Адрианов Н.М.
Иванов А.Б.

Алгоритмы и структуры данных

Абстрактные типы данных
Ассоциативные массивы

Абстрактные типы данных (АТД, ADT)

Стек, очередь, дек, список...

Абстрактные типы данных (АТД, ADT)

Стек, очередь, дек, список...

Абстрактный тип данных = интерфейс

```
interface Stack {
    void push(int value);
    int pop();
    int peek();
    int count();
    boolean isEmpty();
    void clear();
}
```

Реализация стека

Связанный список

Реализация стека

На основе массива

Стандартные коллекции в Java

Интерфейс:

List<T>

```
public interface List<E> extends Collection<E> {
  int size();
  boolean isEmpty();
  boolean add(E e);
  E get(int index);
  E set(int index, E element);
  void add(int index, E element);
  E remove(int index);
  void clear();
  int indexOf(Object o);
  int lastIndexOf(Object o);
  Iterator<E> iterator();
  ...
}
```

Стандартные коллекции в Java

Классы (реализующие интерфейс):

ArrayList<T>, LinkedList<T>

```
List<String> list = new ArrayList<String>();
```

Ассоциативный массив (Map, Dictionary)

Абстрактный тип данных:

набор пар <ключ, значение> (ключ уникальный).

Операции (интерфейс):

- Вставка
- Поиск по ключу
- Удаление по ключу

```
interface Map<K, V> {
    ...
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    ...
}
```

Реализация ассоциативного массива

Реализация ассоциативного массива

- Массив
- Отсортированный массив
- Бинарные деревья поиска
- Хеш-таблицы

C#: интерфейс IDictionary

```
public interface IDictionary<TKey, TValue> :
     ICollection<KeyValuePair<TKey, TValue>>,
     IEnumerable<KeyValuePair<TKey, TValue>>
{
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }

    TValue this[TKey key] { get; set; }
    void Add(TKey key, TValue value);
    bool ContainsKey(TKey key);
    bool Remove(TKey key);
    bool TryGetValue(TKey key, out TValue value);
}

public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
    int Count { get; }
    void Clear();
    ...
}
```

Java: интерфейс Map

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);
    V put(K key, V value);
    V remove(Object key);
    void clear();

    Set<K> keySet();
    Collection<V> values();
    Set<Map.Entry<K, V>> entrySet();

    interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
}
```

Java: интерфейс Set

```
public interface Set<E> implements Iterable<E> {
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    V get(Object key);
    boolean add(E e);
    V remove(Object o);
    void clear();
    ...
}
```

Стандартные классы в Java и C#

Java: `HashSet<T>`, `HashMap<K,V>`
`TreeSet<T>`, `TreeMap<K,V>`

```
public abstract class AbstractSet<E> implements Set<E>
public class HashSet<E> extends AbstractSet<E>
public class TreeSet<E> extends AbstractSet<E>

public abstract class AbstractMap<K,V> implements Map<K,V>
public class HashMap<K,V> extends AbstractMap<K,V>
public class TreeMap<K,V> extends AbstractMap<K,V>
```

C#: `HashSet<T>`, `Dictionary<K,V>`
`SortedSet<T>`, `SortedDictionary<K,V>`

Подсчет количества слов

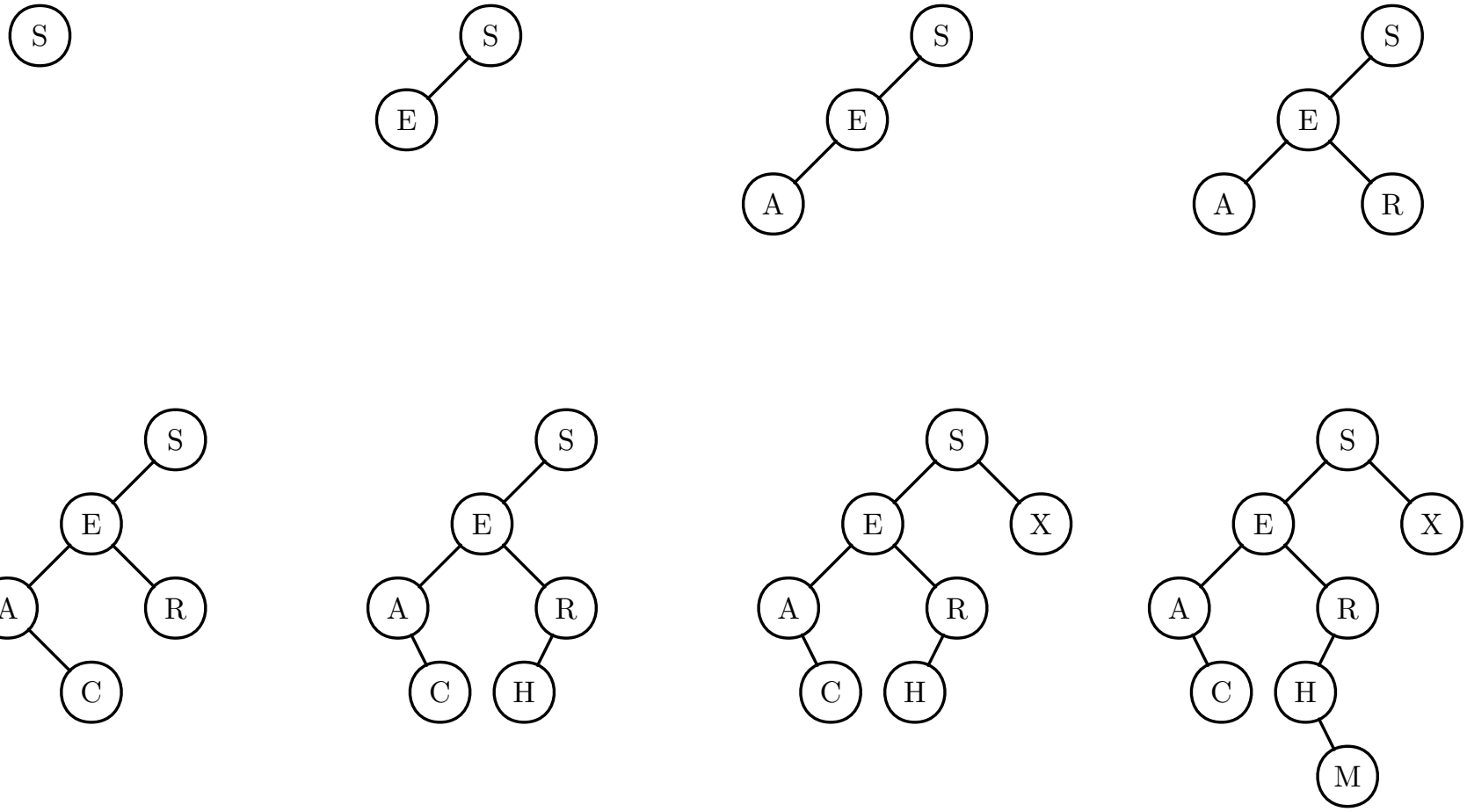
```
private static void CountWords()
{
  var dic = new Dictionary<string, int>();
  while (true)
  {
    string line = Console.ReadLine();
    if (string.IsNullOrEmpty(line))
      break;

    string[] words = line.Split(' ');
    foreach (string word in words)
    {
      if (dic.ContainsKey(word))
        dic[word] = dic[word] + 1;
      else
        dic[word] = 1;
    }
  }

  foreach (var kvp in dic)
    Console.WriteLine("{0} = {1}", kvp.Key, kvp.Value);
}
```

Binary Search Tree: пример

S, E, A, R, C, H, X, M

Binary Search Tree: пример

S, E, A, R, C, H, X, M



Реализация ("Algorithms in Java")

```
public class BST<Key extends Comparable<Key>, Value>
{
    private class Node
    {
        private Key key; // key
        private Value val; // associated value
        private Node left, right; // links to subtrees
        private int N; // # nodes in subtree rooted here

        public Node(Key key, Value val, int N) {
            this.key = key; this.val = val; this.N = N;
        }
    }

    private Node root; // root of BST

    public int size() { return size(root); }
    private int size(Node x) { return x == null ? 0 : x.N; }

    public Value get(Key key) { ... }
    public void put(Key key, Value val) { ... }
}
```

Binary Search Tree

Бинарные деревья поиска используют сравнения

```
interface Comparable<T> {
    int compareTo(T o);
}

public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Java: `Comparable`, `Comparator`

C#: `IComparable`, `IComparer`

Реализация: get

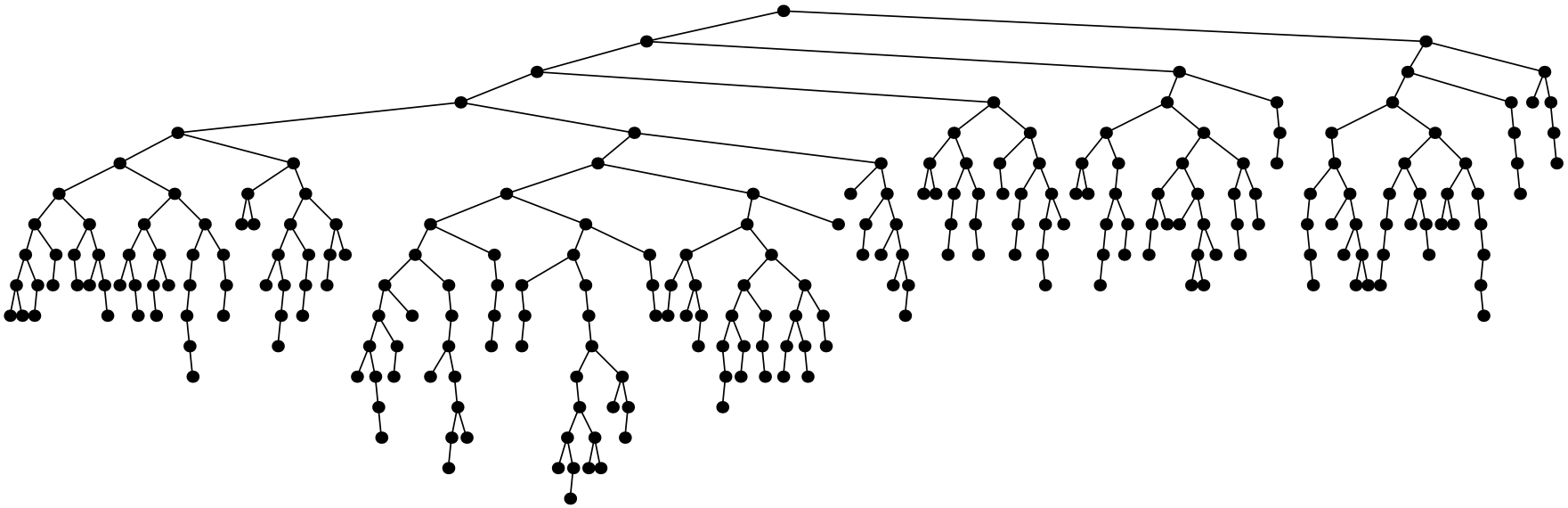
```
public Value get(Key key) {
    return get(root, key);
}

private Value get(Node x, Key key) {
    // Return value associated with key in the subtree rooted at x;
    // return null if key not present in subtree rooted at x.
    if (x == null) return null;

    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```

Реализация: put

```
public void put(Key key, Value val) {
    // Search for key. Update value if found; grow table if new.
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value val) {
    // Change key's value to val if key in subtree rooted at x.
    // Otherwise, add new node to subtree associating key with val.
    if (x == null) return new Node(key, val, 1);

    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;

    x.N = size(x.left) + size(x.right) + 1;
    return x;
}
```

Симуляция 1 ($N = 255$)

$\max = 18 \qquad \text{avg} = 9.31$

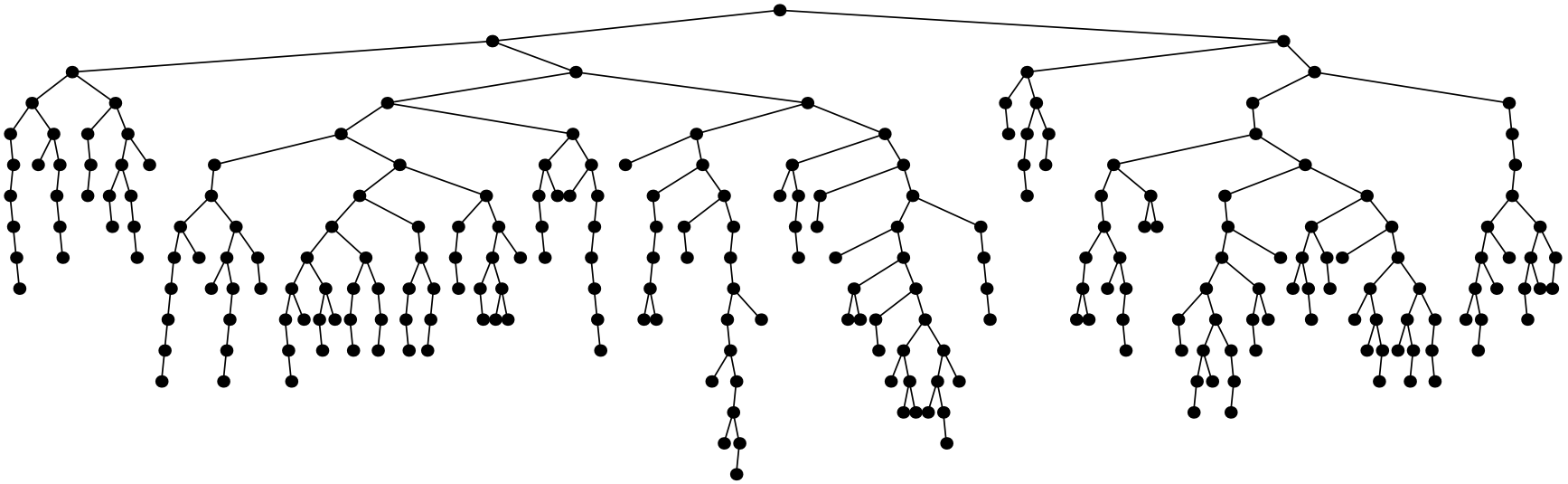


Полностью сбалансированное дерево:

$\max = 8 \qquad \text{avg} = 7.03$

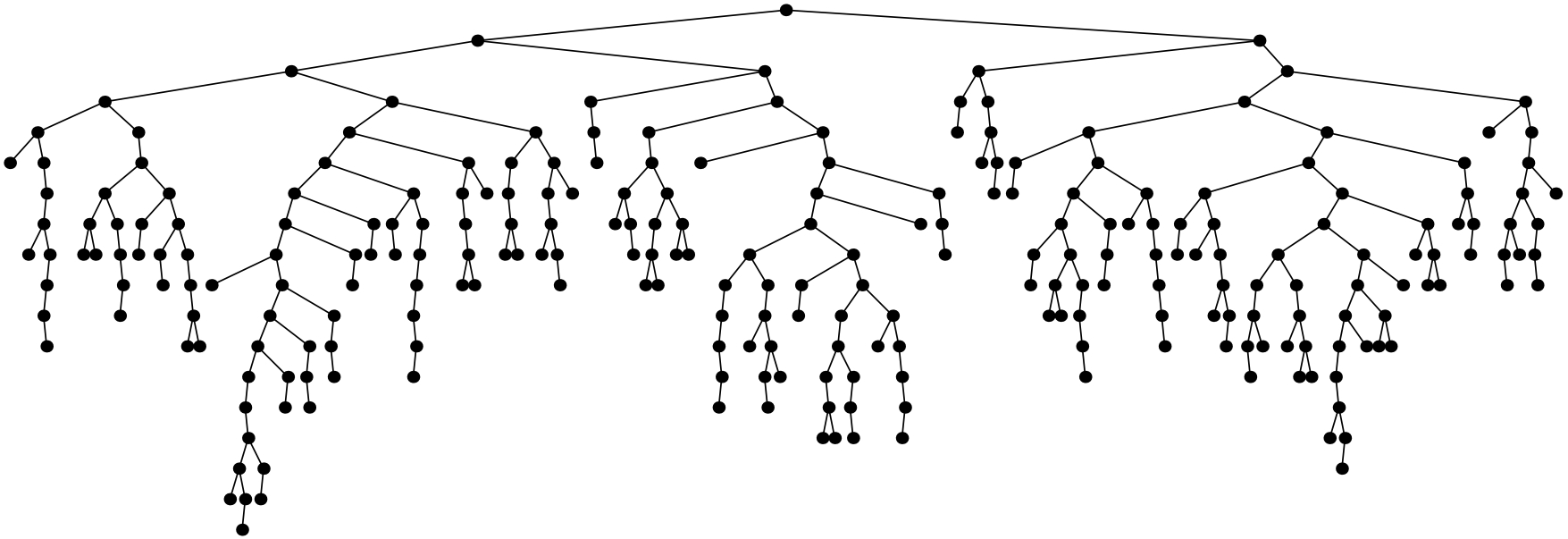
Симуляция 2 ($N = 255$)

$\max = 17 \qquad \text{avg} = 9.13$



Симуляция 3 ($N = 255$)

$\max = 16 \qquad \text{avg} = 9.07$



Сложность в случайном BST

Утв. Search hit в случайном бинарном дереве поиска, содержащем N ключей требует в среднем $\sim 2 \ln N$ ($\approx 1.39 \log N$) сравнений.

Пусть C_N – сумма длин путей до всех узлов в дереве с N узлами (тогда успешный поиск требует в среднем $1 + C_N/N$).

$$C_N = N-1+(C_0+C_{N-1})/N+(C_1+C_{N-2})/N+\ldots+(C_{N-1}+C_0)/N$$

$$C_0 = C_1 = 0$$

(рекурентное соотношение аналогично тому, которое возникало при анализе Quicksort, см. лекцию 3).

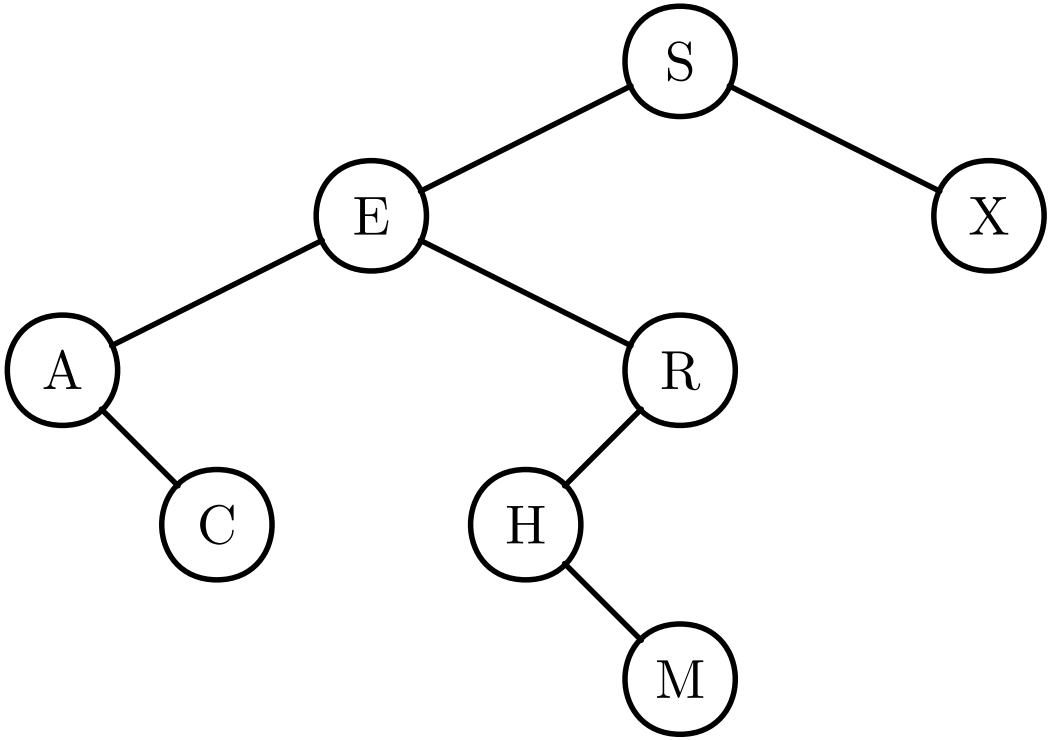
$$C_N \sim 2N \ln N$$

Сложность в случайном BST

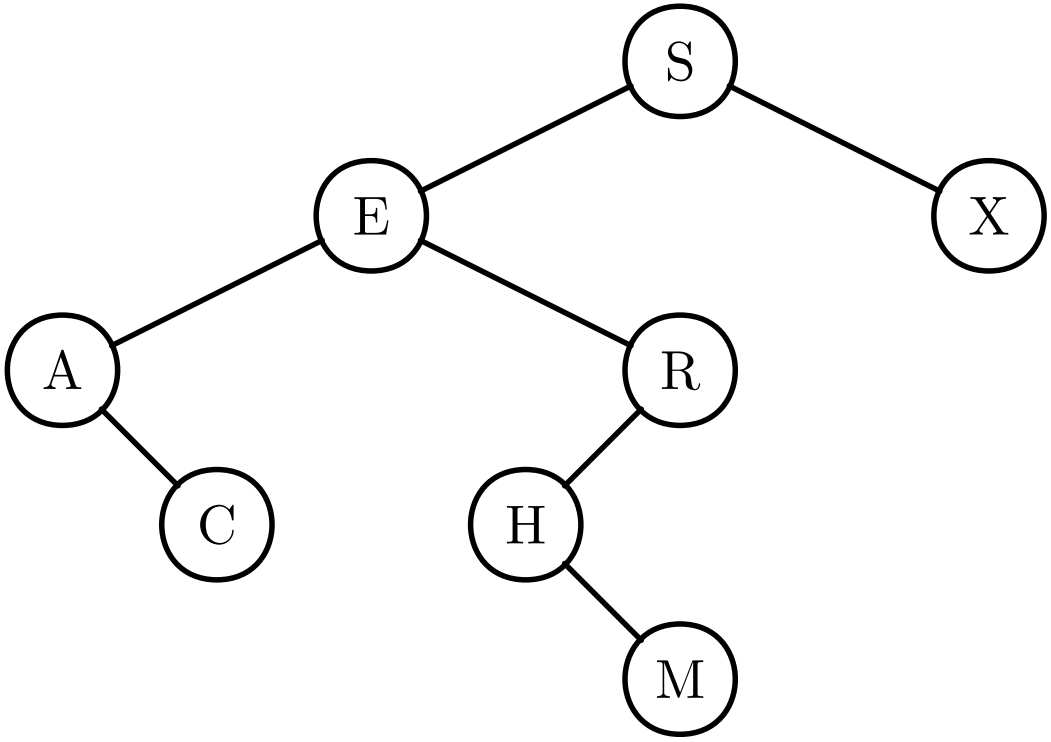
Утв. Вставка и search miss в случайном бинарном дереве поиска, содержащем N ключей требует в среднем $\sim 2 \ln N$ ($\approx 1.39 \log N$) сравнений.

Вставка и search miss требует на 1 сравнение больше, чем search hit (задача).

Удаление в BST



Удаление в BST

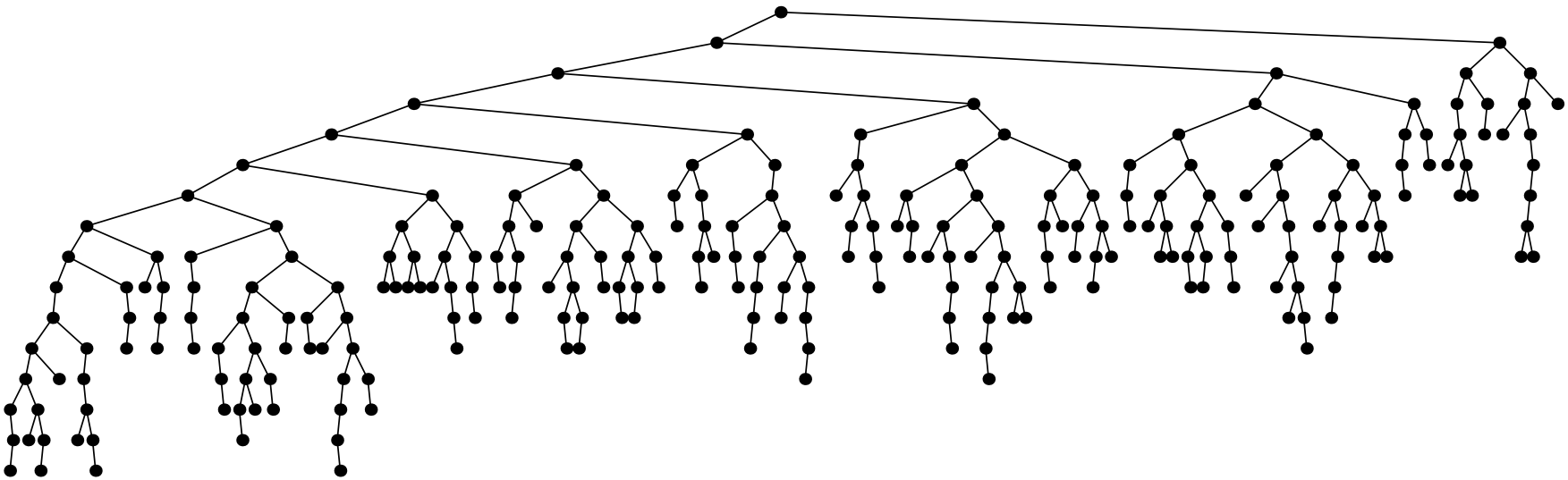


- Tombstone
- Hibbard deletion

Симуляция с использованием удалений

- Добавляем N случайных ключей.
- Цикл N^2 раз:
  - удаляем случайный ключ;
  - добавляем случайный ключ.

$N = 255 \qquad \max = 16 \qquad \text{avg} = 9.14$



Симуляция с использованием удалений

$N = 8191 \qquad \max = 78 \qquad \text{avg} = 34.8$

$N = 16383 \qquad \max = 102 \qquad \text{avg} = 42.2$

$N = 32767 \qquad \max = 147 \qquad \text{avg} = 57.9$

Факт: высота дерева при такой симуляции $\sim \sqrt{N}$.

Сбалансированные деревья

Идеально сбалансированное дерево = все пути от корня до конечных вершин имеют одинаковую длину.

Хочется, чтобы min/max длины отличались не сильно.

- AVL (Г.М. Адельсон-Вельский, Е.М. Ландис, 1962)

  Для любого узла разница между высотами левого и правого поддерева не превосходит 1

- Красно-черные (Rudolf Bayer, 1972)
  Термин: Leonidas J. Guibas, Robert Sedgewick, 1978

  Количество черных ребер в любом пути одинаково,
  количество красных ребер в пути не превосходит количество черных + 1

$\Rightarrow$ гарантированная сложность $O(\log N)$.