CS 600.226: Data Structures

Michael Schatz

Nov 2, 2018

Lecture 27. Treaps

HW6

Assignment 6: Setting Priorities

Out on: October 26, 2018

Due by: November 2, 2018 before 10:00 pm

Collaboration: None

Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

The sixth assignment is all about sets, priority queues, and various forms of experimental analysis aka benchmarking. You'll work a lot with jaybee as well as with new incarnations of the old Unique program. Think of the former as "unit benchmarking" the individual operations of a data structure, think of the latter as "system benchmarking" a complete (albeit small) application.

HW7

Assignment 7: Whispering Trees

Out on: November 2, 2018
Due by: November 9, 2018 before 10:00 pm
Collaboration: None
Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

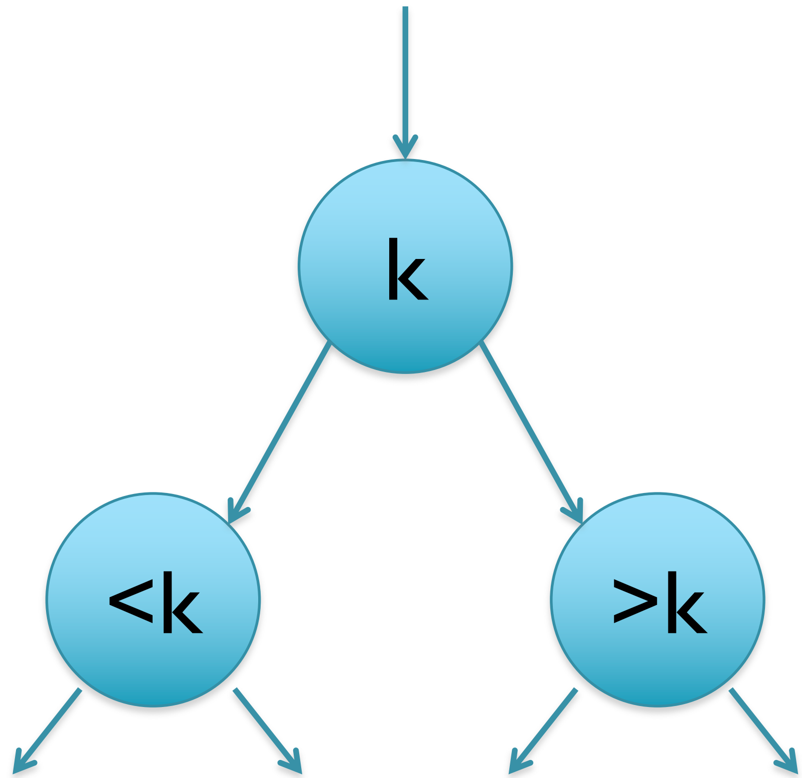
The seventh assignment is all about ordered maps, specifically fast ordered maps in the form of balanced binary search trees. You'll work with a little program called Words that reads text from standard input and uses an (ordered) map to count how often different words appear. We're giving you a basic (unbalanced) binary search tree implementation of OrderedMap that you can use to play around with the Words program and as starter code for your own developments.

Agenda

1. ***Recap on BSTs and AVL Trees***
2. ***Treaps***

Part 1: Binary Search Tree

Binary Search Tree



A BST is a binary tree with a special ordering property:
If a node has value k, then the left child (and its descendants) will have values smaller than k; and the right child (and its descendants) will have values greater than k

Can a BST have duplicate values?

Searching

```
    6
   / \
  4   8
 /   / \
1   7   9
```

has(7):	compare 6 => compare 8 => found 7
has (2):	compare 6 => compare 4 => compare 1 => not found!

What is the runtime for has() ?

Constructing

```
        i(7)      i(5)      i(8)            i(1)            i(3)            i(4)            i(9)
empty   --->  7   --->  7   --->  7         --->      7     --->      7     --->      7     --->      7
                       /         / \                 / \             / \             / \             / \
                      5         5   8               5   8           5   8           5   8           5   8
                                                   /               /               /               /     \
                                                  1               1               1               1       9
                                                                   \               \               \
                                                                    3               3               3
                                                                                     \               \
                                                                                      4               4
```

Note the shape of a general BST will depend on the order of insertions

What is the "worst" order for constructing the BST?

What is the "best" order for constructing the BST?

What happens for a random ordering?

Removing

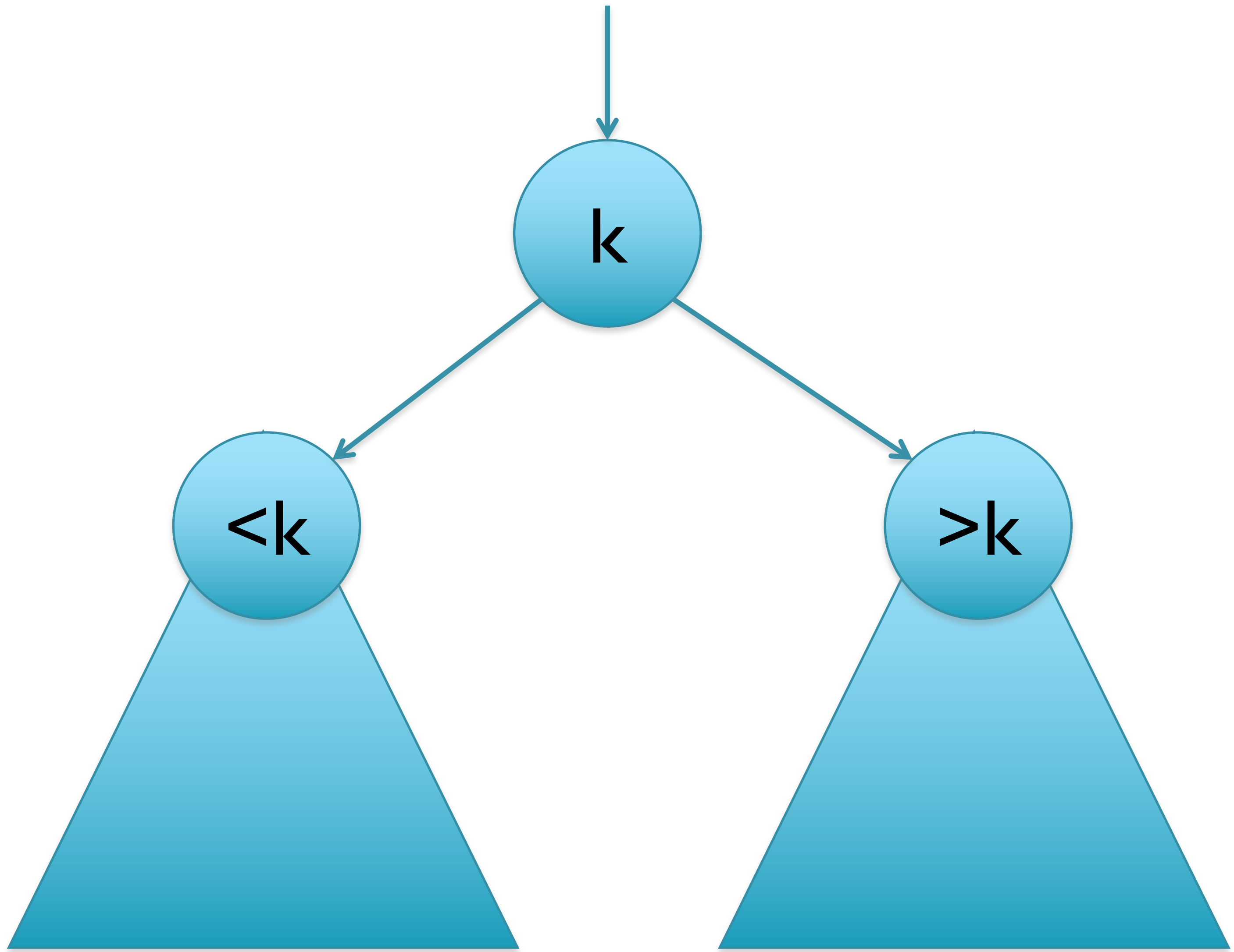
```
      6          r(7)       6          r(8)       6          r(8)
    /   \        --->     /   \        --->     /   \        --->
   3     8               3     8               3     9
  / \   / \             / \     \             / \
 1   4 7   9           1   4     9           1   4
```

Remove leaf is easy

Remove with only one child Is easy

Remove internal node ?

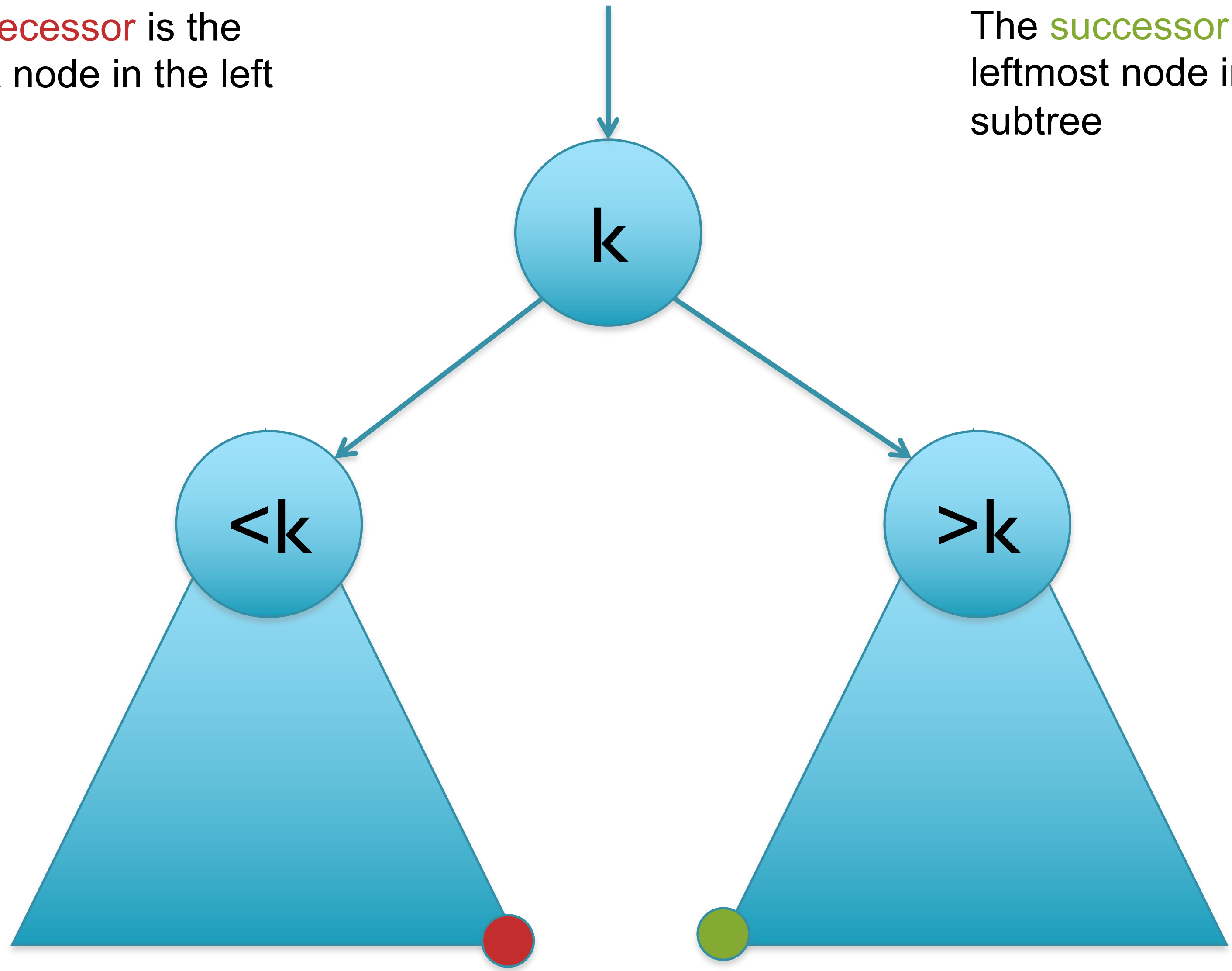
Binary Search Tree



Where is k’s immediate predecessor?
Where is k’s immediate successor?

Binary Search Tree

The predecessor is the rightmost node in the left subtree



The successor is the leftmost node in the right subtree

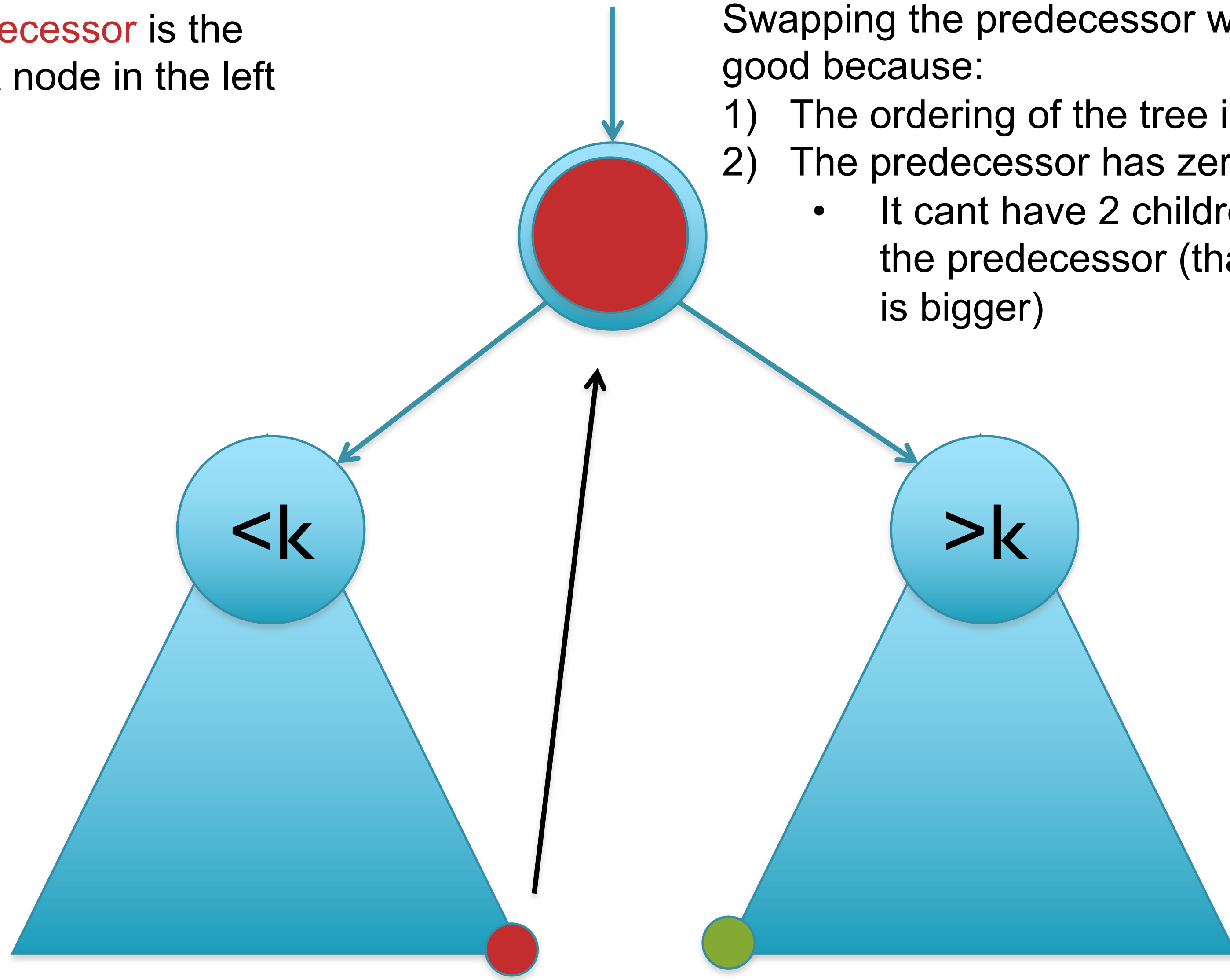
Where is k's immediate predecessor?
Where is k's immediate successor?

Binary Search Tree

The predecessor is the rightmost node in the left subtree

Swapping the predecessor with root is good because:

1) The ordering of the tree is preserved
2) The predecessor has zero or 1 child
    - It cant have 2 children or it is not the predecessor (that right child is bigger)



Where is k's immediate predecessor?
Where is k's immediate successor?

Removing

```
      6          7          6          8          6          6          4       or       9
    /   \       -->       /   \       -->       /   \       -->       /   \             /
   3     8               3     8               3     9               3     9           3
  / \   / \             / \     \             / \                   /                 / \
 1   4 7   9           1   4     9           1   4                 1                 1   4
```

Remove leaf is easy

Remove with only one child Is easy

Remove internal node ?

1. Find k's successor (or predecessor) and swap values with k
2. Remove the node we got that key from (easy, since it has at most one child)

BinarySearchTreeMap (I)

```
      6
     / \
    3   9
   / \
  1   4
```

```
import java.util.Iterator;

public class BinarySearchTreeMap<K extendsComparable<K>,V>
                     implements OrderedMap<K, V> {

    private static class Node<K, V> {
        Node<K, V> left, right;
        K key;
        V value ;
        Node(K k,V v){
            this.key = k;
            this.value =v;
        }
    }

    private Node<K, V> root;

    public boolean has(K k) {
        return this.find(k) != null;
    }
```

Static nested class
Sometimes convenient to use child[0] and child[1]

has() calls find()

BinarySearchTreeMap (2)

```
      6
     / \
    3   9
   / \
  1   4
```

```java
private Node<K, V> find(K k) {
    Node<K, V> n = this.root;
    while (n != null) {
        int cmp = k.compareTo(n.key);
        if (cmp < 0){
            n = n.left;
        } else if (cmp > 0){
            n = n.right;
        } else {
            return n;
        }
    }
    return null;
}

public void put(K k, V v) throws UnknownKeyException {
    Node<K, V> n = this.findForSure(k);
    n.value = v;
}

public V get (K k) throws UnknownKeyException {
    Node<K, V> n = this.findForSure(k);
    return n.value;
}
```

find() iteratively walks the tree, returns null if not found

put()/get() use a special findForSure() method

BinarySearchTreeMap (3)

```
      6
    /   \
   3     9
  / \
 1   4
```

```
private Node<K, V> findForSure(K k) throws UnknownKeyException {
    Node<K, V> n = this.find(k);
    if (n == null) {
        throw new UnknownKeyException();
    }
    return n;
}

public void insert (K k, V v) throws DuplicateKeyException{
    this.root = this.insert(this.root, k, v);
}

private Node<K, V> insert(Node<K, V> n, K k, V v) {
    if (n == null) {
        return new Node<K, V>(k, v);
    }
    int cmp = k.compareTo(n.key);
    if (cmp < 0){
        n.left = this.insert(n.left, k, v);
    } else if (cmp > 0){
        n.right = this.insert(n.right, k, v);
    } else {
        throw new DuplicateKeyException();
    }
    return n;
}
```

Just like find() but throws exception if not there

Recurse to right spot, add the new node, and return the modified tree after insert is complete

(n.left or n.right may be reset to same value for nodes that don't change)

BinarySearchTreeMap (4)

```
      6
     / \
    3   9
   / \
  1   4
```

```
public V remove(K k) throws UnknownKeyException {
    V value = this.get(k);
    this.root = this.remove(this.root, k);
    return value;
}

private Node<K, V> remove(Node<K, V> n, K k) throws UnknownKeyException {
    if (n == null) {
        throw new UnknownKeyException();
    }

    int cmp = k.compareTo(n.key);

    if (cmp < 0){
        n.left = this.remove(n.left , k);
    } else if (cmp > 0){
        n.right = this.remove(n.right, k);
    } else {
        n = this.remove(n);
    }

    return n;
}
```

First get() it so we can return the value, then actually remove

Recurse to right spot, then call the overloaded private remove() function

BinarySearchTreeMap (5)

```
      6
    /   \
   3     9
  / \
 1   4
```

```
private Node<K, V> remove(Node<K, V> n) {
   // 0 and 1 child
   if (n.left == null) {
        return n.right;
   }

   if (n.right == null) {
        return n.left;
   }

   // 2 children
   Node<K, V> max = this.max(n.left);
   n.left = this.removeMax(n.left);
   n.key = max.key;
   n.value = max.value;
   return n;
}

private Node<K, V> max(Node<K, V> n) {
   while (n.right != null) {
        n = n.right ;
   }
   return n;
}
```

Easy cases

Find the max of the subtree rooted on the left child -> its predecessor

Just keep walking right as far as you can

BinarySearchTreeMap (6)

```
      6
     / \
    3   9
   / \
  1   4
```

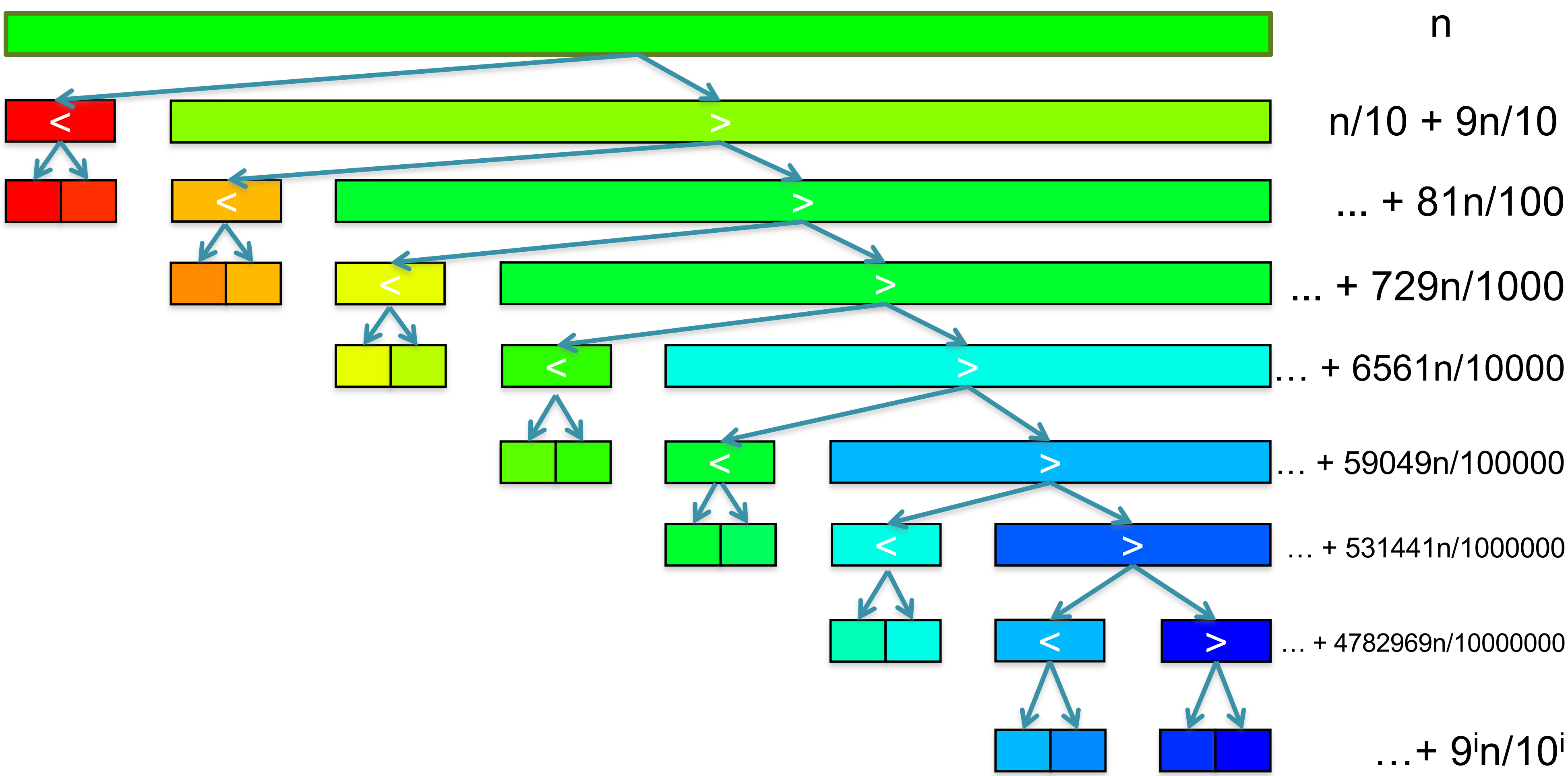
```java
  private Node<K, V> removeMax(Node<K, V> n) {
    if (n.right == null) {
        return n.left;
    }
    n.right = removeMax(n.right);
    return n;
  }

  public Iterator <K> iterator () {
    return null;
  }

  public String toString () {
    return this.toStringHelper(this.root);
  }

  private String toStringHelper (Node<K, V> n) {
    String s = "(";
    if (n != null) {
        s += this.toStringHelper(n.left);
        s += "" + n.key + ": " + n.value;
        s += this . toStringHelper (n.right);
    }
    return s + ")";
}
```

Fix the pointers to maintain BST invariant

Flush out rest of class: recursively traverse the tree to fill up an ArrayList<K> and return its iterator ☺

Binary Search

What if we miss the median and do a 90/10 split instead?



[How many times can we cut 10% off a list?]

Random Tree Height

Trying all permutations of 3 distinct keys

```
$ cat heights.3.log
tree[0]: 1 2 3 maxheight: 3
tree[1]: 1 3 2 maxheight: 3
tree[2]: 2 1 3 maxheight: 2
tree[3]: 2 3 1 maxheight: 2
tree[4]: 3 2 1 maxheight: 3
tree[5]: 3 1 2 maxheight: 3

numtrees: 6 average height: 2.66

maxheights[0]:    0        0.00%
maxheights[1]:    0        0.00%
maxheights[2]:    2        33.33%
maxheights[3]:    4        66.67%
```

Random Tree Height

```
$ cat heights.4.log
tree[0]: 1 2 3 4 maxheight: 4
tree[1]: 1 2 4 3 maxheight: 4
tree[2]: 1 3 2 4 maxheight: 3
tree[3]: 1 3 4 2 maxheight: 3
tree[4]: 1 4 3 2 maxheight: 4
tree[5]: 1 4 2 3 maxheight: 4
tree[6]: 2 1 3 4 maxheight: 3
tree[7]: 2 1 4 3 maxheight: 3
tree[8]: 2 3 1 4 maxheight: 3
tree[9]: 2 3 4 1 maxheight: 3
tree[10]: 2 4 3 1 maxheight: 3
tree[11]: 2 4 1 3 maxheight: 3
tree[12]: 3 2 1 4 maxheight: 3
tree[13]: 3 2 4 1 maxheight: 3
tree[14]: 3 1 2 4 maxheight: 3
tree[15]: 3 1 4 2 maxheight: 3
tree[16]: 3 4 1 2 maxheight: 3
tree[17]: 3 4 2 1 maxheight: 3
tree[18]: 4 2 3 1 maxheight: 3
tree[19]: 4 2 1 3 maxheight: 3
tree[20]: 4 3 2 1 maxheight: 4
tree[21]: 4 3 1 2 maxheight: 4
tree[22]: 4 1 3 2 maxheight: 4
tree[23]: 4 1 2 3 maxheight: 4
```

Trying all permutations of 4 distinct keys

numtrees: 24 average height: 3.33

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	16	66.67%
maxheights[4]:	8	33.33%

Random Tree Height

Trying all permutations of 5 distinct keys

numtrees: 120 average height: 3.80

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	40	33.33%
maxheights[4]:	64	53.33%
maxheights[5]:	16	13.33%

Random Tree Height

Trying all permutations of 10 distinct keys

numtrees: 3,628,800 average height: 5.64

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	253440	6.98%
maxheights[5]:	1508032	41.56%
maxheights[6]:	1277568	35.21%
maxheights[7]:	479040	13.20%
maxheights[8]:	99200	2.73%
maxheights[9]:	11008	0.30%
maxheights[10]:	512	0.01%

Random Tree Height

Trying all permutations of 11 distinct keys

numtrees: 39,916,800 average height: 5.91

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	1056000	2.65%
maxheights[5]:	13501312	33.82%
maxheights[6]:	15727232	39.40%
maxheights[7]:	7345536	18.40%
maxheights[8]:	1950080	4.89%
maxheights[9]:	308480	0.77%
maxheights[10]:	27136	0.07%
maxheights[11]:	1024	0.00%

Random Tree Height

Trying all permutations of 12 distinct keys

numtrees: 479,001,600 average height: 6.17

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	3801600	0.79%
maxheights[5]:	121362560	25.34%
maxheights[6]:	197163648	41.16%
maxheights[7]:	112255360	23.44%
maxheights[8]:	36141952	7.55%
maxheights[9]:	7293440	1.52%
maxheights[10]:	915456	0.19%
maxheights[11]:	65536	0.01%
maxheights[12]:	2048	0.00%

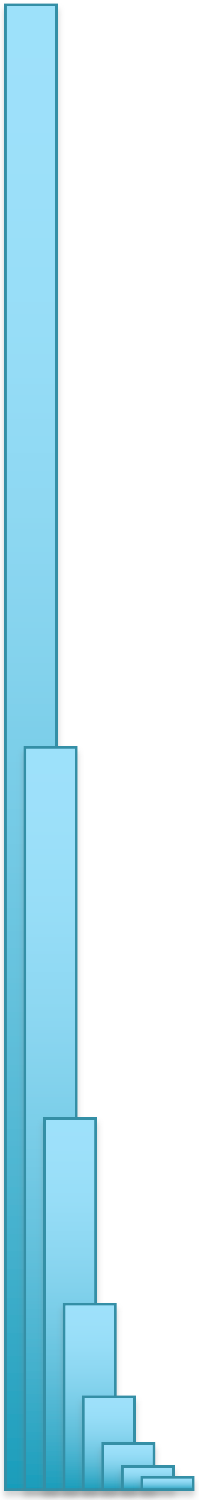
Random Tree Height

Trying all permutations of 13 distinct keys

numtrees: 6,227,020,800 average height: 6.40

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	10982400	0.18%
maxheights[5]:	1099169280	17.65%
maxheights[6]:	1764912384	28.34%
maxheights[7]:	1740445440	27.95%
maxheights[8]:	658214144	10.57%
maxheights[9]:	159805184	2.57%
maxheights[10]:	25572352	0.41%
maxheights[11]:	2617344	0.04%
maxheights[12]:	155648	0.00%
maxheights[13]:	4096	0.00%

Random Tree Height



Trying all permutations of 14 distinct keys

numtrees: 87,178,291,200 average height: 6.63

maxheights[0]:	0	0.00%
maxheights[1]:	0	0.00%
maxheights[2]:	0	0.00%
maxheights[3]:	0	0.00%
maxheights[4]:	21964800	0.03%
maxheights[5]:	10049994240	11.53%
maxheights[6]:	33305510656	38.20%
maxheights[7]:	27624399104	31.69%
maxheights[8]:	12037674752	13.81%
maxheights[9]:	3393895680	3.89%
maxheights[10]:	652050944	0.75%
maxheights[11]:	85170176	0.10%
maxheights[12]:	7258112	0.01%
maxheights[13]:	364544	0.00%
maxheights[14]:	8192	0.00%

Random Tree Height

Trying all permutations of 15 distinct keys

…
tree[44218000000]: 1 9 4 7 13 8 15 12 5 3 6 14 10 11 2 maxheight: 6
tree[44219000000]: 1 9 4 7 13 11 5 6 2 14 10 3 8 12 15 maxheight: 6
tree[44253000000]: 1 9 4 8 11 15 10 13 7 2 5 12 3 14 6 maxheight: 7
tree[44254000000]: 1 9 4 8 12 3 6 11 5 15 2 14 10 7 13 maxheight: 6
tree[44255000000]: 1 9 4 8 12 10 15 2 7 14 6 13 11 5 3 maxheight: 7
tree[44256000000]: 1 9 4 8 12 13 6 3 11 14 2 5 7 10 15 maxheight: 6
tree[44257000000]: 1 9 4 8 13 6 10 14 3 5 12 2 15 11 7 maxheight: 6
tree[44258000000]: 1 9 4 8 13 2 3 5 10 12 14 7 11 6 15 maxheight: 7
tree[44259000000]: 1 9 4 8 13 11 15 6 5 12 3 10 2 14 7 maxheight: 6
tree[44260000000]: 1 9 4 8 13 14 5 10 2 11 12 6 15 3 7 maxheight: 7
tree[44261000000]: 1 9 4 8 14 7 11 3 12 10 5 13 6 2 15 maxheight: 7
tree[44262000000]: 1 9 4 8 14 10 3 5 15 6 11 12 13 7 2 maxheight: 7
tree[44263000000]: 1 9 4 8 14 12 15 13 10 7 6 5 2 11 3 maxheight: 7
tree[44264000000]: 1 9 4 8 14 15 13 11 7 5 10 2 3 12 6 maxheight: 7
tree[44265000000]: 1 9 4 8 15 3 11 10 2 14 7 12 13 6 5 maxheight: 7
…

Part 2: AVL Trees

Balanced Trees

Note that we cannot require a BST to be perfectly balanced:

```
1      2    1        3       3      2       1       1
      /      \      /       /      / \       \       \
     1        2    2       1      1   3       3       2
                  /         \                /         \
                 1           2              2           3
```

Balanced | Not Balanced | Not Balanced | Not Balanced | Not Balanced | Balanced | Not Balanced | Not Balanced

Since neither tree with 2 keys is perfectly balanced we cannot insist on it

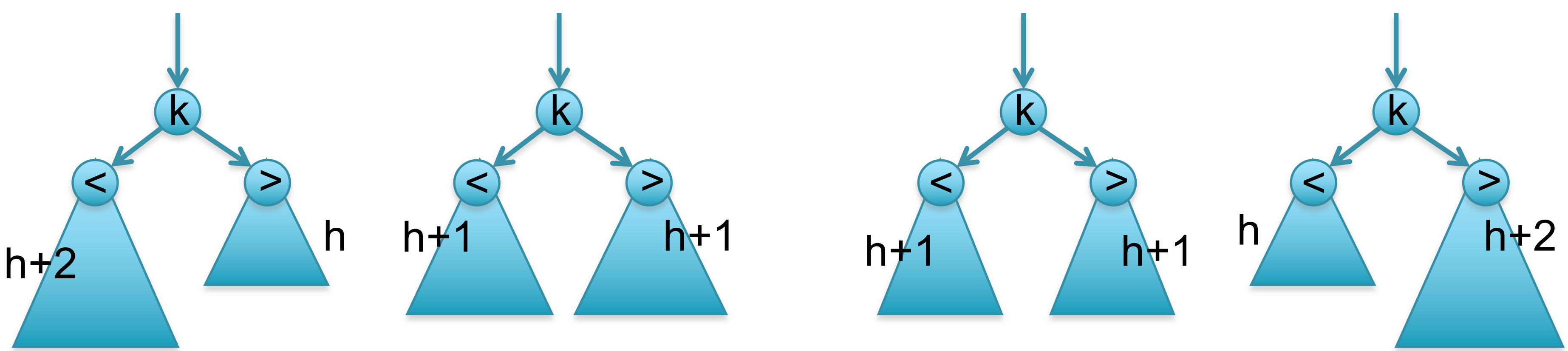
AVL Condition:

For every node n, the height of n's left and right subtree's differ by at most 1

Maintaining Balance

Assume that the tree starts in a slightly unbalanced state:

Inserting a new value can maintain the subtree heights or 1 of 4 possible outcomes:



Insert left, AVL Violation!

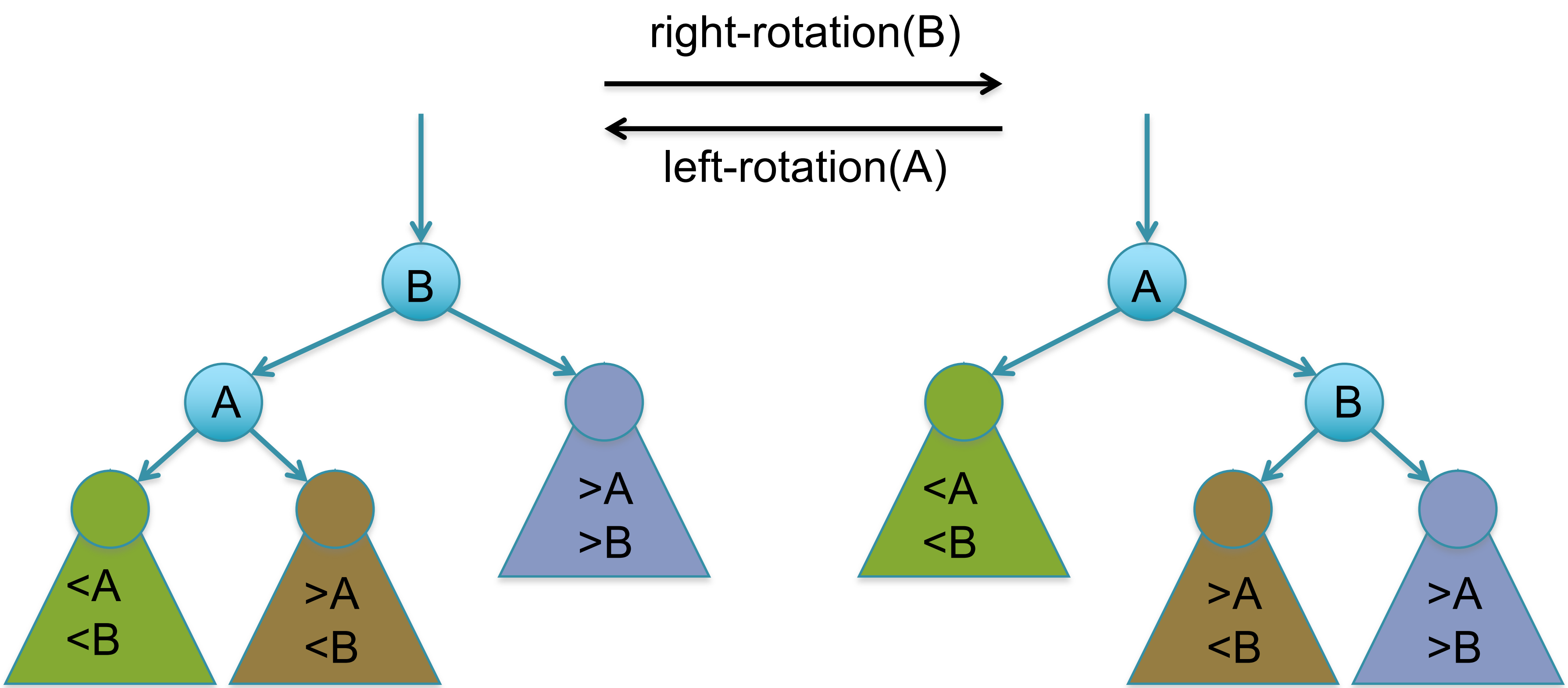
Insert right, Rebalanced

Insert left, Rebalanced

Insert right, AVL Violation!

Woohoo! AVL Violations ;-)

Tree Rotations



Note: Rotating a BST maintains the BST condition!

Green < A < Brown < B < Purple

Complete Example

Insert these values: 4 5 7 2 1 3

```
4
```

ins(5)

```
4
 \
  5
```

ins(7)

```
4
 \
  5
   \
    7
```

l(4)

```
  5
 / \
4   7
```

ins(2)

```
    5
   / \
  4   7
 /
2
```

ins(1)

```
      5
     / \
    4   7
   /
  2
 /
1
```

r(4)

```
    5
   / \
  2   7
 / \
1   4
```

ins(3)

```
    5
   / \
  2   7
 / \
1   4
   /
  3
```

l(2)

```
      5
     / \
    4   7
   /
  2
 / \
1   3
```

r(5)

```
    4
   / \
  2   5
 / \   \
1   3   7
```

Note:
AVL Violations are fixed bottom up

Removing

```
      6
     / \
    3   8
   / \ / \
  1  4 7  9
```

$\xrightarrow{7}$

```
      6
     / \
    3   8
   / \   \
  1  4    9
```

$\xrightarrow{8}$

```
      6
     / \
    3   9
   / \
  1  4
```

$\xrightarrow{6}$

```
      4
     / \
    3   9
   /
  1
```

or

```
      9
     /
    3
   / \
  1  4
```

Remove
leaf
is easy

Remove
with only
one child
Is easy

Remove
Internal Node:
Swap with
predecessor
/successor

Remove from AVL just like removing from regular BST:

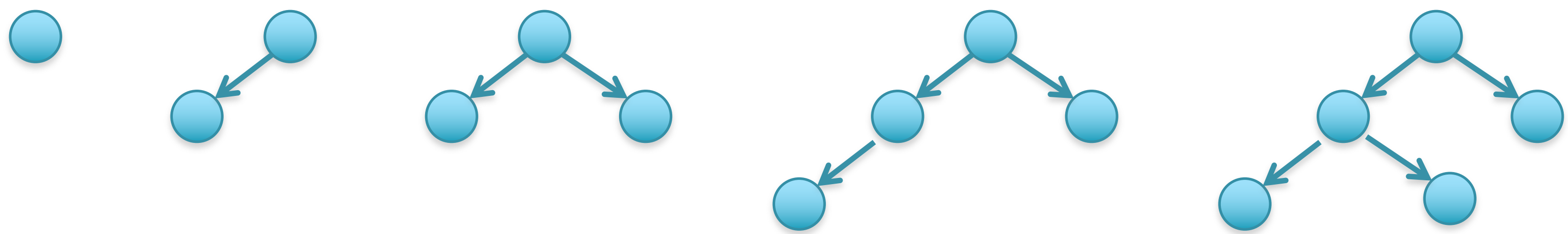
    Find successor
    Swap with that element,
    Remove the node that you just swapped.

Make sure to update the height fields, and rebalance if necessary

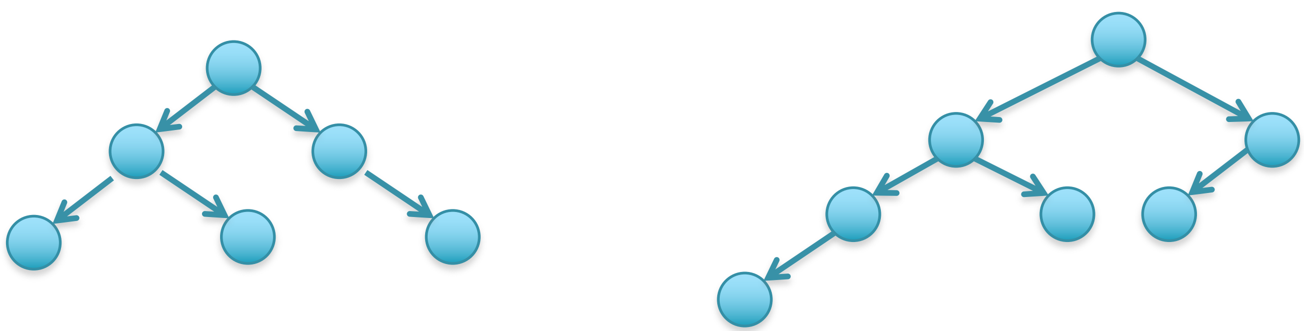
AVL Tree Balance

By construction, an AVL tree can never become "too unbalanced"

- AVL condition ensures left and right children differ by at most 1
- But they arent necessarily "full"



n=1 n=2 n=3 n=4 n=5

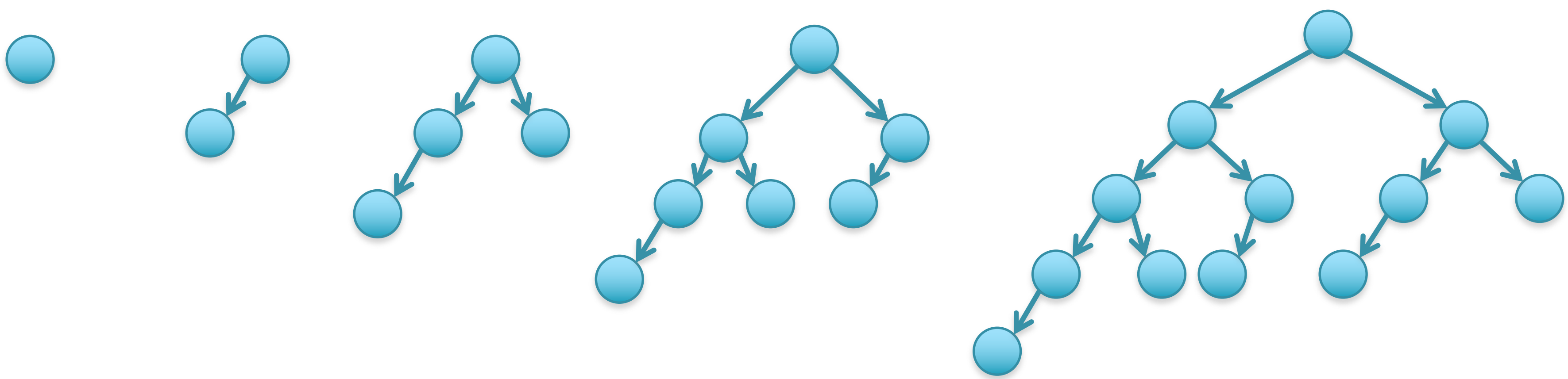


n=6 n=7

Sparse AVL Trees

How many nodes are in the sparsest AVL tree of height h

- Sparse means fewest nodes with height h
- Does it still include an exponential number of nodes?

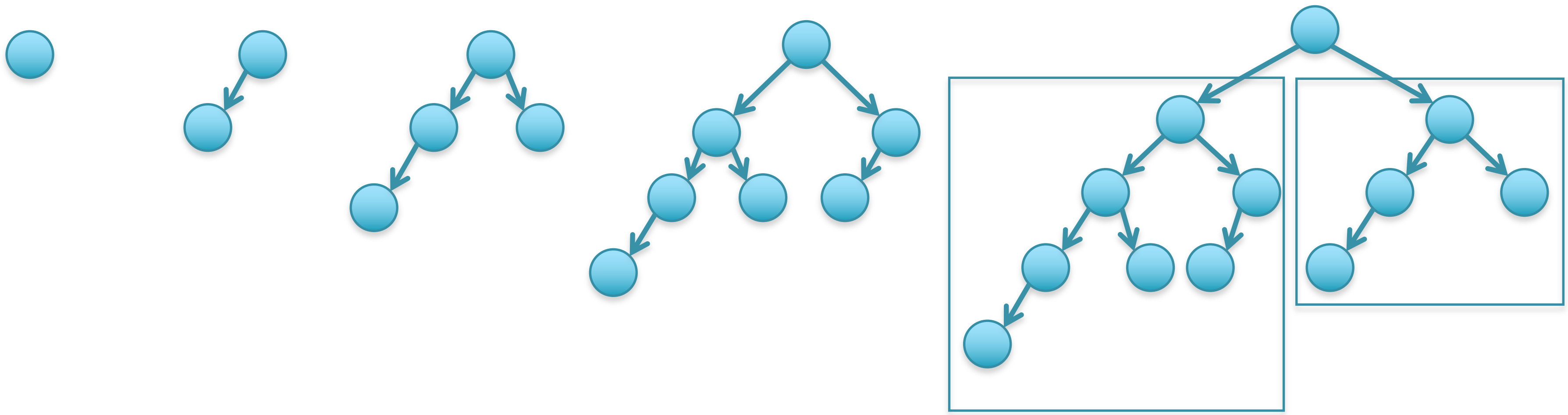


S_1	S_2	S_3	S_4	S_5
h=1 n=1	h=2 n=2	h=3 n=4	h=4 n=7	h=5 n=12

Sparse AVL Trees

How many nodes are in the sparsest AVL tree of height h

- Sparse means fewest nodes with height h
- Does it still include an exponential number of nodes?

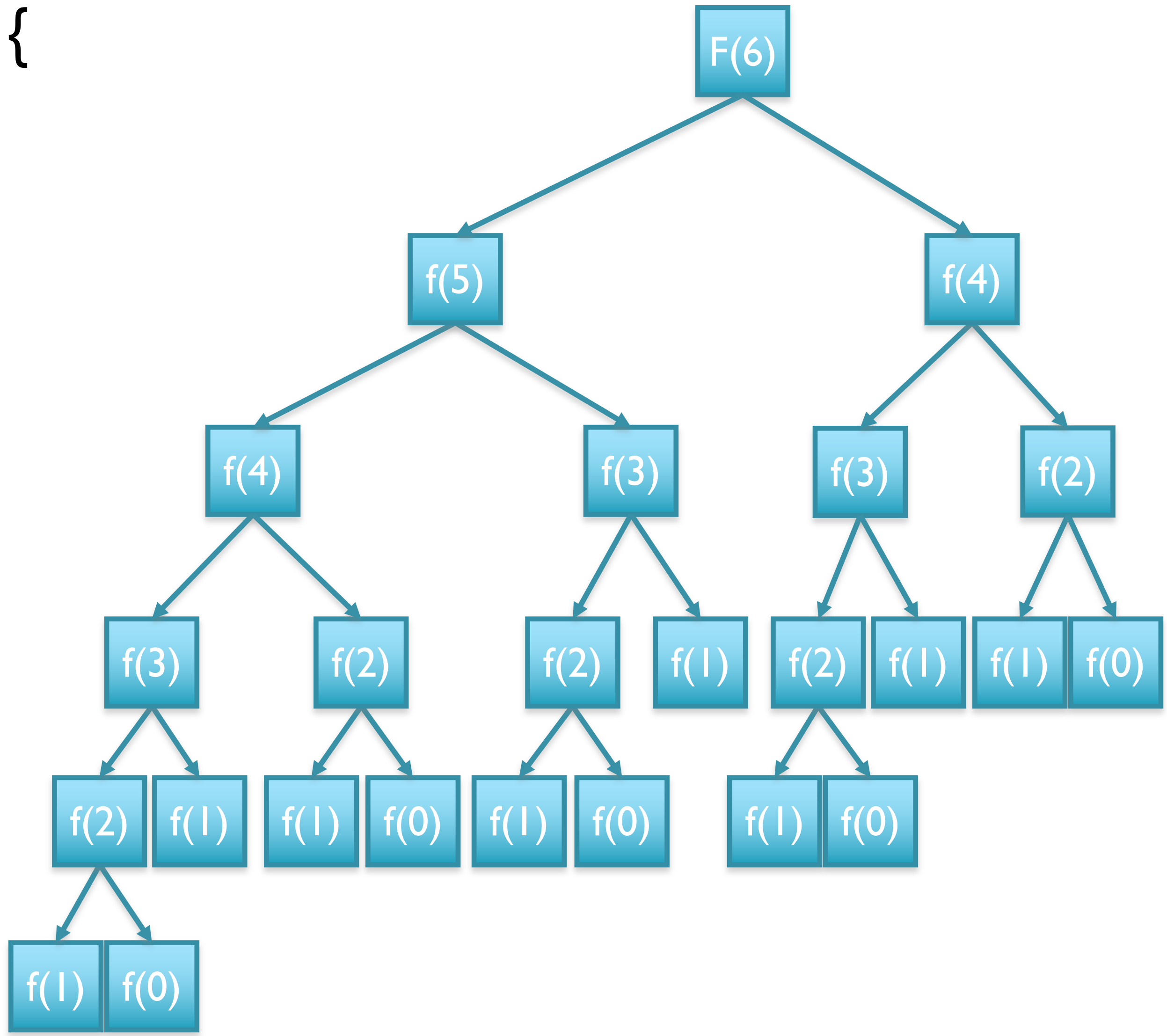


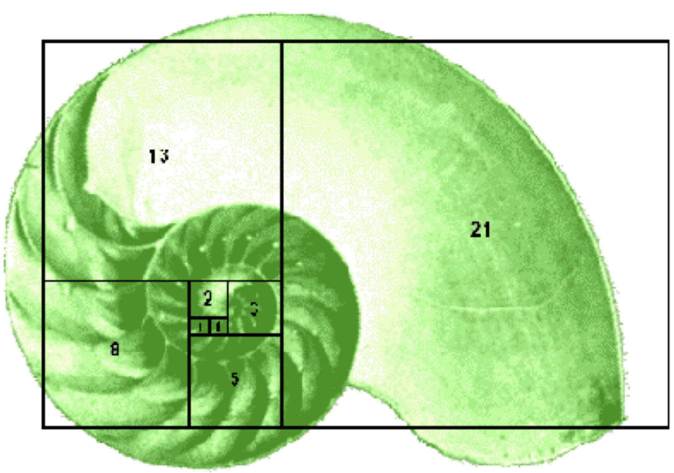
S_1	S_2	S_3	S_4	S_5
h=1 n=1	h=2 n=2	h=3 n=4	h=4 n=7	h=5 n=12
$S_1 = 1$	$S_2 = 2$	$S_3 = S_2 + S_1 + 1$	$S_4 = S_3 + S_2 + 1$	$S_5 = S_4 + S_3 + 1$

$$S_h = S_{h-1} + S_{h-2} + 1$$

Fibonacci Sequence

```
public static int fib(int n) {
   if (n <= 1) {
     return 1;
   }
   return fib(n-1) +
            fib(n-2);
}
```





Nodes in an AVL Tree

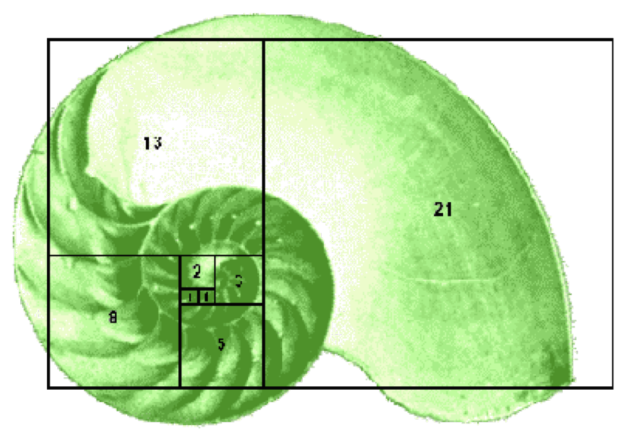
How many nodes are in the sparsest AVL tree of height h

- Sparse means fewest nodes with height h
- Does it still include an exponential number of nodes?

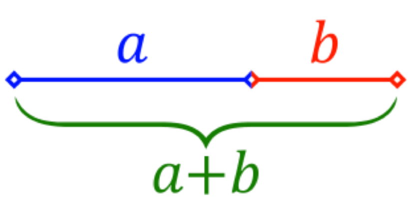
h	0	1	2	3	4	5	6	7	8
$S(h)$	1	2	4	7	12	20	33	54	88
$F(h+3)$	2	3	5	8	13	21	34	55	89

$$S(h) = F(h + 3) - 1$$

Fibonacci grows exponentially at φ^n
AVL Trees grow exponentially at φ^n-1



Therefore the height of ***any*** AVL tree is O(lg n)



$a+b$ is to a as a is to b

$$\varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$

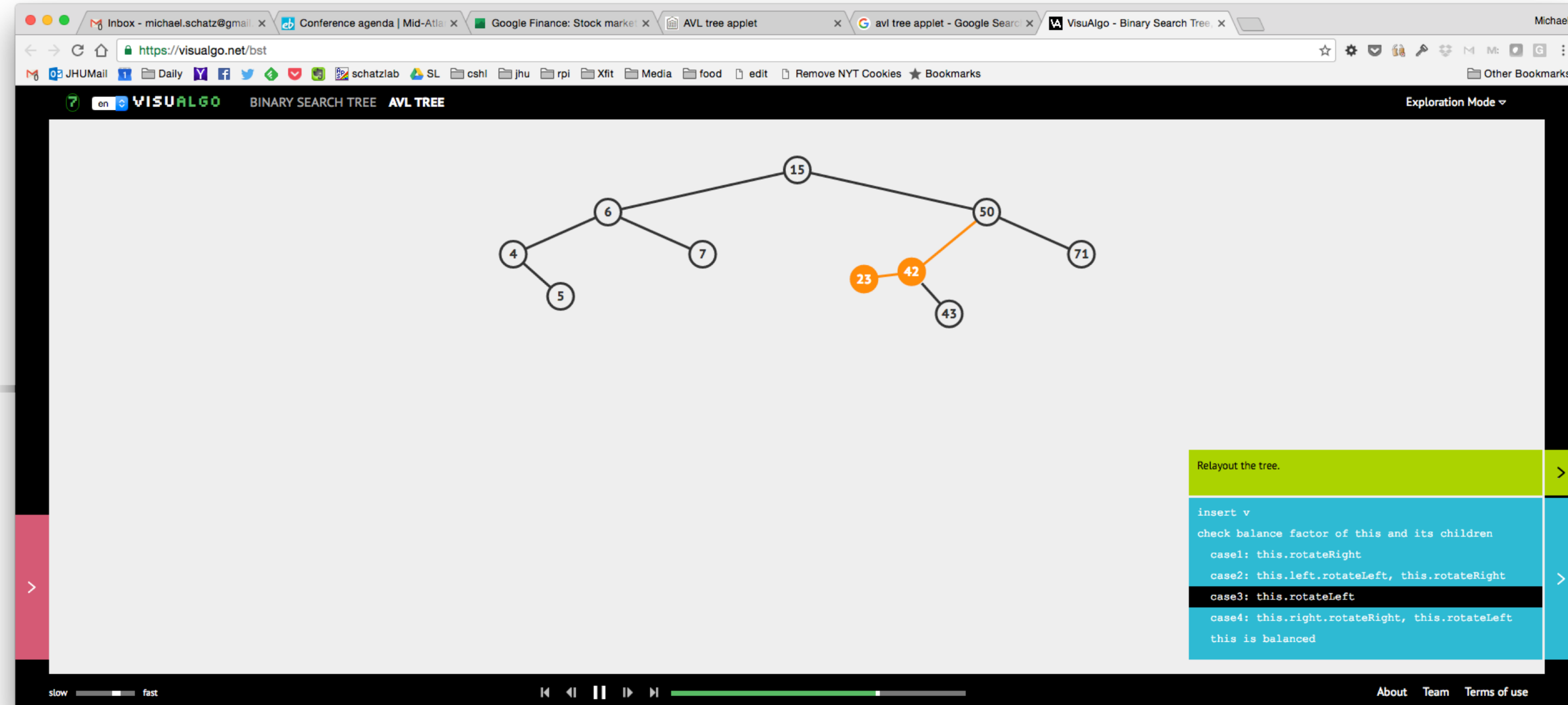
Implementation Notes

- ***Rotations can be applied in constant time!***
  - Inserting a node into an AVL tree requires O(lg n) time and guarantees O(lg(n)) height

- ***Track the height of each node as a separate field***
  - The alternative is to track when the tree is lopsided, but just as hard and more error prone
  - Don't recompute the heights from scratch, it is easy to compute but requires O(n) time!
  - Since we are guaranteeing the tree will have height lg(n), just use an integer
  - Only update the affected nodes

Check out Appendix B for some very useful tips on hacking AVL trees!

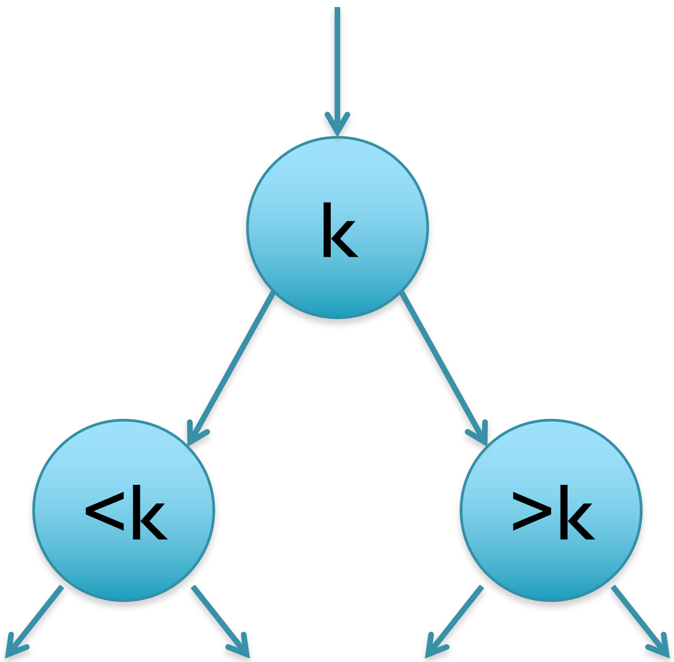
Sample Application



https://visualgo.net/bst

Part 3: Treaps

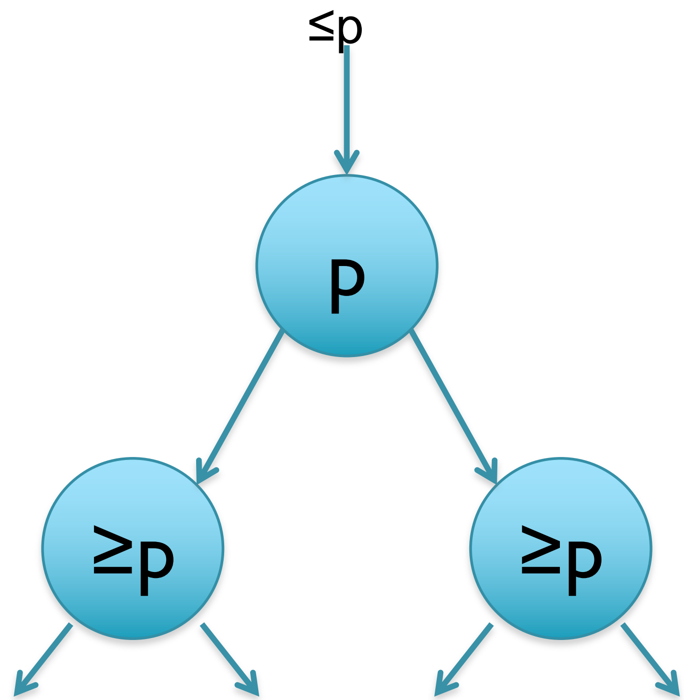
BSTs versus Heaps



BST

All keys in left subtree of k are < k, all keys in right are >k

Tricky to balance, but fast to find

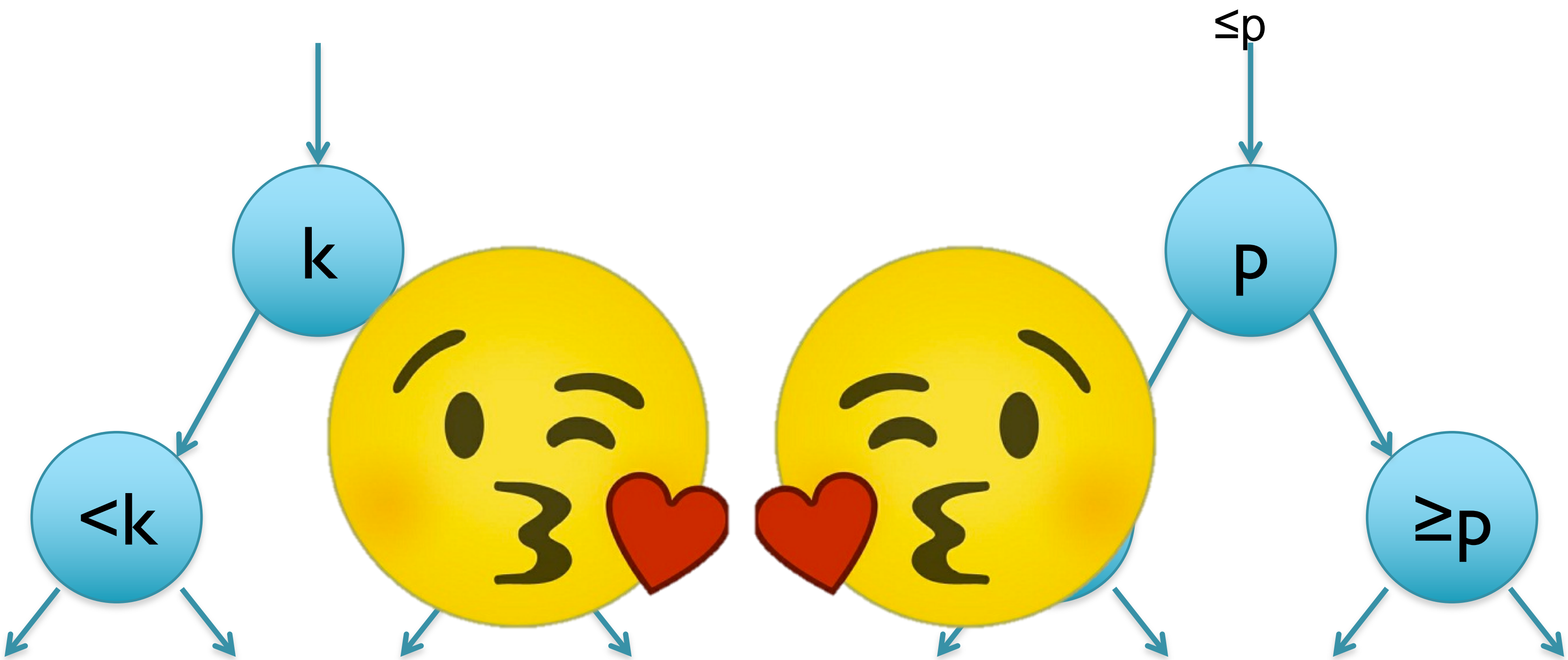


Heap

All children of the node with priority p have priority ≥p

Easy to balance, but hard to find (except min/max)

BSTs versus Heaps



BST

All keys in left subtree of k are < k, all keys in right are >k

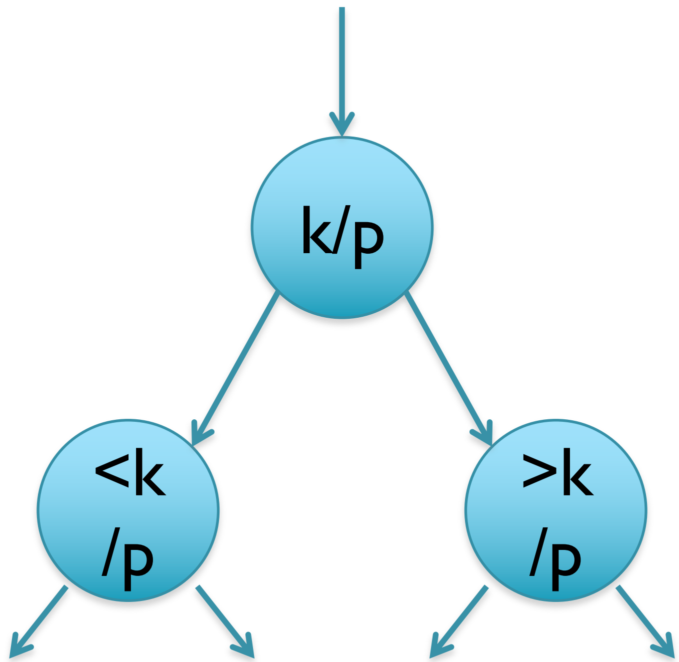
Tricky to balance, but fast to find

Heap

All children of the node with priority p have priority ≥p

Easy to balance, but hard to find (except min/max)

Treap



A treap is a binary search tree where the nodes have both user specified keys (k) and internally assigned priorities (p).

When inserting, use standard BST insertion algorithm, but then use rotations to iteratively move up the node while it has lower priority than its parent (analogous to a heap, but with rotations)

A (boring) example

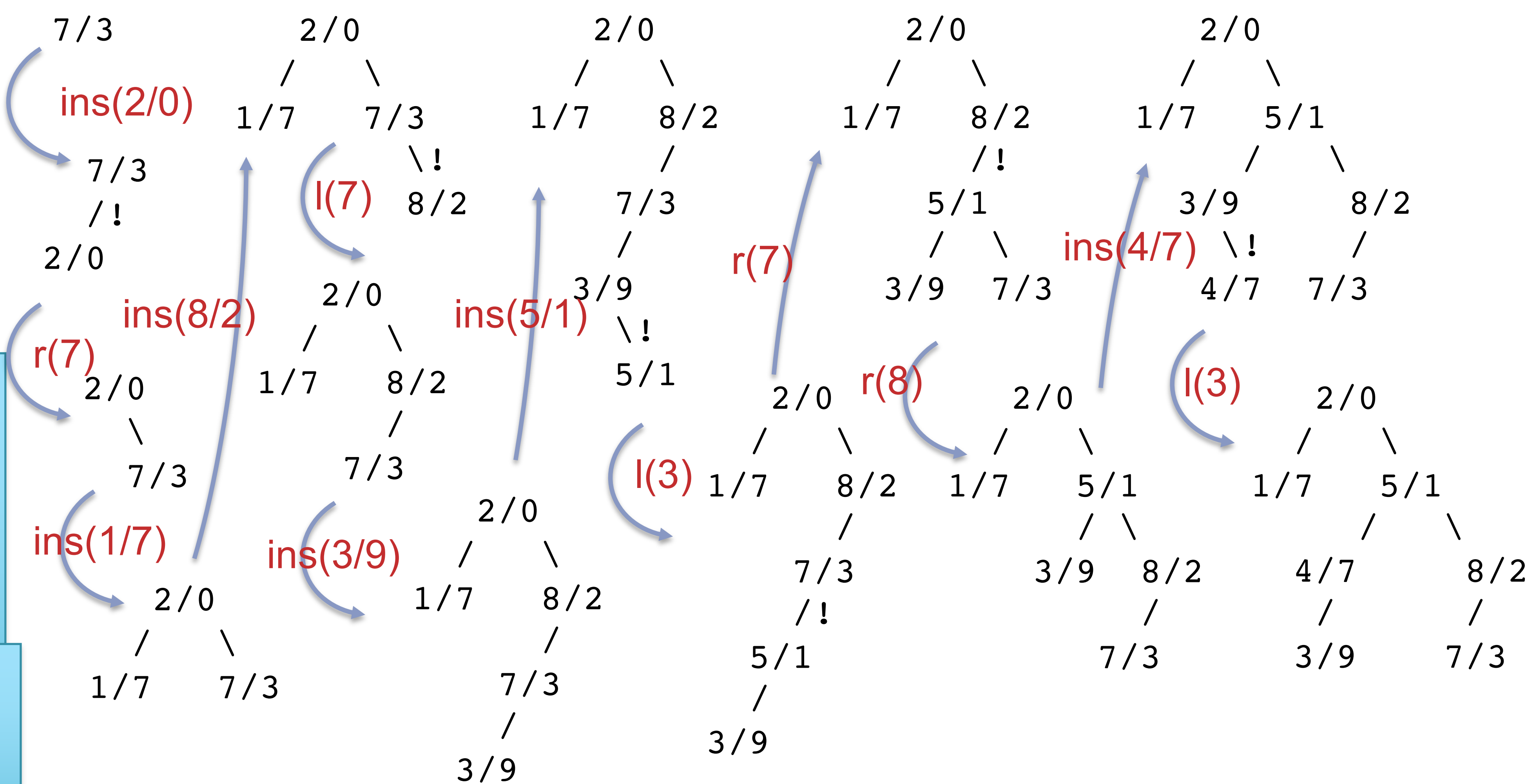
Insert the following pairs: 7/1, 2/2, 1/3, 8/4, 3/5, 5/6, 4/7

```
7/1  7/1   7/1   7/1        7/1         7/1         7/1
    /     /     /   \      /   \      /   \        /   \
  2/2   2/2   2/2   8/4  2/2   8/4   2/2   8/4   2/2   8/4
        /     /          / \         / \         / \
      1/3   1/3         1/3 3/5    1/3 3/5     1/3 3/5
                                         \             \
                                         5/6           5/6
                                                       /
                                                     4/7
```

The priorities were always increasing, so we never had to apply any of the rotations…. booooooring and unbalanced

A (better) example

Insert the following pairs: 7/3, 2/0, 1/7, 8/2, 3/9, 5/1, 4/7



Notice that we inserted the same keys, but with different priorities

Just by changing the priorities, we can improve the balance!

Treap Reflections

Insert the following pairs: 7/1, 2/2, 1/3, 8/4, 3/5, 5/6, 4/7

Since the priorities are in sorted order, becomes a standard BST and may have O(n) height

```
        7/1
       /   \
    2/2     8/4
   /   \
1/3    3/5
          \
           5/6
          /
       4/7
```

Insert the following pairs: 7/3, 2/0, 1/7, 8/2, 3/9, 5/1, 4/7

With a standard BST, for 2 to be root it would have to be the first key inserted, and 5 would have to proceed all the other keys except 1, …

It is as if we saw the sequence:
2,5,8,7,1,4,3
Note priorities in sorted order:
2/0, 5/1, 8/2/, 7/3, 1/7, 4/7, 3/9

```
    2/0
   /   \
1/7     5/1
       /   \
    4/7     8/2
   /       /
3/9     7/3
```

What priorities should we assign to maintain a balanced tree?

Math.random()

Using random priorities essentially shuffles the input data
(which might have bad linear height)

into a ***random permutation***
that we ***expect*** to have O(log n) height
☺

It is possible that we could randomly shuffle into a poor tree configuration, but that is extremely rare.

In most practical applications, a treap will perform just fine, and will often outperform an AVL tree that guarantees O(log n) height but has higher constants

Self Balancing Trees

Understanding the distinction between different kinds of balanced search trees:

- ***AVL trees guarantee worst case O(log n)*** operations by carefully accounting for the tree height
- ***treaps guarantee expected O(log n)*** operations by selecting a random permutation of the input data
- ***splay trees guarantee amortized O(log n)*** operations by periodically applying a certain set of rotations (see lecture notes)

If you have to play it safe and don't trust your random numbers,
=> AVL trees are the way to go.
If you can live with the occasional O(n) op
=> splay trees are the way to go.
And if you trust your random numbers
=> treaps are the way to go.

Next Steps

1. Work on HW7
2. Check on Piazza for tips & corrections!