Алгоритмы и структуры данных

Лекция 3. Таблицы и деревья.

(c) Глухих Михаил Игоревич, glukhikh@mail.ru

Таблица

- Элемент = Ключ + Данные
- Ключи у разных элементов не совпадают
- Операции
  - Search (Key)
  - Insert (Key, Value)
  - Remove (Key)

Виды таблиц

- С прямой адресацией

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован
- Ключи не слишком большие

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован
- Ключи не слишком большие

- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства, Недостатки?

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован
- Ключи не слишком большие

- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства: трудоёмкость всюду O(1)
- Недостатки?

Таблицы с прямой адресацией

- Ключ = Целое неотрицательное число или может быть к нему преобразован
- Ключи не слишком большие

- => Используем ключ как индекс массива, а данные храним в этом массиве
- Достоинства: трудоёмкость всюду O(1)
- Недостатки: ресурсоёмкость O(MaxKey)

Виды таблиц

- С прямой адресацией
- Хэш-таблицы
  - Используем для индексации hash(key) или hash(key) % 2^N

Виды таблиц

- С прямой адресацией
- Хэш-таблицы
  - Используем для индексации hash(key) или hash(key) % 2^N
  - Можем управлять размером нашего массива
  - НО! Коллизии

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
  - С помощью цепочек
  - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
  - Достоинства = Размерность исходного массива ограничивается
  - Недостатки = ?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
  - С помощью цепочек
  - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
  - Достоинства = Размерность исходного массива ограничивается
  - Недостатки = При фиксированной размерности массива трудоёмкость операций O(N) -- в общем случае O(1 + N / m), где m – размерность массива
  - Как с этим бороться?

Виды таблиц

- С прямой адресацией
- Хэш-таблицы: как бороться с коллизиями?
  - С помощью цепочек
  - Каждому значению кода ставится в соответствие не одна пара Ключ + Значение, а несколько, объединённых в связанный список
  - Достоинства = Размерность исходного массива ограничивается
  - Недостатки = При фиксированной размерности массива трудоёмкость операций O(N) -- в общем случае O(1 + N / m), где m – размерность массива
  - Как с этим бороться? Понятно как – изменять размерность массива динамически

Таблица с открытой адресацией

- Нет никаких цепочек и связанных списков
- Всё хранится в едином массиве
- Если есть возможность, хэш-код (по модулю) используем как индекс в массиве
- Если возникает коллизия, то сдвигаем

HASH-INSERT(array[m], key)

- i = hash(key) % m
- start = i
- do:
  - if (array[i] == null):
    - array[i] = key
    - return i
  - i = (i + 1) % m
- while (i != start)
- Error(“Overflow”)

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

NN NN NN NN NN NN NN NN

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

```
NN  NN  NN  NN  NN  NN  NN  NN
NN  NN  2A  NN  NN  5B  NN  7C
```

HASH-INSERT: пример

- Используем 3 бита хэша и массив из 8 элементов
- В качестве ключа используем символ

```
NN NN NN NN NN NN NN NN
NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C
```

HASH-SEARCH(array[m], key)

- i = hash(key) % m
- start = i
- do:
  - if (array[i] == key):
    - return i
  - if (array[i] == null):
    - return null
  - i = (i + 1) % m
- while (i != start)
- return null

HASH-DELETE

- Реализация «в лоб» не проходит

NN NN NN NN NN NN NN NN

NN NN 2A NN NN 5B NN 7C

NN NN 2A 2D NN 5B NN 7C

HASH-DELETE

- Реализация «в лоб» не проходит

```
NN NN NN NN NN NN NN NN
NN NN 2A NN NN 5B NN 7C
NN NN 2A 2D NN 5B NN 7C
// Теперь стираем 2A
NN NN NN 2D NN 5B NN 7C
```

HASH-DELETE

- Реализация «в лоб» не проходит

```
NN NN NN NN NN NN NN NN
NN NN 2A NN NN 5B NN 7C
NN NN 2A 2D NN 5B NN 7C
// Теперь стираем 2A
NN NN NN 2D NN 5B NN 7C
// И ищем 2D – NOT FOUND!!!
```

HASH-DELETE

- А как «не в лоб»? Отдельный вариант “DD” -- DELETED

```
NN NN NN NN NN NN NN NN
NN NN 2A NN NN 5B NN 7C
NN NN 2A 2D NN 5B NN 7C
// Теперь стираем 2A
NN NN DD 2D NN 5B NN 7C
```

HASH-DELETE

- А как «не в лоб»?
  Отдельный вариант “DD” – DELETED
- И при поиске DD не считается null

```
NN NN NN NN NN NN NN NN
NN NN 2A NN NN 5B NN 7C
NN NN 2A 2D NN 5B NN 7C
// Теперь стираем 2A
NN NN DD 2D NN 5B NN 7C
```

Таблица с открытой адресацией

- Трудоёмкость операций зависит от коэффициента заполнения A = n / m (n = число элементов, m = размерность масссива)
- И равна ~ 1 / (1 – A) для большинства операций

Таблица с открытой адресацией

- Трудоёмкость операций зависит от коэффициента заполнения A = n / m (n = число элементов, m = размерность масссива)
- И равна ~ 1 / (1 – A) для большинства операций

- Например, при таблице, заполненной на четверть, мы имеем трудоёмкость 4 / 3

Таблица с открытой адресацией

- Трудоёмкость операций зависит от коэффициента заполнения A = n / m (n = число элементов, m = размерность масссива)
- И равна ~ 1 / (1 – A) для большинства операций

- Например, при таблице, заполненной на четверть, мы имеем трудоёмкость 4 / 3
- А наполовину – уже 2

Таблица с открытой адресацией

- Трудоёмкость операций зависит от коэффициента заполнения $A = n / m$ (n = число элементов, m = размерность масссива)
- И равна ~ $1 / (1 - A)$ для большинства операций

- Например, при таблице, заполненной на четверть, мы имеем трудоёмкость $4 / 3$
- А наполовину – уже 2
- А на три четверти – уже 4
- И наконец, если заполнены все элементы, кроме одного, то требуется m операций

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet
    - Как сравниваем?

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet
    - Comparable OR Comparator

Бинарные деревья поиска

- Элемент = Ключ + Данные
- Ключи у разных элементов не совпадают и поддерживают отношение полного порядка

Отношение полного порядка

- Total ordering
  - <=
  - a <= b AND b <= a <==> a == b
  - a <= b AND b <= c ==> a <= c
  - a <= b OR b <= a <==> ALWAYS

Отношение полного порядка

- Total ordering
  - <=
  - a <= b AND b <= a <==> a == b
  - a <= b AND b <= c <==> a <= c
  - a <= b OR b <= a

  - <
  - a < b AND b < a <==> NEVER
  - a < b AND b < c ==> a < c
  - a < b OR b < a OR a == b <==> ALWAYS

Бинарные деревья поиска

- Элемент = Ключ + Данные
- Ключи у разных элементов не совпадают и поддерживают отношение порядка
- Операции
  - Search (Key)
  - Insert (Key, Value)
  - Remove (Key)
- Дополнительные операции
  - Maximum() / Minimum()

Бинарные деревья поиска

- Элемент = Ключ + Данные
- Ключи у разных элементов не совпадают и поддерживают отношение порядка
- Операции
  - Search (Key)
  - Insert (Key, Value)
  - Remove (Key)
- Дополнительные операции
  - Maximum() / Minimum()
  - Successor(Key) / Predecessor(Key)

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()
    - headSet() / tailSet() / subSet()

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()
    - headSet() / tailSet() / subSet()
  - NavigableSet extends SortedSet
    - higher() / lower() / floor() / ceiling()

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()
    - headSet() / tailSet() / subSet()
  - NavigableSet extends SortedSet
    - higher() / lower() / floor() / ceiling()
    - descendingSet() / descendingIterator()

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()
    - headSet() / tailSet() / subSet()
  - NavigableSet extends SortedSet
    - higher() / lower() / floor() / ceiling()
    - descendingSet() / descendingIterator()
    - pollFirst() / pollLast()

Бинарные деревья поиска

- Java-интерфейсы
  - SortedSet extends Set
    - Как сравниваем = Comparator / Comparable
    - first() / last()
    - headSet() / tailSet() / subSet()
  - NavigableSet extends SortedSet
    - higher() / lower() / floor() / ceiling()
    - descendingSet() / descendingIterator()
    - pollFirst() / pollLast()
- Java-классы
  - TreeSet = на основе красно-чёрного дерева

Бинарные деревья поиска

- Варианты реализации
  A. Листья пустые, внутренние узлы содержат ключи и ссылки (ненулевые) ровно на два потомка
  B. Все узлы содержат ключи и ссылки (возможно, нулевые) на два возможных потомка

Красно-чёрные деревья

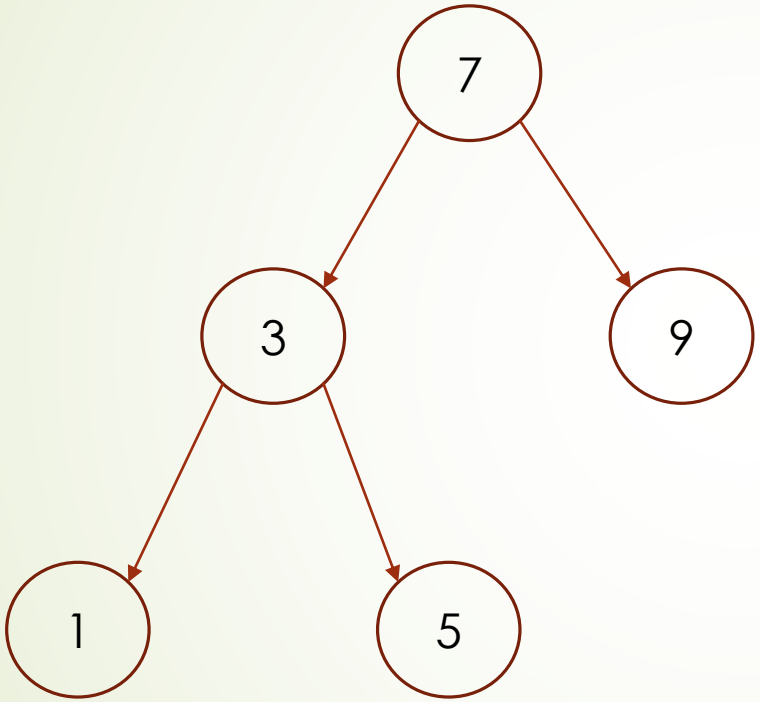
- Свойства
  - Узел = Чёрный ИЛИ Красный
  - Корневой Узел = Чёрный
  - (Нулевые Узлы = Чёрные) (их может и не быть)
  - Потомки Красного Узла = Чёрные
  - Число Чёрных узлов на любом пути вниз = Одинаково (M)

Красно-чёрные деревья

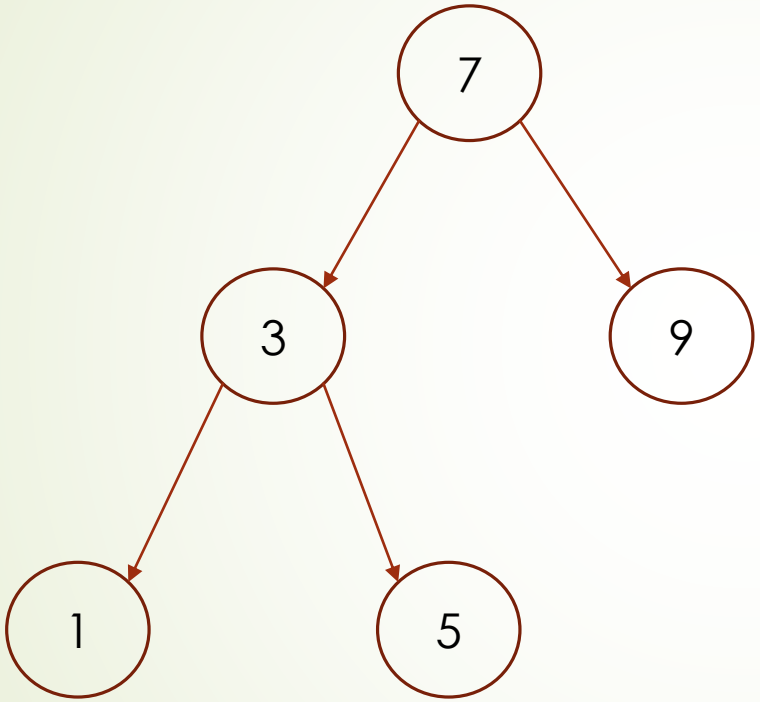
- Свойства
  - Узел = Чёрный ИЛИ Красный
  - Корневой Узел = Чёрный
  - (Нулевые Узлы = Чёрные)
  - Потомки Красного Узла = Чёрные
  - Число Чёрных узлов на любом пути вниз = Одинаково (M)

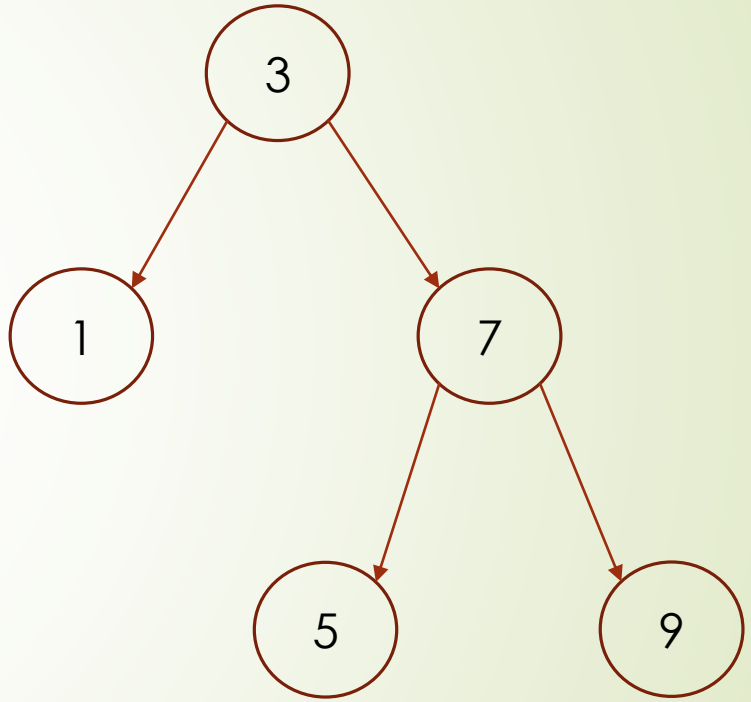
  - Отсюда: число Красных узлов на любом пути вниз < M
  - Отсюда: число ВСЕХ узлов на любом пути вниз < 2M – 1

Операция с деревом – поворот



Операция с деревом – поворот





Операция с деревом – вставка

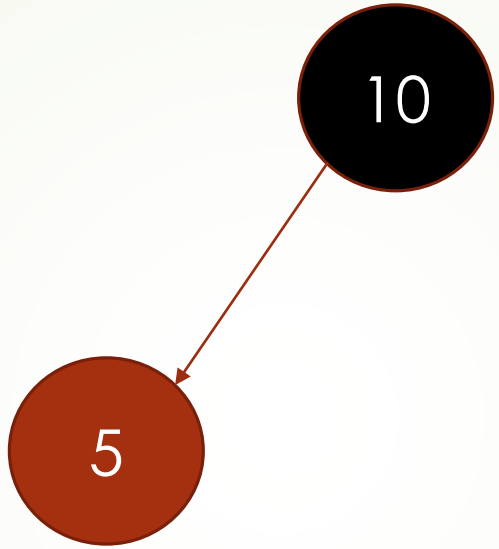
- Вставляем КРАСНЫЙ узел
- Если это первый узел – перекрашиваем в ЧЁРНЫЙ

Пример

Операция с деревом – вставка

- Вставляем КРАСНЫЙ узел
- Если это первый узел – перекрашиваем в ЧЁРНЫЙ
- Если родитель узла ЧЁРНЫЙ – ОК

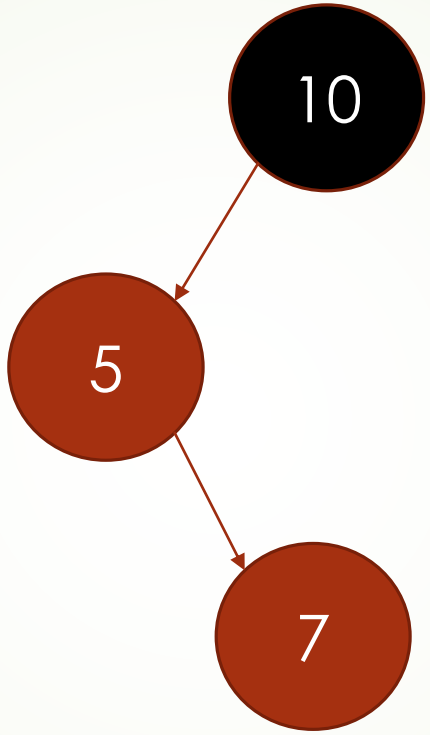
Пример



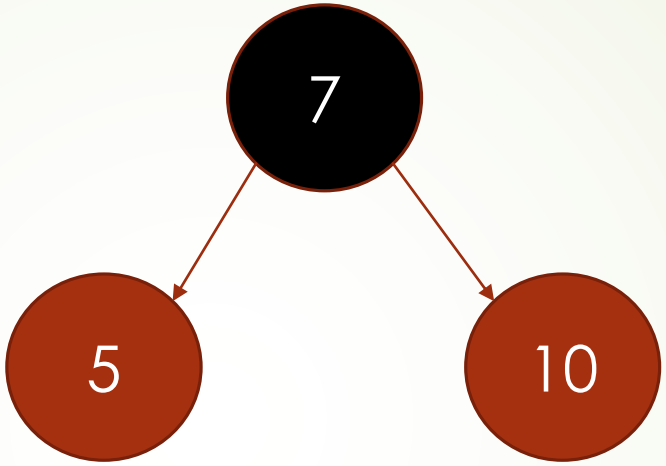
Операция с деревом – вставка

- Вставляем КРАСНЫЙ узел
- Если это первый узел – перекрашиваем в ЧЁРНЫЙ
- Если родитель узла ЧЁРНЫЙ – ОК
- Если родитель и дядя – КРАСНЫЕ, делаем их ЧЁРНЫМИ, а дедушку – КРАСНЫМ

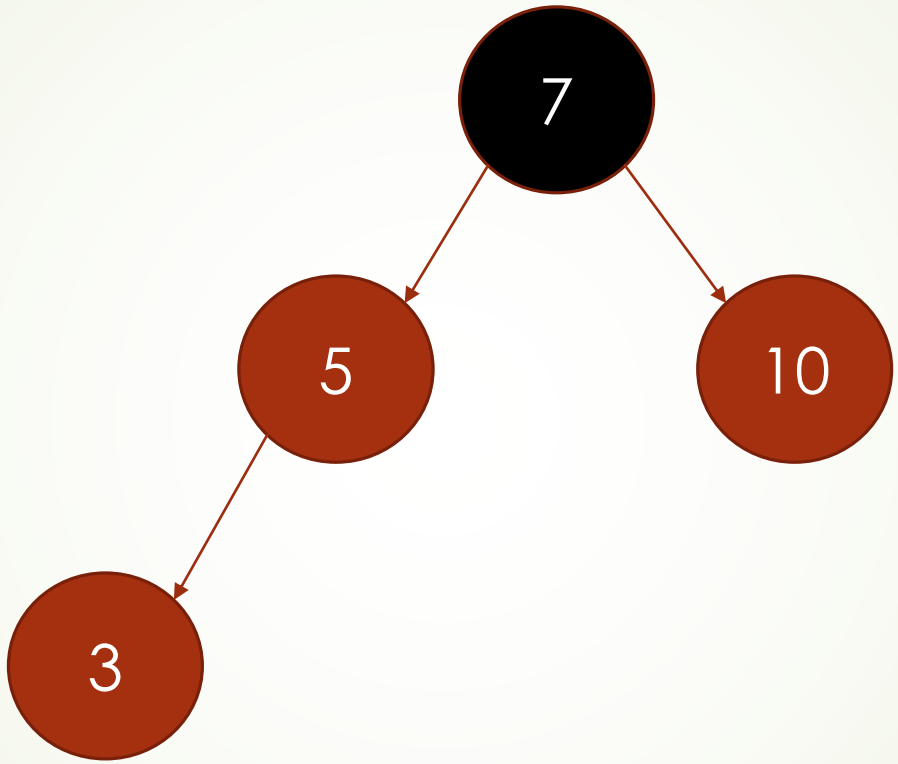
Пример



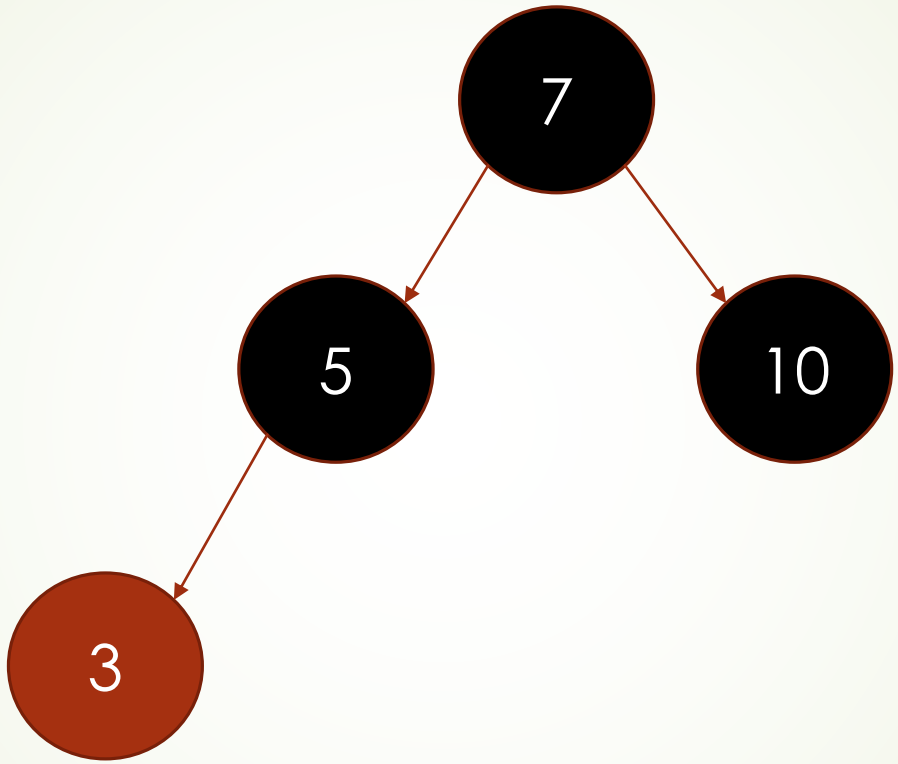
Пример



Пример



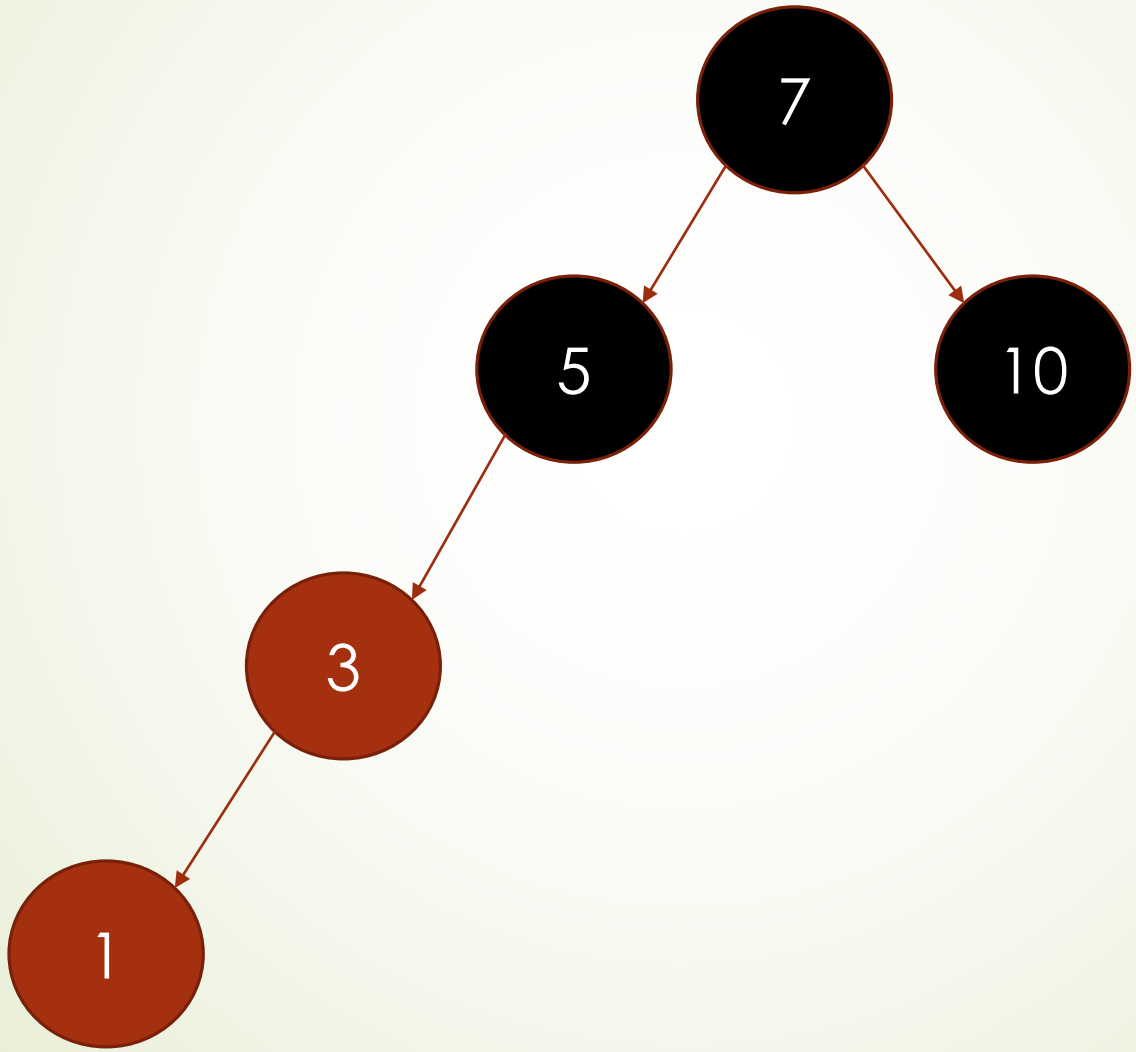
Пример



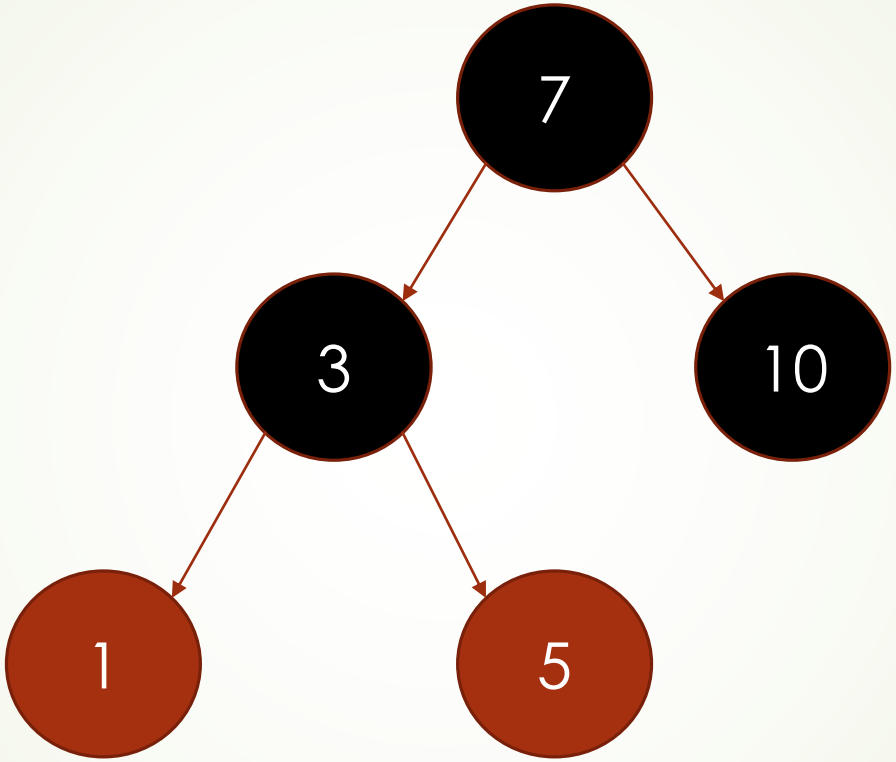
Операция с деревом – вставка

- Вставляем КРАСНЫЙ узел
- Если это первый узел – перекрашиваем в ЧЁРНЫЙ
- Если родитель узла ЧЁРНЫЙ – ОК
- Если родитель и дядя – КРАСНЫЕ, делаем их ЧЁРНЫМИ, а дедушку – КРАСНЫМ
- Родитель – КРАСНЫЙ, дядя – ЧЁРНЫЙ -- повороты

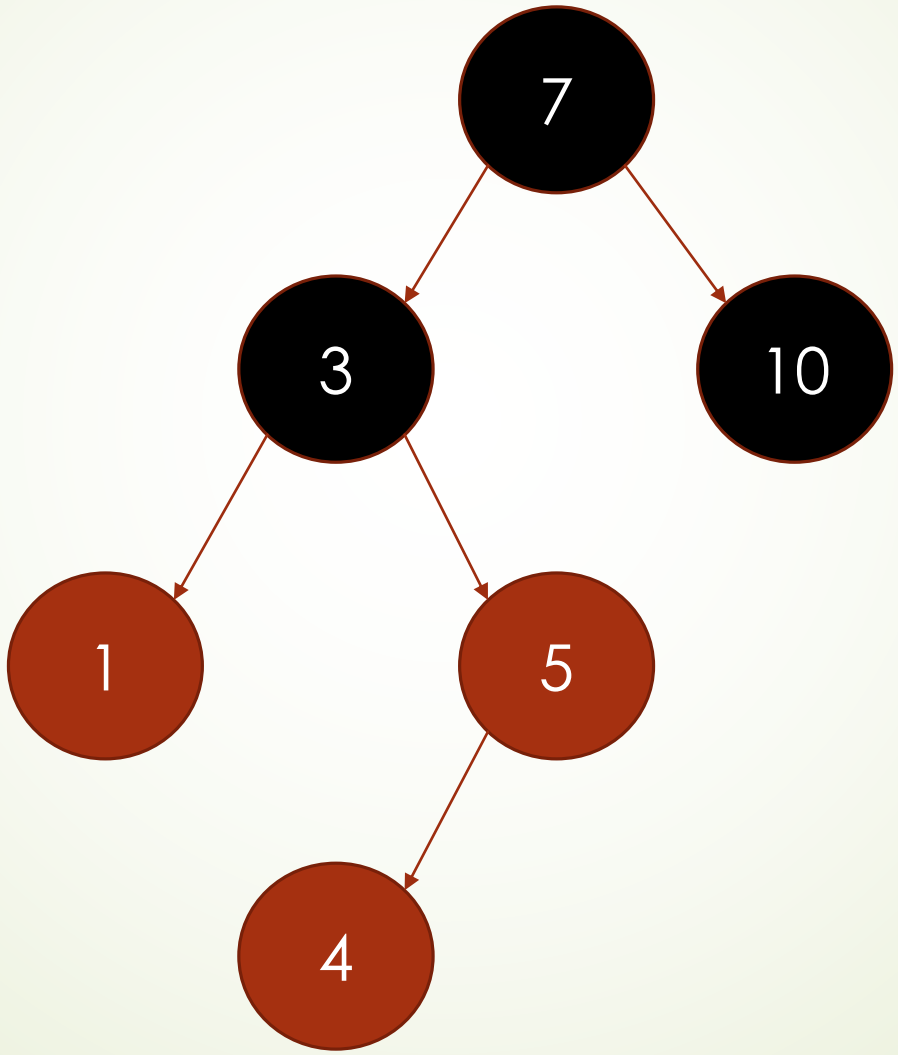
Пример



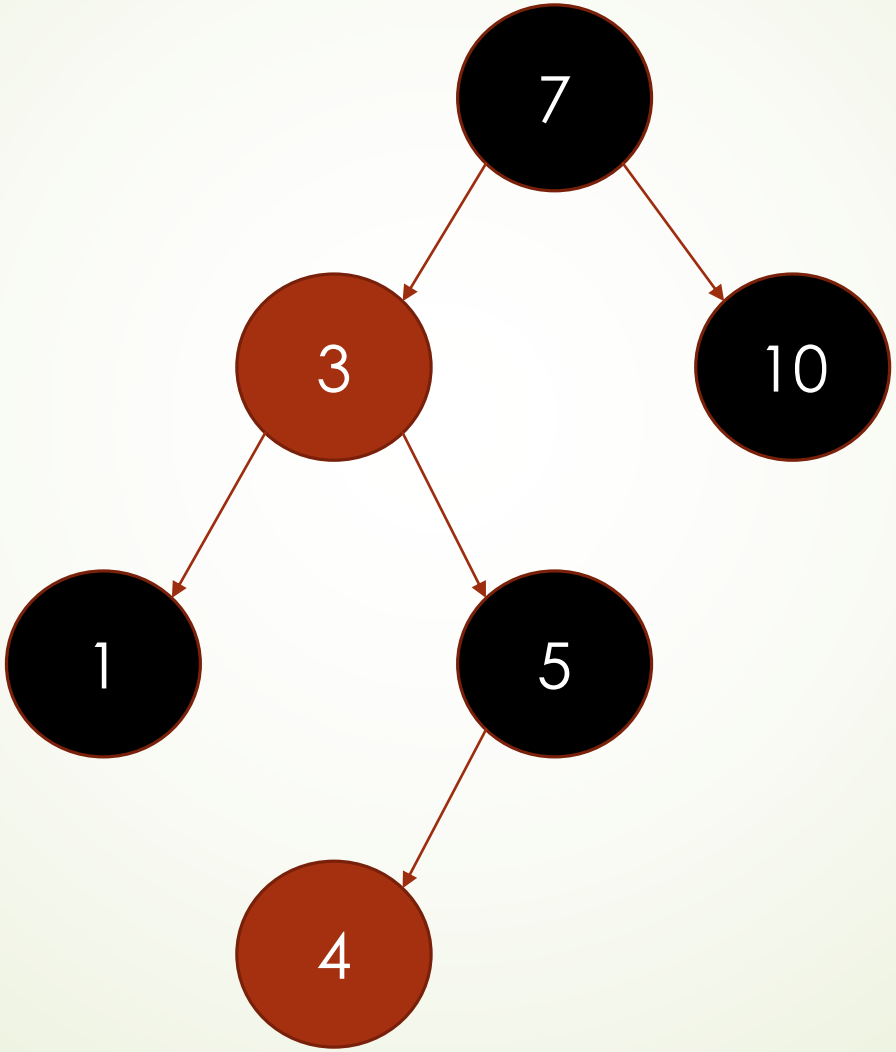
Пример



Пример



Пример



Красно-чёрные деревья

- Визуализация: https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

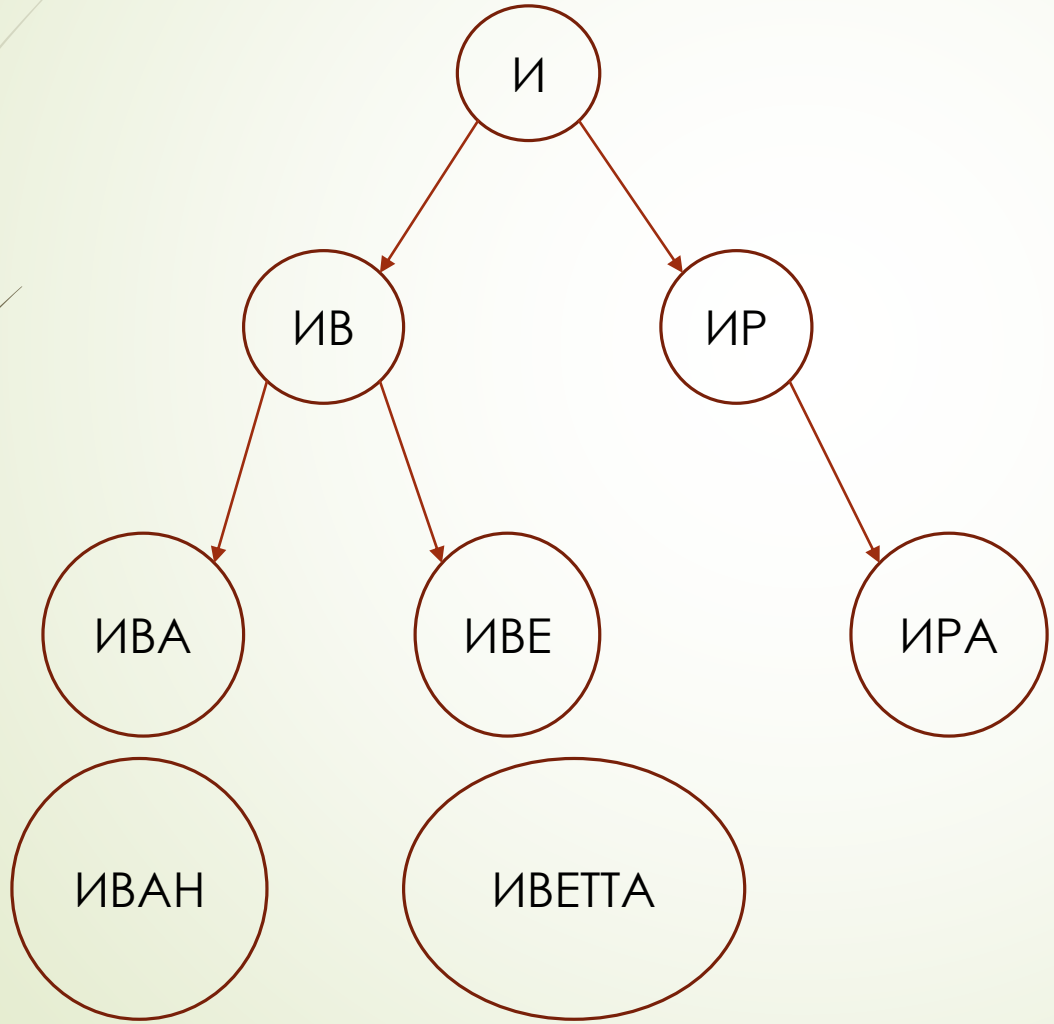
Возможный интерфейс красно-чёрного узла

```
public interface RBNode<T> {
    T getValue();
    default boolean isRed() {
        return true;
    }
    RBNode<T> getLeft();
    RBNode<T> getRight();
}
```

Другие сбалансированные деревья

- АВЛ-дерево
  - Высоты разных ветвей отличаются максимум на 1
- B-дерево
  - Каждая вершина содержит список узлов, и на одного потомка больше, чем есть в этом списке

Trie (префиксное дерево)



Trie (применение)

- Префиксный поиск по словарю
- Поиск по строкам / спискам
- Трудоёмкость операций: O(maxLength)
- См. Класс Trie в обучающем проекте

Итоги

- Рассмотрено
  - Таблицы и хэш-таблицы
    - На основе цепочек
    - С открытой адресацией
  - Бинарные деревья поиска
    - Красно-чёрные
    - АВЛ, B-деревья
  - Префиксные деревья
- Далее
  - Графовые алгоритмы

Домашнее задание (вариант А)

- Файл BinaryTree / KtBinaryTree в обучающем проекте
- Реализовать, на выбор, **два** метода из следующих
  - BinaryTree.remove
  - BinaryTree.headSet
  - BinaryTree.tailSet
  - BinaryTree.subset
  - Iterator.findNext
  - Iterator.remove
- Если сделан один метод – оценивается из «4»
- PR на GitHub, либо просто ссылка на свою копию
- Не забудьте про тесты
- Дедлайн: **12 ноября 23:59**

Домашнее задание (вариант Б)

- Может быть выполнено вместо предыдущего
- В классе Trie в обучающем проекте добавить реализацию метода iterator()
- Условия те же: тесты, дедлайн, PR / своя копия