Курс «Алгоритмы и алгоритмические языки»

Лекция 20

Красно-черные деревья

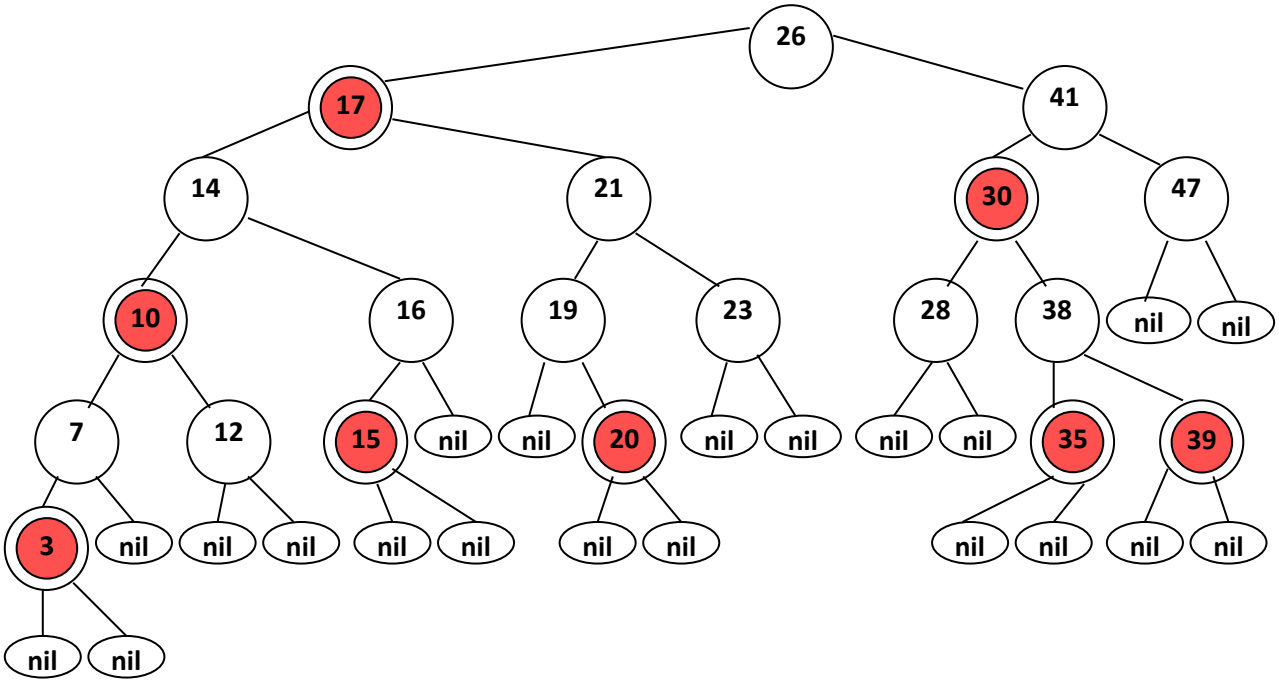
- Красно-черное дерево – двоичное дерево поиска, каждая вершина которого окрашена либо в красный, либо в черный цвет
- Поля – цвет, дети, родители

```
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 $2log_2(n+1)$.

  (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 ее сын 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 $2log_2(n+1)$.

  (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 $h \leq 2log_2(n + 1)$. Лемма доказана.

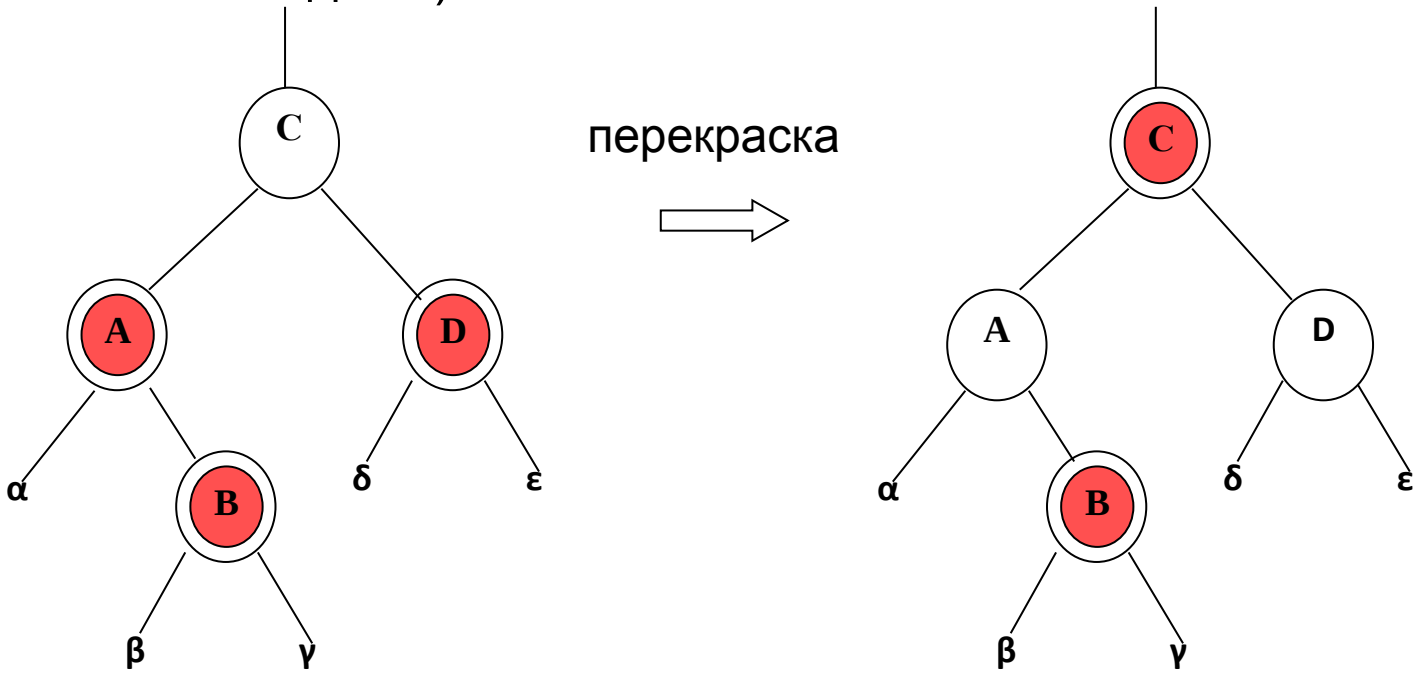
- Следовательно, поиск по красно-черному дереву имеет сложность $O\ (log_2 n)$.

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

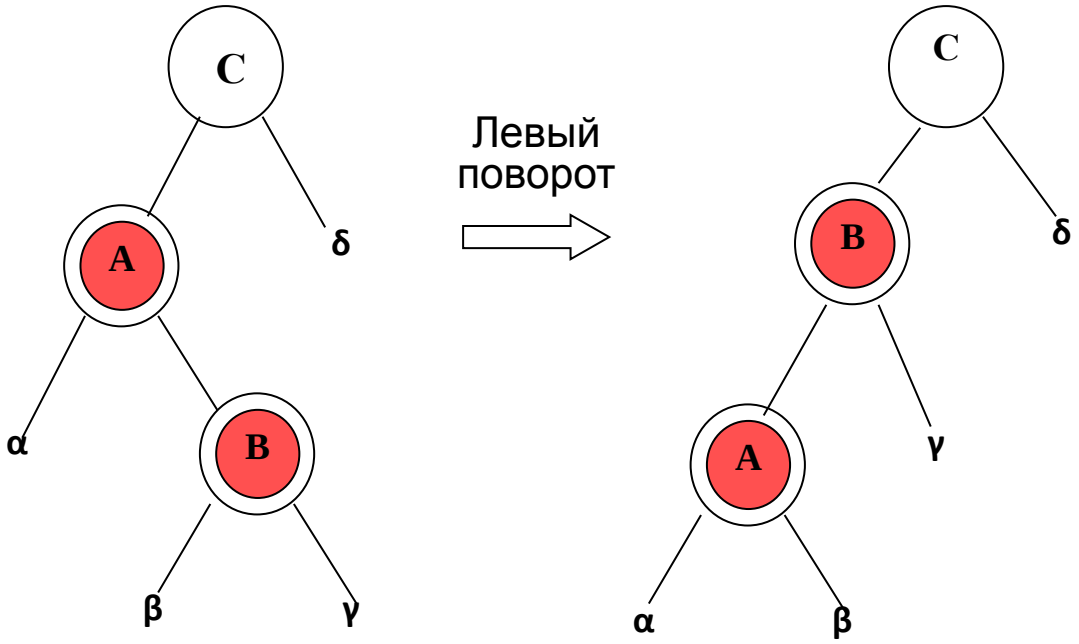
Красно-черные деревья: вставка вершины

- 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)



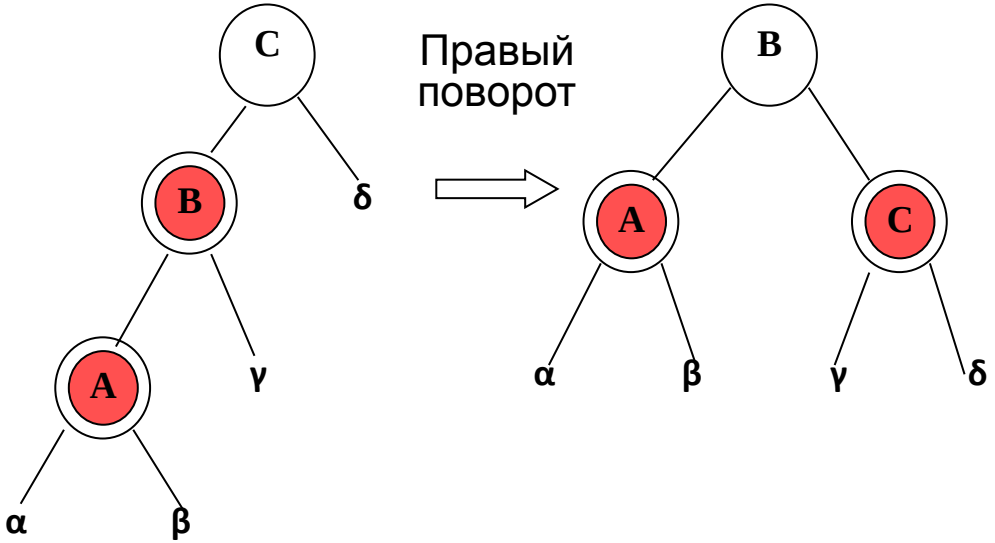
Красно-черные деревья: вставка вершины

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



Красно-черные деревья: вставка вершины

- Случай 2: “дядя” (второй сын родителя родителя текущей вершины) черный
  - Шаг 2: Необходимо выполнить правый поворот вершины C, после чего перекрасить вершины B и C



Хеш-таблицы

- Словарные операции: *добавление*, *поиск* и *удаление* элементов по их ключам.
- Организуется таблица ключей: массив **Index[m]** длины **m**, элементы которого содержат значение ключа и указатель на данные (информацию), соответствующие этому ключу.
  - ***Прямая адресация***. Применяется, когда количество возможных ключей невелико: например, ключи перенумерованы целыми числами из множества $U = \{0, 1, 2, \ldots, m - 1\}$, где m не очень большое число.
  - В случае прямой адресации ключ с номером k соответствует элементу **Index[k]**. Этот ключ обычно не записывается в элемент массива, т.к. совпадает с индексом.
  - Все три словарные операции выполняются за время порядка $O(1)$.
  - Основной недостаток прямой адресации – таблица *Index* занимает слишком много места, если множество всевозможных ключей U достаточно велико (m большое целое число).

Хеш-таблицы

- ***Хеширование*** тоже позволяет обеспечить среднее время операций с данными $T_{ср}(n) = O(1)$ и тоже за счет использования таблицы *Index*.
- Хеш-таблица использует память объемом $\Theta(|K|)$, где $|K|$ – мощность множества использованных ключей (правда это оценка в среднем, а не в худшем случае, да и то при определенных предположениях).
- Пример использования хеширования – таблица идентификаторов программы, составляемая компилятором.
- В случае хеш-адресации элементу с ключом *key* отводится строка таблицы с номером *hash*(*key*), где
  hash: $U \to \{0, 1, 2, \ldots, m - 1\}$ – *хеш-функция*.
  Число *hash*(*key*) называется *хеш-значением* ключа *key*.
- Если хеш-значения ключей key_1 и key_2 совпадают ($hash(key_1) == hash(key_2)$), говорят, что случилась *коллизия*.
  Выбрать хеш-функцию, для которой коллизии исключены, возможно лишь тогда, когда все возможные значения ключей заранее известны.
  В общем же случае коллизии неизбежны, так как $|U| > m$.

Хеш-таблицы

- Простейший способ обработки коллизий – сцепление элементов с одинаковыми значениями хеш-функции: все такие элементы сцепляются в список, а в хеш-таблицу помещается указатель на первый элемент этого списка. В пределах каждого такого списка осуществляется последовательный поиск.
- В случае использования двустороннего списка среднее время выполнения каждой из трех словарных операций будет иметь порядок $O(1)$. Основная трудность – в поиске по списку, но коллизий не очень много и *hash*(*key*) можно выбрать так, чтобы списки были достаточно короткими.
- Примером хеш-таблицы с цепочками является записная книжка с алфавитом.

Хеш-таблицы

- Устройство простой хеш-таблицы (реализация хеширования с цепочками).
  - Задается некоторое фиксированное число *m* (типичные значения *m* от 100 до 1,000,000).
  - Создается массив **Index[m]** указателей начал двунаправленных списков (цепочек), который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения *NULL*.
  - Задается хеш-функция *hash*(), которая получает на вход ключи и выдает значение от 0 до *m* – 1.
  - При добавлении пары (*key*, *value*) вычисляется *h* = *hash*(*key*) и пара добавляется в список **Index[h].**
  - При удалении либо поиске пары (*key*, *value*) вычисляется *h* = *hash*(*key*) и происходит удаление либо поиск пары (*key*, *value*) в списке **Index[h]**.

Хеш-таблицы

- Анализ хеширования с цепочками.
  - Пусть **Index[m]** – хеш-таблица с *m* позициями, в которую занесено *n* пар (*key*, *value*). Отношение $\alpha = n/m$ называется *коэффициентом заполнения* хеш-таблицы.
  - Коэффициент заполнения α позволяет судить о качестве хеш-функции:
    пусть $M = \frac{1}{m}\sum_{i=0}^{m-1} |\, Index[i]\,|$ – средняя длина списков;

    если *hash*(*key*) – «хорошая» хеш-функция, то
    дисперсия $D = \frac{1}{m}\sum_{i=0}^{m-1} \left(M - |\, Index[i]\,|\right)^2 \le \alpha$.
  - Это условие исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список и поиск в этом списке из *n* элементов имеет среднее время $\Theta(n)$.

Хеш-таблицы

- Анализ хеширования с цепочками.
  - *Равномерное хеширование*: хеш-функция подобрана таким образом, что каждый данный элемент может попасть в любую из *m* позиций хеш-таблицы с равной вероятностью, независимо от того, куда попали другие элементы.
  - Условие из предыдущего слайда выполняется и *средняя длина каждого из m списков хеш-таблицы с коэффициентом заполнения* α *равна* α.
  - Среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка α, так как поиск сводится к просмотру одного из списков.
  - Поскольку среднее время вычисления хеш-функции равно $\Theta(1)$, то среднее время выполнения каждой из словарных операций с учетом вычисления хеш-функции равно $\Theta(1 + \alpha)$.

Хеш-таблицы

- ***Теорема***. Пусть T – хеш-таблица с цепочками, имеющая коэффициент заполнения α, причем хеширование равномерно. Тогда при поиске элемента, **отсутствующего** в таблице, будет просмотрено в среднем α элементов таблицы, а, включая время на вычисление хеш-функции, будет равно $\Theta(1 + \alpha)$.
- ***Теорема***. При равномерном хешировании среднее время **успешного** поиска в хеш-таблице с коэффициентом заполнения α есть $\Theta(1 + \alpha)$.
  - ***Замечание***. Теорема не сводится к предыдущей, так как в предыдущей теореме оценивалось среднее число действий, необходимых для поиска случайного элемента, равновероятно попадающего в любую из ячеек таблицы.
  - В этой теореме сначала рассматривается случайно выбранная последовательность элементов, добавляемых в таблицу (на каждом шаге все значения ключа равновероятны и шаги независимы); потом в полученной таблице выбираем элемент для поиска, считая, что все ее элементы равновероятны.
- Из теорем следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть $O(1)$.

Методы построения хеш-функций

- Построение хеш-функции ***методом деления с остатком***.
  - Хеш-функция *hash*(*key*) определяется соотношением ***hash(key) = key % m***.
  - При правильном выборе *m* такая хеш-функция обеспечивает распределение, близкое к равномерному.
  - Правильный выбор *m*: в качестве *m* выбирается достаточно большое простое число, далеко отстоящее от степеней двойки.
  - Например, если устраивает средняя длина списков 3, а число записей, доступ к которым нужно обеспечить с помощью хеш-таблицы ≈ 2000, то можно взять *m* = простым числом близко к 2000/3, иногда используют 701. Тогда *hash*(*key*) = *key %* 701.
  - Недостаток: в качестве *m* нельзя брать степень двойки, так как если $m = 2^p$, то *hash*(*key*) – это просто *p* младших битов числа *key*.

Методы построения хеш-функций

- Построение хеш-функции ***методом умножения***.
  - Пусть количество хеш-значений равно m.
    Выберем и зафиксируем вещественную константу
    v, $0 < v < 1$;
    положим $\boldsymbol{hash(key) = \lfloor m(frac(key \cdot v)) \rfloor}$
    frac(key$\cdot v$) – дробная часть числа $key \cdot v$.
  - Достоинство метода умножения в том, что качество хеш-функции слабо зависит от выбора m. Обычно в качестве m выбирают степень двойки, так как в этом случае умножение на m сводится к сдвигу.
  - ***Пример***. Пусть в используемом компьютере длина слова равна w битам и ключ key помещается в одно слово.
  - Если $m = 2^p$, то вычисление $hash(key)$ можно выполнить следующим образом: умножим key на w-битовое целое число $v \cdot 2^w$; получится $2w$-битовое число $r_1 r_0$.

    В качестве значения $hash(key)$ возьмем старшие p битов r_0 (отбрасывание младших $w-p$ разрядов).
  - Согласно Д. Кнуту выбор $v = \left(\sqrt{5} - 1\right)/2 = 0.6180339887...$ является удачным.