

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

**«Дальневосточный федеральный университет»**

**(ДВФУ)**

|  |
| --- |
| **ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ**  **Департамент математического и компьютерного моделирования** |

**ДОКЛАД**

**о практическом задании по дисциплине АИСД**

«Splay-деревья и AA-деревья»

направление подготовки 09.03.03 «Прикладная информатика»

профиль «Прикладная информатика в компьютерном дизайне»

|  |  |  |
| --- | --- | --- |
|  |  | Выполнил студент  гр. Б9121-09.03.03пикд  Курпас А.В  \_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)* |
| Доклад защищен:  С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  | Руководитель практики  Доцент ИМКТ А.С Кленин  *(должность, уч. звание)*  \_\_\_\_\_\_\_\_\_\_\_\_\_  *(подпись)*  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2022г. |
| Рег. № \_\_\_\_\_\_  «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2022 г. |  |  |

г. Владивосток

2022

[Введение 2](#_Toc116168876)

[Splay-деревья 3](#_Toc116168877)

[Splay-деревья как самобалансирующаяся структура 3](#_Toc116168878)

[Splaying 4](#_Toc116168879)

[Операции splay 7](#_Toc116168880)

[Основные операции 9](#_Toc116168881)

[AA-Деревья 13](#_Toc116168882)

[Структура 13](#_Toc116168883)

[Эффективность 19](#_Toc116168884)

# Введение

Одним их способом решения проблем с медленным доступом к данным является использование бинарных деревьев поиска – структуры для работы с упорядоченными множествами. Они предоставляют возможность выполнять некоторые из вышеперечисленных операций за время, меньшее, чем O(n), благодаря тому что имеют определённый набор ограничений.

Основной компонент бинарного дерева – узел с сравниваемым (англ. *comparable*) ключом. Ключом может быть как числовое значение, так и строка. В программном коде обращения к дереву производится через указатель на его корень (англ. *root*). Любой узел может иметь от 0 до 2 сыновей. Ключ левого сына (англ. *child*) всегда меньше, чем ключ предка (англ. *parent*); ключ правого сына всегда больше, чем ключ предка. Узлы без сыновей называются листьями (англ. *leaves*). Основные операции для работы с бинарным деревом:

* *Access(i)* – возвращает указатель на *i*-тый элемент, если он присутствует в множестве.
* *Insert(i) –* добавляет в множество *i-*тый элемент.
* *Delete(i)* – удаляет из множества *i*-тый элемент.

В данном докладе рассматриваются модификации бинарных деревьев – splay-деревья и AA-деревья.

# Splay-деревья

## Splay-деревья как самобалансирующаяся структура

Splay-дерево является бинарным деревом поиска, в котором поддерживается свойство сбалансированности. Оно принадлежит классу так называемых «саморегулирующихся деревьев», которые поддерживают необходимый баланс ветвления дерева, чтобы обеспечить выполнение операций поиска, добавления и удаления за логарифмическое время от числа хранимых элементов *O(*log*n)*. Это реализуется без использования каких-либо дополнительных полей в узлах дерева (как, например, в Красно-чёрных деревьях или АВЛ-деревьях, где в вершинах хранится, соответственно, цвет вершины или глубина поддерева).

Splay-деревья призваны уменьшить время выполнения операции для наихудшего случая, так как зачастую в некоторых реальных задачах (например, в базах данных) выполняется не одна, а последовательность операций. Будем считать, что важно общее время выполнения последовательности операций (далее – амортизированное время), а не время отдельных операций.

*Определение*. Средняя амортизационная стоимость операций — величина *a*, находящаяся по формуле: , где *t1, t2 … tn*— время выполнения операций *1, 2 … n*, совершённых над структурой данных. Под «амортизированным временем» мы подразумеваем среднее время операции при «наихудшей последовательности операций[[1]](#footnote-1)». Одним из способов получения амортизированной эффективности является использование «самобалансирующиеся» деревьев. Мы позволяем этой структуре данных находиться в произвольном состоянии, но во время каждой операции будем применять простое правило балансировки для повышения эффективности будущих операций.

Преимущества самобалансирующихся структур данных (по ср. с другими видами деревьев):

* Требуется меньше места в памяти, так как мы не храним информацию о высоте, цвете и балансе.
* Их принцип работы достаточно прост, и поэтому алгоритм реализуется быстрее.

В данной структуре данных применяется некая эвристика – мы договариваемся, что после каждого обращению к дереву применяется splaying, который «выталкивает» узел *x* в корень дерева, при этом выполняя последовательность поворотов. Предполагается, что *splay*-деревья так же эффективны, как любая форма динамически обновляемого бинарного дерева для достаточно длинной последовательности запросов (в амортизированном смысле).

Splay-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году.

## Splaying

На практике, когда реализуется структура splay-дерева, мы ссылаемся на дерево благодаря указателю на его корень (указатель на пустоту[[2]](#footnote-2) является пустым деревом).

Предположим, мы хотим выполнить несколько вышеперечисленных операций в дереве. Тогда, для того чтобы минимизировать общее время выполнения операций, мы должны убедиться в том, что наиболее частые по запросам элементы находятся наиболее близко к корню, ввиду того что, например, поиск осуществляется за *O(d),* где *d* – глубина элемента в дереве. Задача *splay*-дерева состоит в том, чтобы «перестроиться», тем самым подняв элемент *x* выше, чтобы получить доступ быстрее во время следующей операции. В качестве минимальной единицы «перестройки», которая выполняется за *O(1),* принимается вращение (англ. *rotation*), которое сохраняет основные свойства дерева.

*Single rotation.* После доступа к элементу *i* после *x*, поворачиваем ребро, соединяющее *x* со своим родителем (если *x* не является корнем).

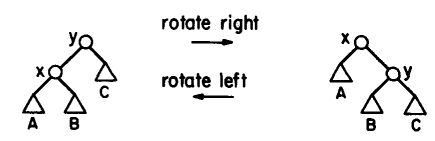


Рисунок . Вращение относительно x. Треугольники обозначают поддеревья.

*Move to root.* После доступа к элементу *i* после *x*, поворачиваем ребро, соединяющее *x* со своим родителем, пока *x* – не корень.

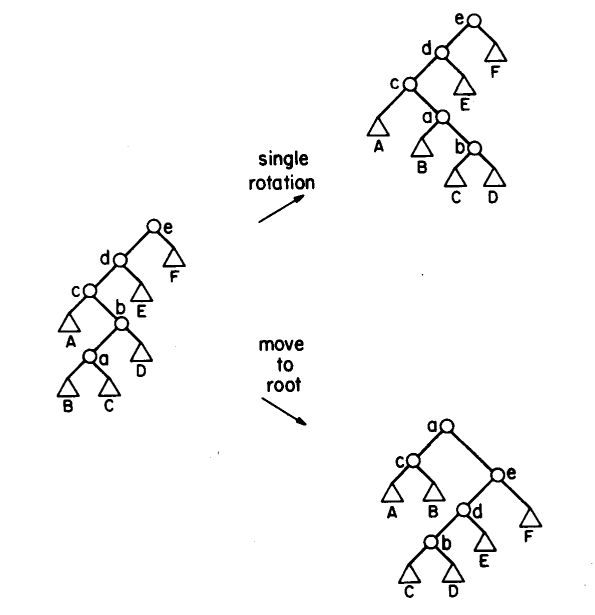
К сожалению, применение данных эвристик не может быть эффективным амортизированном смысле (англ. *amortized sense*), так как для них найдётся такая произвольно длинная последовательность операций, для которых общее время выполнения будет *O(n).*

Рисунок . Применение операций Single rotation и Move to root к узлу a.

Основная эвристика, используемая для перебалансировки дерева называется splaying – она очень похожа на последовательность операций move-to-root, так как она совершает подъём узла в корень. Отличие заключается в том, что это осуществляется при помощи чередующихся операций поворотов. Чтобы осуществить *splay* в узле *x*, мы будем повторять следующие операции, пока *x* не станет корнем.

*Обозначения. p* – родитель (англ. *parent*) узла *x*; *g* – прародитель (англ. *grandfather*)узла *x*, отец *p*.

## Операции splay

*Операция* 1 (zig). Если *p* – родитель *x*, то совершаем поворот относительно ребра, соединяющего *p* и *x*. (Совершается один раз и только в конце).

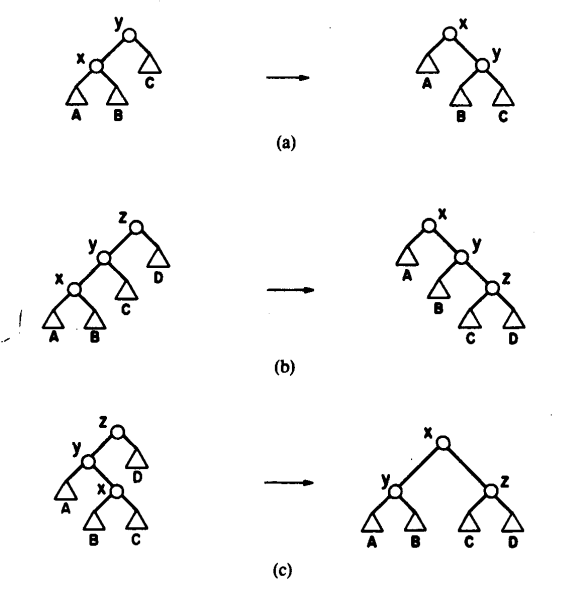
*Операция* 2 (zig-zig). Если *p* – не корень и родитель *x, p* – ребенок *g* и *x,* и *p* – только левые дети или только правые дети, то совершаем сначала поворот относительно ребра, соединяющего *p* и *g*, после этого совершаем поворот относительно ребра, соединяющего *x* и *p*.

Рисунок . Применение splay к узлу x. Каждый вариант поворота имеет свой зеркальный вариант. (a) Zig: окончательное одиночное вращение. (b) Zig-zig: два одиночный вращения. (c) Zig-zag: двойное вращение.

*Операция* 3 (zig-zag). Если *p* – не кореньи *x* - левый ребёнок *p*, а *p* –правый ребёнок *g* ИЛИ *x* - правый ребёнок *p*, а *p* – левый ребёнок *g*, то совершаем сначала поворот относительно ребра, соединяющего *x* и *p*, после этого совершаем поворот относительно ребра, соединяющего *x* и *g*.

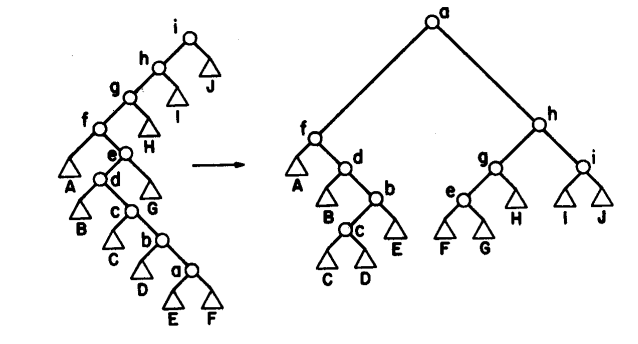
*Splay* в узле *x* на глубине *d* требует времени *O(d),* что пропорционально времени, затраченному на нахождение узла *x*. Операция *splay*, помимо перемещения узла x, на каждой итерации грубо вдвое уменьшает глубину каждого узла (см. Рисунок 4 и 5). Это преимущество не даёт применение более простых эвристик, таких как move-to-root, так как в них не используются такие **подходы как zig-zig и zig-zag.

Рисунок . Применение splay к узлу a.

## Основные операции

Рисунок . Наихудшие случаи для splay. (a) Использовано исключительно zig-zig. (b) Использовано исключительно zig-zag.

* *Access(i, t)* – найти элемент *i* и вернуть указатель на него или нулевой указатель – в противном случае.
* *Insert(i, t) –* вставить элемент *i*, если его ещё нет.
* *Delete(i, t)* – удалить элемент *i*, если он есть в дереве.
* *Join(t1, t2)* – объединить деревья *t1* и *t2* в одно, которое содержит все элементы из обоих деревьев. Операция предполагает, что все элементы из *t1* должны быть меньше, чем минимальный из *t2*. Возвращает указатель на новое дерево, удаляет *t1* и *t2*.
* *Split(i, t)* – вернуть два дерева *t1* и *t2*, где *t1* будет содержать элементы из t, которые больше или равны *i*, и *t2* будет содержать элементы, которые больше или равны *i*. Операция удаляет *t*.

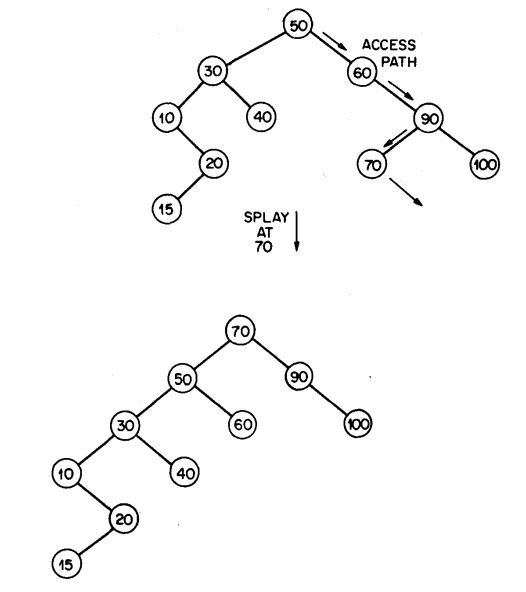
 Мы выполняем эти операции следующим образом. Чтобы выполнить *access(i, t)*, мы ищем от корня. Если поиск достигает узла *x*, содержащего *i*, мы завершаем операцию, начиная *splay(x)* и возвращая указатель на *x*. Если поиск достигает нулевого указателя (это означает, что данного элемента не существует), то мы завершаем операцию, вызывая *splay* от предыдущего узла. Если дерево пусто, мы возвращаем пустой указатель. (см. Рисунок 6.)

Рисунок . Попытка найти узел с ключом 80.

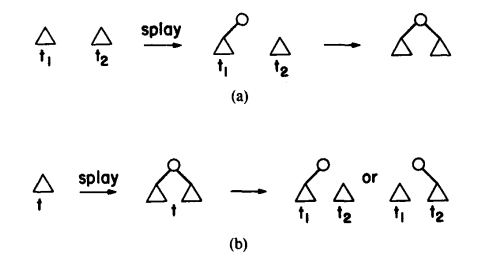
Для реализации функций *insert* и *delete* можно использовать *join* и *split*. Чтобы выполнить *join(t1, t2),* мы ищем самый большой элемент *i* в *t1*. Известно, что, элемент *i* будет иметь нулевой указатель в качестве правого сына. В качестве результата возвращаем новый корень *t*, с правым сыном *t2*. Чтобы выполнить *split(i, t)*, сначала производится *access(i),* а затем в качестве результата возвращаем два дерева, образованные левым и правым сыном *t*. В обеих операциях отдельно рассматривается случай с пустым указателем на месте дерева/поддерева. (см. Рисунок 7.)

Рисунок 7. Реализация *join* и *split*: (a) *join(t1, t2)*. (b) *split(i, t)*

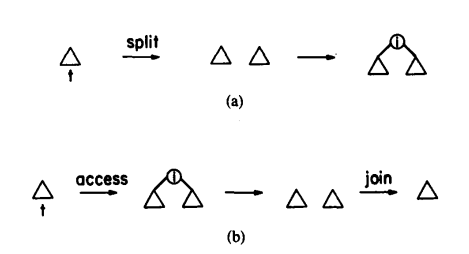
Чтобы выполнить *insert(i, t)*, мы используем *split(i, t)*, а затем заменяем t деревом, состоящим из нового корневого узла *i*, у которого левый и правый сын – деревья *t1* и *t2*. Чтобы выполнить *delete(i, t)*, мы используем *access(i, t)*, после чего заменяем *t* на *join(t1, t2),* где *t1* – левое поддерево *t*, а *t2* – правое поддерево *t*. (см. Рисунок 8.)

Рисунок . Реализация вставки и удаления с помощью *join* и *split*: (a) *insert(i, t)*. (b) *delete(i, t)*

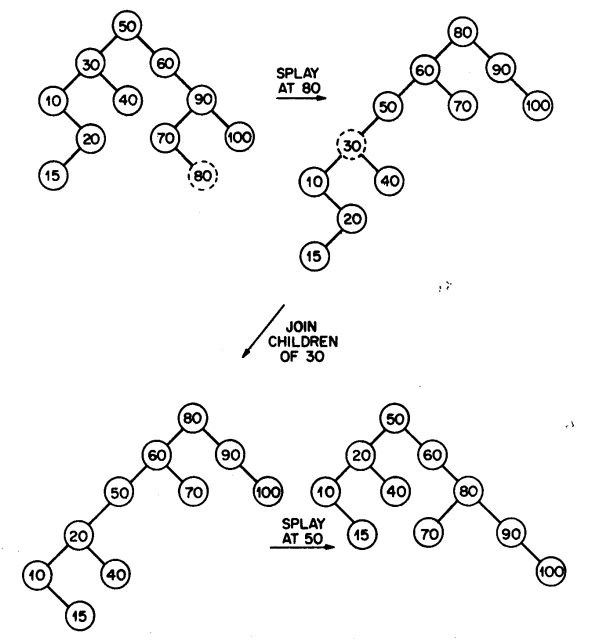
Существуют и альтернативные реализации вставки и удаления, которые имеют несколько лучшие временные характеристики. Чтобы выполнить *insert(i, t)*, мы производим поиск узла, содержащего *i*, и, если достигли пустого указателя, заменяем его новым узлом, содержащим *i*, после чего производим *splay(i)*. Чтобы выполнить *delete(i, t)*,мы ищем узел, содержащий *i*. Пусть этот узел будет *x*, а его родителем будет *y*. Мы заменяем *x* как дочерний элемент *y* результатом *join(x, y*),после чего производим *splay(i)*. (см. Рисунок 9.)

Рисунок . Альтернативные реализации вставки и удаления. После вставки 80 последовало удаление 30.

# AA-Деревья

АA-дерево (англ. AA-Tree) — структура данных, представляющая собой сбалансированное двоичное дерево поиска, которое является разновидностью красно-черного дерева с дополнительными ограничениями.

Наблюдение за другими структурами данных позволило понять, что можно избавиться от некоторых присущего им недостатка: большое количество рассматриваемых случаев во время балансировки можно заменить всего двумя операциями: *skew* и *split*.

АA-дерево названо по первым буквам имени и фамилии изобретателя, Арне Андерссона, который впервые предложил данную модификацию красно-черного дерева в 1993 году.

## Структура

*Определение.* Уровень вершины (англ. *Level*) — вертикальная высота соответствующей вершины.

*Свойства АА-дерева*:

1. Уровень каждого листа равен 1 (в различных реализациях может быть как 0, так и 1).
2. Уровень каждого левого ребенка ровно на один меньше, чем у его родителя.
3. Уровень каждого правого ребенка равен или на один меньше, чем у его родителя.
4. Уровень каждого правого внука строго меньше, чем у его прародителя.
5. Каждая вершина с уровнем больше 1 имеет двоих детей.

Во время работы бинарных деревьев их узлы могут принимать различного рода формы, которые должны быть обязательно рассмотрены для правильной балансировки. Это и есть причина, по которой они становятся осложнёнными. Например, добавление ребра к узлу или  может привести к пяти различным случаям , , ,  и . Таким образом, необходимо иметь большое количество операций для балансировки. AA-дерево решает эту проблему следующим образом: к одной вершине можно присоединить вершину только того же уровня, только одну и только справа (другими словами, красные вершины могут быть добавлены только в качестве правого ребенка).

*Определение*. Горизонтальное ребро (англ. *Horizontal edges*) — ребро, соединяющее вершины с одинаковым уровнем.

Для того, чтобы совершить правильную балансировку, необходимо ввести следующие операции:

*Skew(t)* – устранение левого горизонтального ребра (придерживаемся правила 2). Делаем правое вращение (см. Рисунок 1), чтобы заменить поддерево, содержащее левую горизонтальную связь, на поддерево, содержащее разрешенную правую горизонтальную связь.

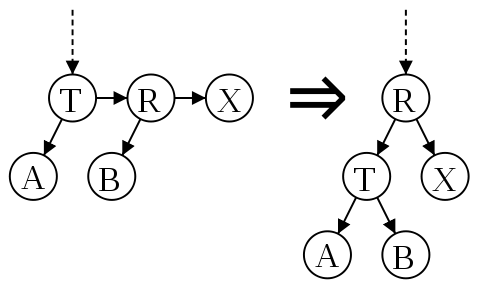
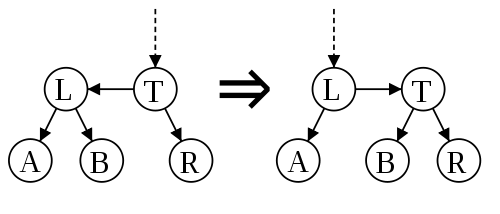
*Split(t)* – устранение двух последовательных правых горизонтальных ребер. Делаем левое вращение (см. Рисунок 1) и увеличиваем уровень, чтобы заменить поддерево, содержащее две или более последовательных правильных горизонтальных связи, на вершину, содержащую два поддерева с меньшим уровнем.

Рисунок 11. Операция split(t). Устранение двух последовательных правых горизонтальных ребёр.

Рисунок 10. Операция *skew(t)*. Устранение левой горизонтальной связи.

Данные операции достаточно быстро реализуются как простые функции. Помимо этого, они позволяют просто реализовать принцип работы алгоритмов вставки и удаления:

* *Insert(t)* – вставка нового узла с ключом, которого нет в множестве, производится на уровне 1; как и обычном бинарном дереве поиска. На каждой итерации подъёма к корню в узле *t* совершается следующая последовательность операций:
  + *Skew(t)*;
  + *Split(t)*;
* *Remove(t)* – удалить узел с уровня 1 (если t не является листом, то необходимо заменить на его «предшественника» или «преемника»). На каждой итерации подъёма в корню в узле t совершается следующая последовательность операций:
  + Обновляем уровни всех вершин: уровень *t* должен быть точно на 1 больше, чем у его сыновей. Если уровень правого ребёнка *t* больше, чем его уровень, то приравниваем уровень правого ребёнка к уровню *p*;
  + *Skew(t);*
  + *Split(t);*

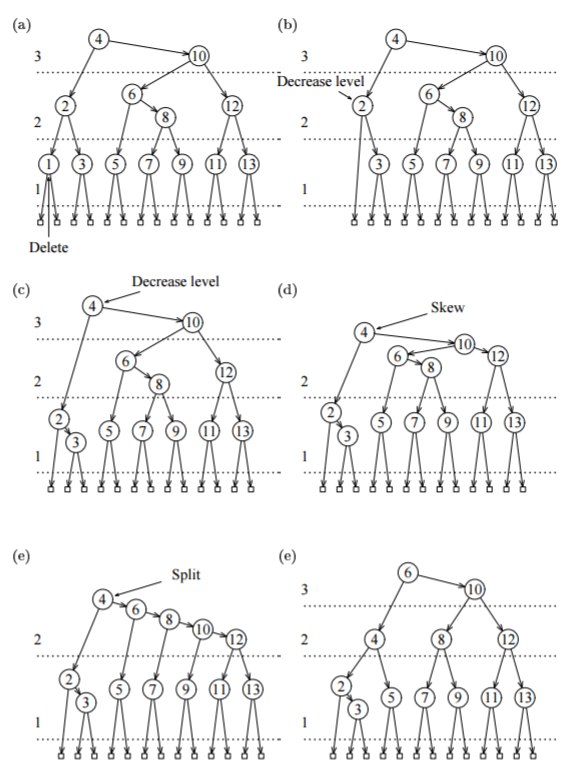


Рисунок 13. Удаление в AA-дереве. Уровни обозначены горизонтальным пунктиром.

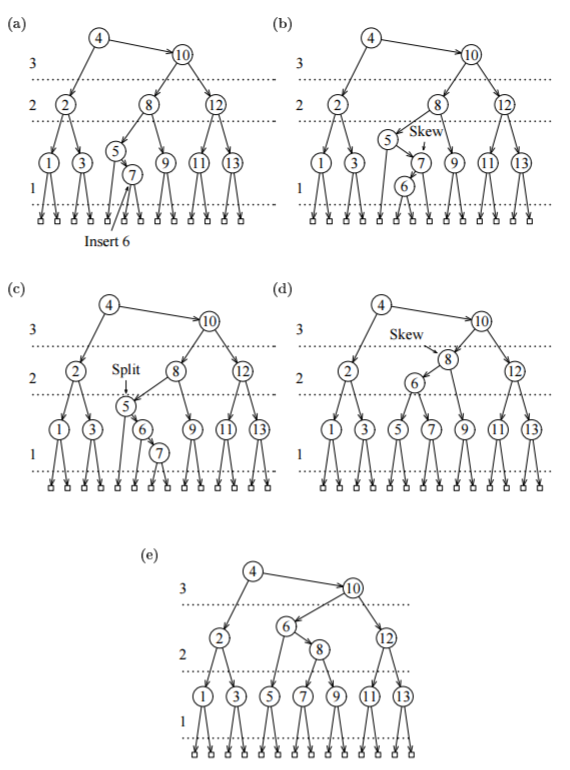


Рисунок 12. Вставка в AA-дереве. Уровни обозначены горизонтальным пунктиром.

## Эффективность

Оценка на высоту деревьев соответствует оценке для красно-черного дерева, 2⋅log 2(*N*), так как AA-дерево сохраняет структуру красно-черного дерева. Следовательно все операции происходят за *O*(log*N*), потому что в сбалансированном двоичном дереве поиска почти все операции реализуются за *O*(*h*), где *h* – максимальная глубина дерева. Скорость работы AA-дерева эквивалентна скорости работы красно-черного дерева, но так как в реализации вместо цвета обычно хранят «уровень» вершины, дополнительные расходы по памяти достигают байта.

1. Наихудшей последовательностью операций для бинарного дереваявляется вставка узлов с ключами в постоянно возрастающем порядке, а потом получение доступа к каждому, так как дерево выродится в линию. [↑](#footnote-ref-1)
2. В языке программирования C - NULL;

   В языке программирования C++ - nullptr; [↑](#footnote-ref-2)