

Алгоритмы и алгоритмические языки

Лекция 23

4 декабря 2019 г.

Красно-черные деревья

- ♦ Красно-черное дерево – двоичное дерево поиска, каждая вершина которого окрашена либо в красный, либо в черный цвет
- ♦ Поля – цвет, дети, родители

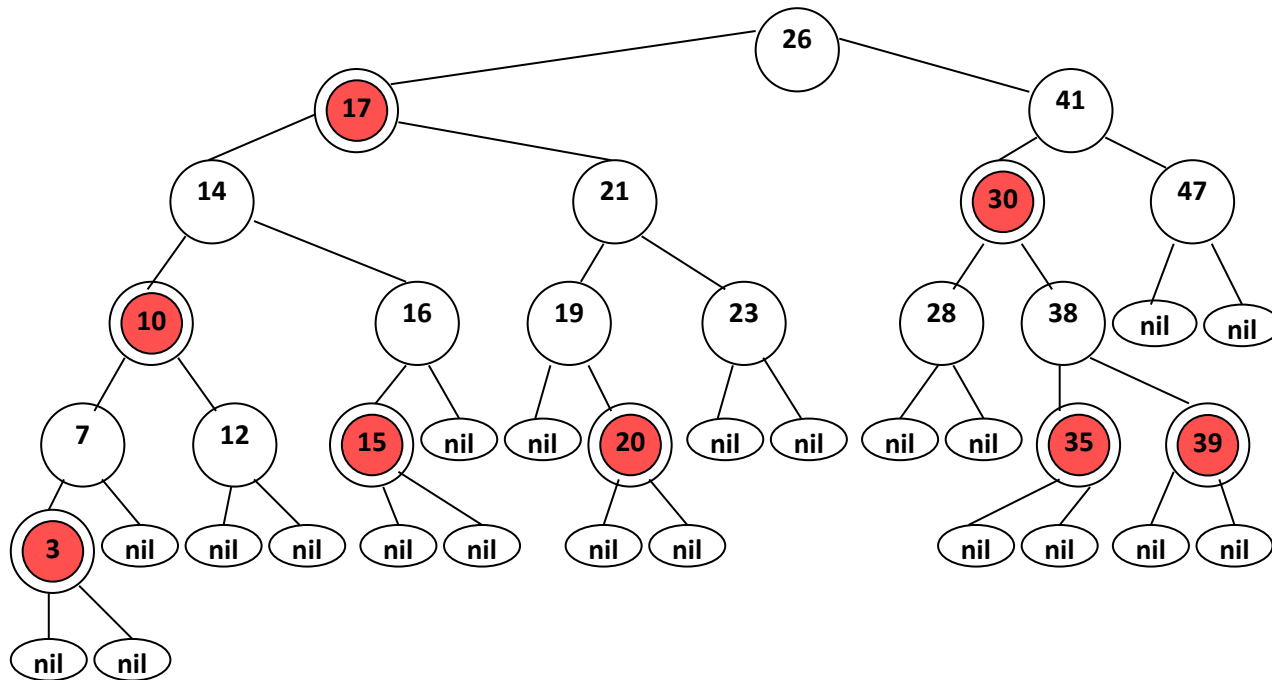
```
typedef struct rbtree {  
    int key;  
    char color;  
    struct rbtree *left, *right, *parent;  
} rbtree, *prbtree;
```
- ♦ Будем считать, что если **left** или **right** равны **NULL**, то это “указатели” на фиктивные листы, т.е. все вершины внутренние

Красно-черные деревья



Свойства красно-черных деревьев:

1. Каждая вершина либо красная, либо черная.
2. Каждый лист (фиктивный) – черный.
3. Если вершина красная, то оба ее сына – черные.
4. Все пути, идущие от корня к любому листу, содержат одинаковое количество черных вершин



Красно-черные деревья

- ♦ Обозначим $bh(x)$ – "черную" высоту поддерева с корнем x (саму вершину в число не включаем), т.е. количество черных вершин от x до листа
- ♦ Черная высота дерева – черная высота его корня
- ♦ *Лемма:* Красно-черное дерево с n внутренними вершинами (без фиктивных листьев) имеет высоту не более $2\log_2(n+1)$.
 - (1) Покажем вначале, что поддерево x содержит не меньше $2^{bh(x)} - 1$ внутренних вершин
 - (1а) Индукция. Для листьев $bh = 0$, т.е. $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
 - (1б) Пусть теперь x – не лист и имеет черную высоту k . Тогда каждый сын x имеет черную высоту не меньше $k - 1$ (красный сын имеет высоту k , черный – $k - 1$).
 - (1в) По предположению индукции каждый сын имеет не меньше $2^{k-1} - 1$ вершин. Поэтому поддерево x имеет не меньше $2^{k-1} - 1 + 2^{k-1} - 1 + 1 = 2^k - 1$.

Красно-черные деревья

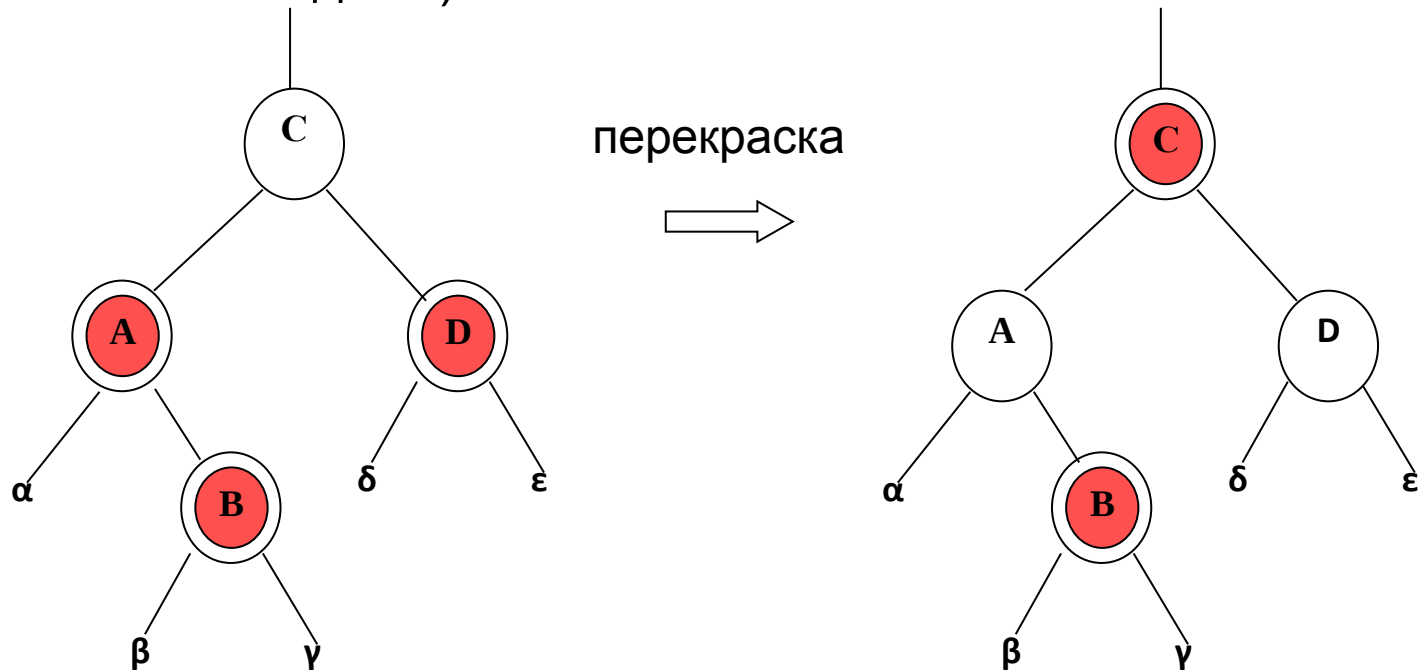
- ◆ Лемма: Красно-черное дерево с n внутренними вершинами (без фиктивных листьев) имеет высоту не более $2\log_2(n+1)$.
 - (2) Теперь пусть высота дерева равна h .
 - (2а) По свойству 3 черные вершины составляют не меньше половины всех вершин на пути от корня к листу. Поэтому черная высота дерева bh не меньше $h/2$.
 - (2б) Тогда $n \geq 2^{h/2} - 1$ и $h \leq 2\log_2(n + 1)$. Лемма доказана.
- ◆ Следовательно, поиск по красно-черному дереву имеет сложность $O(\log_2 n)$.

Красно-черные деревья: вставка вершины

- ◇ Сначала мы используем обычную процедуру занесения новой вершины в двоичное дерево поиска:
 - ◆ красим новую вершину в красный цвет.
- ◇ Если дерево было пустым, то красим новый корень в черный цвет
- ◇ Свойство 4 при вставке изначально не нарушено, т.к. новая вершина красная
- ◇ Если родитель новой вершины черный (новая – красная), то свойство 3 также не нарушено
- ◇ Иначе (родитель красный) свойство 3 нарушено

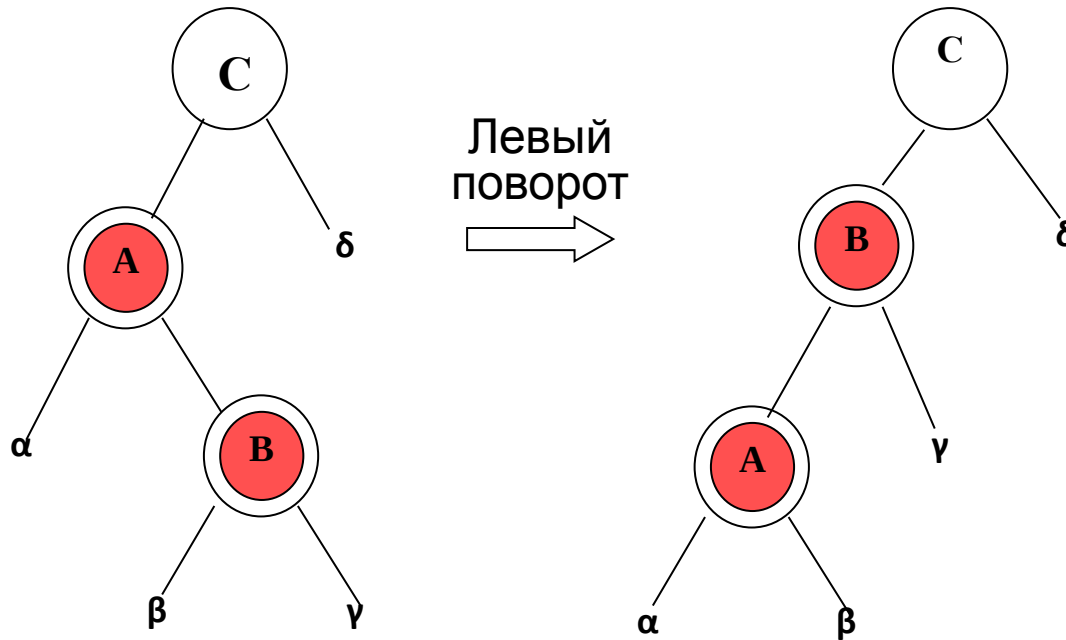
Красно-черные деревья: вставка вершины

- ◆ Случай 1: “дядя” (второй сын родителя родителя текущей вершины) тоже красный (как текущая вершина и родитель)
 - ◆ Возможно выполнить перекраску: родителя и дядю (вершины A и D) – в черный цвет, деда – (вершина C) – в красный цвет
 - ◆ Свойство 4 не нарушено (черные высоты поддеревьев совпадают)



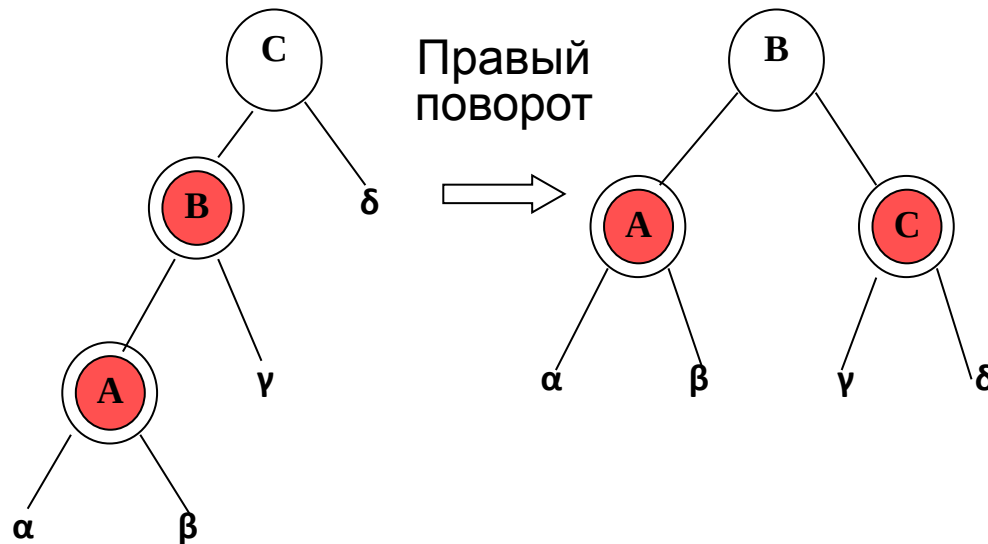
Красно-черные деревья: вставка вершины

- ◆ Случай 2: “дядя” (второй сын родителя родителя текущей вершины) черный
 - ◆ Шаг 1: Необходимо выполнить левый поворот родителя текущей вершины (вершины A)



Красно-черные деревья: вставка вершины

- ◆ Случай 2: “дядя” (второй сын родителя родителя текущей вершины) черный
 - ◆ Шаг 2: Необходимо выполнить правый поворот вершины С, после чего ...
 - ◆ Шаг 3: ... перекрасить вершины В и С
 - ◆ Все поддеревья имеют черные корни и одинаковую черную высоту, поэтому свойства 3 и 4 верны



Самоперестраивающиеся деревья (*splay trees*)

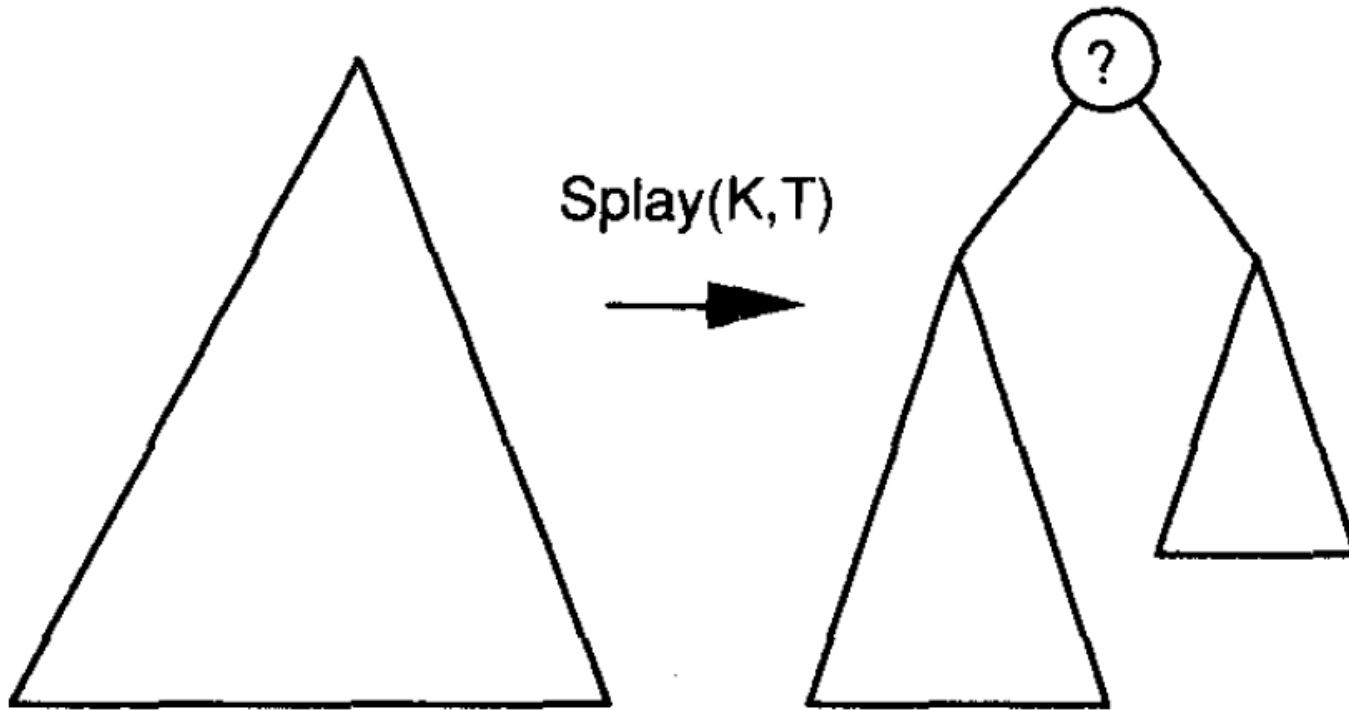
- ◇ Двоичное дерево поиска, не содержащее дополнительных служебных полей в структуре данных (нет баланса, цвета и т.п.)
- ◇ Гарантируется не логарифмическая сложность в худшем случае, а *амортизированная* логарифмическая сложность:
 - ◆ Любая последовательность из m словарных операций (поиска, вставки, удаления) над n элементами, *начиная с пустого дерева*, имеет сложность $O(m \log n)$
 - ◆ Средняя сложность одной операции $O(\log n)$
 - ◆ Некоторые операции могут иметь сложность $\Theta(n)$
 - ◆ Не делается предположений о распределении вероятностей ключей дерева и словарных операций (т.е. что некоторые операции выполнялись чаще других)
- ◇ Хорошее описание в:
Harry R. Lewis, Larry Denenberg. Data Structures and Their Algorithms. HarperCollins, 1991. Глава 7.3.
<http://www.amazon.com/Structures-Their-Algorithms-Harry-Lewis/dp/067339736X>

Самоперестраивающиеся деревья (*splay trees*)

- ◇ Идея: эвристика Move-to-Front
 - ◆ Список: давайте при поиске элемента в списке перемещать найденный элемент в начало списка
 - ◆ Если он потребуется снова в обозримом будущем, он найдется быстрее
- ◇ Move-to-Front для двоичного дерева поиска: операция $Splay(K, T)$ (подравнивание, перемешивание, расширение)
 - ◆ После выполнения операции $Splay$ дерево T перестраивается (оставаясь деревом поиска) так, что:
 - ◆ Если ключ K есть в дереве, то он становится корнем
 - ◆ Если ключа K нет в дереве, то в корне оказывается его предшественник или последователь в симметричном порядке обхода

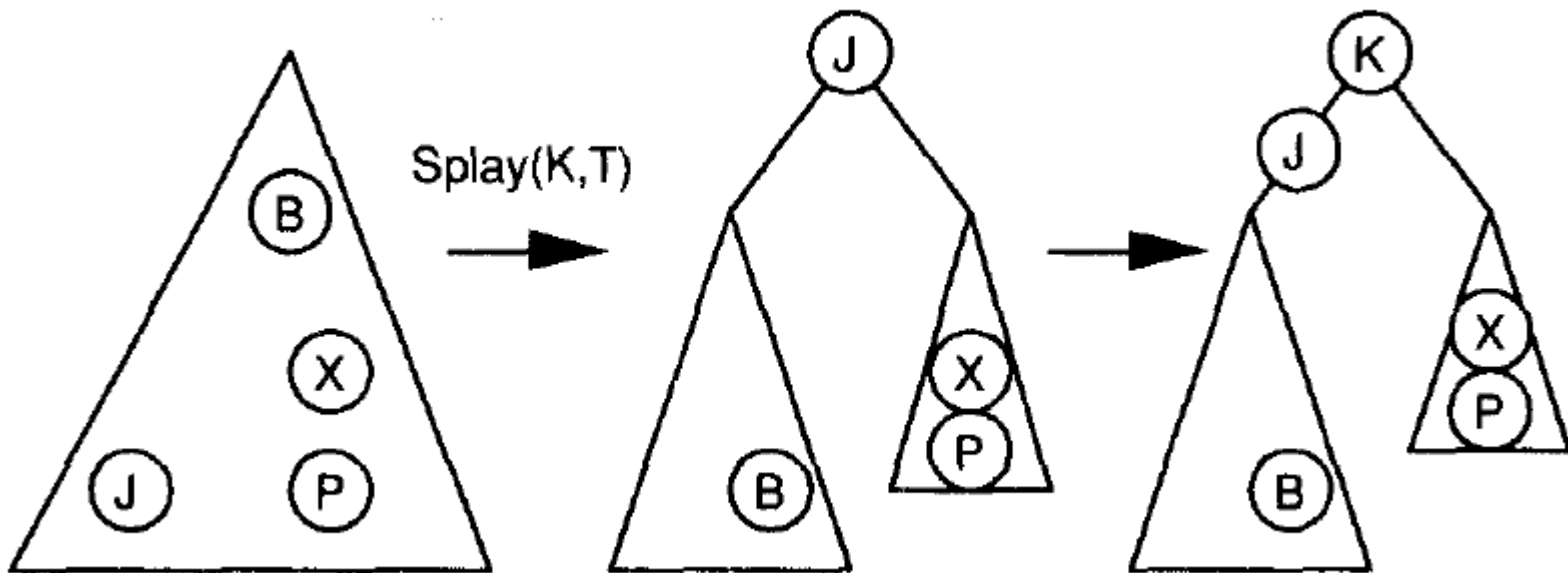
Реализация словарных операций через *splay*

- ◇ Поиск (LookUp): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
 - ◆ если значение равно K , то ключ найден



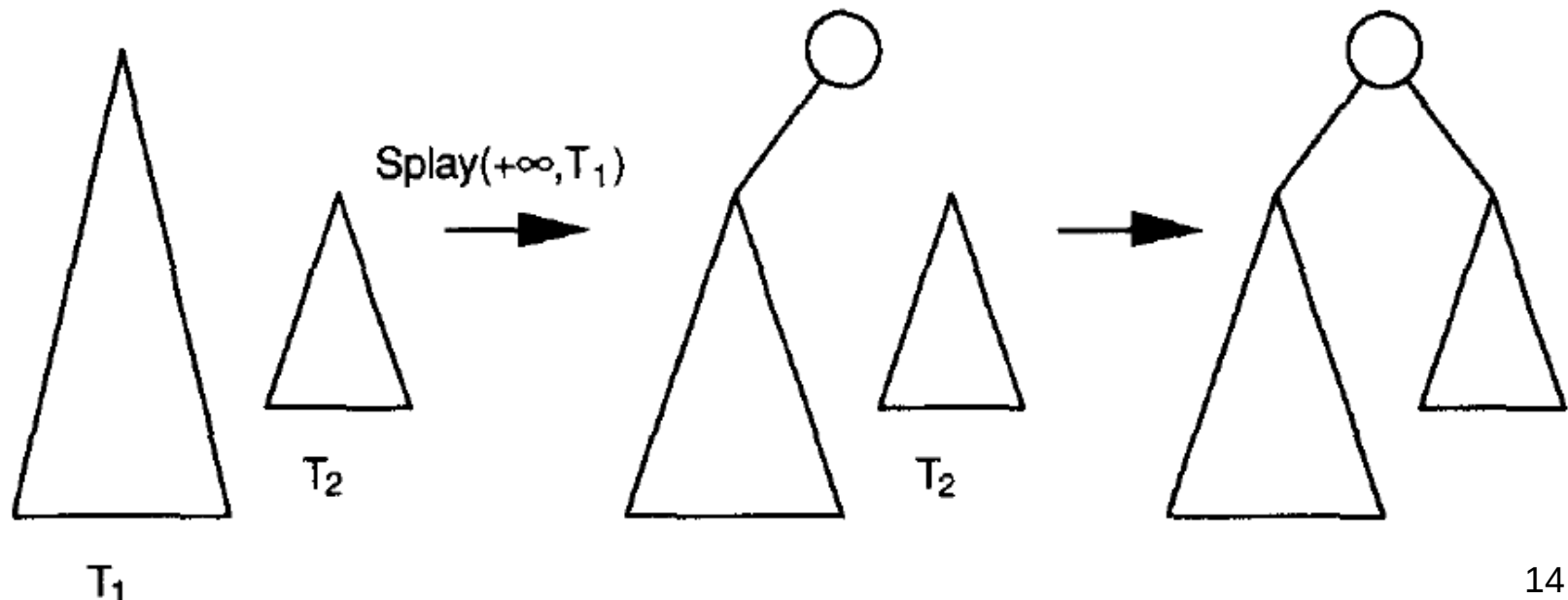
Реализация словарных операций через *splay*

- ◇ Вставка (Insert): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
 - ◆ если значение уже равно K , то обновим данные ключа
 - ◆ если значение другое, то вставим новый корень K и поместим старый корень J слева или справа (в зависимости от значения J)



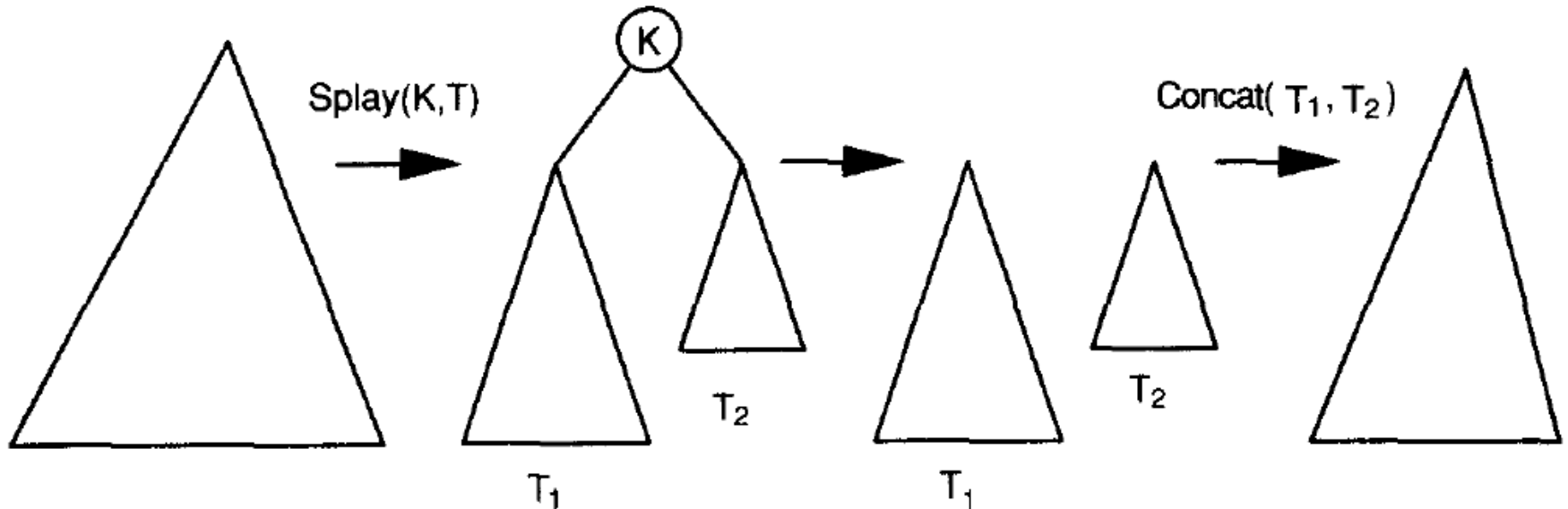
Реализация словарных операций через *splay*

- ❖ Операция *Concat* (T_1, T_2) – слияние деревьев поиска T_1 и T_2 таких, что **все** ключи в дереве T_1 **меньше**, чем **все** ключи в дереве T_2 , в одно дерево поиска
- ❖ Слияние (*Concat*): выполним операцию *Splay*($+\infty, T_1$) со значением ключа, заведомо больше любого другого в T_1
 - ◆ После *Splay*($+\infty, T_1$) у корня дерева T_1 нет правого сына
 - ◆ Присоединим дерево T_2 как правый сын корня T_1



Реализация словарных операций через *splay*

- ◆ Удаление (Delete): выполним операцию *Splay*(K, T) и проверим значение ключа в корне:
 - ◆ если значение **не равно** K , то ключа в дереве нет и удалять нам нечего
 - ◆ иначе (ключ был найден) выполним операцию *Concat* над левым и правым сыновьями корня, а корень удалим

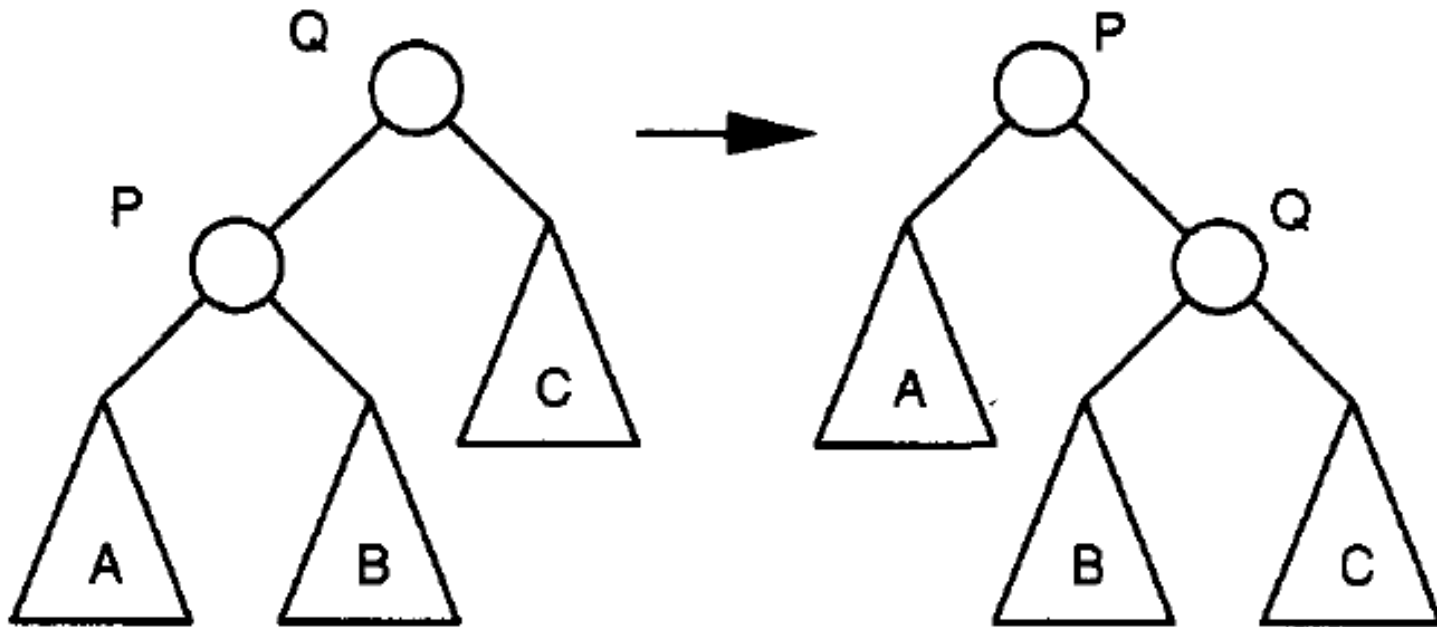


Реализация операции splay

- ◆ Шаг 1: ищем ключ K в дереве обычным способом, запоминая пройденный путь по дереву
 - ◆ Может потребоваться память, линейная от количества узлов дерева
 - ◆ Для уменьшения количества памяти можно воспользоваться *инверсией ссылок* (link inversion)
 - перенаправление указателей на сына назад на родителя вдоль пути по дереву плюс 1 бит на обозначение направления
- ◆ Шаг 2: получаем указатель P на узел дерева либо с ключом K , либо с его соседом в симметричном порядке обхода, на котором закончился поиск (сосед имеет единственного сына)
- ◆ Шаг 3: возвращаемся назад вдоль запомненного пути, перемещая узел P к корню (узел P будет новым корнем)

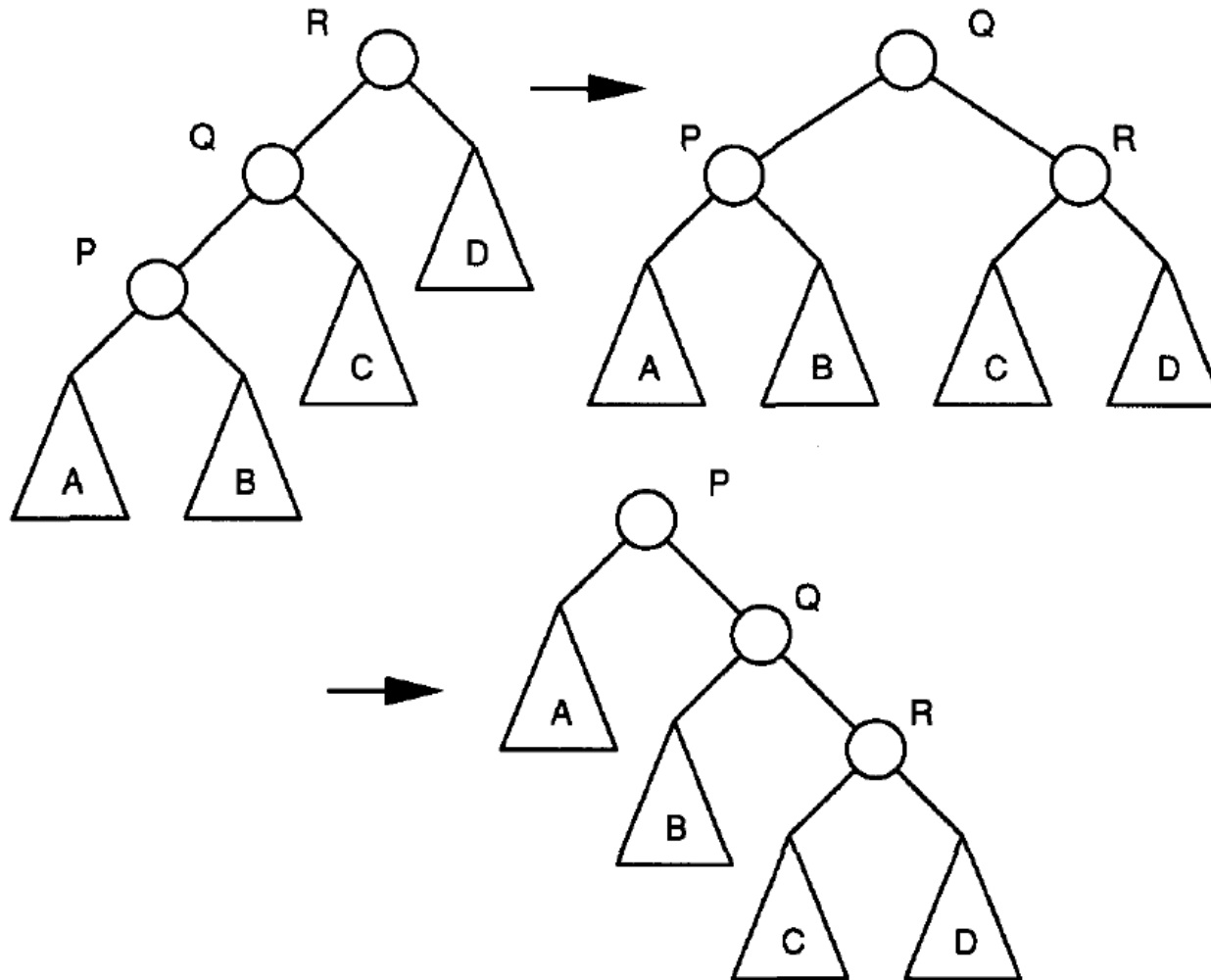
Реализация операции *splay*

- Шаг 3а): отец узла P – корень дерева (или у P нет деда)
 - выполняем однократный поворот налево или направо



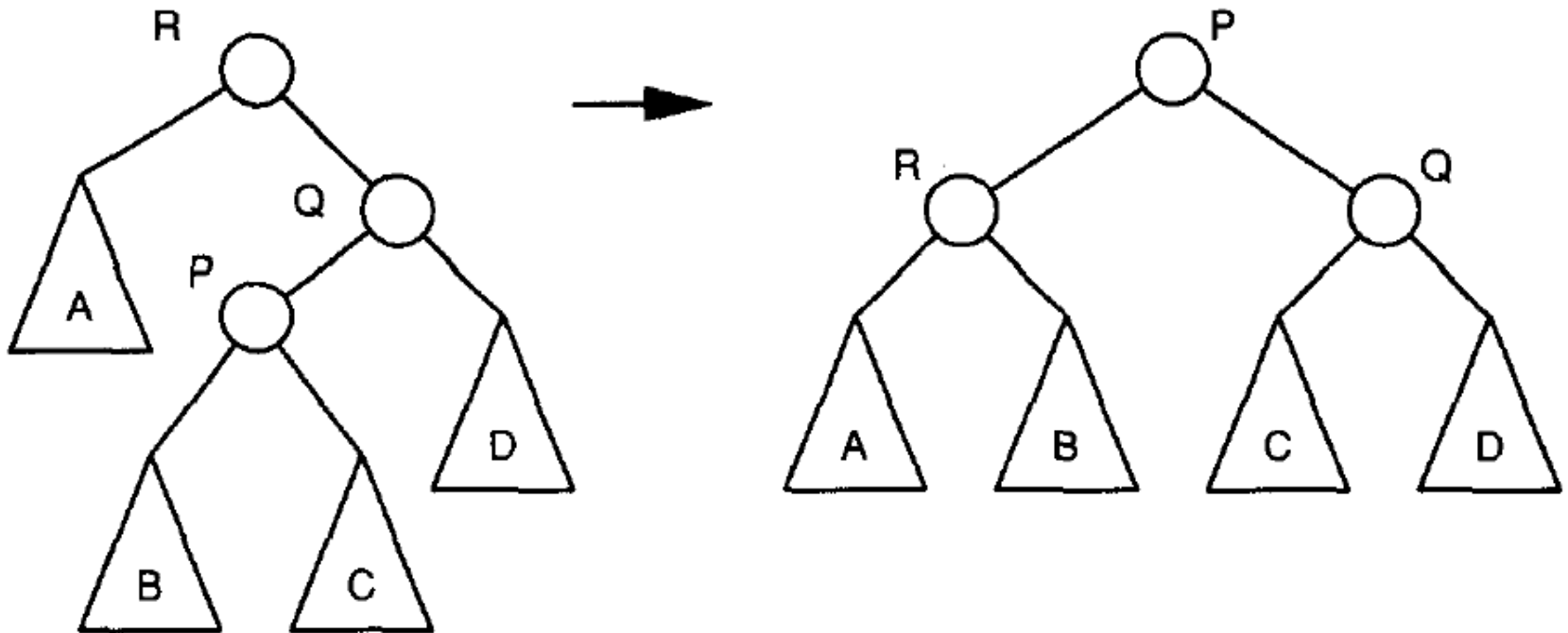
Реализация операции *splay*

- Шаг 3б): узел P и отец узла P – оба левые или правые дети
 - выполняем два однократных поворота направо (налево), сначала вокруг деда P , потом вокруг отца P



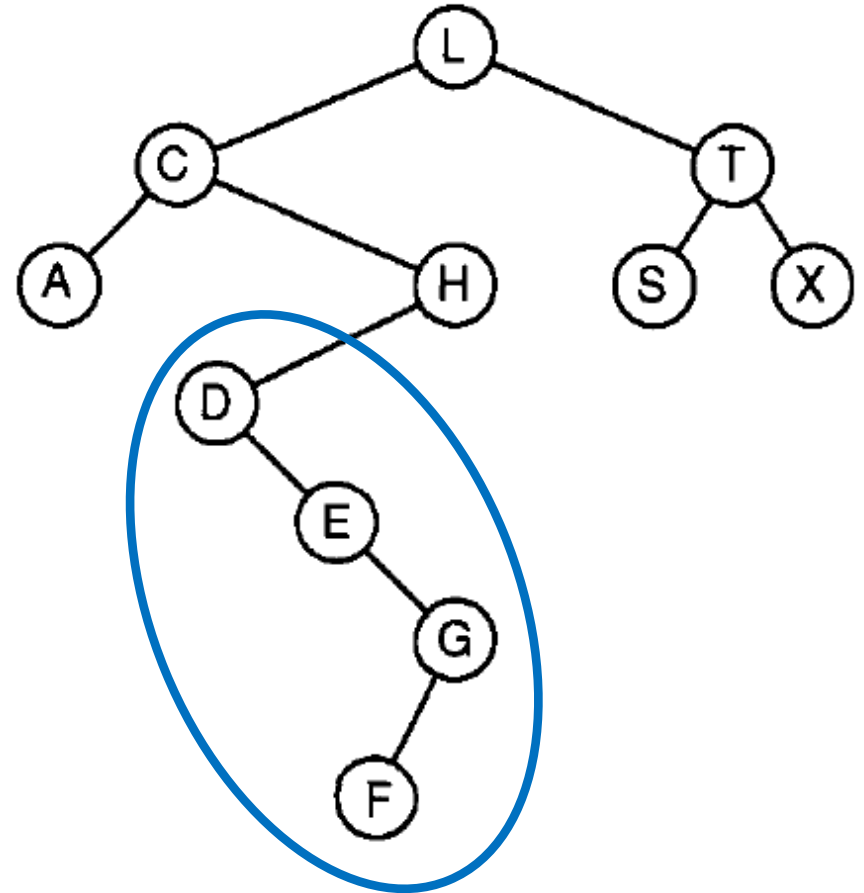
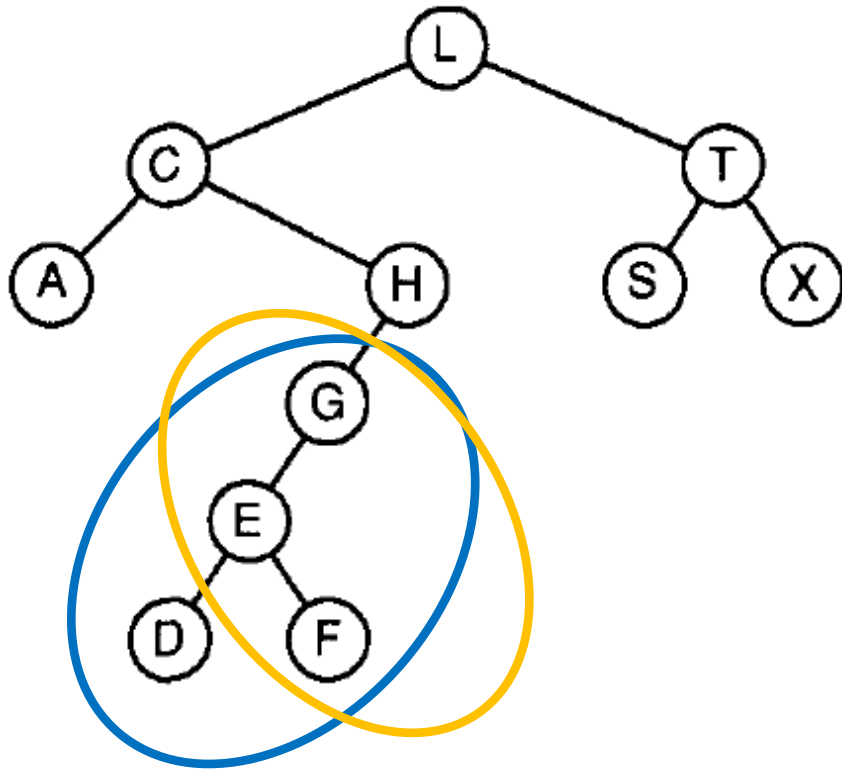
Реализация операции *splay*

- Шаг 3в): отец узла P – правый сын, а P – левый сын (или наоборот)
 - выполняем два однократных поворота в противоположных направлениях (сначала вокруг отца P направо, потом вокруг деда P налево)



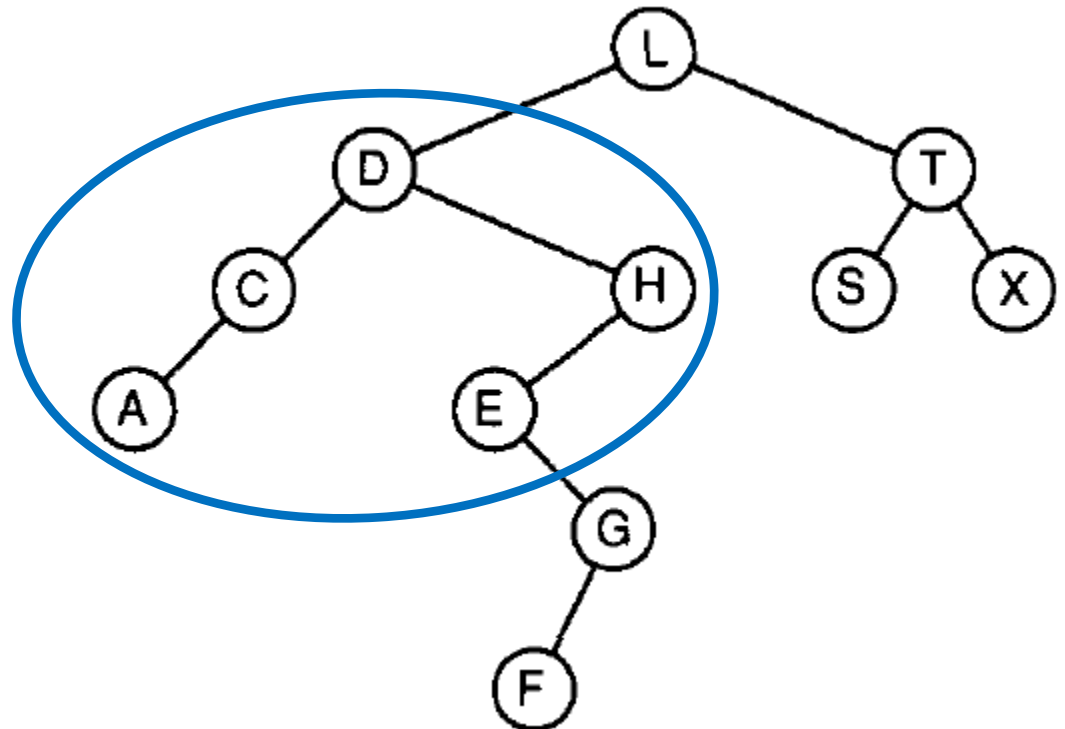
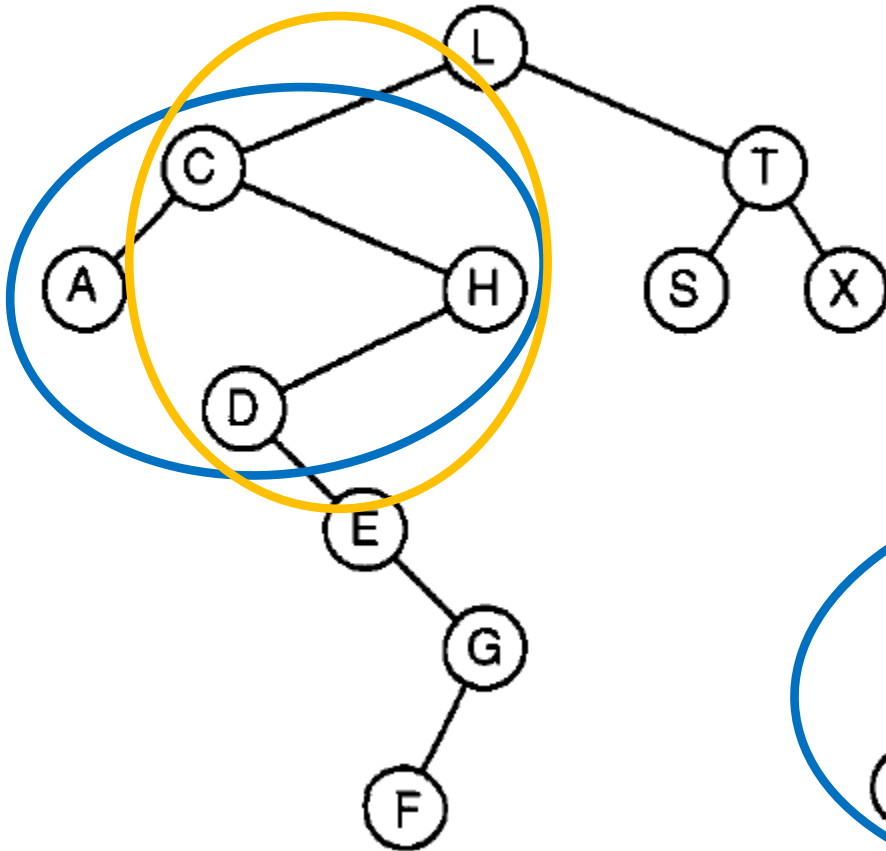
Пример операции *splay* над узлом *D*

◇ Случай б): отец узла *D* (*E*) и сам узел *D* – оба левые сыновья



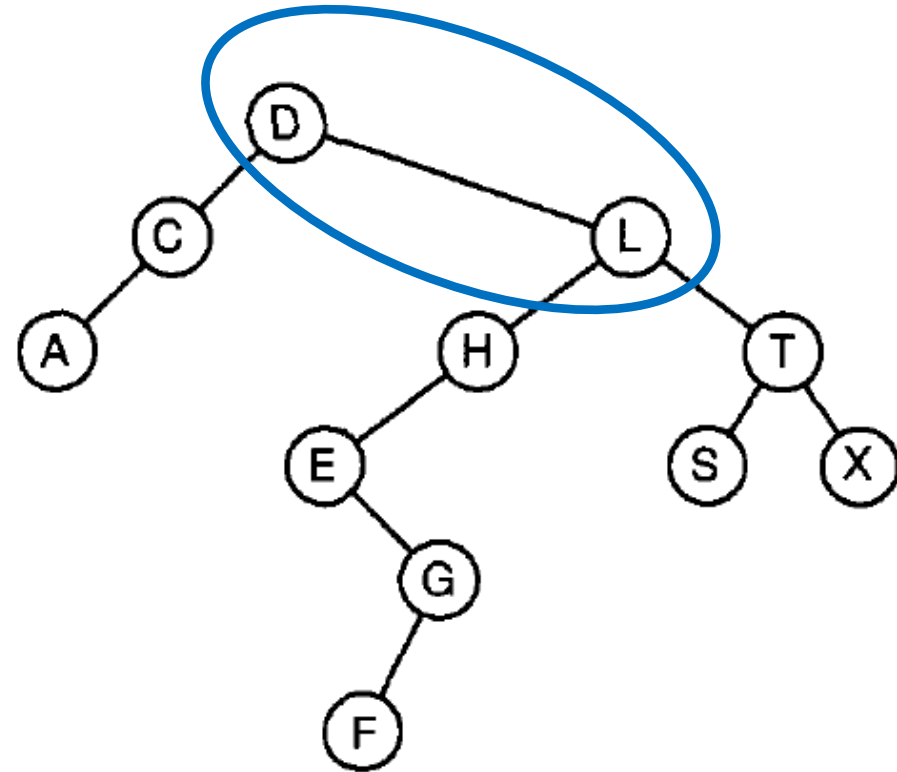
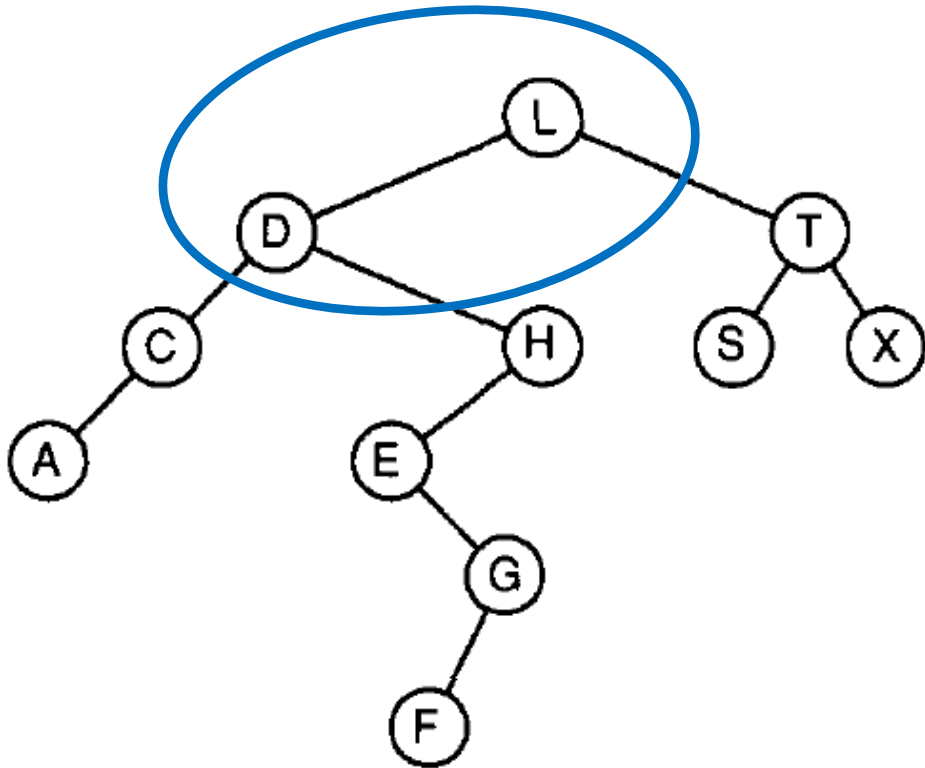
Пример операции *splay* над узлом *D*

◇ Случай в): отец узла *D* (*H*) – правый сын, а сам узел *D* – левый сын



Пример операции *splay* над узлом *D*

◇ Случай а): отец узла *D* (*L*) – корень дерева



Сложность операции *splay*

- ◆ Пусть каждый узел дерева содержит некоторую сумму денег.
 - ◆ Весом узла является количество ее потомков, включая сам узел
 - ◆ Рангом узла $r(N)$ называется логарифм ее веса
 - ◆ Денежный инвариант: во время всех операций с деревом каждый узел содержит $r(N)$ рублей
 - ◆ Каждая операция с деревом стоит фиксированную сумму за единицу времени
- ◆ Лемма. Операция *splay* требует *инвестирования* не более чем в $3\lfloor \lg n \rfloor + 1$ рублей с **сохранением** денежного инварианта.
- ◆ Теорема. 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.
 - ◆ Каждая операция требует не более $O(\log n)$ инвестиций, при этом может использовать деньги узла
 - ◆ По лемме инвестируется всего не более $m(3\lfloor \lg n \rfloor + 1)$ рублей, сначала дерево содержит 0 рублей, в конце содержит ≥ 0 рублей – $O(m \log n)$ хватает на все операции.

Сбалансированные деревья: обобщение через ранги

- ◇ Haeupler, Sen, Tarjan. Rank-balanced trees. ACM Transactions on Algorithms, 2015.
- ◇ Обобщение разных видов сбалансированных деревьев через понятие ранга (rank) и ранговой разницы (rank difference)
 - ◆ AVL, красно-черные деревья, 2-3 деревья, B-деревья
- ◇ Новый вид деревьев: слабые AVL-деревья (weak AVL)
- ◇ Анализ слабых AVL-деревьев, анализ потенциалов

Сбалансированные деревья: понятие ранга

- ◇ Ранг (rank) вершины $r(x)$: неотрицательное целое число
 - ◆ Ранг отсутствующей (null) вершины равен -1
- ◇ Ранг дерева: ранг корня дерева
- ◇ Ранговая разница (rank difference): если у вершины x есть родитель $p(x)$, то это число $r(p(x)) - r(x)$.
 - ◆ У корня дерева нет ранговой разницы
- ◇ i -сын: вершина с ранговой разницей, равной i .
- ◇ i,j -вершина: вершина, у которой левый сын — это i -сын, а правый сын — это j -сын. Один или оба сына могут отсутствовать. i,j - и j,i -вершины не различаются.

Сбалансированные деревья: ранговый формализм

- ◆ Конкретный вид сбалансированного дерева определяется *рангом* и *ранговым правилом*.
- ◆ Ранговое правило должно гарантировать:
 - ◆ Высота дерева (h) превосходит его ранг не более чем в константное количество раз (плюс, возможно, $O(1)$)
 - ◆ Ранг вершины (k) превосходит *логарифм* ее размера (n) не более чем в константное количество раз (плюс, возможно, $O(1)$)
Размер вершины – число ее потомков, включая себя, т.е. размер поддерева с корнем в этой вершине
 - ◆ Т.е. $h = O(k)$, $k = O(\log n) \rightarrow h = O(\log n)$
- ◆ Совершенное дерево:
ранг дерева – его высота; все вершины – 1,1.

Сбалансированные деревья: ранговые правила

- ◇ АВЛ-правило: каждая вершина – 1,1 или 1,2.
 - ◆ Ранг: высота дерева.
(или: все ранги положительны, каждая вершина имеет хотя бы одного 1-сына)
 - ◆ Можно хранить один бит, указывающий на ранговую разницу вершины
- ◇ Красно-черное правило: ранговая разница любой вершины равна 0 или 1, при этом родитель 0-сына не может быть 0-сыном.
 - ◆ 0-сын – красная вершина, 1-сын – черная вершина
 - ◆ Ранг: черная высота
 - ◆ Корень не имеет цвета (т.к. не имеет ранговой разницы!)
- ◇ Слабое АВЛ-правило: ранговая разница любой вершины равна 1 или 2; все листья имеют ранг 0.
 - ◆ Вдобавок к АВЛ-деревьям разрешаются 2,2-вершины
 - ◆ Бит на узел для ранговой разницы или ее *четности*
 - ◆ Балансировка: не более двух поворотов и $O(\log n)$ изменений ранга для вставки/удаления, при этом амортизированно – лишь $O(1)$ изменений.
 - ◆ Слабое АВЛ-дерево является красно-черным деревом