ЛЕКЦИЯ №6

SPLAY-ДЕРЕВЬЯ

САМОПЕРЕСТРАИВАЮЩИЕСЯ ДЕРЕВЬЯ (SPLAY TREES)

Самоперестраивающееся дерево – 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.

Эти деревья не являются перманентно сбалансированными и на отдельных запросах могут работать даже линейное время. Однако, после каждого запроса они меняют свою структуру, что позволяет очень эффективно обрабатывать часто повторяющиеся запросы. Более того, амортизационная стоимость обработки одного запроса у них $O(logn)$, что делает splay-деревья хорошей альтернативой для перманентно сбалансированных собратьев.

Дереву не нужно хранить никакой дополнительной информации, что делает его эффективным с точки зрения использования памяти. После каждого обращения, даже поиска, splay-дерево меняет свою структуру.

Идея самоперестраивающихся деревьев основана на принципе пе-ремещения найденного узла в корень дерева. Эта операция назы-вается splay(*T, k*), где *k* – это ключ, а *T* – двоичное дерево поиска. После выполнения операции splay(*T, k*) двоичное дерево *T* пере-страивается, 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.**

Таким образом, 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.

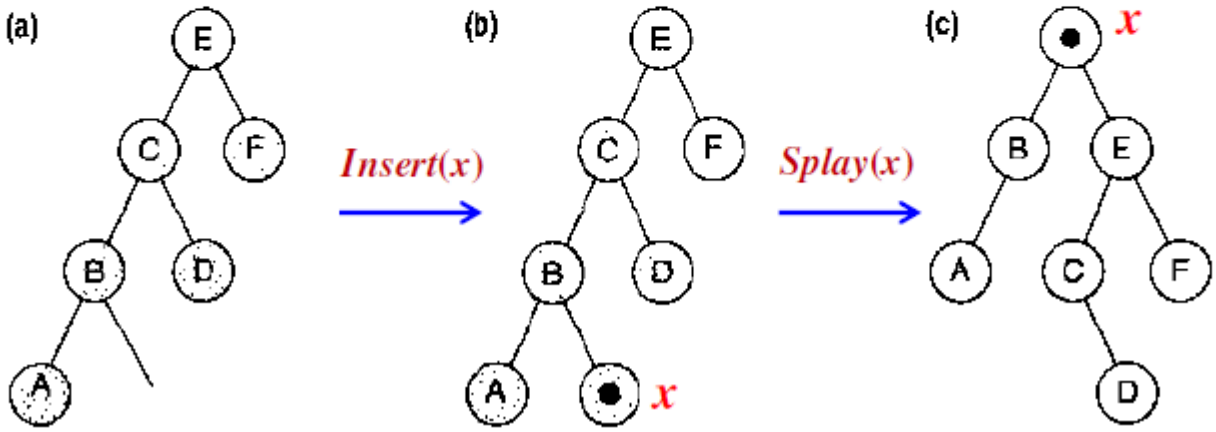
<u>Определение 6:</u> Словарными операциями над деревом называются базовые операции: поиск, вставка и удаление. Рассмотрим реализацию других словарных операций через splay.

ВСТАВКА УЗЛА В САМОПЕРЕСТРАИВАЮЩЕЕСЯ ДЕРЕВО

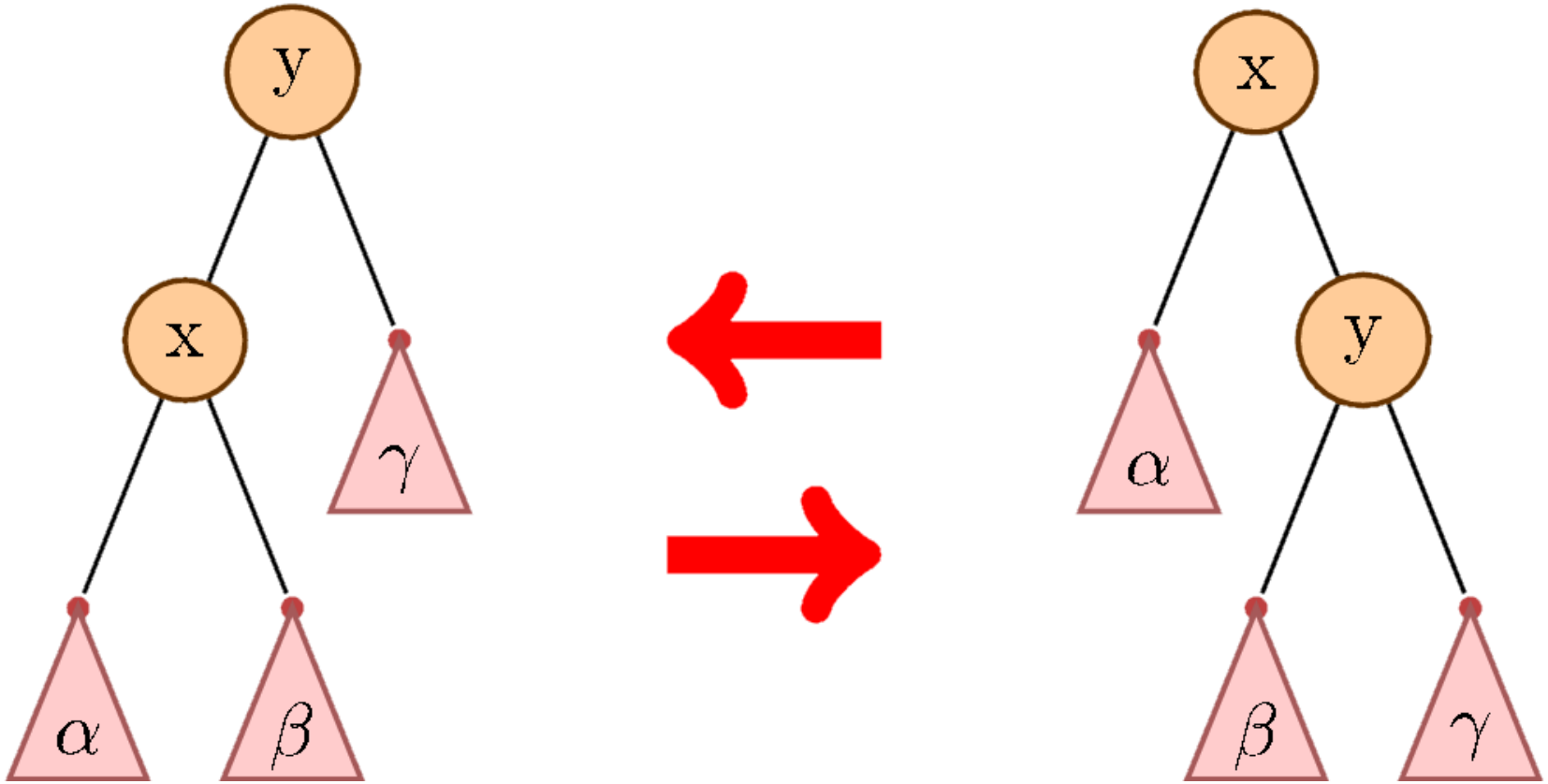
Алгоритм вставки узла в самоперестраивающееся дерево начинает свою работу с поиска узла в этом дереве с помощью описанной выше операции splay(*T*, *k*). Далее, проверяется значение ключа в корне. Если оно равно *k*, значит, узел найден. Если же оно не равно *k*, значит, в корне находится его предшественник или последова-тель. Тогда *k* становится новым корнем и, в зависимости от того, было ли до этого в корне значение, большее *k* или меньшее, ста-рый корень становится правым или, соответственно, левым сыном корня. Пример показан на рисунке:

Таким образом, поиск узла в самоперестраивающемся дереве фактически сводится к выполнению операции splay. Эвристика move-to-root (перемещение найденного узла в корень) основана на предположении, что если тот же самый элемент потребуется в ближайшее время, он будет найден быстрее.

1) Спускаемся по дереву, отыскиваем лист для вставки элемента x (как в традиционных BST) и создаем его
2) Применяем к узлу x процедуру Splay, которая поднимает его в корень дерева



Подъем реализуется через повороты вершин. За один поворот, можно поменять местами родителя с ребенком, как показано на рисунке ниже.



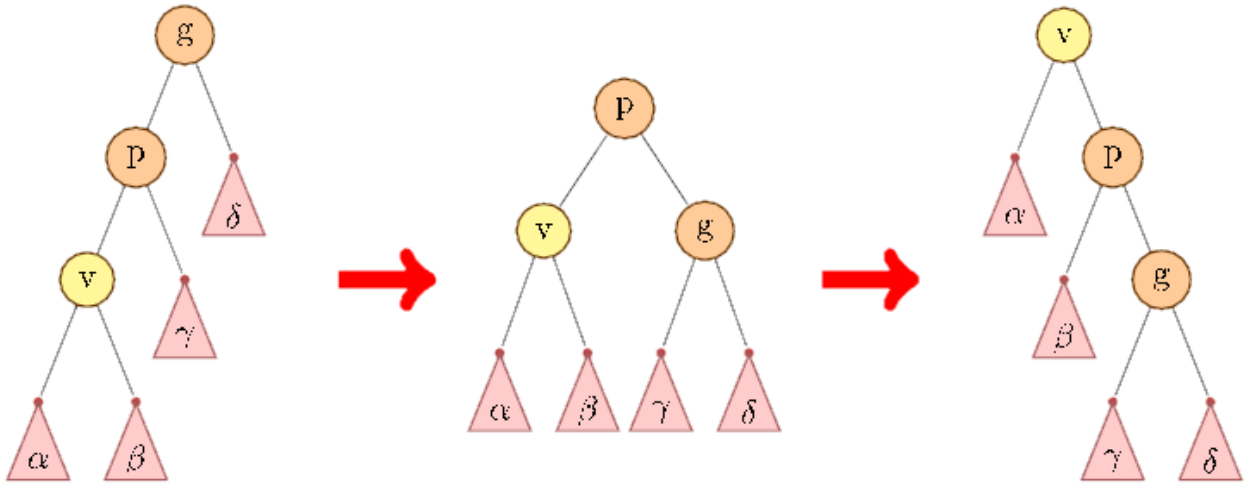
```
        def rotate(parent, child):
gparent = parent.parent
if gparent != None:
  if gparent.left == parent:
    gparent.left = child
  else:
    gparent.right = child
if parent.left == child:
  parent.left, child.right = child.right, parent
else:
  parent.right, child.left = child.left, parent

keep_parent(child)
keep_parent(parent)
child.parent = gparent.
```

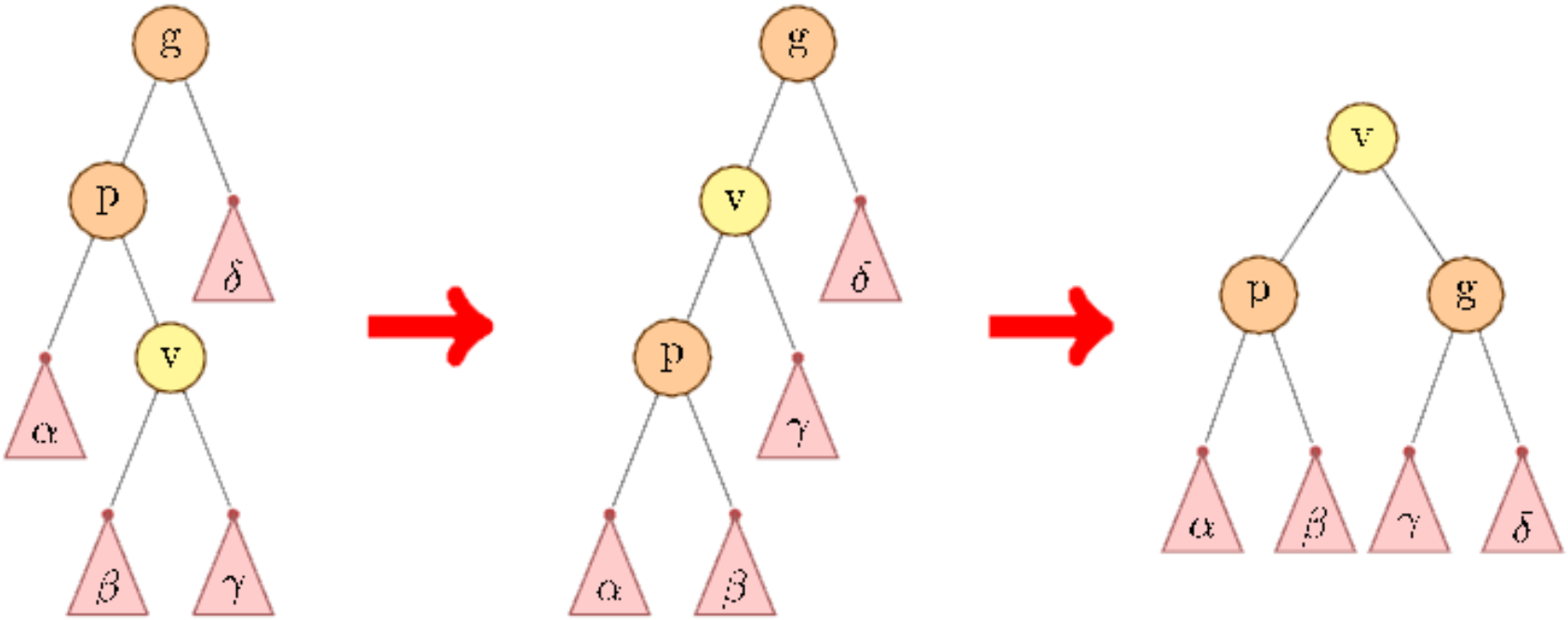
Но просто поворачивать вершину, пока она не станет корнем, недостаточно. Хитрость splay-дерева в том, что при продвижении вершины вверх, расстояние до корня сокращается не только для поднимаемой вершины, но и для всех ее потомков в текущих поддеревьях. Для этого используется техника zig-zig и zig-zag поворотов.

Основная идея zig-zig и zig-zag поворотов, рассмотреть путь от дедушки к ребенку. Если путь идет только по левым детям или только по правым, то такая ситуация называется zig-zig. Как ее обрабатывать показано на рисунке ниже. Сначала повернуть родителя, потом ребенка.

zig-zig

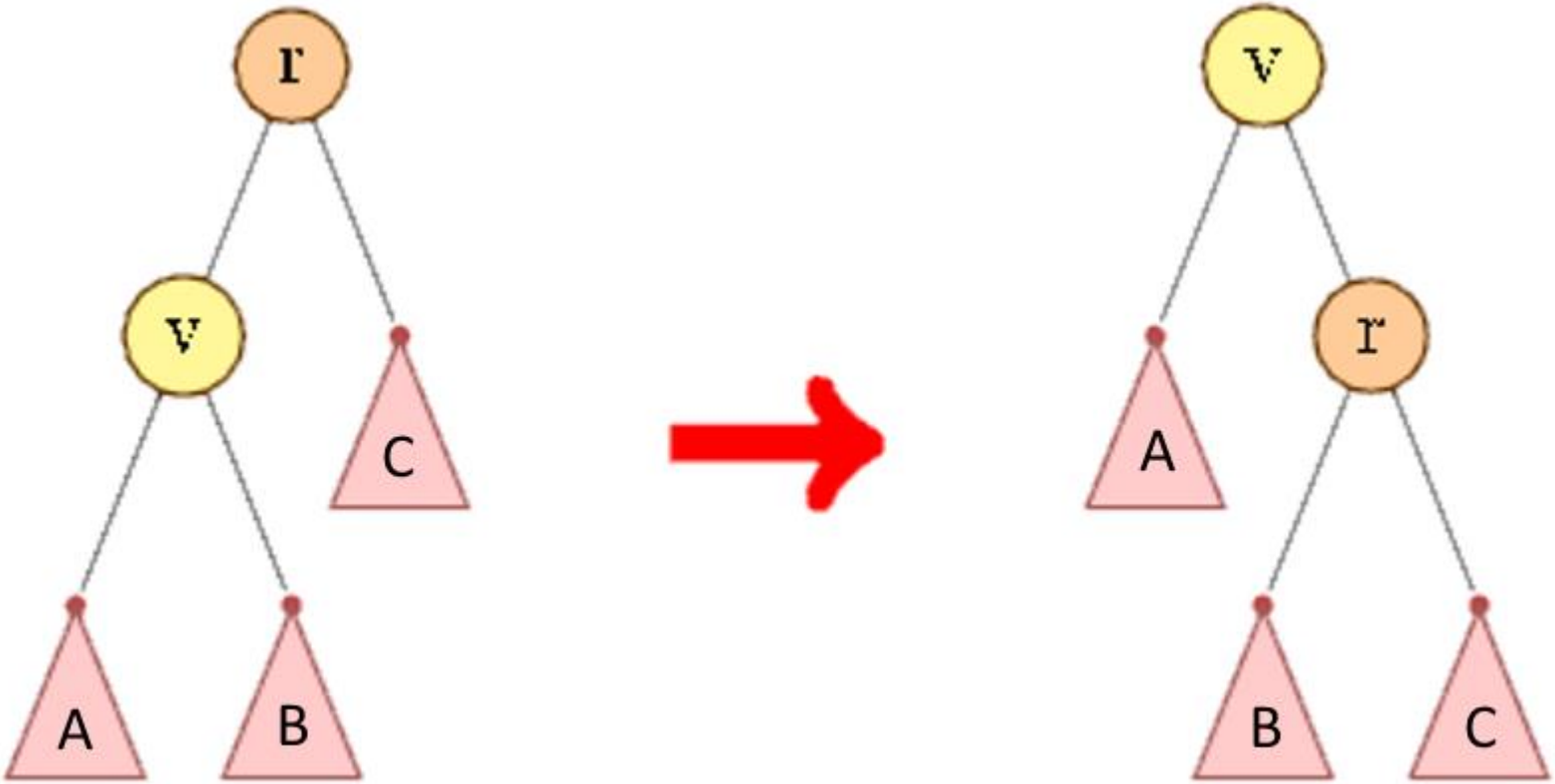


В противном случае, мы сначала меняем ребенка с текущим родителем, потом с новым.



zig-zag

Если у вершины дедушки нет, делаем обычный поворот:



Описанная выше процедура поднятия вершины с помощью zig-zig и zig-zag поворотов является ключевой для splay-дерева.

Алгоритм:

Для описания структуры можно использовать класс, описывающий отдельную вершину:

```
class Node:
  def __init__(self, key, left = None, right = None, parent = None):
    self.left   = left
    self.right  = right
    self.parent = parent
    self.key    = key
```

вспомогательные процедуры для работы с указателями на родителей.

```
def set_parent(child, parent):
  if child != None:
    child.parent = parent

def keep_parent(v):
  set_parent(v.left, v)
  set_parent(v.right, v)
```

Процедура splay, описанная ранее:

```
def splay(v):
  if v.parent == None:
    return v
  parent = v.parent
  gparent = parent.parent
  if gparent == None:
    rotate(parent, v)
    return v
```

Процедура splay, описанная ранее (ПРОДОЛЖЕНИЕ):

```
else:
    zigzig = (gparent.left == parent) == (parent.left == v)
    if zigzig:
      rotate(gparent, parent)
      rotate(parent, v)
    else:
      rotate(parent, v)
      rotate(gparent, v)
    return splay(v)
```

Процедура поиска в splay-дереве отличается от обычной только на последней стадии: после того, как вершина найдена, она перемещается вверх и становится корнем с использованием splay.

```
def find(v, key):
  if v == None:
    return None
  if key == v.key:
    return splay(v)
  if key < v.key and v.left != None:
    return find(v.left, key)
  if key > v.key and v.right != None:
    return find(v.right, key)
  return splay(v)
```

Чтобы реализовать вставку и удаление ключа, необходимо использовать две процедуры: split и merge (разделить и объединить).

Процедура split получает на вход ключ key и делит дерево на два. В одном дереве все значения меньше ключа key, а в другом — больше. Реализуется она просто. Нужно через find найти ближайшую к ключу вершину, поднять ее вверх и потом отделить либо левое, либо правое поддерево (а возможно и оба).

```
        def split(root, key):
  if root == None:
    return None, None
  root = find(root, key)
  if root.key == key:
    set_parent(root.left, None)
    set_parent(root.right, None)
    return root.left, root.right
  if root.key < key:
    right, root.right = root.right,
None
    set_parent(right, None)
    return root, right
else:
    left, root.left = root.left, None
    set_parent(left, None)
    return left, root
```

Чтобы вставить новый ключ, достаточно разделить дерево в месте нового ключа, а затем сделать новую вершину-корень, у которой поддеревьями будет результат операции split:

```
def insert(root, key):
left, right = split(root, key)
root = Node(key, left, right)
keep_parent(root)
return root
```

Процедура merge использует левое left и правое right деревья, причем ключи дерева left должны быть меньше ключей дерева right. Вершина правого дерева right с минимальным ключом поднимается наверх. Затем дерево left присоединяется как левое поддерево:

```
def merge(left, right):
if right == None:
  return left
if left == None:
  return right
right = find(right, left.key)
right.left, left.parent = left, right
return right
```

Для того, чтобы удалить вершину, нужно переместить ее вверх, а потом слить ее левые и правые поддеревья

```
def remove(root, key):
root = find(root, key)
set_parent(root.left, None)
set_parent(root.right, None)
return merge(root.left, root.right)
```

Чтобы splay-дерево поддерживало повторяющиеся ключи, можно поступить двумя способами. Нужно либо каждому ключу сопоставить список, хранящий нужную доп. информацию, либо реализовать процедуру find так, чтобы она возвращала первую в порядке обхода LUR вершину с ключом, большим либо равным заданного.

Анализ

Заметим, что процедуры удаления, вставки, слияния и разделения деревьев работают за + время работы процедуры find.

Процедура find работает пропорционально глубине искомой вершины в дереве. По завершении поиска запускается процедура splay, которая тоже работает пропорционально глубине вершины. Таким образом, достаточно оценить время работы процедуры splay.

Анализ

Заметим, что процедуры удаления, вставки, слияния и разделения деревьев работают за + время работы процедуры find.

Процедура find работает пропорционально глубине искомой вершины в дереве. По завершении поиска запускается процедура splay, которая тоже работает пропорционально глубине вершины. Таким образом, достаточно оценить время работы процедуры splay.

Анализ

Заметим, что процедуры удаления, вставки, слияния и разделения деревьев работают за + время работы процедуры find.

Процедура find работает пропорционально глубине искомой вершины в дереве. По завершении поиска запускается процедура splay, которая тоже работает пропорционально глубине вершины. Таким образом, достаточно оценить время работы процедуры splay.

Во-первых, заметим, что все вышеперечисленные базовые операции (поиск, вставка, удаление) требуют выполнения O(1) операций splay и O(1) дополнительного времени. Действительно, при поиске и вставке требуется одна операция splay, а при удалении – две. Дополнительные действия – просмотр корня, удаление вершины и т.п. занимают фиксированное время, которое не зависит от высоты дерева. Сама операция splay занимает O(n) времени, где n – количество узлов в дереве. Это происходит тогда, когда дерево совсем не сбалансировано. Но, в целом, эти деревья за счет того, что они самоперестраивающиеся, имеют тенденцию к сбалансированности.

Введем следующие понятия.

Определение 7: *Весом узла x называется количество узлов в под-дереве T с корнем в x, включая сам узел:* $|T(x)|$.

Определение 8: *Рангом узла называется двоичный логарифм его веса:* $r(x)=\log_2|T(x)|$

Проведем усредненную оценку сложности операций с помощью так называемого *метода бухгалтерского учета*. Представим, что каждая операция с деревом стоит фиксированную сумму денег за единицу времени. В самом начале каждый узел содержит кредит – $r(x)$ рублей, т.е. сумму, равную его рангу, которая может частично или полностью использоваться для оплаты операций. Также можно инвестировать дополнительную сумму для оплаты операций, помимо кредита. Если после серии операций над деревом и перед началом новых операций суммарный кредит (сумма рангов всех вершин) будет не меньше, чем до них, то будем говорить о *сохранении денежного инварианта*.

Операция splay(T,x) требует инвестирования не более чем рублей с сохранением денежного инварианта. $3\log_2 n+1$ рублей.

Для выполнения поворота и сохранения денежного инварианта требуется не более

$3(r'(x)- r(x))+1$ рублей

Любая последовательность из m словарных операций на самоперестраивающемся дереве, которое было изначально пусто и на каждом шаге содержало не более n узлов, занимает не более $O(m \log n)$ времени

Тогда можно проводить эту серию операций любое количество раз. Докажем следующую лемму.

Лемма 2: Операция splay(*T*, *x*) требует инвестирования не более чем рублей с сохранением денежного инварианта. $3\log_2(n+1)$

Доказательство:

Сложность будет оцениваться по количеству поворотов. Обозначим через *r*(*x*) и *r'*(*x*) значения ранга узла *x* до и после поворота (1-го типа – одинарного, 2-го типа – серии из двух одинарных или 3-го типа – двойного).