Курс «Алгоритмы и алгоритмические языки»

1 семестр 2014/2015

Лекция 23

Цифровой поиск

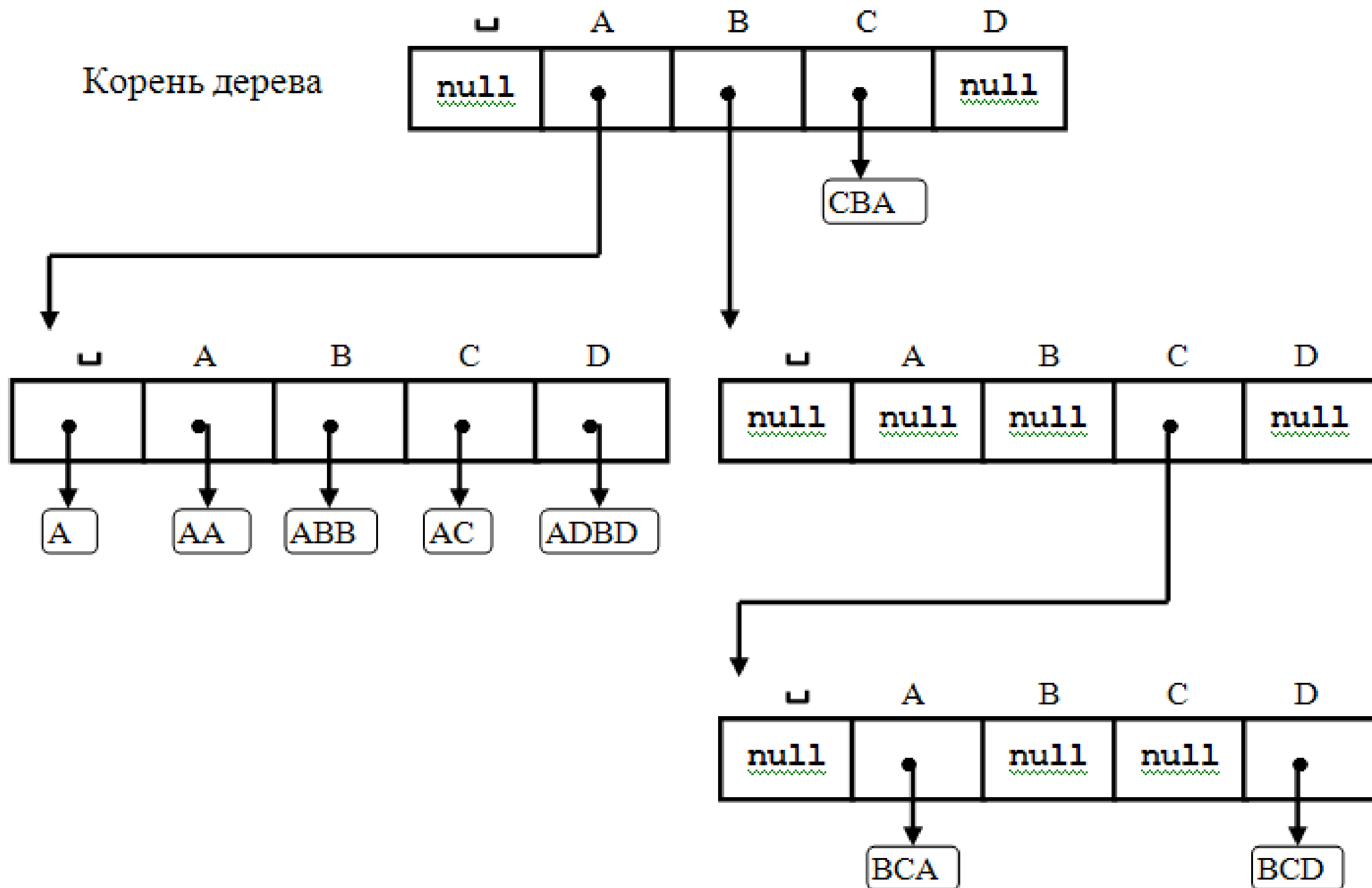
- *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*).
- *Примеры цифрового поиска*: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).
- Хороший пример – *словарь с высечками*, т.е. словарь, в котором обеспечен быстрый доступ к некоторым страницам (например, начальным страницам списков слов, начинающихся на очередную букву алфавита). Иногда используются *многоуровневые высечки*.
- При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). Ожидаемое число сравнений порядка $O(\log_m N)$, где m - число различных букв, используемых в словаре, N – мощность словаря. В худшем случае дерево содержит k уровней, где k – длина максимального слова.

Цифровой поиск

- *Пример*. Пусть множество используемых букв (алфавит) {*A*, *B*, *C*, *D*}. Мы добавим к алфавиту еще одну букву ␣ (пробел). По определению слова *AA*, *AA*␣, *AA*␣␣ совпадают. Пусть {*A*, *AA*, *ABB*, *AC*, *ADBD*, *BCA*, *BCD*, *CBA*} – словарь (множество ключей).
- Построим *m*-ичное дерево, где $m = 5 = |␣, A, B, C, D|$. Следующая небольшая хитрость позволит иногда сократить поиск: если в словаре есть слово $a_1a_2a_3...a_k$ и первые *i* его букв ($i < k$) задают уникальное значение: комбинация $a_1...a_i$ встречается в словаре только один раз, то не нужно строить дерево для $j > i$, так как слово можно идентифицировать по первым *i* буквам.
- Очень важное обобщение цифрового поиска: таким же образом можно обрабатывать любые ключи, не привязываясь к байту (8 битам), который обычно используется для кодирования символов алфавита. Мы можем отсекать от ключа первые *m* бит, использовать 2^m-ичное разветвление, т.е. строить 2^m-ичное дерево поиска (на двоичных деревьях для разветвления берется один бит: $m = 1$).

Цифровой поиск

- Прямоугольниками изображены вершины дерева, в овалах – значения слов (ключей) и связанная с ним информация. Тем самым любая вершина дерева – массив из *m* элементов. Каждый элемент вершины содержит либо ссылку на другую вершину *m*-ичного дерева, либо на овал (ключ).



Цифровой поиск

- Иногда используют комбинации нескольких методов: цифровой поиск вначале, а затем переключение на поиск в последовательных таблицах.
  - Именно так мы и работаем со словарем с высечками: вначале на высечку, а затем либо последовательный поиск, либо дихотомический.
- Обычно предлагается пользоваться цифровым поиском, пока количество различных слов не меньше некоторого k, а затем переключаться на последовательные таблицы.
- Обобщения: поиск по неполным ключам, поиск по образцу.

Цифровой поиск

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define M 5

typedef enum {word, node} tag_t;
struct record {
  char *key;
  int value;
};

struct tree {
  tag_t tag;
  union {
    struct record *r;
    struct tree *nodes[M+1];
  };  /* анонимное объединение */
};
```

Цифровой поиск: *поиск элемента*

```
static inline int ord (char c) {
  return c ? c - 'A' + 1 : 0;
}

struct record *find (struct tree *t, char *key) {
  int i = 0;

  while (t) {
    switch (t->tag) {
    case word:
      for (; key[i]; i++)
        if (key[i] != t->r->key[i])
          return NULL;
      return t->r->key[i] ? NULL : t->r;
    case node:
      t = t->nodes[ord(key[i])];
      if (key[i])
        i++;
    }
  }
  return NULL;
}
```

Цифровой поиск: *вставка – вспомогательные функции*

```
struct record *make_record (char *key, int value) {
  struct record *r = malloc (sizeof (struct record));
  r->key = strdup (key);
  r->value = value;
  return r;
}

struct tree *make_word (char *key, int value) {
  struct tree *t = malloc (sizeof (struct tree));
  t->tag = word;
  t->r = make_record (key, value);
  return t;
}

struct tree *make_node (void) {
  struct tree *t = calloc (1, sizeof (struct tree));
  t->tag = node;
  return t;
}

struct tree *make_from_record (struct record *r) {
  struct tree *t = malloc (sizeof (struct tree));
  t->tag = word;
  t->r = r;
  return t;
}
```

Цифровой поиск: *вставка элемента*

```
struct tree *insert (struct tree *t, char *key, int value) {
  if (!t)
    return make_word (key, value);

  int i = 0;
  struct tree *root = t;

  /* skip all nodes */
  while (t->tag == node) {
    struct tree **p = &t->nodes[ord(key[i++])];
    if (!*p) {
      *p = make_word (key, value);
      return root;
    }
    t = *p;
  }

  /* all word skipped -- key exists, update value */
  if (i && !key[i - 1]) {
    t->r->value = value;
    return root;
  }
```

Цифровой поиск: *вставка элемента*

```
/* compare the remaining part */
int j = i;
for (; key[i]; i++)
  if (key[i] != t->r->key[i])
    break;

/* key already exists -- update value */
if (!key[i] && !t->r->key[i]) {
  t->r->value = value;
  return root;
}

/* turn t into a node */
struct record *other = t->r;
t->tag = node;
memset (t->nodes, 0, sizeof (t->nodes));
```

Цифровой поиск: *вставка элемента*

```
 /* make new nodes for remaining common prefix */
  for (; j < i; j++) {
    struct tree *p = make_node ();
    t->nodes[ord(key[j])] = p;
    t = p;
  }

  /* accommodate both other and new record */
  t->nodes[ord(other->key[i])]
    = make_from_record (other);
  t->nodes[ord(key[i])] = make_word (key, value);
  return root;
}
```

Цифровой поиск: *печать элементов*

```
void print (struct tree *t, char c) {
  static int level = 0;
  if (!t) {
    printf ("empty\n");
    return;
  }
  for (int i = 0; i < level; i++)
    putchar (' ');
  if (level)
    printf ("%c: ", chr (c));
  if (t->tag == word) {
    printf ("word: %s %d\n", t->r->key, t->r->value);
  } else {
    printf ("node: ");
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        printf ("%c ", chr(i));
    putchar ('\n');
    level++;
    for (int i = 0; i < M + 1; i++)
      if (t->nodes[i])
        print (t->nodes[i], i);
    level--;
  }
}
```

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.

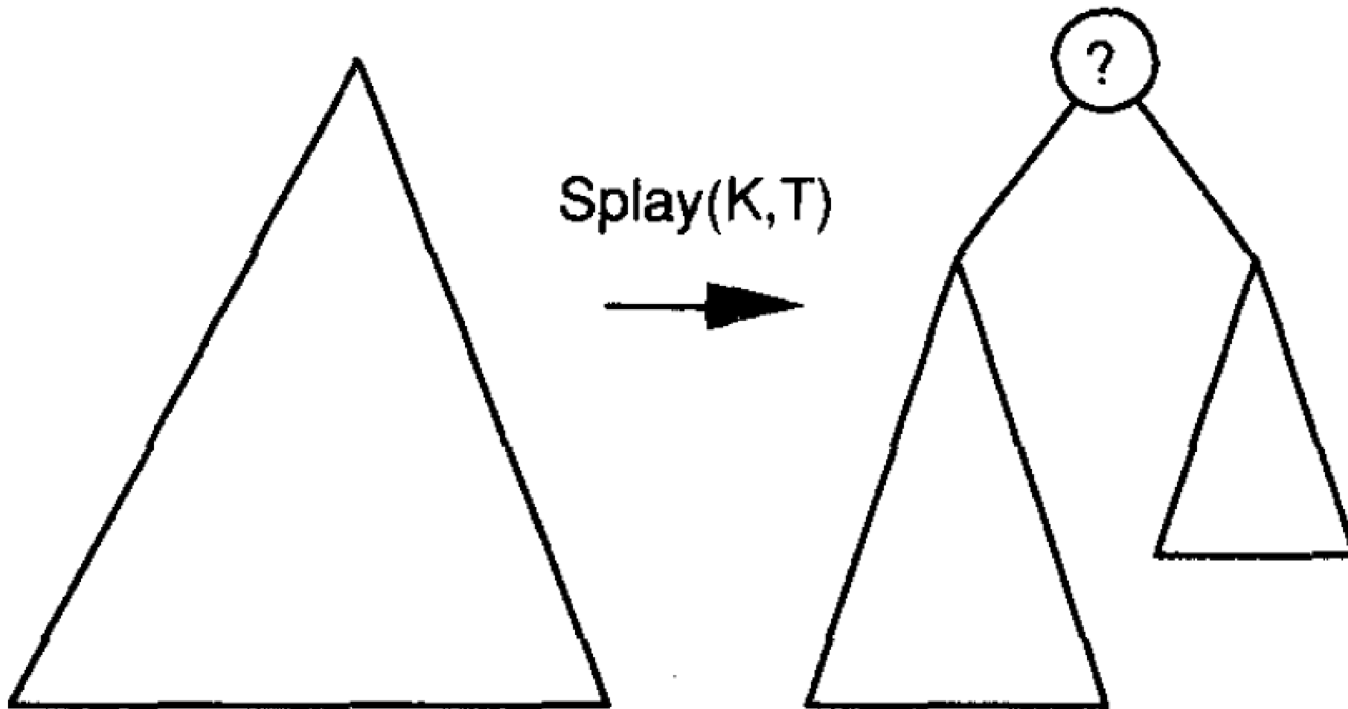
  http://www.amazon.com/Structures-Their-Algorithms-Harry-Lewis/dp/067339736X

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

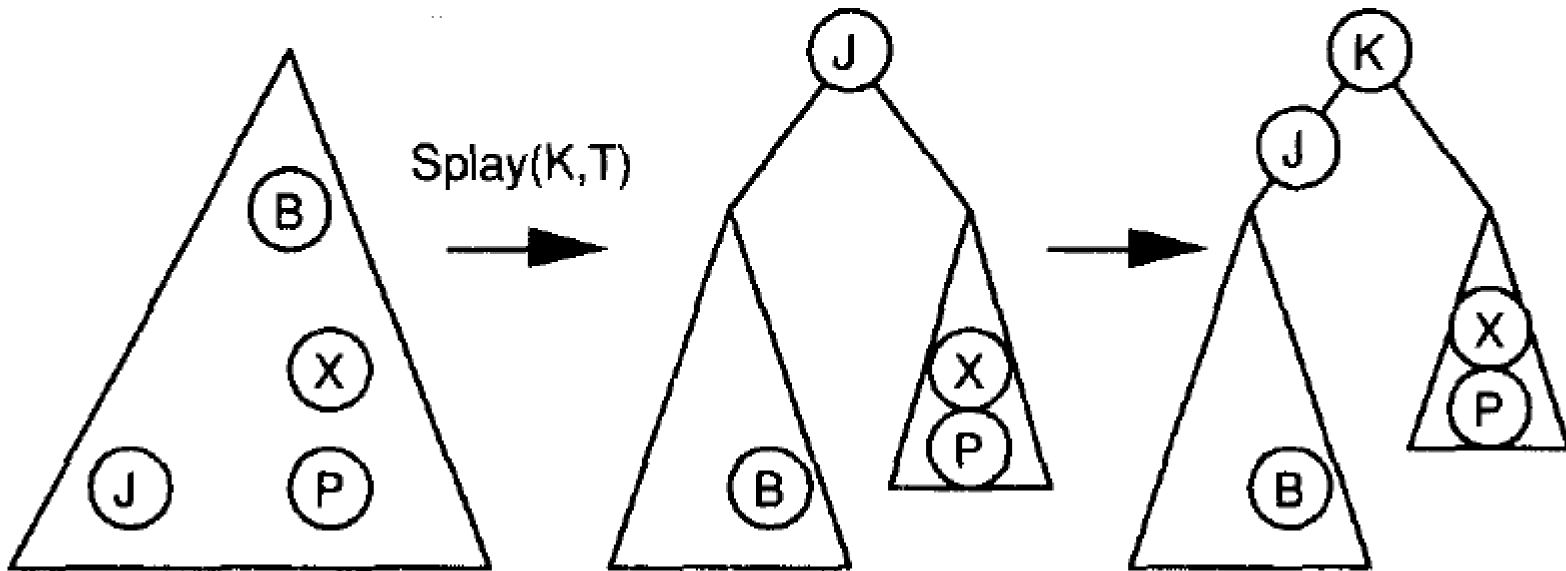
- Поиск (LookUp): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение равно K, то ключ найден



Реализация словарных операций через splay

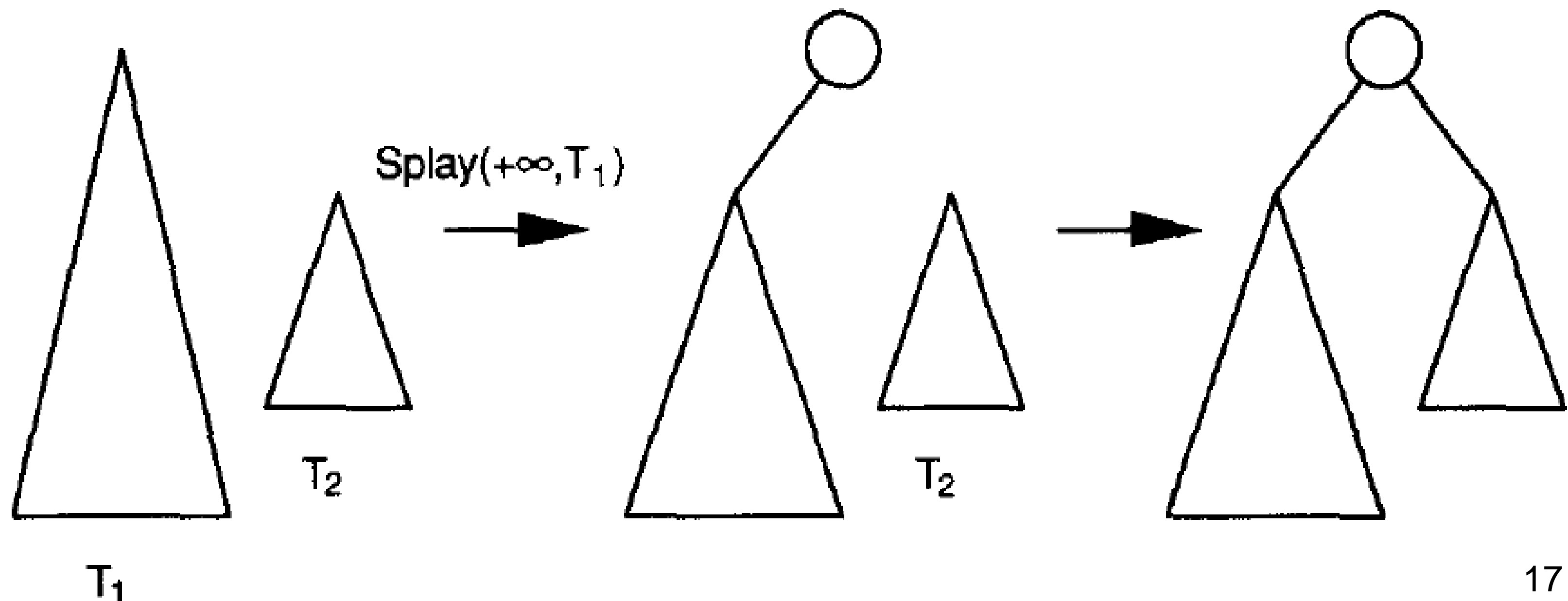
- Вставка (Insert): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение уже равно K, то обновим данные ключа
  - если значение другое, то вставим новый корень K и поместим старый корень J слева или справа

    (в зависимости от значения J)



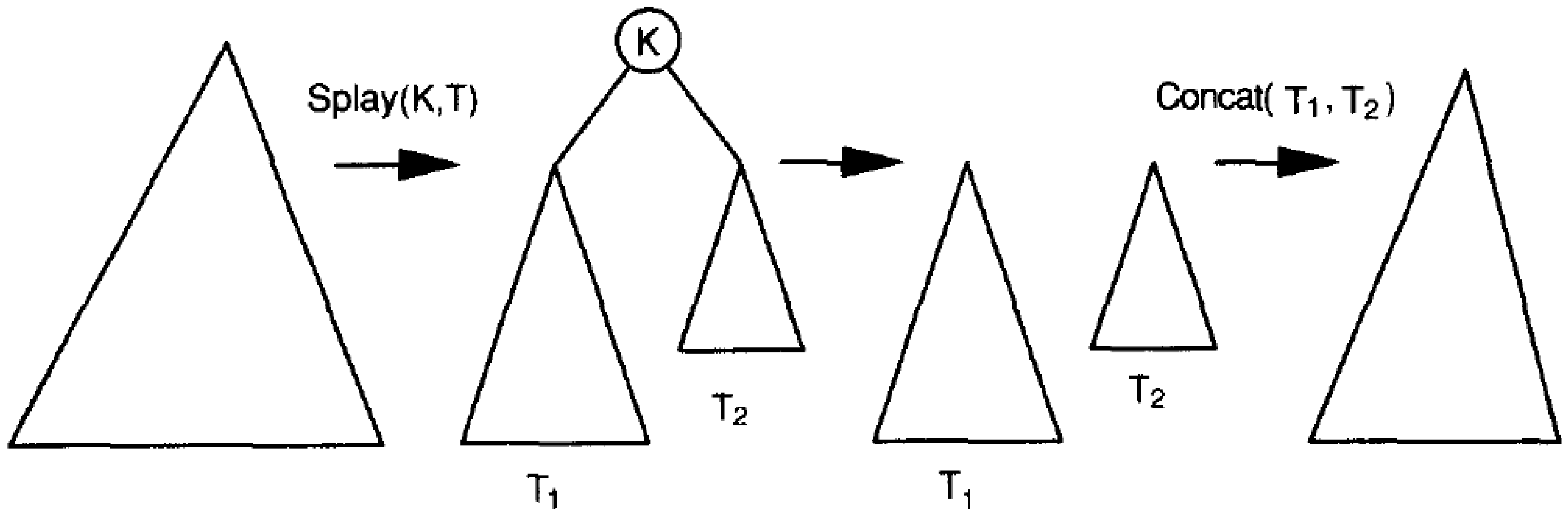
Реализация словарных операций через splay

- 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

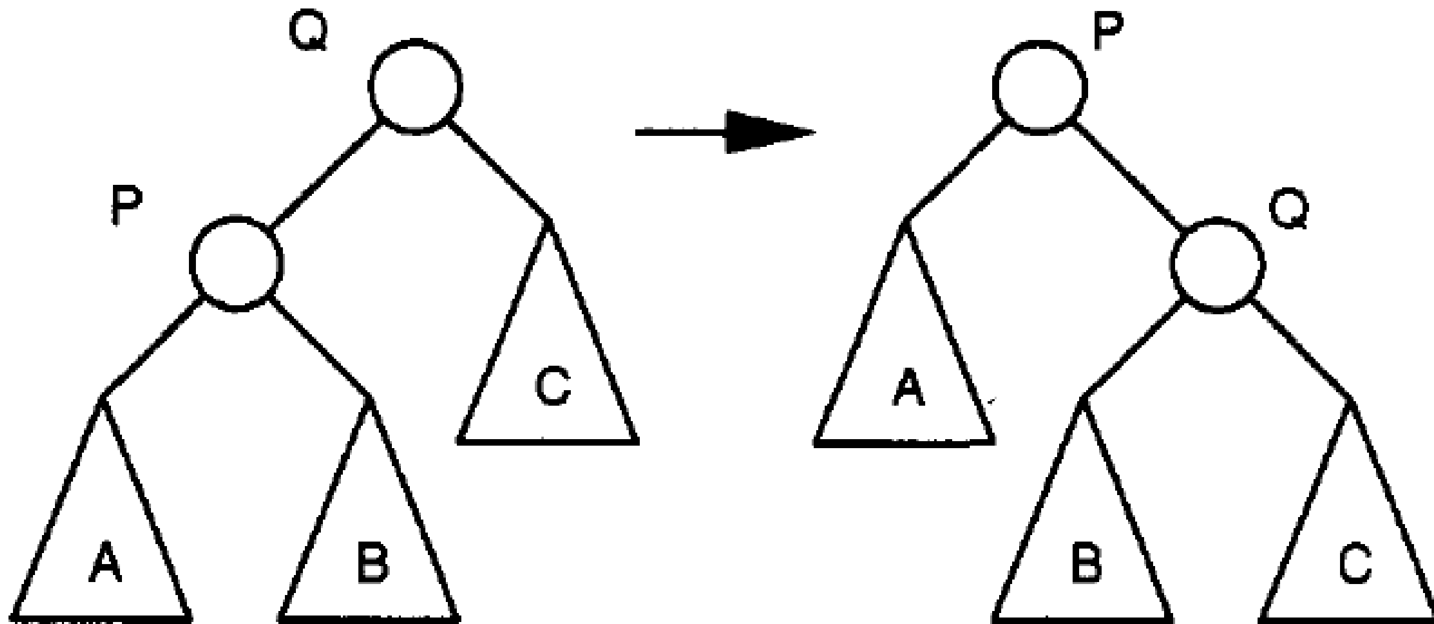


Реализация операции splay

- 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)

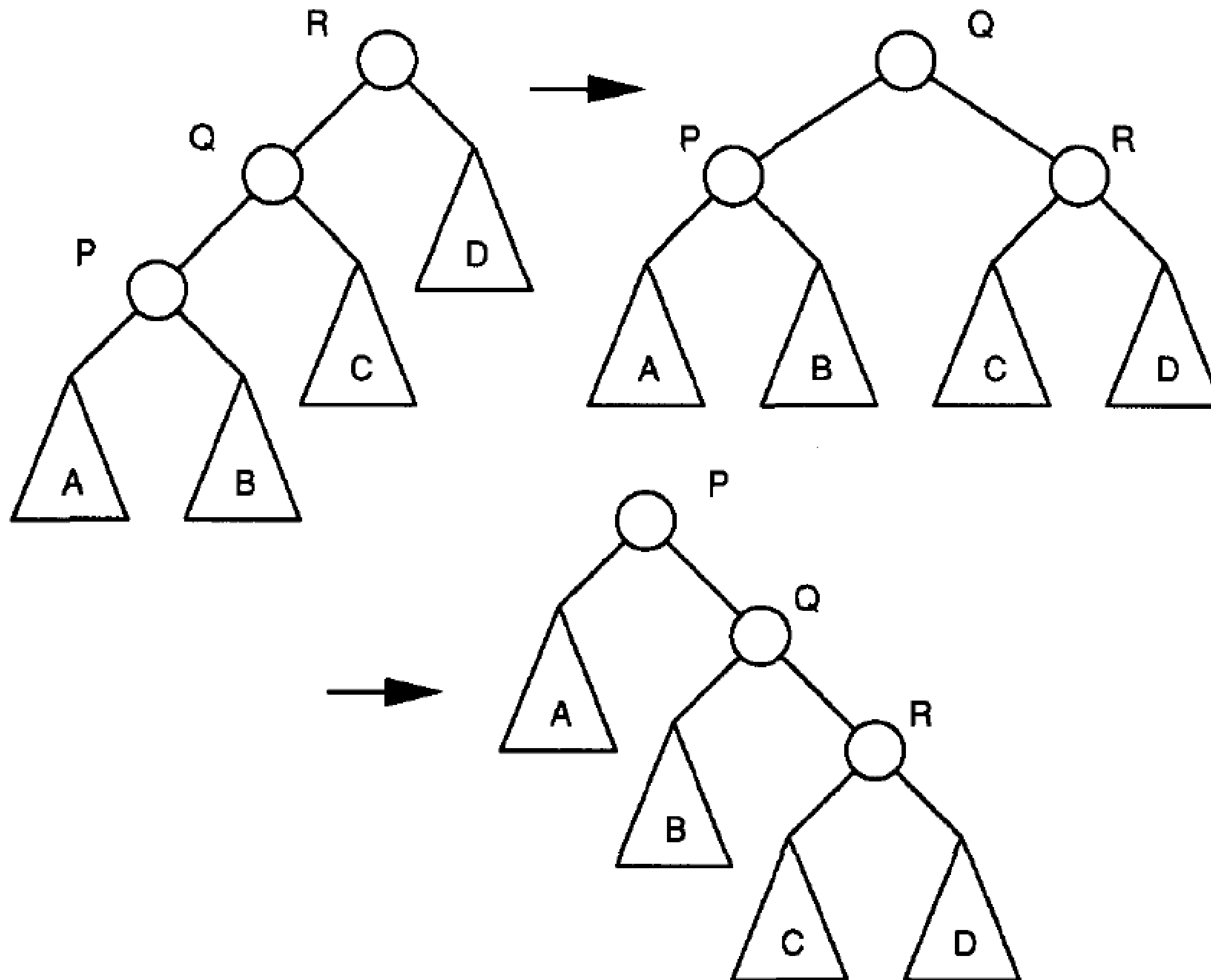
Реализация операции splay

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



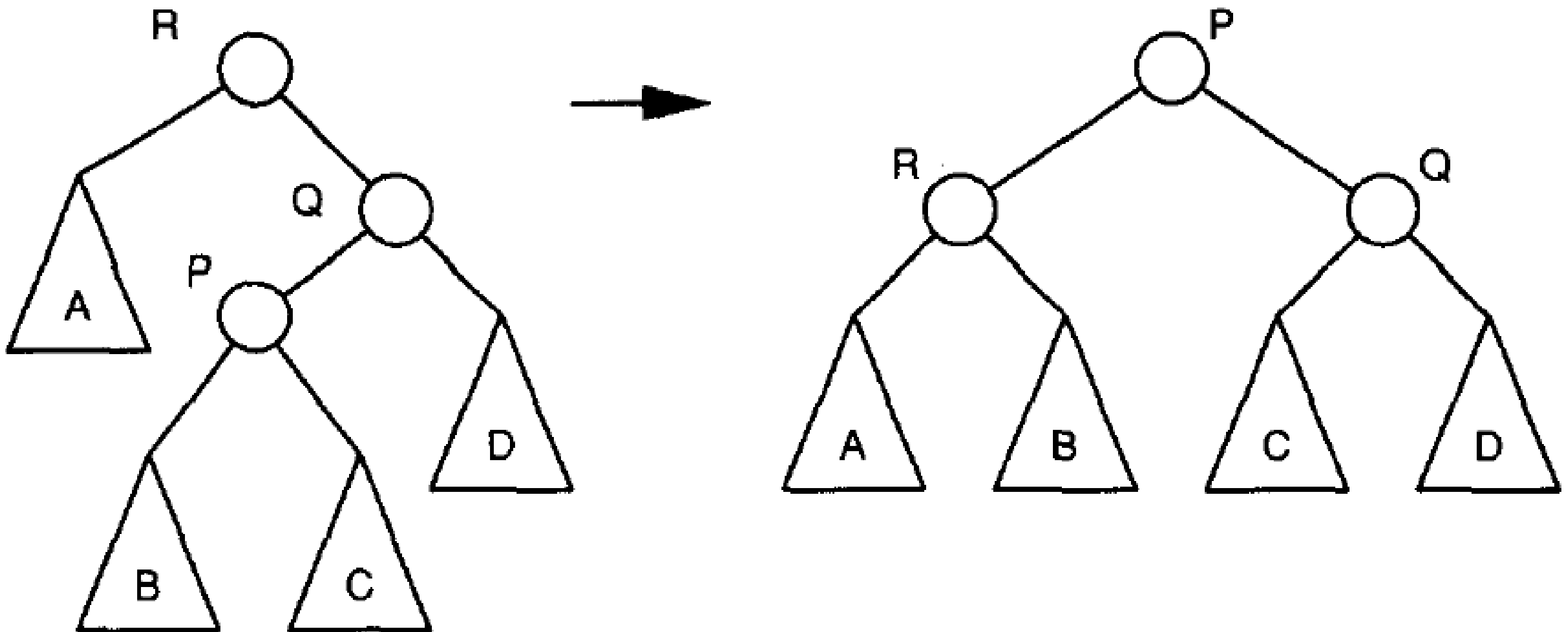
Реализация операции splay

- 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



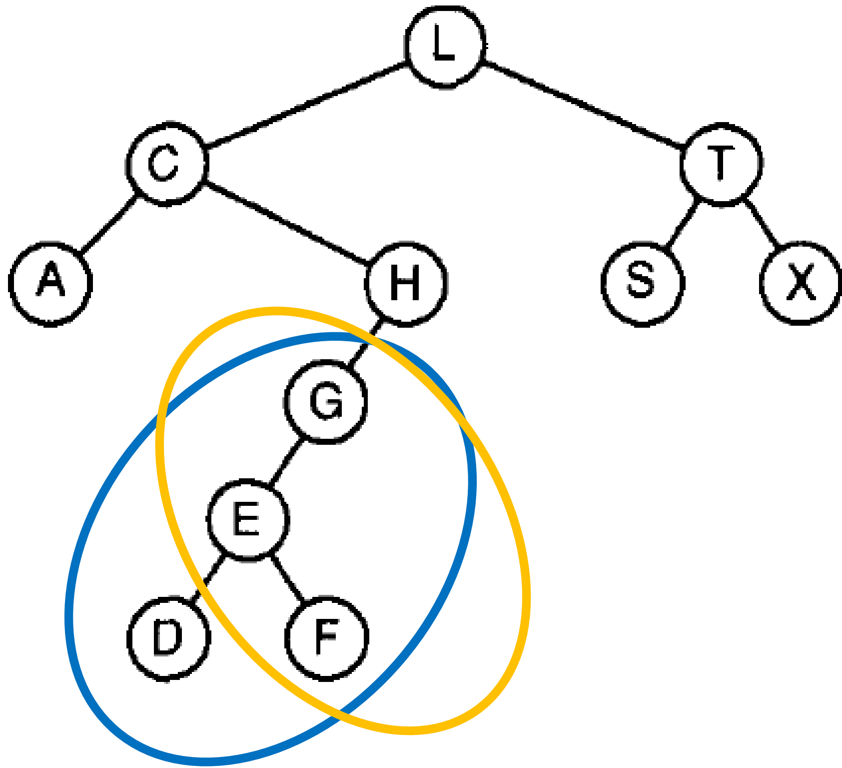
Реализация операции splay

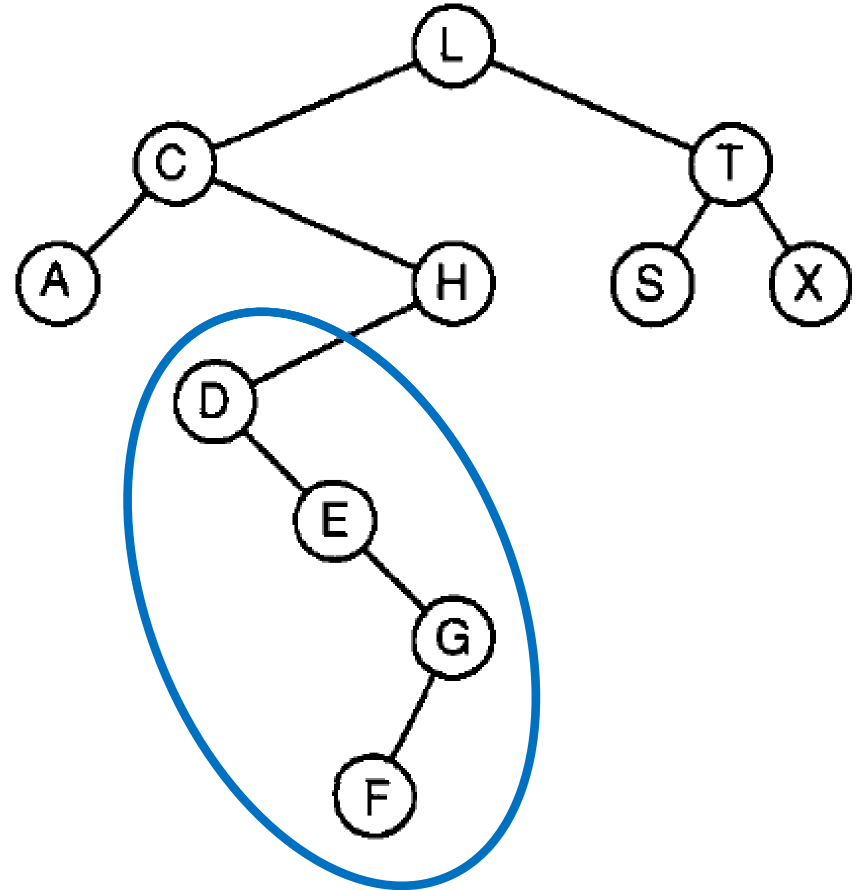
- Шаг 3в): отец узла P – правый сын, а P – левый сын (или наоборот)
  - выполняем два однократных поворота в противоположных направлениях (сначала вокруг отца P направо, потом вокруг деда P налево)



Пример операции splay над узлом D

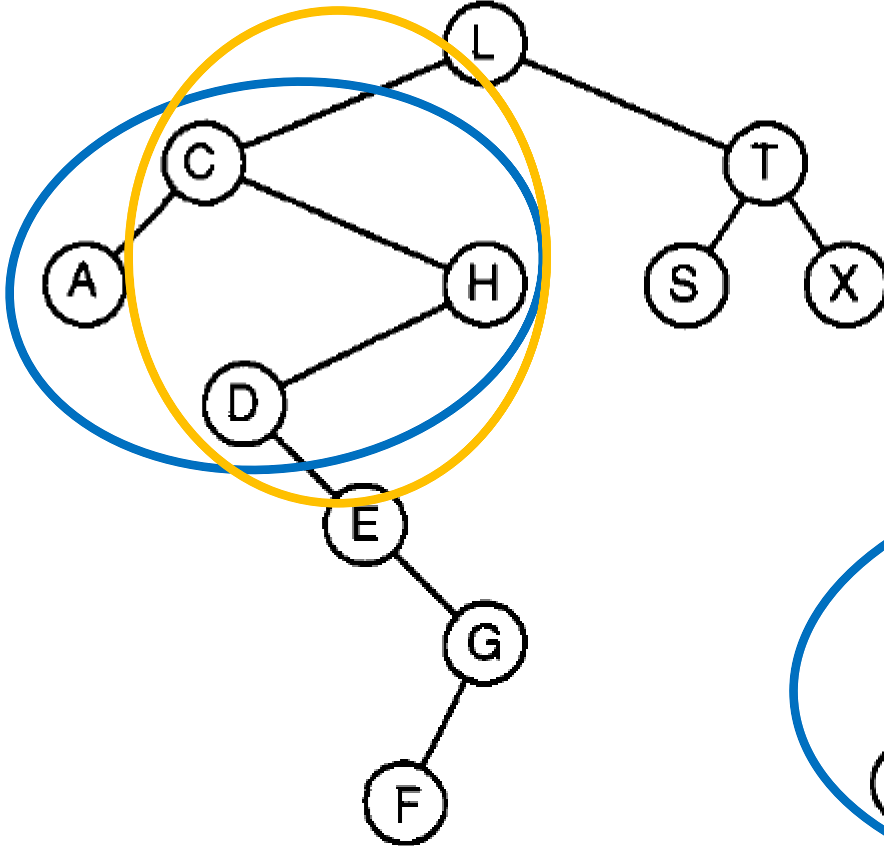
- Случай б): отец узла D (E) и сам узел D – оба левые сыновья

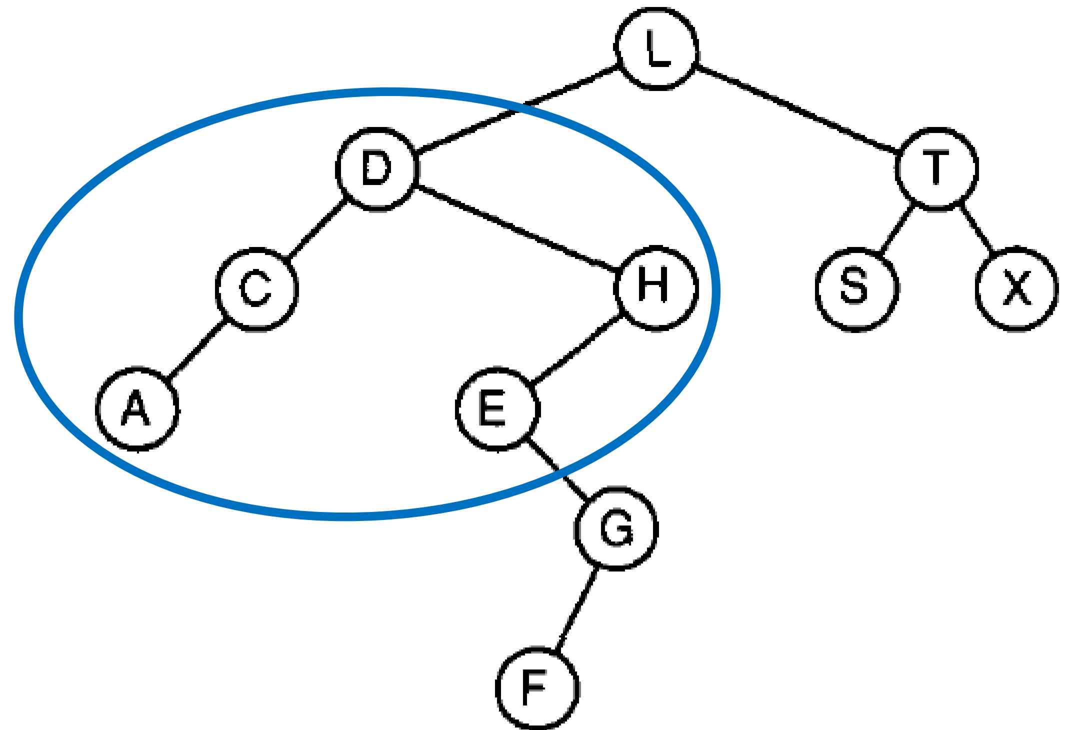




Пример операции splay над узлом D

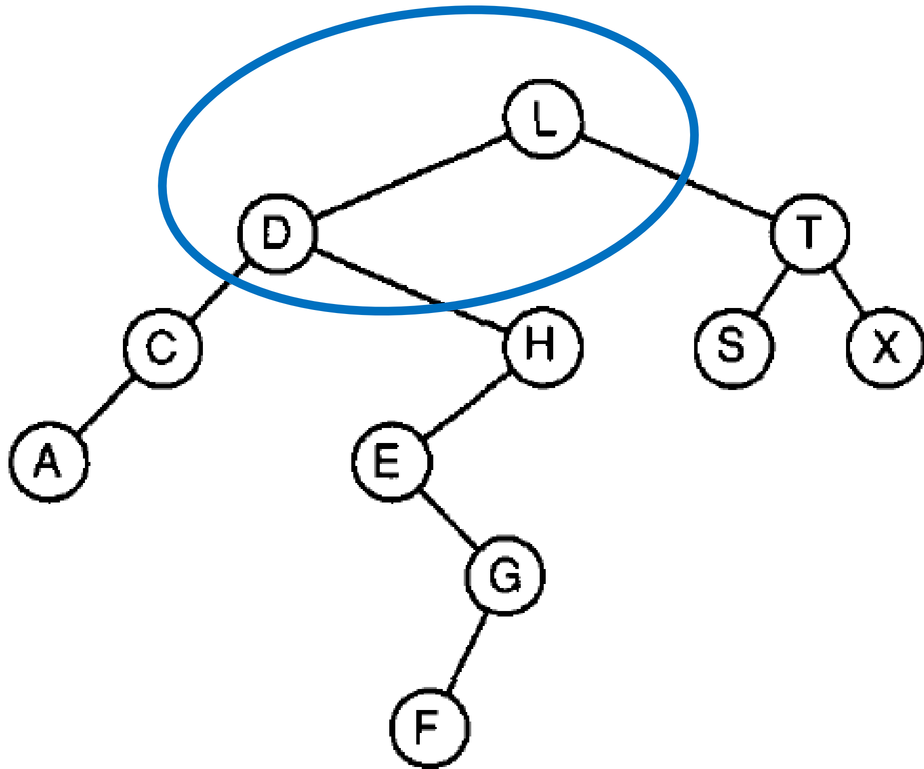
- Случай в): отец узла D (H) – правый сын, а сам узел D – левый сын

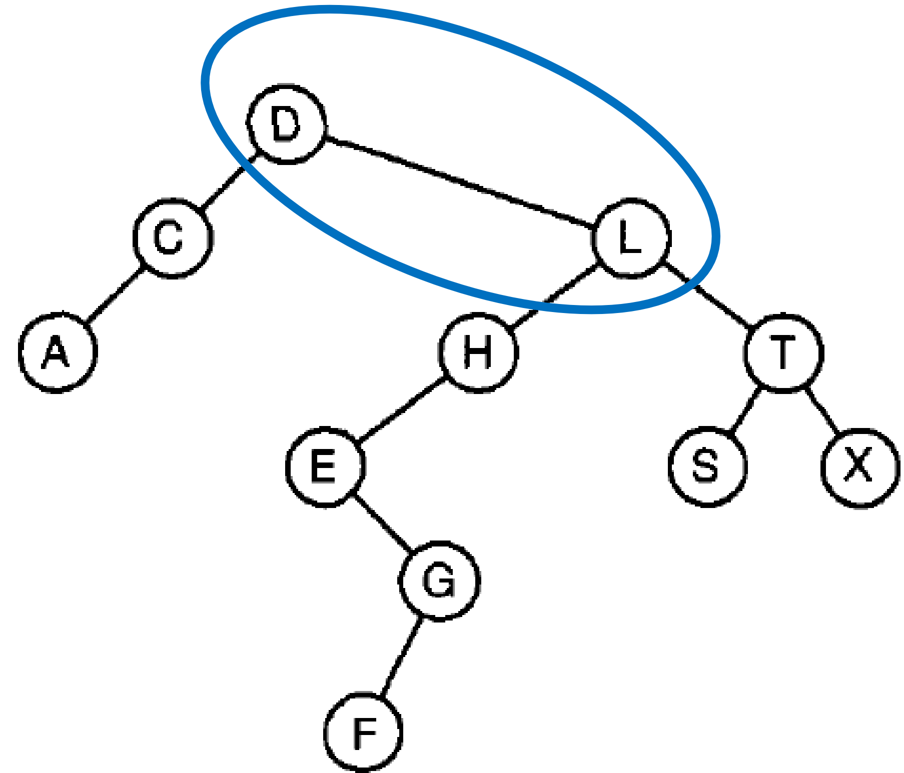




Пример операции splay над узлом D

- 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 $\mathrm{O}(m \log n)$ времени.
  - Каждая операция требует не более $\mathrm{O}(\log n)$ 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 – $\mathrm{O}(m \log n)$ хватает на все операции.

Сбалансированные деревья: *обобщение через ранги*

- Haeupler, Sen, Tarjan. Rank-balanced trees. ACM Transactions on Algorithms, 2014 (to appear).
- Обобщение разных видов сбалансированных деревьев через понятие ранга (rank) и ранговой разницы (rank difference)
    - АВЛ, красно-черные деревья, 2-3 деревья, B-деревья
- Новый вид деревьев: слабые АВЛ-деревья (weak AVL)
- Анализ слабых АВЛ-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