одинарный поворот .

Если добавление происходит в правое поддерево левого сына опорного узла или в левое поддерево правого сына опорного узла, т.е. стороны разноименны, то необходимо произвести двойной поворот.

Определение :Глубина узла равна длине простого пути от корня до этого узла. Таким образом, поворот производится в противоположную сторону. Визуально это правило выглядит так, что если самый глубокий узел(тот, который был добавлен последним)находится слева или справа, то производится одинарный поворот опорного узла относительно поддерева, содержащего этот узел, в противоположную сторону, чтобы выровнять высоты.

Если самый глубокий узел находится посередине, то потребуется двойной поворот.

Алгоритм балансировки

На вход подается дерево T и узел x, в котором надо восстановить баланс, если он был нарушен.

```
AVLRestorBALANCE(T , x){
If (x.balance< –1 then //у узла x высота левого поддерева
//больше высоты правого  поддерева
if (height(left(left(x))))>height((right(left(x))) then/*  самый глубокий узел
rotateLeft(T,x)  слева*/
Else
/* самый глубокий узел посередине */
rotateLR(T,x)
If x.balance > 1then // у узла x высота правого поддерева
//больше высоты левого поддерева
If height(right(right(x)))>height(left(right(x)))then / / самый глубокий
rotateLeft(T,x)  узел справа
Else
// самый глубокий узел посередине
rotateRL(T,x)
}
```

Алгоритм вставки узла в АВЛ дерево

На вход подается дерево T и узел x (указатель), который надо добавить в дерево.

```
insertInAVL(T, x){
insertBTS(T , x) // вставка в бинарное дерево поиска
/* Поскольку мы не знаем, где именно находится опорный узел, проходим по
всем узлам, лежащим на пути от вставленного узла X к корню, и вызываем
процедуру восстановления баланса. */
current ← x
while current ≠ NULL do
AVLRestorBALANCE(T , current)
current ← parent(current)
od
}
```

Удаление узла из АВЛ дерева

1. Удаление узла по алгоритму удаления узла из двоичного дерева поиска

Таким образом, если у узла менее двух сыновей, то удаляется сам узел, а если два сына, то удаляемым узлом становится его последователь, информация (ключ) из которого предварительно переписывается в удаляемый узел.

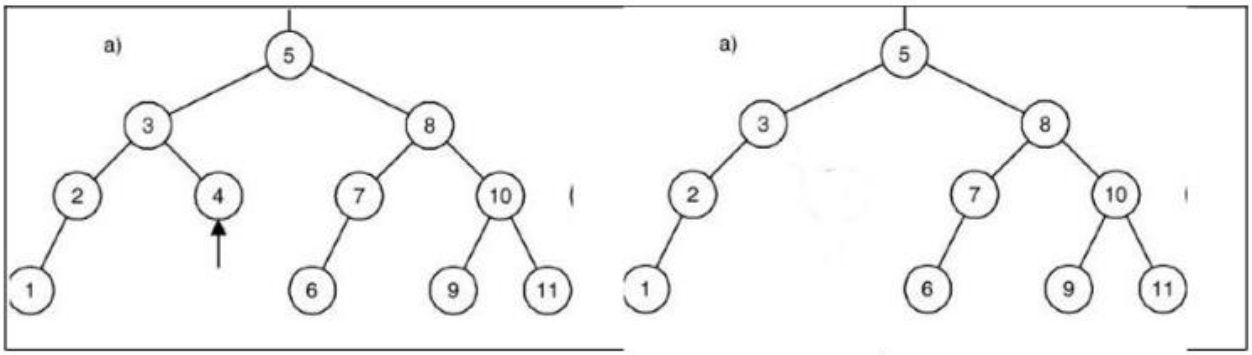
2. Балансировка дерева, если необходимо

Чтобы после удаления сохранились свойства АВЛ-дерева, возможно, понадобится выполнить балансировку. Для этого надо подниматься вверх по пути от удаленного узла к корню и проверять в этих узлах баланс. Если в узле баланс нарушен, то надо выполнить соответствующий поворот – одинарный или двойной. Остановить просмотр можно на том узле, в котором показатель баланса не поменялся. Это означает, что высота его поддерева, левого или правого, в котором производилось удаление, не изменилась.

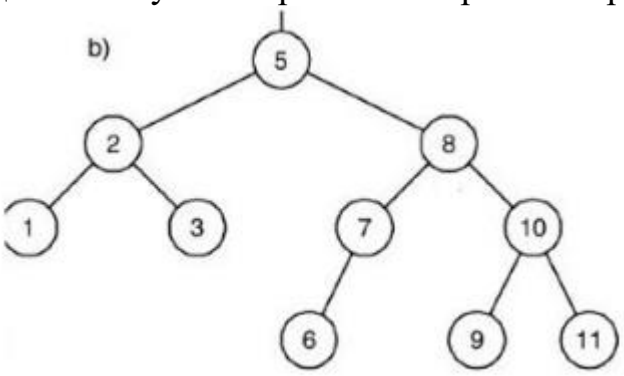
Правила поворотов после удаления узла такие же, как и в случае добавления узла:

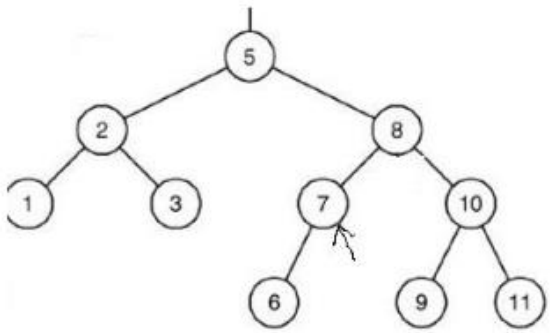
• если самые глубокие узлы находятся справа или слева, то производится одинарный поворот опорного узла в противоположную сторону,

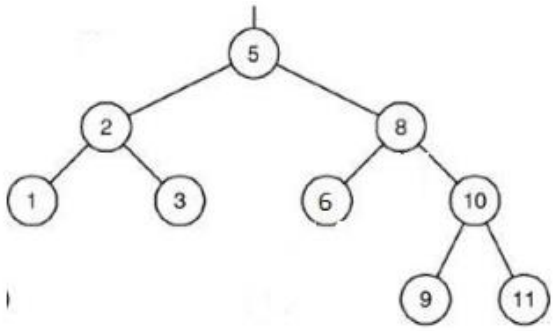
• а если они находятся посередине, то производится двойной поворот



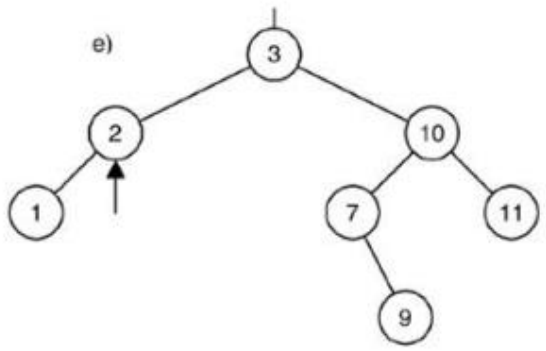
Если удаляемый узел x – лист, то алгоритм удаления сводиться к простому исключению узла x, и подъему к корню с переопределением балансов узлов. В данном случае – правый поворот с опорным узлом 3.

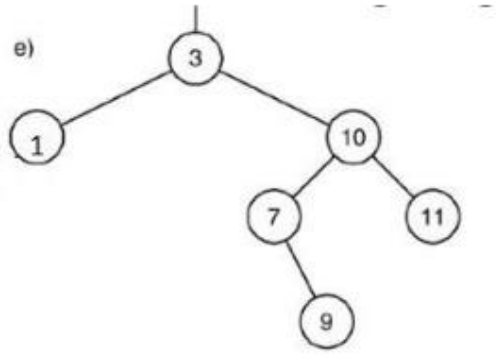






если же x не является листом, то он либо имеет правое поддерево, либо левое. Узел 7 имеет левое поддерево. В этом случае, из свойства АВЛ-дерева, следует, что левое поддерево имеет высоту 1, и здесь алгоритм удаления сводиться к тем же действиям, что и при терминальном узле. Исключению узла x, и подъему к корню с переопределением балансов узлов. В данном случае поворот не потребовался.





Удаление узла 2 простая операция, но в данном случае после удаления узла 2 и пересчета баланса потребовался двойной поворот RL : относительно 10 R и затем относительно 3 L

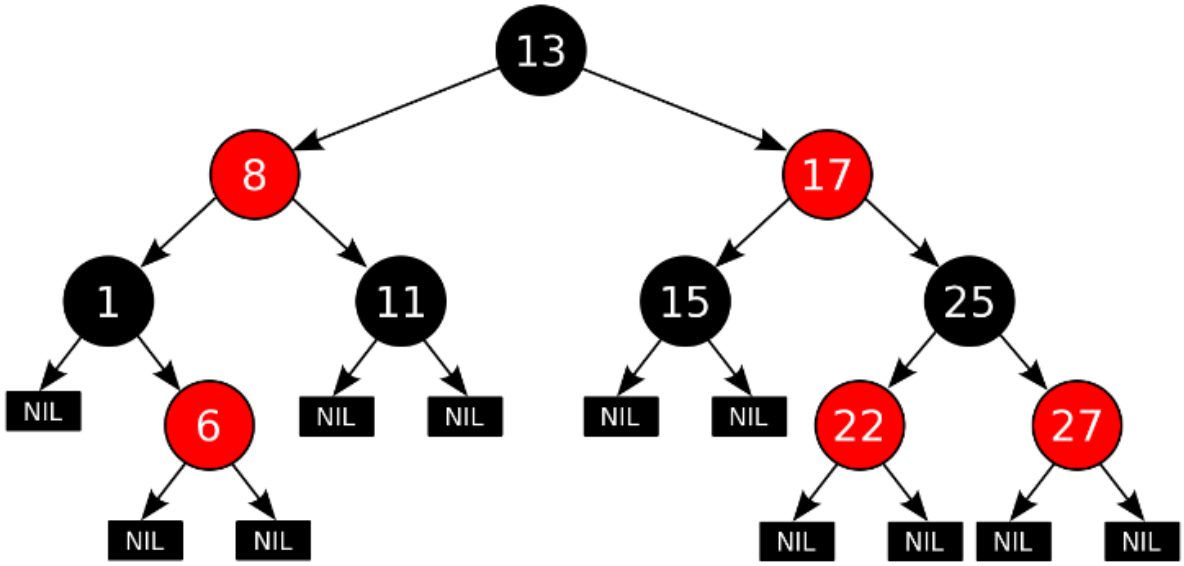
Алгоритм удаления узла из АВЛ дерева

На вход подается дерево T и узел x, который надо удалить из дерева.

```
deleteNodeFromAVL(T, x){
deleted= deleteNodeBST(T , x)//указатель на удален.узел
/* проходим по всем узлам, лежащим на пути от родителя удаленного
узла к корню, и восстанавливаем в каждом из них баланс, если он был
нарушен. */
current ← parent[deleted]
while current ≠ NULL do
AVLRestorBALANCE(T, current)
current ← parent[current]
od
}
```

Красно-черные деревья
АВЛ-деревья исторически были первым примером использования сбалансированных деревьев поиска. В настоящее время более популярны красно-черные деревья (КЧ-деревья).
Изобретателем красно-черного дерева считается немецкий ученый Рудольф Байер. Название эта структура данных получила в статье Леонидаса Гимпаса и РобертаСеджвика 1978 года.

КЧ-деревья – это двоичные деревья поиска, каждый узел которых хранит дополнительное поле color , обозначающее цвет: красный или черный, и для которых выполнены приведенные ниже свойства.
Будем считать, что если left или right равны NULL, то это «указатели» на фиктивные листья.
В КЧ-дереве все узлы – внутренние (нелистовые).
Свойства красно-черных деревьев:
1) Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).
2) Корень окрашен в черный цвет.
3) Листья (так называемые NULL-узлы) окрашены в черный цвет.
4) Каждый красный узел должен иметь два черных дочерних узла. У черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.
5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов (это черная высота).



Почему красно-черное дерево сбалансировано?
Красно-черные деревья не гарантируют строгой сбалансированности (разница высот двух поддеревьев любого узла не должна превышать 1), как в АВЛ-деревьях. Но соблюдение свойств красно-черного дерева позволяет обеспечить выполнение операций вставки, удаления и выборки за время O(logN).

Вставка узла в красно-черное дерево
Чтобы вставить узел, мы сначала ищем в дереве место, куда его следует добавить. Новый узел всегда добавляется как лист, поэтому оба его потомка являются NIL-узлами и предполагаются черными. После вставки красим узел в красный цвет. После этого смотрим на предка и проверяем, не нарушается ли красно-черное свойство. Если необходимо, мы перекрашиваем узел и производим поворот, чтобы сбалансировать дерево.

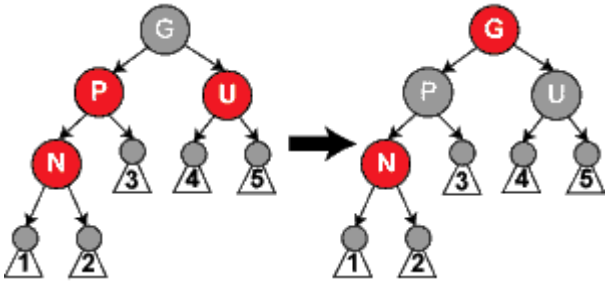
Примечание: Буквой N будем обозначать текущий узел (окрашенный красным). Сначала это новый узел, который вставляется, но эта процедура может применяться рекурсивно к другим узлам (смотрите случай 3). P будем обозначать предка N, через G обозначим дедушку N, а U будем обозначать дядю (узел, имеющий общего родителя с узлом P).

Случай 1: Текущий узел N в корне дерева. В этом случае, он перекрашивается в чёрный цвет, чтобы оставить верным Свойство 2 (Корень — чёрный). Так как это действие добавляет один чёрный узел в каждый путь, Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается.

Случай 2: Предок P текущего узла чёрный, то есть Свойство 4 (Оба потомка каждого красного узла — чёрные) не нарушается. В этом случае дерево остаётся корректным. Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается, потому что текущий узел N имеет двух чёрных листовых потомков, но так как N является красным, путь до каждого из этих потомков содержит такое же число чёрных узлов, что и путь до чёрного листа, который был заменен текущим узлом, так что свойство остается верным.

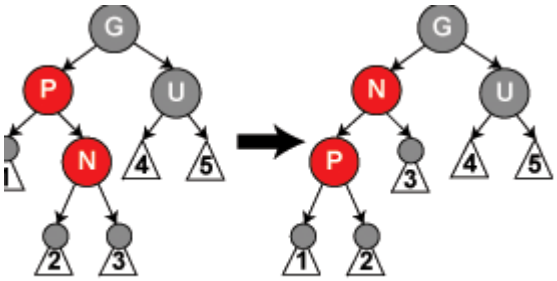
Случай 3: Если и родитель P, и дядя U — красные, то они оба могут быть перекрашены в чёрный, и дедушка G станет красным (для сохранения свойства 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов)). Теперь у текущего красного узла N чёрный родитель. Так как любой путь через родителя или дядю должен проходить через дедушку, число чёрных узлов в этих путях не изменится. Однако, дедушка G теперь может нарушить свойства 2 (Корень — чёрный) или 4 (Оба потомка каждого красного узла — чёрные) (свойство 4 может быть

нарушено, так как родитель G может быть красным). Чтобы это исправить, вся процедура рекурсивно выполняется на G из случая 1.



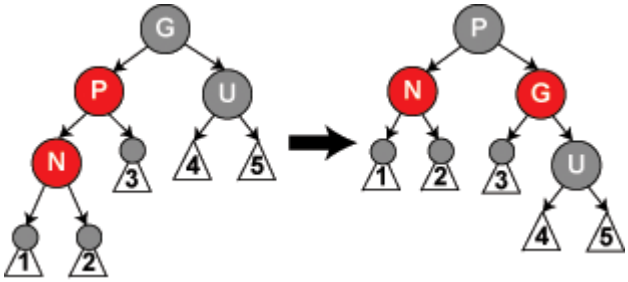
Примечание: В оставшихся случаях предполагается, что родитель P является левым потомком своего предка. Если это не так, необходимо поменять лево и право.

Случай 4: Родитель P является красным, но дядя U — чёрный. Также, текущий узел N — правый потомок P, а P в свою очередь — левый потомок своего предка G. В этом случае может быть произведен поворот дерева, который меняет роли текущего узла N и его предка P. Тогда, для бывшего родительского узла P в обновленной структуре используем случай 5, потому что Свойство 4 (Оба потомка любого красного узла — чёрные) все ещё нарушено. Вращение приводит к тому, что некоторые пути (в поддереве, обозначенном «1» на схеме) проходят через узел N, чего не было до этого. Это также приводит к тому, что некоторые пути (в поддереве, обозначенном «3») не проходят через узел P. Однако, оба эти узла являются красными, так что Свойство 5 (Все пути от любого данного узла до листовых узлов содержат одинаковое число чёрных узлов) не нарушается при вращении. Однако Свойство 4 всё ещё нарушается, но теперь задача сводится к Случаю 5.



Случай 5: Родитель P является красным, но дядя U — чёрный, текущий узел N — левый потомок P и P — левый потомок G. В этом случае выполняется поворот дерева на G. В результате получается дерево, в котором бывший родитель P теперь является родителем и текущего узла N и бывшего дедушки G. Известно, что G — чёрный, так как его бывший потомок P не мог бы в противном случае быть красным (без нарушения Свойства 4). Тогда цвета P и G меняются и в результате дерево удовлетворяет Свойству 4 (Оба потомка любого красного узла — чёрные). Свойство 5 (Все пути от любого данного узла до листовых узлов содержат

одинаковое число чёрных узлов) также остается верным, так как все пути, которые проходят через любой из этих трех узлов, ранее проходили через G, поэтому теперь они все проходят через P. В каждом случае, из этих трёх узлов только один окрашен в чёрный.



Сравнение АВЛ и КЧ

AVL-деревья обеспечивают более быстрый поиск, чем КЧ, потому что они более строго сбалансированы. Красные чёрные деревья обеспечивают более быстрые операции вставки и удаления, чем деревья AVL, так как выполняется меньше поворотов благодаря относительно расслабленной балансировке. Деревья AVL хранят балансовые коэффициенты или высоты для каждого узла, таким образом, требуется хранилище для целого числа на узел, тогда как для КЧ требуется только 1 бит информации на узел. Красные чёрные деревья используются в большинстве языковых библиотек, таких как map , multimap , multiset в C ++, тогда как деревья AVL используются в базах данных, где требуется более быстрый поиск.

Самоперестраивающееся дерево (косое)
Это двоичное дерево поиска, которое, в отличие от предыдущих двух видов деревьев не содержит дополнительных служебных полей в структуре данных (баланс, цвет и т .п.). Оно позволяет находить быстрее те данные, которые использовались недавно.
Самоперестраивающееся дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году .

Идея самоперестраивающихся деревьев основана на принципе перемещения найденного узла в корень дерева. Эта операция называется splay(T, k), где k– это ключ, а T–двоичное дерево поиска. После выполнения операции splay(T, k) двоичное дерево T перестраивается, оставаясь при этом деревом поиска, так, что:
• если узел с ключом k есть в дереве, то он становится корнем;
• если узла с ключом k нет в дереве, то корнем становится его предшественник или последователь.
Поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay . Эвристика move-to-front (перемещение найденного узла в корень) основана на предположении, что если тот же самый элемент потребуется в ближайшее время, он будет найден быстрее

Алгоритм вставки узла в самоперестраивающееся дерево

1. Поиск узла в этом дереве с помощью операции splay(T, k).
2. Затем, проверяется значение ключа в корне
    1) Если оно равно k, значит, узел найден.
    2) Если же оно не равно k, значит, в корне находится его предшественник или последователь.

Тогда k становится новым корнем и, в зависимости от того, было ли до этого в корне значение, большее k или меньшее, старый корень становится правым или, соответственно, левым сыном корня.

На вход подается дерево T и узел x, который надо добавить в дерево.

```
SPLAYINSERT(T, x){
If root[T] = NULL then
root[T] ← x
return
SPLAY(T, key[x]);
If key[root[T]] < key[x] then
/* в корне находится предшественник x */
left[x] ← root[T]
right[x] ← right[root[T]]
if right[x] ≠ NULL then
parent[right[x]] ← x1
else /* в корне находится последователь x */
right[x] ← root[T]
left[x] ← left[root[T]]
if left[x] ≠ NULL then
parent[left[x]] ← x
endif
parent[root[T]] ← x
root[T] ← x
}
```

Алгоритм операции splay(T,k).

1. Поиск узла с ключом k в дереве обычным способом, спускаясь вниз, начиная с корня.

При этом запоминается пройденный путь. В итоге, получаем указатель на узел дерева либо с ключом k, либо с его предшественником или последователем, на котором закончился поиск.

2. Возвращение назад по запомненному пути, с перемещением этого узла к корню. Для того, чтобы при этом сохранялись свойства двоичного дерева поиска, необходимы повороты.

Рандомизированное бинарное дерево поиска (англ. *Randomized binary search tree, RBST*) — структура данных, реализующая бинарное дерево поиска.

Определение: Пусть T — бинарное дерево поиска. Тогда

1. Если T пусто, то оно является **рандомизированным бинарным деревом поиска**.
2. Если T непусто (содержит n вершин, $n>0$), то T — **рандомизированное бинарное дерево поиска** тогда и только тогда, когда его левое и правое поддеревья (L и R) оба являются **RBST**, а также выполняется соотношение $P[size(L)=i]=1/n$, $i=1..n$.

Из определения следует, что каждый ключ в RBST размера n может оказаться корнем с вероятностью 1/n.
Идея RBST состоит в том, что хранимое дерево постоянно является рандомизированным бинарным деревом поиска. Заметим лишь, что хранение RBST в памяти ничем не отличается от хранения обычного дерева поиска: хранится указатель на корень; в каждой вершине хранятся указатели на её сыновей.

Вставка

Рассмотрим рекурсивный алгоритм вставки ключа x в RBST, состоящее из n вершин. С вероятностью 1/(n+1) вставим ключ в корень дерева (разделим дерево по данному ключу и подвесим получившиеся деревья к новому корню), используя процедуру insertAtRoot. С вероятностью 1−1/(n+1)=n/(n+1) вставим его в правое поддерево, если он больше корня, или в левое поддерево, если меньше. Ниже приведён псевдокод процедуры вставки insert, процедуры insertAtRoot, а также процедуры split(k), разбивающей дерево на два поддерева, в одном из которых все ключи строго меньше k, а в другом больше, либо равны; приведена достаточно очевидная рекурсивная реализация (через Node обозначен тип вершины дерева, дерево представляется как указатель на корень).

```
Node insert(t : Node, x : T):
  int r = random(0 …
 t.size)
  if r == t.size
    t = insertAtRoot(t, x)
  if x < t.key
    t = insert(t.left, x)
  else
    t = insert(t.right, x)
  t.size = 1 + t.size
  return t
```

Заметим, что если дерево пусто, то insert с вероятностью 1 делает x корнем.

```
Node insertAtRoot(t : Node, x : T):        // вставляем в дерево t ключ x
  <l, r> = split(t, x)
  t.key = x
  t.left = l
  t.right = r
  return t
<Node, Node> split(t : Node, x : T):              // разделяет дерево t по x,
результат — пара деревьев r и l
  if t.size == 0
    return <null, null>
  else if x < t.key
    <l, r> = split(t.left, x)
    t.left = r
    t.size = 1 + t.left.size + t.right.size
    r = t
    return <l, r>
  else
    <l, r> = split(t.right, x)
    t.right = l
    t.size = 1 + t.left.size + t.right.size
    l = t
    return <l, r>
```

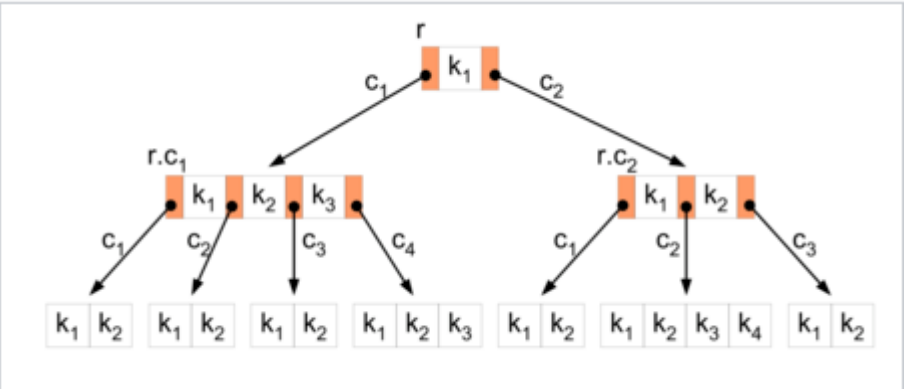
Удаление

Алгоритм удаления использует операцию merge — слияние двух деревьев, удовлетворяющих условию: все ключи в одном из деревьев меньше ключей во втором. Для того, чтобы удалить некоторый ключ x из RBST сначала найдём вершину с этим ключом в дереве, используя стандартный алгоритм поиска. Если вершина не найдена, то выходим из алгоритма; в противном случае сливаем правое и левое поддеревья x (заметим, что ключи в левом поддереве меньше ключей в правом), удаляем x, а корень образовавшегося дерева делаем новым сыном родителя x. Псевдокод процедур удаления и слияния приведён ниже.

```
Node remove(t : Node, x : T):       // удаляет ключ x из дерева T
  if t.size == 0
    t = null
    return t                  // вернуть пустое поддерево
  if x < t.key
    t.left = remove(t.left, x)
  else if x > t.key
    t.right = remove(t.right, x)
  else
    q = merge(t.left, t.right)
    t = q
  return t
Node merge(l : Node, r : Node):          // сливает деревья l и r, результат — дерево t
  int m = l.size
  int n = r.size
  int total = m + n
  if total == 0
    t = null
    return t                      // вернуть пустое поддерево
  int r = random(1 …
 total)
  if r < m
    l.right = merge(l.right, r)         // с вероятностью m / (m + n)
    l.size = 1 + l.left.size + l.right.size
    return l
  r.left = merge(l, r.left)           // с вероятностью n / (m + n)
  r.size = 1 + r.left.size + r.right.size
  return r
```

B-дерево (англ. B-tree) — сильноветвящееся сбалансированное дерево поиска, позволяющее проводить поиск, добавление и удаление элементов за O(logn). B-дерево с n узлами имеет высоту O(logn). Количество детей узлов может быть от нескольких до тысяч (обычно степень ветвления B-дерева определяется характеристиками устройства (дисков), на котором производится работа с деревом). B-деревья также могут использоваться для реализации многих операций над динамическими множествами за время O(logn) B-дерево было впервые предложено Р. Бэйером и Е. МакКрейтом в 1970 году.

B-дерево является идеально сбалансированным, то есть глубина всех его листьев одинакова.
B-дерево имеет следующие свойства (t — параметр дерева, называемый минимальной степенью B-дерева, не меньший 2.):
- Каждый узел, кроме корня, содержит не менее t−1 ключей, и каждый внутренний узел имеет по меньшей мере t дочерних узлов. Если дерево не является пустым, корень должен содержать как минимум один ключ.
- Каждый узел, кроме корня, содержит не более 2t−1 ключей и не более чем 2t сыновей во внутренних узлах
- Корень содержит от 1 до 2t−1 ключей, если дерево не пусто и от 2 до 2t детей при высоте большей 0.
- Каждый узел дерева, кроме листьев, содержащий ключи k1,...,kn, имеет n+1 сына. i-й сын содержит ключи из отрезка [ki−1;ki],k0=−∞,kn+1=∞.
- Ключи в каждом узле упорядочены по неубыванию.
- Все листья находятся на одном уровне.



Пример B-дерева со степенью 3

Структура узла

```
struct Node
  bool leaf    // является ли узел листом
  int  n       // количество ключей узла
```

```
  int  key[]   // ключи узла
  Node c[]     // указатели на детей узла
```

Структура дерева

```
struct BTree
  int  t      // минимальная степень дерева
  Node root   // указатель на корень дерева
```

B-деревья разработаны для использования на дисках (в файловых системах) или иных энергонезависимых носителях информации с прямым доступом, а также в базах данных. B-деревья похожи на красно-чёрные деревья (например, в том, что все B-деревья с n узлами имеют высоту O(logn)), но они лучше минимизируют количество операций чтения-записи с диском.

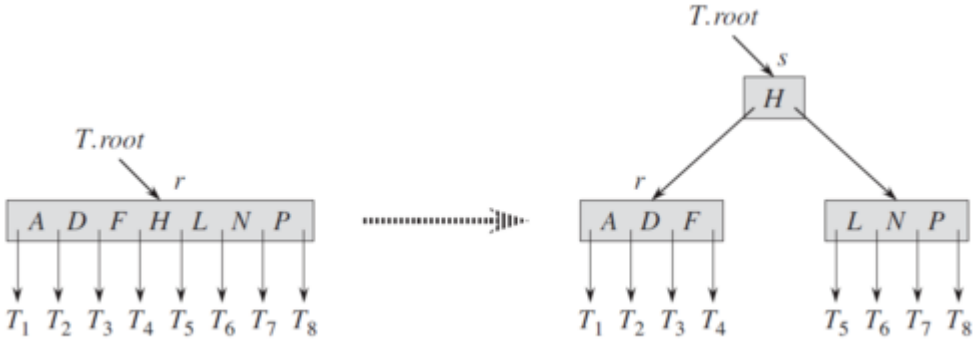
B-деревья представляют собой сбалансированные деревья, поэтому время выполнения стандартных операций в них пропорционально высоте. Однако, как уже было упомянуто выше, алгоритмы B-дерева созданы специально для работы с дисками (или другими носителями информации) и базами данных (или иными видами представления большого количества информация), минимизируя количество операций ввода-вывода.

Поиск ключа

Если ключ содержится в текущем узле, возвращаем его. Иначе определяем интервал и переходим к соответствующему сыну. Повторяем пока ключ не найден или не дошли до листа.

Добавление ключа

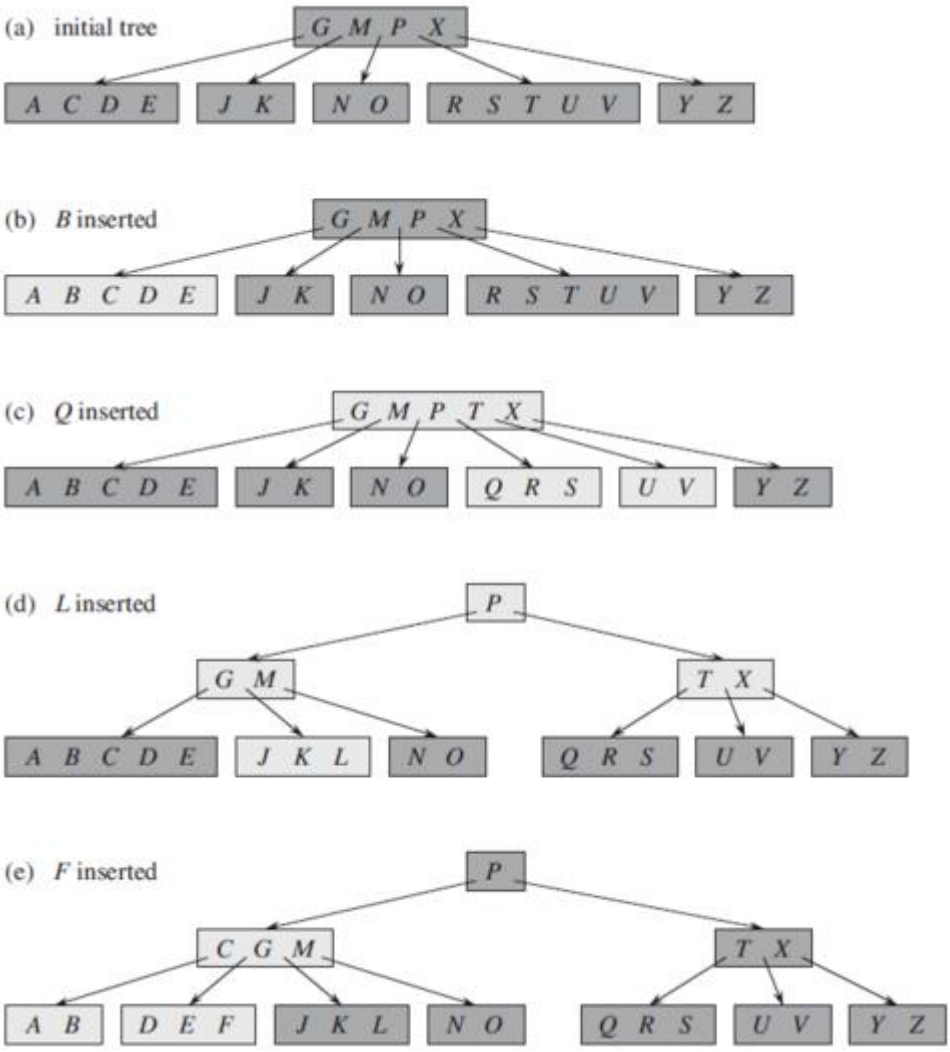
Ищем лист, в который можно добавить ключ, совершая проход от корня к листьям. Если найденный узел незаполнен, добавляем в него ключ. Иначе разбиваем узел на два узла, в первый добавляем первые $t-1$ ключей, во второй — последние $t-1$ ключей. После добавляем ключ в один из этих узлов. Оставшийся средний элемент добавляется в родительский узел, где становится разделительной точкой для двух новых поддеревьев.



Если и родительский узел заполнен — повторяем пока не встретим незаполненный узел или не дойдем до корня. В последнем случае корень разбивается на два узла и высота дерева увеличивается. Добавление ключа в B-дереве может быть осуществлена за один нисходящий проход от корня к листу. Для этого не нужно выяснять, требуется ли разбить узел, в который должен вставляться новый ключ. При проходе от корня к листьям в поисках места для нового ключа будут разбиваться все заполненные узлы, которые будут пройдены (включая и сам лист). Таким образом, если надо разбить какой-то полный узел, гарантируется, что его родительский узел не будет заполнен.

```
void B-Tree-Insert(T: BTree, k: int):
   r = T.root
   if r.n == 2T.t - 1
      s = Allocate-Node()
      T.root = s
      s.leaf = false
      s.n = 0
      s.c[1] = r
      B-Tree-Split-Child(s, T.t, 1)
      B-Tree-Insert-Nonfull(s, k, T.t)
   else
      B-Tree-Insert-Nonfull(r, k, T.t)
void B-Tree-Insert-Nonfull(x: Node, k: int, t: int):
   i = x.n
   if x.leaf
      while i >= 1 and k < x.key[i]
         x.key[i+1] = x.key[i]
         i = i - 1
   x.key[i+1] = k
   x.n = x.n + 1
   Disk-Write(x)
   else
      while i >= 1 and k < x.key[i]
         i = i - 1
      i = i + 1
      Disk-Read(x.c[i])
      if x.c[i].n == 2t - 1
         B-Tree-Split-Child(x, t, i)
         if k > x.key[i]
            i = i + 1
      B-Tree-Insert-Nonfull(x.c[i], k, t)
```

Функция B-Tree-Insert-Nonfull вставляет ключ k в узел x, который должен быть незаполненным при вызове. Использование функции B-Tree-Split-Child гарантирует, что рекурсия не встретится с заполненным узлом. Ниже показана вставка ключей B, Q, L и F в дерево с t=3, т.е. узлы могут содержать не более 5 ключей



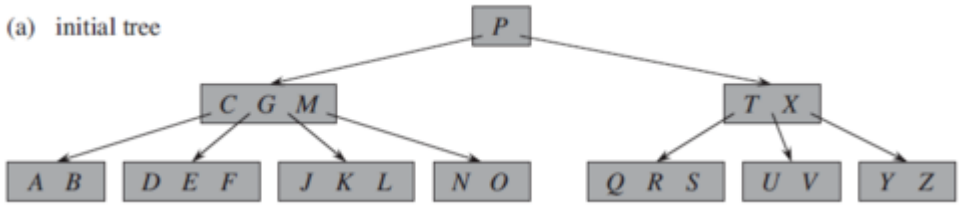
Удаление ключа

Операция удаления ключа несколько сложнее, нежели добавление оного, так как необходимо убедиться, что удаляемый ключ находится во внутреннем узле. Процесс похож на поиск подходящего места для вставки ключа, с той разницей, что перед спуском в поддерево проверяется, достаточность

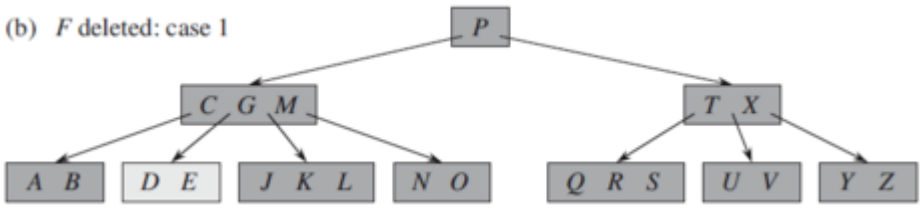
количества ключей (т.е. ⩾t) в нем, а также возможность провести удаление, не нарушив структуры B-дерева. Таким образом, удаление аналогично вставке, и его проведение не потребует последующего восстановления структуры B-дерева. Если поддерево, выбранное поиском для спуска, содержит минимальное количество ключей t−1, производится либо перемещение, либо слияние.

Удаление ключа из листа

Если удаление происходит из листа, смотрим на количество ключей в нем. Если ключей больше t−1, то просто удаляем ключ.



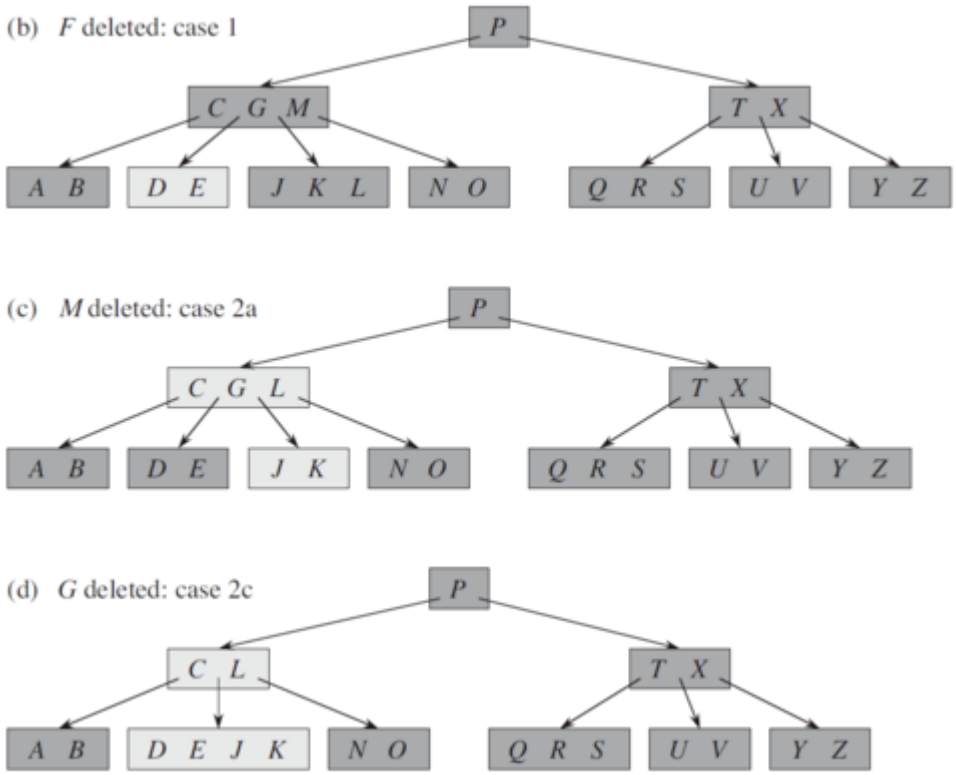
В противном случае, если существует соседний лист с тем же родителем, который содержит больше t−1 ключа, выберем ключ-разделитель из соседа разделяющий оставшиеся ключи соседа и ключи исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Обозначим этот ключ как k1. Выберем другой ключ из родительского узла, разделяющий исходный узел и его соседа, который был выбран ранее. Этот ключ обозначим k2. Удалим из исходного узла ключ, который нужно было удалить, спустим в этот узел k2, а вместо k2 в родительском узле поставим k1. Если все соседи содержат по t−1 ключу, то объединяем узел с каким-либо из соседей, удаляем ключ, и ключ из родительского узла, который был разделителем разделённых соседей, переместим в новый узел.



Удаление ключа из внутреннего узла

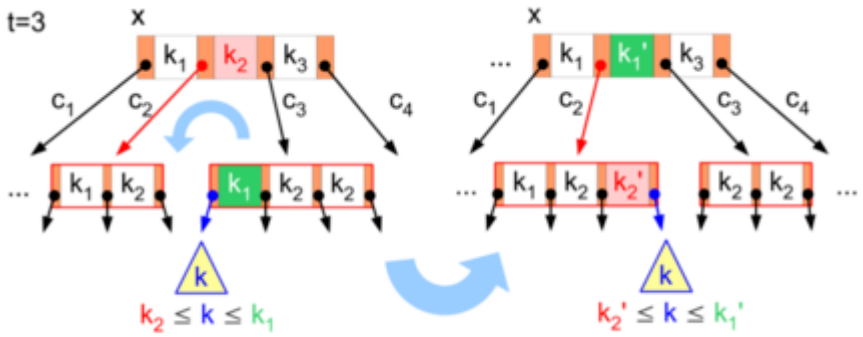
Рассмотрим удаление из внутреннего узла. Имеется внутренний узел x и ключ, который нужно удалить, k. Если дочерний узел, предшествующий ключу k, содержит больше t−1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его. Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t−1 ключа. Если оба (следующий и

предшествующий дочерние узлы) имеют по t−1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла. Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается.



Перемещение ключа

Если выбранное для нисходящего прохода поддерево содержит минимальное количество ключей t−1, и предшествующие и следующие узлы-братья имеют по меньшей мере t ключей, то ключ перемещается в выбранный узел. Поиск выбрал для спуска x.c2 (x.k1<kdelete<x.k2). Этот узел имеет лишь t−1 ключ (красная стрелка). Так как следующий брат x.c3 содержит достаточное количество ключей, самый маленький ключ x.c3.k1 может перемещаться оттуда в родительский узел, чтобы переместить, в свою очередь, ключ x.k2 как дополнительный ключ в выбранный для спуска узел. Левое поддерево x.c3.k1 — новое правое поддерево перемещённого ключа x.k2.



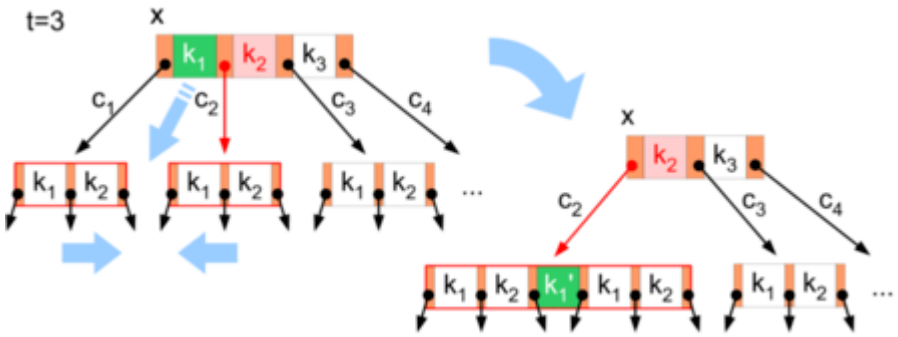
Легко убедиться в том, что эти повороты поддерживают структуру B-дерева: для всех ключей k на отложенном поддереве до и после перенесения выполняется условие $x.k_2 \leqslant k \leqslant x.c_3.k_1$.

Симметричная операция может производиться для перенесения ключа из предшествующего брата.

Слияние

Ниже будет рассмотрено слияние узлов при удалении ключей, то есть слияние узлов равной степени и высоты. Для произвольных же слияний потребуется приведение сливаемых деревьев к одной степени и высоте.

Итак, если выбранное для спуска поддерево *x.c*2 и предшествующий и следующий узел-брат содержит минимальное количество ключей, то перемещение не возможно. На иллюстрации приводится слияние выбранного поддерева с предшествующим или следующим братом для такого случая. Для этого откладывается ключ из родительского узла *x*, который разделяет ключи на два сливаемых узла, в то время средний ключ перемещается в слитый узел. Ссылки на слитые дочерние узлы заменяются ссылкой на новый узел.



Так как алгоритм гарантирует, что узел, в который будет совершаться спуск, содержит по меньшей мере t ключей вместо требуемых условиями B-дерева t−1 ключей, родительский узел x содержит достаточное количество ключей, чтобы выделить ключ для слияния. Это условие может быть нарушено, только в том случае, если два ребенка корня сливаются, так как поиск начинается с этого узла. По условиям B-дерева у корня должен быть как минимум один ключ, если дерево не пусто. При слиянии двух последних детей корня последний ключ перемещается во вновь возникшего единственного ребёнка, что приводит к пустому корневому узлу в не пустом дереве. В этом случае пустой узел корня удаляется и заменяется на единственного ребенка.